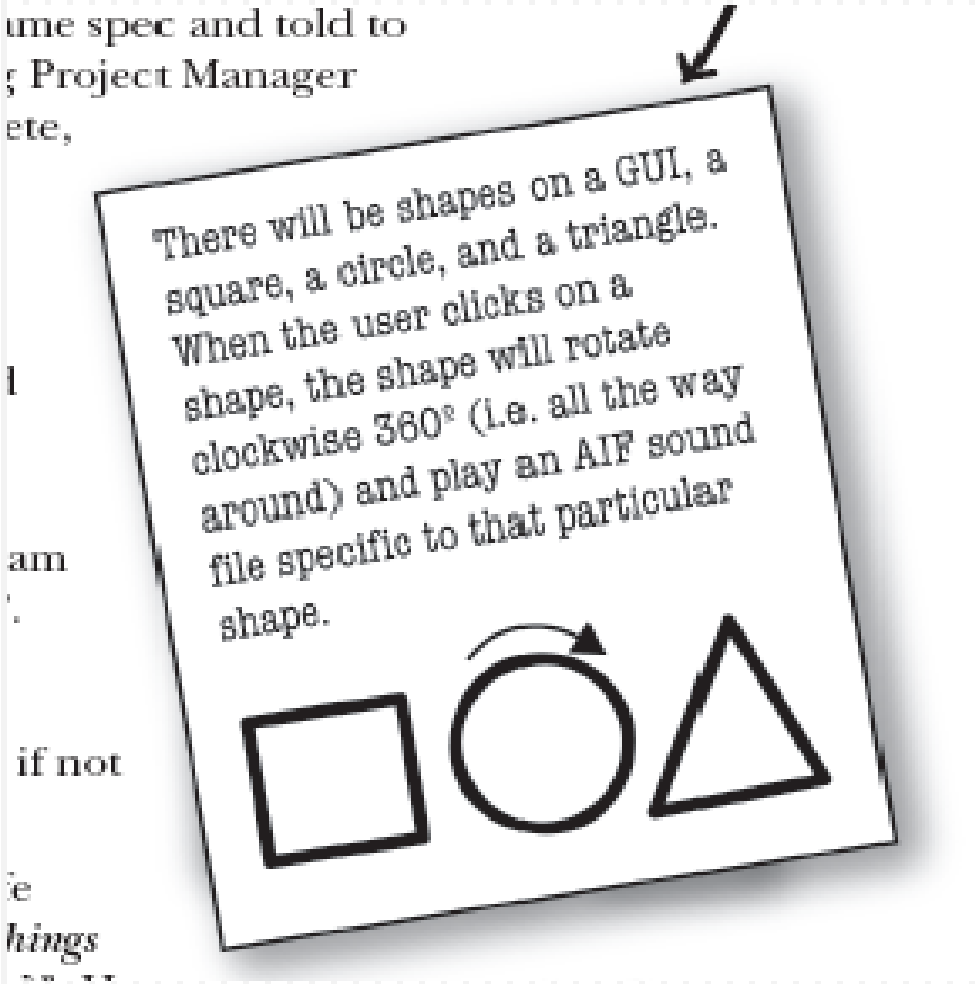
INHERITANCE

CIS*2430 (Fall 2010)

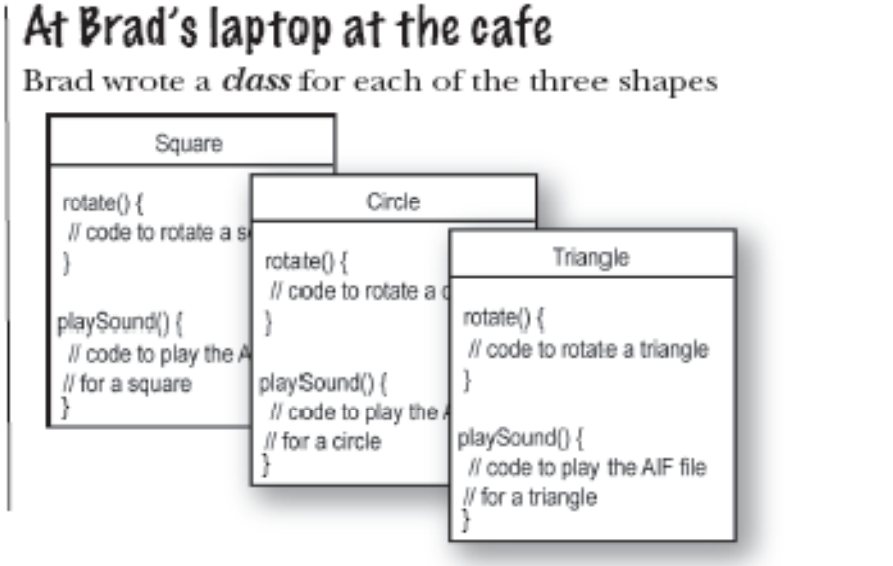
Larry vs Brad



Two Different Approaches

Larry wrote the main procedures

```
rotate(shapeNum) {
  // make the shape rotate 360°
}
playSound(shapeNum) {
  // use shapeNum to lookup which
  // AIF sound to play, and play it
}
```



Spec Changed!

arpal-tunnelled hands.

There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others, and play a .hif sound file.

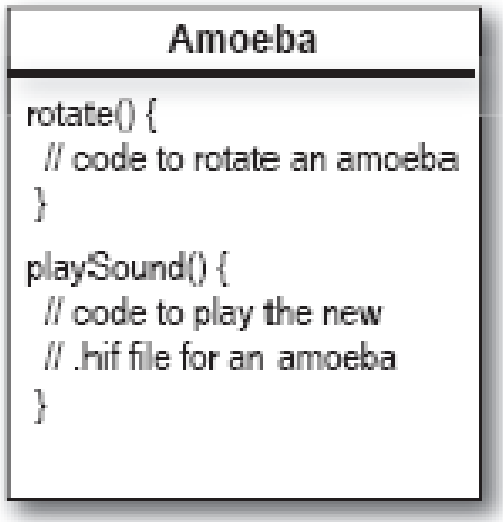
← what got added to the spec

The Two Approaches

Larry had to modify previously tested/working code

```
playSound(shapeNum) {
    // if the shape is not an amoeba,
        // use shapeNum to lookup which
        // AIF sound to play, and play it
    // else
        // play amoeba .hif sound
  }
```

Brad wrote another class



Requirements Misunderstanding

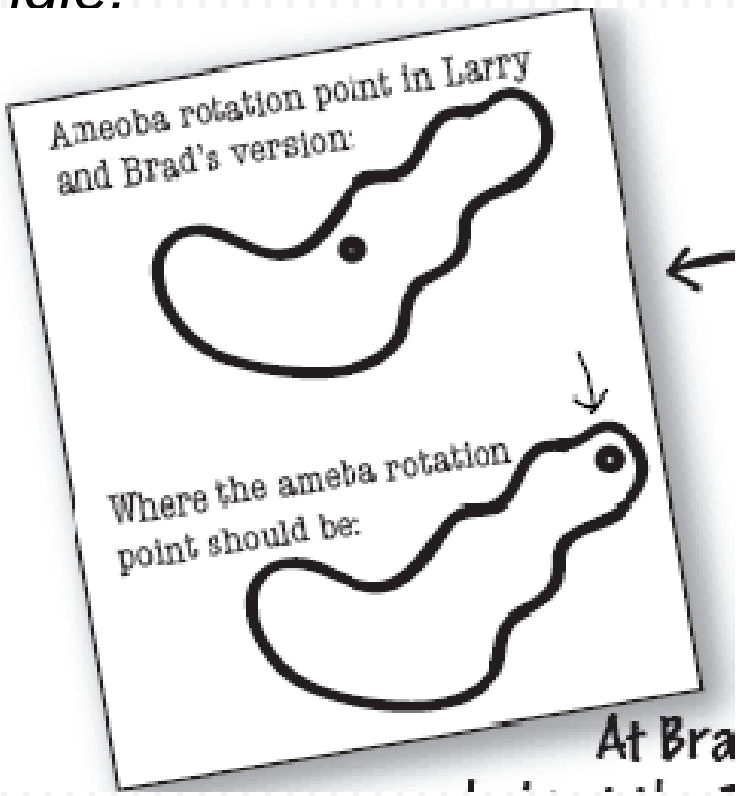
Both programmers wrote rotate code like this:

1) determine the rectangle that surrounds the shape

2) calculate the center of that rectangle.

and rotate the shape around that point.

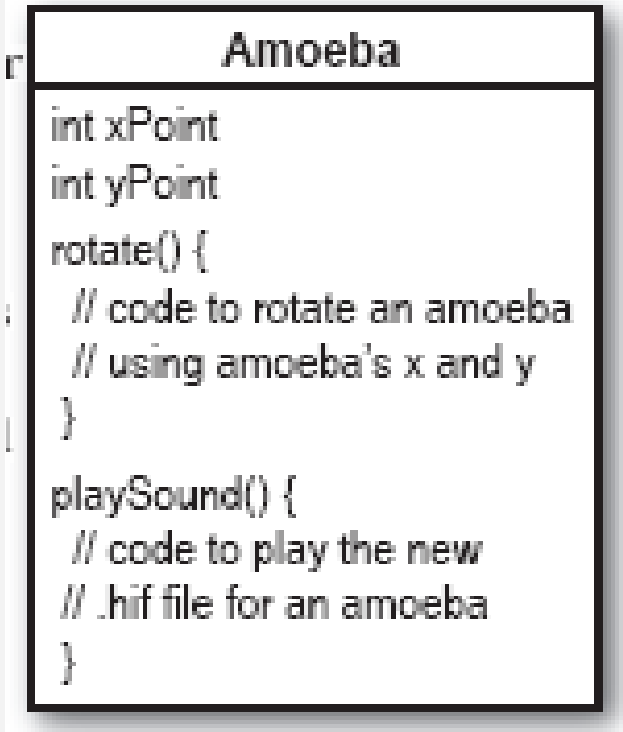
But the amoeba shape was supposed to rotate around a point on one *end*, like a clock hand.



Two Patches

Larry had to retest, recompile, fix introduced bugs
Affected the entire program

```
rotate(shapeNum, xPt, yPt) {
// if the shape is not an amoeba,
// calculate the center point
// based on a rectangle,
// then rotate
// else
// use the xPt and yPt as
// the rotation point offset
// and then rotate
}
```



Brad fixed one method in one class

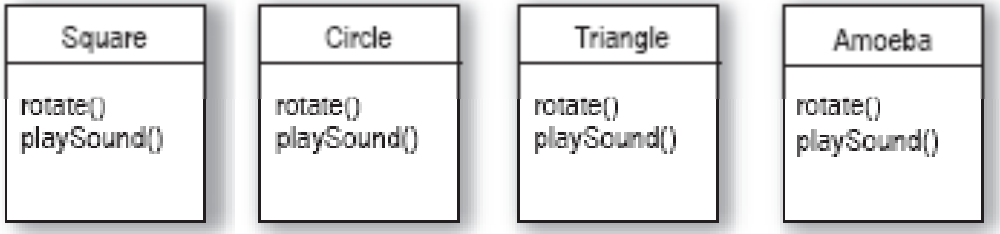
Larry Makes a Last Stand

- So, Brad the OO guy won, right?
- ***Not so fast.*** Larry found a flaw in Brad's approach.
  - **LARRY:** You've got duplicated code! The rotate procedure is in all four Shape things.
  - **BRAD:** It's a ***method***, not a *procedure*. And they're not *things*.
  - **LARRY:** Whatever. It's a stupid design. You have maintain *four* different rotate "methods". How can duplicate code ever be good?
  - **BRAD:** Oh, I guess you didn't see the final design. Let me show you how OO **inheritance** works, Larry.

Brad's Defense

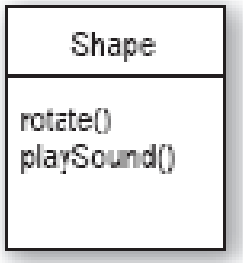
1

I looked at what all four classes have in common.

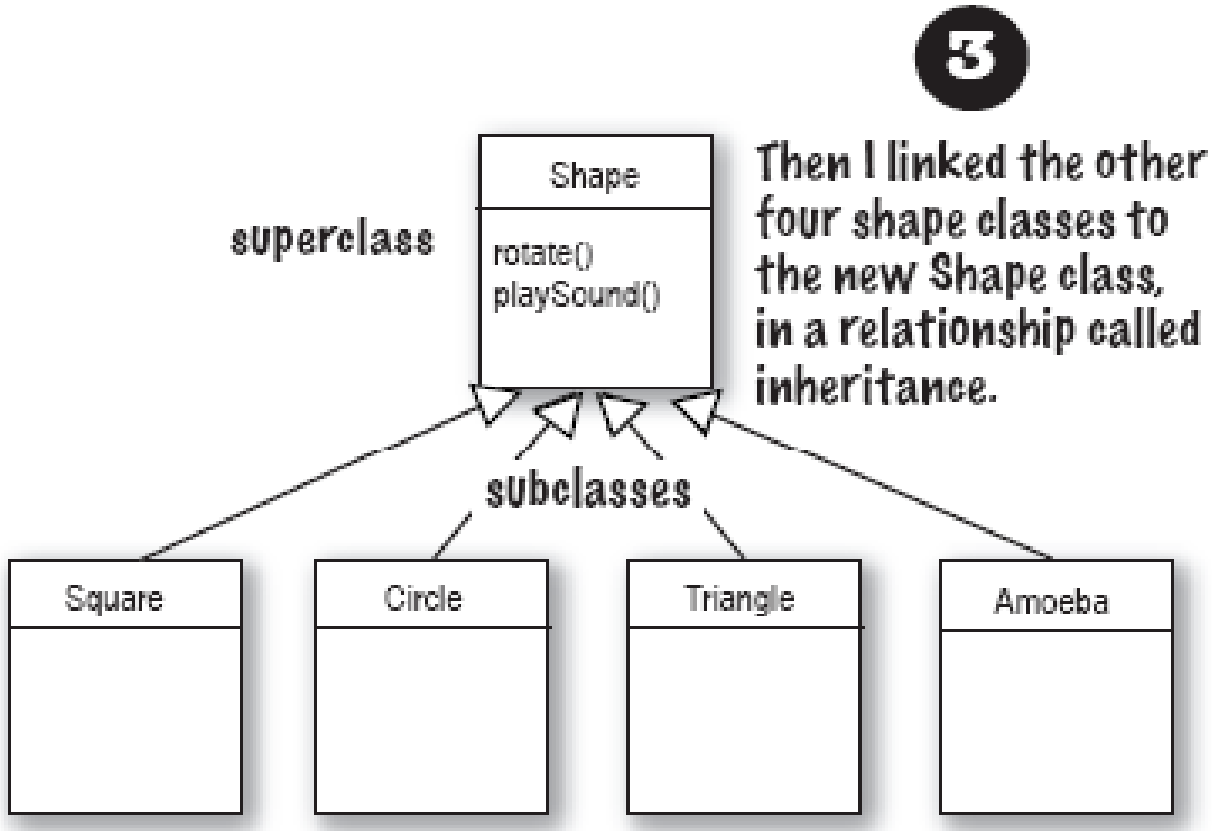


2

They're Shapes, and they all rotate and playSound. So I abstracted out the common features and put them into a new class called Shape.



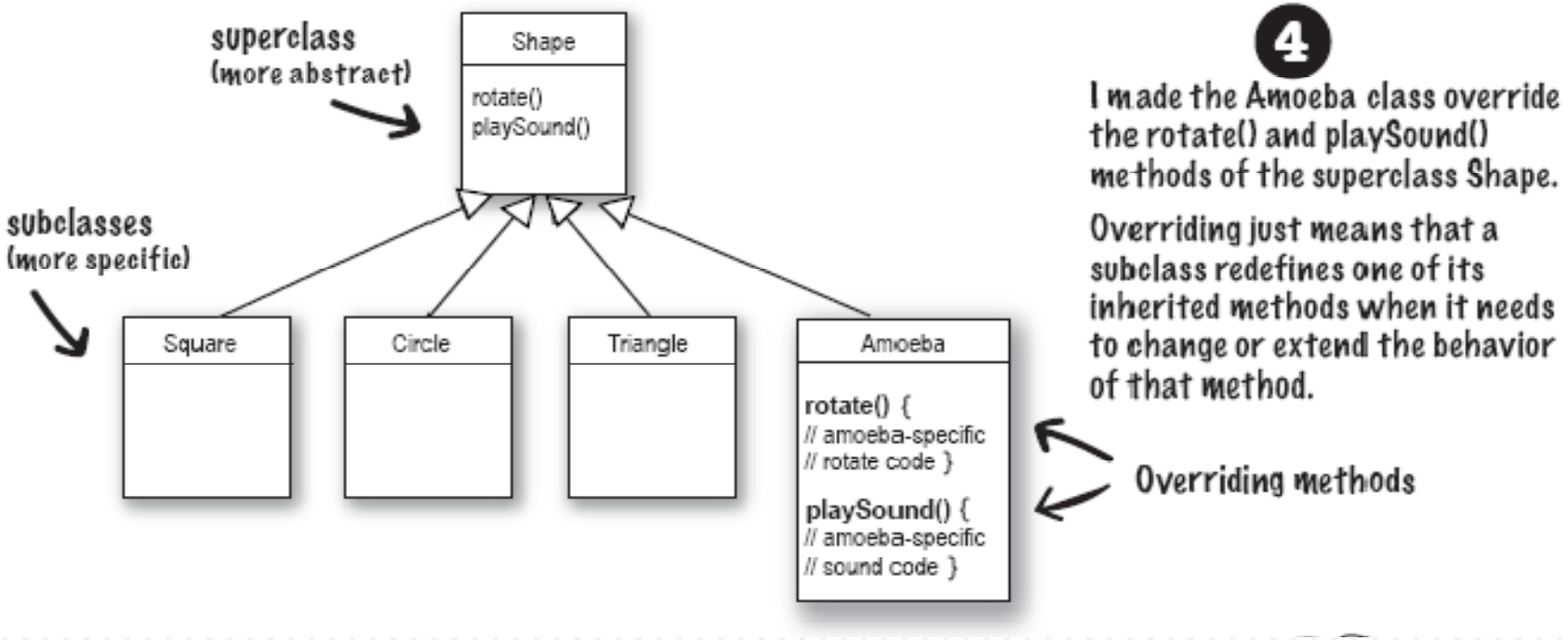
Brad's Defense



What about the Amoeba rotate()?

- **LARRY:** Wasn't that the whole problem here — that the amoeba shape had a completely different rotate and playSound procedure?
- **BRAD: Method.**
- **LARRY:** Whatever. How can amoeba do something different if it "inherits" its functionality from the Shape class?
- **BRAD:** That's the last step. The Amoeba class **overrides** the methods of the Shape class. Then at runtime, the JVM knows exactly which rotate() method to run when someone tells the Amoeba to rotate.

Brad's Defense

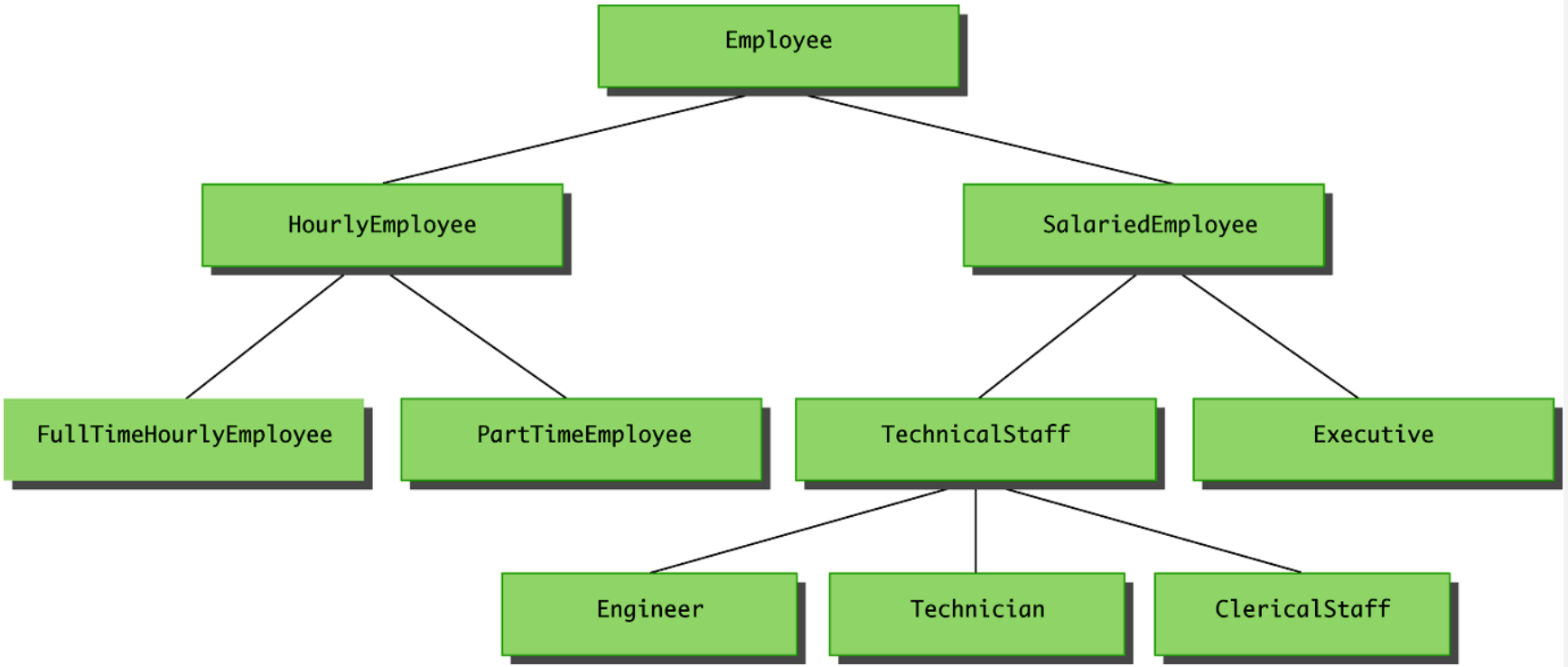


Inheritance

- *Inheritance* is one of the main techniques for object-oriented programming (OOP)
- Create a general form of a class
  - Super class or base class
- Create specialized versions of the class
  - Sub class or derived class
  - Adding instance variables and methods
- The specialized classes *inherit* the methods and instance variables of the general class
- Inheritance allows code to be *reused*

Class Hierarchy

Display 7.1 **A Class Hierarchy**



Employee Class

```
public class Employee
{
    private String name;
    private Date hireDate;

    public Employee( )
    public Employee(String theName, Date theDate)
    public Employee(Employee originalObject)

    public String getName( )
    public void setName(String newName)
    public Date getHireDate( )
    public void setHireDate(Date newDate)
    public String toString( )
    public boolean equals(Employee otherEmployee)
}
```

HourlyEmployee Class

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours;     //for the month

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

    public double getRate( )
    public double getHours( )
    public double getPay( )
    public void setHours(double hoursWorked)
    public void setRate(double newWageRate)
    public String toString( )
    public boolean equals(HourlyEmployee other)
}
```

Derived Classes

- The derived class (subclass) inherits all the public methods, all the public and private instance variables, and all the public and private static variables from the base class
  - These members from the base class are said to be inherited
  - The private methods of the base class are not inheritable
  - The derived class can add more instance variables, static variables, and instance/static methods
  - The derived class can change or override an inherited method if necessary

Overriding toString()

```
// Employee toString
public String toString( )
{
   return (name + " " + hireDate.toString( ));
}


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

Overriding the Returned Type

- 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

Overriding the Returned 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)
  . . .
}
```

Overriding Access Permission

- 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()
```

- Given the following method header in a base case:

```
public void doSomething()
```

- The following method header is not valid in a derived class:

```
private void doSomething()
```

Overriding vs Overloading

- 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

```
public derivedClass(int p1, int p2, double p3)
{
  super(p1, p2);
  instanceVariable = p3;
}
```

- 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**

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

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

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

  - Explicit-value constructor (receives default values):

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

Objects of Multiple Class Types

- An object of a derived class has the type of the derived class, and the type of every one of its ancestor classes
- Therefore, an object of a derived class can be used any place that an object of any of its ancestor types can be used
- However, this relationship does not go the other way
  - An ancestor type can never be used in place of one of its derived types

Pitfall: Using Private Variables

- A private instance variable in a base class is inheritable but not directly accessible *by name* 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
  - An object of the `HourlyEmployee` class can use the `getHireDate` or `setHireDate` methods to access `hireDate`

Protected Access

- If a method or instance variable is modified by **`protected`**, 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

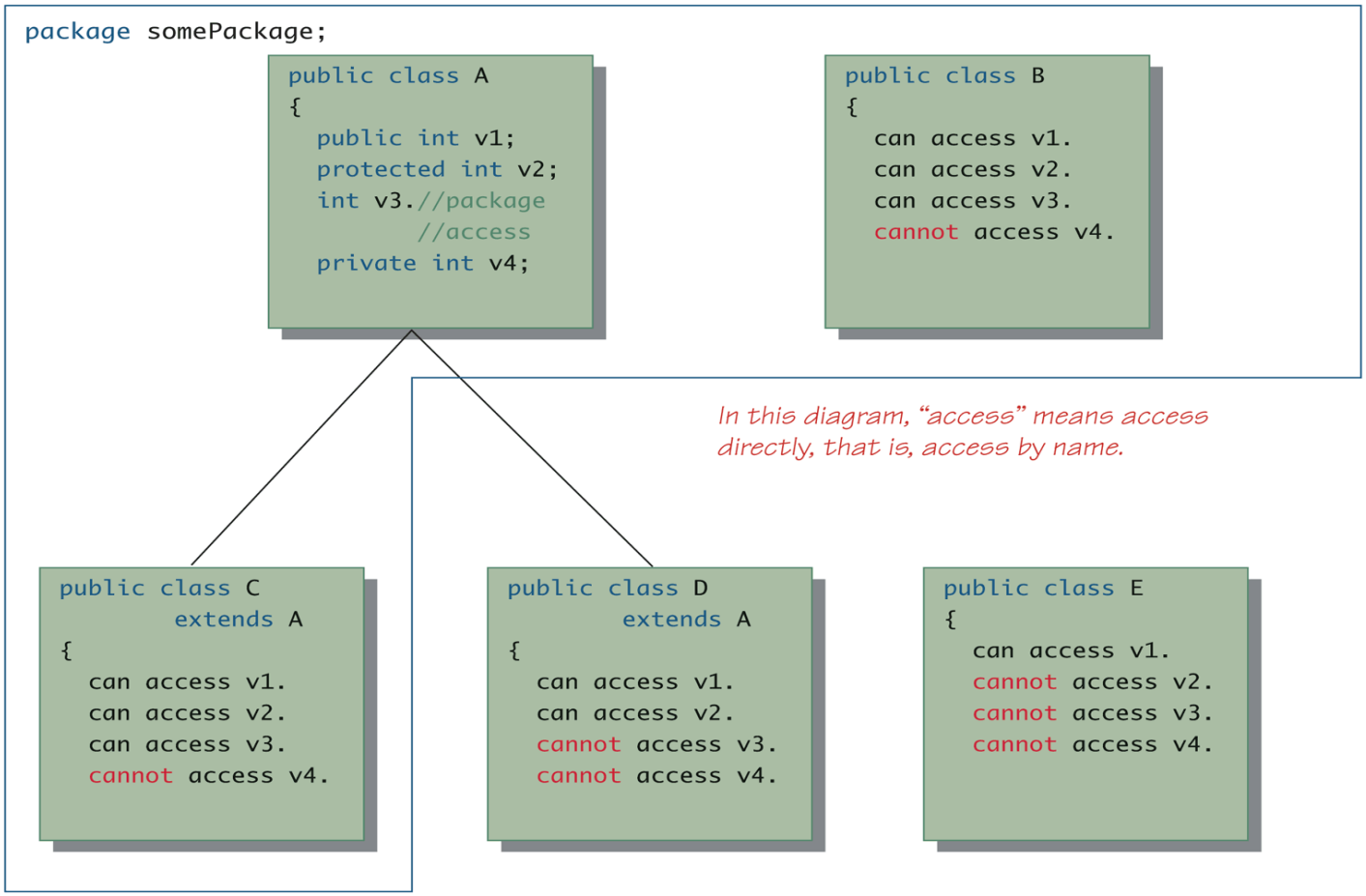
- The **`protected`** modifier provides very weak protection compared to the **`private`** modifier
  - It allows direct access to any programmer who defines a suitable derived class

Package Access

- An instance variable or method definition that is not preceded with a modifier has *package access (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
  - Note that package access is more restricted than `protected`
  - Pitfall: 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

Access Modifiers

Display 7.9 Access Modifiers



Access to an Overridden Method

- Within the definition of a method of a derived class, the base class version of an overridden method can still be invoked

```
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

No Multiple supers

- It is only valid to use **`super`** to invoke a method from a direct parent

- 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 Object Class

- In Java, every class is a descendent of the ***`Object`*** class
- The **`Object`** class is in the package **`java.lang`** which is always imported automatically
- A parameter of type **`Object`** can be replaced by an object of any class whatsoever
- The **`Object`** class has some methods that every class inherits such as **`equals`** and **`toString`** methods

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)
{ . . . }
```

Right Way to Define equals

```
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));
  }
}
```

getClass() vs instanceof

- Many people use the `instanceof` operator in the definition of `equals` instead of the `getClass()` method
- The `instanceof` operator will return `true` if the object is a member of the class for which it is being tested.
- However, the `instanceof` operator will return `true` if it is a descendent of that class as well

Pitfall: getClass() vs 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);  // tested true
boolean testE = h.equals(e);  // tested false
```

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())
```