Introduction

Time to go over some more advanced Java so you can get started on creating the Snake game afterwards! These exercises are marked in the same way as the first ones, so you complete your answers in **Exercise2.java,** and the code gets automatically marked when you run that class. You **don't need to change any code in TestRunner.java**, except from when explicitly asked to during question 4.

Instructions:

- Grab 'Exercise2.zip' from off the Engine
  - Extract the zip file
- Import the project into Netbeans
  - Instructions for how to do this are on last week's worksheet
- Go through this worksheet as you work through the exercises, this document will cover the syntax and include explanations of what is happening.

- **Important:** When you import the project, there will be a compile error in the TestRunner class, as the method 'question1()' doesn't exist yet. This is fine, as in Question 1 you will be creating that method!

Objects and Classes

In Java (and other Object-Oriented languages, e.g C++), there is an important distinction between Objects and Classes:

- Classes are the files you write (e.g Board.java)
- Objects are **instances** of classes
  - So when you create a new variable such as
    `Car car = new Car();`
  - You actually create a Car object, which is an **instance** of the Car class

Constructor

When you create an object, you invoke (or call) the **constructor method** in the class you wrote. The constructor is **always** has the same name as the class.

This means that anything you write in the constructor is executed when you create the object. A lot of the time this is where we set up any variables we need in the class.

An example of an Animal class with a constructor method is shown below:

```
3   public class Animal {
4
5       public Animal () {
6           // Do something here
7       }
8   }
9
```

However, often we want to be able to pass some kind of data to the object, but having a constructor like above doesn't allow us to do this.

Constructor with arguments

Instead we can add **arguments** to the Constructor. Arguments are basically like variables, so when you call the constructor you pass in an actual value, and then in the constructor you can access this value by using the name you give you to the argument:

```
3   public class Animal {
4
5       public Animal (String species) {
6
7           System.out.println(species);
8
9       }
10  }
11
```

In this example, the Animal constructor method takes a String argument called 'species'. We can then use this as we would any other variable in the method.

For example, if we wanted to create an Animal object for a horse:

```
3   public class AnimalTest {
4
5       public static void main(String[] args) {
6
7           Animal horse = new Animal("Horse");
8
9       }
```

When the object is created and the object's **constructor method** is run, the value "Horse" is passed in (and accessed using the `species` argument), so the Animal object will print out "Horse" to the terminal.

Fields

We've briefly covered what is called **scope** in Java, when we talked about the difference between public and private. Scope, in this sense, basically means whether a variable or method can be seen by another method.

For example a private method only has a visible scope to other methods in that class, i.e a method in another class cannot see it.

Variables can also have scope to just a particular method, if we look at the previous example:

```
3   public class AnimalTest {
4
5       public static void main(String[] args) {
6
7           Animal horse = new Animal("Horse");
8
9       }
```

The 'horse' variable can only be accessed inside the main method. This is fine for a lot of variables because they aren't needed in other places.

However, some things **do** need to be accessed by multiple methods. Java allows us to do this by creating **fields**. A field is a private variable, defined in the class, **outside** any other methods. We can assign the species value in the Animal class to a field as follows:

```
3   public class Animal {
4
5       // This is a field called species
6       private String species;
7
8       public Animal (String species) {
9
10          // Here we assign species to the field
11          this.species = species;
12
13      }
14  }
```

Here we define the field called species, but we don't put any value in it.

Here we put the value in the species field. Another way of saying this is that we **instantiate** the field.

Adding a method to a class

To add a method to a class, you can declare it in the same way that you declare the constructor, but remember to include a **return type.** Make sure that the method is declared outside of any other method, but inside the class:

One example might be if we wanted to be able to retrieve the species of an Animal object. To do this we can create a 'getSpecies()' method that returns a String. Inside the method we then want to return the value of the species field.

```
3   public class Animal {
4
5       // This is a field called species
6       private String species;
7
8       public Animal (String species) {
9
10          // Here we assign species to the field
11          this.species = species;
12
13          public String getSpecies() {
14              return species;
15          }
16
17      }
18  }
```

This is wrong because the method is declared **inside another method!**

If you suddenly get a lot of compile errors like in this screenshot, it usually means there is a problem with a set of brackets not matching up properly, and the compiler gets confused.

```
public class Animal {

    // This is a field called species
    private String species;

    public Animal (String species) {

        // Here we assign species to the field
        this.species = species;

    }
}

    public String getSpecies() {
        return species;
    }
```

This is wrong because the method is declared **outside the class.**

```
3   public class Animal {
4
5       // This is a field called species
6       private String species;
7
8       public Animal (String species) {
9
10          // Here we assign species to the field
11          this.species = species;
12
13      }
14
15      public String getSpecies() {
16          return species;
17      }
18  }
```

This is correct, because the method is declared **inside the class**, but **outside of any other methods.**

Arrays

An array is sort of like a container, or a way of grouping different values or objects of the same type. This is similar to Python's list.

As an illustration, an array with 10 int values would look like this:

Index:	0	1	2	3	4	5	6	7	8	9
	12	-3	1	8	17	4	20	0	-1234	8

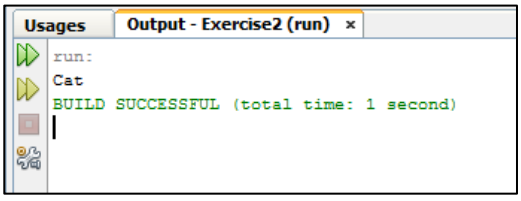
The index you can see above each value relates to how we access or store things into the array. To define an array, the syntax is as follows:

```
Animal[] animals = new Animal[3];
```

- The square brackets next to the data type (Animal) tells Java that this will be an array made up of Animal objects.
- The rest of the syntax is similar to normal Objects, but you must include the number of items in the array at the end (in this case 3).

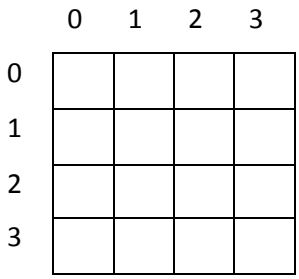
Arrays are **zero-indexed**, meaning that to access the first item in the array, it is 'assigned' the index 0 in the array. To store anything, or access anything stored in the array, you will use this index. The example below shows the Animals array being populated (with 3 Animal objects), and then the second item in the array (located at index 1!) is printed out.

```
public class AnimalTest {

   public static void main(String[] args) {

      Animal[] animals = new Animal[3];

      // Populate the animals array
      animals[0] = new Animal("Horse");
      animals[1] = new Animal("Cat");
      animals[2] = new Animal("Giraffe");

      Animal secondAnimal = animals[1];
      System.out.println(secondAnimal.getSpecies());

   }
}
```



2D Array

The array we looked at previously is called a one-dimensional array, but you can also have a two-dimensional array, which is essentially like a grid:

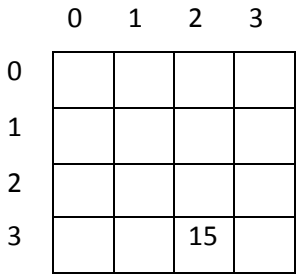


You declare this type of array as follows:

```
1 int[][] numGrid = new int[4][4];
```

- Notice you now need two sets of square brackets
- You also need to include two numbers, one for the x dimension, and another for the y dimension.
- In this example both dimensions are the same, but they don't have to be

Below is an example of storing a value in the 2D array. In the context of the snake game, we will store the values as follows: row index, column index (this is the opposite way round to normal x, y co-ordinates). Here we will store 15 in row 3, column 2:




```
public class AnimalTest {

   public static void main(String[] args) {

      int[][] numGrid = new int[4][4];
      numGrid[3][2] = 15;

   }
}
```

Question 1

This question asks you to create your own method! Until you have completed this question, you won't be able to auto mark the exercise. If you want any more information about how to create a new method, check the section above titled 'Adding a method to a class'.

Instructions:

- Underneath the comment for Question 1, create a new method with the following properties:
  - It is **public** and **static** (these are called modifiers)
  - It should have a return type of **boolean**
  - It should be called '**question1()**'
  - You should return **true** inside the method.

Question 2

This question is about **arrays**! In this question you will declare your own array of integers. There is a section above that contains information about arrays if you need help. If you still aren't quite sure what an array is, feel free to ask!

Instructions:

- In the question2 method, create a new int array with space for 10 items
  - Remember the syntax is **type[] name = new type[number of items];**
- Place the value 6 in the fourth space
  - Remember that's index 3!
- Return the array

Question 3

This question is about **2D arrays!** There is more information in the Arrays section above. In this question you will declare a 2D array of integers.

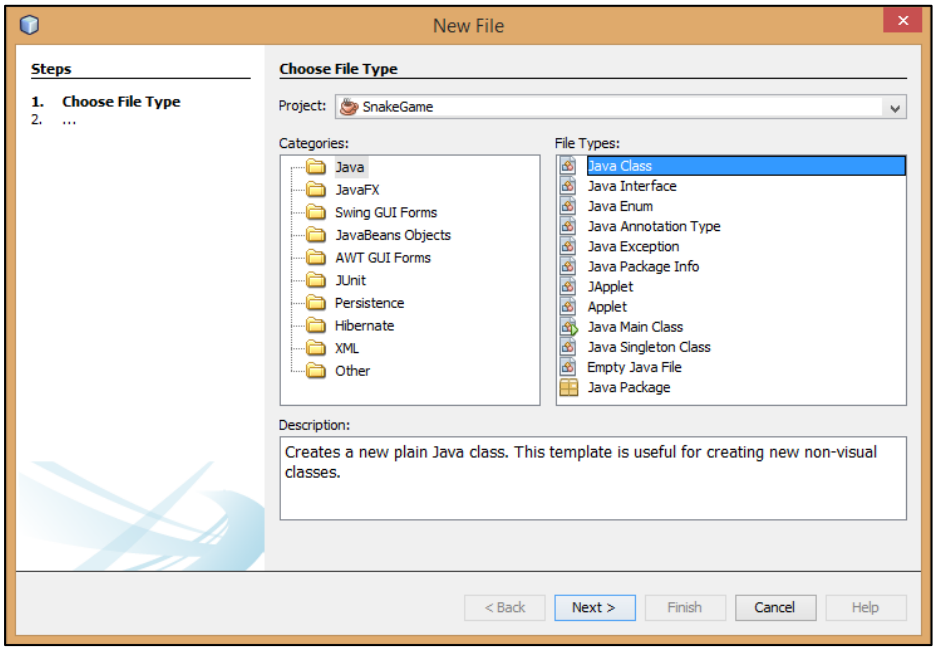
Instructions:

- In the question3 method, create a new 2D int array with space for 4x4 items
  - Remember the syntax is:
    type[][] name = new type[number of rows][number of cols];
- Place the value 15 at the position: row 3, column 0
- Place the value -3 at the position: row 1, column 2
- Return the array

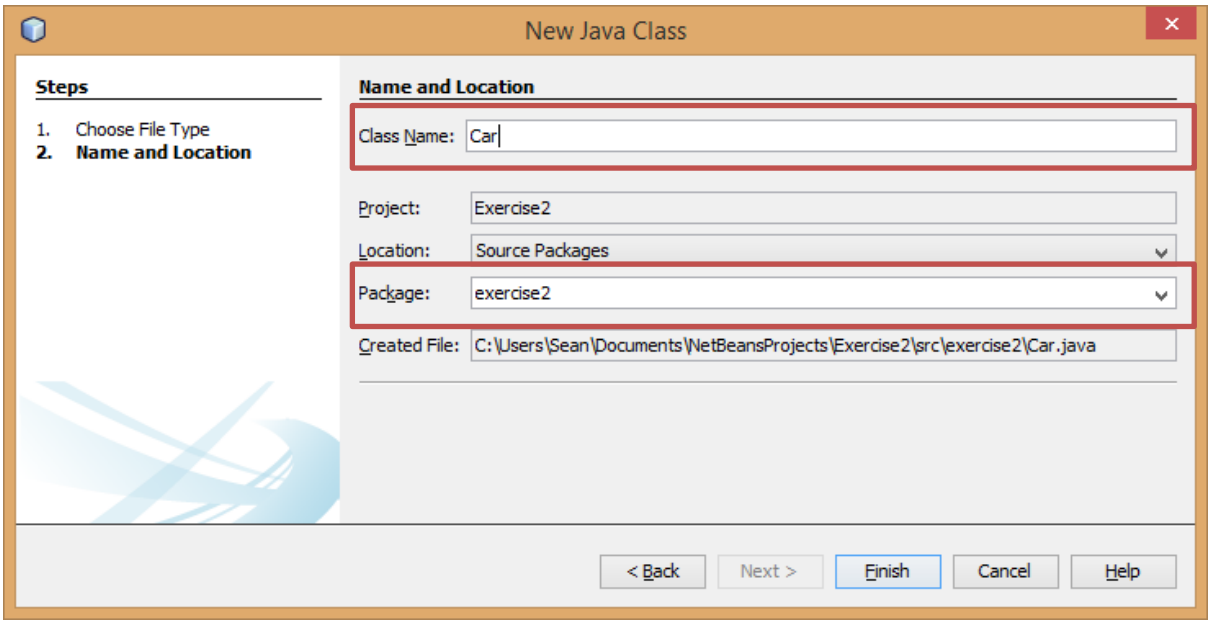
Question 4

This question asks you to start creating your very own class. Remember a class is the file with the .java extension that you write your code in. A class starts with the code `public class ClassName`, then opening and closing curly brackets that contain all the rest of the code.

To create a new class in Netbeans, go to File > New File… and select Java class:



On the next screen, call the new class Car, and make sure package is exercise2 – this last bit is important.



Instructions:

- First, if you haven't already done so, create the Car class as shown above.
- **IMPORTANT**: Go into the TestRunner class, and delete the private Car class at the bottom of the file:

```
122     /**
123      * This is a dummy class so that the code compiles anc
124      * the test. Question 4 asks you to remove this privat
125      * the code)
126      */
127     private class Car {
128
129     }
130
```

In the **Car** class:

- Write a constructor method that doesn't take any parameters
  - You don't need to put anything in between the brackets

In the **question4** method:

- Create a new Car object
- Return the Car object

Question 5

In this question, you will extend the Car class you created earlier, by adding another constructor method! This shows an important feature in Java called **overloaded methods**. This means that you can have multiple methods with the same name, as long as they have different arguments, or a different number of arguments. This means we can have a default constructor (that you wrote previously), and also a constructor that takes arguments!

In the **Car** class:

- Write a new constructor that takes a String argument, you can call the argument 'colour'

In the **question5** method:

- Create a new Car object, passing in any colour you like.
- Return the Car object

Question 6

Now that you've added a parameter to the Car class (giving us a way of passing some data to the Object when we create it), it's time to use that information. Remember that since we declare fields as private, they can't be accessed by other classes. To enable us to access a field from other classes, we usually write a method in the format **getVariable()** (e.g getColour()) that returns the value;

You will assign the colour variable to a field (there is information at the top of the document about fields), and create a method to return that value!

In the **Car** class:

- Create a new String field called colour
- In the constructor, assign the value of the 'colour' argument to the field of the same name. The syntax for this is:
  - **this.colour = colour;**
- Create a new method called **getColour**
  - It shouldn't take any arguments
  - The return type should be String
  - Return the value of the colour field.

In the **question6** method:

- Create a new Car object, passing in **"red"**
- Return the value of calling getColour() on the new Car object

Question 7

Great! Now we can get the value of the colour field, but we can't set it yet…. Can you guess what we'll do in this question? The conventional format in Java is **setVariableName**, e.g setColour.

In the **Car** class:

- Create a method called setColour
  - It should take a String argument (you can call this colour)
  - The return type should be void
- In the method, assign the value passed in to the colour field you created earlier.
  - E.g **this.colour = colour;**

In the **question7** method:

- Create a new Car object, passing in **"red"**
- Call the setColour method on the car object, passing in "**black**"
- Return the value of calling getColour() on the Car object

Question 8

In this question we'll cover **enumerated types**, or **enum**s. Enumerated types are (in a sense) a way to restrict the possible values you can have, and is very useful if you have a small number of options, for example: compass directions can only be North, East, South or West. In Java we can code this as:

```
public enum Direction {
    NORTH, EAST, SOUTH, WEST
}
```

An enum is (normally) declared like a method, i.e inside the class, but outside of any other methods.

It is also possible to create your own enum class, which you will be asked to do later in the Snake game, but we won't cover just yet.

In this question we will use an enum to restrict the possible species that can be used in our Animal class, so (for example), you could only have horses, cats and fish.

By convention, in Java, enum values (and constants in general) are all uppercase.

Now the enum exists, you can use the use the enum type as you would any other, allowing you to create variables of the type 'Direction'.

```
Direction dir = Direction.NORTH;
```

In the **Car** class:

- Create an enum to store the different colours you would like to allow. Make sure you include RED. A sensible name is ColourType
- Change the 'colour' field to be of type ColourType
- Change the constructor which takes a parameter:
  - Change the type of the parameter to ColourType
- Change the return type of getColour to return an ColourType object
- Change the type of the setColour parameter to ColourType

In the **question8** method:

- Create a new Car object, passing in a ColourType parameter of RED
- Call getColour on your new object, assigning the value to a new variable
- Return the value of calling toString() on the ColourType variable
  - toString() is a method that all objects in Java have, that is used to return a human-readable representation of the data in that object.