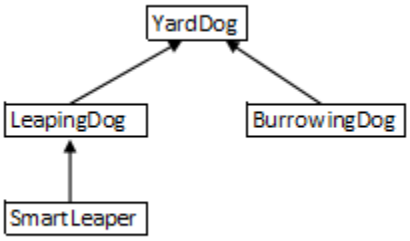
CS 143 Lab 2 Programming Specifications

Checkpoint Due 11:59PM Friday July 13
Lab Due 11:59PM Monday July 16

Problem Overview: This lab is intended to give you experience working with OOP concepts of *inheritance*, *interfaces*, *polymorphism* and *abstract* classes and methods. If you have not read Chapter 9 of the BJP textbook, you should do so now. You are going to write an inheritance tree of classes that simulate bone-burying yard dogs. You will expand on the work you did for Lab 1.



Object-Oriented Programming (OOP): The `YardDog` class will form the base class for the inheritance tree. It will contain everything that is common to all `YardDog` objects, such as the yard with its fence and three bones, the dog's name, and all the methods for managing those data items. In this lab, the `YardDog` class will become purely *abstract*.

LeapingDog objects dig holes by jumping randomly from location to location in the yard. They are not very intelligent and may repeatedly dig holes in the same locations in the yard while searching for a bone.

SmartLeaper objects act like their `LeapingDog` ancestors in the inheritance tree, except they never dig the same hole twice—at least they don't appear to.

BurrowingDog objects start out positioned in the middle of the yard. From there they dig holes by burrowing randomly east, west, north or south from their current location. Being quite stupid as well as destructive, they may dig the same holes numerous times before finally locating a bone.

Saveable Interface: All `YardDog` objects will be "Saveable" which means that you will be able to save and retrieve their data using text files.

Iterative Enhancement: You will approach this lab in the following five stages. This document contains detailed specifications for each stage.

Stage 1: Make the `YardDog` class *abstract*. Implement and test the concrete `LeapingDog` class.

Stage 2: Implement and test the concrete `BurrowingDog` class.

Stage 3: Implement and test the concrete `SmartLeaper` class.

Stage 4: Test polymorphism, and casting from `YardDog` to derived classes.

Stage 5: Implement and test the `save` and `retrieve` methods of the `Saveable` interface of the `YardDog` classes.

Stage 1 Lab 2 Checkpoint: You will make the `YardDog` class *abstract*, and you will implement and test the *concrete* `LeapingDog` class.

- To make the `YardDog` class abstract, only one of the following techniques is needed, but you will use both so that you get practice with the syntax.
    1. Use the keyword `abstract` in the class header as shown in BJP 9.6.

2. Make the `YardDog` class abstract by replacing the Lab-1 `digHoles()` method with the following:

   ```
   public abstract int DigHoles();
   ```

   NOTE: Abstract methods were presented only in the context of interfaces in BJP 9.6, but classes can contain abstract methods too.

- Implement the `LeapingDog` class as indicated in the table below.

`LeapingDog` Class (derived from `YardDog`) **←How will you make this so? See BJP 9.1**	
Method	**Description**
`public LeapingDog()`	Call the default `YardDog` constructor by using the `super` keyword appropriately.
`public LeapingDog(int numRow, int numCols, String name)`	Call the 3-argument `YardDog` constructor using the `super` keyword appropriately.
`public int digHoles()` This method should call the `boneFound()`, `setElement()` and `elementAt()` methods inherited from the `YardDog` class appropriately.	This method overrides the abstract method in the `YardDog` class. It should do the following: • Repeatedly generate random row and column coordinates within the fence line of the yard. • If the coordinate "finds" a bone, an `'H'` character should be placed in the correct location of the yard, otherwise a `'.'` should be placed in that location. • Count each hole as it is dug. • Stop searching for a bone when a bone is "found". • Return the hole count.
`public void leap()`	This method should simulate the dog leaping onto the fence. It should do so by placing the `'@'` character in a random position of the fence line of the `yard`. It should be equally possible for the dog to leap onto any position along any part of the fence line.
`private ???(???)`	You are welcome to create private methods to support the public methods if desired.

- Test Stage 1 of Lab 2 by uncommenting the body of the `stage1_Test()` method of the `YardDogClient.java` file posted in the `Lab2Files` with this assignment. A sample I/O session is shown below.

```
Stages of testing
1. Test modified/abstract YardDog & Leaping dog
2. Test BurrowingDog
3. Test SmartLeaper
4. Test Polymorphism and Casting
5. Test Saveable interface
6. Exit
What stage are you at? 1
```

```
LeapingDog rover = new LeapingDog(10,10,"Rover")
+--------+
|        |
|  A     |
|        |
|        |
|        |
|      C |
|  B     |
|        |
+--------+
Rover
After LeapingDog defaultLeaper = new LeapingDog():
+------------------+
|                  |
|                  |
|                  |
|                  |
|                  |
|     A            |
|                  |
|                  |
|                  |
|     B            |
|                  |
|         C        |
|                  |
+------------------+
Dog
After defaultLeaper.digHoles() and defaultLeaper.leap():
+--@---------------+
|                  |
|                  |
|                  |
|                  |
|          .       |
|     A            |
|     H            |
|                  |
|          .       |
|     B            |
|                  |
|         C        |
|                  |
+------------------+
Dog
Dog dug 3 holes.
Stages of testing
1. Test modified/abstract YardDog & Leaping dog
2. Test BurrowingDog
3. Test SmartLeaper
4. Test Polymorphism and Casting
5. Test Saveable interface
6. Exit
What stage are you at? 6

Good-bye!
```

End Stage 1 Lab 2 Checkpoint

Stage 2: Implement and test the concrete `BurrowingDog` class according to the specifications in the table below:

`BurrowingDog` (derived from `YardDog`)	
Method	Description
`public BurrowingDog()`	Call default `YardDog` constructor appropriately.
`public BurrowingDog(int numRow, int numCols, String name)`	Call 3-parameter `YardDog` constructor appropriately.
`public int digHole()` This method should call the inherited `setElement()`, `elementAt()`, `boneFound()`, `getNumRows()` and `getNumColumns()` methods defined in the base class as appropriate.	1) Initialize dog's current row and column location to the middle of the yard. Initialize a hole counter to 1 2) While current row and column position does not find a bone a) Place a '`.`' character in the dog's location of the yard b) Repeat Generate a random direction value to represent up, down, left or right Until adding/subtracting 1 to/from current row or column position doesn't move the dog under the fence line c) Change the current row/column value to match the random direction change. d) Increment the hole counter 3) Place '`H`' in current row/column position in the yard where bone was found 4) Return hole count value
`public void burrow()`	Simulate the dog burrowing deeply where it's already been by replacing all '`.`' characters in the `yard` with '`@`' characters.
`private ???(???)`	You are welcome to create private methods to support the public methods if desired.

Test Stage 2 of Lab 2 by uncommenting the body of the `stage2_Test()` method of the `YardDogClient.java` file posted in the `Lab2Files` with this assignment. A sample I/O session is shown below.

```
Stages of testing
1. Test modified/abstract YardDog & Leaping dog
2. Test BurrowingDog
3. Test SmartLeaper
4. Test Polymorphism and Casting
5. Test Saveable interface
6. Exit
What stage are you at? 2

BurrowingDog dog = new BurrowingDog(10,10,"Digger")
+--------+
|        |
|        |
```

```
|        |
|  C    B|
|        |
|A       |
|        |
|        |
+--------+
Digger
After dog.digHoles(),
dog yard is:
+--------+
|        |
|        |
|        |
|  CH   B|
|   .    |
|A  ...  |
|   .....|
|   .... |
+--------+
Digger
Digger dug 30 holes.

BurrowingDog dog2 = new BurrowingDog()
+------------------+
|                  |
|                  |
|                  |
|               B  |
|                  |
|                  |
|                 C|
|                  |
|                  |
|                  |
|    A             |
|                  |
|                  |
+------------------+
Dog
After dog2.digHoles() and dog2.burrow(),
dog2 yard is:
+------------------+
|                  |
|                  |
|                  |
|               B  |
|                  |
|                  |
|         @@@      C|
|         @ @      H|
|        @@  @@   @@|
|         @@@@@   @@|
|    A   @@ @@@@@ @|
|       @@@@@@@@@@@|
|      @@@@@ @@@   |
+------------------+
Dog
Dog dug 111 holes.
Stages of testing
```

Etc.

Stage 3: Implement and test the concrete `SmartLeaper` class according to the specifications in the table below:

`SmartLeaper` (derived from `LeapingDog`)	
Method	Description
`public SmartLeaper ()`	Call the default `LeapingDog` constructor appropriately.
`public SmartLeaper(int numRow, int numCols, String name)`	Call the 3-argument `LeapingDog` constructor appropriately.
`public int digHoles()` This method should call the inherited `setElement()`, `elementAt()` and `boneFound()` methods defined in the base class appropriately.	This method repeatedly generates random row and column coordinates within the fence line until they land on a square in the yard that is "empty" or contains a bone. Only then is the 'H' or '.' placed in the yard, depending on whether the hole dug "finds" a bone or not, and the hole is counted. This may give the appearance of the dog being smarter than its ancestor because it won't be seen to dig the same hole more than once. Return the final hole count.
`private ???(???)`	You are welcome to create private methods to support the public methods if desired.

Test Stage 3 of Lab 2 by uncommenting the body of the `stage3_Test()` method of the `YardDogClient.java` file posted in the `Lab2Files` with this assignment. A sample I/O session is shown below.

```
Stages of testing
1. Test modified/abstract YardDog & Leaping dog
2. Test BurrowingDog
3. Test SmartLeaper
4. Test Polymorphism and Casting
5. Test Saveable interface
6. Exit
What stage are you at? 3

Default SmartLeaper after digHoles() and leap():
+------------------+
|                  |
|                  |
|                  |
|                  |
|                  |
|                  |
|      B           |
|        C         |
|       A          |
|                  |
|                  |
|                  |
|                  |
+------------------+
Dog
+------------------+
```

```
|                   |
|                   |
|                   |
|                   |
|                   |
|                   |
|      B            |
|        H          |
|       A           |
|                   |
|                   |
|                   |
|       .           |
@-------------------+
Dog
Dog dug 2 holes.
Alternate SmartLeaper after digHoles() and leap():
+--------@
| HB     |
|        |
|        |
|    .   |
|       C|
|        |
|   A    |
|        |
+--------+
Einstein
Einstein dug 2 holes.

Etc.
```

Stage 4: Write the tester code in the body of the `Stage4_Test()` method of the `YardDogClient.java` file. You are welcome to write additional methods in `YardDogClient.java` in support of your `Stage4_Test()` method. The `Stage4_Test()` method will demonstrate items A and B described below. **You decide what the I/O session for testing Stage 4 of Lab 2 looks like.**

A. **Prove that your `YardDog` inheritance tree of classes supports *polymorphism*.** This means that if you have a `YardDog` variable, invoking its `digHole()` method will cause different behavior depending on whether or not the object that the `YardDog` variable refers to is a `LeapingDog`, `SmartLeaper` or a `BurrowingDog`.

How can you do this?

1. You could call a helper method that accepts a `YardDog` parameter and have that method invoke the parameter's `digHole()` method. See the `printInfo()` method in the `EmployeeMain3` program in BJP 9.3 for an example of this.

   OR…

2. You could declare an array of references to `YardDog` objects, and fill the array with references to a mix of `LeapingDog`, `SmartLeaper` and `BurrowingDog` objects. Then traverse the array invoking the `digHole()` method for each element in the array. See `EmployeeMain4` in the *Polymorphism Mechanics* section of BJP 9.3, or `ShapesMain` in the *Benefits of Interfaces* section of BJP 9.5 for examples of this.

B. **Access the methods unique to the concrete classes when in a polymorphic context**. That is, if you have a `YardDog` variable, be able to invoke the `leap()` method if `YardDog` refers to a `LeapingDog` object (which includes `SmartDog` objects too!), and be able to invoke the `burrow()` method if `YardDog` refers to a `BurrowingDog` object.

How can you do this?

You must use an appropriate cast operation after determining what type of object the `YardDog` variable refers to. Study the `equals()` method shown in the *The `instanceof` Keyword* section of BJP 9.2 for an example of this technique. An alternative to declaring the `other Point` variable in that method would be to do the following:

```
if(o instanceof Point )
   return x == ((Point)o).x && y == ((Point)o).y;
```

As you can see, without using another variable to hold the result of the cast operation, the syntax is quite cumbersome, but now you have seen two ways to cast between object types. Use whichever technique you prefer.

1. If you used strategy 1 in part A above to demonstrate polymorphism, have the helper method determine if the parameter is an `instanceof` a `LeapingDog` or `BurrowingDog`. Then use casting on the parameter to invoke the appropriate `leap()` or `burrow()` method.

   OR…

2. If you used strategy 2 in part A above to demonstrate polymorphism, determine if the array element is an `instanceof` a `LeapingDog` or a `BurrowingDog`. Then use casting on the element to invoke the appropriate `leap()` or `burrow()` method. Study the `ABCDMain` program just above the *Interpreting Inheritance Code* section in BJP 9.3 that declares an array of references to `A` objects. **Now pretend that only `D` objects have `method2()`.** Then the loop body might be changed as follows:

   ```
   if(elements[i] instanceof D) ((D)elements[i]).method2();
   ```

Stage 5: Implement and test the `Saveable` interface. The `Lab2Files` folder posted with this assignment contains a `Saveable.java` file which defined the `Saveable` interface described in the table below.

`Saveable` (Interface)	
Methods (abstract)	Description
`public void save(String filename) throws IOException`	When implemented, the `save()` method should write all of the state information about the specific `YardDog` object into the file specified by the parameter.
`public void retrieve(String filename) throws IOException`	When implemented, the `retrieve()` method should restore all of the state information about the specific `YardDog`.

Considerations: All `YardDog` descendants inherit instances of the base-class data fields, but each subclass can have its own data fields that also need to be saved. Though that is not the case in this

lab, one should consider future possibilities. It makes sense therefore, to distribute the workload of saving and retrieving data between the base class and subclasses shown below.

YardDog	
• Make the `YardDog` class commit all classes in the inheritance tree to implementing the `Saveable` interface with appropriate use of the `implements` keyword introduced in BJP 9.5. • The `YardDog` base class can only save the data fields it knows about. It can't save the data fields of any of its descendent subclasses. It can perform its part of the job as shown in the methods described below, but otherwise it can't implement the `Saveable` interface itself! Notice that the method signatures below don't match those in the `Saveable` interface.	
Methods	**Description**
`public void save(PrintStream output)` .	This method uses statements containing `printf()`, `print()` and/or `println()` calls to write the following data in the order listed to the specified `output` stream: 1. # of rows in the `yard` 2. # of columns in the `yard` 3. All inherited base class data fields as follows: • The `yard` • Dog's `name` HINT: Use `toString()`! REMINDER: When using `printf()` you are responsible for writing `\n` when you wish to end a line in the output stream.
`public void retrieve(Scanner input)` *REMINDER: There is no `nextChar()` method, so you will have to read each row of yard data into a String variable using `nextLine()`, and then transfer each character from the String into the appropriate `yard` location using the `charAt()` String method shown in Table 3.3 in BJP 3.3.*	This method should restore the inherited base class state information by reading the data from the specified `Scanner` into the corresponding data fields, except that the first two values should be used to re-construct the `yard` through the use of the `new` operator. *WARNING: If you are using a mix of tokenized and line input, you must be conscious of consuming `\n` before invoking `nextLine()`!*

`BurrowingDog` and `LeapDog` Implementation of `Saveable` Interface

- `public void save(String filename) throws IOException`
  1. Create a `PrintStream` object connected to file with name provided by parameter
     Reminder: You learned to do this in BJP 6.4.
  2. Call `YardDog`'s `save()` method defined above appropriately.
  3. Write "TBD" into the file to indicate that in the future data "to be determined" could be written into the file.
- `public void retrieve(String filename) throws IOException`
  1. Create a `Scanner` object connected to the file with the name provided by the parameter
     Reminder: You learned to do this in BJP 6.1 and 6.2.
  2. Call `YardDog`'s `retrieve()` method defined above appropriately.

Test Stage 5 of Lab 2 by uncommenting the body of the `stage5_Test()` method of the `YardDogClient.java` file posted in the `Lab2Files` with this assignment. A sample I/O session is shown below.

```
Stages of testing
1. Test modified/abstract YardDog & Leaping dog
2. Test BurrowingDog
3. Test SmartLeaper
4. Test Polymorphism and Casting
5. Test Saveable interface
6. Exit
What stage are you at? 5

SmartLeaper dogL1 = new SmartLeaper(5,5,"Newton");
dogL1:
+---+
|A  |
|   |
|B C|
+---+
Newton
dogL1.save("saveDogL1.txt");
SmartLeaper dogL2 = new SmartLeaper(10,10,"Einstein");
dogL2:
+--------+
|        |
|        |
|        |
|        |
|        |
|      AC|
|        |
|   B    |
+--------+
Einstein
After dogL2.retrieve("saveDogL1.txt")dogL2 should now contain dogL1's data:
+---+
|A  |
|   |
|B C|
+---+
Newton

BurrowingDog dogB1 = new BurrowingDog(5,5,"Digger");
dogB1:
+---+
|   |
|BC |
| A |
+---+
Digger
dogB1.save("saveDogB1.txt");
BurrowingDog dogB2 = new BurrowingDog(10,10,"Rover");
dogB2:
+--------+
|        |
|    C   |
|        |
|        |
|        |
```

```
|    A   |
|        |
|      B |
+--------+
Rover
After dogB2.retrieve("saveDogB1.txt")dogB2 should now contain dogB1's data:
+---+
|   |
|BC |
| A |
+---+
Digger

Stages of testing
1. Test modified/abstract YardDog & Leaping dog
2. Test BurrowingDog
3. Test SmartLeaper
4. Test Polymorphism and Casting
5. Test Saveable interface
6. Exit
What stage are you at? 6

Good-bye!
```

Style and Other Grading Specifications: All programs in this class must adhere to the items listed in the *Good Style Specifications* document posted in the Quick Links module on the Canvas website for this class.

Be sure to read the *How I Grade Weekly Programming Lab Projects* page also posted in the Quick Links module.