301AA - Advanced Programming

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

**AP-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 and 3 of the JVM Specification
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 tables\textsc{\textit{witch}} and lookup\textsc{\textit{switch}})
• 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)

<table>
<thead>
<tr>
<th>Assembly code</th>
<th>Binary instruction code layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload_0</td>
<td>26</td>
</tr>
<tr>
<td>iload_1</td>
<td>27</td>
</tr>
<tr>
<td>iload_2</td>
<td>28</td>
</tr>
<tr>
<td>iload_3</td>
<td>29</td>
</tr>
<tr>
<td>iload n</td>
<td>21 n</td>
</tr>
<tr>
<td>wide iload n</td>
<td>196 21 n</td>
</tr>
</tbody>
</table>

Pushes local variable 0 on operand stack
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

<table>
<thead>
<tr>
<th>Type</th>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>iload</td>
<td>integer load</td>
</tr>
<tr>
<td>long</td>
<td>lload</td>
<td>long load</td>
</tr>
<tr>
<td>short</td>
<td>fload</td>
<td>float load</td>
</tr>
<tr>
<td>byte</td>
<td>dload</td>
<td>double load</td>
</tr>
<tr>
<td>char</td>
<td>aload</td>
<td>reference-type load</td>
</tr>
</tbody>
</table>

i  for int
l  for long
s  for short
b  for byte
c  for char
f  for float
d  for double
a  for reference
Instruction-set: accessing arguments and locals in the Local Variable array

Instruction examples:

- `iload_1`  
- `iload_3`  
- `aload_0`  
- `istore_1`  
- `astore_1`  
- `fstore_3`  

- 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.

**Diagram:**

- Args: Indexes 0 .. #args - 1
- Locals: Indexes #args .. #args + #locals - 1
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>
<thead>
<tr>
<th>opcode</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>char</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tipush</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tconst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tstore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tinc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tastore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tsub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tdiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tneg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tushr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Txor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmpf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmpg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if_TcmpOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treturn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.11.1-A. Type support in the Java Virtual Machine instruction set
### Specification of an instruction: iadd

#### iadd

<table>
<thead>
<tr>
<th>Operation</th>
<th>Add int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>iadd</td>
</tr>
<tr>
<td>Forms</td>
<td>iadd = 96 (0x60)</td>
</tr>
<tr>
<td>Operand Stack</td>
<td>..., value1, value2 →</td>
</tr>
<tr>
<td>Stack</td>
<td>..., result</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
</tbody>
</table>
## Computational Types

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

<table>
<thead>
<tr>
<th>Actual type</th>
<th>Computational type</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>1</td>
</tr>
<tr>
<td>reference</td>
<td>reference</td>
<td>1</td>
</tr>
<tr>
<td>returnAddress</td>
<td>returnAddress</td>
<td>1</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>2</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>2</td>
</tr>
</tbody>
</table>
Compiling Constants, Local Variables, and Control Constructs

- Sample Code

```java
tvoid spin() {
    int i;
    for (i = 0; i < 100; i++) {
        // Loop body is empty
    }
}
```

- Can compile to

```
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 less than (i < 100)
14  return    // Return void when done
```

- Pushing constants on the operand stacks
- Incrementing local variable, comparing
int vs. double: lack of opcodes for double requires longer bytecode

• Sample Code

```java
void dspin() {
    double i;
    for (i = 0.0; i < 100.0; i++) {
        ; // Loop body is empty
    }
}
```

• Can compile to

```
0 dconst_0
1 dstore_1
2 goto 9
5 dload_1
6 dconst_1
7 dadd
8 dstore_1
9 dload_1
10 ldc2_w #4
13 dcmpg
14 iflt 5
17 return
```

// Push double constant 0.0
// Store into local variables 1 and 2
// First time through don't increment
// Push local variables 1 and 2
// Push double constant 1.0
// Add; there is no dinc instruction
// Store result in local variables 1 and 2
// Push local variables 1 and 2
// Push double constant 100.0
// There is no if_dcmplt instruction
// Compare and loop if less than(i < 100.0)
// Return void when done
Accessing literals in the Constant Pool

• Sample Code

```java
void useManyNumeric() {
    int i = 100;
    int j = 1000000;
    long l1 = 1;
    long l2 = 0xffffffff;
    double d = 2.2;
    ...do some calculations...
}
```

• Can compile to

```
0 bipush 100  // Push small int constant with bipush
2 istore_1
3 ldc #1       // Push large int (1000000) with ldc
5 istore_2
6 lconst_1     // A tiny long value uses fast lconst_1
7 lstore_3
8 ldc2_w #6    // Push long 0xffffffff (that is, int -1)
11 lstore 5
13 ldc2_w #8   // Any long can be pushed with ldc2_w
16 dstore 7
...            // Push double constant 2.200000
...do those calculations...
```
Parameter passing: Receiving Arguments

• Sample Code

```java
int addTwo(int i, int j) {
    return i + j;
}
```

• Can compile to

```
0 iload_1
1 iload_2
2 iadd
3 ireturn
```

// Push value of local variable 1 (i)
// Push value of local variable 2 (j)
// Add; leave int result on operand stack
// Return int result

• Local variable 0 used for this in instance methods

• Sample Code

```java
static int addTwo(int i, int j) {
    return i + j;
}
```

• Can compile to

```
0 iload_0
1 iload_1
2 iadd
3 ireturn
```
Invoking Methods

• Sample Code

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

• Can compile to

```
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` causes the allocation of a new frame, pops the arguments from the stack into the local variables of the callee (putting `this` in 0), and passes the control to it by changing the `ps`

• A resolution of the symbolic link is performed

• `ireturn` pushes the top of the current stack to the stack of the caller, and passes the control to it. Similarly for `dreturn`, ...

• `return` just passes the control to the caller
Other kinds of method invocation

- **invokestatic** – for calling methods with “static” modifiers
  - *this* is not passed, arguments are copied to local vars from 0
- **invokespecial** – for calling constructors, which are not dynamically dispatched, *private methods* or *superclass methods*.
  - *this* is always passed
- **invokeinterface** – same as **invokevirtual**, but used when the called method is declared in an interface (requires a different kind of method lookup)
- **invokedynamic** – introduced in Java SE 7 to support dynamic typing
  - We shall discuss it when presenting lambdas
Working with objects

• Sample Code

```java
Object create() {
    return new Object();
}
```

• Can compile to

```
0 new #1     // Class java.lang.Object
3 dup
4 invokespecial #4 // Method java.lang.Object.<init>()V
7 areturn
```

• Objects are manipulated essentially like data of primitive types, but through **references** using the corresponding instructions (e.g. **areturn**)
Accessing fields (instance variables)

• Sample Code

```java
Method void setIt(int)
0  aload_0
1  iload_1
2  putfield #4 // Field Example.i I
5  return

Method int getIt()
0  aload_0
1  getfield #4 // Field Example.i I
4  ireturn
```

• Can compile to

```java
void setIt(int value) {
    i = value;
}
int getIt() {
    return i;
}
```

• Requires resolution of the symbolic reference in the constant pool
• Computes the offset of the field in the class, and uses it to access the field in this
• Similar for static variables, using putstatic and getstatic
Using Arrays

- Sample Code
- Can compile to

```java
void createBuffer() {
    int buffer[];
    int bufsz = 100;
    int value = 12;
    buffer = new int[bufsz];
    buffer[10] = value;
    value = buffer[11];
}
```

```
0 bipush 100
2 istore_2
3 bipush 12
5 istore_3
6 iload_2
7 newarray int
9 astore_1
10 aload_1
11 bipush 10
13 iload_3
14 iastore
15 aload_1
16 bipush 11
18 iaload
19 istore_3
20 return
```

// Push int constant 100 (bufsz)
// Store bufsz in local variable 2
// Push int constant 12 (value)
// Store value in local variable 3
// Push bufsz and...
// ... create new int array of that length
// Store new array in buffer
// Push buffer
// Push int constant 10
// Push value
// Store value at buffer[10]
// Push buffer
// Push int constant 11
// Push value at buffer[11]...
// ...and store it in value
Compiling switches (1)

- Sample Code
- Can compile to

```java
int chooseNear(int i) {
    switch (i) {
        case 0:  return 0;
        case 1:  return 1;
        case 2:  return 2;
        default: return -1;
    }
}
```

```assembly
0 iload_1
1 tableswitch 0 to 2:
    0: 28
    1: 30
    2: 32
default: 34
28  iconst_0
29  ireturn
30  iconst_1
31  ireturn
32  iconst_2
33  ireturn
34  iconst_m1
35  ireturn
```
### tableswitch

**Operation**  
Access jump table by index and jump

**Format**

```
<table>
<thead>
<tr>
<th>tableswitch</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;0-3 byte pad&gt;</td>
</tr>
<tr>
<td>defaultbyte1</td>
</tr>
<tr>
<td>defaultbyte2</td>
</tr>
<tr>
<td>defaultbyte3</td>
</tr>
<tr>
<td>defaultbyte4</td>
</tr>
<tr>
<td>lowbyte1</td>
</tr>
<tr>
<td>lowbyte2</td>
</tr>
<tr>
<td>lowbyte3</td>
</tr>
<tr>
<td>lowbyte4</td>
</tr>
<tr>
<td>highbyte1</td>
</tr>
<tr>
<td>highbyte2</td>
</tr>
<tr>
<td>highbyte3</td>
</tr>
<tr>
<td>highbyte4</td>
</tr>
<tr>
<td>jump offsets...</td>
</tr>
</tbody>
</table>
```

**Forms**  
`tableswitch = 170 (0xaa)`

**Operand**  
...`

**Stack**  
...`

### tableswitch

A `tableswitch` is a variable-length instruction. Immediately after the `tableswitch` opcode, between zero and three bytes must act as padding, such that `defaultbyte1` begins at an address that is a multiple of four bytes from the start of the current method (the opcode of its first instruction). Immediately after the padding are bytes constituting three signed 32-bit values: `default`, `low`, and `high`. Immediately following are bytes constituting a series of `high - low + 1` signed 32-bit offsets. The value `low` must be less than or equal to `high`. The `high - low + 1` signed 32-bit offsets are treated as a 0-based jump table. Each of these signed 32-bit values is constructed as `(byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4`.

The `index` must be of type `int` and is popped from the operand stack. If `index` is less than `low` or `index` is greater than `high`, then a target address is calculated by adding `default` to the address of the opcode of this `tableswitch` instruction. Otherwise, the offset at position `index - low` of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this `tableswitch` instruction. Execution then continues at the target address.

The target address that can be calculated from each jump table offset, as well as the one that can be calculated from `default`, must be the address of an opcode of an instruction within the method that contains this `tableswitch` instruction.
Compiling switches (2)

- Sample Code

```java
int chooseFar(int i) {
    switch (i) {
        case -100: return -1;
        case 0:    return  0;
        case 100:  return  1;
        default:   return -1;
    }
}
```

```java
0 iload_1
1 lookupswitch 3:
    -100: 36
    0:  38
    100: 40
    default: 42
36 iconst_m1
37 ireturn
38 iconst_0
39 ireturn
40 iconst_1
41 ireturn
42 iconst_m1
43 ireturn
```

- `lookupswitch` is used when the cases of the `switch` are sparse
- Each case is a pair `<value: address>`, instead of an offset in the table of addresses
- Cases are sorted, so binary search can be used

Note that only switches on `int` are supported: for other types conversions (`char, byte, short`) or non-trivial translations (`String, using hashcode`) are needed
## Operand stack manipulation

### Sample Code

```java
public long nextIndex() {
    return index++;
}

private long index = 0;
```

### Can compile to

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>aload_0</code></td>
</tr>
<tr>
<td>1</td>
<td><code>dup</code></td>
</tr>
<tr>
<td>2</td>
<td><code>getfield #4</code></td>
</tr>
<tr>
<td>5</td>
<td><code>dup2_x1</code></td>
</tr>
<tr>
<td>6</td>
<td><code>lconst_1</code></td>
</tr>
<tr>
<td>7</td>
<td><code>ladd</code></td>
</tr>
<tr>
<td>8</td>
<td><code>putfield #4</code></td>
</tr>
<tr>
<td>11</td>
<td><code>lreturn</code></td>
</tr>
</tbody>
</table>

- Push this
- Make a copy of it
- One of the copies of this is consumed pushing long field index, above the original this
- The long on top of the operand stack is inserted into the operand stack below the original this
- Push long constant 1
- The index value is incremented...
- ...and the result stored in the field
- The original value of index is on top of the operand stack, ready to be returned
### dup2_x1

**Operation**
Duplicate the top one or two operand stack values and insert two or three values down.

**Format**
```
  dup2_x1
```

**Forms**

- **Form 1:**
  
  ..., value3, value2, value1 →
  
  ..., value2, value1, value3, value2, value1

  where value1, value2, and value3 are all values of a category 1 computational type (§2.11.1).

- **Form 2:**
  
  ..., value2, value1 →
  
  ..., value1, value2, value1

  where value1 is a value of a category 2 computational type and value2 is a value of a category 1 computational type (§2.11.1).

**Description**
Duplicate the top one or two values on the operand stack and insert the duplicated values, in the original order, one value beneath the original value or values in the operand stack.
Throwing Exceptions

• Sample Code

```java
void cantBeZero(int i) throws TestExc {
    if (i == 0) {
        throw new TestExc();
    }
}
```

• Can compile to

```
0 iload_1       // Push argument 1 (i)
1 ifne 12       // If i==0, allocate instance and throw
4 new #1        // Create instance of TestExc
7 dup           // One reference goes to its constructor
8 invokespecial #7  // Method TestExc.<init>()V
11 athrow       // Second reference is thrown
12 return       // Never get here if we threw TestExc
```

• `athrow` looks in the method for a `catch` block for the thrown exception using the exception table

• If it exists, stack is cleared and control passed to the first instruction

• Otherwise the current frame is discarded and the same exception is thrown on the caller

• If no method catches the exception, the thread is aborted
try-catch

- Sample Code
- Can compile to

```java
void catchOne() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
    }
}
```

```
0  aload_0
1  invokevirtual  #6
4  return
5  astore_1
6  aload_0
7  aload_1
8  invokevirtual  #5
11 return
```

- Compiles a catch clause like another method
- The table records boundaries of try and is used by athrow to dispatch the control

**Exception table:**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>5</td>
<td>Class TestExc</td>
</tr>
</tbody>
</table>

- Compilation of finally more tricky
Other Instructions

• Handling synchronization: monitorenter, monitorexit
• verifying instances: instanceof
• checking a cast operation: checkcast
• No operation: nop
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
Coding: What's the difference?

```java
System.out.println("Result = "+i);
System.out.print("Result = ");
System.out.println(i);
```
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:(Ljava/lang/String;)Ljava/lang/StringBuffer;
iload_1
invokevirtual #8; // StringBuffer.append:(I)Ljava/lang/StringBuffer;
invokevirtual #9; // StringBuffer.toString():Ljava/lang/String;
invokevirtual #10; // PrintStream.println:(Ljava/lang/String;)V

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:(Ljava/lang/String;)V
gsetatic    #3; // Field System.out:LPrintStream;
iload_1
invokevirtual #6; // Method PrintStream.println:(I)V