**CS 1302 – Lab 5**

**Relationships between Classes**

This is a tutorial on associations between classesand the *StringBuilder* classes. There are 5 stages to complete this lab:

|  |  |  |
| --- | --- | --- |
| **Stage** | **Title** | **Text Reference** |
| 1 | One-to-One Associations | 10.4 |
| 2 | One-to-One Association, Bi-directional Navigability | 10.4 |
| 3 | One-to-Many Association | 10.4 |
| 4 | One-to-Many, Remove Methods | 10.4 |
| 5 | The StringBuilder Class | 10.11 |

To make this document easier to read, it is recommended that you turn off spell checking and grammar checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”
3. **One-to-One Association**

In this stage we study a one-to-one association between classes.

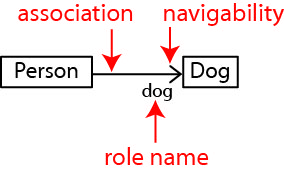
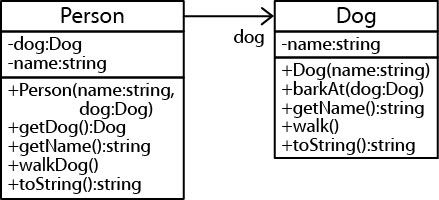
1. (Read, no action required)
2. In the previous chapter, we discussed (1) how a class can be used to model a real-world object, (2) how to write the class, and (3) test the class. In this lab we learn how to model the relationship between two classes. For example:

|  |  |  |
| --- | --- | --- |
| **Relationship** | **Example** | **Class Diagram** |
| *One-to-one* | “A person has a dog” |  |
| *One-to-many* | “A bank has a many customers” |  |
| *Many-to-many* | “A student many courses and a course has many students” |  |

In the remainder of this lab, we consider one-to-one, and one-to-many. We do not consider many-to-many in this course.

1. In terms of a class, what does it mean to say to say, “a person has a dog”? It means that the *Person* class has an instance variable of type *Dog* as shown on the left below.

|  |  |
| --- | --- |
| **public** **class** Person {  ...  **private** Dog dog;  ...  **public** Dog getDog() {  **return** dog;  }  **public** **void** setDog(Dog dog) {  **this**.dog = dog;  }  ...  } | **public** **class** Dog {  **private** String name;  **public** Dog(String name) {  **this**.name = name;  }  ...  } |

1. Consider the class diagram shown on the right. Note:
2. It shows that a *Person has-a Dog*. In UML this is indicated by the solid line between the two classes and is called an *association* (or a *has-a* relationship).
3. The *navigability* is indicated by the arrow at the end of the association. Since the arrow points to *Dog,* this indicates that a *Person* has a *Dog* instance variable.
4. The *role name*, *dog* indicates that this is the name of the instance variable in the *Person* class.
5. In the next few steps we will write the code indicated by the class diagram on the right. Refer back to it as necessary.
6. Do the following:
7. **Establish a Workspace** – Create a folder on your drive where you will put your lab or use an existing one.
8. **Run Eclipse** – As the program begins to run, it will ask you to navigate to the Workspace you want to use.
9. **Create a Project** – Do the following:
10. Choose: File, New, Java Project.
11. Supply a project name, *lab05\_lastName*, *e.g. lab05\_gibson*
12. Choose: *Finish*
13. **Add the Dog Class**
14. Choose: File, New, Class
15. Set the *Package* to “association1”
16. Set the *Name* to “Dog”
17. Choose: Finish
18. Replace everything in the *Dog* class exceptthe package statement at the top.

**public** **class** Dog {

**private** String name;

**public** Dog(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name;

}

**public** String walk() {

**return** name + " is walking";

}

**public** String barkAt(Dog d) {

**return** name + " is barking at " + d.getName();

}

@Override

**public** String toString() {

**return** "dog named " + name;

}

}

1. **Add the Person Class** – This will be the last time I give explicit instructions on creating a class.
2. Select the *association1* node in the *Package Explorer*.
3. Choose: File, New, Class
4. Make sure the *Package* value is “association1”
5. Set the *Name* to “Person”
6. Choose: Finish
7. Replace everything in the *Person* class exceptthe package statement at the top.

**public** **class** Person {

String name;

Dog dog;

**public** Person(String name, Dog dog) {

**this**.name = name;

**this**.dog = dog;

}

**public** String getName() {

**return** name;

}

@Override

**public** String toString() {

**return** name + " has a " + dog;

}

}

1. (Read, no action required)
2. Study the *Person* classes’ constructor:

**public** Person(String name, Dog dog) {

**this**.name = name;

**this**.dog = dog;

}

Notice, that when you create a person, you are required to pass in a dog (as well as the name of the person). Thus, to create a person, we could write code like this:

Dog d = **new** Dog("Juno");

Person p = **new** Person("Xavier", d);

1. Study the *Person* classes’ *toString*:

@Override

**public** String toString() {

**return** name + " has a " + dog;

}

Note:

* *name* is the name of the person
* When we append the string with *dog*, this is implicitly calling the *Dog’s toString* method. Thus:

Dog d = **new** Dog("Juno");

Person p = **new** Person("Xavier", d);

System.***out***.println(p);

Will display:

Xavier has a dog named Juno

1. Add a class named *PersonTest* in the *association1* package and replace the code with:

**public** **class** PersonTest {

**public** **static** **void** main(String[] args) {

*testPersonCreation*();

}

**public** **static** **void** testPersonCreation() {

System.***out***.println("-->testPersonCreation");

Dog d = **new** Dog("Juno");

Person p = **new** Person("Xavier", d);

System.***out***.println(p);

}

}

1. Study the test method, run the class, and verify the output.
2. (Read, no action required) Notice that the *Dog* class has a *walk* method which returns a string:

**public** String walk() {

**return** name + " is walking";

}

Suppose we want to add a way for the person to tell the dog to walk? We can do that by providing a *walkDog* method in the *Person* class that simply tells her dog to walk.

1. Do the following:
2. Add this method to the *Person* class after the constructor:

**public** String walkDog() {

**return** name + " walks dog: " + dog.walk();

}

Notice that the method uses its reference, *dog* and calls the *walk* method. This is called *delegation*. The person is *delegating* to the dog’s *walk* method.

1. Add this test method to *PersonTest*:

**public** **static** **void** testWalkDog() {

System.***out***.println("\n-->testWalkDog");

Dog d = **new** Dog("Juno");

Person p = **new** Person("Xavier", d);

String msg = p.walkDog();

System.***out***.println(msg);

}

1. Add this line of code to *main* to call the method:

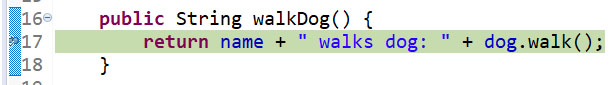
*testWalkDog*();

1. Run the code and verify the output.
2. Next, we will use the debugger to trace the execution of the code when we call this method. Do the following:
3. Open the *Person* class and add a breakpoint on this line in the *walkDog* method:

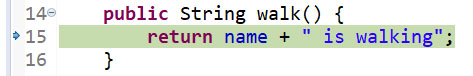
**return** name + " walks dog: " + dog.walk();

Reminder: To add a breakpoint, put your cursor on the line of code above and then double-click in the blue bar in the left margin. A small circle will appear.

1. Open the *PersonTest* class and choose: Run, Debug. Execution should be halted at the line shown below in the *Person* class.



1. Next, we *step into* this line. What will happen is that the *Dog* class will be displayed and execution is stopped in the *walk* method as shown below. Do this now by choosing: Run, Step Into (or press F5)



1. Next, we step over (or into) the line. What will happen is that the line will execute and control will return to the *Person* class. Do this now by choosing: Run, Step Over (or press F6)
2. Press F6 three times. The result is that the message has been displayed and execution is stopped at the end of the *testWalkDog* method in the *PersonTest* class.
3. Choose: Run, Terminate (or press the red square on the tool bar).
4. Return to the Java Perspective by pressing the icon in the upper right of your screen (hover your mouse over the icons and you will see, “Java”)
5. Open the *Person* class and remove the breakpoint (double-click the circle in the left margin)
6. We will add a way for the Person to provide a reference to its Dog. Do the following:
7. Add this method to the *Person* class after the *walkDog* method:

**public** Dog getDog() {

**return** dog;

}

1. Add this test method to *PersonTest*:

**public** **static** **void** testGetDog() {

System.***out***.println("\n-->testGetDog");

Dog d = **new** Dog("Juno");

Person p = **new** Person("Xavier", d);

Dog d2 = p.getDog();

String msg = d2.walk();

System.***out***.println(msg);

}

Notice that this code obtains a reference, *d2* to the person’s dog and then we use it to tell the dog to walk.

1. Add this line of code to *main* to call the method:

*testGetDog*();

1. Run the code and verify the output.

1. **One-to-One Association, Two-way Navigability**

In this stage we study a one-to-one association between classes with navigability in both directions.

1. C:\Users\dgibson\OneDrive - Valdosta State University\Data-Grant\Innovation2015\labs\02\t3.jpg(Read, no action required) Consider the class diagram shown on the right which shows that a *Person has-a Dog* and shows *navigability* in both directions. In other words, the right arrow indications that if you have a *Person*, you can get her *Dog*. The left arrow indicates that if you have a *Dog* you can get its *owner* (*Person*).
2. Do the following:
3. Save and close all open files.
4. Copy the *association1* package and paste it giving the new name *association2*.
5. Do the following:
6. Open the *Dog* class that is in the *association2* package.
7. Add a reference to the owner in the Dog class. Thus, add this instance variable to the Dog class:

**private** Person owner;

1. Add a getter and a setter for the *owner* by adding this code to the Dog class:

**public** Person getOwner() {

**return** owner;

}

**public** **void** setOwner(Person owner) {

**this**.owner = owner;

}

1. Replace the code in *toString* in the *Dog* class with :

**return** "dog name: " + name + ", owner:" + owner.getName();

Study the code carefully so that you understand what it is doing.

1. Open the *Person* class and replace the code in *toString* with:

**return** "person name: " + name + ", dog:" + dog.getName();

Study the code carefully so that you understand what it is doing.

1. Open *PersonTest* and replace all code (except the package statement) with:

**public** **class** PersonTest {

**public** **static** **void** main(String[] args) {

*testPersonCreation*();

}

**public** **static** **void** testPersonCreation() {

System.***out***.println("-->testPersonCreation");

Dog d = **new** Dog("Juno");

Person p = **new** Person("Xavier", d);

d.setOwner(p);

System.***out***.println(p);

System.***out***.println(d);

}

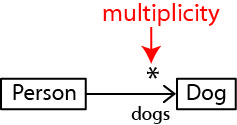
}

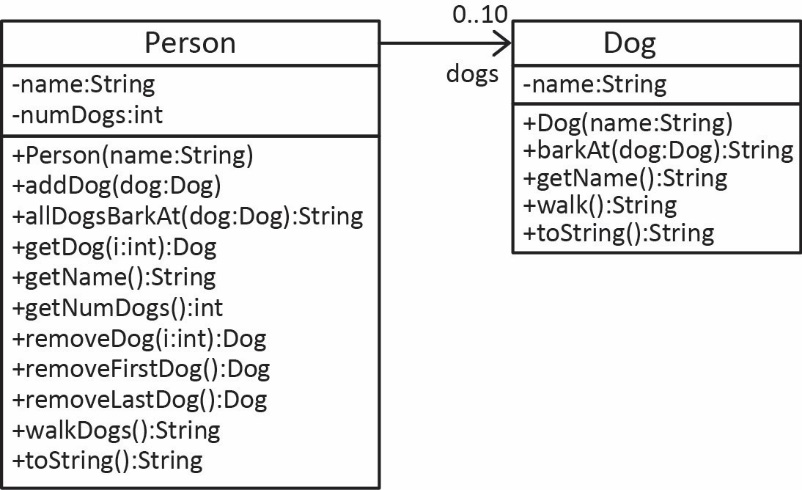
Study the code in the test method carefully. The highlighted line above connects the dog to the person.

1. Run and verify the output.
2. **One-to-Many Association**

In this stage we study a one-to-many association between classes.

1. (Read, no action required). This section is very important. We will be considering the 1-to-many relationship the rest of the semester.



1. Consider the class diagram shown on the right which shows that a *Person has-many Dogs*. The “\*” in the class diagram indicates the *multiplicity.* In this case the multiplicity is “many” (technically, it means any number of Dogs, including 0).
2. *Multiplicity* in a class diagram indicates how many of one object (*Dog*) that another object (*Person*) possesses. For the example we code next (as shown on the right), we will use a multiplicity of “0..10” which means a Person can have up to 10 Dogs. We will implement the new Person class as we go along.
3. The *Person* class needs an instance variable hold the (up to) 10 dogs.

Dog[] dogs = **new** Dog[10];

1. Whenever we implement 1-many, we almost always provide an instance variable to keep track of how many items (*Dogs*) we have. Thus, as shown in the class diagram above, we have a *numDogs* instance variable:

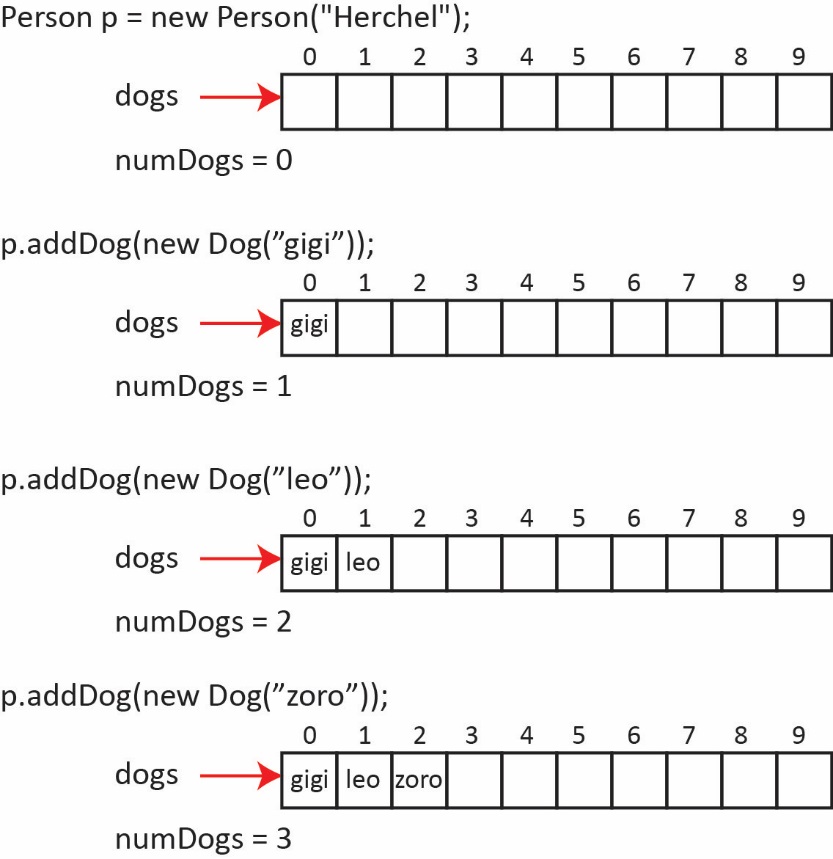
**private** **int** numDogs = 0;

and a getter, *getNumDogs*:

**public** **int** getNumDogs() {

**return** numDogs;

}

1. The usual approach is to provide an “add” method to add items. Thus, we provide an *addDog* method. The standard approach is to store the dogs sequentially, starting at position 0. Notice that the first *Dog* added is added at index 0, the next at index 1, *etc.* And, each time we add a *Dog, numDogs* is incremented.
2. The getter for *numDogs* simply has to return the instance variable.

**public** **int** getNumDogs() {

**return** numDogs;

}

1. We provide an *addDog(dog:Dog)* method that accepts a dog and adds it in the next available slot at the end of the array.

**public** **void** addDog(Dog dog) {

**if**(numDogs<dogs.length) {

dogs[numDogs] = dog;

numDogs++;

}

}

Study the *addDog* method carefully. Note the following:

* It first makes sure the array is not full.
* The *numDogs* instance variable not only stores how many dogs we currently have, it also is the position in the array where the next dog will go. Make sure you see this. Initially, there are 0 dogs. So, where does the first dog that is added go? It goes in position 0. If there are three dogs – the next dog goes in position 3.
* Finally, we increment *numDogs.*

1. We provide a *getDog(i:int)* method that accepts an index for the *Dog* to return. Note that we require that the index be valid: it must be between 0 and the number of dogs that have been added.

**public** Dog getDog(**int** i) {

**if**(i>=0 && i<numDogs)

**return** dogs[i];

**return** **null**;

}

Notice that the stopping condition for the loop above is:

i<numDogs

A common mistake is to use the length of the *dogs* array:

i<dogs.length

which is incorrect because the array is not necessarily full and a runtime error would result if it is not.

1. Do the following:
2. Save & close any open files.
3. Create a new package named: *association3*.
4. Add a new class in that package named: *Person*
5. Copy the *Dog* class from the *association1* package and paste into the *association3* package.
6. Replace the code in the *Person* class with:

**public** **class** Person {

**private** Dog[] dogs = **new** Dog[10];

**private** **int** numDogs = 0;

**private** String name;

**public** Person(String name) {

**this**.name = name;

}

**public** **int** getNumDogs() {

**return** numDogs;

}

**public** **void** addDog(Dog d) {

**if**(numDogs<10 ) {

dogs[numDogs] = d;

numDogs++;

}

}

**public** Dog getDog(**int** i) {

**if**(i>=0 && i<numDogs) {

**return** dogs[i];

}

**return** **null**;

}

}

1. Add a class named, *PersonTest* and replace the code with:

**public** **class** PersonTest {

**public** **static** **void** main(String[] args) {

*testAddDog*();

*testGetDog*();

}

**public** **static** **void** testAddDog() {

System.***out***.println("-->testAddDog");

Person p = **new** Person("Wilma");

System.***out***.println("Num dogs before addDog:" + p.getNumDogs());

p.addDog(**new** Dog("Zoro"));

System.***out***.println("Num dogs after addDog:" + p.getNumDogs());

p.addDog(**new** Dog("Gigi"));

System.***out***.println("Num dogs after addDog:" + p.getNumDogs());

}

**public** **static** **void** testGetDog() {

System.***out***.println("\n-->testGetDog");

Person p = **new** Person("Wilma");

p.addDog(**new** Dog("Zoro"));

p.addDog(**new** Dog("Gigi"));

p.addDog(**new** Dog("Chaps"));

**for**(**int** i=0; i<p.getNumDogs(); i++) {

Dog d = p.getDog(i);

System.***out***.println(d);

}

}

}

1. Study the test code, run, and verify the output. Note in *testGetDog* how we loop over all the dogs:

**for**(**int** i=0; i<p.getNumDogs(); i++) {

Dog d = p.getDog(i);

System.***out***.println(d);

}

The *Person* classes’ *getDog* method knows how many dogs there are.

1. Do the following:
2. Open the *Person* class and add this *toString* method:

@Override

**public** String toString() {

String msg = "";

msg = String.*format*("Person:%s, Dogs:\n", name);

**for**(**int** i=0; i<numDogs; i++) {

msg += String.*format*(" %s\n", dogs[i].getName());

}

**return** msg;

}

Study this code carefully. Notice that we loop over the *dogs* array building a string with each dog’s name. As noted before, the loop does not loop over the entire array (*dogs.length*), it only loops over the actual number of dogs that have been added, *numDogs*. Make sure you understand this.

1. Open the *PersonTest* class and add this test method:

**public** **static** **void** testToString() {

System.***out***.println("-->testToString");

Person p = **new** Person("Wilma");

p.addDog(**new** Dog("Zoro"));

p.addDog(**new** Dog("Gigi"));

p.addDog(**new** Dog("Chaps"));

System.***out***.println(p);

}

1. Add this call to the method in *main*:

*testToString*();

1. Run and verify the output.
2. Next, we provide a method that allows the Person to walk all her Dogs. Do the following:
3. Add this method to the Person class:

**public** String walkDogs() {

String msg = "Dog's are walking...:\n";

**for**(**int** i=0; i<numDogs; i++) {

msg += " " + dogs[i].walk() + "\n";

}

**return** msg;

}

Notice that the method loops over the actual number of Dogs and asks each Dog to walk.

1. Add this test method to the *PersonTest* class:

**public** **static** **void** testWalkDogs() {

System.***out***.println("-->testWalkDogs");

Person p = **new** Person("Sheena");

p.addDog(**new** Dog("Ace"));

p.addDog(**new** Dog("Leo"));

p.addDog(**new** Dog("Fritzy"));

String msg = p.walkDogs();

System.***out***.println(msg);

}

1. Add a call to this method in *main*:

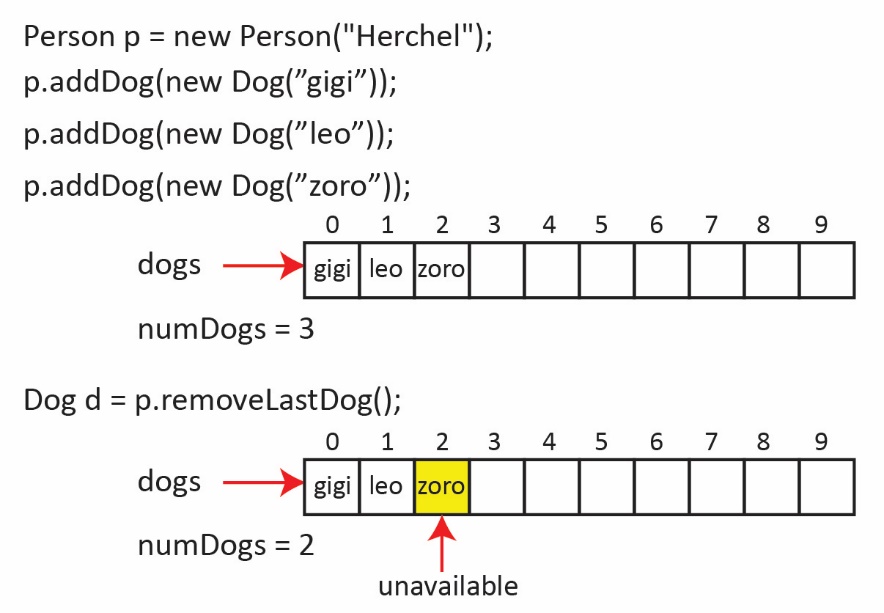
*testWalkDogs*();

1. Run and verify the output.
2. **One-to-Many, Remove Methods**

Next, we consider removing a dog from a person’s array of dogs. We will do this incrementally, by first considering the easiest case, removing the last dog. Thus, below, we will see how to implement the *removeLastDog* method. After this, we consider, *removeFirstDog*. Finally, we consider, *removeDog(position)*.

1. (Read, no action required) We provide a method, *removeLastDog,* to remove the last dog and return it (typically, *remove* methods not only remove the item, but also return it).
2. Where is the last dog located? We know that *numDogs* contains the position where the *next* dog would be added. Thus, if we subtract 1 from *numDogs,* that will be the location of the last dog. Look very carefully at the method below (either version) and verify that it is returning the last dog and decrementing the number of dogs.

|  |  |
| --- | --- |
| **Method** | **Alternate Version** |
| **public** Dog removeLastDog() {  **if**(numDogs>0) {  numDogs--;  **return** dogs[numDogs];  }  **return** **null**;  } | **public** Dog removeLastDog() {  **if**(numDogs>0) {  **return** dogs[--numDogs];  }  **return** **null**;  } |

1. Now, follow the example below. Note, we didn’t actually remove the last dog, we just decremented the number of dogs. Thus, even though the last dog is still at index 2 in the example below, it is unavailable because *addDog*, *getDog*, *etc* all depend on *numDogs*. If this is not perfectly clear, consider executing this line of code at the end of the example below: d=p.getDog(2). Now go back and look at the *getDog* method and see that it will return *null*.
2. Next, we add and test the *removeLastDog* method. Do the following:
3. Add this method to the Person class:

**public** Dog removeLastDog() {

**if**(numDogs>0) {

numDogs--;

**return** dogs[numDogs];

}

**return** **null**;

}

1. Add these test methods to the *PersonTest* class:

**public** **static** **void** testRemoveLastDog\_3\_Dogs() {

System.***out***.println("-->testRemoveLastDog\_3\_Dogs");

Person p = **new** Person("Paul");

p.addDog(**new** Dog("Ace"));

p.addDog(**new** Dog("Leo"));

p.addDog(**new** Dog("Fritzy"));

System.***out***.println("Dogs before removeLastDog:");

**for**(**int** i=0; i<p.getNumDogs(); i++) {

System.***out***.println((i+1) + ". " + p.getDog(i));

}

Dog d = p.removeLastDog();

System.***out***.println("Dogs after removeLastDog:");

**for**(**int** i=0; i<p.getNumDogs(); i++) {

System.***out***.println((i+1) + ". " + p.getDog(i));

}

System.***out***.println("Dog removed:" + d);

}

**public** **static** **void** testRemoveLastDog\_0\_Dogs() {

System.***out***.println("-->testRemoveLastDog\_0\_Dogs");

Person p = **new** Person("Paul");

System.***out***.println("Dogs before removeLastDog:");

**for**(**int** i=0; i<p.getNumDogs(); i++) {

System.***out***.println((i+1) + ". " + p.getDog(i));

}

Dog d = p.removeLastDog();

System.***out***.println("Dogs after removeLastDog:");

**for**(**int** i=0; i<p.getNumDogs(); i++) {

System.***out***.println((i+1) + ". " + p.getDog(i));

}

System.***out***.println("Dog removed:" + d);

}

1. Add a call to these methods in *main*:

*testRemoveLastDog\_3\_Dogs*();

*testRemoveLastDog\_0\_Dogs*();

1. Study the test code carefully, run, and verify the output.
2. (Read, no action required) Next, we provide a method to remove the first dog and return it. The standard approach is to make sure the array holding the dogs has no “holes” in it. Thus, when we remove a dog, we move all the dogs to the right of it over one position to the left.
3. Consider the example showing on the right in the table below: 3 dogs are added, then *removeFirstDog* is called and the 2 dogs to the right are moved over one position to the left.
4. Notice (as we said before) that the last dog, *zoro* in the example below, appears twice, once in its new position (index=1), and once in its original position (index=2). However, we decremented the number of dogs. Thus, even though the last dog is also still at index 2, it is unavailable because *getDog, toString*, *etc* all depend on *numDogs*, which is 2.
5. The algorithm:
6. Get a reference to the first dog
7. Move all the other dogs over one position to the left
8. Decrement the number of dogs
9. Return the first dog
10. The *removeFirstDog* method is shown on the left in the table below.

|  |  |
| --- | --- |
| **Method** | **Example** |
| **public** Dog removeFirstDog() {  **if**(numDogs>0) {  Dog returnDog = dogs[0];  **for**(**int** i=1; i<numDogs; i++) {  dogs[i-1] = dogs[i];  }  numDogs--;  **return** returnDog;  }  **return** **null**;  } | G:\eDataClasses\CS 1302 - Programming 2\notes\03_ch10_OO Thinking\d3.jpg |

1. Next, we add and test the *removeFirstDog* method. Do the following:
2. Add this method to the Person class:

**public** Dog removeFirstDog() {

**if**(numDogs>0) {

Dog returnDog = dogs[0];

**for**(**int** i=1; i<numDogs; i++) {

dogs[i-1] = dogs[i];

}

numDogs--;

**return** returnDog;

}

**return** **null**;

}

1. Add these test methods to the *PersonTest* class:

**public** **static** **void** testRemoveFirstDog\_3\_Dogs() {

System.***out***.println("-->testRemoveFirstDog\_3\_Dogs");

Person p = **new** Person("Paul");

p.addDog(**new** Dog("Ace"));

p.addDog(**new** Dog("Leo"));

p.addDog(**new** Dog("Fritzy"));

System.***out***.println("Dogs before removeFirstDog:");

**for**(**int** i=0; i<p.getNumDogs(); i++) {

System.***out***.println((i+1) + ". " + p.getDog(i));

}

Dog d = p.removeFirstDog();

System.***out***.println("Dogs after removeFirstDog:");

**for**(**int** i=0; i<p.getNumDogs(); i++) {

System.***out***.println((i+1) + ". " + p.getDog(i));

}

System.***out***.println("Dog removed:" + d);

}

**public** **static** **void** testRemoveFirstDog\_0\_Dogs() {

System.***out***.println("-->testRemoveFirstDog\_0\_Dogs");

Person p = **new** Person("Paul");

System.***out***.println("Num dogs before removeFirstDog:" + p.getNumDogs());

Dog d = p.removeFirstDog();

System.***out***.println("Num dogs after removeFirstDog:" + p.getNumDogs());

System.***out***.println("Dog removed:" + d);

}

1. Add a call to these methods in *main*:

*testRemoveFirstDog\_3\_Dogs*();

*testRemoveFirstDog\_0\_Dogs*();

1. Study the test code carefully, run, and verify the output.
2. (Read, no action required) Finally, we provide a method, *removeDog(index:int)* to remove the dog at a particular index and return it. Similar to *removeFirst*, we must move all the dogs after the one we remove, one position to the left. An algorithm:
3. If the index is valid
   1. Get a reference to the dog at the index
   2. Loop over the dogs to the right of the one at the index

* Move current dog one position to the left.
  1. Decrement the number of dogs
  2. Return the dog that was at the index.

1. Else, return null.

Study the code and example carefully, making sure you understand the loop and how it implements step 1b above.

|  |  |
| --- | --- |
| **Method** | **Example** |
| **public** Dog removeDog(**int** i) {  **if**(i>=0 && i<numDogs) {  Dog returnDog = dogs[i];  **for**(**int** j=i+1; j<numDogs; j++) {  dogs[j-1] = dogs[j];  }  numDogs--;  **return** returnDog;  }  **return** **null**;  } | G:\eDataClasses\CS 1302 - Programming 2\notes\03_ch10_OO Thinking\d4.jpg |

1. Do the following:
2. Add the *removeDog* method to the Person class:

**public** Dog removeDog(**int** i) {

**if**(i>=0 && i<numDogs) {

Dog returnDog = dogs[i];

**for**(**int** j=i+1; j<numDogs; j++) {

dogs[j-1] = dogs[j];

}

numDogs--;

**return** returnDog;

}

**return** **null**;

}

1. Write these three test methods in *PersonTest:*

// Add 4 dogs and then remove from index=1

**public** **static** **void** testRemoveDog\_From\_Middle\_4\_Dogs() {

}

// Add 4 dogs and then remove from index=3

**public** **static** **void** testRemoveDog\_From\_End\_4\_Dogs() {

}

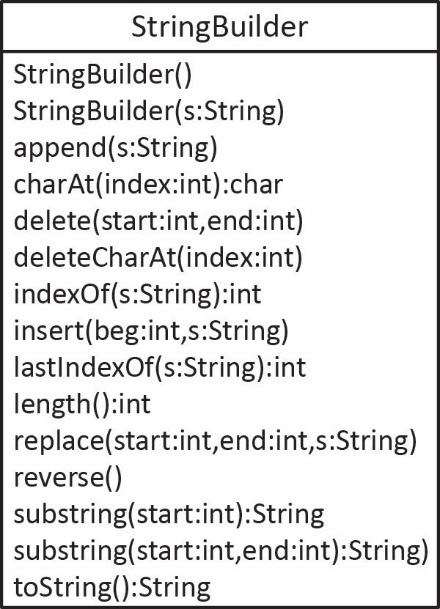
// Add 4 dogs and then remove from index=0

**public** **static** **void** testRemoveDog\_From\_Beginning\_4\_Dogs() {

}

1. Add calls to the new methods, run the test code, and verify the output.
2. **The *StringBuilder* Class**

In this stage we consider the *StringBuilder* class.

1. (Read, no action required)
2. The *StringBuilder* class is similar to the *String* class in that it represents a sequence of characters. However, a *StringBuilder* is much more efficient when the underlying characters are modified or appended, as it does not create a new “string” for every change. Some of its methods are shown in the class diagram on the right.
3. You should use *StringBuilder* any time you need to build or modify a *String* repeatedly through a loop. At the conclusion, simply call *toString*.
4. The example below will only consider the 3 most useful methods:

* *StringBuilder(s:String)* – Constructor, creates a *StringBuilder* object with the initial string, *s.*
* *append(s:String)* – Add *s* to the string inside the *StringBuilder* object.
* *toString()* – Returns the string inside the *StringBuilder* object.

1. Do the following
2. Create a new package named: *stringbuilder\_examples*
3. Create a new class in the *stringbuilder\_examples* package named: *StringBuilderExample*
4. Replace the code (except package statement) with:

**public** **class** StringBuilderExample {

**public** **static** **void** main(String[] args) {

*testStringBuilder*();

}

**public** **static** **void** testStringBuilder() {

String[] pets = {"Chaps", "Moco", "Zoro", "Ace"};

StringBuilder sb = **new** StringBuilder("My Pets:\n");

**for**(**int** i=0; i<pets.length; i++) {

sb.append((i+1) + ". " + pets[i] + "\n");

}

sb.append("There are " + pets.length + " in total");

System.***out***.println(sb.toString());

}

}

1. Study the code carefully. Run and observe the output.
2. If you need to build a string from an arbitrary number of values you should use *StringBuilder* as opposed to concatenating *Strings* as it is much more efficient, especially when the number of values can be large. Next, we will do an experiment to compare using *StringBuilder* and *String* concatenation to build a large string. Do the following:
3. Add this method to the *StringBuilderExample* class:

**public** **static** **void** testStringBuilderEfficiency() {

**int** size = 25000;

**double**[] vals = *buildRandomDoubleArray*(size);

String doubleString1 = *testStringConcat*(vals);

String doubleString2 = *testStringBuilderConcat*(vals);

}

(Read, no action required). This method calls three methods that we will add in just a minute:

* *buildRandomDoubleArray* – Builds an array of 25,000 random doubles.
* *testStringConcat* – Builds a string with all the 25,000 doubles with a comma and a space between each one.

String doublesString = "";

**for**(**double** d : vals) {

doublesString += d + ", ";

}

* *testStringBuilderConcat* – Builds a string (using *StringBuilder*) with all the 25,000 doubles with a comma and a space between each one.

StringBuilder doublesString = **new** StringBuilder();

**for**(**double** d : vals) {

doublesString.append(d + ", ");

}

The later two methods have code built into them to time how long they take. The code will display the results of the experiment. You will be surprised at the results.

1. Add these 3 methods to the *StringBuilderExample* class:

**public** **static** **double**[] buildRandomDoubleArray(**int** size) {

**double**[] vals = **new** **double**[size];

**for**( **int** i=0; i<vals.length; i++ ) {

vals[i] = Math.*random*()\*1000.0;

}

**return** vals;

}

**public** **static** String testStringConcat(**double**[] vals) {

System.***out***.println("testStringConcat()");

**long** begTime = System.*currentTimeMillis*();

String doublesString = "";

**for**(**double** d : vals) {

doublesString += d + ", ";

}

**long** endTime = System.*currentTimeMillis*();

**double** totTime = (endTime-begTime)/1000.0;

String msg = String.*format*(" Concatenate %,d doubles = %.3f sec", vals.length, totTime);

System.***out***.println(msg);

**return** doublesString;

}

**public** **static** String testStringBuilderConcat(**double**[] vals) {

System.***out***.println("testStringBuilderConcat()");

**long** begTime = System.*currentTimeMillis*();

StringBuilder doublesString = **new** StringBuilder();

**for**(**double** d : vals) {

doublesString.append(d + ", ");

}

**long** endTime = System.*currentTimeMillis*();

**double** totTime = (endTime-begTime)/1000.0;

String msg = String.*format*(" Concatenate %,d doubles = %.3f sec", vals.length, totTime);

System.***out***.println(msg);

**return** doublesString.toString();

}

1. Add this line to *main* to call the method:

testStringBuilderEfficiency();

1. Comment out the call to any other method calls in *main*.
2. Run the code and observe the output.
3. Change 25000 to 30000 in *testStringBuilderEfficiency,* rerun and observe the output.

**Submission**

1. Do the following
2. Make sure all your files are saved in Eclipse.
3. Although not necessary, I recommend closing Eclipse.
4. Zip the four folders (packages) under the *src* folder*: association1, association2, association3, stringbuilder\_examples* into a zip file named: *lab5\_lastname.zip*
5. Upload your zip file to the *lab5* dropbox in Blazeview.