Java Refresher

Module 4 - Defining Classes (Intermediate)

This is the second of two modules on the topic of writing your own class from scratch. The first module covered basic class definitions with no parameters or return values. In this, the second module, your learning experience will focus on the use of parameters and return values.

Background Material

The Input class

To help you read input from the user, we provide you with a class called `Input`. This class is included in the Module 4 Skeleton Code and provides the following methods:

Javadoc

Class Input

The Input class provides a set of static methods for easily asking the user questions and reading their answers.

Method Summary

Modifier and Type	Method and Description
`static boolean`	`askBoolean(java.lang.String question)` Asks the user the given question, waits for the user to enter a boolean at the keyboard, and then returns this boolean.
`static char`	`askChar(java.lang.String question)` Asks the user the given question, waits for the user to enter a single character at the keyboard, and then returns this character.
`static double`	`askDouble(java.lang.String question)` Asks the user the given question, waits for the user to enter a double at the keyboard, and then returns this double.
`static int`	`askInt(java.lang.String question)` Asks the user the given question, waits for the user to enter a integer at the keyboard, and then returns this integer.
`static java.lang.String`	`askString(java.lang.String question)` Asks the user the given question, waits for the user to enter a string at the keyboard, and then returns this string.

Reference example

Consider the following simple definition for a class of bank accounts:

```
public class BankAccount
{
    private String accountName;
    private double balance;

    public BankAccount()
    {
        System.out.println("Creating bank account");
        accountName = Input.askString("Enter account name:");
        balance = Input.askDouble("Enter account balance:");
    }

    public void show()
    {
        System.out.println(accountName + "'s account has $" + balance);
    }

    public void deposit()
    {
        double amount = Input.askDouble("Enter amount to deposit: $");
        balance = balance + amount;
    }

    public void withdraw()
    {
        double amount = Input.askDouble("Enter amount to withdraw: $");
        balance = balance - amount;
    }
}
```

The constructor initialises the fields by asking the **user**. The `show` method shows the account balance to the **user**. The `deposit` method asks the **user** how much to deposit. And, the `withdraw` method asks the **user** how much to withdraw. These methods work well, but they are also quite limiting in the sense that they require user interaction in order to work. What if someone wanted to use this class in their program, but they wanted to set the initial account name and balance from data retrieved from a data file rather than asking the user? What if this program needed to implement scheduled payments, and regularly deposit or withdraw certain amounts of money at specific dates rather than ask the user? In order to write this program, a different set of constructors and methods would be needed. In Java, the desired flexibility can be supported by adding **parameters** and **return values**.

First, the constructor would become:

```
    public BankAccount(String initialAccountName, double initialBalance)
    {
        accountName = initialAccountName;
        balance = initialBalance;
    }
```

Like the original constructor, this modified constructor initialises the fields. But rather than ask the user what values to initialise the fields to, we allow the calling program to pass in initial values as parameters. The names of these parameters are declared on the first line of the constructor as `initialAccountName` and `initialBalance`. When declaring parameters, it is always necessary to name them and to give them types. If there is more than one parameter, then they are separated by commas. A program wishing to use this constructor would use code such as the following:

```
BankAccount tomsAccount = new BankAccount("Tom Fields", 1000.0);
```

Notice that it is possible to create an new BankAccount with specific initial values without having to interact with the user.

The modified constructor can also be written another way:

```
    public BankAccount(String accountName, double balance)
    {
        this.accountName = accountName;
        this.balance = balance;
    }
```

Notice that this time, the parameter names are not `initialAccountName` and `initialBalance` but are instead identical to the field names. The ambiguity of this apparent variable name clash can be avoided through the use of the `this` keyword. `this` is a reference to the current object. Here, it refers to whatever bank account object is currently being created. `this.accountName` refers to the `accountName` field inside of `this` object. The bare word `accountName` alone, without being prefixed by `this` refers to the parameter. Although this approach is not universally adopted, it is the approach recommended by the inventors of the Java language, and one of its advantages is that you do not have to think hard about coming up with different variable names to avoid name clashes.

Next, let's look at how the BankAccount methods could be modified to not require user interaction. Rather than having a `show` method to print a string to the user, it is common practice in Java to define a method that simply returns a string, and this standard name for this very common method is called `toString`:

```
    public String toString()
    {
        return accountName + "'s account has $" + balance;
    }
```

The first thing to notice is that method header (the first line) declares a return type of `String` rather than `void`. Specifying `String` declares that this method will return a string, whereas specifying `void` declares that this method will not return anything.

This method can be used in two ways. First, a program could simply call this method:

```
System.out.println(tomsAccount.toString());
```

A second, more convenient way to use this method is using Java's implicit `toString` method call:

```
System.out.println(tomsAccount);
```

Whenever Java can figure out from context that a string representation of an object is needed, Java will automatically call its `toString` method.

In addition to the `toString` method, it would be useful to have a method that returns the raw balance itself. The return type for this method would be `double` and the name of the method, by convention, would be `getBalance`:

```
    public double getBalance()
    {
        return balance;
    }
```

This method could be used as follows:

```
if (tomsAccount.getBalance() > 100000.0) {
    System.out.println("Tom has more than $100,000!");
}
```

Finally, let's modify the `deposit` and `withdraw` methods to accept parameters:

```
    public void deposit(double amount)
    {
        balance = balance + amount;
    }

    public void withdraw(double amount)
    {
        balance = balance - amount;
    }
```

These methods permit the calling program to deposit and withdraw specific amounts without needing to ask the user for the amount:

```
tomsAccount.deposit(100.0);
tomsAccount.withdraw(17.0);
```

Exercises

In the provided skeleton code for this module, you will find a `Rectangle` class. While this class compiles and is functional, it has some limitations:

- The dimensions of a rectangle can only be specified by the user at the keyboard. It is not possible to write a program that passes in the desired dimensions as parameters.
- The area of the rectangle once calculated on only be shown to the user. It is not possible to write a program that accesses this calculated area and makes decisions based on this value.

In this module, you will fix both problems with this class by adding parameters and return values.

Exercise 1.

Learning objective: Learn how to define a constructor with parameters. Learn how to define more than one constructor in a class.

Class to edit: `Rectangle`

Locate the following code in your `Rectangle` class:

```
    public Rectangle() {
        System.out.println("Creating new rectangle.");
        width = Input.askInt("Enter width:");
        height = Input.askInt("Enter height:");
    }
```

If you wanted to write a program that creates a new rectangle via this constructor would use the code `Rectangle r = new Rectangle();`. The rectangle would be created with dimensions read from the user.

But what if you wanted to write a program that creates a new rectangle specifically with dimensions 10x5? For this, it would be useful to add a second constructor which accepts the initial width and height as parameters. To do this, your program would use code such as `Rectangle r = new Rectangle(10, 5);` where 10 is the desired width and 5 is the desired height.

In Java, it is possible to provide more than one constructor. Below the existing constructor, define a second constructor that initialises the width and height fields from parameters. The general structure of your code should be:

```
    public Rectangle(int width, int height) {
        INSERT 2 lines here to initialise the two fields.
    }
```

Since this second constructor does not interact with the user at all, there is also no need to print a prompt message such as `"Creating new rectangle."`

Hint: Since the `width` and `height` parameters have the same names as the fields, they will *mask* the fields. Therefore, you will need to use `this` keyword in referring to the fields (See the reference example under Background Material).

Compile your code and if there are no errors, export your project to a jar file being sure to "Include the source files", and then submit this to PLATE to confirm whether your solution is correct.

If your code is incorrect, you may have unlimited attempts to edit your code, re-export to a jar file and resubmit.

Passed? Excellent! As you progress through each exercise below, be sure to continually submit to PLATE to confirm whether your solution is correct. If your solution is not correct, let PLATE's feedback report inform you on what you need to fix.

Exercise 2.

Learning objective: Learn how to define a method that returns a number rather than printing a value.

Class to edit: `Rectangle`

Locate the following method within the `Rectangle` class:

```
    public void showArea() {
        int area = width * height;
        System.out.println("The rectangle's area is " + area);
    }
```

While this method is useful for calculating and showing the area of a rectangle, what if the program using this class wanted to know the area of the rectangle without showing it? There are many reasons why you might want to calculate the area without showing it, but one example is if you wanted to make a decision based on whether area is bigger or smaller than a certain threshold. For this, it would be useful to define another method that calculates the area but rather than printing it to the screen, simply returns the area to the caller, to be used by the caller for whatever its intended purpose happens to be.

Below the `showArea` method, define another method called `calculateArea` which takes no parameters. This method should calculate the area as an integer and return it. Since the area of this rectangle will always be an integer, the return type of your method should be `int`.

Hint: When defining this method, where you would normally write `void`, instead write the type of value that this method is returning. Then inside the method, rather than using a `println` statement to print the area, use a `return` statement to return the area.

Example of usage:

```
Rectangle rect = new Rectangle();
int area = rect.calculateArea();
if (area > 10)
    System.out.println("This rectangle is big.");
else
    System.out.println("This rectangle is small.");
```

Exercise 3.

Learning objective: Learn how to define a method that returns a string rather than printing a string.

Class to edit: `Rectangle`

Locate the following method within the `Rectangle` class:

```
    public void show() {
        System.out.println("The rectangle has dimensions " + width + "x" + height);
    }
```

Below this method, write another method called `toString` that instead of printing the rectangle's dimensions as a string, returns the rectangle's dimensions as a string. Because this method is not interacting with the user, there is no need to produce such a verbose string as `"The rectangle has dimensions 10x7"`. Instead, just return the dimensions string "10x7" itself (where `10` and `7` should be replaced by the rectangle's actual dimensions).

Note: In Java, it is standard practice to include a `toString` method in every class. `toString` methods are useful because the returned string can be used for a number of different purposes other than simply showing the user: e.g. saving to a formatted text file or printing to program log files.

Example of usage:

```
Rectangle rect = new Rectangle(10, 3);
System.out.println("The rectangle dimensions are: " + rect.toString());
```

Because `toString` is a standard method in Java for producing a string representation of an object, the Java language has a special built-in feature to invoke a `toString` method on an object implicitly when it is clear from context that a string representation of the object is needed. For example, the following program is equivalent to the above:

```
Rectangle rect = new Rectangle(10, 3);
System.out.println("The rectangle dimensions are: " + rect);
```

Exercise 4.

Learning objective: Learn how to define a method that takes parameters.

Class to edit: `Rectangle`

So far, you have defined a constructor that takes parameters, and you have defined some methods that return values. You should also hopefully understand why parameters and return values are useful.

But so far, the `Rectangle` class does not provide a method that takes parameters. A useful example of this would be a method that allows you to change the dimensions of an already existing rectangle.

Write a method called `changeDimensions` accepting the new width and new height as parameters. This method should, using two lines of code, update the `width` and `height` fields to the new values supplied via the parameters.

The general structure of your method should be as follows:

```
    public void changeDimensions(COMMA SEPARATED LIST OF PARAMETERS HERE) {
        WRITE TWO LINES OF CODE HERE
    }
```

Hint: The code would essentially be the same as the constructor that you wrote in Exercise 1, except in the form of a method. Unlike the constructor which is absent a return type, your method should have a return type of `void`.

Example of usage:

```
Rectangle rect = new Rectangle(2, 3);
rect.changeDimensions(1, 4);
```

Exercise 5.

Learning objective: Put your `Rectangle` class to use.

Class to edit: `Exercise5`

Double click on the `Exercise5` class and within its `main` method, write a program that uses your Rectangle class, according to the following steps:

1. Create a new rectangle with dimensions 8x4 (i.e. width 8 and height 4) and store it into a variable called `rect`.
2. `System.out.println("The initial rectangle dimensions are " + rect);`
3. Change the dimensions of the rectangle to 6x5.
4. `System.out.println("The new rectangle dimensions are " + rect);`

5. Obtain and store the area of this rectangle into a variable called `area`.
6. If the area is greater than 10, print "This is a big rectangle."
7. Otherwise, print "This is a small rectangle."