Lesson 7
Inheritance

Absolute Java

Walter Savitch

Protected and Package Access

- Private instance variable or private method of the base class cannot be accessed by name, within the definition of a derived class.
- There are two classifications of instance variables and methods that allow them to be accessed by name in a derived class:
    - ➢**protected access**
    - ➢**package access**

Protected and Package Access

- **protected access** - If a method or instance variable is defined as protected then it can be accessed by name
  - ➢inside its own class definition,
  - ➢inside any class derived from it,
  - ➢and by name in the definition of any class in the same package (even if the class in the same package is not derived from it).
- However, the protected method or instance variable cannot be accessed by name in any other classes.
- Thus, if an instance variable is marked protected in the class **Parent** and the class **Child** is derived from the class Parent, then the instance variable can be accessed by name inside any method definition in the class Child.
- **However, in a class that is not in the same package as Parent and is not derived from Parent, it is as if the protected instance variable were private.**

Protected and Package Access

- For example, consider the class **HourlyEmployee** that was derived from the base class **Employee**.
- We were required to use accessor and mutator methods to manipulate the inherited instance variables in the definition of **HourlyEmployee**.
- Consider the definition of the **toString** method of the class **HourlyEmployee**:

```
public String toString()
{
  return (getName() + " " + getHireDate().toString()
    + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

Protected and Package Access

- If the private instance variables **name** and **hireDate** had been marked protected in the class **Employee**, the definition of **toString** in the derived class **HourlyEmployee** could be simplified to the following:

```
public String toString() //Legal if instance variables in
                         // Employee are marked protected
{
   return (name + " " + hireDate.toString()
   + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

Protected and Package Access

- The protected modifier provides very weak protection compared to the private modifier, because it allows direct access to any programmer who is willing to go through the bother of defining a suitable derived class.
- Many programming authorities discourage the use of the protected modifier.
- Instance variables should normally not be marked protected.
- On rare occasions, you may want to have a method marked protected.

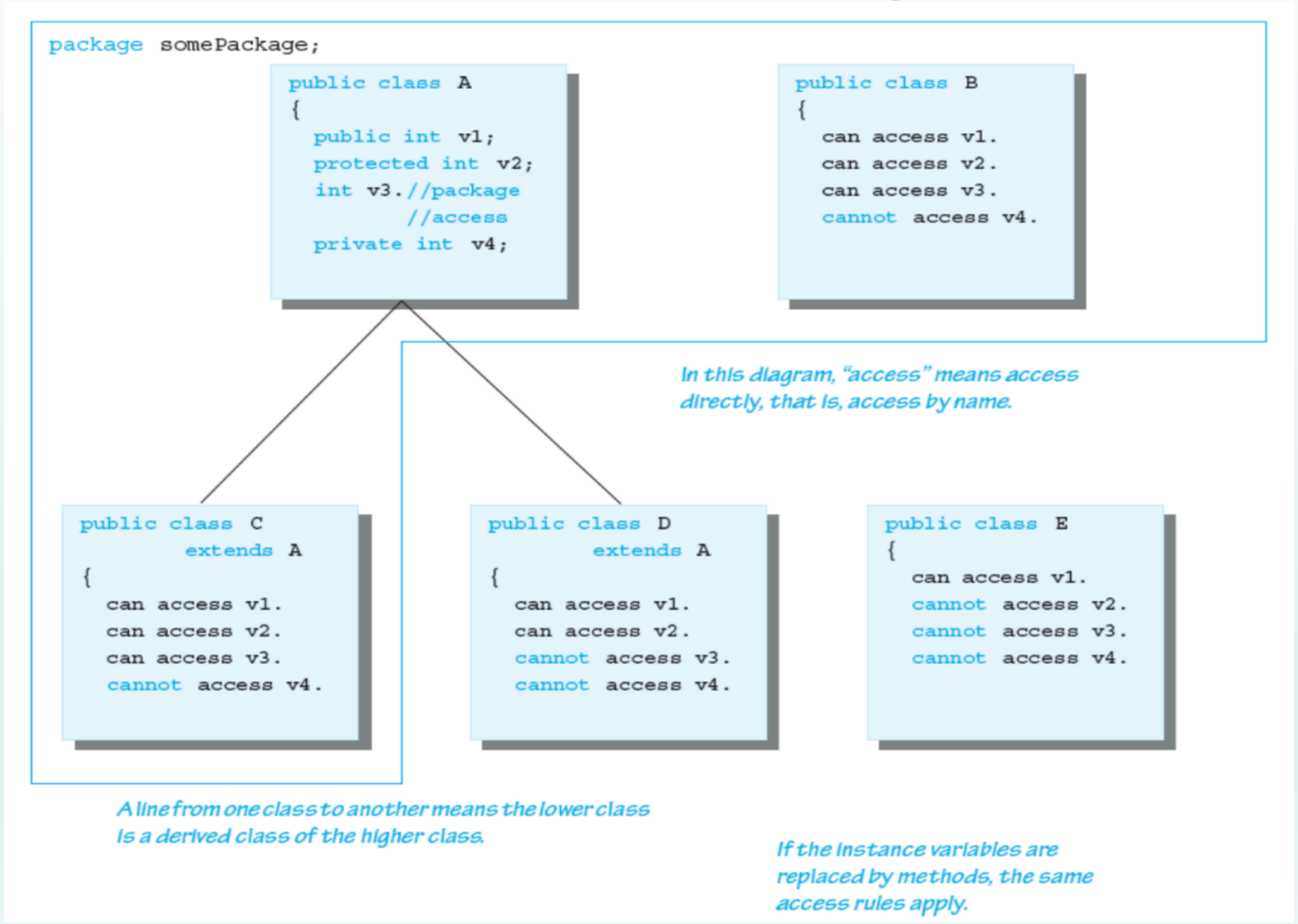
Protected and Package Access

- **Package Access:** If you do not place public, private, or protected modifiers before an instance variable or method definition, then the instance variable or method can be accessed by name inside the definition of any class in the same package but not outside of the package.
- This is called package access, default access, or friendly access.
- Use package access in situations where you have a package of cooperating classes that act as a single encapsulated unit.
- Note that package access is more restricted than protected, and that package access gives more control to the programmer defining the classes.
- If you control the package directory (folder), then you control who is allowed package access.

Protected and Package Access

- The following diagram explains who has access to members with public, private, protected, and package access.
- The diagram tells who can directly access, by name, variables that have public, private, protected, and package access.
- The same access rules apply to methods that have public, private, protected, and package access.

Protected and Package Access



```
package somePackage;
```

```
public class A
{
  public int v1;
  protected int v2;
  int v3.//package
         //access
  private int v4;
```

```
public class B
{
  can access v1.
  can access v2.
  can access v3.
  cannot access v4.
```

In this diagram, "access" means access directly, that is, access by name.

```
public class C
        extends A
{
  can access v1.
  can access v2.
  can access v3.
  cannot access v4.
```

```
public class D
        extends A
{
  can access v1.
  can access v2.
  cannot access v3.
  cannot access v4.
```

```
public class E
{
  can access v1.
  cannot access v2.
  cannot access v3.
  cannot access v4.
```

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.

Protected and Package Access

- **If you do not place any of the modifiers public, private, or protected before an instance variable or method definition, then the instance variable or method is said to have package access.**
- **Package access is also known as default access and as friendly access.**
- If an instance variable or method has package access, it can be accessed by name inside the definition of any class in the same package, but not outside of the package.

Protected and Package Access

- **Default Package:** When considering package access, do not forget the default package. Recall that all the classes in your current directory (that do not belong to some other package) belong to an unnamed package called the default package.
- So, if a class in your current directory is not in any other package, then it is in the default package.
- If an instance variable or method has package access, then that instance variable or method can be accessed by name in the definition of any other class in the default package.

Protected and Package Access

- **A Restriction on Protected Access:** Suppose class D is derived from class B, the instance variable n has protected access in class B, and the classes D and B are in different packages, so the class definitions begin as follows:

```
package one;
public class B
{
    protected int n;
    ...
}

package two;
import one.B;
public class D extends B
{
    ...
}
```

Protected and Package Access

- Then the following is a legitimate method that can appear in the definition of class D:

```
public void demo()
{
    n = 42; //n is inherited from B.
}
```

- The following is also a legitimate method definition for the derived class D:

```
public void demo2()
{
    D object = new D();
    object.n = 42; //n is inherited from B.
}
```

Protected and Package Access

- However, the following is not allowed as a method of D:

```
public void demo3()
{
    B object = new B();
    object.n = 42; //Error
}
```

- The compiler will give an error message saying that n is protected in B.
- Similar remarks apply to protected methods.

Protected and Package Access

- **A class can access its own classes' inherited variables and methods that are marked protected in the base class, but cannot directly access any such instance variables or methods of an object of the base class (or of any other derived class of the base class).**
- In the above example, n is an instance variable of the base class B and an instance variable of the derived class D.
- D can access n whenever n is used as an instance variable of D, but D cannot access n when n is used as an instance variable of B.

Protected and Package Access

- If the classes B and D are in the same package, you will not get the error message because, in Java, protected access implies package access.
- In particular, if the classes B and D are both in the default package, you will not get the error message.

Check your understanding

1. Which is more restricted, protected access or package access?
2. Suppose class D is derived from class B, the method doStuff() has protected access in class B, and the classes D and B are in different packages, so the class definitions begin as follows:

```
package one;
public class B
{
   protected void doStuff()
   {
   ...
}

package two;
import one.B;
public class D extends B
{
   ...
}
```

Check your understanding

a. Is the following a legitimate method that can appear in the definition of the class D?

```
public void demo()
{
   doStuff(); //doStuff is inherited from B.
}
```

b. Is the following a legitimate method that can appear in the definition of the class D?

```
public void demo2()
{
   D object = new D();
   object.doStuff(); //doStuff is inherited from B.

}
```

Check your understanding

c. Is the following a legitimate method that can appear in the definition of the class D?

```
public void demo3()
{
   B object = new B();
   object.doStuff();
}
```

Static Variables Are Inherited

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

The Class Object

- In Java, every class is derived from the class **Object**.
- If a class **C** has a different base class **B**, this base class is derived from **Object**, and so **C** is a derived class of **Object**.
- Thus, every object of every class is of type **Object**, as well as being of the type of its class and all its ancestor classes.
- Even classes that you define yourself without using inheritance are descendant classes of the class **Object**.
- If you do not make your class a derived class of some class, Java will automatically make it a derived class of **Object**.
- The class **Object** is in the package **java.lang**, which is always imported automatically.
- So, you do not need any import statement to make the class Object available to your code.

The Class Object

- The class **Object** does have some methods that every Java class inherits.
- For example, every class inherits the methods **equals** and **toString** from some ancestor class, either directly from the class **Object** or from a class that ultimately inherited the methods from the class **Object**.
- However, the methods **equals** and **toString** inherited from **Object** will not work correctly for almost any class you define.
- Thus, you need to override the inherited method definitions with new, more appropriate definitions.

The Class Object

toString method in Object Class

- The inherited method **toString** takes no arguments.
- The method **toString** is supposed to return all the data in an object, packaged into a string.
- However, the inherited version of **toString** is almost always useless, because it will not produce a nice string representation of the data.
- You need to override the definition of **toString** so it produces an appropriate string for the data in objects of the class being defined.

The Class Object

- For example, the following definition of **toString** could be added to the class **Student**

```
public String toString()
{
    return "Name: " + getName() + "\nStudent number: " +
studentNumber;
}
```

The Class Object

- **Equals method in Object Class**
- The heading for the method equals in our definition of the class Employee (is as follows:

```
public boolean equals(Employee otherEmployee)
```

- On the other hand, the heading for the method equals in the class Object is as follows:

```
public boolean equals(Object otherObject)
```

The Class Object

- The two equals methods have different parameter types, so we have overloaded the method equals.
- The class Employee has both of these methods named equals.
- In most situations, this will not matter.
- However, there are situations in which it does.
- Some library methods assume your class's definition of equals has the following heading, the same as in the class Object:

```
public boolean equals(Object otherObject)
```

The Class Object

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));
     }
 }
```

The Class Object

- Every object inherits the method **getClass()** from the class Object.
- The method getClass() is marked final in the class Object, so it cannot be overridden.
- **For any object o, o.getClass() returns a representation of the class used to create o.**
- For example, after the following is executed:

  o = new Employee();

  o.getClass() returns a representation Employee.

The Class Object

- Thus,

```
if (object1.getClass() == object2.getClass())
    System.out.println("Same class.");
else
    System.out.println("Not the same class.");
```

- will output "Same class." if object1 and object2 were created with the same class when they were created using new, and output "Not the same class." otherwise.

The Class Object

- ***Object* instanceof *Class_Name :*** returns true if Object is of type Class_Name; otherwise it returns false.
- **Example:**
- The following will return true if otherObject is of type Employee:

  If (otherObject instanceof Employee)