UNIT-3 (partial)

Prepared By:
Dr. Deepak Kumar Sharma
Asst. Professor (SG)
SoCS UPES Dehradun

UNIT-III

Unit III: Nested Classes, Exceptions, Multithreading & IO Streams

Nested Classes, Types of Nested Classes, Exception Handling, Exception Handlers, Concurrent Programming, The Thread Class and Runnable Interface, Thread Priorities, Synchronization, Java's I/O Streams, Byte Streams and Character Streams, FileWriter, FileReader.

Java Nested and Inner Class

- Nested Class: Defining a class within another class.

```
class OuterClass {
    // ...
    class NestedClass {
        // ...
    }
}
```

Two types of nested classes can be created in Java:

- Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class
- Static nested class

Non-Static Nested Class (Inner Class)

- Non-static nested classes are known as inner classes.
- It has access to members of the enclosing class (outer class).
- *Must instantiate the outer class first, in order to instantiate the inner class.*

Need of inner class:

- Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

Advantage of Java inner classes

- Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
- Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
- **Code Optimization**: It requires less code to write.

Java Member Inner class

- A non-static class that is created inside a class but outside a method is called **member inner class**.
- It is also known as a **regular inner class**.
- It can be declared with access modifiers like public, default, private, and protected.

Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

```
class TestMemberOuter{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter obj=new TestMemberOuter();
//to create an object of the member inner class
//OuterClassReference.new MemberInnerClassConstructor();
TestMemberOuter.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Output: data is 30

Java Anonymous inner class

- Java anonymous inner class is an inner class without a name and for which only a single object is created.
- An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, *without having to actually subclass a class*.
- Java Anonymous inner class can be created in two ways:
  - Class (may be abstract or concrete).
  - Interface

Example- Java Anonymous inner class

```
class Animal {
     void makeSound(){}
}

public class TestAnonymousInner {
    public static void main(String args[]) {
      //Anonymous Inner Class
        Animal animal = new Animal() {
            void makeSound() {
                System.out.println("Check Animal ");
            }
        };

        animal.makeSound();
    }
}
```

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

Internal Working:

- A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
- An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Output: nice fruits

Java Local inner class

- A class i.e., created inside a method, is called local inner class in java.
- Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body.
- Sometimes this block can be a for loop, or an if clause.
- Local Inner classes are not a member of any enclosing classes.
- They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them.
- However, they can be marked as final or abstract.
- These classes have access to the fields of the class enclosing it.
- If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

```
public class localInner{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner obj=new localInner();
  obj.display();
 }
}
```

output:

```
30
```

Example- Inner Class

```
class CPU {
    double price;
    // nested class
    class Processor{
        // members of nested class
        double cores;
        String manufacturer;
        double getCache(){
            return 4.3;
        }
    }
    // nested protected class
    protected class RAM{
        // members of protected nested class
        double memory;
        String manufacturer;
        double getClockSpeed(){
            return 5.5;
        }
    }
}
```

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

        // create object of Outer class CPU
        CPU cpu = new CPU();

       // create an object of inner class Processor using outer class
        CPU.Processor processor = cpu.new Processor();

        // create an object of inner class RAM using outer class CPU
        CPU.RAM ram = cpu.new RAM();
        System.out.println("Processor Cache = " + processor.getCache());
        System.out.println("Ram Clock speed = " + ram.getClockSpeed());
    }
}
```

Output:

```
Processor Cache = 4.3
Ram Clock speed = 5.5
```

Accessing Members of Outer Class within Inner Class

```
class Car {
    String carName;
    String carType;
    // assign values using constructor
    public Car(String name, String type) {
        this.carName = name;
        this.carType = type;
    }
    // private method
    private String getCarName() {
        return this.carName;
    }
// inner class
    class Engine {
        String engineType;
        void setEngine() {
           // Accessing the carType property of Car
            if(Car.this.carType.equals("4WD")){
                // Invoking method getCarName() of Car
                if(Car.this.getCarName().equals("Crysler")) {
                    this.engineType = "Smaller";
                } else {
                    this.engineType = "Bigger";
                }
            }else{
                this.engineType = "Bigger";
            }
        }
        String getEngineType(){
            return this.engineType;
        }
    }
}
```

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

// create an object of the outer class Car
        Car car1 = new Car("Mazda", "8WD");
        // create an object of inner class using the outer class
        Car.Engine engine = car1.new Engine();
        engine.setEngine();
        System.out.println("Engine Type for 8WD= " + engine.getEngineType());

        Car car2 = new Car("Crysler", "4WD");
        Car.Engine c2engine = car2.new Engine();
        c2engine.setEngine();
        System.out.println("Engine Type for 4WD = " + c2engine.getEngineType());
    }
}
```

- can access the members of the outer class by using **“this”** keyword.

Static Nested Class

- A static class inside another class.
- Static nested classes are not called static inner classes.
- Unlike inner class, a static nested class cannot access the member variables of the outer class. It is because the **static nested class** doesn't require you to create an instance of the outer class.
- It cannot access non-static data members and methods.
- It can access static data members of the outer class, including private.
- Static nested classes can include both static and non-static fields and methods.
- To access the static nested class, we don't need objects of the outer class.

Note: In Java, only nested classes are allowed to be static.

Static Nested class

```
class TestOuter1{
 static int data=30;

  static class Inner{
  void msg(){System.out.println("data is "+data);}
 }

public static void main(String args[]){

TestOuter1.Inner obj=new TestOuter1.Inner();

 obj.msg();
 }
}
```

Output:

```
data is 30
```

Static Nested Class

```
class Animal {

// inner class
  class Reptile {
    public void displayInfo() {
      System.out.println("I am a reptile.");
    }
  }

// static class
  static class Mammal {
    public void displayInfo() {
      System.out.println("I am a mammal.");
    }
  }
}
```

```
class Main {
  public static void main(String[] args) {
    // object creation of the outer class
    Animal animal = new Animal();

    // object creation of the non-static class
    Animal.Reptile reptile = animal.new Reptile();
    reptile.displayInfo();

    // object creation of the static nested class
    Animal.Mammal mammal = new Animal.Mammal();
    mammal.displayInfo();

  }
}
```

Output

```
I am a reptile.
I am a mammal.
```

Java static nested class example with a **static method**

```
public class TestOuter{
  static int data=30;
  static class Inner{
   static void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter.Inner.msg();//no need to create the instance of static nested class
  }
}
```

CheckPoint: Accessing Non-static members

```
class Animal {
  static class Mammal {
   public void displayInfo() {
     System.out.println("I am a mammal.");
   }
  }

 class Reptile {
   public void displayInfo() {
     System.out.println("I am a reptile.");
   }
 }

 public void eat() {
   System.out.println("I eat food.");
 }
}
```

```
class Main {
 public static void main(String[] args) {
   Animal animal = new Animal();
   Animal.Reptile reptile = animal.new Reptile();
   reptile.displayInfo();

   Animal.Mammal mammal = new Animal.Mammal();
   mammal.displayInfo();
   mammal.eat();
 }
}
```

OUTPUT:

```
Main.java:28: error: cannot find symbol
    mammal.eat();
          ^
  symbol:   method eat()
  location: variable mammal of type Mammal
1 error
compiler exit status 1
```

Note: static nested classes can only access the class members (static fields and methods) of the outer class.

CheckPoint?

```
static class Animal {
 public static void displayInfo() {
   System.out.println("I am an animal");
 }
}

class Main {
 public static void main(String[] args) {
   Animal.displayInfo();
 }
}
```

Output

```
Main.java:1: error: modifier static not allowed here
static class Animal {
       ^
1 error
compiler exit status 1
```

Java Nested Interface

- An interface, i.e., declared within another interface or class, is known as a nested interface.
- The nested interfaces are used to group related interfaces so that they can be easy to maintain.
- The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

Syntax of nested interface

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

← **within the interface**

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

← **within the class**

Example of nested interface which is declared within the interface

```
public interface OuterInterface {
    void outerMethod();

    // Nested interface declared within OuterInterface
    interface NestedInterface {
        void nestedMethod();
    }
}

public class MyClass implements OuterInterface.NestedInterface {
    @Override
    public void nestedMethod() {
        System.out.println("Nested interface method implementation");
    }

    public static void main(String[] args) {
        MyClass myClass = new MyClass();
        myClass.nestedMethod();
    }
}
```

Output:

```
Nested interface method implementation
```

Example of nested interface which is declared within the class

```
class OuterClass {
    interface NestedInterface {
        void nestedMethod();
    }
}

class ImplementNestedInterface implements OuterClass.NestedInterface {
    public void nestedMethod() {
        System.out.println("Nested method implementation");
    }
}

public class Main {
    public static void main(String[] args) {
      ImplementNestedInterface instance = new ImplementNestedInterface();
      instance.nestedMethod();
    }
}
```

Output:

```
Nested interface method implementation
```

Can we define a class inside the interface?

```
interface M{
 class A{}
}
```

- Yes, if we define a class inside the interface, the Java compiler creates a **static nested class.**

```
interface MyInterface {
    void myMethod();

    static class MyStaticClass {
        void staticMethod() {
            System.out.println("This is a static nested class method.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyInterface.MyStaticClass staticObj = new MyInterface.MyStaticClass();
        staticObj.staticMethod(); // Outputs: This is a static nested class method.
    }
}
```

Exception Handling in Java

Exception Handling in Java

exception

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

Exception Handling:

- mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

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.

Advantages of Exception Handling

- to maintain the normal flow of the application.

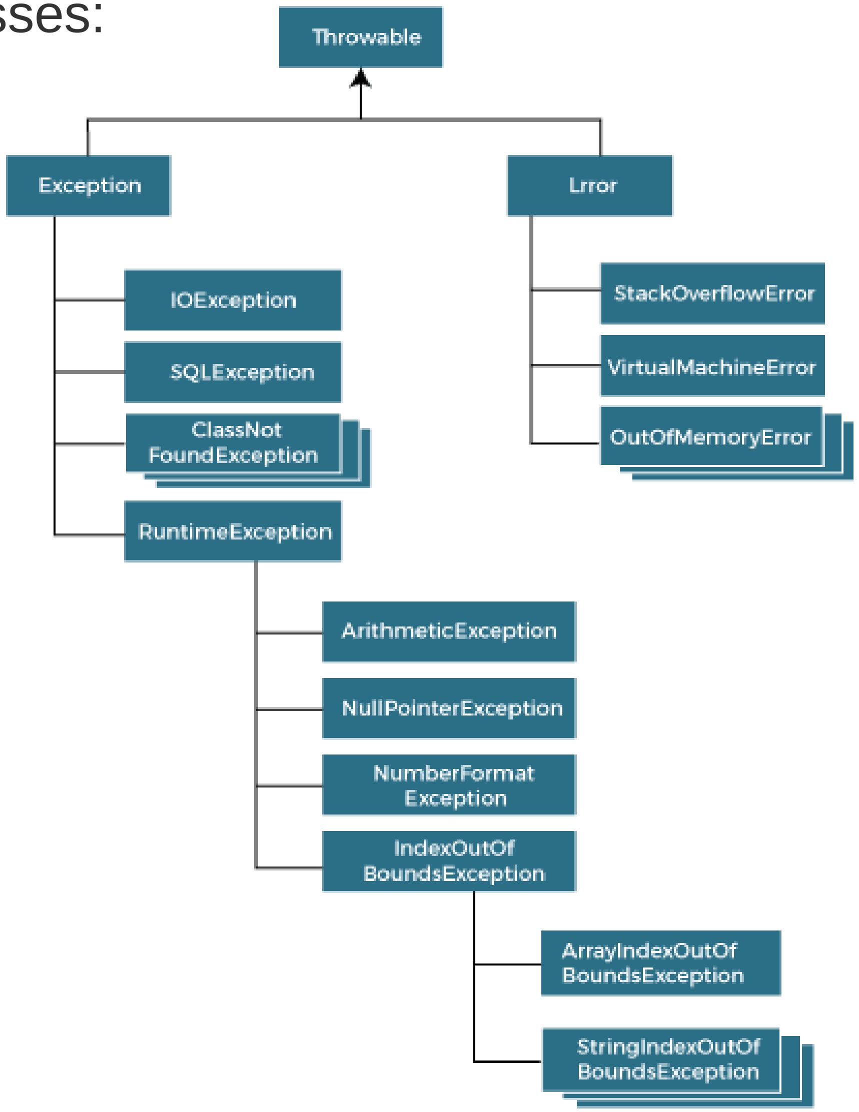
```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

- If an exception occurs at statement 5; the rest of the code will not be executed.
- exception handling in Java will ensure that remaining statements (6 to 10) are executed.

Hierarchy of Java Exception classes

- The **java.lang.Throwable** class is the root class of Java Exception hierarchy inherited by two subclasses:
  - Exception and Error

hierarchy of Java Exception classes



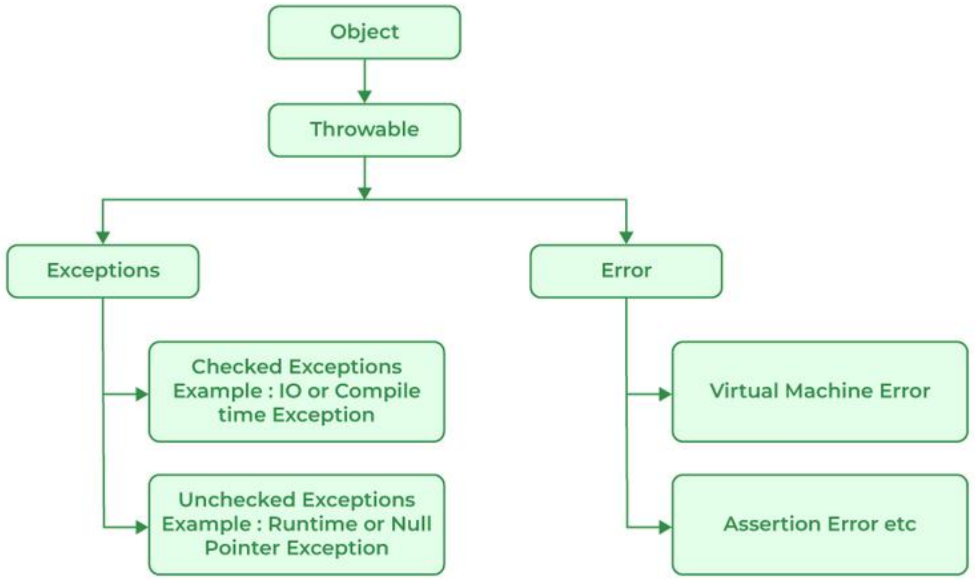
Types of Java Exceptions

1) 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.**

2) 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**.


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc

3) Error

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

Example: Checked and Unchecked

```
import java.io.*;

public class ExceptionExample {

    public static void main(String[] args) {

        // Example 1 of Unchecked Exception
        try {
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Unchecked Exception 1: Can't divide by zero!");
        }

        // Example 2 of Unchecked Exception
        try {
            String str = null;
            str.length(); // This will throw NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Unchecked Exception 2: Null value has no length!");
        }

        // Example 1 of Checked Exception
        try {
            FileReader fileReader = new FileReader("nonexistentfile.txt"); // This will throw FileNotFoundException
        } catch (FileNotFoundException e) {
            System.out.println("Checked Exception 1: File not found!");
        }

        // Example 2 of Checked Exception
        try {
            Thread.sleep(1000); // This will throw InterruptedException
        } catch (InterruptedException e) {
            System.out.println("Checked Exception 2: Thread interrupted!");
        }
    }
}
```

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

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

OUTPUT:

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

Common Scenarios of Java Exceptions

1) A scenario where **ArithmeticException** occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where **NullPointerException** occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

3) A scenario where **NumberFormatException** occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

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

4) A scenario where **ArrayIndexOutOfBoundsException** occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs.

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

Java try-catch block

Java try block

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

Syntax of Java try-catch

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

Syntax of try-finally block

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

Java catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception can 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.

NOTE:

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.

Problem without exception handling

```
public class TryCatchExample1 {

  public static void main(String[] args) {

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


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


  }

}
```

- all the code below the exception won't be executed.

Output:

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

Solution by exception handling

```
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 TryCatchExample3 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
                        // if exception occurs, the remaining statement will not exceute
        System.out.println("rest of the code");
        }
             // handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }

    }
}
```

Output:

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

Example

- Handling the exception using the parent class exception.

```
public class TryCatchExample {

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

}
```

Output:

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

Example

- to print a custom message on exception.

```
public class TryCatchExample {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
                // displaying the custom message
            System.out.println("Can't be divided by zero");
        }
    }

}
```

Output:

```
Can't be divided by zero
```

Example

- Resolving the exception in a catch block

```
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output:

```
25
```

Example

- Enclosing exception code in a catch block

```
public class TryCatchExample7 {

    public static void main(String[] args) {

        try
        {
        int data1=50/0; //may throw exception

        }
            // handling the exception
        catch(Exception e)
        {
           // generating the exception in catch block
        int data2=50/0; //may throw exception

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

Output:

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

enclose exception code within a try block and use catch block only to handle the exceptions.

Java Catch Multiple Exceptions

Points to Note:

- 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[3]=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
```

Example

```
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
```

```java
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");
   }
}
```

Predict output?

Output:

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

Note: Here 30/0 will throw exception before a[5]

Example

- Generated NullPointerException, but didn't provided 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
```

Example

- Handling 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 complet
ed");}
  System.out.println("rest of the code...");
 }
}
```

Output:

Predict output?

Compile-time error

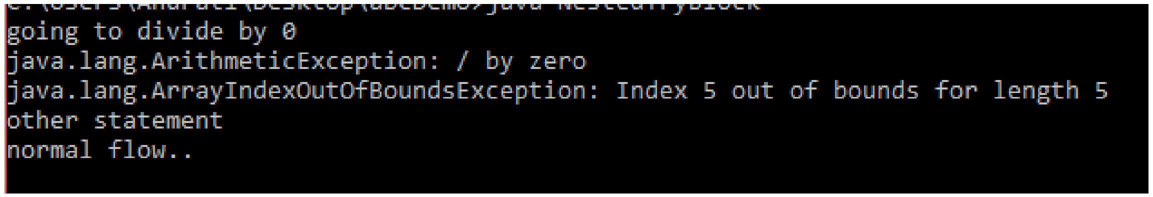
Java Nested try block

- A try block inside another try block -> nested try block

Scenario :

- the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException**
- while the **outer try block** can handle the **ArithemeticException**

```
public class NestedTryBlock{
 public static void main(String args[]){
 //outer try block
  try{
  //inner try block 1
    try{
     System.out.println("going to divide by 0");
     int b =39/0;
   }
    //catch block of inner try block 1
    catch(ArithmeticException e)  { System.out.println(e);  }
    //inner try block 2
    try{
    int a[]=new int[5];
     a[5]=4;
     }
    //catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)  {  System.out.println(e);  }
    System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)  {  System.out.println("handled the exception (outer catch)"); }
  System.out.println("normal flow..");
 }   }
```



```
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

Java finally block

- **Java finally block** is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or not.
- The finally block follows the try-catch block.

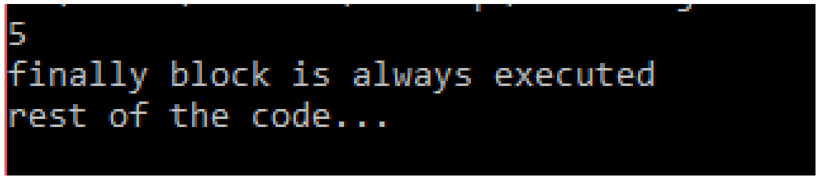
Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Case 1: When an exception does not occur

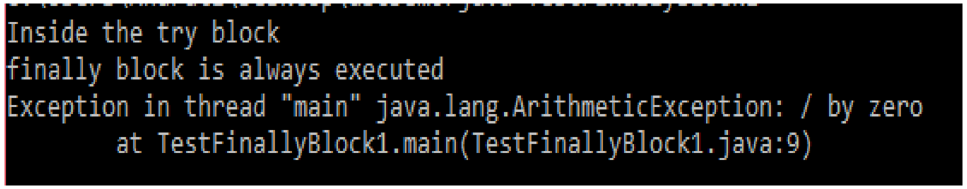
```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

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



```
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occur but not handled by the catch block

```
public class TestFinallyBlock1{
      public static void main(String args[]){
      try {
        System.out.println("Inside the try block");
        //below code throws divide by zero exception
       int data=25/0;
       System.out.println(data);
      }
      //cannot handle Arithmetic type exception
      //can only accept Null Pointer type exception
      catch(NullPointerException e){
        System.out.println(e);
      }
      //executes regardless of exception occured or not
      finally {
        System.out.println("finally block is always executed");
      }

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



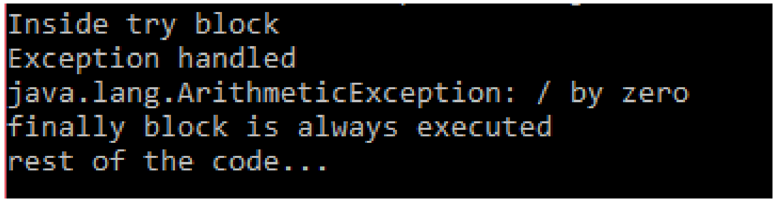
```
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

the finally block is executed after the try block and then the program terminates abnormally.

Case 3: When an exception occurs and is handled by the catch block

```
public class TestFinallyBlock2{
     public static void main(String args[]){

     try {

       System.out.println("Inside try block");

       //below code throws divide by zero exception
      int data=25/0;
      System.out.println(data);
     }
     //handles the Arithmetic Exception / Divide by zero exception
     catch(ArithmeticException e){
       System.out.println("Exception handled");
       System.out.println(e);
     }
     //executes regardless of exception occured or not
     finally {
       System.out.println("finally block is always executed");
     }
     System.out.println("rest of the code...");
     }
  }
```

the finally block is executed after the try-catch block.



```
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

- Rule: For each try block there can be zero or more catch blocks, but only one finally block.
- Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Handling Multiple Exceptions in a Single catch Block

- From Java 7 onwards, you can also catch multiple exceptions in a single catch block using the pipe | character, which reduces code duplication.

```
public class MultipleExceptionsExample {
    public static void main(String[] args) {
        try {
            if (args.length > 0 && "null".equals(args[0])) {
                String str = null;
                System.out.println(str.length()); // This can throw NullPointerException
            } else {
                int[] arr = new int[5];
                System.out.println(arr[10]); // This can throw ArrayIndexOutOfBoundsException
            }
        } catch (NullPointerException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

O/P:

```
Caught Exception: Index 10 out of bounds for length 5
Finally block executed
```

Checked Exceptions

Definition: Checked exceptions are exceptions that are checked at **compile-time**. The compiler ensures that these exceptions are either handled using a try-catch block or declared in the method signature using the throws keyword.

Purpose: They typically represent scenarios outside the program's control, such as I/O errors, database issues, or file access problems.

Examples:

- IOException
- SQLException
- FileNotFoundException

Characteristics:

- Must be handled explicitly.
- Subclasses of Exception (but not RuntimeException).
- Used for recoverable conditions.

Checked Exceptions: Example

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class Main {
    public static void main(String[] args) {
        try {
            FileInputStream file = new FileInputStream("nonexistent.txt");
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

Java throw Exception

- **Purpose:** Explicitly triggers an exception.
- **Usage:** Specify the exception object to be thrown, often with an error message describing the issue.
- **Application:** Can throw both checked and unchecked exceptions. Commonly used for custom exceptions.
- **Flexibility:** Allows developers to define and throw exceptions under specific conditions.
  - For example, we can throw ArithmeticException if we divide a number by another number.

The syntax of the Java throw Instance

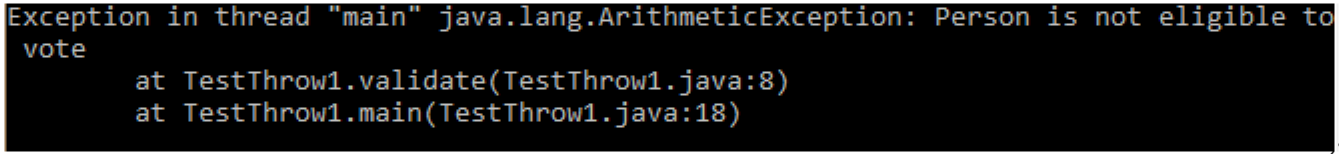
```
throw new exception_class("error message");
```

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

Java throw keyword Example

Example 1: Throwing **Unchecked Exception**

```
public class TestThrow1 {
  //function to check if person is eligible to vote or not
  public static void validate(int age) {
    if(age<18) {
      //throw Arithmetic exception if not eligible to vote
      throw new ArithmeticException("Person is not eligible to vote");
    }
    else {
      System.out.println("Person is eligible to vote!!");
    }
  }

  public static void main(String args[]){
    //calling the function
    validate(13);
    System.out.println("rest of the code...");
 }
}
```



```
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

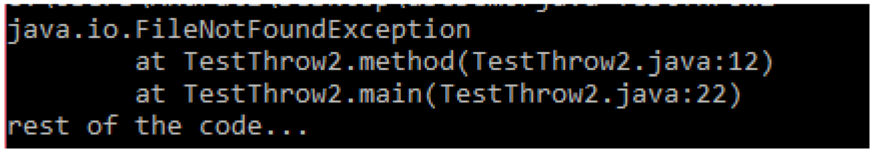
Example 2: Throwing Checked Exception

If we throw a checked exception using throw keyword, it is mandatory to **handle the exception using**

a) catch block or

b) the method must declare it using throws declaration.

```
import java.io.*;
public class TestThrow2 {
  //function to check if person is eligible to vote or not
  //b) method declared exception with throws keyword
  public static void method() throws FileNotFoundException {
    FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
    BufferedReader fileInput = new BufferedReader(file);
    //throw new FileNotFoundException();
  }
  public static void main(String args[]){
    try
    {
      method();
    }
    catch (FileNotFoundException e)
    {
      e.printStackTrace();
    }
    System.out.println("rest of the code...");
 }
}
```



```
java.io.FileNotFoundException
        at TestThrow2.method(TestThrow2.java:12)
        at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

Note:

- Every subclass of RuntimeException and Error is an unchecked exception in Java.
- A checked exception is any other subclass of Throwable that is not a subclass of RuntimeException or Error.

Handling Checked Exception with a catch Block

the checked exception is caught and handled within the method where it is thrown.

```
import java.io.IOException;

public class CatchBlockExample {

    public static void main(String[] args) {
            readFile("invalidFile");
}

    static void readFile(String fileName) {
        try {
            if ("invalidFile".equals(fileName)) {
                throw new IOException("Unable to read file: " + fileName); // Throwing IOException
            }
            System.out.println("File read successfully.");
        } catch (IOException e) {
            // Handling the IOException
            System.out.println(e.getMessage());
        }
    }
}
```

O/P:

```
Unable to read file: invalidFile
```

Java throws keyword

- The **throws keyword** in Java is used to declare that a method might throw one or more exceptions.
- It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

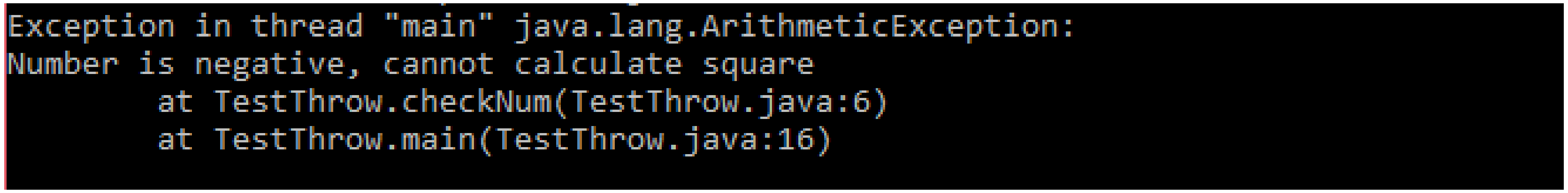
```
return_type method_name() throws exception_class_name1{
//method code
}
```

Example

```
public class ThrowsExample {

    public static void main(String[] args) {
        try {
            methodThrowingException();
        } catch (IOException e) {
            e.printStackTrace(); // Handle the exception here.
        }
    }

    static void methodThrowingException() throws IOException {
        // Some code that may throw IOException
        throw new IOException("An IO exception occurred");
    }
}
```

Java throw Example

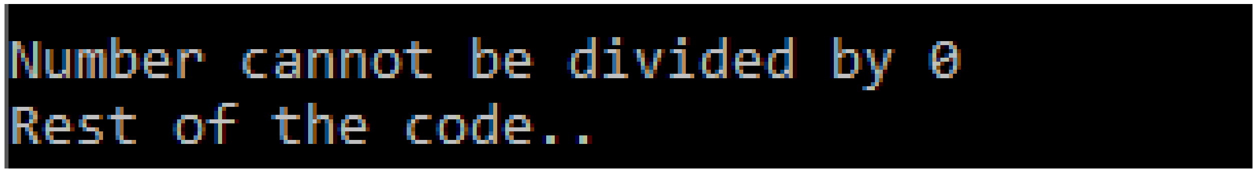
```
public class TestThrow {
  //defining a method
  public void checkNum(int num) {
    if (num < 1) {
      throw new ArithmeticException("\nNumber is negative, cannot calculate square");
    }
    else {
      System.out.println("Square of " + num + " is " + (num*num));
    }
  }
  //main method
  public static void main(String[] args) {
      TestThrow obj = new TestThrow();
      obj.checkNum(-3);
      System.out.println("Rest of the code..");
  }
}
```



```
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

Java throws Example

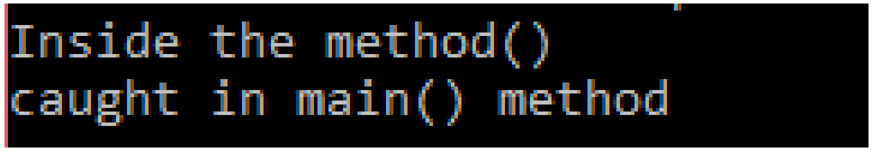
```
public class TestThrows {
  //defining a method
  public int divideNum(int m, int n) throws ArithmeticException {
    int div = m / n;
    return div;
  }
  //main method
  public static void main(String[] args) {
    TestThrows obj = new TestThrows();
    try {
      System.out.println(obj.divideNum(45, 0));
    }
    catch (ArithmeticException e){
      System.out.println("\nNumber cannot be divided by 0");
    }

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



```
Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

```java
public class TestThrowAndThrows
{
   // defining a user-defined method
   // which throws ArithmeticException
   static void method() throws ArithmeticException
   {
      System.out.println("Inside the method()");
      throw new ArithmeticException("throwing ArithmeticException"); //unchecked Exception
   }
   //main method
   public static void main(String args[])
   {
      try
      {
         method();
      }
      catch(ArithmeticException e)
      {
         System.out.println("caught in main() method");
      }
   }
}
```



```
Inside the method()
caught in main() method
```

Understanding throw & throws

Example: throw, throws

```
public class Throws_example
{
   public static double divide(int n,int d) //Made by Team2
{
              double res=n/d;
              return res;
}

public static void main(String[] args)
    {
// To be called by Team1
     double result=divide(20,0);
     System.out.println("Result: " + result);
    }
}
```

Without any exception Handling

```
PS D:\java_prog> java Throws_example
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Throws_example.divide(Throws_example.java:5)
        at Throws_example.main(Throws_example.java:11)
```

Example: throw, throws

```
public class Throws_example
{
   public static double divide(int n,int d) throws ArithmeticException
{
            double res=n/d;
            return res;
}
            public static void main(String[] args)
   {
      double result=divide(20,0);
      System.out.println("Result: " + result);
   }
}
```

- Team 2 telling everyone that function may raise the given exception.
  Team 1 use try catch for this exception or handle it properly.

```
PS D:\java_prog> java Throws_example
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Throws_example.divide(Throws_example.java:5)
        at Throws_example.main(Throws_example.java:11)
```

Example: throw, throws

```
public class Throws_example
{
   public static double divide(int n,int d) throws ArithmeticException
{
            double res=n/d;
            return res;
}
            public static void main(String[] args)
   {
                        try{
                        double result=divide(20,0);
      }catch(Exception e)
                        {
                        System.out.println("Denominator can't be negative");
                        }
                        System.out.println("Result: " + result);

   }
}
```

Exception Handled

```
PS D:\java_prog> java Throws_example
Denominator can't be negative
```

Remaining Topics will be shared soon….