

# Introduction to Programming I

---

## Lab 10

Alexey Shikulin  
Furqan Haider  
Munir Makhmutov  
Sami Sellami

# Agenda

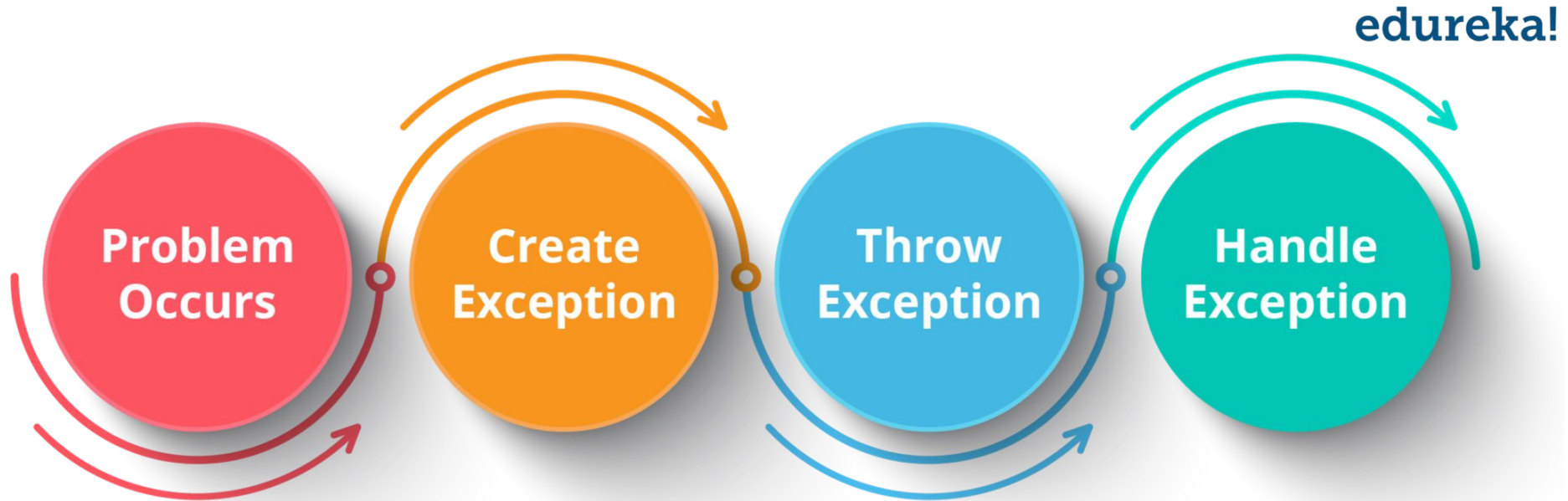
---

- Revision of exceptions in Java: from different angle
- Warm-up exercises
- Solving Problems
- Q&A

## Learning outcome:

- Strengthen the error-handling skills in Java
- Application of error-handling skills in real-world problems
- Code-review & improving code quality

# Exception flow



Exceptions...



# Exceptions in Programming

An exception can occur for many different reasons. 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.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

# Exception Categories

We have three categories of Exceptions:

- **Checked exception** - an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions
- **Unchecked exception (*Runtime Exception*)** - an exception that occurs at the time of execution.
- **Errors** - these are not exceptions, but problems that arise beyond the control of a user or a programmer.

# Exceptions...

```
public class Main {  
  
    public static void main(String args[]) {  
        int num[] = {1, 2, 3, 4};  
        System.out.println(num[5]);  
    }  
}
```

Is it **Checked exception**, or **Unchecked exception** or **Error**?

# Exceptions...

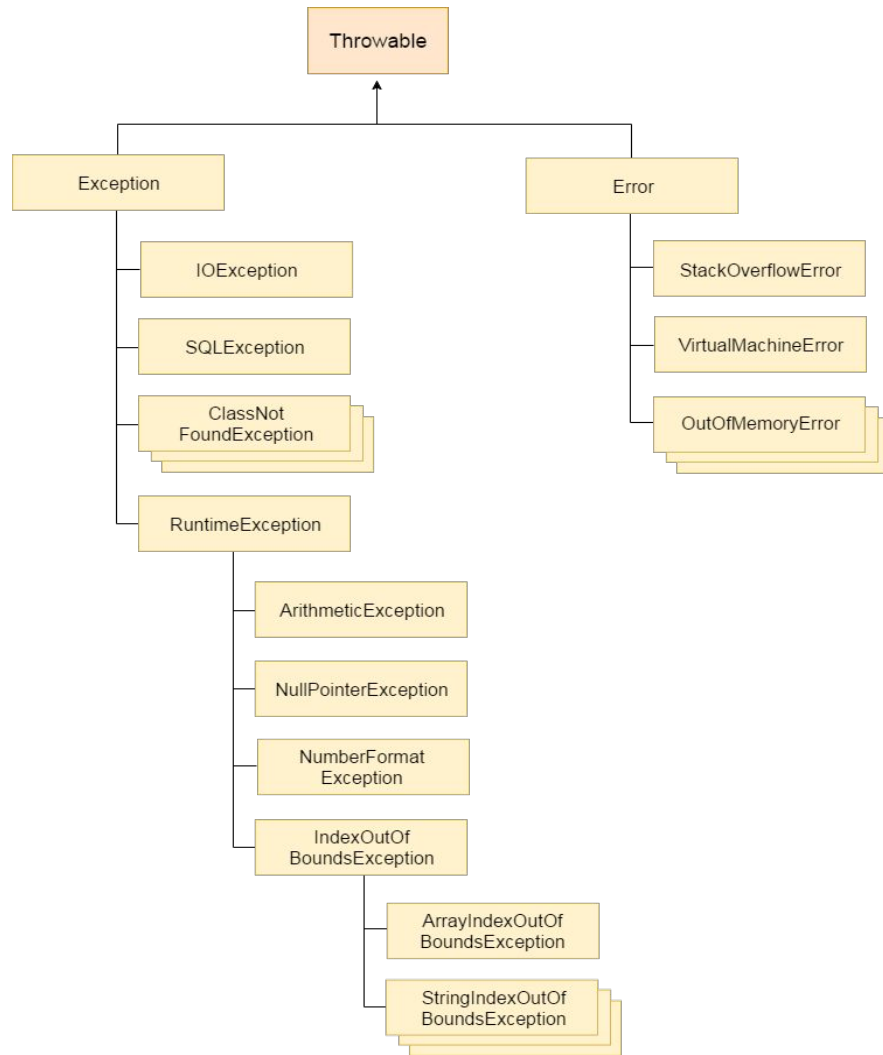
```
import java.io.File;
import java.io.FileReader;

public class Main {
    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

Is it **Checked exception**, or **Unchecked exception** or **Error**?



# Exceptions Hierarchy




# To Decrease Bugs...

Both good programmers and bad programmers make stupid mistakes. The difference is that good programmers:

- write code that is simpler and easier to debug,
- use tools such as JUnit to help ensure that their code is correct,
- are not satisfied with code that "mostly" works.

# For More Details...



The Java™ Tutorials

Hide TOCSearch

Exceptions

What Is an Exception?  
The Catch or Specify Requirement  
Catching and Handling Exceptions  
The try Block  
The catch Blocks  
The finally Block  
The try-with-resources Statement  
Putting It All Together  
Specifying the Exceptions Thrown by a Method  
How to Throw Exceptions  
Chained Exceptions  
Creating Exception Classes  
Unchecked Exceptions — The Controversy  
Advantages of Exceptions  
Summary  
Questions and Exercises

« Previous • Trail • Next »

Home Page > Essential Classes

*The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available. See [Java Language Changes](#) for a summary of updated language features in Java SE 9 and subsequent releases. See [JDK Release Notes](#) for information about new features, enhancements, and removed or deprecated options for all JDK releases.*

## Lesson: Exceptions

The Java programming language uses *exceptions* to handle errors and other exceptional events. This lesson describes when and how to use exceptions.

### What Is an Exception?

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

### The Catch or Specify Requirement

This section covers how to catch and handle exceptions. The discussion includes the `try`, `catch`, and `finally` blocks, as well as chained exceptions and logging.

### How to Throw Exceptions

This section covers the `throw` statement and the `Throwable` class and its subclasses.

### The try-with-resources Statement

<https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>

# Exception handling

If an error occurs, we can use **try...catch** to catch the error and execute some code to handle it.

We can attach a **finally-clause** to a try-catch block. The code inside the finally clause will always be executed.

Note that the finally-clause is **not mandatory**.

Also, it is legal that a **try** statement does not have a **catch** statement as long as it has a **finally** statement.

# Exception handling example

What exception types can be caught by the following handler?

```
public class CheckedException {  
    public static void main(String args[]) {  
        try {  
            int num[] = {1, 2, 3, 4};  
            System.out.println(num[5]);  
        } catch (Exception e) {  
            System.out.println("Exception here:");  
            e.printStackTrace();  
        } finally {  
            System.out.println("We reached the end.");  
        }  
    }  
}
```

# Warm Up Exercise

Is there anything wrong with the following exception handler as written?  
Will this code compile?

```
public class CheckedException {
    public static void main(String args[]) {
        try {
            int num[] = {1, 2, 3, 4};
            System.out.println(num[5]);
        } catch (Exception e) {
            System.out.println("Exception here:");
            e.printStackTrace();
        } catch (ArrayIndexOutOfBoundsException a) {
            System.out.println("ArrayIndexOutOfBoundsException exception here:");
            a.printStackTrace();
        }
        finally {
            System.out.println("We reached the end.");
        }
    }
}
```

# Handling More Than One Type of Exception

By the way, In Java SE 7 and later, a single `catch` block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

```
catch (IOException ex) {  
    ex.printStackTrace();  
}  
  
catch (SQLException ex) {  
    ex.printStackTrace();  
}
```

VS

```
catch (IOException|SQLException ex) {  
    ex.printStackTrace();  
}
```

# Warm Up Exercises

Match each situation in the first list with an item in the second list.

- A. `int[] A;`  
`A[0] = 0;`
- B. The JVM starts running your program, but the JVM can't find the Java platform classes. (The Java platform classes reside in `classes.zip` or `rt.jar`.)
- C. A program is reading a stream and reaches the `end of stream` marker.
- D. Before closing the stream and after reaching the `end of stream` marker, a program tries to read the stream again.

- 1. `__error`
- 2. `__checked exception`
- 3. `__compile error`
- 4. `__no exception`



# The try-with-resources Statement

It is a try statement that declares one or more *resources*. A *resource* is an object that must be closed after the program is finished with it.

```
static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br =  
        new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```

Because the `BufferedReader` instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly

# The try-with-resources Statement

Prior to Java SE 7, you can use a `finally` block to ensure that a resource is closed regardless of whether the `try` statement completes normally or abruptly. The following example uses a `finally` block instead of a `try-with-resources` statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                                    throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

# Suppressed Exceptions

```
public static void demoSuppressedException(String filePath) throws IOException {  
    FileInputStream fileIn = null;  
    try {  
        fileIn = new FileInputStream(filePath);  
    } catch (FileNotFoundException e) {  
        throw new IOException(e);  
    } finally {  
        fileIn.close();  
    }  
}
```

As long as we provide a path to an existing file, no exceptions will be thrown and the method will work as expected. However, suppose we provide a file that doesn't exist. In this case, the try block will throw a `FileNotFoundException`. Because the `fileIn` object was never initialized, it'll throw a `NullPointerException` when we try to close it in our finally block.

# Suppressed Exceptions

```
public static void demoSuppressedException(String filePath) throws IOException {  
    FileInputStream fileIn = null;  
    try {  
        fileIn = new FileInputStream(filePath);  
    } catch (FileNotFoundException e) {  
        throw new IOException(e);  
    } finally {  
        fileIn.close();  
    }  
}
```

Our calling method will only get the `NullPointerException`, and it won't be readily obvious what the original problem was: that the file doesn't exist.

We can take advantage of the `Throwable.addSuppressed` method to provide the original exception:

# Suppressed Exceptions

```
public static void demoAddSuppressedException(String filePath) throws IOException {
    Throwable firstException = null;
    FileInputStream fileIn = null;
    try {
        fileIn = new FileInputStream(filePath);
    } catch (IOException e) {
        firstException = e;
    } finally {
        try {
            fileIn.close();
        } catch (NullPointerException npe) {
            if (firstException != null) {
                npe.addSuppressed(firstException);
            }
            throw npe;
        }
    }
}
```

# Exercise 1

---

*(Extend your program from lab 5)*

Write a program that reads the data from a text file and writes it into another text file. Handle following exceptions:

- input file does not exist
- no write-permission for output file

# Exercise 1: Solution

---

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.FileNotFoundException;

class Task1 {
    public static void main(String[] args) {
        FileReader fr = null;
        FileWriter fw = null;
        try {
            fr = new FileReader("inputFile.txt");
            fw = new FileWriter("outputFile.txt");
            String str = "";
            int i;
            while ((i = fr.read()) != -1) {
                str += (char)i;
            }
            System.out.println(str);
            fw.write(str);
            System.out.println("File reading and writing both done.");
        }

        catch (FileNotFoundException e) {
            System.out.println("No file found !! ");
            e.printStackTrace();
        }

        catch (AccessControlException e) {
            System.out.println("Access denied ");
            e.printStackTrace();
        }

        catch (Exception e) {
            System.out.println("Exception occurred ");
            e.printStackTrace();
        }
    }
}
```

```
finally {
    if (fr != null) {
        try {
            fr.close();
        } catch (IOException io) {
            System.out.println("Exception while closing");
            io.printStackTrace();
        }
    }
    if (fw != null) {
        try {
            fw.close();
        } catch (IOException io) {
            System.out.println("Exception while closing");
            io.printStackTrace();
        }
    }
}
}
```

## Exercise 2

---

Write a program that reads an input from passed arguments (NOT from the console user typed), which accepts two integer parameters, and divides the first integer by the second.

You have to catch all the possible exceptions (such as, parsing errors, non-integer errors, arithmetic errors) and print the appropriate message.



# Exercise 2: Solution

---

```
import java.util.Scanner;
import java.util.InputMismatchException;

class Task2 {
    public static void main(String arg[]) {
        int a,b,c;
        Scanner sc = new Scanner(System.in);

        try
        {
            System.out.print("Enter first number: ");
            a = sc.nextInt();

            System.out.print("Enter second number: ");
            b = sc.nextInt();

            c = a/b;
            System.out.println("Result: " + c);
        }

        catch (InputMismatchException e) {
            System.out.println("InputMismatchException here: ");
            e.printStackTrace();
        }

        catch (ArithmeticException e) {
            System.out.println("ArithmeticException here:");
            e.printStackTrace();
        }

        catch (Exception e) {
            System.out.println("Exception occurred: ");
            e.printStackTrace();
        }

        finally {
            sc.close();
            System.out.println("End of the program...");
        }
    }
}
```

## Exercise 3

---

The method below downloads an image (actually any file). Your task is to edit the code so that it could handle all the possible exceptions.

```
public static void saveImage(String imageUrl) {
    URL url = new URL(imageUrl);
    String fileName = url.getFile();
    String destName = "./figures" + fileName.substring(fileName.lastIndexOf("/"));
    System.out.println(destName);

    InputStream is = url.openStream();
    OutputStream os = new FileOutputStream(destName);

    byte[] b = new byte[2048];
    int length;

    while ((length = is.read(b)) != -1) {
        os.write(b, 0, length);
    }

    is.close();
    os.close();
}
```

## Exercise 4

---

Extend your “Equipment rental” system adding feature to *serialize* and *deserialize* the equipment list (use *java.io.Serializable* for that purpose).

Handle all the exceptions that might occur in the process.

# References

---

- [https://www.tutorialspoint.com/java/java\\_exceptions.htm](https://www.tutorialspoint.com/java/java_exceptions.htm)
- <https://www.cis.upenn.edu/~matuszek/General/JavaSyntax/errors.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>
- <https://www.baeldung.com/java-exceptions>
- [https://www.protechtraining.com/content/java\\_fundamentals\\_tutorial-exceptions](https://www.protechtraining.com/content/java_fundamentals_tutorial-exceptions)