Computer Programming II Instructor: Greg Shaw

COP 3337

# More on Inheritance

### I. *Protected Access*

* *Protected* access is an intermediate level of access control between public and private
* Instance variables and methods designated as protected (via keyword **protected**) may be freely accessed by methods of the class containing the declaration and its subclasses, as well as all classes in the same package, but can not be accessed by other classes
* Some programmers like protected access because it enables methods of subclasses to freely access inherited instance variables. However, this is regarded as a bad idea for two reasons:

1. The author of the superclass has no control over authors of subclasses, so any subclass methods can corrupt the superclass data
2. Classes with **protected** instance variables are hard to modify. If the author of the superclass wants to change the implementation, she cannot, because some subclasses may have methods that directly access those protected fields. So protected access violates *encapsulation*

(For the same reasons, *package* access is also a bad idea for instance variables, although OK for methods)

* Recall that in CH's *BankAccount* hierarchy, instance variable *balance* was declared **private** in the superclass, rather than **protected.** When subclass objects needed to access their balance, they did sovia inherited superclass methods *withdraw*, *deposit*, and *getBalance*

### II. *Overridden Methods May Not Be Made "More Private"*

* If a superclass method is **public**, it must be overridden as **public** in subclasses and cannot be overridden as **private** or **protected** or use *package* access. Similarly, superclass methods declared **protected** or using *package* access may not be overridden with "more private" access.
* The reason the compiler enforces this is because the apparent added security is just an illusion, since subclasses can still call the superclass version of the method
* It is a common error is to forget the **public** specifier on overridden methods, thereby accidentally using *package* access.

### *Final* *Classes*

* When a class is declared *final*, it can't be "subclassed" (inherited from)
* To declare a class to be final, use the keyword **final** in the class declaration, as shown by the declaration of Java's String class:

**public final class** String

(All String objects are *immutable*. That is, they cannot be changed by any of the String class methods. Since nobody can create subclasses of the String class, we know that all String references can be copied without the possibility of mutation.)

### *Final Methods*

* Like classes, methods may also be declared *final*
* There are two reasons why we might want to declare a method to be *final*:

*Security* - as shown in this example:

**public final boolean** checkPassword (String password)

(Since the method is *final*, nobody can override it with a subclass method that simply returns **true**)

*Efficiency*

If a method cannot be overridden, it cannot be called *polymorphically*. Then, since there is no possibility of *late binding*, the compiler can generate more efficient code.

It does this by making an "inline" method call (i.e., copying the method's code right into the compiled *.class* file, and thus avoiding the overhead of method calls and returns)

* If a class is *final*, all of its methods are *implicitly* *final* (i.e., the keyword **final** does not need to appear in the method declaration). After all, if you can't create a subclass, then you can't override the superclass methods

### *Final* *Method Parameters*

This has nothing to do with inheritance, but since we're on the subject of things final...

* Parameters declared final become "read-only" (i.e., even the copy passed to the method cannot be modified in it)
* For object parameters, this means that the parameter cannot be made to “point to” a different object, but the “object pointed to” may still be modified!

**public void** doSomething (**final** Object anObject)