JVM Bytecode

Michael Rasmussen
ZeroTurnaround

JVM Bytecode

```
public class Test {

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

JVM Bytecode

```
00000000  ca fe ba be 00 00 00 31  00 22 0a 00 06 00 14 09  |.......1."......|
00000010  00 15 00 16 08 00 17 0a  00 18 00 19 07 00 1a 07  |................|
00000020  00 1b 01 00 06 3c 69 6e  69 74 3e 01 00 03 28 29  |.....<init>...()|
00000030  56 01 00 04 43 6f 64 65  01 00 0f 4c 69 6e 65 4e  |V...Code...LineN|
00000040  75 6d 62 65 72 54 61 62  6c 65 01 00 12 4c 6f 63  |umberTable...Loc|
00000050  61 6c 56 61 72 69 61 62  6c 65 54 61 62 6c 65 01  |alVariableTable.|
00000060  00 04 74 68 69 73 01 00  06 4c 54 65 73 74 3b 01  |..this...LTest;.|
00000070  00 04 6d 61 69 6e 01 00  16 28 5b 4c 6a 61 76 61  |..main...([Ljava|
00000080  2f 6c 61 6e 67 2f 53 74  72 69 6e 67 3b 29 56 01  |/lang/String;)V.|
00000090  00 04 61 72 67 73 01 00  13 5b 4c 6a 61 76 61 2f  |..args...[Ljava/|
000000a0  6c 61 6e 67 2f 53 74 72  69 6e 67 3b 01 00 0a 53  |lang/String;...S|
.
.
.
000001d0  b6 00 04 b1 00 00 00 02  00 0a 00 00 00 0a 00 02  |................|
000001e0  00 00 00 04 00 08 00 05  00 0b 00 00 00 0c 00 01  |................|
000001f0  00 00 00 09 00 10 00 11  00 00 00 01 00 12 00 00  |................|
00000200  00 02 00 13                                       |....|
```

JVM Bytecode

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

  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
}
```

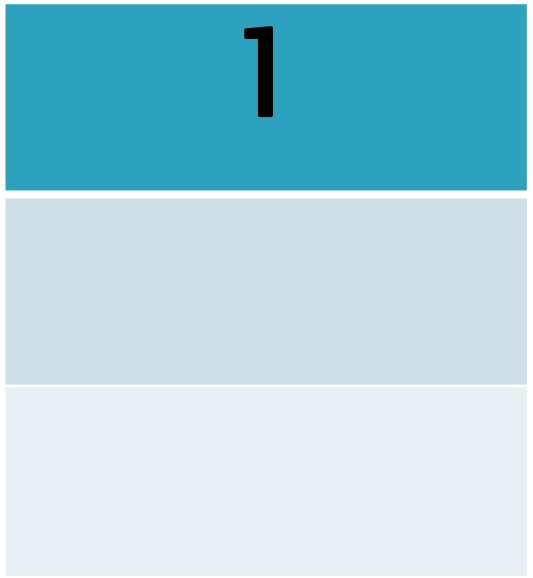
The JVM as a Stack Machine

- The JVM is a Stack Machine
  - Instructions operate on the stack
  - Arguments are popped from the stack
  - Results are pushed on to the stack
- 1 + 2
- Reverse Polish Notation
  - 1 2 +

The JVM as a Stack Machine

- 1 + 2
- Reverse Polish Notation
  - 1 2 +

PUSH 1

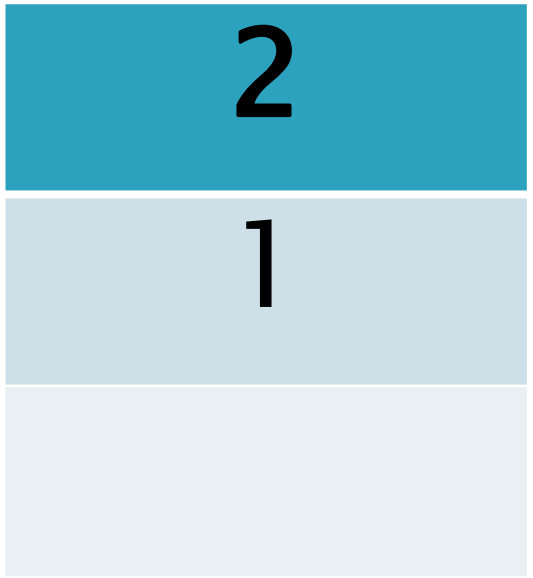


The JVM as a Stack Machine

- 1 + 2

- Reverse Polish Notation
  - 1 2 +

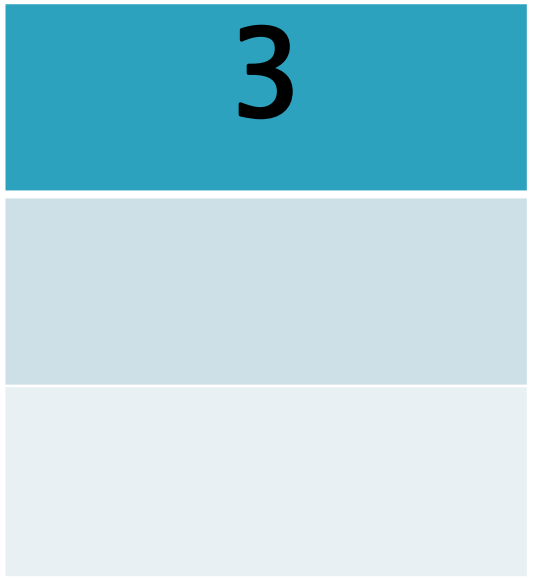
PUSH 1
PUSH 2



The JVM as a Stack Machine

- 1 + 2
- Reverse Polish Notation
  - 1 2 +

PUSH 1
PUSH 2
ADD

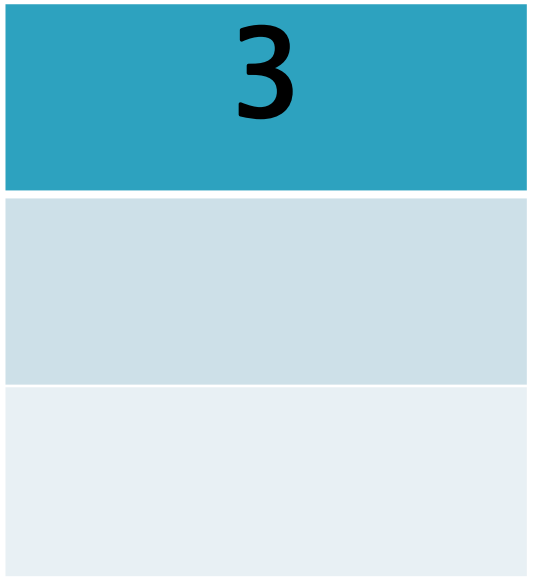


The JVM as a Stack Machine

- 1 + 2

- Reverse Polish Notation
  - 1 2 +

```
ICONST_1
ICONST_2
IADD
```



The JVM as a Stack Machine

- X + Y => X Y +

- Nested expressions?
  - A * B + C * D + E * F

The JVM as a Stack Machine

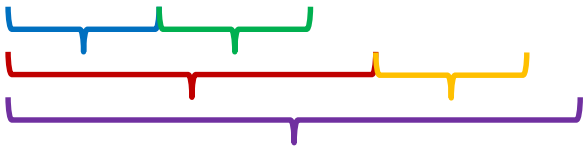
- X + Y => X Y +

- Nested expressions:
  - A * B + C * D + E * F
    - (A * B) + (C * D) + (E * F)
    - ((A * B) + (C * D)) + (E * F)
    - ((A B *) (C D *) +) (E F *) +
    - A B * C D * + E F * +

The JVM as a Stack Machine

- X + Y => X Y +

- Nested expressions:
  - A * B + C * D + E * F
    - (A * B) + (C * D) + (E * F)
    - ((A * B) + (C * D)) + (E * F)
    - ((A B *) (C D *) +) (E F *) +
    - A B * C D * + E F * +



```
ILOAD [A]
ILOAD [B]
IMUL
ILOAD [C]
ILOAD [D]
IMUL
IADD
ILOAD [E]
ILOAD [F]
IMUL
IADD
```

The JVM as a Stack Machine

- Each method invocation creates a new Frame
- Each frame has their own
  - Operand stack
  - Array of locals
  - Function Pointer
- Each thread has its own list of Frames

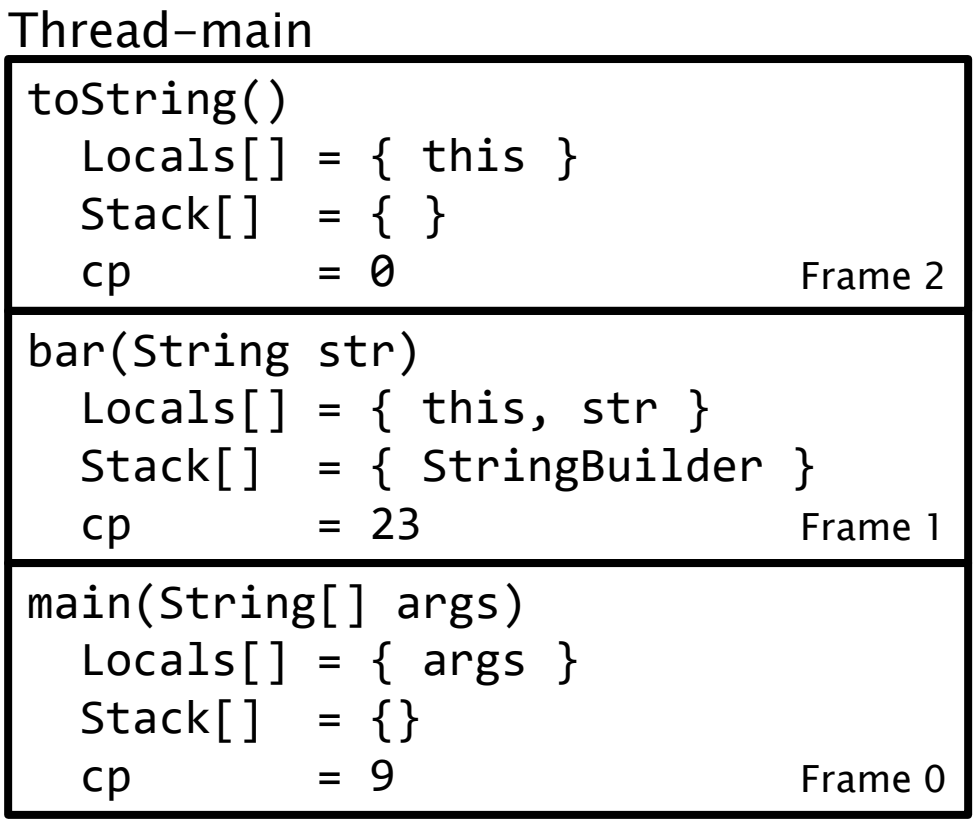
The JVM as a Stack Machine

```
public static void main(String[] args) {
  new Foo().bar("Hello");
}

void bar(String str) {
  String v = str + " " + toString();
  System.out.println(v);
}

public String toString() {
  return "World";
}
```

The JVM as a Stack Machine



Arithmetic and Constants

Type categories

- The JVM is type safe
  - Opcodes must match type

- Opcode categories
  - **I** 8-32 bit integer (1 stack slot)
  - **L** 64 bit integer (2 stack slots)
  - **F** 32 bit float (1 stack slot)
  - **D** 64 bit float (2 stack slots)
  - **A** Objects (1 stack slot)
  - **?A** Arrays (1 stack slot)

Arithmetic

- Arithmetic opcodes
  - `(x + y)` IADD, LADD, FADD, DADD
  - `(x - y)` ISUB, LSUB, FSUB, DSUB
  - `(-x)` INEG, LNEG, FNEG, DNEG
  - `(x * y)` IMUL, LMUL, FMUL, DMUL
  - `(x / y)` IDIV, LDIV, FDIV, DDIV
  - `(x % y)` IREM, LREM, FREM, DREM

  - `(x+=const, x-=const, x++, x--, ++x, --x)` IINC [local], [16-bit signed int]

Arithmetic

- Bitwise opcodes
  - `(x & y)` IAND, LAND
  - `(x | y)` IOR, LOR
  - `(x ^ y)` IXOR, LXOR
  - `(x << y)` ISHL, LSHL
  - `(x >> y)` ISHR, LSHR
  - `(x >>> y)` IUSHR, LUSHR

Type categories

- 0 + 1 (integers)
  - ICONST_0
    ICONST_1
    IADD
- 0.0 + 1.0 (double)
  - DCONST_0
  - DCONST_1
  - DADD

Constant opcodes

- Pushing constants to stack
  - ICONST_M1, ICONST_0 .. ICONST_6
  - LCONST_0, LCONST_1
  - FCONST_0, FCONST_1, FCONST_2
  - DCONST_0, DCONST_1
  - ACONST_NULL
  - LDC [number, string, class]
  - BIPUSH [integer number: -128 .. 127]
    SIPUSH [integer number: -32768 .. 32767]
  - Result
    - Constant pushed to the top of the stack

Constant opcodes

- 15 – 10 (integer)
  - BIPUSH 15 (or SIPUSH 15 or LDC 15)
    BIPUSH 10 (or SIPUSH 10 or LDC 10)
    ISUB
- 15.0 – 10.0 (double)
  - LDC 15.0
    LDC 10.0
    DSUB

Locals

Locals

- Locals are **this**, method parameters, local variables and other temporary values
- **this** and parameters are first
  - For non-static methods and constructors
    - **this** is stored in slot 0
    - First parameter is in slot 1
  - For static methods
    - First parameter is in slot 0
- Afterwards, local variables comes, compiler determines the order (often order of appearance)

- Note: **double** and **long** take up two slots!

Using locals

- Locals are confined to the frame
  - Entering a new frame creates a new list of locals exclusive to that frame
    - Same applies to the operand stack
- Locals retain value while the frame is alive
  - A frame is destroyed when the method exits

Locals

- Loading/storing values from/to locals
  - ILOAD/ISTORE [index]
  - LLOAD/LSTORE [index]
  - FLOAD/FSTORE [index]
  - DLOAD/DSTORE [index]
  - ALOAD/ASTORE [index]
  - Result
    - LOAD: Value from local pushed to the top of the stack
    - STORE: Value from stack is popped and stored in local
  - Note
    - LOAD = Copy, STORE = Move

Locals

- public void method(double d, int i) {
  double y = d * d;
  int x = i * i;
  }

- locals[0] == this (Object)
  locals[1] == d (double)
  locals[3] == i (int)
  locals[4] == y (double)
  locals[6] == x (int)

Locals

- public void method(double d, int i) {
  DLOAD 1
  DLOAD 1
  DMUL
  DSTORE 4

  ILOAD 3
  ILOAD 3
  IMUL
  ISTORE 6
  }

```
public void method(double d, int i) {
  double y = d * d;
  int x = i * i;
}
```

Locals

- public static void method(double d, int i) {
  double y = d * d;
  int x = i * i;
  }

- locals[0] == d (double)
  locals[2] == i (int)
  locals[3] == y (double)
  locals[5] == x (int)

Flow control

Flow control

- Flow control, often have some kind of condition, based on comparison, either explicit; boolean expressions can be considered an implicit == true
- if (a > b) { doThis(); } else { doThat(); }
- while (!isInterrupted()) { doSomething(); }
- for (int i = 0; i < len; i++) { doIt(); }

Flow control, int-comparison

- Conditional jumps
  - Integer comparison (jumps if condition is met)
    - Pop 2 values from the stack, and compare
      - IF_ICMP**EQ** [label] `if (i1 == i2)`
      - IF_ICMP**NE** [label] `if (i1 != i2)`
      - IF_ICMP**GE** [label] `if (i1 >= i2)`
      - IF_ICMP**GT** [label] `if (i1 > i2)`
      - IF_ICMP**LE** [label] `if (i1 <= i2)`
      - IF_ICMP**LT** [label] `if (i1 < i2)`

Flow control, int-comparison

- Conditional jumps
  - Integer comparison (jumps if condition is met)
    - Pop 1 value from the stack, and compare to 0
      - IFEQ [label]
      - IFNE [label]
      - IFGE [label]
      - IFGT [label]
      - IFLE [label]
      - IFLT [label]

```
if (i == 0)
if (bool != true)
if (i != 0)
if (bool == true)
if (i >= 0)
if (i  > 0)
if (i <= 0)
if (i  < 0)
```

Flow control, unconditional

- Unconditional jump
  - GOTO [label]
- Exiting methods
  - Normal return
    - RETURN, ARETURN, IRETURN, LRETURN, FRETURN, DRETURN
  - Throwing exception
    - ATHROW

Control Flow

```
public int gt(int x, int y) {
  if (x > y)
    return 1;
  else
    return -1;
}
```

Control flow

- Resulting bytecode:
  - ILOAD 1 // push local[1] {x}
    ILOAD 2 // push local[2] {y}
    IF_ICMPGT :gt // if local[1] > local[2] jump
    ICONST_M1 // push −1
    IRETURN // return value
    :gt
    ICONST_1 // push +1
    IRETURN // return value

```
public int gt(int x, int y) {
  if (x > y)
    return 1;
  else
    return -1;
}
```

Control flow

- Resulting bytecode (reversed logic):
  - ILOAD 1 // push local[1] {x}
    ILOAD 2 // push local[2] {y}
    IF_ICMPLE :le // if local[1] <= local[2] jump
    ICONST_1 // push +1
    IRETURN // return value
    :le
    ICONST_M1 // push -1
    IRETURN // return value

```
public int gt(int x, int y) {
  if (x > y)
    return 1;
  else
    return -1;
}
```

Control flow

```
static int calc(int count) {
  int result = 0;
  while (count > 0) {
    result += count--;
  }
  return result;
}
```

Control flow

- Resulting bytecode:
  - ICONST_0 // push 0
  - ISTORE 1 // store in local[1] {result}
  - :loop
  - ILOAD 0 // push local[0] {count}
  - IFLE :end // if local[0] <= 0 goto :end
  - ILOAD 1 // push local[1]
  - ILOAD 0 // push local[0]
  - IADD // add together
  - ISTORE 1 // store result in local[1]
  - IINC 0, -1 // increment local[0] by -1
  - GOTO :loop
  - :end
  - ILOAD 1 // push local[1]
  - IRETURN // return value

```
static int calc(int count) {
  int result = 0;
  while (count > 0) {
    result += count--;
  }
  return result;
}
```

Object model

Object model

- Method invocations
  - INVOKESTATIC [owner], [name], [desc]
  - INVOKESPECIAL [owner], [name], [desc]
  - INVOKEVIRTUAL [owner], [name], [desc]
  - INVOKEINTERFACE [owner], [name], [desc]
  - INVOKEDYNAMIC [name], [desc], [bsm], [{args}]
    - Note: method invocations pops a variable number of entries of the stack, depending on type and desc.

Fields

- Field access
- Static fields:
  - GETSTATIC [owner], [name], [type]
  - PUTSTATIC [owner], [name], [type]
- Non-static fields:
  - GETFIELD [owner], [name], [type]
  - PUTFIELD [owner], [name], [type]

Object model

- Object allocation
  - NEW [class]
    - Constructor needs to be invoked afterwards

- Array allocation
  - MULTIANEWARRAY [type] [dim] — Multi-dim array
  - ANEWARRAY [type] — Object array
  - NEWARRAY [type] — Primitive array

    - Pops size of array from stack

Method invocation

Method invocation

- Virtual method invocation
  - INVOKEVIRTUAL, –SPECIAL, and –INTERFACE
    - Requires target object to be on stack and arguments of types as described by method descriptor
- Static method invocation
  - INVOKESTATIC
    - Requires arguments to be on stack of types as described by the method descriptor

Method invocation

- Static method invocation:

- ```
  static String getVer () {
      return System.getProperty(
          "java.version",
          "1.6");
  }
  ```
  ```

Method invocation

- Resulting bytecode:
  - LDC “java.version” // push “java.version”
  - LDC “1.6” // push “1.6”
  - INVOKESTATIC // invoke the static method
    java/lang/System // using the two Strings on
    getProperty // stack as arguments
    (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
  - ARETURN // return the resulting value

```
static String getVer () {
   return System.getProperty(
      "java.version",
      "1.6");
}
```

Type descriptors

- **I** 8-32 bit integer
  - **Z** boolean — 8 bit boolean representation
  - **C** char — 16 bit unsigned Unicode character
  - **B** byte — 8 bit signed integer (two's complement)
  - **S** short — 16 bit signed integer (two's complement)
  - **I** int — 32 bit signed integer (two's complement)
- **L** 64 bit integer
  - **J** long — 64 bit signed integer (two's complement)
- **F** 32 bit float
  - **F** float — 32 bit IEEE 754 single-precision float
- **D** 64 bit float
  - **D** double — 64 bit IEEE 754 double-precision float
- **A** Objects
  - **L** Object
- **?A** Arrays
  - **[** Arrays

Method invocation

- Writing to System.out:

- ```
  static void hello() {
    System.out.println("Hello World");
  }
  ```
  ```

Method invocation

- Resulting bytecode:
  - GETSTATIC
    java/lang/System
    out
    Ljava/io/PrintStream; // Get the static variable System.out
    // which is of the type PrintStream

    LDC “Hello World” // Push “Hello World” to the stack

    INVOKEVIRTUAL
    java/io/PrintStream // invoke println on PrintStream
    println // with the pushed string
    (Ljava/lang/String;)V

    RETURN // All methods must return,
    // even void methods

  -

```
static void hello() {
  System.out.println(“Hello World”);
}
```

Other opcodes

Stack manipulation

- On-stack manipulation
  - SWAP
    - Swap top two stack items
  - POP, POP2
    - Remove top/top-2 stack items
  - DUP, DUP2
    - Duplicate top/top-2 stack items
  - DUP_X1, DUP_X2
    - Duplicate top stack item 1 down/2 down
  - DUP2_X1, DUP2_X2
    - Duplicate top-2 stack items 1 down/2 down

Misc opcodes

- No-operation
  - NOP
- Synchronization
  - MONITORENTER
  - MONITOREXIT
- Deprecated sub-routine
  - JSR
  - RET

Tooling

Tooling

- Seeing the bytecode of a class
  - javap
    - Part of the JDK
  - Many IDEs have plugins for this as well
- Popular Java-libraries for bytecode
  - ASM
    - http://asm.ow2.org/
  - Javassist
    - http://www.javassist.org/
  - BCEL
    - http://commons.apache.org/proper/commons-bcel/

Tools: javap

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Tools: javap

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

  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
}
```

Tools: ASM

- Using ASM to generate bytecode
  - Visitor pattern
    - visit the individual parts of a class' bytecode
  - ClassWriter
    - Represents the class when writing
    - toByteArray()
  - MethodVisitor
    - Represents methods in a class

Tools: ASM

- Basics for generating a class
  - ```
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);

    cw.visit(V1_5, ACC_PUBLIC, ”className", null,
       Type.getInternalName(Object.class), null);

    // Visit other class metadata and annotations

    // Visit individual fields and methods
    // cw.visitField(...)
    // cw.visitMethod(...)

    cw.visitEnd();

    byte[] classBytes = cw.toByteArray();
    // define classBytes using a ClassLoader
    ```
    ```

Tools: ASM

- Basics for generating a method
  - ```
    MethodVisitor mv = cw.visitMethod(ACC_PUBLIC | ACC_STATIC, "main",
                                      "([Ljava/lang/String;)V", null, null);

    mv.visitFieldInsn(GETSTATIC,
       Type.getInternalName(System.class),
       "out”,
       Type.getDescriptor(PrintStream.class));

    mv.visitLdcInsn("Hello World!");

    mv.visitMethodInsn(INVOKEVIRTUAL,
       Type.getInternalName(PrintStream.class),
       "println”,
       Type.getMethodDescriptor(Type.getType(void.class),
         Type.getType(String.class)),
       false);

    mv.visitInsn(RETURN);

    mv.visitMaxs(2, 1);
    mv.visitEnd();
    ```
    ```

JMV Bytecode with ASM

Michael Rasmussen
ZeroTurnaround

Binary Name

- Binary names
  - In Java source code file
    - Uses dots to separate, e.g. java.lang.String
    - In accordance with the JLS
      - http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html#jls-13.1
  - In class file format
    - Uses slash to separate, e.g. java/lang/String
      - Also commonly referred to as **Internal name** or **Internal form**
    - In accordance with the JVMS
      - http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.2

Binary Name

- String representing a class' name
- Fully qualified class name
  - uses / as package separator
- Example:
  - java.lang.String "java/lang/String"
  - java.util.ArrayList "java/util/ArrayList"

Type Descriptor

- Internal definition of a type
- Primitive types
  - Single character
    - “J” (long), “I” (int), “S” (short), “B” (byte), “C” (char)
    - “F" (float), “D” (double)
    - “Z” (boolean), “V” (void)

Type Descriptor

- Object types
  - Binary name enclosed by L;
    - “Ljava/lang/String;”
    - “Ljava/util/ArrayList;”
- Array types
  - [followed by type descriptor
    - “[B” byte[]
    - “[Ljava/lang/String;” String[]
    - “[[D” double[][]
      - Multiple [indicates multiple dimensions

Method Descriptor

- Defines parameter and return types
- Consists of:
  - Enclosed in parenthesis: 0 or more Type Descriptors
    - Describing the parameters
  - 1 Type Descriptor
    - Describing the return type

Method Descriptor

- Example
  - “(Ljava/lang/String;)I”
    - Takes a java.lang.String as arguments
    - Returns an int
  - “()V”
    - Takes zero arguments
    - Returns nothing (void)
  - “(DD)D”
    - Takes two arguments of type double
    - Returns a double

Special Methods

- Constructor
  - Special name: “<init>”
  - **Must** invoke constructor on superclass (or another constructor on this class)
- Static initializer
  - Special name: “<clinit>”
  - Static method invoked when the class is initialized
    - Ensured to only be called once
    - Ensured to have been called before any methods on the class is called

Access / Modifiers

- Bit-mask containing access flags
  - ACC_PUBLIC — class, field, method
  - ACC_PRIVATE — class, field, method
  - ACC_PROTECTED — class, field, method
  - ACC_STATIC — field, method
  - ACC_FINAL — class, field, method, parameter
  - ACC_SUPER — class
  - ACC_SYNCHRONIZED — method
  - ACC_VOLATILE — field
  - ACC_BRIDGE — method
  - ACC_VARARGS — method
  - ACC_TRANSIENT — field
  - ACC_NATIVE — method
  - ACC_INTERFACE — class
  - ACC_ABSTRACT — class, method
  - ACC_STRICT — method
  - ACC_SYNTHETIC — class, field, method, parameter
  - ACC_ANNOTATION — class
  - ACC_ENUM — class, field
  - ACC_MANDATED — parameter

JVM Bytecode

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

  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
}
```

Basic ASM Classes

ASM

- Maven dependency:

```
<dependency>
    <groupId>org.ow2.asm</groupId>
    <artifactId>asm-debug-all</artifactId>
    <version>5.2</version>
</dependency>
```

- Gradle dependency:

```
compile 'org.ow2.asm:asm-debug-all:5.2'
```

Basic ASM Classes

- ASM is based on the Visitor pattern

- ClassVisitor – visit Class members
  - ClassWriter – write .class bytes
  - ClassReader – read .class bytes

- MethodVisitor – visit Method bytecode
  - GeneratorAdapter – wrapper to ease generation

- Label – Branch target
- Type – Helper class for types

ClassVisitor

- Base Visitor to visit a class
  - Contains visitors for (among others):
    - Describing the class, superclass, interfaces
    - Visiting fields
    - Visiting methods

ClassVisitor

- visit(...)
  - Visits the overall class
    - int version — Java version
    - int access — Access for class (ACC_PUBLIC)
    - String name — Name of class
    - String signature — Generic signature
    - String superName — Name of superclass
    - String[] interfaces — Names of implemented interfaces

ClassVisitor

- visitField(…) : FieldVisitor
  - Visit a field on the class
    - int access — Access for the field (public/private)
    - String name — Name of the field
    - String desc — Type descriptor of the field
    - String signature — Generic signature for the field
    - Object value — Default value (for static primitives)

ClassVisitor

- visitMethod(…) : MethodVisitor
  - Visit a method on the class
    - int access — Access for the method
    - String name — Name of the method
    - String desc — Method descriptor of the method
    - String signature — Generic signature for the method
    - String[] exceptions — Declared thrown exceptions

ClassWriter

- Extends ClassVisitor
- Basic class to generate a class
- Use toByteArray() to get actual bytecode

ClassReader

- Basic class to parse a class
- Use accept() with a ClassVisitor
  - visits on the passed ClassVisitor the individual parts of the class being parsed

MethodVisitor

- Base Visitor to visit a method
- Has visit methods for individual bytecode opcode-groups

MethodVisitor

- visitInsn(int opcode)
  - Visits a zero operand instruction
    - xRETURN, ATHROW
    - xCONST_n
    - xALOAD, xASTORE, ARRAYLENGTH
    - xADD, xSUB, xMUL, xDIV, xREM, xNEG
    - xOR, xAND, xXOR, xSHL, xSHR, xUSHR
    - I2x, L2x, F2x, D2x
    - LCMP, FCMPx, DCMPx
    - NOP, SWAP, DUP, DUP2, POP, POP2, DUP_x
    - MONITORENTER, MONITOREXIT

MethodVisitor

- visitIntInsn(int opcode, int operand)
  - Visits an instruction with a single int operand
    - BIPUSH, SIPUSH
      - Pushed the value “operand” onto the stack
    - NEWARRAY
      - Creates a primitive array of the type specified by “operand”:
        - T_BOOLEAN, T_CHAR, T_FLOAT, T_DOUBLE, T_BYTE, T_SHORT, T_INT, T_LONG
        - Size of array is popped from the stack

MethodVisitor

- visitVarInsn(int opcode, int var)
  - Visits a local instruction
    - xLOAD
    - xSTORE
  - “var” indicates the local index to access

MethodVisitor

- visitTypeInsn(int opcode, String type)
  - Visits a type instruction.
    - NEW
    - ANEWARRAY
    - CHECKCAST
    - INSTANCEOF
  - “type” is the type’s binary name

MethodVisitor

- visitFieldInsn(int opcode, String owner, String name, String desc)
  - Visits a field instruction
    - GETSTATIC, PUTSTATIC
    - GETFIELD, PUTFIELD
  - “owner” is the binary-name of the class that holds the field
  - “name” is the name of the field
  - “desc” is the type-descriptor for the field

MethodVisitor

- visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf)
  - Visits a method instruction
    - INVOKESTATIC
    - INVOKEVIRTUAL, INVOKESPECIAL, INVOKEINTERFACE
  - “owner” is the binary-name of the class that holds the method
  - “name” is the name of the method
  - “desc” is the method-descriptor of the method
  - “itf” indicates if it’s an interface-method

MethodVisitor

- visitJumpInsn(int opcode, Label label)
  - Visits a jump instruction.
    - GOTO, IF_ICMPx, IFx, IF_ACMPx, etc
  - “label” points to the target location to jump to if condition is met

MethodVisitor

- visitLdcInsn(Object cst)
  - Visits an LDC instruction.
    - LDC
  - If “cst” is of the type Integer, Long, Float or Double, the value of that number is used as a constant.
  - If “cst” is of type String, the content of that String is used as a constant.
  - If “cst” is of type Type, the type indicated by that Type is used as a constant (equivalent to using .class in Java source code)

MethodVisitor

- visitIincInsn(int var, int increment)
  - Visits an IINC instruction.
    - IINC
  - “var” points to the local index to modify
  - “increment” is the value the local should be incremented by (valid range: -32768 .. 32767)

MethodVisitor

- visitLabel(Label label)
  - Visits a label.
  - Adds the target location “label”, for use with, among others, jump opcodes.

GeneratorAdapter

- Wrapper around MethodVisitor
  - Convenience methods for most opcodes
  - Easier to write
  - Takes care of long/double taking two slots!
    - Access arguments by index, not by local
  - Takes care of using the right opcode depending on type

Label

- Symbolizes branch targets in methods
  - Example:
    - ```
      Label labelGoto = new Label();
      mv.visitJumpInsn(GOTO, labelGoto);
      mv.visitLabel(labelGoto);
      ```
      ```

Type

- ASM class for type/method descriptors
  - Contains several useful helper methods
    - Type.getInternalName(Class c) : String
      - Returns the binary-name for a class
    - Type.getDescriptor(Class c) : String
      - Returns the type-description for a class
    - Type.getType(Class c) : Type
      - Return a Type object for the specified class
    - Type.getMethodDescriptor(Type returnType, Type... paramTypes) : String
      - Returns the method-descriptor for the types specified

Generating bytecode

Generating bytecode

- Basics for generating a class
  - ```
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);

    cw.visit(V1_6, ACC_PUBLIC, "className", null,
       Type.getInternalName(Object.class), null);

    // Visit individual fields and methods
    // cw.visitField(...)
    // cw.visitMethod(...)

    cw.visitEnd();

    byte[] classBytes = cw.toByteArray();
    // define classBytes using a ClassLoader
    ```
    ```

Generating bytecode

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

Generating bytecode

- Resulting bytecode:
  -
```
GETSTATIC                    // Get the static variable System.out
     java/lang/System
     out
     Ljava/io/PrintStream;   // which is of the type PrintStream

LDC "Hello World"            // Push “Hello World” to the stack

INVOKEVIRTUAL
   java/io/PrintStream       // invoke println on PrintStream
   println                   // with the pushed string
   (Ljava/lang/String;)V

RETURN                       // All methods must return,
                             // even void methods
```

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

Generating bytecode

- Basics for generating a method
  - ```
    MethodVisitor mv = cw.visitMethod(ACC_PUBLIC | ACC_STATIC, "main",
        Type.getMethodDescriptor(Type.VOID_TYPE, Type.getType(String[].class)),
        null, null);
    mv.visitCode();

    mv.visitFieldInsn(GETSTATIC,
       Type.getInternalName(System.class),
       "out",
       Type.getDescriptor(PrintStream.class));

    mv.visitLdcInsn("Hello World!");

    mv.visitMethodInsn(INVOKEVIRTUAL,
       Type.getInternalName(PrintStream.class),
       "println”,
       Type.getMethodDescriptor(Type.VOID_TYPE, Type.getType(String.class)),
       false);

    mv.visitInsn(RETURN);

    mv.visitMaxs(2, 1);
    mv.visitEnd();
    ```

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

Generating bytecode

- GeneratorAdapter
  - ```
    MethodVisitor mv = cw.visitMethod(ACC_PUBLIC | ACC_STATIC, "main",
        Type.getMethodDescriptor(Type.VOID_TYPE, Type.getType(String[].class)),
        null, null);
    GeneratorAdapter ga = new GeneratorAdapter(mv, ACC_PUBLIC | ACC_STATIC, "main",
        Type.getMethodDescriptor(Type.VOID_TYPE, Type.getType(String[].class)));

    ga.getStatic(Type.getType(System.class), "out", Type.getType(PrintStream.class));

    ga.push("Hello World");

    ga.invokeVirtual(Type.getType(PrintStream.class),
        Method.getMethod("void println(java.lang.String)"));

    ga.returnValue();
    ga.endMethod();
    ```

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

Generating bytecode

```
public int gt(int x, int y) {
  if (x > y)
    return 1;
  else
    return -1;
}
```

Generating bytecode

- Resulting bytecode:
  - ```
    ILOAD 1          // push local[1] {x}
    ILOAD 2          // push local[2] {y}
    IF_ICMPLE :le    // if local[1] <= local[2] jump
    ICONST_1         // push +1
    IRETURN          // return value
    :le
    ICONST_M1        // push -1
    IRETURN          // return value
    ```

```
public int gt(int x, int y) {
  if (x > y)
    return 1;
  else
    return -1;
}
```
```

Generating bytecode

- Basic loop example:

```
public static void count() {
  for (int i = 0; i < 10; i++) {
    System.out.println(i);
  }
}
```

Generating bytecode

- Basic loop example:

```
MethodVisitor mv = cw.visitMethod(ACC_PUBLIC | ACC_STATIC,
    "count", Type.getMethodDescriptor(Type.VOID_TYPE),
    null, null);
Label labelLoop = new Label();
Label labelEnd = new Label();

mv.visitCode();

mv.visitInsn(ICONST_0);
mv.visitVarInsn(ISTORE, 1);

mv.visitLabel(labelLoop);
mv.visitVarInsn(ILOAD, 1);
mv.visitIntInsn(BIPUSH, 10);
mv.visitJumpInsn(IF_ICMPGE, labelEnd);

mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
mv.visitVarInsn(ILOAD, 1);
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(I)V", false);

mv.visitIincInsn(1, 1);

mv.visitJumpInsn(GOTO, labelLoop);

mv.visitLabel(labelEnd);
mv.visitInst(RETURN);

mv.visitMaxs(2, 1);
mv.visitEnd();
```

```
public static void count() {
  for (int i = 0; i < 10; i++) {
    System.out.println(i);
  }
}
```

Generating bytecode

- Basic loop example:
  - ```
    MethodVisitor mv = cw.visitMethod(ACC_PUBLIC | ACC_STATIC, "count",
        Type.getMethodDescriptor(Type.VOID_TYPE), null, null);
    GeneratorAdapter ga = new GeneratorAdapter(mv, ACC_PUBLIC | ACC_STATIC, "count",
        Type.getMethodDescriptor(Type.VOID_TYPE));


    int localI = ga.newLocal(Type.INT_TYPE);
    Label labelLoop = ga.newLabel();
    Label labelEnd = ga.newLabel();

    ga.push(0);
    ga.storeLocal(localI);

    ga.visitLabel(labelLoop);
    ga.loadLocal(localI);
    ga.push(10);
    ga.ifICmp(GeneratorAdapter.GE, labelEnd);

    ga.getStatic(Type.getType(System.class), "out", Type.getType(PrintStream.class));
    ga.loadLocal(localI);
    ga.invokeVirtual(Type.getType(PrintStream.class), Method.getMethod("void println(int)"));

    ga.iinc(localI, 1);
    ga.goTo(labelLoop);

    ga.visitLabel(labelEnd);
    ga.returnValue();

    ga.endMethod();
    ```

```
public static void count() {
  for (int i = 0; i < 10; i++) {
    System.out.println(i);
  }
}
```
```

Generating bytecode

```
mv = cw.visitMethod(ACC_PUBLIC, "calc", "(II)I", null, null);
Label labelStart = new Label();
Label labelEnd = new Label();
mv.visitCode();

mv.visitInsn(ICONST_1);
mv.visitVarInsn(ISTORE, 3);

mv.visitInsn(ICONST_0);
mv.visitVarInsn(ISTORE, 4);

mv.visitLabel(labelStart);
mv.visitVarInsn(ILOAD, 4);
mv.visitVarInsn(ILOAD, 2);
mv.visitJumpInsn(IF_ICMPGE, labelEnd);

mv.visitVarInsn(ILOAD, 3);
mv.visitVarInsn(ILOAD, 1);
mv.visitInsn(IMUL);
mv.visitVarInsn(ISTORE, 3);

mv.visitIincInsn(4, 1);
mv.visitJumpInsn(GOTO, labelStart);

mv.visitLabel(labelEnd);
mv.visitVarInsn(ILOAD, 3);
mv.visitInsn(IRETURN);

mv.visitMaxs(2, 5);
mv.visitEnd();
```

Generating bytecode

```
mv = cw.visitMethod(ACC_PUBLIC, "calc", "(II)I", null, null);
Label labelStart = new Label();
Label labelEnd = new Label();
mv.visitCode();

mv.visitInsn(ICONST_1);
mv.visitVarInsn(ISTORE, 3);                          // int local3 = 1

mv.visitInsn(ICONST_0);
mv.visitVarInsn(ISTORE, 4);                          // int local4 = 0

mv.visitLabel(labelStart);
mv.visitVarInsn(ILOAD, 4);
mv.visitVarInsn(ILOAD, 2);
mv.visitJumpInsn(IF_ICMPGE, labelEnd);               // if (local4 >= local2) goto end

mv.visitVarInsn(ILOAD, 3);
mv.visitVarInsn(ILOAD, 1);
mv.visitInsn(IMUL);
mv.visitVarInsn(ISTORE, 3);                          // local3 = local3 * local1;

mv.visitIincInsn(4, 1);                              // local4++
mv.visitJumpInsn(GOTO, labelStart);                  // goto start

mv.visitLabel(labelEnd);
mv.visitVarInsn(ILOAD, 3);
mv.visitInsn(IRETURN);                               // return result

mv.visitMaxs(2, 5);
mv.visitEnd();
```

Generating bytecode

```
mv = cw.visitMethod(ACC_PUBLIC, "calc", "(II)I", null, null);
Label labelStart = new Label();
Label labelEnd = new Label();
mv.visitCode();

mv.visitInsn(ICONST_1);
mv.visitVarInsn(ISTORE, 3);

mv.visitInsn(ICONST_0);
mv.visitVarInsn(ISTORE, 4);

mv.visitLabel(labelStart);
mv.visitVarInsn(ILOAD, 4);
mv.visitVarInsn(ILOAD, 2);
mv.visitJumpInsn(IF_ICMPGE, labelEnd);

mv.visitVarInsn(ILOAD, 3);
mv.visitVarInsn(ILOAD, 1);
mv.visitInsn(IMUL);
mv.visitVarInsn(ISTORE, 3);

mv.visitIincInsn(4, 1);
mv.visitJumpInsn(GOTO, labelStart);

mv.visitLabel(labelEnd);
mv.visitVarInsn(ILOAD, 3);
mv.visitInsn(IRETURN);

mv.visitMaxs(2, 5);
mv.visitEnd();
```

```
public void calc(int x, int y) {
  int result = ;
  for (int i = 0; i < y; i++) {
    result = result * x;
  }
  return result;
}
```