

Lecture 35: Inheritance

Last time:

1. More Inheritance

Today:

1. Project #7 assigned
2. Method overriding
3. Shadowing
4. Access issues in inheritance



Project #7 Assigned!

- Project due Sunday, 4/29 at 11 pm
- Project is **closed**
 - You must complete the project by yourself
 - Assistance can only be provided by teaching assistants (TAs) and instructors
 - You must not look at other students' code
- **FINISH NOW!**

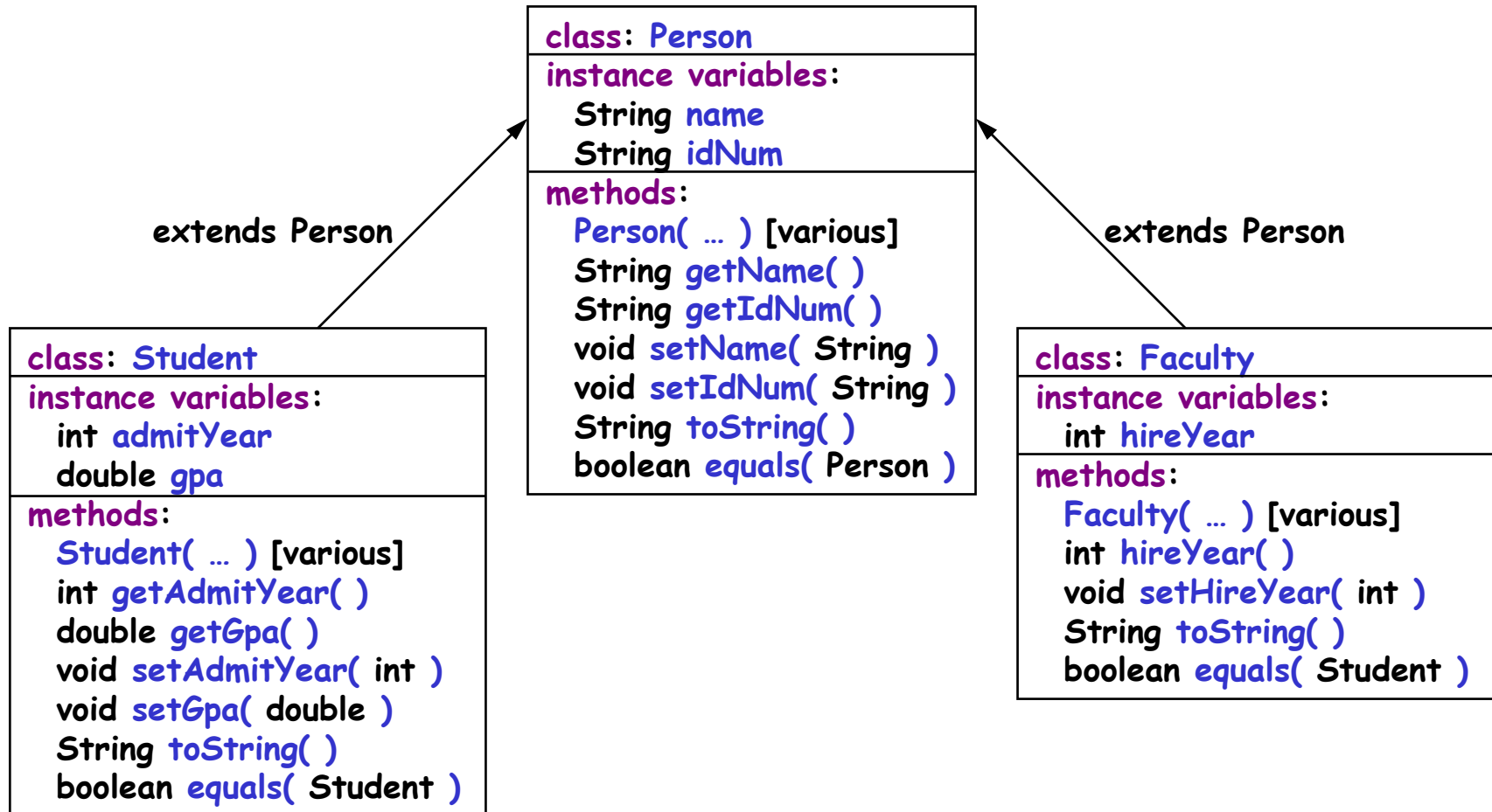
Recap

- Inheritance occurs when one class (**derived class, subclass**) is defined from another class (**base / parent class, superclass**).
- To derive a class D from a base class B, use:

```
public class D extends B { ... }
```
- Derived class inherits all instance variables, methods from base class. It can also define new instance variables, methods
- In derived-class constructor, **super**(...) can be used to invoke constructor from base class
- Derived class can explicitly refer to entities from base class using **super**, e.g. `super.toString()`
- **Polymorphism**: object in derived class can be used anywhere base class is expected (a Student **“is a”** Person!)



University Person Recap





Derived Class: Faculty

```
package university;
```

```
public class Faculty extends Person {
```

```
    private int hireYear;
```

New instance variable

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

Constructors

```
    public Faculty( String n, String id, int yr ) {  
        super(n, id);  
        hireYear = yr;  
    }
```

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

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

```
    int getHireYear( ) { return hireYear; }
```

```
    void setHireYear( int yr ) { hireYear = yr; }
```

Accessors and setters

```
    public String toString( ) {  
        return super.toString( ) + " " + hireYear;  
    }
```

toString and equals

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

```
}
```

Method Overriding

- A derived class can define new instance variables and methods (e.g. `hireYear` and `getHireYear()`)
- A derived class can also redefine (**override**) existing methods

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

Overrides base-class definition of 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 Student, Student toString used

Overriding vs. Overloading

- **Overriding**: a derived class defines a method with same name, parameters as base class
- **Overloading**: two or more methods have the same name, but different parameters
- 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 );
    }
}
```

Base class setName()

Overriding

Overloading

Shadowing

- Can we override instance variables just like methods?
- Yes, but be careful!
 - Overriding instance variable is called **shadowing**
 - Shadowing hides instance variables of base class (can still access them using `super.varName` in subclass, but not in “outside world”)

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

- Confusing! Better to pick a new variable name

super VS. this

- **super**: refers to the base class
 - Can invoke any base class constructor using `super(...)`
 - Can access data and methods in base class (Person) via `super`
E.g., `toString()`, `equals()` invoke the corresponding methods from Person base class using `super.toString()` and `super.equals()`
- **this**: refers to current class / object
 - Can refer to own data and methods using `this` (usually unnecessary)
 - Can invoke any of own constructors using `this(...)`. Like `super`:
 - Can only be done within a constructor
 - Must be the first statement of the constructor
 - Example

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

Inheritance and private

- Student inherits all private data (name and idNum) from Person
- However, private members of base class cannot be accessed directly

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

- Why?
 - Although Student inherits from Person ...
 - ... they are **different** classes

Inheritance and protected

- Private members in classes are not (directly) available to subclasses
- Classes can use **protected** modifier to give such access
- When a class member is declared `protected` it is accessible:
 - to any derived class (and hence to all descendents), and
 - to any class in the same package
- Example:

```
public class Person {  
    ...  
    protected int birthYear;  
    ...  
}
```

```
public class Student extends Person {  
    ...  
    birthYear = 1986;    // OK  
    ...  
}
```

Package access

- When a class member is not given any access modifier (`private`, `public`, `protected`) it is said to have **package access**
- Such members are accessible to any class in the same package
- Example:

```
package person;  
public class Person {  
    int computeAge () { ... } // Package access  
}
```

Which Access Policy To Use?

- **public**
 - Methods in the public interface—these become part of the API.
 - Constant variables (`final`, `static final`).
- **private**
 - Variables other than constants
 - Internal helper/utility methods not intended for use outside class
- **protected**
 - Helper / utility methods that may be useful in subclasses
 - Some style gurus discourage `protected`. They consider package access safer, since trouble can be localized to current package.
- **package**
 - Helper / utility methods that may be useful in rest of package

Note: Some style gurus **discourage use of protected**. Package is safer, since any resulting trouble can be localized to the current package.

Access Example

Package: fooBar

```
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;
}
```

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

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

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

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

Early vs. Late Binding

- Consider:

```
Faculty carol =  
    new Faculty("Carol Tuffteacher", "999-99-9999", 1995);  
Person p = carol;  
System.out.println( p.toString() );
```

- Which version of `toString` – `Person` or `Faculty` – is called?

- **Early (static) binding**

- `p` is declared to be of type `Person`
- Therefore, the `Person` version of `toString` is used

- **Late (dynamic) binding**

- The object to which `p` refers was created as `Faculty` object
- Therefore, the `Faculty` version of `toString` is used

- **Java uses late binding** (C++ uses early binding)

- Early binding is more efficient (decisions about method versions can be made at compile time)
- Late binding respects encapsulation (object defines its operations when it is created)

Polymorphism

- Java's **late binding** makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be **polymorphic** (meaning having many forms).
- **Example:** Create an array of various university people and print.

```
Person[ ] list = new Person[3];  
list[0] = new Person( "Col. Mustard", "000-00-0000" );  
list[1] = new Student ( "Ms. Scarlet", "111-11-1111", 1998, 3.2 );  
list[2] = new Faculty ( "Prof. Plum", "222-22-2222", 1981 );  
for ( int i = 0; i < list.length; i++ )  
    System.out.println( list[i].toString( ) )
```

Output:

```
[Col. Mustard] 000-00-0000  
[Ms. Scarlet] 111-11-1111 1998 3.2  
[Prof. Plum] 222-22-2222 1981
```

- **What type is list[i]?** It can be a reference to any object that is derived from Person. The appropriate toString will be called.

Disabling Overriding with `final`



- Sometimes you do not want to allow method overriding.
 - **Correctness**
Redefining a method for a derived class might break it
 - **Efficiency**
 - Late binding is less efficient than early binding
 - If you know that no subclass will redefine your method, you can force early binding by disabling overriding
- We can disable overriding by declaring a method to be **`final`**

Two Uses for `final`

- Symbolic constants

```
public static final int MAX_BUFFER_SIZE = 1000;
```

- Disabling of overriding

```
public class Person {  
    ...  
    public final String baseName () {  
        return "Person";  
    }  
  
public class Child extends Person {  
    ...  
    public String baseName () { ... } // Illegal!  
}
```

- Uses are related: “No changes allowed”