Dave Thomas

Follow 74 Followers About

Writing Hello World in Java byte code

Dave Thomas Jun 24, 2017 · 21 min read

If you are looking for one of the more tedious ways to write a Hello World program you have come to the right place. I am going to teach you how to write, byte by byte, Hello World. We are going to do this in byte code that will run on the JVM. Yes, you heard right. Straight binary, no assembly middle man language to help you.

In doing so, I am hoping you will get a taste for what makes up a Java class file. Maybe you will learn a thing or two along the way. After all isn't that what Hello World programs are for?

When an interested soul searching developer asks the question, "How do I write Hello World in Java byte code?" They are typically directed towards Jasmine, like any sane person would recommend.

Jasmine is great, it is an assembly like language that can compile to byte code. Sadly you still don't learn the make up of byte code though by learning Jasmine. You may get familiar with the JVM instruction set, but the true structure of a binary class is still a little bit hidden from your sight.

If you are actually looking to write useful programs in byte code, once you are familiar I highly recommend Jasmine.

we can sanely use two hex characters (1-F) to represent a single byte.

Alright let's get started and take a bite of some bytes.

Class File

When you write a Java program and compile it you get a class file. That class file is Java byte code. It is a binary data file that contains instructions for the Java Virtual Machine to execute your program. If we examine the structure of a class file there actually isn't too much to it. Check out the official class file docs.

```
ClassFile {
    4 bytes         Java Magic Number
    2 bytes         Minor Version
    2 bytes         Major Version
    2 bytes         Size of the constant pool
    * bytes         Numerous bytes making up the constant pool
    2 bytes         This class' access modifiers (Ie. public)
    2 bytes         Index of this class in constant pool
    2 bytes         Index of this class' super class in constant pool
    2 bytes         Number of interfaces
    * bytes         Numerous bytes making up interface definitions
    2 bytes         Number of fields in this class
    * bytes         Numerous bytes making up field definitions
    2 bytes         Number of methods in this class
    * bytes         Numerous bytes making up method definitions
    2 bytes         Attributes count ( meta data for class file )
    * bytes         Numerous bytes making up attribute definitions
}
```

Let's dissect that Class File definition a bit.

Magic Number

The Java magic number is an interesting construct. It actually has a funny back story. Check it out if you are interested. It is four bytes that are always at the start of your file. This indicates that it is a Java class file. The four bytes are CA FE BA BE. Yup, "Cafe Babe".

> *When I am working on my laptop in a coffee shop, I like to think of myself as a cafe babe.*

for this exercise. That would be version 52.0 (major first). So for us those bytes will be: 00 00 00 34. If you are interested here is a semi useful version number chart in a stack overflow answer.

Constant Pool

The next 2 bytes, are highly important, and also a little difficult to get right on your first try. They indicate the size of the constant pool. The constant pool, will be the longest part of our program and the most important part.

In class files there actually is a lot of UTF8 character data, along with typing information for the character data. For example your main method, your class name, references to other classes. Everything your class file uses will be here, and surprisingly even for a simple Hello World program there is a bit.

Other areas of the byte code like method definitions will reference into the constant pool table via indexes. The constant pool starts at index 1 and goes until size -1.

After the 2 bytes for the constant pool size, comes the constant pool. This is of variable byte length and is dependent on the data contained with in. Each entry starts with a tag, and based on the tag we know how many bytes long that tag will be. This holds true for other areas that are variable length.

Access Modifiers

Next up after the constant pool completes we have the access modifiers. Based on a combination of the following:

```
ACC_PUBLIC     0x0001
ACC_FINAL      0x0010
ACC_SUPER      0x0020 ( Not final, can be extended )
ACC_INTERFACE  0x0200
ACC_ABSTRACT   0x0400
ACC_SYNTHETIC  0x1000 ( Not present in source code. Generated )
ACC_ANNOTATION 0x2000
ACC_ENUM       0x4000
```

Class Constant Pool References

The next 4 bytes, are the class indexes in the constant pool. 2 bytes each respectively. First is the reference to this class, and second is it's super class. All classes have a super class, even if you don't declare one in which case it is **java/lang/Object**.

The rest, Interfaces, Fields, Methods, and Attributes

I'll go over these quickly so we can start writing some binary. The next sections are all structured similar to the constant pool.

Each section, be it methods or fields etc., starts with 2 bytes indicating its size. Then a variable number of bytes defining the data. Unlike the constant pool the size bytes indicate the actual number of entries, not entries size -1.

The variable byte data is similar to the constant pool variable data, being that each entry starts with a tag that indicates how much data is to come and of what type.

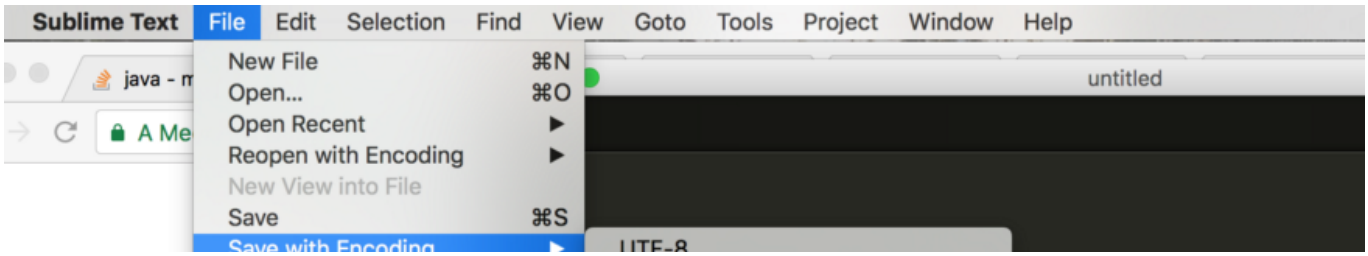
For us we only care about Methods for this program and we'll leave the others blank. To leave a section blank give it a size of zero and no other bytes. Essentially just 0000.

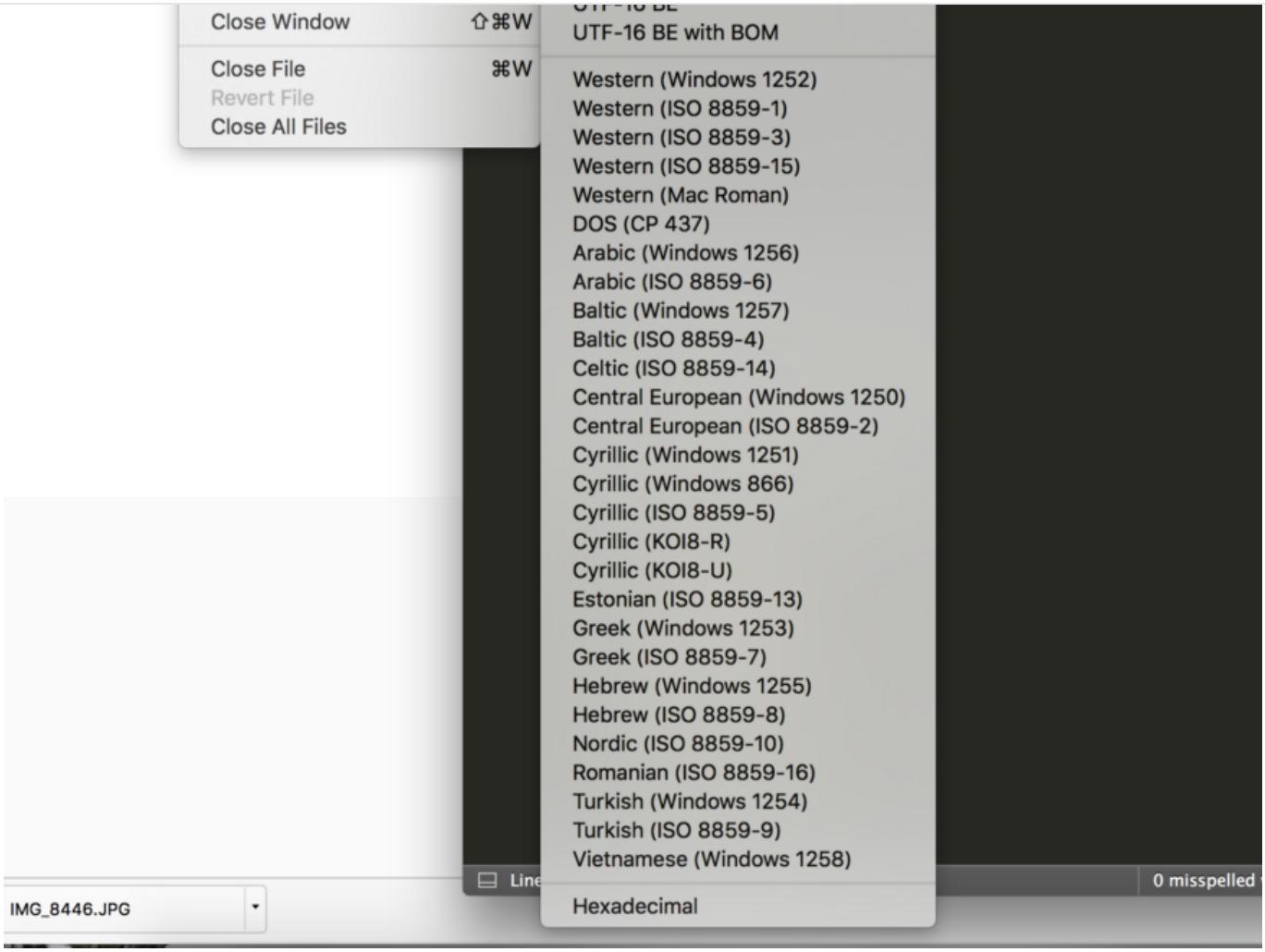
Enough Jibber Jabber, let's code already

Ok so you got an idea of what is inside a class file. Let's write a simple class file! A very crude outline of that structure. It won't run, but it will be a valid file and it we will be able to disassemble it to inspect the contents.

First let's setup some tooling. I am using Sublime Text, and I find it works quite well for writing binary files from scratch in hex. I like it because you can add white space between the Hex you are typing.

In Sublime Text you want to select save with encoding hexadecimal:





Now you can freely type Hex. If you see a dialog saying invalid character found falling back to UTF8, then you have typed a character other than 1 through f. Finding the illegal characters can be a pain as our file grows. So be careful. Note: Sublime Text considers uppercase characters invalid for hexadecimal.

Here is our first class file, type this below:

```
cafe babe 0000 0034 0000 0021 0000 0000
0000 0000 0000 0000
```

Congratulations you wrote some valid byte code. Well… semi-valid. You basically said, “I’m a Java 8 Class file that is Super Public, invalid class indexes, with 0 interfaces, 0 fields, 0 methods, and 0 attributes”

See if you can decipher the binary we typed:

- Java File: CAFE BABE
- Version 8: 0000 0034
- Constant Pool Size of ZERO: 0000
- Super Public: 0021
- Unknown index of class in constant pool: 0000
- Unknown index of super class in constant pool: 0000
- zero interfaces: 0000
- zero fields: 0000
- zero methods: 0000
- zero attributes: 0000

Save the file in hexadecimal as HelloWorld.class, and let's confirm it was done right. I'm assuming you have Java installed (specifically version 8) for this file. Note: if you have lower version installed change your file's version number. Open up your command line. We are going to use the java class dissasembler utility, **javap**. This utility is great for looking at the structure of a class file. At the command line, in the directory where you saved your file type:

```
javap HelloWorld.class
```

If you wrote the file write you'll see this:

```
Error: invalid index #0
public class ??? {
}
```

```
Error: unexpected end of file while reading HelloWorld.class
```

If you run the file via, **java HelloWorld.class**, you will see this output:

```
Error: Could not find or load main class HelloWorld.class
```

That's right, we have no main method! If your first day of Java 101 taught you anything every Java program needs a main method. But we still got a *ways* to go before we get there.

Let's look at what we got so far:

```
Error: invalid index #0
public class ??? {
}
```

What's that invalid index #0? It's talking about our class reference in the constant pool. We never put a class reference into our constant pool, and index 0 does not exist. Hence why our class is ???. We just wrote any empty unknown class.

Adding the HelloWorld Class name

Let's put our class in the constant pool and give it a name! Class entries in the constant pool require two constant pool entries. One for the UTF8 string data that is the class' name and another to indicate that it is a class.

The tag to create a class is 07. The class constant pool entry is three bytes. One for the tag and two for an index pointing to a UTF8 entry in the constant pool.

A UTF8 entry is denoted by the tag 01. The tag is followed by two bytes that indicate the size in bytes of the UTF8 string. That size is not the size of the string, but the number of

The constant pool entries to add a class called “HelloWorld” would be:

```
-- Class at index 2
07 00 02

-- UTF8 10 bytes    H  e  l  l  o  W  o  r  l  d
01 00 0a            48 65 6c 6c 6f 57 6f 72 6c 64
```

Let’s add those bytes to our file, and give our class a name (Remember: Sublime Text will let you add space between the bytes to keep things some what readable for you):

```
cafe babe 0000 0034

0003

0700 02
0100 0a  48 65 6c 6c 6f 57 6f 72 6c 64

0021 0001 0000
0000 0000 0000 0000
```

Notice that the size of our constant pool is 0003? That is because the constant pool size is always 1 bigger than it’s actual size.

Add that to your file and save it. Run **javap HelloWorld.class** again, and you should see:

```
public class HelloWorld {
}
```

Alright, now we are cooking with bytes! Look at that our class has a name! Bet you never thought writing a Hello World program could take this long. We aren’t halfway yet.

Adding the Super Class

Using a string to hex converter get the hex for this string: java/lang/Object. All references to other classes must be fully qualified, and use / instead of a period unlike in Java source code.

Update your binary program to this:

```
cafe babe 0000 0034

0005

0700 02
0100 0a  48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10  6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

0021 0001 0003
0000 0000 0000 0000
```

Make sure you understand what is written there before moving on! A constant pool size of **0005** (4 entries). As we increase our constant pool don't forget to change it's size!

4 Entries in the constant pool. 1 class definition pointing to the UTF8 value HelloWorld at index 2, and another class definition pointing to the UTF8 java/lang/Object at index 4. After the access modifiers we specify which class it is, and what the index of it's super is.

This time run:

```
javap -verbose HelloWorldP1.class
```

This will show us more information so we can confirm our constant pool:

```
Classfile /Users/davethomas/Desktop/HelloWorld.class
  Last modified 22-Jun-2017; size 62 bytes
```

```
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Class              #2             // HelloWorld
  #2 = Utf8               HelloWorld
  #3 = Class              #4             // java/lang/Object
  #4 = Utf8               java/lang/Object
{
}
```

Look at all that juicy information! You should see the same, and confirm you do so you know you are doing things right.

We got our version number of 52, our access modifiers, and our constant pool. The constant pool printed here is a great reference as you hand code binary. Four entries and their respective indexes as we specified.

Filling out the Constant Pool

We are ready to take some big strides now, and work towards our Hello World main method. To do this there are a lot of constant pool entries we need to consider.

Our goal is this Java code:

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

Here is what we need in our constant pool to do that:

- One method reference to **println**, which in turn requires two class references to **System** and **PrintStream** (the type of the **out** variable), a variable reference to **out**.
- A constant string reference to **“Hello World”**.
- UTF 8 data for our method signature, 2 entries: **main** and **([Ljava/lang/String;)V**. Those are used for the method name and return type.

Now that is a lot we are going to be adding our constant pool, so let's build it up slowly and make sure we don't make any mistakes.

First let's add the two extra class references we need. System, and PrintStream.

Change your byte code program to:

```
cafe babe 0000 0034

0009

0700 02
0100 0a  48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10  6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

0700 06
0100 10  6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d

0700 08
0100 13  6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d

0021 0001 0003
0000 0000 0000 0000
```

Run **javap -verbose HelloWorld.class** and confirm you have a constant pool with these 8 entries:

```
Constant pool:
  #1 = Class              #2             // HelloWorld
  #2 = Utf8               HelloWorld
  #3 = Class              #4             // java/lang/Object
  #4 = Utf8               java/lang/Object
  #5 = Class              #6             // java/lang/System
  #6 = Utf8               java/lang/System
  #7 = Class              #8             // java/io/PrintStream
  #8 = Utf8               java/io/PrintStream
```

Next let's add our one and only string constant we will be using in our program. **"Hello World"**

Add a constant pool entry using the tag 08 to indicate a string constant. This entry is similar to the class entries, and should be 3 bytes long. 1 for the tag and 2 for the index of the UTF8 data of string.

Change your program to:

```
cafe babe 0000 0034

000b

0700 02
0100 0a  48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10  6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

0700 06
0100 10  6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d

0700 08
0100 13  6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d

0800 0a
0100 0b  48 65 6c 6c 6f 20 57 6f 72 6c 64

0021 0001 0003
0000 0000 0000 0000
```

Run **javap -verbose HelloWorld.class** and confirm your constant pool has two new entries. These entries should represent our **"Hello World"** string constant.

```
Constant pool:
   #1 = Class              #2             // HelloWorld
   #2 = Utf8               HelloWorld
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
```

```
   #8 = Utf8               java/io/PrintStream
   #9 = String             #10            // Hello World
  #10 = Utf8               Hello World
```

Next let's add our constant reference to the **out** static variable. Update your binary program with these bytes:

```
cafe babe 0000 0034

000f

0700 02
0100 0a   48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

0700 06
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d

0700 08
0100 13   6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d

0800 0a
0100 0b   48 65 6c 6c 6f 20 57 6f 72 6c 64

0900 0500 0c
0c00 0d00 0e
0100 03   6f 75 74
0100 15   4c 6a 61 76 61 2f 69 6f 2f 50 72
          69 6e 74 53 74 72 65 61 6d 3b

0021 0001 0003
0000 0000 0000 0000
```

javap -verbose HelloWorld.class should give you this:

```
Constant pool:
   #1 = Class              #2             // HelloWorld
   #2 = Utf8               HelloWorld
   #3 = Class              #4             // java/lang/Object
```

```
   #7 = Class              #8             // java/io/PrintStream
   #8 = Utf8               java/io/PrintStream
   #9 = String             #10            // Hello World
  #10 = Utf8               Hello World
  #11 = Fieldref           #5.#12         //
java/lang/System.out:Ljava/io/PrintStream;
  #12 = NameAndType        #13:#14        //
out:Ljava/lang/PrintStream;
  #13 = Utf8               out
  #14 = Utf8               Ljava/io/PrintStream;
```

We've added four new constant pool entries, and introduced two new tag types here. Tag 09 a field reference, and tag 0c a name and type reference.

A field reference is made up of 5 bytes. The first is the tag 09. Then the next two byes are the constant pool index to the class that the field belongs to. For us that is #7, PrintStream. The last two bytes point to a name and type reference for the field. All together that entry is the value **0900 0500 0c**.

A name and type reference, is used for fields and other references. They help describe the field's name and type. The name and type reference for **out** we've added is: **0c00 0d00 0e**. That is 5 bytes, where 0c is the tag type for NameAndType. **000d** is the name, a UTF8 entry at index #13, the value **out**. **000e** is the type a UTF8 entry at index #14, the type of **Ljava/io/PrintStream;**.

Notice how we have to put a **L** in front of the type here. **Ljava/io/PrintStream;** Classes referenced as types, always start with an **L**. We also are required to put a semicolon at the end of that string, because it could be followed by another type when defining method parameter types.

If the type was a double it would just be **D**. Integer it would be **I**. Other one letters for other primitives. Knowing that the **L** should make a little more sense.

Let's move on and add our table entries for the **println** call. For these we need a method reference in the constant pool. Update your program to add these entries:

```
cafe babe 0000 0034
```

```
0100 0a   48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

0700 06
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d

0700 08
0100 13   6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d

0800 0a
0100 0b   48 65 6c 6c 6f 20 57 6f 72 6c 64

0900 0500 0c
0c00 0d00 0e
0100 03   6f 75 74
0100 15   4c 6a 61 76 61 2f 69 6f 2f 50 72
          69 6e 74 53 74 72 65 61 6d 3b

0a00 0700 10
0c00 1100 12
0100 07   70 72 69 6e 74 6c 6e
0100 15   28 4c 6a 61 76 61 2f 6c 61 6e 67
          2f 53 74 72 69 6e 67 3b 29 56

0021 0001 0003
0000 0000 0000 0000
```

javap -verbose HelloWorld.class gives us:

```
Constant pool:
   #1 = Class              #2             // HelloWorld
   #2 = Utf8               HelloWorld
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Class              #6             // java/lang/System
   #6 = Utf8               java/lang/System
   #7 = Class              #8             // java/io/PrintStream
   #8 = Utf8               java/io/PrintStream
   #9 = String             #10            // Hello World
  #10 = Utf8               Hello World
  #11 = Fieldref           #5.#12         //
java/lang/System.out:Ljava/io/PrintStream;
  #12 = NameAndType        #13:#14        //
out:Ljava/io/PrintStream;
```

```
java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = NameAndType        #17:#18        // println:
(Ljava/lang/String;)V
  #17 = Utf8               println
  #18 = Utf8               (Ljava/lang/String;)V
```

We've added a method reference at index #15 of bytes: **0a00 0700 10.** This is tag 0a which means method, 2 bytes indicating constant pool index #7 class PrintStream, and 2 bytes indicating the index of the NameAndType of this method.

The NameAndType we've added consists of 5 bytes of: **0c00 1100 12**. Which indicates that at index #17 is the name of the method, and at index #18 is the type of the method. A method type is written (Params)Return. In this case we take one String as a parameter, and return Void. **V** equals Void.

We are very close to being able to write our main method instruction set. We just need a few more UTF8 entries to do so. Two for our method, the name and type. Plus one more UTF8 entry for the special **Code** attribute. The **Code** attribute will indicate JVM machine instructions are coming.

Let's add those now, update your binary program to:

```
cafe babe 0000 0034

0016

0700 02
0100 0a   48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

0700 06
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d

0700 08
0100 13   6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d

0800 0a
0100 0b   48 65 6c 6c 6f 20 57 6f 72 6c 64
```

```
0100 15   4c 6a 61 76 61 2f 69 61 2f 50 72
          69 6e 74 53 74 72 65 61 6d 3b

0a00 0700 10
0c00 1100 12
0100 07   70 72 69 6e 74 6c 6e
0100 15   28 4c 6a 61 76 61 2f 6c 61 6e 67
          2f 53 74 72 69 6e 67 3b 29 56

0100 04   6d 61 69 6e
0100 16   28 5b 4c 6a 61 76 61 2f 6c 61 6e
          67 2f 53 74 72 69 6e 67 3b 29 56
0100 04   43 6f 64 65

0021 0001 0003
0000 0000 0000 0000
```

javap -verbose HelloWorld.class gives us:

```
Constant pool:
   #1 = Class              #2             // HelloWorld
   #2 = Utf8               HelloWorld
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Class              #6             // java/lang/System
   #6 = Utf8               java/lang/System
   #7 = Class              #8             // java/io/PrintStream
   #8 = Utf8               java/io/PrintStream
   #9 = String             #10            // Hello World
  #10 = Utf8               Hello World
  #11 = Fieldref           #5.#12         //
java/lang/System.out:Ljava/io/PrintStream;
  #12 = NameAndType        #13:#14        //
out:Ljava/io/PrintStream;
  #13 = Utf8               out
  #14 = Utf8               Ljava/io/PrintStream;
  #15 = Methodref          #7.#16         //
java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = NameAndType        #17:#18        // println:
(Ljava/lang/String;)V
  #17 = Utf8               println
  #18 = Utf8               (Ljava/lang/String;)V
  #19 = Utf8               main
  #20 = Utf8               ([Ljava/lang/String;)V
  #21 = Utf8               Code
```

If you are familiar with Java and are wondering where our constructor references are, we don't need them. We can write valid byte code for a class with just static methods, and leave out the constructors. The code will still run. Your Java compiler will not do this. Mind you, you will get runtime errors if you try to instantiate the HelloWorld class.

All in all, this will be valid runnable byte code, and our goal right now is simply, "Hello World"!

Writing the HelloWorld main method

Let's get to writing that method! Let's add an empty static method to our byte code called main.

Method byte code follows this structure:

```
method_info {
    2 bytes         Methods access flags
    2 bytes         Name of method. UTF8 index in constant pool
    2 bytes         Type of method. UTF8 index in constant pool
    2 bytes         Number of attributes
    * bytes         Variable bytes describing attribute_info structs
}

Note: the attribute we prepared for was Code. This will contain our
byte code instructions.
```

Update your program to this:

```
cafe babe 0000 0034

0016

0700 02
0100 0a   48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74
```

```
0700 08
0100 13   6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d

0800 0a
0100 0b   48 65 6c 6c 6f 20 57 6f 72 6c 64

0900 0500 0c
0c00 0d00 0e
0100 03   6f 75 74
0100 15   4c 6a 61 76 61 2f 69 6f 2f 50 72
          69 6e 74 53 74 72 65 61 6d 3b

0a00 0700 10
0c00 1100 12
0100 07   70 72 69 6e 74 6c 6e
0100 15   28 4c 6a 61 76 61 2f 6c 61 6e 67
          2f 53 74 72 69 6e 67 3b 29 56

0100 04   6d 61 69 6e
0100 16   28 5b 4c 6a 61 76 61 2f 6c 61 6e
          67 2f 53 74 72 69 6e 67 3b 29 56
0100 04   43 6f 64 65

0021 0001 0003
0000 0000

0001

0009 0013 0014
0000

0000
```

This increases our number of methods to a count of 1, and adds the method signature. The first bytes **0009** are the access modifiers. Method access modifiers are as so:

```
ACC_PUBLIC        0x0001
ACC_PRIVATE       0x0002
ACC_PROTECTED     0x0004
ACC_STATIC        0x0008
ACC_FINAL         0x0010
ACC_SYNCHRONIZED  0x0020
ACC_BRIDGE        0x0040
ACC_VARARGS       0x0080
ACC_NATIVE        0x0100
```

0009 indicates that the method we defined is **public static**. The second 2 bytes **0013** are index #19 in our constant pool. This is the name of the method; main. The next 2 bytes are **0014,** index #20 in our constant pool. **([Ljava/lang/String;)V,** the type of our method. The type says, this method takes a primitive String array as a parameter and returns Void.

Running **javap HelloWorld.class** should give you:

```
public class HelloWorld {
  public static void main(java.lang.String[]);
}
```

We have our empty method stub! So close now. Let's add our instructions for the method via a **Code** attribute.

Update your program to:

```
cafe babe 0000 0034

0016

0700 02
0100 0a   48 65 6c 6c 6f 57 6f 72 6c 64

0700 04
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

0700 06
0100 10   6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d

0700 08
0100 13   6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d

0800 0a
0100 0b   48 65 6c 6c 6f 20 57 6f 72 6c 64
```

```
0100 15   4c 6a 61 76 61 2f 69 6f 2f 50 72
          69 6e 74 53 74 72 65 61 6d 3b

0a00 0700 10
0c00 1100 12
0100 07   70 72 69 6e 74 6c 6e
0100 15   28 4c 6a 61 76 61 2f 6c 61 6e 67
          2f 53 74 72 69 6e 67 3b 29 56

0100 04   6d 61 69 6e
0100 16   28 5b 4c 6a 61 76 61 2f 6c 61 6e
          67 2f 53 74 72 69 6e 67 3b 29 56
0100 04   43 6f 64 65

0021 0001 0003
0000 0000

0001

0009 0013 0014
0001
0015
0000 0015
0002 0001
0000 0009
b200 0b
1209
b600 0f
b1
0000
0000

0000
```

javap -verbose HelloWorld.class should give you:

```
Classfile /Users/davethomas/Desktop/HelloWorld.class
  Last modified 23-Jun-2017; size 284 bytes
  MD5 checksum 4ca76db1c261ce2295f634569f751d0c
public class HelloWorld
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Class              #2             // HelloWorld
   #2 = Utf8               HelloWorld
```

```
   #6 = Utf8               java/lang/System
   #7 = Class              #8             // java/io/PrintStream
   #8 = Utf8               java/io/PrintStream
   #9 = String             #10            // Hello World
  #10 = Utf8               Hello World
  #11 = Fieldref           #5.#12         //
java/lang/System.out:Ljava/io/PrintStream;
  #12 = NameAndType        #13:#14        //
out:Ljava/io/PrintStream;
  #13 = Utf8               out
  #14 = Utf8               Ljava/io/PrintStream;
  #15 = Methodref          #7.#16         //
java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = NameAndType        #17:#18        // println:
(Ljava/lang/String;)V
  #17 = Utf8               println
  #18 = Utf8               (Ljava/lang/String;)V
  #19 = Utf8               main
  #20 = Utf8               ([Ljava/lang/String;)V
  #21 = Utf8               Code
{
  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #11                 // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #9                  // String Hello World
         5: invokevirtual #15                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
}
```

Let's examine all that byte code we added for our method:

```
0009 - public static

0013 0014 - main ([Ljava/lang/String;)

0001 - attribute size = 1
0015 - Code Attribute ( this is index #21 in our constant pool )

0000 0015 - Code Attribute size of 21 bytes.
```

```
0000 0009 - Size of code. 9 bytes

The actual machine instructions:
b200 0b - b2 = getstatic, 000b = index #11 in constant pool ( out )
1209 - 12 = ldc ( load constant ), 09 = index #19 ( Hello World )
b600 0f - b6 = invokevirtual, 000f = index #15 ( method println )
b1 - b1 = return void


0000 - Exception table of size 0
0000 - Attribute count for this attribute of 0
```

The code attribute can examined in detail here in the docs. Machine instructions can be found here in the docs.

There is lots that happened there, so take time to absorb it, and learn it. Or you can… Yup, you can…

You can run it! You guessed it.

We are done. We are finally done the Hello World program.

If you did it right, running **java HelloWorld, s**hould give you this:

```
Hello World
```

Java Bytecode Hello World JVM