Writing Hello World in Java byte code

Taken from https://medium.com/@davethomas_9528/writing-hello-world-in-java-byte-code-34f75428e0ad and modified by Luke Lambert.

- First I would suggest opening the above link, as copy/paste will be very useful here
- Don’t worry, we’ll use hexadecimal instead of binary
- When you write a Java program and compile it, the result is a **class file**
    - This class file is Java byte code – a binary data file that contains instructions for the java Virtual machine to execute your program

- Let’s examine the structure of a class file:

```
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 break down that Class File definition a bit

Java Magic Number

- It is four bytes that are always at the start of your file
    - Indicates that your file is a Java class file
- The four bytes are: CA FE BA BE or “Café Babe”

Version

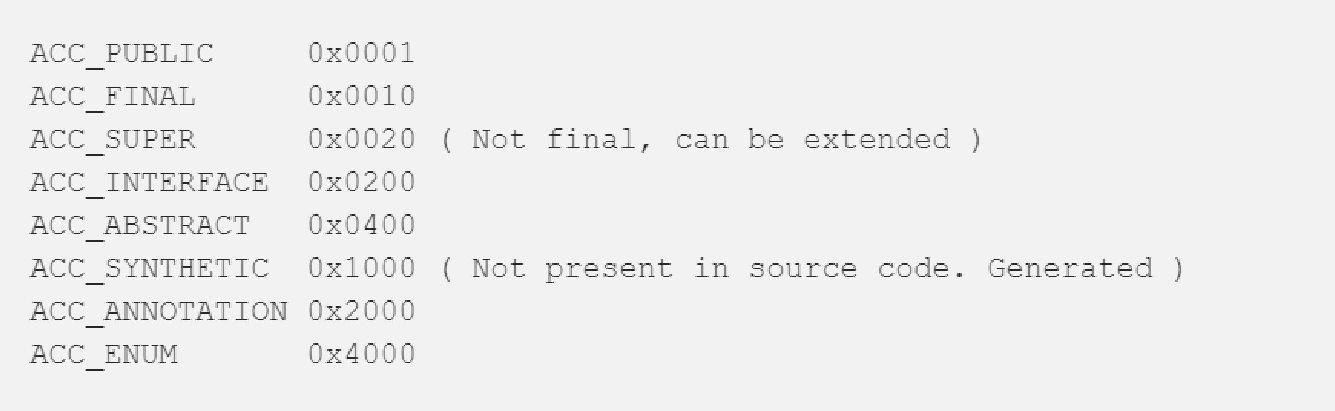
- The next 4 bytes are two 2 byte constructs that make up the version
    - Java 8 is major version 52.0 or 00 00 00 34 in hexadecimal

Constant Pool

- The next two byte are highly important; they indicate the size of the constant pool
- The constant pool is the longest and most important part of our program
- Class files contain a lot of UTF8 character data, along with typing information for the character data.
    - Ex. your main method, your class name, and 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 within
    - Each entry starts with a tag, which tells us how many bytes long that entry will be

Access Modifiers

- After the constant pool completes, we have the access modifiers. This is 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
```

- We get the access modifiers for the specific class we are defining.
    - For our Hello World program, we can just use 0021 – Super Public

Class Constant Pool References

- The next 4 bytes are the class indexes in the constant pool.
    - 2 bytes represent a reference to this class
    - 2 bytes represent a reference to its super class
        - All classes have a super class, even if you don’t declare one, in which case it is **java/lang/Object**

Interfaces, Fields, Methods, and Attributes

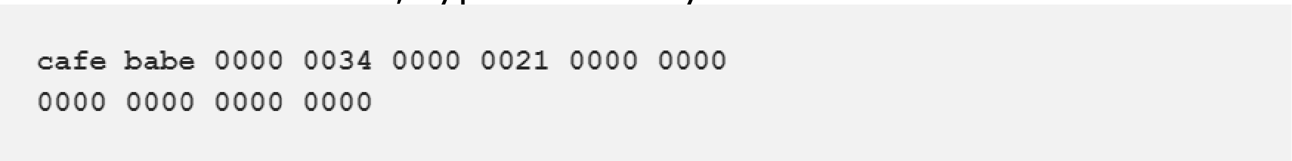
- Similar to the constant pool, each section starts with 2 bytes indicating its size, followed by a variable number of bytes defining the data
- Unlike the constant pool, the size bytes indicate the actual number of entries, not entries size – 1
- Each entry starts with a tag that indicates how much data is to come, and of what type

- For our Hello World program, we only care about Methods and we’ll leave the others blank – just 0000
- Now, we’re finally ready to code!

Okay, Let's Code!

- If you didn't already, download **Sublime Text**
    - Works quite well for writing binary files from scratch in hex
        - allows you to save a file with hexadecimal encoding
    - Improves readability by allowing you to add white space between the Hex digits you are typing

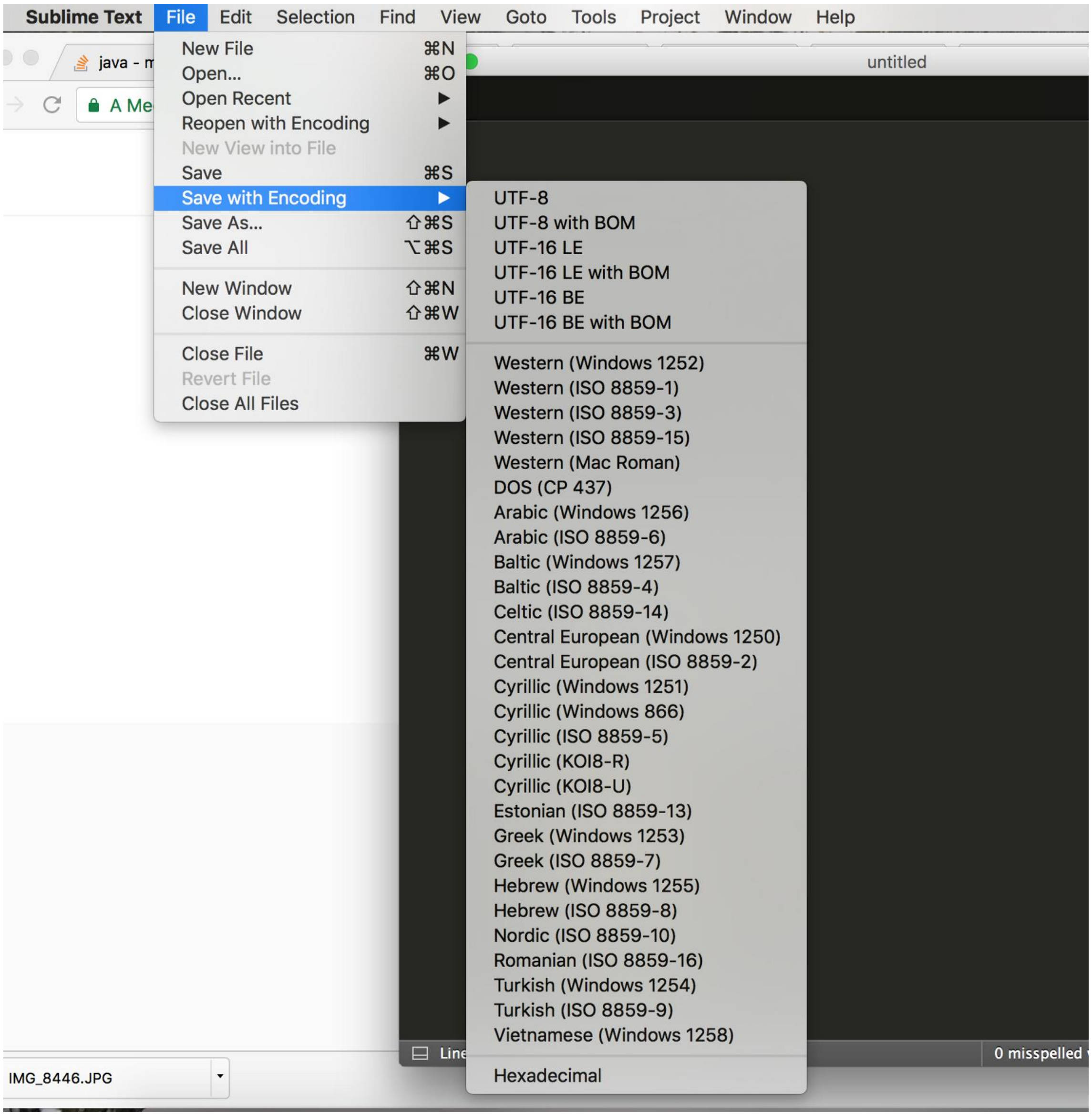
- For our first class file, type this into your editor:



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

    - This basically says it is a Java 8 Class file that is Super Public, invalid class indexes, with 0 interfaces, 0 fields, 0 methods, and 0 attributes
        - 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** encoding as HelloWorld.class



- To confirm it was done correctly, lets use the java class disassembler utility, javap
- At the command line, in the directory where you saved your file type:

```
javap HelloWorld.class
```

- If you wrote the file right, you'll see this:

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

  - This is because we never put a class reference into our constant pool, and index 0 does not exist
  - This is just an empty unknown class

- Anytime you make a mistake you will most likely see this:

```
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
```

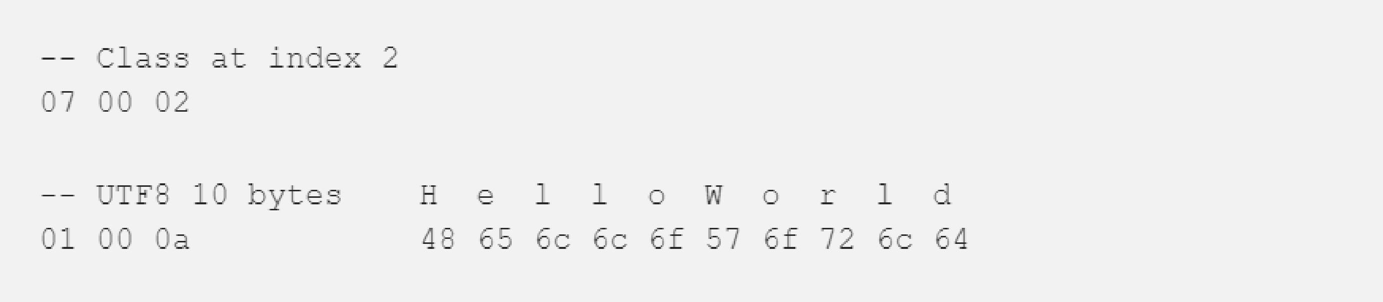
  - This error indicates that we have no main method

Adding the HelloWorld Class name

- Before we can add a main method, we must put our class in the constant pool and give it a name.
  - Class entries in the constant pool require two entries:
    - One to indicate that it is a class
    - Another for the UTF8 string data that is the class' name

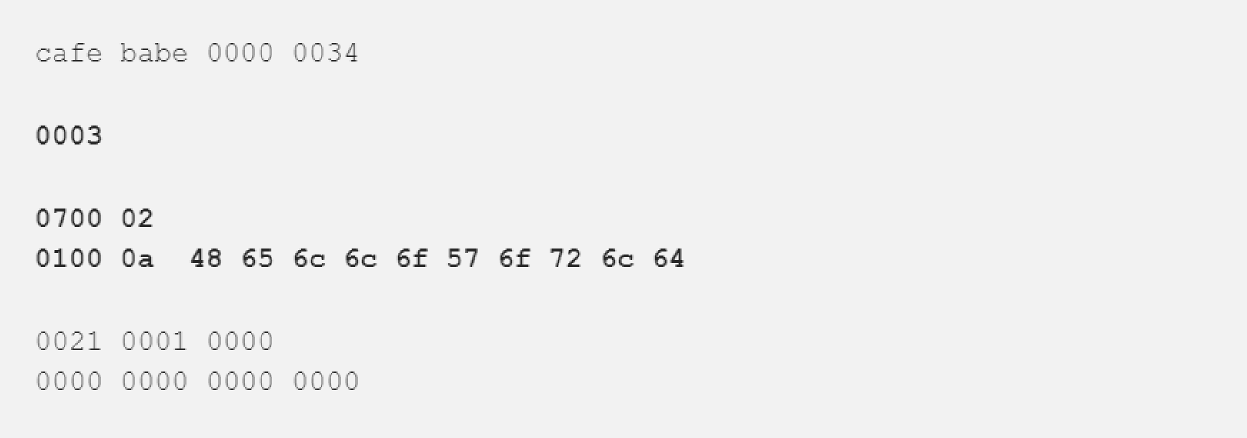
- The tag to create a class is **07**. The class constant pool entry is three bytes.
  - One byte for the tag and two bytes for an index pointing to a UTF8 entry in the constant pool.

- 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 bytes in the UTF8 string.
- 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
```

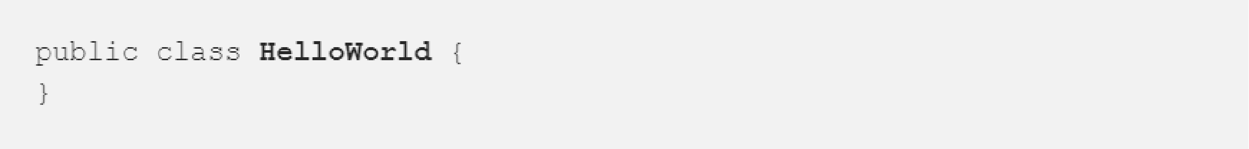
- Let’s add those bytes to our file, and give our class a name:
    - Remember, Sublime Text will let you add space between the bytes to keep things somewhat readable



```
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
```

- Note that the size of our constant pool is **0003**. That is because the constant pool size is always 1 bigger than its actual size.

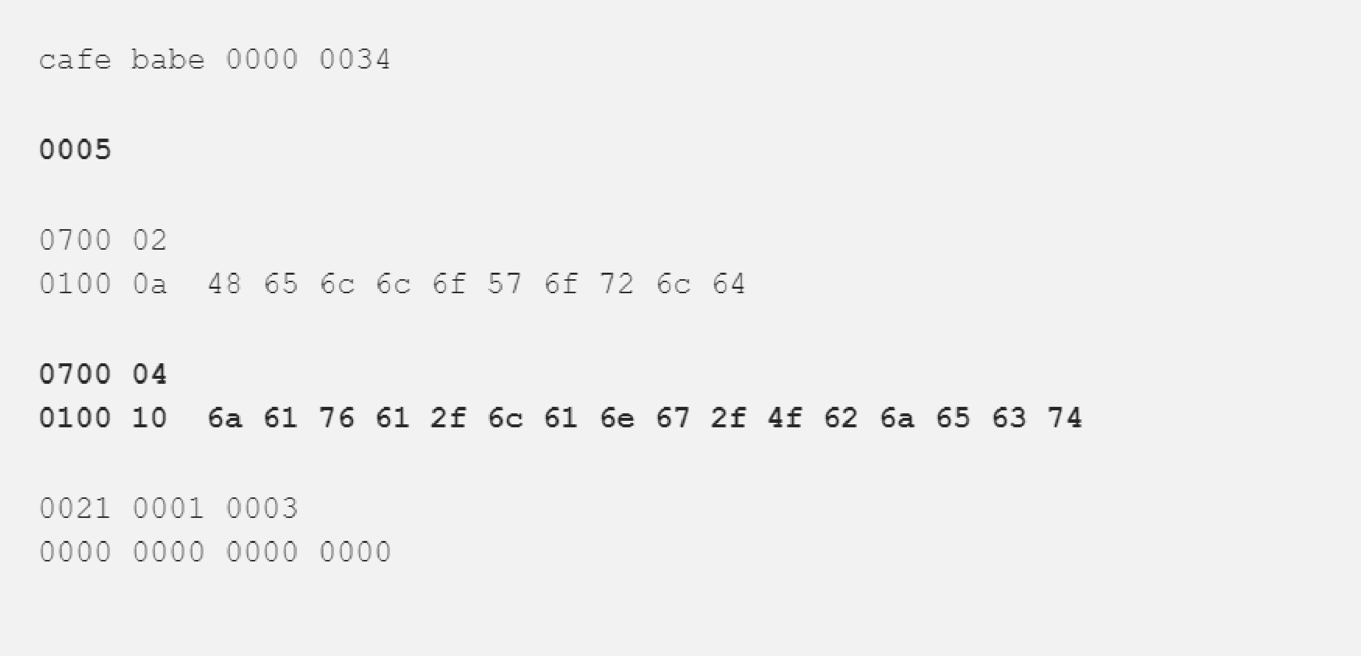
- Add that to your file and save it. Run *javap HelloWorld.class* again, and you should see:



```
public class HelloWorld {
}
```

Adding the Super Class

- Now let’s add a super class for HelloWorld, and make it **java/lang/Object**.
    - use / instead of a period, unlike in Java source code
- Update your 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
```

- Here we have four entries in the constant pool (**0005**). 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 its super is

- This time run:

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

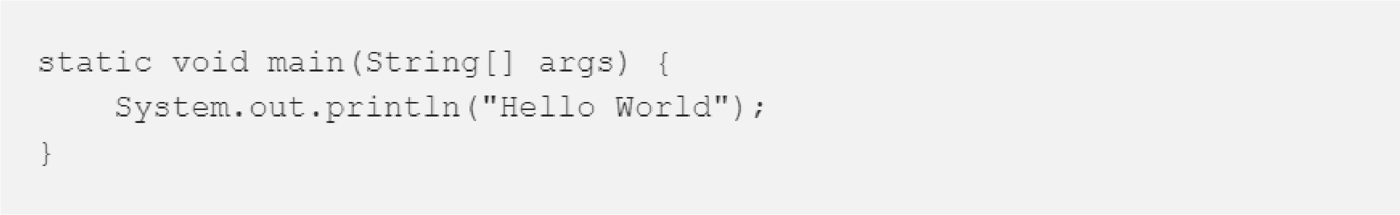
- This will show us more information so we can confirm our constant pool:

```
Classfile /Users/davethomas/Desktop/HelloWorld.class
  Last modified 22-Jun-2017; size 62 bytes
  MD5 checksum e137e3ad27a20f2419de6b0018615f0c
public class HelloWorld
  minor version: 0
  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
{
}
```

- Here we an see our version number of 52, our access modifiers, and our constant pool.
    - This is a great reference when hand coding binary!

Filling out the Constant Pool

- We are now ready to start work towards our Hello World main method!
- 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 in order to accomplish this:
    - One method reference to **println**, which in turn requires *two class references* to **System** and **PrintStream**, a variable reference to **out**

    - A constant string reference to **"Hello World".**

    - UTF8 data for our method signature, 2 entries: **main** and **([Ljava/lang/String;)V**.
        - Those are used for the method name and return type.

    - A UTF8 string representing the special attribute **Code.**
        - This will be needed to indicate the body of the main method's instructions

- First let’s add the two extra class references we need. **System** and **PrintStream**:

```
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:

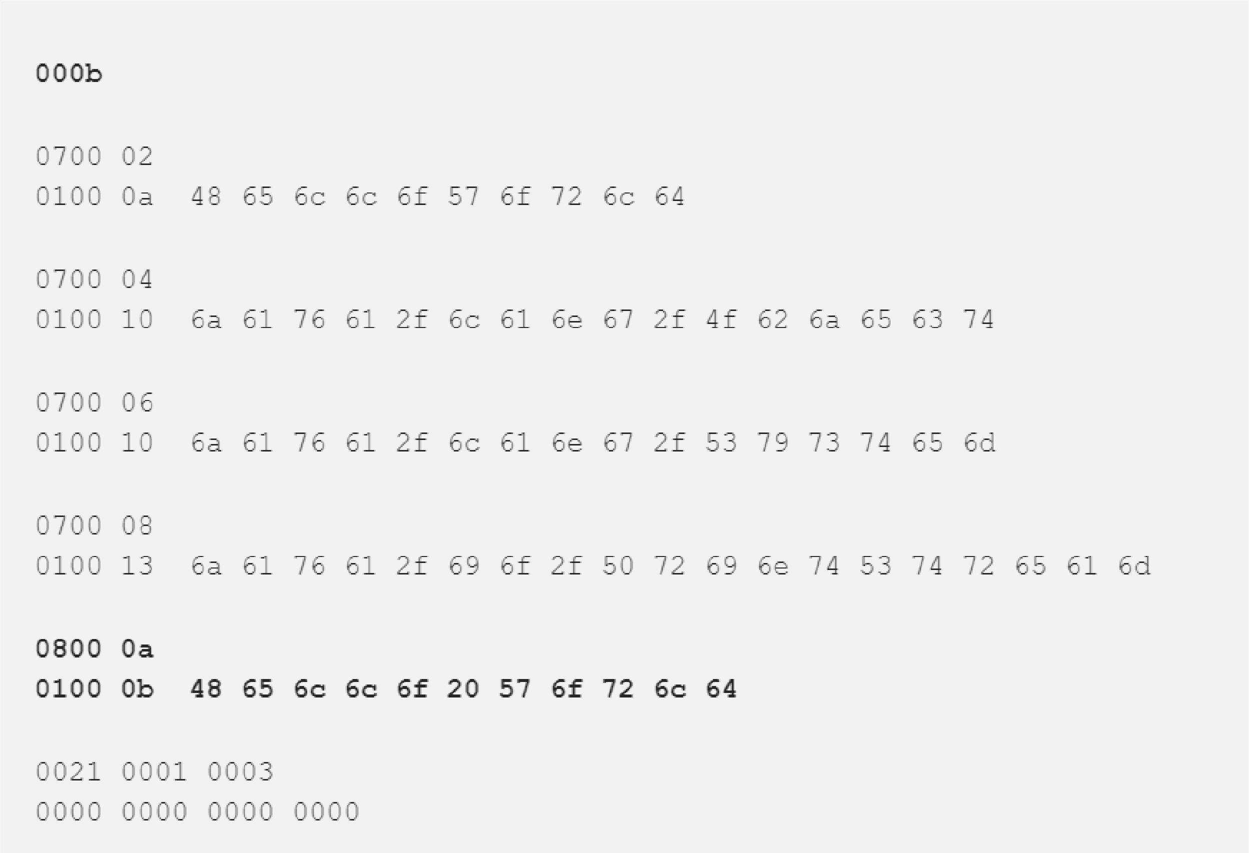
```
Constant pool:
  #1 = Class              #2              // HelloWorld
  #2 = Utf8               HelloWorld
  #3 = Class              #4              // java/lang/Object
  #4 = Utf8               java/lang/Object
  #5 = Class              #6              // java/lang/System
  #6 = Utf8          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

- Notice how the class entries have indexes pointing to the UTF8 entry that describes them. Make sure these line up!

- 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 byte for the tag
  - 2 bytes for the index of the UTF8 data of string

- Change your program to:



```
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
```

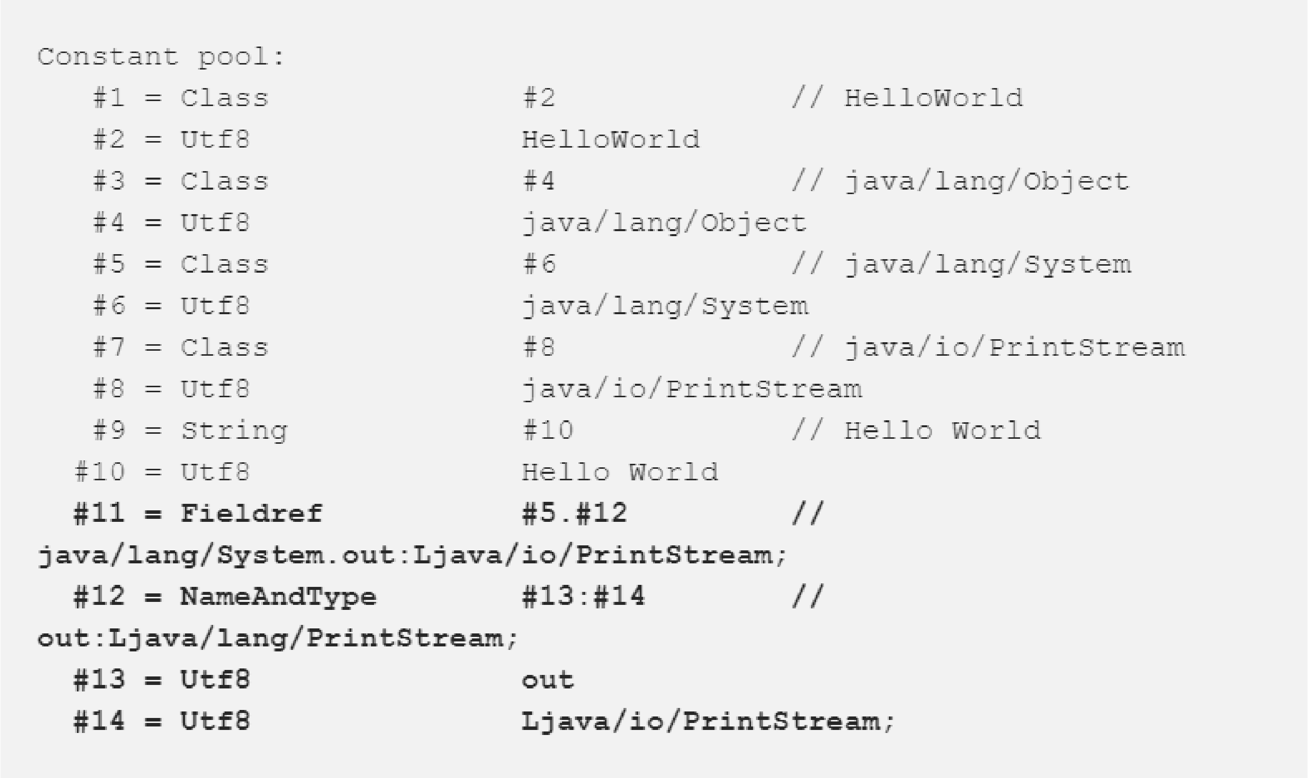
- Run **javap -verbose HelloWorld.class** and confirm your constant pool has two new entries.
  - These represent our **“Hello World”** string constant.

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

- 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
   #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/lang/PrintStream;
  #13 = Utf8               out
  #14 = Utf8               Ljava/io/PrintStream;
```

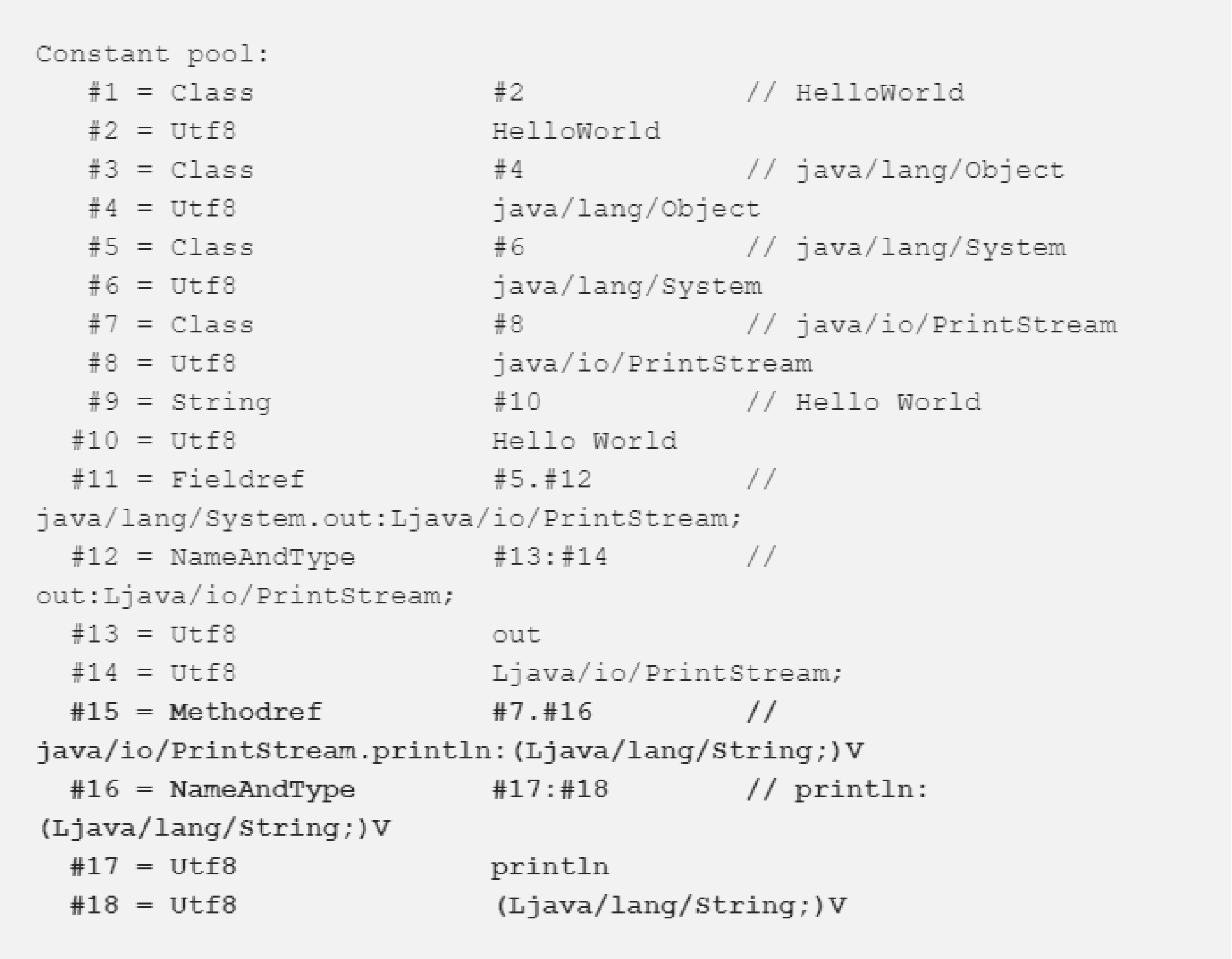
- We've added four new constant pool entries and introduced two new tag types here
  - Tag **09** is a field reference
  - Tag **0c** is a name & type reference
- A field reference is made up of 5 bytes
  - The first is the tag **09**
  - The next two bytes are the constant pool index to the class that the field belongs to
    - #7 **PrintStream** for us
  - The last two bytes point to a name and type reference for the field
  - All together the entry is the value **0900 0500 0c**

- The name & type reference for **out** we've added is: **0c00 0d00 0e**
    - **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;**
        - Note: Classes referenced as types always start with an **L.** We are also required to put a semicolon at the end of that string, because it could be followed by another type when defining method parameter types.

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

```
cafe babe 0000 0034

0013

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

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;
  #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
```

- We've added a method reference at index #15 of bytes: **0a00 0700 10**
  - The first byte is a tag, **0a**, which means method
  - The next 2 bytes indicates the constant pool index #7 class **PrintStream**
  - The final 2 bytes indicate the index of the **NameAndType** of this method

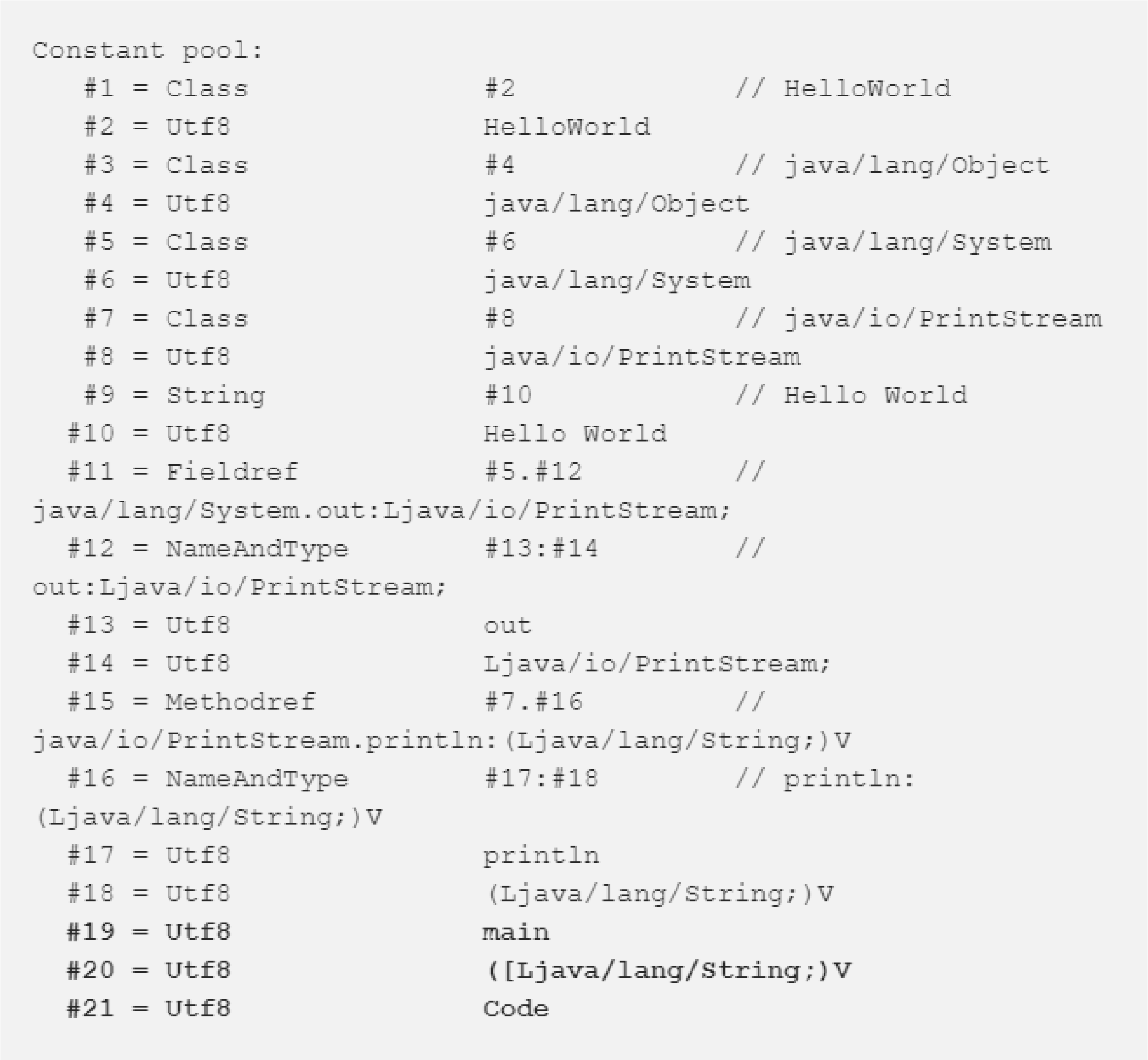
- The NameAndType we've added consists of 5 bytes: **0c00 1100 12**

    - **0c** is the tag type for NameAndType
    - **0011** indicates that the name of the method is at index #17
    - **0012** indicates that the type of the method is at index #18
        - A method type is written in the form: (Parameters)Return. In this case we take one String as a parameter, and return Void
            - **(Ljava/lang/String;)V** at index #18
- Before we finally write our main method instruction set, we need to create a few more UTF8 entries:
    - Two for our method, the name & type
    - Plus one more UTF8 entry for the special **Code** attribute
        - The **Code** attribute will indicate JVM machine instructions are coming

- Update your binary program to:

```
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

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 0000 0000
```

- **javap -verbose HelloWorld.class** gives us:



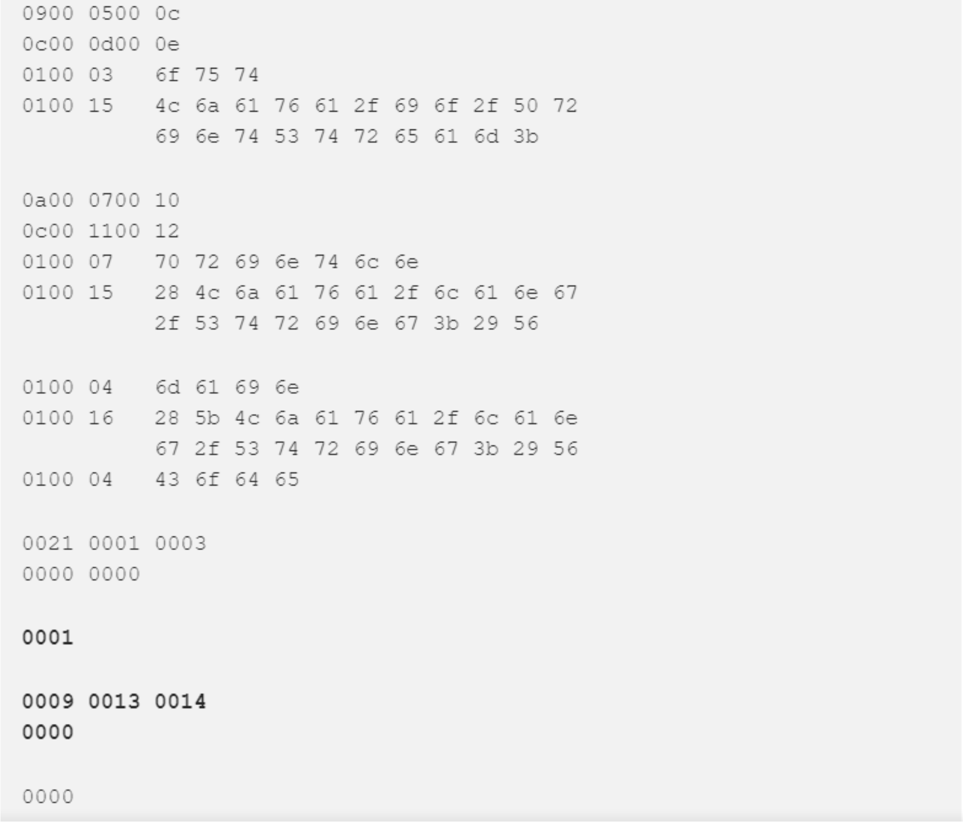
```
Constant pool:
   #1 = Class              #2            // HelloWorld
   #2 = Utf8               HelloWorld
   #3 = Class              #4            // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Class              #6            // java/lang/System
   #6 = Utf8          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

Writing the HelloWorld main method

- We are finally ready to write our main method!
- Let’s add an empty static method to our byte code called main
- Method byte code follows this structure:

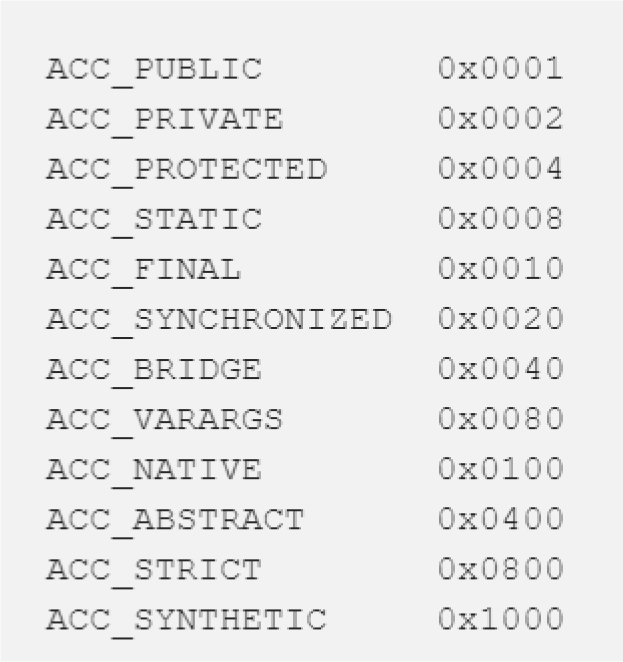
```
method_info {
    2 bytes        Methods access flags
    2 bytes        Name of method. UTF8 index in constant pool
    2 bytes        Type of method. UTF8 index in constant pool
    2 bytes        Number of attributes
    * bytes        Variable bytes describing attribute_info structs
}

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

- Update your program to:



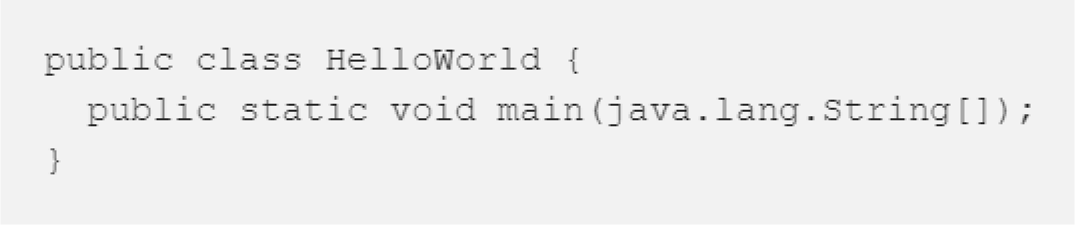
```
0900 0500 0c
0c00 0d00 0e
0100 03   6f 75 74
0100 15   4c 6a 61 76 61 2f 69 6f 2f 50 72
          69 6e 74 53 74 72 65 61 6d 3b

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 (**0001**) 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
ACC_ABSTRACT      0x0400
ACC_STRICT        0x0800
ACC_SYNTHETIC     0x1000
```

- **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 **0014** are index #20 in our constant pool.
  - **([Ljava/lang/String;)V**, the type of our method
  - They type says this method takes a primitive String array as a parameter and returns Void
- Running **javap HelloWorld.class** should give you:



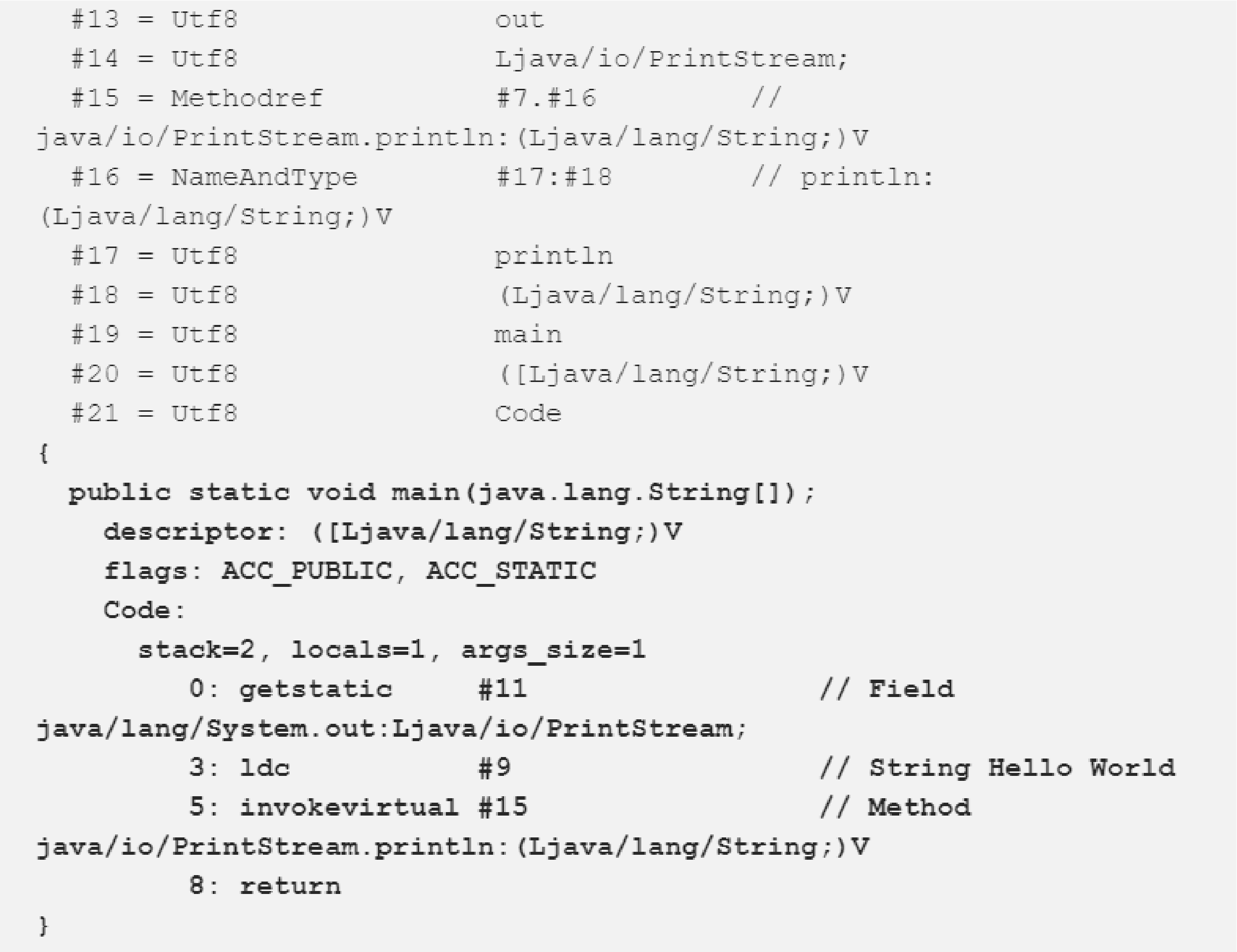
```
public class HelloWorld {
  public static void main(java.lang.String[]);
}
```

- We have our empty method stub! The last thing we need to do is to add our instructions for the method via a **Code** attribute.

- Update your program to:

```
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:



```
  #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.

21 bytes of code attribute:
0002 0001 - Max stack size of 2, and Max local var size of 1

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
```

- Now we can finally run our Hello World program with **java HelloWorld,** and it should give you this:

```
Hello World
```

No deliverable for this part!