# Compiling and running programs from the command line

When you are working within a Java IDE you have simple buttons to click in order to carry out tasks like compiling and running programs. If, however, you are working from a command line, like a DOS prompt for example, you would use the **`javac`** command to compile a source file, and **`java`** to run an application[1].

For our example we shall use a simple "Hello world" program:

***HelloWorld.java***

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

Notice this program is not placed in a package.

Assuming you are in the directory containing this source code file, you can compile this class using the `javac.exe` tool with the following command:

```
javac HelloWorld.java
```

This will produce a Java class file (`HelloWorld.class`).

Assuming you are now in the directory that contains this class file you can then run this program with the `java.exe` tool as follows:

```
java HelloWorld
```

When running a class file you do not include the `.class` extension.

Figure 1 shows how this is done in a Windows® environment - `HelloWorld.java` resides in a directory named `hello`, which is a sub-directory of the root directory.

[1] When installing Java an **environment variable** called PATH should automatically have been set so the operating system can locate the necessary Java tools to compile, run and deploy Java programs. If this has not been done refer to your operating system's instructions for setting this variable The PATH variable should point to the *bin* folder in your JDK folder, for example *C:\ProgramFiles\Java\jdk-21.0.2\bin*.

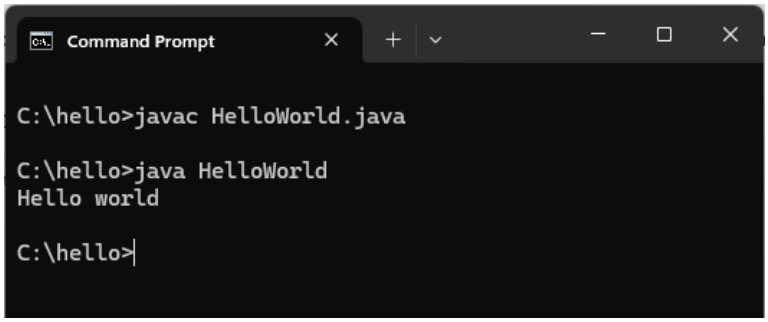


Figure 1

Working in a Linux®/Mac® environment, this would look somewhat different - backslashes would be replaced with forward slashes, for example.

When you run a class that resides in a package you must amend this slightly. As an example, imagine that we amended the `HelloWorld` program (now called `HelloWorld2`) by placing it in a package called `test`, as follows:

***HelloWorld2.java***

```
package test;

public class HelloWorld2
{
    public static void main(String[] args)
    {
        System.out.println ("Hello world");
    }
}
```

In order to compile our file, and for it to be placed in the correct directory, we should first create a directory called `test` and place our `.java` file into it. We should then compile it from within that directory. In our example we have created test within the `hello` directory (see Figure 2).

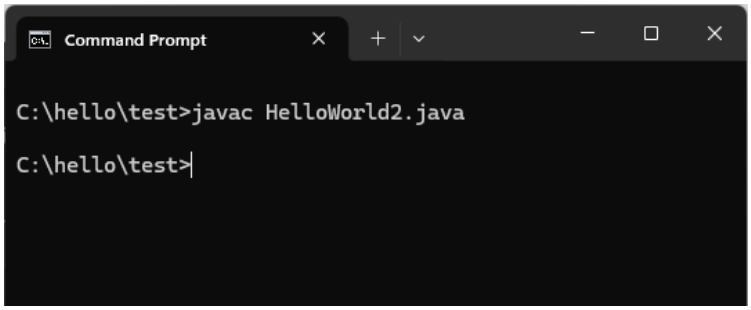


Figure 2

To run a file that is contained in a package, you need to be in the directory ***above*** the package directory. In this case, the directory `hello`. Additionally, you must append the class name onto the name of the package (with a '.' symbol);

```
java test.HelloWorld2
```

This is shown in Figure 3

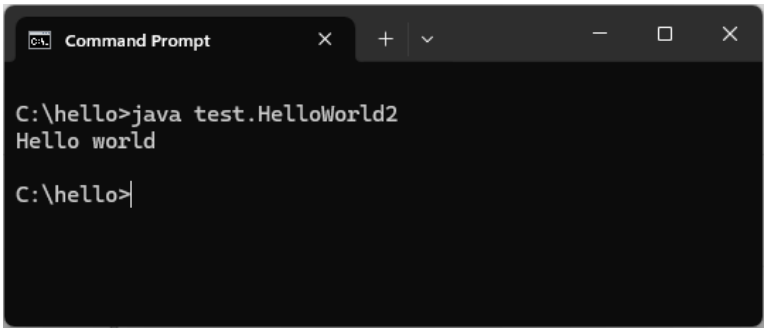


Figure 3

You should note that it is also possible to compile multiple files at once, separated by a space. For example:

```
javac MyProg.java YourProg.java
```

Alternatively you can use a wildcard:

```
javac *.java
```

Or for packages:

```
javac app/*.java test/*.java
```

If you do not wish to run the Java commands from the directories containing the relevant files you can set an environment variable called **CLASSPATH** so that it points to the relevant directory (or directories) and then run these commands from any directory. In Windows, it will be set, for example, as follows:

```
set CLASSPATH=\hello
```

See your operating system's documentation for further details.

Alternatively, you can use the `-cp` switch (note the minus sign). `-cp` is a shorthand for the `-classpath` switch (which can also be written as `--class-path`).

Let us assume that both the `HelloWorld` class and the `test` directory reside in the `hello` directory, which is a sub-directory of the root directory (see Figure 4).

```
root
├── app
│   ├── Rectangle.java
│   └── EasyScanner.java
└── tester
    └── RectangleTester.java
```

Figure 4

In figure 5 we are running first `HelloWorld` and then `HelloWorld2` from the root directory. In each case the classpath indicates the `hello` directory. Note that, as this is a sub-directory of the root directory, a relative path name is acceptable.

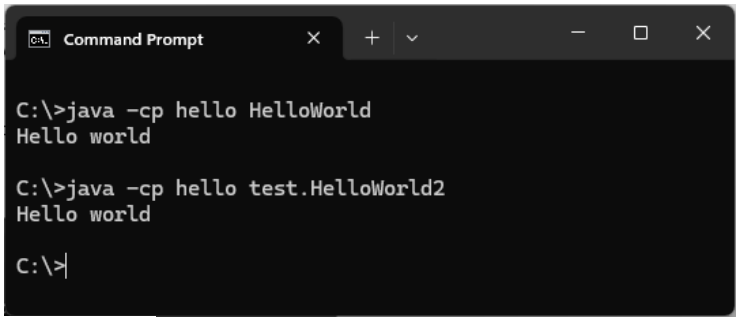


Figure 5

In many cases, an application will consist of many classes, not just a single class. If the classes are located in different directories it is possible to specify more than one classpath, each separated by a semi-colon (;) in Windows or a colon (:) in a Linux/Mac environment.

As an example, consider the `RectangleTester` from chapter 7, which required the presence of `the Rectangle` class and the `EasyScanner` class. Imagine that the `RectangleTester` was located in a folder called `tester`, and the `Rectangle` and `EasyScanner` classes were in a folder called `app`, and that each of these directories were sub-directories of the root. We could run `RectangleTester` as follows:

```
java -cp \app;\tester RectangleTester
```

This is illustrated in Figure 6

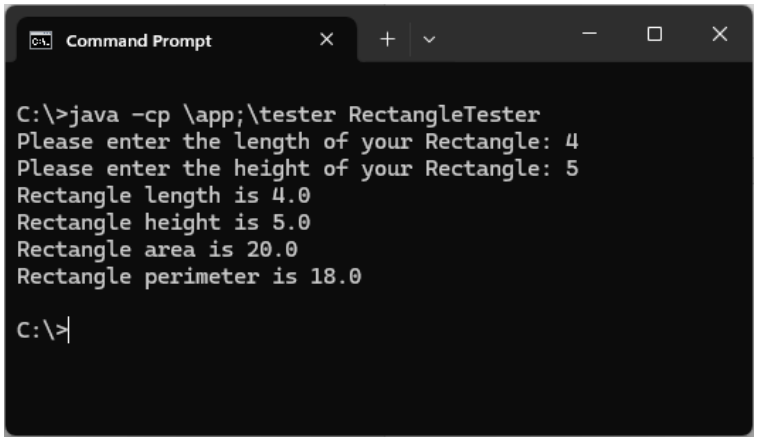


Figure 6

As well as running individual class files from the command line, we can also run `.jar` files. This is described in a separate guide.

## Running JavaFX applications from the command line

We will use as an example the `RectangleGUI` class from chapter 17. This class requires the presence of the `Rectangle` class.

Assume that both of these are contained in a folder called `rect`. We need to set the classpath to the current directory, which is represented by a single dot (.) so that it can find the `Rectangle` class, and we need to set the module path to the directory which contains the JavaFX files that we have downloaded (`\JavaFX16\lib` in this case). We also need to add module components to the runtime environment as explained in the guides. The command we need (in Windows) is:

```
java -cp . --module-path \javafx16\lib --add-modules javafx.controls,javafx.fxml RectangleGUI
```

This is shown in Figure 7.

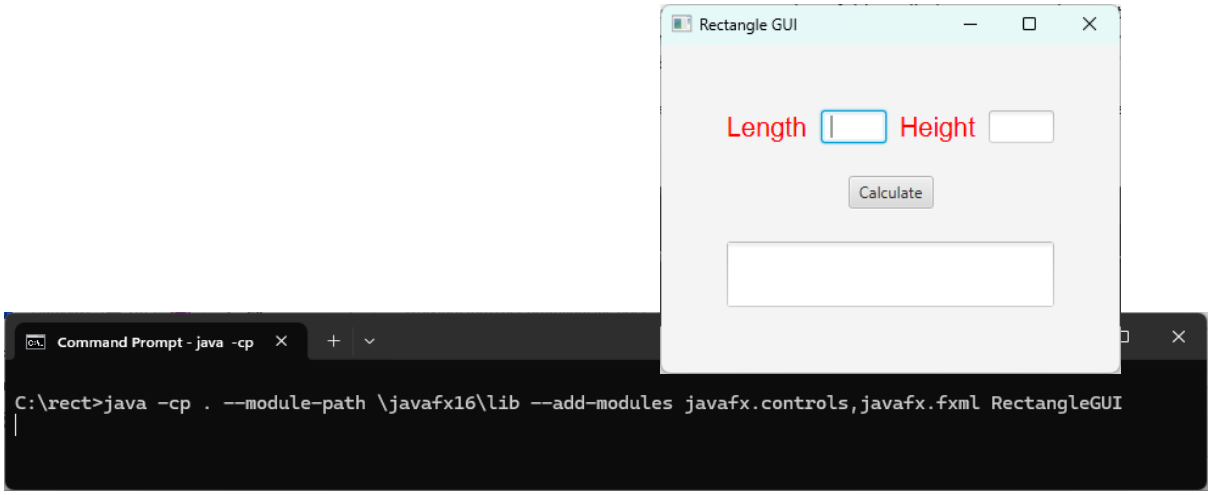


Figure 7

If you wish to run this program by double-clicking on an icon, then it is an easy matter to write this instruction into a batch file (Windows) or a shell script (in Linux/Mac). You can then create a shortcut to link to the batch file - if you don't wish to see the console, you can set the properties of the shortcut so that the console starts off as minimized.

## Command line arguments

Before ending this guide we can take a look at the parameter that we always give to `main` methods:

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

As you know, this means that the `main` method is given an array of `String` objects as a parameter. Values for these strings can be passed to `main` when you run the given class from the command line. Usually there is no need to pass any such strings and this array of strings is effectively empty. Sometimes, however, it is useful to send in such parameters. They are sent to `main` from the command line by listing the strings, one after the other, after the name of the class as follows:

```
java ClassName firstString secondString otherStrings
```

As you can see, the strings are separated by spaces. Any number of strings can be sent in this way. For example, if a program were called `ProcessNames`, two names could be sent to it as follows:

```
java ProcessNames Aaron Quentin
```

Were the strings to contain spaces, they must be enclosed in quotes:

```
java ProcessNames "Aaron Kans" "Quentin Charatan"
```

These strings will be placed into `main`'s array parameter (`args`), with the first string being at `args[0]`, the second at `args[1]` and so on. The number of strings sent to `main` is variable. The `main` method can always determine the number of strings sent by checking the `length` of the array (`args.length`).

The code for the `ProcessNames` class is shown below - it takes the array of strings and displays them on the screen.

***ProcessNames.java***

```
public class ProcessNames
{
    public static void main(String[] args)
    {
       if (args.length != 0) // check whther any arguments have been passed
       {
            // loop through all elements in the 'args' array
            for (int i=0; i<args.length; i++)
            {
                // access individual strings in array
                System.out.println("hello " + args[i]);
            }
       }
    }
}
```

We can run this program from the command line as follows:

```
java ProcessNames "Batman and Robin" Superman
```

Notice "`Batman and Robin`" needs to be surrounded by quotes as it has spaces in it, whereas `Superman` does not. Running this program would produce the following result, as expected:

```
hello Batman and Robin
hello Superman
```

This is illustrated in Figure 8: the `ProcessNames` class is in the hello `directory`.

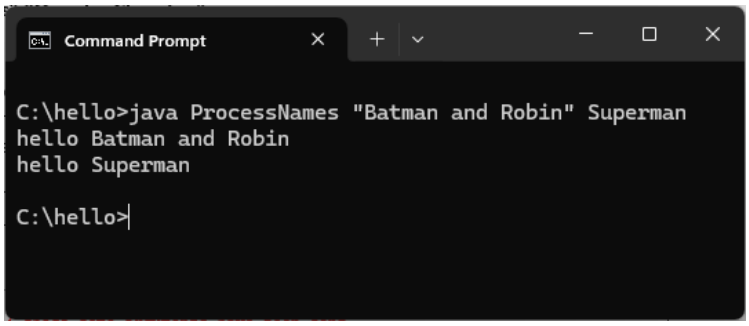


Figure 8