Lecture 2IP65 & 2IP70
Week 6

Specification
Exceptions

Kees Huizing
October 2009
Specification
Specification

- method only works correctly if current state (parameters, variables) satisfies some requirements:
  - precondition (requires clause)
  - method caller may assume that method produces correct result
  - postcondition (ensures clause)

```java
int[] digitFreq(int n) {
    assert n >= 0;
    int[] freqs = new int[10];
    //assert all els of freqs are 0;
    ...
}
```
Specification

- See method as a *product*
  
  - developer offers a product
  
  - programmer uses product
  
  - delivery contract with rights and duties of parties

```
programmer establishes pre

developer assumes pre

programmer assumes post

developer establishes post
```
Design by contract

method user

Contract

pre: ...
post: ...
...

method developer
Specification example

What are the pre- and postcondition?

//pre: n>=0
//post: result == r && r*r <= n && n < (r+1)*(r+1)
f(int n) {
    int r = 0;
    while (r+1)*(r+1) <= n {
        r += 1;
    }
}

pre: n ≥ 0

post: r² ≤ n < (r+1)²
or r == \lfloor \sqrt{n} \rfloor (floor)

At the call

assert x >= 0;
w = f( x );
assert w*w <= x && x < (w+1)*(w+1);

apply pre- and post to actual parameters
Exceptions
How to deal with the unexpected

• consumption = assistant.getBeer();

• what to do with consumption if there is no beer?

  1. immediately terminate program

     not elegant, no solution

  2. assign some arbitrary value to consumption

     dangerous

  3. tell about the problem without producing result

     out of band message
Out of band message

n = scanner.nextInt();

if input contains no integer, but something else:

**exception** is *thrown*

assignment is not executed

if code is prepared for exception => special *catch* code is executed

if code is not prepared => method is aborted and caller of method is informed, i.e.,
Example: without preparation

```java
void calculateResults() {
    level = sc.next();
    goalsFor = sc.nextInt();
    goalsAgainst = sc.nextInt();
    ...
}

public static void main(String[] a) {
    new Competition().calculateResults();
}

Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:840)
    at java.util.Scanner.next(Scanner.java:1461)
    at java.util.Scanner.nextInt(Scanner.java:2091)
    at java.util.Scanner.nextInt(Scanner.java:2050)
    at Competition.calculateResults(Competition.java:41)
    at Competition.main(Competition.java:83)
```

• program is aborted
Example: handling exception immediately

```java
void readMatch() {
    do {
        try {
            ...  
            goalsFor = sc.nextInt();
            ok = true;
        } catch (InputMismatchException e) {
            System.out.println("Wrong input, enter a number");
            ok = false;
        }
    } while (!ok);
    ... continue with rest of input
}
```

- try-block
- catch clause
- program is not aborted, exception is handled
Example: passing an exception to caller

```java
void readMatch() throws InputMismatchException {
    ... 
    goalsFor = sc.nextInt();
    goalsAgainst = sc.nextInt();
    matches += 1;
    ... 
}

void readAll() {
    while (...) {
        try {
            readMatch();
        } catch (InputMismatchException e) {
            System.out.println(e + "Match input ignored");
        }
    }
}
```

- **throws clause**
- *method* is aborted, exception is *passed to caller*
- chain can be as long as you want
Note

- Exceptions should be used to handle the unexpected

- Doubtful if this is the case here

- Probably better is to use `sc.hasNextInt()`
Typical Exceptions

• IO problems
  - file not found
  - permissions not ok

• Network problems
  - host does not exist
  - connection broken

• Programming problems
  - array index out of bounds
  - division by zero \textit{(not overflow!)}
Example: home cooked Exception

```java
int plus(int a, int b) throws OverflowException {
    int a;
    int b;

    if (a>0 && b>0 && a+b<0) {
        throw new OverflowException();
    }
    return a+b;
}
...

class OverflowException extends Exception {
}
```
Hierarchy of Exceptions

- Exceptions are actually objects with their classes ordered in hierarchy (next course more about this)

- Exception: any “regular” exception fits in this class

  - IOException: several exceptions dealing with IO
  - FileNotFoundException: specific IOException

- Catching respects hierarchy:

```
try {
    ...
} catch (FileNotFoundException e1) {
    // if it is a FNFE
} catch (IOException e2) {
    // if it is a IOEx, but not a FNFE
} catch (Exception e3) {
    // if it is not a FNFE
    // but any other exception
}
```
Exceptions are checked

- Most exceptions must be either be caught inside the method or declared (with `throws BlahBlahException`) in the method header

- Compiler checks this and refuses compilation if not ok

- will appear in the API: part of the contract

- sometimes annoying, in general beneficiary

- exceptions are part of the contract
Unchecked Exceptions

• Special class of exceptions need not to be declared
  - can occur in many places, often not tied to a specific method
  - often due to (avoidable) programming errors
  - declaring all these would make program unwieldy

• Class is `RuntimeException`

• Examples:
  - `IndexOutOfBoundsException`: when using wrong index into array
  - `ArithmeticException`: division by zero etc., not overflow :-(
  - `NullPointerException`: when method (or instance var) is accessed via null value
    (more about this later)
Specification of Exceptions

• add clause that describes which exception can occur and when

    // normal behaviour:
    // ...
    // exceptional behaviour:
    // pre: b == 0
    // signals: DivisionByZeroException

    int divide(int a, int b) throws DivisionByZeroException {
        ...
    }

• trade off between normal precondition and exception

    - pre: b != 0 could be added and exception removed from interface
Why exceptions

• give possibility of *out-of-band* message to caller

• let you separate code in normal behaviour and exceptional behaviour: better structure

• ditto in specification

• don’t use exceptions to code normal behaviour:

```java
// to read all input from scanner:
try {
    while (true) {
        a[i] = scanner.next();
        i++;
    }
} catch (NoSuchElementException e) { }
// continue with rest of program

don’t do this!
```
PrintWriter

public PrintWriter(File file)
    throws FileNotFoundException

Creates a new PrintWriter, without automatic line flushing, with the specified file. This convenience constructor creates the necessary intermediate OutputStreamWriter, which will encode characters using the default charset for this instance of the Java virtual machine.

Parameters:
file - The file to use as the destination of this writer. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:
FileNotFoundException - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file
SecurityException - If a security manager is present and checkWrite(file.getPath()) denies write access to the file

From this, you can conclude that you have to catch, or pass on, an FileNotFoundException when you create a PrintWriter this way.
The Java API can be consulted at http://java.sun.com/javase/6/docs/api/.
Files and I/O
General idea

• Several classes (such as Scanner) exist in package java.io

• Make object of such a class by new Scanner(...)

• On the dots fill in an object that does part of the task

• Example

  - System.in is object that handles input from keyboard

  - scanner = new Scanner(System.in) creates object that processes keyboard input on higher level (separates them into tokens (words), etc.)
Scanner (java.util)

- **Scanner** processes input from arbitrary source, chops them into tokens and offers tokens via:

  - `next()`: consume next token from input. Default: whitespace separated `words`
  
  - `nextInt()`: consume next token and interpret as integer (if possible)
  
  - `nextDouble()`, `nextBoolean()`, etc.: ditto
  
  - `nextLine()`: reads the rest of the current line and goes to the next one
  
  - `hasNext()`: returns true if there are tokens on input, false otherwise
    
    - note: usually true on keyboard input (`next()` will wait to see if there is more)
  
  - other token separating behaviour can be specified (with `useDelimiter`)
  
  - `hasNextInt()`, etc.: true if next token is in int (etc.) format
File (java.io)

- When program terminates, all data is lost
- Files store data more permanently
- An object of class `File` records information about a file on disk
- `new File("funnyfile.txt")` creates a `File` object that can be connected to the disk file `funnyfile.txt`
  - Path names are possible: `new File("C:\My Documents\funny.doc")`
- It doesn’t create or connect to a file by itself
Connecting

• a File object can be connected to a Scanner object:

File file = new File("funnyfile.txt");
    Scanner scanner = new Scanner(file);

• then reading all lines into array a (which should be big enough):

    int i = 0;
    while ( scanner.hasNext() ) {
        a[i] = scanner.nextLine();
        i++;
    }
PrintWriter (java.io)

• has objects that can perform print and println like System.out

• File file = new File("funnyfile.txt");
  PrintWriter pw = new PrintWriter(file);
  creates the file funnyfile.txt and an object to write into it.

    - if funnyfile.txt exists already, it is overwritten (contents is removed)

• Suppose a is an array of Strings, then writing the contents of a as lines into
  the file funnyfile.txt:
  for (int i=0; i<a.length; i++) {
    pw.println(a[i]);
  }

• Fixing its contents and making the file available for, e.g., input:
  pw.close();
// copies a text file line by line
import java.util.*;
import java.io.*;

public class FileCopy {
    Scanner scanner;
    File source;
    File target;
    PrintWriter pw;
    String filenameSource = "rhubarb.txt";
    String filenameTarget = "rhubarb_copy.txt";
    void copy() {
        try {
            String line = null;
            source = new File( filenameSource ) ;
            scanner = new Scanner( source );
            target = new File( filenameTarget );
            pw = new PrintWriter( target );

            while (scanner.hasNext()) {
                line = scanner.nextLine();
                pw.println( line );
            }
            pw.close();
        } catch (FileNotFoundException e) {
            System.out.println("Could not open file due to");
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        new FileCopy().copy();
    }
}