Java Revision++

We have been commissioned to write a class definition from which clients can construct objects that represent *fractions*. We are to provide simple fraction arithmetic methods.

This example serves to revise nearly all topics covered last year.

High-Level Summary of our Approach

What to do when you are ready to write a class definition:

- Determine the class interface (if you haven't done so already)
- Build the class definition from a template
- For each item in the interface
  - write a header and a header comment first
  - then write test cases
  - only then write the implementation
  - inspect what has been written
  - compile, run on the test cases and debug

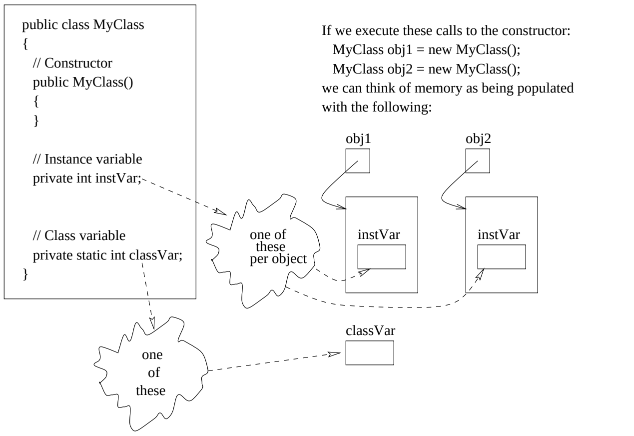
Testing

- Importance of manual source code inspection
- Test-drivers
  - automate part of testing; facilitate regression testing; provide a record of testing
- Test suite
  - Black-box: normal, boundary, exceptional
  - Glass-box: all-paths/all-statements
  - Random: avoids subconsciously under-testing parts you're not sure of

Observations about the `Fraction` class definition

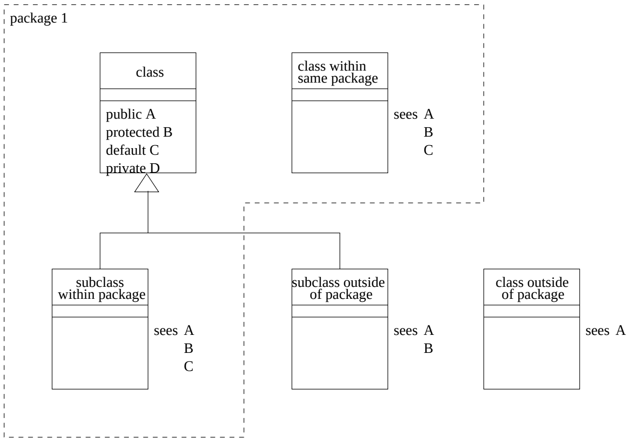
- Heavily commented including method/constructor preconditions
  - three kinds of comments in Java
  - `javadoc` comments for `public/protected` elements
- Clear separation of interface from implementation
  - hidden implementation; low coupling
- Code re-use identified where possible
- More test code than actual code

Java class/instance variables/methods



Instance variables (and methods) versus class (`static`) variables (and methods)

Java visibility



(From Coad et al 1999)

Primitive types	Reference types
Examples	
`int, double, boolean`	arrays, `String`, `Object`, `Integer`, `Burger`, `Fraction`
Contents	
Contains a value	Contains a reference ('pointer') to an object
Equality testing	
`==`	• `==` tests for equal references (pointing to same object) • `equals` inherited from `Object` is like `==` • override `equals` to compare the state of the objects

Primitive types	Reference types
System.out.println(. . .)	
Displays the value	Runs `toString` • `toString` inherited from `Object` displays the pointer • override `toString` to display the state of the object
Assignment	
Copies the value	Copies the reference (hence two pointers to same object)
Passing as a parameter to a method	
Copies the value so changes made in the method do not affect the original	Copies the reference (hence two pointers to same object) so changes to the object made in the method do affect the original