CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Inheritance

Department of Computer Science
University of Maryland, College Park

Inheritance and Private

- **Inheritance and private members**:
  - Student objects **inherit all the private data** (name and id)
  - 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**

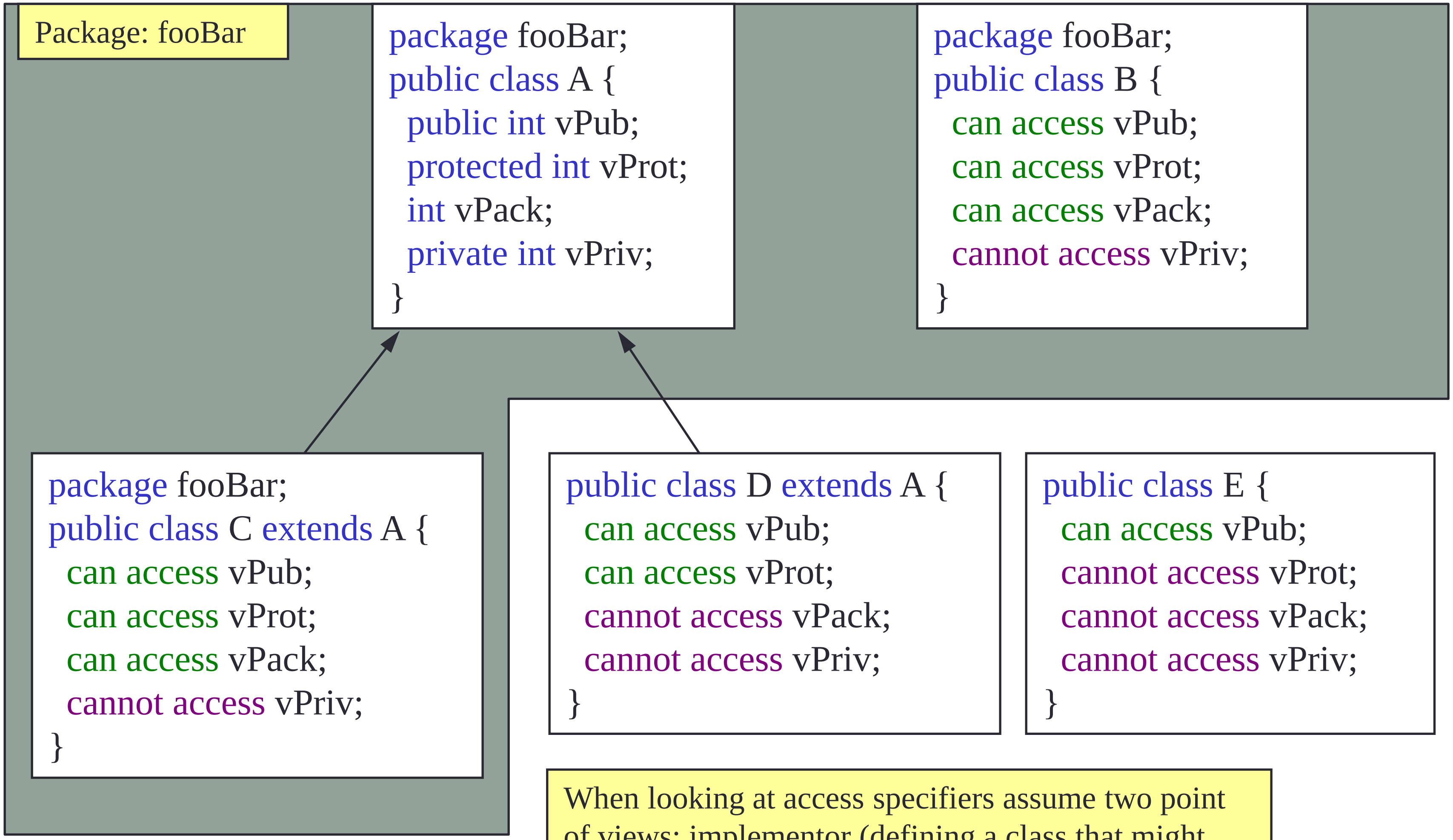
  Example:

  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



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;
}
```

When looking at access specifiers assume two point of views: implementor (defining a class that might extend another or use classes in a package) and user (e.g., some creating instances of a class from a driver class)

The Class Hierarchy and Object

- **Class inheritance tree** 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 therefore it is available to all programs

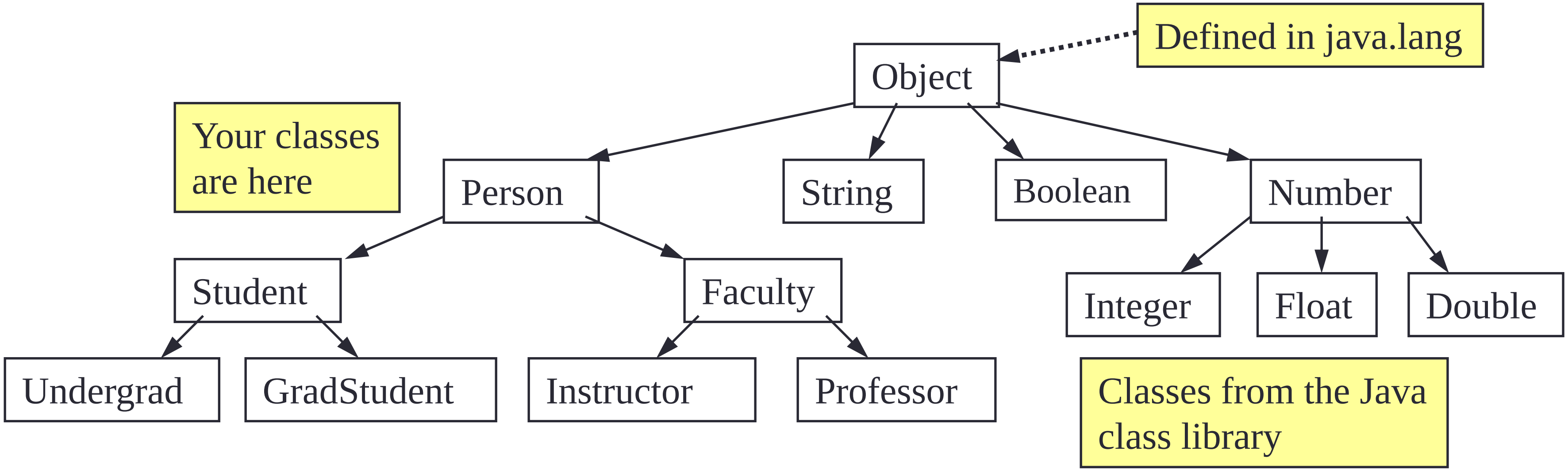
Object

- 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 override these two methods with something that makes sense for your class (hashCode method is also included in the group)
- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html



Early and Late Binding

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

```
Faculty carol = new Faculty( "Carol Tuffteacher", 458, 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)
- **Late (or dynamic) binding:** method that is called depends on an **object’s actual type**, and not the **declared type** of the referring variable

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", 10);
list[1] = new Student ("Ms. Scarlet", 20, 1998, 3.2);
list[2] = new Faculty ("Prof. Plum", 30, 1981);
for ( int i = 0; i < list.length; i++ ) {
    System.out.println( list[i].toString( ) );
}
```

Output:

```
[Col. Mustard] 10
[Ms. Scarlet] 20 1998 3.2
[Prof. Plum] 30 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
- **Example:** Polymorphism.java

getClass() and instanceof Operator

- Objects in Java can access their type information **dynamically**
- **getClass( )**: Returns a reference to an object of a class named **Class**. Instances of the class **Class** represent classes and interfaces in a running Java application. You can determine whether two objects belong to the same class by comparing the value returned by **getClass()**

```
Person bob = new Person( … );
Person ted = new Student( … );

if ( bob.getClass( ) == ted.getClass( ) )  // false (ted is really a Student)
```

- **instanceof**: You can determine whether one object is an instance of a class or derived from a class using **instanceof**. Note that it is an **operator** (!) in Java, not a method call
- Are instanceof and getClass() equivalent? No.
  - A student object is an instance of a Person, but getClass() calls will return different values for a reference to Student and a reference to a Person
- **Example:** InstanceGetClass.java

Up-casting and Down-casting

- We have already seen that we can assign a derived class reference anywhere that a base class is expected (e.g., person1 = student1)
  - **Upcasting**: Casting a reference **to a base class** (casting up the inheritance tree). This is done **automatically** and is **always** safe
  - **Downcasting**: Casting a reference **to a derived class**. The casting will only work if the type of actual object associated with the variable is lower in the inheritance tree (e.g., you are casting a Student to a Person and not the other way around). Before you cast, you must verify (using instanceof) that the actual type of the object allows for the downcasting. If you don't check you risk generating a **ClassCastException** at run-time
- **Example:** UpCastingDownCasting.java
- **Example:** SafeDownCasting.java
  - As elements are removed from the list, they must be **downcast** from **Person** to **Student**, but this can only be done if the object really is a Student