Reasoning about program correctness

Lecture 13 of TDA 540
Object-Oriented
Programming

Jesper Cockx

Fall 2018

Chalmers University of Technology — Gothenburg University

Last week: GUIs and event-driven programming

Last week

- Programming graphical interfaces with Swing
  - Top-level: `JFrame`
  - Second level: `JPanel`
  - Atomic components: `JLabel`, `JButton`, `JTextField`, ...
- Event-driven programming
  - Event handlers implement the `ActionListener` interface
  - Event publishers offer a method to subscribe to events

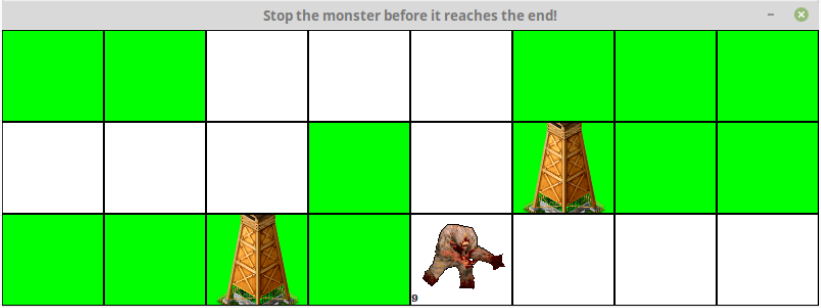
The `Timer` class

```
class TickTock implements ActionListener {
  boolean tick = true;
  public void actionPerformed(ActionEvent e) {
    if (tick)
      System.out.println("tick");
    else
      System.out.println("tock");
    tick = !tick;
  }

Timer timer = new Timer(1000, new TickTock());
timer.start();
```

Lab 8: Tower Defence game

Goal: implement a simple tower defence game:



Assignment is now on the course webpage.

Kahoot: GUI’s and event-driven programming

Program correctness

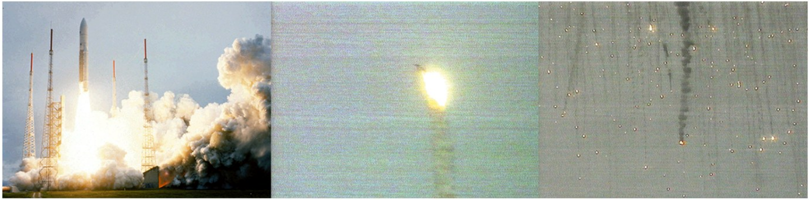
Central question

How can we know a program works correctly?

Even more basic question

Why do we care that
a program works correctly?

The Ariane rocket incident



Unnumbered 7 p347

On June 4, 1996 the Ariane 5 rocket exploded because of an overflow, shutting off the onboard computer.

More reasons to care about program correctness

- Online banking
- Medical equipment
- Self-driving cars
- ...

Program correctness

What does it mean for a program to be correct?

- A program that does not crash?
- A program that passes all the tests?
- A program that matches its specification!

Specification vs implementation

Correctness of a program is always relative to a specification.

The specification is a description of what the program should do (usually implicitly includes “do not explode”).

The implementation (i.e. the actual code) usually contains many more details than the specification.

Specification vs implementation

Specification:

method `sum` takes a non-null reference `a` to an array of integers, and returns the sum of all values in `a`

Implementation:

```
int sum(int[] a) {
  int sum = 0;
  for (int v : a) {
    sum += v;
  }
  return sum;
}
```

Specification: natural language vs symbols

Specification in natural language:

The program returns the first prime number greater than the given number.

Specification in symbolic language:

$$\texttt{r = nextPrime(n)} \Rightarrow$$
$$(\texttt{r > n \&\& isPrime(r) \&\&}$$
$$\forall\, \texttt{m}: (\texttt{m > n \&\& isPrime(m)}) \Rightarrow \texttt{r} \leq \texttt{m})$$

Which is better depends on the situation!

Pre- and post-conditions

Pre- and post-conditions

Pre- and post-conditions are an important part of the specification for a method:

- A precondition = a property that should hold of the method's inputs before it is called
  $\Rightarrow$ responsibility of the one calling the method
- A postcondition = a property that should hold of the method's output after it is done
  $\Rightarrow$ responsibility of the body of the method

Method specifications

Specification:

1. precondition:

   `a != null`

2. postcondition:

   $\texttt{sum}{=}{=}\sum_{0 \leq k < \texttt{a.length}} \texttt{a}[k]$

Implementation:

```
int sum(int[] a) {
  int sum = 0;
  for (int v : a)
    sum += v;
  return sum;
}
```

Pre- and post-conditions in object-oriented programs

In object-oriented programs, the input and output of a method also include the object state before and after executing the method.

Specification:

1. precondition:
   `amount >= 0`
2. postcondition:
   new `volume` =
   old `volume + x`

Implementation:

```
class WaterGlass {
  int volume;

  void addWater(double x) {
    volume += x;
  }
}
```

Pre- and post-conditions in Java

JML is a system for annotating Java programs:

- `@requires` *precondition*
- `@ensures` *postcondition*

```
class WaterGlass {
  double vol; double max;

  // @requires vol + amount <= max;
  // @ensures  vol == \old(vol) + amount;
  void addWater(double amount) {
    volume += amount;
  }
}
```

Invariants

An invariant is a property of the program's state that stays true throughout the execution of the program.

Example: *"The current volume of a glass is between 0 and the maximum volume."*

You can think of an invariant as both a pre- and a post-condition on each method.

Invariants in JML

Invariants are annotated with `@invariant`:

```
class WaterGlass {
  // @invariant 0 <= vol && vol <= max;
  double vol;

  // @invariant max >= 0;
  double max;

  // ...
}
```

Handling invalid inputs

What should you do when an input does not satisfy the preconditions?

- Lazy approach: don't check for invalid inputs
- Self-confident approach: return a default value on invalid inputs
- Pedantic approach: use assertions to check for invalid inputs

Lazy approach

```java
public class WaterGlass {
  // ...
  public void removeWater(double amount) {
    this.volume -= amount;
  }
}
Glass glass = new WaterGlass(100);
glass.removeWater(200); // volume is now negative!
```

+ Takes no extra effort
- You have to be very careful when calling a method

Self-confident approach

```
public class WaterGlass {
  // ...
  public void removeWater(double amount) {
    volume -= amount;
    if (volume < 0) volume = 0;
  }
}
Glass glass = new WaterGlass(100);
glass.removeWater(200); // volume is now 0
```

+ Program doesn't crash
- It's very hard to tell when something goes wrong

Pedantic approach

```java
public class WaterGlass {
  // ...
  public void removeWater(double amount) {
    assert (amount <= volume);
    volume -= amount;
  }
}
Glass glass = new WaterGlass(100);
glass.removeWater(200); // raises assertion error
```

+ You know immediately when something goes wrong
- Program crashes even when error wouldn't matter

Reminder: assertions in Java

Assertions are Java's built-in way to express pre- and post-conditions in a program.

```
assert condition;
```

1. if `condition == true`, execution continues (the assertion passes: no effects)
2. if `condition == false`, an exception `AssertionError` is thrown (the assertion fails)

Important: assertion checking is disabled by default. To enable it run your program with `java -ea MyProgram`.

Assertions vs. Exceptions

Exceptions signal exceptional but possible behaviour

Assertions signal program states that should be impossible

- in a correct program, assertions always evaluate to true (and thus have no effect)
- an assertion evaluating to false indicates that there is a mismatch between specification and implementation (probably a bug)

Assertions are not enabled by default, so in practice Java programmers often instead use exceptions.

Programming principle: Fail fast!

It's usually better to fail fast rather than continue with wrong inputs:

- Easier to find the precise location of the error
- Easier to handle the problem 'one level up'
- Safer to stop the program rather than perform some possibly irreversible operation (e.g. overwriting a file)

More about exceptions

Exceptional behavior

Exceptions signal exceptional but possible behaviour:

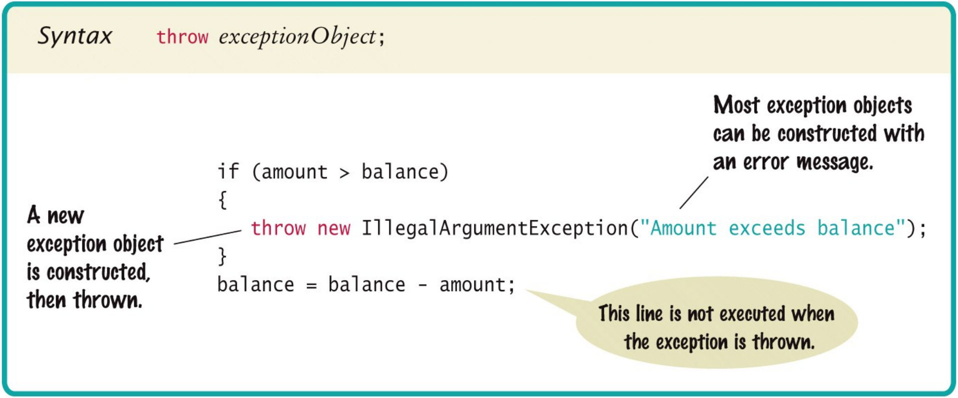
- the user provides invalid input
- the program runs out of memory
- a network connection cannot be established because a website is down
- ...

Programming with exceptions

Programs with exception-handling have two control flows:

1. normal control flow: no exception occurs, exception-handling code is not executed
2. exceptional control flow: exceptions occur, exception-handling code is executed

Throwing exceptions



Syntax 7.1

Exception example

```java
// parse nonnegative integer string
int stringToInt(String str) {
  int result;
  if (str == null) throw new NullPointerException();
  for (int i = 0; i < str.length(); i++) {
    if (!Character.isDigit(str.charAt(i)))
      throw new NumberFormatException(
        str + " is not an integer!");
  } // ... normal behavior ...
  return result;
}
```

Catching exceptions

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage);
}
```

This constructor can throw a FileNotFoundException.

This is the exception that was thrown.

When an IOException is thrown, execution resumes here.

A FileNotFoundException is a special case of an IOException.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

Finally blocks

Syntax

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

This variable must be declared outside the `try` block so that the `finally` clause can access it.

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

This code may throw exceptions.

This code is always executed, even if an exception occurs.

Declaring exceptions

Syntax *modifiers returnType methodName(parameterType parameterName, . . .)*
  throws *ExceptionClass, ExceptionClass, . . .*

```
public static String readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

You **must** specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

Syntax 7.3

Exception objects

Exceptions are represented by exception objects, which are instances of exception classes

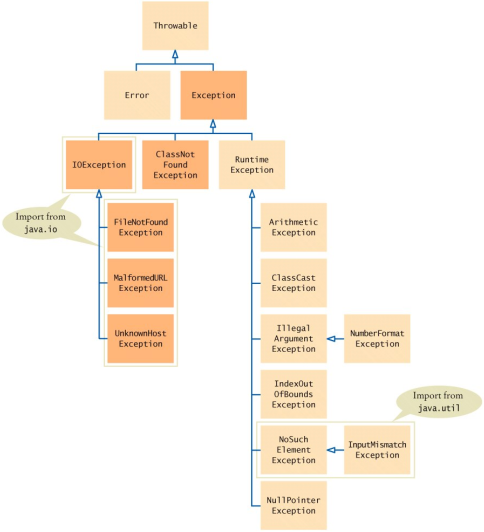
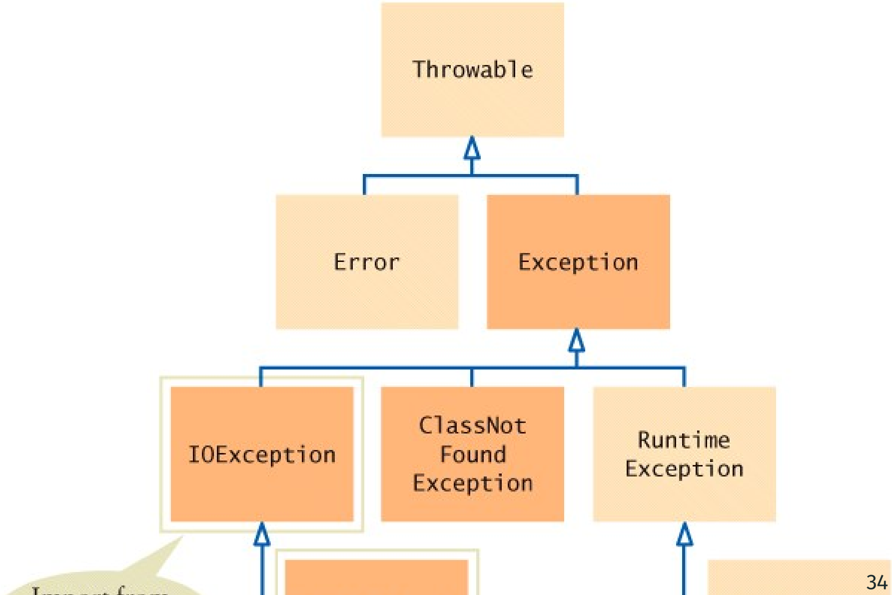


Figure 7.2

The exception hierarchy



Catching exceptions

```
try {
  // ...
} catch (ET e) {
  /* handler code */
}
```

This will catch all exceptions of type `ET` or subtypes of `ET`

Multi-catch blocks

```
try {
  // ...
} catch (ET1 | ET2 | ET3 e) {
  /* handler code */
}
```

This will handle exceptions whose type is a subtype of `ET1`, of `ET2`, or of `ET3`.

`ET1`, `ET2`, and `ET3` must not be related by inheritance.

Declaring a new exception class

You can create your own exceptions by creating a new subclass of `Throwable`:

```
public class NotEnoughCake extends Throwable {
    int missingCakes;

    public NotEnoughCake(int missingCakes) {
        this.missingCakes = missingCakes;
    }
}
```

Throwing your own exceptions

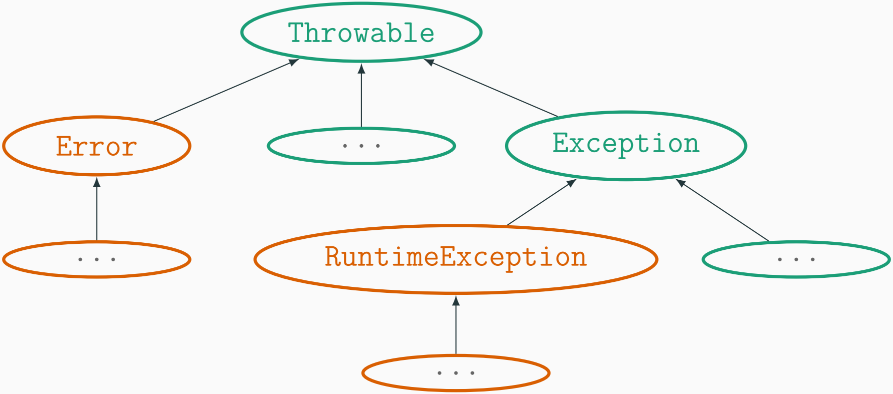
```
private void throwParty()
  throws NotEnoughCake, NotYourBirthday {
    if (!getBirthday().equals(today()))
      throw new NotYourBirthday();
    if (!hasEnoughCake()) {
      int missingCakes = nbOfPeople/2 - nbOfCakes;
      throw new NotEnoughCake(missingCakes);
    }
    eatSomeCake();
    System.out.println("This is an awesome party!");
    eatSomeCake(); // om nom
}
```

Catching custom exceptions

```java
public static void main(String[] args) {
    BirthdayParty party = new BirthdayParty(20);
    try {
      party.throwParty();
    } catch (NotEnoughCake e) {
      System.out.println(
        "We are missing" +
        e.missingCakes + "cakes :(");
    } catch (NotYourBirthday e) {
      System.out.println(
        "It's not your birthday");
    }
}
```

Checked vs. unchecked exceptions

Java exception classes are partitioned in checked and unchecked



Checked vs. unchecked exceptions

Java exceptions are either checked or unchecked

CHECKED EXCEPTIONS	UNCHECKED EXCEPTIONS
must be declared in method signatures with `throws`	**may** or may not be declared
must be handled or propagated	**may** or may not be handled
compiler checks all exceptions are handled	uncaught exceptions may **crash** the program

Checked exception example

When calling a method that may throw a checked exception, you must either declare the exception:

```
void tryToOpenFile(String filename)
throws FileNotFoundException {
  FileReader fr = new FileReader(filename);
}
```

...or handle it:

```
void tryToOpenFile(String filename) {
  try {
    FileReader fr = new FileReader(filename);
  } catch (FileNotFoundException e) {
    System.out.println("Fail!");
  }
}
```

Exceptions: checked or unchecked?

Advantages of checked exceptions:

- behaviour is explicit in method signature, so clients know what exceptions to handle
- no uncaught exceptions at the top-level, so program cannot crash on checked exception

Advantages of unchecked exceptions:

- don’t need exception handlers everywhere
- no need to change public interface of methods

Exceptions: checked or unchecked?

How to choose in practice between checked and unchecked exceptions?

- use a checked exception if the client can do something to recover from the exception
- document the usage of unchecked exceptions too
- usually prefer checked exceptions to error codes

Program verification

Verification

Verification is the process of checking that a program is correct.
$\Rightarrow$ we need a specification before we can do verification

Three main techniques to do verification:

- code inspection: *look at* the code and try to see if it does the correct thing.
- testing: *run* the program on different inputs and check that every run satisfies the specification
- formal verification: mathematically prove that every possible execution of the program satisfies the specification

Code inspection

Just looking at your code is often the first thing to do when trying to find an error.

Often someone who did not write the code can more easily spot errors (pair programming).

Code inspection is never a replacement for proper testing!

Unit testing vs system testing

Two main kinds of 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!

Unit testing example

Method `addWater` under test:

```
class WaterGlass {
  double vol;
  double max;

  WaterGlass (double max) {
    this.max = max;
    this.vol = 0;

  void addWater(double x) {
    vol += x;
  }
}
```

Testing code:

```
WaterGlass g =
  new Waterglass(250);
g.addWater(0);
assert g.volume() == 0;
g.addWater(121);
assert g.volume() == 121;
g.addWater(3);
assert g.volume() == 121 + 3;
```

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
- Regression testing: Whenever you fix an error, add a test to make sure it stays fixed!

Thou shall test your code!

Systematically testing your code is a good practice that every programmer should follow.

- Test extensively: write unit tests for all `public` methods
- Test early: start writing tests as soon as a class has a public interface
- Test often: rerun the tests each time you make a change

Big projects can often have 2-3x *more* tests than actual code!

Formal verification

Formal verification = mathematically *proving* that a program is implemented correctly, often with the help of a computer.

- Model checking: systematically explore all possible program states, using some model of the program.
- Theorem proving: write down a detailed proof that the program works correctly, and let the computer check each step of the proof.

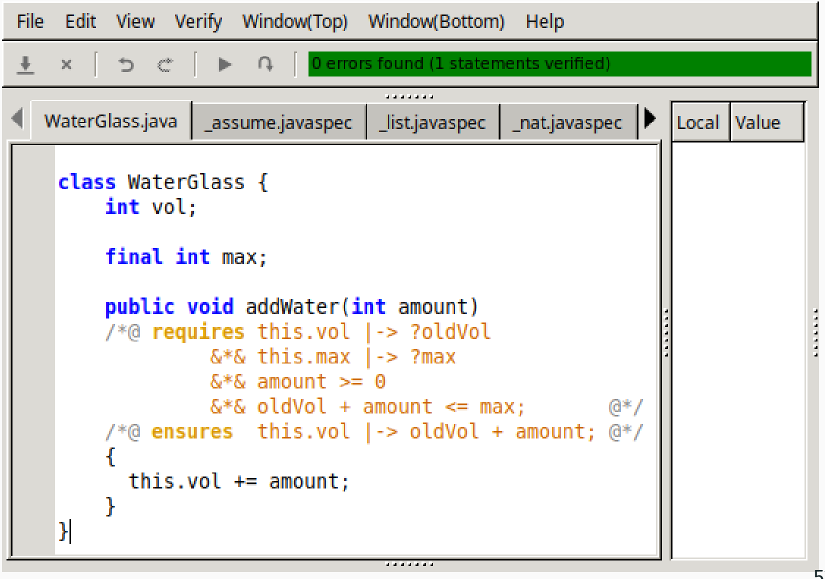
Formal verification often takes a lot of effort but it is the only way to guarantee that the program is implemented correctly.

Program verification in Java

Java has no built-in support for program verification.

Instead, we must rely on external tools to verify Java programs.

Program verification using VeriFast



Program verification in Agda

Other languages like Agda have program verification built-in:

```
record WaterGlass : Set where
  constructor newWaterGlass
  field
    current     : Nat
    maximum     : Nat
    {{invariant}} : current ≤ maximum
```

```
addWater : (g : WaterGlass) (amount : Nat)
         → {{requires : g .current + amount ≤ g .maximum}}
         → WaterGlass
addWater g amount =
  newWaterGlass (g .current + amount) (g .maximum)

addWater-correct : {{requires : g .current + amount ≤ g .maximum}}
                 → addWater g amount .current ≡ g .current + amount
addWater-correct = refl
```

What's next?

Next (and final!) lecture:
Software design & recursion.

To do:

- Read the book:
  - Today: parts of chapter 7
  - Next lecture: parts of chapers 12 & 13[1]
- Start working on the final two labs

[1] Online chapters available at
http://bcs.wiley.com/he-bcs/Books?action=resource&bcsId=6907&itemId=1118063317&resourceId=27347