Java Refresher

Module 3 - Defining Classes (Basic)

After completing this module, you will have written your own simple class from scratch, and will also have put this class to use in a test program.

Background Material

This is the first of two modules on the topic of writing your own class from scratch. In this first module, you will gain experience using a basic subset of features that may be used when building a class, namely: fields, constructors and methods. In the second module, you will learn how to add parameters and return values to your constructors and methods.

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 3 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

This section will provide you with a simple reference example that should help to guide you while you attempt the exercises below.

Example: Your company is developing software for a bank, and you have identified that you would like to store data about each bank account. Each bank account has an account name and a balance (a real bank account would have more data than this!). In the system, you want to be able to have many bank accounts. In terms of Java concepts, what we have here is a **class** of bank account **objects**:

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

A Java class is a blueprint from which many objects can be created. Objects are related to the classes from which they were created. When an object is created from a particular class, we say that that object is an **instance** of that particular class. Every instance of the particular class above will store two pieces of data: an account name, which is of type `String`, and a balance, which is of type `double`. In Java, these two pieces of data are referred to as the **fields** of the object. The decision to use type `double` is based on the fact that money amounts can have two decimal places. However, in a real banking application, this is probably not a good idea, since the `double` type is only an *approximate* datatype and its values become less precise as they become bigger. Although we will not use it here, BigInteger would be a more precise choice of datatype.

We need to provide a way to create new bank accounts, and that is what a **constructor** is for:

```
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:");
    }

}
```

The **primary** responsibility of the constructor is to initialise each of the data fields. In the example above, we assign initial values by asking the user. In the sequel to this module (Defining Classes - Intermediate), you will learn the alternative approach of passing constructor parameters to supply the initial values.

Next, we need to provide various features to make this bank account actually useful. How would we like to use the bank account? We would like to be able to see how much money is in the account, we would like to deposit money and we would like to withdraw money. In terms of Java concepts, these features are implemented as **methods**:

```
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;
    }

}
```

Each method is of the form:

```
    public void METHODNAME()
    {
        CODE GOES HERE
    }
```

Methods will typically read and/or write to the various data fields defined at the top of this class.

One important "gotcha" to avoid when writing your class is to make sure that all fields, constructors and methods are defined **inside** of the class declaration. For example, the following code will not compile because one of the methods was defined outside of the class:

```
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;
    }
```

When this happens, the compiler gives error message that is both confusing and yet accurate:

```
error: class, interface, or enum expected
  public void withdraw()
         ^
```

What the compiler is saying is that at this location in the program, i.e. after the end of the class, the only kind of thing that should be expected is another class (or an interface or an enum, but we won't worry about those), and certainly not a method.

A range of other mysterious compile errors can also occur when code is not nested under the correct parent section. Therefore, if you encounter a compile error in your class that is not easy to understand, it is a good idea to check that blocks of code surrounded by "`{`" and "`}`" braces are nested under the correct parent container. In this case, methods should be contained within classes. It is also good to double check that every open brace "`{`" has a matching brace "`}`" to close off that block.

Exercises

In this module, you will incrementally write from scratch a class called `Rectangle`. No skeleton code is provided for class `Rectangle`, since it is a class written entirely by you. In Exercise 6, you will write a simple program making use of `Rectangle`, and for this, a skeleton class `Exercise6` is provided.

Exercise 1.

Learning objective: Learn how to define a simple class with fields.

Class to edit: `Rectangle`

Since no `Rectangle` class is included in the skeleton code, your first step is to create a new class named `Rectangle`. In BlueJ, you can accomplish this by clicking on the "New Class..." button. **Don't forget:** class names should begin with an uppercase letter, so make sure you type the class name as `Rectangle`, *not* `rectangle`.

After creating this class, double click on it to open the code editor. Select all code in this file and DELETE it. Your objective in this exercise is to write a class from scratch!

Now, you are to write code to declare a class named `Rectangle`. Inside this class, you are to declare two fields named `width` and `height`, both of which should have type `int`.

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: Write a constructor.

Class to edit: `Rectangle`

Define a constructor for class Rectangle that initialises all fields by asking the user for values. The constructor should behave according to the following sample I/O:

```
Creating a new rectangle.
Enter width:     « user inputs 7 »
Enter height:    « user inputs 10 »
```

Your constructor should contain three lines of code:

1. Print the message `"Creating a new rectangle."`
2. Using the supplied `Input` class, ask the user for the width and store it into the `width` field. Careful! Do not declare a new local variable called `width` (e.g. `int width = Input.askInteger("Enter width:"));` because that will *mask* the field with the same name. To store into the `width` field without creating a new local variable, just remove the word `int`.
3. Using the supplied `Input` class, ask the user for the height and store it into the `height` field.

Exercise 3.

Learning objective: Learn how to define a method that uses a local variable.

Class to edit: `Rectangle`

Define a method for class Rectangle called `showArea` that calculates and prints the area of the rectangle. You should use 2 lines of code:

1. Calculate and store the area into a local variable.
2. Print `"The rectangle's area is xxx"` where `xxx` should be substituted by the area that you calculated in the previous step.

Exercise 4.

Learning objective: Concatenate multiple fields into a formatted string.

Class to edit: `Rectangle`

Define a method for class `Rectangle` called `show` that prints the width/height dimensions according to the following sample I/O:

```
The rectangle has dimensions 10x7
```

where (in this particular sample) `10` is the width and `7` is the height.

Hint: You will need to "string together" this output string from 4 parts using code such as part1 + part2 + part3 + part4. For example, part2 will be the current contents of the width field, and part3 will be the "x" that appears between the width and the height.

Exercise 5.

Learning objective: Learn how to define a static method.

Class to edit: `Rectangle`

Define a method for class `Rectangle` called `showNumberOfSides` which shows the number of sides of the rectangle, according to the following sample I/O:

```
Rectangles have 4 sides
```

Because all rectangles have 4 sides regardless of the specific instance, this method should be declared as a static method. Note that this should be the **only** use of the `static` modifier in this class.

Exercise 6.

Learning objective: Put your own class to use.

Class to edit: `Exercise6`

Double click on the `Exercise6` class and within its `main` method, write a program that uses the Rectangle class that you just defined, according to the following steps:

1. Show the number of sides that rectangles have.
2. Create a new rectangle.
3. Show the rectangle
4. Show the rectangle's area