PRA2003: Programming Skills

Week 2 Tasks

Object Oriented Programming

Cameron Browne & Andreea Grigoriu

November 6, 2018

This week you will explore three main concepts in Object Oriented Programming:

- Encapsulation
- Inheritance
- Polymorphism

Task 1. The Benefits of Encapsulation

Refer to your `World` class from Assignment 1. If you have declared the numbers of `rows` and `cols` in your `World` class `private`, remove the `private` visibility modifier from the declaration. Then do the following:

1. Create the world grid as in Assignment 1, but with 5 rows and 4 columns, i.e.

   ```
   final World world = new World(5, 4, 7);
   ```

2. Run the game at this smaller size to check that it works.

3. Change the number of rows to 6 and columns to 5 using unsafe direct assignment (from outside the `World` class), *after* creating the `World` object but *before* starting the main loop, like so:

   ```
   world.rows = 6;
   world.cols = 5;
   ```

4. Now run the game and see what happens.

5. Explain your findings to the instructor.

How could this behaviour have been prevented easily using an OOP approach?

Task 2.2. Extending the Animal Kingdom

Given the following code:

```
1  /**
2   * Animal kingdom example.
3   * @author cambolbro (based on code from previous years).
4   */
5  abstract class Animal
6  {
7      private final String name;
8
9      public Animal(final String name)
```

```
10      {
11          this.name = name;
12      }
13
14      @Override
15      public String toString()
16      {
17          return name + " goes " + makeNoise();
18      }
19
20      public abstract String makeNoise();
21  }
22
23  //-------------------------------------------------
24
25  class Cat extends Animal
26  {
27      public Cat(final String name)
28      {
29          super(name);
30      }
31
32      public String makeNoise()
33      {
34          return "Meow";
35      }
36  }
37
38  //-------------------------------------------------
39
40  class Bird extends Animal
41  {
42      public Bird(final String name)
43      {
44          super(name);
45      }
46
47      public String makeNoise()
48      {
49          return "Caw";
50      }
51  }
52
53  //-------------------------------------------------
54
55  class Robin extends Bird
56  {
57      public Robin(final String name)
58      {
59      super(name);
60      }
61  }
62
63  //-------------------------------------------------
64
65  public class Animals
66  {
67      public static void main(final String[] args)
68      {
69          // Can't do the following, as Animal is an abstract class
70          // Animal a = new Animal("Generic Animal");
71
72          final Cat cat = new Cat("Fluffy");
73          final Animal cat2 = new Cat("Furry");
74
```

```
        final Animal bird1 = new Bird("Generic Bird #1");
        final Animal bird2 = new Robin("Hood");

        System.out.println(cat);
        System.out.println(cat2);
        System.out.println(bird1);
        System.out.println(bird2);
    }
}
```

Add a `Dog` class and a `LabradorRetriever` class to the animal kingdom, both of which make the noise "Woof". Then, add an age attribute (`double`), shared by every animal, whose value is passed to the constructor as per the animal's name. Then add an `isOld()` method that returns true if and only if any of the following condition hold:

- The animal is a cat and its age is greater than or equal to 10.0.
- The animal is a bird its age is greater than or equal to 0.8.
- The animal is a dog and its age is greater than or equal to 12.0.

Implement this through method overriding, as per `makeNoise()`.

Change the `toString()` method to add " and is old" if the animal is old. Change the arguments to the existing constructors so that Fluffy is 8, Furry is 11, Generic Bird #1 is 0.3 and Hood is 0.9. Create a dog named Generic Puppy #1 who is 0.7 and a labrador retriever Cute Doggy Dawg who is 12.1, and print them out as well. To avoid code duplication, use inheritance as much as possible.

Task 2.3. Generalised Bubble Sort

A *comparable* object is one that can be compared to another. Sorting algorithms can be defined generically on any collection of comparable objects. The idea is to implement these sorting algorithms once and reuse them for every comparable set of objects.

In this task, you must create two comparable objects (both subclasses of `Comparable`) which override (implement) the `greaterThan(final Comparable other)` method from the base class. Create two subclasses of this base class:

- A `Song` contains a name and a year. Song A is greater than song B if it was produced in a later year than song B, or if it was produced in the same year but comes after it in lexicographic order.[1]
- A `Student` contains a name and an average. Student A is greater than student B if student A's average is higher than student B's average, or if their averages are equal and student A's name comes after student B's name in lexicographic order.[1]

Your `greaterThan()` methods will need to cast the parameters to the specific subtype using, e.g., the following:

```
final Song otherSong = (Song)other;
```

Also, feel free to refer to the variables in the other objects directly without using getters. This is possible, even though they are declared private because this method is part of the same class. This is *not* considered violating encapsulation since this method is part of the same class.

Build your classes using the following program as a base:

```
import java.util.Arrays;

/**
 * Abstract superclass for deriving comparable object classes.
 */
```

[1] See the `compareTo(String otherString)` method in the `String` class.

```
6  abstract class Comparable
7  {
8      /**
9       * @return True iff this object >= other.
10      */
11     public abstract boolean greaterThan(Comparable other);
12
13     /**
14      * @return String description of object.
15      */
16     public abstract String toString();
17 }
18
19 /**
20  * Add your classes here.
21  *
22  * See the instantiation of objects in the main() method (below) for
23  * the order of parameters in the constructors.
24  */
25 public class OOP
26 {
27     public static void bubSort(Comparable[] array)
28     {
29         boolean swapped = true;
30         while (swapped)
31         {
32             swapped = false;
33             for (int i = 0; i < (array.length-1); i++)
34                 if (array[i].greaterThan(array[i+1]))
35                 {
36                     // Swap
37                     Comparable tmp = array[i];
38                     array[i] = array[i+1];
39                     array[i+1] = tmp;
40                     swapped = true;
41                 }
42         }
43     }
44
45     public static void main(final String[] args)
46     {
47         final Comparable[] students =
48         {
49             new Student("Jerry", 8.7),
50             new Student("Jerrry", 8.7),
51             new Student("David", 8.0),
52             new Student("Gloria", 9.0)
53         };
54         bubSort(students);
55         System.out.println(Arrays.toString(students));
56
57         final Comparable[] songs =
58         {
59             new Song("Seven Nation Army", 2003),
60             new Song("The Real Slim Shady", 2000),
61             new Song("Wonderwall", 1995),
62             new Song("Karma Police", 1997)
63         };
64         bubSort(songs);
65         System.out.println(Arrays.toString(songs));
66     }
67 }
```

Assignment 2 ** Due 23:59 Monday 12 November **

This week's assignment builds on the previous Assignment 1. You may continue using your own program or use the solution provided as a starting point.

First, move the two classes, `World` and `Main` into their own separate files: `World.java`, and `Main.java`. Run the `Main` class to make sure it still works after the separation.

Then add a new class abstract class `WorldObject` in its own file `WorldObject.java`. The `WorldObject` class must have the following methods:

- `public abstract boolean isEdible()` returns true if this object is edible, false otherwise.
- `public abstract boolean hasMass()` returns true if the object can fall (due to gravity), false otherwise.
- `public abstract boolean isVulnerable()` returns true if the object gets destroyed if a rock falls on it, false otherwise.
- `public boolean canMove()` returns false. (This is a "default implementation" inherited by all subclasses; it must be overridden to deviate from the default.)
- `public boolean isPlayer()` returns false.
- `public char getMove()` returns the move made by a monster, defaults to `'?'`.
- `public int getEmeraldValue()` returns the number of emeralds this object is worth when eaten, defaults to 0.

Every object in the world will now be represented by a subclass of `WorldObject`. Implement the following classes in `WorldObject.java`. Every class must also override `public String toString()` to return its representation as a single-character string.

- `Space` is a subclass of `WorldObject`. It is edible, does not have mass, and is vulnerable. This object will be treated different than most of the others. Its string representation is a single period (`.`).
- `Rock` is a subclass of `WorldObject`. Rocks are not edible, have mass, and are not vulnerable. A rock's string representation is `r`.
- `EdibleObject` is an abstract class which is a subclass of `WorldObject`. Its sole purpose is to override `isEdible()`, which always returns true.
- `Moveable` is an abstract class which is a subclass of `WorldObject`. Its sole prupose is to override `canMove()`, which always returns true.
- `Dirt` is a subclass of `EdibleObject`. Dirt is edible, has no mass, and are not vulnerable. Its string representation is `#`.
- `Emerald` is a subclass of `EdibleObject`. Emeralds have mass, and are not vulnerable. They are worth 1 emerald each. As before, the string representation is `e`.
- `Diamond` is a subclass of `EdibleObject`. Diamonds have mass, and are vulnerable. They are worth 3 emeralds each. The string representation is `d`.
- `Alien` is a subclass of `Moveable`. The `getMove()` object returns a random direction, as in the previous assignment.
- `Player` is a subclass of `WorldObject`. Players are not edible, do not have mass, and are vulnerable. This object overrides the `isPlayer()` method, to return true. A player's `getMove()` first creates a scanner, reads a single character from the keyboard, and returns the character read.

Now, change the `World` class so that it uses an array of `WorldObject` references instead of characters, like so:

```
1 WorldObject[][] world;
```

And in the constructor, create the objects like so:

```
// First, make them all dirt
for (int row = 0; row < rows; row++)
    for (int col = 0; col < cols; col++)
        world[row][col] = new Dirt();


// then, make the rest as before, except using objects
// ...
world[randRow][randCol] = new Emerald();
// ...
```

Change your `applyMove()` method so that when the player or alien moves over an edible object, it collects the emerald value for it (if it's the player), moves the player / alien there, and changes the place that it's coming from to space, like so:

```
// e.g. toRow, toCol is where the player is going
// fromRow, fromCol is where the player is moving from

if (!inBounds(toRow,toCol))
{
  // walking off the edge of the world? handle appropriately..
}
else if (world[toRow][toCol].canMove() && !world[toRow][toCol].isPlayer())
{
  // player moving into a monster... return end of game
}
else if (!world[toRow][toCol].isEdible())
{
  // do nothing.. destination is not edible!
}
else if (world[toRow][toCol].isEdible())
{
  // decrease the emeralds remaining
  emeraldsRemaining -= world[toRow][toCol].getEmeraldValue();

  if (emeraldsRemaining <= 0)
    emeraldsRemaining = 0;

  // move the player
  world[toRow][toCol] = world[fromRow][fromCol];

  // leave space where the player came from
  world[fromRow][fromCol] = new Space();
}
```

The code for moving the alien will look quite similar to this code for moving the player.

The `toString()` method in the `World` class must also change as well, because now it must call the `toString()` methods for each of the world objects. But, thanks to polymorphism, this will be a very simple change!

Finally, add three random diamonds when creating the initial world, and a few rocks – say four to six – as well. Increase the required number of emeralds by nine. Play the game to make sure it still works as before, except now with dirt, rocks, and diamonds.

Note: You do not have to implement the gravity for the objects with mass, or destroying objects that are vulnerable! You will implement this in a future assignment.

Do not forget to submit all of your source files, or a single compressed/zip archive containing each file.