COP 2250 – Java Programming I - Inheritance - Chapter 11

Download the zipped project for Chapter 11 and import it into your Eclipse workspace.

Imagine that you are a Java programmer creating a Human Resources application. You have already coded these classes:

public class Employee { public class Manager {

public String name; public String name;

public double salary; public double salary;

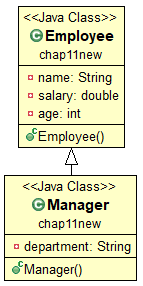
public int age; public int age;

public String department;

} }

Note that Manager is like Employee, but an Employee with an additional attribute or property. There is considerable duplicated code in these classes. Java has a better way!

We can give the Manager class the common attributes from the Employee class with the **extends** keyword. This is called inheritance, one of the characteristics of object-oriented programming. Then, we need only add the one additional attribute in for the Manager.



public class Employee {

private String name;

private double salary;

private int age;

}

public class Manager **extends Employee** {

private String department;

}

The UML for this relationship is shown here 🡪.

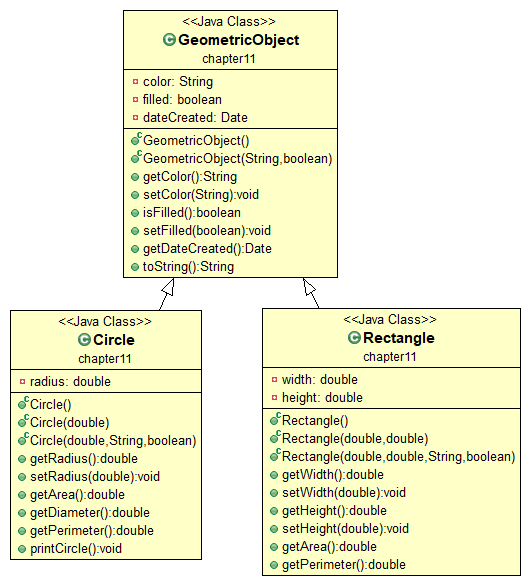
A subclass typically defines additional variables and methods. A subclass inherits the instance variables and instance methods of its superclass, except for its constructors, private instance variables, and private methods. The private instance variables, although not inherited, are stored in a subclass instance.

Superclasses and Subclasses

* Superclasses are coded first, then extended to create subclasses.
* Study the GeometricObject class and its subclasses on pages 413-416.
* Note how UML diagrams are used to indicate inheritance.
* Study the following classes:
  + The superclass GeometricObject
  + GeometricObject subclass named Circle
  + GeometricObject subclass named Rectangle
  + The test class TestCircleRectangle. It has a main method and creates instances.

Try TestCircleRectangle

* Note well the inheritance bullet points on pages 417-418.



## Using the super Keyword

* Subclasses don’t inherit ctors from a superclass.

Problem: superclass instance variables are usually private, so how can the subclass constructor initialize these private instance variables in the superclass? There are two methods:

1. Make the subclass ctor call the superclass ctor with super().
2. Use the **protected** access specifier, instead of private, in the superclass.

* The super( ) call must be the first statement inside the subclass ctor. Eclipse does this for you.
* When a sub class object is instantiated, the ctors of all of its super classes are automatically invoked. This process of **constructor chaining** may continue up to class Object, the zenith of the Java inheritance hierarchy.
* The super keyword can also be used with the dot operator to explicitly call any super class method. See page 418.
* Classes and/or sublasses can be extended by other subclasses to any depth desired, but a class can extend only one superclass. Java uses single inheritance.

Overriding Methods

A sub class can have a new version of a method that was inherited from a superclass. This is called overriding the method. The overriding method in the subclass must:

* Have the **same signature** as the method in the super class.
* Have the **same return type** as the method in the super class.

A subclass can, however, call the superclass version of the overridden method by using the keyword super as in:

super.methodName();

If the method does not exist in the direct superclass, Java will look in the superclass of that class. This can continue up to the highest level class, the **Object class**.

Overriding vs. Overloading

* NOTE the distinction between method overriding and method overloading on page 422.
* NOTE the bullet points on page 423.

The Object Class

* Every Java class has a super class, except for the Object class.
* The Object class is the super class for all other classes.
* It’s at the top of the Java inheritance hierarchy.
* This includes all of the Java API classes and the classes that you write, too.

Methods of the Object Class

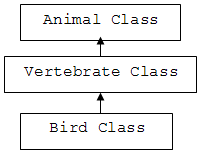
**equals( instance )**

* Used to test that two object instances are equal. It behaves like **==**
* Returns a Boolean. Example: **x.equals( y );**
* You have used this method for comparing Strings, which overrides Object’s equals method
* You will need to override this method for classes that you write.

**toString( )**

* The Object class version returns the class name, the @ character, and a hash code in hexadecimal. You will often override this method in your classes.
* Typically, this is done to reveal the state of the object’s properties.
* Specifying the name of an object variable in System.out.println() will run the object’s toString() method. See page 424.

Polymorphism

* This refers to the fact that an instance of a subclass can take the place of an instance of any of its super classes.
* Put another way, **a variable of a super type can refer to a subclass object**.
* This is logical since, as shown by the class diagrams here, a Bird is a Vertebrate and a Bird is an Animal.
* Polymorphism makes apparently incorrect code acceptable as in this example:

Animal dove = new Bird();

* Note that Animal isn’t even the direct super class of the Bird class.
* In the code above, Animal is the **declared type** but Bird is the **actual type** of the instance.
* A variable of type Object can store a reference to any object type.

Try PolymorphismDemo

Dynamic Binding (page 425)

Object o = new GeometricObject();

System.out.println(o.toString());

* In the code above, the declared type of object o is Object.
* The actual type of object o is GeometricObject.

Question:

* + Both Object and GeometricObject have a toString() method. Which one runs?

Answer:

* + At runtime, the JVM will first search the actual type class for the named method.
  + If it’s not found, the JVM will search the actual type’s immediate super class.
  + The process will continue all the way up to class Object if necessary.
  + In this case, the toString() method of class GeometricObject is executed.
* Since this process happens at runtime, not compile time, it is called dynamic binding.

Try DynamicBindingDemo

Casting Objects and the instanceof operator (page 429-433)

* Casting can be used with objects as well as primitives.

Object ob = new String(“Java”);

* The code above illustrates implicit casting, permissible since String is a subclass of Object.

String str = ob; // won’t compile

* The statement above won’t work. Explicit casting is required as below:

String str = (String) ob; // compiles okay and okay at runtime.

* The Java **instanceof** operator can be used to determine if an object is an instance of a specified class (or of a subclass ).
* If the superclass object is not an instance of the subclass, a ClassCastException is thrown at runtime.

Try CastingDemo

## The Object Class equals Method

## The == operator is used to test primitives for equality. It doesn’t work for reference types.

## To test that two objects of your design are equal, the Object class equals method must be overridden in your classes.

## You have to decide what equals means in your application, typically by comparing the values of properties in the classes.

The ArrayList Class

* This API class solves the issue that an array cannot be re-sized.
* ArrayList is for storing objects, not primitives.
* ArrayList has many methods. See the UML diagram on p435.

Try TestArrayList

* Table 11.1 on page 438 compares and contrasts arrays and ArrayList.

Try DistinctNumbers

Useful Methods for Lists

* Refer to page 440-441 for some important skills for ArrayLists.

**Stacks**

* An ArrayList can be used very nicely to create a custom stack class for storing **objects**.
* Liang provides us with an example called **MySTack.java**.
* Study this class on page 441-442.

Challenge Exercise

Liang didn ‘t provide a “test” program for this MyStack class. Create one called **TestMyStack.java**. In main, create a MyStack instance and add five or six String objects to it. Pop off the last element. Push a new String onto the stack. Process the entire list, displaying all the stored strings.

Protected Data Members and Methods

* This is the third access specifier to our earlier public and private.
* Recall that:
* **public** data and methods can be accessed by ANY class.
* **private** data and methods are accessible ONLY from inside the enclosing class.
* The **protected** keyword allows access to instances and SUBCLASS instances, but no others.
* Note the “visibility increases” continuum on page 443.
* Note Table 11.2. This excellent table explains the visibility of all three access specifiers.

Preventing Extending and Overriding (page 445)

* A class specified with the **final** keyword cannot be subclassed. It can’t be a parent.
* The Math class is a final class.
* The final keyword can also be applied when defining a method.
* This makes it impossible to override that method in a subclass.

## When To Use Inheritance (“is a” and “has a”)

Use the “is a” and “has a” tests:

* Use inheritance when a new class “**is a**” case of an existing class.
* For example, a flower “is a” plant, so it makes sense for class Flower to extend class Plant.
* If “has a” makes sense, don’t use inheritance.
* Since a car “has a” motor, don’t extend class Car to create class Motor.
* This is composition, not inheritance. Class Motor should be an **attribute or property** in class Car.