

CS-302

LAB 2

PART I : Exploring Inheritance

File **Dog.java** contains a declaration for a **Dog** class. Save this file to your directory and study it—notice what instance variables and methods are provided. Files **Labrador.java** and **Yorkshire.java** contain declarations for classes that extend **Dog**. Save and study these files as well.

File **DogTest.java** contains a simple driver program that creates a dog and makes it speak. Study **DogTest.java**, save it to your directory, and compile and run it to see what it does. Now modify these files as follows:

1. Add statements in **DogTest.java** after you create and print the dog to create and print a **Yorkshire** and a **Labrador**. Note that the Labrador constructor takes two parameters: the name and color of the labrador, both strings. Don't change any files besides **DogTest.java**. Now recompile **DogTest.java**; you should get an error saying something like

```
./Labrador.java:18: cannot resolve symbol
symbol   : constructor Dog  ()
location: class Dog
```

But if you look at line 18 of **Labrador.java**, it's just a `{`. In fact, the constructor the compiler can't find (**Dog()**) isn't called anywhere in this file.

- a. What's going on? (Hint: What did we say about constructors in subclasses?)
=>
 - b. Fix the problem (which really is in **Labrador**) so that **DogTest.java** creates and makes the **Dog**, **Labrador**, and **Yorkshire** all speak.
2. Add code to **DogTest.java** to print the average breed weight for both your Labrador and your Yorkshire. Use the **avgBreedWeight()** method for both. What error do you get? Why?
=>

Fix the problem by adding the needed code to the **Yorkshire** class.

3. Add an abstract **int avgBreedWeight()** method to the **Dog** class. Remember that this means that the word **abstract** appears in the method header after **public**, and that the method does not have a body (just a semicolon after the parameter list). It makes sense for this to be abstract, since **Dog** has no idea what breed it is. Now any subclass of **Dog** must have an **avgBreedWeight** method; since both **Yorkshire** and **Labrador** do, you should be all set.

Save these changes and recompile **DogTest.java**. You should get an error in **Dog.java** (unless you made more changes than described above). Figure out what's wrong and fix this error, then recompile **DogTest.java**. You should get another error, this time in **DogTest.java**. Read the error message carefully; it tells you exactly what the problem is. Fix this by changing **DogTest** (which will mean taking some things out).

Part II : A Sorted Integer List

File **IntList.java** contains code for an integer list class. Save it to your directory and study it; notice that the only things you can do are create a list of a fixed size and add an element to a list. If the list is already full, a message will be printed. File **ListTest.java** contains code for a class that creates an **IntList**, puts some values in it, and prints it. Save this to your directory and compile and run it to see how it works.

Now write a class **SortedIntList** that extends **IntList**. **SortedIntList** should be just like **IntList** except that its elements should always be sorted. This means that when an element is inserted into a **SortedIntList**, it should be put into its sorted place, not just at the end of the array. Think carefully about what methods and instance variables you have to define in **SortedIntList** and what you can inherit directly from **IntList**—don't override anything you don't have to.

To test your class, modify **ListTest.java** so that after it creates and prints the **IntList**, it creates and prints a **SortedIntList** containing the same elements.

Part III: Another Type of Employee of the Firm

The files **Firm.java**, **Staff.java**, **StaffMember.java**, **Volunteer.java**, **Employee.java**, **Executive.java**, and **Hourly.java** are from Listings 7.16 - 7.22 in your Java textbook. The program illustrates inheritance and polymorphism. In this exercise you will add one more employee type to the class hierarchy (see Figure 7.8 in the Java textbook). The employee will be one that is an hourly employee but also earns a commission on sales. Hence the class, which we'll name **Commission**, will be derived from the **Hourly** class.

Write a class named **Commission** with the following features:

It extends the **Hourly** class.

It has two instance variables (in addition to those inherited): one is the total sales the employee has made (type double) and the second is the commission rate for the employee (the commission rate will be type double and will represent the percent (in decimal form) commission the employee earns on sales (so .2 would mean the employee earns 20% commission on sales)).

The constructor takes 6 parameters: the first 5 are the same as for **Hourly** (name, address, phone number, social security number, hourly pay rate) and the 6th is the commission rate for the employee. The constructor should call the constructor of the parent class with the first 5 parameters then use the 6th to set the commission rate.

One additional method is needed: **public void addSales (double totalSales)** that adds the parameter to the instance variable representing total sales.

The **pay** method must call the pay method of the parent class to compute the pay for hours worked then add to that the pay from commission on sales. (See the pay method in the **Executive** class.) The total sales should be set back to 0 (note: you don't need to set the **hoursWorked** back to 0—why not?).

The **toString** method needs to call the **toString** method of the parent class then add the total sales to that.

To test your class, update **Staff.java** as follows:

Increase the size of the array to 8.

Add two commissioned employees to the **staffList**—make up your own names, addresses, phone numbers and social security numbers. Have one of the employees earn \$6.25 per hour and 20% commission and the other one earn \$9.75 per hour and 15% commission.

For the first additional employee you added, put the hours worked at 35 and the total sales \$400; for the second, put the hours at 40 and the sales at \$950.

Compile and run the program. Make sure it is working properly.

Part IV: Overriding the *equals* Method

File **Player.java** contains a class that holds information about an athlete: name, team, and uniform number. File **ComparePlayers.java** contains a skeletal program that uses the **Player** class to read in information about two baseball players and determine whether or not they are the same player.

1. Fill in the missing code in **ComparePlayers** so that it reads in two players and prints "Same player" if they are the same, "Different players" if they are different. Use the **equals** method, which **Player** inherits from the **Object** class, to determine whether two players are the same. Are the results what you expect?
2. The problem above is that as defined in the **Object** class, **equals** does an address comparison. It says that two objects are the same if they live at the same memory location, that is, if the variables that hold references to them are aliases. The two **Player** objects in this program are not aliases, so even if they contain exactly the same information they will be "not equal." To make **equals** compare the actual information in the object, you can override it with a definition specific to the class. It might make sense to say that two players are "equal" (the same player) if they are on the same team and have the same uniform number.

Use this strategy to define an **equals** method for the **Player** class. Your method should take a **Player** object and return true if it is equal to the current object, false otherwise.

Test your **ComparePlayers** program using your modified **Player** class. It should give the results you would expect.