1. Programming – Some Basics and Java

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

In this chapter, we'll look at some of the basic concepts of programming and how they apply to Java. We'll skip over some details – the point is to get the overall idea. This is the first of several chapters that cover key points.

Programming is about systems that process input data and generate output results. The simplest ones simply read in input data, do something to it, and generate an output. The very simplest ones just generate output. For example, here's the output of a program that just prints 'Hello World!':

`>Hello World!`

(In the examples I show, code and computer input-output will be in the `Courier` font. Output from computers will be **`bold`**; input will be *`italic`*.)

The Code: What does this look like in Java?

We're going to show the code for all the simple examples we introduce. As we move on, we will explain more about what each line means. *However for now quite a few bits of the text of the programs we show have to be taken on trust.* By the end of the module, you will understand all the terms we use. The text below is the complete program to print out "Hello World!":

```
class HelloWorld {
     public static void main(String[] args) {
          System.out.println("Hello World!");
     }
}
```

In this example, the line:

```
System.out.println("Hello World!");
```

is the one that actually generates the output. So what do all the others mean? They set up the necessary framework for the program to actually work – and we'll skip most of the details for now – but see KEY POINT below.

We read out the bit `System.out.println` "System dot out dot print line" and it prints the data that follows it in brackets to the screen.

KEY POINT: Program Names & File Names

We can ignore most of the meaning of the lines other than:

```
System.out.println("Hello World!");
```

for now. But one thing we do need to know is that in the line:

```
class HelloWorld {
```

"`HelloWorld `" is the name of the program. And this program ***MUST*** be saved in a file called `HelloWorld.java`

Make sure you know and understand the following points.

- *You can call a program whatever you want* – the computer pays no attention really and does not care. *The most important point is that it is meaningful to people.* Calling a program "`Lemon`" will work, but it won't help someone reading the program to understand it.
- *There are also some conventions on how you should name things in Java* – we'll get back to this later on but for now stick to letters and digits – *you cannot put spaces in names*.
- *Case matters* - depending on what computer operating system you are using the case of the filename might not have to match the case of the identifier (i.e. you *might* be able to call your program "`HelloWorld`" and store it in a file called "`helloworld.java`" – but it is *always safer* if they match). So in this case, your code should be in a file called "`HelloWorld.java`" – note the capital H and W.
- *The name* `HelloWorld` *is an* ***identifier*** – see below.

Advanced Aside – Skip on First Reading if New to Programming

(At various points in the notes there are 'advanced asides' aimed a people with some experience – if you're new to programming you can skip them – but you don't' have to.)
Strictly speaking we are naming something called a *class* here – and later we'll see that programs can have more than one class (and more than one file containing them). But for now, with our simple programs all in one class (and file) we can just think of this name as the program name.

Syntax: Reserved Words, Identifiers, Literals

The language we use to write down programs in Java (or any language) is called the language ***syntax***. Each programming language has its own syntax and rules (some are quite similar to Java and some are very different). You have to get the syntax of a program ***exactly*** correct before you can *compile* it - turn it into code the computer can run. For example, it matters that there is a ";" on the end of the line:

```
System.out.println("Hello World!");
```

And that the "{"…"}" are present in the right place. We'll get back to the details of how to do this later, but for now we are going to concentrate on three things:

- **Reserved Words:** These are words that form part of the Java language itself, and have specific meanings. In the example above, `class, public, static` and `void` are reserved words. You should not try to use these words in any other way (things are unlikely to work if you do) – so, for example, don't try to call your program "`static`", or "`void`". (I know we haven't explained what these particular words *mean* – we will do this later in the module.)
- **Identifiers:** These are words that *represent*, or stand for, something – and which you or some other programmer has chosen. The one we've talked about so far is "`HelloWord`" – but there are others too. For example, "`String`", "`args`", "`main`", "`System.out.println`" (the last one is actually three separate identifiers but don't worry about that for now). Of these identifiers, "`String`" and "`System.out.println`" are names chosen by the programmers who wrote some of the libraries of code that we can use with our programs; "`main`" was chosen by the original Java designers; and "`args`" is something we *can* actually change if we want to something else (*but don't* – everyone uses "`args`" because it's a *convention* – meaning all Java programmers stick to it).
- **Literals:** The only other piece of text in our program is "`Hello World!`" – this is not a reserved word or an identifier but a *literal*. It doesn't represent anything: it ***is*** the sequence of characters (or *String*) Hello World!. Not all literals appear in "" marks as we'll see later (for example, numbers).

To sum up, here's the program again colour-coded to show the different parts:

```
class HelloWorld {
      public static void main(String[] args) {
            System.out.println("Hello World!");
      }
}
```

- Red – Reserved Words
- Purple – Literals
- Orange – Identifers (you can change)
- Green – Identifers (you can't change)
- Blue – Identifiers (you can change but shouldn't)

Indenting and Layout

One thing to notice about our program so far is that some of the lines are *indented* – that is, they don't start at the left-hand margin.

```
class HelloWorld {
|----|public static void main(String[] args) {
|----||----|System.out.println("Hello World!");
|----|}
}
```

The "{" and "}" mark the beginning and end of *blocks* of code and are the way we *logically group* lines of code together.

In the version above, "|----|" stands for the *tab* character: after each "{" in the program, the lines following start one tab further from the left; after each "}" they start one fewer tab from the left. (Don't actually type the "|----|" in – they are just to show were the tabs go.) We do this to make the program more readable to humans. This version would also work perfectly well:

```
class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello World!");
}
}
```

In fact, so would this:

```
class HelloWorld {public static void main(String[] args)
{System.out.println("Hello World!");}}
```

But it's very difficult to read. This leads on to an important KEY POINT.

KEY POINT: Programs are Read by People

Although we write our programs using a strict syntax and follow strict rules so that they can be understood by computers, the most important readers are people.

This means we must write programs in a readable way – it's why we use indenting to make the layout clearer (and sometimes put in blank lines too). It's why we pick meaningful names for identifiers. We also do other things too – as we'll see as we go along.

KEY POINT: You *WILL* Forget What Your Code Means…

Unless you lay it out properly, pick sensible names for the identifiers you choose (and do the other things we'll see later).

It's not just about other people reading your code – it's you. I guarantee at some point you will forget what your own code means (after leaving it for a few weeks, usually). This is one of those things nobody seems to believe – we all have to find it out for ourselves.

Indenting and Laying-Out Code – A Choice

I've shown you one style of laying-out code in the example so far, but there is another one. Here is the same program in the two different styles.

```
class HelloWorld {
     public static void main(String[] args) {
          System.out.println("Hello World!");
     }
}
```

```
class HelloWorld
{
     public static void main(String[] args)
     {
          System.out.println("Hello World!");
     }
}
```

The only difference is where we put the "{" characters. In the first version:

- The "{" goes at the end of a line.
- The closing "}" goes under the first character of the line with the {.

In the second version:

- The "{" goes on the start of the next line.
- The closing "}" goes directly under the matching "{".

Both of these are fine and are allowed by the Java conventions – you can do either (it's a matter of preference). But choose one and stick to it – don't mix (at least in the same program).

Depending on how you've chosen to run your programs, you may find the software you use does this for you (or you can tell it to). *But* if it does, make you sure understand how to do it manually (you won't have software to do it for you in the exam for example; and the software can sometimes get confused).

KEY POINT: Layout and Readability Count!

In case you're thinking "yeah, yeah…" about all this layout, readability and meaningful names stuff – there are real marks for this in the assessment! And employers usually have rules about it too.

Advanced Aside: Tabs vs Spaces

Some of you will have been told "don't use tabs to indent programs – it's bad practice". Equally, some of you will have been told the opposite. It does not matter as long as you are consistent. Provided you stick to one or the other, your programs will look consistent – and, again provided you are consistent, good modern editors will let you replace all tabs with one or more spaces, or replace groups of one or more spaces with tabs. Personally, I prefer tabs because:

- **You get to type less** – you only have to type one character to increase the indenting level; with spaces you need to type at least two, usually four (people who use spaces will recommend you use at least two spaces, and often four, per level of indenting).
- **You can customize them** – modern editors let you decide how many spaces a tab character represents on the screen.
- **More forgiving** – if you accidentally put the odd space in, it won't mess up the indenting.

Running the Program – "Lots of Rules and No Mercy…"

(For this section, you need to have chosen a way to run your Java programs, as described in the chapter "Running Java Programs: Your Choices" and the supporting videos. There are several ways to do this but they all have two steps: turning the program you wrote into something the computer can understand – this is called *compiling* and it's done by a program called a *compiler* – the one for Java is called `javac`. The other step is actually running the code output by the compiler – in the case of Java this is done by an *interpreter* called `java`. Some ways of running Java programs "hide" the compiling step, but it's still happening.)

Using the software of your choice, type in the "Hello World!" program, and compile and run it.

Sounds simple – but you will probably find that you make mistakes typing it in and get to see compiler errors. For example, if we type in this:

```
class HelloWorld {
     public static void main(String[] args) {
          System.out.println("Hello World!")
     }
}
```

(Can you see the mistake?) You'll get this:

```
HelloWorld.java:3: error: ';' expected
          System.out.println("Hello World!")
                                             ^
1 error
```

And if you type in this:

```
class HelloWorld {
     public static void main(String[] args) {
          system.out.println("Hello World!");
     }
}
```

(Again, can you see the mistake?) You'll get this:

```
HelloWorld.java:3: error: cannot find symbol
          System.out.Println("Hello World!");
                     ^
  symbol:   method Println(String)
  location: variable out of type PrintStream
1 error
```

In both cases the errors are small (missing ";" in the first one – a "P" instead of a "p" in the second). You might think these do not matter *but they do*. Programming languages are very sensitive to syntax errors and you must get it right. So look carefully at what you've typed and compare it with the examples in this chapter – and then type them **exactly**.

(If you are wondering how you will ever remember all this stuff don't worry – everyone worries about that and the vast majority of people manage it pretty quickly. The hard part in programming is not remembering all the syntax rules, but learning how to turn the problem you need to solve into steps that you can program).

After that you can run the program – in the case of Hello World, it's very likely to work and the problems you have will mostly be with the syntax. But in most real programs, once you've got the hang of the syntax, the problem is that they *do* run, but *don't output what you expect* – which is when you start *debugging* (more later).

Statements

Programs in Java (and most languages) are made up of collections of *statements* – bits of text that mean something in Java. For example,

```
System.out.println("Hello World!");
```

is a simple statement in Java. Like most statements in Java, it ends with a ";". This might seem unnecessary, but it is possible to put more than one statement on a line (though mostly you shouldn't as it's not considered good style). The ";" is a 'traditional' end-of-statement character in many languages. Two things:

- **You Might Forget the ";"** – it's a common error to start with, particularly if you've programmed in a language that does not use it (e.g. Visual Basic).
- **Not *all* Statements Use It** – We'll see this later on, and it can be a nuisance.

Statements like this are *simple* statements – a single thing on its own. Very commonly we want to group statements together, to make *compound statements*. We do this by surrounding the statements we want to group with "{" and "}". You can examples of this in the Hello World program – it's not obvious there why you need to group things, but as we get to see more complex programs, it will become clearer.

Some statements in programs tell the computer to actually perform some operation; others tell the system something about the program that it needs to know to build a running program. In the program above, only the line

```
System.out.println("Hello World!");
```

actually performs an operation.

EXERCISES

- Try changing Hello World so it prints different things.
- Try changing it so it prints *more than one* thing.
- Try renaming it and seeing what happens; try fixing it.
- Try making mistakes in the syntax (if you didn't already when you typed it in) and see what the messages look like.

A Program With Input

Obviously "Hello World!" is ridiculously basic – for one thing, it *always* does the same (not very interesting) thing. How about a program that accepts some input and does something with it?

For example, here's a simple dialogue with a program that asks you your name and then says hello:

```
>What is your name?
Bob
>Hello Bob!
```

Let's think about the *sequence* of events going on here – the *Algorithm* that the program must *implement.*

1. Step 1 – Print out message "What is your name?"
2. Step 2 – Read in input from user *and store it*
3. Step 3 – Join user input with "Hello " and "!" and output result

The KEY POINT for us here is *storing* data – because we need to do that in this program.

KEY POINT: Variables

Computers store *data* in *memory*, and computer memory is just a very long list of numbered 'boxes' in which you can put that data. But it's very difficult to deal with memory directly. There are usually *billions* of these numbered boxes (so dealing with the numbers – or *memory addresses* – is difficult for humans); and also your program might not end up in the same bit of memory every time it runs (so the numbers can change). We fix this by using *names* for the data we store, and letting the computer work out which memory 'boxes' to put it in. We just worry about the names.

The names we use are another example of identifiers. But this time instead of identifiers that name programs we are talking about identifiers that name *stored data* – we usually call these identifiers *variables*.

The Code: What does this look like in Java?

```
import java.util.Scanner;

class HelloYou {
     public static void main(String[] args) {
          Scanner in = new Scanner(System.in);

          System.out.println("What is your name?");
          String name = in.nextLine();
          System.out.println("Hello " + name + "!");

     }
}
```

Some of the lines in this new program are the same or very similar to our last one. But some are clearly different. We'll look at the key lines below. But the three lines:

```
System.out.println("What is your name?");
String name = in.nextLine();
System.out.println("Hello " + name + "!");
```

are the ones that correspond with our algorithm above. The first one just prints out "`What is your name?`" and is very similar to the "`Hello World!`" example – but the others are slightly more complex.

Creating, or Declaring, Variables

`String name = in.nextLine();` - this line is the one that creates a new variable called `name`. That is, `name` is an identifier that in this case represents some stored data. The *type* of the stored data is `String`. A *String* is a sequence of characters (i.e. a 'string' of characters joined together). `String` is a very commonly used *type* in Java. A single word (e.g. "Java") is a string; a sentence is a string ("Java is the language we use in this module"); even this whole chapter is a string – a string is *any* sequence of characters (and the characters do not have to be from the Latin alphabet – "σουβλάκι" is a string too). You may have noticed that you had to type '`String`' when you enter the line:

```
public static void main(String[] args) {
```

and you might think that `args` is some kind of String variable (you would be right); but that the '`[]`' does something 'funny' to it (you'd be right about that too).

How do we know to call it `name`? *We don't* – just like the program name, *we can choose what we call variables.* But it makes sense to *pick names that help you understand what data you are storing in them*. Because in this case the variable is supposed to represent data corresponding to somebody's name, it makes sense to call it "`name`".

To create, or *declare*, a variable in Java we need to state both the *type* and the what we want to call it. So this is a perfectly good declaration:

```
String name;
```

Notice that this is a simple statement, like those described above, and it ends with a ";".

The 'problem' with this is that we have not told Java *what* data is stored in `name` – just that it's a string of characters, and it's called `name`. Sometimes that's what we want to do – for example, we might not *know* when we create a variable what data we will be putting in it. However, very often we do and so we can *initialize* a variable at the same time we *declare* it:

```
String name = "Neal";
```

This sets the initial value of the variable `name` to the *String literal* "Neal".

Simple Assignments

Another way of doing this is in two steps:

```
String name;
name = "Neal";
```

That is, we first declare the variable, and then separately set it to the value we want – this is equivalent to the single statement that does both. Statements of the kind

```
variable = value;
```

in Java are called *assignments,* or *assignment statements.* There is no reason why we can't then *change* the value of a variable with *another* assignment. For example:

```
String name;
name = "Neil";
name = "Neal";
```

The value stored in name changes as the program runs, as you can see in this table:

Line	**Value stored in name**
`String name;`	-
`name = "Neil";`	Neil
`name = "Neal";`	Neal

It's a bit boring if we are only able to set variables to fixed *literal* values. We can do much more than this of course – more later. But for now, one thing we can also do is set one variable equal to another.

```
String fruit;
String citrus = "Lemon";

fruit = citrus;
```

First we create a variable called `fruit` with no value; then we create a variable called citrus with value "`Lemon`", then we assign the value stored in `citrus` to the variable `fruit`. At the end of this, the variable `fruit` has the value "`Lemon`" – and so does `citrus` because assignments copy data; so setting `fruit` to the value of `citrus` does not change `citrus`. The table below shows the steps.

Line of Code	Value of fruit	Value of citrus
`String fruit;`	-	Does not exist yet
`String citrus = "Lemon";`	-	Lemon
`fruit = citrus;`	Lemon	Lemon

Key Point: Variables Contain One Thing at a Time

You can change the value stored in a variable – we did that with the one called name that first held 'Neil' and then 'Neal'. The way you change the contents is by an assignment. But:

- Variables can only contain one piece of data at a time.
- Once you assign a new value to a variable, the old value is lost.

Aside: Blank Lines

Notice we've put a blank line in our program above – you never need to do this, but it can be handy to break your program into "logical blocks" and make it easier to read.

Aside: Variables vs Identifiers

A variable is an identifier, just like the program name. But just because all variables are identifiers does not mean all identifiers are variables! I make this point because people often confuse the two – there are other kinds of identifiers in programs that are not variables. For example, the program name – and others, we'll see later on.

Back to the Program…

We've been getting away from the example program – so let's get back to it. The line:

```
String name = in.nextLine();
```

Creates a variable called name and sets it equal (*assigns* it) to the value

```
in.nextLine();
```

This is something called a *method* in Java (you might have seen similar things in other languages called functions and/or procedures). It *returns* the string of characters that the user types in to the keyboard. So whatever the user types gets assigned to the variable `name` and ends up stored in it.

Scanner: Libraries and Input in Java

The method `in.nextLine()` is linked to two other lines in our program:

```
import java.util.Scanner;
```

and

```
Scanner in = new Scanner(System.in);
```

A *Scanner* in Java is a way of reading input, and it's part of a library or API (*application program interface*) that is supplied with Java. The `Scanner` is not automatically part of a Java program – we need to tell Java we want to use it. So the line

```
import java.util.Scanner;
```

tells Java "I want to use the Scanner library". (The `java.util` part just says which bit of the library `Scanner` is in.)

But this is not all we need to do – you can use Scanner to read from all sorts of data sources (keyboard, files, even network connections). So we need to tell Java what the *data source* is – in this case, the keyboard.

The line

```
Scanner in = new Scanner(System.in);
```

says make a new Scanner, name it `in`, and connect it to `System.in` (the keyboard). So once we've done this, we can specifically get our new Scanner called in to read from our data source (keyboard): `in.nextLine()`

Look carefully at the line:

```
Scanner in = new Scanner(System.in);
```

and you can see that it's a *variable declaration* – just like the one for a `String` called `name` we saw above. This one creates a variable called `in`, of type `Scanner`, and assigns it to the value `new Scanner(System.in)`, which means "make a new Scanner and connect it to the keyboard").

Just like our other variable declaration – `name` – and the names we pick for our programs, we can choose any name we want for the Scanner. We have picked `in` to reflect that fact that it deals with input.

Advanced Aside

I've skipped over some of the details here, but if you are already familiar with assignments, declarations etc., you might want to know that when we write things like `new Scanner(System.in)` we are creating a new *object*. Objects are a key part of programming languages like Java, and we'll be doing more about them – *Object Oriented Programming* – later in the module.

Keyboard and Screen

You might have spotted a symmetry to do with input and output – we use `System.in` to mean the keyboard; and `System.out` to mean the screen. So, `System.out.println("Lemon")` means print a line of text "Lemon" to the screen. You might be wondering if we can do other kinds of printing to the screen other than just whole lines of text; of if we can read different things in. We can, and more of that later.

The Final Line…

This line of our new program:

```
System.out.println("What is your name?");
```

should be obvious if you've understood our first program at the start of the chapter.

The next line was just explained:

```
String name = in.nextLine();
```

So that just leaves the last one:

```
System.out.println("Hello " + name + "!");
```

If you look at the output of the program above, it should be pretty clear what it does – it *joins* (or *concatenates*) the literal string "Hello " with the value stored in the variable name, and the literal string "!". The "+" is a binary operator, that means "join", and it joins together the strings on either side of it.

At this point you might be thinking – but what about numbers? Aren't we going to write programs that do arithmetic at some point? (Yes, and soon). Don't we need the "+" operator for that too? Yes we do – and it turns out we can use it for both: more later.

Run the Program

As before, enter the program using your software of choice, and compile and run it. This time though try changing some things once you've got it working.

- Try changing the messages that are printed out
- Try changing the variable names – *provided you do it consistently, it should still work* even if you pick "silly" names.
- Try changing the statement `String name = in.nextLine();` into *two* statements: one that just *declares* the variable `name,` and one that assigns it to `in.nextLine()`

EXERCISES

- Change the messages the computer prints out – get it to ask for different input and print out different messages.
- Look carefully at the line `System.out.println("Hello " + name + "!");` - why is there a space after "Hello"?

Sequence

It's probably pretty obvious by now that the lines of code in the programs we have seen so far are executed in order. So, in this program:

```
import java.util.Scanner;

class HelloYou {
     public static void main(String[] args) {
          Scanner in = new Scanner(System.in);

          System.out.println("What is your name?");
          String name = in.nextLine();
          System.out.println("Hello " + name + "!");

     }
}
```

The line

```
Scanner in = new Scanner(System.in);
```

is executed first; then the line:

```
System.out.println("What is your name?");
```

then:

```
String name = in.nextLine();
```

and finally:

```
System.out.println("Hello " + name + "!")
```

What about the other lines? They are not actually executed – instead they are instructions to the compiler to tell it how to create the executable program. So the line:

```
import java.util.Scanner;
```

Essentially tells the compiler “make sure you include the Scanner library”.

Helping People (and You) Understand Your Code – Comments

As well as indenting and picking sensible names for identifiers, there’s a another important way you can help people understand your code – adding *comments.*

Comments are text that is meant *only* for human readers – the computer *ignores* it. There are two ways of putting comments in Java and they’re shown in the program below

```
/* A program to ask the user their name
   and say hello to them
*/

import java.util.Scanner;

class HelloYou {
     public static void main(String[] args) {
          //Connect to the keyboard for input
          Scanner in = new Scanner(System.in);

          System.out.println("What is your name?");
          String name = in.nextLine();
          System.out.println("Hello " + name + "!");

     }
}
```

I've highlighted the comments in colour just to make them more visible. The first kind starts with /* and ends with */ - and is usually used for *multi-line* comments. Notice we've lined up the text within the comment block – sometimes people use extra * characters and write it like this:

```
/* A program to ask the user their name
 * and say hello to them
 */
```

Notice the *'s all line up. You can do it like this if you want to, or not – it's up to you.

The second kind of comment starts with // and goes to the end of the line. This is normally used for *one-line* comments – either on a line on their own, or after a statement on the same line.

Aside: Javadoc Comments

There is actually a *third* type of comments – called Javadoc comments – which start with `/**` instead of `/*`. We will get to these later on – they are used to help automatically generate *documentation.*

Because comments are not needed for a program to actually work, people are a bit too relaxed about them generally. But they are ***VITAL*** – and so there are so many KEY POINTS for this.

KEY POINT – Use Comments!

Most programs need at least some comments to help readers understand them – so put them in! As an incentive, the coursework marking schemes have marks allocated for this. And also industry coding guidelines generally have rule about it too.

KEY POINT – Don't Go Mad…

Having said that, sometimes I get sent programs with ***huge*** amounts of commenting, so the actual code is almost buried within it and barely visible. I assume they do this because they were told to in previous programming classes – but this is misguided and leads on to…

KEY POINT – The Code Should Mainly be Understandable on its Own

You should try to write your code – pick names for identifiers, choose ways of solving problems that mostly makes sense on its own. The comments should be an *additional* help. (You may be wondering how you do these things – they come with practice, don't worry.)

KEY POINT – Don't Say the Obvious

We haven't done anything about arithmetic yet, but hopefully it should be clear what this code does:

```
counter = counter + 1; // Add one to counter
```

Hopefully you can also see that the comment here is *completely pointless*. Don't do things like this – if the code is obvious, no need for a comment.

KEY POINT – Stick to the Indenting

Just as code is indented, so comments should be too. Some people with a bit of programming experience start all comments right at the beginning of lines, even when the code either side is indented. I assume, again, they've been told to do this – *but don't; it's not good practice and makes programs harder to read.*

KEY POINT – A Sensible Approach

If you're wondering now how you can avoid, too much, too little, too obvious, here's a strategy.

- Put a multi-line (/*..*/) comment before each 'logical piece of code saying briefly what it does. A 'logical piece' is a little hard to define, but normally it would be a few lines of code that are doing part of a single, simple, task.
- Put a single-line (//) comment after each variable declaration saying what it represents (unless you think it's completely obvious)
- Put a comment (whatever type is appropriate, depending on length) explaining any bit of code you think is tricky (maybe you found it tricky to write and get to work) so someone else – or you – can more easily understand it later.

This still involves some judgment on your part, and it's something you will get better at with practice, but it's a useful start and will stop you going too wrong.