1 JVM's General Architecture

Main subjects:

- Runtime Systems
- The Java RunTime Environment (JRE)
- JVM as an abstract machine
- JVM Data Types
- JVM RunTime Data Areas
  - Thread
  - Frames
  - Heap
  - Non - Heap
  - Class Data
- Method Area
- Disassembling Java Class Files (Javap)
- Constant Pool
- Loading, linking and veryfing class files
- Initialization
- JVM exit

Notes:

- JVM is a **multi-thread stack based** machine, which means that the memory model supposed by the JVM is a stack and it is **multi-thread** because it allows for parallel execution model(using things like `Threads`, `Runnables`)
- On slide 14 it is said that there is **no type information on local variables at runtime**:
  - This is only true for a `class` file of version previous than 50.0
  - New versions of the JVM attach to every method a `StackMapTable Attribute` which should map type information for local variables.
- Interestingly, each Thread has 3 areas: its **p**rogram **c**ounter, the **java stack** and the **native stack**:
  - **Native stack** contains the information about native functions(in C/C++) called. It is non-blocking the call to the native code so I dont know which message mechanism is implemented for the callback.
- Interestingly, at least for Hotspot JVM, a java `Thread` is mapped to a *native* Thread(i.e. a thread in the operating system running the jvm). So when a Java Thread has to be started, first everything needed is initialized(pc, stack ecc.), then we need the native thread to call the `run()` method on the java's `Thread` object. The `run` call is blocking, thus the OS's thread will only be released when the Java Thread returns.

2 JVM's instruction architecture

- The operand stack has 32 bit pushable words.
  - Nevertheless, instructions are of *variable* length, meaning an instruction is made of two parts, where the second is of variable length:
    - The *opcode*: one byte representing the operation
    - The arguments: one or more byte of arguments.
- The are **4** ways we can address arguments:
  - **Immediate Addressing**: the argument is embedded in the opcode: `iload_1` pushes integer 1 on the stack. Its hex repres. is: `00 00 00 1b`.
  - **Variable Addressing**: the `load` and `store` access variables from the *local variable array* by, respectively, pushing something from the array onto the stack and by popping something from the stack and inserting it in the locals area.
  - **Stack Addressing**: the arguments are taken from the top of the stack, which implies most of the time, popping it. E.g. `ireturn` pops an integer and returns it.
  - **Literal Addressing**: refers to one entry in the *constant pool*, similiar to `Variable Addressing` but for constant pool. E.g. `ldc` (load constant) pushes a constant #index from a constant pool (`String`, `int`, `float`, `Class`, `java.lang.invoke.MethodType`, `java.lang.invoke.MethodHandle`, or a `dynamically-computed constant`) onto the stack.
- Type of method invocation:
  - **invoke_virtual** (virtual is another word for dynamic dispatch):

```
1 int add12and13() {
2     return addTwo(12, 13);
3 }
```

becomes:

```
1           int add12and13();
2           descriptor: ()I
3           flags: (0x0000)
4           Code:
5              stack=3, locals=1, args_size=1
6                  0: aload_0
7                  1: bipush        12
8                  3: bipush        13
9                  5: invokevirtual #7   // Method addTwo:(II)I
10                 8: ireturn
11             LineNumberTable:
12                 line 8: 0
```

  - **invoke_special** (special means we know the class of object on which the method is called at compile time)

    No static flow analysis is done! Thus only really explicit calls are counted e.g. static methods.

```
13 class Invoke {
14     static int add12and13() {
15         return 12+13;
16     }
```

```
17
18      public static void main(String[] args) {
19          Invoke.add12and13();
20
21      }
22  }
```

becomes

```
1       public static void main(java.lang.String[]);
2             descriptor: ([Ljava/lang/String;)V
3              flags: (0x0009) ACC_PUBLIC, ACC_STATIC
4              Code:
5                  stack=1, locals=1, args_size=1
6                      0: invokestatic  #7  // add12and13:()I
7                      3: pop
8                      4: return
```