cs445 - Spring 2016

Homework #1 - Inheritance, Abstraction, and Polymorphism

Goal

This purpose of this homework is to provide you with experience developing classes using inheritance. You will use abstract classes, concrete classes, and interfaces to write a simple program that uses polymorphism.

Overview

- Understand and use interfaces
- Understand and use inheritance hierarchies
- Understand and use abstract classes
- Understand and use interfaces with inheritance mechanisms
- Create a hierarchy of classes
- Learn how to use the `Object.toString()` method
- Use polymorphism in a simple program

1. Creating a class hierarchy [20%]

You will develop a program that creates a class hierarchy of *Things* and *Creatures*.

Click here to see the Javadoc pages that describe these classes. Read this documentation so that you understand what each class does and the behaviors that it provides.

First write a test program called `TestCreature` to test that the class hierarchy works correctly. The `TestCreature` class contains the *main* method. The main() must create an array of Thing instances of size THING_COUNT; see javadocs for specifics.

Please note that `TestCreature` is what one would call an "acceptance test". Just because you're writing this test doesn't mean you don't have to write your regular unit testing using jUnit.

Create the array and fill it with Thing instances (any non-empty name will do). Next, main() must scan through the array and print the String representation of each Thing. Note that you do not need to call the *toString()* method on each object; just print each object reference from the array directly.

The class `Thing` is a concrete superclass in the class hierarchy. A *client program*, such as your test program, can create instances of the class and print them.

The class `Creature` is an abstract superclass in the class hierarchy. An abstract class represents *generalization*: a general concept embodied by one or more specific cases. It is illegal to create an instance of an abstract class.

A *concrete* subclass of a generalization represents a *specialization*: a specific case of some general concept. The idea of inheritance depends on chains of generalizations and specializations.

Read the documentation for the `Thing` and `Creature` classes. Notice that some of the methods have a declaration, and they have no method body. Methods declared but not implemented are *abstract* methods, which all *concrete* subclasses must implement.

Read the javadoc comments for the `toString()` method for `Thing`. The method must create a `String` that contains the name of the Thing, followed optionally by the class name. Since every `Thing` instance has a name, that should be a private data member of `Thing`. The class name of a `Thing` instance, however, is not a property of the class; it is a unique property of all of `Thing`'s subclasses (and their subclasses).

When you store a reference to a Tiger (for example) in a variable of type Thing, the specific class may be difficult to see. How should you deal with getting the class name of an object reference?

One naive and tempting route to take is to use a series of `instanceof` checks on the subclass instance that invokes `Thing.toString()`.

```
String type;
if ( this instanceOf Creature ) {
    // deal with Creatures here; this is a Creature!
} else if ( this instanceof SomethingElse ) {
    // deal with SomethingElse here
} // ...
```

This chain of instanceof checks is a *very poor design practice*, because it requires constant maintenance of the `Thing` class whenever a new subclass is introduced. The paragraphs below describe a *better* way.

Since all classes inherit from `Object`, each class automatically has access to `Object.getClass()` , which returns a reference to the instance's class instance. With a reference to the `Class`, get the class name using `Class.getSimpleName()` . These two calls together will always produce the class name of the instance that is running the method:

```
// ...
String className = getClass().getSimpleName();
// ...
```

After you have written `Thing`, write `Creature`, which is an abstract subclass of `Thing`. Be sure to use the constructor provided by the `Thing` class when writing your `Creature` class by calling the *super* constructor appropriately.

Printing a `Creature` or any of its subclasses, should use `Thing.toString()`; there is no need to implement toString as a method in `Creature`.

Note: a `Creature`. needs to remember only the last thing it eats.

After you have written these classes, compile them and your test program to make sure that they compile without error. Then run these using your test program.

Now write the `Tiger` class to inherit from the hierarchy. Remember: If a subclass can use a method implementation of a superclass *as-is*, there is no need to re-write the method. If the method is not private, a client can access it.

Please use a stubbing approach to write the abstract methods of `Tiger`. A *stub* method does nothing except implement a method signature and ensure that the class compiles.

Finally, revise the `TestCreature` program to create and print a few Tiger instances. Add code to the main() to create several instances of Tiger in the Thing array; this is in addition to the earlier Thing instances. As a result some Thing instances will be Things and others will be Tigers. Make sure that all classes compile without error, and re-run your test program to validate it.

2. Introducing Interfaces for implementing Additional Behavior [40%]

This activity expands the Thing and Creature framework by adding more creatures and giving them different behaviors.

The next job is to write three more classes, `Ant` and `Fly` and `Bat` that inherit from `Creature`. You must not modify the *Thing.toString()* method. Start by reading the javadocs for each class.

Please use a stubbing approach to write a *stub* for the abstract methods. As described above, a stub method does nothing except implement a method signature and ensure that the class compiles.

A Java interface can achieve results similar to multiple inheritance because a single class may implement more than one interface.

Implement an interface called `Flyer` that specifies the fly() method.

Modify your test program to test the Creatures. The main() must create a separate array for Creature instances. Add a new section of code to create several new Creature instances. Consequently, the main() will have a 'Things:' section followed by a 'Creatures:' section of output. The section for the creatures must simply print the instances (the next activity will revise that).

Below is an example of output for this activity. The 'Creatures:' section evolves in the next section.

```
Things:

Banana
Tigger, Pooh's Friend
Locomotive
Tick-Tock the Crocodile

Creatures:

Tigger, Pooh's Friend
Tick-Tock the Crocodile
```

3. Completing the Thing-Creature Framework [40%]

This part of the assignment completes the program by finishing the stubbed methods the subclasses of Things and Creatures. This means implementing the `Creature.eat()`, `Creature.move()`, and `Creature.whatDidYouEat()` for all the concrete subclasses.

The javadocs describe restrictions on the eating diets of each creature. Read carefully to learn which creatures eat only things, which eat only creatures, and which eat anything.

This activity also requires evolving the TestCreature program to call these new methods and validate their operation. The 'Creatures:' section of TestCreature must now test the methods just implemented in the concrete classes. TestCreature now must have a separate Creature[] array containing references on which to iterate and call the move() method on the different kinds of Creature instances. Note that creatures that move by flying should be using their fly method.

Each class's methods must implement behavior by printing a message to standard output. This message is a simple illustration of different behavior provided by the same message (that is, the same method call on different objects produces different results). This is polymorphism.

Reminder: To invoke a method of a superclass with the same name from the subclass, prefix the method name in the call with `super.`, as in `super.toString()`.

4. Varia

Follow the Assignment Submission instructions in the syllabus to submit your work. Don't forget that we'll also have a look at your code repository, make sure you've shared it with us and that you also include instructions for how to check out your code from the command line and how to build it.

Don't forget the check out the syllabus for the test environment of your work and what's needed in terms of automated unit testing.

Last update: Jan 26, 2016	Virgil Bistriceanu	cs445	Computer Science

---

$Id: hw1.html,v 1.1 2016/01/26 23:41:47 virgil Exp $

Class TestCreature

```
java.lang.Object
  └ TestCreature
```

```
public class TestCreature
extends java.lang.Object
```

TestCreature is a tester/demonstration program to exercise the Thing class hierarchy. Students write this and submit it; try uses another tester.

Since:
2007

Field Summary

`static int`	`CREATURE_COUNT` number of creatures to create
`static int`	`THING_COUNT` number of things to create

Constructor Summary

`TestCreature()`

Method Summary

`static void`	`main(java.lang.String[] args)` TestCreature.main tests the hierarchy of Things and Creatures.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

THING_COUNT

```
public static final int THING_COUNT
```

number of things to create

See Also:

Constant Field Values

CREATURE_COUNT

```
public static final int CREATURE_COUNT
```

number of creatures to create

See Also:

Constant Field Values

Constructor Detail

TestCreature

```
public TestCreature()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

TestCreature.main tests the hierarchy of Things and Creatures. Steps:
- create some Creature instances (i.e. an array of them)
- create some simple Thing instances
- add the Creature instances to the array of simple Thing instances
- print a heading "Things:" followed by a blank line
- print each thing, which each print one line about themeselves
- print a blank line
- print a heading "Creatures:" followed by a blank line
- print each creature
- print a blank line
NOTE: this description is for the FINAL version.

Parameters:

`args` - -- unused

Class Thing

```
java.lang.Object
  └ Thing
```

Direct Known Subclasses:

Creature

```
public class Thing
extends java.lang.Object
```

Thing is the top of a class hierarchy. All Things have a name, given to them at birth. Once born, they cannot change their name. Things have no behavior other than producing their string representation.

Since:

2007

Constructor Summary

`Thing(java.lang.String name)`

Method Summary

`java.lang.String`	`toString()` Produce a String description of this instance.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Thing

```
public Thing(java.lang.String name)
```

Parameters:

`name` - -- the given name of this Thing

Method Detail

toString

```
public java.lang.String toString()
```

Produce a String description of this instance. If the class of the instance is Thing, then return only the name of the Thing. Otherwise add the name of the class after the name of the thing, separated by a space. this makes the class name of the thing serve as the thing's last name (surname), and the given name of the thing is its first name (given name).

Overrides:

`toString` in class `java.lang.Object`

Returns:

the description of this instance

Class Creature

```
java.lang.Object
  └Thing
      └Creature
```

Direct Known Subclasses:
Ant, Bat, Fly, Tiger

```
public abstract class Creature
extends Thing
```

A Creature is a Thing that has specific, 'lively' behaviors.

Since:
2007

Constructor Summary

`Creature(java.lang.String name)` Create a Creature with a name.

Method Summary

`void`	`eat(Thing aThing)` Creatures may be asked to eat various Things.
`abstract void`	`move()` Tell the Creature to move.
`void`	`whatDidYouEat()` Make the Creature tell what is in its stomach.

Methods inherited from class Thing

`toString`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Creature

```
public Creature(java.lang.String name)
```

Create a Creature with a name.

Parameters:
`name` - -- the given name of this Creature

Method Detail

eat

```
public void eat(Thing aThing)
```

Creatures may be asked to eat various Things. By default, creatures eat whatever they are told AND print a message stating '{this object} has just eaten a {a Thing}.' substituting the concrete details for the part enclosed inside of {}. (the single quotes are not part of the message.)
A creature only remembers the last thing it ate.

Parameters:
`aThing` - -- what this is told to eat

---

move

```
public abstract void move()
```

Tell the Creature to move. Each creature will print a message stating the way it most commonly moves.

---

whatDidYouEat

```
public void whatDidYouEat()
```

Make the Creature tell what is in its stomach. If there is nothing in its stomach, whatDidYouEat() prints '{name} {class} has had nothing to eat!' If it has something in its stomach, whatDidYouEat() prints '{creature name} {class name} has eaten a {content of stomach}!'
NOTE: The pattern {word} in the text above indicates what attribute value belongs in the output text.

Class Thing

```
java.lang.Object
  └ Thing
```

Direct Known Subclasses:

Creature

```
public class Thing
extends java.lang.Object
```

Thing is the top of a class hierarchy. All Things have a name, given to them at birth. Once born, they cannot change their name. Things have no behavior other than producing their string representation.

Since:

2007

Constructor Summary

`Thing(java.lang.String name)`

Method Summary

`java.lang.String`	`toString()` Produce a String description of this instance.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Thing

```
public Thing(java.lang.String name)
```

Parameters:

`name` - -- the given name of this Thing

Method Detail

toString

```
public java.lang.String toString()
```

Produce a String description of this instance. If the class of the instance is Thing, then return only the name of the Thing. Otherwise add the name of the class after the name of the thing, separated by a space. this makes the class name of the thing serve as the thing's last name (surname), and the given name of the thing is its first name (given name).

Overrides:

`toString` in class `java.lang.Object`

Returns:

the description of this instance

Class Tiger

```
java.lang.Object
  └Thing
      └Creature
          └Tiger
```

```
public class Tiger
extends Creature
```

Tiger is a creature that knows how to speak.

Constructor Summary
`Tiger(java.lang.String name)` Create a Tiger with a name.

Method Summary	
`void`	`move()` When asked to move, a Tiger prints '{name} {class} has just pounced.' (the single quotes are NOT part of the output.)

Methods inherited from class Creature
`eat, whatDidYouEat`

Methods inherited from class Thing
`toString`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Tiger

```
public Tiger(java.lang.String name)
```

Create a Tiger with a name.

Parameters:

`name` - -- the given name of this Tiger

Method Detail

move

```
public void move()
```

When asked to move, a Tiger prints '{name} {class} has just pounced.' (the single quotes are NOT part of the output.)

Specified by:

`move` in class `Creature`

Class Ant

```
java.lang.Object
  └ Thing
      └ Creature
          └ Ant
```

```
public class Ant
extends Creature
```

Ant is a creature.

Constructor Summary

`Ant(java.lang.String name)` Create a Ant with a name.

Method Summary

`void`	`move()` When asked to move, an Ant prints '{name} {class} is crawling around.' (the single quotes are NOT part of the output.)

Methods inherited from class Creature
`eat, whatDidYouEat`

Methods inherited from class Thing
`toString`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Ant

```
public Ant(java.lang.String name)
```

Create a Ant with a name.

Parameters:

`name` - -- the given name of this Ant

Method Detail

move

```
public void move()
```

When asked to move, an Ant prints '{name} {class} is crawling around.' (the single quotes are NOT part of the output.)

Specified by:

`move` in class `Creature`

Class Fly

```
java.lang.Object
  └ Thing
      └ Creature
          └ Fly
```

All Implemented Interfaces:
Flyer

```
public class Fly
extends Creature
implements Flyer
```

Fly is a creature that knows how to fly.

Constructor Summary

Constructor Summary
`Fly(java.lang.String name)` Create a Fly with a name.

Method Summary

Method Summary	
`void`	`eat(Thing aThing)` A Fly eats only Things; a fly will not eat anything that is a Creature.
`void`	`fly()` When asked to fly, a Fly prints the message '{name} {class} is buzzing around in flight.' (the single quotes are not part of the message.)
`void`	`move()` When asked to move, a Fly flies.

Methods inherited from class Creature
`whatDidYouEat`

Methods inherited from class Thing
`toString`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Fly

`public Fly(java.lang.String name)`

Create a Fly with a name.

Parameters:
`name` - -- the given name of this Fly

Method Detail

eat

`public void eat(Thing aThing)`

A Fly eats only Things; a fly will not eat anything that is a Creature. If told to eat a Creature instance, a Fly says: '{name} {class} won't eat a {aThing}.' (the single quotes are not part of the message.) Otherwise a fly eats the thing just like any other Creature would.

Overrides:
`eat` in class `Creature`

Parameters:
`aThing` - -- what this is told to eat

---

move

`public void move()`

When asked to move, a Fly flies. That is, the instance calls its own fly() method.

Specified by:
`move` in class `Creature`

---

fly

`public void fly()`

When asked to fly, a Fly prints the message '{name} {class} is buzzing around in flight.' (the single quotes are not part of the message.)

Specified by:
`fly` in interface `Flyer`

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Bat

```
java.lang.Object
  └ Thing
      └ Creature
          └ Bat
```

All Implemented Interfaces:
Flyer

```
public class Bat
extends Creature
implements Flyer
```

Bat is a creature that knows how to fly.

Constructor Summary

`Bat(java.lang.String name)` Create a Bat with a name.

Method Summary

`void`	`eat(Thing aThing)` A Bat eats only Creatures in the same way that any Creature eats.
`void`	`fly()` When asked to fly, a Bat prints the message '{name} {class} is swooping through the dark.' (the single quotes are not part of the message.)
`void`	`move()` When asked to move, a Bat flies.

Methods inherited from class Creature

`whatDidYouEat`

Methods inherited from class Thing

`toString`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Bat

`public **Bat**(java.lang.String name)`

Create a Bat with a name.

Parameters:
`name` - -- the given name of this Bat

Method Detail

eat

`public void **eat**(Thing aThing)`

A Bat eats only Creatures in the same way that any Creature eats.
However, Bats will not eat simple things. If a bat eats something that is NOT an instance of Creature (i.e. a bat eats an instance of Thing), then it reports '{name} {class} won't eat a {aThing}.' (the single quotes are not part of the message, {name} {class} is the bat's string representation, and aThing is the string representation of aThing.) Finally, if aThing is an instance neither of Creature nor Thing. the bat is silent and does not eat the instance.

Overrides:
`eat` in class `Creature`

Parameters:
`aThing` - -- what this is told to eat

---

move

`public void **move**()`

When asked to move, a Bat flies. That is, the instance calls its own fly() method.

Specified by:
`move` in class `Creature`

---

fly

`public void **fly**()`

When asked to fly, a Bat prints the message '{name} {class} is swooping through the dark.' (the single quotes are not part of the message.)

Specified by:

`fly` in interface `Flyer`

Interface Flyer

All Known Implementing Classes:

Bat, Fly

```
public interface Flyer
```

Fly is the interface for flying.

Method Summary

`void`	`fly()` Ask the Flyer to fly.

Method Detail

fly

```
void fly()
```

Ask the Flyer to fly.