2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- Robustness (as a way of dealing with defects)
- Exceptions*

*Reused and adapted some slide material from Alexandre Denault

Warning

- This course focuses on techniques to develop *larger* programs that must be (re)usable and maintainable by others.
- Examples and exercises illustrate it by *small* program fragments.
- Therefore, the techniques may seem to be *overkill*.

Design By Contract (DBC), Test-Driven Development (TDD)

1. First, design the **contract**.

   For instance, for a module in the context of Divide & Conquer.

2. Next, design and implement **test cases**, serving as critical client.

   These also serve as operational documentation of requirements.

3. Finally, design and implement a service/solution **provider**.

Contract-Driven Development; Contract-First Design (uncommon names)

Example Code with a (Formal) Contract: `gcd`

```
 1 /** Returns the greatest common divisor of a and b.
 2
 3  *  @pre     0 < a  &&  0 < b
 4  *  @return  \old(\max c; c divides a && c divides b; c)
 5  */
 6 public static long gcd (long a, long b) {
 7
 8     while (a != b) {
 9         if (a > b) {
10             a = a - b;
11         } else { // b > a
12             b = b - a;
13         }
14     }
15
16     return a;
17 }
```

Observations about gcd

?

Observations about `gcd`

- Precondition is not just **`true`**: client code has an obligation

  (A precondition **`true`** could be omitted)

- It modifies no external variables

  (Implied by missing `@modifies`)

- Returned value is a function of the initial (`\old`) parameter values

- Method body (implementation) modifies the parameters

  (Otherwise, it would be good to declare them **`final`**; see FAQ)

- Method body consists of non-trivial **`while`** loop

  (Why correct?)

Intermezzo: Why is implementation of `gcd` correct?

Defininitions:

$$\begin{aligned} divisors(a) &= \{\, d \mid (\exists\, k :: a = kd) \,\} \\ cd(a,b) &= divisors(a) \cap divisors(b) \quad \text{(common divisors)} \\ \gcd(a,b) &= \uparrow cd(a,b) \quad \text{(greatest common divisor)} \end{aligned}$$

Properties:

$$\begin{aligned} cd(a,b) &= cd(b,a) \\ \gcd(a,b) &= \gcd(b,a) \end{aligned}$$

$$\begin{aligned} cd(a,b) &= divisors(a) \quad \text{if } a = b \\ \uparrow divisors(a) &= a \\ \gcd(a,b) &= a \quad \text{if } a = b \end{aligned}$$

$$\begin{aligned} cd(a,b) &= cd(a-b,b) \quad \text{if } a > b \\ \gcd(a,b) &= \gcd(a-b,b) \quad \text{if } a > b \end{aligned}$$

Intermezzo: Why is implementation of `gcd` correct?

```
1  /** Returns the greatest common divisor of a and b.
2   *  @pre     0 < a && 0 < b
3   *  @return  \old(\max c; c divides a && c divides b; c)
4   */
5  public static long gcd (long a, long b) {
6      // inv I0: 0 < a <= \old(a)  &&  0 < b <= \old(b)
7      // inv I1: gcd(a, b) == gcd(\old(a), \old(b))
8      // bound: a + b  (for termination)
9      while (a != b) {
10         if (a > b) { // gcd(a, b) == gcd(a - b, b)
11             a = a - b;
12         } else { // b > a: gcd(a, b) == gcd(a, b - a)
13             b = b - a;
14         }
15     }
16     // a == b, hence gcd(\old(a), \old(b)) == gcd(a, b) == a
17     return a;
18 }
```

Intermezzo: Efficiency of `gcd` Implementation

?

Intermezzo: Efficiency of `gcd` Implementation

- This implementation of `gcd` is not so efficient: $O(a + b)$
- Can be improved (why?) by using

```
1          while (a != 0 && b != 0) {
2              if (a > b) {
3                  a = a % b;
4              } else { // b > a
5                  b = b % a;
6              }
7          }
8          // a == 0 || b == 0
9
10         return a + b;
```

How to Deal with Violated Precondition?

- What if `gcd` is called, when its precondition is violated?
- What happens if the client does call `gcd` with $a = 0$ or $b = 0$?

  And what if $a < 0$ or $b < 0$?

'Solution' #1 (to violated precondition): Do Nothing Special

- *Partial function*: "It is up to the client to call `gcd` correctly."
- Partial functions can lead to programs that are *not robust*.

  Anything could happen when a function is invoked outside its precondition.
- A robust program continues to behave reasonably in the presence of defects:
  - At best: provide an approximation of defect-free behavior. Also known as graceful degradation.
  - At worst: halt with a meaningful error message without causing damage.

Solution #2: Return Special Result

- Return 0: 0 cannot be the result of a correctly called `gcd`; so, it can be used as a special result.
- What happens if you return 0 as the result of `gcd`?
  - Now the caller must check for the special result. This is inconvenient.
- Sometimes the whole range of return values is legal, so there is no *special* result to return.
- For example, the `get` method in a `List` returns the value of the list's i-th element (either **`null`** or an `Object`).
  - There are no values to convey that the index is out of bounds.

Solution #3: Use an Exception

- An Exception signals that something *unusual* occurred.
- Exceptions *bypass the normal control flow*:
  - `A()` calls `B()` calls `C()` calls `D()`
  - `D` 'throws' an exception
  - `A` catches the exception
  - All the remaining code in `D`, `C`, and `B` is skipped
- Exceptions cannot be ignored;
  the program terminates if an exception is not caught anywhere.
- The use of exceptions is supported by Java `Exception` types.

Robustness

- **Robustness** of methods (functions, procedures) concerns their behavior when precondition is not satisfied

  A method is called **robust** when its behavior under a violated precondition is well-specified in its contract

- Method `gcd` is **not robust**, because it misbehaves when the precondition is not satisfied

- Method `gcd` can be made **robust** by having it check the precondition, and signaling a violation

Advantages of Robustness

?

Advantages of Robustness

In the end, it is about where to put the blame . . .

Without an explicit signal that a precondition was violated,

> it may appear that the problem is located **inside** the method that seemingly misbehaves

With an explicit signal that a precondition was violated,

> it is clear that the problem is located **outside** the method (it does not misbehave)

Disdvantages of Robustness

?

Disdvantages of Robustness

There is a trade-off.

- Overhead in writing the precondition checks, and in testing them.
- More code to read and understand, during maintenance/evolution.
- Runtime overhead in checking the precondition, even if satisfied.
- Runtime overhead in throwing, propagating, catching exceptions.

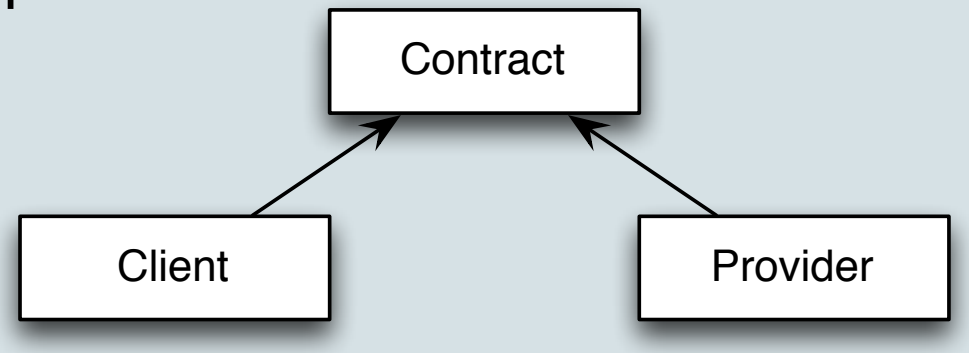
Exceptions in Java and Other Languages

- The exception mechanism in Java has some peculiarities.
- Other programming languages may have an exception mechanism, but it probably differs from Java in the details.
- We present an approach that can also be used in other languages, but we cannot avoid some Java details.

Exceptions in Java: Roadmap

`Exception`
class type

javadoc: `@throws ...Exception if ...`
header: `... m() throws ...Exception`



call:

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

body:

```
if (condition) {
    throw new ...Exception(...);
}
```

Client: (JUnit) test cases; production code

How to Put Exceptions in Contracts

Syntactic part of specificying exceptions The method header lists exceptions that are part of specified behavior in a throws clause:

```
public static long gcd (long a, long b) throws IllegalArgumentException
```

More than one exception can be thrown:

```
public static int find (int x, int [ ] a)
        throws NullPointerException, NotFoundException
```

Semantic part of specifying exceptions The javadoc @throws tag states the conditions for exceptions; the postcondition specifies the resulting state *when no exception was thrown*:

```
/** @throws NullPointerException  if a == null
 *  @throws NotFoundException  if x does not occur in a
 *  @pre a != null && x occurs in a
 *  @post 0 <= \result < a.length && a[\result] == x
 */
```

Contracts That Involve Exceptions

Termination of a method by throwing an exception is ok, but it must be in the contract (and be aware of a performance penalty).

A method that *always throws an exception when its precondition is violated* is said to be robust.

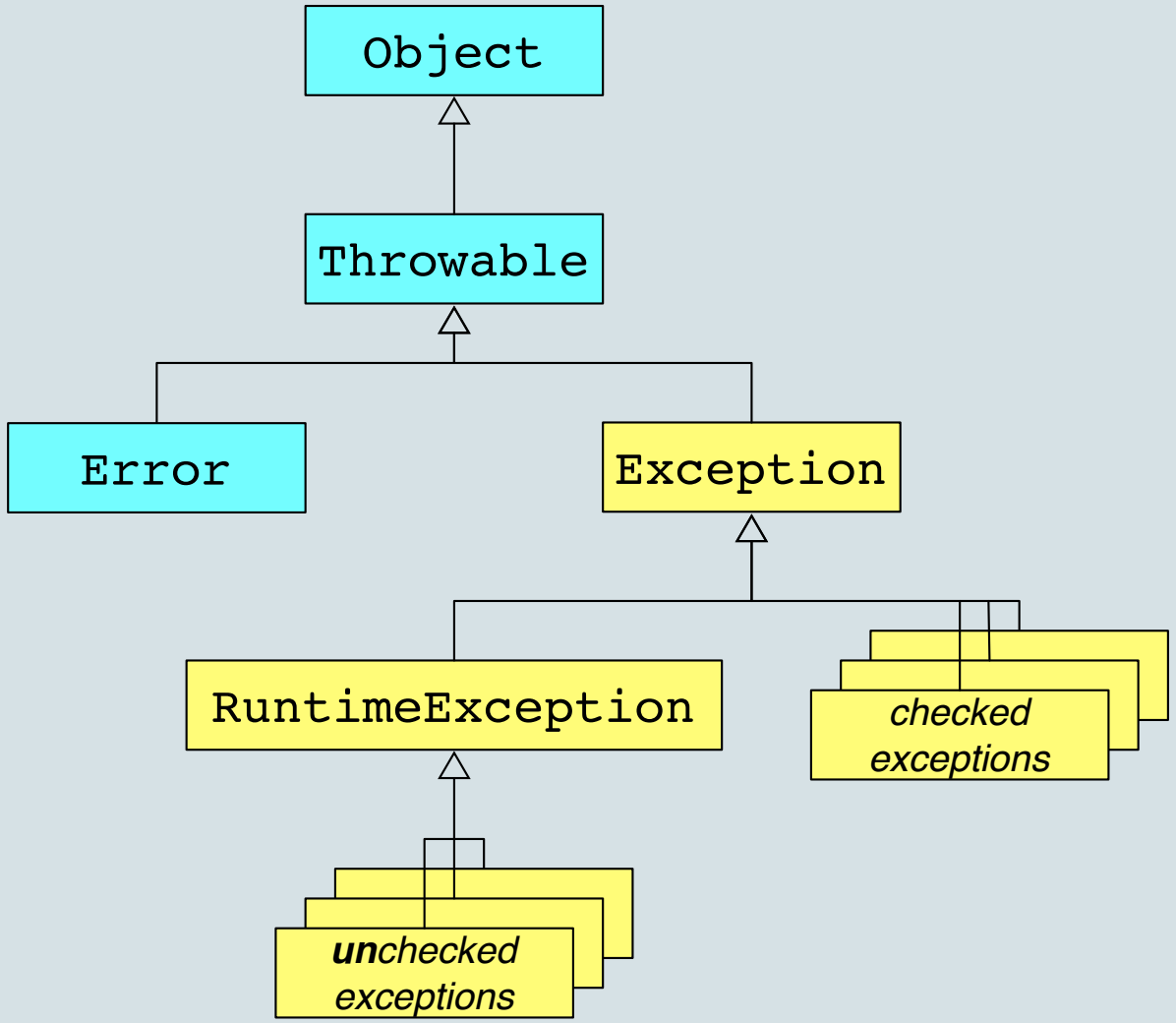
When a method has side effects, its contract should clarify how side effects interact with exceptions.

The `@modifies` clause only lists inputs that *can* be modified, but it does not say under what conditions.

The postcondition should explicitly specify under what circumstances modifications occur and when not.

Exception Type Hierarchy in Package `java.lang`

In Java, Exceptions are objects, existing in the object hierarchy:



Constructing Exception Objects

- A simple exception; only its type conveys information (avoid this):

```
Exception e1 = new MyException();
```

- An exception with some extra info in a string (recommended):

```
Exception e2 = new MyException(
    "where and why this exception occurred");
```

- Or even passing an object (not needed in 2IPC0):

```
Exception e3 = new MyException(
    "where and why this exception occurred",
    someObjectWhichCausedTheException);
```

Checking Preconditions and Throwing Exceptions

```
public static long gcd (long a, long b) throws IllegalArgumentException {
   if (a <= 0) {
      throw new IllegalArgumentException(
         "Num.gcd.pre violated: a == " + a + " <= 0");
   }
   if (b <= 0) {
      throw new IllegalArgumentException(
         "Num.gcd.pre violated: b == " + b + " <= 0");
   }
   ...
```

- What to put in the message string?
- Convey information about what went wrong where: minimally the class and method that threw the Exception.
- Note that many methods may throw the same exception.

Handling Exceptions

- If an exception is thrown, and no code explicitly handles it, then the program terminates with a stack trace (**crash**)
- Exceptions can be handled in various ways

  (See below and next week)

Catching Exceptions

```
try {
    c = gcd(a, b); // a and b are unvalidated user input
} catch (IllegalArgumentException e) {
    // in here, use e
    System.out.println(e); // provide feedback to the user
}
```

- This code handles the exception explicitly.

  N.B. If an exception is caught nowhere, execution terminates.

- If an `IllegalArgumentException` is thrown anywhere in the try block, then execution proceeds at the start of the catch block, where the thrown exception is assigned to `e`.

Variations on Catching

- Multiple catch blocks can be used to handle different types of exceptions:

```
try { x.foobar() }
catch (OneException e) { ... }
catch (AnotherException e) { ... }
catch (YetAnotherException e) { ... }
```

- You can also use nested try statements.

- Here, the inner try statement can throw `AnotherException`.
  It is handled by the outer catch block.

```
try {
    ...
    try { ... throw new AnotherException(); ...
    } catch (SomeException e) {... throw new AnotherException(); ...}
    ...
} catch (AnotherException e) { ... }
```

Exceptions & Subtypes

```
try {...
    throw new OneException();
    ...
    throw new AnotherException();
    ...
} catch (RuntimeException e) { ... }
} catch (Exception e) { ... }
```

- Here, all exceptions occurring within the **`try`** block will be caught.
- The **`catch`** block can list a *supertype* of the exception, so that multiple exceptions that are subclasses can be caught (handled) by the same catch block.
- A compile error will occur, if a **`catch`** block for a superclass exception appears *before* a **`catch`** block for a subclass exception.

Catching Exceptions: complete version of `try` statement

```
try {
    statements_in_try_block
} catch (Type_1_Exception identifier_1) {
    statements_handling_type_1_exception
} catch (Type_n_Exception identifier_n) {
    statements_handling_type_n_exception
} finally {
    statements_to_wrap_up
}
```

1. The statements within the **`try`** block are executed.
2. *If* an exception occurs, execution of the **`try`** block stops, and the first **`catch`** block *matching the exception* is executed.
3. Statements in the **`finally`** block are *always* executed, *even if* the exception is not caught, or the catch block contains **`return`**.

Exception Handling: A word of warning

- Control flow in **`try`** statements is implicit and invisible, depending on the occurence of exceptions.
- Pretty weird control flows are possible when handling exceptions, especially when using the **`finally`** block.
- Do not abuse exceptions for 'fancy' control flow:

  `catch` blocks are usually implemented inefficiently because they are *exceptional*; that is, there is a (heavy) performance penalty when exceptions occur.

Exceptions are Exceptional

- Exceptions should not be thrown as a result of normal use of a program.
  - Catching exceptions is not efficient.
  - When reading code, you should be able to understand 'normal' behavior, by ignoring exceptions.
  - Other ways to discontinue execution in non-exceptional cases:
    - **`break`**: terminates switch, for, while, do
    - **`continue`**: skips to end of loop body
    - **`return`**: terminates method

Exceptions and Javadoc

In javadoc comment: `@throws` *Exception-class-name Condition*

```
1  /**
2   * Returns the greatest common divisor of a and b.
3   *
4   * @param   a  positive integer
5   * @param   b  positive integer
6   * @return  {@code \old(\max c; c divides a && c divides b; c)}
7   * @throws  IllegalArgumentException  if {@code a <= 0 || b <= 0}
8   * @pre     {@code 0 < a && 0 < b}
9   */
10 public static long gcd (long a, long b) throws IllegalArgumentException {
11     if (a <= 0) {
12         throw new IllegalArgumentException(
13                 "Num.gcd.pre violated: a == " + a + " <= 0");
14     } // 0 < a
15     ...
16 }
```

Exceptions and Unit Testing: Principles

When the contract of a method involves exceptions, it is necessary to test for their proper occurrence in unit tests:

- Create situations that are intended to trigger an exception.
- When the appropriate exception is thrown, the test passes.
- When no exception is thrown, the test fails.
- When the wrong exception is thrown, the test also fails.

Advise: Also test that the exception message is present.

Exceptions and Unit Testing: Simple Scheme for JUnit 4.x

```
1     /** Tests {@link xxx} for proper exceptions. */
2     @Test(expected = YyyException.class)
3     public void testXxxExceptions() {
4         xxx(Expr1, ...);
5     }
```

This test case passes

- if `xxx` throws an exception that is equal to or a subtype of `YyyException`

This test case fails

- if `xxx` does not throw an exception
- if `xxx` throws an exception that is not equal to or a subtype of `YyyException`

Exceptions and Unit Testing: General Scheme

```
1     /** Tests {@link xxx} for proper exceptions. */
2     @Test
3     public void testXxxExceptions() {
4         Class expected = YyyException.class;
5         try {
6             xxx(Expr1, ...);
7             fail("should have thrown " + expected);
8         } catch (Exception e) {
9             assertTrue("type; " + e.getClass().getName()
10                    + " should be instance of " + expected,
11                    expected.isInstance(e));
12            assertNotNull("message should not be empty", e.getMessage());
13        }
14    }
```

Also tests that exception message is not null.

Exceptions and Unit Testing: Example

```
1     /** Test of {@link WordLibrary#getWord}. */
2     @Test public void testGetWordForException() {
3        checkIndex(-1, ArrayIndexOutOfBoundsException.class);
4        checkIndex(WordLibrary.getSize(), ArrayIndexOutOfBoundsException.class)
5     }
6     /** Checks whether index i throws expected exception
7      *  @param i  the index to check */
8     private void checkIndex(int i, Class expected) {
9         try {
10            WordLibrary.getWord(i);
11            fail("index " + i + " should have thrown " + expected);
12        } catch (Exception e) {
13            assertTrue("type; " + e.getClass().getName()
14                    + " should be instance of " + expected,
15                    expected.isInstance(e));
16            assertNotNull("message should not be empty", e.getMessage());
17        }
18    }
```

Assignments Series 2

- Refresher: consult book by Eck: 3.7, 8.3, 8.4.1
- Apply Test-Driven Development, including Exceptions, to `CountDigitsWithRadix` and `Powerize`

Summary

- The contract of a robust method specifies behavior in *all* cases. When precondition is violated, an `Exception` is thrown.
- Exceptions provide a mechanism to bypass normal control flow, in case of *failures* or *special situations*, to *inform* and *avoid harm*.
- Java exceptions involve:
  - objects that are instances of `Exception` or its subclasses
  - **`throws`** clauses in method headers
  - contracts that specify 'which exceptions are thrown when', by `@throws` tags in javadoc comments
  - **`throw`** statements
  - **`try ... catch ... finally`** statements