Java Package

A java package is a group of similar types of classes, interfaces and sub-packages.

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose. We can reuse existing classes from the packages as many time as we need it in our program.

How packages work?

Package names and directory structure are closely related. For example if a package name is college.staff.cse, then there are three directories, college, staffand cse such that cse is present in staff and staff is present inside college. Also, the directory college is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.

Package naming conventions : Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new java file to define a public class, otherwise we can add the new class to an existing .java file and recompile it.

Subpackages: Packages that are inside another package are the subpackages. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example :

```
import java.util.*;
```

util is a subpackage created inside java package.

Accessing classes inside a package

Consider following two statements :

// import the Vector class from util package.

```
import java.util.vector;
```

// import all the classes from util package

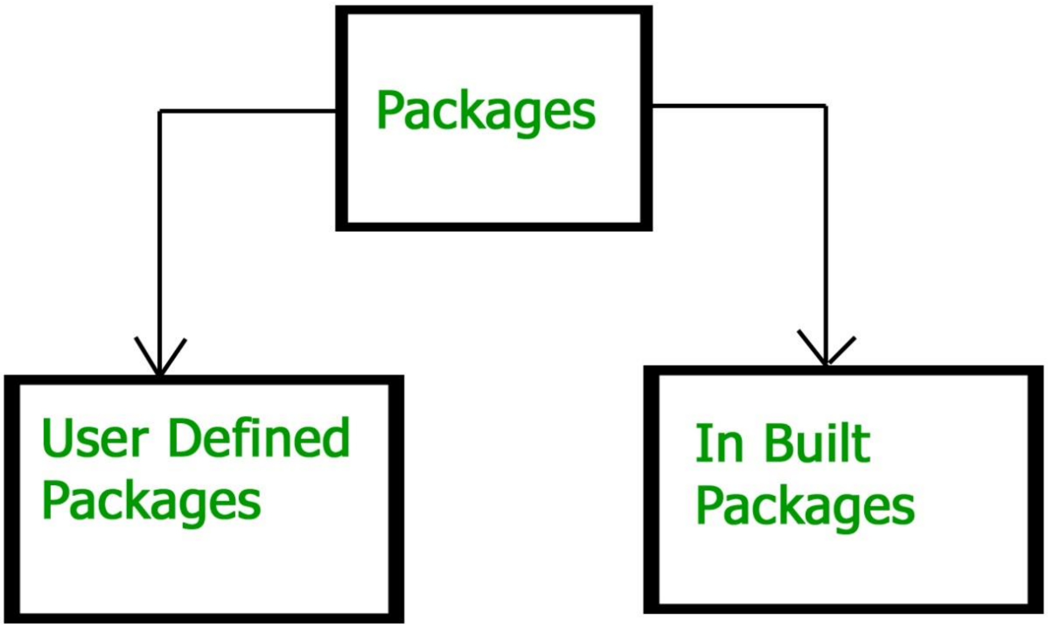
```
import java.util.*;
```

First Statement is used to import Vector class from util package which is contained inside java.

Second statement imports all the classes from util package.

```
// All the classes and interfaces of this package
// will be accessible but not subpackages.
import package.*;

//  Only  mentioned  class  of  this  package  will  be
accessible.
import package.classname;

// Class name is generally used when two packages have
the same
// class name. For example in below code both packages
have
// date class so using a fully qualified name to avoid
conflict
import java.util.Date;
import my.package.Date;
```

Types of packages:



Built-in Packages

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

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

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

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;


public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the MyClass class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();

      obj.getNames(name);
   }
}
```

Note : MyClass.java must be saved inside the myPackage directory since it is a part of the package.

Using Static Import

Static import is a feature introduced in Java programming language (versions 5 and above) that allows members (fields and methods) defined in a class as public static to be used in Java code without specifying the class in which the field is defined. Following program demonstrates static import:

```
// Note static keyword after import.
import static java.lang.System.*;

class StaticImportDemo {
    public static void main(String args[])
    {
        // We don't need to use 'System.out'
        // as imported using static.
        out.println("GeeksforGeeks");
    }
}
```

Output:

```
 GeeksforGeeks
```

Handling name conflicts

The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

```
import java.util.*;
import java.sql.*;

//And then use Date class, then we will get a compile-
time error :

Date today ; //ERROR-- java.util.Date or java.sql.Date?
```

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:

```
import java.util.Date;
import java.sql.*;
```

If we need both Date classes then, we need to use a full package name every time we declare a new object of that class. For Example:

```
java.util.Date deadLine = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

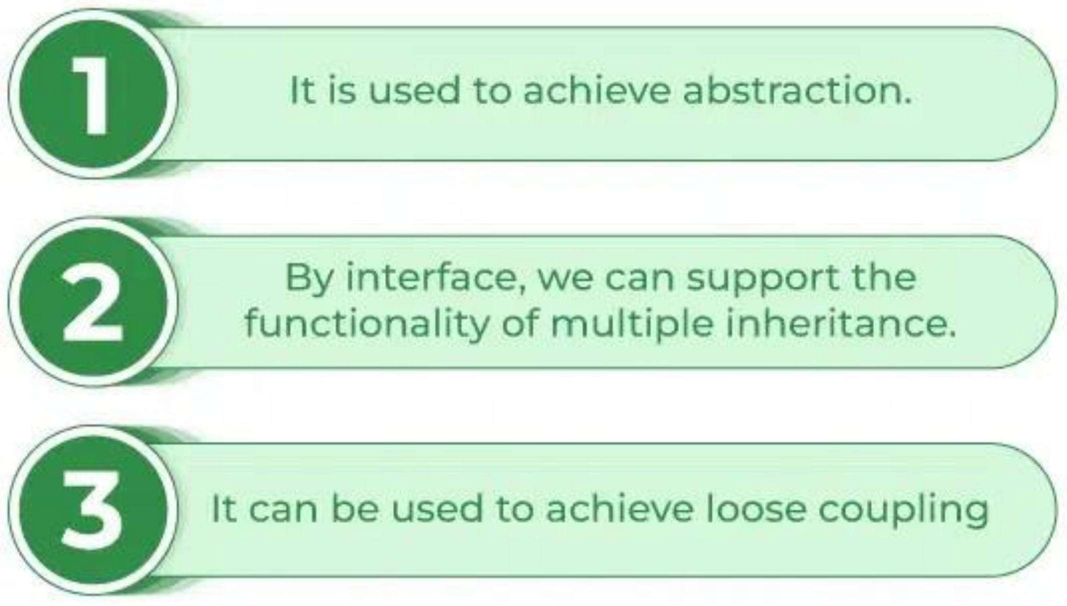
Interfaces in Java

An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

What are Interfaces in Java?
The interface in Java is a mechanism to achieve abstraction. Traditionally, an interface could only have abstract methods (methods without a body) and public, static, and final variables by default. It is used to achieve abstraction and multiple inheritances in Java. In other words, interfaces primarily define methods that other classes must implement. Java Interface also represents the IS-A relationship.

In Java, the abstract keyword applies only to classes and methods, indicating that they cannot be instantiated directly and must be implemented.

When we decide on a type of entity by its behavior and not via attribute we should define it as an interface.


1
It is used to achieve abstraction.
2
By interface, we can support the functionality of multiple inheritance.
3
It can be used to achieve loose coupling

Syntax for Java Interfaces

```
interface {

    // declare constant fields

    // declare methods that abstract

    // by default.

}
```

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement the interface, use the implements keyword.

Uses of Interfaces in Java

Uses of Interfaces in Java are mentioned below:

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- Any class can extend only 1 class, but can any class implement an infinite number of interfaces.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction.

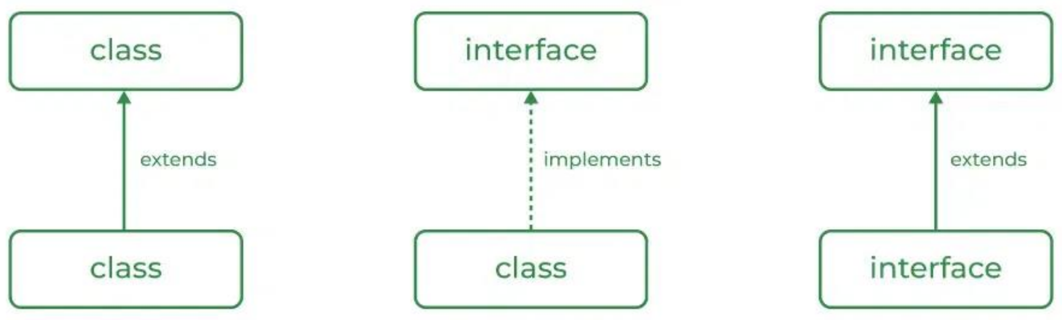
So, the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

```
// A simple interface

interface Player

{

    final int id = 10;
    int move();
}
```

Relationship Between Class and Interface

A class can extend another class similar to this an interface can extend another interface. But only a class can extend to another interface, and vice-versa is not allowed.



Difference Between Class and Interface

Although Class and Interface seem the same there have certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

Class	**Interface**
In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
A class can contain concrete (with implementation) methods	The interface cannot contain concrete (with implementation) methods.
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Implementation: To implement an interface, we use the keyword implements

```
// Java program to demonstrate working of
// interface

import java.io.*;

// A simple interface
interface In1 {

    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}
```

```
// A class that implements the interface.
class TestClass implements In1 {

    // Implementing the capabilities of
    // interface.
    public void display(){
      System.out.println("Geek");
    }

    // Driver Code
    public static void main(String[] args)
    {
        TestClass t = new TestClass();
        t.display();
        System.out.println(t.a);
    }
}
```

Output

```
Geek
10
```

Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

- Without bothering about the implementation part, we can achieve the security of the implementation.

- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

Access Modifiers in Java

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

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

There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

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

1) Private

The private access modifier is accessible only within the 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 a 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
   }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

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

Note: A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

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.

3) Protected

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.

It provides more accessibility than the default modifer.

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

4) Public

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

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

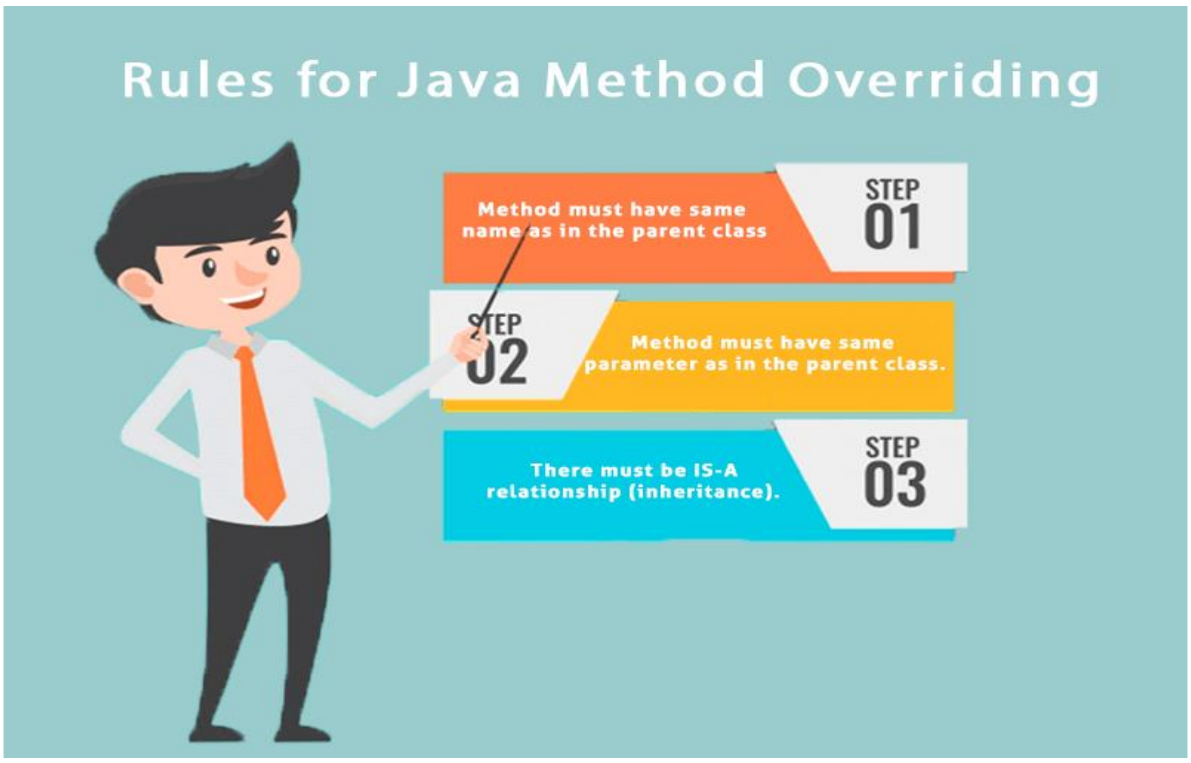
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).


Rules for Java Method Overriding
Method must have same name as in the parent class
STEP 01
STEP 02
Method must have same parameter as in the parent class.
There must be IS-A relationship (inheritance).
STEP 03

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output:

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

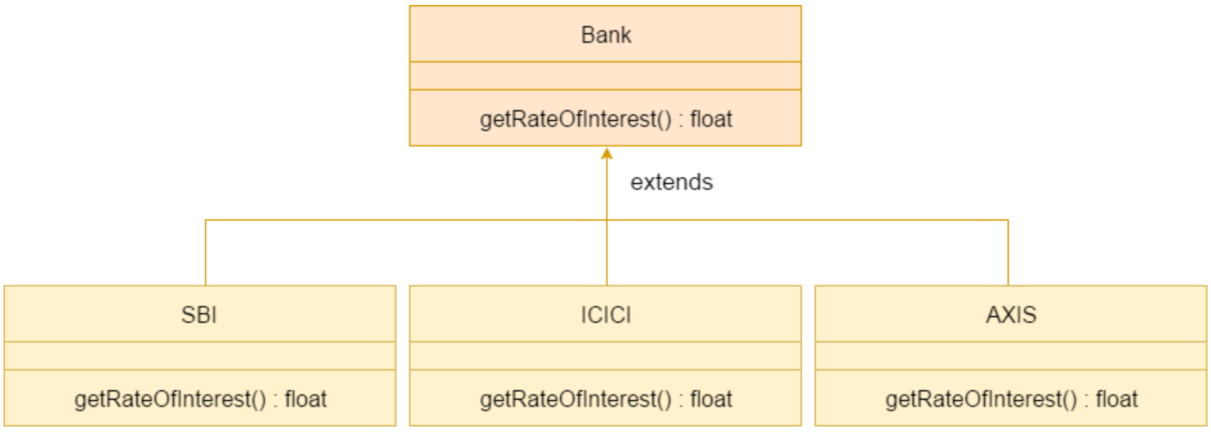
```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

Bike is running safely

A real example of Java Method Overriding

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


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

```
//Java Program to demonstrate the real scenario of Java Method Overriding

//where three classes are overriding the method of a parent class.

//Creating a parent class.

class Bank{

int getRateOfInterest(){return 0;}

}

//Creating child classes.
```

```
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI      Rate      of      Interest:
"+s.getRateOfInterest());
System.out.println("ICICI     Rate     of     Interest:
"+i.getRateOfInterest());
System.out.println("AXIS     Rate     of     Interest:
"+a.getRateOfInterest());
}
}
```

Output:

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

Garbage collection in Java

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

What is Garbage Collection?

In C/C++, a programmer is responsible for both the creation and destruction of objects. Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing OutOfMemoryErrors.

But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying unreachable objects. The garbage collector is the best example of the Daemon thread as it is always running in the background.

How Does Garbage Collection in Java works?

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Types of Activities in Java Garbage Collection

Two types of garbage collection activity usually happen in Java. These are:

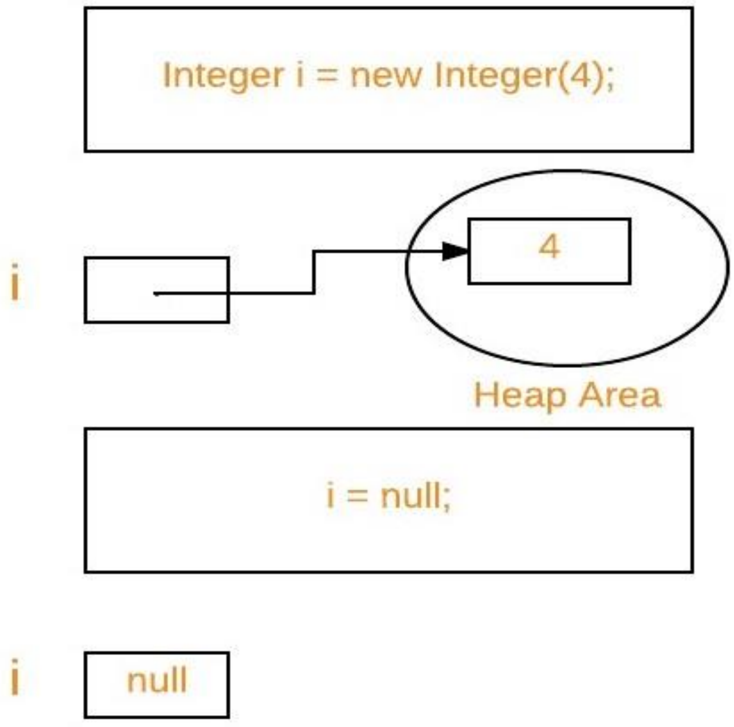
Minor or incremental Garbage Collection: It is said to have occurred when unreachable objects in the young generation heap memory are removed.

Major or Full Garbage Collection: It is said to have occurred when the objects that survived the minor garbage collection are copied into the old generation or permanent generation heap memory are removed. When compared to the young generation, garbage collection happens less frequently in the old generation.

Important Concepts Related to Garbage Collection in Java

1. Unreachable objects: An object is said to be unreachable if it doesn't contain any reference to it. Also, note that objects which are part of the island of isolation are also unreachable.

```
Integer i = new Integer(4);

// the new Integer object is reachable  via the
reference in 'i'

i = null;

// the Integer object is no longer reachable.
```



2. Eligibility for garbage collection: An object is said to be eligible for GC(garbage collection) if it is unreachable. After i = null, integer object 4 in the heap area is suitable for garbage collection in the above image.

Ways to make an object eligible for Garbage Collector

Even though the programmer is not responsible for destroying useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.

There are generally four ways to make an object eligible for garbage collection.

1. Nullifying the reference variable
2. Re-assigning the reference variable
3. An object created inside the method
4. Island of Isolation

Ways for requesting JVM to run Garbage Collector

- Once we make an object eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
- We can also request JVM to run Garbage Collector. There are two ways to do it :

1. **Using System.gc() method:** System class contain static method gc() for requesting JVM to run Garbage Collector.
2. Using Runtime.getRuntime().gc() method: Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.
3. There is no guarantee that any of the above two methods will run Garbage Collector.
4. The call System.gc() is effectively equivalent to the call : Runtime.getRuntime().gc()

Finalization

Just before destroying an object, Garbage Collector calls finalize() method on the object to perform cleanup activities. Once finalize() method completes, Garbage Collector destroys that object.

finalize() method is present in Object class with the following prototype.

```
protected void finalize() throws Throwable
```

Based on our requirement, we can override finalize() method for performing our cleanup activities like closing connection from the database.

1. The finalize() method is called by Garbage Collector, not JVM. However, Garbage Collector is one of the modules of JVM.
2. Object class finalize() method has an empty implementation. Thus, it is recommended to override the finalize() method to dispose of system resources or perform other cleanups.
3. The finalize() method is never invoked more than once for any object.
4. If an uncaught exception is thrown by the finalize() method, the exception is ignored, and the finalization of that object terminates.

Advantages of Garbage Collection in Java

The advantages of Garbage Collection in Java are:

1. It makes java memory-efficient because the garbage collector removes the unreferenced objects from heap memory.
2. It is automatically done by the garbage collector(a part of JVM), so we don’t need extra effort.

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

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.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

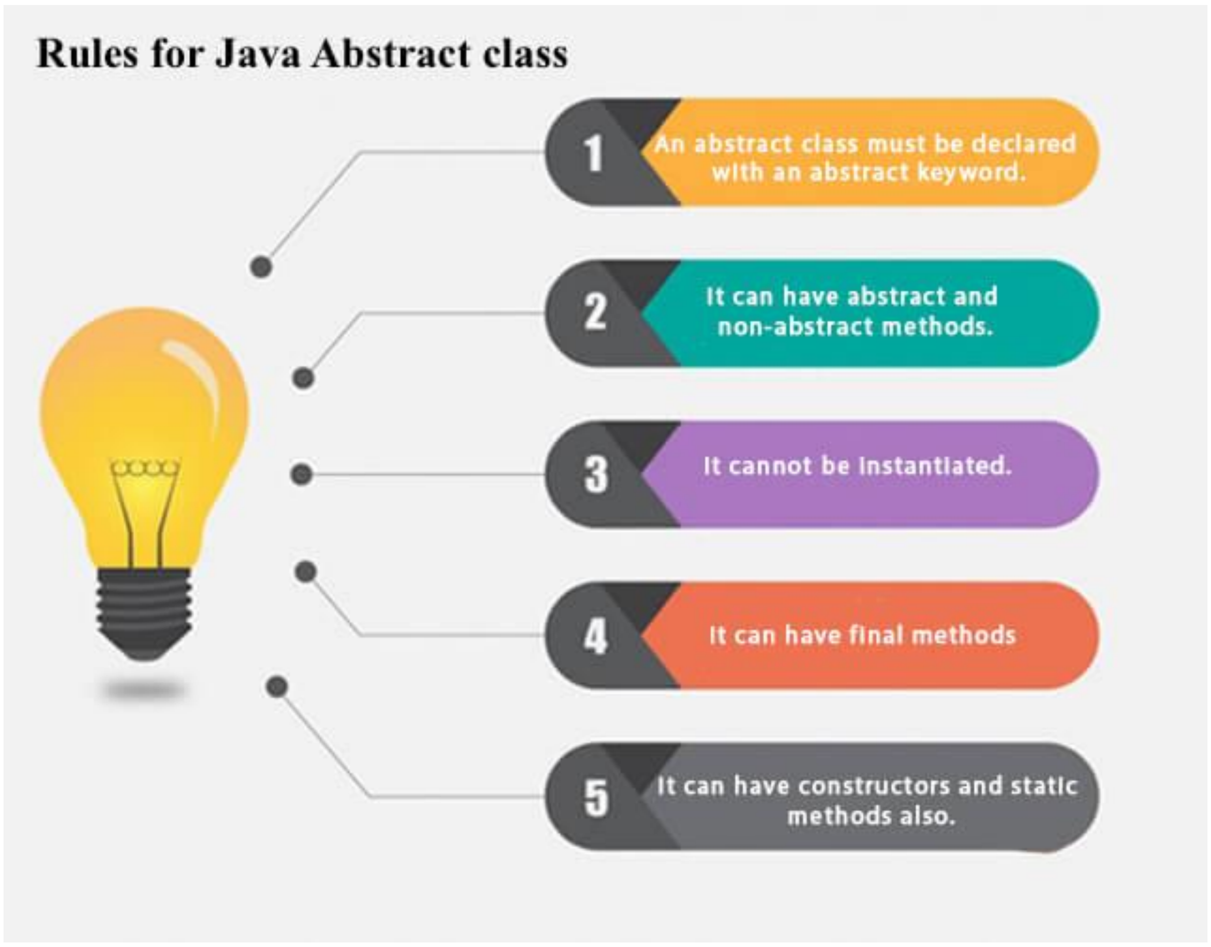
1. Abstract class (0 to 100%)
2. Interface (100%)

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.


Rules for Java Abstract class
1 An abstract class must be declared with an abstract keyword.
2 It can have abstract and non-abstract methods.
3 It cannot be instantiated.
4 It can have final methods
5 It can have constructors and static methods also.

Example of abstract class

```
abstract class A{}
```

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
```

Example of Abstract class that has an abstract method

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

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

Output

```
running safely
```

Exception Handling in Java

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

What is Exception in Java?
Dictionary Meaning: Exception is an abnormal condition.

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

What is Exception Handling?
Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling
The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
```

statement 8;

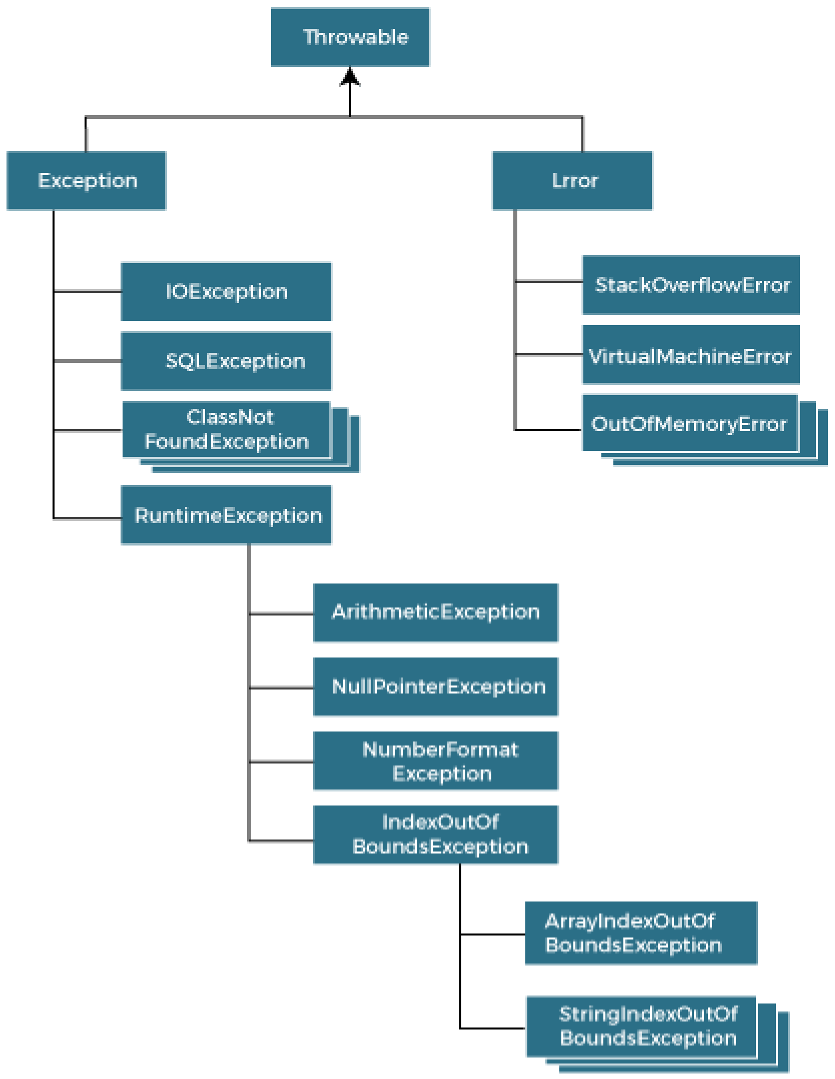
statement 9;

statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

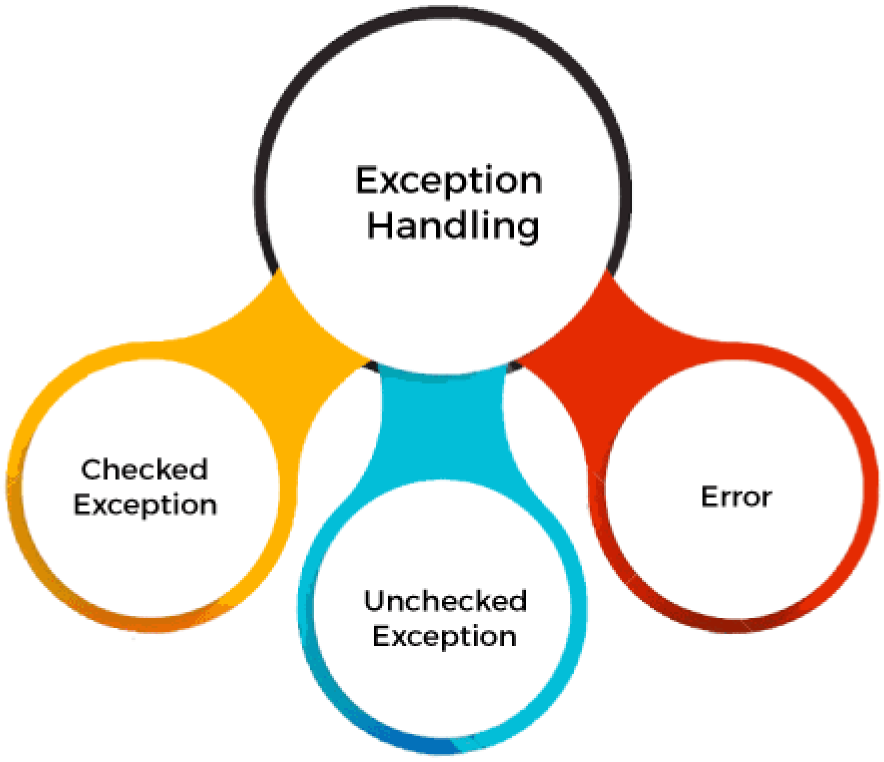
The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

- Checked Exception
- Unchecked Exception
- Error



Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords
Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example
Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
public class JavaExceptionExample{
 public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException
e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Ouput

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions
There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Throw And Throws in Java

In Java, Exception Handling is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

In this article, we will learn about throw and throws in Java which can handle exceptions in Java.

Java throw

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Syntax in Java throw

```
throw Instance
```

```
Example:
```

```
throw new ArithmeticException("/ by zero");
```

But this exception i.e., Instance must be of type Throwable or a subclass of Throwable.

For example, an Exception is a sub-class of Throwable and user-defined exceptions typically extend the Exception class. Unlike C++, data types such as int, char, floats, or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked, and so on. If no matching catch is found then the default exception handler will halt the program.

Java throw Examples

Example:

```
// Java program that demonstrates the use of throw
class ThrowExcep {
    static void fun()
    {
        try {
            throw new NullPointerException("demo");
        }
        catch (NullPointerException e) {
            System.out.println("Caught inside 
fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try {
            fun();
        }
        catch (NullPointerException e) {
            System.out.println("Caught in main.");
        }
```

```
      }
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Java throws

throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

```
Syntax of Java throws
type method_name(parameters) throws exception_list

exception_list is a comma separated list of all the
exceptions which a method might throw.
```

In a program, if there is a chance of raising an exception then the compiler always warns us about it and compulsorily we should handle that checked exception, Otherwise, we will get compile time error saying unreported exception XXX must be caught or declared to be thrown. To prevent this compile time error we can handle the exception in two ways:

1. By using try catch
2. By using the throws keyword

We can use the throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then the caller method is responsible to handle that exception.

Java throws Examples

```
// Java program to illustrate error in case
// of unhandled exception
class tst {
    public static void main(String[] args)
    {
        Thread.sleep(10000);
        System.out.println("Hello Geeks");
    }
}
```

Output
error: unreported exception InterruptedException; must be caught or declared to be thrown

Explanation
In the above program, we are getting compile time error because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute the main() method which will cause InterruptedException.

Important Points to Remember about throws Keyword
throws keyword is required only for checked exceptions and usage of the throws keyword for unchecked exceptions is meaningless.

throws keyword is required only to convince the compiler and usage of the throws keyword does not prevent abnormal termination of the program.

With the help of the throws keyword, we can provide information to the caller of the method about the exception.

Java try-catch block

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

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

Syntax of try-finally block

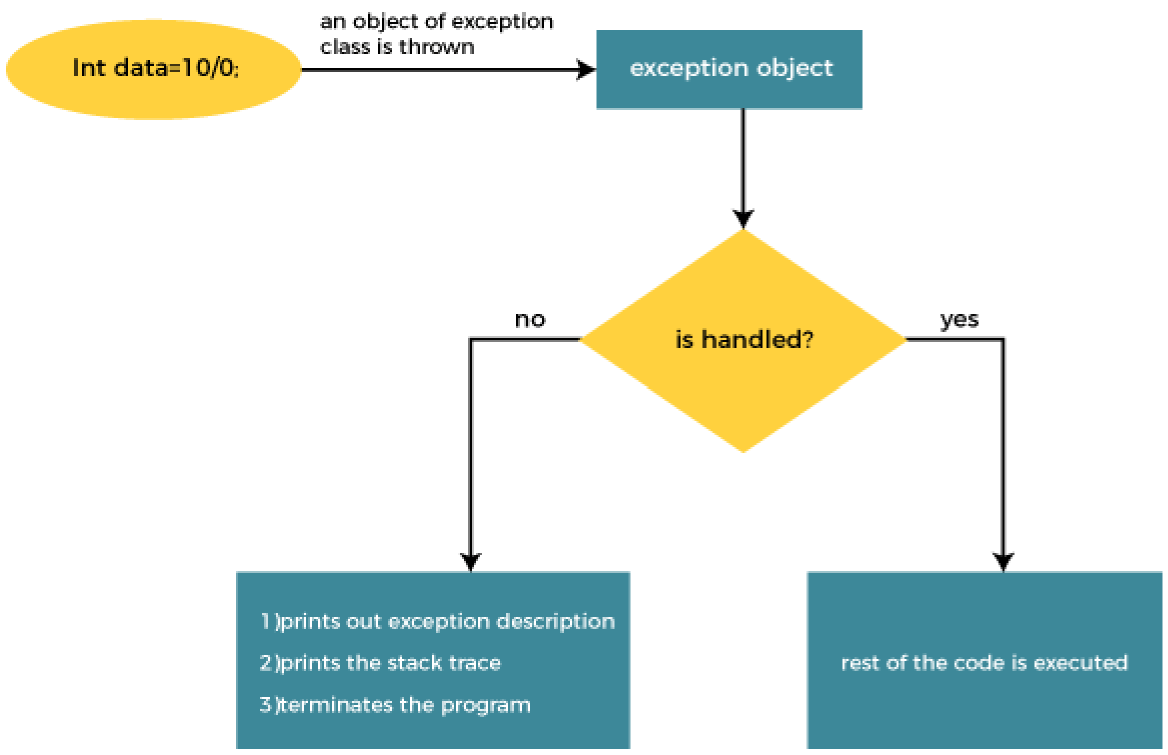
```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block


Int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Multithreading in Java

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

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

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

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

Advantages of Java Multithreading

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

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

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

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.

- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

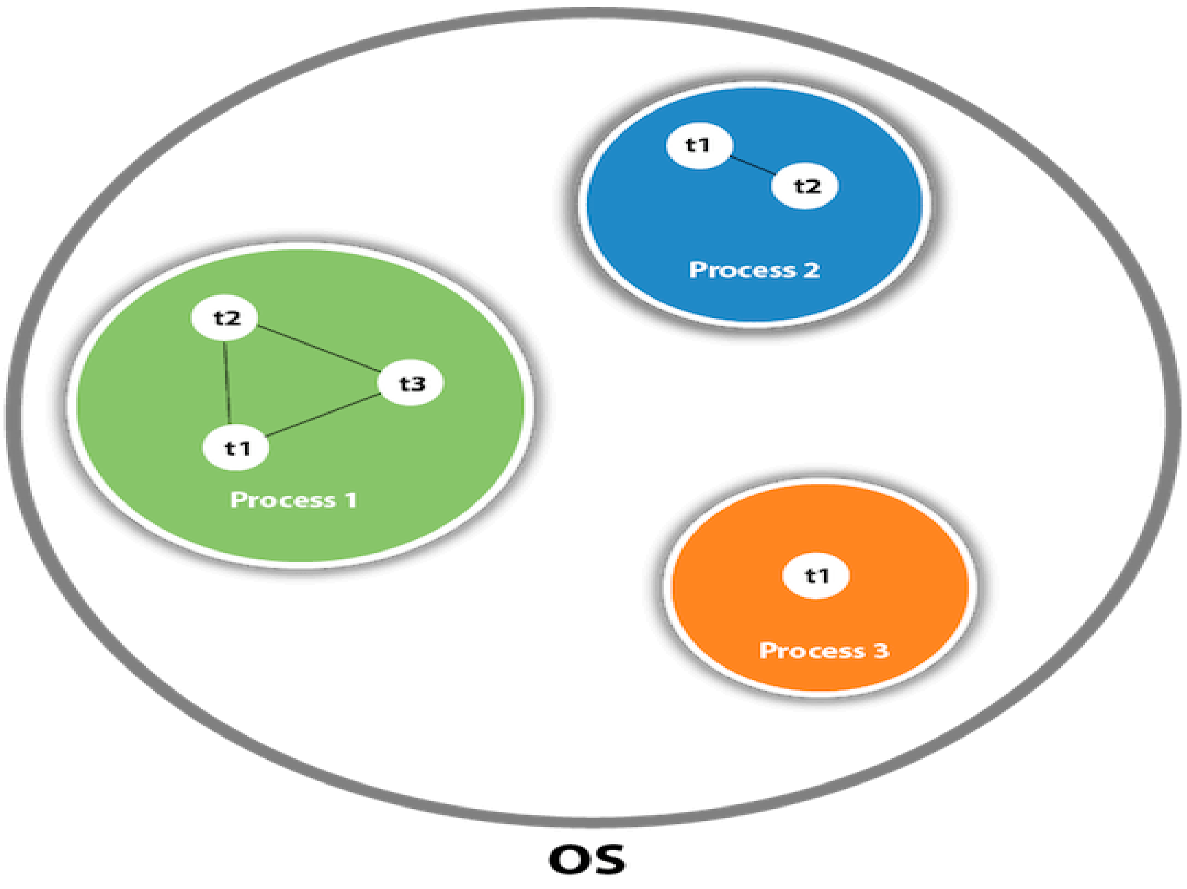
2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java
A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Java Thread class

Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

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

Why use Synchronization?

The synchronization is mainly used to

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

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

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

1. Mutual Exclusive
    a. Synchronized method.
    b. Synchronized block.
    c. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.