

# Java 2

---

## Student Client Side

Object

Pointer

Constructor

Message

Receiver

## Method

Code stored in class

Message-Method Lookup

Message sent to a receiver

Receiver looks in its class for matching method

That method executes against the receiver

## Receiver Relative

The code runs "on" or "against" the receiving object

Any state operations happen to the receiver

x.foo() changes the state of "x"

y.foo() changes the state of "y"

Method code is written with a "receiver relative" style -- very convenient

## Receiver Relative (Method, Ctor)

e.g. "units"

instance variables automatically that of the receiver

Self message send

"setUnits(units - drop);" -- easy to send a message keeping the same receiver

"this"

"this" is a pointer to the receiver

Don't write "this.units", write: "units"

Don't write "this.setUnits(5)", write "setUnits(5);"

## How it's implemented

Like an extra, hidden receiver pointer parameter

## Receiver/Noun Style

You think a little differently about your code -- code is grouped around the noun it operates on

## Constructor

Like a method

## Bug control

Make it easier for the client to do the right thing since objects are always put into an initialized state

## Every var goes in Ctor

Every time you add an instance variable to a class, go add the line to the ctor that inits that variable.

## "private" (3 layers)

Visible to class only

Use: ivars, private utility methods

Methods, ctors

Instances can see in each other

## 3 Layers

1. instance sees its own data, and nobody else does-- best, strict form
2. instance sees inside another instance **of its class** -- ok maybe (e.g. alice can see inside bob -- they are both Student objects)
3. instance can see inside an instance of some other class -- bad, forbidden

## "public"

Visible to all

"Official" class interface

## Not removed (vs. deprecated)

public features will not be removed in a future rev

Other classes can depend on these features

Sun deprecates some public features, so new code won't be written with them, but it very rarely removes a formerly public feature

private things can be removed from an implementation at will

## "protected"

Similar to "private" but allows access to subclasses

## static

### Method or variable

### Exists once

Not associated with an instance

### No receiver

### `public static void main(String[] args);`

Execution begins at a method like this

### `Student.demo()`, NOT `a.demo()`;

`a.demo()` actually compiles, but it discards `a` as a receiver and translates to the same thing as `Student.demo()` (there is no receiver object for a static method)

```
// Student.java
/*
 Demonstrates the most basic features of a class.
 Language points (not the usual sort of comments I would
 put in production code) are marked with "NOTE".
*/

/*
 A student is defined by their current number of units.
 There are standard get/set accessors for units.

The student responds to getStress() to report
their current stress level which is a function
of their units.

NOTE A well documented class should include an introductory
comment like this. Don't get into all the details -- just
introduce the landscape.
*/
public class Student extends Object {
    // NOTE this is an "instance variable" named "units"
    // Every Student object will have its own units state
    // "protected" means clients cannot access it directly
    protected int units;

    /* NOTE
     "public static final" declares a public readable constant that
     is associated with the class -- it's full name is Student.MAX_UNITS.
     It's a convention to put constants like that in upper case.
    */
    public static final int MAX_UNITS = 20;

    // Constructor for a new student
    public Student(int initUnits) {
        units = initUnits;
    }
}
```

```

}

// Standard accessors for units
public int getUnits() {
    return(units);
}

public void setUnits(int units) {
    if ((units < 0) || (units > MAX_UNITS)) {
        return;
        // Could use a number of strategies here: throw an
        // exception, print to stderr, return false
    }
    this.units = units;
    // NOTE: trick to allow param and ivar to use same name
}

/*
Stress is a linear function of units.

NOTE example of "Receiver Relative" coding
-- "units" refers to the state of the RECEIVER.
Code is written relative to an implicitly present receiver.
*/
public int getStress() {
    return(units*10);
}

/*
Try to drop the given number of units.
Does not drop if would go below 9 units.
Returns true if the drop succeeds,
*/
public boolean dropAClass(int drop) {
    if (units-drop >= 9) {
        setUnits(units - drop);
        return(true);
    }
    return(false);
}

/*
Here's a static test function with some simple
client-of-Student code.
NOTE Invoking the "Student" class from the command line runs this.
It's handy to put test/demo/sample client code in the main() of a class.
*/
public static void main(String[] args) {
    // Make two students
    Student a = new Student(15);
    Student b = new Student(12);

    // They respond to getUnits() and getStress()
    System.out.println("a units:" + a.getUnits() +
        " stress:" + a.getStress());
}

```

```

System.out.println("b units:" + b.getUnits() +
    " stress:" + b.getStress());

//a.setUnits(a.getUnits() - 4); // drop that class!
a.dropAClass(4); // Now there's a method that just does it

System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());

// Now "b" points to the same object as "a"
b = a;
b.setUnits(10);

// So the "a" units have been changed
System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());

/*
OUTPUT...
  a units:15 stress:150
  b units:12 stress:120
  a units:11 stress:110
  a units:10 stress:100
*/
}
}

/*
Things to notice...

-Demonstrates the Object-lifecycle -- clients create the object (must go
through constructor), then send it messages. Hard for the client to mess
up the state of the object. Note how setUnits() can maintain the internal
correctness of the object.

-The implementation code can refer to instance variables like "units"
by name. It automatically binds to the ivar of the receiver.

-"units" is declared protected. Therefore, a client cannot write something like
"a.units++". The client must go through public messages like setUnits().
This promotes a less fragile design. The client may access things declared
"public".

-State vs. Computation -- notice that the client can't really tell if stress is
stored or computed. It just appears to be a property that Students have. Whether
it is stored or computed is just a detail. This is a nice separation between
the abstraction exposed by client and how it is actually implemented.
*/

```