Java VM

Terence Parr
partially derived with permission from Bill Venners' book,
"Inside the Java virtual machine"
http://www.artima.com/insidejvm/ed2/

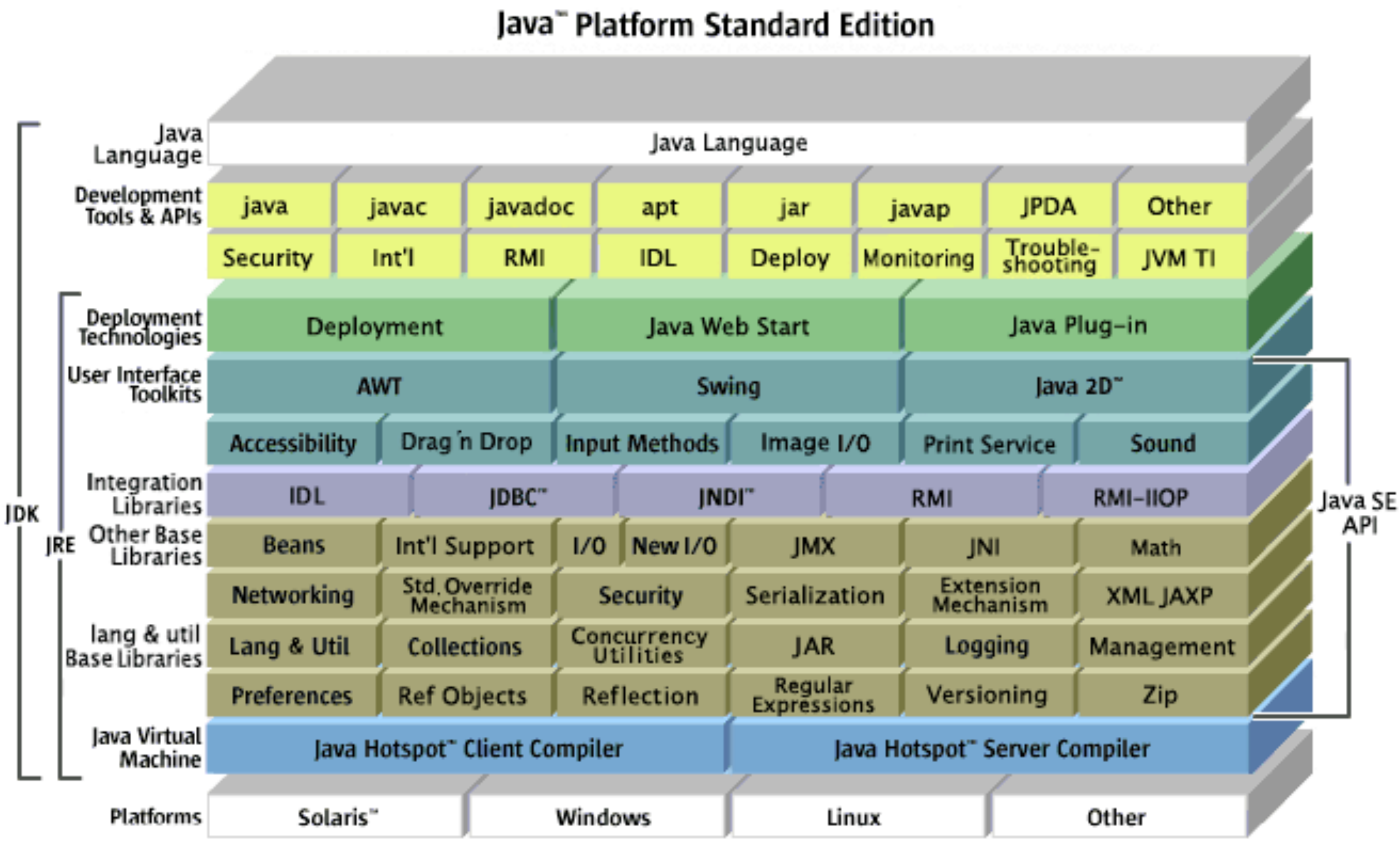
Architecture

- Java has 4 key elements:
  - language
  - .class file format
  - same API across machines
  - virtual machine

Platform

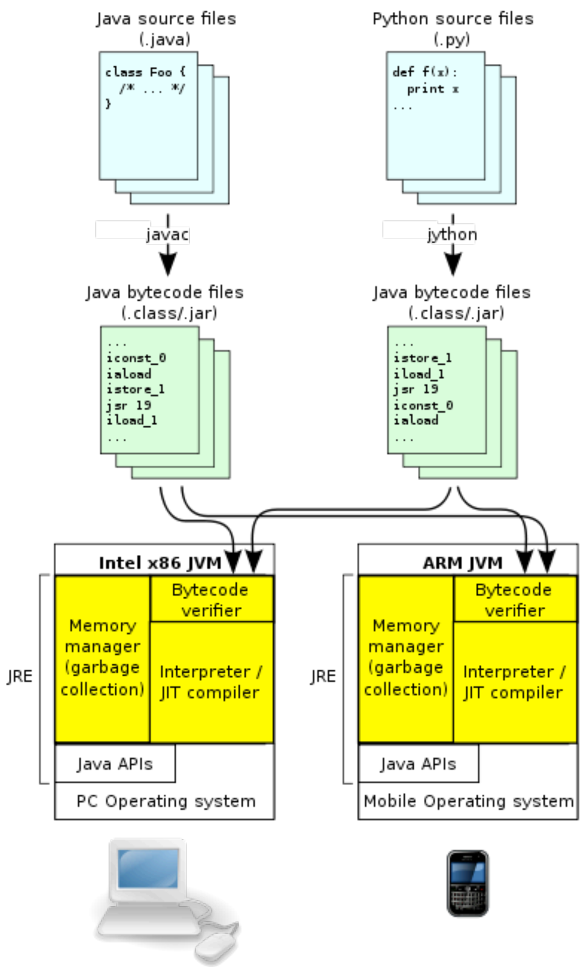
- API/VM provide cross-platform execution
- VM isolates the program from x68, PPC, ...
- API mostly in Java with calls to native libraries via JNI

Platform II



http://www.oracle.com/technetwork/java/whitepaper-135217.html

Retargeting the JVM



Can run Clojure, Scala, etc... on the Java VM

http://upload.wikimedia.org/wikipedia/commons/thumb/3/3a/Java_virtual_machine_architecture.svg/400px-Java_virtual_machine_architecture.svg.png

Program lifecycle

1. VM launches via "java MyClass"
2. ClassLoader loads java libraries and then MyClass on-demand (there is a boot loader)
3. Verification of bytecodes
4. Invoke main()
5. Create instances, GC tosses them out
6. Unload classes
7. VM exit when all user-level threads stop

Class loaders

- Finds .class files for a Class instance
- java -verbose to see where classes are loaded from
- Bootstrap loader is part of the VM (in C?)
- Bootstrap loads the core classes; can't redefine Object
- User-defined class loaders: can load classes from disk (our project) or by downloading classes from a website

Loading classes manually

```
loader = new URLClassLoader(
                  new URL[] { url1, url2 },
                  parentLoader);

final Class mainClass = loader.loadClass(className);
final Method mainMethod =
     mainClass.getDeclaredMethod(methodName);
```

Class loaders II

- Each class loader has a set of class definitions; own namespace
- Same class can be loaded into multiple loaders
- Classes are loaded on demand so linking actually occurs as the program executes, unlike C, which statically loads: cc -o myprog a.o b.o c.o
- Care must be taken that CLASSPATH is the same on development and deployment computers
- Classes unloaded when the class loader is a candidate for GC

Verification

- JVM does not assume valid/safe compiler
- Very specific verification rules that sometimes disallow valid Java programs; e.g., when incident edges flow graph state have different stack sizes, verification fails even if it would be okay.
- Verifier verifies a lot of things including:
  - the stack does not grow forever (verifies size)
  - branches to valid locations (within same a method)
  - data is always initialized, type safe; cannot convert an integer to address
  - access modifiers such as private are respected (dynamic)
- Tries to prevent crashes, access to trusted code from on trusted applications

see also: http://www.artima.com/insidejvm/ed2/lifetype3.html

Class init

- Init direct superclass if hasn't been done
- Execute any static code blocks
- Initializing interface means executing static code block
- The initializer's poor class also initialize static fields to default values or specified values
- The initializer also creates static array objects with specified initialization values

Instance instantiation

- via: new, reflection, clone(), or deserializing object
- creates space on the heap for the instance variables:
  - this class and the superclass' fields
  - the bookkeeping variables (class ptr, lock)
- call <init>() to initialize object (if via "new" op)
- init:
  - call super.init
  - initialize instance variables
  - bytecode for constructor specifies in java

Runtime verification

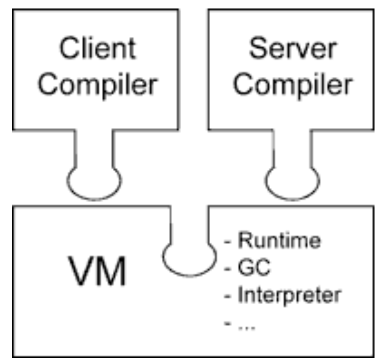
- Check type casts
- No pointer arithmetic
- GC; no free(); no still pointers
- Array bounds checking
- Null dereference checks

Runtime security

java.lang.SecurityManager

- Remote classes in Applets execute within a restricted sandbox
- JVM tries to prevent malicious code from executing on the client
- Applets cannot
  - read/write to the local file system
  - make network connection to any host, except the host from which the applet came
  - create new process
  - load a new dynamic library and directly call native method
- Customizable

JIT, HotSpot



- JVM initially interprets bytecodes
- At a threshold, dynamic compiler translates to native code
- Key principle: vast majority of time spent in a minority of code
- Can re-optimize later if it's truly a hotspot
- Debugging complicates this; don't wanted to run in interpret only mode
- Some JVM's immediately compile code instead of initially interpreting (JIT)

JIT vs Dynamic compilation

- JIT not as good as dynamic compilation: consider that variables at static and JIT time are constants at runtime
- Why waste time compiling code that runs once or never?
- Compiling all code is an unnecessary waste of memory.
- By observing the program, we can gain useful profiling information such as:
  - how many times loops go around
  - caller/callee for in-lining (inlining reduces method call overhead and creates bigger blocks of code for the optimizer to chew on)
- Might have to de-optimize then reoptimize when new classes are loaded due to new methods coming available

Client compiler

- 3 phases:
  - by code to high-level intermediate representation (HIR) in static single assignment (SSA) form to enable more optimizations
  - platform specific "back-end" generates a lower level IR (LIR)
  - LIR to machine code with peephole optimization
- Focuses on local code quality, not global optimization (expensive)

Server compiler

- High-end fully optimizing compiler (uses SSA) including:
  - dead code elimination, range-check elimination
  - loop invariant hoisting, loop unrolling
  - global code motion (move code from main path to more control dependent patterns)
  - common subexpression elimination
  - constant propagation
  - global value numbering (tag var/expr w/same value with same number)
- register allocator is a global graph coloring allocator
- highly portable, relying on a machine description file to describe all aspects of the target hardware

Class file structure

http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

```
ClassFile {
    u4             magic; // 0xCAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class; // index into constant pool
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count]; // filename, etc...
}
```

Sample bytecodes

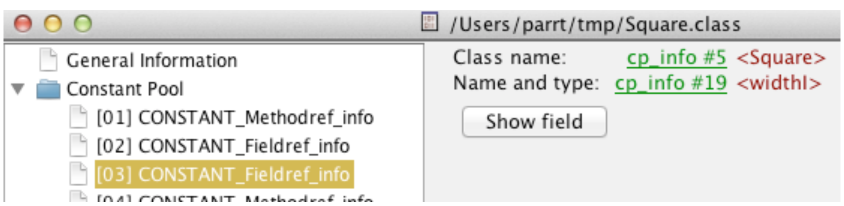
```
public class Shape {
}
class Square extends Shape {
    int width;
    public void print() {
        System.out.println(width);
    }
}
```

```
public void print();
  LineNumberTable:
   line 6: 0
   line 7: 10
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   aload_0
   4:   getfield    #3; //Field width:I
   7:   invokevirtual   #4; //Method java/io/PrintStream.println:(I)V
   10:  return
  LineNumberTable:
   line 6: 0
   line 7: 10
```

Constant pool

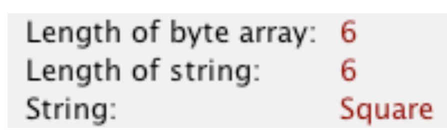
```
0:   getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;
3:   aload_0
4:   getfield    #3; //Field width:I
7:   invokevirtual   #4; //Method java/io/PrintStream.println:(I)V
```

Const pool #3 is width field:

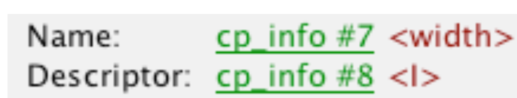


cp_info #5 is Class name:

#22 is Utf8:



#19 is NameAndType:



Method info

Name:	cp_info #13	<print>
Descriptor:	cp_info #10	<()V>
Access flags:	0x0001	[public]

Generic info:

Attribute name index:	cp_info #12
Attribute length:	10

Specific info:

Nr.	start_pc	line_number
0	0	6
1	10	7

jasmin assembler

- http://jasmin.sourceforge.net/
- Assembles bytecodes (text) to .class files

```
.method public static main([Ljava/lang/String;)V
   .limit stack 2   ; System.out and "hi mom" on stack
   .limit locals 1   ; the arg

   getstatic       java/lang/System/out Ljava/io/PrintStream;
   ldc             "Hi mom"
   invokevirtual   java/io/PrintStream/println(Ljava/lang/String;)V
   return
.end method
```

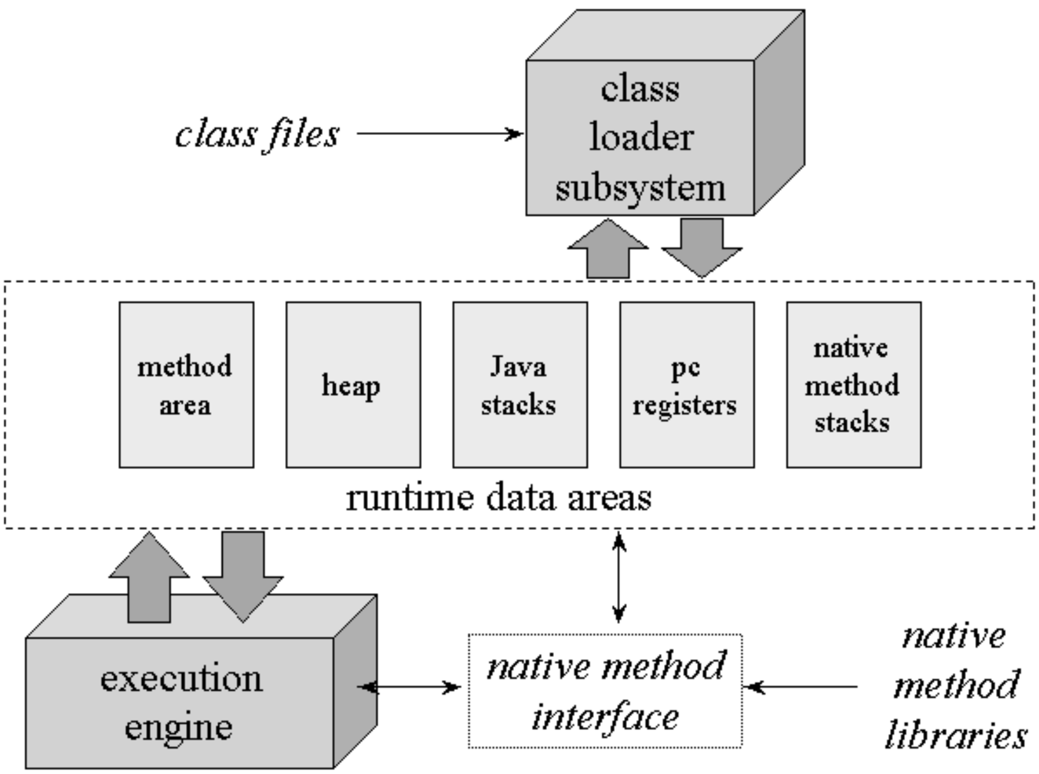
ASM

- http://asm.ow2.org/
- Visitor-based bytecode library
- We'll use this for a bytecode project

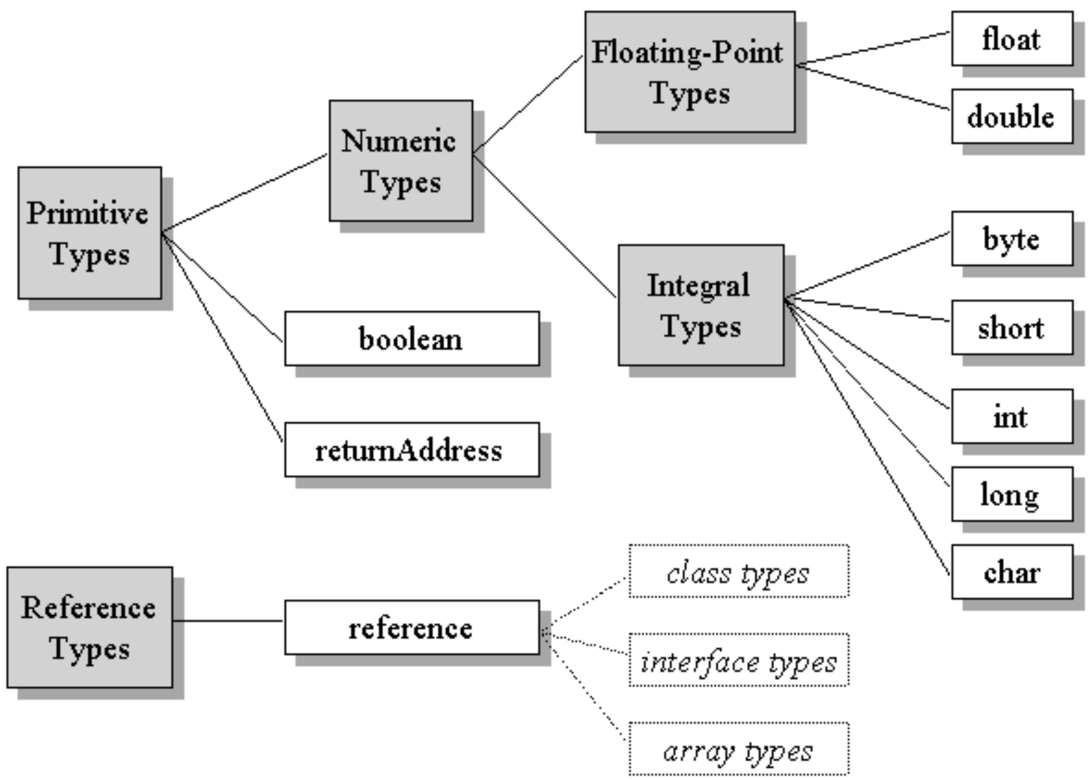
```
ClassReader cr = new ClassReader(is);
ClassWriter cw = new ClassWriter(0);
ClassVisitor cv = new MyVisitor(cw);
cr.accept(cv, 0); // visit
byte[] b = cw.toByteArray();
FileOutputStream fos = new FileOutputStream(...);
fos.write(b);
fos.close();
```

JVM execution engine

Architecture



Data types

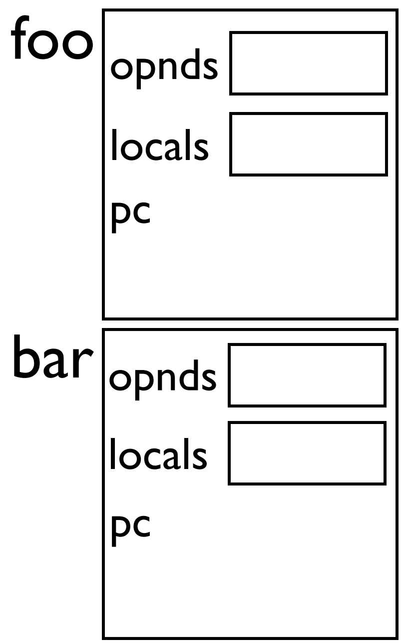

Floating-Point Types
float
double
Numeric Types
Primitive Types
byte
Integral Types
short
boolean
int
returnAddress
long
char
class types
Reference Types
reference
interface types
array types

VM elements

- Java stack (call stack) of method invocation frames
- A stack frame state of one method invocation; has:
  - local variables (size determined statically), parameters become local variables during method invocation
  - operand stack (32 bits, max size determined statically)
  - return values, return address, parent frame
  - other runtime information needed to perform dynamic linking such as pointer to the constant pool
- Current frame/method pointer
- Each thread gets its own java stack, pc, native code stack

Example method call

```
void foo() {
    int x = bar(9,10);
}
int bar(int x, int y) {
    return x+y;
}
```



Heap

- All non-primitive types are allocated in the heap (via new)
- Primitive types can be allocated on the stack like C (so could objects in some cases)
- The heap is managed by the garbage collector
- No way to explicitly free something
- Objects live as long as there is a path to it from outside the heap

Instruction set

- One byte opcodes == Bytecodes
- 0 or more operands after bytecode (or encoded within the byte code)
- Some operands are on the operand stack
- Typed by first letter:
  iload, lload, fload, dload, aload
- Compiler must encode byte, short sometimes (sign extend to int), boolean/char 0-extend
  - Only push,aload,astore,i2b,i2s take byte/short

http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

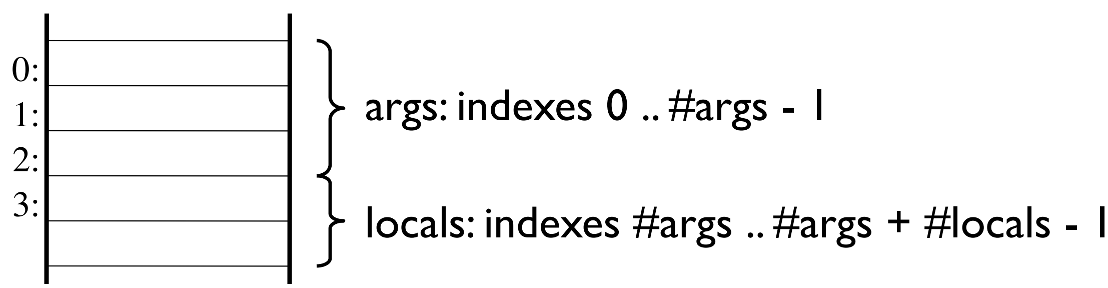
Load and store

iload forms:

```
iload_0
iload_1
iload_2
iload_3
iload n
wide iload n
```

- Load local variable onto stack: iload, iload_n, aload, ...
- Store from operand stack into local: istore, istore_n, astore, ...
- *wide* modifier gives access to a wider index

Arguments, locals inside stack frame



http://cs.ua.edu/434/Classes/20_JVM.ppt

Arithmetic instructions

Arithmetic

add: `iadd, ladd, fadd, dadd`

subtract: `isub, lsub, fsub, dsub`

multiply: `imul, lmul, fmul, dmul`

etc.

Conversion

```
i2l, i2f, i2d,
l2f, l2d, f2d,
f2i, d2i, …
```

http://cs.ua.edu/434/Classes/20_JVM.ppt

More...

Operand stack manipulation

`pop, pop2, dup, dup2, swap, ...`

Control transfer

Unconditional: `goto, jsr, ret, ...`

Conditional: `ifeq, iflt, ifgt, if_icmpeq, ...`

http://cs.ua.edu/434/Classes/20_JVM.ppt

Example loop

for (int i = 0; i < 100; i++) { }

```
0   iconst_0        // Push int constant 0
1   istore_1        // Store into local variable 1 (i=0)
2   goto 8          // First time through don't increment
5   iinc 1 1        // Increment local variable 1 by 1 (i++)
8   iload_1         // Push local variable 1 (i)
9   bipush 100      // Push int constant 100
11  if_icmplt 5     // Compare and loop if i<100
14  return          // Return void when done
```

http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

Example numeric value in constant pool

```
int i = 100;
int j = 1000000;
long l1 = 1;
long l2 = 0xffffffff;
double d = 2.2;
```

```
0   bipush 100
2   istore_1
3   ldc #1
5   istore_2
6   lconst_1
7   lstore_3
8   ldc2_w #6
11  lstore 5
13  ldc2_w #8
16  dstore 7
```

http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

Non-local memory access

accessing locals and arguments: `load` and `store` instructions

accessing fields in objects: `getfield, putfield`

accessing static fields: `getstatic, putstatic`

Note: Static fields are a lot like global variables. They are allocated in the "method area" where also code for methods and representations for classes (including method tables) are stored.

Note: `getfield` and `putfield` access memory in the heap.

http://cs.ua.edu/434/Classes/20_JVM.ppt

Heap Memory Allocation

Create new class instance (object):

`new`

Create new array:

`newarray`**:** for creating arrays of primitive types.

`anewarray, multianewarray`**:** for arrays of reference types.

http://cs.ua.edu/434/Classes/20_JVM.ppt

Calling methods

Method invocation:

`invokevirtual`**:** usual instruction for calling a method on an object.

`invokeinterface`**:** same as `invokevirtual`, but used when the called method is declared in an interface (requires a different kind of method lookup)

`invokespecial`**:** for calling things such as constructors, which are not dynamically dispatched (this instruction is also known as `invokenonvirtual`).

`invokestatic`**:** for calling methods that have the "static" modifier (these methods are sent to a class, not to an object).

Returning from methods:

`return, ireturn, lreturn, areturn, freturn, …`

http://cs.ua.edu/434/Classes/20_JVM.ppt