301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

AP-2018-06*: The JVM instruction set*

Outline

- The JVM instruction set architecture
  - Execution model
  - Instruction format & Addressing modes
  - Types and non-orthogonality of instructions
  - Classes of instructions

  ➔ Chapter 2 of JVM Specification
- Bytecode generated by simple statements

http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html

The JVM interpreter loop

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
}   while (there is more to do);
```

Instruction set properties

- 32 bit stack machine
- Variable length instruction set
  - One-byte opcode followed by arguments
- Simple to very complex instructions
- Symbolic references
- Only relative branches
- Byte aligned (except for operands of `tableswitch` and `lookupswitch`)
- Compactness vs. performance

JVM Instruction Set

- Load and store (operand stack <-> local vars)
- Arithmetic
- Type conversion
- Object creation and manipulation
- Operand stack manipulation
- Control transfer
- Method invocation and return
- Monitor entry/exit

Instruction format

- Each instruction may have different "forms" supporting different kinds of operands.
- **Example:** different forms of "iload" (i.e. push)

Assembly code	Binary instruction code layout			
`iload_0`	26			Pushes local variable 0 on operand stack
`iload_1`	27			
`iload_2`	28			
`iload_3`	29			
`iload` *n*	21	*n*		
`wide iload` *n*	196	21	*n*	

Runtime memory

- Memory:
  - Local variable array (frame)
  - Operand stack (frame)
  - Object fields (heap)
  - Static fields (method area)
- JVM stack instructions
  - implicitly take arguments from the top of the operand stack of the current frame
  - put their result on the top of the operand stack
- The operand stack is used to
  - pass arguments to methods
  - return a result from a method
  - store intermediate results while evaluating expressions
  - store local variables

JVM Addressing Modes

- JVM supports three addressing modes
  - Immediate addressing mode
    - Constant is part of instruction
  - Indexed addressing mode
    - Accessing variables from **local variable array**
  - Stack addressing mode
    - Retrieving values from **operand stack** using pop

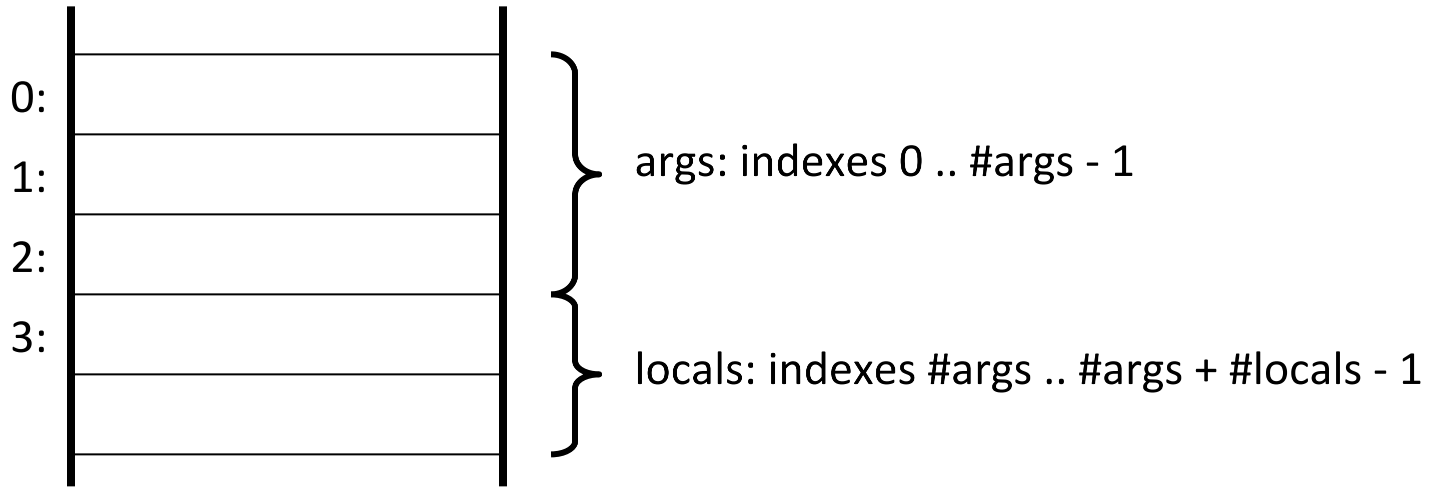
Instruction-set: typed instructions

- JVM instructions are explicitly typed: different opCodes for instructions for integers, floats, arrays, reference types, etc.
- This is reflected by a naming convention in the first letter of the opCode mnemonics
- **Example:** different types of "load" instructions

`i`	`int`
`l`	`long`
`s`	`short`
`b`	`byte`
`c`	`char`
`f`	`float`
`d`	`double`
`a`	`for reference`

`iload`	integer load
`lload`	long load
`fload`	float load
`dload`	double load
`aload`	reference-type load

Instruction-set: accessing arguments and locals in the Local Variable array


0:
1:
2:
3:
args: indexes 0 .. #args - 1
locals: indexes #args .. #args + #locals - 1

Instruction examples:

iload_1	istore_1
iload_3	astore_1
aload 5	fstore_3
aload_0	

- A *load* instruction takes something from the args/locals area and pushes it onto the top of the operand stack.
- A *store* instruction pops something from the top of the operand stack and places it in the args/locals area.

Opcode “pressure” and non-orthogonality

- Since op-codes are bytes, there are at most 256 distinct ones
- Impossible to have for each instruction one opcode *per type*
- Careful selection of which types to support for each instruction
- Non-supported types have to be converted
- Result: **non-orthogonality** of the Instruction Set Architecture

Type support in the JVM instruction set

- Design choice: almost no support for `byte`, `char` and `short` – using `int` as “computational type”

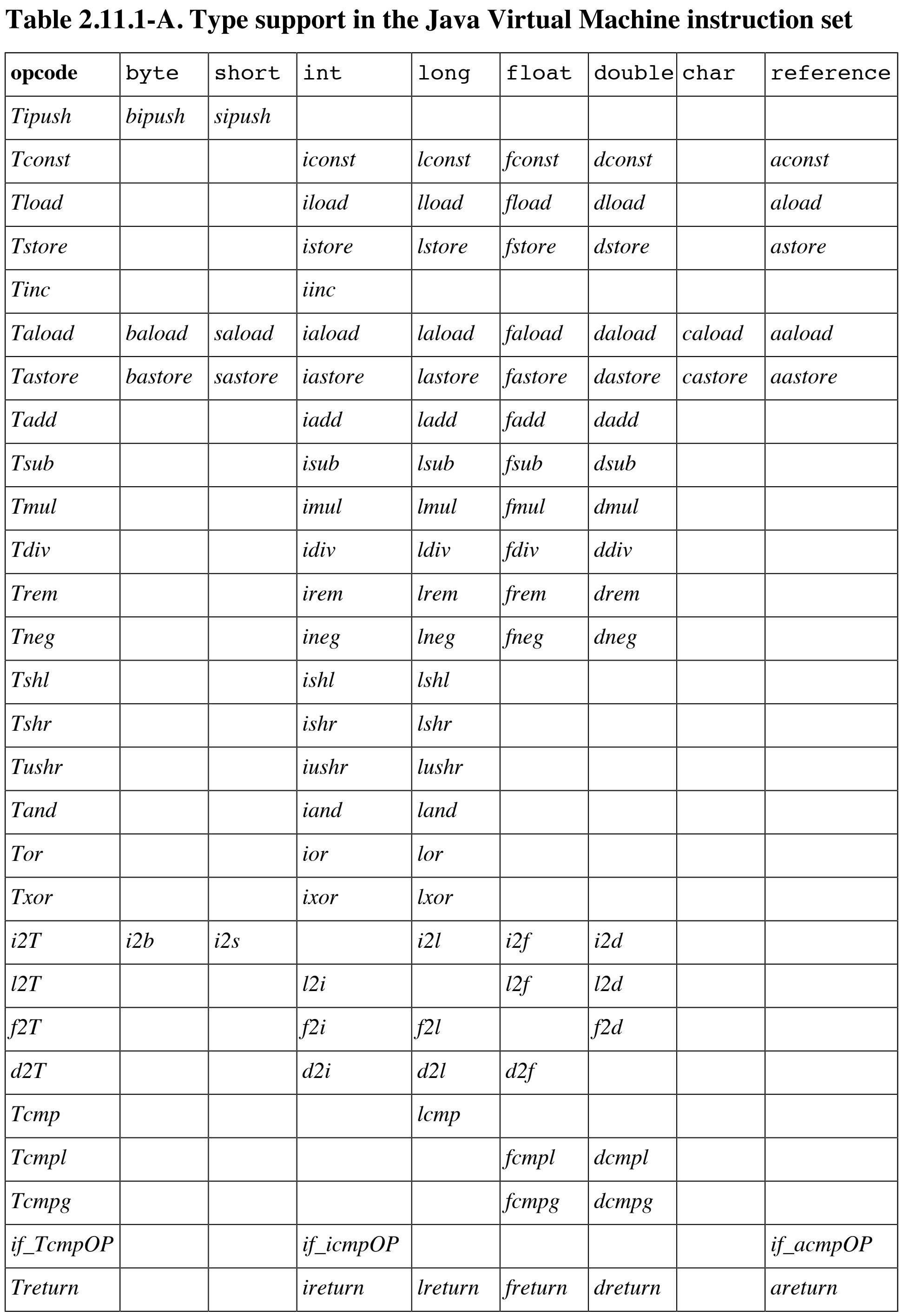


Table 2.11.1-A. Type support in the Java Virtual Machine instruction set

opcode	`byte`	`short`	`int`	`long`	`float`	`double`	`char`	`reference`
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			*iushr*	*lushr*				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

Computational Types

Table 2.11.1-B. Actual and Computational types in the Java Virtual Machine

Actual type	**Computational type**	**Category**
`boolean`	`int`	1
`byte`	`int`	1
`char`	`int`	1
`short`	`int`	1
`int`	`int`	1
`float`	`float`	1
`reference`	`reference`	1
`returnAddress`	`returnAddress`	1
`long`	`long`	2
`double`	`double`	2

Instruction-set: non-local memory access

- In the JVM, the contents of different "kinds" of memory can be accessed by different kinds of instructions.
- **accessing locals and arguments:** `load` and `store` instructions, as seen
- **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.

Instruction-set: operations on numbers

Arithmetic

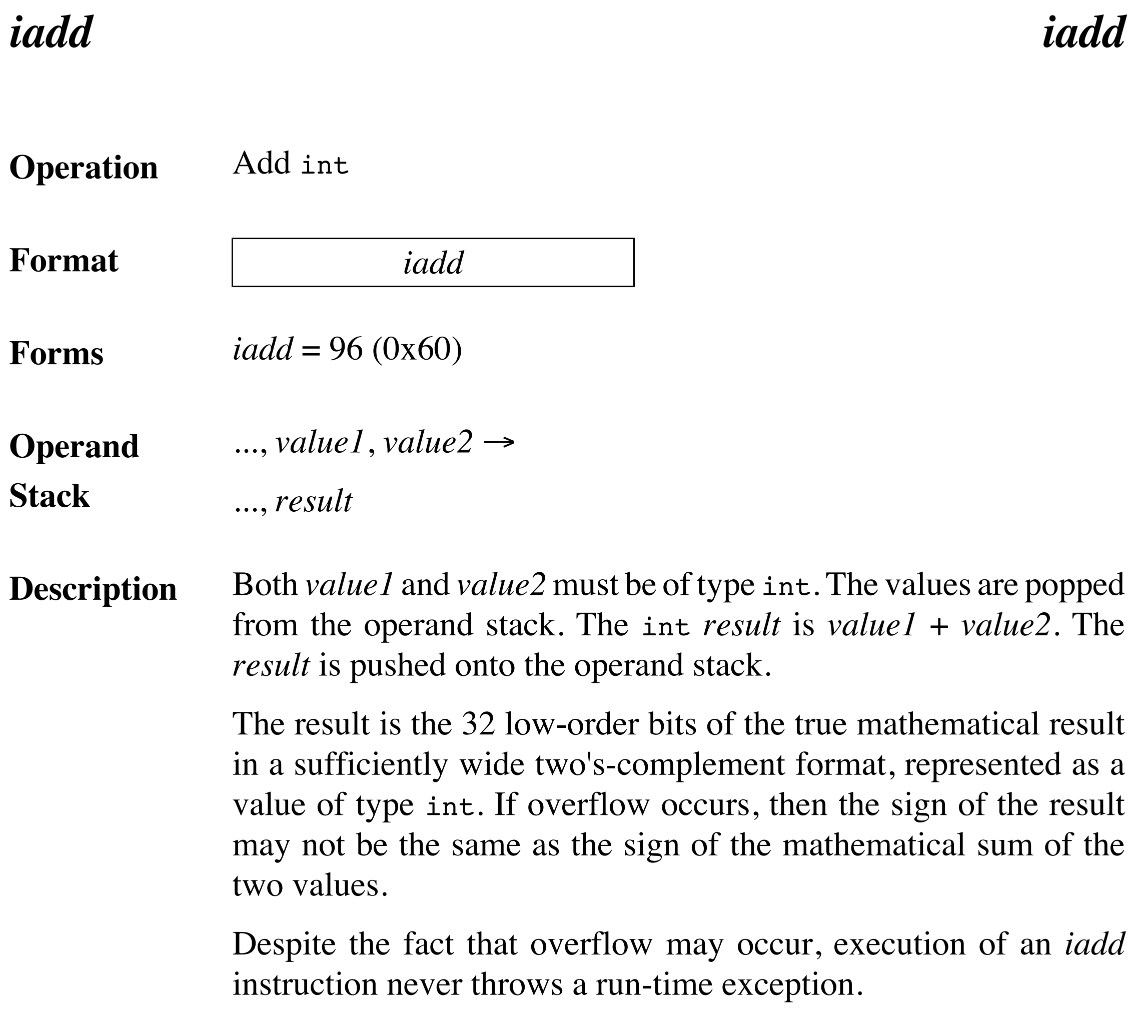
add: `iadd, ladd, fadd, dadd`
subtract: `isub, lsub, fsub, dsub`
multiply: `imul, lmul, fmul, dmul`

etc.

Conversion

```
i2s, i2b,
i2l, i2f, i2d,
l2f, l2d, f2d,
f2i, d2i, …
```

Specification of an instruction: iadd



iadd ***iadd***

Operation	Add `int`
Format	*iadd*
Forms	*iadd* = 96 (0x60)
Operand Stack	..., *value1*, *value2* → ..., *result*
Description	Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack. The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values. Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

Instruction-set: array instructions

3 main types of operations:

- Load: `baload, caload, daload, faload, iaload, laload, saload`
  - ..., *arrayref*, *index* → ..., *value*
- Store: `bastore, castore, dastore, fastore, iastore, lastore, sastore`
  - ..., *arrayref*, *index*, *value* → ...
  - Note: `baload` and `bastore` for both `byte` and `boolean` arrays
- Utilities: `newarray, anewarray, multinewarray, arraylength`

Instruction-set: stack instructions and control transfer

- **Operand stack manipulation**
  - `pop, pop2, dup, dup2, swap, dup_x1, dup_x2, dup2_x1, dup2_x2`
- **Control transfer**
  - **Unconditional:** `goto, jsr, ret,` …
  - **Conditional:** `ifeq, iflt, ifgt, if_icmpeq,` …

Method invocation

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

`Invokedynamic` and return

- **Method invocation:**
  - `invokedynamic`: invokes the method which is the target of the call site object bound to the invokedynamic instruction. The call site object was bound to a specific lexical occurrence of the invokedynamic instruction by the Java Virtual Machine as a result of running a bootstrap method before the first execution of the instruction. Therefore, each occurrence of an invokedynamic instruction has a unique linkage state, unlike the other instructions which invoke methods. (See also **http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html**)

- **Returning from methods:**
  - `return, ireturn, lreturn, areturn, freturn,` …

JVM Method Calls

- Sample Code

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

- Compiles to

```
Method int add12and13()
0 aload_0           // Push local variable 0 (this)
1 bipush 12         // Push int constant 12
3 bipush 13         // Push int constant 13
5 invokevirtual #4       // Method Example.addtwo(II)I
8 ireturn           // Return int on top of operand stack; it
                    //is the int result of addTwo()
```

invokevirtual *invokevirtual*

Operation Invoke instance method; dispatch based on class

Format

invokevirtual
indexbyte1
indexbyte2

Forms *invokevirtual* = 182 (0xb6)

Operand Stack ..., *objectref*, [*arg1*, [*arg2* ...]] →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3).

The resolved method must not be an instance initialization method, or the class or interface initialization method (§2.9).

If the resolved method is `protected`, and it is a member of a superclass of the current class, and the method is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

If the resolved method is not signature polymorphic (§2.9), then the *invokevirtual* instruction proceeds as follows.

Let *C* be the class of *objectref*. The actual method to be invoked is selected by the following lookup procedure:

1. If *C* contains a declaration for an instance method *m* that overrides (§5.4.5) the resolved method, then *m* is the method to be invoked.

2. Otherwise, if `C` has a superclass, a search for a declaration of an instance method that overrides the resolved method is performed, starting with the direct superclass of `C` and continuing with the direct superclass of that class, and so forth, until an overriding method is found or no further superclasses exist. If an overriding method is found, it is the method to be invoked.
3. Otherwise, if there is exactly one maximally-specific method (§5.4.3.3) in the superinterfaces of `C` that matches the resolved method's name and descriptor and is not `abstract`, then it is the method to be invoked.

The *objectref* must be followed on the operand stack by *nargs* argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is `synchronized`, the monitor associated with *objectref* is entered or reentered as if by execution of a *monitorenter* instruction (§*monitorenter*) in the current thread.

If the method is not `native`, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable 0, *arg1* in local variable 1 (or, if *arg1* is of type `long` or `double`, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

and 3 more pages…

Instruction-set: Heap Memory Allocation, Monitors

Create new class instance (object):

`new`

Create new array:

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

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

Critical Sections:

`monitorenter, monitorexit`

Special Instruction

- No operation: nop
- throw exception: athrow
- verifying instances: instanceof
- checking a cast operation: checkcast

Example: Factorial

```
class Factorial {

    int fac(int n) {
        int result = 1;
        for (int i=2; i<n; i++) {
            result = result * i;
        }
        return result;
    }
}
```

Compiling and Disassembling

```
% javac Factorial.java
% javap -c -verbose Factorial

Compiled from Factorial.java
class Factorial extends java.lang.Object {
    Factorial();
        /* Stack=1, Locals=1, Args_size=1 */
    int fac(int);
        /* Stack=2, Locals=4, Args_size=2 */
}

Method Factorial()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
```

Compiling and Disassembling …

```
                  // address: 0    1 2      3
Method int fac(int) // stack: this n result i
  0 iconst_1        // stack: this n result i 1
  1 istore_2        // stack: this n result i
  2 iconst_2        // stack: this n result i 2
  3 istore_3        // stack: this n result i
  4 goto 14
  7 iload_2         // stack: this n result i result
  8 iload_3         // stack: this n result i result i
  9 imul            // stack: this n result i result*i
  10 istore_2       // stack: this n result i
  11 iinc 3 1       // stack: this n result i
  14 iload_3        // stack: this n result i i
  15 iload_1        // stack: this n result i i n
  16 if_icmplt 7    // stack: this n result i
  19 iload_2        // stack: this n result i result
  20 ireturn
```

Limitations of the Java Virtual Machine

- Max number of entries in **constant pool**: **65535** (count in ClassFile structure)
- Max number of **fields**, of **methods**, of **direct superinterfaces**: **65535** (idem)
- Max number of **local variables** in the local variables array of a frame: **65535**, also by the 16-bit local variable indexing of the JVM instruction set.
- Max **operand stack** size: **65535**
- Max number of **parameters** of a method: **255**
- Max length of field and method names: **65535** characters by the 16-bit unsigned length item of the CONSTANT_Utf8_info structure
- Max number of **dimensions** in an **array**: **255**, by the size of the *dimensions* opcode of the *multianewarray* instruction and by the constraints imposed on the *multianewarray*, *anewarray*, and *newarray* instructions

Example: initialized field

```
public class SimpleClass {
   public int simpleField = 100;
}
```

```
public int simpleField;
    Signature: I
    flags: ACC_PUBLIC
```

```
public SimpleClass();
        Signature: ()V
        flags: ACC_PUBLIC
        Code:
        stack=2, locals=1, args_size=1
                0: aload_0
                1: invokespecial #1 // Method java/lang/Object."<init>":()V
                4: aload_0
                5: bipush 100
                7: putfield #2                    // Field simpleField:I 10: return
```

Example of bytecode obtained by compiling simple Java programs

- Examples of conditionals
- Examples of switch statements
  - Compiling as `tableswitch`
  - Compiling as `lookupswitch`
  - Switch over a string

See:

http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html

Coding: What's the difference?

```
System.out.println("Result = "+i);
```

```
System.out.print("Result = ");
System.out.println(i);
```

Coding: Avoiding garbage

```
System.out.println("Result = "+i);
```

```
getstatic     #3; // Field System.out:Ljava/io/PrintStream;
new           #4; // class StringBuffer
dup
invokespecial     #5; // StringBuffer."<init>":()V
ldc               #6; // String Result =
invokevirtual     #7; // StringBuffer.append:(LString;)LStringBuffer
iload_1
invokevirtual     #8; // StringBuffer.append:(I)LStringBuffer;
invokevirtual     #9; // StringBuffer.toString:()LString;
invokevirtual     #10;// PrintStream.println:(LString;)V
```

Coding: Avoiding garbage

```
System.out.print("Result = ");
System.out.println(i);
```

```
getstatic       #3; //Field System.out:Ljava/io/PrintStream;
ldc             #4; //String Result =
invokevirtual   #5; //Method PrintStream.print:(LString;)V
getstatic       #3; //Field System.out:LPrintStream;
iload_1
invokevirtual   #6; //Method PrintStream.println:(I)V
```