Unit IV

Exception Handling

Tutor: Pema Galey

Learning Outcomes

In this session, you will learn about:

- Errors vs Exception
- Types of Exception
- Exception handlers
- Annotations
- Assertions

What is the output?

```
class A {
  public static void main(String[] args) {
        int divideByZero = 5 / 0;
        System.out.println(”Hello everyone!");
        }
}
```

Output:

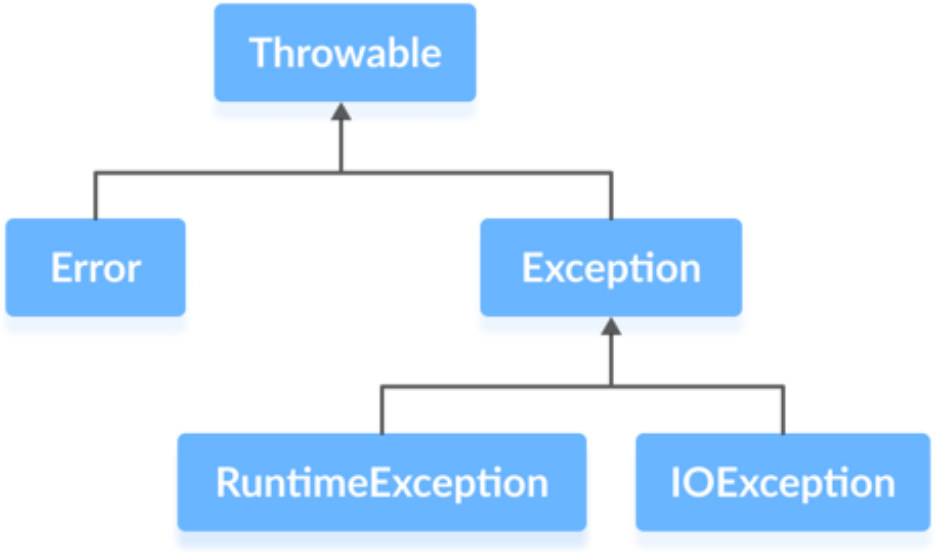
Exception in thread "main" java.lang.ArithmeticException: / by zero
at A.main(A.java:3)

Introduction: Exception

- An exception is an unexpected event that occurs during program execution.
- It affects the flow of the program instructions which can cause the program to terminate abnormally.
- 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

Java Exception hierarchy

- The exception hierarchy in Java.



- The Throwable class is the root class in the hierarchy.
- Note that the hierarchy splits into two branches: Error and Exception.

Error vs Exception

- **Errors** represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

- Errors are usually beyond the control of the programmer and we should not try to handle errors.

Error vs Exception

- **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 state of the program when the exception occurred.

Exception Types

- The exception hierarchy also has two branches:

1. **`RuntimeException` (also known as Unchecked Exception)**
2. **`IOException` (also known as Checked Exception)**

1. RuntimeException

- A **runtime exception** happens due to a programming error. They are also known as ***unchecked exceptions***.
- These exceptions are not checked at compile-time but at run-time. Some of the common runtime exceptions are:
  - Improper use of an API - **`IllegalArgumentException`**
  - Null pointer access (missing the initialization of a variable) - **`NullPointerException`**
  - Out-of-bounds array access - **`ArrayIndexOutOfBoundsException`**
  - Dividing a number by 0 - **`ArithmeticException`**

"If it is a runtime exception, it is your fault".

2. IOException

- An **IOException** is also known as a **checked exception**.
- They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.
- Some of the examples of checked exceptions are:
  - Trying to open a file that doesn't exist results in **`FileNotFoundException`**
  - Trying to read past the end of a file

Exception Handling

- List of different approaches to handle exceptions in Java (exception handlers):
  - `try`
  - `catch`
  - `finally`
  - `throw`
  - `throws`

1. try…catch block

- The try-catch block is used to handle exceptions in Java. Here's the syntax of *try...catch* block:

```
try {
        // code
} catch(Exception e) {
        // code
}
```

- Here, we have placed the code that might generate an exception inside the **try** block. Every **try** block is followed by a **catch** block.
- When an exception occurs, it is caught by the **catch** block. The **catch** block cannot be used without the **try** block.

Example: Exception handling using try...catch

```
class Main {
        public static void main(String[] args) {
            try {
                    // code that generate exception
                    int divideByZero = 5 / 0;
                    System.out.println("Rest of code in try block");
            } catch (ArithmeticException e) {
                    System.out.println("ArithmeticException => " + e.getMessage());
            }
        }
}
```

Output:
ArithmeticException => / by zero

2. Java finally block

- In Java, the **finally** block is always executed no matter whether there is an exception or not.
- The **finally** block is optional. And, for each try block, there can be *only one finally* block.
- The basic syntax of finally block is:

```
try {
        //code
} catch (ExceptionType1 e1) {
        // catch block
}
finally {
        // finally block always executes
 }
```

2. Java finally block

- We can also use the **try** block along with a finally block.
- The finally block is always executed whether there is an exception inside the try block or not.
- If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block.
- For each try block, there can be only one finally block.

Example: Exception Handling using try-finally block

```
class Main {
        public static void main(String[] args) {
                try {
                        int divideByZero = 5 / 0;
                }
                finally {
                   System.out.println("Finally block is always executed");
                }
        }
}
```

Output:
Finally block is always executed Exception in thread "main"
java.lang.ArithmeticException: / by zero at Main.main(Main.java:4)

Example: Exception Handling using try-catch-finally block

```
class Main {
        public static void main(String[] args) {
          try {
                // code that generates exception
                int divideByZero = 5 / 0;
           } catch (ArithmeticException e) {
                System.out.println("ArithmeticException => " + e.getMessage());
           }
        finally {
                System.out.println("This is the finally block");
        }
} }
```

Output:
ArithmeticException => / by zero
This is the finally block

Multiple Catch Blocks

- For each ***try*** block, there can be zero or more ***catch*** blocks.
  Multiple catch blocks allow us to handle each exception differently.

- The argument type of each catch block indicates the type of exception that can be handled by it.

Example: Multiple Catch

```
class ListOfNumbers {
        public int[] arr = new int[10];
        public void writeList() {
                try {
                        arr[10] = 11;
                } catch (NumberFormatException e1) {

        System.out.println("NumberFormatException => " +
                                        e1.getMessage());
                } catch (IndexOutOfBoundsException e2) {
        System.out.println("IndexOutOfBoundsException => "
                                + e2.getMessage());
                }
} }
```

```
class Main {
        public static void main(String[] args) {
                ListOfNumbers list = new
                        ListOfNumbers();
                list.writeList();
        }
}
```

Output:
IndexOutOfBoundsException => Index 10 out of bounds for length 10

Catching Multiple Exceptions

- We can catch more than one type of exception with one **catch** block.
- This reduces code duplication and increases code simplicity and efficiency.
- Each exception type that can be handled by the **catch** block is separated using a vertical bar |.
- Its syntax is:

```
try {
        // code
 } catch (ExceptionType1 | Exceptiontype2 ex) {
        // catch block
}
```

Example: Multiple Catch Block

```
class Main {
        public static void main(String[] args) {
            try {
                int array[] = new int[10];
                array[10] = 30 / 0;
            }
            catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
                System.out.println(e.getMessage());
            }
        }
}
```

Output:
/ by zero

3. Java throw and throws keyword

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

- When we throw an exception, the flow of the program moves from the try block to the catch block.

Example: Exception handling using Java throw

```
class Main {
        public static void divideByZero() {
                // throw an exception
                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:4)
        at Main.main(Main.java:7)
```

Java throws keyword

- We use the throws keyword in the method declaration to declare the type of exceptions that might occur within it.
- Its syntax is:

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

- As you can see from the above syntax, we can use throws to declare **multiple** exceptions.

Example 1: 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);
                }
        }
}
```

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.

Catching base Exception

- When catching multiple exceptions in a single catch block, the rule is generalized to specialized.

- This means that if there is a hierarchy of exceptions in the catch block, we can catch the base exception only instead of catching multiple specialized exceptions.

- We know that all the exception classes are subclasses of the **Exception** class. So, instead of catching multiple specialized exceptions, we can simply catch the Exception class.

Example: Catching base exception class only

```
class Main {
        public static void main(String[] args) {
                try {
                        int array[] = new int[10];
                        array[10] = 30 / 0;
                }
                catch (Exception e) {
                        System.out.println(e.getMessage());
                }
        }
}
```

Output
/ by zero

Custom Exception

- You can create your own exception by extending Exception class

```
class MyException extends Exception{}
```

Java Annotations

- Java annotations are metadata (data about data) for our program source code.
- They provide additional information about the program to the compiler but are not part of the program itself. These annotations do not affect the execution of the compiled program.
- Annotations start with @. Its syntax is:

    @AnnotationName

Example: Annotations

- Let's take an example of **@Override** annotation.
- The **@Override** annotation specifies that the method that has been marked with this annotation overrides the method of the superclass with the same method name, return type, and parameter list.
- It is not mandatory to use **@Override** when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

Note: There are different types of annotations

Example: Annotation

```
class Animal {
        public void displayInfo() {
                System.out.println("I am an animal.");
} }
class Dog extends Animal {
        @Override
        public void displayInfo() {
                System.out.println("I am a dog.");
} }
class Main {
        public static void main(String[] args) {
                Dog d1 = new Dog();
                d1.displayInfo();
        }
}
```

Output
I am a dog.

Types of Annotations

1. **Predefined annotations**

- @Deprecated
- @Override
- @SuppressWarnings
- @SafeVarargs
- @FunctionalInterface

2. **Meta-annotations**

- @Retention
- @Documented
- @Target
- @Inherited
- @Repeatable

Java Assertions

- Assertions in Java help to detect bugs by testing code we assume to be true.
- An assertion is made using the assert keyword.
- Its syntax is:

```
assert condition;
```

- Here, condition is a boolean expression that we assume to be true when the program executes.

Enabling Assertions

- By default, assertions are disabled and ignored at runtime.
- To enable assertions, we use:

  java -ea:arguments
- OR

  java -enableassertions:arguments

- When assertions are enabled and the condition is true, the program executes normally.
- But if the condition evaluates to false while assertions are enabled, JVM throws an AssertionError, and the program stops immediately.

Example: Java Assertion

```
class Main {
        public static void main(String args[]) {
                String[] weekends = {"Friday", "Saturday", "Sunday"};
                assert weekends.length == 2;
                System.out.println("There are " + weekends.length + "
                                        weekends in a week");
        }
}
```

Output (Before enabling assertion):
There are 3 weekends in a week

Output (After Enabling assertion):
Exception in thread "main" java.lang.AssertionError

Disabling Assertions

- To disable assertions, we use:

  java -da arguments

OR

  java -disableassertions arguments

- To disable assertion in system classes, we use:

  java -dsa:arguments

OR

  java -disablesystemassertions:arguments

- The arguments that can be passed while disabling assertions are the same as while enabling them.

Thank you!