Unit 5 Problem Set

Part II

I. Overview

Consult the Reges textbook as needed (particularly chapters 8 and 9), and also Oracle's on-line documentation at http://docs.oracle.com/javase/7/docs/api/ . This assignment is designed to reinforce your understanding of how to construct and use class hierarchies through *inheritance*, *abstract* classes and methods, *polymorphism*, and a few other important concepts.

Unless otherwise indicated, everything the *user would type* to the computer has been **underlined**. Everything the computer types is not. Work carefully! The programming problems that are done on the computer will be graded partly on the basis of *style* as well as *correctness*.

Electronic submission of this assignment is due in its entirety prior to midnight on Monday, March 5.

II. Pencil-and-Paper Exercises (20 points total)

[1] 8 points (2 points for each) use file *Cell.txt*

Consider the following classes:

```
public class Cell { … }
public class BloodCell extends Cell { … }
public class RedBloodCell extends BloodCell { … }
```

Which of the following assignments are valid? In a clear and concise sentence, explain *why* the illegal ones are invalid.

- ✦ `Cell c = new BloodCell();`
- ✦ `Cell c = new RedBloodCell();`
- ✦ `BloodCell c = new RedBloodCell();`
- ✦ `RedBloodCell c = new BloodCell();`

[2] 5 points (one-half point for each correct answer) use file *Extends.txt*

Which of the following relationships make sense, and which ones do *not* make sense?

```
(a)  class Oven extends Kitchen
(b)  class Guitar extends Instrument
(c)  class Person extends Employee
(d)  class Ferrari extends Engine
(e)  class FriedEgg extends Food
(f)  class Beagle extends Pet
(g)  class Container extends Jar
(h)  class Beverage extends Martini
(i)  class Metal extends Titanium
(j)  class RollingStones extends RockBand
```

[3] 7 points use file *Pet.java*

Consider the following code which allows a user to keep track of their pet dogs and pet cats:

```
public class PetTest
{
    public static void main (String [] args)
    {
        Pet [] myPets = new Pet [3];

        myPets [0] = new Dog ("Boo-Boo", 2008);
        myPets [1] = new Cat ("Tigi", 2016);
        myPets [2] = new Cat ("Sheldon", 2011);

        for (Pet p : myPets)
        {
            System.out.println (p.getName() + " says "
                                + p.speak() );
        }
    }
}


public class Dog extends Pet
{
    public Dog (String name, int year)
    {
        super (name, year);
    }

    public String speak ()
    {
        return "Woof!";
    }
}


public class Cat extends Pet
{
    public Cat (String name, int year)
    {
        super (name, year);
    }

    public String speak ()
    {
        return "Meoooowww!";
    }
}
```

Define the *abstract* class **Pet** so that running the program **PetTest.java** will correctly produce the output:

```
Boo-Boo says Woof!
Tigi says Meoooowww!
Sheldon says Meoooowww!
```

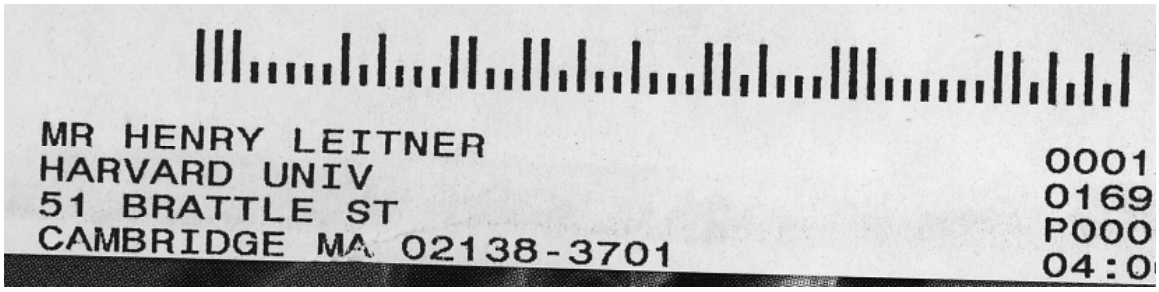
III. Programming Problems (45–85 points)

[4] 45 points *files BarCode.java* and *BarCodeTest.java*

For faster sorting of letters, the United States Postal Service encourages companies that send large volumes of mail to use a *bar code* that denotes the ZIP code.

You will write a program that takes a 5-digit zip code as input, and prints the bar code using the symbol **|** for a full bar and **:** for a half bar. Your program should also be able to take a bar code as input and print the corresponding 5-digit zip code.

Here's a typical mailing label that shows the equivalent bar code, although this one actually uses a 9-digit (rather than a 5-digit) ZIP code:



The solution to this problem will involve developing a class that represents a postal bar code.

Observe that the encoding scheme for a five-digit (or a 9-digit) zip code is as follows:

- Each digit is represented by a sequence of 5 bars.
- The encoding starts and ends with a *Frame Bar*, which is a single full bar.
- The five (or nine) encoded digits are followed by a *check digit*, which is computed as follows:
  a. Add up all the digits
  b. Choose a *check digit* to make the previous sum the next multiple of 10.
  c. For example: The digits in the zip code 02138-3701 (ignoring the hyphen) sum to 25: 0+2+1+3+8+3+7+0+1 = 25. Therefore the correction digit is **5** to make the sum equal to **30.**

Digit encoding: The individual digits (including the check digit) are encoded using the patterns shown in the rows of the following table. In the table a 0 represents a half bar and a 1 represents a full bar. Notice that each digit is encoded with 2 full bars and 3 half bars in some order.

	7	4	2	1	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	1	0	0	0	1
8	1	0	0	1	0
9	1	0	1	0	0
0	1	1	0	0	0

From the above table, you should be able to verify why the bar code shown in

the address label for 02138-3701 contains the precise sequence of full and half bars shown in that label. Do not continue until you have convinced yourself how the bar codes work.

Decoding digits: The decoding process produces the ZIP code digits corresponding to a given barcode. Each digit can be easily computed by a weighted sum of the individual bars. (The weights are 7, 4, 2, 1 and 0.) If a digit's bar code is $b_1\ b_2\ b_3\ b_4\ b_5$, the decoded digit is $7*b_1 + 4*b_2 + 2*b_3 + 1*b_4 + 0*b_1$. This works for everything except the digit 0, which decodes to 11000 and must be handled as a special case. Here are some instances, but remember that each 0 is a half-bar, and each 1 is a full bar:

00011 = 0*7 + 0*4 + 0*2 + 1*1 + 1*0 = **1**

00101 = 0*7 + 0*4 + 1*2 + 0*1 + 1*0 = **2**

00110 = 0*7 + 0*4 + 1*2 + 1*1 + 0*0 = **3**

01100 = 0*7 + 1*4 + 1*2 + 0*1 + 0*0 = **6**

10100 = 1*7 + 0*4 + 1*2 + 0*1 + 0*0 = **9**

Example: 95014 is encoded as | | **:** | **: : :** | **:** | **:** | | **: : : : : :** | | **:** | **: :** | **: : :** | | |

Example: | | **: : :** | | **: :** | **: : :** | **:** | **:** | **: :** | | | **: : :** | **:** | **: :** | is decoded as **78240**.

Here are some details regarding how we suggest you organize your program. If you decide to deviate from this design, please carefully document and explain what you have done. Note that error handling (e.g., in the case of constructors receiving illegal arguments) is a bit problematic. Later on, when we cover *Exception* handling, this sort of situation will be able to get handled more gracefully.

You will develop a **`BarCode`** class to represent a postal bar code and we will supply you with a `BarCodeTest` file to test the **`Barcode`** class, described below.

The BarCode Class

The data fields:

- `String myZipCode` represents the five digit zip code.[1]
- `String myBarCode` represents the 32-character bar code. (You do understand why 32 characters are needed, right?)

Constructor

- Takes a zip code as a parameter and stores it in **`myZipCode`**; then calls the `encode` method to initialize **`myBarCode`**.
  -or-
- Takes a bar code as a parameter and stores it in **`myBarCode`**; then calls **`decode`** to initialize **`myZipCode`**.

  Your constructor will need to figure out whether it has been passed a zip code or a bar code. That's easy to do if you consider the length of the String argument passed.

The accessor ("getter") methods:

- **getZipCode** and **getBarCode**

The private helper methods:

- **digitToCode** takes a digit 0-9 as a parameter and returns the five bars that represent the digit.
- **codeToDigit** takes a five bar code as a parameter and returns the corresponding digit 0-9.
- **isValidBarCode** checks the 32 symbol bar code (a *String* parameter) to determine if it is valid. This method checks for correct delimiters, correct digit patterns and a correct check digit. It returns a boolean value.
- **isValidZipCode** checks its *String* parameter to determine if it is valid, and returns a boolean value.

[1] If we stored this value as an *int* instead of a *String*, the leading 0 (as in 02138) could not be represented.

- **getCheckDigit** returns the integer 0-9 that is necessary for the sum of the digits to equal the next multiple of 10
- **`encode`** tests the 5 digit zip code myZipCode and returns the 32 symbol bar code. It should return the empty string "" if passed an invalid argument.
- **`decode`** tests the 32 symbol bar code myBarCode and returns the 5 digit zip code as a *String*. If the bar code is invalid, this method also returns `""` .

Here is the file BarCodeTest.java for testing your program. The file is available also in the directory ~cscie10/unit5

```
//  BarCodeTest.java
/*********************************
 *  Tester class to grade the BarCode problem
 *  CSCI E-10b, Homework 5, part 2 (2018)
 *
 *  @author   Jan Jackson, modified by H. Leitner
 *  @version  1.2
 **/

public class BarCodeTest
{
    public static void main( String [] args )
    {
     /*  The items in the array below represent the following:
         [0]  valid zip code - 02138
         [1]  valid bar code - 95014
         [2]  valid bar code - 78240
         [3]  bad bar code - bad check digit
         [4]  bad bar code - length 31
         [5]  bad bar code - missing final frame bar
         [6]  bad zip code (contains letter)

        If there are other codes you would like to test,
         include them in the array below.
        When your program is graded, we will test with additional examples.
    */
     String [] tester = { "02138",    "||:|:||:::|:|:||::::::||:|::|:::|||",
                                      "||:::||::|:::|:|:|::|||:::|:|::|",
                                      "||:::||::|:::|:|:|::|||:::||:::|",
                                      "||:::||::|:::|:|:|::|||:::||:::|",
                                      "||:|:||:::|:|:||::::::||:|::|:::||:", "0231a"
                         };
```

```
        for ( int i = 0; i < tester.length; i++ ) {
            System.out.println();
            try {
                  System.out.println( "Creating bar code with " + tester[i] );
                  BarCode bc = new BarCode ( tester[i] );
                  System.out.println ( "\tCode created!   ");
                  System.out.println ( "\tBar code is : " + bc.getBarCode() +
                                       ";  zip is : " + bc.getZipCode() );
            }
            catch ( IllegalArgumentException iae ) {
               System.out.println ( iae.getMessage() );
            }
            System.out.println();
        }
      }
    }

    /*  EXPECTED OUTPUT FOR THIS PROGRAM APPEARS BELOW:

    Creating bar code with 02138
            Code created!
            Bar code is : |||:::::|:|:::||::||:|::|::||::|;  zip is : 02138


    Creating bar code with ||:|:::|:|:||::::::||:|::|:::|||
            Code created!
            Bar code is : ||:|:::|:|:||::::::||:|::|:::|||;  zip is : 95014


    Creating bar code with ||:::||::|:::|:|:|::|||:::|:|::|
            Code created!
            Bar code is : ||:::||::|:::|:|:|::|||:::|:|::|;  zip is : 78240


    Creating bar code with ||:::||::|:::|:|:|::|||:::||:::|
    Illegal bar code values: ||:::||::|:::|:|:|::|||:::||:::|


    Creating bar code with ||:::||::|:::|:|:|::|||:::||:::|
    Illegal bar code values: ||:::||::|:::|:|:|::|||:::||:::|


    Creating bar code with ||:|:::|:|:||::::::||:|::|:::||:
    Illegal bar code values: ||:|:::|:|:||::::::||:|::|:::||:


    Creating bar code with 0231a
    Illegal zip code values: 0231a
    */
```

It is important to note that error-checking should be done by "throwing an exception." For example, if your constructor determines (by calling helper methods such as *isValidBarCode*) that the input is not valid, then it can execute

```
throw new IllegalArgumentException("Illegal bar code value: "
+ theCode );
```

In this example, variable **theCode** is a *String* variable that represents an invalid bar code.

Java's exception-handling mechanism is an elegant way to handle "exceptional situations" that occur at runtime; it lets you put all your error-handling code in one easy-to-read place. It's based on you knowing that the method you're calling is risky (i.e. that the method might generate an exception), so that you can write code to deal with that possibility. If you know you might get an exception when you call a particular method, you can be prepared for — possibly even recover from — the problem that caused the exception. That is why the main method we are asking you to run uses the **try-catch** statements whenever it tries to construct an object (the constructor invokes methods *isValidBarCode* and *isValidZipCode)*.

[5] 40 points

THIS PROBLEM IS REQUIRED FOR ALL GRADUATE-CREDIT STUDENTS only. Undergraduate-credit students may solve this problem for "extra credit."

Design an *abstract* **`Ship`** class that has the following members:

- a field for the *name* of the **Ship** (a *String*)
- a field for the *year* that the **Ship** was built (an *int*)
- a field for the *kind of main engine*. This should be an enumerated type whose possible values are: STEAM_ENGINE, STEAM_TURBINE, GAS_TURBINE, DIESEL, ELECTRIC, WIND, HUMAN_FORCE
- a *constructor* and appropriate *accessor* (getter) and *mutator* (setter) methods
- a **toString** method that displays the ship's name, the engine type, and year it was built

Now design a **Cruiseship** class that extends the **Ship** class. The **Cruiseship** class should have the following members:[2]

- a field for the maximum number of passengers (an int)
- a boolean field that indicates whether the dreaded *norovirus* is currently infecting passengers and causing illness
- a constructor and appropriate accessors and mutators
- a **toString** method that overrides the **toString** method in the base class. The **Cruiseship** class **toString** method should display only the ship's name and the maximum number of passengers.

Next, design a **CargoShip** class that extends the **Ship** class. The **CargoShip** class should have the following members:

- a field for the cargo capacity in tonnage (a double)
- a constructor and appropriate accessors and mutators
- a **toString** method that *overrides* the **toString** method in the base class. The **CargoShip** class **toString** method should display only the ship's name and the ship's cargo capacity.

[2] Try to not think of recent Carnival cruise ships that turned into a floating crisis …

There are various types of cargo ships: *container* vessels, *bulk* vessels (for shipping commodities like wheat and coal), *tanker* vessels (for transporting cargoes like oil), *LNG* carriers (for liquified natural gas), *reefer* vessels (for transporting frozen or temperature-controlled items like meat and fish), and others. You should now extend the **CargoShip** class and implement the necessary methods for one of the above types of cargo ships. Your new class must contain at least one specialized field that is appropriate for the type of vessel (you might need to Google around to learn a tiny bit about these new types of cargo ship).

Finally, you should convincingly demonstrate the above classes in a main program that defines an array of **Ships**. Assign two **CruiseShip**s, two **CargoShip**s and two of your specialized cargo ship type of objects to the array elements.

Thus your array will contain 6 different **Ship** objects. Your program should then traverse the array (using a "for-each" loop) and print out the values in the array by calling each object's **toString** method.

IV. Extra Credit Problem-Solving For Everyone (10 points)

Broaden the Unit 5 example that displays the squares "attacked" after placing a bishop or knight on an 8 by 8 chessboard. You should *extend* the `Piece.java` class so that the program will work for *King* pieces (i.e., create a `King.java` class). Begin by copying the files **Chessboard.java**, **Piece.java**, **Bishop.java** and **Knight.java** from the course Cloud9 directory to your own account.

For those of you who don't already know, a *King* can move just one square in any direction (vertically, horizontally or diagonally).

You will need to make minor modifications to the main program in file `Chessboard.java`.