CMSC 131
Object-Oriented Programming I

MVC, Inheritance II

Dept of Computer Science
University of Maryland College Park

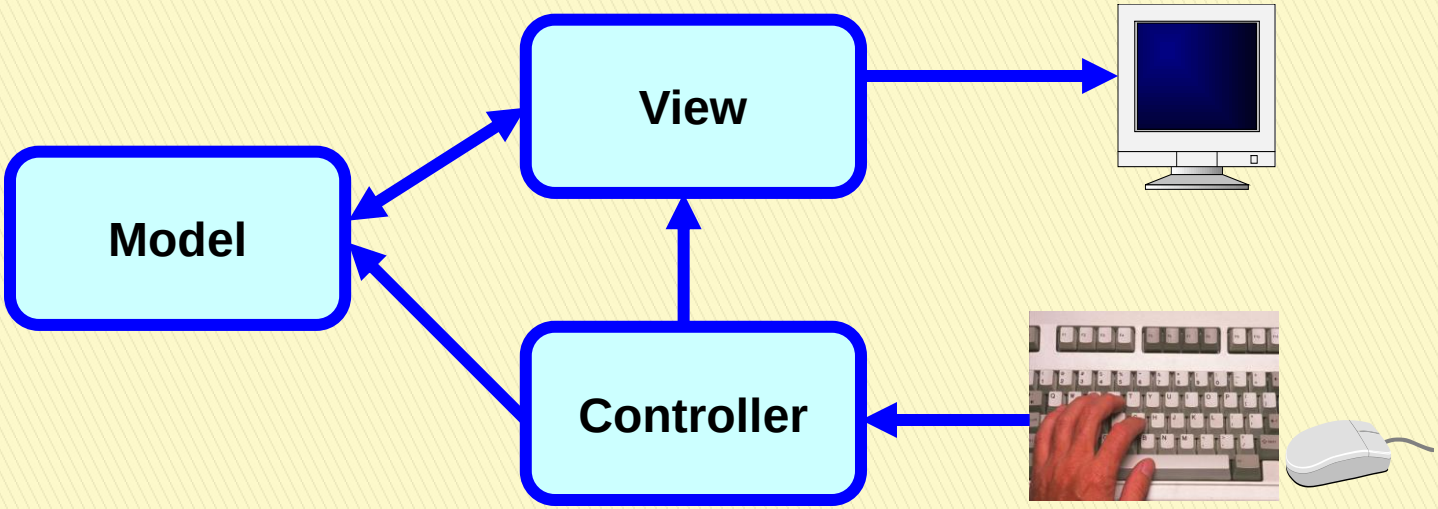
This material is based on material provided by Ben Bederson, Bonnie Dorr, Fawzi Emad, David Mount, Jan Plane

Overview

- MVC
- Inheritance

Model View Controller

- Model for GUI programming (Xerox PARC '78)
- Separates GUI into 3 components
  1. Model ⇒ application data
  2. View ⇒ visual interface
  3. Controller ⇒ user interaction



Model View Controller

- Model
  - Should perform actual work
  - Should be independent of the GUI
- Controller
  - Lets user control what work the program is doing
- View
  - Lets user see what the program is doing
  - Should not display what controller thinks is happening (base display on model, not controller)

Inheritance: Quick Recap

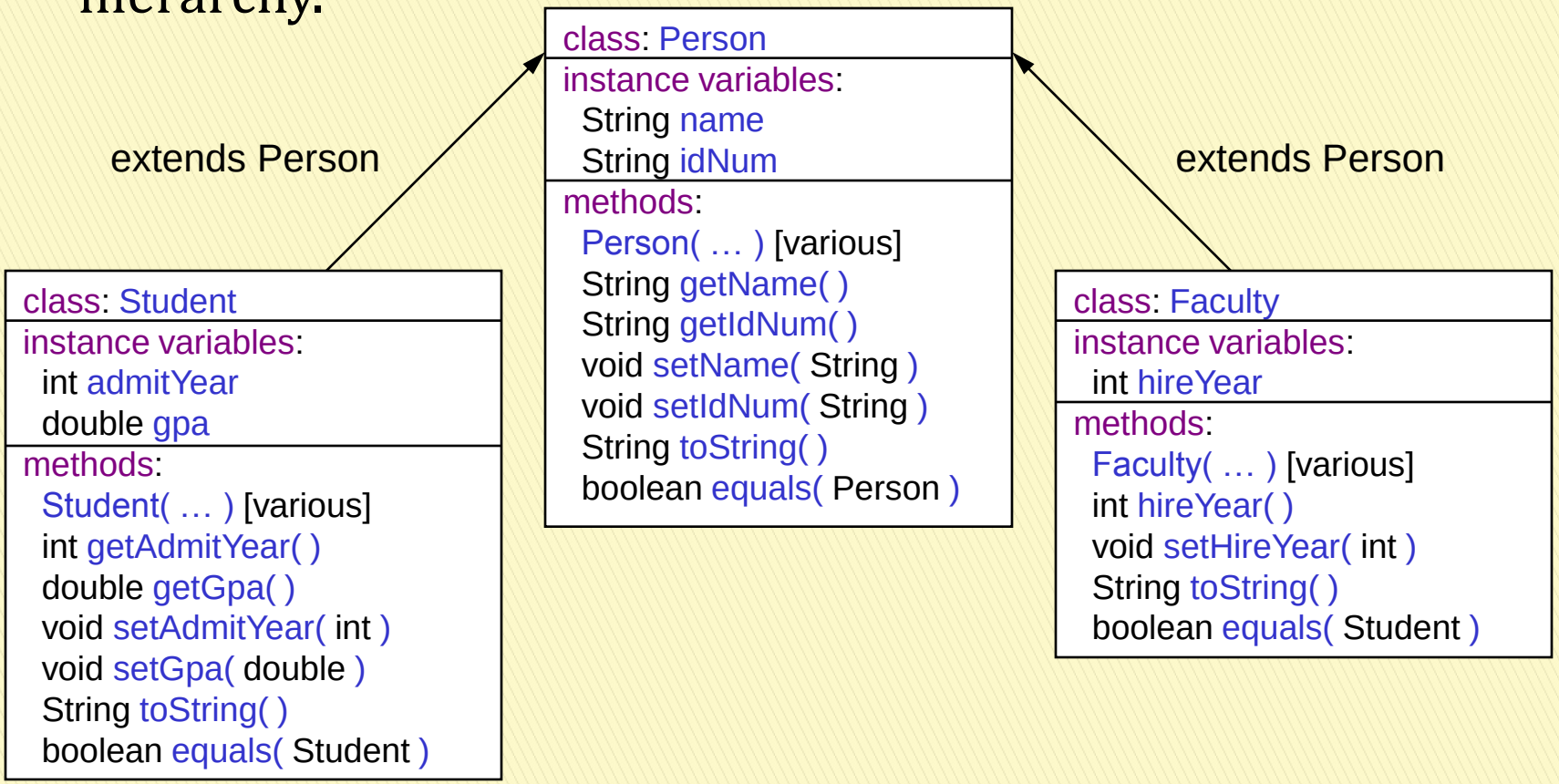
- **Recap**:
  - Inheritance is when one class (**derived class** or **subclass**) is defined from another class (the **base class** or **superclass**)
  - To derive a class D from a base class B, we use the declaration:

    public class D extends B { … }
  - The derived class **inherits** all the instance variables and the methods from the base class. It can also define its own instance variables and its own methods
  - When a derived class is initialized, it can use **super(…)** to invoke the constructor for its base class
  - A derived class can explicitly refer to entities from the base class using **super**. For example, **super.toString( )** invokes the base class's toString method
  - A reference to a derived class can be used anywhere where a reference to the base class is expected.

Remember: A Student "is a" Person.

Inheritance: Quick Recap

- **University People Example**: We defined a three-class hierarchy.



Derived Class: Faculty

```
package university;
public class Faculty extends Person {
   private int hireYear;                  // year when hired

   public Faculty() { super(); hireYear = -1; }

   public Faculty(String n, String id, int yr) {
      super(n, id);
      hireYear = yr;
   }
   public Faculty( Faculty f ) {
      this(f.getName( ), f.getIdNum( ), f.hireYear);
   }
   int getHireYear() { return hireYear; }
   void setHireYear(int yr) { hireYear = yr; }

   public String toString() {
      return super.toString() + " " + hireYear;
   }
   public boolean equals(Faculty f) {
       return super.equals(f) && hireYear == f.hireYear;
   }
}
```

Derived class: Faculty

Note the use of "this" in the copy constructor. It calls our standard constructor.

Accessors and setters

toString and equals

Overriding Methods

- **New Methods**: A derived class can define **entirely new** instance variables and new methods (e.g. hireYear and getHireYear()).
- **Overriding**: A derived class can also **redefine existing** methods.

```
public class Person {
   ...
   public String toString() { ... }
}
public class Student extends Person {
   ...
   public String toString() { ... }
}
```

The base class defines the method toString( )

The derived class can redefine this method.

```
Student bob = new Student( "Bob Goodstudent", "123-45-6789", 2004, 4.0);
System.out.println("Bob's info: " + bob.toString());
```

Since bob is of type Student, this invokes the Student toString( )

Overriding and Overloading

- Don't confuse method **overriding** with method **overloading**.

  Overriding: occurs when a derived class defines a method with the **same name** and **parameters** as the base class.

  Overloading: occurs when two or more methods have the **same name**, but have **different parameters** (different signature).

  Example:

```
public class Person {
   public void setName(String n) { name = n; }
   ...
}
public class Faculty extends Person {
   public void setName(String n) {
      super.setName("The Evil Professor " + n);
   }
   public void setName(String first, String last) {
      super.setName(first + " " + last);
   }
}
```

The base class defines a method setName( )

Overriding: Same name and parameters; different definition.

Overloading: Same name, but different parameters.

Overriding Variables: Shadowing

- **We can override methods, can we override instance variables too?**
- **Answer**: Yes, it is possible, but **not recommended**.
  - Overriding an instance variable is called **shadowing**, because it makes the base instance variables of the base class inaccessible. (We can still access it explicitly using **super.varName**).

```
public class Person {
   String name;
   // ...
}
```

```
public class Staff extends Person {
   String name;
   // ... name refers to Staff's name
}
```

  - This can be **confusing** to readers, since they may not have noticed that you redefined name. Better to just pick a new variable name.

super and this

- **super**: refers to the base class object
  - We can invoke any base class constructor using **super(...)**.
  - We can access data and methods in the base class (Person) through **super**. E.g., toString( ) and equals( ) invoke the corresponding methods from the Person base class, using **super.toString( )** and **super.equals( )**.
- **this**: refers to this object
  - We can refer to our own data and methods using "**this.**" but this usually is not needed
  - We can invoke any of our own constructors using **this(...)**. As with the super constructor, this can only be done **within a constructor**, and must be the **first statement** of the constructor. Example:

```
public Faculty( Faculty f ) {
   this(f.getName( ), f.getIdNum( ), f.hireYear);
}
```

Inheritance and Private

- **Inheritance and private members**:
  - Student objects **inherit all the private data** (name and idNum).
  - However, **private members** of the base class **cannot** be accessed directly.

Example: (Recall that name is a private member of Person.)

```
public class Student extends Person {
    ...
    public void someMethod( ) { name = "Mr. Foobar"; }  // Illegal!
    public void someMethod2( ) { setName( "Mr. Foobar" ); } // Okay
}
```

- **Why is this?** After you have gone to all the work of setting up privacy, it wouldn't be fair to allow someone to simply **extend** your class and now have access to all the **private** information.

Protected and Package Access

- The derived class cannot access private base elements. So can a base class grant any special access to its derived classes?
- **Special Access for Derived Classes**:

Protected: When a class element (instance variable or method) is declared to be **protected** (rather than public or private) it is accessible:

- To any **derived class** (and hence to all descendents), and
- To any class in the **same package**

Example:

```
protected void someMethod( ) { ... }  // has protected access
```

Package: When a class element is **not given any** access modifier (private, public, protected) it is said to have **package access**. It is accessible:

- To any class in the **same package**

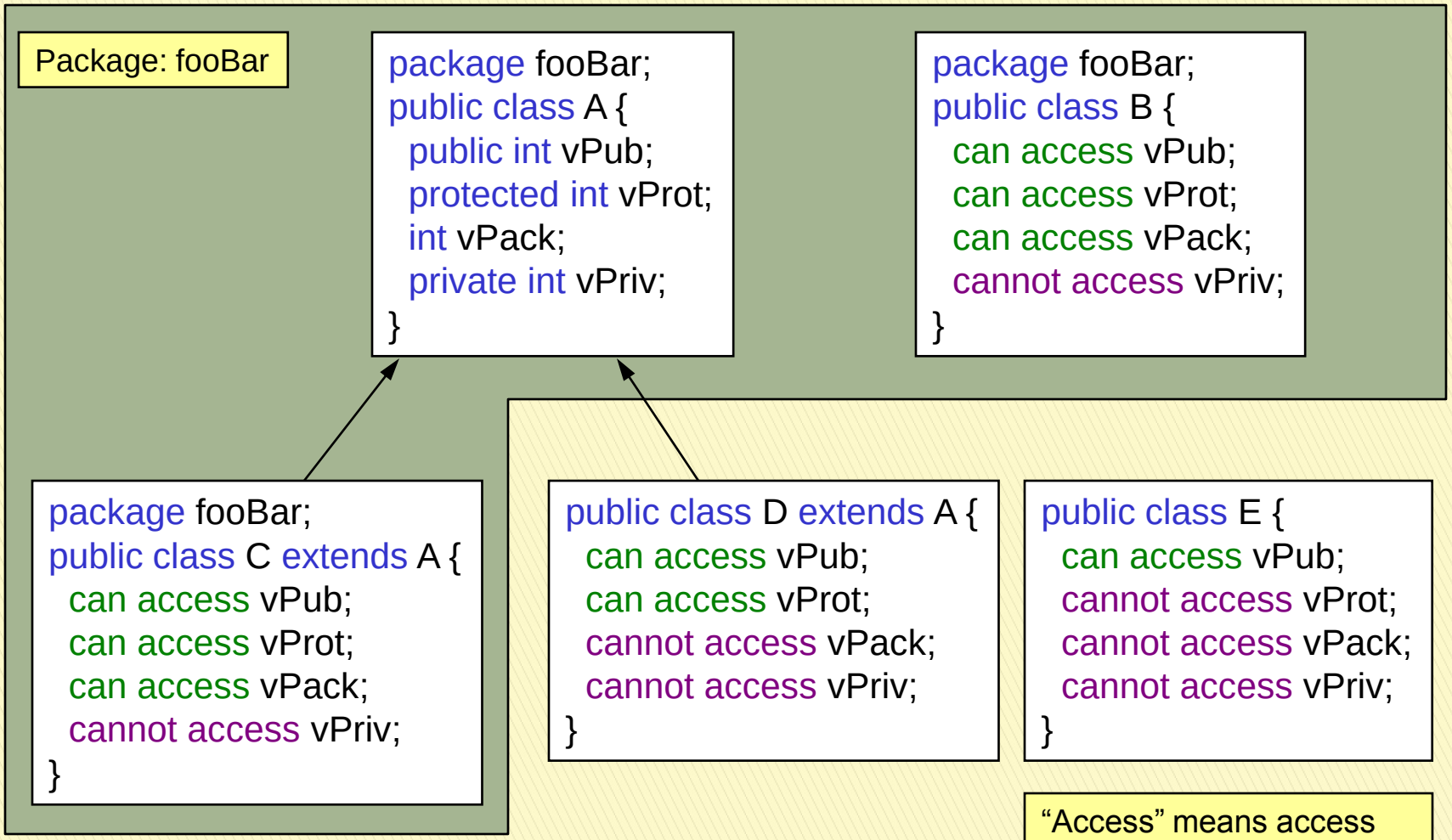
Example:

```
void someOtherMethod( ) { ... }        // has package access
```

Access to Base Class Elements

- **Which should I use?** : private, protected, package, or public?
- **Public**:
  - Methods of the object's **public interface**
  - **Constant** instance variables (static final)
- **Private**:
  - **Instance variables** (other than constants)
  - Internal **helper/utility methods** (not intended for use except in this class)
- **Protected/Package**:
  - Internal **helper/utility methods** (for use in this class and related classes)
- **Note**: Some style gurus **discourage the use of protected**. Package is safer, since any resulting trouble can be localized to the current package

Access Modifiers



"Access" means access by name, e.g.:

```
a = new A( );
a.vProt = 2;
```