Lecture 21

Inheritance and Polymorphism continued
Inheritance

• Bit of review:
  • What’s a superclass? What’s a subclass?
  • What is inherited in a subclass?
  • How do we call superclass constructors?
  • What’s the difference between overloading and overriding methods?
  • Do all classes inherit from a superclass?
Polymorphism

• A class defines a type

• Subtype vs supertype

• A subclass is a specialization of a superclass, so every instance of a class is an instance of the supertype.

• This isn’t the same the other way around

• Think of our Fish and Animal classes.

• Every Fish is an Animal, but every Animal is not a Fish
Polymorphism

• So… this means that you can always pass an instance of a subclass to a parameter of its superclass type.

• ???
Polymorphism

Listing 11.5 PolymorphismDemo.java

```java
public class PolymorphismDemo {
    /** Main method */
    public static void main(String[] args) {
        // Display circle and rectangle properties
        displayObject(new CircleFromSimpleGeometricObject
        (1, "red", false));
        displayObject(new RectangleFromSimpleGeometricObject
        (1, 1, "black", true));
    }

    /** Display geometric object properties */
    public static void displayObject(SimpleGeometricObject object) {
        System.out.println("Created on "+ object.getDateCreated() +
        " Color is "+ object.getColor());
    }
}
```

Created on Mon Mar 09 19:25:20 EDT 2011. Color is red
Created on Mon Mar 09 19:25:20 EDT 2011. Color is black
public class Polymorphic {

    public static void main(String[] args) {
        Animal myAnimal = new Animal("Bigfoot", "GRUNT");
        Fish myFish = new Fish("Wanda", "glub glub", true);
        Bird myBird = new Bird("Poly", "Poly wants a cracker", true, 3.0);

        printOut(myAnimal);
        printOut(myFish);
        printOut(myBird);
    }

    public static void printOut(Animal animal) {
        System.out.println(animal.getName() + " says " + animal.getSound());
    }
}
Polymorphism

• An object of a subclass can be used wherever its superclass object is used
Dynamic binding

- You can declare objects by their super type, like:
  - Animal myFish = new Fish();
- The declared type is Animal, the actual type is Fish
- So what happens when I run myFish.getSound()?
Dynamic binding

- Dynamic binding means that at runtime, the JVM will pick the method of the *actual* type to run or move up the inheritance chain until it finds a compatible method...
Polymorphism

• So why would we use this?!?

  • We can use subtypes as if they were the supertypes

  • When we override methods, we can write fewer, more generic methods that can handle all the specific classes
Polymorphism

• What happens if we try to run the `myFish.getLivesInOcean()`

• Can’t do it! Because the Animal class doesn’t know about that method

• BUT myFish is not only an Animal, its a Fish too
Casting

• We can cast an object to its *correct* subclass


((Fish) myFish).getLivesInOcean();

• But be careful! If you tried this on myBird, we’d get an error
• To check if an instance is the right class, you can use the `instanceof` keyword
if (animal instanceof Fish) {
    if ( ((Fish) animal).getLivesInOcean() ){
        System.out.println(animal.getName() + " lives in the ocean");
    } else {
        System.out.println(animal.getName() + " does not live in the ocean");
    }
}
instanceof

- BUT… we should probably use override methods instead (cleaner, extensible)
Object’s equals

• Ripe for overriding!

• In our example, you could return true if the name is the same, or all the data fields are the same, for example
Arraylist

• Don’t be mad….
ArrayList

- Arrays are great for storing data, but they are a bit rigid, inflexible.
  - The size is fixed
  - You can only store items of the same type
- ArrayLists solve this problem
import java.util.ArrayList;

public class ArrayListExample {

    public static void main(String[] args) {
        ArrayList<String> streets = new ArrayList();

        streets.add("main");
        streets.add("water");
        streets.add("86th");
        streets.add("Bleecker");

        System.out.println(streets.size());
        System.out.println(streets.contains("main"));
        System.out.println(streets.indexOf("main"));
        System.out.println(streets.get(3));

        System.out.println();
        System.out.println(streets.remove("main"));
        System.out.println(streets.size());
        System.out.println(streets.contains("main"));
        System.out.println(streets.indexOf("main"));
        System.out.println(streets.get(2));

    }
}
# Arrays vs. ArrayList

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>ArrayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an array/ArrayList</td>
<td>String[] a = new String[10]</td>
<td>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</td>
</tr>
<tr>
<td>Accessing an element</td>
<td>a[index]</td>
<td>list.get(index);</td>
</tr>
<tr>
<td>Updating an element</td>
<td>a[index] = &quot;London&quot;;</td>
<td>list.set(index, &quot;London&quot;);</td>
</tr>
<tr>
<td>Returning size</td>
<td>a.length</td>
<td>list.size();</td>
</tr>
<tr>
<td>Adding a new element</td>
<td></td>
<td>list.add(&quot;London&quot;);</td>
</tr>
<tr>
<td>Inserting a new element</td>
<td></td>
<td>list.add(index, &quot;London&quot;);</td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td>list.remove(index);</td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td>list.remove(Object);</td>
</tr>
<tr>
<td>Removing all elements</td>
<td></td>
<td>list.clear();</td>
</tr>
</tbody>
</table>
**ArrayList**

- They’re great.

- **Note:** you can only store *reference* types, so no primitive data types to store an ArrayList of ints, for example, you have to do:

```
ArrayList<Integer> list = new ArrayList<>();
```

- Use a foreach loop to go through a list

```java
for (String s : streets){
    System.out.print(s + " ");
}
```
Practice: ArrayLists

• Create an ArrayList of Animals
• Add 3 animals of different subclasses to the list
• Loop through and print out the summary for each animal
• Remove the 2nd animal, then print the size of the ArrayList
11.12 Useful Methods of lists

- Copy to and from arrays, sort shuffle
- Sorting and shuffling come from java.util.Collections
Protected Visibility Modifier

- Remember that private data fields and methods can only be accessed from within their own class. That excludes subclasses!

- *Protected* data fields and methods can be seen and accessed in subclasses whether or not they are in the same package.
## Protected Visibility Modifier

<table>
<thead>
<tr>
<th>Modifier on members in a class</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass in a different package</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>default (no modifier)</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Protected Visibility Modifier

```java
package p1;

public class C1 {
    public int x;
    protected int y;
    int z;
    private int u;

    protected void m() {
    }
}

package p2;

public class C2 {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    can access o.z;
    cannot access o.u;

    can invoke o.m();
}

public class C3 extends C1 {
    can access x;
    can access y;
    can access z;
    cannot access u;

    can invoke m();
}

public class C4 extends C1 {
    can access x;
    can access y;
    cannot access z;
    cannot access u;

    can invoke m();
}

public class C5 {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;
    cannot access o.u;

    cannot invoke o.m();
}
```
Packages

- Packages are used to organize code into modules
  - Keeps code organized
  - Makes sure names don’t collide
  - Allows for more fine-grained control of access
## Packages

### Table 11.2 Data and Methods Visibility

<table>
<thead>
<tr>
<th>Modifier on members in a class</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass in a different package</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>protected</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>–</td>
</tr>
<tr>
<td>default (no modifier)</td>
<td>✔</td>
<td>✔</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>private</td>
<td>✔</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Packages

• We use import to use classes from a different package

```java
import animals.*;

public class TestingVisibility {

    public static void main(String[] args) {
        Animal myAnimal = new Animal("Bigfoot", "GRUNT");
        System.out.println(myAnimal.getName());
    }
}
```
• Remember that a final data field is a constant, meaning it can’t change

• You can do the same for classes and methods so they cannot be extended or overridden!

```java
public final class A {
}
```

• Examples of final classes are Math, String, StringBuilder, etc