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 like java.awt. 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.

The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

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

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

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

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));
        }
}
```

Step 1: Save the above file with Addition.java

```
package pack;

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));
           }
}
```

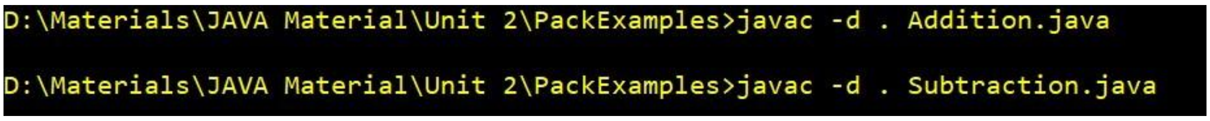
Step 2: Save the above file with Subtraction.java

Step 3: Compilation

To compile the java files use the following commands

```
javac  -d   directory_path     name_of_the_java file
Javac –d    .     name_of_the_java file
```

Note: -d is a switching options creates a new directory with package name. Directory path represents in which location you want to create package and . (dot) represents
current working directory.



```
D:\Materials\JAVA Material\Unit 2\PackExamples>javac -d . Addition.java

D:\Materials\JAVA Material\Unit 2\PackExamples>javac -d . Subtraction.java
```

Step 4: Access package from another package

There are three ways to use package in another package:

1. With fully qualified name.

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

2. import package.classname;

```
import  pack.Add;
import  pack.Sub;
 class TestPack
{
        public static void main(String arg[])
        {
                 Add a=new  Add(10,15);
                a.sum();
                Sub  s=new Sub(20,15);
                s.diff();
        }
}
```

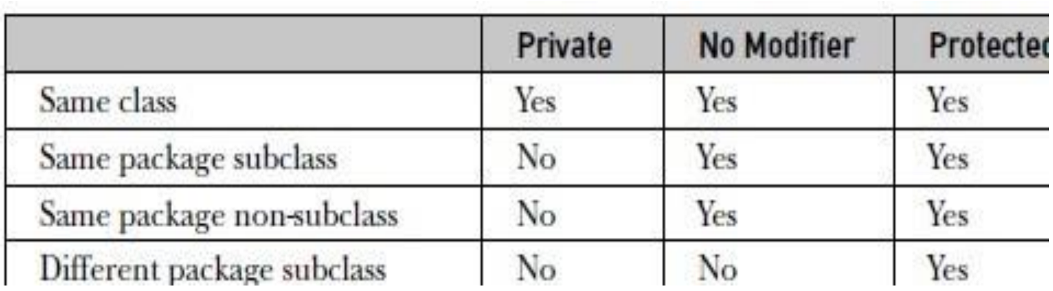
3. import package.*;

```
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();
        }
}
```

Note: Don’t place Addition.java, Subtraction.java files parallel to the pack directory. If you place JVM searches for the class files in the current working directory not in the pack directory.

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:
  1. Visible to the class only (private).
  2. Visible to the package (default). No modifiers are needed.
  3. Visible to the package and all subclasses (protected)
  4. Visible to the world (public)



	Private	No Modifier	Protecte
Same class	Yes	Yes	Yes
Same package subclass	No	Yes	Yes
Same package non-subclass	No	Yes	Yes
Different package subclass	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, and SamePackage.

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_publl = " + 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
```

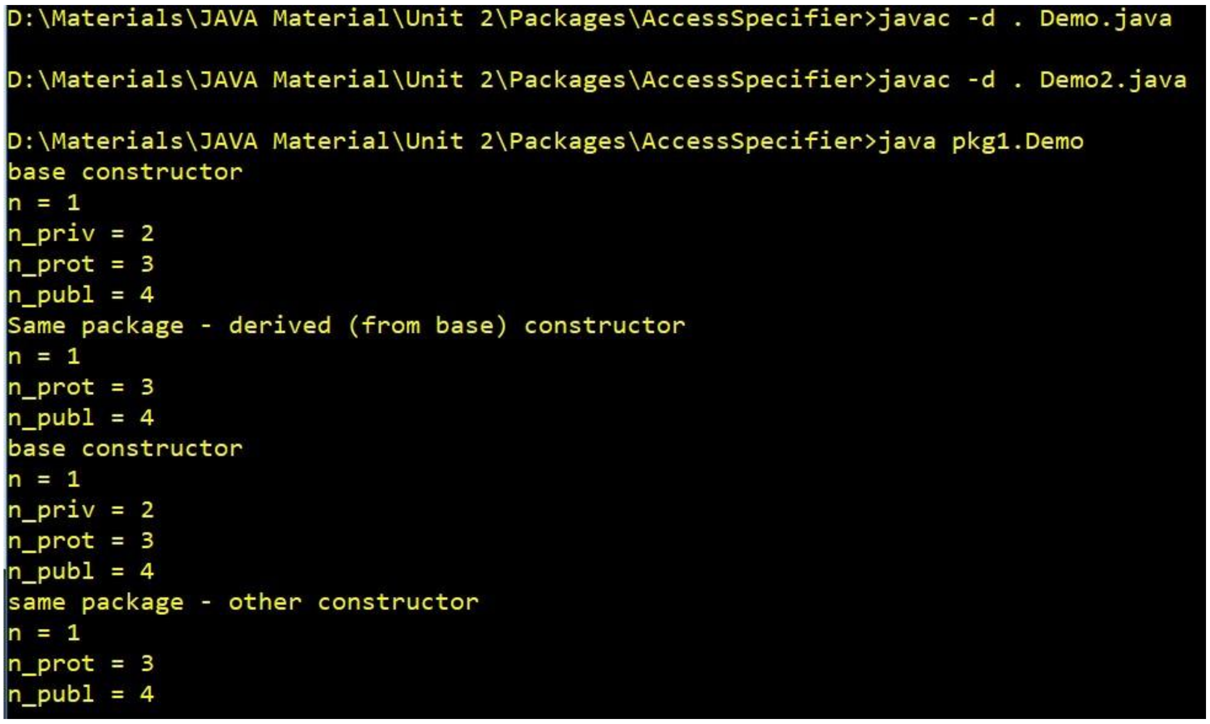
```
         *  System.out.println("n_prot = " + pro.n_prot); */

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

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

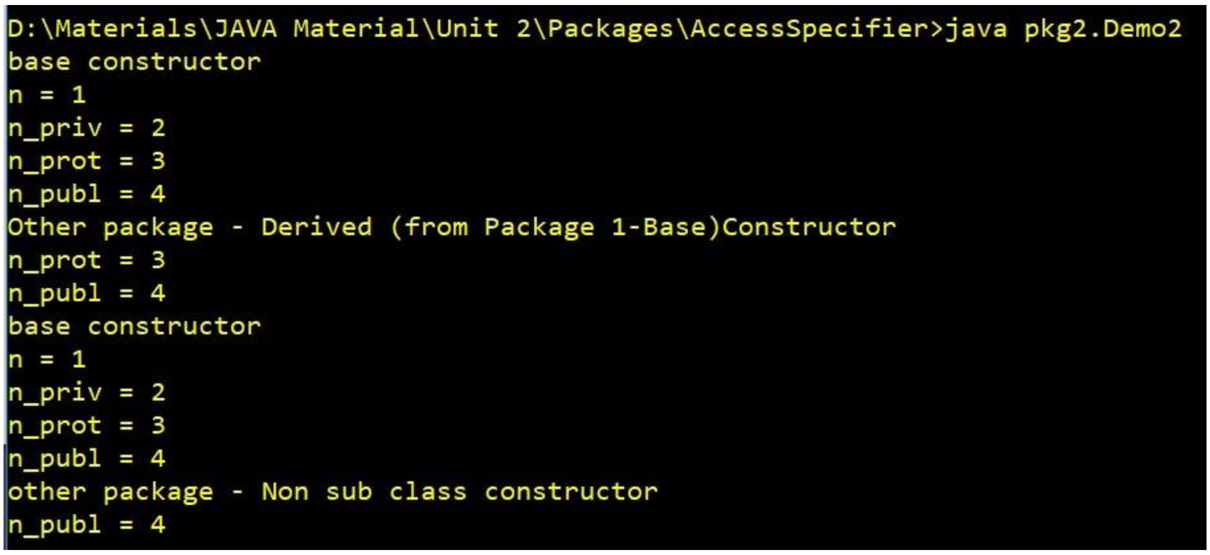
```
/* demo package pkg1 */

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();
   }
}
```



```
D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>javac -d . Demo.java

D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>javac -d . Demo2.java

D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>java pkg1.Demo
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
Same package - derived (from base) constructor
n = 1
n_prot = 3
n_publ = 4
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
same package - other constructor
n = 1
n_prot = 3
n_publ = 4
```



```
D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>java pkg2.Demo2
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
Other package - Derived (from Package 1-Base)Constructor
n_prot = 3
n_publ = 4
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
other package - Non sub class constructor
n_publ = 4
```

Exceptions in Java

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

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:

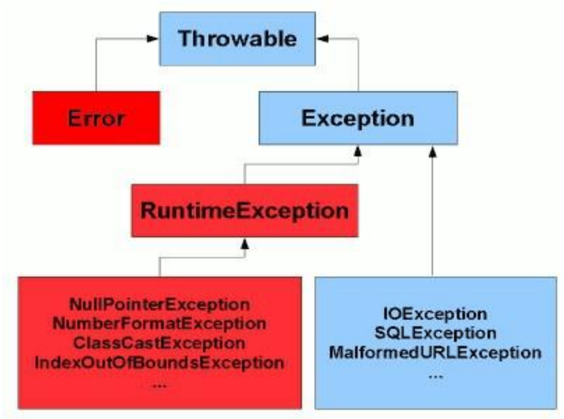
```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed after try block ends
}
```

Here, ExceptionType is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

Exception Types

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. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

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.



Uncaught Exception:

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;
        }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because 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. Any exception that is not caught by your program will ultimately be processed by the default handler. 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:4)
```

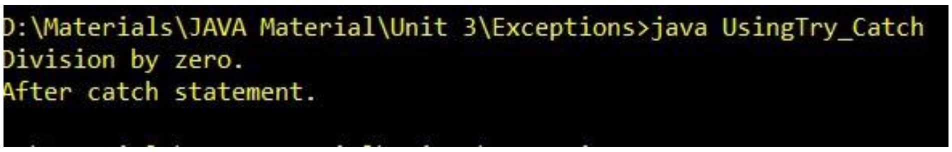
Using try and catch:

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

Example:

```
class UsingTry_Catch
{
        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.");
        }
}
```

Output:



```
D:\Materials\JAVA Material\Unit 3\Exceptions>java UsingTry_Catch
Division by zero.
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.
- If one catch statement is executed, the others are bypassed, and execution continues after the try / catch block.
- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their super classes. This is because a catch statement that uses a super class will catch exceptions of that type plus any of its subclasses. Subclass would never be reached if it came after its super class.
- A subclass must come before its super class in a series of catch statements. If not

  unreachable code will be created and a compile time error will result.

Example:

```
// Demonstrate multiple catch statements.
class MultipleCatches
{
        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(Exception e)
                {
                        System.out.println("Array index out of bounds: " + e);
                }
                System.out.println("After try/catch blocks.");
        }
}
```

Nested try Statements

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

Syntax:

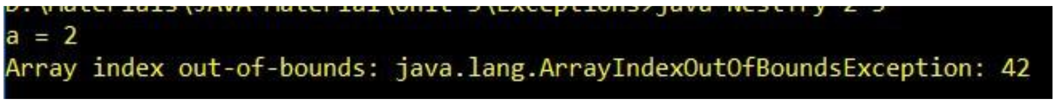
```
try{
        try  {
        statement 1;
        statement 2;
        try
         {
            statement 1;
             statement 2;
          }
        catch(Exception e)      {      }
   }

}
catch(Exception e){}
```

- 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 (default handler).

Example:

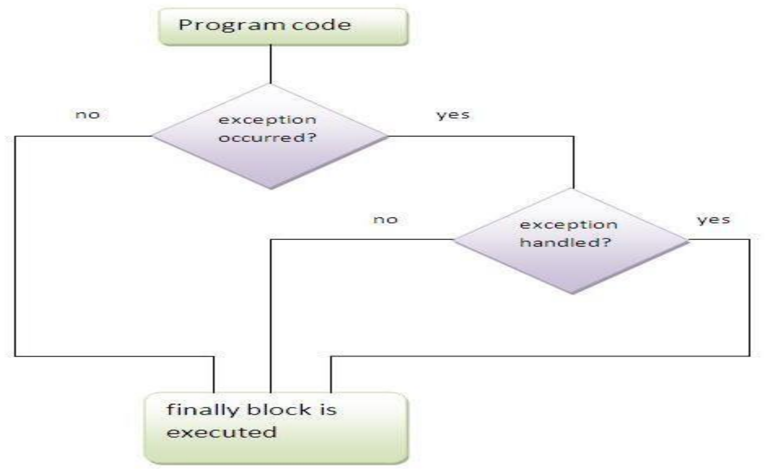
```
class NestTry
{
    public static void main(String args[])
    {
            try
            {
                    int a = args.length;
                    /* If no command-line args are present, the following statement will
                    generate a divide-by-zero exception. */

                    int b = 42 / a;
                    System.out.println("a = " + a);

                    try { // nested try block
                            /* If one command-line arg is used, then a divide-by-zero
                          exception         will be generated by the following code. */

                            if(a==1)
                                    a = a/(a-a); // division by zero

                        /* If two command-line args are used, then generate an out-of-
                        bounds  exception. */
                            if(a==2)
                            {
                                    int c[] = { 1 };
                                    c[42] = 99; // generate an out-of-bounds exception
                            }
                    }
                    catch(ArrayIndexOutOfBoundsException e)
                    {
                            System.out.println("Array index out-of-bounds: " + e);
                    }
            }
            catch(ArithmeticException e)
            {
                    System.out.println("Divide by 0: " + e);
            }
    }
}
```



```
D:\Materials\JAVA Material\Unit 3\Exceptions>java NestTry 2 3
a = 2
Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException: 42
```

finally block:

- Java finally block is a block that is used to execute important code such as closing connections (databases, network, disks, commit in databases) etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.



Example 1:

```
class Finally_Case1   //exception not occured
{
        public static void main(String args[])
        {
            try{
                    int data=25/25;
                    System.out.println(data);
            }
            finally
            {
                        System.out.println("finally block is always executed");
            }
            System.out.println("rest of the code...");
    }
}
```

Example 2:

```
class Finally_Case2   //exception occured and not handled.
{
        public static void main(String args[])
        {
            try{
                    int data=25/0;
                    System.out.println(data);
            }
            catch(ArrayIndexOutOfBoundsException e)
```

```
                {
                        System.out.println(e);
                  }
                finally
                {
                        System.out.println("finally block is always executed");
                }
                System.out.println("rest of the code...");
            }
        }
```

Example 3:

```
        class Finally_Case3   //exception occured and handled.
        {
                public static void main(String args[])
                {
                  try{
                        int data=25/0;
                        System.out.println(data);
                  }
                  catch(ArithmeticException e)
                  {
                        System.out.println(e);
                    }
                  finally
                  {
                        System.out.println("finally block is always executed");
                  }
                  System.out.println("rest of the code...");
            }
        }
```

<u>Use of throw</u>

- 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.

- There are two ways to obtain a Throwable instance:
  - creating one with the new operator

    throw new exception_class("error message");
  - using the parameter in catch clause - throw exception;

- 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.

  Syntax:

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

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.*/

public class UseofThrow
{
      static void validate(int age)
     {
            try{
                    if(age<18)
                       throw new ArithmeticException("not valid\n");
                    else
                            System.out.println("Welcome to participate in voting");
            }
            catch(ArithmeticException e)
            {
                    System.out.println(e);
                    throw e;
            }
      }
     public static void main(String args[])
     {
            validate(13);
            System.out.println("rest of the code...");
     }
}
```

Exceptions and its types:

There are two types of exceptions.

1. Unchecked exceptions
2. Checked exceptions

Unchecked Exceptions: Found during running of a program

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.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Checked Exceptions: Found at compilation time

ClassNotFoundException	Class not found.
IllegalAccessException	Access to a class is denied.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating user-defined exception:

- We can also create our own exception by creating a sub class simply by extending java Exception class.
- Define a constructor for Exception sub class (not compulsory) and override the toString() method to display customized message in catch clause.
  - Exception();
  - Exception(parameter);
- The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Syntax of toString():

- String toString( )

  - Returns a String object containing a description of the exception.

  - This method is called by println( ) when outputting a Throwable object.

  - belongs to Object class

Example:

```
class MyException extends Exception
{
        String s ;
        MyException( String s)
        {
                this.s=s;
        }
        public String toString()
        {
            return ("User Defined " +s) ;
        }
}
class UserDefinedException
{
        public static void main(String args[])
        {
          try{
                        throw new MyException("Exeption"); // throw is used to create
                        //a new exception and throw it.
          }
```

```
                    catch(MyException e)
                    {
                             System.out.println(e) ;
                    }
                  }
          }
```

Threading in JAVA

Multithreading in java

is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

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

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

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

Multitasking

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

- Process-based Multitasking(Multiprocessing)

- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

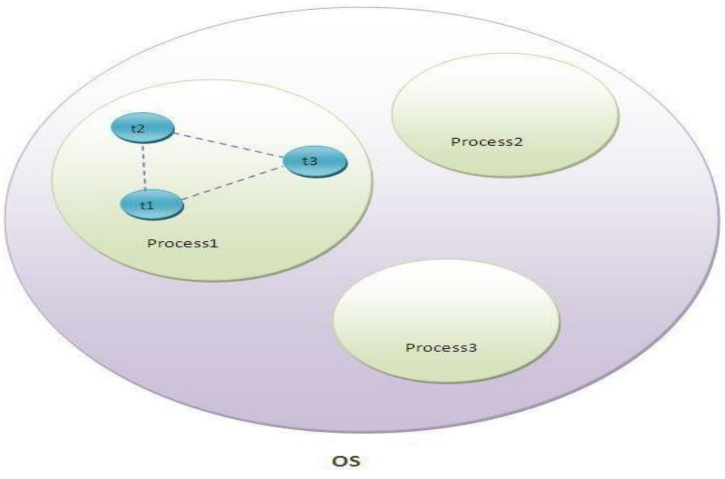
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

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

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

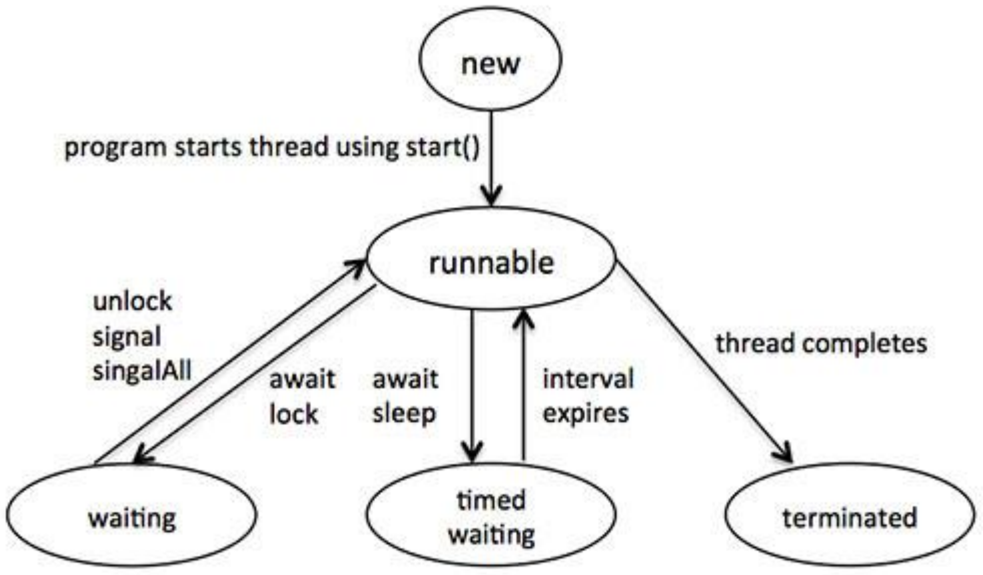
Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



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

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting: Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Ways of Creating Threads

I) Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing Runnable interface. You will need to follow three basic steps:

Step 1: As a first step you need to implement a run() method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

```
public void run( )
```

Step 2: At second step you will instantiate a Thread object using the following constructor: Thread(Runnable threadObj, String threadName);

Where, threadObj is an instance of a class that implements the Runnable interface andthreadName is the name given to the new thread.

Step 3: Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

void start( );

NOTE: Follow Example Program in notes

II) **Create Thread by Extending Thread Class**:

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override run( ) method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

```
void start( );
```

NOTE: Follow Example Program in notes

Thread class Methods:

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.

7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

SN	Methods with Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack()

	Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

isAlive () and join() Methods :

```
final boolean isAlive()
```

The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

```
final void join() throws InterruptedException

final void join(long milliseconds) throws
InterruptedException
```

This method waits until the thread on which it is called terminates.

Using join(), we tell thread to wait until the specified thread completes its execution.

There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

Example for isAlive() :

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                        Thread.sleep(500);
                }
        catch(InterruptedException ie)
        {
                System.out.println("Interrupted");

        }
        System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
                t2.start();
                System.out.println(t1.isAlive());
                System.out.println(t2.isAlive());
        }
}
```

<u>Example Program using join():-</u>

```
public class MyThreadJTest extends Thread
{

public void run()
        {
        System.out.println("r1 ");
        try {
                Thread.sleep(500);
                }
      catch(InterruptedException ie)
        {
                  System.out.println("Interrupted");
              }
        System.out.println("r2 ");
        }

public static void main(String[] args)
        {
                MyThreadJTest t1=new MyThreadJTest();
                MyThreadJTest t2=new MyThreadJTest();
                t1.start();

                try
                  {
                      t1.join();  //Waiting for t1 to finish
                }catch(InterruptedException ie)
                {System.out.println(" main thread Interrupted");
                 }

                t2.start();
        }
}
```

Creation of Multiple Threads :

The following example demonstrates how to create multiple threads.

Example program for multiple thread creation :

```
            class NewThread implements Runnable
   {
  String name;
  Thread t;
  NewThread(String n)
  {
    name=n;
    t=new Thread(this.name);
    System.out.println("Child Thread:"+t);
    t.start();
  }
  public void run()
  {
    try
    {
      for(int i=5;i>0;i--)
      {
        System.out.println(name+":"+i);
          Thread.sleep(500);
      }
     }
     catch(InterruptedException e)
     {
         System.out.println(name+"Interrupted");
     }
      System.out.println(name+"Terminated!!");
   }
}
class ThreadDemo
{
  public static void main(String args[])
  {
    new NewThread("One");
    new NewThread("Two");
    new NewThread("Three");
    try
    {
        Thread.sleep(1000);
    }
    catch(InterruptedException e)
    {
      System.out.println("Main Thread Interrupted");
    }
    System.out.println("Main Thread Terminated");
   }
}
```

Java synchronization:

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Synchronization in Java:

Synchronization in java is the capability to control the access of multiple threads to any shared resource. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization: The synchronization is mainly used to

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

Thread Synchronization

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

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.

    NOTE: Follow Example Program Given in Notes and give explanation for it

2. Cooperation (Inter-thread communication in java)

Inter-thread communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

1) **wait() method**

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

2**) notify() method**

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

3) **notifyAll() method**

Wakes up all threads that are waiting on this object's monitor.

Syntax:

public final void notifyAll()

Example : Follow Exercise 7 c and it’s additional Program