Chapter – 1
Introduction to Core Java

History of Java:

James Gosling

The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape. Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc.

James Gosling, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**. Originally designed for small, embedded systems in electronic appliances like set-top boxes.

Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt. After that, it was called **Oak** and was developed as a part of the Green project. Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.

In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies. The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.

Java is an island of Indonesia where first coffee was produced (called java coffee). Notice that Java is just a name not an acronym. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995. In 1995, Time magazine called Java one of the Ten Best Products of 1995. JDK 1.0 released in (January 23, 1996).

Java Version History:

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)

10. Java SE 8 (18th March, 2014)

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:

According to Sun, Java language is simple because syntax is based on C++ (so easier for programmers to learn it after C++). It 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.

2. Object-Oriented:

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

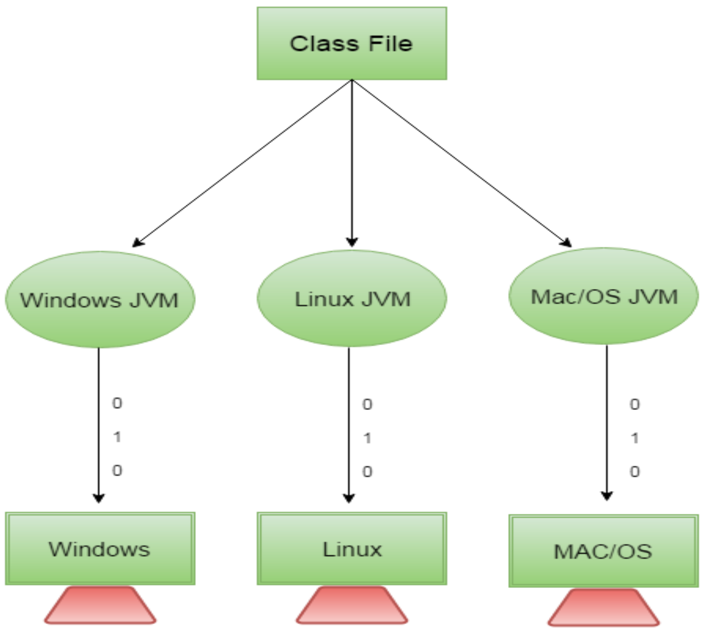
- ❖ Object
- ❖ Class
- ❖ Inheritance
- ❖ Polymorphism
- ❖ Abstraction
- ❖ Encapsulation

3. Platform Independent:

Java is 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:

a. Runtime Environment
b. API(Application Programming Interface)

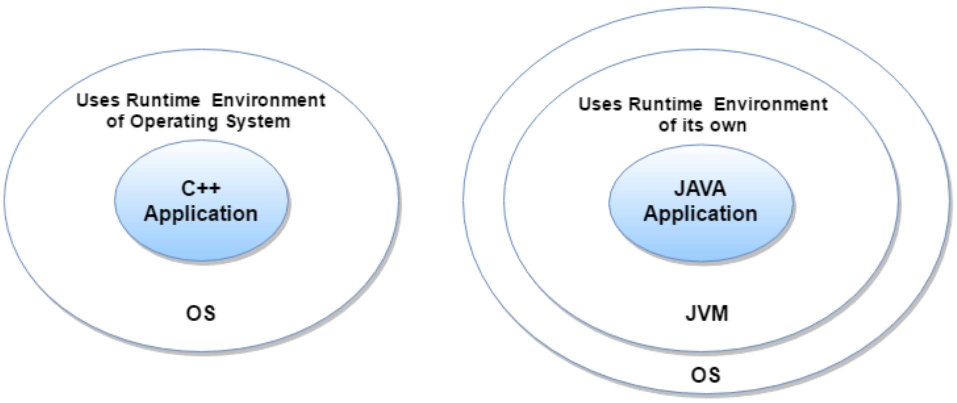
Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere (WORA).



4. Secured:

Java is secured because:

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


Uses Runtime Environment of Operating System
C++ Application
OS
Uses Runtime Environment of its own
JAVA Application
JVM
OS

- ❖ **Classloader:** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- ❖ **Bytecode 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.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, and Cryptography etc.

5. 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.

6. Architecture-Neutral:

There is no implementation dependent features e.g. size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

7. Portable:

We may carry the java bytecode to any platform.

8. 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++)

9. Distributed:

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

10. Multi-Threaded:

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

11. Dynamic:

Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

12. Interpreted:

Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

Introduction to JVM Architecture:

JVM stands for Java Virtual Machine and is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

It is a specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies. Its implementation is known as JRE (Java Runtime Environment). Whenever we write java command on the command prompt to run the java class, an instance of JVM is created.

The JVM performs following operation:

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

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

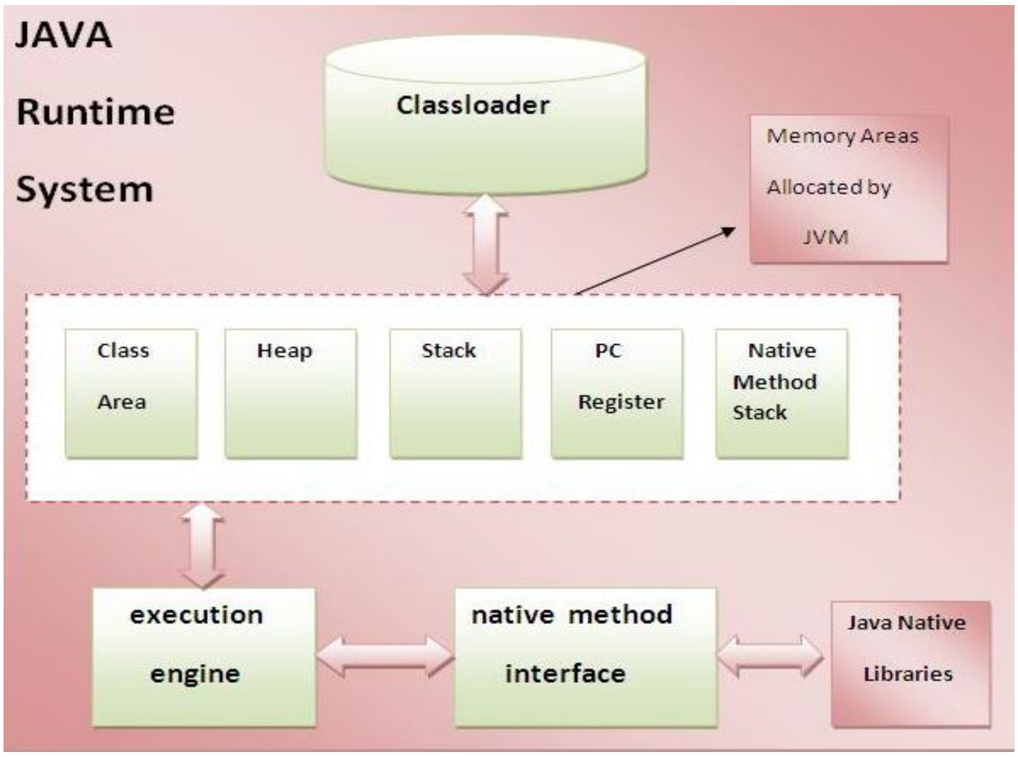


Fig: Internal Architecture of JVM

1. **Classloader:**

Classloader is a subsystem of JVM that is used to load class files.

2. **Class(Method) Area:**

Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3. **Heap:**

It is the runtime data area in which objects are allocated.

4. **Stack:**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5. **Program Counter Register:**

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6. **Native Method Stack:**

It contains all the native methods used in the application.

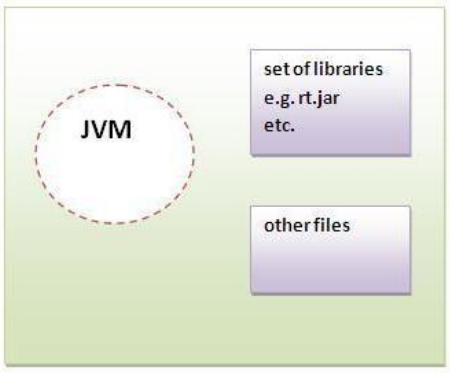
7. **Execution Engine:**

It contains:

a. A virtual processor
b. Interpreter: Read bytecode stream then execute the instructions.
c. Just-In-Time (JIT) compiler: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term Compiler refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

INTRODUCTION TO JRE:

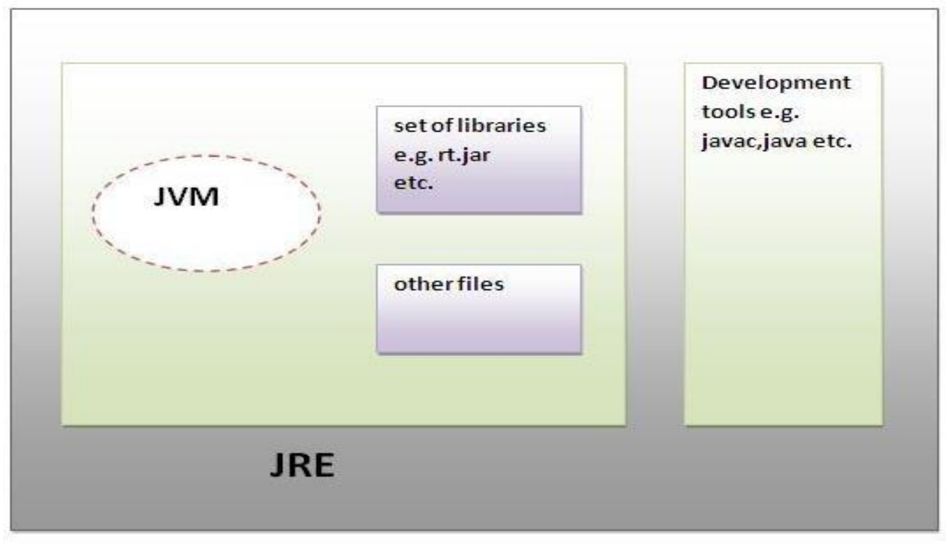
JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime. Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

INTRODUCTION TO JDK:

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.



Development tools e.g. javac,java etc.
set of libraries e.g. rt.jar etc.
JVM
other files
JRE

JDK

OBJECT ORIENTED FEATURES:

1. Encapsulation:

Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a "black box," or a device that you can use without regard to the internal mechanisms. A programmer can access and use the methods and data contained in the black box but cannot change them. Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments. Properties can be accessed using getXXX() methods which are having public access modifiers.

Example:

```
public class Mobile {
    private String manufacturer;
    private String operatingSystem;
    public String model;
    private int cost;
    //Constructor to set properties/characteristics of object
    Mobile(String man, String o,String m, int c){
            this.manufacturer = man;
            this.operatingSystem=o;
            this.model=m;
            this.cost=c;
    }
    //Method to get access Model property of Object
    public String getModel(){
            return this.model;
    }
    public String getManufacturer(){
            return this.manufacturer;
    }
    public String getOperatingSystem(){
            return this.operatingSystem;
    }
    public int getCost(){
            return this.cost;
    }
    public static void main(String [] args){
            Mobile obj = new Mobile("Samsung","Lolipop","J5",18000);
            System.out.println("Mobile Specification");
            System.out.println("Manufacturer: "+obj.getManufacturer());
            System.out.println("Operating System: "+obj.getOperatingSystem());
            System.out.println("Model: "+obj.getModel());
            System.out.println("Cost: "+obj.getCost());
    }
}
```

2. Inheritance:

An important feature of object-oriented programs is inheritance. Inheritance is the ability to create classes that share the attributes and methods of existing classes, but with more specific features. Inheritance is mainly used for code reusability. So we are making use of already written the classes and further extending on that. That why we discussed the code reusability the concept. In general one line definition, we can tell that deriving a new class from existing class, it's called as Inheritance. We can look into the following example for inheritance concept.

```
public class Mobile {
    private String manufacturer;
    private String operatingSystem;
    private String model;
    private int cost;
    public Mobile(String man, String o,String m, int c){
            manufacturer = man;
            operatingSystem=o;
            model=m;
            cost=c;
    }
    public String getModel(){
            return model;
    }
    public String getManufacturer(){
            return manufacturer;
    }
    public String getOperatingSystem(){
            return operatingSystem;
    }
    public int getCost(){
            return cost;
    }
}
public class Android extends Mobile{
    private int number;
    Android(int no){
            super("Samsung","Lolipop","J5",18000);
            this.number = no;
    }
    public int getNumber(){
            return this.number;
    }
    public static void main(String [] args){
            Android obj = new Android(15000);
            System.out.println("\nMobile Specification");
            System.out.println("Manufacturer: "+obj.getManufacturer());
            System.out.println("Operating System: "+obj.getOperatingSystem());
            System.out.println("Model: "+obj.getModel());
            System.out.println("Cost: "+obj.getCost());
            System.out.println("Sale Set: "+obj.getNumber());
    }
}
```

3. Polymorphism:

Polymorphism definition is that **Poly** means **many** and **morphos** means **forms**. It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. For example, in English, the verb "run" means different things if we use it with "a footrace," "a business," or "a computer." We understand the meaning of "run" based on the other words used with it. Object-oriented programs are written so that the methods having the same name works differently in different context. Java provides two ways to implement polymorphism.

a. Static Polymorphism (Compile Time Polymorphism/ Method Overloading):

The ability to execute different method implementations by altering the argument used with the method name is known as method overloading. In below program, we have three print methods each with different arguments. When we properly overload a method, we can call it providing different argument lists, and the appropriate version of the method executes.

```
class Overload{
    public void print(String s){
            System.out.println("First Method with only String: "+ s);
    }
    public void print (int i){
            System.out.println("Second Method with only int: "+ i);
    }
    public void print (String s, int i){
            System.out.println("Third Method with both String and Integer: "+ s + " and " + i);
    }
    public static void main(String[] args) {
            Overload obj = new Overload();
            obj.print(10);
            obj.print("Amit");
            obj.print("Hello", 100);
    }
}
```

b. <u>Dynamic Polymorphism (Run Time Polymorphism/ Method Overriding):</u>

When we create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, we want to override the parent class members. Let's take the example used in inheritance explanation.

```
class Parent{
            public void function(){
            System.out.println("This is parent class");
            }
    }
    class Child extends Parent{
            public void function(){
            System.out.println("This is child class");
            }
    }
    public class Help{
            public static void main(String [] args){
            Child obj = new Child();
            Parent obj1 = new Parent();
            obj.function();
            obj1.function();
            }
    }
```

4. Abstraction:

An essential element of object-oriented programming is an abstraction. Hiding internal details and showing functionality is known as abstraction. Humans manage complexity through abstraction. When we drive our car we do not have to be concerned with the exact internal working of our car (unless we are a mechanic). What we are concerned with is interacting with our car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers of car have different implementation of the car working but its basic interface has not changed (i.e. we still use the steering wheel, brake pedal, accelerator pedal etc. to interact with our car). Hence the knowledge we have of our car is abstract.

In java, we use abstract class and interface to achieve abstraction. An abstract class is something which is incomplete and we cannot create an instance of the abstract class. If we want to use it we need to make it complete or concrete by extending it. A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended. By the way, Java has a concept of abstract classes, abstract method but a variable cannot be abstract in Java.

```
abstract class VehicleAbstract {
    public abstract void start();
    public abstract void stop();
}
class TwoWheeler extends VehicleAbstract{
    public void start() {
            System.out.println("Starting Two Wheeler");
    }
    public void stop(){
            System.out.println("Stopping Two Wheeler");
    }
}
class FourWheeler extends VehicleAbstract{
    public void start() {
            System.out.println("Starting Four Wheeler");
    }
    public void stop(){
            System.out.println("Stopping Four Wheeler");
    }
}
public class VehicleTesting {
    public static void main(String[] args) {
            TwoWheeler my2Wheeler = new TwoWheeler();
            FourWheeler my4Wheeler = new FourWheeler();
            my2Wheeler.start();
            my2Wheeler.stop();
            my4Wheeler.start();
```

```
                my4Wheeler.stop();
        }
    }
```

CLASS AND OBJECT:

A class is a template, blueprint, or contract that defines what an object's data fields and methods will be. An object is an instance of a class. We can create many instances of a class. A Java class uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.
Objects are made up of attributes and methods. Attributes are the characteristics that define an object; the values contained in attributes differentiate objects of the same class from one another. To understand this better let's take the example of Mobile as an object. Mobile has characteristics like a model, manufacturer, cost, operating system etc. So if we create "Samsung" mobile object and "IPhone" mobile object we can distinguish them from characteristics. The values of the attributes of an object are also referred to as the object's state.

OPERATORS:

Operators are used to manipulate primitive data types. Java operators can be classified as unary, binary, or ternary that means they can take one, two, or three arguments, respectively. A unary operator may appear before (prefix) its argument or after (postfix) its argument. A binary or ternary operator appears between its arguments. Operators in java fall into 8 different categories and are explained one by one:

1. Arithmetic Operator:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Description
+ (Addition)	Adds values on either side of the operator.
- (Subtraction)	Subtracts right-hand operand from left-hand operand.
* (Multiplication)	Multiplies values on either side of the operator.
/ (Division)	Divides left-hand operand by right-hand operand.
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.
++ (Increment)	Increases the value of operand by 1.
-- (Decrement)	Decreases the value of operand by 1.

Example:

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

2. Relational Operators:

Relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. We just have to keep in mind that we must use "==", not "=", while testing if two primitive values are equal. The outcome of these operations is a Boolean value. The following table lists the relational operators:

Operator	Description
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Example:

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

3. Bitwise Operators:

Java provides Bitwise operators to manipulate the contents of variables at the bit level. These variables must be of numeric data type (byte, char, short, int, long). These operators act upon the individual bits of the operators. These operators are less commonly used. Java provides seven bitwise operators.

Operator	Description
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

Example:

```
public class Test {
  public static void main(String args[]) {
    int a = 60;  /* 60 = 0011 1100 */
    int b = 13;  /* 13 = 0000 1101 */
    int c = 0;
    c = a & b;       /* 12 = 0000 1100 */
    System.out.println("a & b = " + c );
    c = a | b;       /* 61 = 0011 1101 */
    System.out.println("a | b = " + c );
    c = a ^ b;       /* 49 = 0011 0001 */
    System.out.println("a ^ b = " + c );
    c = ~a;          /*-61 = 1100 0011 */
    System.out.println("~a = " + c );
    c = a << 2;      /* 240 = 1111 0000 */
    System.out.println("a << 2 = " + c );
    c = a >> 2;      /* 15 = 1111 */
```

```
        System.out.println("a >> 2  = " + c );
        c = a >>> 2;     /* 15 = 0000 1111 */
        System.out.println("a >>> 2 = " + c );
    }
}
```

4. Logical Operators:

Logical operators return a true or false value based on the state of the Variables. Each argument to a logical operator must be a Boolean data type, and the result is always a Boolean data type. Logical operators are known as Boolean operators or bitwise logical operators. The Boolean operator operates on Boolean values to create a new Boolean value. The following table lists the logical operators:

Operator	Description
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Example:

```
public class Test {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    System.out.println("a && b = " + (a&&b));
    System.out.println("a || b = " + (a||b) );
    System.out.println("!(a && b) = " + !(a && b));
  }
}
```

5. Assignment Operator:

The java assignment operator statement has the following syntax: <variable> = <expression>. The values generated by the right hand side expression will be assign to the variable in the left side, If the value already exists in the variable it is overwritten by the assignment operator (=). Following are the assignment operators supported by Java language:

Operator	Description
=	Simple assignment operator. Assigns values from right side operands to left side operand.
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.

-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.
***=**	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.
<<=	Left shift AND assignment operator.
>>=	Right shift AND assignment operator.
&=	Bitwise AND assignment operator.
^=	Bitwise exclusive OR and assignment operator.
|=	Bitwise inclusive OR and assignment operator.

Example:

```
public class Test {
  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 0;
    c = a + b;
    System.out.println("c = a + b = " + c );
    c += a ;
    System.out.println("c += a  = " + c );
    c -= a ;
    System.out.println("c -= a = " + c );
    c *= a ;
    System.out.println("c *= a = " + c );
    a = 10;
    c = 15;
    c /= a ;
    System.out.println("c /= a = " + c );
    a = 10;
    c = 15;
    c %= a ;
    System.out.println("c %= a  = " + c );
    c <<= 2 ;
    System.out.println("c <<= 2 = " + c );
    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );
    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );
    c &= a ;
```

```
      System.out.println("c &= a  = " + c );
      c ^= a ;
      System.out.println("c ^= a   = " + c );
      c |= a ;
      System.out.println("c |= a   = " + c );
   }
}
```

6. Conditional Operator (? :)

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator evaluates the first argument and, if true, evaluates the second argument. If the first argument evaluates to false, then the third argument is evaluated. The conditional operator is the expression equivalent of the if-else statement. The operator is written as:

variable x = (expression) ? value if true : value if false

Example:

```
public class Test {
   public static void main(String args[]) {
      int a, b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );
      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

7. Instanceof Operator:

The instanceof operator is use to test whether the object is an instance of the specified type (class or sub class or interface). instanceof operator is written as:

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example:

```
class Vehicle {}
public class Car extends Vehicle {
   public static void main(String args[]) {
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

DATA TYPES:

Variables are nothing but reserved memory locations to store values. This means that when we create a variable we reserve some space in the memory. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, we can store integers, decimals, or characters in these variables. There are two data types available in Java:

1. Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword.

a. 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

b. 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

c. 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

d. 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

e. 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

f. 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

g. 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

h. Char:

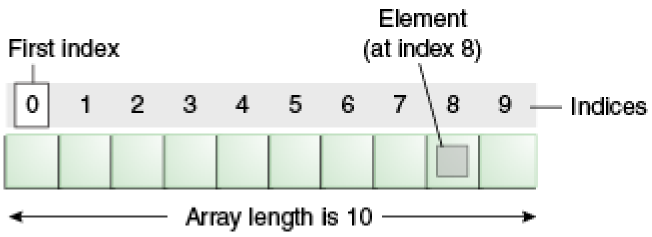
- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

2. REFERENCE DATA TYPES:

- 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");

JAVA ARRAY:

Normally, 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.



Advantage of Java Array

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

Disadvantage of Java Array

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

PROGRAMMING WITH ARRAYS

1. **Zero-Based Indexing:**

We always refer to the first element of an array a[] as a[0], the second as a[1], and so forth. It might seem more natural to us to refer to the first element as a[1], the second value as a[2], and so forth, but starting the indexing with 0 has some advantages and has emerged as the convention used in most modern programming languages.

2. **Array Length:**

Once we create an array, its length is fixed. We can refer to the length of an a[] in our program with the code a.length.

3. **Default Array Initialization:**

For economy in code, we often take advantage of Java's default array initialization convention. For example

```
double[] a = new double[n];
```

The default initial value is 0 for all numeric primitive types and false for type Boolean.

4. **Memory Representation:**

When we use new to create an array, Java reserves space in memory for it (and initializes the values). This process is called memory allocation.

5. **Bounds Checking:**

When programming with arrays, we must be careful. It is our responsibility to use legal indices when accessing an array element.

6. **Setting Array Values At Compile Time:**

When we have a small number of literal values that we want to keep in array, we can initialize it by listing the values between curly braces, separated by a comma. For example, we might use the following code in a program that processes playing cards.

```
String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
String[] RANKS = {"2", "3", "4", "5", "6", "7", "8", "9", "10","Jack", "Queen", "King", "Ace"};
```

After creating the two arrays, we might use them to print a random card name such as Queen of Clubs, as follows.

```
int i = (int) (Math.random() * RANKS.length);
int j = (int) (Math.random() * SUITS.length);
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

7. **Setting Array Values At Run Time:**

A more typical situation is when we wish to compute the values to be stored in an array For example, we might use the following code to initialize an array of length 52 that represents a deck of playing cards, using the arrays RANKS[] and SUITS[] just defined.

```
String[] deck = new String[RANKS.length * SUITS.length];
for (int i = 0; i < RANKS.length; i++)
   for (int j = 0; j < SUITS.length; j++)
      deck[SUITS.length*i + j] = RANKS[i] + " of " + SUITS[j];
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

Types of Array in java:

1. **One Dimensional Array:**

One Dimensional Array has only a single subscript or index. The one dimensional array is also known as vector. It stores data only row wise or column wise.

Syntax:

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

Example 1:

```
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]);
}}
```

Example 2:

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

2. **Multi-Dimensional Array:**

When we declare array more than one dimensional it is known as multi-dimensional array. It consists of two subscripts in which first given number is of row size and second given number is of column size.

Syntax:

```
datatype [][] arrayRefVar=new datatype[row][column];
```

Example 1:

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

PASSING ARRAY TO METHOD IN JAVA:

We can pass the java array to method so that we can reuse the same logic on any array.

Example:

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};
min(a);//passing array to method
}}
```

INHERITANCE IN JAVA:

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object. The idea behind inheritance in java is that we can create new classes that are built upon existing classes. When we inherit from an existing class, we can reuse methods and fields of parent class, and we can add new methods and fields also. Inheritance represents the **IS-A relationship**, also known as parent-child relationship.
The **extends keyword** indicates that we are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality. In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass. We perform inheritance for:

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

TYPES OF INHERITANCE IN JAVA:

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only.

1. **Single Inheritance:**

Inheritance is a property of OOP in which the characteristics of one class comes into derived class. In such case a derived class and only one base class are used to share some properties of one another classes.

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

2. **Multilevel Inheritance:**

Multilevel inheritances can be achieved when we place super class A serves as a base class for the derived class B which again behaves as the base class for the derived class C. The B is known as intermediate base class since it provides a link for the inheritance between A and B. The chain

ABC is known as inheritance path. This process can be extended to any number of levels depending upon the requirement.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

3. <u>**Hierarchical Inheritance:**</u>

In hierarchical inheritance there is single parents from which child or sibling are derived.

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

INTERFACE IN JAVA:

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. Java Interface also represents IS-A relationship. It cannot be instantiated just like abstract class.
Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

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.

An interface is different from a class in following ways:

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

Declaring Interfaces:

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

Example

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements
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. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
```

IMPLEMENTING INTERFACES:

When a class implements an interface, we can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {
  public void eat() {
    System.out.println("Mammal eats");
  }
  public void travel() {
    System.out.println("Mammal travels");
  }
  public int noOfLegs() {
    return 0;
  }
  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

When overriding methods defined in interfaces, there are several rules to be followed:

- ❖ Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- ❖ The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- ❖ An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules:

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

EXTENDING INTERFACES:

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

Example

```
// Filename: Sports.java
public interface Sports {
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}
// Filename: Football.java
public interface Football extends Sports {
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
// Filename: Hockey.java
public interface Hockey extends Sports {
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

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

EXTENDING MULTIPLE INTERFACES:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Example

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces:

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

Example

```
package java.util;
public interface EventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

- **Creates a common parent**:

As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

- **Adds a data type to a class**:

This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

Java Package:

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

Advantage of Java Package:

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

How to run java package program:

We need to use fully qualified name e.g. mypack.Simple etc to run the class.

- 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 (.) represents the current folder.

How to access package from another package?

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

1. **import package.*:**

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

Example

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

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

2. **import package.classname:**

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

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

3. **fully qualified name:**

If we use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But we need to use fully qualified name every time when we 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:

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

If we import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, we need to import the subpackage as well.

SUBPACKAGE IN JAVA:

Package inside the package is called the subpackage. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

Example:

```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

- To Compile: javac -d . Simple.java
- To Run: java com.javatpoint.core.Simple

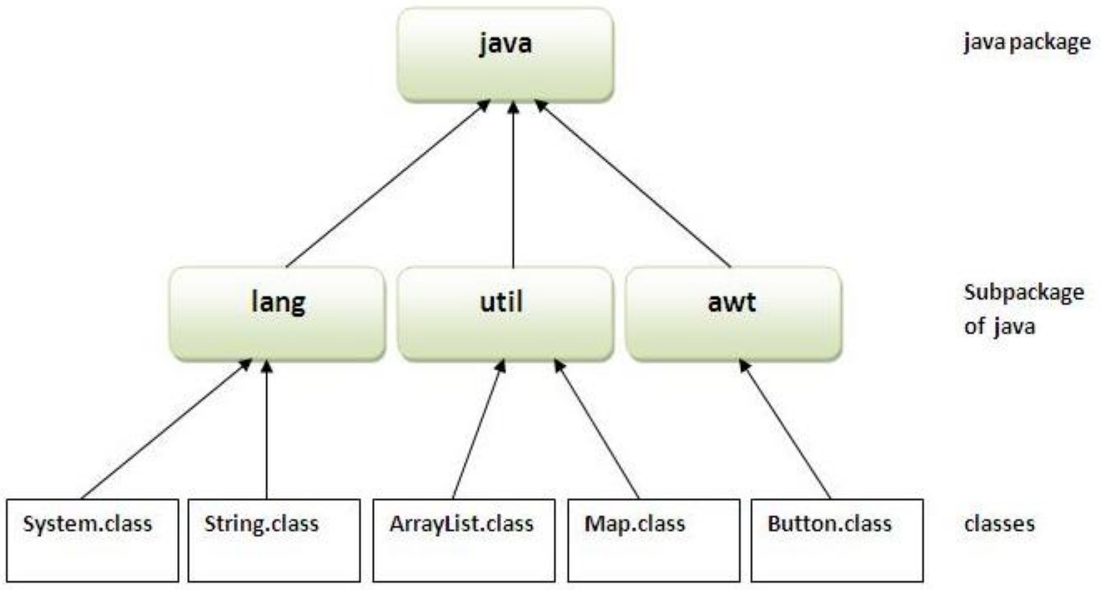


Fig: Java Package

EXCEPTION HANDLING:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

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

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. Based on these, we have three categories of Exceptions:

1. Checked exceptions:

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.
For example, if we use **FileReader** class in our program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

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

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

2. <u>Unchecked exceptions:</u>

An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
For example, if we have declared an array of size 5 in our program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.

Example

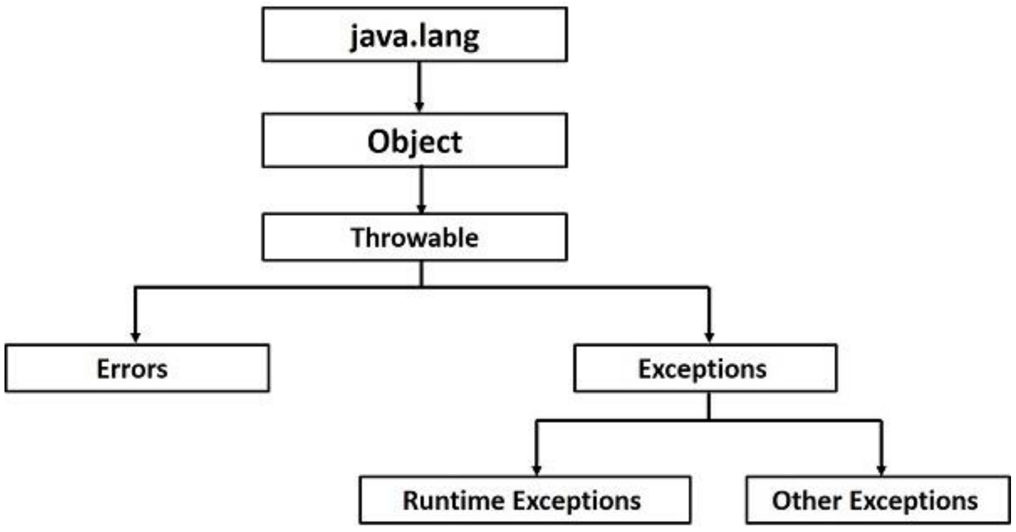
```
public class Unchecked_Demo {
  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
  }
}
```

3. <u>Errors:</u>

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

<u>Exception Hierarchy:</u>

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.
The Exception class has two main subclasses: IOException class and RuntimeException Class.



Exceptions Methods:

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

S.N.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

Syntax

```
try {
  // Protected code
}catch(ExceptionName e1) {
  // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.
A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example
The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest {
  public static void main(String args[]) {
    try {
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

MULTIPLE CATCH BLOCKS:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
}catch(ExceptionType3 e3) {
  // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try {
  file = new FileInputStream(fileName);
  x = (byte) file.read();
}catch(IOException i) {
  i.printStackTrace();
  return -1;
}catch(FileNotFoundException f) // Not valid! {
  f.printStackTrace();
  return -1;
}
```

THE THROWS/THROW KEYWORDS:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature. We can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException:

Example

```
import java.io.*;
public class className {
  public void deposit(double amount) throws RemoteException {
    // Method implementation
    throw new RemoteException();
  }
  // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException.

Example

```
import java.io.*;
public class className {
  public void withdraw(double amount) throws RemoteException,
    InsufficientFundsException {
```

```
    // Method implementation
  }
  // Remainder of class definition
}
```

THE FINALLY BLOCK:

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
}catch(ExceptionType3 e3) {
  // Catch block
}finally {
  // The finally block always executes.
}
```

Example

```
public class ExcepTest {
  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

Note:

- ❖ A catch clause cannot exist without a try statement.
- ❖ It is not compulsory to have finally clauses whenever a try/catch block is present.
- ❖ The try block cannot be present without either catch clause or finally clause.
- ❖ Any code cannot be present in between the try, catch, finally blocks.

THE TRY-WITH-RESOURCES:

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

Example

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class ReadData_Demo {
  public static void main(String args[]) {
    FileReader fr = null;
    try {
      File file = new File("file.txt");
      fr = new FileReader(file); char [] a = new char[50];
      fr.read(a);   // reads the content to the array
      for(char c : a)
      System.out.print(c);   // prints the characters one by one
    }catch(IOException e) {
      e.printStackTrace();
    }finally {
      try {
        fr.close();
      }catch(IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

try-with-resources, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.
To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

Syntax

```
try(FileReader fr = new FileReader("file path")) {
  // use the resource
  }catch() {
    // body of catch
  }
}
```

Following is the program that reads the data in a file using try-with-resources statement.
Example

```
import java.io.FileReader;
import java.io.IOException;
public class Try_withDemo {
  public static void main(String args[]) {
    try(FileReader fr = new FileReader("E://file.txt")) {
      char [] a = new char[50];
      fr.read(a);   // reads the contentto the array
      for(char c : a)
      System.out.print(c);   // prints the characters one by one
    }catch(IOException e) {
      e.printStackTrace();
    }
  }
}
```

Following points are to be kept in mind while working with try-with-resources statement.

- To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
- We can declare more than one class in try-with-resources statement.
- While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block.
- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

USER-DEFINED EXCEPTIONS:

We can create our own exceptions in Java. Keep the following points in mind when writing our own exception classes:

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

We can define our own Exception class as below:

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

We just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example

```
// File Name InsufficientFundsException.java
import java.io.*;
public class InsufficientFundsException extends Exception {
  private double amount;
  public InsufficientFundsException(double amount) {
    this.amount = amount;
  }
  public double getAmount() {
    return amount;
  }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;
public class CheckingAccount {
  private double balance;
  private int number;
  public CheckingAccount(int number) {
    this.number = number;
  }
  public void deposit(double amount) {
    balance += amount;
  }
  public void withdraw(double amount) throws InsufficientFundsException {
    if(amount <= balance) {
      balance -= amount;
    }else {
      double needs = amount - balance;
      throw new InsufficientFundsException(needs);
    }
  }
  public double getBalance() {
    return balance;
  }
  public int getNumber() {
    return number;
  }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo {

  public static void main(String [] args) {
    CheckingAccount c = new CheckingAccount(101);
    System.out.println("Depositing $500...");
    c.deposit(500.00);
    try {
      System.out.println("\nWithdrawing $100...");
      c.withdraw(100.00);
      System.out.println("\nWithdrawing $600...");
      c.withdraw(600.00);
    }catch(InsufficientFundsException e) {
      System.out.println("Sorry, but you are short $" + e.getAmount());
      e.printStackTrace();
    }
  }
}
```

Common Exceptions:

In Java, it is possible to define two categories of Exceptions and Errors.

- **JVM Exceptions**:

These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.

- **Programmatic Exceptions**:

These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.