1. Inner Class

1.1. Class inside Class

- It is possible to define a class within another class; such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- There are two types of nested classes: static and non-static. A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
- The most important type of nested class is the inner class.
- An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

Example.

```
// Demonstrate an inner class.
class Outer
{
        int outer_x = 100;
        void test()
         {
        Inner inner = new Inner();
        inner.display();
        }
// this is an inner class
        class Inner {
                        void display()
                        {
                        System.out.println("display: outer_x = " + outer_x);

                        }

                }

}

class InnerClassDemo
 {
public static void main(String args[])
 {
Outer outer = new Outer();
```

```
outer.test();
}
}
```

- An inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

Example:

```
class Outer
{
    int outer_x = 100;
    void test()
    {
    Inner inner = new Inner();
    inner.display();
    }

// this is an inner class
    class Inner
    {
    int y = 10; // y is local to Inner
    void display()
                {
    System.out.println("display: outer_x = " + outer_x);
                    }
     }

void showy() {
System.out.println(y); // error, y not known here!
              }
}

class InnerClassDemo
 {
public static void main(String args[])
{
Outer outer = new Outer();
outer.test();
}
}
```

1.2. method local inner class

- Inner class can be defined within method of outer class

```
class Outer {
    int outer_x = 100;
    void test()
    {
            for(int i=0; i<10; i++) {
                        class Inner {
                        void display() {
                        System.out.println("display: outer_x = " + outer_x);
                                    }
                            }
                Inner inner = new Inner();
                inner.display();
                    }
}
}
class InnerClassDemo {
public static void main(String args[])
{
Outer outer = new Outer();
outer.test();
}
}
```

1.3. anonymous inner class

- Java anonymous inner class is an inner class without a name and for which only a single object is created.
- In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface.

- Anonymous class using interface

```
// Java program to demonstrate Need for
// Anonymous Inner class

// Interface
interface Age {

    // Defining variables and methods
    int x = 21;
    void getAge();
}

// Class 1
// Helper class implementing methods of Age Interface
class MyClass implements Age {

    // Overriding getAge() method
     public void getAge()
    {
            // Print statement
            System.out.print("Age is " + x);
    }
}

// Class 2
// Main class
// AnonymousDemo
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
            // Class 1 is implementation class of Age interface
            MyClass obj = new MyClass();

            // calling getage() method implemented at Class1
            // inside main() method
            obj.getAge();
    }
}
```

In the above program, interface Age is created with getAge() method and x=21. Myclass is written as an implementation class of Age interface. As done in Program, there is no need to

write a separate class Myclass. Instead, directly copy the code of Myclass into this parameter, as shown here:

```
// Java Program to Demonstrate Anonymous inner class

// Interface
interface Age {
    int x = 21;
    void getAge();
}

// Main class
class AnonymousDemo {

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

        // Myclass is hidden inner class of Age interface
        // whose name is not written but an object to it
        // is created.
        Age oj1 = new Age() {

            public void getAge()
            {
                // printing age
                System.out.print("Age is " + x);
            }
        };

        oj1.getAge();
    }
}
```

- Anonymous class using abstract class

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
```

```
 };
 p.eat();
 }
}
```

2. Package

2.1. Use of Package

- Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- The package is both a naming and a visibility control mechanism.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are only exposed to other members of the same package.
- Defining a package
    - To create a package is quite easy: simply include a package command as the first statement in a Java source file. The package statement defines a name space in which classes are stored.

      package pkg;

    - If you omit the package statement, the class names are put into the default package, which has no name.
    - Command
        - Javac –d . [filename].java
        - java package2.[filecontaining main method]
    - d

2.2. CLASSPATH

- How does the Java run-time system know where to look for packages that you create?
    - First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
    - Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
    - package MyPack;
        - In order for a program to find MyPack, one of two things must be true. Either the program is executed from a directory immediately above MyPack, or CLASSPATH must be set to include the path to MyPack.
        - The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the .class files into IT.

2.3. Import statements

- There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package.
- Java includes the import statement to bring certain classes, or entire packages, into visibility.
    - import java.util.Date;
    - import java.io.*;

2.4. Access control

- Refer UNIT-1

3. Exception Handing

3.1. Exception and Error

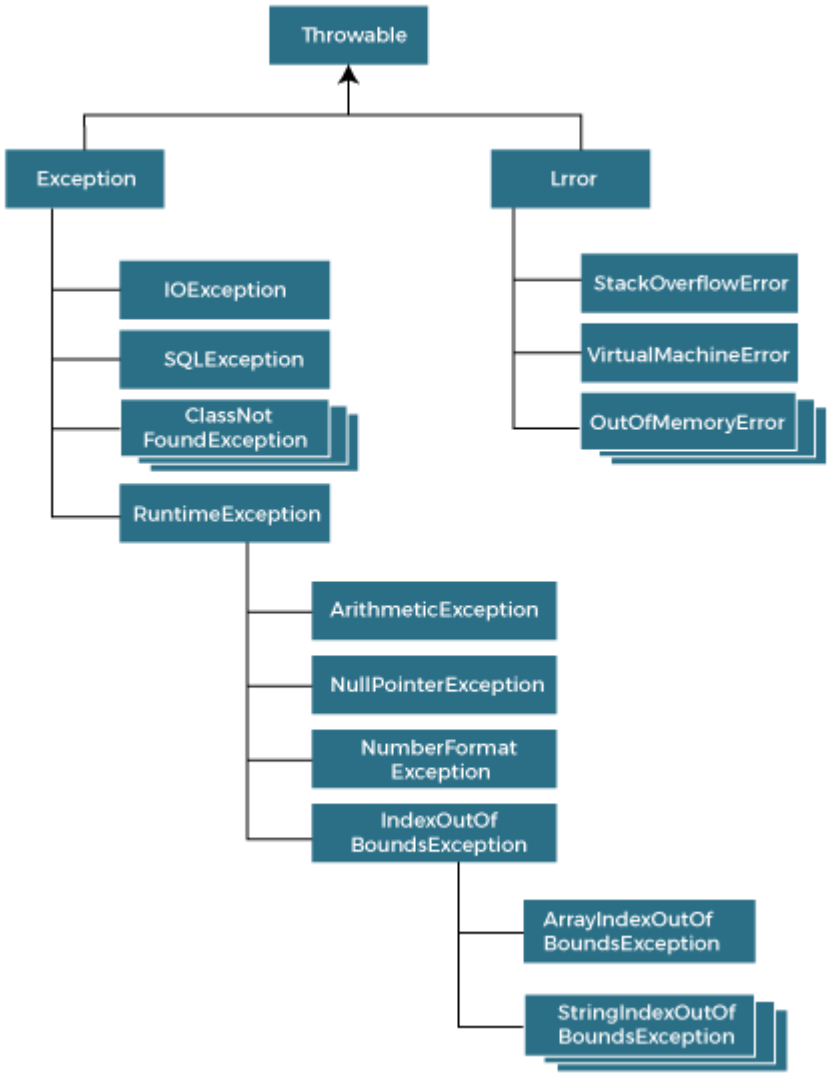
- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
    - Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
    - Manually generated exceptions are typically used to report some error condition to the caller of a method.
- Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.

```
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 before try block ends
}
```

- Exception Types
    - All exception types are subclasses of the built-in class Throwable.
    - Throwable is at the top of the exception class hierarchy.

- One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. 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.



- Types of Java Exceptions
  - There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:
  - Checked Exception
    - The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.
  - Unchecked Exception

- The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

- Error
  - Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

3.2. use of try

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

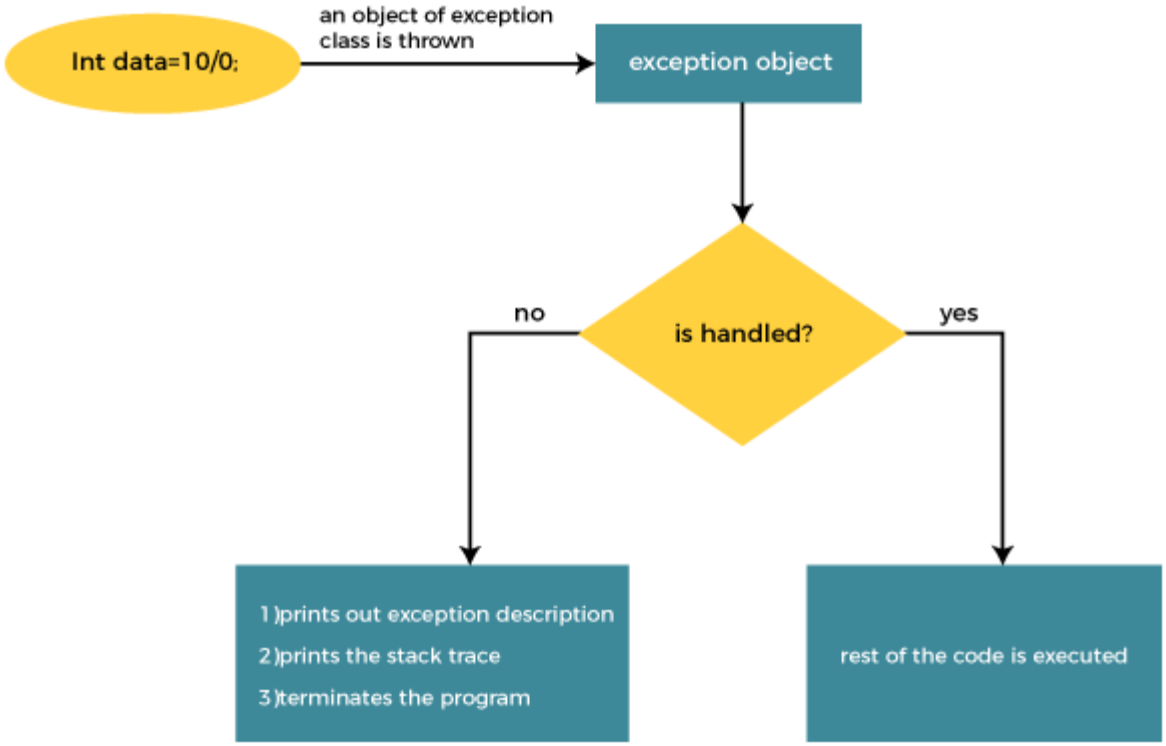
Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

3.3. Catch

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only. You can use multiple catch block with a single try block.
- The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:
  - Prints out exception description.
  - Prints the stack trace (Hierarchy of methods where the exception occurred).
  - Causes the program to terminate.

- But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.



Problem without exception handling

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

```
public class TryCatchExample1 {

        public static void main(String[] args) {

            int data=50/0; //may throw exception

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

}
```

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

Solution by exception handling

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

```
public class TryCatchExample2 {

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

}
```

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

Example:

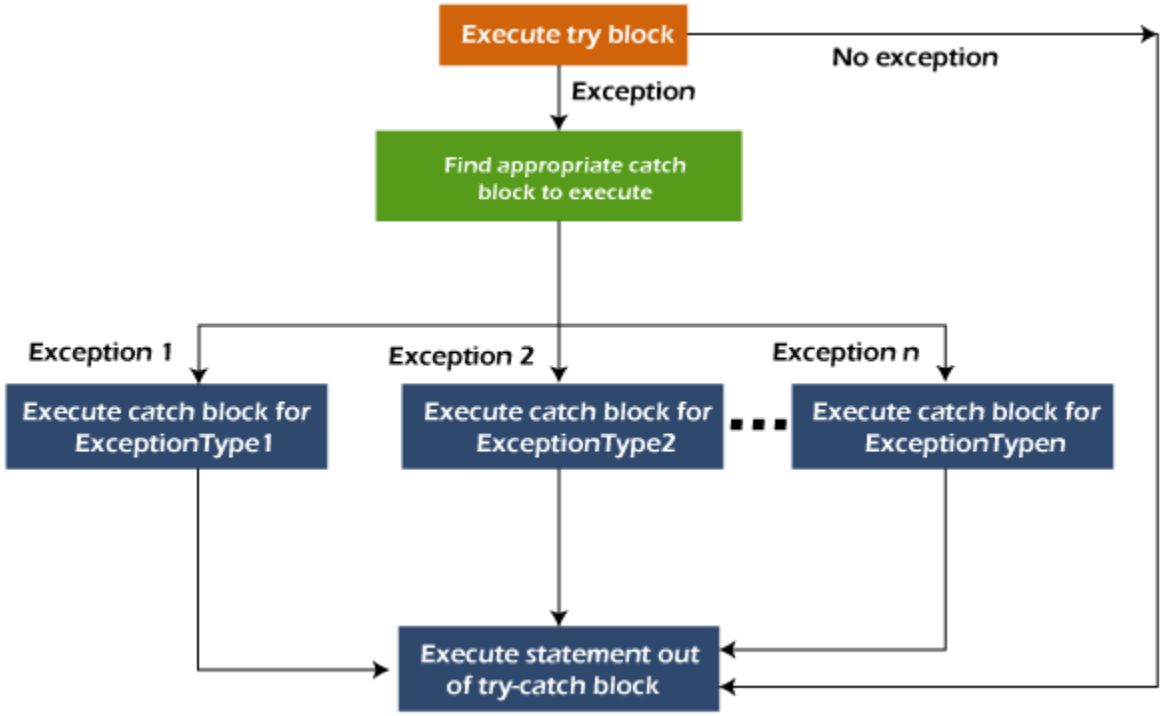
```
public class TryCatchExample9 {

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

}
```

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

- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.



Example:

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
```

```
   }
}
```

Output:
Arithmetic Exception occurs
rest of the code

```
public class MultipleCatchBlock2 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Output:
ArrayIndexOutOfBounds Exception occurs
rest of the code

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

```
public class MultipleCatchBlock3 {

   public static void main(String[] args) {

       try{
           int a[]=new int[5];
           a[5]=30/0;
```

```
            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
      }
    }
```

Output:
Arithmetic Exception occurs
rest of the code

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

```
public class MultipleCatchBlock4 {

  public static void main(String[] args) {

      try{
          String s=null;
          System.out.println(s.length());
         }
         catch(ArithmeticException e)
           {
            System.out.println("Arithmetic Exception occurs");
           }
         catch(ArrayIndexOutOfBoundsException e)
           {
            System.out.println("ArrayIndexOutOfBounds Exception occurs");
           }
         catch(Exception e)
           {
            System.out.println("Parent Exception occurs");
```

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

Output:
Parent Exception occurs
rest of the code

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

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

Output:
Compile-time error

3.4. throw
3.5. throws and finally
3.6. Built in Exception
3.7. Custom exception
3.8. Throwable Class Runtime Stack Mechanism
3.9. nested try