

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI**  
**CS F213**  
**LAB 7**

**AGENDA**

**TIME: 2Hrs**

- Comparable and Comparator Interfaces
- Exceptions, Checked/Unchecked Exceptions
- try/catch-finally block
- throw, throws, user-defined exceptions

## **7.1 Comparable and Comparator Interfaces:**

In your previous lab sheets, you have already studied about Interfaces and their uses. You have also explored the collections framework offered by Java, which allows users to perform powerful operations on collections. The comparable and comparator interfaces are two predefined interfaces in Java, which help you to leverage the power of the existing collections framework with user-defined datatypes. Let's have a look at each of these interfaces in detail.

### **7.1.1 Comparable Interface**

Let's say that you have created a custom class to hold your data. Now, you would like to sort this data, but you don't want to write the sorting logic yourself, as that would be like re-inventing the wheel. You would like to take the help of the sort function already defined in the collections interface, but the problem is that the compiler doesn't know how to compare your custom data-classes.

This is where the Comparable interface comes into the picture. By implementing the comparable interface, you are essentially conveying to the compiler:

- Two different objects of this class can be compared (There exists a [natural ordering](#) among the objects of this class)
- To compare two objects, there is a compareTo method implemented within the class. The compareTo method compares "this" object with another object of the same class that is passed as a parameter.
- If the "this" object comes before the object received as a parameter in terms of sorting order, the method should return a negative number. If, on the other hand, the "this" object comes after the object received as a parameter, the method should return a positive number. Otherwise, 0 is returned.

Let's understand this better with an example:

### Program 7.1.a:

```
public class Member implements Comparable<Member> {
    private String name;
    private int height;

    public Member(String name, int height) {
        this.name = name;
        this.height = height;
    }

    public String getName() {
        return this.name;
    }

    public int getHeight() {
        return this.height;
    }

    @Override
    public String toString() {
        return this.getName() + " (" + this.getHeight() + ")";
    }

    @Override
    public int compareTo(Member member) {
        return this.height - member.getHeight();
    }
}
```

We can now simply sort instances of the Member class as below:

```
import java.util.*;

public class Driver {
    public static void main(String args[]) {
        List<Member> member = new ArrayList<Member>();
        member.add(new Member("mikael", 182));
        member.add(new Member("matti", 187));
        member.add(new Member("ada", 184));
        // Printing the initial list of members
        member.stream().forEach(m -> System.out.println(m));
    }
}
```

```

        System.out.println();
        // sorting a list with the sort-method of the Collections class
        Collections.sort(member);
        // Printing the list after sorting
        member.stream().forEach(m -> System.out.println(m));
    }
}

/**
What happens if you remove <Member> from the line public class Member
implements Comparable<Member>?
*/

```

## 7.1.2 Comparator Interface

Now, let's say that the business requirements of your program change in the future, and now you want to sort the members in a lexicographic order, instead of their heights. To do this using the Comparable interface, one will have to modify the Member class every time such requirements change. To solve this problem, we have the Comparator interface in Java.

### 7.1.2.1 Introduction

The comparator interface, much like the Comparable interface, is used to order objects of user-defined classes. The key difference between the two is that while the Comparable interface compares two objects of the same class, the comparator interface can be used when we need to compare objects of two different classes.

This interface is present in the java.util package, and contains 2 methods:

- compare(Object obj1, Object obj2)
- equals(Object element).

We can then use the overloaded sort method of the Collections class to sort a list of objects.

```

// To sort a given list. ComparatorClass must implement
// Comparator interface.
public void sort(List list, ComparatorClass c)

```

### 7.1.2.2 Internal Working of the Collections.sort method

Internally the Sort method does call the compare method of the classes it is sorting. To compare two elements, it asks "Which is greater?" Compare method returns -1, 0, or 1 to say if it is less

than, equal, or greater to the other. It uses this result to then determine if they should be swapped for their sort.

### 7.1.2.3 Example

#### Program 7.1.b

```
// Java program to demonstrate working of Comparator
// interface
import java.io.*;
import java.lang.*;
import java.util.*;

// A class to represent a student.
class Student {
    int rollno;
    String name, address;
    int age;

    // Constructor
    public Student(int rollno, String name, String address, int age)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
        this.age = age;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name + " "
            + this.address + " " + this.age;
    }
}

class Sortbyroll implements Comparator<Student> {
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}
```

```

    }
}

class Sortbyname implements Comparator<Student> {
    // Used for sorting in ascending order of
    // name
    public int compare(Student a, Student b)
    {
        return a.name.compareTo(b.name);
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london", 18));
        ar.add(new Student(131, "aaaa", "nyc", 18));
        ar.add(new Student(121, "cccc", "jaipur", 19));

        System.out.println("Unsorted");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyname());

        System.out.println("\nSorted by name");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));
    }
}

```

### 7.1.3 Exercises

1. Modify Program 7.1.b, so that the students are first sorted based on their age, and if there are two students with the same age, they are sorted based on their roll number.
2. Consider the following classes given below

```
public class Address implements Comparable<Address> {
    public String line1;
    public String line2;
    public String city;
    public String state;
    public int pincode;

    public Address(String line1, String line2, String city, String state,
int pincode) {
        this.line1 = line1;
        this.line2 = line2;
        this.city = city;
        this.state = state;
        this.pincode = pincode;
    }

    @Override
    public int compareTo(Address address) {
        if(this.state.equals(address.state)) {
            if(this.city.equals(address.city)) {
                return this.pincode - address.pincode;
            }
            return this.city.compareTo(address.city);
        }
        return this.state.compareTo(address.state);
    }
}

public class Delivery {
    Address address;
    int profit;

    public void Delivery(Address address, int profit) {
        this.address = address;
        this.profit = profit;
    }
}
```

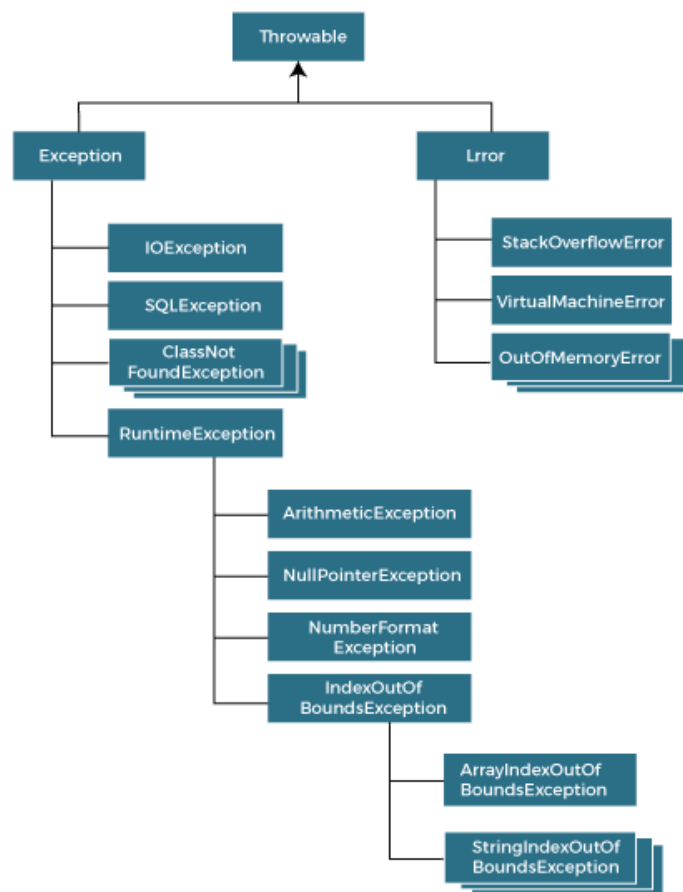
Create a comparator class that sorts objects of the class Delivery, first based on the descending order of profits. If two deliveries have the same profit, then sort them in the order of their addresses. Also create a driver class, containing the main function, that creates some instances of the Delivery class, and then sorts them using the above comparator.

## 7.2 Exception:

An Exception is an error that on occurrence can terminate the program abnormally.

Exception Handling in Java is one of the powerful mechanisms to handle runtime errors so that the normal flow of the application can be maintained.

Hierarchy of Java Exception classes :



### 7.2.1 Types of Java Exceptions

There are mainly 3 types of Exceptions :

1. Checked Exception
2. Unchecked Exception
3. Error

### 7.2.1.1 Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, FileNotFoundException etc. Checked exceptions are checked at compile-time.

#### Program 7.2.a

```
import java.io.*;
class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\\test\\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

The program above will throw a compile-time error.

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source
code - unreported exception java.io.FileNotFoundException; must be caught
or declared to be thrown at Main. main(Main.java:5)
```

Try This: To convert it to the Unchecked exception *add throws i.e. runtime exception*.

### 7.2.1.2 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. This can also be implemented through the use of throw

Eg: Arithmetic Exception i.e. divide by 0.

### Program 7.2.b

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

### 7.2.1.3 Error

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

### Program 7.2.c

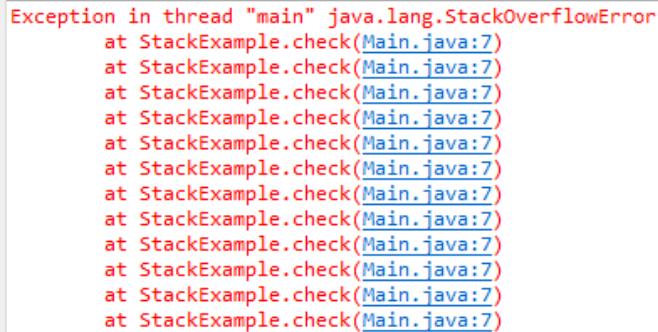
```
class StackExample {
    public static void check(int i)
    {
        if (i == 0)
            return;
        else {
            check(i++);
        }
    }
}

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

```

{
    StackExample.check(5);
}
}

```



```

Exception in thread "main" java.lang.StackOverflowError
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)
    at StackExample.check(Main.java:7)

```

## 7.2.2. Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

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 in a NumberFormatException. Suppose we have a string variable that has characters; converting this variable into a digit will cause NumberFormatException.

```
String s="abc";
```

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

4) A scenario where *ArrayIndexOutOfBoundsException* occurs: When an array exceeds its size, the *ArrayIndexOutOfBoundsException* occurs. There may be other reasons for the *ArrayIndexOutOfBoundsException*. Consider the following statements.

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

## 7.3 Try Catch Finally Block:

Programs developed by you are often used by end-users who have no knowledge of the technologies being used. Programs often rely on input given by these end-users, and in some cases, the input might not be as expected by the programmer. In such cases, instead of abruptly stopping the program, programmers can use the try/catch block.

### 7.3.1 Try/Catch Block

The try statement allows you to define a block of code to be tested for errors while it is being executed. The catch statement allows you to define a block of code to be executed, if an error occurs in the try block. The try and catch keywords are always used in pairs. The basic syntax looks like this:

```
try {  
    // Block of code to try  
}  
catch(Exception e) {  
    // Block of code to handle errors  
}
```

Let's assume that your program is expecting the user to enter an integer. However, the user, due to their negligence, ends up entering a string instead. To protect the program from crashing in such cases, we could use the try/catch block.

#### Program 7.3.a

```

import java.util.*;

public class Driver {
    public static void main(String args[]) {
        int x;
        Scanner sc = new Scanner(System.in);
        System.out.println("Please enter an integer");
        try {
            // System.out.println("Before taking integer input");
            x = sc.nextInt();
            System.out.println("Integer entered = " + x);
        } catch (Exception e) {
            x = 0;
            System.out.println("Please enter an integer only.");
        }
    }
}

// Try entering an integer, and observe the o/p
// Try entering a character, and observe the o/p
// Try uncommenting the print statement, and then repeat the above 2
// exercises
// Try commenting out the catch block, and see what happens

```

If you observe the output of the above program carefully, you see that the control either remains in the try block (skipping the catch block after the try block is completely executed), or it goes to the catch block (skipping the lines in the try block after which the error has occurred).

### 7.3.2 Finally Block

Now, What if you had some common code that you wanted to execute, irrespective of the result of the try/catch block? For now, it seems like you would have to duplicate your code in both the blocks. Using the *finally* block is a better solution.

The **finally** statement lets you execute code, after **try...catch**, regardless of the result

#### Program 7.3.b

```

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

```

```

        int[] myNumbers = {1, 2, 3};
        System.out.println(myNumbers[10]);
    } catch (Exception e) {
        System.out.println("Something went wrong.");
    } finally {
        System.out.println("The 'try catch' is finished.");
    }
}
}
}

```

## 7.3.3 Multiple Try/Catch Blocks

### 7.3.3.1 Multiple Catch Blocks

Let's say that when an error occurs, you want to display different messages based on the kind of error that occurred. After all, you might wish to inform the user what is the problem with their input. Is the input not in the correct format, or the input is correct, but the file with that name doesn't exist, etc. For this, we use multiple catch blocks.

#### Program 7.3.c

```

public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            // Line 1: a[5]=30/0;
            // Line 2: System.out.println(a[10]);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException
occurs");
        }
        catch(Exception e)
        {

```

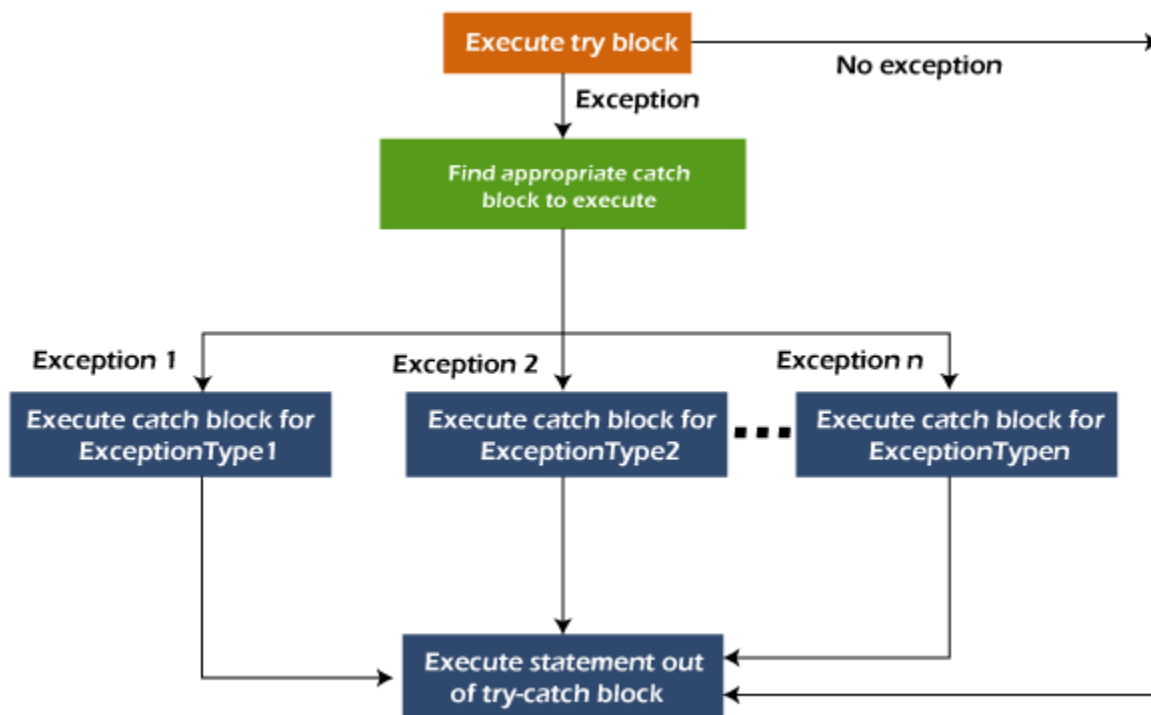
```

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

// What is the expected output when you run the above code?
// What happens when you uncomment Line 1 only, and run the code?
// What happens when you uncomment Line 2 only, and run the code?
// What happens when you uncomment both Line 1 and 2, and run the code?

```

The flow of execution of the above program could easily be understood with the following chart:



### 7.3.3.2 Nested Try Catch Blocks

#### Program 7.3.c

```

class NestingDemo{

```

```

public static void main(String args[]){
    //main try-block
    try{
        //try-block2
        // Line 1: int b =45/0;
        try{
            //try-block3
            try{
                int arr[]={1,2,3,4};
                // Line 2: System.out.println(arr[10]);
            }catch(ArithmeticException e){
                System.out.print("Arithmetic Exception");
                System.out.println(" handled in try-block3");
            }
        }
        catch(ArithmeticException e){
            System.out.print("Arithmetic Exception");
            System.out.println(" handled in try-block2");
        }
    }
    catch(ArithmeticException e3){
        System.out.print("Arithmetic Exception");
        System.out.println(" handled in main try-block");
    }
    catch(ArrayIndexOutOfBoundsException e4){
        System.out.print("ArrayIndexOutOfBoundsException");
        System.out.println(" handled in main try-block");
    }
    catch(Exception e5){
        System.out.print("Exception");
        System.out.println(" handled in main try-block");
    }
}
}

```

*// Uncomment lines 1 and 2 in the above example, and observe the outputs  
 // Play around a bit by placing line 1 and line 2 in different blocks, and  
 observing the output.*

### 7.3.4 Exercises

1. Try and modify the behaviour of Program 7.3.a so that the program keeps on prompting the user to enter an integer, till a valid input has been entered.

## 7.4 Throw, Throws and User Defined Exceptions:

The Java throw keyword is used to throw an exception explicitly. We specify the exception object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

The syntax of the Java throw keyword is given below.

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

Let's see the example of throw IOException.

```
accessModifier returnType methodName() throws ExceptionType1,
ExceptionType2 ... {
    // code
}
```

### Which exception should be declared?

**Ans:** Checked exception only, because:

- **unchecked exception:** indicate programming errors, and under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if a VirtualMachineError or StackOverflowError occurs.

### Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in the call stack).

It provides information to the caller of the method about the exception.

### Program 7.4.a : Java throws Keyword

```
import java.io.*;
class Main {
```



```

public static void findFile() throws IOException {
    // code that may produce IOException
    File newFile=new File("test.txt");
    FileInputStream stream=new FileInputStream(newFile);
}

public static void main(String[] args) {
    try{
        findFile();
    } catch(IOException e){
        System.out.println(e);
    }
}
}

```

Output

```
java.io.FileNotFoundException: test.txt (No such file or directory)
```

When we run this program, if the file test.txt does not exist, FileInputStream throws a FileNotFoundException which extends the IOException class. If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the throws clause so that methods further up in the call stack can handle them or specify them using throws keyword themselves.

## Throwing multiple exceptions

Here's how we can throw multiple exceptions using the throws keyword.

```

import java.io.*;
class Main {
    public static void findFile() throws NullPointerException, IOException,
    InvalidClassException {

        // code that may produce NullPointerException
        ... ..

        // code that may produce IOException
        ... ..
    }
}

```

```

    // code that may produce InvalidClassException
    ... ..
}

public static void main(String[] args) {
    try{
        findFile();
    } catch(IOException e1){
        System.out.println(e1.getMessage());
    } catch(InvalidClassException e2){
        System.out.println(e2.getMessage());
    }
}
}

```

Here, the findFile() method specifies that it can throw NullPointerException, IOException, and InvalidClassException in its throws clause.

Note that we have not handled the NullPointerException. This is because it is an unchecked exception. It is not necessary to specify it in the throws clause and handle it.

### 7.4.1 throws keyword vs. try...catch...finally

There might be several methods that can cause exceptions. Writing try...catch for each method will be tedious and code becomes long and less-readable. throws is also useful when you have checked exception (an exception that must be handled) that you don't want to catch in your current method.

- In case we declare an exception and it does not occur, the code will be executed fine.
- In case we declare the exception and it occurs, it will be thrown at runtime because **throws** does not handle the exception.

Let's see examples for both scenarios.

#### ***A) If exception does not occur***

##### **Program 7.4.b**

```

import java.io.*;
class M{
    void method()throws IOException{
        System.out.println("device operation performed");
    }
}

```

```

    }
}
class Testthrows3{
    public static void main(String args[])throws IOException{//declare
exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}
Output:
device operation performed
        normal flow...

```

### ***B) If exception occurs***

#### **Program 7.4.c**

```

import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
class Testthrows4{
    public static void main(String args[])throws IOException{//declare
exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}
Output:

```

```
Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)
```

## 7.4.2 Java throw keyword

The throw keyword is used to explicitly throw a single exception.

When an exception is thrown, the flow of program execution transfers from the try block to the catch block. We use the throw keyword within a method.

Its syntax is:

```
throw throwableObject;
```

A throwable object is an instance of class Throwable or subclass of the Throwable class.

### Program 7.4.d Java throw keyword

```
class Main {
    public static void divideByZero() {
        throw new ArithmeticException("Trying to divide by 0");
    }

    public static void main(String[] args) {
        divideByZero();
    }
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to
divide by 0
    at Main.divideByZero(Main.java:3)
    at Main.main(Main.java:7)
exit status 1
```

In this example, we are explicitly throwing an `ArithmeticException`.

**Note:** `ArithmeticException` is an unchecked exception. It's usually not necessary to handle unchecked exceptions.

#### Program 7.4.e : Throwing checked exception

```
import java.io.*;
class Main {
    public static void findFile() throws IOException {
        throw new IOException("File not found");
    }

    public static void main(String[] args) {
        try {
            findFile();
            System.out.println("Rest of code in try block");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Output

File not found

The `findFile()` method throws an `IOException` with the message we passed to its constructor. Note that since it is a checked exception, we must specify it in the `throws` clause. The methods that call this `findFile()` method need to either handle this exception or specify it using `throws` keyword themselves. We have handled this exception in the `main()` method. The flow of program execution transfers from the `try` block to the `catch` block when an exception is thrown. So, the rest of the code in the `try` block is skipped and statements in the `catch` block are executed.

### 7.4.3 User-defined Custom Exception in Java

Java provides us the facility to create our own exceptions which are basically derived classes of `Exception`. For example `MyException` in below code extends the `Exception` class.

#### Program 7.4.f

```
// A Class that represents use-defined exception
```

```

class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}
// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("Attention!");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}

```

Output: Caught  
Attention!

In the above code, constructor of MyException requires a string as its argument. The string is passed to the parent class Exception's constructor using super(). The constructor of [Exception](#) class can also be called without a parameter and call to super is not mandatory.

#### Program 7.4.g

```

// A Class that represents use-defined expception
class MyException extends Exception

```

```

{
}
// A Class that uses above MyException
public class setText
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException();
        }
        catch (MyException ex)
        {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
}
Output: Caught
null

```

#### 7.4.4 Exercises

Q1. What should be the appropriate changes to get output of this program ?

```

class ThrowsExecp
{
    static void fun() _ IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
        try
        {

```

```

        _;
    }
    catch(IllegalAccessException e)
    {
        System.out.println("caught in main.");
    }
}

```

#### Output:

Inside fun().  
caught in main.

Q2. Fill in the missing keyword to get the output.

```

private static List<Integer> integers = new ArrayList<Integer>();

public static void addInteger(Integer value) throws
IllegalArgumentException {
    if (integers.contains(value)) {
        _ IllegalArgumentException("Integer already added.");
    }
    integers.add(value);
}

public static void main(String[] args) {
    try {
        addInteger(1);
    } catch (IllegalArgumentException iae) {
        iae.printStackTrace();
    }
}

```

## 7.4.5 Take Home

Q1. What exception can we add at the blank?

```

public double divide(double a, double b) {

```



```

    if (b == 0) {
        _____ }
    return a / b;
}

```

Q2. Can we make our own exception in the above case? If yes, which class we can 'inherit' from?

Q3. Modify the following cat method so that it will compile.

```

public static void cat(File file) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(file, "r");
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
        return;
    } finally {
        if (input != null) {
            input.close();
        }
    }
}

```

Q4 Now with the above code you have to use user defined exceptions to handle the errors.

Q5 Modify Q2 from exercise 7.4.4 such that

- If the user adds a negative value to the ArrayList, throw a `NegativeValEnteredException`, which is an extension of `ArithmeticException`. The display of the error information should display the negative number that was entered.
- If the user adds zero to the ArrayList, throw a **`ZeroValEnteredException`**, which

is an extension of **ArithmeticException**.