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
{
    int pedalRpm;
    int gear;
    double wheelDiameter;
    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;
     }

     }
```

3.1 Purpose of the Bicycle class

Before we look at that code, let's consider why we want to have this class. Remember how we thought this class could be used in a program. A `Bicycle` class should allow objects to be created that model the way that the road speed of a real bike responds to change in pedalling speed and gear selection.

This would be important in some program where the speed of a bicycle is important. For example, it could be a game that involves characters racing or chasing each other on bicycles, and the pedalling speed and gear are set by the players' controls. If the `Bicycle` class models the bicycles' behaviour correctly, then the game will make sense to the players. Or, it could be a program that is built in to a computerised exercise bike, where rider wants to see a display of the speed he or she would be achieving for the amount of effort being put in.

For now it doesn't matter, we are just looking at the `Bicycle` class itself, not a whole program.

3.2 Code blocks

In Java, lines of code that belong together are grouped as **code blocks**. The start and end of a code block are marked by **curly braces**:

```
{

   line or lines of code
   inside code block

}
```

Code inside a block is often **indented** as shown in the example above. This makes the organisation of the code much easier to understand while reading it. Actually the compiler doesn't care about indenting, but it is **good practice to indent your code carefully** so that other programmers (and yourself) can read it easily. The compiler does care, very much, about code blocks, though. Putting braces in the wrong place or missing them out is a common source of compiler errors.

Code blocks can be **nested** inside other blocks – you will see shortly why this is useful. Nested blocks are usually further indented for readability.

```
{
        line or lines of code
        inside code block
        {
                line or lines of code
                inside nested code block
        }
}
```

The way Java uses blocks is an important part of its syntax, and is very similar to the syntax of a range of other popular programming languages.

3.3 Creating a class

The most important kind of code block is a class. The code for a whole class is **enclosed in a single block**. In order to define any useful behaviour, this code will almost always include other blocks nested inside it. The only part of the class outside the block is the **class header**, which needs to come just before the opening brace.

```
public class Bicycle
{
        code for class
}
```

The header's main job is to specify the **name** of the class, which, by convention should start with an **upper case (capital) letter**. The **case matters** – as far as Java is concerned, `Bicycle` is definitely **not** the same thing as `bicycle`. Using names with the wrong case is another common source of compiler errors. A further widely followed convention is that names which consist of more than one word are written with an upper case character starting each subsequent word – look out for examples of that as you study the code.

Good programmers follow conventions when writing code, and naming classes is a good example. Like indentation, the compiler doesn't care (as long as you are consistent), but if you follow naming conventions it is much easier for other programmers to read your code and see at a glance what is the name of a class and what isn't. It works the other way, too, making it easier for you to read someone else's code.

Key words

The words **public** and **class** in the `Bicycle` class header are examples of **key words.** A key word has a special meaning in Java and is not the name of something. In fact, you should never try to use a key word as the name of anything in a program. BlueJ helpfully colour codes key words differently from other words to make them obvious when you read the code.

The meaning of the key word *class* is probably fairly obvious – it indicates that the following code block contains a class. You will learn something about the meaning of *public*, and see more key words, later.

Comments

When you look at the code you will see some lines which begin with `//` or `/*`. BlueJ usually highlights these in grey. These lines are **comments**, which are not part of the code. They are **intended for other programmers**, and should explain the purpose of parts of the code and details of how certain parts work. You will learn later how you can write comments that are really useful. The compiler ignores comments completely when checking the code, so you can write anything you like in a comment (although it is good practice to make them relevant to the code!).

3.4 Fields

Let's start to look inside the class. The first things you see defined after the class's opening brace are the **fields**.

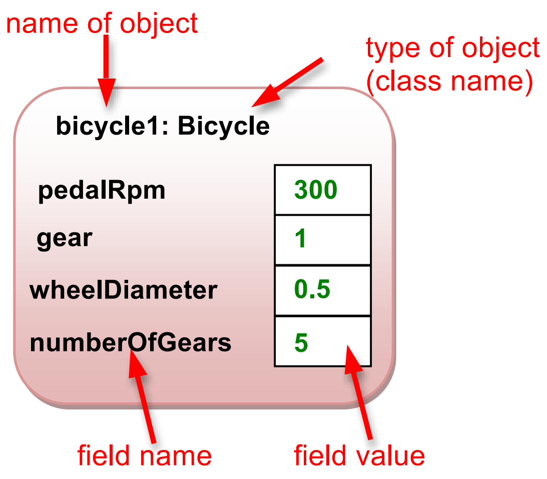
```
public class Bicycle
{
     int pedalRpm;
     int gear;
     double wheelDiameter;
     int numberOfGears;
```

Fields are the properties specific to an object created from that class. Each bicycle object will have its own value for `pedalRpm`, and so on. These are the values you see when you inspect an object in the object bench in BlueJ. Each field has a **data type**, e.g `int` and a **name**, e.g. `pedalRpm`.
The field definitions are always:

- **inside the class block**
- **not inside any other code block within the class**

They are usually defined at the start of the class, as in this example, but they don't have to be as long as they are not inside a nested block. As usual, though, it is a good idea to follow common convention and put them at the start.

It is sometimes useful to draw a diagram of an object showing the fields and their values, rather like the window you see when you inspect an object in BlueJ. Here is an **object diagram** of an example bicycle object.



Variables

The **value** of a field can **change** after the object is created. The value of `pedalRpm` changes to model the rider speeding up or slowing down the rate of pedalling. A field can in some cases change many times during the lifetime of an object.

In fact, a field is an example of a **variable**. You have seen that a variable is a piece of information, with a name and a value, which is stored and the value of which can change. You have also seen that you need to declare a variable in order to use it.

The field definitions in the `Bicycle` class are also examples of variable declaration. Once the field has been declared, code anywhere else in the class can refer to the field by its name.

A field is a variable that contains information which belongs to an instance of a class (that is, an object), and for that reason is also known as an **instance variable**.

3.5 Constructors

When you create an object, you want to make sure that the object is ready to use. This usually means that you want the fields to have sensible values. You may want the actual values to be decided at the time of the object creation, for example by the program code that actually creates the object.

In the previous lecture you saw that when you create a bicycle object in BlueJ you are prompted to enter some information that is used to assign values t the fields, and you saw also how this information can be written as parameter values when creating an object using code.

The part of the class which sets up, or **initialises**, the instances of that class is called the **constructor**. The constructor is a code block with a header that contains the name of the class (`Bicycle`) and a list of parameters. It also contains in this example the key word `public`, which we will look at later.

```
    public Bicycle(double wheelDiameter, int numberOfGears)
    {
        // initialise instance variables
        this.pedalRpm = 300;
        this.gear = 1;
        this.wheelDiameter = wheelDiameter;
        this.numberOfGears = numberOfGears;
    }
```

The code in the constructor runs **only once**, when an object is created. In this example, the constructor assigns values to the fields of the object (although constructors can do far more than this). The `pedalRpm` and `gear` fields are assigned default values. The other fields are initialised using values passed in as parameters.

The parameter list for this constructor is in round brackets () after the constructor name. Each parameter is listed as a data type and a name. That sounds familiar – in fact **parameters are another kind of variable**. The parameter list actually **declares** the parameters.

There is another key word lurking in this code – **this**. Here, it just means *"the object which is being created"*. So, `this.numberOfGears` means *"the field numberOfGears of the object which is being created"*. It's used here because we gave the parameters, e.g. `numberOfGears`, the same names as the fields that their values initialise. So

```
numberOfGears = numberOfGears;
```

would assign the parameter's value to the parameter itself, which is not very useful, but

```
this.numberOfGears = numberOfGears;
```

assigns the parameter's value to the field, which is what we want. Could there be a different way to tell parameters from fields? How would this change the code?

Note the **dot notation** in the example – to refer to a field of an object you write *object name.field name*. This notation is used widely when programming in Java.

3.6 Creating objects

You saw how to create objects using BlueJ in lecture 1, and you saw that objects can also be created by a Java program using code. Note that to make a useful Java program you need to write both of the following:

- Code that **defines classes**, and so defines what kind of objects can be created and what they can do
- Code that **creates objects** and uses the capabilities of those objects

For now we will look at those separately, but you will soon find that they are not always different things.

Let's look again at some code that creates `Bicycle` objects and see how it relates to the constructor of the `Bicycle` class. Here is the code that BlueJ showed in the terminal window when creating an object.

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

There's quite a lot going on here, so let's break it down a bit:

What is `bicycle1`? That is the name of the object that is being created. In fact, `bicycle1` is the **name of a variable**. This variable is little bit different from the instance variables you saw earlier - it exists in the program code that creates the object. It is an example of another kind of variable – a **local variable**. A local variable is a variable that is **declared and used within a block of code**.

Like any other variables, local variables need to be declared by specifying also the data type of the variable. The **data type** here is not a simple type such as `int` or `boolean` (these are known in Java as **primitive types**). It is the type of the object, which is the **name of the class which the object will be created from**, `Bicycle`.

The code

```
Bicycle bicycle1
```

<u>declares a variable</u> called `bicycle1` which refers, not to a simple value, but to an **object which is an instance of the class** `Bicycle`.

VARIABLE TYPES IN JAVA

*The example here emphasises something very important about Java variables. The **type** of a variable can be either a **primitive type** (a single piece of information such as an int) or an **object type**. Primitive types are limited to those defined as part of the Java language, but object types can include any Java classes that are included your project.*

The key word `new` is very important in Java - it **creates a new object**. The type of object to be created is specified after the key word.

The code

```
new Bicycle
```

<u>creates a new object</u> which is an instance of the class `Bicycle`. We call the process of creating an instance instantiation.

When an object is instantiated, the code in the constructor of the class is run, and the parameter values are passed in to that code.

The code

```
Bicycle(0.5, 5)
```

passes parameter values into the constructor of `Bicycle`. These are actual values of numbers here, but parameters can also be the names of variables that have already been assigned values. Note that we don't specify the data type here. The parameter values are used in the constructor to **initialise** the new object. Look at the code for the constructor above – what will the values of all its fields be after initialisation?

So, the whole line of code

1. Declares a variable of type Bicycle
2. Instantiates an object of type Bicycle
3. Initialises the object
4. Assigns the object to the variable which has been declared

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

declare assign instantiate initialise

This doesn't all have to be done in one line of code. It is quite common to declare an object variable

```
Bicycle bicycle1;
```

bicycle1 is a variable which is a null reference

then create the actual object later and assign it

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

bicycle1 is now a variable which is a reference to an object

In this case, `bicycle1` initially does not refer to an object, because no object has been assigned to it – it is a **null reference** until an object is assigned.

3.7 Accessing fields of an object

When a `Bicycle` object is created the code in the constructor assigns values to the fields. We can now write code that reads the field values, or changes them by assigning new values. Assuming that we have run the code in the previous section, and have a variable `bicycle1` that is a reference to a `Bicycle` object, the following code expression represents the `gear` field of the object:

```
bicycle1.gear
```

Similarly, `bicycle1.numberOfGears` represents the `numberOfGears` field.

Note the dot notation as mentioned above – *object name.field name.* Here, the object name is the name of the variable that is a reference to the object. The name we use to refer to the object also gives a way to work with that object.

This line of code declares a new int variable `x` and assigns to it the current value of the `gear` field of `bicycle1`.

```
int x = bicycle1.gear;
```

We can also change the field value by assigning a new value to it:

```
bicycle1.gear = 4;
```

3.8 Constructor parameters

What would happen if you tried to create an object with this code?

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

What about this?

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

The code for the constructor declares two parameters, the first of type `double`, the second of type `int`. These two lines supply parameters that **don't match the constructor**. The first supplies only one parameter; the second supplies a second parameter that is a string, not an `int`.

These lines are not valid ways of creating a bicycle object. The supplied parameters must **match the number and types of the parameters declared in the constructor**. In either case, the **compiler** will find the mistake and report it as a **compiler error**. The compiler does a very important job of checking code and finding mistakes like this.
What about this?

```
bicycle1 = new Bicycle();
```

This is actually valid! A class can have more than one constructor – as many as you like, as long as the parameter list is different for each one. This is called **constructor overloading**. If you look at the code for the whole `Bicycle` class, you will see that there is a second constructor that takes no parameters at all, which matches the line of code above. This constructor sets sensible default values when no actual values are specified, and is known as a default constructor.

So, when the compiler finds code which creates a new object it:

1. Checks the class to see if there is a constructor that has a matching parameter list
2. Reports an error if not (and the program doesn't compile successfully)
3. Selects the constructor with the matching parameter list

Summary

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

Code blocks, Code for classes, Fields, Variables, Constructors, Creating Objects

In the next lecture you will start learn how to write methods in your classes that allow objects to perform actions.