Project 1

Introduction

This assignment is designed to allow you to practice writing basic class definitions, consisting of fields, constructor and methods. In some of the methods you will need to use if-statements to make decisions.

Core background material can be found in Chapter 2 of the course text book, Objects First with Java.

Please read the assignment description in full to be sure that the work you submit conforms to the full specification.

Task Overview

The scenario is that you are part of a team working on a software system for a company that manages car parks. You have been asked to write a class whose instances can store some basic details about the state of an individual car park and each car park will be monitored by its own instance.

The attributes of an instance are the car park's location, the maximum number of cars it can hold (its capacity) and the current number of cars parked in it (its occupancy). The capacity and occupancy level are whole numbers (integers) and the location is text. The main operations on instances cause the occupancy level to be increased or decreased, the capacity to be changed and the current availability of free places to be printed. More specific details of the class are given below under Detailed Requirements.

You will find that the TicketMachine class discussed in chapter 2 of the course textbook has a lot of functionality that implements similar ideas, so you can use that for ideas to help you write your own class. Aim to study and practice using Chapter 2 as part of your preparation for undertaking this assignment.

Detailed Requirements

- Write a class called CarPark.
- The class must have exactly three fields: two integer fields and one String field. There must be no additional fields defined in the class, even if you think they are required.
- The values of the location and capacity must be received by the class constructor, while the occupancy level must always be set to zero when an instance is created.
- The class must define accessor methods for the location, capacity and occupancy. These must return the current value of the relevant field.
- The value in the field for the occupancy must always be less than or equal to the capacity.
- The class must define a method called `park` that takes no parameters. If the current occupancy is less than the capacity then the occupancy level is increased by 1. Otherwise, an error message must be printed saying that the car park is full and the occupancy is not changed.
- The class must define a method called `leave` that takes no parameters. If the current occupancy is greater than zero then the occupancy level is decreased by 1. Otherwise, an error message must be printed saying that the car park is empty and the occupancy is not changed.
- The class must define a mutator method called `changeCapacity` that allows the capacity to be changed either up or down. It takes a single integer parameter that may be either positive or negative. Normally this must be added to the current capacity regardless of its sign but there are two special cases to consider:
    - If the occupancy is greater than zero and adding the parameter would make the capacity less than the current occupancy level then the capacity must not be changed and an error message must be printed. This case has priority over the following one.

    - If adding the parameter would make the capacity negative then the capacity must be set to zero and a message printed to state that the car park is now closed. This case would only be possible if there are no cars currently in the car park.

- The class must define a method called `printDetails`. It must print details of the car park in exactly the following format:

  ```
  Keimyung St car park has 32 spaces.
  ```

  Where “Keimyung St” is the location of one particular car park and the difference between the capacity and the occupancy is currently 32. Each instance of the CarPark class will have a different location and the details must reflect that.

Please be careful to reproduce both the spacing and punctuation of this format. If there is only 1 space then you may still print “spaces” rather than “space” if you wish.

- Each method must be preceded by an informative javadoc-style comment. Look at how this has been done in the example code provided with the course projects, such as *figures* and *naïve-ticket-machine*, and in the text book and follow that pattern.

- Your code must be neatly formatted and you must follow standard Java guidelines for the methods of classes, variables and methods. The following link has style guidelines to help:

  https://www.bluej.org/objects-first/styleguide.html

  You will also find that the Edit/Auto-layout option in the BlueJ editor will help you to format your code neatly.

This description of the requirements might seem overwhelming at first, so below there is some advice to help you build your final version in small steps. For instance, it will be a good idea to develop the code one field at a time and, initially, it will be easier to ignore the ‘greater than or equal to’ constraints and not worry about checking the values of the parameters. Those details can be added later once the basic fields, constructor and methods are in place.

Remember to try to follow the fundamental principles I have used when coding in lectures: document as you go; write small amounts of code; use the compiler to find and fix errors quickly. Don’t write a lot of code before trying to find the errors, and don’t write more code if you haven’t fixed the errors in the code you have written so far.

Tips for getting started

Start a new project in BlueJ using the Project/New project option. Choose a name and location for the project.

Select the New Class button to start creating a new class. Give your class the name CarPark. BlueJ will create an outline class definition for you in the main area of the project. You will probably find that the source code in the class looks something like the following:

```
/**
```

```
 * Write a description of class CarPark here.
*
 * @author (your name)
 * @version (a version number or a date)
 */
public class CarPark
{
    // instance variables - ...
    private int x;

    /**
     * Constructor for objects of class CarPark
     */
    public CarPark()
    {
        // initialise instance variables
        x = 0;
    }

    /**
     * An example of a method - ...
*
     * @param  y   a sample parameter for a method
     * @return     the sum of x and y
     */
    public int sampleMethod(int y)
    {
        // put your code here
        return x + y;
    }
}
```

BlueJ has given you a sample field, constructor and method, but these are mostly not relevant to this assignment and it is best to delete everything inside the class body and leave yourself with just the following:

```
/**
 * Write a description of class CarPark here.
*
 * @author (your name)
 * @version (a version number or a date)
 */
public class CarPark
{

}
```

Even though this contains no fields, constructors or methods, it is still a valid class definition, so press the Compile button to make sure that you haven’t deleted too much or too little from the code. If you get any compiler errors then read the error

message and see if you can work out what is wrong in the code. Your version should look exactly like the code above.

Add a Field

Instances of the CarPark class will need a String field that can store the location of the car park. Add a field with an appropriate name of your choosing (but avoid meaningless, single-letter names like c, x, y, etc.). Don't forget to include an appropriate visibility modifier for the field. If you are not sure what that is, look at the fields in some of the example classes you have already seen and model your field on them.

Compile the code and fix any errors you see, such as missing semicolons or mis-spelled type names. A good rule to remember is that all Java keywords (like class, int, boolean, public, private, etc.) consist entirely of lower-case letters. However, commonly-used types like String and System are classes and have an initial upper-case letter.

Even though the class has no methods, you can still create objects from it. Do this and use the inspector to show the state of an object. You should see the field you have defined with a value of `null`. At this point, you won't be able to get or change that (default) value because you have no methods to operate on it. Those will come later.

Constructor

Add a constructor to the class with no parameters. In the body of the constructor initialise the location field to have a value of "Keimyung St". Compile your class, fix any errors, then create an object and check that the field has a value of "Keimyung St" as soon as it is created.

In order to set the location in different instances to different values, add a String parameter to the constructor. Use the value of the parameter to initialize the field. Use the TicketMachine class to help you with this if you need to look up how to do this – see the way that different ticket prices can be set. Even though price is an integer field and the location is a String, the principles of assignment from parameters to fields are the same.

Once again, compile the class, fix any errors, create some objects using the new constructor and different parameter values to check that things are working as they must. The thing to look out for is that the location field's value you see when you inspect an object matches the value that you pass in as a parameter to the constructor. If it is still `null` then you need to fix that in the constructor.

The sort of cycle we have been following up to this point – add a little code; compile; fix errors; compile; create objects and test – is a fundamental practice that you should follow whenever you are writing software, no matter whether you are a novice or an expert. One of the advantages of doing things this way is that you catch errors very quickly, and will know roughly where to look for the problem because it will likely have something to do with the code you most recently added or changed. We will take it for granted that you will follow this cycle and won't keep repeating that you should do so.

Accessor Method

Add an accessor method to your class that will return the location of the car park. Accessor methods often have names beginning with 'get' and followed by the name of the field they are associated with; for instance, getTotal for a field called total, getAddress for a field called address, getDate, etc. Give your method public visibility and a return type that is the same as the type of the field. The method must take no parameters. In the method body, there must be a return statement to return the field's value. Once again, take a look at the accessor methods in the TicketMachine class if you need some code to refer to and make the necessary adjustments for this different context. Compile and test your code.

Review

At this point, you should have a class that defines a String field to store the location. It has a constructor to initialize the location and an accessor for it.

Further fields and accessors

Continue by adding a field for the capacity. Initialize it in the constructor and add an accessor. Compile, fix, compile and test. Then add a field for the occupancy and follow the same process. Always use the Inspector for an object to make sure it has been initialized correctly.

The remaining features

One by one, add further methods to the class, always remembering to do things in small increments and to compile and test after each change. In your first versions of the various mutator methods – `park`, `leave` and `changeCapacity` – don't worry about implementing the rules to stop the capacity or occupancy going out of bounds. Just concentrate on the basic arithmetic operations. Once you have those correct you can add the necessary checks – see below.

Using if statements to implement checks

It is now time to add some checking functionality so that the occupancy and capacity fields are protected from being set or modified in inappropriate ways. For this you will need to use if-statements. The basic syntax of an if-statement is:

```
if(test-some-condition) {
    do this part if the condition was true
}
else {
    do this part if the condition was false
}
```

For instance, in a sendText method from a mobile phone class it might be necessary to check whether the phone has sufficient credit, as follows:

```
// Make sure there is enough credit for a text.
if(credit < textCost) {
    System.out.println("You do not have enough credit.");
}
else {
    // Reduce the credit by the cost of a text message.
    credit = credit - textCost;
}
```

Making the Class Robust

Add tests to the methods that change the fields to make sure that the detailed requirements are fulfilled.

When you add new code, be sure to test all methods at least twice so that an error message is output on one test and not output on another, and make sure that fields are not changed inappropriately in the error cases.

Finally

When writing software, it is essential to check that your code meets the requirements if you want to obtain good marks. Unfortunately, this is something that is almost always forgotten and marks are lost unnecessarily. Confidence in the correctness of your code can only be gained through testing it thoroughly. You should, therefore, extensively check your code as you are developing it, as well as when you have finished it. Just because your code compiles does not mean that it operates correctly. Test your code, therefore, by creating objects and calling their methods to cover all of the different cases described above. Although this might seem tedious at times, you will often reveal errors that you had not imagined existed.

Before you submit any assessment, thoroughly test the whole class to make sure that nothing you added later has broken anything added earlier. If you are unable to complete the class – even if you cannot get it to compile – still submit what you have done because it is likely that you will get at least some credit for it.