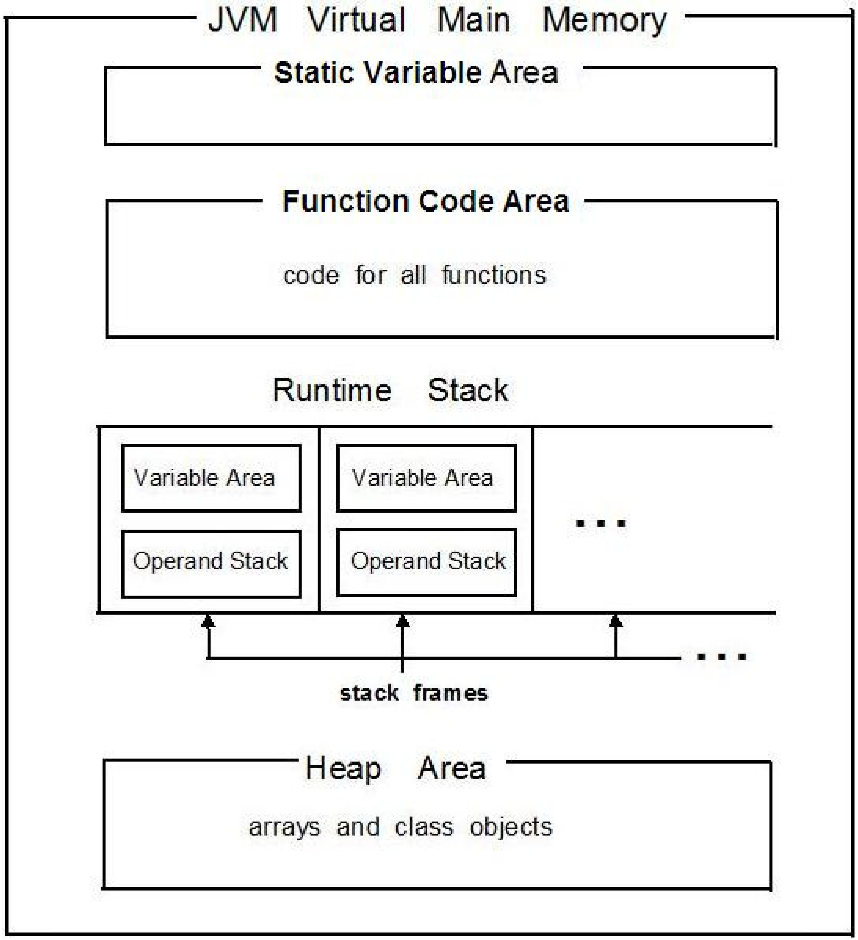
Java Virtual Machine (JVM) is a member of the family of stack-based machines that use operand stacks for expression evaluations. They have no separate, conventional registers – or we could say operand stacks serve as high-level, virtual registers. The JVM has four main components:

- The **static variable area** that contains memory cells for static variables.
- The **function code area** that contains the instructions implementing functions.
- The **runtime stack** that controls function calls. Every function call pushes a new stack frame onto the runtime stack; it is popped when the function returns. Each stack frame contains a **variable area** and an **operand stack**. The variable area contains memory cells for the function's formal parameters and local variables. The operand stack is used to evaluate expressions that appear in the function body; operand stacks are sometimes called evaluation stacks.
- The **heap area** where arrays and class objects are dynamically allocated.



The Common Intermediate Language, used for example in Microsoft .NET Framework, is also based on a similar stack-based virtual machine.

Consider this example function where S represents a statement schema.

```
void example()
{
        int i = 0;
        while ( i < 100 )
        {
                S;
                i++;
        }
}
```

The following is an example JVM code that a compiler could generate. Every source function code is compiled to start from relative address 0. In the action description, "stack" refers to the operand stack. First, let's assume S is empty.

```
0    iconst_0            push int constant 0 onto stack
```

```
1   istore_0         pop stack and store into address 0 in variable area (i=0)
2   iload_0          push value at address 0 onto stack
3   bipush  100      push int constant 100 onto stack
5   if_icmpge  14    pop top two values from stack; if stack[top-1] ≥ stack[top] then go to 14 (test if i < 100)
8   iinc  0  1       increment value at address 0 by 1 (i++)
11  goto  2
14  return
```

For a non-empty S, we simply insert JVM code for S after the `if_icmpge` instruction and make necessary increments to instruction addresses.

```
0     iconst_0         push int constant 0 onto stack
1     istore_0         pop stack and store into address 0 in variable area (i=0)
2     iload_0          push value at address 0 onto stack
3     bipush  100      push int constant 100 onto stack
5     if_icmpge  X+14  pop top two values from stack; if stack[top-1] ≥ stack[top] then go to X+14 (test if i < 100)
      ...
      code for S
      ...
X+8   iinc  0  1       increment value at address 0 by 1 (i++)
X+11  goto  2
X+14  return
```

Here X is the # of bytes occupied by the code for S.

As you might have guessed, any type of "load" instruction pushes a data item onto an operand stack, while any type of "store" instruction pops a data item from the stack and stores it at a location in the variable area.

Every JVM instruction occupies one byte, hence JVM "bytecode". An operand occupies one or more bytes depending on its instruction. Although the actual ---.class files contain JVM code in binary format, we can see their contents in text format by the JDK disassembler command javap.
The binary contents of *example.class* compiled from the following *example.java*

```
class example
{
        static int j = 0;

        static void example()
        {
                int i = 0;
                while ( i < 100 )
                {
                        j = j+i;
                        i++;
                }
        }
}
```

can be disassembled to text format by the command "javap -c example" producing

```
Compiled from "example.java"
class example extends java.lang.Object{
static int j;

example();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."":()V
   4:   return

static void example();
  Code:
   0:   iconst_0
   1:   istore_0
   2:   iload_0
   3:   bipush  100
   5:   if_icmpge       22
   8:   getstatic       #2; //Field j:I
   11:  iload_0
   12:  iadd
   13:  putstatic       #2; //Field j:I
   16:  iinc    0, 1
   19:  goto    2
   22:  return
```

```
static {};
  Code:
   0:   iconst_0
   1:   putstatic       #2; //Field j:I
   4:   return

}
```