4. ADDING METHODS TO CLASSES

Methods

Once an object has been created, what might the program want to do with it? It might want to:

- use the value stored in any of the object's fields
- change the value stored in any of the object's fields

You have seen how fields are defined in the code for a class, and how the values stored in those fields can be accessed.

However, one of the most important features of object-oriented programming is that objects don't just store information – they can also perform **actions**. So, once an object has been created the program might (and almost certainly will) also want to:

- ask the object to perform any of its actions by **calling a method**

When you call a method you can think of this as **sending a message** to that object. A message asks an object to perform an action. The action may be simply an instruction to do something, with no response needed. Alternatively, the action can be like a question that requires a response. In the latter case, the method needs to return a value.

In this lecture you will learn how to write code within a class to define the methods for that class. You will also see code that calls methods. As you will see in more detail in the next lecture, objects can send messages to each other, by calling methods, and this is how they collaborate with each other

Defining a method

Now let's see how a method is defined. In lecture 1 you saw that there is a method called `speed` in the example `Bicycle` class that returns a value representing the road speed of the bicycle. Here's the code for that method:

```
    public double speed()
    {
        // speed is pedal RPM * wheel circumference /60 * gear
        double wheelCircumference = 3.14 * this.wheelDiameter;
        return this.pedalRpm * wheelCircumference / 60 *
             this.gear;
    }
```

From this, you can see that the method has a code block, just like a constructor, and has a header that contains the name of the method. Methods and constructors look quite similar, but there are some important differences:

Constructor	Method
Called only once, when object is created	Can be called many times after the object has been created
has to have the same name as the class	can't have the same name as the class
Doesn't return anything so has no return type	Can have a return value so need to specify return type (void if no response required)

So the method `speed` has a return type of `double`, which is specified before the name of the method. If a method does not return a value (because it performs an action that doesn't require a response) you need to specify `void` as the return type.

This method has no parameters, as it doesn't need to be supplied with any additional information to do its job. You need to specify a parameter list for every method, which declares parameters just like in a constructor, but in this case the list is empty so is just written as empty round brackets ().

Note the name of the method. By convention in Java, **method names start with lower case characters**, so you should stick to this when naming methods.

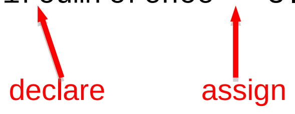
The combination of the name and the parameter list of a method is known as its **signature**. You can have more than one method with the same name, as long as the signatures are different. This is called method overloading, and is similar to the constructor overloading you saw in lecture 2.

Let's look inside the code block now, to see the code that runs when the method is called. This code defines a set of instructions that carry out an action that an instance of this class can do. In lecture 1 we said that a 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`. So the speed method needs to do a calculation. Note that the first line inside the method, starting with `//`, is simply a comment that explains how the calculation is done.

You will learn more about writing expressions to do calculations later, so don't worry about the details of the method just yet. There are a couple of things that are important to note at the moment, though.

The calculation is done in two steps. First, the circumference of the wheel is calculated from the wheel diameter (which is stored in the object as a field). The calculated value is **assigned** to a `double` variable `wheelCircumference`, which is declared inside the method. A variable declared inside a code block, for example a method, is a **local variable** – it only exists inside that block. If you try to use it outside that block, the compiler will report an error.

```
double wheelCircumference = 3.14 * this.wheelDiameter;
```



The second step completes the calculation and produces the result we want the method to return. Instead of assigning this to a variable, we simply use the `return` key word to indicate that this value should be returned. **A method which has a return type other than void must include a return statement** like this.

Calling a method

You saw in lecture 3 how to create an object and a variable that refers to the object, for example:

```
Bicycle bicycle1 = new Bicycle(0.5, 5);
```

You can call the `speed` method of this object as in this expression:

```
bicyle1.speed()
```

You saw code similar to this in lecture 1 when you called the method in BlueJ and looked at the equivalent code. Note that the method is called when the expression is evaluated.

Remember, `bicycle1` is a variable that refers to an object. To call a method of an object you use the dot notation mentioned earlier - you write *object name.method name(parameter values)*.

When a method returns a value you usually want to use that value in some way. For example, you could write a statement to declare a variable and assign the value that is the result of the method call to that variable. The variable type must match the method return type:

```
double currentSpeed = bicycle1.speed();
```

This statement evaluates the expression that calls the `speed` method. Sometimes you call a method directly in a statement. This is common when the method does not return a value. For example, the `Bicycle` class has a method `changeGear` that has the return type `void`, and would usually be called with a statement like this:

```
bicycle1.changeGear(1);
```

You saw previously that when creating an object your code must supply a valid parameter list. This is also the case for calling methods. The supplied parameters must **match the number and types of the parameters declared in the method**.

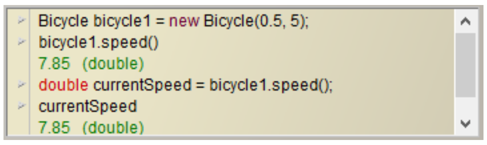
Additionally, if you assign the return value of a method to a variable, the variable type **must match the return type** of the method. If either of these conditions is not met the compiler will find the mistake and report it as a compiler error.

For example, the following are **not** valid method calls for a `Bicycle` object:

```
double currentSpeed = bicycle1.speed(2);
int currentSpeed = bicycle1.speed();
int newGear = bicycle1.changeGear(0);
bicycle1.changeGear(0, 1);
```

Can you see what is wrong with each of these? You will need to look at the listing of the `Bicycle` class at the end of this lecture to see the code for the `changeGear` method.

> ***CodePad*** *– open the bicycle project for lecture 4 and try the following in the CodePad*
>
> - *Declare a Bicycle variable, and initialise it by instantiating a new object*
> - *Enter a call to the object's speed method as an **expression** – note that the expression is evaluated by calling the method*
> - *Enter a **statement** that declares a variable of type double and assign to it the value returned by the object's speed method*
> - *Enter the name of the variable to see its value*



```
Bicycle bicycle1 = new Bicycle(0.5, 5);
bicycle1.speed()
7.85  (double)
double currentSpeed = bicycle1.speed();
currentSpeed
7.85  (double)
```

Scope of variables

You have now seen quite a few variables, and learned that variables can be used in different ways. The use of variables is related to their **scope**. Scope is **the extent in the code within which a variable exists and can be used in that code**. For example, the scope of a local variable is limited only to the block in which the variable was declared, whereas a field is a variable which belongs to the whole object and can be used in any method. Note that the `speed` method uses the values of several fields to do its calculation as well as declaring a local variable to help with the calculation.

Variable	Scope
local variable	the block in which the variable was declared
field (instance variable)	any code in the class which contains the field
parameter	the constructor or method which declares the parameter

Getter and setter methods

As well as adding methods, we have changed the field declarations of the `Bicycle` class slightly. They now have the keyword `private` in their declarations. Let's see why. It might be useful to be able to find and use the value of, for example, the `wheelDiameter` field of a bicycle object. You could use dot notation to access this value:

```
int diameter = bicycle1.wheelDiameter;
```

If you can access a variable, you can also assign a new value to it, for example

```
bicycle1.wheelDiameter = 0.3;
```

This might not be useful, and in fact, you might want code to only be able to read the value, and not to be able to write new data to the field. You don't necessarily want anyone to be able to come along and put a smaller or larger wheel on your bike! You may want to make this field **read-only**.

The problem is that you can't get one without the other – if you can access a variable you can change it. So, marking the `wheelDiameter` field `private` prevents access to it almost completely. The exception is that code inside the `Bicycle` class can access the field. The constructor needs to be able to access it to set its initial value, for example. The line of code above, in the program code that created the object, which tries to assign a value to 0.3 will now give a compiler error because the field is private.

So how do you give read-only access to the value? This can be done by including a method that simply returns the value of the field. This is known as a **getter method**, or to use a posh term, an **accessor method**.

```
    public double getWheelDiameter()
    {
        return this.wheelDiameter;
    }
```

Note that:

- The getter method is part of the `Bicycle` class so can access the private field
- The `getWheelDiameter` method is **public**, so other code can call the method
- As a result, other code can find the value of the field using the getter, but have no access to change the value

If you want to provide read and write access, you can also include a **setter method** (or **mutator method**). Bicycle doesn't have one for the `wheelDiameter` field, but if it did it would look like the following. This takes a new value to assign to the field as a parameter and does not return a value, so has a return type of void.

```
    public void setNWheelDiameter(double wheelDiameter)
    {
         this.wheelDiameter = wheelDiameter;
    }
```

Setters can also be used to control the way a field value can be assigned so that a nonsensical value can't be set. Look at the method `setPedalRpm` – what limitation does this impose on the value that can be set?

The ability to protect parts of a class by making them private, and providing controlled access through public methods, is known as **encapsulation**. It is common good practice to encapsulate fields in a class.

Note that the key word *public* allows a method, constructor or field to be called or accessed by code outside the class. It's not just getter and setter methods that are defined as public: any method that should be called from code outside the class itself should be defined as public. When applied to the class itself public has a slightly different meaning that you will learn about later.

Conditionals

Finally, let's look at one more method of `Bicycle`. If you look at the code you will see a getter, and no setter, for the `gear` field. This is strange – surely you want to be able to change gear (or model the action of changing gear)?

Well, you can, but in this model bicycle we want to set some **strict rules** about how the gear change works:

- There are a limited number of gears, and you can't select a gear number below or above that range. On a 5-gear bike you shouldn't be able to select gear 10, or gear -1, for example
- The gear mechanism is sequential – you can't go straight from gear 1 to gear 3 without selecting gear 2 on the way

Rather than a simple setter, changing the gear field is done with a method called `changeGear` which enforces these rules. The use of the method was demonstrated in lecture 1.The method takes a parameter that allows the calling code to specify whether to change up or down.
This is more complicated than `setPedalRpm` because there are some **decisions** to be made as the code runs:

- Change up or down?
- Is it possible to make that change with the available gears?

Decisions require the use of an **if-else**, or **conditional**, statement. This has the form

```
if(condition){
      Do something
}
else{
      Do something else
}
```

The code in the actual `changeGear` method is quite complicated, so let's look at a simpler version to get the idea. This version makes one decision only – change up or down?

```
    public void changeGear(int direction)
    {
        if (direction == 0){
            this.gear--;
       }
        else{
            this.gear++;
       }
    }
```

How do we know which way to change? Often a decision is based on the **current value of a variable** – here, it's the value of the `direction` parameter which has been passed in from the current method call. If it's 0, change down, otherwise change up.

We write this as the **condition** for the conditional statement. A condition is a value whose data type is `boolean`, i.e. true or false. This could be written as:

- The actual value – `if (true)` (not very useful)
- The value of a boolean variable – `if(boolValue)`
- An expression which evaluates to a boolean value – `if(direction == 0)`

The last of these is very common. In the example, it will be true if the parameter direction has the value 0, false otherwise. Note that the == (double equals) sign must be used for this, not a single equals sign (a very common mistake!). The use of equals signs can be described as follows:

Sign	Example	Meaning
=	x = 3	**Assign** the value 3 to the variable named x
==	x == 3	true if the value of the variable named x is **equal** to 3

Note that the code above uses the commonly used shorthand way, mentioned in lecture 2, of incrementing or decrementing an integer variable by 1

```
this.gear++
```

Has the same effect as

```
this.gear = this.gear + 1;
```

If you are feeling brave, look at the full `changeGear` method and see if you can understand how it implements the rules described above.

Wrap up

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

Methods, Messages, Scope, Getters/setters, Public/private, Conditionals

In the next lecture you will start learn how to write classes to create objects that work together, and how to write a program that uses these classes

Code for Bicycle class

```
/**
 * class Bicycle models the behaviour of a bicycle when pedal
 * RPM and gear are changed
 *
 * @author JP
 * @version 1.0
 */
public class Bicycle
{
    private int pedalRpm;
    private int gear;
    private double wheelDiameter;
    private int numberOfGears;

    /**
     * Constructor for objects of class Bicycle
     *
     * @oparam wheelDiameter in meters
     * @param numberOfGears
     */
    public Bicycle(double wheelDiameter, int numberOfGears)
    {
        // initialise instance variables
        this.pedalRpm = 300;
        this.gear = 1;
        this.wheelDiameter = wheelDiameter;
        this.numberOfGears = numberOfGears;
    }

    /**
     * Constructor for objects of class Bicycle
     */
    public Bicycle()
    {
        // initialise instance variables
        this.pedalRpm = 300;
        this.gear = 1;
        this.wheelDiameter = 0.5;
        this.numberOfGears = 3;
    }

     /**
     * gets the value of wheelDiamter
     *
     * @return  wheel diameter
     */
    public double getWheelDiameter()
    {
        return this.wheelDiameter;
    }
    /**
     * gets the value of numberOfGears
     *
     * @return  number of gears
     */
    public int getNumberOfGears()
    {
        return this.numberOfGears;
    }

    /**
     * gets the value of current gear
     *
     * @return  current gear
     */
    public int getGear()
    {
        return this.gear;
    }

    /**
     * gets the value of pedalRpm
     *
     * @return  pedal rpm
     */
    public int getPedalRpm()
    {
        return this.pedalRpm;
    }

    /**
     * changes value of pedalRpm
     *
     * @param  pedalRpm
     */
    public void setPedalRpm(int pedalRpm)
    {
        this.pedalRpm = pedalRpm;
        if (this.pedalRpm <=0) this.pedalRpm = 0;
    }

    /**
     * changes value of gear
     *
     * @param  direction   0 for down, 1 for up, no change
     * otherwise
     */
    public void changeGear(int direction)
    {
        if (direction == 0){
            this.gear--;
            if (this.gear <=1) this.gear = 1;
        }
        else if (direction ==1){
            this.gear++;
            if (this.gear > numberOfGears)
                this.gear = numberOfGears;
        }
     }

    /**
     * calculates the road speed of the bicycle in
     * meters per second
     *
     */
    public double speed()
    {
        // speed is pedal RPM * wheel circumference /60 * gear
        double wheelCircumference = 3.14 * this.wheelDiameter;
        return this.pedalRpm * wheelCircumference / 60 *
             this.gear;
    }
}
```