



# PROGRAMMING I

M1I322909

# I. INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Introduction .....	2
Objects .....	4
Classes .....	4
Creating objects .....	5
Inspecting an object.....	7
Calling a method .....	8
Data types.....	8
Return values .....	9
Programming objects .....	10
Object interaction.....	10
Source code.....	11
Wrap up.....	13

## Introduction

The difference between a computer and other devices is that a computer can be **programmed**. For example, a very basic toaster is not a computer – it contains a simple electrical circuit designed to do one job only: toasting bread for a set length of time. A computer, on the other hand, can be instructed in principle to do any job. The designer of the computer does not know what jobs it will be required to do after it has been built.

### Programs

To get a computer to do a job, you need to give it instructions in a language that it can understand. Computers are able to understand, a range of languages that have more limited vocabulary and stricter rules than the natural languages that humans speak. These are known as **programming languages**. The language you will be learning to use in this module is called **Java**.

You often want a computer to do the same job over and over again. Most jobs require a sequence of many instructions to be carried out. Rather than requiring a user to issue these instructions every time, it is better to assemble them into a collection of instructions, known as a **program**, which can be **executed**, or **run**, each time the job needs to be done. The text that defines the instructions is known as program **code**. Programs are often known as **software**, in contrast to the physical components of a computer, which are known as **hardware**.

Programming is the process of developing a program by writing code in a programming language, and is one of the key skills needed by a Computing professional.

### Computers everywhere

In the early 1960's there were a few thousand computers in the world, and these were large systems that would each fill an entire room. Nowadays, as well as the billions of personal computers and laptops which are in use, computers are all

around us in places which are not all obvious. Many people carry phones which are actually sophisticated computers which can run a huge range of programs, or **apps**. Computers are embedded in many everyday products, such as cars, washing machines, televisions and so on (including some more expensive digital toasters!), and these computers have been programmed to provide ways of controlling the functions of these items. When you browse the web, for example to shop online, you are interacting, not just with the device you are holding or sitting in front of, but also over the internet with powerful computers known as **servers** which are programmed to provide the functions of the web sites you use.



In all these cases, the computer will not do anything useful unless it is programmed to do so. For example, when you use the on-screen schedule guide on a modern television you are running a program which a programmer has written to instruct the computer that is built into the television.

### Computers and programs

Every computer is designed to provide a basic set of capabilities:

- Executing instructions
- Storing information needed while executing instructions
- Communicating with external devices, such as keyboards, screens, permanent storage devices, networks

A programming language lets you, as a programmer, write program code which makes use of these capabilities to perform some useful function. You are likely to learn about the components that computers have in order to provide these capabilities elsewhere in your course (for example in the module Fundamentals of Computing).

### Software engineering

A program can be just a few lines of code that perform a simple task. However, many of the programs that are written today are very complex. Building complex software is in some ways like building a complex physical structure such as a bridge or an aircraft, and should be done similarly using an engineering process involving: analysing what the software needs to be able to do; designing a solution to meeting those needs; implementing the software according to the design; and testing to make sure it all works.

This process is called **software engineering**, and you may learn more about this elsewhere (for example in the module Fundamentals of Software Engineering). Programming is essentially one – a very important aspect! – of the process of implementing software. However, programming is also closely related to other parts of the software engineering process, and in this module you will learn how to apply some important software engineering techniques.

## Objects

Java, like many of the most widely-used languages, is an **object-oriented programming language**, and a program written in Java is an object-oriented program. This means that Java allows you to create in the computer a **model** of some part of the world. This model is built up from **objects**, and is at the core of an object-oriented program.

Objects in many cases represent “things” which exist in the real world. For example, a program that allows customers to buy books online will use objects representing customers, books and orders. A word processor program will deal with objects representing words and paragraphs. A computer game may deal with objects representing characters and scenes.

When an object-oriented program is running, it **creates objects** which do their own jobs and also work together, or **collaborate**, to perform the required actions. In the real world there are many examples of components that work together like this, whether it is parts of a machine or people in a team. Just like in the real world, the success of a program depends on all the objects doing their job, and also, crucially, on the communication between them.

## Classes

An OO program creates the objects it needs as it runs. Often there will be more than one object of the same **type** created. A game may have several characters of the same type, for example, and a word processor will need to deal with many words. These objects will be similar, and will be able to do the same things, but will also have **properties** that distinguish each one from others of the same type.

In order to know how to create an object of some type, the program needs to have a **template** which specifies what an object of that type can do. This template is called a **class**. In fact, when you write the code for a program, you are actually writing the classes that are used to create objects. **An object is a single instance of a class** – there can be many objects of the same type.

A class specifies the following for the objects it is used to create:

- The **name** of the type of object (e.g. Customer)
- The **properties** which each object will have (e.g. name)
- The **actions** which each object can perform (e.g. change password)
- The way in which each object is **linked with other objects** (e.g. with Order objects)

## Working with objects

Let's see how this works by creating and working with some actual Java objects. For now we will use classes that have already been written – later on you will learn to write your own classes in Java. Note that the steps you follow here introduce quite a few of the really important ideas of object-oriented programming, and there is a lot to take in! Don't worry, we will look again at all these ideas and how you make use of them to write your own programs as we go through the module.

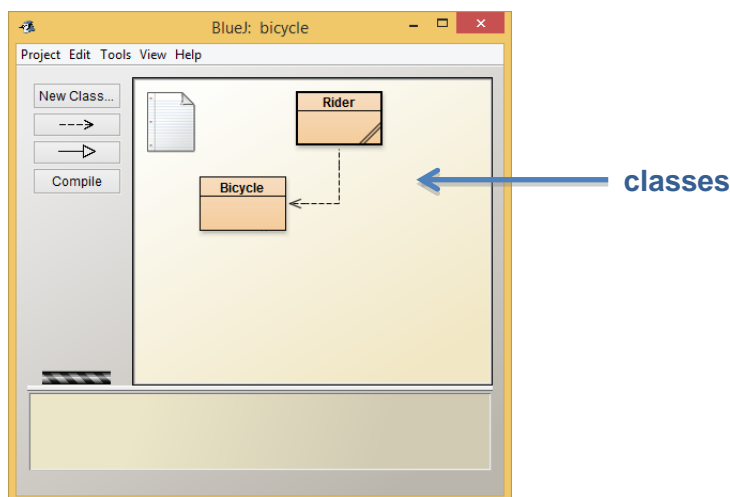
### Using BlueJ

You will use the **BlueJ Integrated Development Environment** (IDE) throughout this module to create objects, classes and programs in Java. To get the most out of reading these notes you should **follow through the example activities with BlueJ** as you do so.

#### GETTING HOLD OF BLUEJ

*BlueJ is installed on the PCs in the computer labs on campus (it may be in a virtual machine). If you want to use it on your own computer it is easy to install. You will also need to install the Oracle Java SDK which BlueJ needs to create and run Java programs (and to run at all, as it is a Java program itself). You can find the installer and full instructions at [www.bluej.org](http://www.bluej.org)*

With BlueJ your work is organised in **projects**. A project called *bicycles* has already been created (you can download this from GCU Learn as a ZIP archive and extract the contents). A project is contained in a folder and looks like this when you open it in BlueJ using the Project>Open Project menu option and browse to the project folder.



The main area of the BlueJ window shows the classes that belong to the project, in this case there are classes called `Bicycle` and `Rider`. You will see shortly how these classes were created, but for now we'll just use them.

## The example classes

The presence of a `Bicycle` class in the project means that objects of that type can be created (and the same applies to `Rider`). But why would we want to create a bicycle object in a program? As we said earlier, objects often **model** some part of the world. What do we mean by “model”? An object which models a real-world “thing” (or entity) should behave in some way like the real thing. It doesn’t have to do everything that the real thing can do, but needs to model the **behaviour that is relevant to the program** that will need to use the object.

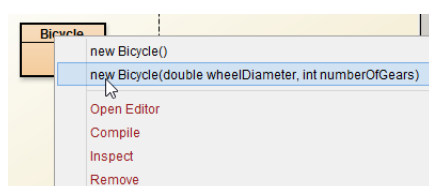
Software objects representing bicycles might be useful in, say, a bike simulator program or a game program that involves bikes. The user or player would use some input device to give commands corresponding to the actions, changing gear for example, of a real rider on a bike, and the “model” bike should respond like a real bike would to those actions.

Not everything about a real entity will be relevant. To take a different example, a customer object in an online bookstore program represents a real person, and it is not necessary to include, say, the customer’s favourite colour in the model as it is not relevant to buying books. The customer’s favourite *author* might be included in the model, though, as this might help the program recommend new books to buy.

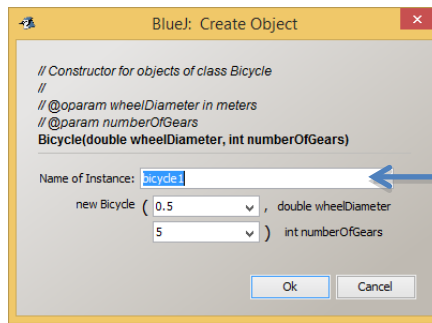
In the example you are looking at here, our `Bicycle` objects will simply model the way that the road speed of a real bike responds to change in pedalling speed (revolutions per minute, or RPM) and gear selection. This is a very simple model, which doesn’t include complications such as steering (everything in this particular “world” happens in a straight line!)

## Creating an object

Let’s try creating some objects. To create a `Bicycle` object (an instance of the `Bicycle` class) in BlueJ syou right-click on the class and select the option shown below.

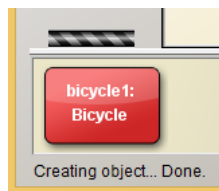


A dialog box asks you for some information about the object you want to create - for example how many gears will this particular bicycle have? We will use the values shown below:



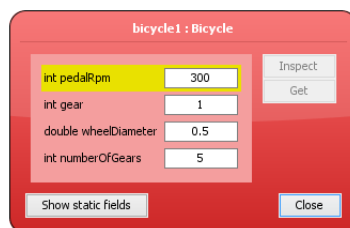
name of new object,  
BlueJ suggests a  
default value

The new object is created and shown in BlueJ in the Object Bench at the foot of the main window as a red rectangle. Note that the rectangle shows the name of this particular object (`bicycle1`) and the name of the class it was created from `Bicycle`. By convention, we give classes names that begin with capitals and objects names that begin with lower case characters. If you create another object from the `Bicycle` class in the same way, this object will also appear in the Object Bench, with another name (for example `bicycle2`) and the same class name – you can try this for another one or two bicycle objects.

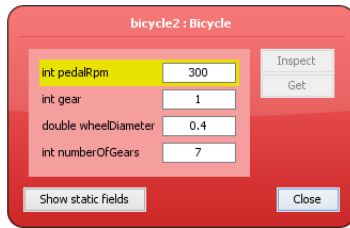


## Inspecting an object

So what can you do with this object? Firstly, let's see what it looks like. If you right-click on the object, and choose **Inspect** from the pop-up menu, you will see a more detailed picture of the object. This shows the properties and their values for this particular object.

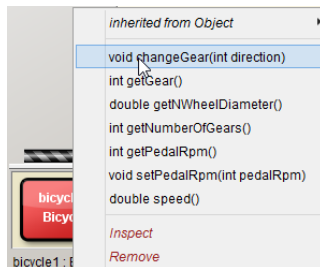


You can create another bicycle object, which will have the same set of properties, but may have different values for these, for example 7 gears instead of 5. Note that some properties have been set up automatically when the object was created – all bicycles are in gear 1 and being pedalled at 300rpm to start with. The other values are the ones that were specified when you created the object.

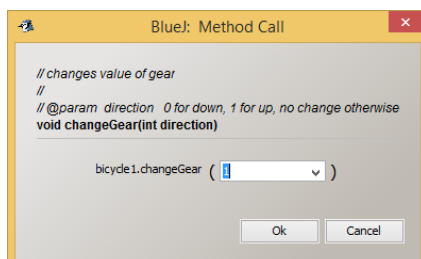


## Calling a method

Let's make the bicycle object do something. You usually make a real bicycle change gear by moving a lever – how do we make this model object change gear? Right-clicking on the object also gives you the option to make the object perform one of its **actions**. These actions are performed by **calling methods**, and each menu choice is the name of a method. Calling the method `changeGear` is the model equivalent of moving a real gear lever.



When you call the method you are prompted for some additional information about the action you want to be performed. With a real bicycle you can change gear up or down by moving the lever in one direction or another. With the model, you choose the direction by entering a 0 or a 1. This additional piece of information is known as a **parameter** of the method call.



Call the method with a `direction` parameter value of 1, and inspect the object again. You should see that the `gear` property has changed. What happens if you change up repeatedly? Or change down repeatedly? Does this behaviour make sense in terms of a real bicycle?

## Data types

What kind of information is the `direction` parameter? The method call dialog asks you to enter an `int` value, which means the value should be a whole number, or integer. `int` is the **type** of the parameter data.



When you inspect the object, you can see that most of the properties are also `int` values. However, the `wheelDiameter` is a different type, as the size of a wheel should not necessarily be an integer. The data type of `wheelDiameter` is called `double`, which means the value can be a floating-point number with a fractional part.

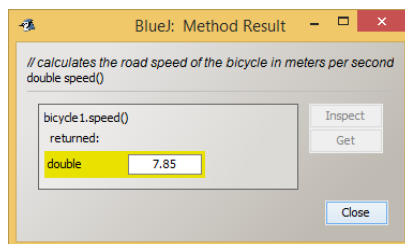
Java supports a number of data types for the information used in programs, including numbers, characters, `true/false`, and many others, and you will learn more about these shortly.

## Return values

Calling the `changeGear` method had no immediate visible effect. You had to inspect the object to see that it had made a difference. The method was essentially an instruction to the bicycle. The bicycle carries out the instruction but does not respond to the user to say it has been done.

Sometimes you want an object to respond to a method call, so that the method call is more like a question than an instruction. Let's say you want to know how fast the bicycle is actually going. This will depend on how rapidly it is being pedalled and also what gear it is in (increasing the gear will cause the road wheel to turn more times for each rotation of the pedals, so that the bicycle goes faster). Our bicycle can calculate its speed by using the values of its `pedalRpm` and `gear` properties. You can ask it to do so by calling the method `speed`.

Call the `speed` method. When you do so, you are not prompted for any parameters, as this operation doesn't need any additional information. What you do see is the response, or **return value**, of the method call. This is a value of type `double`. Note that the return type is shown before the name of the method in the method call menu. What is the return type of `changeGear`, and what do you think this means?



Try changing gear and checking the speed of the bicycle? Try also changing the pedal RPM – can you find a method that allows you to do this? What information is needed as a parameter for this method? Does the bicycle behave as it should – in other words, is the bicycle object a reasonable approximation to the behaviour of a real bicycle?

Sometimes it makes sense for a method to have both parameters and a return type. When you change gear on a real bicycle you may be able to get an immediate response to the action that tells you what gear the bicycle is, such as a numeric indicator or simply the physical position of the lever. It might improve our model of the bicycle if the `changeGear` method returned a value that indicated the new value of gear. What would be the return type of the method if it were modified to do this?

### WHAT'S GOING ON IN THE COMPUTER?

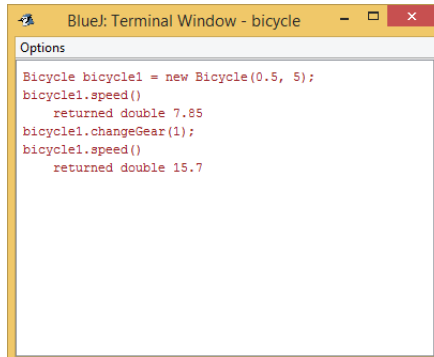
*How is Java making use of the capabilities of the computer while you manipulate the bicycle object?*

- *Executing instructions – a method is a set of instructions, and the computer executes these instructions when the method is called*
- *Storing information needed while executing instructions - parameter values and the properties of the object need to be stored so that they can be used by the instructions as the method runs*

## Programming objects

So far you have seen how to create objects and call their methods using BlueJ's menus. This is useful when learning about object-oriented programming and creating and testing classes.

However, as we said earlier, a Java program creates and uses objects as it runs. A program will do so as a result of running instructions written in **Java code**. You can see the equivalent Java code when you are working in BlueJ by opening the **Terminal** window, and switching on the **Record Method Calls** option. Creating a `Bicycle` object and calling some methods will produce the following in the Terminal:



```
BlueJ: Terminal Window - bicycle
Options
Bicycle bicycle1 = new Bicycle(0.5, 5);
bicycle1.speed()
    returned double 7.85
bicycle1.changeGear(1);
bicycle1.speed()
    returned double 15.7
```

The first line shows that you create an object with the word `new`, and specify the object name and type (class name). The following lines call methods of the object.

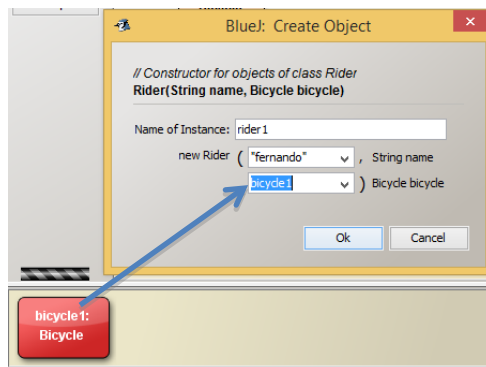
You will learn to write your own code using the Java language soon.

## Object interaction

Most object-oriented programs need more than one type of object. So far you have only created bicycles. If you look at the project in BlueJ you will see there is another class, `Rider`, so it looks like you can put a rider onto each bicycle. Let's create a rider object using this class. First, create a bicycle object `bicycle1` as before in the Object Bench, and then right-click the `Rider` class and select the option:

```
new Rider(string name, Bicycle bicycle);
```

The dialog box prompts you for information to initialise the object. These items of information are actually parameters for a **constructor** that belongs to the `Bicycle` class – a constructor is similar to a method but is called only when a new object is created. The `name` parameter is straightforward as its type is `String` (remember to put quote marks round the string value).



The other parameter represents the **bicycle object** that the new rider object will be riding. This is an example of one object that is related to another object. What is the type of this parameter? It is not one of the basic data types mentioned earlier. Instead, it is the **name of the class from which the related object was created**, namely `Bicycle`.

So what do you put in the prompt? You can't somehow squeeze the whole bicycle object into that text box, so instead you enter the name of the object, `bicycle1`. This name represents the whole object – we say that it is a **reference** to the object. In BlueJ you can either type the object name or simply click on the object in the Object Bench and its name will be copied to the prompt. Click OK and the new rider object `rider1` will be added to the Object Bench.

The two objects in the Object Bench are now related to each other. To observe this in action, inspect `bicycle1` and note its property values. Then, call the method `speedUp` of `rider1`. Not much will happen obviously as this method does not return a value. However, you can inspect `bicycle1` again and note the change that has happened. This shows that an **action by one object has caused the related object to perform one of its own actions**. Try experimenting with other methods of `rider1` and note the effect on `bicycle1`.

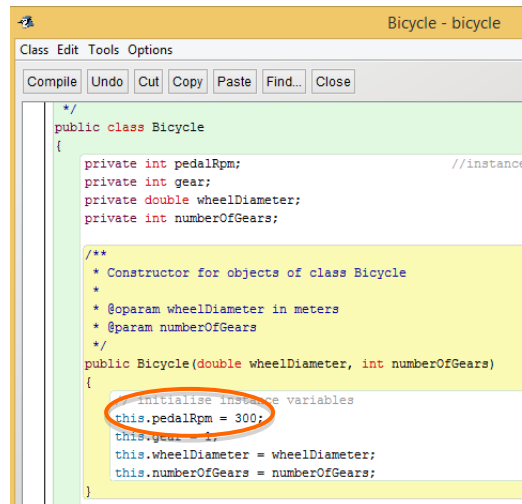
In this example you used an object as a constructor parameter. Method parameters can also be objects. To test this, create another bicycle object, `bicycle2`, and call the `changeBicycle` method of `rider1` so that the rider is now riding the `bicycle2`. Then, check by inspecting that calls to the methods of `rider1` now affect `bicycle2`, not `bicycle1`.

## Source code

You have seen how to create objects in BlueJ as instances of classes. The classes define what properties and actions these objects can have. Sometimes when you are writing a program you can take classes that someone has created and use them, as you have seen in these examples. Most often, though, you need to **create your own classes** which allow you to define exactly what kind of objects your program can

use, and what properties and actions they can have. In many cases you will use a combination of your own classes and classes that have already been written.

You create a class by writing Java **source code**. In fact, when you write an object-oriented program you are mainly writing or modifying classes. You will learn in detail how to write source code to define a class shortly. For now, let's try to make a simple change to the `Bicycle` class. Right-click on the class and select the **Open Editor** option. A BlueJ editor window should open, showing the source code for this class.



Find this line of code (highlighted in the screenshot):

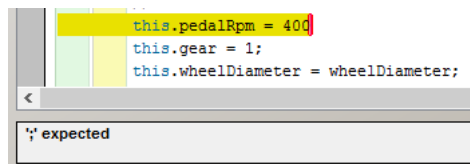
```
this.pedalRpm = 300;
```

and change the number from 300 to 400 (there are actually two lines which contain this same code – choose the first one). The way the `Bicycle` class is displayed in the BlueJ window will now have changed – it will now be shown cross-hatched. This indicates that the source code for the class has been changed and not yet **compiled**. Click the **Compile** button in the editor. You should (hopefully) see a message “**Class compiled – no syntax errors**” at the bottom of the editor, and the `Bicycle` class should be displayed without cross-hatching. You should then be able to create a new bicycle object, inspect it and observe the effect of your code edit.

Why do you need to compile the code? Well, we said earlier that to program a computer you need to give it instructions in a language it understands. Unfortunately, the computer doesn't actually understand Java (or any other programming language, in fact). It only really understands very simple instructions which consist of binary numbers, called **machine code**. It is very difficult for human programmers to create useful programs by writing these instructions. Programming languages such as Java are known as high-level languages, which can be translated, or compiled, by another program called a **compiler** into machine code that the computer can execute. Because programming languages have strict rules (syntax) it is feasible to translate source code to machine code – it would be unfeasible to compile instructions given in natural language. It is much easier to write useful programs using a high-level language than using machine code.

The compiler doesn't just translate – it also checks that your source code obeys the rules of the programming language. If your code breaks any rules it can't be translated into machine code successfully, and the compiler will give you one or more **error messages**. Seeing a compiler error message is not something to get upset about. **All programmers, from novices to professionals, encounter compiler errors as they write code.** An important part of programming is reading error messages, identifying the errors and fixing them. This can seem frustrating to beginners, but in fact is a process of the programmer working together with the compiler to craft perfect code!

Let's try to break a rule and see how the compiler helps. In the code editor for the Bicycle class find the line of code you change before. Delete the semi-colon from the end of the line and click Compile. You will see an error message at the foot of the editor, and the line you edited will be highlighted.



This error message gives a pretty strong clue to what is wrong – in Java, there is a rule that statements must end with a semi-colon. You can fix this and re-compile – you should see the “no syntax errors” message again. However, some compiler error messages can be a bit more difficult to interpret than this. BlueJ has a button with a ? to the right of the error message, and clicking on this can sometimes give more useful information about the error.

## Wrap up

You've been introduced in this lecture to the following concepts:

***Computers and Programming, Objects, Classes, Methods, Parameters, Data types, Return values, Object interaction, Source code and Compilers***

In the next lecture you will start to learn how to write your own Java code