Developing a Java program

There is a tradition in computer science, that the first application a student of a new language will develop is the so-called: "Hello World" application. In this application, the goal is to get the words "Hello World" to print to the screen. The earliest known reference to this application is in Brian Kernighan's 1972 *"A Tutorial Introduction to the Language B"*. This tutorial described a language that ended up being a precursor to the C language that was developed a few years later. Many of the language elements and behaviors of java (and many other modern programming languages) are based on the C language.

Here is the HelloWorld program written in java:

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

While even this most basic program might seem daunting at first glance. Fear not! You will understand and be able to create these programs in seconds in the very near future. This program is a collection of words and characters. This collection of characters is intended to be read and formulated into instructions for a computer to execute. Computers are not as forgiving readers as humans, however. This fun excerpt that routinely gets passed around email lists illustrates the difference between human and computer reading abilities in the face of typos:

> "I cdnuolt blveiee taht I cluod aulaclty uesdnatnrd waht I was rdanieg. The phaonmneal pweor of the hmuan mnid, aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, t he olny iprmoatnt tihng is taht the frist and lsat ltteer be in the rgh it pclae. The rset can be a taotl mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe. Amzanig huh? yaeh and I awlyas tghuhot slpeling was ipmorantt! if you can raed tihs psas it on !!"

While the majority of the words in this passage were misspelled or had other various mistakes, you may have been able to understand and get meaning from all or most of the passage. However, a computer given the program below, would not be able to do a thing with it:

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

A simple missing letter 'i' in 'println' would render this program unusable by the computer. When writing programs for a computer, every word and every letter must be precise. Anything out of place or misspelled will cause the program to be incompressible to the machine, or worse, cause the program to execute and produce results that are not what the programmer had intended.

Practice writing your HelloWorld java program in the box below:

The Programming Cycle

Developing a program is a four-step cycle:

1. edit (either write a new program, or change an existing program)
2. compile (encode program into bit string form)
3. run program through tests to see how well it works
4. debug – find error(s) in the test results and deduce their cause(s)

We will look at each of those four steps in detail now.

Edit: Step 1 of writing a program is to type the program into a file using a text editor.

A text editor is a program that is similar to a word processor. Text editors allow you to type and edit text, but they provide custom features that facilitate computer program development, rather than the collection of features important for writing term papers and novels. In particular, text editors:

- provide many of the same features as word processors, such as allowing the entering and editing of text, support for cut-and-paste, search-and-replace, and the capacity to save and open files.

- provide some features very useful for programming, such as automatically indenting in complex ways, color-coding certain syntax features, ensuring you have a close-parenthesis for every open parenthesis, etc.

- save the program as plain text. That is to say, the file a text editor saves contains nothing but the characters you entered (without extra formatting information), and thus should be conveniently viewable using a different text editor. This is in contrast to using a word processor, where generally if you save a file in a word processor, there's various extra formatting added to the file, and so either you have to use that particular word processor to view your file later, or else someone needs to do a lot of work to enable some other piece of software to read your file.

Some examples of text editors are *pico*, *emacs*, *xemacs*, *vi*, and *vim* on Unix. *TextEdit*, *TextWrangler*, and *BBEdit* on Mac OS X, and *NotePad* and *TextPad* on Windows. In addition, some of those editors are actually cross-platform; Xemacs, for example, is available on all three platforms. A good text editor can save you a great deal of time in developing your programs, so it is worth your time to become proficient in the use of a good text editor! A little research now into the features and complexity of the various text editors currently available for your preferred computing platform will save you lots of time editing programs in the future, and it will make your software development experience much more enjoyable. Learning how to use a text editor well is one of the many situations where much of the work will be left up to you. You can accomplish most of the programming tasks in this course using the most basic text editor features but you'll find that spending a bit more time learning more features of your chosen text editor can save you a great deal of time in the future.

Once you type in your program and save the file, the file is known as a ***source file*** or ***source code file*** and the text inside that file is known as ***source code***. In Java, we will name our source code files with the .java suffix, so our files will have names such as GardenTips.java or MyAwesomeApp.java. Our first program above would go into a file named HelloWorld.java, because the name on the first line after public class is HelloWorld.

Compile: Step 2 of writing any program is to compile the source code for the program.

When you **compile** a program, you are encoding the source code into bit string form so that the machine can understand it. The process of encoding your source code into bit string form is known as **compiling** and the software that does this encoding for you is called a **compiler**. This step is necessary because the source code is meaningless to the actual hardware. As we outlined in previous chapters, your computer hardware is nothing but a collection of transistors and wires, so you need to convert your instructions (the program you've written) into a form that the hardware can make sense of – namely, bit strings. In Java, the encoded or compiled file has a .class suffix and otherwise has the same name as the source code file. For example, if you sent the file HelloWorld.java as input to the Java compiler, the output would be a file named HelloWorld.class, and it is this file that would contain the encoded version of your program.

The compiler's job is actually more than just encoding your program. This is because the compiler can only encode complete and syntactically correct Java programs. If you sent the compiler the following:

```
LBIELK SLKwljvaj 873%(&857 slul ll7l73kjgl7 { oiu fau uuuaa52g }
```

then the compiler can't produce an encoded, since it isn't a Java program to begin with – it's just a bunch of random characters. There are particular rules about what is and what isn't a Java program – our first "Hello World" example earlier is indeed a Java program, and the random garbage above is clearly not a legal program. But, the second "Hello World" example, the one with the single character typo, isn't a legal java program either, although it is closer to being a legal program than is the above.

Programming languages are different than languages that people speak (e.g., English, Spanish, Japanese, etc.). We as people understand these natural languages, so you might think they'd be ideal for writing programs. Unfortunately, natural languages are not precise enough. The same sentence can have multiple meanings depending on context, the tone of voice it is spoken in, etc. Whereas when we specify a set of instructions to the computer, we don't want the computer to assume a different meaning than we intended. Therefore, we have very exact specifications for programming languages, so that it is very clear what is a legal program in that language and what isn't a legal program, as well as exactly what each legal program should do when it is run.

You can imagine a spectrum of possibilities:

natural languages
high-level languages
bit strings

The far right end of the spectrum is where we have instructions encoded as bit strings – the only language the machine actually understands. For this reason, the language of bit strings that have meaning to the machine is also also known as **machine language.** The far left end of the spectrum is the collection of languages we speak, which are very understandable to us but not precise enough for usage as programming languages. Close to natural languages, but not quite so far to the left, are the languages in which most modern programming is done. These programming languages are called **high-level languages** because of how much their syntax resembles natural languages instead of bit strings. These languages are closer to our level of speech, than bit strings would be. It is much easier to develop programs in a high level language than it would be to develop programs by writing out bit string after bit string from scratch.

If you send a file of source code that is NOT a valid program for that language, then the compiler cannot produce any encoded version of your program, since you haven't actually given the compiler a valid program from which to start. However, what the compiler will do for you, in this case, is tell you what locations in your program violate the syntax rules of the language. That is, the compiler will tell you what lines had errors, and what it thinks those errors are. For example, if the earlier "Hello World" program with the one typo was sent in as input to a Java compiler, the Java compiler might tell us that there is an error on line 5, and furthermore, would tell you that the word "prntln" on that line is not something the compiler understands.

These kinds of errors – where a program is written that does not conform to the specification of the language – are called **syntax errors**, and the way you fix them is by changing your source code so that what you have written does conform to the specification of the language. **The compiler's job is to identify and convey to you what syntax errors you have in your source code, and, if you don't have any syntax errors, produce an encoded version of your source code.**

Run: Step 3 of program development is to ***run***, or ***execute***, your program.

At this point, you have an encoded version of your source code. That is, you've translated your source code so that it is in a form the machine can understand. However, there are many different kinds of machines. Some of these machines are similar to each other – for example, a computer built around the newest Intel Pentium processor (which is the processor most commonly used in Windows machines), probably isn't all that different from a computer built around last year's Intel Pentium processor, and so the encoding for the two machines is probably the same, or at least, very very similar. On the other hand, the difference between a Pentium processor, and a processor used in a Cray supercomputer or an Android or iOS device, is somewhat more significant. Because those processors are so different from each other, the encoding you use for one machine is different than the encoding you use for the other machine.

This is a big part of why software you can buy at the store for a Windows machine, may not run on a Macintosh, and certainly won't execute on a Cray supercomputer or an Android or iOS phone or tablet. If you were to write a story in English, and translate it to Spanish, someone who only understands Japanese isn't going to understand the original story, nor will this person understand the Spanish translation. Now, you could go back to your original story and create a second translation if you wanted – a translation to Japanese – but if you have only provided the one translation to Spanish, then anyone who can't understand the original English version and who also can't understand the Spanish translation, cannot read the story.

Similarly, you can have the compiler translate your source code to run on a Pentium processor, but a Cray processor will not be able to make sense of the Pentium translation. You can run a second translation, and translate your source code to the Cray machine code if you want, but if all you've done is the Pentium translation, then a Cray processor cannot run your program. It doesn't understand the original source code and it doesn't understand the encoding for a Pentium.

There is another option. It is possible you could invent a new *intermediate* machine and translate your source code into the machine code for the intermediate machine. In this case, the machine code you end up with, would not run on a Pentium, nor would it run on a Cray. In fact, it is designed to only run on this new processor which you dreamed up, but which no one has built yet. Why would one do this? The difficult part about performing an encoding, is going from the high level language to some machine language. Once you have a machine language version of a program, translating that to a different machine language, is relatively easy by comparison. So, you could do the hard work of translating your code for a machine (your imaginary intermediate machine, or **virtual machine)** and then put the code out on the internet. People who want your program could then download it and would only have a small amount of work to have to do. They would need to translate your encoding to the encoding for their own processor. For example, if your computer had a Pentium processor, you could download an encoding for the virtual machine, and then further translate that for your Pentium machine. Since the translation from one machine code to another is not hard, your task is not hard. Someone else could download the encoding for the virtual machine and then further translate that run on their Cray supercomputer.

This is the Java model. To run a Java program, you will have to run a separate program – the Java virtual machine – which is a piece of software that (among other things) translates the encoded .class file into a form that can run on the actual machine you are on. The advantage to this process is that it makes the .class files somewhat portable – they can be run on any computer, as long as that computer has virtual machine software installed. If you write a program in Java, you don't need to release one version for the Pentium processor, and another version for the Cray processor. You can simply provide the .class files, and they should run on any processor that has a virtual machine available. The downside is that running your program can take a bit longer, since the virtual machine is performing the additional translation as your program runs. If speed is your top priority, then Java may not be the best language to use. But if portability is a higher priority, and you can afford to have things slow down a little, then the Java model might be a useful one for you.

Debug: Step 4 of program development is to debug your program (if necessary) and then return to step 1.

Once you have a program and have translated it to machine language, you then can run the program, as we just discussed. However, the machine might not do what you want. Certainly, the machine will do what your program tells it to do, but if you have specified the wrong sequence of instructions in your program, then of course the wrong things will be done.

Computer scientists have an idiom for this idea: "garbage in, garbage out." These are known as logic errors – when the program you've written doesn't do what you intended it to do. When this happens, you will need to change the program to one that does do what you want it to do. That is, you need to figure out where you made an error in your chosen sequence of instructions, and then correct that error. This is called **debugging** your code. Debugging can sometimes be easy and sometimes be difficult. Debugging is often the main difficulty and time sink in program development. When software you use crashes or makes some other sort of mistake, it's because the programmer of that software made some logic errors that were not found and fixed before the program was made available for use.

The debugging cycle (the cycle for fixing logic errors) is basically the same as the cycle for fixing syntax errors. That is, you make what you think are the correct changes to your program to get it to work correctly and then you re-translate using the compiler and try to run the program again. You repeat this cycle until the program works correctly.

Beginning programmers often take a haphazard approach to the debugging cycle. They change random things in the hopes that it will work, and then try again. Don't do this. You want to reason through your errors. Look carefully at your program code (the high-level language instructions you have written). Step through each line of code while looking carefully at the output your program is producing. Try to understand why the program is producing incorrect output instead of the output you want it to produce. Once you've figured out why the program is doing what it is doing, you will then know what needs to be done to fix the program. The satisfaction achieved when you find the solution to a logic error is directly proportional to the time spent searching for the solution.

A compiler plays two roles in the development of a program. They are:

1.
2.

The program development cycle has 4 steps. They are:

1.
2.
3.
4.

A syntax error is:

A logic error is:

We say source code written in java is portable. Why?