Object Oriented Programming (OOP)

Lecture5: Exception Handling & JavaDocs

Prepared by:
Mohamed Mabrouk

Those slides are based on slides by:
Dr. Sherin Mousa and Dr. Sally Saad

Lecture Outline

- What exceptions are
- Exception handling
- try-catch & try-catch-finally
- Throwing exceptions
- User defined exceptions
- JavaDocs

Exceptions

Exceptions

- According to Oracle an exception is defined as: "An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions."
- "When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception."

Exceptions

- In other words, an exception is a problem/error that occurs during the normal flow of a program
- It causes the normal flow to get disrupted and the program terminates abnormally, unless there is an exception handling block to help handling it gracefully → Exception handling

Exceptions

- Can you think of some exceptions???

Exceptions

- Following is just a couple of exceptions defined in Java:
  - `ArithmeticException`
  - `ArrayIndexOutOfBounds-Exception`
  - `FileNotFoundException`
  - `NullPointerException`

Exception Handling

- Letting a program terminate abnormally is considered a bad practice
- Rather, exceptions should be handled gracefully, in order to:
  - Try to recover;
  - Retry operation;
  - Displaying meaningful error;
  - Terminating the whole program but in a better way.

Exception Handling Example

```
public float divide(int dividend, int divisor){
	return dividend/divisor;
}
```

Exception Handling Example

```
public float divide(int dividend, int divisor){
	return dividend/divisor;
}
```

- What if divisor is zero?

Exception Handling Example

```
public float divide(int dividend, int divisor){
	return dividend/divisor;
}
```

- What if divisor is zero? → a division by zero error should occur

Exception Handling Example

```
public float divide(int dividend, int divisor){
    return dividend/divisor;
}
```

- What if divisor is zero? → a division by zero error should occc
- This would lead to an `ArithmeticException`

Exception Handling Example

```
public float divide(int dividend, int divisor){
	try {
		return dividend / divisor;
	} catch (ArithmeticException exp){
		System.out.println("Invalid division");
		return -1F;
	}
}
```

Exception Handling Example

```
public void displayArrayItem(int[] arr, int index){
	System.out.println(arr[index]);
}
```

Exception Handling Example

```
public void displayArrayItem(int[] arr, int index){
	System.out.println(arr[index]);
}
```

What if index is -1, or >= size???

Exception Handling Example

```
public void displayArrayItem(int[] arr, int index){
	System.out.println(arr[index]);
}
```

What if index is -1, or >= size???

In that case an exception of type `ArrayIndexOutOfBoundsException` is thrown

Exception Handling Example

```
public void displayArrayItem(int[] arr, int index){
	try {
		System.out.println(arr[index]);
	} catch (ArrayIndexOutOfBoundsException exp){
		System.out.println("Invalid index");
	}
}
```

Exception Handling Example

```
public void displayArrayItem(int[] arr, int index){
	try {
		System.out.println(arr[index]);
	} catch (ArrayIndexOutOfBoundsException exp){
		System.out.println("Invalid index");
	}
}
```

In that case the program will terminate gracefully and displays an indicative error message

Types of Exceptions

- There are two types of exceptions
  - Checked exceptions
  - Unchecked exceptions

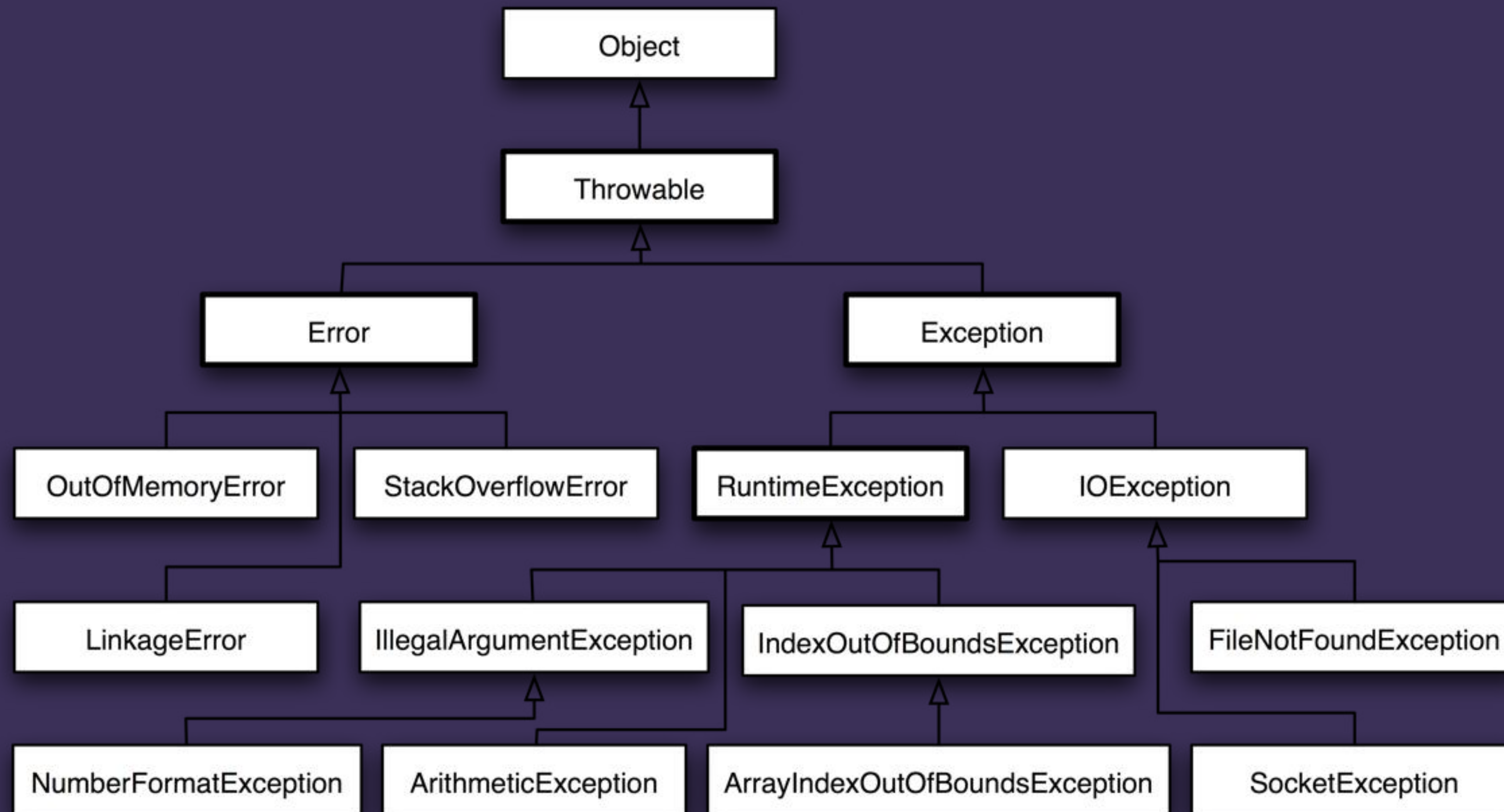
Checked Exceptions

- Checked exceptions are the ones that the program should anticipate and handle
- They are checked at compile time → you *MUST* handle them, otherwise you get a compilation error
- You have to handle them using `try-catch` or `try-catch-finally`
- *MUST EXTEND* `Throwable` class either directly or indirectly
- Examples are; `FileNotFoundException`, and `PrinterException`

Unchecked Exceptions

- Unchecked exceptions are exceptions that occur outside the program
- Most of the time cannot be expected or recovered from
- They are not checked at compile time → you *MAY,* or *MAY NOT* handle them.
- No compilation error if not handled
- You can handle them using `try-catch` or `try-catch-finally`
- *MUST EXTEND* `RuntimeException` class either directly or indirectly
- Examples are; `NullPointerException`, and `ClassCastException`

Exception Hierarchy



Exception Example

```
public void writeArrayItemsToFile(int[] arr, int length){
	PrintWriter printWriter = new PrintWriter(new FileWriter(”E:/input.txt"));
	for (int i = 0; i < length; i++) {
		printWriter.write(arr[i] + "\t");
	}
	 printWriter.close();
}
```

Exception Example

```
public void writeArrayItemsToFile(int[] arr, int length){
        PrintWriter printWriter = new PrintWriter(new FileWriter(”E:/input.txt"));
        for (int i = 0; i < length; i++) {
                printWriter.write(arr[i] + "\t");
        }
         printWriter.close();
}
```

- Two types of exceptions can be thrown here; `IOException` and `ArrayIndexOutOfBoundsException`

Exception Example

- Compiler will prompt you about `IOException` only
- Why???

Exception Example

- Compiler will prompt you about `IOException` only
- Why??? → IOException is a checked exception while `ArrayIndexOutOfBoundsException` is not
- In order to resolve that error, this code fragment should be enclosed in a `try-catch` block

Exception Example

```
public void writeArrayItemsToFile(int[] arr, int length){
	try{
		PrintWriter printWriter = new PrintWriter(new FileWriter(”E:/input.txt"));
		for (int i = 0; i < length; i++) {
			printWriter.write(arr[i] + "\t");
		}
		printWriter.close();
	} catch(IOException exp){
		System.out.println(exp.getMessage());
	}
}
```

Exception Example

```
public void writeArrayItemsToFile(int[] arr, int length){
	try{
		PrintWriter printWriter = new PrintWriter(new FileWriter(”E:/input.txt"));
		for (int i = 0; i < length; i++) {
			printWriter.write(arr[i] + "\t");
		}
		printWriter.close();
	} catch(IOException exp){
		System.out.println(exp.getMessage());
	}
}
```

- There is one minor problem with that code, can you spot it???

Multiple Catch Blocks

- Sometimes a code block may throw several types of exceptions → multiple catch blocks are required

Multiple Catch Blocks

- Sometimes a code block may throw several types of exception → multiple catch blocks are required

```
try{
    ....
} catch(IOException exp){
    System.out.println(exp.getMessage());
} catch(ArithmaticException exp){
    System.out.println(exp.getMessage());
} catch(Exception exp){
    System.out.println(exp.getMessage());
}
```

Catching All Exceptions

- Since all exceptions extend class `Exception`, if there is a catch block for `Exception`, then all exceptions will be caught by that catch block
- The exceptions should be ordered from the very special to the very general → `Exception` class *MUST BE* placed at the very end

Finally Block

- `finally` block is the last part of try-catch-finally block
- It always gets executed regardless of whether an exception is thrown or not

Finally Example

- Moving back to that example → can we improve it

```
public void writeArrayItemsToFile(int[] arr, int length){
	try{
		PrintWriter printWriter = new PrintWriter(new FileWriter(”E:/input.txt"));
		for (int i = 0; i < length; i++) {
			printWriter.write(arr[i] + "\t");
		}
		printWriter.close();
	} catch(IOException exp){
		System.out.println(exp.getMessage());
	}
}
```

Finally Example

- What if an exception is thrown, will the `PrintWriter` get closed?

Finally Example

- What if an exception is thrown, will the `PrintWriter` get closed?
- The answer is *NO*
- The `PrintWriter` will never get closed reserving unwanted memory
- In that case, we can use `finally` block
- `finally` block will get executed regardless of whether there is an exception or not

Finally Example

```
public void writeArrayItemsToFile(int[] arr, int length){
	PrintWriter printWriter = null;
	try{
		printWriter = new PrintWriter(new FileWriter(”E:/input.txt"));
		for (int i = 0; i < length; i++) {
			printWriter.write(arr[i] + "\t");
		}
	} catch(IOException exp){
		System.out.println(exp.getMessage());
	} finally {
		if(printWriter!=null) {
			printWriter.close();
		}
	}
}
```

Throwing an Exception

- If in your method, there is a critical error that you need to notify about → you can throw an exception
- You may throw one of those:
  - one of Java standard exceptions, e.g. `ArithmaticException`
  - define your own exception class (user defined exception)
- In order to declare that a method throws and exception, you can use `throws` keyword

User Defined Exception

- Let's think of a simple calculator, what happens if the numbers are very large?
- We need all numbers to be between 1 and 100, otherwise an exception is thrown

User Defined Exception Example

```
public class NumberRangeException extends Exception {
	public NumberRangeException(int lowerBound, int upperBound){
	super("The number must be between " + lowerBound + " and " + upperBound);
	}
}
```

User Defined Exception Example

```
public class NumberRangeException extends Exception {
	public NumberRangeException(int lowerBound, int upperBound){
	super("The number must be between " + lowerBound + " and " + upperBound);
	}
}
```

User Defined Exception Example

```
public class NumberRangeException extends Exception {
	public NumberRangeException(int lowerBound, int upperBound){
	super("The number must be between " + lowerBound + " and " + upperBound);
	}
}
```

```
public int add(int num1, int num2) throws NumberRangeException{
	if(num1 < 1 || num1 > 100 || num2 <1 || num2 > 100){
		throw new NumberRangeException(1, 100);
	}
	return num1 + num2;
}
```

User Defined Exception Example

```
public class NumberRangeException extends Exception {
	public NumberRangeException(int lowerBound, int upperBound){
	super("The number must be between " + lowerBound + " and " + upperBound);
	}
}
```

```
public int add(int num1, int num2) throws NumberRangeException{
	if(num1 < 1 || num1 > 100 || num2 <1 || num2 > 100){
		throw new NumberRangeException(1, 100);
	}
	return num1 + num2;
}
```

JavaDocs

What are JavaDocs

- JavaDocs are used to document your own code and/or APIs
- This enables other users to understand and use them
- JavaDocs can be generated based on your code
- IDEs, e.g. NetBeans and IntelliJ can help you generate them

JavaDocs Format

- Once JavaDocs are generated, a couple of HTML files containing documentation are generated
- The classes are grouped by their respective packages
- The documentation has to be enclosed in /** …. */

JavaDoc Example

```
/**
* This is a simple calculator
*/
public class Calculator {
	/**
	* Adds two numbers and returns their sum
	* @param num1 First number
	* @param num2 Second number
	* @return The sum of the two numbers
	*/
	public int add(int num1, int num2){
		return num1 + num2;
	}
}
```

JavaDoc Output

Package org.example.string

Class Summary

Class	Description
Calculator	This is a simple calculator

Constructors

Constructor	Description
`Calculator()`	

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`int`	`add(int num1, int num2)`	Adds two numbers and returns their sum

Thank You!