CS101 Lecture Notes, Mock
Overview of Java Programming

Hello, World

The first program that many people write is one that outputs a line of text. In keeping with this vein, we will start with a program that prints, "Hello, world". This program is also in the textbook, but I have changed it slightly to impart more of an Object Oriented flavor. First, here is the program in its entirety. It actually consists of two files:

File: HelloWorld.java

```
// Put your name and date here
// followed by a description of the code
// This is a Hello World object that has a single
// method to print "hello world"
class HelloWorld {
      public void printHello() {
            System.out.println("hello, world");
      }
}
```

File: UseHello.java

```
// This file is the main object, when the program
// starts it beings executing in main
class UseHello {
      public static void main(String[] args) {
            HelloWorld myHello;   // Declare HelloWorld object
            myHello = new HelloWorld();
            myHello.printHello();
      }
}
```

Before we try to compile and run this program, let's go through a description of what is in here. First, any part of a line that begins with // is considered a comment and from the // on to the end of the line, the text is ignored. Descriptive text about what the program does should go into the comment fields. We'll use these to describe the input, output, and dependency behavior of the program. Also use comments as a place to type your name with the file!

A Java program can be composed of many files. In this case, our program consists of two files. Each file corresponds to what is called a **class**. A class is Java's way of defining an object. However, a Java program should have only one file that contains the special function called **main**. In our case, this is in the file UseHello.java, so let's start there.

After the comments, the first line of UseHello.java is "class UseHello". This defines the name of the class (i.e. the object). **The class name must match the name of the file!** Java is case-sensitive, so Java will complain about a file named "usehello.java" but the

contents of this file contains “UseHello” instead. We will place each class object into its own separate file.

The left curly brace is used to denote where the body of the class begins. The right curly brace denotes where the body of the class ends. It is common notation to put the right end brace on a line on its own in the same column as where the class begins, so one can see which curly brace matches with what statement. There is great debate regarding the placement of the curly braces. At this point it is worth mentioning that the compiler doesn’t care about **whitespace** between instructions. Whitespace is spaces or carriage returns. You can add as many spaces or blank lines as you wish so that the program is easier for humans to read. You need at least one whitespace character to separate instructions, but others will be ignored. If you wanted to, you could write the entire program on a single line! This would not make it very human-readable, but the compiler will not care.

The next line is “public static void main(String[] args) {“. This defines a function called “main” in the UseHello object. As indicated earlier, main is a special function. This is where your program begins execution! In Java, functions are often referred to as **methods**. I will use these terms interchangeably. You can think of a method as a collection of code that does some specific task. The main method has a few terms that will look cryptic. You’ll learn more about these things later, but for now you can consider this as “boiler plate” that you will just put in all your programs to make them work. Nevertheless, here is a short description of what these terms mean:

public	- The method is made available to anyone that wants to use it
static	- Only make one copy of this method
void	- This method should not return any value (e.g., functions can be designed to return values, f(x))
String[]	- A String refers to a block of ASCII characters like a word or sentence. The [] indicates that we want an array, or a collection, of many strings.
args	- This is a variable name used to refer to the strings In most of our programs we won’t use this, but it refers to command-line arguments passed to our program

The next line is “HelloWorld myHello; “. This entire line is called a **statement**. Every statement must end with the semicolon, also known as a statement terminator. One of the things that gets tricky for beginning programmers is where to put the semicolons; it just takes experience to learn where the semicolons go!

This particular statement declares a **variable** to be of type “HelloWorld”. HelloWorld is another object that we are defining in a separate file. In other words, we are going to create an instance of this other object. At this point, we are only defining that we would like a HelloWorld object, and we will reference this object through the name “myHello”.

The programmer has the luxury of picking whatever name is appropriate to assign to this variable.

The next line "myHello = new HelloWorld();" actually creates a new instance of the HelloWorld object. This allocates the memory to put this object into memory and also runs any code that might be associated with initializing the object. The key operator here is the word **new** which tells the Java compiler to allocate the object.

The next line "myHello.printHello();" invokes the function, or method, named "printHello" that is defined for the "myHello" variable. Since myHello is defined to be of type HelloWorld, we will be invoking the printHello function in the HelloWorld object. The parentheses are used to indicate that this is a method or function that we are invoking. This will do the actual work of printing out "Hello, world."

The final lines in this file are closing curly braces to denote the end of the main function and the end of the class.

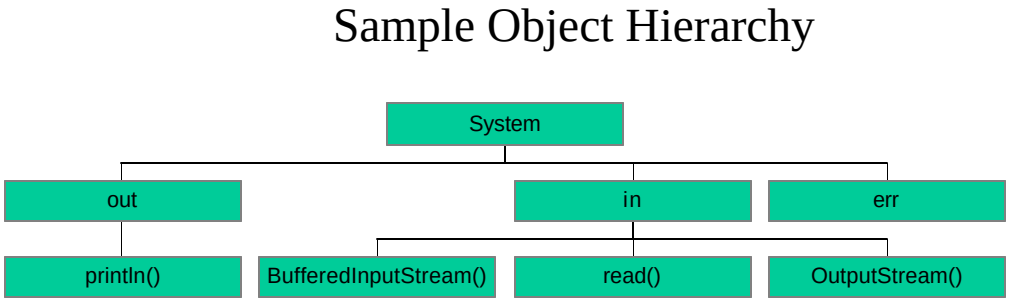
Now let's turn our attention to the HelloWorld.java file. This file does not contain a main function, because we can have only one main function in a Java program, and we already put one in UseHello.java. This file defines a class object called HelloWorld. This class also defines a single method, or function, called printHello:

```
public void printHello() {
      System.out.println("hello, world");
}
```

As with the main function, the keyword "public" indicates that anyone may invoke this method. The keyword "void" indicates that there is no return value for the function. Finally, we give the name of the function, in this case, "printHello". The parenthesis identify this as a function.

All this function does is output the text "hello, world". To do this, it uses the most basic Java operation to produce output: System.out.println("….text string here…");

The dotted notation of using periods starts at a high-level Java object, and each dot indicates a more specific Java object. Objects form a hierarchy. For example, here is a small object hierarchy for System:



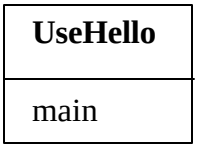
System.out.println() invokes the method to print a string of data to the screen. Similarly, System.in.read() would invoke a function to read input from the keyboard. Each "dot" moves us down the hierarchy to a more specific function or object.

To print a string, note that we used double-quotes. Any double-quoted block of characters is considered a literal string, and is created by concatenating together the appropriate ASCII characters.

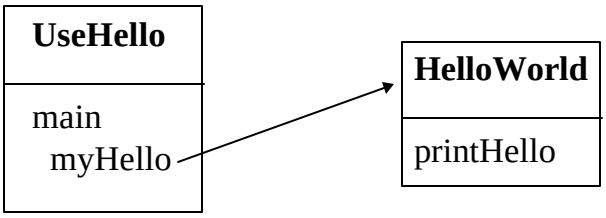
Finally, the file is terminated with right curly braces to close the printHello() method and the class. Once again, you might not understand everything that is going on here, but it should make more sense as we go along.

Execution

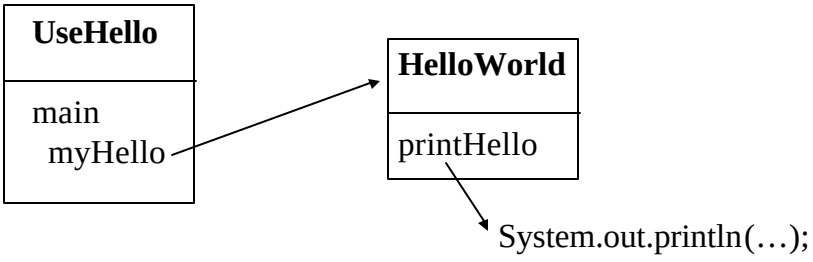
When we first run the program, initially Java will create an object for UseHello. This is depicted below:



As we execute the code in main, the first thing we do is create a variable of class HelloWorld. The **new** command creates the actual instance of this object:



Finally, when we invoke myHello.printHello(), we execute the code contained in the printHello() method of the HelloWorld object:



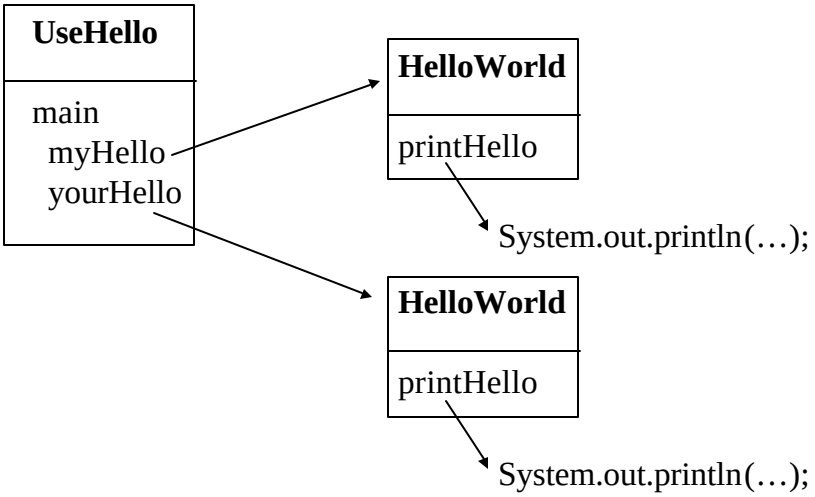
You might be wondering why I have put the code to print out "Hello World" into a separate object. Couldn't we just put it into UseHello directly as shown below?

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

The answer is yes, we could do this! However, I've created a separate object to show off the object-oriented properties of Java. By splitting off the code into an object, we can create multiple "Hello World" objects if we like. Consider the following:

```
class UseHello {
      public static void main(String[] args) {
            HelloWorld myHello;
            HelloWorld yourHello = new HelloWorld();
            myHello = new HelloWorld();
            myHello.printHello();
            yourHello.printHello();
      }
}
```

In this example, we are creating two objects of type HelloWorld. One is called "myHello" and the other is called "yourHello". Notice that Java allows us to create a new instance of this object on the same line that we define it, instead of using up two lines of code like in the original version. This code can be depicted in the diagram below:



We've created two objects of class HelloWorld. Each one invokes their own printHello function. This is not very interesting for the HelloWorld program, but imagine if the HelloWorld class was more complicated. For example, let's say it is a class that computes someone's income tax. If we wanted a program that could compute the income

tax for two people (say, Fred and Mary), then we could create two of these objects, one for Fred and one for Mary and each could compute their taxes individually using their own object. However, we are sharing a common code base in defining the class, so this greatly facilitates code sharing!

Compilation

The process of entering and compiling your program is different depending upon what development environment you are using. We'll be using a most primitive development environment – using a text editor in conjunction with the java command-line compiler. Other development environments include programs that allow more graphical views of your code. These are called IDE's (Integrated Development Environment). You'll want to use an IDE if you continue to develop larger and more complex Java programs, but the text editor method will work for the simpler programs we'll study in this class. You are also welcome to use an IDE as well, but we won't explicitly support one in class. The book describes using an IDE called Kawa. There are also IDE's you can download from the java.sun.com web page for free.

To compile your programs, first enter them using a text editor. For example, you could use vi under unix, or Notepad under Windows. The CS lab has a text editor called TextPad that you can use too. Make sure the name of the file matches the name of the class.

After the files have been created, compile them using the javac command. The arguments are the names of the Java source code programs:

- javac HelloWorld.java UseHello.java

If there were any errors, the compiler will complain and tell you it encountered problems. If all goes well, you'll be returned to the prompt and you should now have a files named "HelloWorld.class" and "UseHello.class" in your current directory. These are the compiled versions of the Java source code. To run them, use the Java interpreter:

- java UseHello

This will run the program. Note that we didn't enter the ".java" or the ".class" upon executing the program. This command invokes the Java interpreter which will convert the java byte code into native machine code and then run it.