UNIT IV

Packages and Java Library: Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path, Access Control, Packages in Java SE, Java.lang Package and its Classes, Class Object, Enumeration, class Math, Wrapper Classes, Auto-boxing and Auto-unboxing, Java util Classes and Interfaces, Formatter Class, Random Class, Time Package, Class Instant (java.time.Instant), Formatting for Date/Time in Java, Temporal Adjusters Class, Temporal Adjusters Class.

Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throwable, Unchecked Exceptions, Checked Exceptions, try-with-resources, Catching Subclass Exception, Custom Exceptions, Nested try and catch Blocks, Rethrowing Exception, Throws Clause.

4.1 PACKAGES AND JAVA LIBRARY

4.1.1 Introduction

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

Javaprovides a mechanism for partitioning the class name space into more manageablechunks. This mechanism is the package. The package is both a naming and a visibilitycontrol mechanism. You can define classes inside a package that are not accessible bycode outside that package. You can also define class members that are only exposedto other members of the same package. This allows your classes to have intimateknowledge of each other, but not expose that knowledge to the rest of the world.

4.1.2 Defining a Package

To create a package is quite easy: simply include a package command as the first statementin 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 thepackage statement, the class names are put into the default package, which has no name.While the default packageis fine for short, sample programs, it is inadequate for real applications. Most of the time,you will define a package for your code.
This is the general form of the package statement:
package pkg;
Here, pkg is the name of the package.

For example, the following statement creates a packagecalled MyPackage.
package MyPackage;

Java uses file system directories to store packages. For example, the .class files for anyclasses 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 nameexactly.

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

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

package pkg1[.pkg2[.pkg3]];

Apackage hierarchy must be reflected in the file system of your Java developmentsystem.

For example, a package declared aspackage java.awt.image;

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

4.1.3 Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories.

This raises an important question:How does the Java run-time system know where to look for packages that you create?

Theanswer has three parts. First, by default, the Java run-time system uses the current workingdirectory as its starting point. Thus, if your package is in a subdirectory of the currentdirectory, it will be found. Second, you can specify a directory path or paths by setting theCLASSPATH environmental variable. Third, you can use the -classpath option with javaand javac to specify the path to your classes.

For example, consider the following package specification:
package MyPack

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

When the second two options are used, the class path must not include MyPack, itself.It must simply specify the path to MyPack.

For example, in a Windows environment, if thepath to MyPack is
C:\MyPrograms\Java\MyPack

Then the class path to MyPack is
C:\MyPrograms\Java

The easiest way to try the examples shown in this book is to simply create the packagedirectories below your current development directory, put the .class files into theappropriate directories, and then execute the programs from the development directory.

This is the approach used in the following example.
A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

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

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

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above MyPack when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section tospecify the path MyPack.)

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

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

4.1.4 Access Protection

In the preceding chapters, you learned about various aspects of Java's access control mechanismand its access specifiers. For example, you already know that access to a private member ofa class is granted only to other members of that class. Packages add another dimension toaccess control. As you will see, Java provides many levels of protection to allow fine-grainedcontrol over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name spaceand scope of variables and methods. Packages act as containers for classes and othersubordinate packages. Classes act as containers for data and code.

The class is Java'ssmallest unit of abstraction. Because of the interplay between classes and packages, Javaaddresses four categories of visibility for class members:
• Subclasses in the same package
• Non-subclasses in the same package
• Subclasses in different packages
• Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of waysto produce the many levels of access required by these categories. Table 9-1 sums up theinteractions.

While Java's access control mechanism may seem complicated, we can simplify it asfollows. Anything declared public can be accessed from anywhere. Anything declaredprivate cannot be seen outside of its class. When a member does not have an explicit accessspecification, it is visible to subclasses as well as to other classes in the same package. This isthe default access. If you want to allow an element to be seen outside your current package,but only to classes that subclass your class directly, then declare that element protected.

	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

Table applies only to members of classes. A non-nested class has only two possibleaccess levels: default and public. When a class is declared as public, it is accessible by anyother code. If a class has default access, then it can only be accessed by other code within itssame package. When a class is public, it must be the only public class declared in the file,and the file must have the same name as the class.

An Access Example

The following example shows all combinations of the access control modifiers. This examplehas two packages and five classes. Remember that the classes for the two differentpackages need to be stored in directories named after their respective packages—in thiscase, p1 and p2.

The source for the first package defines three classes: Protection, Derived, and SamePackage.The first class defines four int variables in each of the legal protection modes. The variable nis declared with the default protection, n_pri is private, n_pro is protected, and n_pub ispublic.

Each subsequent class in this example will try to access the variables in an instanceof this class. The lines that will not compile due to access restrictions are commented out.

Before each of these lines is a comment listing the places from which this level of protectionwould allow access.

The second class, Derived, is a subclass of Protection in the same package, p1. Thisgrants Derived access to every variable in Protection except for n_pri, the private one. Thethird class, SamePackage, is not a subclass of Protection, but is in the same package andalso has access to all but n_pri.

This is file Protection.java:

```
package p1;
public class Protection {
int n = 1;
private intn_pri = 2;
protected intn_pro = 3;
public intn_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file Derived.java:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file SamePackage.java:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, p2. The two classes defined in p2cover the other two conditions that are affected by access control. The first class, Protection2, isa subclass of p1.Protection. This grants access to all of p1.Protection’s variables except forn_pri (because it is private) and n, the variable declared with the default protection. Remember,the default only allows access from within the class or the package, not extra-packagesubclasses.
Finally, the class OtherPackage has access to only one variable, n_pub, whichwas declared public.

This is file Protection2.java:

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file OtherPackage.java:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one forpackage p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}   }
```

4.1.4 Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classesfrom each other, it is easy to see why all of the built-in Java classes are stored in packages.

There are no core Java classes in the unnamed default package; all of the standard classesare stored in some named package. Since classes within packages must be fully qualifiedwith their package name or names, it could become tedious to type in the long dot-separatedpackage path name for every class you want to use.

For this reason, Java includes the importstatement to bring certain classes, or entire packages, into visibility. Once imported, a classcan be referred to directly, using only its name. The import statement is a convenience tothe programmer and is not technically needed to write a complete Java program. If you aregoing to refer to a few dozen classes in your application, however, the import statement willsave a lot of typing.

In a Java source file, import statements occur immediately following the package statement(if it exists) and before any class definitions. This is the general form of the import statement:

```
import pkg1[.pkg2].(classname|*);
```

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinatepackage inside the outer package separated by a dot (.). There is no practical limit on thedepth of a package hierarchy, except that imposed by the file system. Finally, you specifyeither an explicit classname or a star (*), which indicates that the Java compiler should importthe entire package.

This code fragment shows both forms in use:
```
importjava.util.Date;
import java.io.*;
```

NOTE: The star form may increase compilation time—especially if you import several largepackages. For this reason it is a good idea to explicitly name the classes that you want to userather than importing whole packages. However, the star form has absolutely no effect on therun-time performance or size of your classes.

All of the standard Java classes included with Java are stored in a package called java.The basic language functions are stored in a package inside of the java package calledjava.lang. Normally, you have to import every package or class that you want to use, butsince Java is useless without much of the functionality in java.lang, it is implicitly importedby the compiler for all programs.

This is equivalent to the following line being at the top ofall of your programs:
```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using thestar form, the compiler will remain silent, unless you try to use one of the classes. In that case,you will get a compile-time error and have to explicitly name the class specifying its package.

It must be emphasized that the import statement is optional. Any place you use a classname, you can use its fully qualified name, which includes its full package hierarchy.

Forexample, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the import statement looks like this:

```
classMyDate extends java.util.Date {
}
```

In this version, Date is fully-qualified.

As shown in Table, when a package is imported, only those items within the packagedeclared as public will be available to non-subclasses in the importing code.

For example,if you want the Balance class of the package MyPack shown earlier to be available as astand-alone class for general use outside of MyPack, then you will need to declare it aspublic and put it into its own file, as shown here:

```
package MyPack;

/* Now, the Balance class, its constructor, and itsshow() method are public. This means that they can
be used by non-subclass code outside their package.*/
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n;
bal = b;
}
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

As you can see, the Balance class is now public. Also, its constructor and its show( )method are public, too. This means that they can be accessed by any type of code outsidethe MyPack package.

For example, here TestBalance imports MyPack and is then able tomake use of the Balance class:

```
importMyPack.*;
class TestBalance {
public static void main(String args[]) {
/* Because Balance is public, you may use Balance
class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

As an experiment, remove the public specifier from the Balance class and then trycompiling TestBalance. As explained, errors will result.

4.1.5 Packages in JAVA SE 8

java.applet	java.text
java.awt	java.time
java.beans	java.util
java.io	javax.accessibility
java.lang	javax.activaion
java.math	javax.activity
java.net	javax.annotation
java.nio	javax.crypto
java.rmi	javax.imageio
java.security	javax.jws
java.sql	. . . etc

Source: https://docs.oracle.com/javase/8/docs/api/

4.1.6 Java.lang Package and its Classes

This topic discusses those classes and interfaces defined by java.lang. As you know, java.lang is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of Java programming. It is Java’s most widely used package.

java.lang includes the following classes:

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

java.lang defines the following interfaces:

Appendable	Cloneable	Readable
AutoCloseable	Comparable	Runnable
CharSequence	Iterable	Thread.UncaughtExceptionHandler

```
package Pack;
import java.lang.*;
public class ByteDemo
{
  public static void main(String[] args) {
        Byte b1, b2;
        String s1, s2;
```

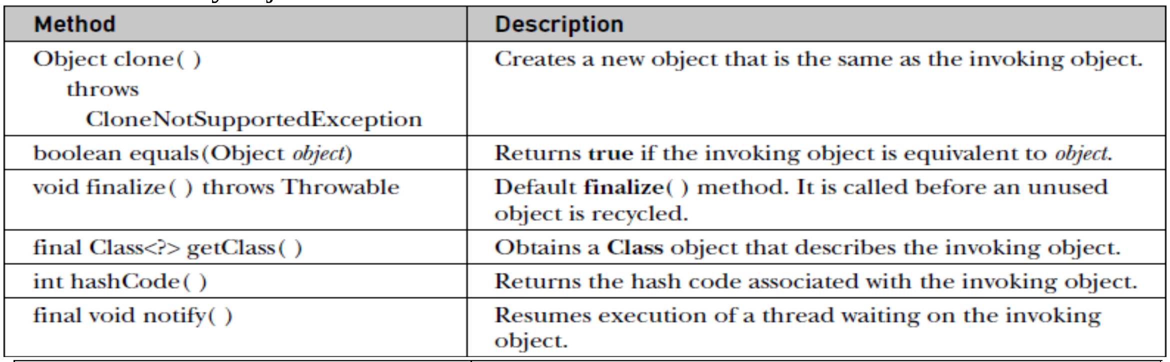
```
                b1 = new Byte("123");
                b2 = new Byte("-123");
          s1 = b1.toString();
          s2 = b2.toString();

      String str1 = "String value of Byte " + b1 + " is " + s1;
      String str2 = "String value of Byte " + b2 + " is " + s2;
          System.out.println( str1 );
          System.out.println( str2 );
   }
}
```

OUTPUT:

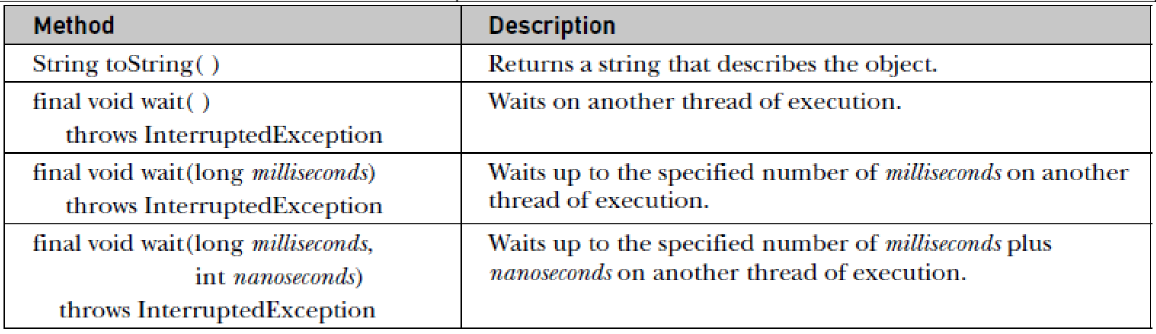
```
String value of Byte 123 is 123
String value of Byte -123 is -123
```

Class Object

Object is a superclass of all other classes. Object defines the methods shown in Table, which are available to every object.



Method	Description
Object clone( ) throws CloneNotSupportedException	Creates a new object that is the same as the invoking object.
boolean equals(Object *object*)	Returns **true** if the invoking object is equivalent to *object*.
void finalize( ) throws Throwable	Default **finalize**( ) method. It is called before an unused object is recycled.
final Class<?> getClass( )	Obtains a **Class** object that describes the invoking object.
int hashCode( )	Returns the hash code associated with the invoking object.
final void notify( )	Resumes execution of a thread waiting on the invoking object.



Method	Description
String toString( )	Returns a string that describes the object.
final void wait( ) throws InterruptedException	Waits on another thread of execution.
final void wait(long *milliseconds*) throws InterruptedException	Waits up to the specified number of *milliseconds* on another thread of execution.
final void wait(long *milliseconds*, int *nanoseconds*) throws InterruptedException	Waits up to the specified number of *milliseconds* plus *nanoseconds* on another thread of execution.

```
package Pack;
import java.lang.*;
public class ClassDemo
{
   public static void main(String[] args) {
      // returns the Class object associated with this class
      ClassDemo cl = new ClassDemo();
      Class c1Class = cl.getClass();
      // returns the name of the class
      String name = c1Class.getName();
      System.out.println("Class Name = " + name);
   }
}
```

OUTPUT:

```
Class Name = Pack.ClassDemo
```

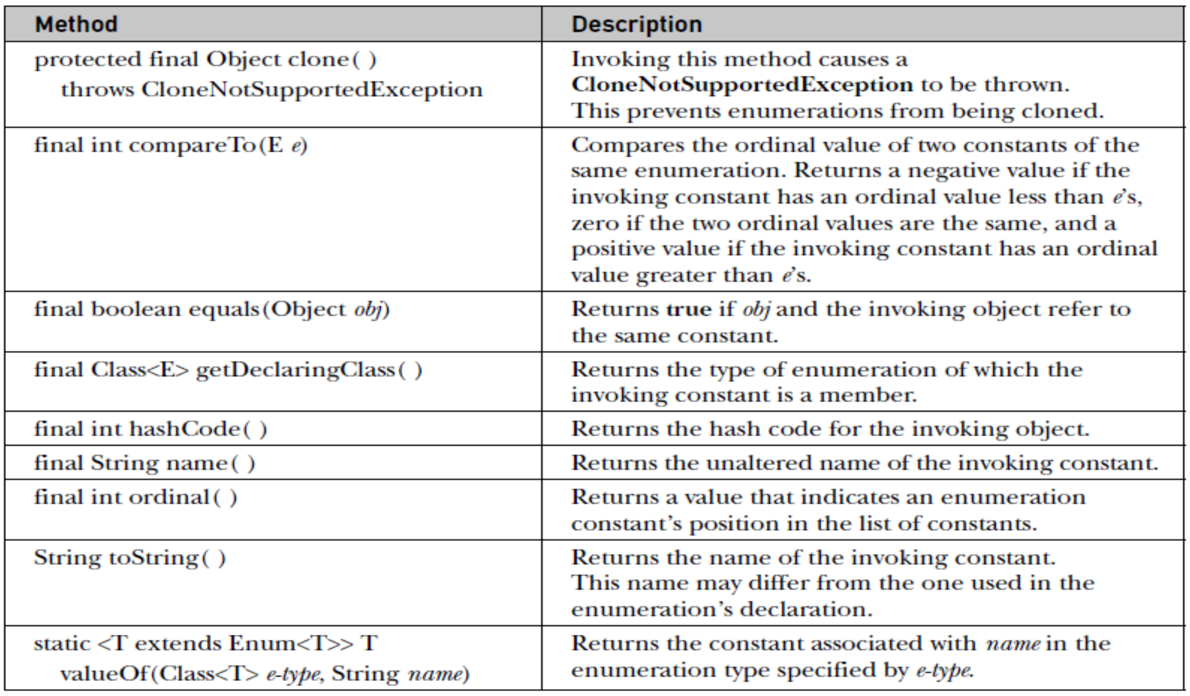
4.1.7 Enumeration

An enumeration is a list of named constants. An enumeration is created by using the keyword enum. All enumerations automatically inherit Enum.

Enum is a generic class that is declared as shown here:

```
class Enum<E extends Enum<E>>
```

Here, E stands for the enumeration type. Enum has no public constructors. Enum defines several methods that are available for use by all enumerations, which are shown in Table.



Method	**Description**
protected final Object clone( ) throws CloneNotSupportedException	Invoking this method causes a **CloneNotSupportedException** to be thrown. This prevents enumerations from being cloned.
final int compareTo(E *e*)	Compares the ordinal value of two constants of the same enumeration. Returns a negative value if the invoking constant has an ordinal value less than *e*'s, zero if the two ordinal values are the same, and a positive value if the invoking constant has an ordinal value greater than *e*'s.
final boolean equals(Object *obj*)	Returns **true** if *obj* and the invoking object refer to the same constant.
final Class<E> getDeclaringClass( )	Returns the type of enumeration of which the invoking constant is a member.
final int hashCode( )	Returns the hash code for the invoking object.
final String name( )	Returns the unaltered name of the invoking constant.
final int ordinal( )	Returns a value that indicates an enumeration constant's position in the list of constants.
String toString( )	Returns the name of the invoking constant. This name may differ from the one used in the enumeration's declaration.
static <T extends Enum<T>> T valueOf(Class<T> *e-type*, String *name*)	Returns the constant associated with *name* in the enumeration type specified by *e-type*.

Description:
The java.lang.Enum.toString() method returns the name of this enum constant, as contained in the declaration.

Declaration:
Following is the declaration for java.lang.Enum.toString() method

```
 package Pack;
 import java.lang.*;
 enum Language {   C, Java, PHP;    }
 public class EnumDemo
 {
  public static void main(String args[])
  {
      System.out.println("Programming in " + Language.C.toString());
      System.out.println("Programming in " + Language.Java.toString());
      System.out.println("Programming in " + Language.PHP.toString());
  }
```

```
}
```

OUTPUT:

```
Programming in C
Programming in Java
Programming in PHP
```

4.1.8 class Math

The Math class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods.
Math defines two double constants:
E (approximately 2.72) and PI (approximately 3.14).

Trigonometric Functions

The following methods accept a double parameter for an angle in radians and return the result of their respective trigonometric function:

Method	Description
static double sin(double *arg*)	Returns the sine of the angle specified by *arg* in radians.
static double cos(double *arg*)	Returns the cosine of the angle specified by *arg* in radians.
static double tan(double *arg*)	Returns the tangent of the angle specified by *arg* in radians.

The next methods take as a parameter the result of a trigonometric function and return, in radians, the angle that would produce that result. They are the inverse of their non-arc companions.

Method	Description
static double asin(double *arg*)	Returns the angle whose sine is specified by *arg*.
static double acos(double *arg*)	Returns the angle whose cosine is specified by *arg*.
static double atan(double *arg*)	Returns the angle whose tangent is specified by *arg*.
static double atan2(double *x*, double *y*)	Returns the angle whose tangent is *x*/*y*.

The next methods compute the hyperbolic sine, cosine, and tangent of an angle:

Method	Description
static double sinh(double *arg*)	Returns the hyperbolic sine of the angle specified by *arg*.
static double cosh(double *arg*)	Returns the hyperbolic cosine of the angle specified by *arg*.
static double tanh(double *arg*)	Returns the hyperbolic tangent of the angle specified by *arg*.

Exponential Functions

Math defines the following exponential methods:

Method	Description
static double cbrt(double *arg*)	Returns the cube root of *arg*.
static double exp(double *arg*)	Returns e to the *arg*.
static double expm1(double *arg*)	Returns e to the *arg*–1.
static double log(double *arg*)	Returns the natural logarithm of *arg*.
static double log10(double *arg*)	Returns the base 10 logarithm for *arg*.
static double log1p(double *arg*)	Returns the natural logarithm for *arg* + 1.
static double pow(double *y*, double *x*)	Returns *y* raised to the *x*; for example, pow(2.0, 3.0) returns 8.0.
static double scalb(double *arg*, int *factor*)	Returns $arg \times 2^{factor}$.
static float scalb(float *arg*, int *factor*)	Returns $arg \times 2^{factor}$.
static double sqrt(double *arg*)	Returns the square root of *arg*.

Rounding Functions

The Math class defines several methods that provide various types of rounding operations. They are shown in Table. Notice the two ulp( ) methods at the end of the table. In this context, ulp stands for units in the last place. It indicates the distance between a value and the next higher value. It can be used to help assess the accuracy of a result.

Method	Description
static int abs(int *arg*)	Returns the absolute value of *arg*.
static long abs(long *arg*)	Returns the absolute value of *arg*.
static float abs(float *arg*)	Returns the absolute value of *arg*.
static double abs(double *arg*)	Returns the absolute value of *arg*.
static double ceil(double *arg*)	Returns the smallest whole number greater than or equal to *arg*.
static double floor(double *arg*)	Returns the largest whole number less than or equal to *arg*.
static int max(int *x*, int *y*)	Returns the maximum of *x* and *y*.
static long max(long *x*, long *y*)	Returns the maximum of *x* and *y*.
static float max(float *x*, float *y*)	Returns the maximum of *x* and *y*.
static double max(double *x*, double *y*)	Returns the maximum of *x* and *y*.
static int min(int *x*, int *y*)	Returns the minimum of *x* and *y*.
static long min(long *x*, long *y*)	Returns the minimum of *x* and *y*.
static float min(float *x*, float *y*)	Returns the minimum of *x* and *y*.

The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.max(x,y)

The Math.max(x,y) method can be used to find the highest value of x and y:

Example: Math.max(5, 10);

Math.min(x,y)

The Math.min(x,y) method can be used to find the lowest value of x and y:

Example: Math.min(5, 10);

Math.sqrt(x)

The Math.sqrt(x) method returns the square root of x:

Example: Math.sqrt(64);

```
public class ABC
{
 public static void main(String[] args)
{
  System.out.println(Math.sqrt(64));
 }
}
```

OUTPUT:

```
8.0
```

4.1.9 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
Primitive Data Type	Wrapper Class

```
public class Main1
{
  public static void main(String[] args)
  {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
      System.out.println(myInt);
      System.out.println(myDouble);
      System.out.println(myChar);
  }
}
```

OUTPUT:

```
5
5.99
A
```

For example, the following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

```
public class Main2 {
public static void main(String[] args) {
  Integer myInt = 5;
  Double myDouble = 5.99;
  Character myChar = 'A';
     System.out.println(myInt.intValue());
     System.out.println(myDouble.doubleValue());
     System.out.println(myChar.charValue());
  }
}
```

OUTPUT:

```
5
5.99
A
```

Another useful method is the toString() method, which is used to convert wrapper objects to strings.
In the following example, we convert an Integer to a String, and use the length() method of the String class to output the length of the "string":

```
public class Main3
{
  public static void main(String[] args)
  {
   Integer myInt = 100;
   String myString = myInt.toString();
   System.out.println(myString.length());
  }
}
```

OUTPUT:

```
3
```

4.1.10 Auto-boxing and Auto-unboxing

With the beginning of JDK 5, Java added two more important features; Autoboxing and Auto unboxing.
The automatic adaptation of primitive data types into its corresponding Wrapper type is known as boxing, and reverse operation is known as unboxing.
One advantage of using this feature is that programmers do not require to convert between primitives and Wrappers manually and hence less coding is needed.
Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrappers whenever an object of the type is needed.
There is no need to construct an object explicitly.
Autoboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when the program requires its value.

Occur in Expressions:

In general, autoboxing and auto-unboxing take place whenever a conversion into an object or form an object is required.
It applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the object is automatically reboxed, when necessary.

Here is a simple program showing the working of autoboxing and auto-unboxing:

```
public class Main4
{
  public static void main(String args[])
  {
    Integer iOb = 60, iOb2; // autobox an int
    int i = iOb; // auto-unbox
    System.out.println(i + " " + iOb);
    iOb2 = iOb+(iOb/3);
    System.out.println(i + " " + iOb2);
  }
}
```

OUTPUT:

```
60 60
60 80
```

4.1.11 Java.util Classes and Interfaces

java.util has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data.
The java.util package also contains one of Java's most powerful subsystems: the Collections Framework.
The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
java.util contains a wide array of functionality, it is quite large. Here is a list of its top-level classes:

AbstractCollection	EventObject	PropertyResourceBundle
AbstractList	FormattableFlags	Random
AbstractMap	Formatter	ResourceBundle
AbstractQueue	GregorianCalendar	Scanner
AbstractSequentialList	HashMap	ServiceLoader
AbstractSet	HashSet	SimpleTimeZone
ArrayDeque	Hashtable	Stack
ArrayList	IdentityHashMap	StringTokenizer
Arrays	LinkedHashMap	Timer
BitSet	LinkedHashSet	TimerTask
Calendar	LinkedList	TimeZone
Collections	ListResourceBundle	TreeMap
Currency	Locale	TreeSet
Date	Objects (Added by JDK 7.)	UUID
Dictionary	Observable	Vector

EnumMap	PriorityQueue	WeakHashMap
EnumSet	Properties	
EventListenerProxy	PropertyPermission	

The interfaces defined by java.util are shown next:

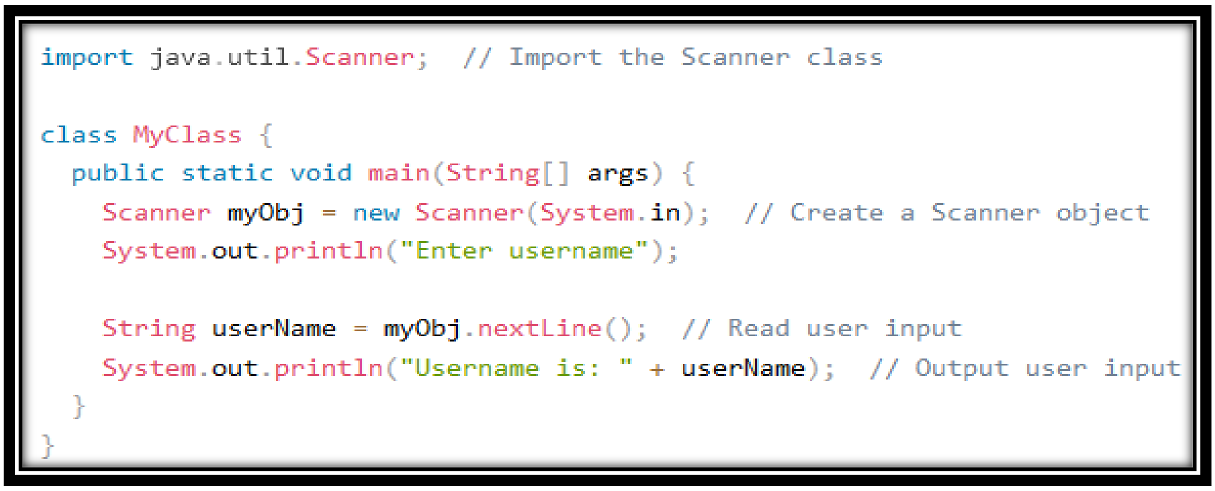
Collection	List	Queue
Comparator	ListIterator	RandomAccess
Deque	Map	Set
Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap	SortedSet
Formattable	NavigableSet	
Iterator	Observer	

Collections Overview

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
The Collections Framework was designed to meet several goals. First, the framework had to be high-performance.
The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
Third, extending and/or adapting a collection had to be easy.
Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the Collections class.

Scanner Class

The Scanner class is used to get user input, and it is found in the java.util package.
To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.
In our example, we will use the nextLine() method, which is used to read Strings:



```
import java.util.Scanner;  // Import the Scanner class

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

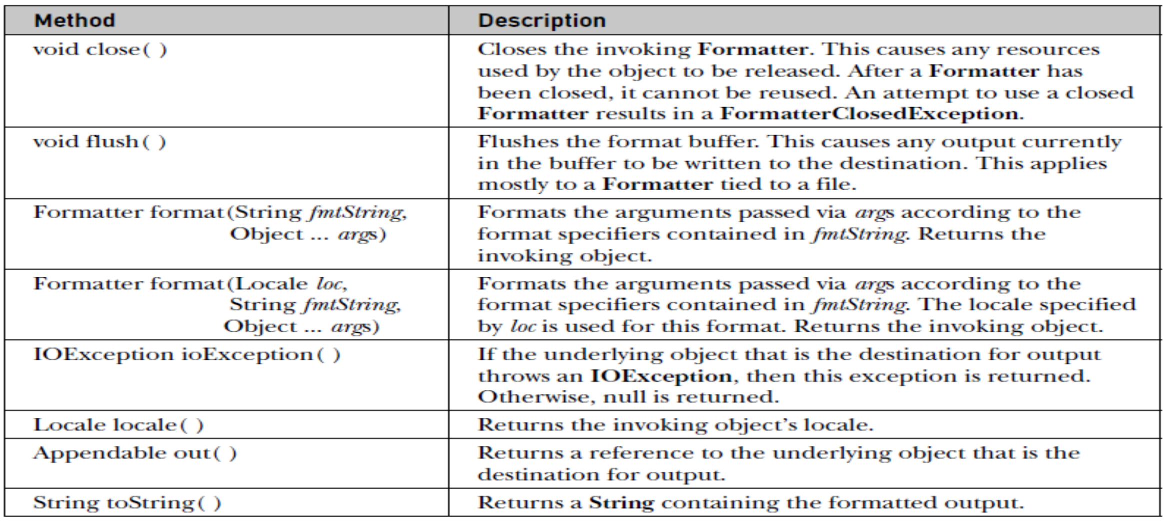
Method	Description
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

4.1.12 Formatter Class

It provides format conversions that let you display numbers, strings, and time and date in virtually any format you like.
It operates in a manner similar to the C/C++ printf( ) function, which means that if you are familiar with C/C++, then learning to use Formatter will be very easy.

The Formatter Constructors
Before you can use Formatter to format output, you must create a Formatter object. In general, Formatter works by converting the binary form of data used by a program into formatted text.
The Formatter class defines many constructors, which enable you to construct a Formatter in a variety of ways.
Formatter( )
Formatter(Appendable buf)
Formatter(Appendable buf, Locale loc)
Formatter(String filename) throws FileNotFoundException
Formatter(String filename, String charset) throws FileNotFoundException, UnsupportedEncodingException
Formatter(File outF) throws FileNotFoundException
Formatter(OutputStream outStrm)

Formatter defines the methods shown in Table



Method	Description
void close( )	Closes the invoking **Formatter**. This causes any resources used by the object to be released. After a **Formatter** has been closed, it cannot be reused. An attempt to use a closed **Formatter** results in a **FormatterClosedException**.
void flush( )	Flushes the format buffer. This causes any output currently in the buffer to be written to the destination. This applies mostly to a **Formatter** tied to a file.
Formatter format(String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. Returns the invoking object.
Formatter format(Locale *loc*, String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. The locale specified by *loc* is used for this format. Returns the invoking object.
IOException ioException( )	If the underlying object that is the destination for output throws an **IOException**, then this exception is returned. Otherwise, null is returned.
Locale locale( )	Returns the invoking object's locale.
Appendable out( )	Returns a reference to the underlying object that is the destination for output.
String toString( )	Returns a **String** containing the formatted output.

```
        Formatter fmt = new Formatter();
        fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
```

This sequence creates a Formatter that contains the following string:

```
        Formatting with Java is easy 10 98.600000
```

4.1.13 Random Class

The Random class is a generator of pseudorandom numbers. These are called pseudorandom numbers because they are simply uniformly distributed sequences.

Random defines the following constructors:

```
                Random( )
                Random(long seed)
```

The first version creates a number generator that uses a reasonably unique seed.
The second form allows you to specify a seed value manually.
The public methods defined by Random are shown in Table

There are seven types of random numbers that you can extract from a Random object.

Method	Description
boolean nextBoolean( )	Returns the next **boolean** random number.
void nextBytes(byte *vals*[ ])	Fills *vals* with randomly generated values.
double nextDouble( )	Returns the next **double** random number.
float nextFloat( )	Returns the next **float** random number.
double nextGaussian( )	Returns the next Gaussian random number.
int nextInt( )	Returns the next **int** random number.
int nextInt(int *n*)	Returns the next **int** random number within the range zero to *n*.
long nextLong( )	Returns the next **long** random number.
void setSeed(long *newSeed*)	Sets the seed value (that is, the starting point for the random number generator) to that specified by *newSeed*.

```
import java.util.*;
public class Demo
{
        public static void main( String args[] )
        {
                Random r = new Random();
   System.out.println("random value " + r.nextInt());
   }
 }
```

OUTPUT:
Any random number from int type range

4.1.14 Time Package

This package provides support for date and time related features for program developers.
It comprises classes that represent date /time concepts include instant, duration, date, time, time zones, and periods.
The date and time framework in java prior to Java SE 8 consisted of classes such as java.util.Date, java.util.Calander, and java.util.Simple date Formatter.
From Java SE 8 consists of the primary package java.time and four sub packages that are java.time.format, java.time.chrono, java.time.temporal, and java.time.zone.
The java.time package contains many classes; however, the mostly used classes of time package are described in table.

Class	Description
Clock	A clock providing access to the current instant, date and time using a time-zone.
Duration	A time-based amount of time, such as '34.5 seconds'.
Instant	An instantaneous point on the time-line.
LocalDate	A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.
LocalDateTime	A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30.
LocalTime	A time without a time-zone in the ISO-8601 calendar system, such as 10:15:30.
MonthDay	A month-day in the ISO-8601 calendar system, such as --12-03.
OffsetDateTime	A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00.
OffsetTime	A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 10:15:30+01:00.
Period	A date-based amount of time in the ISO-8601 calendar system, such as '2 years, 3 months and 4 days'.
Year	A year in the ISO-8601 calendar system, such as 2007.
YearMonth	A year-month in the ISO-8601 calendar system, such as 2007-12.
ZonedDateTime	A date-time with a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00 Europe/Paris.
ZoneId	A time-zone ID, such as Europe/Paris.
ZoneOffset	A time-zone offset from Greenwich/UTC, such as +02:00.

4.1.15 Class Instant (java.time.Instant)

The Instant class is the core class of new time framework in Java.

The time object returned by this class is expressed in seconds counted from midnight of 1st January 1970, which is also called Epoch.

Any instant before the Epoch has a negative value. The units of instant are always seconds.

For conversion into other units such as years, days, hours, and minutes, one can use the class LocalDate Time.

Some of the methods of Instant class are listed in table.

Method	Description
isAfter	Compare two time instants Instant.now() isAfter Instant.now().minusHours(1);
isBefore	Compare two time instants Instant.now() isBefore Instant.now().plusHours(1);
plus	Adds time to Instant
minus	Substact time from Insatnt
until	Returns how much time exits between two time Instant objects

```
import java.time.Instant;
    public class Demo1 {
      public static void main(String[] args) {
        Instant instant = Instant.now();
        System.out.println(instant);
      }
    }
```

```
OUTPUT:
2017-02-03T06:11:01.194Z
```

```
import java.time.Instant;
public class Demo
{
  public static void main(String[] args)
  {
    Instant ins = Instant.now();
    System.out.println("The present time is "+ins);
    Instant  b = Instant.now().minusMinutes(30);
    System.out.println("The time before"+b);
     Instant  a = Instant.now().plusMinutes(40);
    boolean x = a.isAfter(b);
   }
}
```

```
OUTPUT:
The present time is 2021-05-26T09:57:10.193Z
The time before 2021-05-26T09:47:10.285Z
true
```

4.1.16 Formatting for Date/Time in Java

Java Date and Time

Display Current Date

To display the current date, import the java.time.LocalDate class, and use its now() method:

```
import java.time.LocalDate;
// import the LocalDate class
public class Main1 {
  public static void main(String[] args) {
    LocalDate myObj = LocalDate.now();
// Create a date object
    System.out.println(myObj);
// Display the current date
  }
}
```

OUTPUT:
Current Date

Display Current Time

To display the current time (hour, minute, second, and nanoseconds), import the java.time.LocalTime class, and use its now() method:

```
import java.time.LocalTime;
// import the LocalTime class
public class Main2 {
  public static void main(String[] args) {
    LocalTime myObj = LocalTime.now();
    System.out.println(myObj);
  }
}
```

OUTPUT:

Current Time

Display Current Date and Time

To display the current date and time, import the java.time.LocalDateTime class, and use its now() method:

```
import java.time.LocalDateTime; // import the LocalDateTime class
public class Main3 {
  public static void main(String[] args) {
    LocalDateTime myObj = LocalDateTime.now();
    System.out.println(myObj);
  }
}
```

OUTPUT:

```
Current Date & Time
```

Formatting Date and Time

The "T" in the example above is used to separate the date from the time.

You can use the DateTimeFormatter class with the ofPattern() method in the same package to format or parse date-time objects.

The following example will remove both the "T" and nanoseconds from the date-time:

```
import java.time.LocalDateTime; // Import the LocalDateTime class
import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class
public class Main4 {
  public static void main(String[] args) {
    LocalDateTime myDateObj = LocalDateTime.now();
    System.out.println("Before formatting: " + myDateObj);
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy
                                                     HH:mm:ss");
    String formattedDate = myDateObj.format(myFormatObj);
    System.out.println("After formatting: " + formattedDate);
  }
}
```

OUTPUT:

```
Before Formatting: 2021-05-26T14:45:53.489996
After Formatting: 26-05-2021 14:45:53
```

4.1.17 Temporal Adjusters Class

The methods of this class are used to determine things such as first day of month and last date of year among others.

The java.time.temporal defines an interface by name TemporalAdjuster that defines methods that take a temporal values and return an adjusted temporal value according to a user’s specifications.

The package also defines a class TemporalAdjusters, which contains predifined adjusters that may be directly used by a programmer.

firstDayofMonth()
lastDayofMonth()
firstDayofYear()
lastDayofYear()
firstDayofWeek()
lastDayofWeek()
firstInMonth(DayofWeek.Monday)
lastInMonth(DayofWeek.FRIDAY)

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
public class Main5 {
    public static void main(String[] args) {
        LocalDate localDate = LocalDate.now();
        System.out.println("current date : " + localDate);
        LocalDate with = localDate.with(TemporalAdjusters.firstDayOfMonth());
        System.out.println("firstDayOfMonth : " + with);
        LocalDate with1 = localDate.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("lastDayOfMonth : " + with1);
        LocalDate with2 = localDate.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
        System.out.println("next monday : " + with2);
        LocalDate with3 = localDate.with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println("firstDayOfNextMonth : " + with3);
    }
}
```

OUTPUT:

```
current date : 2021-05-26
firstDayOfMonth : 2021-05-01
lastDayOfMonth : 2021-05-31
next monday : 2021-05-31
firstDayOfNextMonth : 2021-06-01
```

Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throwable, Unchecked Exceptions, Checked Exceptions, try-with-resources, Catching Subclass Exception, Custom Exceptions, Nested try and catch Blocks, Rethrowing Exception, Throws Clause.

4.2 Exception Handling

4.2.1 Introduction

What is an Exception?

- **An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error.**
- **In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.**
- **Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.**

Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on

- **A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.**
- **When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.**
- **That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.**
- **Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.**
- **Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.**
- **Briefly, here is how they work.**
- **Program statements that you want to monitor for exceptions are contained within a try block.**
- **If an exception occurs within the try block, it is thrown.**
- **Your code can catch this exception (using catch) and handle it in some rational manner.**
- **System-generated exceptions are automatically thrown by the Java runtime system.**
- **To manually throw an exception, use the keyword throw.**
- **Any exception that is thrown out of a method must be specified as such by a throws clause.**

- **Any code that absolutely must be executed after a try block completes is put in a finally block.**

This is the general form of an exception-handling block:

```
try {
        // block of code to monitor for errors
  }
catch (ExceptionType1 exOb) {
  // exception handler for ExceptionType1
  }
catch (ExceptionType2 exOb) {
  // exception handler for ExceptionType2
  }
  // ...
finally {
        // block of code to be executed after try block ends
  }
```

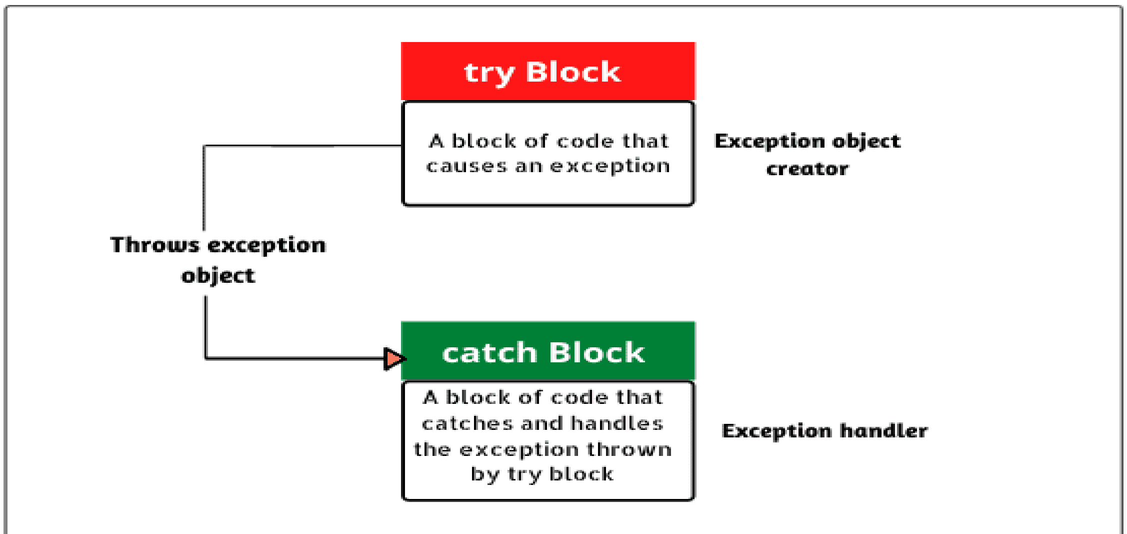
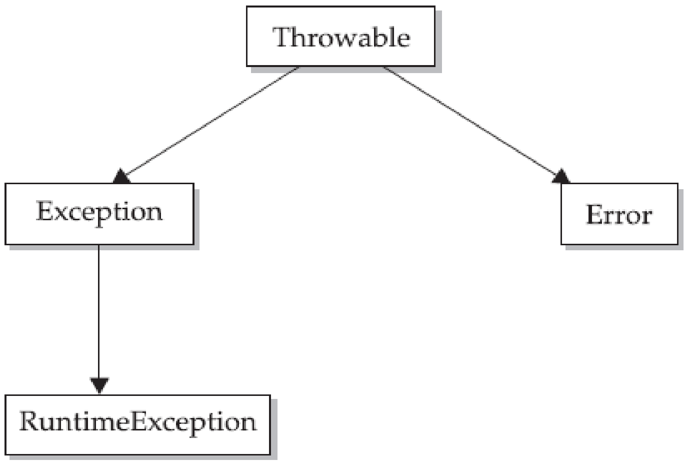

try Block
A block of code that causes an exception
Exception object creator
Throws exception object
catch Block
A block of code that catches and handles the exception thrown by try block
Exception handler

Fig: Exception handling mechanism

4.2.2 Hierarchy of Standard Exception Classes

The top-level exception hierarchy is shown here:
All exception types are subclasses of the built-in class **Throwable.** Thus, Throwable is at the top

of the exception class hierarchy.



- **Immediately below Throwable are two subclasses that partition exceptions into two distinct branches.**
- **One branch is headed by Exception.**
- **This class is used for exceptional conditions that user programs should catch such as division by zero and invalid array indexing.**
- **There is an important subclass of Exception, called RuntimeException.**
- **The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program.**
- **Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.**
- **Stack overflow is an example of such an error.**

Java's Built-in Exceptions:

Inside the standard package java.lang, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type RuntimeException. These exceptions need not be included in any method's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions

Types of exceptions

There are two types of exceptions in Java:
1)Checked exceptions
2)Unchecked exceptions

Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions. (Added by JDK 7.)

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Uncaught Exceptions:

- Before we learn how to handle exceptions in your program, it is useful to see what happens when we don't handle them.
- This small program includes an expression that intentionally causes a divide-by-zero error:

  class Exc0

```
  {
          public static void main(String args[])
          {
               int d = 0;
               int a = 42 / d;    // divide by zero (Exception)
          }
     }
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
          java.lang.ArithmeticException: / by zero
          at Exc0.main(Exc0.java:6)
```

Notice how the class name, Exc0; the method name, main; the filename, Exc0.java; and the line number, 4, are all included in the simple stack trace.

Also, notice that the type of exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened.

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from main( ):

```
class Exc1
{
     static void subroutine()
     {
          int d = 0;
```

```
                int a = 10 / d;
        }
        public static void main(String args[])
        {
                Exc1.subroutine();
        }
}
```

```
//Error
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:6)
at Exc1.main(Exc1.java:10)
```

4.2.3 Keywords of exception handling

Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

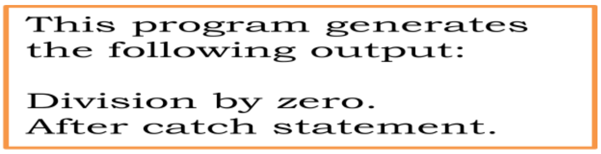
Syntax of try block

```
try{
   //statements that may cause an exception
}
```

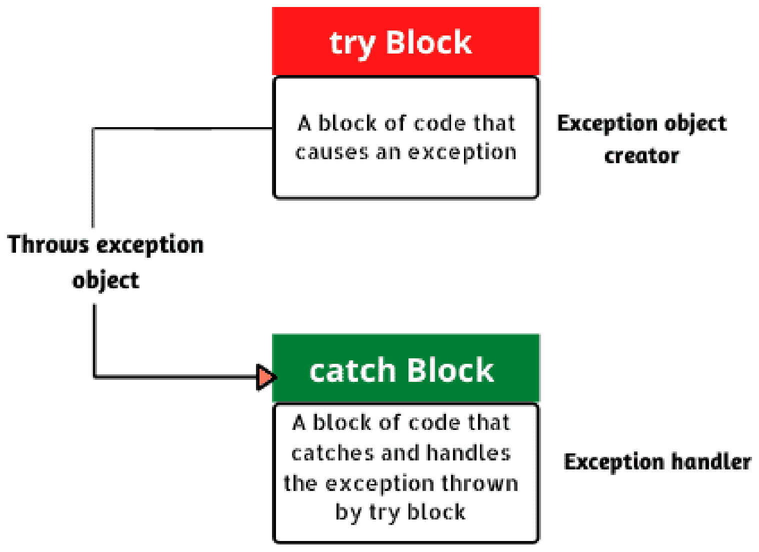
- Although the default exception handler provided by the Java run-time system is useful for debugging.
- But we usually want to handle an exception ourselves.
- Doing so provides two benefits.
- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.
- To handle a run-time error, simply enclose the code that we want to monitor inside a try block.
- Immediately following the try block, include a catch clause that specifies the exception type that we wish to catch.

- To illustrate how easily this can be done, the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error

```
class Exc2 {
        public static void main(String args[]) {
                int d, a;
                try {   // monitor a block of code.
                        d = 0;
                        a = 42 / d;
                   System.out.println("This will not be printed.");
                }
                catch (ArithmeticException e)
                {   // catch divide-by-zero error
                   System.out.println("Division by zero.");
                }
        System.out.println("After catch statement.");
        }
}
```



```
This program generates
the following output:

Division by zero.
After catch statement.
```


try Block
A block of code that causes an exception
Exception object creator
Throws exception object
catch Block
A block of code that catches and handles the exception thrown by try block
Exception handler

- Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try / catch mechanism.

Important point about try and catch

- A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described shortly).
- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.)
- We cannot use try on a single statement.

Example try and catch:

```
import java.util.Random;
class Exc3 {
public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();
    for(int i=0; i<100; i++) {
        try {
            b = r.nextInt();
                System.out.println("b: " + b);
            c = r.nextInt();
                System.out.println(“c: " + c);
            a = 12345 / (b/c);        }
    catch (ArithmeticException e)
     {
      System.out.println("Division by zero.");
      a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
      }
     }
    }
```

We can display this description in a println( ) statement by simply passing the exception as an argument.

For example, the catch block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
        System.out.println("Exception: " + e);
        a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-byzero error displays the following message:

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

Another example of try and catch:

```
class Exc5 {
        public static void main(String args[])
        {
                try {
                        int [ ] num = {1, 2, 3};
                        System.out.println(num[10]);
                    System.out.println("This will not be printed.");
                }
```

```
                    catch (ArrayIndexOutOfBoundsException e) {
                          System.out.println("Exception: " + e);
                    or  e.printStackTrace();
                    }
          System.out.println("After catch statement.");
          }
}
```

```
This program generates the following output:

Exception: java.lang.
ArrayIndexOutOfBoundsException
After catch statement.
```

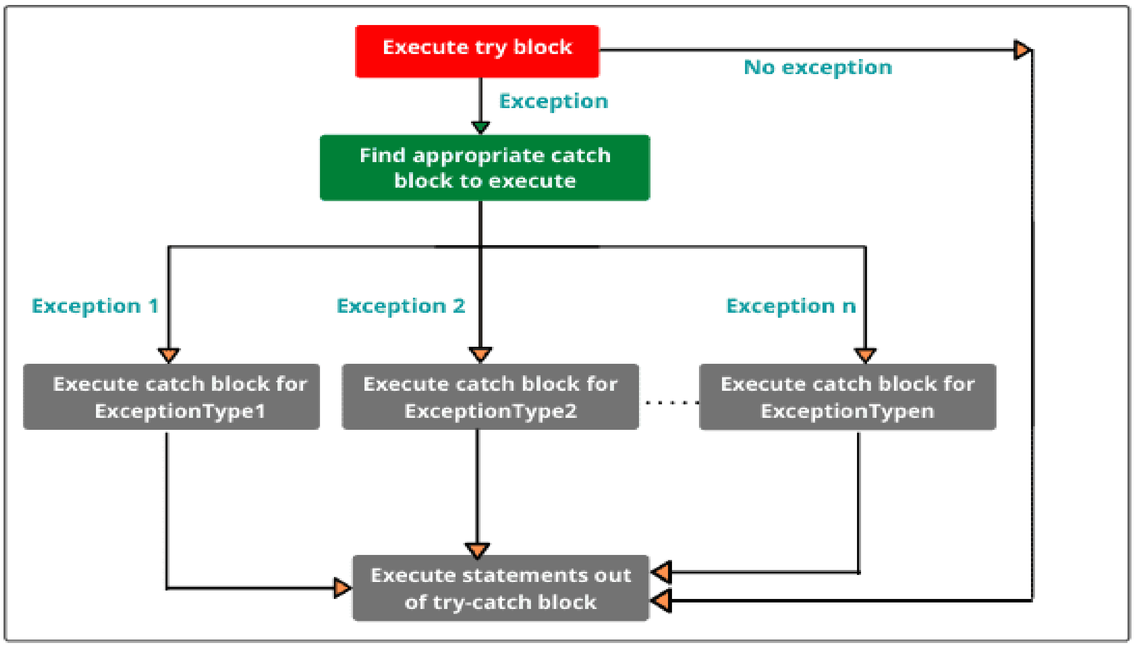
Multiple catch Clauses:

In some cases, more than one exception could be raised by a single piece of code.

To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.

After one catch statement executes, the others are bypassed, and execution continues after the try / catch block.


Execute try block
No exception
Exception
Find appropriate catch block to execute
Exception 1
Exception 2
Exception n
Execute catch block for ExceptionType1
Execute catch block for ExceptionType2
Execute catch block for ExceptionTypen
Execute statements out of try-catch block

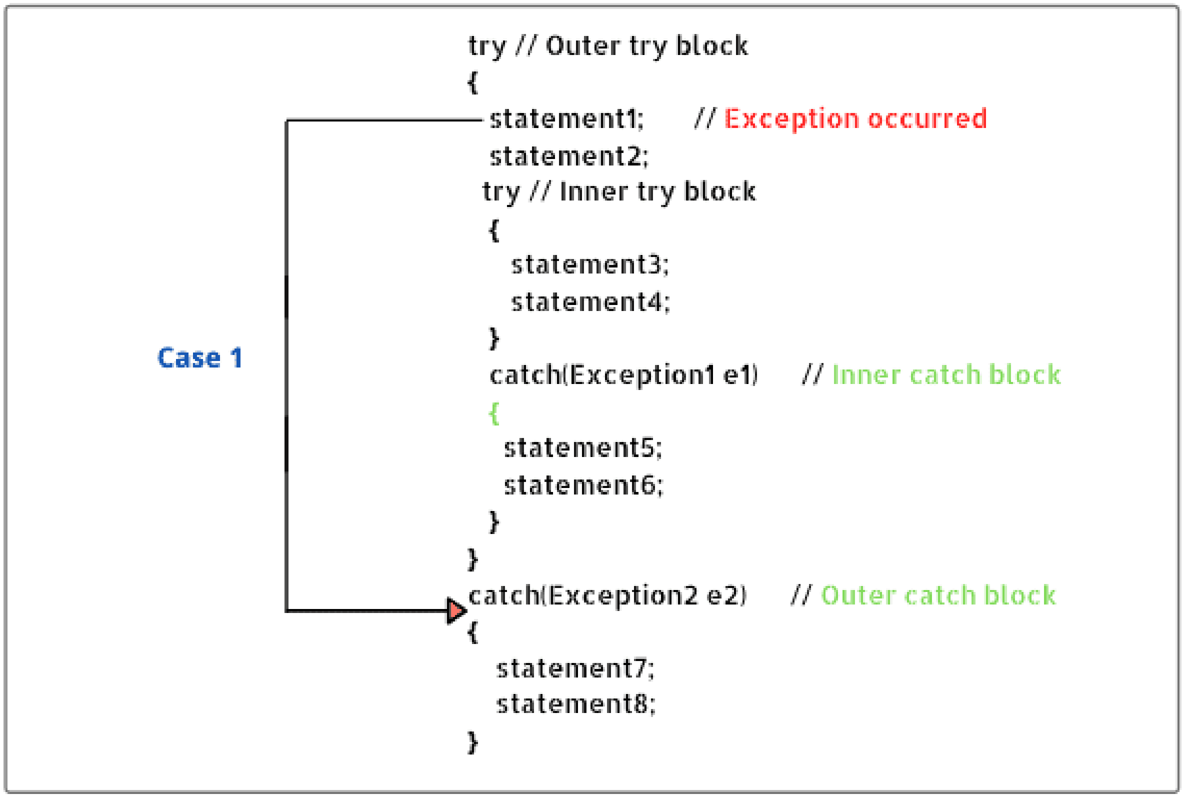
```
class Exc6
{
    public static void main(String args[])
    {
      try {
              int a = args.length;
```

```
        System.out.println("a = " + a);
        int b = 42 / a;
        int c[ ] = { 1 };
        c[42] = 99;
      }
catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catchblocks.");
   }
}
```

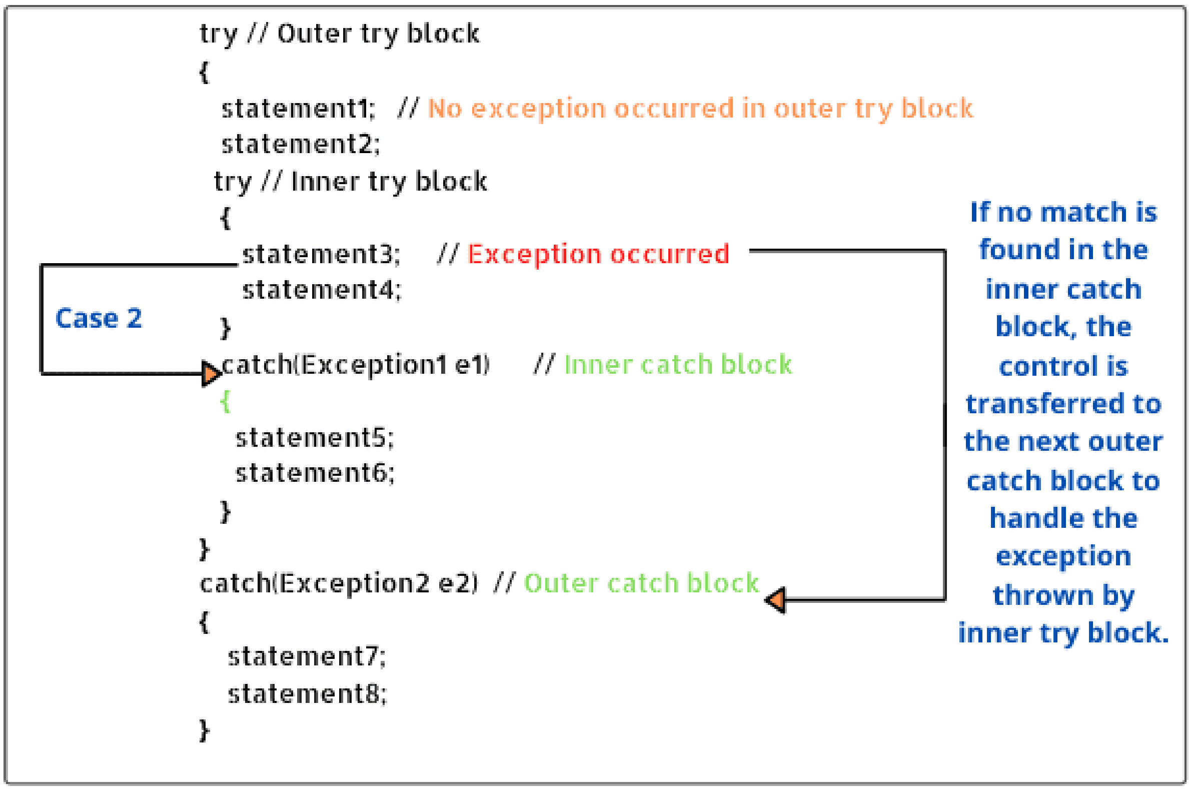
- This program will cause a division-by-zero exception if it is started with no commandline arguments, since a will equal zero.
- It will survive the division if you provide a command line argument, setting a to something larger than zero.
- But it will cause an ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

Nested try Statements:

- The try statement can be nested. That is, a try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement’s catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.



```
try // Outer try block
{
  statement1;     // Exception occurred
  statement2;
 try // Inner try block
 {
    statement3;
    statement4;
 }
 catch(Exception1 e1)     // Inner catch block
 {
   statement5;
   statement6;
 }
}
catch(Exception2 e2)     // Outer catch block
{
   statement7;
   statement8;
}
```



```
try // Outer try block
{
  statement1;  // No exception occurred in outer try block
  statement2;
 try // Inner try block
 {
    statement3;   // Exception occurred
    statement4;
 }
 catch(Exception1 e1)     // Inner catch block
 {
    statement5;
    statement6;
 }
}
catch(Exception2 e2)  // Outer catch block
{
   statement7;
   statement8;
}
```

If no match is found in the inner catch block, the control is transferred to the next outer catch block to handle the exception thrown by inner try block.

class Exc9 {

public static void main(String args[]) {

try {

```
                int a = args.length;
                int b = 42 / a;
                System.out.println("a = " + a);
                try {    // nested try block
        if(a==1) a = a/(a-a);  // division by zero
        if(a==2) {
                int c[] = { 1 };
                c[42] = 99;
                // generate an out-of-bounds exception
                }
        }
catch(ArrayIndexOutOfBoundsException e)
{
        System.out.println(e);
    }
}
catch(ArithmeticException e) {
        System.out.println(e);
   }
  }
}
```

throw Keywords:

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for your program to throw an exception explicitly, using the throw statement.
- The general form of throw is shown here:

  throw *ThrowableInstance*;

- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions

- There are two ways you can obtain a Throwable object: using a parameter in a catch clause or creating one with the new operator.
- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.

- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on.

If no matching catch is found, then the default exception handler halts the program and prints the stack trace

```
public class Exc12
{  static void checkAge(int age)
   {   try{  if (age < 18) {
    throw new ArithmeticException();    }
     else
System.out.println("Access granted - You are old enough!");     }
   catch(ArithmeticException e) {
     System.out.println("Access denied - You must be at least 18 years old.");
    }
 }
public static void main(String[] args)
{
   int Age = 15;
   System.out.println("Age of a person: " + Age);
   checkAge(Age);
  }
}
```

OUTPUT:

```
Age of a person: 15
Access denied - You must be at least 18
years old.
```

throws Keyword:

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a throws clause in the method's declaration.
- A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Example:

```
class Exc14 {
        static void throwOne() throws IllegalAccessException {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
  public static void main(String args[]) {
```

```
            try {throwOne();     } catch (IllegalAccessException e) {
        System.out.println("Caught " + e);   }
   }
}
```

```
OUTPUT:  inside throwOne
caught java.lang.IllegalAccessException: demo
```

Important points to remember about throws keyword:

- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
- By the help of throws keyword we can provide information to the caller of the method about the exception.

Another example on throws keyword:

```
class Exc18
{     static void method(int a) throws ArithmeticException,
                                          ArrayIndexOutOfBoundsException
      {    int b = 42 / a;
           int c[ ] = { 1 };
           c[42] = 99;       }
public static void main(String args[])
     {   try {    int a = args.length;
           System.out.println("a = " + a);
            method(a);        } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
            }          catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index oob: " + e);
            }
}
}
```

Throw vs throws:

S.No	throw	throws
1	**Used to throw an exception for a method**	**Used to indicate what exception type may be thrown by a method**
2	**Cannot throw multiple exceptions**	**Can declare multiple exceptions**
3	**Syntax: throw is followed by an object (new type) used inside the method**	**Syntax: throws is followed by a class and used with the method signature**

finally Keyword:

- finally creates a block of code that will be executed after a try /catch block has completed and before the code following the try/catch block.

- The finally block will execute whether or not an exception is thrown.
- The finally clause is optional.

```
public class Exc19 {
  public static void main(String[] args) {
    try {
            int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]);
    }
    catch (Exception e) {
            System.out.println("Exception is caught: "+ e);
    }
    finally {
            System.out.println("The 'try & catch' is finished.");
     }
  }
}
```

OUTPUT:

```
Exception is caught:
java.lang.ArrayIndexOutOfBoundsExcep
tion: 10
The 'try & catch' is finished.
```

- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- Any time a method is about to return to the caller from inside a try/catch block, the finally clause is also executed just before the method returns.
- However, each try statement requires at least one catch or a finally clause.

```
class Exc20
{
    static void procA() {     // Through an exception out of the method.
            try {
                    System.out.println("inside procA");
                    throw new RuntimeException("demo");
                }
  Finally
 {
System.out.println("procA's finally");
  }
    }
    static void procB() {     // Return from within a try block.
            try {
                   System.out.println("inside procB");
                    return;
                }
 finally {
```

```
                System.out.println("procB's finally");
            }
        }
        static void procC() {      // Execute a try block normally.

    try {
                    System.out.println("inside procC");
        }
    finally
    {
    System.out.println("procC's finally");
      }
        }
   public static void main(String args[])
   {
        try {
                procA();
            }
        catch (Exception e) {
                System.out.println("Exception caught");
        }

        procB();
        procC();
   }
}
```

Rethrowing Exception:

Sometimes we may need to rethrow an exception in Java. If a catch block cannot handle the particular exception it has caught, we can rethrow the exception.

```
class Demo
{
   static void method()
   {
      try
      {
         throw new NullPointerException("demo");
      }
      catch(NullPointerException e)
      {
         System.out.println("Exception Caught inside method().");
         throw e;        // rethrowing the exception
      }
   }
 public static void main(String args[])
   {
      try
      {
         method();
      }
      catch(NullPointerException e)
      {
        System.out.println("Exception Caught in main.");
      }
    }
}
```

```
OUTPUT:
Caught inside method().
Caught in main.
```

Catching Subclass Exception:

- 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.

```
import java.io.*;
   class Parent{
     void msg(){  System.out.println("parent");  }
   }
   class Exc22 extends Parent{
     void msg()throws IOException{
       System.out.println("Child");
     }
     public static void main(String args[]){
      Parent p=new Exc22();
      p.msg();
     }
   }
```

```
OUTPUT:
Exc22.java:7: error: msg() in Exc22
cannot override msg() in Parent
     void msg()throws IOException{
          ^
  overridden method does not throw
IOException
1 error
```

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

```
import java.io.*;
class Parent{
void msg(){  System.out.println("parent");  }
}
class Exc23 extends Parent{
void msg()throws ArithmeticException{
System.out.println("child");
}
public static void main(String args[]){
Parent p=new Exc23();
p.msg();
}
}
```

OUTPUT:

```
child
```

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

Class Throwable:

- The Throwable class is the superclass of all errors and exceptions in the Java language.
- Java Throwable class provides several methods.
  - getMessage()
  - getSuppressed()
  - getStackTrace()
  - fillInStackTrace()
  - getLocalizedMessage()
  - initCause()
  - getCause()
  - printStackTrace()

The getMessage() method of Java Throwable class is used to get a detailed message of the Throwable.

```
        public class Exc24 {
                public static void main(String[] args)throws Throwable {
                        try{
                        int i=10/0;
                        }
                        catch(Throwable t){
                        System.out.println(t.getMessage());
                        }
                }
        }
```

printStackTrace()

- The printStackTrace() method of Java Throwble class is used to print the Throwable along with other details like classname and line number where the exception occurred.

```
import java.lang.Throwable;
public class Exc25 {
   public static void main(String[] args) throws Throwable {
  try{
     int i=4/0;
  }catch(Throwable e){
     e.printStackTrace();
     System.err.println("Cause : "+e.getCause());
  }
}
}
```

```
OUTPUT:
java.lang.ArithmeticException: / by zero
      at Exc25.main(Exc25.java:5)
Cause : null
```

try-with-resources:

```
import java.io.*;
class Exc26 {
  public static void main(String[] args) {
    String line;
    try(BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {
      while ((line = br.readLine()) != null) {
        System.out.println("Line =>"+line);
      }
    } catch (IOException e) {
      System.out.println("IOException in try block =>" + e.getMessage());
    }
  }
}
```

OUTPUT: if test.txt file not exist in current directory

```
IOException in try block =>test.txt
(The system cannot find the file specified)
```

OUTPUT: if test.txt file exist in current directory

```
Line =>Hello KHIT
Line =>This is CSE
```

Custom Exceptions:

- 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);   //Optional
        }
}
class Exc28 {
         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 occured: "+m);}

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

OUTPUT:

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