Inheritance

CHAPTER 7

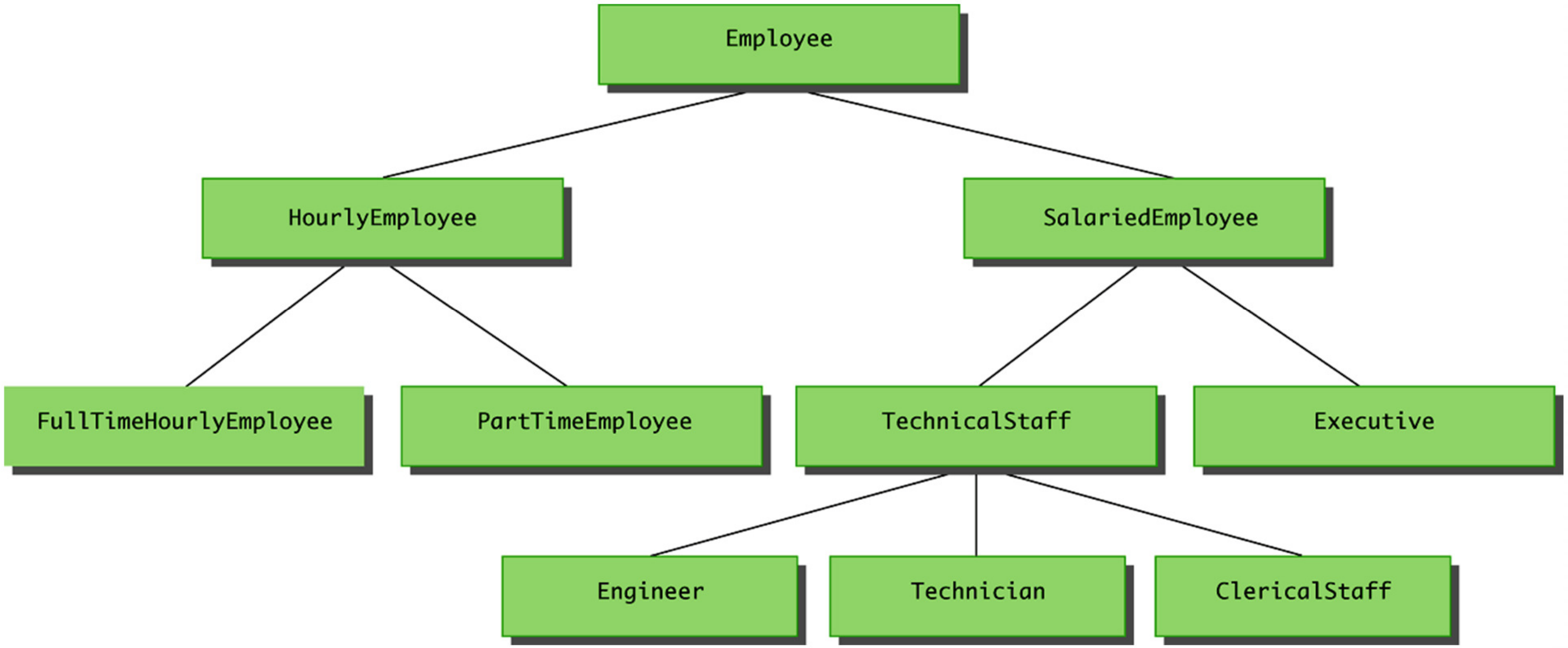
Slides prepared by Rose Williams, *Binghamton University*

Introduction to Inheritance

- *Inheritance* is one of the main techniques of object-oriented programming (OOP)
- Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
  - The specialized classes are said to *inherit* the methods and instance variables of the general class

A Class Hierarchy

Display 7.1 A Class Hierarchy



Introduction to Inheritance

- Inheritance is the process by which a new class is created from another class
  - The new class is called a ***derived/child/sub*** *class*
  - The original class is called the ***base/parent/super*** *class*
- A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well
- Inheritance is especially advantageous because it allows code to be ***reused***, without having to copy it into the definitions of the derived classes

Derived Classes

When create a constructor for the child class, use the super() to call a constructor from the parent class.
Give the super() arguments just liking using the constructor of the parent class.
But when we are copy an object of the child class, give the super() the object as the arguments. Why??
We can simply use the public method of the parent class considering the method defined in the child class.
If the variable in the parent class is private, we can only have access to these variables using the get and set method within the child class. But the child class do has these variables.
Simply print the object of the class will call the to string method automatically.

- When designing certain classes, there is often a natural hierarchy for grouping them
  - In a record-keeping program for the employees of a company, there are hourly employees and salaried employees
  - Hourly employees can be divided into full time and part time workers
  - Salaried employees can be divided into those on technical staff, and those on the executive staff

Derived Classes

- All employees share certain characteristics in common
  - All employees have a name and a hire date
  - The methods for setting and changing names and hire dates would be the same for all employees
- Some employees have specialized characteristics
  - Hourly employees are paid an hourly wage, while salaried employees are paid a fixed wage
  - The methods for calculating wages for these two different groups would be different

Derived Classes

- Eg.
  - Employee.java,
  - HourlyEmployee.java,
  - SalariedEmployee.java
- Within Java, a class called **`Employee`** can be defined that includes all employees
- This class can then be used to define classes for hourly employees and salaried employees
  - In turn, the **`HourlyEmployee`** class can be used to define a **`PartTimeHourlyEmployee`** class, and so forth

Derived Classes: Syntax

- Since an hourly employee is an employee, it is defined as a *derived* class of the class **`Employee`**
  - Employee is the *base class*
  - The phrase **`extends BaseClass`** must be added to the derived class definition:

```
public class HourlyEmployee extends Employee
```

Derived Classes: Inheritance

- Class **`Employee`** defines the instance variables **`name`** and **`hireDate`** in its class definition
- Class **`HourlyEmployee`** also has these instance variables, but they are not specified in its class definition
- Class **`HourlyEmployee`** has additional instance variables **`wageRate`** and **`hours`** that are specified in its class definition

Derived Class

- **Inherited Members**: The derived class inherits all the public methods, all the instance variables, and all the static variables from the base class

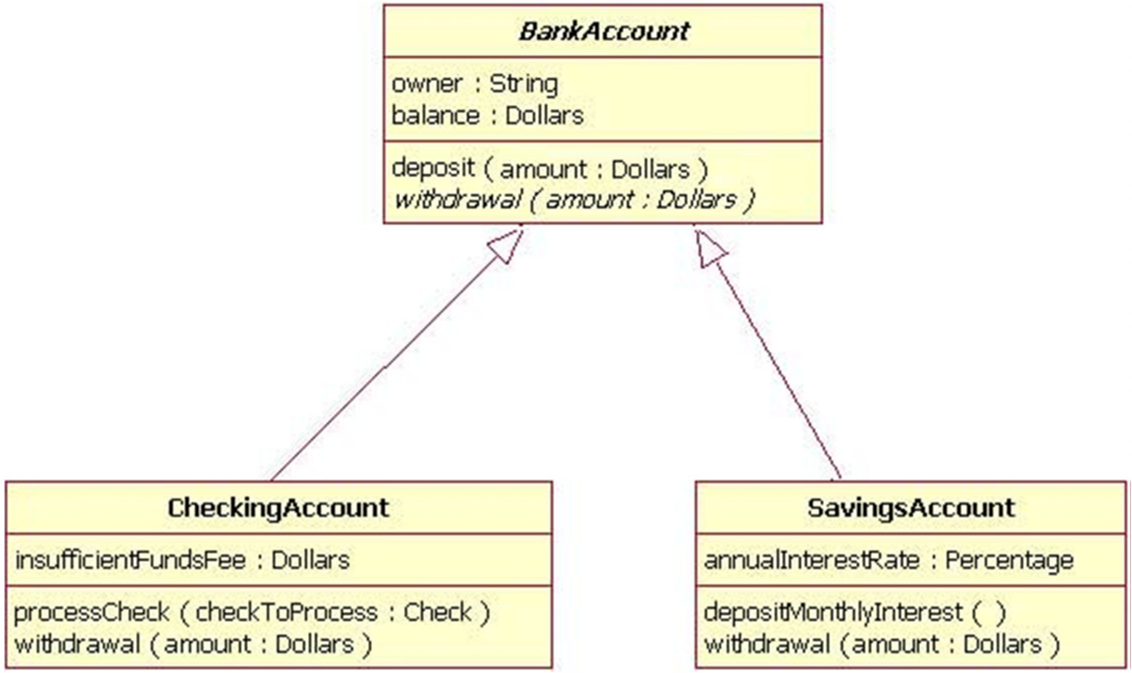
- The derived class can add more instance variables, static variables, and/or methods

- InheritanceDemo.java

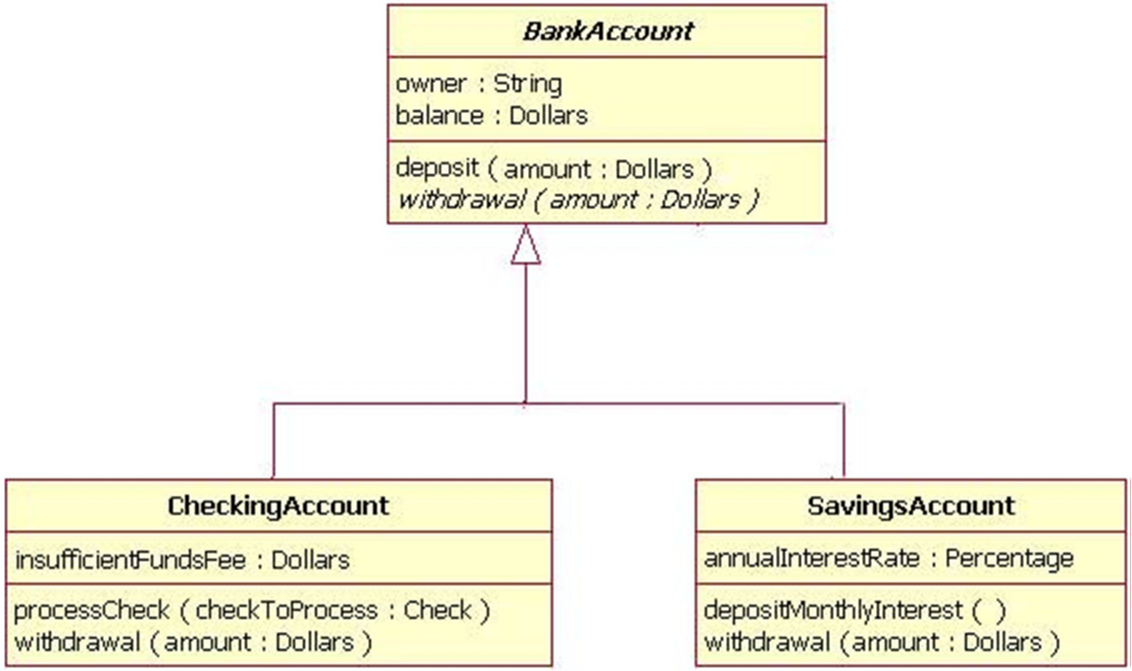
Ancestor and *descendent* Classes

- A base class is often called the ***parent** class*
  - A derived class is then called a ***child** class*
- These relationships are often extended such that a class that is a parent of a parent . . . of another class is called an ***ancestor** class*
  - If class **`A`** is an ancestor of class **`B`**, then class **`B`** can be called a ***descendent*** of class **`A`**

Derived Classes (UML)



Derived Classes (UML)



Overriding a Method Definition

- Although a derived class inherits methods from the base class, it can change or *override* an inherited method if necessary
  - In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class

Changing the Return Type of an Overridden Method

- Ordinarily, the type returned may not be changed when overriding a method
- However, if it is a class type, then the returned type may be changed to that of any descendent class of the returned type
- This is known as a *covariant return type*
  - *Covariant return types* are new in Java 5.0; they are not allowed in earlier versions of Java

Covariant Return Type

- Given the following base class:

```
public class BaseClass
{ . . .
  public Employee getSomeone(int someKey)
  . . .
```

- The following is allowed in Java 5.0:

```
public class DerivedClass extends BaseClass
{ . . .
  public HourlyEmployee getSomeone(int someKey)
  . . .
```

Changing the Access Permission of an Overridden Method

- The access permission of an overridden method can be changed from private in the base class to public (or some other more permissive access) in the derived class
- However, the access permission of an overridden method can not be changed from public in the base class to a more restricted access permission in the derived class

Changing the Access Permission of an Overridden Method

- Given the following method header in a base case:

```
private void doSomething()
```

- The following method header is valid in a derived class:

```
public void doSomething()
```

- However, the opposite is not valid
- Given the following method header in a base case:

```
public void doSomething()
```

- The following method header is <u>not</u> valid in a derived class:

```
private void doSomething()
```

Pitfall: Overriding Versus Overloading

- Do not confuse *overriding* a method in a derived class with *overloading* a method name
  - When a method is overridden, the new method definition given in the derived class has the exact same number and types of parameters as in the base class
  - When a method in a derived class has a different signature from the method in the base class, that is overloading
  - Note that when the derived class overloads the original method, it still inherits the original method from the base class as well

The `final` Modifier

- If the modifier `final` is placed before the definition of a *method*, then that method may not be redefined in a derived class
- It the modifier `final` is placed before the definition of a *class*, then that class may not be used as a base class to derive other classes

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);
  instanceVariable = p3;
}
```

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

  - Refer to HourlyEmployee.java, SalariedEmployee.java

The **`super`** Constructor: Rules

- A call to the base class constructor can never use the name of the base class, but uses the keyword **`super`** instead
- 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`**

If you don’t include an super constructor in the child class, the super constructor of the parent class with no argument automatically.
it is suggested that always define a constructor with no arguments.

The `super` Constructor: default

- 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

The `this` Constructor

- Within the definition of a constructor for a class, `this` can be used as a name for invoking another constructor in the same class
  - The same restrictions on how to use a call to `super` apply to the `this` constructor
- If it is necessary to include a call to both `super` and `this`, the call using `this` must be made first, and then the constructor that is called must call `super` as its first action

The `this` Constructor: typical usage

- Often, a no-argument constructor uses **`this`** to invoke an explicit-value constructor
  - No-argument constructor (invokes explicit-value constructor using **`this`** and default arguments):

```
public ClassName()
{
  this(argument1, argument2);
}
```

  - Explicit-value constructor (receives default values):

```
public ClassName(type1 param1, type2 param2)
{
  . . .
}
```

The `this` Constructor: example

```
public HourlyEmployee()
{
   this("No name", new Date(), 0, 0);
}
```

- The above constructor will cause the constructor with the following heading to be invoked:

```
public HourlyEmployee(String theName,
   Date theDate, double theWageRate,
   double theHours)
```

Tip: 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

Tip: An Object of a Derived Class Has More than One Type

- In fact, a derived class object can be used anyplace 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
- Eg. IsADemo.java

An Enhanced `StringTokenizer` Class (SKIP, but examable)

- Thanks to inheritance, most of the standard Java library classes can be enhanced by defining a derived class with additional methods
- For example, the `StringTokenizer` class enables all the tokens in a string to be generated one time
  - However, sometimes it would be nice to be able to cycle through the tokens a second or third time

An Enhanced `StringTokenizer` Class (SKIP)

- This can be made possible by creating a derived class:
  - For example, `EnhancedStringTokenizer` can inherit the useful behavior of `StringTokenizer`
  - It inherits the `countTokens` method unchanged
- The new behavior can be modeled by adding new methods, and/or overriding existing methods
  - A new method, `tokensSoFar`, is added
  - While an existing method, `nextToken`, is overriden

An Enhanced `StringTokenizer` Class (Part 1 of 4) (SKIP)

Display 7.7 **EnhancedStringTokenizer**

```
import java.util.StringTokenizer;

public class EnhancedStringTokenizer extends StringTokenizer
{
    private String[] a;
    private int count;

    public EnhancedStringTokenizer(String theString)
    {
        super(theString);
        a = new String[countTokens()];
        count = 0;
    }

    public EnhancedStringTokenizer(String theString, String delimiters)
    {
        super(theString, delimiters);
        a = new String[countTokens()];
        count = 0;
    }
```

The method `countTokens` *is inherited and is not overridden.*

(continued)

An Enhanced **StringTokenizer** Class (Part 2 of 4) (SKIP)

Display 7.7 **EnhancedStringTokenizer**

```java
    /**
     Returns the same value as the same method in the StringTokenizer class,
     but it also stores data for the method tokensSoFar to use.
    */
    public String nextToken()
    {
        String token = super.nextToken();
        a[count] = token;
        count++;
        return token;
    }
```

This method **nextToken** *has its definition overridden.*

super.nextTokens *is the version of* **nextToken** *defined in the base class* **StringTokenizer***. This is explained more fully in Section 7.3.*

(continued)

An Enhanced `StringTokenizer` Class (Part 3 of 4) (SKIP)

Display 7.7 **EnhancedStringTokenizer**

```
    /**
     Returns the same value as the same method in the StringTokenizer class,
     changes the delimiter set in the same way as does the same method in the
     StringTokenizer class, but it also stores data for the method tokensSoFar to use.
    */
    public String nextToken(String delimiters)
    {
        String token = super.nextToken(delimiters);
        a[count] = token;
        count++;
        return token;
    }
```

This method `nextToken` *also has its definition overridden.*

`super.nextTokens` *is the version of* `nextToken` *defined in the base class* `StringTokenizer`*.*

(continued)

An Enhanced `StringTokenizer` Class (Part 4 of 4) (SKIP)

Display 7.7 **EnhancedStringTokenizer**

```
    /**
     Returns an array of all tokens produced so far.
     Array returned has length equal to the number of tokens produced so far.
    */
    public String[] tokensSoFar()
    {
        String[] arrayToReturn = new String[count];
        for (int i = 0; i < count; i++)
            arrayToReturn[i] = a[i];
        return arrayToReturn;
    }
}
```

`tokensSoFar` *is a new method.*

Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from the Base Class

- An instance variable that is private in a base class is not accessible *by name* in the definition of a method in any other class, not even in a method definition of a derived class
  - For example, an object of the **`HourlyEmployee`** class cannot access the private instance variable **`hireDate`** by name, even though it is inherited from the **`Employee`** base class
- Instead, a private instance variable of the base class can only be accessed by the public accessor and mutator methods defined in that class
  - An object of the **`HourlyEmployee`** class can use the **`getHireDate`** or **`setHireDate`** methods to access **`hireDate`**

Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from the Base Class

- If private instance variables of a class were accessible in method definitions of a derived class, then anytime someone wanted to access a private instance variable, they would only need to create a derived class, and access it in a method of that class
  - This would allow private instance variables to be changed by mistake or in inappropriate ways (for example, by not using the base type's accessor and mutator methods only)

Pitfall: Private Methods Are Effectively Not Inherited

- The private methods of the base class are like private variables in terms of not being directly available
- However, a private method is completely unavailable, unless invoked indirectly
  - This is possible only if an object of a derived class invokes a public method of the base class that happens to invoke the private method
- This should not be a problem because private methods should just be used as helping methods
  - If a method is not just a helping method, then it should be public, not private

`Protected` and `Package` 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` and `Package` Access

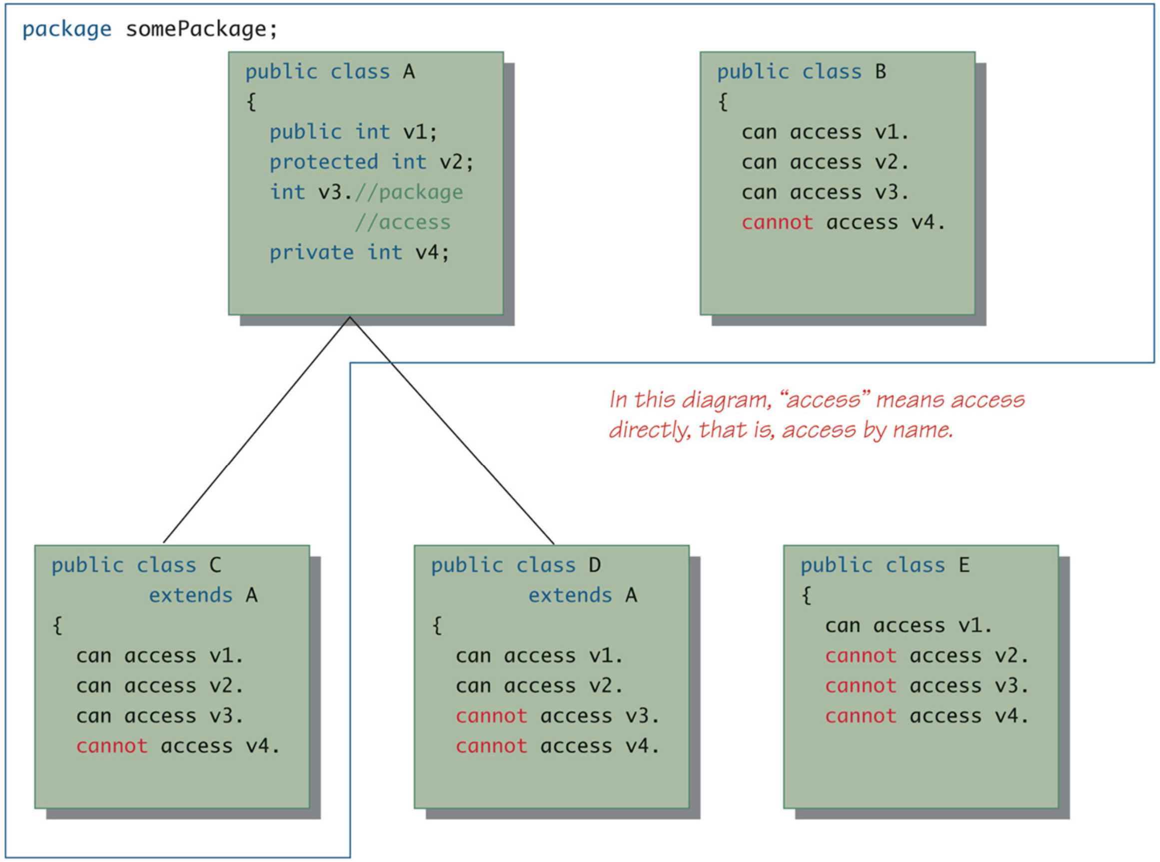
- An instance variable or method definition that is not preceded with a modifier has *package access*
  - Package access is also known as *default* or *friendly access*
- Instance variables or methods having package access can be accessed *by name* inside the definition of any class in the same package
  - However, neither can be accessed outside the package

`Protected` and `Package` Access

- Note that package access is more restricted than `protected`
  - Package access gives more control to the programmer defining the classes
  - Whoever controls the package directory (or folder) controls the package access

Access Modifiers

Display 7.9 Access Modifiers



A line from one class to another means the lower class is a derived class of the higher class.

If the instance variables are replaced by methods, the same access rules apply.

Pitfall: Forgetting About the Default Package

- When considering package access, do not forget the default package
  - All classes in the **current directory** (not belonging to some other package) belong to an unnamed package called the ***default package***
- If a class in the current directory is not in any other package, then it is in the default package
  - If an instance variable or method has package access, it can be accessed by name in the definition of any other class in the default package

Pitfall: A Restriction on Protected Access

- If a class **`B`** is derived from class **`A`**, and class **`A`** has a protected instance variable **`n`**, but the classes **`A`** and **`B`** are in *different packages*, then the following is true:
  – A method in class **`B`** can access **`n`** by name (**`n`** is inherited from class **`A`**)
  – A method in class **`B`** can create a local object of itself, which can access **`n`** by name (again, **`n`** is inherited from class **`A`**)

Pitfall: A Restriction on Protected Access

- However, if a method in class **`B`** creates an object of class **`A`**, it can not access **`n`** by name
  - A class knows about its own inherited variables and methods
  - However, it cannot directly access any instance variable or method of an ancestor class *unless they are public*
  - Therefore, **`B`** can access **`n`** whenever it is used as an instance variable of **`B`**, but **`B`** cannot access **`n`** when it is used as an instance variable of **`A`**
- This is true if **`A`** and **`B`** are *not* in the same package
  - If they were in the same package there would be no problem, because **`protected`** access implies package access

Tip: "Is a" Versus "Has a"

- A derived class demonstrates an *"is a"* relationship between it and its base class
  - Forming an "is a" relationship is one way to make a more complex class out of a simpler class
  - For example, an **`HourlyEmployee`** ***"is an"*** **`Employee`**
  - **`HourlyEmployee`** is a more complex class compared to the more general **`Employee`** class

Tip: "Is a" Versus "Has a"

- Another way to make a more complex class out of a simpler class is through a *"has a"* relationship
  - This type of relationship, called *composition*, occurs when a class contains an instance variable of a class type
  - The **`Employee`** class contains an instance variable, **`hireDate`**, of the class **`Date`**, so therefore, an **`Employee`** ***"has a"*** **`Date`**

Tip: "Is a" Versus "Has a"

- Both kinds of relationships are commonly used to create complex classes, often within the same class
  - Since `HourlyEmployee` is a derived class of `Employee`, and contains an instance variable of class `Date`, then `HourlyEmployee` ***"is an"*** `Employee` and ***"has a"*** `Date`

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`**, and package access have the same meaning for static variables as they do for instance variables

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

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

- 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 `supers`

- 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 `Employee` class were derived from the class `Person`, and the `HourlyEmployee` class were derived form the class `Employee` , it would not be possible to invoke the `toString` method of the `Person` class within a method of the `HourlyEmployee` class

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

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 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(Employee otherEmployee)
{ . . . }
```

- 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., `Employee)`
  - 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 **`equals`** Method for the Class **`Employee`**

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;
  else if(getClass( ) != otherObject.getClass( ))
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name) &&
      hireDate.equals(otherEmployee.hireDate));
  }
}
```

Tip: `getClass` Versus `instanceof`

- Many authors suggest using the **`instanceof`** operator in the definition of **`equals`**
  - Instead of the **`getClass()`** method
- The **`instanceof`** operator will return **`true`** if the object being tested is a member of the class for which it is being tested
  - However, it will return **`true`** *if it is a descendent of that class* as well
- It is possible (and especially disturbing), for the **`equals`** method to behave inconsistently given this scenario

Tip: `getClass` Versus `instanceof`

- Here is an example using the class **`Employee`**

```
. . . //excerpt from bad equals method
else if(!(OtherObject instanceof Employee))
  return false; . . .
```

- Now consider the following:

```
Employee e = new Employee("Joe", new Date());
HourlyEmployee h = new
  HourlyEmployee("Joe", new Date(),8.5, 40);
boolean testH = e.equals(h);
boolean testE = h.equals(e);
```

Tip: `getClass` Versus `instanceof`

- `testH` will be `true`, because `h` is an `Employee` with the same name and hire date as `e`
- However, `testE` will be `false`, because `e` is not an `HourlyEmployee`, and cannot be compared to `h`
- Note that this problem would not occur if the `getClass()` method were used instead, as in the previous `equals` method example

`instanceof` and `getClass`

- Both the `instanceof` operator and the `getClass()` method can be used to check the class of an object
- However, the `getClass()` method is more exact
  - The `instanceof` operator simply tests the class of an object
  - The `getClass()` method used in a test with `==` or `!=` tests if two objects *were created with* the same class

The `instanceof` Operator

- The **`instanceof`** operator checks if an object is of the type given as its second argument

  `Object instanceof ClassName`

  – This will return **`true`** if **`Object`** is of type **`ClassName`**, and otherwise return **`false`**

  – Note that this means it will return **`true`** if **`Object`** is the type of *any descendent class* of **`ClassName`**

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())
```