Unit 1

Java Features

Java is one of the most popular and widely used programming language.

- Java has been one of the most popular programming language for many years.
- Java is Object Oriented.
- The Java codes are first compiled into byte code (machine independent code). Then the byte code is run on **J**ava **V**irtual **M**achine (JVM) regardless of the underlying architecture.
- Java syntax is similar to C/C++. But, Java codes are always written in the form of classes and objects.
- Java is used in all kind of applications like Mobile Applications (Android is Java based), desktop applications, web applications, client server applications, enterprise applications and many more.
- When compared with C++, Java codes are generally more maintainable because Java does not allow many things which may lead bad/inefficient programming if used incorrectly. For example, non-primitives are always references in Java. So we cannot pass large objects (like we can do in C++) to functions, we always pass references in Java.
- When compared with Python, Java kind of fits between C++ and Python. The programs written in Java typically run faster than corresponding Python programs and slower than C++. Like C++, Java does static type checking, but Python does not.

Comparision of Java with C and C++

C,C++ and Java all are the programming Languages.

- C is a procedural language,c++ is a object oriented language .Java is a pure object oriented language.
- We can create our own package in Java(set of classes) but not in c and c++.
- Internet programming like Frame,Applet is used in Java not in C,C++.
- Java uses compiler and interpreter but in C & C++ uses compiler only.
- We use multiple inheritance in C++ not in Java .In Java we use Interface instead of multiple inheritance. In c there is no inheritance.
- C & C++ both are platform dependent that means you can't run the execute code in any other operating system.Java is a platform independent language.
- In C we use stdio.h header file .In C++ we use iostream.h,conio.h headerfile but Java does not support any header files.
- Pointers are used in C and C++ language, but Java will not support for pointers.
- There is no Exception handling in C, but it supported by Java & C++.
- In C no overloading, In C++ supports overloading & in Java operator overloading not support.
- Storage classes: auto, extern are supported by C and C++, but in Java not supported.

Java and Internet

Java is fast, reliable and secure. From desktop to web applications, scientific supercomputers to gaming consoles, cell phones to the Internet, Java is used in every nook and corner.

Java Environment

Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of the system known as Java Development Kit(JDK) and the classes and methods are part of the Java Standard Library(JSL), also known as the application Programming Interface(API).

Java is one of the most popular and widely used programming language and platform. A platform is an **environment** that helps to develop and run programs written in any programming language. **Java** is easy to learn and its syntax is simple and easy to understand. It is based on C++ (so easier for programmers who know C++).

Java Program structure

A Java program involves the following sections:

- Documentation Section- $\longrightarrow$ Suggested
- Package Statement $\longrightarrow$ Optional
- Import Statements $\longrightarrow$ Optional
- Interface Statement $\longrightarrow$ Optional
- Class Definition $\longrightarrow$ Optional
- Main Method Class $\longrightarrow$ Essential
    - Main Method Definition

Section	Description
Documenta tion Section	You can write a comment in this section. Comments are beneficial for the programmer because they help them understand the code. These are optional, but we suggest you use them because they are useful to understand the operation of the program, so you must write comments within the program.
Package statement	You can create a package with any name. A package is a group of classes that are defined by a name. That is, if you want to declare many classes within one element, then you can declare it within a package. It is an optional part of the program, i.e., if you do not want to declare any package, then there will be no problem with it, and you will not get any errors. Here, the package is a keyword that tells the compiler that package has been created. It is declared as: `package package_name;`
Import statements	This line indicates that if you want to use a class of another package, then you can do this by importing it directly into your program. Example: `import calc.add;`
Interface statement	Interfaces are like a class that includes a group of method declarations. It's an optional section and can be used when programmers want to implement multiple inheritances within a program.
Class Definition	A Java program may contain several class definitions. Classes are the main and essential elements of any Java program.
Main Method Class	Every Java stand-alone program requires the main method as the starting point of the program. This is an **essential part of a Java program**. There may be many classes in a Java program, and only one class defines the main method. Methods contain data type declaration and executable statements.

Java Tokens

A **token** is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

1. Keywords
2. Identifiers
3. Literals
4. Special Symbols (Separators)
5. Operators

Keyword: Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. **Java** language supports following keywords:

abstract	**assert**	**boolean**
break	**byte**	**case**
catch	**char**	**class**
const	**continue**	**default**
do	**double**	**else**
enum	**exports**	**extends**
final	**finally**	**float**
for	**goto**	**if**
implements	**import**	**instanceof**
int	**interface**	**long**
module	**native**	**new**
open	**opens**	**package**
private	**protected**	**provides**
public	**requires**	**return**
short	**static**	**strictfp**
super	**switch**	**synchronized**
this	**throw**	**throws**
to	**transient**	**transitive**
try	**uses**	**void**
volatile	**while**	

2. Identifiers: Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character. Identifier names must differ in spelling and case from any keywords. You cannot use keywords as identifiers; they are reserved for special use. Once declared, you can use the identifier in later program statements to refer to the associated value. A special kind of identifier, called a statement label, can be used in goto statements.

3. **Constants/Literals:** Constants are also like normal variables. But, the only difference is, their values can not be modified by the program once they are defined. Constants refer to fixed values. They are also called as literals.

Constants may belong to any of the data type.

Syntax:
final data_type variable_name;

4. **Special Symbols:** The following special symbols are used in Java having some special meaning and thus, cannot be used for some other purpose.

   [] () {}, ; * =

   1. **Brackets[]:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.
   2. **Parentheses():** These special symbols are used to indicate function calls and function parameters.
   3. **Braces{}:** These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.
   4. **comma (,):** It is used to separate more than one statements like for separating parameters in function calls.
   5. **semi colon :** It is an operator that essentially invokes something called an initialization list.
   6. **asterick (*):** It is used to create pointer variable.
   7. **assignment operator:** It is used to assign values.

5. **Operators:** Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-
   1. Arithmetic Operators
   2. Unary Operators
   3. Assignment Operator
   4. Relational Operators
   5. Logical Operators
   6. Ternary Operator
   7. Bitwise Operators
   8. instance of operator

Implementing a Java Program

Implementation of a Java application program involves a series of steps. They include :

- **Creating the program**
- **Compiling the program**
- **Running the program**

Creating the program

We can create a program using any text editor.

We must save this program in a file called **Javaapp.java** ensuring that the file name contains the class name properly. This file is called the **source file**. Note that all Java source files will have the extension **java**.

Compiling the program

To compile the program, we must run the Java Compiler **javac**, with the name of the source file on the command line
Running the program
To run the program, we must run the Java interpreter **java**, with the name of the class file on the command line

Java Virtual Machine

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

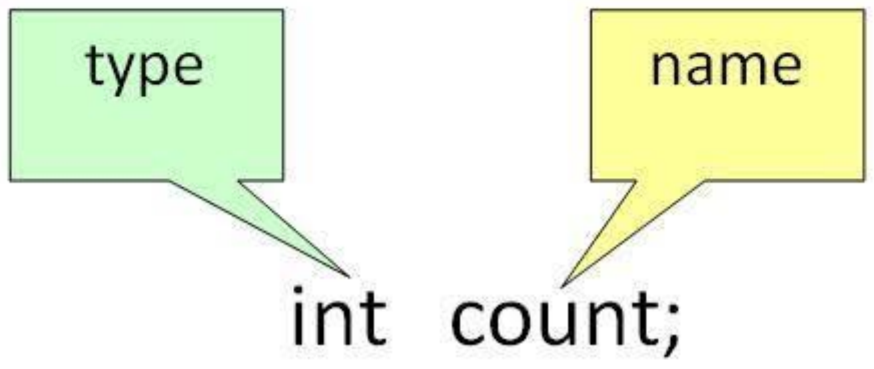
Constants
A **constant** is a name or an identifier for a fixed value. **Constant** are like variables, except that once they are defined, they cannot be undefined or changed (except magic **constants**). **Constants** are very useful for storing data that doesn't change while the script is running.

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.

We can declare variables in java as follows:

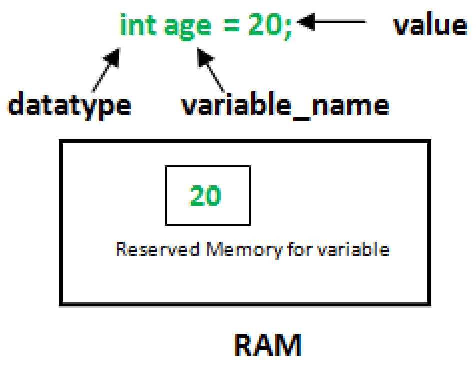


type: Type of data that can be stored in this variable.

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



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.

Data Types

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

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

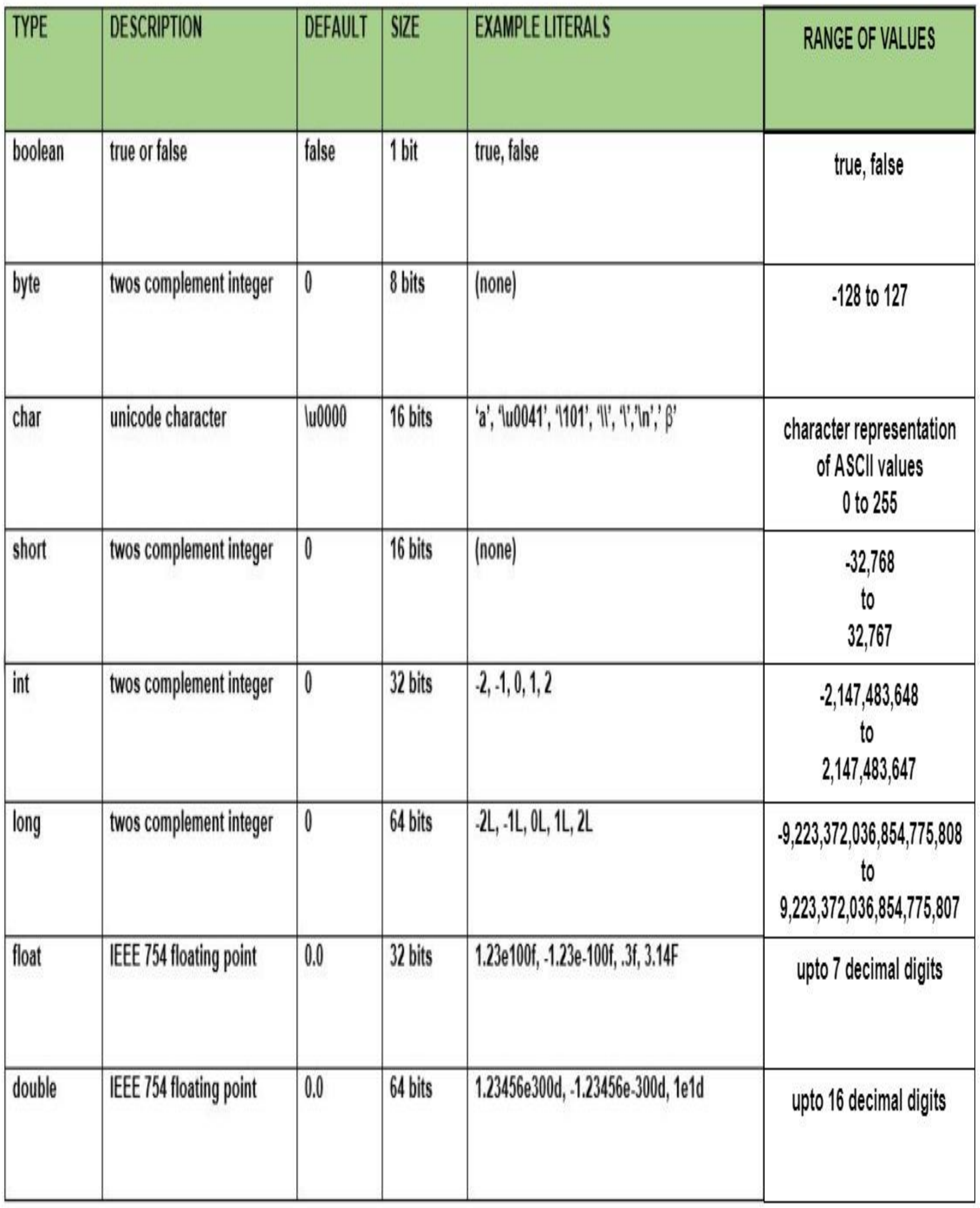
Java Primitive Data Types

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

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type

- float data type
- double data type



TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	‘a’, ‘\u0041’, ‘\101’, ‘\\’, ‘\’,’\n’,’ β’	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

Scope of Variables

Scope refers to the visibility of **variables**. In other words, which parts of your program can see or use it. Normally, every **variable** has a global **scope**. Once defined, every part of your program can access a **variable**. It is very useful to be able to limit a **variable's scope** to a single function.

Type casting

Type casting in Java is to **cast** one **type**, a class or interface, into another **type** i.e. another class or interface. ... **Type casting** also comes with the risk of ClassCastException in **Java**, which is quite common with a method which accepts Object **type** and later **types cast** into more specific **type**.

Operators and Expressions

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-

1. Arithmetic Operators
2. Increment and Decrement Operator
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Conditional Operator (Ternary Operator)
7. Bitwise Operators
8. instance of operator

Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

- *** :** Multiplication
- **/ :** Division
- **% :** Modulo
- **+ :** Addition
- **– :** Subtraction

Increment and Decrement Operator

- **– :Unary minus**, used for negating the values.
- **+ :Unary plus**, used for giving positive values. Only used when deliberately converting a negative value to positive.
- **++ :Increment operator**, used for incrementing the value by 1. There are two varieties of increment operator.
    - **Post-Increment :** Value is first used for computing the result and then incremented.
    - **Pre-Increment :** Value is incremented first and then result is computed.
- **— : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operator.
    - **Post-decrement :** Value is first used for computing the result and then decremented.
    - **Pre-Decrement :** Value is decremented first and then result is computed.

Assignment Operator : '=' Assignment operator is used to assign a value to any variable. It has a right to left associativity, i.e value given on right hand side of operator is assigned to the variable on the left and therefore right hand side value must be declared before using it or should be a constant. General format of assignment operator is,

```
variable = value;
```

In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a = a+5, we can write a **+=** 5.

- **+=**, for adding left operand with right operand and then assigning it to variable on the left.
- **-=**, for subtracting left operand with right operand and then assigning it to variable on the left.
- ***=**, for multiplying left operand with right operand and then assigning it to variable on the left.
- **/=**, for dividing left operand with right operand and then assigning it to variable on the left.
- **%=**, for assigning modulo of left operand with right operand and then assigning it to variable on the left.

```
int a = 5;

a += 5; //a = a+5;
```

Relational Operators : These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements. General format is,

variable **relation_operator** value

Some of the relational operators are-

- **==, Equal to :** returns true of left hand side is equal to right hand side.
- **!=, Not Equal to :** returns true of left hand side is not equal to right hand side.
- **<, less than :** returns true of left hand side is less than right hand side.
- **<=, less than or equal to :** returns true of left hand side is less than or equal to right hand side.
- **>, Greater than :** returns true of left hand side is greater than right hand side.
- **>=, Greater than or equal to:** returns true of left hand side is greater than or equal to right hand side.

Logical Operators : These operators are used to perform "logical AND" and "logical OR" operation, i.e. the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Conditional operators are-

- **&&, Logical AND :** returns true when both conditions are true.
- **||, Logical OR :** returns true if at least one condition is true.
- **! : Logical not operator**, used for inverting a boolean value.

Ternary operator : Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary. General format is-

- condition **?** if true **:** if false
- The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

Bitwise Operators : These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit by bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inversed.
- **<< shift left**
- **>> shift right**
- **>>> shift right with zero fill**

Instance of operator : Instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass or an interface. General format-

- object **instance of** class/subclass/interface

Expressions

Expressions are essential building blocks of any **Java** program, usually created to produce a new value, although sometimes an **expression** assigns a value to a variable. Expressions are built using values, variables, operators and method calls.

Decision Making,Branching and Looping

Decision Making in Java (if, if-else, switch, break, continue, jump)

Decision Making in programming is similar to decision making in real life. In programming also we face some situations where we want a certain block of code to be executed when some condition is fulfilled.
A programming language uses control statements to control the flow of execution of program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

Java's Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

Branching

Java provides three branching statements break, continue and return. The break and continue in Java are two essential keyword beginners needs to familiar while using loops (for loop, while loop and do while loop). break statement in java is used to break the loop and transfers control to the line immediate outside of loop while continue is used to escape current execution (iteration) and transfers control back to the start of the loop. Both break and continue allow the programmer to create sophisticated algorithm and looping constructs.

Looping

Java provides three repetition statements/looping statements that enable programmers to control the flow of execution by repetitively performing a set of statements as long as the continuation condition remains true. These three looping statements are called *for*, *while*, and *do...while* statements. The *for* and *while* statements perform the repetition declared in their body **zero or more times**. If the loop continuation condition is *false*, it stops execution. The *do...while* loop is slightly different in the sense that it executes the statements within its body for **one or more times**.

There is a common structure of all types of loops, such as:

- There is a **control variable**, called the **loop counter**.
- The control variable must be **initialized**; in other words, it must have an initial value.
- The **increment/decrement** of the control variable, which is modified each time the iteration of the loop occurs.
- The **loop condition** that determines if the looping should continue or the program should break from it.

Unit II

CLASSES AND ARRAYS

Defining a Class

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 (Refer this for details).
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **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.
4. **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.
5. **Body:** The class body surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

There are various types of classes that are used in real time applications such as nested classes, anonymous classes, lambda expressions.

Constructots

Constructors are used to initialize the object's state. Like methods, a constructor also contains **collection of statements(i.e. instructions)** that are executed at time of Object creation.

Need of Constructor

Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in computer's memory), then can a box be there with no value defined for its dimensions. The answer is no.
So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

Constructor called

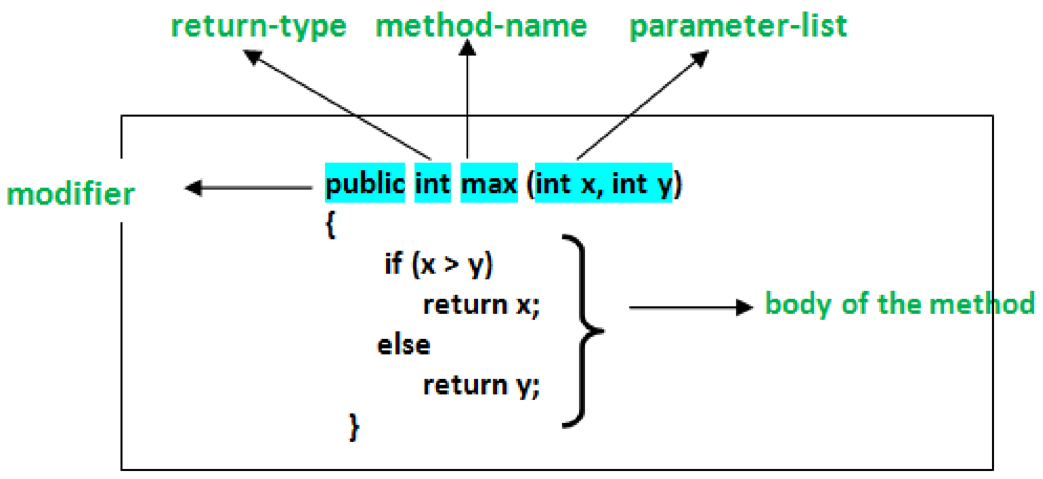
Each time an object is created using **new()** keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the **data members** of the same class.

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 :

- **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.
  - public: accessible in all class in your application.
  - protected: accessible within the class in which it is defined and in its **subclass(es)**
  - private: accessible only within the class in which it is defined.
  - default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.
- **The return type** : The data type of the value returned by the method or void if does not return a value.
- **Method Name** : the rules for field names apply to method names as well, but the convention is a little different.
- **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 ().
- **Exception list** : The exceptions you expect by the method can throw, you can specify these exception(s).
- **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;
}
```

Overloading

Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.

Static Members

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.

Nesting of Methods

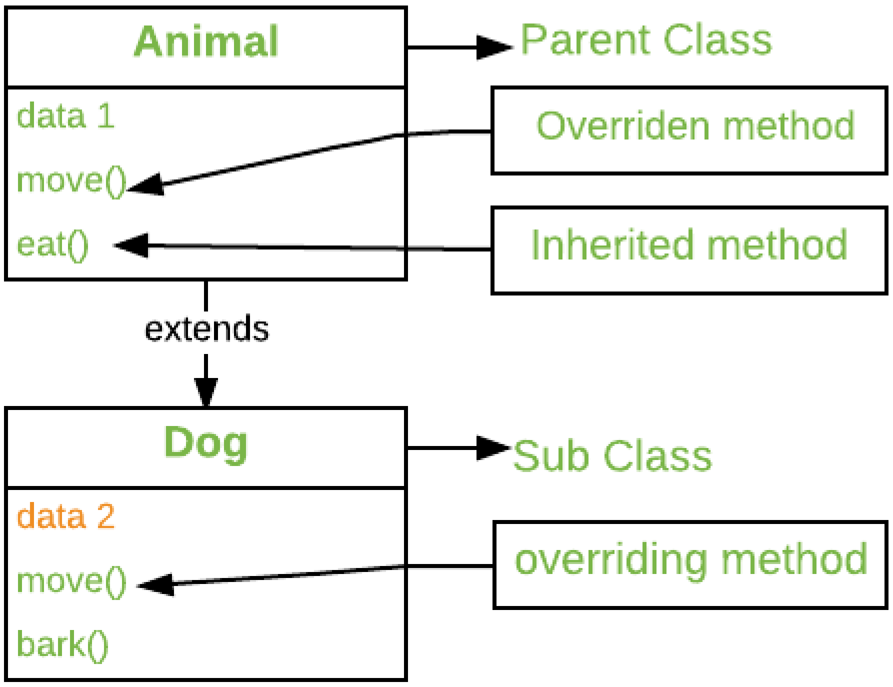
When a method in java calls another method in the same class, it is called Nesting of methods.

Example:

Enter length, breadth and height as input. After that we first call the volume method. From volume method we call area method and from area method we call perimeter method. Hence we get perimeter, area and volume of cuboid as output.

Overriding Method

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.



Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

Final Classes

A **final class** is a **class** that can't be extended. Also methods could be declared as **final** to indicate that cannot be overridden by subclasses. Preventing the **class** from being subclassed could be particularly useful if you write APIs or libraries and want to avoid being extended to alter base behaviour.

Abstract Classes

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

```
// An example abstract class in Java
abstract class Shape {
    int color;

    // An abstract function (like a pure virtual function in C++)
    abstract void draw();
}
```

Visibility Control

Java access modifiers are known as Visibility Control in Java. Modifiers are used to define where members, function and class can be used and where these can't be accessed .

Java supports 13 modifiers. These can be divided into 3 groups.
Accessibility modifiers
1. private
2. default
3. protected
4. public

Execution level modifiers

5. static
6. final
7. abstract
8. native
9. transient
10. volatile
11. synchronized
12. strictfp

File level modifiers

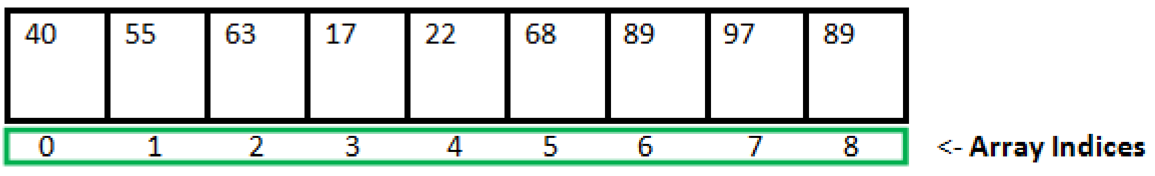
13. interface

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 point about Java arrays.

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using member 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 value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Array can contains primitives data types as well as objects of a class depending on the definition of array. In case of primitives data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.



Array Length = 9
First Index = 0
Last Index = 8

Creating an Array

One-Dimensional Arrays :

The general form of a one-dimensional array declaration is

```
type var-name[];

OR

type[] var-name;
```

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like array of int type, we can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element type for the array determines what type of data the array will hold.

Two Dimensional Arrays

The Two Dimensional Array in Java programming language is nothing but an Array of Arrays. In Java Two Dimensional Array, data stored in row and columns, and we can access the record using both the row index and column index

If the data is linear, we can use the One Dimensional Array. However, to work with multi-level data, we have to use the Multi-Dimensional Array. Two Dimensional Array in Java is the simplest form of Multi-Dimensional Array.

Two Dimensional Array Declaration

The following code snippet shows the two dimensional array declaration in Java Programming Language:

```
Data_Type[][] Array_Name;
```

- **Data_type:** It decides the type of elements it will accept. For example, If we want to store integer values, then the Data Type will be declared as int. If we want to store Float values, then the Data Type will be float.

- **Array_Name:** This is the name to give it to this Java two dimensional array. For example, Car, students, age, marks, department, employees, etc.

Strings

Strings in Java are Objects that are backed internally by a char array. Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created.

Below is the basic syntax for declaring a string in **Java programming** language.

Syntax:
<String_Type> <string_variable> = "<sequence_of_string>";

Example:
```
String str = "India";
```

String Arrays

A Java String Array is an object that holds a fixed number of String values. Arrays in general is a very useful and important data structure that can help solve many types of problems. It's simplicity on how to access contents through index makes it powerful yet user-friendly. Here are some examples on how to use String Array in Java.

String Array Declaration
Square brackets is used to declare a String array. There are two ways of using it. The first one is to put square brackets after the String reserved word. For example:

```
String[] thisIsAStringArray;
```

Another way of declaring a String Array is to put the square brackers after the name of the variable. For example:

```
String thisIsAStringArray[];
```

Both statements will declare the variable "thisIsAStringArray" to be a String Array. Note that this is just a declaration, the variable "thisIsAStringArray" will have the value null. And since there is only one square brackets, this means that the variable is only a one-dimensional String Array. Examples will be shown later on how to declare multi-dimensional String Arrays.

String is a sequence of characters, for e.g. "Hello" is a string of 5 characters. In java, string is an immutable object which means it is constant and can cannot be changed once it has been created. In this tutorial we will learn about String class and String methods in detail along with many other Java String tutorials.

Creating a String

There are two ways to create a String in Java

1. String literal
2. Using new keyword

String literal

In java, Strings can be created like this: Assigning a String literal to a String instance:

```
String str1 = "Welcome";
String str2 = "Welcome";
```

The problem with this approach: As I stated in the beginning that String is an object in Java. However we have not created any string object using new keyword above. The compiler does that task for us it creates a string object having the string literal (that we have provided , in this case it is “Welcome”) and assigns it to the provided string instances.

But if the object already exist in the memory it does not create a new Object rather it assigns the same old object to the new instance, that means even though we have two string instances above(str1 and str2) compiler only created on string object (having the value “Welcome”) and assigned the same to both the instances. For example there are 10 string instances that have same value, it means that in memory there is only one object having the value and all the 10 string instances would be pointing to the same object.

What if we want to have two different object with the same string? For that we would need to create strings using **new keyword**.

Using New Keyword

As we saw above that when we tried to assign the same string object to two different literals, compiler only created one object and made both of the literals to point the same object. To overcome that approach we can create strings like this:

```
String str1 = new String("Welcome");
String str2 = new String("Welcome");
```

In this case compiler would create two different object in memory having the same string.

String Methods

Here are the list of the methods available in the Java String class. These methods are explained in the separate tutorials with the help of examples. Links to the tutorials are provided below:

1. char charAt(int index): It returns the character at the specified index. Specified index value should be between 0 to length() -1 both inclusive. It throws IndexOutOfBoundsException if index<0||>= length of String.
2. boolean equals(Object obj): Compares the string with the specified string and returns true if both matches else false.
3. boolean equalsIgnoreCase(String string): It works same as equals method but it doesn't consider the case while comparing strings. It does a case insensitive comparison.
4. int compareTo(String string): This method compares the two strings based on the Unicode value of each character in the strings.
5. int compareToIgnoreCase(String string): Same as CompareTo method however it ignores the case during comparison.
6. boolean startsWith(String prefix, int offset): It checks whether the substring (starting from the specified offset index) is having the specified prefix or not.
7. boolean startsWith(String prefix): It tests whether the string is having specified prefix, if yes then it returns true else false.
8. boolean endsWith(String suffix): Checks whether the string ends with the specified suffix.
9. int hashCode(): It returns the hash code of the string.

StringBuffer class

StringBuffer is a peer class of **String** that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.
StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
StringBuffer Constructors
StringBuffer( ): It reserves room for 16 characters without reallocation.

```
StringBuffer s=new StringBuffer();
```

StringBuffer(int size) It accepts an integer argument that explicitly sets the size of the buffer.

```
StringBuffer s=new StringBuffer(20);
```

StringBuffer(String str): It accepts a **String** argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

```
StringBuffer s=new StringBuffer("GeeksforGeeks");
```

Methods
Some of the most used methods are:

- **length( ) and capacity( ):** The length of a StringBuffer can be found by the length( ) method, while the total allocated capacity can be found by the capacity( ) method.

Vectors

The **vector** class implements a growable array of objects. Like an array, it contains the component that can be accessed using an integer index. **Vector** is very useful if we don't know the size of an array in advance or we need one that can change the size over the lifetime of a **program**.

Wrapper classes

A **Wrapper class** is a **class** whose object wraps or contains a primitive data types. When we create an object to a **wrapper class**, it contains a field and in this field, we can store a primitive data types. In other words, we can **wrap** a primitive value into a **wrapper class** object.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

Unit III

INHERITANCE,INTERFACES AND PACKAGES

Defining a subclass

A class that is derived from another class is called a **subclass** (also a derived class, extended class, or child class). The class from which the **subclass** is derived is called a superclass (also a base class or a parent class).

A subclass is defined as follows

```
class subclassname extnds superclassname

{

variables declaration;

method declaration;
```

The keyword extends signifies that the properties of the superclassname are extended to the subclassname.

Subclass Constructor

A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword **super** to invoke the constructor method of the superclass. The keyword super is used subject to the following condition

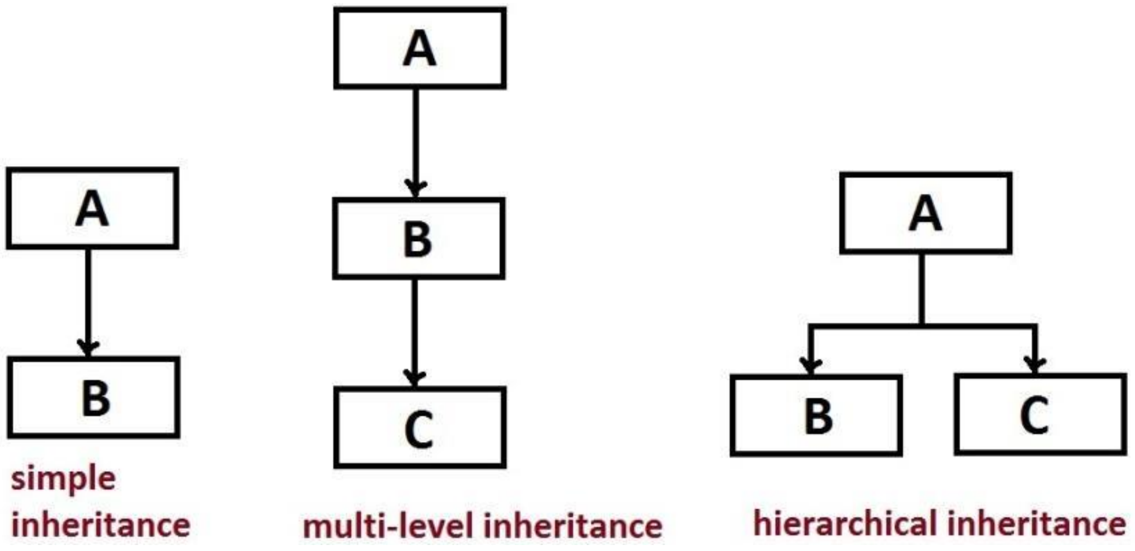
1. **super** may only be used within a subclass constructor method.
2. The call to superclass constructor must appear as the first statement within the subclass constructor.
3. The parameters in the **super** call must match the order and type of the instance variable declared in the superclass.

Multilevel Inheritance

In Java (and in other object-oriented languages) a class can get features from another class. This mechanism is known as **inheritance.**

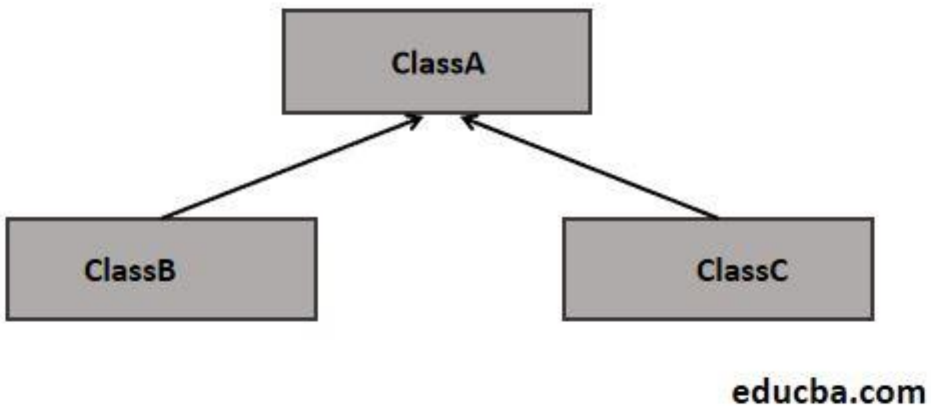
- When multiple classes are involved and their parent-child relation is formed in a chained way then such formation is known as multi-level inheritance.
- In multilevel inheritance, a parent a class has a maximum of one direct child class only.
- In multi-level inheritance, the inheritance linkage is formed in a linear way and minimum 3 classes are involved.

Code re-usability can be extended with multi-level inheritance.



Hierarchical Inheritance

Hierarchical Inheritance in Java is one of the types of inheritance in java. Inheritance is one of the important features of an Object-Oriented programming system (oops). An inheritance is a mechanism in which one class inherits or acquires all the attributes and behaviors of the other class. The class from which inherits the attributes and behaviors are called parent or super or base class and the class which inherits the attributes and behaviors are called child or derived class. In Hierarchical Inheritance, the multiple child classes inherit the single class or the single class is inherited by multiple child class. The use of inheritance in Java is for the reusability of code and for the dynamic polymorphism (method overriding). We can understand the Hierarchical Inheritance more clearly with the help of the below diagram.



As in the above example figure, the ClassB and ClassC inherit the same or single class ClassA. So the ClassA variables and methods are reuse in both classes, ClassB and ClassC. As the above diagram showing that more than one child classes have the same parent class, so this type of inheritance is called Hierarchical Inheritance.

Defining Interface

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.

- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface

Extending Interfaces

An **interface** can **extend** another **interface** in the same way that a class can **extend** another class. The **extends** keyword is used to **extend an interface**, and the child **interface** inherits the methods of the parent **interface**. The following Sports **interface** is **extended** by Hockey and Football **interfaces**.

Implementing Interfaces

An interface is just like Java Class, but it only has static constants and abstract method. Java uses Interface to implement multiple inheritance. A Java class can implement multiple Java Interfaces. All methods in an interface are implicitly public and abstract.

Syntax for Declaring Interface

```
interface {
//methods
}
```

To use an interface in your class, append the keyword "implements" after your class name followed by the interface name.

Example for Implementing Interface

```
class Dog implements Pet
interface RidableAnimal extends Animal, Vehicle
```

Java Application Programming Interface (API) Packages

Java API provides a large number of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API.

Java application programming interface (API) is a list of all classes that are part of the Java development kit (JDK). It includes all Java packages, classes, and interfaces, along with their methods, fields, and constructors. These prewritten classes provide a tremendous amount of functionality to a programmer. A programmer should be aware of these classes and should know how to use them. A complete listing of all classes in Java API can be found at Oracle's website: http://docs.oracle.com/javase/7/docs/api/. Please visit the above site and bookmark it for future reference. Please consult this site often, especially when you are using a new class and would like to know more about its methods and fields. If you browse through the list of packages in the API, you will observe that there are packages written for GUI programming, networking programming, managing input and output, database programming, and many more. Please browse the complete list of packages and their descriptions to see how they can be used. In order to use a class from Java API, one needs to include an import statement at the start of the program. For example, in order to use the Scanner class, which allows a program to accept input from the keyboard, one must include the following import statement: import java.util.Scanner; The above import statement allows the programmer to use any method listed in the Scanner class. Another choice for including the import statement is the wildcard option shown below: import java.util.*; This version of the import statement imports all the classes in the API's java.util package and makes them available to the programmer. If you check the API and look at the classes written in the java.util package, you will observe that it includes some of the classes that are used often, such as Arrays, ArrayList, Formatter, Random, and many others. Another Java package that has several commonly used classes is the java.lang package. This package includes classes that are fundamental to the design of Java language. The java.lang package is automatically imported in a Java program and does not need an explicit import statement. Please note that some of the classes that we use very early in Java programming come from this package. Commonly used classes in the java.lang package are: Double, Float, Integer, String, StringBuffer, System, and Math.

Creating a Package

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are −

- **java.lang** − bundles the fundamental classes
- **java.io** − classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* –

```
/* File name : Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

Creating a **package** in **Java** is a very easy task. Choose a name for the **package** and include a **package** command as the first statement in the **Java** source file. The **java** source file can contain the classes, interfaces, enumerations, and annotation types that you want to include in the **package**.

using packages

Packages are used in **Java** in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

Accessing a Package

A java system package can be accessed ether using a fully qualified class name or using a shortcut approach the **import** statement. we use the import statement when there are many references to a particular package or the package name is too long and wieldy.

The same approaches can be used to access the user-defined packages as well. The import statement can be used to search a list of packages for a particular class.

Syntax,

import package1 [.packag2] [.package3].**classname**;

Here **package1** is the name of the top level package, **package2** is the name of the package that is inside the package1 and so on. we can have any number of packages in a package hierarchy. finally the explicit **classname** is specified.

Adding Classes to Packages

1. Put the Java source file inside a directory matching the Java package you want to put the class in.
2. Declare that class as part of the package.

Hiding Classes

When we import a package within a program, only the classes declared as **public** in that package will be made accessible within this program. In other words, the classes not declared as public in that package will not be accessible within this program.
We shall profitably make use of the above fact. Sometimes, we may wish that certain classes in a package should not be made accessible to the importing program. In such cases, we need not declare those classes as public. When we do so, those classes will be hidden from being accessed by the importing class.

UNIT IV : MULTITHREADING, EXCEPTION HANDLING, FILES AND CREATING THREADS

Extending Thread Class:

One way to create a **thread** is to create a new **class** that **extends Thread**, and then to create an instance of that **class**. The **extending class** must override the run() method, which is the entry point for the new **thread**. It must also call start() to begin execution of the new **thread**.

Example:

```
class ExtendingThread extends Thread
        {
                String s[]={"Welcome","to","Java","Programming","Language"};
                public static void main(String args[])
                    {
                   ExtendingThread t=new ExtendingThread("Extending Thread
Class");
                    }
                       public ExtendingThread (String n)
                         {
                               super(n);
                               start();
                      }
                      public void run()
                         {
                                 String name=getName();
                                 for(int i=0;i<s.length;i++)
                                       {
                                       try
                                       {
                                           sleep(500);
                                       }
                                           catch(Exception e)
                                                {
                                                }
                                                  System.out.println(name+":"+s[i]);
                                       }
                                 }
           }
```

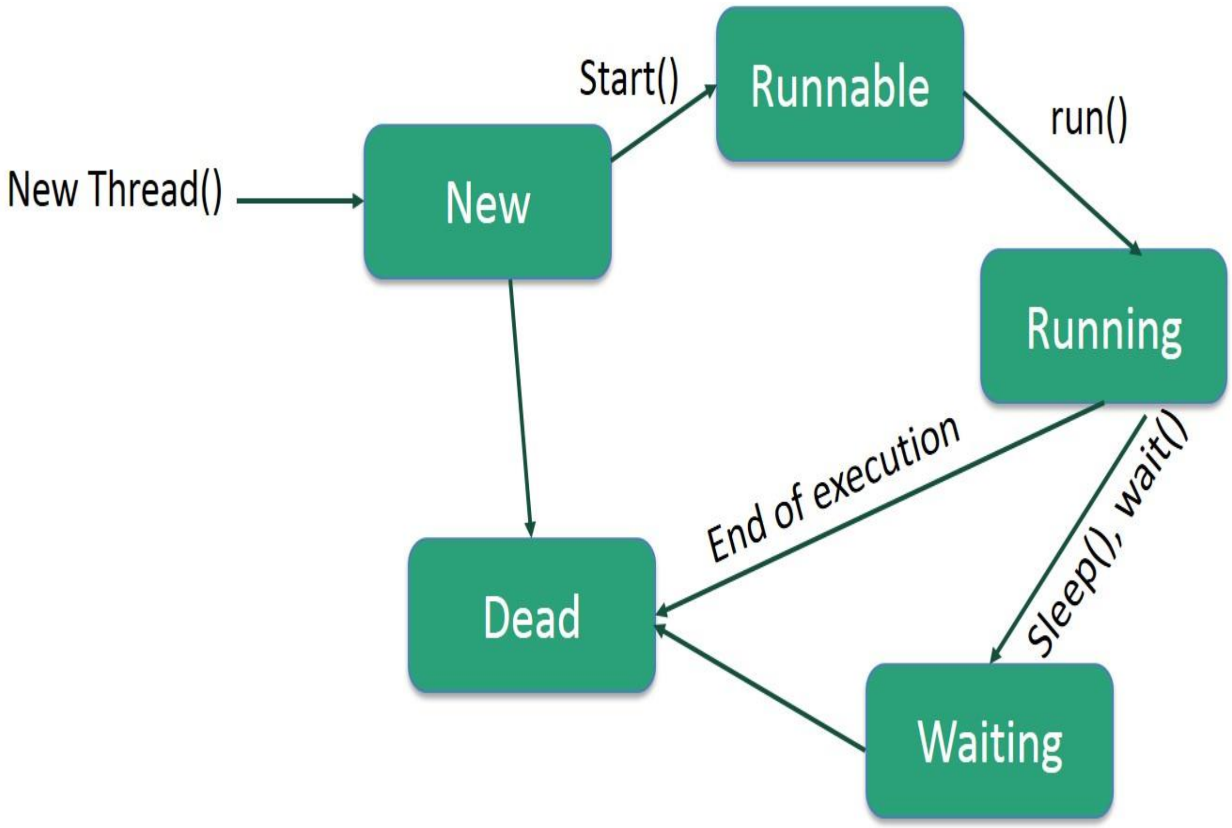
Thread Life Cycle:

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

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



Following are the stages of the life cycle –

- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Exception

Thread is the independent path of execution run inside the program. Many Thread run concurrently in the program. Multithread are those group of more than one thread that runs concurrently in a program. Thread in a program is imported from java.lang.thread class.InMultithread,the thread run concurently,synchronous or asynchronous.

Advantage of Multithread

1)Multithread are lightweight as compared to any processes.

2)Context Switching between the thread is less expensive as compared to processes.

3)Intercommunication between thread is relatively economically than processes.

4)Multithread occupy the same address and data space.

5)Thread can run independently in program.

Method in Object and Thread Class

Object	**Thread**
1)Notify( )	1)Sleep( )
2)Notify all( )	2)Yield( )
3)wait( )	

How to Create Thread

There are method to create thread

1)Extends the Threads Class(java.lang.thread)

2)Implement Runnable interface(java .lang. thread)

Understand Exception in Threads.

1.A class name RunnableThread implements the Runnable interface gives you the run( ) method executed by the thread. Object of this class is runnable

2. The Thread constructor is used to create an object of RunnableThread class by passing runnable object as parameter.. The Thread object has a Runnable object that implements the run( ) method.

3. The start( ) method is invoked on the Thread object . The start( ) method returns immediately once a thread has been spawned.

4. The thread ends when the run( ) method ends which is to be normal termination or caught exception.

5.runner = new Thread(this,threadName) is used to create a new thread

6 .runner. start( ) is used to start the new thread.

7.public void run( ) is overrideable method used to display the information of particular thread

8.Thread.currentThread().sleep(2000) is used to deactivate the thread untill the next thread started execution or used to delay the current thread.

Thread Priority

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Synchronization

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

```
// Only one thread can execute at a time.

// sync_object is a reference to an object

// whose lock associates with the monitor.
// The code is said to be synchronized on
// the monitor object
synchronized(sync_object)
{
   // Access shared variables and other
   // shared resources
}
```

Runnable interface

java.lang.Runnable is an interface that is to be implemented by a class whose instances are intended to be executed by a thread. There are two ways to start a new Thread – Subclass Thread and implement Runnable. There is no need of subclassing Thread when a task can be done by overriding only run() method of Runnable.

Steps to create a new Thread using Runnable :

1. Create a Runnable implementer and implement run() method.

2. Instantiate Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instance.

3. Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start(), creates a new Thread which executes the code written in run(). Calling run() directly doesn't create and start a new Thread, it will run in the same thread. To start a new line of execution, call start() on the thread.

Example,

```
public class RunnableDemo {

    public static void main(String[] args)
    {
        System.out.println("Main thread is- "
                + Thread.currentThread().getName());
        Thread t1 = new Thread(new RunnableDemo().new RunnableImpl());
        t1.start();
    }

    private class RunnableImpl implements Runnable {

        public void run()
        {
            System.out.println(Thread.currentThread().getName()
                    + ", executing run() method!");
        }
    }
}
```

Output:

```
Main thread is- main

Thread-0, executing run() method!
```

Output shows two active threads in the program – main thread and Thread-0, main method is executed by the Main thread but invoking start on RunnableImpl creates and starts a new thread – Thread-0.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic steps −

Step 1

As a first step, you need to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method −

```
public void run( )
```

Step 2

As a second step, you will instantiate a **Thread** object using the following constructor −

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

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

Step 3

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

```
void start();
```

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor

Exceptions

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.

- **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.

Example

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **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.

Example

```
public class Unchecked_Demo {

   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
	at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

- **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Throwing own exceptions

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

```
class MyException extends Exception {
}
```

You just need to extend the predefined Exception class to create your own Exception.

Example

Following is an exception which is thrown if the value passed is greater than 10.

```
class MyException extends Exception {
   int id;

   public MyException(int x) {
      id = x;
   }

   public String toString() {
      return "CustomException[" + id + "]";
   }
}

public class Sampleee {
   static void compute(int a) throws MyException {
      if (a > 10)
```

```
        throw new MyException(a);

        System.out.println("No error in prog. no exception caught");

  }


  public static void main(String args[]) {


     try {

        compute(5);

        compute(12);

     } catch(MyException ex1) {

        System.out.println(ex1);

     }

  }

}
```

Output

```
No error in prog. no exception caught
CustomException[12]
```

Concept of Stream

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

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

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

Byte Streams

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

Example

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

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

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

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

```
This is test for copy file.
```

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

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

Character Streams

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

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

Example

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

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

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

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

```
This is test for copy file.
```

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

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

Standard Streams

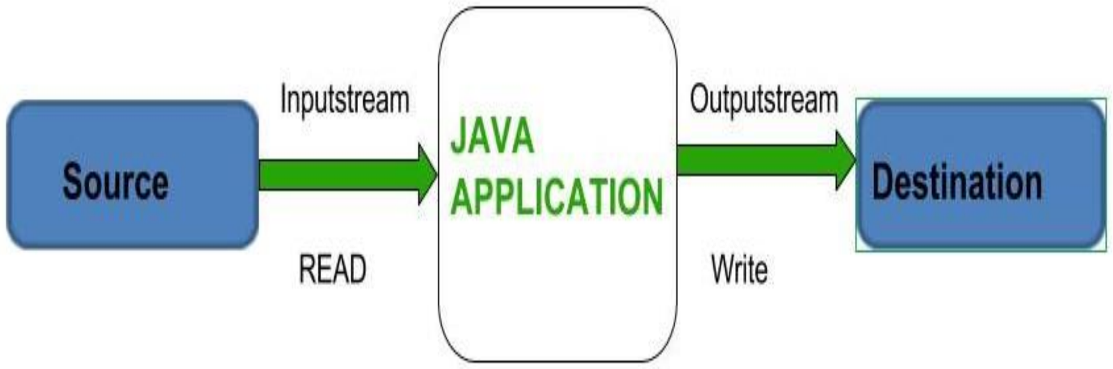
All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

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

Stream Classess

Java.io.InputStream Class in Java

InputStream class is the superclass of all the io classes i.e. representing an input stream of bytes. It represents input stream of bytes. Applications that are defining subclass of InputStream must provide method, returning the next byte of input.
A reset() method is invoked which re-positions the stream to the recently marked position.



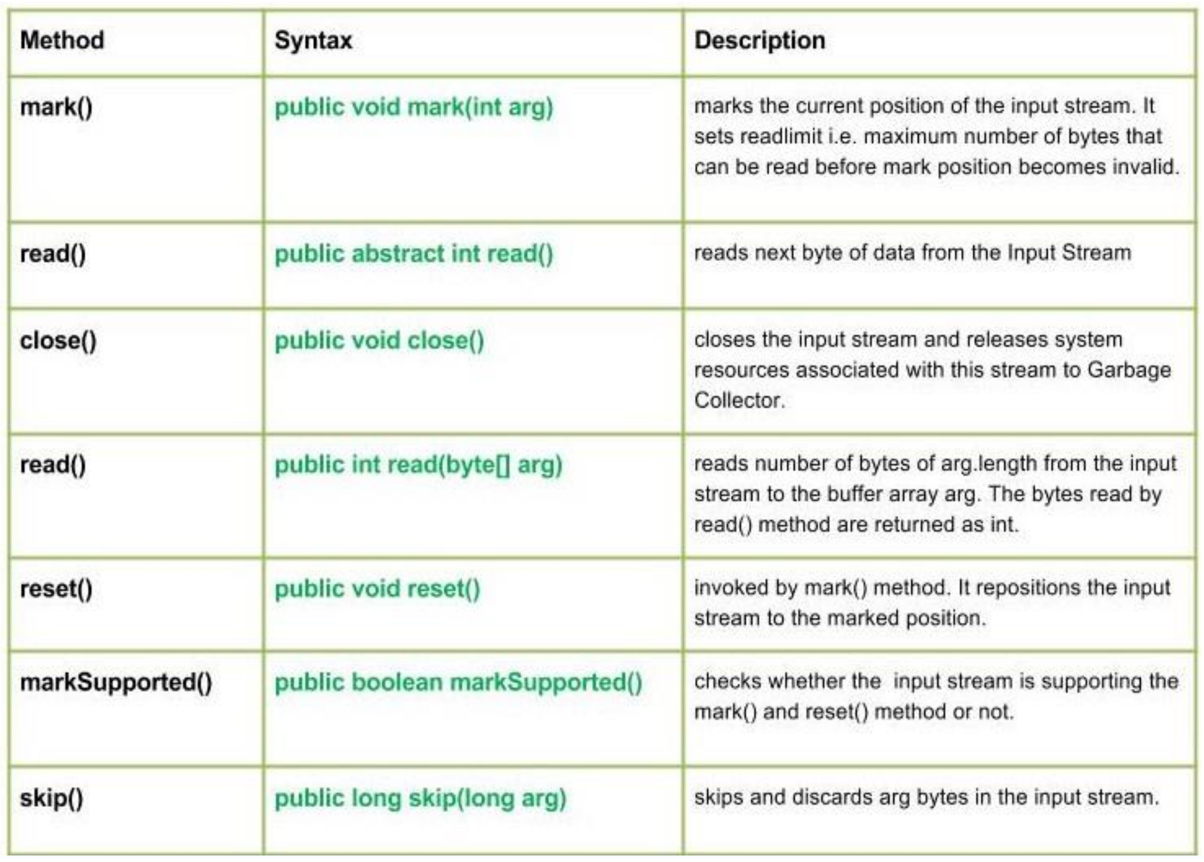
Declaration :

```
public abstract class InputStream
  extends Object
    implements Closeable
```

Constructor :

- InputStream() : Single Constructor

Methods:



Method	Syntax	Description
mark()	public void mark(int arg)	marks the current position of the input stream. It sets readlimit i.e. maximum number of bytes that can be read before mark position becomes invalid.
read()	public abstract int read()	reads next byte of data from the Input Stream
close()	public void close()	closes the input stream and releases system resources associated with this stream to Garbage Collector.
read()	public int read(byte[] arg)	reads number of bytes of arg.length from the input stream to the buffer array arg. The bytes read by read() method are returned as int.
reset()	public void reset()	invoked by mark() method. It repositions the input stream to the marked position.
markSupported()	public boolean markSupported()	checks whether the input stream is supporting the mark() and reset() method or not.
skip()	public long skip(long arg)	skips and discards arg bytes in the input stream.

Byte Stream Classes

These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.

The InputStream and OutputStream classes (abstract) are the super classes of all the input/output stream classes: classes that are used to read/write a stream of bytes. Following are the byte array stream classes provided by Java −

InputStream	**OutputStream**
FIleInputStream	FileOutputStream
ByteArrayInputStream	ByteArrayOutputStream
ObjectInputStream	ObjectOutputStream
PipedInputStream	PipedOutputStream
FilteredInputStream	FilteredOutputStream
BufferedInputStream	BufferedOutputStream
DataInputStream	DataOutputStream

Character Streams Classes

These handle data in 16 bit Unicode. Using these you can read and write text data only.

The Reader and Writer classes (abstract) are the super classes of all the character stream classes: classes that are used to read/write character streams. Following are the character array stream classes provided by Java −

Reader	**Writer**
BufferedReader	BufferedWriter
CharacterArrayReader	CharacterArrayWriter
StringReader	StringWriter
FileReader	FileWriter
InputStreamReader	InputStreamWriter
FileReader	FileWriter

Java.io.OutputStream class

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink. Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

Constructor and Description

- **OutputStream() :** Single Constructor

Methods:

- **void close() :** Closes this output stream and releases any system resources associated with this stream.
- **Syntax :**public void close()
- throws IOException
- **Throws:**
  IOException
- **void flush() :** Flushes this output stream and forces any buffered output bytes to be written out.
- **Syntax :**public void flush()
- throws IOException
- **Throws:**
  IOException

Character Stream Vs Byte Stream

I/O Stream
A stream is a method to sequentially access a file. I/O Stream means an input source or output destination representing different types of sources e.g. disk files.The java.io package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text.

Stream – A sequence of data.
Input Stream: reads data from source.
Output Stream: writes data to destination.

Character Stream
In Java, characters are stored using Unicode conventions . Character stream automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source and write to destination.

Using Streams

The **Stream** API is **used** to process collections of objects. A **stream** is a sequence of objects that supports various methods which can be pipelined to produce the desired result. **Streams** don't change the original data structure, they only provide the result as per the pipelined methods.

Using File Class

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

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

Java.io.File Class

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

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

Other Stream Classes

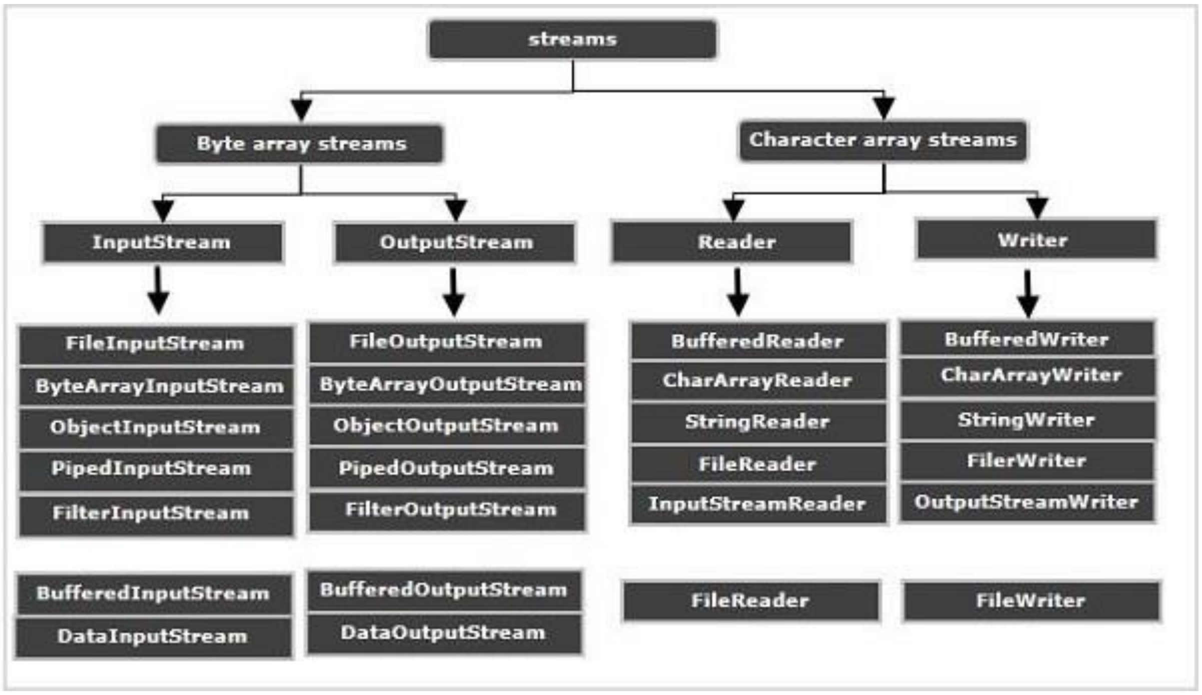
Java provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, i/o devise, other program etc.In general, a Stream will be an input stream or, an output stream.

- **InputStream** − This is used to read data from a source.
- **OutputStream** − This is used to write data to a destination.

Based on the data they handle there are two types of streams −

- **Byte Streams** − These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
- **Character Streams** − These handle data in 16 bit Unicode. Using these you can read and write text data only.

Following diagram illustrates all the input and output Streams (classes) in Java.



Standard Streams

In addition to above mentioned classes Java provides 3 standard streams representing the input and, output devices.

- **Standard Input** − This is used to read data from user through input devices. keyboard is used as standard input stream and represented as System.in.
- **Standard Output** − This is used to project data (results) to the user through output devices. A computer screen is used for standard output stream and represented as System.out.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

Example

Following Java program reads the data from user using BufferedInputStream and writes it into a file using BufferedOutputStream.

```
import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class BufferedInputStreamExample {

   public static void main(String args[]) throws IOException {

      //Creating an BufferedInputStream object

      BufferedInputStream inputStream = new BufferedInputStream(System.in);

      byte bytes[] = new byte[1024];

      System.out.println("Enter your data ");

      //Reading data from key-board

      inputStream.read(bytes);

      //Creating BufferedOutputStream object

      FileOutputStream out= new FileOutputStream("D:/myFile.txt");

      BufferedOutputStream outputStream = new BufferedOutputStream(out);

      //Writing data to the file

      outputStream.write(bytes);

      outputStream.flush();

      System.out.println("Data successfully written in the specified file");

   }

}
```

Output

```
Enter your data
Hi welcome ...
Data successfully written in the specified file
```

UNIT 5 : APPLET AND SWING

Difference between Application and Applets

A Java program can be classified into two types, one is an Application and another is an Applet

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

.Application

An application is a stand-alone java program that runs with the support of a virtual machine in a client or server-side.

- A java application is designed to perform a specific function to run on any Java-compatible virtual machine regardless of the computer architecture.
- An application is either executed for the user or for some other application program.
- Examples of java applications include database programs, development tools, word processors, text and image editing programs, spreadsheets, web browsers, etc.

Example

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

```
   }

}
```

Output

```
Welcome
```

Applet

- An applet is specifically designed to be executed within an HTML web document using an external API.
- They are basically small programs, more like the web version of an application that requires a Java plugin to run on the client browser.
- Applets run on the client-side and are generally used for internet computing.
- When we see an HTML page with an applet in a Java-enabled web browser, the applet code gets transferred to the system and is finally run by the Java-enabled virtual machine on the browser.

Example

```
import java.awt.*;

import java.applet.*;

public class AppletDemo extends Applet{

   public void paint(Graphics g) {

      g.drawString("Welcome ", 50, 50);

   }

}

/* <applet code="AppletDemo.class" width="300" height="300">

   <applet>*/
```

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.

- **destroy** – This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** – Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

Creating an Executable Applet

An applet is a small Internet-based program written in Java, a programming language for the Web, which can be downloaded by any computer.

Types of applet

1)Stand alone applet

2)Local applet

Executable applet is nothing but the .class file of applet, which is obtained by compiling the source code of the applet. Compiling the applet is exactly the same as compiling an application using following command.

```
javac appletname.java
```

The compiled output file called appletname.class should be placed in the same directory as the source file.

The following are the steps that are involved in developing and testing and applet.

1. Buliding an applet code(.java file)
2. Creating an executable applet(.class file)
3. Designing a web page using HTML
4. Preparing <Applet Tag>
5. Incorporating <Applet> tag into the web page.
6. Creating HTMl file.
7. Testing the applet code.

Creating a basic Applet

Following example demonstrates how to create a basic Applet by extending Applet Class. You will need to embed another HTML code to run this program.

```
import java.applet.*;
import java.awt.*;

public class Main extends Applet {
  public void paint(Graphics g) {
    g.drawString("Welcome in Java Applet.",40,20);
  }
}
```

Now compile the above code and call the generated class in your HTML code as follows –

```
<HTML>
```

```
  <HEAD>
  </HEAD>

  <BODY>
    <div >
      <APPLET CODE = "Main.class" WIDTH = "800" HEIGHT = "500"></APPLET>
    </div>
  </BODY>
</HTML>
```

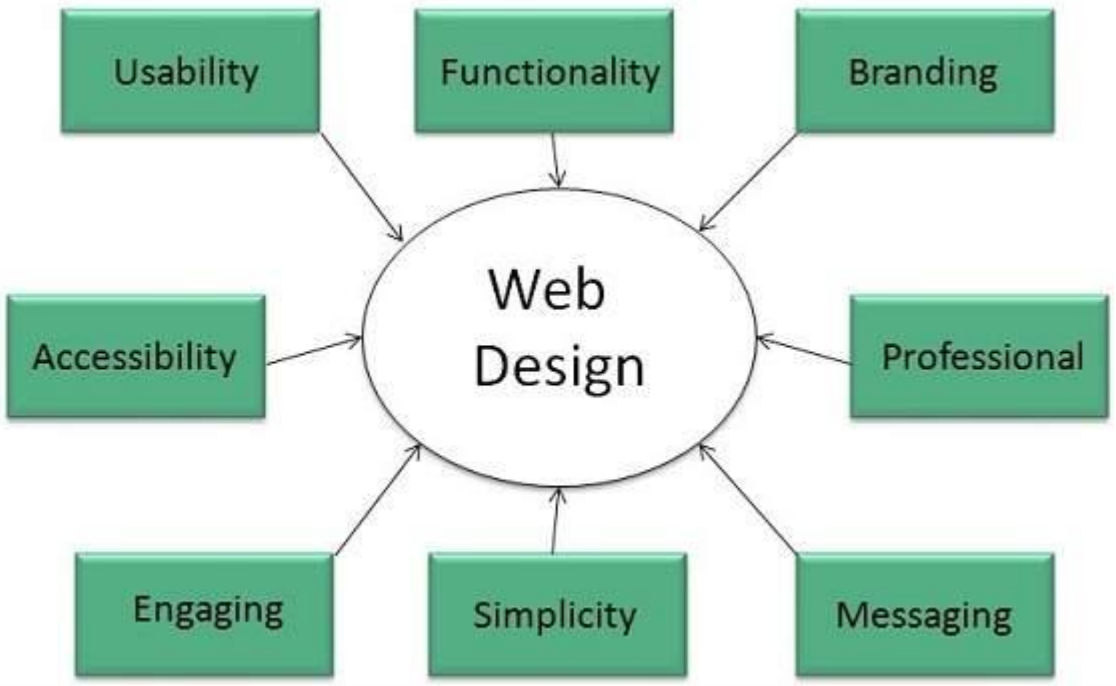
Result

The above code sample will produce the following result in a java enabled web browser.

Welcome in Java Applet.

Designing a Web Page.

Web designing has direct link to visual aspect of a web site. Effective web design is necessary to communicate ideas effectively.



Web designing is subset of web development. However these terms are used interchangeably.

Key Points

Design Plan should include the following:

- Details about information architecture.
- Planned structure of site.
- A site map of pages

Wireframe

Wireframe refers to a visual guide to appearance of web pages. It helps to define structre of web site, linking between web pages and layout of visual elements.

Following things are included in a wireframe:

- Boxes of primary graphical elements
- Placement of headlines and sub headings
- Simple layout structure
- Calls to action
- Text blocks

Web Page Anatomy

A web site includes the following components:

Containing Block

Container can be in the form of page's body tag, an all containing div tag. Without container there would be no place to put the contents of a web page.

Logo

Logo refers to the identity of a website and is used across a company's various forms of marketing such as business cards, letterhead, brouchers and so on.

Naviagation

The site's **navigation system** should be easy to find and use. Oftenly the anvigation is placed rigth at the top of the page.

Content

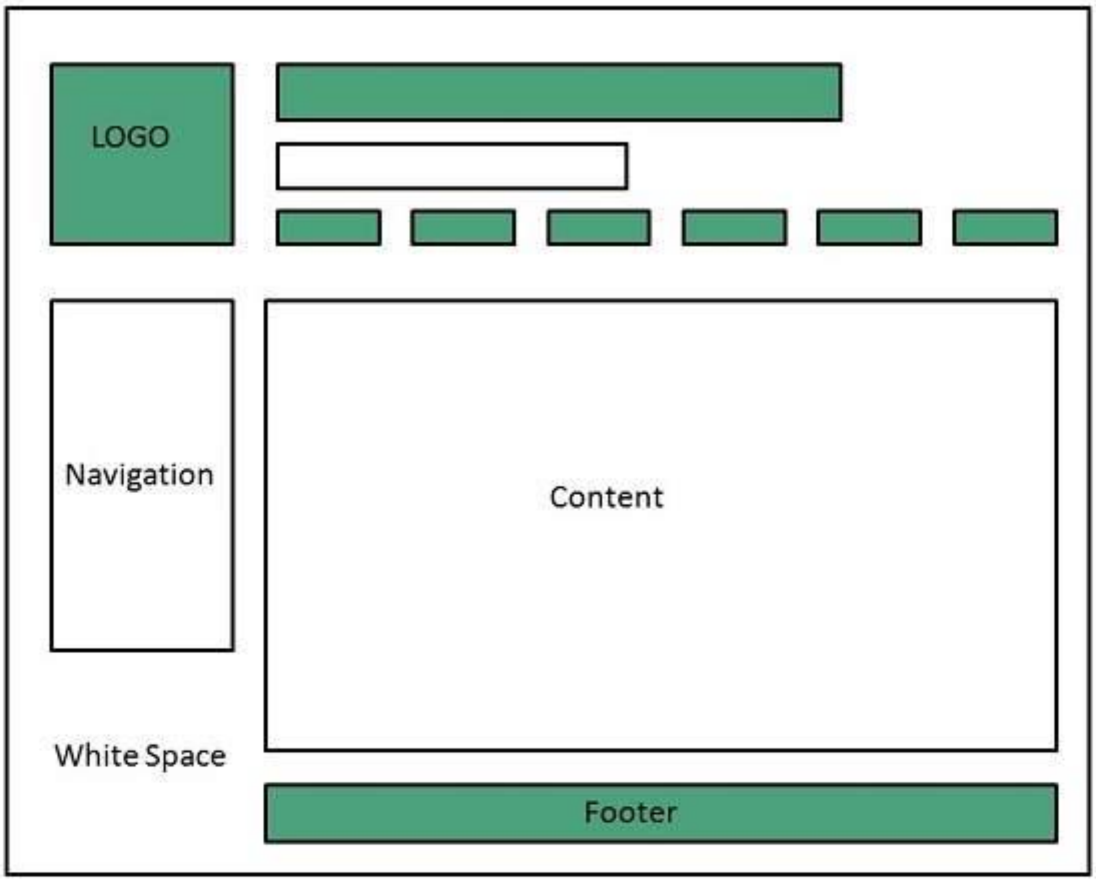
The content on a web site should be relevant to the purpose of the web site.

Footer

Footer is located at the bottom of the page. It usually contains copyright, contract and legal information as well as few links to the main sections of the site.

Whitespace

It is also called as **negative space** and refers to any area of page that is not covered by type or illustrations.



Web design Mistakes

One should be aware of the following common mistakes should always keep in mind:

- Website not working in any other browser other internet explorer.
- Using cutting edge technology for no good reason
- Sound or video that starts automatically
- Hidden or disguised navigation
- 100% flash content.

Adding Applet to HTML File

The HTML <applet> tag specifies an applet. It is used for embedding a Java applet within an HTML document.

Example

```
<!DOCTYPE html>
<html>

  <head>
    <title>HTML applet Tag</title>
  </head>

  <body>
    <applet code = "newClass.class" width = "300" height = "200"></applet>
  </body>

</html>
```

Here is the *newClass.java* file –

```
import java.applet.*;
import java.awt.*;

public class newClass extends Applet {
   public void paint (Graphics gh) {
      g.drawString("Tutorialspoint.com", 300, 150);
   }
}
```

Passing Parameters to Applets

Parameters are passed to applets in NAME=VALUE pairs in <PARAM> tags between the opening and closing APPLET tags. Inside the applet, you read the values passed through the PARAM tags with the getParameter() method of the java.applet.Applet class.

The program below demonstrates this with a generic string drawing applet. The applet parameter "Message" is the string to be drawn.

```
import java.applet.*;
import java.awt.*;

public class DrawStringApplet extends Applet {

  private String defaultMessage = "Hello!";

  public void paint(Graphics g) {

    String inputFromPage = this.getParameter("Message");
    if (inputFromPage == null) inputFromPage = defaultMessage;
    g.drawString(inputFromPage, 50, 25);

  }

}
```

You also need an HTML file that references your applet. The following simple HTML file will do:

```
<HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>

<BODY>
This is the applet:<P>
```

```
<APPLET code="DrawStringApplet" width="300" height="50">
<PARAM name="Message" value="Howdy, there!">
This page will be very boring if your
browser doesn't understand Java.
</APPLET>
</BODY>
</HTML>
```

Of course you are free to change "Howdy, there!" to a "message" of your choice. You only need to change the HTML, not the Java source code. PARAMs let you customize applets without changing or recompiling the code.

This applet is very similar to the HelloWorldApplet. However rather than hardcoding the message to be printed it's read into the variable inputFromPage from a PARAM element in the HTML.

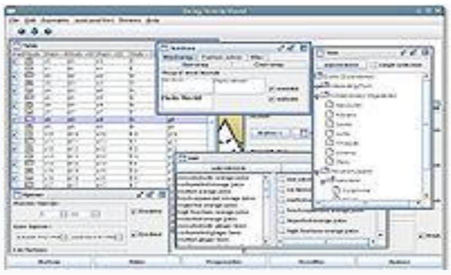
You pass getParameter() a string that names the parameter you want. This string should match the name of a PARAM element in the HTML page. getParameter() returns the value of the parameter. All values are passed as strings. If you want to get another type like an integer, then you'll need to pass it as a string and convert it to the type you really want.

The PARAM element is also straightforward. It occurs between <APPLET> and </APPLET>. It has two attributes of its own, NAME and VALUE. NAME identifies which PARAM this is. VALUE is the string value of the PARAM. Both should be enclosed in double quote marks if they contain white space.

An applet is not limited to one PARAM. You can pass as many named PARAMs to an applet as you like. An applet does not necessarily need to use all the PARAMs that are in the HTML. Additional PARAMs can be safely ignored.

Creating Swing Applet and Application

Swing is a tool kit in Java which provides a way to build cross platform user interfaces. It is built on top of and designed as a replacement for AWT, the other UI toolkit built into Java.



Example of a Swing application

Swing provides many controls and widgets to build user interfaces with. Swing class names typically begin with a J such as JButton, JList, JFrame. This is mainly to differentiate them from their AWT counterparts and in general are one-to-one replacements. Swing is built on the concept of *Lightweight components* vs AWT and SWT's concept of *Heavyweight components*. The difference between the two is that the Lightweight components are rendered (drawn) using purely Java code, such as drawLine and drawImage, whereas Heavyweight components use the native operating system to render the components.

Some components in Swing are actually heavyweight components. The top-level classes and any derived from them are heavyweight as they extend the AWT versions. This is needed because at the root of the UI, the parent windows need to be provided by the OS. These top-level classes include JWindow, JFrame, JDialog and JApplet. All Swing components to be rendered to the screen must be able to trace their way to a root window or one of those classes.

Using Swing.

- Controls: Buttons, Check Boxes, Lists, Trees, Tables, Combo boxes (dropdown list), Menus, Text fields
- Displays: Labels, Progress bars, Icons, Tool Tips
- Pluggable look and feels (PLAFs): Windows, CDE/Motif, Mac. Allows for "skinning" the application without changing any code
- Standard Top-Level Windows: Windows, Frames, Dialogs etc.
- Event Listener APIs
- Key bindings & mnemonics: Allow keystrokes to map to specific actions.

Swing Applet

An **applets** are client side web based program i.e executes on web browser. **Swing applets** are same as AWT **applets**, the variation is that **Swing** extends JApplet and

JApplet consists of all the features of **Applet** because of JApplet is derived from **Applet**. JApplet is a high level container that includes panes.

An applets are client side web based program i.e executes on web browser. To write an applet developer must write access specifier public .

Swing applets are same as AWT applets, the variation is that Swing extends JApplet and JApplet consists of all the features of Applet because of JApplet is derived from Applet.

JApplet is a high level container that includes panes.Applet life cycle uses five methods such as init(), start(), paint(), stop(), destroy() methods.

The init() and destroy() methods of an applet gets executed only once where as remaining methods of an applet gets executed every time when applet comes into focus uses start() or lost focus uses stop().

Syntax

Here class name should extend the Applet and this applet class available in import java.applet.*; package.
import java.applet.*;
Public class MyApplet extends Applet

Example

Create a package and import all the required packages.

```
package swing;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

Create a class that should extend JApplet to inherit the properties.

```
public class HTMLLabelApplet extends JApplet
```

Create buttons and labels.

```
JButton jbtnOne;
  JButton jbtnTwo;

  JLabel jlab;
```

Call the init() method, where invokeAndWait belongs to Swing utility classes used to update the GUI thread.Swing is not thread safe and only repaint() is thread safe in swing. If user tries to update any thing exception will be occured.

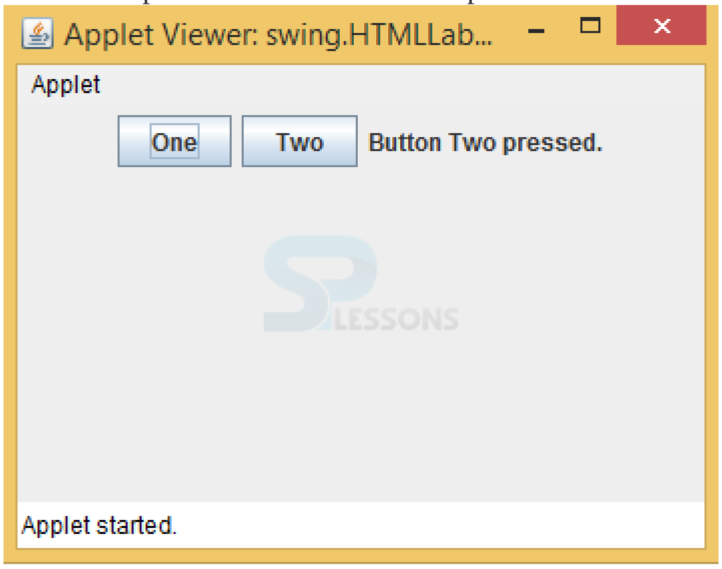
```
public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
          public void run() {
            guiInit(); // initialize the GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
    }
  }
```

If Applet is restarted then start() method will be called.

```
public void start() { }
```
If Applet is stopped then stop() method will be called.
```
public void stop() { }
```
If Applet is destroyed then destroy() method will be called.
```
public void destroy() { }
```
Initialize GUI setup and flow layout.
```
private void guiInit() {
    setLayout(new FlowLayout());
```
Create buttons, labels and add ActionListeners to the those components.
```
jbtnOne = new JButton("One");
    jbtnTwo = new JButton("Two");
 jlab = new JLabel("Press a button.");
jbtnOne.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent le) {
        jlab.setText("Button One pressed.");
      }
    });
```
Add components to the content pane.
```
getContentPane().add(jbtnOne);
    getContentPane().add(jbtnTwo);
    getContentPane().add(jlab);
```
HTMLLabelApplet.java
```
package swing;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class HTMLLabelApplet extends JApplet {
  JButton jbtnOne;
  JButton jbtnTwo;

  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
          public void run() {
            guiInit(); // initialize the GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
    }
  }
 public void start() {

  }

  public void stop() {

  }
 public void destroy() {

  }
 private void guiInit() {

    setLayout(new FlowLayout());
    // Create  buttons and a label.
    jbtnOne = new JButton("One");
    jbtnTwo = new JButton("Two");

    jlab = new JLabel("Press a button.");

    // Add action listeners for the buttons.
    jbtnOne.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent le) {
        jlab.setText("Button One pressed.");
      }
    });

    jbtnTwo.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent le) {
        jlab.setText("Button Two pressed.");
      }
    });

    // Add the components to the applet's content pane.
    getContentPane().add(jbtnOne);
    getContentPane().add(jbtnTwo);
    getContentPane().add(jlab);
  }
}
```

Output:

The Swing Applet Output will be as follows. When click on button one and two, it displays Button one pressed and Button two is pressed.



Programming using Panes

A layered **pane** is a **Swing** container that provides a third dimension for positioning components: depth, also known as Z order. When adding a component to a layered **pane**, you specify its depth as an integer.

Below programs illustrate the use of Pane Class:

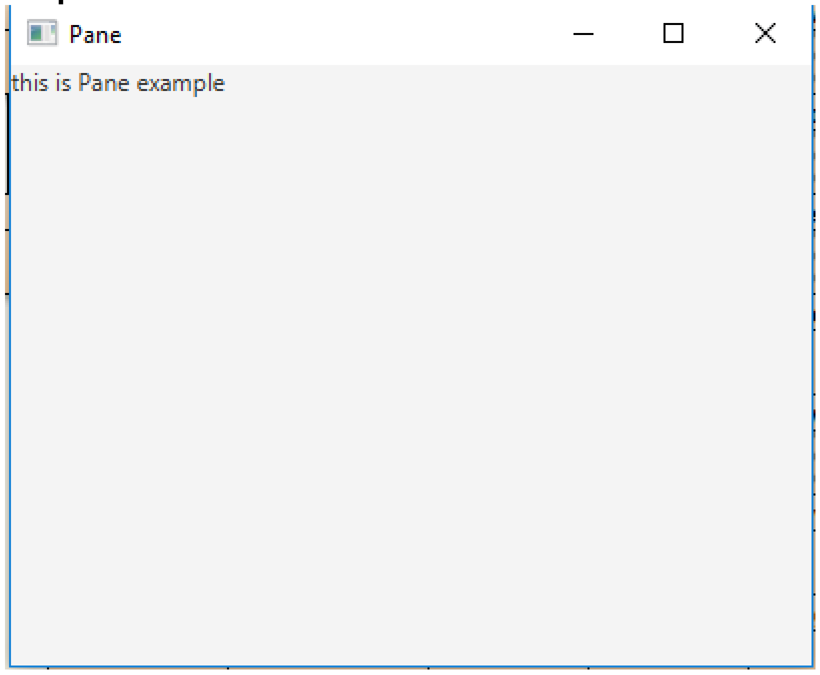
Java Program to create a Pane and add label to the Pane and add it to the stage:

In this program we are creating a Pane named *pane* and a Label named *label*. Now add this label to the pane by passing it as an argument of the constructor of the *pane*. Then add the pane to the Scene and the scene to the stage. Call the *show()* function to display the final results.

```
// Java Program to create a Pane
// and add label to the Pane
// and add it to the stage
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.canvas.*;
import javafx.scene.web.*;
import javafx.scene.layout.Pane;
import javafx.scene.shape.*;

public class Pane_0 extends Application {

    // launch the application
    public void start(Stage stage)
    {

        try {

            // set title for the stage
            stage.setTitle("Pane");

            // create a label
            Label label = new Label("this is Pane example");

            // create a Pane
            Pane pane = new Pane(label);

            // create a scene
            Scene scene = new Scene(pane, 400, 300);

            // set the scene
            stage.setScene(scene);

            stage.show();
        }
```

```
            catch (Exception e) {

                System.out.println(e.getMessage());
            }
        }

        // Main Method
        public static void main(String args[])
        {

            // launch the application
            launch(args);
        }
    }
```

Output:



this is Pane example

Pluggable Look-and-Feel (PLAF)

The Look and Feel module defines how all of the components in the application look. Each Java Swing Application has an entirely separate L&F from the rest of the applications on the computer, including other Java Applications.

Some examples of L&Fs are: Nimbus (really new), Metal, Aqua(Macs only), Windows Aero & Windows Classic (Windows only), and Motif(Highly customizable by the User of the Application). There are also many third-party Look and Feels available.

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.

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.

Labels

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

JLabel class declaration

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

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

TextFields

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

JTextField class declaration

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

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

Buttons

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

JButton class declaration

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

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

ToggleButtons

JToggleButton is used to create toggle button, it is two-states button to switch on or off.

Nested Classes

Modifier and Type	Class	Description
protected class	JToggleButton.AccessibleJToggleButton	This class impl
static class	JToggleButton.ToggleButtonModel	The ToggleButt

CheckBoxs

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

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

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

RadioButtons

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

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

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

Viewports

The JViewport class is used to implement scrolling. JViewport is designed to support both logical scrolling and pixel-based scrolling. The viewport's child, called the view, is scrolled by calling the JViewport.setViewPosition() method.

Scroll Panes

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.

Scroll Bars

The object of JScrollbar class is used to add horizontal and vertical scrollbar. It is an implementation of a scrollbar. It inherits JComponent class.

JScrollBar class declaration

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

public class JScrollBar **extends** JComponent **implements** Adjustable, Accessible

List

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

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

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

ComboBox

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

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

public class JComboBox **extends** JComponent **implements** ItemSelectable, ListD
ataListener, ActionListener, Accessible

Progress Bars

The JProgressBar class is used to display the progress of the task. It inherits JComponent class.

JProgressBar class declaration

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

```
public class JProgressBar extends JComponent implements SwingConstants, Ac
cessible
```

Menu Bar

The JMenuBar class is used to display menubar on the window or frame. It may have several menus.

The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.

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

JMenuBar class declaration

```
public class JMenuBar extends JComponent implements MenuElement, Accessi
ble
```

ToolBars

JToolBar container allows us to group other components, usually buttons with icons in a row or column. JToolBar provides a component which is useful for displaying commonly used actions or controls.

Layered Panes

The JLayeredPane class is used to add depth to swing container. It is used to provide a third dimension for positioning component and divide the depth-range into several different layers.

JLayeredPane class declaration

```
public class JLayeredPane extends JComponent implements Accessible
```

Tabbed Panes

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

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

public class JTabbedPane **extends** JComponent **implements** Serializable, Accessi
ble, SwingConstants

Split Panes

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.

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.

Windows

JWindow is a part of Java Swing and it can appear on any part of the users desktop. It is different from JFrame in the respect that JWindow does not have a title bar or window management buttons like minimize, maximize, and close, which JFrame has. JWindow can contain several components such as buttons and labels.

Constructor of the class are:

1. **JWindow()** : creates an empty Window without any specified owner
2. **JWindow(Frame o)** :creates an empty Window with a specified frame as its owner
3. **JWindow(Frame o)** : creates an empty Window with a specified frame as its owner
4. **JWindow(Window o)** : creates an empty Window with a specified window as its owner
5. **JWindow(Window o, GraphicsConfiguration g)** : creates an empty window with a specified window as its owner and specified graphics Configuration.
6. **JWindow(GraphicsConfiguration g)** :creates an empty window with a specified Graphics Configuration

Dialog Boxes

Java JOptionPane. The JOptionPane class is used to provide standard **dialog boxes** such as **message dialog box**, confirm **dialog box** and input **dialog box**. These **dialog boxes** are used to display information or get input from the user. The JOptionPane class inherits JComponent class.

Inner Frame

Some times you will need to display a frame till the application is running and want to do operations on an another frames lying within the internal frames.

To create internal frames in Java you may use javax.swing.JInternalFrame class.

This class has various constructors using which you can create frames within a frame with different features like maximizable/not maximizable internal frame, resizable/not resizable internal frame, closable/not closable internal frame.

---oOo----