Lab sheet: Abstract classes and inheritance

This activity testing the understanding in the inheritance and abstract properties in the OOP concepts. Create four classes Dog.java, Labrador.java, and Yorkshire.java. All types of dogs have **name.** In addition to the name, Labrador have **colour** and String type speak() method is make all types of dogs talk.

Consider the given DogTest.java below

```
public class DogTest
{
   public static void main(String[] args)
   {
      Dog dog = new Dog("Spike");
      System.out.println(dog.getName() + " says " + dog.speak());


   }
}
```

- Fix the problem (which really is in Labrador) so that DogTest.java creates and makes the Dog, Labrador, and Yorkshire all speak.
- Add code to DogTest.java to create both a Labrador and a Yorkshire and add print statements similar to what exists to have both of them "speak" as well. Make sure the three types of dogs "say" different things.
- Now add the following method to DogTest.java:

```
  public static void poke(Dog d) {
    System.out.println(""+d+" says "+d.speak());
        // System.out.println("and its average weight is "+d.avgBreedWeight());
  }
```

and add three invocations of poke(), one for each of the dogs, in main(). What output is generated by the three calls to poke?

Notice that though there is only one line of code d.speak() (the one in poke()), all three of the speak() methods (one in Dog, one in Labrador, one in Yorkshire) are invoked by the three invocations of poke.

As a further experiment, try removing the comment marks in *poke*. What error is generated?

- Adding **avgBreedWeight** to Dog allowed the program to run, but it's not really a satisfactory solution. The problem is that a Dog is a "generic" dog. Because it is generic, it doesn't belong to any particular breed, and hence really shouldn't have an "average breed weight".

- What we really want to enforce is that every descendant class of Dog should represent a particular breed, and hence every descendant class should be required to provide an *averageBreedWeight* method.

  Change the *int* avgBreedWeight() method in the Dog class to be abstract.

- You'll also have to add the keyword abstract between the "public" and "class" at the top of Dog.java. This specifies that Dog is now an "abstract class". It makes sense for Dog to be "abstract", since Dog has no idea what breed it is.

- Now any subclass of Dog must have an avgBreedWeight method; because both Yorkshire and Laborador already do, you should be all set on that count.

- Now, though, you should have some new errors. These revolve around the invocation of *new* in DogTest.java that creates the Dog named "spike". Because Dog is now abstract, it is impossible to create any instances of that class.

- Fix this by changing DogTest (which will mean commenting out some things). Your program should now work just fine.

A Sorted Integer List

Create a file IntList.java contains code for an integer list class. Save it to your directory; 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.

1. 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.

2. 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.