523232
OBJECT-ORIENTED TECHNOLOGY

9
EXCEPTION HANDLING

Komsan Srivisut, PhD
School of Computer Engineering

EXCEPTION

- An exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.
- When an exception occurs within a method, it creates an object.
  - This object is called the exception object.
  - It contains information about the exception such as the name and description of the exception and the state of the program when the exception occurred.

EXCEPTION

- An exception can occur for many reasons. Some of them are:
  - Invalid user input
  - Device failure
  - Loss of network connection
  - Physical limitations (out of disk memory)
  - Code errors
  - Opening an unavailable file

ERROR

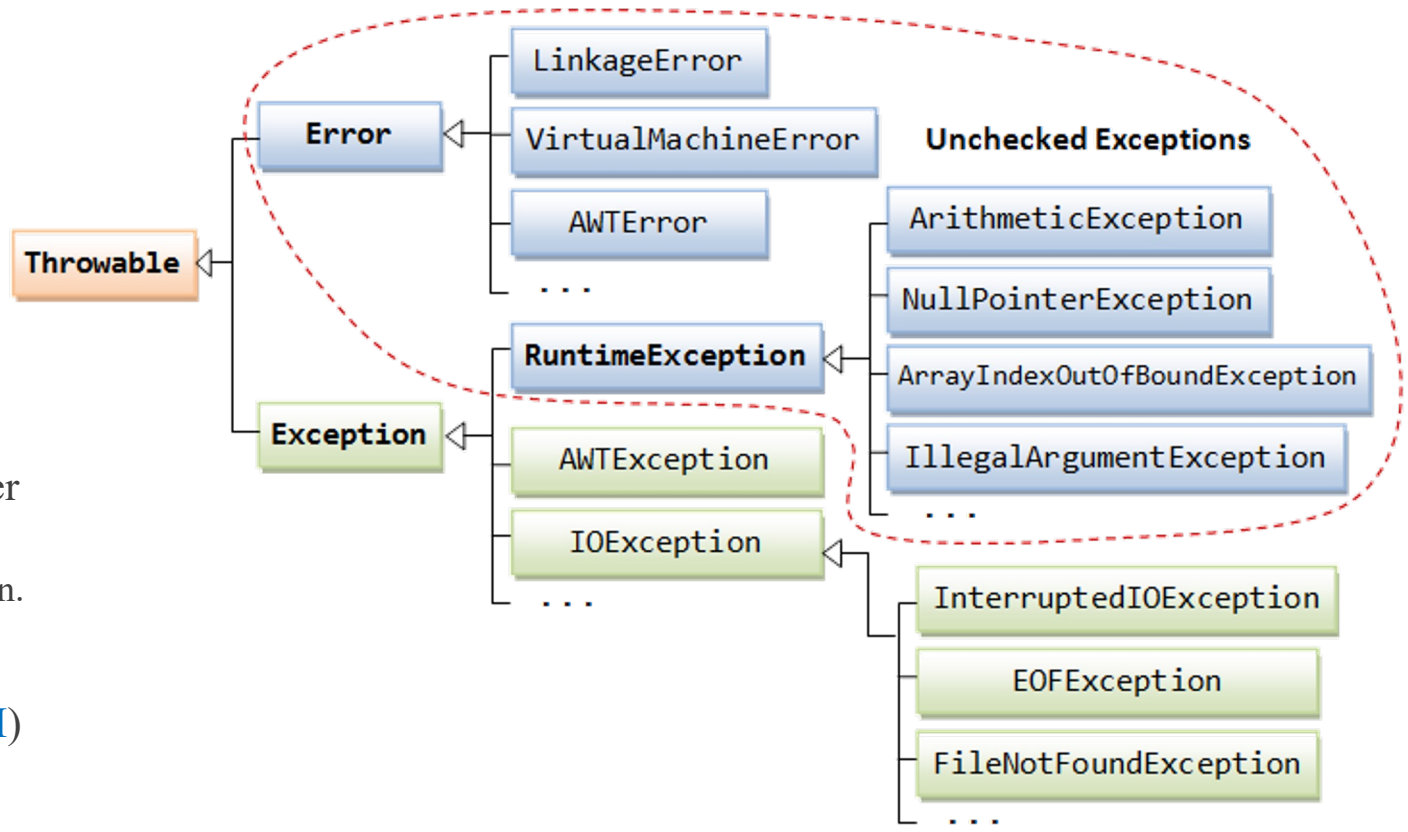
- Errors represent irrecoverable conditions, such as:
  - Java virtual machine (JVM) running out of memory
  - Memory leaks
  - Stack overflow errors
  - Library incompatibility
  - Infinite recursion
- Errors are usually beyond the control of the programmer and we should not try to handle errors.

EXCEPTION VS ERROR

- **Exception**: Exception indicates conditions that a reasonable application might try to catch.
- **Error**: An Error indicates a serious problem that a reasonable application should not try to catch.

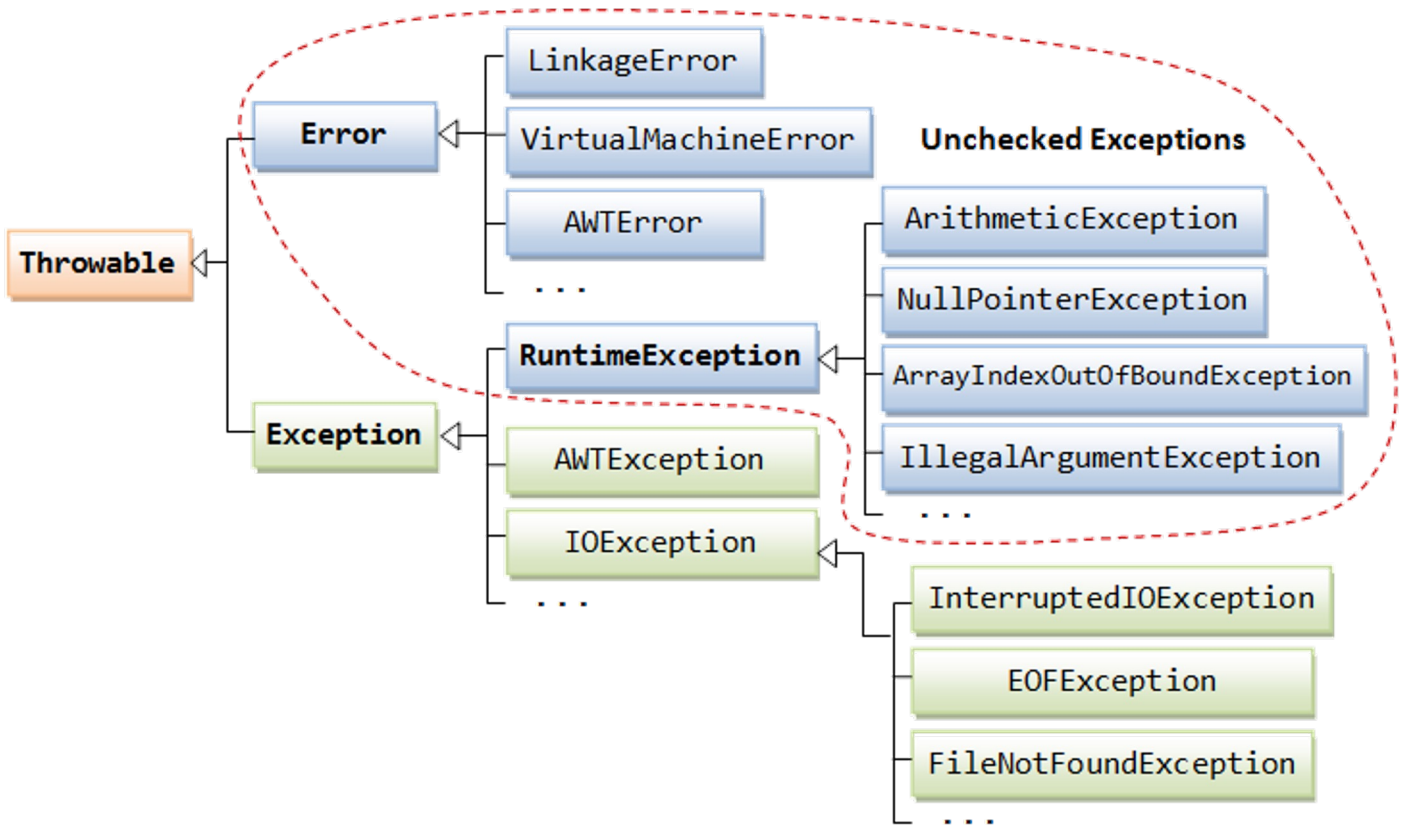
EXCEPTION HIERARCHY

- All exception and errors types are subclasses of class `Throwable`, which is the base class of the hierarchy.
- One branch is headed by `Exception`.
  - This class is used for exceptional conditions that user programs should catch.
    - `NullPointerException` is an example of such an exception.
- Another branch is headed by `Error`.
  - This class is used by the Java run-time system (JVM) to indicate errors having to do with the run-time environment itself (JRE).
    - `StackOverflowError` is an example of such an error.



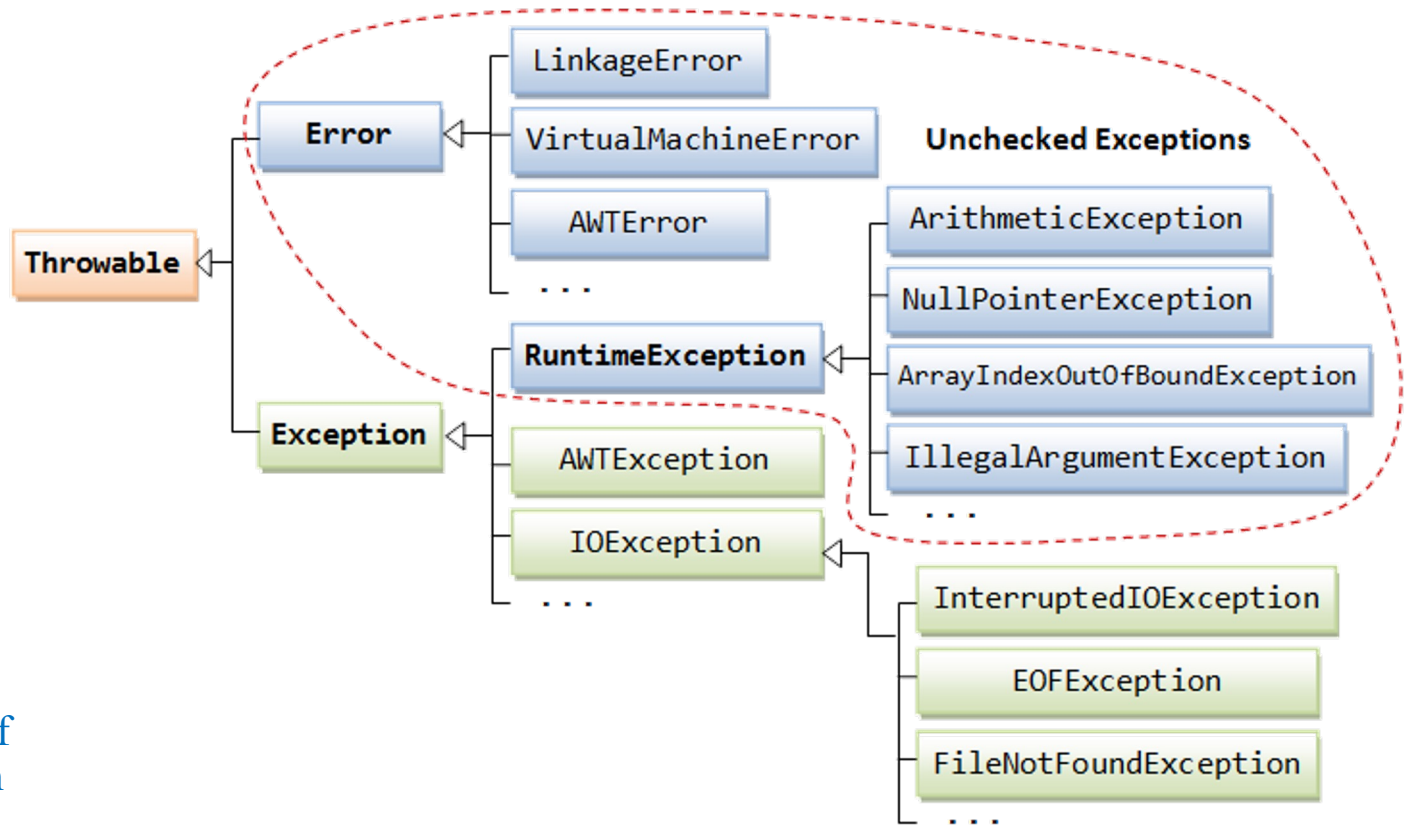
TYPES OF EXCEPTIONS

- Java defines several types of exceptions that relate to its various class libraries.
- Java also allows users to define their own exceptions.
- Exceptions can be categorised into 2 ways:
  1) Built-in Exceptions
     - Checked Exception
     - Unchecked Exception
  2) User-Defined Exceptions



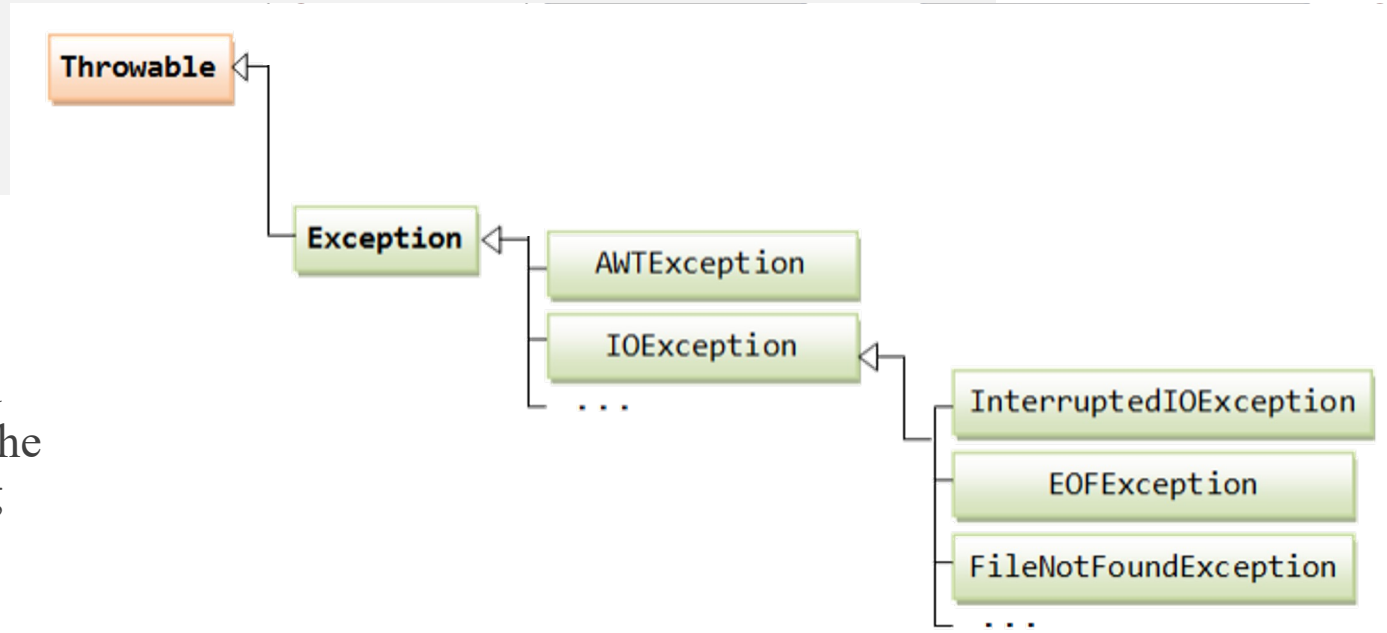
BUILT-IN EXCEPTIONS

- Built-in exceptions are the exceptions that are available in Java libraries.
- These exceptions are suitable to explain certain error situations.
  - Checked Exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
  - Unchecked Exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.
-



CHECKED EXCEPTIONS

- These are the exceptions that are checked at compile time.
- If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the `throws` keyword.



EXAMPLE: CHECKED EXCEPTION

- The program doesn't compile, because the method `main()` uses `FileReader()` and `FileReader()` throws a checked exception `FileNotFoundException`.
- It also uses `readLine()` and `close()` methods, and these methods also throw checked exception `IOException`.

```java
package week11.examples;

// Importing I/O classes
import java.io.*;

public class Example11_1 {

    public static void main(String[] args) {

        // Creating a file and reading from local repository
        FileReader file = new FileReader("src/week11/examples/a.txt
");

        // Reading content inside a file
        BufferedReader fileInput = new BufferedReader(file);

        // Printing first 3 lines of the file
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        // Closing all file connections using close() method
        // Good practice to avoid any memory leakage
        fileInput.close();
    }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        Unhandled exception type FileNotFoundException
        Unhandled exception type IOException
        Unhandled exception type IOException

        at week11.examples.Example11_1.main(Example11_1.java:15)
```

EXAMPLE: CHECKED EXCEPTION

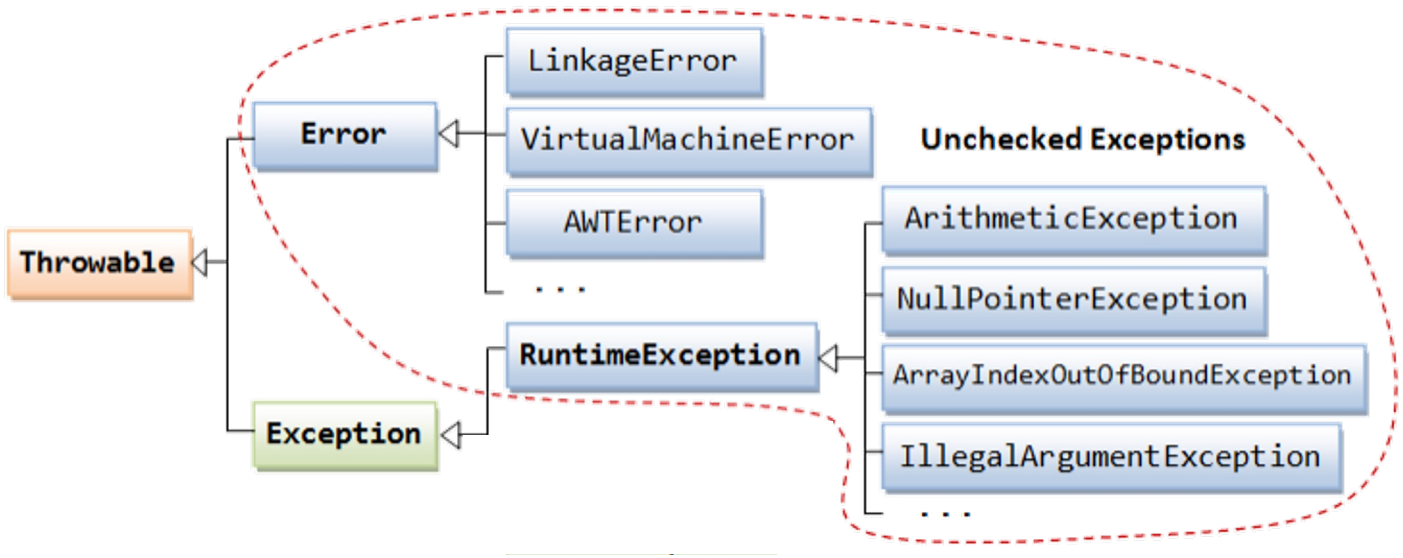
- To fix the program, we either need to specify a list of exceptions using `throws`, or we need to use a try-catch block.
- We have used `throws` in the program. Since `FileNotFoundException` is a subclass of `IOException` to make the program compiler-error-free.

```java
package week11.examples;

// Importing I/O classes
import java.io.*;

public class Example11_1 {

    public static void main(String[] args) throws IOException {

        // Creating a file and reading from local repository
        FileReader file = new FileReader("src/week11/examples/a.txt
");

        // Reading content inside a file
        BufferedReader fileInput = new BufferedReader(file);

        // Printing first 3 lines of the file
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        // Closing all file connections using close() method
        // Good practice to avoid any memory leakage
        fileInput.close();
    }
}
```

```
Exception in thread "main" java.io.FileNotFoundException: src/week11/examples/a.txt (No such file or directory)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:211)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:153)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:108)
        at java.base/java.io.FileReader.<init>(FileReader.java:60)
        at week11.examples.Example11_1.main(Example11_1.java:15)
```

UNCHECKED EXCEPTIONS

- These are the exceptions that are not checked at compile time.
- In Java exceptions under `Error` and `RuntimeException` classes are unchecked exceptions, everything else under throwable is checked.



EXAMPLE: UNCHECKED EXCEPTION

- It compiles fine, but it throws `ArithmeticException` when run.
- The compiler allows it to compile because `ArithmeticException` is an unchecked exception.

```java
package week11.examples;

public class Example11_2 {
    public static void main(String[] args) {
        // Here we are dividing by 0
        // which will not be caught at compile time
        // as there is no mistake but caught at runtime
        // because it is mathematically incorrect
        int x = 0;
        int y = 10;
        int z = y / x;
    }
}
```

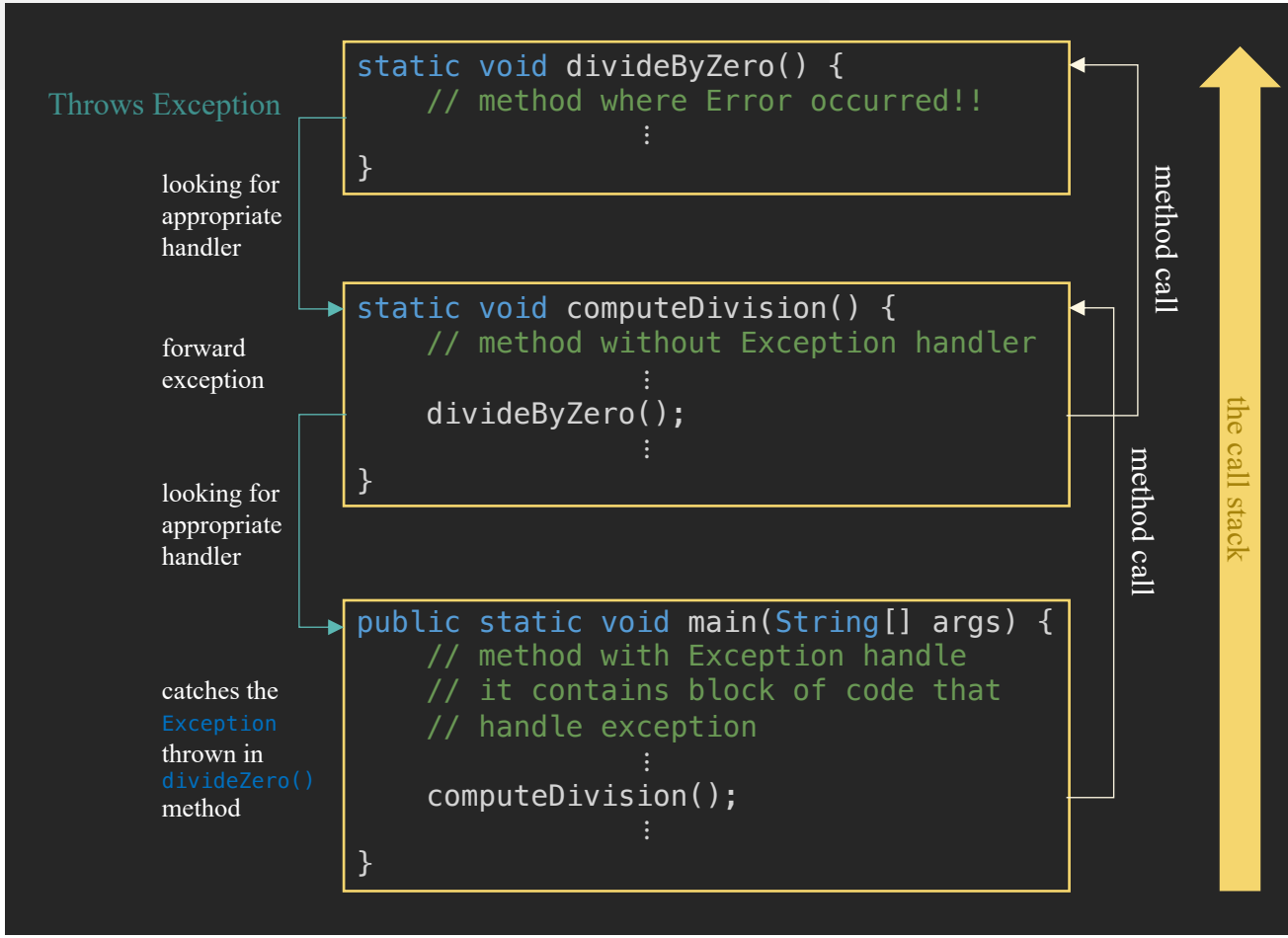
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at week11.examples.Example11_2.main(Example11_2.java:11)
```

HOW DOES JVM HANDLE AN EXCEPTION?

- Default Exception Handling: whenever inside a method, if an exception has occurred, the method creates an `Object` known as `Exception` object and hands it off to the run-time system (JVM).
- The `Exception` object contains the name and description of the exception and the current state of the program where the exception has occurred.
- Creating the `Exception` object and handling it to the run-time system is called throwing an exception.
- There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called Call Stack.

HOW DOES JVM HANDLE AN EXCEPTION?

- Now the following procedure will happen.
  - The run-time system (JVM) searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an Exception handler.
  - The JVM starts searching from the method in which the exception occurred, proceeds through the call stack in the reverse order in which methods were called.
  - If it finds an appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
  - If the JVM searches all the methods on the call stack and couldn't have found the appropriate handler then the JVM handover the `Exception` Object to the default exception handler, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program abnormally.




```
Exception in thread "xxx" Name of Exception: Description
        ... // call stack
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at week11.examples.Example11_2.main(Example11_2.java:11)
```

EXAMPLE: EXCEPTION HANDLINING

```java
package week11.examples;

public class Example11_3 {
    public static void main(String[] args) {
        int a = 1;
        int b = 0;
        int i = computeDivision(a, b);
        System.out.println(i);
    }

    static int computeDivision(int a, int b) {
        int res = 0;
        res = divideByZero(a, b);
        return res;
    }

    static int divideByZero(int a, int b) {
        // this statement will cause ArithmeticException(/ by zero)
        int i = a / b;
        return i;
    }
}
```

Example of an Exception generated by system

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at week11.examples.Example11_3.divideByZero(Example11_3.java:34)
        at week11.examples.Example11_3.computeDivision(Example11_3.java:23)
        at week11.examples.Example11_3.main(Example11_3.java:9)
```

HOW PROGRAMMER HANDLES AN EXCEPTION?

- Customized Exception Handling: Java exception handling is managed via five keywords: `try`, `catch`, `throw`, `throws`, and `finally`.
- Program statements that you think can raise exceptions are contained within a `try` block. If an exception occurs within the `try` block, it is thrown.
- Your code can catch this exception (using `catch` block) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword `throw`.
- Any exception that is thrown out of a method must be specified as such by a `throws` clause.
- Any code that absolutely must be executed after a `try` block completes is put in a `finally` block.

KEYWORDS: try AND catch

- The try block contains set of statements where an exception can occur.

```
try {
    // statement(s) that might cause exception
}
```

- The catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

```
catch (Exception ex) {
    // statement(s) that handle an exception
    // examples, closing a connection, closing
    // file, exiting the process after writing
    // details to a log file.
}
```

EXAMPLE: EXCEPTION HANDLINING

```
/ by zero
```

```java
package week11.examples;

public class Example11_3 {
    public static void main(String[] args) {
        int a = 1;
        int b = 0;

        try {
            int i = computeDivision(a, b);
            System.out.println(i);
        }
        // matching ArithmeticException
        catch (ArithmeticException ex) {
            // getMessage will print description of exception (here / by zero)
            System.out.println(ex.getMessage());
        }
    }

    static int computeDivision(int a, int b) {
        int res = 0;
        try {
            res = divideByZero(a, b);
        }
        // doesn't matches with ArithmeticException
        catch (NumberFormatException ex) {
            System.out.println("NumberFormatException is occurred");
        }
        return res;
    }

    static int divideByZero(int a, int b) {
        // this statement will cause ArithmeticException(/ by zero)
        int i = a / b;
        return i;
    }
}
```

KEYWORDS: throw, throws AND finally

- The `throw` keyword is used to transfer control from `try` block to `catch` block.
- The `throws` keyword is used for exception handling without `try` and `catch` block. It specifies the exceptions that a method can throw to the caller and does not handle itself.
- The `finally` block is executed after `catch` block. We basically use it to put some common code when there are multiple catch blocks.

EXAMPLE: EXCEPTION HANDLINING

```java
package week11.examples;

public class Example11_3 {

    static int i = 0;

    public static void main(String[] args) {
        int a = 1;
        int b = 0;

        try {
            i = computeDivision(a, b);
        }
        // matching ArithmeticException
        catch (ArithmeticException ex) {
        // getMessage will print description of exception (here / by zero)
            System.out.println(ex.getMessage());
        }
        finally {
            System.out.println(i);
        }
    }

    static void computeDivision(int a, int b) throws ArithmeticException {
        divideByZero(a, b);
        throw new ArithmeticException();
    }

    static void divideByZero(int a, int b) throws ArithmeticException {
        // this statement will cause ArithmeticException(/ by zero)
        i = a / b;
        throw new ArithmeticException(); // a new exception object is created
    }

}
```

```
/ by zero
0
```

EXAMPLE: TRY-CATCH CLAUSE

```java
package week11.examples;

public class Example11_4 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];
        // this statement causes an exception
        int i = arr[4];
        // the following statement will never execute
        System.out.println("Hi, I want to execute");
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
        at week11.examples.Example11_4.main(Example11_4.java:9)
```

```java
try {
    // block of code to monitor for errors
    // the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// optional
finally {
    // block of code to be executed after try block ends
}
```

- An array is defined with the size of 4, i.e. you can access elements only from index 0 to 3.
- But you trying to access the elements at index 4 (by mistake) that's why it is throwing an exception.
- In this case, JVM terminates the program abnormally. The statement `System.out.println("Hi, I want to execute");` will never execute.
- To execute it, we must handle the exception using try-catch. Hence to continue the normal flow of the program, we need a try-catch clause.

EXAMPLE: TRY-CATCH CLAUSE

```java
package week11.examples;

public class Example11_4 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];

        try {
            // this statement causes an exception
            int i = arr[4];
        }
        catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println(ex.getMessage());
        }
        finally {
            System.out.println("Hi, I want to execute");
        }
    }
}
```

```
Index 4 out of bounds for length 4
Hi, I want to execute
```

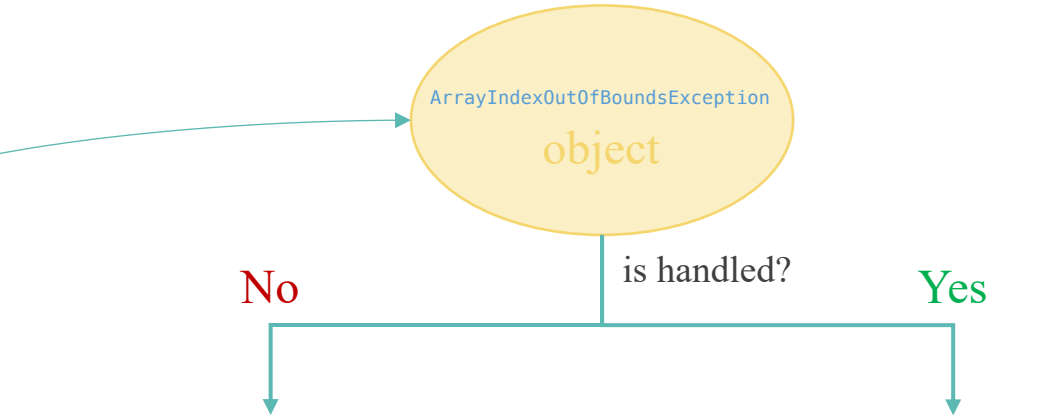
Points to remember:

- In a method, there can be more than one statement that might throw an exception, so put all these statements within their own `try` block and provide a separate exception handler within their own `catch` block for each of them.
- If an exception occurs within the `try` block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put a `catch` block after it. There can be more than one exception handlers. Each `catch` block is a exception handler that handles the exception of the type indicated by its argument. The argument, `ExceptionType` declares the type of exception that it can handle and must be the name of the class that inherits from the `Throwable` class.
- For each `try` block there can be zero or more `catch` blocks, but only one `finally` block.
- The `finally` block is optional. It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after `try` and `catch` blocks. And if an exception does not occur then it will be executed after the `try` block. The `finally` block in Java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

EXAMPLE: TRY-CATCH CLAUSE

```java
package week11.examples;

public class Example11_4 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];

        try {
            // this statement causes an exception
            int i = arr[4];
        }
        catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println(ex.getMessage());
        }
        finally {
            System.out.println("Hi, I want to execute");
        }
    }
}
```

```
Index 4 out of bounds for length 4
Hi, I want to execute
```



1. Print out exception description, i.e. what type of the exception occurred.
2. Print Stack Trace.
3. Terminates the running program.

Rest of the program will be executed.

EXCEPTION OCCURS IN TRY BLOCK AND HANDLED IN CATCH BLOCK

```java
package week11.examples;

public class Example11_5 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];

        try {
            // this statement causes an exception
            int i = arr[4];
            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }
        catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Exception caught in Catch block");
        }
        // rest program will be executed
        System.out.println("Outside try-catch clause");
    }
}
```

```
Exception caught in Catch block
Outside try-catch clause
```

- If a statement in `try` block raised an exception, then the rest of the `try` block doesn't execute and control passes to the corresponding `catch` block.
- After executing the `catch` block, the control will be transferred to the rest program will be executed.

EXCEPTION OCCURS IN TRY BLOCK AND HANDLED IN CATCH BLOCK

```java
package week11.examples;

public class Example11_5 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];

        try {
            // this statement causes an exception
            int i = arr[4];
            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }
        catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Exception caught in Catch block");
        }
        finally {
            System.out.println("finally block executed");
        }
        // rest program will be executed
        System.out.println("Outside try-catch-finally clause");
    }
}
```

```
Exception caught in Catch block
finally block executed
Outside try-catch-finally clause
```

- If a statement in `try` block raised an exception, then the rest of the `try` block doesn't execute and control passes to the corresponding `catch` block.
- After executing the `catch` block, the control will be transferred to `finally` block (if present) and then the rest program will be executed.

EXCEPTION OCCURRED IN TRY-BLOCK IS NOT HANDLED IN CATCH BLOCK

```java
package week11.examples;

public class Example11_6 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];

        try {
            // this statement causes an exception
            int i = arr[4];
            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }
        // not a appropriate handler
        catch (NullPointerException ex) {
            System.out.println("Exception has been caught");
        }

        // rest program will not execute
        System.out.println("Outside try-catch clause");
    }
}
```

- In this case, default handling mechanism is followed.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
        at week11.examples.Example11_6.main(Example11_6.java:10)
```

EXCEPTION OCCURRED IN TRY-BLOCK IS NOT HANDLED IN CATCH BLOCK

```java
package week11.examples;

public class Example11_6 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];

        try {
            // this statement causes an exception
            int i = arr[4];
            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }
        // not a appropriate handler
        catch (NullPointerException ex) {
            System.out.println("Exception has been caught");
        }
        finally {
            System.out.println("finally block executed");
        }

        // rest program will not execute
        System.out.println("Outside try-catch-finally clause");
    }
}
```

- If `finally` block is present, it will be executed followed by default handling mechanism.

```
finally block executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
        at week11.examples.Example11_6.main(Example11_6.java:10)
```

EXCEPTION DOESN'T OCCUR IN TRY-BLOCK

```java
package week11.examples;

public class Example11_7 {
    public static void main(String[] args) {
        try {
            String str = "123";
            int num = Integer.parseInt(str);
            // this statement will execute
            // as no any exception is raised by above statement
            System.out.println("Inside try block");
        }
        catch(NumberFormatException ex) {
            System.out.println("catch block executed...");
        }

        System.out.println("Outside try-catch clause");
    }
}
```

```
Inside try block
Outside try-catch clause
```

- In this case `catch` block never runs as they are only meant to be run when an exception occurs.
- Then the rest of the program will be executed.

EXCEPTION DOESN'T OCCUR IN TRY-BLOCK

```java
package week11.examples;

public class Example11_7 {
    public static void main(String[] args) {
        try {
            String str = "123";
            int num = Integer.parseInt(str);
            // this statement will execute
            // as no any exception is raised by above statement
            System.out.println("try block fully executed");
        }
        catch(NumberFormatException ex) {
            System.out.println("catch block executed...");
        }
        finally {
            System.out.println("finally block executed");
        }

        System.out.println("Outside try-catch-finally clause");
    }
}
```

```
try block fully executed
finally block executed
Outside try-catch-finally clause
```

- In this case `catch` block never runs as they are only meant to be run when an exception occurs.
- The `finally` block (if present) will be executed followed by rest of the program.

TRY-FINALLY CLAUSE

```java
package week11.examples;

public class Example11_8 {
    public static void main(String[] args) {
        // array of size 4.
        int[] arr = new int[4];

        try {
            // this statement causes an exception
            int i = arr[4];
            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }
        finally {
            System.out.println("finally block executed");
        }

        // rest program will not execute
        System.out.println("Outside try-finally clause");
    }
}
```

- In this case, no matter whether an exception occur in `try` block or not, `finally` will always be executed.
- But control flow will depend on whether exception has occurred in try block or not.
- Exception raised: if an exception has occurred in `try` block then control flow will be `finally` block followed by default exception handling mechanism.

```
finally block executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
        at week11.examples.Example11_8.main(Example11_8.java:10)
```

TRY-FINALLY CLAUSE

```java
package week11.examples;

public class Example11_9 {
    public static void main(String[] args) {
        try {
            String str = "123";
            int num = Integer.parseInt(str);
            // this statement will execute
            // as no any exception is raised by above statement
            System.out.println("Inside try block");
        }
        finally {
            System.out.println("finally block executed");
        }

        // rest program will be executed
        System.out.println("Outside try-finally clause");
    }
}
```

- Exception not raised: if an exception does not occur in `try` block then control flow will be `finally` block followed by rest of the program

```
Inside try block
finally block executed
Outside try-finally clause
```

BUILT-IN EXCEPTION: ArithmeticException

```java
package week11.examples;

public class Example11_10 {
    public static void main(String[] args) {
        try {
            int a = 30, b = 0;
            int c = a / b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch (ArithmeticException ex) {
            System.out.println ("Can't divide a number by 0");
        }
    }
}
```

```
Can't divide a number by 0
```

BUILT-IN EXCEPTION: NullPointerException

```java
package week11.examples;

public class Example11_11 {
    public static void main(String args[]) {
        try {
            String a = null; //null value
            System.out.println(a.charAt(0));
        } catch (NullPointerException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

```
Cannot invoke "String.charAt(int)" because "a" is null
```

BUILT-IN EXCEPTION: StringIndexOutOfBoundsException

```java
package week11.examples;

public class Example11_12 {
    public static void main(String args[]) {
        try {
            String a = "This is like chipping "; // length is 22
            char c = a.charAt(24); // accessing 25th element
            System.out.println(c);
        }
        catch (StringIndexOutOfBoundsException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

```
String index out of range: 24
```

BUILT-IN EXCEPTION: FileNotFoundException

```java
package week11.examples;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Example11_13 {
    public static void main(String args[]) {
        try {
            // Following file does not exist
            File file = new File("src/week11/examples/a.txt");
            FileReader fr = new FileReader(file);
        } catch (FileNotFoundException ex) {
            System.out.println("File does not exist");
            System.out.println(ex.getMessage());
        }
    }
}
```

```
File does not exist
src/week11/examples/a.txt (No such file or directory)
```

BUILT-IN EXCEPTION: NumberFormatException

```java
package week11.examples;

public class Example11_14 {
    public static void main(String args[]) {
        try {
            // "oop" is not a number
            double num = Double.parseDouble("oop");
            System.out.println(num);
        } catch (NumberFormatException ex) {
            System.out.println("Number format exception");
            System.out.println(ex.getMessage());
        }
    }
}
```

```
Number format exception
For input string: "oop"
```

BUILT-IN EXCEPTION: ArrayIndexOutOfBoundsException

```java
package week11.examples;

public class Example11_15 {
    public static void main(String args[]) {
        try {
            int a[] = new int[5];
            a[6] = 9; // accessing 7th element in an array of size 5
        }
        catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Array Index is Out Of Bounds");
            System.out.println(ex.getMessage());
        }
    }
}
```

```
Array Index is Out Of Bounds
Index 6 out of bounds for length 5
```

EXERCISE 1

- Handle all exceptions of the following program by using try-catch clause:

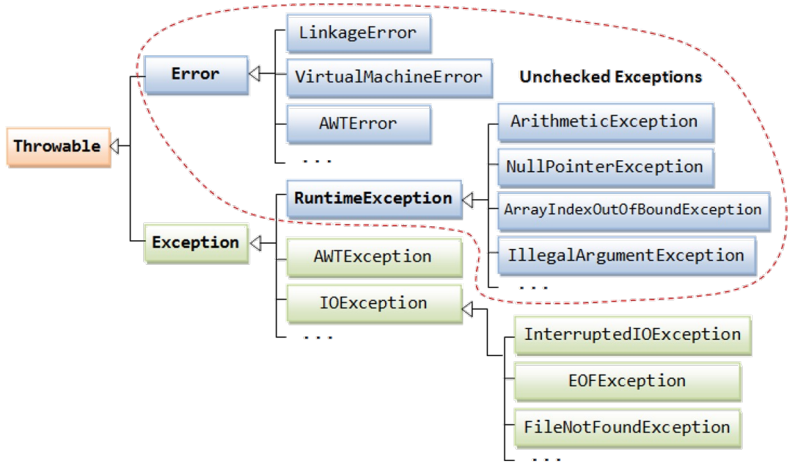
```
package week11.exercise;

public class Exercise1 {
    public static void main(String[] args) {
        int[] x = {1, 2, 3};
        String s1 = "Hello";
        String s2 = null;

        System.out.println(x[4]);
        System.out.println(s1.charAt(8));
        System.out.println(x[1] / 0);
        System.out.println(s2.length());

    }
}
```

```
Index 4 out of bounds for length 3
String index out of range: 8
/ by zero
Cannot invoke "String.length()" because "s2" is null
```

USER-DEFINED EXCEPTIONS

- Sometimes, the built-in exceptions in Java are not able to describe a certain situation.
- In such cases, users can also create exceptions which are called 'user-defined Exceptions'.
- The advantages of Exception Handling in Java are as follows:
  - Provision to complete program execution
  - Easy identification of program code and error-handling code
  - Propagation of errors
  - Meaningful error reporting
  - Identifying error types

CREATION OF USER-DEFINED EXCEPTION



- The user should create an exception class as a subclass of `Exception` class. Since all the exceptions are subclasses of `Exception` class, the user should also make the class a subclass of it. This is done as:

```
public class MyException extends Exception
```

- We can write a default constructor in it own exception class.

```
public MyException() {}
```

- We can also create a parameterised constructor with a string as a parameter. We can use this to store exception details. We can call super class (`Exception`) constructor from this and send the string there.

```
public MyException(String str) {
    super(str);
}
```

- To raise exception of user-defined type, we need to create an object to the exception class and throw it using `throw` clause, as:

```
MyException me = new MyException("Exception details");
throw me;
```

or

```
throw new MyException("Exception details");
```

EXAMPLE: USER-DEFINED EXCEPTION

```java
package week11.examples;

public class MyException extends Exception {
    // default constructor
    public MyException() {}

    // parameterised constructor
    public MyException(String str) {
        super(str);
    }
}
```

```java
package week11.examples;

// This program throws an exception whenever balance amount is below 1000.00
public class Example11_16 {
    //store account information
    private static int accno[] = {1001, 1002, 1003, 1004, 1005};
    private static String name[] = {"A", "B", "C", "D", "E"};
    private static double balance[] = {10000.00, 12000.00, 5600.0, 999.00, 1100.55};

    public static void main(String[] args) {
        try {
            // display the heading for the table
            System.out.println("ACCNO" + "\t" + "CUSTOMER" + "\t" + "BALANCE");
            // display the actual account information
            for (int i = 0; i < 5 ; i++) {
            System.out.println(accno[i] + "\t" + name[i] + "\t\t" + balance[i]);
            // display own exception if balance < 1000
                if (balance[i] < 1000) {
                    throw new MyException("Balance is less than 1000");
                }
            }
        }
        catch (MyException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
ACCNO   CUSTOMER        BALANCE
1001    A               10000.0
1002    B               12000.0
1003    C               5600.0
1004    D               999.0
week11.examples.MyException: Balance is less than 1000
        at week11.examples.Example11_16.main(Example11_16.java:19)
```

EXERCISE 2

- Student ID must length 8.
- Student ID must start with an uppercase letter and followed by 7 digits.
- Student name must separate between first name and last name (with “ ”).
- Student name must contain only letters.

- Create a program to check the following conditions by using exception handling (create your own one named `StudentIDFormatException`):

```
Please enter your student ID: 123456789
Please enter your name: Timothy   Dalton
ID length is not 8
```

```
Please enter your student ID: 12345678
Please enter your name: Timothy   Dalton
ID format not start with letter
```

```
Please enter your student ID: BM123456
Please enter your name: Timothy   Dalton
ID number format is incorrect
```

```
Please enter your student ID: B6300001
Please enter your name: Timothy   Dalton
Name must contain first name and last name
```

```
Please enter your student ID: B6300001
Please enter your name: TimothyDalton
Name must contain first name and last name
```

```
Please enter your student ID: B6300001
Please enter your name: Timothy1 Malton
Name must contain only letters
```

```
Please enter your student ID: B6300001
Please enter your name: Timothy Dalton
```