

Intro to Programming II

More Inheritance

Chris Brooks

Department of Computer Science
University of San Francisco

Department of Computer Science — University of San Francisco — p. 177

14-2: Inheritance Review

- Inheritance allows us to reuse existing code.
- Allows us to define a hierarchy of classes.
- *Base class* has the most general behavior
- *Derived classes* have more specific behavior.

Department of Computer Science — University of San Francisco — p. 277

14-3: Example

```
public class Person
{
    public String lastName;
    public String id;
    public void eat() { };
    public void sleep() { };
}
```

Department of Computer Science — University of San Francisco — p. 277

14-4: Example

```
public class Professor extends Person
{
    public String officeNum;
    public void teach() { };
    public void grade() { };
    public void forget() { };
}
```

Department of Computer Science — University of San Francisco — p. 477

14-5: Example

- What if we wanted to make a Student class that was a subclass of Person?
 - methods attendClass, doHomework()
- What if we also wanted to make a GradStudent that was a subclass of Student?
 - New instance variable: public Professor advisor.
- Note that Student is an *is-a* relationship, and advisor is a *has-a* relationship.

Department of Computer Science — University of San Francisco — p. 577

14-6: More examples: Java I/O

- The Java I/O package provides some nice examples of inheritance.
- InputStream is a base class
 - Provides read(), skip(), close() methods.
 - Very basic functionality.
- FileInputStream and FilterInputStream subclass InputStream
 - They *override* the behavior of InputStream for particular data sources.

Department of Computer Science — University of San Francisco — p. 677

14-7: Abstract classes

- On Monday, you did a lab in which you created a Shape class
 - This had an area() method
- You then subclassed it with Circle and Rectangle classes.
- One problem with this: you may not want users to ever create Shapes
 - You just want anything that inherits from Shape to have an area() method.

14-8: Abstract classes

- Solution: define Shape as an *abstract class*

```
public abstract class Shape
{
    public int locX;
    public int locY;
    public abstract double area(); // note: semi-colon, no method body
}
```

14-9: Abstract classes

- An abstract class is one that has one or more abstract methods
 - Can also have concrete methods.
- Classes that subclass from an abstract class must override all abstract methods.
- An abstract class therefore provides a common *interface* for a set of subclasses

14-10: Example

- How would we rewrite Shape as an abstract class?

14-11: Dynamic binding

- What if we have this situation:

```
public class A {
    public void m1() { }
    public void m2() { }
}
public class B extends A {
    public void m2() { }
}
B bex = new B();
A aex = new B();
bex.m1();
aex.m1();
bex.m2();
```

- Which methods are called?

14-12: Dynamic binding

- Java resolves this via *dynamic binding*
 - The actual type of an object is determined at runtime.
 - That object's class is searched for the corresponding method.
 - If the method doesn't exist in that class, the parent class is checked.
- What is the advantage of dynamic binding?
- What is the disadvantage?

14-13: Constructors and Inheritance

- Suppose that the Person constructor looks like this:

```
public Person(String lname) {
    lastName = lname;
}
```

- How can we still set the last name in derived class constructors?

14-14: Constructors and Inheritance

- We could do something like this:

```
public Professor(String lname, String office) {
    lastName = lname;
    officeNum = office;
}
```

- Except:

- We just had to cut and paste code. (we hate that!)
- If the base class changes, all derived classes need to change.

14-15: Constructors and Inheritance

- Instead, let's just indicate that the parent class' constructor should be called.
- We do this with `super()`

```
public Professor(String lname, String office) {
    super(lname);
    officeNum = office;
}
```
- Now we don't need to worry about what the base class' constructor does anymore.

14-16: More on `super()`

- We can also use `super` to explicitly call a superclass' method. This lets us *extend* a method, rather than overriding it.
- For example, let's say that Person has the following method:

```
/* in Person.java */
public void greet() {
    System.out.println("Nice to meet you");
}
```

14-17: More on `super()`

- We'd like for professors to do this greeting, plus a little extra. (They're long-winded.) So we can do this:

```
/* in Professor.java */
public void greet() {
    System.out.print("I do say, old chap,");
    super.greet();
}
```

14-18: Exercise

- Write a base class called Animal. Give it two instance variables (name and furColor). Give it a constructor that sets both of these.
- Give Animal a `printSelf` method that prints out the following:
 - My name is (name). My fur is (furColor).
- Now create a class called Cat that inherits from Animal. Cat should have one instance variable: age.
- Write a constructor for Cat that takes three arguments: name, furColor, and age. It should set age itself, then call `super` with the other two arguments.
- Write a method in Cat called `printSelf`. It should print "I am a cat", then call the superclass' `printSelf` method.

14-19: Next time

- Dealing with constructors
- Interfaces vs subclasses
- Polymorphism