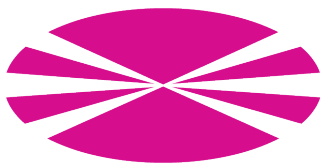
Practice 2: Exceptions in Java

Software Design (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Department of Computer Science and Information Technology
Faculty of Computer Science



UNIVERSIDADE DA CORUÑA

Table of Contents

Table of Contents

Error management

- **Traditional approach**
  - Methods return an error code.
  - For example, the function `fopen` in C tries to open a file. If this fails, `fopen` returns `null`.
- **Drawbacks of the traditional approach**
  - Whoever calls the function must remember to catch the return value.
  - The program may end up consisting in a sequence of error checks.
  - Whoever calls the function may not be prepared to deal with the error, and may need to delegate this task.

Error management

Definition of Exception (Computer Science)

Anomalous or exceptional event that occurs while running a program and must be addressed in a special way.

- Exceptions are constructions provided by programming languages, intended for error management.
- Programming languages provide the means for creating, catching and processing exceptions or, alternatively, delegating this task to other parts of the program.
- Exceptions must not be used for normal, predictable events (e.g., reaching the end of a list).

Benefits of exceptions

- Separating error management code from "regular" code.
- Allowing to propagate errors easily and safely, following the call stack until they can be properly addressed.
- Allowing to categorize and group distinct types of errors into separate classes.

Exception classes

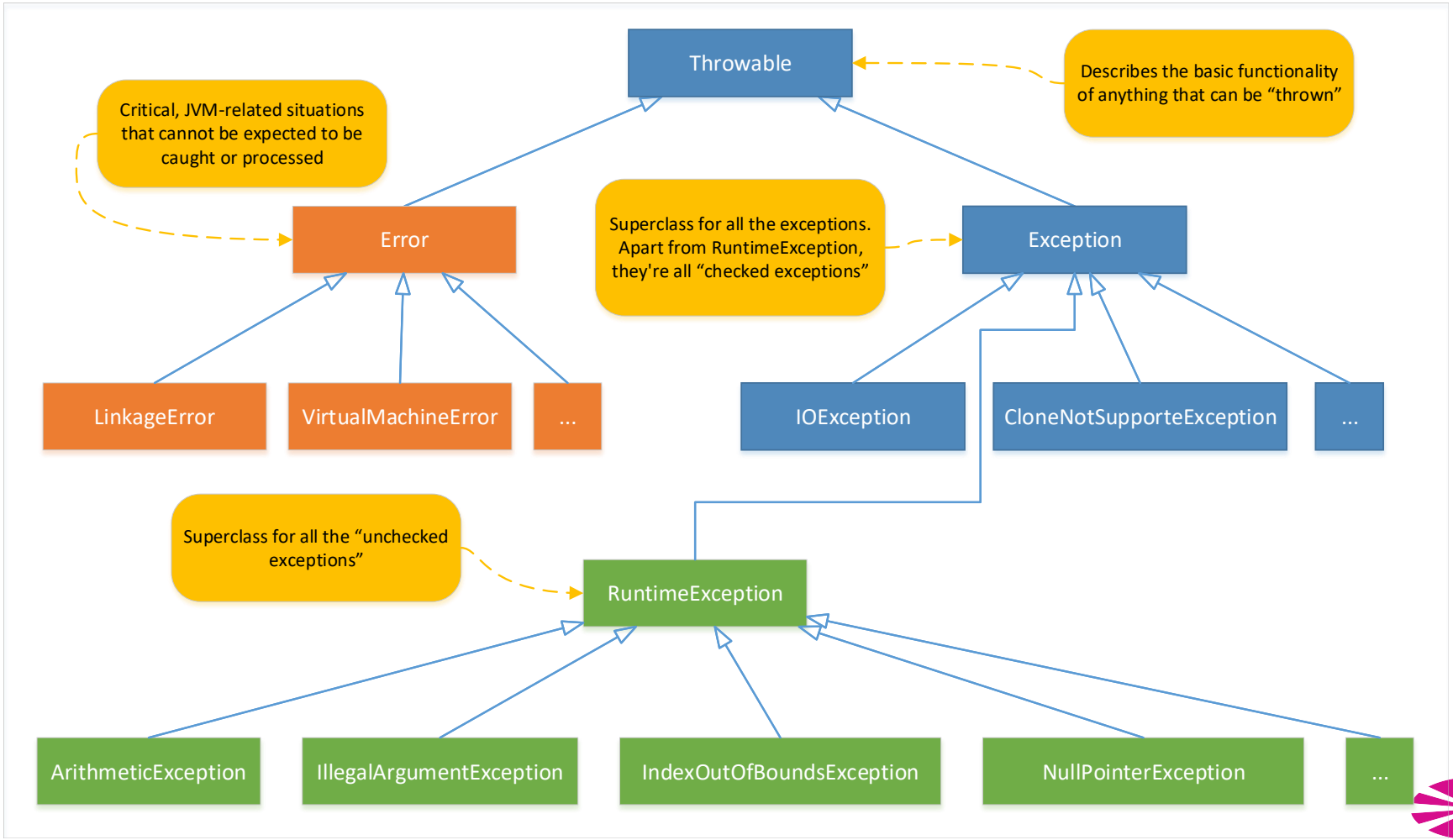


Table of Contents

Throwing exceptions

- Exceptions are thrown using the keyword `throw` and an object of the class `Throwable` or any of its subclasses.

Clause `throws`

```
throw new ExceptionClass();
```

Catching exceptions

- Catching of exceptions is done using the `try-catch-finally` block.

`try-catch-finally` block

```
try {
    // some code
} catch (ExceptionClass name) {
    // manages ExceptionClass
} finally {
    // frees up resources
}
```

- **`try` block**
  - Code to execute that can throw exceptions.
- **`catch` block**
  - Declare one or more exceptions to manage.
  - If the exceptions are thrown in the `try` block the control is transfered to the corresponding `catch` block.
- **`finally` block**
  - Executed always at the end.
  - They are useful to free up resources (close a file, a network connection, etc.).

Catching exceptions

- A `try` statement can be followed by multiple `catch` clauses (as long as they catch different exception classes).
- Since Java version 7, it is also possible to include multiple exception classes in the *same* `catch` clause.

Catching multiple types of exceptions with the same statement

```
public static void init4() {
    try {
        Files.inputFiles("colors.txt");
    }
    // Two exceptions are caught simultaneously
    catch (FileNotFoundException | FileSystemException e) {
        System.out.println("Problem: " + e.getMessage());
        System.out.println("Enter a valid file name...");
    }
    // Another catch clause, if the previous one is not executed
    catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

Table of Contents

Exception classes

Unchecked exceptions

Out-of-the-ordinary situations that are internal to the program, and difficult to predict and recover from.

- They are typically programming *bugs* or incorrect API calls.
- For example, null pointers as parameters, wrong array indexes, "illegal" mathematical operations (e.g., division by zero).
- The programmer is not forced to catch and process them. It is optional to do so (assuming it is even possible).
- Because it is optional to deal with them, uncaught exceptions may cause a program to stop unexpectedly (admittedly, this is sometimes the best way to identify bugs).

Throwing unchecked exceptions

- The method simply throws the exception.
- Whoever calls the method should catch the exception (but is under no obligation to do so). Failing to address the problem will stop the execution.

Throwing an unchecked exception

```
public Object pop() {
    Object obj;

    if (size == 0) { // I cannot return anything
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

Catching unchecked exceptions

- Catching unchecked exceptions is optional (the compiler does not force to do so).

Catching unchecked exceptions

```
// We make no attempt to catch exceptions
public void removeOneElement() {
    pop();  // If an exception occurs, execution stops
}

// If the EmptyStackException occurs, a message is shown
public void removeTwoElements() {
    try {
        pop();
        pop();
    } catch (EmptyStackException e) {
        System.out.println("Stack is empty!");
    }
}
```

Managing Preconditions

- The best way of implementing preconditions in methods is through the use of exceptions as you inform the client that he has not fulfilled his part in the contract).
- A typical exception used in preconditions is `IllegalArgumentException` that is unchecked.
- Invariants and postconditions are better managed using assertions.

Checking preconditions with exceptions

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 *           rate > MAX_REFRESH_RATE.
 */
 public void setRefreshRate(int rate) {
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
         throw new IllegalArgumentException("Illegal rate: " + rate);

     setRefreshInterval(1000/rate);
```

Table of Contents

Exception classes

Checked exceptions

Out-of-the-ordinary situations that can be nevertheless predicted and recovered from.

- They are typically errors that prevent a program from running as expected, be it due to user error (e.g., incorrect information) or environment error (e.g., file does not exist).
- Checked exceptions **force** the programmer to deal with them, as they are typically resolvable (e.g., ask for a new filename).

Throwing checked exceptions

- Throwing checked exceptions (such as `FileNotFoundException`) forces us to modify the method's signature to warn that the method may throw that exception (adding `throws ExceptionName` to the signature).
- Whoever calls our method is, in turn, forced to deal with the exception.

Throwing a checked exception

```
// inputFiles warns that it may throw an exception
public static void inputFiles(String name) throws FileNotFoundException {
    File inputFile = new File(name);
    if (!inputFile.exists()) { // Checked exception
        throw new FileNotFoundException("File not found");
    }
    ColorSet colorSetter = new ColorSet(inputFile);
}
```

Catching checked exceptions

- Catching is mandatory (otherwise a compilation error will occur).

Catching checked exceptions (version 1)

```
// We neglect checking for exceptions
public static void init1() {
    Files.inputFiles("colors.txt"); // Compilation error!
}

// If the FileNotFoundException occurs, a message is shown
public static void init2() {
    try {
        Files.inputFiles("colors.txt");
    } catch (FileNotFoundException e) {
        System.out.println("Problem: " + e.getMessage());
        System.out.println("Enter a valid file name...");
    }
}
```

Catching checked exceptions

- Alternatively, a method can avoid the responsibility of directly catching a checked exception if it appends a `throws` clause to its signature.
- This way, the exception is sent to an upper level, in which it can be managed properly.

Catching checked exceptions (version 2)

```
public static void init3() throws FileNotFoundException {
    Files.inputFiles("colors.txt");
}

// The caller must now catch the exception
public static void main(String[] args) {
    Files.init3(); // Compilation error!
}
```

Controversy over checked exceptions

Argument

- If you can recover from an exception, it ought to be defined as a checked exception.
- Otherwise, it should be an unchecked exception.

Controversy over checked exceptions

- **Problems**
  - Whether one can recover or not often depends on the context. A `FileNotFoundException` is fatal if the file does not exist, but it can be recovered from if the problem is simply a wrong filename.
  - Checked exceptions are explicitly declared in a method's signature $\Rightarrow$ They expose internal implementation and break encapsulation.
- **Consequences**
  - APIs that overuse checked exceptions can be cumbersome and may force programmers to catch exceptions trivially if they want compilation to succeed $\Rightarrow$ The real errors may remain hidden.
  - Even though checked exceptions can reduce the number of runtime errors, the consensus is to avoid them if possible.

Table of Contents

Assertions

Assertion

A sentence to check the veracity of assumptions such as invariants, postconditions, etc.

Format

```
assert boolean_expression;
assert boolean_expression : expression_2;
```

- The boolean expression is evaluated. If it's false, the `AssertionError` is thrown.
- `expression_2` is evaluated as a `String` and is passed as information to the `AssertionError` constructor.

Assertions

- Assertions are used to test invariants.

Example 1

```
if (i % 3 == 0) {
    // ...
} else if (i % 3 == 1) {
    // ...
} else {
    assert i % 3 == 2 : i;
    // ...
}
```

Example 2

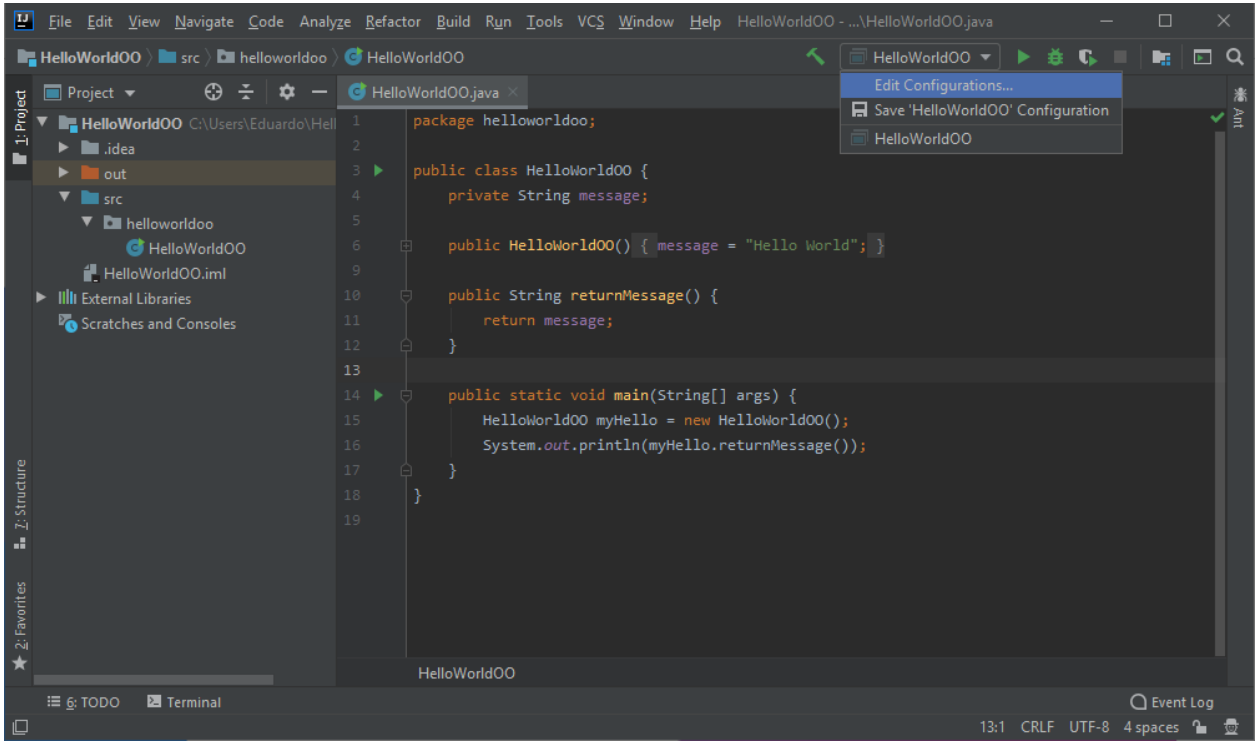
```
switch (suit) {
    case SPADES: /* ... */ break;
    case HEARTS:   /* ... */ break;
    case CLUBS:  /* ... */ break;
    case DIAMONDS:    /* ... */ break;
    default:
        assert false : "Wrong suit: " + suit;
}
```

Benefits of assertions

- Facilitating the detection of errors.
- Documenting code.
- Can be enabled and disabled using compiler directives: `ea` (enable assertions) and `da` (disable assertions; by default).
  - `java -ea MyClass` Executes MyClass with the assertions activated
  - `java -da MyClass` Executes MyClass with the assertions deactivated (default version)

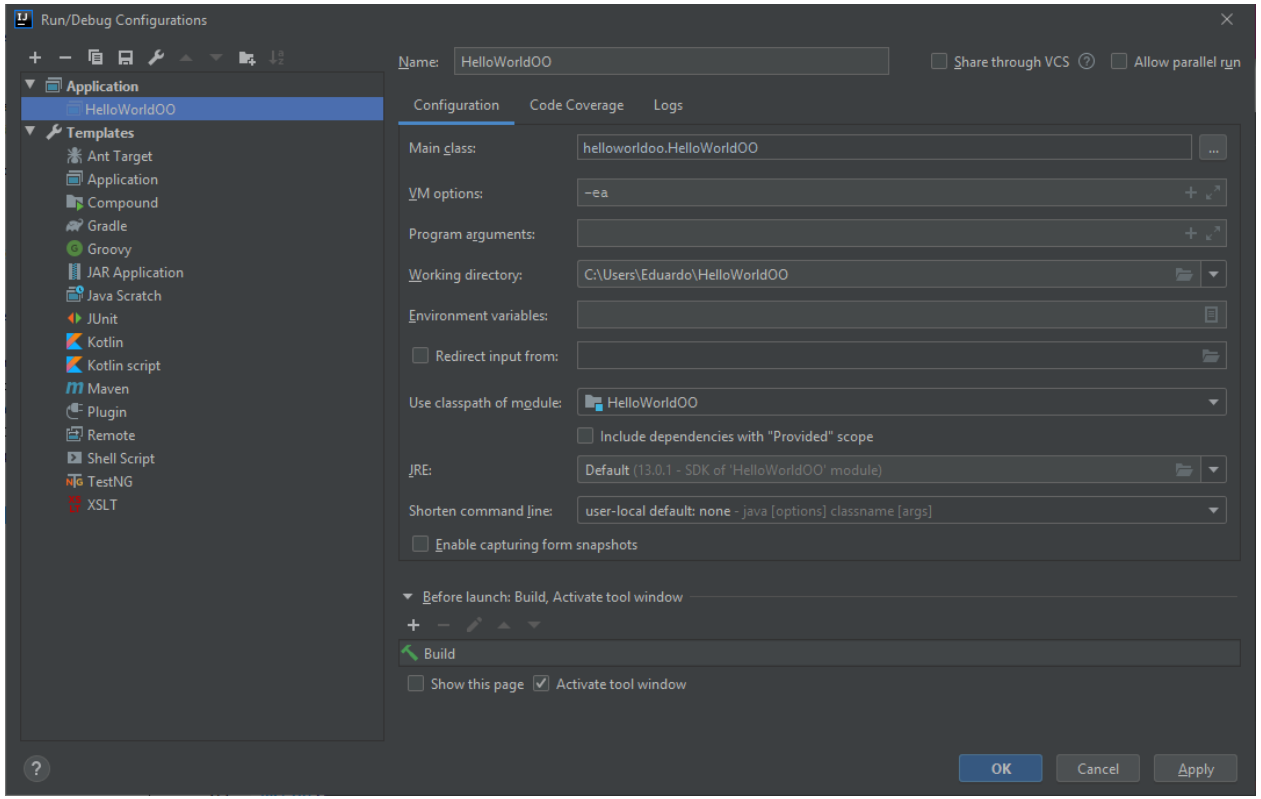
Assertions on IntelliJ IDEA

- On IntelliJ IDEA we can edit the execution configuration to change the compiler's directives.



Assertions on IntelliJ IDEA

- We enable the `-ea` switch on *VM options*.
- We can also modify the *Template Application* to make it applicable to all future configurations.



Assertions vs. exceptions

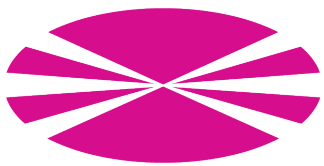
- Exceptions are possible problems that you perhaps cannot address at that point.
- Assertions are things you know (or should be true).
- Even though assertions are useful for testing preconditions, using exceptions is preferable.
  - If a method's precondition is not fulfilled, it is better to throw an exception (because assertions might be disabled).
  - Assertions always throw the same exception – `AssertionError`. Exceptions are more informative, e.g., `IllegalArgumentException`, `EmptyStackException`, `IndexOutOfBoundsException`.

Practice 2: Exceptions in Java

Software Design (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Department of Computer Science and Information Technology
Faculty of Computer Science



UNIVERSIDADE DA CORUÑA