UNBC Laboratory Assignment
Computer Science 101—Winter 2024

Spike Classes

Purpose

This lab assignment explores the distiction between *state* (the internal representation of an object's information) and *attributes* (the user-visible information that an object contains). It does this by having you build three different implmentations of a `Spike` class suitable for use in a Score Four project.

The lab assignment should also consolidate your understanding of methods, pre-conditions, post-conditions, and packaging.

Outcomes

After completing this assignment, you should

- be able to code with `package` statements;
- explain the effect of adding a `public String toString()` method to a class;
- be able to give an example of two classes with the same attributes and behaviours, but differing state representations.

Due Date

The completed lab assignment is due Monday, 2024-02-12 *by the beginning of lecture*.

Spike Classes — the General idea

The general idea of this lab is to create multiple different implemenations of a `Spike` class suitable for use in a Score Four game. Although the implementations are different, they all have the same public methods.

The point of doing so is to illustrate that the first stage of design should be focussed on *what* a class does, not *how* it is implemented.

Bead Colours and Enum

Spikes contain between zero and four beads that are either black or white. In this lab assignment we represent bead colours by an `enume`. Create an `enum interfaces.BeadColour` that looks like Figure 1 in a separate `.java` file.

(See Special Topic 5.4 in the textbook for a little more information about `enums`. Unlike the book suggestion, you can also place an `enum` declaration in its own file.)

```
package /* lab3.*/interfaces;

public enum BeadColour { WHITE, BLACK ;  }
```

Figure 1: BeadColour specification

Spike Class — version 1a

Write a simple `Spike` class with whose state consists of two private member variables: a BeadColour [4] array, and an `int` height variable. Put your `Time` class in a package called `version1`.

It should have the methods specified in Figure 2 on the next page.

⇒ Write a test class that uses the various methods of the `Time` class to show that they work. When testing this version, the test code and the spike class code should be in different directories, and the test code should contain an "`import version1a.Spike;`" statement.[1] Be sure to test adding a bead to a full column, and asking for the colour the bead at a non-existent location.

⇒ Also test what happens when you convert a `Spike` to a string, as in "`System.out.println("spike 1 is "+spike1);`".

⇒ Also test how equaiity works. Create two separate empty spikes `s1` and `s2`, and see they compare equal with `s1.equals(s2)`.

[1] You may choose to use deeper packaging, for instance, `import lab3.version1a.Spike;`. The same remark applies for all of the versions.

All `Spike` classes should have the following public methods.

- Constructors
  - `Spike()` (creates an empty spike),
- Accessor methods
  - `int height()`,

    The height should be betwen 0 and 4. `null`.
  - `int numberOfBlack()`,
  - `int numberOfWhite()`,
  - `boolean isEmtpy()`,
  - `boolean isFull()`, and
  - `BeadColour getColourAt(int k)`

    The `getColourAt(k)` method should return the colour of the Bead at height k, where $0 \leq <$ height. If k is not appropriate `getColourAt(k)` should return `null`.
- Mutator methods
  - `clear()` and
  - `addBead(BeadColour bead)`.

    If the spike is full, silently do nothing.

Figure 2: Basic `Spike` class methods

Spike Class — version 1b

This class should be the same as `version1a`, except that it should have methods

```
public String toString() { ... }
public boolean equals(Spike s) { ... }
```

⇒ Again test what happens when you convert a `Spike` to a string, as in "`System.out.println("spike 1 is "+spike1);`". Explain your result in comments in the code.

⇒ Also test how equaiity works. Create two separate empty spikes, and see what "`s1.equals(s2)`" returns.

Spike Class — version 1c

Implement an interface like that shown in Figure 3 on the following page.

The class `version1c.Spike` should be similar to `version1b.Spike` but should also the interface `interfaces.Spike`, and any extra methods that that entails.

⇒ Test whether

```
interfaces.Spike s1b = new version1b.Spike() ;
```

compiles. Also test whether

```
interfaces.Spike s1b = new version1c.Spike() ;
```

compiles. Explain your results in comments in the test code.

Spike Class — version 2

The `version2.Spike` class should be nearly identical to `version1c.Spike` (including implementing the interface `interfaces.Spike`).

This sole difference should be that the only instance variables is of type `ArraryList<BeadColour>`, and the changes that causes to various method implementations.

⇒ Again, write a test class that uses the various methods of the `version2.Spike` class to show that they work.

⇒ Test wheter

```
interfaces.Spike s1c = new version1c.Spike() ;
interfaces.Spike s2  = new version2 .Spike() ;
System.out.println(
   "Equailty has "+(s1c.equals(s2) ? "" : "not ")+
   "been achieved.") ;
```

works as expected.

```
package interfaces;

public interface Spike
{
  // heaviours
  abstract public void clear() ;
  abstract public void addBead(BeadColour b) ;

  // queries
  abstract public int height() ;
  abstract public int numberOfBlack() ;
  abstract public int numberOfWhite() ;

  abstract public BeadColour beadColourAt(int i) ;

  abstract public boolean isFull() ;
  abstract public boolean isEmpty() ;

  abstract public boolean equals(Spike another)  ;
}
```

Figure 3: Spike interface specification

Spike Class — version 3

Package this version in a package called "`version3`".

This version should have identical public method signatures and testing, but each `Spike` object should have a single member variable of type `byte`.

Spike Class — version 3 representation

It's perhaps not too surprising that a `Spike` can have a very small state. After all, there are exactly 2^k arrangements of k `BLACK` or `WHITE` beads, and the height k is in the range $0 \leq k \leq 4$, and $2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 2^5 - 1$ or $1 + 2 + 4 + 8 + 16 = 31$. What's perhaps a little more surprising is that there's a fairly direct way to represent each possible `Spike` state by a number between 1 and 31 (inclusive).

The technique involves binary numbers. We write the beads on the column as '0's for white beads, and '1's for black beads from top to bottom. We need a little more as we need to be able to disntinguish between, say, 'WWBW' and 'BW', but the binary numbers 0010_2 (Java `0b0010`) and 10_2 (Java `0b10`) are identiacl. For that reason we insert a leading 1, so 'WWBW' becomes $10010_2 = 18$, and 'BW' becomes $110_2 = 6$.

See Appendix G of the textbook for more information about binary numbers and bitwise operators. We can use bit operations cleverly to implement functions like `getColourAt`.

```
1  public BeadColour baedColourAt(int k)
2      {
3      int testBit = (1 << k) ;
4      boolean inRange = bits >= 2*testBit ;
5      boolean isBlack = (bits & testBit) > 0 ;
6      if (!inRange)           return null ;
7      else if (isBlack)       return BeadColour.BLACK ;
8      else                    return BeadColour.WHITE;
9      }
```

⇒ Again, write a test class that uses the various methods of the `Spike` class to show that they work.

⇒ Can you create a `interfaces.Spike` array that contains a mixture of `version1c.Time`, `version2.Time`, and `version3.Time` `version2c.Time` objects? Comment on the answer in your test code.

⇒ Can you create put a `version3.Spike` in an array of `Version2.Spikes`? Comment on the answer in your test code.

Hand-In Format

In this laboratory assignment, there are multiple places where correctly completing the lab means creating code that does *not* compile. Please be sure to capture the results of each failure, and submit them with the rest of your assignment.

If the failures are compile-time failures, you should be able to capture the failure text from the compiler window in your IDE, or redirect `javac` output to a text (`.txt`) file.

If the failures are run-time failures, you should be able to capture the failure text from the run window in your IDE, or redirect `java` output to a text file.

Put the error outputs in the top-level of the `.jar` or `zip`-file that you submit. You may also create an `answers.txt` file if that helps communicate what you are doing.

There are multiple `packages` that you need to create for this assignment, as described below. Figure 4 on the next page summarizes them.

Here is a list summarizing the packages to create:

version1a A version that uses a `Colour []` array, and an explicit height variable, but no `toString` or `equals` methods (See Figure 2 on page 3).

version1b A version that uses a `Colour []` array, and an explicit height variable, that also has `toString` or `equals` methods.

version1c Like `version1b`, but also implements interface `interfaces.Spike`.

version2 Like `version1c` but uses a single `ArrayList<Colour>` instance variable.

version3 Like `version1c` but uses a single `byte` instance variable.

interfaces contains the `Spike` interface, and teh the `BeadColour` enum.

- various test packages separate from the above.

All of the package names *may* be nested deeper, for instance, `lab3.version2c`.

Figure 4: Packages to create in Lab 3