Some Simple Java Programming

Contents

1 Skills/Concepts **2**

1.1 Interpreted Languages . . . 2

1.2 Modules/Libraries . . . 2

1.3 Functions/Methods . . . 2

1.4 Console Use . . . 2

2 Game Design **2**

3 Preparation **2**

4 Hello World **3**

4.1 Background . . . 3

4.2 Process . . . 3

4.3 Conclusion . . . 3

5 Better Hello World **4**

5.1 Background . . . 4

5.2 Process . . . 4

5.3 Conclusion . . . 4

6 Easier Hello World **4**

6.1 Background . . . 4

6.2 Process . . . 5

6.3 Conclusion . . . 5

7 Graphical Hello World **5**

7.1 Background . . . 5

7.2 Process . . . 5

7.3 Conclusion . . . 6

8 More Lies! That's not Hello World! **6**

8.1 Background . . . 6

8.2 Process . . . 6

8.3 Conclusion . . . 7

1 Skills/Concepts

This module will involve creating some simple, hello-world style Java programs. This will involve the following concepts:

- interpreted languages
- modules/libraries
- functions
- console use

1.1 Interpreted Languages

Historically Java is considered an interpreted language because rather than compile into machine language that executes on hardware it is compiled into an intermediate language, Java bytecode, and the compiled bytecode is traditionally executed by the Java virtual machine. Howevwer, specialized hardware now exists that is capable of executing byte-code directly and compilers exist that can produce native executables from Java. So the definition is somewhat debatable.

1.2 Modules/Libraries

Java statements are placed in methods that are encapsulated by classes that exist in packages. Libraries in java are collections of packages that contain classes.

1.3 Functions/Methods

In the parlance of Java, functions are called methods. Methods are a logical collection of statements that generally have a specific purpose. By collecting statements into a single functional unit, it can be easily re-used. This avoids naively copying and pasting code, but more importantly it reduces duplicate code. Well written methods make it easier to re-use code and solve problems.

Generally, functions have a name, a return type, and take parameters. Examples come later.

1.4 Console Use

This module relies heavily on the command-line. You will be using a terminal emulator to execute the following commands.

Command	Description
ant	A tool for simplifying source compilation
git	A version control system
touch	Creates or updates access times
ls	Lists files
java	Executes Java bytecode
javac	Compiles Java code
javdoc	Generates Java documentation
mkdir	Creates directories

2 Game Design

Eventually this project will be a port of the “Hungry Shapes” game, so once again, we wont be performing any game design tasks that reflect how actual game design would be carried out.

3 Preparation

You should be fine booting up any Linux distribution as long as it has a Java SDK.

We will be using an existing git repository that contains a fair amount of code instead of producing a new program from scratch. This is to avoid copy and paste errors and allow us to cover material faster.

To get the code, execute the following commands:

```
git clone https://www.github.com/cyberpost500/java-project/
```

The code will be in the `java-project` directory.

4 Hello World

4.1 Background

Java code must be encapsulated in a class. Even a simple program that just outputs "Hello, World!" must consist of a class.

Before that code can be executed it has to be compiled into a class file by the Java compiler. Then it can be executed using the Java Virtual Machine.

4.2 Process

0. Check out the appropriate branch

```
git checkout 00_hello_world
```

1. Compile the code

```
javac HelloWorld.java
```

2. Execute the class

```
java HelloWorld
```

3. Take a look at the code

For example, using sublime: `subl HelloWorld.java`

4. Modify the code to print something else
5. Compile the code again
6. Execute the class again
7. Discard the changes

```
git checkout HelloWorld.java
```

4.3 Conclusion

This is just about the simplest Java program that can be created, aside from one that exits without doing anything. The purpose was to introduce the concepts and structure of a basic Java program and related tools.

5 Better Hello World

5.1 Background

Java code is usually organized into methods in classes, the classes are placed in packages, and the packages are usually kept in Java archive files, or jar files. The package names also correspond to directory names and the class names (as seen in the previous module) are stored in a file of the same name.

Classes, fields (or variables), and methods are documented using specially formatted comments that preceed those elements. This module introduces the aforementioned file structure and illustrates how these comments compile into documentation.

5.2 Process

0. Check out the appropriate branch

```
git checkout 01_better_hello_world
```

1. Compile the code

```
javac io/github/cyberpost500/helloworld/HelloWorld.java
```

2. Execute the class

```
java io.github.cyberpost500.helloworld.HelloWorld
```

3. Generate documentation for the class

```
mkdir docs
javadoc -d docs io/github/cyberpost500/helloworld/HelloWorld.java
```

4. View the documentation

```
x-www-browser docs/index.html
```

5.3 Conclusion

This is still just about the simplest Java program that can be created, aside from one that exits without doing anything. The purpose here was to show proper package hierarchy and introduce how comments should look. In addition, the `javadoc` tool was demonstrated. If you ever have to work on someon else's Java code, you might just end up having to generate this sort of documentation.

6 Easier Hello World

6.1 Background

Other than very rare exceptions, no one executes the `java` compiler by hand, especially for projects that incorporate thousands of files. Instead, developers tend to rely on other programs to help simplify the sort of actions that are frequently carried out when building, running, and testing software.

One such tool is Ant. It's a sophisticated build tool with a somewhat straightforward XML based configuration file.

6.2 Process

0. Check out the appropriate branch

```
git checkout 02_easier_hello_world
```

1. Compile the code

```
ant build
```

2. Execute the class

```
ant run
```

3. Generate documentation for the class

```
ant docs
```

4. View the documentation for the project build file

```
ant -p
```

5. View the `build.xml` in an editor

6.3 Conclusion

Although there weren't very many input source files to manage, it should still be somewhat apparent that using the `ant` command to manage the project is much easier than having to remember and type several different commands with all of the correct arguments.

7 Graphical Hello World

7.1 Background

As exciting as text can be, the goal of these modules is to eventually reach a point where the program being developed is a Java port of the "Hungry Shapes" Python game. Whereas python made use of the Simple Direct Media Layer, instead of using SDL, Java has several built-in components for accomplishing similar goals.

To create a window, we will be using the `JFrame` class. This provides a very basic window that can be constructed, made visible, resized, and eventually closed and destroyed.

7.2 Process

0. Check out the appropriate branch

```
git checkout 03_frames
```

1. Compile the code

```
ant build
```

2. Execute the class

```
ant run
```

5. View the `MainWindow.java` file in an editor
6. Change the window title to something else, save it, and run the changes
7. Comment out line 14 (setDefaultCloseOperation) by putting two slash characters in front of it.

For example:

```
        //frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

8. Save it, run it, and when the window displays, close it. What happened in the terminal where the program was executed? Uncomment the line before resuming.
9. Modfy the window size

Line 15 controls the window size. Modify it, save it, and run it.

10. Modify the window title

Line 13 controls the window title. Modify it, save it, and run it.

11. Revert the changes

```
git checkout src/io/github/cyberpost500/hungryshapes/MainWindow.java
```

7.3 Conclusion

This is the minimal amount of code necessary to create a window, set the title, set the size, make it visible, and then have it respond appropriately to being closed.

8 More Lies! That's not Hello World!

8.1 Background

Maybe you noticed the "Hello World" application produces no such message.

Check out the `04_hello_world` branch to see a version of the `MainWindow` that actually displays "Hello, World!".

8.2 Process

0. Check out the appropriate branch

```
git checkout 04_frames
```

1. Compile the code

```
ant build
```

2. Execute the class

```
ant run
```

5. View the `MainWindow.java` file in an editor
6. Modify the text that gets displayed

```
                g.setFont(new Font("Default", Font.PLAIN, 32));
                g.setColor(Color.BLUE);
                g.drawString("Hello, World!", 10, 10);
```

Line 30 creates a new Font (derived from the system default font), sets the style to PLAIN, and the size to 32. Line 31 sets the font color and then line 32 draws the text. Modify these values and witness the effect by running the program again.

7. Revert the changes

8.3 Conclusion

Now the program is a basic, graphical “Hello, World”. Some explanation is necessary regarding the program flow of when this code is executed, and a brief description of inner classes is needed. This will be provided in the form of a discussion rather than embedded in this document, but to summarize:

An inner class is a class defined within another class. Line 28 defines an Anonymous inner class in that there is no name given, instead an existing class, the `JPanel` is used as a base class for a new, un-named class whose definition appears in-line.

This new class is exactly like the `JPanel` class, except for two minor differences: first, it has access to everything from the `MainWindow` class, and second, it has its own definition of the `paintComponent` method from the `JPanel` that defines custom behavior to execute instead of the default `paintComponent` body.

The `@Override` you see on line 29 is what’s called an annotation. It provides additional information regarding the line it is on or the line that follows for both the Java compiler and for the reader. The `Override` annotation tells the reader (and the compiler) that the `paintComponent` method already exists in the parent class and that what follows will over ride that existing method.

And as for when it is called, the `paintComponent` method is called whenever the `MainWindow` needs to redraw it’s content due to some child portion being invalidated. For example, if you resize the window to smaller than the display area and then enlarge it, or just when another window is dragged over the `MainWindow`.