

# Java 3

---

## OOP Part 1 — Modularity

### Abstraction vs. Implementation

Client vs. Implementor

**2x500 better than 1x1000**

CS is about dividing your 1000 line program into two 500 line programs.

OOP modularity is a good way to do this

### Advantages

Parallel team progress

Work on smaller parts

Libraries

### Library appeal

One person writes the class, a million use it later on. Depends on good modularity.

## OOP Modularity e.g. — Tetris

### Piece object

-rotate

-getWidth

-getHeight

### Board object

-drawBoard

-placePiece

-unPlacePiece

-clearRows

### Brain object

-computeMove

## The Point

Divide by nouns

Associate behavior with noun

## OOP Part 2 — Inheritance

Arrange several related classes in a way to avoid duplication / promote code re-use.

### OOP Hierarchy

### Superclass / Subclass

### Inheritance

### Overriding

## Student Inheritance

Student defined by int units

Grad is everything that a Student is + the idea of yearsOnThesis

"isa" relationship with its superclass -- Grad isa Student

Subclass has **all** the properties of its superclass + a few

Grad overrides getStress() with a specialized version

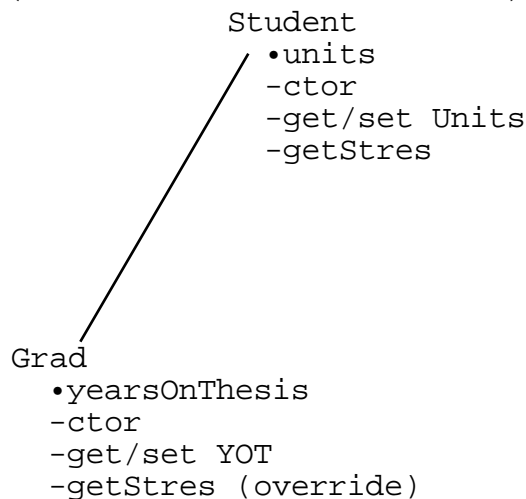
Grad (subclass) has more properties / is more constrained / more specific

Student (superclass) has fewer properties / is less constrained./ more general

## Student/Grad Design Diagram

The following is an excellent sort of diagram to make early in the design to think about the division of responsibility between a Superclass and its Subclass.

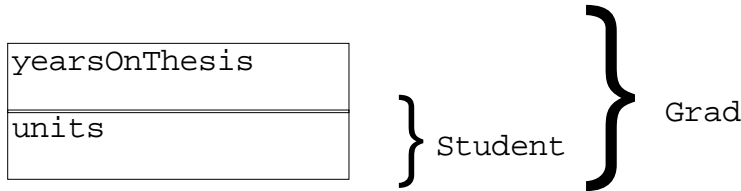
('•' = instance variable, '-' = method)



## Student/Grad Memory Layout

The ivars of the subclass are layered on top of the ivars of the superclass

Result: have a pointer to the base - can treat subclass same as superclass and it all works.



## Never Forget Class

No matter what code is being executed, the receiver is the same receiver (even if the code is in a different class) and the receiver never forgets its class.

EG even `getUnits()` (Student class) executing on a Grad, remembers that the receiver is a Grad

## Semantics of "Student s;"

NO: "s points to a Student object"

YES: "s points to an object that is at least a Student"

YES: "s points to an object that responds to all the messages Students respond to"

YES: "s points to a Student, or a subclass of Student"

## Substitution Example

Subclass can be used in a context which call for the superclass

This works because of the ISA property -- Grad ISA Student

```
Student s = new Student(10);
```

```
Grad g = new Grad(10);
```

```
s.getStress(); // ok -- goes to Student.getStress
```

```
g.getStress(); // ok -- goes to Grad.getStress -- OVERRIDING
```

```
g.getUnits(); // ok -- goes to Student.getUnits -- INHERITANCE
```

## Insomnia Example

```
static boolean insomnia(Student s) {
    return(s.getStress() > 100);
}
```

1. Can pass it a Student or Grad instance to operate on
2. The `s.getStress()` lines still does the right thing

## Insomnia 2

Suppose `insomnia` is implemented up in the Student class...

```
...
public boolean insomnia() {
    return(getStress() > 100);
    // Pops DOWN to Grad.getStress()
}
```

```
// if the receiver is a Grad
}

..
client code...
Student s = new Student(...);
Grad g = new Grad(...);

s.insomnia();          // does the right thing
g.insomnia();          // does the right thing
```

## g.insomnia() Series

Where does the code flow go....

1. Student.insomnia()
2. Grad.getStress() // pop-down
3. Student.getStress() // the super.getStress call in Grad.getStress

## Up-Down Rule

The receiver knows its class

The flow of control jumps around different classes

No matter where the code is executing, the receiver knows its class and does message->method mapping correctly

## Class Relationships x 3

### 1) Subclass

A very strong statement about the relationship between two classes

#### **Asymmetric**

Animal : Bird ("bird isa animal")  
 Vehicle : Watercraft : Kayak  
 Collection : Stack  
 Drawable Thing : Drawable Thing which can also be clicked on : Button  
 Airplane : Engine NO ("engine isa airplane"? no)  
 Collection : Int NO  
 Stack : Collection NO (it's backwards)

### 2) Has-A

The common "owning" relationship

#### **Asymmetric**

Airplane : Engine  
 HashTable : String  
 TetrisGame : Tetris Piece

### 3) Cooperates With

Common "cooperates with" / sends messages to relationship

# Symmetric

TetrisGame : KeystrokeInputObject  
 TetrisGame : DrawingArea  
 Airplane : Airport  
 // Grad.java

## Grad.java

```

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

  Grad overrides getStress() to provide a Grad specific version.
*/
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);
    }

    public static void main(String[] args) {

```

```

Student s = new Student(10);
Grad g = new Grad(15, 2);
Student x = null;

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

g.dropAClass(3); // drop that class!
// Note how g responds to everthing s responds to

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

/*
  OUTPUT...
s 100
g 302
g 242
*/

// Substitution rule -- subclass may play the role of superclass
x = 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.
x.getUnits();

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

/*
Things to notice...

-The ctor takes both Student and Grad state -- the Student state is passed up
to the Student ctor by the first "super" line in the Grad 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) overrident such as getStress()

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

```