6. More About Methods

Methods and cohesion 1
Writing methods 2
Strings 2
Type wrappers 3
Control flow statements 4
Selection 4
Iteration 6
Arrays 8
Arrays of values 8
Arrays of objects 10
Wrap up 12

Methods and cohesion

Methods contain the code that allows objects to carry out the actions that they are designed to perform. A method should implement **a single action** with a clear purpose. If an object needs to be able to do lots of different actions, then it should have lots of different method to implement them. For example, the `Bicycle` class that you have seen previously has a method `changeGear`. That method causes a bicycle object to change gear – and that's all it does. It doesn't speed up pedalling, it doesn't turn a corner, it doesn't switch on a light, or anything else – it just changes gear.

A class in which the methods are designed this way is said to have a **high degree of cohesion**. On the other hand, a class with methods which each do several unrelated tasks is said to have a low degree of cohesion.

Why is this important? Code that is cohesive is widely considered to be easier to understand and modify, more robust and less prone to programming mistakes.

Note that the idea of cohesion also applies to classes. A class with high cohesion represents a single type of "thing" with a clear purpose, and only contains methods clearly related to that purpose. Do you think the `Bicycle` and `Rider` classes are cohesive – remember what we said about responsibilities in the previous lecture?

Here are a few examples of the kinds of actions that methods are commonly written to perform:

- Changing the state of one or more objects
- Doing calculations
- Processing information
- ..and so on

As an example, which of the above do you think would best describe the method `changeGear` in `Bicycle`?

Writing methods

When you create a method in a Java class you write a set of instructions to perform the action. In some cases this will be a very short set of instructions. There are situations in even the most complex object oriented program where you can write a method with only one or two lines of code. If you find that all your methods have many lines of code you might want to ask yourself whether those methods are cohesive.

Some actions, though, are unavoidably complicated. The `speedUp` method that you saw previously in the `Rider` class is quite complicated because it represents some knowledge that the rider has about controlling the speed of a bicycle, and involves making a decision.

To perform an action you need to do some combination of the following:

- Carry out a sequence of instructions in order
- Choose between alternative instructions depending on some condition
- Carry out the same instruction or set of instructions repeatedly

In order to create a method that performs an action correctly, you have to apply or design an **algorithm**. An algorithm is a procedure or formula for carrying out a procedure or solving a problem. You then have to be able to write Java code that implements that algorithm. In this lecture you will learn more about the **features of the Java language** that allow you to build methods to implement a range of algorithms. In the next you will learn some further features and then at some examples of how to **design an algorithm to solve a particular problem**.

Strings

You have used strings already, but let's look at them in a bit more detail. A string is a collection of characters. Unlike characters, **strings in Java are objects**. `String` is the name of a class which is provided by the Java platform and which you can use in your programs.

Strings can be created using the *new* keyword, or more conveniently using a string literal:

e.g.
```
String hello1 = new String("Hello World!");
String hello2 = "Hello World!";
```

Since a string is an object, it has some methods that perform actions appropriate for a string. There are quite a few of these methods, but a couple of the more useful ones are:

indexOf(String) returns an `int`, the index, or position, in the original string of the first occurrence of the specified sub-string

e.g.
```
String bigString = "This is the whole string";
String weeString = "is";
int where = bigString.indexOf(weeString);
```

equals(String) returns a `boolean`

e.g. `if (aString.equals(anotherString)) {…}`

Using `equals` is the correct way to check whether two strings are equal. You should not use `aString == anotherString` as you would for primitive types as this will not always work as you expect.

You can join, or **concatenate**, strings using the **+** operator. The + operator works differently for strings than it does for numbers.

e.g.
```
String s1 = "Hello ";
String s2 = s1 + "World!";
```

`s2` will be the string `"Hello World!".` Note that concatenating always creates a new string – you can't change the contents of a string, for example by appending some characters, once it has been created. There is another class called `StringBuilder` that does allow changes to its content and which should be used if you want to build up a string by adding characters.

Type wrappers

As you have seen, simple data items can be represented using the primitive types. However it is sometimes useful for something as simple as a number to be able to perform actions and store information other than the value itself. For example, wouldn't it be nice to have a number which knows not just its current value but also the minimum and maximum values which it can hold?

There is a set of Java classes which can contain equivalent data to the primitive data types. These are known as the *type-wrapper classes*.

Primitive type	Type wrapper
byte	**Byte**
short	**Short**
int	**Integer**
long	**Long**
float	**Float**
double	**Double**
char	**Character**
boolean	**Boolean**
void	**Void**

`Byte.MAX_VALUE` and *`Byte.MIN_VALUE`* which contain the maximum and minimum values which a *Byte* object can have, and *Byte* objects have a method `byteValue` which returns the value as a primitive *byte*. Other type wrappers have similar methods.

Type wrappers also have useful methods that help **convert** from one type to another. For example, the following method call takes a string as a parameter, converts its value to an integer if possible and returns it as an int.

```
int i = Integer.parseInt("123");
```

CodePad *- Open the bicycle project for lecture 6 and try the following in the CodePad:*

- *Use the expression `Integer.MAX_VALUE` to find how large an integer can be*
- *Create a new Integer object using*

```
Integer i = new Integer(5);
```

- *Call `i.intValue()` and note the result and its type*
- *Call `Integer.parseInt("123")` and note the result and its type*
- *Call `Integer.parseInt("12A")` - what happens?*

Control flow statements

Simple statements are executed one at a time, as a **sequence** in the order they are written in the code. However, not all algorithms can be realised with a simple sequence of instructions. **Control flow statements** allow your code to make decisions on which instructions to execute (we call this **selection**) and to execute the same instructions repeatedly (we call this **iteration**). A control flow statement can appear any place that a simple statement can. It includes one or more code blocks that can themselves contain any number of statements (including other control flow statements!). The following sections show the commonly used control flow statements in Java.

Selection

You have already seen an example of the *if-else* statement. More generally, it has the form:

```
if (condition) {
      statement(s)
}
[else {
      statements(s)
}]
```

Here, *condition* can be **any Boolean expression that evaluates to true**. This could be the name of a Boolean variable, a method call that returns a Boolean, or commonly, an expression containing a relational operator, for example:

```
x == 5
x > 0
x <= 10
x != 5
```

The square brackets [ ] round the **else branch** do not actually appear in code, they simply indicate here that the else branch is optional – without it, the statement just does nothing if *condition* is false. It is also possible to have multiple options with one or more **else if** branches before the else branch.

It is quite common to use a compound condition, which combines two or more simple expressions with conditional operators. The following example shows an if-else statement which uses the || (or) operator in its condition:

```
if ((num <= 0) || (num >= 10))
{
      System.out.println(num + " out of range");
}
else
{
      System.out.println(num + " within range");
}
```

If a block in an if statement contains only a single statement, the brackets are not needed. This can make your code more compact, but be careful, as it is less obvious what code belongs to the if statement.

```
if ((num <= 0) || (num >= 10))
      System.out.println(num + " out of range");
else
      System.out.println(num + " within range");
```

Note that the computer doesn't actually **decide** which branch to execute – rather, the value of the condition **determines** which branch should be executed. If the value of `num` is 5 then the output will be "5 within range". It doesn't matter how many times you run the code, it will always do the same thing if the input is the same. If the value of num is -5 then the output will always be "-5 out of range".

INDENTATION

*By convention you should **indent** lines that are inside a code block relative to the position of the start of the block. When you use control flow statements inside methods, indentation becomes really important in making your code easy to understand for yourself and other programmers. Don't be lazy with indentation – the editor in BlueJ helps you anyway, but sometimes you need to "tweak" it manually.*

Iteration

The statements that allow iteration are commonly known as **loops**. There are three types of loop. The choice you make when implementing an algorithm depends on two factors:

- Do you know exactly how many times you want to repeat the instructions in the loop, or should the repetition continue until some condition becomes true
- Should it be possible for the loop to finish without executing the instructions at all

The **while** and **do-while** statements, or loops, conditionally execute a block while the condition remains true.

The **while** statement has the form:

```
while(condition)
{
      statement(s)
}
```

The following code snippet shows a while statement. What do you think would happen if you missed out the second statement inside this loop?

```
while (num <= 10)
{
      System.out.println("number " + num);
      num++;
}
```

The **do-while** statement places the condition after the loop, so that it is checked after the statements have executed. This loop must execute at least once, even if the condition is false immediately. This has the form:

```
do
{
      statement(s)
} while(conditions);
```

The following code snippet shows a do-while statement:

```
do
{
      System.out.println("number " + num);
      num++;
} while (num != 10);
```

The **for** loop iterates over a range of values. It does so using an `int` loop variable. At the start of the for statement you specify the starting value of this variable, the stopping value and the amount by which the variable changes each time round the loop. Because you specify these, you know exactly how many times the loop will execute. You can make use of this variable inside the loop, but you don't (and shouldn't) change its value in the loop.

The for statement has the form

```
for(initialisation;termination;increment)
{
      statement(s)
}
```

The following code snippet shows a for statement. This will execute 11 times – the **loop variable** `num` takes the all the values from 0 to 10, including 10 due to the <= operator.

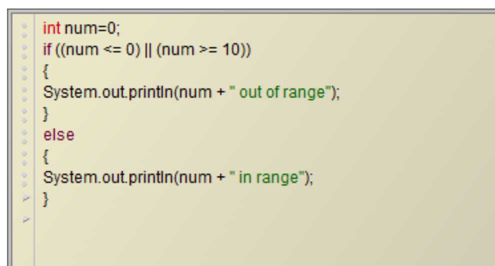
```
for(int num=0; num<10; num++)
{
      System.out.println("number " + num);
}
```

As mentioned previously, control flow statements can contain other control flow statements **nested** inside them. Quite a common example of this is **nested for loops**. The following code snippet uses nested for loops to print a multiplication table:

```
for(int x=1; x<=10; x++)
{
      for(int y=1; y<=10; y++)
      {
            System.out.format("%4d",x*y);
      }
      System.out.print("\n");
}
```

CodePad *- Clear the CodePad and try the following:*

- *Declare and initialise int variable num=0*
- *Enter the if statements listed above and observe the output in the terminal window. When entering a multi-line statement in the CodePad press SHIFT-ENTER at the end of each line, and ENTER at the end of the whole statement. Don't bother indenting in the CodePad – the code there is not meant to be read*



- *Enter each of the while, do while and for statements listed above and observe the output in the terminal. Reset the value of num to 0 after the while loop. Note that the for loop declares a loop variable num, which is then a local variable within the loop – it is not the same variable as the num you declared earlier*

- *Enter the first for statement again with the initialisation and termination set to int num=1;num<=10, and observe the output*
- *Enter the first for statement again with the initialisation, termination and increment set to int num=1;num<=10; num+=3, and observe the output*

Arrays

The variables we have seen so far all hold a single value or refer to a single object. However, it is often useful to be able to work with more than one value of the same type together. We can do this using **arrays**. In Java, as in most programming languages, an array is a structure that holds **multiple values of the same type**. A Java array is itself **an object**, while the values it holds can be either primitive types or object types. You will see examples later in this lecture and in the following lectures of situations where arrays are useful.

Arrays of values

An array can contain primitive data type values. As it is an object, an array must be declared and instantiated using the **new** key word. The size of the array is specified when it is instantiated. For example

```
int[] anArray;
anArray = new int[10];
```

These two lines declare and instantiate an array that can hold 10 integer values. The values themselves are not specified here, so the contents of the array will all have the default value for an integer, which is 0.

An array can also be created using a shortcut. For example:

```
int[] anArray = {3,2,7,8,12,9,1,11,3,7};
```

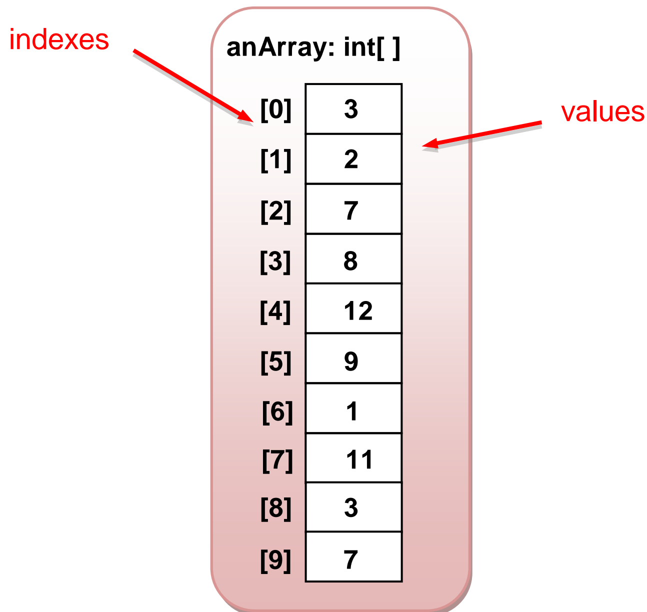
This single line declares and instantiates the array **and** sets values for all of its contents.

An array element can be accessed using an index value. For example:

```
int i = anArray[5];
```

Note that in Java **array indexes start from 0**. The value of `anArray[0]` is 1, while `anArray[5]` is actually the 6th element, with value 9 in this example.

As an array is an object, we can draw an object diagram that represents this array and its contents.



The size of an array can be found using the length attribute. For example:

```
int len = anArray.length;
```

You can also declare an array that contains arrays. This is known as a **multidimensional array**. To declare a multidimensional array you need to specify two or more sets of brackets, for example:

```
int[][] aMultiArray = new int[3][3];
```

Since there are two brackets, this is a two-dimensional array. Two-dimensional arrays are useful for representing tables of data. In this case it is an array of 3 arrays, each of length 3, which could represent a 3x3 table of numbers.

You can also create a multidimensional array with a shortcut:

```
int[][] aMultiArray = {{3,2,7},{8,12,9},{1,11,3}};
```

To access an element of this array you need to specify two indexes. For example:

```
int i = aMultiArray[1][0];
```

This would assign the value 8 to the variable `i` – the first index 1 selects the second array `{8,12,9}` while the second index 0 selects the first element of this array.

> ***CodePad*** *– Clear the CodePad and try the following:*
> - *Declare and initialise an array variable of type* `int`
> - *Type the name of the variable – you should see that it is an object reference, and if you click on the small object symbol in the left margin you can inspect the object that the variable refers to. Note that the object inspector for an array is similar to the object diagram above.*

```
int[] anArray = {3,2,7,8,12,9,1,11,3,7};
anArray
<object reference>  (int[])
```

- *Click on the object symbol and drag it to the Object Bench, giving it a suitable name when prompted. Note that an array is represented in the Object Bench by a symbol that resembles a stack of objects*

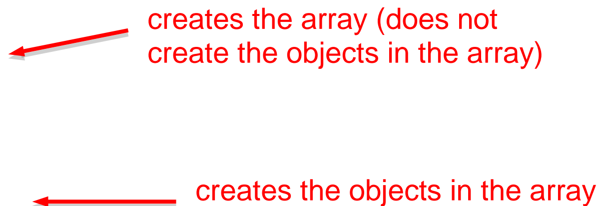
- *Declare and initialise a two-dimensional array variable of type* `int`*. Drag to the Object Bench and inspect the array. How does BlueJ represent the array? This might make more sense after you read the following section.*

Arrays of objects

Arrays can also hold objects of any type. Actually, the array doesn't hold the objects themselves – it holds **references** to objects. It is important to realise that when creating an array of objects, the array itself must be declared and instantiated, and that each individual element in the array must also be instantiated. This code creates an array of `Bicycle` objects.

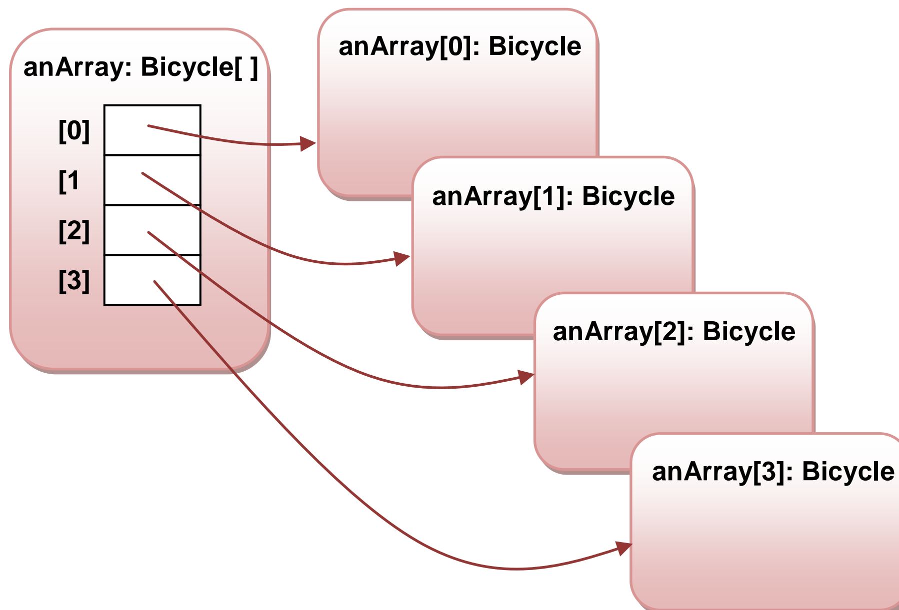
```
Bicycle[] anArray = new Bicycle[4];

for (int i=0;i<anArray.length;i++)
{
      anArray[i] = new Bicycle();
}
```



Note that you must create the array first, and then create the objects in the array. If you miss out the second of these steps, you will probably get errors when you run any code that tries to access the objects in the array as the array will simply contain **null** references – the elements in the array do not have any objects to point to as no objects have been created.

The object diagram after this code has run looks like this (to simplify the diagram the fields for the `Bicycle` objects are not shown – in fact each of these objects will have all the fields shown in the diagrams you have seen previously of `Bicycle` objects).



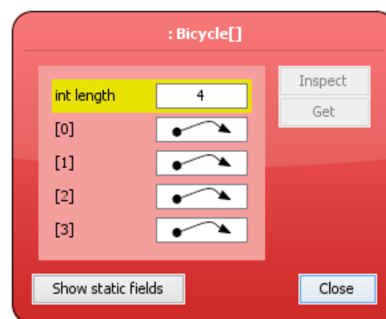
Each element of the array is a separate `Bicycle` object. You can access any of these by its array index and call methods, for example:

```
int rpm = anArray[2].getPedalRpm;
anArray[1].changeGear(1);
```

***CodePad** – Clear the CodePad and try the following*

- *Declare and initialise an array variable of type* `Bicycle` *and length 4*
- *Use a* `for` *statement as in the example above to fill the array with new Bicycle objects*
- *Type the name of the variable – you should see that it is an object reference, and if you click on the small object symbol in the left margin you can inspect the object. Note that the array elements are themselves shown in the object inspector as object references rather than values*

```
Bicycle[] anArray = new Bicycle[4];
for (int i=0;i<anArray.length;i++)
{
anArray[i] = new Bicycle();
}
anArray
<object reference>   (Bicycle[])
```



- *Click on the first array element in the object inspector window and click the Inspect button. You should see another object inspector showing a* `Bicycle` *object with the default field values. Close this* `Bicycle` *object inspector*
- *In the CodePad, write code to call the set the pedal RPM of the first object in the array to 50*
- *Call the* `speed` *method of the first object and note the value returned*
- *Call the* `speed` *method of the second object in the array and note that it is now different from the first – the objects in the array are separate and can be*

accessed and manipulated independently. You can also inspect the objects in the array to see their current field values

```
anArray
<object reference>   (Bicycle[])
anArray[0].setPedalRpm(50);
anArray[0].speed()
0.6541666666666667   (double)
anArray[1].speed()
0.0   (double)
```

- *Now create another array, this time of type* `Integer`*. Remember that Integer is a type wrapper class, so an* `Integer` *variable refers to an object which wraps a primitive* `int` *value. Inspect the array and note the difference from an array of type* `int`

```
Integer[] anotherArray = new Integer[5];
for (int i=0;i<anotherArray.length;i++)
{
anotherArray[i] = new Integer(i);
}
anotherArray
<object reference>   (Integer[])
```

Wrap up

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

Methods and Cohesion, Control Flow Statements, Strings, Type Wrappers, Arrays

In the next lecture you will learn how to design algorithms that allow methods to be written to solve real problems, and look in more detail at what happens when a method is called