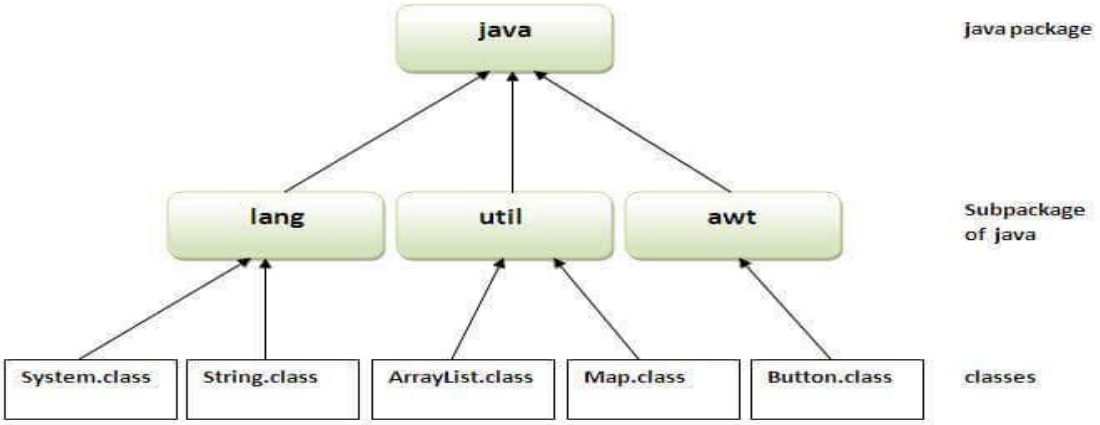
Packages

Introduction to Packages

One of the main features of the Object Oriented Programming language is the ability to *reuse the code* that is already created. One way for achieving this is by extending the classes and implementing the interfaces. Java provides a mechanism to partition the classes into smaller *chunks*. This mechanism is the *Package*. The *Package is container of classes*. The class is the container of the data and code. The package also addresses the problem of name *space collision* that occurs when we use same name for multiple classes. Java provides convenient way to keep the same name for classes as long as their subdirectories are different.

Benefits of packages:

1. The classes in the packages can be easily reused.
2. In the packages, classes are unique compared with other classes in the other packages.
3. Packages provide a way to hide classes thus prevent classes from accessing by other programs.
4. Packages also provide way to separate "design" from "coding".



Creating a Package

While creating a package, you should choose a name for the package and include a package statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

javac -d Destination_folder file_name.java

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

```
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.

1. import package.*;
2. import package.classname;
3. 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

Finding the Packages and CLASSPATH

Packages are mirrored by the directories. This raises an import question: how does Java-Run time system look for the packages that you create? The answer has three parts: (1) By default, Java- Run time system uses the current working directory as its starting point. Thus your package is in a subdirectory of your directory, it will be found. (2) You can specify a directory path by setting the CLASSPATH environment variable. (3) You can use - classpath option with java and iavac to specify the path for the classes.

A Short Example Package: Calc.java

Write a JAVA program that import and use the defined your package in the previous Problem .

```
package Arith; //package name
public class Calc
{
        public float add(float x,float y)
        {
                return (x+y);
        }
        public float sub(float x,float y)
        {
                return (x-y);
        }
        public float mul(float x,float y)
        {
                return (x*y);
        }
        public float div(float x,float y)
        {
                return (x/y);
        }
        public float rem(float x,float y)
        {
                return (x%y);
        }
}
```

Using the package in another class called “MathTest.java”

```
import Arith.*;          //accessing the package
class MathTest
{
public static void main(String args[])
{
                Calc c=new Calc();
                System.out.println("The sum is:"+c.add(12,3));
                System.out.println("The subtraction is is:"+c.sub(12,3));
                System.out.println("The product is:"+c.mul(12,3));
                System.out.println("The Division is:"+c.div(12,3));
                System.out.println("The Rem is:"+c.rem(12,5));
  }
  }
```

Package and Member Accessing

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

There are four types of Java access modifiers:

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

Class Member Access:

Sl No	Class member	Private Member	Default Member	Protected Member	Public Member
1	Visible within same class	YES	YES	YES	Yes
2	Visible within the same package by subclasses	No	YES	YES	YES
3	Visible within same package by non-subclass	No	YES	YES	YES
4	Visible within different packages by subclasses	NO	NO	YES	YES
5	Visible within different packages by Non-subclasses	NO	NO	NO	YES

1) **Private**
The private access modifier is accessible only within the class.

Simple example of private access modifier

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

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

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

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

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

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

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

Example of default access modifier

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

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

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

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

3**) Protected**

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

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

It provides more accessibility than the default modifer.

Example of protected access modifier

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

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

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

Output:Hello

4**) Public**

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

Example of public access modifier

```
//save by A.java
```

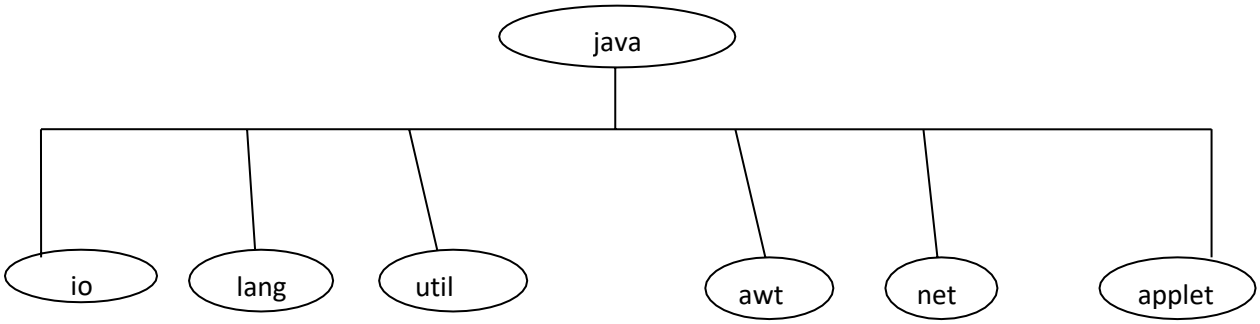
```
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

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

Output:Hello

Categories of Packages

The Java Packages are categorized into two categories (i) Java API Packages (ii) User Defined Packages.

Java API Packages –Java API provides large number of classes grouped into different packages according to the functionality. Most of the time we use the package available with Java API.



Sl No	Package Name	Contents
1	java.lang	Contains language support classes. The java compiler automatically uses this package. This includes the classes such as primitive data types, String, StringBuffer, StringBuilde etc;
2	java.util	Contains the language utility classes such asa Vectors, Hash Table , Date, StringTokenizer etc;
3	java.io	Contains the classes that support input and output classes.
4	java.awt	Contains the classes for implementing the graphical user interfaces
5	Java.net	Contains the classes for networking
6	Java.applet	Contains the classes for creating and implementing the applets.

Java.lang Package

The java.lang package includes the following classes.

Boolean	Enum	Package	StackTraceElement	ThreadGroup
Byte	Float	Process	StrictMath	ThreadLocal
Character	Integer	ProcessBuilder	String	Throwable
Class	Long	Runtime	StringBuffer	Void
ClassLoader	Math	RuntimePermission	StringBuilder	
Compiler	Number	SecurityManager	System	
Double	Object	Short	Thread	

Using the System Packages

Packages are organized in hierarchical structure, that means a package can contain another package, which in turn contains several classes for performing different tasks. There are two ways to access the classes of the packages.
fully qualified class name- this is done specifying package name containing the class and appending the class to it with dot (.) operator.
Example: java.util.StringTokenizer(); Here, "java.util" is the package and "StringTokenizer()" is the class in that package. This is used when we want to refer to only one class in a package.

import statement –this is used when we want to use a class or many classes in many places in our program. Example: (1) import java.util.*;
(2) import java.util.StringTokenizer;

Naming Conventions

Packages can be named using the Java Standard naming rules. Packages begin with "lower case" letters. It is easy for the user to distinguish it from the class names. All the class Name by convention begin with "upper case" letters. Example:

```
double d= java.lang.Math.sqrt(3);
```

Here, "java.lang" is the package, and "Math" is the class name, and "sqrt()" is the method name.

User Define Packages

To create a package is quite easy: simply include a package command in the first line of the source file. A class specified in that file belongs to that package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put in the default package, which has no name.
The general form of the package statement will be as followed: package pkg;
Here, "pkg" is the package name. Example: package MyPackage;

Java Uses file system directories to store packages. For example, the " .class" files for any classes you declare to be part of the "MyPackage" must be store in the "MyPackage" directory. The directory name "MyPackage" is different from "mypackage".
Notes: 1. The case for the package name is significant.
The directory name must match with package name.

We can create hierarchy of packages. To do so, simply separate package name from the above one with it by use of the dot (.) operator. The general form of multileveled package statement is shown here: package pkg1.pkg2.pkg3;

Object class in Java

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Using Object class methods

There are methods in Object class:

toString() : toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, it is defined as:

```
public String toString()
{
   return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

It is always recommended to override toString() method to get our own String representation of Object. For more on override of toString() method refer – Overriding toString() in Java

hashCode() : For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It convert the internal address of object to an integer by using an algorithm. The hashCode() method is native because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.

Use of hashCode() method : Returns a hash value that is used to search object in a collection. JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc. The main advantage of saving objects based on hash code is that searching becomes easy.

```
// Java program to demonstrate working of
// hasCode() and toString()
public class Student
{
   static int last_roll = 100;
   int roll_no;

   // Constructor
   Student()
   {
      roll_no = last_roll;
      last_roll++;
   }

   // Overriding hashCode()
```

```
    @Override
    public int hashCode()
    {
       return roll_no;
    }

    // Driver code
    public static void main(String args[])
    {
       Student s = new Student();

       // Below two statements are equivalent
       System.out.println(s);
       System.out.println(s.toString());
    }
}
```

Output :
Student@64
Student@64
Note that $4*16^0 + 6*16^1 = 100$
equals(Object obj) : Compares the given object to "this" object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override equals(Object obj) method to get our own equality condition on Objects. For more on override of equals(Object obj) method refer – Overriding equals method in Java
Note : It is generally necessary to override the hashCode() method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.
getClass() : Returns the class object of "this" object and used to get actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don't override it.

```
// Java program to demonstrate working of getClass()
public class Test
{
   public static void main(String[] args)
   {
      Object obj = new String("GeeksForGeeks");
      Class c = obj.getClass();
      System.out.println("Class of Object obj is : "
                         + c.getName());
   }
}
```

Output:
Class of Object obj is : java.lang.String
Note :After loading a .class file, JVM will create an object of the type *java.lang.Class* in the Heap area. We can use this class object to get Class level information. It is widely used in Reflection

finalize() method : This method is called just before an object is garbage collected. It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.
Note :finalize method is called just once on an object even though that object is eligible for garbage collection multiple times.

```
// Java program to demonstrate working of finalize()
public class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.hashCode());

        t = null;

        // calling garbage collector
        System.gc();

        System.out.println("end");
    }

    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

Output:
366712642
finalize method called
end
clone() : It returns a new object that is exactly the same as this object. For clone() method refer Clone()

Java Enums

The Enum in Java is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in

capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more powerful. Here, we can define an enum either inside the class or outside the class.

Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

1. Points to remember for Java Enum
2. Enum improves type safety
3. Enum can be easily used in switch
4. Enum can be traversed
5. Enum can have fields, constructors and methods
6. Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple Example of Java Enum

```
class EnumExample1{
//defining the enum inside the class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//main method
public static void main(String[] args) {
//traversing the enum
for (Season s : Season.values())
System.out.println(s);

}

}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
```

Java Math class

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Example 1

```
public class JavaMathExample1
{
   public static void main(String[] args)
   {
      double x = 28;
      double y = 4;

      // return the maximum of two numbers
      System.out.println("Maximum number of x and y is: " +Math.max(x, y));

      // return the square root of y
      System.out.println("Square root of y is: " + Math.sqrt(y));

      //returns 28 power of 4 i.e. 28*28*28*28
      System.out.println("Power of x and y is: " + Math.pow(x, y));

      // return the logarithm of given value
      System.out.println("Logarithm of x is: " + Math.log(x));
      System.out.println("Logarithm of y is: " + Math.log(y));

      // return the logarithm of given value when base is 10
      System.out.println("log10 of x is: " + Math.log10(x));
      System.out.println("log10 of y is: " + Math.log10(y));

      // return the log of x + 1
      System.out.println("log1p of x is: " +Math.log1p(x));

      // return a power of 2
      System.out.println("exp of a is: " +Math.exp(x));

      // return (a power of 2)-1
      System.out.println("expm1 of a is: " +Math.expm1(x));
   }
```

```
}
```

Output:

```
Maximum number of x and y is: 28.0
Square root of y is: 2.0
Power of x and y is: 614656.0
Logarithm of x is: 3.332204510175204
Logarithm of y is: 1.3862943611198906
log10 of x is: 1.4471580313422192
log10 of y is: 0.6020599913279624
log1p of x is: 3.367295829986474
exp of a is: 1.446257064291475E12
expm1 of a is: 1.446257064290475E12
```

Wrapper classes in Java

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java
Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
Synchronization: Java synchronization works with objects in Multithreading.
java.util package: The java.util package provides the utility classes to deal with objects.
Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.
The eight classes of the java.lang package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer

long Long
float Float
double Double

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

20 20 20

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

3 3 3

Java.util Package in Java

It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Following are the Important Classes in Java.util package :

1. AbstractCollection: This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.
2. AbstractList: This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
3. AbstractMap<K,V>: This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.
4. AbstractMap.SimpleEntry<K,V>: An Entry maintaining a key and a value.
5. AbstractMap.SimpleImmutableEntry<K,V>: An Entry maintaining an immutable key and value.
6. AbstractQueue: This class provides skeletal implementations of some Queue operations.
7. AbstractSequentialList: This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
8. AbstractSet: This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.
9. ArrayDeque: Resizable-array implementation of the Deque interface.
10. ArrayList: Resizable-array implementation of the List interface.
11. Arrays: This class contains various methods for manipulating arrays (such as sorting and searching).
12. BitSet: This class implements a vector of bits that grows as needed.
13. Calendar: The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.
14. Collections: This class consists exclusively of static methods that operate on or return collections.
15. Currency: Represents a currency.
16. Date: The class Date represents a specific instant in time, with millisecond precision.
17. Dictionary<K,V>: The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
18. EnumMap,V>: A specialized Map implementation for use with enum type keys.
19. EnumSet: A specialized Set implementation for use with enum types.
20. EventListenerProxy: An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.

Formatter Class

The Formatter is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming. The Formatter class is defined as final class inside the java. util package.

Introduction

The **java.util.Formatter** class provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.Following are the important points about Formatter −

- Formatters are not necessarily safe for multithreaded access.Thread safety is optional and is the responsibility of users of methods in this class.

Class declaration

Following is the declaration for **java.util.Formatter** class −

```
public final class Formatter
  extends Object
  implements Closeable, Flushable
```

Class constructors

Sr.No.	Constructor & Description
1	**Formatter()** This constructor constructs a new formatter.
2	**Formatter(Appendable a)** This constructor constructs a new formatter with the specified destination.
3	**Formatter(Appendable a, Locale l)** This constructor constructs a new formatter with the specified destination and locale.
4	**Formatter(File file)**

	This constructor constructs a new formatter with the specified file.
5	**Formatter(File file, String csn)** This constructor constructs a new formatter with the specified file and charset.
6	**Formatter(File file, String csn, Locale l)** This constructor constructs a new formatter with the specified file, charset, and locale.
7	**Formatter(Locale l)** This constructor constructs a new formatter with the specified locale.
8	**Formatter(OutputStream os)** This constructor constructs a new formatter with the specified output stream.
9	**Formatter(OutputStream os, String csn)** This constructor constructs a new formatter with the specified output stream and charset.
10	**Formatter(OutputStream os, String csn, Locale l)** This constructor constructs a new formatter with the specified output stream, charset, and locale.
11	**Formatter(PrintStream ps)** This constructor constructs a new formatter with the specified print stream.
12	**Formatter(String fileName)** This constructor constructs a new formatter with the specified file name.

13	**Formatter(String fileName, String csn)** This constructor constructs a new formatter with the specified file name and charset.
14	**Formatter(String fileName, String csn, Locale l)** This constructor constructs a new formatter with the specified file name, charset, and locale.

Java Random class

Java Random class is used to generate a stream of pseudorandom numbers. The algorithms implemented by Random class use a protected utility method than can supply up to 32 pseudorandomly generated bits on each invocation.

Methods

Methods	Description
doubles()	Returns an unlimited stream of pseudorandom double values.
ints()	Returns an unlimited stream of pseudorandom int values.
longs()	Returns an unlimited stream of pseudorandom long values.
next()	Generates the next pseudorandom number.
nextBoolean()	Returns the next uniformly distributed pseudorandom boolean value from the random number generator's sequence
nextByte()	Generates random bytes and puts them into a specified byte array.
nextDouble()	Returns the next pseudorandom Double value between 0.0 and 1.0 from the random number generator's sequence
nextFloat()	Returns the next uniformly distributed pseudorandom Float value

	between 0.0 and 1.0 from this random number generator's sequence
nextGaussian()	Returns the next pseudorandom Gaussian double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.
nextInt()	Returns a uniformly distributed pseudorandom int value generated from this random number generator's sequence
nextLong()	Returns the next uniformly distributed pseudorandom long value from the random number generator's sequence.
setSeed()	Sets the seed of this random number generator using a single long seed.

Example 1

```
1.  import java.util.Random;
2.  public class JavaRandomExample1 {
3.     public static void main(String[] args) {
4.        //create random object
5.        Random random= new Random();
6.        //returns unlimited stream of pseudorandom long values
7.           System.out.println("Longs value : "+random.longs());
8.       // Returns the next pseudorandom boolean value
9.        boolean val = random.nextBoolean();
10.       System.out.println("Random boolean value : "+val);
11.       byte[] bytes = new byte[10];
12.       //generates random bytes and put them in an array
13.       random.nextBytes(bytes);
14.       System.out.print("Random bytes = ( ");
15.       for(int i = 0; i< bytes.length; i++)
16.       {
```

```
17.             System.out.printf("%d ", bytes[i]);
18.         }
19.         System.out.print(")");
20.     }
21. }
```

Output:

```
Longs value : java.util.stream.LongPipeline$Head@14ae5a5
Random boolean value : true
Random bytes = ( 57 77 8 67 -122 -71 -79 -62 53 19 )
```

Java Date and Time

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time. Following classes are important for dealing with date in java.

Java Date/Time API
Java has introduced a new Date and Time API since Java 8. The java.time package contains Java Date and Time classes.

java.time.LocalDate class
java.time.LocalTime class
java.time.LocalDateTime class
java.time.MonthDay class
java.time.OffsetTime class
java.time.OffsetDateTime class
java.time.Clock class
java.time.ZonedDateTime class
java.time.ZoneId class
java.time.ZoneOffset class
java.time.Year class
java.time.YearMonth class
java.time.Period class
java.time.Duration class
java.time.Instant class
java.time.DayOfWeek enum
java.time.Month enum

TemporalAdjuster Class

TemporalAdjuster is to do the date mathematics. For example to get the "Second Saturday of the Month" or "Next Tuesday".

Let's see them in action.

Create the following java program using any editor of your choice in say C:/> JAVA

Java8Tester.java

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;

public class Java8Tester {
  public static void main(String args[]) {
    Java8Tester java8tester = new Java8Tester();
    java8tester.testAdjusters();
  }

  public void testAdjusters() {
    //Get the current date
    LocalDate date1 = LocalDate.now();
    System.out.println("Current date: " + date1);

    //get the next tuesday
    LocalDate nextTuesday = date1.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
    System.out.println("Next Tuesday on : " + nextTuesday);

    //get the second saturday of next month
    LocalDate firstInYear = LocalDate.of(date1.getYear(),date1.getMonth(), 1);

    LocalDate secondSaturday = firstInYear.with(
      TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY)).with(
      TemporalAdjusters.next(DayOfWeek.SATURDAY));
    System.out.println("Second saturday on : " + secondSaturday);
  }
}
```

Verify the result

Compile the class using javac compiler as follows

```
C:\JAVA>javac Java8Tester.java
```

Now run the Java8Tester to see the result

```
C:\JAVA>java Java8Tester
```

See the result.

```
Current date: 2014-12-10
Next Tuesday on : 2014-12-16
Second saturday on : 2014-12-13
```

Exception Handling in Java

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

In this page, we will learn about Java exceptions, its type 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

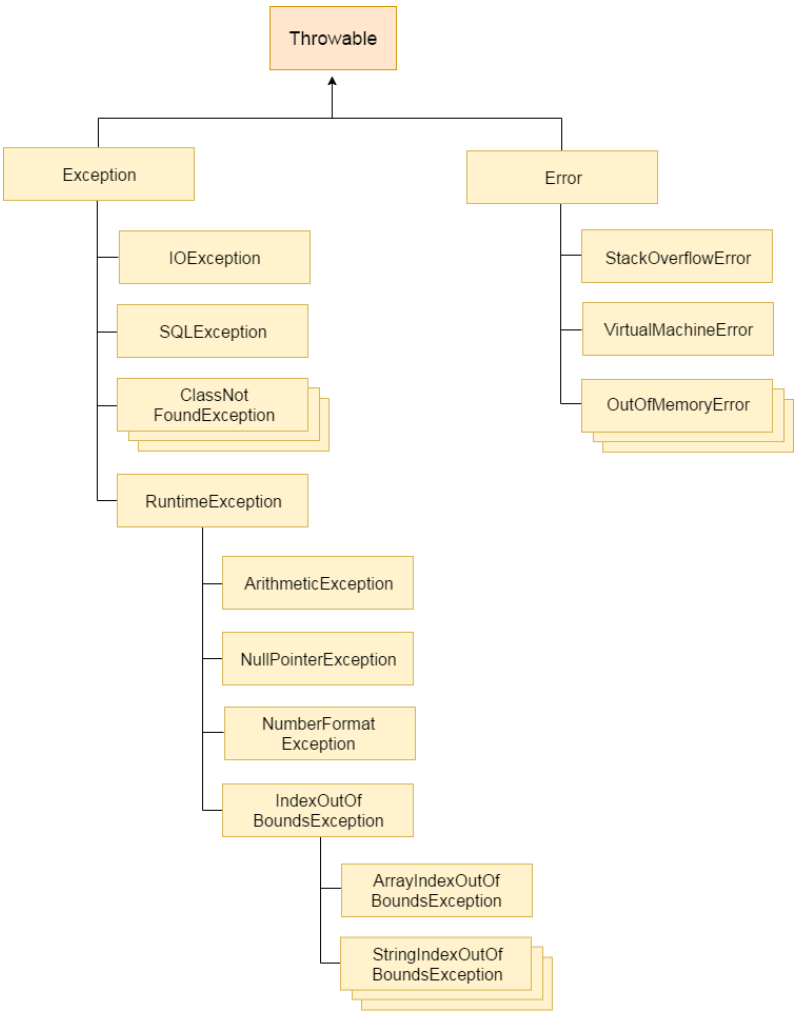
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement 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 which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

```
1. public class JavaExceptionExample{
2.   public static void main(String args[]){
3.    try{
4.       //code that may raise exception
5.       int data=100/0;
6.    }catch(ArithmeticException e){System.out.println(e);}
7.    //rest code of the program
8.    System.out.println("rest of the code...");
9.   }
10.}
```

Output:

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

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.

```
1. 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.

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

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

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

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

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

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 of try block, the rest of the block code will not execute. So, it is recommended not to keeping 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.

Example 1

```
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
```

Example 2

```
public class TryCatchExample2 {

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

```
4.      try
5.      {
6.      int data=50/0; //may throw exception
7.      }
8.         //handling the exception
9.      catch(ArithmeticException e)
10.     {
11.        System.out.println(e);
12.     }
13.     System.out.println("rest of the code");
14.  }
15.
16.}
```

Output:

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

Example 3

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

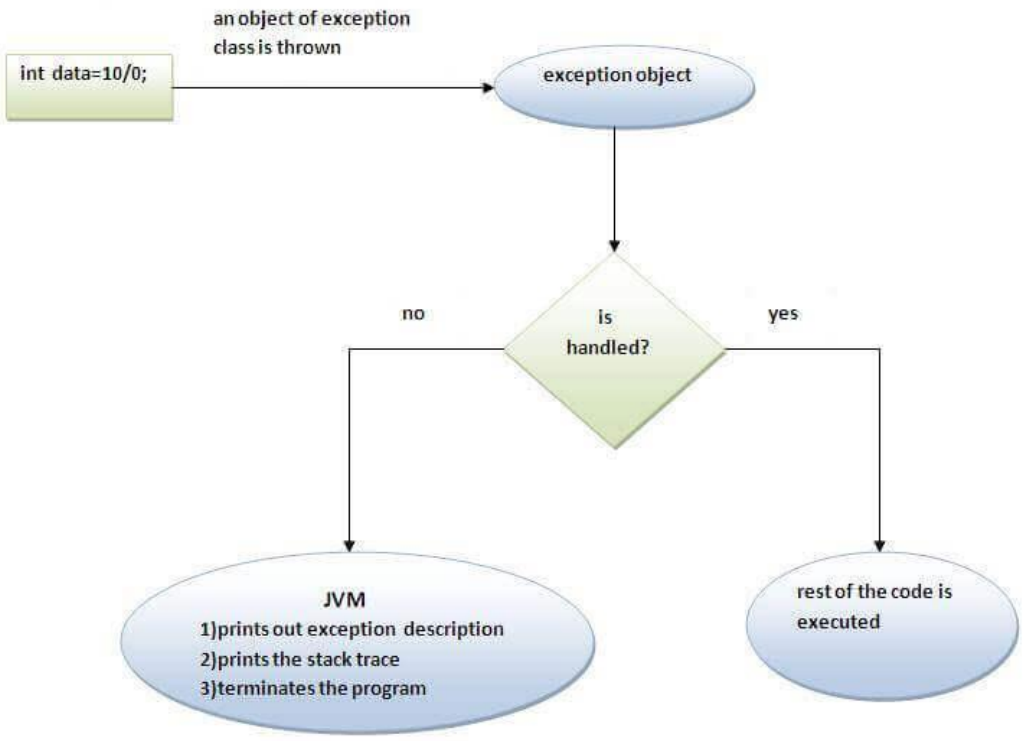
```
1.  public class TryCatchExample3 {
2.
3.     public static void main(String[] args) {
4.        try
5.        {
6.        int data=50/0; //may throw exception
7.                     // if exception occurs, the remaining statement will not exceute
8.        System.out.println("rest of the code");
9.        }
10.          // handling the exception
11.       catch(ArithmeticException e)
12.       {
13.          System.out.println(e);
14.       }
15.
16.    }
```

17.}

Output:

```
java.lang.ArithmeticException: / by zero
```

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 exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

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.

Example 1

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

```
1. public class MultipleCatchBlock1 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.            }
9.            catch(ArithmeticException e)
10.             {
11.              System.out.println("Arithmetic Exception occurs");
12.             }
13.           catch(ArrayIndexOutOfBoundsException e)
14.             {
15.              System.out.println("ArrayIndexOutOfBounds Exception occurs");
16.             }
17.           catch(Exception e)
18.             {
19.              System.out.println("Parent Exception occurs");
```

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

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2

```
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("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Example 3

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

```
1.  public class MultipleCatchBlock3 {
2.
3.     public static void main(String[] args) {
4.
5.          try{
6.              int a[]=new int[5];
7.              a[5]=30/0;
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
18.            catch(Exception e)
19.              {
20.               System.out.println("Parent Exception occurs");
21.              }
22.            System.out.println("rest of the code");
23.   }
24.}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Java Nested try block

The try block within a try block is known as nested try block in java.

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:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
.....
```

Java nested try example

Let's see a simple example of java nested try block.

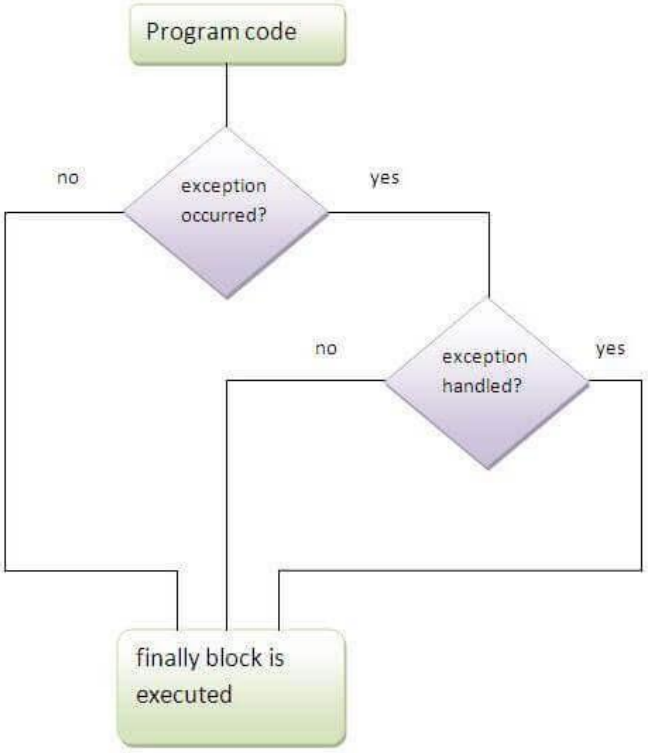
```
1. class Excep6{
2.  public static void main(String args[]){
3.   try{
4.     try{
5.      System.out.println("going to divide");
6.      int b =39/0;
7.     }catch(ArithmeticException e){System.out.println(e);}
8.
9.     try{
10.    int a[]=new int[5];
11.    a[5]=4;
12.    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.    System.out.println("other statement);
15.  }catch(Exception e){System.out.println("handeled");}
16.
17.  System.out.println("normal flow..");
18. }  }
```

Java finally block

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

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



Usage of Java finally

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1. class TestFinallyBlock{
2.   public static void main(String args[]){
3.   try{
4.    int data=25/5;
5.    System.out.println(data);
6.   }
7.   catch(NullPointerException e){System.out.println(e);}
8.   finally{System.out.println("finally block is always executed");}
9.   System.out.println("rest of the code...");
10.  }
11.}
```

```
12.  Output:5
13.          finally block is always executed
14.          rest of the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
1. class TestFinallyBlock1{
2.   public static void main(String args[]){
3.   try{
4.    int data=25/0;
5.    System.out.println(data);
6.   }
7.   catch(NullPointerException e){System.out.println(e);}
8.   finally{System.out.println("finally block is always executed");}
9.   System.out.println("rest of the code...");
10.  }
11.}
```

```
Output:finally block is always executed
       Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
1. public class TestFinallyBlock2{
2.   public static void main(String args[]){
3.   try{
4.    int data=25/0;
5.    System.out.println(data);
6.   }
7.   catch(ArithmeticException e){System.out.println(e);}
8.   finally{System.out.println("finally block is always executed");}
9.   System.out.println("rest of the code...");
10.  }
11.}
```

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```

Java throw keyword

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

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
1. throw exception;
```

Let's see the example of throw IOException.

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

java throw keyword example

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.

```
1. public class TestThrow1{
2.   static void validate(int age){
3.     if(age<18)
4.      throw new ArithmeticException("not valid");
5.     else
6.      System.out.println("welcome to vote");
7.   }
8.   public static void main(String args[]){
9.     validate(13);
10.    System.out.println("rest of the code...");
11. }
12.}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

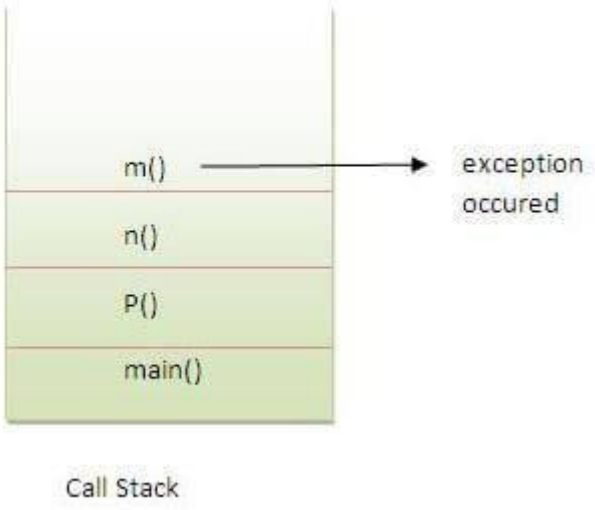
Java Exception propagation

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.

Program of Exception Propagation

```
1. class TestExceptionPropagation1{
2.   void m(){
3.     int data=50/0;
4.   }
5.   void n(){
6.     m();
7.   }
8.   void p(){
9.    try{
10.    n();
11.   }catch(Exception e){System.out.println("exception handled");}
12. }
13. public static void main(String args[]){
14.   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
15.   obj.p();
16.   System.out.println("normal flow...");
17. }
18.}
```

```
Output:exception handled
       normal flow...
```



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

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

Program which describes that checked exceptions are not propagated

```
1. class TestExceptionPropagation2{
2.   void m(){
3.     throw new java.io.IOException("device error");//checked exception
4.   }
5.   void n(){
6.     m();
7.   }
8.   void p(){
9.    try{
10.    n();
11.   }catch(Exception e){System.out.println("exception handeled");}
12. }
13. public static void main(String args[]){
14.  TestExceptionPropagation2 obj=new TestExceptionPropagation2();
15.  obj.p();
16.  System.out.println("normal flow");
17. }
18.}
```

```
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 normal flow 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 performing check up before the code being used.

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you 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.

```
1. import java.io.IOException;
2. class Testthrows1{
3.   void m()throws IOException{
4.     throw new IOException("device error");//checked exception
5.   }
```

```
6.   void n()throws IOException{
7.     m();
8.   }
9.   void p(){
10.   try{
11.    n();
12.   }catch(Exception e){System.out.println("exception handled");}
13.  }
14.  public static void main(String args[]){
15.   Testthrows1 obj=new Testthrows1();
16.   obj.p();
17.   System.out.println("normal flow...");
18.  }
19.}
```

Output:

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

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    throw new IOException("device error");
5.   }
6.  }
7.  public class Testthrows2{
8.    public static void main(String args[]){
9.     try{
```

```
10.     M m=new M();
11.     m.method();
12.    }catch(Exception e){System.out.println("exception handled");}
13.
14.    System.out.println("normal flow...");
15.  }
16.}
```

```
Output:exception handled
       normal flow...
```

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    System.out.println("device operation performed");
5.   }
6.  }
7.  class Testthrows3{
8.    public static void main(String args[])throws IOException{//declare exception
9.      M m=new M();
10.     m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14.}
```

```
Output:device operation performed
       normal flow...
```

B)Program if exception occurs

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
```

```
4.    throw new IOException("device error");
5.   }
6.  }
7.  class Testthrows4{
8.     public static void main(String args[])throws IOException{//declare exception
9.        M m=new M();
10.     m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14.}
```

```
Output:Runtime Exception
```

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
1. void m(){
2. throw new ArithmeticException("sorry");
3. }
```

Java throws example

```
1. void m()throws ArithmeticException{
2. //method code
3. }
```

Java throw and throws example

```
1. void m()throws ArithmeticException{
2. throw new ArithmeticException("sorry");
3. }
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
1. class FinalExample{
2. public static void main(String[] args){
3. final int x=100;
4. x=200;//Compile Time Error
5. }}
```

Java finally example

```
1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
6. finally{System.out.println("finally block is executed");}
7. }}
```

Java finalize example

```
1. class FinalizeExample{
2. public void finalize(){System.out.println("finalize called");}
3. public static void main(String[] args){
4. FinalizeExample f1=new FinalizeExample();
5. FinalizeExample f2=new FinalizeExample();
6. f1=null;
7. f2=null;
8. System.gc();
9. }}
```

ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- **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

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class TestExceptionChild extends Parent{
7.   void msg()throws IOException{
8.     System.out.println("TestExceptionChild");
9.   }
10.  public static void main(String args[]){
11.   Parent p=new TestExceptionChild();
12.   p.msg();
13.  }
14.}
      15.  Output:Compile Time Error
```

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class TestExceptionChild1 extends Parent{
```

```
7.   void msg()throws ArithmeticException{
8.     System.out.println("child");
9.   }
10.  public static void main(String args[]){
11.   Parent p=new TestExceptionChild1();
12.   p.msg();
13.  }
14.}
```

```
Output:child
```

Example in case subclass overridden method declares parent exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws ArithmeticException{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild2 extends Parent{
7.    void msg()throws Exception{System.out.println("child");}
8.
9.    public static void main(String args[]){
10.   Parent p=new TestExceptionChild2();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15.}
```

```
Output:Compile Time Error
```

Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
1. class InvalidAgeException extends Exception{
2.  InvalidAgeException(String s){
3.   super(s);
4.  }
5. }
```

```
1. class TestCustomException1{
2.
3.   static void validate(int age)throws InvalidAgeException{
4.     if(age<18)
5.      throw new InvalidAgeException("not valid");
6.     else
7.      System.out.println("welcome to vote");
8.   }
9.
10.  public static void main(String args[]){
11.    try{
12.    validate(13);
13.    }catch(Exception m){System.out.println("Exception occured: "+m);}
14.
15.    System.out.println("rest of the code...");
16. }
17.}
```

```
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```