3TC36: Object-Oriented Programming in Java

Exception handling and logging

Dominique Blouin
dominique.blouin@telecom-paris.fr
Le March 20, 2026

Learning objectives

- Errors during program execution
- Exceptions in Java
- Application exceptions
- Logging

Error handling in programming

- Handling errors that may occur during the execution of a program is **essential**.
- Poor (or lack of) error handling can be the cause of **several problems**.
- This is an aspect that is often **overlooked** in many software applications.
- It can lead to **unexpected** program behaviour...

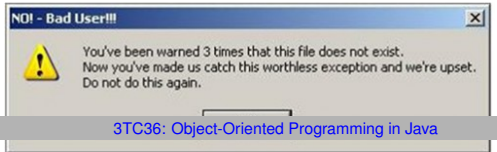
Example of poor Error Handling

- Or discouraging error messages...

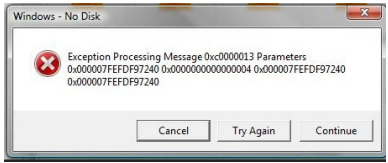
The following errors were found, please correct them and re-submit the form:

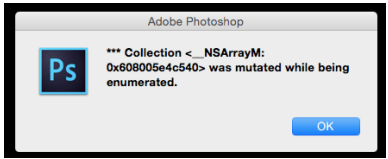
- Fields must not be the same:
- Password must be at least 6 characters
- Forename is a required field.
- Please enter a valid email address.
- Surname is a required field.
- Confirm Email is a required field.
- Password is a required field.
- Confirm Password is a required field.
- Job Title is a required field.
- Name of organisation is a required field.
- Head office based in is a required field.
- Company Sector is a required field.
- No. of employees is a required field.
- Company Turnover is a required field.
- The value you entered for the captcha validation was incorrect.

- Or condescending / arrogant ...

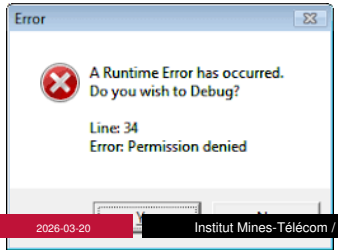


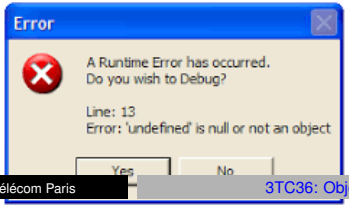
Unclear or confusing error messages (technical) or messages that do not provide information

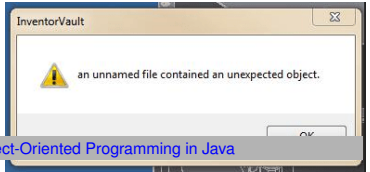




- Or messages that have nothing to do with the real problem...







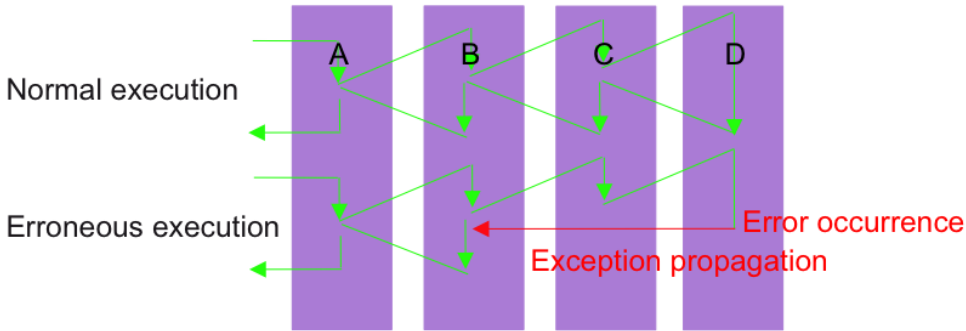
Error handling that potentially reveals confidential data (security)

```
1  try {
2  openDbConnection();
3  }
4  //print exception message that includes exception message and configuration file location
5  catch (Exception $e) {
6  echo 'Caught exception: ', $e->getMessage(), '\n';
7  echo 'Check credentials in config file at: ', $Mysql_config_location, '\n';
8  }
9
```

Error handling in Java

- An error typically occurs in a method, which was called by a method, which was called by a method, and so on up to the initial `main` method.
- When the error occurs, Java creates an object of type `Exception` that contains information about the error.
  - It is said that an exception is **thrown**.
- The exception **travels up** the sequence of method calls to the initial main method.
  - It is said that the exception is **propagated** to the `main()` method.
- The program stops and an error message is displayed on the console, taking into account the information collected by the **exception** object during its propagation.

Execution trace of a program, error occurrence, and exception propagation



A common error in Java

```
java.lang.Exception: java.lang.NullPointerException
        at utils.CheckData.checkNotEmpty(CheckData.java:63)
        at submission.Submission.handle(Submission.java:146)
        at Main.main(Main.java:16)
```

- We see the **nested method calls**: `main` (at the bottom) called `handle` which called `checkNotEmpty` where the error occurred on line 63.
- We also see the **line numbers** of these calls in the source files.
- We see the **name of the error** and the **line number of the file** where it occurred.
- `Note:` In Eclipse, clicking on the exception trace automatically opens the relevant file at the given line.

Different types of errors in Java

- First type: **Severe** errors.
  - They cannot be handled by the programmer and almost always cause the **program to stop**.
- They are represented by objects of the `Error` class.
- **Examples:**
  - Out of memory in the virtual machine (`OutOfMemoryError`).

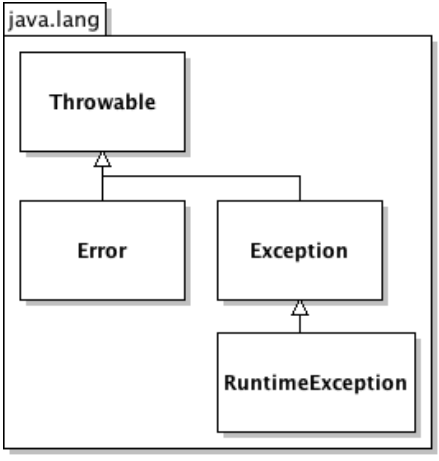
Different types of errors in Java

- Second type: Errors that must be handled by the **programmer**.
- They are represented by objects of the `Exception` class.
- **Example:**
  - Input/output errors (`IOException`).

Different types of errors in Java

- Third type: Errors related to the **language**.
- This is a special case of the second type.
- Represented by objects of the `RuntimeException` class
  - Which is a subclass of the `Exception` class.
- **Example:**
  - Arithmetic errors (division by 0).
    - `ArithmeticException`

Class diagram of exceptions



- All **propagated** errors are subclasses of the `Throwable` class.

Error propagation

- When an error occurs, an object of the `Throwable` class is created and propagated back to the initial `main()` method.
- This object can be of the following types:
  - `Error` (severe errors).
  - `Exception` (common errors).
  - Subclass of `Exception` (specific errors).
  - `RuntimeException` (language errors).
  - Subclass of `RuntimeException` (specific errors).
- Subclasses of `Exception` or `RuntimeException` are specializations representing **specific types of errors**.
- Search: JAVA SE Exception.

Error handling

- It is possible to **stop the propagation** of an error using a `try-catch` statement:

```
try {
    ... // instructions that
    ... // may trigger an error
}
catch (Exception ex) {
    ... // instructions to execute
    ... // in case of an error
}
```

- When error propagation is **stopped**, **substitute** instructions located in the `catch` block are **executed**, where the `Exception` object is retrieved.

Error handling

- During the execution of a block:

```
try {
    // ...
}
catch (Exception ex) {
    // ...
} // ...
```

- The instructions inside the `try` block are executed.
  - If no exception is thrown and propagated during the execution of this block, the instructions after the `catch` block are executed.
  - If an exception is thrown, it is caught and stored in the `ex` variable, and the code inside the `catch` block is executed.
- Inside the `catch` block, we can:
  - Display a specific error message.
  - Correct the error conditions and perform another calculation.
  - Etc.

Example of handling a severe error (`Error` class)

- These errors **cannot be fixed**, so we simply display an error message.

```
public static void main(String[] args) {
    int size = Integer.parseInt(args[0]) ;

    try {
        List<Integer> manyInts = new ArrayList<Integer>();
        for (int index = 0; index < size; index++) {
            manyInts.add(new Integer(0));
        }
    }
    catch (Error err) {
        System.out.println("Error: " + err);
    }
}
```

- If the value of `size` is large enough, we might encounter:
  - Error: java.lang.OutOfMemoryError: Java heap space

Language error handling

- Exceptions of type `RuntimeException` correspond to ordinary errors related to the execution of the program.
- Search: JAVA SE ArrayList
  - Let's look at the `get(int index)` method.
- This method can throw an exception of type `IndexOutOfBoundsException`, which is a subclass of `RuntimeException`.

Selective error handling

- It is possible to **selectively** handle exceptions.

```
try {
    // ...
}
catch (IndexOutOfBoundsException ex) {
    // ...
}
catch (ArithmeticException ex) {
    // ...
}
catch (Exception ex) {
    // ...
}
```

- Only the first `catch` block whose type matches the propagated exception will be executed.
- Therefore, the **order** of `catch` blocks is important.

The `finally` clause

- The `finally` clause is placed at the end of a `try-catch` block.
- It allows specifying a sequence of instructions that will be executed **after** the execution of the `try-catch` block, **regardless of how** the execution of that block was completed.

```
try {
    // ...
}
catch (IndexOutOfBoundsException ex) {
    // ...
}
catch (Exception ex) {
    // ...
}
finally {
    // ...
}
```

Classic example of using the `finally` clause

```
try {
    ... // opening a file for reading
    ... // reading and processing the file
}
catch (IOException ex) {
    ... // displaying an I/O error message
}
catch (Exception ex) {
    ... // handling other errors
}
finally {
    ... // closing the file
}
```

- This ensures the file is closed **no matter what happens**.

Displaying the call stack trace

- Research: JAVA SE Exception.
- An important method is `printStackTrace()`, which displays the sequence of method calls at the moment the error occurred.

```
try {
    // ...
}
catch (Exception ex) {
    // ...
    ex.printStackTrace();
    // ...
}
```

Programming the triggering of an exception

- A programmer can **trigger** an exception using the `throw` statement:

```
public String computeFullName(String firstName, String lastName) {

    if (firstName == null || firstName.isEmpty()) {
        throw new Exception("First name is invalid");
    }

    if (lastName == null || lastName.isEmpty()) {
        throw new Exception("Last name is invalid");
    }

    return firstName + " " + lastName;
}
```

Checked exceptions

- The previous `computeFullName()` method **does not compile**!
- All exceptions other than `RuntimeException` are called **checked exceptions**.
  - `RuntimeException` (and its subclasses): unchecked exceptions.
- If a method can throw a **checked exception**, it must **declare** this with the `throws` keyword:

```
public String computeFullName(String firstName, String lastName)
throws Exception {

    if (firstName == null || firstName.isEmpty()) {
        throw new Exception("First name is invalid");
    }
    if (lastName == null || lastName.isEmpty()) {
        throw new Exception("Last name is invalid");
    }

    return firstName + " " + lastName;
}
```

- Otherwise, there will be a **compilation error**.

Defining custom exceptions

- Programmers can define their **own** exceptions as **subclasses** of the `Exception` class.

```
public class InvalidFirstNameException extends Exception  {

    private final String name; // Data embedded into the exception

    public InvalidFirstNameException(String name) {
        super("Invalid first name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

Defining custom exceptions

- Another subclass of `Exception`.

```
public class InvalidLastNameException extends Exception  {

    private final String name; // Data embedded into the exception

    public InvalidLastNameException(String name) {
        super("Invalid last name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

Using custom exceptions

- The programmer can **throw** custom exceptions:

```
public String computeFullName(String firstName, String lastName)
throws InvalidFirstNameException,
       InvalidLastNameException {

    if (firstName == null || firstName.isEmpty()) {
        throw new InvalidFirstNameException(firstName);
    }

    if (lastName == null || lastName.isEmpty()) {
        throw new InvalidLastNameException(lastName);
    }

    return firstName + " " + lastName;
}
```

Handling your own exceptions

```
try {
   // ...
   String fullName = computeFullName(firstName, lastName);
   // ...
}
catch (InvalidFirstNameException ex) {
    String name = ex.getName();
    ... // Display error message to log file or error dialog box
}
catch (InvalidLastNameException ex) {
   // ...
}
catch (Exception ex) {
   // ...
}
```

Exercise:
Propose a *better modelling* of these two exceptions

```java
public class InvalidFirstNameException extends
     Exception {
    private final String name; // Data
        embedded into the exception

    public InvalidFirstNameException(String
        name) {
        super("Invalid first name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

```java
public class InvalidLastNameException extends
    Exception {
    private final String name; // Data
        embedded into the exception

    public InvalidLastNameException(String
        name) {
        super("Invalid last name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

Good use of exceptions

- Is this implementation good?

```
public Student findStudent(int index) {
    try {
        return studentsList.get(index);
    }
    catch (IndexOutOfBoundsException ex) {
        return null;
    }
}
```

Use exceptions only for exception handling!

```
public Student findStudent(int index) {

    if (index < 0 || index >= studentsList.size()) {
        return null;
    }

    return studentsList.get(index);
}
```

Use standard control structures (better performance)

Logging

- Involves managing the emission and storage of **messages** (traces) following various events that occur during the execution of a program.
- Until now, we have mainly used the `System.out.println()` method to display execution messages to the console.
- What if we want to **keep these messages** (in a file for example), whether it is for:
  - Debugging.
  - Obtaining execution traces (start/stop, information, warnings, runtime errors, ...).
  - Or facilitating the search for a source of anomaly (stacktrace, ...).
- `System.out.println()` is then clearly **insufficient**...

Logging

- To solve this problem, a **Logging** API is used.
- This API typically involves three main components:
  - **Logger**: Emits messages, usually with an associated severity level.
  - **Formatter**: Formats the content of the messages.
  - **Appender**: Sends the message to a specified **storage target** (console, file, database, email, etc.).
- Different logging libraries:
  - **Apache Log4j**: One of the earliest logging frameworks.
  - **Java Logging**: Included in the JDK starting from Java 4.
  - **SLF4J** (Simple Logging Facade for Java): A flexible abstraction for various logging frameworks.
  - Others...

Example with JDK logging

```
import java.util.logging.Logger;

public class Main {

    private static final Logger LOGGER = Logger.getLogger(Main.class.getName());

    public static void main(String[] args) {
        LOGGER.info("Starting the simulation...");

        LOGGER.config("With parameters " + Arrays.toString(args) + "...");
            // ...

        try {
            // ...
        }
        catch (Exception ex) {
            LOGGER.log(Level.SEVERE, "Error...", ex);
        }
    } // ...
```

Logging levels and message display

- Different logging levels:
  - Level.SEVERE: initialized with value 1000
  - Level.WARNING: initialized with value 900
  - Level.INFO: initialized with value 800
  - Level.CONFIG: initialized with value 700
  - Level.FINE: initialized with value 500
  - Level.FINER: initialized with value 400
  - Level.FINEST: initialized with value 300
- Then, we configure the logger level for the application (or per appender or handler): `LOGGER.setLevel(Level.INFO);`
- For each call to the log method, the message will be displayed only if the message's logging level is equal to or higher than the application's level:

```
LOGGER.info("Starting the simulation..."); // Will be displayed
LOGGER.config("With parameters " + Arrays.toString(args) + "...");
                                          // Will not be displayed
```

Defining the logging level of the application

- In a configuration file named `logging.properties`:

```
# Limit the messages that are printed on the console to FINE and above.
java.util.logging.ConsoleHandler.level = FINE
```

- Specify the **location** of the configuration file:
  - By default, a file is provided in the directory `jre/conf/logging.properties`
  - You can redefine the location and name of the file via a JVM argument:

```
-Djava.util.logging.config.file=<my application dir>/config/logging.properties
```

- Define the logging level from the code:

```
LOGGER.setLevel(Level.INFO);
```

Example with JDK logging

- The `Handler` class:
  - `StreamHandler`: Sends messages to an output stream.
  - `ConsoleHandler`: Sends messages to the standard error output.
  - `FileHandler`: Sends messages to a file.
  - `SocketHandler`: Sends messages over a network socket.
- Example with `FileHandler`:

```
Handler fileHandler;

try {
    fileHandler = new FileHandler("TestLogging.log");
    LOGGER.addHandler(fileHandler);
}
// Catches any of SecurityException or IOException
catch (SecurityException | IOException ex) {
    LOGGER.log(Level.SEVERE, "Error...", ex);
}
```

Conclusion

- The Java error handling mechanism is very powerful.
- It allows programmers to handle errors more easily.
  - The programmer lets the potential error occur and then catches the exception using a `try-catch` block to process / handle it.
- The use of exceptions should be **reserved for handling exceptional situations** where performance is not critical.
- Use **loggers** as much as possible. This is an essential feature for complex applications.
- Avoid directly using `System.out.println(...)` and `ex.printStackTrace()`.
  - The `Logger` internally uses these elements when needed.