Lecture 18

We are still discussing the *process* of producing near-professional-quality software.

This lecture may serve several useful purposes. First, it may be a source of ideas that lead you to writing better code. Second, the lecture will act as a revision lecture for many of the parts of Java that we have covered so far — this is a useful thing to do before we move on to more new technical material. Third, it will give some insight into how Java programs are executed on different platforms.

Summary: What can a class definition contain?

- Constructors
- Instance methods
- Class methods (including, in particular, a `main` method)
- Instance variables
- Class variables
- Other (not really covered in this course): static initialization blocks, nested class definitions

Constructors

- If the programmer doesn't define any constructors, Java supplies one default constructor (the so-called 'no-arg' constructor) 'for free':

  ```
  public Wreck()
  {
  }
  ```

- If it didn't do this, how would we create objects from class definitions with no programmer-defined constructors?
- If there is at least one programmer-defined constructor (whether it be no-arg or not), Java doesn't supply the default

```
Wreck w1 = new Wreck(); // compiles
Fraction f = new Fraction(); // doesn't
```


Methods: class methods & instance methods

- *Class methods* do not use `this` (either implicitly or explicitly)
  - Some (`private`) helper methods will be class methods, e.g. `gcd`
  - `main` methods are class methods
  - Hence, our test-drivers are class methods
  - Some utility functions will be (`public`) class methods, e.g. `Math.abs`, `Math.random`
- All other methods will be *instance methods*

Variables: Java's 4 kinds

- *Local variables:* for temporary storage during the body of a method
- *Formal parameters:* for temporary storage of actual parameters during the body of a method
- *Instance variables & class variables:* to store things between one method invocation and another
  - *Instance variables:* per-instance — the value may be different for different objects
  - *Class variables:* per-class — 'only' two uses:
    - named constants (`final`)
    - when you are assigning unique numbers to objects (to keep track of last number used)

Good use of variables

- Use each variable that you declare
- Ensure proper initialisation
- Use each variable for only one purpose
- Minimise identifier scope
- Minimise variable lifetime
- Minimise variable visibility
- Use named constants, not magic numbers/strings

Major determinants of software quality

- A good design
  - procedural abstraction, data abstraction
  - high cohesion, low coupling
- A good process
  - especially inspection & testing
- Application of consistent coding conventions
- Constant review of all aspects of quality
  - e.g. by pair programming

Readability

Programming is primarily about communicating with humans, and only secondarily about communicating with machines.

- Readability is about
  - Consistency & Convention
- E.g.
  - a scheme for identifiers
  - a scheme for alignment, indentation, spaces
  - a scheme for organisation of the contents of a class definition
  - self-documenting code (due to good variable & method names and good commenting)

Portability

- Java was designed with Internet programming in mind. Portability was paramount.
- Slogan

  Write once, run anywhere
- Java's approach:
  - there is a Java language specification
  - which is unusually comprehensive (e.g. specifies the representation for `int`, `double`, etc.)
  - and it has a basis in law (licensing)
  - Plus the use of *bytecode*. . .

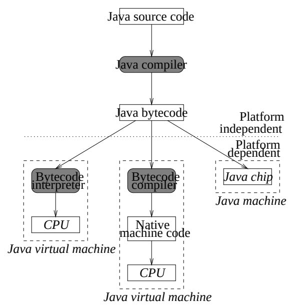
Revision: Compilers & interpreters

- *Compilers* translate programs, e.g. from Java to machine code
- *Interpreters* take statements of a program in turn, decode them, and execute coresponding machine code instructions
- *Just-in-time (JIT) compilers* translate methods, method-by-method, just before they are called
- Java has a compiler but it translates Java programs into *bytecode*, rather than *native machine code*

Bytecode

- Bytecode is the machine code of the *Java virtual machine* (JVM)
- The JVM is an abstract machine. There are different ways of getting your machine to act like this abstract machine (i.e. to understand the bytecode programs):
  - Use an *interpreter* to take bytecode instructions and execute corresponding native machine code operations on the underlying CPU
  - Use a *just-in-time compiler* to translate the bytecode to native machine code on the fly and execute the native machine code
  - Use a *Java chip* (one that has bytecode as its native machine code); the bytecode can be executed directly

The platform-independence of Java programs



Once the program has been compiled, it can be executed on any number of platforms without changing the program and without recompiling.