Lab 6 Inheritance

Exploring Inheritance

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

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

1. Add statements in DogTest.java to 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:14: Dog(java.lang.String) in Dog cannot be applied to ()

{

^

1 error

   If you look at line 14 of Labrador.java it's just a {, and the constructor the compiler can't find (Dog()) isn't called anywhere in this file.

   a. What's going on? (Hint: What call must be made in the constructor of a subclass?)

   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 Laborador 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).

A Sorted Integer List

File *IntList.java* contains code for an integer list class. 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 itCompile 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 in sorted order from smallest to largest. 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. To do this you'll need to do two things when you add a new element:

- Walk down the array until you find the place where the new element should go. Since the list is already sorted you can just keep looking at elements until you find one that is at least as big as the one to be inserted.
- Move down every element that will go after the new element, that is, everything from the one you stop on to the end. This creates a slot in which you can put the new element. Be careful about the order in which you move them or you'll overwrite your data!

Now you can insert the new element in the location you originally stopped on.

All of this will go into your add method, which will override the add method for the IntList class. (Be sure to also check to see if you need to expand the array, just as in the IntList add method.) What other methods, if any, do you need to override?

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 (inserted in the same order). When the list is printed, they should come out in sorted order.

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.

Extending Adapter Classes

Files *Dots.java* and *DotsPanel.java* contain a program that draws dots where the user clicks and counts the number of dots that have been drawn. Compile and run Dots to see how it works.

Now study the code in *DotsPanel.java*. (Dots just creates an instance of DotsPanel and adds it to its content pane.) DotsPanel defines an inner class, DotsListener, that implements the MouseListener interface. Notice that it defines the mousePressed method to draw a dot at the click point and gives empty bodies for the rest of the MouseListener methods.

1. Modify the DotsListener class so that instead of implementing the MouseListener interface, it extends the MouseAdapter class. What other code does this let you eliminate? For now, just comment out this code. Test your modified program.
2. We have been using inner classes to define event listeners. Another common strategy is to make the panel itself be a MouseListener, eliminating the need for the inner class. Do this as follows:
   - Modify the header to the DotsPanel class to indicate that it implements the MouseListener interface (it still extends JPanel).
   - Delete the DotsListener class entirely, moving the five MouseListener methods into the DotsPanel class.
   - The DotsPanel constructor contains the following statement:

     ```
     addMouseListener (new DotsListener());
     ```

     This creates an instance of the DotsListener class and makes it listen for mouse events on the panel. This will have to change, since the DotsListener class has gone away. The listener is now the panel itself, so the argument to addMouseListener should change to this, that is, the current object. So the panel is listening for its own mouse events! Compile and run your modified program; it should behave just as before.
3. When we were using the DotsListener class we saw that it could either implement the MouseListener interface or extend the MouseAdapter class. With the new approach in #2, where the DotsPanel is also a MouseListener, can we do the same thing – that is, can DotsPanel extend MouseAdapter instead of implementing MouseListener? Why or why not? If you're not sure, try it and explain what happens.

Rebound Revisited

The files *Rebound.java* and *ReboundPanel.java* contain a program that has an image that moves around the screen, bouncing back when it hits the sides (you can use any PNG or JPEG you like). Open ReboundPanel.java in the editor and observe the following:

- The constructor instantializes a Timer object with a delay of 20 (the time, in milliseconds, between generation of action events). The Timer object is started with its start method.
- The constructor also gives the initial position of the ball (x and y) and the distance it will move at each interval (moveX and moveY).
- The actionPerformed method in the ReboundListener inner class "moves" the image by adding the value of moveX to x and of moveY to y. This has the effect of moving the image to the right when moveX is positive and to the left when moveX is negative, and similarly (with down and up) with moveY. The actionPerformed method then checks to see if the ball has hit one of the sides of the panel—if it hits the left side (x <= 0) or the right side (x >= WIDTH_IMAGE_SIZE) the sign of moveX is changed. This has the effect of changing the direction of the ball. Similarly the method checks to see if the ball hit the top or bottom.

Now do the following:

1. First experiment with the speed of the animation. This is affected by two things—the value of the DELAY constant and the amount the ball is moved each time.
   - Change DELAY to 100. Save, compile, and run the program. How does the speed compare to the original?
   - Change DELAY back to 20 and change moveX and moveY to 15. Save, compile, and run the program. Compare the motion to that in the original program.
   - Experiment with other combinations of values.
   - Change moveX and moveY back to 3. Use any value of DELAY you wish.
2. Now add a second image to the program by doing the following:
   - Declare a second ImageIcon object as an instance variable. You can use the same image as before or a new one.
   - Declare integer variables x2 and y2 to represent the location of the second image, and moveX2 and moveY2 to control the amount the second ball moves each time.
   - Initialize moveX2 to 5 and moveY2 to 8 (note—this image will have a different trajectory than the first—no longer a 45 degree angle)
   - In actionPerformed, move the second image by the amount given in the moveX2 and moveY2 variable, and add code to check to see if the second image has hit a side.
   - In paintComponent, draw the second image as well as the first.
3. Compile and run the program. Make sure it is working correctly.