JVM Bytecode

Michael Rasmussen
ZeroTurnaround

Today’s Outline

- Intro
- The JVM as a Stack Machine
- Bytecode taxonomy
- Stack manipulation
- Using locals
- Control flow
- Method invocation
- Tooling
- Next time

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

- Welcome my son
  Welcome to the machine
  Where have you been?
  It's alright we know where you've been.

The JVM as a Stack Machine

- The JVM is a Stack Machine
- Each method invocation creates a new Frame
- Each frame has their own
  - Operand stack
  - Array of locals

The JVM as a Stack Machine

- 1 + 2

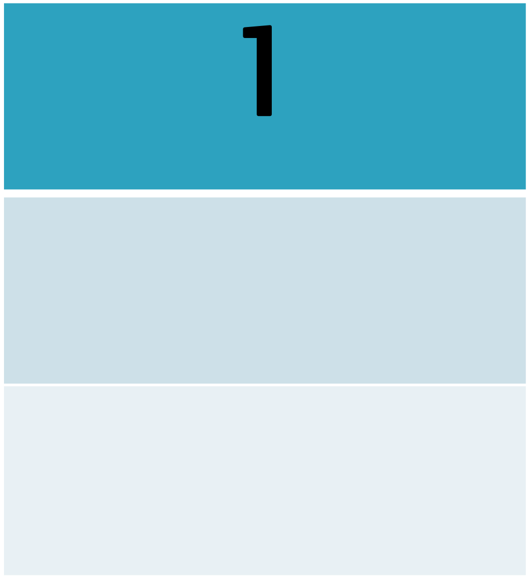
- Reverse Polish Notation
  - 1 2 +

The JVM as a Stack Machine

- 1 + 2

- Reverse Polish Notation
  - 1 2 +

PUSH 1



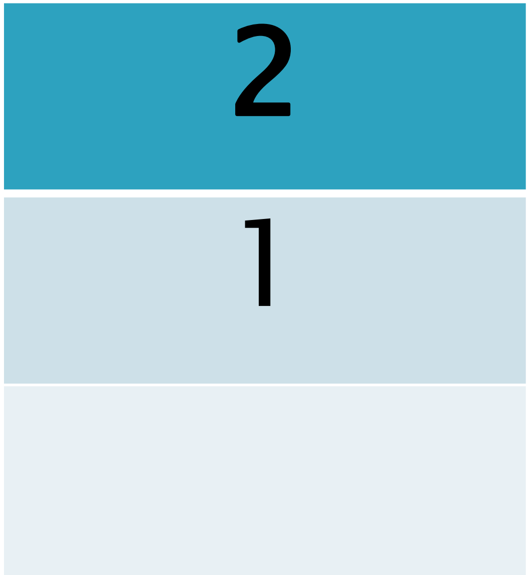
1

The JVM as a Stack Machine

- 1 + 2

- Reverse Polish Notation
  - 1 2 +

PUSH 1
PUSH 2



2

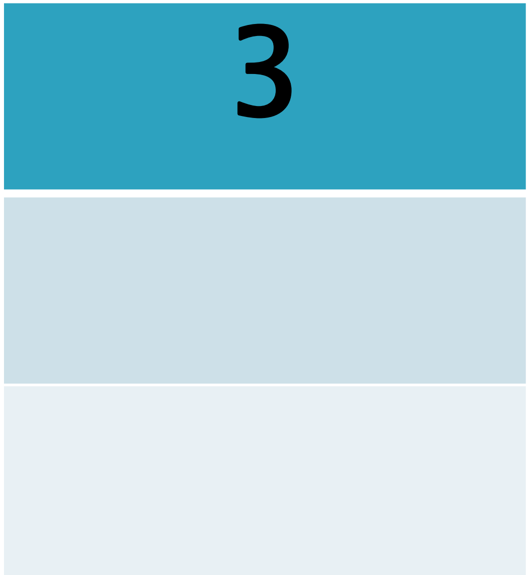
1

The JVM as a Stack Machine

- 1 + 2

- Reverse Polish Notation
  - 1 2 +

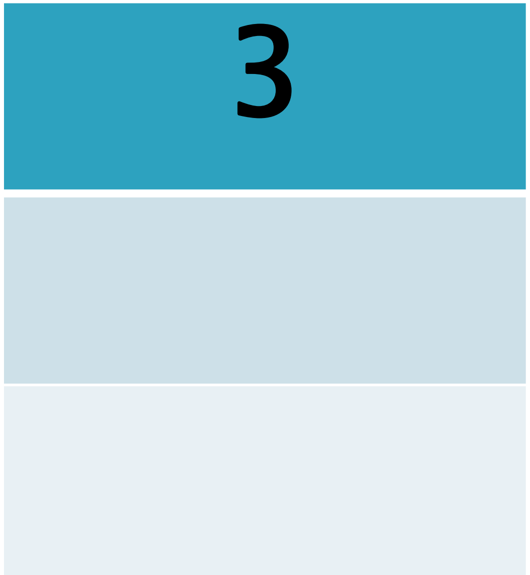
```
PUSH 1
PUSH 2
ADD
```



The JVM as a Stack Machine

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

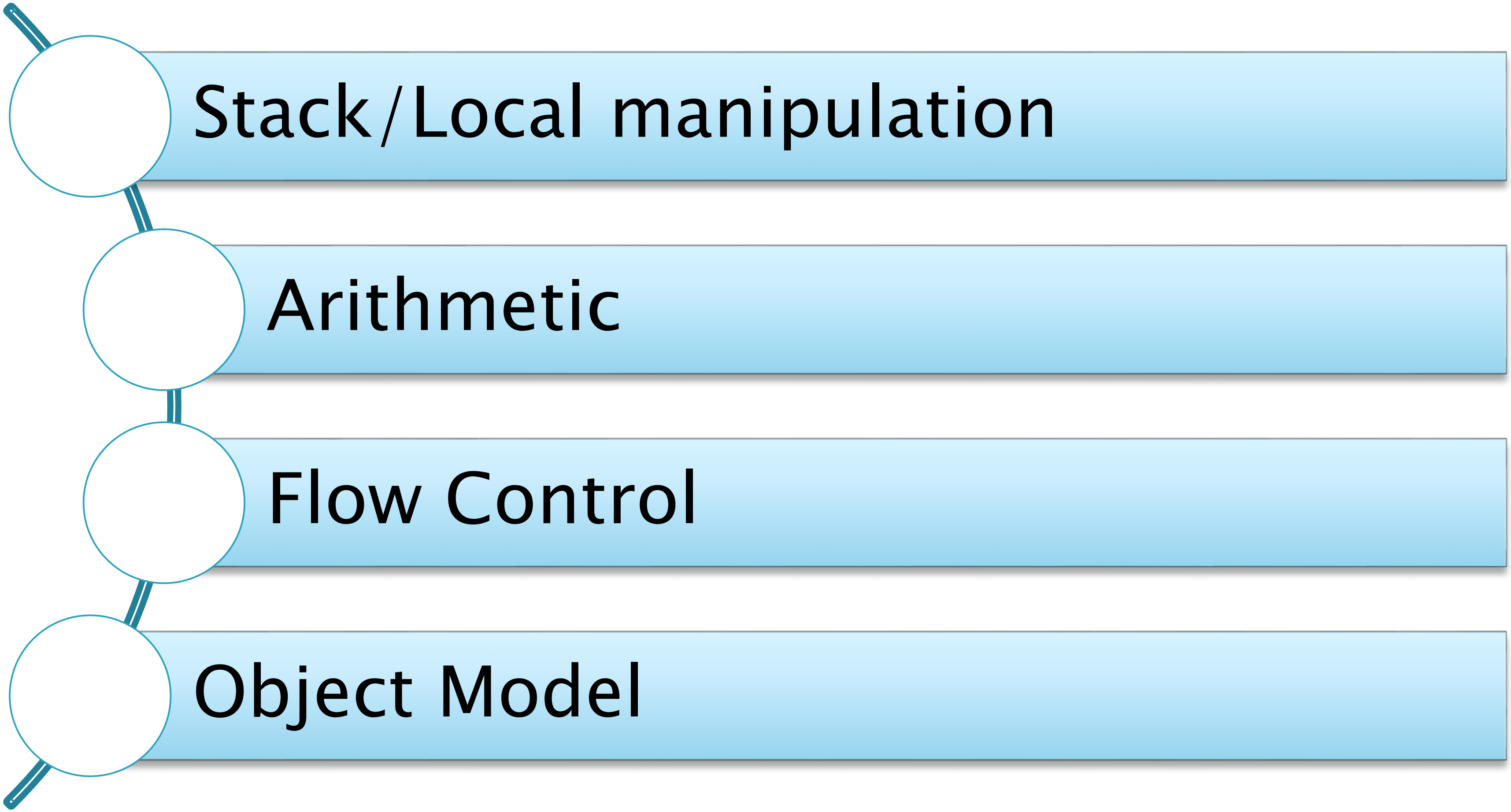
```
ICONST_1
ICONST_2
IADD
```



Bytecode taxonomy

- All we hear is
  Radio Ga Ga
  Radio Goo Goo
  Radio Ga Ga

Bytecode Taxonomy



- Stack/Local manipulation
- Arithmetic
- Flow Control
- Object Model

Type categories

- JVM is typesafe
  - 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 slot)
A	Objects	(1 stack slot)
?A	Arrays	(1 stack slot)

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	

Stack manipulation

- Pushing constants to stack
  - ICONST_M1, ICONST_0 .. ICONST_6
  - LCONST_0, LCONST_1
  - FCONST_0 .. FCONST_2
  - DCONST_0, DCONST_1
  - ACONST_NULL
  - LDC [number, string, class]
  - BIPUSH [byte number], SIPUSH [short number]
  - Result
    - Constant pushed to the top of the stack

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

Local manipulation

- Loading values from locals
    - ILOAD [index]
    - LLOAD [index]
    - FLOAD [index]
    - DLOAD [index]
    - ALOAD [index]
  - Result
    - Value from local pushed to the top of the stack
- Locals are *this*, method parameters, local variables and other temporary values

Local manipulation

- Storing values in locals
    - ISTORE [index]
    - LSTORE [index]
    - FSTORE [index]
    - DSTORE [index]
    - ASTORE [index]
  - Result
    - Top of the stack is popped and stored in the local

Arithmetic

- Arithmetic opcodes
  - Addition `(x + y)`
    - IADD, LADD, FADD, DADD
  - Subtraction `(x - y)`
    - ISUB, LSUB, FSUB, DSUB
  - Negate `(-x)`
    - INEG, LNEG, FNEG, DNEG
  - In-local integer increment
    - IINC [index], [16-bit value] `(x += val)`

Arithmetic

- Arithmetic opcodes
  - Multiplication `(x * y)`
    - IMUL, LMUL, FMUL, DMUL
  - Division `(x / y)`
    - IDIV, LDIV, FDIV, DDIV
  - Remainder `(x % y)`
    - IREM, LREM, FREM, DREM

Arithmetic

- Bitwise opcodes
  - AND `(x & y)`
    - IAND, LAND
  - OR `(x | y)`
    - IOR, LOR
  - XOR `(x ^ y)`
    - IXOR, LXOR

Arithmetic

- Bitwise opcodes
  - Shift left `(x << y)`
    - ISHL, LSHL
  - Signed shift right `(x >> y)`
    - ISHR, LSHR
  - Unsigned shift right `(x >>> y)`
    - IUSHR, LUSHR

Arithmetic

- Float and Long comparison
  - DCMPL, DCMPG
  - FCMPL, FCMPG
  - LCMP
    - –1 if v1 < v2
    - 0 if v1 == v2
    - +1 if v1 > v2
- Difference between L and G versions
  - L pushes –1 on the stack if either number is NaN
  - G pushes +1 on the stack if either number is NaN

Arithmetic

- Conversion
  - int to long/float/double/byte/char/short
    - I2L, I2F, I2D, I2B, I2C, I2S
  - long to int/float/double
    - L2I, L2F, L2D
  - float to int/long/double
    - F2I, F2L, F2D
  - double to int/long/float
    - D2I, D2L, D2F

Flow control

- Unconditional jump
    - GOTO

- Conditional jumps
    - Object comparison (jumps if condition is met)

IF_ACMPEQ	`if (a1 == a2)`
IF_ACMPNE	`if (a1 != a2)`
IFNONNULL	`if (a  != null)`
IFNULL	`if (a  == null)`

Flow control

- Conditional jumps
  - Integer comparison (jumps if condition is met)

IF_ICMPEQ	**if (i1 == i2)**
IF_ICMPNE	**if (i1 != i2)**
IF_ICMPGE	**if (i1 >= i2)**
IF_ICMPGT	**if (i1 > i2)**
IF_ICMPLE	**if (i1 <= i2)**
IF_ICMPLT	**if (i1 < i2)**

Flow control

- Conditional jumps
  - Integer comparison (jumps if condition is met)

IFEQ	`if (i == 0)`
	`if (bool == false)`
IFNE	`if (i != 0)`
	`if (bool == true)`
IFGE	`if (i >= 0)`
IFGT	`if (i  > 0)`
IFLE	`if (i <= 0)`
IFLT	`if (i  < 0)`

Flow control

- Switch statements
  - LOOKUPSWITCH, TABLESWITCH
- Exiting methods
  - Normal return
    - RETURN, ARETURN, IRETURN, LRETURN, FRETURN, DRETURN
  - Throwing exception
    - ATHROW

Array manipulation

- Array manipulation
  - Getting the size of an array
    - ARRAYLENGTH
  - Setting/Getting elements in an array

CASTORE, CALOAD	char array
BASTORE, BALOAD	byte/boolean array
SASTORE, SALOAD	short array
IASTORE, IALOAD	int array
LASTORE, LALOAD	long array
FASTORE, FALOAD	float array
DASTORE, DALOAD	double array
AASTORE, AALOAD	Object array

Object model

- Method invocations
  - INVOKESTATIC

  - INVOKESPECIAL
  - INVOKEVIRTUAL
  - INVOKEINTERFACE

  - INVOKEDYNAMIC

    - Note: Unlike all other opcodes, method invocations pops a variable number of entries of the stack!

Object model

- Field access
  - GETSTATIC
  - PUTSTATIC

  - GETFIELD
  - PUTFIELD

Object model

- Object casting
  - CHECKCAST [class]

- Object type check
  - INSTANCEOF [class]
    - Pushes 1 on stack if match, 0 otherwise

Object model

- Object creation
  - NEW [class]

- Array creation

MULTIANEWARRAY	Multi-dim Object array
ANEWARRAY	Object array
NEWARRAY	Primitive-type array

Misc opcodes

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

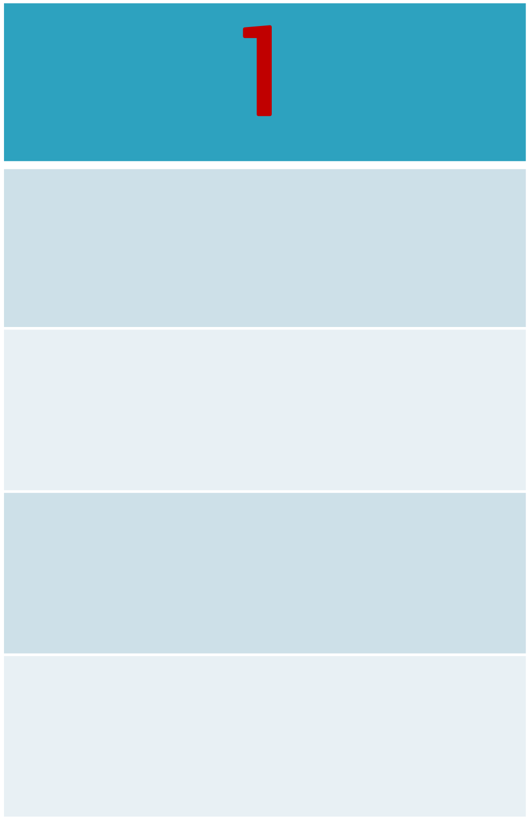
Stack manipulation

- Now, push it
  Ah, push it
  Push it real good

Stack manipulation examples

- SWAP
- POP
- DUP

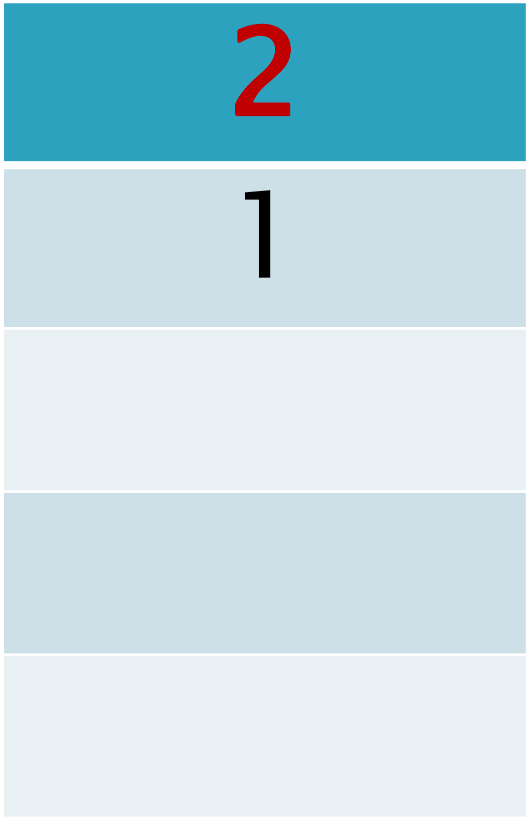
ICONST_1



Stack manipulation examples

- SWAP
- POP
- DUP

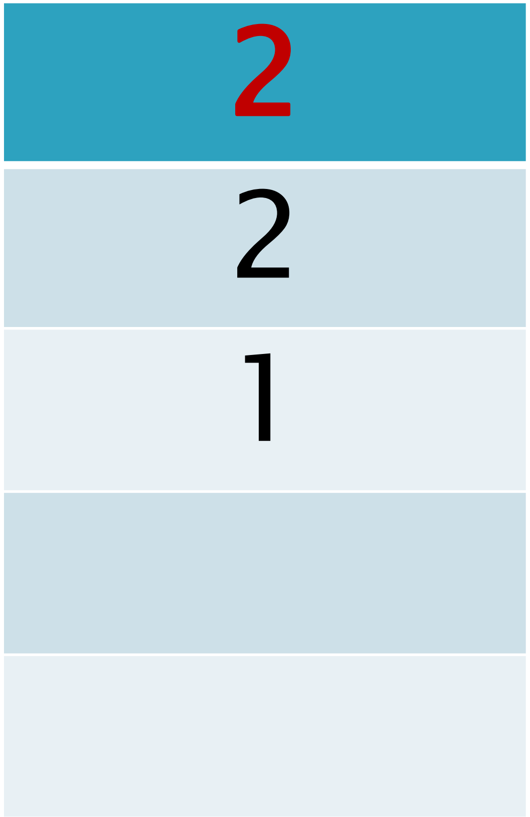
```
ICONST_1
ICONST_2
```



Stack manipulation examples

- SWAP
- POP
- DUP

```
ICONST_1
ICONST_2
DUP
```



2

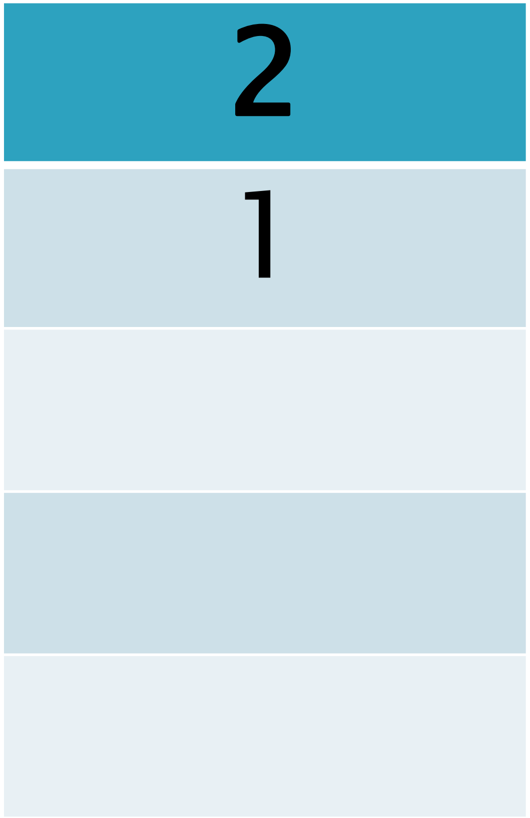
2

1

Stack manipulation examples

- SWAP
- POP
- DUP

```
ICONST_1
ICONST_2
DUP
POP
```



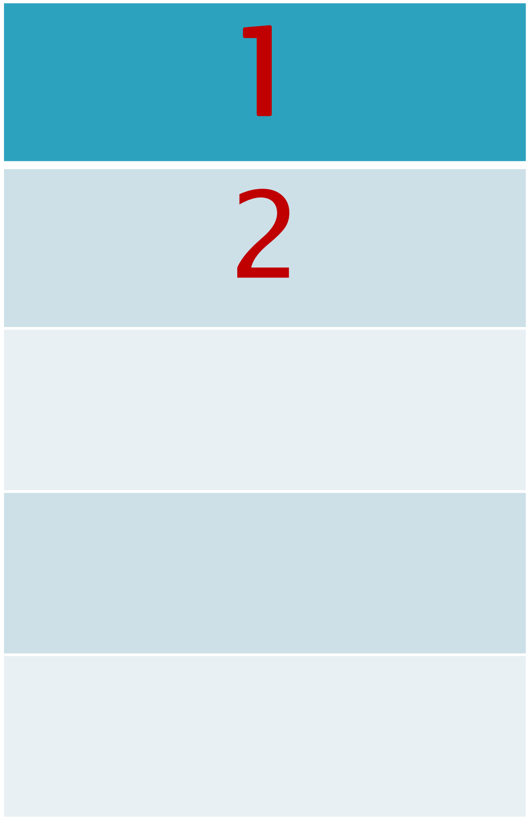
2

1

Stack manipulation examples

- SWAP
- POP
- DUP

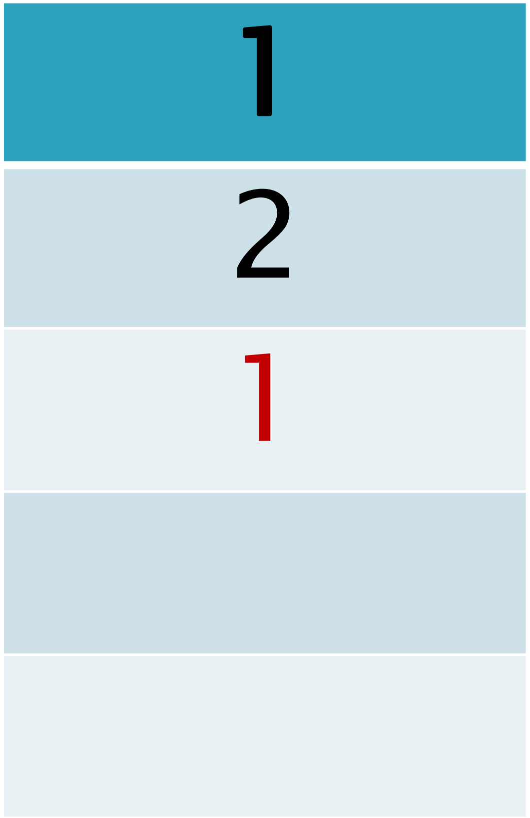
```
ICONST_1
ICONST_2
DUP
POP
SWAP
```



Stack manipulation examples

- SWAP2

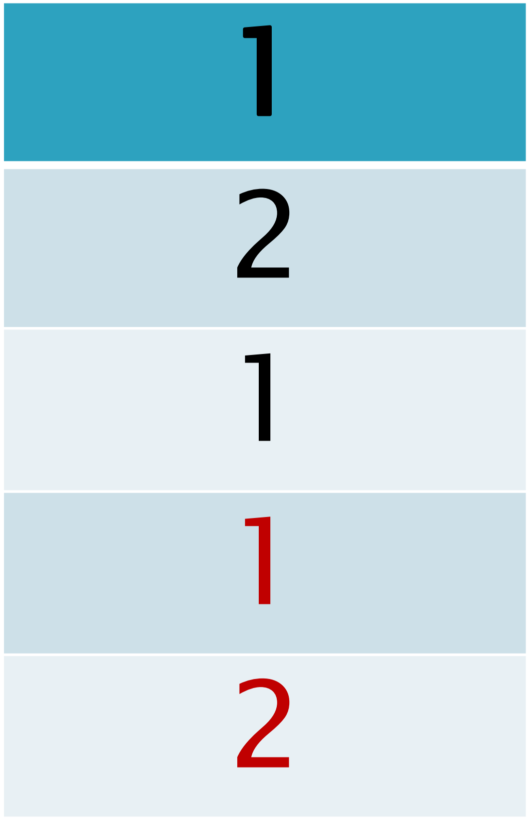
```
ICONST_1
ICONST_2
DUP
POP
SWAP
DUP_X1
```



Stack manipulation examples

- SWAP
- POP
- DUP

```
ICONST_1
ICONST_2
DUP
POP
SWAP
DUP_X1
DUP2_X1
```



1

2

1

1

2

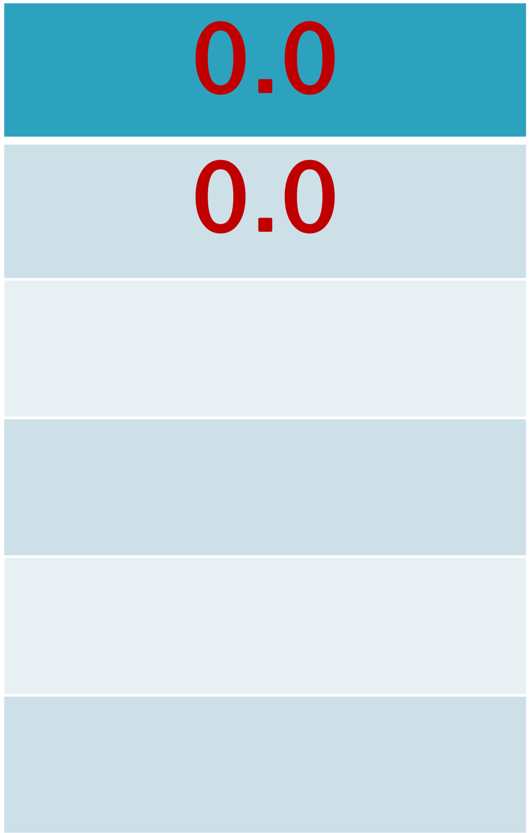
Stack manipulation examples

- How to swap two doubles?
- Double and Long take up two slots
- SWAP won’t work
- SWAP2 doesn’t exist

Stack manipulation examples

- How to swap two doubles?

DCONST_0



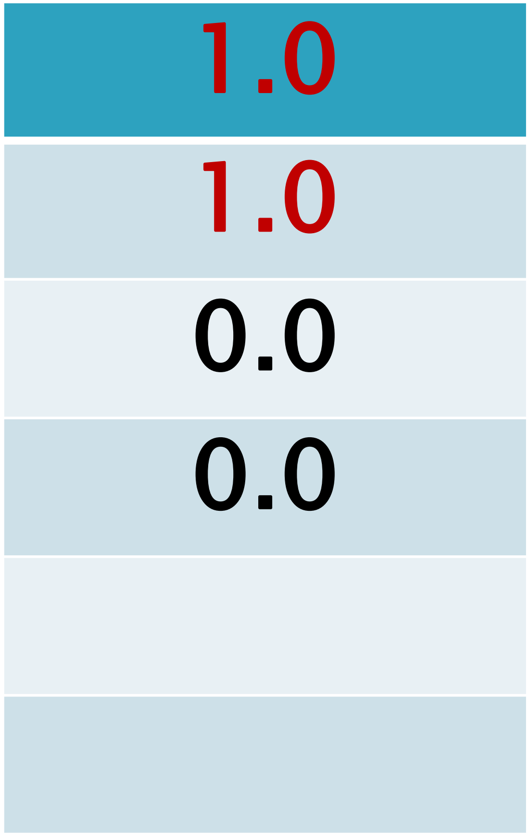
0.0

0.0

Stack manipulation examples

- How to swap two doubles?

DCONST_0
DCONST_1



1.0

1.0

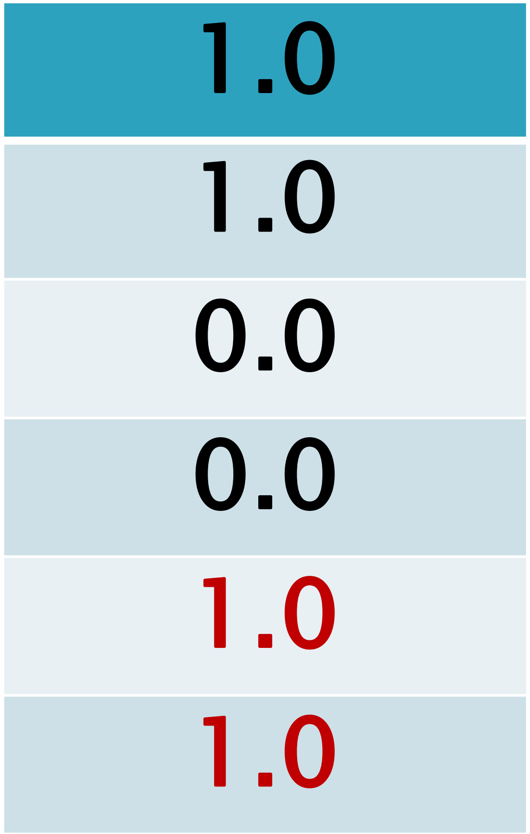
0.0

0.0

Stack manipulation examples

- How to swap two doubles?

```
DCONST_0
DCONST_1
DUP2_X2
```



1.0

1.0

0.0

0.0

1.0

1.0

Stack manipulation examples

- How to swap two doubles?

```
DCONST_0
DCONST_1
DUP2_X2
POP2
```

0.0
0.0
1.0
1.0

Stack manipulation examples

- How to swap two doubles?

```
DCONST_0
DCONST_1
DUP2_X2
POP2
```

0.0
0.0
1.0
1.0

Using locals

- Somebody save me
  I don’t care how you do it
  just stay, stay
  c’mon

Using locals

- Method parameters are stored in locals
  - For instance methods
    - *this* is stored in slot 0
    - First parameter is in slot 1
  - For static methods
    - First parameter is in slot 0

- Double and Long take up two locals

Using locals

- Assume the following code:

-
  ```
  static int sum(int a, int b) {
      return a * b;
  }
  ```

- Resulting bytecode:
  - ILOAD 0
  - ILOAD 1
  - IMUL
  - IRETURN

Using locals

- Assume the following code:

-
  ```
  static double sum(double a, double b) {
      return a * b;
  }
  ```

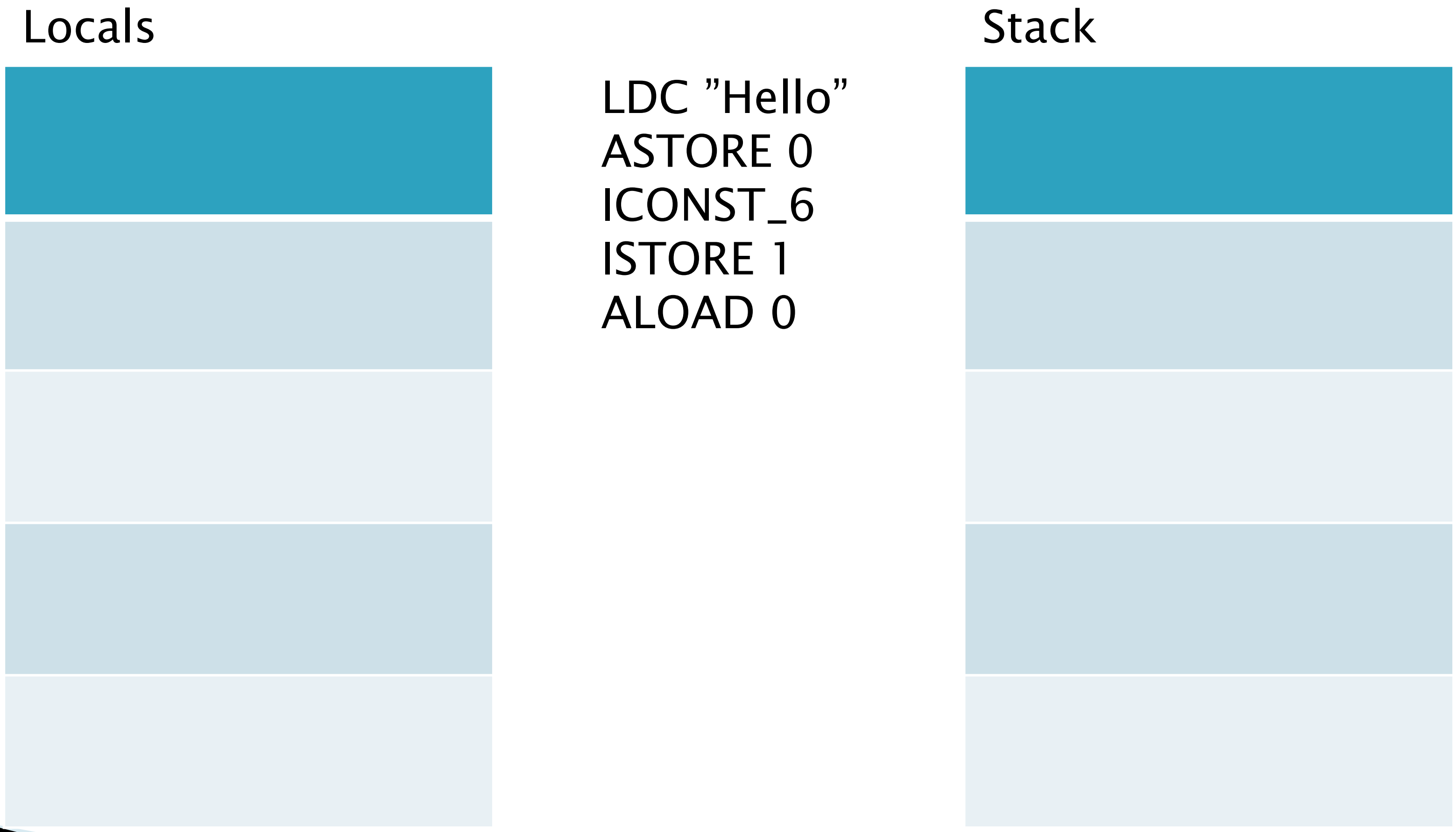
- Resulting bytecode:
  - DLOAD 0
  - DLOAD 2
  - DMUL
  - DRETURN

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 returns

Using locals

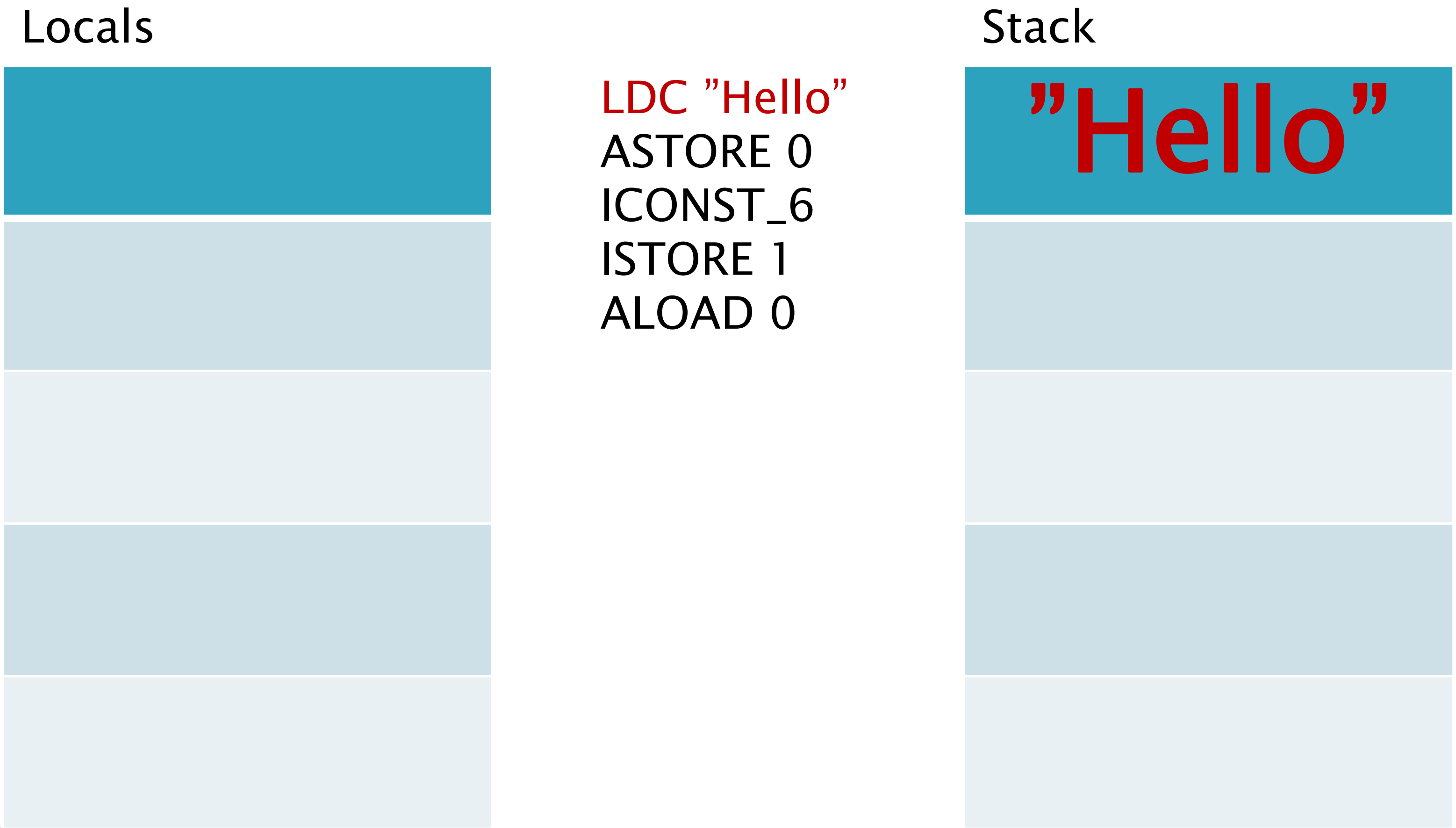


Locals

Stack

```
LDC ”Hello”
ASTORE 0
ICONST_6
ISTORE 1
ALOAD 0
```

Using locals



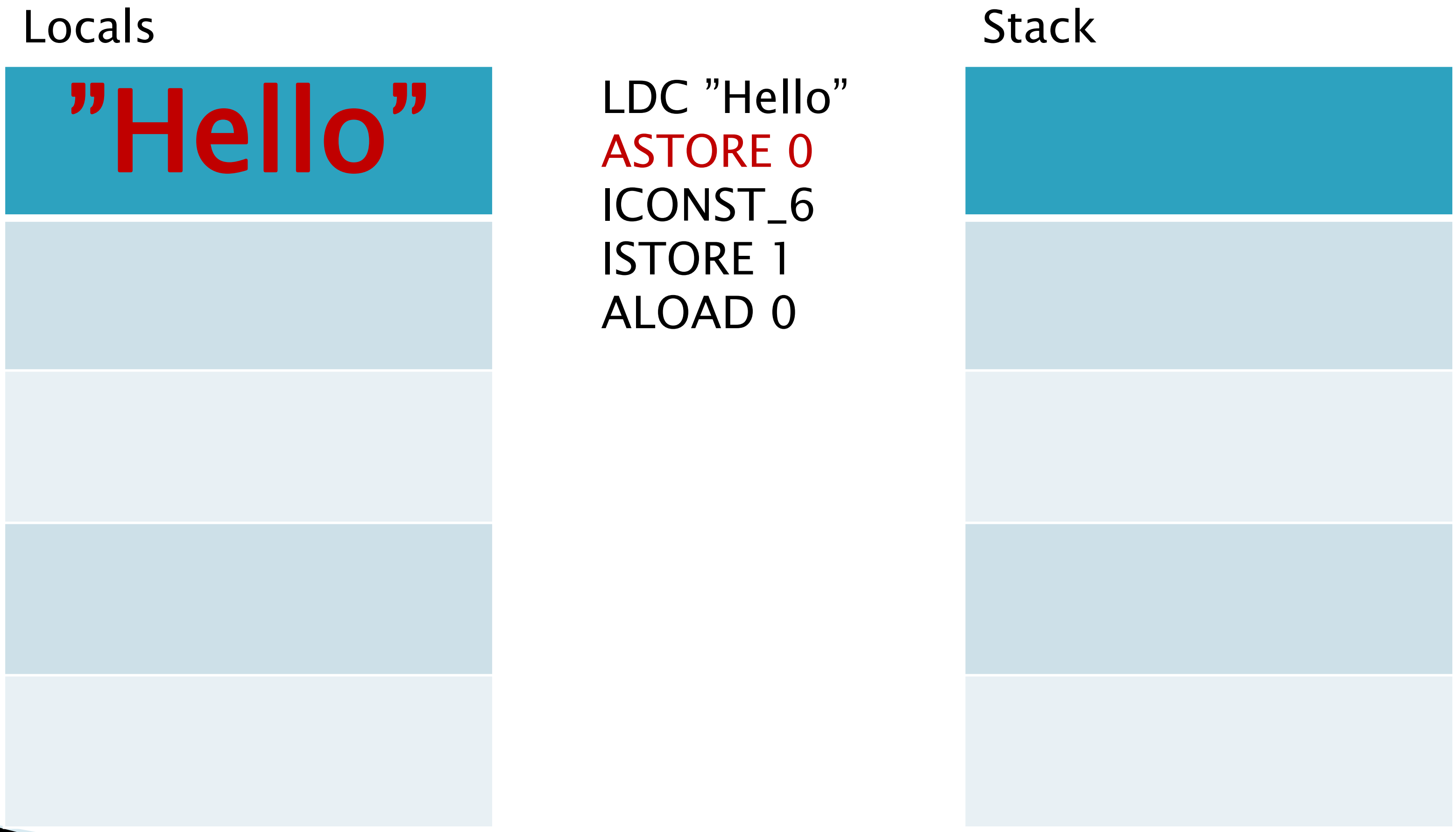
Locals

```
LDC "Hello"
ASTORE 0
ICONST_6
ISTORE 1
ALOAD 0
```

Stack

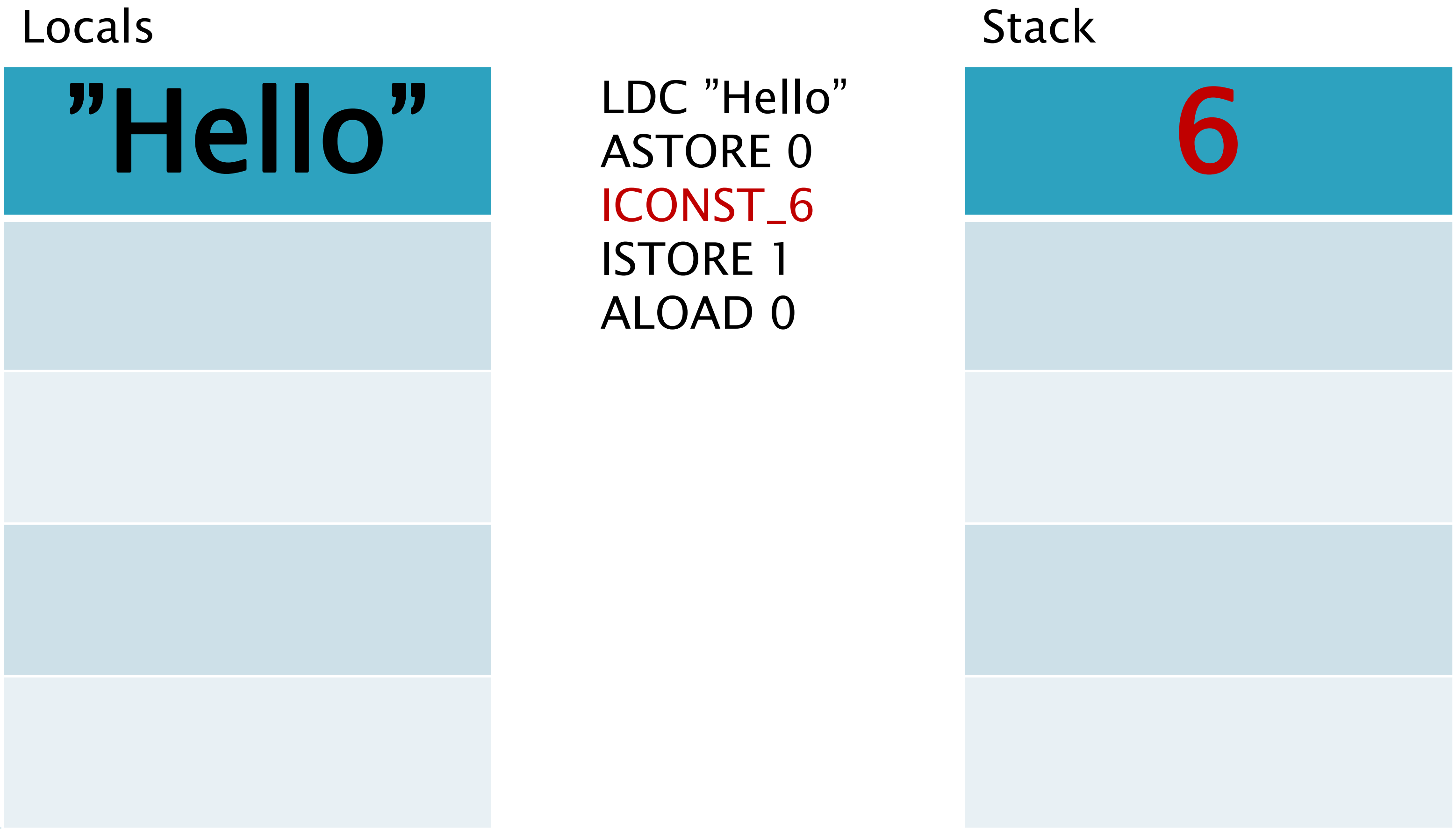
"Hello"

Using locals



```
LDC "Hello"
ASTORE 0
ICONST_6
ISTORE 1
ALOAD 0
```

Using locals



Locals

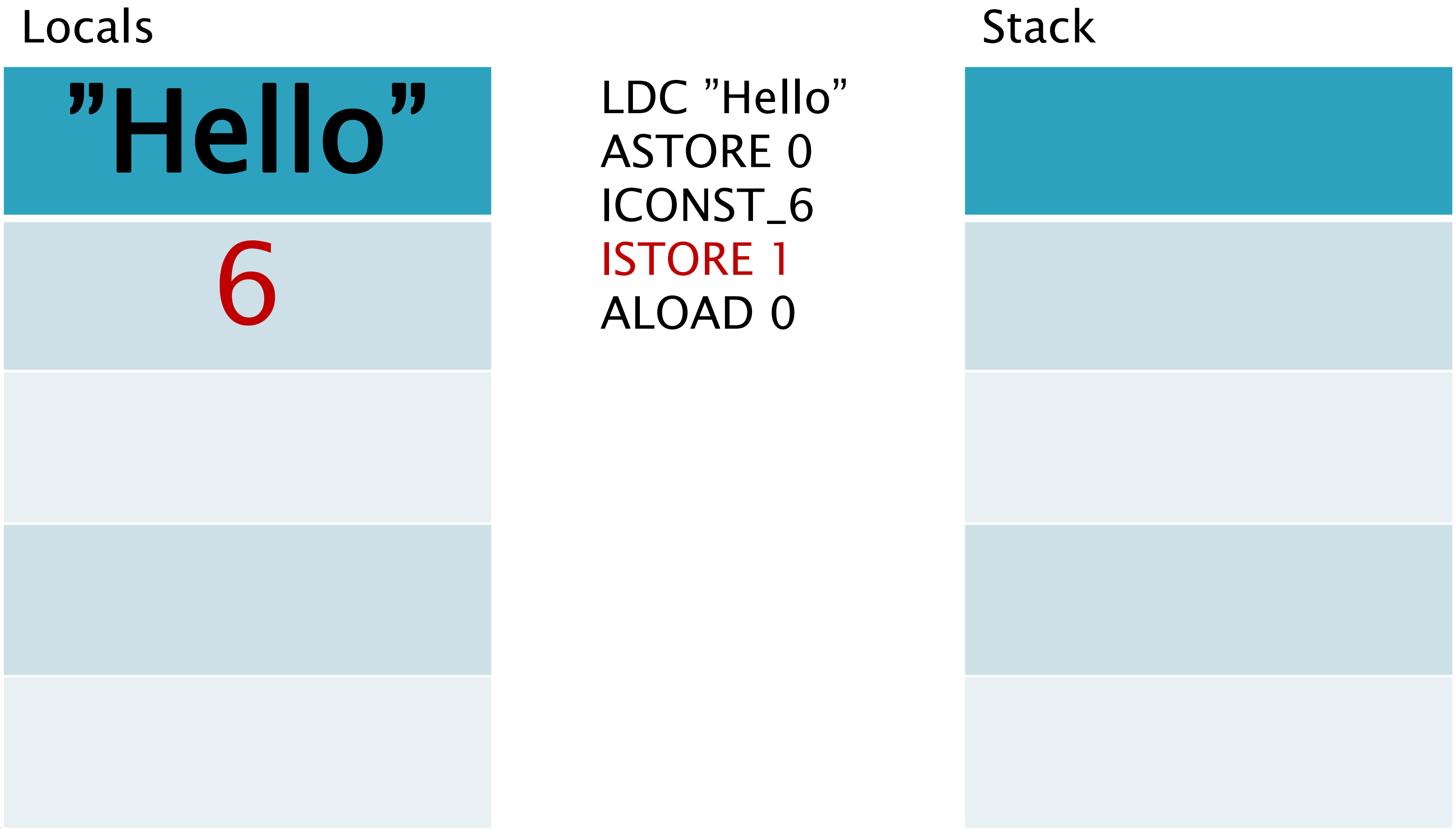
"Hello"

```
LDC "Hello"
ASTORE 0
ICONST_6
ISTORE 1
ALOAD 0
```

Stack

6

Using locals



```
LDC "Hello"
ASTORE 0
ICONST_6
ISTORE 1
ALOAD 0
```

Using locals

Locals
”Hello”
6

```
LDC ”Hello”
ASTORE 0
ICONST_6
ISTORE 1
ALOAD 0
```

Stack
”Hello”

Control flow

- I might as well jump!
  Jump!
  Go ahead and jump!

Control flow

- Simple if statement

```
static int gt(int a, int b) {
   if (a > b)
      return 1;
   else
      return -1;
}
```

Control flow

- Resulting bytecode:

```
ILOAD 0               // push local[0] {a}
ILOAD 1               // push local[1] {b}
IF_ICMPGT :gt         // if local[0] > local[1] goto :gt
ICONST_M1             // push –1
IRETURN               // return value
:gt
ICONST_1              // push +1
IRETURN               // return value
```

Control flow

- While loop and arithmetic example

-
  ```
  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

Method invocation

- Hey, I just met you
  and this is crazy
  but here’s my number
  so call me maybe?

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 the
    “getProperty”, // stack as arguments
    “(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;”
  - ARETURN // return the resulting value

Method invocation

- Virtual method invocation:

```
static int getLength (String str) {
   return str.length();
 }
```

Method invocation

- Resulting bytecode:
  - ALOAD 0 // push local[0] {str}
  - INVOKEVIRTUAL
    ```
        “java/lang/String”,        // invoke length on local[0]
        “length”,                  // pushing the returned int
        “()I”                      // value on to the stack
    ```
  - IRETURN // return the value

Tooling

- Left a good job in the city working for the Man every night and day

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();

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();
    ```

Next time

- Do you know what you got into?
  Can you handle what I’m ‘bout to do?
  ‘cause it’s about to get rough for you,
  I’m here for your entertainment!

Next time

- From AST to bytecode
  - Using ASM to generating bytecode
    - Covering the basics
    - Labels, descriptors, binary name?
      - Why should I care about those?
    - Control flow templates
    - Seeing if it actually works?!
- Why bother with bytecode transformation if not writing a compiler?

Section–subscript credits

- Pink Floyd – Welcome To The Machine
- Queen – Radio Ga Ga
- Salt–n–Pepa – Push It
- Remy Zero – Save Me
- Van Halen – Jump
- Carly Rae Jensen – Call Me Maybe
- Creedence Clearwater Revival – Proud Mary
- Adam Lambert – For Your Entertainment