

Inheritance II

CMSC 202

Protected Access

- If a method or instance variable is modified by **protected** (rather than **public** or **private**), then it can be accessed *by name*.
 - Inside its own class definition
 - Inside any class derived from it
 - In the definition of any class in the same package
- The **protected** modifier provides very weak protection compared to the **private** modifier.
 - It allows direct access to any programmer who defines a suitable derived class.
 - Therefore, instance variables should normally **not** be marked **protected**.

Protected Members

- Derived classes can directly access inherited protected class members.

```
public class Vehicle {  
    protected int speed;  
}
```

```
public class Automobile extends Vehicle {  
    // class definition
```

```
    public void applyEmergencyBrake() {  
        speed = 0;  
    }
```

Direct access to a base Class' instance variable

```
    public static void main(String[] args) {  
        Automobile auto = new Automobile("GMC", "Hummer");  
        auto.speed = 100;  
    }
```

Problem: Public access to an instance variable of vehicle

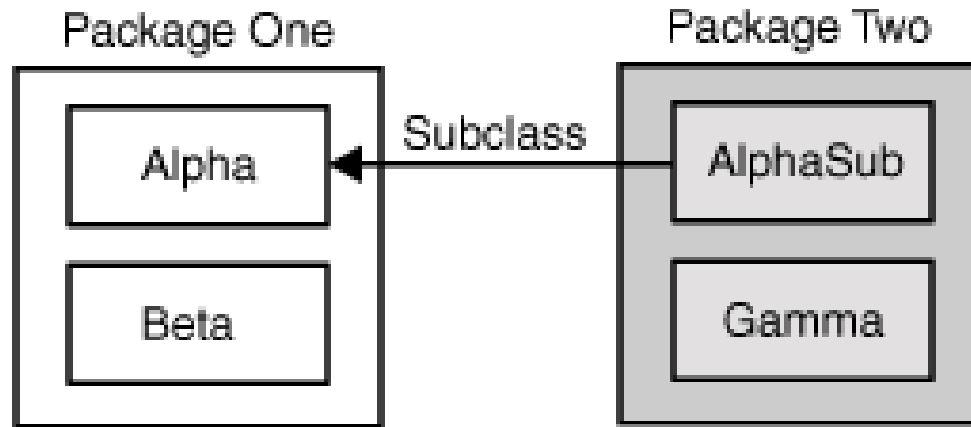
Package Access

- If you don't explicitly specify an access control modifier, Java defaults to ***package access***.
 - Also known as “package-private” or “default”
- Package visibility modifiers imply access rights that are unique to package.
 - All classes within the same package can access protected members as if they are public.
 - This may or may not be a problem.

Access Levels

Modifier	Same Class	Same Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

Visibility of Alpha's Members



Modifier	From within Alpha	From within AlphaSub	From within Beta	From within Gamma
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✗	✓	✗
private	✓	✗	✗	✗

Inherited Constructors?

A Vehicle constructor cannot be used to create Automobile objects. Why not?

We must implement a specialized constructor for Automobile. But how can the Automobile constructor initialize the private instance variables in the Vehicle class since it doesn't have direct access?

The `super` Constructor

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class.
 - In order to invoke a constructor from the base class, it uses a special syntax.

```
public DerivedClass(int p1, int p2, double p3)
{
    super(p1, p2);
    derivedClassInstanceVariable = p3;
}
```

- In the above example, `super(p1, p2);` is a call to the base class constructor.

The `super` Constructor

- Calling the base class' constructor uses the keyword `super ()`.
- A call to `super` must always be the first action taken in a constructor definition.
- An instance variable cannot be used as an argument to `super`. **Why not?**

The `super` Constructor

- If a derived class constructor does not include an invocation of `super`, then the no-argument constructor of the base class will automatically be invoked.
 - This can result in an error if the base class has not defined a no-argument constructor.
- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to `super` should always be used.

Vehicle Constructor

```
public class Vehicle {  
  
    protected int speed;  
  
    private int vin;  
    private Color color;  
    private int numOperators;  
    private int numPassangers;  
  
    private static int serialNumber = 111111;  
  
    public Vehicle() {  
        this(Color.blue, 1);  
    }  
  
    public Vehicle(Color cc, int numOperators) {  
        vin = serialNumber++;  
        color = cc;  
        this.numOperators = numOperators;  
        numPassengers = 0;  
    }  
}
```

Automobile Constructor

```
public class Automobile extends Vehicle {
    // instance variables local to the derived class extends
    private String make;
    private String model;
    private boolean locked;

    // Note we have not taken care to implement any class
    // invariant checking. However, each class should validate
    // its own state.
    public Automobile(String make, String model, Color color,
                      int numOperators) {
        // calling which constructor of vehicle?
        super(color, numOperators);

        this.make = make;
        this.model = model;
        this.locked = false;
    }

    public Automobile() {
        this("Mazda", "CX-9", Color.RED, 1);
    }
}
```

Access to a Redefined Base Method

- Within the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked.
 - Simply preface the method name with `super` and a dot.

```
// Automobile's toString( ) might be
public String toString( )
{
    return (super.toString() + "$" + getRate( ));
}
```

- However, using an object of the derived class outside of its class definition, there is no way to invoke the base class version of an overridden method.

You Cannot Use Multiple `super`s

- It is only valid to use `super` to invoke a method from a direct parent.
 - Repeating `super` will not invoke a method from some other ancestor class.
- For example, if the `Helicopter` class were derived from the class `Aircraft`, and the `Aircraft` class were derived from the class `Vehicle`, it would not be possible to invoke the `toString` method of the `Vehicle` class within a method of the `Helicopter` class.
 - You must use composition to accomplish that task.

```
super.super.toString() // ILLEGAL!
```

An Object of a Derived Class Has More Than One Type

- An object of a derived class has the type of the derived class, and it also has the type of the base class.
- More generally, an object of a derived class has the type of every one of its ancestor classes.
 - Therefore, an object of a derived class can be assigned to a variable of any ancestor type.

An Object of a Derived Class Has More Than One Type

- An object of a derived class can be plugged in as a parameter in place of any of its ancestor classes.
- In fact, a derived class object can be used any place that an object of any of its ancestor types can be used.
- Note, however, that this relationship does not go the other way.
 - An ancestor type can never be used in place of one of its derived types.

Base/Derived Class Summary

Assume that class D (Derived) is derived from class B (Base).

1. Every object of type D is a B, but not vice versa.
1. D is a more specialized version of B.
1. **Anywhere an object of type B can be used, an object of type D can be used just as well**, but not vice versa.

(Adapted from: *Effective C++*, 2nd edition, pg. 155)

Tip: Static Variables Are Inherited

- Static variables in a base class are inherited by any of its derived classes.
- The modifiers **public**, **private**, and **protected** have the same meaning for static variables as they do for instance variables.

The Class Object

- In Java, every class is a descendent of the class *Object*.
 - Every class has *Object* as its ancestor.
 - Every object of every class is of type *Object*, as well as being of the type of its own class.
- If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class *Object*.

The Class **Object**

- The class **Object** is in the package `java.lang`, which is always imported automatically.
- Having an **Object** class enables methods to be written with a parameter of type **Object**.
 - A parameter of type **Object** can be replaced by an object of any class whatsoever.
 - For example, some library methods accept an argument of type **Object** so that they can be used with an argument that is an object of any class.

The Class Object

- The class `Object` has some methods that every Java class inherits.
 - For example, the `equals` and `toString` methods
- Every object inherits these methods from some ancestor class.
 - Either the class `Object` itself, or a class that itself inherited these methods (ultimately) from the class `Object`
- However, these inherited methods should be overridden with definitions more appropriate to a given class.
 - Some Java library classes assume that every class has its own version of such methods.

The Right Way to Define `equals`

- Since the `equals` method is always inherited from the class `Object`, methods like the following simply overload it.

```
public boolean equals(Vehicle otherVehicle)
{ . . . }
```

- However, this method should be **overridden**, not just overloaded.

```
public boolean equals(Object otherObject)
{ . . . }
```

The Right Way to Define `equals`

- The overridden version of `equals` must meet the following conditions.
 - The parameter `otherObject` of type `Object` must be type cast to the given class (e.g., `Vehicle`) .
 - However, the new method should only do this if `otherObject` really is an object of that class, and if `otherObject` is not equal to `null` .
 - Finally, it should compare each of the instance variables of both objects .

A Better Vehicle Class Equals()

```
public boolean equals(Object otherObject) {  
    if (otherObject == null) {  
        return false;  
    }  
    if (otherObject.getClass() != this.getClass()) {  
        return false;  
    }  
    // Downcast so that we can access the instance variables  
    // and methods of the Vehicle Class  
    Vehicle v = (Vehicle) otherObject;  
    if (v.vin == this.vin) {  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

Prevent null pointer exception

Prevent class mismatch exception

Finally, check to see if the two vehicles are the same vehicle based on the state of each instance.

The `getClass ()` Method

- Every object inherits the same `getClass ()` method from the `Object` class.
 - This method is marked `final`, so it cannot be overridden.
- An invocation of `getClass ()` on an object returns a representation *only* of the class that was used with `new` to create the object.
 - The results of any two such invocations can be compared with `==` or `!=` to determine whether or not they represent the exact same class.

```
(object1.getClass () == object2.getClass ())
```