

LECTURE NOTES
ON
OBJECT ORIENTED PROGRAMMING (1805PC05)

II YEAR II SEMESTER

PREPARED BY

A.RADHA RANI
Assoc. Professor

A DAMODAR
Asst. Professor

M RAGHUPATHI
Asst. Professor



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLAREDDY ENGINEERING COLLEGE FOR WOMEN

(Autonomous Institution-UGC, Govt. of India)

Accredited by NBA & NAAC with 'A' Grade, UGC, Govt. of India NIRF Indian Ranking–2018,
Accepted by MHRD, Govt. of India, Permanently Affiliated to JNTUH, Approved by AICTE,
ISO 9001:2015 Certified Institution AAAA+ Rated by Digital Learning Magazine, AAA+ Rated
by Careers 360 Magazine, 3rd Rank CSR, Platinum Rated by AICTE-CII Survey, Top 100 Rank
band by ARIIA, MHRD, Govt. of India National Ranking-Top 100 Rank band by Outlook,

National Ranking-Top 100 Rank band by Times News Magazine
Maisammaguda, Dhullapally, Secunderabad, Kompally-500100

2020-2021

MALLA REDDY ENGINEERING COLLEGE FOR WOMEN

(1805PC05) OBJECT ORIENTED PROGRAMMING

B. Tech II Year II Sem

L T P C

3 0 0 3

Course Objective:

- The objective of this course is to provide object oriented concepts through which robust, secured and reusable software can be developed.
- To understand object oriented principles like abstraction, encapsulation, inheritance and polymorphism and apply them in solving problems.
- To understand the principles of inheritance and polymorphism and demonstrate how they relate to the design of abstract classes.
- To understand the implementation of packages and interfaces.
- To understand the concepts of exception handling, multithreading and collection classes.
- To understand the design of Graphical User Interface using applets and swing controls.

Course Outcomes:

At the end of the course the students are able to:

- An understanding of the principles and practice of object oriented analysis and design in the construction of robust, maintainable programs which satisfy their requirements;
- A competence to design, write, compile, test and execute straight forward programs using a high level language;
- An appreciation of the principles of object oriented programming;
- An awareness of the need for a professional approach to design and the importance of good documentation to the finished programs.
- Be able to implement, compile, test and run Java programs comprising more than one class, to address a particular software problem.
- Demonstrate the ability to use simple data structures like arrays in a Java program.
- Be able to make use of members of classes found in the Java API.
- Demonstrate the ability to employ various types of selection constructs in a Java program. Be able to employ a hierarchy of Java classes to provide a solution to a given set of requirements.
- Able to develop applications using Applet and Swings.

UNIT-I

Object-oriented thinking- A way of viewing world – Agents and Communities, messages and methods, Responsibilities, Classes and Instances, Class Hierarchies- Inheritance, Method binding, Overriding and Exceptions, Summary of Object-Oriented concepts.

An Overview of Java -History of Java, comments, Data types, Variables, Constants, Scope and Lifetime of variables, Operators, Type conversion and casting, Enumeration, Control flow- block scope, conditional statements, loops, break and continue statements, simple java stand alone programs, arrays, console input and output, classes, methods, constructors, static, this keyword, recursion, exploring string classes and garbage collection.

UNIT – II

Inheritance –Inheritance hierarchy, super keyword, preventing inheritance: final classes and methods, the Object class and its methods. **Polymorphism** – dynamic binding, Constructor and method overloading, method overriding, abstract classes.

Interfaces-Interfaces Vs Abstract classes, defining an interface, implementing interfaces, accessing implementations through interface references, extending interface, inner class.

Packages- Defining, creating and accessing a package, CLASSPATH, Access modifiers, importing packages.

UNIT - III

Exception handling - Fundamentals of exception handling, Exception types, Termination or presumptive models, Uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws and finally, built- in exceptions, creating own exception sub classes.

Multithreading- Differences between thread-based multitasking and process-based multitasking, Java thread model, creating threads, thread priorities, synchronizing threads, inter thread communication.

UNIT- IV

Collection Framework in Java – Introduction to java collections, Overview of java collection framework, commonly used collection classes-ArrayList, LinkedList, HashSet, TreeSet, Map-HashMap, TreeMap, Legacy Classes-Vector, Stack, Hashtable.

Other Utilities-Scanner, String Tokenizer, Random, Date. **Files-Streams**-Byte streams, Character streams, Text input/output, Binary input/output, File management using File class.

UNIT-V

Applets – Inheritance hierarchy for applets, differences between applets and applications, Life cycle of an applet and Passing parameters to applets.

GUI Programming - Swing - The AWT class hierarchy, Introduction to Swing, Swing Vs AWT, Hierarchy for Swing components, Overview of Swing components –JButton, JLabel, JTextField, JCheckBox, RadioButton, JTextArea, etc simple Swing applications, Layout managers– FlowLayout, BorderLayout, GridLayout and GridbagLayout.

Event Handling-Events, Event sources, Event classes, Event Listeners, Delegation event model, Handling Mouse and Key events, Adapter classes.

TEXT BOOKS:

1. Java The complete reference, 9th edition, Herbert Schildt, McGraw Hill Education (India) Pvt. Ltd.
2. Understanding Object-Oriented Programming with Java, updated edition, T. Budd, Pearson Education.

REFERENCE BOOKS:

1. An Introduction to programming and OO design using Java, J. Nino and F.A. Hosch, John Wiley & sons.
2. Introduction to Java programming, Y. Daniel Liang, Pearson Education.
3. Object Oriented Programming through Java, P. Radha Krishna, Universities Press.
4. Programming in Java, S. Malhotra, S. Chudhary, 2nd edition, Oxford Univ. Press.
5. Java Programming and Object Oriented Application Development, R. A. Johnson, Cengage Learning.

II B.Tech II Semester

1805PC05: OBJECT ORIENTED PROGRAMMING

Course Objectives:

- To introduce the object oriented programming concepts.
- To understand object oriented programming concepts, and apply them in solving problems.
- To introduce the principles of inheritance and polymorphism; and demonstrate how they relate to the design of abstract classes.
- To introduce the implementation of packages and interfaces
- To introduce the concepts of exception handling and multithreading.
- To introduce the design of Graphical User Interface using applets and swing controls.

Course Outcomes:

- Able to solve real world problems using OOP techniques.
- Able to understand the use of abstract classes.
- Able to solve problems using java collection framework and I/O classes.
- Able to develop multithreaded applications with synchronization.
- Able to develop applets for web applications.
- Able to design GUI based applications

INDEX

Sl. No.	TOPIC NAME	PAGE NO.
1	OBJECT-ORIENTED THINKING	1
1.1	A way of viewing world	1
1.2	Object-Oriented Concepts	2
1.3	Java Buzzwords	4
1.4	A Overview of Java	6
1.5	Data Types	7
1.6	Variables	8
1.7	Arrays	12
1.8	Operators	15
1.9	Expressions	18
1.10	Control statements	19
1.11	Classes and Methods	30
1.12	String handling	32
1.13	Inheritance Concepts	35
1.14	Constructors	37
1.15	Polymorphism	44
1.16	Method Overriding	46
1.17	Abstract Classes	47
1.18	Object Class	50
1.19	Forms of Inheritance	50
1.20	Benefits and Cost Of Inheritance	51

2	PACKAGES AND INTERFACES	53
2.1	Defining Packages	53
2.2	CLASSPATH	54
2.3	Access Protection	56
2.4	Importing Packages	58
2.5	Defining An Interface	60
2.6	Implementing Interface	61
2.7	Nested Interfaces	64
2.8	Variables in Interfaces And Extending Interfaces	66
2.9	Stream classes	68
2.10	Reading and Writing Console	76
2.11	File class	80
2.12	RandomAccessFile	85
2.13	Console class	89
2.14	Serialization	91
2.15	Enumerations	94
2.16	Auto Boxing	97
2.17	Generics	98
3	EXCEPTION HANDLING AND MULTITHREADING	101
3.1	Fundamentals of Exception Handling	101
3.2	Exception Types	102
3.3	Termination or Resumptive models	105
3.4	Uncaught Exceptions	105
3.5	Using try and catch	106
3.6	Multiple catch clauses	108

3.7	Nested try statements	109
3.8	Throw, Throws and finally	110
3.9	Built-in exceptions	114
3.10	Creating own exception sub classes	116
3.11	Differences between thread based multitasking and process based multitasking	118
3.12	Creating threads	120
3.13	Thread priorities	123
3.14	Synchronizing threads	125
3.15	Inter thread communication	129
4	THE COLLECTIONS FRAMEWORK	134
4.1	Collections overview	134
4.2	Collection Interfaces	135
4.3	Collection Classes	142
4.4	Accessing a Java Collection via a Iterator	170
4.5	Map Interfaces and Classes	173
4.6	Comparators	178
4.7	Collection algorithms	185
4.8	The Legacy Classes and Interfaces	187
5	GUI PROGRAMMING WITH JAVA	216
5.1	Introduction	216
5.2	Limitations of AWT	217
5.3	MVC architecture	217
5.4	Components	218
5.5	Containers	219

5.6	Understanding layout managers	226
5.6.1	Flow layout	227
5.6.2	Border layout	228
5.6.3	Grid layout	229
5.6.4	Card layout	231
5.6.5	Grid Bag out layout	232
5.7	The delegation Event handling	234
5.8	Event sources	235
5.9	Event Listener	235
5.10	Event classes	236
5.11	Handling mouse and keyboard events	238
5.12	Adapter classes	240
5.13	Inner classes	241
5.14	Anonymous inner classes	242
5.15	A simple swing Application	244
5.16	Applets	246
5.17	Security issues	249
5.18	Applets and Applications	250
5.19	Passing parameters to applets	251
5.20	Creating a swing Applet	252
5.21	Painting in Swing	253
5.22	A paint example	255
5.23	Exploring Swing controls	256
5.23.1	JLable and Image Icon	258
5.23.2	JText Field	260

5.23.3	The swing Buttons	262
5.23.4	JButtons	262
5.23.5	JToggle Button	265
5.23.6	JCheck Box	268
5.23.7	JRadio Button	270
5.23.8	JTabbed Button	273
5.23.9	JScrollPane	276
5.23.10	JList	278
5.23.11	JComboBox	282
5.24	Swing Menus	285
5.25	Dialogs	287
	TUTORIAL QUESTIONS	291
	ASSIGNMENT QUESTIONS	297
	UNIT WISE IMPORTANT QUESTION QUESTIONS	300
	OBJECTIVE QUESTIONS	307
	INTERNAL QUESTION PAPERS	324
	PREVIOUS QUESTION PAPERS	327

UNIT - I

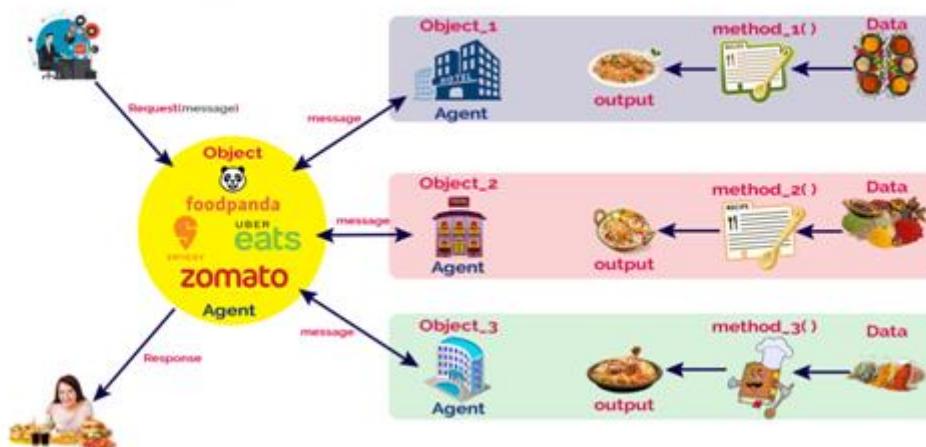
OBJECT-ORIENTED THINKING

1.1 A way of viewing world:

A way of viewing the world is an idea to illustrate the object-oriented programming concept with an example of a real-world situation. Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?

To solve the problem, let me call zomato (an **agent** in food delivery community), tell them the variety and quantity of food and the hotel name from which I wish to deliver the food to my family members. Look at the following image.

A way of viewing world with OOP



Agents and Communities

To solve my food delivery problem, I used a solution by finding an appropriate agent (Zomato) and pass a message containing my request. It is the responsibility of the agent (Zomato) to satisfy my request. Here, the agent uses some method to do this. I do not need to know the method that the agent has used to solve my request. This is usually hidden from me. So, in object-oriented programming, problem-solving is the solution to our problem which requires the help of many individuals in the community. We may describe agents and communities as follows.

An object-oriented program is structured as a community of interacting agents, called objects. Where each object provides a service (data and methods) that is used by other members of the community. In our example, the online food delivery system is a community in which the agents are zomato and set of hotels. Each hotel provides a variety of services that can be used by other members like zomato, myself, and my family in the community.

Messages and Methods

To solve my problem, I started with a request to the agent zomato, which led to still more requests among the members of the community until my request has done. Here, the members of a community interact with one another by making requests until the problem has satisfied.

In object-oriented programming, every action is initiated by passing a message to an agent (object), which is responsible for the action. The receiver is the object to whom the message was sent. In response to the message, the receiver performs some method to carry out the request. Every message may include any additional information as arguments. In our example, I send a request to zomato with a message that contains food items, the quantity of food, and the hotel details. The receiver uses a method to food get delivered to my home.

Responsibilities

In object-oriented programming, behaviors of an object described in terms of responsibilities. In our example, my request for action indicates only the desired outcome (food delivered to my family). The agent (zomato) free to use any technique that solves my problem. By discussing a problem in terms of responsibilities increases the level of abstraction. This enables more independence between the objects in solving complex problems.

Classes and Instances

In object-oriented programming, all objects are instances of a class. The method invoked by an object in response to a message is decided by the class. All the objects of a class use the same method in response to a similar message. In our example, the zomato a class and all the hotels are sub-classes of it. For every request (message), the class creates an instance of it and uses a suitable method to solve the problem.

Classes Hierarchies

A graphical representation is often used to illustrate the relationships among the classes (objects) of a community. This graphical representation shows classes listed in a hierarchical tree-like structure. In this more abstract class listed near the top of the tree, and more specific classes in the middle of the tree, and the individuals listed near the bottom. In object-oriented programming, classes can be organized into a hierarchical inheritance structure. A child class inherits properties from the parent class that higher in the tree.

Method Binding, Overriding, and Exception

In the class hierarchy, both parent and child classes may have the same method which implemented individually. Here, the implementation of the parent is overridden by the child. Or a class may provide multiple definitions to a single method to work with different arguments (overloading).

The search for the method to invoke in response to a request (message) begins with the class of this receiver. If no suitable method is found, the search is performed in the parent class of it. The search continues up the parent class chain until either a suitable method is found or the parent class chain is exhausted. If a suitable method is found, the method is executed. Otherwise, an error message is issued.

1.2 Object-Oriented Concepts

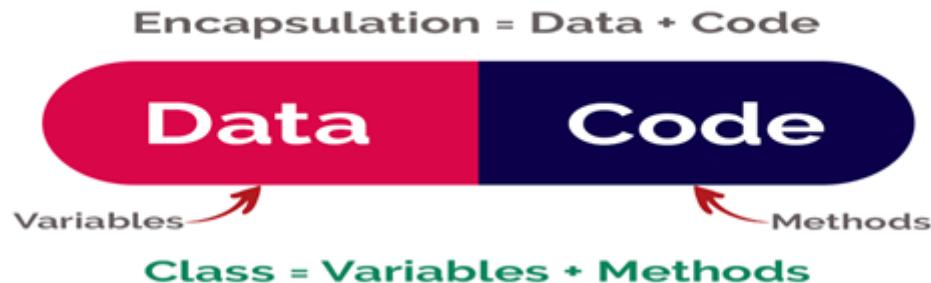
OOP stands for Object-Oriented Programming. OOP is a programming paradigm in which every program is follows the concept of object. In other words, OOP is a way of writing programs based on the object concept.

The object-oriented programming paradigm has the following core concepts.

- **Encapsulation**
- **Inheritance**
- **Polymorphism**
- **Abstraction**

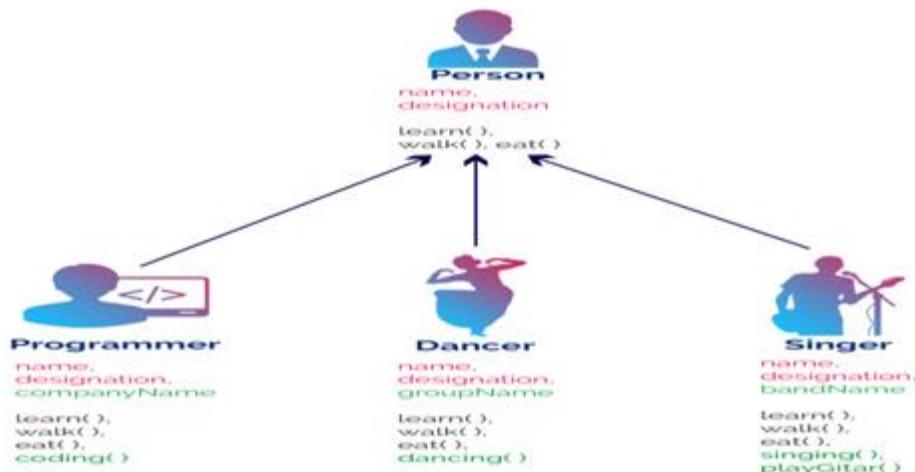
The popular object-oriented programming languages are Smalltalk, C++, Java, PHP, C#, Python, etc.

Encapsulation



Encapsulation is the process of combining data and code into a single unit (object / class). In OOP, every object is associated with its data and code. In programming, data is defined as variables and code is defined as methods. The java programming language uses the class concept to implement encapsulation.

Inheritance



Inheritance is the process of acquiring properties and behaviors from one object to another object or one class to another class. In inheritance, we derive a new class from the existing class. Here, the new class acquires the properties and behaviors from the existing class. In the inheritance concept, the class which provides properties is called as parent class and the class which receives the properties is called as child class. The parent class is also known as base class or super class. The child class is also known as derived class or sub class. In the inheritance, the properties and behaviors of base class extended to its derived class, but the base class never receive properties or behaviors from its derived class.

In java programming language the keyword extends is used to implement inheritance.

Polymorphism



Polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

The java uses method overloading and method overriding to implement polymorphism.

Method overloading - multiple methods with same name but different parameters.

Method overriding - multiple methods with same name and same parameters.

Abstraction



Abstraction is hiding the internal details and showing only essential functionality. In the abstraction concept, we do not show the actual implementation to the end user, instead we provide only essential things. For example, if we want to drive a car, we do not need to know about the internal functionality like how wheel system works? how brake system works? how music system works? etc.

1.3 Java Buzzwords:

Java is the most popular object-oriented programming language. Java has many advanced features, a list of key features is known as Java Buzz Words. The java team has listed the following terms as java buzz words.

- Simple
- Secure

- Portable
- Object-oriented
- Robust
- Architecture-neutral (or) Platform Independent
- Multi-threaded
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java programming language is very simple and easy to learn, understand, and code. Most of the syntaxes in java follow basic programming language C and object-oriented programming concepts are similar to C++. In a java programming language, many complicated features like pointers, operator overloading, structures, unions, etc. have been removed. One of the most useful features is the garbage collector it makes java more simple.

Secure

Java is said to be more secure programming language because it does not have pointers concept, java provides a feature "applet" which can be embedded into a web application. The applet in java does not allow access to other parts of the computer, which keeps away from harmful programs like viruses and unauthorized access.

Portable

Portability is one of the core features of java which enables the java programs to run on any computer or operating system. For example, an applet developed using java runs on a wide variety of CPUs, operating systems, and browsers connected to the Internet.

Object-oriented

Java is said to be a pure object-oriented programming language. In java, everything is an object. It supports all the features of the object-oriented programming paradigm. The primitive data types java also implemented as objects using wrapper classes, but still, it allows primitive data types to archive high-performance.

Robust

Java is more robust because the java code can be executed on a variety of environments, java has a strong memory management mechanism (garbage collector), java is a strictly typed language, it has a strong set of exception handling mechanism, and many more.

Architecture-neutral (or) Platform Independent

Java has invented to archive "write once; run anywhere, any time, forever". The java provides JVM (Java Virtual Machine) to to archive architectural-neutral or platform-independent. The JVM allows the java program created using one operating system can be executed on any other operating system.

Multi-threaded

Java supports multi-threading programming, which allows us to write programs that do multiple operations simultaneously.

Interpreted

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. The byte code is interpreted to any machine code so that it runs on the native machine.

High performance

Java provides high performance with the help of features like JVM, interpretation, and its simplicity.

Distributed

Java programming language supports TCP/IP protocols which enable the java to support the distributed environment of the Internet. Java also supports Remote Method Invocation (RMI), this feature enables a program to invoke methods across a network.

Dynamic

Java is said to be dynamic because the java byte code may be dynamically updated on a running system and it has a dynamic memory allocation and deallocation (objects and garbage collector).

1.4 A Overview of Java:

Java is a computer programming language. Java was created based on C and C++. Java uses C syntax and many of the object-oriented features are taken from C++. Before Java was invented there were other languages like COBOL, FORTRAN, C, C++, Small Talk, etc. These languages had few disadvantages which were corrected in Java. Java also innovated many new features to solve the fundamental problems which the previous languages could not solve. Java was invented by a team of 13 employees of Sun Microsystems, Inc. which is lead by James Gosling, in 1991. The team includes persons like Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan, etc., Java was developed as a part of the Green project. Initially, it was called Oak, later it was changed to Java in 1995.

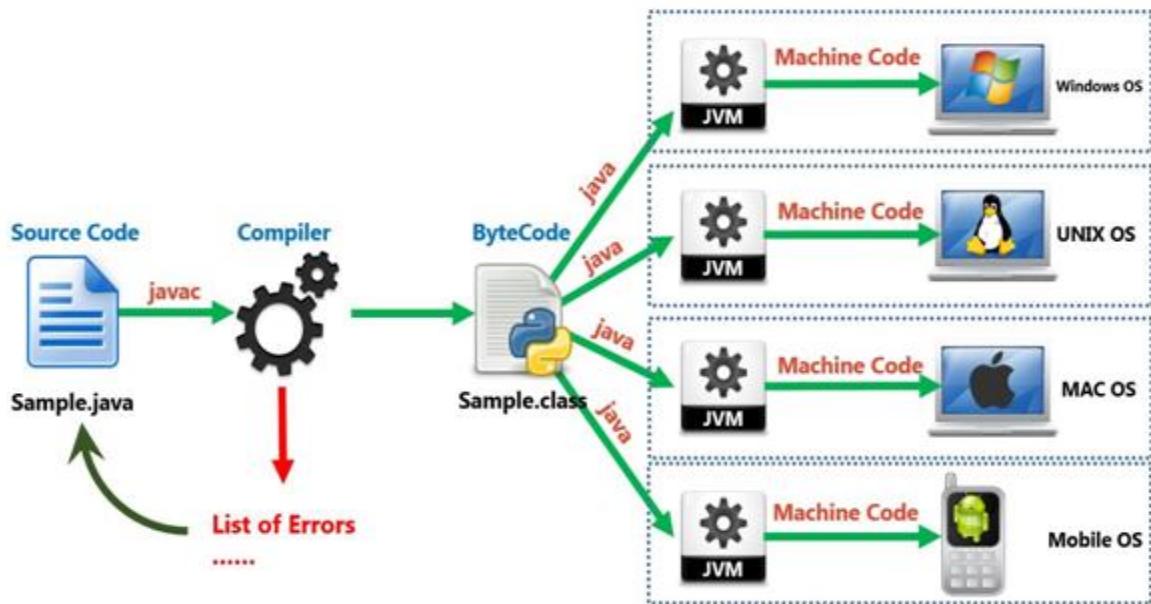
History of Java

- The C language developed in 1972 by Dennis Ritchie had taken a decade to become the most popular language.
- In 1979, Bjarne Stroustrup developed C++, an enhancement to the C language with included OOP fundamentals and features.
- A project named “Green” was initiated in December of 1990, whose aim was to create a programming tool that could render obsolete the C and C++ programming languages.
- Finally in the year of 1991 the Green Team was created a new Programming language named “OAK”.
- After some time they found that there is already a programming language with the name “OAK”.
- So, the green team had a meeting to choose a new name. After so many discussions they want to have a coffee. They went to a Coffee Shop which is just outside of the Gosling’s office and there they have decided name as “JAVA”.

Execution Process of Java Program

The following three steps are used to create and execute a java program.

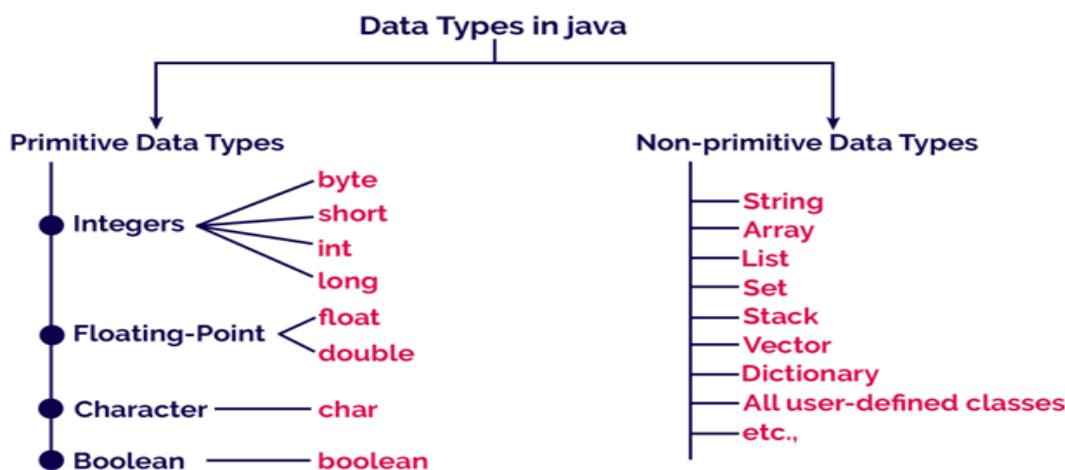
- Create a source code (java file).
- Compile the source code using javac command.
- Run or execute .class file using java command.



1.5 Data Types :

Java programming language has a rich set of data types. The data type is a category of data stored in variables. In java, data types are classified into two types and they are as follows.

- **Primitive Data Types**
- **Non-primitive Data Types**



Primitive Data Types

The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size. The primitive data types do not have any additional methods.

In java, primitive data types includes **byte**, **short**, **int**, **long**, **float**, **double**, **char**, and **boolean**.

Data type	Meaning	Memory size	Range	Default Value
byte	Whole numbers	1 byte	-128 to +127	0
short	Whole numbers	2 bytes	-32768 to +32767	0
int	Whole numbers	4 bytes	-2,147,483,648 to +2,147,483,647	0
long	Whole numbers	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0L
float	Fractional numbers	4 bytes	-	0.0f
double	Fractional numbers	8 bytes	-	0.0d
char	Single character	2 bytes	0 to 65535	\u0000
boolean	unsigned char	1 bit	0 or 1	0 (false)

Non-primitive Data Types

In java, non-primitive data types are the reference data types or user-created data types. All non-primitive data types are implemented using object concepts. Every variable of the non-primitive data type is an object. The non-primitive data types may use additional methods to perform certain operations. The default value of non-primitive data type variable is null.

In java, examples of non-primitive data types are **String**, **Array**, **List**, **Queue**, **Stack**, **Class**, **Interface**, etc.

1.6 Variables:

A variable is a named memory location used to store a data value. A variable can be defined as a container that holds a data value.

In java programming language variables are classified as follows.

- Local variables
- Instance variables or Member variables or Global variables
- Static variables or Class variables
- Final variables

Local variables

The variables declared inside a method or a block are known as local variables. A local variable is visible within the method in which it is declared. The local variable is created when execution control enters into the method or block and destroyed after the method or block execution completed.

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Let's look at the following example java program to illustrate local variable in java.

```
public class Test {  
    public void pupAge() {  
        int age = 0;  
        age = age + 7;  
        System.out.println("Puppy age is : " +age);  
    }  
  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

Instance variables or member variables or global variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.

- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

Let's look at the following example java program to illustrate instance variable in java

```
import java.io.*;
public class Employee {

    // this instance variable is visible for any child class.
    public String name;
    // salary variable is visible in Employee class only.
    private double salary;
    // The name variable is assigned in the constructor.
    public Employee (String empName) {
        name = empName;
    }
    // The salary variable is assigned a value.
    public void setSalary(double empSal) {
        salary = empSal;
    }
    // This method prints the employee details.
    public void printEmp() {
        System.out.println("name : " + name );
        System.out.println("salary :" + salary);
    }
    public static void main(String args[]) {
        Employee empOne = new Employee("Ransika");
        empOne.setSalary(1000);
        empOne.printEmp();
    }
}
```

Static variables or Class variables

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.

- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name `ClassName.VariableName`.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

Example

```
import java.io.*;  
  
public class Employee {  
  
    // salary variable is a private static variable  
  
    private static double salary;  
  
    // DEPARTMENT is a constant  
  
    public static final String DEPARTMENT = "Development ";  
  
    public static void main(String args[]) {  
  
        salary = 1000;  
  
        System.out.println(DEPARTMENT + "average salary:" + salary);  
  
    }  
  
}
```

Final variables

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to a different object. However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the *final modifier* often is used with static to make the constant a class variable.

Example

```
public class Test {  
    final int value = 10;  
  
    // The following are examples of declaring constants:  
  
    public static final int BOXWIDTH = 6;  
  
    static final String TITLE = "Manager";  
  
    public void changeValue() {  
  
        value = 12; // will give an error  
  
    }  
}
```

1.7 Arrays:

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

Syntax

```
dataType[] arrayRefVar; // preferred way.  
or  
dataType arrayRefVar[]; // works but not preferred way.
```

Note – The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax –

```
double[] myList; // preferred way.
```

or

```
double myList[]; // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax –

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things –

It creates an array using new dataType[arraySize].

It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below –

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows –

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

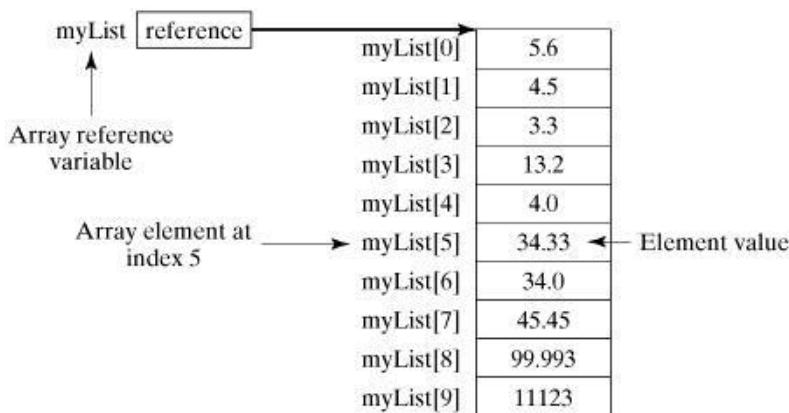
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList –

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays –

```
public class TestArray
```

```
{
```

```
    public static void main(String[] args) {  
  
        double[] myList = {1.9, 2.9, 3.4, 3.5};  
  
        // Print all the array elements  
  
        for (int i = 0; i < myList.length; i++) {  
  
            System.out.println(myList[i] + " ");  
  
        }  
  
        // Summing all elements  
  
        double total = 0;  
  
        for (int i = 0; i < myList.length; i++) {  
  
            total += myList[i];  
  
        }  
  
        System.out.println("Total is " + total);  
  
        // Finding the largest element  
  
        double max = myList[0];  
  
        for (int i = 1; i < myList.length; i++) {  
  
            if (myList[i] > max) max = myList[i];  
  
        }  
  
        System.out.println("Max is " + max);  
  
    }  
}
```

The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example

The following code displays all the elements in the array myList –

```
public class TestArray {
    public static void main(String[] args) {
        double[] myList = {1.9, 2.9, 3.4, 3.5};
        // Print all the array elements
        for (double element: myList) {
            System.out.println(element);
        }
    }
}
```

1.8 Operators:

An operator is a symbol used to perform arithmetic and logical operations. Java provides a rich set of operators.

In java, operators are classified into the following four types.

- Arithmetic Operators
- Relational (or) Comparison Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Conditional Operators

Let's look at each operator in detail.

Arithmetic Operators

In java, arithmetic operators are used to performing basic mathematical operations like addition, subtraction, multiplication, division, modulus, increment, decrement, etc.,

Operator	Meaning	Example
+	Addition	$10 + 5 = 15$
-	Subtraction	$10 - 5 = 5$
*	Multiplication	$10 * 5 = 50$
/	Division	$10 / 5 = 2$
%	Modulus - Remainder of the Division	$5 \% 2 = 1$
++	Increment	$a++$
--	Decrement	$a--$

Relational Operators ($<$, $>$, $<=$, $>=$, $==$, $!=$)

The relational operators are the symbols that are used to compare two values. That means the relational operators are used to check the relationship between two values. Every relational operator has two possible results either **TRUE** or **FALSE**. In simple words, the relational operators are used to define conditions in a program. The following table provides information about relational operators.

Operator	Meaning	Example
$<$	Returns TRUE if the first value is smaller than second value otherwise returns FALSE	$10 < 5$ is FALSE
$>$	Returns TRUE if the first value is larger than second value otherwise returns FALSE	$10 > 5$ is TRUE
$<=$	Returns TRUE if the first value is smaller than or equal to second value otherwise returns FALSE	$10 <= 5$ is FALSE
$>=$	Returns TRUE if the first value is larger than or equal to second value otherwise returns FALSE	$10 >= 5$ is TRUE
$==$	Returns TRUE if both values are equal otherwise returns FALSE	$10 == 5$ is FALSE
$!=$	Returns TRUE if both values are not equal otherwise returns FALSE	$10 != 5$ is TRUE

Logical Operators

The logical operators are the symbols that are used to combine multiple conditions into one condition. The following table provides information about logical operators.

Operator	Meaning	Example
$&$	Logical AND - Returns TRUE if all conditions are TRUE otherwise returns FALSE	$false \& true \Rightarrow false$
$ $	Logical OR - Returns FALSE if all conditions are FALSE otherwise returns TRUE	$false true \Rightarrow true$
$^$	Logical XOR - Returns FALSE if all conditions are same otherwise returns TRUE	$true ^ true \Rightarrow false$
$!$	Logical NOT - Returns TRUE if condition is FALSE and returns FALSE if it is TRUE	$!false \Rightarrow true$
$&&$	short-circuit AND - Similar to Logical AND ($\&$), but once a decision is finalized it does not evaluate remaining.	$false \& true \Rightarrow false$
$ $	short-circuit OR - Similar to Logical OR ($ $), but once a decision is finalized it does not evaluate remaining.	$false true \Rightarrow true$

Assignment Operators

The assignment operators are used to assign right-hand side value (Rvalue) to the left-hand side variable (Lvalue). The assignment operator is used in different variants along with arithmetic operators. The following table describes all the assignment operators in the java programming language.

Operator	Meaning	Example
=	Assign the right-hand side value to left-hand side variable	$A = 15$
+=	Add both left and right-hand side values and store the result into left-hand side variable	$A += 10$
-=	Subtract right-hand side value from left-hand side variable value and store the result into left-hand side variable	$A -= B$
*=	Multiply right-hand side value with left-hand side variable value and store the result into left-hand side variable	$A *= B$
/=	Divide left-hand side variable value with right-hand side variable value and store the result into the left-hand side variable	$A /= B$
%=	Divide left-hand side variable value with right-hand side variable value and store the remainder into the left-hand side variable	$A %= B$
&=	Logical AND assignment	-
=	Logical OR assignment	-
^=	Logical XOR assignment	-

Bitwise Operators

The bitwise operators are used to perform bit-level operations in the java programming language. When we use the bitwise operators, the operations are performed based on binary values. The following table describes all the bitwise operators in the java programming language.

Let us consider two variables A and B as $A = 25$ (11001) and $B = 20$ (10100).

Operator	Meaning	Example
&	the result of Bitwise AND is 1 if all the bits are 1 otherwise it is 0	$A \& B$
		$\Rightarrow 16$ (10000)
	the result of Bitwise OR is 0 if all the bits are 0 otherwise it is 1	$A B$
		$\Rightarrow 29$ (11101)
^	the result of Bitwise XOR is 0 if all the bits are same otherwise it is 1	$A ^ B$
		$\Rightarrow 13$ (01101)
~	the result of Bitwise once complement is negation of the bit (Flipping)	$\sim A$
		$\Rightarrow 6$ (00110)
<<	the Bitwise left shift operator shifts all the bits to the left by the specified number of positions	$A << 2$

Conditional Operators

The conditional operator is also called a **ternary operator** because it requires three operands. This operator is used for decision making. In this operator, first, we verify a condition, then we perform one operation out of the two operations based on the condition result. If the condition is TRUE the first option is performed, if the condition is FALSE the second option is performed. The conditional operator is used with the following syntax.

Syntax : Condition ? True part: False part;

1.9 Expressions:

In any programming language, if we want to perform any calculation or to frame any condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression. In the java programming language, an expression is defined as follows.

An expression is a collection of operators and operands that represents a specific value.

In the above definition, an **operator** is a symbol that performs tasks like arithmetic operations, logical operations, and conditional operations, etc. **Operands** are the values on which the operators perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

In the java programming language, expressions are divided into THREE types. They are as follows.

- Infix Expression
- Postfix Expression
- Prefix Expression

The above classification is based on the operator position in the expression.

Infix Expression

The expression in which the operator is used between operands is called infix expression. The infix expression has the following general structure.

Example



Postfix Expression

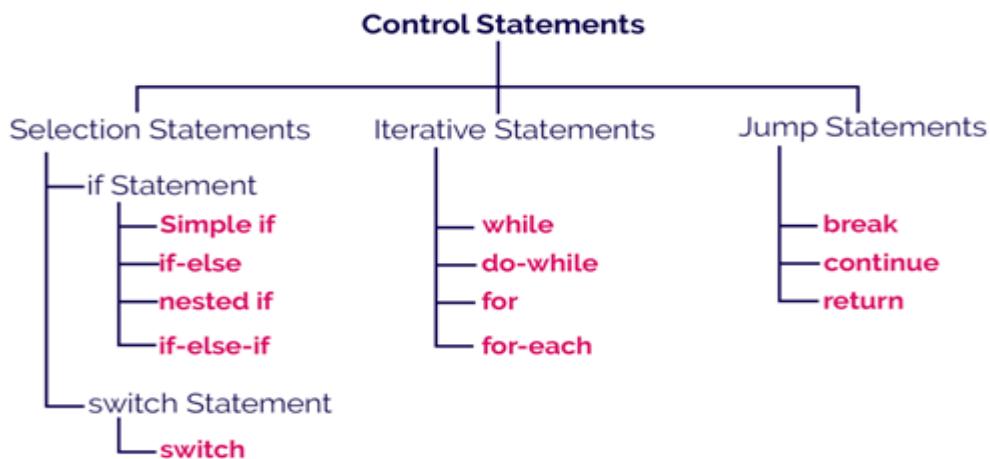
The expression in which the operator is used after operands is called postfix expression. The postfix expression has the following general structure.

Example**Prefix Expression**

The expression in which the operator is used before operands is called a prefix expression. The prefix expression has the following general structure.

Example**1.10 Control statements:**

In java, the default execution flow of a program is a sequential order. But the sequential order of execution flow may not be suitable for all situations. Sometimes, we may want to jump from line to another line, we may want to skip a part of the program, or sometimes we may want to execute a part of the program again and again. To solve this problem, java provides control statements.



In java, the control statements are the statements which will tell us that in which order the instructions are getting executed. The control statements are used to control the order of execution according to our requirements. Java provides several control statements, and they are classified as follows.

Types of Control Statements

In java, the control statements are classified as follows.

- Selection Control Statements (Decision Making Statements)
- Iterative Control Statements (Looping Statements)
- Jump Statements

Let's look at each type of control statements in java.

Selection Control Statements

In java, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be executed based on a condition. Java provides the following selection statements.

If Statement:

An **if** statement consists of a Boolean expression followed by one or more statements.

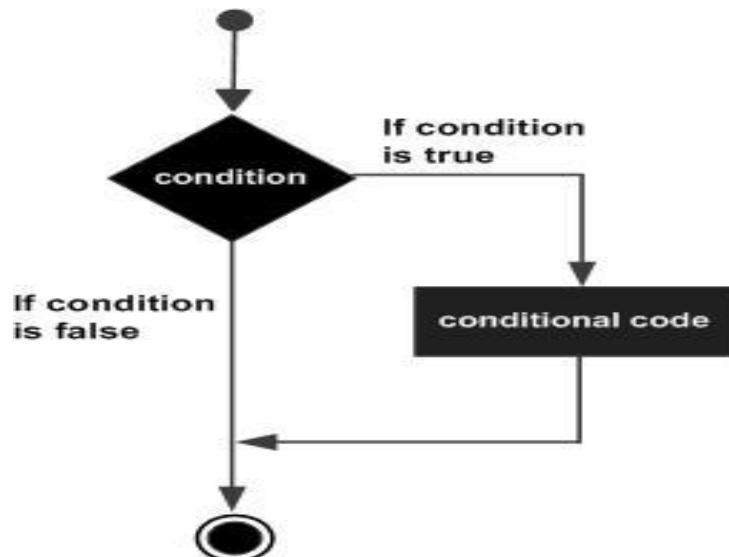
Syntax

Following is the syntax of an if statement –

```
if(Boolean_expression)
{
    // Statements will execute if the Boolean expression is true
}
```

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



```
public class Test
{
    public static void main(String args[])
    {
        int x = 10;
        if( x < 20 )
        {
            System.out.print("This is if statement");
        }
    }
}
```

If –else Statement :

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

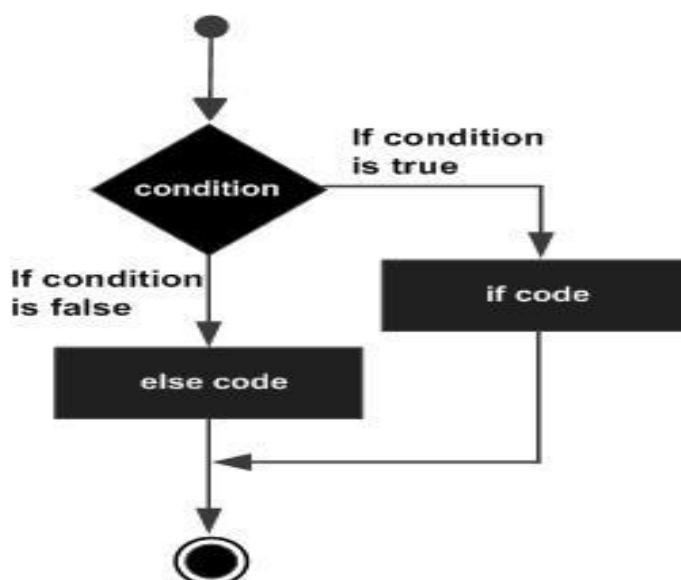
Syntax

Following is the syntax of an if...else statement –

```
if(Boolean_expression)
    // Executes when the Boolean expression is true
}else
    // Executes when the Boolean expression is false
}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
public class Test {  
    public static void main(String args[]) {  
        int x = 30;  
        if( x < 20 ) {  
            System.out.print("This is if statement");  
        }else {  
            System.out.print("This is else statement"); } } }
```

Nested If statement:

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax

The syntax for a nested if...else is as follows –

```
if(Boolean_expression 1) {  
    // Executes when the Boolean expression 1 is true  
    if(Boolean_expression 2) {  
        // Executes when the Boolean expression 2 is true  
    }  
}
```

You can nest **else if...else** in the similar way as we have nested *if* statement.

Example

```
public class Test {  
    public static void main(String args[]) {  
        int x = 30;  
        int y = 10;  
        if( x == 30 ) {  
            if( y == 10 ) {  
                System.out.print("X = 30 and Y = 10"); } } }
```

Switch Statement:

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

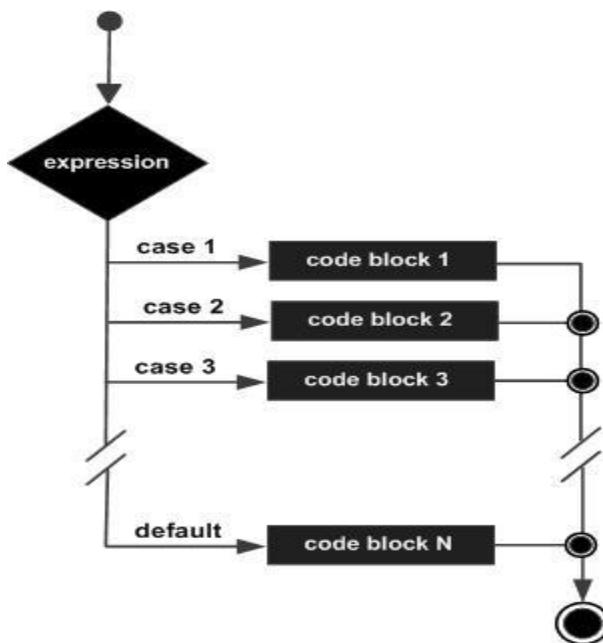
The syntax of enhanced for loop is –

```
switch(expression) {  
    case value :  
        // Statements  
        break; // optional  
  
    case value :  
        // Statements  
        break; // optional  
  
    // You can have any number of case statements.  
    default : // Optional  
        // Statements  
}
```

The following rules apply to a **switch** statement –

- The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a *break*. If no *break* appears, the flow of control will *fall through* to subsequent cases until a *break* is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No *break* is needed in the default case.

Flow Diagram



Example :

```
public class Test {  
    public static void main(String args[]) {  
        // char grade = args[0].charAt(0);  
        char grade = 'C';  
        switch(grade)  
        {  
            case 'A':  
                System.out.println("Excellent!");  
                break;  
            case 'B':  
            case 'C':  
                System.out.println("Well done");  
                break;  
            case 'D':  
                System.out.println("You passed");  
        }  
    }  
}
```

```
case 'F' :  
    System.out.println("Better try again");  
    break;  
  
default :  
    System.out.println("Invalid grade");}  
  
System.out.println("Your grade is " + grade);  
}  
}
```

Iterative Control Statements

In java, the iterative statements are also known as looping statements or repetitive statements. The iterative statements are used to execute a part of the program repeatedly as long as the given condition is True. Using iterative statements reduces the size of the code, reduces the code complexity, makes it more efficient, and increases the execution speed. Java provides the following iterative statements.

while statement:

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop is –

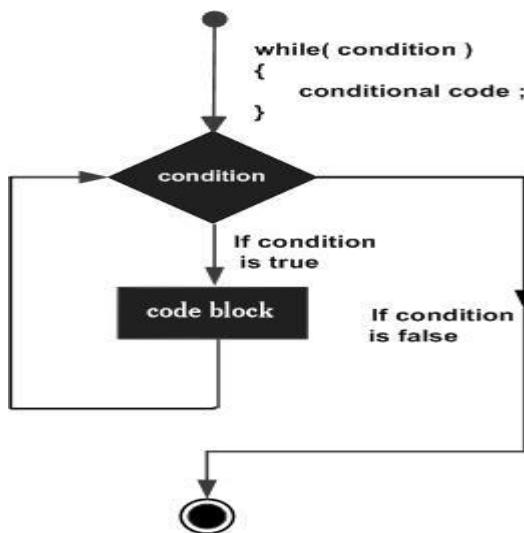
```
while(Boolean_expression)  
{  
    // Statements  
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```

public class Test {
    public static void main(String args[]) {
        int x = 10;
        while( x < 20 ) {
            System.out.print("value of x : " + x );
            x++;
            System.out.print("\n");
        } } }
  
```

do-while statement :

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

Following is the syntax of a do...while loop –

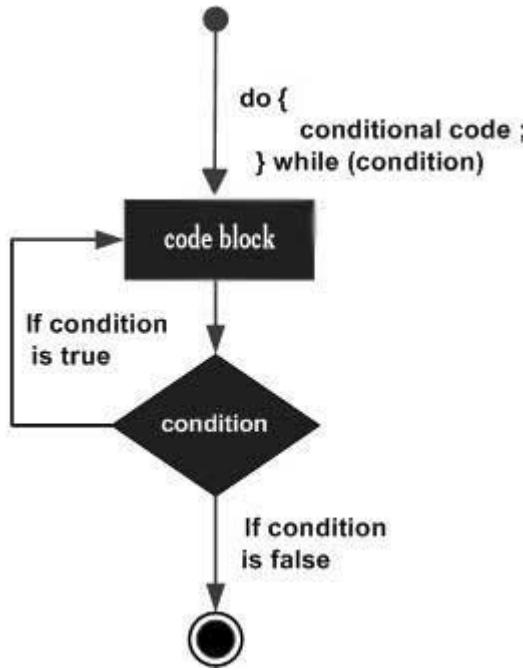
```

do {
    // Statements
}while(Boolean_expression);
  
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Flow Diagram



Example

```

public class Test
{
    public static void main(String args[])
    {
        int x = 10;
        do {
            System.out.print("value of x : " + x );
            x++;
            System.out.print("\n");
        }
        while( x < 20 );
    }
}
  
```

for statement:

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. A **for** loop is useful when you know how many times a task is to be repeated.

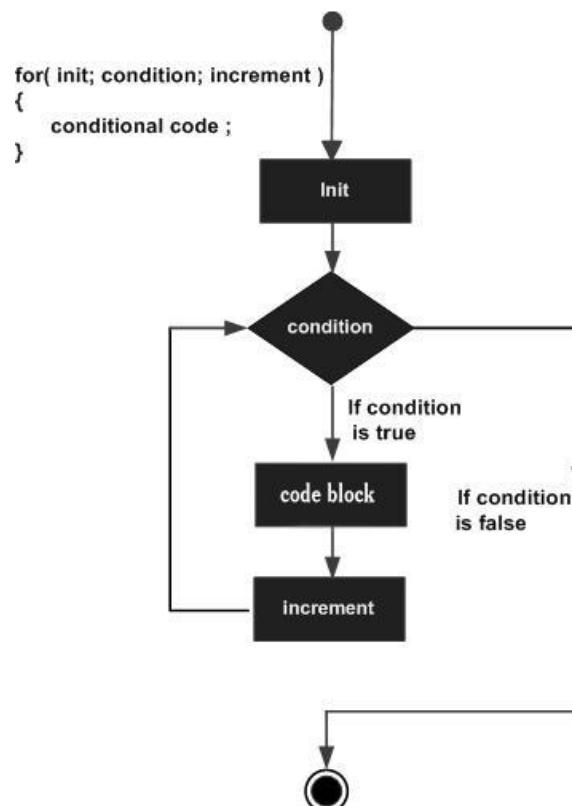
Syntax

The syntax of a for loop is –

```
for(initialization; Boolean_expression; update) {
    // Statements
}
```

Here is the flow of control in a **for** loop –

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram

Example

Following is an example code of the for loop in Java.

```
public class Test {  
  
    public static void main(String args[]) {  
  
        for(int x = 10; x < 20; x = x + 1) {  
  
            System.out.print("value of x : " + x );  
  
            System.out.print("\n"); } } }
```

Jump Statements

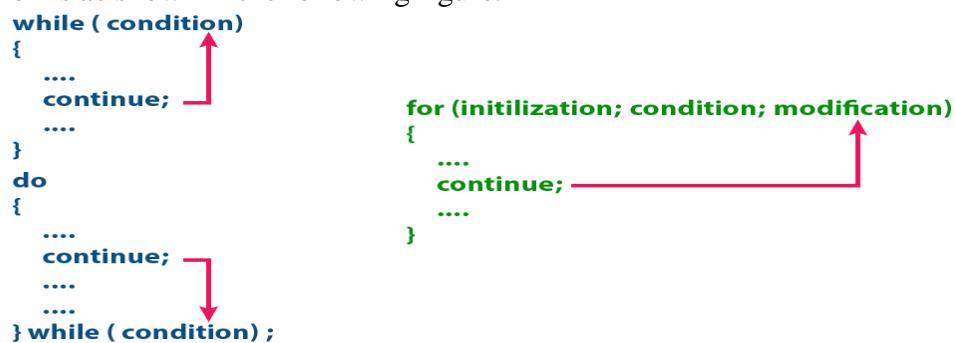
In java, the jump statements are used to terminate a block or take the execution control to the next iteration. Java provides the following jump statements.

- break

Continue:

The continue statement is used to move the execution control to the beginning of the looping statement. When the continue statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop. The continue statement can be used with looping statements like while, do-while, for, and for-each.

- When we use continue statement with while and do-while statements, the execution control directly jumps to the condition. When we use continue statement with for statement the execution control directly jumps to the modification portion (increment/decrement/any modification) of the for loop. The continue statement flow of execution is as shown in the following figure.

**Example**

```
public class JavaContinueStatement {  
  
    public static void main(String[] args) {  
  
        int list[] = { 10, 20, 30, 40, 50 };
```

```

for(int i : list) {

    if(i == 30)

        continue;

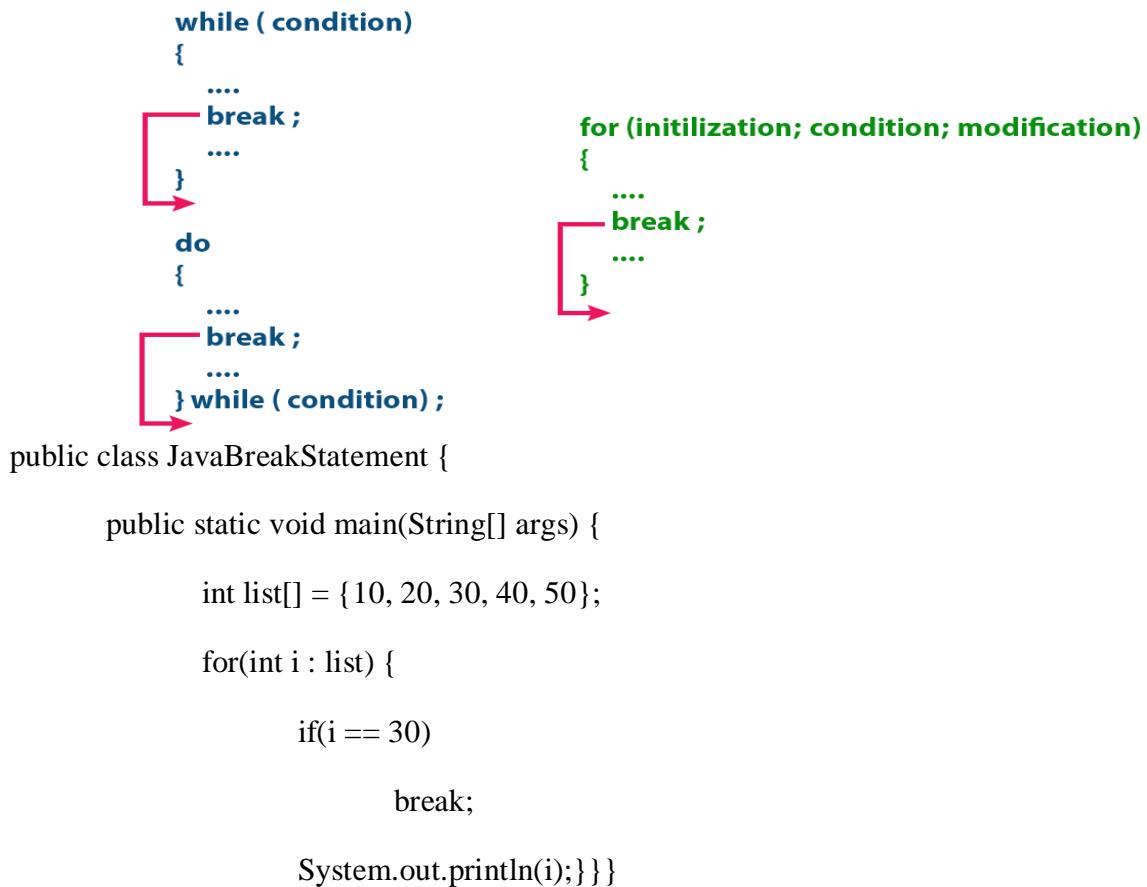
    System.out.println(i); } } }

```

Break:

The break statement in java is used to terminate a switch or looping statement. That means the break statement is used to come out of a switch statement and a looping statement like while, do-while, for, and for-each.

The following picture depicts the execution flow of the break statement.



1.11 Classes and Methods :

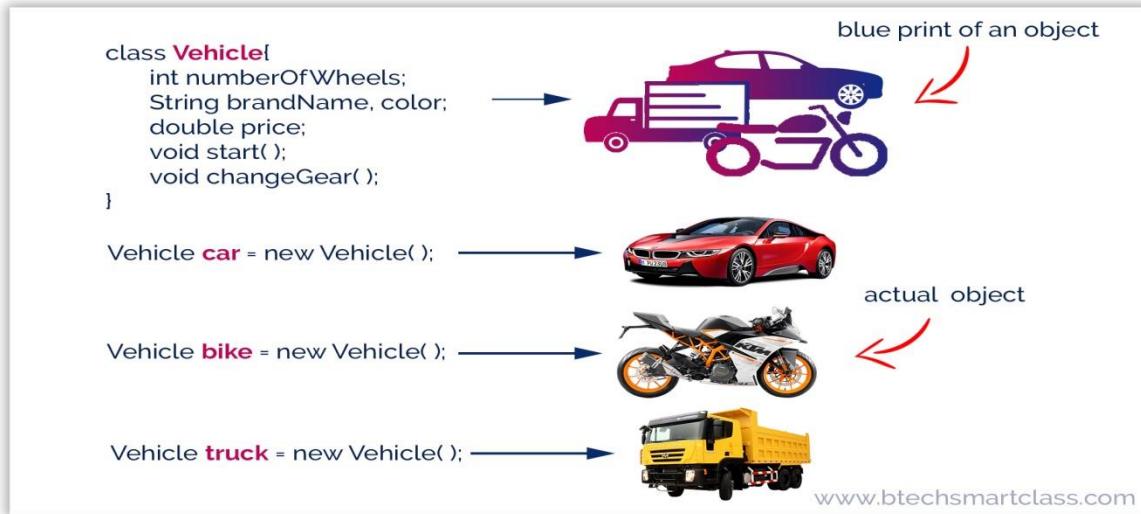
Java is an object-oriented programming language, so everything in java program must be based on the object concept. In a java programming language, the class concept defines the skeleton of an object.

The java class is a template of an object. The class defines the blueprint of an object. Every class in java forms a new data type. Once a class got created, we can generate as many objects as we want. Every class defines the properties and behaviors of an object. All the objects of a class have the same properties and behaviors that were defined in the class.

Every class of java programming language has the following characteristics.

- **Identity** - It is the name given to the class.
- **State** - Represents data values that are associated with an object.
- **Behavior** - Represents actions can be performed by an object.

Look at the following picture to understand the class and object concept.



Creating a Class

In java, we use the keyword `class` to create a class. A class in java contains properties as variables and behaviors as methods. Following is the syntax of class in the java.

```

class class_name{
    // Members of a class
}

```

Creating an Object

In java, an object is an instance of a class. When an object of a class is created, the class is said to be instantiated. All the objects that are created using a single class have the same properties and methods. But the value of properties is different for every object. Following is the syntax of class in the java.

```
Class_name Object_name=new Class_name();
```

Methods:

A method is a block of statements under a name that gets executes only when it is called. Every method is used to perform a specific task. The major advantage of methods is code re-usability (define the code once, and use it many times).

In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class.

Every method in java must be declared inside a class.

Every method declaration has the following characteristics.

- **returnType** - Specifies the data type of a return value.
- **name** - Specifies a unique name to identify it.
- **parameters** - The data values it may accept or receive.
- **{ }** - Defines the block belongs to the method.

Creating a method

A method is created inside the class and it may be created with any access specifier. However, specifying access specifier is optional.

Following is the syntax for creating methods in java.

```
class <ClassName>{
<accessSpecifier><returnType><methodName>( parameters ){
    ...
    block of statements;
    ...
}
}
```

1.12 String handling:

A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class String.

In the background, the string values are organized as an array of a character data type.

The string created using a character array can not be extended. It does not allow to append more characters after its definition, but it can be modified.

Let's look at the following example java code.

Example

```
char[] name = {'J', 'a', 'v', 'a', ' ', 'T', 'u', 't', 'o', 'r', 'i', 'a', 'T', 's'};
//name[14] = '@';      //ArrayIndexOutOfBoundsException
name[5] = '-';
System.out.println(name);
```

The **String** class defined in the package **java.lang** package. The String class implements **Serializable**, **Comparable**, and **CharSequence** interfaces.

The string created using the **String** class can be extended. It allows us to add more characters after its definition, and also it can be modified.

Let's look at the following example java code.

Example

```
String siteName = "btechsmartclass.com";
siteName = "www.btechsmartclass.com";
```

Creating String object in java

In java, we can use the following two ways to create a string object.

- Using string literal
- Using String constructor

Let's look at the following example java code.

Example

```
String title = "Java Tutorials";           // Using literals
String siteName = new String("www.btechsmartclass.com"); // Using constructor
```

String handling methods

In java programming language, the String class contains various methods that can be used to handle string data values. It contains methods like concat(), compareTo(), split(), join(), replace(), trim(), length(), intern(), equals(), comparison(), substring(), etc.

The following table depicts all built-in methods of String class in java.

Method	Description	Return Value
charAt(int)	Finds the character at given index	char
length()	Finds the length of given string	int
compareTo(String)	Compares two strings	int
compareToIgnoreCase(String)	Compares two strings, ignoring case	int
concat(String)	Concatenates the object string with argument string.	String
contains(String)	Checks whether a string contains sub-string	boolean
contentEquals(String)	Checks whether two strings are same	boolean
equals(String)	Checks whether two strings are same	boolean

Method	Description	Return Value
equalsIgnoreCase(String)	Checks whether two strings are same, ignoring case	boolean
startsWith(String)	Checks whether a string starts with the specified string	boolean
endsWith(String)	Checks whether a string ends with the specified string	boolean
getBytes()	Converts string value to bytes	byte[]
hashCode()	Finds the hash code of a string	int
indexOf(String)	Finds the first index of argument string in object string	int
lastIndexOf(String)	Finds the last index of argument string in object string	int
isEmpty()	Checks whether a string is empty or not	boolean
replace(String, String)	Replaces the first string with second string	String
replaceAll(String, String)	Replaces the first string with second string at all occurrences.	String
substring(int, int)	Extracts a sub-string from specified start and end index values	String
toLowerCase()	Converts a string to lower case letters	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends	String
toString(int)	Converts the value to a String object	String
split(String)	splits the string matching argument string	String[]
intern()	returns string from the pool	String
join(String, String, ...)	Joins all strings, first string as delimiter.	String

1.13 Inheritance Concepts:

The inheritance is a very useful and powerful concept of object-oriented programming. In java, using the inheritance concept, we can use the existing features of one class in another class. The inheritance provides a great advantage called code re-usability. With the help of code re-usability, the commonly used code in an application need not be written again and again.

The inheritance can be defined as follows.

The inheritance is the process of acquiring the properties of one class to another class.

Inheritance Basics

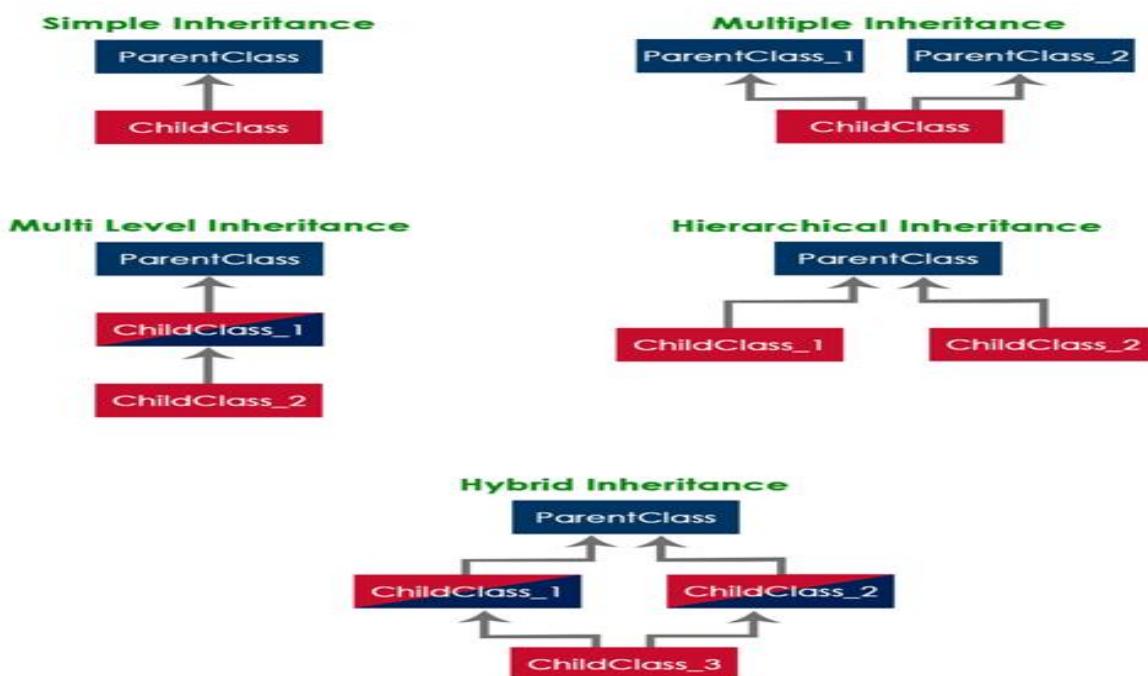
In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.

The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.

There are five types of inheritances, and they are as follows.

- Simple Inheritance (or) Single Inheritance
- Multiple Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

The following picture illustrates how various inheritances are implemented.



Member access:

In Java, the access specifiers (also known as access modifiers) used to restrict the scope or accessibility of a class, constructor, variable, method or data member of class and interface. There are four access specifiers, and their list is below.

- default (or) no modifier
- public
- protected
- private

In java, we can not employ all access specifiers on everything. The following table describes where we can apply the access specifiers.

Access Specifier ► Item ▼	Default	Public	Protected	Private
Class	Yes	Yes	No	No
Inner Class	Yes	Yes	Yes	Yes
Interface	Yes	Yes	No	No
Interface Inside Class	Yes	Yes	Yes	Yes
enum	Yes	Yes	No	No
enum Inside Class	Yes	Yes	Yes	Yes
enum inside Interface	Yes	No	No	No
Constructor	Yes	Yes	Yes	Yes
methods & data inside class	Yes	Yes	Yes	Yes
methods & data inside Interface	Yes	No	No	No

Let's look at the following example java code, which generates an error because a class does not allow private access specifier unless it is an inner class.

In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

Access control for members of class and interface in java

Access Specifier \ Accessibility Location	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

1.14 Constructors :

It is very important to understand how the constructors get executed in the inheritance concept. In the inheritance, the constructors never get inherited to any child class.

In java, the default constructor of a parent class called automatically by the constructor of its child class. That means when we create an object of the child class, the parent class constructor executed, followed by the child class constructor executed.

Let's look at the following example java code.

Example

```
class ParentClass{
    int a;
    ParentClass(){
        System.out.println("Inside ParentClass constructor!");
    }
}

class ChildClass extends ParentClass{
    ChildClass(){
        System.out.println("Inside ChildClass constructor!");
    }
}
```

```

        System.out.println("Inside ChildClass constructor!!");

    }

}

class ChildChildClass extends ChildClass{

    ChildChildClass(){

        System.out.println("Inside ChildChildClass constructor!!");

    }

}

public class ConstructorInInheritance {

    public static void main(String[] args) {

        ChildChildClass obj = new ChildChildClass();

    }

}

```

Super Keyword:

In java, super is a keyword used to refers to the parent class object. The super keyword came into existence to solve the naming conflicts in the inheritance. When both parent class and child class have members with the same name, then the super keyword is used to refer to the parent class version.

In java, the super keyword is used for the following purposes.

- To refer parent class data members
- To refer parent class methods
- To call parent class constructor

super to refer parent class data members

When both parent class and child class have data members with the same name, then the super keyword is used to refer to the parent class data member from child class.

Let's look at the following example java code.

Example

```

class ParentClass{

    int num = 10;

}

```

```
class ChildClass extends ParentClass{  
    int num = 20;  
    void showData() {  
        System.out.println("Inside the ChildClass");  
        System.out.println("ChildClass num = " + num);  
        System.out.println("ParentClass num = " + super.num);  
    }  
}  
  
public class SuperKeywordExample {  
    public static void main(String[] args) {  
        ChildClass obj = new ChildClass();  
        obj.showData();  
        System.out.println("\nInside the non-child class");  
        System.out.println("ChildClass num = " + obj.num);  
        //System.out.println("ParentClass num = " + super.num); //super can't be used  
here  
    }  
}
```

super to refer parent class method

When both parent class and child class have method with the same name, then the super keyword is used to refer to the parent class method from child class.

Let's look at the following example java code.

Example

```
class ParentClass{  
    int num1 = 10;  
    void showData() {  
        System.out.println("\nInside the ParentClass showData method");  
        System.out.println("ChildClass num = " + num1);}}}
```

```
class ChildClass extends ParentClass{  
    int num2 = 20;  
    void showData() {  
        System.out.println("\nInside the ChildClass showData method");  
        System.out.println("ChildClass num = " + num2);  
        super.showData(); } }  
  
public class SuperKeywordExample {  
    public static void main(String[] args) {  
        ChildClass obj = new ChildClass();  
        obj.showData();  
        //super.showData(); // super can't be used here  
    }  
}
```

super to call parent class constructor

When an object of child class is created, it automatically calls the parent class default- constructor before it's own. But, the parameterized constructor of parent class must be called explicitly using the super keyword inside the child class constructor.

Let's look at the following example java code.

Example

```
class ParentClass{  
  
    int num1;  
  
    ParentClass(){  
        System.out.println("\nInside the ParentClass default constructor");  
        num1 = 10;  
    }  
}
```

```
ParentClass(int value){  
    System.out.println("\nInside the ParentClass parameterized constructor");  
    num1 = value;  
}  
  
class ChildClass extends ParentClass{  
    int num2;  
    ChildClass(){  
        super(100);  
        System.out.println("\nInside the ChildClass constructor");  
        num2 = 200;  
    }  
}  
  
public class SuperKeywordExample {  
    public static void main(String[] args) {  
        ChildClass obj = new ChildClass(); } }  

```

Final With Inheritance:

In java, the final is a keyword and it is used with the following things.

- With variable (to create constant)
- With method (to avoid method overriding)
- With class (to avoid inheritance)

Let's look at each of the above.

final with variables

When a variable defined with the **final** keyword, it becomes a constant, and it does not allow us to modify the value. The variable defined with the final keyword allows only a one-time assignment, once a value assigned to it, never allows us to change it again.

Let's look at the following example java code.

Example

```
public class FinalVariableExample {  
    public static void main(String[] args) {  
        final int a = 10;  
        System.out.println("a = " + a);  
        a = 100;      // Can't be modifier  
    }  
}
```

final with methods

When a method defined with the **final** keyword, it does not allow it to override. The final method extends to the child class, but the child class can not override or re-define it. It must be used as it has implemented in the parent class.

Let's look at the following example java code.

Example

```
class ParentClass{  
    int num = 10;  
    final void showData() {  
        System.out.println("Inside ParentClass showData() method");  
        System.out.println("num = " + num);  
    }  
}
```

```
class ChildClass extends ParentClass{  
    void showData() {  
        System.out.println("Inside ChildClass showData() method");  
        System.out.println("num = " + num);  
    }  
}
```

```
public class FinalKeywordExample {  
    public static void main(String[] args)  
    {  
        ChildClass obj = new ChildClass();  
        obj.showData();  
    }  
}
```

final with class

When a class defined with final keyword, it can not be extended by any other class.

Let's look at the following example java code.

Example

```
final class ParentClass{  
    int num = 10;  
    void showData() {  
        System.out.println("Inside ParentClass showData() method");  
        System.out.println("num = " + num);  
    }  
}  
  
class ChildClass extends ParentClass  
{  
}  
  
public class FinalKeywordExample  
{  
    public static void main(String[] args)  
    {  
        ChildClass obj = new ChildClass();  
    }  
}
```

1.15 Polymorphism:

The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

In java, polymorphism implemented using method overloading and method overriding.

Ad hoc polymorphism

The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments. In a java programming language, ad hoc polymorphism carried out with a method overloading concept.

In ad hoc polymorphism the method binding happens at the time of compilation. Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments.

The ad hoc polymorphism implemented within the class only.

Let's look at the following example java code.

Example

```
import java.util.Arrays;

public class AdHocPolymorphismExample {

    void sorting(int[] list) {
        Arrays.parallelSort(list);
        System.out.println("Integers after sort: " + Arrays.toString(list));
    }

    void sorting(String[] names) {
        Arrays.parallelSort(names);
        System.out.println("Names after sort: " + Arrays.toString(names));
    }

    public static void main(String[] args) {
        AdHocPolymorphismExample obj = new AdHocPolymorphismExample();
        int list[] = {2, 3, 1, 5, 4};
        obj.sorting(list);      // Calling with integer array
        String[] names = {"rama", "raja", "shyam", "seeta"};
        obj.sorting(names);    // Calling with String array
    }
}
```

```
    }  
}
```

Pure polymorphism

The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept.

In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference.

When a child class has a definition for a member function of the parent class, the parent class function is said to be overridden.

The pure polymorphism implemented in the inheritance concept only.

Let's look at the following example java code.

Example

```
class ParentClass{  
    int num = 10;  
    void showData() {  
        System.out.println("Inside ParentClass showData() method");  
        System.out.println("num = " + num);  
    }  
}  
  
class ChildClass extends ParentClass{  
    void showData() {  
        System.out.println("Inside ChildClass showData() method");  
        System.out.println("num = " + num);  
    }  
}  
  
public class PurePolymorphism {  
    public static void main(String[] args) {  
        ParentClass obj = new ParentClass();
```

```

        obj.showData();

        obj = new ChildClass();

        obj.showData();

    }

}

```

1.16 Method Overriding:

The method overriding is the process of re-defining a method in a child class that is already defined in the parent class. When both parent and child classes have the same method, then that method is said to be the overriding method.

The method overriding enables the child class to change the implementation of the method which aquired from parent class according to its requirement.

In the case of the method overriding, the method binding happens at run time. The method binding which happens at run time is known as late binding. So, the method overriding follows late binding.

The method overriding is also known as **dynamic method dispatch** or **run time polymorphism** or **pure polymorphism**.

Let's look at the following example java code.

Example

```

class ParentClass{

    int num = 10;

    void showData() {

        System.out.println("Inside ParentClass showData() method");

        System.out.println("num = " + num);

    }

}

class ChildClass extends ParentClass{

    void showData() {

        System.out.println("Inside ChildClass showData() method");

        System.out.println("num = " + num);

    }

}

```

```
public class PurePolymorphism {  
    public static void main(String[] args) {  
        ParentClass obj = new ParentClass();  
        obj.showData();  
        obj = new ChildClass();  
        obj.showData();  
    }  
}
```

Rules for method overriding

While overriding a method, we must follow the below list of rules.

- Static methods can not be overridden.
- Final methods can not be overridden.
- Private methods can not be overridden.
- Constructor can not be overridden.
- An abstract method must be overridden.
- Use super keyword to invoke overridden method from child class.
- The return type of the overriding method must be same as the parent has it.
- The access specifier of the overriding method can be changed, but the visibility must increase but not decrease. For example, a protected method in the parent class can be made public, but not private, in the child class.
- If the overridden method does not throw an exception in the parent class, then the child class overriding method can only throw the unchecked exception, throwing a checked exception is not allowed.
- If the parent class overridden method does throw an exception, then the child class overriding method can only throw the same, or subclass exception, or it may not throw any exception.

1.17 Abstract Classes :

An abstract class is a class that created using abstract keyword. In other words, a class prefixed with abstract keyword is known as an abstract class.

In java, an abstract class may contain abstract methods (methods without implementation) and also non-abstract methods (methods with implementation).

We use the following syntax to create an abstract class.

Syntax

```
abstract class <ClassName>{
```

```
...
```

```
}
```

Let's look at the following example java code.

Example

```
import java.util.*;  
  
abstract class Shape {  
  
    int length, breadth, radius;  
  
    Scanner input = new Scanner(System.in);  
  
    abstract void printArea();  
  
}  
  
class Rectangle extends Shape {  
  
    void printArea() {  
  
        System.out.println("/** Finding the Area of Rectangle **");  
  
        System.out.print("Enter length and breadth: ");  
  
        length = input.nextInt();  
  
        breadth = input.nextInt();  
  
        System.out.println("The area of Rectangle is: " + length * breadth);  
  
    }  
  
}  
  
class Triangle extends Shape {  
  
    void printArea() {  
  
        System.out.println("\n/** Finding the Area of Triangle **");  
  
        System.out.print("Enter Base And Height: ");  
  
        length = input.nextInt();  
  
    }  
  
}
```

```
        breadth = input.nextInt();

        System.out.println("The area of Triangle is: " + (length * breadth) / 2);

    }

}

class Cricle extends Shape {

    void printArea() {

        System.out.println("\n*** Finding the Area of Cricle ***");

        System.out.print("Enter Radius: ");

        radius = input.nextInt();

        System.out.println("The area of Cricle is: " + 3.14f * radius * radius);

    }

}

public class AbstractClassExample {

    public static void main(String[] args) {

        Rectangle rec = new Rectangle();

        rec.printArea();

        Triangle tri = new Triangle();

        tri.printArea();

        Cricle cri = new Cricle();

        cri.printArea();

    }

}
```

Rules for method overriding

An abstract class must follow the below list of rules.

- An abstract class must be created with abstract keyword.
- An abstract class can be created without any abstract method.
- An abstract class may contain abstract methods and non-abstract methods.
- An abstract class may contain final methods that can not be overridden.
- An abstract class may contain static methods, but the abstract method can not be static.

- An abstract class may have a constructor that gets executed when the child class object created.
- An abstract method must be overridden by the child class, otherwise, it must be defined as an abstract class.
- An abstract class can not be instantiated but can be referenced.

1.8 Object Class:

In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the **java.lang** package. Every class in the java programming language is a subclass of Object class by default. The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.

Methods of Object class

The following table depicts all built-in methods of Object class in java.

Method	Description	Return Value
getClass()	Returns Class class object	object
hashCode()	returns the hashcode number for object being used.	int
equals(Object obj)	compares the argument object to calling object.	boolean
clone()	Compares two strings, ignoring case	int
concat(String)	Creates copy of invoking object	object
toString()	eturns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void
wait(long,int)	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

1.19 Forms of Inheritance:

The inheritance concept used for the number of purposes in the java programming language. One of the main purposes is substitutability. The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with

the child class object. For example, if B is a child class of A, anywhere we expect an instance of A we can use an instance of B.

The substitutability can achieve using inheritance, whether using extends or implements keywords.

The following are the different forms of inheritance in java.

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Specialization

It is the most ideal form of inheritance. The subclass is a special case of the parent class. It holds the principle of substitutability.

Specification

This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them. The Java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.

Construction

This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.

Extension

This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

Limitation

This is another form of inheritance where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.

Combination

This is another form of inheritance where the subclass inherits properties from multiple parent classes. Java does not support multiple inheritance type.

1.20 Benefits and Cost Of Inheritance:

The inheritance is the core and more useful concept Object Oriented Programming. It provides inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance

leads to less development and maintenance costs. It provides lot of benefits and few of them are listed below.

Benefits of Inheritance

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

Costs of Inheritance

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

UNIT - II
PACKAGES AND INTERFACES

2.1 Defining Packages

In java, a package is a container of classes, interfaces, and sub-packages. We may think of it as a folder in a file directory.

We use the packages to avoid naming conflicts and to organize project-related classes, interfaces, and sub-packages into a bundle.

In java, the packages have divided into two types.

- Built-in Packages
- User-defined Packages

Built-in Packages

The built-in packages are the packages from java API. The Java API is a library of pre-defined classes, interfaces, and sub-packages. The built-in packages were included in the JDK.

There are many built-in packages in java, few of them are as java, lang, io, util, awt, javax, swing, net, sql, etc.

We need to import the built-in packages to use them in our program. To import a package, we use the import statement.

User-defined Packages

The user-defined packages are the packages created by the user. User is free to create their own packages.

Defining a Package in java

We use the **package** keyword to create or define a package in java programming language.

Syntax : package Package_name;

Let's consider the following code to create a user-defined package myPackage.

Example

```
package myPackage;

public class DefiningPackage {
    public static void main(String[] args) {
        System.out.println("This class belongs to myPackage.");
    }
}
```

Now, save the above code in a file **DefiningPackage.java**, and compile it using the following command.

javac -d . DefiningPackage.java

The above command creates a directory with the package name myPackage, and the DefiningPackage.class is saved into it.

Run the program use the following command.

java myPackage.DefiningPackage

2.2 CLASSPATH: CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .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 CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end. The following points describe what should be the end of the CLASSPATH.

- If a JAR or zip, the file contains class files, the CLASSPATH end with the name of the zip or JAR file.
- If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
- If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name, that is the first package in the full package name.

The default value of CLASSPATH is a dot (.). It means the only current directory searched. The default value of CLASSPATH overrides when you set the CLASSPATH variable or using the -classpath command (for short -cp). Put a dot (.) in the new setting if you 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 and use it, irrespective of the same name class presents in another directory which is also included in the CLASSPATH.

If you want to set multiple classpaths, then you need to separate each CLASSPATH by a semicolon (;).

The third-party applications (MySQL and Oracle) that use the JVM can modify the CLASSPATH environment variable to include the libraries they use. The classes can be stored in directories or archives files. The classes of the Java platform are stored in rt.jar.

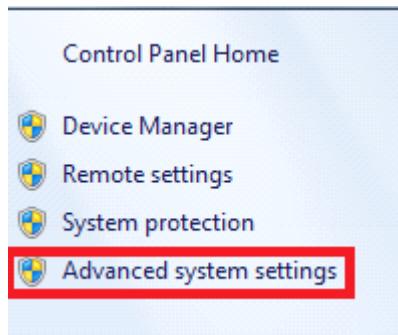
There are two ways to set CLASSPATH: through Command Prompt or by setting Environment Variable.

Let's see how to set CLASSPATH of MySQL database:

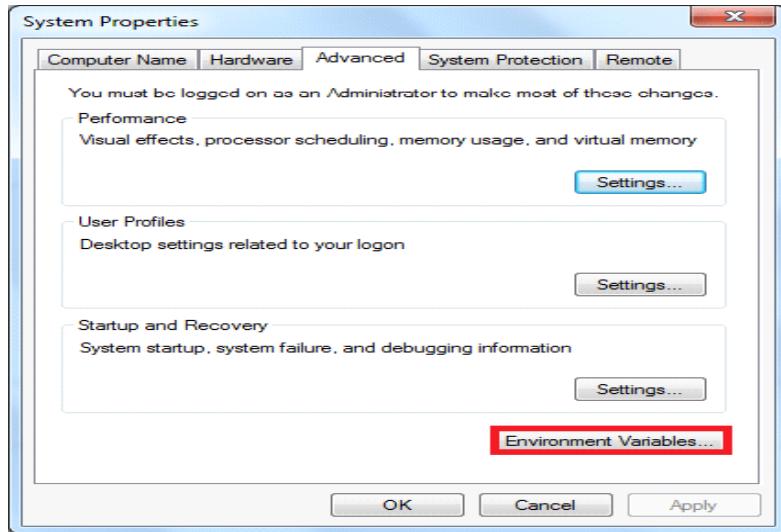
Step 1: Click on the Windows button and choose Control Panel. Select System.



Step 2: Click on **Advanced System Settings**.



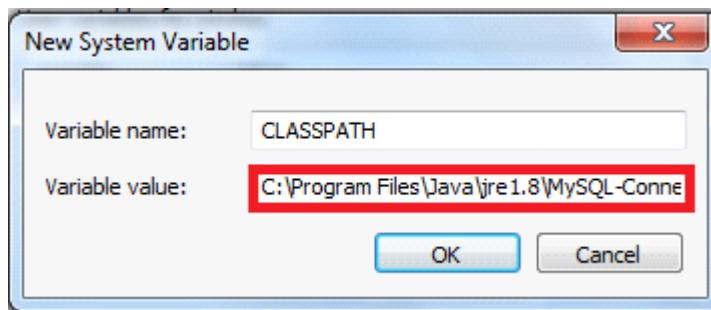
Step 3: A dialog box will open. Click on Environment Variables.



Step 4: If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as *C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;;*

Remember: Put *;;* at the end of the CLASSPATH.



Difference between PATH and CLASSPATH

PATH	CLASSPATH
PATH is an environment variable.	CLASSPATH is also an environment variable.
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.
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.

2.3 Access Protection:

In java, the access modifiers define the accessibility of the class and its members. For example, private members are accessible within the same class members only. Java has four access modifiers, and they are default, private, protected, and public.

In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.

In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

- The public members can be accessed everywhere.
- The private members can be accessed only inside the same class.
- The protected members are accessible to every child class (same package or other packages).
- The default members are accessible within the same package but not outside the package.

Access control for members of class and interface in java

Access Specifier \ Accessibility Location	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Let's look at the following example java code.

Example

```
class ParentClass{
    int a = 10;
    public int b = 20;
    protected int c = 30;
    private int d = 40;
    void showData() {
        System.out.println("Inside ParentClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

```
class ChildClass extends ParentClass{
```

```
    void accessData() {
        System.out.println("Inside ChildClass");
        System.out.println("a = " + a);
    }
}
```

```

        System.out.println("b = " + b);
        System.out.println("c = " + c);
        //System.out.println("d = " + d); } } // private member can't be accessed

public class AccessModifiersExample {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
        obj.showData();
        obj.accessData(); } }

```

2.4 Importing Packages :

In java, the import keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly. The import statement must be after the package statement, and before any other statement.

Importing specific class

Using an importing statement, we can import a specific class. The following syntax is employed to import a specific class.

Syntax

```
import packageName.ClassName;
```

Let's look at an import statement to import a built-in package and Scanner class.

Example

```

package myPackage;

import java.util.Scanner;

public class ImportingExample {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
        int i = read.nextInt();
        System.out.println("You have entered a number " + i);
    }
}

```

In the above code, the class ImportingExample belongs to myPackage package, and it also importing a class called Scanner from java.util package.

Importing all the classes

Using an importing statement, we can import all the classes of a package. To import all the classes of the package, we use * symbol. The following syntax is employed to import all the classes of a package.

Syntax

```
import packageName.*;
```

Let's look at an import statement to import a built-in package.

Example

```
package myPackage;  
  
import java.util.*;  
  
public class ImportingExample {  
  
    public static void main(String[] args) {  
  
        Scanner read = new Scanner(System.in);  
  
        int i = read.nextInt();  
  
        System.out.println("You have entered a number " + i);  
  
        Random rand = new Random();  
  
        int num = rand.nextInt(100);  
  
        System.out.println("Randomly generated number " + num);}}}
```

In the above code, the class ImportingExample belongs to myPackage package, and it also importing all the classes like Scanner, Random, Stack, Vector, ArrayList, HashSet, etc. from the java.util package.

The import statement imports only classes of the package, but not sub-packages and its classes. We may also import sub-packages by using a symbol '.' (dot) to separate parent package and sub-package.

Consider the following import statement.

```
import java.util.*;
```

The above import statement util is a sub-package of java package. It imports all the classes of util package only, but not classes of java package.

2.5 Defining An Interface:

In java, an interface is similar to a class, but it contains abstract methods and static final variables only. The interface in Java is another mechanism to achieve abstraction. We may think of an interface as a completely abstract class. None of the methods in the interface has an implementation, and all the variables in the interface are constants.

All the methods of an interface, implemented by the class that implements it.

The interface in java enables java to support multiple-inheritance. An interface may extend only one interface, but a class may implement any number of interfaces.

- An interface is a container of abstract methods and static final variables.
- An interface, implemented by a class. (class implements interface).
- An interface may extend another interface. (Interface extends Interface).
- An interface never implements another interface, or class.
- A class may implement any number of interfaces.
- We can not instantiate an interface.
- Specifying the keyword abstract for interface methods is optional, it automatically added.
- All the members of an interface are public by default.

Defining an interface in java

Defining an interface is similar to that of a class. We use the keyword interface to define an interface. All the members of an interface are public by default. The following is the syntax for defining an interface.

Syntax

```
interface InterfaceName{  
    ...  
    members declaration;  
    ...  
}
```

Let's look at an example code to define an interface.

Example

```
interface HumanInterfaceExample {  
    void learn(String str);  
    void work();  
    int duration = 10; }
```

In the above code defines an interface HumanInterfaceExample that contains two abstract methods learn(), work() and one constant duration. Every interface in Java is auto-completed by

the compiler. For example, in the above example code, no member is defined as public, but all are public automatically.

The above code automatically converted as follows.

Converted code

```
interface HumanInterfaceExample {  
    public abstract void learn(String str);  
    public abstract void work();  
    public static final int duration = 10;  
}
```

In the next tutorial, we will learn how a class implements an interface to make use of the interface concept.

2.6 Implementing Interface:

In java, an interface is implemented by a class. The class that implements an interface must provide code for all the methods defined in the interface, otherwise, it must be defined as an abstract class.

The class uses a keyword `implements` to implement an interface. A class can implement any number of interfaces. When a class wants to implement more than one interface, we use the `implements` keyword followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defining a class that implements an interface.

Syntax

```
class className implements InterfaceName{  
    ...  
    boby-of-the-class  
    ...  
}
```

Let's look at an example code to define a class that implements an interface.

Example

```
interface Human {  
    void learn(String str);  
    void work();  
    int duration = 10;  
}  
  
class Programmer implements Human{  
    public void learn(String str) {  
        System.out.println("Learn using " + str);  
    }  
    public void work() {  
        System.out.println("Develop applications");  
    }  
}  
  
public class HumanTest {  
    public static void main(String[] args) {  
        Programmer trainee = new Programmer();  
        trainee.learn("coding");  
        trainee.work();  
    }  
}
```

In the above code defines an interface Human that contains two abstract methods learn(), work() and one constant duration. The class Programmer implements the interface. As it implementing the Human interface it must provide the body of all the methods those defined in the Human interface.

Implementing multiple Interfaces

When a class wants to implement more than one interface, we use the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defining a class that implements multiple interfaces.

Syntax

```
class className implements InterfaceName1, InterfaceName2, ...{  
    ...  
    boby-of-the-class  
    ...  
}
```

Let's look at an example code to define a class that implements multiple interfaces.

Example

```
interface Human {  
    void learn(String str);  
    void work();  
}  
  
interface Recruitment {  
    boolean screening(int score);  
    boolean interview(boolean selected);  
}  
  
class Programmer implements Human, Recruitment {  
    public void learn(String str) {  
        System.out.println("Learn using " + str);  
    }  
    public boolean screening(int score) {  
        System.out.println("Attend screening test");  
        int threshold = 20;  
        if(score > threshold)  
            return true;  
        return false;    }  
}
```

```
public boolean interview(boolean selected) {  
    System.out.println("Attend interview");  
    if(selected)  
        return true;  
    return false;  
}  
  
public void work() {  
    System.out.println("Develop applications");  
}  
}  
  
public class HumanTest {  
    public static void main(String[] args) {  
        Programmer trainee = new Programmer();  
        trainee.learn("Coding");  
        trainee.screening(30);  
        trainee.interview(true);  
        trainee.work();}}}
```

In the above code defines two interfaces Human and Recruitment, and a class Programmer implements both the interfaces.

2.7 Nested Interfaces:

In java, an interface may be defined inside another interface, and also inside a class. The interface that defined inside another interface or a class is known as nested interface. The nested interface is also referred as inner interface.

- The nested interface declared within an interface is public by default.
- The nested interface declared within a class can be with any access modifier.
- Every nested interface is static by default.

The nested interface cannot be accessed directly. We can only access the nested interface by using outer interface or outer class name followed by dot(.), followed by the nested interface name.

Nested interface inside another interface

The nested interface that defined inside another interface must be accessed as OuterInterface.InnerInterface.

Let's look at an example code to illustrate nested interfaces inside another interface.

Example

```
interface OuterInterface{  
    void outerMethod();  
  
    interface InnerInterface{  
        void innerMethod(); } }  
  
class OnlyOuter implements OuterInterface{  
    public void outerMethod() {  
        System.out.println("This is OuterInterface method");  
    }  
}  
  
class OnlyInner implements OuterInterface.InnerInterface{  
    public void innerMethod() {  
        System.out.println("This is InnerInterface method");  
    }  
}  
  
public class NestedInterfaceExample  
{  
    public static void main(String[] args) {  
        OnlyOuter obj_1 = new OnlyOuter();  
        OnlyInner obj_2 = new OnlyInner();  
        obj_1.outerMethod();  
        obj_2.innerMethod();  
    }  
}
```

Nested interface inside a class

The nested interface that defined inside a class must be accessed as `ClassName.InnerInterface`.

Let's look at an example code to illustrate nested interfaces inside a class.

Example

```
class OuterClass

{

    interface InnerInterface{

        void innerMethod();

    }

}

class ImplementingClass implements OuterClass.InnerInterface{

    public void innerMethod() {

        System.out.println("This is InnerInterface method");

    }

}

public class NestedInterfaceExample {

    public static void main(String[] args) {

        ImplementingClass obj = new ImplementingClass();

        obj.innerMethod();}

}
```

2.8 Variables In Interfaces And Extending Interfaces:

In java, an interface is a completely abstract class. An interface is a container of abstract methods and static final variables. The interface contains the static final variables. The variables defined in an interface can not be modified by the class that implements the interface, but it may use as it defined in the interface.

- The variable in an interface is public, static, and final by default.
- If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.
- No access modifier is allowed except the public for interface variables.
- Every variable of an interface must be initialized in the interface itself.
- The class that implements an interface can not modify the interface variable, but it may use as it defined in the interface.

Let's look at an example code to illustrate variables in an interface.

Example

```
interface SampleInterface{  
    int UPPER_LIMIT = 100;  
    //int LOWER_LIMIT; // Error - must be initialised  
}  
  
public class InterfaceVariablesExample implements SampleInterface{  
    public static void main(String[] args) {  
        System.out.println("UPPER LIMIT = " + UPPER_LIMIT);  
        // UPPER_LIMIT = 150; // Can not be modified  
    }  
}
```

In java, an interface can extend another interface. When an interface wants to extend another interface, it uses the keyword extends. The interface that extends another interface has its own members and all the members defined in its parent interface too. The class which implements a child interface needs to provide code for the methods defined in both child and parent interfaces, otherwise, it needs to be defined as abstract class.

- An interface can extend another interface.
- An interface can not 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.

Let's look at an example code to illustrate extending an interface.

Example

```
interface ParentInterface{  
    void parentMethod();  
}  
  
interface ChildInterface extends ParentInterface{  
    void childMethod();  
}
```

```
class ImplementingClass implements ChildInterface{  
    public void childMethod() {  
        System.out.println("Child Interface method!!");  
    }  
    public void parentMethod() {  
        System.out.println("Parent Interface method!");  
    }  
}  
  
public class ExtendingAnInterface {  
    public static void main(String[] args) {  
        ImplementingClass obj = new ImplementingClass();  
        obj.childMethod();  
        obj.parentMethod();  
    }  
}
```

2.9 Stream classes:

In java, the IO operations are performed using the concept of streams. Generally, a stream means a continuous flow of data. In java, a stream is a logical container of data that allows us to read from and write to it. A stream can be linked to a data source, or data destination, like a console, file or network connection by java IO system. The stream-based IO operations are faster than normal IO operations.

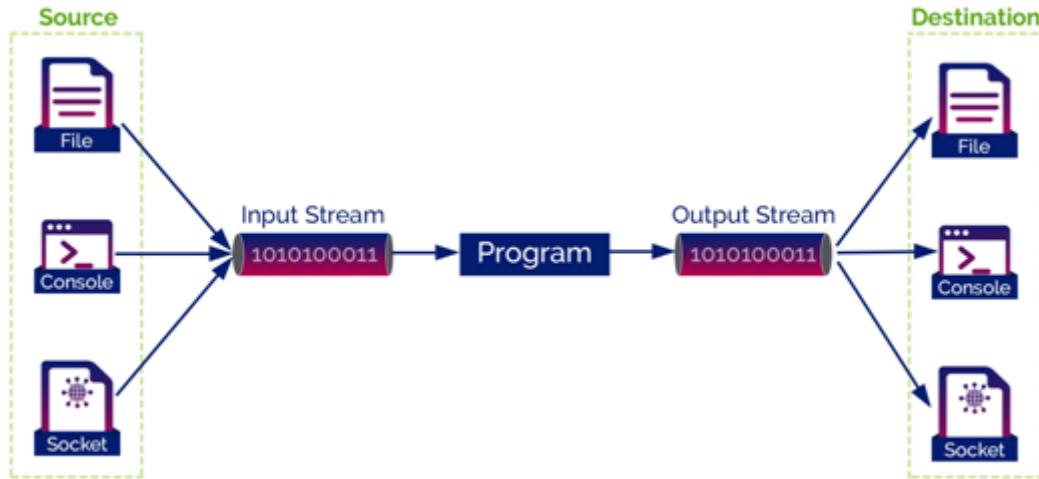
The Stream is defined in the `java.io` package.

To understand the functionality of java streams, look at the following picture.

In java, the stream-based IO operations are performed using two separate streams input stream and output stream. The input stream is used for input operations, and the output stream is used for output operations. The java stream is composed of bytes.

In Java, every program creates 3 streams automatically, and these streams are attached to the console.

- **System.out**: standard output stream for console output operations.
- **System.in**: standard input stream for console input operations.
- **System.err**: standard error stream for console error output operations.

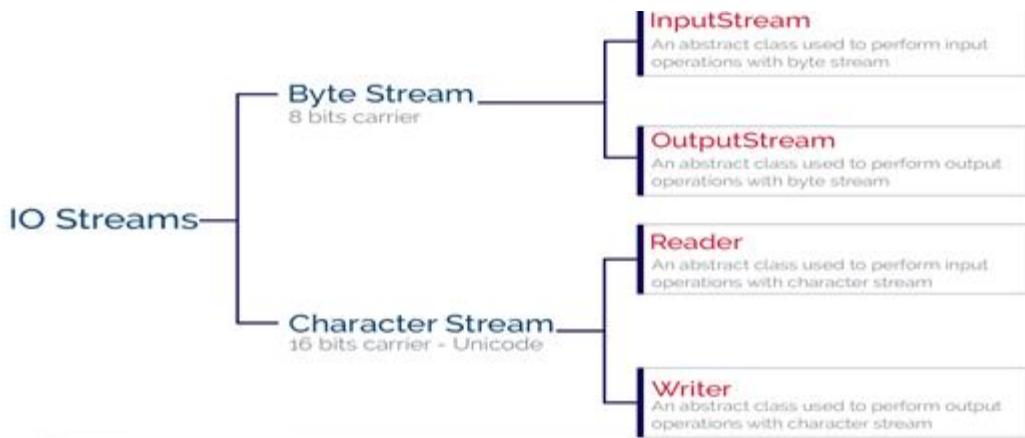


The Java streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

Java provides two types of streams, and they are as follows.

- **Byte Stream**
- **Character Stream**

The following picture shows how streams are categorized, and various built-in classes used by the java IO system.



Both character and byte streams essentially provides a convenient and efficient way to handle data streams in Java.

Byte Stream Classes:

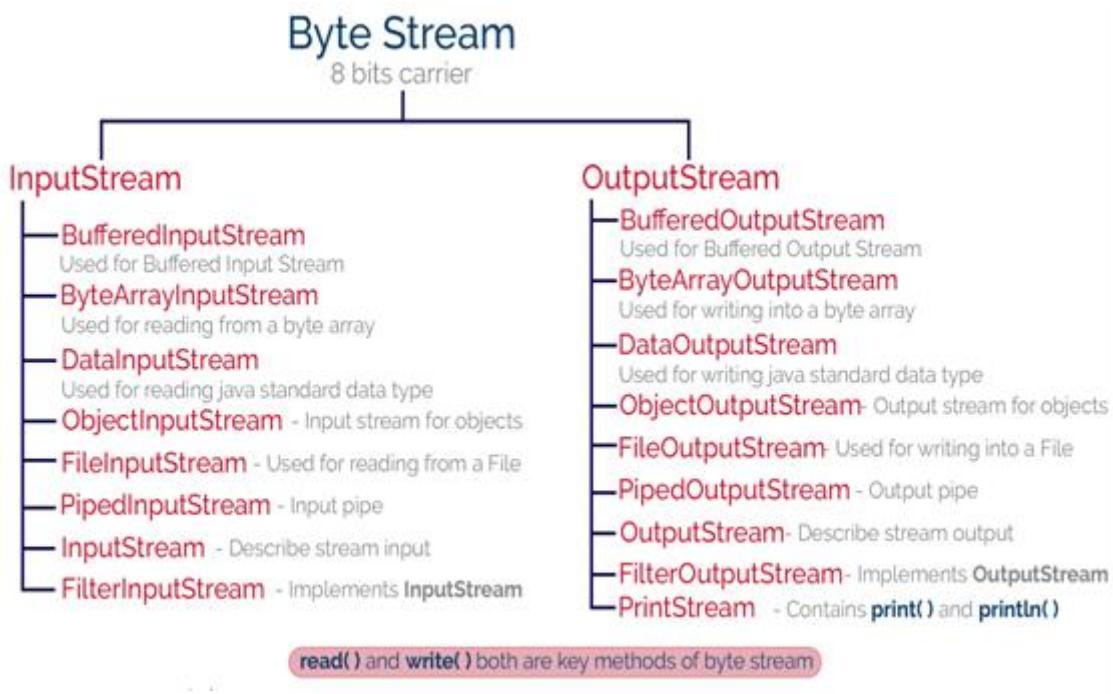
In java, the byte stream is an 8 bits carrier. The byte stream in java allows us to transmit 8 bits of data.

In Java 1.0 version all IO operations were byte oriented, there was no other stream (character stream).

The java byte stream is defined by two abstract classes, `InputStream` and `OutputStream`. The `InputStream` class used for byte stream based input operations, and the `OutputStream` class used for byte stream based output operations.

The `InputStream` and `OutputStream` classes have several concrete classes to perform various IO operations based on the byte stream.

The following picture shows the classes used for byte stream operations.



InputStream class

The `InputStream` class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	<code>int available()</code> It returns the number of bytes that can be read from the input stream.
2	<code>int read()</code> It reads the next byte from the input stream.
3	<code>int read(byte[] b)</code> It reads a chunk of bytes from the input stream and store them in its byte array, b.
4	<code>void close()</code> It closes the input stream and also frees any resources connected with this input stream.

OutputStream class

The OutputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	void write(int n) It writes byte(contained in an int) to the output stream.
2	void write(byte[] b) It writes a whole byte array(b) to the output stream.
3	void flush() It flushes the output stream by forcing out buffered bytes to be written out.
4	void close() It closes the output stream and also frees any resources connected with this output stream.

Reading data using BufferedInputStream

We can use the BufferedInputStream class to read data from the console. The BufferedInputStream class use a method read() to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using BufferedInputStream.

Example 1 - Reading from console

```
import java.io.*;

public class ReadingDemo {

    public static void main(String[] args) throws IOException {
        BufferedInputStream read = new BufferedInputStream(System.in);
        try {
            System.out.print("Enter any character: ");
            char c = (char)read.read();
            System.out.println("You have entered '" + c + "'");
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

```
    finally {  
        read.close();}}}
```

Writing data using **BufferedOutputStream**

We can use the **BufferedOutputStream** class to write data into the console, file, socket. The **BufferedOutputStream** class use a method **write()** to write data.

Let's look at an example code to illustrate writing data into a file using **BufferedOutputStream**.

Example - Writing data into a file

```
import java.io.*;  
  
public class WritingDemo {  
  
    public static void main(String[] args) throws IOException {  
  
        String data = "Java tutorials by BTech Smart Class";  
  
        BufferedOutputStream out = null;  
  
        try {  
  
            FileOutputStream fileOutputStream = new FileOutputStream(new  
File("C:\\\\Raja\\\\dataFile.txt"));  
  
            out = new BufferedOutputStream(fileOutputStream);  
  
            out.write(data.getBytes());  
  
            System.out.println("Writing data into a file is success!");  
  
        }  
  
        catch(Exception e) {  
  
            System.out.println(e);  
  
        }  
  
        finally {  
  
            out.close();}}}
```

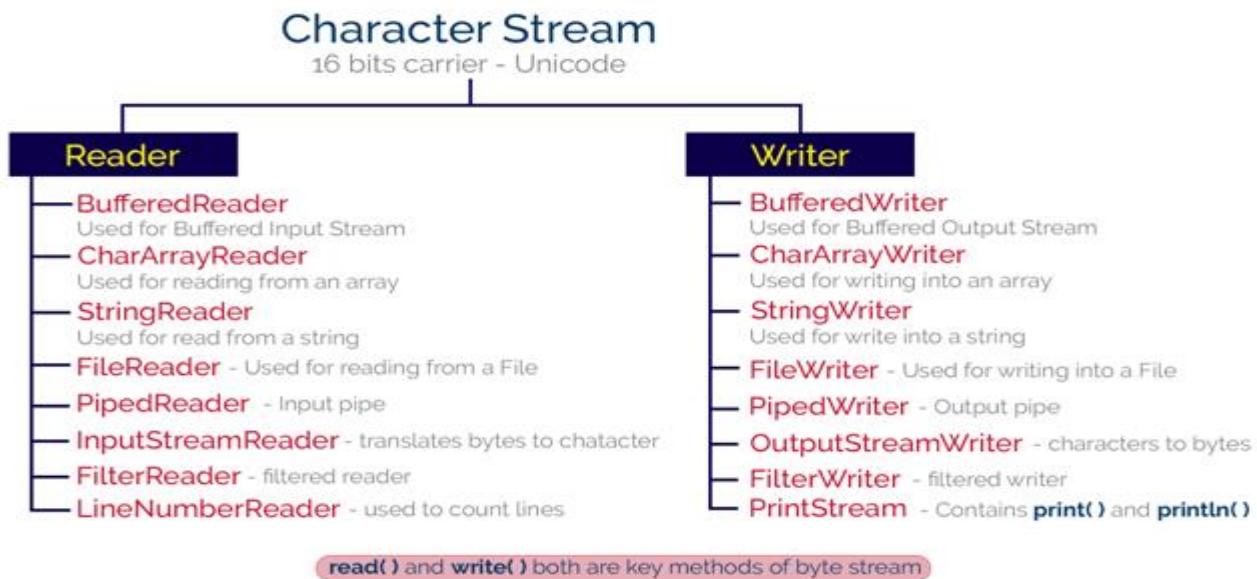
Character Stream classes

In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream. The unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a

character set. In java, the character stream is a 16 bits carrier. The character stream in java allows us to transmit 16 bits of data. The character stream was introduced in Java 1.1 version. The character stream. The java character stream is defined by two abstract classes, Reader and Writer. The Reader class used for character stream based input operations, and the Writer class used for character stream based output operations.

The Reader and Writer classes have several concrete classes to perform various IO operations based on the character stream.

The following picture shows the classes used for character stream operations.



Reader class

The Reader class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	int read() It reads the next character from the input stream.
2	int read(char[] cbuffer) It reads a chunk of characters from the input stream and store them in its byte array, cbuffer.
3	int read(char[] cbuf, int off, int len) It reads characters into a portion of an array.
4	int read(CharBuffer target) It reads characters into into the specified character buffer.
5	String readLine()

	It reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.
6	boolean ready() It tells whether the stream is ready to be read.
7	void close() It closes the input stream and also frees any resources connected with this input stream.

Writer class

The Writer class has been defined as an abstract class, and it has the following methods which have been implemented by its concrete classes.

S.No.	Method with Description
1	void flush() It flushes the output stream by forcing out buffered bytes to be written out.
2	void write(char[] cbuf) It writes a whole array(cbuf) to the output stream.
3	void write(char[] cbuf, int off, int len) It writes a portion of an array of characters.
4	void write(int c) It writes single character.
5	void write(String str) It writes a string.
6	void write(String str, int off, int len) It writes a portion of a string.
7	Writer append(char c) It appends the specified character to the writer.

Reading data using BufferedReader

We can use the BufferedReader class to read data from the console. The BufferedInputStream class needs InputStreamReader class. The BufferedReader uses a method read() to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using BufferedReader.

Example 1 - Reading from console

```
import java.io.*;  
  
public class ReadingDemo {  
  
    public static void main(String[] args) throws IOException {  
  
        InputStreamReader isr = new InputStreamReader(System.in);  
  
        BufferedReader in = new BufferedReader(isr);  
  
        String name = "";  
  
        System.out.print("Please enter your name: ");  
  
        name = in.readLine();  
  
        System.out.println("Hello, " + name + "!");  
  
    }  
  
}
```

Writing data using FileWriter

We can use the `FileWriter` class to write data into the file. The `FileWriter` class use a method `write()` to write data.

Let's look at an example code to illustrate writing data into a file using `FileWriter`.

Example - Writing data into a file

```
import java.io.*;  
  
public class WritingDemo {  
  
    public static void main(String[] args) throws IOException {  
  
        Writer out = new FileWriter("C:\\\\Raja\\\\dataFile.txt");  
  
        String msg = "The sample data";  
  
        try {  
  
            out.write(msg);  
  
            System.out.println("Writing done!!!");  
  
        }  
  
        catch(Exception e) {  
  
            System.out.println(e);  
        }  
    }  
}
```

```

        }

    finally {
        out.close();
    }

}

}

```

2.10 Reading and Writing Console:

Reading console input in java

In java, there are three ways to read console input. Using the 3 following ways, we can read input data from the console.

- Using BufferedReader class
- Using Scanner class
- Using Console class

Let's explore the each method to read data with example.

1. Reading console input using BufferedReader class in java

Reading input data using the BufferedReader class is the traditional technique. This way of the reading method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the console.

The BufferedReader class has defined in the java.io package.

Consider the following example code to understand how to read console input using BufferedReader class.

Example

```

import java.io.*;

public class ReadingDemo {

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String name = "";
        try {
            System.out.print("Please enter your name : ");

```

```
        name = in.readLine();

        System.out.println("Hello, " + name + "!");
    }

    catch(Exception e) {

        System.out.println(e);
    }

    finally {

        in.close();}}}
```

2. Reading console input using Scanner class in java

Reading input data using the Scanner class is the most commonly used method. This way of the reading method is used by wrapping the System.in (standard input stream) which is wrapped in a Scanner, we can read input from the console.

The Scanner class has defined in the java.util package.

Consider the following example code to understand how to read console input using Scanner class.

Example

```
import java.util.Scanner;

public class ReadingDemo {

    public static void main(String[] args) {

        String name = "";

        Scanner in=new Scanner(System.in);

        System.out.print("Please enter your name : ");

        name = in.next();

        System.out.println("Hello, " + name + "!");}}
```

3. Reading console input using Console class in java

Reading input data using the Console class is the most commonly used method. This class was introduced in Java 1.6 version.

The Console class has defined in the java.io package.

Consider the following example code to understand how to read console input using Console class.

Example

```
import java.io.*;

public class ReadingDemo {

    public static void main(String[] args) {

        String name;

        Console con = System.console();

        if(con != null) {

            name = con.readLine("Please enter your name : ");

            System.out.println("Hello, " + name + "!!");

        }

        else {

            System.out.println("Console not available.");

        }

    }

}
```

Writing console output in java

In java, there are two methods to write console output. Using the 2 following methods, we can write output data to the console.

- Using print() and println() methods
- Using write() method

Let's explore the each method to write data with example.

1. Writing console output using print() and println() methods

The PrintStream is a built-in class that provides two methods print() and println() to write console output. The print() and println() methods are the most widely used methods for console output.

Both print() and println() methods are used with System.out stream.

The print() method writes console output in the same line. This method can be used with console output only.

The `println()` method writes console output in a separate line (new line). This method can be used with console and also with other output sources.

Let's look at the following code to illustrate `print()` and `println()` methods.

Example

```
public class WritingDemo {  
    public static void main(String[] args) {  
        int[] list = new int[5];  
        for(int i = 0; i < 5; i++)  
            list[i] = i*10;  
        for(int i:list)  
            System.out.print(i);    //prints in same line  
        System.out.println("");  
        for(int i:list)  
            System.out.println(i); //Prints in separate lines  
    }  
}
```

2. Writing console output using `write()` method

Alternatively, the `PrintStream` class provides a method `write()` to write console output.

The `write()` method takes integer as argument, and writes its ASCII equivalent character on to the console, it also accept escape sequences.

Let's look at the following code to illustrate `write()` method.

Example

```
public class WritingDemo {  
    public static void main(String[] args) {  
        int[] list = new int[26];  
        for(int i = 0; i < 26; i++) {  
            list[i] = i + 65;  
        }  
    }  
}
```

```

for(int i:list) {

    System.out.write(i);

    System.out.write('\n');

}
}
}

```

2.11 File class:

The File is a built-in class in Java. In java, the File class has been defined in the java.io package. The File class represents a reference to a file or directory. The File class has various methods to perform operations like creating a file or directory, reading from a file, updating file content, and deleting a file or directory.

The File class in java has the following constructors.

S.No.	Constructor with Description
1	File(String pathname) It creates a new File instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.
2	File(String parent, String child) It creates a new File instance from a parent abstract pathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.
3	File(File parent, String child) It creates a new File instance from a parent abstract pathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.
4	File(URI url) It creates a new File instance by converting the given file: URI into an abstract pathname.

The File class in java has the following methods.

S.No.	Methods with Description
1	String getName() It returns the name of the file or directory that is referenced by the current File object.
2	String getParent() It returns the pathname of the pathname's parent, or null if the pathname does not name a parent directory.
3	String getPath() It returns the path of the current File.
4	File getParentFile() It returns the path of the current file's parent; or null if it does not exist.

Let's look at the following code to illustrate file operations.

Example

```
import java.io.*;  
  
public class FileClassTest {  
  
    public static void main(String args[]) {  
  
        File f = new File("C:\\Raja\\datFile.txt");  
  
        System.out.println("Executable File : " + f.canExecute());  
  
        System.out.println("Name of the file : " + f.getName());  
  
        System.out.println("Path of the file : " + f.getAbsolutePath());  
  
        System.out.println("Parent name : " + f.getParent());  
  
        System.out.println("Write mode : " + f.canWrite());  
  
        System.out.println("Read mode : " + f.canRead());  
  
        System.out.println("Existance : " + f.exists());  
  
        System.out.println("Last Modified : " + f.lastModified());  
  
        System.out.println("Length : " + f.length());  
  
        //f.createNewFile()  
  
        //f.delete();  
  
        //f.setReadOnly(){}  
    }  
}
```

Let's look at the following java code to list all the files in a directory including the files present in all its subdirectories.

Example

```
import java.util.Scanner;  
  
import java.io.*;  
  
public class ListingFiles {  
  
    public static void main(String[] args) {  
  
        String path = null;  
  
        Scanner read = new Scanner(System.in);  
  
        System.out.print("Enter the root directory name: ");  
    }  
}
```

```
path = read.next() + ":\\\";  
File f_ref = new File(path);  
if (!f_ref.exists()) {  
    printLine();  
    System.out.println("Root directory does not exists!");  
    printLine();  
} else {  
    String ch = "y";  
    while (ch.equalsIgnoreCase("y")) {  
        printFiles(path);  
        System.out.print("Do you want to open any sub-directory (Y/N):  
");  
        ch = read.next().toLowerCase();  
        if (ch.equalsIgnoreCase("y")) {  
            System.out.print("Enter the sub-directory name: ");  
            path = path + "\\\\" + read.next();  
            File f_ref_2 = new File(path);  
            if (!f_ref_2.exists()) {  
                printLine();  
                System.out.println("The sub-directory does not  
exists!");  
                printLine();  
                int lastIndex = path.lastIndexOf("\\");  
                path = path.substring(0, lastIndex);  
            }  
        }  
    }  
}
```

```
System.out.println("***** Program Closed *****");

}

public static void printFiles(String path) {

    System.out.println("Current Location: " + path);

    File f_ref = new File(path);

    File[] filesList = f_ref.listFiles();

    for (File file : filesList) {

        if (file.isFile())

            System.out.println("- " + file.getName());

        else

            System.out.println("> " + file.getName());

    }

}

public static void printLine() {

    System.out.println("-----");

}

}
```

In java, there are multiple ways to read data from a file and to write data to a file. The most commonly used ways are as follows.

- Using Byte Stream (FileInputStream and FileOutputStream)
- Using Character Stream (FileReader and FileWriter)

Let's look each of these ways.

File Handling using Byte Stream

In java, we can use a byte stream to handle files. The byte stream has the following built-in classes to perform various operations on a file.

FileInputStream - It is a built-in class in java that allows reading data from a file. This class has implemented based on the byte stream. The FileInputStream class provides a method read() to read data from a file byte by byte.

FileOutputStream - It is a built-in class in java that allows writing data to a file. This class has implemented based on the byte stream. The FileOutputStream class provides a method write() to write data to a file byte by byte.

Let's look at the following example program that reads data from a file and writes the same to another file using FileInputStream and FileOutputStream classes.

Example

```
import java.io.*;  
  
public class FileReadingTest {  
  
    public static void main(String args[]) throws IOException {  
  
        FileInputStream in = null;  
  
        FileOutputStream out = null;  
  
        try {  
  
            in = new FileInputStream("C:\\Raja\\Input-File.txt");  
  
            out = new FileOutputStream("C:\\Raja\\Output-File.txt");  
  
            int c;  
  
            while ((c = in.read()) != -1) {  
  
                out.write(c);  
  
            }  
  
            System.out.println("Reading and Writing has been success!!!");  
  
        }  
  
        catch(Exception e){  
  
            System.out.println(e);  
  
        }finally {  
  
            if (in != null) {  
  
                in.close();  
  
            }  
  
            if (out != null) {  
  
                out.close();  
  
            }  
        }  
    }  
}
```

```

    }
}

}
}

```

2.12 RandomAccessFile:

In java, the `java.io` package has a built-in class `RandomAccessFile` that enables a file to be accessed randomly. The `RandomAccessFile` class has several methods used to move the cursor position in a file.

A random access file behaves like a large array of bytes stored in a file.

RandomAccessFile Constructors

The `RandomAccessFile` class in java has the following constructors.

S.No.	Constructor with Description
1	<code>RandomAccessFile(File fileName, String mode)</code> It creates a random access file stream to read from, and optionally to write to, the file specified by the <code>File</code> argument.
2	<code>RandomAccessFile(String fileName, String mode)</code> It creates a random access file stream to read from, and optionally to write to, a file with the specified <code>fileName</code> .

Access Modes

Using the `RandomAccessFile`, a file may be created in the following modes.

`r` - Creates the file with read mode; Calling write methods will result in an `IOException`.

`rw` - Creates the file with read and write mode.

`rwd` - Creates the file with read and write mode - synchronously. All updates to file content is written to the disk synchronously.

`rws` - Creates the file with read and write mode - synchronously. All updates to file content or meta data is written to the disk synchronously.

RandomAccessFile methods

The RandomAccessFile class in java has the following methods.

S.No.	Methods with Description
1	int read() It reads byte of data from a file. The byte is returned as an integer in the range 0-255.
2	int read(byte[] b) It reads byte of data from file upto b.length, -1 if end of file is reached.
3	int read(byte[] b, int offset, int len) It reads bytes initialising from offset position upto b.length from the buffer.
4	boolean readBoolean() It reads a boolean value from from the file.
5	byte readByte() It reads signed eight-bit value from file.
6	char readChar() It reads a character value from file.
7	double readDouble() It reads a double value from file.
8	float readFloat() It reads a float value from file.
9	long readLong() It reads a long value from file.
10	int readInt() It reads a integer value from file.
11	void readFully(byte[] b) It reads bytes initialising from offset position upto b.length from the buffer.
12	void readFully(byte[] b, int offset, int len) It reads bytes initialising from offset position upto b.length from the buffer.

13	String readUTF()
	t reads in a string from the file.
14	void seek(long pos)
	It sets the file-pointer(cursor) measured from the beginning of the file, at which the next read or write occurs.
15	long length()
	It returns the length of the file.
16	void write(int b)
	It writes the specified byte to the file from the current cursor position.
17	void writeFloat(float v)
	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
18	void writeDouble(double v)
	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

Let's look at the following example program.

Example

```
import java.io.*;
public class RandomAccessFileDemo
{
    public static void main(String[] args)
    {
        try
        {
            double d = 1.5;
            float f = 14.56f;
            // Creating a new RandomAccessFile - "F2"
            RandomAccessFile f_ref = new RandomAccessFile("C:\\Raja\\Input-File.txt", "rw");
        }
    }
}
```

```
// Writing to file  
f_ref.writeUTF("Hello, Good Morning!");  
  
// File Pointer at index position - 0  
f_ref.seek(0);  
  
// read() method :  
  
System.out.println("Use of read() method : " + f_ref.read());  
f_ref.seek(0);  
  
byte[] b = {1, 2, 3};  
  
// Use of .read(byte[] b) method :  
  
System.out.println("Use of .read(byte[] b) : " + f_ref.read(b));  
  
// readBoolean() method :  
  
System.out.println("Use of readBoolean() : " + f_ref.readBoolean());  
  
// readByte() method :  
  
System.out.println("Use of readByte() : " + f_ref.readByte());  
f_ref.writeChar('c');  
  
f_ref.seek(0);  
  
// readChar() :  
  
System.out.println("Use of readChar() : " + f_ref.readChar());  
f_ref.seek(0);  
  
f_ref.writeDouble(d);  
f_ref.seek(0);  
  
// read double  
  
System.out.println("Use of readDouble() : " + f_ref.readDouble());  
f_ref.seek(0);  
  
f_ref.writeFloat(f);  
f_ref.seek(0);  
  
// readFloat() :
```

```
System.out.println("Use of readFloat() : " + f_ref.readFloat());
f_ref.seek(0);

// Create array upto geek.length

byte[] arr = new byte[(int) f_ref.length()];

// readFully():

f_ref.readFully(arr);

String str1 = new String(arr);

System.out.println("Use of readFully() : " + str1);

f_ref.seek(0);

// readFully(byte[] b, int off, int len):

f_ref.readFully(arr, 0, 8);

String str2 = new String(arr);

System.out.println("Use of readFully(byte[] b, int off, int len) : " + str2);

}

catch (IOException ex)

{

    System.out.println("Something went Wrong");

    ex.printStackTrace();

}

}

}
```

2.13 Console class:

In java, the `java.io` package has a built-in class `Console` used to read from and write to the console, if one exists. This class was added to the Java SE 6. The `Console` class implements the `Flushable` interface. In java, most the input functionalities of `Console` class available through `System.in`, and the output functionalities available through `System.out`.

Console class Constructors

The `Console` class does not have any constructor. We can obtain the `Console` class object by calling `System.console()`.

Console class methods

The Console class in java has the following methods.

S.No.	Methods with Description
1	void flush() It causes buffered output to be written physically to the console.
2	String readLine() It reads a string value from the keyboard, the input is terminated on pressing enter key.
3	String readLine(String promptingString, Object...args) It displays the given promptingString, and reads a string from the keyboard; input is terminated on pressing Enter key.
4	char[] readPassword() It reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
5	char[] readPassword(String promptingString, Object...args) It displays the given promptingString, and reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
6	Console printf(String str, Object....args) It writes the given string to the console.
7	Console format(String str, Object....args) It writes the given string to the console.
8	Reader reader() It returns a reference to a Reader connected to the console.
9	PrintWriter writer() It returns a reference to a Writer connected to the console.

Let's look at the following example program for reading a string using Console class.

Example

```
import java.io.*;
public class ReadingDemo {
    public static void main(String[] args) {
        String name;
        Console con = System.console();
        if(con != null) {
```

```

        name = con.readLine("Please enter your name : ");
        System.out.println("Hello, " + name + "!!");
    }
    else {
        System.out.println("Console not available.");
    }
}

```

Let's look at the following example program for writing to the console using Console class.

Example

```

import java.io.*;

public class WritingDemo {
    public static void main(String[] args) {
        int[] list = new int[26];
        for(int i = 0; i < 26; i++) {
            list[i] = i + 65;
        }
        for(int i:list) {
            System.out.write(i);
            System.out.write('\n');
        }
    }
}

```

2.14 Serialization:

In java, the Serialization is the process of converting an object into a byte stream so that it can be stored on to a file, or memory, or a database for future access. The deserialization is reverse of serialization. The deserialization is the process of reconstructing the object from the serialized state.

Using serialization and deserialization, we can transfer the Object Code from one Java Virtual machine to another.

Serialization in Java

In a java programming language, the Serialization is achieved with the help of interface Serializable. The class whose object needs to be serialized must implement the Serializable interface.

We use the ObjectOutputStream class to write a serialized object to write to a destination. The ObjectOutputStream class provides a method writeObject() to serializing an object.

We use the following steps to serialize an object.

Step 1 - Define the class whose object needs to be serialized; it must implement Serializable interface.

Step 2 - Create a file reference with file path using FileOutputStream class.

Step 3 - Create reference to ObjectOutputStream object with file reference.

Step 4 - Use writeObject(object) method by passing the object that wants to be serialized.

Step 5 - Close the FileOutputStream and ObjectOutputStream.

Let's look at the following example program for serializing an object.

Example

```
import java.io.*;  
  
public class SerializationExample {  
  
    public static void main(String[] args) {  
  
        Student stud = new Student();  
  
        stud.studName = "Rama";  
  
        stud.studBranch = "IT";  
  
        try {  
  
            FileOutputStream fos = new FileOutputStream("my_data.txt");  
  
            ObjectOutputStream oos = new ObjectOutputStream(fos);  
  
            oos.writeObject(stud);  
  
            oos.close();  
  
            fos.close();  
  
            System.out.println("The object has been saved to my_data file!");  
  
        }  
  
        catch(Exception e) {  
  
            System.out.println(e);  
        }  
    }  
}
```

```
    }  
}  
}
```

Deserialization in Java

In a java programming language, the Deserialization is achieved with the help of class `ObjectInputStream`. This class provides a method `readObject()` to deserializing an object.

We use the following steps to serialize an object.

Step 1 - Create a file reference with file path in which serialized object is available using `FileInputStream` class.

Step 2 - Create reference to `ObjectInputStream` object with file reference.

Step 3 - Use `readObject()` method to access serialized object, and typecaste it to destination type.

Step 4 - Close the `FileInputStream` and `ObjectInputStream`.

Let's look at the following example program for deserializing an object.

Example

```
import java.io.*;  
  
public class DeserializationExample {  
    public static void main(String[] args) throws Exception{  
        try {  
            FileInputStream fis = new FileInputStream("my_data.txt");  
            ObjectInputStream ois = new ObjectInputStream(fis);  
  
            Student stud2 = (Student) ois.readObject();  
            System.out.println("The object has been deserialized.");  
  
            fis.close();  
            ois.close();  
            System.out.println("Name = " + stud2.studName);  
            System.out.println("Department = " + stud2.studBranch);  
        }  
    }  
}
```

```

        }

    catch(Exception e) {

        System.out.println(e);

    }

}

```

2.15 Enumerations:

In java, an Enumeration is a list of named constants. The enum concept was introduced in Java SE 5 version. The enum in Java programming the concept of enumeration is greatly expanded with lot more new features compared to the other languages like C, and C++.

In java, the enumeration concept was defined based on the class concept. When we create an enum in java, it converts into a class type. This concept enables the java enum to have constructors, methods, and instance variables. All the constants of an enum are public, static, and final. As they are static, we can access directly using enum name. The main objective of enum is to define our own data types in Java, and they are said to be enumeration data types.

Creating enum in Java

To create enum in Java, we use the keyword enum. The syntax for creating enum is similar to that of class.

In java, an enum can be defined outside a class, inside a class, but not inside a method.

Let's look at the following example program for creating a basic enum.

Example

```

enum WeekDay{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY;
}

public class EnumerationExample {

    public static void main(String[] args) {

        WeekDay day = WeekDay.FRIDAY;

        System.out.println("Today is " + day);

        System.out.println("\nAll WeekDays: ");
}

```

```
for(WeekDay d:WeekDay.values())
    System.out.println(d);
}
```

- Every enum is converted to a class that extends the built-in class `Enum`.
- Every constant of an enum is defined as an object.
- As an enum represents a class, it can have methods, constructors. It also gets a few extra methods from the `Enum` class, and one of them is the `values()` method.

Constructors in Java enum

In a java programming language, an enum can have both the type of constructors default and parameterized. The execution of constructor depends on the constants we defined in the enum. For every constant in an enum, the constructor is executed.

If an enum has a parameterized constructor, the parameter value should be passed when we define constant.

Let's look at the following example program for illustrating constructors in enum.

Example

```
enum WeekDay{
    MONDAY, TUESDAY, WEDNESSDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY("Holiday");

    String msg;

    WeekDay(){
        System.out.println("Default constructor!");
    }

    WeekDay(String str){
        System.out.println("Parameterized constructor!");
        msg = str;
    }
}

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

```
WeekDay day = WeekDay.SUNDAY;  
System.out.println("\nToday is " + day + " and its " + day.msg);}}
```

Methods in Java enum

In a java programming language, an enum can have methods. These methods are similar to the methods of a class. All the methods defined in an enum can be used with all the constants defined by the same enum.

Let's look at the following example program for illustrating methods in enum.

Example

```
enum WeekDay{  
    MONDAY, TUESDAY, WEDNESSDAY, THURSDAY, FRIDAY, SATURDAY,  
    SUNDAY("Holiday");  
  
    String msg;  
  
    WeekDay(){  
        System.out.println("Default constructor!");  
    }  
  
    WeekDay(String str){  
        System.out.println("Parameterized constructor!");  
        msg = str;  
    }  
  
    void printMessage() {  
        System.out.println("Today is also " + msg);  
    }  
  
}  
  
public class EnumerationExample {  
    public static void main(String[] args) {  
        WeekDay day = WeekDay.SUNDAY;  
        System.out.println("\nToday is " + day);  
        day.printMessage();}}}
```

2.16 Auto Boxing :

In java, all the primitive data types have defined using the class concept, these classes known as wrapper classes. In java, every primitive type has its corresponding wrapper class. All the wrapper classes in Java were defined in the `java.lang` package.

The following table shows the primitive type and its corresponding wrapper class.

S.No.	Primitive Type	Wrapper class
1	Byte	Byte
2	Short	Short
3	Int	Integer
4	Long	Long
5	Float	Float
6	Double	Double
7	Char	Character
8	Boolean	Boolean

The Java 1.5 version introduced a concept that converts primitive type to corresponding wrapper type and reverses of it.

Autoboxing in Java

In java, the process of converting a primitive type value into its corresponding wrapper class object is called autoboxing or simply boxing. For example, converting an int value to an Integer class object.

The compiler automatically performs the autoboxing when a primitive type value has assigned to an object of the corresponding wrapper class.

Let's look at the following example program for autoboxing.

Example - Autoboxing

```
import java.lang.*;

public class AutoBoxingExample {

    public static void main(String[] args) {
        // Auto boxing : primitive to Wrapper
    }
}
```

```

        int num = 100;

        Integer i = num;

        Integer j = Integer.valueOf(num);

        System.out.println("num = " + num + ", i = " + i + ", j = " + j);

    }

}

```

Auto un-boxing in Java

In java, the process of converting an object of a wrapper class type to a primitive type value is called auto un-boxing or simply unboxing. For example, converting an Integer object to an int value.

The compiler automatically performs the auto un-boxing when a wrapper class object has assigned to a primitive type.

Let's look at the following example program for autoboxing.

Example - Auto unboxing

```

import java.lang.*;

public class AutoUnboxingExample {

    public static void main(String[] args) {

        // Auto un-boxing : Wrapper to primitive

        Integer num = 200;

        int i = num;

        int j = num.intValue();

        System.out.println("num = " + num + ", i = " + i + ", j = " + j);}}
```

2.17 Generics :

The java generics is a language feature that allows creating methods and class which can handle any type of data values. The generic programming is a way to write generalized programs, java supports it by java generics. The java generics is similar to the templates in the C++ programming language.

The java generics feature was introduced in Java 1.5 version. In java, generics used angular brackets "<>". In java, the generics feature implemented using the following.

- Generic Method
- Generic Class

Generic methods in Java

The java generics allows creating generic methods which can work with a different type of data values.

Using a generic method, we can create a single method that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Let's look at the following example program for generic method.

Example - Generic method

```
public class GenericFunctions {
    public <T, U> void displayData(T value1, U value2) {
        System.out.println("(" + value1.getClass().getName() + ", " + value2.getClass().getName() + ")");
    }

    public static void main(String[] args) {
        GenericFunctions obj = new GenericFunctions();
        obj.displayData(45.6f, 10);
        obj.displayData(10, 10);
        obj.displayData("Hi", 'c');
    }
}
```

In the above example code, the method `displayData()` is a generic method that allows a different type of parameter values for every function call.

Generic Class in Java

In java, a class can be defined as a generic class that allows creating a class that can work with different types.

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

Let's look at the following example program for generic class.

Example - Generic class

```
public class GenericsExample<T> {  
    T obj;  
    public GenericsExample(T anotherObj) {  
        this.obj = anotherObj;  
    }  
    public T getData() {  
        return this.obj;  
    }  
    public static void main(String[] args) {  
        GenericsExample<Integer> actualObj1 = new GenericsExample<Integer>(100);  
        System.out.println(actualObj1.getData());  
        GenericsExample<String> actualObj2 = new GenericsExample<String>("Java");  
        System.out.println(actualObj2.getData());  
        GenericsExample<Float> actualObj3 = new GenericsExample<Float>(25.9f);  
        System.out.println(actualObj3.getData());  
    }  
}
```

UNIT-III

Exception Handling

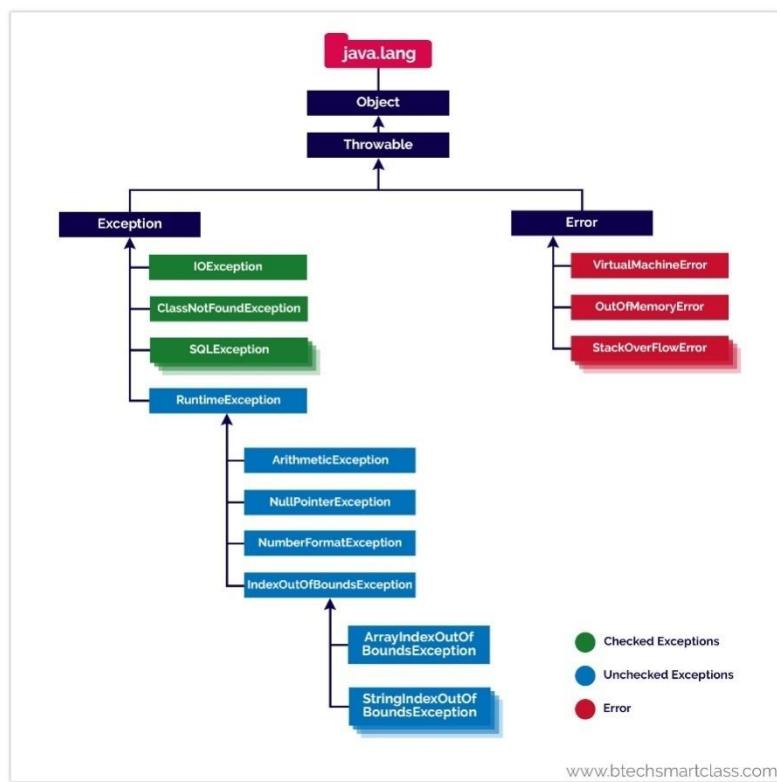
3.1) Fundamentals of Exception handling:

An exception in java programming is an abnormal situation that is araised during the program execution. In simple words, an exception is a problem that arises at the time of program execution.

When an exception occurs, it disrupts the program execution flow. When an exception occurs, the program execution gets terminated, and the system generates an error. We use the exception handling mechanism to avoid abnormal termination of program execution.

Java programming language has a very powerful and efficient exception handling mechanism with a large number of built-in classes to handle most of the exceptions automatically.

Java programming language has the following class hierarchy to support the exception handling mechanism.



Reasons for Exception Occurrence

Several reasons lead to the occurrence of an exception. A few of them are as follows.

- When we try to open a file that does not exist may lead to an exception.
- When the user enters invalid input data, it may lead to an exception.

- When a network connection has lost during the program execution may lead to an exception.
- When we try to access the memory beyond the allocated range may lead to an exception.
- The physical device problems may also lead to an exception.

3.2 Types of Exception

In java, exceptions have categorized into two types, and they are as follows.

- **Checked Exception** - An exception that is checked by the compiler at the time of compilation is called a checked exception.
- **Unchecked Exception** - An exception that can not be caught by the compiler but occurs at the time of program execution is called an unchecked exception.

How exceptions handled in Java?

In java, the exception handling mechanism uses five keywords namely *try*, *catch*, *finally*, *throw*, and *throws*.

In java, exceptions are mainly categorized into two types, and they are as follows.

- **Checked Exceptions**
- **Unchecked Exceptions**

Checked Exceptions

The checked exception is an exception that is checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If it is not handled, the compiler displays a compilation error using built-in classes.

The checked exceptions are generally caused by faults outside of the code itself like missing resources, networking errors, and problems with threads come to mind.

The following are a few built-in classes used to handle checked exceptions in java.

- IOException
- FileNotFoundException
- ClassNotFoundException
- SQLException
- DataAccessException
- InstantiationException
- UnknownHostException

Let's look at the following example program for the checked exception method.

Example - Checked Exceptions

```
import java.io.*;  
  
public class CheckedExceptions  
{  
    public static void main(String[] args)  
    {  
        File f_ref = new File("C:\\Users\\User\\Desktop\\Today\\Sample.txt");  
        try  
        {  
            FileReader fr = new FileReader(f_ref);  
        }  
        catch(Exception e)  
        {  
            System.out.println(e);  
        }  
    }  
}
```

Unchecked Exceptions

The unchecked exception is an exception that occurs at the time of program execution. The unchecked exceptions are not caught by the compiler at the time of compilation.

The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.

The following are a few built-in classes used to handle unchecked exceptions in java.

- `ArithmaticException`
- `NullPointerException`
- `NumberFormatException`
- `ArrayIndexOutOfBoundsException`
- `StringIndexOutOfBoundsException`

The unchecked exception is also known as a runtime exception.

Let's look at the following example program for the unchecked exceptions.

Example - Unchecked Exceptions

```
public class UncheckedException
{
    public static void main(String[] args)
    {
        int list[] = {10, 20, 30, 40, 50};

        System.out.println(list[6]); //ArrayIndexOutOfBoundsException

        String msg=null;

        System.out.println(msg.length()); //NullPointerException

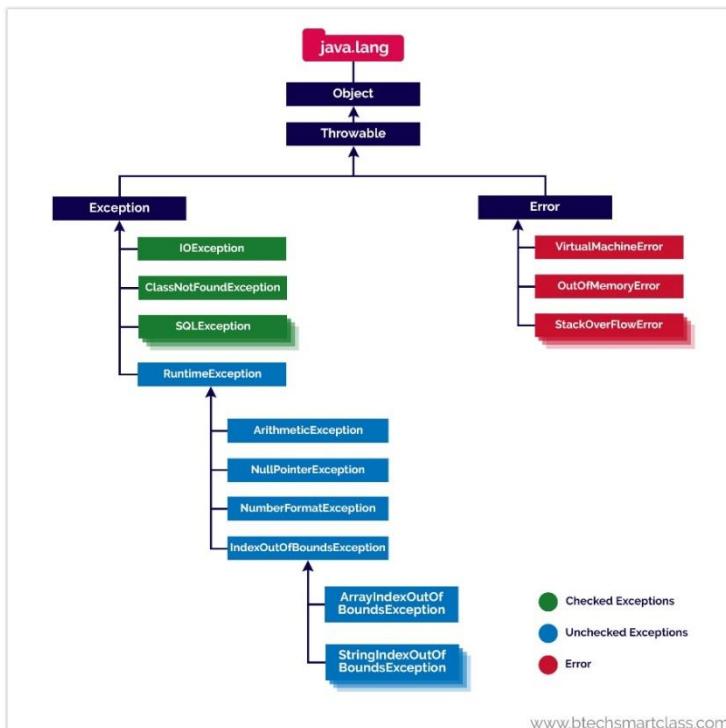
        String name="abc";

        int i=Integer.parseInt(name); //NumberFormatException

    }
}
```

Exception class hierarchy:

In java, the built-in classes used to handle exceptions have the following class hierarchy.



3.3) Exception Models in Java:

In java, there are two exception models. Java programming language has two models of exception handling. The exception models that java supports are as follows.

- **Termination Model**
- **Resumptive Model**

Let's look into details of each exception model.

Termination Model

In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.

In other words we can say that in termination model the error is so critical there is no way to get back to where the exception occurred.

Resumptive Model

The alternative of termination model is resumptive model. In resumptive model, the exception handler is expected to do something to stabilize the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.

In resumptive model we may use a method call that wants resumption-like behavior. We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.

3.4) Uncaught Exceptions in Java:

In java, assume that, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints an exception message with the help of uncaught exception handler.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Java programming language has a very strong exception handling mechanism. It allows us to handle the exception using the keywords like try, catch, finally, throw, and throws.

When an uncaught exception occurs, the JVM calls a special private method known as **dispatchUncaughtException()**, on the Thread class in which the exception occurs and terminates the thread.

The Division by zero exception is one of the examples for uncaught exceptions. Look at the following code.

Example

```
import java.util.Scanner;

public class UncaughtExceptionExample

{
    public static void main(String[] args)

    {
        Scanner read = new Scanner(System.in);

        System.out.println("Enter the a and b values: ");

        int a = read.nextInt();

        int b = read.nextInt();

        int c = a / b;

        System.out.println(a + "/" + b + " = " + c);

    }
}
```

3.5 try and catch in Java:

In java, the **try** and **catch**, both are the keywords used for exception handling.

The keyword **try** is used to define a block of code that will be tested for the occurrence of an exception. The keyword **catch** is used to define a block of code that handles the exception occurred in the respective **try** block.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Both **try** and **catch** are used as a pair. Every **try** block must have one or more **catch** blocks. We can not use **try** without at least one **catch**, and **catch** alone can be used (**catch** without **try** is not allowed).

The following is the syntax of try and catch blocks.

Syntax

```
try{ ...  
  code to be tested ...}  
catch(ExceptionType object)  
{ ...  
  code for handling the exception ...}
```

Consider the following example code to illustrate try and catch blocks in Java.

Example

```
import java.util.Scanner;  
  
public class TryCatchExample  
{  
  public static void main(String[] args)  
  {  
    Scanner read = new Scanner(System.in);  
    System.out.println("Enter the a and b values: ");  
    try  
    {  
      int a = read.nextInt();  
      int b = read.nextInt();  
      int c = a / b;  
      System.out.println(a + "/" + b + " = " + c);  
    }  
    catch(ArithmeticException ae)  
    {  
      System.out.println("Problem info: Value of divisor can not be ZERO");  
    }  
}
```

3.6 Multiple catch clauses:

In java programming language, a try block may have one or more number of catch blocks. That means a single try statement can have multiple catch clauses.

When a try block has more than one catch block, each catch block must contain a different exception type to be handled.

The multiple catch clauses are defined when the try block contains the code that may lead to different type of exceptions.

Let's look at the following example Java code to illustrate multiple catch clauses.

Example

```
public class TryCatchExample
{
    public static void main(String[] args)
    {
        try
        {
            int list[] = new int[5];
            list[2] = 10;  list[4] = 2;  list[10] = list[2] / list[4];
        }
        catch(ArithmaticException ae)
        {
            System.out.println("Problem info: Value of divisor can not be ZERO.");
        }
        catch(ArrayIndexOutOfBoundsException aie)
        {
            System.out.println("Problem info: ArrayIndexOutOfBoundsException has occurred.");
        }
        catch(Exception e)
        {
    }
```

```
System.out.println("Problem info: Unknown exception has occured.");
}
}}
```

3.7 Nested try statements:

The java allows to write a try statement inside another try statement. A try block within another try block is known as nested try block.

When there are nested try blocks, each try block must have one or more separate catch blocks.

Let's look at the following example Java code to illustrate nested try statements.

Example

```
public class TryCatchExample
{
    public static void main(String[] args)
    {
        try
        {
            int list[] = new int[5];
            list[2] = 10;
            list[4] = 2;
            list[0] = list[2] / list[4];
            try {
                list[10] = 100;
            }
            catch(ArrayIndexOutOfBoundsException aie)
            {
                System.out.println("Problem info: ArrayIndexOutOfBoundsException has occured.");
            }
        }
    }
}
```

```
        catch(ArithmeticException ae
        ) {
            System.out.println("Problem info: Value of divisor can not be ZERO.");
        }
        catch(Exception e)
        {
            System.out.println("Problem info: Unknown exception has occurred.");
        }
    }
}
```

3.8 throw, throws, and finally keywords in Java:

In java, the keywords throw, throws, and finally are used in the exception handling concept. Let's look at each of these keywords.

throw keyword in Java

The throw keyword is used to throw an exception instance explicitly from a try block to corresponding catch block. That means it is used to transfer the control from try block to corresponding catch block.

The throw keyword must be used inside the try block. When JVM encounters the throw keyword, it stops the execution of try block and jump to the corresponding catch block.

The following is the general syntax for using throw keyword in a try block.

Syntax : throw instance;

Here the instance must be throwable instance and it can be created dynamically using new operator.

Let's look at the following example Java code to illustrate throw keyword.

Example

```
import java.util.Scanner;
public class Sample
{
    public static void main(String[] args)
```

```
{  
Scanner input = new Scanner(System.in);  
int num1, num2, result;  
System.out.print("Enter any two numbers: ");  
num1 = input.nextInt();  
num2 = input.nextInt();  
try  
{  
if(num2 == 0)  
throw new ArithmeticException("Division by zero is not possible");  
result = num1 / num2;  
System.out.println(num1 + "/" + num2 + "=" + result);  
}  
catch(ArithmaticException ae)  
{  
System.out.println("Problem info: " + ae.getMessage());  
}  
System.out.println("End of the program");  
}  
}
```

throws keyword in Java

The throws keyword specifies the exceptions that a method can throw to the default handler and does not handle itself. That means when we need a method to throw an exception automatically, we use throws keyword followed by method declaration.

Let's look at the following example Java code to illustrate throws keyword.

Example

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

```
int num1, num2, result;  
Scanner input = new Scanner(System.in);  
void division() throws ArithmeticException  
{  
    System.out.print("Enter any two numbers: ");  
    num1 = input.nextInt();  
    num2 = input.nextInt();  
    result = num1 / num2;  
    System.out.println(num1 + "/" + num2 + "=" + result);  
}  
  
public static void main(String[] args){  
    try  
    {  
        new ThrowsExample().division();  
    }  
    catch(ArithmeticException ae)  
    {  
        System.out.println("Problem info: " + ae.getMessage());  
    }  
    System.out.println("End of the program");  
}
```

finally keyword in Java

The finally keyword used to define a block that must be executed irrespective of exception occurrence. The basic purpose of finally keyword is to cleanup resources allocated by try block, such as closing file, closing database connection, etc.

Let's look at the following example Java code to illustrate throws keyword.

Example

```
import java.util.Scanner;

public class FinallyExample

{

public static void main(String[] args)

{

int num1, num2, result;Scanner input = new Scanner(System.in);

System.out.print("Enter any two numbers: ");

num1 = input.nextInt();

num2 = input.nextInt();

try

{

if(num2 == 0)

throw new ArithmeticException("Division by zero");

result = num1 / num2;

System.out.println(num1 + "/" + num2 + "=" + result);

}

catch(ArithmeticException ae)

{

System.out.println("Problem info: " + ae.getMessage());

}

finally

{

System.out.println("The finally block executes always");

}

System.out.println("End of the program");

}
```

1

3.9 Built-in Exceptions in Java:

The Java programming language has several built-in exception class that support exception handling. Every exception class is suitable to explain certain error situations at run time.

All the built-in exception classes in Java were defined a package **java.lang**.

List of checked exceptions in Java

The following table shows the list of several checked exceptions.

S. No.	Exception Class with Description
1	ClassNotFoundException It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.
2	CloneNotSupportedException Used to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.
3	IllegalAccessException It is thrown when one attempts to access a method or member that visibility qualifiers do not allow.
4	InstantiationException It is thrown when an application tries to create an instance of a class using the newInstance method in class Class , but the specified class object cannot be instantiated because it is an interface or is an abstract class.
5	InterruptedException It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.
6	NoSuchFieldException It indicates that the class doesn't have a field of a specified name.
7	NoSuchMethodException It is thrown when some JAR file has a different version at runtime than it had at compile time, a NoSuchMethodException occurs during reflection when we try to access a method that does not exist.

List of un checked exceptions in Java:

a) ArithmeticException:

It handles the arithmetic exceptions like division by zero

b) **ArrayIndexOutOfBoundsException:**

It handles the situations like 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.

c)ArrayStoreException:

It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects

d) Assertion Error:

It is used to indicate that an assertion has failed.

e)ClassCastException:

It handles the situation when we try to improperly cast a class from one type to another.

f)IllegalArgumentException

This exception is thrown in order to indicate that a method has been passed an illegal or inappropriate argument.

g) illegalMonitorStateException:

This indicates that the calling thread has attempted to wait on an object's monitor, or has attempted to notify other threads that wait on an object's monitor, without owning the specified monitor.

h) IllegalStateException

It signals that a method has been invoked at an illegal or inappropriate time.

i) IllegalThreadStateException:

It is thrown by the Java runtime environment, when the programmer is trying to modify the state of the thread when it is illegal.

j) IndexOutOfBoundsException

It is thrown when attempting to access an invalid index within a collection, such as an array , vector , string , and so forth.

k)NegativeArraySizeException:

It is thrown if an applet tries to create an array with negative size.

l)NullPointerException:

it is thrown when program attempts to use an object reference that has the null value.

m) SecurityException:

It is thrown by the Java Card Virtual Machine to indicate a security violation.

n)StringIndexOutOfBoundsException:

It is thrown by the methods of the String class, in order to indicate that an index is either negative, or greater than the size of the string itself.

3.10 Creating Own Exceptions in Java:

The Java programming language allow us to create our own exception classes which are basically subclasses built-in class **Exception**.

To create our own exception class simply create a class as a subclass of built-in Exception class.

We may create constructor in the user-defined exception class and pass a string to Exception class constructor using **super()**. We can use **getMessage()** method to access the string.

Let's look at the following Java code that illustrates the creation of user-defined exception.

Example

```
import java.util.Scanner;class NotEligibleException extends Exception
{
    NotEligibleException(String msg)
    {
        super(msg);
    }
    class VoterList
    {
        int age;
        VoterList(int age)
        {
            this.age = age;
        }
        void checkEligibility()
        {
            try
```

```
{  
if(age < 18)  
{  
throw new NotEligibleException("Error: Not eligible for vote due to under age.");  
}  
System.out.println("Congrates! You are eligible for vote.");  
}  
catch(NotEligibleException nee)  
{  
System.out.println(nee.getMessage());  
}  
}  
}  
public static void main(String args[])  
{Scanner input = new Scanner(System.in);  
System.out.println("Enter your age in years: ");  
int age = input.nextInt();  
VoterList person = new VoterList(age);  
person.checkEligibility(); }
```

Multithreading in java

Thread:

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light weight process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

3.11 Difference between Thread based multithreading and Process based multithreading:

In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking. There are two types of multitasking, and they are as follows.

- Process-based multitasking
- Thread-based multitasking

It is important to know the difference between process-based and thread-based multitasking. Let's distinguish both.

Process-based multitasking	Thread-based multitasking
It allows the computer to run two or more programs concurrently	It allows the computer to run two or more threads concurrently
In this process is the smallest unit.	In this thread is the smallest unit.
Process is a larger unit.	Thread is a part of process.
Process is heavy weight.	Thread is light weight.
Process requires separate address space for each.	Threads share same address space.
Process never gain access over idle time of CPU.	Thread gain access over idle time of CPU.
Inter process communication is expensive.	Inter thread communication is not expensive.

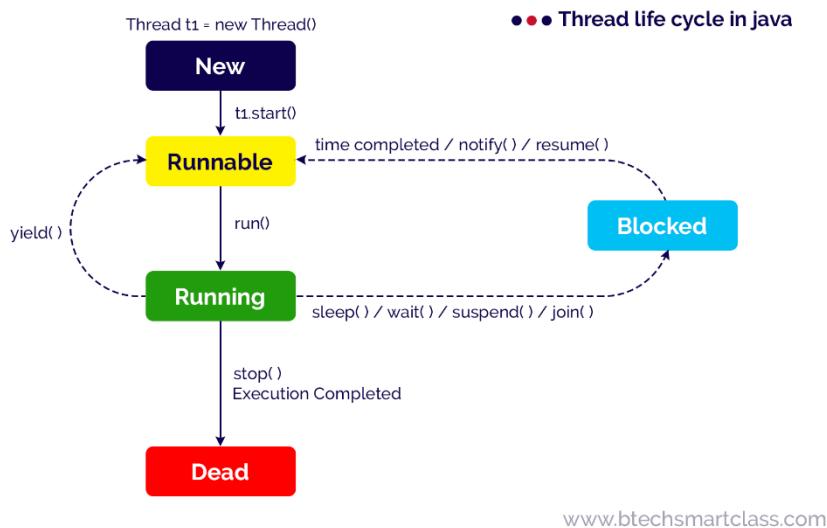
The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light weight process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may be in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



New

When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

Example

Thread t1 = new Thread(); **Runnable / Ready**

When a thread calls start() method, then the thread is said to be in the Runnable state. This state is also known as a Ready state. **Running**

When a thread calls run() method, then the thread is said to be Running. The run() method of a thread called automatically by the start() method.

Blocked / Waiting

A thread in the Running state may move into the blocked state due to various reasons like sleep() method called, wait() method called, suspend() method called, and join() method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify() or notifyAll() method called, resume() method called, etc.

Example

Thread.sleep(1000);wait(1000);wait();suspended();notify();notifyAll();resume();

Terminated:

A thread in the Running state may move into the dead state due to either its execution completed or the stop() method called. The dead state is also known as the terminated state.

3.12 Creating threads:

In java, a thread is a lightweight process. Every java program executes by a thread called the main thread. When a java program gets executed, the main thread created automatically. All other threads called from the main thread.

The java programming language provides two methods to create threads, and they are listed below.

- **Using Thread class (by extending Thread class)**
- **Using Runnable interface (by implementing Runnable interface)**

Let's see how to create threads using each of the above.

Extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

- **Step-1:** Create a class as a child of Thread class. That means, create a class that extends Thread class.
- **Step-2:** Override the run() method with the code that is to be executed by the thread. The run() method must be public while overriding.
- **Step-3:** Create the object of the newly created class in the main() method.
- **Step-4:** Call the start() method on the object created in the above step.

Look at the following example program.

Example

```
class SampleThread extends Thread
{
    public void run()
    {
        System.out.println("Thread is under Running...");
        for(int i= 1; i<=10; i++)
        {
            System.out.println("i = " + i);
        }
    }
}
```

```
}

public class My_Thread_Test

{

public static void main(String[] args)

{

SampleThread t1 = new SampleThread();

System.out.println("Thread about to start...");

t1.start();

}

}
```

When we run this code, it produce the following output.

Implementng Runnable interface:

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below.

- **Step-1:** Create a class that implements Runnable interface.
- **Step-2:** Override the run() method with the code that is to be executed by the thread. The run() method must be public while overriding.
- **Step-3:** Create the object of the newly created class in the main() method.
- **Step-4:** Create the Thread class object by passing above created object as parameter to the Thread class constructor.
- **Step-5:** Call the start() method on the Thread class object created in the above step.

Look at the following example program.

Example

```
class SampleThread implements Runnable

{

public void run()

{

System.out.println("Thread is under Running...");
```

```
for(int i= 1; i<=10; i++)  
{  
    System.out.println("i = " + i);  
}  
}  
}  
}  
  
public class My_Thread_Test  
{  
    public static void main(String[] args)  
    {  
        SampleThread threadObject = new SampleThread();  
        Thread thread = new Thread(threadObject);  
        System.out.println("Thread about to start...");  
        thread.start();  
    }  
}
```

More about Thread class

The Thread class in java is a subclass of Object class and it implements Runnable interface. The Thread class is available inside the java.lang package. The Thread class has the following syntax.

class Thread extends Object implements Runnable{...}The Thread class has the following constructors.

- **Thread()**
- **Thread(String threadName)**
- **Thread(Runnable objectName)**
- **Thread(Runnable objectName, String threadName)**

The Thread classs contains the following methods.

Method	Description	Return Value
run()	Defines actual task of the thread.	Void
start()	It moves thre thread from Ready state to Running state by calling run() method.	Void
setName(String)	Assigns a name to the thread.	Void
getName()	Returns the name of the thread.	String
setPriority(int)	Assigns priority to the thread.	Void
getPriority()	Returns the priority of the thread.	Int
getId()	Returns the ID of the thread.	Long
activeCount()	Returns total number of thread under active.	Int
currentThread()	Returns the reference of the thread that currently in running state.	Void
sleep(long)	moves the thread to blocked state till the specified number of milliseconds.	void
isAlive()	Tests if the thread is alive.	Boolean
yield()	Tells to the scheduler that the current thread is willing to yield its current use of a processor.	Void
join()	Waits for the thread to end.	Void

3.13 Thread Priority:

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority. The java programming language Thread class provides two methods **setPriority(int)**, and **getPriority()** to handle thread priorities. The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

- **MAX_PRIORITY** - It has the value 10 and indicates highest priority.
- **NORM_PRIORITY** - It has the value 5 and indicates normal priority.
- **MIN_PRIORITY** - It has the value 1 and indicates lowest priority.

setPriority() method

The setPriority() method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

The regular use of the setPriority() method is as follows.

Example

```
threadObject.setPriority(4);or threadObject.setPriority(MAX_PRIORITY);getPriority( ) method
```

The getPriority() method of Thread class used to access the priority of a thread. It does not takes any argument and returns name of the thread as String.

The regular use of the getPriority() method is as follows.

Example

String threadName = threadObject.getPriority();**Look at the following example program.**

Example

```
class SampleThread extends Thread
{
    public void run()
    {
        System.out.println("InsideSampleThread");
        System.out.println("Current Thread: " + Thread.currentThread().getName());
    }
}

public class My_Thread_Test
{
    public static void main(String[] args)
    {
        SampleThreadad threadObject1 = new SampleThread();
        SampleThread.threadObject2 = new SampleThread();
        threadObject1.setName("first");
```

```
threadObject2.setName("second");

threadObject1.setPriority(4);

threadObject2.setPriority(Thread.MAX_PRIORITY);

threadObject1.start();

threadObject2.start();

}

}
```

3.14 Java Thread Synchronization:

The java programming language supports multithreading. The problem of shared resources occurs when two or more threads get execute at the same time. In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.

The synchronization is the process of allowing only one thread to access a shared resource at a time. In java, the synchronization is achieved using the following concepts.

- Mutual Exclusion
- Inter thread communication

Mutual Exclusion

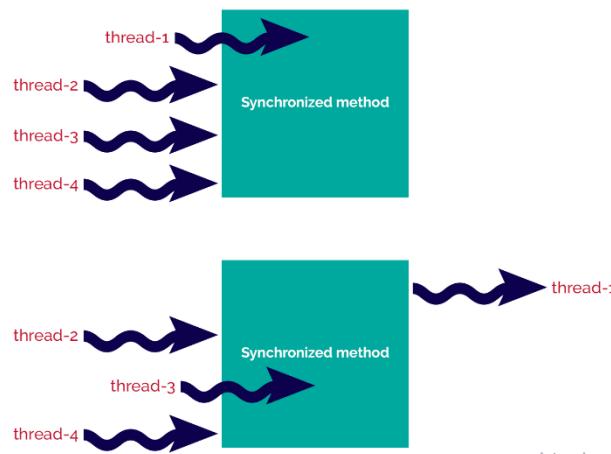
Using the mutual exclusion process, we keep threads from interfering with one another while they accessing the shared resource. In java, mutual exclusion is achieved using the following concepts.

- Synchronized method
- Synchronized block

Synchronized method

When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.

● ● ● Java thread execution with synchronized method



www.btechsmartclass.com

In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method). When thread-1 completes its task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example

```
class Table
{
    synchronized void printTable(int n)
    {
        for(int i = 1; i <= 10; i++)
            System.out.println(n + " * " + i + " = " + i*n);
    }
}

class MyThread_1 extends Thread
{
    Table table = new Table();
    int number; MyThread_1(Table table, int number)
    {
        this.table = table; this.number = number;
    }
    public void run()
    {
        for(int i = 1; i <= 10; i++)
            System.out.println(number + " * " + i + " = " + number*i);
    }
}
```

```
{  
table.printTable(number);  
}  
}  
  
class MyThread_2 extends Thread  
{  
  
Table table = new Table();  
  
int number;  
  
MyThread2(Table table, int number)  
{  
  
this.table = table;  
  
this.number = number;  
}  
  
public void run()  
{  
  
table.printTable(number);  
}  
}  
}  
}  
  
public class ThreadSynchronizationExample  
{  
  
public static void main(String[] args)  
{  
  
Table table = new Table();  
  
MyThread_1 thread_1 = new MyThread_1(table, 5);  
  
MyThread_2 thread_2 = new MyThread_2(table, 10);  
  
thread_1.start();
```

```
thread_2.start();  
}  
}
```

Synchronized block

The synchronized block is used when we want to synchronize only a specific sequence of lines in a method. For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.

The following syntax is used to define a synchronized block.

Syntax:

```
synchronized(object)  
{ ... block code ... }
```

Look at the following example code to illustrate synchronized block.

Example

```
import java.util.*;class NameList  
{  
    String name = "";  
    public int count = 0;  
    public void addName(String name,List<String> namesList)  
    {  
        synchronized(this)  
        {  
            this.name = name;  
            count++;  
        }  
        namesList.add(name);  
    }  
}
```

```
public int getCount()
{
    return count;
}

public class SynchronizedBlockExample
{
    public static void main (String[] args)
    {
        NameList namesList_1 = new NameList();
        NameList namesList_2 = new NameList();
        List<String> list = new ArrayList<String>();
        namesList_1.addName("Rama", list);
        namesList_2.addName("Seetha", list);
        System.out.println("Thread1: " + namesList_1.name + ", " + namesList_1.getCount() + "\n");
        System.out.println("Thread2: " + namesList_2.name + ", " + namesList_2.getCount() + "\n");
    }
}
```

3.15 Java Inter Thread Communication:

Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**. In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true. That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task. The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- `wait()`
- `notify()`
- `notifyAll()`

The following table gives detailed description about the above methods.

Method	Description
void wait()	It makes the current thread to pause its execution until other thread in the same monitor calls notify()
void notify()	It wakes up the thread that called wait() on the same object.
void notifyAll()	It wakes up all the threads that called wait() on the same object.
.	.

Let's look at an example problem of producer and consumer. The producer produces the item and the consumer consumes the same. But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced. So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same. Here we use the inter-thread communication to implement the producer and consumer problem.

The sample implementation of producer and consumer problem is as follows.

Example:

```
class ItemQueue
{
    int item;
    boolean valueSet = false;
    synchronized int getItem()
    {
        while (!valueSet)
            try
            {
                {wait();}
            }
            catch(InterruptedException e)
            {
                System.out.println("InterruptedException caught");
            }
    }
}
```

```
System.out.println("Consummed:" + item);
valueSet = false;
try
{
    Thread.sleep(1000);
}
catch(InterruptedException e)
{
    System.out.println("InterruptedException caught");
}
notify();
return item;
}

synchronized void putItem(int item)
{
    while (valueSet)
        try { wait(); }
    catch (InterruptedException e)
    {
        System.out.println("InterruptedExceptioncaught");
    }
    this.item = item;
    valueSet = true;
    System.out.println("Produced: " + item);
}
```

```
Thread.sleep(1000);

}

catch (InterruptedException e)

{

System.out.println("InterruptedException caught");

}

notify();

}

}

class Producer implements Runnable

{

ItemQueue itemQueue;Producer(ItemQueue itemQueue)

{

this.itemQueue = itemQueue;

new Thread(this, "Producer").start();

}

public void run()

{

int i = 0;

while(true)

{

itemQueue.putItem(i++);

}

}

}

class Consumer implements Runnable

{



}
```

```
ItemQueue itemQueue;  
  
Consumer(ItemQueue itemQueue)  
{  
    this.itemQueue = itemQueue;  
    new Thread(this, "Consumer").start();  
}  
  
public void run()  
{  
    while(true)  
    {  
        itemQueue.getItem();  
    }  
}  
  
}  
  
class ProducerConsumer  
{  
    public static void main(String args[])  
    {  
        ItemQueue itemQueue = new ItemQueue();  
        new Producer(itemQueue);  
        new Consumer(itemQueue);  
    }  
}
```

UNIT – IV

THE COLLECTIONS FRAMEWORK

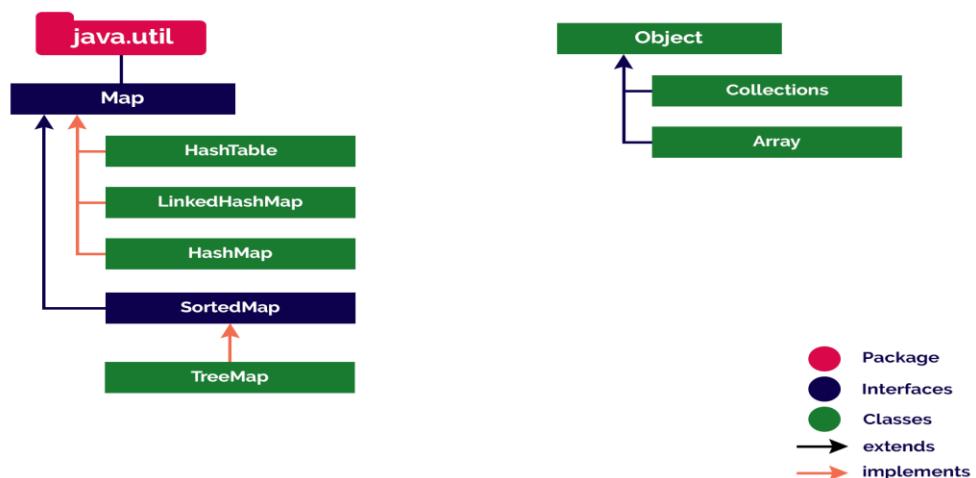
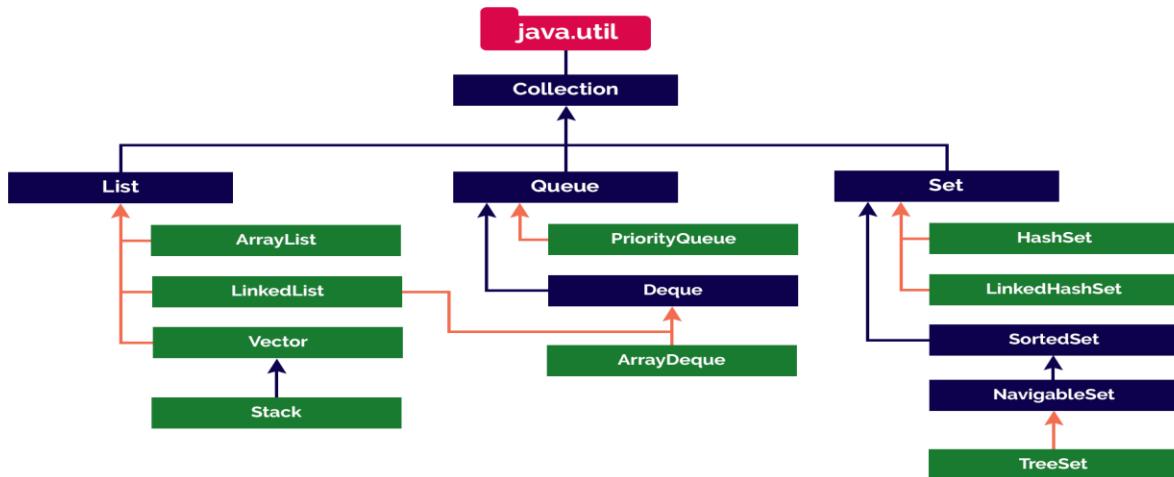
4.1 Collections overview:

Java collection framework is a collection of interfaces and classes used to storing and processing a group of individual objects as a single unit. The java collection framework holds several classes that provide a large number of methods to store and process a group of objects. These classes make the programmer task super easy and fast.

Java collection framework was introduced in java 1.2 version.

Java collection framework has the following hierarchy.

● ● ● Java Collection Framework



- Package
- Interfaces
- Classes
- extends
- implements

Before the collection framework in java (before java 1.2 version), there was a set of classes like **Array**, **Vector**, **Stack**, **HashTable**. These classes are known as **legacy classes**.

The java collection framework contains List, Queue, Set, and Map as top-level interfaces. The List, Queue, and Set stores single value as its element, whereas Map stores a pair of a key and value as its element.

4.2 Collection Interfaces:

The Collection interface is the root interface for most of the interfaces and classes of collection framework. The Collection interface is available inside the `java.util` package. It defines the methods that are commonly used by almost all the collections.

The Collection interface defines the following methods.

••• Methods of **Collection** interface in java



The Collection interface extends Iterable interface.

Let's consider an example program on ArrayList to illustrate the methods of Collection interface.

Example

```
import java.util.*;  
  
public class CollectionInterfaceExample {  
  
    public static void main(String[] args) {  
  
        List list_1 = new ArrayList();  
  
        List<String> list_2 = new ArrayList<String>();  
  
        list_1.add(10);  
  
        list_1.add(20);  
  
        list_2.add("BTech");  
  
        list_2.add("Smart");  
  
        list_2.add("Class");  
  
        list_1.addAll(list_2);  
  
        System.out.println("Elements of list_1: " + list_1);  
  
        System.out.println("Search for BTech: " + list_1.contains("BTech"));  
  
        System.out.println("Search for list_2 in list_1: " + list_1.containsAll(list_2));  
  
        System.out.println("Check whether list_1 and list_2 are equal: " +  
list_1.equals(list_2));  
  
        System.out.println("Check is list_1 empty: " + list_1.isEmpty());  
  
        System.out.println("Size of list_1: " + list_1.size());  
  
        System.out.println("Hashcode of list_1: " + list_1.hashCode());  
  
        list_1.remove(0);  
  
        System.out.println(list_1);  
  
        list_1.retainAll(list_2);  
  
        System.out.println(list_1);  
  
        list_1.removeAll(list_2);  
    }  
}
```

```

        System.out.println(list_1);

        list_2.clear();

        System.out.println(list_2);

    }

}

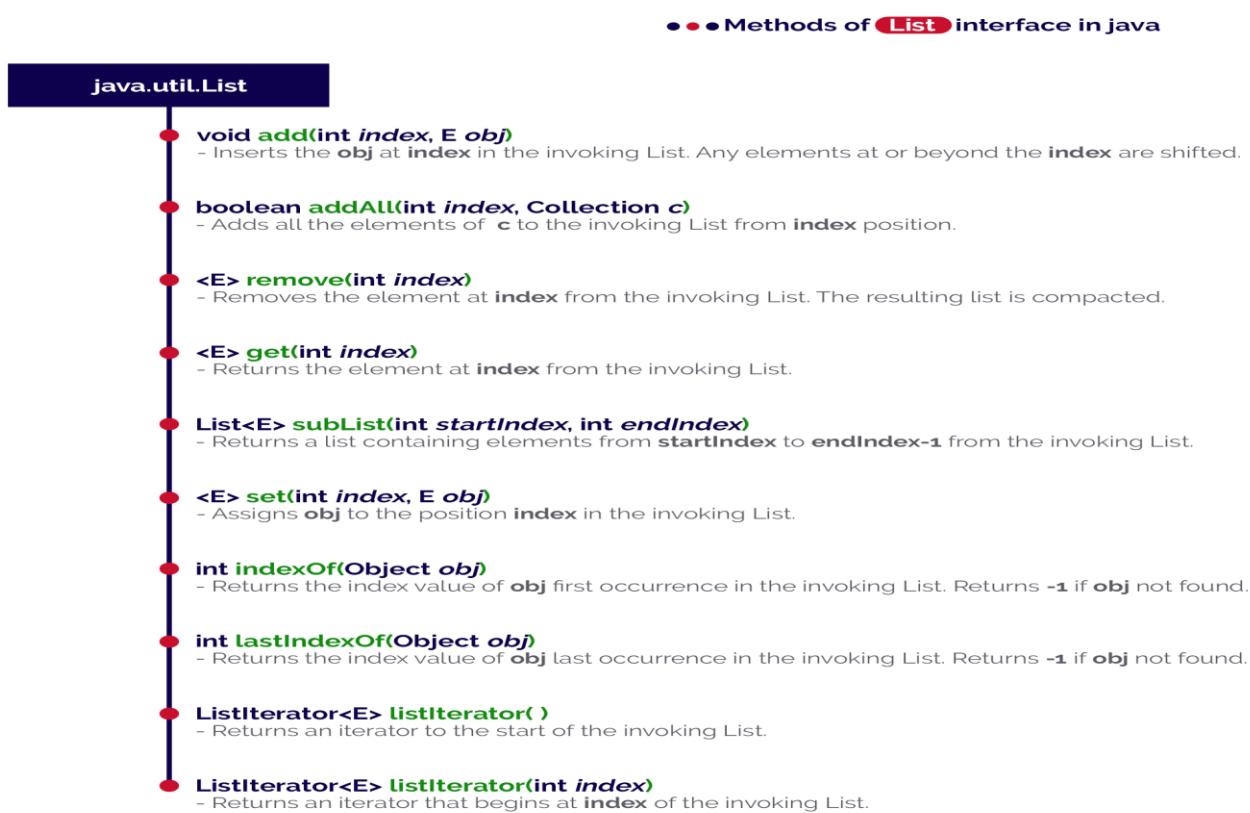
```

List interface

The List interface is a child interface of the Collection interface. The List interface is available inside the `java.util` package. It defines the methods that are commonly used by classes like `ArrayList`, `LinkedList`, `Vector`, and `Stack`.

- The List interface extends Collection interface.
- The List interface allows duplicate elements.
- The List interface preserves the order of insertion.
- The List allows to access the elements based on the index value that starts with zero.

The List interface defines the following methods.



Let's consider an example program on ArrayList to illustrate the methods of List interface.

Example

```
import java.util.ArrayList;  
import java.util.List;  
  
public class ListInterfaceExample {  
  
    public static void main(String[] args) {  
  
        List list_1 = new ArrayList();  
  
        List<String> list_2 = new ArrayList<String>();  
  
        list_1.add(0, 10);  
  
        list_1.add(1, 20);  
  
        list_2.add(0, "BTech");  
  
        list_2.add(1, "Smart");  
  
        list_2.add(2, "Class");  
  
        list_1.addAll(1, list_2);  
  
        System.out.println("\nElements of list_1: " + list_1);  
  
        System.out.println("\nElement at index 3: " + list_1.get(3));  
  
        System.out.println("\nSublist : " + list_1.subList(2, 5));  
  
        list_1.set(2, 10);  
  
        System.out.println("\nAfter updating the value at index 2: " + list_1);  
  
        System.out.println("\nIndex of value 10: " + list_1.indexOf(10));  
  
        System.out.println("\nLast index of value 10: " + list_1.lastIndexOf(10));  
  
    }  
}
```

Queue interface

The Queue interface is a child interface of the Collection interface. The Queue interface is available inside the `java.util` package. It defines the methods that are commonly used by classes like `PriorityQueue` and `ArrayDeque`.

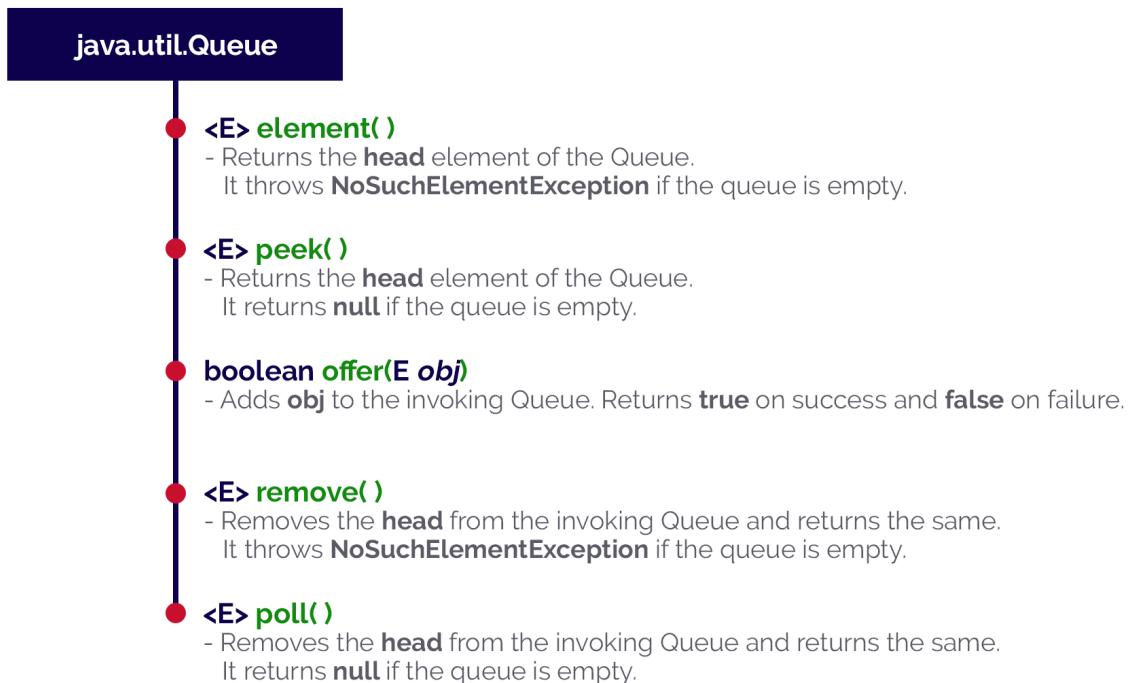
The Queue is used to organize a sequence of elements prior to the actual operation.

The Queue interface extends Collection interface.

- The Queue interface allows duplicate elements.
- The Queue interface preserves the order of insertion.

The Queue interface defines the following methods.

● ● ● Methods of Queue interface in java



www.btechsmartclass.com

Let's consider an example program on PriorityQueue to illustrate the methods of Queue interface.

Example

```
import java.util.*;

public class QueueInterfaceExample {
    public static void main(String[] args) {
        Queue queue = new PriorityQueue();
        queue.offer(10);
        queue.offer(20);
        queue.offer(30);
    }
}
```

```
queue.offer(40);

System.out.println("\nQueue elements are - " + queue);

System.out.println("\nHead element - " + queue.element());

System.out.println("\nHead element again - " + queue.peek());

queue.remove();

System.out.println("\nElements after Head removal - " + queue);

queue.poll();

System.out.println("\nElements after one more Head removal - " + queue);

}

}
```

Set Interface

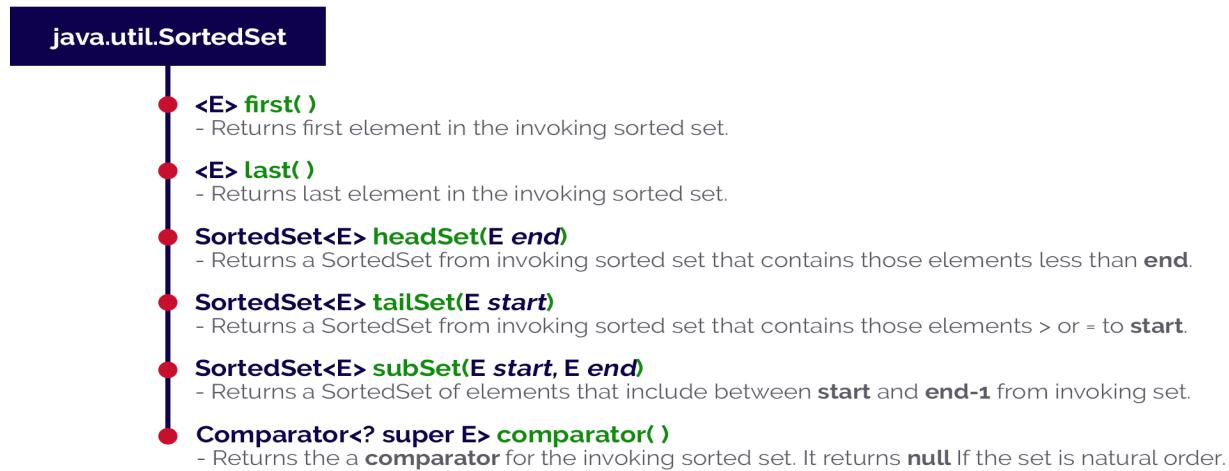
The Set interface is a child interface of Collection interface. It does not defines any additional methods of it, it has all the methods that are inherited from Collection interface. The Set interface does not allow duplicates. Set is a generic interface.

SortedSet Interface

The SortedSet interface is a child interface of the Set interface. The SortedSet interface is available inside the `java.util` package. It defines the methods that are used by classes `HashSet`, `LinkedHashSet`, and `TreeSet`.

- The SortedSet interface extends Set interface.
- The SortedSet interface does not allow duplicate elements.
- The SortedSet interface organise the elements based on the ascending order.

The SortedSet interface defines the following methods.



Let's consider an example program on TreeSet to illustrate the methods of SortedSet interface.

Example

```
import java.util.*;

public class SortedSetInterfaceExample {

    public static void main(String[] args) {

        SortedSet sortedSet = new TreeSet();

        sortedSet.add(10);
        sortedSet.add(20);
        sortedSet.add(5);
        sortedSet.add(40);
        sortedSet.add(30);

        System.out.println("\nElements of sortedSet: " + sortedSet);

        System.out.println("\nFirst element: " + sortedSet.first());
        System.out.println("\nLast element: " + sortedSet.last());
        System.out.println("\nSubset with upper limit: " + sortedSet.headSet(30));
        System.out.println("\nSubset with lower limit: " + sortedSet.tailSet(30));
        System.out.println("\nSubset with upper and lower limit: " + sortedSet.subSet(10,
30));
    }
}
```

```
    }  
}
```

4.3 Collection Classes:

ArrayList class

The ArrayList class is a part of java collection framework. It is available inside the java.util package. The ArrayList class extends AbstractList class and implements List interface. The elements of ArrayList are organized as an array internally. The default size of an ArrayList is 10.

The ArrayList class is used to create a dynamic array that can grow or shrunk as needed.

- The ArrayList is a child class of AbstractList
- The ArrayList implements interfaces like List, Serializable, Cloneable, and RandomAccess.
- The ArrayList allows to store duplicate data values.
- The ArrayList allows to access elements randomly using index-based accessing.
- The ArrayList maintains the order of insertion.
- ArrayList class declaration

The ArrayList class has the following declaration.

Example

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess,  
Cloneable, Serializable
```

ArrayList class constructors

The ArrayList class has the following constructors.

ArrayList() - Creates an empty ArrayList.

ArrayList(Collection c) - Creates an ArrayList with given collection of elements.

ArrayList(int size) - Creates an empty ArrayList with given size (capacity).

Operations on ArrayList

The ArrayList class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The ArrayList class has the following methods to add items.

boolean add(E element) - Appends given element to the ArrayList.

boolean addAll(Collection c) - Appends given collection of elements to the ArrayList.

void add(int index, E element) - Inserts the given element at specified index.

boolean addAll(int index, Collection c) - Inserts the given collection of elements at specified index.

Let's consider an example program to illustrate adding items to the ArrayList.

Example

```
import java.util.*;  
  
public class ArrayListExample {  
  
    public static void main(String[] args) {  
  
        ArrayList<String> list_1 = new ArrayList<String>();  
  
        ArrayList list_2 = new ArrayList();  
  
        //Appending  
  
        list_1.add("BTech");  
  
        System.out.println("list_1: " + list_1);  
  
        list_1.add("Class");  
  
        System.out.println("list_1: " + list_1);  
  
        //Inserting at specified index  
  
        list_1.add(1, "Smart");  
  
        System.out.println("list_1: " + list_1);  
  
        //Appending a collection of elements  
  
        list_2.addAll(list_1);  
  
        System.out.println("list_2: " + list_2);  
  
        //Inserting collection of elements at specified index  
  
        list_2.addAll(2, list_1);  
  
        System.out.println("list_2: " + list_2);  
  
    }  
}
```

Accessing Items

The ArrayList class has the following methods to access items.

E get(int index) - Returns element at specified index from the ArrayList.

ArrayList subList(int startIndex, int lastIndex) - Returns an ArrayList that contains elements from specified startIndex to lastIndex-1 from the invoking ArrayList.

int indexOf(E element) - Returns the index value of given element first occurrence in the ArrayList.

int lastIndexOf(E element) - Returns the index value of given element last occurrence in the ArrayList.

Let's consider an example program to illustrate accessing items from the ArrayList.

Example

```
import java.util.*;

public class ArrayListExample {

    public static void main(String[] args) {

        ArrayList<String> list_1 = new ArrayList<String>();

        list_1.add("BTech");
        list_1.add("Smart");
        list_1.add("Class");
        list_1.add("-");
        list_1.add("Java");
        list_1.add("Tutorial");
        list_1.add("Class");

        System.out.println("Element at index 4 is " + list_1.get(4));
        System.out.println("Sublist from index 1 to 4: " + list_1.subList(1, 5));
        System.out.println("Index of element \"Class\" is " + list_1.indexOf("Class"));

        System.out.println("Last index of element \"Class\" is " + list_1.lastIndexOf("Class"));

    }
}
```

Updating Items

The ArrayList class has the following methods to update or change items.

E set(int index, E newElement) - Replace the element at specified index with newElement in the invoking ArrayList.

ArrayList replaceAll(UnaryOperator e) - Replaces each element of invoking ArrayList with the result of applying the operator to that element.

Let's consider an example program to illustrate updating items in the ArrayList.

Example

```
import java.util.*;

public class ArrayListExample {

    public static void main(String[] args) {

        ArrayList<String> list_1 = new ArrayList<String>();

        list_1.add("BTech");

        list_1.add("Smart");

        list_1.add("Class");

        list_1.add("-");

        list_1.add("Java");

        list_1.add("Tutorial");

        list_1.add("Class");

        System.out.println("\nList before update: " + list_1);

        list_1.set(3, ":");

        System.out.println("\nList after update: " + list_1);

        list_1.replaceAll(e -> e.toUpperCase());

        System.out.println("\nList after update: " + list_1);

    }

}
```

Removing Items

The ArrayList class has the following methods to remove items.

E remove(int index) - Removes the element at specified index in the invoking ArrayList.

boolean remove(Object element) - Removes the first occurrence of the given element from the invoking ArrayList.

boolean removeAll(Collection c) - Removes the given collection of elements from the invoking ArrayList.

void retainAll(Collection c) - Removes all the elements except the given collection of elements from the invoking ArrayList.

boolean removeIf(Predicate filter) - Removes all the elements from the ArrayList that satisfies the given predicate.

void clear() - Removes all the elements from the ArrayList.

Let's consider an example program to illustrate removing items from the ArrayList.

Example

```
import java.util.*;

public class ArrayListExample {

    public static void main(String[] args) {

        ArrayList<String> list_1 = new ArrayList<String>();

        ArrayList list_2 = new ArrayList();
        ArrayList list_3 = new ArrayList();

        list_1.add("BTech");
        list_1.add("Smart");
        list_1.add("Class");
        list_1.add("-");
        list_1.add("Java");
        list_1.add("Tutorial");
        list_1.add("Classes");
        list_1.add("on");
        list_1.add("Collection");
    }
}
```

```
list_1.add("framwork");

list_1.add("-");

list_1.add("ArrayList");

list_2.add("Tutorial");

list_2.add("Java");

list_3.add("BTech");

list_3.add("Smart");

list_3.add("Class");

System.out.println("\nList_1 before remove:\n" + list_1);

System.out.println("\nList_2 before remove:\n" + list_2);

System.out.println("\nList_3 before remove:\n" + list_3);

list_1.remove(3);

System.out.println("\nList after removing element from index 3:\n" + list_1);

list_1.remove("Tutorial");

System.out.println("\nList after removing 'Tutorial' element:\n" + list_1);

list_1.removeAll(list_2);

System.out.println("\nList after removing all elements of list_2 from list_1:\n" +
list_1);

list_1.removeIf(n -> n.equals("Classes"));

System.out.println("\nList after removing all elements that are equal to
'Classes':\n" + list_1);

list_1 retainAll(list_3);

System.out.println("\nList after removing all elements from list_1 except
elements of list_3:\n" + list_1);

list_1.clear();

System.out.println("\nList after removing all elements from list_1:\n" + list_1);

}

}
```

LinkedList

The `LinkedList` class is a part of java collection framework. It is available inside the `java.util` package. The `LinkedList` class extends `AbstractSequentialList` class and implements `List` and `Deque` interface.

The elements of `LinkedList` are organized as the elements of linked list data structure.

- The `LinkedList` class is used to create a dynamic list of elements that can grow or shrink as needed.
- The `LinkedList` is a child class of `AbstractSequentialList`
- The `LinkedList` implements interfaces like `List`, `Deque`, `Cloneable`, and `Serializable`.
- The `LinkedList` allows to store duplicate data values.
- The `LinkedList` maintains the order of insertion.

LinkedList class declaration

The `LinkedList` class has the following declaration.

Example

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>,
Deque<E>, Cloneable, Serializable
```

LinkedList class constructors

The `LinkedList` class has the following constructors.

`LinkedList()` - Creates an empty List.

`LinkedList(Collection c)` - Creates a List with given collection of elements.

Operations on LinkedList

The `LinkedList` class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The `LinkedList` class has the following methods to add items.

`boolean add(E element)` - Appends given element to the List.

`boolean addAll(Collection c)` - Appends given collection of elements to the List.

`void add(int position, E element)` - Inserts the given element at specified position.

`boolean addAll(int position, Collection c)` - Inserts the given collection of elements at specified position.

`void addFirst(E element)` - Inserts the given element at beggining of the list.

void addLast(E element) - Inserts the given element at end of the list.

boolean offer(E element) - Inserts the given element at end of the list.

boolean offerFirst(E element) - Inserts the given element at beginning of the list.

boolean offerLast(E element) - Inserts the given element at end of the list.

void push(E element) - Inserts the given element at beginning of the list.

Let's consider an example program to illustrate adding items to the LinkedList.

Example

```
import java.util.*;

public class LinkedListExample {

    public static void main(String[] args) {

        LinkedList<String> list_1 = new LinkedList<String>();

        LinkedList list_2 = new LinkedList();

        list_2.add(10);

        list_2.add(20);

        list_2.addFirst(5);

        list_2.addLast(25);

        list_2.offer(2);

        list_2.offerFirst(1);

        list_2.offerLast(10);

        list_2.push(40);

        list_1.addAll(list_2);

        System.out.println("List_1: " + list_1);

        System.out.println("List_2: " + list_2);

    }

}
```

Accessing Items

The `LinkedList` class has the following methods to access items.

`E get(int position)` - Returns element at specified position from the `LinkedList`.

`E element()` - Returns the first element from the invoking `LinkedList`.

`E getFirst()` - Returns the first element from the invoking `LinkedList`.

`E getLast()` - Returns the last element from the invoking `LinkedList`.

`E peek()` - Returns the first element from the invoking `LinkedList`.

`E peekFirst()` - Returns the first element from the invoking `LinkedList`, and returns null if list is empty.

`E peekLast()` - Returns the last element from the invoking `LinkedList`, and returns null if list is empty.

`int indexOf(E element)` - Returns the index value of given element first occurrence in the `LinkedList`.

`int lastIndexOf(E element)` - Returns the index value of given element last occurrence in the `LinkedList`.

`E pop()` - Returns the first element from the invoking `LinkedList`.

Let's consider an example program to illustrate accessing items from the `LinkedList`.

Example

```
import java.util.*;

public class LinkedListExample {

    public static void main(String[] args) {

        LinkedList list_1 = new LinkedList();

        for(int i = 1; i <= 10; i++)

            list_1.add(i);

        System.out.println("List is " + list_1 + "\n");

        System.out.println("get(position) - " + list_1.get(3));

        System.out.println("getFirst() - " + list_1.getFirst());

        System.out.println("getLast() - " + list_1.getLast());

        System.out.println("element() - " + list_1.element());
```

```
System.out.println("peek() - " + list_1.peek());  
System.out.println("peekFirst() - " + list_1.peekFirst());  
System.out.println("peekLast() - " + list_1.peekLast());  
System.out.println("pop() - " + list_1.pop());  
System.out.println("indexOf(element) - " + list_1.indexOf(5));  
System.out.println("lastIndexOf(element) - " + list_1.lastIndexOf(5));  
  
}  
}
```

Updating Items

The LinkedList class has the following methods to update or change items.

E set(int index, E newElement) - Replace the element at specified index with newElement in the invoking LinkedList.

Let's consider an example program to illustrate updating items in the LinkedList.

Example

```
import java.util.*;  
  
public class LinkedListExample {  
  
    public static void main(String[] args) {  
  
        LinkedList list_1 = new LinkedList();  
  
        for(int i = 1; i <= 10; i++)  
  
            list_1.add(i);  
  
        System.out.println("List is " + list_1 + "\n");  
  
        list_1.set(3, 50);  
  
        System.out.println("List after update at index 3 is\n" + list_1 + "\n");  
  
    }  
}
```

Removing Items

The `LinkedList` class has the following methods to remove items.

`E remove()` - Removes the first element from the invoking `LinkedList`.

`E remove(int index)` - Removes the element at specified index in the invoking `LinkedList`.

`boolean remove(Object element)` - Removes the first occurrence of the given element from the invoking `LinkedList`.

`E removeFirst()` - Removes the first element from the invoking `LinkedList`.

`E removeLast()` - Removes the last element from the invoking `LinkedList`.

`boolean removeFirstOccurrence(Object element)` - Removes from the first occurrence of the given element from the invoking `LinkedList`.

`boolean removeLastOccurrence(Object element)` - Removes from the last occurrence of the given element from the invoking `LinkedList`.

`E poll()` - Removes the first element from the `LinkedList`, and returns null if the list is empty.

`E pollFirst()` - Removes the first element from the `LinkedList`, and returns null if the list is empty.

`E pollLast()` - Removes the last element from the `LinkedList`, and returns null if the list is empty.

`E pop()` - Removes the first element from the `LinkedList`.

`void clear()` - Removes all the elements from the `LinkedList`.

Let's consider an example program to illustrate removing items from the `LinkedList`.

Example

```
import java.util.*;

public class LinkedListExample {

    public static void main(String[] args) {

        LinkedList list_1 = new LinkedList();

        for(int i = 1; i <= 10; i++)

            list_1.add(i);

        System.out.println("List initially is " + list_1);

        list_1.remove();
```

```
        System.out.println("\nList after remove()\n" + list_1);
        list_1.remove(3);
        System.out.println("\nList after remove(index)\n" + list_1);
        list_1.removeFirst();
        System.out.println("\nList after removeFirst()\n" + list_1);
        list_1.removeLast();
        System.out.println("\nList after removeLast()\n" + list_1);
        list_1.removeFirstOccurrence(4);
        System.out.println("\nList after removeFirstOccurrence()\n" + list_1);

        list_1.removeLastOccurrence(7);
        System.out.println("\nList after removeLastOccurrence()\n" + list_1);

        list_1.pop();
        System.out.println("\nList after pop()\n" + list_1);
        list_1.clear();
        System.out.println("\nList after clear()\n" + list_1);
    }
}
```

HashSet

The HashSet class is a part of java collection framework. It is available inside the java.util package. The HashSet class extends AbstractSet class and implements Set interface. The elements of HashSet are organized using a mechanism called hashing. The HashSet is used to create hash table for storing set of elements.

The HashSet class is used to create a collection that uses a hash table for storing set of elements.

- The HashSet is a child class of AbstractSet
- The HashSet implements interfaces like Set, Cloneable, and Serializable.
- The HashSet does not allow to store duplicate data values, but null values are allowed.
- The HashSet does not maintain the order of insertion.
- The HashSet initial capacity is 16 elements.
- The HashSet is best suitable for search operations.

HashSet class declaration

The HashSet class has the following declaration.

Example

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable
```

HashSet class constructors

The HashSet class has the following constructors.

HashSet() - Creates an empty HashSet with the default initial capacity (16).

HashSet(Collection c) - Creates a HashSet with given collection of elements.

HashSet(int initialCapacity) - Creates an empty HashSet with the specified initial capacity.

HashSet(int initialCapacity, float loadFactor) - Creates an empty HashSet with the specified initial capacity and loadFactor.

Operations on HashSet

The HashSet class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The HashSet class has the following methods to add items.

boolean add(E element) - Inserts given element to the HashSet.

boolean addAll(Collection c) - Inserts given collection of elements to the HashSet.

Let's consider an example program to illustrate adding items to the HashSet.

Example

```
import java.util.*;

public class HashSetExample {

    public static void main(String[] args) {

        HashSet set = new HashSet();

        HashSet anotherSet = new HashSet();

        set.add(10);

        set.add(20);

        set.add(30);
```

```
        set.add(40);
        set.add(50);
        System.out.println("\nHashSet is\n" + set);
        anotherSet.addAll(set);
        System.out.println("\nanotherSet is\n" + anotherSet);
    }
}
```

Removing Items

The HashSet class has the following methods to remove items.

boolean remove(Object o) - Removes the specified element from the invoking HashSet.

boolean removeAll(Collection c) - Removes all the elements of specified collection from the invoking HashSet.

boolean removeIf(Predicate p) - Removes all of the elements of HashSet collection that satisfy the given predicate.

boolean retainAll(Collection c) - Removes all of the elements of HashSet collection except specified collection of elements.

void clear() - Removes all the elements from the HashSet.

Let's consider an example program to illustrate removing items from the HashSet.

Example

```
import java.util.*;
public class HashSetExample {
    public static void main(String[] args) {
        HashSet set = new HashSet();
        HashSet anotherSet = new HashSet();
        set.add(10);
        set.add(20);
        set.add(30);
        set.add(40);
```

```
set.add(50);

System.out.println("\nHashSet is\n" + set);

anotherSet.addAll(set);

System.out.println("\nanotherSet is\n" + anotherSet);

set.remove(20);

System.out.println("\nHashSet after remove(20) is\n" + set);

anotherSet.removeAll(set);

System.out.println("\nanotherSet after removeAll(set) is\n" + anotherSet);

set.retainAll(anotherSet);

System.out.println("\nset after retainAll(anotherSet) is\n" + set);

anotherSet.clear();

System.out.println("\nanotherSet after clear() is\n" + anotherSet);

}

}
```

TreeSet

The TreeSet class is a part of java collection framework. It is available inside the java.util package. The TreeSet class extends AbstractSet class and implements NavigableSet, Cloneable, and Serializable interfaces. The elements of TreeSet are organized using a mechanism called tree. The TreeSet class internally uses a TreeMap to store elements. The elements in a TreeSet are sorted according to their natural ordering.

- The TreeSet is a child class of AbstractSet
- The TreeSet implements interfaces like NavigableSet, Cloneable, and Serializable.
- The TreeSet does not allow to store duplicate data values, but null values are allowed.
- The elements in a TreeSet are sorted according to their natural ordering.
- The TreeSet initial capacity is 16 elements.
- The TreeSet is best suitable for search operations.

TreeSet class declaration

The TreeSet class has the following declaration.

Example sorting order

```
public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable
```

TreeSet class constructors

The TreeSet class has the following constructors.

`TreeSet()` - Creates an empty TreeSet in which elements will get stored in default natural sorting order.

`TreeSet(Collection c)` - Creates a TreeSet with given collection of elements.

`TreeSet(Comparator c)` - Creates an empty TreeSet with the specified sorting order.

`TreeSet(SortedSet s)` - This constructor is used to convert SortedSet to TreeSet.

Operations on TreeSet

The TreeSet class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The TreeSet class has the following methods to add items.

`boolean add(E element)` - Inserts given element to the TreeSet if it does not exist.

`boolean addAll(Collection c)` - Inserts given collection of elements to the TreeSet.

Let's consider an example program to illustrate adding items to the TreeSet.

Example

```
import java.util.*;

public class TreeSetExample {

    public static void main(String[] args) {

        TreeSet set = new TreeSet();

        TreeSet anotherSet = new TreeSet();

        set.add(10);

        set.add(20);

        set.add(15);

        set.add(5);

        System.out.println("\nset is\n" + set);

        anotherSet.addAll(set);

        System.out.println("\nanotherSet is\n" + anotherSet);
    }
}
```

```
    }  
}
```

Accessing Items

The TreeSet class has provides the following methods to access items.

E First() - Returns the first (smallest) element from the invoking TreeSet.

E last() - Returns the last (largest) element from the invoking TreeSet.

E higher(E obj) - Returns the largest element e such that e>obj. If it does not found returns null.

E lower(E obj) - Returns the largest element e such that e<obj. If it does not found returns null.

E ceiling(E obj) - Returns the smallest element e such that e>=obj. If it does not found returns null.

E floor(E obj) - Returns the largest element e such that e<=obj. If it does not found returns null.

SortedSet subSet(E fromElement, E toElement) - Returns a set of elements that lie between the given range which includes fromElement and excludes toElement.

NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) - Returns a set of elements that lie between the given range from the invoking TreeSet.

SortedSet tailSet(E fromElement) - Returns a set of elements that are greater than or equal to the specified fromElement from the invoking TreeSet.

NavigableSet tailSet(E fromElement, boolean inclusive) - Returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element from the invoking TreeSet.

SortedSet headSet(E fromElement) - Returns a set of elements that are smaller than or equal to the specified fromElement from the invoking TreeSet.

NavigableSet headSet(E fromElement, boolean inclusive) - Returns a set of elements that are smaller than or equal to (if, inclusive is true) the specified element from the invoking TreeSet.

Let's consider an example program to illustrate accessing items from a TreeSet.

Example

```
import java.util.*;  
  
public class TreeSetExample {  
    public static void main(String[] args) {  
        TreeSet set = new TreeSet();
```

```
Random num = new Random();

for(int i = 0; i < 10; i++)

    set.add(num.nextInt(100));

System.out.println("\nset is\n" + set);

System.out.println("\nfirst() - " + set.first());

System.out.println("\nlast() - " + set.last());

System.out.println("\nhigher(20) - " + set.higher(20));

System.out.println("\nlower(20) - " + set.lower(20));

System.out.println("\nceiling(30) - " + set.ceiling(30));

System.out.println("\nfloor(30) - " + set.floor(30));

System.out.println("\nsubSet(10, 50)\n" + set.subSet(10, 50));

System.out.println("\nsubSet(10, false, 50, true)\n" + set.subSet(10, false, 50,
true));

System.out.println("\nheadSet(20)\n" + set.headSet(20));

System.out.println("\nheadSet(20, false)\n" + set.headSet(20, false));

System.out.println("\ntailSet(20)\n" + set.tailSet(20));

System.out.println("\ntailSet(20, false)\n" + set.tailSet(20, false));

}

}
```

Updating Items

The TreeSet class has no methods to update or change items.

Removing Items

The TreeSet class has the following methods to remove items.

boolean remove(Object o) - Removes the specified element from the invoking TreeSet.

boolean removeAll(Collection c) - Removes all the elements those are in the specified collection from the invoking TreeSet.

boolean removeIf(Predicate p) - Removes all of the elements of the TreeSet collection that satisfy the given predicate.

boolean retainAll(Collection c) - Removes all the elements except those are in the specified collection from the invoking TreeSet.

E pollFirst() - Removes the first (smallest) element from the invoking TreeSet, and returns the same.

E pollLast() - Removes the last (largest) element from the invoking TreeSet, and returns the same.

void clear() - Removes all the elements from the TreeSet.

Let's consider an example program to illustrate removing items from the TreeSet.

Example

```
import java.util.*;  
  
public class TreeSetExample {  
  
    public static void main(String[] args) {  
  
        TreeSet set = new TreeSet();  
  
        TreeSet anotherSet = new TreeSet();  
  
        Random num = new Random();  
  
        for(int i = 0; i < 10; i++)  
  
            set.add(num.nextInt(100));  
  
        anotherSet.add(10);  
  
        anotherSet.add(20);  
  
        anotherSet.add(30);  
  
        System.out.println("\nset is\n" + set);  
  
        System.out.println("\nanotherSet is\n" + anotherSet);  
  
        set.remove(50);  
  
        System.out.println("\nset after remove(50) is\n" + set);  
  
        set.removeIf(n->n.equals(60));  
  
        System.out.println("\nset after removeIf(n->n.equals(60)) is\n" + set);  
  
  
        set.pollFirst();  
  
        System.out.println("\nset after pollFirst( ) is\n" + set);  
    }  
}
```

```

        set.pollLast();

        System.out.println("\nset after pollLast( ) is\n" + set);

        set.removeAll(anotherSet);

        System.out.println("\nset after removeAll(anotherSet) is\n" + set);

        set.retainAll(anotherSet);

        System.out.println("\nset after retainAll(anotherSet) is\n" + set);

    }

}

```

PriorityQueue

The PriorityQueue class is a part of java collection framework. It is available inside the java.util package. The PriorityQueue class extends AbstractQueue class and implements Serializable interface.

The elements of PriorityQueue are organized as the elements of queue data structure, but it does not follow FIFO principle. The PriorityQueue elements are organized based on the priority heap.

The PriorityQueue class is used to create a dynamic queue of elements that can grow or shrunk as needed.

- The PriorityQueue is a child class of AbstractQueue
- The PriorityQueue implements interface Serializable.
- The PriorityQueue allows to store duplicate data values, but not null values.
- The PriorityQueue maintains the order of insertion.
- The PriorityQueue used priority heap to organize its elements.

PriorityQueue class declaration

The PriorityQueue class has the following declaration.

Example

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable
```

PriorityQueue class constructors

The PriorityQueue class has the following constructors.

PriorityQueue() - Creates an empty PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.

PriorityQueue(Collection c) - Creates a PriorityQueue with given collection of elements.

PriorityQueue(int initialCapacity) - Creates an empty PriorityQueue with the specified initial capacity.

PriorityQueue(int initialCapacity, Comparator comparator) - Creates an empty PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

PriorityQueue(PriorityQueue pq) - Creates a PriorityQueue with the elements in the specified priority queue.

PriorityQueue(SortedSet ss) - Creates a PriorityQueue with the elements in the specified SortedSet.

Operations on PriorityQueue

The PriorityQueue class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The PriorityQueue class has the following methods to add items.

boolean add(E element) - Appends given element to the PriorityQueue.

boolean addAll(Collection c) - Appends given collection of elements to the PriorityQueue.

boolean offer(E element) - Appends given element to the PriorityQueue.

Let's consider an example program to illustrate adding items to the PriorityQueue.

Example

```
import java.util.*;

public class PriorityQueueExample {

    public static void main(String[] args) {

        PriorityQueue queue = new PriorityQueue();
        PriorityQueue anotherQueue = new PriorityQueue();

        queue.add(10);
        queue.add(20);
        queue.add(15);

        System.out.println("\nQueue is " + queue);

        anotherQueue.addAll(queue);

        System.out.println("\nanotherQueue is " + anotherQueue);
    }
}
```

```
anotherQueue.offer(25);

System.out.println("\nanotherQueue is " + anotherQueue);

}

}
```

Accessing Items

The PriorityQueue class has the following methods to access items.

E element() - Returns the first element from the invoking PriorityQueue.

E peek() - Returns the first element from the invoking PriorityQueue, returns null if this queue is empty.

Let's consider an example program to illustrate accessing items from the PriorityQueue.

Example

```
import java.util.*;

public class PriorityQueueExample {

    public static void main(String[] args) {

        PriorityQueue queue = new PriorityQueue();

        queue.add(10);

        queue.add(20);

        queue.add(15);

        System.out.println("\nQueue is " + queue);

        System.out.println("\nelement() - " + queue.element());

        System.out.println("\npeek() - " + queue.peek());

    }

}
```

Updating Items

The PriorityQueue class has no methods to update or change items.

Removing Items

The PriorityQueue class has the following methods to remove items.

E remove() - Removes the first element from the invoking PriorityQueue.

boolean remove(Object element) - Removes the first occurrence of the given element from the invoking PriorityQueue.

boolean removeAll(Collection c) - Removes all the elements of specified collection from the invoking PriorityQueue.

boolean removeIf(Predicate p) - Removes all of the elements of this collection that satisfy the given predicate.

boolean retainAll(Collection c) - Removes all the elements except those are in the specified collection from the invoking PriorityQueue.

E poll() - Removes the first element from the PriorityQueue, and returns null if the list is empty.

void clear() - Removes all the elements from the PriorityQueue.

Let's consider an example program to illustrate removing items from the PriorityQueue.

Example

```
import java.util.*;

public class PriorityQueueExample {

    public static void main(String[] args) {

        PriorityQueue queue = new PriorityQueue();
        PriorityQueue anotherQueue = new PriorityQueue();

        for(int i = 1; i <= 10; i++)

            queue.add(i);

        anotherQueue.add(5);
        anotherQueue.add(7);
        anotherQueue.add(8);

        System.out.println("\nQueue initially is\n" + queue);

        queue.remove();

        System.out.println("\nQueue after remove() is\n" + queue);

        queue.remove(8);

        System.out.println("\nQueue after remove(8) is\n" + queue);

        queue.poll();

        System.out.println("\nQueue after poll() is\n" + queue);
    }
}
```

```

        queue.removeIf(n->n.equals(6));

        System.out.println("\nQueue after removeIf(n->n.equals(6)) is\n" + queue);

        queue.addAll(anotherQueue);

        System.out.println("\nQueue after addAll(anotherQueue) is\n" + queue);

        queue.removeAll(anotherQueue);

        System.out.println("\nQueue after removeAll(anotherQueue) is\n" + queue);

        queue.clear();

        System.out.println("\nQueue after clear() is\n" + queue);

    }

}

```

ArrayDeque

The `ArrayDeque` class is a part of java collection framework. It is available inside the `java.util` package. The `ArrayDeque` class extends `AbstractCollection` class and implements `Deque`, `Cloneable`, and `Serializable` interfaces. The elements of `ArrayDeque` are organized as the elements of double ended queue data structure. The `ArrayDeque` is a special kind of array that grows and allows users to add or remove an element from both the sides of the queue.

The `ArrayDeque` class is used to create a dynamic double ended queue of elements that can grow or shrunk as needed.

- The `ArrayDeque` is a child class of `AbstractCollection`
- The `ArrayDeque` implements interfaces like `Deque`, `Cloneable`, and `Serializable`.
- The `ArrayDeque` allows to store duplicate data values, but not null values.
- The `ArrayDeque` maintains the order of insertion.
- The `ArrayDeque` allows to add and remove elements at both the ends.
- The `ArrayDeque` is faster than `LinkedList` and `Stack`.

ArrayDeque class declaration

The `ArrayDeque` class has the following declaration.

Example

```
public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable
```

ArrayDeque class constructors

The `PriorityQueue` class has the following constructors.

ArrayDeque() - Creates an empty ArrayDeque with the default initial capacity (16).

ArrayDeque(Collection c) - Creates a ArrayDeque with given collection of elements.

ArrayDeque(int initialCapacity) - Creates an empty ArrayDeque with the specified initial capacity.

Operations on ArrayDeque

The ArrayDeque class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The ArrayDeque class has the following methods to add items.

boolean add(E element) - Appends given element to the ArrayDeque.

boolean addAll(Collection c) - Appends given collection of elements to the ArrayDeque.

void addFirst(E element) - Adds given element at front of the ArrayDeque.

void addLast(E element) - Adds given element at end of the ArrayDeque.

boolean offer(E element) - Adds given element at end of the ArrayDeque.

boolean offerFirst(E element) - Adds given element at front of the ArrayDeque.

boolean offerLast(E element) - Adds given element at end of the ArrayDeque.

void push(E element) - Adds given element at front of the ArrayDeque.

Let's consider an example program to illustrate adding items to the ArrayDeque.

Example

```
import java.util.*;  
  
public class ArrayDequeExample {  
  
    public static void main(String[] args) {  
  
        ArrayDeque deque = new ArrayDeque();  
  
        ArrayDeque anotherDeque = new ArrayDeque();  
  
        deque.add(10);  
  
        deque.addFirst(5);  
  
        deque.addLast(15);  
  
        deque.offer(20);
```

```
        deque.offerFirst(10);
        deque.offerLast(30);
        System.out.println("\nDeque is\n" + deque);
        anotherDeque.addAll(deque);
        System.out.println("\nanotherDeque is\n" + anotherDeque);
        anotherDeque.push(40);
        System.out.println("\nanotherDeque after push(40) is\n" + anotherDeque);
    }
}
```

Accessing Items

The `ArrayDeque` class has the following methods to access items.

`E element()` - Returns the first element from the invoking `ArrayDeque`.

`E getFirst()` - Returns the first element from the invoking `ArrayDeque`.

`E getLast()` - Returns the last element from the invoking `ArrayDeque`.

`E peek()` - Returns the first element from the invoking `ArrayDeque`, returns null if this queue is empty.

`E peekFirst()` - Returns the first element from the invoking `ArrayDeque`, returns null if this queue is empty.

`E peekLast()` - Returns the last element from the invoking `ArrayDeque`, returns null if this queue is empty.

Let's consider an example program to illustrate accessing items from the `ArrayDeque`.

Example

```
import java.util.*;
public class ArrayDequeExample {
    public static void main(String[] args) {
        ArrayDeque deque = new ArrayDeque();
        for(int i = 1; i <= 10; i++)
            deque.add(i);
```

```
        System.out.println("\nDeque is\n" + deque);
        System.out.println("\nelement() - " + deque.element());
        System.out.println("\ngetFirst() - " + deque.getFirst());
        System.out.println("\ngetLast() - " + deque.getLast());
        System.out.println("\npeek() - " + deque.peek());
        System.out.println("\npeekFirst() - " + deque.peekFirst());
        System.out.println("\npeekLast() - " + deque.peekLast());
    }
}
```

Updating Items

The `ArrayDeque` class has no methods to update or change items.

Removing Items

The `ArrayDeque` class has the following methods to remove items.

`E remove()` - Removes the first element from the invoking `ArrayDeque`.

`E removeFirst()` - Removes the first element from the invoking `ArrayDeque`.

`E removeLast()` - Removes the last element from the invoking `ArrayDeque`.

`boolean remove(Object o)` - Removes the specified element from the invoking `ArrayDeque`.

`boolean removeFirstOccurrence(Object o)` - Removes the first occurrence of the specified element in this `ArrayDeque`.

`boolean removeLastOccurrence(Object o)` - Removes the last occurrence of the specified element in this `ArrayDeque`.

`boolean removeIf(Predicate p)` - Removes all of the elements of `ArrayDeque` collection that satisfy the given predicate.

`boolean retainAll(Collection c)` - Removes all of the elements of `ArrayDeque` collection except specified collection of elements.

`E poll()` - Removes the first element from the `ArrayDeque`, and returns null if the list is empty.

`E pollFirst()` - Removes the first element from the `ArrayDeque`, and returns null if the list is empty.

`E pollLast()` - Removes the last element from the `ArrayDeque`, and returns null if the list is empty.

E pop() - Removes the first element from the ArrayDeque.

void clear() - Removes all the elements from the PriorityQueue.

Let's consider an example program to illustrate removing items from the ArrayDeque.

Example

```
import java.util.*;  
  
public class ArrayDequeExample {  
  
    public static void main(String[] args) {  
  
        ArrayDeque deque = new ArrayDeque();  
  
        for(int i = 1; i <= 10; i++)  
  
            deque.add(i);  
  
        System.out.println("\nDeque is\n" + deque);  
  
        deque.remove();  
  
        System.out.println("\nDeque after remove()\n" + deque);  
  
        deque.removeFirst();  
  
        System.out.println("\nDeque after removeFirst()\n" + deque);  
  
        deque.removeLast();  
  
        System.out.println("\nDeque after removeLast()\n" + deque);  
  
        deque.remove(5);  
  
        System.out.println("\nDeque after remove(5)\n" + deque);  
  
        deque.poll();  
  
        System.out.println("\nDeque after poll()\n" + deque);  
  
        deque.pollFirst();  
  
        System.out.println("\nDeque after pollFirst()\n" + deque);  
  
        deque.pollLast();  
  
        System.out.println("\nDeque after pollLast()\n" + deque);  
  
        deque.pop();  
  
        System.out.println("\nDeque after pop()\n" + deque);  
    }  
}
```

```

        deque.clear();

        System.out.println("\nDeque after clear()\n" + deque);

    }

}

```

4.4 Accessing a Java Collection via a Iterator :

The java collection framework often we want to cycle through the elements. For example, we might want to display each element of a collection. The java provides an interface Iterator that is available inside the java.util package to cycle through each element of a collection.

- The Iterator allows us to move only forward direction.
- The Iterator does not support the replacement and addition of new elements.

We use the following steps to access a collection of elements using the Iterator.

Step - 1: Create an object of the Iterator by calling collection.iterator() method.

Step - 2: Use the method hasNext() to access to check does the collection has the next element. (Use a loop).

Step - 3: Use the method next() to access each element from the collection. (use inside the loop).

The Iterator has the following methods.

Method	Description
Iterator iterator()	Used to obtain an iterator to the start of the collection.
boolean hasNext()	Returns true if the collection has the next element, otherwise, it returns false.
E next()	Returns the next element available in the collection.

Let's consider an example program to illustrate accessing elements of a collection via the Iterator.

Example

```

import java.util.*;

public class TreeSetExample {

    public static void main(String[] args) {

        TreeSet set = new TreeSet();

```

```
Random num = new Random();

for(int i = 0; i < 10; i++)

    set.add(num.nextInt(100));

Iterator collection = set.iterator();

System.out.println("All the elements of TreeSet collection:");

while(collection.hasNext())

    System.out.print(collection.next() + ", ");

}

}
```

Accessing a collection using for-each

We can use the for-each statement to access elements of a collection.

Let's consider an example program to illustrate accessing items from a collection using a for-each statement.

Example

```
import java.util.*;

public class TreeSetExample {

    public static void main(String[] args) {

        TreeSet set = new TreeSet();

        ArrayList list = new ArrayList();

        PriorityQueue queue = new PriorityQueue();

        Random num = new Random();

        for(int i = 0; i < 10; i++) {

            set.add(num.nextInt(100));

            list.add(num.nextInt(100));

            queue.add(num.nextInt(100));

        }

        System.out.println("\nAll the elements of TreeSet collection:");

        for(Object element:set) {
```

```
        System.out.print(element + ", ");

    }

    System.out.println("\n\nAll the elements of ArrayList collection:");

    for(Object element:list) {

        System.out.print(element + ", ");

    }

    System.out.println("\n\nAll the elements of PriorityQueue collection:");

    for(Object element:queue) {

        System.out.print(element + ", ");

    }

}
```

The For-Each alternative

Using for-each, we can access the elements of a collection. But for-each can only be used if we don't want to modify the contents of a collection, and we don't want any reverse access.

Alternatively, we can use the Iterator to access or cycle through a collection of elements.

Let's consider an example program to illustrate The for-each alternative.

Example

```
import java.util.*;

public class TreeSetExample {

    public static void main(String[] args) {

        ArrayList list = new ArrayList();
        PriorityQueue queue = new PriorityQueue();
        Random num = new Random();

        for(int i = 0; i < 10; i++) {

            list.add(num.nextInt(100));
            queue.add(num.nextInt(100));

        }

    }

}
```

```

    }

    // Accessing using for-each statement

    System.out.println("\n\nAll the elements of ArrayList collection:");
    for(Object element:list) {
        System.out.print(element + ", ");
    }

    // Accessing using Iterator

    Iterator collection = queue.iterator();

    System.out.println("\n\nAll the elements of PriorityQueue collection:");

    while(collection.hasNext()) {
        System.out.print(collection.next() + ", ");
    }

}

```

4.5 Map Interfaces and Classes:

The java collection framework has an interface Map that is available inside the java.util package. The Map interface is not a subtype of Collection interface.

The Map interface has the following three classes.

Class	Description
HashMap	It implements the Map interface, but it doesn't maintain any order.
LinkedHashMap	It implements the Map interface, it also extends HashMap class. It maintains the insertion order.
TreeMap	It implements the Map and SortedMap interfaces. It maintains the ascending order.

Commonly used methods defined by Map interface

Method	Description
Object put(Object k, Object v)	It performs an entry into the Map.
Object putAll(Map m)	It inserts all the entries of m into invoking Map.
Object get(Object k)	It returns the value associated with given key.
boolean containsKey(Object k)	It returns true if map contain k as key. Otherwise false.
Set keySet()	It returns a set that contains all the keys from the invoking Map.
Set valueSet()	It returns a set that contains all the values from the invoking Map.
Set entrySet()	It returns a set that contains all the entries from the invoking Map.

Now, let's look at each class in detail with example programs.

HashMap Class

The HashMap class is a child class of AbstractMap, and it implements the Map interface. The HashMap is used to store the data in the form of key, value pair using hash table concept.

Key Properties of HashMap

HashMap is a child class of AbstractMap class.

HashMap implements the interfaces Map, Cloneable, and Serializable.

HashMap stores data as a pair of key and value.

HashMap uses Hash table concept to store the data.

HashMap does not allow duplicate keys, but values may be repeated.

HashMap allows only one null key and multiple null values.

HashMap does not follow any order.

HashMap has the default capacity 16 entries.

Let's consider an example program to illustrate HashMap.

Example

```
import java.util.*;  
  
public class HashMapExample {  
    public static void main(String[] args) {  
        Scanner read = new Scanner(System.in);  
  
        HashMap employeeInfo = new HashMap();  
  
        HashMap contactInfo = new HashMap();  
  
        employeeInfo.put(1, "Raja");  
        employeeInfo.put(2, "Gouthami");  
        employeeInfo.put(3, "Heyansh");  
        employeeInfo.put(4, "Yamini");  
        employeeInfo.put(5, "ManuTej");  
  
        System.out.println("Employee Information\n" + employeeInfo);  
  
        System.out.println("\nPlease enter the ID and Contact number");  
  
        System.out.println("Employee IDs : " + employeeInfo.keySet());  
  
        System.out.print("Enter ID: ");  
  
        int id = read.nextInt();  
  
        System.out.print("Enter Contact Number: ");  
  
        long contactNo = read.nextLong();  
  
        if(employeeInfo.containsKey(id)) {  
            contactInfo.put(id, contactNo);  
        }  
  
        System.out.println("\n\nEmployee Contact Information\n");  
  
        System.out.println("ID - " + id);  
  
        System.out.println("Name - " + employeeInfo.get(id));  
  
        System.out.println("Contact Number - " + contactInfo.get(id));  
    }  
}
```

LinkedHashMap Class

The LinkedHashMap class is a child class of HashMap, and it implements the Map interface. The LinkedHashMap is used to store the data in the form of key, value pair using hash table and linked list concepts.

Key Properties of LinkedHashMap

LinkedHashMap is a child class of HashMap class.

LinkedHashMap implements the Map interface.

LinkedHashMap stores data as a pair of key and value.

LinkedHashMap uses Hash table concept to store the data.

LinkedHashMap does not allow duplicate keys, but values may be repeated.

LinkedHashMap allows only one null key and multiple null values.

LinkedHashMap follows the insertion order.

LinkedHashMap has the default capacity 16 entries.

Let's consider an example program to illustrate LinkedHashMap.

Example

```
import java.util.*;

public class HashMapExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);

        LinkedHashMap employeeInfo = new LinkedHashMap();
        LinkedHashMap contactInfo = new LinkedHashMap();

        employeeInfo.put(1, "Raja");
        employeeInfo.put(2, "Gouthami");
        employeeInfo.put(3, "Heyansh");
        employeeInfo.put(4, "Yamini");
        employeeInfo.put(5, "ManuTej");

        System.out.println("Employee Information\n" + employeeInfo);

        System.out.println("\nPlease enter the ID and Contact number");
```

```

        System.out.println("Employee IDs : " + employeeInfo.keySet());

        System.out.print("Enter ID: ");

        int id = read.nextInt();

        System.out.print("Enter Contact Number: ");

        long contactNo = read.nextLong();

        if(employeeInfo.containsKey(id)) {

            contactInfo.put(id, contactNo);

        }

        System.out.println("\n\nEmployee Contact Information\n");

        System.out.println("ID - " + id);

        System.out.println("Name - " + employeeInfo.get(id));

        System.out.println("Contact Number - " + contactInfo.get(id));

    }

}

```

TreeMap Class

The TreeMap class is a child class of AbstractMap, and it implements the NavigableMap interface which is a child interface of SortedMap. The TreeMap is used to store the data in the form of key, value pair using a red-black tree concepts.

Key Properties of TreeMap

- TreeMap is a child class of AbstractMap class.
- TreeMap implements the NavigableMap interface which is a child interface of SortedMap interface.
- TreeMap stores data as a pair of key and value.
- TreeMap uses red-black tree concept to store the data.
- TreeMap does not allow duplicate keys, but values may be repeated.
- TreeMap does not allow null key, but allows null values.
- TreeMap follows the ascending order based on keys.
- Let's consider an example program to illustrate TreeMap.

Example

```

import java.util.*;

public class HashMapExample {

    public static void main(String[] args) {

```

```
Scanner read = new Scanner(System.in);

TreeMap employeeInfo = new TreeMap();

TreeMap contactInfo = new TreeMap();

employeeInfo.put(1, "Raja");

employeeInfo.put(4, "Gouthami");

employeeInfo.put(5, "Heyansh");

employeeInfo.put(3, "Yamini");

employeeInfo.put(2, "ManuTej");

System.out.println("Employee Information\n" + employeeInfo);

System.out.println("\nPlease enter the ID and Contact number");

System.out.println("Employee IDs : " + employeeInfo.keySet());

System.out.print("Enter ID: ");

int id = read.nextInt();

System.out.print("Enter Contact Number: ");

long contactNo = read.nextLong();

if(employeeInfo.containsKey(id)) {

    contactInfo.put(id, contactNo);

}

System.out.println("\n\nEmployee Contact Information\n");

System.out.println("ID - " + id);

System.out.println("Name - " + employeeInfo.get(id));

System.out.println("Contact Number - " + contactInfo.get(id));

}

}
```

4.6 Comparators:

The Comparator is an interface available in the `java.util` package. The `java Comparator` is used to order the objects of user-defined classes. The `java Comparator` can compare two objects from two different classes.

Using the java Comparator, we can sort the elements based on data members of a class. For example, we can sort based on rollNo, age, salary, marks, etc.

The Comparator interface has the following methods.

Method	Description
int compare(Object obj1, Object obj2)	It is used to compares the obj1 with obj2 .
boolean equals(Object obj)	It is used to check the equity between current object and argumented object.

The Comparator can be used in the following three ways.

Using a separate class that implements Comparator interface.

Using anonymous class.

Using lambda expression.

Using a separate class

We use the following steps to use Comparator with a separate class.

Step - 1: Create the user-defined class.

Step - 2: Create a class that implements Comparator interface.

Step - 3: Implement the compare() method of Comparator interface inside the above defined class(step - 2).

Step - 4: Create the actual class where we use the Comparator object with sort method of Collections class.

Step - 5: Create the object of Comparator interface using the class created in step - 2.

Step - 6: Call the sort method of Collections class by passing the object created in step - 6.

Step - 7: Use a for-each (any loop) to print the sorted information.

Let's consider an example program to illustrate Comparator using a separate class.

Example

```
import java.util.*;
```

```
class Student{
```

```
    String name;
```

```
    float percentage;
```

```
Student(String name, float percentage){  
    this.name = name;  
    this.percentage = percentage;  
}  
}  
  
class PercentageComparator implements Comparator<Student>{  
    public int compare(Student stud1, Student stud2) {  
        if(stud1.percentage < stud2.percentage)  
            return 1;  
        return -1;  
    }  
}  
  
public class StudentCompare{  
    public static void main(String args[]) {  
        ArrayList<Student> studList = new ArrayList<Student>();  
        studList.add(new Student("Gouthami", 90.61f));  
        studList.add(new Student("Raja", 83.55f));  
        studList.add(new Student("Honey", 85.55f));  
        studList.add(new Student("Teja", 77.56f));  
        studList.add(new Student("Varshith", 80.89f));  
        Comparator<Student> com = new PercentageComparator();  
        Collections.sort(studList, com);  
        System.out.println("Avg % --> Name");  
        System.out.println("-----");  
        for(Student stud:studList) {  
            System.out.println(stud.percentage + " --> " + stud.name);  
        }  
    }  
}
```

```
    }  
}
```

Using anonymous class

We use the following steps to use Comparator with anonymous class.

Step - 1: Create the user-defined class.

Step - 2: Create the actual class where we use the Comparator object with sort method of Collections class.

Step - 3: Create the object of Comparator interface using anonymous class and implement compare method of Comparator interface.

Step - 4: Call the sort method of Collections class by passing the object created in step - 6.

Step - 5: Use a for-each (any loop) to print the sorted information.

Let's consider an example program to illustrate Comparator using a separate class.

Example

```
import java.util.*;  
  
class Student{  
    String name;  
    float percentage;  
    Student(String name, float percentage){  
        this.name = name;  
        this.percentage = percentage;  
    }  
}  
  
public class StudentCompare{  
    public static void main(String args[]) {  
        ArrayList<Student> studList = new ArrayList<Student>();  
        studList.add(new Student("Gouthami", 90.61f));  
        studList.add(new Student("Raja", 83.55f));  
        studList.add(new Student("Honey", 85.55f));  
    }  
}
```

```
studList.add(new Student("Teja", 77.56f));
studList.add(new Student("Varshith", 80.89f));

Comparator<Student> com = new Comparator<Student>() {
    public int compare(Student stud1, Student stud2) {
        if(stud1.percentage < stud2.percentage)
            return 1;
        return -1;
    }
};

Collections.sort(studList, com);

System.out.println("Avg % --> Name");
System.out.println("-----");
for(Student stud:studList) {
    System.out.println(stud.percentage + " --> " + stud.name);
}
}
```

Using lamda expression

We use the following steps to use Comparator with lamda expression.

Step - 1: Create the user-defined class.

Step - 2: Create the actual class where we use the Comparator object with sort method of Collections class.

Step - 3: Create the object of Comparator interface using lamda expression and implement the code for compare method of Comparator interface.

Step - 4: Call the sort method of Collections class by passing the object created in step - 6.

Step - 5: Use a for-each (any loop) to print the sorted information.

Let's consider an example program to illustrate Comparator using a separate class.

Example

```
import java.util.*;  
  
class Student{  
  
    String name;  
  
    float percentage;  
  
    Student(String name, float percentage){  
  
        this.name = name;  
  
        this.percentage = percentage;  
  
    }  
  
}  
  
public class StudentCompare{  
  
    public static void main(String args[]){  
  
        ArrayList<Student> studList = new ArrayList<Student>();  
  
        studList.add(new Student("Gouthami", 90.61f));  
  
        studList.add(new Student("Raja", 83.55f));  
  
        studList.add(new Student("Honey", 85.55f));  
  
        studList.add(new Student("Teja", 77.56f));  
  
        studList.add(new Student("Varshith", 80.89f));  
  
        Comparator<Student> com = (stud1, stud2) -> {  
  
            if(stud1.percentage < stud2.percentage)  
  
                return 1;  
  
            return -1;  
  
        };  
  
        Collections.sort(studList, com);  
  
        System.out.println("Avg % --> Name");  
  
        System.out.println("-----");  
  
        for(Student stud:studList) {  
  
    }
```

```

        System.out.println(stud.percentage + " --> " + stud.name);

    }

}

}

```

4.7 Collection algorithms:

The java collection framework defines several algorithms as static methods that can be used with collections and map objects. All the collection algorithms in the java are defined in a class called Collections which defined in the java.util package.

All these algorithms are highly efficient and make coding very easy. It is better to use them than trying to re-implement them.

The collection framework has the following methods as algorithms.

Method	Description
void sort(List list)	Sorts the elements of the list as determined by their natural ordering.
void sort(List list, Comparator comp)	Sorts the elements of the list as determined by Comparator comp.
void reverse(List list)	Reverses all the elements sequence in list.
void rotate(List list, int n)	Rotates list by n places to the right. To rotate left, use a negative value for n.
void shuffle(List list)	Shuffles the elements in list.
void shuffle(List list, Random r)	Shuffles the elements in the list by using r as a source of random numbers.
void copy(List list1, List list2)	Copies the elements of list2 to list1.
List nCopies(int num, Object obj)	Returns num copies of obj contained in an immutable list. num can not be zero or negative.
void swap(List list, int idx1, int idx2)	Exchanges the elements in the list at the indices specified by idx1 and idx2.
int binarySearch(List list, Object value)	Returns the position of value in the list (must be in the sorted order), or -1 if value is not found.
int binarySearch(List list, Object value, Comparator c)	Returns the position of value in the list ordered according to c, or -1 if value is not found.
int indexOfSubList(List list, List subList)	Returns the index of the first match of subList in the list, or -1 if no match is found.
int lastIndexOfSubList(List list, List subList)	Returns the index of the last match of subList in the list, or -1 if no match is found.
Object max(Collection c)	Returns the largest element from the collection c as determined by natural ordering.

Object max(Collection c, Comparator comp)	Returns the largest element from the collection c as determined by Comparator comp.
Object min(Collection c)	Returns the smallest element from the collection c as determined by natural ordering.
Object min(Collection c, Comparator comp)	Returns the smallest element from the collection c as determined by Comparator comp.
void fill(List list, Object obj)	Assigns obj to each element of the list.
boolean replaceAll(List list, Object old, Object new)	Replaces all occurrences of old with new in the list.
Enumeration enumeration(Collection c)	Returns an enumeration over Collection c.
ArrayList list(Enumeration enum)	Returns an ArrayList that contains the elements of enum.
Set singleton(Object obj)	Returns obj as an immutable set.
List singletonList(Object obj)	Returns obj as an immutable list.
Map singletonMap(Object k, Object v)	Returns the key(k)/value(v) pair as an immutable map.
Collection synchronizedCollection(Collection c)	Returns a thread-safe collection backed by c.
List synchronizedList(List list)	Returns a thread-safe list backed by list.
Map synchronizedMap(Map m)	Returns a thread-safe map backed by m.
SortedMap synchronizedSortedMap(SortedMap sm)	Returns a thread-safe SortedMap backed by sm.
Set synchronizedSet(Set s)	Returns a thread-safe set backed by s.
SortedSet synchronizedSortedSet(SortedSet ss)	Returns a thread-safe set backed by ss.
Collection unmodifiableCollection(Collection c)	Returns an unmodifiable collection backed by c.
List unmodifiableList(List list)	Returns an unmodifiable list backed by list.
Set unmodifiableSet(Set s)	Returns an unmodifiable thread-safe set backed by s.
SortedSet unmodifiableSortedSet(SortedSet ss)	Returns an unmodifiable set backed by ss.
Map unmodifiableMap(Map m)	Returns an unmodifiable map backed by m.
SortedMap unmodifiableSortedMap(SortedMap sm)	Returns an unmodifiable SortedMap backed by sm.

Let's consider an example program to illustrate Collections algorithms.

Example

```
import java.util.*;  
  
public class CollectionAlgorithmsExample {  
  
    public static void main(String[] args) {  
  
        ArrayList list = new ArrayList();  
  
        PriorityQueue queue = new PriorityQueue();  
  
        HashSet set = new HashSet();  
  
        HashMap map = new HashMap();  
  
        Random num = new Random();  
  
        for(int i = 0; i < 5; i++) {  
  
            list.add(num.nextInt(100));  
  
            queue.add(num.nextInt(100));  
  
            set.add(num.nextInt(100));  
  
            map.put(i, num.nextInt(100));  
  
        }  
  
        System.out.println("List => " + list);  
  
        System.out.println("Queue => " + queue);  
  
        System.out.println("Set => " + set);  
  
        System.out.println("Map => " + map);  
  
        System.out.println("-----");  
  
        Collections.sort(list);  
  
        System.out.println("List in ascending order => " + list);  
  
        System.out.println("Largest element in set => " + Collections.max(set));  
  
        System.out.println("Smallest element in queue => " + Collections.min(queue));  
  
        Collections.reverse(list);  
  
        System.out.println("List in reverse order => " + list);  
    }  
}
```

```

        Collections.shuffle(list);

        System.out.println("List after shuffle => " + list);

    }

}

```

4.8 The Legacy Classes and Interfaces:

Dictionary

In java, the package `java.util` contains a class called `Dictionary` which works like a `Map`. The `Dictionary` is an abstract class used to store and manage elements in the form of a pair of key and value. The `Dictionary` stores data as a pair of key and value. In the dictionary, each key associates with a value. We can use the key to retrieve the value back when needed.

The `Dictionary` class is no longer in use, it is obsolete. As `Dictionary` is an abstract class we can not create its object. It needs a child class like `Hashtable`.

The `Dictionary` class in java has the following methods.

S. No.	Methods with Description
1	<code>Dictionary()</code> It's a constructor.
2	<code>Object put(Object key, Object value)</code> Inserts a key and its value into the dictionary. Returns null on success; returns the previous value associated with the key if the key is already exist.
3	<code>Object remove(Object key)</code> It returns the value associated with given key and removes the same; Returns null if the key does not exist.
4	<code>Object get(Object key)</code> It returns the value associated with given key; Returns null if the key does not exist.
5	<code>Enumeration keys()</code> Returns an enumeration of the keys contained in the dictionary.
6	<code>Enumeration elements()</code> Returns an enumeration of the values contained in the dictionary.
7	<code>boolean isEmpty()</code> It returns true if dictionary has no elements; otherwise returns false.
8	<code>int size()</code> It returns the total number of elements in the dictionary.

Let's consider an example program to illustrate methods of Dictionary class.

Example

```
import java.util.*;  
  
public class DictionaryExample {  
  
    public static void main(String args[]) {  
  
        Dictionary dict = new Hashtable();  
  
        dict.put(1, "Rama");  
  
        dict.put(2, "Seetha");  
  
        dict.put(3, "Heyansh");  
  
        dict.put(4, "Varshith");  
  
        dict.put(5, "Manutej");  
  
        System.out.println("Dictionary\n=> " + dict);  
  
        // keys()  
  
        System.out.print("\nKeys in Dictionary\n=> ");  
  
        for (Enumeration i = dict.keys(); i.hasMoreElements();) {  
  
            System.out.print(" " + i.nextElement());  
        } // elements()  
  
        System.out.print("\n\nValues in Dictionary\n=> ");  
  
        for (Enumeration i = dict.elements(); i.hasMoreElements();) {  
  
            System.out.print(" " + i.nextElement());  
        }  
  
        //get()  
  
        System.out.println("\n\nValue associated with key 3 => " + dict.get(3));  
  
        System.out.println("Value associated with key 30 => " + dict.get(30));  
  
        //size()
```

```

        System.out.println("\nDictionary has " + dict.size() + " elements");

        //isEmpty()

        System.out.println("\nIs Dictionary empty? " + dict.isEmpty());

    }

}

```

Hashtable

In java, the package `java.util` contains a class called `Hashtable` which works like a `HashMap` but it is synchronized. The `Hashtable` is a concrete class of `Dictionary`. It is used to store and manage elements in the form of a pair of key and value.

The `Hashtable` stores data as a pair of key and value. In the `Hashtable`, each key associates with a value. Any non-null object can be used as a key or as a value. We can use the key to retrieve the value back when needed.

The `Hashtable` class is no longer in use, it is obsolete. The alternate class is `HashMap`.

- The `Hashtable` class is a concrete class of `Dictionary`.
- The `Hashtable` class is synchronized.
- The `Hashtable` does not allow null key or value.
- The `Hashtable` has the initial default capacity 11.

The `Hashtable` class in java has the following constructors.

S.No.	Constructor with Description
1	<code>Hashtable()</code> It creates an empty hashtable with the default initial capacity 11.
2	<code>Hashtable(int capacity)</code> It creates an empty hashtable with the specified initial capacity.
3	<code>Hashtable(int capacity, float loadFactor)</code> It creates an empty hashtable with the specified initial capacity and loading factor.
4	<code>Hashtable(Map m)</code> It creates a hashtable containing elements of Map m.

The Hashtable class in java has the following methods.

S. No.	Methods with Description
1	V put(K key, V value) It inserts the specified key and value into the hash table.
2	void putAll(Map m) It inserts all the elements of Map m into the invoking Hashtable.
3	V putIfAbsent(K key, V value) If the specified key is not already associated with a value associates it with the given value and returns null, else returns the current value.
4	V getOrDefault(Object key, V defaultValue) It returns the value associated with given key; or defaultValue if the hashtable contains no mapping for the key.
5	V get(Object key) It returns the value associated with the given key.
6	Enumeration keys() Returns an enumeration of the keys of the hashtable.
7	Set keySet() Returns a set view of the keys of the hashtable.
8	Collection values() It returns a collection view of the values contained in the Hashtable.
9	Enumeration elements() Returns an enumeration of the values of the hashtable.
10	Set entrySet() It returns a set view of the mappings contained in the hashtable.
11	int hashCode() It returns the hash code of the hashtable.
12	Object clone() It returns a shallow copy of the Hashtable.
13	V remove(Object key) It returns the value associated with given key and removes the same.
14	boolean remove(Object key, Object value) It removes the specified values with the associated specified keys from the hashtable.

15	boolean contains(Object value)
	It returns true if the specified value found within the hash table, else return false.
16	boolean containsValue(Object value)
	It returns true if the specified value found within the hash table, else return false.
17	boolean containsKey(Object key)
	It returns true if the specified key found within the hash table, else return false.
18	V replace(K key, V value)
	It replaces the specified value for a specified key.
19	boolean replace(K key, V oldValue, V newValue)
	It replaces the old value with the new value for a specified key.
20	void replaceAll(BiFunction function)
	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
21	void rehash()
	It is used to increase the size of the hash table and rehashes all of its keys.
22	String toString()
	It returns a string representation of the Hashtable object.
23	V merge(K key, V value, BiFunction remappingFunction)
	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
24	void forEach(BiConsumer action)
	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
25	boolean isEmpty()
	It returns true if Hashtable has no elements; otherwise returns false.
26	int size()
	It returns the total number of elements in the Hashtable.
27	void clear()
	It is used to remove all the elements of a Hashtable.
28	boolean equals(Object o)
	It is used to compare the specified Object with the Hashtable.

Let's consider an example program to illustrate methods of Hashtable class.

Example

```
import java.util.*;  
  
public class HashtableExample {  
  
    public static void main(String[] args) {  
  
        Random num = new Random();  
  
        Hashtable table = new Hashtable();  
  
        //put(key, value)  
  
        for(int i = 1; i <= 5; i++)  
  
            table.put(i, num.nextInt(100));  
  
        System.out.println("Hashtable => " + table);  
  
        //get(key)  
  
        System.out.println("\nValue associated with key 3 => " + table.get(3));  
  
        System.out.println("Value associated with key 30 => " + table.get(30));  
  
        //keySet()  
  
        System.out.println("\nKeys => " + table.keySet());  
  
        //values()  
  
        System.out.println("\nValues => " + table.values());  
  
        //entrySet()  
  
        System.out.println("\nKey, Value pairs as a set => " + table.entrySet());  
  
        //hashCode()  
  
        System.out.println("\nHash code => " + table.hashCode());  
  
        //hashCode()  
  
        System.out.println("\nTotal number of elements => " + table.size());  
  
        //isEmpty()  
  
        System.out.println("\nEmpty status of Hashtable => " + table.isEmpty());  
  
    } }

---


```

Properties

In java, the package `java.util` contains a class called `Properties` which is a child class of `Hashtable` class. It implements interfaces like `Map`, `Cloneable`, and `Serializable`.

Java has this built-in class `Properties` which allow us to save and load multiple values from a file. This makes the class extremely useful for accessing data related to configuration.

The `Properties` class used to store configuration values managed as key, value pairs. In each pair, both key and value are `String` values. We can use the key to retrieve the value back when needed.

The `Properties` class provides methods to get data from the properties file and store data into the properties file. It can also be used to get the properties of a system.

- The `Properties` class is child class of `Hashtable` class.
- The `Properties` class implements `Map`, `Cloneable`, and `Serializable` interfaces.
- The `Properties` class used to store configuration values.
- The `Properties` class stores the data as key, value pairs.
- In `Properties` class both key and value are `String` data type.
- Using `Properties` class, we can load key, value pairs into a `Properties` object from a stream.
- Using `Properties` class, we can save the `Properties` object to a stream.

The `Properties` class in java has the following constructors.

S. No.	Constructor with Description
1	<code>Properties()</code> It creates an empty property list with no default values.
2	<code>Properties(Properties defaults)</code> It creates an empty property list with the specified defaults.

The `Properties` class in java has the following methods.

S.No.	Methods with Description
1	<code>void load(Reader r)</code> It loads data from the Reader object.
2	<code>void load(InputStream is)</code> It loads data from the InputStream object.
3	<code>void store(Writer w, String comment)</code> It writes the properties in the writer object.
4	<code>void store(OutputStream os, String comment)</code> It writes the properties in the OutputStream object.
5	<code>String getProperty(String key)</code>

	It returns value associated with the specified key.
6	String getProperty(String key, String defaultValue) It returns the value associated with given key; or defaultValue if the Properties contains no mapping for the key.
7	voidsetProperty(String key, String value) It calls the put method of Hashtable.
8	Enumeration propertyNames() It returns an enumeration of all the keys from the property list.
9	Set stringPropertyNames() Returns a set view of the keys of the Properties.
10	void list(PrintStream out) It is used to print the property list out to the specified output stream.
11	void loadFromXML(InputStream in) It is used to load all of the properties represented by the XML document on the specified input stream into this properties table.
12	void storeToXML(OutputStream os, String comment) It writes the properties in the writer object for generating XML document.
13	void storeToXML(Writer w, String comment, String encoding) It writes the properties in the writer object for generating XML document with the specified encoding.

Let's consider an example program to illustrate methods of Properties class to store a user configuration details to a properties file.

Example

```

import java.io.*;
import java.util.*;

public class PropertiesClassExample {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        File configFile = null;
        try {
            configFile = new File("config.properties");

```

```

fos = new FileOutputStream(configFile);
Properties configProperties = new Properties();
configProperties.setProperty("userName", "btechsmartclass");
configProperties.setProperty("password", "java");
configProperties.setProperty("email", "user@btechsmartclass.com");
configProperties.store(fos, "Login Details");
fos.close();
System.out.println("Configuration saved!!!"); }

catch(Exception e) {
    System.out.println("Something went wrong while opening file");
}
}
}
}

```

Stack

In java, the package `java.util` contains a class called `Stack` which is a child class of `Vector` class. It implements the standard principle `Last-In-First-Out` of stack data structure.

The `Stack` has `push` method for insertion and `pop` method for deletion. It also has other utility methods.

In `Stack`, the elements are added to the top of the stack and removed from the top of the stack.

The `Stack` class in java has the following constructor.

S. No.	Constructor with Description
1	<code>Stack()</code> It creates an empty <code>Stack</code> .

The `Stack` class in java has the following methods.

S.No.	Methods with Description
1	<code>Object push(Object element)</code> It pushes the element onto the stack and returns the same.
2	<code>Object pop()</code> It returns the element on the top of the stack and removes the same.
3	<code>int search(Object element)</code> If element found, it returns offset from the top. Otherwise, <code>-1</code> is returned.
4	<code>Object peek()</code> It returns the element on the top of the stack.
5	<code>boolean empty()</code> It returns true if the stack is empty, otherwise returns false.

Let's consider an example program to illustrate methods of Stack class.

Example

```
import java.util.*;
public class StackClassExample {
    public static void main(String[] args) {
        Stack stack = new Stack();
        Random num = new Random();
        for(int i = 0; i < 5; i++)
            stack.push(num.nextInt(100));
        System.out.println("Stack elements => " + stack);
        System.out.println("Top element is " + stack.peek());
        System.out.println("Removed element is " + stack.pop());
        System.out.println("Element 50 availability => " + stack.search(50));
        System.out.println("Stack is empty? - " + stack.isEmpty());
    }
}
```

Vector

In java, the package `java.util` contains a class called `Vector` which implements the `List` interface. The `Vector` is similar to an `ArrayList`. Like `ArrayList` `Vector` also maintains the insertion order. But `Vector` is synchronized, due to this reason, it is rarely used in the non-thread application. It also leads to poor performance.

- The `Vector` is a class in the `java.util` package.
- The `Vector` implements `List` interface.
- The `Vector` is a legacy class.
- The `Vector` is synchronized.

The `Vector` class in java has the following constructor.

S. No.	Constructor with Description
1	<code>Vector()</code> It creates an empty <code>Vector</code> with default initial capacity of 10.
2	<code>Vector(int initialSize)</code> It creates an empty <code>Vector</code> with specified initial capacity.
3	<code>Vector(int initialSize, int incr)</code> It creates a <code>vector</code> whose initial capacity is specified by size and whose increment is specified by incr.
4	<code>Vector(Collection c)</code> It creates a <code>vector</code> that contains the elements of collection c.

Let's consider an example program to illustrate methods of Vector class.

Example

```
import java.util.*;  
  
public class VectorClassExample  
{  
  
    public static void main(String[] args)  
    {  
  
        Vector list = new Vector();  
  
        list.add(10);  
  
        list.add(30);  
  
        list.add(0, 100);  
  
        list.addElement(50);  
  
        System.out.println("Vector => " + list);  
  
        System.out.println("get(2) => " + list.get(2));  
  
        System.out.println("firstElement() => " + list.firstElement());  
  
        System.out.println("indexOf(50) => " + list.indexOf(50));  
  
        System.out.println("contains(30) => " + list.contains(30));  
  
        System.out.println("capacity() => " + list.capacity());  
  
        System.out.println("size() => " + list.size());  
  
        System.out.println("isEmpty() => " + list.isEmpty());  
  
    }  
  
}
```

StringTokenizer

The StringTokenizer is a built-in class in java used to break a string into tokens. The StringTokenizer class is available inside the java.util package. The StringTokenizer class object internally maintains a current position within the string to be tokenized.

A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

The StringTokenizer class in java has the following constructor.

S. No.	Constructor with Description
1	StringTokenizer(String str) It creates StringTokenizer object for the specified string str with default delimiter.
2	StringTokenizer(String str, String delimiter) It creates StringTokenizer object for the specified string str with specified delimiter.
3	StringTokenizer(String str, String delimiter, boolean returnvalue) It creates StringTokenizer object with specified string, delimiter and returnvalue.

The StringTokenizer class in java has the following methods.

S.No.	Methods with Description
1	String nextToken() It returns the next token from the StringTokenizer object.
2	String nextToken(String delimiter) It returns the next token from the StringTokenizer object based on the delimiter.
3	Object nextElement() It returns the next token from the StringTokenizer object.
4	boolean hasMoreTokens() It returns true if there are more tokens in the StringTokenizer object. otherwise returns false.
5	boolean hasMoreElements() It returns true if there are more tokens in the StringTokenizer object. otherwise returns false.
6	int countTokens() It returns total number of tokens in the StringTokenizer object.

Let's consider an example program to illustrate methods of StringTokenizer class.

Example

```
import java.util.StringTokenizer;

public class StringTokenizerExample {

    public static void main(String[] args) {

        String url = "www.btechsmartclass.com";
        String title = "BTech Smart Class";
    }
}
```

```

 StringTokenizer tokens = new StringTokenizer(title);

 StringTokenizer anotherTokens = new StringTokenizer(url, ".");

 System.out.println("\nTotal tokens in title is " + tokens.countTokens());

 System.out.print("Tokens in the title => ");

 while(tokens.hasMoreTokens()) {

     System.out.print(tokens.nextToken() + ", ");

 }

 System.out.println("\n\nTotal tokens in url is " + anotherTokens.countTokens());

 System.out.println("Tokens in the url with delimiter(.) => ");

 while(anotherTokens.hasMoreElements()) {

     System.out.print(anotherTokens.nextElement() + ", ");

 }

}
}
}

```

BitSet

The BitSet is a built-in class in java used to create a dynamic array of bits represented by boolean values. The BitSet class is available inside the java.util package.

The BitSet array can increase in size as needed. This feature makes the BitSet similar to a Vector of bits.

- The bit values can be accessed by non-negative integers as an index.
- The size of the array is flexible and can grow to accommodate additional bit as needed.
- The default value of the BitSet is boolean false with a representation as 0 (off).
- BitSet uses 1 bit of memory per each boolean value.

The BitSet class in java has the following constructor.

S. No.	Constructor with Description
1	BitSet() It creates a default BitSet object.
2	BitSet(int noOfBits) It creates a BitSet object with number of bits that it can hold. All bits are initialized to zero.

The BitSet class in java has the following methods.

S.No.	Methods with Description
1	void and(BitSet bitSet) It performs AND operation on the contents of the invoking BitSet object with those specified by bitSet.
2	void andNot(BitSet bitSet) For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3	void flip(int index) It reverses the bit at specified index.
4	void flip(int startIndex, int endIndex) It reverses all the bits from specified startIndex to endIndex.
5	void or(BitSet bitSet) It performs OR operation on the contents of the invoking BitSet object with those specified by bitSet.
6	void xor(BitSet bitSet) It performs XOR operation on the contents of the invoking BitSet object with those specified by bitSet.
7	int cardinality() It returns the number of bits set to true in the invoking BitSet.
8	void clear() It sets all the bits to zeros of the invoking BitSet.
9	void clear(int index) It set the bit specified by given index to zero.
10	void clear(int startIndex, int endIndex) It sets all the bits from specified startIndex to endIndex to zero.
11	Object clone() It duplicates the invoking BitSet object.
12	boolean equals(Object bitSet) It retruns true if both invoking and argumented BitSets are equal, otherwise returns false.
13	boolean get(int index) It retruns the present state of the bit at given index in the invoking BitSet.
14	BitSet get(int startIndex, int endIndex) It returns a BitSet object that consists all the bits from startIndex to endIndex.
15	int hashCode() It returns the hash code of the invoking BitSet.
16	boolean intersects(BitSet bitSet) It returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
17	boolean isEmpty() It returns true if all bits in the invoking object are zero, otherwise returns

	false.
17	int length() It returns the total number of bits in the invoking BitSet.
18	int nextClearBit(int startIndex) It returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex.
19	int nextSetBit(int startIndex) It returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, -1 is returned.
20	void set(int index) It sets the bit specified by index.
21	void set(int index, boolean value) It sets the bit specified by index to the value passed in value.
22	void set(int startIndex, int endIndex) It sets all the bits from startIndex to endIndex.
23	void set(int startIndex, int endIndex, boolean value) It sets all the bits to specified value from startIndex to endIndex.
24	int size() It returns the total number of bits in the invoking BitSet.
25	String toString() It returns the string equivalent of the invoking BitSet object.

Let's consider an example program to illustrate methods of BitSet class.

Example

```
import java.util.*;

public class BitSetClassExample {

    public static void main(String[] args) {

        BitSet bSet_1 = new BitSet();
        BitSet bSet_2 = new BitSet(16);

        bSet_1.set(10);
        bSet_1.set(5);
        bSet_1.set(0);
        bSet_1.set(7);
        bSet_1.set(20);
        bSet_2.set(1);
```

```

bSet_2.set(15);

bSet_2.set(20);

bSet_2.set(77);

bSet_2.set(50);

System.out.println("BitSet_1 => " + bSet_1);

System.out.println("BitSet_2 => " + bSet_2);

bSet_1.and(bSet_2);

System.out.println("BitSet_1 after and with bSet_2 => " + bSet_1);

bSet_1.andNot(bSet_2);

System.out.println("BitSet_1 after andNot with bSet_2 => " + bSet_1);

System.out.println("Length of the bSet_2 => " + bSet_2.length());

System.out.println("Size of the bSet_2 => " + bSet_2.size());

System.out.println("Bit at index 2 in bSet_2 => " + bSet_2.get(2));

bSet_2.set(2);

System.out.println("Bit at index 2 after set in bSet_2 => " + bSet_2.get(2));

}

}

```

Date

The Date is a built-in class in java used to work with date and time in java. The Date class is available inside the java.util package. The Date class represents the date and time with millisecond precision. The Date class implements Serializable, Cloneable and Comparable interface. Most of the constructors and methods of Date class has been deprecated after Calendar class introduced.

The Date class in java has the following constructor.

S. No.	Constructor with Description
1	Date() It creates a Date object that represents current date and time.
2	Date(long milliseconds) It creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
3	Date(int year, int month, int date) – Deprecated

	It creates a date object with the specified year, month, and date.
4	Date(int year, int month, int date, int hrs, int min) – Deprecated It creates a date object with the specified year, month, date, hours, and minutes.
5	Date(int year, int month, int date, int hrs, int min, int sec) – Deprecated It creates a date object with the specified year, month, date, hours, minutes and seconds.
5	Date(String s) – Deprecated It creates a Date object and initializes it so that it represents the date and time indicated by the string s, which is interpreted as if by the parse(java.lang.String) method.

The Date class in java has the following methods.

S.No.	Methods with Description
1	long getTime() It returns the time represented by this date object.
2	boolean after(Date date) It returns true, if the invoking date is after the argumented date.
3	boolean before(Date date) It returns true, if the invoking date is before the argumented date.
4	Date from(Instant instant) It returns an instance of Date object from Instant date.
5	void setTime(long time) It changes the current date and time to given time.
6	Object clone() It duplicates the invoking Date object.
7	int compareTo(Date date) It compares current date with given date.
8	boolean equals(Date date) It compares current date with given date for equality.
9	int hashCode() It returns the hash code value of the invoking date object.
10	Instant toInstant() It converts current date into Instant object.
11	String toString() It converts this date into Instant object.

Let's consider an example program to illustrate methods of Date class.

Example

```
import java.time.Instant;
import java.util.Date;
public class DateClassExample {
    public static void main(String[] args) {
        Date time = new Date();
        System.out.println("Current date => " + time);
        System.out.println("Date => " + time.getTime() + " milliseconds");
        System.out.println("after() => " + time.after(time) + " milliseconds");
        System.out.println("before() => " + time.before(time) + " milliseconds");
        System.out.println("hashCode() => " + time.hashCode());
    }
}
```

Calendar

The Calendar is a built-in abstract class in java used to convert date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. The Calendar class is available inside the java.util package.

The Calendar class implements Serializable, Cloneable and Comparable interface. As the Calendar class is an abstract class, we can not create an object using it. We will use the static method Calendar.getInstance() to instantiate and implement a sub-class.

The Calendar class in java has the following methods.

S.No.	Methods with Description
1	Calendar getInstance() It returns a calendar using the default time zone and locale.
2	Date getTime() It returns a Date object representing the invoking Calendar's time value.
3	TimeZone getTimeZone() It returns the time zone object associated with the invoking calendar.

4	<code>String getCalendarType()</code>
	It returns an instance of Date object from Instant date.
5	<code>int get(int field)</code>
	It returns the value for the given calendar field.
6	<code>int getFirstDayOfWeek()</code>
	It returns the day of the week in integer form.
7	<code>int getWeeksInWeekYear()</code>
	It returns the total weeks in week year.
8	<code>int getWeekYear()</code>
	It returns the week year represented by current Calendar.
9	<code>void add(int field, int amount)</code>
	It adds the specified amount of time to the given calendar field.
10	<code>boolean after (Object when)</code>
	It returns true if the time represented by the Calendar is after the time represented by when Object.
11	<code>boolean before(Object when)</code>
	It returns true if the time represented by the Calendar is before the time represented by when Object.
12	<code>void clear(int field)</code>
	It sets the given calendar field value and the time value of this Calendar undefined.
13	<code>Object clone()</code>
	It returns the copy of the current object.
14	<code>int compareTo(Calendar anotherCalendar)</code>
	It compares and returns the time values (millisecond offsets) between two calendar objects.
15	<code>void complete()</code>
	It sets any unset fields in the calendar fields.
16	<code>void computeFields()</code>
	It converts the current millisecond time value to calendar field values in fields[].
17	<code>void computeTime()</code>
	It converts the current calendar field values in fields[] to the millisecond time value.
18	<code>boolean equals(Object object)</code>
	It returns true if both invoking object and argumented object are equal.
19	<code>int getActualMaximum(int field)</code>
	It returns the Maximum possible value of the specified calendar field.
20	<code>int getActualMinimum(int field)</code>
	It returns the Minimum possible value of the specified calendar field.
21	<code>Set getAvailableCalendarTypes()</code>
	It returns a string set of all available calendar type supported by Java Runtime Environment.

22	Locale[] getAvailableLocales()
	It returns an array of all locales available in java runtime environment.
23	String getDisplayName(int field, int style, Locale locale)
	It returns the String representation of the specified calendar field value in a given style, and local.
24	Map getDisplayNames(int field, int style, Locale locale)
	It returns Map representation of the given calendar field value in a given style and local.
25	int getGreatestMinimum(int field)
	It returns the highest minimum value of the specified Calendar field.
26	int getLeastMaximum(int field)
	It returns the highest maximum value of the specified Calendar field.
27	int getMaximum(int field)
	It returns the maximum value of the specified calendar field.
28	int getMinimalDaysInFirstWeek()
	It returns required minimum days in integer form.
29	int getMinimum(int field)
	It returns the minimum value of specified calendar field.
30	long getTimeInMillis()
	It returns the current time in millisecond.
31	int hashCode()
	It returns the hash code of the invoking object.
32	int internalGet(int field)
	It returns the value of the given calendar field.
33	boolean isLenient()
	It returns true if the interpretation mode of this calendar is lenient; false otherwise.
34	boolean isSet(int field)
	If not set then it returns false otherwise true.
35	boolean isWeekDateSupported()
	It returns true if the calendar supports week date. The default value is false.
36	void roll(int field, boolean up)
	It increase or decrease the specified calendar field by one unit without affecting the other field
37	void set(int field, int value)
	It sets the specified calendar field by the specified value.
38	void setFirstDayOfWeek(int value)
	It sets the first day of the week.
39	void setMinimalDaysInFirstWeek(int value)
	It sets the minimal days required in the first week.
40	void setTime(Date date)
	It sets the Time of current calendar object.
41	void setTimeInMillis(long millis)

	It sets the current time in millisecond.
42	void setTimeZone(TimeZone value)
	It sets the TimeZone with passed TimeZone value.
43	void setWeekDate(int weekYear, int weekOfYear, int dayOfWeek)
	It sets the current date with specified integer value.
44	Instant toInstant()
	It converts the current object to an instant.
45	String toString()
	It returns a string representation of the current object.

Let's consider an example program to illustrate methods of Calendar class.

Example

```
import java.util.*;

public class CalendarClassExample {

    public static void main(String[] args) {

        Calendar cal = Calendar.getInstance();

        System.out.println("Current date and time : \n=>" + cal);

        System.out.println("Current Calendar type : " + cal.getCalendarType());

        System.out.println("Current date and time : \n=>" + cal.getTime());

        System.out.println("Current date time zone : \n=>" + cal.getTimeZone());

        System.out.println("Calendar filed 1 (year): " + cal.get(1));

        System.out.println("Calendar day in integer form: " + cal.getFirstDayOfWeek());

        System.out.println("Calendar weeks in a year: " + cal.getWeeksInYear());

        System.out.println("Time in milliseconds: " + cal.getTimeInMillis());

        System.out.println("Available Calendar types: " + cal.getAvailableCalendarTypes());

        System.out.println("Calendar hash code: " + cal.hashCode());

        System.out.println("Is calendar supports week date? " + cal.isWeekDateSupported());

        System.out.println("Calendar string representation: " + cal.toString());

    }

}
```

Random

The Random is a built-in class in java used to generate a stream of pseudo-random numbers in java programming. The Random class is available inside the `java.util` package. The Random class implements `Serializable`, `Cloneable` and `Comparable` interface.

- The Random class is a part of `java.util` package.
- The Random class provides several methods to generate random numbers of type `integer`, `double`, `long`, `float` etc.
- The Random class is thread-safe.
- Random number generation algorithm works on the seed value. If not provided, seed value is created from system nano time.

The Random class in java has the following constructors.

S.No.	Constructor with Description
1	<code>Random()</code> It creates a new random number generator.
2	<code>Random(long seedValue)</code> It creates a new random number generator using a single long seedValue.

The Random class in java has the following methods.

S.No.	Methods with Description
1	<code>int next(int bits)</code> It generates the next pseudo-random number.
2	<code>Boolean nextBoolean()</code> It generates the next uniformly distributed pseudo-random boolean value.
3	<code>double nextDouble()</code> It generates the next pseudo-random double number between 0.0 and 1.0.
4	<code>void nextBytes(byte[] bytes)</code> It places the generated random bytes into an user-supplied byte array.
5	<code>float nextFloat()</code> It generates the next pseudo-random float number between 0.0 and 1.0..
6	<code>int nextInt()</code> It generates the next pseudo-random int number.
7	<code>int nextInt(int n)</code> It generates the next pseudo-random integer value between zero and n.
8	<code>long nextLong()</code> It generates the next pseudo-random, uniformly distributed long value.
9	<code>double nextGaussian()</code>

	It generates the next pseudo-random Gaussian distributed double number with mean 0.0 and standard deviation 1.0.
10	void setSeed(long seedValue) It sets the seed of the random number generator using a single long seedValue.
11	DoubleStream doubles() It returns a stream of pseudo-random double values, each conforming between 0.0 and 1.0.
12	DoubleStream doubles(double start, double end) It retruns an unlimited stream of pseudo-random double values, each conforming to the given start and end.
13	DoubleStream doubles(long streamSize) It returns a stream producing the pseudo-random double values for the given streamSize number, each between 0.0 and 1.0.
14	DoubleStream doubles(long streamSize, double start, double end) It returns a stream producing the given streamSizenumber of pseudo-random double values, each conforming to the given start and end.
15	IntStream ints() It returns a stream of pseudo-random integer values.
16	IntStream ints(int start, int end) It retruns an unlimited stream of pseudo-random integer values, each conforming to the given start and end.
17	IntStream ints(long streamSize) It returns a stream producing the pseudo-random integer values for the given streamSize number.
18	IntStream ints(long streamSize, int start, int end) It returns a stream producing the given streamSizenumber of pseudo-random integer values, each conforming to the given start and end.
19	LongStream longs() It returns a stream of pseudo-random long values.
20	LongStream longs(long start, long end) It retruns an unlimited stream of pseudo-random long values, each conforming to the given start and end.
21	LongStream longs(long streamSize) It returns a stream producing the pseudo-random long values for the given streamSize number.
22	LongStream longs(long streamSize, long start, long end) It returns a stream producing the given streamSizenumber of pseudo-random long values, each conforming to the given start and end.

Let's consider an example program to illustrate methods of Random class.

Example

```
import java.util.Random;

public class RandomClassExample {

    public static void main(String[] args) {

        Random rand = new Random();

        System.out.println("Integer random number - " + rand.nextInt());

        System.out.println("Integer random number from 0 to 100 - " +
rand.nextInt(100));

        System.out.println("Boolean random value - " + rand.nextBoolean());

        System.out.println("Double random number - " + rand.nextDouble());

        System.out.println("Float random number - " + rand.nextFloat());

        System.out.println("Long random number - " + rand.nextLong());

        System.out.println("Gaussian random number - " + rand.nextGaussian());

    }

}
```

Formatter

The Formatter is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming. The Formatter class is defined as final class inside the java.util package.

The Formatter class implements Cloneable and Flushable interface.

The Formatter class in java has the following constructors.

S.No.	Constructor with Description
1	Formatter() It creates a new formatter.
2	Formatter(Appendable a) It creates a new formatter with the specified destination.
3	Formatter(Appendable a, Locale l) It creates a new formatter with the specified destination and locale.
4	Formatter(File file)

	It creates a new formatter with the specified file.
5	Formatter(File file, String charset) It creates a new formatter with the specified file and charset.
6	Formatter(File file, String charset, Locale l) It creates a new formatter with the specified file, charset, and locale.
7	Formatter(Locale l) It creates a new formatter with the specified locale.
8	Formatter(OutputStream os) It creates a new formatter with the specified output stream.
9	Formatter(OutputStream os, String charset) It creates a new formatter with the specified output stream and charset.
10	Formatter(OutputStream os, String charset, Locale l) It creates a new formatter with the specified output stream, charset, and locale.
11	Formatter(PrintStream ps) It creates a new formatter with the specified print stream.
12	Formatter(String fileName) It creates a new formatter with the specified file name.
13	Formatter(String fileName, String charset) It creates a new formatter with the specified file name and charset.
14	Formatter(String fileName, String charset, Locale l) It creates a new formatter with the specified file name, charset, and locale.

The Formatter class in java has the following methods.

S.No.	Methods with Description
1	Formatter format(Locale l, String format, Object... args) It writes a formatted string to the invoking object's destination using the specified locale, format string, and arguments.
2	Formatter format(String format, Object... args) It writes a formatted string to the invoking object's destination using the specified format string and arguments.
3	void flush() It flushes the invoking formatter.
4	Appendable out() It returns the destination for the output.
5	Locale locale() It returns the locale set by the construction of the invoking formatter.
6	String toString() It converts the invoking object to string.
7	IOException ioException() It returns the IOException last thrown by the invoking formatter's Appendable.
8	void close() It closes the invoking formatter.

Let's consider an example program to illustrate methods of Formatter class.

Example

```
import java.util.*;  
  
public class FormatterClassExample {  
  
    public static void main(String[] args) {  
  
        Formatter formatter=new Formatter();  
  
        formatter.format("%2$5s %1$5s %3$5s", "Smart", "BTech", "Class");  
  
        System.out.println(formatter);  
  
        formatter = new Formatter();  
  
        formatter.format(Locale.FRANCE,"% .5f", -1325.789);  
  
        System.out.println(formatter);  
  
        String name = "Java";  
  
        formatter = new Formatter();  
  
        formatter.format(Locale.US,"Hello %s !", name);  
  
        System.out.println("+" + formatter + " " + formatter.locale());  
  
        formatter = new Formatter();  
  
        formatter.format("% .4f", 123.1234567);  
  
        System.out.println("Decimal floating-point notation to 4 places: " + formatter);  
  
        formatter = new Formatter();  
  
        formatter.format("%010d", 88);  
  
        System.out.println("value in 10 digits: " + formatter);  
    }  
}
```

Scanner

The Scanner is a built-in class in java used for read the input from the user in java programming. The Scanner class is defined inside the java.util package. The Scanner class implements Iterator interface.

The Scanner class provides the easiest way to read input in a Java program.

The Scanner object breaks its input into tokens using a delimiter pattern, the default delimiter is whitespace.

The Scanner class in java has the following constructors.

S.No.	Constructor with Description
1	Scanner(InputStream source) It creates a new Scanner that produces values read from the specified input stream.
2	Scanner(InputStream source, String charsetName) It creates a new Scanner that produces values read from the specified input stream.
3	Scanner(File source) It creates a new Scanner that produces values scanned from the specified file.
4	Scanner(File source, String charsetName) It creates a new Scanner that produces values scanned from the specified file.
5	Scanner(String source) It creates a new Scanner that produces values scanned from the specified string.
6	Scanner(Readable source) It creates a new Scanner that produces values scanned from the specified source.
7	Scanner(ReadableByteChannel source) It creates a new Scanner that produces values scanned from the specified channel.
8	Scanner(ReadableByteChannel source, String charsetName) It creates a new Scanner that produces values scanned from the specified channel.

The Scanner class in java has the following methods.

S.No.	Methods with Description
1	String next() It reads the next complete token from the invoking scanner.
2	String next(Pattern pattern) It reads the next token if it matches the specified pattern.
3	String next(String pattern) It reads the next token if it matches the pattern constructed from the specified string.
4	boolean nextBoolean() It reads a boolean value from the user.
5	byte nextByte() It reads a byte value from the user.
6	double nextDouble() It reads a double value from the user.
7	float nextFloat()

	It reads a floating-point value from the user.
8	int nextInt() It reads an integer value from the user.
9	long nextLong() It reads a long value from the user.
10	short nextShort() It reads a short value from the user.
11	String nextLine() It reads a string value from the user.
12	boolean hasNext() It returns true if the invoking scanner has another token in its input.
13	void remove() It is used when remove operation is not supported by this implementation of Iterator.
14	void close() It closes the invoking scanner.

Let's consider an example program to illustrate methods of Scanner class.

Example

```
import java.util.Scanner;

public class ScannerClassExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in); // Input stream is used

        System.out.print("Enter any name: ");

        String name = read.next();

        System.out.print("Enter your age in years: ");

        int age = read.nextInt();

        System.out.print("Enter your salary: ");

        double salary = read.nextDouble();

        System.out.print("Enter any message: ");

        read = new Scanner(System.in);

        String msg = read.nextLine();

        System.out.println("\n-----");

        System.out.println("Hello, " + name);
    }
}
```

```
System.out.println("You are " + age + " years old.");  
System.out.println("You are earning Rs." + salary + " per month.");  
System.out.println("Words from " + name + " - " + msg);}}
```

UNIT-V**GUI Programming with Java**

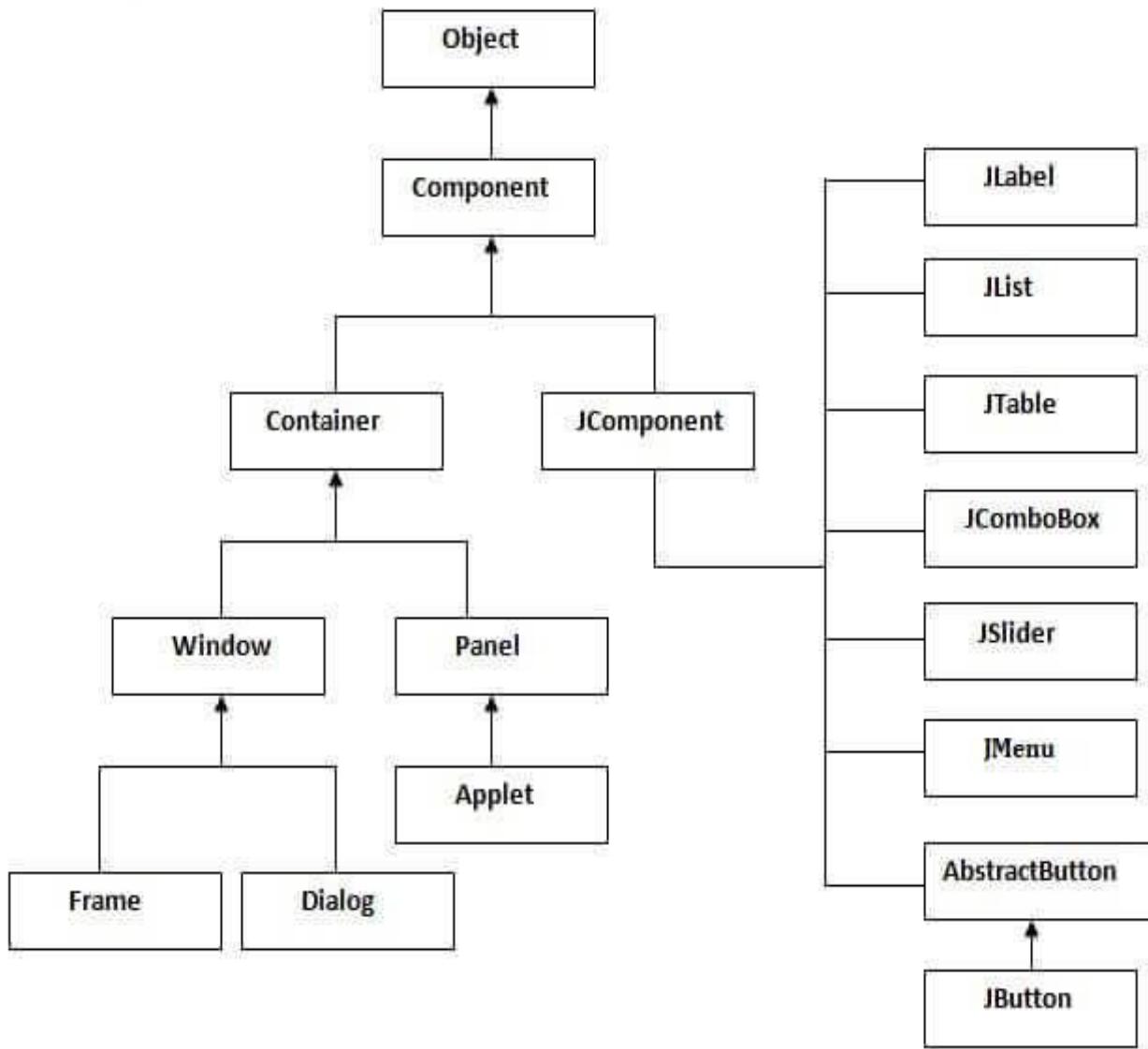
Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Hierarchy of Java Swing classes:

The hierarchy of java swing API is given below.



5.2 Limitations of AWT:

In Java, using AWT may create problems due to the following limitations:

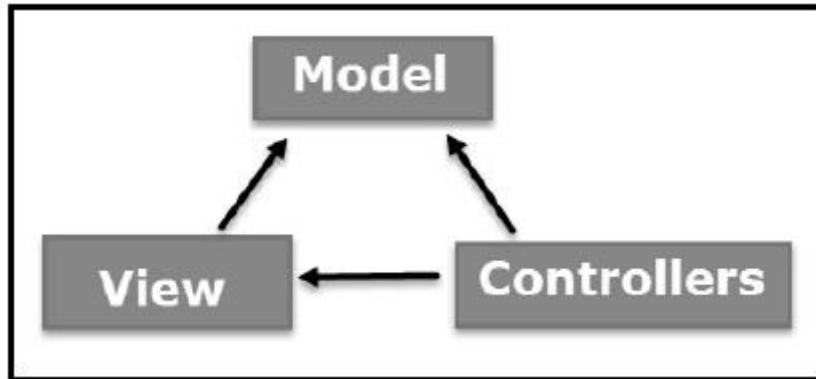
- Data structures like tables and trees are not included in the AWT toolkits.
- You can not set images for buttons.
- Being a local component, most of the AWT components can not be inherited. Only AWT's canvas components can be extended by other classes. Each graphical tool in AWT is mapped to any graphical tool rendered by the operating system. The functionalities supported by the toolkits will vary from one operating system to another as graphical library sub-routines are different for every platform. Hence, there may be a scenario in which it would create a problem as the Java program is platform-independent (i.e. after writing on one operating system, it can be executed on other systems)

5.3 MVC Architecture:

The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, the **view**, and the **controller**. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

MVC Components

Following are the components of MVC –



Model:

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

View:

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

Controller:

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

5.4 Components:

A graphical user interface is composed of individual building blocks such as push buttons, scrollbars, and pull-down menus. Some programmers know these individual building blocks as controls, while others call them widgets. In Java, they are typically called components because they all inherit from the base class `java.awt.Component`.

Component Name	Description
Button	A graphical push button.
Canvas	A heavyweight component that displays a blank canvas, allowing a program to display custom graphics.
Checkbox	A toggle button that can be selected or unselected. Use the Checkbox group to enforce mutually exclusive or radio button behavior among a group of Checkbox components.
CheckboxMenuItem	A toggle button that can appear within a Menu.
Choice	An option menu or drop-down list. Displays a menu of options when clicked on and allows the user to select among this fixed set of options.
Component	The base class for all AWT and Swing components. Defines many basic methods inherited by all components.
FileDialog	Allows the user to browse the filesystem and select or enter a filename.
Label	Displays a single line of read-only text. Does not respond to user input in any way.
List	Displays a list of choices (optionally scrollable) to the user and allows the user to select one or more of them.
Menu	single pane of a pull-down menu
MenuBar	A horizontal bar that contains pull-down menus.
MenuComponent	The base class from which all menu-related classes inherit.
MenuItem	A single item within a pull-down or pop-up menu pane.

PopupMenu	A menu pane for a pop-up menu.
Scrollbar	A graphical scrollbar.
TextArea	Displays multiple lines of plain text and allows the user to edit the text.
TextComponent	The base class for both TextArea and TextField.
TextField	Displays a single line of plain text and allows the user to edit the text.

5.5 Containers:

Java JFrame:

The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class.

JFrame works like the main window where components like labels, buttons, textfields are added

to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

JFrame Example

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class JFrameExample {
    public static void main(String s[]) {
        JFrame frame = new JFrame("JFrame Example");
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());
        JLabel label = new JLabel("JFrame By Example");
        JButton button = new JButton();
        button.setText("Button");
        panel.add(label);
        panel.add(button);
        frame.add(panel);
        frame.setSize(200, 300);
```

```

frame.setLocationRelativeTo(null);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
} }

```

JApplet:

As we prefer Swing to AWT. Now we can use JApplet that can have all the controls of swing.

The JApplet class extends the Applet class.

Example of EventHandling in JApplet:

```

import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener{
JButton b;
JTextField tf;
public void init(){
tf=new JTextField();
tf.setBounds(30,40,150,20);
b=new JButton("Click");
b.setBounds(80,150,70,40);
add(b);add(tf);
b.addActionListener(this);
setLayout(null);
}

```

```

public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
} }

```

In the above example, we have created all the controls in init() method because it is invoked only once.

```

myapplet.html
<html>
<body>
<applet code="EventJApplet.class" width="300" height="300">
</applet>

```

```
</body>  
</html>
```

JDialog:

The JDialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Dialog class.

Unlike JFrame, it doesn't have maximize and minimize buttons.

JDialog class declaration

Let's see the declaration for javax.swing.JDialog class.

```
public class JDialog extends Dialog implements WindowConstants, Accessible,  
RootPaneContainer
```

Commonly used Constructors:

Constructor Description

JDialog() It is used to create a modeless dialog without a title and without a specified Frame owner.

JDialog(Frame owner) It is used to create a modeless dialog with specified Frame as its owner and an empty title.

JDialog(Frame owner, String title, boolean modal)

It is used to create a dialog with the specified title, owner Frame and modality.

Java JDialog Example:

```
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
public class DialogExample {  
    private static JDialog d;  
    DialogExample() {  
        JFrame f = new JFrame();  
        d = new JDialog(f, "Dialog Example", true);  
        d.setLayout( new FlowLayout() );  
        JButton b = new JButton ("OK");  
        b.addActionListener ( new ActionListener()  
        {  
            public void actionPerformed( ActionEvent e )  
            {  
                DialogExample.d.setVisible(false);  
            }  
        }  
    }  
}
```

```
});
```

Output:

```
d.add( new JLabel ("Click button to continue."));  
d.add(b);  
d.setSize(300,300);  
d.setVisible(true);  
}  
public static void main(String args[])  
{  
new DialogExample();  
} }
```

JPanel:

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class.

It doesn't have title bar.

JPanel class declaration:

1. public class JPanel extends JComponent implements Accessible

Java JPanel Example:

```
import java.awt.*;  
import javax.swing.*;  
public class PanelExample {  
PanelExample()  
{  
JFrame f= new JFrame("Panel Example");  
JPanel panel=new JPanel();  
panel.setBounds(40,80,200,200);  
panel.setBackground(Color.gray);  
JButton b1=new JButton("Button 1");  
b1.setBounds(50,100,80,30);  
b1.setBackground(Color.yellow);  
JButton b2=new JButton("Button 2");  
b2.setBounds(100,100,80,30);
```

```
b2.setBackground(Color.green);
panel.add(b1); panel.add(b2);
f.add(panel);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[])
{
new PanelExample();
} }
```

Overview of some Swing Components

Java JButton:

The JButton class is used to create a labeled button that has platform independent implementation. The

application result in some action when the button is pushed. It inherits AbstractButton class.

JButton class declaration:

Let's see the declaration for javax.swing.JButton class.

1. public class JButton extends AbstractButton implements Accessible

Java JButton Example

```
import javax.swing.*;
public class ButtonExample {
public static void main(String[] args) {
JFrame f=new JFrame("Button Example");
JButton b=new JButton("Click Here");
b.setBounds(50,100,95,30);
f.add(b);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true); } }
```

Java JLabel:

The object of JLabel class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

JLabel class declaration:

Let's see the declaration for javax.swing.JLabel class.

1. public class JLabel extends JComponent implements SwingConstants, Accessible
- Commonly used Constructors:

Java JLabel Example

```
import javax.swing.*;  
class LabelExample  
{  
    public static void main(String args[])  
    {  
        JFrame f= new JFrame("Label Example");  
        JLabel l1,l2;  
        l1=new JLabel("First Label.");  
        l1.setBounds(50,50, 100,30);  
        l2=new JLabel("Second Label.");  
        l2.setBounds(50,100, 100,30);  
        f.add(l1); f.add(l2);  
        f.setSize(300,300);  
        f.setLayout(null);  
        f.setVisible(true);  
    }  
}
```

JTextField:

The object of a JTextField class is a text component that allows the editing of a single line text. It

inherits JTextComponent class.

JTextField class declaration

Let's see the declaration for javax.swing.JTextField class.

1. public class JTextField extends JTextComponent implements SwingConstants
- 2.

Java JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
{
JFrame f= new JFrame("TextField Example");
JTextField t1,t2;
t1=new JTextField("Welcome to Javatpoint.");
t1.setBounds(50,100, 200,30);
t2=new JTextField("AWT Tutorial");
t2.setBounds(50,150, 200,30);
f.add(t1); f.add(t2);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
} }
```

Java JTextArea

The object of a JTextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

JTextArea class declaration:

```
public class JTextArea extends JTextComponent
```

Java JTextArea Example:

```
import javax.swing.*;
public class TextAreaExample
{
TextAreaExample()
{
JFrame f= new JFrame();
JTextArea area=new JTextArea("Welcome to javatpoint");
area.setBounds(10,30, 200,200);
f.add(area);
f.setSize(300,300);
f.setLayout(null);
```

```
f.setVisible(true);  
}  
public static void main(String args[])  
{  
new TextAreaExample();  
}  
}
```

Simple Java Applications:

```
import javax.swing.JFrame;  
import javax.swing.SwingUtilities;  
public class Example extends JFrame  
{  
public Example()  
{  
setTitle("Simple example");  
setSize(300, 200);  
setLocationRelativeTo(null);  
setDefaultCloseOperation(EXIT_ON_CLOSE);  
}  
public static void main(String[] args)  
{  
Example ex = new Example();  
ex.setVisible(true);  
}
```

5.6 Understanding Layout Management:

Java LayoutManagers:

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers.

5.6.1 Java FlowLayout:

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.

Fields of FlowLayout class:

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

Constructors of FlowLayout class:

1. **FlowLayout()**: creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align)**: creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap)**: creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example of FlowLayout class:

```
import java.awt.*;  
import javax.swing.*;  
  
public class MyFlowLayout{  
  
    JFrame f;  
  
    MyFlowLayout(){  
        f=new JFrame();  
  
        JButton b1=new JButton("1");  
  
        JButton b2=new JButton("2");  
  
        JButton b3=new JButton("3");
```

```
 JButton b4=new JButton("4");
 JButton b5=new JButton("5");
 f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
 f.setLayout(new FlowLayout(FlowLayout.RIGHT));
 //setting flow layout of right alignment
 f.setSize(300,300);
 etVisible(true);
}

public static void main(String[] args) {
new MyFlowLayout();
}
}
```

5.6.2 BorderLayout:

The BorderLayout provides five constants for each region:

1. public static final int NORTH
2. public static final int SOUTH
3. public static final int EAST
4. public static final int WEST
5. public static final int CENTER

Constructors of BorderLayout class:

BorderLayout(): creates a border layout but with no gaps between the components.

JBorderLayout(int hgap, int vgap): creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class:

```
import java.awt.*; Output:  
import javax.swing.*;  
  
public class Border  
{  
JFrame f;  
  
Border()  
{  
f=new JFrame();  
  
JButton b1=new JButton("NORTH");;  
  
JButton b2=new JButton("SOUTH");;  
  
JButton b3=new JButton("EAST");;  
  
JButton b4=new JButton("WEST");;  
  
JButton b5=new JButton("CENTER");;  
  
f.add(b1,BorderLayout.NORTH);  
  
f.add(b2,BorderLayout.SOUTH);  
  
f.add(b3,BorderLayout.EAST);  
  
f.add(b4,BorderLayout.WEST);  
  
f.add(b5,BorderLayout.CENTER);  
  
f.setSize(300,300);  
  
etVisible(true);  
}  
  
public static void main(String[] args)  
{  
new Border();
```

```
 }  
 }
```

5.6.3 Java Grid Layout:

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

1. **GridLayout()**: creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns)**: creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap)**: creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

Example of GridLayout class:

```
1. import java.awt.*;  
2. import javax.swing.*;  
  
public class MyGridLayout  
{  
    JFrame f;  
  
    MyGridLayout()  
    {  
        f=new JFrame();  
  
        JButton b1=new JButton("1");  
        JButton b2=new JButton("2");  
        JButton b3=new JButton("3");  
        JButton b4=new JButton("4");  
        JButton b5=new JButton("5");  
    }  
}
```

```
 JButton b6=new JButton("6");
 JButton b7=new JButton("7");
 JButton b8=new JButton("8");
 JButton b9=new JButton("9");
 f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
 f.add(b6);f.add(b7);f.add(b8);f.add(b9);
 f.setLayout(new GridLayout(3,3));
 //setting grid layout of 3 rows and 3 columns
 f.setSize(300,300);
 etVisible(true);
}

public static void main(String[] args)
{
    new MyGridLayout();
}
```

5.6.4 Java CardLayout:

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout class

1. **CardLayout():** creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

Commonly used methods of CardLayout class

public void next(Container parent): is used to flip to the next card of the given container.

public void previous(Container parent): is used to flip to the previous card of the given container.

public void first(Container parent): is used to flip to the first card of the given container.

public void last(Container parent): is used to flip to the last card of the given container.

public void show(Container parent, String name): is used to flip to the specified card with the given name.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutExample extends JFrame implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
CardLayoutExample(){
c=getContentPane();
card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver space
c.setLayout(card);
b1=new JButton("Apple");
b2=new JButton("Boy");
b3=new JButton("Cat");
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
c.add("a",b1);c.add("b",b2);c.add("c",b3);
}
public void actionPerformed(ActionEvent e) {
card.next(c);
}
public static void main(String[] args) {
CardLayoutExample cl=new CardLayoutExample();
cl.setSize(400,400);
cl.setVisible(true);
cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
```

5.6.5 Java GridBagLayout:

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of constraints object we arrange component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.

Fields:

Modifier and Type	Field	Description
double[]	columnWeights	It is used to hold the overrides to the column weights.
int[]	columnWidths	It is used to hold the overrides to the column minimum width.
protected Hashtable<Component,GridBag Constraints>	Comptable	It is used to maintains the association between a component and its gridbag constraints.
protected GridBagConstraints	defaultConstraints	It is used to hold a gridbag constraints instance containing the default values.
protected GridBagLayoutInfo	layoutInfo	It is used to hold the layout information for the gridbag.
protected static int	MAXGRIDSIZE	No longer in use just for backward compatibility
protected static int	MINSIZE	It is smallest grid that can be laid out by the grid bag layout.
protected static int	PREFERREDSIZE	It is preferred grid size that can be laid out by the

		grid bag layout.
int[]	rowHeights	It is used to hold the overrides to the row minimum heights.
double[]	rowWeights	It is used to hold the overrides to the row weights.

Example:

```

import java.awt.Button;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.*;
public class GridBagLayoutExample extends JFrame{
public static void main(String[] args) {
GridBagLayoutExample a = new GridBagLayoutExample();
}
public GridBagLayoutExample() {
GridBagLayoutgrid = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(grid);
setTitle("GridBag Layout Example");
GridBagLayout layout = new GridBagLayout();
this.setLayout(layout);
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridx = 0;
gbc.gridy = 0;
this.add(new Button("Button One"), gbc);
gbc.gridx = 1;
gbc.gridy = 0;
this.add(new Button("Button two"), gbc);
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.ipady = 20;
gbc.gridx = 0;
gbc.gridy = 1;
}

```

```
this.add(new Button("Button Three"), gbc);
gbc.gridx = 1;
gbc.gridy = 1;
this.add(new Button("Button Four"), gbc);
gbc.gridx = 0;
gbc.gridy = 2;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridwidth = 2;
this.add(new Button("Button Five"), gbc);
setSize(300, 300);
setPreferredSize(getSize());
setVisible(true);
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
```

5.7 Event Handling:

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The `java.awt.event` package provides many event classes and Listener interfaces for event handling.

Types of Event

The events can be broadly classified into two categories:

Foreground Events :- Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

Background Events :- Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed

when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

The Delegation Event Model:

The Delegation Event Model has the following key participants namely:

Event Source

Event Listener

5.8 Event Source:

The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides as with classes for source object.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

5.9 Event Listener:

It is also known as event handler. Listener is responsible for generating response to an event. From Java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event and then returns.

Event Listener Interfaces:

Following is the list of commonly used event listeners.

S. No.	Control & Description
1	<u>ActionListener</u> This interface is used for receiving the action events.
2	<u>ComponentListener</u> This interface is used for receiving the component events.
3	<u>ItemListener</u>

	This interface is used for receiving the item events.
4	<u>KeyListener</u> This interface is used for receiving the key events.
5	<u>MouseListener</u> This interface is used for receiving the mouse events.
6	<u>TextListener</u> This interface is used for receiving the text events.
7	<u>WindowListener</u> This interface is used for receiving the window events.
8	<u>AdjustmentListener</u> This interface is used for receiving the adjustment events.
9	<u>ContainerListener</u> This interface is used for receiving the container events.
10	<u>MouseMotionListener</u> This interface is used for receiving the mouse motion events.
11	<u>FocusListener</u> This interface is used for receiving the focus events.

5.10 Event Classes:

Following is the list of commonly used event classes.

S. No.	Control & Description
1	<u>AWTEvent</u> It is the root event class for all AWT events. This class and its subclasses supersede the original <code>java.awt.Event</code> class.
2	<u>ActionEvent</u> The <code>ActionEvent</code> is generated when button is clicked or the item of a list is double clicked.
3	<u>InputEvent</u> The <code>InputEvent</code> class is root event class for all component-level input events.
4	<u>KeyEvent</u> On entering the character the <code>Key</code> event is generated.
5	<u>MouseEvent</u>

	This event indicates a mouse action occurred in a component.
6	<u>TextEvent</u> The object of this class represents the text events.
7	<u>WindowEvent</u> The object of this class represents the change in state of a window.
8	<u>AdjustmentEvent</u> The object of this class represents the adjustment event emitted by Adjustable objects.
9	<u>ComponentEvent</u> The object of this class represents the change in state of a window.
10	<u>ContainerEvent</u> The object of this class represents the change in state of a window.
11	<u>MouseMotionEvent</u> The object of this class represents the change in state of a window.
12	<u>PaintEvent</u> The object of this class represents the change in state of a window.

Event classes and Listener interfaces:

Event Classes

ActionEvent
MouseEvent
MouseWheelEvent
KeyEvent
ItemEvent I
TextEvent
AdjustmentEvent
WindowEvent
ComponentEvent
ContainerEvent
FocusEvent

Listener Interfaces

ActionListener
MouseListener and MouseMotionListener
MouseWheelListener
KeyListener
ItemListener
TextListener
AdjustmentListener
WindowListener
ComponentListener
ContainerListener
FocusListener

Steps to perform Event Handling:

Following steps are required to perform event handling:

1. Implement the Listener interface and overrides its methods
2. Register the component with the Listener

For registering the component with the Listener, many classes provide the registration methods.

For example:

Button:

```
public void addActionListener(ActionListener a){ }
```

MenuItem:

```
public void addActionListener(ActionListener a){ }
```

TextField:

```
public void addActionListener(ActionListener a){ }
```

```
public void addTextListener(TextListener a){ }
```

TextArea:

```
public void addTextListener(TextListener a){ }
```

Checkbox:

```
public void addItemListener(ItemListener a){ }
```

Choice:

```
public void addItemListener(ItemListener a){ }
```

List:

```
public void addActionListener(ActionListener a){ }
```

```
public void addItemListener(ItemListener a){ }
```

5.11 Handling Mouse and Keyboard events:

Java MouseListener Interface:

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface:

The signature of 5 methods found in MouseListener interface are given below:

1. public abstract void mouseClicked(MouseEvent e);
2. public abstract void mouseEntered(MouseEvent e);
3. public abstract void mouseExited(MouseEvent e);
4. public abstract void mousePressed(MouseEvent e);
5. public abstract void mouseReleased(MouseEvent e);

Java MouseListener Example:

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener{
Label l;
MouseListenerExample(){
addMouseListener(this);
l=new Label();
l.setBounds(20,50,100,20);
```

```

add(l);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void mouseClicked(MouseEvent e) {
l.setText("Mouse Clicked");
}
public void mouseEntered(MouseEvent e) {
l.setText("Mouse Entered");
}
public void mouseExited(MouseEvent e) {
l.setText("Mouse Exited");
}
public void mousePressed(MouseEvent e) {
l.setText("Mouse Pressed");
}
public void mouseReleased(MouseEvent e) {
l.setText("Mouse Released");
}
public static void main(String[] args) {
new MouseListenerExample();
}

```

Java KeyListener Interface:

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package. It has three methods. Methods of KeyListener interface

The signature of 3 methods found in KeyListener interface are given below:

1. public abstract void keyPressed(KeyEvent e);
2. public abstract void keyReleased(KeyEvent e);
3. public abstract void keyTyped(KeyEvent e);

Java KeyListener Example

```

import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener{
Label l;
TextArea area;
KeyListenerExample(){
l=new Label();
l.setBounds(20,50,100,20);
area=new TextArea();
area.setBounds(20,80,300, 300);
}

```

```

area.addKeyListener(this);
add(l);
add(area);
setSize(400,400);
setLayout(null);
setVisible(true);
}
public void keyPressed(KeyEvent e) {
l.setText("Key Pressed");
}
public void keyReleased(KeyEvent e) {
l.setText("Key Released");
}
public void keyTyped(KeyEvent e) {
l.setText("Key Typed");
}
public static void main(String[] args) {
new KeyListenerExample(); } }

```

5.12 Java Adapter Classes:

Java adapter classes provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Java WindowAdapter Example

```

1. import java.awt.*;
import java.awt.event.*;
public class AdapterExample{
Frame f;
AdapterExample(){
f=new Frame("Window Adapter");

```

```
f.addWindowListener(new WindowAdapter(){  
    public void windowClosing(WindowEvent e) {  
        f.dispose(); } };  
    e.setSize(400,400);  
    f.setLayout(  
    null);  
    f.setVisible(  
    true);  
}  
public static void main(String[] args) {  
  
    new AdapterExample();  
} }
```

5.13 Inner class in java:

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

- 1) Nested Inner class
- 2) Method Local inner classes
- 3) Anonymous inner classes
- 4) Static nested classes

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier. Like class, interface can also be nested and can have access specifiers.

Following example demonstrates a nested class.

```
classOuter {  
  
    // Simple nested inner class  
  
    classInner {  
  
        public void show() {  
  
            System.out.println("In a nested class method");  
  
        }  
  
    }  
}
```

```

classMain
{
    public static void main(String[] args)
    {
        Outer.Inner in = new Outer().new Inner();
        in.show();
    }
}

```

5.14 Anonymous Inner Class:

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows –

Syntax:

```

AnonymousInner an_inner = new AnonymousInner() {
    public void my_method() {
        .....
        .....
    }
};

```

The following program shows how to override the method of a class using anonymous inner class.

Example

```

abstract class AnonymousInner{
    public abstract void mymethod();
}

public class Outer_class{
    public static void main(String args[]){
        AnonymousInner inner = new AnonymousInner(){
            public void mymethod(){
                System.out.println("This is an example of anonymous inner class");
            }
        };
    }
}

```

```
}

};

inner.mymethod();

}

}
```

If you compile and execute the above program, you will get the following result –

Output:

This is an example of anonymous inner class

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument –

```
obj.myMethod(new My_Class() {
    public void Do() {
        ....
        ....
    }
});
```

The following program shows how to pass an anonymous inner class as a method argument.

Example

```
// interface

interface Message

{
```

```
    String greet();

}
```



```
public class My_class

{
```



```
    // method which accepts the object of interface Message
```

```
public void displayMessage(Message m)
{
    System.out.println(m.greet() +
        ", This is an example of anonymous inner class as an argument");

}

public static void main(String args[]) {
    // Instantiating the class
    My_class obj = new My_class();

    // Passing an anonymous inner class as an argument
    obj.displayMessage(new Message() {
        public String greet() {
            return "Hello";
        }
    });
}
```

If you compile and execute the above program, it gives you the following result –

Output

Hello, This is an example of anonymous inner class as an argument

5.15 A Simple Swing Application:

Java Swing components are basic building blocks of a Java Swing application. In this chapter we will use `JFrame`, `JButton`, and `JLabel` components. `JFrame` is a top-level window with a title and a border. It is used to organize other components, commonly referred to as child components.

`JButton` is a push button used to perform an action. `JLabel` is a component used to display a short text string or an image, or both.

Java Swing first example:

The first example shows a basic window on the screen.

```
import java.awt.EventQueue;  
  
import javax.swing.JFrame;  
  
public class SimpleEx extends JFrame {  
  
    public SimpleEx() {  
  
        initUI();  
    }  
  
    private void initUI() {  
  
        setTitle("Simple example");  
        setSize(300, 200);  
        setLocationRelativeTo(null);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
    }  
  
    public static void main(String[] args) {  
  
        EventQueue.invokeLater(() ->  
        {  
  
            var ex = new SimpleEx();  
            ex.setVisible(true);  
        });  
    }  
}
```

While this code is very short, the application window can do quite a lot. It can be resized, maximised, or minimised. All the complexity that comes with it has been hidden from the application programmer.

```
import java.awt.EventQueue;
import javax.swing.JFrame;
```

Here we import Swing classes that will be used in the code example.

```
public class SimpleEx extends JFrame {
```

`SimpleEx` inherits from the `JFrame` component. `JFrame` is a top-level container. The basic purpose of containers is to hold components of the application.

```
public SimpleEx() {
    initUI();
}
```

It is a good programming practice not to put the application code into constructors, but delegate the task to a specific method.

```
setTitle("Simple example");
```

Here we set the title of the window using the `setTitle()` method.

```
setSize(300, 200);
```

The `setSize()` resizes the window to be 300 px wide and 200 px tall.

```
setLocationRelativeTo(null);
```

This line centers the window on the screen.

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

This `setDefaultCloseOperation()` closes the window if we click on the Close button of the titlebar. By default nothing happens if we click on the button.

```
EventQueue.invokeLater(() -> {
    var ex = new SimpleEx();
    ex.setVisible(true);
});
```

We create an instance of our code example and make it visible on the screen. The `invokeLater()` method places the application on the Swing Event Queue. It is used to ensure that all UI updates are concurrency-safe. In other words, it is to prevent GUI from hanging in certain situations.

5.16 Applets: Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side. Advantage of Applet: There are many advantages of applet. They are as follows:

- o It works at client side so less response time.

Lifecycle of Java Applet Hierarchy of Applet:

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

3. public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. public void destroy(): is used to destroy the Applet. It is invoked only once.

java.awt.Component class:

The Component class provides 1 life cycle method of applet.

1. public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
1. //First.java

import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
    public void paint(Graphics g){
        g.drawString("welcome",150,150);
    }
}
```

Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
1. //First.java

import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
} }
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
```

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax:

```
public String getParameter(String parameterName)
```

Example of using parameter in Applet:

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet
```

```
{  
public void paint(Graphics g)  
{  
String str=getParameter("msg");  
g.drawString(str,50, 50);  
}  
}  
  
myapplet.html  
  
<html>  
<body>  
<applet code="UseParam.class" width="300" height="300">  
<param name="msg" value="Welcome to applet">  
</applet>  
</body>  
</html>
```

5.17. Security Issues with the Applet:

Java applet is run inside a web browser. But an applet is restricted in some areas, until it has been deemed trustworthy by the end user. The security restriction is provided for protecting the user by malicious code, like copy important information from the hard disk or deleting the files. Generally, applets are loaded from the Internet and they are prevented from: the writing and reading the files on client side. Some security issues to applet are following :

Applets are loaded over the internet and they are prevented to make open network connection to any computer, except for the host, which provided the .class file. Because the html page come from the host or the host specified codebase parameter in the applet tag, with codebase taking precedence.

They are also prevented from starting other programs on the client. That means any applet, which you visited, cannot start any rogue process on you computer. In UNIX, applets cannot start any exec or fork processes. Applets are not allowed to invoke any program to list the contents of your file system that means it cant invoke System.exit() function to terminate you web browser. And they are not allowed to manipulate the threads outside the applets own thread group.

Applets are loaded over the net. A web browser uses only one class loader that is established at start up. Then the system class loader can not be overloaded, overridden, extended, replaced. Applet is not allowed to create the reference of their own class loader.

They can't load the libraries or define the native method calls. But if it can define native method calls then that would give the applet direct access to the underlying computer.

5.18) Applets and Applications:

Definition of Application and Applet – Applets are feature rich application programs that are specifically designed to be executed within an HTML web document to execute small tasks or just part of it. Java applications, on the other hand, are stand-alone programs that are designed to run on a stand-alone machine without having to use a browser.

Execution of Application and Applet – Applications require main method() to execute the code from the command line, whereas an applet does not require main method() for execution. An applet requires an HTML file before its execution. The browser, in fact, requires a Java plugin to run an applet.

Compilation of Application and Applet – Application programs are compiled using the “javac” command and further executed using the java command. Applet programs, on the other hand, are also compiled using the “javac” command but are executed either by using the “appletviewer” command or using the web browser.

Security Access of Application and Applet – Java application programs can access all the resources of the system including data and information on that system, whereas applets cannot access or modify any resources on the system except only the browser specific services.

Restrictions of Application and Applet – Unlike applications, applet programs cannot be run independently, thus require highest level of security. However, they do not require any specific deployment procedure during execution. Java applications, on the other hand, run independently and do not require any security as they are trusted.

Difference between Applet and Application programming:

Java Application	Applet
Java application contains a main method	An applet does not contain a main method
Does not require internet connection to execute	Requires internet connection to execute
Is stand alone application	Is a part of web page
Can be run without a browser	Requires a Java compatible browser
Uses stream I/O classes	Use GUI interface provided by AWT or Swing
Entry point is main method	Entry point is init method
Generally used for console programs	Generally used for GUI interfaces

5.19) Passing Parameters to Applet:

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax:

```
public String getParameter(String parameterName)
```

Example of using parameter in Applet:

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet{
public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}}
```

myapplet.html:

```
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
```

```
</applet>  
</body>  
</html>
```

5.20 Creating a swing applet:

We can even create applets based on the Swing package. In order to create such applets, we must extend JApplet class of the swing package. JApplet extends Applet class, hence all the features of Applet class are available in JApplet as well, including JApplet's own Swing based features. Swing applets provides an easier to use user interface than AWT applets.

Here is the code of the program :

```
import javax.swing.*;  
import java.applet.*;  
import java.awt.*;  
import java.awt.event.*;  
  
public class SApplet extends Applet implements ActionListener {  
    TextField input,output;  
    Label label1,label2;  
    Button b1;  
    JLabel lbl;  
    int num, sum = 0;  
    public void init(){  
        label1 = new Label("please enter number : ");  
        add(label1);  
        label1.setBackground(Color.yellow);  
        label1.setForeground(Color.magenta);  
        input = new TextField(5);  
        add(input);  
        label2 = new Label("Sum : ");  
        add(label2);  
        label2.setBackground(Color.yellow);  
        label2.setForeground(Color.magenta);  
        output = new TextField(20);  
        add(output);  
        // input.addActionListener( this );  
        b1 = new Button("Add");  
        add(b1);  
        b1.addActionListener(this);  
        lbl = new JLabel("This is the Swing Applet Example. ");  
        add(lbl);
```

```
setBackground(Color.yellow);

// output.addActionListener( this );
}

public void actionPerformed(ActionEvent ae){
try{
// num = Integer.parseInt(ae.getActionCommand());
num = Integer.parseInt(input.getText());
sum = sum+num;
input.setText("");
output.setText(Integer.toString(sum));
lbl.setForeground(Color.blue);
lbl.setText("Output of the second Text Box : " + output.getText());
}

catch(NumberFormatException e){
lbl.setForeground(Color.red);
lbl.setText("Invalid Entry!");
}

}

}
```

Here is the HTML code :

```
<html>
<body>
<applet code = "SApplet" width = "260"height = "200"></applet>
</body>
</html>
```

5.21 Painting in Swing:

Painting Swing components is based on the AWT callback method, so it supports the paint and repaint methods. Swing painting also extends the functionality of paint operations with a number of additional features.

The Swing API, `RepaintManager`, can be used to customize the painting of Swing components. In addition, Swing painting supports Swing structures, such as borders and double-buffering.

Double-buffering is supported by default. Removing double-buffering is not recommended.

Because lightweight components are contained within heavyweight components, the painting of lightweight components is triggered by the paint method of the heavyweight component.

When the paint method is called, it is translated to all lightweight components using the `java.awt.Container` class's paint method. This causes all defined areas to be repainted.

There are three customized callbacks for Swing components, which factor out a single paint method into three subparts. These are

paintComponent()

paintBorder()

paintChildren()

paintComponent():

You use the paintComponent method to call the UI delegate object's paint method. The paintComponent method passes a copy of the Graphics object to the UI delegate object's paint method. This protects the rest of the paint code from irrevocable changes.

You cannot call the paintComponent method if UI delegate is set to null.

paintBorder():

You use the paintBorder method to paint a component's border.

paintChildren():

You use the paintChildren method to paint a component's child components.

You need to bear several factors in mind when designing a Swing paint operation.

Firstly, application-triggered and system-triggered requests call the paint or repaint method of a Swing component, but never the update method. Also, the paint method is never called directly. You can trigger a future call to it by invoking the repaint method.

As with AWT components, it is good practice to define the clip rectangle using the arguments of the repaint method. Using the clip rectangle to narrow down the area to be painted makes the code more efficient.

You can customize the painting of Swing components using two properties, namely

opaque

optimizedDrawingEnabled

opaque:

You use the opaque property to clear the background of a component and repaint everything contained within the paintComponent method. To do this, opaque must be set to true, which is the default. Setting opaque to true reduces the amount of screen clutter caused by repainting a component's elements.

optimizedDrawingEnabled

The optimizedDrawingEnabled property controls whether components can overlap. The default value of the optimizedDrawingEnabled property is true.

Setting either the opaque or the optimizedDrawingEnabled property to false is not advised unless absolutely necessary, as it causes a large amount of processing.

You use the paintComponent method for Swing component extensions that implement their own paint code. The scope of these component extensions need to be set in the paintComponent method.

5.22 Paint

Example:

The paintComponent method is structured in the same way as the paint method.

```
// Custom JPanel with overridden paintComponent method
```

```
class MyPanel extends JPanel {  
    public MyPanel(LayoutManager layout) {  
        super (layout) ;  
    }  
    public void paintComponent(Graphics g) {  
        // Start by clearing the current screen  
        g.clearRect( getX(), getY(), getWidth(), getHeight());  
        g.setColor (Color.red) ;  
        int[] x = {30, 127, 56, 355, 240, 315} ;  
        int[] y = {253, 15, 35, 347, 290, 265} ;  
        //Draw complex shape on-screen and fill it  
        g.drawPolygon (x, y, 5);  
        g.fillPolygon (x, y, 5);  
    }  
}
```

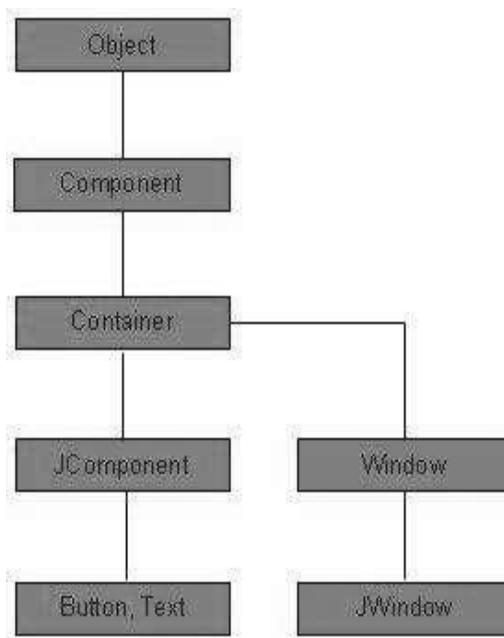
5.23) Swing controls in java:

Every user interface considers the following three main aspects –

UI Elements – These are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex, which we will cover in this tutorial.

Layouts – They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in the Layout chapter.

Behavior – These are the events which occur when the user interacts with UI elements. This part will be covered in the Event Handling chapter.



Every SWING controls inherits properties from the following Component class hierarchy.

S.No.	Class & Description
1	<u>Component</u> A Component is the abstract base class for the non menu user-interface controls of SWING. Component represents an object with graphical representation
2	<u>Container</u> A Container is a component that can contain other SWING components
3	<u>JComponent</u> A JComponent is a base class for all SWING UI components. In order to use a SWING component that inherits from JComponent, the component must be in a containment hierarchy whose root is a top-level SWING container

SWING UI Elements

Following is the list of commonly used controls while designing GUI using SWING.

S.No.	Class & Description
1	<u>JLabel</u> A JLabel object is a component for placing text in a container.
2	<u>JButton</u> This class creates a labeled button.
3	<u>JColorChooser</u> A JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.
4	<u>JCheckBox</u> A JCheckBox is a graphical component that can be in either an on (true) or off (false) state.
5	<u>JRadioButton</u> The JRadioButton class is a graphical component that can be in either an on (true) or off (false) state. in a group.
6	<u>JList</u> A JList component presents the user with a scrolling list of text items.
7	<u>JComboBox</u> A JComboBox component presents the user with a to show up menu of choices.
8	<u>JTextField</u> A JTextField object is a text component that allows for the editing of a single line of text.
9	<u>JPasswordField</u> A JPasswordField object is a text component specialized for password entry.
10	<u>JTextArea</u> A JTextArea object is a text component that allows editing of a multiple lines of text.
11	<u>ImageIcon</u> A ImageIcon control is an implementation of the Icon interface that paints Icons from Images
12	<u>JScrollbar</u> A Scrollbar control represents a scroll bar component in order to enable the user to select from range of values.
13	<u>JOptionPane</u> JOptionPane provides set of standard dialog boxes that prompt users for a value or informs them of something.
14	<u>JFileChooser</u>

	A JFileChooser control represents a dialog window from which the user can select a file.
15	<u>JProgressBar</u> As the task progresses towards completion, the progress bar displays the task's percentage of completion.
16	<u>JSlider</u> A JSlider lets the user graphically select a value by sliding a knob within a bounded interval.
17	<u>JSpinner</u> A JSpinner is a single line input field that lets the user select a number or an object value from an ordered sequence.

5.23.1 JLabel and ImageIcon:

JLabel is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at JLabel a bit more closely. JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. JLabel defines several constructors. Here are three of them:

`JLabel(Icon icon)` `JLabel(String str)`

`JLabel(String str, Icon icon, int align)`

Here, str and icon are the text and icon used for the label. The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: LEFT, RIGHT, CENTER, LEADING, or TRAILING. These constants are defined in the SwingConstants interface, along with several others used by the Swing classes.

Notice that icons are specified by objects of type Icon, which is an interface defined by Swing. The easiest way to obtain an icon is to use the ImageIcon class. ImageIcon implements Icon and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the ImageIcon constructor used by the example in this section:

`ImageIcon(String filename)`

It obtains the image in the file named filename.

The icon and text associated with the label can be obtained by the following methods:

`Icon getIcon()` `String getText()`

The icon and text associated with a label can be set by these methods:

`void setIcon(Icon icon)` `void setText(String str)`

Here, icon and str are the icon and text, respectively. Therefore, using setText() it is possible to change the text inside a label during program execution.

The following applet illustrates how to create and display a label containing both an icon and a string. It begins by creating an ImageIcon object for the file hourglass.png, which depicts an hourglass. This is used as the second argument to the JLabel constructor. The first and last arguments for the JLabel constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
// Demonstrate JLabel and ImageIcon.

import java.awt.*;
import javax.swing.*;

<applet code="JLabel" width=250 height=200></applet>

/*
public class JLabelDemo extends JApplet {

    public void init() { try {

        SwingUtilities.invokeAndWait( new Runnable() {

            public void run() { makeGUI();

            }

        }

    }

    );} catch (Exception exc) {

        System.out.println("Can't create because of " + exc);

    }

}

private void makeGUI() {

    // Create an icon.

    ImageIcon ii = new ImageIcon("hourglass.png");

    // Create a label.

    JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);

    // Add the label to the content pane.

    add(jl);

}
```

```
 }  
 }
```

5.23.2 JTextField:

JTextField is the simplest Swing text component. It is also probably its most widely used text component. JTextField allows you to edit one line of text. It is derived from JTextComponent, which provides the basic functionality common to Swing text components. JTextField uses the Document interface for its model. Three of JTextField's constructors are shown here:

```
JTextField(int cols) JTextField(String str, int cols) JTextField(String str)
```

Here, str is the string to be initially presented, and cols is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.

JTextField generates events in response to user interaction. For example, an ActionEvent is fired when the user presses enter. A CaretEvent is fired each time the caret (i.e., the cursor) changes position. (CaretEvent is packaged in javax.swing.event.) Other events are also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call getText().

The following example illustrates JTextField. It creates a JTextField and adds it to the content pane. When the user presses enter, an action event is generated. This is handled by displaying the text in the status window.

```
// Demonstrate JTextField.  
  
import java.awt.*;  
  
import java.awt.event.*;  
  
import javax.swing.*; /*  
  
<applet code="JTextFieldDemo" width=300 height=50></applet>  
  
*/  
  
public class JTextFieldDemo extends JApplet { JTextField jtf;  
  
public void init() { try {  
  
SwingUtilities.invokeAndWait( new Runnable() {  
  
public void run() { makeGUI();
```

```
}

}

);

} catch (Exception exc) {

System.out.println("Can't create because of " + exc);

}

}

private void makeGUI() {

//Change to flow layout.

setLayout(new FlowLayout());

//Add text field to content pane.

jtf = new JTextField(15); add(jtf);

jtf.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent ae)

{

//Show text when user presses ENTER.

showStatus(jtf.getText());

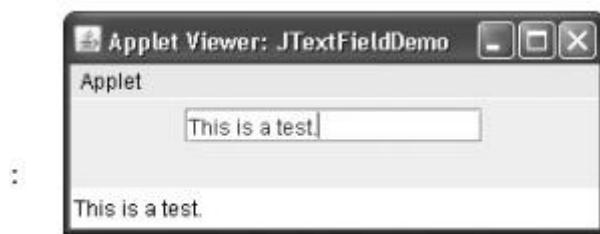
}

}

);

}
```

Output from the text field example is shown here:



5.23.3 The Swing Buttons:

Swing defines four types of buttons: JButton, JToggleButton, JCheckBox, and JRadioButton. All are subclasses of the AbstractButton class, which extends JComponent. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a rollover icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```
void setDisabledIcon(Icon di) void setPressedIcon(Icon pi) void setSelectedIcon(Icon si) void
setRolloverIcon(Icon ri)
```

Here, di, pi, si, and ri are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

```
String getText()
```

```
void setText(String str)
```

Here, str is the text to be associated with the button.

The model used by all buttons is defined by the ButtonModel interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

5.23.4 JButton:

The JButton class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. JButton allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

```
JButton(Icon icon) JButton(String str) JButton(String str, Icon icon)
```

Here, str and icon are the string and icon used for the button.

When the button is pressed, an ActionEvent is generated. Using the ActionEvent object passed to the actionPerformed() method of the registered ActionListener, you can obtain the action command string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling setActionCommand() on the button. You can obtain the action command by calling getActionCommand() on the event object. It is declared like this:

```
String getActionCommand()
```

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button displays an icon that represents a timepiece. When a button is pressed, the name of that timepiece is displayed in the label.

// Demonstrate an icon-based JButton.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*; /*

<applet code="JButtonDemo" width=250 height=750></applet>
*/
public class JButtonDemo extends JApplet implements ActionListener {
JLabel jlab;
public void init() { try {
SwingUtilities.invokeAndWait( new Runnable() {
public void run() { makeGUI();
}
}
);
}catch (Exception exc) {
System.out.println("Can't create because of " + exc);
}
}
private void makeGUI() {
Change to flow layout.setLayout(new FlowLayout());
Add buttons to content pane.
ImageIcon hourglass = new ImageIcon("hourglass.png");
JButton jb = new JButton(hourglass);
jb.setActionCommand("Hourglass");
}
```

```
jb.addActionListener(this);

add(jb);

ImageIcon analog = new ImageIcon("analog.png");

jb = new JButton(analog);

jb.setActionCommand("Analog Clock");

jb.addActionListener(this);

add(jb);

ImageIcon digital = new ImageIcon("digital.png");

jb = new JButton(digital);

jb.setActionCommand("Digital Clock");

jb.addActionListener(this);

add(jb);

ImageIcon stopwatch = new ImageIcon("stopwatch.png");

jb = new JButton(stopwatch);

jb.setActionCommand("Stopwatch"); jb.addActionListener(this);

add(jb);

// Create and add the label to content pane.

jlab = new JLabel("Choose a Timepiece"); add(jlab);

}

// Handle button events.

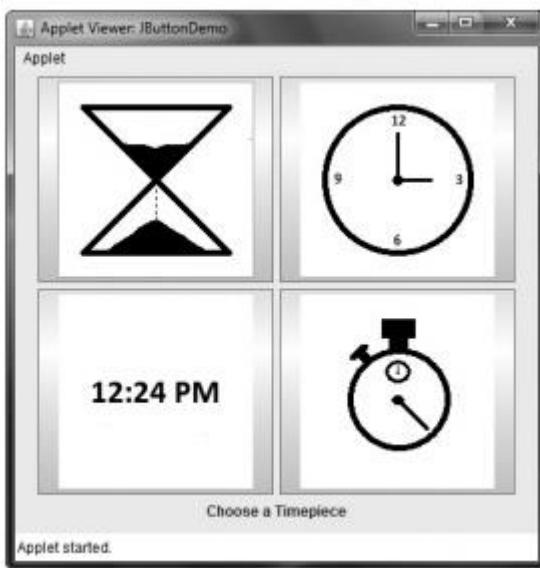
public void actionPerformed(ActionEvent ae) {

jlab.setText("You selected " + ae.getActionCommand());

}

}
```

Output from the button example is shown here:



5.23.5 JToggleButton:

A useful variation on the push button is called a toggle button. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

Toggle buttons are of the `JToggleButton` class. `JToggleButton` implements `AbstractButton`. In addition to creating standard toggle buttons, `JToggleButton` is a superclass for two other Swing components that also represent two-state controls. These are `JCheckBox` and `JRadioButton`, which are described later in this chapter. Thus, `JToggleButton` defines the basic functionality of all two-state components.

`JToggleButton` defines several constructors. The one used by the example in this section is shown here:

```
JToggleButton(String str)
```

This creates a toggle button that contains the text passed in `str`. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

`JToggleButton` uses a model defined by a nested class called `JToggleButton.ToggleButtonModel`. Normally, you won't need to interact directly with the model to use a standard toggle button.

Like JButton, JToggleButton generates an action event each time it is pressed. Unlike JButton, however, JToggleButton also generates an item event. This event is used by those components that support the concept of selection. When a JToggleButton is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the ItemListener interface. Recall from Chapter 24, that each time an item event is generated, it is passed to the itemStateChanged() method defined by ItemListener. Inside itemStateChanged(), the getItem() method can be called on the ItemEvent object to obtain a reference to the JToggleButton instance that generated the event. It is shown here:

```
Object getItem( )
```

A reference to the button is returned. You will need to cast this reference to JToggleButton. The easiest way to determine a toggle button's state is by calling the isSelected() method

(inherited from AbstractButton) on the button that generated the event. It is shown here: boolean isSelected()

It returns true if the button is selected and false otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls isSelected() to determine the button's state.

```
// Demonstrate JToggleButton.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

<applet code="JToggleButtonDemo" width=200 height=80></applet>

*

public class JToggleButtonDemo extends JApplet {
    JLabel jlab;
    JToggleButton jtbn;
    public void init() { try {
        SwingUtilities.invokeAndWait( new Runnable() {
            public void run() { makeGUI();
            }
        });
    }
}
```

```
    } catch (Exception exc) {  
        System.out.println("Can't create because of " + exc);  
    }  
}  
  
private void makeGUI() {  
    //Change to flow layout.  
    setLayout(new FlowLayout());  
  
    //Create a label.  
    jlab = new JLabel("Button is off.");  
  
    // Make a toggle button.  
    jtbn = new JToggleButton("On/Off");  
  
    // Add an item listener for the toggle button.  
    jtbn.addItemListener(new ItemListener() {  
        public void itemStateChanged(ItemEvent ie) {  
            if(jtbn.isSelected())  
                jlab.setText("Button is on.");  
            else  
                jlab.setText("Button is off.");  
        }  
    });  
  
    // Add the toggle button and label to the content pane.  
    add(jtbn);  
    add(jlab);  
}  
}
```

The output from the toggle button example is shown here:



5.23.6 JCheck Box:

The JCheckBox class provides the functionality of a check box. Its immediate superclass is JToggledbutton, which provides support for two-state buttons, as just described. JCheckBox defines several constructors. The one used here is

`JCheckBox(String str)`

It creates a check box that has the text specified by str as a label. Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a check box, an ItemEvent is generated. You can obtain a reference to the JCheckBox that generated the event by calling `getItem()` on the

ItemEvent passed to the `itemStateChanged()` method defined by ItemListener. The easiest way to determine the selected state of a check box is to call `isSelected()` on the JCheckBox instance.

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an ItemEvent is generated. Inside the `itemStateChanged()` method, `getItem()` is called to obtain a reference to the JCheckBox object that generated the event. Next, a call to `isSelected()` determines if the box was selected or cleared. The `getText()` method gets the text for that check box and uses it to set the text inside the label.

```
// Demonstrate JCheckbox.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/*<applet code="JCheckBoxDemo" width=270 height=50></applet>

*/
public class JCheckBoxDemo extends JApplet implements ItemListener {
    JLabel jlab;
```

```
public void init() { try {  
    SwingUtilities.invokeAndWait( new Runnable() {  
        public void run() {  
            makeGUI();  
        }  
    });  
}  
} catch (Exception exc) {  
    System.out.println("Can't create because of " + exc);  
}  
}  
  
private void makeGUI() {  
    //Change to flow layout.  
    setLayout(new FlowLayout());  
  
    //Add check boxes to the content pane.  
    JCheckBox cb = new JCheckBox("C"); cb.addItemListener(this);  
    add(cb);  
  
    cb = new JCheckBox("C++");  
    cb.addItemListener(this); add(cb);  
  
    cb = new JCheckBox("Java");  
    cb.addItemListener(this); add(cb);  
  
    cb = new JCheckBox("Perl");  
    cb.addItemListener(this); add(cb);  
  
    // Create the label and add it to the content pane.  
    jlab = new JLabel("Select languages");  
    add(jlab);  
}  
  
// Handle item events for the check boxes.  
  
public void itemStateChanged(ItemEvent ie) {
```

```
JCheckBox cb = (JCheckBox)ie.getItem();
if(cb.isSelected())
    jlab.setText(cb.getText() + " is selected");
else
    jlab.setText(cb.getText() + " is cleared");
}
}
```

Output from this example is shown here:



5.23.7 JRadio Buttons:

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the JRadioButton class, which extends JToggleButton. JRadioButton provides several constructors. The one used in the example is shown here:

```
JRadioButton(String str)
```

Here, str is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the ButtonGroup class. Its default constructor is invoked for this purpose. Elements are

then added to the button group via the following method: void add(AbstractButton ab)

Here, ab is a reference to the button to be added to the group.

A JRadioButton generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the ActionListener interface. Recall that the only method defined by ActionListener is actionPerformed(). Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by

calling `getActionCommand()`. By default, the action command is the same as the button label, but you can set the action command to something else by calling `setActionCommand()` on the radio button. Second, you can call `getSource()` on the `ActionEvent` object and check that reference against the buttons. Third, you can check each radio button to find out which one is currently selected by calling `isSelected()` on each button. Finally, each button could use its own action event handler implemented as either an anonymous inner class or a lambda expression. Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by `actionPerformed()`. Within that handler, the `getActionCommand()` method gets the text that is associated with the radio button and uses it to set the text within a label.

```
// Demonstrate JRadioButton

import java.awt.*;
import java.awt.event.*;
import javax.swing.*; /*

<applet code="JRadioButtonDemo" width=300 height=50></applet>

*/
public class JRadioButtonDemo extends JApplet implements ActionListener {
    JLabel jlab;
    public void init() { try {
        SwingUtilities.invokeAndWait( new Runnable() {
            public void run() {
                makeGUI();
            }
        });
    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}
```

```
private void makeGUI() {  
    //Change to flow layout.  
    setLayout(new FlowLayout());  
  
    //Create radio buttons and add them to content pane.  
    JRadioButton b1 = new JRadioButton("A");  
    b1.addActionListener(this);  
    add(b1);  
  
    JRadioButton b2 = new JRadioButton("B");  
    b2.addActionListener(this);  
    add(b2);  
  
    JRadioButton b3 = new JRadioButton("C");  
    b3.addActionListener(this);  
    add(b3);  
  
    //Define a button group.  
    ButtonGroup bg = new ButtonGroup();  
    bg.add(b1);  
    bg.add(b2);  
    bg.add(b3);  
  
    //Create a label and add it to the content pane.  
    jlab = new JLabel("Select One");  
    add(jlab);  
}  
  
// Handle button selection.  
  
public void actionPerformed(ActionEvent ae) {  
    jlab.setText("You selected " + ae.getActionCommand());  
}  
}
```

Output from the radio button example is shown here:



5.23.8 JTabbedPane:

JTabbedPane encapsulates a tabbed pane. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront. Tabbed panes are very common in the modern GUI, and you have no doubt used them many times. Given the complex nature of a tabbed pane, they are surprisingly easy to create and use.

JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides. JTabbedPane uses the SingleSelectionModel model.

Tabs are added by calling addTab(). Here is one of its forms: void addTab(String name, Component comp)

Here, name is the name for the tab, and comp is the component that should be added to the tab. Often, the component added to a tab is a JPanel that contains a group of related components. This technique allows a tab to hold a set of components.

The general procedure to use a tabbed pane is outlined here:

Create an instance of JTabbedPane.

Add each tab by calling addTab().

Add the tabbed pane to the content pane.

The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color. The third tab is titled "Flavors" and contains one combo box. This enables the user to select one of three flavors.

```
// Demonstrate JTabbedPane.
```

```
import javax.swing.*;  
/*<applet code="JTabbedPaneDemo" width=400 height=100></applet>  
*/
```

```
public class JTabbedPaneDemo extends JApplet {  
    public void init() { try {  
        SwingUtilities.invokeAndWait( new Runnable() {  
            public void run() { makeGUI();  
            }  
        });  
    } catch (Exception exc) {  
        System.out.println("Can't create because of " + exc);  
    }  
    private void makeGUI() {  
        JTabbedPane jtp = new JTabbedPane();  
        jtp.addTab("Cities", new CitiesPanel());  
        jtp.addTab("Colors", new ColorsPanel());  
        jtp.addTab("Flavors", new FlavorsPanel());  
        add(jtp);  
    }  
}  
// Make the panels that will be added to the tabbed pane.  
class CitiesPanel extends JPanel {  
    public CitiesPanel() {  
        JButton b1 = new JButton("New York");  
        add(b1);  
        JButton b2 = new JButton("London");  
        add(b2);  
        JButton b3 = new JButton("Hong Kong");  
        add(b3);  
    }  
}
```

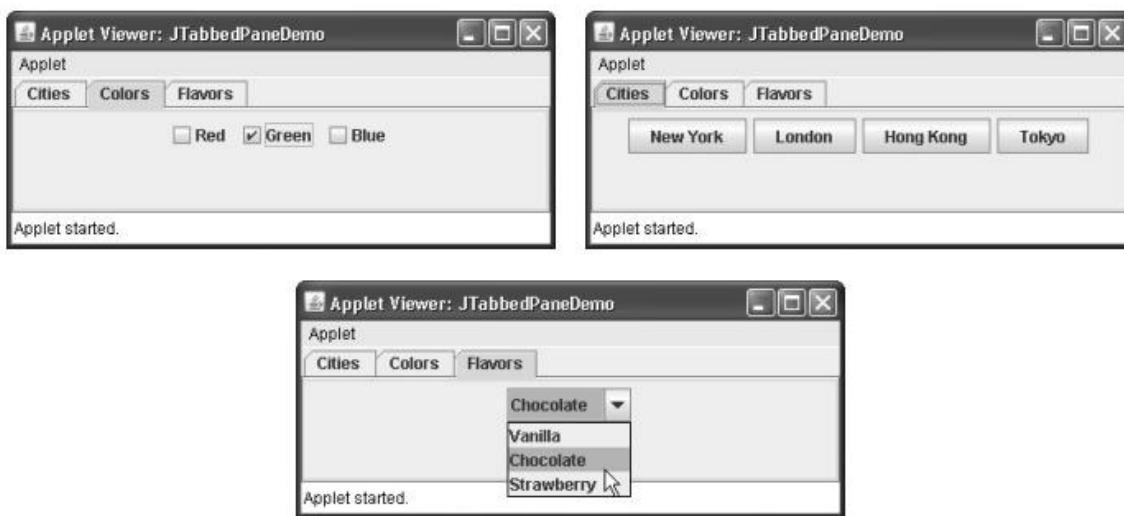
```
 JButton b4 = new JButton("Tokyo");
add(b4);
}

}

class ColorsPanel extends JPanel {
public ColorsPanel() {
JCheckBox cb1 = new JCheckBox("Red");
add(cb1);
JCheckBox cb2 = new JCheckBox("Green");
add(cb2);
JCheckBox cb3 = new JCheckBox("Blue");
add(cb3);
}
}

class FlavorsPanel extends JPanel {
public FlavorsPanel() {
JComboBox<String> jcb = new JComboBox<String>();
jcb.addItem("Vanilla"); jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}
}
```

Output from the tabbed pane example is shown in the following three illustrations:



5.23.9 JScrollPane:

JScrollPane is a lightweight container that automatically handles the scrolling of another component. The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a JPanel. In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because JScrollPane automates scrolling, it usually eliminates the need to manage individual scroll bars.

The viewable area of a scroll pane is called the viewport. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport. In its default behavior, a JScrollPane will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors. The one used in this chapter is shown here: `JScrollPane(Component comp)`

The component to be scrolled is specified by `comp`. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

Create the component to be scrolled.

Create an instance of JScrollPane, passing to it the object to scroll.

Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a JPanel object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

```
// Demonstrate JScrollPane.  
import java.awt.*;  
import javax.swing.*;  
<applet code="JScrollPaneDemo" width=300 height=250></applet>  
*/  
public class JScrollPaneDemo extends JApplet {  
    public void init() {try {  
        SwingUtilities.invokeAndWait( new Runnable() {  
            public void run() {  
                makeGUI();  
            }  
        });  
    } catch (Exception exc) {  
        System.out.println("Can't create because of " + exc);  
    }  
    }  
    private void makeGUI() {  
        // Add 400 buttons to a panel.  
        JPanel jp = new JPanel();  
        jp.setLayout(new GridLayout(20, 20));  
        int b = 0;  
        for(int i = 0; i < 20; i++) {  
            for(int j = 0; j < 20; j++) {  
                jp.add(new JButton("Button " + b));  
                ++b;  
            }  
        }  
    }  
}
```

```

//Create the scroll pane.

JScrollPane jsp = new JScrollPane(jp);

//Add the scroll pane to the content pane.

//Because the default border layout is used,
//the scroll pane will be added to the center.

add(jsp, BorderLayout.CENTER);

}

}

```

Output from the scroll pane example is shown here:



5.23.10 JList:

In Swing, the basic list class is called `JList`. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. `JList` is so widely used in Java that it is highly unlikely that you have not seen one before.

In the past, the items in a `JList` were represented as `Object` references. However, beginning with `JDK 7`, `JList` was made generic and is now declared like this:

```
class JList<E>
```

Here, `E` represents the type of the items in the list.

`JList` provides several constructors. The one used here is `JList(E[] items)`

This creates a `JList` that contains the items in the array specified by `items`.

JList is based on two models. The first is ListModel. This interface defines how access to the list data is achieved. The second model is the ListSelectionModel interface, which defines methods that determine what list item or items are selected.

Although a JList will work properly by itself, most of the time you will wrap a JList inside a JScrollPane. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the JList component.

A JList generates a ListSelectionEvent when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing ListSelectionListener. This listener specifies only one method, called valueChanged(), which is shown here:

```
void valueChanged(ListSelectionEvent le)
```

Here, le is a reference to the event. Although ListSelectionEvent does provide some methods of its own, normally you will interrogate the JList object itself to determine what has occurred. Both ListSelectionEvent and ListSelectionListener are packaged in javax.swing.event.

By default, a JList allows the user to select multiple ranges of items within the list, but you can change this behavior by calling setSelectionMode(), which is defined by JList. It is shown here:

```
void setSelectionMode(int mode)
```

Here, mode specifies the selection mode. It must be one of these values defined by

ListSelectionModel:

SINGLE_SELECTION

SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling getSelectedIndex(), shown here:

```
int getSelectedIndex()
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, -1 is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling getSelectedValue():

E getSelectedValue()

It returns a reference to the first selected value. If no value has been selected, it returns null. The following applet demonstrates a simple JList, which holds a list of cities. Each time a city is selected in the list, a ListSelectionEvent is generated, which is handled by the valueChanged() method defined by ListSelectionListener. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

// Demonstrate JList.

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="JListDemo" width=200 height=120></applet>
*/
public class JListDemo extends JApplet { JList<String> jlst;
JLabel jlab; JScrollPane jsrlp;
// Create an array of cities.
String Cities[] = { "New York", "Chicago", "Houston", "Denver", "Los Angeles", "Seattle",
"London", "Paris", "New Delhi", "Hong Kong", "Tokyo", "Sydney" };
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
});
}
}
}
```

```
catch (Exception exc)
{
    System.out.println("Can't create because of " + exc);
}

private void makeGUI() {
    //Change to flow layout.

    setLayout(new FlowLayout());

    //Create a JList.

    jlst = new JList<String>(Cities);

    //Set the list selection mode to single selection.

    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    //Add the list to a scroll pane.

    jscrlp = new JScrollPane(jlst);

    //Set the preferred size of the scroll pane.

    jscrlp.setPreferredSize(new Dimension(120, 90));

    //Make a label that displays the selection.

    jlab = new JLabel("Choose a City");

    //Add selection listener for the list.

    jlst.addListSelectionListener(new ListSelectionListener() {

        public void valueChanged(ListSelectionEvent le) {

            //Get the index of the changed item.

            int idx = jlst.getSelectedIndex();

            // Display selection, if item was selected.

            if(idx != -1)

                jlab.setText("Current selection: " + Cities[idx]);
        }
    });
}
```

```
else
// Otherwise, reprompt.

jlab.setText("Choose a City");

}
});

// Add the list and label to the content pane.

add(jscrlp);

add(jlab);

}
}
```

Output from the list example is shown here:



5.23.11 JComboBox:

Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry. You can also create a combo box that lets the user enter a selection into the text field.

In the past, the items in a JComboBox were represented as Object references. However, beginning with JDK 7, JComboBox was made generic and is now declared like this:

```
class JComboBox<E>
```

Here, E represents the type of the items in the combo box.

The JComboBox constructor used by the example is shown here: JComboBox(E[] items)

Here, items is an array that initializes the combo box. Other constructors are available. JComboBox uses the ComboBoxModel. Mutable combo boxes (those whose entries can be changed) use the MutableComboBoxModel.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the addItem() method, shown here:

```
void addItem(E obj)
```

Here, obj is the object to be added to the combo box. This method must be used only with mutable combo boxes.

JComboBox generates an action event when the user selects an item from the list. JComboBox also generates an item event when the state of selection changes, which occurs when an item is selected or deselected. Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.

One way to obtain the item selected in the list is to call getSelectedItem() on the combo box. It is shown here:

```
Object getSelectedItem()
```

You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for "Hourglass", "Analog", "Digital", and "Stopwatch". When a timepiece is selected, an icon-based label is updated to display it. You can see how little code is required to use this powerful component.

```
// Demonstrate JComboBox.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/*
<applet code="JComboBoxDemo" width=300 height=200></applet>

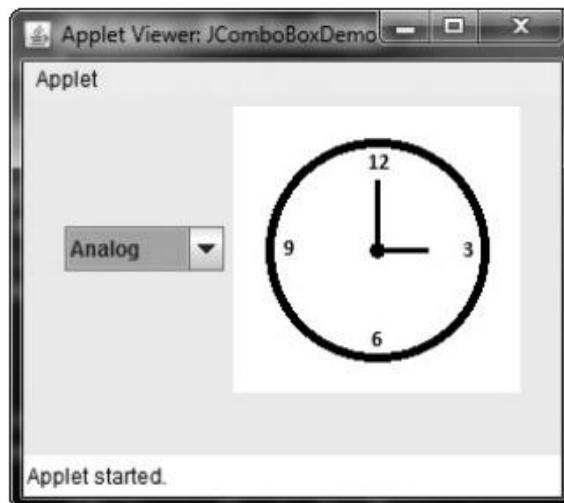
public class JComboBoxDemo extends JApplet {
    JLabel jlab;
    ImageIcon hourglass, analog, digital, stopwatch; JComboBox<String> jcb;
    String timepieces[] = { "Hourglass", "Analog", "Digital", "Stopwatch" };
```

```
public void init() {
    try {
        SwingUtilities.invokeAndWait( new Runnable() {
            public void run() {
                makeGUI();
            }
        });
    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}

private void makeGUI() {
    //Change to flow layout.
    setLayout(new FlowLayout());
    //Instantiate a combo box and add it to the content pane.
    jcb = new JComboBox<String>(timepieces);
    add(jcb);
    //Handle selections.
    jcb.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent ae) {
            String s = (String) jcb.getSelectedItem();
            jlab.setIcon(new ImageIcon(s + ".png"));
        }
    });
    // Create a label and add it to the content pane.
    jlab = new JLabel(new ImageIcon("hourglass.png"));
}
```

```
add(jlab);  
}  
}
```

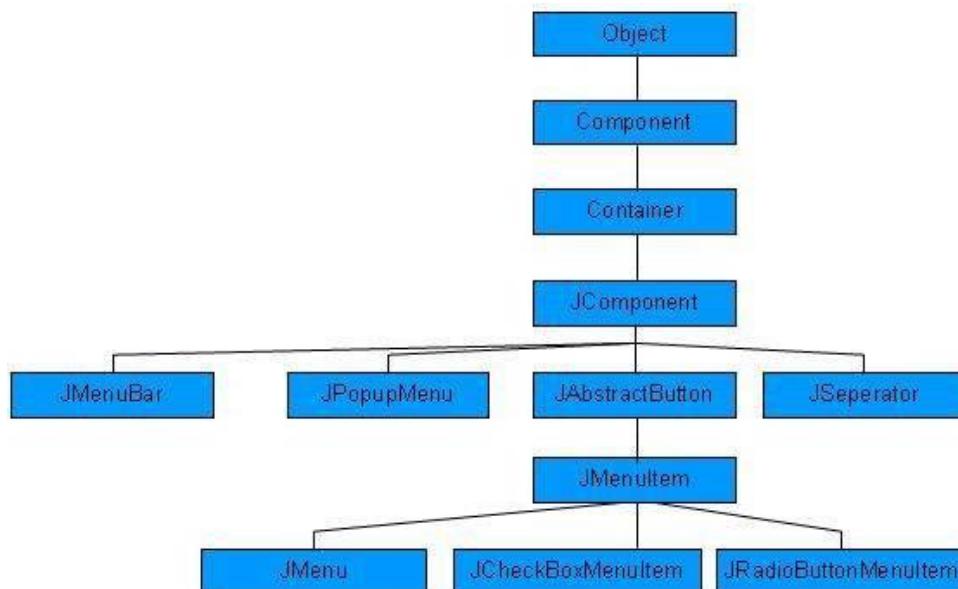
Output from the combo box example is shown here:



5.24 SWING - Menu Classes:

As we know that every top-level window has a menu bar associated with it. This menu bar consists of various menu choices available to the end user. Further, each choice contains a list of options, which is called drop-down menus. Menu and MenuItem controls are subclass of MenuComponent class.

Menu Hierarchy:



Menu Controls:

Sr.No.	Class & Description
1	<u>JMenuBar</u> The JMenuBar object is associated with the top-level window.
2	<u>JMenuItem</u> The items in the menu must belong to the JMenuItem or any of its subclass.
3	<u>JMenu</u> The JMenu object is a pull-down menu component which is displayed from the menu bar.
4	<u>JCheckboxMenuItem</u> JCheckboxMenuItem is the subclass of JMenuItem.
5	<u>JRadioButtonMenuItem</u> JRadioButtonMenuItem is the subclass of JMenuItem.
6	<u>JPopupMenu</u> JPopupMenu can be dynamically popped up at a specified position within a component.

5.25 Dialogs:

JDialog is a part Java swing package. The main purpose of the dialog is to add components to it. JDialog can be customized according to user need .

Constructor of the class are:

JDialog() : creates an empty dialog without any title or any specified owner

JDialog(Frame o) :creates an empty dialog with a specified frame as its owner

JDialog(Frame o, String s) : creates an empty dialog with a specified frame as its owner and a specified title

JDialog(Window o) : creates an empty dialog with a specified window as its owner

JDialog(Window o, String t) : creates an empty dialog with a specified window as its owner and specified title.

JDialog(Dialog o) :creates an empty dialog with a specified dialog as its owner

JDialog(Dialog o, String s) : creates an empty dialog with a specified dialog as its owner and specified title.

```
// java Program to create a simple JDialog

import java.awt.event.*;
import java.awt.*;
import javax.swing.*;

class solve extends JFrame implements ActionListener {

    // frame
    static JFrame f;

    // main class

    public static void main(String[] args)

    {

        // create a new frame
        f = new JFrame("frame");

        // create a object
        solve s = new solve();

        // create a panel
        JPanel p = new JPanel();

        JButton b = new JButton("click");

        // add actionlistener to button
        b.addActionListener(s);

        // add button to panel
        p.add(b);

        f.add(p);

        // set the size of frame
        f.setSize(400, 400);

        f.show();

    }

}
```

```
public void actionPerformed(ActionEvent e)
{
    String s = e.getActionCommand();
    if (s.equals("click"))
    {
        // create a dialog Box
        JDialog d = new JDialog(f, "dialog Box");
        // create a label
        JLabel l = new JLabel("this is a dialog box");
        d.add(l);
        //setszie of dialog
        d.setSize(100, 100);
        // set visibility of dialog
        d.setVisible(true);
    }
}
```


TUTORIAL QUESTIONS**UNIT – I**

1. Write a Java program to print an int, a double and a char on screen.
2. Suppose the values of variables 'a' and 'b' are 6 and 8 respectively, write two programs to swap the values of the two variables.
1 - first program by using a third variable.
2 - second program without using any third variable
(Swapping means interchanging the values of the two variables E.g.- If entered value of x is 5 and y is 10 then after swapping the value of x and y should become 10 and 5 respectively.)
3. Create a class 'Degree' having a method 'getDegree' that prints "I got a degree". It has two subclasses namely 'Undergraduate' and 'Postgraduate' each having a method with the same name that prints "I am an Undergraduate" and "I am a Postgraduate" respectively. Call the method by creating an object of each of the three classes.
4. Write a program to check if the letter 'e' is present in the word 'Umbrella'.
5. A school has following rules for grading system:
 - a. Below 25 - F
 - b. 25 to 45 - E
 - c. 45 to 50 - D
 - d. 50 to 60 - C
 - e. 60 to 80 - B
 - f. Above 80 - A
6. Ask user to enter marks and print the corresponding grade.7) Take integer inputs from user until he/she presses q (Ask to press q to quit after every integer input). Print average and product of all numbers.
7. Write a program to print the sum of two numbers entered by user by defining your own method.
8. Create a class with a method that prints "This is parent class" and its subclass with another method that prints "This is child class". Now, create an object for each of the class and call
1 - method of parent class by object of parent class
2 - method of child class by object of child class
3 - method of parent class by object of child class
9. Create a class 'Degree' having a method 'getDegree' that prints "I got a degree". It has two subclasses namely 'Undergraduate' and 'Postgraduate' each having a method with the same name that prints "I am an Undergraduate" and "I am a Postgraduate" respectively. Call the method by creating an object of each of the three classes.

10. Take an array of 10 elements. Split it into middle and store the elements in two different arrays.

E.g.-

INITIAL array :

58	24	13	15	63	9	8	81	1	78
----	----	----	----	----	---	---	----	---	----

After splitting :

58	24	13	15	63
9	8	81	1	78

UNIT – II

- 1) Write a program to enter the values of two variables 'a' and 'b' from keyboard and then check if both the conditions ' $a < 50$ ' and ' $a < b$ ' are true.
- 2) Write a program to give a simple example for abstract class.
- 3) Write a program to create interface named test. In this interface the member function is square. Implement this interface in arithmetic class. Create one new class called ToTestInt in this class use the object of arithmetic class.
- 4) Write a program to give example for multiple inheritance in Java.
- 5) Write a generic method to exchange the positions of two different elements in an array.
- 6) write a java program to illustrate the autoboxing.
- 7) Write a program to print the area and perimeter of a triangle having sides of 3, 4 and 5 units by creating a class named 'Triangle' with constructor having the three sides as its parameters.
- 8) Take name, roll number and field of interest from user and print in the format below : Hey, my name is xyz and my roll number is xyz. My field of interest are xyz.
- 9) Write a java program to implement nested interfaces.
- 10) write a java program to read the data from keyboard using DataInputStream class.

UNIT – III

- 1) Implement a java program for exception handling of Dividing by zero and Array Index out of bounds and Null pointer exception.
- 2) Write a program for example of multiple catch statements occurring in a program.
- 3) Write a program to create a class MyThread in this class a constructor, call the base class constructor, using super and starts the thread. The run method of the class starts after this. It can be observed that both main thread and created child thread are executed concurrently.
- 4) Write a program to get the reference to the current thread by calling currentThread() method.
- 5) Create a multithreaded program by creating a subclass of Thread and then creating, initializing, and starting two Thread objects from your class. The threads will execute concurrently and display Java is hot, aromatic, and invigorating to the console window.
- 6) Write a program to illustrate creation of threads using runnable class.(start method start each of the newly created thread. Inside the run method there is sleep() for suspend the thread for 500 milliseconds).
- 7) write a java program to implement the user defined thread.
- 8) write a java program to implement the rethrowing of an exception.
- 9) write a java program to implement thread synchronization.
- 10) write a java program to implement thread priorities.

UNIT – IV

- 1) write a java program to join 2 arraylists.
- 2) write a java program for How do you remove an element from a particular position of an ArrayList?
- 3) write a java program for How do you check whether the given element is present in an ArrayList or not?
- 4) write a java program for how to read all elements in vector by using iterator?
- 5) write a java program for how to find does vector contains all list elements or not?
- 6) write a java program for how to get all keys from HashMap?
- 7) write a java program for how to iterate through HashMap?
- 8) write a java program for how to create a TreeSet with a List?
- 9) write a java program for how to get highest and least element from a set
- 10) write a java program for how do you decrease the current capacity of an ArrayList to the current size?

UNIT – V

1) Implement a program with a GUI that looks like the one shown below. Put the main method in a class named MyDemo1.



2) write a java program to create a frame with three button.

3) Write a Java program to create three radio buttons. When any of them is selected, an appropriate message is displayed.

4) Write a program to display "All The Best" in 5 different colors on screen. (Using AWT/Swing)

5) Write a java program to create a frame containing three buttons (Yes, No, Close). When button yes or no is pressed, the message "Button Yes/No is pressed" gets displayed in label control. On pressing CLOSE button frame window gets closed.

6) Write a Java program to create a combo box which includes list of subjects. Display the selected subject in the text field using Swing.

7) write a java program that creates a simple applet and displays a message Hello World.

8) write a java program to perform any mouse and keyboard events.

9) Write a java program to create a grid layout for numbers 1 to 9.

10) write a java program to create a flowlayout.

ASSIGNMENT QUESTIONS

UNIT - I

1. Explain how the concept of polymorphism is implemented in Java?
2. Explain any five object oriented features supported by Java with examples.
3. Discuss about various conditional statements in Java with suitable examples.
4. Interpret type conversion and casting with an example.
5. Explain about for each loop with an example.
6. Explain about different loop structures in Java with an example.
7. How Java supports platform independence?
8. Explain method overriding with example program.
9. Write difference between constructor and method.
10. What is inheritance? Is multiple inheritance supported by Java? Summarize the use of “final” keyword within inheritance.

UNIT - II

1. Develop a Java code to implement the interface concept for finding the sum and average of given N numbers.
2. Describe output streams and input streams in Java.
3. What is the major difference between an interface and a class?
4. Explain Byte streams and character streams.
5. Discuss various Random access file operations in Java.
6. Discuss the role of classpath in packages.
7. Describe various member access rules and explain with an example.
8. Discuss in detail creating and importing package in Java.
9. What is an interface? How is it implemented?
10. What is a Package? What are the benefits of using packages? Write down the steps in creating a package and using it in a Java program with an example.

UNIT - III

1. Explain Daemon threads with an example.
2. Interpret various methods of thread class.
3. Explain with an example how Java performs thread synchronization.
4. What are checked and unchecked exceptions.
5. What classes of exceptions may be caught by a catch clause?
6. Distinguish between exception and error.
7. Compare and contrast between process and thread
8. Describe try, catch, and finally keywords with an example. Illustrate use of throws keyword with a program
9. Illustrate built in exceptions with suitable example
10. Differentiate multiprocessing and multithreading with a program.

UNIT - IV

1. Discuss Properties class with an example.
2. Discuss various collection algorithms.
3. Discuss various Legacy classes and interfaces
4. Explain Tree Set and Priority Queue with example.
5. Describe Date and Calendar classes.
6. Define Java collection Framework.
7. Write the function of StringTokenizer with example.
8. Write hash table with an example.
9. Explain various map interfaces.
10. Write Scanner class with an example

UNIT - V

1. What are the different buttons available in Swing
2. What are components and containers? Define Layout manager and its types.
3. Explain JButton and JToggleButton with an example program.
4. Discuss security issues in Applets.
5. What are the different buttons available in Swing
6. Describe JButton, JLabel, JTextField and JTextArea.
7. Explain briefly about adapter class?
8. Distinguish between applet and application?
9. Write short notes on Events, Event sources and Event Classes
10. Explain the MVC architecture

UNIT WISE IMPORTANT QUESTION QUESTIONS**UNIT-I****PART-A-UNIT-I (2 or 3 marks)**

1. DefineInheritance
2. What are the types of inheritance?
3. How is multiple inheritance achieved in java?
4. What is the use of superkeyword?
5. What are the OOPPrinciples?
6. What isEncapsulation?
7. What isPolymorphism?
8. What isInheritance?
9. What are the features of Java Language?
10. What is platformindependency?
11. What is ArchitectureNeutral?
12. How is a constant defined inJava?
13. What is the use of finalkeyword?
14. What are the different types of operators used inJava?
15. What is short-Circuitoperator?
16. Compare static constants and final constants.
17. What is the difference between a constructor and a method?
18. What is the use of thiskeyword?
19. What is methodoverloading?
20. Explain Javabuzzwords

PART-B-UNIT-I (5-10 marks)

1. Explain OOP Principles.
2. Explain the features of Java Language.
3. Compare and Contrast Java withC.
4. Explain Constructors withexamples.
5. Explain the methods available under String and String BufferClass.
6. Illustrate with examples: static and final.
7. Explain method overriding with exampleprogram.
8. Explain the methods under “object” class and “class”class.

9. What is inner class? What is the need for inner classes? What are the rules for inner class?
10. Explain various forms of Inheritance.
11. How Java supports platform independence?
12. Explain method overriding with example program.
13. Write difference between constructor and method
14. What is inheritance? Is multiple inheritance supported by Java? Summarize the use of “final” keyword within inheritance.
15. Illustrate the Use of “Super” keyword in method overriding with example.

UNIT-II

PART-A-UNIT-II (2 or 3marks)

1. Give details about java.io package.
2. Define an interface.
3. What is the need for an interface?
4. What is generic programming?
5. Explain Serialization?
6. What is auto boxing and unboxing
7. Explain byte stream.
8. Explain character stream
9. How do we extend interface?
10. Discuss about importing a package
11. Write how packages are accessed.
12. What is an interface? How is it implemented?
13. What do you mean by a package in Java?
14. Explain the differences between packages and interfaces.
15. What is input stream and output stream.
16. Define CLASSPATH?

PART-B-UNIT-II (5-10 marks)

1. Define interfaces. Discuss in detail about extending interfaces with an example
2. Write in detail about creating, importing and accessing packages
3. Explain about various packages in Java.
4. Write short notes on access specifiers and modifiers in Java.
5. What is a Package? What are the benefits of using packages? Write down the steps in creating a package and using it in a Java program with an example
6. Write extending interfaces with an example
7. Write working procedure of CLASSPATH
8. Write Member access rules with an example
9. Explain Byte streams and character streams.
10. Discuss various Random access file operations in Java.
11. Describe output streams and input streams in Java.
12. What is the major difference between an interface and a class?

13. Write extending interfaces with an example
14. Explain various methods involved in reading and writing files.
15. Explain the benefits of inheritance with an example.

UNIT-III

PART-A-UNIT-III (2 or 3 marks)

1. Define Exception.
2. State the use of try and catch blocks.
3. Define unchecked exceptions
4. Distinguish between exception and error
5. Describe the benefits of exception handling
6. List the different ways to create a thread.
7. Describe the various states of threads.
8. Interpret the different thread priorities
9. Distinguish between throw and throws.
10. Define wait() state of the thread
11. Differentiate throw and finally
12. Compare and contrast between process and thread
13. Write the classification of exceptions
14. Define inter-thread communication?
15. Write how thread class implements Runnable interface

PART-B-UNIT-III (5-10 marks)

1. Write about exception handling mechanisms
2. Write with an example how java performs threadsynchronization?
3. Describe the need for rethrowing an exception in java
4. Describe how do we set the path for class which is present in anotherclass?
5. Exemplify the different type ofexception.
6. Explain Daemon threads with anexample.
7. Interpret various methods of threadclass.
8. Explain with an example how java performs threadsynchronization.
9. What are checked and unchecked exceptions.
10. What classes of exceptions may be caught by a catchclause?
11. Distinguish between exception anderror.
12. Compare and contrast between process andthread
13. Describe try, catch , and finally keywords with an example. Illustrate use of throws keyword with aprogram
14. Illustrate built in exceptions with suitableexample
15. Differentiate multiprocessing and multithreading with aprogram.

UNIT-IV

PART-A-UNIT-IV (2 or 3 marks)

1. Define collections?
2. Define Java collection Framework.
3. Define ArrayList with syntax.
4. Define Vector with syntax.
5. Define stack with syntax.
6. Define random class.
7. Explain about scanner class
8. Define Calendar class.
9. Write the function of StringTokenizer.
10. Discuss about Iterator.
11. What are legacy classes
12. List collection classes
13. Describe various ways to access collection
14. List various collection algorithms
15. Explain arrays.

PART-B-UNIT-IV (5-10 marks)

1. Discuss Properties class with an example.
2. Discuss various collection algorithms.
3. Discuss various Legacy classes and interfaces
4. Explain TreeSet and PriorityQueue with example.
5. Describe Date and Calendar classes.
6. Write the function of StringTokenizer with example.
7. Write hash table with an example.
8. Explain various map interfaces.
9. Write Scanner class with an example
10. Explain about various legacy classes and interfaces
11. Write enumeration with an example
12. Explain Bit Set and Date classes
13. Explain in detail Random and Formatter classes
14. Explain the following interfaces:
15. a) Dictionary, b) Hashtable, c) Properties
16. Explain any five collection classes with example.

UNIT-V

PART-A-UNIT-V (2 or 3 marks)

1. Define Events, Event Sources and Eventclasses.
2. Describe the various keyboard and mouselisteners.
3. Give details about Delegation eventmodel
4. Distinguish between Containers andcomponents
5. List all the User InterfaceComponents.
6. What are adapterclasses.
7. Describe the applet life cyclemodel
8. Describe the MVArchitecture
9. Give details about Swingcomponents.
10. Describe applet securityissues?
11. Write Event Listeners
12. Write hierarchy forswing?
13. Define Layoutmanagement?
14. List Layout manager types – border and gridflow?
15. Define JFrame,JApplet.

PART-B-UUNIT-V (5-10 marks)

1. What are the different buttons available inSwing
2. What are components and containers? Define Layout manager and its types.
3. Explain JButton and JToggle Button with an exampleprogram
4. Discuss security issues inApplets
5. Write in detail about hierarchy forswing?
6. What are the different buttons available inSwing
7. Describe JButton, JLabel, JTextField andJTextArea.
8. Explain briefly about adapterclass?
9. Distinguish between applet andapplication?
10. Write short notes on Events, Event sources and EventClasses
11. Explain the MVArchitecture
12. Write a program for passing parameters toapplet?
13. What is an applet? How do applets differ from an applicationprograms?
14. Explain the steps involved to execute and run an applet. Write a method in Java applet to display a circle.
15. Explain Inner classes and Anonymousclasses

OBJECTIVE QUESTIONS

UNIT - I

1) Which is the term used for information hiding in OOPS?

- a) abstraction b) encapsulation c) polymorphism d) None

Answer : B

2) Which of the following features is supported by Object Oriented Programming?

- a) Abstraction b) Encapsulation c) Inheritance d) All the above

Answer : D

3) The default package imported by all java programs is

- a) java.lang b) java.swing c) java.awt d) java.applet

Answer : A

4) Which is used to get the value of the instance variables?

- (a).(Dot) (b)>> (c)<< (d)->

Answer: A

5) A constructor is used to _____

- a. destroy memory b. initialize newly created object c. import packages
d. create a JVM for applets

Answer: B

6) Which operator is used to create and concatenate string?

- (a)++ (b)&& (c)& (d) +

Answer:D

7) Which of these is returned by operators & , ?

- a) Integer b) Boolean c) Character d) Float Answer: c

8) Literal can be of which of these datatypes?

- a) integer b) float c) boolean d) all of the mentioned

Answer:d

9) Which of these cannot be used for a variable name in Java?

- a) identifier b) keyword c) both a & b d) None of the mentioned

Answer:b

10) Which of the following statements is true considering OOP (Object Oriented Programming)?

- a) State defines the attribute of an object b) Behaviour defines the attribute of an object
c) State defines the functions to modify the behaviour of an object d) none of the above

Answer : C

11) What is the order of variables inEnum?

- a) Ascending order b) Descending order c) Random order d) depends on the order() method

Answer: a

12) Which of these operators is used to allocate memory to array variable in Java?

- a)malloc b)alloc c)new d) new malloc

Answer:c

13) Which of the following is used with switchstatement?

- a)Continue
 - b)Exit
 - c)break
 - d) do

Answer:c

14) Which component is used to compile, debug and execute java program?

- a) JVM
 - b) JDK
 - c) JIT
 - d) JRE

Answer:b

15) What is not type of inheritance?

- a) Single inheritance b) Double inheritance c) Hierarchical inheritance
 - d) Multiple inheritance

Answer: b

16) Which of the following is used for implementing inheritance through class?

- a) inherited
 - b) using
 - c) extends
 - d) implements

Answer:c

17) Which of these is correct way of calling a constructor having no parameters, of superclass A by subclassB?

- a)super(void); b)superclass.(); c)super.A(); d) super(); Answer:d

18) Which of these packages contains abstract keyword?

- a)java.lang b)java.util c)java.io d) java.system

Answer:a

19) What is the process of defining a method in terms of itself, that is a method that calls itself?

- a) Polymorphism
 - b) Abstraction
 - c) Encapsulation
 - d) Recursion

Answer:d

UNIT – 2

1. Which of these class contains the methods print() &println()?

- a) System
- b) System.out
- c) BUfferedOutputStream
- d) PrintStream

Answer:d

2. Which of these is a process of writing the state of an object to a bytestream?

- a) Serialization
- b) Externalization
- c) File Filtering
- d) All of the mentioned

Answer:a

3. Which of these keywords is used to define packages inJava?

- a) pkg
- b) Pkg
- c) package
- d) Package

Answer:c.

4. Which of these is a mechanism for naming and visibility control of a class and itscontent?

- a) Object
- b) Packages
- c) Interfaces
- d) None of the Mentioned.

Answer:b

5. Which of these can be used to fully abstract a class from itsimplementation?

- a) Objects
- b) Packages
- c) Interfaces
- d) None of the Mentioned

Answer:c

6. Which of the following package stores all the standard javaclasses?

- a.lang
- b)java .util
- c)java.packages

Answer:b

7. Which of these is a type of stream in Java?

- a) Integerstream
- b) Shortstream
- c) Bytestream
- d) Long stream

Answer:c

8. Which of these class is used to read from bytearray?

- a) InputStream.
- b) BufferedInputStream.
- c) ArrayInputStream.
- d) ByteArrayInputStream.

Answer:d

9. Which of these is an correct way of defining genericclass?

- a) class name(T1, T2, ..., Tn) { /* ... */}
- b) class name { /* ... */}
- c) class name[T1, T2, ..., Tn] { /* ... */ }
- d) class name{T1, T2, ..., Tn} { /* ... */ }

Answer:b

10. Which of these stream contains the classes which can work on characterstream?

- a) InputStream
- b) OutputStream
- c) CharacterStream
- d) All of the mentioned

Answer:c

11) Which of these method of InputStream is used to read integer representation of next available byte input?

- a) read()
- b) scanf()
- c) get()
- d) getInteger()

Answer: a

12) Which of these class can be used to implement input stream that uses a character arrays thesource?

- a) BufferedReader
- b) FileReader
- c) CharArrayReader
- d) FileArrayReader

Answer:c

13) Which of these stream contains the classes which can work on characterstream?

- a) InputStream
- b) OutputStream
- c) CharacterStream
- d) All of the mentioned

Answer:c

14) Which of the following modifier means a particular variable cannot be accessed within thepackage?

- a) private
- b) public
- c) protected
- d) default

Answer:a

15) What is true of final class?

- a) Final class cause compilation failure
- b) Final class cannot be instantiated
- c) Final class cause runtime failure
- d) Final class cannot be inherited

Answer:d

16) Which of these keywords is used by a class to use an interface defined previously?

- a) import
- b) Import
- c) implements
- d) Implements

Answer:c

17) Which of these can be used to fully abstract a class from its implementation?

- a) Objects
- b) Packages
- c) Interfaces
- d) None of the Mentioned

Answer:c

18) Which of the following is correct way of importing an entire package ‘pkg’?

- a) importpkg.
- b) Importpkg.
- c) Import pkg.*

Answer:c

19) Why are generics used?

- a) Generics make code more fast
- b) Generics make code more optimised and readable
- c) Generics add stability to your code by making more of your bugs detectable at compile time
- d) Generics add stability to your code by making more of your bugs detectable at runtime

Answer:c

20) Which of these process occur automatically by java run time system?

- a) Serialization
- b) Memory allocation
- c) Deserialization
- d) All of the mentioned

Answer:d

UNIT – 3

1. Which of the following statements about the try{ } block in Java is false?
 - a) Some of the statements in a try{ } block will never throw an exception.
 - b) The try{ } block must appear before the catch{ } blocks.
 - c) The try{ } block cannot contain loops or branches.
 - d) The statements in a try{ } block may throw several types of exception.

Answer:c

_____ is at the top of the exception class hierarchy.

- a) try.
- b) throwable.
- c) exception class.
- d) catch.

Answer:b

2. Which of these keywords is not a part of exception handling?.

- a) finally.
- b) catch.
- c) thrown
- d) try

Answer:c

- 5) Which of these exceptions will be thrown if we use null reference for an arithmetic operation?

- a) ArithmeticException
- b) NullPointerException
- c) IllegalAccessException
- d) IllegalOperationException

Answer:b

- 6) Which of these exceptions handles the situations when illegal argument is used to invoke a method?

- a) IllegalException
- b) ArgumentException
- c) IllegalArgumentException
- d) IllegalMethodArgumentException

Answer:c

7) Which of these packages contain all the Java's built in exceptions?

- a) java.io
- b) java.util
- c) java.lang
- d) java.net

Answer:c

8) Which of these keywords is not a part of exception handling?

- a) try
- b) finally
- c) thrown
- d) catch

Answer:c

9) Which of these handles the exception when no catch is used?

- a) Defaulthandler
- b) finally
- c) throwhandler
- d) Java run time system

Answer:a

10) Which of these classes is used to define exceptions?

- a) Exception
- b) Trowable
- c) Abstract
- d) System

Answer:a

11) Which of these interface is implemented by Thread class?

- a) Runnable
- b) Connections
- c) Set
- d) MapConnections

Answer:a

12) Which function of pre defined class Thread is used to check whether current thread being checked is still running?

- a) isAlive()
- b) Join()

- c) isRunning()
- d) Alive()

Answer:a

13) Thread priority in Java is?

- a) Integer
- b) Float
- c) double
- d) long

Answer:a

14) Which of these statements is incorrect?

- a) By multithreading CPU's idle time is minimized, and we can take maximum use of it
- b) By multitasking CPU's idle time is minimized, and we can take maximum use of it
- c) Two threads in Java can have same priority
- d) A thread can exist only in two states, running and blocked

Answer:d

15) What does not prevent JVM from terminating?

- a) Process
- b) Daemon Thread
- c) User Thread
- d) JVM Thread

Answer:b

16) What should not be done to avoid deadlock?

- a) Avoid using multiple threads
- b) Avoid holding several locks at once
- c) Execute foreign code while holding a lock
- d) Use interruptible locks

Answer:c

17) Which of the following stops execution of a thread?

- a) Calling SetPriority() method on a Thread object
- b) Calling notify() method on an object
- c) Calling wait() method on an object
- d) Calling read() method on an InputStream object

Answer:b

18) Which of these keywords are used to implementsynchronization?

- a) sunchronize
- b) syn
- c) synch
- d)synchronized

Answer:d

19) Which of these method wakes up the first thread that calledwait()?

- a) wake()
- b) notify()
- c) start()
- d) notifyAll()

Answer:b

20) Which of these method waits for the thread to terminate?

- a) sleep()
- b) isAlive()
- c) join()
stop()

Answer: c

UNIT -4:

1)Which of these class can generate an array which can increase and decrease in size automatically?

- a) ArrayList()
- b) DynamicList()
- c) LinkedList()
- d) MallocList()

Answer:a

2) What is the difference between length() and size() ofArrayList?

- a) length() and size() return the samevalue
- b) length() is not defined inArrayList
- c) size() is not defined in ArrayList
- d) length() returns the capacity of ArrayList and size() returns the actual number of elements
- e) stored in thelist

Answer: d

3) Which class provides thread safe implementation ofList?

- a) ArrayList
- b) CopyOnWriteArrayList
- c) HashList
- d) List

e) Answer:b

4) Which of these classes is not part of Java's collectionframework?

- a) Maps
- b) Array
- c) Stack
- d) Queue

Answer:a

5) Which of these packages contain all the collectionclasses?

- a) java.lang
- b) java.util
- c) java.net
- d) java.awt

Answer:b

6) What is the default clone ofHashSet?

- a) Deepclone
- b) Shallowclone
- c) Plain clone
- d) Hollow clone

Answer:b

7) What is the unique feature ofLinkedHashSet?

- a) It is not a validclass
- b) It maintains the insertion order and guaranteesuniqueness
- c) It provides a way to store key values withuniqueness
- d) The elements in the collection are linked to each other

Answer:b

8) Which of these interface declares core method that all collections willhave?

- a) Set
- b) b)EventListner
- c) Comparator

d) Collection

Answer:d

9) Which of these interface is not a part of Java's collectionframework?

- a) List
- b) Set
- c) SortedMap
- d) SortedList

Answer:d

10) Which of these return type of hasNext() method of an iterator?

- a) Integer
- b) Double
- c) Boolean
- d) Collections Object

Answer:c

11) Which of these methods can be used to move to next element in a collection?

- a) next()
- b) move()
- c) shuffle()
- d) hasNext()

Answer:a

12) Which of these exceptions is thrown by remove() method?

- a) IOException
- b) SystemException
- c) ObjectNotFoundException
- d) IllegalStateException

Answer:d

13) Which of these class object uses key to store value?

- a) Dictionary
- b) Map
- c) Hashtable
- d) All of the mentioned

Answer:d

14) Which of these is a class which uses String as a key to store the value in object?

- a) Array

b) ArrayList

c) Dictionary

15) Which of these method is used to insert value and itskey?

a) put()

b) set()

c) insertElement()

d) addElement()

Answer:a

16) Which of these class object has architecture similar to that of array?

a) Bitset

b) Map

c) Hashtable

d) All of the mentioned

Answer:a

17) Which of these method is used to calculate number of bits required to hold the BitSet object?

a) size()

b) length()

c) indexes()

d) numberofBits()

Answer:b

18) Which of these methods is used to retrieve elements in BitSet object at specificlocation?

a) get()

b) Elementat()

c) ElementAt()

d) getProperty()

Answer:a

19) Which of these methods sets every element of a List to a specifiedobject?

a) set()

b) fill()

c) Complete()

d) add()

Answer:b

20) Which of these methods can convert an object into a List?

- a) SetList()
- b) ConvertList()
- c) singletonLis()
- d) ConvertList()
- e) CopyList()
- f) Answer:c

UNIT – V:

1) Which of these events is generated when the a window is closed?

- a) TextEvent
- b) MouseEvent
- c) FocusEvent
- d) WindowEvent

Answer:d

2) Which of these methods can be used to obtain the coordinates of a mouse?

- a) getPoint()
- b) getCoordinates()
- c) getMouseXY()
- d) getMouseCoordinates()

Answer:a

3) MouseEvent is subclass of which of these classes?

- a) ComponentEvent
- b) ContainerEvent
- c) ItemEvent
- d) InputEvent

Answer:d

4) Which of these is superclass of WindowEvent class?

- a) WindowEvent
- b) ComponentEvent
- c) ItemEvent
- d) InputEvent

Answer:b

5) Which of these packages contains all the event handling interfaces?

- a) java.lang
- b) java.awt
- c) java.awt.event
- d) java.event

e) Answer:c

6) Which of these interfaces define a method actionPerformed()?

- a) ComponentListener
- b) ContainerListener
- c) ActionListener
- d) InputListener

Answer:c

7) Which of these packages contains all the classes and methods required for even handling in Java?

- a) java.applet
- b) java.awt
- c) java.event
- d) java.awt.event

Answer:d

8) Event class is defined in which of these libraries?

- a) java.io
- b) java.lang
- c) java.net
- d) java.util

Answer:d

9) Which of these events will be notified if scroll bar is manipulated?

- a) ActionEvent
- b) ComponentEvent
- c) AdjustmentEvent
- d) WindowEvent

Answer:c

10) Which of these methods can be used to obtain the command name for invoking ActionEvent object?

- a) getCommand()
- b) getActionCommand()
- c) getActionEvent()
- d) getActionEventCommand()

Answer:b

8) Which of these methods can be used to know which key ispressed?

- a) getKey()
- b) getModifier()
- c) getActionKey()
- d) getActionEvent()

Answer:b

9) Which of these events will be notified if scroll bar ismanipulated?

- a) ActionEvent
- b) ComponentEvent
- c) AdjustmentEvent
- d) WindowEvent

Answer:c

10) Which of these events is generated when the component is added orremoved?

- a) ComponentEvent
- b) ContainerEvent
- c) FocusEvent
- d) InputEvent

Answer:b

11) Which of these are integer constants of ComponentEventclass?

- a) COMPONENT_HIDDEN
- b) COMPONENT_MOVED
- c) COMPONENT_RESIZE
- d) All of the mentioned

Answer:d

12) Which of these events will be notified if scroll bar is manipulated?

- a) ActionEvent
- b) ComponentEvent
- c) AdjustmentEvent
- d) WindowEvent

Answer:c

13) Which of these is superclass of ContainerEvent class?

- a) WindowEvent
- b) ComponentEvent
- c) ItemEvent
- d) InputEvent

Answer:b

14) What is an event in delegation event model used by Java programming language?

- a) An event is an object that describes a state change in a source
- b) An event is an object that describes a state change in processing
- c) An event is an object that describes any change by the user and system
- d) An event is a class used for defining object, to create events

Answer:a

15) Which of these methods can be used to determine the type of event?

- a) getID()
- b) getSource()
- c) getEvent()
- d) getEventObject()

Answer:a

16) Which of these interfaces define four methods?

- a) ComponentListener
- b) ContainerListener
- c) ActionListener
- d) InputListener

Answer:a

INTERNAL QUESTION PAPERS**MALLA REDDY ENGINEERING COLLEGE FOR WOMEN****(Autonomous Institution – UGC, Govt of India)***(Permanently Affiliated to JNTUH, Hyderabad, Approved by AICTE - ISO 9001:2015 Certified)***Accredited by NBA & NAAC with 'A' Grade****II B.TECH II SEMESTER (2018-BATCH) I MID EXAMINATION FEBRUARY-2020****Sub: Object Oriented Programming****Duration : 1 Hour 40Min****Year/Sec: II/D,IT****Max. Marks : 20**

Answer any Four Questions (4*5=20M)

- 1) a) Explain briefly about the OOPS Concepts.
b) Define a constructor? Explain default constructor with an example java program
- 2) a) Briefly explore Access modifiers of a class.
b) Identify the uses of super keyword with an illustrative program
- 3) a) Name and describe various Forms of Inheritance.
b) Write a java program to illustrate use of Abstract class.
- 4) a) Write a java program to explore use of inner classes in java.
b) Explain about how multiple Inheritance achieved in java with example program
- 5) a) Write about Method overloading and method overriding with an example.
b) Define a Package? Create a user define package and how to import the package with an example program
- 6) a) Explain about usage of try and catch blocks in Exception handling.
b) Discuss briefly about final, finally, finalized keywords.



MALLA REDDY ENGINEERING COLLEGE FOR WOMEN
(Autonomous Institution – UGC, Govt of India)
(Permanently Affiliated to JNTUH, Hyderabad, Approved by AICTE - ISO 9001:2015 Certified)
Accredited by NBA & NAAC with 'A' Grade

II B.TECH II SEMESTER (2018-BATCH) I MID EXAMINATION FEBRUARY-2020

Sub: Object Oriented Programming

Duration : 1 Hour 40Min

Year/Sec: II/D,IT

Max. Marks : 20

Answer any Four Questions (4*5=20M)

- 1) a) List various parameter passing methods in java.
b) Define a constructor? Explain parameterized constructor with an example program.
- 2) a) Summarize the functionality of various methods in String class of java.
b) Identify the uses of final keyword with an illustrative program.
- 3) a) Name and describe various methods defined in an Object class.
b) With an example write about an abstract class with supporting example.
- 4) a) Briefly explain the usage of inner class with an example program.
b) Explain about Interface with an example program.
- 5) a) Write about polymorphism in detail.
b) Discuss the various levels of access protection available for packages and their implications.
- 6) a) Discuss about various keywords used in handling the exceptions.
b) Differentiate between final and finalize.



MALLA REDDY ENGINEERING COLLEGE FOR WOMEN

(Autonomous Institution – UGC, Govt of India)

(Permanently Affiliated to JNTUH, Hyderabad, Approved by AICTE - ISO 9001:2015 Certified)
Accredited by NBA & NAAC with 'A' Grade

II B.TECH II SEMESTER (2018-BATCH) II MID EXAMINATION APRIL 2020

Sub: Object Oriented Programming

Duration : 1 Hour 40Min

Year/Sec: II/D. IT

Max. Marks : 20

Answer any Four Questions (4*5=20M)

1. Does java support thread priorities? Justify your answer with suitable program.
 2. Describe producer consumer pattern using inter thread communication.
 3. Discuss the differences between Hash List and Hash Map, Set and List.
 4. Write a program to create and read a text file.
 5. How mouse events can be handle in Java. With an example.
 6. Explain layout types in AWT with the help of an example.
 7. Elaborate about Life Cycle of Applet.
 8. Discuss about model view controller.

PREVIOUS QUESTION PAPERS

R16**Code No: 133BM****JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD****B.Tech II Year I Semester Examinations, May/June - 2019****OBJECT ORIENTED PROGRAMMING THROUGH JAVA****(Common to CSE, IT)****Time: 3 Hours****Max. Marks: 75****Note:** This question paper contains two parts A and B.

Part A is compulsory which carries 25 marks. Answer all questions in Part A.

Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b as sub questions.

PART- A**(25 Marks)**

- | | | |
|------|---|-----|
| 1.a) | Differentiate between class and object. | [2] |
| b) | What is meant by ad-hoc polymorphism? | [3] |
| c) | How to define a package in Java? | [2] |
| d) | Contrast between abstract class and interface. | [3] |
| e) | Define exception. | [2] |
| f) | Differentiate between a thread and a process. | [3] |
| g) | Which methods of deque enable it to be used as a stack? | [2] |
| h) | Make a comparison of List, array and ArrayList. | [3] |
| i) | Give the AWT hierarchy. | [2] |
| j) | What are the various classes used in creating a swing menu? | [3] |

PART-B**(50 Marks)**

- | | | |
|-----------|--|-------|
| 2.a) | What are the responsibilities of an agent? | |
| b) | What is the purpose of constructor in Java programming? | [5+5] |
| OR | | |
| 3. | Define inheritance. What are the benefits of inheritance? What costs are associated with inheritance? How to prevent a class from inheritance? | [10] |
| 4. | Write a program to demonstrate hierarchical and multiple inheritance using interfaces. | [10] |

OR

- | | | |
|-----------|---|-------|
| 5.a) | Demonstrate ordinal() method of enum. | |
| b) | What is type wrapper? What is the role of auto boxing? | [5+5] |
| 6. | Write a program to create three threads in your program and context switch among the threads using sleep functions. | [10] |
| OR | | |
| 7.a) | Write a program with nested try statements for handling exception. | |
| b) | How to create a user defined exception? | [5+5] |

8. Write a program to read a file content and extract words using StringTokenizer class.
Display the file if it contains the user query term/search key. [10]

OR

- 9.a) Contrast sorted map and navigable map interfaces.
b) What is the purpose of BitSet class? What is the functionality of the following functions of BitSet class: cardinality(), flip() and intersects() [5+5]

- 10.a) Illustrate the use of Grid Bag layout.
b) What are the subclasses of JButton class of swing package? [5+5]

OR

- 11.a) Create a simple applet to display a smiley picture using Graphics class methods.
b) Write a short note on delegation event model. [5+5]

---0000---

R13

Code No: 114CX**JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD****B.Tech II Year II Semester Examinations, December - 2018****JAVA PROGRAMMING****(Common to CSE, IT)****Time: 3 Hours****Max. Marks: 75****Note:** This question paper contains two parts A and B.

Part A is compulsory which carries 25 marks. Answer all questions in Part A.

Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART- A**(25 Marks)**

- 1.a) What is the significance of Java's byte code? [2]
- b) List the applications of object oriented programming. [3]
- c) Differentiate class, abstract class and interface. [2]
- d) How to create and use a package in Java program? [3]
- e) How does Java support inter thread communication? [2]
- f) List any four unchecked exception. [3]
- g) What is the use of Iterator class? [2]
- h) Compare byte streams with character streams. [3]
- i) Give the subclasses of JButton class. [2]
- j) Differentiate between grid layout and border layout managers. [3]

PART- B**(50 Marks)**

- 2.a) What are the drawbacks of procedural languages? Explain the need of object oriented programming with suitable program.
- b) Discuss the lexical issues of Java. [5+5]

OR

- 3.a) What are the primitive data types in Java? Write about type conversions.
- b) What is a constructor? What is its requirement in programming? Explain with program. [5+5]

- 4.a) With suitable code segments illustrate various uses of 'final' keyword.
- b) Discuss about anonymous inner classes. [5+5]

OR

- 5. What are the benefits of inheritance? Explain the various forms of inheritance with suitable code segments. [10]

- 6.a) With a program illustrate user defined exception handling
- b) How to handle multiple catch blocks for a nested try block? Explain with an example. [5+5]

OR

- 7.a) Describe how to create a thread with an example.
- b) Write a program to explain thread priorities usage. [5+5]

8. What support is provided by File class for file management? Illustrate with suitable scenarios. [10]

OR

- 9.a) Describe different types of JDBC drivers.
b) Explain the random access file operations with a suitable program. [5+5]

- 10.a) What is the role of event listeners in event handling? List the Java event listeners
b) Write an applet to display the mouse cursor position in that applet window.[5+5]

OR

- 11.a) Discuss various AWT containers with examples.
b) Explain about the adapter class with an example. [5+5]

--ooOoo--

20 20 20 20 20
R13

Code No: 114CX
JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

B.Tech II Year II Semester Examinations, May - 2017

JAVA PROGRAMMING

(Common to CSE, IT)

Time: 3 Hours

26

26

26

26

Max. Marks: 75

26

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A.
Part B consists of 5 Units. Answer any one full question from each unit.
Each question carries 10 marks and may have a, b, c as sub questions.

26

26

PART - A

26

26 (25 Marks)

26

- 1.a) What are the features of Java language? [2]
- b) Explain the types of operators used in Java. [3]
- c) What is static inner class? [2]
- d) Explain implicit and explicit import statement. [3]
- e) Explain the differences between process and thread. [2]
- f) Explain how a multiple catch statement works. [3]
- g) Explain the use of string tokenizer with an example. [2]
- h) Explain any three methods defined by iterator. [3]
- i) Explain the use of layout managers. [2]
- j) Explain the life cycle of an applet. [3]

PART - B

(50 Marks)

- 2.a) Describe the different types of data types used in Java. [5+5]
- b) Write a program to convert the temperature in Fahrenheit to centigrade.

OR

- 3.a) Compare and contrast between the overloading and overriding methods with an example. [5+5]
- b) Write a java program to display the following output.

1 2
1 2 3
1 2 3 4
1 2 3 4 5

- 4.a) Explain the importance of anonymous inner class with an example. [5+5]
- b) Write a program to find the sum of the given number.

OR

- 5.a) What is java package? What is CLASSPATH? Explain how to create and access a java package with an example. [5+5]
- b) Create an interface with at least one method and implement that interface by within a method which returns a reference to your interface. [5+5]

6.a) Write a program for user defined exception that checks the internal and external marks if the internal marks are greater than 40 it raise the exception "internal marks are exceed", if the external marks are greater than 60 exception is raised and display the message the "external marks are exceed".

b) Explain the synchronization methods with an example.

[5+5]

OR

7.a) Write a program to implement a producer and consumer problem by using multithreading.

b) Explain the java built in exceptions.

[5+5]

8.a) Write a program to compute an average of the values in a file.

b) Explain the methods defined by Math.

[5+5]

OR

9.a) Explain the different types of drivers used in JDBC.

b) Write a program to store the names of bank depositors and their current balances by using hash table.

[5+5]

10.a) Write a java program to design a scientific calculator using AWT.

b) What are the different types of Event listeners supported by java?

[5+5]

OR

11.a) Write a program using an applet which will print "key pressed" on the status Window when you press the key, "key released" on status window when you release the key and when you type the characters it should print "hello" at co-ordinates (50,50) on Applet.

b) Explain the various components in Swing.

[5+5]

--000oo--