CSE215L Programming Language II Lab

Faculty: Dr. Mohammad Rashedur Rahman (RRn)
Lab Manual 15: Exception Handling
Lab Instructor: Md. Mustafizur Rahman

Objective:

- To get an overview of exceptions and exception handling
- To know about the implementation details of exception handling means how to declare, throw, and handle exceptions

Exception Handling: Exception handling enables a program to deal with exceptional situations and continue its normal execution. *Runtime errors* occur while a program is running if the JVM detects an operation that is impossible to carry out. In Java, runtime errors are thrown as exceptions. An *exception* is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally.

For example, if we try to divide any number by zero, a runtime error occurs. The following code shows how to deal with a basic exception.

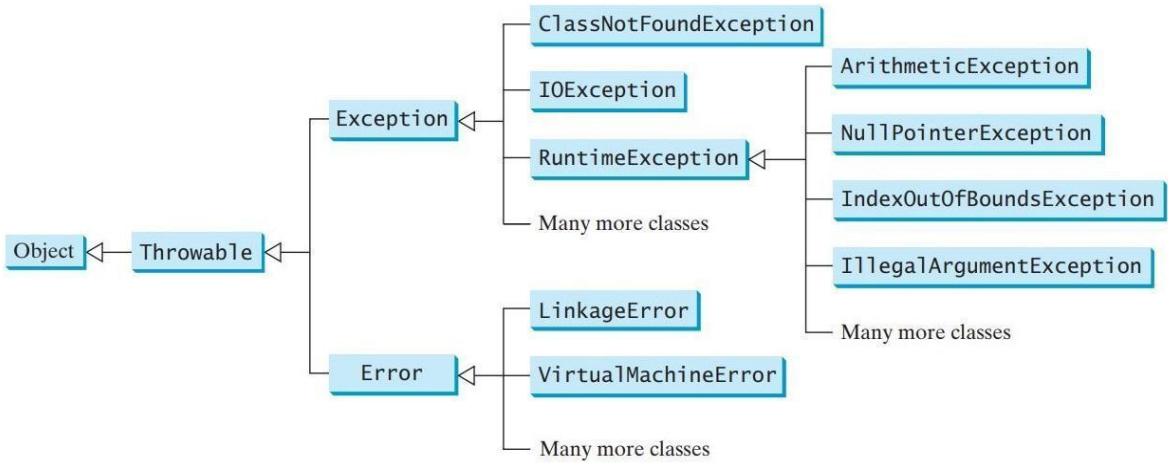
QuotientWithException.java

```
public class QuotientWithException {

	public static void main(String[] args) {
		int number1 = 10, number2 = 0;

		try {
			int result = number1 / number2;
			System.out.println(number1+"/"+number2+" = "+result);
		}
		catch(ArithmeticException ex) {
			ex.printStackTrace();
		}
	}
}
```

Exception Types: Exceptions are objects, and objects are defined using classes. The root class for exceptions is *java.lang.Throwable*. The following diagram shows that the exceptions thrown are instances of these classes or subclasses of one of these classes.



The *Throwable* class is the root of exception classes and all Java exception classes inherit directly or indirectly from *Throwable*. The exception classes can be classified into three major types: *system errors*, *exceptions*, and *runtime exceptions*.

I. **System Errors:** System errors are thrown by the JVM and are represented in the *Error* class. The *Error* class describes internal system errors, though such errors rarely occur. If one does, there is little we can do beyond notifying the user and trying to terminate the program gracefully.

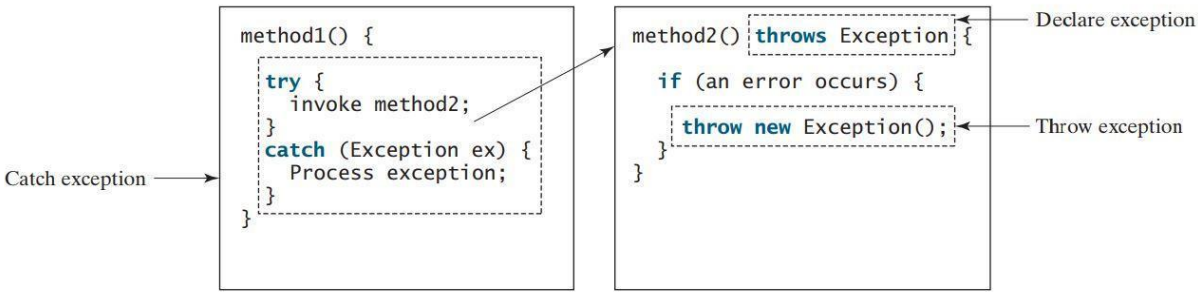
II. **Exceptions:** Exceptions are represented in the *Exception* class, which describes errors caused by our program and by external circumstances. These errors can be caught and handled by our program.

III. **Runtime Exceptions:** Runtime exceptions are represented in the *RuntimeException* class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions are generally thrown by the JVM.

[**N.B.** The class names *Error*, *Exception*, and *RuntimeException* are somewhat confusing. All three of these classes are exceptions, and all of the errors occur at runtime.]

Unchecked vs Checked exceptions: *RuntimeException*, *Error*, and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with them in a *try-catch* block or declare it in the method header. In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable. Also, unchecked exceptions can occur anywhere in a program. To avoid cumbersome overuse of *try-catch* blocks, Java does not mandate that we write code to catch or declare unchecked exceptions.

Exception Handling Model: Java's exception-handling model is based on three operations- *declaring an exception*, *throwing an exception*, and *catching an exception*, as shown in the following figure.



1. **Declaring Exceptions:** In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the *main* method to start executing a program. Every method must state the types of *checked exceptions* it might throw. This is known as *declaring exceptions*. To declare an exception in a method, use the ***throws*** keyword in the method header, as in this example,

   ```
   public void myMethod() throws Exception1, Exception2, . . ., ExceptionN
   ```

2. **Throwing Exceptions:** A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. For example, a program can create an instance of *ArithmeticException* and throw it as follows,

   ```
   ArithmeticException ex = new ArithmeticException("Divisor can't be zero!");
   throw ex;
   ```

   ```
   or, throw new ArithmeticException("Divisor can't be zero!");
   ```

3. **Catching Exceptions:** The code that handles the exception is called the *exception handler*; it is found by *propagating the exception* backward through a chain of method calls, starting from the current method. The process of finding a handler is called *catching an exception*. When an exception is thrown, it can be caught and handled in a *try-catch* block, as follows,

   ```
   try {
     statements; // Statements that may throw exceptions
   }
   catch (Exception1 exVar1) {
     handler for exception1;
   }
   catch (Exception2 exVar2) {
     handler for exception2;
   }
   ...
   catch (ExceptionN exVarN) {
     handler for exceptionN;
   }
   ```

Now, the following code shows how to declare throws and catch an exception altogether.

QuotientWithException2.java

```
public class QuotientWithException2 {

	// Declaring an exception in method header using 'throws' keyword
	public static int quotient(int number1, int number2)
			throws ArithmeticException {

		// throwing an exception using 'throw' keyword
		if(number2 == 0)
			throw new ArithmeticException("Divisor can't be zero!");

		return number1/number2;
	}

	public static void main(String[] args) {
		int number1 = 10, number2 = 0;

		// catching an exception using try-catch-finally block
		try {
```

```
                int result = quotient(number1, number2);

                System.out.println(number1+"/"+number2+" = "+result);
            }
            catch(ArithmeticException ex) {
                ex.printStackTrace();
            }
            finally {
                System.out.println("Whatever happens," +
                                   "finally block will be executed!");
            }
        }
}
```

Class with Exception: The following example demonstrate declaring, throwing, and catching exceptions by modifying the *setRadius()* method in the *Circle* class. The new *setRadius()* method throws an exception if the radius is negative.

Circle.java

```
public class Circle {
    private double radius;
    private static int numberOfObjects = 0;

    public Circle(double radius) {
        setRadius(radius);
        numberOfObjects++;
    }

    public double getRadius() {
        return radius;
    }

    // Declaring an exception in method header using 'throws' keyword
    public void setRadius(double radius)
        throws IllegalArgumentException {

        // throwing an exception using 'throw' keyword
        if(radius >= 0)
            this.radius = radius;
        else
            throw new IllegalArgumentException("Radius can't be negative");
    }

    public static int getNumberOfObjects() {
        return numberOfObjects;
    }

    public double findArea() {
        return radius*radius*3.1416;
    }
}
```

CircleMain.java

```
public class CircleMain {
	public static void main(String[] args) {
		// catching an exception using try-catch-finally block
		try {
			Circle c1 = new Circle(5);
			Circle c2 = new Circle(-5);
			Circle c3 = new Circle(0);
		}
		catch(IllegalArgumentException ex) {
			System.out.println(ex);
		}
		finally {
			System.out.println("Number of object created: "
					+ Circle.getNumberOfObjects());
		}
	}
}
```

Custom Exception Class: A custom exception class can be defined by extending the *java.lang.Exception* class. The following example demonstrates how to create and use a custom exception class.

InvalidRadiusException.java

```
public class InvalidRadiusException extends Exception {
	private double radius;

	public InvalidRadiusException(double radius) {
		super("Invalid radius: " + radius);
		this.radius = radius;
	}

	public double getRadius() {
		return radius;
	}
}
```

CircleCustomException.java

```
public class CircleCustomException {
	private double radius;
	private static int numberOfObjects = 0;

	public CircleCustomException(double radius)
		throws InvalidRadiusException {
		setRadius(radius);
		numberOfObjects++;
	}

	public double getRadius() {
		return radius;
	}

	// Declaring a custom exception in method header
	public void setRadius(double radius)
```

```
            throws InvalidRadiusException {

            // throwing a custom exception
            if(radius >= 0)
                  this.radius = radius;
            else
                  throw new InvalidRadiusException(radius);
      }

      . . . .
}
```

CircleCustomExceptionMain.java

```
public class CircleCustomExceptionMain {
      public static void main(String[] args) {
            // catching an exception using try-catch-finally block
            try {
                  CircleCustomException c1 = new CircleCustomException(5);
                  CircleCustomException c2 = new CircleCustomException(-5);
                  CircleCustomException c3 = new CircleCustomException(0);
            }
            catch(InvalidRadiusException ex) {
                  System.out.println(ex);
            }
            finally {
                  System.out.println("Number of object created: "
                              + CircleCustomException.getNumberOfObjects());
            }
      }
}
```

Tasks:

1. Write a program that creates an integer array of size 100 and initializes it with random values:

   int a = (int) (Math.random() * 100);

   The program then takes an integer from the user, uses it as an index, and tries to

   print the corresponding element of that array. If the index is out of bounds, then the program should catch it and display an appropriate message. If the index number is valid, then try to divide the corresponding number of that index by an integer, and print that result. Also, try to catch the necessary exception if it tries to divide by zero. Finally, print the line “Program Ends”.

2. Write a program that takes 10 positive integers from the user and prints the sum. If any negative value is entered, the program should catch it as an exception and display “Input positive integer only”. The program must continue taking input until it gets 10 positive integers.

3. Create a **Triangle** class. Now create a user-defined exception class named **IllegalTriangleException** class that extends **Exception**. If the sum of any two sides is not greater than the third side, the **Triangle** class should **throw IllegalTriangleException.**

4. Create a user-defined exception class named **MyException** which extends **Exception**. Now in your main class which contains the main method, create another static method named getSquareRoot() to compute the square root of any given number. From the main method, call the getSquareRoot() method and it **throws** exception of **MyException** class. The exception is thrown when we pass a negative number as parameter for computing square root. The method from which getSquareRoot() is called, will handle the exception.