Java Programming

Introduction to Java

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let application developers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities (such as reflection and runtime code modification) that are typically not available in traditional compiled languages. As of 2019, Java was one of the most popular programming languages in use according to GitHub, particularly for client-server web applications, with a reported 9 million developers.

Java is a programming language developed by James Gosling and others at Sun Microsystems. It's expressly designed for use in the distributed environment of the internet. It was designed to have the "look and feel" of C++ language, but it is simpler to use that C++.

History Of Java

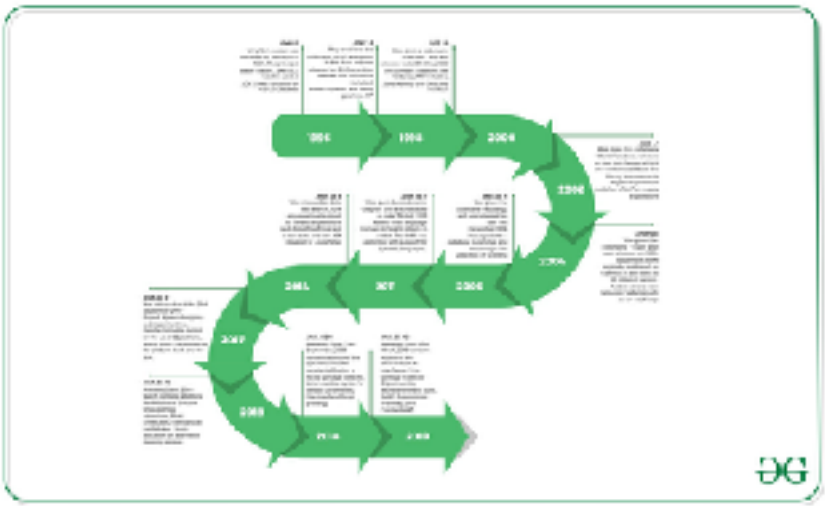
The history of java starts from Green Team. Java team members (also known as Green Team), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc. For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape. Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

2) Originally designed for small, embedded systems in electronic appliances like set- top boxes.

3) Firstly, it was called "Greentalk" by James Gosling and file extension was .gt.

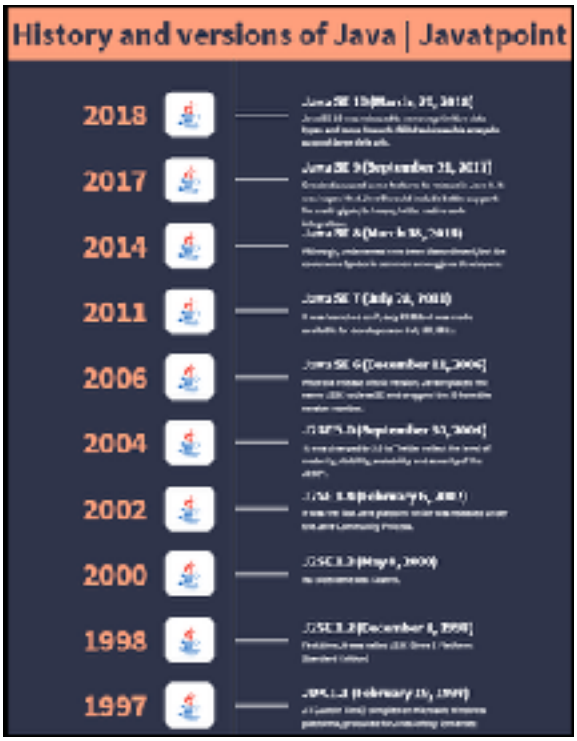
4) After that, it was called Oak and was developed as a part of the Green project.



Complete History of Java

Versions Of Java

The Java language has undergone several changes since JDK 1.0 as well as numerous additions of classes and packages to the standard library. Since J2SE 1.4, the evolution of the Java language has been governed by the Java Community Process (JCP), which uses Java Specification Requests (JSRs) to propose and specify additions and changes to the Java platform. The language is specified by the Java Language Specification (JLS); changes to the JLS are managed under JSR 901. In addition to the language changes, other changes have been made to the Java Class Library over the years, which has grown from a few hundred classes in JDK 1.0 to over three thousand in J2SE 5. Entire new APIs, such as Swing and Java2D, have been introduced, and many of the original JDK 1.0 classes and methods have been deprecated. Some programs allow conversion of Java programs from one version of the Java platform to an older one (Regarding Oracle Java SE Support Roadmap, version 11 is the currently supported long-term support (LTS) version, together with Java 8 LTS, where Oracle Customers will receive Oracle Premier Support. Java 8 LTS last free software public update for commercial use was released by Oracle in January 2019, while Oracle continues to release no-cost public Java 8 updates for development and personal use indefinitely. Java 10 a previously supported rapid release version, had its support ended in September 2018 the same date support for Java 11 began. Java 7 is no longer publicly supported. For Java 11, long-term support will not be provided by Oracle for the public; instead, the broader OpenJDK community, as AdoptOpenJDK or others, is expected to perform the work. Java 16 General Availability occurred on March 16, 2021, with Java 17 now also in development



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

Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

Example: //This is single line comment

```
public class CommentExample1 {
public static void main(String[] args) {
int i=10; //Here, i is a variable
System.out.println(i);
} }
```

Output:

```
10
```

Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

Example: /* This is multi line comment */

```
public class CommentExample2 {

public static void main(String[] args) { /* Let's declare and print variable in java. */

int i=10;

System.out.println(i);

}}
```

Output:

```
10
```

Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

Example: /** This is documentation comment. */

```
/** The Calculator class provides methods to get addition and
subtraction of given 2 numbers.*/ public class Calculator {
```

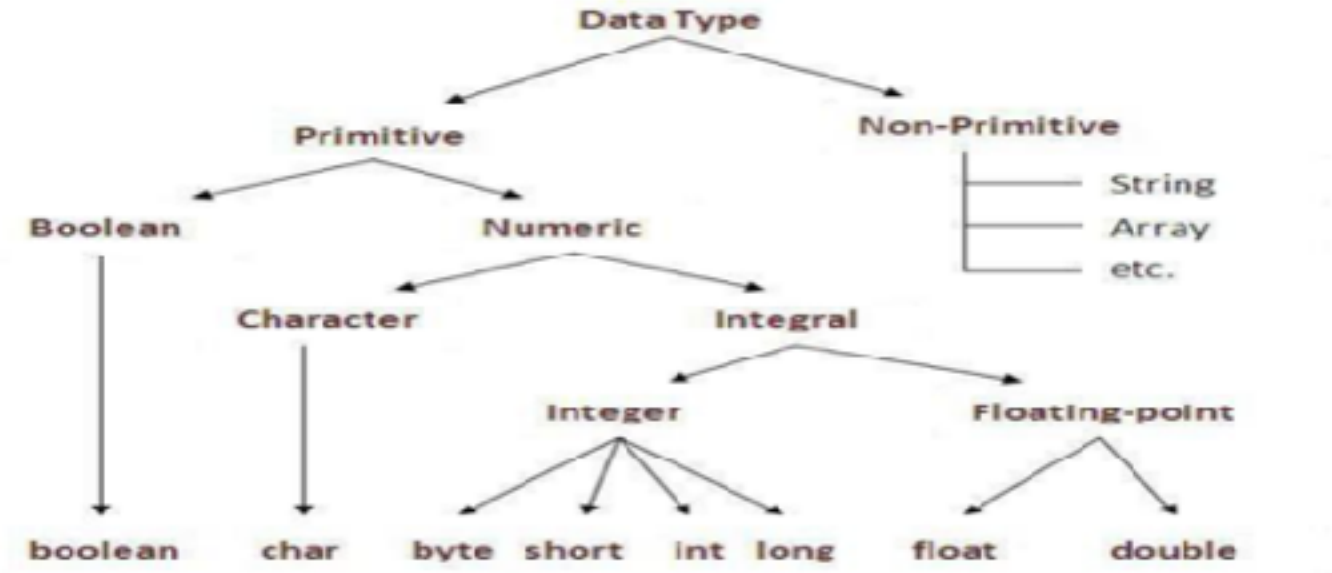
```
/** The add() method returns addition of given numbers.*/ public static
int add(int a, int b){return a+b;}

/** The sub() method returns subtraction of given numbers.*/ public
static int sub(int a, int b){return a-b;} }
```

Variables And Data Types In Java

Data types represent the different values to be stored in the variable. In java, there are two types of data types:

- Primitive data types
- Non-primitive data types



Data Type	Default Value	Default size
boolean	False	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Java Variable Example: Add Two Numbers

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

Output:20

Variables and Data Types in Java

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in java: primitive and non-primitive.

Types of Variable

There are three types of variables in java:

- local variable
- instance variable
- static variable

1)Local Variable

A variable which is declared inside the method is called local variable.

2) Instance Variable

A variable which is declared inside the class but outside the method, is called instance variable . It is not declared as static.

3) Static variable

A variable that is declared as static is called static variable. It cannot be local.

Example to understand the types of variables in java

```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Constants in Java

A constant is a variable which cannot have its value changed after declaration. It uses the **'final' keyword.**

Syntax

```
modifier final dataType variableName = value; //global constant

modifier static final dataType variableName = value; //constant within a c
```

Scope and Life Time of Variables

The scope of a variable defines the section of the code in which the variable is visible. As a general rule, variables that are defined within a block are not accessible outside that block. The lifetime of a variable refers to how long the variable exists before it is destroyed. Destroying variables refers to deallocating the memory that was allotted to the variables when declaring it. We have written a few classes till now. You might have observed that not all variables are the same. The ones declared in the body of a method were different from those that were declared in the class itself. There are three types of variables: instance variables, formal parameters or local variables and local variables.

Instance variables

Instance variables are those that are defined within a class itself and not in any method or constructor of the class. They are known as instance variables because every instance of the class (object) contains a copy of these variables. The scope of instance variables is determined by the access specifier that is applied to these variables. We have already seen about it earlier. The lifetime of these variables is the same as the lifetime of the object to which it belongs. Object once created do not exist for ever. They are destroyed by the garbage collector of Java when

there are no more reference to that object. We shall see about Java's automatic garbage collector later on.

Argument variables

These are the variables that are defined in the header oaf constructor or a method. The scope of these variables is the method or constructor in which they are defined. The lifetime is limited to the time for which the method keeps executing. Once the method finishes execution, these variables are destroyed.

Local variables

A local variable is the one that is declared within a method or a constructor (not in the header). The scope and lifetime are limited to the method itself.

One important distinction between these three types of variables is that access specifiers can be applied to instance variables only and not to argument or local variables.

In addition to the local variables defined in a method, we also have variables that are defined in bocks life an if block and an else block. The scope and is the same as that of the block itself.

Operators In Java

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:

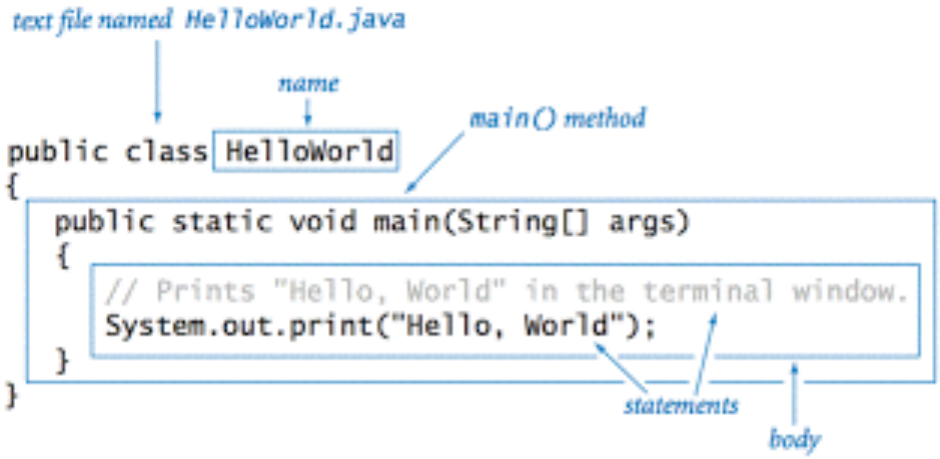
- Unary Operator,
- Arithmetic Operator,
- shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Operators Hierarchy

Operator Precedence

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Syntax



Strings In Java

Strings are used for storing text. A string variable contains a collection of characters surrounded by double quotes:
Create a variable of type string and assign it a value:
Syntax:
String greeting = “Hello";

String Length
A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length() method:

Example:
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());

String Cases:
There are many string methods available, for example toUpperCase() and toLowerCase():

Example:
```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world”
```

Finding a Character in a String
The indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace):

Example:
```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

String Concatenation
The + operator can be used between strings to combine them. This is called concatenation:

Example:
```
String firstName = "John";
String lastName = "Doe";
System.out.println(firstName + " " + lastName);
```

You can also use the **concat()** method to concatenate two strings:

Example:
```
String firstName = "John ";
String lastName = "Doe";
System.out.println(firstName.concat(lastName));
```

Adding Numbers and Strings
If you add two numbers, the result will be a number:

Example:
```
int x = 10;
int y = 20;
int z = x + y;      // z will be 30 (an integer/number)
```

Booleans In Java

In Java, the boolean keyword is a primitive data type. It is used to store only two possible values, either true or false. It specifies 1-bit of information and its "size" can't be defined precisely.
The boolean keyword is used with variables and methods. Its default value is false. It is generally associated with conditional statements.

Simple boolean example

```
public class BooleanExample1 {
  public static void main(String[] args) {
        int num1=10;
        int num2=20;
    boolean b1=true;
    boolean b2=false;
if(num1<num2)
{
    System.out.println(b1);
}
else
{
    System.out.println(b2);
}

    }       }
```

Comparing the variables of boolean type

```
public class BooleanExample2 {
    public static void main(String[] args) {
    boolean b1=true;
    boolean b2=false;
    boolean b3=(b1==b2);
    System.out.println(b1);
    System.out.println(b2);
    System.out.println(b3);

    }
}
```

Finding a prime number

```
public class BooleanExample5 {
  public static void main(String[] args) {
        int num=7;
        boolean flag=false;
           for(int i=2;i<num;i++)
         {
            if(num%i==0)
            {
                flag=true;
                      break;
            } }
        if(flag)
        {
           System.out.println("Not prime");
        }
        else
        {
           System.out.println("prime");
        } }  }
```

Arrays In Java

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar; // preferred way.
or
dataType arrayRefVar[]; // works but not preferred way.
```

Note:The styledataType[] arrayRefVar is preferred. The style dataType arrayRefVar[] comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example:
The following code snippets are examples of this syntax:

```
double[] myList; // preferred way.
or
double myList[]; // works but not preferred way.
```

Creating Arrays:

You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using new dataType[arraySize];
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

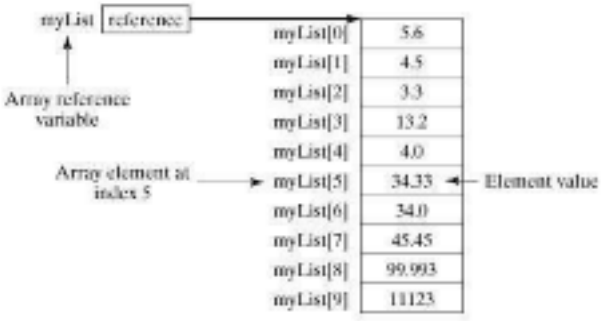
The array elements are accessed through the index. Array indices are 0-based; that is, they start from 0 to arrayRefVar.length-1.

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



- To declare an array, define the variable type with square brackets:
  String[] cars;
- We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:
  String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
- To create an array of integers, you could write:
  int[] myNum = {10, 20, 30, 40};

Processing Arrays:

When processing array elements, we often use either for loop or for each loop because all of the elements in an array are of the same type and the size of the array is known.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray
{
  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};
          //Print all the
    array elements
    for (int i = 0; i <
    myList.length; i+
    +) {
      System.out.println(myList[i] + " ");
    }
    //Summing all elements
```

```
        double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);
    //Finding
    the largest
    element
    double
    max =
    myList[0];
    for (int i = 1; i <
      myList.length; i+
      +) { if (myList[i] >
      max) max =
      myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```

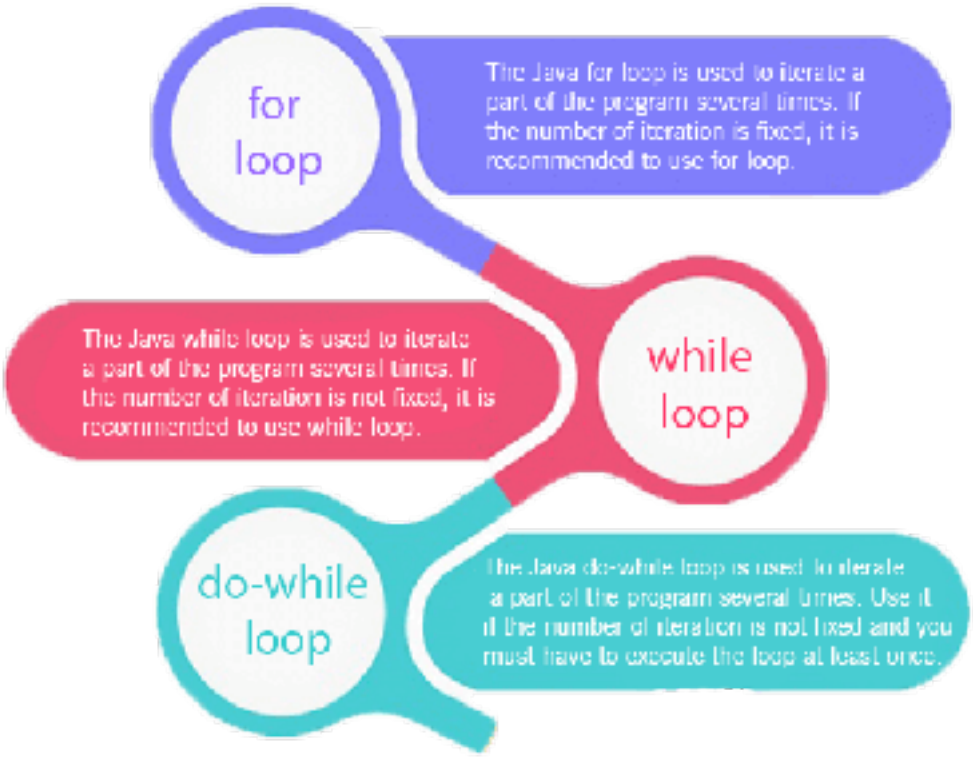
This would produce the following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Loops In Java

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

- for loop
- while loop
- do-while loop



Java For Loop vs While Loop vs Do While Loop

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	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.

Syntax	for(init;condition;incr/decr){
	// code to be executed
	}
		while(condition){
		//code to be executed
		}
			do{
			//code to be executed
			}while(condition);

Example	//for loop
	for(int i=1;i<=10;i++){
	System.out.println(i);
	}
		//While loop
		int i=1;
		while(i<=10){
		System.out.println(i);
		i++;
		}
			//do while loop
			int i=1;
			do{
			System.out.println(i);
			i++;
			}while(i<=10);

Syntax for infinitive loop	for(;;){
	//code to be executed
	}
		while(true){
		//code to be executed
		}
			do{
			//code to be executed
			}while(true);

Java For Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop. There are three types of for loops in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

Java Simple For Loop

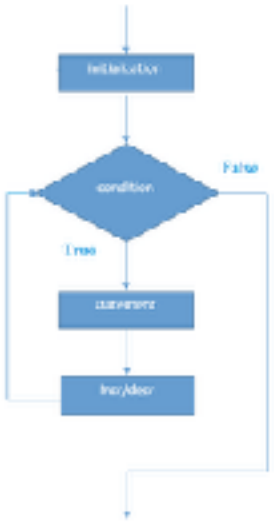
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

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

Syntax:

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

Flowchart



Java Infinitive For Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

Syntax:

```
for(;;){
//code to be executed
}
```

Example:

```
//Java program to demonstrate the use of infinite for loop
//which prints an statement
public class ForExample {
public static void main(String[] args) {
    //Using no condition in for loop
    for(;;){
        System.out.println("infinitive loop");
    } } }
```

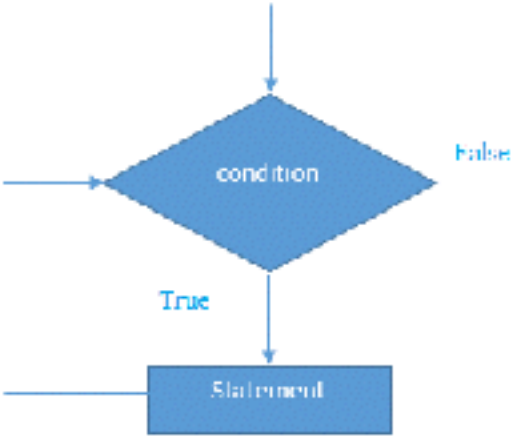
Java While Loop

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
}
```

FLOWCHART



Java Infinitive While Loop

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

Syntax:

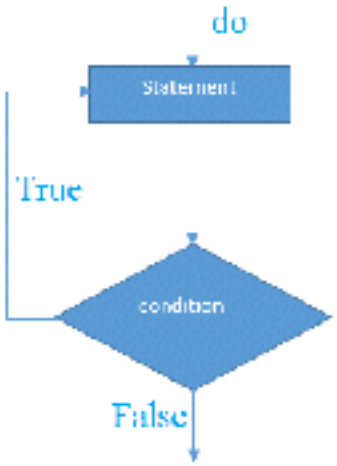
```
while(true){
//code to be executed
}
```

Example:

```
public class WhileExample2 {
public static void main(String[] args) {
    while(true){
        System.out.println("infinitive while loop");
    }
}
}
```

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<=10);
}
}
```

Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
The Java break statement 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.
We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.
Syntax:

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

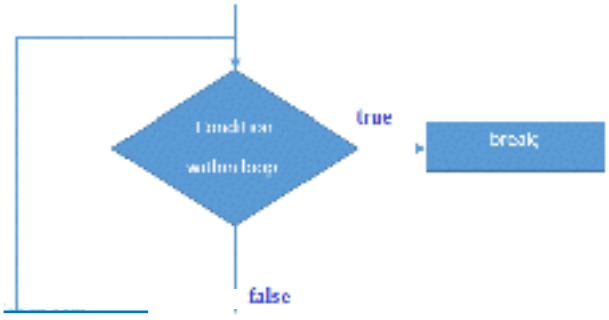


Figure: Flowchart of break statement

Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.
The Java continue statement is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.
We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.
Syntax:

jump-statement;
continue;
Java Continue Statement Example

Example:

```
//
Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
    //for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //using continue statement
            continue;//it will skip the rest statement
        }
        System.out.println(i);
    }
}
}
```

Java Object Class

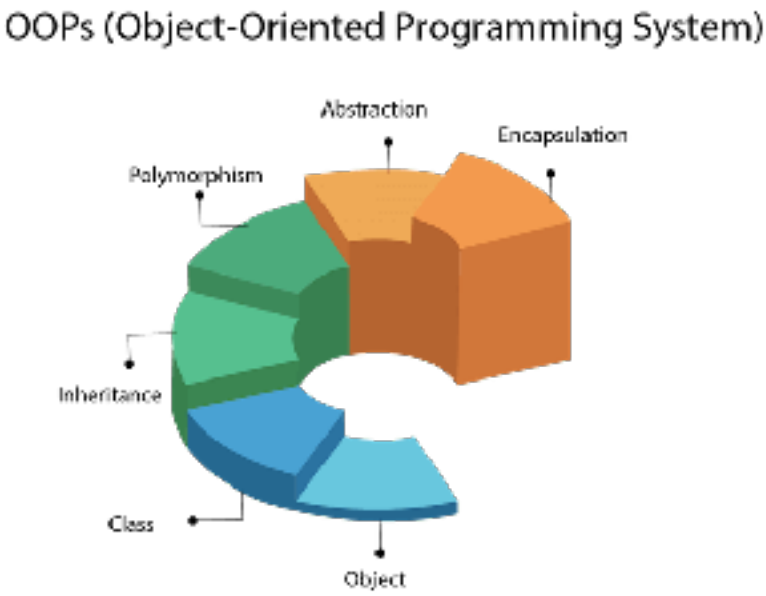
Java OOPs Concept

OOPs (Object-Oriented Programming) is a paradigm that provides many concepts, such as inheritance, data binding, polymorphism, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language. The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.



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

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

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

Object

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

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects. Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism. Another example can be to speak something; for example, a cat speaks meow, 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 are known as encapsulation. For example, a 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.

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a has-a relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

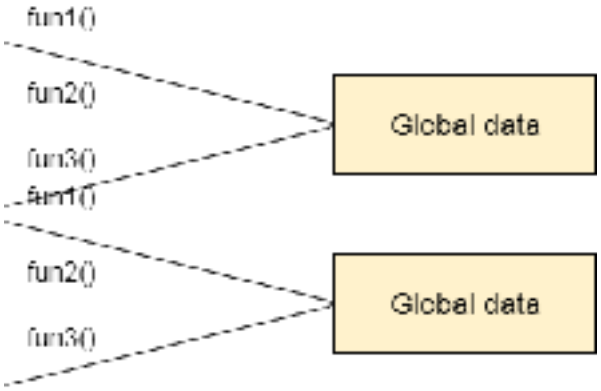


Figure: Data Representation in Procedure-Oriented Programming

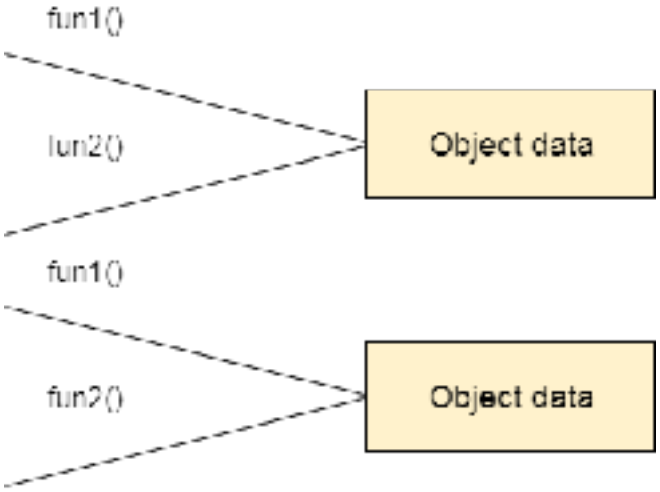


Figure: Data Representation in Object-Oriented Programming

Java Naming conventions

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

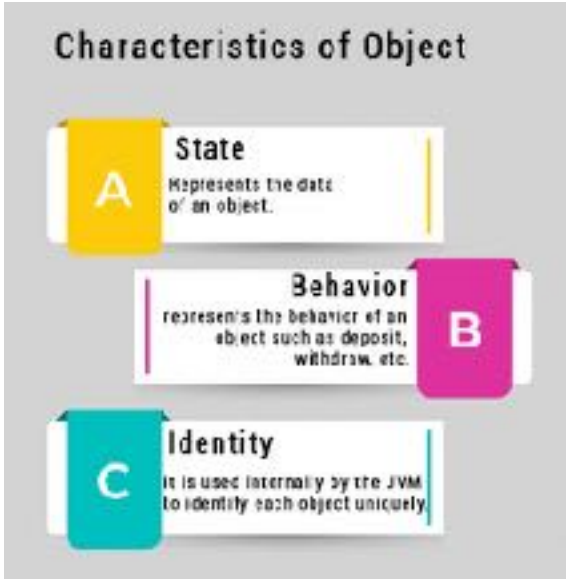
- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), $ (dollar), _ (underscore).

Objects and Classes in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:
- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

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

3 Ways to initialize object

There are 3 ways to initialize object in Java.
- By reference variable
- By method
- By constructor

1) **Object and Class Example:** Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
```

```
        Student s1=new Student();
        s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//
printing members with a white space
 }
}
```

2) **Object and Class Example:** Initialization through method
In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

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

3) **Object and Class Example:** Initialization through a constructor

Constructors In Java

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.
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-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

<class_name>(){}

Example of default constructor

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

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

Output: Bike is created

Example of parameterized constructor

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

```
class Student4{
 int id;
 String name;

 Student4(int i,String n){
 id = i;
 name = n;
 }
 void display(){System.out.println(id+" "+name);}

 public static void main(String args[]){
 Student4 s1 = new Student4(111,"Karan");
 Student4 s2 = new Student4(222,"Aryan");
 s1.display();
 s2.display();
} }
```

Output:

```
111.Karan
222.Aryan
```

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
class Student5{
int id;
String name;
int age;
Student5(int i,String n){
id = i;
name = n;
}
Student5(int i,String n,int a){
id = i;
name = n;
age=a;
}
void display(){System.out.println(id+" "+name+" "+age);}

public static void main(String args[]){
Student5 s1 = new Student5(111,"Karan");
Student5 s2 = new Student5(222,"Aryan",25);
s1.display();

s2.display();
} }
```

Output:

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

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method Object class

Example

In this example, we are going to copy the values of one object into another using java constructor.

```
class Student6{
int id;
String name;
Student6(int i,String n){
id = i;
name = n;
}

Student6(Student6 s){
id = s.id;
name =s.name;
}
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student6 s1 = new Student6(111,"Karan");
Student6 s2 = new Student6(s1);
s1.display();
s2.display();
} }
```

Output:

```
111.Karan
111.Karan
```

Java - Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.**println()** method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

Creating Method

Considering the following example to explain the syntax of a method −

Syntax

```
public static int methodName(int a, int b) {
  // body
}
```

Here,

- public static − modifier
- int − return type
- methodName − name of the method
- a, b − formal parameters
- int a, int b − list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax –

Syntax

```
modifier returnType
  nameOfMethod (Parameter
  List) { // method body
}
```

The syntax shown above includes −

- modifier − It defines the access type of the method and it is optional to use.
- returnType − Method may return a value.
- nameOfMethod − This is the method name. The method signature consists of the method name and the parameter list.
- Parameter List − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- method body − The method body defines what the method does with the statements.

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{
 int data=50;
 void change(int data){
 data=data+100;//changes will be in the local variable only
 }
 public static void main(String args[]){
  Operation op=new Operation();
  System.out.println("before change "+op.data);
  op.change(500);
  System.out.println("after change "+op.data);
 }
}
```

Output:before change 50
after change 50

In Java, parameters are always passed by value. For example, following program prints i = 10, j = 20.

```
 //Test.java class Test {
  // swap() doesn't swap i and j
  public static void swap(Integer i, Integer j) {
    Integer temp = new Integer(i);
```

```
        i = j;
        j = temp;
  }
  public static void main(String[] args) {
    Integer i = new Integer(10);
    Integer j = new Integer(20);
    swap(i, j);
    System.out.println("i = " + i + ", j = " + j);

}
```

Static Fields and Methods

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique

for each object) e.g. company name of employees,college name of students etc.

- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program memory efficient (i.e it saves memory).

Understanding problem without static variable

```
class Student{
int rollno;
String name;
String college="ITS";
}
```

Example of static variable

```
//Program of static variable
class Student8{
  int rollno;
  String name;
  static String college ="ITS";
  Student8(int r,String n){
  rollno = r;
  name = n;
  }
 void display (){System.out.println(rollno+" "+name+"
"+college);}
 public static void main(String args[]){
```

```
        Student8 s1 = new Student8(111,"Karan");
Student8 s2 = new Student8(222,"Aryan");


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

Output:

```
111
Karan
ITS
222
Aryan
ITS
```

Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```
//Program of changing the common property of all objects(static field).

class Student9{
   int rollno;
   String name;
   static String college = "ITS";
```

```
    static void change(){
    college = "BBDIT";


}
Student9(int r, String n){
rollno = r;
name = n;
}
 void display ()
{System.out.println(rollno+"
"+name+" "+college);} public static
void main(String args[])
{ Student9.change();
Student9 s1 = new Student9 (111,"Karan");
Student9 s2 = new Student9 (222,"Aryan");
Student9 s3 = new Student9 (333,"Sonoo");
s1.display();
s2.display();
s3.display();
} }
```

Output:111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT

Java static block

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

Example of static block

```
class A2{
 static{System.out.println("
 static block is invoked");}
 public static void
 main(String args[])
 { System.out.println("Hell
 o main"); } }
```

Output: static block is invoked
Hello main

Access Control

Access Modifiers in java

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

The access modifiers in java specifies accessibility
(scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers

1. private
2. default
3. protected
4. public

private access modifier

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
private int data=40;
private void msg()
{System.out.println("Hello
java");}
 }
public class Simple{
 public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);//Compile Time Error
  obj.msg();//Compile Time Error
  }
  }
```

Default access modifier

If you don't use any modifier, it is treated as default bydefault. The default modifier is accessible only within package.

Example of default access modifier

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

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

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

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

protected access modifier

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

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

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

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

Output:Hello

public access modifier

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

Example of public access modifier

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

Output:Hello

Understanding all java access modifiers

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

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

this keyword in java

Usage of java this keyword

Here is given the 6 usage of java this keyword.

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

Example:

```
class Student{ int rollno; String name; float fee;
Student(int rollno,String name,float fee){ this.rollno=rollno;
this.name=name; this.fee=fee;
}
void display(){
System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f); Student s2=new
                Student(112,"sumit",6000f); s1.display();
s2.display();
}}
```

Output:
111 ankit 5000
112 sumit 6000

Difference between constructor and method in java

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
class Student5{
int id; String
name; int
age;
Student5(int i,String n){
id = i;
name = n;
}
Student5(int i,String n,int a){
id = i;
name = n;
age=a;
}
void display(){System.out.println(id+" "+name+" "+age);}

public static void main(String args[]){
Student5 s1 = new Student5(111,"Karan");
Student5 s2 = new Student5(222,"Aryan",25);
s1.display();
s2.display();
}
}
```

Output:

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

Method Overloading in java

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

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

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

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

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

Output:

22

33

Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

Java Recursion Example 1: Factorial Number

```
public class RecursionExample3 {
static int factorial(int n){
if (n == 1)
return 1;
else
return(n * factorial(n-1));
} }
public static void main(String[] args)
{ System.out.println("Factorial of 5 is: "+factorial(5)); } }
```

Output:

Factorial of 5 is: 120

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

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

gc() method

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

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

Simple Example of garbage collection in java

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

Inheritance In Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system). The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a *parent-child* relationship.

Why use inheritance in java????

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

Terms used in Inheritance

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

- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

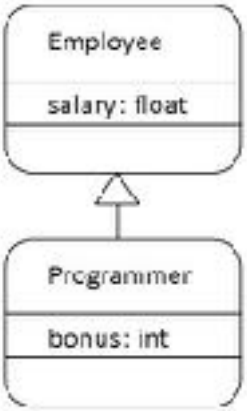
The syntax of Java Inheritance

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

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.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



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

```
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 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.

Single Inheritance

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

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

Output:

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

Multilevel Inheritance

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

```
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

When two or more classes inherits a single class, it is known as hierarchical inheritance.
In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

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

Output:

```
meowing…
eating...
```

Encapsulation in Java

Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines. We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
The Java Bean class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class read-only or write-only. In other words, you can skip the getter or setter methods. It provides you the control over the data. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods. It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.The encapsulate class is easy to test. So, it is better for unit testing. The standard IDE's are providing the facility to generate the getters and setters. So, it is easy and fast to create an encapsulated class in Java.

Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

```
//A Java class which is a fully encapsulated class.
//
It has a private data member and getter and setter methods.

package com.javatpoint;
public class Student{
//private data member
private String name;
//getter method for name
public String getName(){
return name;
}
//setter method for name
public void setName(String name){
this.name=name
}
}

//A Java class to test the encapsulated class.
package com.javatpoint;
class Test{
public static void main(String[] args){
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```

Output:
vijay

Polymorphism in Java

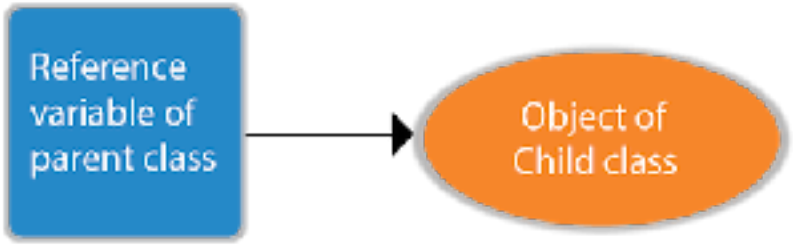
Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java. Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
class A{}
class B extends A{}

A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Here, the relationship of B class would be:

B IS-A A
B IS-A I
B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime. Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run()
{System.out.println("running safely with 60km");}
  public static void main(String args[]){
```

```
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output:
running safely with 60km.

Java Runtime Polymorphism Example: Shape

```
class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
class TestPolymorphism2{
public static void main(String args[]){
Shape s;
s=new Rectangle();
s.draw();
s=new Circle();
s.draw();
s=new Triangle();
s.draw();
}
}
```

Output:
drawing rectangle...
drawing circle...
drawing triangle...

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.
Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java
- Abstract class (0 to 100%)
- Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

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

Abstract Method in Java

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

Example of abstract method

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

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

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

output:

running safely

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the factory method.

A factory method is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

```
abstract class Shape{
abstract void draw();
}
//
In real scenario, implementation is provided by others i.e. un
known by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//
In a real scenario, object is provided through method, e.g., g
etShape() method
s.draw();
}
}
```

output

drawing circle

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.
In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
Java Interface also represents the IS-A relationship.
It cannot be instantiated just like the abstract class.

Since Java 8, we can have default and static methods in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable{
void print();
```

```
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
output:
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
```

```
Drawable d=new Circle();//
In real scenario, object is provided by method e.g. getDrawa
ble()
d.draw();
}}
```

Output:

```
drawing circle
```

Exception Handling In Java

The exception handling in java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

What is exception

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. Exception normally disrupts the normal flow of the application that is why we use exception handling.

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

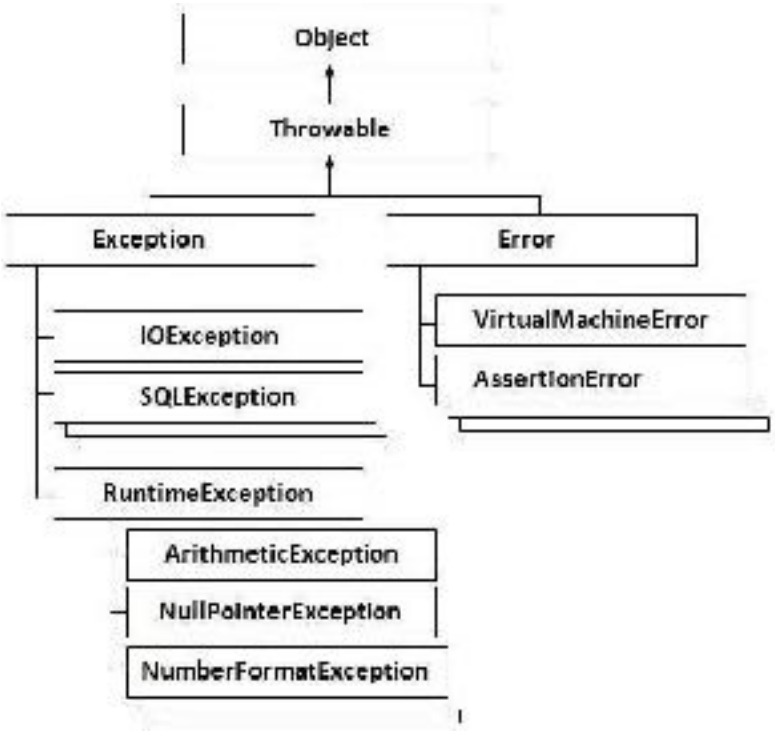
Difference between checked and unchecked exceptions

1) Checked Exception: The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

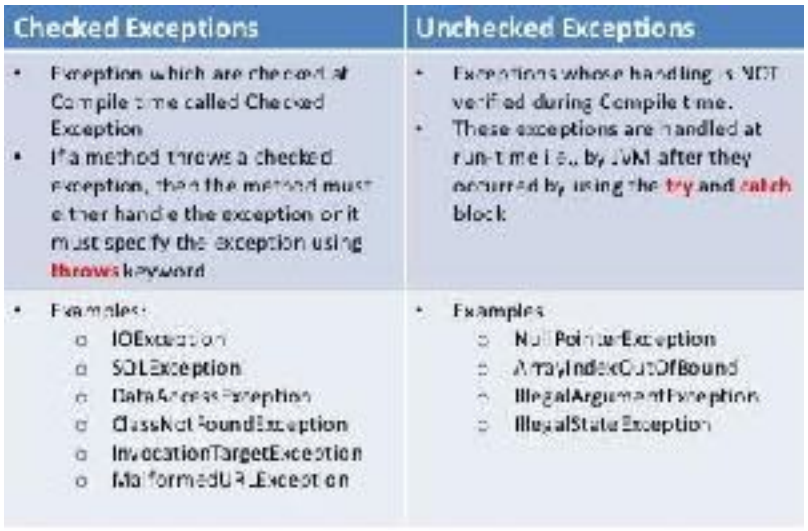
2) Unchecked Exception: The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error: Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionErroretc.

Hierarchy of Java Exception classes



Checked and UnChecked Exceptions



Checked Exceptions	Unchecked Exceptions
• Exception which are checked at Compile time called Checked Exception • If a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword	• Exceptions whose handling is NOT verified during Compile time. • These exceptions are handled at run-time i.e., by JVM after they occurred by using the try and catch block
• Examples: ○ IOException ○ SQLException ○ DataAccessException ○ ClassNotFoundException ○ InvocationTargetException ○ MalformedURLException	• Examples ○ NullPointerException ○ ArrayIndexOutOfBound ○ IllegalArgumentException ○ IllegalStateException

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

Syntax of java try-catch

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

Syntax of try-finally block

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

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch

block with a single try.

Multithreading In Java

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

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

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

1)Itdoesn’tblocktheuserbecausethreadsareindependentandyoucanperformmultiple operations at same time.

2) You can perform many operations together so it saves time.
3) Threads are independent so it doesn't affect other threads if exception occur in a single thread.

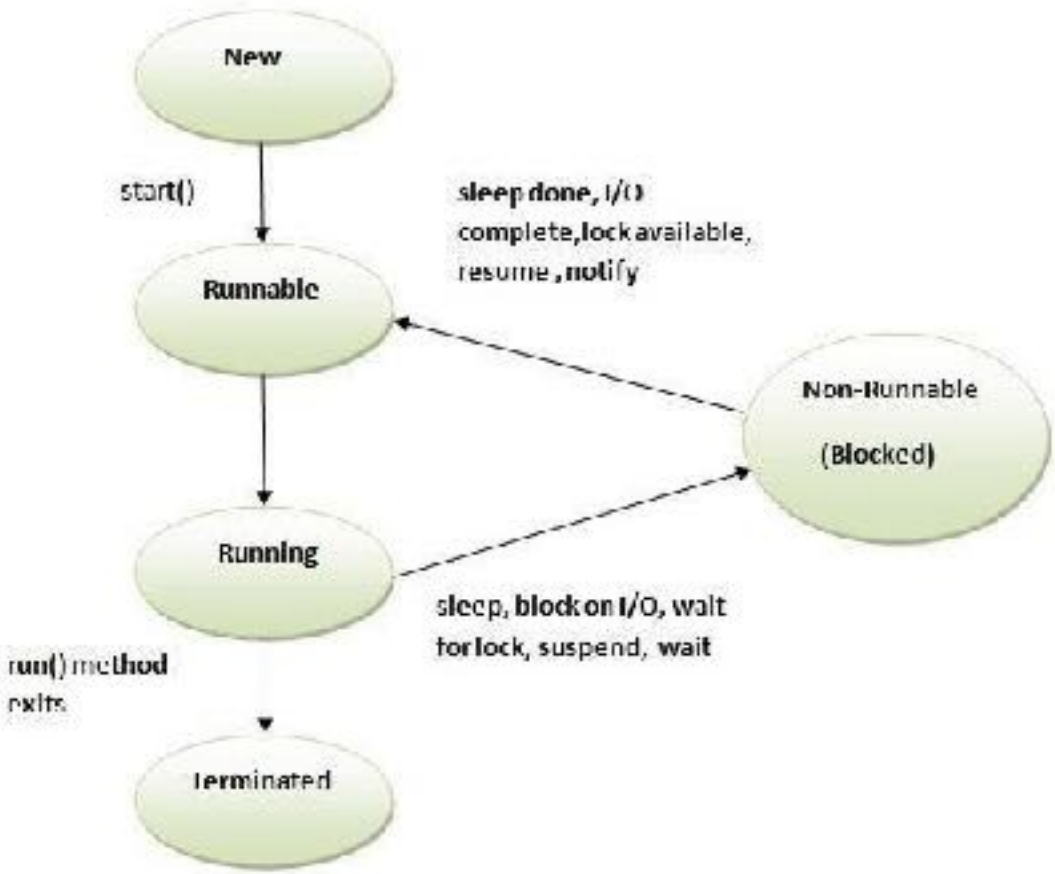
Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



How to create thread

There are two ways to create a thread:

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

Thread class:

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

Commonly used Constructors of Thread class:

o Thread()
oThread(String name) oThread(Runnable r)
oThread(Runnable r,String name)

Commonly used methods of Thread class:

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

6. public int getPriority(): returns the priority of the thread.
7. public int setPriority(int priority): changes the priority of the thread.
8. public String getName(): returns the name of the thread.
9. public void setName(String name): changes the name of the thread.
10. public Thread currentThread(): returns the reference of currently executing thread.
11. public int getId(): returns the id of the thread.
12. public Thread.State getState(): returns the state of the thread.
13. public boolean isAlive(): tests if the thread is alive.
14. public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. public void suspend(): is used to suspend the thread(depricated).
16. public void resume(): is used to resume the suspended thread(depricated).
17. public void stop(): is used to stop the thread(depricated).
18. public boolean isDaemon(): tests if the thread is a daemon thread.

19. public void setDaemon(boolean b): marks the thread as daemon or user thread.

20. public void interrupt(): interrupts the thread.

21. public boolean isInterrupted(): tests if the thread has been interrupted.

22. public static boolean interrupted(): tests if the current thread has been interrupted.

Runnable interface:
The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().
public void run(): is used to perform action for a thread.

Starting a thread:

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

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

Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){ System.out.println("thread is running..."); }

public static void main(String args[]){ Multi t1=new Multi();
t1.start();

}}
```

```
class Multi3 implements Runnable{ public void run()
{ System.out.println("thread is running..."); }

public static void main(String args[]){ Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();

}}
```

Output:thread is running...

Java Thread Example by implementing Runnable interface

Output:thread is running...

Priority of a Thread (Thread Priority):

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

3 constants defined in Thread class:

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

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

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{ public void run(){

System.out.println("running thread name
is:"+Thread.currentThread().getName());

System.out.println("running thread priority
is:"+Thread.currentThread().getPriority()); }

public static void main(String args[]){

TestMultiPriority1 m1=new TestMultiPriority1();
TestMultiPriority1 m2=new TestMultiPriority1();
m1.setPriority(Thread.MIN_PRIORITY);
m2.setPriority(Thread.MAX_PRIORITY); m1.start();

m2.start(); }}
```

Output:

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