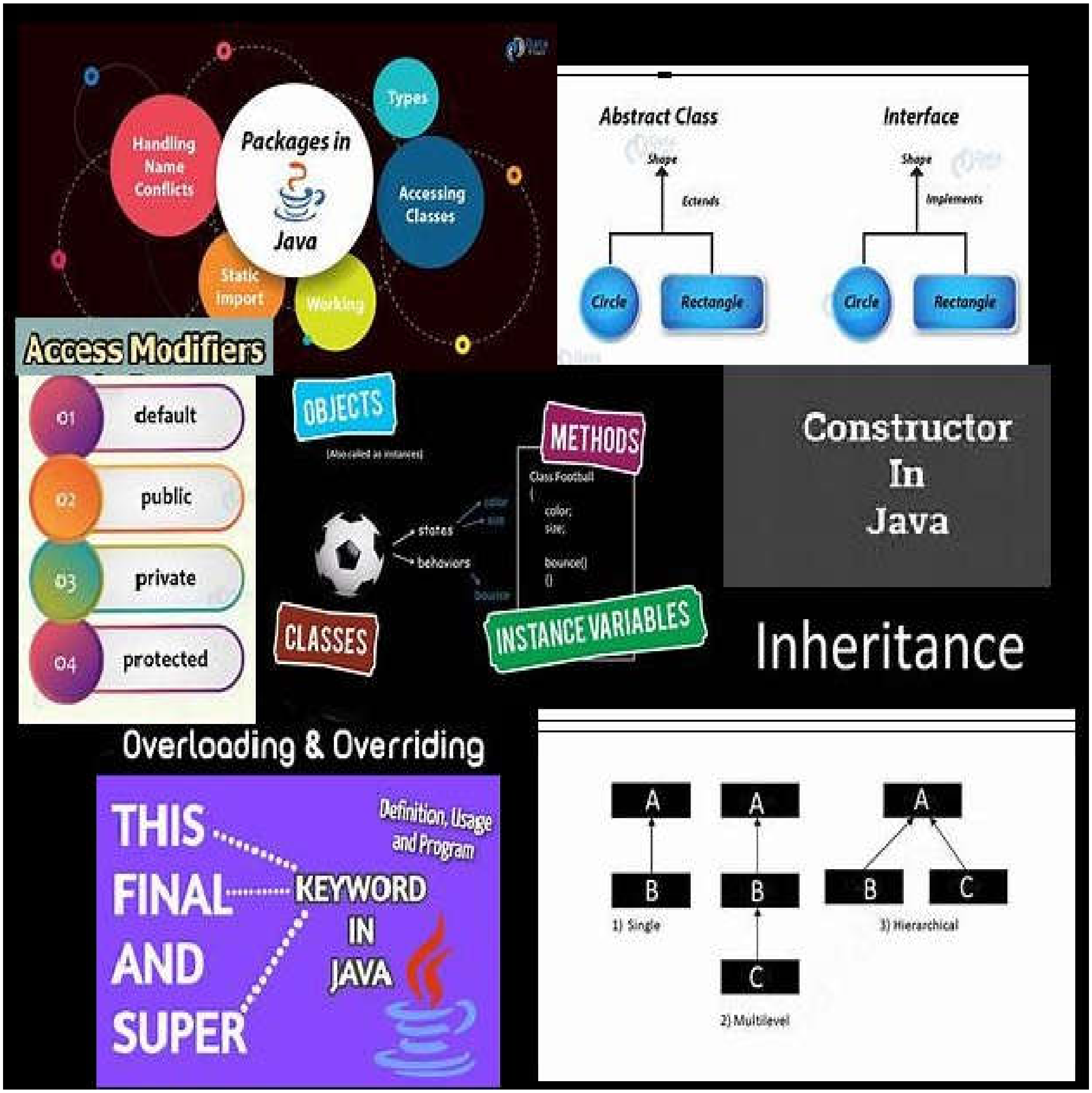
# PROGRAMMING IN JAVA



O.S. ABDUL QADIR

JAMAL MOHAMED COLLEGE,
TRICHY – 20.

**SEMESTER – III: CORE-V: JAVA PROGRAMMING**

| | | | |
|---|---|---|---|
| **Subject Code:** | **20UCA3CC5** | **Max. Marks:** | **100** |
| **Hours:** | **4** | **Internal Marks:** | **25** |
| **Credits:** | **4** | **External Marks:** | **75** |

**Objective** To understand the basic concepts of Object Oriented Programming with Java language

**UNIT I** **15 Hours**

Introduction to Java Programming: Introduction – Features of Java – Java Developer Kit. Java Language Fundamentals: The Building Blocks of Java – Data Types – Variable Declarations: Declaring, Initializing and Variables – Variable Types in Java. Wrapper Classes – Operators and Assignment – Control Structures – Arrays – #Strings#

**UNIT II** **15 Hours**

Java as an OOP Language – Defining Classes – Defining Methods – Knowing This – Passing Arguments to Methods – Overloading Methods – Constructor Methods – Inheritance– Overriding Methods – Modifiers: The Four Ps of Protection – Finalizing Classes, Methods and Variables – Abstract Classes and Methods – Packages – Interfaces

**UNIT III** **15 Hours**

Exception Handling: Introduction – Basics of Exception Handling in Java – Exception Hierarchy – Constructors and Methods in Throwable Class – Handling Exceptions in Java – Throwing User Defined Exceptions. Multithreading – Overview of Threads – Creating Threads – Thread Life-cycle – #Thread Priorities and Thread Scheduling#

**UNIT IV** **15 Hours**

Files and I/O Streams: Java I/O – File Streams – FileInputStream and FileOutputStream – Filter Streams – RandomAccessFile – Serialization. Applets: Introduction – Java Applications Versus Java Applets – Applet Life Cycle – Working with applets – The HTML APPLET Tag

**UNIT V** **15 Hours**

The Abstract Window Toolkit: Basic Classes in AWT – Drawing with Graphics class - Class Hierarchy in AWT – Event Handling – AWT Controls – Layout Managers

**Text Book**

P. Radha Krishna, *Object Oriented Programming through JAVA*, Universities Press, 2007

**Unit I:** Chapter 1 & 2 **Unit II:** Chapter 3 **Unit III**: Chapter 5 & 6
**Unit IV:** Chapter 7 & 8 **Unit V:** Chapter 10

**Reference Book:**

Herbert Schildt, *The Complete* Reference Java, Fifth Edition, Tata McGRAW-Hill, 2008

**Web Reference:**

https://www.programiz.com/java-programming

# Contents

## UNIT IV

# UNIT I

## Introduction to JAVA

- 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.

**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.

**Java Example**

```
class Simple
{
   public static void main(String args[])
    {
    System.out.println("Hello Java");
    }
}
```

## Editions of Java

Each edition of Java has different capabilities.
There are three editions of Java:

1. **Java Standard Editions (JSE):** It is used to create programs for a desktop computer.
2. **Java Enterprise Edition (JEE):** It is used to create large programs that run on the server and manages heavy traffic and complex transactions.
3. **Java Micro Edition (JME):** It is used to develop applications for small devices such as set-top boxes, phone, and appliances.

## Features of Java

- The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language.
- The features of Java are also known as java buzzwords.
- A list of most important features of Java language is given below.
    1. Simple
    2. Object-Oriented
    3. Portable
    4. Platform independent
    5. Secured
    6. Robust
    7. Architecture neutral
    8. Interpreted
    9. High Performance
    10. Multithreaded
    11. Distributed
    12. Dynamic

1. **Simple**
   Java is very easy to learn, and its syntax is simple, clean and easy to understand.
   According to Sun, Java language is a simple programming language because:
   - Java syntax is based on C++ (so easier for programmers to learn it after C++).
   - Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
   - There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java

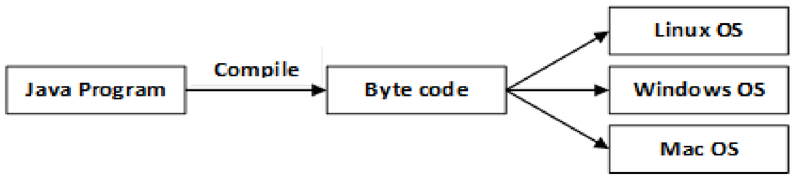
2. **Object-oriented**
   Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.
   - **Object-oriented programming (OOPs)** is a methodology that simplifies software development and maintenance by providing some rules.
   - Basic concepts of **OOPs** are:
     1. **Object** - An object is a real-world entity that can be identified distinctly. Data fields with their current values represent the state of an object (also known as its properties or attributes).
     2. **Class** - A class is a template or blueprint or prototype that defines data members and methods of an object. An object is the instance of the class.
     3. **Inheritance** - The mechanism in which one class acquire all the features of another class. It achieve by using **"extends"** keyword. It facilitates the reusability of the code.
     4. **Polymorphism** - The polymorphism is the ability to appear in many forms. There are two types of polymorphism: run time polymorphism and compile-time polymorphism.
     5. **Abstraction** - A method of hiding irrelevant information from the user. Java use abstract class and interface to achieve abstraction.
     6. **Encapsulation** - The process of binding data and functions into a single unit. A class is an example of encapsulation.

3. **Platform Independent**
   - ✓ Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.
   - ✓ A platform is the hardware or software environment in which a program runs.
   - ✓ There are two types of platforms software-based and hardware-based.
   - ✓ Java provides a software-based platform.

     The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms.

     It has two components:
     1. Runtime Environment
     2. API(Application Programming Interface)

   Java code can be run on multiple platforms. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).



4. **Secured**
   Java is best known for its security. With Java, we can develop virus-free systems.
   Java is secured because:
   - ✓ **No explicit pointer**
   - ✓ **Java Programs run inside a virtual machine sandbox**
   - ✓ **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine (JVM) dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
   - ✓ **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
   - ✓ **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

   Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

5. **Robust**
   Robust simply means strong.
   Java is robust because:
   - ✓ It uses strong memory management.
   - ✓ There is a lack of pointers that avoids security problems.
   - ✓ There is automatic garbage collection in java which runs on the JVM to get rid of objects which are not being used by a Java application anymore.
   - ✓ There are exception handling and the type checking mechanism in Java. All these points make Java robust.

6. **Architecture-neutral**
   Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
   In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

7. **Portable**
   Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

8. **High-performance**
   Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.
   Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

9. **Distributed**
   Java is distributed because it facilitates users to create distributed applications in Java.

RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

**10. Multi-threaded**

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads.
The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

**11. Dynamic**

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand.
It also supports functions from its native languages, i.e., C and C++.
Java supports dynamic compilation and automatic memory management (garbage collection).

## Java Development Kit (JDK)

- 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:
  1. Standard Edition Java Platform
  2. Enterprise Edition Java Platform
  3. 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.

### Components of JDK

Following is a list of primary components of JDK:

| Sl. No | Component Name | Description |
|---|---|---|
| 1 | Appletviewer | This tool is used to run and debug Java applets without a web browser. |
| 2 | Javac | It specifies the Java compiler, which converts source code into Java bytecode. |
| 3 | Javadoc | The documentation generator, which automatically generates documentation from source code comments |
| 4 | Javap | the class file disassembler. |
| 5 | Javah | the C header and stub generator, used to write native methods. |
| 6 | JConsole | Java Monitoring and Management Console. |
| 7 | Jar | Specifies the archiver, which packages related class libraries into a single JAR file. This tool also helps manage JAR files. |

## The Building blocks of a Java program

1. Lexical Tokens
    a. Identifiers
    b. Keywords
2. Literals
    a. Integer Literals
    b. Floating-Point Literals
    c. Boolean Literals
    d. Character Literals
    e. String Literals
3. White Spaces
4. Comments

### 1. Lexical tokens

- ➢ A lexical token may consist of one or more characters, and every single character is in exactly one token.
- ➢ The tokens can be keywords, comments, numbers, white space, or strings. All lines should be terminated by a semi-colon (;).

**a. Identifiers**

- ✓ Identifier a name chosen by the programmer to identify something defined inside a program
- ✓ There are some rules in Java that you must follow to form the identifier name
- ✓ An identifier consists of a number of characters
    - ▪ There is no limit on the number of characters in the identifier (but do not try using identifiers that are too long because you will have to type it yourself...)
    - ▪ The first character of an identifier must be one of the following:
        - **A letter (a, b, ..., z, A, B, ..., Z), or**
        - **The underscore character**
    - ▪ The subsequent characters of an identifier must be one of the following:
        - **A letter (a, b, ..., z, A, B, ..., Z), or**
        - **The underscore character, or**
        - **A digit (0, 1, ..., 9)**
    - ▪ Identifiers are **case-sensitive**!

**b. Keywords**

- ✓ Keywords have special meaning in a programming language.
- ✓ It is also known as ***Reserved Words***
- ✓ You cannot use a keyword as identifier (keywords are reserved for a specific purpose!)

| | | | | | | |
|---|---|---|---|---|---|---|
| abstract | continue | for | new | switch | case | try |
| extends | default | goto* | package | synchronized | enum**** | catch |
| boolean | do | if | private | this | instanceof | char |
| Break | double | implements | protected | throw | return | final |
| Byte | else | import | public | throws | transient | interface |
| Class | finally | long | strictfp** | volatile | int | static |
| const* | float | native | super | while | short | void |

## 2. Literals

- A literal is a source code representation of a fixed value.
- They are represented directly in the code without any computation.
- Literals can be assigned to any primitive type variable.

### a. Integral Literals in Java

We can specify the integer literals in 4 different ways –

*a.* ***Decimal (Base 10)***

Digits from 0-9 are allowed in this form. ***int x = 101;***

*b.* ***Octal (Base 8)***

Digits from 0 – 7 are allowed. It should always have a prefix 0.

***int x = 0146;***

*c.* ***Hexa-Decimal (Base 16)***

Digits 0-9 are allowed and also characters from a-f are allowed in this form. Furthermore, both uppercase and lowercase characters can be used.

***int x = 0X123Face;***

### b. Boolean Literals in Java

✓ They allow only two values i.e. true and false.

***boolean b = true;***

### c. Floating-Point Literals in Java

✓ Here, datatypes can only be specified in decimal forms and not in octal/hexadecimal.

✓ Every floating type is a double type and this the reason why we cannot assign it directly to float variable, to escape this we use f or F as suffix, and for double we use d or D.

### d. Char Literals in Java

There are the four types of char literals in Java

1. ***Single quote*** - Java Literal can be specified to a char data type as a single character within a single quote.

   **char ch = 'a';**

2. ***Char literal***

   - A char literal in Java can specify as integral literal which also represents the Unicode value of a character.

   - Furthermore, an integer can specify in decimal, octal and even hexadecimal type, but the range is 0-65535.

   **char ch = 062;**

3. ***Escape sequences***

   - A character preceded by a backslash (\) is an escape sequence and has a special meaning to the compiler.

   - The following table shows the Java escape sequences.

| **Escape Sequence** | **Description** |
|---|---|
| \t | Inserts a tab in the text at this point. |
| \b | Inserts a backspace in the text at this point. |
| \n | Inserts a newline in the text at this point. |
| \r | Inserts a carriage return in the text at this point. |
| \f | Inserts a form feed in the text at this point. |

| | |
|---|---|
| \' | Inserts a single quote character in the text at this point. |
| \" | Inserts a double quote character in the text at this point. |
| \\ | Inserts a backslash character in the text at this point. |

4. ***Unicode representation***

   - Char literals can specify in Unicode representation '\uxxxx'. Here XXXX represents 4 hexadecimal numbers.

   **char ch = '\u0061';**// Here /u0061 represent a.

5. **String literals in Java**

   Java String literals are any sequence of characters with a double quote.

   **String s = "Hello";**

   They may not contain unescaped newline or linefeed characters.

3. **White Spaces**
   - ✓ White space can contain the characters for tabs, blanks, newlines, and form feeds.
   - ✓ These characters are ignored except when they serve to separate other tokens.
   - ✓ However, blanks and tabs are significant in strings.

4. **Comments**
   - ➢ A text inside a Java program that is ignored by the Java compiler.
   - ➢ Comments are used to annotate the program to help humans understand the operation of the Java program.

   **Single line comment** syntax:

   ```
   // ... single line comment
   ```

   The text on the line following the symbol // will be ignored

   **Multiple lines comment** syntax:

   ```
    /* comment line 1
      comment line 2
        ...
   */
   ```

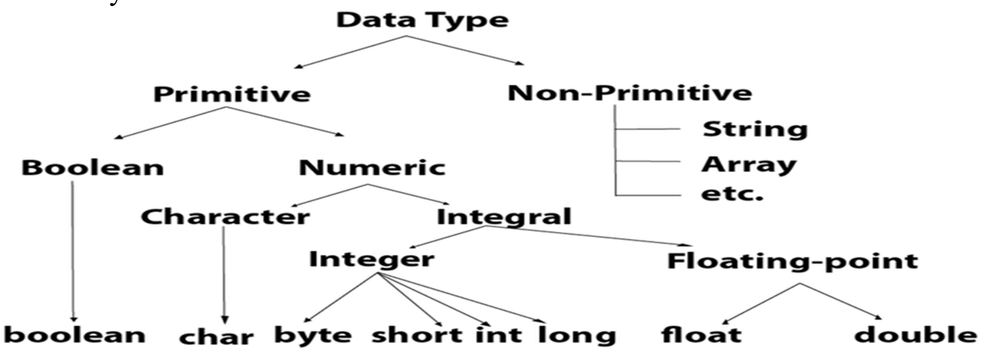
   All text between the comment brackets /* ..... */ will be ignored

## Data Types in Java

- ➢ 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.



**Java 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.
- ✓ Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.
- ✓ There are 8 types of primitive data types:

1. **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.

2. **byte data type**
   - It is an 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. It saves space because a byte is 4 times smaller than an integer.

3. **char data type**
   - It 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.

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

5. **int data type**
   - It is a 32-bit signed two's complement integer.
   - Its value-range lies between - 2,147,483,648 to 2,147,483,647. Its minimum value is - 2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

6. **long data type**
   - It is a 64-bit two's complement integer.
   - Its value-range lies between minimum value is 9,223,372,036,854,775,808 and 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.

7. **float data type**
   - It is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited.
   - Its default value is 0.0F.

8. **double data type**
   - It is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. Its default value is 0.0d.

| Data Type | Default Value | Default size | Example |
|---|---|---|---|
| Boolean | False | 1 bit | Boolean one = false |
| char | '\u0000' | 2 byte | char letterA = 'A' |
| byte | 0 | 1 byte | byte a = 10, byte b = -20 |
| short | 0 | 2 byte | short s = 10000, short r = -5000 |
| int | 0 | 4 byte | int a = 100000, int b = -200000 |
| long | 0L | 8 byte | long a = 100000L, long b = -200000L |
| float | 0.0f | 4 byte | float f1 = 234.5f |
| double | 0.0d | 8 byte | double d1 = 12.3 |

## Variables in Java

**Variable**

Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

**Types of Variables:**

There are three types of variables:

1. Local Variables
2. Instance Variables
3. Static or Class variables

1. **Local Variables**

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

2. **Instance Variable**

A variable declared inside the class but outside the body of the method, is called instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created.

3. **Static or Class variables**

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

## Wrapper classes in Java

- ➢ The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.
- ➢ The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

**Use of Wrapper classes in Java**

- ✓ Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.

- ✓ **Change the value in Method**: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- ✓ **Serialization**: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- ✓ **Synchronization**: Java synchronization works with objects in Multithreading.
- ✓ **java.util package**: The java.util package provides the utility classes to deal with objects.
- ✓ **Collection Framework**: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.
- ✓ **The eight classes of the java.lang package are known as wrapper classes in Java**.
- ✓ The list of eight wrapper classes are given below:

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

**Autoboxing**

- ✓ The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- ✓ Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

  **Wrapper class Example: Primitive to Wrapper**

  //Java program to convert primitive into objects. Autoboxing example of int to Integer

```
public class WrapperExample1
{
        public static void main(String args[])
        {
            int a=20;
            Integer i=Integer.valueOf(a);//converting int into Integer explicitly
            Integer j=a;// now compiler will write Integer.valueOf(a) internally
            System.out.println(a+" "+i+" "+j);
        }
}
```

**Unboxing**

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

- ✓ It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

  **Wrapper class Example: Wrapper to Primitive**

  //Java program to convert object into primitives. Unboxing example of Integer to int

```
public class WrapperExample2
{
        public static void main(String args[])
        {
                Integer a=new Integer(3);
                int i=a.intValue();//converting Integer to int explicitly
                int j=a;// now compiler will write a.intValue() internally
                System.out.println(a+" "+i+" "+j);
        }
}
```

## Operators in Java

- Operator in Java is a symbol which is used to perform operations. For ex: +, -, *, / etc.
- There are many types of operators in Java which are given below:
  - ✓ Unary Operator,
  - ✓ Arithmetic Operator,
  - ✓ Shift Operator,
  - ✓ Relational Operator,
  - ✓ Bitwise Operator,
  - ✓ Logical Operator,
  - ✓ Ternary Operator and
  - ✓ Assignment Operator.

**Java Unary Operator**

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- ✓ incrementing/decrementing a value by one
- ✓ negating an expression
- ✓ inverting the value of a boolean

**Example: ++ and --**

```
class OperatorExample
{
        public static void main(String args[])
        {
                int x=10, a=10;
                System.out.println(x++);//10 (11)
                System.out.println(++x);//12
                System.out.println(x--);//12 (11)
                System.out.println(--x);//10
                System.out.println(a++ + ++a);//10+12=22
               System.out.println(~a);//-11 (minus of total positive value which
        starts from 0)
        }
}
```

**Java Arithmetic Operators**

Java arithmatic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

**Example**

```
class OperatorExample
{
        public static void main(String args[])
        {
                int a=10;
                int b=5;
                System.out.println(a+b);//15
                System.out.println(a-b);//5
                System.out.println(a*b);//50
                System.out.println(a/b);//2
                System.out.println(a%b);//0
        }
}
```

**Java Left Shift Operator**

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

**Example**

```
class OperatorExample
{
        public static void main(String args[])
        {
                System.out.println(10<<2);//10*2^2=10*4=40
                System.out.println(10<<3);//10*2^3=10*8=80
                System.out.println(20<<2);//20*2^2=20*4=80
                System.out.println(15<<4);//15*2^4=15*16=240
        }
}
```

**Java Right Shift Operator**

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

**Example**

```
class OperatorExample
{
        public static void main(String args[])
        {
                System.out.println(10>>2);//10/2^2=10/4=2
                System.out.println(20>>2);//20/2^2=20/4=5
                System.out.println(20>>3);//20/2^3=20/8=2
        }
}
```

**Java Shift Operator Example: >> vs >>>**

```
class OperatorExample
{
        public static void main(String args[])
        {
           //For positive number, >> and >>> works same
           System.out.println(20>>2);
           System.out.println(20>>>2);
           //For negative number, >>> changes parity bit (MSB) to 0
           System.out.println(-20>>2);
           System.out.println(-20>>>2);
        }
}
```

**Java AND Operator Example: Logical && and Bitwise &**

- ✓ The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.
- ✓ The bitwise & operator always checks both conditions whether first condition is true or false.

**Example**

```
class OperatorExample
{
        public static void main(String args[])
        {
                int a=10;
                int b=5;
                int c=20;
                System.out.println(a<b&&a<c);//false && true = false
                System.out.println(a<b&a<c);//false & true = false
        }
}
```

**Java OR Operator Example: Logical || and Bitwise |**

- ✓ The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.
- ✓ The bitwise | operator always checks both conditions whether first condition is true or false.

**Example**

```
class OperatorExample
{
        public static void main(String args[])
        {
                int a=10;
                int b=5;
                int c=20;
                System.out.println(a>b||a<c);//true || true = true
                System.out.println(a>b|a<c);//true | true = true
                //|| vs |
                System.out.println(a>b||a++<c);//true || true = true
                System.out.println(a);//10 because second condition is not checked
                System.out.println(a>b|a++<c);//true | true = true
                System.out.println(a);//11 because second condition is checked
        }
}
```

**Java Ternary Operator**

- ✓ It is used as one liner replacement for if-then-else statement and used a lot in Java programming. This is the only conditional operator which takes three operands.

**Example**

```
class OperatorExample
{
        public static void main(String args[])
        {
                int a=2;
                int b=5;
                int min=(a<b)?a:b;
                System.out.println(min);
        }
}
```

**Java Assignment Operator**

- ✓ Java assignment operator is one of the most common operator.
- ✓ It is used to assign the value on its right to the operand on its left.

**Example**

```
class OperatorExample
{
        public static void main(String args[])
        {
                int a=10;
                int b=20;
                a+=4;//a=a+4 (a=10+4)
                b-=4;//b=b-4 (b=20-4)
                System.out.println(a);
                System.out.println(b);
        }
}
```

**Java Operator Precedence**

| **Operator Type** | **Category** | **Precedence** |
|---|---|---|
| Unary | postfix | expr++ expr-- |
| prefix | ++expr --expr +expr -expr ~ ! | |
| Arithmetic | multiplicative | * / % |
| | additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| | equality | == != |

| | | |
|---|---|---|
| Bitwise | bitwise AND | & |
| | bitwise exclusive OR | ^ |
| | bitwise inclusive OR | \| |
| Logical | logical AND | && |
| | logical OR | \|\| |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= *= /= %= &= ^= \|= <<= >>= >>>= |

**Control Structures**

**Blocks**

- ✓ A block statement is a sequence of zero or more statements enclosed in braces.
- ✓ A block statement is generally used to group together several statements, so they can be used in a situation that requires you to use a single statement.
- ✓ In some situations, you can use only one statement. If you want to use more than one statement in those situations, you can create a block statement by placing all your statements inside braces, which would be treated as a single statement.

  An example of block statement is given below.

```
{ //block start
    int var = 20;
    var++;
} //block end
```

***Scope of variables***

- ✓ Please note that all the variables declared in a block statement can only be used within that block. In other words, you can say that all variables declared in a block have local scope.

```
{ //block start
    int var = 20;
    var++;
} //block end
// A compile-time error. var has been declared inside a block and
// so it cannot be used outside that block
System.out.println(var);
```

  - ✓ Similarily, you can also nest a block statement inside another block statement. All the variables declared in the enclosing blocks (outer blocks) are available to the enclosed blocks (inner blocks). However, the variables declared in the enclosed inner blocks are not available in enclosing outer blocks.

***During object creation***

- ✓ Another thing which may interest you that block statements need not to be only inside methods.
- ✓ You can write them as other class members such as class variables and methods.

```
public class MyDemoAction
{
   private Interger variable = 10;
    public MyDemoAction(){
      System.out.println("MyDemoAction Constructor");
    }
```

```
        {
           //Non-static block statement
        }
        static {
           //Static block statement
        }
        private void someMethod() {
           System.out.println("HowToDoInJava.com");
        }
    }
```

- ✓ Please note that when block statements are declared in such way, non-static blocks will be executed everytime an instance of class is created.
- ✓ Static block will be execute only once when class is loaded by JVM clas loaders (Much like other static variables present at class level).

**Loops in Java**
In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

1. while loop
2. do…while loop
3. for loop

**1. while loop**

The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

***Syntax:***

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

***Example***

```
public class WhileExample
{
            public static void main(String[] args)
            {
               int i=1;
               while(i<=10)
            {
                  System.out.println(i);
                  i++;
            }
        }
}
```

**2. do…..while loop**

- ✓ The Java do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

- ✓ The Java do-while loop is executed at least once because condition is checked after loop body.

***Syntax:***

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

***Example:***

```
public class DoWhileExample
{
        public static void main(String[] args)
        {
           int i=1;
           do
           {
              System.out.println(i);
              i++;
           }while(i<=10);
        }
}
```

3. **For loop**

- ✓ The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

It consists of four parts:

1. ***Initialization:*** It is the initial condition which is executed once when the loop starts.
   Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. ***Condition:*** It is the second condition which is executed each time to test the condition of the loop.
   It continues execution until the condition is false.
   It must return boolean value either true or false.
   It is an optional condition.
3. ***Statement:*** The statement of the loop is executed each time until the second condition is false.
4. ***Increment/Decrement:*** It increments or decrements the variable value. It is an optional condition.

   ***Syntax:***

   ```
   for(initialization;condition;incr/decr)
   {
   //statement or code to be executed
   }
   ```

   ***Example:***

   ```
   //Java Program to demonstrate the example of for loop which prints table of 1
           public class ForExample
           {
                   public static void main(String[] args)
                   {
   ```

```
                //Code of Java for loop
                for(int i=1;i<=10;i++)
                {
                    System.out.println(i);
                }
        }
}
```

**Branching Statement**

- 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 if Statement**

✓ The Java if statement tests the condition. It executes the if block if condition is true.

***Syntax:***

```
if(condition)
{
//code to be executed
}
```

***Example:***

```
//Java Program to demonstate the use of if statement.
public class IfExample
{
        public static void main(String[] args)
        {
                int age=20;
                if(age>18)          //checking the age
                {
                        System.out.print("Age is greater than 18");
                }
        }
}
```

**Java if-else Statement**

✓ The Java if-else statement also tests the condition.

✓ It executes the “if” block if condition is true otherwise else block is executed.

***Syntax:***

```
if(condition)
{
        //code if condition is true
}
else
{
        //code if condition is false
}
```

***Example: ODD or EVEN***

//A Java Program to demonstrate the use of if-else statement.

```
public class IfElseExample
{
        public static void main(String[] args)
        {
          //defining a variable
          int number=13;
          //Check if the number is divisible by 2 or not
          if(number%2==0)
          {
             System.out.println("even number");
          }
        else
        {
             System.out.println("odd number");
        }
     }
}
```

**Break and Continue Statement**

- ✓ 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 unlabeled break statement is used to terminate the loop that is inside the loop.
    - It is also used to stop the working of the switch statement.
    - We use the unlabeled break statement to terminate all the loops available in Java.

    **Syntax:**

    ```
    for (int; testExpression; update)
    {
       //Code
       if(condition to break)
       {
          break;
       }
    }
    ```
- ✓ The continue statement skips the current execution and pass the control to the start of the loop.
    - It continues the current flow of the program and stop executing the remaining code at the specified condition.

    **Syntax**

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

    **Example: ContinueExample.java**

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

```
        {
            int x = 1;
            int y = 10;
            //Using do while loop for using continue statement
            do
            {
                if(x == y/2)
                {
                    x++;
                    continue;// skips the remaining statement
                }
                System.out.println(x);
                x++;
            }while(x <= y);
        }
    }
```

**Switch statement**

- ✓ The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement.
- ✓ In other words, the switch statement tests the equality of a variable against multiple values.
- ✓ The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow variables.
- ✓ The case value can have a *default label* which is optional.
- ✓ The case values must be *unique*. In case of duplicate value, it renders compile-time error.

**Syntax**

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......
default:
 code to be executed if all cases are not matched;
}
```

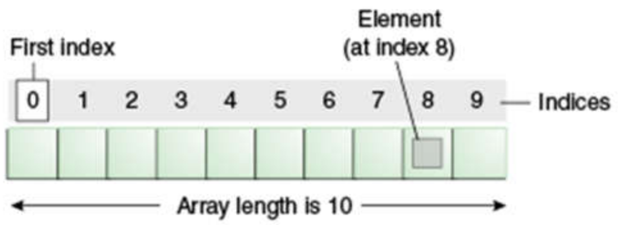
**Example:**

```
public class SwitchExample
{
    public static void main(String[] args)
    {
        //Declaring a variable for switch expression
        int number=20;
        //Switch expression
        switch(number)
        {
```

```
                    //Case statements
                            case 10: System.out.println("10");
                                    break;
                            case 20: System.out.println("20");
                                    break;
                            case 30: System.out.println("30");
                                    break;
                            //Default case statement
                            default:System.out.println("Not in 10, 20 or 30");
                    }
            }
    }
```

**Arrays**

- An array is a collection of similar type of elements which has contiguous memory location.
- The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements.
- 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**

There are two types of array.

1. **1.** Single Dimensional Array
2. **2.** Multidimensional Array

1. ***Single Dimensional Array in Java***

**Syntax**

***dataType[] arr; (or)***
***dataType []arr; (or)***
***dataType arr[];***

Instantiation of an Array in Java

***arrayRefVar=new datatype[size];***

**Example**: ***Java Array***

*//Java Program to illustrate how to declare, instantiate, initialize and traverse the Java array.*

```
class Testarray
{
    public static void main(String args[])
    {
            int a[]=new int[5];          //declaration and instantiation
            a[0]=10;   a[1]=20;  ⎤
            a[2]=70;   a[3]=40;  ⎬ //initialization
            a[4]=50;             ⎦
            //traversing array
            for(int i=0;i<a.length;i++)//length is the property of array
                System.out.println(a[i]);
    }
 }
```

***Declaration, Instantiation and Initialization of Java Array***

We can declare, instantiate and initialize the java array together by:

***int a[]={33,3,4,5}***;//declaration, instantiation and initialization

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1
{
        public static void main(String args[])
        {
                int a[]={33,3,4,5};
        //declaration, instantiation and initialization
        //printing array
                for(int i=0;i<a.length;i++)
        //length is the property of array
                System.out.println(a[i]);
        }
}
```

2. ***Multidimensional Array in Java***

   In such case, data is stored in row and column based index (also known as matrix form).

   Syntax to Declare Multidimensional Array in Java

   ***dataType[][] arrayRefVar; (or)***
   ***dataType [][]arrayRefVar; (or)***
   ***dataType arrayRefVar[][]; (or)***
   ***dataType []arrayRefVar[];***

   - Example to instantiate Multidimensional Array in Java

     int[][] arr=new int[3][3];//3 row and 3 column

   - Example to initialize Multidimensional Array in Java

     arr[0][0]=1;
     arr[0][1]=2;
     arr[1][0]=3;

```
        arr[1][1]=4;
        arr[2][0]=5;
        arr[2][1]=6;
```

**Example**

*//Java Program to illustrate the use of multidimensional array*

```
class Testarray3
{
        public static void main(String args[])
        {
                //declaring and initializing 2D array
                int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
                //printing 2D array
                for(int i=0;i<3;i++)
                {
                        for(int j=0;j<3;j++)
                        {
                                System.out.print(arr[i][j]+" ");
                        }
                        System.out.println();
                }
        }
}
```

**Strings**

- In Java, string is basically an object that represents sequence of char values. \
- An array of characters works same as Java string.
  For example:
  ```
          char[] ch={'j','a','v','a','s','t','r','i','n','g'};
          String s=new String(ch);
      is same as:
          String s="javastring";
  ```
- Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
- The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.

**Java String class methods**

✓ The java.lang.String class provides many useful methods to perform operations on sequence of char values.

| No. | Method | Description |
|---|---|---|
| 1 | char charAt(int index) | returns char value for the particular index |
| 2 | int length() | returns string length |
| 3 | static String format(String format, Object... args) | returns a formatted string. |
| 4 | static String format(Locale l, String format, Object... args) | returns formatted string with given locale. |
| 5 | String substring(int beginIndex) | returns substring for given begin index. |

| | | |
|---|---|---|
| 6 | String substring(int beginIndex, int endIndex) | returns substring for given begin index and end index. |
| 7 | boolean contains(CharSequence s) | returns true or false after matching the sequence of char value. |
| 8 | boolean equals(Object another) | checks the equality of string with the given object. |
| 9 | boolean isEmpty() | checks if string is empty. |
| 10 | String concat(String str) | concatenates the specified string. |
| 11 | int indexOf(int ch) | returns the specified char value index. |
| 12 | int indexOf(int ch, int fromIndex) | returns the specified char value index starting with given index. |
| 13 | int indexOf(String substring) | returns the specified substring index. |
| 14 | int indexOf(String substring, int fromIndex) | returns the specified substring index starting with given index. |
| 15 | String toLowerCase() | returns a string in lowercase. |
| 16 | String toLowerCase(Locale l) | returns a string in lowercase using specified locale. |
| 17 | String toUpperCase() | returns a string in uppercase. |
| 18 | String toUpperCase(Locale l) | returns a string in uppercase using specified locale. |
| 19 | String trim() | removes beginning and ending spaces of this string. |
| 20 | static String valueOf(int value) | converts given type into string. It is an overloaded method. |

**Java String valueOf()**

✓ The java string valueOf() method converts different types of values into string.

✓ By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

Internal implementation

```
public static String valueOf(Object obj)
{
    return (obj == null) ? "null" : obj.toString();
}
```

Signature

- The signature or syntax of string valueOf() method is given below:

*public static String valueOf(boolean b)*
*public static String valueOf(char c)*
*public static String valueOf(char[] c)*
*public static String valueOf(int i)*
*public static String valueOf(long l)*
*public static String valueOf(float f)*
*public static String valueOf(double d)*
*public static String valueOf(Object o)*

Returns

string representation of given value

**Example**

```
public class StringValueOfExample
{
        public static void main(String args[])
        {
                int value=30;
                String s1=String.valueOf(value);
                System.out.println(s1+10);//concatenating string with 10
        }
}
```

**Java String equals()**

- ✓ The java string equals() method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.
- ✓ The String equals() method overrides the equals() method of Object class.

**Example**

```
public class EqualsExample
{
        public static void main(String args[])
        {
                String s1="javatpoint";
                String s2="javatpoint";
                String s3="JAVATPOINT";
                String s4="python";
                System.out.println(s1.equals(s2));//true because content & case is same
                System.out.println(s1.equals(s3));//false because case is not same
                System.out.println(s1.equals(s4));//false because content is not same
        }
}
```

## UNIT II

**Java as an OOP Language**

- Java program consists of a set of objects that interact and communicate with each other.
- Classes are the blueprints or construction plans for these objects and specify the properties / state and behaviour of objects.

**DEFINING CLASSES**

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.

**Objects and Classes**

- An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.
- The characteristics of an object are:
  - ✓ **State:** represents the data (value) of an object.
  - ✓ **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.

**Example:**

Pen is an object. Its state include
name is Reynolds;
color is white.
Its behavior is writing.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

**Object Definitions:**

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

***How to create an object?***

**Classname Object name = new classname();**
Rectangle r1=**new** Rectangle() // Rectangle class name and r1 object name

***Multiple objects*** can be created as

Rectangle r1=**new** Rectangle(), r2=**new** Rectangle(); //creating two objects

**What is a class in Java?**

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

***Syntax to declare a class:***

```
class <class_name>
{
    field;
    method;
}
```

**Method in Java**

In Java, a method is like a function which is used to expose the behavior of an object.

**Advantage of Method**

- Code Reusability
- Code Optimization

**new keyword in Java**

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

**Object and Class**

**Example1:**

In this example, we have created a Student class which has two data members, id and name. We are creating the object of the Student class by new keyword and printing the object's value.

```
class Student
{
        //defining fields
        int id;             //field or data member or instance variable
        String name;
        //creating main method inside the Student class
        public static void main(String args[])
        {
                //Creating an object or instance
                Student s1=new Student();     //creating an object of Student
                //Printing values of the object
accessing member through reference variable
                System.out.println(s1.id);
                System.out.println(s1.name);
        }
}
```

**Object and Class**

**Example2:**

```
class Student
{
        int id;
        String name;
}
class TestStudent1
{
        public static void main(String args[])
        {
                Student s1=new Student();
                System.out.println(s1.id);
                System.out.println(s1.name);
        }
}
```

**Object and Class**
**Example3:**

```
// Initializing an object means storing data into the object.
// Initializing an object through a reference variable
class Student
{
        int id;
        String name;
}
class TestStudent2
{
        public static void main(String args[])
        {
                Student s1=new Student();
                s1.id=101;
                s1.name="Sonoo";
                System.out.println(s1.id+" "+s1.name);
                //printing members with a white space
        }
}
```

**Object and Class**
**Example4:**

```
// creating multiple objects
class Student
{
        int id;
        String name;
}
class TestStudent3
{
        public static void main(String args[])
        {
                //Creating objects
                Student s1=new Student();
                Student s2=new Student();
                //Initializing objects
                s1.id=101;
                s1.name="Sonoo";
                s2.id=102;
                s2.name="Amit";
                //Printing data
                System.out.println(s1.id+" "+s1.name);
                System.out.println(s2.id+" "+s2.name);
        }
}
```

**Object and Class**
**Example5:**

```
// Passing Arguments to Methods
class Student
{
        int rollno;
        String name;
        void insertRecord(int r, String n)
        {
                rollno=r;
                name=n;
        }
        void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4
{
        public static void main(String args[])
        {
                Student s1=new Student();
                Student s2=new Student();
                s1.insertRecord(111,"Karan");
                s2.insertRecord(222,"Aryan");
                s1.displayInformation();
                s2.displayInformation();
        }
}
```

**Variable Types in Java**
- ➢ **Variable** is name of *reserved area allocated in memory*.
- ➢ In other words, it is a *name of memory location*. It is a combination of "vary + able" that means its value can be changed.

**Types of Variables:**
There are three types of variables:
1. Local Variables
2. Instance Variables
3. Static or Class variables

**1. Local Variables**
- ✓ A variable declared inside the body of the method is called local variable.
- ✓ This variable can use only within that method and the other methods in the class aren't even aware that the variable exists.

**2. Instance Variable**
- ✓ A variable declared inside the class but outside the body of the method, is called instance variable.

- ✓ Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created.

**3. Static or Class variables**

- ✓ A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class.
- ✓ Memory allocation for static variable happens only once when the class is loaded in the memory.

**Knowing this Keyword**

***this*** is a **reference variable** that refers to the current object.

Usage of ***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.

**Example**

```
class Student
{

        int rno;//instance variable
        String name;
        static String college = "Jamal Mohamed College";
        Student(int rno, String y)      //local variable
        {
                rno=rno;\\local variable
                this.rno=rno;\\instance
                name=y;
        }
        void display()
        {
                System.out.println("Roll No:"+ rno+ "Name:" +name+ "College:"
+college);
        }
}
class thisdemo
{
        public static void main(String args[])
        {
                Student s1 = new Student(5001, "SAQ");
                Student s2 = new Student(5002, "Asif");

                s1.display();
                s2.display();
        }
}
```

**Variable Scope and Method Definitions**

Scope of a variable is the part of the program where the variable is accessible. Scope of a variable can determined at compile time and independent of function call stack. Java programs are organized in the form of classes. Every class is part of some package.

**Member Variables (Class Level Scope)**

Java scope rules can be covered under following categories.

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't affect scope of them within a class.
- Member variables can be accessed outside a class with following rules

| Modifier | Package | Subclass | World |
|---|---|---|---|
| public | Yes | Yes | Yes |
| protected | Yes | Yes | No |
| Default (nomodifier) | Yes | No | No |
| private | No | No | No |

Some Important Points about Variable scope in Java:

1. In general, a set of curly brackets { } defines a scope.
2. In Java we can usually access a variable as long as it was defined within the same set of brackets as the code we are writing or within any curly brackets inside of the curly brackets where the variable was defined.
3. Any variable defined in a class outside of any method can be used by all member methods.
4. When a method has the same local variable as a member, "this" keyword can be used to reference the current class variable.
5. For a variable to be read after the termination of a loop, It must be declared before the body of the loop.

**Passing arguments to methods**

➢ There are mainly two ways of passing arguments to methods:
   1. Pass by value
   2. Pass by reference

➢ Java directly supports passing by value; however, passing by reference will be accessible through reference objects.

***1. Pass by value:***

✓ When the arguments are passed using the pass by value, only a copy of the variables are passed which has the scope within the method which receives the copy of these variables.

✓ Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as **call by value.**

**Example**

```
class CallByValue
{
        public static void Example(int x, int y)
        {
                x++;
                y++;
        }
}
public class Main
{
        public static void main(String[] args)
        {
                int a = 10;
                int b = 20;
                CallByValue object = new CallByValue();
                System.out.println("Value of a: " + a + " & b: " + b);
                // Passing variables in the class function
                object.Example(a, b);
                // displaying values after calling the function
                System.out.println("Value of a: "+ a + " & b: " + b);
        }
}
```

**OUTPUT:**

```
Value of a: 10 & b: 20
Value of a: 10 & b: 20
```

2. ***Pass by reference:***

- ✓ When parameters are passed to the methods, the calling method returns the changed value of the variables to the called method.
- ✓ Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as **call by reference.**
- ✓ This method is efficient in both time and space.

**Example**

```
class CallByReference
{
        int a, b;
        CallByReference(int x, int y)
        {
                a = x;          b = y;
        }
        void ChangeValue(CallByReference obj)
        {
                obj.a += 10;    obj.b += 20;
        }
}
public class Main
{
        public static void main(String[] args)
        {
                CallByReference object = new CallByReference(10, 20);
                System.out.println("Value of a: "+ object.a + " & b: "+ object.b);
                object.ChangeValue(object);
                System.out.println("Value of a: "+ object.a + " & b: "+ object.b);
        }
}
```

**OUTPUT:**

```
Value of a: 10 & b: 20
Value of a: 20 & b: 40
```

**Method Overloading**

- The process of defining methods with same name but with different tasks is termed method overloading.
- Java differentiates overloaded methods based on the number and type of parameters and not on the return type of the method.
- A compiler error would occur when two methods with the same name and same parameter list but different return types are created.

**Example**

```
class Student
{
        int rno;
        String name;
        static String college = "Jamal Mohamed College";
        Student()
        {
                System.out.println("RollNo:"+rno+"\nName:"+name);
                System.out.println("College:"+college);
        }
        Student(int x, String y)
        {
                rno=x;
                name=y;
        }
        void display()
        {
                System.out.println("RollNo:"+rno+"\nName:"+name);
                System.out.println("College:"+college);
        }
        void display(int a)
        {
                System.out.println("The value of a is:"+a);
        }
}
class OverloadDemo
{
        public static void main(String args[])
        {
                Student s1 = new Student(5001, "SAQ");
                Student s2 = new Student(5002, "Asif");
                Student s3 = new Student();
                s1.display();           s2.display();
                s1.display(10);

        }
}
```

**Constructors**

- 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.

1. Types of constructors
    a. Default Constructor
    b. Parameterized Constructor
2. Constructor Overloading
3. Does constructor return any value?
4. Copying the values of one object into another
5. Does constructor perform other tasks instead of the initialization

**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

| Java Constructor | Java Method |
|---|---|
| A constructor is used to initialize the state of an object. | A method is used to expose the behaviour of an object. |
| 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. |

**Constructor Example**

```
class Student
{
        int rno;
        String name;
        //String college;
        static final String college = "Jamal Mohamed College";
        Student()
        {
                System.out.println("Roll No:" + rno + "\tName:" + name);
                System.out.println("\tCollege:" + college);
        }
        void display()
        {
                System.out.println("Roll No:" + rno + "\tName:" + name);
                System.out.println("\tCollege:" + college);

        }
```

```
	}
	class ConstructorDemo
	{
		public static void main(String args[])
		{
			Student s1 = new Student();
			Student s2 = new Student();
//			Student s3 = new Student();
			s1.rno = 5001;
			s1.name="SAQ";
//			s2.rno=5002;
//			s2.name="Asif";
			s1.display();
//			s2.display();
//			s3.display();

		}
	}
```

**Parameterized Constructor Example**

```
	class Student
	{

		int rno;
		String name;
		//String college;
		static String college = "Jamal Mohamed College";
	/*	Student(int rno, String name)
		{
			this.rno=rno;
			this.name=name;
		}*/
		Student(int x, String y)
		{
			rno=x;
			name=y;
		}
		void display()
		{
			System.out.println("Roll No:" + rno + "\tName:" + name);
			System.out.println("\tCollege:" + college);
		}
	}
	class PCDemo
	{
		public static void main(String args[])
		{
			Student s1 = new Student(5001, "SAQ");
			Student s2 = new Student(5002, "Asif");
```

```
                s1.display();
                s2.display();
        }
}
```

**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**

```
//Java program to overload constructors
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(111,"Karan");
                Student5 s2 = new Student5(222,"Aryan",25);
                s1.display();
                s2.display();
        }
}
```

**INHERITANCE**

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Moreover, you can add new methods and fields in your current class also.

**Terms used in Inheritance**

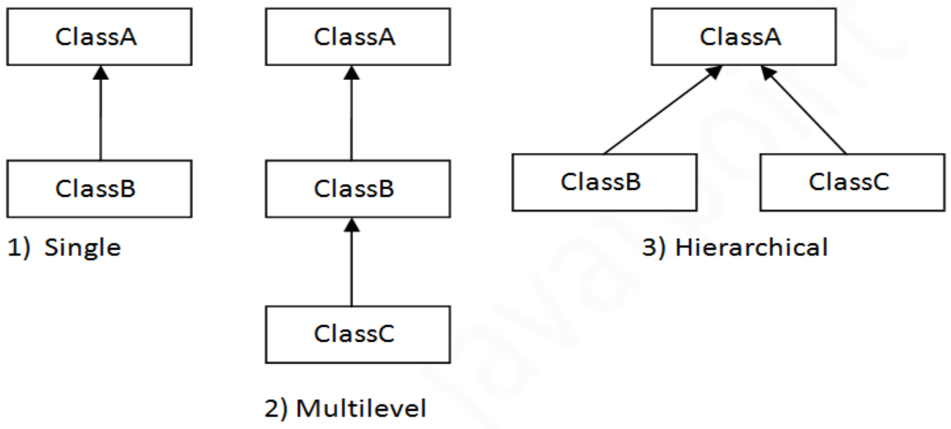
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent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

**Types of inheritance in java**

1. Single
2. Multilevel
3. Hierarchical.



**1. Single Inheritance**

- When a class inherits another class, it is known as a *single inheritance*.

**Example**

```
class Animal
{
        void eat()
```

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

**2. Multilevel Inheritance:**

- When a class extends a class, which extends anther class then this is called **multilevel inheritance**.
- For example class C extends class B and class B extends class A then this type of **inheritance** is known as **multilevel inheritance**.
- When there is a chain of inheritance, it is known as *multilevel inheritance*.
  In the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.
  **Example**

```
        class Animal
        {
                void eat()
                {
                        System.out.println("eating...");
                }
        }
        class Dog extends Animal
        {
                void bark()
                {
                        System.out.println("barking...");
                }
        }
        class BabyDog extends Dog
        {
                void weep()
                {
                        System.out.println("weeping...");
                }
```

```
}
class TestInheritance2
{
        public static void main(String args[])
        {
                BabyDog d=new BabyDog();
                d.weep();       d.bark();
                d.eat();
        }
}
```

**3. Hierarchical Inheritance:**

- Hierarchical inheritance is a kind of inheritance where more than one class is inherited from a single parent or base class.
- When several classes are derived from common base class it is called **hierarchical inheritance**.
- When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
class Animal
{
        void eat()
        {
                System.out.println("eating...");
        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("barking...");
        }
}
class Cat extends Animal
{
        void meow()
        {
                System.out.println("meowing...");
        }
}
class TestInheritance3
{
        public static void main(String args[])
        {
                Cat c=new Cat();
                c.meow();
                c.eat();
        }
}
```

**Method Overriding in Java**

- If subclass (child class) has the same method as declared in the parent class, it is known as ***method overriding*** in Java.
- In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as ***method overriding***.

**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).

**Example1**

```
class Vehicle
{
        void run()
        {
                System.out.println("Vehicle is running");
        }
}
//Creating a child class
class Bike extends Vehicle
{
        public static void main(String args[])
        {
                //creating an instance of child class
                Bike obj = new Bike();
                //calling the method with child class instance
                obj.run();
        }
}

//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle
{
        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
		}
	}
```

**Example2**

```
	class Bank
	{
		int getRateOfInterest()
		{
			return 0;
		}
	}
	//Creating child classes.
	class SBI extends Bank
	{
		int getRateOfInterest()
		{
			return 8;
		}
	}
	class ICICI extends Bank
	{
		int getRateOfInterest()
		{
			return 7;
		}
	}
	class AXIS extends Bank
	{
		int getRateOfInterest()
		{
			return 9;
		}
	}

	//Test class to create objects and call the methods
	class Test2
	{
		public static void main(String args[])
		{
			SBI s=new SBI();
			ICICI i=new ICICI();
			AXIS a=new AXIS();
		System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
		System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
```

```
                System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
                }
        }
```

Difference between method overloading and method overriding in java

| No. | Method Overloading | Method Overriding |
|---|---|---|
| 1 | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2 | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3 | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4 | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5 | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

## MODIFIERS

There are two types of modifiers in Java:

1. ***Access Modifiers*** - controls the access level
2. ***Non-Access Modifiers*** - do not control access level, but provides other functionality

### 1. Access Modifiers

There are four types of Java access modifiers:

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

**The four access levels are −**

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Let's understand the access modifiers in Java by a simple table.

| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
|---|---|---|---|---|
| **Private** | Yes | No | No | No |
| **Default** | Yes | Yes | No | No |
| **Protected** | Yes | Yes | Yes | No |
| **Public** | Yes | Yes | Yes | Yes |

**1. Private**

- The private access modifier is accessible only within the class.
- **Simple example of private access modifier**
- In this example, we have created two classes A and Simple.
- A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
        }
}

public class Simple
{
        public static void main(String args[])
        {
                A obj=new A();
                System.out.println(obj.data);  //Compile Time Error
                obj.msg();                          //Compile Time Error
        }
}
```

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A
{
        private A()
        {}                                          //private constructor
        void msg()
        {
                System.out.println("Hello java");
        }
}
public class Simple
{
        public static void main(String args[])
        {
                A obj=new A();                      //Compile Time Error
        }
}
```

**2) Default**

- If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package.
- It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
        void msg()
        {
                System.out.println("Hello");
        }
}

//save by B.java
package mypack;
import pack.*;
class B
{
        public static void main(String args[])
        {
                A obj = new A();                //Compile Time Error
                obj.msg();                      //Compile Time Error
        }
}
```

**3) Protected**

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor.
- It can't be applied on the class.
- It provides more accessibility than the default modifer.

  **Example of protected access modifier**

  In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
```

```
public class A
{
        protected void msg()
        {
                System.out.println("Hello");
        }
}

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

**4) Public**

➢ The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.
  Example of public access modifier

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

**Non-Access Modifiers**

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables.

- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

For **classes**, you can use either final or abstract:

| **Modifier** | **Description** |
|---|---|
| final | The class cannot be inherited by other classes |
| abstract | The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. |

For **attributes and methods**, you can use the one of the following:

| **Modifier** | **Description** |
|---|---|
| final | Attributes and methods cannot be overridden/modified |
| static | Attributes and methods belongs to the class, rather than an object |
| abstract | Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). |
| transient | Attributes and methods are skipped when serializing the object containing them |
| synchronized | Methods can only be accessed by one thread at a time |
| volatile | The value of an attribute is not cached thread-locally, and is always read from the "main memory" |

**Finalizing Classes, Methods and Variables**
**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).
**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();
		}
	}
```

**OUTPUT:**
Compile Time Error

**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();
		}
	}
```

**OUTPUT:**
Compile Time Error

**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.
- The 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.

**Program of the counter without static variable**

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.
class Counter
{
	int count=0;//will get memory each time when the instance is created
	Counter()
	{
		count++;//incrementing value
		System.out.println(count);
	}
	public static void main(String args[])
	{
		Counter c1=new Counter();
		Counter c2=new Counter();
		Counter c3=new Counter();
	}
}
```

**Program of counter by static variable**

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
//Java Program to illustrate the use of static variable which is shared with all objects.
class Counter2
{
	static int count=0;
	//will get memory only once and retain its value
	Counter2()
	{
		count++;		//incrementing the value of static variable
		System.out.println(count);
	}
	public static void main(String args[])
	{
		//creating objects
		Counter2 c1=new Counter2();
		Counter2 c2=new Counter2();
		Counter2 c3=new Counter2();
	}
}
```

**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 = "ITS";

        //static method to change the value of static variable
        static void change()
        {
                college = "BBDIT";
        }

        //constructor to initialize the variable
        Student(int r, String n)
        {
                rollno = r;
                name = n;
        }

        //method to display values
        void display()
        {
                System.out.println(rollno+" "+name+" "+college);
        }
}

//Test class to create and display the values of object
public class TestStaticMethod
{
        public static void main(String args[])
        {
                Student.change();//calling change method
                Student s1 = new Student(111,"Karan");
                Student s2 = new Student(222,"Aryan");
                Student s3 = new Student(333,"Sonoo");
                //Calling objects
                s1.display();
                s2.display();
                s3.display();
        }
}
```

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method
 class Calculate
{
        static int cube(int x)
        {
                return x*x*x;
        }
        public static void main(String args[])
        {
                int result=Calculate.cube(5);
                System.out.println(result);
        }
}
```

**Restrictions for the static method**
There are two main restrictions for the static method.
They are:

1. The static method cannot use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A
{
        int a=40;       //non static
        public static void main(String args[])
        {
                System.out.println(a);
        }
}
```

**3) Java static block**

✓ Is used to initialize the static data member.
✓ It is executed before the main method at the time of class loading.

**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");
        }
}
```

**Abstract class in Java**

➢ Abstract classes are classes whose sole purpose is to provide common information for sub-classes. They can have no instances i.e we cannot create object.
➢ Abstract methods are methods with signatures, but no implementation. The sub-classes of the class that contains that abstract method must provide its actual implementation.
➢ 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.

**Rules for Abstract class**

- An abstract class must be declared with an **abstract** keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

```
//Example of an abstract class that has abstract and non-abstract methods
 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 TestAbstraction
{
     public static void main(String args[])
     {
          Bike obj = new Honda();
          obj.run();
          obj.changeGear();
      }
}
```

**OUTPUT**

```
bike is created
running safely..
gear changed
```

**PACKAGES**

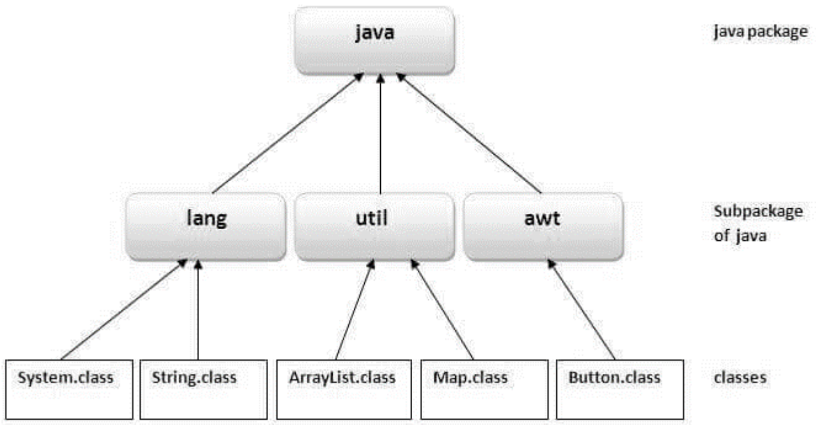
- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

  **Advantage of Java Package**

  1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
  2. Java package provides access protection.
  3. Java package removes naming collision.

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple
{
        public static void main(String args[])
        {
                System.out.println("Welcome to package");
        }
}
```



**Access package from another package**

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

**1.** Using **packagename.***

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

  **Example**

```
//save by A.java
package pack;
```

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

**2.** Using **packagename.classname**

- If you import package.classname then only declared class of this package will be accessible.

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

**3)** Using **fully qualified name**

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

```
//save by A.java
        package pack;
        public class A
        {
                public void msg()
                {
                        System.out.println("Hello");
                }
        }
        //save by B.java
        package mypack;
        class B
        {
                public static void main(String args[])
                {
                        pack.A obj = new pack.A();//using fully qualified name
                        obj.msg();
                }
        }
```

**Package Class**

The package class provides methods to get information about the specification and implementation of a package. It provides methods such as getName(), getImplementationTitle(), getImplementationVendor(), getImplementationVersion() etc.

**Example**

```
class PackageInfo
{
    public static void main(String args[])
    {
        Package p=Package.getPackage("java.lang");
        System.out.println("package name: "+p.getName());
        System.out.println("Specification Title: "+p.getSpecificationTitle());
        System.out.println("Specification Vendor: "+p.getSpecificationVendor());
        System.out.println("Specification Version: "+p.getSpecificationVersion());
        System.out.println("Implementaion Title: "+p.getImplementationTitle());
        System.out.println("Implementation Vendor: "+p.getImplementationVendor());
        System.out.println("Implementation Version: "+p.getImplementationVersion());
        System.out.println("Is sealed: "+p.isSealed());  0
    }
}
```

**OUTPUT:**

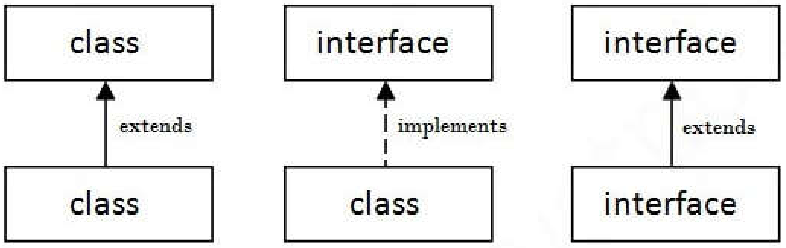
```
        package name: java.lang
        Specification Title: Java Plateform API Specification
        Specification Vendor: Sun Microsystems, Inc.
        Specification Version: 1.6
        Implemenation Title: Java Runtime Environment
        Implemenation Vendor: Sun Microsystems, Inc.
        Implemenation Version: 1.6.0_30
        IS sealed: false
```

## INTERFACES

- Interfaces, like abstract classes and methods, provide templates of behaviour that other classes are expected to implement.
- An ***interface*** 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 inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- It cannot be instantiated just like the abstract class.

***Interfaces and Classes***

- An interface in Java is a blueprint of a class. It has final static constants and abstract methods.
- A class extends another class, an interface extends another interface, but a class implements an interface.



**Creating an interface**

- To create an interface, the keyword **interface** should be used.
- 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.

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

The methods within the interface can carry modifiers such as **public** and **abstract.**

```
public interface ExampleInterface1
{
        public static final int triple = 3;
        int twice = 2;
        public abstract void method1();      //explicitly public and abstract
        void method2();                      //effectively public and abstract
}
```

**Implementing Interfaces**

- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

**Example**

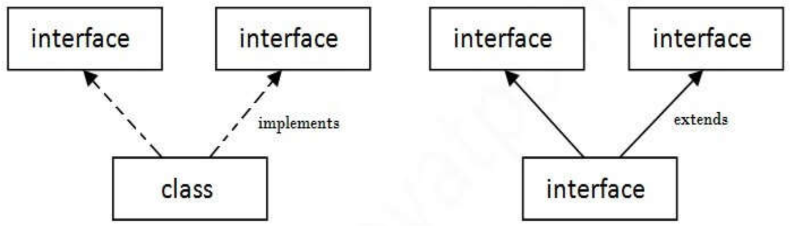
```
interface printable
{
                void print();
}
class A6 implements printable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public static void main(String args[])
        {
                A6 obj = new A6();
                obj.print();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

**Example 2**

```
interface Drawable
{
        void draw();
}

//Implementation: by second user
class Rectangle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing rectangle");
        }
}
class Circle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing circle");
        }
}

//Using interface: by third user
class TestInterface1
{
```

```
            public static void main(String args[])
            {
                    Drawable d=new Circle();
                    d.draw();
            }
    }
```

**Multiple inheritance in Java by interface**
If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



**Multiple Inheritance in Java**

**Example**

```
interface Printable
{
            void print();
}
interface Showable
{
            void show();
}
class 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();
            }
}
```

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
interface Printable
{
        void print();
}
interface Showable
{
        void print();
}

class TestInterface3 implements Printable, Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public static void main(String args[])
        {
                TestInterface3 obj = new TestInterface3();
                obj.print();
         }
}
```

**Extending Interfaces**

- An interface can extend another interface in the same way that a class can extend another class.
- The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

***Interface inheritance***

A class implements an interface, but one interface extends another interface.

```
interface Printable
{
        void print();
}
interface Showable extends Printable
{
        void show();
}
class TestInterface4 implements Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
```

```
	public void show()
	{
		System.out.println("Welcome");
	}

	public static void main(String args[])
	{
		TestInterface4 obj = new TestInterface4();
		obj.print();
		obj.show();
	}
}
```

## UNIT III

**Exception Handling**

- An **Exception** is an abnormal condition occurring during the execution of a program. An exception is an event that may cause abnormal termination of the program during its execution.
- Different types of errors like user errors, logic errors or system errors can cause exception.
  Example: Division by zero, out of array bounds, running out of virtual memory, opening an invalid file.
- **Exception Handling** is a mechanism that enables programs to detect and handle errors before they occur.
  **Example:**

```
class ExceptionDemo
{
    public static void main(String[] args)
    {
            int a, b, c;
            System.out.println("Program Begins...");

            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            c = a / b;
            System.out.println (a + "/" + b + "=" + c);
            System.out.println("Program Ends...");
    }
}
```

**Basics of Exception Handling**

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

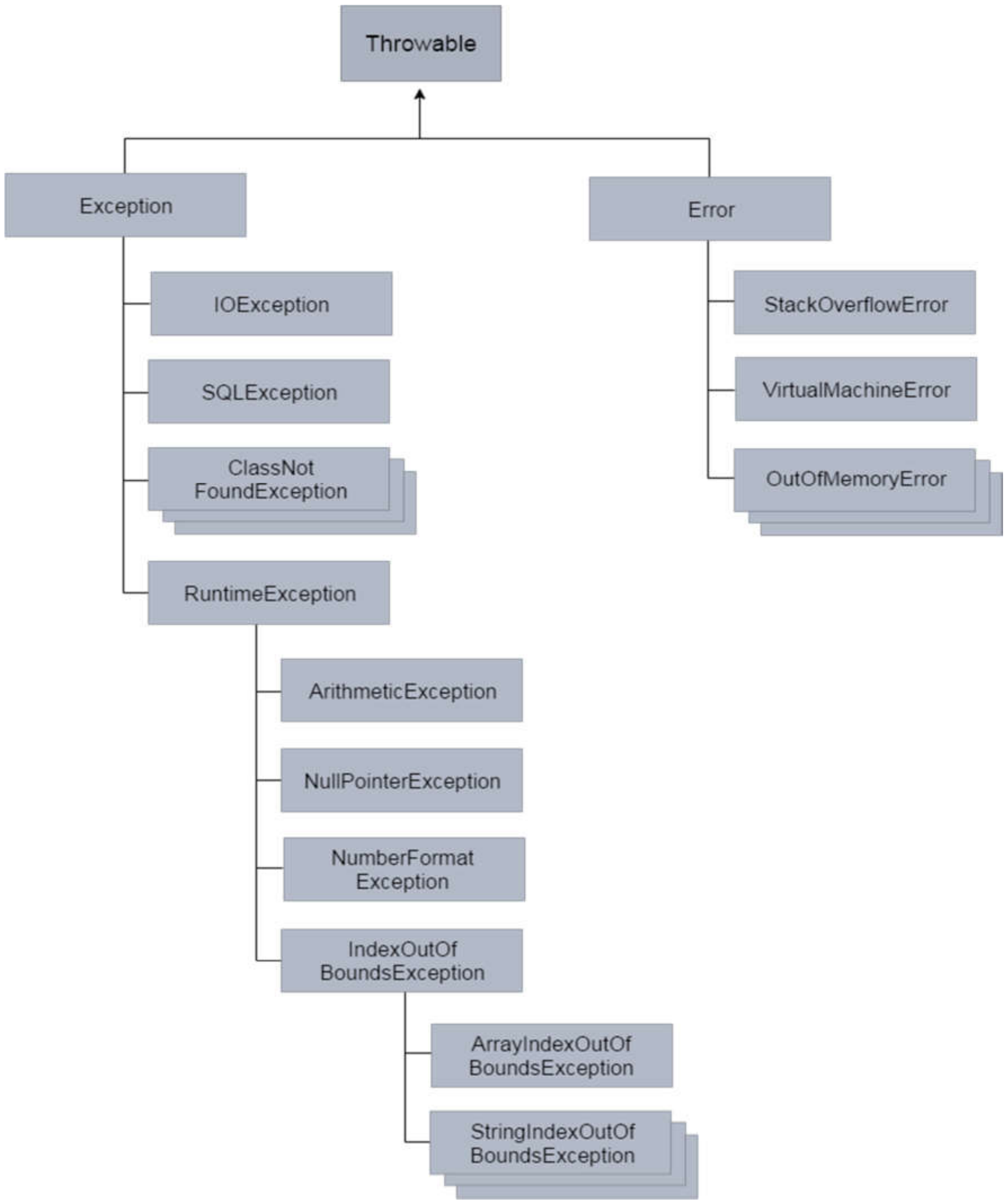
| Keyword | Description |
|---|---|
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded 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. |

The three important features that an exception usually carries are

1. The type of exception – determined by the class of the exception object
2. Where the exception occur – the stack trace
3. Context information – the error message and other state information

**Exception Hierarchy**

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



**Constructors and Methods in Throwable Class**

There are four constructors in the Throwable class:

| S.No. | Constructor | Description |
|---|---|---|
| 1 | Throwable() | This constructs a new throwable with null as its detail message. |
| 2 | Throwable(String message) | This constructs a new throwable with the specified detail message. |
| 3 | Throwable(String message, Throwable cause) | This constructs a new throwable with the specified detail message and cause. |
| 4 | Throwable(Throwable cause) | This constructs a new throwable with the specified cause and a detail message of (cause==null ? null : cause.toString()) (which typically contains the class and detail message of cause). |

**Methods in Throwable Classes**

The following is the list of three useful methods that provide information about an exception

| S.No. | Methods | Description |
|---|---|---|
| 1 | getMessage() | Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | toString() | Returns the name of the class concatenated with the result of getMessage(). |
| 3 | printStackTrace() | Prints the result of toString() along with the stack trace to System.err, the error output stream. |

The methods of Throwable class that deal with StackTrace are the following

| S.No. | Methods | Description |
|---|---|---|
| 1 | fillInStackTrace() | Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace. |
| 2 | getStackTrace() | Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. |
| 3 | printStackTrace() | Prints the result of toString() along with the stack trace to System.err, the error output stream. |
| 4 | setStackTrace (StackTraceElement [] stackTrace) | |

**Unchecked and Checked Exception**

**Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked.

1. Checked Exception
2. Unchecked Exception

**1) Checked Exception**

➢ The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions. Checked exceptions are checked at compile-time.

- These exceptions cannot be ignored.
  e.g. IOException, SQLException etc.

**2) Unchecked Exception**

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

**Handling Exceptions in Java**

- Exception handling in java is accomplished by using five keywords:
  1. try,
  2. catch,
  3. throw,
  4. throws and
  5. finally
- In java the code that may generate an exception is enclosed in a try block.
- The try block can be followed immediately by one or more catch blocks with a finally block as the last block. The try block can also end without a catch block, and the catch blocks need not always have finally as the last block. If there are no catch blocks following try block, the finally block is required.
- If the exception type thrown matches the parameter type in one of the catch blocks, the code for that catch block is executed.

**1. 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 try-catch block**

```
try
{
        //code that may throw an exception
}
catch(Exception_class_Name ref)
{
}
```

**Syntax of try-finally block**

```
try
{
        //code that may throw an exception
}
finally
{
}
```

**2. Catch Block**

- When an exception occurs in a try block, program control is transferred to the appropriate catch block. The catch block is specified by the keyword **catch** followed by a single argument within parenthesis **()**.
- The catch block must be used after the try block only.

**Example:**

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
                int a, b, c;
                System.out.println("Program Begins...");
                try
                {
                        a = Integer.parseInt(args[0]);
                        b = Integer.parseInt(args[1]);
                        c = a / b;
                        System.out.println (a + "/" + b + "=" + c);
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
                System.out.println("Program Ends...");
        }
}
```

**OUTPUT**

```
java ExceptionDemo 4 2
Program Begins...
4/2 = 2
Program Ends...
```

```
java ExceptionDemo a 3
Program Begins...
java.lang.NumberFormatException: a
Program Ends...
```

```
java ExceptionDemo 4 0
Program Begins...
java.lang.ArithmeticException: / by zero
Program Ends...
```

```
java ExceptionDemo 4
Program Begins...
java.lang.ArrayIndexOutOfBoundsException
Program Ends...
```

**3. Finally Block**

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.
- **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 ExceptionDemo
{
        public static void main(String[] args)
        {
                int a, b, c;
                System.out.println("Program Begins...");
                try
                {
                        a = Integer.parseInt(args[0]);
                        b = Integer.parseInt(args[1]);
                        c = a / b;
                        System.out.println (a + "/" + b + "=" + c);
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
                finally
                {
                        System.out.println("Finally blocks always get executed");
                }
                System.out.println("Program Ends...");
        }
}
```

**OUTPUT**

```
java ExceptionDemo 4 2
Program Begins...
4/2 = 2
Finally blocks always get executed
Program Ends...
java ExceptionDemo 4 0
Program Begins...
java.lang.ArithmeticException: / by zero
Finally blocks always get executed
Program Ends...
```

**4.** The Keyword **throw**

- An exception can be caught only if it is identified, or, in other words, thrown.
- Exceptions can be thrown by the Java run-time system for the code that has been written. This can be achieved also by using the **throw** statement explicitly.
- This statement starts with the keyword **throw** followed by a single argument.

  **Syntax of throw statement**

  **throw <Exception object>**
- In a specific case where an instance of 'Exception object' is to be thrown, it takes the following form:

  **throw new <Exception object>**

Usually the above statement is used to pass a string argument along with the exceptional object, so that the string can be displayed when the exception is handled.

**Example:**

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
                int number1=15, number2=10;
                if(number1 > number2) //Conditional statement.
                        throw new Exception("number1 is 15");
                else
                        System.out.println("number2 is 15");
        }
}
```

**OUTPUT**

```
Exception in thread "main" java.Exception: number1 is 15 as TestThrow.main
(TestThrow.java:8)
```

**Multiple Catch Blocks**

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. 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.

**Example:**

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
                int a, b, c;
                System.out.println("Program Begins...");
                try
                {
                        a = Integer.parseInt(args[0]);
                        b = Integer.parseInt(args[1]);
                        c = a / b;
                        System.out.println (a + "/" + b + "=" + c);
                }
                catch(NumberFormatException e)
                {
                        System.out.println("Arguments passed should be valid Numbers");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Second Argument should not be Zero");
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Pass Proper Arguments");
                }
                System.out.println("Program Ends...");
        }
}
```

**Nested Try Statements**

- The try block within a try block is known as nested try block in java.
- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

**Example:**

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
                int a, b, c;
                System.out.println("Program Begins...");
                try
                {
                        a = Integer.parseInt(args[0]);
                        b = Integer.parseInt(args[1]);
                        try
                        {
                                c = a / b;
                                System.out.println (a + "/" + b + "=" + c);
                        }
                        catch(ArithmeticException e)
                        {
                                System.out.println("Second Argument should not be Zero");
                        }
                }
                catch(NumberFormatException e)
                {
                        System.out.println("Arguments passed should be valid Numbers");
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Pass Proper Arguments");
                }
                System.out.println("Program Ends...");
        }
}
```

**Exception and Inheritance**

- While using multiple catch statements, it is important to be aware of the order of exception classes and arrange them correctly.
- If a super-class exception is coded first in the catch block then, when an exception occurs, the super-class exception gets executed whereas the sub-class exception catch block never gets executed.

**Incorrect ordering of catch blocks**

```
class ExceptionDemo
{
	public static void main(String[] args)
	{
		int a, b, c;
		System.out.println("Program Begins...");
		try
		{
			a = Integer.parseInt(args[0]);
			b = Integer.parseInt(args[1]);
			c = a / b;
			System.out.println (a + "/" + b + "=" + c);
		}
		catch(ArithmeticException e)
		{
			System.out.println("Second Argument should not be Zero");
		}
		catch(Exception e)
		{
			System.out.println("Exception caught"+e);
		}
	}
}
```

**Correct ordering of catch blocks**

```
class ExceptionDemo
{
	public static void main(String[] args)
	{
		int a, b, c;
		System.out.println("Program Begins...");
		try
		{
			a = Integer.parseInt(args[0]);
			b = Integer.parseInt(args[1]);
			c = a / b;
			System.out.println (a + "/" + b + "=" + c);
		}
		catch(Exception e)
		{
			System.out.println("Exception caught"+e);
		}
		catch(ArithmeticException e)
		{
			System.out.println("Second Argument should not be Zero");
		}
	}
}
```

**Throwing User-defined Exceptions**
User-defined exceptions are useful to handle business applicaton-specific error conditions. When creating user-defined exceptions, the name of the exception type should not be a reserved exception type name.
User-defined exceptions are provided by an application provider or a Java API provider. Generally, all the exceptions are sub-classes of the class **Throwable.**
User-defined exceptions should be sub-classes of **Exception**.
**Syntax of user-defined exception as an instance**

```
    Throwable UserException = new Throwable();
                        or
    Throwable UserException = new Throwable("This is user exception message");
```

**Syntax of user-defined exception using extends keyword**

```
    class UserException extends Exception
    {
        public UserException()
        {
            .....
        }
        public UserException(String str)
        {
            .....
        }
        .....
    }
```

**Example**

```
  class UserException extends Exception
  {
        public UserException()
        {
              super(s);
              System.out.println("From User Exception Constructor");
        }
  }
  class ExceptionDemo
  {
        public static void main(String[] args)
        {
              int a, b, c;
              System.out.println("Program Begins...");
              try
              {
                    a = Integer.parseInt(args[0]);
                    b = Integer.parseInt(args[1]);
                    if(b==0)
```

```
                        {
                        throw new UserException(“Second Argument should not be Zero”);
                        }
                        c = a / b;
                        System.out.println (a + "/" + b + "=" + c);
                }
                catch(UserException e)
                {
                        System.out.println(“From UserException catch Statement”);
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }

        }
}
```

**OUTPUT**

```
java ExceptionDemo 4 0
Program Begins...
From User Exception Constructor
From UserException catch Statement
UserException: Second Argument should not be Zero
Program Ends...
```

**Redirecting and Rethrowing Exception**

**1. Redirecting Exception using *throws***

- Recall that the code capable of throwing an exception is kept in the try block and the exceptions are caught in the catch block.
- When there is no appropriate catch block to handle the (checked) exception that was thrown by an object, the compiler does not compile the program. To overcome this, Java allows the programmer to redirect exceptions that have been raised up the call stack, by using the keyword throws.
- Thus, an exception thrown by a method can be handled either in the method itself or passed to a different method in the call stack.
- To pass exceptions up to the call stack, the method must be declared with a throws clause.

All the exceptions thrown by a method can be declared with a single throws clause; the clause consists of the keyword throws followed by a comma-separated list of all the exceptions, as shown below:

```
void RedirectExMethod( ) throws Exception A, Exception B, Exception C
{
}
```

Program illustrates two methods redirecting an exception Exception from the method throwing it, that is, ConvertAndDivide, to the calling method, main.

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

```
		{
			System.out.println("\n Program Execution starts here\n");
			try
			{
				convertAndDivide ( args[0],args[1]);
			}
			catch(Exception e)
			{
				System.out.println (e.getMessage () +"\n");
				e.printStackTrace ();
			}
			System.out.println("\n Program Execution Completes here");
		}
		static void convertAndDivide (String s 1,String s2) throwsException
		{
			int a, b, c;
			a = Integer.parselnt (s1);
			b = Integer.parselnt (s2);
			c = divide (a, b);
			System.out.println( a + "/" + b + "=" + C );
		}
		static int divide(int x, int y) throws Exception
		{
			if (y==0)
				throw new Exception("Second Argument is Zero ...");
			return x/y;
		}
	}
```

**OUTPUT**

```
Program Execution starts here
Second Argument is Zero …
Program Execution Completes here
1
java Divide 4 a
Program Execution starts here a
java.lang.NumberFormatException: a
at java.lang.lnteger.parse Int (1nteger.java:426)
at java.lang.lnteger.parsel Int (1ntege r.java:476)
at Divide.convertAndDivide (Divide.java:26)
at Divide.main (Divide,java:9)
Program Execution Completes here
```

**Rethrowing an Exception**

- An exception that is caught in the try block can be thrown once again and can be handled. The try block just above the rethrow statement will catch the rethrown object. If there is no try block just above the rethrow statement then the method containing the rethrow statement handles it.

- To propagate an exception, the catch block can choose to rethrow the exception by using the throw statement. Note that there is no special syntax for rethrowing. Program illustrates how an exception can be rethrown.

Program Rethrowing exceptions.

```
public class Divide
{
    public static void main(String[] args)
    {
            int a, b, c;
            try
            {
                    a = Integer.parselnt (args [0]);
                    b = Integer.parselnt (args [l]);
                    try
                    {
                            c = a/b;
                            System.out.println ( a + "I" + b + "=" + c);
                    }
                    catch (ArithmeticException e)
                    {
                            System.out.println ("Second Argument Should not be Zero");
                            System.out.println ("Rethrowing the object again");
                            throw e;
                    }
            }
            catch(ArraylndexOutOfBoundsException e)
            {
                    System.out.println("Arguments passed should be valid Numbers");
                    System.out.println(Pass Proper Arguments");
            }
            System.out.println("\n Program Execution Completes here");
    }
}
```

- Often first-time readers may get confused about the use of the three keywords: throw, throws and Throwable. It is therefore useful to dwell on the three concepts together in order to clarify their meaning.
- Throwable is a class. Though the Throwable class is derived from the java.lang.Object class in the Java class library, Throwable is the super-class of all classes that handle exceptions.
- The keyword throw is a statement that throws an exception. Note that an exception can be thrown either by the throw statement or when an error occurs during the execution of any other statement.

- The keyword throws is a clause specified in the method definition which indicates that the method throws the exceptions mentioned after the keyword throws, which are handled in the called methods.

**Advantages of the Exception-Handling Mechanism**

The main advantages of the exception-handling mechanism in object oriented programming over the traditional error-handling mechanism are the following:

1. ***The separation of error-handling code from normal code:*** Unlike traditional programming languages, there is a clear cut distinction between the normal code and the error-handling code. This separation results in less complex and more readable code. Further it is also more efficient, in the sense that the checking of errors in the normal execution path is not needed, and thus requires CPU cycles.

2. ***A logical grouping of error types:*** Executions can be used to group together errors that are related. This will enable us to handle related exceptions using a single exception handler. When an exception is thrown, an object of one of the exception classes is passed as a parameter. Objects are instances of classes, and classes fall into an inheritance hierarchy in java. This hierarchy can be used to logically group exceptions. Thus, an exception handler can catch exceptions of the class specified by its parameter, or can catch exceptions of any of its sub-classes.

3. ***The ability to propagate errors up the call stack:*** Another important advantage of exception handling in object oriented programming is the ability to propagate errors up the call stack. Exception handling allows contextual information to be captured at the point where the error occurs and to propagate it to a point where it can be effectively handled. This is different from traditional error-handling mechanisms in which the return values are checked and propagated to the calling function.

**Multithreading**

- Java is a multithreaded programming language which means we can develop multithreaded program using Java.
- A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.
- By definition multitasking is when multiple processes share common processing resources such as a CPU.
- Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel.
- The OS divides processing time not only among different applications, but also among each thread within an application.
- Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

**Creating Threads**

**1. Create Thread by Implementing Runnable Interface:**

If your class is intended to be executed as a thread then you can achieve this by implementing Runnable interface.

You will need to follow three basic steps:

***Step 1:***

As a first step you need to implement a run method provided by Runnable interface.

This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run method:

***public void run( )***

***Step 2:***

At second step you will instantiate a Thread object using the following constructor:

***Thread(Runnable threadObj, String threadName);***

- Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

***Step 3***

Once Thread object is created, you can start it by calling start method, which executes a call to run method. Following is simple syntax of start method:

***void start( );***

**Example:**

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable
{
        private Thread t; private String threadName;
        RunnableDemo( String name)
        {
                threadName = name;
                System.out.println("Creating " + threadName );
        }
        public void run()
        {
                System.out.println("Running " + threadName );
                try
                {
                        for(int i = 4; i > 0; i--)
                        {
                                System.out.println("Thread: " + threadName + ", " + i);
                                Thread.sleep(50); // Let the thread sleep for a while.
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("Thread " + threadName + " interrupted.");
                }
                System.out.println("Thread " + threadName + " exiting.");
        }
        public void start ()
        {
                System.out.println("Starting " + threadName );
                if (t == null)
                {
                        t = new Thread (this, threadName);
                        t.start ();
                }
        }
}
public class TestThread
{
        public static void main(String args[])
        {
                RunnableDemo R1 = new RunnableDemo( "Thread-1");
                R1.start();
                RunnableDemo R2 = new RunnableDemo( "Thread-2");
                R2.start();
        }
}
```

**2. Create Thread by Extending Thread Class:**

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

***Step 1***

You will need to override run method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run method:

***public void run( )***

***Step 2***

Once Thread object is created, you can start it by calling start method, which executes a call to run method. Following is simple syntax of start method:

***void start( );***

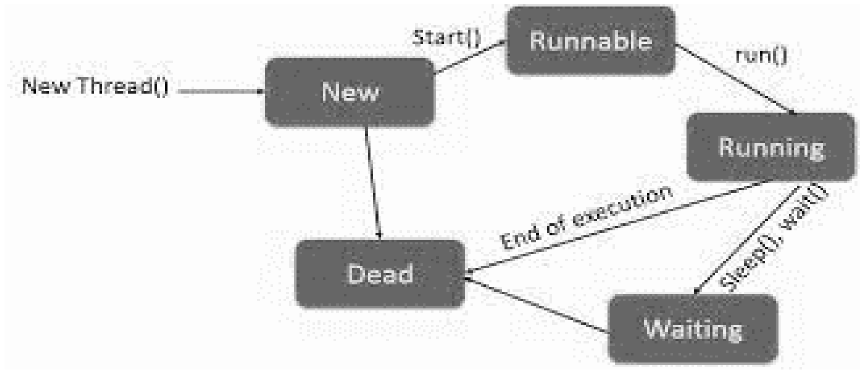
**Example:**

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread
{
        private Thread t;
        private String threadName;
        ThreadDemo( String name)
        {
                threadName = name;
                System.out.println("Creating " + threadName );
        }
        public void run()
        {
                System.out.println("Running " + threadName );
                try
                {
                        for(int i = 4; i > 0; i--)
                        {
                                System.out.println("Thread: " + threadName + ", " + i);
                                // Let the thread sleep for a while.
                                Thread.sleep(50);
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("Thread " + threadName + " interrupted.");
                }
                System.out.println("Thread " + threadName + " exiting.");
        }
        public void start ()
        {
                System.out.println("Starting " + threadName );
                if (t == null)
                {
                        t = new Thread (this, threadName);
                        t.start ();
                }
        }
}
public class TestThread
{
        public static void main(String args[])
        {
                ThreadDemo T1 = new ThreadDemo( "Thread-1");
                T1.start();
                ThreadDemo T2 = new ThreadDemo( "Thread-2");
                T2.start();
        }
}
```

**Thread Life Cycle**

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

**1.** ***New*****:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

**2.** ***Runnable*****:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

**3.** ***Waiting*****:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

**4.** ***Timed waiting*****:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

**5.** ***Terminated Dead*****:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

**Thread Methods:**

Following is the list of important methods available in the Thread class.

| S. No | Methods | Description |
|---|---|---|
| 1 | public void start | Starts the thread in a separate path of execution, then invokes the run method on this Thread object. |
| 2 | public void run | If this Thread object was instantiated using a separate Runnable target, the run method is invoked on that Runnable object. |
| 3 | public final void setNameStringname | Changes the name of the Thread object. There is also a getName method for retrieving the name. |
| 4 | public final void setPriorityintpriority | Sets the priority of this Thread object. The possible values are between 1 and 10. |
| 5 | public final void setDaemonbooleanon | A parameter of true denotes this Thread as a daemon thread. |

|  |  |  |
|---|---|---|
| 6 | public final void joinlongmillisec | The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. |
| 7 | public void interrupt | Interrupts this thread, causing it to continue execution if it was blocked for any reason. |
| 8 | public final boolean isAlive | Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |

**Thread Priorities**

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java thread priorities are in the range between **MIN_PRIORITY** a constant of 1 and **MAX_PRIORITY** a constant of 10. By default, every thread is given priority **NORM_PRIORITY** a constant of 5.
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

**Thread Scheduling**

- Threads run one at a time in such a way as to provide an illusion of concurrency.
- Execution of multiple threads on a single CPU in some order is called ***scheduling***.
- The Java runtime environment supports a very simple, deterministic scheduling algorithm called ***fixed-priority scheduling.***
- This algorithm schedules threads on the basis of their priority relative to other Runnable threads.
- When a thread is created, it inherits its priority from the thread that created it. You also can modify a thread's priority at any time after its creation by using the ***setPriority*** method.
- Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class). The higher the integer, the higher the priority.
- At any given time, when multiple threads are ready to be executed, the runtime system chooses for execution the Runnable thread that has the highest priority. Only when that thread stops, yields, or becomes Not Runnable will a lower-priority thread start executing.
- If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run.

The chosen thread runs until one of the following conditions is true:

1. A higher priority thread becomes runnable.
2. It yields, or its run method exits.
3. On systems that support time-slicing, its time allotment has expired.

Then the second thread is given a chance to run, and so on, until the interpreter exits.

# UNIT IV

## FILES AND I / O STREAMS

- A flow of data is often referred to as a data stream. A stream is an ordered sequence of bytes that has a SOURCE (input stream) or a DESTINATION (output stream).
- A stream is a logical device that represents flow of a sequence of characters.
  A stream can be associated with a file, an Internet resource, to a pipe of a memory buffer.

## Java I/O

- Input and Output is used *to process the input* and *produce the output*.
- A stream is a sequence of data. In Java, a stream is composed of bytes.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- In general, streams are classified into two types known as Character streams and the byte streams. The java.io package provides two sets of class hierarchies to handle character and byte streams for reading and writing.
  1. InputStream and OutputStream classes are operated on bytes for reading and writing.
  2. Classes Reader and Writer are operated on characters for reading and writing.
- There are two other classes that are useful for handling input and output.
  1. File class and
  2. RandomAccessFile class.

### *Character streams*

- Reader and Writer classes are used to read/write the 16-bit characters input and output stream. However, it is an abstract class and can't be instantiated, but there are various subclasses that inherit these classes and override the methods of it. Methods of these classes throw an IOException. All the methods in the Writer class havae return type void.
- They are normally divided into two types. (i) Those that only read from or on write on to streams and (ii) those that also process the data that was read/written.

**Description of the four classes in the package java.io.**

| Name of class | Description |
|---|---|
| Reader, Writer | Supports read/write 16-bit Unicode characters. Used for only text data. |
| InputStream, OutputStream | Define basic methods for I/O streams of data. These classes are used only for binary data |
| File | Enables creating, deleting and renaming files, navigating through the file system, testing file existence and finding information about files. |
| RandomAccessFile | Enables the program to read/write from/to any location in the file, not just the beginning/end of the file, is the case as in the usual sequential access. The file works as a random-access disk file |

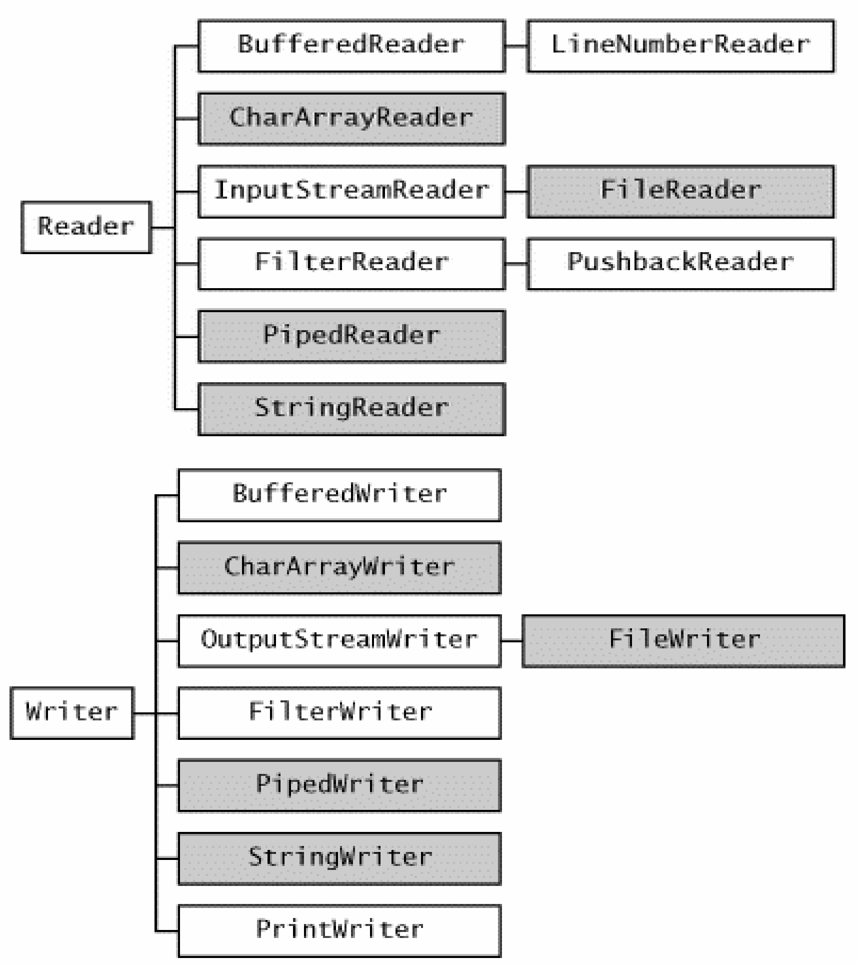


Figure shows Reader and Writer class hierarchy

| Various Reader and Writer classes and their description | |
|---|---|
| **Class Name** | **Description** |
| BufferedReader | This class provides methods to read characters from the buffer. |
| BufferedWriter | This class provides methods to write characters to the buffer. |
| CharArrayReader | This class provides methods to read characters from the char array. |
| CharArrayWriter | This class provides methods to write the characters to the character array. |
| FileReader | This class provides methods to read characters from the file. |
| FileWriter | This class provides methods to write characters to the file. |
| FilterReader | This class provides methods to read characters from the underlying character input stream. |

| | |
|---|---|
| FilterWriter | This class provides methods to read characters from the underlying character output stream. |
| InputStreamReader | This class provides methods to convert bytes to characters. |
| OutpuStreamWriter | This class provides methods to convert from bytes to characters. |
| PipedReader | This class provides methods to read characters from the connected piped output stream. |
| PipedWriter | This class provides methods to write the characters to the piped output stream. |
| StringReader | This class provides methods to read characters from a string. |
| StringWriter | This class provides methods to write the characters to the string. |

**Example: Writer and FileWriter**

```
import java.io.*;
public class WriterExample
{
        public static void main(String[] args)
        {
                try
                {
                        Writer w = new FileWriter("file.txt");
                        String content = "I love my country";
                        w.write(content);
                        w.close();
                        System.out.println("Done");
                }
                catch (IOException e)
                {
                        e.printStackTrace();
                }
        }
}
```

**OUTPUT**

The above program write the statement ***"I Love my country"*** into the ***"file.txt"*** file.

**Example: Reader and FileReader**

```
import java.io.*;
public class ReaderExample
        public static void main(String[] args)
        {
                try
                {
                        Reader reader = new FileReader("file.txt");
                        int data = reader.read();
                        while (data != -1)
                        {
                                System.out.print((char) data);
                                data = reader.read();
                        }
                        reader.close();
                }
                catch (Exception ex)
                {
                        System.out.println(ex.getMessage());
                }
        }
}
```

**OUTPUT**

The above program read the file ***"file.txt"*** and prints the statement ***"I Love my country"***

***Byte Streams***

- Java Byte streams are used to perform input and output of 8-bit bytes.
- The InputStream and OutputStream classes (abstract) are the super classes of all the input/output stream classes: classes that are used to read/write a stream of bytes.

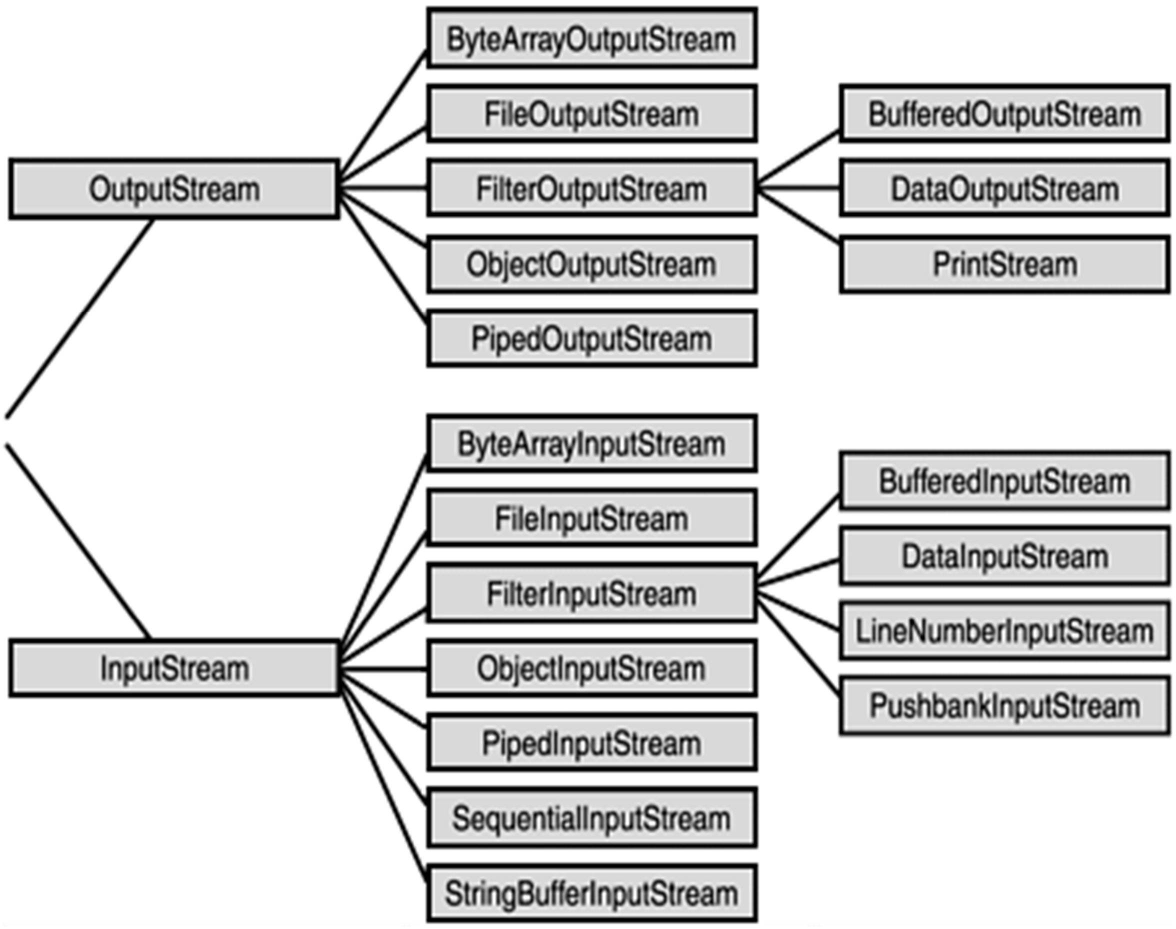


Figure shows InputStream and OutputStream class hierarchy

**Example: FileOutputStream Class**

```
import java.io.FileOutputStream;
public class FileOutputStreamExample
{
	public static void main(String args[])
	{
		try
		{
			FileOutputStream fout=new
	FileOutputStream("D:\\testout.txt");
			String s="Welcome to JAVA";
			byte b[]=s.getBytes();//converting string into byte array
			fout.write(b);
			fout.close();
			System.out.println("Program is executed successfully...");
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
}
```

**OUTPUT:**

```
Program is executed successfully...
```

The content of a text file ***testout.txt*** is set with the data ***Welcome to JAVA***

**Example: FileInputStream Class**

```
import java.io.FileInputStream;
public class DataStreamExample
{
	public static void main(String args[])
	{
		try
		{
			FileInputStream fin=new FileInputStream("D:\\testout.txt");
			int i=fin.read();                //int i=0;
			System.out.print((char)i);       //while((i=fin.read())!=-1){
			                                 //System.out.print((char)i);}
			fin.close();
		}
		catch(Exception e){System.out.println(e);}
	}
}
```

**OUTPUT:**

```
Welcome to JAVA
```

**NOTE**: Before running the code, a text file named as "testout.txt" is required to be created.

**FilterStream Class**

- Java FilterOutputStream class implements the OutputStream class. It provides different sub classes such as BufferedOutputStream and DataOutputStream to provide additional functionality.
- Java FilterInputStream class implements the InputStream. It contains different sub classes as BufferedInputStream, DataInputStream for providing additional functionality.

**Example: FilterInputStream Class**

```
import java.io.*;
public class FilterExample
{
        public static void main(String[] args) throws IOException
        {
                File data = new File("D:\\testout.txt");
                FileInputStream  file = new FileInputStream(data);
                FilterInputStream filter = new BufferedInputStream(file);
                int k =0;
                while((k=filter.read())!=-1)
                {
                        System.out.print((char)k);
                }
                file.close();           filter.close();
   }
}
```

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

**OUTPUT:**

```
        Welcome to JAVA
```

**Example: FilterOutputStream Class**

```
import java.io.*;
public class FilterExample
{
        public static void main(String[] args) throws IOException
        {
                File data = new File("D:\\testout.txt");
                FileOutputStream file = new FileOutputStream(data);
                FilterOutputStream filter = new FilterOutputStream(file);
                String s="Welcome to Jamal Mohamed College.";
                byte b[]=s.getBytes();
                filter.write(b);
                filter.flush();   filter.close();   file.close();
                System.out.println("Program is executed successfully....");
        }
}
```

**OUTPUT:**

```
        Program is executed successfully...
```

**RANDOM ACCESS FILE**

- This class is used for reading and writing to random access file.
- A random access file behaves like a large array of bytes. There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations.
- If end-of-file is reached before the desired number of byte has been read than EOFException is thrown. It is a type of IOException.

**Constructor**

| Constructor | Description |
|---|---|
| RandomAccessFile(File file, String mode) | Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument. |
| RandomAccessFile(String name, String mode) | Creates a random access file stream to read from, and optionally to write to, a file with the specified name. |

**Example**

```
// Java Program illustrating use of io.RandomAccessFile class methods
// read(), read(byte[] b), readBoolean(), readByte(), readInt()
// readFully(byte[] b, int off, int len), readFully(), readFloat()
// readChar(), readDouble(),
 import java.io.*;
public class NewClass
{
    public static void main(String[] args)
    {
        try
        {
            double d = 1.5;
            float f = 14.56f;

            // Creating a new RandomAccessFile - "testout"
            RandomAccessFile RAF = new RandomAccessFile("testout.txt", "rw");

            // Writing to file
            RAF.writeUTF("I Love my Country");

            // File Pointer at index position - 0
            RAF.seek(0);

            // read() method :
            System.out.println("Use of read() method : " + RAF.read());
            RAF.seek(0);
            byte[] b = {1, 2, 3};

            // Use of .read(byte[] b) method :
```

```
            System.out.println("Use of .read(byte[] b) : " + RAF.read(b));

            // readBoolean() method :
            System.out.println("Use of readBoolean() : " + RAF.readBoolean());

            // readByte() method :
            System.out.println("Use of readByte() : " + RAF.readByte());
            RAF.writeChar('c');
            RAF.seek(0);

            // readChar() :
            System.out.println("Use of readChar() : " + RAF.readChar());
            RAF.seek(0);
            RAF.writeDouble(d);
            RAF.seek(0);

            // read double
            System.out.println("Use of readDouble() : " + RAF.readDouble());
            RAF.seek(0);
            RAF.writeFloat(f);
            RAF.seek(0);

            // readFloat() :
            System.out.println("Use of readFloat() : " + RAF.readFloat());
            RAF.seek(0);

            // Create array upto RAF.length
            byte[] arr = new byte[(int) RAF.length()];
            // readFully() :
            RAF.readFully(arr);
            String str1 = new String(arr);
            System.out.println("Use of readFully() : " + str1);
            RAF.seek(0);

            // readFully(byte[] b, int off, int len) :
            RAF.readFully(arr, 0, 8);
            String str2 = new String(arr);
            System.out.println("Use of readFully(byte[] b, int off, int len) : " + str2);
        }
        catch (IOException ex)
        {
            System.out.println("Something went Wrong");
            ex.printStackTrace();
        }
    }
}
```

**OUTPUT**

```
Use of read() method : 0
Use of .read(byte[] b) : 3
Use of readBoolean() : true
Use of readByte() : 108
Use of readChar() : c
Use of readDouble() : 1.5
Use of readFloat() : 14.56
Use of readFully() : I Love my Country
Use of readFully(byte[] b, int off, int len) : I Love my Country
```

## SERIALIZATION and DESERIALIZATOIN

### 1. Serialization

- Serialization in Java is a mechanism of writing the state of an object into a byte-stream.
- The serialization is platform-independent, it means you can serialize an object on one platform.
- For serializing the object, we call the ***writeObject()*** method of *ObjectOutputStream* class.

**Advantages of Java Serialization**

It is mainly used to travel object's state on the network (that is known as marshalling).

**Serializable Interface**

- ✓ ***Serializable*** is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability.
- ✓ The Serializable interface must be implemented by the class whose object needs to be persisted.
- ✓ The String class and all the wrapper classes implement the ***java.io.Serializable*** interface by default.

**Example: Serialization**

//***Student*** class implements Serializable interface. Its objects can be converted into stream.

```
import java.io.Serializable;
public class Student implements Serializable
{
        int id;
        String name;
        public Student(int id, String name)
        {
                this.id = id;
                this.name = name;
        }
}
```

//The main class (***Persist***) serialize the object of Student class from above code.

//The ***writeObject()*** method of ***ObjectOutputStream*** class provides the functionality to serialize the object.

```
import java.io.*;
class Persist
{
        public static void main(String args[])
        {
            try
            {
                //Creating the object
                Student s1 =new Student(211,"ravi");

                //Creating stream and writing the object
                FileOutputStream fout=new FileOutputStream("f.txt");
                ObjectOutputStream out=new ObjectOutputStream(fout);
                out.writeObject(s1);
                out.flush();
                //closing the stream
                out.close();
                System.out.println("Program is executed Successfully…");
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
}
```

The above program saving the state of the object in the file named f.txt.

**OUTPUT**

```
Program is executed Successfully…
```

**2. Deserialization**

- Deserialization is the process of reconstructing the object from the serialized state.
- The reverse operation of serialization is called ***deserialization*** where byte-stream is converted into an object.
- Deserialization process is platform-independent, it means you can deserialize it on a different platform.
- For deserialization we call the ***readObject()*** method of *ObjectInputStream* class.

**Example**

//An example where we are reading the data from a deserialized object

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

```
        {
                try
                {
                        //Creating stream to read the object
                        ObjectInputStream in=new ObjectInputStream (new FileInputStream
                        ("f.txt"));
                        Student s=(Student)in.readObject();
                        System.out.println(s.id+" "+s.name);
                        in.close();  //closing the stream
                  }
                  catch(Exception e)
                  {
                        System.out.println(e);
                  }
          }
    }
```

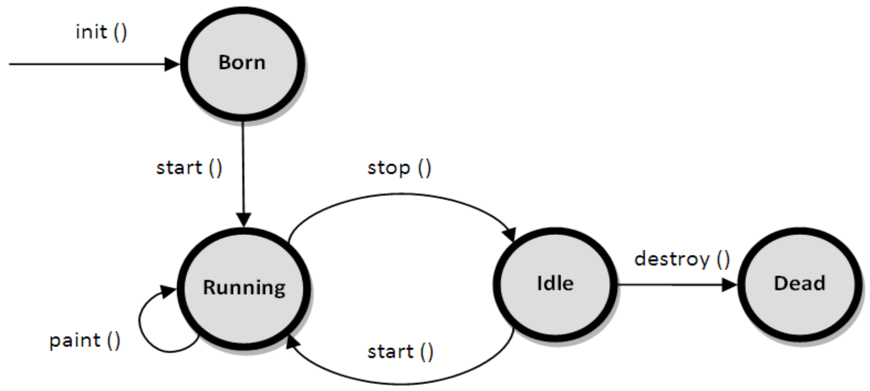
## Applets

- An applet is a software component that enables client-side programming and facilitates text, graphics, audio, imaging, animation and networking, besides live updating and securing two-way interaction in web pages.
- An applet is a Java program that runs in a Web browser. It runs inside the web browser and works at client side.
- An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server.

| Java Application Versus Java Applet | |
|---|---|
| **Java Application** | **Java Applet** |
| Applications are just like a Java programs that can be execute independently without using the web browser. | Applets are small Java programs that are designed to be included with the HTML web document. They require a Java-enabled web browser for execution. |
| Application program requires a main function for its execution. | Applet does not require a main function for its execution. |
| Java application programs have the full access to the local file system and network. | Applets don't have local disk and network access. |
| Applications can access all kinds of resources available on the system. | Applets can only access the browser specific services. They don't have access to the local system. |
| Applications can executes the programs from the local system. | Applets cannot execute programs from the local machine. |
| An application program is needed to perform some task directly for the user. | An applet program is needed to perform small tasks or the part of it. |

**Life Cycle of an Applet**

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



It is important to understand the order in which the various methods shown in the above image are called.

- When an applet begins, the following methods are called, in this sequence
  1. init( )
  2. start( )
  3. paint( )
- When an applet is terminated, the following sequence of method calls takes place
  1. stop( )
  2. destroy( )

**1. init ( ) :**

init() method is used to initialize an applet. It is invoking only once at the time of initialization. Initialized objects are created by the web browser. We can compare this method with a Thread class born state.

**2. start ( ) :**

start() method is used to start the applet. It is invoking after the init()method invoked. It is invoking each time when browser loading or refreshing. Until init() method is invoked start() method is inactive state.

**3. paint ( ) :**

- ✓ The **paint ()** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons.
- ✓ **paint ()** method is used for painting any shapes like square, rectangle, trapezium, eclipse, etc. paint() method has parameter as ClassGraphics. This Graphics class gives painting features in an applet.
- ✓ **paint ()** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.
- ✓ **paint** () method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

**Note:** This is the only method among all the method mention above, which is parametrised. Its prototype is

***public void paint(Graphics g)***

- where g is an object reference of class Graphic.

**4. stop ( )** :

✓ stop() method is used to stop the applet. It is invoked every time when browser stopped or minimized or abrupt failure of the application.

✓ After stop() method called, we can also start() method whenever we want. This method mainly deals with clean up code.

**5. destroy ( )** :

destroy() method is used to destroy the application once we have done with our applet work. It is invoked only once. Once applet is destroyed we can't start()the applet.

**How does Applet Life-Cycle Works in Java?**

1. Applet is a Java application that runs in any web browser and working at a client-side window. As it is running in the browser so, it doesn't have a main () method so Applet is designed to be placed within an HTML page.

2. java.applet.Appletclass provides init(), start(), stop() and destroy() methods.

3. java.awt.Componentclass provides another method of paint().

4. In Java, if any class wants to become Applet Class, it must inherit the Applet class.

5. init() method
   **Syntax:**
   ```
   public void init()
   {
           //initialized objects
   }
   ```

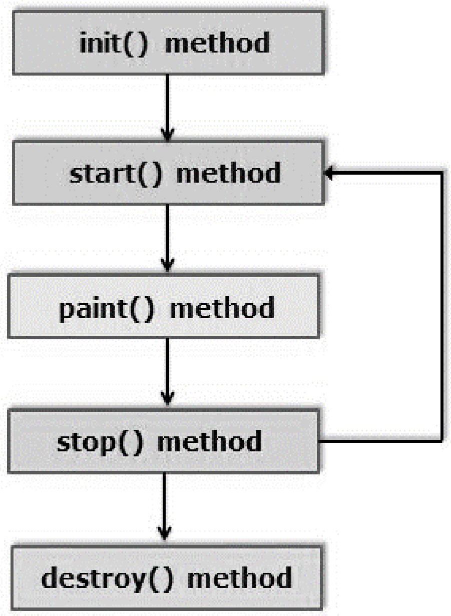
6. start() method
   **Syntax:**
   ```
   public void start()
   {
           //start the applet code
   }
   ```

7. stop() method
   **Syntax:**
   ```
   public void stop()
   {
           //stop the applet code
   }
   ```

8. destroy() method
   **Syntax:**
   ```
   public void destroy()
   {
           //destroy the applet code
   }
   ```

9. paint() method
   **Syntax:**
   ```
   public void paint(Graphics graphics)
   {
           //any shapes code
   }
   ```

10. All of the above methods are automatically called by the browser. We no need to call explicitly. Even though each method has its own specification to full fill the requirement as we discussed above.
    The flow of the methods is given below.



Applet Life Cycle - Flow of the methods

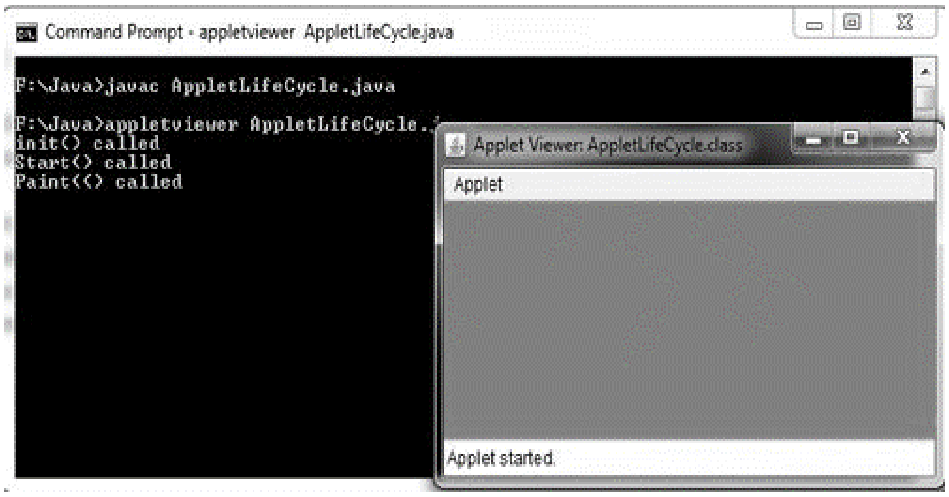
11. Applet life cycle is managed by Java Plug-in software.

12. There are **two standard ways** in which you can run an applet:
    a. Executing the applet within a **Java-compatible web browser**.
    b. Using an applet viewer, such as the standard tool, **applet-viewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

**Example: Life Cycle of Applet in Java**

**AppletLifeCycle.java**

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.*;
 /*<applet code="AppletLifeCycle.class" width="350" height="150"> </applet>*/
public class AppletLifeCycle extends Applet
{
        public void init()
        {
                setBackground(Color.CYAN);
                System.out.println("init() called");
        }
        public void start()
        {
                System.out.println("Start() called");
        }
        public void paint(Graphics g)
        {
                System.out.println("Paint(() called");
        }
        public void stop()
        {
                System.out.println("Stop() Called");
        }
        public void destroy()
        {
                System.out.println("Destroy)() Called");
        }
}
```

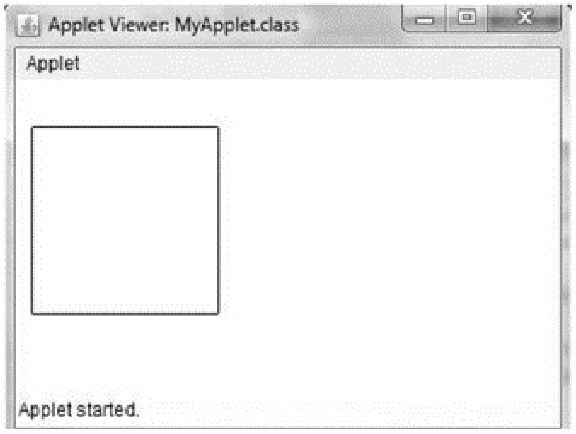


Command Prompt - appletviewer AppletLifeCycle.java

```
F:\Java>javac AppletLifeCycle.java

F:\Java>appletviewer AppletLifeCycle.j
init() called
Start() called
Paint(() called
```

**Example of an Applet**

```
import java.applet.*;
import java.awt.*;
public class MyApplet extends Applet
{
        int height, width;
        public void init()
        {
                height = getSize().height;
                width = getSize().width;
                setName("MyApplet");
         }
        public void paint(Graphics g)
        {
                g.drawRoundRect(10, 30, 120, 120, 2, 3);
        }
}
```



**How to run an Applet Program**

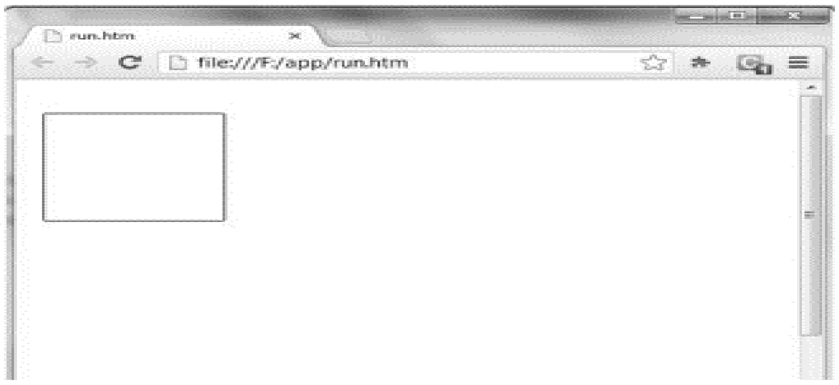
In the same manner as you compiled your console programs, an Applet program is compiled. There are, however, two methods of running an applet.

- Running the Applet in a web browser compatible with Java.
- Use an applet viewer, like the normal instrument, to view applets. In a window, an applet viewer runs your applet.

Create brief HTML file in the same folder to execute an Applet in a web browser. Include the following code in the file's body tag. (Applet tag loads class Applet).

***< applet code = "MyApplet" width=400 height=400 >***
***< /applet >***

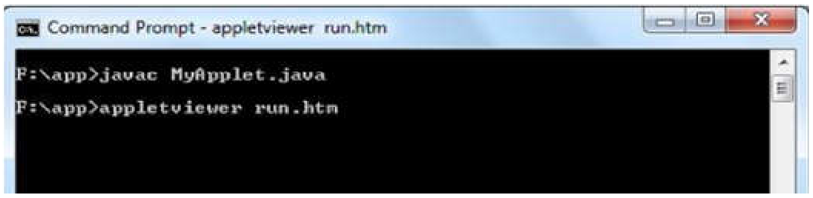
**Run the HTML file HTML file**



**Running Applet using Applet Viewer**

Write a brief HTML file as mentioned above to run an Applet with an applet viewer. If you name it as run.htm, your applet program will operate the following command.

**f:/>appletviewer run.htm**



```
F:\app>javac MyApplet.java
F:\app>appletviewer run.htm
```

**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:**

**UseParam.java**

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet
{
        public void paint(Graphics g)
        {
                String str=getParameter("msg");
                g.drawString(str,50, 50);
        }
}
```

**myapplet.html**

```
<html>
<body>
        <applet code="UseParam.class" width="300" height="300">
```

```
		<param name="msg" value="Welcome to applet">
		</applet>
	</body>
	</html>
```

**Example:**

**GraphicsDemo.java**

```
	import java.applet.Applet;
	import java.awt.*;
	public class GraphicsDemo extends Applet
	{
		public void paint(Graphics g)
		{
			g.setColor(Color.red);
			g.drawString("Welcome",50, 50);
			g.drawLine(20,30,20,300);
			g.drawRect(70,100,30,30);
			g.fillRect(170,100,30,30);
			g.drawOval(70,200,30,30);
			g.setColor(Color.pink);
			g.fillOval(170,200,30,30);
			g.drawArc(90,150,30,30,30,270);
			g.fillArc(270,150,30,30,0,180);
		}
	}
```

**myapplet.html**

```
	<html>
		<body>
			<applet code="GraphicsDemo.class" width="300" height="300">
			</applet>
		</body>
	</html>
```

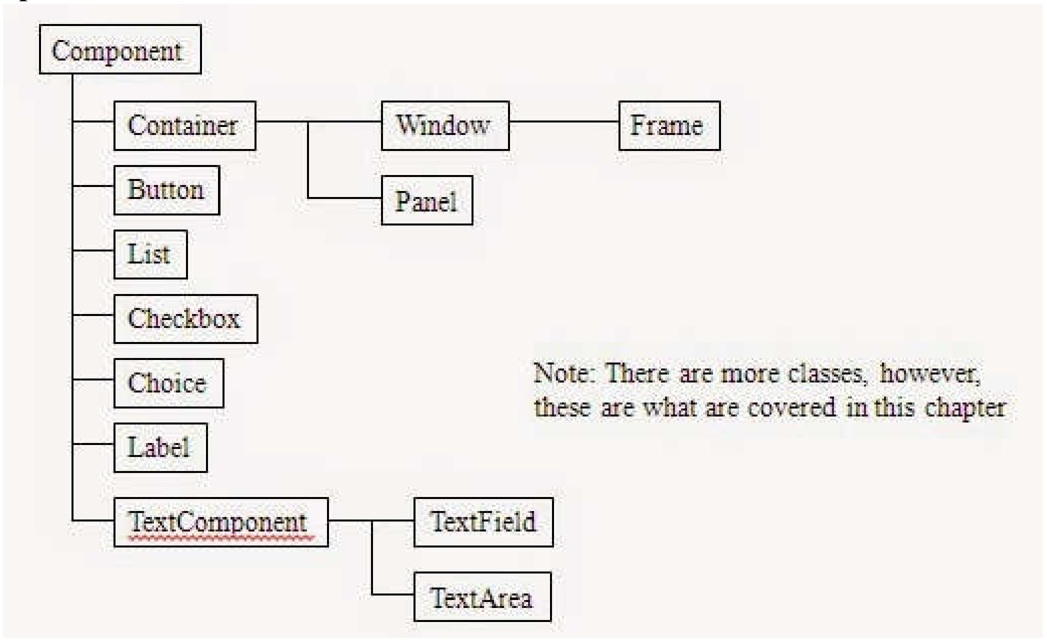
**UNIT V**

**Abstract Window Toolkit (AWT)**

- Abstract Window Toolkit (AWT) is a set of application program interfaces (APIs) used by Java programmers to create graphical user interface (GUI) objects, such as buttons, scroll bars, and windows.
- The Abstract Window Toolkit (AWT) is Java's original platform-independent windowing, graphics, and user-interface widget toolkit.
- The AWT is part of the Java Foundation Classes (JFC) — the standard API for providing a graphical user interface (GUI) for a Java program. AWT is also the GUI toolkit for a number of Java ME profiles

**AWT CLASS HIERARCHY**

**Component**



- Java.awt package contain all GUI Components
- Component is the superclass of most of the displayable classes defined within the AWT. Note: it is abstract.
- MenuComponent is another class which is similar to Component except it is the superclass for all GUI items which can be displayed within a drop-down menu.
- The Component class defines data and methods which are relevant to all Components
  - setBounds
  - setSize
  - setLocation
  - setFont
  - setEnabled
  - setVisible
  - setForeground
  - coloursetBackground -- colour

- The Component class contains the common features to all items which can be displayed in a GUI. Often, these items are called "**widgets**".
- In the AWT, all widgets are components and, as such, inherit all the data and methods of the Component class.

**AWT Controls**

**Container**

- Container is a subclass of Component. (ie. All containers are themselves, Components) Containers contain components. For a component to be placed on the screen, it must be placed within a Container.
- The Container class defined all the data and methods necessary for managing groups of Components:
  - add
  - getComponent
  - getMaximumSize
  - getMinimumSize
  - getPreferredSize
  - remove
  - removeAll
- The Container class is an abstract class which encapsulates the logic for managing Components.

**WINDOWS AND FRAMES**

- Generally speaking, the Window class is not used very often. The Frame class, on the other hand, is used quite extensively for GUI based applications.
- Another subclass of Window, which is not described here, I the Dialog class. It is used to display Dialog Boxes.
- Dialog Boxes are generally used to convey important information to the user, and must be dismissed by the user before the application can continue.
- It should be noted that dialog boxes disrupt the flow of an application and can cause great user frustration if not used appropriately.
- The Window class defines a top-level Window with no Borders or Menu bar.
- Usually used for application splash screens.
  - Frame defines a top-level Window with Borders and a Menu Bar
  - Frames are more commonly used than Windows Once defined, a Frame is a Container which can contain Components

```
Frame aFrame = new Frame("Hello World");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);
aFrame.setVisible(true);
```

**Panels**

- The Panel class is probably the most important class within the AWT.
- Panels can contain Components (which includes other Panels).
- It allows the GUI screen to be partitioned into manageable pieces.

- Panels should contain Components which are functionally related. For example, if an application wished to allow the user to input their name, address, phone number and other relevant contact information, it would be good design to place all of the necessary GUI Components on a Panel.
- That panel can be then added to and removed from other Containers within the application.

```
Panel aPanel = new Panel();
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Cancel"));
Frame aFrame = new Frame("Button Test");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);
aFrame.add(aPanel);
```

**Buttons**

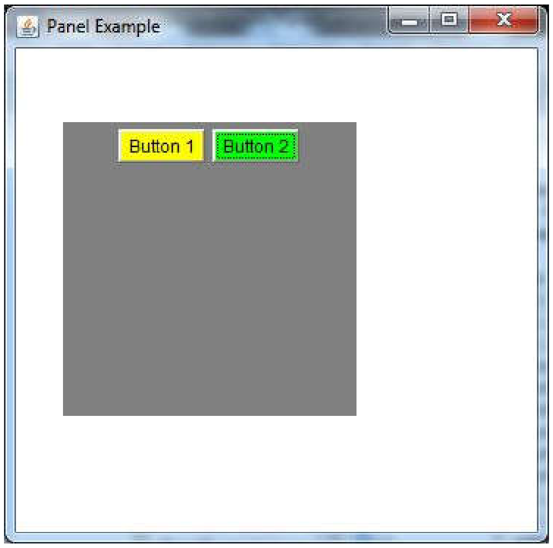
- All GUI systems offer some form of push button. The Button class in Java represents that functionality.
- Buttons are typically single purpose (i.e. their function does not change).
- When a button is pressed, it notifies its Listeners.
- To be a Listener for a button, an object must implement the ActionListener Interface.

```
Panel aPanel = new Panel();
Button okButton = new Button("Ok");
Button cancelButton = new Button("Cancel");
aPanel.add(okButton)); aPanel.add(cancelButton));
okButton.addActionListener(controller2);
cancelButton.addActionListener(controller1);
```

**Example: Implementatio of Panel and Buttons**

```
import java.awt.*;
public class PanelButton
{
        PanelButton()
        {
                Frame f= new Frame("Panel Example");
                Panel panel=new Panel();
                panel.setBounds(40,80,200,200);
                panel.setBackground(Color.gray);

                Button b1=new Button("Button 1");
                b1.setBounds(50,100,80,30);
                b1.setBackground(Color.yellow);
                Button b2=new Button("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 PanelButton();
        }
}
```

**OUTPUT**



**Labels**

- This class is a Component which displays a single line of text.
- Labels are read-only.
- That is, the user cannot click on a label to edit the text it displays.
- Text can be aligned within the label

```
Label aLabel = new Label("Enter password:");
aLabel.setAlignment(Label.RIGHT);
aPanel.add(aLabel);
```

**List**

- The List class comes under many names in different GUI systems.
- Lists provide a list of strings which can be selected by the user.
- The programmer may allow the user to select multiple strings within the list.
- This class is a Component which displays a list of Strings.
- The list is scrollable, if necessary.
- Sometimes called Listbox in other languages.
- Lists can be set up to allow single or multiple selections.

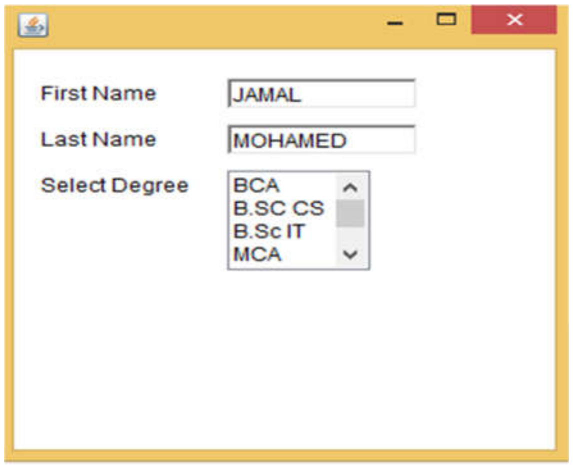
- The list will return an array indicating which Strings are selected

```
List aList = new List();
aList.add("Calgary");
aList.add("Edmonton");
aList.add("Regina");
aList.add("Vancouver");
aList.setMultipleMode(true);
```

**Example: Implementation of Label and List**

```
import java.awt.*;
class LabelList extends Frame
{
        LabelList()
        {
                Label firstName = new Label("First Name");
                firstName.setBounds(20, 50, 80, 20);

                Label lastName = new Label("Last Name");
                lastName.setBounds(20, 80, 80, 20);

                TextField firstNameTF = new TextField();
                firstNameTF.setBounds(120, 50, 100, 20);

                TextField lastNameTF = new TextField();
                lastNameTF.setBounds(120, 80, 100, 20);

                Label SelItems = new Label("Select Degree");
                SelItems.setBounds(20, 110, 80, 20);

                List l1=new List(4);
                l1.setBounds(120,110, 75,65);
                l1.add("BCA");
                l1.add("B.SC CS");
                l1.add("B.Sc IT");
                l1.add("MCA");

                add(firstName);  add(lastName);
                add(firstNameTF); add(lastNameTF);
                add(SelItems);   add(l1);
                setSize(300,300);  setLayout(null);
                setVisible(true);
        }
        public static void main(String[] args)
        {
                LabelList awt = new LabelList();
        }
}
```

**OUTPUT**



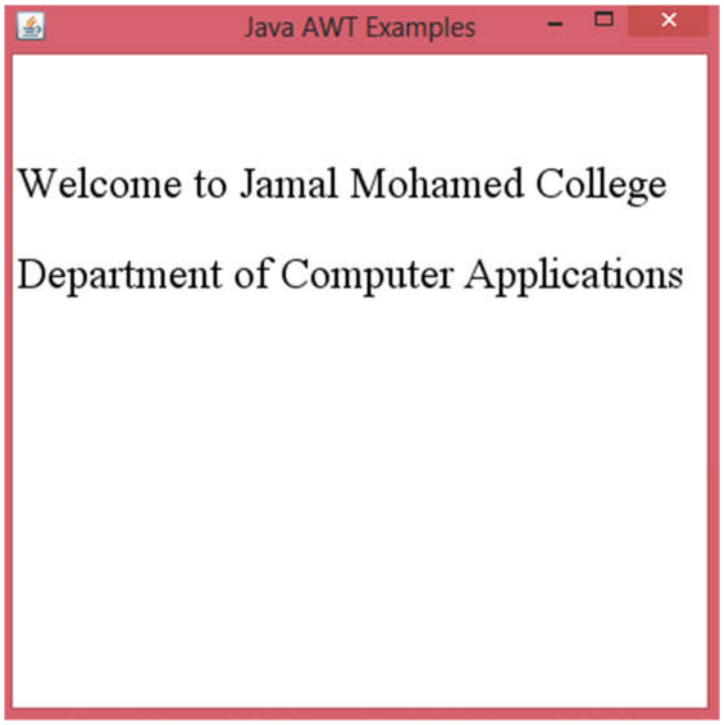
```
First Name      JAMAL
Last Name       MOHAMED
Select Degree   BCA
                B.SC CS
                B.Sc IT
                MCA
```

**Drawing with Graphics Class**

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

class AWTGraphicsDemo extends Frame
{

        public AWTGraphicsDemo()
        {
                super("Java AWT Examples");
                prepareGUI();
        }

  public static void main(String[] args)
  {
        AWTGraphicsDemo  awtGraphicsDemo = new AWTGraphicsDemo();
        awtGraphicsDemo.setVisible(true);
  }

  private void prepareGUI()
  {
        setSize(400,400);
        addWindowListener(new WindowAdapter()
        {
                public void windowClosing(WindowEvent windowEvent)
                {
                        System.exit(0);
                }
        });
  }
```

```
        @Override
        public void paint(Graphics g)
        {
                Graphics2D g2 = (Graphics2D)g;
                Font font = new Font("Times New Roman", Font.PLAIN, 24);
                g2.setFont(font);
                g2.drawString("Welcome to Jamal Mohamed College", 10, 110);
                g2.drawString("Department of Computer Applications", 10, 160);
        }
      }
```

OUTPUT:



```
Welcome to Jamal Mohamed College

Department of Computer Applications
```

**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:

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

2. ***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 and then returns.

- The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

***Steps Involved In Event Handling***

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class. The method is now get executed and returns.

***Callback Methods***

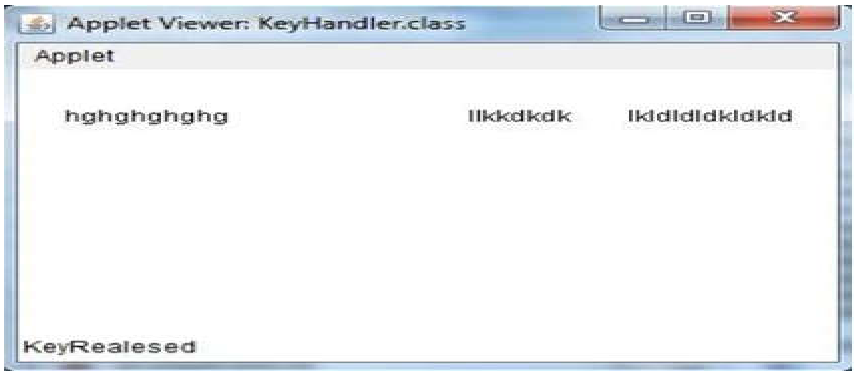
- These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.
- If a component wants some listener will listen to its events the source must register itself to the listener.

**Event Handling Example**

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
public class Test extends Applet implements KeyListener
{
        String msg="";
        public void init()
        {
           addKeyListener(this);
        }
        public void keyPressed(KeyEvent k)
        {
          showStatus("KeyPressed");
        }
        public void keyReleased(KeyEvent k)
        {
           showStatus("KeyRealesed");
        }
        public void keyTyped(KeyEvent k)
        {
          msg = msg+k.getKeyChar();
          repaint();
        }
        public void paint(Graphics g)
        {
          g.drawString(msg, 20, 40);
        }
  }
```

**HTML code:**

```
<applet code="Test" width=300, height=100>
</applet>
```



hghghghghg    llkkdkdk    lkdldldkldkld

KeyRealesed

**Layouts**

Layout means the arrangement of components within the container.
In other way we can say that placing the components at a particular position within the container. The task of layouting the controls is done automatically by the Layout Manager.

**LAYOUT MANAGER**

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. There are following classes that represents the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

***AWT Layout Manager Classes:***

Following is the list of commonly used controls while designed GUI using AWT.

**BorderLayout**

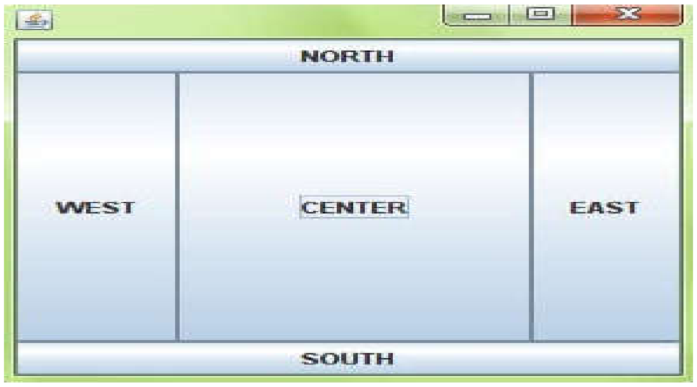
The borderlayout arranges the components to fit in the five regions: east, west, north, south and center. The BorderLayout provides five constants for each region:

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

Example

```
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**



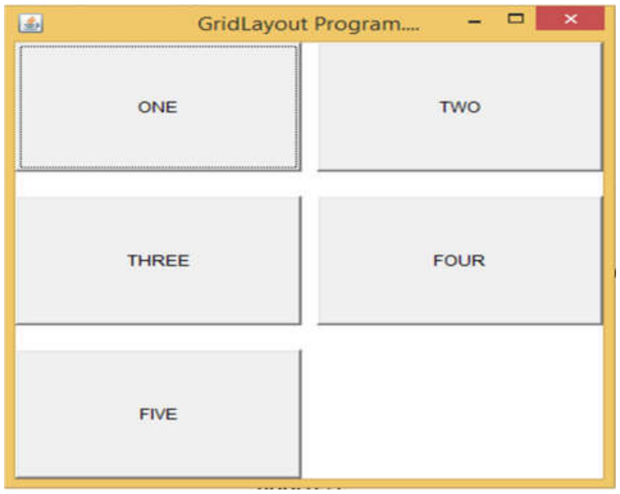
**GridLayout**

The GridLayout manages the components in form of a rectangular grid.

**Example**

```
import java.awt.*;
class griddemo extends Frame
{
        Button b1, b2, b3, b4, b5;
        griddemo()
        {
                        setSize(400,400);
                        setTitle("GridLayout Program....");
                        setVisible(true);
                        setLocation(200,150);
                        setLayout(new GridLayout(3,2,10,20));
                        b1 = new Button("ONE");
                        b2 = new Button("TWO");
                        b3 = new Button("THREE");
                        b4 = new Button("FOUR");
                        b5 = new Button("FIVE");

                        add(b1);
                        add(b2);
                        add(b3);
                        add(b4);
                        add(b5);
        }
        public static void main(String a[])
        {
                new griddemo();
        }
}
```

**OUTPUT**



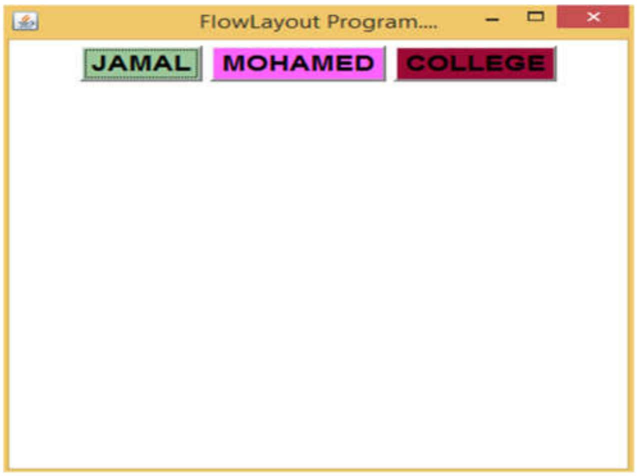
**FlowLayout**

The FlowLayout is the default layout.It layouts the components in a directional flow.

**Example**

```
import java.awt.*;
class flowdemo extends Frame
{
        Button b1, b2, b3;
        flowdemo()
        {
                        setSize(400,400);
                        setTitle("FlowLayout Program....");
                        setVisible(true);
                        setLocation(300,150);
                        setLayout(new FlowLayout());
                        Font f = new Font("Algerian",Font.BOLD,18);
                        Color c1 = new Color(155,200,155);
                        Color c2 = new Color(255,100,255);
                        Color c3 = new Color(155,10,55);

                        b1 = new Button("JAMAL");
                        b2 = new Button("MOHAMED");
                        b3 = new Button("COLLEGE");

                        b1.setFont(f);  b2.setFont(f);  b3.setFont(f);

                        b1.setBackground(c1);         b2.setBackground(c2);
                        b3.setBackground(c3);

                        add(b1);  add(b2);  add(b3);
        }
        public static void main(String a[])
        {
                new flowdemo();
        }
}
```

**OUTPUT**



# INTERVIEW TIPS TO IMPROVE YOUR PERFORMANCE

- *Dress for the job or company*
- *Keep your mobile switched off*
- *Be punctual at your interview*
- *Remember Body Language, Avoid Bad Habits.*
- *Listen - Good communication skills*
- *Don't appear desperate*
- *Be polite with everyone*
- *Be prepared for your interview*
- *Prepare for common job interview questions*
- *Close on a positive note*

*O S ABDUL QADIR*

*saq@jmc.edu*