Voortgezet Programmeren

Lecture 3: Programming by contract++

Tommi Tervonen

Econometric Institute, Erasmus University Rotterdam

Counting sum of the elements of an integer array

```java
public int countSum(int[] array) {
  for (int i=1;i<array.length;i++) {
    array[i] = array[i] + array[i-1];
  }
  return array[array.length-1];
}
```

Counting sum of the elements of an integer array

```
public int countSum(int[] array) {
  for (int i=1;i<array.length;i++) {
    array[i] = array[i] + array[i-1];
  }
  return array[array.length-1];
}
```

- Returns the correct value, but also modifies the parameter array as a *side effect*.
- What would you expect from:

```
public int countSum(int[] array)
```

Side effects

- Unexpected side effects make code difficult to understand
- There are also *desired* side effects, e.g. sorting the contents of an array
- In Java we have
  - Accessor methods: returning a value but not modifying contents of the object (`public int getAge()`)
  - Mutator methods: modifying the contents of the object but not returning a value (`public void setAge(int age)`)

Functions and procedures

- In imperative programming, we similarly classify methods into
  - **Functions**, that return a value but do not alter the parameters in any way
  - **Procedures**, that alter some of the parameters but do not return a value

```
void setElement(Matrix m, int rInd, int cInd,
  double newElement)

double getElement(Matrix m, int rInd, int cInd)
```

- Note: if the language does not support exceptions (e.g. C), procedures often do return a value for signifying error conditions

Parameter passing schemes

- For side effects to be possible, parameters have to be passed *by reference*: only a reference (memory address) of the variable is passed to the called method
- Other main technique for parameter passing is to pass *by value*: a local copy of the variable is created within the called method
- Matlab passes everything by value, although Matrices are passed by references until they are modified the first time, at which point a local copy is created (!)
- Java passes primitives by value, objects by reference
- Example: passing schemes Matlab vs Java

Programming by contract

- When you design methods, there is a *contract* between the supplier (you) and the consumer (possibly someone else)
- The contract is partially defined by the signature:

```
void sortArrayFromIndex(int[] array, int index)
```

Programming by contract

- When you design methods, there is a *contract* between the supplier (you) and the consumer (possibly someone else)
- The contract is partially defined by the signature:

```
void sortArrayFromIndex(int[] array, int index)
```

Contract:

1. The `index` has to be in the range `[0, array.length-1]` (responsibility of the consumer)
2. If consumer calls the method adhering to (1), then after the method call the following holds: `array[index] <= array[index+1] <= ... <= array[array.length-1]` (responsibility of the supplier)

Pre- and post-conditions

```
/**
 * Sorts the array in ascending order starting
 * from index. I.e. guarantees the post-condition:
 * array[index] <= ... <= array[array.length-1]
 *
 * @param array the array to sort.
 * @param index the starting index.
 *   PRECOND: 0 <= index < array.length-1
 */
void sortArrayFromIndex(array, index)
```

- Responsibilities of the consumer are method *pre-conditions* ("Requires")
- Responsibilities of the supplier are method *post-conditions* ("Ensures")
- (PRECOND, METHOD) ⇒ POSTCOND

Violating pre-conditions

- As a supplier, if the pre-condition is violated, you are not responsible for what happens
- In practice you should crash the program execution by throwing an unchecked exception (e.g. `IllegalArgumentException`), as the mistake is in the logic
- By convention, `null` references should never be passed in Java (or `NullPointerException` is thrown)
- Never try to catch these exceptions

```java
public void sortFromIndex(int[] array, int index) {
  if(index < 0 || index >= array.length) {
    throw new IllegalArgumentException(”outofb”);
  }
  // ... do the actual sorting
}
```

Assertions in Java

- In addition to unchecked exceptions (e.g. `IllegalArgumentException`), java has `assert` keyword that checks for a condition
- Assertions are only enabled during development as they can do computationally expensive checks (similar convention in C, but not in Matlab!)
- Need to be enabled in Eclipse (run as / run configurations / arguments / VM arguments: add "-ea")
- Failed assertions throw `AssertionException` that you should never catch
- Do not use assertions to check pre-conditions of public methods!

When to use pre- and post-conditions

- If you cannot handle a possible parameter value, you should declare the accepted range as a pre-condition (and check / throw `IllegalArgumentException`)
- Post-conditions are often stated in a more informal manner in the method documentation
- Document post-conditions formally when making complex mathematical programs, and when you have problems finding bugs

Class invariants

- Classes can have **invariants** that hold after the constructor has finished, and before and after each method call (often stated informally)
- Throw `IllegalStateException` if the class invariant does not hold (usually a sanity check)
- Use class invariants rather than pre-conditions to have to call methods in a certain order

```
DataSet s = new DataSet("food.dat");
double[] x = s.getColumn(0);
// ^ IllegalStateException: data not loaded
s.loadData();
```

Method overloading

- Single method can have different implementations with different parameters. e.g.

```
public String() // constructs an empty string
public String(char[] value)
 // constructs a string with contents
```

- The constructor is **overloaded**. For constructors this is crucial as their name is fixed (otherwise we could have only 1 way to construct an object)
- Overloading is defined by method name and parameters (not by exceptions or return value!)

Data hiding

```
public class DataSet {

  private ArrayList<double[]> data;

  public DataSet(String fname) throws IOException,
  FileNotFoundException { ... }

  public int getNrColumns() { ... }

  public double[] getColumn(int index) { ... }

}
```

Private / public visibility

- `public` are visible to everyone
- `private` are visible only within the **class**: also other objects of the same class can access them (motivation: if you modify the variable type, you can also modify use of the uses)

```
public class Matrix {
  private double[] data;
  private int nrows;

  public Matrix(Matrix other) {
    this.nrows = other.nrows;
    copyData(other.data);
  }
  ...
}
```

Private methods

- Maximize the use of private methods for code clarity and to avoid redundance (also, in Eclipse: refactor/extract method)
- Rule of thumb: every method should fit in one screen of code

```
public DataSet(String fname) throws IOException {
  this.fname = fname;
  FileReader fr = new FileReader(fname);
  BufferedReader rdr = new BufferedReader(fr);
  loadData(fname);
}

private void loadData(Reader rdr)
  throws IOException {
    ...
}
```

Final variables

- `final` keyword declares that the value of the variable cannot be re-set

```
final int x = 2;
x = 3; // error

final Student s = new Student("tommi");
s.setName("tommi2"); // ok
s = new Student("tommi3"); // error
```

Static variables and methods

- In OOP, most methods are bound to an object they operate on (and cannot be called without the object being constructed first)
- `static` allows to create variables and methods that exist statically, i.e. can be called without the object

```
public class Math {
  ...
  public static final double PI = 3.141592654;
  ...
  public static double abs(double x) { ... }
  ...
}
```