Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Programming 2 with Java

Exceptions

Rolf Haenni – hnr1
Marcel Pfahrer – prm1

Outline - Exceptions

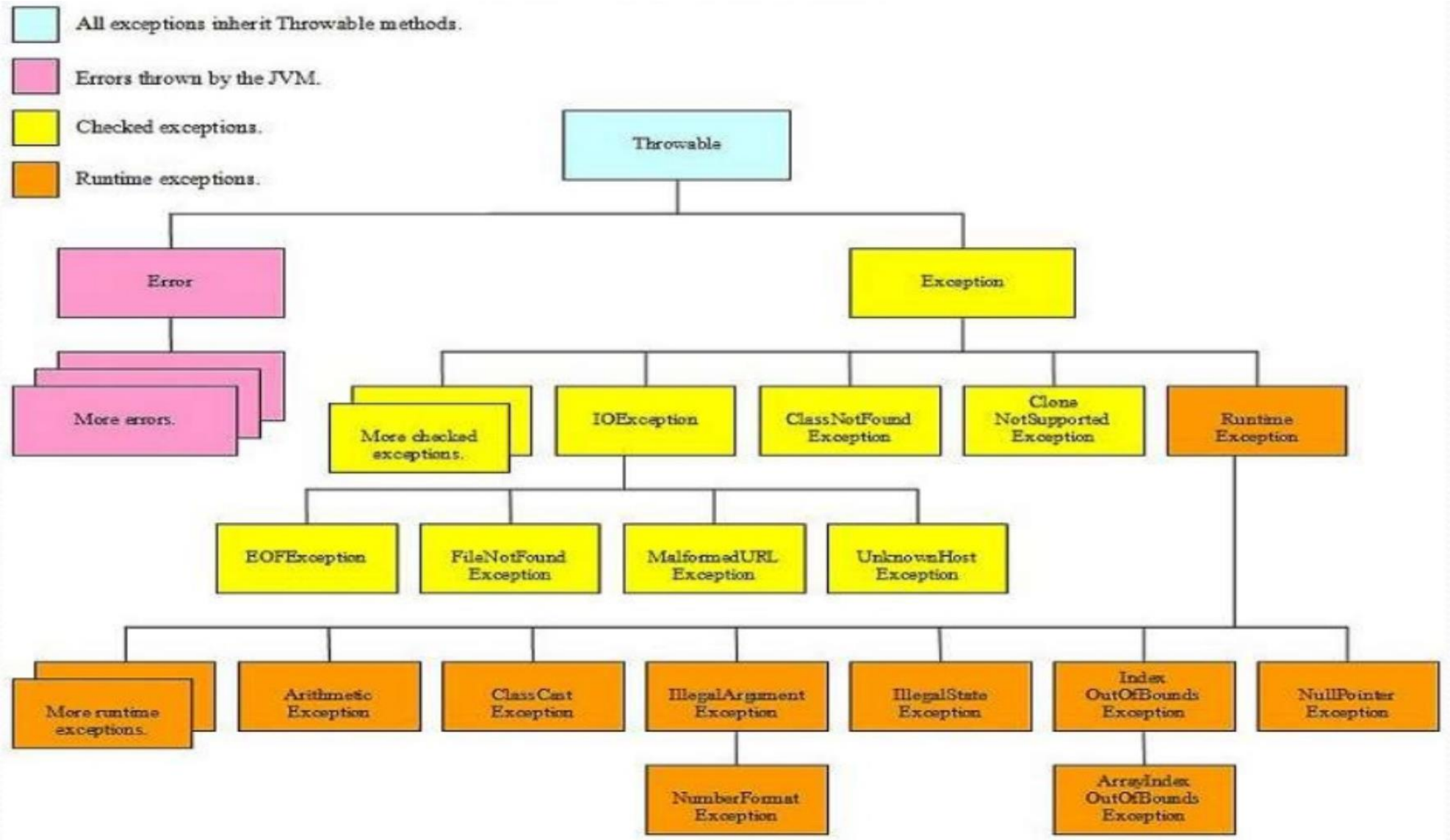
- Repetition of basic exception handling concepts
  - Based on module “Programming 1”
- Exception types
- Handling exceptions
- Finally clause & try-with-resource
- Throwing exceptions
- Custom made exceptions

Exception Types

Checked vs. Unchecked Exceptions

- **Checked exceptions**
  - Events that a well-written application should anticipate and recover from
  - E.g. `java.io.FileNotFoundException` if you try to open a non-existent file
  - Must either be handled locally or declared in a `throws` clause
- **Unchecked exceptions**
  - Errors
    - Events that are external to the application
    - Application cannot anticipate or recover from
    - E.g. hardware failure or system malfunction
  - Runtime exceptions
    - Events that are internal to the application
    - Typically avoidable with well-written code and applications are not expected to recover from
    - E.g. `ArithmeticException`, `NullPointerException`, `IndexOutOfBoundsException`, …

The Java Exception Hierarchy



Source: https://www.slideshare.net/CassianRaja/exception-handling-in-java-63432505

Handling Exceptions

Option 1: `try` / `catch`

- Handle the exception locally with a `try`/`catch` block
- Recover from the exceptional situation
  - Show meaningful error messages to the user
  - Ignore unusable values
  - Apply default values where appropriate
  - …
- Be as specific as possible
  - First catch exceptions from the lower levels in the hierarchy
  - If necessary, catch superclasses with more generic handling code later (or declare them in the `throws` clause
    - An exception already catched before will not be catched again in a catch clause with its superclass

- Hint: avoid empty `catch` blocks

Option 2: `throws`

- If your method does not handle a checked exception, it must be declared in the `throws` clause
- Methods calling another method with a `throws` clause must either
  - Handle the exception, or
  - Declare it in its own `throws` clause
- Methods overriding methods from a superclass or implementing a method from an interface may not declare additional exceptions in their `throws` clause
- Avoid that checked exceptions are never handled in your application
  - The runtime system will handle the exception in a very generic manner
  - The user does not know what happened
  - In many cases, the system admin is never informed about the exception

Option 3: **catch** and rethrow

- It is also possible to handle an exception and then rethrow it
- The rethrown is not handled in other catch clauses on the same level

```
        }
    }
    catch (InputMismatchException ex) {
        System.out.println("This is not a correct number");
        throw ex;
    }
    catch (Exception ex) {
        System.out.println("Something went wrong");
    }
```

Benefits of exception handling

- Separating error-handling code from “regular" code
- Propagating errors up the call Stack to have more context
- Grouping and differentiating error types

- For details see
  http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html

The Finally Clause and try-with-resource

The `finally` Clause

- Sometimes things must be done independent of whether an exception occurs or not
  - E.g. closing an open file, a database connection
  - Doing it in the `try` block and in every `catch` block is a bad idea
- The `finally` clause allows to declare code that is execute independent of whether an exception occurs or not
  - When no exception is thrown
  - When an exception is thrown and handled
  - When an exception is thrown and not handled

- Hints:
  - The code in the finally block does not know whether an exception has been thrown or not
  - And thus, it does not know which statements in the `try` block have been executed
  - The code in the finally block may throw an exception itself

Nesting **try** blocks

- Nesting try blocks and separating the **finally** and the **catch** clauses can solve some problems

```
try {
    Scanner scanner = null;
    try {
        scanner = new Scanner(new File(".\\files\\noFile.txt"));
        System.out.println("Reading number from file");
        int res = scanner.nextInt();
        System.out.println("number is " + res);
    }
    finally {
        if (scanner != null)
            scanner.close();
        System.out.println("Scanner closed");
    }
}
catch (InputMismatchException ex) {
    System.out.println("This is not a correct number");
}
catch (Exception ex) {
    System.out.println("Something went wrong");
}
```

Exercise

- What is the output produced by this code?

```java
public static void BadMethod() {
    throw new RuntimeException();
}
Run | Debug
public static void main(String[] args) {
    try {
        BadMethod();
        System.out.println("A");
    }
    catch (RuntimeException ex) {
        System.out.println("B");
    }
    catch (Exception ex) {
        System.out.println("C");
    }
    finally {
        System.out.println("D");
    }
    System.out.println("E");
}
```

Try-with-resources statements

- Available since Java 7
- The try-with-resources statement is a try statement that declares one or more resources
- A resource is an object that must be closed after the program has finished using it
- The try-with-resources statement ensures that each resource is closed at the end of the statement
  - Independent of whether an exception occurs or not
- Any object that implements `java.lang.AutoCloseable`, which includes all objects implementing `java.io.Closeable` can be used as a resource
- Replaces the finally statement that calls the `close()` method

Try-with-resources statements

- Example PrintWriter …

```
try {
  PrintWriter out =
     new PrintWriter(
      new FileWriter(
        "OutFile.txt"));

   for (int i = 0; i < 100; i++) {
      out.println(i);
   }
   out.close();

} catch (IOException e) {
   e.printStackTrace();
}
```

```
…
try (
   PrintWriter out =
     new PrintWriter(
       new FileWriter(
          "OutFile.txt"))) {

   for (int i = 0; i < 100; i++) {
      out.println(i);
   }
  // out.close()

} catch (IOException e) {
   e.printStackTrace();
}
```

Not needed anymore, but still can throw exceptions

Source: slides from A. Laube

Try-with-resources statements

Example FileReader …

```
try {
  InputStream in =
    new FileInputStream(
     "OutFile.txt");

  try {
    …
    int next = in.read();
    …
  } finally {
    in.close();
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

```
  …
try (
  InputStream in =
    new FileInputStream(
      "OutFile.txt")) {

  // no nested try
  …
  int next = in.read();
  …
  // no finally
  // no close()

} catch (IOException e) {
  e.printStackTrace();
}
```


Source: slides from A. Laube

Throwing Exceptions

Explicitly throw an exception

- Throwing an exception is done with the keyword throw follow by an instance from any class in the exception type hierarchy
  - Typically we create a new instance when throwing it
- Choose a class that best describes the problem occurred
- Exception classes typically have more than one constructor
  - One of them takes a message string as the only parameter

```
public static void method1(int i) throws InputMismatchException {
    if (i < 0)
        throw new InputMismatchException("must not be negative");
    …
}
```

Inner Exception / Cause

- An exception may be caused by another exception
- All exception class may be instantiated with a reference to another one
- The inner exception (or cause) is stored in the outer exception and may be retrieved with the `getCause()` method inherited from **`Throwable`**

```
catch (InputMismatchException ex) {
    System.out.println("This is not a correct number");
    Exception outer = new Exception(ex);
    System.out.println("Inner exception is: " + outer.getCause());
    throw outer;
}
```

Custom made Exceptions

Program your own exception class

- Since exceptions are just normal java classes, they can be inherited
- This allows to program your own exception class
  - Makes exceptions more specific
  - May have additional properties and methods
  - May be used in combination with interfaces to allow different implementations with different exception causes
- Carefully select your superclass
  - Checked or unchecked?
  - More specific exception class?

Exercises

try-with-resource

- Get the class `FinallyGoodPractice` and reprogram it by using the try-with-resource concept

Average Calculator

- Implement a console application that calculates the average of a sequence of integer values
- The sequence is given by objects implementing the interface `INumberSource`
- Implement the exception class `InvalidNumberException` used by the interface
- Implement different data sources based on the interface
  - `RandomNumberSource`
  - `StringNumberSource` (with a string containing comma separated integer values)
  - `FileNumberSource`
- Take care of a good exception handling in your application