OOP Inheritance

Thanks to Nick Parlante for much of this handout

Previously we looked at basic OOP encapsulation and API design. Here we start looking at inheritance.

Inheritance Warning

- Inheritance is a clever and appealing technology.
- However, it is best applied in somewhat rare circumstances -- where you have several deeply similar classes.
- It is a common error for beginning OOP programmers to try to use inheritance for everything.
- In contrast, applications of modularity and encapsulation and API design may be less flashy, but they are incredibly common.
- That said, for the cases where inheritance fits, it is a fantastic solution.

Inheritance Vocabulary

- OOP Hierarchy
- Superclass / Subclass
- Inheritance
- Overriding
- "isa" -- an instance of a subclass isa instance of the superclass. The subclass is just a refined form of the superclass. A subclass instance has **all** the properties that instances of the superclass are supposed to have, and it may have some additional properties as well.

General vs. Specific

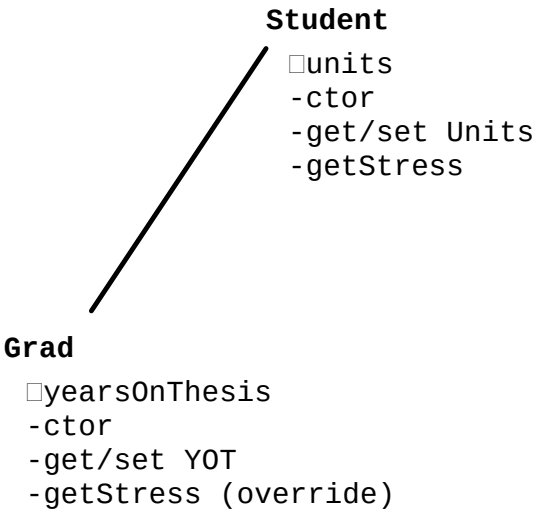
- The "super" and "sub" terms can be a little counterintuitive...
- Superclass has fewer properties, is less constrained, is more general (confusingly, the word "super" can suggests a class with more properties)
- Subclass has more properties, is more constrained, is more specific

Grad Subclassing Example

- Suppose we have a Student class that encapsulates an int "units" variable, and responds to getUnits(), `setUnits()`, and `getStress()`. Stress is a function of units. (Source code for the `Student` example is below)
- Suppose we want to add a `Grad` class based on the `Student` class -- `Grad` students are like students, but with two differences...
  * `yearsOnThesis (yot)` -- a grad has a count of the number of years worked on thesis
  - `getStress()` is different -- Grads are more stressed. Their stress is `(2* the Student stress) + yearsOnThesis`
- Code Student and Grad is below. (The student example code is also available in the hw directory.)

Student/Grad Inheritance Design Diagram

- The following is a good sort of diagram to make when thinking about an OOP inheritance design. Plan the division of responsibility between a superclass and subclass.
- ('•' = instance variable, '-' = method)



Grad **ISA** Student

- Student is largely defined by the `units` ivar
- `Grad` is everything that a `Student` is + the idea of `yearsOnThesis (yot)`
- "isa" relationship with its superclass -- `Grad` isa `Student`
- `Grad` has **all** the properties of its superclass + a few
  - `Grad` has a units ivar, like St©udent
  - `Grad` responds to `get/setUnits` and `getStress`, like `Student`
  - `Grad` has the concept of `yot`, which is beyond what `Student` has
- (As opposed to a "has-a" relationship, where one object merely holds a pointer to another.)

Simple Inheritance Client Code

```
Student s = new Student(10);
Grad g = new Grad(10, 2);       // ctor takes units and yot
s.getStress();           // (100) goes to Student.getStress()
g.getUnits();            // (10) goes to Student.getUnits() -- INHERITANCE
g.getStress();           // (202) goes to Grad.getStress() -- OVERRIDING
```

Object Never Forgets its Class

- In Java, no matter what code is being executed, the receiver object never forgets its class.
- e.g. in the above `g.getUnits()` example, the code executing against the receiver is in the `Student` class, but the receiver knows that it is a `Grad`.

Semantics of "Student s;"

- What does a declaration like `Student s;` mean in the face of inheritance?
- NO: "`s` points to a `Student` object"
- YES: "`s` points to an object that responds to all the messages that Students respond to"
- YES: "`s` points to a `Student`, or a subclass of `Student`"

OOP Pointer Substitution

- A subclass object can be used in a context that calls for a superclass object
- This works because of the ISA property -- `Grad` ISA `Student`, so `Grad` can be used in place of `Student`
- A pointer to a `Grad` object can be stored in a variable of type `Student`.
  - `Student s = new Student(10);`

```
- Grad g = new Grad(10, 0);
- s = g; // ok -- subclass may be used in place of superclass
- // what operations are allowed on s?
```

- The reverse is not allowed however

```
- Student s = new Student(10);
- Grad g = new Grad(10, 0);
- g = s;       // NO, does not compile
```

Compile-time and Run-time Types

- Every value that appears in your program has two types -- a compile-time time in the source code (CT) and a run-time type as the code runs (RT)

Compile-Time types (CT, static)

- The compile-time type system is made simply of the declared types of the variables in the source code.
- Compile time types work in the source code, long before the program actually runs. Compile time types are also known in CS as "static" types, since they don't change -- unfortunate word choice since "static" means so many other things in Java too.
- In Java, every variable and every expression has a well-defined type in the source code -- basically the type given where the variable or parameter is declared.

```
// here we declare a few variables, each with a CT type…
Student s;
Grad g;
String str;

// For any use of "s" down here, its compile-time type is Student
```

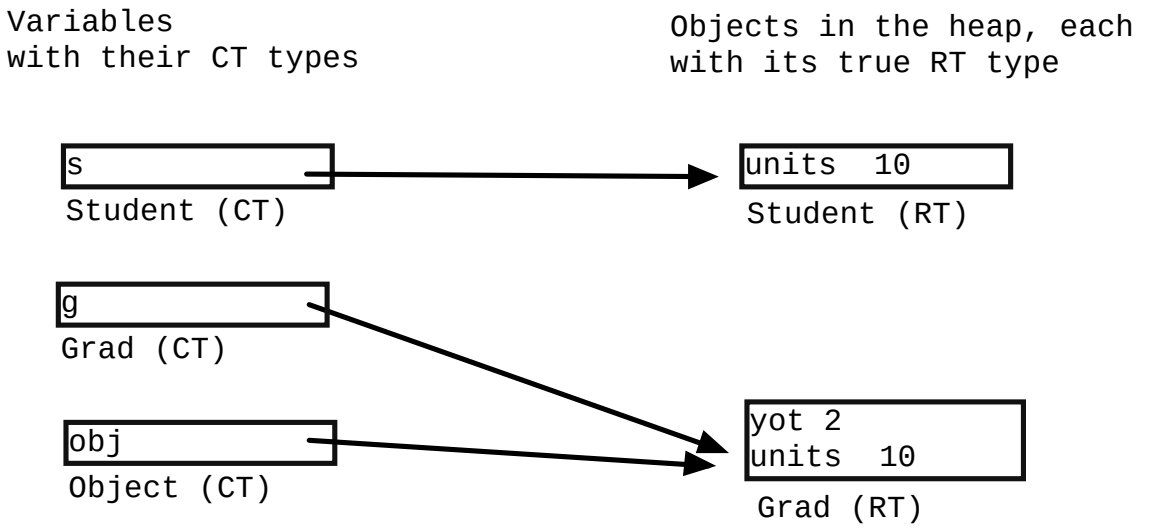
- Compile-time types are used for error-checking at compile time, making sure that various pieces of code -- variables, method, parameters -- match up with each other correctly.

Run-Time Types (RT)

- An object, created in the heap with `new` is given a true, run-time type -- the class of that object -- e.g. `new Student()` yields a `Student` object, `new Grad()` yields a `Grad`. The run-time type (the class) of an object never changes once the object is created -- it is a fundamental part of the object.

```
Student s = new Student(10);
Grad g = new Grad(10, 2);

// Store pointer to Grad in an Object variable (this is allowed)
Object obj = g;
```



- Run-time types exist for every object, obviously, at run-time. The main role of RT types is for "message resolution" -- when we send a message like `getStress()` or `equals()` to an object, the RT type of the receiver object is used to figure out which method to run.
- In particular, method overriding is resolved using the true, RT type of the receiver object. The CT type of the variable holding the pointer to the object is **not used** for message resolution in Java. (In some cases, C++ uses CT type for message resolution, and in some other cases it uses RT type, which can make interpreting source code quite tricky. Java is simple: message resolution always uses the RT type.)

Compile Time -- Error Check

- The compiler will only allow code where it is 100% clear that the receiver responds to the given message.
- The compiler's checking is all based on the compile time type system in the source code -- the declared types of ivars and other variables
- Because we can store different types of objects in a `Student` pointer, the CT and RT type systems can diverge. They do not conflict, but they differ. The CT system is looser, less precise, while the RT type system is exact.
  - e.g., with the following method, the compiler, only knows that `s` points to either a `Student` object or a `Grad` object
  - `void foo(Student s) {`
  - `  // s points to Student or Grad -- don't know for sure.`
- With this inexact information, the compiler will only allow code that works in all possible cases.
- Given a `Student s;` pointer, the compiler will allow the `getUnits()` and `getStress()` messages, since those work for both `Student` and `Grad`. However, `getYearsOnThesis()` is not allowed, since that only works if the object is a `Grad`, and that may or may not be the case with a `Student s;` pointer.

Run Time -- Message Send, ("resolution")

- In Java, the run time type system is exact -- the receiver object knows exactly what class it is.
- The run time type system is used to resolve message sends (i.e. "message-method resolution") at the time of the call.
- This is also known as "late-binding", since the exact method to run is only figured out at the last moment, depending on the class of the receiver.
- The CT type of the variable holding the pointer to the object is not used for message resolution.

Does The Right Thing -- DTRT

- A message send looks at the true class of the receiver object in the heap at run time and does the message/method resolution using that class.
- In CS jargon, we might say that it Does The Right Thing on message send -- DTRT.

Inheritance Client Code

- Same as before, but storing the Grad in a Student pointer, so the CT and RT type systems diverge, and it all still works...

```
Student s = new Student(10);
Grad g = new Grad(10, 2);
s = g; // ok
s.getStress();   // (202) ok -- goes to Grad.getStress() (overriding)
s.getUnits();    // (10) ok -- goes to Student.getUnits (inheritance)
s.getYearsOnThesis();  // NO -- does not compile (s is compile time type Student)

((Grad)s).getYearsOnThesis(); // ok, put in downcast
    // only works if s, in fact, does point to a Grad object at runtime
```

Downcast

- We may place a cast in the code to give the compiler more specific type information
- In the above example, the compiler knows that the variable s is at least `Student`, but does not know the stronger (more specific) claim of type `Grad`.
- We could put a `(Grad)` cast around s at the end of the above example like this...
  - `((Grad)s).getYearsOnThesis();`
  - This does not permanently change the s variable, the cast is just a part of that expression
- This is known as a "downcast", since it makes the more specific, stronger claim, which is in the down direction in the superclass/subclass diagram
- In Java, all casts are checked at run time, and if they are not correct, they throw a `ClassCastException`.
- In C or C++, if a cast turns out at run time to be wrong, horrible random crashing tends to result (The C++ dynamic_cast operator attempts to work around this problem)

Compile Time vs Run Time

- The compiler works with the compile-time type system, and only allows message sends that are guaranteed to work at runtime. The programmer can put in casts to edit the compile-time types of expressions.
- At run-time, the message-method resolution uses the run-time type of the receiver, not the compile-time type -- this is a feature. Some languages use compile-time types for message resolution, but that coding style is very unintuitive. C++ will use either compile-time or run-time information, depending on whether a method is declared "virtual" (run-time) or not. In Java, and most modern languages, "virtual" is the only behavior.

Inheritance Syntax and Features

Subclass Ctor

- In almost all cases, each subclass needs a constructor.
- A subclass ctor has two problems...
  - Construct the part of the object that is inherited, using the superclass ctor (e.g. `units`)
  - Construct the part of the object due to the class itself (e.g. yot)
- The ctor should take as arguments the data needed for the class itself, and also any arguments needed by the superclass ctor.
- On its first line, the subclass ctor can invoke the superclass ctor using the `super` keyword...

```
public class Grad extends Student {
    ...
    public Grad(int units, int yot) {
        super(units);          // invoke the Student ctor
        yearsOnThesis = yot;   // init our own state
    }
    ...
}
```

- If no `super` ctor is specified, the default (zero-argument) superclass ctor will be called. Some superclass constructor **must** be called -- something has to set up those superclass ivars.
- Typically, each subclass needs its own ctor complete with all its arguments spelled out. The ctor must be present, even if it just one line that calls the superclass ctor. In this sense, ctors are not inherited -- the subclass has to define its own.

this Ctor

- One ctor in a class can call another ctor in that class using "this" on the first line.
- In the following code, the default ctor calls the 2-argument ctor

```
// two-arg ctor
public Grad(int units, int yot) {
    ...
}

// default ctor calls the above ctor
public Grad() {
    this(10, 0);
}
```

- Java does not have C++ default parameter arguments, but you can get some of the same the effect by having multiple constructors or methods with various arguments that all call over to one implementation.

Grad getStress() Override -- Classic Inheritance Moment

- The `Student` class contains a basic `getStress()`.
- For grads, the definition of stress is: it's double the value for regular students + the yearsOnThesis.
- To accomplish this, override `getStress()` in the `Grad` class.
- Important: we do not copy/paste the code from the `Student` class. We call the `Student` version using `super.getStress()` (described below).

Grad getStress() Code

```
    /*
     getStress() override
     Grad stress is 2 * Student stress + yearsOnThesis.
    */
    @Override()
    public int getStress() {
        // Use super.getStress() to invoke the Student
        // version of getStress() instead of copy/pasting that
        // code down here. The whole point of inheritance
        // is not duplicating code.
        int stress = super.getStress();

        return(stress*2 + yearsOnThesis);
    }
```

Method Override

- To override a method, a subclass just defines a method with **exactly** the same prototype -- same name and arguments. With Java 5, you can provide an @Override annotation just before the method, and the compiler will check for you that it matches some superclass method.
- If the method differs by name or by its arguments, such as uppercase 'G' `GetStress()`, or `getStress(int arg),` overriding does not happen. Instead, we have defined a new method

`GetStress()` or `getStress(int arg)` that is different from `getStress()`. The compiler does not provide any warning about this by default (but @Override provides a warning).

- If your subclass method is not getting called, double check that the prototype is exactly right, or put in @Override so the compiler checks.

super.getStress()

- The `super` keyword is used in methods and ctors to refer to code up in the superclass or higher in the hierarchy.
- In the `Grad` code, the message send `super.getStress();` means...
  - Send the `getStress()` message
  - In the message/method resolution, do not use the `getStress()` method in the `Grad` class.
  - Instead, search for a matching method beginning with the superclass, Student.
- This syntax is necessary so that an override method, such as `getStress()`, can still refer to the original version up in the superclass.
- Often, an override method is not written from scratch. Instead, it is built on the superclass version.
- In C++, a method can be named at compile time by its class, e.g. Student::getStress(), but there is no equivalent in Java. "Super" does the full runtime message-method resolution, just starting the search one class higher.

Student.java

```
public class Student {
    protected int units;

    // 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.
    }

    // 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);
    }

    <rest of code snipped>
```

Grad.java

```
// Grad.java

/*
```

```
 Grad is a subclass of Student -- a simple example of subclassing.
 -adds the state of yearsOnThesis
 -overrides getStress() to provide a Grad specific version
*/
public class Grad extends Student {
    private int yearsOnThesis;

    /*
     Ctor takes an initial units and initial years on thesis.
    */
    public Grad(int units, int thesis) {
        // we use "super" to invoke the superclass ctor
        // to init that part of ourselves
        super(units);

        // init our own ivars
        yearsOnThesis = thesis;
    }


    /*
     Default ctor builds a Grad with 10 units and 0 yot.
    */
    public Grad() {
        this(10, 0);    // "this" on first line of a ctor calls
                        // a different ctor in the same class
    }


    /*
     getStress() override
     Grad stress is 2 * Student stress + yearsOnThesis.
    */
    @Override()
    public int getStress() {
        // Use super.getStress() to invoke the Student
        // version of getStress() instead of copy/pasting that
        // code down here. The whole point of inheritance
        // is not duplicating code.
        int stress = super.getStress();

        return(stress*2 + yearsOnThesis);
    }

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

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


    /*
     Example client code of Student and Grad, demonstrating
     inheritance concepts.
    */
    public static void main(String[] args) {
        Student s = new Student(13);
        Grad g = new Grad(13, 2);
        Student x = null;

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

        // Note how g responds to everything s responds to
        // with a combination of inheritance and overriding...
        s.dropUnits(3);
        g.dropUnits(3);
        System.out.println("s " + s.getStress());
        System.out.println("g " + g.getStress());

        /*
         OUTPUT...
             s 130
```

```
            g 262
            s 100
            g 202
         */

        // s.getYearsOnThesis();  // NO does not compile
        g.getYearsOnThesis(); // ok

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

        // At runtime, this goes to Grad.getStress()
        // 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.getStress();      // returns 202

        // g = x;  // NO -- does not compile,
        // substitution does not work that direction

        // x.getYearsOnThesis();  // NO, does not compile

        ((Grad)x).getYearsOnThesis(); // insert downcast
        // Ok, so long as x really does point to a Grad at runtime
    }

    /*
     Example .equals() method in the Grad class --
     true if two Grad objects have the same state.
    */
    @Override
    public boolean equals(Object other) {
        // Common optimization for == case
        if (this == other) return true;

        // Is the other object the right class?
        // (instanceof is false for null)
        if (!(other instanceof Grad)) return false;

        // Look inside the other object
        // (example of "sibling" access -- can access "private" here.
        //  Any Grad can look in any other Grad object)
        Grad grad = (Grad)other;
        return(grad.units==units && grad.yearsOnThesis==yearsOnThesis);
    }

}
/*
 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) overridden such as getStress()

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

isIrate() Example

- Suppose we define an `isIrate()` method in the `Student` class that returns true if the receiver has a stress over 100.
- (In Java, messages that do a `boolean` test on the receiver often start with the word "is" or "has".)

```
public class Student {
    ...
```

```
    public boolean isIrate() {
        return(getStress() >= 100);
        // POPS DOWN to Grad.getStress()
        // if the receiver is a Grad
    }
    ...
}
```

- Question: how does this work if we send the isIrate() message to a Grad object?

```
Student s = new Student(10);
Grad g = new Grad(10, 2);
s.isIrate(); // false
g.isIrate(); // true
```

- Short answer: the code Does The Right Thing. The `isIrate()` code is up in `Student`. However, the receiver object knows that it is a `Grad`, and on the `getStress()` message send, it correctly pops down to the `Grad` `getStress()` override.

g.isIrate() Series

- Where does the code flow go when sending `isIrate()` to a `Grad` object?
- `1. Student.isIrate()`
- `2. Grad.getStress()  // pop-down`
- `3. Student.getStress()  // the super.getStress() call in Grad.getStress`

"Pop-Down" Rule

- The receiver object knows its class
- The code being executed comes from different classes as the program proceeds
- No matter where the code is executing, the receiver knows its class and does message-to-method resolution correctly for each message send.
- e.g. Receiver is a subclass (`Grad`), executing a method up in the superclass (`Student`), a message send that `Grad` overrides will "pop-down" to the `Grad` definition (`getStress()`).
- The logic also applies with an `isIrate2(Student s)` method that takes a `Student` argument. We can call `isIrate2()` passing in a `Grad` object. Passing in a `Grad` object for a `Student` argument works because of the substitution rule.

Inheritance / Notification Style

- Here is an illustration of how all this inheritance stuff is actually used...
- Suppose there is a `Car` class with `go()`, `stop()`, and `turn()` methods
- Suppose there is a `driveToWork()` method in `Car` that sends lots of messages to the car over time: `go()`, `stop()`, `go()`, ...
- You want to create your own subclass of `Car`, but that turns differently, it beeps every time it turns, ...

```
class MyCar extends Car {
  //Override turn()...
  public void turn() {
    beep();
    super.turn();
  }
```

- You make a `MyCar` object and pass it to the system, which stores a pointer to it in a variable type `Car`.
- The system can call `driveToWork()`, which calls `go()`, `stop()`, etc. and it all works without any change for `MyCar`.
- A `turn()` message will pop down to use your `turn()` override, and then pop back up and continue using the standard `Car` code.

Applied Notification Style

- Inheritance is frequently used to integrate your code with library code -- subclass off a library class, 90% inherit the standard behavior, and 10% override a few key methods. This is the most common use of inheritance.
- This works best if the superclass code is deliberately factored into methods to support overriding. A class written the most obvious way will not just automatically support inheritance well.
- e.g. `JComponent` (the drawable class) -- subclass off `JComponent` to inherit the ability to fit in on screen, resize, and so on. Override `paintComponent()` to provide custom drawing of the component.
- e.g. Collections -- subclass off `AbstractList` to inherit `addAll()`, `toString()`, .. and lots of other methods. It pops down to use your implementation of the core functions `add()`, `iterator()`, ...
- e.g. Servlets -- inherit the standard HTTP Servlet behavior and define custom behavior in a few key method overrides.

Subclassing vs. Client Coding Strategy

- Being a client of a class is fairly easy
  - Use public ctor and methods -- read the docs which explain how to use them
  - The implementation of the class handles most of the complexity internally
- Writing a subclass off an existing superclass is not like that
- Authoring a subclass correctly often requires some understanding of the superclass implementation in order for the subclass to "fit in" correctly with the superclass --
  - Which methods should it override precisely?
  - When are those methods called, what are the pre/post conditions?
  - What variables make up the object implementation, and how are they maintained?
  - The subclass may be exposed to some or all of those details -- the relationship is not as clean as with ordinary class/client
- Things declared `private` in the superclass are not exposed to the subclass or to clients
- Things declared `protected` (ivars or methods) are for the use of subclasses (but clients cannot see them)
- Things declared `public` are available to the subclass and to everyone else as well
- Ideally, the superclass is designed and documented specifically to identify what a subclass needs to do.

Subclass Implementation Themes

- The first step in writing a subclass is understanding the superclass
- Write the subclass to fit in with the design, naming, and assumptions set out by the superclass
- Avoid duplicating code from the superclass -- use inheritance and use `super.foo()` to call the code up in the superclass as much as possible.
- i.e. avoid copy/paste code from the superclass -- probably you should be calling superclass methods instead. May require the superclass to break its code from one big method into smaller methods, so that subclasses can override just one part of the superclass behavior.

Horse/Zebra -- Key Example

- With inheritance, we define classes in terms of other classes. This can be a great shortcut if we have a family of classes with common aspects. Suppose you have a hierarchy of all the animals, except the zebra was omitted and you have been asked to add it in.
- Wrong: define the zebra from scratch
- Right: locate the `Horse` class. Introduce `Zebra` as a subclass of `Horse`
  - `Zebra` inherits 90% of its behavior (no coding required)
  - In the `Zebra` class, see how `Horse` works, and then define (override) the few things that are features of Zebras but not Horses
- This is the key feature of inheritance -- arrange classes to factor out code duplication