OBJECT ORIENTED PROGRAMMING THROUGH JAVA

MCA II SESMESTER

S.Jaswanthi

Assistant professor

MCA Department

K.G.R.L P.G Courses

Bhimavaram

MCA-20202 Object Oriented Programming through JAVA

UNIT I

Introduction to OOP: Introduction, Principles of Object Oriented Languages, Applications of OOP, Programming Constructs: Variables, Primitive Datatypes, Identifiers- Naming Conventions, Keywords, Literals, Operators-Binary, Unary and ternary, Expressions, Precedence rules and Associativity, Primitive Type Conversion and Casting, Flow of control- Branching, Conditional, loops. Classes and Objects- classes, Objects, Creating Objects, Methods, constructors-Constructor overloading, cleaning up unused objects-Garbage collector, Class variable and Methods-Static keyword, this keyword, Arrays, Command line arguments.
Inheritance: Types of Inheritance, Deriving classes using extends keyword, Method overloading, super keyword, final keyword, Abstract class.

UNIT II

Interfaces, Packages and Enumeration: Interface-Extending interface, Interface Vs Abstract classes, Packages-Creating packages, using Packages, Access protection, java.lang package. **Exceptions & Assertions** – Introduction, Exception handling techniques- try… catch, throw, throws, finally block, user defined exception, Exception Encapsulation and Enrichment, Assertions.

UNIT III

MultiThreading: java.lang.Thread, The main Thread, Creation of new threads, Thread priority, Multithreading- Using isAlive () and join (), Synchronization, suspending and Resuming threads, Communication between Threads Input/Output: reading and writing data, java.io package, **Applets**– Applet class, Applet structure, An Example Applet Program, Applet: Life Cycle, paint(), update() and repaint().

UNIT IV

Event Handling -Introduction, Event Delegation Model, java.awt.event Description, Sources of Events, Event Listeners, Adapter classes, Inner classes.
Abstract Window Toolkit:Why AWT?, java.awt package, Components and Containers, Button, Label, Checkbox, Radio buttons, List boxes, Choice boxes, Text field and Text area, container classes, Layouts, Menu, Scroll bar, **Swing**: Introduction, JFrame, JApplet, JPanel, Components in swings, Layout Managers, JList and JScroll Pane, Split Pane, JTabbedPane, Dialog Box Pluggable Look and Feel.

Text Books:

1. The Complete Reference Java, 8ed, Herbert Schildt, TMH
2. Programming in JAVA, Sachin Malhotra, Saurabhchoudhary, Oxford.

References:

1. JAVA for Beginners, 4e, Joyce Farrell, Ankit R. Bhavsar, Cengage Learning.

OBJECT ORIENTED PROGRAMMING Through JAVA

UNIT I

Introduction to OOP: Introduction, Principles of Object Oriented Languages, Applications of OOP, Programming Constructs: Variables, Primitive Datatypes, Identifiers- Naming Conventions, Keywords, Literals, Operators-Binary, Unary and ternary, Expressions, Precedence rules and Associativity, Primitive Type Conversion and Casting, Flow of control- Branching, Conditional, loops. Classes and Objects- classes, Objects, Creating Objects, Methods, constructors-Constructor overloading, cleaning up unused objects-Garbage collector, Class variable and Methods-Static keyword, this keyword, Arrays, Command line arguments.
Inheritance: Types of Inheritance, Deriving classes using extends keyword, Method overloading, super keyword, final keyword, Abstract class.

Introduction To OOP

Introduction:

JAVA was developed by James Gosling at **Sun Microsystems** Inc in the year **1991**, later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs.
Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to write once run anywhere that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is similar to c/c++.

History:

Java's history is very interesting. It is a programming language created in 1991. James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the **Green team** initiated the Java language in 1991. **Sun Microsystems** released its first public implementation in 1996 as **Java 1.0**. It provides no-cost -run-times on popular platforms. Java1.0 compiler was re-written in Java by Arthur Van Hoff to strictly comply with its specifications. With the arrival of Java 2, new versions had multiple configurations built for different types of platforms.

In 1997, Sun Microsystems approached the ISO standards body and later formalized Java, but it soon withdrew from the process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System.

On November 13, 2006, Sun released much of its Java virtual machine as free, open-source software. On May 8, 2007, Sun finished the process, making all of its JVM's core code available under open-source distribution terms.

The principles for creating java were simple, robust, secured, high performance, portable, multi-threaded, interpreted, dynamic, etc. **James** Gosling in 1995 developed Java, who is known as the Father of Java. Currently, Java is used in mobile devices, internet programming, games, e-business, etc.

*****

Principles of Object Oriented Languages:

As the name suggests, Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOPs Concepts:

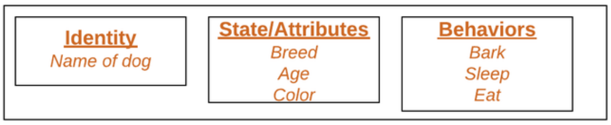
1.Class
2.Objects
3.Data Abstraction
4.Encapsulation
5.Inheritance
6.Polymorphism
7.Dynamic Binding
8.Message Passing

1. Class:

A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

2. Object:

It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.



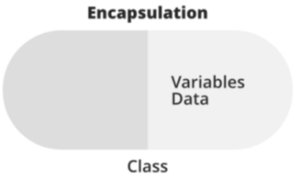
Object

3. Data Abstraction:

Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

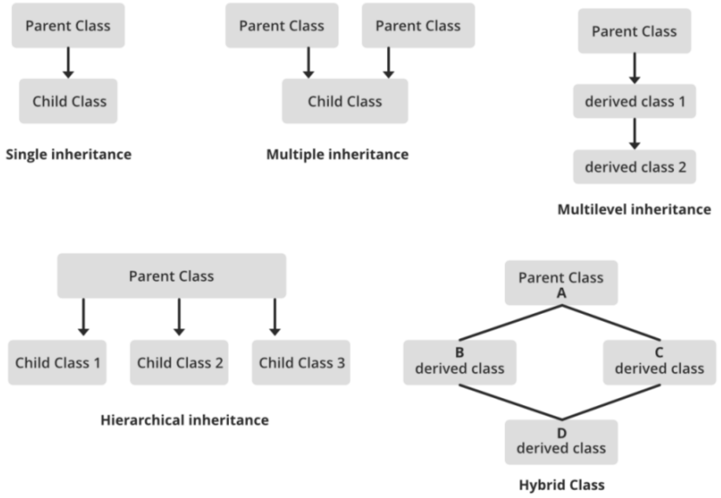
4. Encapsulation:

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.



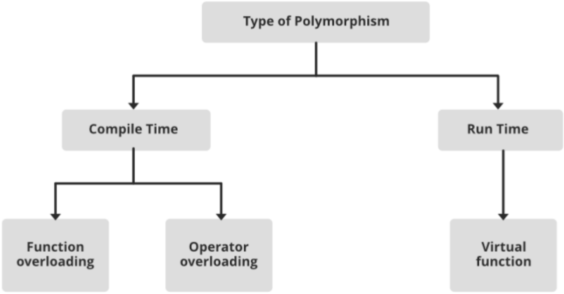
5. Inheritance:

Inheritance is an important pillar of OOP(Object-Oriented Programming). The capability of a class to derive properties and characteristics from another class is called Inheritance. When we write a class, we inherit properties from other classes. So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.



6. Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.



7. Dynamic Binding:

In dynamic binding, the code to be executed in response to the function call is decided at runtime. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time.

8. Message Passing:

It is a form of communication used in object-oriented programming as well as parallel programming. Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

*****

Applications of OOP:

- OOP has become one of the programming buzz words today. There appears to be a great deal of excitement and interest among software engineers in using OOP.
- Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as window. Hundreds of windowing systems have been developed, using the OOP techniques. Real-business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem. The promising areas of application of OOP include:

Main application areas of OOP are:

1.User interface design such as windows, menu.
2. Real Time Systems
3.Simulation and Modeling
4.Object oriented databases
5.AI and Expert System
6.Neural Networks and parallel programming
7.Decision support and office automation systems etc.

Benefits of OOP:

- It is easy to model a real system as real objects are represented by programming objects in OOP. The objects are processed by their member data and functions. It is easy to analyze the user requirements.
- With the help of inheritance, we can reuse the existing class to derive a new class such that the redundant code is eliminated and the use of existing class is extended. This saves time and cost of program.
- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that can not be invaded by code in other part of the program.
- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.
- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.
- It is possible to have multiple instances of an object to co-exist without any interference i.e. each object has its own separate member data and function.

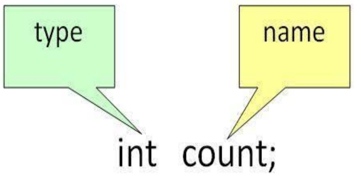
*****

Programming Constructs:-

1. Variables:

A variable is a name given to a memory location. It is the basic unit of storage in a program.

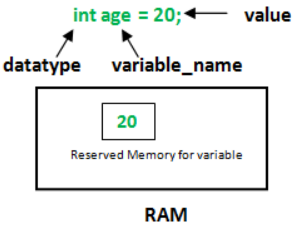
- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.



type: Type of data that can be stored in this variable.
name: Name given to the variable.
In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input

How to initialize variables?



datatype: Type of data that can be stored in this variable.
variable_name: Name given to the variable.
value: It is the initial value stored in the variable.

Example:

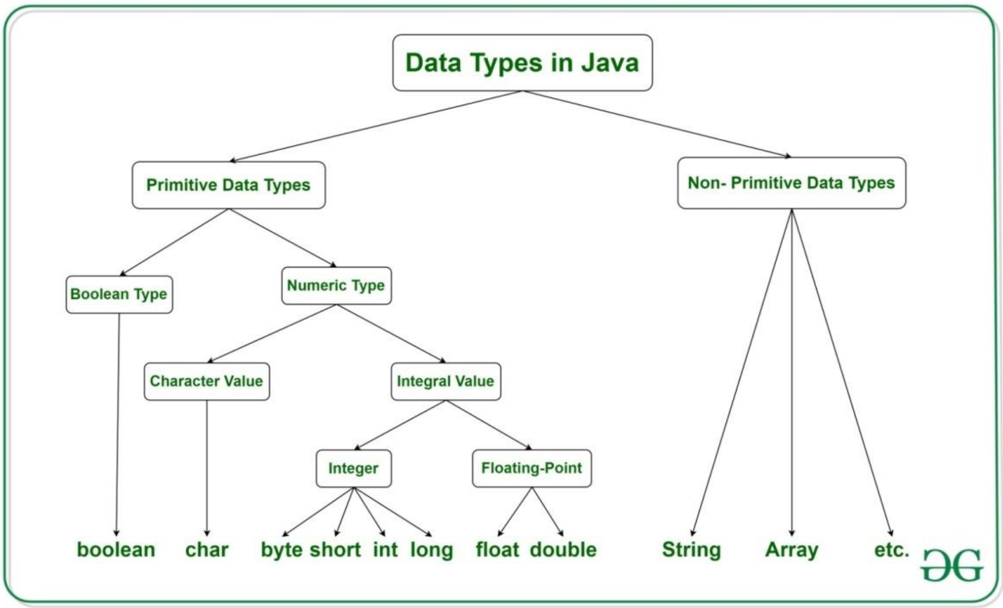
float simpleInterest; //Declaring float variable

int time = 10, speed = 20; //Declaring and Initializing integer variable

char var = 'h'; // Declaring and Initializing character variable

*

2. Primitive Datatypes:



Java has two categories of data:

- **Primitive Data Type:** such as boolean, char, int, short, byte, long, float, and double
- **Non-Primitive Data Type or Object Data type:** such as String, Array, etc.

Primitive Data Type:Primitive data are only single values and have no special capabilities. There are **8 primitive data types**:

1. boolean: boolean data type represents only one bit of information **either true or false**, but the size of the boolean data type is **virtual machine-dependent**. Values of type boolean are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.
Syntax: boolean booleanVar;
Size: virtual machine dependent
Values: true, false
Default Value: false

2. byte: The byte data type is an 8-bit signed two's complement integer. The byte data type is useful for saving memory in large arrays.
Syntax: byte byteVar;
Size: 1 byte (8 bits)
Values: -128 to 127
Default Value: 0

3. short: The short data type is a 16-bit signed two's complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.
Syntax: short shortVar;
Size: 2 byte (16 bits)
Values: -32, 768 to 32, 767 (inclusive)

Default Value: 0

4. int: It is a 32-bit signed two's complement integer.
Syntax: int intVar;
Size: 4 byte (32 bits)
Values: -2, 147, 483, 648 to 2, 147, 483, 647 (inclusive)
Default Value: 0

5. long: The long data type is a 64-bit two's complement integer.
Syntax: long longVar;
Size: 8 byte (64 bits)
Values:-9, 223, 372, 036, 854, 775, 808
to

9, 223, 372, 036, 854, 775, 807

(inclusive)

Default Value: 0

6. float: The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers.
Syntax: float floatVar;
Size: 4 byte (32 bits)
Values: upto 7 decimal digits
Default Value: 0.0

7. double: The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice.
Syntax: double doubleVar;
Size: 8 byte (64 bits)
Values: upto 16 decimal digits
Default Value: 0.0

8. char: The char data type is a single 16-bit Unicode character.
Syntax: char charVar;
Size: 2 byte (16 bits)
Values: '\u0000' (0) to '\uffff' (65535)
Default Value: '\u0000'

*

3.Identifiers:

In programming languages, identifiers are used for identification purposes. In Java, an identifier can be a class name, method name, variable name, or label.

For example :

```
public class Test
{
   public static void main(String[] args)
   {
      int a = 20;
   }
}
```

In the above java program, we have 5 identifiers namely :

- **Test** : class name.
- **main** : method name.
- **String** : predefined class name.
- **args** : variable name.
- **a** : variable name.

Rules for defining Java Identifiers:

There are certain rules for defining a valid java identifier. These rules must be followed, otherwise we get compile-time error. These rules are also valid for other languages like C,C++.

- The only allowed characters for identifiers are all alphanumeric characters([**A-Z**],[**a-z**],[**0-9**]), '**$**'(dollar sign) and '**_**' (underscore).For example "geek@" is not a valid java identifier as it contain '@' special character.
- Identifiers should **not** start with digits(**[0-9]**). For example "123geeks" is a not a valid java identifier.
- Java identifiers are **case-sensitive**.
- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
- Reserved Words can't be used as an identifier. For example "int while = 20;" is an invalid statement as while is a reserved word. There are **53** reserved words in Java.

Examples of valid identifiers :

MyVariable

MYVARIABLE

myvariable

x

i

x1

i1

_myvariable

$myvariable

sum_of_array

geeks123

Reserved Words:

Any programming language reserves some words to represent functionalities defined by that language. These words are called reserved words.They can be briefly categorised into two parts : **keywords**(50) and **literals**(3). Keywords define functionalities and literals define a value. Identifiers are used by symbol tables in various analyzing phases(like lexical, syntax, semantic) of a compiler architecture.

*

4.Naming Conventions:

Below are some naming conventions of java programming language. They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constants.

Camel case in Java Programming : It consists of compound words or phrases such that each word or abbreviation begins with a capital letter or first word with a lowercase letter, rest all with capital.

1. **Classes and Interfaces :**
   - Class names should be **nouns**, in mixed case with the **first** letter of each internal word capitalised. Interfaces name should also be capitalised just like class names.
   - Use whole words and must avoid acronyms and abbreviations.

   Examples:

   ```
   interface  Bicycle
   class MountainBike implements Bicyle
   interface Sport
   class Football implements Sport
   ```

2. **Methods :**

   Methods should be **verbs**, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalised.

   Examples:

   ```
   void changeGear(int newValue);
   void speedUp(int increment);
   void applyBrakes(int decrement);
   ```

3. **Variables :** Variable names should be short yet meaningful.
   - Variables can also start with either underscore('_') or dollar sign '$' characters.
   - Should be mnemonic i.e, designed to indicate to the casual observer the intent of its use.
   - **One-character variable names should be avoided** except for temporary variables.
   - Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

   Examples:

   ```
   // variables for MountainBike class

   int speed = 0;
   ```

int gear = 1;

4. **Constant variables:**
    - Should be **all uppercase** with words separated by underscores (“_”).
    - There are various constants used in predefined classes like Float, Long, String etc.

Examples:

static final int MIN_WIDTH = 4;

// Some Constant variables used in predefined Float class

public static final float POSITIVE_INFINITY = 1.0f / 0.0f;

public static final float NEGATIVE_INFINITY = -1.0f / 0.0f;

public static final float NaN = 0.0f / 0.0f;

5. **Packages:**
    - The prefix of a unique package name is always written in **all-lowercase ASCII letters** and should be one of the top-level domain names, like com, edu, gov, mil, net, org.
    - Subsequent components of the package name vary according to an organisation’s own internal naming conventions.

Examples:

com.sun.eng

com.apple.quicktime.v2

// java.lang packet in JDK

java.lang

*

5.Keywords:

Keywords or Reserved words are the words in a language that are used for some internal process or represent some predefined actions. These words are therefore not allowed to use as a variable names or objects. Doing this will result into a **compile time error**.
Java also contains a list of reserved words or keywords. These are:

1. **abstract** -Specifies that a class or method will be implemented later, in a subclass
2. **assert** -Assert describes a predicate (a true–false statement) placed in a Java program to indicate that the developer thinks that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort.
3. **boolean** – A data type that can hold True and False values only
4. **break** – A control statement for breaking out of loops
5. **byte** – A data type that can hold 8-bit data values
6. **case** – Used in switch statements to mark blocks of text
7. **catch** – Catches exceptions generated by try statements
8. **char** – A data type that can hold unsigned 16-bit Unicode characters
9. **class** -Declares a new class
10. **continue** -Sends control back outside a loop
11. **default** -Specifies the default block of code in a switch statement

12. **do** -Starts a do-while loop
13. **double** – A data type that can hold 64-bit floating-point numbers
14. **else** – Indicates alternative branches in an if statement
15. **enum** – A Java keyword used to declare an enumerated type. Enumerations extend the base class.
16. **extends** -Indicates that a class is derived from another class or interface
17. **final** -Indicates that a variable holds a constant value or that a method will not be overridden
18. **finally** -Indicates a block of code in a try-catch structure that will always be executed
19. **float** -A data type that holds a 32-bit floating-point number
20. **for** -Used to start a for loop
21. **if** -Tests a true/false expression and branches accordingly
22. **implements** -Specifies that a class implements an interface
23. **import** -References other classes
24. **instanceof** -Indicates whether an object is an instance of a specific class or implements an interface
25. **int** – A data type that can hold a 32-bit signed integer
26. **interface** – Declares an interface
27. **long** – A data type that holds a 64-bit integer
28. **native** -Specifies that a method is implemented with native (platform-specific) code
29. **new** – Creates new objects
30. **null** -Indicates that a reference does not refer to anything
31. **package** – Declares a Java package
32. **private** -An access specifier indicating that a method or variable may be accessed only in the class it's declared in
33. **protected** – An access specifier indicating that a method or variable may only be accessed in the class it's declared in (or a subclass of the class it's declared in or other classes in the same package)
34. **public** – An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible)
35. **return** -Sends control and possibly a return value back from a called method
36. **short** – A data type that can hold a 16-bit integer
37. **static** -Indicates that a variable or method is a class method (rather than being limited to one particular object)
38. **strictfp** – A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability.
39. **super** – Refers to a class's base class (used in a method or class constructor)
40. **switch** -A statement that executes code based on a test value
41. **synchronized** -Specifies critical sections or methods in multithreaded code
42. **this** -Refers to the current object in a method or constructor
43. **throw** – Creates an exception
44. **throws** -Indicates what exceptions may be thrown by a method
45. **transient** -Specifies that a variable is not part of an object's persistent state
46. **try** -Starts a block of code that will be tested for exceptions
47. **void** -Specifies that a method does not have a return value
48. **volatile** -Indicates that a variable may change asynchronously
49. **while** -Starts a while loop

*

6.Literals:

Any constant value which can be assigned to the variable is called as literal/constant.
// Here 100 is a constant/literal.

int x = 100;

Integral literals:

For Integral data types (byte, short, int, long), we can specify literals in 4 ways:-

1. **Decimal literals (Base 10) :** In this form the allowed digits are 0-9.
   int x = 101;

2. **Octal literals (Base 8) :** In this form the allowed digits are 0-7.
   // The octal number should be prefix with 0.

   int x = 0146;

3. **Hexa-decimal literals (Base 16) :** In this form the allowed digits are 0-9 and characters are a-f. We can use both uppercase and lowercase characters. As we know that java is a case-sensitive programming language but here java is not case-sensitive.
   // The hexa-decimal number should be prefix

   // with 0X or 0x.

   int x = 0X123Face;

4. **Binary literals :** From 1.7 onward we can specify literals value even in binary form also, allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.
   int x = 0b1111;

*****

Operators:

1.Binary Operators:

Bitwise operators are used to performing manipulation of individual bits of a number. They can be used with any of the integral types (char, short, int, etc).

BitwiseOR(|):

This operator is a binary operator, denoted by ‘|’. It returns bit by bit OR of input values, i.e, if either of the bits is 1, it gives 1, else it gives 0.

For example,

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7

```
  0101

| 0111

 ________

  0111  = 7 (In decimal)
```

BitwiseAND(&):

This operator is a binary operator, denoted by ‘&’. It returns bit by bit AND of input values, i.e, if both bits are 1, it gives 1, else it gives 0.

For example,

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise AND Operation of 5 and 7

```
  0101
& 0111
 ________
  0101  = 5 (In decimal)
```

BitwiseXOR(^):

This operator is a binary operator, denoted by '^'. It returns bit by bit XOR of input values, i.e, if corresponding bits are different, it gives 1, else it gives 0.

For example,

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7

```
  0101
^ 0111
 ________
  0010  = 2 (In decimal)
```

Bitwise Complement(~):

This operator is a unary operator, denoted by '~'. It returns the one's complement representation of the input value, i.e, with all bits inverted, which means it makes every 0 to 1, and every 1 to 0.

For example,

a = 5 = 0101 (In Binary)

Bitwise Compliment Operation of 5

```
~ 0101
 ________
  1010  = 10 (In decimal)
```

2.Unary Operators:

Java unary operators are the types that need only one operand to perform any operation like increment, decrement, negation etc. It consists of various arithmetic, logical and other operators that operate on a single operand. Let's look at the various unary operators in detail and see how they operate.

(i) Unary minus(-): This operator can be used to convert a negative value to a positive one.
Syntax:-(operand)
Example:a = -10

Below is the program to illustrate Java unary – operator.

```
// Java code to illustrate unary :
import java.io.*;
class Unary {
 public static void main(String[] args) {
// variable declaration
    int n1 = 20;
    System.out.println("Number = " + n1);
    // Performing unary operation
     n1 = -n1;
      // Print the result number
      System.out.println("Result = " + n1);
    }}
```

Output:

```
Number = 20

Result = -20
```

(ii) 'NOT' Operator(!): This is used to convert true to false or vice versa. Basically it reverses the logical state of an operand.

Syntax:!(operand)

Example:cond = !true;

// cond < false

Below is the program to illustrate Java unary ! operator.

```
// Java code to illustrate
// unary NOT operator
import java.io.*;
class Unary {
  public static void main(String[] args) {
    // initializing variables
        boolean cond = true;
        int a = 10, b = 1;
        // Displaying cond, a, b
        System.out.println("Cond is: " + cond);
        System.out.println("Var1 = " + a);
        System.out.println("Var2 = " + b);
        // Using unary NOT operator
        System.out.println("Now cond is: " + !cond);
        System.out.println("!(a < b) = " + !(a < b));
        System.out.println("!(a > b) = " + !(a > b));
    }}
```

Output:

```
Cond is: true

Var1 = 10

Var2 = 1

Now cond is: false

!(a < b) = true

!(a > b) = false
```

(iii) Increment(++): It is used to increment the value of an integer. It can be used in two separate ways:

a. **Post-increment operator:** When placed after the variable name, the value of the operand is incremented but the previous value is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.
Syntax:num++
Example:num = 5
num++ = 6

b. **Pre-increment operator:** When placed before the variable name, the operand's value is incremented instantly.
Syntax:++num
Example:num = 5
++num = 6

Below is the program to illustrate Java unary Increment(++) operator.

```
// Java code to illustrate increment operator
import java.io.*;
class Unary {
public static void main(String[] args){
  // initializing variables
   int num = 5;
    // first 5 gets printed and then
     // increment to 6
      System.out.println("Post "
                    + "increment = " + num++);
        // num was 6, incremented to 7
        // then printed
        System.out.println("Pre "
                    + "increment = " + ++num);
    }}
```

Output: Post increment = 5

```
Pre increment = 7
```

(iv) Decrement(--): It is used to decrement the value of an integer. It can be used in two separate ways:

c. **Post-decrement operator:** When placed after the variable name, the value of the operand is decremented but the previous values is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.
Syntax:num--

Example:num = 5
num-- = 4

d. **Pre-decrement operator:** When placed before the variable name, the operand’s value is decremented instantly.
Syntax:--num
Example:num = 5
--num = 4

Below is the program to illustrate Java unary Decrement(--) operator.

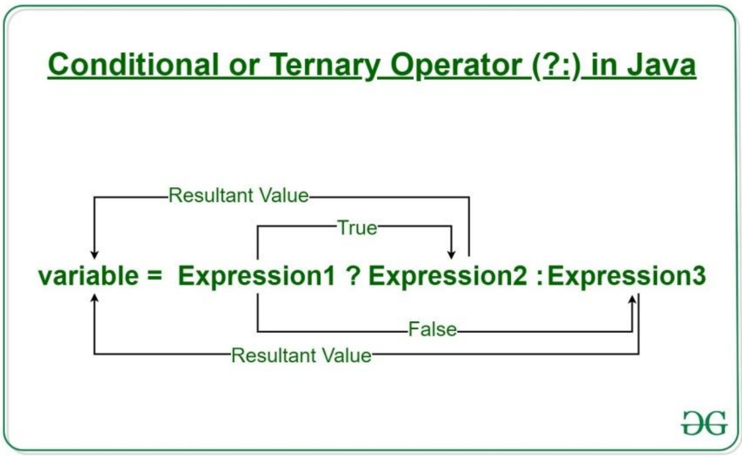
```
 // Java code to illustrate decrement operator
 import java.io.*;
 class Unary {
  public static void main(String[] args){
    // initializing variables
     int num = 5;
     // first 5 gets printed and then
      // decremented to 4
      System.out.println("Post "
                       + "decrement = " + num--);
       System.out.println("num = " + num);
       // num was 4, decremented to 3
       // then printed
       System.out.println("Pre "
                       + "decrement = " + --num);
          }}
```

Output:

```
Post decrement = 5

num = 4

Pre decrement = 3
```

3.Ternary Operators:

Java ternary operator is the only conditional operator that takes three operands. It’s a one-liner replacement for if-then-else statement and used a lot in Java programming. We can use the ternary operator in place of if-else conditions or even switch conditions using nested ternary operators. Although it follows the same algorithm as of if-else statement, the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.


Conditional or Ternary Operator (?:) in Java
Resultant Value
True
variable = Expression1 ? Expression2 : Expression3
False
Resultant Value

Syntax:

variable = Expression1 ? Expression2: Expression3

If operates similarly to that of the if-else statement as in **Exression2** is executed if **Expression1** is true else **Expression3** is executed.

```
if(Expression1)
{
   variable = Expression2;
}
else
{
   variable = Expression3;
}
```

Example:

num1 = 10;

num2 = 20;

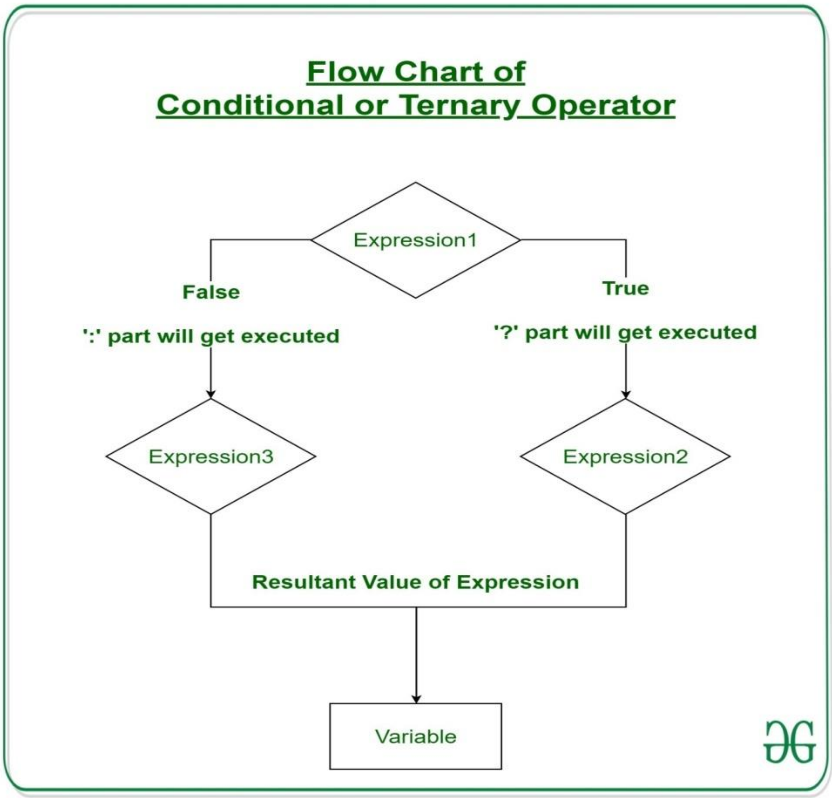
res=(num1>num2) ? (num1+num2):(num1-num2)

Since num1<num2,

the second operation is performed

res = num1-num2 = -10

Flowchart of Ternary Operation:


Flow Chart of
Conditional or Ternary Operator
Expression1
False
True
':' part will get executed
'?' part will get executed
Expression3
Expression2
Resultant Value of Expression
Variable

Example 1:

```
// Java program to find largest among two
// numbers using ternary operator
import java.io.*;
class Ternary {
 public static void main(String[] args)
    {
        // variable declaration
        int n1 = 5, n2 = 10, max;
        System.out.println("First num: " + n1);
        System.out.println("Second num: " + n2);
        // Largest among n1 and n2
        max = (n1 > n2) ? n1 : n2;
        // Print the largest number
        System.out.println("Maximum is = " + max);
    }}
```

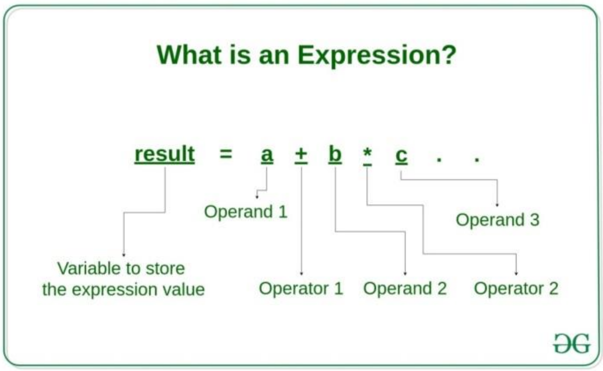
Output:

```
First num: 5

Second num: 10

Maximum is = 10
```

*****

Expressions:

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.



Example:a+b

c

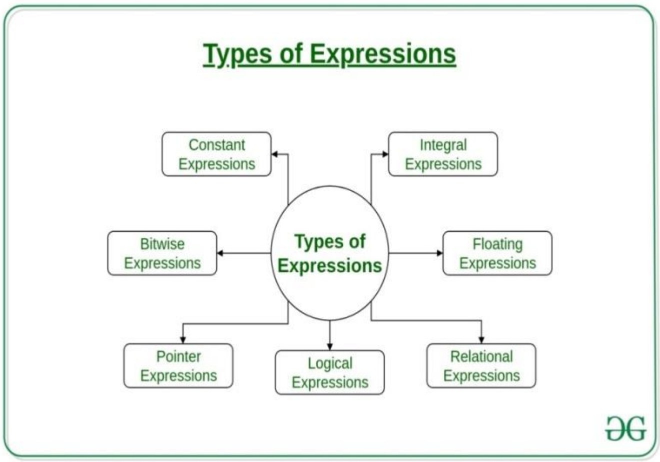
s-1/7*f

.

.

etc

Types of Expressions: Expressions may be of the following types:



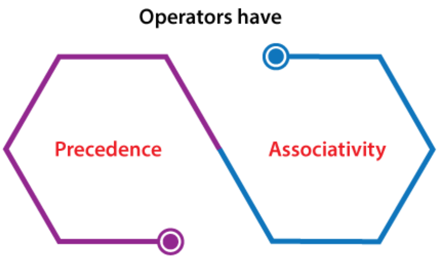
- **Constant expressions**: Constant Expressions consists of only constant values. A constant value is one that doesn't change.
  Examples:5, 10 + 5 / 6.0, 'x'
- **Integral expressions**: Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.
  Examples:x, x * y, x + int(5.0)
  where x and y are integer variables.
- **Floating expressions**: Float Expressions are which produce floating point results after implementing all the automatic and explicit type conversions.
  Examples:x + y, 10.75
  where x and y are floating point variables.
- **Relational expressions**: Relational Expressions yield results of type bool which takes a value true or false. When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions.
  Examples:x <= y, x + y > 2
- **Logical expressions**: Logical Expressions combine two or more relational expressions and produces bool type results.
  Examples:x > y && x == 10, x == 10 || y == 5
- **Pointer expressions**: Pointer Expressions produce address values.
  Examples:&x, ptr, ptr++
  where x is a variable and ptr is a pointer.
- **Bitwise expressions**: Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.
  Examples:
  x << 3

  shifts three bit position to left
  y >> 1
  shifts one bit position to right.
  Shift operators are often used for multiplication and division by powers of two.

*****

Precedence rules and Associativity:

When we talk about precedence in Java, the operator comes first in mind. There are certain rules defined in Java to specify the order in which the operators in an expression are evaluated. **Operator precedence** is a concept of determining the group of terms in an expression. The operator precedence is responsible for evaluating the expressions. In Java, **parentheses()** and **Array subscript[]** have the highest precedence in Java. For example, Addition and Subtraction have higher precedence than the Left shift and Right shift operators.



Below is a table defined in which the lowest precedence operator show at the top of it.

Precedence	Operator	Type	Associativity
1)	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left
2)	? :	Ternary conditional	Right to left
3)	||	Logical OR	Left to right

4)	&&	Logical AND	Left to right
5)	|	Bitwise inclusive OR	Left to right
6)	^	Bitwise exclusive OR	Left to right
7)	&	Bitwise AND	Left to right
8)	!= ==	Relational is not equal to Relational is equal to	Left to right
9)	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
10)	>> << >>>	Bitwise right shift with sign extension Bitwise left shift Bitwise right shift with zero	Left to right

		extension	
11)	- +	Subtraction Addition	Left to right
12)	* / %	Multiplication Division Modulus	Left to right
13)	- + ~ ! (type)	Unary minus Unary plus Unary bitwise complement Unary logical negation Unary typecast	Right to left
14)	++ --	Unary post-increment Unary post-decrement	Right to left
15)	. () []	Dot operator Parentheses Array subscript	Left to Right

Precedence order:

When two operators share a single operand, the operator having the highest precedence goes first. For example, x + y * z is treated as x + (y * z), whereas x * y + z is treated as (x * y) + z because * operator has highest precedence in comparison of + operator.

Associativity:

Associative is a concept related to the operators applied when two operators with the same precedence come in an expression. The associativity concept is very helpful to goes from that situation. Suppose we have an expression a + b - c **(+ and - operators have the same priority)**, and this expression will be treated as **(a + (b - c))** because these operators are right to left-associative. On the other hand, a+++--b+c++ will be treated as **((a++)+((--b)+(c++)))** because the unary post-increment and decrement operators are right to left-associative.

An example is defined below to understand how an expression is evaluated using precedence order and associativity?

Expression: x = 4 / 2 + 8 * 4 - (5+ 2) % 3

Solution:

1) In the above expression, the highest precedence operator is **()**. So, the parenthesis goes first and calculates first.

x = 4 / 2 + 8 * 4 - 7 % 3

2) Now, **/, *** and **%** operators have the same precedence and highest from the **+** and **-** Here, we use the associativity concept to solve them. The associative of these operators are from left to right. So, **/** operator goes first and then ***** and **%** simultaneously.

x = 2 + 8 * 4 - 7 % 3
x = 2 + 32 - 7 % 3
x = 2 + 32 - 1

3) Now, **+** and **-** operators both also have the same precedence, and the associativity of these operators lest to the right. So, **+** operator will go first, and then **-** will go.

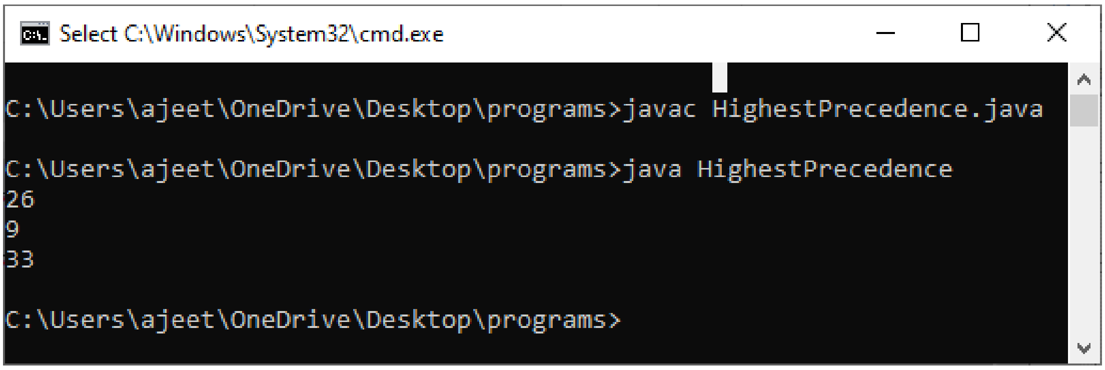
x = 34 - 1
x = 33

HighestPrecedence.java

```
//import classes
import java.util.*;
//creating HighestPrecedence class to evaluate the expression
public class HighestPrecedence {
  //main() method starts
   public static void main(String[] args) {
  //initialize variables with default values
      int x = 2;
      int y = 5;
      int z = 12;
     //calculating exp1, exp2, and exp3
      int exp1 = x +(z/x+(z%y)*(z-x)^2);
      int exp2 = z/x+y*x-(y+x)%z;
      int exp3 = 4/2+8*4-(5+2)%3;
      //printing the result
      System.out.println(exp1);
      System.out.println(exp2);
      System.out.println(exp3);
```

} }

Output:

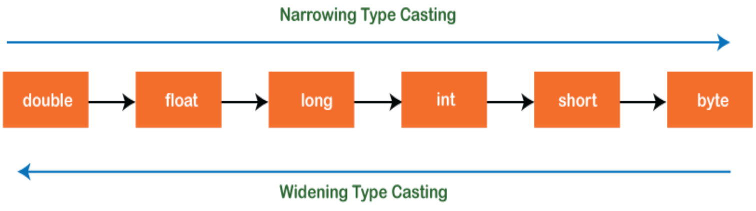


```
C:\Users\ajeet\OneDrive\Desktop\programs>javac HighestPrecedence.java

C:\Users\ajeet\OneDrive\Desktop\programs>java HighestPrecedence
26
9
33

C:\Users\ajeet\OneDrive\Desktop\programs>
```

*****

Primitive Type conversion and Casting:

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



Type Casting in Java

Type casting:

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting:

There are two types of type casting:

1.Widening Type Casting

2.Narrowing Type Casting

1.Widening Type Casting:-

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

byte -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output:

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

2.Narrowing Type Casting:-

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

*****

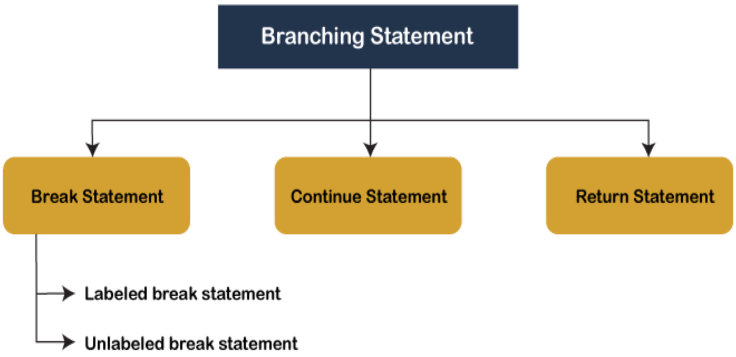
Flow of Control:-

Java compiler executes the java code from top to bottom. The statements are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of java code. Such statements are called control flow statements.

1.Branching:

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

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



The break Statement:

The **labeled** and **unlabeled** break statement are the two forms of break statement in Java. The break statement is used for terminating a loop based on a certain condition. Let's understand each form of **break** statement one by one with their examples.

i) Unlabeled break statement: 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:

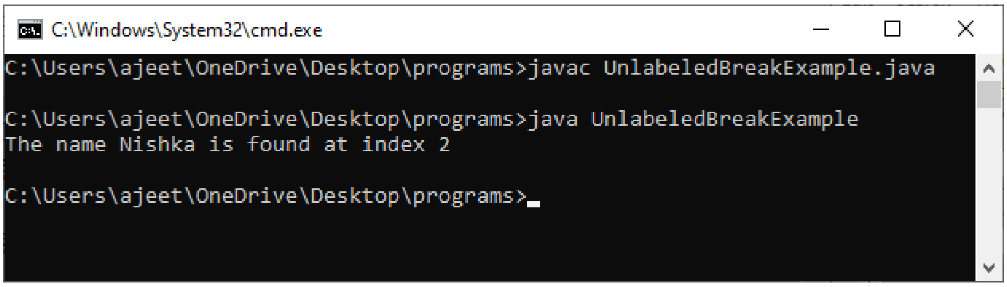
```
for (int; testExpression; update){
   //Code
   if(condition to break){
      break;
   }  }
```

Let's take an example to understand how the **unlabeled break** statement works to terminate the loop.

UnlabeledBreakExample.java

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

 String[] arr = { "Shubham", "Anubhav", "Nishka", "Gunjan", "Akash" };
 String searchName = "Nishka";
      int j;
      boolean foundName = false;
      for (j = 0; j < arr.length; j++) {
      if (arr[j] == searchName) {
      foundName = true;
         break;
          }  }

       if (foundName) {
          System.out.println("The name " + searchName + " is found at index " + j);
       } else {
        System.out.println("The name " +searchName + " is not found in the array");
    }  }  }
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac UnlabeledBreakExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UnlabeledBreakExample
The name Nishka is found at index 2

C:\Users\ajeet\OneDrive\Desktop\programs>
```

ii) Labeled break statement: Labeled break statement is another form of break statement. If we have a nested loop in Java and use the break statement in the innermost loop, then it will terminate the innermost loop only, not the outermost loop. The labeled break statement is capable of terminating the outermost loop.

Syntax:

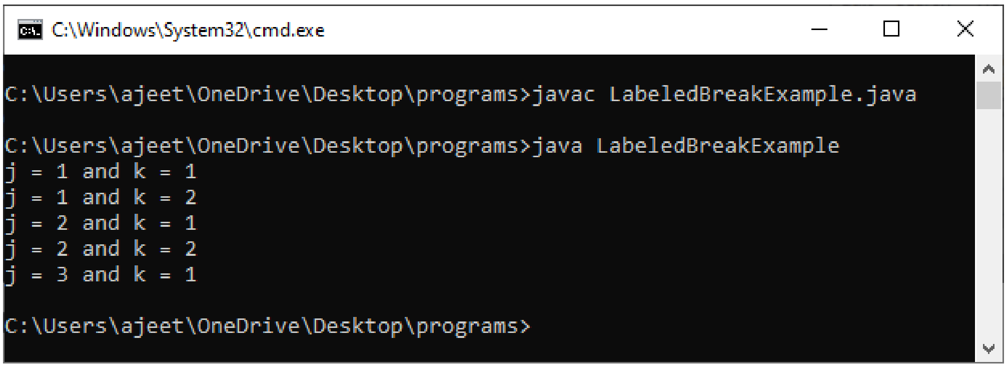
```
label:
for (int; testExpression; update){
  //Code
  for (int; testExpression; update){
   //Code
   if(condition to break){
   break label;
     }  }  }
```

Let's take an example to understand how the **labeled break** statement works to terminate the loop.

LabeledBreakExample.java

```
class LabeledBreakExample {
 public static void main(String[] args) {
         int j, k;
 // Labeling the outermost loop as outerMost
        outerMost:
       for(j=1; j<5; j++) {
   // Labeling the innermost loop as innerMost
      innerMost:
       for(k=1; k<3; k++ ) {
       System.out.println("j = " + j + " and k = " +k);
     // Terminating the outemost loop
        if ( j == 3)
        break outerMost; }  }  }  }
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac LabeledBreakExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java LabeledBreakExample
j = 1 and k = 1
j = 1 and k = 2
j = 2 and k = 1
j = 2 and k = 2
j = 3 and k = 1

C:\Users\ajeet\OneDrive\Desktop\programs>
```

The continue Statement:

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

When we have a nested for loop, and we use the continue statement in the innermost loop, it continues only the innermost loop. We can use the continue statement for any control flow statements like **for, while**, and **do-while**.

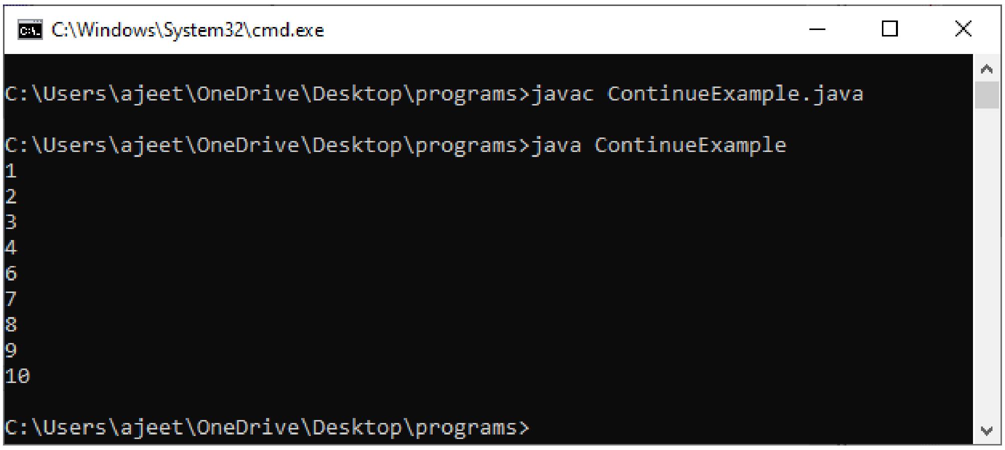
Syntax:

control-flow-statement;
continue;

ContinueExample.java

```
public class ContinueExample {
public static void main(String[] args) {
//Declare variables
int x = 1;
 int y = 10;
 //Using do while loop for using contiue statement
 do{
 if(x == y/2){
  x++;
    continue;//The continue statement skips the remaining statement
     }
    System.out.println(x);
     x++;
     }while(x <= y);    }    }
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac ContinueExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java ContinueExample
1
2
3
4
6
7
8
9
10

C:\Users\ajeet\OneDrive\Desktop\programs>
```

The return Statement:

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

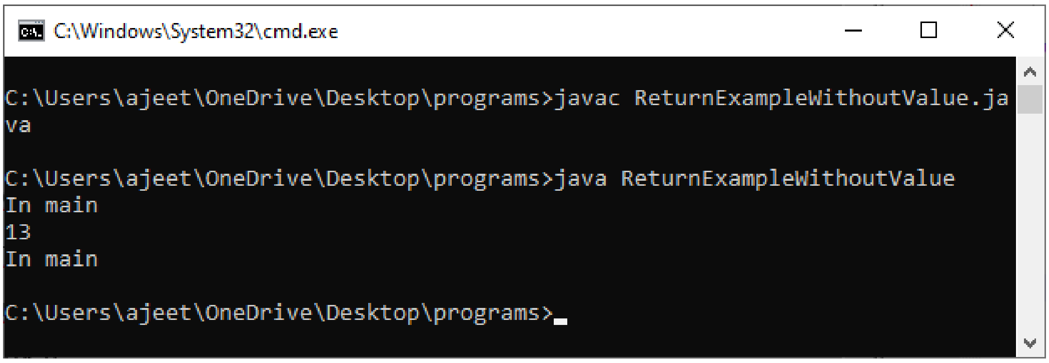
Syntax:

return value;
Or,
return;

ReturnExampleWithoutValue.java

```
class ReturnExampleWithoutValue {
 //Declare calling method
 void increment(int number)
 {
  if (number < 10)
  return; //pass the control flow to where this method call
  number++;
  System.out.println(number);
    }
    public static void main(String[] args)
    {
        ReturnExampleWithoutValue obj = new ReturnExampleWithoutValue();
        obj.increment(4);
        System.out.println("In main");
        obj.increment(12);
        System.out.println("In main");    }   }
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac ReturnExampleWithoutValue.ja
va

C:\Users\ajeet\OneDrive\Desktop\programs>java ReturnExampleWithoutValue
In main
13
In main

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

*

Conditional:

Decision-making statements evaluate the Boolean expression and control the program flow depending upon the condition result. There are two types of decision-making statements in java, I.e., If statement and switch statement.

If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the condition result that is a Boolean value, either true or false. In java, there are four types of if-statements given below.

1. if statement
2. if-else statement
3. else-if statement
4. Nested if-statement

1. if statement:

This is the most basic statement among all control flow statements in java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(<condition>) {
//block of code
}
```

Consider the following example in which we have used the **if** statement in the java code.

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than       20");
} } }
```

Output:

```
x + y is greater than 20
```

2. if-else statement:

The if-else statement is an extension to the if-statement, which uses another block of code, I.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Consider the following example.

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
```

Output:

```
x + y is greater than 20
```

3. else-if statement:

The else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter any block of code. We can also define an else statement at the end of the chain.

Consider the following example.

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
} } }
```

Output:

```
Delhi
```

4. Nested if-statement:

In nested if-statements, the if statement contains multiple if-else statements as a separate block of code. Consider the following example.

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in india");
}  }  }
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement enables us to check the variable for the range of values defined for multiple case statements. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program. The syntax to use the switch statement is given below.

```
switch <variable> {
Case <option 1>:
//block of statements
..
..
..
Case <option n>:
//block of statements
Default:
//block of statements
}
```

Consider the following example to understand the flow of the switch statement.

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}  }  }
```

Output: 2

*

Loops:

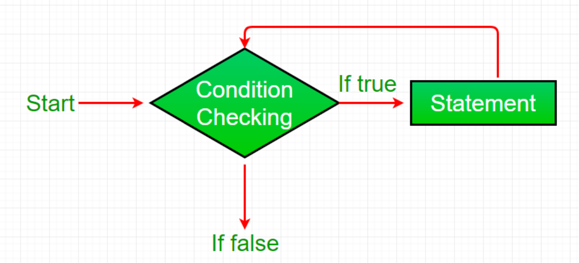
Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

i)while loop:A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax :while (boolean condition)

```
    {

     loop statements...

    }
```

Flowchart:



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

```
// Java program to illustrate while loop
classwhileLoopDemo{
 publicstaticvoidmain(String args[]) {
  intx = 1;
  // Exit when x becomes greater than 4
    while(x <= 4)  {
    System.out.println("Value of x:"+ x);
    // Increment the value of x for
     // next iteration
       x++;
        } }}
```

Output: Value of x:1
Value of x:2

Value of x:3

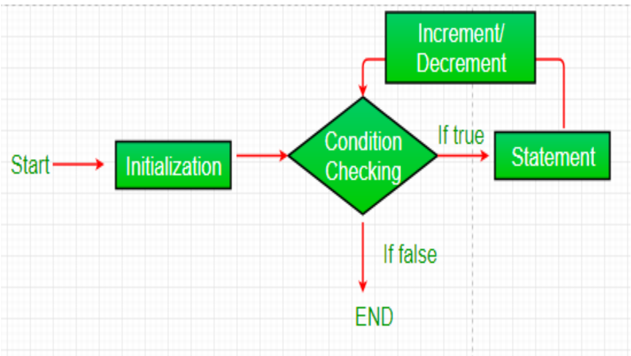
Value of x:4

ii)for loop: for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

Syntax:for (initialization condition; testing condition;
increment/decrement)

```
{
 statement(s)
}
```

Flowchart:



Initialization condition: Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
Testing Condition: It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.

Statement execution: Once the condition is evaluated to true, the statements in the loop body are executed.
Increment/ Decrement: It is used for updating the variable for next iteration.
Loop termination: When the condition becomes false, the loop terminates marking the end of its life cycle.

```
// Java program to illustrate for loop.
classforLoopDemo{
 publicstaticvoidmain(String args[]){
  // for loop begins when x=2
  // and runs till x <=4
   for(intx = 2; x <= 4; x++)
   System.out.println("Value of x:"+ x);
    }}
```

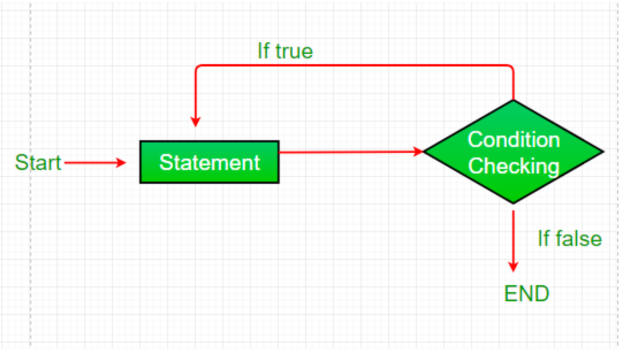
Output:
```
Value of x:2
Value of x:3

Value of x:4
```

iii)do while: do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop.**

Syntax:

```
do
{
statements..
}
while (condition);
```

Flowchart:



- do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.

- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

```
// Java program to illustrate do-while loop
classdowhileloopDemo{
 publicstaticvoidmain(String args[]){
  intx = 21;
    do
    {
          // The line will be printed even
          // if the condition is false
    System.out.println("Value of x:"+ x);
     x++;
    }
    while(x < 20);}}
```

Output: Value of x: 21

*****

Classes and Objects:

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

Classes:

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

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

Objects:

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

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.

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

Example of an object: dog

*****

Creating Objects:

The **object** is a basic building block of an OOPs language. In **Java**, we cannot execute any program without creating an **object**. There is various way to create an object in Java that we

Java provides five ways to create an object.

1.Using **new** Keyword
2.Using **clone()** method
3.Using **newInstance()** method of the **Class** class
4.Using **newInstance()** method of the **Constructor** class
5.Using **Deserialization**

1.Using new Keyword: Using the **new** keyword is the most popular way to create an object or instance of the class. When we create an instance of the class by using the new keyword, it allocates memory (heap) for the newly created **object** and also returns the reference of that object to that memory. The new keyword is also used to create an array.

The syntax for creating an object is:

ClassName object = **new** ClassName();

Let's create a program that uses new keyword to create an object.

CreateObjectExample.java

```
public class CreateObjectExample1
{
void show()
{
System.out.println("Welcome to java");
}
public static void main(String[] args)
{
//creating an object using new keyword
CreateObjectExample1 obj = new CreateObjectExample1();
//invoking method using the object
obj.show();
}
}
```

Output: Welcome to java

2.Using clone() Method: The **clone()** method is the method of **Object** class. It creates a copy of an object and returns the same copy. The JVM creates a new object when the clone() method is invoked. It copies all the content of the previously created object into new one object. Note that it does not call any constructor. We must implement the **Cloneable** interface while using the clone() method. The method throws **CloneNotSupportedException** exception if the object's class does not support the Cloneable interface. The subclasses that override the clone() method can throw an exception if an instance cannot be cloned.

Syntax:

```
protected Object clone() throws CloneNotSupportedException
```

We use the following statement to create a new object.

```
ClassName newobject = (ClassName) oldobject.clone();
```

CreateObjectExample.java

```
public class CreateObjectExample implements Cloneable
{
@Override
protected Object clone() throws CloneNotSupportedException
{
//invokes the clone() method of the super class
return super.clone();
}
String str = "New Object Created";
public static void main(String[] args)
{
//creating an object of the class
CreateObjectExample3 obj1 = new CreateObjectExample3();
//try catch block to catch the exception thrown by the method
try
{
//creating a new object of the obj1 suing the clone() method
CreateObjectExample3 obj2 = (CreateObjectExample3) obj1.clone();
System.out.println(obj2.str);
}
catch (CloneNotSupportedException e)
{
e.printStackTrace();   }   }   }
```

Output: New Object Created

3.Using newInstance() Method of Class class: The **newInstance()** method of the Class class is also used to create an object. It calls the default constructor to create the object. It returns a newly created instance of the class represented by the object. It internally uses the newInstance() method of the Constructor class.

Syntax:

```
public T newInstance() throws InstantiationException, IllegalAccessException
```

It throws the **IllegalAccessException, InstantiationException, ExceptionInInitializerError** exceptions.

In the following program, we have creates a new object using the newInstance() method.

CreateObjectExample.java

```
public class CreateObjectExample4
{
void show()
{
System.out.println("A new object created.");
}
public static void main(String[] args)
{
try
{
//creating an instance of Class class
Class cls = Class.forName("CreateObjectExample4");
//creates an instance of the class using the newInstance() method
CreateObjectExample4 obj = (CreateObjectExample4) cls.newInstance();
//invoking the show() method
obj.show();
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
catch (InstantiationException e)
{
e.printStackTrace();
}
catch (IllegalAccessException e)
{
e.printStackTrace();   }   }   }
```

Output:A new object created.

4.Using newInstance() Method of Constructor class:

It is similar to the **newInstance()** method of the **Class** class. It is known as a reflective way to create objects. The method is defined in the **Constructor** class which is the class of java.lang.reflect package. We can also call the parameterized constructor and private constructor by using the **newInstance()** method. It is widely preferred in comparison to newInstance() method of the Class class.

Syntax:

public T newInstance(Object... initargs) **throws** InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException

We can create an object in the following way:

1. Constructor<Employee> constructor = Employee.**class**.getConstructor();
2. Employee emp3 = constructor.newInstance();

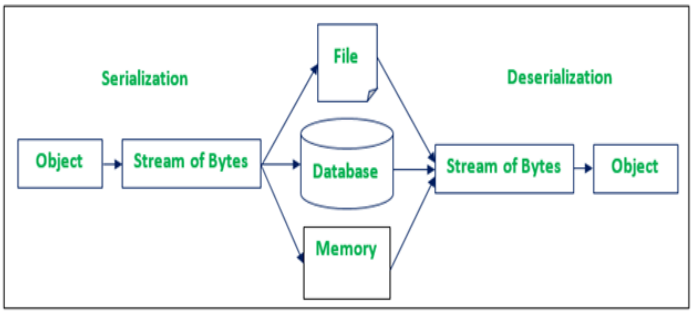
Let's create a program that creates an object using the newInstance() method.

CreateObjectExample.java

```
import java.lang.reflect.*;
public class CreateObjectExample
{
private String str;
CreateObjectExample5()
{
}
public void setName(String str)
{
this.str = str;
}
public static void main(String[] args)
{
try
{
Constructor<CreateObjectExample5> constructor = CreateObjectExample5.class.getDeclaredConstructor();
CreateObjectExample5 r = constructor.newInstance();
r.setName("Java");
System.out.println(r.str);
}
catch (Exception e)
{
e.printStackTrace();   }   }   }
```

Output: Java

5.Using Deserialization: In Java, **serialization** is the process of converting an object into a sequence of byte-stream. The reverse process (byte-stream to object) of serialization is called **deserialization**. The JVM creates a new object when we serialize or deserialize an object. It does not use constructor to create an object. While using deserialization, the **Serializable** interface (marker interface) must be implemented in the class.



Serialization: The **writeObject()** method of the **ObjectOutputStream** class is used to serialize an object. It sends the object to the output stream.

Syntax:

public final void writeObject(object x) **throws** IOException

Deserialization: The method **readObject()** of **ObjectInputStream** class is used to deserialize an object. It references objects out of a stream.

Syntax:

public final Object readObject() **throws** IOException,ClassNotFoundException

Let's understand the serialization and deserialization through a program.

Employee.java

```
import java.io.Serializable;
public class Employee implements Serializable
{
int empid;
String empname;
public Empoyee(int empid, String empname)   {
this.empid = empid;
this.empname = empname;    }    }
```

*****

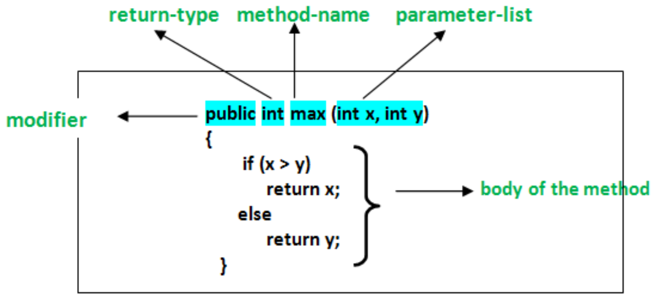
Methods:

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++, and Python. Methods are time savers and help us to reuse the code without retyping the code.

Method Declaration:

In general, method declarations has six components :

1.Modifier:Defines access type of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
(i)public: accessible in all class in your application.
(ii)protected:accessible within the class in which it is defined and in its **subclass(es)**
(iii)private: accessible only within the class in which it is defined.
(iv)default: (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.
2.The return type: The data type of the value returned by the method or void if does not return a value.
3.Method Name: the rules for field names apply to method names as well, but the convention is a little different.
4.Parameter list: Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
5.Exception list: The exceptions you expect by the method can throw, you can specify these exception(s).
6.Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations.



```
public int max (int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
 }
```

Calling a method:

The method needs to be called for using its functionality. There can be three situations when a method is called:
A method returns to the code that invoked it when:

- It completes all the statements in the method
- It reaches a return statement
- Throws an exception

// Program to illustrate methodsin java
import java.io.*;

```
class Addition {
  int sum = 0;
  public int addTwoInt(int a, int b){
   // adding two integer value.
    sum = a + b;
   //returning summation of two values.
     return sum;   }}
  class GFG {
  public static void main (String[] args) {
   // creating an instance of Addition class
   Addition add = new Addition();
   // calling addTwoInt() method to add two integer using instance created
   // in above step.
   int s = add.addTwoInt(1,2);
  System.out.println("Sum of two integer values :"+ s);  } }
```

Output :

```
Sum of two integer values :3
```

*****

Constructors:

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

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java constructor:

There are two rules defined for the constructor.

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

Types of Java constructors:

There are two types of constructors in Java:

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

1.Default constructor (no-arg constructor):

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

Syntax of default constructor:<class_name>(){}

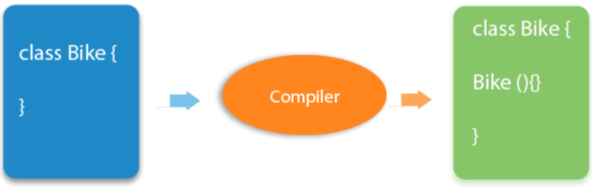
Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

```
Bike is created
```



2.Parameterized Constructor:

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

Why use the parameterized constructor?

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

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

//Java Program to demonstrate the use of the parameterized constructor.

```
class Student4{
 int id;
 String name;
//creating a parameterized constructor
 Student(int i,String n){
 id = i;
  name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}
     public static void main(String args[]){
    //creating objects and passing values
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    //calling method to display the values of object
    s1.display();
 s2.display();
   }
}
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading:

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

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

Example of Constructor Overloading

```
//Java program to overload constructors
class Student{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student(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[]){
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan 0
222 Aryan 25
```

*****

Cleaning up unused objects:

Many other object-oriented languages require that you keep track of all the objects you create and that you destroy them when they are no longer needed. Writing code to manage memory in this way is tedious and often error-prone. Java saves you from this by allowing you to create as many objects as you want (limited of course to whatever your system can handle) but never having to destroy them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is known as garbage collection.

Garbage Collector:

In java, garbage means unreferenced objects.
Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

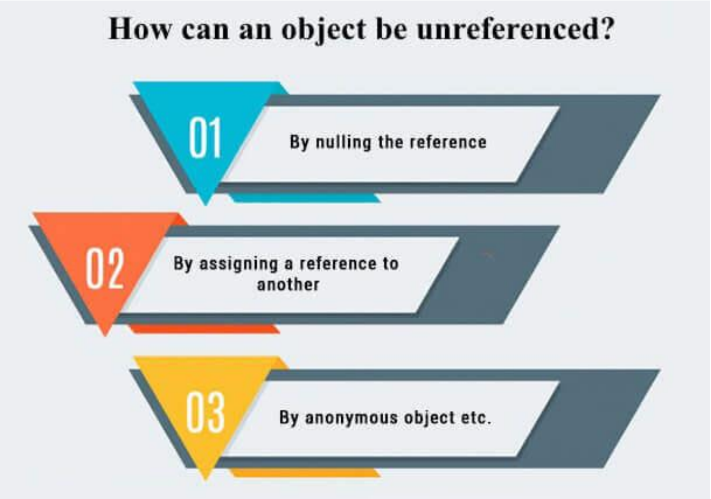
Advantage of Garbage Collection

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

How can an object be unreferenced?

There are many ways:

1.By nulling the reference
2.By assigning a reference to another
3.By anonymous object etc.


How can an object be unreferenced?
01
By nulling the reference
02
By assigning a reference to another
03
By anonymous object etc.

1) By nulling a reference:

```
     Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

finalize() method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}
```

gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

Simple Example of garbage collection in java

```
public class TestGarbage1{
public void finalize(){System.out.println("object is garbage collected");}
public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();  }  }
```

Output:

```
object is garbage collected
object is garbage collected
```

*****

Class Variable:

A variable provides us with named storage that our programs can manipulate. Java provides three types of variables.

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

2.Instance variables: Instance variables are declared in a class, but outside a method. When space is allocated for an object in the heap, a slot for each instance variable value is created. Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.

3.Local variables:Local variables are declared in methods, constructors, or blocks. Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.

Example:

```
publicclassVariableExample{
int myVariable;
staticint data =30;
publicstaticvoid main(String args[]){
int a =100;
VariableExample obj =newVariableExample();
System.out.println("Value of instance variable myVariable: "+obj.myVariable);
System.out.println("Value of static variable data: "+VariableExample.data);
System.out.println("Value of local variable a: "+a);
}
}
```

Output:

```
Value of instance variable myVariable: 0
Value of static variable data: 30
Value of local variable a: 100
```

Class Methods:

Class methods are methods that are called on the class itself, not on a specific object instance. The static modifier ensures implementation is the same across all class instances. Many standard built-in classes in Java (for example, Math) come with static methods (for example, Math.abs(int value)) that are used in many Java programs.

Syntax:

```
publicclassclassName{
modifier static dataType methodName(inputParameters){ //static method
//block of code to be executed
   }
}
//calling the method, from anywhere
className.methodName(passedParams);
```

*****

Static Keyword:

Static is a non-access modifier in Java which is applicable for the following:

1. blocks
2. variables
3. methods
4. nested classes

To create a static member(block,variable,method,nested class), precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. For example, in below java program, we are accessing static method m1() without creating any object of Test class.

1.Static blocks: If you need to do computation in order to initialize your static variables**,** you can declare a static block that gets executed exactly once, when the class is first loaded. Consider the following java program demonstrating use of static blocks.

2.Static variables: When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables :-

- We can create static variables at class-level only.
- static block and static variables are executed in order they are present in a program.

3.Static methods: When a method is declared with *static* keyword, it is known as static method. The most common example of a static method is *main( )* method.As discussed above, Any static member can be accessed before any objects of its class are created, and without reference to any object.Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.

4.Nested Classes in Java: In Java, it is possible to define a class within another class, such classes are known as nested classes. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, and creates more readable and maintainable code.

- The scope of a nested class is bounded by the scope of its enclosing class. A nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e., the enclosing class does not have access to the members of the nested class.

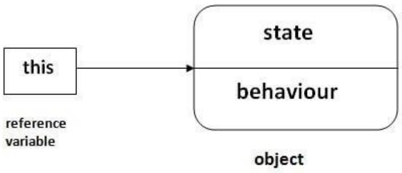
- A nested class is also a member of its enclosing class.
- Nested classes are divided into two categories:
  1. **static nested class:** Nested classes that are declared *static* are called static nested classes.
  2. **inner class :** An inner class is a non-static nested class.

*****

this keyword:

The this keyword refers to the current object in a method or constructor.

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



Usage of java this keyword:

Here is given the 6 usage of java this keyword.

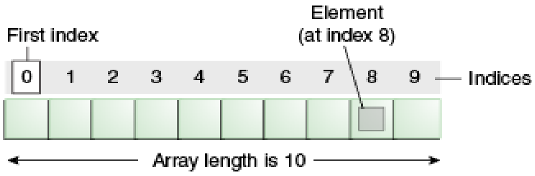
1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.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

*****

Arrays:

An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

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



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.Single Dimensional Array
2.Multidimensional Array

1.Single Dimensional Array in Java

Syntax to Declare an Array in Java

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

Instantiation of an Array in Java : arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an 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;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
50
```

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[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//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();
}
}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

*****

Command Line Arguments:

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}
}
```

```
Output: Your first argument is: sonoo
```

*****

INHERITANCE

Inheritance:

Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.

Important terminology:

1.Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).

2.Sub Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

3.Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is **extends**.

Syntax :

```
class derived-class extends base-class
{
  //methods and fields
}
```

Example: In the below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class that extends Bicycle class and class Test is a driver class to run program.

```
// Java program to illustrate the
// concept of inheritance
// base class
class Bicycle {
 // the Bicycle class has two fields
    public int gear;
    public int speed;
    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed){
     this.gear = gear;
     this.speed = speed;  }
    // the Bicycle class has three methods
    public void applyBrake(int decrement){
       speed -= decrement;  }
    public void speedUp(int increment)
    {
       speed += increment;
    }
    // toString() method to print info of Bicycle
    public String toString()
    {
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }}
// derived class
class MountainBike extends Bicycle {
    // the MountainBike subclass adds one more field
    public int seatHeight;
    // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }
    // the MountainBike subclass adds one more method
    public void setHeight(int newValue){
        seatHeight = newValue;  }
    // overriding toString() method
    // of Bicycle to print more info
    @Override public String toString()
    {
        return (super.toString() + "\nseat height is "
                + seatHeight);
    }}
// driver class
public class Test {
    public static void main(String args[])
    {
        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }}
```

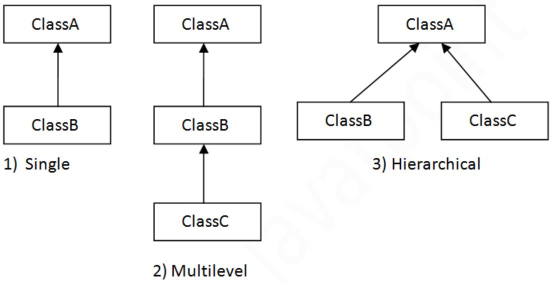
Output:

```
No of gears are 3
speed of bicycle is 100

seat height is 25
```

*****

Types of inheritance

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

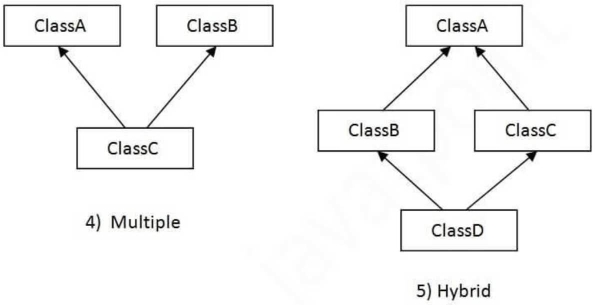


1) Single

2) Multilevel

3) Hierarchical

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple

5) Hybrid

1.Single Inheritance:

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
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();
}}
```

Output:

```
barking...
eating...
```

2.Multilevel Inheritance:

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
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();
}}
```

Output:

```
weeping...
barking...
eating...
```

3.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();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

4.Multiple inheritance:

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
      }
}
```

Output:

```
Compile Time Error
```

*****

Deriving classes using extends keyword:

In Java, the extends keyword is used to indicate that a new class is derived from the base class using inheritance. So basically, extends keyword is used to extend the functionality of the class.

A program that demonstrates the extends keyword in Java is given as follows:

Example:

```
class A {
  int a =9;
}
class B extends A {
  int b =4;
}
publicclassDemo{
  publicstaticvoid main(String args[]){
    B obj =new B();
    System.out.println("Value of a is: "+ obj.a);
    System.out.println("Value of b is: "+ obj.b);
  }
}
```

Output

```
Value of a is: 9
Value of b is: 4
```

Now let us understand the above program.

The class A contains a data member a. The class B uses the extends keyword to derive from class A. It also contains a data member b. A code snippet which demonstrates this is as follows:

```
class A {
  int a =9;
}
class B extends A {
  int b =4;  }
```

In the main() method in class Demo, an object obj of class B is created. Then the values of a and b are printed. A code snippet which demonstrates this is as follows:

```
publicclassDemo{
  publicstaticvoid main(String args[]){
    B obj =new B();
    System.out.println("Value of a is: "+ obj.a);
    System.out.println("Value of b is: "+ obj.b);   } }
```

*****

Method Overloading:

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading:

Method overloading increases the readabilityof the program.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments: In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments: In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

*****

Super Keyword:

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

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

Usage of Java super Keyword

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

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

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

```
Class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}  }
class TestSuper1{
```

```
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method:

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

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

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor:

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

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

Output:

```
animal is created
dog is created
```

*****

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

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

Java Final Keyword
⇨ Stop Value Change
⇨ Stop Method Overridding
⇨ Stop Inheritance
javatpoint.com

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

Example of final variable

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

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

Output:

Compile time error

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

*****

Abstract Class:

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.

- 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 abstract class:-

```
abstract class A{}
```

Abstract Method:

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

Example of abstract method:-

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method:-

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

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

Output:

```
running safely
```

*****

UNIT-II

Interfaces, Packages and Enumeration: Interface-Extending interface, Interface Vs Abstract classes, Packages-Creating packages, using Packages, Access protection, java.lang package. **Exceptions & Assertions** – Introduction, Exception handling techniques- try… catch, throw, throws, finally block, user defined exception, Exception Encapsulation and Enrichment, Assertions.

INTERFACES,PACKAGES AND ENUMERATION

Interface:

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Syntax :

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

Extending Interface:

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.

The following Sports interface is extended by Hockey and Football interfaces.

Example:

```
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

*****

Interface Vs Abstract Classes:

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

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

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

public abstract class Shape{
public abstract void draw();
}
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

*****

Packages:

PACKAGE in Java is a collection of classes, sub-packages, and interfaces. It helps organize your classes into a folder structure and make it easy to locate and use them. More importantly, it helps improve code reusability.

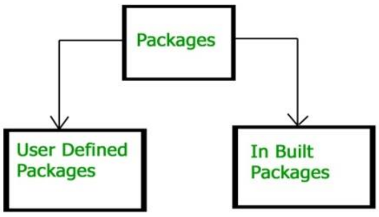
Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group.

Although interfaces and classes with the same name cannot appear in the same package, they can appear in different packages. This is possible by assigning a separate namespace to each Java package.

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.

Types of packages:



Built-in Packages:

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

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

User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

Simple example of java package

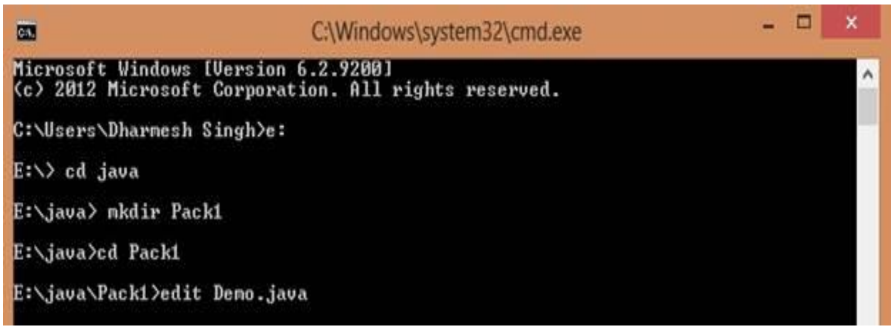
The package keyword is used to create a package in java.

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

Creating Packages:

Creating a package is a simple task as follows

1. First create a directory within name of package.
2. Create a java file in newly created directory.
3. In this java file you must specify the package name with the help of package keyword.
4. Save this file with same name of public class
5. Now you can use this package in your program.



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\Dharmesh Singh>e:

E:\> cd java

E:\java> mkdir Pack1

E:\java>cd Pack1

E:\java\Pack1>edit Demo.java
```

```
Package Pack1
public class Demo  {
 public void Show()   {
System.out.print("Package called");
}  }
```

After that use this package in your program,

```
import Pack1.*
class A    {
 public static void main(String …args)    {
  Demo ob1= new demo();
ob1.Show();
}    }
```

Using Packages:

The types that comprise a package are known as the *package members*.

To use a public package member from outside its package, you must do one of the following:

- Refer to the member by its fully qualified name
- Import the package member
- Import the member's entire package

Each is appropriate for different situations, as explained in the sections that follow.

Referring to a Package Member by Its Qualified Name

You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported.

However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. Here is the fully qualified name for the Rectangle class declared in the graphics package in the previous example.

```
graphics.Rectangle
```

You could use this qualified name to create an instance of graphics.Rectangle:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Importing a Package Member:-

To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one. Here's how you would import the Rectangle class from the graphics package created in the previous section.

```
import graphics.Rectangle;
```

Now you can refer to the Rectangle class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the graphics package. But if you use many types from a package, you should import the entire package.

Importing an Entire Package

To import all the types contained in a particular package, use the import statement with the asterisk (*) wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the graphics package by its simple name.

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the graphics package that begin with A.

```
// does not work
import graphics.A*;
```

Instead, it generates a compiler error. With the import statement, you generally import only a single package member or an entire package.

*****

Access protection:

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

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.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

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

Access Modifier	within	within	outside package by subclass	outside

	class	package	only	package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

*****

java.lang package:

Provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Following are the Important Classes in Java.lang package :

1. **Boolean**: The Boolean class wraps a value of the primitive type boolean in an object.
2. **Byte**: The Byte class wraps a value of primitive type byte in an object.
3. **Character -Set 1**, **Set 2:** The Character class wraps a value of the primitive type char in an object.
4. **Character.Subset**: Instances of this class represent particular subsets of the Unicode character set.
5. **Class – Set 1, Set 2 :** Instances of the class Class represent classes and interfaces in a running Java application.
6. **ClassLoader:** A class loader is an object that is responsible for loading classes.
7. **ClassValue:** Lazily associate a computed value with (potentially) every type.
8. **Compiler**: The Compiler class is provided to support Java-to-native-code compilers and related services.
9. **Double**: The Double class wraps a value of the primitive type double in an object.
10. **Enum**: This is the common base class of all Java language enumeration types.
11. **Float:** The Float class wraps a value of primitive type float in an object.
12. **Integer**:The Integer class wraps a value of the primitive type int in an object.
13. **Long**: The Long class wraps a value of the primitive type long in an object.
14. **Math – Set 1**, **Set 2**: The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
15. **Number:** The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.
16. **Object**: Class Object is the root of the class hierarchy.
17. **Package**: Package objects contain version information about the implementation and specification of a Java package.

*****

Exceptions & Assertions

Introduction:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

1.Checked exceptions :A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.

2.Unchecked exceptions : An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Exception handling Techniques:

An exception is an "unwanted or unexpected event", which occurs during the execution of the program i.e, at run-time, that disrupts the normal flow of the program's instructions. When an exception occurs, execution of the program gets terminated.

Blocks & Keywords used for exception handling

1.try: The try block contains set of statements where an exception can occur.

```
try
{
   // statement(s) that might cause exception
}
```

2.catch : Catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

catch

```
{
   // statement(s) that handle an exception
   // examples, closing a connection, closing
   // file, exiting the process after writing
   // details to a log file.
}
```

3.**throw**: Throw keyword is used to transfer control from try block to catch block.
4.**throws**: Throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.

5.**finally**: It is executed after catch block. We basically use it to put some common code when there are multiple catch blocks.

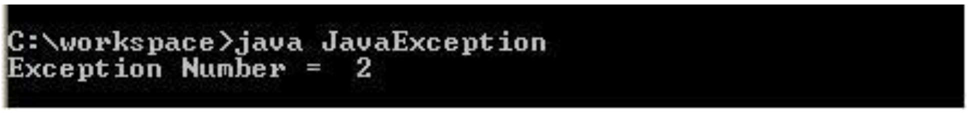
*****

User defined exception:

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as **user-defined** or **custom** exceptions.

```
classJavaException{
   public static void main(String args[]){
  try{
       throw new MyException(2);
       // throw is used to create a new exception and throw it.
  }
 catch(MyException e){
    System.out.println(e) ;
 }
}
}
class MyException extends Exception{
   int a;
   MyException(int b) {
     a=b;
   }
public String toString(){
return ("Exception Number =  "+a) ;
  }
}
```

Output:



```
C:\workspace>java JavaException
Exception Number =  2
```

*****

Exception encapsulation:

Exceptions in Java are still very common in real work. When to throw an exception, this must be known.

Of course, the exceptions that are actively thrown in the real work are all packaged, and you can define the error code and exception description yourself.

Checked exception:

Indicates invalid, not predictable in the program. For example, invalid user input, the file does not exist, and the network or database link is incorrect. These are all external reasons, none of which can be controlled within the program.

Unchecked exception:

Indicates an error, the logic of the program is wrong. It is a subclass of RuntimeException, such as IllegalArgumentException, NullPointerException and IllegalStateException.

Exception enrichment:

In exception enrichment you do not wrap exceptions. Instead you add contextual information to the original exception and rethrow it. Rethrowing an exception does not reset the stack trace embedded in the exception.

Here is an example:

```
public void method2() throws EnrichableException{
  try{
    method1();
  } catch(EnrichableException e){
    e.addInfo("An error occurred when trying to ...");
    throw e;
  }
}

public void method1() throws EnrichableException {
  if(...) throw new EnrichableException(
    "Original error message");
}
```

As you can see the method1() throws an EnrichableException which is a superclass for enrichable exceptions. This is not a standard Java exception, so you will have to create it yourself. There is an example EnrichableException at the end of this text.

Notice how method2() calls the addInfo() method on the caught EnrichableException, and rethrow it afterwards. As the exception propagates up the call stack, each catch block can add relevant information to the exception if necessary.

*****

Assertions:

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.

Advantage of Assertion:

It provides an effective way to detect and correct programming errors.

Syntax Assertion:

There are two ways to use assertion. First way is:

```
assert expression;

          (Or)

assert expression1 : expression2;
```

Simple Example of Assertion :

```
import java.util.Scanner;

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

  Scanner scanner = new Scanner( System.in );
  System.out.print("Enter ur age ");

  int value = scanner.nextInt();
  assert value>=18:" Not valid";

  System.out.println("value is "+value);
 }
}
```

Output:

```
Enter ur age:11
Exception in thread “main” java.lang.Assertion
Error: Not valid
```

*****

UNIT III

MultiThreading: java.lang.Thread, The main Thread, Creation of new threads, Thread priority, Multithreading- Using isAlive () and join (), Synchronization, suspending and Resuming threads, Communication between Threads Input/Output: reading and writing data, java.io package, **Applets**– Applet class, Applet structure, An Example Applet Program, Applet: Life Cycle, paint(), update() and repaint().

Multi Threading

java.lang.Thread:

The **java.lang.Thread** class is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.Following are the important points about Thread −

- Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority
- Each thread may or may not also be marked as a daemon.
- There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread and,
- the other way to create a thread is to declare a class that implements the Runnable interface

Class Declaration:

```
public class Thread
  extends Object
    implements Runnable
```

Field:

Following are the fields for **java.lang.Thread** class −

- **static int MAX_PRIORITY** − This is the maximum priority that a thread can have.
- **static int NORM_PRIORITY** − This is the default priority that is assigned to a thread.

*****

Main Thread:

As we know every java program has a **main method**. The **main method** is the entry point to execute the program. So, when the **JVM** starts the execution of a program, it creates a thread to run it and that thread is known as the **main thread.**

If you are creating a simple program to print **"Hello world"** it means JVM will create the **main thread** to execute it. You must saw if you try to run a Java program with compilation errors the JVM shows the error in the **main thread**.

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

Output:

```
Exception in thread “main” java.lang.Error: Unresolved compilation problems:     Syntax
error, insert “VariableDeclarators” to complete LocalVariableDeclaration a cannot be resolved at
ExampleOfThreadCreation.main(ExampleOfThreadCreation.java:5)
```

The **JVM** automatically creates the **main thread** but we can verify it and also control this thread. To control the **main thread,** we need to get the reference of the thread. We can get the refence of the main thread by calling the method **currentThread( )**, which is a **public static** member of **Thread**. This method will return the reference of the thread in which it is called. So when we call it in the **main thread** it will return the reference of the **main thread**. After that, you can control it just like any other thread. Let’s try it with an example

```
class ExampleOfThreadCreation
{
public static void main(String args[])
{
Thread obj = Thread.currentThread();
System.out.println("Name of thread :" +obj.getName());
System.out.println("Priority of thread :" +obj.getPriority());
// We can set the Name of main thread
obj.setName("Main thread");
System.out.println("Name of thread :" +obj.getName());
System.out.println("Hello world");
}
}
```

Output:

```
Name of thread :main
Priority of thread :5
Name of thread :Main thread
Hello world
```

*****

Creation of New Threads:

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

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

Runnable interface:

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

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

Starting a thread:

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

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

1) Java Thread Example by extending Thread class

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

```
Output:thread is running...
```

2) Java Thread Example by implementing Runnable interface

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

```
Output:thread is running...
```

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

*****

Thread priority:

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

3 constants defined in Thread class:

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

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

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

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

*****

Multi Threading:

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

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

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

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

Advantages of Java Multithreading

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

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

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

Using isAlive():

The **isAlive()** method of thread class tests if the thread is alive. A thread is considered alive when the start() method of thread class has been called and the thread is not yet dead. This method returns true if the thread is still running and not finished.

Syntax:

public final boolean isAlive()

Return:

This method will return true if the thread is alive otherwise returns false.

Example:

```
public class JavaIsAliveExp extends Thread
{
 public void run()
   {
        try
        {
            Thread.sleep(300);
            System.out.println("is run() method isAlive "+Thread.currentThread().isAlive());
        }
        catch (InterruptedException ie) {
        }
    }
    public static void main(String[] args)
    {
        JavaIsAliveExp t1 = new JavaIsAliveExp();
        System.out.println("before starting thread isAlive: "+t1.isAlive());
        t1.start();
        System.out.println("after starting thread isAlive: "+t1.isAlive());
    }
}
```

Output:

```
before starting thread isAlive: false
after starting thread isAlive: true
is run() method isAlive true
```

join():

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

Example of join() method:

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

```
Output:
1
    2
    3
    4
    5
    1
    1
    2
    2
    3
    3
    4
    4
    5
    5
```

As you can see in the above example,when t1 completes its task then t2 and t3 starts executing.

*****

Synchonization:

Synchronization in java is the capability to control the access ofmultiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization:

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive:

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++)
{
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }
 }
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
  }
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
10
15
20
25
100
200
300
400
500
```

*****

Suspending and Resuming threads:

The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Syntax:

public final void suspend()

Return:

This method does not return any value.

Example:

```
public class JavaSuspendExp extends Thread
{
   public void run()
   {
        for(int i=1; i<5; i++)
        {
            try
            {
               // thread to sleep for 500 milliseconds
                 sleep(500);
                 System.out.println(Thread.currentThread().getName());
            }catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        // creating three threads
        JavaSuspendExp t1=new JavaSuspendExp ();
        JavaSuspendExp t2=new JavaSuspendExp ();
        JavaSuspendExp t3=new JavaSuspendExp ();
        // call run() method
        t1.start();
        t2.start();
        // suspend t2 thread
        t2.suspend();
        // call run() method
        t3.start();
    }
}
```

Output:

```
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
```

```
4
```

The **resume()** method of thread class is only used with suspend() method. This method is used to resume a thread which was suspended using suspend() method. This method allows the suspended thread to start again.

Syntax:

public final void resume()

Return value:

This method does not return any value.

Example:

```
public class JavaResumeExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
         try
         {
            // thread to sleep for 500 milliseconds
             sleep(500);
             System.out.println(Thread.currentThread().getName());
         }catch(InterruptedException e){System.out.println(e);}
         System.out.println(i);
      }
   }
   public static void main(String args[])
   {
      // creating three threads
      JavaResumeExp t1=new JavaResumeExp ();
      JavaResumeExp t2=new JavaResumeExp ();
      JavaResumeExp t3=new JavaResumeExp ();
      // call run() method
      t1.start();
      t2.start();
      t2.suspend(); // suspend t2 thread
      // call run() method
      t3.start();
      t2.resume(); // resume t2 thread
   }
}
```

Output:

```
Thread-0
1
```

```
Thread-2
1
Thread-1
1
Thread-0
2
Thread-2
2
Thread-1
2
Thread-0
3
Thread-2
3
Thread-1
3
Thread-0
4
Thread-2
4
Thread-1
4
```

*****

Communication between Threads Input/Output:

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

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

1) wait() method: Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method: Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

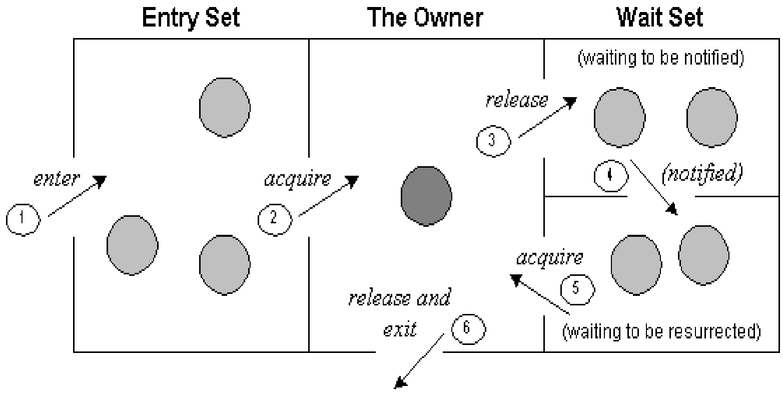
public final void notify()

3) notifyAll() method: Wakes up all threads that are waiting on this object's monitor.

Syntax:

public final void notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Reading and Writing data:

- A thread is reading a file, another thread is opened to write a file;
- A thread is writing a file, another thread is opened to read the file;
- A thread is writing a file, another thread is created to write a file

In short, read-write mutual exclusion, write-read mutual exclusion, write-write mutual exclusion, only read-read compatible (can be asynchronous).

If a thread wants to read the resource, it is okay as long as no threads are writing to it, and no threads have requested write access to the resource. By up-prioritizing write-access requests we assume that write requests are more important than read-requests. Besides, if reads are what happens most often, and we did not up-prioritize writes, starvation could occur. Threads requesting write access would be blocked until all readers had unlocked the ReadWriteLock. If new threads were constantly granted read access the thread waiting for write access would remain blocked indefinately, resulting in starvation. Therefore a thread can only be granted read access if no thread has currently locked the ReadWriteLock for writing, or requested it locked for writing.

A thread that wants write access to the resource can be granted so when no threads are reading nor writing to the resource. It doesn't matter how many threads have requested write access or in what sequence, unless you want to guarantee fairness between threads requesting write access.

*****

java.io.package:

This package provides for system input and output through data streams, serialization and the file system. Unless otherwise noted, passing a null argument to a constructor or method in any class or interface in this package will cause a NullPointerException to be thrown.

Following are the important classes in Java.io package:

- BufferedInputStream
- BufferedOutputStream
- BufferedReader
- BufferedWriter
- ByteArrayInputStream
- ByteArrayOutputStream
- CharArrayReader
- CharArrayWriter – Set1 Set2
- Console
- DataInputStream – Set1 Set2
- DataOutputStream
- File
- FileDescriptor
- FileInputStream
- FileOutputStream
- FilePermission
- FileReader and FileWriter
- FilterInputStream
- FilterOutputStream
- FilterReader
- FilterWriter
- InputStream
- InputStreamReader

*****

Applets

Applet Class:

Every applet is an extension of the java.applet.Applet class. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

Applet Structure:

- **The Applet class:**
    - All applets extend java.applet.Applet
    - There is no main() method
    - init() is not static
    -
        - Our applet is run as an instance of the **HelloApplet** class
- **The init() method:**
    - Called automatically when the applet is loaded
    - Rarely called directly

The Applet Class:

All applets extend Java's built in Applet class. This resides in a package called java.applet; hence the first import statement. Other things to notice:
There is no main() method.
Our one method, init(), is not static. That means our applet is run as an instance of the HelloApplet class.

Applet Methods:

init()
The init() method is the heart of this applet:
It is called automatically when the applet is first loaded. You will rarely, if ever, call it directly.
When called, it instantiates a JLabel and adds it to the applet.
The JLabel class lives in the javax.swing package, hence the second import statement.

An Example Applet Program:

```
// A Hello World Applet
// Save file as HelloWorld.java

import java.applet.Applet;
import java.awt.Graphics;

// HelloWorld class extends Applet
public class HelloWorld extends Applet
{
   // Overriding paint() method
   @Override
   public void paint(Graphics g)
   {
      g.drawString("Hello World", 20, 20);
   }
}
```

1. The above java program begins with two import statements. The first import statement imports the Applet class from applet package. Every AWT-based(Abstract Window Toolkit) applet that you create must be a subclass (either directly or indirectly) of Applet class. The second statement import the Graphics class from AWT package.
2. The next line in the program declares the class HelloWorld. This class must be declared as public because it will be accessed by code that is outside the program. Inside HelloWorld, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet.
3. Inside **paint( )** is a call to drawString( ), which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location. It has the following general form:
4. void drawString(String message, int x, int y)

   Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to drawString( ) in the applet causes the message “Hello World” to be displayed beginning at location 20,20.

Notice that the applet does not have a **main( )** method. Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

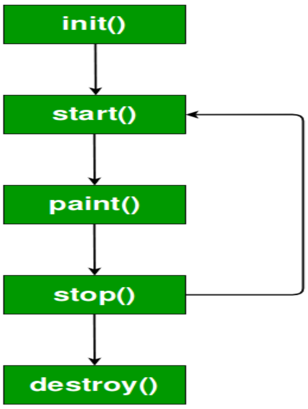
Running the HelloWorld Applet :

After you enter the source code for HelloWorld.java, compile in the same way that you have been compiling java programs(using javac command). However, running HelloWorld with the *java* command will generate an error because it is not an application.

Output:

```
java HelloWorld

Error: Main method not found in class HelloWorld, please define the main method as:

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

Applet Life Cycle:



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( )

Let's look more closely at these methods.

1.init( ) : The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.

2.start( ) : The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )** is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

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. For example, the window in which the

applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.
Paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.

The **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.

4.stop( ): The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

5.destroy( ) : The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

*****

paint( ) :

public void paint(Graphics)
Every Java Component implements **paint(Graphics)**, which is responsible for painting that component in the Graphics context passed in the parameter. When you extend a Component (like when you write a Applet), if you want to display it differently than its superclass, you override **public void paint(Graphics)** .

update():

public void update(Graphics)

By default **update(Graphics)** fills the drawable area of a Component with its background color, and then sends **paint(Graphics)** to the object. Thus, flicker that comes from redrawing the background over and over,

repaint():

The **repaint()** method is sent to a Component when it needs to be repainted. This happens when a window is moved, or resized, or unhidden. It also happens when a webpage contains an image and the pixels of the image are arriving slowly down the wire.

When a Container, like a **Frame**, is painted, all of its Components (Buttons, TextFields, whatever) must be repainted. This is accomplished (roughly), in Java, by sending **repaint()** to every Component in the Container, in the order they were added to the container.

*****

UNIT IV

Event Handling -Introduction, Event Delegation Model, java.awt.event Description, Sources of Events, Event Listeners, Adapter classes, Inner classes.

Abstract Window Toolkit:Why AWT?, java.awt package, Components and Containers, Button, Label, Checkbox, Radio buttons, List boxes, Choice boxes, Text field and Text area, container classes, Layouts, Menu, Scroll bar, **Swing**: Introduction, JFrame, JApplet, JPanel, Components in swings, Layout Managers, JList and JScroll Pane, Split Pane, JTabbedPane, Dialog Box Pluggable Look and Feel.

Event Handling

Introduction:

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Advantages of event Handling:

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.

Delegation Model:

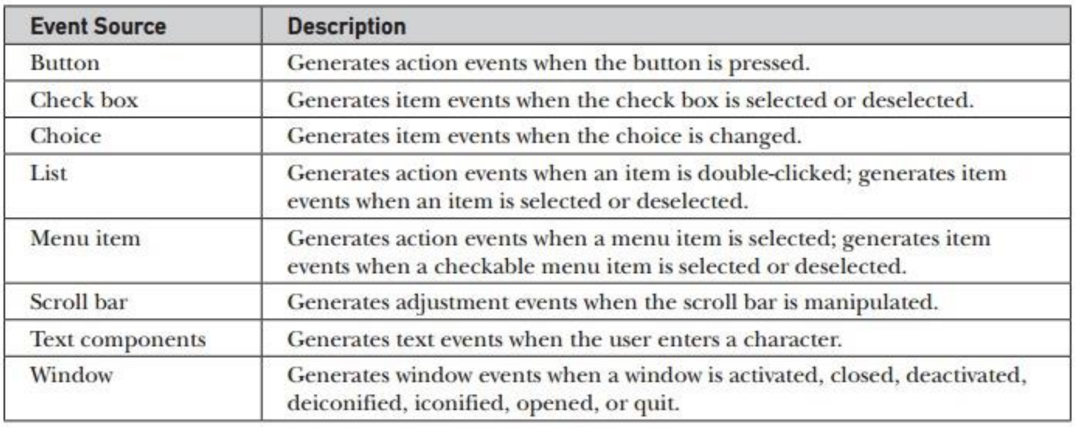
- The modern approach to handling events is based on the *delegation event model,*
- which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners.*
- In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to “delegate” the processing of an event to a separate piece of code.
- In the delegation event model, listeners must register with a source in order to receive an event notification.
- This provides an important benefit: notifications are sent only to listeners that want to receive them.
- This is a more efficient way to handle events than the design used by the original Java 1.0 approach.
- Previously, an event was propagated up the containment hierarchy until it was handled by a component.

- This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

java.awt.event Description:

Class and Description
ActionEvent A semantic event which indicates that a component-defined action occurred.
ActionListener The listener interface for receiving action events.
AdjustmentEvent The adjustment event emitted by Adjustable objects like **Scrollbar** and **ScrollPane**.
AdjustmentListener The listener interface for receiving adjustment events.
AWTEventListener The listener interface for receiving notification of events dispatched to objects that are instances of Component or MenuComponent or their subclasses.
ComponentEvent A low-level event which indicates that a component moved, changed size, or changed visibility (also, the root class for the other component-level events).
ComponentListener The listener interface for receiving component events.
ContainerEvent A low-level event which indicates that a container's contents changed because a component was added or removed.
ContainerListener The listener interface for receiving container events.

Sources of Events:



Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

*****

Event Listeners:

Event listeners represent the interfaces responsible to handle events. Java provides various Event listener classes, however, only those which are more frequently used will be discussed. Every method of an event listener method has a single argument as an object which is the subclass of EventObject class. For example, mouse event listener methods will accept instance of MouseEvent, where MouseEvent derives from EventObject.

EventListner Interface

It is a marker interface which every listener interface has to extend. This class is defined in **java.util** package.

Class Declaration

Following is the declaration for **java.util.EventListener** interface −

public interface EventListener

SWING Event Listener Interfaces

Following is the list of commonly used event listeners.

Sr.No.	Class & Description
1	ActionListener This interface is used for receiving the action events.
2	ComponentListener This interface is used for receiving the component events.
3	ItemListener This interface is used for receiving the item events.
4	KeyListener This interface is used for receiving the key events.
5	MouseListener This interface is used for receiving the mouse events.
6	WindowListener This interface is used for receiving the window events.
7	AdjustmentListener This interface is used for receiving the adjustment events.
8	ContainerListener This interface is used for receiving the container events.
9	MouseMotionListener This interface is used for receiving the mouse motion events.

10	FocusListener This interface is used for receiving the focus events.

*****

Adapter classes:

Java adapter classes provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

The adapter classes are found in **java.awt.event**, **java.awt.dnd** and **javax.swing.event** packages. The Adapter classes with their corresponding listener interfaces are given below.

java.awt.event Adapter classes:

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

*****

Inner Classes:

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

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

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

Syntax of Inner class:

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }  }
```

Advantage of java inner classes:

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

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

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

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

*****

Abstract Window Tool Kit

Why AWT?

Abstract Window Toolkit (AWT) is a set of application program interfaces (API s) used by Java programmers to create graphical user interface (GUI) objects, such as buttons, scroll bars, and windows. AWT is part of the Java Foundation Classes (JFC) from Sun Microsystems, the company that originated Java. The JFC are a comprehensive set of GUI class libraries that make it easier to develop the user interface part of an application program.

A more recent set of GUI interfaces called Swing extends the AWT so that the programmer can create generalized GUI objects that are independent of a specific operating system's windowing system.

java.awt.package:

The java.awt Package The java.awt package is the main package of the AWT, or Abstract Windowing Toolkit. It contains classes for graphics, including the Java 2D graphics capabilities introduced in the Java 2 platform, and also defines the basic graphical user interface (GUI) framework for Java. java.awt also includes a number of heavyweight GUI objects, many of which have been superseded by the javax.swing package. java.awt also has a number of important subpackages. The most important graphics classes in java.awt are Graphics and its Java 2D extension, Graphics2D. These classes represent a drawing surface, maintain a set of drawing attributes, and define methods for drawing and filling lines, shapes, and text. Classes that represent graphics attributes include Color, Font, Paint, Stroke, and Composite. The Image class and Shape interface represent things that you can draw using a Graphics object and the various graphics attributes.

Interface	Description
ActiveEvent	An interface for events that know how to dispatch themselves.
Adjustable	The interface for objects which have an adjustable numeric value contained within a bounded range of values.
Composite	The Composite interface, along withCompositeContext, defines the methods to compose a draw primitive with the underlying graphics area.
ItemSelectable	The interface for objects which contain a set of items forwhich zero or more can be selected.
LayoutManager	Defines the interface for classes that know how to lay out Containers.
MenuContainer	The super class of all menu related containers.

Paint	This Paint interface defines how color patterns can begenerated for **Graphics2D** operations.
PaintContext	The PaintContext interface defines the encapsulated and optimized environment to generate color patterns in device space for fill or stroke operations on a **Graphics2D**.

*****

Components and Containers:

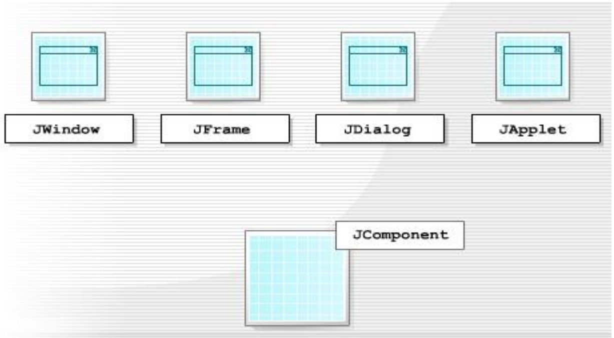
In Java, a component is the basic user interface object and is found in all Java applications. Components include lists, buttons, panels, and windows.

To use components, you need to place them in a container.

A container is a component that holds and manages other components. Containers display components using a layout manager.

Swing components inherit from the javax.Swing.JComponent class, which is the root of the Swing component hierarchy. JComponent, in turn, inherits from the Container class in the Abstract Windowing Toolkit (AWT). So Swing is based on classes inherited from AWT.

Swing provides the following useful top-level containers, all of which inherit from JComponent:



JWindow:JWindow is a top-level window that doesn't have any trimmings and can be displayed anywhere on a desktop. JWindow is a heavyweight component. You usually use JWindow to create pop-up windows and "splash" screens. JWindow extends AWT's Window class.

JFrame:JFrame is a top-level window that can contain borders and menu bars. JFrame is a subclass of JWindow and is thus a heavyweight component. You place a JFrame on a JWindow. JFrame extends AWT's Frame class.

JDialog:JDialog is a lightweight component that you use to create dialog windows. You can place dialog windows on a JFrame or JApplet. JDialog extends AWT's Dialog class.

JApplet:JApplet is a container that provides the basis for applets that run within web browsers. JApplet is a lightweight component that can contain other graphical user interface (GUI) components. JApplet extends AWT's Applet class.

Button:

The button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

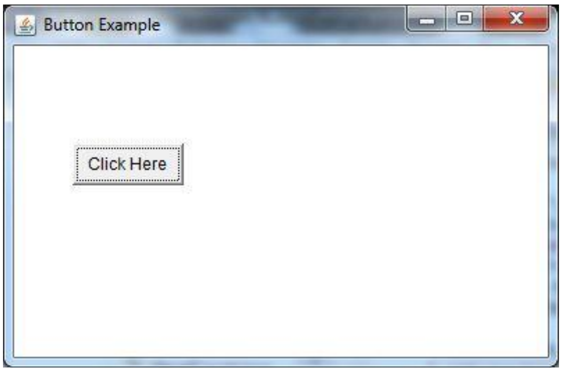
AWT Button Class declaration

public class Button extends Component implements Accessible

Java AWT Button Example

```
import java.awt.*;
public class ButtonExample {
public static void main(String[] args) {
   Frame f=new Frame("Button Example");
   Button b=new Button("Click Here");
   b.setBounds(50,100,80,30);
   f.add(b);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
}  }
```

Output:



***

Label:

The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

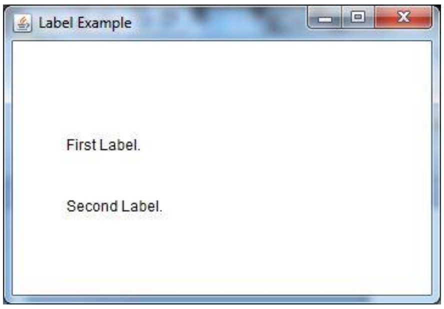
AWT Label Class Declaration

public class Label extends Component implements Accessible

Java Label Example

```
import java.awt.*;
class LabelExample{
public static void main(String args[]){
   Frame f= new Frame("Label Example");
   Label l1,l2;
   l1=new Label("First Label.");
   l1.setBounds(50,100, 100,30);
   l2=new Label("Second Label.");
   l2.setBounds(50,150, 100,30);
   f.add(l1); f.add(l2);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
}  }
```

Output:



First Label.

Second Label.

***

Checkbox:

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

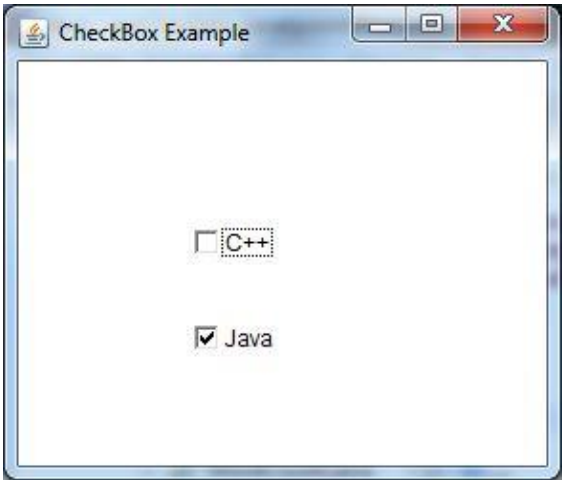
AWT Checkbox Class Declaration

public class Checkbox extends Component implements ItemSelectable, Accessible

Java AWT Checkbox Example:

```
import java.awt.*;
public class CheckboxExample
{
     CheckboxExample(){
       Frame f= new Frame("Checkbox Example");
        Checkbox checkbox1 = new Checkbox("C++");
        checkbox1.setBounds(100,100, 50,50);
        Checkbox checkbox2 = new Checkbox("Java", true);
        checkbox2.setBounds(100,150, 50,50);
        f.add(checkbox1);
        f.add(checkbox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
    new CheckboxExample();
}
}
```

Output:



***

Radio buttons:

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

JRadioButton class declaration
public class JRadioButton extends JToggleButton implements Accessible

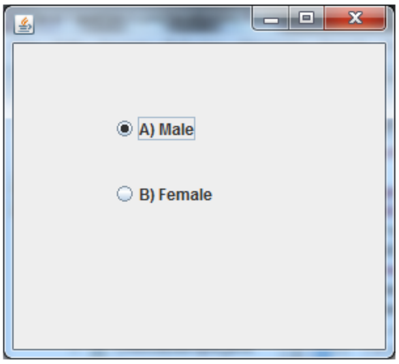
Commonly used Constructors:

Constructor	Description
JRadioButton()	Creates an unselected radio button with no text.
JRadioButton(String s)	Creates an unselected radio button with specified text.
JRadioButton(Strings,boolean selected)	Creates a radio button with the specified text and selected status.

Java JRadioButton Example:

```
import javax.awt.*;
public class RadioButtonExample {
JFrame f;
RadioButtonExample(){
f=new JFrame();
JRadioButton r1=new JRadioButton("A) Male");
JRadioButton r2=new JRadioButton("B) Female");
r1.setBounds(75,50,100,30);
r2.setBounds(75,100,100,30);
ButtonGroup bg=new ButtonGroup();
bg.add(r1);bg.add(r2);
f.add(r1);f.add(r2);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String[] args) {
    new RadioButtonExample();
}
}
```

Output:



***

List boxes:

The object of List class represents a list of text items. By the help of list, user can choose either one item or multiple items. It inherits Component class.

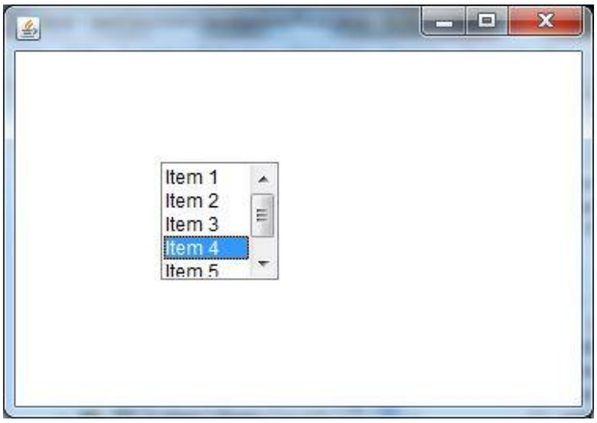
AWT List class Declaration

public class List extends Component implements ItemSelectable, Accessible

Java AWT List Example

```
import java.awt.*;
public class ListExample
{
     ListExample(){
        Frame f= new Frame();
        List l1=new List(5);
        l1.setBounds(100,100, 75,75);
        l1.add("Item 1");
        l1.add("Item 2");
        l1.add("Item 3");
        l1.add("Item 4");
        l1.add("Item 5");
        f.add(l1);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
   new ListExample();
}
}
```

Output:



```
Item 1
Item 2
Item 3
Item 4
Item 5
```

***

Choice boxes:

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

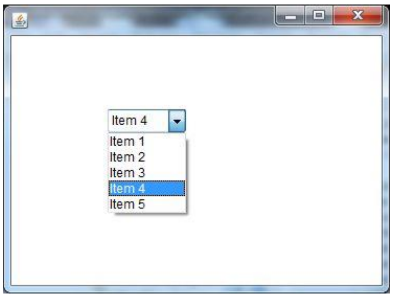
AWT Choice Class Declaration

public class Choice extends Component implements ItemSelectable, Accessible

Java AWT Choice Example

```
import java.awt.*;
public class ChoiceExample
{
        ChoiceExample(){
        Frame f= new Frame();
        Choice c=new Choice();
        c.setBounds(100,100, 75,75);
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
        f.add(c);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
  }
public static void main(String args[])
{
   new ChoiceExample();
}
}
```

Output:



***

Text field and Text area:

Text Field:

The object of a TextField class is a text component that allows the editing of a single line text. It inherits TextComponent class.

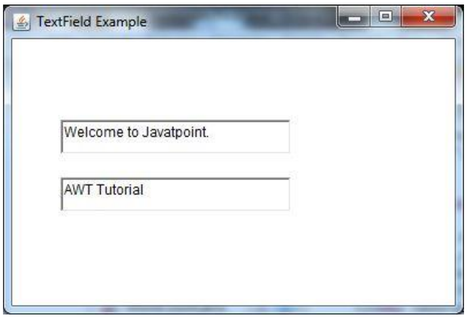
AWT TextField Class Declaration

public class TextField extends TextComponent

Java AWT TextField Example

```
import java.awt.*;
class TextFieldExample{
public static void main(String args[]){
    Frame f= new Frame("TextField Example");
    TextField t1,t2;
    t1=new TextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new TextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



Welcome to Javatpoint.

AWT Tutorial

Text Area:

The object of a TextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

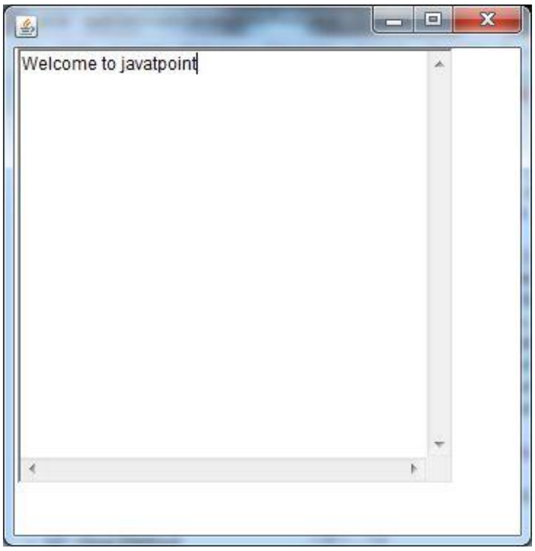
AWT TextArea Class Declaration

public class TextArea extends TextComponent

Java AWT TextArea Example

```
import java.awt.*;
public class TextAreaExample
{
    TextAreaExample(){
      Frame f= new Frame();
         TextArea area=new TextArea("Welcome to javatpoint");
      area.setBounds(10,30, 300,300);
      f.add(area);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
public static void main(String args[])
{
  new TextAreaExample();
}
}
```

Output:



Welcome to javatpoint

***

Container classes:

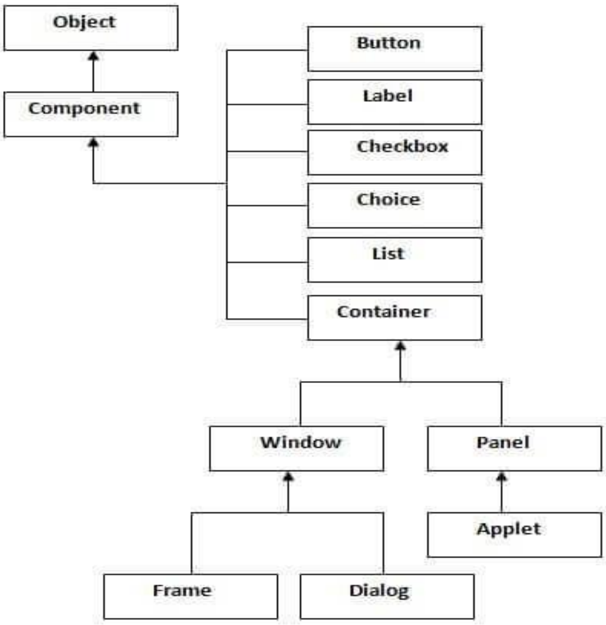
Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy:

The hierarchy of Java AWT classes are given below.



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

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

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

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

***

Layouts:

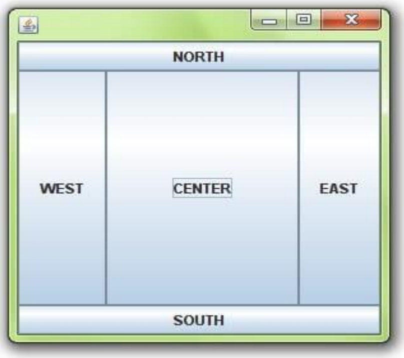
The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

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.

1.Java BorderLayout:

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. 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



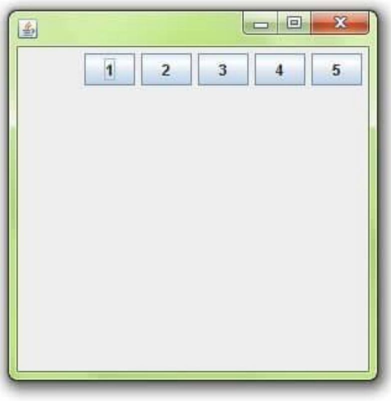
2.Java FlowLayout:

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.

Fields of FlowLayout class

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

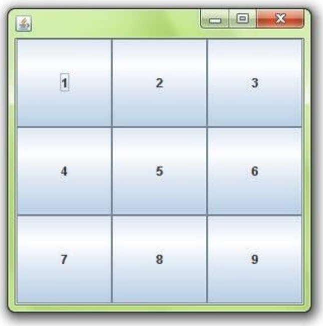
Example of FlowLayout class:



3.Java GridLayout:

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

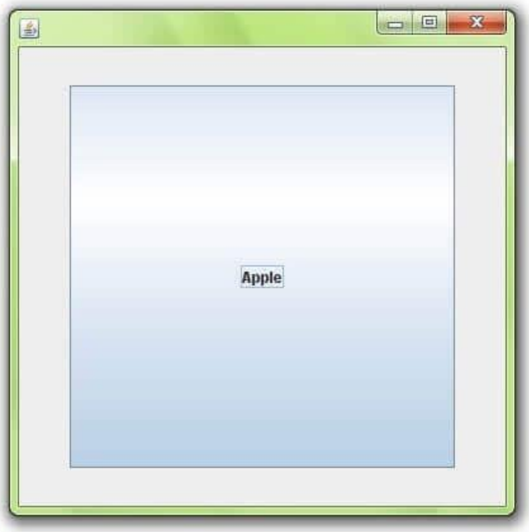
Example of GridLayout class:



4.Java CardLayout:

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Example of CardLayout class:

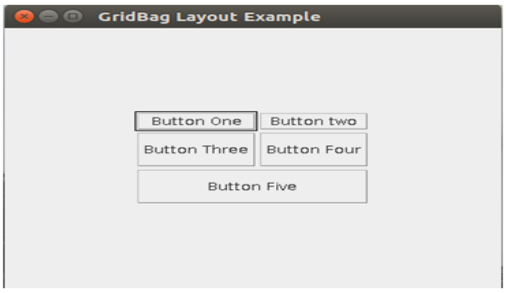


5.Java GridBagLayout:

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of constraints object we arrange component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.

Output:



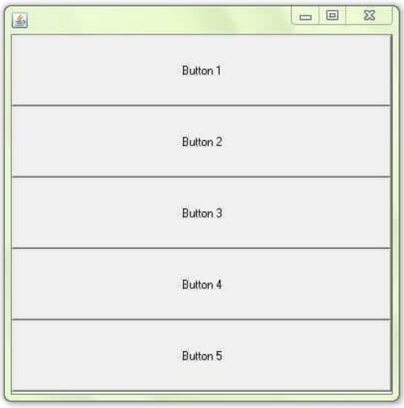
6.Java BoxLayout:

The BoxLayout is used to arrange the components either vertically or horizontally. For this purpose, BoxLayout provides four constants. They are as follows:

Fields of BoxLayout class:

1. public static final int X_AXIS
2. public static final int Y_AXIS
3. public static final int LINE_AXIS
4. public static final int PAGE_AXIS

Example of BoxLayout class with Y-AXIS:



7.GroupLayout:

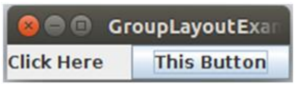
GroupLayout groups its components and places them in a Container hierarchically. The grouping is done by instances of the Group class.

Group is an abstract class and two concrete classes which implement this Group class are SequentialGroup and ParallelGroup.

SequentialGroup positions its child sequentially one after another where as ParallelGroup aligns its child on top of each other.

The GroupLayout class provides methods such as createParallelGroup() and createSequentialGroup() to create groups.

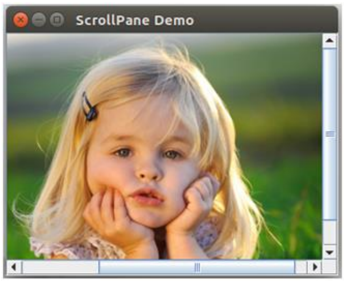
Output:



8.ScrollPaneLayout:

The layout manager used by JScrollPane. JScrollPaneLayout is responsible for nine components: a viewport, two scrollbars, a row header, a column header, and four "corner" components.

Output:



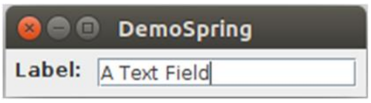
9.Java SpringLayout:

A SpringLayout arranges the children of its associated container according to a set of constraints.Constraints are nothing but horizontal and vertical distance between two component edges. Every constrains are represented by a SpringLayout.Constraint object.

Each child of a SpringLayout container, as well as the container itself, has exactly one set of constraints associated with them.

Each edge position is dependent on the position of the other edge. If a constraint is added to create new edge than the previous binding is discarded. SpringLayout doesn't automatically set the location of the components it manages.

Output:



Label: A Text Field

***

Menu:

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration

public class MenuItem extends MenuComponent implements Accessible

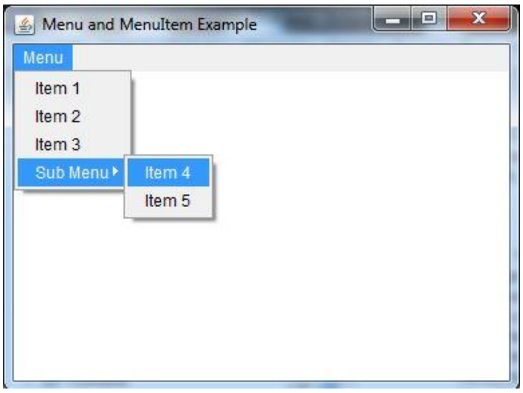
AWT Menu class declaration

public class Menu extends MenuItem implements MenuContainer, Accessible

Java AWT MenuItem and Menu Example

```
import java.awt.*;
class MenuExample
{
    MenuExample(){
        Frame f= new Frame("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("Menu");
        Menu submenu=new Menu("Sub Menu");
        MenuItem i1=new MenuItem("Item 1");
       MenuItem i2=new MenuItem("Item 2");
        MenuItem i3=new MenuItem("Item 3");
        MenuItem i4=new MenuItem("Item 4");
        MenuItem i5=new MenuItem("Item 5");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        submenu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```

Output:



***

Scroll bar:

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

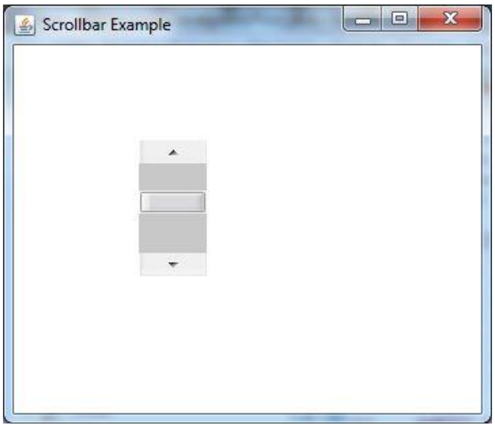
AWT Scrollbar class declaration:

public class Scrollbar extends Component implements Adjustable, Accessible

Java AWT Scrollbar Example

```
import java.awt.*;
class ScrollbarExample{
ScrollbarExample(){
Frame f= new Frame("Scrollbar Example");
Scrollbar s=new Scrollbar();
 s.setBounds(100,100, 50,100);
f.add(s);
f.setSize(400,400);
 f.setLayout(null);
  f.setVisible(true);
}
public static void main(String args[]){
      new ScrollbarExample();
}
}
```

Output:



*****

Swing:

Introduction:

Java Swing is a part of Java Foundation Classes (JFC) which was designed for enabling large-scale enterprise development of Java applications.

Java Swing is a set of APIs that provides the graphical user interface (GUI) for Java programs. Java Swing is also known as the Java GUI widget toolkit.

Java Swing or Swing was developed based on earlier APIs called Abstract Windows Toolkit (AWT). Swing provides richer and more sophisticated GUI components than AWT.

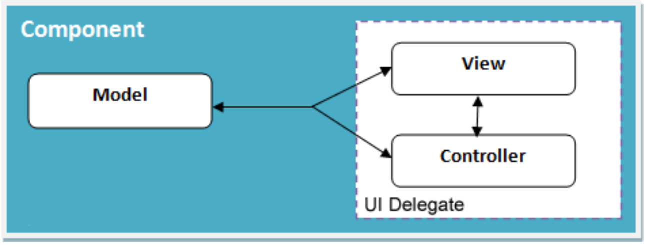
Swing Architecture:

Swing is a platform-independent and enhanced MVC (Model –View – Controller) framework for Java applications. Here are the most important features in Swing architecture.

1.Pluggable look and feel: Swing supports several looks and feels and currently supports Windows, UNIX, Motif, and native Java metal look and feel. Swing allows users to switch look and feel at runtime without restarting the application. By doing this, users can make their own choice to choose which look and feel is the best for them instantly.

2.Lightweight components: All swing components are lightweight except for some top-level containers. Lightweight means the component renders or paints itself using drawing primitives of the Graphics object instead of relying on the host operating system (OS). As a result, the application presentation is rendered faster and consumed less memory than previous Java GUI applications.

3.Simplified MVC: Swing uses simplified model-view-architecture (MVC) as the core design behind each component called model-delegate. Based on this architecture, each swing component contains a model and a UI delegate. A UI delegate wraps a view and a controller in MVC architecture, as in the picture below. UI delegate is responsible for painting screens and handling GUI events. Model is in charge of maintaining information or states of the component.



Java Swing MVC – Model Delegate

*****

JFrame:

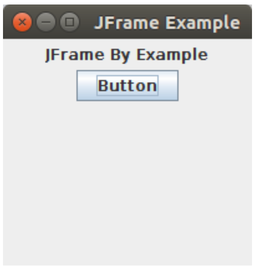
The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class JFrameExample {
   public static void main(String s[]) {
      JFrame frame = new JFrame("JFrame Example");
      JPanel panel = new JPanel();
      panel.setLayout(new FlowLayout());
      JLabel label = new JLabel("JFrame By Example");
      JButton button = new JButton();
      button.setText("Button");
      panel.add(label);
      panel.add(button);
      frame.add(panel);
      frame.setSize(200, 300);
      frame.setLocationRelativeTo(null);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        frame.setVisible(true);
    }
}
```

Output:



***

JApplet:

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

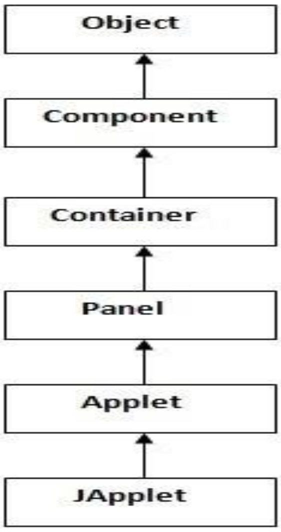
There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Hierarchy of Applet:



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

Lifecycle of Java Applet:

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

Applet Lifecycle

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

Lifecycle methods for Applet:

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

java.applet.Applet class:

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

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

5. ****
6. ******

7. **********8c:\>app

JPanel:

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class.

It doesn't have title bar.

JPanel class declaration

public class JPanel extends JComponent implements Accessible

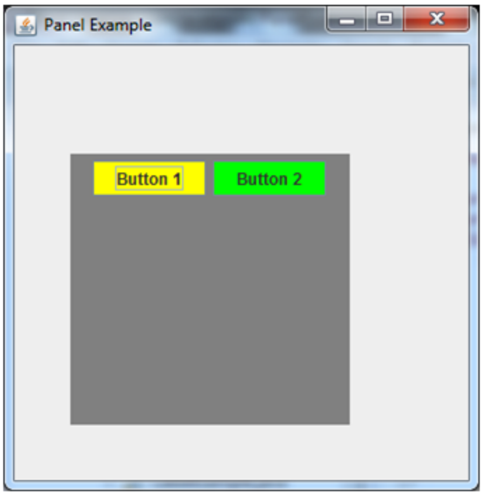
Commonly used Constructors:

Constructor	Description
JPanel()	It is used to create a new JPanel with a double buffer and a flow layout.
JPanel(boolean isDoubleBuffered)	It is used to create a new JPanel with FlowLayout and the specified buffering strategy.
JPanel(LayoutManager layout)	It is used to create a new JPanel with the specified layout manager.

Java JPanel Example

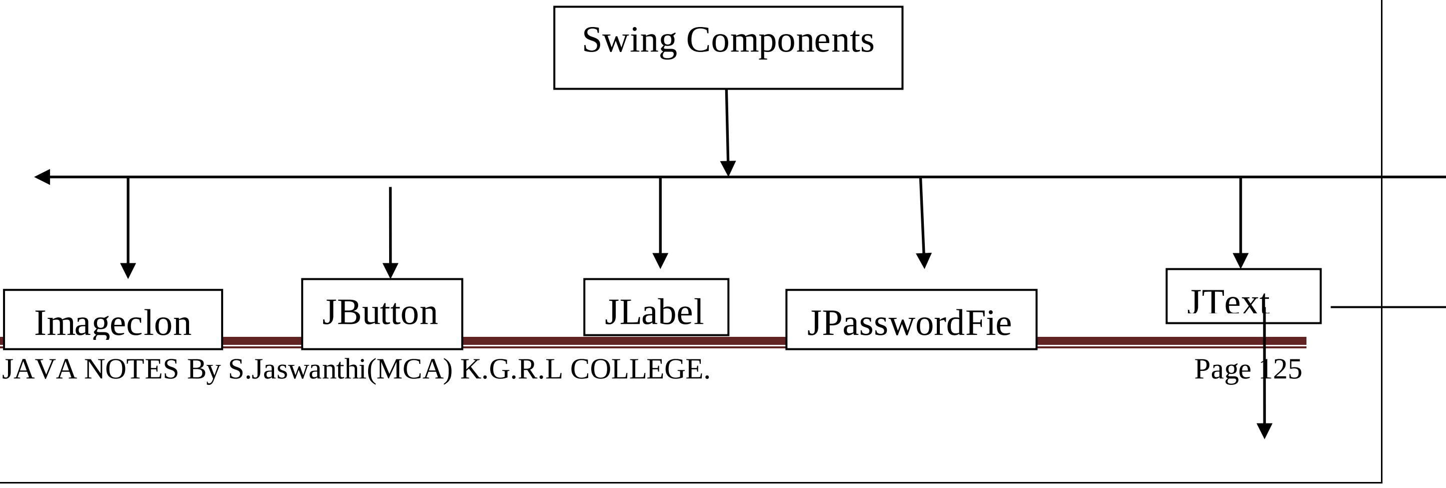
```
import java.awt.*;
import javax.swing.*;
public class PanelExample {
    PanelExample()
      {
      JFrame f= new JFrame("Panel Example");
    JPanel panel=new JPanel();
    panel.setBounds(40,80,200,200);
    panel.setBackground(Color.gray);
    JButton b1=new JButton("Button 1");
    b1.setBounds(50,100,80,30);
    b1.setBackground(Color.yellow);
    JButton b2=new JButton("Button 2");
    b2.setBounds(100,100,80,30);
    b2.setBackground(Color.green);
    panel.add(b1); panel.add(b2);
    f.add(panel);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
    }
    public static void main(String args[])
    {
    new PanelExample();
   }
}
```

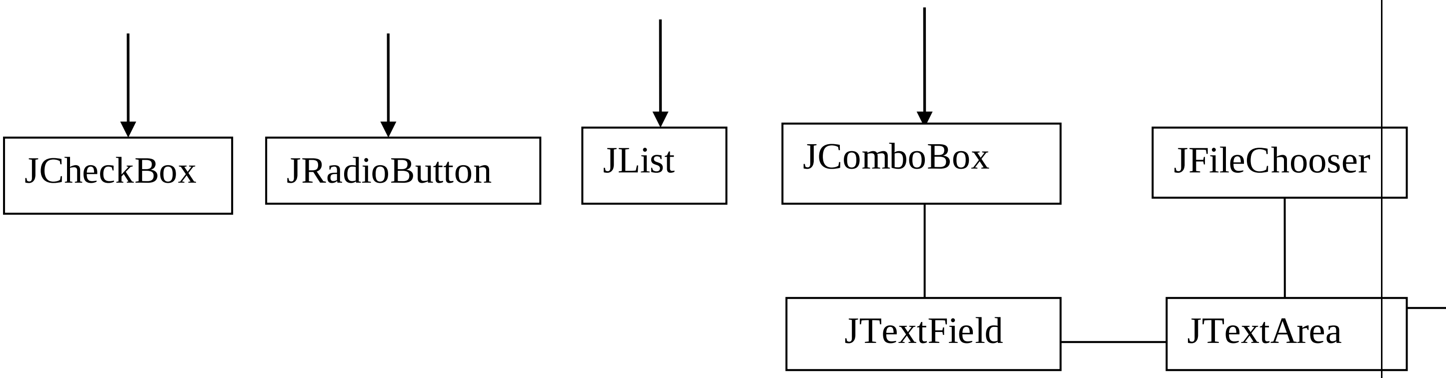
Output:



***

Components in swings:





1.Imageclon:

The class **ImageIcon** is an implementation of the Icon interface that paints Icons from Images.

Class Declaration

```
public class ImageIcon
   extends Object
      implements Icon, Serializable, Accessible
```

JCheckBox:

The JCheckBox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".It inherits JToggleButton class.

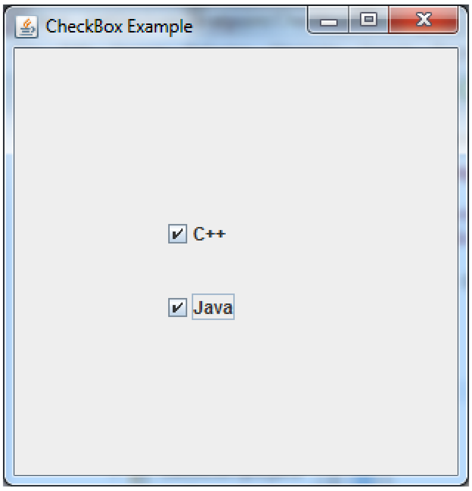
JCheckBox class declaration

public class JCheckBox **extends** JToggleButton **implements** Accessible

Java JCheckBox Example

```
import javax.swing.*;
public class CheckBoxExample
{
     CheckBoxExample(){
        JFrame f= new JFrame("CheckBox Example");
        JCheckBox checkBox1 = new JCheckBox("C++");
        checkBox1.setBounds(100,100, 50,50);
        JCheckBox checkBox2 = new JCheckBox("Java", true);
        checkBox2.setBounds(100,150, 50,50);
        f.add(checkBox1);
        f.add(checkBox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
    {
    new CheckBoxExample();
    }}
```

Output:



2.JButton:

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

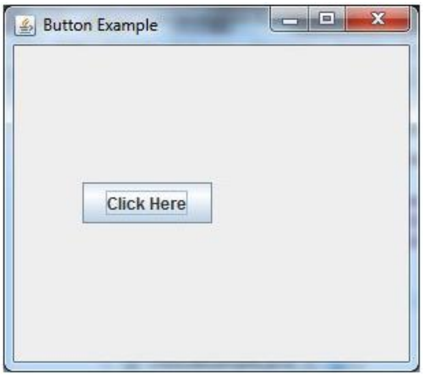
JButton class declaration:

public class JButton **extends** AbstractButton **implements** Accessible

Example:

```
import javax.swing.*;
public class ButtonExample {
public static void main(String[] args) {
    JFrame f=new JFrame("Button Example");
    JButton b=new JButton("Click Here");
    b.setBounds(50,100,95,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



JRadioButton:

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

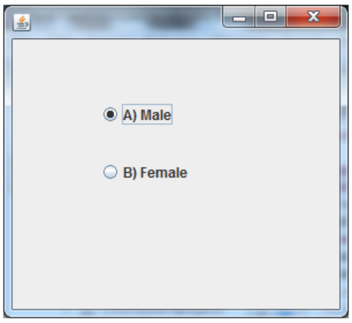
JRadioButton class declaration

public class JRadioButton **extends** JToggleButton **implements** Accessible

Java JRadioButton Example

```
import javax.swing.*;
public class RadioButtonExample {
JFrame f;
RadioButtonExample(){
f=new JFrame();
JRadioButton r1=new JRadioButton("A) Male");
JRadioButton r2=new JRadioButton("B) Female");
r1.setBounds(75,50,100,30);
r2.setBounds(75,100,100,30);
ButtonGroup bg=new ButtonGroup();
bg.add(r1);bg.add(r2);
f.add(r1);f.add(r2);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String[] args) {
   new RadioButtonExample();
}
}
```

Output:



3.JLabel:

The object of JLabel class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

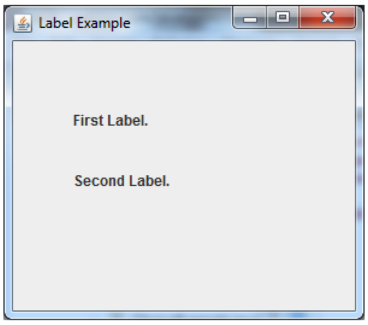
JLabel class declaration

public class JLabel **extends** JComponent **implements** SwingConstants, Accessible

Java JLabel Example

```
import javax.swing.*;
class LabelExample
{
public static void main(String args[])
   {
   JFrame f= new JFrame("Label Example");
   JLabel l1,l2;
   l1=new JLabel("First Label.");
   l1.setBounds(50,50, 100,30);
   l2=new JLabel("Second Label.");
   l2.setBounds(50,100, 100,30);
   f.add(l1); f.add(l2);
   f.setSize(300,300);
   f.setLayout(null);
   f.setVisible(true);
   }
   }
```

Output:



JList:

The object of JList class represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items. It inherits JComponent class.

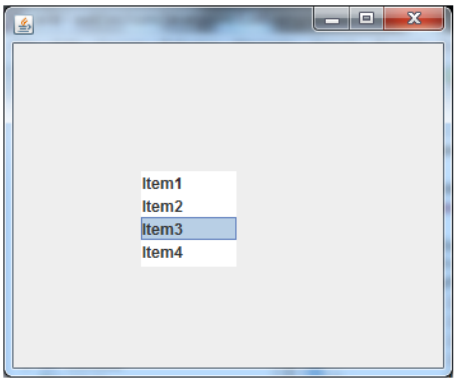
JList class declaration

```
public class JList extends JComponent implements Scrollable, Accessible
```

Java JList Example:

```
import javax.swing.*;
public class ListExample
{
    ListExample(){
      JFrame f= new JFrame();
      DefaultListModel<String> l1 = new DefaultListModel<>();
        l1.addElement("Item1");
        l1.addElement("Item2");
        l1.addElement("Item3");
        l1.addElement("Item4");
        JList<String> list = new JList<>(l1);
        list.setBounds(100,100, 75,75);
        f.add(list);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String args[])
   {
  new ListExample();
   }}
```

Output:



```
Item1
Item2
Item3
Item4
```

4.JPasswordField:

The object of a JPasswordField class is a text component specialized for password entry. It allows the editing of a single line of text. It inherits JTextField class.

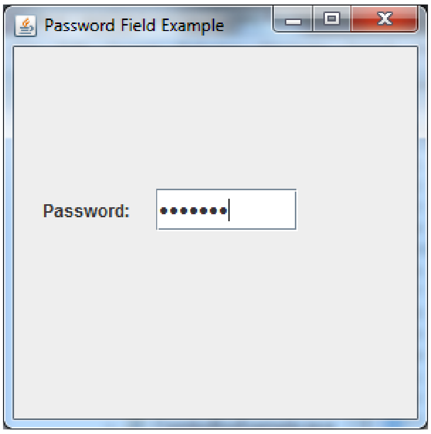
JPasswordField class declaration

public class JPasswordField **extends** JTextField

Java JPasswordField Example

```
import javax.swing.*;
public class PasswordFieldExample {
   public static void main(String[] args) {
   JFrame f=new JFrame("Password Field Example");
    JPasswordField value = new JPasswordField();
    JLabel l1=new JLabel("Password:");
      l1.setBounds(20,100, 80,30);
       value.setBounds(100,100,100,30);
         f.add(value);  f.add(l1);
         f.setSize(300,300);
         f.setLayout(null);
         f.setVisible(true);
}  }
```

Output:



JComboBox:

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits JComponent class.

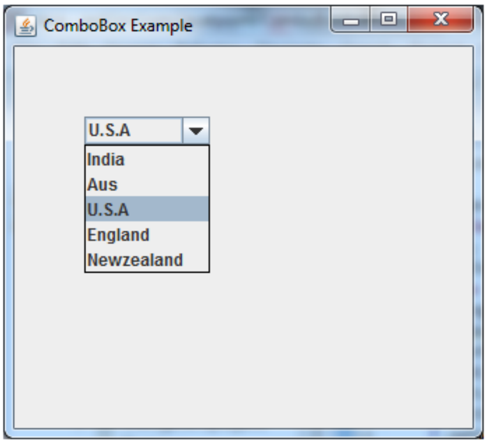
JComboBox class declaration

```
public class JComboBox extends JComponent implements ItemSelectable, ListDataListener, ActionListener, Accessible
```

Java JComboBox Example

```
import javax.swing.*;
public class ComboBoxExample {
JFrame f;
ComboBoxExample(){
   f=new JFrame("ComboBox Example");
   String country[]={"India","Aus","U.S.A","England","Newzealand"};
   JComboBox cb=new JComboBox(country);
   cb.setBounds(50, 50,90,20);
   f.add(cb);
   f.setLayout(null);
   f.setSize(400,500);
   f.setVisible(true);
}
public static void main(String[] args) {
   new ComboBoxExample();
}   }
```

Output:



JTextField:

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

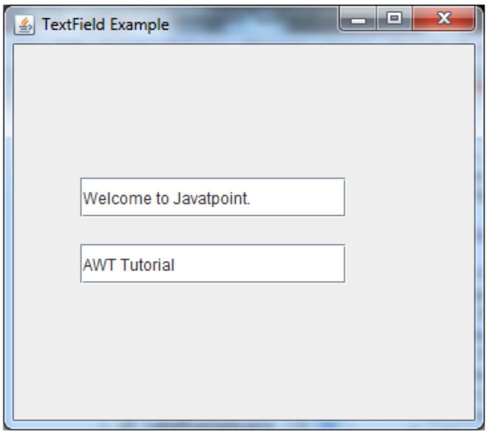
JTextField class declaration

```
public class JTextField extends JTextComponent implements SwingConstants
```

JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
   {
   JFrame f= new JFrame("TextField Example");
   JTextField t1,t2;
   t1=new JTextField("Welcome to Javatpoint.");
   t1.setBounds(50,100, 200,30);
   t2=new JTextField("AWT Tutorial");
   t2.setBounds(50,150, 200,30);
   f.add(t1); f.add(t2);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
   }   }
```

Output:



5.JText:

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

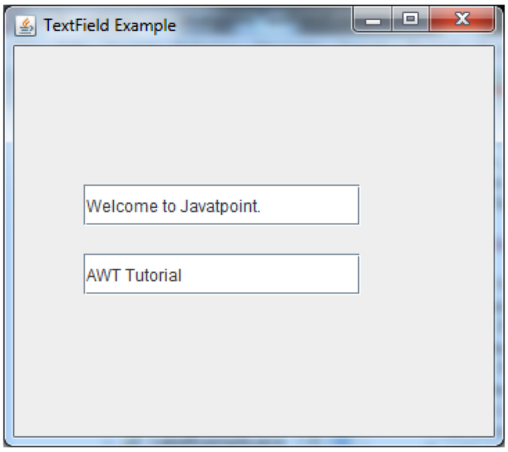
JTextField class declaration

```
public class JTextField extends JTextComponent implements SwingConstants
```

JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
   {
   JFrame f= new JFrame("TextField Example");
   JTextField t1,t2;
   t1=new JTextField("Welcome to Javatpoint.");
   t1.setBounds(50,100, 200,30);
   t2=new JTextField("AWT Tutorial");
   t2.setBounds(50,150, 200,30);
   f.add(t1); f.add(t2);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
   }
   }
```

Output:



Welcome to Javatpoint.

AWT Tutorial

JFileChooser:

The object of JFileChooser class represents a dialog window from which the user can select file. It inherits JComponent class.

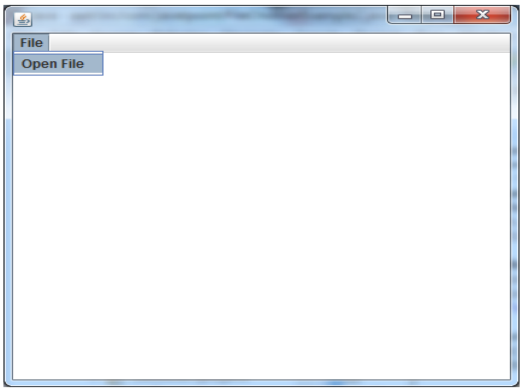
JFileChooser class declaration

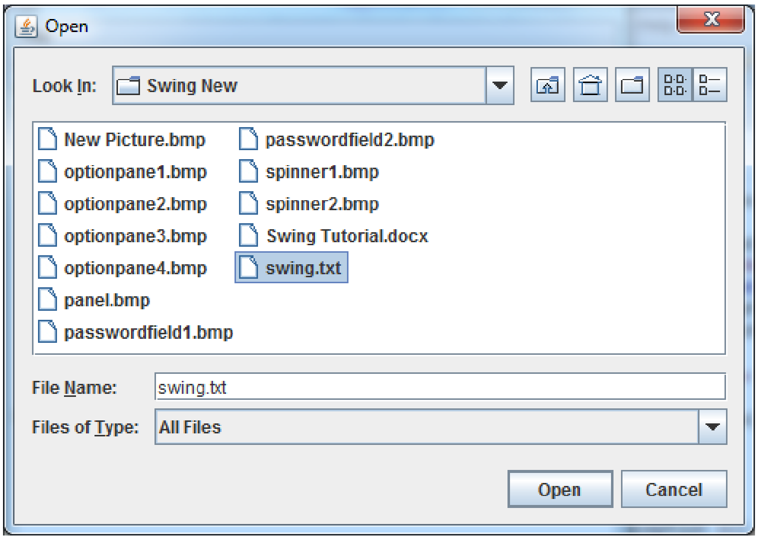
```
public class JFileChooser extends JComponent implements Accessible
```

JFileChooser Example

```
import javax.swing.*;
import java.awt.event.*;
import java.io.*;
public class FileChooserExample extends JFrame implements ActionListener{
JMenuBar mb;
JMenu file;
JMenuItem open;
JTextArea ta;
FileChooserExample(){
open=new JMenuItem("Open File");
open.addActionListener(this);
file=new JMenu("File");
file.add(open);
mb=new JMenuBar();
mb.setBounds(0,0,800,20);
mb.add(file);
ta=new JTextArea(800,800);
ta.setBounds(0,20,800,800);
add(mb);
add(ta);
}
public void actionPerformed(ActionEvent e) {
if(e.getSource()==open){
   JFileChooser fc=new JFileChooser();
   int i=fc.showOpenDialog(this);
   if(i==JFileChooser.APPROVE_OPTION){
      File f=fc.getSelectedFile();
      String filepath=f.getPath();
      try{
      BufferedReader br=new BufferedReader(new FileReader(filepath));
      String s1="",s2="";
      while((s1=br.readLine())!=null){
      s2+=s1+"\n";
      }
      ta.setText(s2);
      br.close();
      }catch (Exception ex) {ex.printStackTrace();  }
   }    }    }
public static void main(String[] args) {
   FileChooserExample om=new FileChooserExample();
         om.setSize(500,500);
         om.setLayout(null);
         om.setVisible(true);
         om.setDefaultCloseOperation(EXIT_ON_CLOSE);
}    }
```

Output:



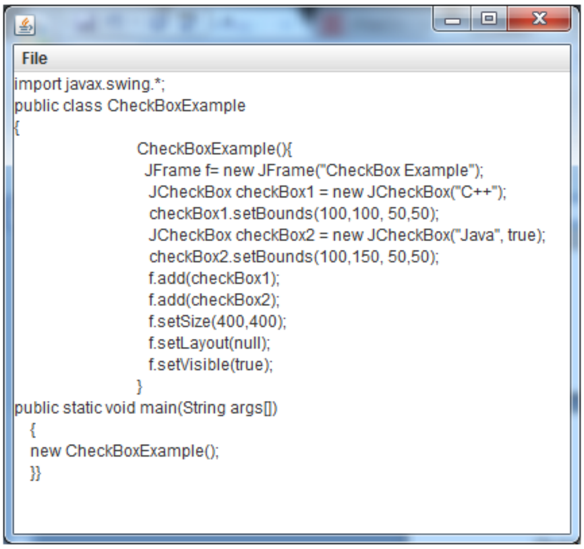


Look In: Swing New

New Picture.bmp
optionpane1.bmp
optionpane2.bmp
optionpane3.bmp
optionpane4.bmp
panel.bmp
passwordfield1.bmp
passwordfield2.bmp
spinner1.bmp
spinner2.bmp
Swing Tutorial.docx
swing.txt

File Name: swing.txt

Files of Type: All Files



```
import javax.swing.*;
public class CheckBoxExample
{
                CheckBoxExample(){
                 JFrame f= new JFrame("CheckBox Example");
                  JCheckBox checkBox1 = new JCheckBox("C++");
                  checkBox1.setBounds(100,100, 50,50);
                  JCheckBox checkBox2 = new JCheckBox("Java", true);
                  checkBox2.setBounds(100,150, 50,50);
                  f.add(checkBox1);
                  f.add(checkBox2);
                  f.setSize(400,400);
                  f.setLayout(null);
                  f.setVisible(true);
                }
public static void main(String args[])
  {
  new CheckBoxExample();
  }}
```

JTextArea:

The object of a JTextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

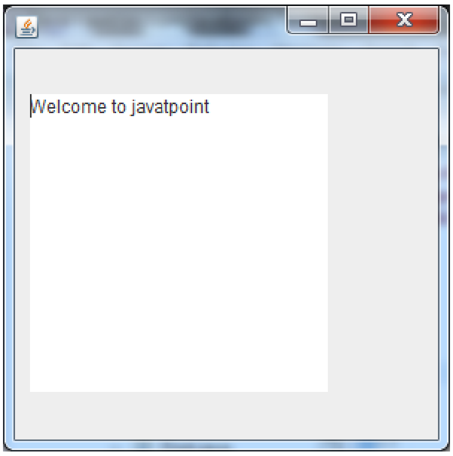
JTextArea class declaration

public class JTextArea **extends** JTextComponent

JTextArea Example

```
import javax.swing.*;
public class TextAreaExample
{
     TextAreaExample(){
        JFrame f= new JFrame();
        JTextArea area=new JTextArea("Welcome to javatpoint");
        area.setBounds(10,30, 200,200);
        f.add(area);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
    {
   new TextAreaExample();
    }}
```

Output:



Welcome to javatpoint

*****

JList and JScroll Pane:

Java JList

The object of JList class represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items. It inherits JComponent class.

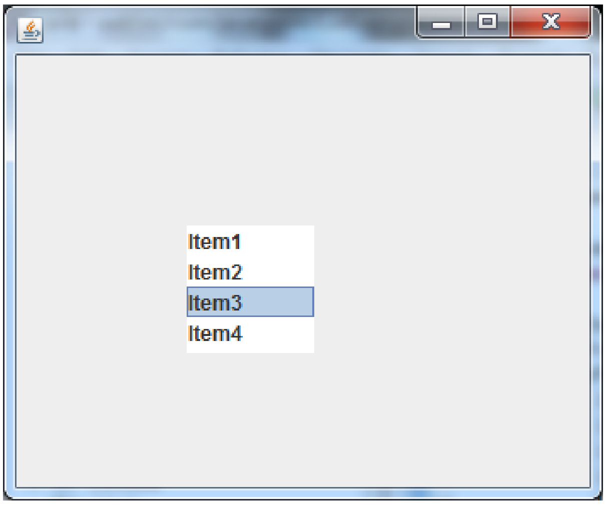
JList class declaration

public class JList **extends** JComponent **implements** Scrollable, Accessible

JList Example

```
import javax.swing.*;
public class ListExample
{
     ListExample(){
        JFrame f= new JFrame();
        DefaultListModel<String> l1 = new DefaultListModel<>();
          l1.addElement("Item1");
          l1.addElement("Item2");
          l1.addElement("Item3");
          l1.addElement("Item4");
          JList<String> list = new JList<>(l1);
          list.setBounds(100,100, 75,75);
          f.add(list);
          f.setSize(400,400);
          f.setLayout(null);
          f.setVisible(true);
     }
public static void main(String args[])
    {
   new ListExample();
    }}
```

Output:



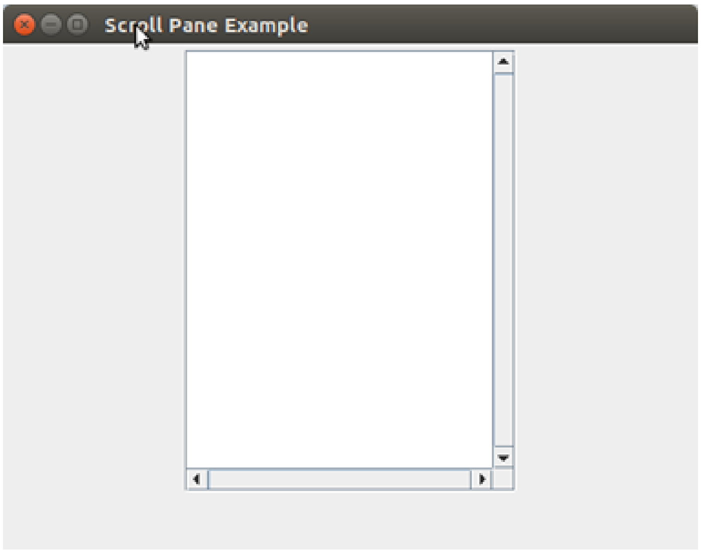
JScroll Pane:

A JscrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.

JScrollPane Example

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JtextArea;

public class JScrollPaneExample {
   private static final long serialVersionUID = 1L;

   private static void createAndShowGUI() {

      // Create and set up the window.
      final JFrame frame = new JFrame("Scroll Pane Example");

      // Display the window.
      frame.setSize(500, 500);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // set flow layout for the frame
      frame.getContentPane().setLayout(new FlowLayout());

      JTextArea textArea = new JTextArea(20, 20);
      JScrollPane scrollableTextArea = new JScrollPane(textArea);

      scrollableTextArea.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_
ALWAYS);
      scrollableTextArea.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALW
AYS);

      frame.getContentPane().add(scrollableTextArea);
   }
   public static void main(String[] args) {


      javax.swing.SwingUtilities.invokeLater(new Runnable() {

         public void run() {
            createAndShowGUI();
         }
      });
   }
}
```

Output:



*****

Split Pane:

JSplitPane is used to divide two components. The two components are divided based on the look and feel implementation, and they can be resized by the user. If the minimum size of the two components is greater than the size of the split pane, the divider will not allow you to resize it.

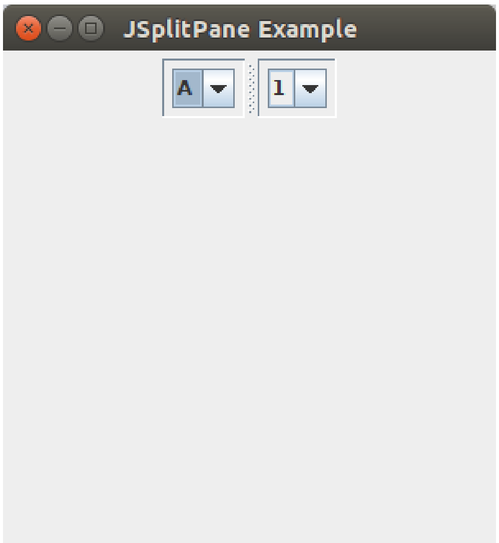
The two components in a split pane can be aligned left to right using JSplitPane.HORIZONTAL_SPLIT, or top to bottom using JSplitPane.VERTICAL_SPLIT. When the user is resizing the components the minimum size of the components is used to determine the maximum/minimum position the components can be set to.

JSplitPane Example

```
import java.awt.FlowLayout;
import java.awt.Panel;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JSplitPane;
public class JSplitPaneExample {
   private static void createAndShow() {
      // Create and set up the window.
      final JFrame frame = new JFrame("JSplitPane Example");
      // Display the window.
      frame.setSize(300, 300);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // set flow layout for the frame
      frame.getContentPane().setLayout(new FlowLayout());
      String[] option1 = { "A","B","C","D","E" };
      JComboBox box1 = new JComboBox(option1);
      String[] option2 = {"1","2","3","4","5"};
      JComboBox box2 = new JComboBox(option2);
      Panel panel1 = new Panel();
```

```
        panel1.add(box1);
        Panel panel2 = new Panel();
        panel2.add(box2);
        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, panel1, panel2);
        // JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
        // panel1, panel2);
        frame.getContentPane().add(splitPane);
    }
    public static void main(String[] args) {
        // Schedule a job for the event-dispatching thread:
        // creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShow();  }
        });
    }  }
```

Output:



*****

JTabbedPane:

The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class.

JTabbedPane class declaration

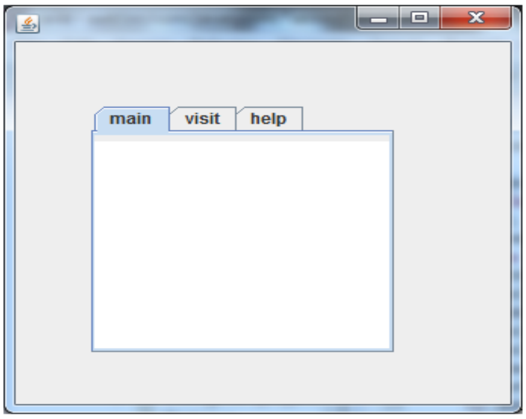
public class JTabbedPane **extends** JComponent **implements** Serializable, Accessible, SwingConstants

JTabbedPane Example

```
import javax.swing.*;
public class TabbedPaneExample {
```

```
JFrame f;
TabbedPaneExample(){
   f=new JFrame();
   JTextArea ta=new JTextArea(200,200);
   JPanel p1=new JPanel();
   p1.add(ta);
   JPanel p2=new JPanel();
   JPanel p3=new JPanel();
   JTabbedPane tp=new JTabbedPane();
   tp.setBounds(50,50,200,200);
   tp.add("main",p1);
   tp.add("visit",p2);
   tp.add("help",p3);
   f.add(tp);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);   }
public static void main(String[] args) {
   new TabbedPaneExample();
}}
```

Output:



*****

Dialog Box Pluggable Look and Feel:

Swing is **GUI Widget Toolkit** for Java. It is an API for providing Graphical User Interface to Java Programs. Unlike AWT, Swing components are written in Java and therefore are platform-independent. Swing provides platform specific Look and Feel and also an option for pluggable Look and Feel, allowing application to have Look and Feel independent of underlying platform. Initially there were very few options for colors and other settings in Java Swing, that made the entire application look boring and monotonous. With the growth in Java framework, new changes were introduced to make the UI better and thus giving developer opportunity to enhance the look of a Java Swing Application.

“Look” refers to the appearance of GUI widgets and “feel” refers to the way the widgets behave.

Sun’s JRE provides the following L&Fs:

1. **CrossPlatformLookAndFeel:** this is the “Java L&F” also known as “Metal” that looks the same on all platforms. It is part of the Java API (javax.swing.plaf.metal) and is the default.
2. **SystemLookAndFeel:** here, the application uses the L&F that is default to the system it is running on. The System L&F is determined at runtime, where the application asks the system to return the name of the appropriate L&F.
   For Linux and Solaris, the System L&Fs are “GTK+” if GTK+ 2.2 or later is installed, “Motif” otherwise. For Windows, the System L&F is “Windows”.
3. **Synth:** the basis for creating your own look and feel with an XML file.
4. **Multiplexing:** a way to have the UI methods delegate to a number of different look and feel implementations at the same time.

*****