Object Oriented Programming Using Java

(c) Amity University Online

AMITY UNIVERSITY | DIRECTORATE OF DISTANCE & ONLINE EDUCATION

© Amity University Press

All Rights Reserved

No parts of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission of the publisher.

Advisory Committee

Chairman : Ms. Monica Agarwal

Members : Prof. Arun Bisaria

Dr. Priya Mary Mathew

Prof. Aindril De

Mr. Alok Awtans

Dr. Coral J Barboza

Dr. Monica Rose

Mr. Sachit Paliwal

SLM Review Committee

Mr. Gaurav Agarwal

Ms. Nitika Khanna

Ms. Rashmi Saxena

Ms. Renu Singh

Ms. Mona Chaudhary

Published by Amity University Press for exclusive use of Amity Directorate of Distance and Online Education, Amity University, Noida-201313

(c) Amity University Online

Contents

Page No.

Module – 1 : OOP and Introduction to Java 01

1.1 Introduction to Object Oriented Programming

1.1.1 Fundamentals of object oriented programming:Procedure Oriented Programming Vs. Object Oriented Programming (OOP)

1.1.2 Concept of Classes and Object

1.1.3 Object reference, Abstraction,Encapsulation

1.1.4 Inheritance, Polymorphism

1.2 Introduction to Java

1.2.1 History of Java, Java vs C++.

1.2.2 JDK, JVM and Other tools for Java Application

1.2.3 IDE for developing Java Application

1.2.4 “Constructs used in JAVA :

1.2.5 Variables, types and type declarations”

1.2.6 Data types, Functions

1.3 Statements & Operators in Java

1.3.1 Increment and Decrement Operators

1.3.2 Relational and Logical operators

1.3.3 Conditional Expressions, if then else clause

1.3.4 Input using scanner class and output statement

1.3.5 Loops,Switch case

1.3.6 Arrays, Methods

1.4 Inheritance in Java

1.4.1 Type of Inherirtance with code snippets

1.4.2 Concept of method overriding with code snippet,, Final classes

1.4.3 Concept of Abstract classes with code snippet

1.4.4 Concept of Final classes with code snippet

Module - 2 : Package and Exception in Java 206

2.1 Introduction to Packages

2.1.1 Packages : Defining Package, CLASSPATH

2.1.2 Type of Packages

2.1.3 Importing Packages

2.1.4 Access Specifiers

2.1.5 Interfaces: Implementing Interfaces

2.1.6 Interfaces vs Abstract Class

2.2 Exception Handing

2.2.1 Introduction to Exceptions

2.2.2 Exception Handling by using Try-Catch

2.2.3 Catching Multiple Exceptions

(c) Amity University Online

2.2.4 Types of Exceptions
2.2.5 final vs finally clause ,throw vs throws
2.3 Throwing Exceptions
2.3.1 Implementation of finally, throw & throws
2.3.2 Fundamentals of throwing exceptions
2.3.3 Java's buit-in Exceptions
2.3.4 Writing Exception Subclasses
2.3.5 Implementation of Exception Handling Mechanism.

Module - 3: Package and Exception in Java 171

3.1 Introduction to Packages
3.1.1 Constructors
3.1.2 Object & Object Reference
3.1.3 Types of Constructors
3.1.4 Implementation of Constructors in Inheritance
3.1.5 order of invocation of constructors
3.2 Wrapper classes and String class
3.2.1 Introduction to Wrapper Classes
3.2.2 Strings and Characters : Fundamentals of Characters and Strings
3.2.3 The String Class , String Operations
3.2.4 Data Conversion using Value Of ( ) Methods
3.3 StringBuffer class
3.3.1 Fundamental Concept of StringBUffer Class
3.3.2 Properties of StringBuffer Class
3.3.3 Methods of StringBuffer class
3.3.4 Code snippet showing the difference between String and StringBuffer Class

Module - 4 : Interface and Threads in Java 241

4.1 Introduction to Interface
4.1.1 Introduction to Interface
4.1.2 Code snippet showing implementation of Interface
4.1.3 Reference variable in interface
4.1.4 Extending Interface
4.2 Interfcae Methods
4.2.1 Type of Methods in interface
4.2.2 Default method in interface
4.2.3 Static methods in interface
4.2.4 Constant variables in interface
4.3 Multithreading
4.3.1 Introduction to Multithreading
4.3.2 Concept of Thread in Multithreading
4.3.3 Lifecycle of Thread
4.3.4 Implementing a Thread

(c) Amity University Online

4.3.5 Thread vs Runnable method of creating thread
4.3.6 Methods of Thread class
4.3.7 Thread Priorities
4.4 Thread Synchronization
4.4.1 Introduction to Synchronization
4.4.2 Implementing Synchronization in Thread
4.4.3 Benefits and Loopholes of Synchronization
4.4.4 Resuming and stopping Threads
4.4.5 A code snippet to demonstrate thread with and without synchronization

Module - 5 : Applet and Graphics Programming **320**

5.1 Introduction to Applet Class
5.1.1 Introduction to applets
5.1.2 Lifecycle of Applet
5.1.3 Methods of Applet class
5.1.4 Applet vs Application
5.1.5 Embedding Applet in HTML Page
5.1.6 A code snippet to implement Applet
5.2 Introduction to Graphic class
5.2.1 Introduction to Graphic class
5.2.2 Methods in Graphic class
5.2.3 A code snippet to create different shapes.
5.2.4 Using loops in applet
5.3 Introduction to AWT
5.3.1 Introduction to AWT package
5.3.2 Classes in AWT package
5.3.3 Role of Layout manager in applet
5.3.4 A code snippet to implement applet with layout manager

(c) Amity University Online

(c) Amity University Online

Module – 1 : OOP and Introduction to Java

Structure:

1.1 Introduction to Object Oriented Programming

1.1.1 Fundamentals of object oriented programming:Procedure Oriented Programming Vs. Object Oriented Programming (OOP)

1.1.2 Concept of Classes and Object

1.1.3 Object reference, Abstraction,Encapsulation

1.1.4 Inheritance, Polymorphism

1.2 Introduction to Java

1.2.1 History of Java, Java vs C++.

1.2.2 JDK, JVM and Other tools for Java Application

1.2.3 IDE for developing Java Application

1.2.4 “Constructs used in JAVA :

1.2.5 Variables, types and type declarations”

1.2.6 Data types, Functions

1.3 Statements & Operators in Java

1.3.1 Increment and Decrement Operators

1.3.2 Relational and Logical operators

1.3.3 Conditional Expressions, if then else clause

1.3.4 Input using scanner class and output statement

1.3.5 Loops,Switch case

1.3.6 Arrays, Methods

1.4 Inheritance in Java

1.4.1 Type of Inherirtance with code snippets

1.4.2 Concept of method overriding with code snippet,, Final classes

1.4.3 Concept of Abstract classes with code snippet

1.4.4 Concept of Final classes with code snippet

(c) Amity University Online

Notes

Unit - 1.1 : Introduction to Object Oriented Programming

Objectives

At the end of this unit, you will be able to understand:

- Basics of Object Oriented programming Difference between OOP and procedure oriented programming
- Classes and object and their concept
- Understanding concept of Object reference, Abstraction, Encapsulation
- Understanding concept of Inheritance, Polymorphism

Introduction

In order to unravel a problem, you would like to understand the “method/procedure” otherwise you got to have the “know-how”. In computer parlance, we call this an algorithm. An algorithm or a program may be a sequence of steps to be followed, which results in a desired output. The sequence of steps is often called a procedure, and, in turn, a gaggle of procedures are often termed as a program. For solving an easy problem, a sequence of steps is sufficient, but if the matter is complex, a programming environment that has well-integrated and cohesive programming elements, constructs and data structures is required.

A paradigm is often thought of as a method of programming, which involves elements like functions, data and data structures. Several programming paradigms are used successfully. The structured programming paradigm with C as implementing language, during which the programmer breaks down the task to be accomplished into subtasks and specifies a step-by-step procedure or algorithm to realize this task, has been very successful and popular amongst programmers, owing largely to its reusable components within the sort of procedural calls and its ability to control memory and thus handle hardware integration.

The object-oriented programming may be a different approach to programming. It has been created with a view to extend programmer’s productivity by overcoming the weaknesses found in procedural programming approach. Over the years many object-oriented programming languages like C++, Java have come up and are getting quite popular within the market. The main need for developing such languages was to manage the ever-increasing size and complexity of programs.

Object-oriented programming (OOP) promises wide reusability features through inheritance, library functions in the structure of standard template library (STL), etc. C++ is one of the most powerful OOP languages that supports together structured as well as OOP paradigms.

(c) Amity University Online

Notes

Prior to the invention of C and C++, programming languages could be broadly categorized as follows:

High-level languages such as FORTRAN, COBOL and Pascal, catered to programmers with friendly features such as English-like coding languages. In these languages, the code written by programmers is compiled and converted into an object code. The implementation and hardware details such as addresses are hidden from the users. Hence users can code the application software for scientific and business communities. COBOL is a language widely used in the business world. FORTRAN and Pascal, on the other hand, are languages that are used by the scientific communities.

Low-level languages such as assemblers require extensive knowledge of hardware, addressing mechanisms and operating systems. Accordingly, these types of languages are used by experts, academicians and system developers. Most of the interfaces involving hardware integration with software are written using these languages.

Middle-level languages such as C. These languages combine the strengths of both high-level and low-level languages.

- Code is almost English like. It is easy to understand and write programs in this language.
- It has the ability to handle hardware as many of its commands support the direct handling of underlying hardware such as memory and hardware devices.
- Both business programming and scientific calculations can be easily handled.

1.1.1 Fundamentals of object oriented programming: Procedure Oriented Programming Vs. Object Oriented Programming (OOP)

Building Blocks of Programs

There are two basic aspects of programming: data and instructions. To figure with data, you would like to know variables and types; to figure with instructions, you would like to know control structures and subroutines. You'll spend an outsized a part of the course becoming conversant in these concepts

A variable is simply a memory location (or several locations treated as a unit) that has been given a reputation in order that it are often easily mentioned and utilized in a program. The programmer only has got to worry about the name; it's the compiler's responsibility to stay track of the memory location. The programmer does got to confine mind that the name refers to a sort of "box" in memory which will hold data, albeit the programmer doesn't need to know where in memory that box is found.

In Java and most other languages, a variable features a type that indicates what kind of data it can hold. One sort of variable might hold integer numbers like 5, -14, and 0— while another holds floating point numbers with decimal points like 2.14, -2.7, or 17.0. There could even be types for individual characters ('Z', ';', etc.), strings ("Hi", "A string can include many characters", etc.), and fewer common types like dates, time, sounds, or the other sort of data that a program might got to store.

Programming languages always have commands for getting data into and out of variables and for doing computations with data. for instance , the subsequent "assignment statement," which could appear during a Java program, tells the pc to require the amount stored within the variable named "principal", multiply that number by 0.012, then store the end in the variable named "interest":

```
Interest=Principal*0.012;
```

(c) Amity University Online

Notes

There also are “input commands” for getting data from the user or from files on the computer’s disks and “output commands” for sending data within the other direction.

These basic commands—for moving data from place to put and for performing computations—are the building blocks for all programs. These building blocks are combined into complex programs using control structures and subroutines.

Objects and Object-oriented Programming

Programs must be designed. Nobody can just sit down at the pc and compose a program of any complexity. The discipline called software engineering cares with the development of correct, working, well-written programs. The programmer tends to use accepted and proven methods for analyzing the matter to be solved and for designing a program to unravel that problem.

The procedural approach to programming was the de facto approach within the youth of programming. Here, code is modularized supported a system’s processes. as an example , in developing a Hotel application system, we might have considered processes like the checking-in and Check-out of rooms, making reservations of rooms, cataloging of hotels room, and so on. Problem solving would involve the analysis of those processes in terms of the procedural tasks administered and therefore the production of a system whose representation is predicated on the procedural flow of the processes.

Object-oriented programming, on the opposite hand, models objects and therefore the interactions within the problem space and the production of a system supported these objects and their interactions.

Since the real-world problem domain is characterized by objects and their interactions, a software application developed using the object-oriented programming approach will end in the assembly of a computing system that features a closer representation of the real-world problem domain than would be the case if the procedural programming approach is employed.

During the 1970s and into the 80s, the first software engineering methodology was structured programming. The structured programming approach to program design was supported the subsequent advice: to unravel an outsized problem, break the matter into several pieces and work on each bit separately; to unravel each bit, treat it as a replacement problem which may itself be weakened into smaller problems; eventually, you’ll work your way right down to problems which will be solved directly, without further decomposition. This approach is named top-down programming.

There is nothing wrong with top-down programming. It’s a valuable and often-used approach to problem-solving. However, it’s incomplete. For one thing, it deals almost entirely with producing the instructions necessary to unravel aissue. But as time went on, people realized that the planning of the info structures for a program was as least as important because the design of subroutines and control structures. Top-down programming doesn’t give adequate consideration to the info that the program manipulates.

Another problem with strict top-down programming is that it makes it difficult to reuse work finished other projects.

(c) Amity University Online

Notes

So, in practice, top-down design is usually combined with bottom-up design. In bottom-up design, the approach is to start out "at rock bottom ," with problems that you simply already skills to unravel (and that you would possibly have already got a reusable software component at hand). From there, you'll work upwards towards an answer to the general problem

The reusable components should be as "modular" as possible. A module may be a component of a bigger system that interacts with the remainder of the system during a simple, well-defined, straightforward manner. The thought is that a module are often "plugged into" a system. The small print of what goes on inside the module aren't important to the system as an entire, as long because the module fulfills its assigned role correctly. This is often called information hiding, and it's one among the foremost important principles of software engineering.

Modules that would support this type of information-hiding became common in programming languages within the early 1980s. Since then, a more advanced sort of an equivalent idea has more or less appropriated software engineering. This latest approach is named object-oriented programming, often abbreviated as OOP.

The central concept of object-oriented programming is that the object, which may be a quite module containing data and subroutines. The point-of-view in OOP is that an object may be a quite independent entity that has an indoor state (the data it contains) which can answer messages (calls to its subroutines). A list object, for instance, features a state consisting of an inventory of names and numbers. If you send it a message telling it to feature a reputation, it'll respond by modifying its state to reflect the change. If you send it a message telling it to print itself, it'll respond by printing out its list of names and numbers.

Objects and Their Interactions in the Real World

Let us consider a real-world situation. There are two persons, Ashok and his wife, Naina. They are customers of Pepperfry, a company dealing in luxurious furniture. Pepperfry sells a variety of Double bed sets. Each Double bed set is labeled with an identification number and a price tag. After viewing the Double bed sets for an hour, Ashok and Naina decide to purchase a White leather Double bed set. They approach Mukesh, a salesperson, to place their order.

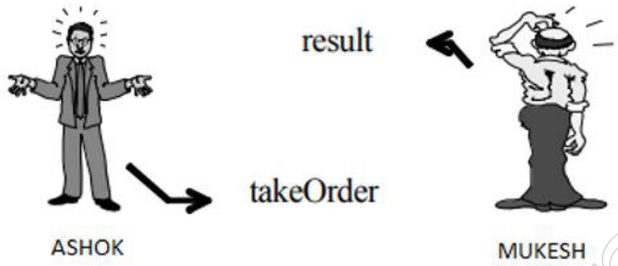
In making his request known to Ashok, Ashok sends a message to Mukesh, "I would like to purchase this White leather, Double bed set. Can you please have it sent to me by next Friday?"

The message that Ashok has sent to Mukesh is a takeOrder message. It contains information such as the type of sofa set (a White leather, Double bed set) and the date of delivery (next Friday). This information is known as the parameters of the takeOrder message.

In response to Ashok's message, Mukesh replies to Ashok by returning the result of his request. We can represent the interaction between Ashok and Mukesh graphically using Figure below. takeOrder result Ashok Mukesh Figure below:

(c) Amity University Online

Notes



Interaction between Ashok and Mukesh. Mukesh was able to respond to Ashok's takeOrder message because he understood it and had the means to handle Ashok's request. Although Mukesh knew how to satisfy Ashok's request, Ashok did not. In fact, most of the time, customers do not know how a salesperson has satisfied their orders. All they get from salespersons are replies such as, "I am sorry, madam, we are unable to satisfy your request because the sofa you wanted has been sold," or "Sir, your request has been satisfied. We will deliver the goods on Friday between 10 am to 11 am to the address indicated. Thank you for your order."

Mukesh, as a salesperson at Pepperfry, has a responsibility towards Ashok. He maintains his responsibility by applying a set of operations:

1. He determines if there is sufficient stock to satisfy Ashok's request.
2. He determines if the requested date for delivery is a suitable date.
3. He instructs the warehouse staff to deliver the goods to Ashok's address on the requested date, if the above conditions are satisfied.
4. Finally, he informs Ashok the result of his request.

Objects and Their Interactions in Programming

The interactions between Ashok and Mukesh in the above real-world situation can be represented in object-oriented programming terms. For instance, Ashok and Mukesh are objects that interact by sending messages. Ashok is thus a message-sending object, while Mukesh is a message-receiving object. Alternatively, we can label Ashok as a sender and Mukesh as a receiver.

The takeOrder request from Ashok to Mukesh is an example of a message. It may have additional, accompanying information known as parameters (or arguments) of the message. The fact that Mukesh responded to Ashok's message indicates that the message is a valid message. Each valid message corresponds to a method that Mukesh uses to fulfill his responsibility to Ashok.

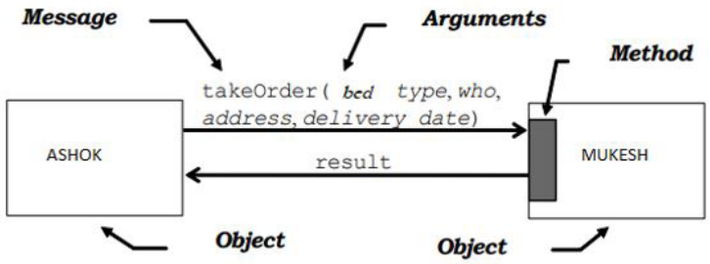
An invalid message, on the other hand, is one that the receiver does not have the capability to respond to, that is, the receiver does not have a corresponding method to match the message. For example, if Ashok had requested a discount on the price, his request would have been rejected because Mukesh, being a salesperson, would not have the capability (or a corresponding method) to respond to the message.

A method contains a number of operations detailing how Mukesh is to satisfy the demand Ashok put on Mukesh through the request.

Figure below summarizes the relationships of these terms:

(c) Amity University Online

Notes



Object interactions in object-oriented programming terms.

While Ashok may know what Mukesh can do through his methods, he may not know how Mukesh does them. This is an important principle of object-oriented programming known as information hiding: the sender of a message does not recognize how a receiver is going to satisfy the request in the message.

Procedural Programming:

"Procedural Programming is a, programming model which is a derivative from structured programming, based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a chain of computational steps to be carried out. During a program's execution, any given procedure might be called at some point, as well as by other procedures or itself. Procedural Programming languages examples are C, ALGOL, COBOL."

Object Oriented Programming:

Object oriented programming is a programming model which is based on the concept of objects. Objects contain data in the type of attributes and code in the type of methods. In object oriented programming, computer programs are planned using the concept of objects that cooperate with real world. Object oriented programming languages are a variety of but the most popular ones are class-based, significance that objects are instances of classes, which also determine their types. Object Oriented Programming languages examples are JAVA, C++, PYTHON.

Difference between Procedure Oriented Programming and Object Oriented Programming:

Procedure Oriented Programming	Object Oriented Programming
Program is divided into small parts called functions	Program is divided into parts called objects.
Importance is not given to data but to functions as well as sequence of actions to be done	Importance is given to the data rather than procedures or functions because it works as a real world.
follows Top Down approach	OOP follows Bottom Up approach.
It does not have any access specifier	OOP has access specifiers named Public, Private, Protected, etc
Data can move freely from function to function in the system	To add new data and function in POP is not so easy

(c) Amity University Online

Notes

Most function uses Global data for sharing that can be accessed freely from function to function in the system	objects can move and communicate with each other through member functions
It does not have any proper way for hiding data so it is less secure	OOP provides an easy way to add new data and function.
It does not have any proper way for hiding data so it is less secure	OOP provides Data Hiding so provides more securely
Overloading is not possible	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data

1.1.2 Concept of Classes and Object

Class

A class is a user defined design or sample from which objects are created. It represents the set of properties or methods that are similar to all objects of one type. In general, class declarations can comprise of these components, in order:

1. **Modifiers:** A class can be public or has default access.
2. **Class keyword**: class keyword is used to create a class.
3. **Class name**: The name should begin with an initial letter (capitalized by convention).
4. **Superclass** (if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces** (if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body surrounded by braces, { }.

Object

It is an essential unit of Object-Oriented Programming and represents the actual life entities. A typical Java program creates many objects, which as you know, interact by calling methods. An object consists of:

1. **State:** It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Any entity that has state and behavior is understood as an object. For instance , a chair, pen, table, keyboard, bike, etc. It is often physical or logical.

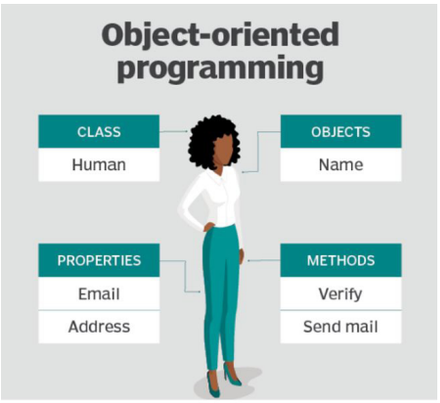


(c) Amity University Online

Notes

An Object is often defined as an instance of a category. An object contains an address and takes up some space in memory. Objects can communicate without knowing the small print of every other's data or code. The sole necessary thing is that the sort of message accepted and therefore the sort of response returned by the objects.

Example: A cat is an object because its states like color, name, moustache, breed, etc. also as behaviors like wagging the tail, playing, eating, etc.



Lets understand class and object this way: In Chapter 1, we introduced Ashok. Now, meet Ravi, another customer at Pepperfry. As customers of Pepperfry, Ashok and Ravi share some similar information. For example, both have a name, phone number, and a budget—information that is relevant when describing customers. This information is recognized as object attributes.

An object attribute definition permits objects to have independent attribute values. For example, Ashok may have a larger budget and thus a larger budget value (say Rs.20000) than Ravi whose budget may be Rs.10000. together; the values of an object's attributes represent the state of the object.

Besides attributes, Ashok and Ravi also exhibit some behavior typical of a customer. For instance, Ashok and Ravi execute a method when making a purchase. Let us call this method buy(). The method buy() is made up of a set of operations that Ashok and Ravi would use to send a purchase request to a salesperson.

Structurally, Ashok and Ravi can be represented as follows:

Ashokas an Object

Attributes:

name = Ashok

phone number = 1234567890

budget = 20000

Methods:

buy() {send a purchase request to a salesperson}

getBudget() {return budget}

Ravi as an Object

(c) Amity University Online

Notes

Attributes:

name = Ravi

phone number = 2134567890

budget = 10000

Methods:

buy() {send a purchase request to a salesperson}

getBudget() {return budget}

Name, phone number and budget are attributes while buy() and getBudget() are methods of the two objects. Note that both objects share a common definition of attributes and methods. In fact, all customers of Pepperfry share the same set of attribute and method definitions. They all have attributes name, phone number and budget, and methods buy() and getBudget(). In defining these objects, a common definition known as class is used.

A class is a definition template for organizing and creating objects with the same attributes and methods. Ashok and Ravi, being customers of Pepperfry, can therefore be defined by a class called Customer as follows:

Class Customer

Attributes:

name

phone number

budget

Methods:

buy() {send a purchase request to a salesperson}

getBudget() {return budget}

"One major difference between objects and class is in the means of attributes and methods are treated in objects and classes. A class is a definition about objects; the attributes and methods in a class are thus declarations that do not contain values. However, objects are created instances of a class. Each has its own attributes and methods. The values of the set of attributes describe the state of the objects."

Let us now examine the salespersons. Salespersons also have attributes and methods. Mukesh and koyal are two salespersons at Pepperfry. They are capable of a behavior typical of a salesperson, for example, taking orders from customers. To fulfill their role as salespersons in a purchase transaction, Mukesh and koyal perform a method. We shall call this method takeOrder(), and represent Mukesh and koyal as follows:

Mukesh as an Object

Attributes:

name = "Mukesh"

(c) Amity University Online

Notes

Methods:

takeOrder() {

check with warehouse on stock availability

check with warehouse on delivery schedule

if ok

then {instruct warehouse to deliver stock(address, date)

return ok}

else return not ok

}

Koyal as an Object

Attributes:

name = “Koyal”

Methods:

takeOrder() {

check with warehouse on stock availability

check with warehouse on delivery schedule

if ok

then {instruct warehouse to deliver stock(address, date)

return ok}

else return not ok

}

Being salespersons, Mukesh and Koyal share parallel attributes and methods as expected. Like the customers, Sales person can be described by a class called SalesPerson with the following representation:

Class SalesPerson

Attributes:

name

Methods:

takeOrder() {

check with warehouse on stock availability

check with warehouse on delivery schedule

if ok

(c) Amity University Online

Notes

then {instruct warehouse to deliver stock(address, date)

return ok}

else return not ok

}

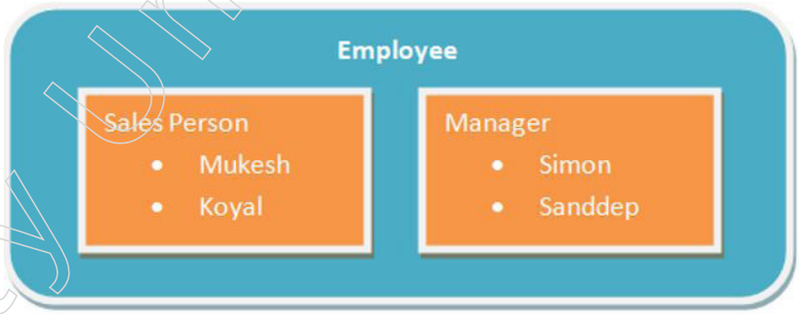
Make sure that the definition of the SalesPerson class is unlike the Customerclass since customers and salespersons behave differently—customers make ordersand salespersons take orders.

Hierarchical Relationship of Classes

Classes formed can be organized in a hierarchical manner. Based on the position of a class in the hierarchy, it may be known as a superclass or a subclass of a class.

Earlier, we introduced Mukesh and Koyal as salespersons of Pepperfry. We also mentioned that Mukesh and Koyal are objects of the SalesPerson class. Let us introduce two more employees, Simon and Sandeep. Specifically, Simon and Sandeep are reporting managers with properties that are slightly different from Mukesh and Koyal. We will classify Simon and Sandeep as objects of a different class: Reporting Manager. All four persons are employees of Pepperfry and objects of another class: Employee. The relationships of these classes and objects are illustrated in Figure below.

Note that Mukesh and Koyal are shown as instances of the SalesPerson class while Simon and Sandeep are instances of the Reporting Manager class. The enclosing border of the Employee class over these objects indicates that the objects are also instances of the Employee class.



Employee, Sales Person and Manager Classes

Mukesh and Koyal therefore belong to two classes: the Employee class and SalesPerson class. Likewise, Simon and Sandeep belong to the Employee and Reporting Manager class. This implies that the information about Mukesh and Koyal as employees is also true of them as salespersons. We can thus refer to Mukesh as an employee or a salesperson.

Which class Mukesh is referred to is a matter of generality. When Mukesh is referred to as an employee, we are being general about who he is but when he is referred to as a salesperson, specific information about his role and employment is specified. For example, Mukesh takes orders and earns a commission for each sale since he is a salesperson but this does not apply to Sandeep who is a manager, despite

(c) Amity University Online

Notes

the fact that both are employees of Pepperfry. Similarly, when we speak of an object as an employee, we are being general and its differences with objects of other classes are ignored.

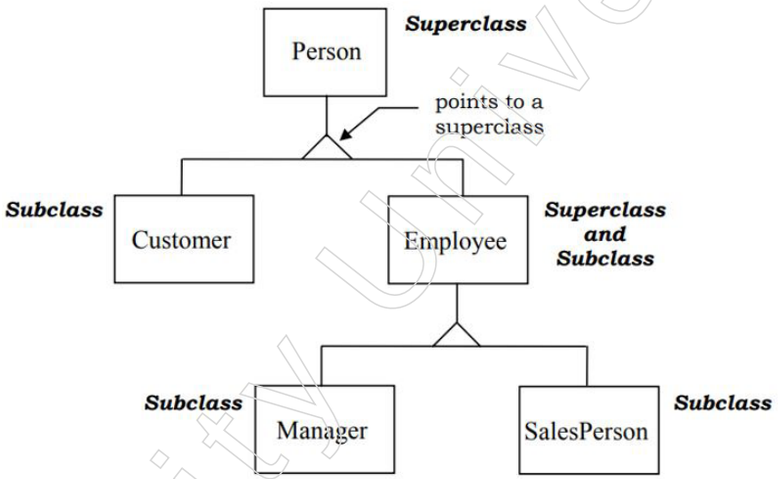
The Employee class is said to be a generalized class of SalesPerson and Reporting Manager. Conversely, SalesPerson and Reporting Manager are said to be specialized classes of the Employee class. Generalized and specialized classes can be organized into a class hierarchy with the generalized classes placed toward the top of the hierarchy and the specialized classes toward the bottom of the hierarchy.

A specialized class is known as a subclass of a class while the generalized class is known as a superclass of a subclass in object-oriented terms. For example, SalesPerson is a subclass of the Employee class which is also the superclass of SalesPerson.

A Class Hierarchy Diagram

The hierarchical relationships among classes can be seen in a class hierarchy diagram in below. A box in the figure below represents a class while a triangle denotes the hierarchical relationship between classes with a superclass positioned at the peak. Subclasses are placed toward the foot of a class hierarchy diagram.

A class can be a superclass to a class or a subclass of another class or both depending on its position in the hierarchy top or bottom.



A class hierarchy diagram.

- Reporting manager and SalesPerson are subclasses of the Employee class.
- Employee is a subclass of the Person class.
- Person is a superclass of Customer and Employee class.
- Employee is thus a superclass (to Reporting manager and SalesPerson) and a subclass (of Person) in the hierarchy.

Generalization

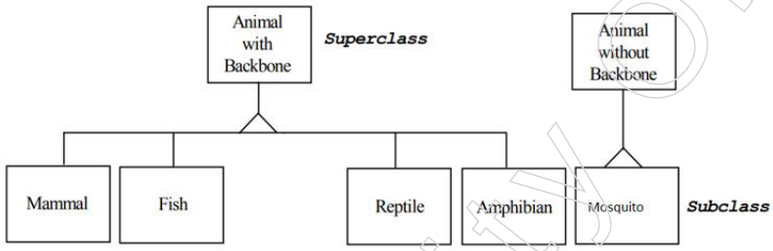
Generalization is the act of capturing similarities between classes and defining the similarities in a new generalized class; the classes then become subclasses of the

(c) Amity University Online

Notes

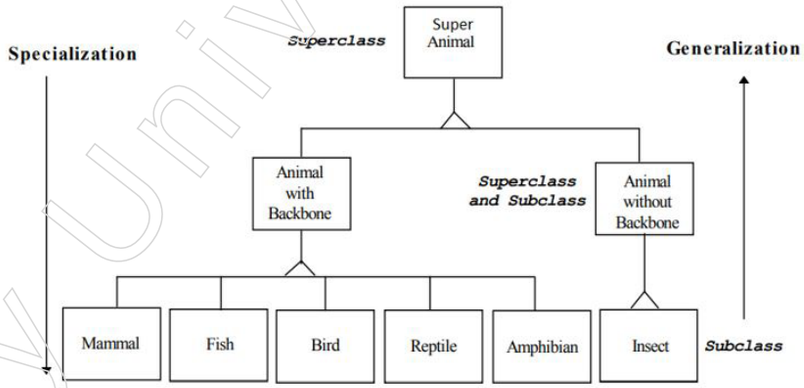
generalized class. For example, the Mammal, Fish, Reptile, and Amphibian classes introduced earlier, are similar in that all objects from these classes have a backbone. Based on this similarity, we can refer to them via a new superclass, say Animal-with-Backbone.

Hence, we can refer to Kermit (an object of the Amphibian class) as an object of the Animal-with-Backbone class too. Similarly, the Mosquito class can be generalized into an Animal-without-Backbone class since objects from the Insert class are without a backbone. Flowchart below summarizes the relationships of these classes.



Generalizing classes.

The Animal-with-Backbone class and Animal-without-Backbone class can be more generalized by taking into consideration the similarities of objects from these two classes. Let us call the generalized class, Super Animal. The generalization of properties of Animal with-Backbone class and Animal-without-Backbone class into the Super Animal class is shown in diagram below.



A class hierarchy for animals.

The Super Animal class, being the topmost class in the class hierarchy, is thus the most general class of the entire Super Animal class hierarchy. This means that Swift, which is an object of the Fish class, is also an object of the Animal-with-Backbone class and Super Animal class, for example. When we refer to Swift as an object of the Animal class, we are being general about it and we would be ignoring specific information about Swift as a Fish in this reference.

Specialization:

On the contrary, specialization is the act of capturing differences among objects in a class and creating new different subclasses with the differences. In this fashion, we are specializing information about objects of the superclass into subclasses. For

(c) Amity University Online

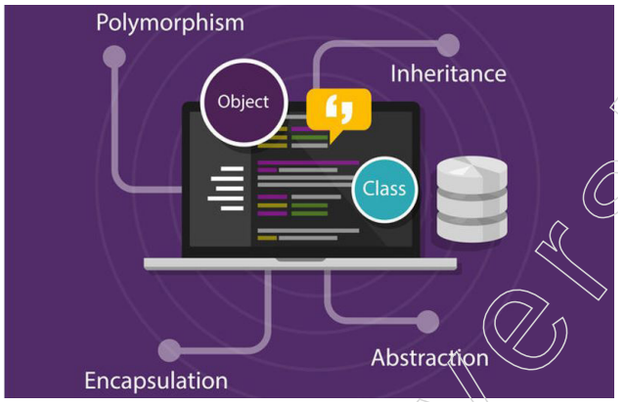
Notes

example, in creating Animal-with-Backbone class and Animal-without-Backbone class from the Super Animal class, we are distinguishing information about objects with a backbone from others without a backbone into Animal-with-Backbone and Animal-without Backbone classes.

Eventually, only objects with a backbone would be classified into the Animal-with-Backbone class and the others into the Animal-without-Backbone class. Similarly, objects from the Animal-with-Backbone class can be further classified into the Mammal, Fish, Reptile, or Amphibian classes depending on their properties definition.

1.1.3 Object reference, Abstraction, Encapsulation

Four fundamentals of object-oriented programming



Primitive Types

"Java has 8 built-in data types and is called primitive types. These data types signify the building blocks for Java objects as all Java objects are just a complex collection of these primitive data types."

Data Type	Size	Range	Default
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32767	0
int	4 bytes	-2147,483,648 to 2147,483,647	0
long	8 bytes	-9223,372,036,854,755,808 to 9223,372,036,854,755,807	0
float	4 bytes	±1.4E-45 to ±3.40282347E+38F	0.0
double	8 bytes	±439E-324 to ±1.7976931348623157E+308	0.0
char	2 bytes	\u0000 to \uffff	\u0000
boolean	1 byte	true or false	false

Object Reference:

In Java, variables did not contain objects, but only the references to objects.

The objects are made and assigned in memory independently from the declarations of variables. Specifically:

Objects denoted by a literal (such as literals of type String, int e.g., "Hi" "Hello", 1, 2 etc.) are allocated in memory at compilation time;

Notes

All other objects must be constructed and assigned through an explicit statement.

A variable whose type is a class have a reference to an object of the class (i.e., the address of the memory location where the object is allocated).

Example:

String a;

a = “xyz;

The first statement declares a variable a of type String. Such a variable is not initialized yet. The second statement assigns to such a variable the reference to the object denoted by “xyz”.

Notice that two variables may have a reference to the same object.

Example:

String a, b;

a = “xyz”;

b = c;

After these two statements, both a and b contain a reference to the object denoted by “xyz”.

Variables of type object reference may have also an unusual value, namely null. Such value defines that the variable does not represent any object. Do not get confuse by the fact that, variables whose value is null with variables that are not initialized. A variable which is not initialized does not have any value, not even null.

Abstraction:

Abstraction inobject-oriented programming “shows” only necessary things and “hides” unnecessary information. The main purpose of abstraction is hiding the needless details from the users. Abstraction is selecting data from a larger group to show only appropriate details of the object to the user. It helps in plummeting programming difficulty and efforts. It is one of the main concepts of OOPs.

Encapsulation is supported by two subsidiary concepts: bundling and information hiding. Bundling is the act of combining a set of methods with a set of data as the only means of disturbing the values of the data. Related data and methods are bring together in bundling, thus increasing the cohesiveness of object explanation.

Information hiding refers to the hiding of internal demonstration of data and methods from the users of these data and methods. By using information hiding, data access on an object is limited to a set of widely available methods. While the client is conscious of the existence of the methods, it does not know how the processes are internally structured. In this way, information hiding enables the separation of the what from the how of object definition.

Let’s Study Abstraction in OOPs with example:

We want to create a hotel application and you are asked to collect all the information about your client. And we get the following information about the customer:

(c) Amity University Online

Notes



For creating a hotel application we don't need all the information provided by the customer.So we only select he useful information for our hotel application from that pool. Fields like Name, Address, Favorite food, contact number which make sense for a hotel application which is an Abstraction example in OOPs.

Since we have selected the necessary information and removed the unnecessary information from a bigger pool, the process is referred as Abstraction in OOPs.

Nevertheless, the same information once extracted can be re-used for a wide range of applications. For instance, you can use the same data for Job application, hospitalportal application, a bankingdatabase, etc. with some modification. Hence, it becomes your Master Data.

Advantages of Abstraction

The main benefit of using an Abstraction in Programming is that it allows you to group several related classes as siblings.

Abstraction in Object Oriented Programming helps to reduce the complexity of the design and implementation process of software.

Encapsulation: Encapsulation is defined as wrap up variables (data) and methods (information) under a single unit. In Object Oriented Programming, Encapsulation is explained as combining together the data and the functions that manipulates them.

Consider a real life example of encapsulation, in a PSU there are different sections like the accounts section, finance section, HR section etc. The finance section handles all the financial activity and keep records of all the data related to finance. Similarly the HR section handles all the HR related activities and keep records of all the HR activity. Now there may be circumstances when for some reason an official from HR section needs all the data about finance in a particular month. In this case, HR team member is not allowed to directly access the data of finance section. He/she will first have to contact some other officer in the finance section and then request him to give the particular data. This is what encapsulation is. Here the data of finance section and the employees that can manipulate them are wrapped under a single name "finance section".

Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like HR, finance or accounts is hidden from any other section.

(c) Amity University Online

Notes

In C++ encapsulation can be implemented using Class and access modifiers. Look at the following example:

```
// Java program to demonstrate encapsulation
class Encapsulate {

    private String DemoName;
    private int DemoRoll;
    private int Demoage;

    public int getAge() { return DemoAge; }
    public String getName() { return DemoName; }

    public int getRoll() { return DemoRoll; }
    public void setAge(int newAge) { DemoAge = newAge; }
    public void setName(String newName)
    {
        DemoName = newName;
    }
    public void setRoll(int newRoll) { DemoRoll = newRoll; }
}

public class TestEncapsulation {
    public static void main(String[] args)
    {
        Encapsulate obj = new Encapsulate();

        // setting values of the variables
        obj.setName("Amit");
        obj.setAge(20);
        obj.setRoll(01);

        // Displaying values of the variables
```

(c) Amity University Online

Notes

```
            System.out.println("Demo's name: " + obj.getName());
            System.out.println("Demo's age: " + obj.getAge());
            System.out.println("Demo's roll: " + obj.getRoll());
        }
    }
```

Output

Demo's Name: Amit

Demo's Age: 20

D3mo's Roll: 01

In the above program the variable a is made private. This variable can be used and manipulated only using the functions get() and set() which are there inside the class. Thus we can assume that, the variable a and the functions get() and set() are clubbed together which is nothing but encapsulation.

Role of access specifiers in encapsulation

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

- The data members should be labeled as private using the private access specifiers.
- The member function which manipulates the data members should be labeled as public using the public access specifier.

1.1.4 Inheritance, Polymorphism

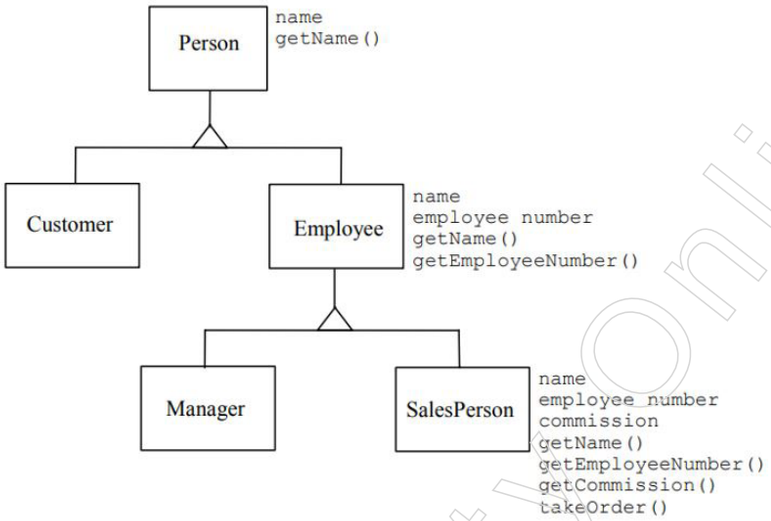
Inheritance

From a software reuse standpoint, generalized properties defined in superclasses should be made available to subclasses without having to declare them explicitly in the subclasses. In object-oriented programming, such reuse is possible via inheritance.

Inheritance is the capability of a subclass to take on the common properties of superclasses in the inheritance chain. The properties then form part of the subclass' definition. Inheritance enables superclasses' properties to be submerged downward to the subclasses in a class hierarchy, and makes the properties accessible as part of the subclasses' definition. These properties are supposed to be inherited (or taken on) by the subclasses. Using inheritance, the SalesPerson class of Figure below can now be defined by a combination of properties from:

(c) Amity University Online

Notes



Common properties in classes.

Using inheritance, the SalesPerson class of Figure above can now be defined by acombination of properties from:

- the Employee class;
- the Person class; and
- its own specific attribute and method definition.

Figure below shows the modified class hierarchy for persons (with inherited properties highlighted in bold). The Person class has the following explanation:

Class Person {

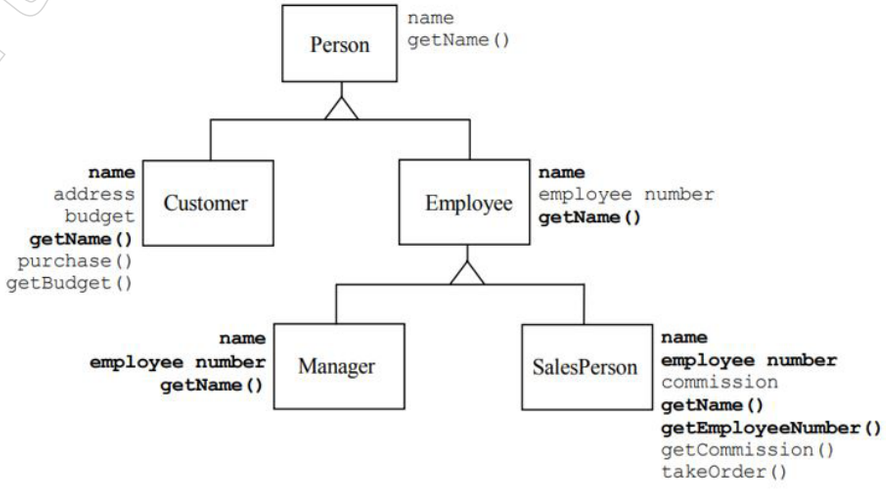
Attributes :

name

Methods :

getName() {return name}

}



(c) Amity University Online

Classes with inherited properties.

Notes

The Employee class is reduced to (with inherited properties highlighted in bold in above figure):

Class Employee {

Attributes :

name (inherited from Person class)

employee number

Methods :

getName() {return name} (inherited from Person class)

getEmployeeNumber(){return employee number}

}

The SalesPerson class with inherited properties simplified into (highlighted in bold in above figure)

Class SalesPerson {

Attributes :

name (inherited from Person Class)

employee number(inherited from Employee class)

commission

Methods :

takeOrder(who, stock, address, date) {

check with warehouse on stock availability

check with warehouse on delivery schedule

if ok

then {instruct warehouse to deliver stock

to address on date

return ok}

else return not ok

}

getName() {return name} (inherited from Person class)

getEmployeeNumber(){return employee number}

(c) Amity University Online

Notes

(inherited from Employee class)

getCommission() {return commission}

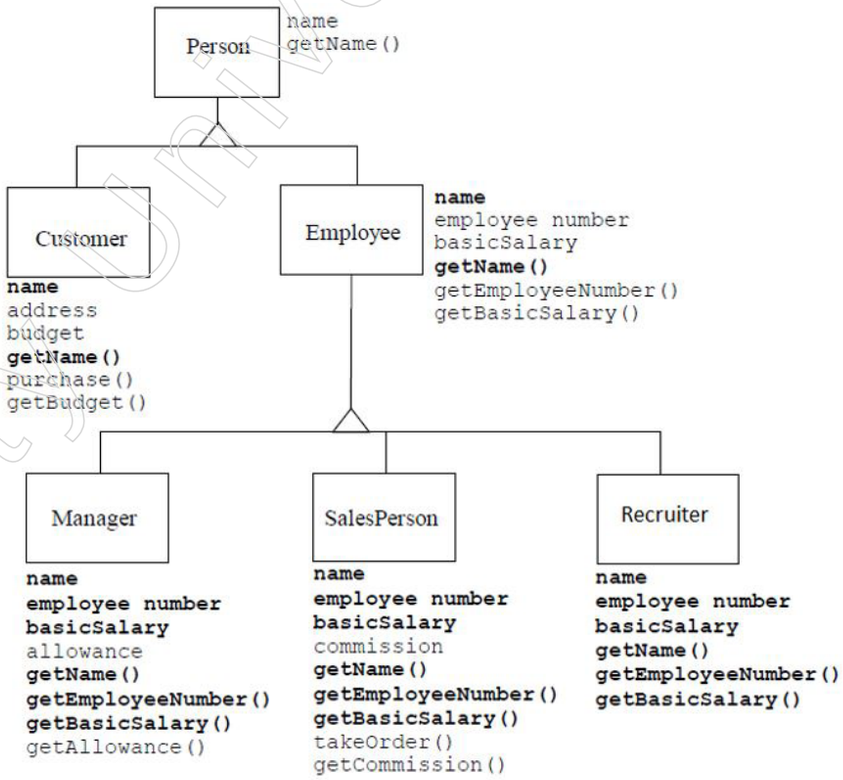
}

Attributes name and employee name, and methods getName( ) and getEmployeeNumber () of the SalesPerson class are not clearly defined in the SalesPerson class but are propagated downward from the superclasses through inheritance.

Only downward propagation of properties from superclasses to subclasses is allowable. There is no uphill propagation of properties in object-oriented programming. Therefore, information specific to subclasses are inimitable to subclasses and are not propagated to superclasses. Consequently, attribute commission and method getCommission() of the SalesPerson class do not form part of the Employee class description.

Implementing Inheritance

Let us now extend the Person class hierarchy to include a new class, Recruiter. Figure below shows the adapted class hierarchy with inherited properties highlighted in bold. The information in the extended hierarchy suggests that all employees have a basic salary except managers and salespersons, which are paid an allowance and commission, respectively.



Including the Recruiter class.

The Superclass–Subclass relationship in a class hierarchy is denoted in the code by the keyword extends. This suggests that a subclass is an addition of a superclass.

(c) Amity University Online

Notes

For example, the following code fragment suggests that Employee is an extension of the Person class; Manager, SalesPerson and Recruiter, being subclasses, are extensions of the Employee class:

```
class Employee extends Person {
 ...
}

class Manager extends Employee {
 ...
}
class SalesPerson extends Employee {
 ...
}
class Recruiter extends Employee {
 ...
}
```

Code 1 is the code implementing the Person hierarchy. Code execution begins with the Employee class because it is the only class that contains static void main( ). To execute main( ), type the following at the command prompt line:

$ java Employee

Executing the code produces the following output:

The Manager Simon (employee number 01234M) has a salary of 9000

The Recruiter Selene (employee number 98765S) has a salary of 2500

The Manager Simon also has an allowance of 2200

The output suggests that some information was made available from the superclasses in deriving the manager's and Recruiter's salary. Objects instantiated from the Manager or Recruiter class were able to respond to requests for their basicSalary because they have inherited from the Employee class the attribute basicSalary and method getBasicSalary().

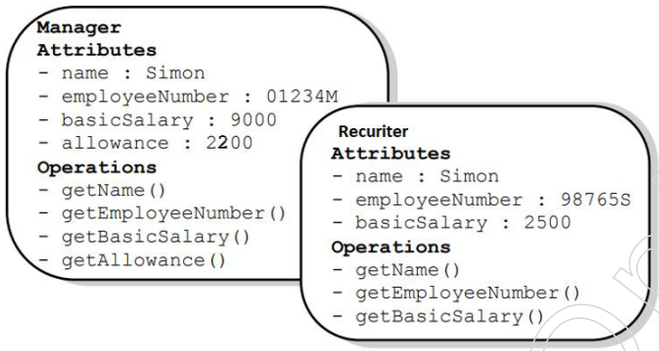
Let us examine main(). The code begins with the instantiation of two objects, a manager and a Recruiter. The manager object is referenced by variable m while the Recruiter object is referenced by variable s:

Manager m = new Manager("Simon", "01234M", 9000.0f, 2200.0f); Recruiter s = new Recruiter("Selene", "98765S", 2500.0f);

The state of the two objects is depicted in below figure:

(c) Amity University Online

Notes



State of Manager and Recruiter object.

Code 1: Inheritance in the extended Person hierarchy

```
class Person {
 private String name;
Person(String aName) {name=aName;}
 public String getName() { return name; }
}
class Employee extends Person {
 private float basicSalary;
 private String employeeNumber;
Employee(String aName, String aEmployeeNumber,
 float aBasicSalary) {
 super(aName);
employeeNumber = aEmployeeNumber;
basicSalary = aBasicSalary;
 }
 public String getEmployeeNumber() { return employeeNumber; }
 public float getBasicSalary() { return basicSalary; }
 public static void main(String argv[]) {
 Manager m = new Manager(
System.out.print("The Manager "+m.getName()+
 " (employee number "+m.getEmployeeNumber()+")");
System.out.println(" has a salary of "+m.getBasicSalary());
System.out.print("The Recruiter "+s.getName()+
 " (employee number "+s.getEmployeeNumber()+")");
```

(c) Amity University Online

Notes

```
System.out.print("The Manager "+m.getName());
System.out.println(" also has an allowance of " +m.getAllowance());
 }
}
class Manager extends Employee {
 private float allowance;
Manager(String aName, String aEmployeeNumber,
 float aBasicSalary, float aAllowanceAmt) {
super(aName, aEmployeeNumber, aBasicSalary);
 allowance = aAllowanceAmt;
 }
 public float getAllowance() {
 return allowance;
 }
}
class Recruiter extends Employee {
Recruiter (String aName, String aEmployeeNumber,
 float aBasicSalary) {
super(aName, aEmployeeNumber, aBasicSalary);
 }
}
```

The remainder of main() are output statements. Some methods from various classes, invoked in producing the outputs, for example, getName() from the Person class, and getEmployeeNumber() and getBasicSalary() from the Employee class, are propagated through the inheritance mechanism to the subclasses, Manager and Recruiter.

Code Reuse

By allowing information of a superclass to be used by subclasses, the information is said to be reused at the subclass level. All newly created instances of the subclasses would have as part of their description the inherited information. employee number, basic salary, and getEmployeeNumber() of the Employee class and name and getName() of the Person class are said to be reused by the Manager and Recuiter class.

Making Changes in Class Hierarchy

Changes to software condition are inevitable. Let's discuss how changes in a class

(c) Amity University Online

Notes

hierarchy affect software maintenance as a whole. The following situations will be discussed:

- Change in property definition for all subclasses.
- Change in property definition for some subclasses.
- Adding/deleting a class.

1. Change in Property Definition for All Subclasses

Suppose a change in representational scheme of the employee number in Figure (Including the Recuriter class). is required. This change will impact not only the attribute employee number but also the method getEmployeeNumber() and possibly other classes that inherit employee number.

We examine this change in two situations:

- inheritance is not available;
- inheritance is available.

Inheritance Is Not Available: In situations where inheritance is not available, the attribute employee number and method getEmployeeNumber() would have to be defined in all the relevant classes, for example, Employee, Manager, SalesPerson and Recruiter. The change in representational scheme of employee number would therefore have to be effected individually on these classes. The redundancy coming from the multiple definition of employee number and getEmployeeNumber() may lead to inconsistency in description if the change is not carried out properly.

Inheritance Is Available: With inheritance, the case is different. We would first define attribute employee number and method getEmployeeNumber() in Employee class and let subclasses Manager, SalesPerson and Recruiter inherit these definitions from Employee class. The necessary change in representational scheme for attribute employee number would be limited to the Employee class. The change would be propagated to the subclasses via inheritance. In this way, the change is thus restricted to the superclass, enabling a uniform and constant property description for all subclasses. Besides, redundancy in property description at the subclass level can be minimized and software maintenance enhanced.

2. Change in Property:

Description for Some Subclasses In some situations, a change in property description at the superclass level may not necessarily apply to all subclasses. The above solution would therefore not apply in these situations. To illustrate, let us extend the Person class hierarchy further to include two more employee classes: Engineer and Clerk. Let us assume the following for a Pepperfry employee:

- a manager—basic salary plus allowance;
- a salesperson—basic salary plus commission;
- a recruiter—basic salary;
- a engineer—basic salary;
- a clerk—basic salary.

At the Employee class, a getPay() method is defined to return the monthly pay of

(c) Amity University Online

Notes

an employee given that the method applies to all classes of employee. The description of the Person class remains the same as before:

```
class Person {
 private String name;
Person(String aName) {name=aName;}
 public String getName() { return name; }
}
```

Employee extends Person as follows:

```
class Employee extends Person {
private float basicSalary;
private String employeeNumber;
Employee(String aName, String aEmployeeNumber,
float aBasicSalary) {
super(aName);
employeeNumber = aEmployeeNumber;
basicSalary = aBasicSalary;
}

public String getEmployeeNumber() { return employeeNumber; }
public float getBasicSalary() { return basicSalary; }
public float getPay() { return basicSalary; }
public static void main(String argv[]) {

Manager m = new Manager(“Simon”, “01234M”, 9000.0f, 2200.0f);
Recruiter s = new Recruiter(“Selene”, “98765S”, 2500.0f);
Engineer t = new Engineer(“Terrence”, “42356T”, 2000.0f);
Clerk c = new Clerk(“Nancy”, “68329C”, 1200.0f);
System.out.print(“The Manager “+m.getName()+
“ (employee number “+m.getEmployeeNumber()+”)”);
System.out.println(“ has a pay of “+m.getPay());
```

(c) Amity University Online

Notes

```
System.out.print("The Recruiter "+s.getName()+
" (employee number "+s.getEmployeeNumber()+")");
System.out.println(" has a pay of "+s.getPay());
System.out.print("The Engineer "+t.getName()+
System.out.println(" has a pay of "+s.getPay());
" (employee number "+t.getEmployeeNumber()+")");
System.out.println(" has a pay of "+t.getPay());
System.out.print("The Clerk "+c.getName()+
" (employee number "+c.getEmployeeNumber()+")");
System.out.println(" has a pay of "+c.getPay());
}
}
```

As before, main() is defined in the Employee class with additional code for Engineer and Clerk class highlighted in bold. There is no change in class definition for Manager and Recruiter. Engineer and Clerk extend Employee, since they are subclasses of Employee:

```
class Engineer extends Employee {
Engineer (String aName, String aEmployeeNumber,
float aBasicSalary) {
super(aName, aEmployeeNumber, aBasicSalary);
}
}
class Clerk extends Employee {
Clerk (String aName, String aEmployeeNumber,
float aBasicSalary) {
super(aName, aEmployeeNumber, aBasicSalary);
}
}
```

Executing main() produces the following output:

```
The Manager Simon (employee number 01234M) has a pay of 9000
```

(c) Amity University Online

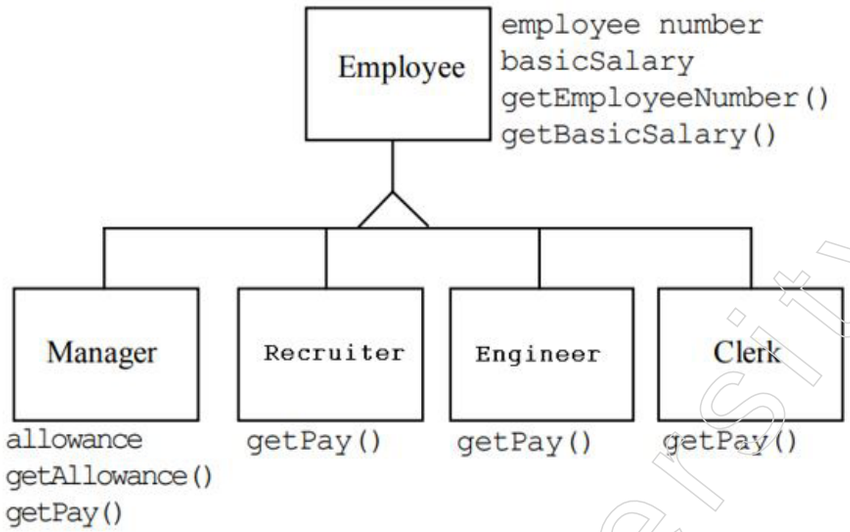
Notes

The Recruiter Selene (employee number 98765S) has a pay of 2500

The Engineer Terrence (employee number 42356T) has a pay of 2000

The Clerk Nancy (employee number 68329C) has a pay of 1200.

A brief examination of the output reveals a mistake in the manager's pay: an omission of allowance amounting to 2200. What has gone wrong?



Extended Employee class hierarchy.

The above problem can be approached in two ways:

- Remove the getPay() method from the Employee class and define it individually in the subclasses (Recruiter, Engineer, Clerk, and Manager).
- Maintain the definition of getPay() method in Employee class and redefine it in the Manager class.

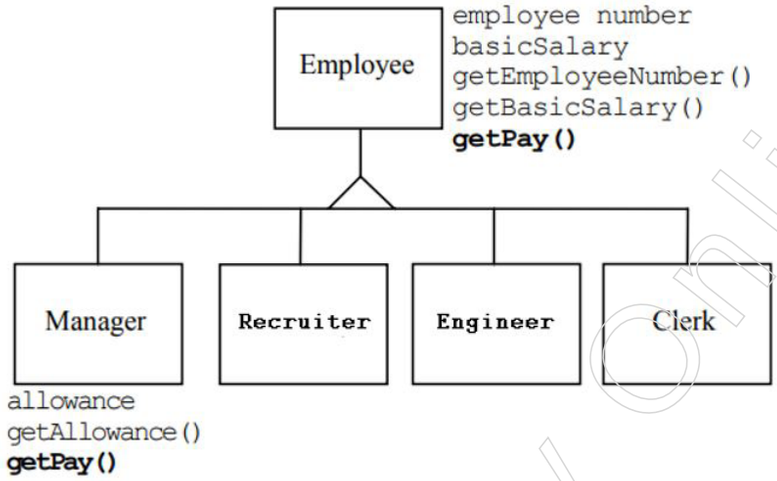
Above Figure(Extended Employee class hierarchy.) illustrates a class diagram for the first approach. Each of the subclasses has its own implementation of the getPay() method. One drawback of this approach is that the description of the getPay() method has to be repeated in all the subclasses. This is highly unproductive and can be difficult to maintain especially in cases where the number of subclasses is large.

In the second approach, the definition of the getPay() method is maintained at the Employee class but redefined in the Manager class. This ensures that the getPay() method is inherited by all subclasses of Employee, including the Manager class.

Since a similar getPay() method is defined in Manager, the getPay() method of the Manager class would be used in the resolution of method call by the objectoriented system instead. This is depicted in Figure below and code-2.

(c) Amity University Online

Notes



Redefining getPay() method of Manager.

The getPay() method of the Manager class is said to redefine the getPay() method of the Employee class. Note that a redefined method has the same method name and parameter definition of a redefining method. While a redefining method has the same method signature with the redefined method, the execution of the methods may differ. In this situation, the getPay() method of the Manager class includes an additional calculation of the allowance component.

Code 2: Redefining the getPay() method.

```
class Person {
...
}
class Employee extends Person {
...
public float getPay() { return basicSalary; }
public static void main(String argv[]) {
...
}
}
class Manager extends Employee {
private float allowance;
Manager(String aName, String aEmployeeNumber,
float aBasicSalary, float aAllowanceAmt) {
super(aName, aEmployeeNumber, aBasicSalary);
allowance = aAllowanceAmt;
```

(c) Amity University Online

Notes

```
}
public float getAllowance() {
return allowance;
}
public float getPay() {
return (basicSalary + allowance);
}
}
class Recruiter extends Employee {
...
}
class Engineer extends Employee {
...
}
class Clerk extends Employee {
...
}
```

Judging from the outcome of the two solutions, both approaches are correct:

The Manager Simon (employee number 01234M) has a pay of 11200

The Recruiter Selene (employee number 98765S) has a pay of 2500

The Engineer Terrence (employee number 42356T) has a pay of 2000

The Clerk Nancy (employee number 68329C) has a pay of 1200

However, the second approach is better than the first approach as it enhances software reuse and reduces the effect of change on other classes.

3. Adding/Deleting a Class

Adding a class into an existing class hierarchy can be unfavorable to the stability of the hierarchy. It is always suggested that the addition of a new class be created as a subclass in the class hierarchy. The description of existing classes will not be adversely affected by this approach. To illustrate, let us consider an example of geometrical shapes.

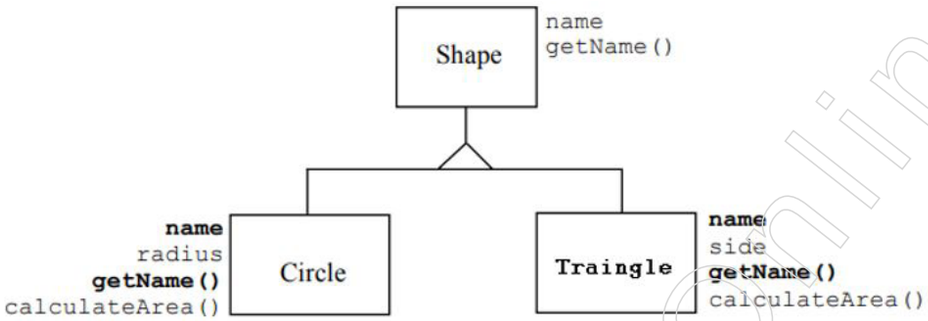
Figure below is a class hierarchy of shapes. Shape is a generalized class of Circle and Traiangle. All shapes have a name and a measurement by which the area of the shape is computaed.

The attribute name and method getName() are defined as properties of Shape. Circle and Traiangle, being subclasses of Shape, inherit these properties (highlighted in

(c) Amity University Online

Notes

bold in Figure below).



Class hierarchy of Shape, Circle and Traiangle

The Shape class has the following description:

```
class Shape {
 private String name;
Shape(String aName) {name=aName;}
 public String getName() {return name;}
 public float calculateArea() {return 0.0f;}
 public static void main(String argv[]) {
Circle c = new Circle("Circle C");
Traiangle s = new Traiangle("Traiangle S");
 Shape shapeArray[] = {c, s};
for (int i=0; i<shapeArray.length; i++) {
System.out.println("The area of " + shapeArray[i].getName()
     + " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
}
 }
 }
```

The attribute name is declared as private in the Shape class. To make it known to other objects, a getName() method is defined in the Shape class to return the value of attribute name.

Circle and Traiangle extend Shape and have the following description:

```
class Circle extends Shape {
 private int radius;
Circle(String aName) {
 super(aName);
 radius = 3;
```


(c) Amity University Online

```
}
public float calculateArea() {
float area;
area = (float) (3.14 * radius * radius);
return area;
}
}
class Traiangle extends Shape {
private float side, height;
Traiangle(String aName) {
super(aName);
side = 3;
height = 5;
}
public float calculateArea() {
floatarea;
area = 1/2*side * height;
return area;
}
}
```

As usual, program execution begins with main() and the following output is produced when main() is executed:

The area of Circle C is 28.26 sq. cm.

The area of Traiangle S is 7.5 sq. cm.

Two objects are created in main()—a Circle object referenced by the variable c and a Traiangle object referenced by the variable s. The creation of a Circle object involves a call to the Circle class constructor method via the new keyword. A name parameter is required to activate the Circle constructor method. For Circle, the name parameter is the string "Circle C".

A call is made to the Circle's superclass constructor method via the statement:

```
super(aName);
```

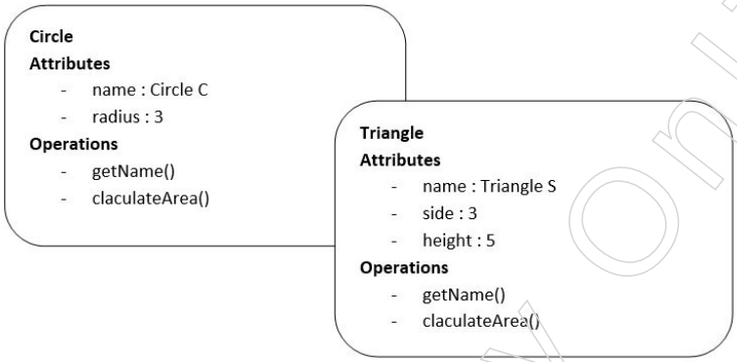
The call assigns the value of the parameter ("Circle C") to the Circle object's attribute name. When the assignment is done, control returns to the Circle class's constructor method. Subsequently, the radius attribute of the Circle object is assigned the value 3 via the statement:

(c) Amity University Online

Notes

radius = 3;

Likewise, the Traiangle object is created and its attributes updated in the execution. By now, the Circle and Traiangle objects have a state as illustrated in Figure below.



State of Circle and Traiangle object.

An array shapeArray is declared in the next statement. The reference variables c and s are assigned to the array elements. Iterating through the array (via the forloop), the area of the respective shape is produced by the statement

System.out.println("The area of " + shapeArray[i].getName() + " is " + shapeArray[i]. calculateArea()+" sq. cm.\n");

Suppose we want to add to the Shape class hierarchy a new class called Square. Code-3 shows the modified code with new additions highlighted in bold.

Code-3 Adding a square

```
class Shape {
...
public static void main(String argv[]) {
Circle c = new Circle("Circle C");
Triangle s = new Triangle("Triangle S");
Square t = new Square("Square T");
Shape shapeArray[] = {c, s, t};
for (int i=0; i<shapeArray.length; i++) {
System.out.println("The area of " + shapeArray[i].getName()
+ " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
}
}
}
class Circle extends Shape {
...
```


(c) Amity University Online

Notes

```
}
class Traingle extends Shape {
...
}
class square extends Shape {
private int side;
Square(String aName) {
super(aName);
side = 4;
}
public int calculateArea() {
int area = side * side;
return area;
}
}
```

To add the new Square class, the following is involved:

1. Add a statement to create a square object in main().
2. Add a statement to include the newly created square into shapeArray in main().
3. Create a new square class as a subclass of Shape.

It is clear that subclassing the new Square class into the class hierarchy does not affect the description of the other three classes. Subclassing is specialization and is thus a desired design practice in object-oriented software engineering because it has less impact on software maintenance.

Thus, the deletion of subclasses that are not superclasses to other classes has a less impact on software maintenance.

Polymorphism

The word polymorphism means having many forms. In simple words, we can describe polymorphism as the capability of a message to be displayed in more than one form.

Real life example of polymorphism: A person at the same time can have different characteristic. Like a woman at the same time is a mother, a wife, an employee. So the same person posses different behavior in diverse situations. This is called polymorphism.

(c) Amity University Online

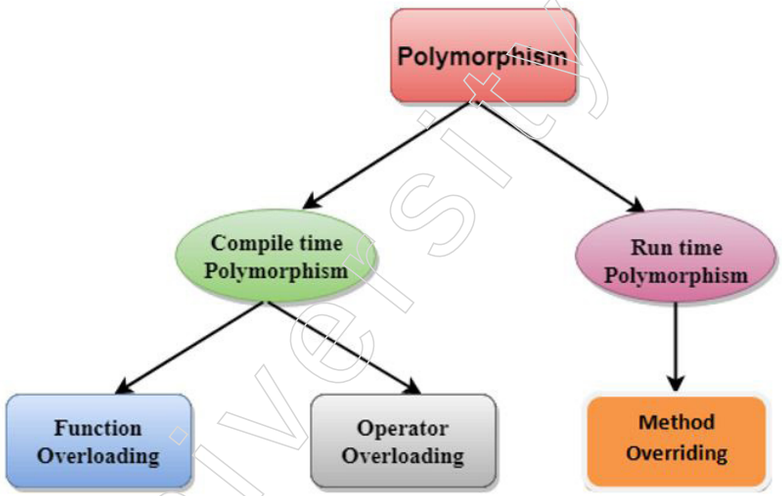
Notes

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, so it means many forms.

Types of Polymorphism in Oops

In Object-Oriented Programming (OOPS) language, there are two types of polymorphism as below:

- Static Binding (or Compile time) Polymorphism, e.g., Method Overloading
- Dynamic Binding (or Runtime) Polymorphism, e.g., Method overriding



1. Compile Time or Static Polymorphism

With Method Overloading, static polymorphism is achieved in Object-Oriented Programming languages that allow the programmer to implement various methods. The names they use are often an equivalent, but their parameters are different. Certain situations are conducive for static polymorphism as below:

- Types of All Parameters should vary.
- The sequence of the Parameters are often different.
- The number of parameters of 1 method should differ from another method.

In the static binding polymorphism, the matching type and number of arguments call upon the overloaded functions.

- As all of this information is out there during the compile time, the compiler selects the acceptable function.
- The function overloading does it, and operator overloading is additionally termed as static binding or early binding.

Program

```
class A                    //  base class
{
```


(c) Amity University Online

Notes

```
int a;
public:
void display()
{
cout<< “Class A “;
}
};

class B: public A          //  derived class
{
int b;
public:
void display()
{
cout<<“Class B”;
}
};
```

Runtime or Dynamic Polymorphism

In the Dynamic Polymorphism, a call to one overridden method is solved during a program’s runtime. Method overriding is one among the prominent samples of Runtime Polymorphism. during this process, the overriding is completed through pointers and virtual functions.

- In Method Overriding, one method is said during a sub-class present during a parent class. the kid class gains a way for implementation.
- During Runtime Polymorphism, the category offers the specification of its own to a different inherited method. This transfer between methods is achieved without modifying the parent class object codes.

Program

```
//  DemoJava program to illustrate Dynamic Method

class ABC
{
    void m1()
```

(c) Amity University Online

Notes

```
    {
        System.out.println("Inside ABC's m1 method");
    }
}

class XYZ extends ABC
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside XYZ's m1 method");
    }
}

class MNO extends ABC
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside MNO's m1 method");
    }
}

// Driver class
class Dispatch
{
    public static void main(String args[])
    {
        // object of type ABC
        ABC a = new ANC();

        // object of type XYZ
```

(c) Amity University Online

Notes

```
        XYZ b = new XYZ();

        // object of type MNO
        MNO c = new MNO();

            ABCref;

        ref = a;
ref.m1();

        ref = b;

        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

```
Inside ABC's m1 method
Inside XYZ's m1 method
Inside MNO's m1 method
```

- In the Run time polymorphism, the object's method is invoked at the run time rather than compile time.
- It is formed possible by method overriding, which is additionally coined as dynamic binding or late binding.
- The process of dynamic polymorphism is relatively slower but flexible than compile-time polymorphism.

Difference between Compile-Time and Run-Time Polymorphism

(c) Amity University Online

Notes

	Compile-Time Polymorphism	Run-Time Polymorphism
Invoking of Function	The function is invoked at the run time	The function is invoked at the compile time.
Common Terms	It is known as overloading, early binding, and static binding.	It is known as overriding, late binding, and dynamic binding.
Method Name and Parameters	In Overloading, more than one method has the same name but with a different number or type of parameters.	In overriding, ore than one method has the same name, number, and type of parameters.
Carriers	It is achieved with function and operator overloading.	It is achieved with virtual functions and pointers.
Execution Time	It executes faster than run-time polymorphism at the compile time.	It executes slower than compile-time polymorphism at the run time.
Flexibility	It is less flexible as everthing executes at the compile time.	It is more flexible as everthing executes at the run time.

Check your Understanding

1. There are two basic aspects of programming _____ and _____.
2. Object oriented programming is a programming model which is based on the concept of _____ .
3. An object contains an address and takes up some space in _____ .
4. Generalization is the act of capturing _____ between _____ and defining the similarities in a new generalized class.
5. Java has 8 built-in data types and is called _____ types.
6. Adding a class into an existing class hierarchy can be unfavorable to the stability of the _____ .
7. A class is a user defined design or sample from which _____ are created.
8. Variables did not contain objects, but only the _____ to objects.

Summary

- An overview of object-oriented programming concepts and their applicability for modeling and representing real-world entities and their interactions in the problem-solving process.
- Object-oriented concepts of object, message, and method.
- An overview of the Java programming language and the potential of productive software development.
- Objects are defined by classes.
- Objects from the same class share the same definition of attributes and methods.
- Objects from the same class may not have the same attribute values.

(c) Amity University Online

Notes

- Objects from different classes do not share the same definition of attributes or methods.
- Objects created from the same class share the same definition of attributes and methods but their state may differ.
- This variable can be used and manipulated only using the functions get() and set() which are there inside the class
- Polymorphism in Java is the ability of an object to take many forms. To simply put, olymorphism in java allows us to perform the same action in many different ways.

Activity

1. Create a program for online hotel reservation using inheritance.

Questions and exercises

1. Define Fundamentals of object oriented programming?
2. What is difference between Procedure Oriented Programming Vs. Object Oriented Programming (OOP)?
3. What is Classes?
4. What is Object?
5. Define object reference?
6. Define Abstraction?
7. What do you mean by Encapsulation?
8. Define Inheritance?
9. What do you mean by polymorphism?

Glossary

- Objevct Oriented Programming: Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic.
- Object: An object can be defined as a data field that has unique attributes and behavior.
- Object Refrence: A link to an object. Object references can be used exactly like the linked objects. The concept of object references becomes clear when assigning the same object to more than one property
- Abstraction: Abstraction is the concept of object-oriented programming that “shows” only essential attributes and “hides” unnecessary information. The main purpose of abstraction is hiding the unnecessary details from the users.
- Encapsulation: Encapsulation is a method of making a complex system easier to handle for end users. The user need not worry about internal details and complexities of the system. Encapsulation is a process of wrapping the data and the code, that operate on the data into a single entity. You can assume it as a protective wrapper that stops random access of code defined outside that wrapper.

(c) Amity University Online

Notes

- Inheritance: The capability of a class to derive properties and characteristics from another class is called Inheritance.
- OOP: Object Oriented Programming

Further Readings:

1. Object Oriented Programming, Dusty Phillips
2. Object-oriented Software Construction, Bertrand Meyer
3. Object-oriented programming with C++, Raimund K. Ege
4. The Object-Oriented Thought Process, Matt A. Weisfeld

Check Your Understanding:

Answers

1. data and instructions
2. objects
3. memory
4. Similarities classes
5. Primitive
6. Hierarchy
7. Objects
8. references

(c) Amity University Online

Unit - 1.2 : Introduction to Java

Notes

Objectives

At the end of this unit, you will be able to understand:

- Difference between Java and C++, and a brief history about Java
- Tools used for Java application: JDK, JRE, JVM
- Use of IDE for developing Java Application
- Types of variables and declaration
- Data types and function used in Java application

Introduction

Java languagewas developed by James Gosling at Sun Microsystems in 1991, later acquired by Oracle Corporation. It's an easy programming language. Java helps in writing codes and also makes compiling, and debugging programming easy. It helps programmer to make reusable code and modular programs.

Java possibly will be a class-based, object-oriented programing language and is meant to hold few implementation dependencies as possible. Java is general-purpose programing language made for developers to write down once run anywhere that's compiled with Java code can run on all OS that support Java. Java applications are compiled to byte code which will run on any Java Virtual Machine.

After the name OAK, the team decided to offer a replacement name there to and therefore the suggested words were Silk, Jolt, revolutionary, DNA, dynamic, etc. These all names were easy to spell and fun to mention , but all of them wanted the name to reflect the essence of technology. As per with James Gosling, Java the among the highest names alongside Silk, and since java was a singular name so most of them preferred it.

Java is that the name of an island in Indonesia where the primary coffee(named java coffee) was produced. And this name was selected by James Gosling whilehehaving coffee in his office. Note that Java is simply a reputation, not an acronym.

```
// Demo Java program
// Importing classes from packages
importjava.io.*;
// Main class
publicclassGFG {
    // Main driver method
    publicstaticvoidmain(String[] args)
    {
```

(c) Amity University Online

Notes

```
        // Print statement
        System.out.println("Welcome to the world of Java");
    }
}
```

Output

Welcome to the world of Java



1.2.1 History of Java, Java v/s C++.

Java was first introduced in 1995 as an easy and secure object-oriented programming language. It's a singular language therein, being a replacement language at that point; it had been ready to attract tons of interest from the computing community. Within two years after Java was launched, there have been an estimated 400,000 Java programmers and over 100 books on Java programming.

There are a couple of possible reasons for the exceptional interest in Java. The year 1995 saw a maturing of Web technologies, and Java's multiplatform capability, which enabled a Java program to execute on any computer, was exceedingly attractive, especially on an open network just like the Internet. Java is implemented via part compilation and subsequent execution on an interpreter implemented in software. Java applications are therefore code portable as long as a Java virtual machine is implemented for the target machine.

The popularity of Java is additionally ironically thanks to its similarity with its close rival C++. Java takes the pain out of learning a replacement language by reusing much of C and C++. At an equivalent time, safe programming practice in Java and language facilities for automatic memory management were benefits that were attractive to programmers on the verge of deserting their C/C++ camps. In reference to the web, Java applets have given rise to a replacement generation of distributed applications with low software distribution and maintenance costs. As applets are embedded in an HTML document via <APPLET> tags, its transmission to the client machine for execution is implicitly handled by the underlying network protocols and thus makes the standard channels of distribution and installation obsolete.

While the object-oriented programming framework promotes reusability of software and code, this very practice has been demonstrated within the rich set of sophistication libraries seen within the Java language. The Java foundation class libraries provide for windowing and graphical interface programming, network communications, and multimedia facilities. Together, they demonstrate the sensible and productive work wiped out Java.

(c) Amity University Online

Notes

Different between C++ and Java

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (/** ... */) to create documentation for java source code.

(c) Amity University Online

Notes

unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like>> operator.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.

2.1.2 JDK, JVM and Other tools for Java Application

Java Development Kit(JDK):

"JDK is a software development environment which is used for creating applets and Java applications. Java developers can use it on different operating systems like Windows, macOS, Solaris, and Linux. JDK helps them to code and executeJava programs. One can install more than one JDK version on the same computer."

Why use JDK?

Important reasons of using JDK:

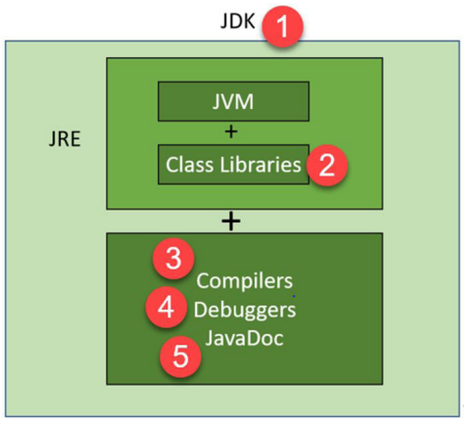
- JDK contains necessary tools to write Java programs, and JRE to execute them.
- It contains a compiler, Java application launcher, Appletviewer, etc.
- Compiler turn the code written in Java into byte code.
- Java application launcher opens a JRE, loads the essential class, and executes its main process.

(c) Amity University Online

Notes

Features of JDK

- It enables you to handle multiple extensions in a single catch block.
- JDK includes all features that JRE has.
- It contains development tools such as a compiler, debugger, etc.
- JDK provides the environment to develop and execute Java source code.
- It can be installed on Windows, UNIX, and Mac operating systems.
- Diamond operator can be used in specifying a generic type interface instead of writing the exact one.



JDK Functionality

Important component of JDK

- JDK and JRE: The JDK enables programmers to generate core Java programs that can be run by the JRE, which consist of JVM and class libraries.
- Class Libraries: It is a cluster of dynamically loadable libraries that Java program can call at run time.
- Compilers: It is a Java program which accepts text file of developers and compiles into Java class file. It is the widespread form of productivity given by compiler, which contains Java byte code. In Java, the primary compiler is JavaC.
- Debuggers: Debugger is a Java program that helps programmers to test and debug Java programs.
- JavaDoc: JavaDoc is documentation created by Sun Microsystems for the Java. JavaDoc can be used for creating API documentation in HTML file from the main program.

Java Virtual Machine(JVM)

JVM is an engine that provides a runtime environment to run the Java Code or applications. It transforms Java bytecode into machine language. Java Virtual Machine is a component of Java Run Environment (JRE). It cannot be independently downloaded or installed. In order to use JVM, you need to install JRE.

In other programming languages, the compiler creates machine code for a specific system. However, in Java language compiler produces code for a virtual machine which is called as JVM.

(c) Amity University Online

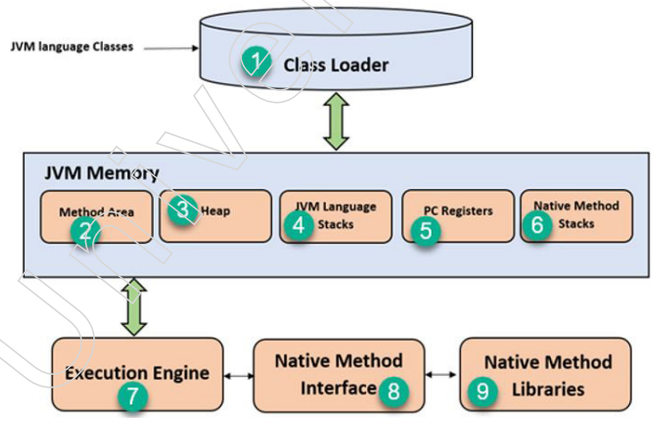
Notes

Reasons for Using JVM

- JVM gives a platform-independent way of executing Java source code.
- It has abundant of libraries, tools, and frameworks.
- Once you run Java program, you can execute on any platform and save lots of time.
- JVM comes with in-built JIT(Just-in-Time) compiler that converts Java source code into low-level machine language. Hence, it executes faster as a regular application.

Features of Java Virtual Machine

- It enables you to run applications in a cloud environment or in your device.
- Java Virtual Machine converts byte code to the machine-specific code.
- It provides basic java functions like memory management, security, garbage collection, and more.
- JVM runs the program by using libraries and files given by Java Runtime Environment.
- JDK and JRE both contain Java Virtual Machine.
- It can execute the java program line by line hence it is also called as interpreter.
- JVM is easily customizable for example; you can allocate minimum and maximum memory to it.
- It is independent from hardware and the operating system. So, you can write a java program once and run anywhere.



JVM Functionality

Components of Java Virtual Machine

1. Class Loader: The class loader is a subsystem which is used for loading class files. It performs three key functions those are Loading, Linking, and Initialization.
2. Method Area: Java Virtual Machine Method Area stores structure of class similar to metadata, the code for Java methods, and the constant runtime pool.
3. Heap: All the Arrays, Objects and instance variables are saved in a heap. Heap is shared across multiple threads.
4. JVM language Stacks: Java language Stacks store local variables, and its part results. All the thread has its own JVM language stack, created concurrently as the thread is created. A new frame is created when method is called, and it is removed when method invocation process is finished.

(c) Amity University Online

Notes

5. PC Registers: PC registers store the location of the Java virtual machine instruction, which is currently executing. In Java, each thread has its own PC register.
6. Native Method Stacks: Native method stacks hold the information of native code based on the native library. It provides memory on native heaps or uses any type of stack.
7. Execution Engine: It is used to test software, hardware, or complete systems. The test execution engine not at all carries any information about the tested product.
8. Native Method interface: The Native Method Interface is a programming structure. It allows Java code, which is compelling in a JVM to call by libraries and native applications.
9. Native Method Libraries: Native Libraries is a compilation of the Native Libraries (C, C++), which are required by the Execution Engine.

Java Runtime Environment (JRE)

JRE is a piece of a software which is planned to run other software. It holds the class libraries, loader class, and JVM. In simple terms, if you want to execute a Java program you need JRE. If you are not a developer, you don't need to install JDK, but just JRE to run Java programs. Even if, all JDK versions bundled with Java Runtime Environment, so you do not need to download and install the JRE separately in your computer.

Reasons of using JRE:

- JRE contains pre-defined class libraries, JVM, and other supporting files. It does not include any tool for Java development like a debugger, compiler, etc.
- It uses significant package classes likeswingetc, util, lang, awt, and runtime libraries.
- If you want to execute Java applets, then JRE must be installed in your system.

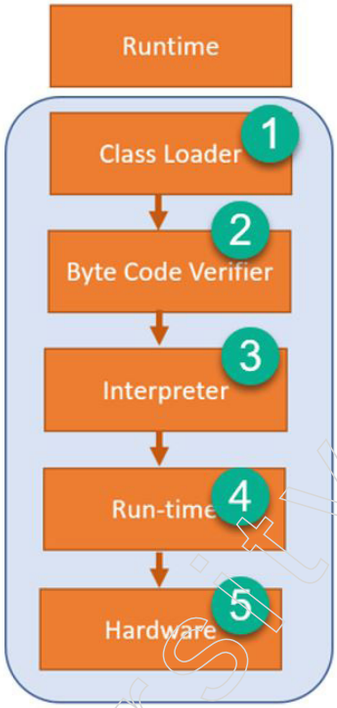
Features of JRE

- Java Runtime Environment is a set of tools using which the JVM actually runs.
- JRE contains deployment technology, including Java Web Start and Java Plug-in.
- Developers can easily run the source code in JRE, but he/she cannot write and compile the Java program.
- It includes integration libraries like Java Database Connectivity (JDBC), Remote Method Invocation (RMI), Java Naming and Directory Interface (JNDI), and more.
- JRE has JVM and Java HotSpot virtual machine client.

Functionality of JRE

(c) Amity University Online

Notes



Components of JRE

- Class loaders: The class loader loads various classes that are obligatory for running a Java program. JVM uses three class loaders defined as bootstrap class loader, extensions class loader, and system class loader.
- Byte code verifier: Byte code verifier checks the bytecode so that the code doesn't disturb the interpreter.
- Interpreter: Once the classes get loaded, and the code is checked, the interpreter reads the code line by line.
- Run-time: Run-time is a system used mostly in programming to explain time period during which a particular program is running.
- Hardware: Once you compile Java native code, it runs on a explicit hardware platform.

1.2.3 IDE for developing Java Application

IDE is defined as integrated development environment for programming in Java; many also provide functionality for other languages,IDEs typically provide a, a compiler or interpreter, editor and a debugger that the developer accesses through a combined graphical user interface (GUI). Java IDEs comprises of language-specific component such as Maven and Ant build tools and TestNG and JUnit testing.

(c) Amity University Online

Notes

Features of IDE

- A good Java IDE should highlight the syntax
- It should allow you to quickly go to the definition of a class or method
- It should support shortcuts for ease of access
- It must provide easy to use user interface
- A Java IDE must offer a real-time output of the Java program
- It should provide all the libraries for writing Java code
- A Java programming editor should automatically complete the code

Best Online Java Programming IDEs & Compilers: There are many IDE available in markets we have discuss 5 major IDE's

1. Eclipse: Eclipse is an IDE for writing and compiling Java programs. It highlights the syntax you have written. This tool enables you to easily debug the program.

Features of Eclipse:

- **It provides you readymade code template.**
- **This Java IDE for Windows automatically validates syntax.**
- **It supports code refactoring. (A process of improving internal code structure).**
- **Eclipse enables you to manage the project remotely.**
- **The software has a user-friendly interface with drag and drop facility.**
- **You can extend the functionality of Eclipse using plugins.**

2. BlueJ: BlueJ is a free Java IDE for beginners. It is one of the best editors for Java which enables a developer to scan code visually much faster. The tool offers many extensions to expand its functionality.

Features of BlueJ:

- It has a smaller, simple, and portable interface.
- It allows you to run Java objects while you run the program.
- You can use this tool without installing it.

3. IntelliJ IDEA: IntelliJ IDEA is a multi-purpose IDE which concentrate on Java development. It is one of the best Java compilers that offer superior help for the web, mobile, and also hybrid application expansion.

(c) Amity University Online

Notes

Features of IntelliJ IDEA

- It analyses source code between all project files and languages.
- This tool gives a list of the most relevant symbols applicable in the current context.
- It allows cross-language refactoring.
- This tool searches duplicate code fragments on the fly.
- Inspections and quick fixes.
- IntelliJ IDEA offers an editor-centric environment.
- It allows users to use static methods or constants easily.

4. Kite: Kite is IDE for Java that routinely completes multiple line codes. This editor accepts more than 16 languages. It helps you to code quickly with no problem.

Features of Kite:

- It offers Java documentation.
- This editor provides a function signature as you type.
- You will get a tooltip on mouse hover.
- Provides support in email.
- Uses machine learning models for Java language.

5. Apache NetBeans: Apache NetBeans is an integrated development environment for writing and compiling Java programs. It has a project window that shows a list of all projects at present exists.

Features of NetBeans:

- You can easily see the structure of the Java class.
- It automatically completes the brackets.
- This app has service windows that show currently available external services.
- This Linux Java IDE offers readymade templates for writing a script.
- You can format a piece of code using a keyboard shortcut.
- The tool highlights Java variables and keywords.
- Netbeans provides parameter hints by typing $ symbol.

Others popular IDE are as follows:

- MyEclipse
- Xcode
- jGRASP
- Codota

(c) Amity University Online

Notes

- Codenvy
- Slickedit
- JBoss Forge
- JDeveloper
- JEdit

1.2.4 Constructs used in JAVA :Variables, types and type declarations

The variable is the necessary unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable

In Java, all variables must be defined upfront so that they can be used later. A variable declaration is shown here:

data type identifier [= value];

Int Age = 20;

Type is the data type Int in above example. The identifier Age is the name of the variable. You can initialize the variable by specifying an equal sign and a value 20 as declared in above statement. We need to make sure that the initialization expression must result in a value of the compatible or same type as that specified for the variable.

To declare more than one variable of the particular type, use a comma to separate them.

int x, y, z; // declares three ints, x, y, and z.

int m = 3, n, o = 5; // declares three more ints, initializing

// m and o.

byte z = 22; // initializes z.

double pi = 3.14159; // declares an approximation of pi.

char a = 'a'; // the variable a has the value 'a'.

There are three different types of variable:

1. Local Variables
2. Instance Variables
3. Static Variables

1. Local Variables:A local variable is a variable defined inside a method body, block or constructor. It means variable is only accessible within the method, block or constructor that declared it.

Example of Local variable:

int area()

{

(c) Amity University Online

Notes

```
int length=10; //local variable
int breadth = 5; //local variable
int Rarea = length*breadth; //local variable
return Rarea;
}
```

Important Points to Remember:

- Local variables are declared in a blocks, methods or constructors.
- Local variables are created when the block, method or constructor is started and the variable will be destroyed once it exits the block, method or constructor.
- Access modifiers cannot be used for declaring local variables.
- Local variables are implemented at stack level internally.
- Default values are not assigned to local variables in Java.
- Variables can also be define inside statement blocks i.e. do, for, while loop. These are only visible inside that block.

Program Example of Local Variable

```
public class Ravi


{
   public void AgeSet()
   {
      int age = 0; //local variable
      age = age + 6;
System.out.println("Ravi age is : " + age);
   }
   public static void main(String args[]){
      Ravi d = new Ravi();
d.AgeSet();
   }
}
```

Output

Ravi age is : 6

Here, age is a local variable. This variable is defined under AgeSet() method and its scope is limited to this method only:

2. Instance Variable: Instance variables are non-static variables and are defined in a class outside any method, constructor or block.

(c) Amity University Online

Notes

Important Points to Remember

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
- Initialization of Instance Variable is not mandatory. Its default value is 0
- Instance Variable can be accessed only by creating objects

Program for explainingInstance Variable

```
public class Record{
    public String address;// this instance variable is visible for any child class.
     private int phonenumber;// this instance age variable is visible in Record class only.
     public Record (String Recaddress)
     {
        address = Recadre;
     }
     public void setphonenumber(int RecSal)
     {
phonenumber = RecSal;
     }
     public void printRec()
     {
System.out.println("address : " + address ); // print the value for "address"
System.out.println("phonenumber :" + phonenumber); //prints the value for "phonenumber"
     }
     public static void main(String args[])
     {
       Record r = new Record("1, Zone West");
r.setphonenumber(123456);
r.printRec();
     }
```

(c) Amity University Online

Notes

```
}
```

Output:

Adress : 1, Zone West

phonenumber :123456

In above, we take two instance variables one is address variable of string type and can be accessed by any child class because it is public and the second is Phonenumber variable of integer type and can be accessed in the same class record because it is private.

3. Static variables:These variables also known as Class variables. These variables are defined similarly as instance variable; the difference is that static variables are defined using the static keyword inside a class outside any method constructor or block.

The below program has a class Patient in which we use static variable Hospital. This variable is common to all Patients so we make it static. The Patient information will be displayed with different Patient no and names but same Hospital. The advantage to make Hospital variable static is that it save memory as it loads once in a class area at class loading time:

```
class Patient
{
   int rollno;
   String name;
   static String Hospital =”GGGI”; //Static Variable gets memory once
   /*Constructor of Patient class*/
Patient(int r,String n)
   {
Patientno= r;
      name = n;
   }
   /*Method For Displaying Patient Details*/
   void display()
   {
System.out.println(patientno+” “+name+” “+Hospital); // print the value of roll no, name and Hospital
   }
```

(c) Amity University Online

Notes

```
    public static void main(String args[])
    {
Patient s1 = new Patient(101,"Ravi");
Patient s2 = new Patient(102,"Ram");
        s1.display();  // call the display function using the s1 object
        s2.display();   // call the display function using the s2 object
    }
}
```

Output:

101 Ravi GGGI

102 Ram GGGI

Important Points to Remember:

- Static variables are also known as Class variables which are declared with the "static" keyword in a class.
- A single copy of each variable per class is to be shared by all instances of the class.
- Static variables are stored in static memory. Static variables are rarely used other than it is declared final and used as either public or private constants.
- When the program starts static variables are created and when the program stops, static variables are destroyed.
- Default values of static variables are same as instance variables. For numbers the default value is 0, for Boolean the default value is false and for object references the default value is null.
- When static or class variables are declared as public static final, then variables names are all in upper case. The naming syntax is the same as instance and local variables in case of static variables are not declared public and final.
- Static variables can be accessed by calling with the class name. It does not need any object.
- Static variables are initialized only one time, i.e. at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.

1.2.5 Data types, Functions

It is imperative to state at the outset that Java may be a powerfully typed language. Indeed, part of Java's safety and robustness comes from this fact.

- Every variable features a type, every expression features a type, and each type is strictly defined.
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility

Java has eight different primitive types of data: byte, short, int, long, char, float, double, and boolean. These all 8 types of data can be clubbed in four groups:

1. Integers: This group includes byte, short, int, and long, those are for whole-valued signed numbers.

(c) Amity University Online

Notes

2. Floating-point numbers: This group has float and double, which symbolize numbers with fractional precision.
3. Characters: This group includes char, which symbolize symbols in a character set, like letters and numbers.
4. Boolean:This group has boolean, which is a special type on behalf of true/false values.

The primitive types represent single values—not complex objects. Although Java isotherwise completely object-oriented, the primitive types aren't. They're analogous tothe simple types found in most other non–object-oriented languages. The rationale for this is oftenefficiency. Making the primitive types into objects would have degraded performance an excessive amount of.

Integers:

Java consists of four integer types: byte, short, int, and long. Every one of those is signed, positive and negative values. Java doesn't support unsigned, positive-only integers. There are other computer languages support both signed and unsigned integers. On the other hand, Java's developer felt that unsigned integers were unnecessary. Purposely, they felt that the perception of unsigned was used mostly to identify the behavior of the high-order bit, which shows the sign of an integer value.

The width of an integer type shouldn't be consideration of because the amount of storage it consumes,but rather because the behavior it defines for variables and expressions of that type. The Java run-timeenvironment is liberal to use whatever sizes it wants, as long because the types behave as you declaredthem. The width and ranges of those integer types vary widely, as shown during this table:

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

Things to Remember

1. **byte:** The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the byte keyword.
2. **short:** short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type.
3. **int:** The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case. The reason is that when byte and short values are used in an expression they are promoted to int when the expression is evaluated. Hence, int is often the best choice when an integer is needed.
4. **long:** long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

(c) Amity University Online

Notes

```
// Calculate the distance of light travels using long variables.
class Light {
public static void main(String args[]) {
 int l;
long d;
long s;
long d;
// speed of light in miles per secondapproximately
l = 186000;
d = 1000; // specify number of days here
s = days * 24 * 60 * 60; // convert to seconds
d = l * s; // compute distance
System.out.print(“In “ + days);
System.out.print(“ days light will travel about “);
System.out.println(distance + “ miles.”);
}
}
```

This program generates the following output:

In 1000 days light will travel about 16070400000000 miles.

Clearly, the output cannot have been held in an int variable so we need an other variable to hold this value.

Floating-Point Types

As we see in previous example that large value cannot be hold be in integer types for this we required another data type called Floating point number.

Floating-point numbers, also acknowledged as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as cube root, or transcendental such as sine cosine, tan result in a value whose accuracy needs a floating-point type.

There are two kinds of floating-point types,

- float
- double

Name	Width In Bits	Approximate Range
double	64	4.9e-324 to 1.8e + 308
float	32	1.4e-045 to 3.4e + 038

(c) Amity University Online

Notes

Things to Remember

- **Float:** The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision. For example, float can be useful when representing dollars and cents.
- **double:** Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), tan() and Cqurt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

```
// Compute the area of a triangle.
class Area {
public static void main(String args[]) {
 double b, h ;
 b = 2.5; // base of triangle
 h = 3.5; // height of triangle
 a = .5 * b * h; // compute area
System.out.println("Area of triangle is " + a);
}
}
```

Characters

In Java, the data type is being used to store characters is char. Java uses Unicode to represent characters. Unicode defines a totally international list which will represent all of the characters found altogether human languages.

It's a unification of many character sets, like Greek, Arabic, Hebrew, Katakana, Hangul, and lots of other sets of character. For this purpose, it requires 16 bits. Thus, in Java char may be a 16-bit type. Thecharrange is 0 to 65,536. There are not any negative chars. the quality set of characters referred to as ASCII still ranges from 0 to 127 as always, and therefore the extended 8-bit list , ISO-Latin-1, ranges from 0 to 255.

Program that demonstrates char variables:

```
// Demo of char data type.
class DemoChar {
public static void main(String args[]) {
 char ca1, ca2;
 ca1 = 15; // code for A
 ca2 = 'B';
```

(c) Amity University Online

Notes

```
System.out.print("ca1 and ca2: ");
System.out.println(ca1 + " " + ca2);
}
}
```

Output:

ca1 and ca2: AB

In above program you notice that ca1 is assigned the value 15, which is the ASCII (and Unicode) value that corresponds to the letter A. As described, the ASCII character set occupies the first 127 values in the Unicode character set.

```
// char variables like integers.
class DemoChar2 {
public static void main(String args[]) {
char cha;
ca1 = 'A';
System.out.println("ca1 contains " + ca1);
ca1++; // increment ca1
System.out.println("ca1 is now " + ca1);
}
}
```

Output

ca1 contains A

ca1 is now B

In the program, ca1 is first given the value X. Next, ca1 is incremented. This results in ca1 containing B, the next character in the ASCII (and Unicode) sequence.

Booleans

Booleans used for logical values. It can have only had two possible values, either true or false. This is the type returned by all relational operators, as in the case of x < y. Boolean is also the kind required by the conditional expressions that administer the control statements such as if and for.

Program that defines the Boolean type:

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[]) {
booleana;
```

(c) Amity University Online

Notes

```
a = false;
System.out.println(“a is “ + a);
a = true;
System.out.println(“a is “ + a);
// boolean value control the if statement
if(b) System.out.println(“This is executed.”);
b = false;
if(b) System.out.println(“This is not executed.”);
// Result of a relational operator is always aboolean value
System.out.println(“21 > 19 is “ + (21>19));
}
}
```

Output

```
a is false
a is true
This is executed.
21>19 is true
```

Check your Understanding

1. Java language was developed _____ by at Sun Microsystems in _____ .
2. _____ is an engine that provides a runtime environment to run the Java Code or applications.
3. _____ variables also known as Class variables
4. _____ can have only had two possible values, either true or false.
5. The data type is being used to store characters is _____ .

Summary

- Java language was developed by James Gosling at Sun Microsystems in 1991, later acquired by Oracle Corporation. It’s an easy programming language. Java helps in writing codes and also makes compiling, and debugging programming easy. It helps programmer to make reusable code and modular programs.
- JDK is a software development environment which is used for creating applets and Java applications. Java developers can use it on different operating systems like Windows, macOS, Solaris, and Linux.

Important component of JDK

(c) Amity University Online

Notes

- JDK and JRE: The JDK enables programmers to generate core Java programs that can be run by the JRE, which consist of JVM and class libraries.
- Class Libraries: It is a cluster of dynamically loadable libraries that Java program can call at run time.
- Compilers: It is a Java program which accepts text file of developers and compiles into Java class file. It is the widespread form of productivity given by compiler, which contains Java byte code. In Java, the primary compiler is JavaC.
- Debuggers: Debugger is a Java program that helps programmers to test and debug Java programs.
- Java Doc: Java Doc is documentation created by Sun Microsystems for the Java. JavaDoc can be used for creating API documentation in HTML file from the main program.
- The variable is the necessary unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Activity

1. Create a program to distribute student in a class based on their percentage using nested if?

Questions and exercises

1. What is the difference between java and C++?
2. Define various tools for java application? JDK and JVM
3. What is the importance of IDE for developing Java Application?
4. Define various types of variables and declaration?
5. Define the data types used in Java application?
6. Define the function used in Java application?

Glossary

- JDK: Java Development Kit
- JRE: Java Run Environment
- JVM: Java virtual Machine
- IDE: integrated development environment
- Java.util: Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Further Readings:

1. Programmer's Guide to Java SE 8 Oracle Certified Associate (OCA), Mughal/ Rasmussen

(c) Amity University Online

Notes

2. Deploy, Manage and Update Java Runtime Environment in the Enterprise with System Center Configuration, Trevor Jones
3. Java Fundamentals, Herbert Schildt (Author), Dale Skrien (Author)
4. Programming in Java, Sachin Malhotra, Saurabh Choudhary

Check Your Understanding:

Answers

1. James Gosling, 1991
2. JVM
3. Static variable
4. Boolean
5. char

(c) Amity University Online

Unit - 1.3 : Statements & Operators in Java

Notes

Objectives

At the end of this unit, you will be able to understand:

- Uses of Increment and Decrement Operators
- What are Relational and Logical operators
- When we can use Conditional Expressions, if then else clause
- Input using scanner class and output statement
- What is Loops and Switch case
- What are Arrays and Methods

Introduction

Statements are similar to sentences in the English language. A sentence forms a complete idea which can include one or more clauses. Likewise, a statement in Java forms a complete command to be executed and can include one or more expressions.

In simpler terms, a Java statement is just an instruction that explains what should happen.

Types of Java Statements

Java supports three different types of statements:

- Expression statements change values of variables, call methods, and create objects.
- Declaration statements declare variables.
- Control-flow statements determine the order that statements are executed. Typically, Java statements parse from the top to the bottom of the program. However, with control-flow statements, that order can be interrupted to implement branching or looping so that the Java program can run particular sections of code based on certain conditions.

1.3.1 Increment and Decrement Operators

Increment Operator: The increment operator is used when a developer needs to increment the value of a variable in an expression. In the Pre-Increment, value is incremented first and then used inside the expression. But in the Post-Increment, value is first used inside the expression and then it's getincremented.

We used two different prefix while using increment operator

a = 42;

b = ++a;

In this case, b is set to 43 as you would expect, because the increment occurs before a is assigned

(c) Amity University Online

Notes

to b. Thus, the line b = ++a; is the equivalent of these two statements:

a= a + 1;

b = a;

However, when written like this,

a = 42;

b = x++;

The value of a is obtained before the increment operator is executed, so the value of b is 42.

Of course, in both cases a is set to 43. Here, the line b = a++; is the equivalent of these two

statements:

b =a;

a = a + 1;

Example for increment operator

```
// Demonstrate ++ Operator.
class IncDec {
public static void main(String args[]) {
int x = 5;
int y = 7;
int z;
int u;
z = ++y;
u = x++;
z++;
System.out.println(“x = “ + x);
System.out.println(“y = “ + y);
System.out.println(“u = “ + z);
System.out.println(“z = “ + u);
}
}
```

Output:

x = 6

y = 8

(c) Amity University Online

Notes

z = 9

u = 5

Decrement Operators: The decrement operator is used when a developer want to decrement the value of a variable in an expression. In the Pre-Decrement, value is decremented first and then used inside the expression. Whereas in the Post-Decrement, value is first used inside the expression and then it's get decremented.

Decrement operator can be repressed as

'a- -'which is equal to

a=a-1;

```
// Demonstrate -- Operator.
class IncDec {
public static void main(String args[]) {
int x = 5;
int y = 7;
int z;
int u;
z = --y;
u = x--;
z --;
System.out.println("x = " + x);
System.out.println("y = " + y);
System.out.println("u = " + z);
System.out.println("z = " + u);
}
}
```

Output:

x = 4

y = 7

z = 6

u = 5

(c) Amity University Online

Notes

Difference between increment and decrement Operator

Increment Operators	Decrement Operators
Increment Operator adds 1 to the operand.	Decrement Operator substrats 1 from the operand.
Postfix increment operator means the expression is evaluated first using the original value of the variable and then the variable is incremented (increased)	Postfix decrement operator means the expression is evaluated first using the original value of the variable and then the variable is decremented (decreased)
Prefix increment operator means the variable is incremented first and then the expression is evaluated using the new value of the variable.	Prefix decrement operator means the variable is decremented first and then the expression is evaluted using the new value of the variable.
Generally, we use this in decision making and looping.	This is also used in decision making and looping.

1.3.2 Relational and Logical operators

Relational Operator: The relational operators determine the relationship that one variable has with the other variable. Explicitly, they establish equality and ordering.

Types of relational Operator:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of using any of these operators is a boolean value. These relational operators are often used in the expressions that control the if statement and the various loop statements.

Program using Relational Operator

```
public class RelationalOperator {
public static void main(String args[]) {
    int x = 10;
    int y = 20;
System.out.println("x == y = " + (x == y) );
System.out.println("x != y = " + (x != y) );
System.out.println("x > y = " + (x > y) );
System.out.println("x < y = " + (x < y) );
System.out.println("y>= x = " + (y>= x) );
```

(c) Amity University Online

Notes

```
System.out.println("y<= x = " + (y<= x) );
  }
}
```

Output:

x == y = false

x != y = true

x > y = false

x < y = true

y>= x = true

y<= x = false

Boolean Logical Operators: The Boolean logical operators shown here operate only on boolean values. All of the binary logical operators combine two boolean values to form a resulting boolean value.

Operator	Result
&	Logical AND
I	Logical OR
^	Logical XOR (exclusive OR)
II	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
I=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state: !true == false and !false == true. The following table shows the effect of each logical operation:

A	B	A I B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Program Using Boolean Operator:

```
class BoolLogic {
public static void main(String args[]) {
```


(c) Amity University Online

Notes

```
boolean x = true;
boolean y = false;
boolean z = a | b;
boolean s = a &b;
boolean t = a ^ b;
boolean u = (!a& b) | (a & !b);
boolean v= !a;
System.out.println(“ x = “ + x);
System.out.println(“ y = “ + y);
System.out.println(“ x|y = “ + z);
System.out.println(“ x&y = “ + s);
System.out.println(“ x^y = “ + t);
System.out.println(“!x&y|x&!y = “ + u);
System.out.println(“ !x = “ + v);
}
}
```

Output

```
x = true
y = false
x|y = true
x&y = false
x^y = true
x&y|x&!y = true
!x = false
```

1.3.3 Conditional Expressions, if then else clause

Java uses two selection statements: if and switch. These statements helps developer to control the run of developer’s program execution based upon conditions used in program during run time.

If Statement:

The if statement is Java’s conditional branch declaration. It can be used to set the course of program execution through two different paths. General form of the if statement:

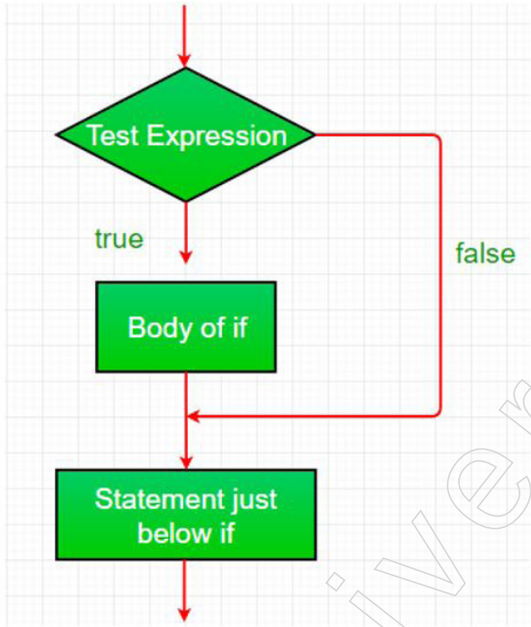
(c) Amity University Online

Notes

```
if (condition) statement1;
else statement2;
```

The Java if statement is the simplest decision-making statement. It is used to decide whether a certain statement will be executed or not that is if a certain condition is true then a statement is executed otherwise not.

Flow of information in if statement:



Example of if:

```
class IfDemo {
   public static void main(String args[])
   {
      int a = 10;

      if (a< 15)
System.out.println("10 is less than 15");

System.out.println("Outside if-block");
   }
}
```

Output

10 is less than 15

Outside if-block

© Amity University Online

Notes

Nested ifs A nested if is an if expression that is the target of another if or else. Nested ifs are very general in programming. When you nest ifs, the main thing to remember is that an else statement at all times targeted to the nearest if statement that is within the same block as the else and that is not associated with an else.

```
if(y == 10) {
if(x < 20) m = n;
if(z > 100) n = o; // this if is
else m = n; // associated with this else
}
else m = o; // this else refers to if(y == 10)
```

As the comments indicate, the final else is not associated with if(x<20) because it is notin the same block (even though it is the nearest if without an else). Rather, the final elseis associated with if(y==10). The inner else refers to if(z>100) because it is the closest ifwithin the same block.

The if-else-if Ladder: A frequent programming construct that is based on a sequence of nested ifs is the if-else-if ladder. It syntax look like this:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
```

The if statements are run from the top to down. If any one of the conditions controlling the if is right or true, the statement attached to with that if is executed, and the remaining of the ladder is bypassed. If not a single conditions is true, then the last else statement will be executed.

Example of If-else-is statement

```
class IfElse {
public static void main(String args[]) {
int month = 10; // October
```

(c) Amity University Online

Notes

```
String season;
if(month == 12 || month == 1 || month == 2)
season = “Winter”;
else if(month == 3 || month == 4 || month == 5)
season = “Spring”;
else if(month == 6 || month == 7 || month == 8)
season = “Summer”;
else if(month == 9 || month == 10 || month == 11)
season = “Autumn”;
else
season = “Bogus Month”;
System.out.println(“October is in the “ + season + “.”);
}
}
```

Output

October is in the Autumn.

If month value is 13 then we get the last else statement i.e Bogus Month

1.3.4 Input using scanner class and output statement

Scanner is a predefined class inside java.util package used to get the input of the primitive types like float, double, etc. and strings from the user. It is the easiest way to accept input from a user in a Java program; it is not a very efficient way if developers want an input method for scenarios where time is a constraint like in competitive programming.

Important point to remember

- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
- To read strings, we use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

```
import java.util.Scanner;
public class ScannerDemo1
{
```

Amity University Online

Notes

```
    public static void main(String[] args)
    {
        // predefined standard input object
        Scanner sc = new Scanner(System.in);
        // String input
        String name = sc.nextLine();
        // Character input
        char gender = sc.next().charAt(0);
        int age = sc.nextInt();
        long mobileNo = sc.nextLong();
        double cgpa = sc.nextDouble();
        // Print the values if the input was correctly received.
        System.out.println("Name: "+name);
        System.out.println("Gender: "+gender);
        System.out.println("Age: "+age);
        System.out.println("Mobile Number: "+mobileNo);
        System.out.println("Percentage: "+percentage);
    }
}
```

Input by the user:

Rituj

M

31

1234567890

78

Output From the program:

Name: Rituj

Gender: M

Age: 31

Mobile Number: 1234567890

Percentage: 78

Sometimes, we have to see if the next value we scan is of a certain type or if the

(c) Amity University Online

Notes

input has ended.

Then, we check if the scanner's input is of the type we want with the help of has NextABC() functions where ABC is the type we are concerned with. The function returns true if the scanner has a symbol of that type, otherwise false. For example, in the below code, we have used has NextFloat(). To check if string is avilable, we use hasNextLine(). likewise, to check if a single character is avilable, we use has Next(). charAt(0).

Let us look at the code to take input some numbers from console and print their average.

```
import java.util.Scanner;
public class ScannerDemo2
{
    public static void main(String[] args)
    {
            Scanner sc = new Scanner(System.in);
            float addition = 0, count = 0;
            // Check if float value is available
            while (sc.hasNextInt())
            {
                    float num = sc.nextInt();
                    addition += num;
                    count++;
            }
            float average = addition / count;
            System.out.println("Avaerage: " + average);
    }
}
```

Input from User

```
15
18
16
25
25.6
18.9
```

(c) Amity University Online

Notes

22.5

Output

Average: 20.142857142

1.3.5 Loops,Switch case

Switch Case: The switch statement is Java's multiway branch statement. It gives a simplest way to dispatch execution to different parts of the developer code based on the value of an expression. As such, it regularly provides a better option than a large series of if-else-if statements. Syntax for the switch statement:

```
switch (expression) {
case value1:
break;
case value2:
break;
.
.
.
case valueN:
break;
default:
}
```

The expression must be a data type like float, double, int: each of the values given in the case statements must be of a type compatible with the statement.

The switch statement perform like this: The value of the expression is checked with each of the literal values in the case statements. If the case statement true, the code following that case is executed. If none of the cases matches the value of the statement, then the default statement is executed.

```
class SampleSwitch {
public static void main(String args[]) {
for(int a=0; a<5; a++)
switch(a) {
case 0:
System.out.println("a is zero.");
break;
case 1:
```

(c) Amity University Online

Notes

```
System.out.println("a is one.");
break;
case 2:
System.out.println("a is two.");
break;
case 3:
System.out.println("a is three.");
break;
default:
System.out.println("a is greater than 3.");
}
}
}
```

Output

```
a is zero.
a is one.
a is two.
a is three.
a is greater than 3.
```

As you can see in above program, each time through the loop, the statements associated with the case constant that matches a are executed. All others are bypassed. After a is greater than 3, no case statements match, so the default statement is executed.

Loops: We have three types of loop in java

- while
- do-while
- for

1. while loop: The while loop is Java's most basic loop declaration. It repeats a statement or block till its controlling statement is true.

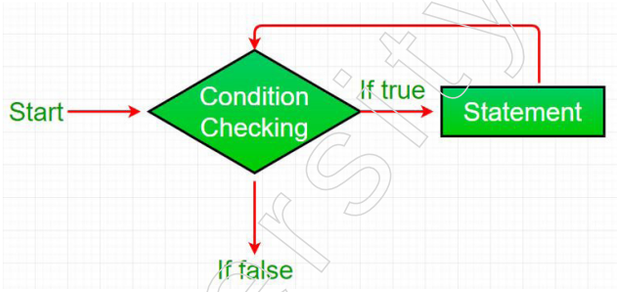
(c) Amity University Online

Notes

How while loop executes

1. Control falls into the while loop.
2. The flow jumps to Condition
3. Condition is tested.
   a. If Condition states true, the flow goes into the Body.
   b. If Condition states false, the flow goes outside the loop
4. The expression within the body of the loop gets executed.
5. Updating takes place.
6. Control flows back to Step 2.
7. The do-while loop has ended and the flow has gone outside.

Flow Chart for while loop



Java program to see how while loop works:

```
class whileLoopDemo {
    public static void main(String args[])
    {
            int a = 1;
                while (a< 6) {
                System.out.println("Hello Java");
                                    a++;
            }
    }
}
```

Output is:

```
Hello Java
Hello Java
Hello Java
Hello Java
Hello Java
```

(c) Amity University Online

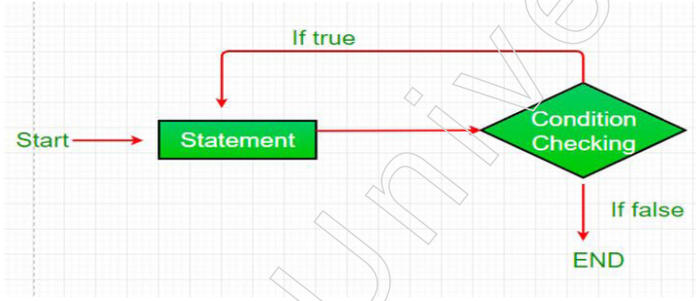
Notes

2. Do while: If the conditional expression controlling a while loop is false at starting, then the body of the loop will not be executed at all. However, sometimes developers want to execute the body of a loop at least once, even if the conditional expression is false at the starting. There are times when a developer wants to like to test the termination expression at the end of the loop rather than at the starting. Luckily, Java supplies a loop that does just that what a developer wants: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the end of the loop.

How do while loop executes

- Control falls into the do-while loop.
- The statements inside the body of the loop get executed.
- Updation takes place.
- The flow jumps to Condition
- Condition is tested.
- If Condition yields true, goto Step 6.
- If Condition yields false, the flow goes outside the loop
- Flow goes back to Step 2.

Flow of a do while loop



Java program to see how do while loop works:

```
class dowhileloopDemo {
    public static void main(String args[])
    {
            int a = 7;
            do {
                    System.out.println("Hello Java");

                    // update expression
                    a++;
            }
            while (a < 6);
```

(c) Amity University Online

Notes

```
    }
}
```

Output

Hello Java

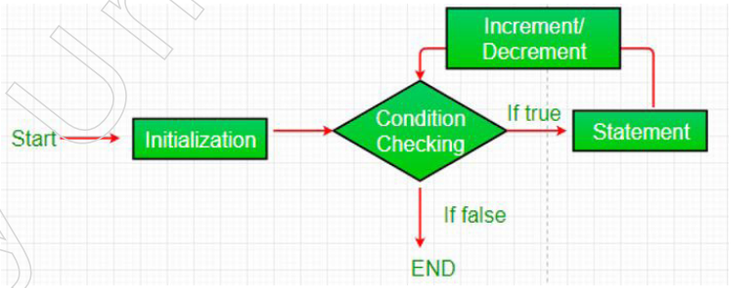
Here we see the first condition was false but the statement is printed once because the condition statement at the end.

3. For loop: Java provides a short way of writing the loop structure. The for statement contains the initialization, condition statement and increment/decrement operator in one line thus providing a shorter and simplest way to debug structure of looping.

How a for loop executes:

1. Control falls into the for loop. Initialization is done
2. The flow jumps to Condition
3. Condition is tested.
   a. If Condition yields true, the flow goes into the Body
   b. If Condition yields false, the flow goes outside the loop
4. The statements inside the body of the loop get executed.
5. The flow goes to the Updation
6. Updation takes place and the flow goes to Step 3 again
7. The for loop has ended and the flow has gone outside.

Flow of for loop



Java program to see how for loop works:

```
class forLoopDemo {
    public static void main(String args[])
    {
        int sum = 0;

        for (int a = 1; a <= 20; a++) {
            sum = sum + a;
```

(c) Amity University Online

Notes

```
            }
            System.out.println("Sum: " + sum);
        }
    }
```

Output

Sum: 210

1.3.6 Arrays, Methods

Arrays: An array is a group of like-typed variables or data types that are referred to by a common name. Arrays in Java work differently as compared C/C++.

Important points to Remember

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each has an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The size of an array must be specified by an int or short value and not long.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Array comprises of different data types (int, float, etc.) as well as object (or non-primitive data types) references of a class depending on the definition of the array. In data types, the real values are stored in nearby memory locations.

One-Dimensional Arrays:To define an array we need two components: the type and the name. Type defines the element type of the array. The element type determines the data type of each element that contains the array.

```
class GFG
{
    public static void main (String[] args)
    {
    int[] arr;
    arr = new int[5];
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
```

(c) Amity University Online

Notes

```
        arr[4] = 50;
        for (int xy = 0; xy<arr.length; xy++)
                System.out.println("array contains " + a +
                                                        " : "+ arr[a]);
        }
    }
```

Output:

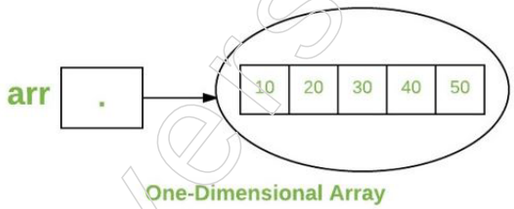
Array contains0 : 10

Array contains1 : 20

Array contains 2 : 30

Array contains3 : 40

Array contains4 : 50



One-Dimensional Array

Creating Array of objects:

```
class Student
{
    public int roll_no;
    public String name;
    Student(int roll_no, String name)
    {
        this.roll_no = roll_no;
        this.name = name;
    }
}
public class abc
{
    public static void main (String[] args)
    {
        Student[] arr;
```

(c) Amity University Online

Notes

```
            arr = new Student[5];
            arr[0] = new Student(1,"Nandika");
arr[1] = new Student(2,"Ravi");
            arr[2] = new Student(3,"shikar");
            arr[3] = new Student(4,"rituj");
            arr[4] = new Student(5,"mohit");
            for (int xy = 0; xy<arr.length; xy++)
            System.out.println("Element at " + a + " : " +
                                                arr[xy].roll_no +" "+ arr[xy].name);
    }
}
```

Output

Element at 0 : 1 Nandika

Element at 1 : 2 Ravi

Element at 2 : 3 shikar

Element at 3 : 4 Rituj

Element at 4 : 5 mohit

Multidimensional Arrays:Multidimensional arrays are arrays of arrays with each of element of the array hold the reference of another array.

```
class multiDimensional
{
    public static void main(String args[])
    {
            int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };
            for (int xy=0; xy<3 ;xy++)
            {
                    for (int yz=0; yz<3 ;yz++)
                            System.out.print(arr[xy][yz] + " ");
                    System.out.println();
            }
    }
}
```

(c) Amity University Online

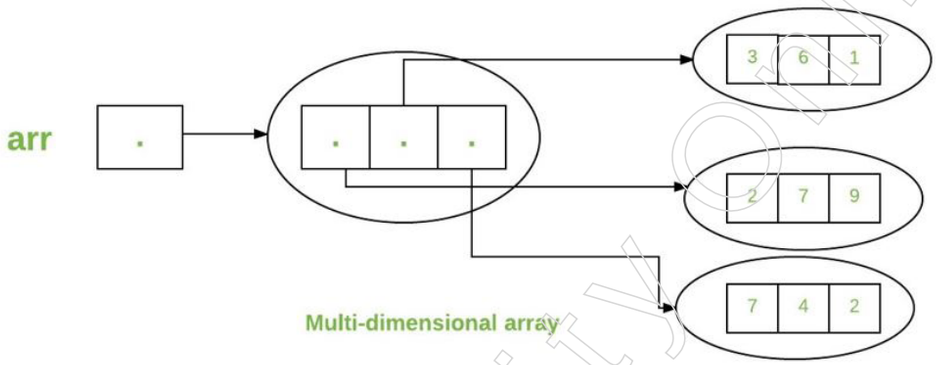
Notes

Output:

2 7 9

3 6 1

7 4 2



Check your Understanding

1. _____ is used when a developer needs to increment the value of a variable in an expression.
2. _____ determine the relationship that one variable has with the other variable.
3. Scanner is a predefined class inside _____ package
4. _____ arrays are arrays of arrays with each of element of the array hold the reference of another array.
5. The _____ is used when a developer want to decrement the value of a variable in an expression.

Summary

- The increment operator is used when a developer needs to increment the value of a variable in an expression. In the Pre-Increment, value is incremented first and then used inside the expression. But in the Post-Increment, value is first used inside the expression and then it's get incremented.
- The increment and decrement operators are unary operators that add or subtract one to or from their operand. In most cases, they are implemented in imperative programming languages. C-like languages have two versions of each operator (pre- and post-) with slightly different semantics.
- The switch statement is Java's multi way branch statement. It gives a simplest way to dispatch execution to different parts of the developer code based on the value of an expression.
- The increment and decrement operators are unary operators that add or subtract one to or from their operand, respectively. They are typically implemented in imperative programming languages. Each operator in C-like languages has two variants (pre- and post-) with slightly different semantics.

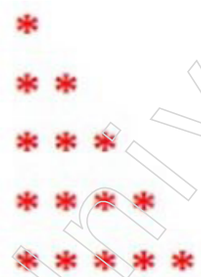
(c) Amity University Online

Notes

- An array in Java is a group of like-typed variables referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.
- In Java, all arrays are dynamically allocated. (discussed below)
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning from 0.
- Java array can be also be used as a static field, a local variable, or a method parameter.
- The size of an array must be specified by int or short value and not long.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Activity

1. Create a below design using loops?



Questions and exercises

1. Define Increment and Decrement Operators?
2. What do you understand from Relational and Logical operators?
3. Define Conditional Expressions, if then else clause?
4. Define scanner class?
5. Define array and methods?

Glossary

Operator	Result
&	Logical AND
I	Logical OR
^	Logical XOR (exclusive OR)
II	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment

(c) Amity University Online

Notes

|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

- **Java.util:** Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Further Readings:

1. Programmer's Guide to Java SE 8 Oracle Certified Associate (OCA), Mughal/ Rasmussen
2. Deploy, Manage and Update Java Runtime Environment in the Enterprise with System Center Configuration, Trevor Jones
3. Java Fundamentals, Herbert Schildt (Author), Dale Skrien (Author)
4. Programming in Java, Sachin Malhotra, Saurabh Choudhary

Check Your Understanding:

Answers

1. increment operator
2. Relational operators
3. java.util
4. Multidimensional
5. decrement operator

(c) Amity University Online

Unit - 1.4 : Inheritance in Java

Notes

Objectives

At the end of this unit, you will be able to understand:

- What are the different type of inheritance used in Java
- Define the concept behind overriding
- What are abstract classes
- What are final classes

Introduction

Inheritance in Java is a instrument in which one object acquires all the properties and characterstics of a parent object. It is an essential part of OOPs (Object Oriented programming system).

The thought at the back of mind is that, inheritance in Java is that developer can create new classes that are built upon existing classes. When you inherit from an accessible class, developer can reuse methods and fields of the parent class. Moreover, a developer can add new methods and fields in his/her current class as well.

Important points to remember:

- **Super Class:** The class whose features are inherited is known as superclass (or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as a subclass (or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
- **Default superclass:** Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritances with classes. Although with interfaces, multiple inheritances are supported by java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

1.4.1 Type of Inheritance with code snippets

There five type of inheritance used in java:

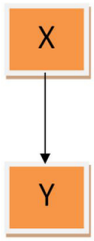
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance

(c) Amity University Online

Notes

4. Multiple Inheritance (Through Interfaces
5. Hybrid Inheritance(Through Interfaces)

1. Single Inheritance: “In single inheritance, subclasses inherit the features of one superclass.In the image below, class X serves as a base class for the derived class Y.”



Single Inheritance

Example of Single Inheritance:

```
// single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;
class single {
    public void print_welocome()
    {
            System.out.println(“Welcome”);
    }
}
class two extends one {
    public void print_to() { System.out.println(“to”); }
}
// Driver class
public class Main {
    public static void main(String[] args)
    {
            two g = new two();
            g.print_welcome();
            g.print_to();
            g.print_java();
```

(c) Amity University Online

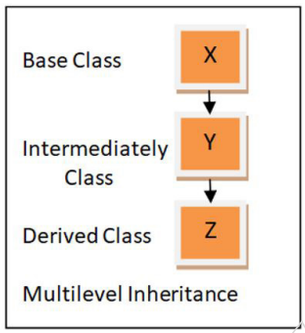
Notes

```
    }
}
```

Output

```
Welcome
to
java
```

2. Multilevel Inheritance: “In Multilevel Inheritance, a resulting class will be inheriting a base class as well as the derived class also act as the base class to other class. In the below image, class X serves as a base class for the derived class Y, which in turn serves as a base class for the derived class Z.”



Example of Multilevel Inheritance:

```
// Java program to understand single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;
class X {
    public void print_Welcome()
    {
            System.out.println(“Welcome”);
    }
}
class Y extends X {
    public void print_to() { System.out.println(“to”); }
}
// Driver class
public class Main {
    public static void main(String[] args)
```

© Amity University Online

Notes

```
    {
        two g = new two();
        g.print_Welcome();
        g.print_to();
        g.print_Java();
    }
}
```

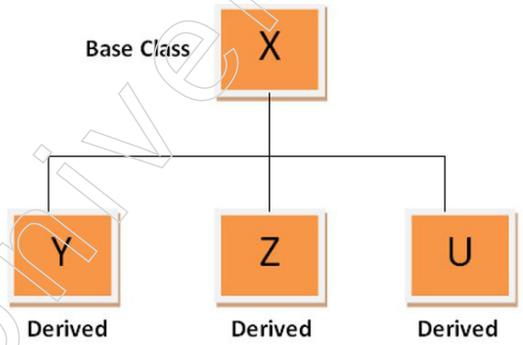
Output:

Welcome

to

java

3. Hierarchical Inheritance: "In Hierarchical Inheritance, one class represent as a superclass (base class) for more than one subclass. In the below image, class X serves as a base class for the derived class Y, Z and U."



Hierarchical Inheritance

```
// Java program to understandtheconcept of Hierarchical inheritance
class X {
    public void print_X() { System.out.println("Class X"); }
}
class Y extends X {
    public void print_Y() { System.out.println("Class Y"); }
}
class Z extends X {
    public void print_Z() { System.out.println("Class Z"); }
}
class U extends X {
```

(c) Amity University Online

Notes

```
    public void print_U() { System.out.println("Class U"); }
}
public class Test {
    public static void main(String[] args)
    {
            Y obj_Y = new Y();
            obj_Y.print_X();
            obj_Y.print_Y();

            Z obj_Z = new Z();
            obj_Z.print_X();
            obj_Z.print_Z();
Uobj_U = new U();
            obj_U.print_X();
            obj_U.print_U();
    }
}
```

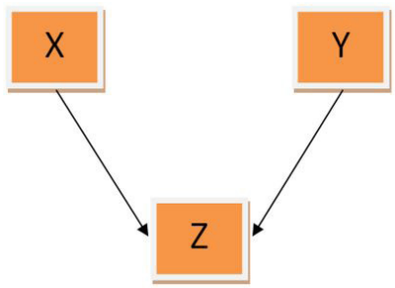
Output:

```
Class X
Class Y
Class X
Class Z
Class X
Class U
```

4. Multiple Inheritance (Through Interfaces):"In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In java, we can achieve multiple inheritances only through Interfaces. In the image below, Class Z is derived from interface X and Y."

(c) Amity University Online

Notes



Multiple Inheritances

Example of Multiple Inheritances

```
// Java program to understandtheconcept of Multiple inheritance
import java.io.*;
import java.lang.*;
import java.util.*;
interface X {
    public void print_Welcome();
}
interface Y {
    public void print_To();
}
interface Z extends X, Y {
    public void print_Java();
}
class child implements three {
    @Override public void print_Java()
    {
            System.out.println("Java");
    }
    public void print_for() { System.out.println("for"); }
}
// Drived class
public class Main {
    public static void main(String[] args)
```

(c) Amity University Online

```
    {
        child c = new child();
        c.print_Welcome();
        c.print_To();
        c.print_Java();
    }
}
```

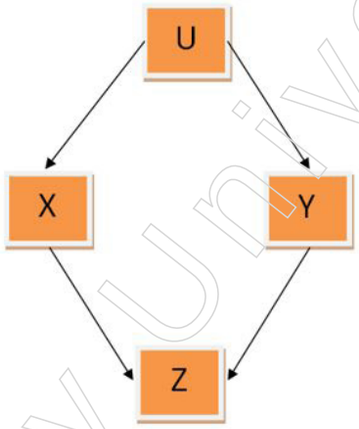
Output

Welcome

To

Java

5. Hybrid Inheritance(Through Interfaces):"It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces."



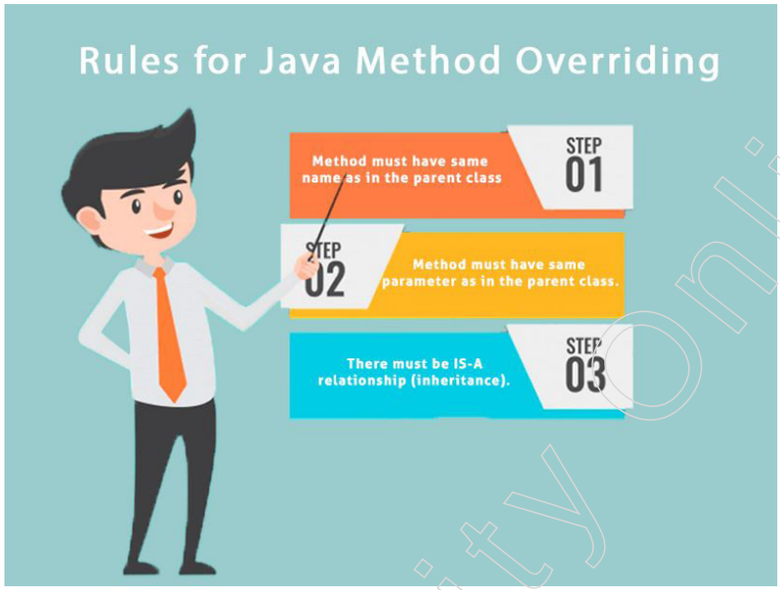
Hybrid Inheritances

1.4.2 Concept of method overriding with code snippet

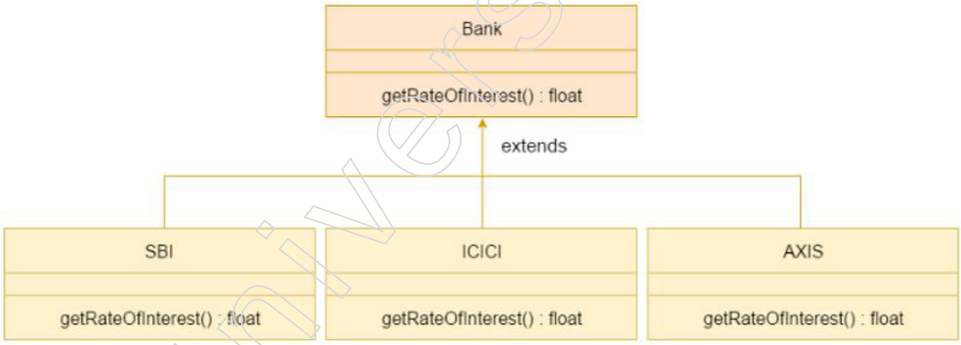
If subclass (child class) has the similar method as stated in the parent class, it is known as method overriding in Java. If a subclass provides the specific execution of the method that has been declared by one of its parent class, it is known as method overriding.

(c) Amity University Online

Notes



Example for method of overriding:



//To show the real scenario of Java Method Overriding where three classes are overriding the method of a parent class.

```
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 10;}
}
class ICICI extends Bank{
int getRateOfInterest(){return 9;}
}
class AXIS extends Bank{
```

(c) Amity University Online

Notes

```
int getRateOfInterest(){return 13;}
}
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

Output

```
SBI Rate of Interest: 10
ICICI Rate of Interest: 9
AXIS Rate of Interest: 13
```

1.4.3 Concept of Abstract classes with code snippet

Abstract Classes in Java: A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body). But in Java we have separate word abstract to make a class abstract.

Important observations about abstract classes in Java:

1. An instance of an abstract class cannot be created.
2. Constructors are allowed.
3. We can have abstract class without any abstract method.
4. Abstract classes can also have final methods
5. We are not allowed to create object for any abstract class.
6. We can define static methods in an abstract class

Let's discuss all these observation with the help of Java program

1. "An instance of an abstract class cannot be created."

```
// Class 1
// Abstract class
abstract class Race {
    abstract void fun();
}
```

(c) Amity University Online

Notes

```
// Class 2
class Derived extends Race {
    void fun()
    {
        System.out.println("Welcome to() Java");
    }
}
// Class 3
// Main class
class Main {
    // Main driver method
    public static void main(String args[])
    {
        // We can have references of Race type.
        Race b = new Derived();
        b.to();
    }
}
```

Output

```
Welcome to Java
```

2. "An abstract class can contain constructors in Java".

```
// Class 1
// Abstract class
abstract class Save {
    // Constructor of class 1
    Save()
    {
        // Print statement
        System.out.println("Save the world");
    }
    // Abstract method inside class1
    abstract void fun();
```

(c) Amity University Online

Notes

```
}
// Class 2
class Derived extends Save {
    // Constructor of class2
    Derived()
    {
        System.out.println("Save the Earth");
    }
    // Method of class2
    void fun()
    {
        System.out.println("Derived fun() called");
    }
}
// Class 3
// Main class
class GFG {
    public static void main(String args[])
    {
        // Creating object of class 2
        // inside main() method
        Derived d = new Derived();
    }
}
```

Output

```
Save the World
Save the Earth
```

3. "we can have an abstract class without any abstract method"

```
// Class 1
// An abstract class without any abstract method
abstract class Base {
    void fun()
```

(c) Amity University Online

Notes

```
    {
        // Print message if class 1 function is called
        System.out.println(
            "Base class function is called");
    }
}
// Class 2
class Derived extends Base {
}
// Class 3
class Main {
    // Main driver method
    public static void main(String args[])
    {
        // Creating object of class 2
        Derived d = new Derived();

        // Calling function defined in class 1 inside main()
        // with object of class 2 inside main() method
        d.fun();
    }
}
```

Output

```
Base class function is called
```

4. "Abstract classes can also have final methods"

```
// Class 1
// Abstract class
abstract class Base {
    final void fun()
    {
        System.out.println("Welcometo() Java");
    }
```

(c) Amity University Online

Notes

```
}
// Class 2
class Derived extends Base {
}
// Class 3
// Main class
class GFG {
    // Main driver method
    public static void main(String args[])
    {
            // Creating object of abstract class
            Base b = new Welcome();
            // Calling method on object created above
            // inside main()
            b.to();
    }
}
```

Output:

```
Welcome to Java
```

5. “For abstract class instantiation is not possible”

```
// Main class
// An abstract class
abstract class ABC {
    // Main driver method
    public static void main(String args[])
    {
        // Trying to create an object
        ABC abc = new ABC();
    }
}
```

Output: It will show error

```
Abc.java:11: error: ABC is abstract; cannot be instantiated
```

(c) Amity University Online

Notes

```
ABC abc = new ABC();
```

6. "Similar to the interface we can define static methods in an abstract class".

```
// Class ABC
// Abstract class
abstract class Helper {
    // Abstract method
    static void demofun()
    {
        // Print statement
        System.out.println("Welcome to Java");
    }
}
// Class xyz
// Main class extending Helper class
public class mno extends Helper {
    // Main driver method
    public static void main(String[] args)
    {
        // Calling method inside main()
        // as defined in above class
        Helper.demofun();
    }
}
```

Output

```
Welcome to Java
```

1.4.4 Concept of Final classes with code snippet

When a variable is declared with final keyword, its value can't be changed, essentially, a constant. This also means that you must given a final variable. If the final variable is a reference, this means that the variable cannot be recoil to reference another object, but the inner state of the object pointed by that reference variable can be altered i.e. you can add or subtract elements from the final array or final collection.

(c) Amity University Online

Notes

Initializing a final Variable

We must initialize a final variable; otherwise, the compiler will throw a compile-time error. A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable:

- You can initialize a final variable when it is declared. This approach is the most common. A final variable is called a blank final variable if it is not initialized while declaration. Below are the two ways to initialize a blank final variable.
- A blank final variable can be initialized inside an instance-initializer block or inside the constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise, a compile-time error will be thrown.
- A blank final static variable can be initialized inside a static block.

```
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
            // Creating sn object of StringBuilder class
            // Final reference variable
            final StringBuilder sb = new StringBuilder("Welcome");
            // Printing the element in StringBuilder object
            System.out.println(sb);
            // changing internal state of object reference by
            // final reference variable sb
            sb.append("to Java");
            // Again printing the element in StringBuilder
            // object after appending above element in it
            System.out.println(sb);
    }
}
```

Output

```
Welcome
Welcome to Java
```

Final Classes: "When a class is declared with final keyword, it is called a final class. A final class cannot be extended (inherited)."

(c) Amity University Online

Notes

Usage 1: One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float, etc. are final classes. We cannot extend them.

```
final class A
{
    // methods and fields
}
// The following class is illegal
class B extends A
{
}
```

Output

When we try to execute is gives us a COMPILE-ERROR! Can't subclass A because class has been declared final.

Usage 2: The other use of final with classes is to create an immutable class like the predefined String class. One cannot make a class immutable without making it final.

Final Methods

"When a method is declared with final keyword, it is called a final method. A final method cannot be override by other method. The Object class does this—a number of its methods are final. We must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes."

```
class A
{
  final void m1()
  {
System.out.println("This is a final method.");
  }
}
class B extends A
{
  void m1()
  {
    // Compile-error! We cannot override
```

(c) Amity University Online

Notes

```
System.out.println("Illegal!");
    }
}
```

Check your Understanding

1. In _____ subclasses inherit the features of one superclass.
2. In _____ one class can have more than one superclass and inherit features from all parent classes.
3. If subclass (child class) has the similar method as stated in the parent class, it is known as _____ in Java.
4. When a variable is declared with final keyword, its value can't be _____ .
5. When a method is declared with final keyword, it is called a _____ .

Summary

- Using the inheritance technique, a class can inherit the properties of a superclass.
- Inheritance refers to a subclass's capacity to inherit the generic properties of classes higher up in a class hierarchy.
- Properties can only be passed from a superclass to a subclass.
- Inheritance allows for code reuse.
- Inheritance improves software maintenance.
- Inheritance allows for class extension via subclassing.
- Single inheritance refers to a class that inherits properties from only one superclass.
- Multiple inheritance occurs when a class inherits properties from two or more superclasses.
- Because multiple inheritance is not supported in Java, the interface construct, which is accessed via the interface keyword, is an alternate option for achieving multiple inheritance. This solution, however, has limits.
- An abstract class is a class that is declared abstract —it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclasses. However, if it does not, then the subclass must also be declared abstract.

Activity

1. What will be the output of below program

```
interface I {
void x();
```

(c) Amity University Online

Notes

```
 void y();
}
class A implements I {
 A() {}
}
class B extends A {
 B() {}
 public void y() {
 }
b) z.printLine(2, “Object-Oriented Programming”)
c) z.printLine(“Java”)
z.printLine(“Object-Oriented Programming”, “Java”)
e) z.printLine(“Object-Oriented Programming”, 3)
 public void w() {System.out.println(“in A.w”);}
 public void x() {System.out.println(“in A.x”);}
 public void y() {System.out.println(“in A.y”);}
 System.out.println(“in B.y”);
 void z() {
 w();
 x();
 }
 static public void main(String args[]) {
 A aa = new A();
 B bb = new B();
 bb.z();
 bb.y();
 }
}
```

Questions and exercises

1. Define different Type of Inheritance?
2. What do you mean by method overriding?

(c) Amity University Online

3. Define Abstract classes
4. What do you mean by Final classes?

Notes

Glossary

- OOPs: Object Oriented programming
- Abstract Class: A class which is declared with the abstract keyword is known as an abstract class in Java.
- Hybrid Inheritance: It is a mix of two or more of the above types of inheritance.
- Hierarchical Inheritance: In Hierarchical Inheritance, one class represent as a superclass (base class) for more than one subclass.

Further Readings:

1. Object-Oriented Data Structures Using Java, Nell Dale, Daniel T. Joyce, Chip Weems , Jones and Bartlett
2. Effective Java, Joshua Bloch
3. Java Performance, Binu John and Charlie Hunt
4. JavaScript: Programming Basics for Absolute Beginners, Nathan CLARK

Check Your Understanding:

Answers

1. single inheritance
2. Multiple inheritances
3. method overriding
4. changed
5. final method

(c) Amity University Online

Notes

Module - 2 : Package and Exception in Java

Structure:

2.1 Introduction to Packages

- 2.1.1 Packages : Defining Package, CLASSPATH
- 2.1.2 Type of Packages
- 2.1.3 Importing Packages
- 2.1.4 Access Specifiers
- 2.1.5 Interfaces: Implementing Interfaces
- 2.1.6 Interfaces vs Abstract Class

2.2 Exception Handing

- 2.2.1 Introduction to Exceptions
- 2.2.2 Exception Handling by using Try-Catch
- 2.2.3 Catching Multiple Exceptions
- 2.2.4 Types of Exceptions
- 2.2.5 final vs finally clause ,throw vs throws

2.3 Throwing Exceptions

- 2.3.1 Implementation of finally, throw & throws
- 2.3.2 Fundamentals of throwing exceptions
- 2.3.3 Java's buit-in Exceptions
- 2.3.4 Writing Exception Subclasses
- 2.3.5 Implementation of Exception Handling Mechanism.

(c) Amity University Online

Unit - 2.1 : Introduction to Packages

Notes

Objectives

At the end of this unit, you will be able to understand:

- What is package and CLASSPATH
- Understand different type of Packages in Java
- How to import a Importing Packages
- What are Access Specifiers
- What are Interfaces & How to Implement Interfaces
- Difference between Abstract Class and Interfaces

Introduction

Package in Java is a system to sum up a group of classes, sub packages and interfaces. Packages are used for:

- To stop naming issue. For example there can be two classes with name Employee in two packages, Hospital.staff.cse.Employee and Hospital.staff.ee.Employee
- It is use to make searching/locating easier, usage of classes, interfaces, enumerations and annotations.
- To authorize Prescribed access: A protected member can be used by classes in the same package and its subclasses. A default member (without any access specifier) can be used by classes in the same package only.
- Packages can be framed as data encapsulation (or data-hiding).

2.1.1 Packages: Defining Package, Classpath

To generate a package is quite easy: just include a package command in the start of statement in a Java source file. Any classes defined in that file will go to the specified package. The package statement defines a name space in which classes are saved. If you erase the package statement, the class names will automatically be put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is ok for short, sample programs, it is complex for real applications. More often than not, developer will define a package for your code. General form of the package is represented as:

package pkg;

Java language uses the file system directories to store packages. For example, the .classxyz files for any classes you defined should be a part of MyPackage must be stored in a directory called MyPackage. Keep in mind case is significant, and the directory name is obliged to match the package name exactly.

More than one file can use the similar package statement. The package statement specifies to which package the classes defined in a file belong. It does not eliminate

(c) Amity University Online

Notes

other classes in other files from being part of that same package. In most real world problem packages are spread transversely in many files.

You can create a ladder of packages. To do that a developer need to separate each package name from the one above it by use of a period. General form of the package is represented as:

package pkg1[.pkg2[.pkg3]]

How packages work: Package names and directory structure are linked to each other. For example if a package name is Hospital.staff.cse, then there are three directories, Hospital, staff and cse such that cse is present in staff and staff is present hospital. Also, the directory Hospital is accessible through CLASSPATH variable, i.e., path of parent directory of hospital present in CLASSPATH. The idea is to make sure that classes are easily to trace.

Package naming conventions: Packages are named in reverse order of domain names, i.e., org.java.practice. For example, in a hospital, the recommended convention is hospital.tech.cse, hospital.tech.ee, hospital.art.history, etc.

Adding a class to a Package: We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new java file to define a public class, otherwise we can add the new class to an existing .java file and recompile it.

Subpackages: Packages that are inside another package are the subpackages. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Basic program for a Package:

```
package MyPackdemo;
class Statement {
String name;
double bal;
Statement(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

(c) Amity University Online

Notes

```
class Accountstatement {
public static void main(String args[]) {
Statement current[] = new Statement[3];
current[0] = new Statement("Ravi", 123.23);
current[1] = new Statement ("Will Tell", 157.02);
current[2] = new Statement ("Suresh", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file AccountStatement.java and put it in a directory called MyPackdemo.

Next, compile the file. Make sure that the resulting .class file is also in the MyPackdemo directory. Then, try executing the Accountstatement class, using the following command line:

java MyPackdemo.AccountStatement

Remember, you will need to be in the directory above MyPackdemo when you execute this command. (Developer can use one of the other two options described in the preceding section to specify the path MyPackdemo.)

As explained, AccountStatement is now part of the package MyPackdemo. This means that it cannot be executed by itself. That is, you cannot use this command line:

java AccountStatement

AccountStatementmust be qualified with its package name.

Access Protection: Classes and packages both are used for encapsulating and containing the name space and scope of variables and methods. Packages role is to play as containers for classes and other subordinate packages. The class is Java's least unit of abstraction. Because of the relationship between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

There are three types of access specifiers:

- Private
- Public
- Protected

(C) Amity University Online

Notes

	Private	No Modifier	Protected	Public
Same class	No	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Class Number Access

While Java's access control mechanism may seem complex, we can make it simpler.

- Anything defined public can be accessed from anywhere.
- Anything defined private cannot be seen outside of its class.
- When a member does not have an explicit access requirement, it is noticeable to subclasses as well as to other classes in the same package. This is the default access.

CLASSPATH: CLASSPATH is an environment variable which is used by Application ClassLoader to find and load the .class files. The CLASSPATH determine the path, to find third-party and user-defined classes that are not extensions or part of Java environment. Itincludes all the directories which have .class files and JAR files when setting the CLASSPATH.

You need to set the CLASSPATH if:

- You need to load a class that is not present in the current directory or any sub-directories.
- You need to load a class that is not in a location specified by the extensions mechanism.

The default value of CLASSPATH is a dot (.). It states that the only current directory searched. The default value of CLASSPATH is changed when you set the CLASSPATH variable or using the -classpath command (for short -cp). Put a dot (.) in the new setting if developer want to include the current directory in the search path.

If CLASSPATH finds a class file which is present in the current directory, then it will load the class from the current directory and use it, without considering the same name class presents in another directory which is also included in the CLASSPATH.

If you want to multiple classpaths, then developer need to put a semicolon (;) after each CLASSPATH.

Difference between PATH and CLASSPATH

PATH	CLASS PATH
It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.

(c) Amity University Online

Notes

PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command.

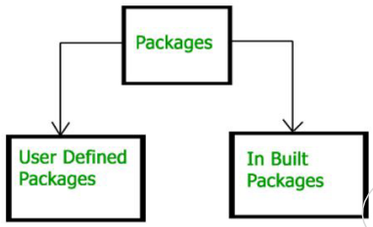
How to Set CLASSPATH in Windows Using Command Prompt:

```
set CLASSPATH=%CLASSPATH%;C:\Program Files\Java\jre1.8\rt.jar;
```

2.1.2 tType of Packages

Packages can be divided in two parts:

1. Java API Packages or Built-In Packages
2. User Defined Packages



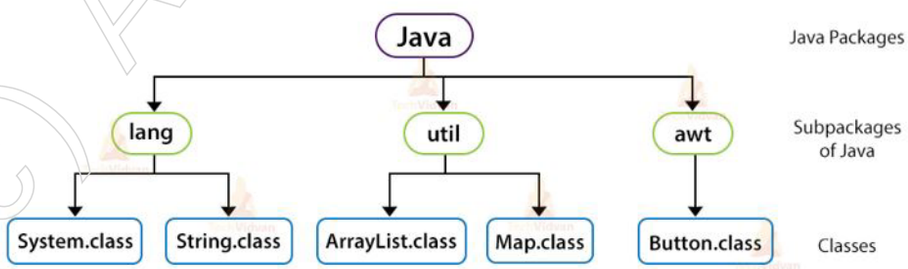
Types of Packages

1. Built in Packages: These packages containa large number of classes which are a part of Java API. Frequent using built in packages are:

1) java.lang: Contains language support classes (e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) java.io: Contains classed for supporting input / output operations.
3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support; for Date / Time operations.
4) java.applet: Contains classes for creating Applets.
5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button, menus etc).
6) java.net: Contain classes for supporting networking operations.

2. User-defined packages: These are the packages that are created by the developer.

Built-in Packages in Java



© Amity University Online

Notes

First developer create a directory myPackagedemo (name should be same as the name of the package). Then define the MyClassdemo inside the directory with the first statement being the package names.

```
// Name of the package must be similar to the directory
package myPackagedemo;
public class MyClassdemo
{
    public void getNames(String s)
    {
System.out.println(s);
    }
}
```

Now developer can use the MyClassdemo class in our program.

```
/* import 'MyClassdemo' class from 'names' myPackagedemo */
import myPackagedemo.MyClassdemo;
public class PrintName
{
  public static void main(String args[])
  {
     // Initializing the String variable
     // with a value
     String name = "Welcome to Java";

MyClassdemo obj = new MyClassdemo();
obj.getNames(name);
  }
}
```

MyClass.java need to be saved inside the myPackagedemo directory because it is a part of the defined package.

Using Static Import

Static import is a feature introduced in Java programming language (versions 5 and above) that permits members (fields and methods) defined in a class as public static to be incorporate in Java code without specifying the class in which the field is defined.

(c) Amity University Online

Notes

```
// Note static keyword after import.
import static java.lang.System.*;
class StaticImportDemo
{
public static void main(String args[])
{
        // as imported using static.
        out.println("Welcome to Java");
}
}
```

Output

Welcome to Java

Handling name conflicts: The only time we need to pay attention to packages is when we have a name conflict it means we have packages with same name class. For example both, java.util and java.sql packages have a class named called Date. So if we import both packages in same program it will show a compile error:

```
import java.util.*;
import java.sql.*;
Date today; //ERROR-- java.util.Date or java.sql.Date?
```

The compiler will not be able to understand which Date class program want. This problem can be resolved by using a precise import statement:

```
    import java.util.Date;
import java.sql.*;
```

If a developer wants to use both Date classes then, developer needs to use a full package name every time as he/she declares a new object of that class.

```
    java.util.DatedeadLine = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

2.1.3 Importing Packages

To import java package into a class, developer need to use java import keyword this is used to get the right to use package and its classes into the java program.

Use import keyword to access built-in and user-defined packages into your java source file so that your class can refer to a class that is insomeother package by directly using its name.

(c) Amity University Online

Notes

There are 3 different ways to refer to any class that is present in a different package:

- without import the package
- import package with specified class
- import package with all classes

1. Accessing package without import keyword: If you employ fully qualified name to import any class into your program, then only that specific class of the package are going to be accessible in your program, other classes within the same package won't be accessible. For this approach, there's no got to use the import statement. But developers need to use the fully qualified name whenever you're accessing the category or the interface. This is often generally used when two packages have classes with same names. For example: java.util and java.sql packages contain Date class.

```
package demopack;

public class ABC {

   public void msg() {

System.out.println("Hello Java");

   }

}

package mypack;

class XYZ {

   public static void main(String args[]) {

pack.ABC obj = new pack.ABC();  //using fully qualified name

obj.msg();

   }

}
```

Output

Hello Java

2. Import the Specific Class: Package can have many classes but sometimes we would like to access only specific class in our program in that case, Java allows us to specify class name along side the package name. If developer want to use import packagenamedemo.classnamedemo statement then only the class with name classnamedemo in the package will be available for use.

```
package demopack;

public class Demo {

   public void msg() {

System.out.println("Hello Java");
```

(c) Amity University Online

Notes

```
    }
}
package mypackdemo;
import demopack.Demo;
class Test {
    public static void main(String args[]) {
        Demo obj = new Demo();
obj.msg();
    }
}
```

Output

Hello Java

3. Import all classes of the package: If a developers use packagename.* statement, then all the classes and interfaces of this package are going to be accessible but the classes and interface inside the sub-packages won't be available to be used .

The import keyword is employed to form the classes of another package accessible to the present package.

```
package learnjava;
public class First{
    public void msg() {
System.out.println("Hello Java");
    }
}
package Java;
import learnjava.*;
class Second {
    public static void main(String args[]) {
        First obj = new First();
obj.msg();
    }
}
```

Output

Hello Java

(c) Amity University Online

Notes

2.1.4 Access Specifiers or Modifiers

There is no difference between access specifier and access modifier in Java. They both mean the same. Access modifier is the new and official term used instead of access specifier. The access modifiers in Java specify the accessibility or scope of a classes, field methods and constructors.

Four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1. Private: The private access specifier is available only within the class.

```
// private specifier
package p1;
class A
{
private void display()
    {
            System.out.println("Welcome to Java");
    }
}
class B
{
public static void main(String args[])
    {
            A obj = new A();
```

(c) Amity University Online

Notes

```
        // Trying to access private methodof another class
        obj.display();
    }
}
```

Output

error: display() has private access in A obj.display()

2. Default: If developers don't use any specifier, it is act as default by default. The default specifier is reachable only within package. It cannot be used from outside the package. It provides more convenience than private. But, it is further restrictive than protected, and public.

```
//save by ABC.java
package pack;
class ABC{
  void msg(){System.out.println("Hello");}
}
//save by XYZ.java
package mypack;
import pack.*;
class XYZ{
  public static void main(String args[]){
   ABC obj = new ABC();
obj.msg();
  }
}
```

Output

Compile time Error

3. Protected: The protected access specifier is available within package and outside the package but with the help inheritance only. The protected access specifier can be applied on the data member, method and constructor protected access specifier cannot be applied on the class.

```
//save by ABC.java
package pack;
public class ABC{
protected voidmsg(){System.out.println("Welcome to Java");}
```


(c) Amity University Online

Notes

```
}
//save by XYZ.java
package mypack;
import pack.*;
class XYZ extends ABC{
  public static void main(String args[]){
   XYZ obj = new XYZ();
obj.msg();
  }
}
```

Output

Welcome to Java

4. Public: The public access specifier is specified using the keyword public. The public access specifier has the widest scope among all other access specifier. Classes, methods, or data members that are declared as public are reachable from everywhere in the program. There is no constraint on the scope of public data members.

```
package demopack;
public class ABC{
public void msg(){System.out.println("Hello Java");}
}
package demomypack;
import demopack.*;
class XYZ{
  public static void main(String args[]){
   ABC obj = new ABC();
obj.msg();
  }
}
```

Output

Hello Java

2.1.5 Interfaces: Implementing Interfaces

Using the keyword interface, you'll fully abstract a class' interface from its implementation. That is, using interface, you'll specify what a category must do, but not

(c) Amity University Online

Notes

how it does it. Interfaces are syntactically almost like classes, but they lack instance variables, and their methods are declared with none body. In practice, this suggests that you simply can define interfaces that don't make assumptions about how they're implemented. Once it's declared, any number of classes can implement an interface. Also, one class can implement any number of interfaces

To implement an interface, a category must create the entire set of methods defined by the interface. However, each class is liberal to determine the small print of its own implementation. By assigning the interface keyword, Java allows you to completely utilize the "one interface, multiple methods" aspect of polymorphism

Interfaces are designed to support dynamic method resolution at run time. Normally, so as for a way to be called from one class to a different , both classes got to be present at compile time therefore the Java compiler can check to make sure that the tactic signatures are compatible. This condition by itself makes for a static and nonextensible classing environment.

Certainly throughout a system like this, functionality gets pushed up higher and better within the class hierarchy in order that the mechanisms are going to be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a way or set of methods from the inheritance hierarchy. Since interfaces are during a different hierarchy from classes, it's possible for classes that are unrelated in terms of the category hierarchy to implement an equivalent interface. this is often where the important power of interfaces is realized.

An interface can be defined as:

```
access interface demoname {
return-type method-demoname1(parameter-list);
return-type method-demoname2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
return-type method-demonameN(parameter-list);
type final-varnameN = value;
}
```

When no access specifier is included, then default access results, and therefore the interface is merely available to other members of the package during which it's declared. When it's declared as public, the interfaces are often employed by the other code. During this case, the interface must be the sole public interface declared within the file, and therefore the file must have an equivalent name because the interface. Name is that the name of the interface, and may be any valid identifier. Notice that the methods that are declared haven't any bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there is often no default implementation of any method specified within an interface. Each class that has an interface must implement all of the methods.

(c) Amity University Online

Notes

Variables are often declared inside interface declarations. They're implicitly final and static, meaning they can't be changed by the implementing class. They need to even be initialized. All methods and variables are implicitly public.

Here is an example of an interface definition. It defines a simple interface that contains one method called callbackdemo( ) that takes a single integer parameter.

```
interface Callbackdemo {
void callback(int param);
}
```

Uses of Java Interfaces:



Important Points to Remember

- A Java class can implement multiple Java Interfaces. It is necessary that the class must implement all the methods declared in the interfaces.
- Class should override all the abstract methods declared in the interface
- The interface allows sending a message to an object without concerning which classes it belongs.
- Class needs to provide functionality for the methods declared in the interface.
- All methods in an interface are implicitly public and abstract
- An interface cannot be instantiated
- An interface reference can point to objects of its implementing classes
- An interface can extend from one or many interfaces. Class can extend only one class but implement any number of interfaces
- An interface cannot implement another Interface. It has to extend another interface if needed.
- An interface which is declared inside another interface is referred as nested interface
- At the time of declaration, interface variable must be initialized. Otherwise, the compiler will throw an error.
- The class cannot implement two interfaces in java that have methods with same name but different return type.

Implementing Interfaces

Once an interface has been declared, one or many classes can use that interface. To implement an interface, developers need to include the implements clause in

(c) Amity University Online

Notes

a class definition, and then create the methods declared by the interface. Syntax for implementing interface:

```
class classnamedemo [extends superclass] [implements interface [,interface...]] {
// class-body
}
```

"If a class implements one or more interface, then the interfaces are separated with a comma. Ifa class implements two interfaces that declare the same method, then the same method willbe used by clients of either interface. The methods that implement an interface must bedeclared public. Also, the type signature of the implementing method must match exactlythe type signature specified in the interface definition."

```
class Client implements Demo {
// Implement Demo's interface
public void Demo(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that Demo( ) is declared using the public access modifier.

It is both allowable and common for classes that implement interfaces to declare added members of their own. For example, the following version of Client implementscallbackdemo( ) and adds the method adds the method Demo( ):

Demo( ):

```
class Client implements Callbackdemo {
// Implement Callback's interface
public void callbackdemo(int p) {
System.out.println("callback called with " + p);
}
void Demo() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

Accessing Implementations through Interface References

Developer can declare variables as object references that use an interface before class type. Any instance of any class that implements the declared interface are

(c) Amity University Online

Notes

often mentioned by such a variable. Once you call a way through one among these references, the right version are going to be called supported the particular instance of the interface being mentioned. This is often one among the key features of interfaces. The tactic to be executed is searched dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to understand anything about the "callee."

The following example calls the callbackdemo( ) method via an interface reference variable:

```
class Client implements Callbackdemo {
// Implement Callback's interface
public void callbackdemo(int p) {
System.out.println("callback called with " + p);
}
class demo {
public static void main(String args[]) {
Callbackdemo a = new Client();
a.callbackdemo(91);
}
}
```

Output

```
callback called with 91
```

Notice that variable a is declared to be of the interface type Callbackdemo, yet it was assigned an instance of Client. Although a can be used to access the callbackdemo( ) method, it cannot access any other members of the Client class. An interface reference variable only has information of the methods declared by its interface declaration. Thus, a could not be used to access demo( ) since it is defined by Client but not Callbackdemo.

While the previous program shows, mechanically, how an interface reference variable can use an implementation object, but it does not demonstrate the polymorphic power ofsuch a reference. To sample this usage, first create the second implementation of Callbackdemo,shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callbackdemo {
// Implement Callback's interface
public void callbackdemo(int p) {
System.out.println("Another version of callback");
System.out.println("p squared is " + (p*p));
```


(c) Amity University Online

Notes

```
}
}
class demo2 {
public static void main(String args[]) {
Callbackdemod  = new Client();
AnotherClientob = new AnotherClient();
d.callbackdemo(91);
d = ob; // d now refers to AnotherClient object
d.callback(91);
}
}
```

Output

```
callbackdemo called with 91
Another version of callback
p squared is 1764
```

As we can see in above program, the version of callbackdemo( ) that is called is determined by the type of object that d refers to at run time.

Partial Implementations

If a class contains an interface but does not fully implement the methods declared by that interface, then that class must be declared as an abstract class. For example:

```
abstract class Incomplete implements Callbackdemo {
int a, b;
void show() {
System.out.println(a + " " + b);
}
}
```

Here, the class incomplete does not implement callbackdemo( ) and must be declared as abstract. Any class that inherits Incomplete must implement callbackdemo( ) or need to be declared abstract itself.

(c) Amity University Online

Notes

Nested Interfaces

An interface can be defined as member of a class or another interface. Such an interface is called a **member interface** or a **nested interface**. A nested interface can be defined as public, private, or protected. This differs from a top-level interface, which must either be declared as public or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Example for Nested Interfaces:

```
class ABC {
// this is a nested interface
public interface NestedIFdemo {
booleanisNotNegative(int y);
}
}
// XYZ implements the nested interface.
class XYZ implements ABC.NestedIFdemo {
public booleanisNotNegative(int y) {
return y<0 ? false : true;
}
}
class NestedIFDemo1 {
public static void main(String args[]) {
// use a nested interface reference
ABC.NestedIFdemo demo = new XYZ();
if(demo.isNotNegative(10))
System.out.println(“10 is not negative”);
if(demo.isNotNegative(-12))
System.out.println(“this won’t be displayed”);
}
}
```

Notice that ABC defines a member interface called NestedIFdemo and that it is declared public. Next, XYZ implements the nested interface by specifying

(c) Amity University Online

Notes

implements ABC.NestedIFdemo

The name is fully qualified by the enclosing class' name. Inside the main( ) method, an ABC.NestedIFdemo reference called demo is created, and it is assigned a reference to a XYZ object. Because XYZ implements XYZ.NestedIFdemo, this is legal.

Interfaces Can Be Extended

One interface can inherit the characteristics another interface by use of the keyword extends. The declaration is the same as for inheriting classes. When a class uses an interface that inherits the properties of another interface, it must provide implementations for all other methods declared within the interface inheritance chain.

Example:

```
// One interface can extend another.
interface ABC {
void methX();
void methY();
}
// XYZ now includes methX() and methY() -- it adds methZ().
interface XYZ extends ABC {
void methZ();
}
// This class must implement all of ABC and XYZ
class MyClassdemo implements XYZ {
public void methX() {
System.out.println("Implement methX().");
}
public void methZ() {
System.out.println("Implement methY().");
}
public void methZ() {
System.out.println("Implement methZ().");
}
}
class IFExtenddemo {
public static void main(String arg[]) {
```

(c) Amity University Online

Notes

```
MyClassdemoob = new MyClassdemo();
ob.methX();
ob.methY();
ob.methZ();
}
}
```

To try, developer might want to try removing the implementation for methX( ) in MyClassdemo. This will show a compile-time error. As told earlier, any class that implements an interface must implement all methods declared by that interface, together with any that are inherited from other interfaces.

2.1.6 Interfaces v/s Abstract Class

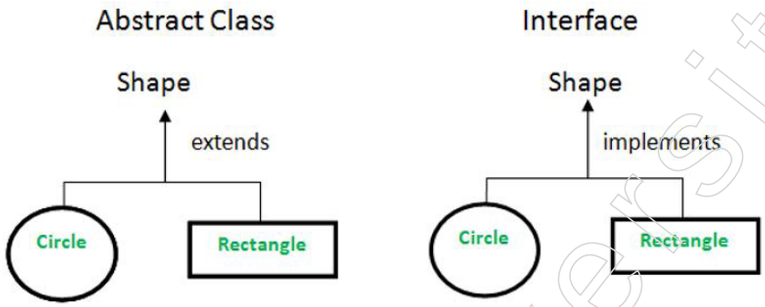
Basis	Interface	Abstract Class
Type of methods:	Interface can have only abstract methods	An abstract class can have abstract and non-abstract methods.
Final Variables:	Variables declared in a Java interface are by default final.	An abstract class may include non-final variables.
Type of variables	The interface has only static and final variables.	Abstract class can have final, non-final, static and non-static variables
Implementation:	Interface can't provide the implementation of an abstract class.	Abstract class can provide the implementation of the interface.
Inheritance v/s Abstraction:	A Java interface can be implemented using the keyword "implements"	An abstract class can be extended using the keyword "extends".
Multiple implementations:	An interface can extend another Java interface only	Abstract class can extend another Java class and implement multiple Java interfaces.
Accessibility of Data Members:	Members of a Java interface are public by default	A Java abstract class can have class members like private, protected, etc.
Inheritance	Interface supports multiple inheritances.	Abstract class doesn't support multiple inheritances.
Example	`public interface Drawable{ void draw(); }`	`public abstract class Shape{ public abstract void draw(); }`

Basis	Interface	Abstract Class
Type of methods:	Interface can have only abstract methods	An abstract class can have abstract and non-abstract methods.
Final Variables:	Variables declared in a Java interface are by default final	An abstract class may include non-final variables.
Type of variables	The interface has only static and final variables.	Abstract class can have final, non-final, static and non-static variables
Implementation:	Interface can't provide the implementation of an abstract class.	Abstract class can provide the implementation of the interface.

(c) Amity University Online

Notes

Inheritance v/s Abstraction	A Java interface can be implemented using the keyword “implements”	An abstract class can be extended using the keyword “extends”
Multiple implementations:	An interface can extend another Java interface only	Abstract class can extend another Java class and implement multiple Java interfaces.
Accessibility of Data Members:	Members of a Java interface are public by default	A Java abstract class can have class members like private, protected, etc
Inheritance	Interface supports multiple inheritances.	public abstract class Shape(public abstract void draw();
Example	public interface Drawable (void draw ();	



Program to show the concept of abstract class

```
import java.io.*;
// abstract class
abstract class Shape {
    // declare fields
    String objectName = " ";
    Shape(String name) { this.objectName = name; }
    // declare non-abstract methods
    // it has default implementation
    public void moveTo(int m, int n)
    {
        System.out.println(this.objectName + " "
                                + "has been moved to"
                                + " m = " + m + " and n = " + n);
    }
    // abstract methods which will be
    // implemented by its subclass(es)
```

(c) Amity University Online

Notes

```
    abstract public double area();
    abstract public void draw();
}
class Rectangle extends Shape {
    int length, width;
    // constructor
    Rectangle(int length, int width, String name)
    {
        super(name);
        this.length = length;
        this.width = width;
    }
    @Override public void draw()
    {
        System.out.println("Rectangle has been drawn ");
    }
    @Override public double area()
    {
        return (double)(length * width);
    }
}
class Circle extends Shape {
    double pi = 3.14;
    int r;
    // constructor
    Circle(int r, String name)
    {
        super(name);
        this.radius = radius;
    }
    @Override public void draw()
    {
```

(c) Amity University Online

Notes

```
            System.out.println("Circle has been drawn ");
        }
        @Override public double area()
        {
            return (double)((pi * r * r) / 2);
        }
    }
    class GFG {
        public static void main(String[] args)
        {
            Shape rect = new Rectangle(2, 3, "Rectangle");
            System.out.println("Area of rectangle: "
                                                + rect.area());
            rect.moveTo(1, 2);
            System.out.println(" ");
            Shape circle = new Circle(2, "Circle");
            System.out.println("Area of circle: "
                                                + circle.area());
            circle.moveTo(2, 4);
        }
    }
```

Output

```
Area of rectangle: 6.0
Rectangle has been moved to a = 1 and b = 2
Area of circle: 12.56
Circle has been moved to a = 2 and b = 4
```

(c) Amity University Online

Notes

When to Use Abstract Class

- In the java application, there are some related classes that need to share some lines of code then you can put these lines of code within the abstract class and this abstract class should be extended by all these related classes.
- You can define the non-static or non-final field(s) in the abstract class so that via a method you can access and modify the state of the Object to which they belong.
- You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

Program to illustrate the concept of interface

```
import java.io.*;
interface Shape {
    // abstract method
    void draw();
    double area();
}
class Rectangle implements Shape {
    int length, width;
    // constructor
    Rectangle(int length, int width)
    {
        this.length = length;
        this.width = width;
    }
    @Override public void draw()
    {
        System.out.println("Rectangle has been drawn ");
    }
    @Override public double area()
    {
        return (double)(length * width);
    }
}
class Circle implements Shape {
```

(c) Amity University Online

Notes

```
        double pi = 3.14;
        int r;
        // constructor
        Circle(int r) { this.r = r; }
        @Override public void draw()
        {
                System.out.println("Circle has been drawn ");
        }
        @Override public double area()
        {
                return (double)((pi * r * r) / 2);
        }
    }
    class GFG {
        public static void main(String[] args)
        {
                Shape rect = new Rectangle(2, 3);
                System.out.println("Area of rectangle: "
                                                + rect.area());
                        Shape circle = new Circle(2);
                System.out.println("Area of circle: "
                                                + circle.area());
        }
    }
```

Output

Area of rectangle: 6.0

Area of circle: 12.56

When to use Interface

- It is a total abstraction; all methods declared within an interface must be implemented by the class (es) that implements this interface.
- A class can implement more than one interface. It is called multiple inheritances.
- You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.

© Amity University Online

Notes

Check your Understanding

1. General form of the package is represented as _____ .
2. CLASSPATH is an environment variable which is used by Application ClassLoader to find and load the _____ files.
3. The default value of CLASSPATH is a _____ .
4. _____ are the packages that are created by the developer.
5. Java allows us to specify class name alongside the _____ name.
6. Developers don't use any specifier, it is act as _____ by default.

Summary

- The package statement defines a name space in which classes are saved. If you erase the package statement, the class names will automatically be put into the default package, which has no name.
- Classes and packages both are used for encapsulating and containing the name space and scope of variables and methods. Packages role is to play as containers for classes and other subordinate packages.
- The CLASSPATH determine the path, to find third-party and user-defined classes that are not extensions or part of Java environment. It includes all the directories which have .class files and JAR files when setting the CLASSPATH.
- Built in Packages: These packages contain a large number of classes which are a part of Java API.

1) java.lang: Contains language support classes (e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) java.io: Contains classed for supporting input / output operations.
3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support; for Date / Time operations.
4) java.applet: Contains classes for creating Applets.
5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button, menus etc).
6) java.net: Contain classes for supporting networking operations.

- To import java package into a class, developer need to use java import keyword this is used to get the right to use package and its classes into the java program.
- Once an interface has been declared, one or many classes can use that interface. To implement an interface, developers need to include the implements clause in a class definition, and then create the methods declared by the interface.
- In the java application there are some related classes that need to share some lines of code then you can put these lines of code within the abstract class and this abstract class should be extended by all these related classes.

Activity

1. Create a program using multiple interfaces?

(c) Amity University Online

Notes

Questions and exercises

1. Define package?
2. What do you mean by CLASSPATH?
3. Define different type of packages?
4. How to import packages?
5. What do you mean by access specifiers?
6. What are interfaces?
7. How one can implement interfaces?
8. What is the difference between Interfaces and Abstract Class?

Glossary

-cp: Classpath

- Static Import: Static import is a feature introduced in Java programming language (versions 5 and above) that permits members (fields and methods) defined in a class as public static to be incorporate in Java code without specifying the class in which the field is defined.
- Nested Interface: An interface can be defined as member of a class or another interface such an interface is called a member interface or nested interface.

Further Readings:

1. The Java Programming Language, James Gosling and Ken Arnold
2. JavaScript: Programming Basics for Absolute Beginners, Nathan Clark
3. Java Performance, Binu John and Charlie Hunt
4. Java: A Beginner's Guide, Eighth Edition, Herbert Schildt

Check Your Understanding:

Answers

1. package pkg;
2. .class
3. dot (.)
4. User-defined packages
5. package
6. default

(c) Amity University Online

Notes

Unit - 2.2 : Exception Handling

Objectives

At the end of this unit, you will be able to understand:

- All about Exception and Exception Handling
- How to handle Exception by using Try-Catch
- What is Catching Multiple Exceptions
- Different Types of Exceptions
- Difference between final v/s finally clause andhrow v/s throws

Introduction

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this unit, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

2.2.1 Introduction to Exceptions

An exception is an unwanted or unexpected event, which happens during the execution of a program i.e at run time that disturbed the regular flow of the program's instructions.

People often confuse between error and exception. To make this clear, An Error indicates towards the serious issue that a reasonable application should not try to catch. Exception indicates towards the conditions that a reasonable application might try to catch.

Syntax of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
```

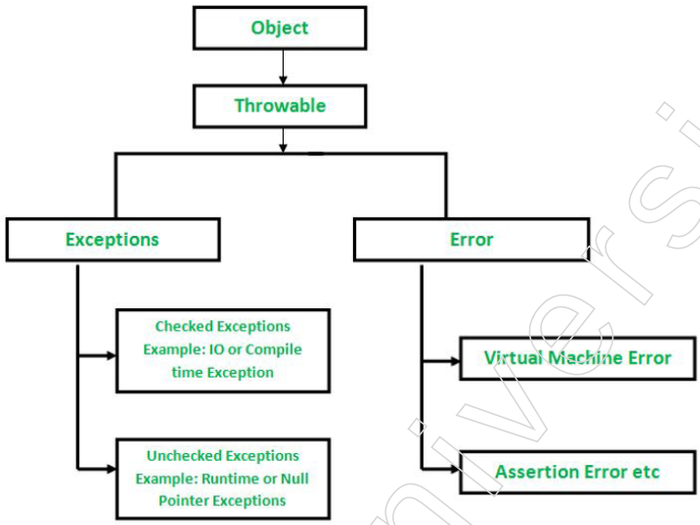
(c) Amity University Online

Notes

```
finally {
// block of code to be executed after try block ends
}
```

Exception Hierarchy

"All exception and errors types are sub classes of class Throwable, which is base class of hierarchy.One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch,Error are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error."



Default Exception Handling:

Whenever inside a program code, if an exception has occurred, the tactic creates an Object referred to as Exception Object and hands it off to the run-time system (JVM). "The exception object includes name and definition of the exception, and current state of the program code where exception has occurred. Creating the Exception Object and handling by the developer it to the run-time system is named throwing an Exception." There could be the list of the tactics that had been called to urge to the method where exception was occurred. This ordered list of the methods is named Call Stack.

Now the subsequent procedure will happen.

- The run-time system searches the decision stack to seek out the tactic that contains block of code which will handle the occurred exception. The block of the code is named Exception handler.
- The run-time system starts looking from the method during which exception occurred proceeds through call stack in the opposite order during which methods were called.
- If it locates appropriate handler then it passes the occurred exception thereto. Appropriate handler defines itself as the sort of the exception object thrown matches the sort of the exception object it can handle.

(c) Amity University Online

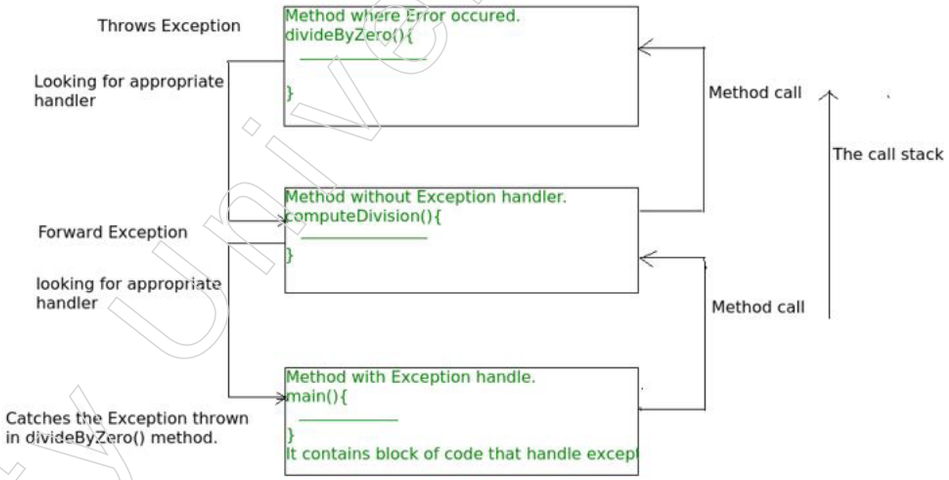
Notes

- If run-time system looks for all the methods on call stack and couldn't have locate the acceptable handler then run-time system handover the Exception Object to default exception handler, which is a module of run-time system. This exception handler prints the exception details within the following format and ends the program unusually.

```
// Example to show how exception is thrown.
class ThrowsExecpDemo{
    public static void main(String args[]){
        String str = null;
        System.out.println(str.length());
    }
}
```

Output

```
Exception in thread "main" java.lang.NullPointerException
at ThrowsExecp.main(File.java:8)
```



The call stack and searching the call stack for exception handler.

Example of how run-time system searched for exception handling

```
class ExceptionThrowndemo
{
        static int divideByZero(int x, int y){
        // this statement will cause ArithmeticException(/ by zero)
        int i = x/y;
        return i;
    }
```


(c) Amity University Online

Notes

```
    static int computeDivision(int x, int y) {
            int a =0;
            try
            {
            a = divideByZero(x,y);
            }
            // doesn't matches with ArithmeticException
            catch(NumberFormatException ex)
            {
            System.out.println("NumberFormatException is occured");
            }
            return a;
    }
    public static void main(String args[]){
            int x = 1;
            int y = 0;
            try
            {
                    int i = computeDivision(x,y);
            }
                    catch(ArithmeticException ex)
            {
                    System.out.println(ex.getMessage());
            }
    }
}
```

Output

```
            / by zero.
```

Customized Exception Handling:

Java exception handling is managed with the help of 5 keywords: try, catch, throw, throws, and finally.

- Program statements thatadeveloper can think raise exceptions are controlled within a try block.

© Amity University Online

Notes

- If an exception comes under the try block, it is thrown.
- Developer code can catch this exception (using catch block) and figure out in some rational manner.
- System-generated exceptions are mechanically thrown by the Java run-time system.
- If a developer wants to manually throw an exception, then he/she must use the keyword throw.
- Any exception that is thrown out of a method must be specified as such by a throws clause.
- Any code that completely must be executed or run after a try block completes is put in a finally block.

Points to remember:

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own try block and provide separate exception handler within own catch block for each of them.
- If an exception occurs within the try block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put catch block after it. There can be more than one exception handlers. Each catch block is an exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from Throwable class.
- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not. If exception occurs, then it will be executed after try and catch blocks. And if exception does not occur then it will be executed after the try block. The finally block in Java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

2.2.2 Exception handling by using Try-Catch

Though the default exception handler given by the Java run-time system is helpful in debugging, Developer will usually want to handle an exception by himself/herself. Doing this provides two advantages.

- It allows developer to fix the error.
- It prevents the program from terminating abstractedly.

Most users would be puzzled (to say the least) if your program stopped executing and printed a stack trace at any time an error occurred! Luckily, it is very easy to stop it.

To safeguard against and handle a run-time error, simply put that set of code that you want to monitor inside a try block. Right away following the try block, includes a catch clause that provides the information of exception type that you wish to catch. To demonstrate how simply this can be achieved, the following program code includes a try block and a catch clause that executes the ArithmeticException created by the division-by-zero error:

```
class Excdemo {
```


(c) Amity University Online

Notes

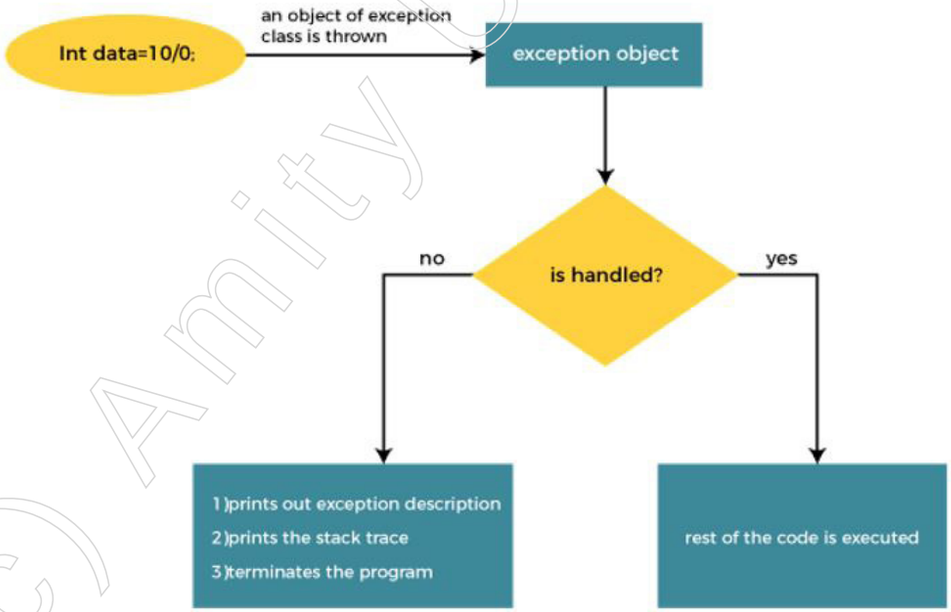
```
public static void main(String args[]) {
int x, y;
try {
x = 0;
y = 24 / x;
System.out.println("Error.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

Output:

Division by zero.

If you notice in above program that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control put it out of the try block to the catch block.

In other words, catch is not "called," so execution never "returns" to the try block from a catch. Thus, the line "Error." is not displayed. Once the catch statement has executed, program control continues the execution with the next line in the program to follow the entire try/catch instrument.



Internal working of a try-Block

(c) Amity University Online

Notes

A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement. A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described shortly). The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement.

The objective of most of the well-constructed catch clauses is to resolve the exceptional condition and then continue to execute the program as if the error had never happened.

For example, in the next program each iteration of the for loop obtains two arbitrary integers. Those two integers are then divided by each other, and the result is used to divide the value 34567. The final result is put into ABC. If either division operation causes a divide-by-zero error, it is caught, the value of ABC is set to zero, and the program continues.

```
//How to Handle an exception and move on.

import java.util.Random;

class HandleErrorDemo {

public static void main(String args[]) {

int x=0, b=y, z=0;

Random a = new Random();

for(int i=0; i<32000; i++) {

try {

y = a.nextInt();

z = a.nextInt();

x = 56789 / (y/z);

} catch (ArithmeticException e) {

System.out.println(“Division by zero.”);

x = 0; // set a to zero and continue to the program

}

System.out.println(“x: “ + x);

                }

    }

}
```

Handling Arithmetic Exception with a different type of exception class (ArrayIndexOutOfBoundsException).

(c) Amity University Online

Notes

```
public class TryCatchExampledemo {
   public static void main(String[] args) {
      try
      {
      int data=25/0;
      }
// try to handle the Exception using ArrayIndexOutOfBoundsException
      catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

2.2.3 Catching Multiple Exceptions

There are cases where one can found more than one exception could be raised by a single piece of code. To tackle this, developer can specify two or more catch clauses, each catching a type of exception which is raised. When an exception is thrown, each catch statement is inspected in order, and whichever the first one whose type matches that of the resulting exception is get executed. After one of the catch statements gets executes, the others are automatically bypassed, and execution continues will run after the try/catch block.

Example of two different exception types:

```
// multiple catch statements.
class MultiCatchdemo {
public static void main(String args[]) {
try {
int x = args.length;
System.out.println("x = " + x);
int y = 42 / x;
int z[] = { 1 };
z[52] = 99;
} catch(ArithmeticException e) {
```


(c) Amity University Online

Notes

```
System.out.println(“Divide by 0: “ + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println(“Array index oob: “ + e);
}
System.out.println(“After try/catch blocks.”);
}
}
```

This code will cause a division-by-zero exception if it is initiated with no command line arguments, since x will equal zero. It will endure the division if you give a command-line argument, setting x to something more than zero. But it will cause an ArrayIndexOutOfBoundsException, since the int array z has a length of 1, yet the program attempts to assign a value to z[42].

Here is the output by executing it in two different ways:

```
C:\>java MultiCatch
x = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatchTestArg
x = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:52
After try/catch blocks.
```

When a developer use more than one catch statements, it is vital to keep in mind that exception subclasses must come before any of their superclasses. It is because a catch statement that uses a superclass will grasp exceptions of that type plus any of its subclasses. Thus, a subclass would never be executes if it is came after superclass. additional, in Javalanguage,a unreachable code is lead to an error.

For example:

```
class SuperSubCatch {
public static void main(String args[]) {
try {
int x = 0;
int y = 42 / a;
} catch(Exception e) {
System.out.println(“Generic Exception catch.”);
```

(c) Amity University Online

Notes

```
}

/*ArithmeticException is a subclass of Exception that's why this catch never get executed */

catch(ArithmeticException e) { // ERROR - unreachable

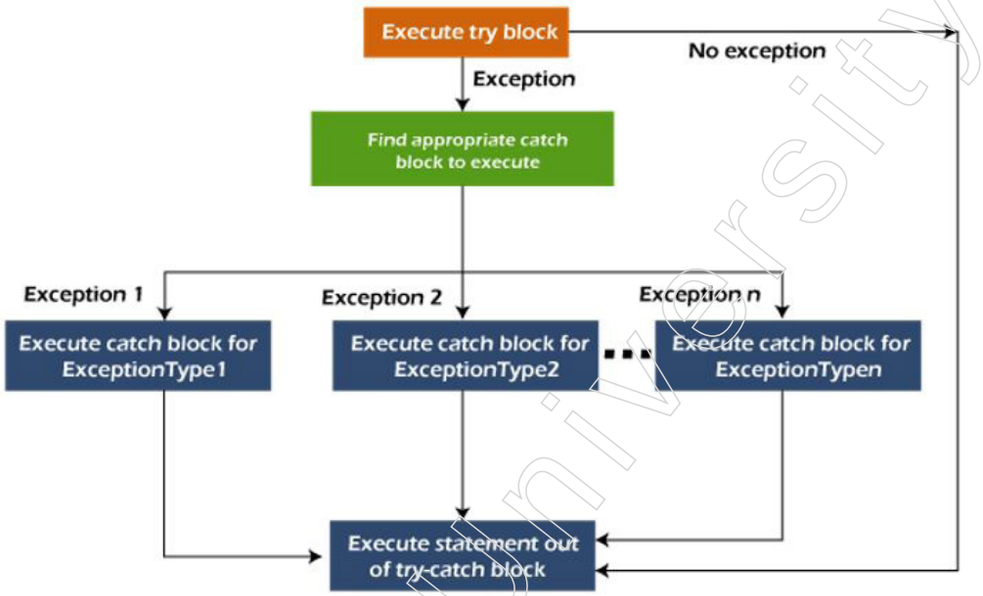
System.out.println("This is never reached.");

            }

    }

}
```

Flowchart for Multi-Catch Block:



Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

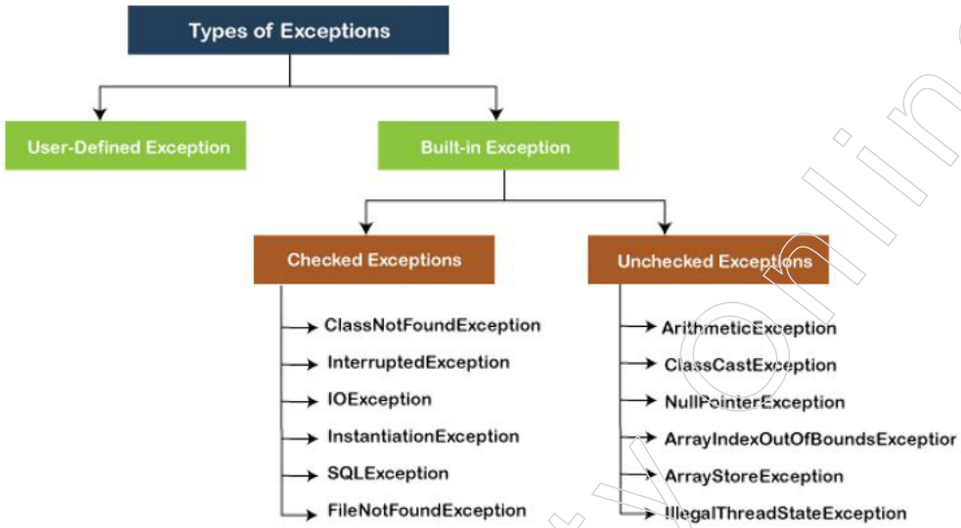
2.2.4 Types of Exceptions

Java defines different types of exceptions that narrate to its various class libraries. Java also allows a developer to define and create their own exceptions. In java exceptions are majorly categorized into two types:

- Built-In Exception
- User Defined Exception

(c) Amity University Online

Notes



1. Built-in Exceptions: Built-in exceptions are the pre-defined exceptions which are already exist in Java libraries

   Some of important built-in exceptions in Java listed below:

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method which is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

Some of the example of Built in function:

Nullpointerexception:

```
//Program for NullPointerException

class NullPointer_Demo

{

    public static void main(String args[])
```


(c) Amity University Online

Notes

```
        {
                try {
                        String a = null; //null value
                        System.out.println(a.charAt(0));
                } catch(NullPointerException e) {
                        System.out.println("NullPointerException..");
                }
        }
    }
```

Output

NullPointerException..

StringIndexOutOfBound Exception

```
    //  programforStringIndexOutOfBoundsException
    class StringIndexOutOfBound_Demo
    {
        public static void main(String args[])
        {
        try {
                String x = "Welcome to Java";
                        char c = x.charAt(24); //
                        System.out.println(c);
                }
                catch(StringIndexOutOfBoundsException e) {
                        System.out.println("StringIndexOutOfBoundsException");
                }
        }
    }
```

Output

StringIndexOutOfBoundsException

FileNotFound Exception

```
    //Program forFileNotFoundException
```

(c) Amity University Online

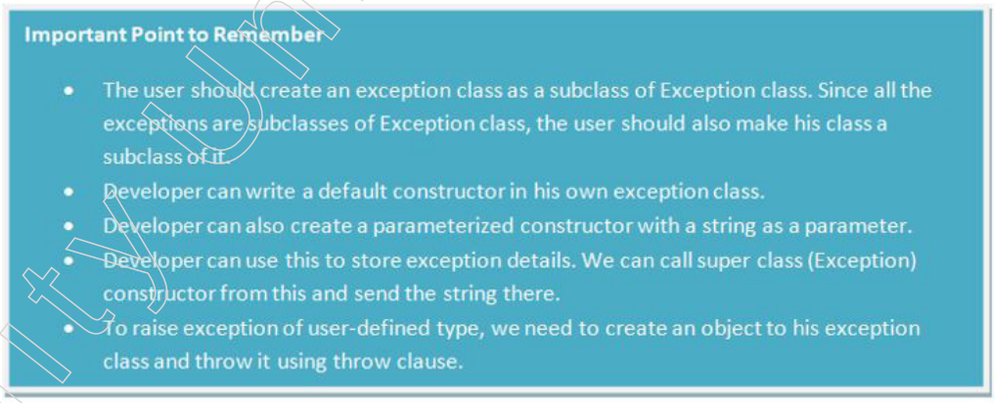
Notes

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo {
    public static void main(String args[]) {
            try {
                    File file = new File("C://file.txt");
                    FileReaderfr = new FileReader(file);
            } catch (FileNotFoundException e) {
            System.out.println("File does not exist");
            }
    }
}
```

Output

File does not exist

2. User-Defined Exception: There are situation where the built-in exceptions in Java are not able to describe a certain issues. In those cases, developer needs to create exceptions which are called 'user-defined Exceptions'.



Example for user defined Exception

```
// Program for user defined exception
class MyException extends Exception
{
//store account information
private static int accno[] = {101, 102, 103, 104};
private static String name[] =
```

(c) Amity University Online

Notes

```
                {"Nisha", "Shubham", "Sushant", "Abhinav", "Avni"};
    private static double bal[] =
        {20000.00, 22000.00, 4600.0, 888.00, 1133.55};
    MyException() { }
    MyException(String str) { super(str); }
    // write main()
        public static void main(String[] args)
        {
        try {
                // display the Table heading
                System.out.println("ACCNO" + "\t" + "CUSTOMER NAME" +"\t" + "ACC
BALANCE");
                        // display the actual account information
                for (int i = 0; i<5 ;i++)
                {
                System.out.println(accno[i] + "\t" + name[i] +"\t" + bal[i]);
        // display own exception if balance < 1000
                if (bal[i] < 1000)
                        {
                        MyException me =new MyException("Balance is less than
1000");
                        throw me;
                                }
                        }
                } //end of try

                catch (MyException e) {
                        e.printStackTrace();
                }
        }
    }
```

Output:

RunTime Error

(c) Amity University Online

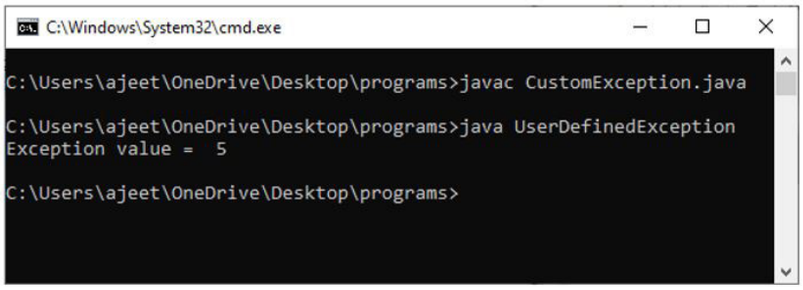
Notes

```
MyException: Balance is less than 1000
at MyException.main(fileProperty.java:36)
ACCNO        CUSTOMER NAME        ACC BALANCE
101          Nisha                20000.0
102          Shubham              22000.0
103          Sushant              4600.0
104          Abhinav              888.0
Another example for User defined exception
import java.util.*;
class UserDefinedExceptiondemo{
   public static void main(String args[]){
      try{
         throw new NewException(5);
      }
      catch(NewException ex){
         System.out.println(ex) ;
      }
   }
}
class NewException extends Exceptiondemo{
   int x;
   NewException(int y) {
      x=y;
   }
   public String toString(){
      return ("Exception value =  "+x) ;
   }
}
```

Output:

(c) Amity University Online

Notes



2.2.5 throw v/s throws and final v/s finally clause

Difference between throw v/s throws

Throw	Throws
Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
Using throw keyword, we can only propagate unchecked exception ke, the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only
The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
throw is used within the method.	throws is used with the method signature.
We are allowed to throw only one exception at a time ie, we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Difference between final v/s finally

Final	Finally
final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.
Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.
Once declared final variable becomes constant and cannot be modified.	finally block runs the important code even if exception occurs or not.
final method cannot be overridden by sub class.	finally block cleans up all the resources used in try block
Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.

Amity University Online

Notes

Check your Understanding

1. The Exception Handling in Java is one of the powerful mechanism to handle the _____ .
2. There could be the list of the tactics that had been called to urge to the method where exception was occurred. This ordered list of the methods is named Call _____ .
3. There are cases where one can found more than one exception could be raised by a single piece of code is called catching _____ .
4. Java _____ is used to throw an exception explicitly in the code, inside the function or block of code.

Summary

- The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error.
- All exception and errors types are sub classes of class Throwable, which is base class of hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch.
- The run-time system searches the decision stack for the tactic that contains the block of code that will handle the thrown exception. The code block is titled Exception handler.
- The run-time system begins by looking at the method in which the exception occurred and proceeds through the call stack in the reverse order in which the methods were called.
- If it finds an appropriate handler, it passes the thrown exception to it. Appropriate handler defines itself as the type of exception object thrown matches the type of exception object it can handle.
- If the run-time system searches all methods on the call stack and cannot find an acceptable handler, the Exception Object is passed to the run-time system's default exception handler, which is a module. This exception handler prints the exception details in the format shown below and terminates the programme in an unusual way.
- When a developer use more than one catch statements, it is vital to keep in mind that exception subclasses must come before any of their superclasses. It is because a catch statement that uses a superclass will grasp exceptions of that type plus any of its subclasses.
- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Activity

1. Create a program to handle exception using try catch block?

(c) Amity University Online

Notes

Questions and exercises

1. What do you understand from the word Exceptions?
2. How ro handle Exception using Try-Catch
3. How to catch Multiple Exceptions
4. Define various types of Exceptions
5. What are the difference between final vs finally clause?
6. What are the difference between throw vs throws?

Glossary

- JVM: Java virtual Machine
- Built-in Exceptions: Built-in exceptions are the pre-defined exceptions which are already exist in Java libraries.
- ArithmeticException: It is thrown when an exceptional condition has occurred in an arithmetic operation.
- ArrayIndexOutOfBoundsException: It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
- ClassNotFoundException: This Exception is raised when we try to access a class whose definition is not found
- FileNotFoundException: This Exception is raised when a file is not accessible or does not open.
- IOException: It is thrown when an input-output operation failed or interrupted
- InterruptedException: It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
- NoSuchFieldException: It is thrown when a class does not contain the field (or variable) specified
- NoSuchMethodException: It is thrown when accessing a method which is not found.
- NullPointerException: This exception is raised when referring to the members of a null object. Null represents nothing
- NumberFormatException: This exception is raised when a method could not convert a string into a numeric format.
- RuntimeException: This represents any exception which occurs during runtime.
- StringIndexOutOfBoundsException: It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

(c) Amity University Online

Notes

Further Readings:

1. JavaScript: Programming Basics for Absolute Beginners, Nathan Clark
2. Java: A Beginner's Guide, Eighth Edition, Herbert Schildt
3. The Java Programming Language, James Gosling and Ken Arnold
4. Java Performance, Binu John and Charlie Hunt

Check Your Understanding:

Answers

1. runtime errors
2. Stack
3. multiple exceptions
4. throw keyword

(c) Amity University Online

Unit - 2.3 : Throwing Exceptions

Notes

Objectives

At the end of this unit, you will be able to understand:

- How to implement finally, throw & throws
- Basics of throwing exceptions
- Explain Java's built-in Exceptions
- How to write Exception Subclasses
- How to Implement of Exception Handling Mechanism.

Introduction

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

The Java throw keyword is used to throw an exception explicitly.

We specify the exception object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

2.3.1 Implementation of throws, finally and throw

throws

If a method is able of causing an exception that it does not able to handle, it must told this behavior so that callers of the method can protect themselves against that type of exception. Developer does this by including a throws clause in the method's definition. A throws clause lists the all the types of exceptions that a method can throw.

This is essential for all the exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can caused must be defined in the throws clause. If they are not defined, then a compile-time error will occur.

Syntax for throws:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Exception list contains all the possible exception that a program can throw and separated by commas.

(c) Amity University Online

Notes

Example

```
class ThrowsDemo {
static void throwOnedemo() {
System.out.println(“Inside throwOnedemo.”);
throw new IllegalAccessException(“demo”);
}
public static void main(String args[]) {
throwOne();
}
}
```

This particular code does not provide a throws clause hence the program will not compile.

To make the above code to executes developer need to make two changes

1. Developer need to declare that throwsonedemo throws illegalAccessException.
2. Main( ) must declared try catch statement

Correct code will look like this:

```
class ThrowsDemo {
static void throwOnedemo() throws IllegalAccessException {
System.out.println(“Inside throwOnedemo.”);
throw new IllegalAccessException(“demo”);
}
public static void main(String args[]) {
try {
throwOnedemo();
} catch (IllegalAccessException e) {
System.out.println(“Caught “ + e);
                }
    }
}
```

Output:

```
inside throwOnedemo
caught java.lang.IllegalAccessException: demo
```

(c) Amity University Online

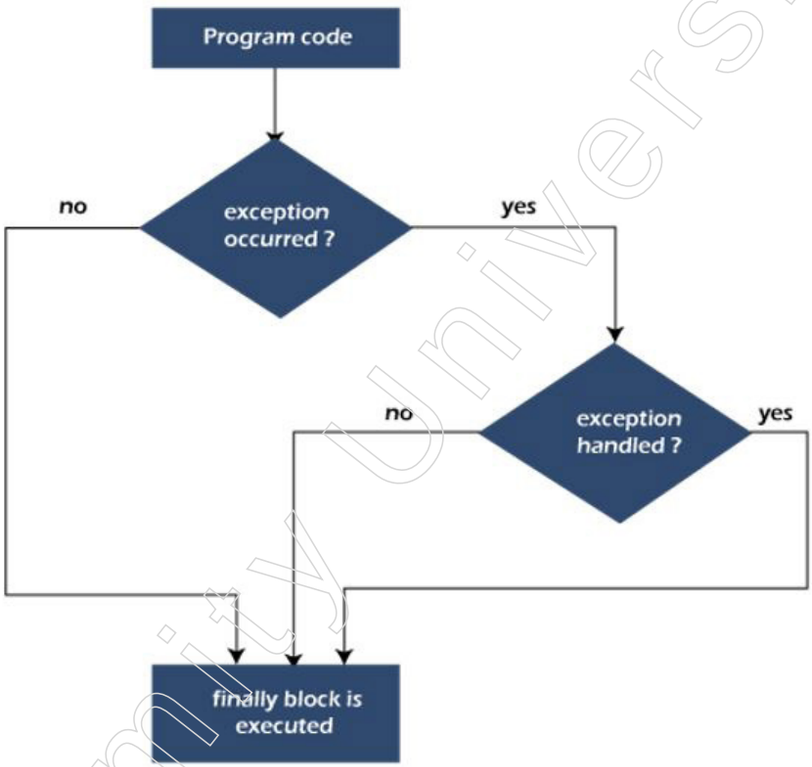
Notes

Finally

When exceptions are thrown, execution in a method takes a rather unexpected and nonlinear way that changes flow through the method. Depending upon how the method is coded by developer, it is also possible for an exception to root the method to return ahead of time. This could be an issue in methods. For example, if a method opens a file in starting and closes it upon the end, then developer will not want the code that closes the file to be through by the exception-handling mechanism. The finally keyword is designed to address this issue.

finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.

Flowchart of finally block



Usage of Finally:

1. **Exception does not occur:**

```
class TestFinallyBlockdemo {
  public static void main(String args[]){
try{
  int data=30/6;
```


(c) Amity University Online

Notes

```
System.out.println(data);
 }
catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally executed");
}
System.out.println("remaining code...");
 }
}
```

Output

```
5
finally block executed
remaining code
```

2. Exception occur but not handled by the catch block:

```
public class TestFinallyBlockdemo{
    public static void main(String args[]){
    try {
System.out.println("Inside try block");
 int data=35/0;
System.out.println(data);
    }
    catch(NullPointerException e){
System.out.println(e);
    }
System.out.println("finally block is executed");
    }
System.out.println("remaining code...");
    }
```

(c) Amity University Online

```
    }
```

Output

Inside Try Block

finally block executed

Exception in thread :main" java.lang.ArithmeticExceptio: /by zero at TestfinallyBlockdemo.main(testfinallyblockdemo.java:9)

3. Exception occurs, handled by the catch block

```
public class TestFinallyBlockdemo{
    public static void main(String args[]){
    try {
System.out.println("Inside try block");
     int data=35/0;
System.out.println(data);
    }
catch(ArithmeticException e){
System.out.println("Exception handled");
System.out.println(e);
    }
    finally {
System.out.println("finally block executed");
    }
System.out.println("remaining code");
    }
  }
```

Output

Inside try block

Exception handled

Java.lang.arithmeticException: /by zero

Finally block executed

Remaining code

Throw

Till know, we have only been catching exceptions which are thrown by the Java

(c) Amity University Online

Notes

run-time system. Though, it is possible for a developer code to throw an exception explicitly, using the throw statement. The syntax of throw:

```
throw ThrowableInstance;
```

ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive type data, such as int, float or double, as well as non-Throwable classes, like String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator.

The flow of program execution stops as soon as the throw statement; any further code statements are not executed. The adjoining try block is checked to see if it has any catch statement that matches the type of exception defined by user. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is checked, and so on. If no similar catch is found, then the default exception handler stops the program and prints the stack trace.

Example of a sample program that creates and throws an exception

```
class ThrowDemo {
static void demoprocthrow() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoprocthrow();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
                }
    }
}
```

This program will get 2 chances to deal with the same issue. First, main( ) sets up an exception context and then calls demoprocthrow( ). The demoprocthrow( ) method then sets up another exceptionhandling context and instantly throws a new instance of NullPointerException, which is caught on the next line.

(c) Amity University Online

Notes

Output

Caught inside demoprocthrow. Recaught: java.lang.NullPointerException: demo

2.3.2 Fundamentals of throwing exceptions

A Java exception is an object that describes an exceptional state that has occurred while running a piece of program code. When an exceptional situation arises, an object which represents that exception is made and thrown within the method that has caused the issue or error. That method may prefer to handle the exception itself, or pass it on. At some point, the exception will be caught and processed. Exceptions are often generated by the Java run-time system, or they will be manually generated by developer code. Exceptions thrown by Java language relate to basic errors that violate the principles of the Java language or the constraints of the Java development environment. Manually generated exceptions are typically wont to report some error condition to the caller of a way.

Java exception handling is handled with the help 5 keywords: try, catch, throw, throws, and finally. Briefly, here is how they work.

- Program statements that you simply want to watch for exceptions are contained within a try block. If an exception occurs within the try block, it's thrown.
- Developer code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are repeatedly thrown by the Java run-time system.
- To manually throw an exception, developer need use the keyword throw.
- Any exception that's thrown out of a way must be specified intrinsically by a throws clause.
- Any code that absolutely must be executed after a try block completes is put during a finally block.

2.3.3 Java's built-in Exceptions

Inside the normal package java.lang, Java describes several exception classes. The most common of these exceptions are subclasses of the ordinary type RuntimeException. Java built in function further divided into two parts i.e Checked and unchecked exception.

Checked Exception:Checked exceptions are also defined compile-time exceptions. These exception are checked by the compiler at time of compiling the code. The compiler make sure that the developer handles the exception or not. The developer need to check the exception upfront else it show the compiler error.

Some of the checked exception defined in java.lang is as follows:

Exception	Meaning
Class Not Found Exception	Class not found.
Clone Not Supported Exception	Attempt to clone an object that does not implement the Cloneable interface.
Illegal Access Exception	Access to a class is denied.
Instantiation Exception	Attempt to create an object of an abstract class or interface.

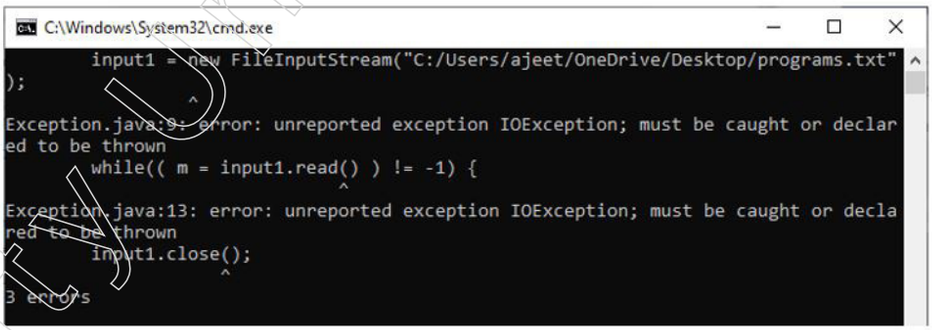
Notes

Interrupted Exception	One thread has been interrupted by another thread.
No Such Field Exception	A requested field does not exists.
No Such Method Exception	A requested method does not exists.

Example for checked exception:

```
import java.io.*;
class CheckedExceptionExample {
   public static void main(String args[]) {
FileInputStreamdeomfile_data = null;
file_data = new FileInputStreamdemo("C:/Users/OneDrive/Desktop/Welcome to java.txt");
      int a;
while(( a = file_data.read() ) != -1) {
System.out.print((char)m);
      }
file_data.close();
   }
}
```

Output:



In the above code, we are trying to read the welcome to java.txt file and display the content of file on the screen. Following are the exception throw by the code:

- The FileInputStreamdemo(File filename) constructor throws the FileNotFoundException that is checked exception.
- The read() method of the FileInputStreamdemo class throws the IOException.
- The close() method also throws the IOException.

These exceptions can be handled in two ways:

1. The exceptions occur in the main method. We can remove these compilation errors by defining the exception in the main method using the throws keywords. We only need to declare the IOException, not FileNotFoundException, because of the

(c) Amity University Online

Notes

relationship between these two exceptions. TheFileNotFoundExceptionclass is the child class ofIOException, so this exception will cover by IOException. Developer need to define in following way

```
class Exception{
   public static void main(String args[])  throws IOException {
      ...
      ...
}
```

If we compile above code exception will be removed.

2. We can also tackle these exception using try-catch method.

```
import java.io.*;
class Exceptiondemo{
   public static void main(String args[]) {
FileInputStreamfile_data = null;
try{
file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs/Hell.txt");
}catch(FileNotFoundExceptionfnfe){
System.out.println("File Not Found!");
      }
      int m;
try{
while(( m = file_data.read() ) != -1) {
System.out.print((char)m);
         }
file_data.close();
}catch(IOExceptionioe){
System.out.println("I/O error occurred: "+ioe);
      }
   }
```

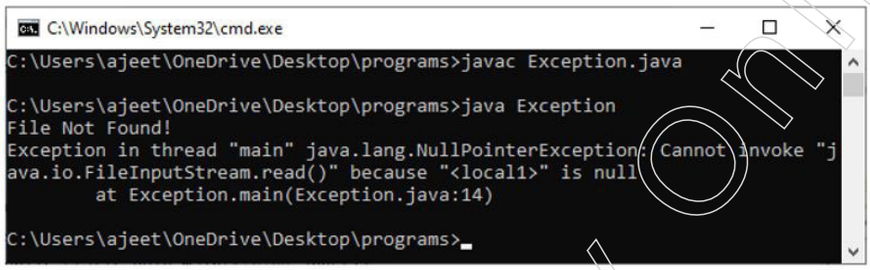
(c) Amity University Online

Notes

}

As per the code if file is not in the location we will get proper error message “File Not Found”

Output



Unchecked Exceptions: The unchecked exceptions are contradictory to checked exceptions. The exceptions will not be checked at compile time by the compiler. If a program shows an unchecked exception, and even if the developer didn’t handle or define it, the code will not give a compilation error. Typically, it occurs when the user gives wrong data through the interaction with the program.

Exception	Meaning
Arithmetic Exception	Arithmetic error, such as divide-by-zero.
Array Index Out Of Bounds Exception	Array index is out-of-bounds.
Array Store Exception	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
Enum Constant Not Present Exception	An attempt is made to use an undefined enumeration value.
Illegal Argument Exception	Illegal argument used to invoke a method.
Illegal Monitor State Exception	Illegal monitor operation, such as waiting on an unlocked thread.
Illegal State Exception	Environment or application is in incorrect state.
Illegal Thread State Exception	Requested operation not compatible with current thread state.
Index Out Of Bounds Exception	Some type of index is out-of-bounds.
Negative Array Size Exception	Array created with a negative size
Null Pointer Exception	Invalid use of a null reference.
Number Format Exception	Invalid conversion of a string to a numeric format.
Security Exception	Attempt to violate security.
String Index Out Of Bounds	Attempt to index outside the bounds of a string.
Type Not Present Exception	Type not found.
Unsupported Operation Exception	An unsupported operation was encountered.

Example for Unchecked Exception:

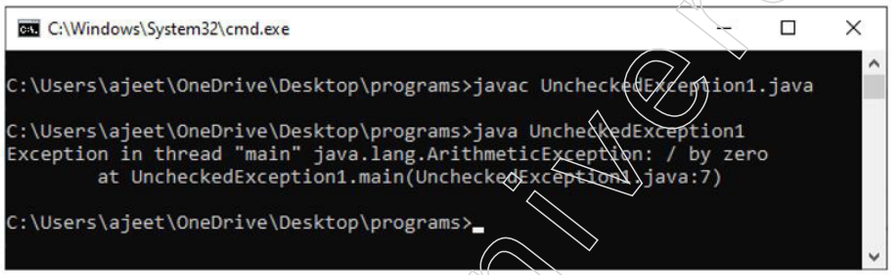
(c) Amity University Online

Notes

```
class UncheckedExceptiondemo {
  public static void main(String args[])
  {
   int a = 35;
   int zero = 0;
   int result = a/zero;
   //Give Unchecked Exception here.
System.out.println(result);
  }
}
```

Output

We have divided a by 0 but at the time of compilation it didn't throw any exception but at the run time it will show arithmeticexception.



Example for unchecked exception in string

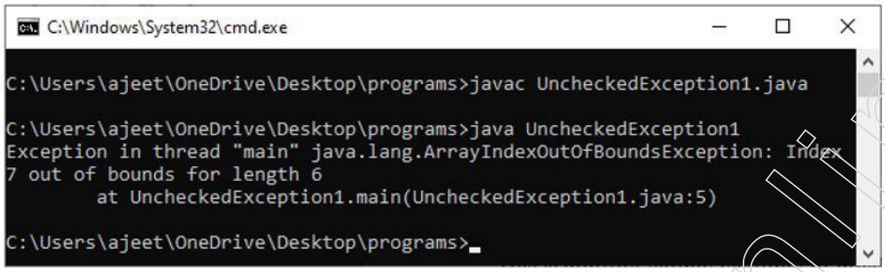
```
class UncheckedException1 {
  public static void main(String args[])
  {
   int num[] ={10,20,30,40,50,60};
System.out.println(num[7]);
  }
}
```

In this example we try to get the item which is on 7th position, but the giver=n array has length of 6. This program will not show any compilation error but at the runt time it will showArrayIndexOutOfBoundsException.

Output:

(c) Amity University Online

Notes



2.3.4 Writing Exception Subclasses

As we know Java's built-in exceptions handle most of the common errors, but sometimes developer will probably want to create his/her own exception types to handle situations specific to his/her applications. This is very simple to do in java developer just needs to declare a subclass of Exception (which is, of course, a subclass of Throwable). Your subclasses don't need to do anything--it's existence in the type system is permits developer to utilize them as exceptions.

The Exception class does not declare any methods by themselves. It only inherit those methods given by Throwable. Therefore, all exceptions including those that a developer states have the methods declared by Throwable available to them. Some common methods defined by throwable are below in table.

Method	Description
Throwable fill in Stack Trace()	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable get Cause()	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String get Localized Message()	Returns a localized description of the exception.
String get Message()	Returns a description of the exception.
Stack Trace Element[] get Stack Trace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
Throwable in it Cause (Throwable cause Exc)	Associates causeExc with the invoking exception as a cause of the invoking exception Returns a reference to the exception.
void print Stack Trace()	Displays the stack trace
void print Stack Trace (Print Stream stream)	Sends the stack trace to the specified stream

(c) Amity University Online

Notes

void print Stack Trace (Print Writer stream)	Sends the stack trace to the specified stream.
void set Stack Trace (Stack Trace Element	Sets the stack trace to the elements passed in elements[]) elements. This method is for specialized applications not normal use.
String to String()	Returns a String object containing a description of the exception This method is called by printin() when outputting a Throwable object.

Developer want to override one or more of these methods in exception classes that he/she creates.

Exception defines 4 constructors. Two were added by JDK 1.4 to support chained exceptions.The other two are:

Exception( )

Exception(String msg)

The first form represents an exception that has no explanation. The second form lets developer specify an explanation of the exception.

Even though explaining a description when an exception is created is frequently helpful, from time to time it is better to override toString( ). The version of toString( ) defined by Throwable (and incorporatedby Exception) first shows the name of the exception and follows by a colon, which is then followed by developers explanation. By overriding toString( ), developer can stop prevent the exception name and colon from being displayed. This will make a code for a cleaner output, which is pleasing in some cases.

Example to show whatever we discussed above:

```
class MyExceptiondemo extends Exception {
private int detail;
MyExceptiondemo(int a) {
detail = a;
}
public String toString() {
return “MyExceptiondemo[“ + detail + “]”;
   }
}
class ExceptionDemo1 {
static void calculate(int a) throws MyExceptiondemo {
System.out.println(“Called calculate(“ + a + “)”);
if(a > 10)
```

(c) Amity University Online

Notes

```
throw new MyExceptiondemo(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
calculate(2);
calculate(22);
} catch (MyExceptiondemo e) {
System.out.println("Caught " + e);
    }
      }
}
```

Output

```
Called calculate(2)
Normal exit
Called calculate(22)
Caught MyException[22]
```

This code declares a subclass of Exception called MyExceptiondemo. It has only one constructor plus an overloadedtoString( ) method that shows the significance of the exception. The ExceptionDemo1 class declares a method named calculate( ) that throws a MyExceptiondemo object. The exception is thrown when calculate( )'s integer parameter is greater than 10. The main( ) method sets up an exception handler for MyExceptiondemo, then calls calaculate( ) with a legal value (less than 10) and an illegal one to show both paths through the code.

Chained Exceptions

Starting with Java Development Kit 1.4, a new characteristic has been put into the exception subsystem: chained exceptions. The chained exception attribute allows developer to attach another exception with an exception. This second exception defines the reason of the first exception.

For example, Think of a situation in which a method throws an ArithmeticException because of user attempt to divide by zero. Though, the actual reason of the trouble was that an I/O error occurred, which raised the divisor to be set improperly. Even though the method mostlythrow an ArithmeticException, as that is the error that happened, developer might also want to let the calling code know that the fundamental cause was an I/O error. Chained exceptions enable developer to handle this, and any other cases in which layers of exceptions presents.

(c) Amity University Online

Notes

To permits chained exceptions, two constructors and methods were added to Throwable directory. Those are:

Throwable getCause( )

Throwable initCause(Throwable causeExc)

The getCause( ) method returns the exception that is within the present exception. If there is no within exception, null is returned. The initCause( ) technique acquaintances causeExc with the invokngexception and returns a reference to the other exception. Thus, a developer can link a cause with an exception after the exception has been shaped. However, it can be set only once. Thus, developer can call initCause( ) only one time for each exception object.

Example for handling chained exceptions:

```
class ChainExceptionsDemo {
static void demoexception() {
NullPointerException e =
new NullPointerException(“top tier”);
// add a cause
e.initCause(new ArithmeticException(“Reason”));
throw e;
}
public static void main(String args[]) {
try {
demoexception();
} catch(NullPointerException e) {
// display top level exception
System.out.println(“Caught: “ + e);
// display cause exception
System.out.println(“Original cause: “ +
e.getCause());
    }
  }
}
```

Output

```
Caught: java.lang.NullPointerException: top tier
Original cause: java.lang.ArithmeticException: Reason
```

(c) Amity University Online

Notes

In above code, the top-tier exception is NullPointerException. To it is added a cause exception, ArithmeticException. When the exception is thrown out of demoexception( ), it is caught by main( ). There, the top-level exception is showed, followed by the within exception, which is obtained by calling getCause( ).

Important point to remember

- Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.
- Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

2.3.5 Implementation of Exception Handling Mechanism.

Exception handling provides developer a powerful mechanism for scheming complex programs that have many vibrant run-time properties. It is significant to think of try, throw, and catch as unsoiled ways to tackle errors and unusual boundary conditions in developers program's logic. In other languages in which error return codes are used to specify failure, Java language uses exceptions. Thus, when a method cannot execute or compile, it throw an exception. This is a nice way to tackle any failure modes.

One last thing to remember Java's exception-handling statements should not be measured as universal mechanism for nonlocal branching. If developers do so, it will get developer code more confusing and make it hard to maintain.

Check your Understanding

1. A Java _____ is an object that describes an exceptional state that has occurred while running a piece of program code.
2. Checked exceptions are also defined as _____ exceptions.
3. The _____ attribute allows developer to attach another exception with an exception.
4. To permits chained exceptions, two constructors and methods were added to Throwable directory. Those are _____ and _____.
5. Exception handling provides developer a powerful mechanism for scheming complex programs that have many vibrant _____ properties.

Summary

- When exceptions are thrown, execution in a method takes a rather unexpected and nonlinear way that changes flow through the method. Depending upon how the method is coded by developer, it is also possible for an exception to root the method to return ahead of time.
- When an exceptional situation arises, an object which represents that exception is made and thrown within the method that has caused the issue or error. That method may prefer to handle the exception itself, or pass it on.

(c) Amity University Online

Notes

- The unchecked exceptions are contradictory to checked exceptions. The exceptions will not be checked at compile time by the compiler. If a program shows an unchecked exception, and even if the developer didn't handle or define it, the code will not give a compilation error. Typically, it occurs when the user gives wrong data through the interaction with the program.
- Java's exception-handling statements should not be measured as universal mechanism for nonlocal branching. If developers do so, it will get developer code more confusing and make it hard to maintain.

Activity

1. Create a program using java built in exception?

Questions and exercises

1. How to implement finally, throw & throws?
2. Define the fundamentals of throwing exceptions?
3. What are Java's built-in Exceptions? And define the various types?
4. How a developer can write Exception Subclasses?
5. Define implementation of Exception Handling Mechanism?

Glossary

- JDK: Java Development Kit

Exception	Meaning
Class Not Found Exception	Class not found.
Clone Not Supported Exception	Attempt to clone an object that does not implement the Cloneable interface.
Illegal Access Exception	Access to a class is denied.
Instantiation Exception	Attempt to create an object of an abstract class or interface.
Interrupted Exception	One thread has been interrupted by another thread.
No Such Field Exception	A requested field does not exists.
No Such Method Exception	A requested method does not exists.

Exception	Meaning
Arithmetic Exception	Arithmetic error, such as divide-by-zero.
Array Index Out Of Bounds Exception	Array index is out-of-bounds.
Array Store Exception	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
Enum Constant Not Present Exception	An attempt is made to use an undefined enumeration value.
Illegal Argument Exception	Illegal argument used to invoke a method.

(c) Amity University Online

Notes

Illegal Monitor State Exception	Illegal monitor operation, such as waiting on an unlocked thread.
Illegal State Exception	Environment or application is in incorrect state.
Illegal Thread State Exception	Requested operation not compatible with current thread state.
Index Out Of Bounds Exception	Some type of index is out-of-bounds.
Negative Array Size Exception	Array created with a negative size
Null Pointer Exception	Invalid use of a null reference.
Number Format Exception	Invalid conversion of a string to a numeric format.
Security Exception	Attempt to violate security.
String Index Out Of Bounds	Attempt to index outside the bounds of a string.
Type Not Present Exception	Type not found.
Unsupported Operation Exception	An unsupported operation was encountered.

Further Readings:

1. JavaScript: Programming Basics for Absolute Beginners, Nathan Clark
2. Java: A Beginner's Guide, Eighth Edition, Herbert Schildt
3. The Java Programming Language, James Gosling and Ken Arnold
4. Java Performance, Binu John and Charlie Hunt

Check Your Understanding:

Answers

1. Exception
2. compile-time
3. chained exception
4. Throwable getCause( )and Throwable initCause(Throwable causeExc)
5. run-time

(c) Amity University Online

Module - 3: Package and Exception in Java

Notes

Structure:

3.1 Introduction to Packages

3.1.1 Constructors

3.1.2 Object & Object Reference

3.1.3 Types of Constructors

3.1.4 Implementation of Constructors in Inheritance

3.1.5 order of invocation of constructors

3.2 Wrapper classes and String class

3.2.1 Introduction to Wrapper Classes

3.2.2 Strings and Characters : Fundamentals of Characters and Strings

3.2.3 The String Class , String Operations

3.2.4 Data Conversion using Value Of ( ) Methods

3.3 StringBuffer class

3.3.1 Fundamental Concept of StringBUffer Class

3.3.2 Properties of StringBuffer Class

3.3.3 Methods of StringBuffer class

3.3.4 Code snippet showing the difference between String and StringBuffer Class

(c) Amity University Online

Notes

Unit - 3.1 : Introduction to Constructor

Objectives

At the end of this unit, you will be able to understand:

- What are Constructors
- Description of Object & Object Reference
- Different types of Constructors in Java
- How to Implement Constructors in Inheritance
- Order of invocation of constructors

Introduction

It can be tedious task to initialize all of the variables in a class each time when a developer creates an instance. Even when developer add convenience functions like setDim( ), it would be easier and more precise to have all of the arrangement done at the time the object creation. Because the specification for initialization is so regular, Java language permits objects to initialize themselves when they are created. This automatic initialization is done with help of the use of a constructor. A constructor initializes an object instantly ahead of creation. It has the similar name as the class in which it stays and is syntactically same to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes. Constructors look a little different because they don't have any return type, not even the void because the inherent return type of a class' constructor is always the class type itself.

3.1.1 Constructors

As the name indicates it is a terminology which has been use construct something in our programs. So basically if we look deeper we forever were defining variables and later on the code we initializing them. Java language being object-oriented developer's goal is to bring code closer to real-world where developer can create objects now somebody is needed to inspect the construction of these objects to get to how they are within working.

Animal Lion = new Animal() ;

Now here Lion is an object and developer cannot set it to 0 or null just not likely int, char, float, etc so essentially we need to initialize these objects as per specification. Developers can have multiple types of constructors.

Rules for writing constructor

1. Constructor(s) of a class must have the same name as the class name in which it resides.
2. A constructor in Java cannot be abstract, final, static and synchronized.
3. Access modifiers can be used in constructor declaration to control its access i.e. which other class can call the constructor.

(c) Amity University Online

Notes

Now you must be thinking that they were unaware of this idea still they were successfully able to execute programs. The answer is simple as there is always a default constructor which have an empty body which is automatically been called, if there were no constructor is written that is the reason behind the successful initialization of objects at the backend. Now you must be wondering if this constructor is calling itself accordingly then why there is the requirement for multiple constructors.

Let's look an example to understand it more:

```
/* Here, Table uses a constructor to initialize the
Dimensions of a Table.
*/
class Table {
double width;
double height;
double depth;
// This is the constructor for Table.
Table() {
System.out.println("Constructing Table");
width = 10;
height = 10;
depth = 10;
}
// calculate and return volume
double volume() {
return width * height * depth;
}
}
class TableDemo {
public static void main(String args[]) {
Table myTable1 = new Table();
Table table = new table ();
double vol;
// get volume of first Table
vol = myTable1.volume();
System.out.println("Volume is " + vol);
```

(c) Amity University Online

Notes

```
// get volume of second Table
vol = myTable2.volume();
System.out.println("Volume is " + vol);
}
}
```

When this program is run, it generates the following results:

```
Constructing Table
Constructing Table
Volume is 1000.0
Volume is 1000.0
```

As you can see, both myTable1 and myTable2 were initialized by the Table( ) constructor when they were created. Since the constructor gives all Tables the same dimensions, 10 by 10 by 10, both myTable1 and myTable2 will have the same volume. The println( ) statement inside Table( ) is for the sake of understanding only. Most of the constructors will not show anything. They will only create an object.

How constructors are different from methods in Java?

- Constructors must have the same name as the class within which it is defined while it is not necessary for the method in java.
- Constructors do not return any type while method(s) have the return type or void if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of time.

3.1.2 Object & Object Reference

When developers create a class, developers are creating a new data type. Developers can utilize this type to define objects of that type. On the other hand, getting objects of a class is a two-step process.

1. Developer must define a variable of the class type. This variable not able to declare an object. in its place, it is only a variable that can be used to refer to an object.
2. Developer must obtain an actual, physical copy of the object and allocate it to that variable. Developer is able to do this using the new operator.

The new operator animatedly assigns (that is, allocates at run time) memory for an object and transfers a reference to it. This reference is, more or less, the location in memory of the object assigned by new. After this reference is stored in the variable. In Java language all class objects must be dynamically assigned.

Example in a previous section, a line similar to the following is used to declare an object of type Table:

```
Table mytable = new Table;
```

(c) Amity University Online

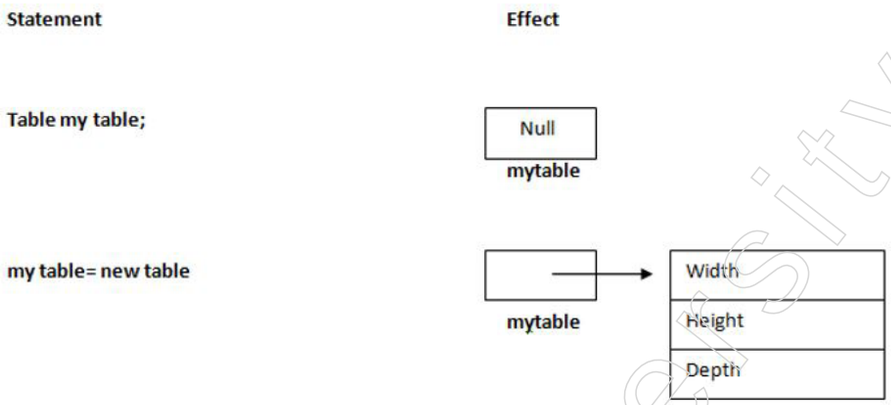
Notes

This statement shows the two steps just described. It can also be written as :

table mytable;

mytabel = new table;

The first line declares myTable as a reference to an object of type Table. After this line executes, myTable contains the value null, which indicates that it does not yet point to an actual object. Any attempt to use myTable at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to myTable. After the second line executes, you can use myTable as if it were a Table object. But in reality, myTable simply holds the memory address of the actual Table object.



A Closer Look at new

As just explained, the new operator dynamically assigns memory for an object. It has this universal form:

class-var = new classsection( );

In this, class-var is a variable of the class type being formed. The classsection is the name of the class that is being instantiated. The class name followed by parentheses defines the constructor for the class.

Now, you might be thinking why you don't need to use new for data types like integers or characters. Because Java's primitive data types are not carry out as objects. Somewhat, they are carry out as "normal" variables. It increases the efficiency of code. Objects have different characteristics and attribute that need Java language to treat them in a different way than it used to treats the primitive data types. By not giving the similar transparency to the primitive types of data that applies to objects, Java language can carry out the primitive types of data more efficiently.

It is significant to recognize that new allocates memory for an object during execution time. The benefit of this thinking is that developer program can create as many or as few objects as it requires during the run time of developer program. Though, since memory is limited, it is likely that new will not be able to assign location for an object because inadequate memory exists. If this happens, a run-time exception will occur. For the sample codes in this book, you won't need to think about running out of memory, but it is important to consider this possibility in real-world programs that you write.

(C) Amity University Online

Notes

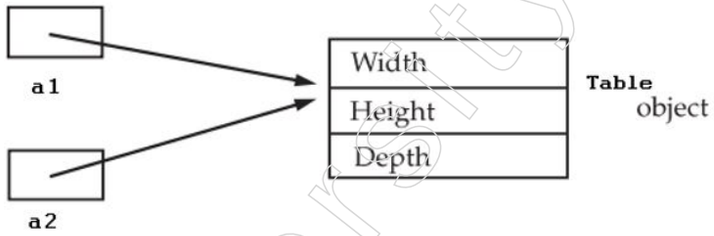
Assigning Object Reference Variables

Object reference variables act in a different way than you might expect when a project takes place. For example, what do you think the following statement does?

Table a1 = new Table();

Table a2 = a1;

You might think that a2 is being assigned a reference to a copy of the object referred to by a1. That is, you might think that a1 and a2 refer to separate objects. But, this would be wrong. In its place, after this statement executes, a1 and a2 will both refer to the same object. The assignment of a1 to a2 did not assign any location copy any part of the original object. It simply makes a2 refer to the same object as a1. So, any changes made to the object through a2 will automatically affect the object to which a1 is referring, since they are the same object. The circumstance is depicted here:



Though a1 and a2 both suggest to the same object, they are not connected in any other way. For example, a subsequent task to a1 will simply unhook a1 from the original object without upsetting the object or upsetting a2. For example:

Table a1 = new Table();

Table a2 = a1;

// ...

a1 = null;

Here, a1 has been set to null, but a2 still shows to the original object.

Important points about Reference variable

1. Reference variable is used to point object/values.

2. Classes, interfaces, arrays, enumerations, and, annotations are reference types in Java. Reference variables hold the objects/values of reference types in Java.

3. Reference variable can also store null value. By default, if no object is passed to a reference variable then it will store a null value.

4. You can access object members using a reference variable using dot syntax.

(c) Amity University Online

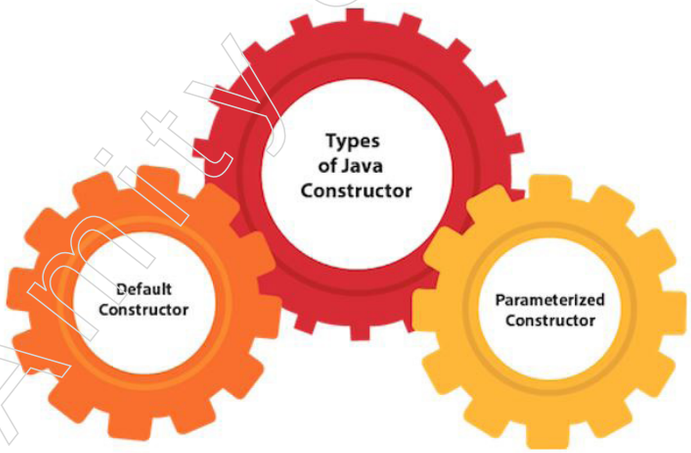
Notes

Comparison between Object and reference:

Parameter of Comparison	Object	Reference
Basic definition	It is the instance of a class and all the elements it contains are based on the blueprint of the class.	A simply memory reference that points to where the object is stored in memory slot.
Format for creation	There is a simple format for creation of an object ClassName reference_variable = new ClassName(with parameter),	The memory reference is created alongside object. It is used in the format for object creation and is given a variable name
Elements	It contains methods and variables based on the class.	It contains a sequence of bits that store the address of the object.
Mutable	Objects have states and behaviors that can be changed, i.e., the state of the object can be changed.	The reference variable value cannot be changed. It can only remain as the data type that it was declared as.
Virtual meaning	It is a real world entity that holds some form of memory or data	It is nothing but a variable name, which has no real meaning. It is like the name of a person that references that person.

3.1.3 Types of Constructors

There are two different types of constructor in Java named:



1. Parameterized Constructors

A constructor that has parameters in it is called as parameterized constructor.

While the Table( ) constructor in the previous example does initialize a Table object, it will be not very useful if all Tables have the different dimensions. What is needed is

(c) Amity University Online

Notes

a way to construct Table objects of various dimensions. The simple way out is to add parameters to the constructor.

For example, the following version of Table defines a parameterized constructor that sets the dimensions of a Table as specified by those parameters.

```
/* Here, Table uses a parameterized constructor todimensions of a Table.
*/
class Table {
double width;
double height;
double depth;
// This is the constructor for Table.
Table(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class TableDemo7 {
public static void main(String args[]) {
Table myTable1 = new Table(5, 20, 15);
Table myTable2 = new Table(4, 6, 9);
double vol;
// get volume of first Table
vol = myTable1.volume();
System.out.println("Volume is " + vol);
// get volume of second Table
vol = myTable2.volume();
System.out.println("Volume is " + vol);
}
```

(c) Amity University Online

Notes

```
}
```

The output from this program is shown here:

Volume is 1500.0

Volume is 216.0

As you can see in previous code, each object is initialized as specified in the parameters to its constructor. For example,

Table myTable1 = new Table(5, 20, 15);

The values 5, 20, and 15 are passed to the Table( ) constructor when new creates the object. Thus, myTable1's copy of width, height, and depth will contain the values 5, 20, and 15, respectively.

Another example for understanding Parameterized Constructors using string:

```
import java.io.*;
class Demo {
    // data members of the class.
    String name;
    int id;
    Demo(String name, int id)
    {
            this.name = name;
            this.id = id;
    }
}
// Class 2
class Demo1 {
    // main driver method
    public static void main(String[] args)
    {
            Demo Demo1 = new Demo("Rituj", 5);
            System.out.println("DemoName :" + Demo1.name
                                        + " and DemoId :" + Demo1.id);
    }
}
```

(c) Amity University Online

Notes

Output

Rituj and Demoid: 5

2. Default Constructor

A constructor that has no parameter in it is known as the default constructor or No-argument constructor. Default constructor gives the default values to the object which are 0, null, etc. based on their type.

Example for a default constructor:

```
import java.io.*;
class demo
{
    int num;
    String name;
    demo()
    {
        System.out.println("Constructor called");
    }
}

Class demo1
{
    public static void main (String[] args)
    {
        // this would invoke default constructor.
        Demo Demo1 = new Demo();
        System.out.println(Demo1.name);
        System.out.println(Demo1.num);
    }
}
```

Output

Contructor called

Null

0

(c) Amity University Online

Notes

The this Keyword

There are times when a method will require referring to the object that invoked it. To permits this, Java language declares the 'this' keyword. 'this' can be used anywhere inside any method to symbolize to the current object. Specifically, 'this'keyword is always a reference to the object on which the method was called. Developer can use 'this' keyword anywhere a reference to an object of the current class type is acceptable.

To better understand what 'this' keyword refers to, consider the following version of Table( ):

```
// A redundant use of this.

Table(double w, double h, double d) {

this.width = w;

this.height = h;

this.depth = d;

}
```

This version of Table( ) operates exactly like the earlier version. The use of 'this' keyword is redundant,but perfectly correct. Inside Table( ), 'this' keyword will always refer to the invoking object. While it isredundant in this case, 'this' keyword is useful in other contexts, one of which is explained in the nextsection.

Instance Variable Hiding

As we know, it is not allowed in Java language to declare two local variables with the similar name insidethe same or enclosing scopes. Fascinatingly, developers can have local variables, including formal parameters to methods, which can be related with the names of the class' instance variables. Still,when a local variable has the similar name as of an instance variable, the local variable hides theinstance variable. This is because in previous example of table width, height, and depth were not used as the names of the parameters to the Table( ) constructor inside the Table class. If they had been, then width wouldhave referred to the formal parameter, hiding the instance variable width. It is usually to easier use different name, there is another solution to this situation. Because this lets developer to refer directly to the object, developer can use this to resolve any name space complication that may arise between instance variables and local variables.

For example, here is another version of Table( ), which uses width, height, and depth for parameter names and then usesthis to access the instance variables by the similar name:

```
Table(double width, double height, double depth) {

this.width = width;

this.height = height;

this.depth = depth;

}
```

(c) Amity University Online

Notes

The use of 'this' keyword in such a case can sometimes be confusing, and some developers are careful not to use any local variables and formal parameter names thathide instance variables. Certainly, other developers believe the contrary—that it is a goodconvention to use the same names for clarity, and use this to overcome the instance variable hiding. It is a matter of choice which approach a developer adopts.

Garbage Collection

Since objects are dynamically assigned by using the new operator, you might be intersted how such objects are destroyed and their memory released for further allocation. In some languages, such as C++, dynamically assigned objects must be manually free by use of a delete operator.

Java language takes an unusual approach; it tackles reallocation for developer automatically. The technique that completes this is called garbage collection. It executes like this: when there no references to an object, that object is considered as that it will no longer required, and the memory engaged by the object can be assigned to someone else.

There is no explicit want to destroy objects as a developer required in C++. Garbage collection only occurs sporadically (if at all) during the execution of a developer program. It will not occur when one or more objects exist that are no longer used by developer. In addition, different Java run-time implementations will take different approaches to garbage collection, but for the most part, developer should not have to consider about it while writing his/her codes.

3.1.4 Implementation of Constructors in Inheritance

While implementing inheritance in a Java program, every class should have its own constructor. Hence the execution of the constructors initiate after the object initialization. It follows a certain series according to the class hierarchy. There can be diverse orders of execution based on the type of inheritance.

1. Order of execution of constructor in Single inheritance:

```
class FatherClass
{
FatherClass()
   {
System.out.println("FatherClass constructor executed.");
   }
}
/* Son Class */
class SonClass extends FatherClass
{
   /* Constructor */
```


(c) Amity University Online

```
SonClass()
   {
System.out.println("SonClass constructor executed.");
   }
}
 public class OrderofExecution1
{
   /* Driver Code */
   public static void main(String ar[])
   {
      /* Create instance of SonClass */
System.out.println("Order of constructor execution...");
      new SonClass();
   }
}
```

Output

```
Order of constructor execution...
ParentClass constructor executed.
ChildClass constructor executed.
```

In code above if we see, after creating an instance of SonClass the FatherClass constructor is invoked first and then the SonClass.

2. Order of execution of constructor in multilevel inheritance

```
class Hospital
{
   /* Constructor */
Hospital()
   {
System.out.println("Hospital constructor executed");
   }
}

class Department extends Hospital
```

(c) Amity University Online

Notes

```
{
    /* Constructor */
Department()
    {
System.out.println("Department constructor executed");
    }
}
class Patient extends Department
{
    /* Constructor */
Patient()
    {
System.out.println("Patient constructor executed");
    }
}
public class OrderofExecution2
{
        /* Driver Code */
    public static void main(String ar[])
    {
        /* Create instance of Patient class */
System.out.println("Order of constructor execution in Multilevel inheritance...");
        new Patient();
    }
}
```

```
Order of constructor execution in Multilevel inheritance...
College constructor executed
Department constructor executed
Student constructor executed
```

In the above code, an instance of Student class is created and it invokes the constructors of Hospital, Department and Patient accordingly.

3. Calling same class constructor using this keyword

```
public class OrderofExecution3
```

(c) Amity University Online

Notes

```
{
    /* Default constructor */
    OrderofExecution3()
    {
        this("CallParam");
System.out.println("Default constructor executed.");
    }
    /* Parameterized constructor */
    OrderofExecution3(String str)
    {
System.out.println("Parameterized constructor executed.");
    }
    /* Driver Code */
    public static void main(String ar[])
    {
        /* Create instance of the class */
System.out.println("Order of constructor execution...");
        OrderofExecution3 obj = new OrderofExecution3();
    }
}
```

Output

```
Order of constructor execution...
Parameterized constructor executed.
Default constructor executed.
```

4. Calling superclass constructor using super keyword

A child class constructor or method can use the base class constructor or method by using the 'super' keyword.

```
/* Parent Class */
class ParentClass
{
    int a;
    ParentClass(int x)
```

(c) Amity University Online

Notes

```
    {
        a = x;
    }
}
/* Child Class */
class ChildClass extends ParentClass
{
    int b;
ChildClass(int x, int y)
    {
        /* Accessing ParentClass Constructor */
        super(x);
        b = y;
    }
    /* Method to show value of a and b */
    void Show()
    {
System.out.println("Value of a : "+a+"\nValue of b : "+b);
    }
}
public class OrderofExecution4
{
    /* Driver Code */
    public static void main(String ar[])
    {
System.out.println("Order of constructor execution...");
ChildClass d = new ChildClass(79, 89);
d.Show();
    }
}
```

(c) Amity University Online

Notes

Output

```
Order of constructor execution...
Value of a : 79
Value of b : 89
```

In the above program, the ChildClass calls the ParentClass constructor using a 'super' keyword that decides the order of execution of constructors.

3.1.5 Order of invocation of constructors

When a class hierarchy is constructed, in which order are the constructors for the classes that create up the hierarchy called. For example, given a subclass called B and a superclass called A, is A's constructor called before B's, or vice versa? The answer is that in a class hierarchy, constructors are invoked in order of derivation, from superclass to subclass. Further, since super( ) must be the first expression to get executed in a subclass' constructor, this order is the similar whether or not super( ) is utilized. If super( ) is not utilized, then the default or parameter less constructor of every superclass will be executed. The below code exemplify when constructors are executed:

```
class ABC {
ABC() {
System.out.println("Inside ABC's constructor.");
}
}
class XYZ extends ABC {
XYZ() {
System.out.println("Inside XYZ's constructor.");
}
}
class MNO extends XYZ {
MNO() {
System.out.println("inside MNO's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
MNO z = new MNO();
}
```

(c) Amity University Online

Notes

```
}
```

Output

```
Inside ABC's constructor
Inside XYZ's constructor
Inside MNO's constructor
```

As we can see, the constructors are invoked in order of derivation.

If you think about it, it creates sense that constructors are executed in order of derivation. Because a superclass has no information about any subclass, any initialization it requires to carry out separate from and possibly prerequisite to any initialization performed by the subclass. Hence, it have to be executed first.

Check your Understanding

1. When developers create a class, developers are creating a new _____ .
2. The new operator dynamically assigns _____ for an object.
3. A constructor that has parameters in it is called as _____ constructor.
4. A constructor that has no parameter in it is known as the _____ .
5. While implementing _____ in a Java program, every class should have its own constructor.

Summary

- A constructor initializes an object instantly ahead of creation. It has the similar name as the class in which it stays and is syntactically same to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- Objects have different characteristics and attribute that need Java language to treat them in a different way than it used to treats the primitive data types. By not giving the similar transparency to the primitive types of data that applies to objects, Java language can carry out the primitive types of data more efficiently.
- There are times when a method will require referring to the object that invoked it. To permits this, Java language declares the 'this' keyword. 'this' can be used anywhere inside any method to symbolize to the current object. Specifically, 'this' keyword is always a reference to the object on which the method was called. Developer can use 'this' keyword anywhere a reference to an object of the current class type is acceptable.
- Java language takes an unusual approach; it tackles reallocation for developer automatically. The technique that completes this is called garbage collection. It executes like this: when there no references to an object, that object is considered as that it will no longer required, and the memory engaged by the object can be assigned to someone else.

(c) Amity University Online

Notes

- While implementing inheritance in a Java program, every class should have its own constructor. Hence the execution of the constructors initiate after the object initialization. It follows a certain series according to the class hierarchy.

Activity

1. Create a program using constructor inheritance in exception?

Questions and exercises

1. Define Constructors?
2. What do you understand by Object & Object Reference?
3. Define various types of Constructors?
4. How to implement constructors in Inheritance?
5. What do you mean by order of invocation of constructors?

Glossary

- JDK: Java Development Kit
- Object: It is the instance of class and all the elements it contains are based on the blueprint of the class.
- Reference: A simple memory reference that points to where the object is stored in memory slot.
- Constructor: Constructor is a terminology which has been use construct something in our programs.

Further Readings:

1. Constructor in Java: Constructor, Bhupendra Singh Mandloi
2. Programming in Java, Sachin Malhotra
3. The Java Programming Language, James Gosling and Ken Arnold
4. Java Performance, Binu John and Charlie Hunt

Check Your Understanding:

Answers

1. data type
2. memory
3. parameterized
4. default constructor
5. inheritance

(c) Amity University Online

Notes

Unit 3.2 : Wrapper classes and String class

Objectives

At the end of this unit, you will be able to understand:

- What are Wrapper Classes
- Basics of Characters and Strings
- How we use string class and operation
- Data Conversion using Value Of ( ) Methods

Introduction

As we know, Java language uses primitive data types (also called simple types), such as int or float, to grasp the essential data types supported by the java language. Primitive data types, rather than objects, are utilized for these quantities just for the sake of performance. Using objects for these values would put in an intolerable overhead to even the easiest of calculations. Thus, the primitive data types are not part of object hierarchy, and they do not able to inherit Object.

Despite the many performance advantage offered by the primitive data types, there are times when a developer will need an object representation. For example, you can't pass a primitive data type by reference to a method. Also, many of the regular data structures implemented by Java operate on objects, which means that developer can't use these data structures to save the primitive data types. To tackle these (and other) situations, Java provides a solution i.e. type wrappers, which are classes that enclose a primitive data type within an object.

3.2.1 Introduction to Wrapper Classes

A Wrapper class is a class whose object wraps or encapsulates primitive data types. When we create an object to a wrapper class, it includes a field and in this field, developer can store primitive data types. The wrapper class in Java provides the instrument to convert primitive data types into object and object into primitive data type.

Use of Wrapper classes in Java

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the java.lang package are recognized as wrapper classes in Java.

(c) Amity University Online

Notes

List of wrapper classes are given below:

Primitive Type	Wrapper Class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Onteger
long	Long
float	Float
double	Double

1. Character

Character is a wrapper for a char. The constructor for Character is

Character(char a)

At this point, 'a' specifies the character that will be wrapped by the Character object being formed.

To get the char value restricted in a Character object, call charValue( ), shown here:

char charValue( )

It will return the enclosed character.

2. Boolean

Boolean is a wrapper for boolean values. It declares these constructors:

Boolean(booleanboolValue)

Boolean(String boolString)

In first statement boolValue is either true or false. In the second statement, if boolString includes the string "true" (in uppercase or lowercase), then the new Boolean object will betrue.ifboolString includes the string "false"then it will be false.

To get a boolean value from a Boolean object, developer needs to use booleanValue( ), as describe below:

booleanbooleanValue( )

It returns the boolean equivalent of the calling object.

3. The Numeric Type Wrappers

Undoubtedly, the most regularly used type wrappers are those that symbolize numeric values.These are Byte, Short, Integer, Long, Float, and Double. All of the numeric type wrappers encapsulate the abstract class Number. Number defines methods that return the value of anobject in each of the different number types. These methods are :

byte byteValue( )

(c) Amity University Online

Notes

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( )

For example, doubleValue( )always returns the value of an object as a double data type, floatValue( ) always returns the value as a float data type, and so on. These methods are used by each of the numeric type wrappers.

All ofthe numeric type wrappers declare the constructors that permit an object to be constructedfrom a given assessment, or a string depiction of that value. For example, here are the constructors declares for Integer data type:

Integer(int a)

Integer(String a)

If a does not include an applicable numeric value, then a NumberFormatException is thrown.

Program to demonstrate the numeric type wrapper:

```
// Demonstrate a type wrapper.
class Wrapdemo {
public static void main(String args[]) {
Integer a = new Integer(30);
int x = a.intValue();
System.out.println(x + " " + a);
}
}
```

This program wraps the integer value 30 inside an Integer object called a. The program then gets this value by calling intValue( ) and stores the result in x.

The procedure of encapsulating a value within an object is called boxing. Thus, in theabovecode,this line boxes the value 100 into an Integer:

Integer a = new Integer(100);

The procedure of extracting a value from a type wrapper is called unboxing. For example, the code unboxes the value in 'a' with this expression:

int x = a.intValue();

Autoboxing

Beginning with Java Development Kit 5, Java language has further added two important attributes: autoboxing and auto-unboxing. Autoboxing is the procedure by

(c) Amity University Online

Notes

which a primitive data type is without human intervention encapsulated (boxed) into its corresponding type wrapper whenever an object of that type is required. There is no requirement to clearly construct an object.

Auto-unboxing is the procedure by which the value of an encapsulated object without human intervention is extracted (unboxed) from a type wrapper when its value is required. There is no requirement to call a method such as doubleValue( ).

The addition of autoboxing and auto-unboxing very much streamlines the coding of more than a few algorithms, removing the boredom of manually boxing and unboxing values. It also helps avert errors. Additionally, it is very significant to generics, which operates barely on objects

With autoboxing it is no longer essential to manually construct an object to wrap a primitive data type. Developer need only allocate that value to a type-wrapper reference. Java automatically constructs the object for the developer. For example: A developer in the modern way to construct an Integer object that has the value 30:

Integer a = 30;

To unbox an object, simply allocates that object reference to a primitive-type variable. For example, to unbox a, developer can use this statement:

int x = a;

Here is the previouscode rewritten to use autoboxing/unboxing:

```
class AutoBoxdemo {
public static void main(String args[]) {
Integer x = 100; // autobox an int
int i = x; // auto-unbox
System.out.println(i + " " + iOb); // displays 100 100
}
}
```

Autoboxing and Methods

Additionally to the simple case of assignments, autoboxing automatically comes whenever a primitive data type must be transformed into an object; auto-unboxing takes place each time an object must be transformed into a primitive data type. Thus, autoboxing/unboxing might come when an argument is transferred to a method, or when a value is returned by a method

Example

```
class AutoBoxdemo {
// Take an Integer parameter and return
// an int value;
static int a(Integer b) {
```


(c) Amity University Online

Notes

```
return b ; // auto-unbox to int
}
public static void main(String args[]) {
// Pass an int to b() and assign the return value
Integer x = b(30);
System.out.println(x);
}
}
```

Output

30

In the program, notice that a( ) defines an Integer parameter and returns an int output. Inside main( ), a( ) is passed the value 30. Because a( ) is expecting an Integer, this value is automatically boxed. Then, a( ) returns the int corresponding of its argument. This causes b to be auto-unboxed. Next, this int value is assigned to x in main( ), which causes the int return value to be autoboxed.

Autoboxing/Unboxing Occurs in Expressions

Usually, autoboxing and unboxing occur whenever a conversion into an object or froman object is necessary. This applies to expressions. Inside an expression, a numeric object isautomatically unboxed. The outcome of the statement is reboxed, if required. For example,consider the following code:

```
// Autoboxing/unboxing occurs inside expressions.
class AutoBoxdemo {
public static void main(String args[]) {
Integer a, b;
int i;
x= 100;
System.out.println(“Original value of x: “ + x);
// The following automatically unboxes x,performs the increment, and then reboxes
// the result back into x.
++x;
System.out.println(“After ++x: “ + x);
// Here, x is unboxed, the expression isevaluated, and the result is reboxed and
// assigned to b.
b = x+ (x/ 3);
```

(c) Amity University Online

Notes

```
System.out.println("b after expression: " + b);
// The same expression is evaluated, but the
// result is not reboxed.
i = x+ (x/ 3);
System.out.println("i after expression: " + i);
}
}
```

The output is shown here:

```
Original value of x: 100
After ++x: 101
b after expression: 134
i after expression: 134
```

In the program, pay special attention to this line:

```
++x;
```

This causes the value in x to be incremented. It works like this: x is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also permits developer to mix different types of numeric objects in a statement. Once the values are unboxed, the normal type promotions and conversions are functional. For example, the following code is perfectly valid:

```
class AutoBoxdemo {
public static void main(String args[]) {
Integer x= 200;
Double y = 98.6;
y = y + x;
System.out.println("Y after expression: " + y);
}
}
```

Output

```
Y after expression: 298.6
```

As you can see, both the Double object y and the Integer object x participated in the addition, and the result was reboxed and stored in y.

Because of auto-unboxing, you can use integer numeric objects to control a switchstatement. For example, consider this fragment:

```
Integer m = 2;
```


© Amity University Online

Notes

```
switch(m) {
case 1: System.out.println("one");
break;
case 2: System.out.println("two");
break;
default: System.out.println("error");
}
```

When the switch expression is checked, m is unboxed and its int value is obtained. As in the code show above, because of autoboxing/unboxing, using numeric objects in a statement is both instinctive and effortless.

Autoboxing/Unboxing Boolean and Character Values

As described earlier, Javalanguage also provisions wrappers for boolean and char. These are Booleanand Character. Autoboxing/unboxing applies to these wrappers as well.

For example

```
// Autoboxing/unboxing a Boolean and Character.
class AutoBoxdemo {
public static void main(String args[]) {
// Autobox/unbox a boolean.
Boolean x = true;
// Below, x is auto-unboxed when used in a conditional expression, such as an if.
if(x) System.out.println("x is true");
// Autobox/unbox a char.
Character ch = 'x'; // box a char
char ch2 = ch; // unbox a char
System.out.println("ch2 is " + ch2);
}
}
```

Output

```
x is true
ch2 is x
```

The most significant thing to notice about this code is the auto-unboxing of x inside the if conditional statement. The conditional statement that controlsan if must estimate to type boolean. Because of auto-unboxing, the boolean value included within

(c) Amity University Online

Notes

x is automatically unboxed when the conditional expression is checked. Therefore, withthe arrival of autoboxing/unboxing, a Boolean object can be used to control an if declaration.

Because of auto-unboxing, a Boolean object can now also be used to control any of Java's language loop expression. When a Boolean is used as the conditional statement of a while, for, ordo/while, it is automatically unboxed into its boolean correspondent. For example, this is nowperfectly valid code:

```
Boolean x;
// ...
while(x) { // ...
```

Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help preventerrors. For example, consider the following program:

```
// An error produced by manual unboxing.
class UnboxingError {
public static void main(String args[]) {
Integer x = 1000; // autobox the value 1000
int y = x.byteValue(); // manually unbox as byte !!!
System.out.println(y); // does not display 1000 !
}
}
```

This code shows not the predictable value of 1000, but –24! The reason is that the valueinsidex is manually unboxed by calling byteValue( ), which causes the truncation of thevalue stored in x, which is 1,000. This results in the garbage value of –24 being allocated to y. Auto-unboxing prevents this type of error because the value in x will always autounbox into a value well-matched with int.

3.2.2 Strings and Characters: Fundamentals of Characters and Strings

In Java language a string is a series of characters. But, unlike many other languages that implement strings as character arrays, Java language implements strings as objects of type String. Executing strings as built-in objects allows Java language to provide a full complement of characteristics that make string handling convenient and easy. For example, Java language has different methods to compare two strings, search for a substring, concatenate two strings, and changes the case of letters within a string.

In addition, String objects can be constructed by developer in number of ways, making it easy to find a string when required.

Somewhat unexpectedly, when developers create a String object, developer is

(C) Amity University Online

Notes

creating a string that cannot be altered. That means once a String object has been created, developer cannot modify the characters that include that string. At first, this may seem to be a grave constraint. Though, it is not the case.

Developers can still perform all kinds of string operations. The difference is that each time developer need a distorted version of an existing string, a new String object is produced that contains the changes. The original string is left unaltered. This perspective is used because fixed, immutable strings can be executed more efficiently than changeable ones. For those cases in which a adaptable string is preferred, Java provides two options: StringBuffer and StringBuilder. Both hold strings that can be customized after they are created.

The String Constructors

The String class supports more than a few constructors. To make an empty String, developer calls the default constructor. For example,

String s = new String();

Above code will create an instance of String with no characters in it.

Developer also wants to construct strings that have initial values. The String class gives a diversity of constructors to handle this. To create a String initialized by an array of characters, use the constructor shown here:

String(char chars[ ])

Example:

char chars[] = { ‘x’, ‘y’, ‘z’ };

String a = new String(chars);

This constructor initializes a with the string “xyz”.

Developer can declare a subrange of a character array as an initializer using the following constructor:

String(char chars[ ], int start, int numCharsdemo)

Here, start specifies the index at which the subrange begins, and numCharsdemo specifies the number of characters to use.

Example:

char charsdemo[] = { ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’ };

String z = new String(charsdemo, 3, 4, 5);

This initializes z with the characters cde .

You can build a String object that includes the same character sequence as another String object using this constructor:

String(String strObj)

Here, strObj is a String object. Consider this example:

(c) Amity University Online

Notes

```
// Construct one String from another.
class MakeStringdemo {
public static void main(String args[]) {
char demo[] = {'B', 'C', 'A', };
String a1 = new String(demo);
String a2 = new String(a1);
System.out.println(s1);
System.out.println(s2);
}
}
```

Output

BCA

BCA

As you can see, a1 and a2 include the same string.

Although Java's language char type uses 16 bits to symbolize the basic Unicode character set, the distinctive arrangement for strings on the internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are general, the String class gives constructors that initialize a string when given a byte array. It be represent as

String(byte asciiChars[ ])

String(byte asciiChars[ ], int startIndex, int numChars)

Above in first statement asciiChars specifies the array of bytes. The second form statement you to specify a subrange. In both these constructors, the byte-to-character conversion is finished with the help of default character encoding of the platform. The following program shows these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
public static void main(String args[]) {
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String a1 = new String(ascii);
System.out.println(a1);
String a2 = new String(ascii, 2, 3);
System.out.println(a2);
}
```

(c) Amity University Online

Notes

```
}
Output
    ABCDEF
CDE
```

Characters

In Java language, the data type characters used to store characters is char. Char in Java language is not the same as char in C or C++. In C/C++ language char is 8 bits wide. But in java language is not the case in Java. Instead, Java language uses Unicode to symbolize characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.

Java language needs 16 bits. Thus, in Java language char is a 16-bit type. The array of a char is 0 to 65,536. There are no negative chars. The normal set of characters known as ASCII still ranges from 0 to 127 as forever, and the extensive 8-bit character set, ISO-Latin-1, range from 0 to 255. Since Java language is designed to permits code to be written for worldwide use, it makes sense that it would use Unicode to symbolize characters. Certainly, the use of Unicode is rather inefficient for languages such as English, French Spanish, or Japanese, whose characters can with any trouble be contained within 8 bits.

For eg:

```
class CharDemo {
public static void main(String args[]) {
char a1, a2;
a1 = 88; // code for X
a2 = 'Y';
System.out.print("a1 and a2: ");
System.out.println(a1 + " " + a2);
}
}
```

Output

a1 and a2: X Y

Character Literals

Characters in Java language are indices into the Unicode character set. They are 16-bit values that can be transformed into integers and used with the integer operators, such as the addition and subtraction operators. A literal character is symbolized inside a pair of single quotes. All of the noticeable ASCII characters can be straight entered inside the quotes, such as 'a', 'z', and '@'. For characters that are not possible to enter straight, there are quite a few escape sequences that permit developer to enter the

(c) Amity University Online

Notes

character developer require, such as '\'' for the single-quote character itself and '\n'for the newline character.

\tEscape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	tab
\b	Backspace

Escape Sequence in character

The constructor for Character is declared as:

Character(char a)

Here, a specifies the character that will be wrapped by the Character object being created.

To get the char value restricted in a Character object, call charValue( ), as shown below:

char charValue( )

It returns the character.

The Character class defines several constants, including the following:

MAX_RADIX	The largest radix
MIN_RADIX	The smallest radix
MAX-VALUE	The largest character value
MIN_VALUE	The smallest character value
TYPE	The Class object for char

Characters also contain several static methods that categorize characters and modify their case.

Method	Description
static boolean isDefined(char ch)	Returns true if ch is defined by Unicode. Otherwise, it returns false.
static boolean isDigit(char ch)	Returns true if ch is a digit. Otherwise, it returns false.
static boolean isIdentifierignorable(char ch)	Returns true if ch should be ignored in an identifier. Otherwise, it returns false.
static boolean isISOControl(char ch)	Returns true if ch is an ISO control character. Otherwise, it returns false.

Notes

static boolean isjavaldentifierPart(char ch)	Returns true if ch is allowed as part of a Java identifier (other than the first character) Otherwise, it returns false.
static boolean isJavaldentifierStart(char ch)	Returns true if ch is allowed as the first character of a Java identifier Otherwise, it returns false.
static boolean isLetter(char ch)	Returns true if ch is a letter. Otherwise, it returns false.
static boolean isLetterOrDigit(char ch)	Returns true if ch is a letter or a digit. Otherwise, it returns false.
static boolean isLowerCase(char ch)	Returns true if ch is a lowercase letter. Otherwise, it returns false.

Method	Description
static boolean isMirrored(char ch)	Returns mirrored character is one reversed for that displayed right-to-left
static boolean isSpaceChar(char	Returns true Unicode Otherwise, it returns false
static boolean TitleCase(char ch)	Returns true ch Unicode Otherwise, it returns false.
Static boolean is UnicodeldentifierPart(char ch)	Returns true ch allowed identifier (other than the character) Otherwise, it returns false.
Static boolean is UnicodeldentifierStart(char	Returns true ch allowed of Unicode identifier. Otherwise, returns false
Static boolean isUpperCase(char ch)	Returns true ch uppercase letter. Otherwise, it returns false.
Static boolean isWhitespace(char ch)	Returns true ch whitespace returns false.
Static char to lower case (char ch)	Returns lower case equivalent of ch.
Static char to Title Case (char ch)	Returns title case equivalent of ch.
Static char to Upper Case (char ch)	Returns uppercase equivalent of ch.

Example using some of above methods

```
class Demochar {
public static void main(String args[]) {
char x[] = {'m', 'n', '7', '?', 'M', ' '};
for(int y=0; y<x.length; y++) {
if(Character.isDigit(x[y]))
System.out.println(x[y] + " is a digit.");
if(Character.isLetter(x[y]))
System.out.println(x[y] + " is a letter.");
```

(c) Amity University Online

Notes

```
if(Character.isWhitespace(x[y]))
System.out.println(x[y] + " is whitespace.");
if(Character.isUpperCase(x[y]))
System.out.println(x[y] + " is uppercase.");
if(Character.isLowerCase(x[y]))
System.out.println(x[y] + " is lowercase.");
}
}
}
```

Output

```
m is a letter.
m is lowercase.
n is a letter.
n is lowercase.
7 is a digit.
M is a letter.
M is uppercase.
is whitespace.
```

Character declares two methods, forDigit( ) and digit( ), that enable developer to convertbetween integer values and the digits they symbolize.

```
static char forDigit(int a, int rx)
static int digit(char digit, int rx)
```

In first statmentforDigit( ) returns the digit character associated with the value of a. The radix of the conversion is specified by rx. In second statementdigit( ) returns the integer value associated with the given character (which is presumably a digit) according to the specified radix.

Another method declared by Character is compareTo( ), which has the syntax:

```
int compareTo(Character a)
```

It returns zero if the calling object and a have the same value. It returns a negative value if the calling objects has a lower value. Else, it gives a positive value.

3.2.3 The String Class, String Operations

The String Class: String is probably the most regularly used class in Java's language class library. The understandable reason for this is that strings are a very significant part of programming or coding.

(c) Amity University Online

Notes

The first thing to recognize about strings is that every string you generate is actually an object of type String. Even string constants are in fact String objects only. For example:

System.out.println("Welcome to Java");

In the above expression the string "Welcome to Java" is a String constant.

The second thing to recognize about strings is that objects of type String are unchallengeable; once a String object is created, its contents cannot be changed or modified. While this may seem like a severe restriction, but it is not:

If you need to change a string, you can always create a new one that contains the modifications.

Java defines a peer class of String, called StringBuffer, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

Strings can be created in many ways. The simplest is to use a expression like this:

String myString = "Wlecom to Java";

Once developer has constructed a String object, you can utilize it anywhere that a string is permissible. For example, this expression displays welcome to java:

System.out.println(welcome java);

Java declares one operator for String objects: +. It is used to merge two strings. For example, this statement

String myString = "Welcome" + "to" + "Java";

results in myString containing "Welcometo Java"

Letsreate a program where we can use this terminology:

```
// Demonstrating Strings.
class StringDemo {
public static void main(String args[]) {
String a1 = "Welcome";
String a2 = "to Java";
String a3 = a1 + " and " + a2;
System.out.println(a1);
System.out.println(a2);
System.out.println(a3);
}
}
```

Output:

Welcome

(c) Amity University Online

To Java

Welcome and to Java

String Literals

For each string literal in your code, Java language automatically constructs a String object. Thus, developer can use a string literal to start a String object. For example, the following program creates two equivalent strings:

```
char chars[] = { 'x', 'y', 'z' };

String s1 = new String(chars);

String s2 = "xyz"; // use string literal
```

Because a String object is constructed for every string literal, developer can use a string literal any place developer can use a String object. For example, developer can call methods directly on a quoted string as if it was an object reference, as the following expression shows. It calls the length( )method on the string "xyz". As predictable, it prints "3".

```
System.out.println("abc".length());
```

String Concatenation

Usually, Java language does not allow operators to be applied to String objects. The one exclusion to this rule is the + operator, which joins two strings, driving a String object as the result. This allows developer to combine together a series of + operations. For example, the code concatenates three different strings:

```
String age = "50";

String a = "She is " + age + " years old.";

System.out.println(a);
```

Output:

She is 50 year old

This displays the string "He is 9 years old."

One sensible use of string concatenation is found when developer wants to create very long strings.in place of letting the long strings wrap around within your source code, developer can break them intosmaller pieces, using the + to join them. Here is an example:

```
// Using concatenation to prevent long lines.

class ConCat {

public static void main(String args[]) {

String longstring = "Welcome to java " +

"here a very long line that would have " +

"wrapped around. But string concatenation " +
```

(c) Amity University Online

Notes

"stops this.";

System.out.println(longString);

}

}

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightlydifferent version of the earlier example:

int age = 50;

String a = "She is " + age + " years old.";

System.out.println(a);

In this case, age is an int rather than another String, but the output produced is the sameas before. This is because the int value in age is automatically converted into its stringrepresentation within a String object. This string is then concatenated as before. The compilerwill convert an operand to its string equivalent whenever the other operand of the + is aninstance of String.

Be careful when you mix other types of operations with string concatenation expressions,however. You might get surprising results. Consider the following:

String a = "four: " + 2 + 2;

System.out.println(a);

Output

four: 22

You might be thinking of output 4that you probably expected. Operator priority causes the concatenation of"four" with the string corresponding of 2 to take place first. This result is then joinedwiththe string alike of 2 a second time. To complete the integer addition first, developer must use expression, like this:

String s = "four: " + (2 + 2);

Now s contains the string "four: 4"

String Conversion and toString( )

When Java language converts data into its string symbol through concatenation, developer does so by occupation one of the overloaded version of the string conversion method valueOf( ) declaredby String. valueOf( ) is filled to capacity for all the simple types and for type Object. For the simpletypes, valueOf( ) returns a string that includes the human-readable corresponding of the valuewith which it is called. For objects, valueOf( ) request to the toString( ) method on the object

Now, let's inspect the toString( )method, because it is the process by which you can define the string demonstration forobjects of classes that you create.

(c) Amity University Online

Notes

Every class implements toString( ) since it is declared by Object. Nevertheless, the defaultimplementation of toString( ) is rarely enough. For most important classes that a developer create, developer want to override toString( ) and provide his/her own string representations. Luckily,this is easy to do. The toString( ) method has below mention universal form:

String toString( )

To put into practice toString( ), just return a String object that includes the human-readablestring that suitably define an object of developerclass.By overriding toString( ) for classes that developer create, developerpermits them to be fully integratedinto Java's language programming environment. For example, developer can be used in print( ) and println( )expressionand in concatenation statements.

Below program demonstrates this byoverridingtoString( ) for the Table class:

```
// Override toString() for Table class.
class Table {
double width;
double height;
double depth;
Table(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " +
depth + " by " + height + ".";
}
}
class toStringDemo {
public static void main(String args[]) {
Table x = new Table(10, 12, 14);
String z = "Table b: " + b; // concatenate Table object
System.out.println(x); // convert Table to string
System.out.println(z);
}
}
```

(c) Amity University Online

Notes

The output of this program is shown here:

Dimensions are 10.0 by 14.0 by 12.0

Table b: Dimensions are 10.0 by 14.0 by 12.0

As you can see in above code, Table's toString( ) method is automatically called when a Tableobjectis used in a concatenation statements or in a call to println( ).

String Comparison

The String class contains a number of methods that evaluate strings or substrings within strings.

equals( ) and equalsIgnoreCase( )

To measure up to two strings for equality, use equals( ). It has this general figure

booleanequals(Object str)

Here, str is the String object being evaluated with the calling String object. It returns true if the strings include the identical characters in the same order, and false if not. Thecontrast is case-sensitive.

To carry out a evaluation that ignores case differences, call equalsIgnoreCase( ). When it evaluate two strings, it considers A-Z to be the similar as a-z. It has below mention universal form:

booleanequalsIgnoreCase(String a)

Now, a is the String object being checked with the calling String object. It, also, returns true if the strings includes the similar characters in the similar order, and false is the strings are not matching.

Here is an example that demonstrates equals( ) and equalsIgnoreCase( ):

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
public static void main(String args[]) {
String a1 = “Java”;
String a2 = “Java”;
String a3 = “Good-bye”;
String a4 = “JAVA”;
System.out.println(a1 + “ equals “ + a2 + “ -> “ +
a1.equals(a2));
System.out.println(a1 + “ equals “ + a3 + “ -> “ +
s1.equals(a3));
System.out.println(a1 + “ equals “ + a4 + “ -> “ +
```

(c) Amity University Online

Notes

```
a1.equals(a4));
System.out.println(a1 + " equalsIgnoreCase " + a4 + " -> " +
a1.equalsIgnoreCase(a4));
}
}
```

Output

```
Java equals Java -> true
Java equals Good-bye -> false
Java equals JAVA -> false
Java equalsIgnoreCase JAVA -> true
```

regionMatches( )

The regionMatches( ) method evaluate a definite region inside a string with another definite region in another string. There is an overloaded form that permits developer to ignore case in such evaluation. Below mentions the usual forms of representation for these two methods:

```
booleanregionMatches(int startdemo, String a2,
int a2StartDemo, int numCheck)
booleanregionMatches(booleanignoreCase,
int startdemo, String a2,
int a2Startdemo, int numcheck)
```

For both statements, startdemo specifies the Demo at which the region begins within the calling String object. The String being evaluate is specified by a2. The Demo at whichthecomparison will start within a2 is specified by a2Startdemo. The length of the substring being evaluate is passed in numCheck. In the second statement, if ignoreCase is true, the case of the characters willbe not being compared. Otherwise, case is important.

startsWith( ) and endsWith( )

String class defines two routines that are, more or less, expert forms of regionMatches( ).The startsWith( ) method checks whether a given String starts with a specified string. On the other hand, endsWith( ) determines whether the String in statement ends with a particular string. Below mentions there universal forms

```
booleanstartsWith(String a1)
booleanendsWith(String a2)
```

At this time, a1 and a2 is the String being checked? If the string matches the statement, true is returned. else, falseis returned by the code. For example,

(c) Amity University Online

Notes

"Demojava".endsWith("java")

and

"Demojava".startsWith("Demo")

In this case both statement matches hence it will return true.

A second statement of startsWith( ), shown below, its lets developer to specify a starting point:

booleanstartsWith(String a1, int startdemo)

At this time, startdemo specifies the demo into which the calling string at which point the search will need to start. For example,

"Demojava".startsWith("Demo", 3)

returns true.

equals( ) Versus ==

It is significant to recognize that the equals( ) method and the == operator execute twounusual operations. As just explained, the equals( ) method check the characters insidea String object. The == operator evaluates two object references to check whether they referto the matching instance. The following code displays how two different String objects can include the same characters, but references for these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTodemo {
public static void main(String args[]) {
String a1 = "Welcome";
String a2 = new String(a1);
System.out.println(a1 + " equals " + a2 + " -> " +
a1.equals(a2));
System.out.println(a1 + " == " + a2 + " -> " + (a1 == a2));
}
}
```

The variable a1 refers to the String instance created by "Welcome". The object referred to by a2 is created with a1 as an initializer. Thus, the inside of the two String objects are equal,but they are distinct objects. This defines that a1 and a2 do not refer to the similar objects andare, consequently, not ==, as is shown here by the output of the previous example:

Welcome equals Welcome -> true

Welcome == Welcome -> false

(c) Amity University Online

Notes

compareTo( )

Over and over again, it is not adequate to simply know whether the two strings are similar or not. For sorting programs, developer require to know which is less than, equal to, or greater than the next. A string is fewer than another if it comes previous to the other in dictionary order. A string is larger than another if it comes subsequent to the other in dictionary order. The String method compareTo( ) solves this function. Below mention its common form:

int compareTo(String str)

At this time, str is the String being evaluate with the calling String. The result of the evaluation is returned and is interpret, as shown here below:

Value	Meaning
Less than zero	The invoking string is less than str.
Greater than zero	The invoking string is greater than str.
Zero	The two strings are equal.

Here is a program that sorts an array of strings. The developer uses compareTo( ) to define sort ordering for a bubble sort:

```
class SortStringdemo {
static String arr[] = {
“Now”, “is”, “the”, “time”, “for”, “all”, “good”, “men”,
“to”, “come”, “to”, “the”, “aid”, “of”, “their”, “country” };
public static void main(String args[]) {
for(int i = 0; i<arr.length; i++) {
for(int a = i + 1;a <arr.length; a++) {
if(arr[a].compareTo(arr[i]) < 0) {
String m = arr[i];
arr[a] = arr[i];
arr[a] = m;
}
}
System.out.println(arr[i]);
}
    }
}
```

Output

Now

(c) Amity University Online

Notes

aid

all

come

country

for

good

is

men

of

the

the

their

time

to

to

As you can see from the output of this code, compareTo( ) takes into description uppercaseand lowercase letters. The word “Now” came out before everyone because it starts withan uppercase letter N, which shows that it has a lower value in the character, setASCII. If developer want to ignore case dissimilarity them comparing two strings, by using command compareToIgnoreCase( ), as shown below:

int compareToIgnoreCase(String a)

This method returns the same output as compareTo( ), except that case dissimilarly are ignored. Developer might want to try substituting it into the previous program. After doing this, “Now”will no longer be first word in output.

Searching Strings

The String class gives two methods that permit developer to look for a string for a particularcharacter or substring:

- indexOf( ) look for the first incidence of a character or substring.
- lastIndexOf( ) looks for the last incidence of a character or substring.

These two types of methods are overloaded in numerous different ways. In all cases, the methods return the value at which the character or substring was found, and –1 if the value is not found.

To look for the first incidence of a character, use

int indexOf(int a)

To look for the last incidence of a character, use

(c) Amity University Online

Notes

int lastIndexOf(int a)

Here, a is the character being look for.

To lookfor the first or last incidence of a substring, use

int indexOf(String a)

int lastIndexOf(String a)

Here, a declares the substring.

You can declare a starting point for the search using these statements:

int indexOf(int a, int startIndexdemo)

int lastIndexOf(int a, int startIndexdemo)

int indexOf(String a, int startIndexdemo)

int lastIndexOf(String a, int startIndexdemo)

Here, startIndexdemo specifies the index at which point the checking begins. For indexOf( ), thesearch runs from startIndexdemo to the end of the string. For lastIndexOf( ), the search runs fromstartIndexdemo to zero.

Example to show how to use the various index methods

```
class indexOfDemo {
public static void main(String args[]) {
String a = “Now is the time for all good men “ +
“to come to the aid of their country.”;
System.out.println(a);
System.out.println(“indexOf(t) = “ +
a.indexOf(‘b’));
System.out.println(“lastIndexOf(b) = “ +
a.lastIndexOf(‘b’));
System.out.println(“indexOf(the) = “ +
a.indexOf(“the”));
System.out.println(“lastIndexOf(the) = “ +
a.lastIndexOf(“the”));
System.out.println(“indexOf(b, 10) = “ +
a.indexOf(‘b’, 10));
System.out.println(“lastIndexOf(b, 60) = “ +
a.lastIndexOf(‘b’, 60));
System.out.println(“indexOf(the, 10) = “ +
```

(c) Amity University Online

Notes

```
a.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +
a.lastIndexOf("the", 60));
    }
}
```

Output

```
Now is the time for all good men to come to the aid of their country.
indexOf(b) = 7
lastIndexOf(b) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(b, 10) = 11
lastIndexOf(b, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

Because String objects are incontrovertible, at any time developer want to modify a String, developer must either copy it into a StringBuffer or StringBuilder, or developer need use one of the following String methods, which will create a new replica of the string with for developer modifications to complete.

substring( )

You can extract a substring using substring( ). It has two expression. The first is

String substring(int startIndexdemo)

Here, startIndexdemo specifies the index at which the substring will start. This statement returns a replica of the substring that begins at startIndexdemo and runs to the end of the calling string.

The second statement of substring( ) permits developer to identify both the starting and endingindex of the substring:

String substring(int startIndexdemo, int endIndexdemo)

Here, startIndexdemo specifies the starting index, and endIndexdemo specifies the stopping point. The string returned includes all the characters from the start index, up to, but not containing, the ending index.

The following example in which code uses substring( ) to change all instances of one substring withotherinside a string:

(c) Amity University Online

Notes

```
// Substring replacement.
class StringReplacedemo {
public static void main(String args[]) {
String a = “This is a Java test. This is, too.”;
String b = “is”;
String c = “was”;
String result = “”;
int x;
do { // replace all matching substrings
System.out.println(a);
x = a.indexOf(b);
if(x != -1) {
result = a.substring(0, x);
result = result + c;
result = result + a.substring(x + b.length());
a = result;
}
} while(x != -1);
}
}
```

Output

from this program is shown here:

```
This is a Java test. This is, too.
Thwas is a java test. This is, too.
Thwas was a java test. This is, too.
Thwas was a java test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

concat( )

As we study early we join two string using ‘+’ but there is method as well which helps developer do so, Developer can join two strings using concat( ), whose syntax is shown below:

(c) Amity University Online

Notes

String concat(String a)

This method creates a new object that includes the calling string with the contents of a appended to the end. concat( ) performs the same function as +. For example look at below code

```
class Concatdemo {
    public static void main(String args[])
    {
            String a = “Welcome to”;
            a = a.concat(“Java”);

            // Print and display combined string
            System.out.println(a);
    }
}
```

Output

Wecome to Java

Another example of concat()

```
public class concatdemo {
    public static void main(String args[])
    {
            String a1 = “Mechanical-”;
            String a2 = “-Engineering”;
            String a3 = a1.concat(a2);
            System.out.println(a3);
            String a4 = “-Branchl”;
            String a5 = a3.concat(a4);
            System.out.println(a5);
    }
}
```

Output

Mechanical—Engineering

Mechanical—Engineering-Branch

(c) Amity University Online

Notes

If you notice in this example we first join two string a1 and a2 in a3 and then we also join that new string a3 to a4 to produce a5.

replace( )

Thereplace( ) method has two statement. The first statement replaces all occurrences of a particular character in the calling string with another character. It has the following expression:

String replace(char a, char b)

In above statement, a defines the character to be replaced by the character specified by b.

The resulting string is returned. For example,

String a = “Hello”.replace(‘l’, ‘w’);

Output of the above statement will be ‘Hewwo’ word l is replaced by w.

The second statement of replace( ) change one character sequence with another. It has thisgeneralstatement:

String replace(CharSequence original, CharSequence replacement)

Example of replace

1. String replace()

```
public class repdemo1 {
public static void main(String args[]) {
    // Initialising String
    String a = new String(“Welcome to world of Java”);
    System.out.print(“After replacing all w with a : “ );
    System.out.println(Str.replace(‘w’, ‘A’));
    // Using replace to replace characters
    System.out.print(“After replacing all a with D : “ );
    System.out.println(Str.replace(‘a’, ‘D’));
}
}
```

Output

After replacing all w with A: Aelcome to Aorld of Java

After replacing all a with D:Welcome to world of JDvD

2. String replaceAll()

```
public class rep2 {
```


(c) Amity University Online

Notes

```
public static void main(String args[]) {
    String a = new String("Welcome to Java");
    System.out.print("Original String : " );
    System.out.println(a);
    // Using replaceAll to replace regex with replace_str
    System.out.print("After replacing regex with replace_str : " );
    System.out.println(Str.replaceAll("(.*)Java(.*)", "OOPS"));
}
}
```

Output

Original String : Welcome to Java

After replacing regex with replace_str :OOPS

3. String replaceFirst()

```
public class repdemo3 {
public static void main(String args[]) {
    // Initialising String
    String a = new String("Welcome Java to Java");
    System.out.print("Original String : " );
    System.out.println(a);
System.out.print("After replacing 1st occurrence of regex with replace_str : " );
    System.out.println(Str.replaceFirst("Java", "World"));
}
}
```

Output

Original String : Welcome to Java to java

After replacing 1st occurrence of regex with replace_str : Welcome to World for Java

trim( )

The trim( ) method returns a replica of the calling string from which any leading and stragglingwhitespace has been removed. It has this common form:

String trim( )

Here is an example:

(c) Amity University Online

Notes

String a = " Hello World ".trim();

This puts the string "Hello World" into a.

The trim( ) method is reasonably helpful when developer process user commands. For example, the following code prompts the user to enter name of a state and then displays that state'scapital. It uses trim( ) to remove any leading or trailing whitespace that may have unintentionally been given by the user.

```
import java.io.*;
class UseTrim {
public static void main(String args[])
throws IOException
{
BufferedReader a = new
BufferedReader(aInputStreamReader(System.in));
String b;
System.out.println("Enter 'stop' to quit.");
System.out.println("Enter State: ");
do {
b = a.readLine();
b = b.trim(); // remove whitespace
if(b.equals("Illinois"))
System.out.println("Capital is Springfield.");
else if(b.equals("Missouri"))
System.out.println("Capital is Jefferson City.");
else if(b.equals("California"))
System.out.println("Capital is Sacramento.");
else if(b.equals("Washington"))
System.out.println("Capital is Olympia.");
// ...
} while(!b.equals("stop"));
}
}
```

Additional String Method

There is many other string method which are described below in table:

(c) Amity University Online

Notes

Method	Description
int code Point At (int)	Returns the Unicode code point at the location specified by L Added by J2SE 5.
int code Point Before (int i)	Returns the Unicode code point at the location that precedes that specified by . Added by J2SE 5.
int code Point Count (int start, int end)	Returns the number of code points in the portion of the invoking String that are between start and end-1 Added by J2SE 5.
boolean contains (Char Sequence str)	Returns true if the invoking object contains the string specified by str Retums false, otherwise, Added by J2SE 5.
boolean content Equals (Char Sequence str)	Returns true if the invoking string contains the same string as str. Otherwise, returns false. Added by J2SE 5.
boolean content Equals (String Buffer str)	Retums true if the invoking string contains the same sining as str. Otherwise returns false.
static String format (String fmtstr Object args)	Returns a string formatted as specified by fmtstr. (See Chapter 18 for details on formatting) Added by J2SE 5.
static String format(Locale loc, String fmtstr Object args)	Returns a string formatted as specified by fmtstr. Formatting is governed by the locale specified by loc. (See Chapter 18 for details on formatting.) Added by J2SE 5.
boolean matches (string reg Exp)	Returns true if the invoking string matches the regular expression passed in regExp. Otherwise returns false
int offset By Code Points (int start, int num)	Returns the index with the invoking string that is num code points beyond the starting index specified by start. Added by J2SE 5.
String replace First (String reg Exp, String new Str	Retums a string in which the first substring that matches the regular expression specified by regExp is replaced by newStr
String replace All (String reg Exp String newStr)	Returns a string in which all substrings that match the regular expression specified by regExp are replaced by newStr

Method	Description
String [ ] split (String reg Exp)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp
String [ ] split (String reg Exp, int max)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp. The number of pieces is specified by max. If max is negative, then the invoking string is fully decomposed. Otherwise, if max contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If max is zero, the invoking string is fully decomposed

(c) Amity University Online

Notes

Char Sequence sub Sequence (int start index, int stop index)	Returns a substring of the invoking string, beginning at startindex and stopping at stopindex This method is required by the CharSequence interface, which is now implemented by String.

3.2.4 Data Conversion using Value Of ( ) Methods

The valueOf( ) method change data from its interior format into a human-readable statements. It is a static method that is overloaded inside String for all of Java's language built-in types so that each type can be transformed properly into a string. valueOf( ) is also overloaded for type Object, so an object of any class type developer can construct also be used as an argument. Few statements to represent Value of()

static String valueOf(double a)

static String valueOf(long b)

static String valueOf(Object c)

static String valueOf(char d[ ])

As we studied earlier, valueOf( ) is required when a string demonstration of some other type of data is looked-for—example, during concatenation operations. Developer can invoke this method directly with any data type and get a reasonable String expression. All of the simple types are transformed to their common String representation. Any object that developer pass to valueOf( ) will return the outcome of a call to the object's toString( ) method. Actually, developer could just call toString( ) straight and get the same outcome.

For majority of arrays, valueOf( ) returns a quite cryptic string, which define that it is an array of some type. For arrays of char, however, a String object is constructed that includes the characters in the char array. There is a special description of valueOf( ) that permits developer to specify a subset of a char array. It can be represented as:

static String valueOf(abc chars[ ], int startIndexdemo, int numChars)

Here, abc is the array that holds the characters, startIndexdemo is the index into the array of characters at which the desired substring begins, and abcChars specifies the length of the substring.

Example to show working of Value Of( ) Methods:

```
class ValueOfdemo {
    public static void main(String arg[])
    {
            int i = 90;
            double f = 3.56789;
            String a = "91";
            // Returns the string representation of int iNum.
```

(c) Amity University Online

Notes

```
            String sample = String.valueOf(i);
            System.out.println(sample);
            System.out.println(sample + i);
            sample = String.valueOf(f);
            System.out.println(sample);
            System.out.println(a + sample);
        }
    }
```

Output

```
90
9091
3.56789
913.56789
```

Check your Understanding

1. The wrapper class in Java provides the instrument to convert primitive data types into _____ and _____ into primitive data type.
2. The eight classes of the java.lang package are recognized as _____ in Java.
3. _____ might come when an argument is transferred to a method.
4. In Java language a string is a series of _____ .
5. In C/C++ language char is _____ wide.
6. At any time developer want to modify a String, developer must either copy it into a _ _____ .

Summary

- A Wrapper class is a class whose object wraps or encapsulates primitive data types. When we create an object to a wrapper class, it includes a field and in this field, developer can store primitive data types.
- Autoboxing is the procedure by which a primitive data type is without human intervention encapsulated (boxed) into its corresponding type wrapper whenever an object of that type is required. There is no requirement to clearly construct an object.
- Auto-unboxing is the procedure by which the value of an encapsulated object without human intervention is extracted (unboxed) from a type wrapper when its value is required. There is no requirement to call a method such as doubleValue( ).

(c) Amity University Online

Notes

- Java language has different methods to compare two strings, search for a substring, concatenate two strings, and changes the case of letters within a string.
- Characters in Java language are indices into the Unicode character set. They are 16-bit values that can be transformed into integers and used with the integer operators, such as the addition and subtraction operators. A literal character is symbolized inside a pair of single quotes.
- The valueOf( ) method change data from its interior format into a human-readable statements. It is a static method that is overloaded inside String for all of Java's language built-in types so that each type can be transformed properly into a string.

Activity

1. Create a program using Value Of ( ) Methods?

Questions and exercises

1. What do you mean by Wrapper Classes
2. Define Strings and Characters?
3. Define Fundamentals of Characters and Strings?
4. Define The String Class?
5. What are String Operations?
6. What do you mean by Data Conversion using Value Of ( ) Methods ,

Glossary

- The String Class: String is probably the most regularly used class in Java's language class library. The understandable reason for this is that strings are a very significant part of programming or coding.
- regionMatches( ): The regionMatches( ) method evaluate a definite region inside a string with another definite region in another string.

Primitive Type	Wrapper Class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Onteger
long	Long
float	Float
double	Double

(c) Amity University Online

Notes

Further Readings:

1. Fundamentals of Java Programming, Mitsunori Ogihara
2. Fundamentals of Java Programming, Ogihara Mitsunori
3. Think Java, Downey Allen B
4. Java Performance, Binu John and Charlie Hunt

Check Your Understanding:

Answers

1. Object and object
2. wrapper classes
3. autoboxing/unboxing
4. characters
5. 8 bits
6. StringBuffer

(c) Amity University Online

Unit - 3.3 : StringBuffer class

Notes

Objectives

At the end of this unit, you will be able to understand:

- What is StringBuffer Class
- What are the Properties of StringBufferClass
- What are the Methods used in forStringBufferclass
- Difference between String and StringBuffer Class with code snippet

Introduction

StringBuffer is a examine class of String that provides a great deal of the functionality of strings. As you know, String definess fixed-length, unassailable character sequence. On the contrary, StringBuffer shows growable and writeable character sequences. StringBuffer might contain characters and substrings that are entered in the middle or appended to the end. StringBuffer will without human intervention grow to make room for such additions and frequently has more characters preassigned than are actually required, to permit room for growth. Java languages uses both classes heavily, but many developers deal only with String and let Java language influence StringBuffers behind the scenes by using the overloaded + operator.

3.3.1 StringBuffer Class

Java language StringBuffer class is used when a developer wants to create mutable (modifiable) String objects. The StringBuffer class in Java is similar as String class except it is mutable i.e. it can be altered.

Important points to remember

- A string buffer is like a String, but can be modified.
- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
- They are safe for use by multiple threads.
- Every string buffer has a capacity.

StringBuffer Constructors

StringBuffer declares these four constructors:

- StringBuffer( )
- StringBuffer(int size)
- StringBuffer(String str)
- StringBuffer(CharSequence chars)

(c) Amity University Online

Notes

- The default constructor means the first statement (the one with no parameters) holds room for 16 characters without redistribution.
- The second expression accepts an integer argument that clearly sets the size of the buffer.
- The third expression accepts a String argument that sets the first contents of the StringBuffer object and saves room for 16 more characters without reallocation. StringBuffer assign room for 16 additional characters when no explicit buffer length is needed, because reallocation is a expensive process with respect to time. Also, regular reallocations can fragment memory. By assigning room for a few extra characters, StringBuffer lowers the number of reallocations that occur.
- The fourth constructor creates an object that includes the character sequence enclosed in chars.

3.3.2 Properties of StringBuffer Class

java.lang.StringBuffer - A class symbolize a mutable sequence of characters stored in an interior buffer. A request of StringBuffer has 3 important properties:

- Buffer: The storage where the characters are stored.
- Capacity: The size of the storage.
- Length: The number of characters stored in the buffer.

The sequence of characters in the buffer can be manipulated by the following methods:

- append(): Adding more characters to the end of the sequence.
- insert(): Inserting more characters at a given position of the sequence.
- delete(): Removing some characters from the sequence.
- replace(): Replacing some characters in the sequence.
- reverse(): Reversing the positions of all characters in the sequence.
- setLength(): Truncating the sequence or padding the sequence with '\u0000' to make the number of characters equal to the specified value.

The capacity, size of the buffer, may be affected by the following methods:

- append(), insert(), and replace(): May cause the capacity to be increased, if the current capacity is not large enough to hold the increased number of characters.
- setLength(): May cause the capacity to be increased, if the current capacity is smaller than the specified value.
- ensureCapacity(): May cause the capacity to be increased, if the current capacity is smaller than the specified value.

3.3.3 Methods of StringBuffer class

Replace( )

Developer can replace one group of characters with another groupwiyinaStringBuffer object by invoking replace( ). Its expression is shown below:

StringBufferreplace(int Start, int End, String a)

(c) Amity University Online

Notes

The substring being replaced is specified by the indexes Start and End. Thus, the substring at Start through End1 is replaced. The replacement string is passed in – a.The resulting StringBuffer object is returned.

Code demonstrate the uses of replace()

```
class replaceDemo {
public static void main(String args[]) {
StringBuffer a = new StringBuffer("This was a java.");
a.replace(5, 7, "is");
System.out.println("After replace: " + a);
}
}
```

Here is the output:

After replace: This is a Java.

ensureCapacity( )

If you want to pre assign a room for a certain number of characters after a StringBuffer has been cretated, developer can use ensureCapacity( ) to set the range of the buffer. This is useful if developer know in advance that you will be includinga large number of small strings to aStringBuffer. ensureCapacity( )can be expressed as:

```
void ensureCapacity(int range)
```

Above, range specifies the size of the buffer.

length( ) and capacity( )

The current length of a StringBuffer can be found via the length( ) method, while the totalowned capacity can be found through the capacity( ) method.

```
int length( )
int capacity( )
```

Example for using length() and capacity()

```
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer a = new StringBuffer("Welcome");
System.out.println("Character = " + a);
System.out.println("length = " + a.length());
System.out.println("capacity = " + sb.capacity());
}
}
```


(c) Amity University Online

Notes

Output

character = Welcome

length = 7

capacity = 23

Since a is initialized with the string “Hello” when it is constructed, its length is 7. Its capacity is 23 because room for 16 additional characters is automatically added by the strigbuffer.

setLength( )

To set the length of the buffer inside a StringBuffer object, developer can use setLength( ). Itscan be expressed as:

void setLength(int length)

Here, length defines the length of the buffer. This value has to be nonnegative. When developer extend the size of the buffer, nuli characters are supplementary added to the end of the current buffer. If developer call setLength( ) with a value which is less than the current value returned bylength( ), then the characters will be saved further than the new length will be lost.

charAt( ) and setCharAt( )

The value of a solitary character can be retrieve by using a StringBuffer via the charAt( ) method. Developer can set the value of a character inside a StringBuffer using setCharAt( ). It can be represented as:

char charAt(int abc)

void setCharAt(int abc, char x)

For charAt( ), abc specifies the index of the character need to be obtained. For setCharAt( ),abc specifies the index of the character need to be set, and x defines the new value of that character. For both statements, abc must be nonnegative and must not defines a locationbeyond the end of the buffer.

The following example demonstrates charAt( ) and setCharAt( ):

```
// Demonstrate charAt() and setCharAt().

class setCharAtDemo {

public static void main(String args[]) {

StringBuffer a = new StringBuffer(“Hello”);

System.out.println(“buffer before = “ + a);

System.out.println(“charAt(1) before = “ + a.charAt(1));

sb.setCharAt(1, ‘i’);

sb.setLength(2);

System.out.println(“buffer after = “ + a);
```


(c) Amity University Online

Notes

```
System.out.println("charAt(1) after = " + a.charAt(1));
}
}
```

Output:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars( )

To create replica of a substring using aStringBuffer into an array, developer uses the getChars( ) method. It can be represented as:

```
void getChars(int Start, int End, char collect[ ],
int targetStartdemo)
```

Here, Start specifies the index of the starting of the substring, and Endspecifiesan index that is one past the end of the defined substring. This states that the substring includes the characters from Start through End–1. The array that will collect the characters is specified by collect. The index inside target at which the substring will be clichédis conceded in targetStartdemo. It must need to be taken care to assure that the collect array is large enough tograsp the number of characters in the specified substring.

append( )

The append( ) method concatenates the string expression of any other type of data to the end of the callingStringBuffer object. It has more than a few overloaded versions. Here are a fewof its expression:

```
StringBufferappend(String str)
StringBufferappend(int num)
StringBufferappend(Object obj)
```

String.valueOf( ) is called for each parameter to obtain its string depiction. The output is appended to the current StringBuffer object. The buffer itself is returned by eachdescription of append( ). This permits following calls to be chained together, as shown in the below code:

```
// Demonstrate append().
class appendDemo {
public static void main(String args[]) {
String a;
int i = 45;
```


© Amity University Online

Notes

```
StringBuffer a = new StringBuffer(40);

a = a.append("i = ").append(i).append("!").toString();

System.out.println(a);

}

}
```

Output

```
i = 45!
```

The append( ) method is most frequently called when the + operator is being used on String objects. Java language automatically changes it to a String instance into parallel operations on aStringBuffer instance. Therefore, a concatenation calls append( ) on a StringBufferobject.After the concatenation has been executed, the compiler inserts a call to toString( ) to transform the modifiable StringBuffer back into a constant String.

All of this may seem irrationally complex. Why not just have only one string class and have it act more or less likeStringBuffer? The counter is performance. There are a lot of optimizations that the Java runtime can make significant that String objects are immutable. Fortunately, Java language hides most of the difficulty of conversion between Strings and StringBuffers. Actually, a lot of developer will never feel the requirement to use StringBuffer directly and will be able to represent most operationsin terms of the + operator on String variables.

insert( )

The insert( ) method inserts one string into other string. It is overloaded to recognize values of all the simple types, plus Strings, Objects, and CharSequences. Like append( ), it calls String.valueOf( ) to attain the string representation of the value it is called with. This string is then inserted into the callingStringBuffer object. These are a few of its representation:

```
StringBufferinsert(int abc, String str)

StringBufferinsert(int abc, char ch)

StringBufferinsert(int abc, Object obj)
```

Here, abc specifies the index at which point the string will be inserted into the calling StringBuffer object.

The following code inserts "to" between "Welcome" and "Java":

```
class insertDemo {

public static void main(String args[]) {

StringBuffer a = new StringBuffer("Welcome Java!");

a.insert(2, "to ");

System.out.println(a);
```


(c) Amity University Online

Notes

```
}
}
```

Output:

```
Welcome to java
```

reverse( )

Developers can change the direction of the characters inside a StringBuffer object using reverse( ), It represent as:

```
StringBufferreverse()
```

This method returns the reversed object on which it was cinvoked.

Code to demonstrate the use of reverse

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
public static void main(String args[]) {
StringBuffer a = new StringBuffer("uvwxyz");
System.out.println(a);
a.reverse();
System.out.println(a);
}
}
```

Output

```
uvwxyz
zyxwvu
```

delete( ) and deleteCharAt( )

Developer can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ). These methods are representing below:

```
StringBufferdelete(int start, int end)
StringBufferdeleteCharAt(int loc)
```

The delete( ) method deletes a sequence of characters from the calling object. Here, Start specifies the index of the first character which needs to be remove, and End specifies an index one past the last character that has to be remove. Therefore, the substring deleted runs from Start to End–1. The outputStringBuffer object is returned. ThedeleteCharAt( ) method deletes the character at the index particular by loc. It returns the outputStringBuffer object.

Code that show usage the delete( ) and deleteCharAt( ) methods:

(c) Amity University Online

Notes

```
class deleteDemo {
public static void main(String args[]) {
StringBuffer a = new StringBuffer("This is a Exam .");
a.delete(4, 7);
System.out.println("After delete: " + a);
a.deleteCharAt(0);
System.out.println("After deleteCharAt: " + a);
}
}
```

Output:

```
After delete: This a exam.
After deleteCharAt: his a exam.
```

substring( )

Developer can obtain a part of a StringBuffer by calling substring( ). It has the following two expressions:

```
String substring(int start)
String substring(int start, int End)
```

The first form returns the substring that starts at start and runs to the end of thecallingStringBuffer object. The second expression returns the substring that starts at start and runs through End–1.

Aside from subSequence( ), which used a method needed by the CharSequenceinterface, the other methods permits a StringBuffer to be looked for an occurrence of a String.

The m demonstrates indexOf( ) and lastIndexOf( ):

```
class IndexOfDemo {
public static void main(String args[]) {
StringBuffer a = new StringBuffer("one two one");
int b;
b = a.indexOf("one");
System.out.println("First index: " + b);
B = a.lastIndexOf("one");
System.out.println("Last index: " + b);
}
```

(c) Amity University Online

}

Output

First index: 0

Last index: 8

Additional StringBuffer Methods

In addition to those methods we just discussed, StringBuffer includes several others methods as well. They are briefed in the following table:

Method	Description
int offset By Code Points (int start, int num)	Returns the index with the invoking string that is num code points beyond the starting index specified by start. Added by J2SE 5
Char Sequence subsequence (int start index, int stop Index)	Returns a substring of the invoking string, beginning at startindex and stopping at stopindex. This method is required by the CharSequence interface, which is now implemented by StringBuffer
void trimTo Size()	Reduces the size of the character buffer for the invoking object to exactly fit the current contents. Added by J2SE 5.

Method	Description
StringBuffer appendCodePoint(int ch)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned. Added by J2SE 5.
int codePointAt(int)	Returns the Unicode code point at the location specified by I. Added by J2SE 5.
int codePointBefore(int i)	Returns the Unicode code point at the location that precedes that specified by I. Added by J2SE 5.
int codePointCount(int start, int end)	Returns the number of code points in the portion of the invoking String that are between start and end-1. Added by J2SE 5.
int indexOf(String str	Searches the invoking StringBuffer for the first occurrence of str. Returns the index of the match, or -1 if no match is found.
int indexOf(String str, int startindex)	Searches the invoking StringBuffer for the first occurrence of str, beginning at startindex Returns the index of the match, or -1 if no match is found.
int lastindexOf(String str)	Searches the invoking StringBuffer for the last occurrence of str Returns the index of the match, or -1 if no match is found
int lastindexOf(String str, int startindex)	Searches the invoking StringBuffer for the last occurrence of str beginning at startindex. Returns the index of the match, or -1 if no match is found

© Amity University Online

Notes

String Builder Class:The StringBuilder in Java language represents a changeable sequence of characters. Since the String Class in Java creates an unchallengeable sequence of characters, the StringBuilder class gives an option to String Class, as it constructs a mutable sequence of characters. The function of StringBuilder is very much related to the StringBuffer class, as both of them provide a substitute to String Class by making a changeable sequence of characters.

The StringBuilder class provides no assurance of synchronization whereas the StringBuffer class provides synchronization. So this class is designed for use as a drop-in substitute for StringBuffer in places where the StringBuffer was being utlized by a single thread. Wherever possible, it is recommended that this class be used in first choice to StringBuffer as it will be quicker under most use cases.

It can be representing as:

public final class StringBuilder

extends Object

implements Serializable, CharSequence

Constructors in Java StringBuilder:

StringBuilder(): Constructs a string builder with no characters in it and an initial capacity of 16 characters.

StringBuilder(int capacity): Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.

StringBuilder(CharSequence seq): Constructs a string builder that contains the same characters as the specified CharSequence.

StringBuilder(String str): Constructs a string builder initialized to the contents of the specified string.

Stringbuilder example:

```
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;
public class abc {
    public static void main(String[] argv)
            throws Exception
    {
            StringBuilder a= new StringBuilder();
            a.append(“ABC”);
            System.out.println(“String = “+ a.toString());
            StringBuilder a1= new StringBuilder(“XXXYYYZZZ”);
```


(c) Amity University Online

Notes

```
        System.out.println("String1 = " + a1.toString());
        StringBuilder a2= new StringBuilder(10);
        System.out.println("String2 capacity = "+ a2.capacity());
        StringBuilder a3= new StringBuilder(a1.toString());
        System.out.println("String3 = " + a3.toString());
    }
}
```

Output

```
String = ABC
String1 = XXXXYYYYZZZZ
String2 capacity = 10
String3 = XXXXYYYYZZZZ
```

Methods in Java StringBuilder:

Method	Description
public String Builder append (String s)	It is used to append the specified string with this string The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public String Builder insert (int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public String Builder replace (int startindex, int endIndex, String str)	It is used to replace the string from specified startindex and endindex.
public String Builder delete (int start index, int end index)	It is used to delete the string from specified startindex and endIndex
public String Builder reverse()	It is used to reverse the string
public int capacity()	It is used to return the current capacity
public void ensure Capacity (int minimum Capacity)	It is used to ensure the capacity at least equal to the given minimum.
public char char At (int index)	It is used to return the character at the specified position
public int length()	It is used to return the length of the string ie total number of characters
public String substring (int begin Index)	It is used to return the substring from the specified beginIndex
public String substring (int beginindex, int endindex)	It is used to return the substring from the specified beginindex and endindex.

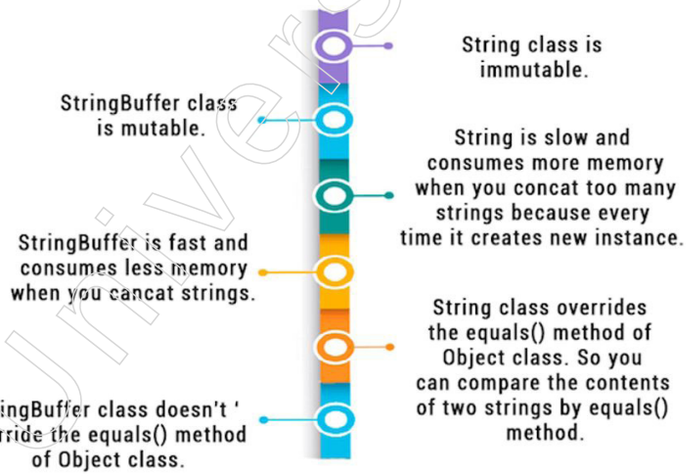
(c) Amity University Online

Notes

3.3.4 Code snippet showing the difference between String and StringBuffer Class

String	String Buffer
The String class is immutable.	The StringBuffer class is mutable.
String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
String class overrides the equals() method of Object class So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override equals() method of Object class.
String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation..
String class uses String constant pool.	StringBuffer uses Heap memory

String Buffer vs String



Performance Test of String and StringBuffer

```
public class ConcatTestdemo{
  public static String concatString()   {
        String a = “Java”;
        for (int x=0; x<10000; x++){
a = a + “Tpoint”;
        }
        return a;
    }
  public static String concatStringBuffer(){
StringBuffer b = new StringBuffer(“Java”);
```

(c) Amity University Online

Notes

```
        for (int x=0; x<10000; x++){
    b.append("Tpoint");
        }
        return b.toString();
    }
    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
    concatString();
    System.out.println("Time taken by Concating with String: "+(System.
currentTimeMillis()-startTime)+"ms");
    startTime = System.currentTimeMillis();
    concatStringBuffer();
    System.out.println("Time taken by ConcatingwithStringBuffer: "+(System.
currentTimeMillis()-startTime)+"ms");
    }
}
```

Output:

```
Time taken by Concating with String: 578ms
Time taken by ConcatingwithStringBuffer: 0ms
```

String and StringBufferHashCode Test

```
public class InstanceTest{
    public static void main(String args[]){
System.out.println("Hashcode test of String:");
        String a="Java";
System.out.println(a.hashCode());
a=a+"tpoint";
System.out.println(a.hashCode());

System.out.println("Hashcode test of StringBuffer:");
StringBuffer b=new StringBuffer("Java");
System.out.println(b.hashCode());
```

(c) Amity University Online

Notes

```
b.append("tpoint");
System.out.println(b.hashCode());
   }
}
```

Output

```
Hashcode test of String:
3254818
229541438
Hashcode test of StringBuffer:
118352462
118352462
```

Check your Understanding

1. Java language StringBuffer class is used when a developer wants to create mutable (modifiable) _____ objects.
2. The number of characters stored in the buffer is known as _____ .
3. To create replica of a substring using aStringBuffer into an array, developer uses the _____ method.
4. The append( ) method is most frequently called when the _____ is being used on String objects.
5. Developer can delete characters within a StringBuffer by using the methods _____ and _____ .

Summary

- StringBuffer is a peer class of String that provides a lot of string functionality. String, as you may know, represents fixed-length, immutable character sequences. StringBuffer, on the other hand, represents growable and writeable character sequences.
- Characters and substrings can be inserted in the middle or appended to the end of a StringBuffer.
- StringBuffer will automatically expand to make room for such additions, and it frequently has more characters preallocated than are actually required to allow for growth.
- The insert( ) method inserts one string into other string. It is overloaded to recognize values of all the simple types, plus Strings, Objects, and CharSequences. Like append( ), it calls String.valueOf( ) to attain the string representation of the value it is called with. This string is then inserted into the callingStringBuffer object.

(c) Amity University Online

Notes

- The function of StringBuilder is very much related to the StringBuffer class, as both of them provide a substitute to String Class by making a changeable sequence of characters.
- The StringBuilder class provides no assurance of synchronization whereas the StringBuffer class provides synchronization. So this class is designed for use as a drop-in substitute for StringBuffer in places where the StringBuffer was being utlized by a single thread.

Activity

1. Create a same program using two different class methods stringbuffer and stringbuilder?

Questions and exercises

1. Define the Concept of StringBuffer Class?
2. What are the properties of StringBuffer Class?
3. What are the methods of StringBuffer class?
4. What are the difference between String and StringBuffer Class?

Glossary

- java.lang.StringBuffer - A class symbolize a mutable sequence of characters stored in an interior buffer. A request of StringBuffer has 3 important properties:
  - Buffer: The storage where the characters are stored.
  - Capacity: The size of the storage.

append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method
capacity()	the total allocated capacity can be found by the capacity( ) method
delete()	Deletes a sequence of characters from the invoking object
deleteCharAt()	Deletes the character at the index specified by loc
ensureCapacity()	Ensures capacity is at least equals to the given minimum.
insert()	Inserts text at the specified index position
length()	Returns length of the string
reverse()	Reverse the characters within a StringBuffer object
replace()	Replace one set of characters with another set inside a StringBuffer object

Further Readings:

1. Fundamentals of Java Programming, Mitsunori Ogihara
2. Fundamentals of Java Programming, Ogihara Mitsunori
3. Think Java, Downey Allen B

(c) Amity University Online

Notes

4. Java Performance, Binu John and Charlie Hunt

Check Your Understanding:

Answers

1. String
2. Length
3. getChars( )
4. + operator
5. delete( ) deleteCharAt( )

(c) Amity University Online

Module - 4 : Interface and Threads in Java

Notes

Structure:

4.1 Introduction to Interface

- 4.1.1 Introduction to Interface
- 4.1.2 Code snippet showing implementation of Interface
- 4.1.3 Reference variable in interface
- 4.1.4 Extending Interface

4.2 Interfcae Methods

- 4.2.1 Type of Methods in interface
- 4.2.2 Default method in interface
- 4.2.3 Static methods in interface
- 4.2.4 Constant variables in interface

4.3 Multithreading

- 4.3.1 Introduction to Multithreading
- 4.3.2 Concept of Thread in Multithreading
- 4.3.3 Lifecycle of Thread
- 4.3.4 Implementing a Thread
- 4.3.5 Thread vs Runnable method of creating thread
- 4.3.6 Methods of Thread class
- 4.3.7 Thread Priorities

4.4 Thread Synchronization

- 4.4.1 Introduction to Synchronization
- 4.4.2 Implementing Synchronization in Thread
- 4.4.3 Benefits and Loopholes of Synchronization
- 4.4.4 Resuming and stopping Threads
- 4.4.5 A code snippet to demonstrate thread with and without synchronization

(c) Amity University Online

Notes

Unit - 4.1 : Introduction to Interface

Objectives

At the end of this unit, you will be able to understand:

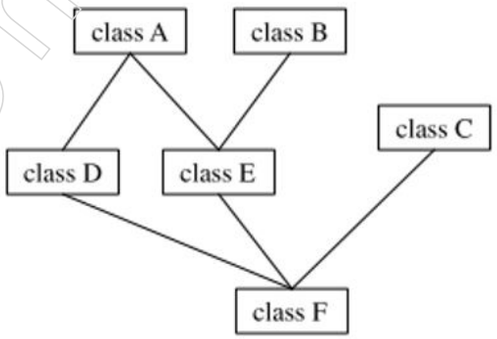
- What is Interface
- Code snippet showing implementation of Interface
- What are Reference variable in interface
- What is Extending Interface

Introduction

In general, an interface is a tool or a system that unrelated entities use to work together. To simplify this, a TV remote is an interface among you and a television set, the English language is an interface between two people, and the etiquette of behavior compulsory in the military is the interface among people of different ranks.

Within the Java programming language, an interface is a kind, just as a class is a type. Like a class, an interface declares methods. Unlike a class, an interface never executes methods; instead, classes that implement the interface execute the methods declared by the interface. A class can employ several interfaces.

A number of object-oriented programming languages, such as C++, let a class to extend two or more superclasses. This is known as multiple inheritance. In the Figure below, for example, class E is shown as have mutually class A and class B as direct superclasses, while class F has three straight superclasses.



Multiple inheritance (NOT allowed in Java)

Such type of inheritance is not permissible in Java. The designers of Java language required to keep the language sensibly simple, and felt that the advantages of multiple inheritance were not worth the cost in amplified difficulty. However, Java language does have an attribute that can be utilized to achieve many of the similar goals as multiple inheritance is known interfaces.

4.1.1 Introduction to Interface

An interface in Java language is a blueprint of a class. It has fixed constants and abstract methods.

(c) Amity University Online

Notes

The interface in Java language is a instrument to achieve abstraction. There can be only abstract methods in the Java language interface, not method body. It is utilized to accomplish abstraction and multiple inheritances in Java language.

To implement an interface, a category must create the entire set of methods defined by the interface. However, each class is liberal to determine the small print of its own implementation. By assigning the interface keyword, Java allows you to completely utilize the “one interface, multiple methods” aspect of polymorphism.

To define an interface, developer requires to use interface keyword. It is used to present total abstraction. That means all the methods in an interface are defined with an blank body and are public and all objects are public, fixed and final by default. A class that employs an interface must implement all the methods defined in the interface. Interface can be expressed as:

```
interface <interface_name> {
// declare constant fields
// declare methods that abstract
// by default.
}
```

Some Key Points:

1. Interfaces specify what a class must do and not how. It is the blueprint of the class.
2. An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move (). So it specifies a set of methods that the class has to implement.
3. If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
4. A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Interfaces are utilized pervasively in real Java language code. Not every class is connected with an interface, but there are a few superior reasons to transport an interface into the picture.

The point of all this is that, though interfaces are not classes, they are somewhat very same. An interface is very similar to an abstract class, i.e., a class that can never be utilized for creatingobjects, but can be used as a foundation for making subclasses. The subroutinesin an interface are abstract methods, which must be executed in any existing class that employ the interface. And as with abstract classes, even though you can’t createanobject from an interface, developer can define a variable whose type is given by the interface. Forexample, if Drawabledemo is an interface, and if Line and FilledCircledemo are classes that implementDrawabledemo:

```
Drawabledemofigure;
figure = new Line(); // figure now refers to an object of class Line
```


Amity University Online

Notes

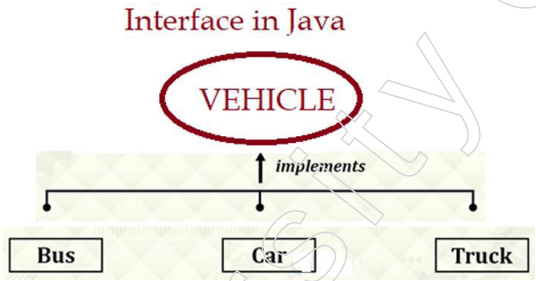
figure.draw(a); // calls draw() method from class Line

figure = new FilledCircledemo(); // Now, figure refers to an object

// of class FilledCircle.

figure.draw(a); // calls draw() method from class FilledCircle

A variable of type Drawable demo can refer to any object of any class that implements the Drawable demo interface. A statement like figure.draw(a), above, is permissible because figure is of type Drawabledemo,and any Drawable object has a draw() method. So, anything object figure symbolize to, thatobject have to have a draw() method.



- Interfaces store condition and details about a class. They include information about the class behavior.
- It includes all the functions that a class can act upon. For example an interface for a car would have to have break (), accelerate (), turn () methods.
- A class whilst executing an interface must realize all the methods within the interface. Nevertheless, it can choose not to employ all the methods if the class itself is abstract.
- One of the well-liked interfaces in Java is the Comparator interface which is helpful for organization containers.

Important Points to remember:

1. We can't create instance (interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
2. A class can implement more than one interface.
3. An interface can extends another interface or interfaces (more than one interface).
4. A class that implements interface must implements all the methods in interface.
5. All the methods are public and abstract. And all the fields are public, static, and final.
6. It is used to achieve multiple inheritance.
7. It is used to achieve loose coupling.

Every implementation class can encompass its own implementation for abstract methods specified in the interface as shown in the below figure.

(c) Amity University Online

Notes

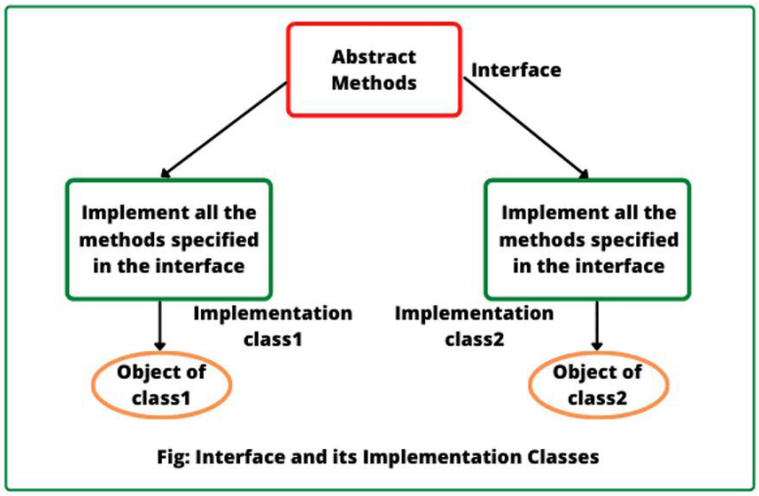


Fig: Interface and its Implementation Classes

While the execution classes will have all the methods with a body, it is promising to construct an instance of implementation classes.

Why developer uses interface:

There are several reasons for using interfaces, we have sum them in below five points:

1. Architect-level people construct interfaces, and then it is given to programmers for writing classes by executing interfaces given.
2. Using interfaces is the optimal way to take away developer's project's API to some other projects. In other words, developers can provide interface methods to the third-party vendors for their execution. To take example from now days, that bank can expose methods or interfaces to various shopping website to implement the payment process .
3. Developers use interface to tailor made features of software for multipurpose use.
4. Interface is used to attain full abstraction in java.
5. By using interfaces, developers can realize the functionality of multiple inheritances.

4.1.2 Code snippet showing implementation of Interface

Example showing implementation of interfaces

```
import java.io.*;
// A simple interface
interface I1
{
    // public, static and final
    final int x = 10;
    void display();
}
```

(c) Amity University Online

Notes

```
{
        public void display()
    {
        System.out.println("Welcome");
    }
        public static void main (String[] args)
    {
        TestClassi = new TestClass();
        i.display();
        System.out.println(x);
    }
}
```

Output:

Welcome

10

Let's consider the example of vehicles like bicycle, car, bike, they have common characteristics. So developer makes an interface and put all these common attributes. And allows Bicycle, Bike, car etc executes all these functionalities in their personal class in their own way.

```
import java.io.*;
interface Vechiledemo {
    void modifyGear(int x);
    void increasespped(int x);
    void putBrakes(int x);
}
class Bicycle implements Vechiledemo{
    int s;
    int g;
    public void modifyGear(int newg){
    g = newg;
    }
    public void increasespped(int increment){
        s = s + increment;
```

(c) Amity University Online

Notes

```
    }
        public void putBrakes(int decrement){
        s = s - decrement;
    }
    public void printStates() {
        System.out.println("speed: " + s
            + " gear: " + g);
    }
}
class Bike implements Vechiledemo {
    int sb;
    int gb;

    public void modifyGear(int nG){
            gb = nG;
    }
    public void increasespped(int increment){

        sb = sb + increment;
    }

    public void putBrakes(int decrement){
        sb = sb - decrement;
    }
    public void printStates() {
        System.out.println("speed: " + sb
            + " gear: " + gb);
    }
}
class XYZ {
        public static void main (String[] args) {
```

(c) Amity University Online

Notes

```
        // creating an inatance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.modifyGear(2);
        bicycle.increasespped(3);
        bicycle.putBrakes(1);
        System.out.println("Bicycle present state :");
        bicycle.printStates();
        // creating instance of the bike.
        Bike bike = new Bike();
        bike.modifyGear(1);
        bike.increasespped(4);
        bike.putBrakes(3);
        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

Output:

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

4.1.3 Reference variable in interface

The name of an interface can be utilized to identify the type of a reference. A reference variable declared using an interface name as its kind can only reference instances of classes that employ that interface. For example, Javalanguagesupports an interface called Runnable. Java language also provides a class called Thread that executes Runnable. It can be represented as:

```
Runnable r;
r = new Thread();
```

The Java language compiler does not lets a value to be allocated to a variable defining using an interface type if not the compiler can be sure that the object referenced by the value implements the particular interface. Invoking a reference variable to an interface type permits the variable to be allocated to that interface type,

(c) Amity University Online

Notes

because the invoke operation provides its own assurance that the object implements the specific interface. Except the compiler is able to establish the actual class of the object that will be referenced at execution time, the cast produces code that verifies at execution time that the object being invoke really does employed the specified interface. At execution time, if the object being invoke does not employ the needed interface, a ClassCastException is thrown.

4.1.4 Extending Interface

In java language, an interface can extend to another interface. When an interface wants to extend to another interface, developer requires to use keyword extends. The interface that extends another interface has its own members and all the members declared in its parent interface too. The class which employ a child interface requires giving code for the methods declared in both child and parent interfaces, or else, it needs to be declared as abstract class.

- An interface can extend another interface.
- An interface cannot extend multiple interfaces.
- An interface can implement neither an interface nor a class.
- The class that implements child interface needs to provide code for all the methods defined in both child and parent interfaces.

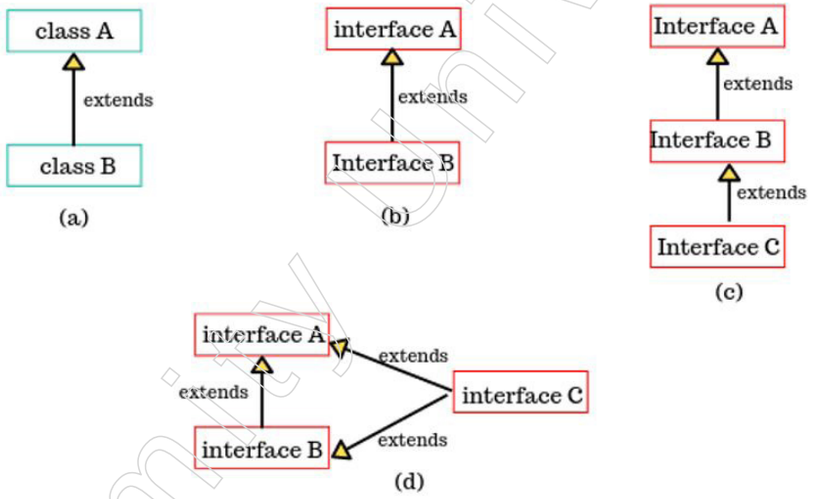


Fig: Various forms of extending Interface in Java

1. Developer can declare all the constants into one interface and methods in other interface. Developer can use constants in classes where methods are not necessary. Letsse how developer can achieve this.

```
interface XYZ
{
 int x = 10;
```


(c) Amity University Online

Notes

```
  int y = 20;


}


interface Abc extends XYZ
{
  void show();
}
```

In this the interface Abc will encapsulate both the constant variable x & y.

2. Developer can also extend various interfaces to a single interface. Its syntax will look like this

```
interface ABC
{
  int x = 20;
  int y = 30;
 }
interface XYZ extends ABC
{
  void show();
 }
interface MNO extends ABC, XYZ
{
  . . . . . .
 }
```

Let's understand interface with the help of program:

```
interfaceParentInterface{
    voidparentMethod();
}


interfaceChildInterfaceextendsParentInterface{
    voidchildMethod();
```

(c) Amity University Online

Notes

```
}
classImplementingClassimplementsChildInterface{
publicvoidchildMethod(){
        System.out.println("Welcome to Java!!");
    }
        publicvoidparentMethod(){
        System.out.println("Interface extend!");
    }
}
publicclassExtendingAnInterface{
    publicstaticvoidmain(String[]args){
        ImplementingClassobj=newImplementingClass();
        obj.childMethod();
        obj.parentMethod();
    }
}
```

Output

Welcome to java!!

Interface extend!

Check your Understanding

1. The interface in Java language is a instrument to achieve _____ .
2. To define an interface, developer requires to use _____ keyword.
3. Java language also provides a class called _____ that executes Runnable.
4. When an interface wants to extend to another interface, developer requires to use keyword _____ .

Summary

- An interface is a kind, just as a class is a type. Like a class, an interface declares methods. Unlike a class, an interface never executes methods; instead, classes that implement the interface execute the methods declared by the interface. A class can employ several interfaces.
- To implement an interface, a category must create the entire set of methods defined by the interface. However, each class is liberal to determine the small print of its own implementation. By assigning the interface keyword, Java allows you to

(c) Amity University Online

Notes

completely utilize the “one interface, multiple methods” aspect of polymorphism.

- Interfaces store condition and details about a class. They include information about the class behavior.
- The Java language compiler does not lets a value to be allocated to a variable defining using an interface type if not the compiler can be sure that the object referenced by the value implements the particular interface.
- The interface that extends another interface has its own members and all the members declared in its parent interface too. The class which employ a child interface requires giving code for the methods declared in both child and parent interfaces, or else, it needs to be declared as abstract class.

Activity

1. Create a program using extended interface?

Questions and exercises

1. What do you mean by Interface?
2. How to implement Interface?
3. What do you mean by reference variable in interface?
4. How to Extend Interface?

Glossary

- Reference: A reference variable declared using an interface name as its kind can only reference instances of classes that employ that interface.
- Extended Interface: When an interface wants to extend to another interface, developer requires to use keyword extends.

Further Readings:

1. The Java Native Interface: Programmer’s Guide and Specification, Sheng Liang
2. Essential JNI: Java Native Interface, Rob Gordon
3. The Java Application Programming Interface, Frank Yellin and James Gosling
4. Fundamentals of Java Programming, Mitsunori Ogihara
5. Fundamentals of Java Programming, Ogihara Mitsunori

Check Your Understanding:

Answers

1. abstraction
2. interface
3. Thread
4. extends

(c) Amity University Online

Unit - 4.2 : Interface Methods

Notes

Objectives

At the end of this unit, you will be able to understand:

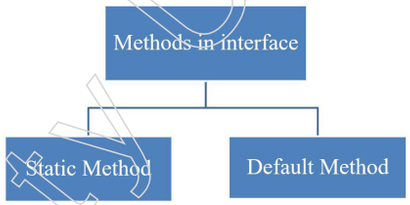
- How many Types of Methods in interface
- What is Default method in interface
- What are Static methods in interface
- What are Constant variables in interface

Introduction

Designing interfaces have forever been a tough task because if we want to add supplementary methods in the interfaces, it will need change in all the employing classes. As interface grows old, the number of classes employing it might develop to an amount that it's not possible to extend interfaces. That's why when designing an application, most of the frameworks give a base employing class and then we expand it and override methods that are appropriate for our application.

Java 8 comes up with a new concept of default method implementation in interfaces. This ability is added for backward compatibility so that old interfaces can be utilized to influence the lambda expression capability of Java 8. For example, the List interface is not having the forEach() method. If they put in this method as a conceptual method then all List implementation class must have to give the implementation since it is common usability is to iterate the list. So they had to bring in this concept to improve features and support for the backward classes.

4.2.1 Types of method in interface



4.2.2 Default method in interface

Before Java 8, interfaces possibly will have only abstract methods. The execution of these methods has to be given in a separate class. So, if a new method is needed to be added in an interface, then its execution code has to be provided in the class implementing the identical interface. To overcome this problem, Java 8 has introduced the concept of default methods which permits the interfaces to have methods with execution without disturbing the classes that execute the interface.

The default methods were instigated to offer backward compatibility so that accessible interfaces can use the lambda statement without executing the methods in the implementation class. Default methods are also called as defender methods or virtual extension methods.

(c) Amity University Online

Notes

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Example for default interface

```
interface TestInterface
{
    public void square(int x);
    // default method
    default void show()
    {
    System.out.println("Checking Default Method");
    }
}
class TestClass implements TestInterface
{
    // implementation of square abstract method
    public void square(int x)
    {
            System.out.println(x*x);
    }
    public static void main(String args[])
    {
            TestClass b = new TestClass();
            b.square(8);

            // default method executed
            b.show();
    }
}
```

Output:

```
Checking Default Method
```

(c) Amity University Online

64

Notes

Default Methods and Multiple Inheritance

If both the implemented interfaces include default methods with same method statement, the implementing class should clearly identify which default method is to be used or it supposed to override the default method.

Example to show multiple inheritance through default method

```
interface TestInterfacedemo
{
    // default method
    default void show()
    {
            System.out.println("DefaultInterfaceMethod");
    }
}
interface TestInterface2
{
    default void show()
    {
            System.out.println("Default Interface Method 2");
    }
}
// Implementation class code
class TestClass implements TestInterface1, TestInterface2
{
    // Overriding default show method
    public void show()
    {
            Testinterfacedemo.super.show();

            TestInterface2.super.show();
    }
    public static void main(String args[])
    {
```

(c) Amity University Online

Notes

```
        TestClass d = new TestClass();
        d.show();
    }
}
```

Output:

Default Interface Method

Default Interface Method1

Important Point to remember

- Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
- Java interface default methods have bridge down the differences between interfaces and abstract classes.
- Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
- Java interface default methods will help us in removing base implementation classes; we can provide default implementation and the implementation classes can chose which one to override.
- One of the major reasons for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
- If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it's because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That's why to avoid confusion, we can't have default methods that are overriding Object class methods.
- Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

4.2.3 Static methods in interface

Static Methods in Interface are those methods, which are declared in the interface with the keyword static. Unlike other methods in Interface, these static methods include the total definition of the function and given that the definition is complete and the method is static, consequently these methods cannot be overridden or misrepresented in the implementation class.

Alike to Default Method in Interface, the static method in an interface can be declared in the interface, but cannot be overridden in execution Classes. To utilize a static method, Interface name should be instantiated with it, as it is a component of the Interface only.

Example 1: To demonstrate use of Static method in Interface.

In this program, a simple static method is declared in an interface which is being called in the main() method of the Implementation Class InterfaceDemo. Unlike the

(c) Amity University Online

Notes

default method, the static method defines in Interface hello(), cannot be overridden in implementing the class.

```
interface NewInterface {
    static void hello()
    {
            System.out.println("Welcome to Java");
    }
    void overrideMethod(String str);
}
public class InterfaceDemo implements NewInterface {

    public static void main(String[] args)
    {
            InterfaceDemointerfaceDemo = new InterfaceDemo();
            NewInterface.hello();
InterfaceDemo.overrideMethod("Override Mathod");
    }
    @Override
    public void overrideMethod(String str)
    {
            System.out.println(str);
    }
}
```

Output

```
Welcome to java
Override Method
```

Example 2: To display Scope of Static method.

```
interface PrintDemo {
    static void hello()
    {
            System.out.println("Interface PrintDemo");
    }
```

(c) Amity University Online

Notes

```
}

public class InterfaceDemo implements PrintDemo {
    public static void main(String[] args)
    {
            PrintDemo.hello();
            // Call Class static method
            hello();
    }
    // Class Static method is defined
    static void hello()
    {
            System.out.println("Invoke from Class");
    }
}
```

Output

Interface PrintDemo

Invoke from Class

Important Point to Remember

1. Java interface static method is part of interface; we can't use it for implementation class objects.
2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Java interface static method helps us in providing security by not allowing implementation classes to override them.
4. We can't define interface static method for Object class methods; we will get compiler error as "This static method cannot hide the instance method from Object". This is because it's not allowed in java, since Object is the base class for all the classes and we can't have one class level static method and another instance method with same signature.
5. We can use java interface static methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use.

4.2.4 Constant variables in interface

Developer can use interfaces to call shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the specified values. Once developer include that interface during a class (that is, once developer "implement" the interface), all of these variable names are going to be in scope as constants. (This is analogous to employing a header enter C/C++ to make an outsized

(c) Amity University Online

number of #defined constants or const declarations.) If an interface contains no methods, then any class that has such an interface doesn't actually implement anything.

Notes

It is as if that class was calling the constant fields into the category name space as final variables.

Below program uses this system to implement an automatic "decision maker":

```
import java.util.Random;
interface SharedConstantsdemo {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstantsdemo {
Random R = new R();
int ask() {
int a = (int) (100 * R.nextDouble());
if (a< 30)
return NO; // 30%
else if (a< 60)
return YES; // 30%
else if (a< 75)
return LATER; // 15%
else if (a< 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMedemo implements SharedConstants {
static void answer(int result) {
```

© Amity University Online

Notes

```
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
    public static void main(String args[]) {
Questiondemo q = new Questiondemo();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Notice that this program makes use of one of Java's regular classes: Random. This class contains pseudorandom numbers and provides them when they call. It provides several methods that allow you to obtain random numbers in the form required by your program. In this program, developer used method nextDouble( ) is used. It returns

(c) Amity University Online

casual numbers between in the range 0.0 to 1.0.

In this sample program, the two classes, Questiondemo and AskMedemo, both implement the SharedConstantsdemo interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly.

Output

Later

Soon

No

Yes

Check your Understanding

1. The interface in Java language is a instrument to achieve _____ .
2. Before Java 8, interfaces possibly will have only _____ methods.
3. Static Methods in Interface are those methods, which are declared in the _____ with the keyword static.
4. Developer can use _____ to call shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the specified values.

Summary

- Java 8 comes up with a new concept of default method implementation in interfaces. This ability is added for backward compatibility so that old interfaces can be utilized to influence the lambda expression capability of Java 8.
- The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.
- We can't create instance(interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface).
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.
- It is used to achieve loose coupling.
- Alike to Default Method in Interface, the static method in an interface can be declared in the interface, but cannot be overridden in execution Classes. To utilize

(c) Amity University Online

Notes

a static method, Interface name should be instantiated with it, as it is a component of the Interface only.

Activity

1. What will be the output of below program

```
import java.io.*;
{
    // public, static and final
    final int a = 10;
    // public and abstract
    void display();
}
{
    public void display()
    {
            System.out.println("Geek");
    }
            public static void main (String[] args)
    {
            TestClass t = new TestClass();
            t.display();
            System.out.println(a);
    }
}
```

Questions and exercises

1. Define different type of Methods in interface?
2. What do you mean by default method in interface?
3. Define static methods in interface?
4. What do you mean by constant variables in interface?

Glossary

- Lambda expression: Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

(c) Amity University Online

Notes

- Interface: Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

Further Readings:

1. The Java Native Interface: Programmer's Guide and Specification, Sheng Liang
2. Essential JNI: Java Native Interface, Rob Gordon
3. The Java Application Programming Interface, Frank Yellin and James Gosling

Check Your Understanding:

Answers

1. abstraction
2. abstract
3. Interface
4. interfaces

(c) Amity University Online

Notes

Unit - 4.3 : Multithreading

Objectives

At the end of this unit, you will be able to understand:

- Define Multithreading
- Understand the Concept of Thread in Multithreading
- Define Lifecycle of Thread
- How to Implement a Thread
- Compare Thread v/s Runnable method of creating thread
- Define Methods of Thread class
- What are Thread Priorities

Introduction

Contrasting to many other computer languages, Java language gives built-in support for multithreaded programming. A multithreaded program includes two or more parts that can execute alongside. Each fraction of such a program is called a thread, and each thread declares a separate way of execution. Thus, multithreading is a dedicated type of multitasking.

Developers are certainly familiar with multitasking, because it is supported by nearly all modern operating systems. Nevertheless, there are two separate types of multitasking: process-based and thread-based.

Multithreading enables developer to write very efficient codes that make maximum use of the CPU, because idle time can be reserved to a minimum. This is significant for the interactive, networked environment in which Java language operates, because inactive time is common. For example, the broadcast rate of data over a system is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a very slower pace than they can be processed by the CPU. And, of course, user contribution is much slower than the computer. In a single-threaded environment, developer program has to wait for each of these everyday jobs to end before it can go on to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain admission to this inactive time and put it to good use.

If developer have programmed for operating systems such as Windows, then you are already recognizable with multithreaded programming. However, the fact that Java language manages threads makes multithreading particularly suitable, because many of the details are handled for you.

4.3.1 Introduction to Multithreading

Multithreading is a Java characteristic that lets simultaneous execution of two or more parts of a program for maximum consumption of CPU. Each part of such program is called a thread. So, threads are light-weight processes inside a process.

(c) Amity University Online

Notes

Though, we use multithreading than multiprocessing because threads use a common memory area. They don't assign separate memory region so saves memory, and context-switching among the threads takes less time than process.

There are two distinct types of multitasking: process based and thread-based. It is important to understand the difference between the two.

Process-based multitasking:

A process is, in core, a code that is executing. Thus, process-based multitasking is the characteristic that lets your computer to run two or more programs alongside. For example, process-based multitasking facilitates you to run the Java compiler at the same time that when you are working on a text editor. In process based multitasking, a program is the smallest unit of code that can be transmitted through the scheduler.

Each process has an address in memory. In other words, each process allocates a separate memory area.

A process is heavyweight.

Cost of communication between the processes is high.

Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

Thread-based multitasking:

The thread is the least unit of dispatch able code. This means that a single code can carry out two or more tasks at the same time. For example, text editors can layout text at the same time that it is printing, providing these two events are being executed by two separate threads. Thus, process-based multitasking engaged with the "big picture," and thread-based multitasking takes care of the details.

Multitasking threads need less overhead than multitasking processes. Processes are hardwearing tasks that want their separate location space. Inter-process communication is luxurious and inadequate. Framework switching from one process to an additional is also expensive.

Alternatively, threads are lightweight. They allocate the similar address space and agreeably share the same heavyweight process. Inter-thread communication is reasonably priced, and context switching among one thread to another is not that costly. While Java language codes make use of process based multitasking environments, process-based multitasking is not beneath the control of Java language.

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

Advantages of Multithreading

1. It doesn't stop the user because threads are self-governing, and developer can carry out multiple operations at the equivalent time.

© Amity University Online

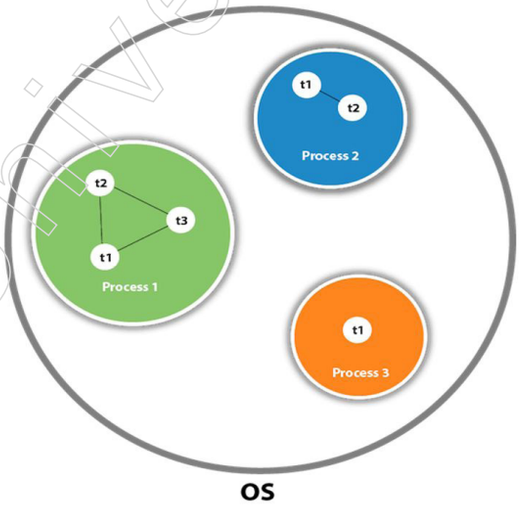
Notes

2. To carry-out many operations at similar time, hence it saves lots o time
3. Threads are independent, so it does not impact other threads if an exception occurs in a single thread.

4.3.2 Concept of Thread in Multithreading

In Java language, a single task is called a thread. The term “thread” refers to a “thread of control” or “thread of execution,” meaning a series of commands that are executed one after another— the thread extends during time, connecting each command to the next. In a multithreaded program, there can be many threads of execution, weaving during time in equivalent and creating the complete framework of the program.

Every Java language program has at least one thread; when the Java virtual machine executes your program, it constructs a thread that is accountable for executing the main routine of the code. This main thread can in turn construct other threads that can carry on even after the main thread has terminated. In a GUI code, there is at least one supplementary thread, which is in charge for managing events and drawing components on the display. This GUI thread is shaped when the first window is opened. So in fact, developer has already done parallel programming! When a main schedule opens a window, together the main thread and the GUI thread can carry on to execute in parallel. Of course, parallel programming can be utilized in much more exciting ways.



As depicted in the figure above, a thread is executed inside the different processes. There is context-switching between the threads. There can be numerous processes inside the operating system, and single process can have numerous threads.

Java language offer Thread class to attain thread programming. Thread class present constructors and methods to construct and carry out operations on a thread. Thread class expand Object class and execute Runnable interface.

The Java language run-time system rely on threads for several things, and all the class libraries are intended with multithreading in mind. In actual fact, Java language uses threads to enable the whole environment to be asynchronous. This helps in reducing incompetence by avoiding the waste of CPU cycles.

(c) Amity University Online

Notes

The significance of a multithreaded environment is best understood in disparity to its counterpart. Single-threaded scheme use an approach known as an event loop with polling. In this model, a single thread of control executes in an infinite loop, polling a single event queue to make a decision what to do next. Once this polling system returns with, declare, a signal that a network file is all set to be read, then the event loop transmits control to the suitable event handler.

Till this event handler returns, system gets into an idle state. This takes too much time of a CPU. It can also result in one part of a program taking control of the system and avoiding any other events from being executed or processed. In common, a singled-threaded environment, when a thread blocks (that is, suspends execution) because it is looking for some resources, the entire code stops executing.

The advantage of Java's language multithreading is that the main loop/polling system is eliminated. One thread can be break without impacting on the other parts of developer program. For example, the idle time created when a thread compile data from a network or look for user contribution can be used somewhere else. Multithreading permits animation loops to snooze for a second among each frame without affecting the whole system to pause. When a thread blocks in a Java language program, only the single thread that is blocked pauses. All other threads continue to execute.

Threads presents in several states. A thread can be running. It can be ready to execute as soon as it gets CPU time. A running thread can be stopped, which for the time being suspends its activity. A suspended thread can then be recommence, letting it to pick up where it left off. A thread can be barren when looking for a resource. At whatever time, a thread can be ended, which bring to an end to its execution immediately. Once terminated, a thread cannot be start again.

Messaging

After developers divide his/her code into separate threads, developer requires declaring how they will connect with each other. When programming with most other languages, developer must depend on the operating system to set up communication among threads. This adds overhead. By contrast, Java language provides a clean, low-cost method for two or more threads to communicate with each other, via calls to predefined methods that all objects have. Java's language messaging framework permits a thread to enter a synchronized method on an object, and then stay there until some other thread explicitly informs it to come out.

Thread Priorities

Java language allocates to every thread a priority that declares how that thread should be indulgence with respect to the others. Thread priorities are integers that state the comparative precedence of one thread to another. As a complete value, a priority is worthless; a higher-priority thread doesn't execute any faster as compared to a lower-priority thread if it is the only thread executing. In its place, a thread's priority is used to determine when to jumps from one running thread to the next. This is called a context switch. The rules that determine when a context switch takes place are simple:

(Amity University Online

Notes

- A thread can willingly hand over control. This is done by clearly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are inspected, and the highest-priority thread that is prepared to execute is given the CPU.
- A thread can be obstructing by a higher-priority thread. In this case, a lower-priority thread that does not give up the processor is simply holds—no matter what it is doing— by a higher-priority thread. On the whole, as soon as a higher-priority thread wants to executes, it does irrespective of any other process. This is called preemptive multitasking.

In cases where two threads with the similar priority are challenging for CPU cycles, the circumstances are a bit difficult. For operating systems such as Windows, threads of identical priority are time-sliced repeatedly in round-robin fashion. For other types of operating systems, threads of equal priority have to willingly yield managed to their peers. If they don't, the other threads will not execute.

Synchronization

Since multithreading brings in an asynchronous behavior to your code, there must be a way for developer to implement synchronicity when developer require it. For example, if you want two threads to converse and contribute to a complicated data structure, such as a linked list, developer need some way to make sure that they don't argue among each other. That is, developer must stop one thread from writing data while another thread is in the middle of reading it. For this purpose, Java language employs an elegant twist on an age-old model of inter-process synchronization: the monitor. The monitor is a manage mechanism first explain by C.A.R. Hoare. Developer can think of a screen as a very small box that can grasp only one thread. Once a thread enters a screen, all other threads have to wait until that thread exits the screen. In this fashion, a monitor can be utilized to shield a shared asset from being influenced by more than one thread at a time.

Most multithreaded systems expose screens as objects that your code must openly acquire and manipulate. Java language provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implied monitor that is automatically go into when one of the object's synchronized methods is invoked. Once a thread is within a synchronized method, no other thread can invoke any other synchronized method on the same object. This facilitates developer to write very clear and crisp multithreaded code, because synchronization support is built within the language.

The Main Thread

The Main Thread when a Java language program sets up, one thread starts executing right away. This is generally invoked the main thread of your code, because it is the one that is running when his/her program starts. The main thread is vital for two reasons:

1. It is the thread from which other "child" threads will be generated.
2. Often, it must be the last thread to come to an end execution because it carry outs a variety of shutdown actions.

Although the main thread is constructed without human intervention when your program is begins, it can be managed through a Thread object. To carry out this,

(c) Amity University Online

Notes

developer must get a reference to it by invoking the method currentThread( ), which is a public static member of Thread. Its general form can be represented as:

static Thread currentThread( )

This method returns a reference to the thread in which it is invoked. Once developer has a reference to the main thread, developer can control it just like any other thread.

Let's look into below code and examine it

```
class CurrentThreadDemo1 {
public static void main(String args[]) {
Thread td = Thread.CurrentThreadDemo();
System.out.println("Current thread: " + td);
// change the name of the thread
td.setName("My Thread");
System.out.println("After name change: " + td);
try {
for(int n = 7; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted");
}
}
}
```

In this program, a reference to the current thread (the main thread, in this case) is acquired by invoking CurrentThreadDemo( ), and this reference is stored in the local variable td. Next, the program shows information about the thread. The program then invoke setName( ) to modify the internal name of the thread. Information as regards the thread is then redisplayed. Next, a loop counts down from seven, pausing one second between each line.

The pause is achieved by the sleep( ) method. The argument to sleep( ) identify the holdup period in milliseconds. Observe the try/catch block around this loop. The sleep( ) method in Thread may throw an InterruptedException. This would happen if some other thread wanted to break off this sleeping one. This example just prints a message if it gets broken up. In a real code, developer would need to tackle this differently. Here is the output generated by this program:

(c) Amity University Online

Notes

Output

Current thread: Thread[main,7,main]

After name change: Thread[My Thread,7,main]

7

6

5

4

3

2

1

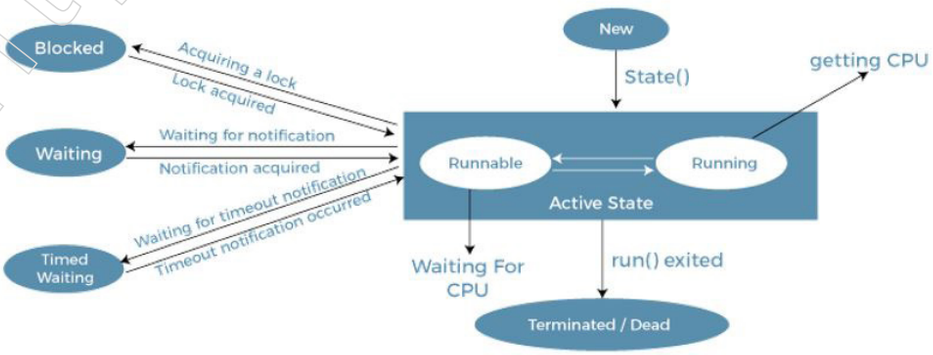
Notice the output produced when td is used as an argument to println( ). Then shows, in order: the name of the thread, its priority, and the given name of its group. By default, the name of the main thread is main.

Its priority is 7, which is the default value, and main is also the name of the group of threads to which this thread fit in. A thread group is a data structure that manages the pahse of a collection of threads as a complete. After the name of the thread is modified, td is again output. This time, the new name of the thread is showed.

4.3.3 Lifecycle of Thread

In Java language, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated



Life Cycle of a Thread

(c) Amity University Online

Notes

1. New Thread: When a new thread is constructed, it is in the new state. The thread has not yet in progress to execute when thread is in this state. When a thread stays in the new state, its code is yet to be run and hasn't on track to execute.

2. Runnable State: A thread which is ready to run is moved to runnable state. In this state, a thread might actually be executing or it might be ready run at any point of time. It is the accountability of the thread scheduler to provide the thread, time to run.

   A multi-threaded code assigns a fixed amount of time to each entity thread. Each and every thread runs for a little while and then pauses and abandon the CPU to another thread, so that other threads can acquire a chance to run. When this occurs, all such threads that are prepared to run, waiting for the CPU and the currently running thread located in runnable state.

3. Blocked or Waiting: Whenever a thread is still for duration of time (not permanently) then, either the thread is in the blocked phase or is in the waiting phase.

   For example, a thread (let's say its name is X) may want to print some information from the printer. However, at the similar time, the other thread (let's say its name is Y) is utilizing the printer to print some information. Consequently, thread X has to hang about for thread Y to utilize the printer. Thus, thread X is in the blocked Phase. X thread in the blocked state is not capable to carry out any execution and thus never use any cycle of the Central Processing Unit (CPU). Hence, we can say that thread X remains idle until the thread scheduler reinitiate thread X, which is in the waiting or blocked phase or state.

   When the main thread calls the join() method then, it is thought that the main thread is in the waiting state. The main thread then stops for the child threads to finish their duties. When the child threads finish their job, an announcement is transmits to the main thread, which once more moves the thread from waiting to the active state.

   If there are a group of threads in the waiting or blocked state, then it is the responsibility of the thread scheduler to decide which thread to select and which one to negelect, and the chosen thread is then given the chance to run.

4. Timed Waiting: Sometimes, waiting for takes code to starvation. For example, a thread (its name is X) has entered the significant section of a code and is not ready to leave that significant section. In such a scenario, another thread (its name is Y) has to hold everlastingly, which leads to starvation. To keep away from such circumstances, a timed waiting state is given to thread Y. Thus, thread located in the waiting state for a definite duration of time, and not forever. A real example of timed waiting is when we calls the sleep() method on a particular thread. The sleep() method locate the thread in the timed wait state. After the time complete, the thread wakes up and beginning its execution from where it has left earlier.

5. Terminated State: A thread ends because of either of the following reasons:

   - Because it exists as planned. This take place when the code of thread has completely executed by the program.
   - Because there take place some abnormal incorrect event, like segmentation error or an unhandled exception.

   A thread that located in a terminated state does no longer eat any cycles of CPU.

(c) Amity University Online

Notes

Implementing Thread States in Java

In Java language, to get the present state of the thread, use Thread.getState() method to get the present state of the thread. Java language provides java.lang.Thread. State class that declares the ENUM constants for the state of a thread, as synopsis of which is given below:

1. Constant type: New

public static final Thread.State NEW

Explanation: Thread state for a thread which has not yet started.

2. Constant type: Runnable

Expression: public static final Thread.State RUNNABLE

Explanation: Thread state for a runnable thread. A thread in the runnable state is running in the Java virtual machine but it might be looking for other resources from the operating system such as processor.

3. Constant type: Blocked

Expression: public static final Thread.State BLOCKED

Explanation: Thread state for a thread blocked looking for a monitor lock. A thread in the blocked state is looking for a monitor lock to enter a synchronized block/method or re-enter a synchronized block/method after invoking Object.wait().

4. Constant type: Waiting

Expression: public static final Thread.State WAITING

Explanation: Thread state for a waiting thread. Thread state for a waiting thread. A thread is in the waiting state due to invoking one of the following methods:

- Object.wait with no timeout
- Thread.join with no timeout
- LockSupport.park

A thread in the waiting state is looking for another thread to carry out a meticulous action.

5. Constant type: Timed Waiting

Declaration: public static final Thread.State TIMED_WAITING

Explanation: Thread state for a waiting thread with a particular waiting time. A thread is in the timed waiting state due to invoking one of the methods described below with a particular positive waiting time:

- Thread.sleep
- Object.wait with timeout

(c) Amity University Online

Notes

- Thread.join with timeout
- LockSupport.parkNanos
- LockSupport.parkUntil

6. Constant type: Terminated

Expression: public static final Thread.State TERMINATED

Explanation: Thread state for a terminated thread. The thread has finished execution.

Example: Java code to demonstrate different thread state

```
class thread implements Runnable
{
    public void run()
    {
            // moving thread02 to timed waiting state
            try
            {
                    Thread.sleep(1500);
            }
            catch (InterruptedException e)
            {
                    e.printStackTrace();
            }
            System.out.println("State of thread1 even as it invokedjoin() method on
thread02 -"+
                    Test.thread1.getState());
            try
            {
                    Thread.sleep(200);
            }
            catch (InterruptedException e)
            {
                    e.printStackTrace();
            }
    }
```

(c) Amity University Online

Notes

```
}
public class Test implements Runnable
{
    public static Thread thread01;
    public static Test ob;
    public static void main(String[] args)
    {
        ob = new Test();
        thread01 = new Thread(ob);
        // thread1 constructed and is currently in the NEW state.
        System.out.println("State of thread01 after constructing it - " + thread1.
getState());
        thread1.start();
        // thread1 moved to Runnable state
        System.out.println("State of thread1 after invoking .start() method on it -
" +
            thread1.getState());
    }
    public void run()
    {
        thread myThread = new thread();
        Thread thread02 = new Thread(myThread);
        // thread1 condtructed and is currently in the NEW state.
        System.out.println("State of thread02 after constructing it - "+ thread02.
getState());
        thread02.start();
        // thread02 moved to Runnable state
        System.out.println("State of thread02 after invoking .start() method on
it - " +
            thread02.getState());
        // moving thread1 to timed waiting state
        try
        {
```

(c) Amity University Online

Notes

```
                    //moving thread1 to timed waiting state
                    Thread.sleep(200);
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
                System.out.println("State of thread02 after invoking .sleep() method on
it - "+
                    thread02.getState() );
                try
                {
                    // waiting for thread02 to die
                    thread02.join();
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
                System.out.println("State of thread02 when it has completes its
execution - " +
                    thread02.getState());
        }
        }
```

Output

State of thread1 after constructing it - NEW

State of thread1 after invoke .start() method on it - RUNNABLE

State of thread02 after constructing it - NEW

State of thread02 after invoke .start() method on it - RUNNABLE

State of thread02 after invoke .sleep() method on it - TIMED_WAITING

State of thread1 while it invokedjoin() method on thread02 -WAITING

State of thread02 when it has completes its execution – TERMINATED

(c) Amity University Online

Notes

4.3.4 Implementing a Thread

Creating a Thread

In the most general sense, developer constructs a thread by defining an object of type Thread. Java defines two ways in which this can be completed:

- Developer can implement the Runnable interface.
- Developer can extend the Thread class, itself.

Common used method in thread class

Class	Declaration
public void run()	is used to perform action for a thread.
public void start()	starts the execution of the thread.JVM calls the run() method on the thread
public void sleep(long miliseconds)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
public void join()	waits for a thread to die.
public void jointlong miliseconds)	waits for a thread to die for the specified miliseconds.
public int getPriority()	returns the priority of the thread.
public int setPriority(int priority)	changes the priority of the thread.
public String getName()	returns the name of the thread.
public void setNamel String name)	changes the name of the thread
public Thread currentThread)	returns the reference of currently executing thread
public int getid)	returns the id of the thread.
public Thread StategetState()	returns the state of the thread.
public booleanisAlive)	tests if the thread is alive
public void yield()	causes the currently executing thread object to temporarily pause and allow other threads to execute.
public void suspendi)	is used to suspend the thread.
public void resume)	is used to resume the suspended thread.
public void stopi)	is used to stop the thread.
public booleanis Daemont)	tests if the thread is a daemon thread
public void setDaemon(boolean b)	marks the thread as daemon or user thread
public void interrupt()	interrupts the thread.
public boolesnisinterrupted)	tests if the thread has been interrupted
public static boolean interrupted()	tests if the current thread has been interrupted.

(c) Amity University Online

Notes

Implementing Runnable:

The simplest way to construct a thread is to create a class that employs the Runnable interface. Runnable abstracts a element of executable program. Developer can create a thread on any object that executes Runnable. To employ Runnable, a class recquire only implement a single method known run( ), which is expressed like this:

public void run( )

Inside run( ), developer will declare the program that include the new thread. It is significant to appreciate that run( ) can invoke other methods, use other classes, and define variables, just like the main thread is able to. The only disparity is that run( ) establishes the access point for another, simultaneous thread of execution inside his/her program. This thread will completes when run( ) returns.

After developerconstructs a class that implements Runnable, developer will instantiate an object of typeThread from inside that class. Thread declares several constructors. The one that we will use is shown below:

Thread(Runnable threadObj, String threadDemo)

In this constructor, threadObj is an example of a class that executes the Runnable interface.Thisdeclares where implementation of the thread will start. The name of the new thread is specifiedbythreadDemo.After the new thread is constructed, it will not start executing until you call its start( )method,which is definedinside Thread. In core, start( ) executes a call to run( ).

Example of creating a new thread

```
// Create a second thread.
class NewThreaddemo implements Runnable {
Thread td;
NewThreaddemo() {
td = new Threaddemo(this, “Demo Thread”);
System.out.println(“Child thread: “ + td);
td.start(); // Begin the thread
}
public void run() {
try {
for(int x = 5; x > 0; x--) {
System.out.println(“This is Child Thread: “ + x);
Thread.sleep(700);
}
} catch (InterruptedException e) {
```

(c) Amity University Online

Notes

```
System.out.println("the Child thread interrupted.");
}
System.out.println("Exiting the child thread.");
}
}
class ThreadDemo {
public static void main(String args[]) {
new NewThreaddemo(); // create a new thread
try {
for(int x = 5; x > 0; x--) {
System.out.println("First Thread: " + x);
Thread.sleep(1500);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Inside NewThreaddemo's constructor, a new Thread object is constructed by the followingstatement:

```
td = new Thread(this, "Demo Thread");
```

Passing this as the first argument shows that developer wish for the new thread to call the run( )method on this object. Next, start( ) is invoked, which starts the thread of execution starting at the run( ) method. This cause the child thread's for loop to start. After invokingstart( ),NewThreaddemo's constructor returns to main( ). When the main thread resumes, it go intoforloop. Both threads carry on executing, allocating the CPU, until their loops end.

Output

```
Child thread: Thread[Demo Thread,5,main]
First Thread: 5
This is Child Thread: 5
This is Child Thread: 4
First Thread: 4
```

(c) Amity University Online

Notes

This is Child Thread: 3

This is Child Thread: 2

First Thread: 3

This is Child Thread: 1

Exiting the child thread.

First Thread: 2

First Thread: 1

Main thread exiting.

As told earlier, in a multithreaded code, frequently the main thread has to be thelast thread to end running. In fact, for some older Java Virtual Machines, if the main thread end beforea child thread has finished, then the Java run-time system may "suspend." The previous example of program make sure that the main thread finishes at the end, because the main thread sleeps for 1,500milliseconds between iterations, but the child thread sleeps for only 700 milliseconds. This is the reason the child thread to end earlier than the main thread.

Extending Thread

The second method to create a thread is to construct a new class that extends Thread, and then to create a module of that class. The extending class have to override the run( ) method, which is the starting point for the new thread. It must also invoke start( ) to start running of the new thread.

Here is the same program rewritten to define extend Thread:

```
// Create a second thread by extending Thread
class NewThreadDemo extends Thread {
NewThreadDemo() {
// Create a new, second thread
super("Demo Thread");
System.out.println("This is Child thread: " + this);
start();
}
// This is the starting point for the second thread.
public void run() {
try {
for(int x= 5; x > 0; x--) {
System.out.println("This is Child Thread: " + x);
Thread.sleep(700);
```

(c) Amity University Online

Notes

```
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int x = 5; x > 0; x--) {
System.out.println("Main Thread: " + x);
Thread.sleep(1500);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

This program generates the same output as the preceding version. As you can see, the childthread is created by instantiating an object of NewThreadDemo, which is resultant from Thread.Notice the call to super( ) inside NewThreadDemo. This calls the following form of theThread constructor:

public Thread(String threadst)

Here, threadst state the name of the thread.

(c) Amity University Online

Notes

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run( ). This is, of course, the same method required when you implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable. This is up to you, of course.

4.3.5 Thread v/s Runnable method of creating thread

Basis For Comparison	Thread	Runnable
Basic	Each thread creates a unique object and gets associated with it.	Multiple threads share the same objects.
Memory	As each thread creates a unique object, more memory required.	As multiple threads share the same object less memory is used.
Extending	In Java, multiple inheritance not allowed hence, after a class extends Thread class, it cannot extend any other class.	If a class define thread implementing the Runnable interface it has a chance of extending one class.
Use	A user must extend thread class only if it wants to override the other methods in Thread class.	If you only want to specialize run method then implementing Runnable is a better option.
Coupling	Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread.	Implementing Runnable interface introduces loose coupling as the code of Thread is separate from the job of Threads.

Example of runnable:

```
class RunnableDemo implements Runnable{
  public void run(){
System.out.println("Thread is running for Runnable Implementation Demo");
  }
  public static void main(String args[]){
RunnableExample runnable=new RunnableExample();
    Thread td1 =new Thread(runnable);
    td1.start();
  }
}
```

(c) Amity University Online

Notes

Output

Thread is running for Runnable Implementation Demo

Example of Thread implementation:

```
class ThreadDemo extends Thread{
  public void run(){
System.out.println(“Thread Demo is running”);
  }
  public static void main(String args[]){
ThreadDemox=new ThreadDemo ();
x.start();
  }
}
```

Output

Thread demo is running.

Creating Multiple Threads

So far, we have been with only two threads: the main thread and one child thread. Though, developer program can initiate as much threads as it requires. For example, the following code constructs three child threads:

```
// Create multiple threads.
class NewThreadDemo implements Runnable {
String name; // name of thread
Thread td1;
NewThreadDemo(String threadname) {
name = threadname;
td1 = new Thread(this, name);
System.out.println(“New thread: “ + td1);
td1.start(); // Start the thread
}
public void run() {
try {
for(int a = 5; a > 0; a--) {
System.out.println(name + “: “ + a);
```

(c) Amity University Online

Notes

```
Thread.sleep(1200);
}
} catch (InterruptedException e) {
System.out.println(name + “Interrupted”);
}
System.out.println(name + “ exiting.”);
}
}
class MultiThreadDemo {
public static void main(String args[]) {
new NewThreadDemo(“One”); // start threads
new NewThreadDemo(“Two”);
new NewThreadDemo(“Three”);
try {
// wait for other threads to end
Thread.sleep(12000);
} catch (InterruptedException e) {
System.out.println(“Main thread Interrupted”);
}
System.out.println(“Main thread exiting.”);
}
}
```

Output

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
```

(c) Amity University Online

Notes

One: 3

Three: 3

Two: 3

One: 2

Three: 2

Two: 2

One: 1

Three: 1

Two: 1

One exiting.

Two exiting.

Three exiting.

Main thread exiting.

In above example, once started, all three child threads split the CPU. Notice the call to sleep(12000) in main( ). This is reasons the main thread to sleep for twelve seconds and ensures that it will end last.

4.3.6 Methods of Thread class

Thread a line of execution inside a code. Each code can have multiple connected threads. Each thread has a priority which is used by the thread scheduler to decide which thread must run at which place. Java language provides a thread class that has various method calls in order to administer the behavior of threads by giving constructors and methods to carry out operations on threads.

Method name	Method Code	Description
current Thread	public static Thread current Thread	the currently executing thread.
yield	public static void yield()	Yield is a heuristic attempt to improve relative progression between threads that would otherwise over-utilise a CPU, Its use should be combined with detailed profiling and benchmarking to ensure that it actually has the desired effect.
sleep	public static void sleep(long mills)	Causes the c e currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors Lowners!

(c) Amity University Online

Notes

sleep	public static void sleep(long mills, int nanos)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors
clone	protected Object clone()	Throws Clone NotSupported Exception as a Thread can not be meaningfully cloned. Construct a new Thread instead
start	public void start()	Causes this thread to begin execution the Java Virtual Machine calls the run method of this thread.
Run	public void run()	If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called, otherwise, this method does nothing and returns.
stop	public final void stop()	Forces the thread to stop executing
interrupted	public void interrupt()	Interrupts this thread
interrupted	public static boolean interrupted()	The interrupted status of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false funless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it. A thread interruption ignored because a thread was not alive at the time of the Interrupt will be reflected by this method returning false.
is interrupted	public boolean is interrupted()	Tests whether this thread has been interrupted. The interrupted status of the thread is unaffected by this method. A thread interruption ignored because a thread was not alive at the time of the interrupt will be reflected by this method returning false.
is interrupted	public boolean is Interrupted()	Tests if this thread is alive. A thread is alive fit has been started and has not yet died.
is Alive	public final boolean is Alive()	First, the checkAccess method of thes thread is called with no arguments. This may result in throwing a SecurityException (in the current thread if the thread is alive, it is suspended and makes no further progress unlest and until It is resumed.
suspend	public final void suspend()	First, the checkAccess method of this thread is called with no arguments. This may result in throwing a SecurityException in the current thread) if the thread is alive but suspended, it is resumed and is permitted to make progress in its execution.

(c) Amity University Online

Notes

resume	public final void resume()	First the checkAccess method of this thread is called with no arguments. This may result in throwing a SecurityException otherwise, the priority of this thread is set the smaller of the specified newfrienty and the main permitted pilarity of the thead's thread group.
set Priority	public final void set Priority (int new Priority)	Returns this thread's priority.
set Name	public final void set Name (String Name)	Changes the name of this thread to be equal to the argument name. First the checkAccess method of this thread is called with no arguments. This may result in throwing a SecurityException.
get Name	public final string get Name()	Returns this thread's name.
get Thread Group	public final Thread Group get Thread Group()	Returns the thread group to which this thread belongs. This method returns null if this thread has died (been stopped) Returns an estimate of the number of active threads in the current thread's thread group and its subgroups. Recursively iterates over all subgroups in the current thread's thread group.
active Count	public static int active count()	Copies into the specified array every active thread in the current thread's thread group and its subgroups. This method simply Invokes the ThreadGroup. enumerate(Thread[]) method of the current thread's thread group.
enumerate	public static int enumerate (thread[] tarray)	Counts the number of stack frames in this thread. The thread must be suspended.
count Stack Frames	public int count stack frames()	Waits at mest millis milliseconds for this thread to die. A timeout of 0 means to wait forever.
join	public final void join (long millis)	Marks this thread as either a daemon thread or a user thread. The Java Virtual Machine exits when the only threads running are all daemon threads.
set Daemon	public final void set Daemon (boolean on)	Tests if this thread is a daemon thread.
is Daemon	public final boolean is Daemon()	Determines if the currently running thread has permission t to modify this thread. if there is a security manager its checkAccess method is s called with this thread as its argument. This may result in throwing a SecurityException
Check Access	public final void check access	Returns a string representation of this thread, including the thread's name. priority, and thread group

(c) Amity University Online

Notes

to string	public string to string()	Returns a string representation of this thread, including the thread's name, priority, and thread group.
get Context Class Loader	public Class Loader get Context Class Loader()	Returns the context ClassLoader for this Thread. The context ClassLoader is provided by the creator of the thread for use by code running in this thread when loading classes and resources. If not set, the default is the of the parent Thread ClassLoader context text Classtoader of the primordial The thread is typically set to the class loader used to load the application.
set Context Class Loader	public void set Context Class Loader (Class Loader cl)	Sets the context ClassLoader for this Thread The context ClassLoader can be set when a thread is created, and allows the creator of the thread to provide the appropriate class loader through getContextClassLoader to code running in the thread when loading classes and resources
holds Lock	public static boolean holds Lock (Object obj)	Sets the context ClassLoader for this Thread. The context Classtoader can be set when a thread is created, and allows the creator of the thread to provide the appropriate class loader, through ContextClassLoader, to code running in the thread when loading classes and resources if security manager is present, its checkPermission method is invoked with a RuntimePermission("setContextClass Loader") permission to see if setting the context ClassLoader is permitted.
holds Lock	public static boolean holds Lock (Object obj)	Returns true if and only if the current thread holds the monitor lock on the specified object This method is designed to allow a program to assert that the current thread already holds a specified lock
get Stack Trace	public Stack Trace Element[] get Stack Trace()	Returns an array of stack trace elements representing the stack dump of this thread This method will return a zero-length array if this thread has not started, has started but has not yet been scheduled to run by the system, or has terminated. If the returned array is of non-zero length then the first element of the array represents the top of the stack, which is the most recent method invocation in the sequence. The last element of the array represents the bottom of the stack, which is the least recent method invocation in the sequence.

Using isAlive( ) and join( )

Often developer wishes for the main thread to finish at the end. In the previous examples, this is completed by invoking sleep( ) inside main( ), with a sufficient delay to

Notes

make sure that all child threads finish prior to the main thread. On the other hand, this is barely a reasonable solution, however there class provide us a solution for that

There are two ways to define whether a thread has ended or not.

First, developer can call isAlive( ) on the thread. This method is declared by Thread, and it can be represented as:

final booleanisAlive( )

The isAlive( ) method returns true if the thread upon which it is invoke is still running. It returns false else.

While isAlive( ) is infrequently useful, the method that developer will more commonly use to wait for a thread to finish is known join( ), its syntax is given below

final void join( ) throws InterruptedException

This method stay until the thread on which it is invokes ends. Its name comes from the idea of the invoking thread waiting until the particular thread joins it. Supplementary forms of join( ) lets developer to indicate a maximum amount of time that developer want to wait for the particular thread to end.

Example to shoe isAlive() and join() method:

```
class NewThreadDemo implements Runnable {
String name; // name of thread
Thread a;
NewThreadDemo(String threadname) {
name = threadname;
a = new Thread(this, name);
System.out.println(“New thread: “ + a);
a.start(); // Start the thread
}
public void run() {
try {
for(int x = 3; x > 0; x--) {
System.out.println(name + “: “ + x);
Thread.sleep(1200);
}
} catch (InterruptedException e) {
System.out.println(name + “ interrupted.”);
}
```

(c) Amity University Online

Notes

```
System.out.println(name + " exiting.");
}
}
class DemoJoin {
public static void main(String args[]) {
NewThreadDemo ob1 = new NewThreadDemo("One");
NewThreadDemo ob2 = new NewThreadDemo("Two");
NewThreadDemo ob3 = new NewThreadDemo("Three");
System.out.println("Thread One is alive: "
+ ob1.a.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.a.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.a.isAlive());
// wait for threads to complete
try {
System.out.println("Waiting for threads to complete.");
ob1.a.join();
ob2.a.join();
ob3.a.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "
+ ob1.a.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.a.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.a.isAlive());
System.out.println("Main thread exiting.");
}
}
```

(c) Amity University Online

Notes

Output

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

New thread: Thread[Three,5,main]

Thread One is alive: true

Thread Two is alive: true

Thread Three is alive: true

Waiting for threads to complete.

One: 3

Two: 3

Three: 3

One: 2

Two: 2

Three: 2

One: 1

Two: 1

Three: 1

Two exiting.

Three exiting.

One exiting.

Thread One is alive: false

Thread Two is alive: false

Thread Three is alive: false

Main thread exiting.

4.3.7 Thread Priorities

Thread priority in Java language is a number allocated to a thread i.e. utilized by Thread scheduler to identify which thread should be permissible to run.

In Java language, each thread is dispersing a different priority that will choose the preference in which it is scheduled for executing.

Thread priorities are symbolized by a number from 1 to 10 that specify the priority of one thread to another. The thread with the uppermost priority is chosen by the scheduler to be executed primary.

(c) Amity University Online

Notes

In theory, higher-priority threads acquire more CPU instance than lower-priority threads. In practice, the sum of CPU time that a thread find often based on several factors in addition to its priority. (For example, how an operating system realizes multitasking can influence the relative availability of CPU time.) A higher-priority thread is able to anticipate a lower-priority one. For instance, when a lower-priority thread is executing and a higher-priority thread starts again (from sleeping or waiting on I/O, for example), it will anticipate the lower priority thread.

The default priority of a thread is 5. Thread class in Java language also provides quite a few priority constants to describe the priority of a thread. These are:

1. MIN_PRIORITY = 1
2. NORM_PRIORITY = 5
3. MAX_PRIORTY = 10

These 3 constant can be represented as:

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

These constants are public, final, and static associate of the Thread class.

In all the previous examples, threads we have discussed are of the same priority (equal priority). The threads of the similar priority are given equal time by Java language scheduler.

Thread scheduler choose the thread for execution on the first-come, first-serve foundation. That is, the threads having identical priorities split the processor time on the first-come, first-serve basis.

When multiple threads are prepared for execution, the uppermost priority thread is preferred and executed by Java virtual machine. In case when high priority threads discontinue, give ways, or enters the blocked state, then a low priority thread starts running.

If any high priority thread goes into the runnable state, it will block the currently running thread forcing it to shift to the runnable state. Note that the highest priority thread forever prevents any lower priority thread to execute.

How to get Priority of Current Thread in Java?

Thread class offers a method named getPriority() that is used by developer to decide the priority of a thread. It returns the priority of a thread through which it is invoked.

Program to understand getpriority()

```
public class ABC implements Runnable
{
public void run()
```


(c) Amity University Online

Notes

```
{
System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
{
 ABC a = new ABC();
 Thread x = new Thread(a, “NewThreadDemo”);

System.out.println(“Priority of Thread: “ +x.getPriority());
System.out.println(“Name of Thread: “ +x.getName());
x.start();
    }
}
```

Output:

```
Priority of Thread: 5
Name of Thread: NewThreadDemo
Thread[NewThread,5,main]
```

In the above code, the getPriority() method is invoked using reference variable x of Thread class, therefore it returns the precedence of current thread through Thread.

How to set Priority of Thread in Java:

The setPriority() of Thread class is used by developer to position the priority of a thread. This method recognizes an integer value as an argument and locates that value as priority of a thread through which it is invoked. SetPriority() can be represented as:

```
ThreadName.setPriority(a);
```

Where, a is an integer value which variety from 1 to 10.

Program to understand setpriority()

```
public class ABC implements Runnable
{
public void run()
{
System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
```

(c) Amity University Online

```
{
 ABC a = new ABC();
 Thread x = new Thread(a, “NewThreadDemo”);
x.setPriority(3); // Setting the priority of thread.
System.out.println(“Priority of Thread: “ +x.getPriority());
System.out.println(“Name of Thread: “ +x.getName());
x.start();
  }
}
```

Output:

Priority of Thread: 3

Name of Thread: NewThreadDemo

Thread[NewThread,3,main]

In above code, the setPriority() method sets the priority of Thread x to 3.

Implementations of Java may have radically different behavior when it comes to scheduling. The Windows XP/98/NT/2000/2007 versions work, more or less, as you would expect. However, other versions may work quite differently. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

The below program exhibit two threads at unusual priorities, which do not execute on a preemptive platform in the same way as they execute on a non preemptive platform.

```
class XYZ implements Runnable {
long click = 0;
Thread x;
private volatile boolean running = true;
public XYZ(int a) {
x = new Thread(this);
x.setPriority(a);
}
public void run() {
while (running) {
click++;
```

(c) Amity University Online

Notes

```
}
}
public void stop() {
running = false;
}
public void start() {
x.start();
}
}
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
XYZ hi = new XYZ(Thread.NORM_PRIORITY + 2);
XYZ lo = new XYZ(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
hi.x.join();
lo.x.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
```

(c) Amity University Online

Notes

```
}
}
```

The output of this program, shown below when execute under Windows, indicates thatthe threads did background switch, even though neither willingly yielded the CPU nor blockedfor I/O. The higher-priority thread got the greater part of the CPU time.

Low-priority thread: 4408112

High-priority thread: 589626904

Check your Understanding

1. Developers are certainly familiar with multitasking, because it is supported by nearly all modern operating systems. Nevertheless, there are two separate types of multitasking _____ and _____.
2. In Java language, a single task is called a _____ .
3. _____ scheme use an approach known as an event loop with polling.
4. A thread's priority is used to determine when to jumps from one running thread to the next is called a _____ switch.
5. Most multithreaded systems expose screens as objects that your code must openly _____ and _____ .
6. A thread that located in a _____ state does no longer eat any cycles of CPU.
7. In Java language, to get the present state of the thread, use _____ method to get the present state of the thread.
8. The second method to create a thread is to _____ a new class that extends Thread.
9. Each thread is dispersing a different priority that will choose the preference in which it is _____ for executing.

Summary

- Process-based multitasking is the characteristic that lets your computer to run two or more programs alongside.
- In a GUI code, there is at least one supplementary thread, which is in charge for managing events and drawing components on the display. This GUI thread is shaped when the first window is opened. So in fact, developer has already done parallel programming
- Java language offer Thread class to attain thread programming. Thread class present constructors and methods to construct and carry out operations on a thread. Thread class expand Object class and execute Runnable interface.
- The advantage of Java's language multithreading is that the main loop/polling system is eliminated. One thread can be break without impacting on the other parts of developer program. For example, the idle time created when a thread compile data from a network or look for user contribution can be used somewhere else.

(c) Amity University Online

Notes

- Java language allocates to every thread a priority that declares how that thread should be indulgence with respect to the others. Thread priorities are integers that state the comparative precedence of one thread to another.
- A thread can willingly hand over control. This is done by clearly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are inspected, and the highest-priority thread that is prepared to execute is given the CPU.
- A thread can be obstructing by a higher-priority thread. In this case, a lower-priority thread that does not give up the processor is simply holds—no matter what it is doing— by a higher-priority thread. On the whole, as soon as a higher-priority thread wants to executes, it does irrespective of any other process. This is called preemptive multitasking.
- A thread always exists in any one of the following states. These states are:
  1. New
  2. Active
  3. Blocked / Waiting
  4. Timed Waiting
  5. Terminated
- The simplest way to construct a thread is to create a class that employs the Runnable interface. Runnable abstracts a element of executable program. Developer can create a thread on any object that executes Runnable. To employ Runnable, a class recquire only implement a single method known run( ).
- Thread priority in Java language is a number allocated to a thread i.e. utilized by Thread scheduler to identify which thread should be permissible to run.
- When multiple threads are prepared for execution, the uppermost priority thread is preferred and executed by Java virtual machine. In case when high priority threads discontinue, give ways, or enters the blocked state, then a low priority thread starts running.

Activity

1. Create a program and include multiple thread and assign different priorities to them?

Questions and exercises

1. Define Multithreading:
2. What do you understand from the word Thread in Multithreading?
3. Define a lifecycle of Thread?
4. How to implement a Thread?
5. What are the differences between Thread and Runnable method of creating thread?
6. Define various methods of Thread class
7. What are thread priorities?

(c) Amity University Online

Notes

Glossary

- Multithreading: Multithreading is a Java characteristic that lets simultaneous execution of two or more parts of a program for maximum consumption of CPU.
- Thread: The term “thread” refers to a “thread of control” or “thread of execution,” meaning a series of commands that are executed one after another— the thread extends during time, connecting each command to the next.
- GUI: Graphical User Interface
- CPU: Central processing Unit
- New Thread: When a new thread is constructed, it is in the new state.
- Runnable State: A thread which is ready to run is moved to runnable state. In this state, a thread might actually be executing or it might be ready run at any point of time.
- Blocked or Waiting: Whenever a thread is still for duration of time (not permanently) then, either the thread is in the blocked phase or is in the waiting phase.
- Timed Waiting: Sometimes, waiting for takes code to starvation.
- Terminated State: A thread ends because of either of the following reasons:
  - Because it exists as planned. This take place when the code of thread has completely executed by the program.
  - Because there take place some abnormal incorrect event, like segmentation error or an unhandled exception.

Further Readings:

1. The Java Native Interface: Programmer’s Guide and Specification, Sheng Liang
2. Essential JNI: Java Native Interface, Rob Gordon
3. The Java Application Programming Interface, Frank Yellin and James Gosling

Check Your Understanding:

Answers

1. process-based thread-based
2. thread
3. Single-threaded
4. Context
5. acquire and manipulate
6. terminated
7. Thread.getState()
8. Construct
9. scheduled

(c) Amity University Online

Notes

Unit - 4.4 : Thread Synchronization

Objectives

At the end of this unit, you will be able to understand:

- Understand the concept of Synchronization
- How to Implement Synchronization in Thread
- Define the Benefits and Loopholes of Synchronization
- Understanding the concept of Resuming and stopping Threads
- A code snippet to demonstrate thread with and without synchronization

Introduction

It is already mentioned in above segments that threads in Java language are executing in the same memory space, and therefore it is easy to converse between two threads. Inter-thread connections allow threads to talk to or wait on each other. Again, because all the threads in a code share the similar memory space, it is promising for two threads to use the same variables and methods in object.

Problems may happen when two or more threads are accessing the same data alongside, for example, one thread stores data into the shared object and the other threads reads data, and there can be synchronization trouble if the first thread has not completed storing the data earlier than the second one goes to read it. So developer needs to make sure to utilize the data by only one thread process at a same time. Java language offer unique language level support for such synchronization. In this segment we will learn how synchronization device and inter-thread communications are possible in Java language.

4.4.1 Introduction to Synchronization

When two or more threads require admission to a shared resource, they require some way to make sure that the resource will be used by only one thread at a time. The procedure by which this is achieved is known synchronization. As you will see, Java language gives unique, language-level support for it.

Explanation to synchronization is the perception of the monitor (also called a semaphore). A monitor is an object that is utilized as a reciprocally exclusive lock, or mutex. Only one thread can possess a monitor at a particular time. When a thread obtain a lock, it is said to encompass entered the monitor. All other threads attempting to go into the locked monitor will be hovering until the first thread exits the monitor. These other threads are supposed to be waiting for the monitor. A thread that possess a monitor can reenter the same monitor if it so needs.

Most multithreaded method expose monitors as objects that developer program must clearly obtain and manipulate. Java language gives a cleaner solution. There is no class “Monitor”; in its place, Each object has its own understood monitor that is automatically entered when one of the object’s. Synchronized methods is invoked. Once a thread is inside a synchronized method, no other Thread can invoke any other

(c) Amity University Online

synchronized system on the same object. This facilitates developer to write very clear and brief multithreaded code.

Notes

Let's understand the concept of synchronization with a help of program.

Suppose we want to sustain a bank account of customers. Several transactions, such as deposits withdrawal some amount from an account etc. are possible. Now, for a given account, if two or more transactions come concurrently then only one transaction should be allowed at a time instead of simultaneous transaction giving out so that data inconsistency will never takes place. So, what developer needs to do is to synchronize the transaction.

```
class Accountdemo {
    public int balance;
    public int accountNo;
    void displayBalance() {
            System.out.println("Account No:" + accountNo + "Balance: " + balance);
    }
      synchronized void deposit(int amount){
                    balance = balance + amount;
                    System.out.println( amount + " is deposited");
                    displayBalance();
      }

      synchronized void withdraw(int amount){
                     balance = balance - amount;
                    System.out.println( amount + " is withdrawn");
                    displayBalance();
      }
}
class TransactionDeposit implements Runnable{
    int amount;
    AccountdemoaccountX;
    TransactionDeposit(Accountdemo x, int amount){
            accountX = x;
            this.amount = amount;
            new Thread(this).start();
```


(c) Amity University Online

Notes

```
    }
        public void run(){
        accountX.deposit(amount);
    }
}

class TransactionWithdraw implements Runnable{
    int amount;
    AccountdemoaccountY;
    TransactionWithdraw(Accountdemo y, int amount) {
        accountY = y;
        this.amount = amount;
        new Thread(this).start();
    }
    public void run(){
        accountY.withdraw(amount);
    }
}
class Demonstration_119{
    public static void main(String args[]) {
        Accountdemo ABC = new Accountdemo();
        ABC.balance = 1000;
        ABC.accountNo = 0218;
        TransactionDepositt1;
        TransactionWithdrawt2;
        t1 = new TransactionDeposit(ABC, 600);
        t2 = new TransactionWithdraw(ABC,1000);
    }
}
```

Output:

600 is deposited

Account No:0218Balance: 1600

(c) Amity University Online

Notes

1000 is withdrawn

Account No:0218Balance: 600

In the above code, the keyword synchronized is utilizedfor the methods void deposits() and void withdraw so that these two methods will never execute for the same object instance concurrently.

4.4.2 Implementing Synchronization in Thread

Using Synchronized Methods

Synchronization is easy in Java language, because all objects have their own implied monitor linked with them. To enter an object's monitor, just invoke a method that has been customized with the synchronized keyword. While a thread is within a synchronized method, all other threads that try to invoke it (or any other synchronized method) on the similar instance have to wait. To exit the monitor and hand over control of the object to the next coming up thread, the holder of the monitor just returns from the synchronized method.

To understand the requirement for synchronization, let's begin with a simple example that does not use it—but should have to use. The below program has three simple classes. The first one, Callmedemo, has a single method named call( ). The call( ) method obtain a String parameter called m. This method attempt to print the m string inside of square brackets. The interesting thing to notice is that after call( ) prints the opening bracket and the m string, it calls Thread .sleep(1200), which pauses the current thread for 1.2 second.

The constructor of the next class, Callerdemo, takes a reference to an instance of the Callmedemo class and a String, which are stored in target and m, respectively. The constructor also constructs a new thread that will call this object's run( ) method. The thread is started instantly. The run( ) method of Callerdemo invokes the call( ) method on the target instance of Callmedemo, passing in the m string. Finally, the Synch class starts by constructing a single instance of Callmedemo, and three instances of Callerdemo, each with a exclusive message string. The same instance of Callmedemo is passed to each Callerdemo.

```
// This code is not synchronized.
class Callmedemo {
void call(String m) {
System.out.print("[" + m);
try {
Thread.sleep(1200);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
```

Notes

```
}
}
class Callerdemo implements Runnable {
String m;
Callmedemotarget;
Thread t;
public Callerdemo(Callmedemotarg, String s) {
target = targ;
m = s;
t = new Thread(this);
t.start();
}
public void run() {
target.call(m);
}
}
class Synch {
public static void main(String args[]) {
Callmedemo target = new Callmedemo();
Callerdemo ob1 = new Callerdemo(target, “Hello”);
Callerdemo ob2 = new Callerdemo(target, “Synchronized”);
Callerdemo ob3 = new Callerdemo(target, “World”);
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println(“Interrupted”);
}
}
}
```

(c) Amity University Online

Notes

Output

Hello[Synchronized[World]

]

]

As we can see in above code, by calling sleep( ), the call( ) method lets execution to switch to another thread. This consequences in the mixed-up output of the three message strings. In this code, nothing exists to end all three threads from invoking the same method, on the same object, at the similar time.

This is called as a race condition, because the three threads are racing among other to complete the method. This example used sleep( ) to compose the effects repeatable and obvious. In most situations, a race situation is more subtle and less expected, because developer can't be sure when the context switch will occur. This can be a reason for program to run right one time and wrong the next.

To correct the previous code, developer must serialize access to call( ). i.e., developer must restrict its right of entry to only one thread at a time. To do this, developer simply require to precede call( )'s statement with the keyword synchronized, as shown below:

class Callmedemo {

synchronized void call(String m) {

This stopss other threads from entering call( ) while another thread is utilizing it. After synchronized has been added to call( ), the output of the program will look like this:

[Hello]

[Synchronized]

[World]

Any time that developers have a method, or group of method, that influence the internal situation of an object in a multithreaded situation, developer should use the synchronized keyword to safeguard the state from race conditions. Remember, once a thread go into any synchronized method on an instance, no other thread can enter any other synchronized method on the same time.

Using Synchronized block

If developer want to synchronize right of entry to an object of a class or only a part of a method to be synchronized then developer can use synchronized block for it. It is able to make any part of the object and method synchronized.

Using synchronized block for single thread at a time.

```
class First
{
  public void display(String k)
```

(c) Amity University Online

Notes

```
 {
System.out.print ("["+k);
   try
   {
Thread.sleep(1000);
   }
catch(InterruptedException e)
   {
e.printStackTrace();
   }
System.out.println ("]");
 }
}
class Second extends Thread
{
  String k;
  First fobj;
  Second (First fp,String str)
  {
fobj = fp;
k = str;
start();
  }
  public void run()
  {
    synchronized(fobj)      //Synchronized block
    {
fobj.display(k);
    }
  }
}
public class MyThread
```

(c) Amity University Online

Notes

```
{
  public static void main (String[] args)
  {
    First fnew = new First();
    Second ss = new Second(fnew, “welcome”);
    Second ss1= new Second (fnew,”To”);
    Second ss2 = new Second(fnew, “Java”);
  }
}
```

Output

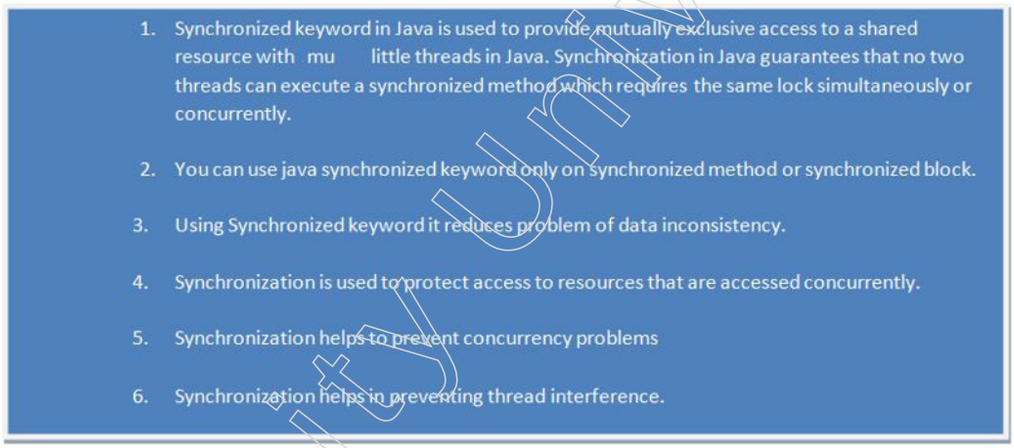
[welcome]

[To]

[Java]

4.4.3 Benefits and Loopholes of Synchronization

Benefits of synchronization



Loopholes of Synchronization

(c) Amity University Online

Notes

1. One Major disadvantage of Java synchronized keyword is that it doesn't allow concurrent read, which can potentially limit scalability. By using the concept of lock stripping and using different locks for reading and writing, you can overcome this limitation of synchronized in Java. You will be glad to know that java.util.concurrent.locks.ReentrantReadWriteLock provides ready-made implementation of ReadWriteLock in Java.

2. Limitation of java synchronized keyword is that it can only be used to control access to a shared object within the same JVM. If you have more than one JVM and need to synchronize access to a shared file system or database, the Java synchronized keyword is not at all sufficient. You need to implement a kind of global lock for that.

3. Java synchronized keyword incurs a performance cost. A synchronized method in Java is very slow and can degrade performance. So use synchronization in java when it absolutely requires and consider using java synchronized block for synchronizing critical section only.

4. Java synchronized code could result in deadlock or starvation while accessing by multiple threads if synchronization is not implemented correctly.

5. It increases waiting time of thread. At a time only one thread can operate on object so other threads have to wait. So, it creates performance problem.

4.4.4 Resuming and stopping Threads

Prior to Java 2, a code utilized suspend( ) and resume( ), which are methods declared by Thread, to pause and restart the execution of a thread. They can be represent as:

final void suspend( )

final void resume( )

Examples to show use of suspends and resume methods:

```
class NewThreadDemo implements Runnable {
String name; // name of thread
Thread t;
NewThreadDemo(String threadname) {
name = threadname;
x = new Thread(this, name);
System.out.println(“New thread: “ + x);
x.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 15; i> 0; i--) {
System.out.println(name + “: “ + i);
```

(c) Amity University Online

Notes

```
Thread.sleep(200);
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
}
class SuspendResume {
public static void main(String args[]) {
NewThreadDemo ob1 = new NewThreadDemo("One");
NewThreadDemo ob2 = new NewThreadDemo("Two");
try {
Thread.sleep(1000);
ob1.x.suspend();
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.x.resume();
System.out.println("Resuming thread One");
ob2.x.suspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.x.resume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.x.join();
ob2.x.join();
```

(c) Amity University Online

Notes

```
} catch (InterruptedException e) {
System.out.println(“Main thread Interrupted”);
}
System.out.println(“Main thread exiting.”);
}
}
```

Output

```
New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
```

(c) Amity University Online

Notes

Resuming thread Two

Waiting for threads to finish.

Two: 5

One: 5

Two: 4

One: 4

Two: 3

One: 3

Two: 2

One: 2

Two: 1

One: 1

Two exiting.

One exiting.

Main thread exiting.

As the suspend( ), resume( ), and stop( ) methods declared by Thread seem to be a completely sensible and suitable approach to managing the execution of threads, they must not be utilized for new Java language programs. Here's why.

The suspend( ) method of the Thread class was criticize by Java 2 several years ago. This was done because suspend( ) can from time to time root of serious system failures. The resume( ) method is also criticize. It does not create problems, but cannot be used exclusive of the suspend( ) method as its complement.

Because developer can't now use the suspend( ), resume( ), or stop( ) methods to manage a thread, developer might be thinking that no way exists to pause, restart, or terminate a thread. But, luckily, this is not true. In its place, a thread must be considered so that the run( ) method regularly checks to establish whether that thread should suspend, resume, or stop its own execution. Typically, this is achieved by creating a flag variable that signifies the execution condition of the thread.

Providing this flag is set to "running," the run( ) method must carry on to allow the thread execute. If this variable is set to "suspend," then the thread must break. If it is set to "stop," then the thread must end.

The below example exemplify how the wait( ) and notify( ) methods that are incorporated from Object can be used to manage the execution of a thread. This example is identical to the program in the previous section. Though, the criticized method calls have been removed.

The NewThreadDemo class includes a boolean instance variable named suspendFlag, which is utilized to manage the execution of the thread. It is set to false by the constructor. The run( ) method includes a synchronized statement block that

(c) Amity University Online

Notes

looks for suspendFlag. If that variable is true, the wait( ) method is called to suspend the execution of the thread. The mysuspend( ) method puts suspendFlag to true. The myresume( ) method sets suspendFlag to false and calls notify( ) to wake up the thread. Finally, the main( ) method has been changed to call the mysuspend( ) and myresume( ) methods.

```
class NewThreadDemo implements Runnable {
String name; // name of thread
Thread t;
booleansuspendFlag;
NewThreadDemo(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println(“New thread: “ + t);
suspendFlag = false;
t.start(); // Start the thread
}
public void run() {
try {
for(int i = 15; i> 0; i--) {
System.out.println(name + “: “ + i);
Thread.sleep(200);
synchronized(this) {
while(suspendFlag) {
wait();
}
}
}
} catch (InterruptedException e) {
System.out.println(name + “ interrupted.”);
}
System.out.println(name + “ exiting.”);
}
void mysuspend() {
```

(c) Amity University Online

Notes

```
suspendFlag = true;
}
synchronized void myresume() {
suspendFlag = false;
notify();
}
}
class SuspendResume {
public static void main(String args[]) {
NewThreadDemo ob1 = new NewThreadDemo("One");
NewThreadDemo ob2 = new NewThreadDemo("Two");
try {
Thread.sleep(1000);
ob1.mysuspend();
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
} catch (InterruptedException e) {
```

(c) Amity University Online

Notes

```
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Output:

The output of this program is identical to the previous code output.

4.4.5 A code snippet to demonstrate thread with and without synchronization

Multithreading Example without Synchronization:

```
class PrintDemo {
  public void printCount() {
    try {
for(int i = 5; i> 0; i--) {
System.out.println("Counter   ---   "  +i );
      }
    } catch (Exception e) {
System.out.println("Thread  interrupted.");
    }
  }
}
class ThreadDemo extends Thread {
  private Thread x;
  private String threadName;
PrintDemo  PD;

ThreadDemo( String name,  PrintDemo pd) {
threadName = name;
    PD = pd;
  }
  public void run() {
PD.printCount();
```

(c) Amity University Online

```
System.out.println("Thread " +  threadName + " exiting.");
  }
  public void start () {
System.out.println("Starting " +  threadName );
    if (x == null) {
x = new Thread (this, threadName);
x.start ();
    }
  }
}
public class TestThread {
  public static void main(String args[]) {

PrintDemo PD = new PrintDemo();
ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
    T1.start();
    T2.start();

    // wait for threads to end
      try {
      T1.join();
      T2.join();
    } catch ( Exception e) {
System.out.println("Interrupted");
    }
  }
}
```

Output

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
```

Notes

(c) Amity University Online

Notes

```
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Multithreading Example with Synchronization

```
class PrintDemo {
  public void printCount() {
    try {
for(int i = 5; i> 0; i--) {
System.out.println("Counter   ---   "  +i );
      }
    } catch (Exception e) {
System.out.println("Thread  interrupted.");
    }
  }
}
class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;
PrintDemo  PD;
ThreadDemo( String name,  PrintDemo pd) {
threadName = name;
    PD = pd;
  }
  public void run() {
```

(c) Amity University Online

Notes

```
synchronized(PD) {
PD.printCount();
    }
System.out.println("Thread " +  threadName + " exiting.");
  }
  public void start () {
System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
t.start ();
    }
  }
}
public class TestThread {
  public static void main(String args[]) {
PrintDemo PD = new PrintDemo();
ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
    T1.start();
    T2.start();
    // wait for threads to end
    try {
      T1.join();
      T2.join();
    } catch ( Exception e) {
System.out.println("interrupted");
    }
  }
}
```

Output

```
Starting Thread - 1
Starting Thread - 2
```

(c) Amity University Online

Notes

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread - 1 exiting.

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread - 2 exiting.

Check your Understanding

1. Once a thread is inside a synchronized method, no other Thread can invoke any other _____ system on the same object.
2. Thread .sleep(1200), which pauses the current thread by _____ second.
3. Prior to Java 2, a code utilized suspend( ) and resume( ), which are methods declared by Thread, to _____ and _____ the execution of a thread.
4. If developer wants to synchronize right of entry to an object of a class or only a part of a method to be synchronized then developer can use _____ for it.

Summary

- Synchronized methods is invoked. Once a thread is inside a synchronized method, no other Thread can invoke any other synchronized system on the same object. This facilitates developer to write very clear and brief multithreaded code.
- Synchronization is easy in Java language, because all objects have their own implied monitor linked with them. To enter an object's monitor, just invoke a method that has been customized with the synchronized keyword.
- Multithreaded method expose monitors as objects that developer program must clearly obtain and manipulate. Java language gives a cleaner solution. There is no class "Monitor"; in its place, Each object has its own understood monitor that is automatically entered when one of the object's.
- The synchronised method is used to secure an object's access to a shared resource. When a thread calls a synchronised method, the lock for that object is automatically acquired and released when the thread completes its task.
- The synchronized keyword in Java provides locking, which ensures mutually exclusive access to the shared resource and prevents data race.

(c) Amity University Online

Notes

- Synchronized keyword also prevents reordering of code statement by the compiler which can cause a subtle concurrent issue if we don't use synchronized or volatile keywords.
- Synchronized keyword involves locking and unlocking. before entering into a synchronized method or block thread needs to acquire the lock, at this point it reads data from main memory than cache and when it releases the lock, it flushes write operation into main memory which eliminates memory inconsistency errors.

Activity

1. What will be the output of below program?

```
class Line
{
        synchronized public void getLine()
    {
        for (int i = 0; i < 3; i++)
        {
            System.out.println(i);
            try
            {
                Thread.sleep(400);
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
class Train extends Thread
{
        Line line;
    Train(Line line)
    {
        this.line = line;
    }
```

(c) Amity University Online

Notes

```
    @Override
    public void run()
    {
        line.getLine();
    }
}
class GFG
{
    public static void main(String[] args)
    {
        Line obj = new Line();
        // we are creating two threads which share
        // same Object.
        Train train1 = new Train(obj);
        Train train2 = new Train(obj);
        // both threads start executing .
        train1.start();
        train2.start();
    }
}
```

Questions and exercises

1. What do you understand from the term Synchronization
2. How to implement Synchronization in Thread?
3. What are benefits and Loopholes of Synchronization?
4. What do you mean by resuming and stopping Threads
5. Create a program to demonstrate thread with and without synchronization?

Glossary

- Synchronization: When two or more threads require admission to a shared resource, they require some way to make sure that the resource will be used by only one thread at a time. The procedure by which this is achieved is known synchronization.
- Suspend Method: The suspend() method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a

(c) Amity University Online

Notes

thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

- Race condition: If a multiple threads want to operate on a same object simultaneously. There may be problem of data inconsistency. This condition is called Race condition.

Further Readings:

1. Java threads, Scott Oaks
2. Mastering Concurrency Programming with Java 8, Javier Fernández Gonzalez
3. Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors, Venkat Subramaniam
4. The Java Native Interface: Programmer's Guide and Specification, Sheng Liang
5. Essential JNI: Java Native Interface, Rob Gordon
6. The Java Application Programming Interface, Frank Yellin and James Gosling

Check Your Understanding:

Answers

1. synchronized
2. 1.2
3. pause restart
4. synchronized block

(c) Amity University Online

Notes

Module - 5 : Applet and Graphics Programming

Structure:

5.1 Introduction to Applet Class

- 5.1.1 Introduction to applets
- 5.1.2 Lifecycle of Applet
- 5.1.3 Methods of Applet class
- 5.1.4 Applet vs Application
- 5.1.5 Embedding Applet in HTML Page
- 5.1.6 A code snippet to implement Applet

5.2 Introduction to Graphic class

- 5.2.1 Introduction to Graphic class
- 5.2.2 Methods in Graphic class
- 5.2.3 A code snippet to create different shapes.
- 5.2.4 Using loops in applet

5.3 Introduction to AWT

- 5.3.1 Introduction to AWT package
- 5.3.2 Classes in AWT package
- 5.3.3 Role of Layout manager in applet
- 5.3.4 A code snippet to implement applet with layout manager

(c) Amity University Online

Unit - 5.1 : Introductions to Applet Class

Notes

Objectives

At the end of this unit, you will be able to understand:

- What are applets
- Lifecycle of Applet
- Methods of Applet class
- Compare Applet v/s Application
- Embedding Applet in HTML Page
- A code snippet to implement Applet

Introduction

Java language was launched on 23-Jan-1996(JDK 1.0) and at that time it only compatible CUI(Character User Interface) application. But in 1996 Visual Basic of Microsoft was favored for GUI programming. So the Java developers in rush (i.e. within 7 days) have given the compatible for GUI from Operating System (OS). Now, the mechanism like button, etc. was platform-dependent (i.e. in each platform there will be different size, shape button). But they did the connection of such mechanism from all platforms and gave a small library which contains these intersections and it is available in AWT(Abstract Window Toolkit) technology but it doesn't have advanced features like dialogue box, etc.

Now to execute Applet, javalanguagerequires a browser and at that time only "Internet Explorer" was there of Microsoft but Microsoft believes in control. So "SUN Micro-System" (the company which developed Java language) tie-up with other company known as "Netscape" (which developed Java Script) and now a days the "Netscape" company is also known as "Mozilla Firefox" which we all be familiar with is a browser. Now, these two companies have urbanized a technology called "SWING" and the advantage is that the SWING mechanisms are produced by Java language itself. Therefore now it is independent as well as some additional attributes have also been plug-in which were not in AWT technology. So we can articulate that SWING is much more advanced as compared to AWT technology.

5.1.1 Introduction to Applets

Two Types of Applets

It is important to define at the beginning that there are two varieties of applets. The first are those based directly on the Applet class. These applets employ the Abstract Window Toolkit (AWT) to give the graphic user interface (or use no GUI at all). This style of applet has been available since Java language was first produced.

The second types of applets are those which are based on the Swing class JApplet. Swing applets utilize the Swing classes to offer the Graphical User Interface. Swing offers a better-off and often easier-to-use user interface than does the AWT. Therefore, Swing-based applets are now the most popular. However, traditional

© Amity University Online

Notes

AWT-based applets are still used by the developers, especially when only a very straightforward user interface is necessary. Consequently, both AWT- and Swing-based applets are applicable.

Because JApplet encapsulate Applet, all the features and characteristics of Applet are also accessible in JApplet, and most of the information is applies to both types of applets.

Applets

Applets are little applications that are examined on an Internet server, elated over the Internet, without human intervention installed, and execute as part of a web document. After an applet retrieve on the client side, it has restricted access to resources so that it can construct a graphical user interface and run intricate computations without introducing the threat of viruses or breaching data reliability.

The essentials connected to the construction of an applet are accessible below, because applets are not planned in the same way as the programs that have been used so far. Applets vary from console-based applications in a number of key areas.

Let's start with the simple applet code shown below:

```
import java.awt.*;

import java.applet.*;

public class SimpleAppletdemo extends Applet {

public void paint(Graphics a) {

a.drawString("A Simple Java Applet", 10, 10);

}

}
```

This applet starts with two import statements. The first invoke the Abstract Window Toolkit (AWT) classes. Applets work together with the end-user (either directly or indirectly) with the help of AWT, not with the help of console-based I/O classes. The AWT includes sustain for a window-based, graphical user interface. As developer might expect, the AWT is quite big and sophisticated. Luckily, this straightforward applet makes very limited use of the AWT. The second call statement invokes the applet package, which include the class Applet. Every applet that a developers create must be a subclass of Applet.

The next line in the code defines the class SimpleAppletdemo. This class must be defined as public, because it will be used by code that is exterior the program.

Inside SimpleAppletdemo, paint( ) function is defined. This method is declared in AWT and has to be overridden by the applet. paint( ) is called each time that the applet must redisplay its result. This situation can arise for quite a lot of reasons. For example, the window in which the applet is executing can be overwritten by an additional window and then uncovered. Or, the applet window can be minimized and then come back. paint( ) is also called when the applet starts execution. Whatever the reason, whenever the applet have to redraw its result, paint( ) is called. The paint( ) method has one

(c) Amity University Online

Notes

stricture of type Graphics. This parameter includes the graphics context, which defines the graphics environment in which the applet is executing. This context is utilized whenever output to the applet is needed.

Inside paint( ) is a invoke to drawString( ), which is a associate of the Graphics class. This method outputs a string start at the particular a,b location. It has the following syntax:

void drawString(String message, int a, int b)

Here, message is the string to be output starting at a,b. In Java language window, the upper-left corner is location 0,0. The call to drawString( ) in the applet causes the message “A Simple Java Applet” to be displayed starting at location 10,10.

If you have pay attention to the code you would have realize that the applet does not have a main( ) method. Unlike Java language code, applets do not start execution at main( ). Actually, many of applets don’t still have a main( ) method. In its place, an applet starts execution when the name of its class is passed or invoked to an applet viewer or to a network browser.

After developer enter the source code for SimpleApplet, compile it in the same way that developer have been compiling codes. Though, executing SimpleApplet involves a special procedure. In fact, there are two ways in which developer can execute an applet:

1. Running the applet inside a Java-compatible web browser.
2. Using an applet viewer, such as the regular tool, appletviewer. An applet viewer runs developer applet in a window. This is usually the quickest and simplest way to check your applet

   Both of these methods are defined below.

To run an applet in a web browser, developer need to write a short HTML text file that includes a tag that loads the applet. At present, Java language suggests using the APPLET tag for this purpose. Here is the HTML code that runs SimpleApplet:

<applet code=”SimpleApplet” width=300 height=80>

</applet>

The width and height expression defines the dimensions of the display area used by the applet. After developercreate this file, developer can run this on his/her browser and then load this file, which causes SimpleApplet to be executed.

To runSimpleApplet with an applet viewer, developer might also execute the HTML file shown above. For example, if the earlier HTML file is known as RunAppdemo.html, then the following command line will executeSimpleApplet:

C:\>appletviewer RunAppdemo.html

Though, a more suitable method exists that developer can use to accelerate testing. Simply contain a comment at the head of his/her Java language source code file that includes the APPLET tag. By doing so, developer code is documented with a prototype of the needed HTML statements, and developer can test your compiled

(c) Amity University Online

Notes

applet just by opening the applet viewer with your Java language source code file. If developers use this method, the SimpleApplet source code file looks like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=300 height=80>
</applet>
*/
public class SimpleAppletdemo extends Applet {
public void paint(Graphics a) {
a.drawString("A Simple Java Applet", 10, 10);
}
}
```

Important Point to remember

- All applets are sub-classes (either directly or indirectly) of java.applet.Applet class.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at main() method.
- Output of an applet window is not performed by System.out.println(). Rather it is handled with various AWT methods, such as drawString().

Applet Architecture

An applet is a window-based program or code. As such, Applet architecture is unlike from the console-based programs.

First, applets are derived by an event. It is significant to recognize in a broad way how the event-driven architecture impacts the plan of an applet. An applet seems like a set of interrupt service routines. An applet waits till an event happened. The run-time system informs the applet about an event by invoking an event handler that has been given by the applet. Once this occurs, the applets have to take suitable action and then swiftly return.

This is a vital point. For the majority part, developer applet should not enter a "mode" of operation in which it manages control for an extensive period. In its place, it must carry out precise actions in reply to events and then return control to the run-time system. In those phases in which developer applet needed to carry out a frequent task on its own (for example, displaying a scrolling message across its window), developer must start an extra thread of execution.

(c) Amity University Online

Notes

Second, the end-user starts interacting with an applet not the other way around. As you know, in a non windowed program, when the program needs contribution from the user, it will prompt the user and then invoke some input method, such as readLine( ). But this not works in an applet. In its place, the user connects with the applet as they wants, when they want.

These communications are being forwarded to the applet as proceedings to which the applet must act in response. For example, when an end-user presses any key inside the applet's window, a key-pressed event is generated. Whenever the end user communicates with one of these controls, always there will be an event which is generated.

While the architecture of an applet is not as simple to recognize as that of a console-based program, Java language makes it easier. If developer has written programs for Windows, developer must know how threatening that environment can be. Luckily, Java language provides a much cleaner approach that is more rapidly mastered.

5.1.2 Lifecycle of Applet

All but the most insignificant applets override a set of methods that gives the basic instrument by which the browser or applet viewer interfaces to the applet and manages its execution. Four of these functions are init( ), start( ), stop( ), and destroy( ), relate to all applets and are decaled by Applet. Default implementations for all of these methods are given. Applets do not reacquire to override those methods they do not employ. Though, only very simple applets will not require declaring all of them.

AWT-based applets will also supersede the paint( ) method, which is declared by the AWT Component class. This method is invoked when the applet's output need to be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.). Example showing how 5 methods assembled

```
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width 320 height=120>
</applet>
*/
public class AppletSkeldemo extends Applet {
// Called first.
public void init() {
}
/* Called second, after init(). Also called whenever
the applet is restarted. */
```

(c) Amity University Online

Notes

```
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
}
```

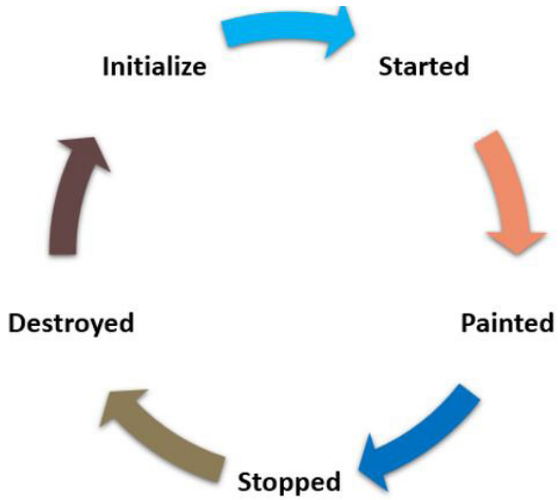
Output



Applet Life Cycle:

(c) Amity University Online

Notes



1. init( ): The init( ) method is the first method which is invoked. This is where developer should initialize the variables which will be used in program. This method is called only once during the execution time of your applet. It can be represent as:

```
public void init()
{
//Define variables//
}
```

2. start( ): This is the second phase called after init(). It is also invoked to restart an applet after it has been stopped. Whereas init( ) is called once—the first time when an applet is loaded—start( ) is called all time an applet's HTML document is showed onscreen. So, if a end-user leaves a web page and return back, the applet resumes running from start( ). This can be represent as:

```
public void start()
{
// To start the applet code
}
```

3. paint( ) The paint( ) method is invoked each time your applet's result must be redrawn. This situation can happen for quite a few reasons. For example,the applet window possibly will be minimized and then restored or the window in which the applet is running may be hide by another window and then unhide. paint( ) is also called when the applet starts execution. There might be any reason but when the applet must redraw its output, paint( ) method is invoked. The paint( ) method has only single parameter which of Graphics type. This parameter will include the graphics context, which declares the graphics environment in which the applet is running. It can be expressed as:

```
public void paint(Graphics graphics)
{
// Any shape's code
```

(c) Amity University Online

Notes

}

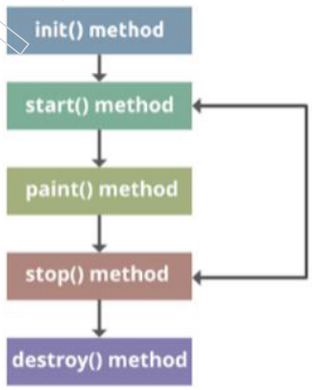
4. stop( ): The stop( ) method is invoked when a web browser exits the HTML document including the applet—when it goes to another page, for example. When stop( ) is invoked, the applet is probably executing. Developer should use stop( ) to hang threads that don't require to execute when the applet is not able to be seen. You can restart them when it appears again and invoke the start( ) if the user returns to the page. Its syntax will look like this.

```
public void stop()
{
 // To stop the applet code
}
```

5. destroy( ): The destroy( ) method is invoked when the environment defines that your applet desires to be detached completely from memory. At this point, developer should free up any resources the applet may be utilizing. It can be represent as:

```
public void destroy()
{
 // To destroy the applet
}
```

The stop( ) method is always called previous to destroy( )



Program to demonstrate How applet life cycle works:

```
import java.awt.Graphics;
import javax.swing.JApplet;
public class Triangledemo extends JApplet {
    int a[] = new int[3];
    int b[] = new int[3];
    int painted;
    //First phase of applet life cycle
```

(c) Amity University Online

Notes

```
    public void init()
    {
        infomsgdemo("init");
        double c = 81;
        double d = 100;
        double e = 100;
        double phi = -Math.PI / 2.0;
        double delta = 2.0*Math.PI / 3.0;
        for (int i=0; i<3; i++) {
            a[i] = (int)(d + c*Math.cos(phi+i*delta));
            b[i] = (int)(e + c*Math.sin(phi+i*delta));
        }
        painted = 0;
    }
//Second phase of applet life cycle
    public void start()
    {
        infomsgdemo("start");
    }
//Third phase of applet life cycle
    public void stop()
    {
        infomsgdemo("stop");
    }
//Fourth phase of applet life cycle
    public void destroy()
    {
        infomsgdemo("destroy");
    }
//Fifth phase of applet life cycle
    public void paint(Graphics g)
    {
```

(c) Amity University Online

Notes

```
            infomsgdemo("paint " + ++painted);
            g.drawLine(a[0],b[0],a[1],b[1]);
            g.drawLine(a[1],b[1],a[2],b[2]);
            g.drawLine(a[2],b[2],a[0],b[0]);
    }
            public void infomsgdemo(String str)
    {
            System.out.println(str);
    }
}
```

5.3.3 Methods of Applet class

getDocumentBase( ) and getCodeBase( )

Time and again, Developer will construct applets that will require to explicitly loading media and text. Java language will permits the applet to load data from the index holding the HTML file that initiated the applet (the document base) and the index from which the applet's class file was loaded (the code base). These directories are gives back as URL objects by getDocumentBase( ) and getCodeBase( ). They can be joined with a string that names the file developer wants to load. To really load another file, developer will utilize the showDocument( )method declared by the AppletContextinterface.The below applet show how to usethesegetDocumentBase( ) and getCodeBase( ) methods:

```
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=320 height=150>
</applet>
*/
public class Basesdemo extends Applet{
// Display code and document bases.
public void paint(Graphics a) {
String i;
URL url = getCodeBase(); // get code base
i = "Code base: " + url.toString();
```

(c) Amity University Online

```
a.drawString(i, 20, 30);
url = getDocumentBase(); // get document base
i = “Document base: “ + url.toString();
a.drawString(msg, 30, 50);
}
}
```

Output:



AppletContext and showDocument( )

One application of Java language is to use lively images and animation to offer a graphical means of navigating the Web that is more striking than easy text-based links. To permit your applet to shift control to another URL, developer must use the showDocument( ) method declared by the AppletContext interface.

Applet Context is an interface that allows developer to get information from the applet's running environment. The context of the at present executing applet is get by a call to the getAppletContext( ) method declared by Applet.

Inside an applet, once developer have obtained the applet's context, developer can get another document into view by invoking showDocument( )method. This showDocument() method has no return value and throws no exception if it fails to execute, so need to use it cautiously.

There are two showDocument( ) methods. The method showDocument(URL) shows the document at the particular URL.

The method showDocument(URL, String) shows the particular document at the particular location inside the browser window. Developer can also identify a name, which causes the document to be revealed in a new browser window by that name.

Example of applet demonstrates AppletContext and showDocument( ).

```
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code=”ACDemo” width=320 height=100>
</applet>
*/
```

(c) Amity University Online

Notes

```
public class ACDemo extends Applet{
public void start() {
AppletContext ac = getAppletContext();
URL url = getCodeBase(); // get url of this applet
try {
ac.showDocument(new URL(url+"Test.html"));
} catch(MalformedURLException e) {
showStatus("URL not found");
}
}
}
```

Method	Description
Applet getApplet(String appletName)	Returns the applet specified by appletName if it is within the current applet context. Otherwise, null is returned.
Enumeration<Applet> getApplets()	Returns an enumeration that contains all of the applets within the current applet context.
AudioClip getAudioClip(URL urt)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by urt.
Image getimage(URL ur)	Returns an Image object that encapsulates the image found at the location specified by url.
InputStream getStream(String key)	Returns the stream linked to key. Keys are linked to streams by using the setStream() method. A null reference is returned if no stream is linked to key.
Iterator<String> getStreamKeys()	Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See getStream() and setStream()
void setStream(String key InputStream strm)	Links the stream specified by strm to the key passed in key The key is deleted from the invoking object if strm is null.
void showDocument(URL urf)	Brings the document at the URL specified by url into view. This method may not be supported by applet viewers.
void showDocument(URL urf,	Brings the document at the URL specified by urf into view. This method may not be supported by applet viewers. The placement of the document is specified by where as described in the text.
void showStatus(String str) String where)	Displays str in status window.

The Methods Defined by the AppletContext Interface

(c) Amity University Online

Notes

Passing Parameters to Applets

The APPLET tag in HTML permits developer to pass parameters to your applet. To get back a parameter, developer use the getParameter( ) method. It returns the value of the particular parameter in the form of a String object. Therefore, for Boolean and numeric values, developer will required to transform their string representations into their internal Types.

Example to explain passing parameters:

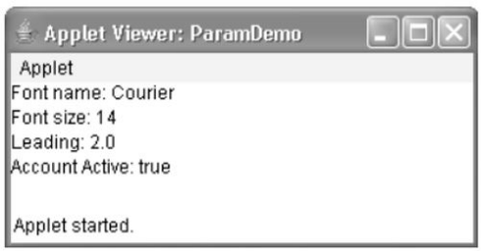
```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code=”ParamDemo” width=200 height=60>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet{
String a;
int b;
float leading;
booleanactive;
// Initialize the string to be displayed.
public void start() {
String param;
a = getParameter(“a”);
if(a == null)
a = “Not Found”;
param = getParameter(“b”);
try {
if(param != null) // if not found
b = Integer.parseInt(param);
else
```

(c) Amity University Online

Notes

```
catch(NumberFormatException e) {
b = -1;
}
param = getParameter("leading");
try {
if(param != null) // if not found
leading = Float.valueOf(param).floatValue();
else
leading = 0;
} catch(NumberFormatException e) {
leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g) {
g.drawString("Font name: " + a, 0, 10);
g.drawString("Font size: " + b, 0, 26);
g.drawString("Leading: " + leading, 0, 42);
g.drawString("Account Active: " + active, 0, 58);
}
}
```

Output



Using the Status Window

In addition to showing information in its window, an applet is able to generate result

(c) Amity University Online

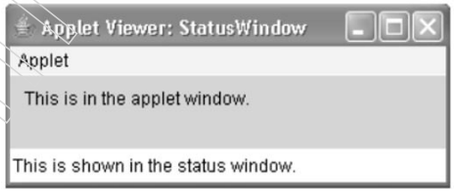
Notes

message to the status window of the browser or applet viewer on which it is executing. For this developer require to invoke showStatus( ) with the string that developer want displayed. The status window is an ideal place to give the end-user advice about what is happening in the applet, put forward options, or possibly report some types of issues. The status window also makes an excellent debugging help, because it gives developer a straightforward way to output information about your applet.

Example of applet using showStatus( ):

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=200 height=30>
</applet>
*/
public class StatusWindow extends Applet{
public void init() {
setBackground(Color.blue);
}
// Display msg in applet window.
public void paint(Graphics a) {
a.drawString("This is in the applet window.", 20, 30);
showStatus("This is shown in the status window.");
}
}
```

Output



Applet Display Methods:

To set the background color of an applet's window, developer need to use setBackground( ). To set the foreground color (the color in which text is shown, for example), use setForeground( ). These methods are declared by Component, and they have the following syntax:

```
void setBackground(Color newColor)
```

© Amity University Online

Notes

void setForeground(Color newColor)

Here, newColor defines the new color which developer want to use. The class Color declares the constants revealed here that can be used to identify colors: Colors code are given below

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lighGray	

An excellent place to set the foreground and background colors is inside the init( ) method. Certainly developer can change these colors as frequently as essential during the running of his/her applet.

Developer can obtain the current settings for the background and foreground colors by invoking getBackground( ) and getForeground( ), in that order. They are also declared by Component and are shown here:

Color getBackground( )

Color getForeground( )

Example show in how to use these methods:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
String a;
// set the foreground and background colors.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
a = "Inside init( ) --";
}
```

(c) Amity University Online

Notes

```
public void start() {
a += " Inside start( ) --";
}
// Display a in applet window.
public void paint(Graphics g) {
a += " Inside paint( ).";
g.drawString(a, 10, 30);
}
}
```

Output:



A Simple Banner Applet

To demonstrate repaint( ), a simple banner applet is urbanized. This applet moves a message, from right to left, across the applet's window. Since the moving of the message is a recurring task, it is done by another thread, created by the applet when it is started. The banner applet is shown here:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/
public class SimpleBannerDemo extends Applet implements Runnable {
String s = " A Simple Moving Banner.";
Thread a = null;
int state;
booleanstopFlag;
// Set colors and initialize thread.
public void init() {
setBackground(Color.cyan);
```

(c) Amity University Online

Notes

```
setForeground(Color.red);
}
// Start thread
public void start() {
a = new Thread(this);
stopFlag = false;
a.start();
}
// Entry point for the thread that runs the banner.
public void run() {
char c;
// Display banner
for( ; ; ) {
try {
repaint();
Thread.sleep(250);
c = s.charAt(0);
s = s.substring(1, s.length());
s += c;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
// Pause the banner.
public void stop() {
stopFlag = true;
a = null;
}
// Display the banner.
public void paint(Graphics g) {
g.drawString(s, 50, 30);
```

(c) Amity University Online

Notes

```
}
}
```

Output



Let's see how this applet work. First, notice that SimpleBannerDemo extends Applet, as predicted, but it also implements Runnable. This is essential, since the applet will be constructing a second thread of execution that will be used to move the banner. Inside init( ), the foreground and background colors of the applet are declared.

After starting, the run-time system invoksstart( ) to start the applet running. Inside start( ), a new thread of execution is constructed and allocated to the Thread variable a. Then, the boolean variable stopFlag, which controls the running of the applet, is kept to false. Next, the thread is initiated by a invoke to a.start( ). Remember that a.start( ) calls a method declared by Thread, which causes run( ) to begin executing. It does not basis a call to the version of start( ) declared by Applet. These are two separate methods.

Inside run( ), the characters in the string contained in s are frequently rotated left. flanked by each rotation, a call to repaint( ) is complete. This finally causes the paint( ) method to be called, and the current contents of s are displayed. Between each iteration, run( ) sleeps for a section of a second. The net effect of run( ) is that the contents of s are scrolled right to left in a continuously moving display. The stopFlag variable is checked on each step. When it is true, the run( ) method ends.

The AudioClip Interface

The AudioClip interface declares methods: play( ) (play a clip from the starting), stop( ) (end playing the clip), and loop( ) (play the loop endlessly). Developer can load an audio clip using getAudioClip( ) method.

5.1.4 Applet v/s Application

Java Application:

Java Application is just like a Java language program that executes on a fundamental operating system with the support of a virtual machine. It is also recognized as an application program. The graphical user interface is not essential to run the java applications.

(c) Amity University Online

Notes

Features of java Application

1. These are very similar to the Java programs.
2. Developer can independently execute these without making use of the web browser.
3. The execution of these apps requires a main function.
4. A Java application is capable of executing various programs via the local system.
5. The applications have full access to the local file systems and also networks.
6. A user needs an application program when they need to perform a task directly.

Java Applet:

An applet is a Java language program that can be entrenched into a web page. It runs within the web browser and works at user side. An applet is surrounded in an HTML page using the APPLET or OBJECT tag and hosted on a web server. Applets are used to create the web site more lively and enjoyable.

Features of java Applet

1. A Java applet is a small Java program.
2. Developer can easily include a Java applet along with various HTML documents.
3. Execution of these applets does not require any main function.
4. One needs a web browser (Java-based) for the execution of the applets.
5. They don't have any network access or local disk.
6. These can only access the services that are browser-specific.
7. These can't perform the execution of the programs from the local machines.
8. A Java applet cannot establish any access to the local system.
9. One requires a Java applet for performing small tasks. Conversely, it can also serve as a part of any task.

Difference between Applet and application

Parameters	Java Application	Java Applet
Meaning and Basics	A Java Application is a type of program that can get independently executed on a computer.	A Java Applet is a small program that makes use of another application program so that we can execute it.
Main() Method	The execution of the Java application begins with the main() method The usage of the main() is a prerequisite here.	The Java applet initializes through the init(). It does not require the usage of any main() method.
Execution	It cannot run alone, but it requires IRE for its execution.	It cannot run independently but requires APIs for its execution [Ex APIs like Web API).

(c) Amity University Online

Notes

Installation	One needs to install a Jave application priorly and explicitly on a local computer.	A Java applet does not require any prior installation.
Communication among other Servers	it is possible to establish communication with the other servers.	it cannot really establish communication with the other servers.
Read and Write Operations	The Java applications are capable of performing the read and write operations on various files present in a local computer.	A Java applet cannot perform these applications on any local computer.
Restrictions	These can easily access the file or data present in a computer system or device.	These cannot access the file or data available on any system or local computers.
Security	Jave applications are pretty trusted, and thus, come with no security concerns.	Java applets are not very trusted Thus, they require security.

5.1.5 Embedding Applet in HTML Page

The HTML APPLET Tag

Sun Micro system currently suggests that the APPLET tag be used to initiate an applet from both an HTML document and from an applet viewer. An applet viewer will carry out each APPLET tag that it finds in a separate window, while web browsers will permit many applets on a single page.

The representation for a fuller form of the APPLET tag is shown below.

```
< APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
</APPLET>
```

Each of them is explained in the below table:

© Amity University Online

Notes

CODEBASE	CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.
CODE	CODE is a required attribute that gives the name of the file containing your applet's compiled .class file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
ALT	The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.
NAME	NAME is an optional attribute used to specify a name for the applet instance, Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use getApplet(), which is defined by the AppletContext interface.
WIDTH and HEIGHT	WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.
ALIGN	ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP BOTTOM, MIDDLE, BASELINE, TEXTTOP ABSMIDDLE, and ABSBOTTOM.
VSPACE and HSPACE	These attributes are optional VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.
PARAM NAME and VALUE	The PARAM tag allows you to specify applet-specific arguments in an HTML page. Applets access their attributes with the getParameter() method.

5.1.6 A code snippet to implement Applet

Displaying Image using applet:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class ImageDemo extends Applet {
private Image image;
private AppletContextcontext;
public void init() {
```

(c) Amity University Online

Notes

```
context = this.getAppletContext();
String iURL = this.getParameter("image");
if(iURL == null) {
iURL = "java.jpg";
    }
    try {
      URL url = new URL(this.getDocumentBase(), iURL);
      image = context.getImage(url);
    } catch (MalformedURLException e) {
e.printStackTrace();
      // Display in browser status bar
context.showStatus("Could not load image!");
    }
  }
  public void paint(Graphics s) {
context.showStatus("Displaying image");
s.drawImage(image, 0, 0, 200, 84, null);
s.drawString("www.demo.com", 35, 100);
  }
}
```

Playing Audio Using Applet

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet {
  private AudioClipclip;
  private AppletContextcontext;
  public void init() {
    context = this.getAppletContext();
    String aURL = this.getParameter("audio");
if(aURL == null) {
aURL = "default.au";
```

(c) Amity University Online

Notes

```
        }
        try {
          URL url = new URL(this.getDocumentBase(), aURL);
          clip = context.getAudioClip(url);
        } catch (MalformedURLException e) {
      e.printStackTrace();
      context.showStatus("Could not load audio file!");
        }
    }
    public void start() {
    if(clip != null) {
    clip.loop();
        }
    }
    public void stop() {
    if(clip != null) {
    clip.stop();
        }
    }
    }
```

Check your Understanding

1. To run an applet in a web browser, developer need to write a short _____ text file that includes a tag that loads the applet.
2. An applet is a window-based _____ or _____.
3. Developer applet should not enter a "mode" of operation in which it manages _____ for an extensive period.
4. Applets are derived by an _____ .
5. _____ is an interface that allows developer to get information from the applet's running environment.
6. To set the background color of an applet's window, developer need to use _____ .
7. Sun Micro system currently suggests that the _____ tag be used to initiate an applet from both an HTML document and from an applet viewer.

(c) Amity University Online

Notes

Summary

- An applet is a type of programme that is embedded in a website to generate dynamic content. It operates on the client side and runs within the browser.
- Applets are little applications that are examined on an Internet server, elated over the Internet, without human intervention installed, and execute as part of a web document.
- There are two ways in which developer can execute an applet:
  - Running the applet inside a Java-compatible web browser.
  - Using an applet viewer, such as the regular tool, appletviewer. An applet viewer runs developer applet in a window. This is usually the quickest and simplest way to check your applet
- It is significant to recognize in a broad way how the event-driven architecture impacts the plan of an applet. An applet seems like a set of interrupt service routines. An applet waits till an event happened.
- All but the most insignificant applets override a set of methods that gives the basic instrument by which the browser or applet viewer interfaces to the applet and manages its execution. Four of these functions are init( ), start( ), stop( ), and destroy( ), relate to all applets and are decaled by Applet.
- Java Application is just like a Java language program that executes on a fundamental operating system with the support of a virtual machine. It is also recognized as an application program. The graphical user interface is not essential to run the java applications.

Activity

1. Create a program to insert below image using applet?



Questions and exercises

1. What do you mean by applets?
2. Define lifecycle of Applet?
3. What are different methods of Applet class?
4. What is difference between Applet and Application?
5. How to insert Applet in HTML Page?
6. Write a code to implement Applet?

(c) Amity University Online

Notes

Glossary

- CUI: Character User Interface
- GUI: Graphic User Interface
- OS: Operating System
- AWT: Abstract Window Toolkit
- HTML: Hypertext Markup Language
- <applet code="SimpleApplet" width=300 height=80>
- </applet>
- The width and height expression defines the dimensions of the display area used by the applet. After developer creates this file, developer can run this on his/her browser and then load this file, which causes Simple Applet to be executed.
- init( ): The init( ) method is the first method which is invoked. This is where developer should initialize the variables which will be used in program.
- start( ): This is the second phase called after init(). It is also invoked to restart an applet after it has been stopped.
- paint( ) The paint( ) method is invoked each time your applet's result must be redrawn.
- stop( ): The stop( ) method is invoked when a web browser exits the HTML document including the applet
- destroy( ): The destroy( ) method is invoked when the environment defines that your applet desires to be detached completely from memory.

CODEBASE	CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.
CODE	CODE is a required attribute that gives the name of the file containing your applet's compiled .class file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
ALT	The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.
NAME	NAME is an optional attribute used to specify a name for the applet instance, Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use getApplet(), which is defined by the AppletContext interface.

(c) Amity University Online

Notes

WIDTH and HEIGHT	WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.
ALIGN	ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP BOTTOM, MIDDLE, BASELINE, TEXTTOP ABSMIDDLE, and ABSBOTTOM.
VSPACE and HSPACE	These attributes are optional VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.
PARAM NAME and VALUE	The PARAM tag allows you to specify applet-specific arguments in an HTML page. Applets access their attributes with the getParameter() method.

Further Readings:

1. An Introduction to Programming with Java Applets, Elizabeth Boese
2. Java Applet Powerpack, Tad Ringo, Robin Drake
3. Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors, Venkat Subramaniam
4. The Java Application Programming Interface, Frank Yellin and James Gosling
5. Developing Professional Java Applets, K. C. Hopson and Stephen E. Ingram

Check Your Understanding:

Answers

1. HTML
2. program or code
3. control
4. event
5. Applet Context
6. setBackground( )
7. APPLET

(c) Amity University Online

Notes

Unit - 5.2 : Introduction to Graphic class

Objectives

At the end of this unit, you will be able to understand:

- Describe Graphic class
- What are Methods in Graphic class
- A code snippet to create different shapes.
- How to use loops in applet

Introduction

The AWT supports a rich collection of graphics methods. All graphics are pinched relative to a window. This can be the major window of an applet, a child window of an applet, or a separate application window. The source of each window is at the top-left corner and is 0,0. Coordinates are described in pixels. All result to a window takes place all the way through a graphics context. A graphics context is embedded by the Graphics class and is received in two ways:



5.2.1 Introduction to Graphic class

The Graphics class declares a number of drawing functions. Each shape is able to be drawn edge-only or filled. Objects are drawn and filled in the at present selected graphics color, which is always black by default. When a graphics object is made that surpass the dimensions of the window, output is without human intervention clipped to fit into applet window.

The Graphics class can be defined as conceptual super class for all graphics contexts which permit an application to draw onto mechanism that can be display on various devices or against off-screen images as well.

A Graphics object includes all state information necessary for the basic rendering operations that Java language supports.

(c) Amity University Online

Notes

State information includes the following properties.

- The Component object on which to draw.
- A translation origin for rendering and clipping coordinates.
- The current clip.
- The current color.
- The current font.
- The current logical pixel operation function.
- The current XOR alternation color

Expression for Java.awt.Graphics class is:

public abstract class Graphics extends Object

Example using Graphic class

```
package com.tutorialspoint.gui;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
public class GraphicsDemo extends Frame {
   public GraphicsDemo(){
super("Java Demo");
prepareGUI();
  }
  public static void main(String[] args){
GraphicsDemoGraphicsDemo = new GraphicsDemo();
GraphicsDemo.setVisible(true);
  }
  private void prepareGUI(){
setSize(400,400);
addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEventwindowEvent){
System.exit(0);
      }
    });
```

(c) Amity University Online

Notes

```
  }
  public void paint(Graphics gp) {
gp.setColor(Color.GRAY);
    Font font = new Font("Courier New", Font.PLAIN, 20);
gp.setFont(font);
gp.drawString("Welcome to Java", 50, 150);
  }
}
```

Output



5.2.2 Methods in Graphic class

Some of the common method in graphic class:

Method	Description
drawstring (String str, int x, int y)	Draws the text given by the string str. The string is drawn using the current color and font of the graphics context. x specifies the position of the left end of the string. y is the y coordinate of the baseline of the string. The baseline is a horizontal line on which the characters rest. Some parts of the characters, such as the tail on a y or g. extend below the baseline
drawLine(int x1, int y1, int x2, int y2)	-Draws a line from the point (x1 y1) to the point (x2,y2) The line is drawn as if with a pen that hangs one pixel to the right and one pixel down from the (x,y) point where the pen is located. For example, if g refers to an object of type Graphics, then the command g.drawLine(x,y,x,y). which corresponds to putting the pen down at a point, colors the single pixel with upper left corner at the point (xy).
drawOval(int x; int y, int width, int height)	Draws the outline of an oval. The oval is one that just fits inside the rectangle specified by x, y, width, and height. If width equals height, the oval is a circle.
fillArc(int x; int y, int width, int height, int startAngle, int arcAngle)	Draw a filled-in arc. This looks like a wedge of pie, whose crust is the arc that would be drawn by the drawArc method.

(c) Amity University Online

Notes

draw Ract (Int x, int y, int width, int height)—	Draws the outline of a rectangle. The upper left comer is at (x,y), and the width and height of the rectangle are as specified. If width equals height, then the rectangle is a square. If the width or the height is negative, then nothing is drawn. The rectangle is drawn with the same pen that is used for drawLine(). This means that the actual width of the rectangle as drawn is width+1, and similarly for the height. There is an extra pixel along the right edge and the bottom edge. For example, if you want to draw a rectangle around the edges of the component, you can say drawRect(0, 0, getWidth()-1, getHeight()-1),” where g is a graphics context for the component. If you use gdrawRect(0, 0, getWidth(). getHeight) then the right and bottom edges of the rectangle will be drawn outside the component.
drawRoundRect(int x, Inty, int width, int height, int xdiam, int ydiam)	Draws the outline of a rectangle with rounded corners. The basic rectangle is specified by x, y, width, and height, but the corners are rounded. The degree of rounding is given by adiam and ydiam. The corners are arcs of an ellipse with horizontal diameter xdiam and vertical diameter ydiam. A typical value for xdiam and ydiam is 16, but the value used should really depend on how big the rectangle is
drawsDRect(int x, int y, int width, Int height, boolean raised)	Draws the outline of a rectangle that is supposed to have a three dimensional effect, as if it is raised from the screen or pushed into the screen. The basic rectangle is specified by x, y, width, and height The raised parameter tells whether the rectangle seems to be raised from the screen or pushed into it. The 3D effect is achieved by using brighter and darker versions of the drawing color for different edges of the rectangle. The documentation recommends setting the drawing color equal to the background color before using this method The effect won’t work well for some colors.
drawArc(int x, Inty, int width, int height, int startAngle, int arcAngle)	Draws part of the oval that just fits inside the rectangle specified by x, y, width, and height. The part drawn is an arc that extends arcAngle degrees from a starting angle at startAngle degrees. Angles are measured with 0 degrees at the 3 ddock position (the positive direction of the horizontal axis). Positive angles are measured counterdockwise from cero, and negative angles are measured clockwise. To get an arc of a circle, make sure that width is equal to height
illRoundRect(int x, Int y, int width, int height, int xdiam, int ydiam)	Draws a filled-in rounded rectangle

(c) Amity University Online

Notes

fillRect(int x, Inty, int width, int height)	Draws a filled-in rectangle. This fills in the interior of the rectangle that would be drawn by drawRect(x,y,width,height). The extra piel along the bottom and right edges is not included. The width and height parameters give the exact width and height of the rectangle For example, if you wanted to fill in the entire component, you could say'g AliRect(0, 0, getWidth(), getHeight());

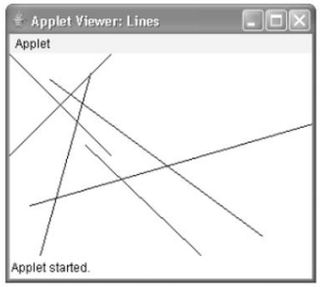
5.2.3 A code snippet to create different shapes.

1. Drawing Lines

Lines are drawn by using of the drawLine( ) method.

```
// Draw lines
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
public void paint(Graphics gp) {
gp.drawLine(0, 0, 80, 100);
gp.drawLine(0, 50, 100, 0);
gp.drawLine(40, 25, 150, 180);
gp.drawLine(75, 80, 400, 400);
gp.drawLine(20, 150, 400, 40);
gp.drawLine(5, 290, 65, 19);
    }
}
```

Output



(c) Amity University Online

Notes

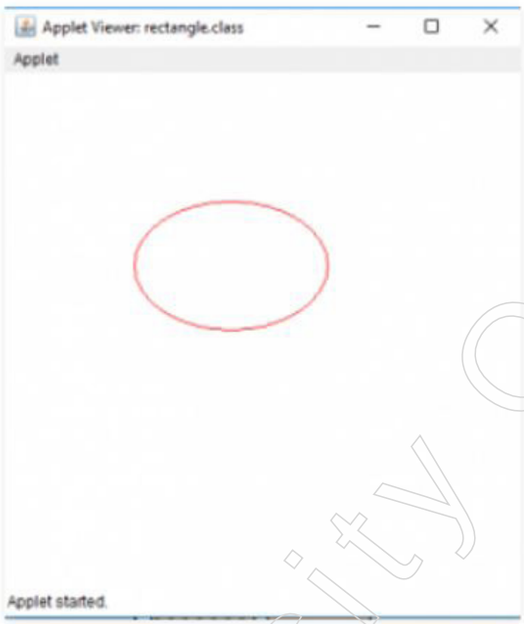
2. Program to draw a ellipse using drawLine function

```
import java.awt.*;
import javax.swing.*;
public class ellipse extends JApplet {
    public void init()
    {
            setSize(300, 300);
    }
    public void paint(Graphics g)
    {
center of the
            int x, y;
            x = 150;
            y = 175;
            double A = 75, B = 50, px = 0, py = 0;
            g.setColor(Color.red);
            for (int i = 0; i<= 360; i++) {
                    double x, y;
                    a = A * Math.sin(Math.toRadians(i));
                    b = B * Math.cos(Math.toRadians(i));
                    if (i != 0) {
                            // draw a line joining previous and new point .
                            g.drawLine((int)px + x, (int)py + y,
                                                    (int)x + x, (int)y + y);
                    }
                    // store the previous points
                    px = a;
                    py = b;
            }
    }
}
```

Output:

(c) Amity University Online

Notes

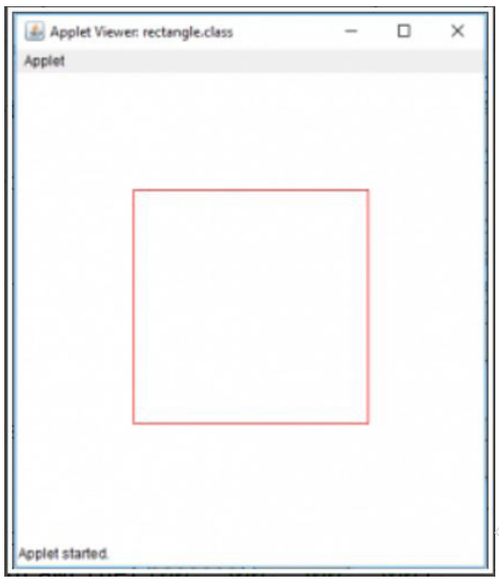


3. Draw a rectangle using drawLine(int x, int y, int x1, int y1)

```
import java.awt.*;
import javax.swing.*;
public class rectangle extends JApplet {
    public void init()
    {
            setSize(400, 400);
            repaint();
    }
    public void paint(Graphics gp)
    {
            // set Color for rectangle
            gp.setColor(Color.green);
            // draw a rectangle by drawing four lines
            gp.drawLine(100, 100, 100, 300);
            gp.drawLine(100, 300, 300, 300);
            gp.drawLine(300, 300, 300, 100);
            gp.drawLine(300, 100, 100, 100);
    }
}
```

(c) Amity University Online

Output:



Notes

4. **Drawing Multiple rectangles in program: In this code we are constructing four different types of rectangle, filled rectangle, rounded rectangle, filled round rectangle. They can be represented as:**

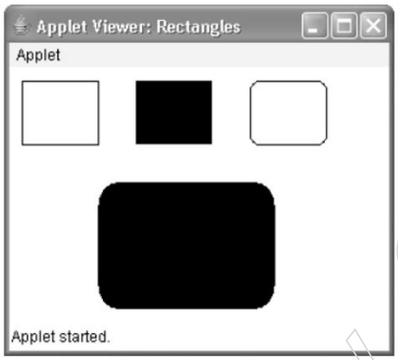
```
void drawRect(int top, int left, int w, int h);
void fillRect(int top, int left, int w, int h);
void drawRoundRect(int top, int left, int w, int h, int xDiam, int yDiam);
void fillRoundRect(int top, int left, int w, int h, int xDiam, int yDiam);
import java.awt.*;
import java.applet.*;
/*
<applet code=”Rectangles” width=300 height=200>
</applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics gp) {
gp.drawRect(20, 20, 60, 50);
gp.fillRect(110, 10, 60, 40);
gp.drawRoundRect(190, 20, 60, 50, 15, 15);
gp.fillRoundRect(70, 90, 160, 100, 30, 40);
}
```

(c) Amity University Online

Notes

```
}
```

Output



5. Java Program to draw a ellipse using drawOval(int x, int y, int width, int height)

```
import java.awt.*;
import javax.swing.*;
public class ellipse extends JApplet {
    public void init()
    {
            // set size
            setSize(400, 400);
            repaint();
    }
            public void paint(Graphics g)
    {
            g.setColor(Color.red);
            // draw a ellipse
            g.drawOval(100, 100, 150, 100);
    }
}
```

Output

(c) Amity University Online

Notes



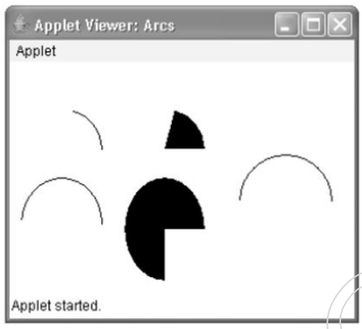
6. Drawing Arcs: Arc can be drawn using method drawArc( ) and fillArc( ). It can be represent as:

```
void drawArc(int top, int left, int width, int height, int startAngle,
int sweepAngle)
void fillArc(int top, int left, int width, int height, int startAngle,
int sweepAngle)
import java.awt.*;
import java.applet.*;
/*
<applet code=”Arcs” width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
public void paint(Graphics gp) {
gp.drawArc(10, 40, 70, 70, 0, 75);
gp.fillArc(100, 40, 70, 70, 0, 75);
gp.drawArc(10, 100, 70, 80, 0, 175);
gp.fillArc(100, 100, 70, 90, 0, 270);
gp.drawArc(200, 80, 80, 80, 0, 180);
    }
}
```

(c) Amity University Online

Notes

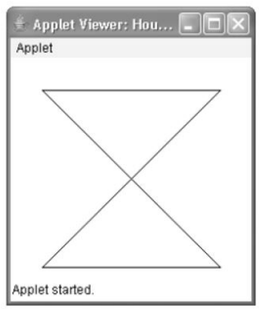
Output



7. Drawing Polygons Polygon can be draw using method drawPolygon( ) and fillPolygon( ), These can be represented as:

```
void drawPolygon(int x[ ], int y[ ], int numPoints)
void fillPolygon(int x[ ], int y[ ], int numPoints)
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/
public class HourGlass extends Applet {
public void paint(Graphics gp) {
int xpoints[] = {30, 200, 30, 200, 30};
int ypoints[] = {30, 30, 200, 200, 30};
int num = 5;
gp.drawPolygon(xpoints, ypoints, num);
    }
}
```

Output



(c) Amity University Online

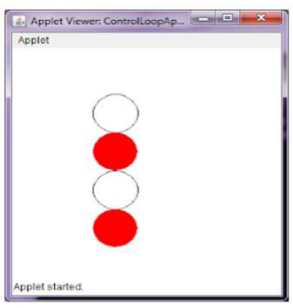
5.2.4 Using loops in applet

Notes

1. Implementing for loop in applet

```
import java.awt.*;
import java.applet.*;
 public class ControlLoopAppletdemo extends Applet
{
 public void paint(Graphics gs)
 {
for(int a=1;a<=4;a++)
 {
  if(a%2==0)
  {
gs.fillOval(90,a*50+10,50,50);
gs.setColor(Color.black);
  }
  else
  {
gs.drawOval(90,a*50+10,50,50);
gs.setColor(Color.red);
  }
 }
 }
}
/* <applet code=ControlLoopApplet width=300 height=300>
</applet> */
```

Output:



(c) Amity University Online

Notes

2. Implementing while loop in Applet

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class PieAppletWhile extends Applet {
public void paint (Graphics gp)
 {
   // The starting coordinates for each pie slice:
   int x = 50, y = 50;
   int w = 100, h = 100;
   // Number of slices. We'll set it to 20, as an example.
   int numSlices = 20;
   int degrees = (int) (360.0 / numSlices);
   int startAngle = 0;
   int slice = 1;
   // Loop:
   while (slice <= numSlices) {
     // First slice is different.
     if (slice != 1) {
       //Inital value of startAngle must be 0
startAngle = startAngle + degrees;
     }
     if (slice % 2 == 0) {
g.setColor (Color.yellow);
g.fillArc (x,y,w,h, startAngle, degrees);
     }
     else {
g.setColor (Color.red);
g.fillArc (x,y,w,h, startAngle, degrees);
     }
     // Increment counter:
     slice ++;
```

(c) Amity University Online

Notes

```
    } // end-while
  }
}
```

3. Loop in Animation applet

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class BouncingBallApplet extends Applet implements Runnable {
  int dd;
int ff;
  Thread animatorThread;
  public void init ()
  {
    // We'll pause 100 ms between frames.
dd = 100;
    // We are going to use "null" as a true/false indicator.
    // Need to initialize to "false" (null).
animatorThread = null;
  }
  public void start ()
  {
    // Initially, we're at frame 0.
ff = 0;
    // If there isnt' an animator thread, create one.
    if (animatorThread == null) {
animatorThread = new Thread (this);
    }
animatorThread.start ();
  }
  public void stop ()
  {
    // Stop the animation by indicating "false".
```

(c) Amity University Online

Notes

```
animatorThread = null;
 }
 public void run ()
 {
   // Potentially, an infinite loop!
   while (animatorThread != null) {
ff ++;
     // Draw next frame.
     repaint ();
     try {
Thread.sleep (dd);
     }
     catch (InterruptedException e) {
break;
     }
   }
 }
 public void paint (Graphics gp)
 {
   // Set foreground and background colors.
setBackground (Color.red);
gp.setColor (Color.white);
   // Here's how to get the size of the applet bounds within the program:
   // d.width is the width, d.height is the height.
   Dimension d = getSize();
   if (ff % 2 == 1) {
     // On odd frames, draw the ball at the top.
gp.fillOval (0, 0, d.width/2, d.height/2);
   }
   else {
     // On even frames, draw the ball at the bottom.
gp.fillOval (0, d.height/2, d.width/2, d.height/2);
```

(c) Amity University Online

```
    }
  }
}
```

Notes

Check your Understanding

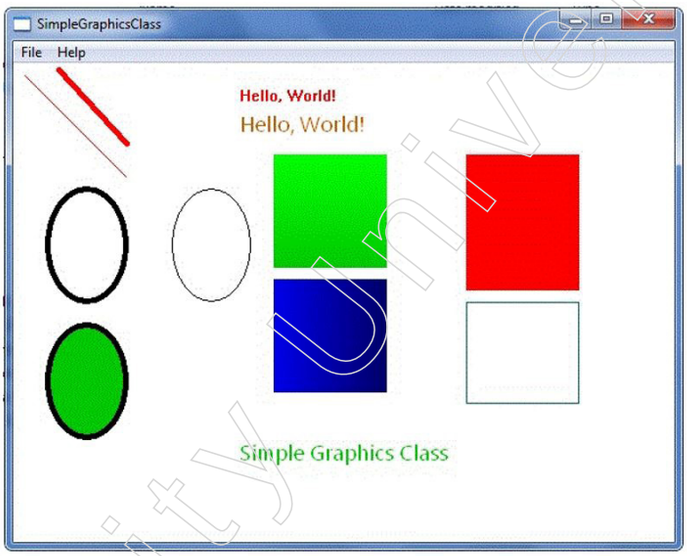
1. The Graphics class declares a number of _____ functions.
2. A _____ includes all state information necessary for the basic rendering operations that Java language supports.

Summary

- A number of drawing functions are declared in the Graphics class. Each shape can be drawn either edge-only or filled. Objects are drawn and filled in the graphics colour that is currently selected, which is always black by default. When a graphics object is created that exceeds the dimensions of the window, the output is clipped to fit into the applet window without the need for human intervention.

Activity

1. Create a code to using graphic class replicate below image.



Questions and exercises

1. What is Graphic class?
2. What are different methods in Graphic class?
3. How to use loops in applet?

Glossary

- AWT: Abstract Window Toolkit
- Graphics Class: The Graphics class can be defined as conceptual super class for all graphics contexts which permit an application to draw onto mechanism that can be display on various devices or against off-screen images as well.

© Amity University Online

Notes

Method	Description
drawstring (String str, int x, int y)	Draws the text given by the string str. The string is drawn using the current color and font of the graphics context. x specifies the position of the left end of the string. y is the y coordinate of the baseline of the string. The baseline is a horizontal line on which the characters rest. Some parts of the characters, such as the tail on a y or g. extend below the baseline
drawLine(int x1, int y1, int x2, int y2)	-Draws a line from the point (x1 y1) to the point (x2,y2) The line is drawn as if with a pen that hangs one pixel to the right and one pixel down from the (x,y) point where the pen is located. For example, if g refers to an object of type Graphics, then the command g.drawLine(x,y,x,y). which corresponds to putting the pen down at a point, colors the single pixel with upper left corner at the point (xy).
drawOval(int x; int y, int width, int height)	Draws the outline of an oval. The oval is one that just fits inside the rectangle specified by x, y, width, and height. If width equals height, the oval is a circle.
fillArc(int x; int y, int width, int height, int startAngle, int arcAngle)	Draw a filled-in arc. This looks like a wedge of pie, whose crust is the arc that would be drawn by the drawArc method.
draw Ract (Int x, int y, int width, int height)—	Draws the outline of a rectangle. The upper left comer is at (x,y), and the width and height of the rectangle are as specified. If width equals height, then the rectangle is a square. If the width or the height is negative, then nothing is drawn. The rectangle is drawn with the same pen that is used for drawLine(). This means that the actual width of the rectangle as drawn is width+1, and similarly for the height. There is an extra pixel along the right edge and the bottom edge. For example, if you want to draw a rectangle around the edges of the component, you can say drawRect(0, 0, getWidth()-1, getHeight()-1),” where g is a graphics context for the component. If you use gdrawRect(0, 0, getWidth().getHeight) then the right and bottom edges of the rectangle will be drawn outside the component.
drawRoundRect(int x, Inty, int width, int height, int xdiam, int ydiam)	Draws the outline of a rectangle with rounded corners. The basic rectangle is specified by x, y, width, and height, but the corners are rounded. The degree of rounding is given by adiam and ydiam. The corners are arcs of an ellipse with horizontal diameter xdiam and vertical diameter ydiam. A typical value for xdiam and ydiam is 16, but the value used should really depend on how big the rectangle is

(c) Amity University Online

Notes

drawsDRect(int x, int y, int width, Int height, boolean raised)	Draws the outline of a rectangle that is supposed to have a three dimensional effect, as if it is raised from the screen or pushed into the screen. The basic rectangle is specified by x, y, width, and height The raised parameter tells whether the rectangle seems to be raised from the screen or pushed into it. The 3D effect is achieved by using brighter and darker versions of the drawing color for different edges of the rectangle. The documentation recommends setting the drawing color equal to the background color before using this method The effect won't work well for some colors.
drawArc(int x, Inty, int width, int height, int startAngle, int arcAngle)	Draws part of the oval that just fits inside the rectangle specified by x, y, width, and height. The part drawn is an arc that extends arcAngle degrees from a starting angle at startAngle degrees. Angles are measured with 0 degrees at the 3 ddock position (the positive direction of the horizontal axis). Positive angles are measured counterdockwise from cero, and negative angles are measured clockwise. To get an arc of a circle, make sure that width is equal to height
illRoundRect(int x, Int y, int width, int height, int xdiam, int ydiam)	Draws a filled-in rounded rectangle
fillRect(int x, Inty, int width, int height)	Draws a filled-in rectangle. This fills in the interior of the rectangle that would be drawn by drawRect(x,y,width,height). The extra piel along the bottom and right edges is not included. The width and height parameters give the exact width and height of the rectangle For example, if you wanted to fill in the entire component, you could say'g AllRect(0, 0, getWidth(), getHeight());

Further Readings:

1. Graphic Java 1.2: Mastering the JFC, David M. Geary Java Applet Powerpack, Tad Ringo, Robin Drake
2. Java 2D graphics, Jonathan Knudsen
3. The Java Application Programming Interface, Frank Yellin and James Gosling
4. Developing Professional Java Applets, K. C. Hopson and Stephen E. Ingram

Check Your Understanding:

Answers

1. Drawing
2. Graphics object

(c) Amity University Online

Notes

Unit - 5.3 : Introduction to AWT

Objectives

At the end of this unit, you will be able to understand:

- What are AWT package
- Define the classess in AWT package
- What is the Role of Layout manager in applet
- A code snipet to implement applet with layout manager

Introduction

The AWT Abstract window toolkit includes several classes and methods that permit developer to create and manage windows. It is also the basis on which Swing is made. The AWT is quite big and a full explanation would easily fill an entire book. Therefore, it is not doable to describe in detail every AWT class, method, or instance variable.

Although a frequent use of the AWT is in applets, it is also utilized to construct stand-alone windows that execute in a GUI environment, such as Windows. To execute them, developer need to use an applet viewer or a Java-compatible web browser.

One other point before starting. Today, most Java language programs utilize user interfaces based on Swing. Because Swing provides better-off implementations than does the AWT of some general GUI controls, such as buttons, lists, and check boxes, it is easy to jump to the end that the AWT is no longer significant, that it has been outdated by Swing. This assumption is, however, quite incorrect. As mentioned, Swing is built on the AWT. Thus, many features of the AWT are also features of Swing.

5.3.1 Introduction to AWT package

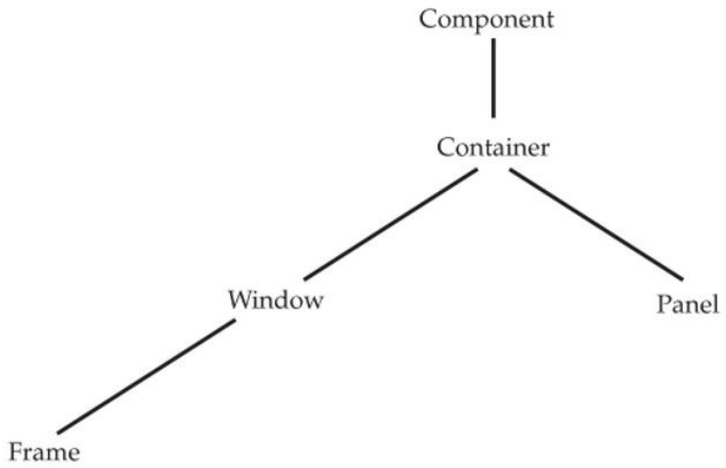
Java AWT (Abstract Window Toolkit) is an API to expand Graphical User Interface (GUI) or windows-based applications in Java language.

Java AWT components are platform-dependent i.e. components are showed according to the analysis of OS. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The java.awt package provides different classes for AWT API such as Event, choice, AWTevent, polygon, List, Robot, panel etc.

The AWT describes windows according to a class hierarchy that adds usability and specificity with each segments. The two most widespread windows are those which are carryout from Panel, which is utilized by applets, and those which were taken out from Frame, which construct a standard application window. Much of the performance of these windows is carries out from their parent classes. Consequently, a explanation of the class hierarchies connecting these two classes is essential to their perceptive. Below Figure displays the class hierarchy for Panel and Frame.

(c) Amity University Online



The class hierarchy for Panel and Frame

Component

In starting of the Abstract Window Toolkit hierarchy is the Component class. Component is a conceptual class that contains all of the characteristics of a visual component. All user interface basics that are showed on the monitor and that connect with the end-user are the subclasses of Component. It declares over a hundred public methods that are accountable for administrating events, such as mouse click and keyboard input by user, positioning of window and sizing the window. A Component object is accountable for identification of the recent foreground and background colors and the resently selected text font.

Container

The Container class is a child class for Component. It has multiple methods that permits other Component objects to be encapsulate in it. Other Container objects can be stocked within a Container. This makes it a multi-tiered containment structure. A container is accountable for positioning out several components that it includes.

Panel

The Panel class is an actual subclass of Container. It doesn't put in any fresh methods; it simply execute Container. A Panel may well be considering of as a repeatedly nestable, existing screen part. Panel is the superclass for Applet. When screen output is heading for an applet, it is traced on the exterior of a Panel object. In essence, a Panel is a window that does not enclose any of these title bar, menu bar, or border. This is why at client side they don't see this substance when an applet is execute with in a browser. When you execute an applet by means of an applet viewer, the applet viewer just only gives the title and border.

Other components can be added by a developer using a Panel object by its add( ) method. Once these mechanism have been added, developer can point and restructured them manually using the components that are declared by setLocation( ), setSize( ), setPreferredSize( ), or setBounds( ).

(c) Amity University Online

Notes

Window

The Window class constructs a top-level window. A top-level window is not included within any other object; it sits in a straight line on the monitor. Usually, developer won't create Window stuff directly. In its place, developer will use a subclass of Window called Frame, which is explained next.

Frame

Frame contains what is usually consideration as "window." It is a subclass of Window and unilike panel it has a title bar, menu bar, borders, and resizing corners. If developer constructs a Frame object from inside an applet, it will have a caution message, like "Java Applet Window," to the end-user that an applet window has been constructed. This communication warns end-users that the window they see was initiated by an applet and not by software executing on their computer. When a Frame window is produced by a stand-alone function rather than an applet, a normal window is formed.

5.3.2 Classes in AWT package

The AWT classes are included in the java.awt package. It is one of Java's language biggest packages. Luckily, it is logically ordered in a top-down, hierarchical fashion, it is easier to recognize some of the many AWT classes are list below in table.

Class	Description
AWT Event	Encapsulates AWT events.
AWT Event Multicaster	Dispatches events to multiple listeners.
Border Layout	The border layout manager Border layouts use five components: North, South, East West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
Card Layout	The card layout manager Card layouts emulate index cards. Only the one on top is showing.
Check box	Creates a check box control.
Check box group	Creates a group of check box controls.
Check box menultem	Creates an on/off menu item
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of Component that can hold other components.
Curson	Encapsulates a bitmapped cursor Creates a dialog window.
Dialong	Specifies the dimensions of an object. The width is stored in width and the height is stored in height
Event	Encapsulates events.
Event Queue	Queues events.
File Dialog	Creates a window from which a file can be selected.
Flow Layout	The flow layout manager. Flow layout positions components left to right, top to bottom.

(c) Amity University Online

Notes

Font	Encapsulates a type font.
Font Metrics	Encapsulates various information related to a font. This information helps you display text in a window.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
Graphics Device	Describes a graphics device such as a screen or printer.
Graphics Environment	Describes the collection of available Font and GraphicsDevice objects.
Grid Bag Constraints	Defines various constraints relating to the GridBagLayout class.
Grid Bag Layout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints.
Grid Layout	The grid layout manager. Grid layout displays components in a two-dimensional grid.
Image	Encapsulates graphical images.
Insets	Encapsulates the borders of a container.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
Media Tracker	Manages media objects.
Menu	Creates a pull-down menu.
Menu Bar	Creates a menu bar.
Menu Component	An abstract class implemented by various menu classes.
Menu Item	Creates a menu item.
Menu Shortcut	Encapsulates a keyboard shortcut for a menu item.
Panel	The simplest concrete subclass of Container.
Point	Encapsulates a Cartesian coordinate pair, stored in x and y.
Polygon	Encapsulates a polygon.
Pop up Menu	Encapsulates a pop-up menu.
Print Job	An abstract class that represents a print job.
Rectangle	Encapsulates a rectangle.
Robot	Supports automated testing of AWT-based applications.
Scrollbar	Creates a scroll bar control.
Scroll Pane	A container that provides horizontal and/or vertical scroll bars for another component.
System Colour	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
Text Area	Creates a multiline edit control.
Text Component	A superclass for TextArea and TextField.
Text Field	Creates a single-line edit control.

(c) Amity University Online

Notes

Handling Events in a Frame Window

Since Frame is a subclass of Component, it contains all the characteristics declared by Component. This means that developer can use and run a frame window just like developer administer an applet's main window.

```
// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="WindowEvents" width=280 height=40>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame
implements MouseListener, MouseMotionListener {
String s = "";
int msX=10, msY=40;
int mvX=0, mvY=0;
SampleFrame(String title) {
super(title);
// register this object to receive its own mouse events
addMouseListener(this);
addMouseMotionListener(this);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
}
// Handle mouse entered.
public void mouseEntered(MouseEventevtObj) {
```

(c) Amity University Online

Notes

```
// save coordinates
msX = 20;
msY = 64;
s = “Mouse just entered child.”;
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEventevtObj) {
// save coordinates
msX = 20;
msY = 64;
s= “Mouse just left child”;
repaint();
}
// Handle mouse pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
msX = me.getX();
msY = me.getY();
s = “Down”;
repaint();
}
// Handle mouse released.
public void mouseReleased(MouseEvent me) {
// save coordinates
msX = me.getX();
msY = me.getY();
s = “Up”;
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
```

(c) Amity University Online

Notes

```
// save coordinates
msX = me.getX();
msY = me.getY();
mvX = me.getX();
mvY = me.getY();
s = "*";
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// save coordinates
mvX = me.getX();
mvY = me.getY();
repaint(0, 0, 80, 60);
}
public void paint(Graphics gp) {
gp.drawString(s, msX, msY);
gp.drawString("Mouse at " + mvX + ", " + mvY, 10, 40);
}
}
class MyWindowAdapter extends WindowAdapter {
SampleFramesampleFrame;
public MyWindowAdapter(SampleFramesampleFrame) {
this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
sampleFrame.setVisible(false);
}
}
// Applet window.
public class WindowEvents extends Applet
implements MouseListener, MouseMotionListener {
```

(c) Amity University Online

Notes

```
SampleFramef;
String s = “”;
int msX=0, msY=10;
int mvX=0, mvY=0;
// Create a frame window.
public void init() {
f = new SampleFrame(“Handle Mouse Events”);
f.setSize(200, 100);
f.setVisible(true);
// register this object to receive its own mouse events
addMouseListener(this);
addMouseMotionListener(this);
}
// Remove frame window when stopping applet.
public void stop() {
f.setVisible(false);
}
// Show frame window when starting applet.
public void start() {
f.setVisible(true);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
// save coordinates
msX = 0;
msY = 24;
s = “Mouse just entered applet window.”;
repaint();
}
```

(c) Amity University Online

Notes

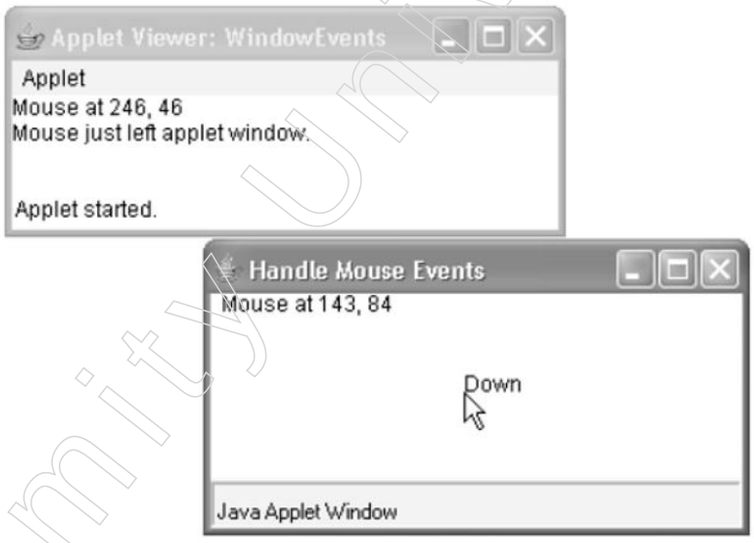
```
// Handle mouse exited.
public void mouseExited(MouseEvent me) {
// save coordinates
msX = 0;
msY = 24;
s = “Mouse just left applet window.”;
repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
msX = me.getX();
msY = me.getY();
s = “Down”;
repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me) {
// save coordinates
msX = me.getX();
msY = me.getY();
s = “Up”;
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
msX = me.getX();
msY = me.getY();
mvX = me.getX();
mvY = me.getY();
s = “*”;
```

(c) Amity University Online

Notes

```
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// save coordinates
mvX = me.getX();
mvY = me.getY();
repaint(0, 0, 100, 20);
}
// Display s in applet window.
public void paint(Graphics gp) {
gp.drawString(s, msX, msY);
gp.drawString("Mouse at " + mvX + ", " + mvY, 0, 10);
    }
}
```

Output



5.3.3 Role of Layout manager in applet

A layout manager is a request of any class that executes the LayoutManager interface. The layout manager is declared by the setLayout( ) method. If no invoke to setLayout( ) is made, then the default layout manager is utilized automatically.

The LayoutManagers are used to position components in a defined manner. The Java LayoutManagers helps developer to manage the positioning and size of the objects in GUI forms.

(c) Amity University Online

Notes

The setLayout method has a following syntax

void setLayout(LayoutManagerObj)

In above expression, Obj is a reference to the preferred layout manager. If developer wish to disable the layout manager and position components manually then he/she recquire pass null for Obj. If developer do this then developer will require to determine the figure and location of each component manually, using the setBounds( ) method declared by Component. Normally, developer will want to use a layout manager.

Java language has several predefined LayoutManager classes out which some are explained below:

BorderLayout

The BorderLayout class executes a common layout style for top-level windows. It has 4 narrow, fixed-width components at the edges and one large area in the center. The four sides are known to as north, south, east, and west. The core area is called the center. BorderLayout can be express as

BorderLayout( )

BorderLayout(int ha, int vb)

The first expression construct a default border layout. The second allows developer to identify the horizontal and vertical space left between components in ha and vb, in that order.

BorderLayout declares the following constants that identify the regions:

Border Layout.CENTER	Border Layout.SOUTH
Border Layout.EAST	Border Layout.WEST
Border Layout.NORTH	

Example for Border Layout:

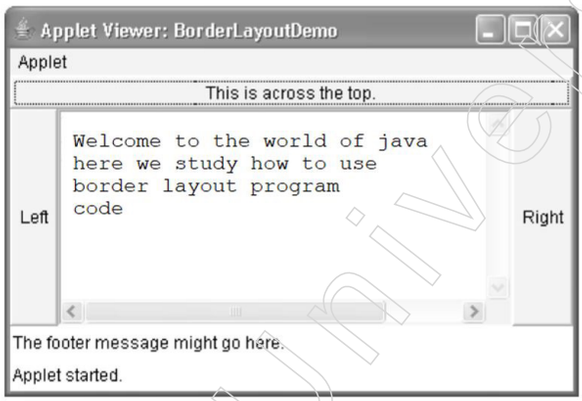
```
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code=”BorderLayoutDemo” width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet {
public void init() {
setLayout(new BorderLayout());
add(new Button(“This is across the top.”),
BorderLayout.NORTH);
```

(c) Amity University Online

Notes

```
add(new Label("The footer message might go here."),
BorderLayout.SOUTH);
add(new Button("Right"), BorderLayout.EAST);
add(new Button("Left"), BorderLayout.WEST);
String s = "Welcome to the world of Java " +
"here we study how to use;\n" +
"Border layout in program " +
"code.\n\n\n"
add(new TextArea(s), BorderLayout.CENTER);
    }
}
```

Output



FlowLayout

FlowLayout is the default layout. FlowLayout executes an easy layout style, which is like to how words flow in a text editor. The path of the layout is administered by the container's component direction property, which, by default, is left to right, top to bottom. So, by default, components are laid out line-by-line starting at the upper-left corner. In all cases, when a line is filled till the end, layout moves to the next line. A small space is left among each component, on top of and ends, as well as left and right. FlowLayout can be expressed as:

```
FlowLayout( )
FlowLayout(int ha)
FlowLayout(int ha, int ht, int vt)
```

The first statement creates the default layout, which enter components at center and leaves five pixels of space between each component. The second statement lets developer to specify how each line is needs to be aligned. Valid values for ha are as follows:

(c) Amity University Online

Notes

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

FlowLayout.LEADING

FlowLayout.TRAILING

The third statement lets developer to specify the horizontal and vertical space left among the components in ht and vt, respectively.

Checkbox code using left aligned flowlayout

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code=”FlowLayoutDemo” width=250 height=200>
</applet>
*/
public class FlowLayoutDemo extends Applet
implements ItemListener {
String s = “”;
Checkbox Win7, Win2010, Linux, macOS;
public void init() {
// set left-aligned flow layout
setLayout(new FlowLayout(FlowLayout.LEFT));
Win7 = new Checkbox(“Win07”, null, true);
Win2010 = new Checkbox(“Windows 2010”);
Linux = new Checkbox(“Linux”);
macOS = new Checkbox(“MacOS OS”);
add(Win7);
add(Win2010);
add(Linux);
add(macOS);
// register to receive item events
Win7.addItemListener(this);
```

(c) Amity University Online

Notes

```
Win2010.addItemListener(this);
Linux.addItemListener(this);
macOS.addItemListener(this);
}
// Repaint when status of a check box changes.
public void itemStateChanged(ItemEventie) {
repaint();
}
// Display current state of the check boxes.
public void paint(Graphics gp) {
s = “Current state: “;
gp.drawString(s, 6, 80);
s = “ Windows 7: “ + Win7.getState();
gp.drawString(s, 6, 100);
s = “ Windows 2010: “ + Win2010.getState();
gp.drawString(s, 6, 120);
s = “ Linux: “ + Linux.getState();
gp.drawString(s, 6, 140);
s = “ MacOS: “ + macOS.getState();
gp.drawString(s, 6, 160);
    }
}
```

Output:



(c) Amity University Online

Notes

GridLayout

GridLayout puts out components in a 2-D grid. When developer Invoke a GridLayout, developer define the number of rows and columns he/she requires. The GridLayout can be expressed as:

GridLayout( )

GridLayout(int nR, int nC)

GridLayout(int nR, int nC, int ht, int vt)

The first statements creates a single-column grid layout. The second statements create a grid layout with the particular number of rows and columns. The third statements lets developer to specify the horizontal and vertical space left between components in ht and vt, respectively. Either nR or nC can be zero. Specifying nR as zero result-in unlimited-length columns. Specifying nC as zero result-in unlimited-length rows.

CardLayout

The CardLayout class is exclusive between the other layout managers in that it stores quite a few diverse layouts. This can be helpful for user interfaces with elective components that can be animatedly enabled and disabled upon user contribution. Developer can arrange the other layouts and have them hidden, ready to be activated when required. CardLayout method can be represented as:

CardLayout( );

CardLayout(int ht, int vt);

The first statements creates a default card layout. The second statement lets developer to identify the horizontal and vertical space left between components in ht and vt respectively.

Use of a card layout needs a bit more work as compared to other layouts. The cards are characteristically seized in an object of type Panel. This panel have to have CardLayout selected as its layout manager. The cards that shape the deck are also characteristically objects of type Panel. Therefore, developers have to construct a panel that includes the deck and a panel for each card in the deck. After that, developer require to add to the suitable panel the components that shape each card. Developer then adds these panels to the panel for which CardLayout is the layout manager. Finally, developers add this panel to the window screen. Once these steps are done, Developer must present some method for the user to select among cards. One general approach is to comprise one push button for each card in the deck.

When all card panels are added to a panel, they are generally given a name. Therefore, usually, developer will use this statement of add( ) when adding cards to a panel:

void add(Component paneldemo, Object N)

Here, N is a string that identifies the name of the card whose panel is specified by paneldemo.

(c) Amity University Online

Notes

After developer has created a deck, developer program activates a card by invoking one of the following methods declared by CardLayout:

void first(Container d)

void last(Container d)

void next(Container d)

void previous(Container d)

void show(Container d, String cN)

Here, d is a reference to the container that kept the cards, and cN is the name of a card. Calling first( ) generate the first card in the d to be shown. To show the last card, call last( ) method to be invoked. To show the next card, call next( ) method to be invoked . To show the previous card, call previous( ) method to be invoked. Both next( ) and previous( ) without human intervention cycle back to the top or bottom of the d , respectively. The show( ) method shows the card whose name is passed in cN.

GridBagLayout

Even though the previous layouts are completely acceptable for loads of uses, some situations will require developer to take a bit more power over how the components needs to be arranged. A superior way to do this is to use a grid bag layout, which is defined by the GridBagLayout class. What makes the grid bag useful is that developer can define the relative placement of components by specifying their positions inside cells within a grid. The main benefit to the grid bag is that each component can have different size, and each row in the grid can have a different number of columns. This is why the layout is called a grid bag. It's a compilation of small grids joined together.

The location and size of each component in a grid bag are defined by a set of constraints associated to it. The constraints are included in an object of type GridBagConstraints. Constraints contains the height and width of a cell, and the position of a component, its alignment, and its anchor point inside the cell.

The common modus operandi for using a grid bag is to first generate a new GridBagLayout object and to make it the current layout manager. Then, set the constraints that be relevant to each component that will be added to the grid bag. Finally, add those components to the layout manager.

GridBagLayout can be declared as:

GridBagLayout()

The key to productively using GridBagLayout is the appropriate setting of the constraints, which are included in a GridBagConstraints object. GridBagConstraints declare several fields that developer can set to administer the size, placement, and spacing of a component. These are show in below table:

(c) Amity University Online

Notes

Fleid	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER.
int fill	Specifies how a component is resized if the component is smaller than its cell Valid values are GridBagConstraints NONE (the default). GridBagConstraints.HORIZONTAL GridBagConstraints. VERTICAL Grid BagConstraints.BOTH.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is Grid BagConstraints RELATIVE.
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is Grid BagConstraints RELATIVE
Insets insets	Specifies the Insets Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell The default is 0.
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

Constraint Fields Defined by GridBagConstraints

GridBagConstraints also declares several static fields that contain standard constraint values, such as GridBagConstraints.CENTER and GridBagConstraints. VERTICAL.

When a component is smaller than its cell, developer can use the anchor field to identify where inside the cell component's top-left corner will be situated. There are three types of values that you can give to anchor. The first are absolute declared below

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

As their names suggest, these values helps the component to be placed at the specific locations. The second type of values that can be given to anchor is relative, The relative values are shown here:

Notes

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

Their names describe the placement. The third type of values that can be given to anchor were added by Java SE 6. They allow you to position components vertically relative to the baseline of the row. They are shown here:

GridBagConstraints.BASELINE	GridBagConstraints. BASELINE_LEADING
GridBagConstraints.BASELINE_TRAILING	GridBagConstraints.ABOVE_ BASELINE
GridBagConstraints.ABOVE_BASELINE_ LEADING	GridBagConstraints.ABOVE_ BASELINE_TRAILING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_ BASELINE_LEADING
GridBagConstraints.BELOW_BASELINE_ TRAILING	

5.1.4 A Code Snippet to Implement Applet with Layout Manager

1. Creating a grid layout:

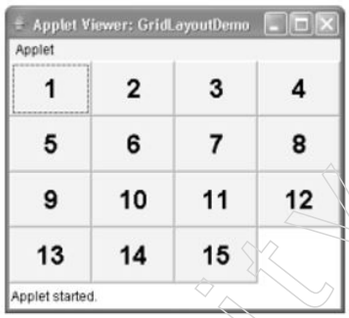
```
import java.awt.*;
import java.applet.*;
/*
<applet code=”GridLayoutDemo” width=300 height=200>
</applet>
*/
public class GridLayoutDemo extends Applet {
static final int a = 4;
public void init() {
setLayout(new GridLayout(a, a));
setFont(new Font(“SansSerif”, Font.BOLD, 24));
for(int x = 0; x < a; x++) {
for(int y = 0; y<a; y++) {
int k = x * a + y;
if(k > 0)
add(new Button(“” + k));
```

(c) Amity University Online

Notes

```
            }
        }
    }
    }
```

Output



2. Creating a program using GridBaglayout

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="GridBagDemo" width=250 height=200>
</applet>
*/
public class GridBagDemo extends Applet
implements ItemListener {
String msg = "";
Checkbox winXP, winVista, solaris, mac;
public void init() {
GridBagLayoutgB = new GridBagLayout();
GridBagConstraintsgbc = new GridBagConstraints();
setLayout(gB);
// Define check boxes.
winXP = new Checkbox("Windows XP ", null, true);
winVista = new Checkbox("Windows Vista");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
```

(c) Amity University Online

Notes

```
// Define the grid bag.
// Use default row weight of 0 for first row.
gC.weightx = 1.0; // use a column weight of 1
gC.ipadx = 200; // pad by 200 units
gC.insets = new Insets(4, 4, 0, 0); // inset slightly from top left
gC.anchor = GridBagConstraints.NORTHEAST;
gC.gridwidth = GridBagConstraints.RELATIVE;
gB.setConstraints(winXP, gC);
gC.gridwidth = GridBagConstraints.REMAINDER;
gB.setConstraints(winVista, gC);
// Give second row a weight of 1.
gC.weighty = 1.0;
gC.gridwidth = GridBagConstraints.RELATIVE;
gB.setConstraints(solaris, gC);
gC.gridwidth = GridBagConstraints.REMAINDER;
gB.setConstraints(mac, gC);
// Add the components.
add(winXP);
add(winVista);
add(solaris);
add(mac);
// Register to receive item events.
winXP.addItemListener(this);
winVista.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
// Repaint when status of a check box changes.
public void itemStateChanged(ItemEventie) {
repaint();
}
// Display current state of the check boxes.
```

(c) Amity University Online

Notes

```
public void paint(Graphics g) {
msg = “Current state: “;
g.drawString(msg, 6, 80);
msg = “ Windows XP: “ + winXP.getState();
g.drawString(msg, 6, 100);
msg = “ Windows Vista: “ + winVista.getState();
g.drawString(msg, 6, 120);
msg = “ Solaris: “ + solaris.getState();
g.drawString(msg, 6, 140);
msg = “ Mac: “ + mac.getState();
g.drawString(msg, 6, 160);
    }
}
```

Output



3. Creating a program using cardlayout

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code=”CardLayoutDemo” width=300 height=100>
</applet>
*/
public class CardLayoutDemo extends Applet
implements ActionListener, MouseListener {
```

(c) Amity University Online

Notes

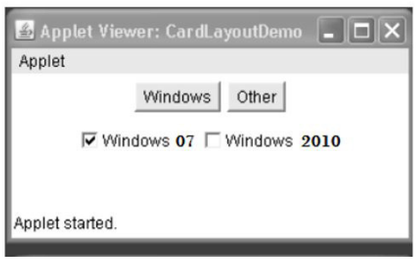
```
Checkbox win07, win2010, linux, mac;
Panel osCards;
CardLayoutcardLO;
Button Win, Other;
public void init() {
Win = new Button("Windows");
Other = new Button("Other");
add(Win);
add(Other);
cardLO = new CardLayout();
osCards = new Panel();
osCards.setLayout(cardLO); // set panel layout to card layout
win07 = new Checkbox("Windows 07", null, true);
win2010 = new Checkbox("Windows 2010");
linux = new Checkbox("linux");
mac = new Checkbox("Mac OS");
// add Windows check boxes to a panel
Panel winPan = new Panel();
winPan.add(win07);
winPan.add(win2010);
// add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(linux);
otherPan.add(mac);
// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");
// add cards to main applet panel
add(osCards);
// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);
```

(c) Amity University Online

Notes

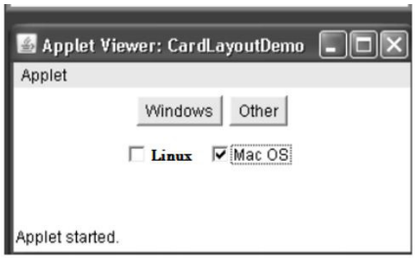
```
// register mouse events
addMouseListener(this);
}
// Cycle through panels.
public void mousePressed(MouseEvent me) {
cardLO.next(osCards);
}
// Provide empty implementations for the other MouseListener methods.
public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) {
}
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}
public void actionPerformed(ActionEvent ae) {
if(ae.getSource() == Win) {
cardLO.show(osCards, “Windows”);
}
else {
cardLO.show(osCards, “Other”);
        }
    }
}
```

Output



(c) Amity University Online

Notes



Check your Understanding

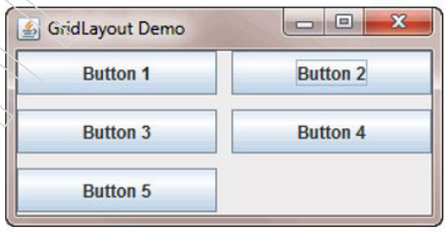
1. Java AWT (Abstract Window Toolkit) is an API to expand _____ .
2. Java AWT components are _____ i.e. components are showed according to the analysis of OS.
3. The AWT classes are included in the _____ .
4. Since Frame is a subclass of Component, it contains all the characteristics declared by _____ .
5. FlowLayout is the _____ layout.

Summary

- The AWT describes windows according to a class hierarchy that adds usability and specificity with each segments. The two most widespread windows are those which are carryout from Panel, which is utilized by applets, and those which were taken out from Frame, which construct a standard application window.
- Today, most Java language programs utilize user interfaces based on Swing. Because Swing provides better-off implementations than does the AWT of some general GUI controls, such as buttons, lists, and check boxes, it is easy to jump to the end that the AWT is no longer significant, that it has been outdated by Swing.
- A layout manager is a request of any class that executes the LayoutManager interface. The layout manager is declared by the setLayout( ) method. If no invoke to setLayout( ) is made, then the default layout manager is utilized automatically.

Activity

1. Create a code to using grid layout replicate below image



Questions and exercises

1. What is Graphic class?
2. What are different methods in Graphic class?
3. How to use loops in applet?

(c) Amity University Online

Notes

Glossary

- AWT: Abstract Window Toolkit
- GUI: Graphical User Interface
- OS: Operating System
- Component: In starting of the Abstract Window Toolkit hierarchy is the Component class. Component is a conceptual class that contains all of the characteristics of a visual component.
- Container: The Container class is a child class for Component. It has multiple methods that permits other Component objects to be encapsulate in it.
- Panel: The Panel class is an actual subclass of Container. It doesn't put in any fresh methods; it simply execute Container.
- Window: The Window class constructs a top-level window. A top-level window is not included within any other object; it sits in a straight line on the monitor.
- BorderLayout: The BorderLayout class executes a common layout style for top-level windows.
- FlowLayout: FlowLayout is the default layout. FlowLayout executes an easy layout style, which is like to how words flow in a text editor.
- GridLayout: GridLayout puts out components in a 2-D grid.

Further Readings:

1. Java AWT reference, John Zukowski
2. Learn Java GUI Applications 11th Edition: A JFC Swing Tutorial, Philip Conrod, Lou Tylee
3. Graphic Java 1.2: Mastering the JFC, David M. Geary Java Applet Powerpack, Tad Ringo, Robin Drake
4. Java 2D graphics, Jonathan Knudsen
5. Java AWT reference, John Zukowski

Check Your Understanding:

Answers

1. Graphical User Interface
2. platform-dependent
3. java.awt package
4. Component
5. Default

(c) Amity University Online