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)
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:
  ```java
  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.

Given a three-class hierarchy:

- **Person**
  - Instance variables: `String name`, `String idNum`
  - Methods:
    - `Person( ... ) [various]`
    - `String getName()`
    - `String getIdNum()`
    - `void setName(String)`
    - `void setIdNum(String)`
    - `String toString()`
    - `boolean equals(Person)`

- **Student**
  - Instance variables: `int admitYear`, `double gpa`
  - Methods:
    - `Student( ... ) [various]`
    - `int getAdmitYear()`
    - `double getGpa()`
    - `void setAdmitYear(int)`
    - `void setGpa(double)`
    - `String toString()`
    - `boolean equals(Student)`

- **Faculty**
  - Instance variables: `int hireYear`
  - Methods:
    - `Faculty( ... ) [various]`
    - `int hireYear()`
    - `void setHireYear(int)`
    - `String toString()`
    - `boolean equals(Student)`

This hierarchy shows how classes can inherit from each other, allowing for shared functionality and easy extension.
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;
    }
}
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.

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

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

```
The base class defines the method toString( )
The derived class can redefine this method.
Since bob is of type Student, this invokes the Student toString( )
```
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**:

```java
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.
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:

```java
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.)

```java
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.
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:

```java
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:

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

package fooBar;
public class A {
    public int vPub;
    protected int vProt;
    int vPack;
    private int vPriv;
}

package fooBar;
public class B {
    can access vPub;
    can access vProt;
    can access vPack;
    cannot access vPriv;
}

package fooBar;
public class C extends A {
    can access vPub;
    can access vProt;
    can access vPack;
    cannot access vPriv;
}

package fooBar;
public class D extends A {
    can access vPub;
    can access vProt;
    cannot access vPack;
    cannot access vPriv;
}

package fooBar;
public class E {
    can access vPub;
    cannot access vProt;
    cannot access vPack;
    cannot access vPriv;
}

When looking at access specifiers assume two point of views: implementor and user

“Access” means access by name, e.g.: a = new A(); a.vProt = 2;