CMSC 131

Object-Oriented Programming I

Inheritance III

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

- Access Specifiers
- Object Class

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**

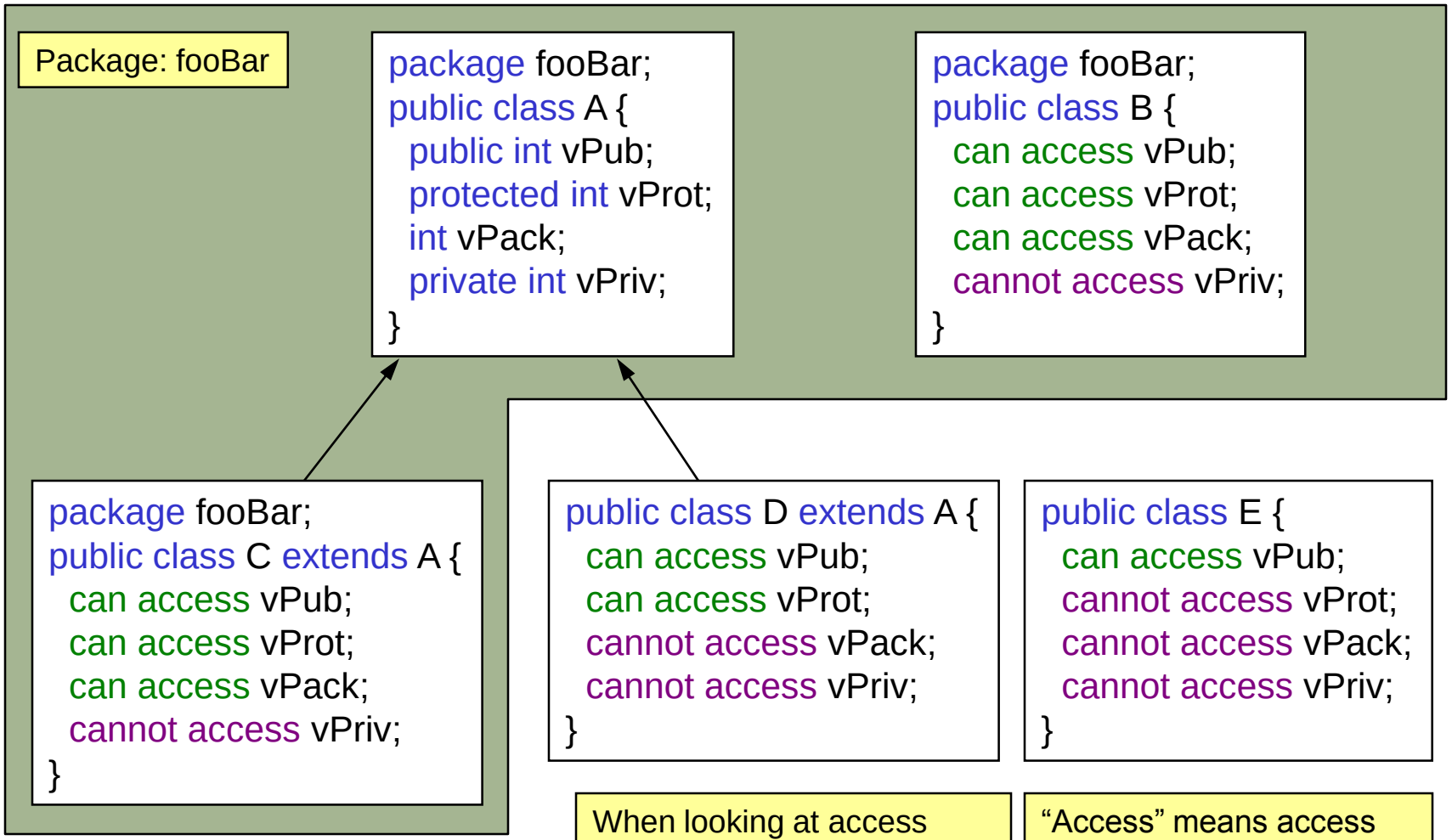
  Example:

  ```
  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



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;
```

The Class Hierarchy and Object

- **Class inheritance** defines a hierarchy:
  - **GradStudent** is a **Student**
  - **Student** is a **Person**
  - **Person** is a **???**
- There is a class at the top of the hierarchy, called **Object**. Every class is derived (either directly or indirectly) from Object
  - If a class is not explicitly derived from some class, it is **automatically derived from Object**. The following are equivalent:

    public class FooBar { … } ↔ public class FooBar extends Object { … }

    - This means that if you write a method with a parameter of type **Object**, you can call this method with an object reference of **any class**
    - Object is defined in **java.lang**, and so it is available to all programs

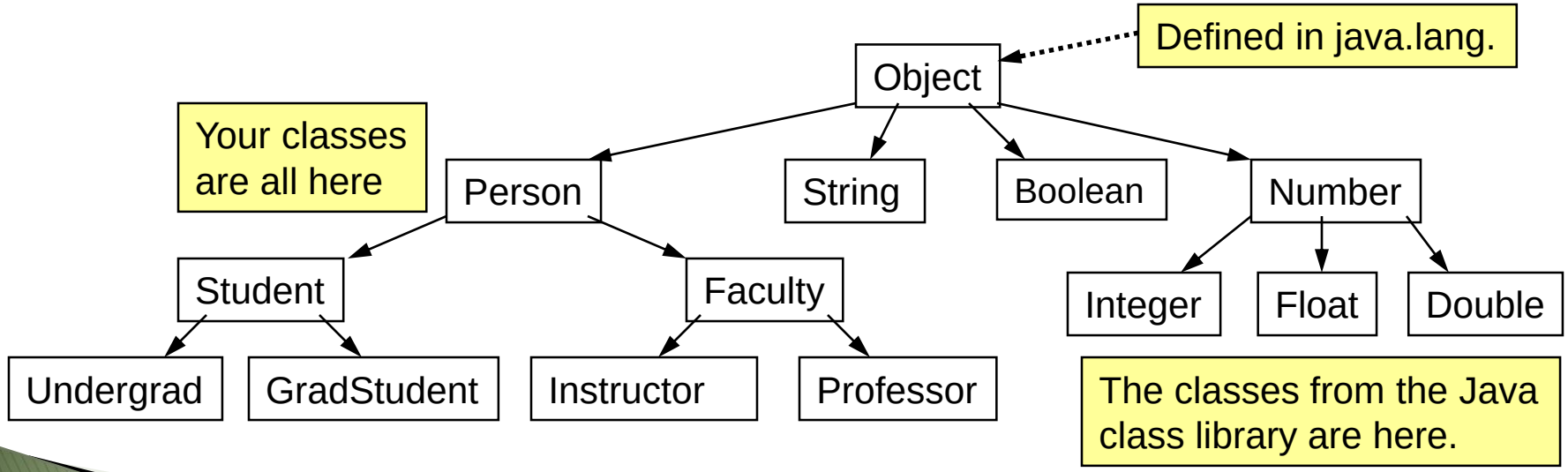
Object

- The class **Object** has no instance variables, but defines a number of methods. These include:

toString( )*: returns a String representation of this object*

equals(Object o)*: **test for equality with another object o***

- Every class you define can, and probably should, overrides these two methods with something that makes sense for your class (hashCode method is also included in the group)
- http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html



Early and Late Binding

- **Motivation**: Consider the following example:

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

- **Q**: Should this call **Person's** toString or **Faculty's** toString?
- **A**: There are good arguments for either choice:

  Early (static) binding: The variable p is **declared** to be of type **Person**. Therefore, we should call the Person's toString

  Late (dynamic) binding: The object to which p refers was **created** as a "new **Faculty**". Therefore, we should call the Faculty's toString

  Pros and cons: Early binding is more efficient, since the decision can be made at compile time. Late binding provides more flexibility

- **Java uses late binding** (by default): so Faculty toString is called
  (**Note**: C++ uses early binding by default.)

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