CS263: Runtime Systems
Lecture: High-level language virtual machines

Part 1 of 4

Chandra Krintz
UCSB Computer Science Department
Child c = new Child();
Parent p = getSomeRandomCrazyObject();

c = p;

p = c;
Child c = new Child();
Parent p = getSomeRandomCrazyObject();

//the following results in a compile time (javac) error  
  b/c Parent objects in general do not have Child parts 
  which is assumed for any Child type variable

c = p;  error: incompatible types: Parent cannot be converted to Child

//OK (no error) b/c c must have "Parent parts"
p = c;
Parent p = getObject();

val = p.field1;
//what is in val?

class Parent {
    ...
    int field1 = 10;
    int field2 = 20;
}
class Child
    extends Parent {
        ...
        int field1 = 5;
        int field3 = 4;
    }
Parent p = getObject();

val = p.field1;
//what is in val? 10
//what if getObject returns a Child type object?

class Parent {
    ...
    int field1 = 10;
    int field2 = 20;
}
class Child
    extends Parent {
    ...
    int field1 = 5;
    int field3 = 4;
}
Parent p = getObject();

val = p.field1;
//what is in val? 10
//what if getObject returns a Child type object? 10

//what happens here? 
p = new Parent();
val = p.field3;
Parent p = getObject();

val = p.field1;
//what is in val? 10

//what if getObject returns a Child type object? 10

//what happens here?
p = new Parent();
val = p.field3; //ERROR, no field3 in type Parent or
//supertype of Parent

```java
class Parent {
    ...
    int field1 = 10;
    int field2 = 20;
}
class Child
    extends Parent {
    ...
    int field1 = 5;
    int field3 = 4;
}
```
// and here?
Child c = new Child();
val = c.field2;

```java
class Parent {
    ...
    int field1 = 10;
    int field2 = 20;
}

class Child extends Parent {
    ...
    int field1 = 5;
    int field3 = 4;
}
```
//and here?
Child c = new Child();
val = c.field2;  // 20

//how do I get to Parent's field1 from the child?
//and here?
Child c = new Child();
val = c.field2;

//how do I get to Parent's field2 from the child?
val = ((Parent)c).field1;

//how do I access f1 and f2?

class Parent {
    ...
    int field1 = 10;
    int field2 = 20;
    static int f1 = 4;
    static int f2 = 7;
}
class Child extends Parent {
    ...
    int field1 = 5;
    int field3 = 4;
    static int f1 = 2;
}
//and here?
Child c = new Child();
val = c.field2;

//how do I get to Parent's field2 from the child?
val = ((Parent)c).field1;

//how do I access f1 and f2?
Parent.f1 = 0;
Child.f1 = -1; //Child’s
Child.f2 = 100; //Parent’s → same as Parent.f2

```java
class Parent {
    ... 
    int field1 = 10;
    int field2 = 20;
    static int f1 = 4;
    static int f2 = 7;
}
class Child extends Parent {
    ... 
    int field1 = 5;
    int field3 = 4;
    static int f1 = 2;
}
```
Parent p = new Child();
//static methods are just like static fields:
Parent.foo();
Child.foo();

//instance method can use static dispatch (C++, C#)
p.bar()
((Parent)p).bar();
((Child)p).bar();

class Parent {
    ...
    static void foo() {...}
    int bar() {...}
}
class Child
    static void foo() {...}
    int bar() {...}
Parent p = new Child();
//static methods are just like static fields:
Parent.foo(); //same as p.foo() if type(p) == Parent
Child.foo(); //same as p.foo() if type(p) == Child

//instance method can use static dispatch (C++, C#)
p.bar() //Parent's bar
((Parent)p).bar(); //Parent's bar
((Child)p).bar(); //Child's bar
Parent c = new Child();
//instance method can use **dynamic** dispatch (Java, C++ and virtual, C# and virtual key word, all dynamically typed languages)

c.foo();
c = new Parent();
c.bar();
((Child)c).foo();
c = new Child();
((Parent)c).bar();
Parent c = new Child();
//instance method can use **dynamic** dispatch (Java, C++ and virtual, C# and virtual key word, all dynamically typed languages)
c.foo(); //Child's foo
c = new Parent();
c.bar(); //Parent's bar
((Child)c).foo(); //dynamic check then Parent's foo
c = new Child()
((Parent)c).bar(); //Child's bar()
Questions?
Portable, Mobile, OO Execution Model

• Execution model embodied by recent PL Implementations
  – Java, all .Net languages, scripting languages
  – Object-oriented
    • Encapsulation and data hiding, inheritance, polymorphism
    • Static methods – static dispatch; virtual methods – dynamic dispatch
      – When same name is used (polymorphism): hidden vs overridden
    • Static fields and instance fields
      – When same name is used (polymorphism): hidden
Portable, Mobile, OO Execution Model

- Execution model embodied by recent PL Implementations
  - Java, all .Net languages, scripting languages
  - Object-oriented
    - Encapsulation and data hiding, inheritance, polymorphism
    - Static methods – static dispatch; virtual methods – dynamic dispatch
      - When same name is used (polymorphism): hidden vs overridden
    - Static fields and instance fields
      - When same name is used (polymorphism): hidden
  - Program is portable and assumed to be mobile
    - **Architecture-independent representation of code and data**
    - Incrementally loaded (transfer unit = class, assembly, archive (jar))
      - Data translated from file/class to VM’s internal representation
        » Class model and object model
    - Code is translated to native code interleaved with execution
      - Instruction level (interpretation), method level (JIT compilation),
        or path level (trace compilation)

Priciple of Laziness!
Java Classfiles

- Architecture-independent
- Format called bytecode
- Dynamically loaded / executed by a Java Virtual Machine

File.java

```
class cls1 {...}
class cls2 {...}
```

source compiler (javac File.java)

Translation from bytecode to native machine code

Java Virtual Machine

- executable code

Runtime environment
- program loading/verif.
- memory management
- thread/synchronization
- optimization

cls1.class

```
...
Java bytecode...
```

cls2.class

```
...
Java bytecode...
```
A Java Program (Compiled Java Source)

- Class files (generated from source using a source compiler)
- Each class file is in a binary bytecode format
- Each class file contains – in a fixed, well-defined format
  - Symbolic information (Metadata in MSIL/.Net binary files)
    - All of the necessary data to execute the class file
  - Instructions (code) in the JVM ISA format

```
Java Class File

Symbolic info

method code

... 

method code
```
A Java Program (Compiled Java Source)

- Class files (generated from source using a source compiler)
- Each class file is in a binary bytecode format
- Each class file contains – in **a fixed, well-defined format**
  - Symbolic information (Metadata in MSIL/.Net binary files)
    - All of the necessary data to execute the class file
  - Instructions (code) in the JVM ISA format

Instead of repeating all of the symbol names throughout, we store all of the symbols in a table (**the constant pool**) and then have everything that uses the symbol (code, other data structures) use a reference into the table instead

**Types (packages, classes, primitives), names, and also method and field data structures**
<table>
<thead>
<tr>
<th>Java Class File</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic number</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td></td>
</tr>
<tr>
<td>Constant pool</td>
<td></td>
</tr>
<tr>
<td>Access flags</td>
<td></td>
</tr>
<tr>
<td>This class</td>
<td></td>
</tr>
<tr>
<td>Super class</td>
<td></td>
</tr>
<tr>
<td>Interfaces</td>
<td></td>
</tr>
<tr>
<td>Fields</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
</tr>
</tbody>
</table>
The Java Class File Format (Symbolic Info)

Java Class File
- Magic number
- Version
- Constant pool
- Access flags
- This class
- Super class
- Interfaces
- Fields
- Methods
- Attributes

Constant pool - an array
- String: “Hello”
- Method: “C.M(int)V”
- Field: “int F”

Fields
- Flags Name Sig Attributes
  - [pub] “x” int
  - [priv] “y” C

Methods
- Flags Name Sig Attributes
  - [pub] “q” ()int code
  - exceptions

Code
- maxstack=5, maxlocals=8
- Code[] = {push, add, store, ldc}
- ExceptionTable[] = { ... }
- Attributes
The Java Class File Format (Symbolic Info)

Java Class File
- Magic number
- Version
- Constant pool
- Access flags
- This class
- Super class
- Interfaces
- Fields
- Methods
- Attributes

Constant pool - an array
- String: "Hello"
- Method: "C.M(int)V"
- Field: "int F"

Fields
- Flags | Name | Sig | Attributes
- [pub] "x" int
- [priv] "y" C

Methods
- Flags | Name | Sig | Attributes
- [pub] "q" ()int code

Attributes

Code
- maxstack=5, maxlocals=8
- Code[] = {push, add, store, ldc}
- ExceptionTable[] = { ... }
- Attributes

References the constant pool
A Java Program  (Compiled Java Source)

• Class files  (generated from source using a source compiler)
• Each class file is in a binary bytecode format
• Each class file contains – in **a fixed, well-defined format**
  – Symbolic information (Metadata in MSIL/.Net binary files)
    • All of the necessary data to execute the class file
  – Instructions (code) in the JVM ISA format

When a class file is read into memory by the runtime, it is stored in a representation internal to the runtime. The runtime keeps a global table which Refers to all these structures, and a map that records which indexes hold which structure. E.g. all internal class representations, static fields, and static methods, are stored in this global table (sometimes called the statics table).
Java Classfiles

- Architecture-independent
- Format called bytecode
- Dynamically loaded / executed by a Java Virtual Machine

File.java

```java
class cls1 {...}
class cls2 {...}
```

source compiler (javac File.java)

• architecture-independent
• architecture-dependent

Java Virtual Machine

Translation from bytecode to native machine code

cls1.class

... Java bytecode...

cls2.class

... Java bytecode...

Runtime environment
- program loading/verif.
- memory management
- thread/synchronization
- optimization

Executable code

Translation from bytecode to native machine code
Java Bytecode Instructions

- Load/store
- Arithmetic
- Type conversions (int to long, etc)
- Object creation (objects/arrays/multi-dim arrays)
- Object manipulation (get/put fields, check properties)
- Operand stack manipulation
- Control transfer (conditionals/unconditionals)
- Method invocation and return
- Exception and finally implementation
- Synchronization

- \[ \text{maxstack}=5, \text{maxlocals}=8 \]

- \[ \text{Code[]} = \{ \text{push}, \text{add}, \text{store}, \text{ldc} \} \]

- \[ \text{ExceptionTable[]} = \{ \ldots \} \]

- \[ \text{Attributes} \]

- \[ \text{http://docs.oracle.com/javase/specs/jvms/se7/html/index.html} \]

- \[ \text{.../html/jvms-4.html#jvms-4.10.1.4} \]
Java Bytecode Instructions

- Load/store
- Arithmetic
- Type conversions (int to long, etc)
- Object creation (objects/arrays/multi-dim arrays)
- Object manipulation (get/put fields, check properties)
- **Operand stack manipulation**
- Control transfer (conditionals/unconditionals)
- Method invocation and return
- Exception and finally implementation
- Synchronization

- .../html/jvms-4.html#jvms-4.10.1.4

```java
maxstack=5, maxlocals=8
Code[] = {push, add, store, ldc}
ExceptionTable[] = { ... }
Attributes
```
Operand Stack

- Do not confuse this with the user/runtime/system stack!
  - User stack: manages local variable scope for functions/methods at runtime
  - **Operand** stack: holds **operands** so that instructions can be executed
  - Both are last-in-first-out (LIFO) queue data structures

- Most modern VM languages execute a stream of instructions using an operand stack
  - Requires that the virtual instruction set architecture (ISA) implements a “stack machine”
Stack-based ISA

- Operands are implicit - on the top of stack (tos)
- Compiler simplification (no register allocation)
- Compact instruction encoding

\[ C = A + B \]

```
push A  //push A on tos
push B  //push B on tos
add     //add the to elements at the tos
pop C   //pop the tos and put the value in C
```

**Operand Stack**

```
tos → A → B → (A+B) →
```

Used in hardware in old HP calculators
Use in most modern language runtimes to enable program portability
JavaCard implements Java bytecode ISA in hardware; as does Azul Vega processor
The Alternative: Register-based ISA

- Operands are explicit (i.e., specified *in* the instruction)
- **Register-register** (aka load-store)
  - Memory accesses via load & store instructions only

  \[
  C = A + B \quad \text{(RISC)}
  \]
  
  ```
  load R1, A  // load memory at &A into reg1
  load R2, B  // load memory at &B into reg2
  add R3, R1, R2  // add reg1+reg2 and store result in reg3
  store C, R3  // store value in reg1 into memory at &C
  ```

- **Register-memory**
  - Operands can be taken from registers or memory directly

  \[
  C = A + B \quad \text{(CISC)}
  \]
  
  ```
  add C, A, B  // load memory from each address
               // perform operation
               // store result in memory
  ```
Java Classfiles

- Architecture-independent
- Format called bytecode
- Dynamically loaded / executed by a Java Virtual Machine

File.java

class cls1 {...}
class cls2 {...}

source compiler (javac File.java)

Java Virtual Machine

cls1.class

... Java bytecode...

Translation from bytecode to native machine code

cls2.class

runtime environment
- program loading/verif.
- memory management
- thread/synchronization
- optimization

Translation from bytecode to native machine code

executable code

- architecture-independent
- architecture-dependent
Java Bytecode Instructions

• Can manipulate data (operands) in following formats: **typed instrs**
  instr. starts with
  – Integers (32-bits) \(i\)
  – Longs (64-bits) \(l\)
  – Floats (32-bits) \(f\)
  – Doubles (64-bits) \(d\)
  – Shorts (16-bits) \(s\)
  – Bytes (8-bits) \(b\)
  – Object references (32-bits) \(a\)
  – Char (16-bits) \(c\)
  – Arrays

• Opcode + data - variable length instructions
  – Opcode: 1-byte wide (256 possible)
  – Data: zero or more values to be operated on (operands)

| 1-byte | opcode | operand | … | operand |

Operands are 1-byte each so
For a 2-byte index you need:
(operand1 << 8) | operand2
Java Bytecode Instructions

- Refer to method parameters and local variables by number
  - 256 is the maximum
  - **wide** version of instruction extends this to 65536
  - EX: `istore_2` pops tos and puts value in local variable number 2

- Java stack (operand stack)
  - Holds the operands during operations
  - All (virtual) computation performed here
  - 32-bits wide; longs/doubles take 2 stack entries

```
maxstack=5, maxlocals=8
Code[] = {push,add,store,ldc}
ExceptionTable[]={ ... }
Attributes
```

Two per method data structures:

Local variable array (max index is 64K-1):

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Index: 0 1 2 3 4 5

Holds params followed by local vars

Operand stack

Grows/shrinks with instructions executed

Each method gets its own
Java Bytecode Instructions

- Refer to method parameters and local variables by number
  - 256 is the maximum
  - **wide** version of instruction extends this to 65536
  - EX: `istore_2` pops tos and puts value in local variable number 2

- Java stack (operand stack)
  - Holds the operands during operations
  - All (virtual) computation performed here
  - 32-bits wide; longs/doubles take 2 stack entries

Two per method data structures:

Local variable array (max index is 64K-1):

<table>
<thead>
<tr>
<th>Index: 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

Holds params followed by local vars

Operand stack

Grows/shrinks with instructions executed

Each method gets its own
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}

public static void main(java.lang.String[]);

Code:
  0:  aload_0
  1:  arrayLength
  2:  istore_1
  3:  ldc     #2;  //int 1000027
  5:  istore_2
  6:  iload_1
  7:  iload_2
  8:  iadd
  9:  putstatic #3;  //Field
      //field1:I
 12:  aconst_null
 13:  putstatic #4;  //Field
      //field2:LSimple;
 16:  invokestatic #5;
      //Method foo():()V
 19:  return
public static void
main(java.lang.String[]);
Code:
0: aload_0
1: arrayLength
2: istore_1
3: ldc #2; //int 1000027
5: istore_2
6: iload_1
7: iload_2
8: iadd
9: putstatic #3; //Field
   //field1:I
12: aconst_null
13: putstatic #4; //Field
   //field2:LSimple;
16: invokestatic #5;
   //Method foo():V
19: return

class Simple {  
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {} 
}

public static void
main(String args[]) {  
    int i = args.length;
    int j = 1000027;
    field1 = i+j;
    field2 = null;
    foo();  
}
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}

Virtual
Local variable array (max index is 64K-1):

Index: 0 1 2 3 4 5
Holds params followed by local vars

class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}

public static void main(java.lang.String[]);

Code:
  0: aload_0
  1: arraylength
  2: istore_1
  3: ldc  #2; //int 1000027
  5: istore_2
  6: iload_1
  7: iload_2
  8: iadd
  9: putstatic #3; //Field
      //field1:I
 12: aconst_null
 13: putstatic #4; //Field
      //field2:LSimple;
 16: invokestatic #5;
      //Method foo:()V
 19: return
}
```java
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String[] args) {
        int i = args.length;
        int j = 1000027;
        field1 = i + j;
        field2 = null;
        foo();
    }
}
```

Virtual
Local variable array (max index is 64K-1):

| &args | i | j |

After instruction 0, operand stack = &args

```java
public static void main(java.lang.String[]);

Code:
0: aload_0
1: arraylength
2: istore_1
3: ldc #2; //int 1000027
5: istore_2
6: iload_1
7: iload_2
8: iadd
9: putstatic #3; //Field
    //field1:I
12: aconst_null
13: putstatic #4; //Field
    //field2:LSimple;
16: invokestatic #5;
    //Method foo:()V
19: return
```
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}

public static void main(java.lang.String[]):
    Code:
    0: aload 0
    1: arrayLength
    2: istore 1
    3: ldc #2; //int 1000027
    5: istore 2
    6: iload 1
    7: iload 2
    8: iadd
    9: putstatic #3; //Field
        //field1:I
    12: aconst_null
    13: putstatic #4; //Field
        //field2:LSimple;
    16: invokestatic #5;
        //Method foo:()V
    19: return
}
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}
}

public static void main(String[] args) {
    int i = args.length;
    int j = 1000027;
    field1 = i+j;
    field2 = null;
    foo();
}

public static void main(java.lang.String[]);
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}

public static void main(java.lang.String[] args) {
    Code:
    0: aload 0
    1: arraylength
    2: istore 1
    3: ldc #2; //int 1000027
    5: istore 2
    6: iload_1
    7: iload_2
    8: iadd
    9: putstatic #3; //Field
        //field1:I
    12: aconst_null
    13: putstatic #4; //Field
        //field2:LSimple;
    16: invokestatic #5;
        //Method foo():V
    19: return
}
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String[] args) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}

public static void main(java.lang.String[]);

Code:
0: aload 0
1: arrayLength
2: istore 1
3: ldc #2; //int 1000027
5: istore 2
6: iload_1
7: iload_2
8: iadd
9: putstatic #3; //Field
    //field1:I
12: aconst_null
13: putstatic #4; //Field
    //field2:LSimple;
16: invokestatic #5;
    //Method foo:()V
19: return
}

```java
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String[] args) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}
```

```java
public static void main(java.lang.String[]);  
Code:
    0:    aload_0
    1:    arraylength
    2:    istore_1
    3:    ldc    #2; //int 1000027
    5:    istore_2
    6:    iload_1
    7:    iload_2
    8:    iadd
    9:    putstatic    #3; //Field
                  //field1:I
   12:   aconst_null
   13:   putstatic    #4; //Field
                  //field2:LSimple;
   16:   invokestatic #5;
                  //Method foo():V
   19:   return
```
class Simple {
    static int field1;
    static Simple field2;
    static int field3;

    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}

class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {}

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        field1 = i+j;
        field2 = null;
        foo();
    }
}
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo() {} 

    public static void main(String args[]) {
        int i = args.length;
        int j = 1000027;
        Simple.field1 = i+j;
        Simple.field2 = null;
        Simple.foo();
    }
}

public static void main(java.lang.String[]);

Code:
0:  aload_0
1:  arrayLength
2:  istore_1
3:  ldc    #2; //int 1000027
5:  istore_2
6:  iload_1
7:  iload_2
8:  iadd
9:  putstatic    #3; //Field
    //field1:I
12:  aconst_null
13:  putstatic    #4; //Field
    //field2:LSimple;
16:  invokestatic    #5;
    //Method foo():V
19:  return
public static void
main(java.lang.String[]);

Code:
0: aload 0
1: arrayLength
2: istore_1
3: ldc #2; //int 1000027
5: istore 2
6: iload_1
7: iload 2
8: iadd
9: putstatic #3; //Field
    //field1:I
12: aconst_null
13: putstatic #4; //Field
    //field2:LSimple;
16: bipush 7
18: invokestatic #5;
    //Method foo:(i)V
21: return

When a method call is made
(using line-by-line interpretation):
Runtime moves arguments to correct
location in caller’s register (locals) array
class Simple {
    static int field1;
    static Simple field2;
    static int field3;

    public static void
        foo(int k) {}

    public static void
        main(String args[]) {
        int i = args.length;
        int j = 1000027;
        Simple.field1 = i+j;
        Simple.field2 = null;
        Simple.foo(i);
    }
}

When a method call is made
(see line-by-line interpretation):
Runtime moves arguments to correct
location in caller’s register (locals) array

public static void
main(java.lang.String[]);

Code:
0: aload 0
1: arrayLength
2: istore 1
3: ldc #2; //int 1000027
5: istore 2
6: iload_1
7: iload_2
9: putstatic #3; //Field
     //field1:I
12: aconst_null
13: putstatic #4; //Field
     //field2:LSimple;
16: ???
17: invokestatic #5;
     //Method foo:(i)V
20: return
}
class Simple {
    static int field1;
    static Simple field2;
    static int field3;
    public static void foo(int k) {} 
}

public static void main(String args[]) {
    int i = args.length;
    int j = 1000027;
    Simple.field1 = i+j;
    Simple.field2 = null;
    Simple.foo(i);
}

When a method call is made (using line-by-line interpretation):
Runtime moves arguments to correct location in caller’s register (locals) array

Code:
0: aload 0
1: arrayLength
2: istore 1
3: ldc #2; //int 1000027
5: istore 2
6: iload_1
7: iload_2
8: iadd
9: putstatic #3; //Field
    //field1:I
12: aconst_null
13: putstatic #4; //Field
    //field2:LSimple;
16: iload_1
17: invokestatic #5; //Method foo:(i)V
20: return
Practice, Practice, Practice!

//dump the bytecode (shows constant pool entries resolved)
javap -c Simple    //Simple must be in your classpath

JVM instructions:
http://cs.au.dk/~mis/dOvs/vmspec/VMSpecIX.fm.html

//http://www.cs.ucsb.edu/~cs263/showme.tar.gz
  //cs263 dir must be in your class path
//dump constant pool (so you can resolve the entries yourself)
java cs263.ShowMeTheStructure Simple.class