Introduction to Java ...

...and Methods of Software Engineering

Dr. Elmar Zander

Institute of Scientific Computing

Institute of Scientific Computing

- Institut für Wissenschaftliches Rechnen (WiRe)
- Head: Prof. H. G. Matthies
- Approx. 10 scientific employees
  (Dr. J. Vondrejc, Dr. D. Liu do the math refresher)
- Main topics: uncertainty quantification, stochastic simulations, parameter estimation, fluid-structure interaction, coupling
- Courses:
  - Introduction to PDE and numerical methods
  - Introduction to Scientific Computing
  - Intermediate Programming
  - Uncertainty Quantification and Parameter Identification
  - ...

Overview

Table of contents

What we will cover this week

- Unix and the Unix shell
  (History, Linux, free software, piping, input and output, important tools...)
- Java
  (History, basic structures, object orientation, standard library...)
- Software Correctness and Debugging
  (debugging, assertions, invariants, pre- & post-conditions, unit testing)
- Version control
  (basics of version control, systems, revisions, branching&merging, conflicts, github, distributed vcs's)

Unix

Overview

- Basics of computers and operating systems
- Files, file systems and related commands
- Unix programming model (input&output)
- Redirections and pipes
- Important commands and combinations
- Shell scripts
- Stuff that didn't fit: etc, apt, free software, networking,

Basics of operating systems 1

- Which ones do you know?

- Why operating systems?
- Makes it easier for the programmer: e.g. abstraction of hardware
- Better usability for end users: security, scheduling, hardware support, rights management...

Basics of operating systems 1

- Which ones do you know?
- Windows, DOS, Unix, Linux, OSX, Android, CPM, etc.
- Why operating systems?
- Makes it easier for the programmer: e.g. abstraction of hardware
- Better usability for end users: security, scheduling, hardware support, rights management...

Basics of operating systems 2

- More to come on Unix/Linux here ...

The file system 1

- Data is stored in **files**
- **files** are stored in **directories**
  also called **folders**
- **files** can contain text, binary data, programs,
  but also directory data, links,
  and other strange stuff (devices, pipes, ...)

The file system 2

- Commands for working with files:
  - Listing `ls`
    list all files `ls -la`
  - Showing `cat`
    with paging `more`, `less`
  - Removing `rm`, `rm -rf`
  - Moving `mv`
  - Editing `gedit`, `nedit`, `emacs`, `vim`,
    also: LibreOffice, IDEs, etc.

The file system 3

- Commands for working with directories:
  - Creating: `mkdir` (make directory)
  - Removing: `rmdir` (remove, must be empty)
  - Moving `mv` (same as with files)
  - Show current: `pwd` (pwd, print working dir)
  - Changing: `cd foobar` (change to dir foobar)
    `cd -` change to last directory
    `cd` change to home directory
- Special directory names:
  - The current directory: `.`
  - The directory one leve higher: `..`
  - The home directory: ~

Unix programming model

- Have no big monolithic programs
- Small programs that do one thing (but do that well)
- Connect those programs to perform larger tasks
- Input $\rightarrow$ Program $\rightarrow$ Output

Unix programming model

- Have no big monolithic programs
- Small programs that do one thing (but do that well)
- Connect those programs to perform larger tasks
- Input → Program → Output
- Input → Program 1 → Program 2 → Output
- E.g. `cat results.txt | cut -c 5- | sort`

Redirections and pipes 1

- Need to connect different programs
- Idea: make ones programs output another programs input
- The pipe symbol | (connect programs)
- The “greater than” symbol $>$ (write to file)
- The “less than” symbol $<$ (read from file)

Redirections and pipes 2

- Some extras:
- The `tee` command: write to file *and* output
- “Greater than” with number 2 $>$ (redirect stderr)
- Standard file descriptors (stdin 0, stdout 1, stderr 2)

Important other commands and combinations 1

- Text commands
  - Output/show a file `cat`
  - Sort a file `sort`
  - Cut stuff from file `cut`
  - Reverse lines `rev`
  - Show only first/last lines: `head`, `tail`
- Search commands
  - Find a files `find`
  - Find string in file(s) `grep`

Important other commands and combinations 2

- Some important/interesting combinations
  - Cut the last 3 characters
    `cat readme.txt | rev | cut -c4- | rev`
  - Find the unique lines in a file
    `cat names.txt | sort | uniq`

Shell scripts 1

- Put often used commands into shell scripts
- Shebang: `#!/usr/bin/bash`
- Arguments `$1, $2, ...`
- Make executable: `chmod a+x filename`
- Also loops or conditions available e.g.

```
for i in *; do cp $i $i.bak; done

if grep -q doobidoo readme.txt; then
  echo Found it; else
  echo Not found;
fi
```

Shell scripts 2

- Example (put in `mkbackup`):

```
#!/usr/bin/bash
if ! -r $HOME/backup; then
  mkdir $HOME/backup;
fi;
cp -r $1 $HOME/backup/
```

- Then `chmod a+x mkbackup`
- Call `./mkbackup src`

Miscellaneous 1

- Multiple users, file access rights, scheduling, networking,
- System load `top`, `ps`, `kill`, `renice`, `lsof`,
- Ownership and rights management `chmod`, `chown`
- Environment variables (`$HOME, $PS1, $LANG, …, set, export`)
- Getting help (`man`, `info`, internet)
- standard file system: `/bin, /etc, /usr/bin, /dev, /home`

Exercises 1

- Preparation:
  - Fire up your browser
  - Go to: https://github.com/ezander/sglib
  - Download zip file
    (Click green button “Clone or download”, then “Download ZIP”)
  - Open terminal and go to download folder (`cd Downloads`)
  - Unzip the downloaded file: `unzip sglib-master.zip`

Exercises 2

- Go to `sglib-master` folder
- list files
- go to `util` folder
- list all files starting with `fun`,
- go up one folder

Exercises 2

- Go to `sglib-master` folder
  ```
  cd sglib-master
  ```
- list files
  ```
  ls
  ```
- go to `util` folder
  ```
  cd util
  ```
- list all files starting with `fun`,
  ```
  ls fun*
  ```
- go up one folder
  ```
  cd ..
  ```

Exercises 3

- Make directory `abcde`
- Remove the directory `abcde`
- Make directory `myfuncs`
- Change into directory `myfuncs`
- Copy sglib README file into current directory

Exercises 3

- Make directory `abcde`

  ```
  mkdir abcde
  ```

- Remove the directory `abcde`

  ```
  rmdir abcde
  ```

- Make directory `myfuncs`

  ```
  mkdir myfuncs
  ```

- Change into directory `myfuncs`

  ```
  cd myfuncs
  ```

- Copy sglib README file into current directory

  ```
  cp ../README .
  ```

Exercises 4

- Search for the string “since” in the README
- Remove the README file
- Copy all the m-files (extension .m) from `mathutil` into current directory
- Search in all files for the string “rosen”

Exercises 4

- Search for the string “since” in the README

  ```
  grep since README
  ```

- Remove the README file

  ```
  rm README
  ```

- Copy all the m-files (extension .m) from `mathutil` into current directory

  ```
  cp ../mathutil/*.m .
  ```

- Search in all files for the string “rosen”

  ```
  grep rosen -r .
  ```

Exercises 5

- Show the contents of the file “unittest_binfun.m”
- Use a pager to show file
- List all the files sorted
- List all the files sorted reversely (use `man sort`)
- List all the files reversing each name
- List all the unittest cutting the `unittest_` prefix

Exercises 5

- Show the contents of the file “unittest_binfun.m”

```
cat unittest_binfun.m
```

- Use a pager to show file

```
cat unittest_binfun.m | less
```

- List all the files sorted

```
ls | sort
```

- List all the files sorted reversely (use `man sort`)

```
ls | sort -r
```

- List all the files reversing each name

```
ls | rev
```

- List all the unittest cutting the `unittest_` prefix

```
ls unittest_*| cut -c10-
```

Miscellaneous 2 1

- Some important keys or key combinations
  - `Ctrl-C` stop job
  - `Ctrl-Z` interrupt job
  - `Ctrl-L` clear terminal
  - `Ctrl-R` search in history
  - `Tab` completion (programs, file names, options, ...)
- Job management
  - Start job in background (append an ampersand &)
    E.g.: `find / | sort | uniq > uniq_files.txt &`
  - Continue job in foreground `fg`
  - Continue job in background `bg`

Java - Basics

Overview

- Some bits of history...
- Developed by Sun Microsystems (James Gosling et al.)
- Intent: Interactive television
- Then: Interactive web content (Java applets)
- Now: desktop and server applications, Android apps, ...
- Dominant features: platform independent (WORA), security model
- Inheritance: C, C++, SmallTalk

Principles

There were five primary goals in the creation of the Java language:

- It must be ”simple, object-oriented, and familiar”.
- It must be ”robust and secure”.
- It must be ”architecture-neutral and portable”.
- It must execute with ”high performance”.
- It must be ”interpreted, threaded, and dynamic”.

(from Wikipedia)

Hello World

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!"); //
        Prints the string to the console.
  }
}
```

Colors: blue =Java keywords, red = Strings, green = Comments,
black = Identifiers and operators

Hello World

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!"); //
        Prints the string to the console.
  }
}
```

Datatypes: `String, "void"`

String literal between quotes

`[]` indicates array

Hello World

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!"); //
        Prints the string to the console.
  }
}
```

Object orientation: `class`, `static`

Special identifiers: `main`, `System.out.println`

Hello World

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!"); //
        Prints the string to the console.
  }
}
```

Code blocks between { and }

End statements with semicolons ;

Hello World

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!"); //
        Prints the string to the console.
  }
}
```

Compile with `javac HelloWorld.java`
Creates `HelloWorld.class`
Run with `java HelloWorld`

Variables and Datatypes

Variables must be **declared** before use!

The most important datatypes:

```
int i = 3;
double d = 6.3;
boolean isItTrue = false;
String name = "Farin Urlaub";
char initial = 'F';
```

New datatypes can be created via `classes`.

Arrays

```
int[] nums = {1, 3, 6, 10};
double[] float_nums = new double[10];
System.out.println(nums.length);
```

- Initialize with array literal or allocate with `new`.
- Get length of array with `.length`.
- No 2d arrays, no vectorised operations.
- Access elements with `[]`, e.g. `nums[2]`
- Indexing is 0-based

Operators

The usual!

- Arithmetic `+`, `-`, `*`, `/` (no power)
- Boolean `&&`, `||`, `!`
- Comparison `==`, `!=`, `>`, `>=`, `<`, `<=` (Important: `.equals()`)
- Other stuff: `.`, `[]`,

```
10+4
true && (i>5)
i*5==j/10
s.equals("abcde")
```

Strings

- Special overload for the `+` operator
- Behaves like array (`[]` and `length`)
- Compare with `equals()`
- Non modifiable

```
String s = "abcde";
s[0] == 'a';
String s2 = s + 3; // s2 is now abcde3
```

String conversion

- To string: `"" + var`
- Objects: `.toString()` method
- Read double number: `double d = Double.parseDouble("1.234")`

```
public static void main(String[] args) {
  double d = Double.parseDouble(args[1]);
}
```

Structured programming

- Branching/conditional constructs: `if`, `else`, `switch`, `case`, `default`
- Looping constructs: `while`, `for`, `do`
- Related: `break`, `continue`
- Objects and methods

Conditionals: if

- Structure: `if(condition){ code block }`
- or: `if(condition){ code block } else {code block 2}`
- or: `if(condition){ code block } else if (condition2) {code block 2} else {code block 3}`

```
double d = Double.parseDouble(args[1]);
if( d>0 )
  System.out.println("It's positive");
else if( d<0 ) {
  System.out.println("It's negative");
}
else
  System.out.println("It's zero");
```

While loops

- Structure: `while(condition){ code block }`
- Loop runs while `condition` is fulfilled

```
double d = 1;
while( Math.abs(d*d-2)>1e-10){
  d = 0.5 * (d + 2/d);
}
```

For loops

- Structure:
  `for(initialisation; condition; increment){code block}`
- `initialisation` runs before execution of the loop
- `condition` must be fulfilled for loop to run/continue
- `increment` executed after each iteration
- Equivalent to

```
initialisation;
while(condition) {
  code block;
  increment;
}
```

For loops 2

Examples:

```
for( int i=0; i<10; i++){
  System.out.println(i);
}
for( double d=2; d<1e10; d*=1.5){
  System.out.println(d);
}
```

Exercise 1

Write a HelloWorld program

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!"); //
        Prints the string to the console.
  }
}
```

- Write/copy the program above with your favorite editor (e.g. gedit) and save into `HelloWorld.java`
- Compile the program with `javac HelloWorld.java`
- Check on the command line that there is a `HelloWorld.class`
- Run the program with `java HelloWorld`

Exercise 2

Write a HelloWorld program

```
public class Hello {
  public static void main(String[] args) {
    // Yours to fill in
  }
}
```

- Modify the last program so that it takes your name and print "Hello WhatEverYourNameIs"
- Compile the program and run it with `java Hello WhatEverYourNameIs`

Exercise 3

Write a program/programs that

- reads two numbers from the command line and outputs the numbers, their sum, and their difference (Remember: `Double.parseDouble`, `System.out.println()`)
- reads one numbers and adds up all numbers from 1 up to the given number, e.g. `java SumIt 4` should print 10
- that outputs all arguments given to it via the command line, e.g `java SumIt 4 foo 3.3` should output ”Argument 1 is 4”, ”Argument 2 is foo” and ”Argument 3 is 3.3”
- that reads a number from the command line and outputs whether it is odd or even

Java - Advanced

Classes

What is a class?

- A template for creating objects
- Objects are created from it (instances of the class)
- Creation using a constructor
- Can contain fields (data members, instance variables, attributes, properties...)
- Can also contain methods (implementations)

A simple class

A very primitive class (but we'll improve it)

```
public class Point {
  public double x;
  public double y;
}

// somewhere else in your code
Point p = new Point();
p.x = 2;
p.y = 4;
```

Constructors 1

Let's make fields private and add a constructor

```
public class Point {
  private double x;
  private double y;

  public Point(double x0, double y0) {
    x = x0;
    y = y0;
  }
}

// somewhere else in your code
Point p = new Point(2, 4);
```

Note: now you can't call `Point()` anymore

Constructors 2

Let's add a second constructor

```
...
  public Point(double x0, double y0) {
    x = x0; y = y0;
  }
  public Point() {
    x = 0; y = 0;
  }
...

// somewhere else in your code
Point p = new Point(2, 4);
Point p2 = new Point();
```

Note: now you can call `Point()` again (initialize to origin)

Field access 1

How to set a coordinate now?

- Either: create a setter method
- Or: make class immutable (don't allow setting coordinates)

```
  public double getX() {
    return x;
  }
  public void setX(double x) {
    this.x = x; // what does "this.x" mean?
  }
...
// somewhere else in your code
Point p = new Point();
p.setX(3); // Now p is at (3,0)
```

Immutable classes 1

Second version

```
  public double getX() {
    return x;
  }
  public Point setX(double x) {
    return new Point(x, this.y);
  }
// somewhere else in your code
Point p = new Point();
Point p2 = p.setX(3); // Now p2 is at (3,0), p
    still at (0,0)
Point p3 = new Point(p2.x, 5); // Now p3 is at
    (3,5)
```

Immutable classes 2

You will find some immutable classes in the Java Standard Library

- `String`
- Numeric classes: `Double`, `Boolean`, `Byte`, `Integer`, ...

Advantages

- Can be used as keys in Maps
- Thread-safety

Static vs. Non-static fields

Static fields don't need an instance of the class

```
public class Point;
  double x, y;
  static double numPoints;

  public Point(double x, double y) {
    this.x = x; this.y = y;
    numPoints = numPoints + 1;
  }
  ...
// somewhere else in your code
Point p = new Point(1,2);
Point p2 = new Point(3,4);
System.out.println(Point.numPoints);
```

Inheritance and Polymorphism 1

Extending a class

```
public class Person {
  String firstname, surname;
}

public class Employee extends Person {
  String role;
  double wage;
}
```

Employees have also a name (like any Person), but also some additional information

Inheritance and Polymorphism 2

```
public class Person {
  String toString() {
    return surname + ", " + firstname;
  }
}
public class Employee extends Person {
  String toString() {
    return super.toString() + "(" + role + ", "
        + wage + ")";
  }
}
```

- The class that was extend is called the **super class**
- Can be referenced by `super`, to disambiguate

Inheritance and Polymorphism 3

```
Employee emp = new Employee("Washington",
    "George", "President", 1000);
Person p = emp;
p.toString(); // calls the Employee's toString
    method
// i.e. it returns "Washington, George
    (President, 1000)"
```

- The variable is of type Person but actually references an Employee object (polymorphism)
- Could reference anything derived from Person, directly or indirectly

Inheritance and Polymorphism 4

```
Person[] persons = new Persons[4];
persons[0] = new Employee(...);
persons[1] = new Person(...);
persons[2] = new BusDriver(...);
persons[3] = new Rockstar(...);

for(int i=0; i<persons.length; i++) {
  System.out.println(persons[i]);
}
```

- The class that was extended is called the **super class**
- Can be referenced by `super`, to disambiguate

Static vs. Non-static methods

Static methods don't need an instance of the class

```
class Point {
  public double distance(Point other) {
    double dx = x - other.x;
    double dy = y - other.y;
    return Math.sqrt( dx*dx + dy*dy );
  }

  public static double distance(Point p1, Point
      p2) {
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y;
  ^^Ireturn Math.sqrt( dx*dx + dy*dy );
  }
}
```

Static vs. Non-static methods

If both (static and non-static) are good to have, implement one in terms of the other

```
public double distance(Point other) {
  return Point.distance(this, other);
}
public static double distance(Point p1, Point
    p2) {/*implementation here*/}
```

...or ...

```
public double distance(Point other) {...}
public static double distance(Point p1, Point
    p2) {
  return p1.distance(p2);
}
```

Packages 1

- A way to group related classes
- Relationship to directory organisation (directory names must match package names, similar to class and file names)
- Classes in directory "./foo" would be in package "foo"
- Classes in directory "./foo/bar" would be in package "foo.bar"
- Class "Baz" in directory "./foo/bar" would be "foo.bar.Baz"

Packages 2

How to use packages (importing and exporting)

```
// In file foo/bar/Baz.java
package foo.bar;
class Baz {
  ...
}
```

```
// In file ./Main.java
import foo.bar.Baz;
import foo.bar.*; // import all from foo.bar

Baz b = new Baz();
```

Packages 3

Standard packages

- `java.lang` - all the basic classes (automatically imported)
- `java.util` - data structures (maps, stacks, vectors, calendar, ...)
- `java.io` - File reading, input and output streams, `Scanner`, see also `java.nio`
- `java.math` - pretty small, only big integers and decimals (standard functions in java.lang.Math)
  for more checkout e.g. Apache Commons Math
- `java.net` - networking facilities

Generics 1

- Generics are ”parametrized” Classes
- The parameter is another type (or class)
- Example: A vector class that can grow the number of elements
- Do not want to implement that for all possible types
- Type becomes a parameter

Generics 2

```
public class Vector<T> {
  T[] elements;
  int num;

  void  add(T t){
    T[num++] = t; // very crude impl.
  }};
// Somewhere else in your code
Vector<Person> persons = new Vector<>();
}
```

Generics 3

Used a lot in the java collections classes, e.g.:

- `List<T>`
- `ArrayList<T>`
- `Vector<E>`
- `Map<K,V>`
- `HashMap<K,V>` (needs Hashable)
- `TreeMap<K,V>` (needs Comparable)

What we did not cover

- Abstract classes
- Interfaces
- Lambda functions
- (Anonymous inner classes)
- Static imports
- Boxing and unboxing

Exercise 0

- Today we'll be working with an IDE (integrated development environment),
  the one we'll be using is called NetBeans
- Go to the command line and type `netbeans`
- Create a new project (of type Java Application), give it some name and click finish
- In the main class add some println statement to the main method and click "Run" (the green triangle)
- Now, go on with the following exercises adding more classes and methods to your project ...

Exercise 1

- Implement the `Point` class
- Implement a constructor a `toString` method and a `main` method in `Point`, so that you can a) run it b) an object on the command line
- Add a distance function
- Print the distance between `Point(1,1)` and `Point(4,5)` (should be 5, right?)
- Create both versions (static and non-static of the distance function)
- Optional: add methods to move the point some distance, mirror it at the origin, mirror it at some axis

Exercise 2

- Implement the `Person` and `Employee` classes
- Add `toString` methods to them
- Create a bunch of objects, put them into an array, and write a loop that outputs all of those objects
- Instead of the standard array try to use a `Vector` or `ArrayList` from `java.util`
- Put your object into a `TreeMap<String,Person>` that get's indexed by the person's last name
- then try to find a person by last name in that map (read the API documentation on how to do that)

Software testing and Debugging

Software bugs

Sad truth about programming:

- There will be bugs ...
- ...always ...

Software bugs

Sad truth about programming:

- There will be bugs ...
- ...always ...
- What can we do about it?

Remedies for software bugs

Making “sure” code is bug free

- Design by contract (preconditions, postconditions, invariants)
- Correctness proofs
- Assertions
- Testing (unstructured vs. structured)
  unit tests, integration tests, acceptance tests, etc.

Finding bugs

- Command line and visual debuggers
- Advanced techniques (e.g. stochastic methods)

Assertions 1

- assert = “make sure, that”
- Java keyword `assert`
- Make sure “expression” is true: `assert expression;`
- Issue some error message otherwise: `assert expression : "some text";`
- Can stay in code (also in production code)
- Can be enabled/disabled via runtime switch (`-ea,-enableassertions`)

Assertions 2

Example 1 (check returned from other functions)

```
...
   Point p = polygon.getLastPoint();
   assert p!=null : "Last point must exist";
   p.move(2, 3);
...
```

Assertions 3

Example 2 (check input parameters)

```
...
   int factorial(int n) {
      assert n>=0 : "n must be non-negative";
      // implementation
   }
...
```

Assertions 3

Example 3 (check your own assumptions)

```
int doStuffMod3(int n) {
  if( n \% 3 == 0 ) {
    System.out.println("n is a multiple of
        3");
  }
  else if( n \% 3 == 1 ) {
    System.out.println("n mod 3 is 1");
  }
  else { // now n \% 3 must be 2
    System.out.println("n mod 3 is 2");
  }
}
```

Assertions 3

Example 3 (check your own assumptions)

```
int doStuffMod3(int n) {
  if( n \% 3 == 0 ) {
  ...
  }
  else { // make comment explicit
  ^^I assert n \% 3 == 2 : "n mod 3 must be
     2, I think";
    System.out.println("n mod 3 is 2");
  }
}
doStuff(-4);
```

Pre- and postconditions 1

- Design by Contract
  idea from Eiffel language (originally from formal verification, correctness proofs, Hoare logic ...)
- Idea: relationship between some supplier and a client
- Client has to fulfill their obligations (e.g. pay fee, specifiy exactly what is to be produced)
- Then supplier has to fulfill their obligations, too
- Specification in a contract

Pre- and postconditions 2

- In software, caller is the client, method is the supplier
- Preconditions have to be fulfilled before a methods runs
- Postconditions have to fulfilled by the method **if** the preconditions were fulfilled
- In derived classes:
  preconditions can only be weakened
  postconditions can only be strengthened

Pre- and postconditions 3

- Example:
- Square root function `sqrt(d)`
- Precondition: `d>=0`
- Postcondition: `s*s=d` with `s=sqrt(d)`
- Possible ways to change the contract: no precondition, but specify what has to be done if `d` is negative (exception, return zero, ...)

Pre- and postconditions 4

- Not directly implemented in Java
- Can be “simulated” with asserts (but not perfectly)
- In Java: use `assert` at start of method and before every `return`

Unit testing 1

- How do you test your software?
- Most typical:

```
public static int factorial(int n) { ... }
public static void main(String[] args){
    System.out.println(factorial(3));
}
```

Does it print 6?

Unit testing 2

- Previous approach does not scale.
- What if you modify your function? (For efficiency, more input parameters, different runtime environment...)
- Solution: automated test i.e.
  let the computer do what you just did,
- In this example:
  compare result of `factorial(3)` to `6`
  but put that into code
- Do this for small testable units: **unit test**

Unit testing 3

- For Java different testing frameworks exist
- Most well-known: JUnit
- Originator Kent Beck (test driven development)

Unit testing 4

Example:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MathUtilTest {
    @Test
    public void testFactorial() {
        int actualRes = MathUtil.factorial(4);
        int expectedRes = 24;
        assertEquals(expectedRes, actualRes);
    }}
```

(Rest shown in the IDE)

Debugging 1

So, you use assertions, unit tests, but still have bugs.
How you find it?

- Debuggers, build into many IDEs directly (most comfortable)
- Also command line (jdb, ddd, gdb, ...)
- Much quicker than "`printf` debugging"

Debugging 2

Tools:

- Breakpoints (unconditional and conditional)
- Variable inspections
- Stepping (in, over, out)
- Call stack

Exercise Assertions 1

- Create a new project with NetBeans
- In the `main` class create a function `int sum(int n)` that computes the sum $1 + 2 + \cdots + n$

Exercise UnitTesting 1

- Create a new project with NetBeans
- In the `main` class create a function `int sum(int n)` that computes the sum $1 + 2 + \cdots + n$
- Create a preliminary test using `System.out` from the `main` function to see that your function works
- Now, in NetBeans in the project pane to the left, right-click the application class (JavaApplication123 or whatever its name is)
- In the context menu go to ”Tools” (second from bottom), then in the submenu to ”Create/Update tests” (last entry on the bottom)
- In the dialog box, deselect everything that contains “Initializer” or “Finalizer” (we don’t need that), then click ”Ok”

Exercise UnitTesting 2

- In the project pane you will find now a category ”Test Packages” (below ”Source Packages”) and there a Java file ”JavaApplication123Test.java”. Open that Java file.
- Inspect the generated code and imagine what it does (and read the generated comments).
- Remove the test code for the `main` method (we won’t test it) and all the calls to the `fail` method.
- Now insert a sensible test into the ”testSum” method (think about a valid argument for *n* what the method is supposed to return for that *n*)
- From the ”Run” menu choose ”Test project (Alt+F6)”
- On the bottom you should see a green line saying something like ”Tests passed” (or red and ”Tests failed” if you had an error)

Exercise UnitTesting 3

- Introduce a bug into the `sum` function (e.g. let it return -16 always) and rerun the tests (click the double green triangles left to the green line)

Version control

Why version control

- Why version control?
- Tracking changes (when did the code break, when was this feature introduced, ...)
- Merging modifications (teams, development on different machines, ...)
- Trying out things (rollback, branching&merging)

Where can version control be used

- Source code (Java, Matlab, C, ...)
- Text files (txt, markdown, LaTeX, ...)
- Configuration files, makefiles, ...
- Works not so well on: binary files, word files (doc, docx, xsl, ...)

Version control systems - A bit of history

- Started in 1972: SCCS (Source Code Control System)
- RCS (Revision Control System)
- CVS (Concurrent Versions System, 1989)
- **SVN** (Subversion, 2000)
- **git** (means nothing, 2005, Linus Torvalds et al.)
- **mercurial** (2005)
- Plus lots of other systems

General steps with a version control system 1

Basic things

- Initialise the **repository**
- Add local files to the repository
- **Commit** a set of changed files to the repository (creates a new **version**)
- **Push** changes to a central repository
- **Update/pull** files from a central repository

General steps with a version control system 2

More advanced things to do

- Look at the version history of your repository or of a single file
- **Check out** an old version
- Compare different versions
- Create a **branch** (e.g. for testing new features)
- **Merge** different branches

First steps with git

- Initialise a repository
- Goto “root” directory (for the new repo)
- Type `git init`
- Add files `git add` (adds only to the stage set/index)
- Commit with `git commit` or `git commit -m "message"`
- Note: per default git asks to enter the message in `vim`
- Set editor with `git config --global core.editor "gedit"`

(BTW: set prompt to something sensible: e.g. `export PS1="\w> "`)

Looking at history

- Use `git log`
- Contains: author, date, commit message
- Furthermore: commit number, parent commits
- Many options to configure: `--oneline`, `--reverse`, `--shortstat`, ...
- Easier: graphical interfaces (e.g. gitk)

Working with remote repos

- Cloning (i.e. download to local file system) `git clone URL`
- Pushing commit `git push` (fetch and merge)
- Just fetching `git fetch`
- Pulling remote changes `git pull`

Branching and merging

- Create a new branch `git branch testbranch`
- Checkout the new branch `git checkout testbranch`
- See on which branch you are `git status`
- Switch back to master branch `git checkout master`
- See difference to branch `git diff master testbranch`
- Merge into master branch `git merge testbranch master`

Viewing differences

- Use `git diff`
- Diff to prev version `git diff HEAD^`
- Diff to second prev version `git diff HEAD^^`
- Get some commit number (SHA1 hash) from log
- Diff to second prev version
  `git diff d0902c2e7cfa6448d61e66f87e3541b5cafe024b`
- Checkout version
  `git checkout d0902c2e7cfa6448d61e66f87e3541b5cafe024b`
- Checkout master `git checkout master`

Exercise 1

For playing around we will first use a repository from github

- Create a directory for your git tests (e.g. `gittest`) and `cd` into it
- Clone the github repo:
  `git clone`
  `https://github.com/PhilJay/MPAndroidChart.git`
- Cd into `MPAndroidChart`
- Look at the history of the project using `git log`
- Try to find out who committed most, when was the first commit,
- Remember the first day? Can you figure out all committers to the project using the shell commands `grep`, `sort` and `uniq`?

Exercise 1

For playing around we will first use a repository from github

- Create a directory for your git tests (e.g. `gittest`) and `cd` into it
- Clone the github repo:
  `git clone`
  `https://github.com/PhilJay/MPAndroidChart.git`
- Cd into `MPAndroidChart`
- Look at the history of the project using `git log`
- Try to find out who committed most, when was the first commit,
- Remember the first day? Can you figure out all committers to the project using the shell commands `grep`, `sort` and `uniq`?

```
git log | grep Author | sort | uniq | less
```

Exercise 2

More history with `git log`

- Try `git log --oneline`
- Try `git log --shortstat`
- Try `git log --reverse`
- Combine the options above
- Try `git log --author="Name"` to find all commit by ”Daniel”

There are many more options. These are here only to show you the possibilities. For more type `git help log`

Exercise 3

Investigating a repository with a GUI

- Start `gitk` in the `MPAndroidChart` directory
- Look at the history of commits and the branch lines
- Scroll down to the bottom to see the first commits
- Click some of the first commits to see which files have been changed
- Click some of the changed files to see what changed exactly in those files

Exercise 4

Creating your own repository

- Create a new directory and `cd` into it
- Type `git init` to initialize the new repo
- Type `git status` to see the status of the repo
- Type `git log` (you will see an error message because there aren't any commits yet)
- Now create a `readme.txt` file e.g. with gedit.
  (Note: the readme file does not have to contain any sensible text.)
- Add the readme to your repository using by first staging it, (i.e. `git add readme.txt`)
- and then committing with `git commit -m "My first commit"`)

Exercise 5

Creating your own repository (2)

- Look again at the history of your repo (do that after every commit)
- Change the readme file, then add and commit again, and revisit the history
- Look at the changes you made with `gitk`
- Add the java files you have created the days before to your repository