

# *Introduction to Programming II*

## *Inheritance*

Chris Brooks

Department of Computer Science

University of San Francisco

## 13-2: Code Reuse

---

- Imagine that we've built a Person class.
- Students have first names and last names.
- They also have a printName method that looks like this:

```
public void printName() {  
    System.out.println(firstName + " " + lastName);  
}
```

## 13-3: Code Reuse

---

- Now we want to make two new classes: Students and Professors
- It turns out that Professors and Students have the same `printName` method.
- So we cut and paste from Person into Student and Professor.
- What are some problems with this approach?

## 13-4: Code Reuse

---

- Now we want to make two new classes: Students and Professors
- It turns out that Professors and Students have the same `printName` method.
- So we cut and paste from `Person` into `Student` and `Professor`.
- What are some problems with this approach?
  - Extra work - easy for us to make a mistake.
  - Making changes is harder; we have to remember all of the places we cut and pasted from.
  - What if we don't have access to the `Person` class?

## 13-5: Inheritance

---

- Inheritance lets us solve this problem more elegantly.
- One of the core features of OO programming.
- Lets us model the world in terms of *is – a* relationships.
  - We can also talk about *has – a* relationships.
- With inheritance, we define one class in terms of another.

## 13-6: Inheritance

---

```
public class Person
{
    public String firstName;
    public String lastName;
    public String id;
    public void eat() {System.out.println("Delicious!"); }
    public void sleep() {System.out.println("zzzzz"); }
}
```

## 13-7: Inheritance

---

```
public class Professor extends Person
    public String officeNum;
    public void teach() System.out.println("blah blah");
    public void grade() System.out.println("very good!");
    public void forget() System.out.println("huh??");
```

## 13-8: Inheritance

---

- Professor *inherits* all of the member variables and methods of Person for free.
- If Person changes, Professor changes automatically.
- A Professor *is – a* Person.
- This is an example of a general-to-specific relationship.

## 13-9: Terminology

---

- Professor *extends* Person
- Professor is a *subclass* of Person.
- Professor is a *child* of Person
- Professor *derives from* or *is derived from* Person.

## 13-10: Terminology

---

- Person is a *superclass* of Professor
- Person is a *parent class* of Professor.
- In Java, all objects implicitly have Object as a parent class.
  - What functionality does this provide?

## 13-11: Inheritance in action

---

```
public static void main() {
    Professor prof = new Professor();
    prof.eat();    // like all persons, prof can eat (eat is inherited)
    prof.grade();
    prof.lastName= "Smith" ; // like all persons, prof has a lastName
    prof.officeNum=123;
    System.out.println(prof.toString());    // like all Objects, prof has toString()

    Person person = new Person();
    person.eat();
    person.grade(); // error!
}
```

## 13-12: Polymorphism

---

- One of the big advantages of OO programming is that it lets us treat all objects with a common parent identically.
- Each object has the same *interface*, but its own *implementation*.
- This idea is known as *polymorphism*.
- Idea: Every object understands the same messages, but each object responds in a way that is appropriate for that object.

## 13-13: Example

---

```
public class Student extends Person {  
    public void sleep() {  
        System.out.println("Is it time for class yet?");  
    }  
}
```

- Student inherits from Person.
- The designer chose to *override* the behavior of the sleep method.

## 13-14: Example

---

```
ArrayList persons = new ArrayList();
Student s = new Student();
Professor p = new Professor();
persons.add(p);
persons.add(s);

// print out ids of all persons
Iterator it = persons.iterator();
while (it.hasNext())
{
    Person p = (Person) it.next();
    p.printName();
    p.sleep();
}
```

## 13-15: Exercise

---

- Make three classes: Shape, Circle, and Rectangle.
  - Circle and Rectangle should inherit from Shape.
- All shapes should have an x and a y that indicate where the top left corner of their bounding box goes.
- Each shape should have additional data members specific to their type. (For example, rectangle should have height and width.)
- Each shape should have an area() method. (For Shape, just return 0.0)

## 13-16: Exercise

---

- Write a class `DrawingGrid` - this will be responsible for managing shapes.
- It should contain an `ArrayList` as a data member.
- It should have two methods:
  - `totalArea` - return the sum of the areas of all shapes in the list.
  - `main` - does the following:
    - Creates a `DrawingGrid`
    - Creates two circles and two rectangles and sets their data members appropriately.
    - Adds the shapes to the `DrawingGrid`
    - Prints out the total area used.