

Packages

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

In the preceding chapters, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.

The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

(This is why you haven’t had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

```
package pkg;
```

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in **java\awt\image** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH - As just explained, packages are mirrored by directories. This raises an important question:

How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example, consider the following package specification:

```
package MyPack
```

In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

When the second two options are used, the class path *must not* include **MyPack**, itself.

It must simply specify the *path to MyPack*. For example, in a Windows environment, if the path to **MyPack** is

```
C:\MyPrograms\Java\MyPack
```

Then the class path to **MyPack** is

```
C:\MyPrograms\Java
```

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example - Keeping the preceding discussion in mind, you can try this simple package:

```
package MyPack;
class Balance{
    String name;
    double bal;
    Balance(String n, double b){
        name = n;
        bal = b;
    }
    void show(){
        if(bal < 0)
            System.out.print("-->");
        System.out.println(name + ": $" + bal);
    }
}

public class AccountBalance{
    public static void main(String args[]){
        Balance current[] = new Balance[3];
        current[0] = new Balance("K.J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);
        for(int i = 0; i < 3; i++) current[i].show();
    }
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command.

(Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)

As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

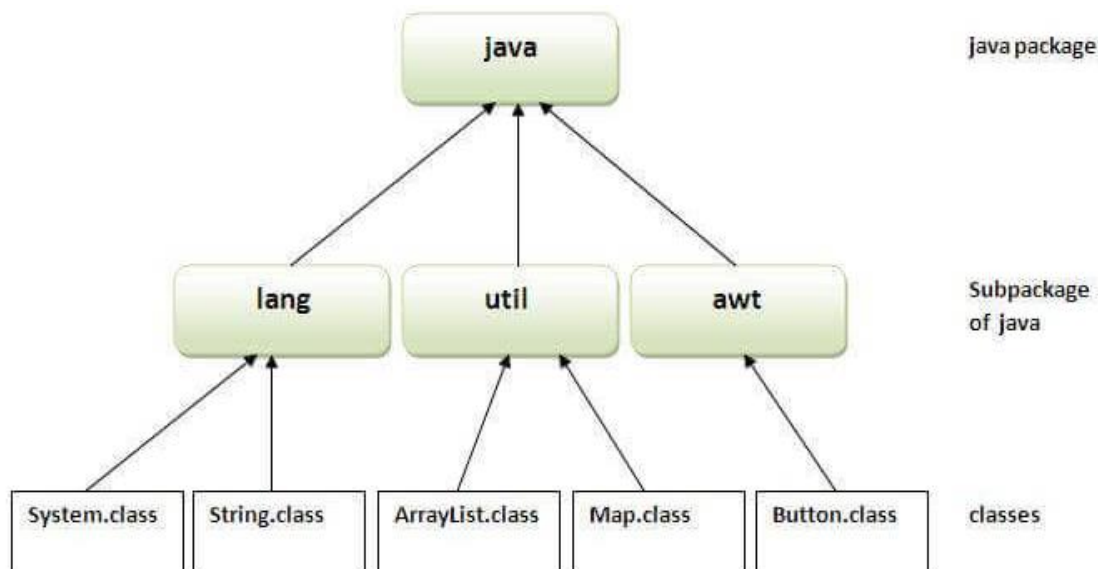
Java Package Revisited:

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

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

Advantage of Java Package

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



The **package keyword** is used to create a package in java.

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

How to compile java package - If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program - You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

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

How to access package from another package? There are three ways to access the package from outside the package.

```
import package.*;
import package.classname;
fully qualified name.
```

1) Using packagename.*

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

Example of package that import the packagename.*

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

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

Output: Hello

2) Using packagename.classname - If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

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

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

```
}
```

Output: Hello

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

Example of package by import fully qualified name

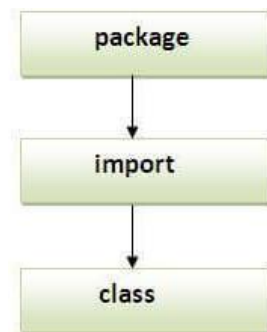
```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

Output: Hello

Note: If you import a package, subpackages will not be imported.

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

Note: Sequence of the program must be package then import then class.



Subpackage in java

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

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

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
package com.javatpoint.core;
class Simple{
    public static void main(String args[]){
```

```

    System.out.println("Hello subpackage");
}
}

```

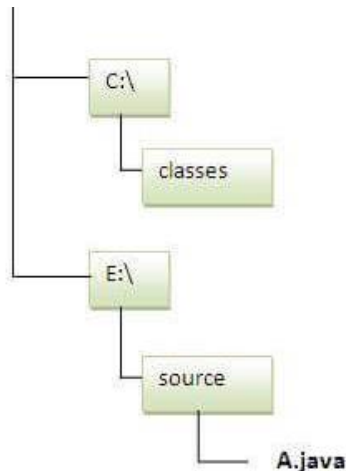
To Compile: `javac -d . Simple.java`

To Run: `java com.javatpoint.core.Simple`

Output: Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



```

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

```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output: Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
 - By setting the classpath in the command prompt
 - By -classpath switch
- Permanent
 - By setting the classpath in the environment variables
 - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
//save as C.java otherwise Compile Time Error
class A{}
class B{}
public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
//save as A.java
package javatpoint;
public class A{}
//save as B.java

package javatpoint;
public class B{}
```

Interface in Java

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.

By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

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.

Since Java 9, we can have **private methods** in an interface.

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

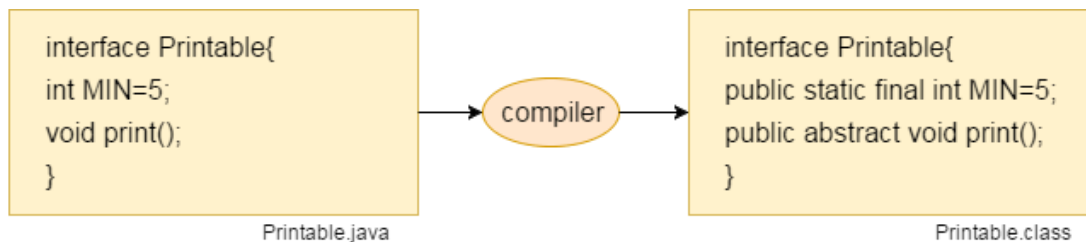
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

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.

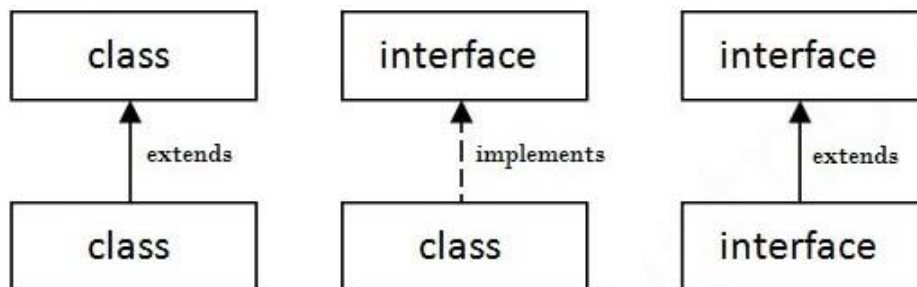
```
interface <interface_name>{  
  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```

Java 8 Interface Improvement - Since Java 8, interface can have default and static methods which is discussed later.

Internal addition by the compiler - The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members. In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



The relationship between classes and interfaces - As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



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.

File: TestInterface1.java


```
//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. getDrawable()
d.draw();
}}
```

Output: drawing circle

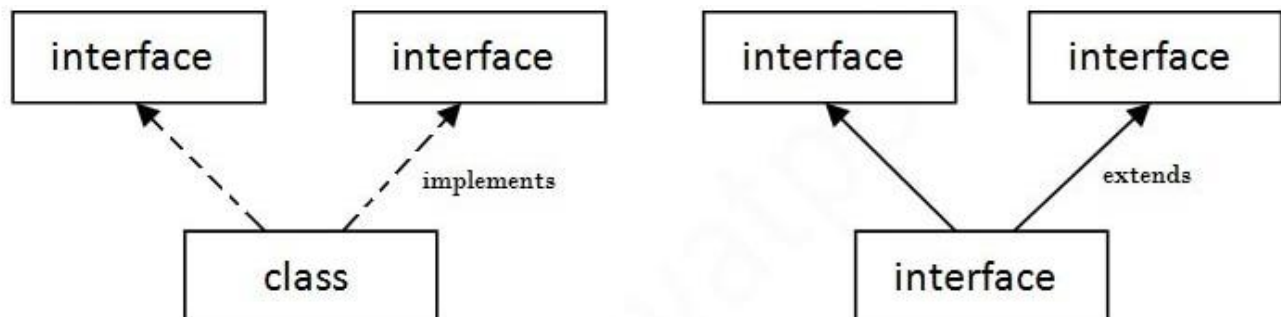
Java Interface Example: Bank - Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Output: ROI: 9.15

Multiple inheritance in Java by interface - If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
```

```

interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}

```

Output: Hello

Welcome

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why? As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```

interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
}
}

```

Output: Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestInterface1, so there is no ambiguity.

Interface inheritance - A class implements an interface, but one interface extends another interface.

```

interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
}
}

```

Output:

Hello
Welcome

Java 8 Default Method in Interface - Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Output:

drawing rectangle
default method

Java 8 Static Method in Interface - Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Output:

drawing rectangle
27

Q) What is marker or tagged interface? An interface which has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{
}
```

Nested Interface in Java

Note: An interface can have another interface which is known as a nested interface. For example:

```
interface printable{
```

```

void print();
interface MessagePrintable{
    void msg();
}
}

```

Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack.

However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push()** and **pop()** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

```

interface IntStack{
    void push(int item); // store an item
    int pop(); // retrieve an item
}

// An implementation of IntStack that uses fixed storage
class FixedStack implements IntStack{
    private int stck[];
    private int tos;

    // allocate and initialize stack
    FixedStack(int size){
        stck = new int[size];
        tos = -1;
    }

    // push an item onto the stack
    public void push(int item){
        if(tos == stck.length - 1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // pop an item from the stack
    public int pop(){
        if(tos < 0){
            System.out.println("Stack underflow.");
            return 0;
        } else {
            return stck[tos--];
        }
    }
}

public class IFTest {
    public static void main(String args[]){
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);

        // push some numbers onto the stack
        for(int i = 0; i < 5; i++) mystack1.push(i);
        for(int i = 0; i < 8; i++) mystack2.push(i);

        // pop those numbers off the stack
    }
}

```

```

        System.out.println("Stack in mystack1: ");
        for(int i = 0; i < 5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2: ");
        for(int i = 0; i < 8; i++)
            System.out.println(mystack2.pop());
    }
}

```

Following is another implementation of `IntStack` that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```

interface IntStack{
    void push(int item);
    int pop();
}
// implement a "growable" stack
class DynStack implements IntStack{
    private int stck[];
    private int tos;
    DynStack(int size){
        stck = new int[size];
        tos = -1;
    }
    public void push(int item){
        if(tos == stck.length - 1){
            int temp[] = new int[stck.length * 2]; // double size
            for(int i = 0; i < stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        } else {
            stck[++tos] = item;
        }
    }
    public int pop(){
        if(tos < 0){
            System.out.println("Stack underflow");
            return 0;
        } else
            return stck[tos--];
    }
}

public class IFTest{
    public static void main(String args[]){
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        // push some numbers onto the stack
        for(int i = 0; i < 12; i++) mystack1.push(i);
        for(int i = 0; i < 20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1: ");
        for(int i = 0; i < 12; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2: ");
        for(int i = 0; i < 20; i++)
            System.out.println(mystack2.pop());
    }
}

```

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.) If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything.

```
import java.util.Random;
interface SharedConstants{
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants{
    Random rand = new Random();
    int ask(){
        int prob = (int) (100 * rand.nextDouble());
        if(prob < 30) return NO; // 30%
        else if(prob < 60) return YES; // 30%
        else if(prob < 75) return LATER; // 15%
        else if(prob < 98) return SOON; // 13%
        else return NEVER; // 2%
    }
}

public class AskMe implements SharedConstants {
    static void answer(int result){
        switch(result){
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }

    public static void main(String args[]){
        Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

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, its 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.

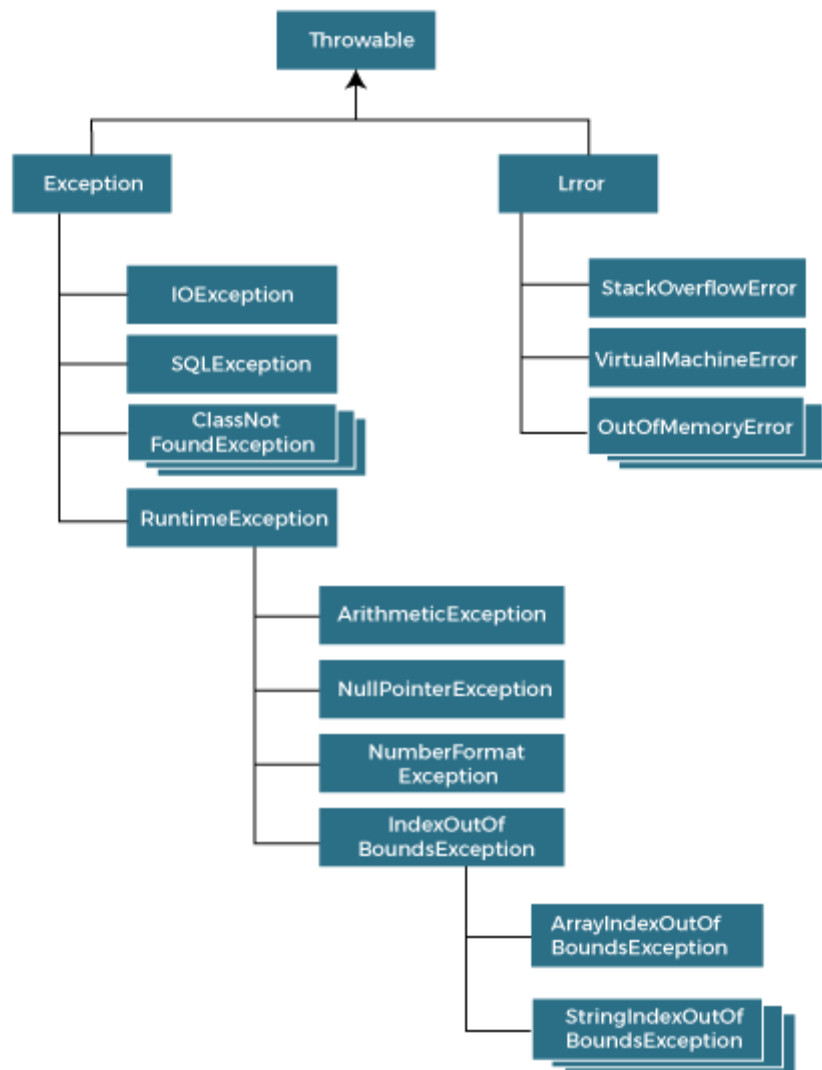
What is Exception Handling? Exception Handling is a mechanism to handle runtime errors such as `ClassNotFoundException`, `IOException`, `SQLException`, `RemoteException`, etc.

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:

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...");
    }
}
```

Output:

```
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 its 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
```

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){}
```

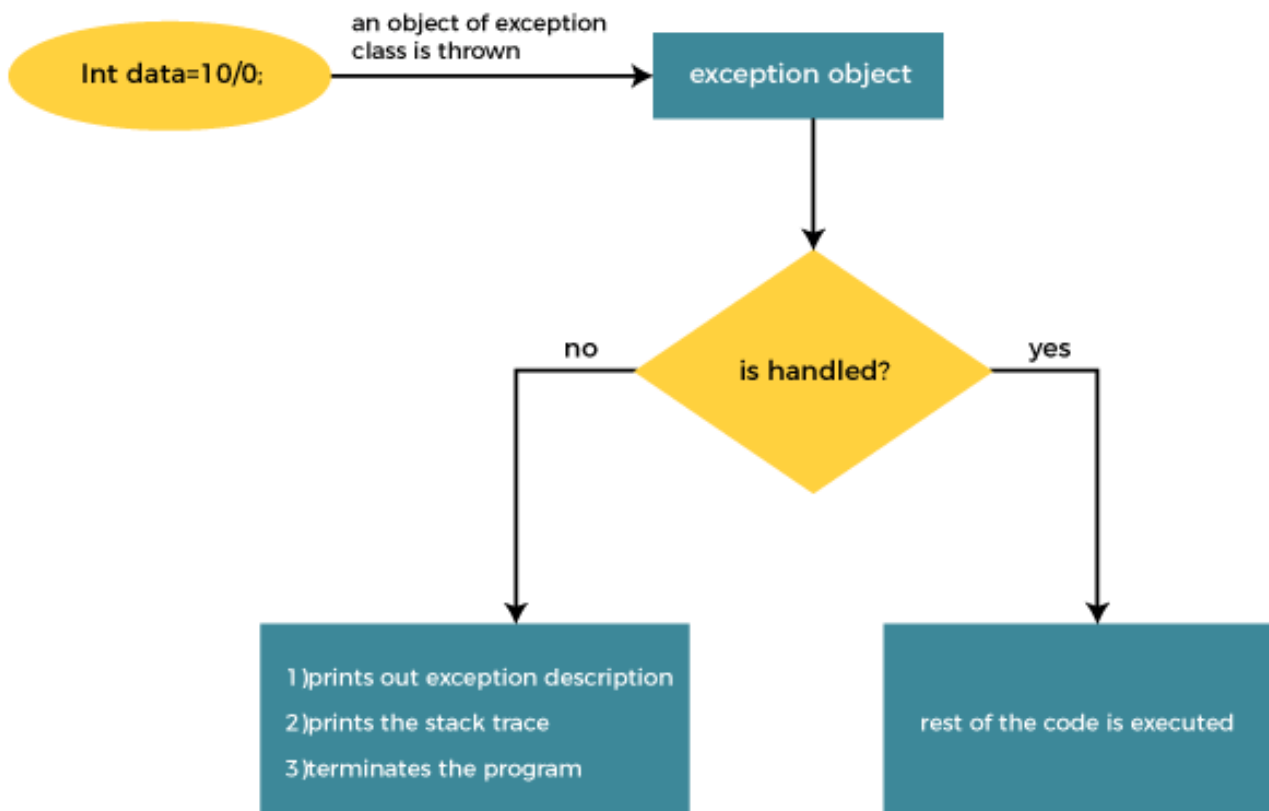
Syntax of try-finally block

```
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



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.

Problem without exception handling - Let's try to understand the problem if we don't use a try-catch block.

TryCatchExample1.java

```
public class TryCatchExample1 {
    public static void main(String[] args) {
        int data=50/0; //may throw exception
        System.out.println("rest of the code");
    }
}
```

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling - Let's see the solution of the above problem by a java try-catch block.

```
public class TryCatchExample2 {
    public static void main(String[] args) {
        try {
            int data=50/0; //may throw exception
        }
        //handling the exception
        catch(ArithmeticException e){
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

In this example, we also kept the code in a try block that will not throw an exception.

```
public class TryCatchExample3 {
    public static void main(String[] args) {
        try
        {
            int data=50/0; //may throw exception
            if exception occurs, the remaining statement will not execute
            System.out.println("rest of the code");
        }
        // handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
    }
}
```

Output: java.lang.ArithmeticException: / by zero

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Here, we handle the exception using the parent class exception.

```
public class TryCatchExample4 {
    public static void main(String[] args) {
        try
        {
            int data=50/0; //may throw exception
        }
        // handling the exception by using Exception class
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Let's see an example to print a custom message on exception.

```
public class TryCatchExample5 {
    public static void main(String[] args) {
        try {
            int data=50/0; //may throw exception
        }
        // handling the exception
        catch(Exception e){
            // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }
}
```

Output: Can't divided by zero

Let's see an example to resolve the exception in a catch block.

```
public class TryCatchExample6 {
    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try {
            data=i/j; //may throw exception
        }
        // handling the exception
        catch(Exception e){
            // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output: 25

In this example, along with try block, we also enclose exception code in a catch block.

```
public class TryCatchExample7 {
    public static void main(String[] args) {
        try {
            int data1=50/0; //may throw exception
```

```

    }
    // handling the exception
    catch(Exception e){
        // generating the exception in catch block
        int data2=50/0; //may throw exception
    }
    System.out.println("rest of the code");
}
}

```

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

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```

public class TryCatchExample8 {
    public static void main(String[] args) {
        try {
            int data=50/0; //may throw exception
        }
        // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}

```

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

Let's see an example to handle another unchecked exception.

```

public class TryCatchExample9 {
    public static void main(String[] args) {
        try {
            int arr[]={1,3,5,7};
            System.out.println(arr[10]); //may throw exception
        }
        // handling the array exception
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}

```

Output:

```

java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code

```

Let's see an example to handle checked exception.

```

import java.io.FileNotFoundException;
import java.io.PrintWriter;
public class TryCatchExample10 {
    public static void main(String[] args) {
        PrintWriter pw;
        try {
            pw = new PrintWriter("jtp.txt"); //may throw exception
            pw.println("saved");
        }
    }
}

```

```

    }
    // providing the checked exception handler
    catch (FileNotFoundException e) {
        System.out.println(e);
    }
    System.out.println("File saved successfully");
}
}

```

Output: File saved successfully

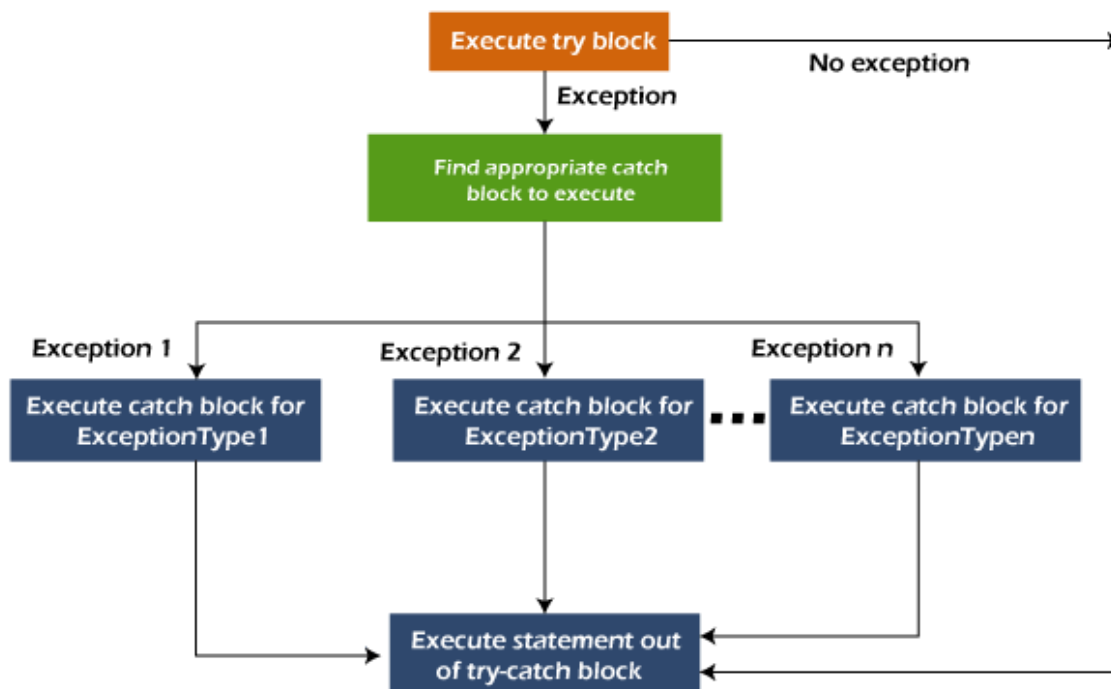
Java Catch Multiple Exceptions

Java Multi-catch block - A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

Flowchart of Multi-catch Block



Let's see a simple example of java multi-catch block.

```

public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
        } catch(ArithmeticException e){
            System.out.println("Arithmetic Exception occurs");
        } catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException occurs");
        } catch(Exception e){
            System.out.println("Parent Exception occurs");
        }
    }
}

```

```

    }
    System.out.println("rest of the code");
}
}

```

Output:

Arithmetic Exception occurs
rest of the code

```

public class MultipleCatchBlock2 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            System.out.println(a[10]);
        }
        catch(ArithmeticException e)
        {
            System.out.println(
"Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}

```

Output:

ArrayIndexOutOfBoundsException Exception occurs
rest of the code

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

```

public class MultipleCatchBlock3 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
            System.out.println(a[10]);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}

```

Output:

Arithmetic Exception occurs
rest of the code

In this example, we generate `NullPointerException`, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will be invoked.

```
public class MultipleCatchBlock4 {
    public static void main(String[] args) {
        try{
            String s=null;
            System.out.println(s.length());
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}
```

Output:

Parent Exception occurs
rest of the code

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
class MultipleCatchBlock5{
    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(Exception e){System.out.println("common task completed");}
        catch(ArithmeticException e){System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
        System.out.println("rest of the code...");
    }
}
```

Output: Compile-time error

Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithmeticException** (division by zero).

Why use nested try block - Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:


```

.....
//main try block
try
{
    statement 1;
    statement 2;
//try catch block within another try block
    try
    {
        statement 3;
        statement 4;
//try catch block within nested try block
        try
        {
            statement 5;
            statement 6;
        }
        catch(Exception e2)
        {
//exception message
        }
        catch(Exception e1)
        {
//exception message
        }
    }
//catch block of parent (outer) try block
    catch(Exception e3)
    {
//exception message
    }
}
.....

```

Java Nested try Example - Let's see an example where we place a try block within another try block for two different exceptions.

```

public class NestedTryBlock{
    public static void main(String args[]){
//outer try block
        try{
//inner try block 1
            try{
                System.out.println("going to divide by 0");
                int b =39/0;
            }
//catch block of inner try block 1
            catch(ArithmeticException e)
            {
                System.out.println(e);
            }

//inner try block 2
            try{
                int a[]=new int[5];

//assigning the value out of array bounds
                a[5]=4;
            }
//catch block of inner try block 2
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println(e);
            }
            System.out.println("other statement");
        }
    }
}

```

```

    }
    //catch block of outer try block
    catch(Exception e)
    {
        System.out.println("handled the exception (outer catch)");
    }

    System.out.println("normal flow..");
}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..

```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

Example 2 - Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

NestedTryBlock.java

```

public class NestedTryBlock2 {
    public static void main(String args[])
    {
        // outer (main) try block
        try {
            //inner try block 1
            try {
                // inner try block 2
                try {
                    int arr[] = { 1, 2, 3, 4 };
                    //printing the array element out of its bounds
                    System.out.println(arr[10]);
                }
                // to handles ArithmeticException
                catch (ArithmeticException e) {
                    System.out.println("Arithmetic exception");
                    System.out.println(" inner try block 2");
                }
            }

            // to handle ArithmeticException
            catch (ArithmeticException e) {
                System.out.println("Arithmetic exception");
                System.out.println("inner try block 1");
            }
        }

        // to handle ArrayIndexOutOfBoundsException
    }
}

```

```

    catch (ArrayIndexOutOfBoundsException e4) {
        System.out.print(e4);
        System.out.println(" outer (main) try block");
    }
    catch (Exception e5) {
        System.out.print("Exception");
        System.out.println(" handled in main try-block");
    }
}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer
(main) try block

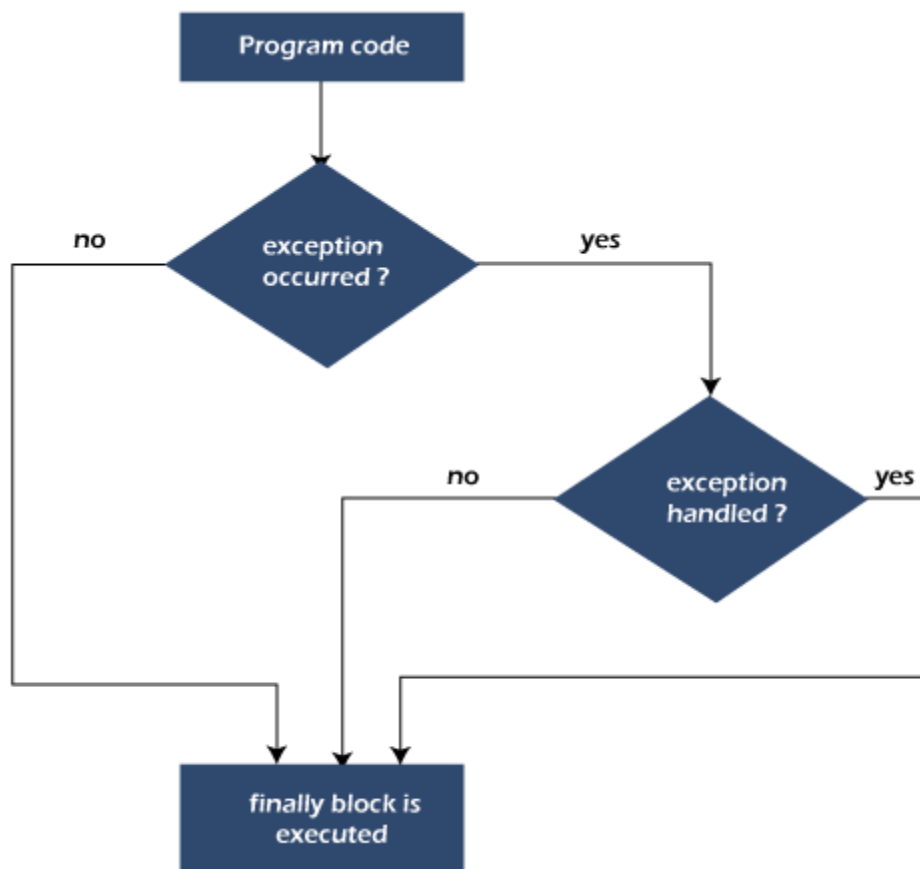
```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not. The finally block follows the try-catch block.

Flowchart of finally block



Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

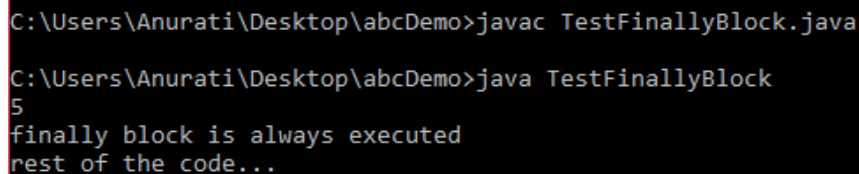
- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally - Let's see the different cases where Java finally block can be used.

Case 1: When an exception does not occur - Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

```
class TestFinallyBlock {
    public static void main(String args[]){
        try{
//below code do not throw any exception
            int data=25/5;
            System.out.println(data);
        }
//catch won't be executed
        catch(NullPointerException e){
            System.out.println(e);
        }
//executed regardless of exception occurred or not
        finally {
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occur but not handled by the catch block - Let's see the following example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

```
public class TestFinallyBlock1{
    public static void main(String args[]){
        try {
            System.out.println("Inside the try block");
            //below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }
//cannot handle Arithmetic type exception
//can only accept Null Pointer type exception
        catch(NullPointerException e){
            System.out.println(e);
        }
//executes regardless of exception occurred or not
        finally {
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Case 3: When an exception occurs and is handled by the catch block

Example: Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

```
public class TestFinallyBlock2{
    public static void main(String args[]){
        try {
            System.out.println("Inside try block");
            //below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }
        //handles the Arithmetic Exception / Divide by zero exception
        catch(ArithmeticException e){
            System.out.println("Exception handled");
            System.out.println(e);
        }
        //executes regardless of exception occurred or not
        finally {
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: /by zero
finally block is always executed
rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword - The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error");
```

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception - In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {
    //function to check if person is eligible to vote or not
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
        }
        else {
            System.out.println("Person is eligible to vote!!");
        }
    }
    //main method
    public static void main(String args[]){
        //calling the function
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
vote
    at TestThrow1.validate(TestThrow1.java:8)
    at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

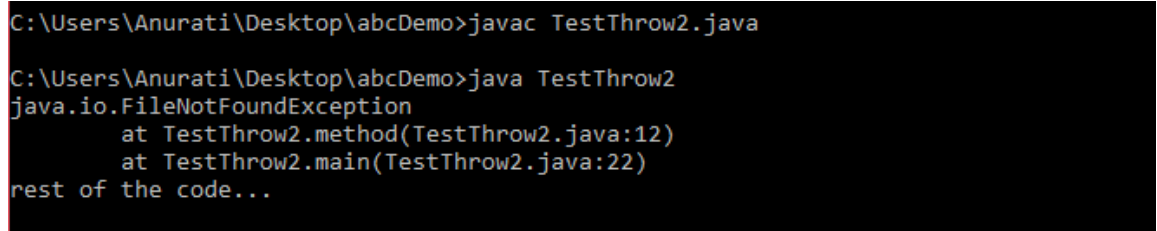
Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Example 2: Throwing Checked Exception - Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

```
import java.io.*;
public class TestThrow2 {
    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {
        FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);
        throw new FileNotFoundException();
    }
    //main method
    public static void main(String args[]){
        try{
            method();
        } catch (FileNotFoundException e){
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
    at TestThrow2.method(TestThrow2.java:12)
    at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

Example 3: Throwing User-defined Exception - exception is everything else under the Throwable class.

```
// class represents user-defined exception
class UserDefinedException extends Exception
{
    public UserDefinedException(String str)
    {
        // Calling constructor of parent Exception
        super(str);
    }
}
// Class that uses above MyException
public class TestThrow3
{
    public static void main(String args[])
    {
        try {
            // throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        } catch (UserDefinedException ude){
            System.out.println("Caught the exception");
            // Print the message from MyException object
            System.out.println(ude.getMessage());
        }
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow3
Caught the exception
This is user-defined exception
```

Java Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

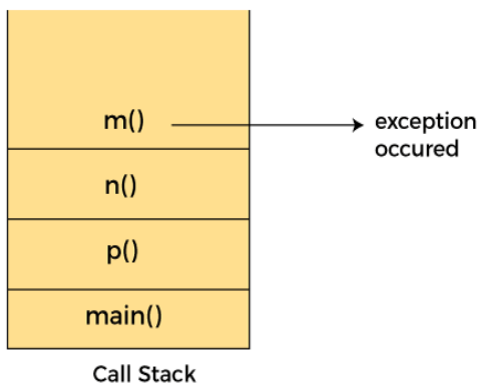
```
class TestExceptionPropagation1{
    void m(){
        int data=50/0;
    }
    void n(){
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        TestExceptionPropagation1 obj=new TestExceptionPropagation1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:

```
exception handled
normal flow...
```

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Exception Propagation Example

```
class TestExceptionPropagation2{
    void m(){
        throw new java.io.IOException("device error");//checked exception
    }
    void n(){
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        TestExceptionPropagation2 obj=new TestExceptionPropagation2();
        obj.p();
        System.out.println("normal flow");
    }
}
```

Output: Compile Time Error

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).
It provides information to the caller of the method about the exception.

Java throws Example - Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
    void m()throws IOException{
        throw new IOException("device error");//checked exception
    }
    void n()throws IOException{
        m();
    }
    void p(){
```

```

    try{
        n();
    }catch(Exception e){System.out.println("exception handled");}
}
public static void main(String args[]){
    Testthrows1 obj=new Testthrows1();
    obj.p();
    System.out.println("normal flow...");
}
}

```

Output:

```

exception handled
normal flow...

```

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

There are two cases:

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block - In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

Testthrows2.java

```

import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
public class Testthrows2{
    public static void main(String args[]){
        try{
            M m=new M();
            m.method();
        }catch(Exception e){System.out.println("exception handled");}

        System.out.println("normal flow...");
    }
}

```

Output:

```

exception handled
    normal flow...

```

Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

Let's see examples for both the scenario.

A) If exception does not occur

```

import java.io.*;
class M{
    void method()throws IOException{
        System.out.println("device operation performed");
    }
}

```

```

    }
}
class Testthrows3{
    public static void main(String args[]){throws IOException{//declare exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}

```

Output:

```

device operation performed
normal flow...

```

B) If exception occurs

```

import java.io.*;
class M{
    void method(){throws IOException{
        throw new IOException("device error");
    }
}
class Testthrows4{
    public static void main(String args[]){throws IOException{//declare exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}

```

Output:

```

Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)

```

Difference between throw and throws

Q) Can we rethrow an exception? Yes, by throwing same exception in catch block.

Difference between throw and throws in Java - The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method. There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	

3. Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4. Declaration	throw is used within the method.	throws is used with the method signature.
5. Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Java throw Example

```
public class TestThrow {
    //defining a method
    public static void checkNum(int num) {
        if (num < 1) {
            throw new ArithmeticException("\nNumber is negative, cannot calculate square");
        }
        else {
            System.out.println("Square of " + num + " is " + (num*num));
        }
    }
    //main method
    public static void main(String[] args) {
        TestThrow obj = new TestThrow();
        obj.checkNum(-3);
        System.out.println("Rest of the code..");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
    at TestThrow.checkNum(TestThrow.java:6)
    at TestThrow.main(TestThrow.java:16)
```

Java throws Example

```
public class TestThrows {
    //defining a method
    public static int divideNum(int m, int n) throws ArithmeticException {
        int div = m / n;
        return div;
    }
    //main method
    public static void main(String[] args) {
        TestThrows obj = new TestThrows();
        try {
            System.out.println(obj.divideNum(45, 0));
        }
        catch (ArithmeticException e){
            System.out.println("\nNumber cannot be divided by 0");
        }

        System.out.println("Rest of the code..");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrows
Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

```
public class TestThrowAndThrows
{
    // defining a user-defined method
    // which throws ArithmeticException
    static void method() throws ArithmeticException
    {
        System.out.println("Inside the method()");
        throw new ArithmeticException("throwing ArithmeticException");
    }
    //main method
    public static void main(String args[])
    {
        try
        {
            method();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught in main() method");
        }
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.

2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.
3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.

Java final Example - Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

```
public class FinalExampleTest {
    //declaring final variable
    final int age = 18;
    void display() {
        // reassigning value to age variable
        // gives compile time error
        age = 55;
    }

    public static void main(String[] args) {
        FinalExampleTest obj = new FinalExampleTest();
        // gives compile time error
        obj.display();
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
    age = 55;
    ^
1 error
```

In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

Java finally Example - Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

```
public class FinallyExample {
    public static void main(String args[]){
        try {
            System.out.println("Inside try block");
            // below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }
        // handles the Arithmetic Exception / Divide by zero exception
        catch (ArithmeticException e){
            System.out.println("Exception handled");
        }
    }
}
```

```

        System.out.println(e);
    }
    // executes regardless of exception occurred or not
    finally {
        System.out.println("finally block is always executed");
    }
    System.out.println("rest of the code...");
}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...

```

Java finalize Example

```

public class FinalizeExample {
    public static void main(String[] args)
    {
        FinalizeExample obj = new FinalizeExample();
        // printing the hashCode
        System.out.println("HashCode is: " + obj.hashCode());
        obj = null;
        // calling the garbage collector using gc()
        System.gc();
        System.out.println("End of the garbage collection");
    }
    // defining the finalize method
    protected void finalize()
    {
        System.out.println("Called the finalize() method");
    }
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java
Note: FinalizeExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample
HashCode is: 746292446
End of the garbage collection
Called the finalize() method

```

Exception Handling with Method Overriding in Java

There are many rules if we talk about method overriding with exception handling.

Some of the rules are listed below:

- **If the superclass method does not declare an exception**
 - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

- **If the superclass method declares an exception**

- If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

Rule 1: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception. Let's consider following example based on the above rule.

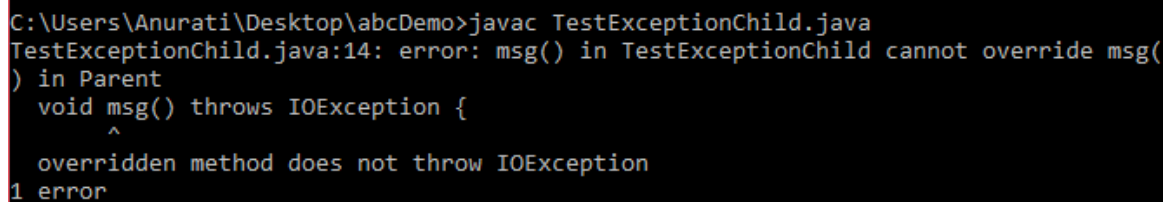
```
import java.io.*;
class Parent{
    // defining the method
    void msg() {
        System.out.println("parent method");
    }
}

public class TestExceptionChild extends Parent{

    // overriding the method in child class
    // gives compile time error
    void msg() throws IOException {
        System.out.println("TestExceptionChild");
    }

    public static void main(String args[]) {
        Parent p = new TestExceptionChild();
        p.msg();
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild.java
TestExceptionChild.java:14: error: msg() in TestExceptionChild cannot override msg(
) in Parent
    void msg() throws IOException {
            ^
    overridden method does not throw IOException
1 error
```

Rule 2: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
import java.io.*;
class Parent{
    void msg() {
        System.out.println("parent method");
    }
}

class TestExceptionChild1 extends Parent{
    void msg()throws ArithmeticException {
        System.out.println("child method");
    }

    public static void main(String args[]) {
        Parent p = new TestExceptionChild1();
        p.msg();
    }
}
```


Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild1.java
C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild1
child method
```

If the superclass method declares an exception

Rule 1: If the superclass method declares an exception, subclass overridden method can declare the same subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
import java.io.*;
class Parent{
    void msg()throws ArithmeticException {
        System.out.println("parent method");
    }
}

public class TestExceptionChild2 extends Parent{
    void msg()throws Exception {
        System.out.println("child method");
    }

    public static void main(String args[]) {
        Parent p = new TestExceptionChild2();

        try {
            p.msg();
        }
        catch (Exception e){}
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild2.java
TestExceptionChild2.java:9: error: msg() in TestExceptionChild2 cannot override msg
() in Parent
    void msg()throws Exception {
        ^
    overridden method does not throw Exception
1 error
```

Example in case subclass overridden method declares same exception

```
import java.io.*;
class Parent{
    void msg() throws Exception {
        System.out.println("parent method");
    }
}

public class TestExceptionChild3 extends Parent {
    void msg()throws Exception {
        System.out.println("child method");
    }

    public static void main(String args[]){
        Parent p = new TestExceptionChild3();
    }
}
```

```

    try {
        p.msg();
    }
    catch(Exception e) {}
}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild3.java

C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild3
child method

```

Example in case subclass overridden method declares subclass exception

```

import java.io.*;
class Parent{
    void msg()throws Exception {
        System.out.println("parent method");
    }
}

class TestExceptionChild4 extends Parent{
    void msg()throws ArithmeticException {
        System.out.println("child method");
    }

    public static void main(String args[]){
        Parent p = new TestExceptionChild4();

        try {
            p.msg();
        }
        catch(Exception e) {}
    }
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild4.java

C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild4
child method

```

Example in case subclass overridden method declares no exception

```

import java.io.*;
class Parent {
    void msg()throws Exception{
        System.out.println("parent method");
    }
}

class TestExceptionChild5 extends Parent{
    void msg() {
        System.out.println("child method");
    }

    public static void main(String args[]){
        Parent p = new TestExceptionChild5();
        try {

```

```

        p.msg();
    }
    catch(Exception e) {}

}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild5.java

C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild5
child method

```

Java Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

In this section, we will learn how custom exceptions are implemented and used in Java programs.

Why use custom exceptions? Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions. Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

```

public class WrongFileNameException extends Exception {
    public WrongFileNameException(String errorMessage) {
        super(errorMessage);
    }
}

```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

Example 1: Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

```

// class representing custom exception
class InvalidAgeException extends Exception
{
    public InvalidAgeException (String str) {
        // calling the constructor of parent Exception
        super(str);
    }
}

// class that uses custom exception InvalidAgeException

```

```

public class TestCustomException1
{
    // method to check the age
    static void validate (int age) throws InvalidAgeException{
        if(age < 18){
            // throw an object of user defined exception
            throw new InvalidAgeException("age is not valid to vote");
        } else {
            System.out.println("welcome to vote");
        }
    }

    // main method
    public static void main(String args[])
    {
        try {
            // calling the method
            validate(13);
        } catch (InvalidAgeException ex) {
            System.out.println("Caught the exception");
            // printing the message from InvalidAgeException object
            System.out.println("Exception occurred: " + ex);
        }
        System.out.println("rest of the code...");
    }
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occurred: InvalidAgeException: age is not valid to vote
rest of the code...

```

Example 2:

```

// class representing custom exception
class MyCustomException extends Exception {}

// class that uses custom exception MyCustomException
public class TestCustomException2
{
    // main method
    public static void main(String args[])
    {
        try {
            // throw an object of user defined exception
            throw new MyCustomException();
        } catch (MyCustomException ex) {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }
        System.out.println("rest of the code...");
    }
}

```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException2.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException2
Caught the exception
null
rest of the code...
```

Multithreaded Programming

Unlike many other computer languages, Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process based and thread-based. It is important to understand the difference between the two. For

most readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost. While Java programs make use of processbased multitasking environments, process-based multitasking is not under the control of

Java. However, multithreaded multitasking is. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.

For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use.

If you have programmed for operating systems such as Windows, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient, because many of the details are handled for you.

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:

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 sub process, 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

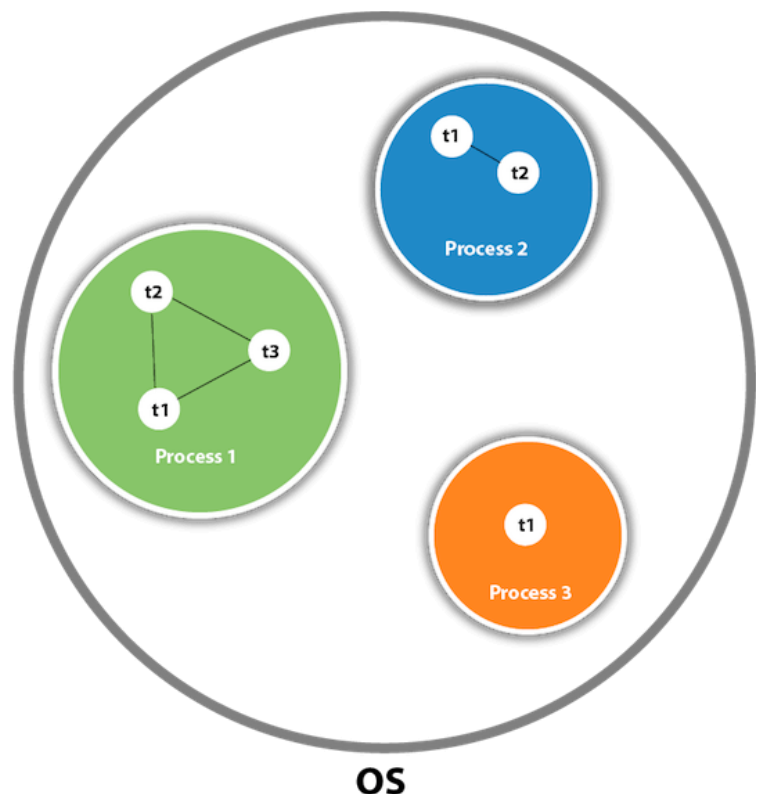
Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.



- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.

A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the `join()` method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

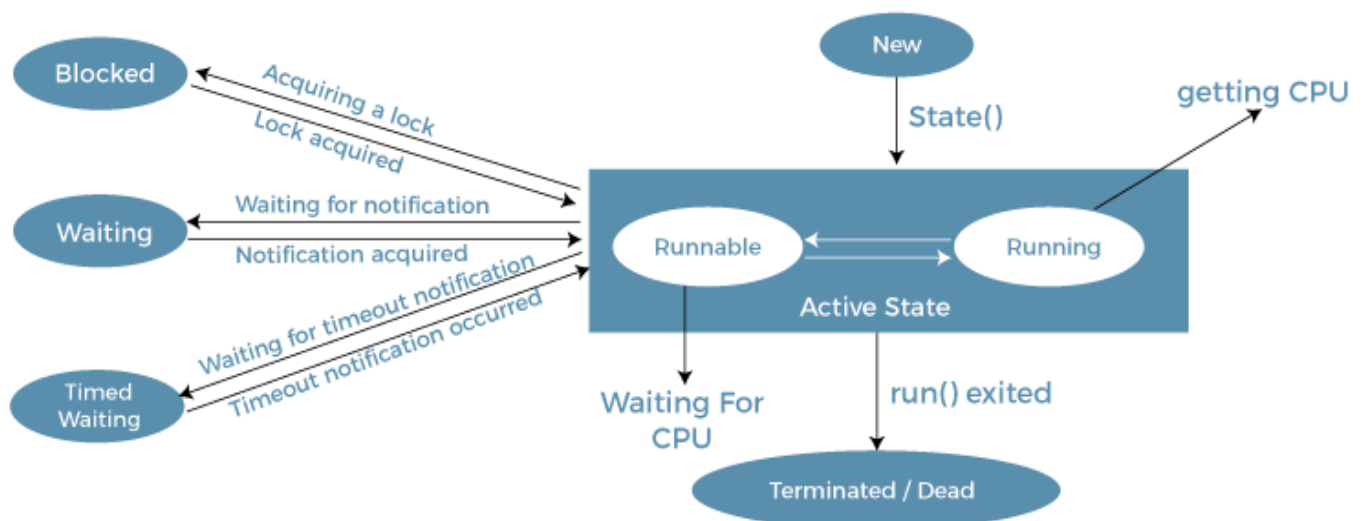
Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the `sleep()` method on a specific thread. The `sleep()` method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.



Life Cycle of a Thread

Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

```
public static final Thread.State NEW
```

It represents the first state of a thread that is the NEW state.

```
public static final Thread.State RUNNABLE
```

It represents the runnable state. It means a thread is waiting in the queue to run.

```
public static final Thread.State BLOCKED
```

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

```
public static final Thread.State WAITING
```

It represents the waiting state. A thread will go to this state when it invokes the `Object.wait()` method, or `Thread.join()` method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

```
public static final Thread.State TIMED_WAITING
```

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

- `sleep`
- `join with timeout`
- `wait with timeout`
- `parkUntil`
- `parkNanos`

```
public static final Thread.State TERMINATED
```

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Java Program for Demonstrating Thread States - The following Java program shows some of the states of a thread defined above.

FileName: ThreadState.java

```
// ABC class implements the interface Runnable
class ABC implements Runnable
{
    public void run()
    {
        // try-catch block
        try
        {
            // moving thread t2 to the state timed waiting
            Thread.sleep(100);
        }
        catch (InterruptedException ie)
        {
            ie.printStackTrace();
        }
        System.out.println("The state of thread t1 while it invoked the method join() on thread t2 - 
        "+ ThreadState.t1.getState());

        // try-catch block
        try
        {
            Thread.sleep(200);
        }
        catch (InterruptedException ie)
        {
            ie.printStackTrace();
        }
    }
}

// ThreadState class implements the interface Runnable
public class ThreadState implements Runnable
{
    public static Thread t1;
    public static ThreadState obj;

    // main method
    public static void main(String args[])
    {
        // creating an object of the class ThreadState
        obj = new ThreadState();
        t1 = new Thread(obj);

        // thread t1 is spawned
        // The thread t1 is currently in the NEW state.
        System.out.println("The state of thread t1 after spawning it - " + t1.getState());

        // invoking the start() method on
        // the thread t1
        t1.start();

        // thread t1 is moved to the Runnable state
        System.out.println("The state of thread t1 after invoking the method start() on it - 
        " + t1.getState());
    }

    public void run()
    {
        ABC myObj = new ABC();
        Thread t2 = new Thread(myObj);

        // thread t2 is created and is currently in the NEW state.
        System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
    }
}
```

```

t2.start();

// thread t2 is moved to the runnable state
System.out.println("the state of thread t2 after calling the method start() on it -
" + t2.getState());

// try-catch block for the smooth flow of the program
try
{
// moving the thread t1 to the state timed waiting
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}

System.out.println("The state of thread t2 after invoking the method sleep() on it -
"+ t2.getState() );

// try-catch block for the smooth flow of the program
try
{
// waiting for thread t2 to complete its execution
t2.join();
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
System.out.println("The state of thread t2 when it has completed it's execution -
" + t2.getState());
}

}

```

Output:

```

The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it - RUNNABLE
The state of thread t2 after spawning it - NEW
the state of thread t2 after calling the method start() on it - RUNNABLE
The state of thread t1 while it invoked the method join() on thread t2 -TIMED_WAITING
The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING
The state of thread t2 when it has completed it's execution - TERMINATED

```

Explanation: Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 get to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish it's execution as it has invoked the method join() on thread t2.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
class CurrentThreadDemo{
    public static void main(String args[]){
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try{
            for(int n = 5; n > 0; n--){
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e){
            System.out.println("Main thread interrupted");
        }
    }
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling `currentThread()`, and this reference is stored in the local variable `t`. Next, the program displays information about the thread. The program then calls `setName()` to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the `sleep()` method. The argument to `sleep()` specifies the delay period in milliseconds.

Notice the try/catch block around this loop. The `sleep()` method in `Thread` might throw an `InterruptedException`. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted.

Java Threads | How to create a thread in Java

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:

- `Thread()`
- `Thread(String name)`
- `Thread(Runnable r)`
- `Thread(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 milliseconds):** 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 milliseconds):** waits for a thread to die for the specified milliseconds.
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(deprecated).
16. **public void resume():** is used to resume the suspended thread(deprecated).
17. **public void stop():** is used to stop the thread(deprecated).
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: The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

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

1) Java Thread Example by extending Thread class

FileName: Multi.java

```
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();
}
}
```

Output: thread is running...

Another Example:

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId() + " is running");
        }
        catch (Exception e) {
```

```

        // Throwing an exception
        System.out.println("Exception is caught");
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}

```

Output

```

Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running

```

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```

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); // Using the constructor Thread(Runnable r)
        t1.start();
    }
}

```

Output: thread is running...

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

Another Example:

```

// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
    }
}

```

```

        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}

```

Output

```

Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running

```

3) Using the Thread Class: Thread(String Name) - We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```

public class MyThread1
{
    // Main method
    public static void main(String args[])
    {
        // creating an object of the Thread class using the constructor Thread(String name)
        Thread t= new Thread("My first thread");

        // the start() method moves the thread to the active state
        t.start();
        // getting the thread name by invoking the getName() method
        String str = t.getName();
        System.out.println(str);
    }
}

```

Output: My first thread

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

FileName: MyThread2.java

```

public class MyThread2 implements Runnable
{
    public void run()
    {
        System.out.println("Now the thread is running ...");
    }
}

```

```

}

// main method
public static void main(String argsv[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();

// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}

```

Output:

```

My new thread
Now the thread is running ...

```

Thread Class vs. Runnable Interface

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread Scheduler Algorithms

On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

First Come First Serve Scheduling: In this scheduling algorithm, the scheduler picks the threads that arrive first in the runnable queue. Observe the following table:

Threads	Time of Arrival
t1	0
t2	1
t3	2

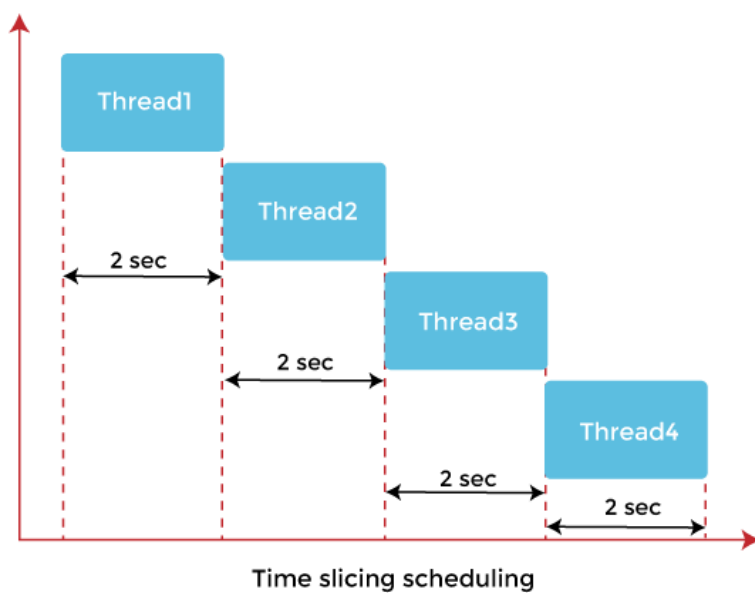
In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.



First Come First Serve Scheduling

Hence, Thread t1 will be processed first, and Thread t4 will be processed last.

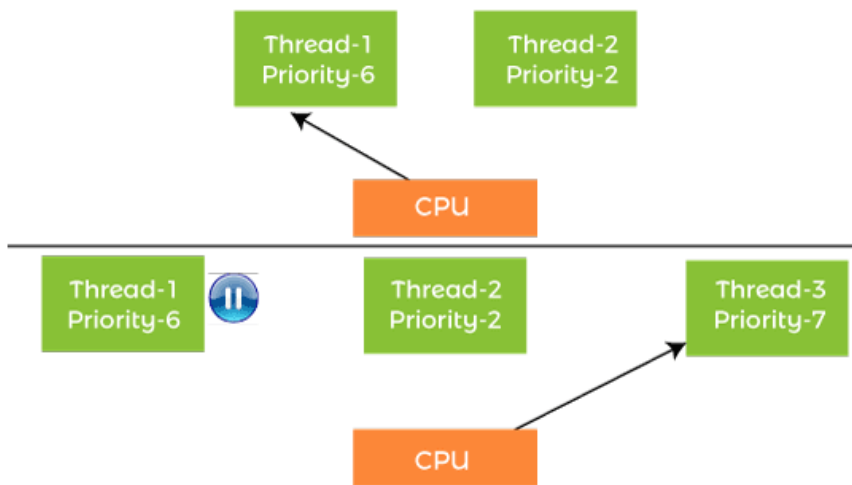
Time-slicing scheduling: Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, sometime-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.



Time slicing scheduling

In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

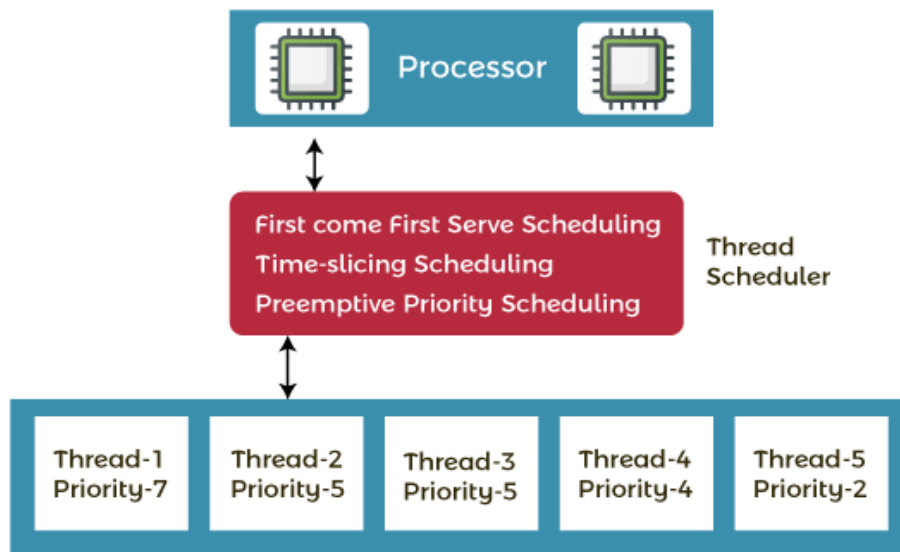
Preemptive-Priority Scheduling: The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



Preemptive-Priority Scheduling

Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

Working of the Java Thread Scheduler



Working of Thread Scheduler

Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time.

The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

Following are the syntax of the sleep() method.

```
public static void sleep(long mls) throws InterruptedException
public static void sleep(long mls, int n) throws InterruptedException
```

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters: The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Example of the sleep() method in Java : on the custom thread - The following example shows how one can use the sleep() method on the custom thread.

FileName: TestSleepMethod1.java

```
class TestSleepMethod1 extends Thread{
    public void run(){
        for(int i=1;i<5;i++){
            // the thread will sleep for the 500 milli seconds
            try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[]){
        TestSleepMethod1 t1=new TestSleepMethod1();
        TestSleepMethod1 t2=new TestSleepMethod1();

        t1.start();
        t2.start();
    }
}
```

Output:

1
1
2
2
3
3
4
4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Example of the sleep() Method in Java : on the main thread

FileName: TestSleepMethod2.java

```
// important import statements
import java.lang.Thread;
import java.io.*;
public class TestSleepMethod2
{
    // main method
    public static void main(String argsv[])
    {
        try {
            for (int j = 0; j < 5; j++)
            {

                // The main thread sleeps for the 1000 milliseconds, which is 1 sec
                // whenever the loop runs
                Thread.sleep(1000);
                // displaying the value of the variable
                System.out.println(j);
            }
        } catch (Exception expn)
        {
            // catching the exception
            System.out.println(expn);
        }
    }
}
```

Output:

0
1
2
3
4

Example of the sleep() Method in Java: When the sleeping time is -ive - The following example throws the exception `IllegalArgumentExpection` when the time for sleeping is negative.

FileName: TestSleepMethod3.java

```
// important import statements
import java.lang.Thread;
import java.io.*;

public class TestSleepMethod3
{
```

```
// main method
public static void main(String argsv[])
{
// we can also use throws keyword followed by
// exception name for throwing the exception
try
{
for (int j = 0; j < 5; j++)
{

// it throws the exception IllegalArgumentException
// as the time is -ive which is -100
Thread.sleep(-100);

// displaying the variable's value
System.out.println(j);
}
}
catch (Exception expn)
{

// the exception is caught here
System.out.println(expn);
}
}
}
```

Output: java.lang.IllegalArgumentException: timeout value is negative

Can we start a thread twice

No. After starting a thread, it can never be started again. If you do so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception. Let's understand it by the example given below:

```
public class TestThreadTwice1 extends Thread{
    public void run(){
        System.out.println("running...");
    }
    public static void main(String args[]){
        TestThreadTwice1 t1=new TestThreadTwice1();
        t1.start();
        t1.start();
    }
}
```

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

What if we call Java run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

FileName: TestCallRun1.java

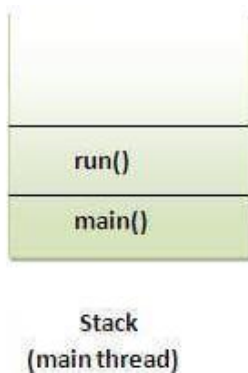
```
class TestCallRun1 extends Thread{
    public void run(){
        System.out.println("running...");
    }
    public static void main(String args[]){
        TestCallRun1 t1=new TestCallRun1();
```

```

    t1.run();//fine, but does not start a separate call stack
}
}

```

Output: running...



Problem if you direct call run() method

FileName: TestCallRun2.java

```

class TestCallRun2 extends Thread{
    public void run(){
        for(int i=1;i<5;i++){
            try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[]){
        TestCallRun2 t1=new TestCallRun2();
        TestCallRun2 t2=new TestCallRun2();

        t1.run();
        t2.run();
    }
}

```

Output:

```

1
2
3
4
1
2
3
4

```

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

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

1. Synchronized method.
2. Synchronized block.
3. 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.

Understanding the problem without Synchronization - In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
class Table{
void printTable(int n){//method not synchronized
    for(int i=1;i<=5;i++){
        System.out.println(n*i);
        try{
            Thread.sleep(400);
        }catch(Exception e){System.out.println(e);}
    }
}

class MyThread1 extends Thread{
    Table t;
    MyThread1(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(5);
    }
}

class MyThread2 extends Thread{
    Table t;
```

```

MyThread2(Table t){
    this.t=t;
}
public void run(){
    t.printTable(100);
}
}

class TestSynchronization1{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}

```

Output:

```

5
100
10
200
15
300
20
400
25
500

```

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```

//example of java synchronized method
class Table{
    synchronized void printTable(int n){//synchronized method
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }
    }
}

class MyThread1 extends Thread{
    Table t;
    MyThread1(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(5);
    }
}

class MyThread2 extends Thread{
    Table t;
    MyThread2(Table t){
        this.t=t;
    }
}

```

```

}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}

```

Output:

```

5
10
15
20
25
100
200
300
400
500

```

Example of synchronized method by using anonymous class - In this program, we have created the two threads by using the anonymous class, so less coding is required.

TestSynchronization3.java

```

//Program of synchronized method by using anonymous class
class Table{
synchronized void printTable(int n){//synchronized method
for(int i=1;i<=5;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
}
}
}

```

```

public class TestSynchronization3{
public static void main(String args[]){
final Table obj = new Table();//only one object

Thread t1=new Thread(){
public void run(){
obj.printTable(5);
}
};
Thread t2=new Thread(){
public void run(){
obj.printTable(100);
}
};

t1.start();
t2.start();
}
}

```


Output:

5
10
15
20
25
100
200
300
400
500

Java Applet Basics

Throwback of making GUI application: Java was launched on 23-Jan-1996(JDK 1.0) and at that time it only supported CUI(Character User Interface) application. But in 1996 VB(Visual Basic) of Microsoft was preferred for GUI programming. So the Java developers in hurry(i.e. within 7 days) have given the support for GUI from Operating System(OS). Now, the components like button, etc. were platform-dependent(i.e. in each platform there will be different size, shape button). But they did the intersection of such components from all platforms and gave a small library which contains these intersections and it is available in AWT(Abstract Window Toolkit) technology but it doesn't have advanced features like dialogue box, etc.

Now to run Applet, java needs a browser and at that time only "Internet Explorer" was there of Microsoft but Microsoft believes in monopoly. So "SUN Micro-System"(the company which developed Java) contracted with other company known as "Netscape"(which developed Java Script) and now the "Netscape" company is also known as "Mozilla Firefox" which we all know is a browser. Now, these two companies have developed a technology called "SWING" and the benefit is that the SWING components are produced by Java itself. Therefore now it is platform-independent as well as some additional features have also been added which were not in AWT technology. So we can say that SWING is much more advanced as compared to AWT technology.

What is Applet? An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server. Applets are used to make the website more dynamic and entertaining.

Important points :

1. All applets are sub-classes (either directly or indirectly) of *java.applet.Applet* class.
2. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
3. In general, execution of an applet does not begin at *main()* method.
4. Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.

Life cycle of an applet :

It is important to understand the order in which the various methods shown in the above image are called. When an applet begins, the following methods are called, in this sequence:

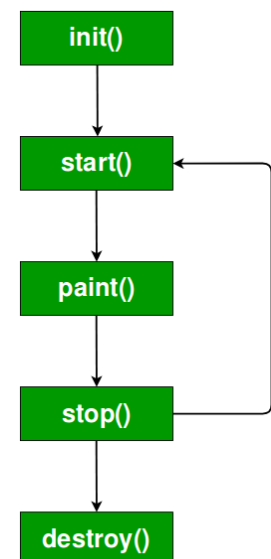
1. *init()*
2. *start()*
3. *paint()*

When an applet is terminated, the following sequence of method calls takes place:

1. *stop()*
2. *destroy()*

Let's look more closely at these methods.

1. *init()* : The *init()* method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.



2. start() : The **start()** method is called after **init()**. It is also called to restart an applet after it has been stopped. Note that **init()** is called once i.e. when the first time an applet is loaded whereas **start()** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start()**.

3. paint() : The **paint()** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint()** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint()** is called.

The **paint()** method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required. Note: This is the only method among all the method mention above, which is parameterized. It's prototype is `public void paint(Graphics g)` where g is an object reference of class Graphic.

Now the **Question Arises:**

Q. In the prototype of `paint()` method, we have created an object reference without creating its object. But how is it possible to create object reference without creating its object?

Ans. Whenever we pass object reference in arguments then the object will be provided by its caller itself. In this case the caller of `paint()` method is browser, so it will provide an object. The same thing happens when we create a very basic program in normal Java programs. For Example:

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

Here we have created an object reference without creating its object but it still runs because it's caller,i.e JVM will provide it with an object.

4. stop() : The **stop()** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop()** is called, the applet is probably running. You should use **stop()** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start()** is called if the user returns to the page.

5. destroy() : The **destroy()** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop()** method is always called before **destroy()**.

How to run an Applet?

There are two ways to run an applet

1. By html file.
2. By appletViewer tool (for testing purpose).

Simple example of Applet by html file: - To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome",150,150);
}
}
```

Note: class must be public because its object is created by Java Plugin software that resides on the browser.

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}
}
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
```

```
c:\>appletviewer First.java
```

Features of Applets over HTML

- Displaying dynamic web pages of a web application.
- Playing sound files.
- Displaying documents
- Playing animations

Restrictions imposed on Java applets - Due to security reasons, the following restrictions are imposed on Java applets:

1. An applet cannot load libraries or define native methods.
2. An applet cannot ordinarily read or write files on the execution host.
3. An applet cannot read certain system properties.
4. An applet cannot make network connections except to the host that it came from.
5. An applet cannot start any program on the host that's executing it.

Displaying Graphics in Applet

java.awt.Graphics class provides many methods for graphics programming. Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).

7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of Graphics in applet:

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet{

    public void paint(Graphics g){
        g.setColor(Color.red);
        g.drawString("Welcome",50, 50);
        g.drawLine(20,30,20,300);
        g.drawRect(70,100,30,30);
        g.fillRect(170,100,30,30);
        g.drawOval(70,200,30,30);

        g.setColor(Color.pink);
        g.fillOval(170,200,30,30);
        g.drawArc(90,150,30,30,30,270);
        g.fillArc(270,150,30,30,0,180);

    }
}
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

Displaying Image in Applet

Applet is mostly used in games and animation. For this purpose image is required to be displayed. The java.awt.Graphics class provide a method drawImage() to display the image.

Syntax of drawImage() method:

public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer): is used draw the specified image.

How to get the object of Image:

The java.applet.Applet class provides getImage() method that returns the object of Image. Syntax:

```
public Image getImage(URL u, String image){}
```

Other required methods of Applet class to display image:

1. **public URL getDocumentBase():** is used to return the URL of the document in which applet is embedded.
2. **public URL getCodeBase():** is used to return the base URL.

Example of displaying image in applet:

```
import java.awt.*;
import java.applet.*;
public class DisplayImage extends Applet {

    Image picture;
    public void init() {
        picture = getImage(getDocumentBase(),"sonoo.jpg");
    }

    public void paint(Graphics g) {
        g.drawImage(picture, 30,30, this);
    }

}
```

In the above example, drawImage() method of Graphics class is used to display the image. The 4th argument of drawImage() method of is ImageObserver object. The Component class implements ImageObserver interface. So current class object would also be treated as ImageObserver because Applet class indirectly extends the Component class.

myapplet.html

```
<html>
<body>
<applet code="DisplayImage.class" width="300" height="300">
</applet>
</body>
</html>
```

Animation in Applet

Applet is mostly used in games and animation. For this purpose image is required to be moved.

```
import java.awt.*;
import java.applet.*;
public class AnimationExample extends Applet {
    Image picture;
    public void init() {
        picture =getImage(getDocumentBase(),"bike_1.gif");
    }

    public void paint(Graphics g) {
        for(int i=0;i<500;i++){
            g.drawImage(picture, i,30, this);

            try{Thread.sleep(100);}catch(Exception e){}
        }
    }
}
```

In the above example, drawImage() method of Graphics class is used to display the image. The 4th argument of drawImage() method of object. The Component class implements ImageObserver interface. So current class object would also be treated as ImageObserver because indirectly extends the Component class.

myapplet.html

```
<html>
<body>
<applet code="DisplayImage.class" width="300" height="300">
</applet>
</body>
</html>
```

EventHandling in Applet

As we perform event handling in AWT or Swing, we can perform it in applet also. Let's see the simple example of event handling in applet that can be performed by clicking on the button.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class EventApplet extends Applet implements ActionListener{
    Button b;
    TextField tf;

    public void init(){
        tf=new TextField();
        tf.setBounds(30,40,150,20);

        b=new Button("Click");
        b.setBounds(80,150,60,50);

        add(b);add(tf);
        b.addActionListener(this);

        setLayout(null);
    }

    public void actionPerformed(ActionEvent e){
        tf.setText("Welcome");
    }
}
```

In the above example, we have created all the controls in init() method because it is invoked only once.

myapplet.html

```
<html>
<body>
<applet code="EventApplet.class" width="300" height="300">
</applet>
</body>
</html>
```

JApplet class in Applet

As we prefer Swing to AWT. Now we can use JApplet that can have all the controls of swing. The JApplet class extends the Applet class.

```
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener{
    JButton b;
    JTextField tf;
    public void init(){

        tf=new JTextField();
        tf.setBounds(30,40,150,20);

        b=new JButton("Click");
        b.setBounds(80,150,70,40);

        add(b);add(tf);
        b.addActionListener(this);

        setLayout(null);
    }
}
```

```

public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
}

```

In the above example, we have created all the controls in init() method because it is invoked only once.

myapplet.html

```

<html>
<body>
<applet code="EventJApplet.class" width="300" height="300">
</applet>
</body>
</html>

```

Painting in Applet

We can perform painting operation in applet by the mouseDragged() method of MouseMotionListener.

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class MouseDrag extends Applet implements MouseMotionListener{
public void init(){
addMouseMotionListener(this);
setBackground(Color.red);
}

public void mouseDragged(MouseEvent me){
Graphics g=getGraphics();
g.setColor(Color.white);
g.fillOval(me.getX(),me.getY(),5,5);
}
public void mouseMoved(MouseEvent me){}

}

```

In the above example, getX() and getY() method of MouseEvent is used to get the current x-axis and y-axis. The getGraphics() method returns the object of Graphics.

myapplet.html

```

<html>
<body>
<applet code="MouseDrag.class" width="300" height="300">
</applet>
</body>
</html>

```

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

```

public String getParameter(String parameterName)

```

Example of using parameter in Applet:

```

import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet{

```

```

public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}

}

```

myapplet.html

```

<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>

```

Difference between a Java Application and a Java Applet

Java Application: Java Application is just like a Java program that runs on an underlying operating system with the support of a virtual machine. It is also known as an **application program**. The graphical user interface is not necessary to execute the java applications, it can be run with or without it.

Java Applet: An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the **APPLET** or **OBJECT** tag and hosted on a web server. Applets are used to make the web site more dynamic and entertaining.

Java Application	Java Applet
Applications are just like a Java programs that can be execute independently without using the web browser.	Applets are small Java programs that are designed to be included with the HTML web document. They require a Java-enabled web browser for execution.
Application program requires a main function for its execution.	Applet does not require a main function for its execution.
Java application programs have the full access to the local file system and network.	Applets don't have local disk and network access.
Applications can access all kinds of resources available on the system.	Applets can only access the browser specific services. They don't have access to the local system.
Applications can executes the programs from the local system.	Applets cannot execute programs from the local machine.
An application program is needed to perform some task directly for the user.	An applet program is needed to perform small tasks or the part of it.