Looking at bytecode

It sometimes helps to understand Java by looking at bytecode. These notes show how.

Background

The code you write is source code. These are class definitions, stored in text files with matching names. So if you write a class named Test, it is saved in a file named Test.java.

A Java compiler translates source code into a bytecode version, written to a corresponding .class file. So Test.java compiles to Test.class.

One or more class files might be compressed into a jar file.

The bytecode might be used by other code as a library, or run as an application by itself. In that case the Java application launcher is used to set up the JRE and the JVM, and run the bytecode on that JVM.

Example setup

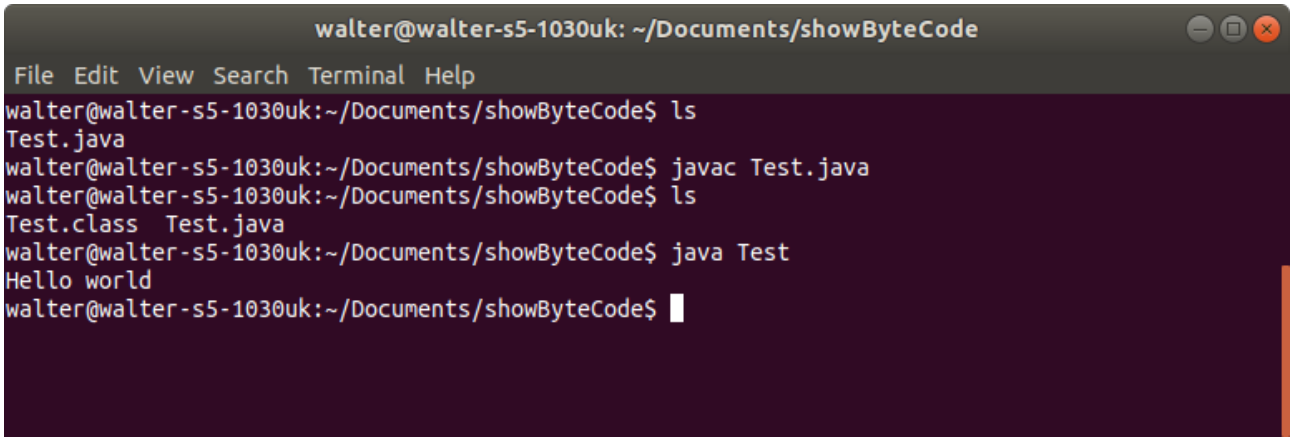
I have source code in a folder named showByteCode, in Documents, with just one class, named Test. To start with this is just:

```
public class Test {

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

}
```

This shows compiling and running it at the command line (in Ubuntu Linux):



In Linux, ls lists the current files and folders. To start with, ls just shows the source code, Test.java.

Then we say javac Test.java, to compile it. This gives us the class file, Test.class. Then we run the Java application launcher by saying java Test, and we see the output.

In MS Windows this would be the same, except dir *.* does the same as ls.

To see the bytecode, the utility javap can be run on a .class file to show the bytecode in it. Like this:

We explain this next.

Bytecode opcodes

Bytecode is made of a sequence of instructions. There is a different code for each possible instruction, which is 1 byte = 8 bits long - which is why it is called *bytecode*. Javap actual lists the *mnemonics* of the instructions, not the bytes. For example it lists return, not b1 in hex, or 1011 0001 in binary - which is what is actually in memory. The binary codes are called *opcodes*, for operation codes.

How do we know what the mnemonics are and what they do? They are listed on the web on various sites - for example

https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

The constant pool

A class file contains more information than just bytecode. The file contains a *constant pool* - a set of values which the bytecode will use. The application launcher loads the constant pool into memory, with the bytecode, when the class is loaded.

We can make javap show the constant pool with the -v option - so

javap -c -l -v Test.class

includes:

```
Constant pool:
   #1 = Methodref          #6.#15         // java/lang/Object."<init>":()V
   #2 = Fieldref           #16.#17        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #18            // Hello world
   #4 = Methodref          #19.#20        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #21            // Test
   #6 = Class              #22            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               main
  #12 = Utf8               ([Ljava/lang/String;)V
  #13 = Utf8               SourceFile
```

```
  #14 = Utf8               Test.java
  #15 = NameAndType        #7:#8          // "<init>":()V
  #16 = Class              #23            // java/lang/System
  #17 = NameAndType        #24:#25        // out:Ljava/io/PrintStream;
  #18 = Utf8               Hello world
  #19 = Class              #26 …
```

We explain this:

```
   #1 = Methodref          #6.#15         // java/lang/Object."<init>":()V
```

so constant number 1 is a reference invoking a method. The #6.#15 says which object and which method. Further down we see

```
 #6 = Class              #22            // java/lang/Object
```

and

```
  #15 = NameAndType        #7:#8          // "<init>":()V
```

so the class is Object, and the method is <init>. In bytecode, constructors are treated as special kinds of methods, with the name <init>. So the #1 is referring to the constructor of Object. We see why later.

Then we have

```
   #2 = Fieldref           #16.#17        // java/lang/System.out:Ljava/io/PrintStream;
```

so this is a reference to a field. Looking at #16 and #17, we see the class is System, and the field is out.

Then

```
   #3 = String             #18            // Hello world
```

so constant 3 is a string. But not the actual Unicode of the String. We need to look at #18, to see

```
  #18 = Utf8               Hello world
```

where the actual string “Hello world” is stored, in Unicode codepoints, using UTF8 encoding.

The stack

Bytecode is a stack-oriented language. This means it makes heavy use of a stack, as a last-in first-out data structure. We often see a code pattern where values are pushed onto the stack, then a method is called, which will take its arguments off the stack, and push the return value back.

We will see many examples of this.

Hello world bytecode

Now we can make some sense of HelloWorld bytecode. It starts off:

```
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return
```

What is this? It is the constructor of class Test. But we did not write a constructor. Remember that if you do not write a *no-args constructor*, the compiler creates one for you. This is it.

The first instruction, aload_0, puts *this* on the stack (check the online bytecode listings). Then invokespecial calls a method on an object. The constant pool #1 (see above) gives the class as Object, and the method as <init>. So this is calling the constructor of Object. Then we have the return.

Then

```
  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello world
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
```

getstatic gets a static field and pushes it on the stack. Constant pool #2 is #16#17, which is class System, and field out. So we have System.out on the stack

ldc pushes a constant on the stack. The constant is #3, which is the string “Hello World”

invokevirtual calls a method. #4 says the method is println, and this expects to find the executing object reference (System.out) and the parameter (“Hello World”) on the stack. The method pops these off the stack, and because it is a void method it does not push anything back on.

We have finished main so we just have a return.

Simple arithmetic

Suppose we have

```
public static void main(String[] args) {
  int x=2;
  int y=3;
  int z =x+y;
}
```

The bytecode for main is

```
public static void main(java.lang.String[]);
  Code:
     0: iconst_2
     1: istore_1
     2: iconst_3
     3: istore_2
     4: iload_1
     5: iload_2
     6: iadd
     7: istore_3
     8: return
```

x y and z are local variables. x is stored at slot 1, y is at 2 and z is at 3

iconst_2 gets integer constant 2 and stores it on the stack

istore_1 pops an int off the stack and stores it in slot 1 IOW x=2

iconst_3 puts 3 on the stack

istore_2 pops it into 2 IOW y=3

iload_1 push slot 1 = x on stack

iload_2 push 2 =y on stack

iadd pop 2 ints off stack, add them, and push result back

istore_3 pop value off stack (x+y) and store in slot 3 = z

The JVM has special instructions for local variable slots 0 to 4, and constant ints 0 to 5. We can make life more difficult with more variables and bigger numbers:

```
public static void main(String[] args) {
  int a,b,c,d,e,f,g;
  e=12;
  f=350;
  g=e+f;
}
```

becomes

```
public static void main(java.lang.String[]);
  Code:
     0: bipush        12
     2: istore        5
```

```
      4: sipush        350
      7: istore        6
      9: iload         5
     11: iload         6
     13: iadd
     14: istore        7
     16: return
```
bipush means byte integer push, and bipush 12 pushes the byte value 12 onto the stack. istore 5 pops this off and puts it in local variable 5, which is e. sipush is short (2 byte) integer pushed onto the stack. iload 5 and 6 puts e and f onto the stack, iadd adds them, and istore 7 pops result off onto local variable 7 which is g.

Declarations do nothing

Suppose we try

```
public static void main(String[] args) {
  int x;
  String y;
}
```
the bytecode is

```
public static void main(java.lang.String[]);
  Code:
     0: return
```
Nothing. No bytecode is generated.

We have two *declarations*. These give information to the compiler - that the type of x is int, and the type of y is String. But, by themselves, they produce no bytecode.

Constant expressions and conditionals

Suppose we try:

```
public static void main(String[] args) {
  int x,y,z;
  x=3+2;
  y=4;
  if (x>y)
    z=8;
  else z=9;

}
```
we get

```
        0: iconst_5
        1: istore_1
        2: iconst_4
        3: istore_2
        4: iload_1
        5: iload_2
        6: if_icmple     15
        9: bipush        8
       11: istore_3
       12: goto          18
       15: bipush        9
       17: istore_3
       18: return
```
x will be in slot 1, so

```
        0: iconst_5
        1: istore_1
```
does x=3+2;

The point is that the compiler has worked out 3+2=5 at compile time. The Java Language Specification requires the compiler to do this. 3+2 is a 'constant expression', and the compiler must work it out. Something like x+2 is not a constant expression and cannot be calculated at compile time.

So instructions 0 to 3 do x=5 and y=4, and steps 4 and 5 push them on the stack. Then step 6:

```
        6: if_icmple     15
```
means 'if integer compare less than or equal to' then the next step is 15

If not, we do

```
        9: bipush        8
       11: istore_3
       12: goto          18
```
so make z to be 8, then go to step 18

At step 15 we

```
       15: bipush        9
       17: istore_3
```
push byte 9 on the stack, then pop into slot 3 which is z.

Loops

```
public static void main(String[] args) {
  int y=0;
  for (int x=0; x<20; x+=3)
    y++;
}
```
becomes (comments added by me)

```
        0: iconst_0
        1: istore_1           // y=0
        2: iconst_0
        3: istore_2           // x=0
        4: iload_2            // copy x onto stack
        5: bipush        20   // push byte 20 onto stack
        7: if_icmpge     19   // compare greater or equal, so x=>20 go to step 19
       10: iinc          1, 1 // y++ - this is loop body
       13: iinc          2, 3 // x = x+3
       16: goto          4    // go back to step 4
       19: return
```
So there are no bytecode loop instructions. We just construct loops using compares and conditional branches.