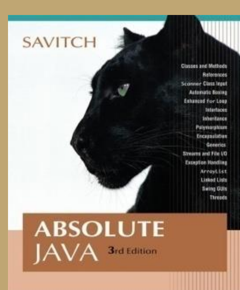
Comp 249
Programming Methodology
Chapter 7 – *Inheritance- Part B*

Prof. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



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
- Instead, a private instance variable of the base class can only be accessed by the public accessor and mutator methods defined in that class

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)
- ***See Inherit10.java***

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`**

See Inherit11.java

Protected **and** Package **Access**

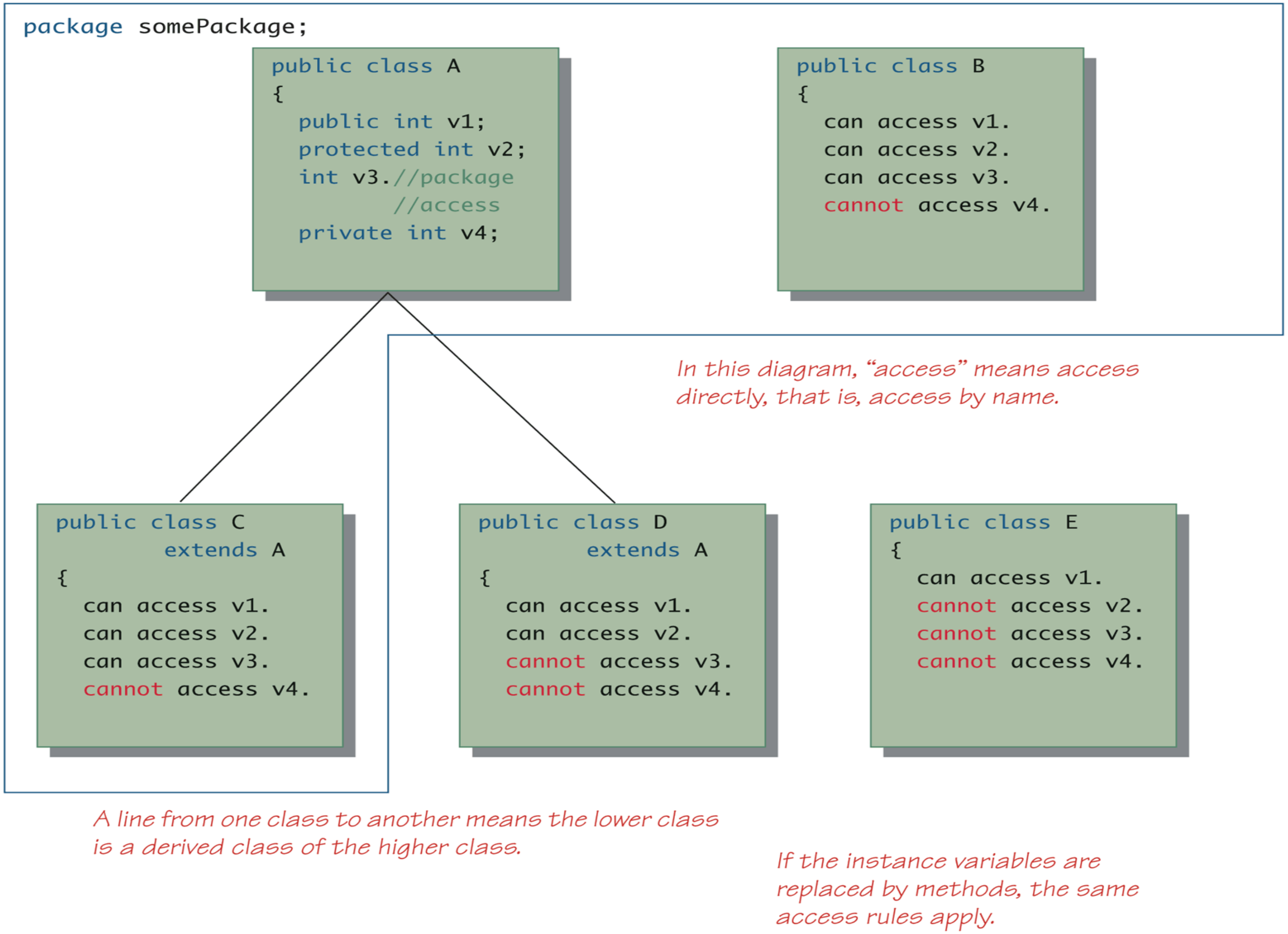
- An instance variable or method definition that is not preceded with a modifier has *package access*
  - Package access is also known as *default* or *friendly access*

- 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

- 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



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`**)

  - ***See Vehicle.java*** **&** ***Inherit12.java***

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`**

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 `super`s

- It is only valid to use **`super`** to invoke a method from a direct parent
  - Repeating **`super`** to invoke a method from some other ancestor class is illegal
- 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
- ***See Object1.java***

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, or ultimately from **`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)
{ . . . }
```

See Object2.java

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

See Object3.java

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

See Object4.java

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())
```

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

Composition

- 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

- For instance, a **`Car`** class may contain instance variables such as **`tires`**, and **`seats`**, which are objects form the of the class **`Tire`**, and so therefore, an **`Seat`** classes
- ***See Composition1.java***