

Java 3 — Inheritance

From last time

`cslibrary.stanford.edu`

The Binky video is there + information on how easy it was to build

See <http://cslibrary.stanford.edu/102/> for a refresher on memory

When your web startup makes you a billionaire, feel free to donate to this project

Abstraction failure

These a good paper to be written here — "The Library Myth -- how far to trust them"

Two levels of "private"

"public" contract

Once a class makes something public, it should not be removed and its abstraction should not change in the future (other class depend on the public features)

Like a CPU instruction set -- instructions get added, but they can **never** be removed.

[Arrays and Strings -- prev handout]

String Methods

Docs

I'm listing these common ones for your convenience -- in reality you should have a browser window open to the sun docs so you can just look this stuff up as you go...

<http://java.sun.com/products/jdk/1.2/docs/api/index.html>

<http://java.sun.com/products/jdk/1.2/docs/api/java/lang/String.html>

`int length()`

`boolean compareTo(String)`

`boolean compareToIgnoringCase(String)`

`String concat(String)`

`String toLowerCase()`

`int indexOf(int ch, [int fromIndex])`

`int indexOf(String, [int fromIndex])`

String substring(int beginIndex, [int
endIndex])

Naming bug: endIndex is actually the index of the position one more than the region copied

char charAt(int index)

OOP Structure

(See the "OOP Concepts" handout)

Hierarchy

Inheritance

Overriding

Superclass (general, fewer constraints)

Subclass (specific, more constraints)

Animal Example

Animal, cow, bird, duck, finch, penguin

Animal.alive -- most common/general

Bird.fly, Bird.feathers

Penguin.fly override

"Collective Noun" feature

e.g. "Bird" refers to something which is "Bird or lower (a subclass of Bird)" -- a natural inheritance hierarchy in language

Bird is actually "abstract" -- you never see literally a bird walking down the street.

You only see the more specific subclasses (finch, penguin, ...)

Bird is an abstraction, not a real thing.

OOP Summary

Logical structure for similar classes

Makes the body of code easier to understand -- for authors or clients

Reduce code repetition

Factor common code up

Less code to write

Better leverage in debugging -- fix things in one place ("never have two copies of anything" (data or code) rule)

Subclassing Design

Student defined by int units

Students has some "stress" level

Grad builds on Student

More stressed
int yearsOnThesis

"ISA "

Grad isa Student -- Grad objects have all the properties that Students have
e.g. Penguin ISA Bird
This is strong constraint on Grad

getStress() override

Grad (subclass) more properties

Grad more constrained
Grade more specific

Student (superclass) fewer properties

Student less constrained
Student more general

Subclass Niche

Exploit superclass as much as possible

Add just what's different -- delta

e.g. "Horse / Zebra"

Student/Grad Design Diagram

Class Diagram -- Very Useful

('•' = instance variable, '-' = method)

State

Behavior

Class hierarchy relationship

```

Student
•units
-ctor
-get/set Units
-getStres

Grad
•yearsOnThesis
-ctor
-get/set YOT
-getStres (override)

```

Student

```
// Student.java
```

```
/*
```

```
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.
```

```
(A well documented class should include an introductory
comment as above. Don't get into the details -- just
introduce the landscape.)
```

```
Language points (not the usual sort of comments I would
put in a class) are marked with "NOTE".
```

```
*/
```

```
public class Student extends Object {
    protected int units;
    public static final int MAX_UNITS = 20;
    /* 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 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);
}

/*
Here's a static test function that acts like a trivial client
of the Student class.
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) {
    Student s = new Student(10);
    Grad g = new Grad(10, 2);

    System.out.println("s " + s.getStress());
    System.out.println("g " + g.getStress());

    g.setUnits(g.getUnits() - 4); // drop that class!

    System.out.println("g " + g.getStress());

    /*
    OUTPUT...
    s 100
    g 202
    g 122
    */

    // Substitution rule -- subclass may play the role of superclass
    s = g; // ok

    // At runtime, this goes to Grad.getUnits()
    // Point: message/method resolution uses the RT class of the receiver,
    // not the CT class in the source code.

```

```

// This is essentially the objects-know-their-class rule at work.
s.getUnits();

//g = s; // NO -- CT error -- substitution does not work this direction
g.setYearsOnThesis(2); // how could this work if the above were
allowed?
}
}
}
/*
Other overall 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.

-units could be declared "private" in which case the subclasses
would need to go through the accessors. There's an "it depends" design
decision: is the subclass designed in close association with the superclass
(protected), or are the two supposed to be as independent as possible
(private).

-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 client and implementor.
*/

```

Grad

```

// Grad.java

/*
Grad is a subclass of Student.
Grad adds the state of yearsOnThesis.

Grad overrides getStress() to provide a Grad specific version.
NOTE another example of a to-the-point overview comment.
*/
public class Grad extends Student {

    private int yearsOnThesis;

    public Grad(int units, int yearsOnThesis) {
        // NOTE "super" must be first if used --
        // chains up to the superclass constructor
        super(units);

        this.yearsOnThesis = yearsOnThesis;
    }

    /*
    Grad stress is based on twice the Student stress

```

plus an additional factor for the yearsOnThesis.

NOTE: avoid code repetition between subclass/superclass at all costs --that's why we use Student.getStress() for the core of our computation.

```

*/
public int getStress() {
    // NOTE "super" still invokes message/method resolution
    // but it starts the search one class higher up
    // (there is no super.super)
    int student = super.getStress();

    return(student*2 + yearsOnThesis);
}

// Standard accessors
public void setYearsOnThesis(int yearsOnThesis) {
    this.yearsOnThesis = yearsOnThesis;
}

public int getYearsOnThesis() {
    return(yearsOnThesis);
}
}

```

/*

Things to notice...

-The ctor takes both Student and Grad state -- the Student state is passed up to the Student ctor.

-getStress() is a classic override. Note that it does not repeat the code from Student.getStress(). It calls it using super, and fixes the result. The whole point of inheritance is to avoid code repetition.

-Grad responds to every message that a Student responds to -- either
a) inherited such as getUnits()
b) overridden such as getStress()

-Grad also responds to things that Students do not, such as getYearsOnThesis().

*/

Summary

Subclass adds to superclass base

Factor common code to superclass

Receiver remembers its RT class

Message/Method resolution uses RT class