2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Subversion repository for 2IP15 material:

  `https://svn.win.tue.nl/repos/2IP15_2012-2013_Student_Material`

  See FAQ for details.

- Robustness (as a way of dealing with defects)
- Exceptions*
- Runtime Assertion Checking (RAC)

*Reused and adapted some slide material from Alexandre Denault

Warning

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

What if the precondition is violated?

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

Intermezzo: Why is implementation of `gcd` correct?

Defininitions:

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

Properties:

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

Intermezzo: Why is implementation of `gcd` correct?

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

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

- *Partial function*: "It is up to the user to call `gcd` correctly."
- What happens if the user does call `gcd` with $n = 0$ or $d = 0$? And what if $n < 0$ or $d < 0$?
- Partial functions lead to programs that are *not robust*.
- 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 to permanent data.

Solution #2: Return Special Result

- Return 0: 0 cannot be the result of a correctly called `gcd`, so 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.

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.

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 int fact (int n) throws NegativeException
```

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.

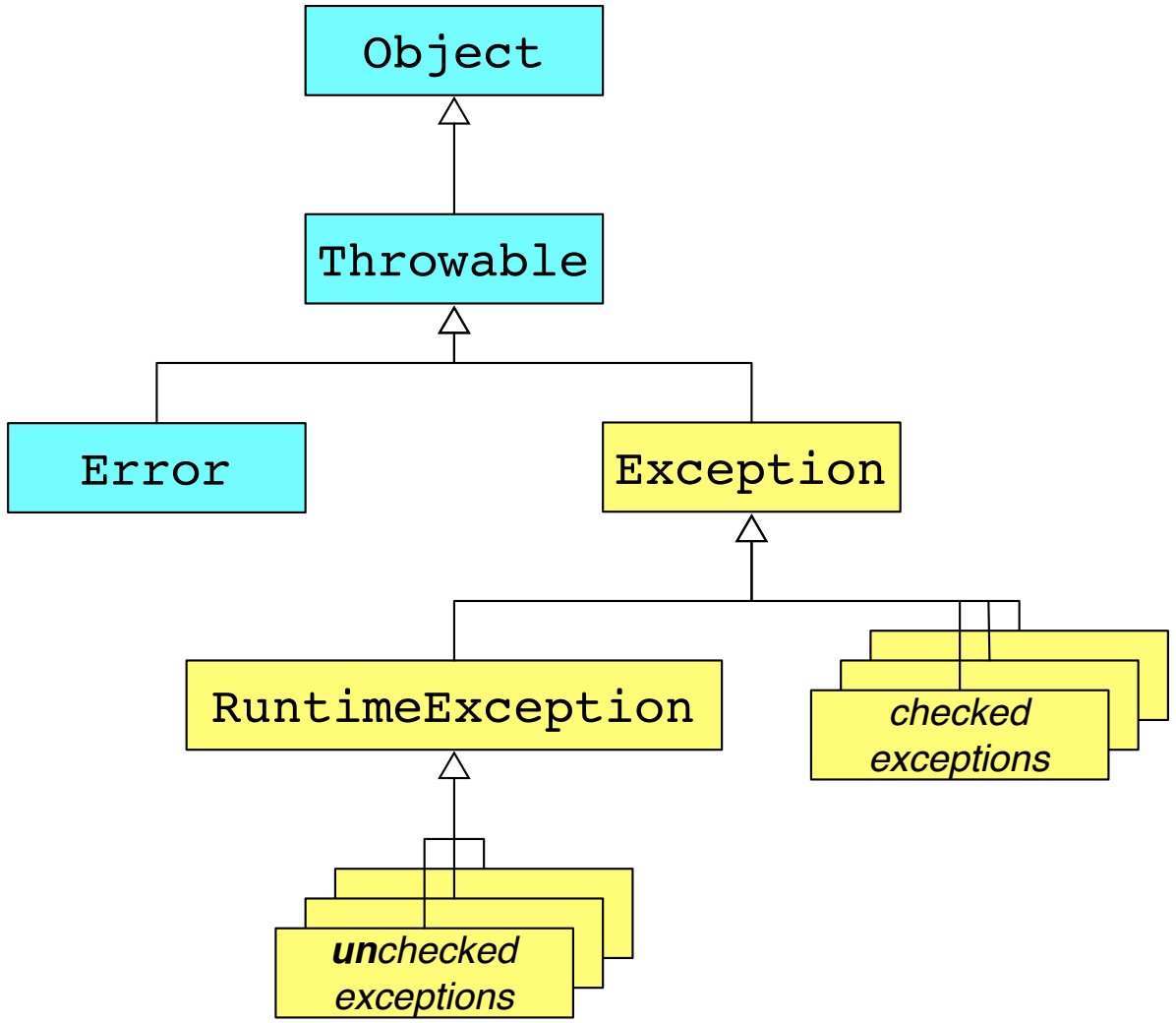
When a method has side effects, its contract should clarify how side effects interact with exceptions.

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 2IP15):

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

Defining Exceptions

- The following is minimal (does not allow a message):

```
public class MyException extends Exception { }
```

- To have a constructor with a message, you need:

```
public class MyException extends Exception {
    MyException() { super(); }
    MyException(String s) { super(s); }
}
```

- Exceptions may be much more elaborate:

```
public class MyException extends Exception {
  Object offensiveObject;
  MyException(String s; Object o) { super(s); offensiveObject = o; }
  Object getOffensiveObject() { return offensiveObject; }
}
```

Defining Exceptions (2)

- You can use `Exception` and `RuntimeException` directly, without using or defining a (new) subtype exception. But this is not good programming style, because it does not convey much information (is too vague, too general).
- You can use other predefined exceptions, when appropriate.
  E.g. `NullPointerException`, `IndexOutOfBoundsException`
- `...Exception` naming is not enforced, but this is good practice.
- In 2IP15, use `IllegalArgumentException` or `IllegalStateException` to signal precondition violations.

Where to Define Exception Types?

- Some Exceptions occur in many packages (e.g.: `NotFoundException`).
- It makes sense to avoid *naming conflicts* and define a separate Exception package.
- However, if special Exceptions are thrown by your package, you can define them inside your package.

Throwing Exceptions

```
public static int fact (int n) throws NegativeException {
    ...
    if (n < 0) {
        throw new NegativeException(
            "Num.fact.pre violated: n == " + n + " < 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.

Catching Exceptions

```
try {
    x = Num.fact(y);
} catch (NegativeException e) {
    // in here, use e
    System.out.println(e);
}
```

- This code handles the exception explicitly.

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

- If a `NegativeException` is thrown anywhere within the try block, execution proceeds at the start of the catch block, after the thrown exception is assigned to `e`.

Variations on Catching

- Multiple catch clauses 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 clauses.

- Here, the inner try block throws `anotherException`. It is handled by the outer catch clause.

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

Exceptions & Subtypes

```
try {...
    throw new oneException();
    ...
    throw new anotherException();
    ...
} catch (Exception e) { ... }
```

- This **catch** clause will catch all exceptions occurring within the try block.
- The **catch** clause can list a *supertype* of the exception so that multiple exceptions that are subclasses can be caught (handled) by the same catch clause.
- A compile error will occur, if a **catch** clause for a superclass exception appears *before* a **catch** clause 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`** clause *matching the exception* is executed.
3. Statements in the **`finally`** block are *always* executed, *even if* the exception is not caught and, hence, will be *propagated*.

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`** clause.
- Do not abuse exceptions for 'fancy' control flow:

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

Kinds of Exceptions: Checked

Checked exceptions: subclass of `Exception` but not of `RuntimeException`

- *must be listed* in the **`throws`** clause of the method that can throw the exception.
- *must be handled* by the caller code, either by
  - propagation, or
  - catching,

  otherwise, a compile-time error occurs.
- *typically, used to signal recoverable special situations*.

Example: `FileNotFoundException` thrown by `java.util.Scanner(...)`

Kinds of Exceptions: Unchecked

Unchecked exceptions: `RuntimeException` and subclasses

- *don't have to be listed* in the **`throws`** clause.

  But it is good practice to list unchecked exceptions that can be thrown, especially if it can be expected that the caller can usefully catch the exception.

- *don't have to be handled* explicitly by the caller.

- *typically, used to signal non-recoverable failures*.

Examples: `NullPointerException`, `IndexOutOfBoundsException`

Handling Exceptions Implicitly

- If method `m` calls method `p` without wrapping the call in a **`try`** clause, then an exception thrown by `p` aborts execution of `m` and the exception is propagated to the method that called `m`.
  (A calls B calls C calls D throws e, A catches e)

- To propagate a *checked* exception, the calling method must list the thrown exception. (If not, a compile error results.)

- *Unchecked* exceptions are automatically propagated until they reach an appropriate **`catch`** clause. (Not enforced by compiler)

- But you can still list unchecked exceptions in the header, and it is good practice to do so.
  (The user is made aware of the unchecked exception and can catch it if desired.)

Programming with Exceptions

- How to handle thrown exceptions?
- Possibilities:
  - Handle specifically: separate **`catch`** clauses deal with each situation in a different way
  - Handle generically: one **`catch`** clause for supertype exception takes generic action like println and halt or restart program from earlier state
  - Reflect the exception: the caller also terminates by throwing an exception, either implicilty by propagation or expclicitly by throwing a different exception (usually better)
  - Mask the exception: the caller catches the exception, ignores it, and continues with normal flow

Reflection (a.k.a. Translation)

```
1  // Coding Standard violated to fit on one slide
2      /** @throws NullPointerException  if a == null
3        * @throws EmptyException  if a.length == 0
4        * @pre a != null && a.length != 0
5        * @post \result == (\min i; a.has(i); a[i]) */
6      public static min (int [ ] a)
7              throws NullPointerException, EmptyException {
8          int m; // minimum of elements of a
9          try {
10             m = a[0];
11         } catch (IndexOutOfBoundsException e) {
12             throw new EmptyException("Arrays.min.pre violated");
13         }
14         for (int element : a) {
15             if (element < m) { m = element; }
16         }
17         return m;
18     }
```

Masking

```
1  // Coding Standard violated to fit on one slide
2      /** @throws NullPointerException  if a == null
3        * @pre a != null
4        * @post \result == a is sorted in ascending order */
5      public static boolean sorted (int [ ] a)
6              throws NullPointerException {
7
8          int prev;
9
10         try { prev = a[0]; }
11         catch (IndexOutOfBoundsException e) { return true; }
12
13         for (int element : a) {
14             if (prev <= element) { prev = element; }
15             else { return false; }
16         }
17         return true;
18     }
```

Design Issues

- When to throw exceptions?
  - To make method contracts *robust*.
  - To avoid encoding information in ordinary or extra results.
  - To signal special —usually erroneous— situations, often in non-local use of functions (that is, propagating across calls).
  - To guarantee that detected failures cannot be ignored.
- When *not* to throw exceptions?
  - When the context of use is local (consider use of `assert`).
  - When the precondition is too expensive or impossible to check.

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.
  - There are 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

Defensive Programming

- *Defensive programming* is the attitude to include run-time checks for bad situations that should never occur (but still could occur due to "unforeseen" circumstances: environment, other programmers). These situations are usually not covered in specifications.

- Exceptions or `assert` statements can be used, because they do not burden the normal control flow.

- Typically, you could use one generic unchecked exception.
  E.g. `FailureException`

```
if (unimaginable) {
    throw new FailureException(
        "class C, method M: the unimaginable happened");
}
```

Exceptions versus `assert` statements

- `assert` *booleanExpr*
- `assert` *booleanExpr* : *stringExpr*
- **`if`** `(` `!` *booleanExpr* `)` **`throw new`** `AssertionError(`*stringExpr*`);`
- Note that `AssertionError` is an `Error`, not an `Exception`.
  Cannot be caught; always aborts execution; cannot unit test it.
- Execution of `assert` is disabled by default; to enable:
  - in DrJava: Preferences > Miscellaneous > JVMs:
    JVM Args for Main JVM `-ea`
  - in NetBeans: Project Properties > Run > VM Options: `-ea`

When to Throw Exceptions in This Course?

- *Non-private* methods:
  - Strive for robustness,
    except when performance would suffer disproportionately.
  - Contracts must explicitly state conditions for exceptions.
    Use **`if`** and throw an exception, e.g. `IllegalArgumentException`
  - Can use `assert` for situations not covered by exceptions.
- *Private* methods:
  - May have stronger precondition without explicit exceptions.
  - You are encouraged to use `assert` to check precondition.

Checked versus Unchecked Exceptions

- Advantages of checked exceptions:
  - Compiler enforces proper use (code must catch or propagate).
  - Prevents wrongly captured exceptions.
- Disadvantages of checked exceptions:
  - Forces developer to deal with them explicitly, even in cases where they provably will not occur.
- Use unchecked exceptions if you expect they will not occur, because they can be conveniently and inexpensively avoided.
- Otherwise, use checked exceptions;
  i.e., when they cannot be avoided easily.

How to Handle Exceptions in This Course?

- In unit tests: Always check that exceptions are thrown properly according to the contract.
- In non-test code: Never catch unchecked exceptions, *unless* . . .
- . . . there is an important reason and good way for recovering.

  Unchecked exceptions should signal the presence of a defect.
- Checked exceptions must be handled: recover or propagate.

  Checked exceptions should signal a recoverable special situation, typically from using a standard library method (e.g. for I/O).

Exceptions and Javadoc

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

```
1  /**
2   * Returns n factorial.
3   *
4   * @param   n  non-negative integer
5   * @return  {@code n} factorial
6   * @throws  IllegalArgumentException  if {@code n < 0}
7   * @pre     {@code 0 <= n}
8   * @post    {@code \result == n!}
9   */
10 public static int fact (int n) throws IllegalArgumentException {
11     if (n < 0) {
12         throw new IllegalArgumentException(
13                 "Num.fact.pre violated: n == " + n + " < 0");
14     } // 0 <= n
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     public void testXxxExceptions() {
3         Class expected = YyyException.class;
4         try {
5             xxx(Expr1, ...);
6             fail("should have thrown " + expected);
7         } catch (Exception e) {
8             assertTrue("type; " + e.getClass().getName()
9                     + " should be instance of " + expected,
10                    expected.isInstance(e));
11            assertNotNull("message should not be empty", e.getMessage());
12        }
13    }
```

Also tests that exception message is not null.

Exceptions and Unit Testing: Example

```
1     /** Test of {@link WordLibrary#getWord}. */
2     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 `IntRelation`

Summary

- The contract of a robust method specifies behavior in *all* cases.

  When precondition is violated, throw an `Exception` or `Error`.

- Use exceptions or assertions to *inform* and to *avoid harm*.

Summary (cont'd)

- `assert` throws an `Error`; terminates execution, cannot be caught.
- Exceptions provide a mechanism to bypass normal control flow, in case of *failures* or *special situations*.
- 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