Organisation

Dates

- Lecture is Wednesday 16–18 and Friday 10–11.
- Tutorial is on Friday 11–12.
- Exercise sheets are available on the website on Wednesday.
- Solution must be mailed to the tutor until next Wednesday.

To successfully participate, you must

- do the exercises,
- actively participate in the tutorial,
- pass an oral examination.
Motivations

Why are formal methods interesting?

- improve code quality,
- improve productivity.
Motivations

Quality

- Leads to better understood code.
- Different view point reveals bugs.
- A formal proof can rule out bugs entirely.

Productivity

- Error detection in early stages of development.
- Modular specifications allow reuse of components.
- Documentation, maintenance.
- Automatic test case generation.
- Clearer specification leads to better software.
public static int factorial(int n) {
    int result = n;
    while (--n > 0)
        result *= n;
    return result;
}

We need a specification!
Adding Pre- and Postcondition

```java
/*@ requires n >= 0;
 @ ensures \result == n! ;
 @*/

public static int factorial(int n) {
    int result = n;
    while (--n > 0)
        result *= n;
    return result;
}

Is program correct?
No: case n=0 gives wrong result.
```
JML – Java Modelling Language

JML is an Extension of Java for Design by Contract.

- http://www.jmlspecs.org/
- Release can be downloaded from http://sourceforge.net/projects/jmlspecs/files
- JML compiler (jmlc)
- JML runtime assertion checker (jmlrac)
In JML the specification precedes the method in `/*@ ... @*/`.

- **requires** formula: The specification only applies if formula holds when function called. Otherwise behaviour of method is **undefined**.
- **ensures** formula: If the function exits normally formula has to hold.
- **assigns** variables: The function only changes values of variables
- **signals** (exception) formula: If the function signals exception then formula holds.
- **signals only** exceptions: The function may only throw exceptions that are a subtype of one of the exceptions. If omitted function can signal only exceptions that appear in `throws` clause.
- **diverges** formula: The function may only diverge if formula holds.
A JML formula is a Java Boolean expression. The Java language is extended by some JML operators:

- \texttt{\textbackslash old(expression)}: The value of expression before the method was called (used in signal and ensures clause)
- \texttt{\textbackslash result}: The return value (used in ensures clause).
- \texttt{F =\Rightarrow G}: States that \texttt{F} implies \texttt{G}. This is an abbreviation for \texttt{! F || G}.
- \texttt{\forall Type t; condition; formula}: States that formula holds for all \texttt{t} of type \texttt{Type} that satisfy condition.
In JML class invariants are also in /*@ . . . @*/.

- **invariant formula**: Whenever a method is called or returns, the invariant has to hold.

- **constraint formula**: A relation between the pre-state and the post-state that has to hold for each method invocation.
If factorial is not a builtin operator

Solutions (1): Weakening of specification
/*@ requires n >= 0;
   @ ensures \result >= 1;
   @*/

public static int factorial(int n) {
    int result = n;
    while (--n > 0)
        result *= n;
    return result;
}

+ Simple Specification
+ Catches the bug
  - Cannot find all bugs
  - Gives no hint, what the function computes
If factorial is not a builtin operator

Solutions (2): JML: Pure java functions.

```java
/*@ requires n >= 0;
   @ ensures (n == 0 ==> \result == 1)
   @ \&\& (n > 0 ==> \result == n*fact(n-1)); */
public static @pure int fact(int n) {
    return n <= 0 ? 1 : n*fact(n-1);
}
```

Pure functions must not have side-effects and must always terminate.
The pure function can be used in specification:

```java
/*@ requires n >= 0;
   @ ensures \result == fact(n);
   @*/
public static int factorial(int n) {
    int result = 1;
    while (n > 0)
        result *= n--;
    return result;
}
```
Partial specifications

Giving a full specification is not always practical.

- Code is repeated in the specification.
- Bugs in the code may also be in the specification
  ⇒ bugs are not always detected.
Example for Partial Specifications

Factorial example:
/*@ requires n >= 0; @*/
@ ensures \(\text{result} > 0\); @*/

Documenting when it throws exceptions:
/*@ requires true; @*/
@ signals (java.lang.IllegalArgumentException) n < 0;
@ ensures \(n \geq 0\) && \(\text{result} > 0\); @*/

Incomplete list of expected behaviour:
/*@ requires true; @*/
@ ensures \(\text{result}.\text{contains}(e)\)
@ \(\land \\forall \text{Elem } f; \this.\text{contains}(f); \text{result}.\text{contains}(f)\); @*/
List add(Elem e);
Semantics for Java

The Java Language Specification (JLS) 3rd edition gives semantics for Java

- The document has 684 pages.
- 118 pages to define semantics of expression.
- 42 pages to define semantics of method invocation.

Semantics are only defined by prosa text.
Example: What does this program print?

```java
class A {
    public static int x = B.x + 1;
}
class B {
    public static int x = A.x + 1;
}
class C {
    public static void main(String[] p) {
        System.err.println("A: " + A.x + ", B: " + B.x);
    }
}
```
Example: What does this program print?

JLS, chapter 12.4.1 “When Initialization Occurs”:
A class $T$ will be initialized immediately before the first occurrence of any one of the following:

- $T$ is a class and an instance of $T$ is created.
- $T$ is a class and a static method declared by $T$ is invoked.
- A static field declared by $T$ is assigned.
- **A static field declared by $T$ is used** and the field is not a constant variable.
- $T$ is a top-level class, and an assert statement lexically nested within $T$ is executed.
Example: What does this program print?

JLS, chapter 12.4.2 “Detailed Initialization Procedure”:
The procedure for initializing a class or interface is then as follows:

1. Synchronize on the Class object that represents the class or interface to be initialized. This involves waiting until the current thread can obtain the lock for that object.

2. . . .

3. If initialization is in progress for the class or interface by the current thread, then this must be a recursive request for initialization. Release the lock on the Class object and complete normally.

4.–8. . .

9. Next, execute either the class variable initializers and static initializers of the class, or the field initializers of the interface, in textual order, as though they were a single block, except that final class variables and fields of interfaces whose values are compile-time constants are initialized first.

10.– . . .
Example: What does this program print?

class A {
    public static int x = B.x + 1;
}

class B {
    public static int x = A.x + 1;
}

class C {
    public static void main(String[] p) {
        System.err.println("A: \_
        + A.x + ",\nB: \_
        + B.x);    
    }
}
Example: What does this program print?

If we run class $C$:

1. main-method of class $C$ first accesses $A.x$.
2. Class $A$ is initialized. The lock for $A$ is taken.
3. Static initializer of $A$ runs and accesses $B.x$.
4. Class $B$ is initialized. The lock for $B$ is taken.
5. Static initializer of $B$ runs and accesses $A.x$.
6. Class $A$ is still locked by current thread (recursive initialization). Therefore, initialization returns immediately.
7. The value of $A.x$ is still 0 (section 12.3.2 and 4.12.5), so $B.x$ is set to 1.
8. Initialization of $B$ finishes.
9. The value of $A.x$ is now set to 2.
10. The program prints “$A: 2, B: 1$”.