

Exception Handling

Handling Errors During the Program Execution



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

sli.do

#java-advanced

1. What Are Exceptions in Java?
 - The **Throwable** and **Exception** Classes
 - Types of Exceptions and Their Hierarchy
2. Handling Exceptions: **try-catch-finally**
3. Raising (Throwing) Exceptions: **throw**
4. **Best Practices** in Exception Handling
5. **Defining Custom** Exceptions Classes





What Are Exceptions?

Notifications about Failed Operations

What Are Exceptions?

- **Exceptions** handle **errors** and **problems** at runtime
- **Throw** an exception to **signal** about a problem

```
if (size < 0)
    throw new Exception("Size cannot be negative!");
```

- **Catch** an exception to **handle** the problem

```
try {
    size = Integer.parseInt(text);
} catch (Exception ex) {
    System.out.println("Invalid size!");
}
```



More About Exceptions



- **Exceptions** occur when the normal flow of the program is interrupted due to a problem (or error)
 - When an operation **fails to execute** at runtime
 - **Example:** trying to read a non-existing file
- **Exceptions** allow problematic situations to be **handled** at multiple levels
 - Simplify code construction and maintenance
- **Exception objects** hold detailed information about the error: **error message, stack trace**, etc.

Unhandled Exception with Stack Trace

```
int x = Integer.parseInt("invalid number");
```



```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Communi
Exception in thread "main" java.lang.NumberFormatException: Create breakpoint : For input string: "invalid number"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:668)
    at java.base/java.lang.Integer.parseInt(Integer.java:786)
    at ExceptionExample.main(ExceptionExample.java:5)
```

Stack trace

Error message

The Throwable Class

- Exceptions in Java are **objects**
- The **Throwable** class is a base for all Java exceptions
 - Contains information for the cause of the problem
 - **Message** – a text description of the exception
 - **StackTrace** – the snapshot of the "call stack" at the moment when the exception is throws



Types of Exceptions in Java

- All Java exceptions inherit from **java.lang.Throwable**
- Direct descendants of **Throwable**:
 - **Error** – not expected to be caught from the program under normal circumstances
 - Examples: **StackOverflowError**, **OutOfMemoryError**
 - **Exception**
 - Used for exceptional conditions that user programs could catch
 - Examples: **ArithmeticException**, **IOException**

- **Exceptions** are two types:
 - **Checked** – an exceptions that should be obligatory handled → checked by the compiler during the compilation

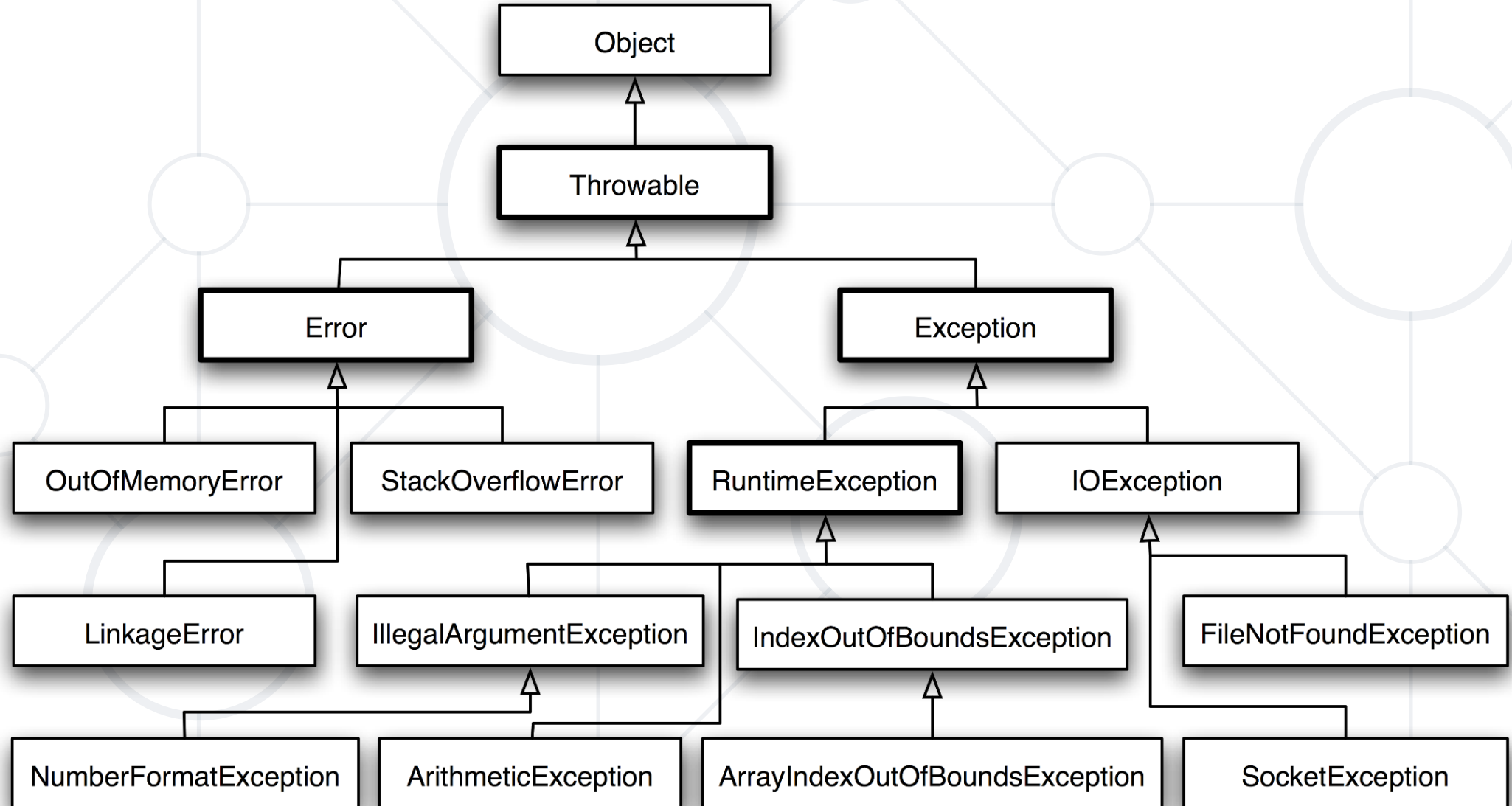
- Also called **compile-time** exceptions

```
public static void main(String args[]) {  
    File file = new File("non-existing-file.txt");  
    FileReader fr = new FileReader(file);  
}
```

FileNotFoundException

- **Unchecked** – exceptions that occur at the time of execution
 - Also called **runtime exceptions**, not obligatory handled

Exception Hierarchy in Java





Handling Exceptions

Using **try-catch-finally**

How Do Exceptions Work?



Handling Exceptions

- In Java exceptions can be handled by the **try-catch** construction

```
try {  
    // Do some work that can raise an exception  
} catch (SomeException) {  
    // Handle the caught exception  
}
```



- **catch** blocks can be used multiple times to process different exception types

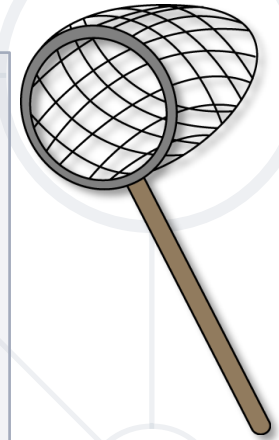


Using try-catch – Example

```
Scanner scanner = new Scanner(System.in);
String s = scanner.nextLine();
try {
    Integer.parseInt(s);
    System.out.printf(
        "You entered a valid integer number: %s", s);
} catch (NumberFormatException ex) {
    System.out.println("Invalid integer number!");
}
```

- When **catching an exception** of a particular class, all its **descendants** (child exceptions) are caught too, e.g.

```
try {  
    // Do some work that can cause an exception  
} catch (IndexOutOfBoundsException iobEx) {  
    // Handle the caught out-of-bounds exception  
}
```



- Handles **IndexOutOfBoundsException** and its descendants **ArrayIndexOutOfBoundsException** and **StringIndexOutOfBoundsException**

Find the Mistake!

```
String str = scanner.nextLine();

try {
    Integer.parseInt(str);
} catch (Exception ex) {
    System.out.println("Cannot parse the number!");
} catch (NumberFormatException ex) {
    System.out.println("Invalid integer number!");
}
```

Should be last

Unreachable code

Problem: Number in Range

- Write a program to **enter an integer** in a **certain range**, e. g. 10-20
 - Read a **range** (two integers **start** \leq **end**) and print the range
 - When an invalid number is entered or the number is out of range, print "**Invalid number: {num}**" and enter a number again
 - When the entered number is valid, print "**Valid number: {num}**"

```
10 20
5
xx
20
```



```
Range: [10...20]
Invalid number: 5
Invalid number: xx
Valid number: 20
```

```
-5 50
hi
-6
-1
```



```
Range: [-5...50]
Invalid number: hi
Invalid number: -6
Valid number: -1
```

Solution: Number in Range

```
private static int readNumberInRange(
    Scanner scanner, int start, int end) {
    while (true) {
        String line = scanner.nextLine();
        try {
            int num = Integer.parseInt(line);
            if (num >= start && num <= end)
                return num; // Valid number (in range)
        } catch (Exception ex) {
            // Parse failed --> invalid number
        }
        System.out.println("Invalid number: " + line);
    }
}
```

Solution: Number in Range

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    String[] range = scanner.nextLine().split(" ");  
    int start = Integer.parseInt(range[0]);  
    int end = Integer.parseInt(range[1]);  
    System.out.printf("Range: [%d...%d]\n", start, end);  
    int num = readNumberInRange(scanner, start, end);  
    System.out.println("Valid number: " + num);  
}
```



Try-Finally

Executing a Cleanup Code in All Cases

The try-finally Statement

- The statement:

```
try {  
    // Do some work that can cause an exception  
} finally {  
    // This block will always execute  
}
```

- Ensures execution of a given block in all cases
 - When an **exception** is raised or **not** in the **try** block
- Used for execution of **cleaning-up code**, e.g. releasing resources

Try-finally – Example

```
static void tryFinallyExample() {  
    System.out.println("Code executed before try-finally.");  
    try {  
        String str = scanner.nextLine();  
        Integer.parseInt(str);  
        System.out.println("Parsing was successful.");  
        return; // Exit from the current method → executes the "finally" block  
    } catch (NumberFormatException ex) {  
        System.out.println("Parsing failed!");  
    } finally {  
        System.out.println("This cleanup code is always executed.");  
    }  
    System.out.println("This code is after the try-finally block.");  
}
```



Throwing Exceptions

Using the **"throw"** Keyword

- **Throwing** an exception with an error message:

```
throw new IllegalArgumentException("Invalid amount!");
```

- Exceptions can accept **message** and **cause** (nested exception):

```
try {  
    ...  
} catch (SQLException sqlEx) {  
    throw new IllegalStateException("Cannot save invoice", sqlEx);  
}
```

- **Note:** if the original exception is not passed, the initial cause of the exception is lost

- Exceptions are **thrown** (raised) by the **throw** keyword
- Used to **notify the calling code** in case of an error or unusual situation
- When an exception is thrown:
 - The program execution **stops immediately**
 - The exception **travels over the stack**
 - Until a matching **catch** block is reached to handle it
- **Unhandled exceptions** display an error message

- Caught exceptions can be **re-thrown** again:

```
try {  
    Integer.parseInt(str);  
} catch (NumberFormatException ex) {  
    System.out.println("Parse failed!");  
    throw ex; // Re-throw the caught exception  
}
```

Throwing Exceptions – Example

```
public static double calcSqrt(double value) {  
    if (value < 0)  
        throw new ArithmeticException(  
            "Sqrt for negative numbers is undefined!");  
    return Math.sqrt(value);  
}  
  
public static void main(String[] args) {  
    try {  
        calcSqrt(-1);  
    } catch (ArithmeticException ex) {  
        System.err.println("Error: " + ex.getMessage());  
        ex.printStackTrace();  
    }  
}
```

Problem: Square Root

- Write a program that **reads an integer** number and calculates and prints its **square root** (with 2 digits after the decimal point)
 - If the number is **invalid** or **negative**, print "**Invalid**"
- In all cases finally print "**Goodbye**"
- Use **try-catch-finally**

9 → 3.00
Goodbye

20 → 4.47
Goodbye

xx → Invalid
Goodbye

-5 → Invalid
Goodbye

Solution: Square Root

```
Scanner scanner = new Scanner(System.in);
try {
    int num = Integer.parseInt(scanner.nextLine());
    double sqrt = calcSqrt(num);
    System.out.printf("%.2f\n", sqrt);
} catch (Exception ex) {
    System.out.println("Invalid");
} finally {
    System.out.println("Goodbye");
}
```



The "throws" in Method Declarations

Forcing Invokers to Handle Certain Exceptions

Using "throws" in Method Declaration

```
static String readTextFile(String fName) throws IOException {  
    BufferedReader reader =  
        new BufferedReader(new FileReader(fName));  
    StringBuilder result = new StringBuilder();  
    try {  
        String line;  
        while ((line = reader.readLine()) != null)  
            result.append(line + System.lineSeparator());  
    } finally {  
        reader.close();  
    }  
    return result.toString();  
}
```


Invoking Method Declared with "throws"

```
public static void main(String[] args) {  
    String fileName = "./src/TextFileReader.java";  
    try {  
        String sourceCode = readTextFile(fileName);  
        System.out.println(sourceCode);  
    } catch (IOException ioex) {  
        System.err.println("Cannot read file: " + fileName);  
        ioex.printStackTrace();  
    }  
}
```

Catching **IOException**
is obligatory!

Throwing from the Main Method

- The `main()` method can declare as "**throws**" all exception classes, which it refuses to handle

```
public static void main(String[] args)
    throws IOException {
    FileWriter file = new FileWriter("example.txt");
    file.write("Some text in the file");
    file.close();
}
```



Custom Exceptions

Declaring Your Own Exception Class

- Custom exceptions inherit an exception class (commonly – **Exception**)

```
public class FileParseException extends Exception {  
    private int lineNumber;  
  
    public FileParseException(String msg, int lineNumber) {  
        super(msg + " (at line " + lineNumber + ")");  
        this.lineNum = lineNumber;  
    }  
  
    public int getLineNum() { return lineNumber; }  
}
```

- Throw your exceptions like any other:

```
throw new FileParseException(  
    "Cannot read setting", 75);
```

```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains'  
Exception in thread "main" FileParseException: Cannot read setting (at line 75)  
    at CustomExceptionsExample.main(CustomExceptionsExample.java:3)
```

- If your exception derives from **Exception** → handle it obligatory
- If it derives from **RuntimeException** → handle it optionally



Best Practices

Working with Exceptions the Right Way

- The **catch** blocks should:
 - Begin with the exceptions **lowest** in the hierarchy
 - Continue with the more general exceptions
 - Otherwise, a **compilation error** will occur
- Each **catch** block should handle only these exceptions, which it expects
 - If a method is not competent to handle an exception, it should leave it unhandled
 - Handling all exceptions disregarding their type is a popular **bad practice** (anti-pattern)!

- When an application attempts to use **null** in a case where an object is required: **NullPointerException**
- A method has been passed an illegal or inappropriate argument: **IllegalArgumentException**
- An array has been accessed with an illegal index: **ArrayIndexOutOfBoundsException**
- An index is either negative or greater than the size of the string: **StringIndexOutOfBoundsException**

- Attempt to convert an inappropriate string to a numeric type: **NumberFormatException**
- When an exceptional arithmetic condition has occurred: **ArithmeticException**
- Attempt to cast an object to a subclass of which it is not an instance: **ClassCastException**
- When a file or network or other input / output operation has failed: **IOException**

- When throwing an exception, always pass to the constructor a **good explanation message**
 - The **error message** should make obvious what the problem is
 - The exception message should explain **what causes the problem** (and give directions **how to solve it**)
 - **Good**: "Size should be integer in range [1...15]"
 - **Good**: "Invalid state. First call Initialize()"
 - **Bad**: "Unexpected error"
 - **Bad**: "Invalid argument"

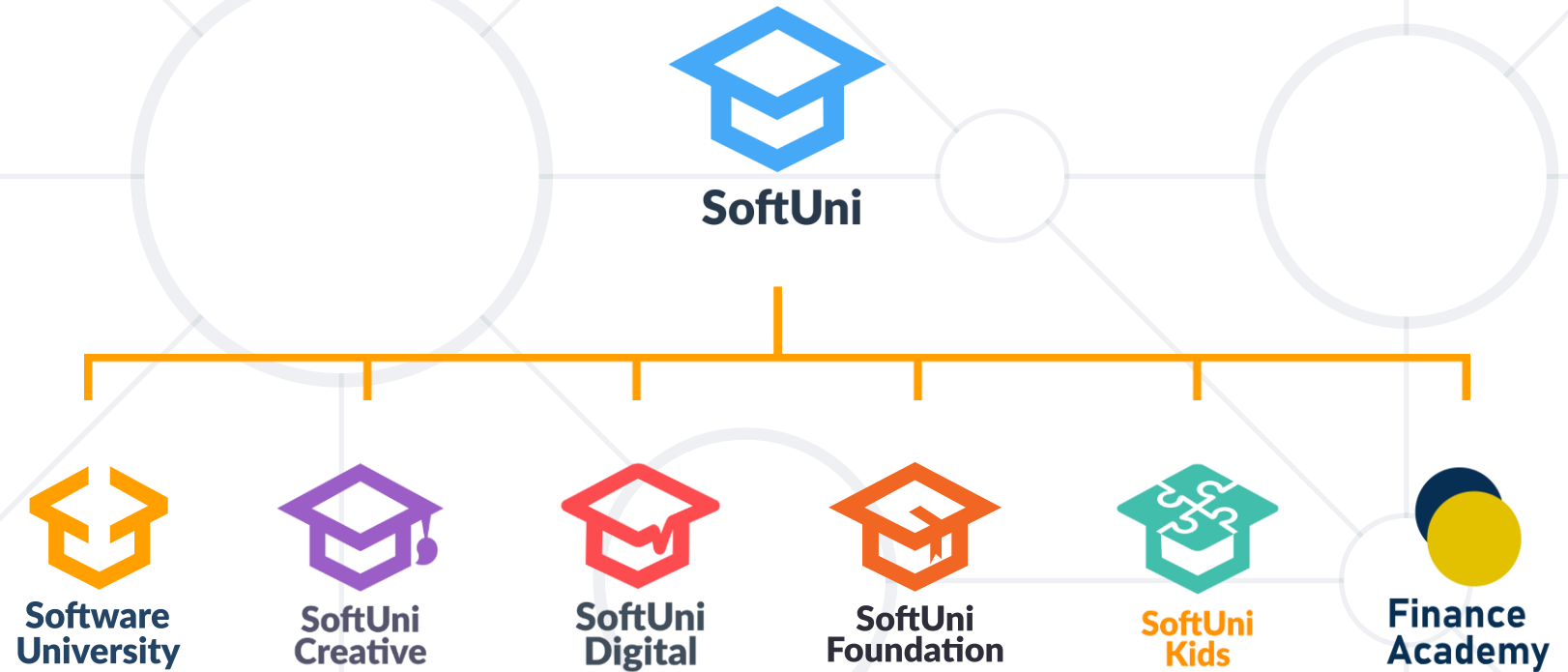


- Exceptions can decrease the application **performance**
 - Throw exceptions only in situations which are really exceptional and should be handled
 - Do not throw exceptions in the normal program control flow
 - The JVM could throw exceptions at any time with no way to predict them
 - E. g. **StackOverflowError** or **OutOfMemoryError**

- **Exceptions** provide a **flexible** error handling mechanism
- **Try-catch** allows exceptions to be handled
- **Unhandled exceptions** crash with an error message
- **Try-finally** ensures a given code block is always executed



Questions?



SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

