**CS 1302 – Lab 6**

**Inheritance & Polymorphism**

This is a tutorial on inheritance. There are 8 stages to complete this lab:

|  |  |
| --- | --- |
| **Stage** | **Title** |
| 1 | Superclass-Subclass Example |
| 2 | Debugging the Example |
| 3 | Adding a protected Instance Variable |
| 4 | Adding Constructors & the super Keyword |
| 5 | Adding a Instance Variable to the Subclass |
| 6 | Polymorphism & Dynamic Binding |
| 7 | Casting |
| 8 | An Array of Dogs |

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

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

In this stage we introduce the concept of inheritance and superclass-subclass relationships.

1. (Read, no action required).
2. Suppose we have the *Dog* class shown on the left and we need a new class, *WolfDog* on the right:



The two classes are identical, except that the *bark* method is different in the *WolfDog* class.

1. Java provides a way to create the *WolfDog* class that *extends* the *Dog* class so that we don’t have to rewrite the (non-private) members that are the same, we only have to rewrite the *bark* method. This is called *inheritance* or *subclassing.*
2. The *Dog* and *WolfDog* classes are shown below on the left and right, respectively.



Note:

* The *Dog* class is referred to as the *superclass* and the *WolfDog* class is referred to as the *subclass.*
* For now, no constructor is explicitly defined. Remember, from Chapter 9, that if a class does not define a constructor explicitly, thus, by default, there is a no-arg constructor. We will consider constructors that accept arguments shortly.
* The *getName* and *setName* methods are *inherited* by the *WolfDog* class. This means that the *WolfDog* has these methods without have to explicitly write them.
* The *name* instance variable is not *inherited* because it is *private.* Thus, the *WolfDog* class cannot directly use the variable. Later, we will see how to make this variable available in the subclass.
1. To depict this relationship in a class diagram, we use a solid line with an open triangle pointing toward the superclass to denote a superclass-subclass relationship as shown in the diagram on the right. We also say that a *WolfDog is-a Dog*
2. Create a Java project in Eclipse named: *lab06\_lastName.*
3. Do the following:
4. Create a package named: *ver1*
5. Create a class named *Dog*
6. Replace everything in the *Dog* class (exceptthe package statement at the top) with the code below:

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

 **private** String name;

 **public** String getName() {

 **return** name;

 }

 **public** **void** setName(String name) {

 **this**.name = name;

 }

 **public** String bark() {

 **return** "bark";

 }

 @Override

 **public** String toString() {

 **return** "Dog named " + name;

 }

}

1. Create a class named: *WolfDog*
2. Replace everything in the *WolfDog* class (exceptthe package statement at the top) with the code below:

**public** **class** WolfDog **extends** Dog {

 @Override

 **public** String bark() {

 **return** "BARK";

 }

 @Override

 **public** String toString() {

 **return** "WolfDog named " + getName();

 }

}

1. Create a class named: *WolfDogTest*
2. Replace everything in the *WolfDogTest* class (exceptthe package statement at the top) with the code below:

**public** **class** WolfDogTest {

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

 *testDogAndWolfDog*();

 }

 **public** **static** **void** testDogAndWolfDog() {

 WolfDog wd = **new** WolfDog();

 // Calling inherited method

 wd.setName("Juno");

 // Calling inherited method

 String name = wd.getName();

 System.***out***.println("WolfDog's name is: " + name);

 // Calling overridden method

 String msg = wd.bark();

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

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

 }

}

1. Study the test code above. As you can see, we can use the *WolfDog* subclass just as we would any class. Run and verify the output.
2. **Debugging the Example**

In this stage we run the debugger to illustrate exactly how inheritance works. If something doesn’t seem to work simply end the debugging session (choose: Run, Terminate or press the red square on the ToolBar) and restart debugging.

1. Do the following:
2. Set a breakpoint on the this line of code:

wd.setName("Juno");

1. Choose: Run, Debug (or F11). The code should stop on the line as shown in the figure on the right.
2. Choose: Run, Step Into (or F5). Execution advances to the *Dog* class’s *setName* method. Note that *WolfDog* instance, *wd* inherits the *setName* method which is found in the *Dog* class.
3. Press F5 two times and execution returns to the *WolfDogTest* class as shown in the figure on the right.
4. Press F5 and execution advances to the inherited *getName* method in the *Dog* class.
5. Press F5 two times and execution returns to the *WolfDogTest* class as shown in the figure on the right.
6. Press F5 and execution stops on the line shown on the right. Next, we will step into the *bark* method which is in the *WolfDog* class.
7. Press F5 and execution advances to the *bark* method in the *WolfDog* class.
8. End the debugging session (Run, Terminate, or press the Red square on the toolbar.
9. Repeat steps *a-I* above making sure you understand the flow of the code.
10. Open the Java perspective (icon in the upper right that says, “Java” when you hover over it).
11. **Adding a *protected* Instance Variable**
12. Do the following:
13. Open the *WolfDog* class and change *bark* method to use the *name* instance variable as shown below:

@Override

**public** String bark() {

 **return** name + " says: BARK";

}

Note that there is a compile error. Click the red x and note the message:

The field Dog.name is not visible.

Remember, private members are not inherited. We will fix this next.

1. Open the *Dog* class and change the visibility of the *name* instance variable from *private* to *protected*

**protected** String name;

1. Save the *Dog* class.
2. Open the *WolfDog* class and save. Notice that there is no compile error now.
3. (Read, no action required) A class member that is declared with *protected* visibility means that it is available in any subclass (and any class in the same package).
4. **Adding Constructors & the *super* Keyword**
5. Do the following:
6. Open the *Dog* class and add this constructor and save.

**public** Dog(String name) {

 **this**.name = name;

}

1. Open the *WolfDog* class and note that it no longer compiles. To fix this, add this constructor:

**public** WolfDog(String name) {

 **super**(name);

}

Save the class and now it has no compile error. However, *WolfDogTest* now has compile errors. We will fix that shortly.

1. (Read, no action required) The *super* keyword is used to call a superclass constructor with a matching signature. This is similar to the way *this* is used to call another constructor in the same class.



1. Open the *WolfDogTest* class and:
2. Add a name to the *WolfDog* constructor.
3. Comment out the call to *setName*.
4. Run, and verify the output.
5. **Adding an Instance Variable to the Subclass**
6. Close all open files, and then copy the *ver1* package and paste giving the new name: *ver2.*
7. Open the *WolfDog* class and do the following:
8. Add an *toughness (int)* instance variable.
9. Add a getter for *toughness*.
10. Add a *toughness* parameter to the constructor.
11. Add a line to save the parameter in the instance variable.
12. The code will look like this:

**public** **class** WolfDog **extends** Dog {

 **private** **int** toughness;

 **public** WolfDog(String name, **int** toughness) {

 **super**(name);

 **this**.toughness = toughness;

 }

 **public** **int** getToughness() {

 **return** toughness;

 }

 ...

1. Open the *WolfDogTest* class and replace the code in *testDogAndWolfDog* with:

WolfDog wd = **new** WolfDog("Juno", 7);

**int** toughness = wd.getToughness();

String name = wd.getName();

System.***out***.println("WolfDog's name is: " + name + ", toughness=" + toughness);

1. Run and verify the output.
2. Next, we will use the debugger to step through the constructor. Do the following:
3. In *WolfDogTest,* place a breakpoint on the first line in the test method.
4. Choose: Run, Debug. Execution is stopped at the breakpoint.
5. Step into the constructor (F5). Execution is stopped in the *WolfDog* class in the constructor.

Note: if this fails, use the fix found in Lab 1, step 12.

1. Press F5 and execution advances to the *Dog* classes constructor.
2. Press F5 three times and execution returns to the *WolfDog’s* constructor.
3. Press F5 two times and execution returns to the *WolfDogTest* class.
4. Choose: Run, Resume. This completes execution.
5. Repeat steps *a-g* above and pay close attention to the flow of execution.
6. Return to the Java perspective.
7. **Polymorphism & Dynamic Binding**

In this stage we will consider the concepts of *polymorphism* and *dynamic binding.*

1. (Read, no action required).
2. When we write a statement like this:



We say that we are using a *WolfDog reference type* to refer to a *WolfDog instance.* Other than the term *reference type* there is nothing new going on here.

1. We can also use a *Dog* reference type to refer to a *WolfDog* instance:

Dog d = **new** WolfDog("Leo");

The reason (rather loosely) that we can do this is that a *WolfDog is-a Dog*. This is called a *polymorphic reference*. **Anytime we have an *is-a* (inheritance, superclass-subclass) relationship, we can refer to a subclass instance (*WolfDog*) with a super-type reference (*Dog*).** Now, why we would want to do this is a bit more challenging to understand. There are tremendous benefits which we will discuss throughout the remainder of the course!

1. Do the following:
2. Add this method to the *WolfDogTest* class:

**public** **static** **void** testPolymorphism() {

 Dog d = **new** WolfDog("Juno", 7);

 String bark = d.bark();

 String name = d.getName();

 System.***out***.println("WolfDog's name is: " + name + ", bark=" + bark);

}

1. Comment out the call to *testDogAndWolfDog* in *main* and add a call to the new method:

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

 //testDogAndWolfDog();

 *testPolymorphism*();

}

1. Run and observe the output:

WolfDog's name is: Juno, bark=Juno says: BARK

Note that even though we used a *Dog* reference, the *WolfDog*’s *bark* method was called. This is called *dynamic binding*. When the *bark* method is called, the actual *bark* method that runs **is determined by the instance of the object (*WolfDog*) not the reference type (*Dog*)**.

1. Next, we will illustrate dynamic binding by using the debugger. Do the following:
2. Place a breakpoint on the line where the *bark* method is called.
3. Run the debugger (Run, Debug, or F11)
4. Step into (F5) the method call and note that execution has advanced to the *bark* method in the *WolfDog* class.
5. Continue to step through the code until it completes.
6. Return to the Java perspective.
7. Open *WolfDogTest* and add this line to the end of the *testPolymorphism* method:

**int** toughness = d.getToughness();

Note:

* The code does not compile.
* The instance is a *WolfDog* which does have a *getToughness* method.
* The reference type, however, is *Dog*.
* **The reference type defines what methods can be called on an instance.** Since the *Dog* class does not define a *getToughness* method, we cannot call this method, even though the actual instance is a *WolfDog.*This is probably very confusing! Reread this sentence several times. We will discuss this more in class.

Comment out this line so that the code now compiles.

1. **Casting**

In this stage we will consider *casting* an object from one reference type to another. We may use a *Dog* reference to refer to a *WolfDog,* but there may be times we want to change the reference to a *WolfDog* reference so that we can call the *getToughness* method.

1. (Read, no action required). Note the following:
2. Java defines the *instanceof* operator which returns *true* if the left operand is an instance of the class specified as the right operand as shown in the figure on the right.
3. Java allows us to *cast* an object from one reference type to another as shown in the figure on the right. This cast will only succeed if the cast is valid, *i.e.* if *d* really is a *WolfDog*. The cast here will succeed because we insured that *d* is a *WolfDog* with the preceding *if* statement.
4. Do the following:
5. Add this method to the *WolfDogTest* class:

**public** **static** **void** testCasting() {

 Dog d = **new** WolfDog("Juno", 7);

 **if**( d **instanceof** WolfDog ) {

 WolfDog wd = (WolfDog)d;

 System.***out***.println(wd.getToughness());

 }

}

1. Comment out the call to *testPolymorphism* in *main* and add a call to the new method:

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

 //testDogAndWolfDog();

 //testPolymorphism();

 *testCasting*();

}

1. Run and observe the output:
2. **An Array of Dogs**

In this stage we illustrate that an array of type *Dog* can hold *Dog* instances as well as instances of any subclass of *Dog*.

1. Do the following:
2. Add this method to the *WolfDogTest* class:

**public** **static** **void** testArrayOfDogs() {

 Dog[] dogs = **new** Dog[5];

 dogs[0] = **new** Dog("Chaps");

 dogs[1] = **new** Dog("Ace");

 dogs[2] = **new** WolfDog("Juno", 7);

 dogs[3] = **new** Dog("Gigi");

 dogs[4] = **new** WolfDog("Mocho", 4);

 **for**(Dog d : dogs) {

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

 **if**( d **instanceof** WolfDog ) {

 WolfDog wd = (WolfDog)d;

 System.***out***.println(" toughness=" + wd.getToughness());

 }

 }

}

1. Study the code carefully. Note that the *Dog[] dogs* array can hold *Dog* instances as well as instances of any subclass.
2. Comment out the call to *testCasting* in *main* and add a call to the new method:

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

 //testDogAndWolfDog();

 //testPolymorphism();

 //testCasting();

 *testArrayOfDogs*();

}

1. Run and observe the output:

**Submission**

1. Do the following
2. Zip all the folders (packages) under the *src* folder into a zip file named: *lab6\_lastname.zip*
3. Upload your zip file to the *lab6* dropbox in Blazeview.

**You are done!**