UNIT 1

INTRODUCTION TO JAVA:

- Java is a programming language created by James Gosling from Sun Microsystems (Sun) in 1991.
- The target of Java is to write a program once and then run this program on multiple operating systems.
- The first publicly available version of Java (Java 1.0) was released in 1995.
- Over time new enhanced versions of Java have been released. The current version of Java is Java 1.8 which is also known as *Java 8*.
- Java is defined by a specification and consists of a programming language, a compiler, core libraries and a runtime (Java virtual machine).
- The Java runtime allows software developers to write program code in other languages than the Java programming language which still runs on the Java virtual machine.
- The *Java platform* is usually associated with the *Java virtual machine* and the *Java core libraries*.

JAVA VIRTUAL MACHINE (JVM):

- The Java virtual machine (JVM) is a software implementation of a computer that executes programs like a real machine.
- The Java virtual machine is written specifically for a specific operating system, e.g., for Linux a special implementation is required as well as for Windows.

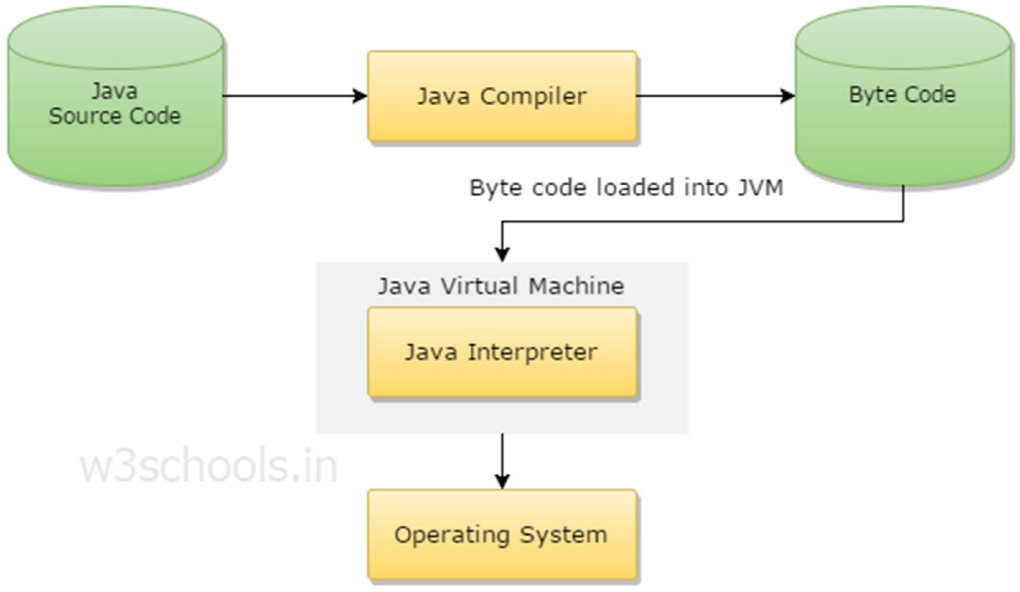


Diagram of JVM

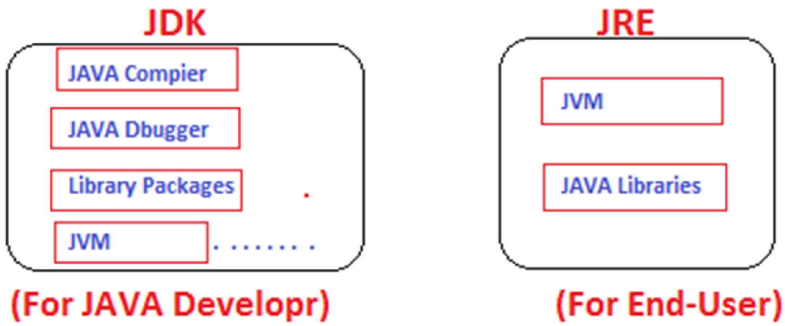
- Java programs are compiled by the Java compiler into *bytecode*. The Java virtual machine interprets this *bytecode* and executes the Java program.

CLASSPATH

- The *classpath* defines where the Java compiler and Java runtime look for `.class` files to load. These instructions can be used in the Java program.
- For example, if you want to use an external Java library you have to add this library to your classpath to use it in your program.

JAVA RUNTIME ENVIRONMENT (JRE) VS. JAVA DEVELOPMENT KIT (JDK)

- A Java distribution typically comes in two flavors, the *Java Runtime Environment* (JRE) and the *Java Development Kit* (JDK).
- The JRE consists of the JVM and the Java class libraries. Those contain the necessary functionality to start Java programs.
- The JDK additionally contains the development tools necessary to create Java programs. The JDK therefore consists of a Java compiler, the Java virtual machine and the Java class libraries.



DEVELOPMENT PROCESS WITH JAVA

- Java source files are written as plain text documents.
- The programmer typically writes Java source code in an *Integrated Development Environment* (IDE) for programming.
- An IDE supports the programmer in the task of writing code, e.g., it provides auto-formating of the source code, highlighting of the important keywords, etc.
- At some point the programmer (or the IDE) calls the Java compiler (`javac`).
- The Java compiler creates the *bytecode* instructions.
- These instructions are stored in `.class` files and can be executed by the Java Virtual Machine.

GARBAGE COLLECTOR

- The JVM automatically re-collects the memory which is not referred to by other objects.
- The Java *garbage collector* checks all object references and finds the objects which can be automatically released.

- While the garbage collector relieves the programmer from the need to explicitly manage memory, the programmer still need to ensure that he does not keep unneeded object references, otherwise the garbage collector cannot release the associated memory.

Where it is used?

According to Sun, 3 billion devices run java. There are many devices where java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javatpoint.com etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games etc.

Types of Java Applications

There are mainly 4 type of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application

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

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

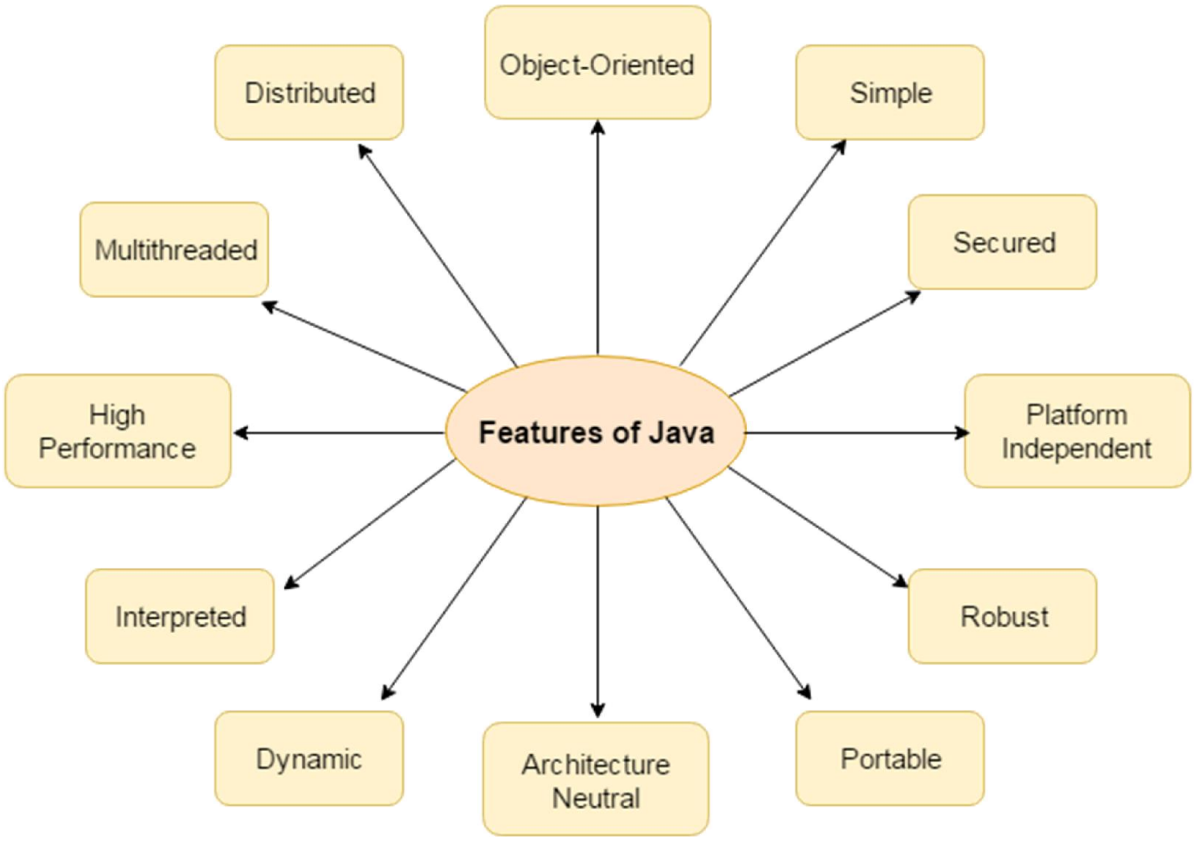
C++ vs. Java

There are many differences and similarities between C++ programming language and Java. A list of top differences between C++ and Java are given below:

Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
Goto	C++ supports goto statement.	Java doesn't support goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. But you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only.	Java uses compiler and interpreter both.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.

FEATURES OF JAVA

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.



1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

Simple

According to Sun, Java language is simple because:

- Syntax is based on C++ (so easier for programmers to learn it after C++).
- Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.
- No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

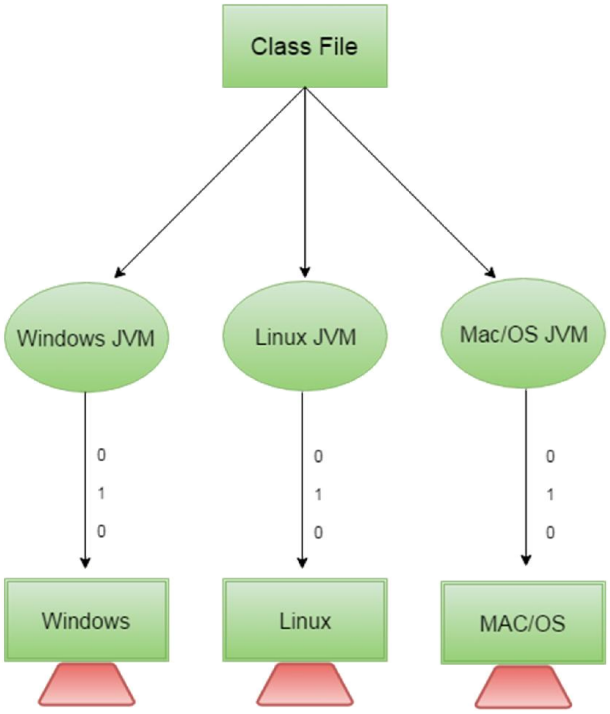
- Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.
- Object-oriented programming (OOPs) is a methodology that simplify software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent

A platform is the hardware or software environment in which a program runs.



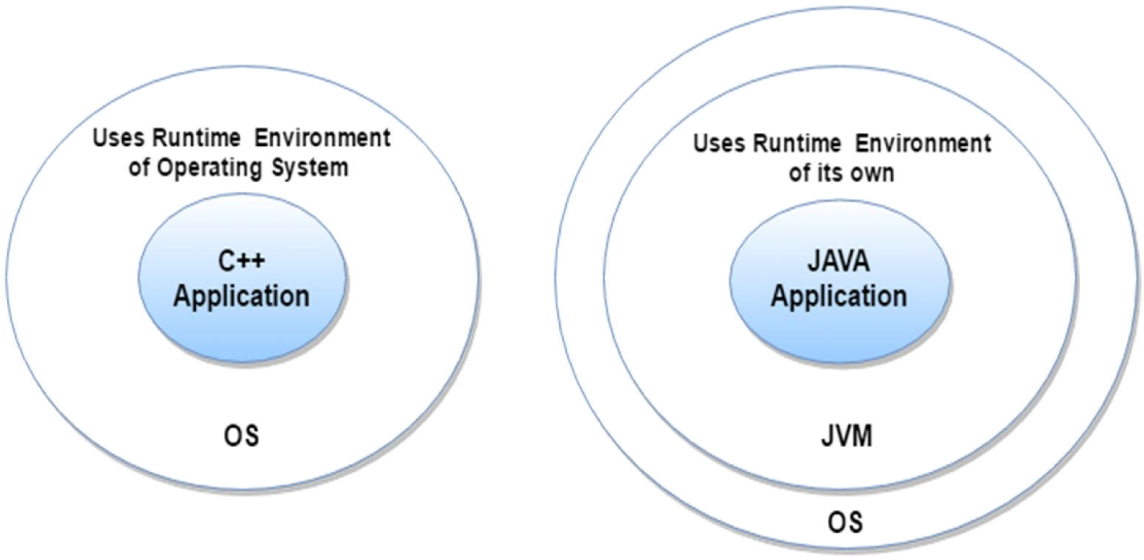
- There are two types of platforms software-based and hardware-based. Java provides software-based platform.
- The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:
    1. Runtime Environment
    2. API(Application Programming Interface)

- Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode.
- This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is secured because:

- No explicit pointer
- Java Programs run inside virtual machine sandbox



- **Class loader:** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Byte code Verifier:** checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** determines what resources a class can access such as reading and writing to the local disk.

Robust

- Robust simply means strong.
- Java uses strong memory management.
- There are lack of pointers that avoids security problem.
- There is automatic garbage collection in java.
- There is exception handling and type checking mechanism in java.

All these points makes java robust.

Architecture-neutral

There is no implementation dependent feature e.g. size of primitive types is fixed.

For Example, In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

Portable

- We May carry java bytecode to any platform. And can execute in any platform.

High-performance

- Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

- We can create distributed applications in java.
- We may access files by calling the methods from any machine on the internet.

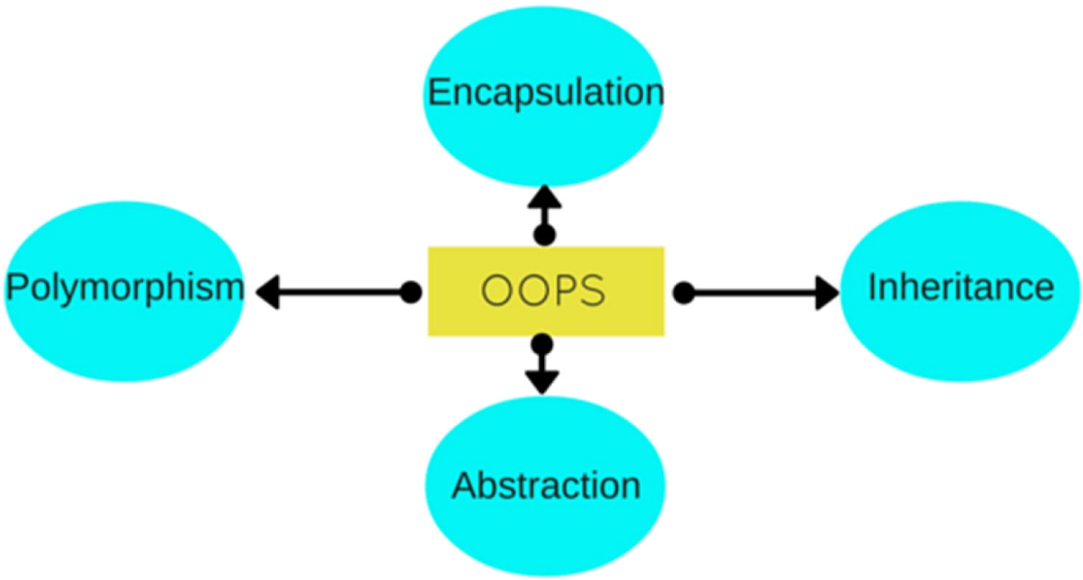
Multi-threaded

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

OOPs (Object Oriented Programming System) Concept:

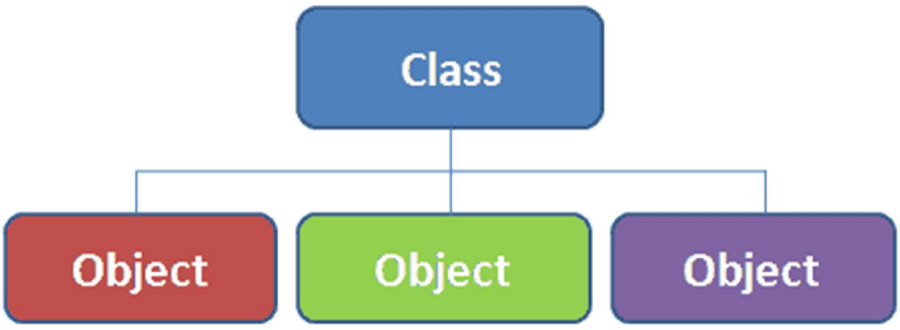
- **Object** means a real word entity such as pen, chair, table etc.
- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.
- It simplifies the software development and maintenance by providing some concepts:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation



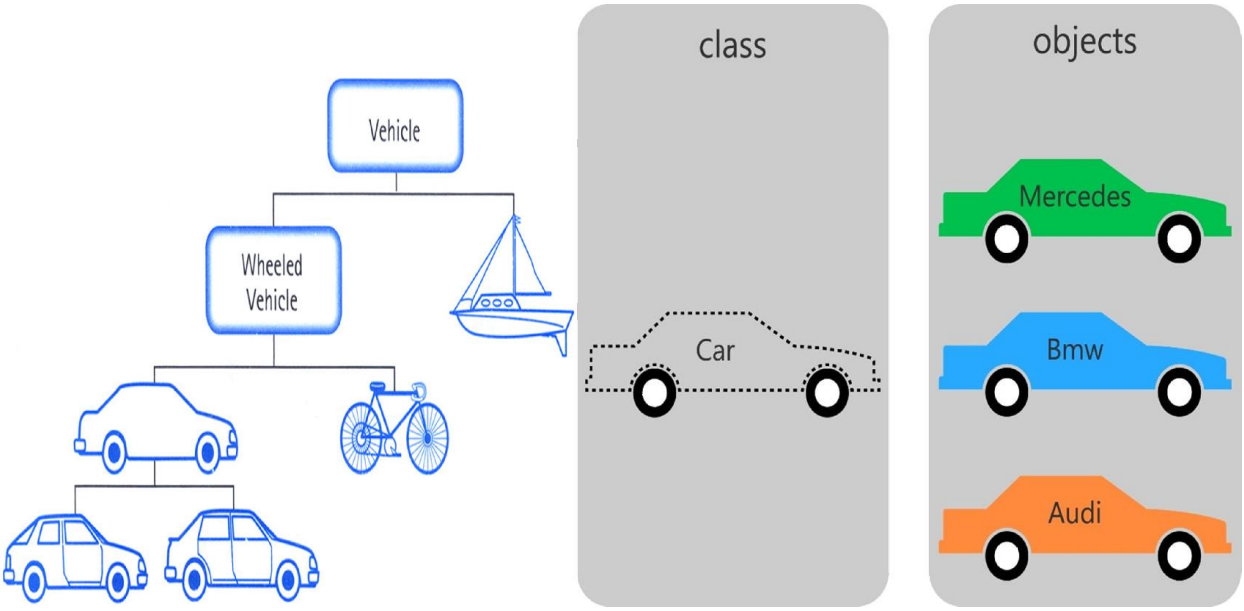
Class

Collection of objects **is called class. It is a logical entity.**



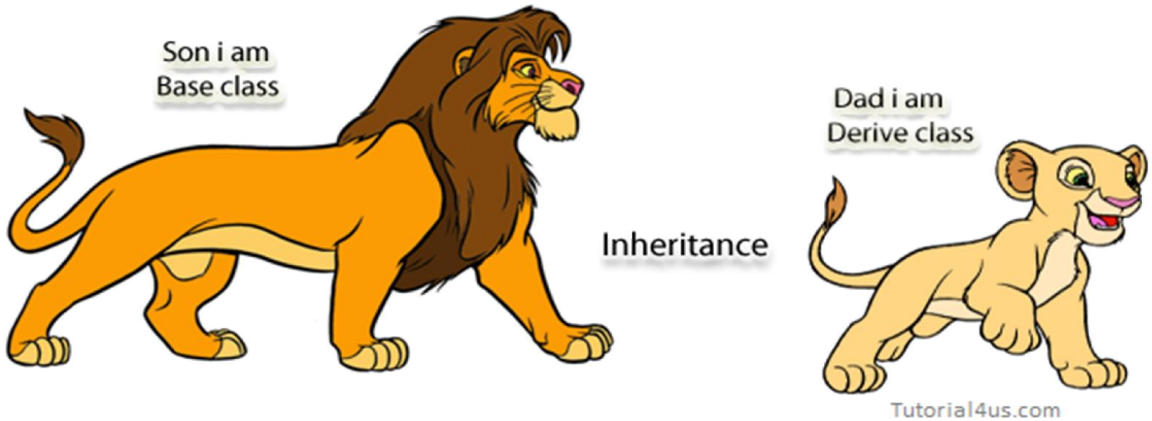
OBJECT

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.



Inheritance

- When one object acquires all the properties and behaviours of parent object i.e. known as inheritance.
- It provides code reusability.
- It is used to achieve runtime polymorphism.



POLYMORPHISM

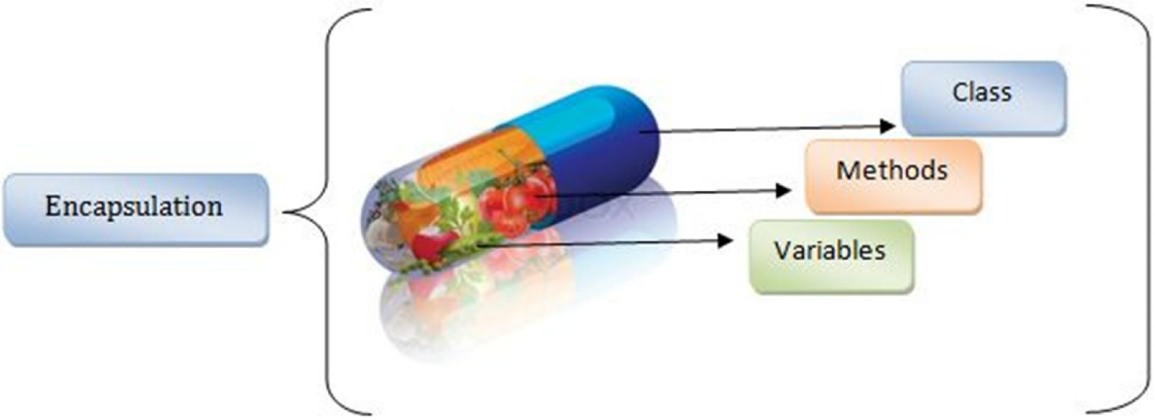
- When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.
- In java, we use method overloading and method overriding to achieve polymorphism.
- Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

ABSTRACTION

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

ENCAPSULATION

- Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation.
- Java bean is the fully encapsulated class because all the data members are private here.

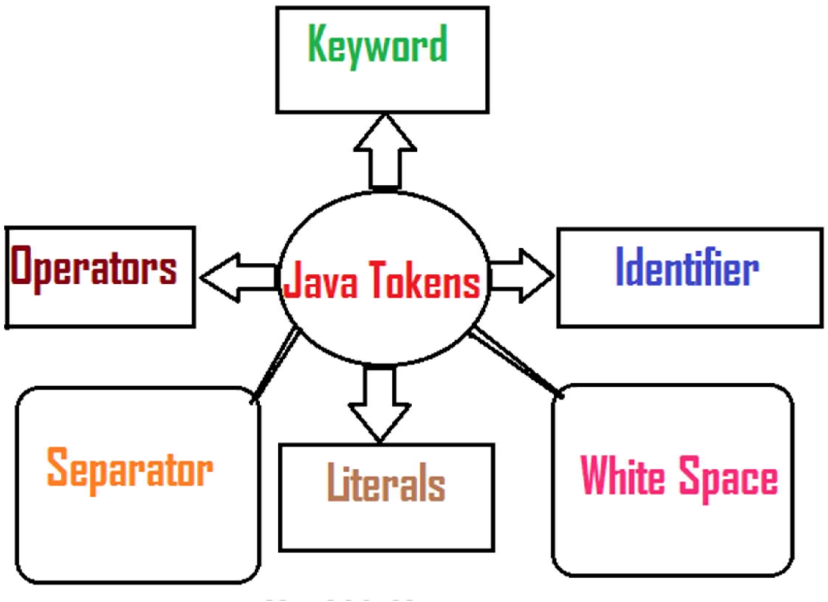


JAVA TOKENS:

- A token is the smallest element of a program that is meaningful to the compiler.
- These tokens define the structure of the Java language.
- While submit a Java program to the Java compiler, the compiler scans the text and extracts individual tokens.

Java tokens can be broken into five categories:

1. Identifiers
2. Keywords
3. Literals
4. Operators
5. Separators



The Java compiler also recognizes and subsequently removes comments and whitespaces.

Example:

```
Public class Hello
{
Public static void main(String args[])
{
System.out.println("welcome in Java");   //print welcome in java
}
}
```

In above Example ,

- The source code contains tokens such as public, class, Hello, {, public, static, void, main, (, String, [], args, {, System, out, println, (, “welcome in Java”, }, }.
- The resulting **tokens·** are compiled into Java **bytecodes** that is capable of being run from within an interpreted java environment.
- **Token** are useful for compiler to detect errors.
- When tokens are not arranged in a particular sequence, the compiler generates an error message.

IDENTIFIERS:

- *Identifiers* are tokens that represent names.
- These names can be assigned to **variables**, **methods**, and **classes** to uniquely identify them to the compiler.

Identifiers are,

- **Sequence of Uppercase letters(A to Z)**
- **Sequence of Lowercase letters (a to z)**
- **Numbers (0 to 9)**
- **Underscore (_)**
- **Dollar Symbol ($)**

RULES OF IDENTIFIERS JAVA

1. Should be single word. That is spaces are not allowed.
   Example: mangoprice is valid but mango price is not valid.
2. Should start with a letter (alphabet) or underscore or $ symbol.
   Example: price, _price and $price are valid identifiers.
3. Should not be a keyword of Java as keyword carries special meaning to the compiler.
   Example: class or void etc.
4. Should not start with a digit but digit can be in the middle or at the end.
   Example: 5mangoescost is not valid and mango5cost and mangocost5 are valid.
5. Length of an identifier in Java can be of 65,535 characters and all are significant.
6. Identifiers are case-sensitive. That is both mango and Mango are treated differently.
7. Can contain all uppercase letters or lowercase letters or a mixture.

Valid and invalid Java identifiers.

Valid	Invalid
HelloWorld	Hello World (uses a space)
Hi_JAVA	Hi JAVA! (uses a space and punctuation mark)
value3	3value(begins with a number)
Tall	short (this is a Java keyword)
$age	#age (does not begin with any other symbol except _ $)

KEYWORDS:

- **A keyword is just a word which has got a special meaning and purpose to the compiler.**
- For this reason, we cannot use them in our program to identify our own (to give a name) a class, variable or method.

List of **48 keywords**

ABSTRACT	ASSERT	BOOLEAN	BREAK	BYTE
CASE	CATCH	CHAR	CLASS	CONTINUE
DEFAULT	DO	DOUBLE	ELSE	ENUM
EXTENDS	FINAL	FINALLY	FLOAT	FOR
IF	IMPLEMENTS	IMPORT	INSTANCEOF	INT
INTERFACE	LONG	NATIVE	NEW	PACKAGE
PRIVATE+	PROTECTED	PUBLIC	RETURN	SHORT
STATIC	STRICTFP	SUPER	SWITCH	SYNCHRONIZED
THIS	THROW	THROWS	TRANSIENT	TRY
VOID	VOLATILE	WHILE		

1. strictfp is added in JDK 1.2

2. assert is added in JDK 1.5

3. enum is added in JDK 1.5

The designers placed the two words, **goto** and **const** in reserved word list and not in keyword list. Designers placed in reserved list for the reason thinking that **goto** and **cosnt** may be required in future versions of Java.

Following is the list of **prohibited 3 literals** (given as values) which cannot be used in Java coding.

True	false	null

SEPARATORS:

- Separators are used to inform the Java compiler of how things are grouped in the code.
- For example, items in a list are separated by commas much like lists of items in a sentence.
- The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements.

Symbol	Name	Purpose
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements. Also used for surrounding cast types.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
.	Period	Used to separate package names from subpackages and classes Also used to separate a variable or method from a reference variable.

LITERALS:

- Java Literals are syntactic representations of boolean, character, numeric, or string data.
- Literals provide a means of expressing specific values in your program.
- For example, in the following statement, an integer variable named count is declared and assigned an integer value.

Types of Java Literals

1. Integer Literals
2. Floating-point Literals
3. Boolean Literals
4. Character Literals
5. String Literals

Literal	type
1	**int**
3.14	**double** (**1.** is a **double** too)
true	**boolean**
'3'	**char** (**'P'** and **'+'** are **char** too)
"CMU ID"	**String**
null	any reference type

COMMENTS AND WHITESPACES:

- The comments and whitespaces are removed by the Java compiler during the tokenization of the source code.
- White space consists of spaces, tabs, and linefeeds. All occurrences of spaces, tabs, or linefeeds are removed by the Java compiler, as are comments.
- Comments can be defined in three different ways, as shown in Table.

Types of comments supported by Java.

Type	Syntax	Usage	Example
Single-line	// comment	All characters after the // up to the end of the line are ignored.	//This is a Single-line style comment.
Multiline	/* comment */	All characters between /* and */ are ignored.	/* This is a Multiline style comment.
Documentation	/** comment */	Same as /* */, except that the comment can be used with the javadoc tool to create automatic documentation.	/** This is a javadoc style comment. */

OPERATORS:

Operator in java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Operator Type	Category	Precedence
Unary	Postfix	*expr*++ *expr*--
	Prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	Multiplicative	* / %
	Additive	+ -
Shift	Shift	<< >> >>>
Relational	Comparison	< > <= >= instanceof
	Equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	Ternary	? :
Assignment	Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

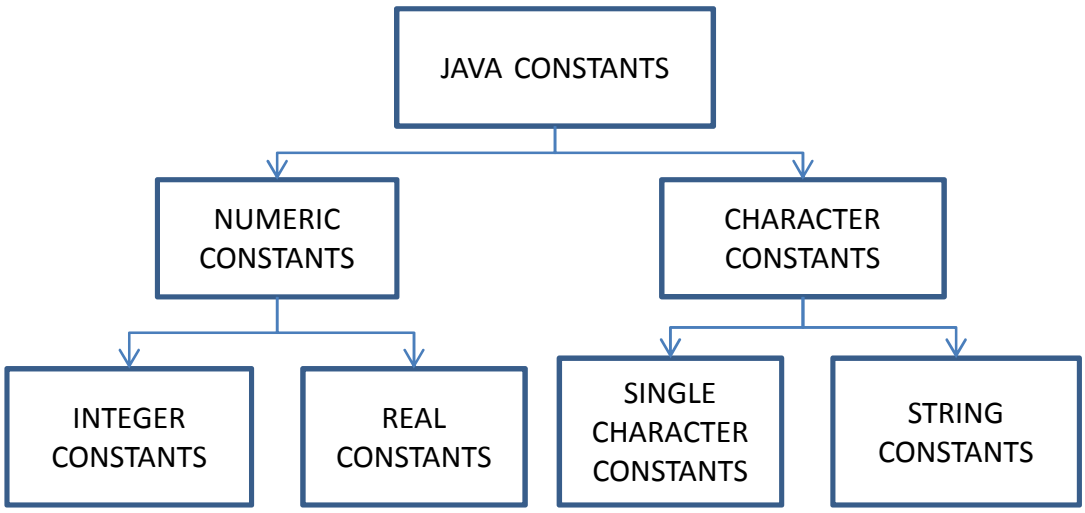
JAVA STATEMENTS:

- A statement is an executable combination of tokens ending with semicolon (;) mark.
- Statements are usually executed in sequence in the order in which they appear.
- It is possible to control the flow of execution, if necessary, using special statements.
- The following image describes the types of statements.

STATEMENTS	DESCRIPTION	REMARKS
Empty Statement	• These do nothing and are used during program development as a place holder	Same as C and C++
Labelled Statement	• Any Statement may begin with a label. • Such labels must not be Keywords • It is used as the arguments of jump statements.	Identical to C and C++
Expression statement	• Most statements are expression statements. • Java has seven expression statements **1. Assignment** **2. Pre-increment** **3. Pre-decrement** **4. Post-Increment** **5. Post-Decrement** **6. Method call** 7. **Allocation Expression**	Same as C++
Selection Statement	• These select one of several control flows • There are Three types of selection statements **1. If statement** **2. If-else statement** 3. **Switch statement**	Same as C and C++
Iteration(Looping) statement	• Determines when looping will take place • There are three types of iteration statements, **1. While loop** **2. Do-while Loop** 3. **For –loop**	Same as C and C++
Jump statement	• It passes control to beginning or end of the current block. • Four types of jump statements are, **1. Break** **2. Continue** **3. Return** 4. **Throw**	C and C++ do not uses jump statement.
Synchronization statement	• used for handling issues with multithreading	Now available in C and C++
Guarding Statement	• It is used for safe handling of code that may cause exceptions. • These statements use the Keywords **try**, **catch** and **finally**.	Same as C++

CONSTANTS:

- Constants in java refer to fixed values that do not change during the execution of a program.
- Java supports several types of constants



Integer constants

- An integer constant refers to a sequence of digits. It includes,
    1. Decimal integer
    2. Octal integer
    3. Hexadecimal integer

1. **Decimal Integer**
    - It consists of a set of digits, 0 through 9, preceded by an optional minus sign.
    - Valid examples of integer constants are, 123, -321, 0 ,654321
    - Embedded spaces, commas & non-digit characters are not permitted b/w digits.

      For eg: 15 750, 20,00 , $1000

2. **Octal integer**
    - It consists of any combination of digits from the set 0 through 7, with a leading 0.

      eg: 037 , 0, 0435, 0551

3. **Hexadecimal integer**
    - A sequence of digits preceded by 0x or 0X is considered as hexadecimal integers.
    - It may also include alphabets A through F or a through f.
    - A letter A through F represents the numbers 10 through 15.

      E.g. 0X2, 0X9F, 0xbcd 0x

Real constants

- Quantities are represented by numbers containing fractional part like 17.678.
- such numbers are called real/floating point constants.

E.g. 0.0083, -0.75 , 435.36

- It is possible that the number may not have digits before the decimal point or digit after the decimal points.

E.g. 215.0, .95 , -.71

- A real numbers may also be expressed in exponential or scientific notation.
(E.g.) the value 215.65 may be written as 2.156e2 in exponential notation. e2 means multiply by 10power2. The general form is,

Mantissa e exponent

Character constants

Single Character constants

- A single character constant or simply character constant contains a single character enclosed within a pair of single quote marks.
- The characters may be alphabets, digits, special characters and blank spaces.

Eg: 'x' , 'A' , '5', ';', ' '

- The character constant '5' is not the same as the number 5. The last constant is a blank space.

String constants

- A string constant is a sequence of characters enclosed b/w double quotes. The characters may be alphabets, digits, special characters and blank spaces.

Eg: "hello java" "2013" "welcome" "#$%....!" "3+2" "x"

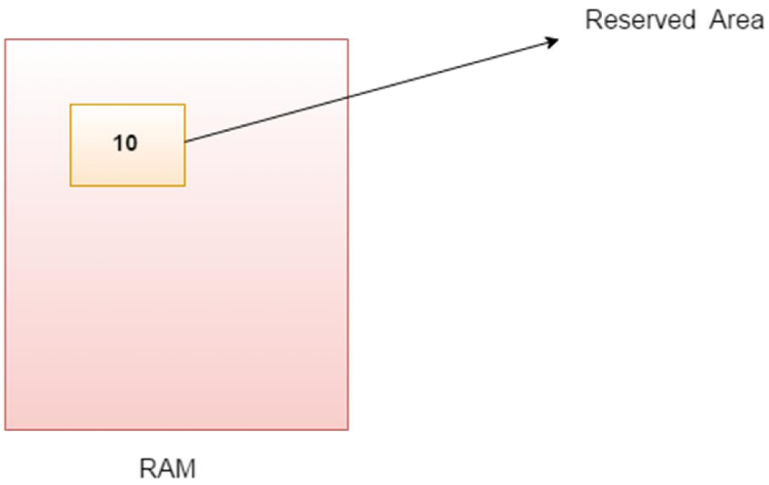
EXAMPLE PROGRAM:

```
class Simple

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

int a=10;

int b=10;

int c=a+b;

System.out.println(c);

}}
```

VARIABLE

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

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.



E.g. int data=50; //Here data is variable

Types of Variable

There are three types of variables in java:

- local variable
- instance variable
- static variable

Local Variables

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

Example

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
public class Test {
  public void pupAge() {
    int age = 0;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }

  public static void main(String args[]) {
    Test test = new Test();
    test.pupAge();
  }
}
```

This will produce the following result −

Output

```
Puppy age is: 7
```

Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables can be accessed directly by calling the variable name inside the class.

```
import java.io.*;
public class Employee {
  // this instance variable is visible for any child class.
  public String name;
```

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

This will produce the following result −

Output

```
name  : Ransika
salary :1000.0
```

Class/Static Variables

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

- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.

Example

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

  // salary  variable is a private static variable
  private static double salary;

  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "Development ";

  public static void main(String args[]) {
    salary = 1000;
    System.out.println(DEPARTMENT + "average salary:" + salary);
  }
}
```

This will produce the following result –

Output

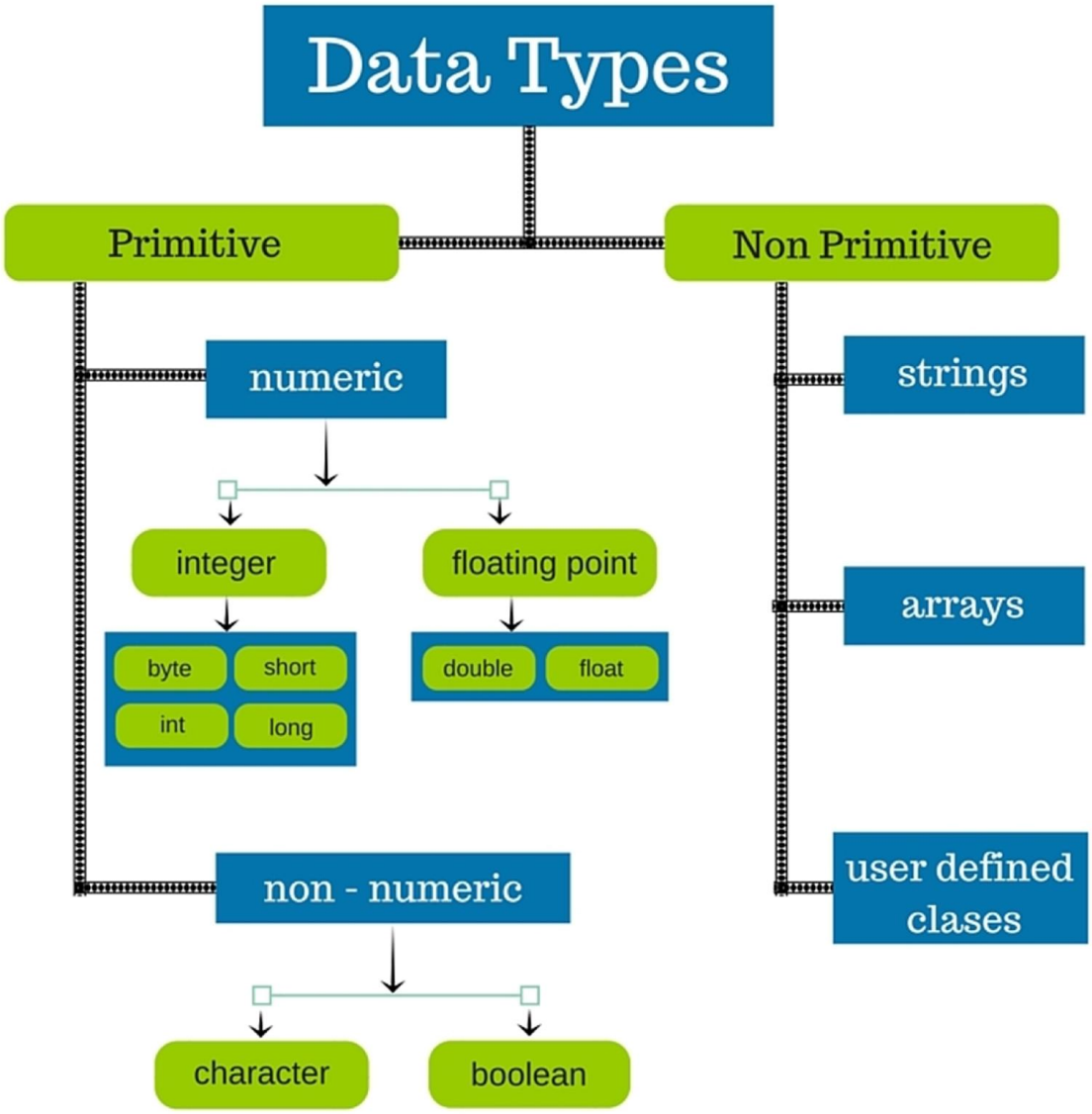
```
Development average salary:1000
```

DATA TYPES:

- Every variable in java has a data type.
- Data types specify the size and type of values that can be stored.
- Java language is rich in its data types.
- Primitive types are also called as *intrinsic* or *built-in* types.
- Derived data types are also known as reference types.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

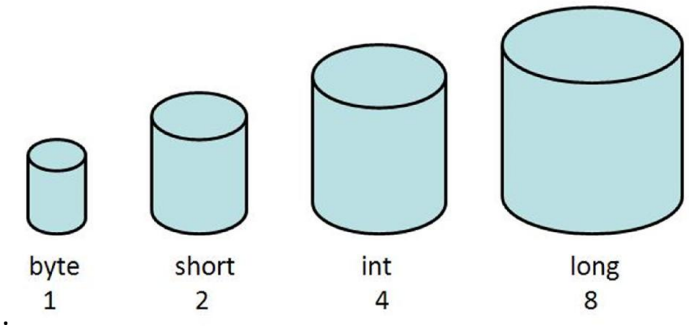
There are two data types available in Java –

- Primitive Data Types
- Reference/Non-Primitive Types



Primitive Data Types

- There are eight primitive datatypes supported by Java.
- Primitive datatypes are predefined by the language and named by a keyword.
- Let us now look into the eight primitive data types in detail

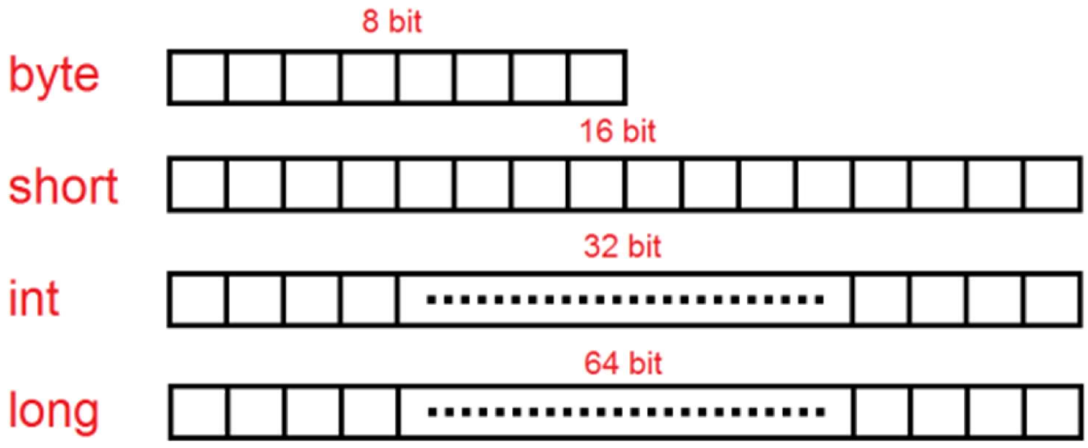


byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000



int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

Example Program:

```
class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
```

Output:

```
10
10.0
```

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

Type	Description	Default	Size	Example Literals
boolean	true or false	false	1 bit	true, false
byte	twos complement integer	0	8 bits	(none)
char	Unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'', '\n', 'ß'
short	twos complement integer	0	16 bits	(none)
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d

char

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

Reference Datatypes

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

TYPE CASTING

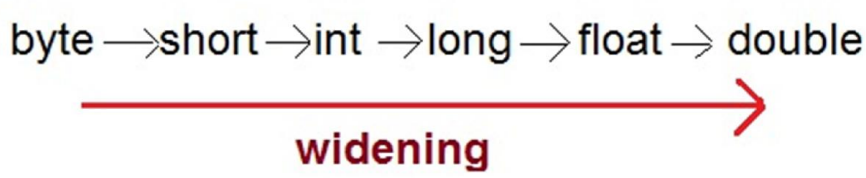
Assigning a value of one type to a variable of another type is known as **Type Casting**.

Example :

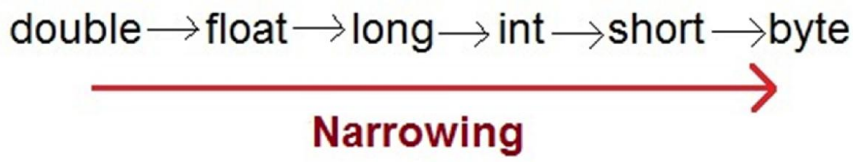
```
int x=10;
byte y=(byte)x;
```

In Java, type casting is classified into two types,

- Widening Casting(Implicit)



- Narrowing Casting(Explicitly done)



Widening or Automatic type conversion

Automatic Type casting take place when,

- the two types are compatible
- the target type is larger than the source type

Example :

```
public class Test
{
   public static void main(String[] args)
   {
     int i = 100;
     long l = i;    //no explicit type casting required
     float f = l;   //no explicit type casting required
```

```
    System.out.println("Int value "+i);
    System.out.println("Long value "+l);
    System.out.println("Float value "+f);
  }
  }
```

Output :

```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit type conversion

While assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

Example :

```
public class Test
{
   public static void main(String[] args)
   {
    double d = 100.04;
    long l = (long)d;  //explicit type casting required
    int i = (int)l;          //explicit type casting required
      System.out.println("Double value "+d);
    System.out.println("Long value "+l);
    System.out.println("Int value "+i);
   }
 }
```

Output :

```
Double value 100.04
Long value 100
Int value 100
```

OPERATORS

- **Operator** in java is a symbol that is used to perform operations.
- For example: +, -, *, / etc.
- There are many types of operators in java which are given below:

1. Arithmetic Operator,
2. conditional Operator,
3. Increment and decrement Operator,
4. Relational Operator,
5. Bitwise Operator,
6. Logical Operator,
7. Assignment Operator and
8. Special Operator.

Arithmetic Operator:

Arithmetic operators are used to perform arithmetic operations on the operands. The following operators are known as Java arithmetic operators.

Operator	Description	Example
+	Addition	10+20 = 30
-	Subtraction	20-10 = 10
*	Multiplication	10*20 = 200
/	Division	20/10 = 2
%	Modulus (Remainder)	20%10 = 0
++	Increment	var a=10; a++; Now a = 11
--	Decrement	var a=10; a--; Now a = 9

Example:

```
class OperatorExample{
public static void main(String args[])
{
System.out.println(10*10/5+3-1*4/2);
}}
```

Output: 21

Relational Operator:

The Java relational operator compares the two operands. The comparison operators are as follows:

Operator	Description	Example
==	Is equal to	10==20 = false
===	Identical (equal and of same type)	10==20 = false
!=	Not equal to	10!=20 = true
!==	Not Identical	20!==20 = false
>	Greater than	20>10 = true
>=	Greater than or equal to	20>=10 = true
<	Less than	20<10 = false
<=	Less than or equal to	20<=10 = false

Example

```
public class Test {
  public static void main(String args[]) {
    int a = 10,b = 20;
    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
    System.out.println("b >= a = " + (b >= a) );
  }}
```

Output

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
```

Bitwise operators

Java defines several bitwise operators that can be applied to the integer types long, int, short, char and byte

Operator	Description	Example
&	Bitwise AND	(10==20 & 20==33) = false
|	Bitwise OR	(10==20 | 20==33) = false
^	Bitwise XOR	(10==20 ^ 20==33) = false
~	Bitwise NOT	(~10) = -10
<<	Bitwise Left Shift	(10<<2) = 40
>>	Bitwise Right Shift	(10>>2) = 2
>>>	Bitwise Right Shift with Zero	(10>>>2) = 2

Logical Operators

The following operators are known as JavaScript logical operators.

Operator	Description	Example
&&	Logical AND	(10==20 && 20==33) = false
||	Logical OR	(10==20 || 20==33) = false
!	Logical Not	!(10==20) = true

Assignment Operators

The following operators are known as JavaScript assignment operators.

Operator	Description	Example
=	Assign	10+10 = 20

+=	Add and assign	var a=10; a+=20; Now a = 30
-=	Subtract and assign	var a=20; a-=10; Now a = 10
*=	Multiply and assign	var a=10; a*=20; Now a = 200
/=	Divide and assign	var a=10; a/=2; Now a = 5
%=	Modulus and assign	var a=10; a%=2; Now a = 0

Special Operators

The following operators are known as JavaScript special operators.

Operator	Description
(?:)	Conditional Operator returns value based on the condition. It is like if-else.
,	Comma Operator allows multiple expressions to be evaluated as single statement.
delete	Delete Operator deletes a property from the object.
In	In Operator checks if object has the given property
instanceof	checks if the object is an instance of given type
New	creates an instance (object)
typeof	checks the type of object.
Void	it discards the expression's return value.
Yield	checks what is returned in a generator by the generator's iterator.

Expressions

- Expressions are a combination of variables, constants & operators arranged as per the syntax of the language.
- Java can handle any complex mathematical expressions.

Algebraic expression	Java expression
Ab-c	A*b-c
(m+n)(x+y)	(m+n)*(x+y)
Ab/c	A*b/c
$3x^2+2x+1$	3*x*x+2*x+1

Evaluation of expression:

- Expressions are evaluated using an assignment statement of the form

Variable=expression;

- Variable is any valid java variable name.
- When the statement is encountered, the expression is evaluated first & the result then replaces the previous value of the variable on the left-hand side.

Ex:

X=a*b-c;

Y=b/c*a;

Z=a-b/c + d;

- The blank spaces around an operator are optional & it is added only to improve readability.
- The variables a, b, c, d must be defined before they are used in the expression.

Example:

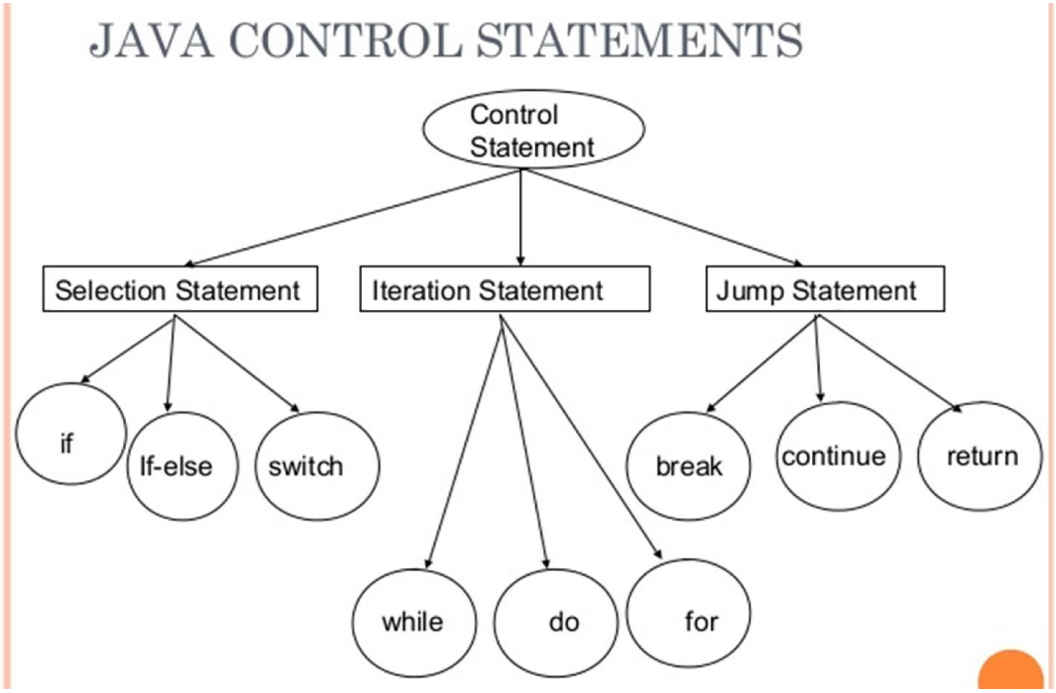
```
class OperatorExample{
public static void main(String args[])
{
System.out.println(10*10/5+3-1*4/2);  //expression
}
}
```

Output:

```
21
```

CONTROL STRUCTURE:

- Control statements are used to provide the **flow of execution** with condition.
- In java program, control structure is can divide in three parts:
    1. Selection statement
    2. Iteration statement
    3. Jumps in statement



Selection Statement:

- Selection statement is also called as Decision making statements because it provides the decision making capabilities to the statements.
- In selection statement, there are two types:
    1. if statement
    2. Switch statement.

Java if statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in java.

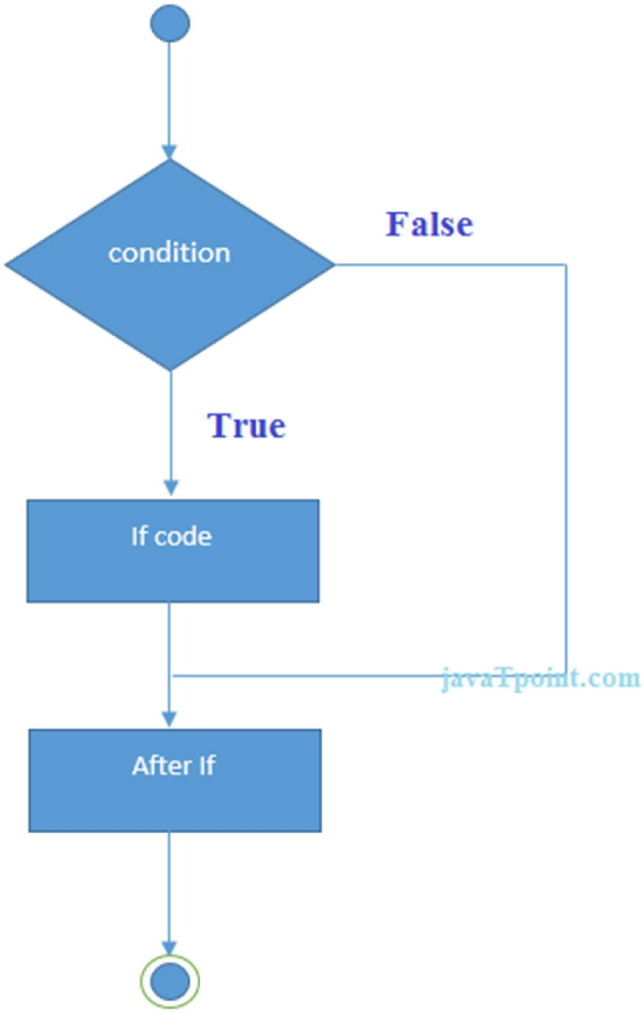
- if statement
- if-else statement
- Nested if-else statement
- if-else-if ladder

IF Statement

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

Syntax:

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



Example:

```
public class IfExample {
public static void main(Strin
g[] arg) {
 int age=20;
   if(age>18){
      System.out.print("Age is g
reater than 18");
   }
}
}
```

Output:

```
Age is greater than 18
```

Java IF-else Statement

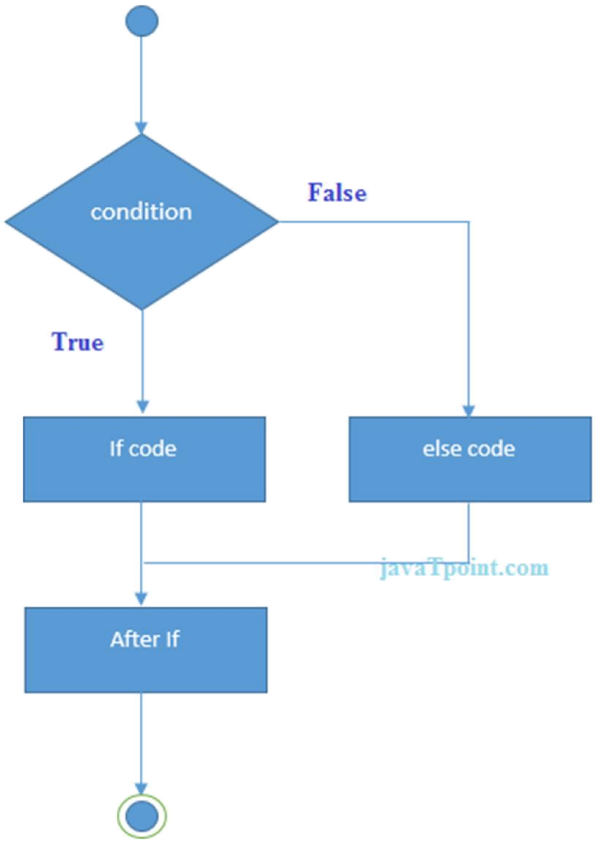
The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

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

Example:

```
public class IfElseExample {
public static void main(String[] args)
 {
   int number=13;
   if(number%2==0){
      System.out.println("even number");
   }
Else
{
      System.out.println("odd number");
   }
}
}
```



Output:

```
odd number
```

Java Nested if – else statement

The Nested if-else statement executes one if or else if statement inside another if or else if statement.

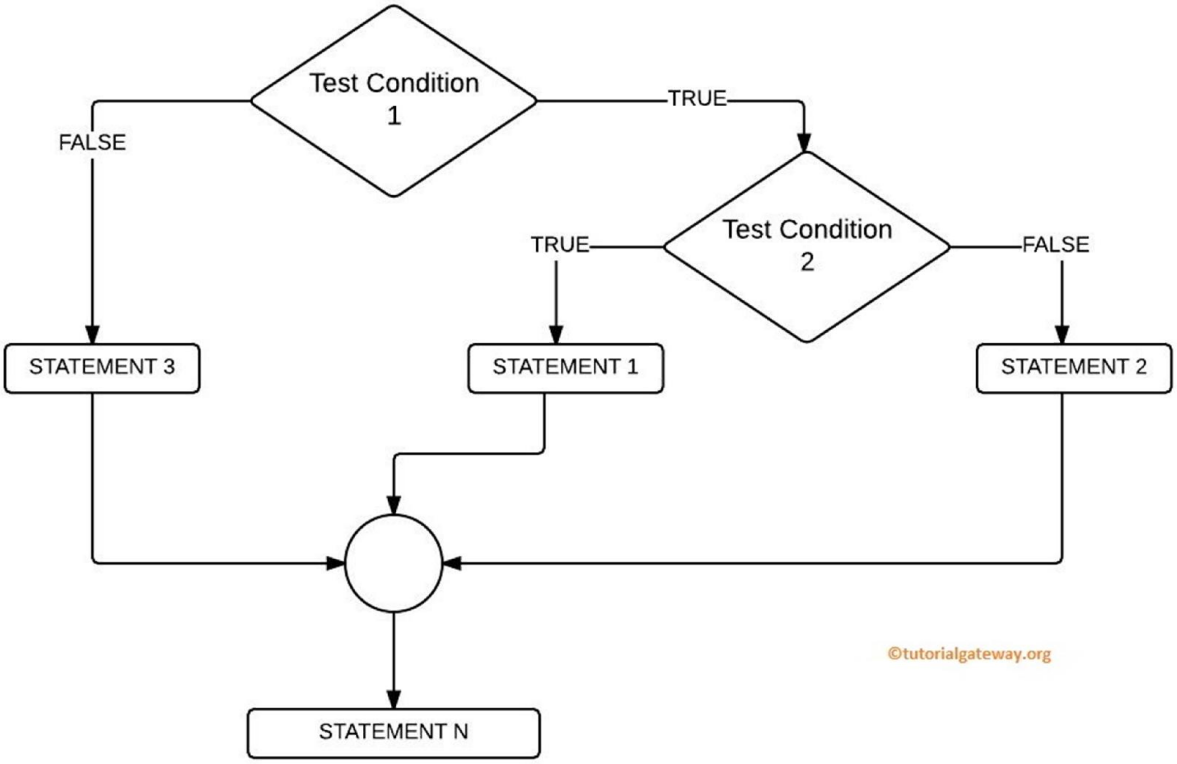
Syntax:

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

Example:

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

```
    int x = 30;
    int y = 10;
    if( x == 30 )
{
      if( y == 10 )
{
        System.out.print("X = 30 and Y = 10");
      }
    }
  }
}
```



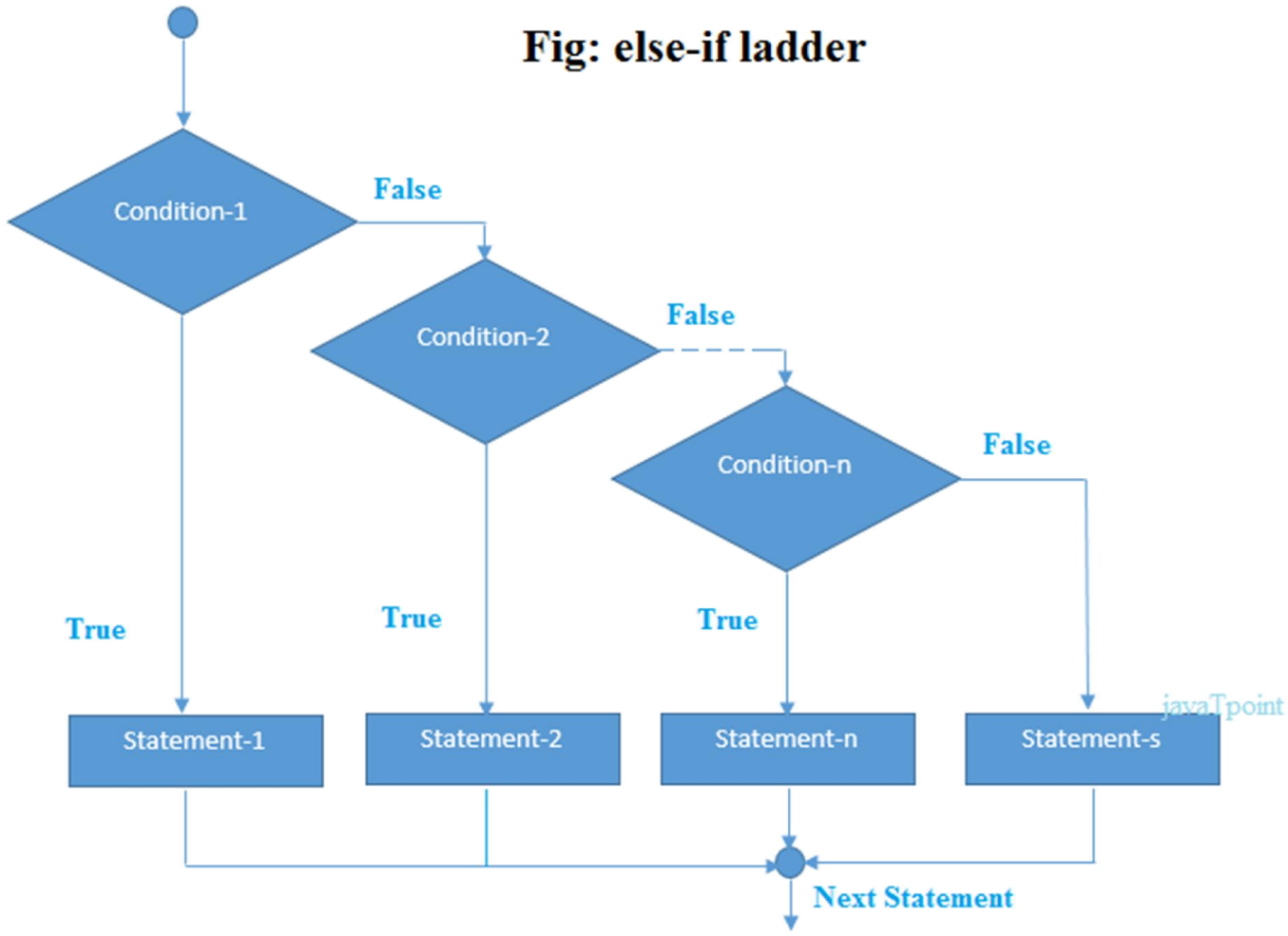
Java IF-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1)
{
//code to be executed if condition1 is true
}else if(condition2)
{
//code to be executed if condition2 is true
}
else if(condition3)
```

```
{
//code to be executed if condition3 is true
}
...
Else
{
//code to be executed if all the conditions are false
}
```



Example:

```
public class IfElseIfExample
 {
public static void main(String[] args)
 {
   int marks=65;
   if(marks<50)
{
      System.out.println("fail");
   }
```

```
    else if(marks>=50 && marks<60)
{
    System.out.println("D grade");
   }
   else if(marks>=60 && marks<70)
{
    System.out.println("C grade");
   }
   else if(marks>=70 && marks<80)
{
    System.out.println("B grade");
   }
   else if(marks>=80 && marks<90)
{
    System.out.println("A grade");
   }else if(marks>=90 && marks<100)
{
    System.out.println("A+ grade");
   }else{
    System.out.println("Invalid!");
   }
}
}
```

Output:

C grade

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
.....
```

```
default:
 code to be executed if all cases are not matched;
}
```

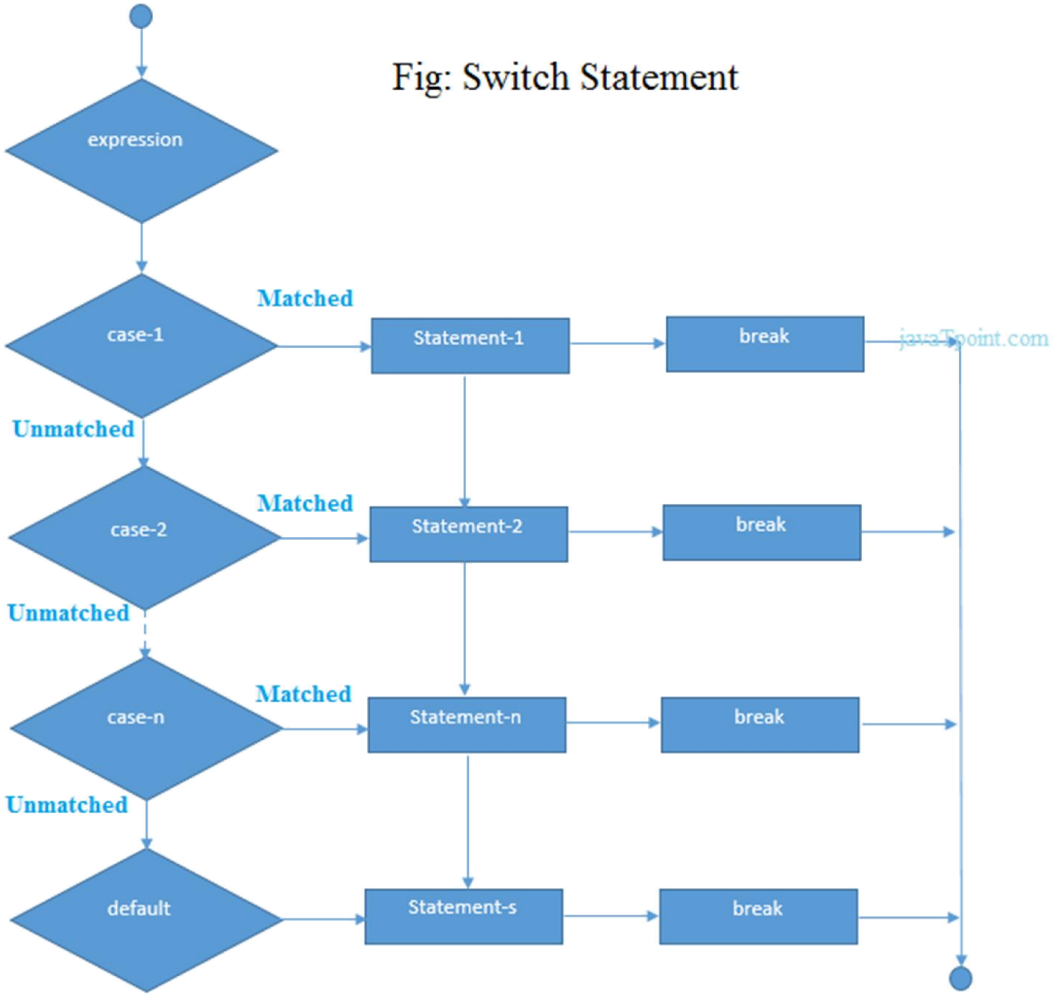


Fig: Switch Statement

Example:

```
public class SwitchExample {
public static void main(String[] args) {
   int number=20;
   switch(number){
   case 10: System.out.println("10");break;
   case 20: System.out.println("20");break;
   case 30: System.out.println("30");break;
   default:System.out.println("Not in 10, 20 or 30");
   }
}
}
```

Output: 20

Iteration Statement:

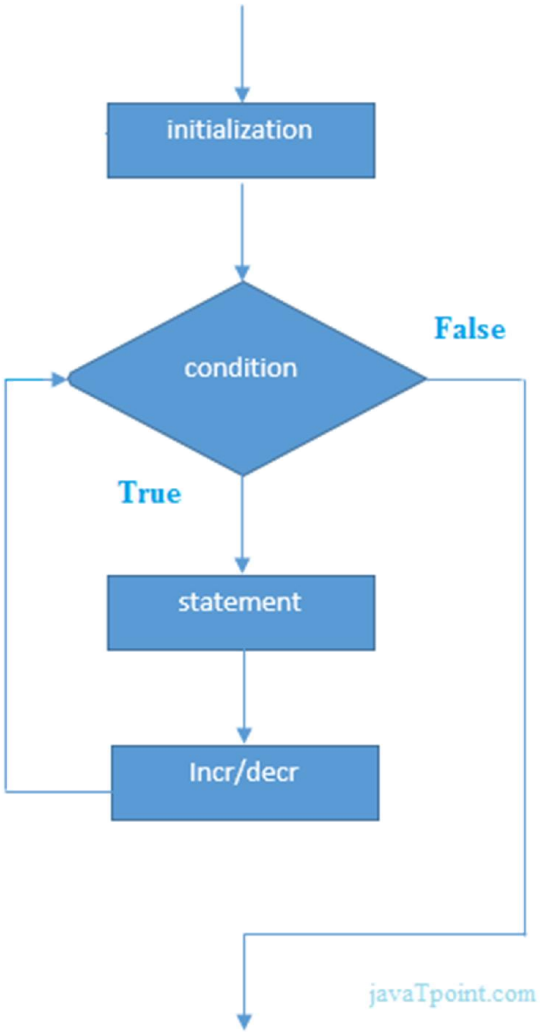
- The process of repeatedly executing a statements and is called as looping. The statements may be executed multiple times (from zero to infinite number).
- If a loop executing continuous then it is called as Infinite loop. Looping is also called as iterations.
- In Iteration statement, there are three types of operation:
    1. for loop
    2. while loop
    3. do-while loop

Java Simple For Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

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



Example:

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

Output:

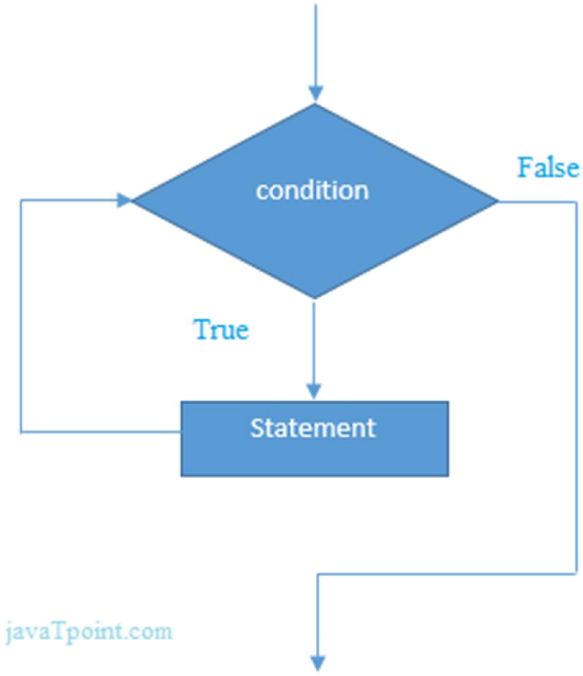
```
1
2
3
4
5
6
7
8
9
10
```

Java While Loop

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

Syntax:

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



Example:

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

  int i=1;
  while(i<=10){
    System.out.println(i);
  i++;
  }
}
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

Syntax:

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

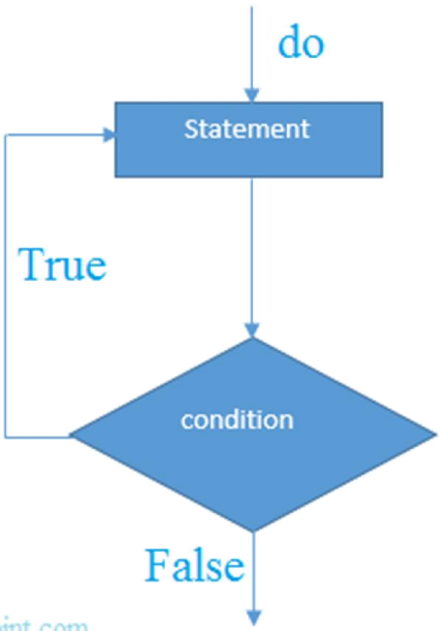
Java do-while Loop

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

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

Syntax:

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



Example:

```
public class DoWhileExample {
public static void main(String[] args) {
  int i=1;
  do{
    System.out.println(i);
  i++;
  }while(i<=8);
}
}
```

Output:

```
1
2
3
4
5
6
7
8
```

Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

Syntax:

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

JUMPS IN STATEMENT:

- Statements or loops perform a set of operations continually until the control variable will not satisfy the condition.
- But if we want to break the loops when condition will satisfy then Java give a permission to jump from one statement to end of loop or beginning of loop as well as jump out of a loop.
- "break" keyword use for exiting from loop and "continue" keyword use for continuing the loop.
- Break statement is used to terminate from a loop while a test condition is true.

1. Break statement
2. Continue statement

Java Break Statement

The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Syntax:

```
jump-statement;
break;
```

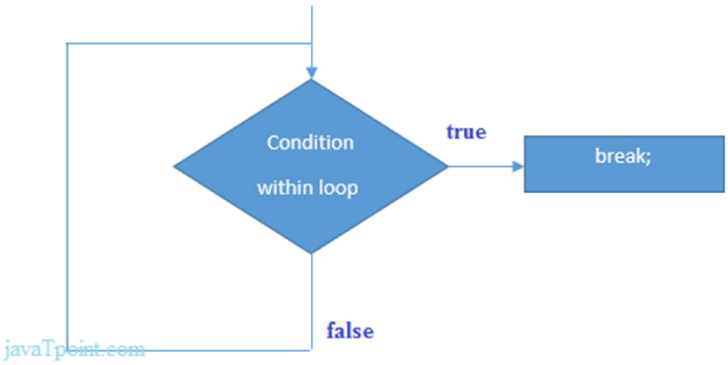


Figure: Flowchart of break statement

Example:

```
public class BreakExample
 {
public static void main(String[] args)
{
   for(int i=1;i<=10;i++)
{
      if(i==5)
{
         break;
      }
      System.out.println(i);
   }
}}
```

Output:

```
1
2
3
4
```

Java Continue Statement

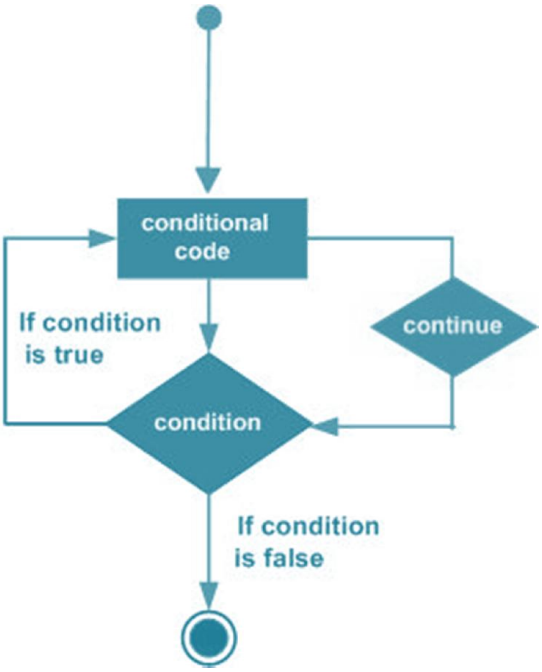
The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Syntax:

```
jump-statement;
continue;
```

Example:

```
public class ContinueExample
{
public static void main(String[] args)
{
   for(int i=1;i<=10;i++){
      if(i==5){
         continue;
      }
      System.out.println(i);
   }
}
}
```



Output:

```
1
2
3
4
6
7
8
9
10
```

UNIT – 2

Classes, Objects and Methods - Constructors - Methods Overloading-Inheritance-Overriding Methods - Finalizer and Abstract Methods-Visibility Control –Arrays, Strings and Vectors-StringBuffer Class-Wrapper Classes.

CLASSES OBJECTS AND METHOD:

- A class is a user defined data type with a template.
- It is a group of object which have common properties.
- After class is defined variables (instances) will be declared.

Syntax:

```
Class classname[extends superclass name]
//optional
{
 [Fields declaration:]    //optional
 [Methods declaration:]   //optional
}
```

- Classname and superclass name are any valid java identifiers.
- The keyword extends indicates that the properties of the superclass are extended to the classname.

For Example:

Proffessors:

1. State – subject handling. (Mp,Java)
2. Behaviour – teaching.

```
Class proffessorAnand extends HOD
{
Deg BCA,Bsc(phy);
void Teaching(Deg BCA, Deg Phy)
{
mp = BCA;
Java = phy;
}
```

FIELDS DECLARATION:

- Data is encapsulated in a class by placing data fields inside the body of the class.
- These variables called instance variable.
- Because they created whenever an object of the class is instantiated.

Ex:

```
Class Rectangle
{
    Int length;
   Int  width;     [also can declare as int length , width;]
}
```

- The class Rectangle contains two integer type instance variables.
- Instance variable doesn't get memory at compile time.
- It gets memory at run time when object is (instance) created.
- That's why it is called instance variable.

METHODS DECLARATION:

- A class with only data fields (no methods) has no life.
- Add methods that are necessary for manipulating the data in the class.
- Methods are declared inside the body of the class after the declaration of instance variables.

Syntax:

```
Return_type methodname(parameter-list)
{
Method–body;
}
```

Method declarations have four basic parts:

1. The name of the method (method name).
2. The type of the value the method returns(type).
3. A list of parameters(parameter – list).
4. The body of the method.

- The type specifies the type of value the method world return. It could be void type, if the method does not return any value.
- The method name is a valid identifier.
- The parameter list is always enclosed in parenthesis. The list contains variable and its types.

Ex: void getdata(int a, int b);

- The body actually describes the operation to be performed on the data.

Example:

```
        Class Rectangle
{
          int length;
           int width;                    \\ field declaration
           void getdata(int x, int y)    \\ method declaration
     {
          Length = x;
          Width = y;
   } }
```

- Return type is void because it does not return any value.
- We pass two integer values to the method, which are then assigned to the instance variable.

Ex:

```
Class Rectangle
{
 int length, width;
 void getData (int x, int y)
   {
     Length = x;
     Width = y;
 }
 int rectArea()
{
   int area = length* width;
     return(area);
}
    }
```

The new method rectArea() computes area of the rectangle and returns the result.

ACCESSING INSTANCE VARIABLE:

- An Instance variable within a method couldn't be accessible from other method.

Ex:

```
Class Access
   {
      int x;
      void method1()
      {
         int y;
         X = 10;      \\ legal
         Y= x;        \\ legal
    }
      void method2()
      {
         int z;
         X=5;         \\ legal
         Z =10;       \\ legal
         Y =1;        \\ illegal   couldn't access y.
} }
```

OBJECT:

- An object is a Real world entity.
- It is Run time entity
- It has state and behaviour.
- It is an instance of a class.

CREATING OBJECTS:

- An object in java is essentially a block of memory that contains space to store all the instance variable.
- Objects in java are created using the new operation.
- The new operator creates an object of the specified class and returns a reference to that object.

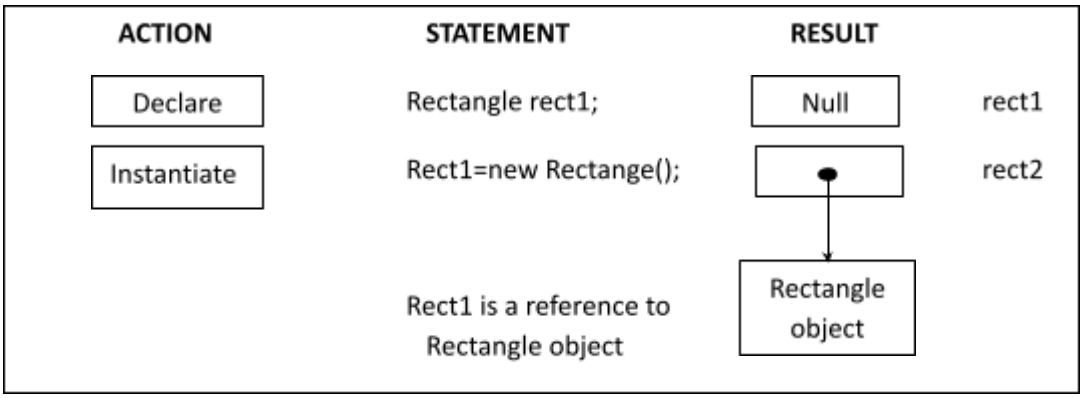
Example:

```
Rectangle rect1;                    \\ Declare the object.
```

Rect1 = new Rectangle (); \\ instantiate the object.

- The first statement declares a variable to hold the object reference.
- The second one actually assigns the object reference to the variable.
- Both statements can be combined into one.

Rectangle rect1 = new Rectangle();



Accessing class members:

- Every object containing its own set of variable.
- To access the class members follow the below syntax.

Syntax:

Objectname.variablename = value;

Objectname.Methodname(parameter-list);

Where,

Object name	= object.
Variable name	= name of the instance variable.
Value	= value for that.
Method name	= method which want to call.
Parameter-list	= actual values.

Example:

```
Main ()
{
int area1.area2;
Rectangle rect1 =new Rectangle();        \\creating object1
Rectangle rect2 =new Rectangle();        \\creating object2
rect1.lenth =15;                         \\accessing variable.
rect1.width =10;                         \\accessing variable.
```

```
Area1 =rect1.length*rect1.width;
rect2.getData (20, 12);                    \\accessing variable.
Area2 =rect2.rectArea();
}
rect1.length =15;
rect1.width =10;
rect2.length =20;
rect2.width =12;
```

Two object rect1 and rect2 store different values as shown below.

	rect1	rect2
Length	15	20
Width	10	12

Example program1:

```
Class Rectangle                    // Class declaration
{
int length, width;                 // field declaration (instance variable)
void getData(int x, int y)         //Method1 declaration
{
length =x;
width =y;
}
int rectArea()                     //Method2 declaration
{
int area =length*width;
return (area);
}}
Class rectArea
{
public static void main(string area [ ])
{
int area1, area2;
Rectangle rect1 =new Rectangle ();      \\ creating objects1
Rectangle rect2=new Rectangle ();       \\ creating objects2
rect1.length =15;                       \\ Accessing variable for Object1
rect1.width =10;                        \\ Accessing variable for Object1
Area1=rect1.length*rect1.width;
rect2 .getData(20,12);                  \\ Accessing method for Object2
Area2=rect2.rectArea();                 \\ Accessing method for Object2
system.out.println("Area1="+area1);
```

```
System.out.println(“ Area2=”+ area2);
}}
```

Output:

Area1=150
Area2=240

Example2:

```
Class employee                          //class declaration
{
int id;                                 //Field declaration (Instance variable)
string name;                            //Field declaration (Instance variable)
float salary;                           //Field declaration (Instance variable)
void insert(int i,string n,float s)     //Method1 delared
{
id=i;
name=n;
salary=s;
}
void display ( )                        //Method2 declared
{
system.out.println(id+” “+name” “+salary);
}
public class Emp
{
public static void main (string [ ]args)
{
Employee e1= new Employee ();           //Object1 declared
Employee e2=new employee ();            //Object2 declared
Employee e3=new employee ();            //Object3 declared
e1.insert(101,”ajeet”, 45000);          // Accessing of instance variable for object1
e2.insert(102,”irfan”, 25000);          // Accessing of instance variable for object2
e3.insert(103,”nakul”, 55000);          // Accessing of instance variable for object3
e1.display ();                          // Accessing of method
e2.display ();                          // Accessing of method
e3.display ();                          // Accessing of method
```

}}

output:

CONSTRUCTOR:

- **Constructor in java** is a *special type of method* that is used to initialize the object.
- Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.
- Constructs construct the values for object.

Rules for creating java constructor

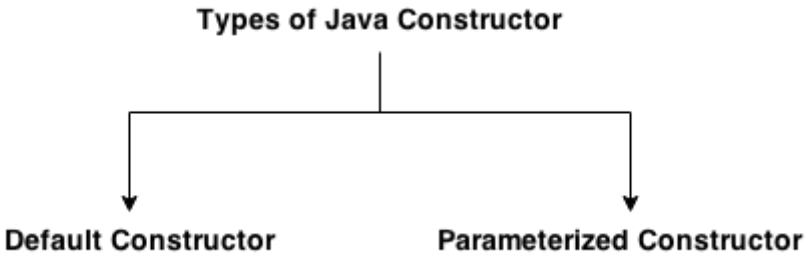
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

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



1. Java Default Constructor

A constructor that have no parameter is called default constructor.

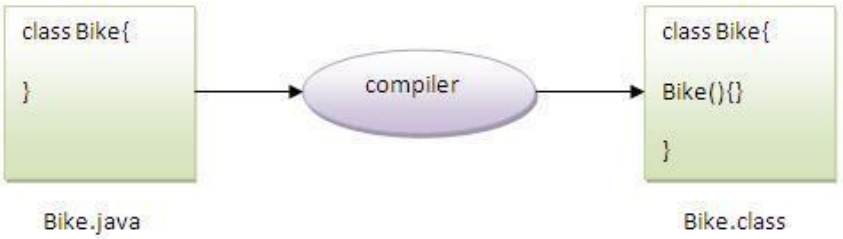
Syntax:

```
<class_name>()
{
Constructor body
}
```

Example 1: (Default constructor)

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

```
class Bike1
{
Bike1()    // constructor declared
{
System.out.println("Bike is created");
}
public static void main(String args[])
{
```




```
Bike1 b=new Bike1();  // constructor called
}}
```

Output:

Bike is created

Example 2: (Default constructor)

```
class Student
{
int id;
String name;
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[])
{
```

```
Student s1=new Student();          // Default constructor called
Student s2=new Student();          // Default constructor called
s1.display();
s2.display();
}}
```

Output:

```
0 null
0 null
```

In the above class, we are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

2. Java parameterized constructor

- A constructor that have parameters is known as parameterized constructor.
- Parameterized constructor is used to provide different values to the distinct objects.

Syntax:

```
<class_name>(parameter list)
{
Constructor body
}
```

Example: (parameterized constructor)

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

```
class Student
{
  int id;
  String name;
  Student(int i, String n)
{
  id = i;
  name = n;
  }
  void display()
{
System.out.println(id+" "+name);
}
  public static void main(String args[])
{
```

```
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

Output:

111 Karan
222 Aryan

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

JAVA STATIC KEYWORD

- The **static keyword** in java is used for memory management mainly.
- Java static keyword used with variables, methods, blocks and nested class.
- The static keyword belongs to the class than instance of the class.
- When the static keyword is used then no need to instantiate the object.

Example:

```
class Calculate
    {
     static int cube(int x)            // static keyword initiated
    {
     return x*x*x;
```

```
    }
    public static void main(String args[])
   {
    int result=Calculate.cube(5);  // As static keyword initiated no need of object.
    System.out.println(result);
    }
   }
```

Output: 125

METHOD OVERLOADING

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- Method overloading is used when objects are required to perform similar tasks but using different input parameters.

Different ways to overload the method

There are two ways to overload the method in java

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

1. Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

EXAMPLE:

```
class Adder
{
static int add(int a,int b)     // method 1  with two parameter declared
{
return a+b;
}
static int add(int a,int b,int c)   // method 2 with three parameter declared
{
return a+b+c;
}}
class TestOverloading1
{
public static void main(String[] args)
{
```

```
System.out.print(Adder.add(11,11));         // calling method 1
System.out.print(Adder.add(11,11,11));     // calling method 2
}}
```

Output: 22 33

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

2. Method Overloading: changing data type of arguments

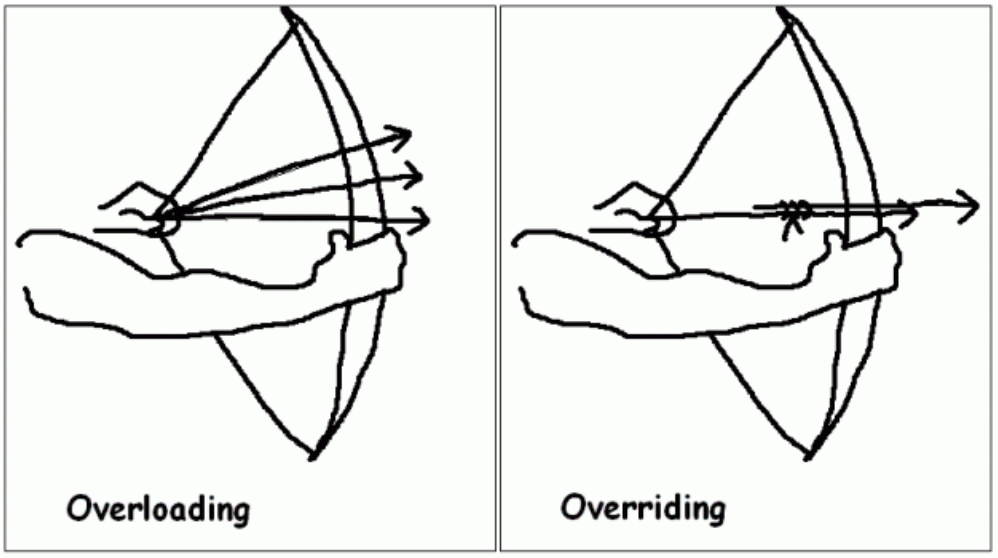
- In this example, two methods are created that differs in data type.
- The first add method receives two integer arguments
- Second add method receives two double arguments.

EXAMPLE:

```
class Adder
{
static int add(int a, int b)    // Method1 declared with int data type
{
return a+b;
}
static double add(double a, double b)     //Method2 declared with double data type
{
return a+b;
} }
class TestOverloading2
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));      // calling Method1
System.out.println(Adder.add(12.3,12.6));  // Calling Method2
}}
```

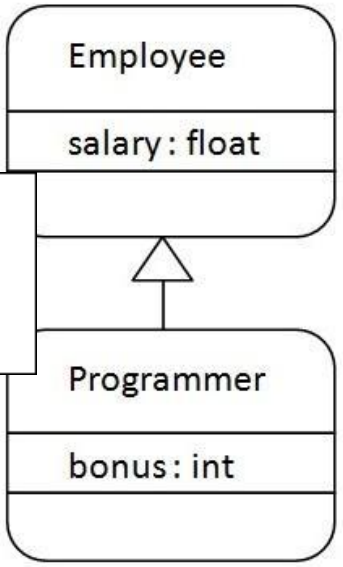
OUTPUT:

```
22
24.9
```



INHERITANCE:

- The mechanism of deriving a new class from an old class is called inheritance.
- The old class is known as the Base class or Parent class
- The new class is called the Sub class or Derived class or Child class.
- The inheritance allows subclasses to inherit all the variables and methods of their parent classes.



SYNTAX

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

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

Java Inheritance Example

- As displayed in the figure
- Programmer is the subclass and Employee is the superclass.
- Relationship between two classes is **Programmer IS-A Employee**.
- It means that Programmer is a type of Employee.

Example:

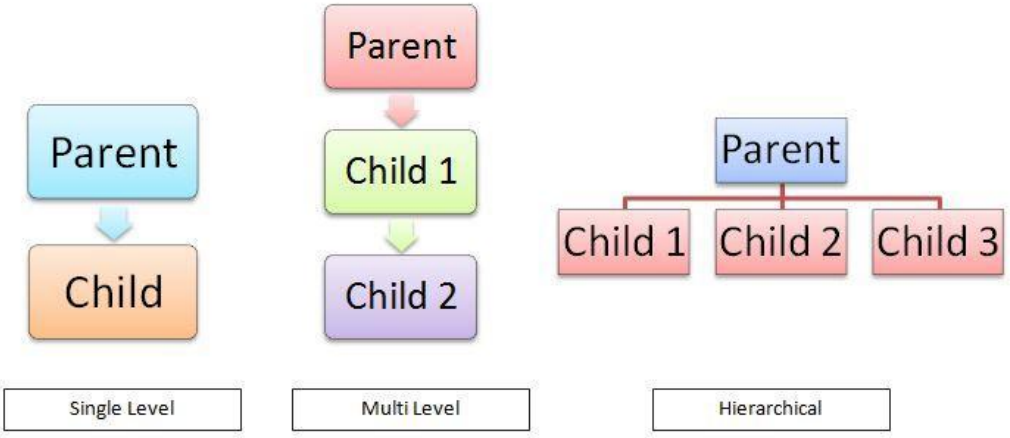
```
class Employee{
float salary=40000;
}
```

```
class Programmer extends Employee
{
int bonus=10000;
public static void main(String args[])
{
Programmer p=new Programmer();
System.out.println("Programmer salary is:"+p.salary);
System.out.println("Bonus of Programmer is:"+p.bonus);
} }
```

OUTPUT:
Programmer salary is:40000.0
Bonus of programmer is:10000

TYPES OF INHERITANCE:

- On the basis of class, there can be three types of inheritance in java
    1. Single
    2. Multilevel
    3. Hierarchical
- Multiple and hybrid inheritance is supported through interface only.



SINGLE INHERITANCE:

- One sub class is derived from one super class
- From the above diagram class A serves as a Super class or parent class
- Class B serves as a Base class or Sub class.

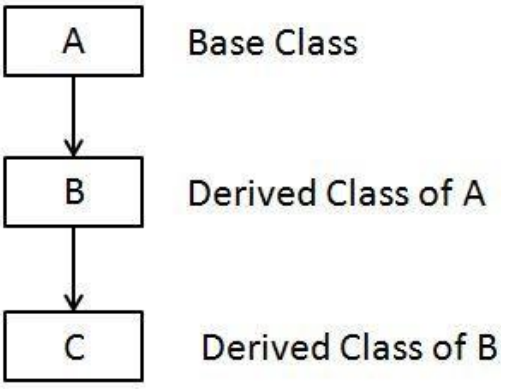
Example:

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

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

OUTPUT:

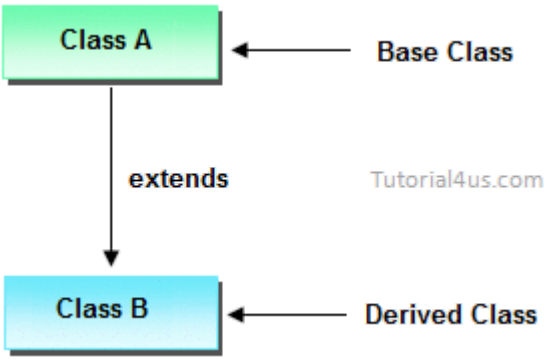
```
barking...
eating...
```



From the above example, A class **Animal** is Super class and class **Dog** is Sub class. Class **Dog** inherits the properties of Class **Animal**.

MULTILEVEL INHERITANCE:

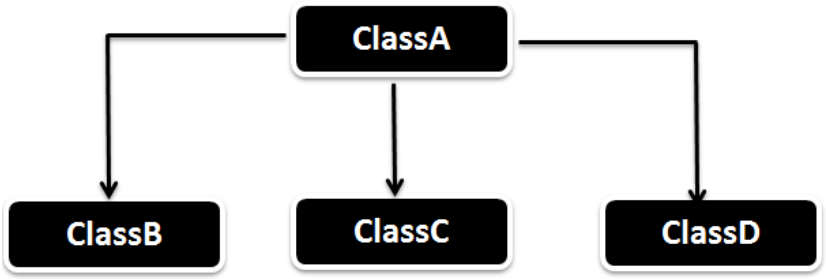
- In Multilevel inheritance there is a concept of grandparent class.
- From the above diagram then class C inherits class B and class B inherits class A
- Which means B is a parent class of C and A is a parent class of B.
- So in this case class C is implicitly inheriting the properties and method of class A along with B that's what is called multilevel inheritance.

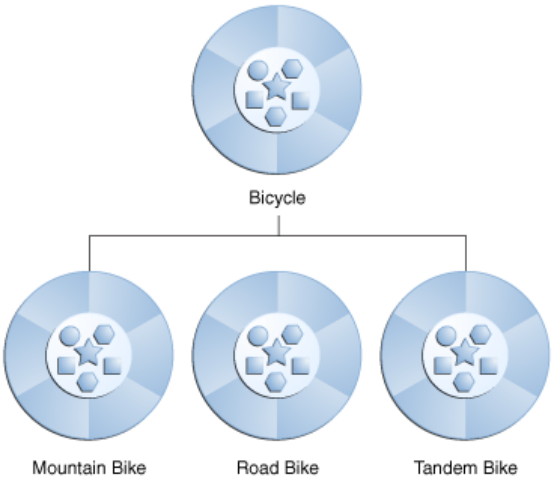


Example:

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

```
System.out.println("barking...");
}}
class BabyDog extends Dog
{
void weep()
{
System.out.println("weeping...");
}}
class TestInheritance2
{
public static void main(String args[])
```





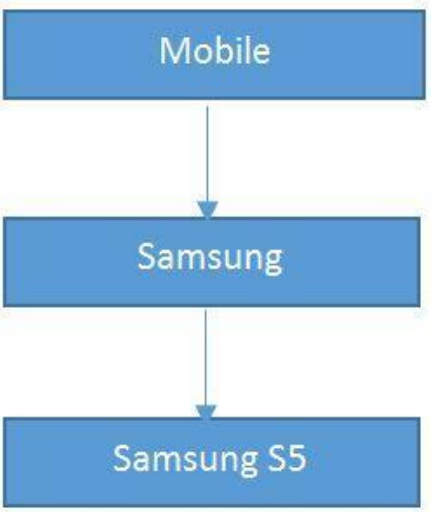

```
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

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

HIERARCHICAL INHERITANCE:

- As in the below diagram that a class has more than one child classes (sub classes).
- In other words more than one child classes have the same parent class then such kind of inheritance is known as hierarchical.
- Class A is a parent class of class B and class C and class C.
- One Parent class has many sub classes.



EXAMPLE:

```
class Animal                    // PARENT CLASS
{
void eat()
{
System.out.println("eating...");
} }
class Dog extends Animal        // Sub class extends parent class
{
```

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

Output:

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

FINAL KEYWORD:

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

1. Variable
2. Method
3. class

THE FINAL VARIABLE:

- The final keyword can be applied with the variables.
- If the variable as final, then the variable cannot change the value of final variable (It will be constant).
- A final variable that have no value it is called blank final variable or uninitialized final variable.
- It can be initialized in the constructor only.

Example:

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

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

Output: Compile Time Error

FINAL METHOD:

- If the java Method made as final, then it won't be overridden.

Example of final method

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

Output: Compile Time Error

FINAL CLASS:

- If the class has final, then the properties of that class couldn't inherits.

Example of final class

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

Output: Compile Time Error

METHOD OVERRIDDING:

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

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

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

Example of method overriding

- In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation.

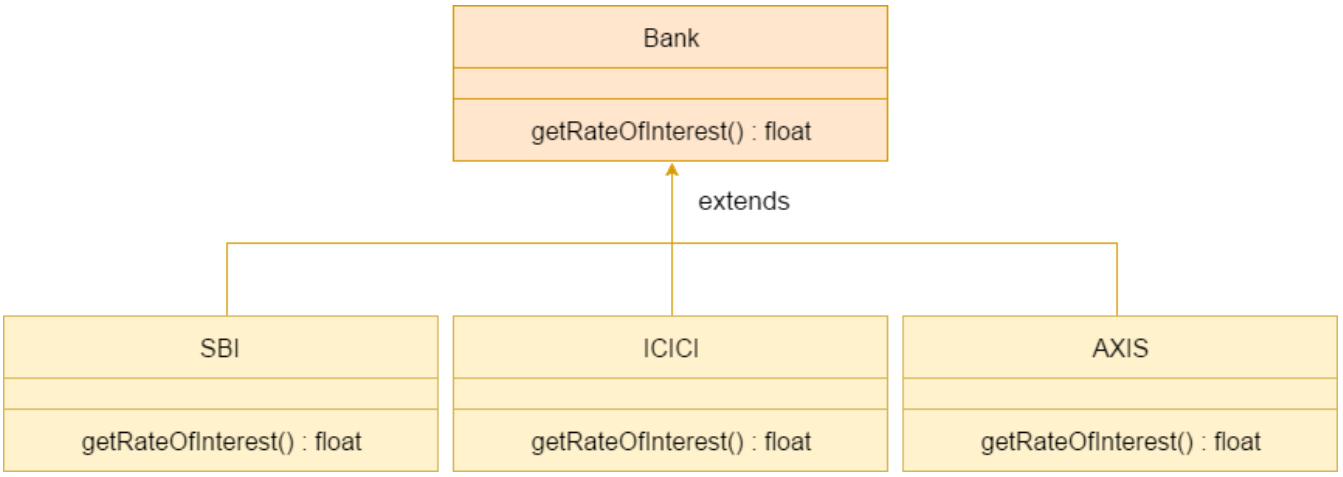
- The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
class Vehicle
{
void run()
{
System.out.println("Vehicle is running");
}}
class Bike2 extends Vehicle
{
void run()
{
System.out.println("Bike is running safely");
}
public static void main(String args[])
{
Bike2 obj = new Bike2();
obj.run();
}
```

Output: Bike is running safely

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



EXAMPLE PROGRAM:

```
class Bank
{
int getRateOfInterest()
```

```
{return 0;}
}
 class SBI extends Bank
{
int getRateOfInterest()
{return 8;}
}
class ICICI extends Bank
{
int getRateOfInterest()
{return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}

class Test2
{
public static void main(String args[])
{
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
} }
```

Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9

DIFFERENCE BETWEEN METHOID OVERLOADING AND OVERRIDING

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.

In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

FINALIZER AND ABSTRACT METHOD:

FINALIZER (finalize()**):**

- Java run time is an automatic garbage collecting system. It automatically frees up the memory resources used by the objects.
- But objects may hold other non-object resources such as file descriptors or window system fonts. The garbage collector cannot free these resources.
- In order to free these resources finalizer method can be used.
- This is similar to destructors in C++.
- The finalizer method is simply **finalize()** and can be added to any class.
- Java calls that method whenever it is about to reclaim the space for that object.

SYNTAX:

```
protected void finalize() throws Throwable
{
    //Keep some resource closing operations here
}
```

ABSTRACT CLASS AND METHODS:

- If a class contain any abstract method then the class is declared as **abstract class**.
- An abstract class is never instantiated.

SYNTAX:

abstract class class_name()

{ class body }

- A Method that is declared without any body within an abstract class are called **abstract method**.
- The method body will be defined by its subclass.
- Abstract method can never be final and static.
- Any class that extends an abstract class must implement all the abstract methods declared by the super class.

EXAMPLE:

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

OUTPUT:

running safely..

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

VISIBILITY CONTROL (VISIBILITY ACCESS) (or) ACCESS MODIFIERS

- It is possible to inherit all the members of a class by a subclass using the keyword **extends.**
- The variables and methods of a class are visible everywhere in the program.
- However, it may be necessary in some situations we may want them to be not accessible outside.
- We can achieve this in Java by applying *visibility modifiers* to instance variables and methods.
- The visibility modifiers are also known as *access modifiers.*

- Access modifiers determine the accessibility of the members of a class.
- Java provides three basic types of visibility modifiers:
    1. **Public**
    2. **private**
    3. **protected**
    4. **Friendly(Default)**
    5. **Private protected**
- They provide different levels of protection as described below.

PUBLIC ACCESS

- Any variable or method is visible to the entire class in which it is defined.
- To make a member accessible outside with objects, we simply declare the variable or method as public.
- A class, method, constructor, interface, etc. declared public can be accessed from any other class.
- A variable or method declared as **public** has the widest possible visibility and accessible everywhere.

Ex:

```
public static void main(String[] arguments) {
   // ...
}
```

Friendly Access (Default):

- When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access.
- The difference between the "public" access and the "friendly" access is that the **public** modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages.

Protected Access:

- The visibility level of a "protected" field lies in between the public access and friendly access.
- That is, the **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages.
- The protected access modifier cannot be applied to class and interfaces.

Private Access:

- Private fields have the highest degree of protection.
- They are accessible only with their own class.
- They cannot be inherited by subclasses and therefore not accessible in subclasses.
- A method declared as private behaves like a method declared as final.
- It prevents the method from being subclasses.

Private protected Access:

- A field can be declared with two keywords **private** and **protected** together.

Private protected int example;

- This gives a visibility level in between the "protected" access and "private" access.
- This modifier makes the fields visible in all subclasses regardless of what package they are in.
- These fields are not accessible by other classes in the same package.

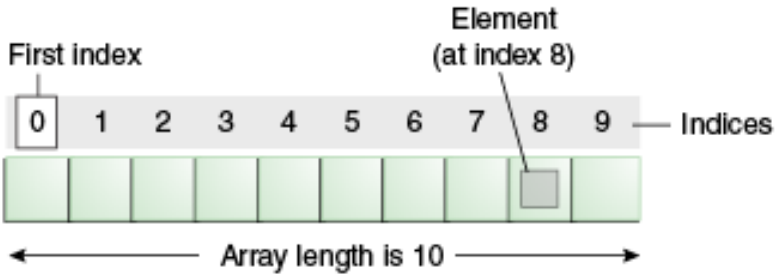
The following table summarises the visibility provided by various access modifiers.

Access modifier	public	protected	Friendly	private protected	private
Own class	✔	✔	✔	✔	✔
Sub class In same package	✔	✔	✔	✔	
Other classes In same package	✔	✔	✔		
Sub class in other package	✔	✔		✔	
Other classes In other package	✔				

JAVA ARRAY

- Array is a collection of similar type of elements that have contiguous memory location.
- **Java array** is an object that contains elements of similar data type.
- It is a data structure where we store similar elements.
- We can store only fixed set of elements in a java array.

- Array in java is index based, first element of the array is stored at 0 index.



Types of Array in java

There are two types of array.

1. Single Dimensional Array
2. Multidimensional Array

SINGLE DIMENSIONAL ARRAY (one dimensional array):

- A list of items can be given one variable name using only one subscript and such a variable is called one dimensional array.
- One dimensional array is a list of variables of same type that are accessed by a common name.
- An individual variable in the array is called an array element.

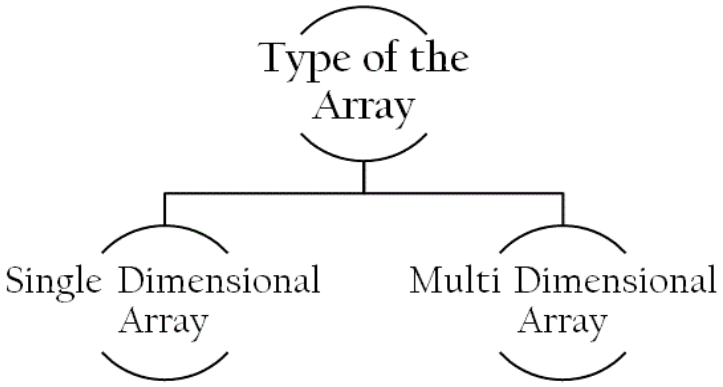
Declaration of arrays:

- To declare an array below is a general form or syntax.

Datatype arrayName[ ];

Where, **Datatype** is valid data type in java and **arrayName** is a name of an array.

Example : int physics[ ];



Creation of arrays

- To allocate space for an array element use below general form or syntax.

arrayName = new type[size];

Where **arrayName** is the name of the array and **type** is a valid java datatype and **size** specifies the number of elements in the array.

Example : physics = new int[10];

Above statement will create an integer of an array with ten elements that can be accessed by physics.

physics[0]
physics[1]
physics[2]
physics[3]
physics[4]
physics[5]
physics[6]
physics[7]
physics[8]
physics[9]

Structure of one dimensional array

- Note that array indexes begins with zero.
- That means if you want to access 1st element of an array use zero as an index.
- To access 3rd element refer below example.
- physics[2] = 10; // it assigns value 10 to the third element of an array.
- java also allows an **abbreviated syntax** to declare an array. General form or syntax of is is as shown below.

Initialization of an Array

The final step is to put values into the array created. This process is known as initialization.

Syntax:

arrayname[subscript] = value;

Example:

```
physics[0]=12;
physics[1]=15;
physics[2]=16;
physics[3]=18;
physics[4]=20;
physics[5]=23;
physics[6]=25;
physics[7]=27;
physics[8]=30;
physics[9]=35;
```

In The above example assigns the value as,

12
15
16
18
20
23
25
27
30
35

physics[0]
physics[1]
physics[2]
physics[3]
physics[4]
physics[5]
physics[6]
physics[7]
physics[8]

Example Program:

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

```
Output:10
       20
       70
       40
       50
```

Example For Declaration, Instantiation and Initialization of Java Array

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

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

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

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

TWO DIMENSIONAL ARRAY:

- Two Dimensional Array in Java is the simplest form of Multi-Dimensional Array.
- In Two Dimensional Array, data is stored in row and columns.
- The record can access using both the row index and column index.

Declaration of Two Dimensional Array in Java

Following the format of declaration of two dimensional array in Java Programming Language:

SYNTAX:

DataType ArrayName[][];
ArrayName = new DataType[][];
(or)
DataType ArrayName[][] = new DataType[][];

Data_type:

- This will decide 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 etc.

Array_Name:

- This is the name you want to give it to array.
- For example Car, students, age, marks, department, employees etc

Creating of Two Dimensional Array in java

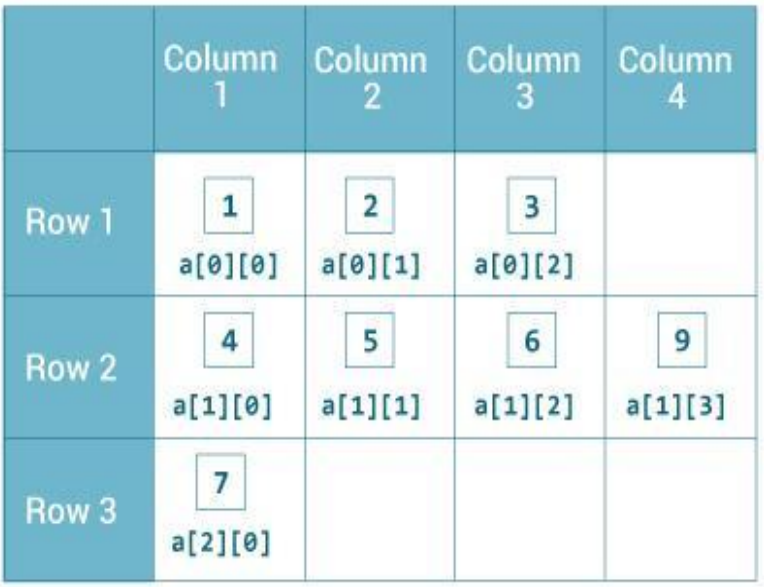
int[][] a = new int[3][4];

- Here, **a** is a two-dimensional array. The array can hold maximum of 12 elements of type int.
- Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1.

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Intialization of Array:

```
int[][] a = {
        {1, 2, 3},
        {4, 5, 6, 9},
        {7},
};
```



Example:

```
class MultidimensionalArray {
  public static void main(String[] args)
{
    int[][] a = {
        {1, 2, 3},
        {4, 5, 6, 9},
        {7},
    };
        System.out.println("Length of row 1: " + a[0].length);
    System.out.println("Length of row 2: " + a[1].length);
    System.out.println("Length of row 3: " + a[2].length);
  }
}
```

Output:

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

JAVA STRING

- String is basically an object that represents sequence of char values.
- An array of characters works same as java string.

For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

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

- String is a sequence of characters.
- String is an object that represents a sequence of characters.
- The java.lang.String class is used to create string object.
- There are two ways to create String object:

    1. By string literal
    2. By new keyword

1. String Literal

- Java String literal is created by using double quotes.

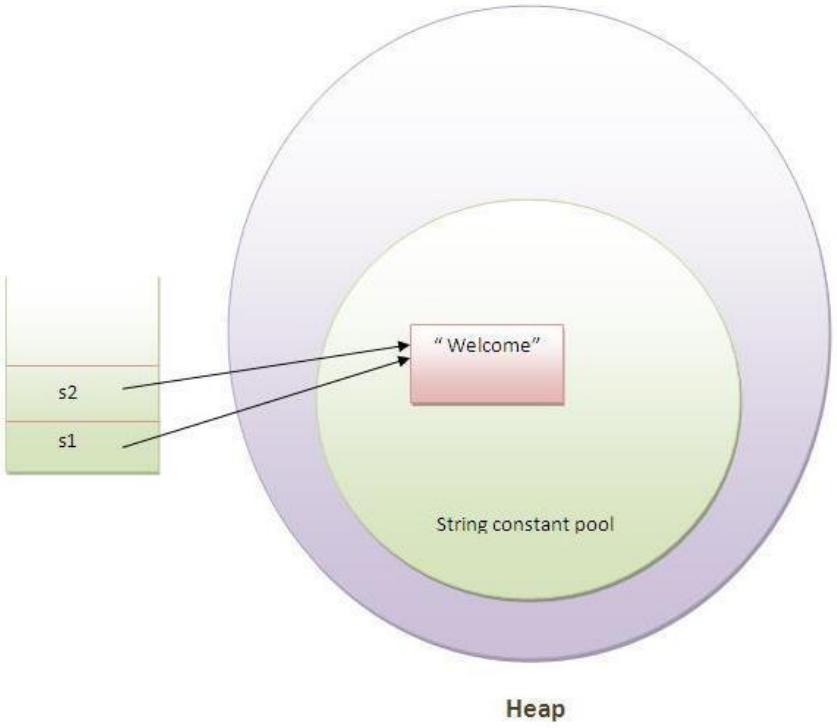
For Example:

```
String s="welcome";
```

- Each time you create a string literal, the JVM checks the string constant pool first.
- If the string already exists in the pool, a reference to the pooled instance is returned.
- If string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

```
String s1="Welcome";
String s2="Welcome";//will not create new instance
```



- In the above example only one object will be created.
- Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object.
- After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

> ***Note: String objects are stored in a special memory area known as string constant pool.***

2. New keyword

String s=**new** String("Welcome");//creates two objects and one reference variable

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

Java String Example

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

```
String s1="java";//creating string by java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

OUTPUT:

java

strings

example

SOME OF THE STRING METHODS:

No.	Method	Description
1	String.replace('x','y')	Replaces x with y
2	string.indexOf(int ch)	Returns specified char value index
3	string.startsWith('x')	Returns true if the x is equal to string
4	string.indexOf(String substring)	Returns specified substring index
5	string.equals(s1)	Returns true if string and s1 is equal.
6	String.toLowerCase()	Returns string in lowercase.
7	string.length()	Gives the length of string
8	String.toUpperCase()	Returns string in uppercase.
9	String.chartAt()	Gives n'th character of string
10	String.trim()	Removes beginning and ending spaces of this string.
11	string.concat(s2)	Concatenates string and s2

Examples for string methods:

Example 1: (Java String toUpperCase() and toLowerCase() method)

```
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
```

Example 2: (Java String trim() method)

```
String s="  Sachin  ";
System.out.println(s);//  Sachin
System.out.println(s.trim());//Sachin
```

Example 3: (Java String startsWith() and endsWith() method)

```
String s="Sachin";
System.out.println(s.startsWith("Sa"));//true
System.out.println(s.endsWith("n"));//true
```

Example 4: (Java String charAt() method)

```
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
```

StringBuffer class:

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

Commonly Used StringBuffer Methods.

METHOD	TASK
String1.append(string2)	Appends the string2 to string1 at the end.
String1.insert(n,str2)	Inserts str2 at the position n of the string1.
String1.replace(Bn,En,str2)	Replaces the str2 from the specified beginIndex(Bn) and endIndex(En).
String.delete(startIndex,endIndex)	Deletes the string from the specified beginIndex to endIndex.
String.reverse()	Reverses the current string

Example:

```
class StringBufferExample{
```

```
public static void main(String args[])
{
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");          //now original string is changed
System.out.println(sb);     //prints Hello Java
sb.insert(1,"Java");        //now original string is changed
System.out.println(sb);     //prints HJavaello
sb.replace(1,3,"Java");
System.out.println(sb);     //prints HJavalo
sb.delete(1,3);
System.out.println(sb);     //prints Hlo
sb.reverse();
System.out.println(sb);     //prints olleH
} }
```

VECTORS:

- Vectors are commonly used instead of arrays.
- Because they expand automatically when new data is added to them.
- Vectors can hold only Objects and not primitive types (eg, int).
- If you want to put a primitive type in a Vector, put it inside an object (eg, to save an integer value use the Integer class or define your own class).
- Vectors are implemented with an array, and when that array is full and an additional element is added, a new array must be allocated.
- But the vector automatically expands the size no need to new vector to be allocated.

Creating a vector:
Method 1:

Syntax: *Vector object= new vector ()*

Example: Vector v = new Vector(); //Declaring without size

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

Method 2:

Syntax: *Vector object= new vector(int initialcapacity)*

Example: Vector v = new Vector(3); // Declaring with size

It will create a Vector of initial capacity of 3.

Method 3:

Syntax: *Vector object= new vector(int initialcapacity, capacityIncrement)*

Example: Vector vec= new Vector(4, 6)

Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

Important Vector Methods.

Method	***Description***
v.add(o)	adds Object o to Vector v
v.add(i, o)	Inserts Object o at index i, shifting elements up as necessary.
v.clear()	removes all elements from Vector v
v.contains(o)	Returns true if Vector v contains Object o
v.firstElement(i)	Returns the first element.
v.get(i)	Returns the object at int index i.
v.lastElement(i)	Returns the last element.
v.isEmpty()	Returns True if empty otherwise False.
v.remove(i)	Removes the element at position i, and shifts all following elements down.
v.set(i,o)	Sets the element at index i to o.
v.size()	Returns the number of elements in Vector v.

Example:

```
import java.util.Vector;
public class BasicVectorOperations
{
   public static void main(String a[])
{
      Vector<String> vct = new Vector<String>();   // creating vector

//adding elements to the end
      vct.add("First");
      vct.add("Second");
      vct.add("Third");
      System.out.println(vct);

//adding element at specified index
      vct.add(2,"Random");
      System.out.println(vct);

//getting elements by index
      System.out.println("Element at index 3 is: "+vct.get(3));
```

```
//getting first element
    System.out.println("The first element of this vector is: "+vct.firstElement());

//getting last element
    System.out.println("The last element of this vector is: "+vct.lastElement());

//how to check vector is empty or not
    System.out.println("Is this vector empty? "+vct.isEmpty());
  }}
```

Output:

```
[First, Second, Third]
[First, Second, Random, Third]
Element at index 3 is: Third
The first element of this vector is: First
The last element of this vector is: Third
Is this vector empty? false
```

WRAPPER CLASS:

- A Wrapper class is a class whose object wraps or contains a primitive data types.
- It provides the mechanism to convert primitive into object and object into primitive.
- **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically.
- The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.
- The eight classes of *java.lang* package are known as wrapper classes in java.

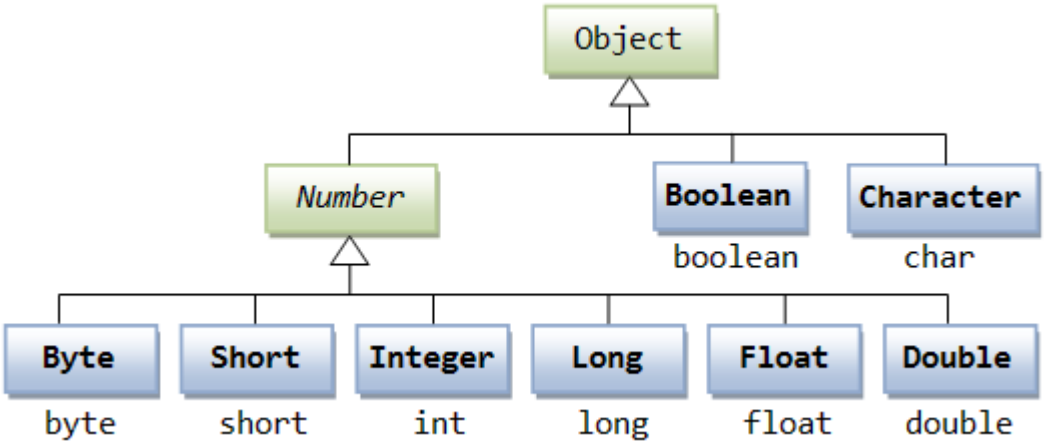
Need of Wrapper Class:

- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

The list of eight wrapper classes are given below:

PRIMITIVE TYPE	WRAPPER CLASS
Boolean	Boolean
Char	Character
Byte	Byte
Short	Short

Int	Integer
Long	Long
Float	Float
Double	Double



Autoboxing (Primitive to object):

- Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing.
- For example – conversion of int to Integer, long to Long, double to Double etc.

Example:

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

Output: 20 20 20

Unboxing (Object to Primitive):

- It is just the reverse process of autoboxing.
- Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing.
- For example – conversion of Integer to int, Long to long, Double to double etc.

Example:

```
public class WrapperExample2
{
```

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

Output: 3 3 3

UNIT 3

Interfaces – Packages - Creating Packages - Accessing a Package - Multithreaded Programming - Creating Threads - Stopping and Blocking a Thread - Life Cycle of a Thread - Using Thread Methods - Thread Priority - Synchronization - Implementing the Runnable Interface.

INTERFACES:

- An **interface in java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in java is **a mechanism to achieve abstraction**.
- There can be only abstract methods in the java interface not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- It cannot be instantiated just like abstract class.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
- Writing an interface is similar to writing a class. But a class describes the attributes and behaviours of an object. And an interface contains behaviours that a class implements.
- 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 INTERFACE:

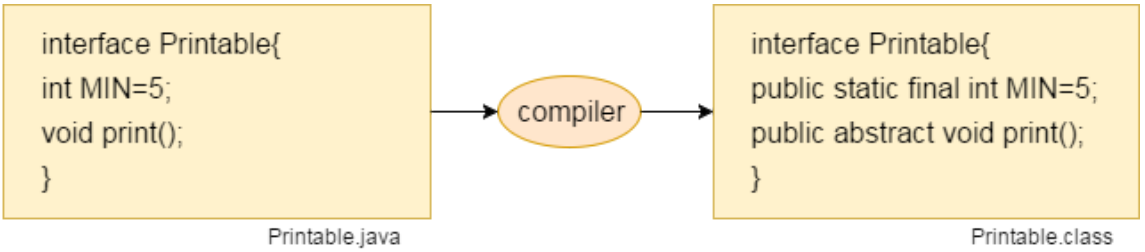
The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface –

SYNTAX:

```
public interface NameOfInterface
{
   // Any number of final, static fields
   // Any number of abstract method declarations
}
```

Interfaces have the following properties

- An interface is implicitly abstract. Not need to use the **abstract** keyword while declaring an interface.
- Each Fields in an interface is also implicitly **static** and **final**, so the static and final keyword is not needed (Refer below diagram).
- Methods in an interface are also implicitly public.



Example:

```
Interface ItemConstants          //interface declared
{
        int code = 1001;         // Variable declared in interface
        string name = "Fan";
        void display();          // Method declared in interface
}
```

VARIABLE DECLARATION:

```
Static final type VariableName=value;
```

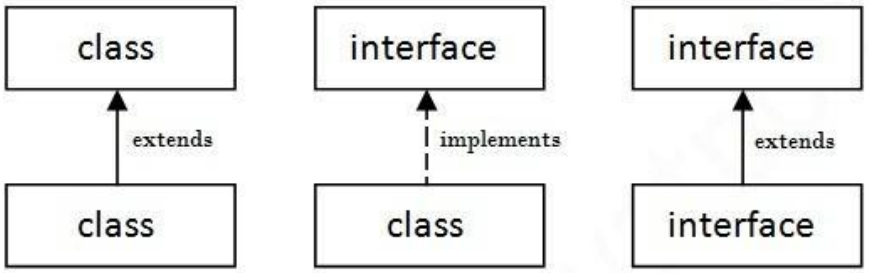
METHOD DECLARATION:

```
Return_type methodName (parameter_list);
```

The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.

EXTENDING INTERFACE:

- An interface can extend another interface in the same way that a class can extend another class.
- The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
- As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



SYNTAX:

```
interface Name2 extends Name1
{
      Body of Name2
}
```

- An interface can be subinterfaced from other interfaces.
- The new subinterface will inherit all the members of the subinterface in the manner similar to subclasses.

Example:

```
Interface ItemConstants
{
        int code = 1001;
        string name = "Fan";
}
Interface Item extends ItemConstants
{
void display();
}
```

<u>IMPLEMENTING INTERFACE:</u>

- Interfaces are used as "Super classes" whose properties are inherited by classes.
- It is therefore necessary to create a class that inherits the given interface.

SYNTAX:

```
Class className implements interfacename
{
        Body of classname
}
```

Example:

```
interface Drawable                          // Interface declared
{
void draw();
}
class Rectangle implements Drawable         //class implements interface
{
public void draw()
{
System.out.println("drawing rectangle");
}}
```

Example:

```
interface Printable                         // Interface 1 declared
{
void print();
}
interface Showable                          // Interface 2 declared
{
void show();
}
class A7 implements Printable,Showable      // implementing interface
{
public void print()
{
System.out.println("Hello");
}
public void show()
{
System.out.println("Welcome");
}
public static void main(String args[])
{
A7 obj = new A7();                          // object declaration
obj.print();
obj.show();
 } }
```

OUTPUT:

Hello
Welcome

JAVA PACKAGES:

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two types,

1. Java API package (Built-in package)

2. User-defined package. (Defined by user)

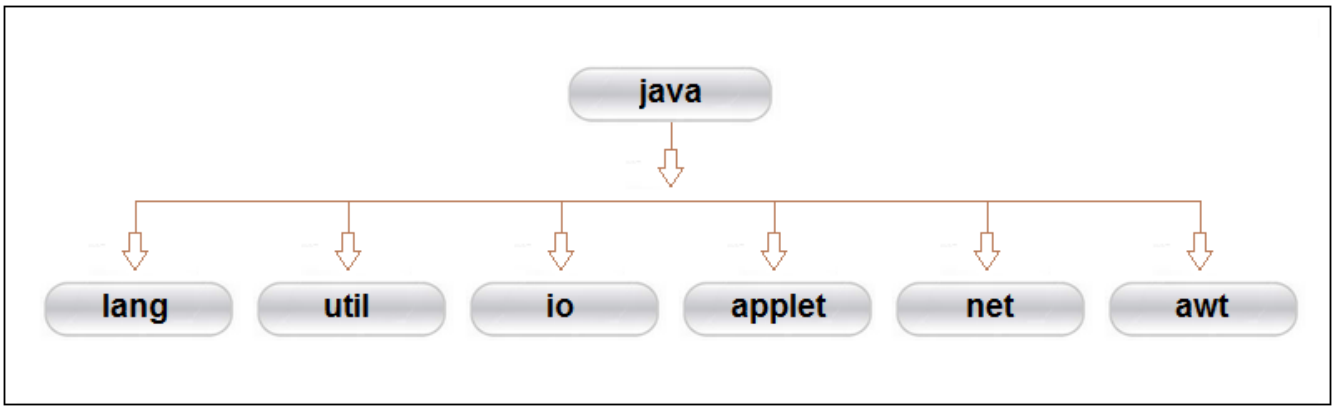
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

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

1. Java API Packages.

- Java **APl(Application Program Interface)** provides a large numbers of classes grouped into different packages according to functionality.
- Most of the time we use the packages available with the the Java API.
- Following figure shows the system packages that are frequently used in the programs.



PACKAGE NAME	CONTENTS
java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.

java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for creating and implementing applets.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

Using API Packages

- The import statements are used at the top of the file, before any class declarations.
- The first statement allows the specified class in the specified package to be imported.

For example,

Import java.awt.color;

- The above statement imports class color and therefore the class name can now be directly used in the program.
- The below statement imports every class contained in the specified package.

Import java.awt.*;

- The above statement will bring all classes of java.awt package.

CREATING PACKAGES:

- To create a package, a name should be selected for the package.
- Include a **package** statement along with that name at the top of every source file that contains the classes, interfaces.
- 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.

Example:

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

To compile and run the package,

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

- The -d is a switch that tells the compiler where to put the class file i.e. it represents destination.
- The . (dot) represents the current folder.

OUTPUT: Welcome to package

ACCESSING A PACKAGE:

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

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

1. Using import package.*;

Example:

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

//save by B.java
package mypack;
import pack.*;

class B
{
  public static void main(String args[])
{
   A obj = new A();
   obj.msg();
  }
}
```

```
Output: Hello
```

2. Using packagename.classname

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

Example of package by import package.classname

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

//save by B.java
package mypack;
import pack.A;

class B
{
  public static void main(String args[])
{
   A obj = new A();
   obj.msg();
  } }
```

```
Output: Hello
```

3. Using fully qualified name

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

Example of package by import fully qualified name

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

```
Output: Hello
```

MULTITHREADED PROGRAMMING:

- **Multithreading in java** is a process of executing multiple process simultaneously.
- Multithreading is a conceptual programming paradigm where a program is divided into two or more subprograms, which can be implemented at the same time in parallel.
- Thread is basically a lightweight sub-process, a smallest unit of processing.

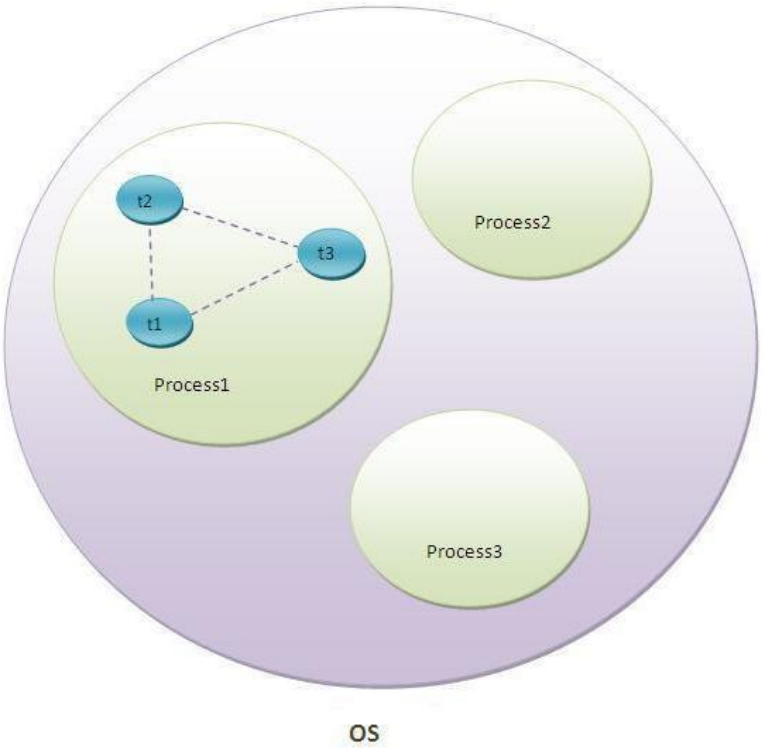
- Multiprocessing and multithreading, both are used to achieve multitasking.
- A ***Thread*** is similar to a program that has a single flow of control. It has a body, and an end, and executes commands sequentially.
- Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

- It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- You **can perform many operations together so it saves time**.
- Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

THREAD:

- A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.
- As shown in the below figure, thread is executed inside the process.



CREATING THREADS:

- Creating threads in java is simple
- Threads are implemented in the form of objects that contain a method called **run()**.
- The **run()** method is the heart and soul of any thread.

- It makes up the entire body of a thread and is the only method in which the thread's behaviour can be implemented.
- The run() should be invoked by an object of the concerned thread.
- The run() method can be initiating with the help of **start()** method.
- There are two ways to create a thread:

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

1. By Extending Thread class

Define a class that extends Thread class and override its run() method with the code required by the thread.

Example:

```
class Multi extends Thread                  // Extending thread class
{
public void run()                           // run() method declared
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();                       //object initiated
t1.start();                                 // run() method called through start()
 }
}
```

Output: thread is running…

2. By implementing Runnable interface

- Define a class that implements Runnable interface.
- The Runnable interface has only one method, run(), that is to be defined in the method with the code to be executed by the thread.

EXAMPLE:

```
class Multi3 implements Runnable            // Implementing Runnable interface
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();                      // object initiated for class
```

```
Thread t1 =new Thread(m1);                    // object initiated for thread
t1.start();
 } }
```

Output: thread is running…

STOPPING AND BLOCKING A THREAD:

Stopping a thread:

- Whenever we want to stop a thread from running further, we may do so by calling its ***stop()*** method.
- This causes a thread to stop immediately and move it to its *dead* state.
- It forces the thread to stop abruptly before its completion i.e.
- it causes premature death.
- To stop a thread we use the following syntax:

thread.stop();

Blocking a Thread:

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

sleep(t) // blocked for 't' milliseconds

suspend() // blocked until resume() method is invoked

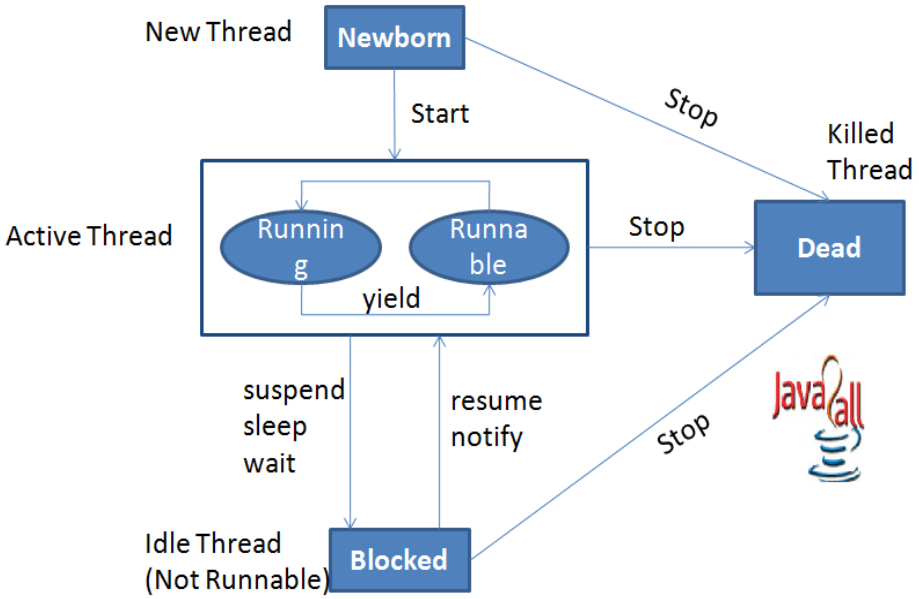
wait() // blocked until notify () is invoked

These methods cause the thread to go into the ***blocked*** **(or *not-runnable*)** state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep( )**, the **resume()** method is invoked in the case of **suspend( )**, and the **notify( )** method is called in the case of **wait()**.

LIFE CYCLE OF A THREAD:

During the life time of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state



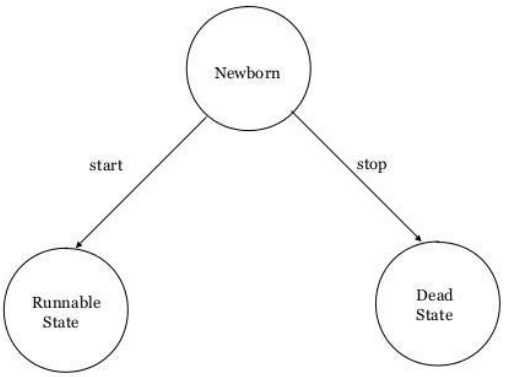
Newborn State:

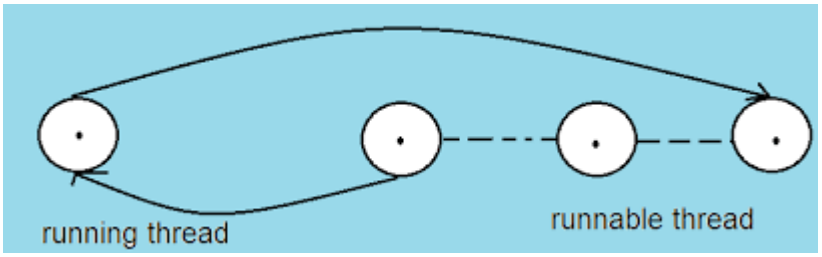
- When we create a thread object, the thread is born and is said to be in newborn state.
- The thread is not yet scheduled for running.
- At this state, we can do only one of the following things with it:
    - Schedule it for running using start() method.
    - Kill it using stop() method.

If scheduled, it moves to the runnable state If we attempt to use any other method at this stage, an exception will be thrown.

Runnable State*:*

- The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.
- That is, the thread has joined the queue of threads that are waiting for execution.
- If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e., first-come, first-serve manner.
- After its turn, the thread joins the queue again and waits for next turn
- This process of assigning time to threads is known as time-slicing.





Running State:

- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it gives up control on its own or taken over by other threads.
- When the thread is in its running state, we can ensure that the control is in run() method of the thread.

Blocked State:

- A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently to the running state.
- This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.
- A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.
- This state is achieved when we invoke suspend() or sleep() or wait() methods.

Dead State:

- Every thread has a life cycle.
- A running thread ends its life when it has completed executing its run( ) method.
- It is a natural death.
- However, we can kill it by sending the stop message to it at any state thus causing a premature death to it.
- A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.
- This state is achieved when we invoke stop() method or the thread completes it execution.

THREAD METHODS:

- Thread is a class found in java.lang package.
- It provides several different methods to perform thread tasks and control thread behaviors.

- The methods of Thread class with their meanings are listed below:

Method Signature	Description
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution
boolean isAlive()	This method will check if thread is alive or dead

THREAD PRIORITY:

- Each thread is assigned a priority, which affects the order in which it is scheduled for running.
- The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a first-come, first-serve basis.
- Java permits us to set the priority of a thread using the setPriority() method as follows:

ThreadName.setPriority(intNumber);

The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:

1. public static int MIN_PRIORITY = 1
2. public static int NORM_PRIORITY = 5
3. public static int MAX_PRIORITY = 10

The default setting is NORM_PRIORITY. Most user-level processes should use NORM_PRIORITY.

Example:

```
class TestMultiPriority1 extends Thread
{
 public void run()
{
  System.out.println("running thread name is:"+Thread.currentThread().getName());
```

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

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

JAVA SYNCHRONIZATION:

- Generally, threads use their own data and methods provided inside their run () methods.
- But if we wish to use data and methods outside the thread's run () method, they may compete for the same resources and may lead to serious problems.
- For example, one thread may try to read a record from a file while another is still writing to the same file.
- Depending on the situation, we may get strange results.
- Java enables us to overcome this problem using a technique known as synchronization.

In case of Java, the keyword **synchronized** helps to solve such problems by keeping a watch on such locations.

For example, the method that will read information from a file and the method that will update the same file may be declared as synchronized as shown below:

```
synchronized void update ()
{
…….. // code here is synchronized
}
```

When the method declared as synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time.

As long as the thread holds the monitor, no other thread can enter the synchronized section of code i.e. other threads cannot interrupt this thread with that object until its complete execution. It is like locking a function.

It is also possible to mark a block of code as synchronized as shown below:

synchronized (lock-object)

{

..... // code here is synchronized

}

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

DEADLOCK:

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.
- This undesirable situation may occur when two or more threads are waiting to gain control on a resource.
- Due to some reasons, the condition on which the waiting threads rely on to gain control does not happen. This results in a situation called as deadlock.
- But, java automatically recognizes this situation and terminates some processes automatically to ensure safer execution.

For example, assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets holds of Method2.