# Java 2

# Student Java Example

As a first example of a java class, we'll look at a simple "Student" class. Each Student object stores an integer number of units and responds to messages like getUnits() and getStress(). The stress of a student is defined to be units * 10.

## Implementation vs. Interface

In OOP, every class has two sides...

1. The **implementation** of the class -- the data structures and code that implement its features.
2. The public **interface** that the class exposes for use by other classes.

With a good OOP design, the interface is smaller and simpler than the implementation. The public interface is as simple and logical as possible -- exposing only aspects that the clients care about.

We'll use the word " client" to refer to code that uses the public interface of a class and "implementation" when talking about the guts of a class.

## Student Client Side

First we'll look at some client code of the Student class.

Client code will typically allocate objects and send them messages.

With good OOP design, being a client should be easy.

Client code plan

- Allocate objects with "new" -- calls constructor
- Objects are always accessed through pointers -- shallow, pointer semantics
- Send messages -- methods execute against the receiver
- Can access public, but not private/protected from client side

## Object Pointers

The declaration "Student x;" declares a pointer "x" to a Student object, but does not allocate the object yet.

Java has a very simple and uniform memory system. Objects and arrays are allocated in the heap and accessed through pointers.

There is no "&" operator to make a pointer to something in the stack and there is no pointer arithmetic. The only pointers that exist in java point to objects and arrays in the heap -- simple.

- Objects and arrays are allocated with the "new" operator (below).
- Using = on an object pointer just copies the pointer, so there are multiple pointers to the one object (aka "shallow" or "sharing").

Likewise, using == on object pointers just compares the pointers (the equals() message can be set up to do a "deep" comparison of two objects).

# new Student() / Constructor

The "new" operator allocates a new object in the heap, runs a constructor to initialize it, and returns a pointer to it.

```
x = new Student(12)
```

Classes define "constructors" that initialize objects at the time new is called. Constructors are similar to methods, but they cannot be called at will. Instead, they are run automatically by the JVM when a new object is created.

The word "constructor" is generally written as "ctor"

The constructor has the same name as the class. e.g. the constructor for the "Student" class is named "Student"

There can be multiple constructors. They are distinguished at compile time by having different arguments -- this is called "overloading".

e.g. The Student class defines one ctor that takes an int argument, and one ctor that takes no arguments.

The ctor that take no arguments is called the "default" ctor. If it is defined, the system uses it by default when no other ctor is specified.

# Message send

Send a message to an object.

```
a.getUnits();
b.getStress();
```

Finds the matching method in the class of the receiver, executes that method against the receiver and returns.

The Java compiler will only allow message sends that the receiver actually responds to.

# Object Lifecycle

The client allocates objects and they are initialized by the class ctor code

The client then sends messages which run class method code on the objects.

The client essentially makes requests -- all the code that actually operates on the objects is defined by the class, not the client.

As a result, if the class is written correctly, the client should not be able to introduce new bugs in the class and visa-versa.

This is the benefit of using public/private to keep the client and the implementation separate.

# Student Client Side Code

```
// Make two students
Student a = new Student(12);  // new 12 unit student
Student b = new Student();    // new 15 unit student (default ctor)

// 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.dropClass(3);   // a drops a class


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


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


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


// NOTE: public vs. private
// A statement like "b.units = 10;" will not compile in a client
// of the Student class when units is declared protected or private

/*
 OUTPUT...
   a units:12 stress:120
   b units:15 stress:150
   a units:9 stress:90
   a units:10 stress:100
*/
```

# Student Implementation Side

Now we'll look at the implementation of the Student class which is in the file Student.java. The complete code listing is given at the end of this section..

## Class Definition

A class defines the instance variables and methods used by its objects.
Each variable and method may be declared as "public" if it may be used by clients, or "private" or "protected" if it is part of the implementation and not for use by clients.
The compiler and JVM enforce the public/private scheme.

## Public Interface

The most common public/private scheme is...
All ivars are declared private
Methods to be used by clients are declared public -- these make up the interface that the class exposes to clients.
Utility methods for the internal use of the class are declared private.

## Java Class

The convention is that java classes have upper case names like "Student" and the code is in a file named "Student.java".
By default, java classes have the special class "Object" as a superclass. We'll look at what that means later when we study superclasses.

Inside the Student.java file, the class definition looks like...

```
public class Student extends Object {
  ... <definition of the Student ivars and methods> ....
}
```

The "extends Object" part can be omitted, since java classes extend Object by default if there is no "extends" clause.

There are not separate .h and .c files to keep in synch -- the class is defined in one place.

This is a nice example of the "never have two copies of anything" rule. Keeping duplicate info in the .h and .c files in synch was a bore -- better to just have one copy.

# Instance Variables

Instance variables (ivars) are declared like ordinary variables -- a type followed by a name.

```
protected int units;
```

An ivar defines a variable that each object of this class will have -- allocates a slot inside the object. In this case, every Student object has a int ivar called "units" in it.

The object itself is allocated in the heap, and its ivars are stored inside it. The ivars may be primitives, such as int, or they may be pointers to other objects or arrays.

# public/private/protected

An ivar or other element declared private is not accessible to client code. The element is only accessible to the implementation inside the class.

Suppose on the client side we have a pointer "s" to a Student object. The statement "s.units = 13;" will not compile if "units" is private or protected.

"protected" is similar to private, but allows access by subclasses or other classes in the same package (we will not worry about those cases)

"public" makes something accessible everywhere

# Constructor (ctor)

A constructor has the same name as the class.

It runs when new objects of the class are created to set up their ivars.

```
public Student(int initUnits) {
  units = initUnits;
}
```

A constructor does not have a return type (unlike a method).

New objects are set to all 0's first, then the ctor (if any) is run to further initialize the object.

Classes can have multiple ctors, distinguished by different arguments (overloading)

If a class has constructors, the compiler will insist that one of them is invoked when new is called.

If a class has no ctors, new objects will just have the default "all 0's" state. As a matter of style, a class that is at all complex should have a ctor.

Bug control

Ctors make it easier for the client to do the right thing since objects are automatically put into an initialized state when they are created.

Every ivar goes in Ctor

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

Or you can give an initial value to the ivar right where it is declared, like this... "private int units = 0;" -- there is not agreement about which ivar init style is better.

# Default Ctor

A constructor with no arguments is known as the "default ctor".

```
public Student() {
 units = 15;
}
```

If a class has a default ctor, and a client creates an instance of that class, but without specifying a ctor, the default ctor is automatically invoked.

e.g. new Student() -- invokes the default ctor, if there is one.

# Method

A method corresponds to a message that the object responds to

```
public int getStress() {
 return(units * 10);
}
```

When a message is sent to an object, the corresponding method runs against that object.

Methods may have a return type, int in the above example, or may return void.

Message-Method Lookup sequence

Message sent to a receiver

Receiver knows its class and looks for a matching method

The matching method executes against the receiver

# Receiver Relative Style (Method, Ctor)

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

Ivar read/write operations in the method code use the ivars of the receiver

Method code is written with a "receiver relative" style where the state of the receiver is implicitly present. This style will really grow on you.

e.g. the "units" ivar in the Student methods is automatically that of the receiver

Likewise, sending a message to the same receiver from inside a method requires no extra syntax.

e.g.. inside the Student dropClass() method, the code sends the setUnits() message to change the number of units with the simple syntax: setUnits(units - drop);

# "this" -- receiver

"this" in a method

"this" is a pointer to the receiver

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

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

Some programmers, like sprinkling "this" around to remind themselves of the OOP structure involved, but I find it distracting. The nice thing about OOP is the effortlessness of the receiver-relative style.

# ivar vs. local var

Usually, you simply refer to each ivar by its name -- e.g. "units". Sometimes you have a local var with the same name as the ivar, in which case the expression "this.units" refers to the ivar. Having a local var with the same name as an ivar is a stylistically questionable, but it can be handy sometimes. Some people prefer to give ivars a distinctive name, such as always starting with an "m" -- e.g. mUnits.

# Student.java Code Example

```
// Student.java
/*
 Demonstrates the most basic features of a class.

 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 variable.
   // "protected" and "private" mean that clients do not get access
   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;
   public static final int DEFAULT_UNITS = 15;


   // Constructor for a new student
   public Student(int initUnits) {
      units = initUnits;
      // NOTE this is example of "Receiver Relative" coding --
      // "units" refers to the ivar of the receiver.
      // OOP code is written relative to an implicitly present receiver.
   }

   // Constructor that that uses a default value of 15 units
   // instead of taking an argument.
   public Student() {
      units = DEFAULT_UNITS;
   }


   // 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: "this" trick to allow param and ivar to use same name
}


/*
 Stress is units *10.

 NOTE another example of "Receiver Relative" coding
*/
public int getStress() {
   return(units*10);
}


/*
 Tries to drop the given number of units.
 Does not drop if would go below 9 units.
 Returns true if the drop succeeds.
*/
public boolean dropClass(int drop) {
   if (units-drop >= 9) {
      setUnits(units - drop);    // NOTE send self a message
      return(true);
   }
   return(false);
}

/*
 Here's a static test function with some simple
 client-of-Student code.
 NOTE Invoking "java Student" 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(12);  // new 12 unit student
   Student b = new Student();    // new 15 unit student

   // 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.dropClass(3);   // a drops a class


   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());


        // NOTE: public vs. private
        // A statement like "b.units = 10;" will not compile in a client
        // of the Student class when units is declared protected or private

        /*
         OUTPUT...
           a units:12 stress:120
           b units:15 stress:150
           a units:9 stress:90
           a units:10 stress:100
        */
    }
}
```