

Packages and Java Library

Packages:

- A Package can be defined as a grouping of related types(classes, interfaces)
- A package represents a directory that contains related group of classes and interfaces.
- Packages are used in Java in order to prevent naming conflicts.
- There are two types of packages in Java.
 1. Pre-defined Packages(built-in)
 2. User Defined Packages

Pre-defined Packages:

Package Name	Description
java.lang	Contains language support classes (for e.g classes which defines primitive data types, math operations, etc.). This package is automatically imported.
java.io	Contains classes for supporting input / output operations.
java.util	Contains utility classes which implement data structures like Linked List, Hash Table, Dictionary, etc and support for Date / Time operations. This package is also called as Collections .
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc).
java.net	Contains classes for supporting networking operations.
javax.swing	This package helps to develop GUI Applications. The 'x' in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, javax.swing is an extended package of java.awt.
java.sql	This package helps to connect to databases like Oracle/Sybase/Microsoft Access to perform different operations.

Defining a Package(User defined):

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 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.

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;
```

```
package pack;
```

Example: Package demonstration

```
package pack;
public class Addition
{
    int x,y;
    public Addition(int a, int b)
    {
        x=a;
        y=b;
    }
    public void sum()
    {
        System.out.println("Sum :"+(x+y));
    }
}

public class Subtraction

{
    int x,y;
    public Subtraction(int a, int b)
    {
        x=a;
        y=b;
    }
    public void diff()
    {
        System.out.println("Difference :"+(x-y));
    }
}
```

1. With fully qualified name.

```
class UseofPack
{
    public static void main(String arg[])
    {
        pack.Addition a=new pack.Addition(10,15);
        a.sum();
        pack.Subtraction s=new pack.Subtraction(20,15);
        s.difference();
    }
}
```

2. import package.classname

```

import pack.Addition;
import pack.Subtraction;
class UseofPack
{
    public static void main(String arg[])
    {
        Addition a=new Addition(10,15);
        a.sum();
        Subtraction s=new Subtraction(20,15);
        s.difference();
    }
}

3. import package.*;

import pack.*;
class UseofPack
{
    public static void main(String arg[])
    {
        Addition a=new Addition(10,15);
        a.sum();
        Subtraction s=new Subtraction(20,15);
        s.difference();
    }
}

```

- To create a package , use the "package" keyword

MyPackageClass.java

```

package mypack;
class MyPackageClass
{
    public static void main(String args[])
    {
        System.out.println("This is my package");
    }
}

```

- c:\Users\513> javac MyPackageClass.java

Access Protection:

Access protection defines actually how much an element (class, method, variable) is exposed to other classes and packages.

- There are four types of access specifiers available in java

- Visible to the class only (private).
- Visible to the package (default). No modifiers are needed.
- Visible to the package and all subclasses (protected)
- Visible to the world (public)

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Example: The following example shows all combinations of the access control modifiers. This example has two packages and five classes. The source for the first package defines three classes:

Protection, Derived & Same Package.

Name of the package: pkg1

This file is Protection.java

```
package pkg1;

public class Protection
{
    int n = 1;
    private int n_priv = 2;
    protected int n_prot = 3;
    public int n_publ = 4;

    public Protection()
    {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_priv = " + n_priv);
        System.out.println("n_prot = " + n_prot);
        System.out.println("n_publ = " + n_publ);
    }
}
```

This is file Derived.java:

```
package pkg1;

class Derived extends Protection
{
    Derived()
    {
        System.out.println("Same package - derived (from base) constructor");
        System.out.println("n = " + n);
```

```

    /* class only
     * System.out.println("n_priv = "4 + n_priv); */

    System.out.println("n_prot = " + n_prot);
    System.out.println("n_publ = " + n_publ);
}
}

```

This is file SamePackage.java

```

package pkg1;

class SamePackage
{
    SamePackage()
    {
        Protection pro = new Protection();
        System.out.println("same package - other constructor");
        System.out.println("n = " + pro.n);

        /* class only
         * System.out.println("n_priv = " + pro.n_priv); */

        System.out.println("n_prot = " + pro.n_prot);
        System.out.println("n_publ = " + pro.n_publ);
    }
}

```

Name of the package: pkg2

This is file Protection2.java:

```

package pkg2;

class Protection2 extends pkg1.Protection
{
    Protection2()
    {
        System.out.println("Other package-Derived (from Package 1-Base)
Constructor");

        /* class or package only
         * System.out.println("n = " + n); */
    }
}

```

```

    /* class only
     * System.out.println("n_priv = " + n_priv); */

    System.out.println("n_prot = " + n_prot);
    System.out.println("n_publ = " + n_publ);
}
}

```

This is file **OtherPackage.java**

```

package pkg2;

class OtherPackage
{
    OtherPackage()
    {
        pkg1.Protection pro = new pkg1.Protection();

        System.out.println("other package - Non sub class constructor");

        /* class or package only
         * System.out.println("n = " + pro.n); */

        /* class only
         * System.out.println("n_priv = " + pro.n_priv); */

        /* class, subclass or package only
         * System.out.println("n_prot = " + pro.n_prot); */

        System.out.println("n_publ = " + pro.n_publ);
    }
}

```

If you want to try these two packages, here are two test files you can use. The package **pkg1** is shown here:

```

/* demo package pkg1 */

package pkg1;

/* instantiate the various classes in pkg1 */
public class Demo
{
    public static void main(String args[])
    {
        Derived obj2 = new Derived();

```

```
        SamePackage obj3 = new SamePackage();  
    }  
}
```

The test file for package pkg2 is

```
package pkg2;  
  
/* instantiate the various classes in pkg2 */  
public class Demo2  
{  
    public static void main(String args[])  
    {  
        Protection2 obj1 = new Protection2();  
        OtherPackage obj2 = new OtherPackage();  
    }  
}
```

CLASSPATH:

Is an environment variable (i.e., global variables of the operating system available to all the processes) needed for the Java compiler and runtime to locate the Java packages/classes used in a Java program.

CLASSPATH can be set permanently in the environment:

In Windows, choose

control panel ⇒ System ⇒ Advanced ⇒ Environment Variables ⇒ choose "System Variables" (for all the users) or "User Variables" (only the currently login user) ⇒ choose "Edit" (if CLASSPATH already exists) or "New" ⇒ Enter "CLASSPATH" as the variable name ⇒ Enter the required directories and JAR files (separated by semicolons)

Class path

c:\> set path=%path%;c:\program files\java\jdk 1.8.0\bin

!! On **%CLASS PATH%** !!

Types of Packages in Java SE(standard Edition)

- **java. Lang:** provides classes that are fundamental to the design
- **java.io:** provides classes for system input and output through data streams
- **java. Util:** contains the collections framework, legacy collection classes, event model, date and time facilities
- **java. Math:** provides precision integer arithmetic
- **java. Awt:** contains classes and interfaces that provide a powerful infrastructure for networking
- **java.net.** contains classes and interfaces that provide a powerful infrastructure for networking

Java.lang packages & its Classes

Core Classes:

1. **Object** - The root class of all Java classes
2. **String** - Represents text strings
3. **StringBuffer** - Mutable sequence of characters (thread-safe)
4. **StringBuilder** - Mutable sequence of characters (not thread-safe)
5. **System** - Provides system-related utilities and standard I/O

Wrapper Classes:

- **Boolean**
- **Byte**
- **Character**
- **Short**
- **Integer**
- **Long**
- **Float**
- **Double**

Thread-related:

- Thread
- ThreadGroup
- Runnable (interface)

Exception Classes:

- Throwable
- Exception
- Error
- RuntimeException

Math and Number:

- Math
- Number
- StrictMath

Other Important Classes:

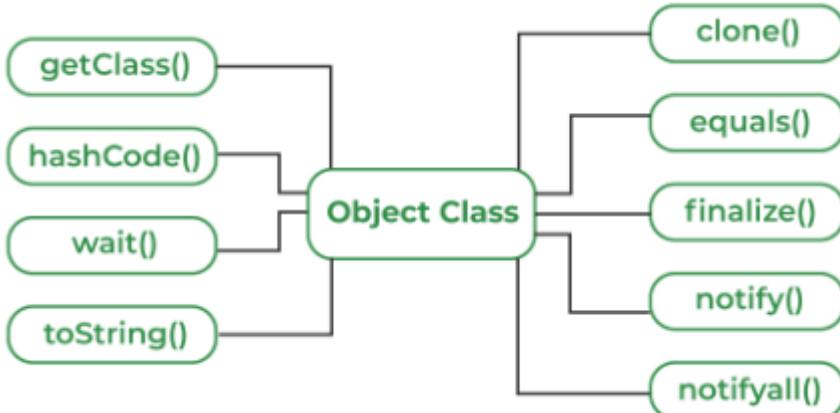
- Class - Represents classes and interfaces at runtime
- ClassLoader - Loads classes and resources
- Process - Represents a native process
- Runtime - Provides interface to the Java runtime system
- SecurityManager - Security check operations
- Package - Package reflection
- Enum - Base class for all Java enumerations
- Void - Placeholder for void type

Class Object:

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 a direct child class of Object and if extends another class then it is indirectly derived.

Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.



Using Object Class Methods

The Object class provides multiple methods which are as follows:

- `toString()` method
- `hashCode()` method
- `equals(Object obj)` method
- `finalize()` method
- `getClass()` method
- `clone()` method
- `wait(), notify(), notifyAll()` methods
- **1. `toString()` method**

The `toString()` provides a String representation of an object and is used to convert an object to a 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

2. `hashCode()` method

For every _object, JVM generates a unique number which is a hashcode. It returns distinct integers for distinct objects

3. `equals(Object obj)` method

It 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.

4. getClass() method

It returns the class object of “this” object and is used to get the actual runtime class of the object. It can also be used to get metadata of this class.

5. finalize() method

This method is called just before an object is garbage collected. It is called the [Garbage Collector](#) on an object when the garbage collector determines that there are no more references to the object. We should override finalize() method to dispose of system resources, perform clean-up activities and minimize memory leaks.

6. clone() method

It returns a new object that is exactly the same as this object.

The remaining three methods **wait()**, **notify()** **notifyAll()** are related to Concurrency.

These methods are used in thread class .

Enumerations:

Enumerations are used to create our own data type.

To use this feature we need to use Enum keyword.

```
1. class EnumExample1  
2. {  
3. //defining the enum inside the class  
4. public enum Season { WINTER, SPRING, SUMMER, FALL }  
5. //main method  
6. public static void main(String[] args)  
7. {  
8. //traversing the enum  
9. for (Season s : Season.values())  
10. System.out.println(s);  
11. }  
12. }
```

CLASS MATH:

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

Math Methods:

Method	Description
Math.sin()	Returns the trigonometric value of the sine of an angle.
Math.cos()	Returns the trigonometric value of the cosine of an angle.
Math.tan()	Returns the trigonometric value of the tangent of an angle.
round	Returns the closest int to the argument, with ties rounding to positive infinity
Math.pow()	Returns pow of a number
Math.min()	Returns Smallest number of two values
Math.max()	Returns largest number of two values
Math.abs()	Returns absolute value of given value
Math.sqrt()	Returns square root of a number

Wrapper Classes:

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

All wrapper classes are called Reference Data types. Wrapper classes are part of Java.lang package.

AutoBoxing:

[Autoboxing](#) refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing.

For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

```

// Java Program to Illustrate Autoboxing

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
class GFG {

    // Main method
    public static void main(String[] args)
    {

        // Creating an empty ArrayList of integer type
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Adding the int primitives type values
        // using add() method
        // Autoboxing
        al.add(1);
        al.add(2);
        al.add(24);

        // Printing the ArrayList elements
        System.out.println("ArrayList: " + al);
    }
}

```

Auto Unboxing:

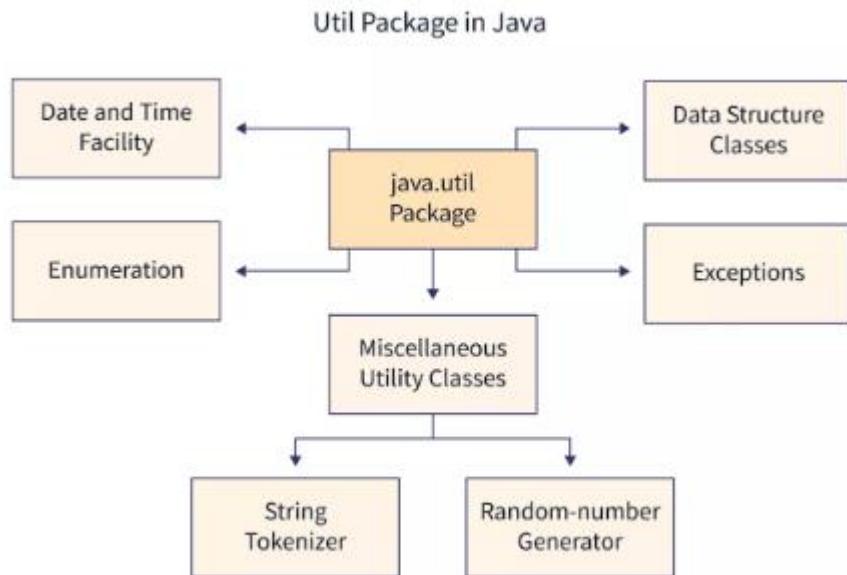
The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

1. class UnboxingExample1
2. {
3. public static void main(String args[])
4. {
5. Integer i=new Integer(50);
6. int a=i;
7. System.out.println(a);
8. }
9. }

Java Util Classes and Interfaces:

Java.util Package

It contains the collections framework, legacy collection classes, event model, date and time facilities.



Class	Description
Collections	Provides methods to represent and manage collections
Formatter	An interpreter for format strings
Scanner	Text scanner used to take user inputs
Arrays	Provides methods for array manipulation
LinkedList	Implements Doubly-linked list data structure
HashMap	Implements Map data structure using Hash tables
TreeMap	Implements Red-Black tree data structure
Stack	Implements last-in-first-out (LIFO) data structure
PriorityQueue	Implements unbounded priority queue data structure
Date	Represents a specific instance of time
Calendar	Provides methods to manipulate calendar fields
Random	Used to generate a stream of pseudorandom numbers.
StringTokenizer	Used to break a string into tokens
Timer, TimerTask	Used by threads to schedule tasks
UUID	Represents an immutable universally unique identifier

java.util Package Interfaces

Sr.No.	Intreface & Description
---------------	------------------------------------

1	Collection<E>
---	----------------------------

This is the root interface in the collection hierarchy.

2	Comparator<T>
---	----------------------------

This is a comparison function, which imposes a total ordering on some collection of objects.

3	Deque<E>
---	-----------------------

This is a linear collection that supports element insertion and removal at both ends.

4	Enumeration<E>
---	-----------------------------

This is an object that implements the Enumeration interface generates a series of elements, one at a time.

5	EventListener
---	----------------------

This is a tagging interface that all event listener interfaces must extend.

6	Formattable
---	--------------------

This is the Formattable interface must be implemented by any class that needs to perform custom formatting using the 's' conversion specifier of Formatter.

7	Iterator<E>
---	--------------------------

This is an iterator over a collection.

8	List<E>
---	----------------------

This is an ordered collection (also known as a sequence).

9	ListIterator<E>
---	------------------------------

This is an iterator for lists that allows the programmer to traverse the list in either direction, move list during iteration, and obtain the iterator's current position in the list.

10	Map<K,V>
----	-----------------------

This is an object that maps keys to values.

11	Map.Entry<K,V>
----	-----------------------------

This is a map entry (key-value pair).

12	NavigableMap<K,V>
----	--------------------------------

This is a SortedMap extended with navigation methods returning the closest matches for given targets.

13	NavigableSet<E>
----	------------------------------

This is a SortedSet extended with navigation methods reporting closest matches for given search targets.

Observer

- 14 This is a class can implement the Observer interface when it wants to be informed of changes in observable objects.

Queue<E>

- 15 This is a collection designed for holding elements prior to processing.

RandomAccess()

- 16 This is the Marker interface used by List implementations to indicate that they support fast (generally constant time) random access.

Set<E>

- 17 This is a collection that contains no duplicate elements.

SortedMap<K,V>

- 18 This is a Map that further provides a total ordering on its keys.

SortedSet<E>

- 19 This is a Set that further provides a total ordering on its elements.

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.

S.No.	Constructor with Description
1	Formatter() It creates a new formatter.
2	Formatter(Appendable a) It creates a new formatter with the specified destination.
3	Formatter(Appendable a, Locale l) It creates a new formatter with the specified destination and locale.
4	Formatter(File file) It creates a new formatter with the specified file.

The Formatter class in java has the following methods.

void flush()

It flushes the invoking formatter.

Appendable out()

It returns the destination for the output.

Locale locale(): It returns the locale set by the construction of the invoking formatter.

Random class:

Random class from the java.util package is used to generate random numbers. It can produce integers, floats, doubles, and booleans.

Methods of Random class:

- nextInt(): Returns a random integer (positive or negative).
- nextDouble(): Returns a random double between 0.0 and 1.0
- nextFloat(): Returns a random float between 0.0 and 1.0
- nextBoolean(): Returns a random boolean (true or false).
- nextLong(): Returns a random long integer.

```
import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        // Creating an instance of Random
        Random random = new Random();

        // Generating different types of random numbers
        int randomInt = random.nextInt(); // any integer
        int randomIntBounded = random.nextInt(100); // integer between 0 (inclusive) and 100
        (exclusive)
        double randomDouble = random.nextDouble(); // double between 0.0 and 1.0
        boolean randomBoolean = random.nextBoolean(); // true or false

        // Printing the random values
        System.out.println("Random Integer: " + randomInt);
        System.out.println("Random Integer (0-99): " + randomIntBounded);
        System.out.println("Random Double (0.0 - 1.0): " + randomDouble);
        System.out.println("Random Boolean: " + randomBoolean);
    }
}
```

Time Package:

Java does not have a built-in Date class, but we can import the java.time package to work with the date and time API.

The package includes many date and time classes. For example:

Class	Description
<code>LocalDate</code>	Represents a date (year, month, day (yyyy-MM-dd))
<code>LocalTime</code>	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
<code>LocalDateTime</code>	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
<code>DateTimeFormatter</code>	Formatter for displaying and parsing date-time objects

Class Instant:

The Instant Class is used to represent the specific time instant on the current timeline. The Instant Class extends the Object Class and implements the Comparable interface.

Method	Description
<code>adjustInto(Temporal temporal)</code>	This method adjusts the specified temporal object to have this instant.
<code>atOffset(ZoneOffset offset)</code>	This method combines this instant with an offset to create an OffsetDateTime.
<code>atZone(ZonedDateTime zone)</code>	This method combines this instant with a time-zone to create a ZonedDateTime.
<code>compareTo(Instant otherInstant)</code>	This method compares this instant to the specified instant.

```
import java.time.Instant;
public class ShowInstantTime
{
    public static void main(String[] args) { // Get the current instant time
        Instant currentInstant = Instant.now(); // Display the current instant time
        System.out.println("Current Instant Time: " + currentInstant);
    }
}
```

Formatting for Date\Time in java:

Formatting is used to separate the Date from time by using DateTimeFormatter.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
```

```

public class DateTimeFormatting
{
    public static void main(String[] args)
    {
        // Get the current date and time
        LocalDateTime currentDateTime = LocalDateTime.now();

        // Define a date and time format
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");

        // Format the current date and time
        String formattedDateTime = currentDateTime.format(formatter);

        // Display the formatted date and time
        System.out.println("Formatted Date and Time: " + formattedDateTime);
    }
}

```

Temporal Adjusters Class:

TemporalAdjusters Class in Java provides Adjusters, which are a key tool for modifying temporal objects

Examples include an adjuster that sets the date like “Second Saturday of the Month” or “Next Tuesday”,

The following two ways of using TemporalAdjuster are equivalent, but it is recommended to use the second approach

```
temporal = thisAdjuster.adjustInto(temporal);
temporal = temporal.with(thisAdjuster);
```

```
firstDayOfMonth()
lastDayOfMonth()
firstDayOfNextMonth()
firstDayOfYear()
```

```
import java.time.LocalDate;
import java.time.DayOfWeek;
import java.time.temporal.TemporalAdjusters;
public class NextMondayExample
{
    public static void main(String[] args)
    {
        // Get today's date
        LocalDate today = LocalDate.now();
```

```
System.out.println("Today's Date: " + today); // Adjust to the next Monday
LocalDate nextMonday = today.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
System.out.println("Next Monday: " + nextMonday);
}
}
```

Exception Handling

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

What is exception

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

Advantage of Exception Handling

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

Types of Exception

There are mainly two types of exceptions:

checked and unchecked

where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

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

- 2) **Unchecked Exception:** The classes that extend RuntimeException are known as unchecked

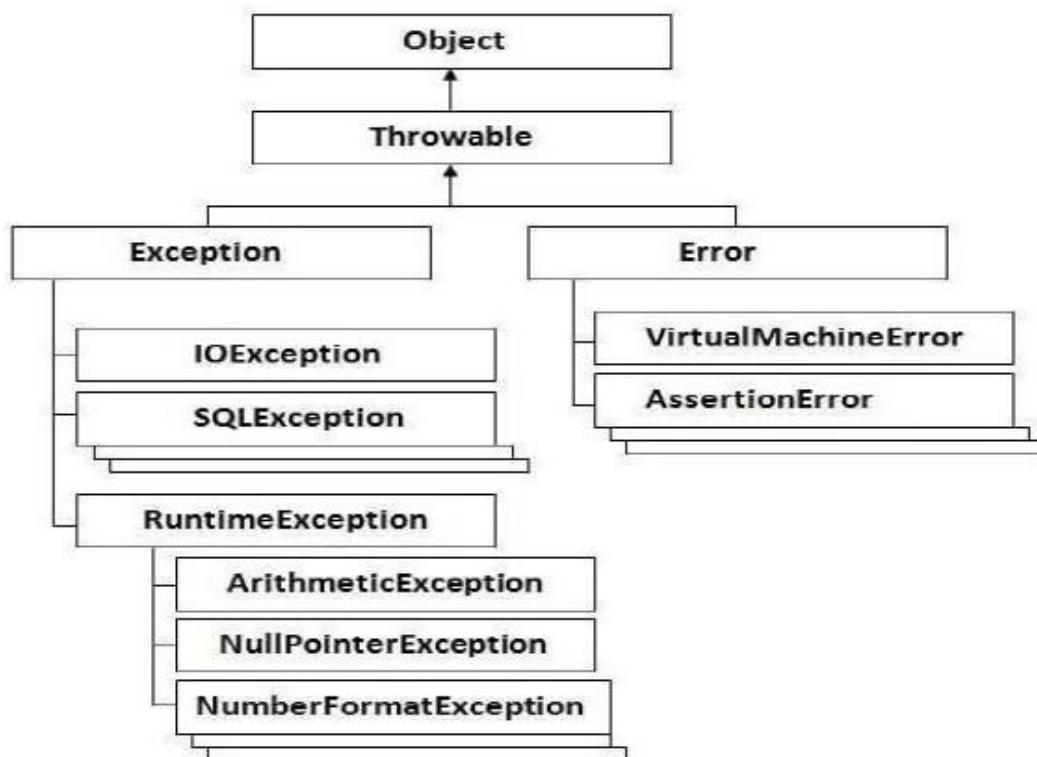
exceptions e.g.

ArithmaticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

- 3) **Error:** Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionErrot etc.

Hierarchy of Java Exception classes



Checked and UnChecked Exceptions

Checked Exceptions	Unchecked Exceptions
<ul style="list-style-type: none">Exception which are checked at Compile time called Checked ExceptionIf a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword	<ul style="list-style-type: none">Exceptions whose handling is NOT verified during Compile time.These exceptions are handled at run-time i.e., by JVM after they occurred by using the try and catch block
<ul style="list-style-type: none">Examples:<ul style="list-style-type: none">IOExceptionSQLExceptionDataAccessExceptionClassNotFoundExceptionInvocationTargetExceptionMalformedURLException	<ul style="list-style-type: none">Examples:<ul style="list-style-type: none">NullPointerExceptionArrayIndexOutOfBoundsExceptionIllegalArgumentExceptionIllegalStateException

Java try block

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

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

Syntax of java try-catch

1. **try{**
2. **//code that may throw exception**
3. **}catch(Exception_class_Name ref){}**

Syntax of try-finally block

1. **try{**
2. **//code that may throw exception**
3. **}finally{}**

Java catch block

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

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1
{
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.ArithmetricException:/ by zero
```

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

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

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

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

1. Output:

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

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

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

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

```
1. public class TestMultipleCatchBlock{
2.     public static void main(String args[]){
3.         try{
4.             int a[] = new int[5];
5.             a[5] = 30/0;
6.         }
7.         catch(ArithmaticException e){System.out.println("task1 is completed");}
8.         catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9.     }
10.    catch(Exception e){System.out.println("common task completed");}
11. }
12. System.out.println("rest of the code...");
13. }
```

```
Output:task1 completed
rest of the code...
```

Java nested try example

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

```

class Excep6
{
public static void main(String args[])
{
try
{
try
{
System.out.println("going to divide");
int b =39/0;
catch(ArithmetricException e)
{
System.out.println(e);
}
try
{
int a[]=new int[5];
a[5]=4;

catch(ArrayIndexOutOfBoundsException e)
{
System.out.println(e);}

System.out.println("other statement");

catch(Exception e)
{
System.out.println("handede");
System.out.println("normal flow..");
}
}
}

```

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**.

```
class TestFinallyBlock
{
    public static void main(String args[])
    {
        Try
        {
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException e)
        {
            System.out.println(e);
        }
        Finally
        {
            System.out.println("finally block is always executed");
            System.out.println("rest of the code...");
        }
    }
}
```

Output:5

```
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 unchecked 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.

```
throw exception;
```

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  
{  
    static void validate(int age)  
    {  
        if(age<18)  
            throw new ArithmeticException("not valid");  
        else  
            System.out.println("welcome to vote");  
    }  
    public static void main(String args[]){  
        validate(13);  
        System.out.println("rest of the code...");  
    }  
}
```

Output:

Exception in thread main java.lang.ArithmaticException:not valid

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. {
2. //method code
3. }
- 4.

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...

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.

```
class InvalidAgeException extends Exception
{
    InvalidAgeException(String s)
    {
        super(s);
    }
}

class TestCustomException1
{
```

```

static void validate(int age) throws InvalidAgeException
{
    if(age<18)
        throw new InvalidAgeException("not valid");
    else
        System.out.println("welcome to vote");
}
public static void main(String args[])
{
    Try
    {
        validate(13);
    }catch(Exception m)
    {
        System.out.println("Exception occurred: "+m);
    }
    System.out.println("rest of the code...");
}
}

```

Output:Exception

occured: InvalidAgeException:not valid rest of the code...

Java I/O and File:

The java.io package contains nearly every class you might ever need to perform input and output(I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams –

❑ InPutStream – The InputStream is used to read data from a source.

❑ OutPutStream – The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this covers very basic functionality related to streams and I/O. We will see the most commonly used

examples one by one –

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes.

Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file.

Example

```
import java.io.*;

public class CopyFile {

    public static void main(String args[]) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {

            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;

            while ((c = in.read()) != -1) {


```

```
    out.write(c);

}

}finally {

    if (in != null) {

        in.close();

    }

    if (out != null) {

        out.close();

    }
}
```

Now let's have a file **input.txt** with the following content –

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt

file with the same content as we have in input.txt. So let's put the above code in CopyFile.java

file and do the following

```
$javac CopyFile.java
```

```
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas

Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two

bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an inputfile (having unicode characters) into an output file –

Example

```
import java.io.*;

public class CopyFile {

    public static void main(String args[]) throws IOException {

        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Now let's have a file **input.txt** with the following content –

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following –

```
$javac CopyFile.java
```

```
$java
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. Java provides the following three standard streams –

❑ **Standard Input** – This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as**System.in**.

❑ **Standard Output** – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.

❑ **Standard Error** – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream

until the user types a "

Example

```
import java.io.*;  
  
public class ReadConsole {  
  
    public static void main(String args[]) throws IOException {  
  
        InputStreamReader cin = null;  
  
        try {  
  
            cin = new InputStreamReader(System.in);  
  
            System.out.println("Enter characters, 'q' to quit.");  
  
            char c;  
  
            do {  
  
                c = (char) cin.read();  
  
                System.out.print(c);  
  
            } while(c != 'q');  
  
        } finally {  
  
            if (cin != null) {  
  
                cin.close();  
  
            } } }}}
```

This program continues to read and output the same character until we press 'q'

```
$javac ReadConsole.java  
$java ReadConsole
```

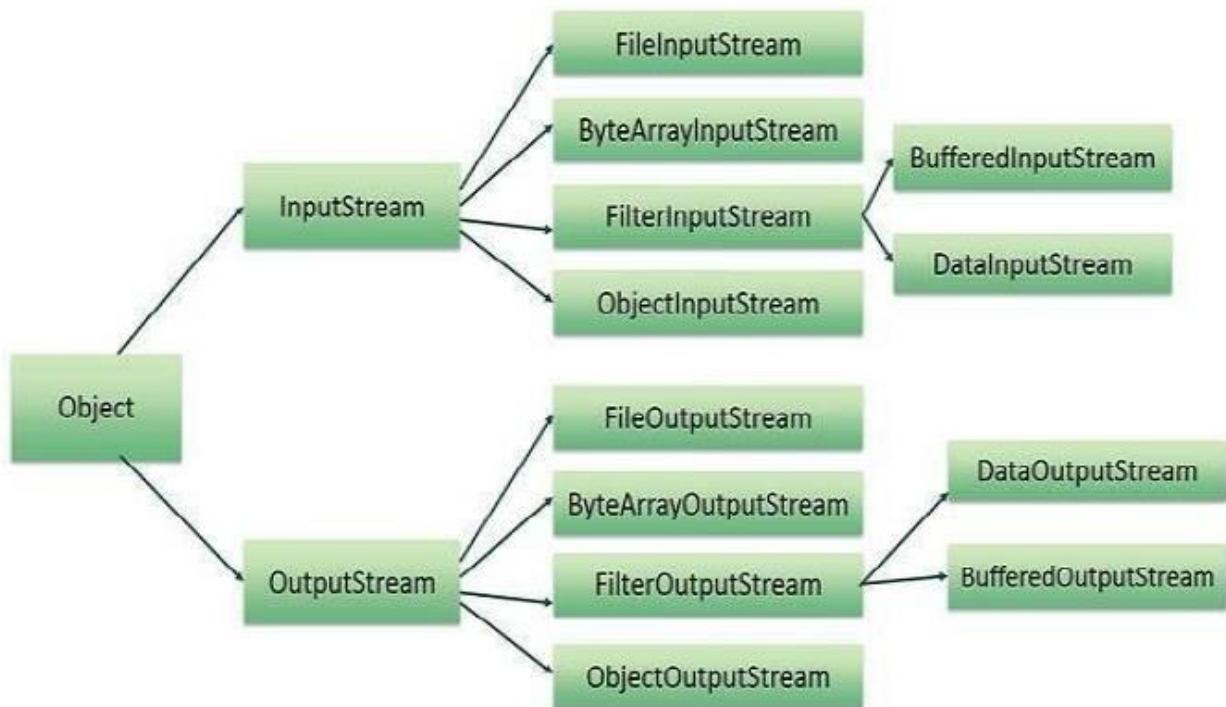
```
Enter characters, 'q' to quit.
```

```
1  
1  
e  
e  
q  
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file –

```
InputStream f= new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file.

First we create a file object using File() method as follows –

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

❑ [ByteArrayInputStream](#)

❑ [DataInputStream](#)

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a *FileOutputStream* object.

Following constructor takes a file name as a string to create an input stream object to write the file –Following constructor takes a file object to create an output stream object to write the file. First,

we create a file object using File() method as follows –

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be

used to write to stream or to do other operations on the stream.

❑ [ByteArrayOutputStream](#)

❑ [DataOutputStream](#)

Example

Following is the example to demonstrate *InputStream* and *OutputStream* –

```
import java.io.*;  
  
public class fileStreamTest  
{  
  
    public static void main(String args[])  
    {  
  
        try
```

```
byte bWrite [] = {11,21,3,40,5};

OutputStream os = new FileOutputStream("test.txt");

for(int x = 0; x < bWrite.length ; x++) {

os.write( bWrite[x] ); // writes the bytes}

os.close();

InputStream is = new FileInputStream("test.txt");

int size = is.available();

for(int i = 0; i < size; i++)

{

System.out.print((char)is.read() + " ");

}

is.close();

} catch (IOException e)

{

System.out.print("Exception");

}

}

}
```

Java Scanner class

There are various ways to read input from the keyboard, the `java.util.Scanner` class is one of them.

The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using regular expression

Java Scanner class extends `Object` class and implements `Iterator` and `Closeable` interfaces.

Commonly used methods of Scanner class

There is a list of commonly used Scanner class methods:

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.

Java Scanner Example to get input from console

Let's see the simple example of the Java Scanner class which reads the int, string and double value as an input:

```
import java.util.Scanner;
class ScannerTest{
public static void main(String args[]){
Scanner sc=new Scanner(System.in);
System.out.println("Enter your rollno");
int rollno=sc.nextInt();
System.out.println("Enter your name");
String name=sc.next();
System.out.println("Enter your fee");
double fee=sc.nextDouble();
System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
sc.close();
} } Output:
```

```
Enter your rollno
111
Enter your name
Ratan
Enter
450000
Rollno:111 name:Ratan fee:450000
```