LATEST 2020-21 EDITION

JAVA PROGRAMMING

FOR BSC III YEAR V SEM
KAKATIYA UNIVERSITY

SRINIVAS DANDU

SYLLABUS

Java Programming

UNIT-I

Introduction Java Essentials, JVM, Java Features, Creation and Execution of Programs, Data Types, TypeConversion, Casting, Conditional Statements, Loops, Branching Mechanism,Classes, Objects, ClassDeclaration, Creating Objects, Method Declaration and Invocation,Method Overloading,

UNIT -II

Constructors-ParameterizedConstructors, Constructor Overloading, Cleaning up unused Objects.Class Variables & Method static Keyword, this Keyword, One Dimensional Arrays, Two-Dimensional Arrays, Command Line Arguments, Inner Class.

Inheritance: Introduction, Types o f Inheritance, extends Keyword, Examples, Method Overriding, super,final Keyword, Abstract classes, Interfaces, Abstract Classes Verses Interfaces.

Packages:Creating and Using Packages, Access Protection, Wrapper Classes, String Class,String BufferClass.

UNIT-III

Exception: Introduction, Types, Exception Handling Techniques, User Defined Exception.

Multithreading: Introduction, Main Threadand Creationof New Threads by Inheriting the Thread Class orImplementing the Runnable Interface, Thr ead Lifecycle, Thread Priority and Synchronization

Input/Output: Introduction, java.io Package, File Class, FileInputStream Class, FileOutputStream Class, Scanner Class, BufferedInputStream Class, BufferedOutputStream Class, RndomAccessFile Class.

UNIT-IV

Applets:Introduction, Example, Life Cycle, Applet Class, Common Methods Used in Displaying the Output.

Event Handling: Introd uction, Types of Events, Example.

AWT: Introduction, Components, Containers, Button, Label, Checkbox, Radio Buttons, Container Class, Layouts. Swing: Introduction, Differences between Swing and AWT, Jframe, Japplet, Jpanel, Components in Swings, Layout Managers, Jtable, Dialog Box.

Database Handling Using JDBC: Introduction, Types of JDBC Drivers, Load the Driver, Establish Connection, Create Statement, Execute Query, Iterate Resultset, Scrolla ble Resultset, developing a JDBCApplication.

UNIT-I

INTRODUCTION:

Java is a programming language and a platform. Java is a high level, robust, object-oriented and secure programming language.
Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the Oak name to Java.
1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
2) Initially designed for small, embedded systems in electronic appliances like set-top boxes.
3) Firstly, it was called "Greentalk" by James Gosling, and the file extension was .gt.
4) After that, it was called Oak and was developed as a part of the Green project.
Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called enterprise application. It has advantages of the high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

ESSENTIALS OF JAVA:

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.
JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is

platform independent. There are three notions of the JVM: specification, implementation, and instance.

The JVM performs the following main tasks:

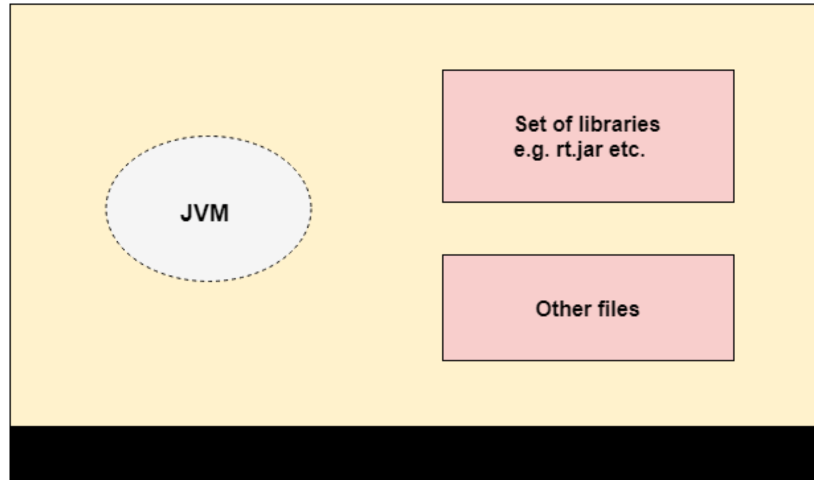
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun MicroSystems.

JDK



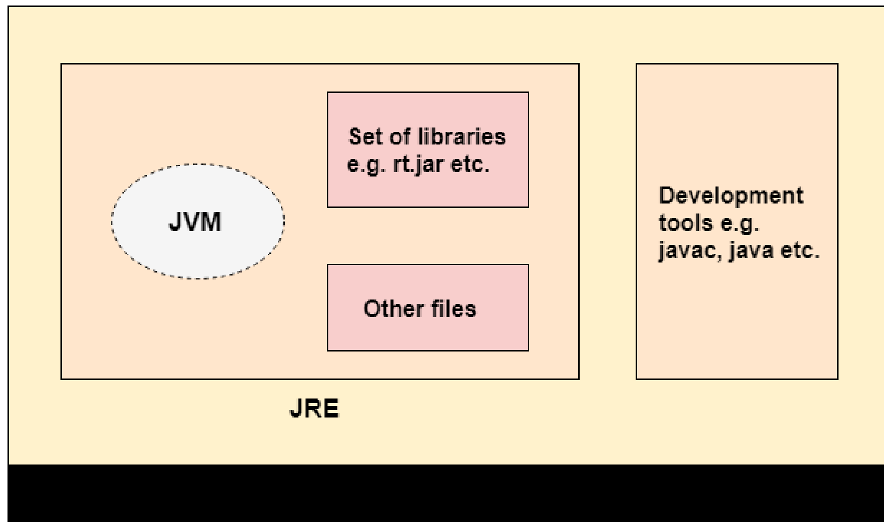
JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



CREATION AND EXECUTION OF PROGRAMS:

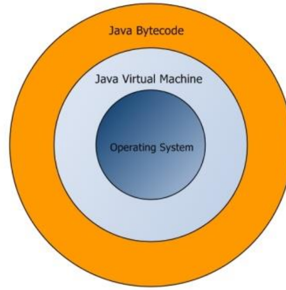
Step 1 : Creation of a Java program :-
By creating of a Java program we mean of writing of a Java program on any editor or IDE. This includes modifying of the program on editor, even after program has been written once. Creation of a Java program is not limit to only write program on editor and then leave it. Generally, creating of a Java program means writing a program on a editor or IDE, making all the corrections needed and then saving the program on secondary storage device as hard drive. After creating a Java program you should provide .java extension to file. It signifies that the particular file is a Java source code.

Step 2 : Compiling a Java Program

Our next step after creation of program is the compilation of Java program. Generally Java program which we have created in step 1 with a .java extension, it is been compiled by the compiler. Suppose if we take example of our program say WelcomeJavaPrograms.java, when we want to compile this we use command such as javac.

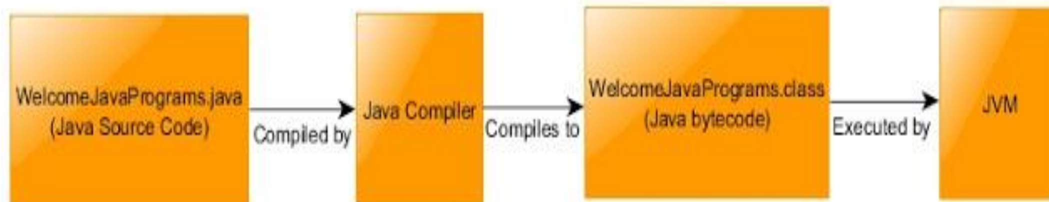
Java program with .java extension as **javac WelcomeJavaPrograms.java.**

Step 3 : Program loading into memory by JVM:-
JVM requires memory to load the .class file before its execution. The process of placing a program in memory in order to run is called as Loading. There is a class loader present in the JVM whose main functionality is to load the .class file into the primary memory for the execution. All the .class files required by our program for the execution is been loaded by the class loader into the memory just before the execution.



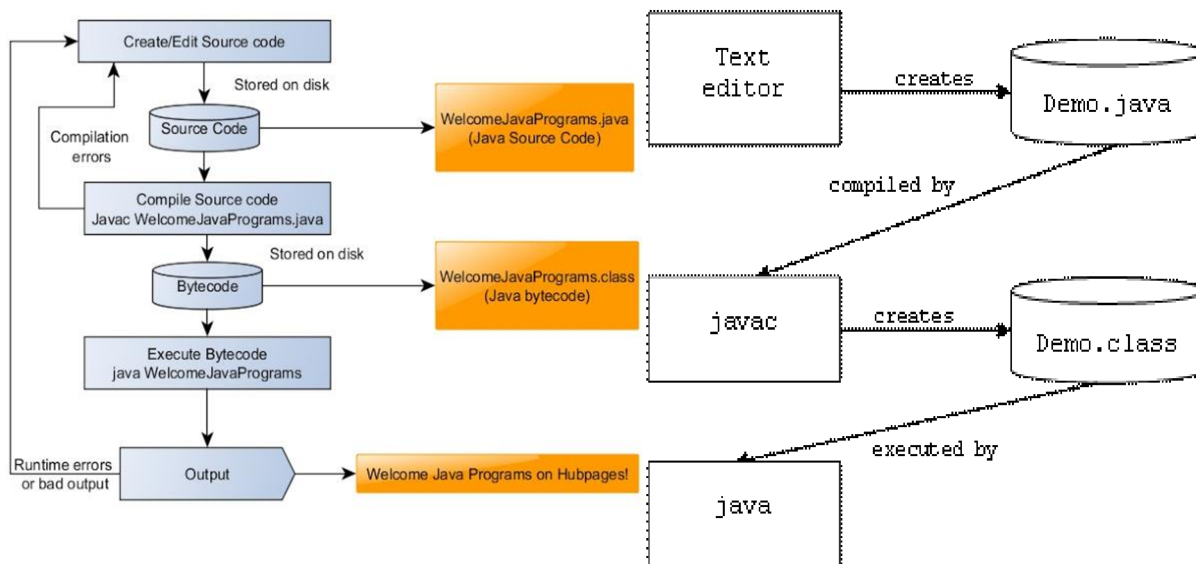
Step 4: Bytecode Verification by JVM :-

In order to maintain security of the program JVM has bytecode verifier. After the classes are loaded in to memory , bytecode verifier comes into picture and verifies bytecode of the loaded class in order to maintain security. It check whether bytecodes are valid. Thus it prevent our computer from malicious viruses and worms.



Step 5 : Execution of Java program : -

Whatever actions we have written in our Java program, JVM executes them by interpreting bytecode. If we talk about old JVM's they were slow, executed and interpreted one bytecode at a time. Modern JVM uses JIT compilation unit to which we even call just-in-time compilation. These compilers executes several instructions in parallel. They are also called as Java HotSpot compiler because JVM uses them to find hot spots in bytecode. By the term hot spots we mean that JVM analyze bytecode by searching those spots in program which are executed frequently.



Example:

Creating and Editing

```
public class Demo {
public static void main(String[] args) {
System.out.println("Hello World!");
```

Saving

c:\java\labs\Demo.java

Compiling

javac Demo.java

Running

java Demo

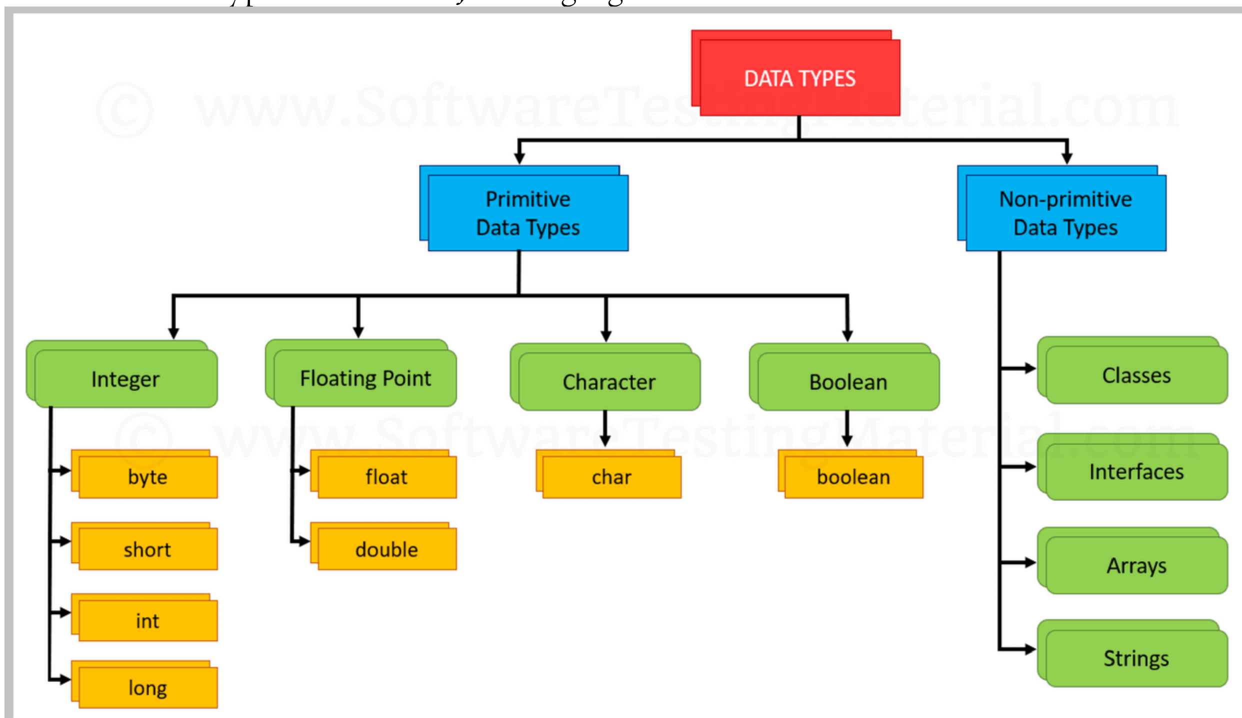
DATATYPES:

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.
2. Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

I. Primitive Data Types :

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.



There are eight primitive data types in Java:

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Boolean Data Type

- The Boolean data type is used to store only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false

Byte Data Type

- The byte data type is an example of primitive data type.
- It isan 8-bit signed two's complement integer.
- Its value-range lies between -128 to 127 (inclusive).
- Its minimum value is -128 and maximum value is 127. Its default value is 0.
- The byte data type is used to save memory in large arrays where the memory savings is most required.
- It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, byte b = -20

Short Data Type

- The short data type is a 16-bit signed two's complement integer.
- Its value-range lies between -32,768 to 32,767 (inclusive).
- Its minimum value is -32,768 and maximum value is 32,767.
- Its default value is 0.
- The short data type can also be used to save memory just like byte data type.

- A short data type is 2 times smaller than an integer.

Example: short s = 10000, short r = -5000

Int Data Type

- The int data type is a 32-bit signed two's complement integer.
- Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive).
- Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647.
- Its default value is 0.
- The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: int a = 100000, int b = -200000

Long Data Type

- The long data type is a 64-bit two's complement integer.
- Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive).
- Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807.
- Its default value is 0.
- The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type

- The float data type is a single-precision 32-bit IEEE 754 floating point.
- Its value range is unlimited.
- It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers.
- The float data type should never be used for precise values, such as currency.
- Its default value is 0.0F.

Example: float f1 = 234.5f

Double Data Type

- The double data type is a double-precision 64-bit IEEE 754 floating point.
- Its value range is unlimited.

- The double data type is generally used for decimal values just like float.
- The double data type also should never be used for precise values, such as currency.
- Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type

- The char data type is a single 16-bit Unicode character.
- Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).
- The char data type is used to store characters.

Example: char letterA = 'A'

2.Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

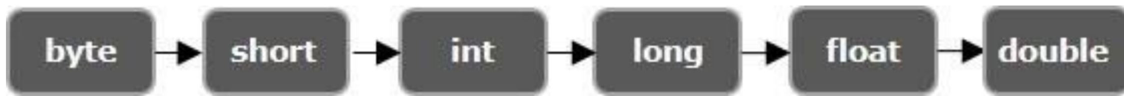
Examples : Non-primitive types are Strings, Arrays, Classes, Interface, etc.

TYPE CONVERSION / TYPE CASTING:

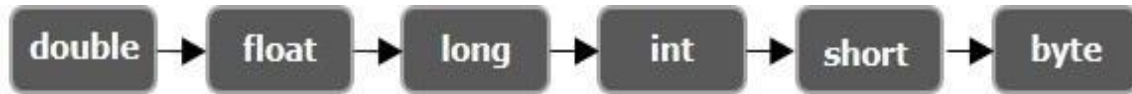
In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Converting one primitive datatype into another is known as type casting (type conversion) in Java. You can cast the primitive datatypes in two ways namely, Widening and, Narrowing.

1. **Implicite or Widening** − Converting a lower datatype to a higher datatype is known as widening. In this case the casting/conversion is done automatically therefore, it is known as implicit type casting. In this case both datatypes should be compatible with each other.



2. **Explicite or Narrowing –** Converting a higher datatype to a lower datatype is known as narrowing. In this case the casting/conversion is not done automatically, you need to convert explicitly using the cast operator "( )" explicitly. Therefore, it is known as explicit type casting. In this case both datatypes need not be compatible with each other.



CONDITIONAL STATEMENTS:

Conditional statements are:

1. if statement
2. if-else statement
3. nested if statement
4. if-else-if statement
5. Switch Case Statement

1.if statement:

The if statement is the most basic of all the control flow statements. The if statement tells our program to execute a certain section of code only if a particular test evaluates to true.

Syntax:

```
if(condition){
  Statement(s);
}
```

2.if-else statement:

If a condition is true then the section of code under if would execute else the section of code under else would execute.

Syntax:

```
if(condition) {

   Statement(s);
```

```
}
else {
  Statement(s);
}
```

3.Nested if statement:

An if statement inside another the statement. If the outer if condition is true then the section of code under outer if condition would execute and it goes to the inner if condition. If inner if condition is true then the section of code under inner if condition would execute.

Syntax:

```
if(condition_1) {
  Statement1(s);
  {
     if(condition_2) {
    Statement2(s);
     }
}
```

4.if-else-if statement

If a condition is true then the section of code under if would execute else the section of code under else would execute. Otherwise check the other condition. It will continue until condition true else it execute else statements.

Syntax:

```
if(condition) {
  Statement(s);
}
else if(condition) {
```

```
  Statement(s);
}
else if(condition) {
  Statement(s);
}
.
.
.
.
else{
  Statement(s);
}
```

5.Switch Case:

The switch statement in Java is a multi branch statement. We use this in Java when we have multiple options to select. It executes particular option based on the value of an expression.

Switch works with the byte, short, char, and int primitive data types. It also works with enumerated types, the String class, and a few special classes that wrap certain primitive types such as Character, Byte, Short, and Integer.

Syntax:

```
switch(expression) {
   case valueOne:
      //statement(s
      break;
   case valueTwo:
      //statement(s
      break;
   default: //optional
   //statement(s) //This code will be executed if all cases are not matched
```

}

LOOPS:

1. While Loop

The while loop loops through a block of code as long as a specified condition is true:

Syntax

```
while (condition) {
  // code block to be executed
}
```

2. For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed

3. The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
```

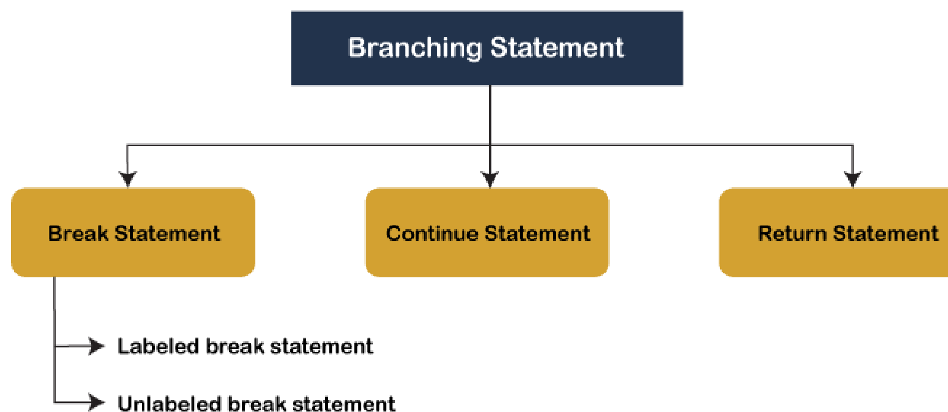
```
}
while (condition);
```

he loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

BRANCHING STATEMENTS:

Branching statements are the statements used to jump the flow of execution from one part of a program to another. The **branching statements** are mostly used inside the control statements. Java has mainly three branching statements, i.e., **continue, break**, and **return**. The **branching statements** allow us to exit from a control statement when a certain condition meet.

In Java, **continue** and **break** statements are two essential branching statements used with the control statements. The **break** statement breaks or terminates the loop and transfers the control outside the loop. The **continue** statement skips the current execution and pass the control to the start of the loop. The **return** statement returns a value from a method and this process will be done explicitly.



break Statement

The **labeled** and **unlabeled** break statement are the two forms of break statement in Java. The break statement is used for terminating a loop based on a certain condition.

```
for (int; testExpression; update){
   //Code
   if(condition to break){
      break;
   }
}
```

continue Statement

The **continue** statement is another branching statement used to immediately jump to the next iteration of the loop. It is a special type of loop which breaks current iteration when the condition is met and start the loop with the next iteration. In simple words, it continues the current flow of the program and stop executing the remaining code at the specified condition.

Syntax

```
control-flow-statement;
continue;
```

return Statement

The **return** statement is also a branching statement, which allows us to explicitly return value from a method. The return statement exits us from the calling method and passes the control flow to where the calling method is invoked. Just like the break statement, the return statement also has two forms, i.e., one that passes some value with control flow and one that doesn't.

Syntax

```
return value;
```

Or,

```
return;
```

CLASSES/OBJECTS

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Main.java

Create a class named "Main" with a variable x:

```
public class Main {
  int x = 5;
}
```

Create an Object

In Java, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

Example

Create an object called "myObj" and print the value of x:

```
public class Main {
  int x = 5;
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

METHOD DECLARATION:

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console.

Creating Method

Considering the following example to explain the syntax of a method −

Syntax

```
public static int methodName(int a, int b) {
   // body
```

}

Here,

public static − modifier

int − return type

methodName − name of the method

a, b − formal parameters

int a, int b − list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax −

Syntax

```
modifier returnType nameOfMethod (Parameter List) {
   // method body
}
```

The syntax shown above includes −

modifier − It defines the access type of the method and it is optional to use.

returnType − Method may return a value.

nameOfMethod − This is the method name. The method signature consists of the method name and the parameter list.

Parameter List − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.

method body − The method body defines what the method does with the statements.

METHOD OVERLOADING:

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Syntax:

```
void func() { ... }
void func(int a) { ... }
```

```
float func(double a) { ... }
float func(int a, float b) { ... }
```

Example:

```
class MethodOverloading {
    private static void display(int a){
        System.out.println("Arguments: " + a);
    }
    private static void display(int a, int b){
        System.out.println("Arguments: " + a + " and " + b);
    }
    public static void main(String[] args) {
        display(1);
        display(1, 4);
    }
}
```

METHOD INVOCATION:

The Java programming language provides two basic kinds of methods: instance methods and class (or static) methods. The difference between these two kinds of methods is:

Instance methods require an instance before they can be invoked; whereas class methods do not. Instance methods use dynamic (late) binding, whereas class methods use static (early) binding.

When the Java virtual machine invokes a class method, it selects the method to invoke based on the type of the object reference, which is always known at compile-time. On the other hand, when the virtual machine invokes an instance method, it selects the method to invoke based on the actual class of the object, which may only be known at run time.

The JVM uses two different instructions, shown in the following table, to invoke these two different kinds of methods: invoke virtual for instance methods, and invoke static for class methods.

UNIT-II

CONSTRUCTORS:

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void). Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

TYPES OF CONSTRUCTORS:

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

a) Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

<class_name>(){}

Example of default constructor

```
class Student{
//creating a default constructor
Student(){System.out.println("Welcome to java");}
//main method
public static void main(String args[]){
//calling a default constructor
Student b=new Student();
}
}
```

Output:

Welcome to java

Example of default constructor that displays the default values.

```
class Student3
{
 int id;
  String name;
  //method to display the value of id and name
  void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[])
{
  //creating objects
  Student3 s1=new Student3();
  Student3 s2=new Student3();
  //displaying values of the object
  s1.display();
  s2.display();
  }
  }
```

Output:

0 null
0 null

b) Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

```
class Student4
{
int id;
String name;
//creating a parameterized constructor
Student4(int i,String n)
{
id = i;

name = n;
}
//method to display the values
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[])
{
//creating objects and passing values
Student4 s1 = new Student4(10,"Abc");
Student4 s2 = new Student4(20,"Xyz");
//calling method to display the values of object
s1.display();
s2.display();
}
}
```

Output:

```
10 Abc
20 Xyz
```

CONSTRUCTOR OVERLOADING:

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
class Student5
```

```
{
int id;
String name;
int age;
//creating two arg constructor
Student5(int i,String n)
{
id = i;
name = n;
}
//creating three arg constructor
Student5(int i,String n,int a){
id = i;
name = n;
age=a;
}
void display()
{
System.out.println(id+" "+name+" "+age);
}
public static void main(String args[])
{
Student5 s1 = new Student5(10,"Abc");
Student5 s2 = new Student5(20"Xyz",25);
s1.display();
s2.display();
}
}
```

Output:

```
10 Abc 0
20 Xyz 25
```

Difference between constructor and method

Java Constructor	**Java Method**
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.

Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

Example of Copy Constructor

```
class Student6
{
int id;
String name;
//constructor to initialize integer and string
Student6(int i,String n)
{
id = i;
name = n;
}
//constructor to initialize another object
Student6(Student6 s)
{
id = s.id;
name =s.name;
}
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[])
{
Student6 s1 = new Student6(111,"Karan");
Student6 s2 = new Student6(s1);
s1.display();
s2.display();
```

}
}

CLEANING UP UNUSED OBJECTS :

The Java runtime environment deletes objects when it determines that they are no longer being used. This process is known as *garbage collection*.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are naturally dropped when the variable goes out of scope. Or you can explicitly drop an object reference by setting the value of a variable whose data type is a reference type to null.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

CLASS VARIABLES:

Declared inside the class definition (but outside any of the instance methods). They are not tied to any particular object of the class, hence shared across all the objects of the class. Modifying a class variable affects all objects instance at the same time.

- **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.
- **Instance variables** − Instance variables are declared in a class, but outside a method. When space is allocated for an object in the heap, a slot for each instance variable value is created. 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.
- **Local variables** − 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.

CLASS METHODS:

You learned from the Java Methods chapter that methods are declared within a class, and that they are used to perform certain actions:

Example

Create a method named myMethod() in Main:

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

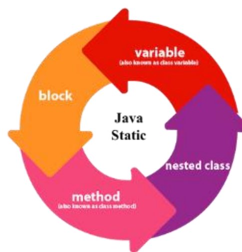
myMethod() prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

STATIC KEYWORD:

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



1) Java static variable

If you declare any variable as static, it is known as a static variable.

- he static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{
     int rollno;
     String name;
     String college="VDC";
}
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.

class Student{
     int rollno;
     String name;
     static String college = "VDC";
     //static method to change the value of static variable
     static void change(){
     college = "VDC";
     }
```

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

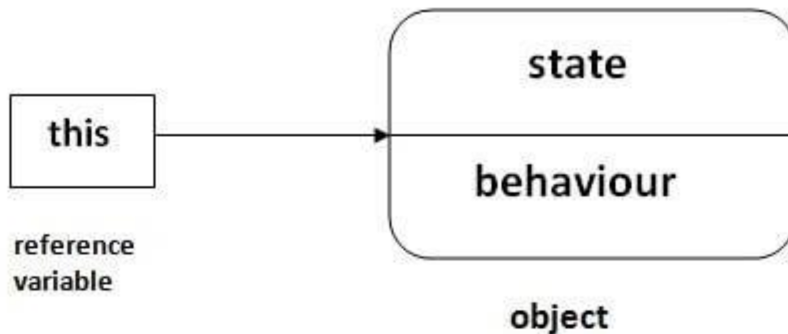
Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
```

```
    }
  }
```

THIS KEYWORD:

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.



Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling.

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes.

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).

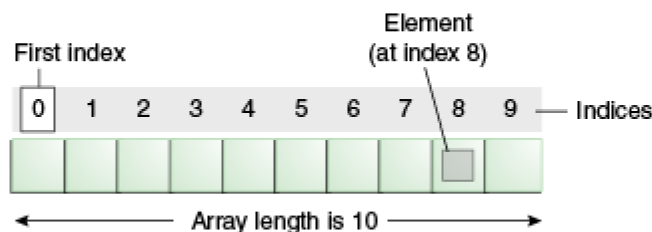
ARRAYS:

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

- Single Dimensional Array
- Multidimensional Array

SINGLE DIMENSIONAL ARRAY :

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

```
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5
for(int i=0;i<a.length;i++)
System.out.println(a[i]);

}

}
```

TWO – DIMENSIONAL ARRAY (2D-ARRAY):

Two – dimensional array is the simplest form of a multidimensional array. A two – dimensional array can be seen as an array of one – dimensional array for easier understanding.

Declaration – Syntax:
data_type[][] array_name = new data_type[x][y];
For example: int[][] arr = new int[10][20];
Initialization – Syntax:
array_name[row_index][column_index] = value;
For example: arr[0][0] = 1;

Multidimensional Arrays can be defined in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

Syntax:

data_type[1st dimension][2nd dimension][]..[Nth dimension] **array_name** = **new data_type**[size1][size2]….[sizeN];

where:

data_type: Type of data to be stored in the array. For example: int, char, etc.

dimension: The dimension of the array created.
For example: 1D, 2D, etc.

array_name: Name of the array

size1, size2, …, sizeN: Sizes of the dimensions respectively.

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

  int[][] arr = new int[10][20];
        arr[0][0] = 1;

        System.out.println("arr[0][0] = " + arr[0][0]);
    }
}
```

A command-line argument is an information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

Example

The following program displays all of the command-line arguments that it is called with -

```
public class CommandLine {
   public static void main(String args[]) {
      for(int i = 0; i<args.length; i++) {
         System.out.println("args[" + i + "]: " + args[i]);
      }
   }
}
```

INNER CLASSES:

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
1. class Java_Outer_class{
2.  //code
3.  class Java_Inner_class{
4.   //code
5.  }
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization:** It requires less code to write

INHERITANCE :

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Advantages:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

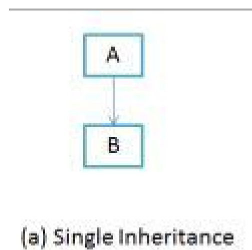
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Types of inheritance in Java:

1) Single Inheritance

Single inheritance is damn easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



(a) Single Inheritance

Single Inheritance example program in Java

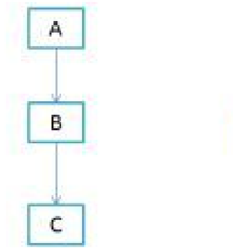
```
Class Parent
{
  public void display()
  {
   System.out.println("Parent class method");
  }
}

Class Child extends Parent
{
  public void show()
  {
   System.out.println("Child class method");
```

```
    }
    public static void main(String args[])
    {
      Child obj = new Child();
      obj.display(); //calling super class method
      obj.show(); //calling local method
   }
}
```

2) Multilevel Inheritance

Multilevel inheritance refers to a mechanism in Object Oriented technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.



(d) Multilevel Inheritance

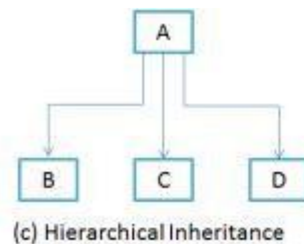
Multilevel Inheritance example program in Java

```
Class GrandParent
{
   public void method1()
   {
     System.out.println("Class GrandParent method");
   }
}
Class Parent extends GrandParent
{
public void method2()
{
System.out.println("class Parent method");
}
}
Class Child extends Parent
{
```

```
  public void method3()
  {
    System.out.println("class Child method");
  }
  public static void main(String args[])
  {
    Child obj = new Child();
    obj.method1(); //calling grand parent class method
    obj.method2(); //calling parent class method
    obj.method3(); //calling local method
  }
}
```

3) Hierarchical Inheritance

In such kind of inheritance one class is inherited by many **sub classes.** In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.



(c) Hierarchical Inheritance

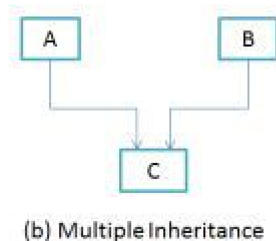
Example of Hierarchical Inheritance:

```
class A
{
  public void methodA()
  {
    System.out.println("method of Class A");
  }
}
class B extends A
{
  public void methodB()
  {
    System.out.println("method of Class B");
  }
}
class C extends A
{
```

```
  public void methodC()
  {
    System.out.println("method of Class C");
  }
}
class D extends A
{
  public void methodD()
  {
    System.out.println("method of Class D");
  }
}
class JavaExample
{
  public static void main(String args[])
  {
    B obj1 = new B();
    C obj2 = new C();
    D obj3 = new D();
    //All classes can access the method of class A
    obj1.methodA();
    obj2.methodA();
    obj3.methodA();
  }
}
```

4) Multiple Inheritance

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.



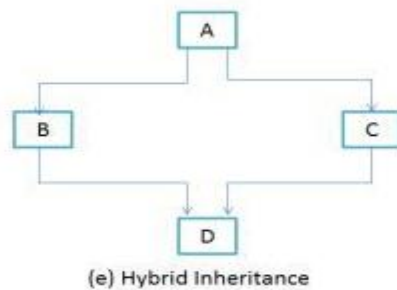
(b) Multiple Inheritance

Note 1: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Note 2: Most of the new OO languages like **Small Talk, Java, C# do not support Multiple inheritance**. Multiple Inheritance is supported in C++.

5) Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

Example of the hybrid inheritance.

```
class C
{
  public void disp()
  {
        System.out.println("C");
  }
}

class A extends C
{
  public void disp()
  {
        System.out.println("A");
  }
}

class B extends C
{
```

```
   public void disp()
   {
          System.out.println("B");
   }

}
class D extends A
{
   public void disp()
   {
          System.out.println("D");
   }
   public static void main(String args[]){

          D obj = new D();
          obj.disp();
   }
}
```

METHOD OVERRIDING :

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example of method overriding

```
    class Vehicle{
//defining a method
void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
```

```
class Bike2 extends Vehicle{
//defining the same method as in the parent class
void run(){System.out.println("Bike is running safely");}
public static void main(String args[]){
Bike2 obj = new Bike2();//creating object
obj.run();//calling method
}

}
```

SUPER KEYWORD :

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
c lass Animal{

void eat(){System.out.println("eating...");}
  }
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
  d.work();
     }}
```

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
     }}
```

FINAL KEYWORD:

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
obj.run();
 }
}
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
```

```
honda.run();
  }
}
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
  class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
      public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

ABSTRACT CLASS:

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Example of abstract class

```
1. abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

1. abstract void printStatus();//no method body and abstract

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
obj.run();
}
}
```

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

```
abstract class Bike{
Bike(){System.out.println("bike is created");}
abstract void run();
void changeGear(){System.out.println("gear changed");}
}
//Creating a Child class which inherits Abstract class
class Honda extends Bike{
void run(){System.out.println("running safely..");}
}
//Creating a Test class which calls abstract and non-abstract methods
class TestAbstraction2{
public static void main(String args[]){
Bike obj = new Honda();
obj.run();
```

```
obj.changeGear();
}
    }
```

INTERFACE:

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritances in Java.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface. Since Java 9, we can have **private methods** in an interface.

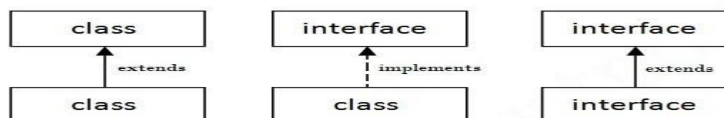
n interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{
   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Interface Example

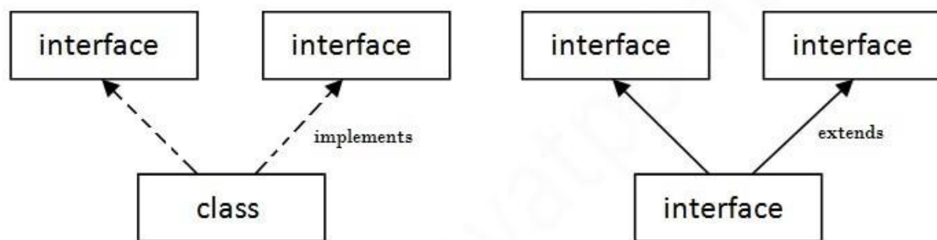
In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}
    public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example Multiple inheritance

```
interface Printable{
void print();
}
interface Showable{
void show();
}
A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
A7 obj = new A7();
```

```
obj.print();
obj.show();
 }
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

PACKAGES:

A package is a collection of similar types of Java entities such as classes, interfaces, subclasses, exceptions, errors, and enums. A package can also contain sub-packages.

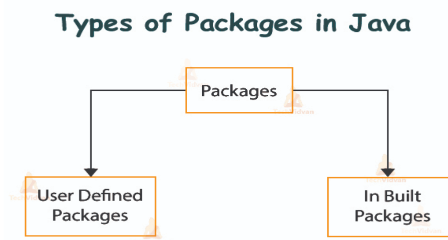
Advantages:

- Make easy searching or locating of classes and interfaces
- Avoid naming conflicts. For example, there can be two classes with the name Student in two packages, university.csdept.Student and college.itdept.Student
- Implement data encapsulation (or data-hiding).
- Provide controlled access: The access specifiers protected and default have access control on package level.
- Reuse the classes contained in the packages of other programs.
- Uniquely compare the classes in other packages.

Types of Packages

They can be divided into two categories:

1. Java API packages or built-in packages and
2. User-defined packages.



1. Java API packages or built-in packages

Java provides a large number of classes grouped into different packages based on a particular functionality.

Examples:

java.lang: It contains classes for primitive types, strings, math functions, threads, and exceptions.

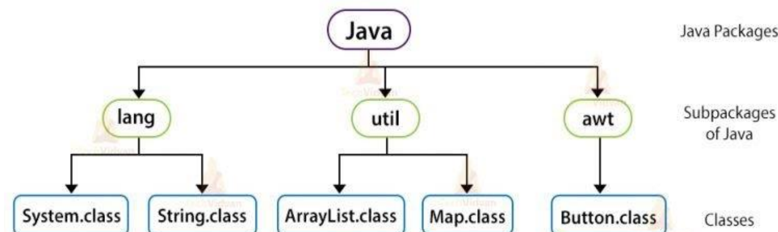
java.util: It contains classes such as vectors, hash tables, dates, Calendars, etc.

java.io: It has stream classes for Input/Output.

java.awt: Classes for implementing Graphical User Interface – windows, buttons, menus, etc.

java.net: Classes for networking

java. Applet: Classes for creating and implementing applets



2. User-defined packages

As the name suggests, these packages are defined by the user. We create a directory whose name should be the same as the name of the package. Then we create a class inside the directory.

Creating a Package

Java supports a keyword called "package" which is used to create user-defined packages in java programming. It has the following general form:

package packageName;

Here, packageName is the name of package.

For example:

```
package myPackage;

public class A {

// class body

}
```

Compiling a Package

If you are using an IDE (Integrated Development Environment), then for compiling the package,

The syntax:

javac -d directory javaFileName

For example,

javac -d . Example.java

Executing Package Program

You need to use a fully qualified name e.g. com.srinivad.MyClass etc to run the class.

To Compile:

javac -d . MyClass.java

Here -d represents the destination. The . represents the current folder.

To run:

java com.srinivas.MyClass

Package Import

import keyword which is used to access package and its classes into the java program.

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

1. without import the package (import package.*;)
2. import package with specified class(fully qualified name)
3. import package with all classes(import package.classname;)

ACCESS PROTECTION IN PACKAGES:

Packages in Java add another dimension to access control. Both classes and packages are a means of data encapsulation. While packages act as containers for classes and other subordinate packages, classes act as containers for data and code. Because of this interplay between packages and classes, Java packages addresses four categories of visibility for class members:

- Sub-classes in the same package
- Non-subclasses in the same package
- Sub-classes in different packages
- Classes that are neither in the same package nor sub-classes

The table below gives a real picture of which type access is possible and which is not when using packages in Java:

	Private	No Modifier	Protected	Public

Same Class	Yes	Yes	Yes	Yes
Same Package Subclasses	No	Yes	Yes	Yes
Same Package Non-Subclasses	No	Yes	Yes	Yes
Different Packages Subclasses	No	No	Yes	Yes
Different Packages Non-Subclasses	No	No	No	Yes

WRAPPER CLASSES:

A Wrapper class is a class which contains the primitive data types (int, char, short, byte, etc). In other words, wrapper classes provide a way to use primitive data types (int, char, short, byte, etc) as objects. These wrapper classes come under java.util package.

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Primitive Type	**Wrapper class**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

a) Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing,

```
public class AutoBoxingTest {
   public static void main(String args[]) {
      int num = 10; // int primitive
      Integer obj = Integer.valueOf(num); // creating a wrapper class object
      System.out.println(num + "" + obj);
   }
}
```

Output

10 10

B) Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.

```
public class UnboxingTest {
   public static void main(String args[]) {
      Integer obj = new Integer(10); // Creating Wrapper class object
      int num = obj.intValue(); // Converting the wrapper object to primitive datatype
      System.out.println(num + "" + obj);
   }
}
```

Output

10 10

STRING CLASS:

String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

There are two ways to create String object:

A)By string literal

Java String literal is created by using double quotes.

For Example:

String s="welcome";

B)By new keyword

String s=new String("Welcome");

JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

STRING BUFFER CLASS:

StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```
class StringBufferExample{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}
```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
```

```
}
}
```

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

UNIT-III

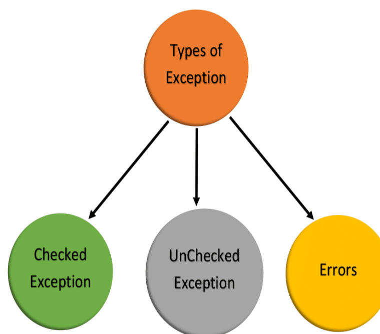
EXCEPTION:

Exception is an abnormal condition.
In Java, an exception is an event that disrupts the normal flow of the program.
It is an object which is thrown at runtime.

Types of Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error



1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions
e.g. IOException, SQLException etc.
Checked exceptions are checked at compile-time.

2) Unchecked Exception

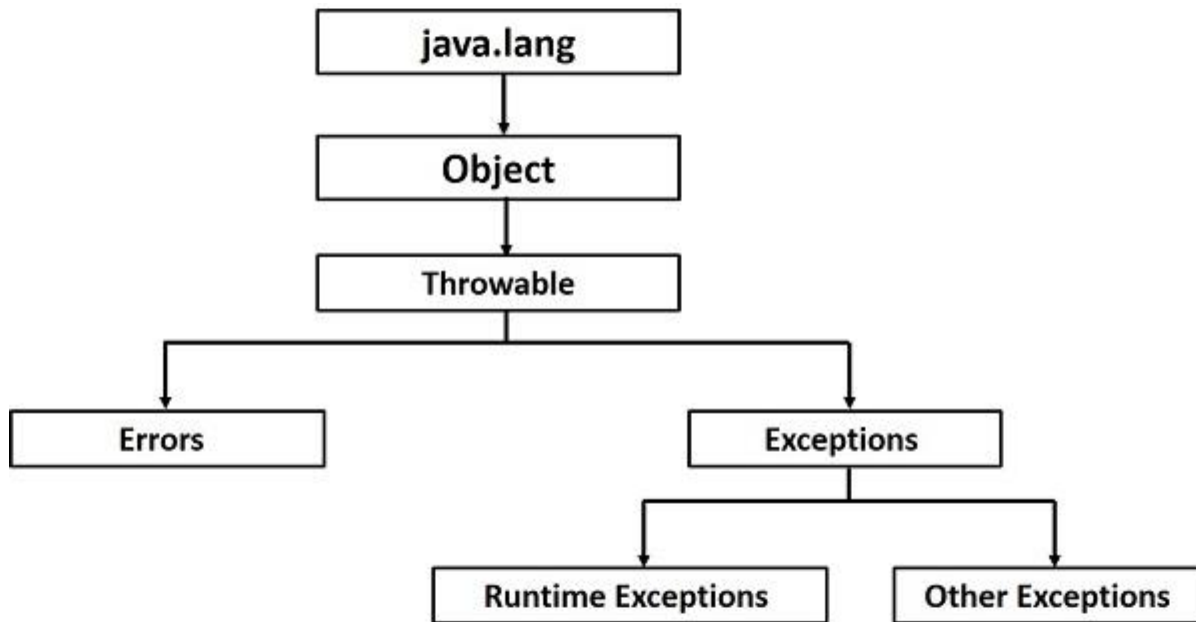
The classes which inherit RuntimeException are known as unchecked exceptions
e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Exception Handling

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



Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.

throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

a)try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

1. try{
2. //code that may throw an exception
3. }catch(Exception_class_Name ref){}

Syntax of try-finally block

1. try{
2. //code that may throw an exception
3. }finally{}

b) catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Example:

```
public class TryCatchExample3 {

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

```
{
int data=50/0; //may throw exception
              // if exception occurs, the remaining statement will not exceute
System.out.println("rest of the code");
}
   // handling the exception
catch(ArithmeticException e)
{
   System.out.println(e);
}
}
}
```

Output:

java.lang.ArithmeticException: / by zero

Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

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

c) finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.

Example:

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

d) throw keyword

The Java throw keyword is used to explicitly throw an exception.
We can throw either checked or unchecked exception in java by throws keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.
The **syntax** of java throw keyword is given below.

```
throw exception;
```

The example of throw IOException.

```
throw new IOException("sorry device error);
```

e) throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.

2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

USER DEFINED EXCEPTION:

User Defined Exception or custom exception is creating your own exception class and throws that exception using ‘throw’ keyword. This can be done by extending the class Exception.

The help of custom exception, you can have your own exception and message.

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
class TestCustomException1{
     static void validate(int age)throws InvalidAgeException{
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }
  public static void main(String args[]){
      try{
      validate(13);
      }catch(Exception m){System.out.println("Exception occured: "+m);}

      System.out.println("rest of the code...");
  }
}
```

Output:

Exception occured: InvalidAgeException:not valid
rest of the code...

MULTITHREADING:

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

How to create thread
There are two ways to create a thread:
1. By extending Thread class
2. By implementing Runnable interface.

THREAD CLASS:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.
Commonly used Constructors of Thread class:
o Thread()
o Thread(String name)
o Thread(Runnable r)
o Thread(Runnable r,String name)

Commonly used methods of Thread class:
1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.

6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

RUNNABLE INTERFACE:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

o A new thread starts(with new callstack).
o The thread moves from New state to the Runnable state.
o When the thread gets a chance to execute, its target run() method will run.

Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){ System.out.println("thread is running...");
}
public static void main(String args[]){ Multi t1=new Multi();
t1.start();
} }
```

Output:thread is running...

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
```

```
 System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
} }
```

Output:thread is running...

LIFE CYCLE OF A THREAD (THREAD STATES) :

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
But for better understanding the threads, we are explaining it in the 5 states.
The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

PRIORITY OF A THREAD (THREAD PRIORITY):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
System.out.println("running thread name is:"+Thread.currentThread().getName());
System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
}
public static void main(String args[]){
```

```
TestMultiPriority1 m1=new TestMultiPriority1();
TestMultiPriority1 m2=new TestMultiPriority1();
m1.setPriority(Thread.MIN_PRIORITY);
m2.setPriority(Thread.MAX_PRIORITY);
m1.start();
m2.start();
} }
```

Output:

running thread name is:Thread-0 running thread priority is:10
running thread name is:Thread-1 running thread priority is:1

SYNCHRONIZATION:

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource.
When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{
wait();
}
catch(Exception e){}
}
this.amount-=amount; System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
 System.out.println("going to deposit...");
this.amount+=amount;
 System.out.println("deposit completed... ");
notify();
}
}
class Test{
public static void main(String args[]){
```

```
final Customer c=new Customer(); new Thread(){
public void run(){
c.withdraw(15000);
}
}
start();
new Thread(){
public void run(){
c.deposit(10000);}
}
start();
}}
```

Output:

going to withdraw...
Less balance; waiting for deposit... going to deposit...
deposit completed... withdraw completed

FILES AND I/O:

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.

- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream.** Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;
public class CopyFile {
public static void main(String args[]) throws IOException { FileInputStream in = null;
FileOutputStream out = null; try {
in = new FileInputStream("input.txt");
```

```
out = new FileOutputStream("output.txt"); int c;
while ((c = in.read()) != -1) {
out.write(c);
}
}finally {
if (in != null) {
in.close();
}
if (out != null) { out.close();
}} }}
```

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile {
public static void main(String args[]) throws IOException {
FileReader in = null; FileWriter out = null; try {
in = new FileReader("input.txt"); out = new FileWriter("output.txt"); int c;
while ((c = in.read()) != -1) { out.write(c);}
}finally {
if (in != null) {
in.close();}
if (out != null) { out.close();
}} }}
```

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as**System.in**.

- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.

- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "

Example

```
import java.io.*;
public class ReadConsole {
public static void main(String args[]) throws IOException { InputStreamReader cin = null;
try {
cin = new InputStreamReader(System.in); System.out.println("Enter characters, 'q' to quit."); char
c;
do {
c = (char) cin.read(); System.out.print(c);
} while(c != 'q');
}finally {
if (cin != null) {
cin.close();
} } }}
```

This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit. 1
1
e e q q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.
Here is a hierarchy of classes to deal with Input and Output streams.
The two important streams are **FileInputStream** and **FileOutputStream**

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.
Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello"); InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

- ByteArrayInputStream
- DataInputStream

FILE OUTPUT STREAM:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.
Here are two constructors which can be used to create a FileOutputStream object.
Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello"); OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

- ByteArrayOutputStream
- DataOutputStream

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;
public class fileStreamTest {
public static void main(String args[]) {
 try {
byte bWrite [] = {11,21,3,40,5};
OutputStream os = new FileOutputStream("test.txt");
```

```
 for(int x = 0; x < bWrite.length ; x++) {
os.write( bWrite[x] ); // writes the bytes
}
os.close();
InputStream is = new FileInputStream("test.txt");
int size = is.available();
for(int i = 0; i < size; i++) {
 System.out.print((char)is.read() + " ");
}
is.close();
} catch (IOException e) {
System.out.print("Exception");
} }}
```

SCANNER CLASS:

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc. To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:

```
Scanner in = new Scanner(System.in);
```

To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class. For Example:

```
Scanner in = new Scanner("Hello Javatpoint");
```

Example

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
```

```
        System.out.println("Name is: " + name);
        in.close();
        }
    }
```

Output:

Enter your name: sonoo jaiswal
Name is: sonoo jaiswal

Input Types

In the example above, we used the nextLine() method, which is used to read Strings. To read other types, look at the table below:

Method	Description
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

In the example below, we use different methods to read data of various types:

Example

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter name, age and salary:");
```

```
    // String input
    String name = myObj.nextLine();

    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}
```

BUFFERED INPUTSTREAM CLASS:

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.
- public class BufferedInputStream extends FilterInputStream

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

Example of Java BufferedInputStream

Let's see the simple example to read data of file using BufferedInputStream:

```
package com.javatpoint;

import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
```

```
FileInputStream fin=new FileInputStream("D:\\testout.txt");
BufferedInputStream bin=new BufferedInputStream(fin);
int i;
while((i=bin.read())!=-1){
 System.out.print((char)i);
}
bin.close();
fin.close();
}catch(Exception e){System.out.println(e);}
}
}
```

Here, we are assuming that you have following data in **"testout.txt"** file:
javaTpoint

BUFFERED OUTPUTSTREAM CLASS:

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

```
OutputStream os= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt"));
```

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

RANDOM ACCESSFILE CLASS:

The **Java.io.RandomAccessFile** class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file.

Class declaration

Following is the declaration for **Java.io.RandomAccessFile** class −

```
public class RandomAccessFile extends Object
implements DataOutput, DataInput, Closeable
```

Class constructors

S.N.	Constructor & Description
1	**RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
2	**RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Methodsinherited

This class inherits methods from the following classes −

- Java.io.Object

Java.io.File Class in Java

The File class is Java's representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

- It is an abstract representation of file and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
- First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file- system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

How to create a File Object?

A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

File a = new File("/usr/local/bin/geeks");

defines an abstract file name for the geeks file in directory /usr/local/bin. This is an absolute abstract file name.

Program to check if a file or directory physically exist or not.

```
// In this program, we accepts a file or directory name from
// command line arguments. Then the program will check if
// that file or directory physically exist or not and
// it displays the property of that file or directory.
*import java.io.File;
// Displaying file property class fileProperty
{
```

```
public static void main(String[] args) {
//accept file name or directory name through command line args
String fname =args[0];
//pass the filename or directory name to File object
File f = new File(fname);
//apply File class methods on File object
System.out.println("File name :"+f.getName());
System.out.println("Path: "+f.getPath());
System.out.println("Absolute path:" +f.getAbsolutePath());
System.out.println("Parent:"+f.getParent());
System.out.println("Exists :"+f.exists());
if(f.exists())
{
System.out.println("Is writeable:"+f.canWrite());
System.out.println("Is readable"+f.canRead());
System.out.println("Is a directory:"+f.isDirectory());
System.out.println("File Size in bytes "+f.length());
}
}
}
```

Output:

File name :file.txt Path: file.txt

Absolute path:C:\Users\akki\IdeaProjects\codewriting\src\file.txt Parent:null

Exists :true

Is writeable:true Is readabletrue

Is a directory:false File Size in bytes 20

UNIT-IV

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.
o Secured
o It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

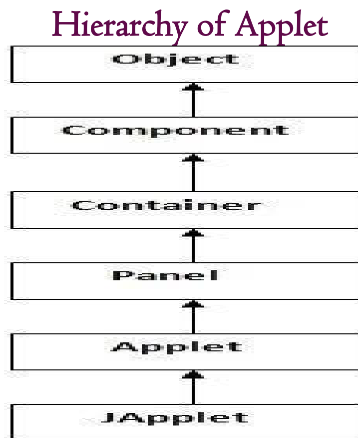
Drawback of Applet

o Plugin is required at client browser to execute applet.

LIFECYCLE OF JAVA APPLET:

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Hierarchy of Applet



Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

APPLET CLASS:

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the 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:

Difference between Applet and Application programming

```
c:\>javac First.java
c:\>appletviewer First.java
```

	Java Applet	Java Application
User graphics	Inherently graphical	Optional
Memory requirements	Java application requirements plus web browser requirements	Minimal java application requirements
Distribution	Linked via HTML and transported via HTTP	Loaded from the file system or by a custom class loading process
Environmental input	Browser client location and size; parameters embedded in the host HTML document	command-line parameters
Method expected by the virtual Machine	init- initialization method start-startup method stop pause/ deactive method destroy-termination method paint-drawing method	Main - startup method
Typical applications	public-access order-entry systems for the web, online multimedia persentations, web page animation	Network server, multimedia kiosks, developer tools, appliance and consumer electronics control and navigation.

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

```
1. <html>
2. <body>
3. <applet code="UseParam.class" width="300" height="300">
4. <param name="msg" value="Welcome to applet">
5. </applet>
6. </body>
7. </html>
```

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 has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

- **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 process the event an then returns.

EVENT CLASSES AND LISTENER INTERFACES:

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

APPLET DISPLAY METHODS:

Applets are displayed in a window and they use the AWT to perform input and output functions. To output a string to an applet, use drawString( ), which is a member of the Graphics class. Typically, it is called from within either update() or paint( ). It has the following general form:

void drawString(String message, int x, int y)

Here, *message* is the string to be output beginning at *x, y.* In a java window, the upper-left corner is location 0, 0. The drawString( ) method will not recognize newline characters. If we want to start a line of text on another line, we must do it manually, specifying the X,Y location where we want the line to begin.

To set the background color of an applet's window, we use setBackground( ). To set the foreground color, we use setForeground(). These methods are defined by Component, and have the general forms:

void setBackground(Color *newColor*)

void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class Color defines the following values that can be used to specify colors:

Color.black Color.magenta Color.blue Color.orange Color.cyan Color.pink Color.darkGray Color. red Color.gray Color.white Color. green Color.yellow Color.lightGray

For example, this sets the background color to blue and the text color to yellow:

setBackground(Color.blue);

setForeground(Color.yellow);

We can change these colors as and when required during the execution of the applet. The default foreground color is black. The default background color is light gray. We can obtain the current settings for the background and foreground colors by calling getBackground( ) and getForeground( ), respectively. They are also defined by Component and bear the general form:

Color getBackground()

Color getForeground()

The example below shows a simple applet to change the color of the foreground and background.

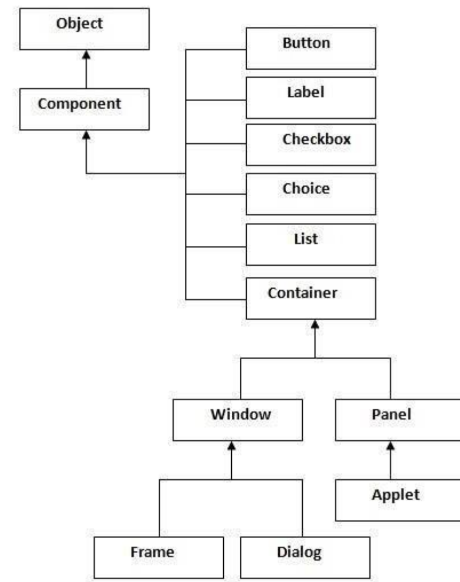
```
import java.awt.*;
 import java.applet.*;
 /* <applet code="Applet_Prog" width=500 height=550> </applet>*/
 public class Applet_Prog extends Applet
 {
         public void paint (Graphics g)
        {
          setBackground(Color.BLUE);
          setForeground(Color.RED);
          g.drawString("Using Colors in An Applet" , 100,250);
        }
  }
```

AWT:

Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java. Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS. The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

JAVA AWT HIERARCHY:

The hierarchy of Java AWT classes are given below.



Container:

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.
o By extending Frame class (inheritance)

o By creating the object of Frame class (association)

AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First extends Frame{ First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){ First f=new First();
}}
```

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

JAVA SWING :

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.

DIFFERENCE BETWEEN AWT AND SWING:

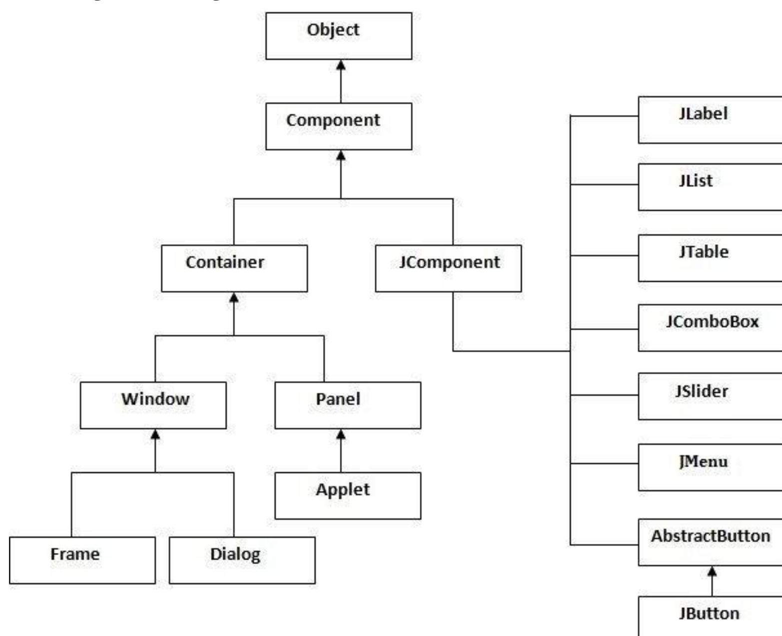
No.	Java AWT	Java Swing
1)	AWT components are **platform dependent.**	Java swing components are **platform-independent**
2)	AWT components are **heavyweight.**	Swing components are **lightweight**
3)	AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components**such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC.**

Commonly used Methods of Component class

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.

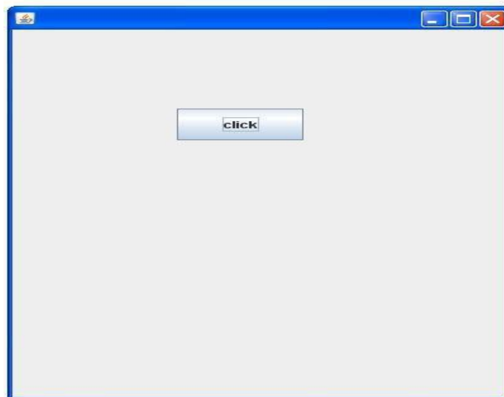


Java Swing Examples
There are two ways to create a frame:
o By creating the object of Frame class (association)

o By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.
Simple Java Swing Example
Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers f.setVisible(true);//making the frame visible
} }
```



CONTAINERS:
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

```
1. <html>
2. <body>
3. <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.

```
1. public class JDialog extends Dialog implements WindowConstants, Accessible, RootPaneConta
iner
```

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.

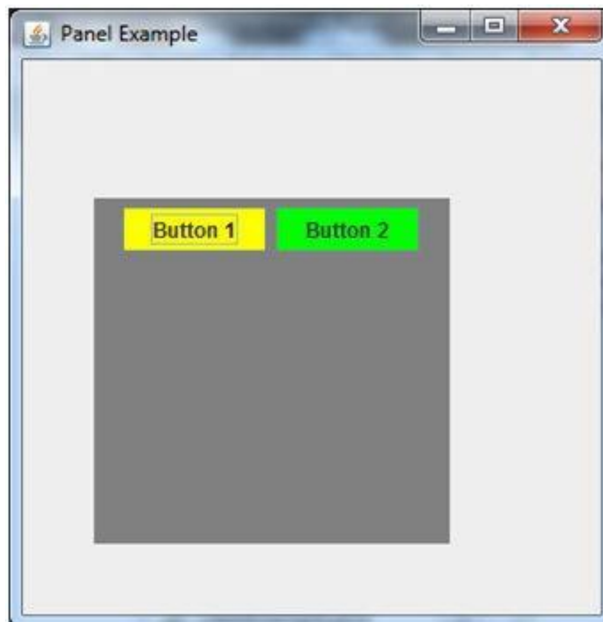
JPanel class declaration

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

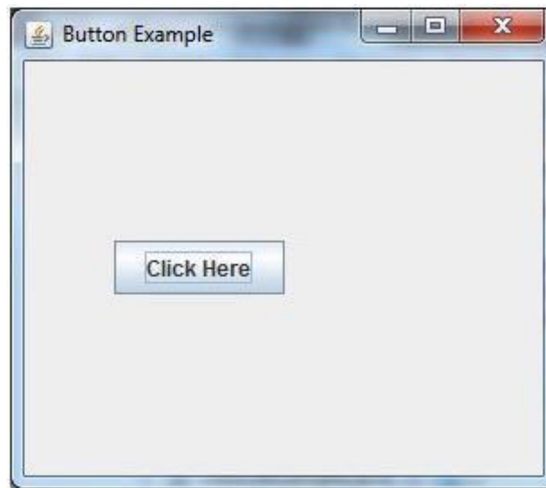
Let's see the declaration for javax.swing.JButton class.

```
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);
} }
```



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:

Constructor	Description
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.
JLabel(String s, Icon i ,int horizontalAlignment)	Creates a JLabel instance with the sp

Commonly used Methods:

Methods	Description
String getText()	t returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.
void setHorizontalAlignment(int alignment)	It sets the alignment of the label's contents along the X axis.

Icon getIcon()	It returns the graphic image that the label displays.
int getHorizontalAlignment()	It returns the alignment of the label's contents along the X axis.

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.

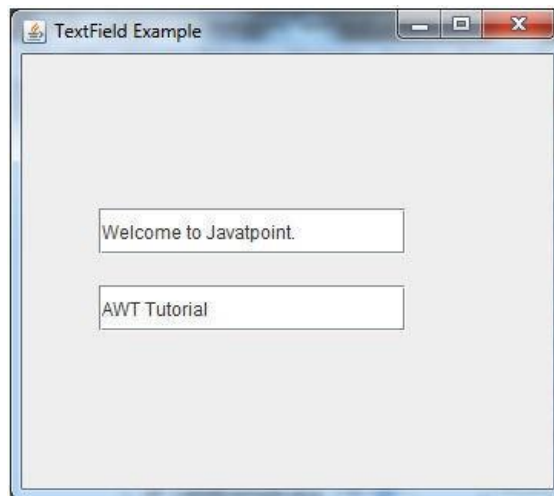
JTextField class declaration

Let's see the declaration for javax.swing.JTextField class.

1. **public class** JTextField **extends** JTextComponent **implements** SwingConstants

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.s etLayout(null);
f.setVisible(true);
}
}
```



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

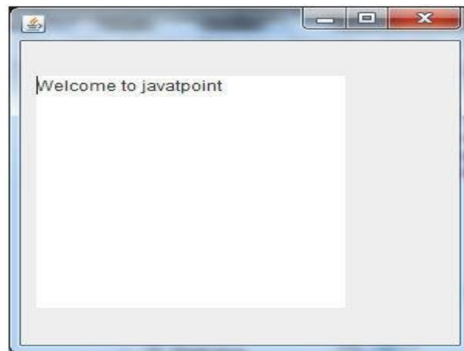
JTextArea class declaration

Let's see the declaration for javax.swing.JTextArea class.

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);
}}
```

LAYOUT MANAGEMENT

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.

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:

o **BorderLayout():** creates a border layout but with no gaps between the components.

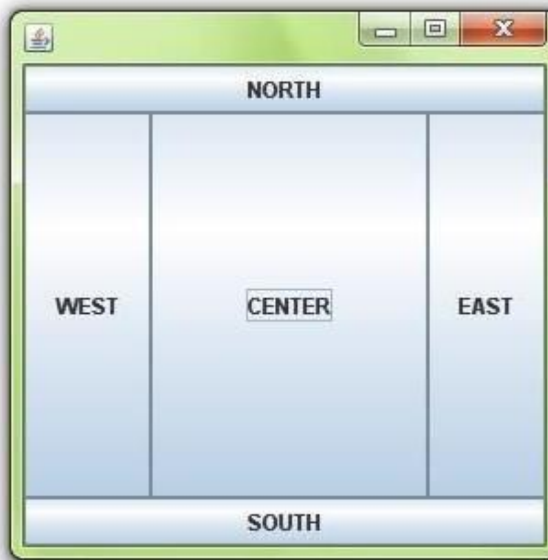
o **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class:

```
import java.awt.*;
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);
f.setVisible(true);
}
public static void main(String[] args)
{
new Border();
```

} }

Output:



Java GridLayout

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

```
import java.awt.*;
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);
f.s etVisible(true);

}
public static void main(String[] args) {
        new MyGridLayout(); }}
```

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);
f.s etVisible(true);

}
public static void main(String[] args) {
new MyFlowLayout();
          } }
```

DATABASE HANDLING USING JDBC:

Connceting to DB

What is JDBCDriver?

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The *Java.sql* package that ships with JDK, contains various classes with their behaviours defined and their actual implementaions are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

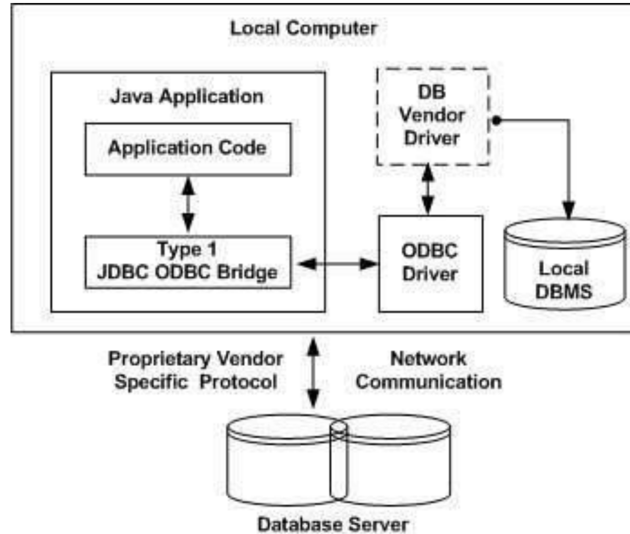
JDBC DRIVERS TYPES :

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below −

Type 1: JDBC-ODBCBridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

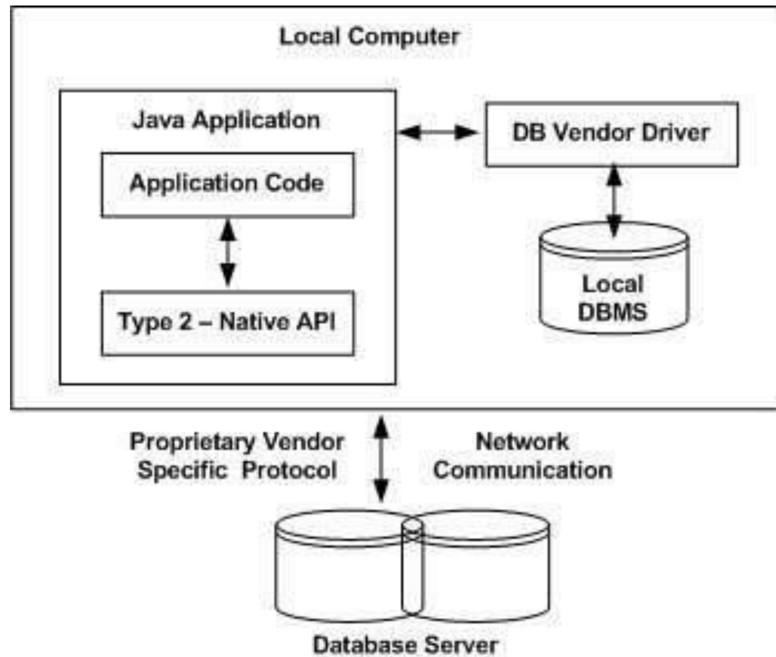


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver

Type 2: JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

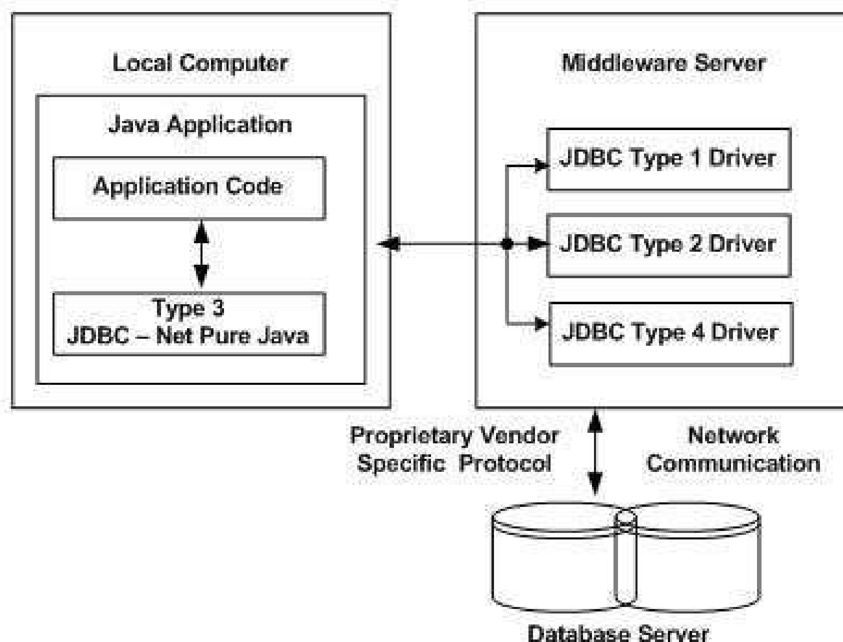


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

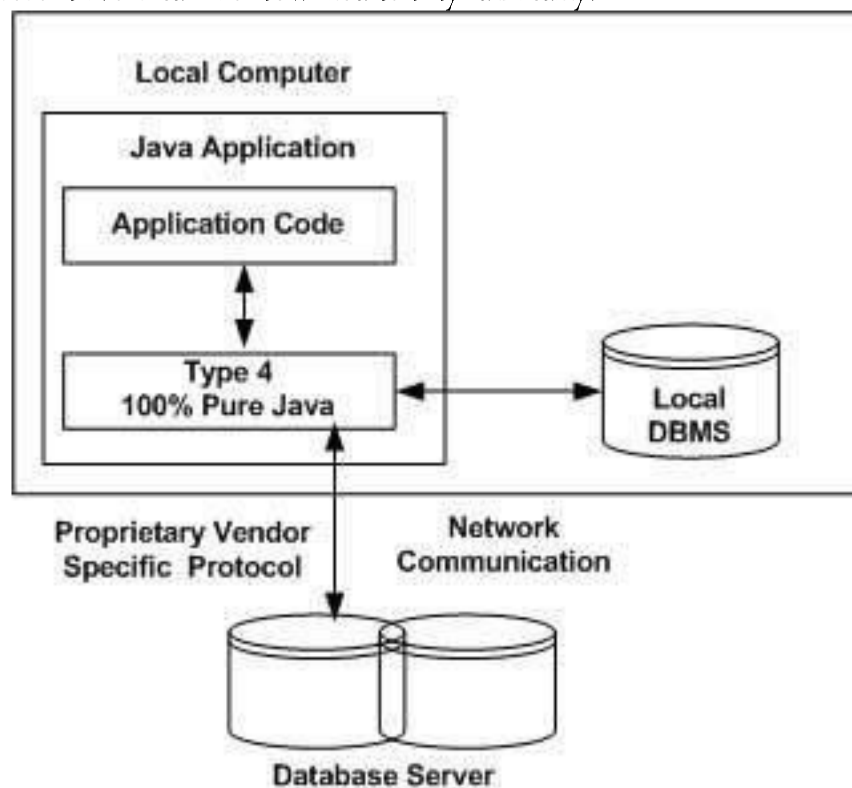
This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.
Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.
This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

The type I driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

Example to connect to the mysql database in java

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.

2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.

3. **Username:** The default username for the mysql database is **root**.

4. **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
1. create database sonoo;

2. use sonoo;

3. create table emp(id int(10),name varchar(40),age int(3));
```

Example to Connect Java Application with mysql database

In this example, sonoo is the database name, root is the username and password.

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
 try{ Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
        //here sonoo is database name, root is username and password
}}
```

Example to connect to the mysql database in java

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.

2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.

3. **Username:** The default username for the mysql database is **root**.

4. **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

1. create database sonoo;

2. use sonoo;

3. create table emp(id **int**(10),name varchar(40),age **int**(3)); **Example to Connect Java Application with mysql database**

In this example, sonoo is the database name, root is the username and password.

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
        Connection con=DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/sonoo","root","root"); //here sonoo is database name,
        root is username and password

}
}
```

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −

- **Navigational methods:** Used to move the cursor around.

- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.

- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.
JDBC provides the following connection methods to create statements with desired ResultSet −

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**

- **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrencyof ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.

Viewinga Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row. There is a get method for each of the possible data types, and each get method has two versions

- One that takes in a column name.

 One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.
There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.
For a better understanding, let us study Viewing - Example Code.
Updatinga Result Set
The ResultSet interface contains a collection of update methods for updating the data of a result set.
As with the get methods, there are two update methods for each data type −
 One that takes in a column name.

 One that takes in a column index.
For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods –

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method, except that the column is specified by its name instead of its index.