Creating Your First Java Program

Installing Java

In order to be able to create and run Java programs on your computer, you will need to install the Java Development Kit (JDK). The JDK is a development environment for building applications, applets, and components using the Java programming language. The JDK includes tools which are necessary for developing and testing programs written in the Java programming language and running on the Java platform.

To install the JDK you will need to download the installation package and install it on your computer. The JDK is available free from Oracle, and the latest version can be downloaded from their website:

http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u3-download-1501626.html

On the **Java SE Development Kit 7 Downloads** page you will find the following section:

Java SE Development Kit 7u3

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux x86 (32-bit)	63.65 MB	jdk-7u3-linux-i586.rpm
Linux x86 (32-bit)	78.66 MB	jdk-7u3-linux-i586.tar.gz
Linux x64 (64-bit)	64.53 MB	jdk-7u3-linux-x64.rpm
Linux x64 (64-bit)	77.3 MB	jdk-7u3-linux-x64.tar.gz
Solaris x86 (32-bit)	135.96 MB	jdk-7u3-solaris-i586.tar.Z
Solaris x86 (32-bit)	81.4 MB	jdk-7u3-solaris-i586.tar.gz
Solaris SPARC (32-bit)	138.92 MB	jdk-7u3-solaris-sparc.tar.Z
Solaris SPARC (32-bit)	86.07 MB	jdk-7u3-solaris-sparc.tar.gz
Solaris SPARC (64-bit)	16.14 MB	jdk-7u3-solaris-sparcv9.tar.Z
Solaris SPARC (64-bit)	12.31 MB	jdk-7u3-solaris-sparcv9.tar.gz
Solaris x64 (64-bit)	14.46 MB	jdk-7u3-solaris-x64.tar.Z
Solaris x64 (64-bit)	9.25 MB	jdk-7u3-solaris-x64.tar.gz
Windows x86 (32-bit)	84.12 MB	jdk-7u3-windows-i586.exe
Windows x64 (64-bit)	87.41 MB	jdk-7u3-windows-x64.exe

Here you may download the version of the JDK which is compatible to your computer's operating system.

If you are a MAC user, you do not need to install the JDK – MACS come with the JDK installed from the factory.

IDE's (Integrated Development Environments)

Once you have the JDK installed, you can begin creating your own Java programs. You do not need any special tools for typing in your java code – you can use any word processor, or even a text editor like Notepad for typing your code.

This is fine if you've got a small program, but as your programs get bigger and include more classes, it becomes difficult to maintain with just a text editor. In such case it becomes necessary to use an **Integrated Development Environment (IDE)** . An IDE is a program that is designed to help programmers and developers build software. An IDE typically contains a code editor, compiler, debugger and other utilities, all integrated into a single package, and presents all this via a graphic interface. With an IDE you can do multiple tasks with the click of a single button.

There are many different IDE's for creating Java programs, but the most popular are Eclipse, Netbeans and IntelliJ IDEA. For learning purpose we recommend JGrasp due to its simplicity of use. The figure below shows a screen from the JGrasp IDE:

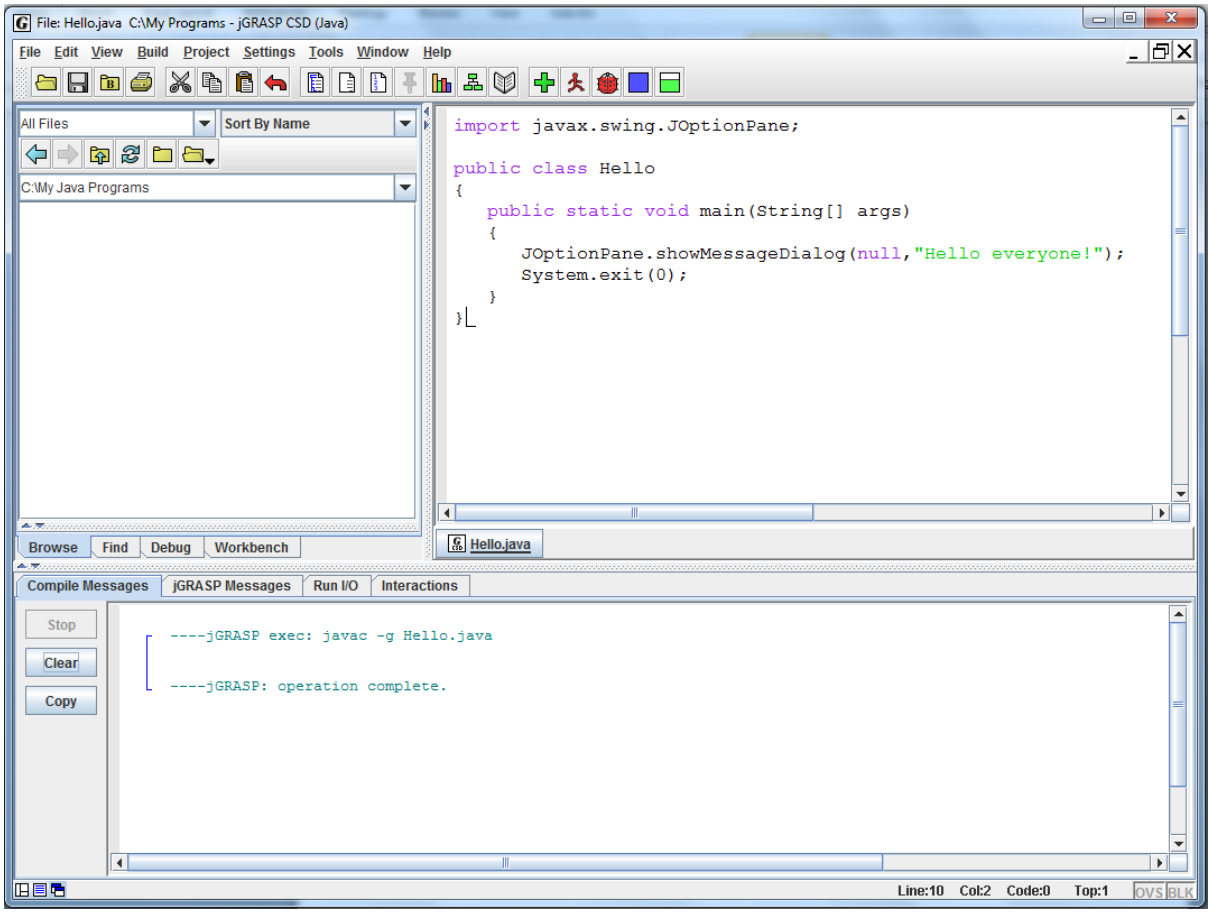


Figure 1-4

Installing JGrasp

JGrasp is available for free, and can be downloaded at the JGrasp Download page:

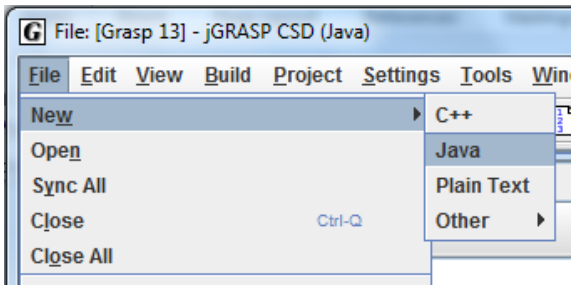
http://spider.eng.auburn.edu/user-cgi/grasp/grasp.pl?;dl=download_jgrasp.html

JGrasp also provides detailed, step-by-step documentation on installation and usage of JGrasp. The documentation is downloadable in PDF format, and can be found here:

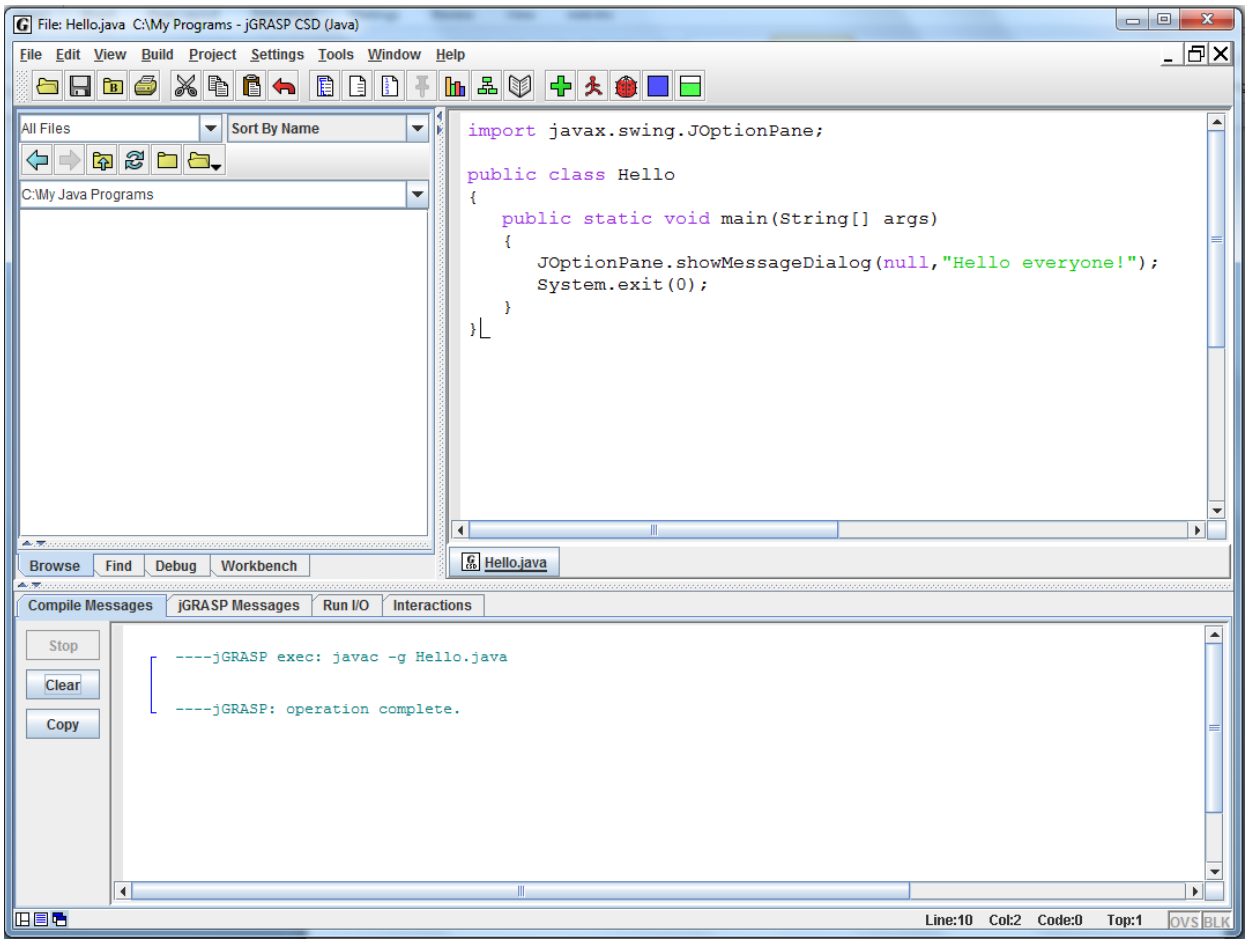
http://www.jgrasp.org/tutorials187/01_Installing.pdf

Your first Java Program

Open JGrasp and open a new Java class by selecting **File** -> **New** -> **Java** as shown below:



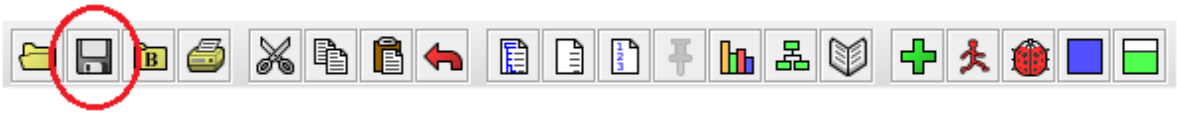
Type the following code into the code editor, as shown below:



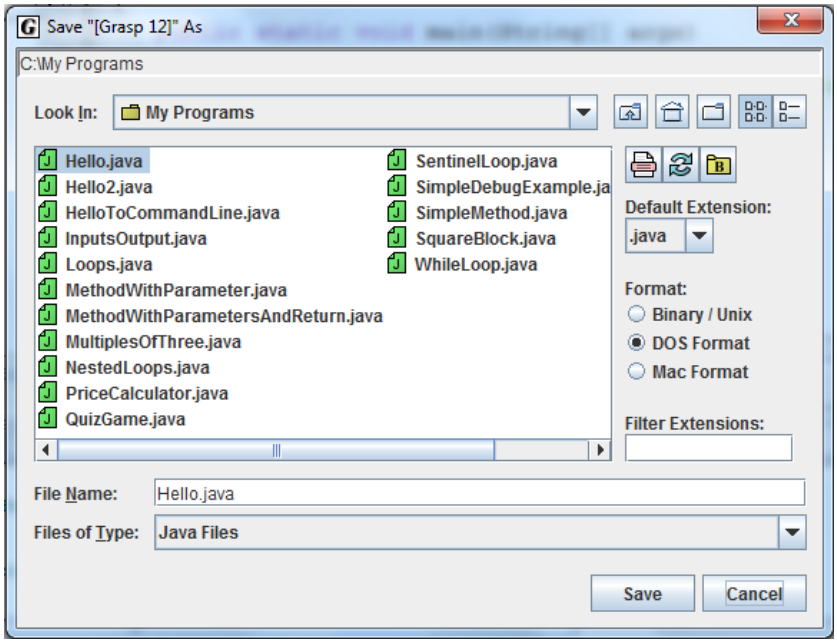
Next, save the program under the name Hello.java (note the capital letter “H” – this is very important!). Compile and run the program:

Saving, compiling and running your code in JGrasp

To save your program, press the **Save** button in the menu bar:



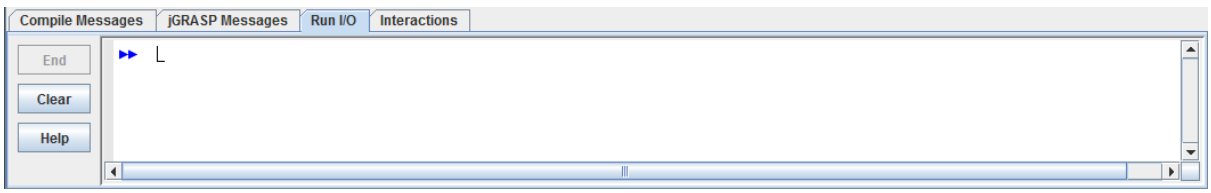
If your program has not been saved as yet, JGrasp will prompt you for a file name:



Notice that the file name in the **File Name** box is the same as the class name. This is one of Java's class naming rules: in a `public` class, the file name should *always* be the same as the class name. If this isn't the case, Java will return an error.

Once your class is saved, you may compile it by pressing the **Compile** button:

Your code will now compile. While your code is compiling, you can watch its progress in the Run I/O tab at the bottom of the screen:



If there are any errors in your code, they will show up in the **Run I/O** tab. If the compiler encounters errors in your code, it will stop compiling. You will need to fix the bugs before you can compile your code. See the **Debugging your code** section below for common coding errors.

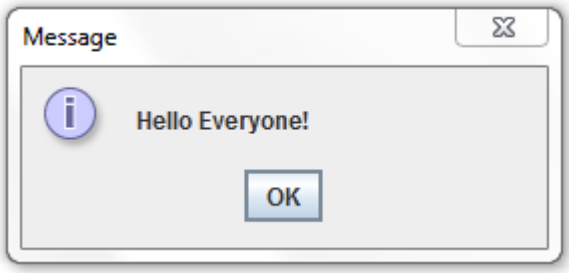
If your code is error-free, you will see the following in the **Run I/O** tab:

```
----jGRASP exec: javac -g Hello.java


----jGRASP: operation complete.
```

This indicates that the compiler has successfully compiled your code, and you may now run it by pressing the **Run** button:

This program displays the following dialog box:



When you click on the OK button, the dialog box closes, and the program will be terminated.

The code explained:

```
 1 import javax.swing.JOptionPane;
 2
 3 public class Hello
 4 {
 5    public static void main(String[] args)
 6    {
 7       JOptionPane.showMessageDialog(null,"Hello Everyone!");
 8       System.exit(0);
 9    }
10 }
```

(Note that the line numbers are for our reference only, and are not a part of the program.)

Line 1: The import statement instructs Java to import a different class, called javax.swing.JOptionPane, into your program. To understand why this is done, you need to know that Java programs are modular. This means that your programs are made up of many

sub-programs, all of which work together, very much like the parts of a machine. When you start a new Java program, it opens up with only the bare necessities. If you require more functionality, you can incorporate it into your program by "importing" it. In this program, we needed functionality to display a dialog box, so we imported the JOptionPane class, which provides that functionality.

The semi colon at the end of the line is necessary, as it signifies the end of the line for Java.

Line 2: This line has been left blank deliberately, to make the code easier for us to read. Java ignores blank lines.

Line 3: Every program in Java is made up of classes. A typical Java program can be made from a single class to thousands of classes. Our program is no exception. In line 3 we declare that we are creating a new class, which we are calling "Hello".

The keyword "public" is an access modifier. We will deal with access modifiers in Informatics 2b.

Line 4: Java classes, methods and code blocks are always wrapped in curly braces. The open curly brace in this line signifies the starting point of the class. It is complimented by the closing curly brace in line 10, which marks the end of the class.

Line 5: Methods are sub-programs within a program. In this line we are declaring a method called *main*. The *main* method is like the front door of your program – it is where Java begins reading and evaluating your code. Without a *main* method, your program will not run, and Java will give you an error. The keywords *public static void* are necessary here. We will look at these at a later stage.

Line 6: The open brace in this line marks the beginning of the *main* method. Remember: where there is an open brace, there must be a closing brace, otherwise Java will return an error. The closing brace on line 9 marks the end of the *main* method.

Line 7: Here is where the real action takes place. In this line, we are actually creating the dialog box with the text in it. You may be surprised to see that we are doing all this, with just one line of code! Actually, this is being done with not just one, but dozens, possibly hundreds of lines of code, but most of the code lives in the JOptionPane class, and is being run behind the scenes. All we are doing in line 7 is to call that code, and Java does the rest.

What is really happening in line 7 is that we are calling a method called *showMessageDialog* which lives in class *JOptionPane*, just as the *main* method lives in our *Hello* class. We are also passing it two parameters: *null*, and "Hello Everyone". We will explain parameters at a later stage.

So line 7 is actually instructing Java: in class *JOptionPane*, there is a method called

showMessageDialog. Please run that method for me here, and pass it the parameters *null* and “Hello Everyone!”

You will have noticed that the name of method *showMessageDialog* is very suggestive of what it actually does. You will find that all Java built-in methods are like this, and when you are writing your own methods, you should bear this in mind as well.

Line 8: Line 8 calls a method called *exit* in class *System*, which ensures that when the *OK* button is pressed, Java “exits” the program, that is, terminates it.

Debugging your code

If the compiler has encountered any errors when compiling your code, they will show up as follows:

```
 ----jGRASP exec: javac -g Hello.java

Hello.java:1: cannot find symbol
symbol  : class JoptionPane
location: package javax.swing
import javax.swing.JoptionPane;
                   ^
Hello.java:7: cannot find symbol
symbol  : variable JOptionPane
location: class Hello
                JOptionPane.showMessageDialog(null,"Hello everyone!");
                ^
2 errors

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

In this case, make sure that you are using uppercase and lower case letters exactly as shown in the code above. Java is case-sensitive, which means it sees uppercase and lower case letters as completely different letters. For example, the class name JOptionPane has an uppercase “J”, “O” and “P”. If you had to incorrectly type it as, for example, JoptionPane (with the lower case “o”), Java will not recognize it. This is a common error.

```
 ----jGRASP exec: javac -g Hello.java

Hello.java:7: cannot find symbol
symbol  : method showMessageDialoge(<nulltype>,java.lang.String)
location: class javax.swing.JOptionPane
                JOptionPane.showMessageDialoge(null,"Hello everyone!");
                            ^
1 error

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

Check that all your class names, method names and keywords are spelt correctly. In the above example, the name of the method `showMessageDialog` is spelt incorrectly as `showMessageDialoge` with an additional "e" at the end.

```
----jGRASP exec: javac -g Hello.java

Hello.java:7: ';' expected
                JOptionPane.showMessageDialog(null,"Hello everyone!")
                                                                    ^
1 error

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.
```

Another common error is to leave out the semi-colons at the end of a line.

```
----jGRASP exec: javac -g Hello.java

Hello.java:9: reached end of file while parsing
        }
         ^
1 error

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.
```

Check that every open brace has a partner closing brace, and vice versa.

Modifying the Output of your Program

In your first program, you called the method *showMessageDialog* to display a dialog box which outputted the words "Hello Everyone!" You did so using the following code:

```
JOptionPane.showMessageDialog(null,"Hello Everyone!");
```

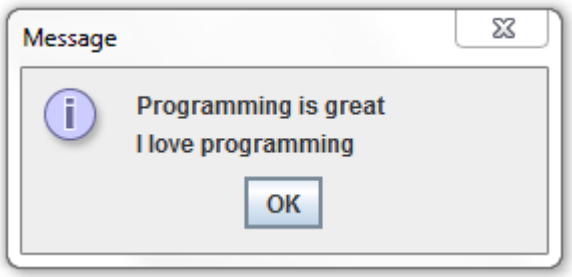
You can change the output of your program by replacing the words "Hello World" with something else. (Remember to keep the double quotes, or you will encounter an error!)

Activity 2.1

Modify your program to output the words "I love Java programming"

Output over multiple lines

Take a look at the dialog box below:



Notice that the output text appears on two separate lines. You can achieve this by using the *newline* character: **\n**

The newline character works as follows:

```
JOptionPane.showMessageDialog(null,"Programming is great\nI love
programming");
```

The newline character has been placed between the words "great" and "I" in the above code. This means that Java will start on a new line after the word "great", resulting in the output shown in the dialog box above.

Activity 2.2

Write a program called **Triangle** to output the following in a dialog box:

```
*
**
***
****
```

Calling a Method More than Once

As with any method, you may call the method *showMessageDialog* as many times as you require, and Java will run the method as many times as you call it. Consider the following program:

```
1 import javax.swing.JOptionPane;
2
3 public class Hello2
4 {
5    public static void main(String[] args)
6    {
7       JOptionPane.showMessageDialog(null,"Hello Everyone!");
8       JOptionPane.showMessageDialog(null,"How are you?");
9       JOptionPane.showMessageDialog(null,"It's good to meet you.");
10      System.exit(0);
11   }
12 }
```

This program, called *Hello2*, is almost identical to your previous program, *Hello*, with two additions: lines 8 and 9. These lines call the method *showMessageDialog* again.

Activity 2.3
Try to figure out what the above program does. Compile and run the code to check if you were correct.

Working with numbers

So far we've created dialog boxes which display text messages to the screen. A very important rule to remember is that whenever you need to output a text message to the screen, you need to type it within double quotes, as in the following example:

```
JOptionPane.showMessageDialog(null,"Hello");
```

If you attempt to display the word "Hello" without the double quotes, as shown below, it will result in an error:

```
JOptionPane.showMessageDialog(null,Hello);
```

What if we wanted to output numbers instead? Numbers can be displayed without the double quotes, so the following code will be correct, and will display the number "3" in your dialog box:

```
JOptionPane.showMessageDialog(null, 3);
```

You can also display numbers within double quotes:

```
JOptionPane.showMessageDialog(null,"3");
```

However, this will cause unexpected results when you are doing calculations (more on calculations a little later).

Outputting to the Command Line

The above programs all outputted information to a dialog box using `JOptionPane.showMessageDialog()`. Java can also output to the command line window using either of the following methods:

```
System.out.print("Hello")
```

```
System.out.println("Hello")
```

The two methods are very similar in that they display text to the command line, but they have one major difference: `println` will display the text as well as a line break, which means that the next output will start on a new line. Consider the following example:

```
1 public class HelloToCommandLine
2 {
3    public static void main(String[] args)
4    {
5       System.out.println("Hello");
6       System.out.println("Hello");
7       System.out.println("Hello");
8
9       System.out.print("Hello");
10      System.out.print("Hello");
11      System.out.print("Hello");
12   }
13 }
```

This code produces the following output:

```
Hello
Hello
Hello
HelloHelloHello
```

The first 3 outputs of the word “Hello” are produced by lines 5, 6 and 7. Because we used the `println()` method, each “Hello” is on a new line.

The last 3 outputs are on a single line, which is due to the fact that we used the `print()` method.