Error handling and testing

Lecture 8 of TDA 540
Object-Oriented
Programming

Jesper Cockx

Fall 2018

Chalmers University of Technology — Gothenburg University

Last week: recap

Last week

- Multi-dimensional arrays
- The ArrayList class
- Wrapper classes

This week

- File input and output
- Exceptions and exception handlers
- Testing strategies

File input and output

Ways to communicate input and output

- Command line (`System.in` & `System.out`)
- Graphical user interfaces (`JOptionPane`)
- Reading and writing files
- Transmitting data over the network
- Getting input/output from another program

Ways to communicate input and output

- Command line (`System.in` & `System.out`)
- Graphical user interfaces (`JOptionPane`)
- Reading and writing files
- Transmitting data over the network
- Getting input/output from another program

The `File` class

An object of class `File` represents a file on the computer's disk (text, image, sound, video, program, ...).

Opening a file:

```
File myFile =
  new File("important_stuff.txt");
```

Reading files

To read text from a file, we combine `File` and `Scanner`:

```
public static ArrayList<Integer>
  readNumbers(String fileName)
  throws FileNotFoundException {
    File myFile = new File("data.txt");
    Scanner input = new Scanner(myFile);
    ArrayList<Integer> numbers =
      new ArrayList<Integer>();
    while (input.hasNextInt()) {
      numbers.add(input.nextInt());
    }
    input.close();
    return numbers;
}
```

Writing files

To write to a text file, we use the class `PrintWriter`:

```
PrintWriter writer =
  new PrintWriter("secret.txt");
writer.println("Secret: ****");
```

`PrintWriter` supports all methods of `System.out`: `print`, `println`, `printf`, ...

PrintWriter example

```
public static void
  makeRandomFile(String filename)
  throws FileNotFoundException {
    PrintWriter writer = new PrintWriter(filename);
    for (int i = 0; i < 10000; i++) {
      int x = (int) Math.random() * 100;
      writer.println(x);
    }
    writer.close();
}
```

Putting it all together

Demo code: read out a list of integers from a file and print the sorted values.

Exceptions

Exceptions

Whenever something unexpected happens while running a program, Java will raise an exception.

If the exception is not handled, the program will crash and print the exception.

Two kinds of exceptions

Unchecked exceptions: some error in the program

- Array index out of bounds
- Division by zero
- Null pointer
- ...

Checked exception: a problem beyond the program

- File not found
- Network disconnected
- ...

Some common exceptions in Java

- `NullPointerException`
- `IndexOutOfBoundsException`
- `InputMismatchException`
- `NoSuchElementException`
- `ArithmeticException`
- `NumberFormatException`
- `IllegalArgumentException`
- `FileNotFoundException` (checked)

Propagating exceptions

Checked exceptions must be mentioned in the method signature:

```
public static void makeFile()
  throws FileNotFoundException {
    PrintWriter writer =
      new PrintWriter("...");
    writer.println("...");
    writer.close();
  }
```

Exception handling

Catching exceptions

You can catch exceptions with `try` and `catch`:

```
public static void makeFile() {
  try {
    PrintWriter writer =
      new PrintWriter("...");
    writer.println("...");
    writer.close();
  } catch (FileNotFoundException e) {
    System.out.println("Sorry!");
  }
}
```

The `Exception` class

Exceptions are objects of a class `Exception`.

You can get the exception message with the `getMessage()` method:

```
try {
  ...
} catch (FileNotFoundException e) {
  String message = e.getMessage();
  System.out.println(message);
}
```

Example: robust user input

```
boolean done = false;
while (!done) {
  String indata =
    JOptionPane.showInputDialog("Input an integer:");
  try {
    int number = Integer.parseInt(indata);
    int res = number * number;
    JOptionPane.showMessageDialog(null,
      "The square is " + res);
    done = true;
  } catch (NumberFormatException e) {
    JOptionPane.showMessageDialog(null,
      "Invalid integer. Try again!");
  }
```

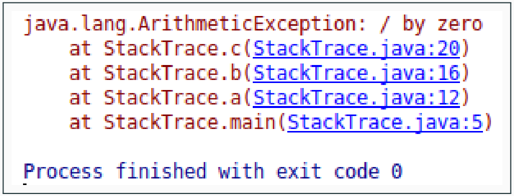
Stack traces

A stack trace lists all methods that lead to the point in the program where an exception was thrown.

You can print the stack trace with the method `printStackTrace()`.

PrintStackTrace example

```
public static void main(String[] args) {
    try {
        a();
    } catch (ArithmeticException e) {
        e.printStackTrace();
    }
}
static void a() {
    b();
}
static void b() {
    c();
}
static void c() {
    int i = 1/0;
}
```



```
java.lang.ArithmeticException: / by zero
	at StackTrace.c(StackTrace.java:20)
	at StackTrace.b(StackTrace.java:16)
	at StackTrace.a(StackTrace.java:12)
	at StackTrace.main(StackTrace.java:5)

Process finished with exit code 0
```

Catching all exceptions

It is possible to catch all exceptions:

```
try {
  ...
} catch (Exception e) {
  ...
}
```

This throws away all error messages,
so fixing bugs becomes very difficult
⇒ don’t do this!

The `finally` block

Code in a `finally` block is executed no matter whether there was an exception or not.

Example: make sure file is always closed

```
PrintWriter writer;
try {
  writer = new PrintWriter("secret.txt");
  writer.println("Password: ****");
} finally {
  writer.close();
}
```

Throwing your own exceptions

Throwing your own exceptions

You can throw exceptions in your own code:

```java
public void withdraw(int amount) {
  if (amount < balance) {
    balance = balance - amount;
  } else {
    throw new IllegalArgumentException
      ("Not enough money!");
  }
}
```

Throwing your own exceptions

You can throw exceptions in your own code:

```
  public void withdraw(int amount)
    throws IllegalArgumentException {
  if (amount < balance) {
    balance = balance - amount;
  } else {
    throw new IllegalArgumentException
      ("Not enough money!");
  }
}
```

Optionally, you can declare the exception in the method signature (required for checked exceptions).

15 min. break

Kahoot! Exceptions in Java

Testing

Reminder: compile-time vs run-time errors

Compile-time errors (aka static errors)

- Syntax errors
- Variable scoping errors
- Type errors
- Missing return statements
- ...

Run-time errors (aka dynamic errors)

- Program crashes
- Uncaught exceptions
- Functional/logical errors
- ...

What counts as a compile-time or run-time error depends on the programming language!

Testing

To see if your program works correctly, you need to test it.

To test effectively, you need to know what the program is supposed to do:
you need a specification.

Modular design helps with testing: you can test each component individually.

Unit testing vs system testing

Unit testing: test functionality of individual components (methods and classes)

System testing: test overall functionality of the whole program

Both kinds of testing are necessary!

Test early, test often

The longer a bug goes undiscovered, the more work it takes to fix it!

$$\begin{array}{l}\boxed{\text{Specification}}\\ \Rightarrow \boxed{\text{Design}}\\ \Rightarrow \boxed{\text{Implementation}}\\ \Rightarrow \boxed{\text{Validation}}\\ \Rightarrow \boxed{\text{Production}}\end{array}$$

Rule of thumb: an bug not fixed in one phase takes 10x more time to fix in the next phase

The limits of testing

Testing can only reveal the presence of bugs, never their absence.

Testing strategies

```java
import javax.swing.*;
public class Postage {
  public static void main(String[] args) {
    String input = JOptionPane.showInputDialog("Weight:");
    double weight = Double.parseDouble(input);
    String output;
    if (weight <= 0.0)
      output = "Weight must be positive!";
    else if (weight <= 20.0)
      output = "Postage is 5.50 kronor.";
    else if (weight <= 100.0)
      output = "Postage is 11.00 kronor.";
    else if (weight <= 250.0)
      output = "Postage is 22.00 kronor.";
    else if (weight <= 500.0)
      output = "Postage is 33.00 kronor.";
    else
      output = "Too heavy: use a packet.";
    JOptionPane.showMessageDialog(null, output);
  }
}
```

Question: how to test this program?

Black-box vs white-box testing

Black-box testing: test a program by looking at its *specification*.
$\Rightarrow$ you don't have to know the implementation

White-box testing: test a program by looking at its *implementation*.
$\Rightarrow$ you can explore all possible code paths

Some strategies for writing tests

- Partition testing: Divide inputs in classes and choose (at least) one ‘typical example’ from each class
    - According to the program logic (black-box)
    - According to the program structure (white-box)
- Boundary value testing: Test inputs at the boundary between classes
- Randomized testing: Test the program on randomly generated input

What's next?

Next lecture (in two weeks):
Recap & FAQ of part 1.

To do:

- Read the book:
    - Today: chapter 7
    - Next lecture: chapters 1-7
- Hand in the fourth lab assignment