Compiler Design
Fall 2018

The Java Virtual Machine
(part of Code Generation)

Bernd Burgstaller
Yonsei University

Lecture 4: The Java Virtual Machine (JVM)

1. JVM Primer

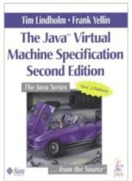
2. MiniC code generation

   Jasmin assembly code is our target language

3. JVM Specification
   - Data types
   - Operand Stack
   - Local variables
     - local variable array & indices
   - Instructions
     - Jasmin instructions
   - Parameter passing
     - Jasmin method invocations

A Brief JVM & Java Bytecode Primer…

- The JVM is a stack-based virtual machine
  - uses a stack instead of registers as intermediate storage for values of a computation
  - Arguments are pushed onto the stack
  - Operations take their operands from the stack and push the result back on the stack.



specifies the meaning of JVM bytecode instructions:

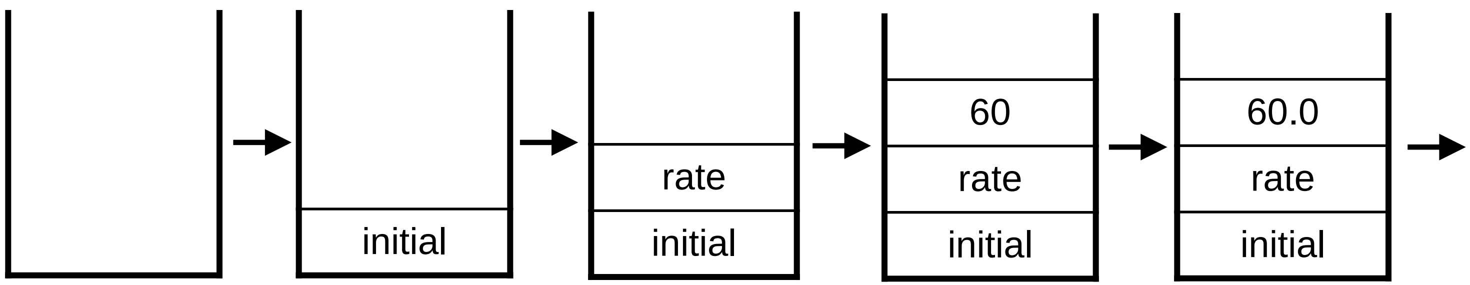
- fload_<n> : push float value of local variable number <n> onto the stack
- bipush <b> : push byte <b> onto the stack
- i2f: convert the topmost stack-element from int to float
- fmul: perform floating-point multiplication of the topmost stack elements, push the result onto the stack
- fadd: compute the sum of the two topmost stack elements, push the result onto the stack.
- fstore_<n>: pop the topmost stack element and store it in local variable number <n>.

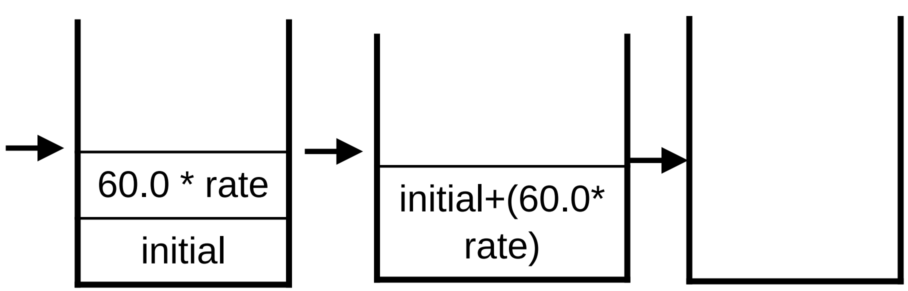
A Brief JVM & Java Bytecode Primer… (cont.)

```
fload_2
fload_3
bipush 60
i2f
fmul
fadd
fstore_1
```

```
float position;  // local var index 1
float initial;   // local var index 2
float rate;      // local var index 3

position = initial + (rate * 60);
```

Initially the JVM stack is empty:





After the last instruction, the stack is again empty, and variable “position” in local variable index slot 1 has been assigned the value “initial+(rate*60).

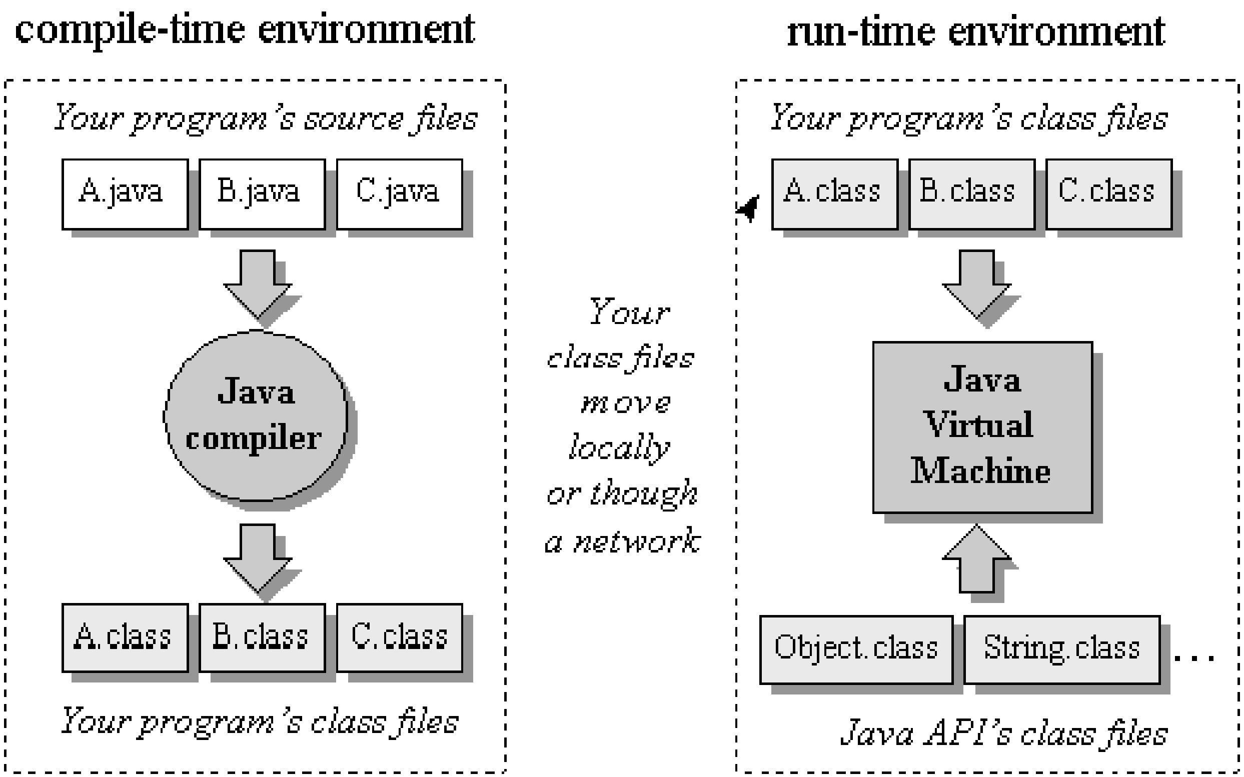
JVM Bytecode Example

MiniC source code is compiled to Jasmin assembly code:

```
//MiniC source code:

void whileInt() {
      int i = 0;
      while (i < 100) {
        i = i + 1;
      }
}
```

```
;; Jasmin assembly code:

.method  whileInt()V
    iconst_0
    istore_1 ;; i's index is 1
Label1:
    iload_1
    bipush 100              loop condition
    if_icmpge Label0
    iload_1
    iconst_1
    iadd                    i=i+1
    istore_1
    goto Label1
Label0:
    return

.end method
```

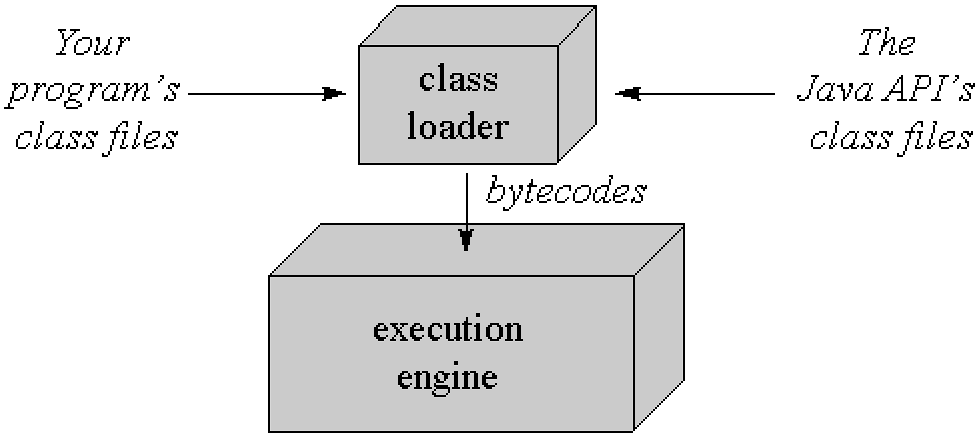
Java Compile-time and Run-time Environment



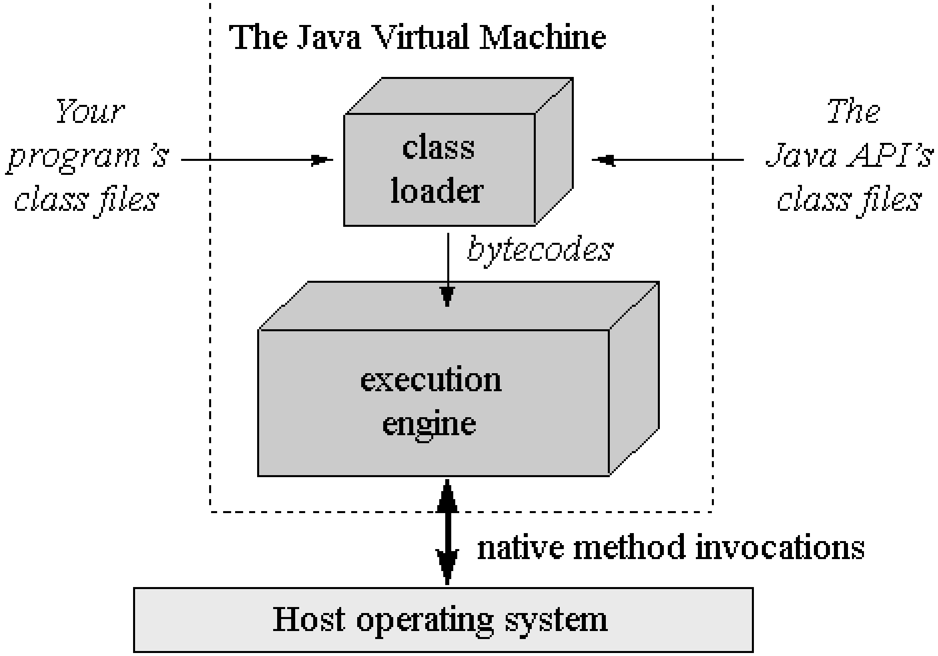
From ``Inside the Java Virtual Machine'' by Bill Venners

- An application's Java code is compiled to bytecode in classfiles.
  - It may call methods in the Java API (provided in classfiles)
  - At runtime, the application classfiles and required API class files are loaded.

Pure JVM versus Native Calls

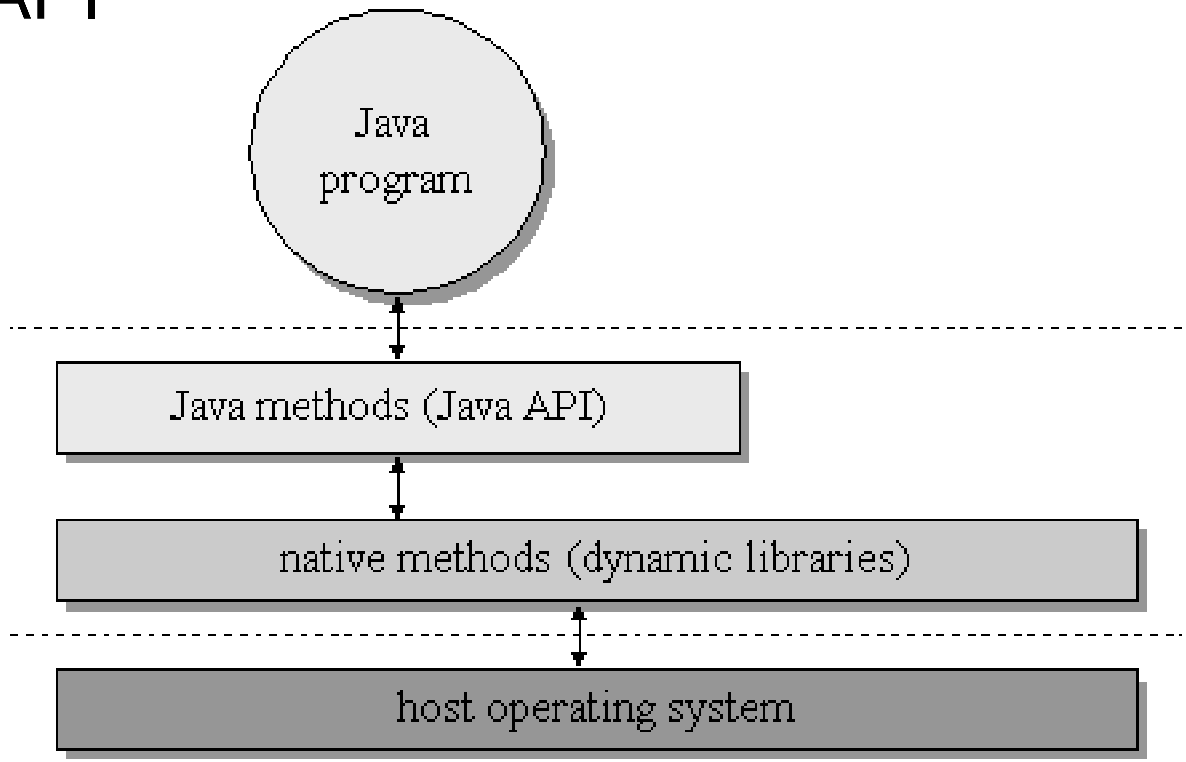


- Pure bytecode interpretation
  - hardware independent



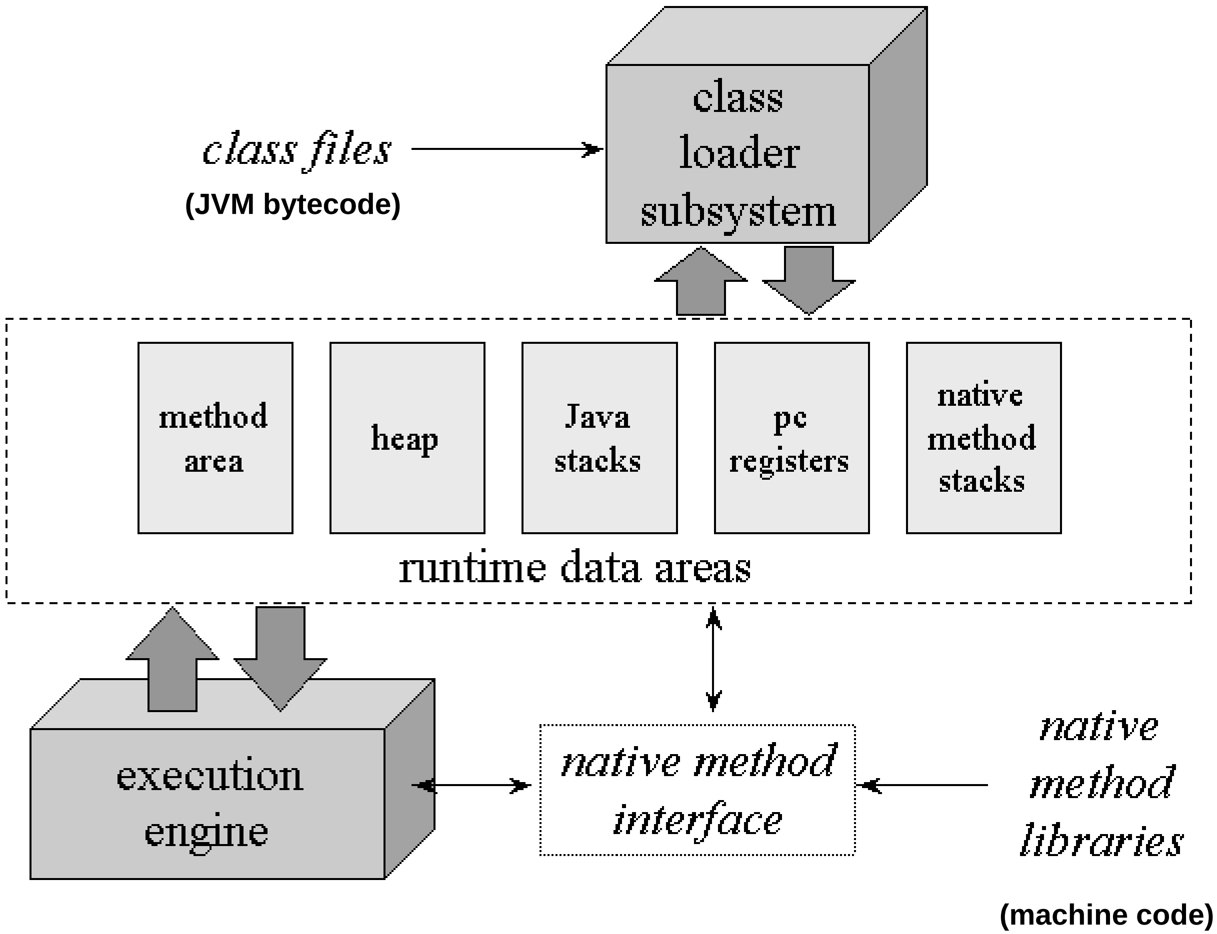
- Bytecode contains calls to machine code
  - through the Java native interface (JNI)
  - not hardware independent anymore
    - platform-specific

The Java API



- Java API consists of a set of run-time libraries to access system resources in a standard way (network, files, GUI).
  - Java programmer may assume that those are available on all Java platforms
- Classfiles from the Java API are host-specific and rely on native methods to do their work
  - such that your Java program does not have to...

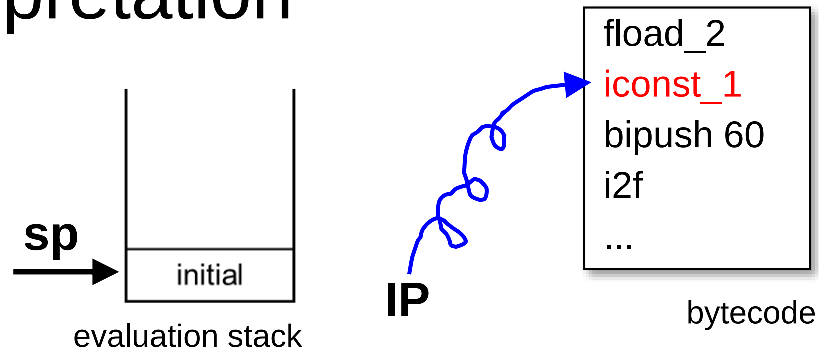
Internal Architecture of the JVM



From ``Inside the Java Virtual Machine'' by Bill Venners

Bytecode Interpretation

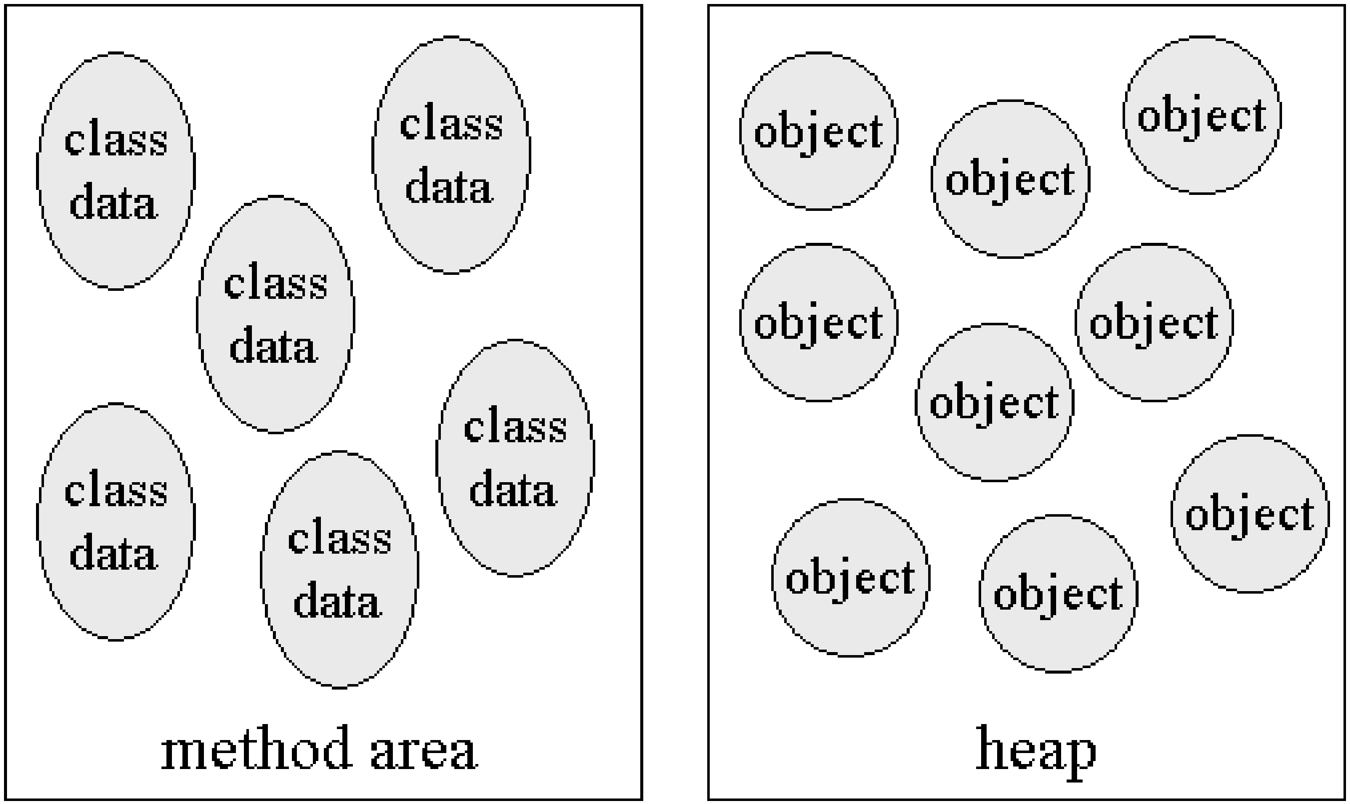
- The interpreter is just a big **loop** with a `switch` statement:
  - **IP** is the instruction pointer
  - **sp** is the stack pointer
  - **bc** is the currently interpreted bytecode
  - The `switch` statement contains one arm (``case:'') for each instruction in the instruction-set of the interpreter.
- Advantages:
  - Memory-efficient
  - very simple, easy to implement
  - easy to port to other architectures
- Disadvantage:
  - slow…
- We will discuss more efficient techniques in the Programming Language grad course.




```
while (1) {
  bc = *IP++;
  switch (bc) {
    ...
    case iconst_1:
      *++sp = ConstOne;
      break;
    ...
  }
}
```

Interpreter (execution engine), implemented in C

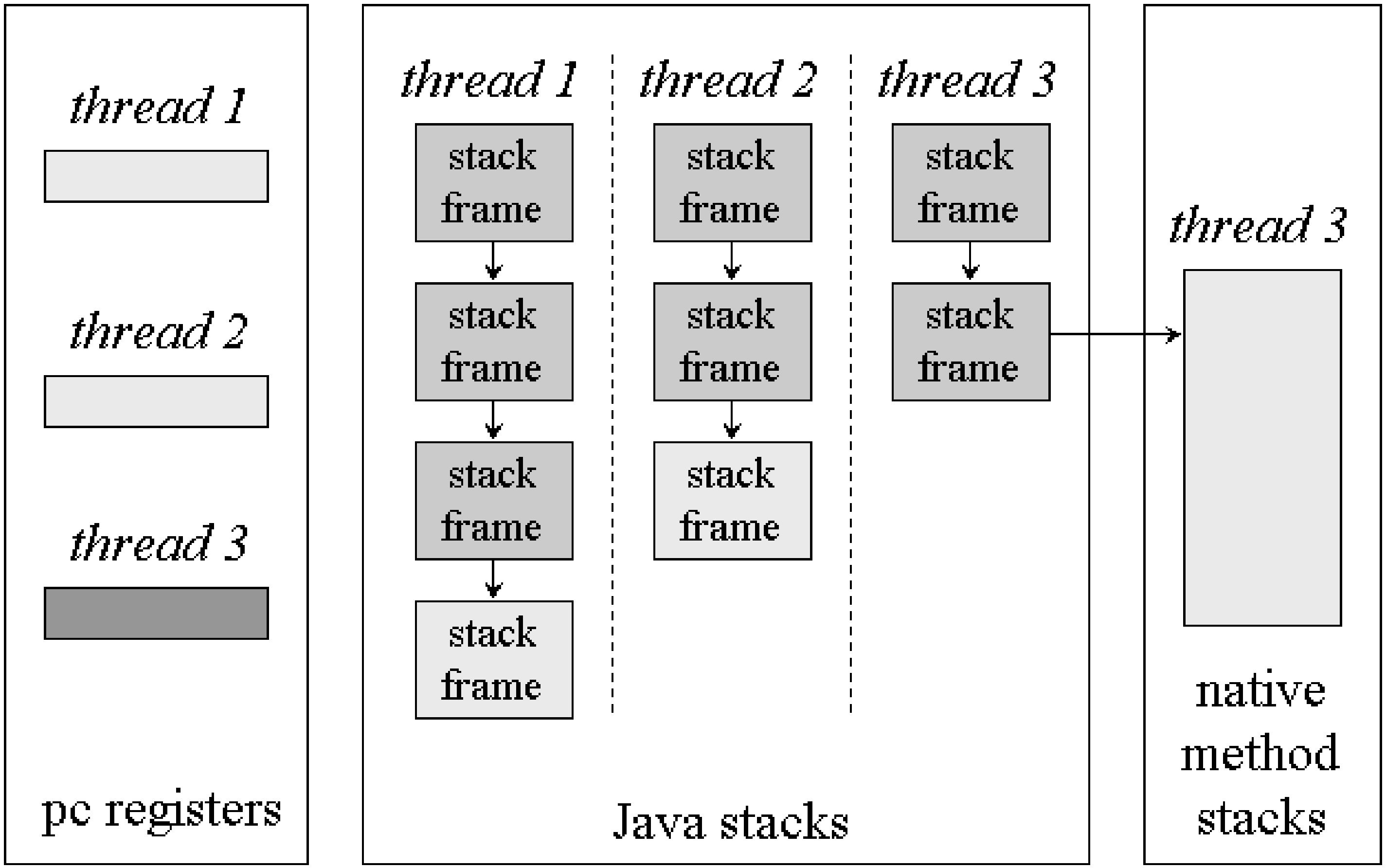
Method Area & Heap



- Arrays are objects

From ``Inside the Java Virtual Machine'' by Bill Venners

Java Stacks



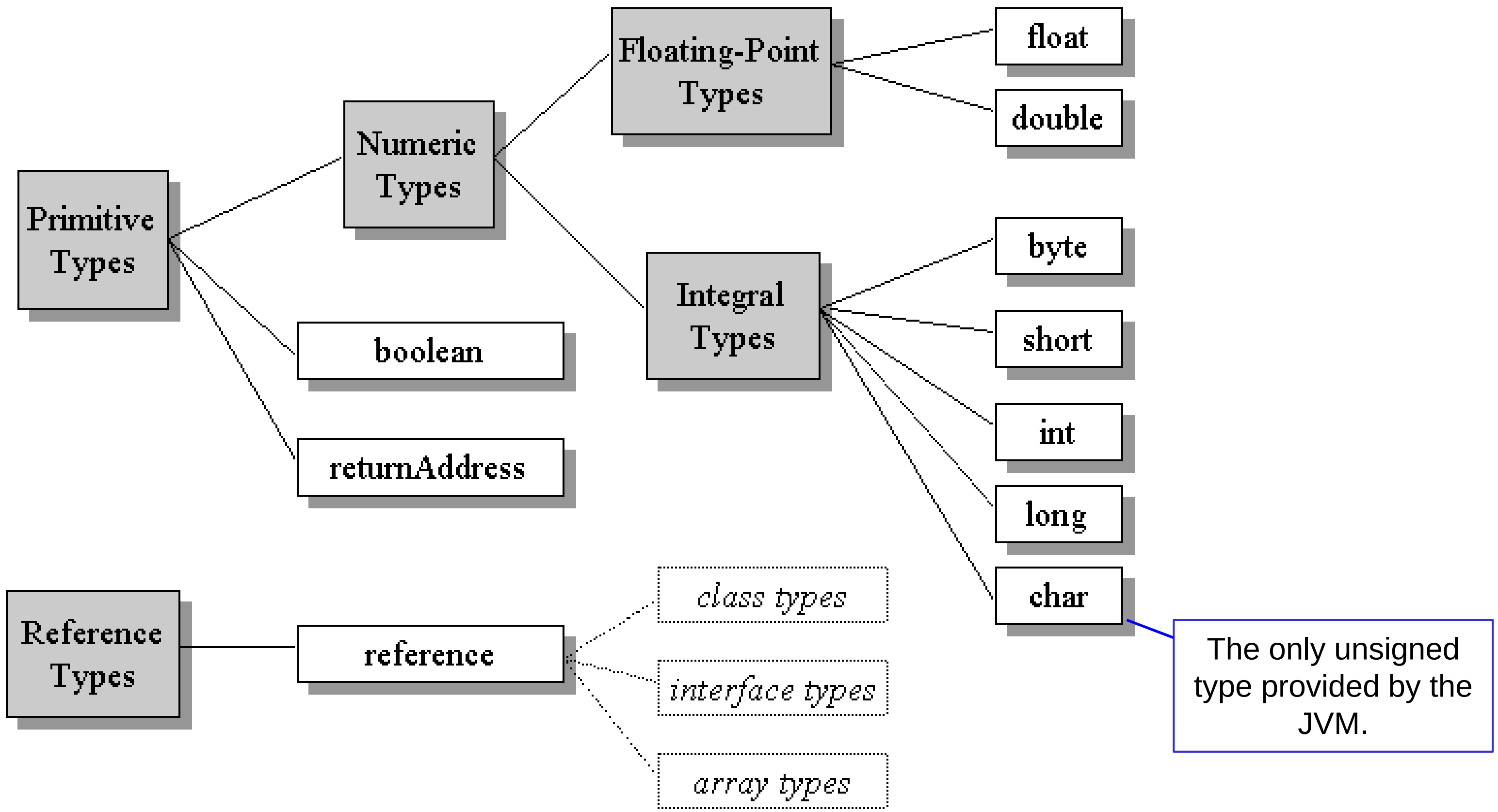
- One stack per thread
- One stack-frame per method invocation

From ``Inside the Java Virtual Machine'' by Bill Venners

Stack Frames

- One stack frame is created for each method invocation
- A stack frame consists of:
  - local variables
    - size of local variable array depends on the invoked method (class-file)
  - operand stack
    - size depends on the invoked method (class file)
  - frame data
    - Ptr to constant pool, information of previous stack frame on stack, exception handling table
    - size depends on the JVM implementation

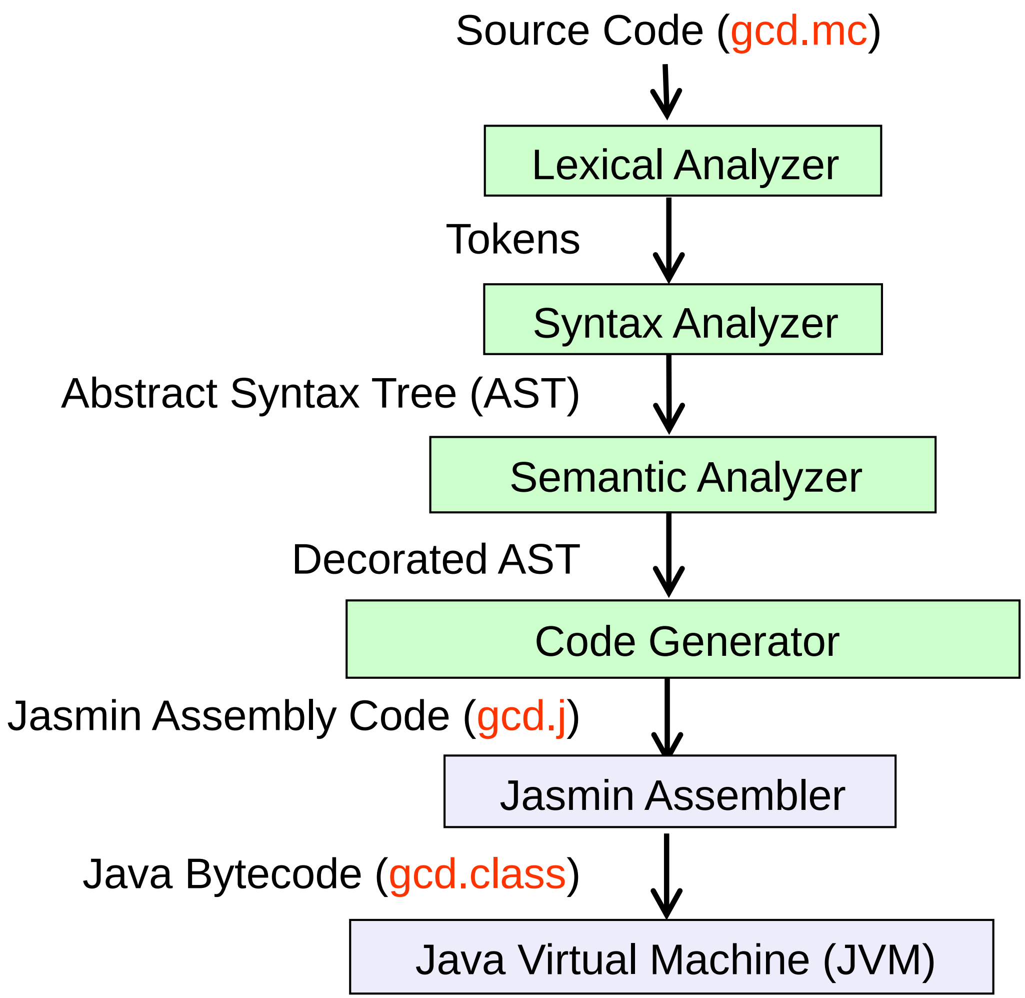
Data Types provided by the JVM



The Dalvik Virtual Machine

- Introduced by Google for the Android mobile phone platform
- Implementation of Java
- Dalvik VM is register-based
  - registers (similar to CPU registers) instead of evaluation stack
- Bytecode format different from the JVM:
  - .dex files instead of .jar class files
  - tool to convert class files to .dex files is provided
  - .dex files are slightly smaller than .jar files
- uses threaded code to interpret bytecode
- Dalvik VM internals:
  - http://www.youtube.com/watch?v=ptjedOZEXPM

Structure of the MiniC Compiler



References

- The Jasmin Homepage
  http://jasmin.sourceforge.net/

- The Jasmin User Guide
  http://jasmin.sourceforge.net/guide.html

- Tim Lindholm, Frank Yellin,
  The Java Virtual Machine Specification
  https://docs.oracle.com/javase/specs/jvms/se10/html/

- Bill Venners
  Inside the Java Virtual Machine, McGraw-Hill, 1999.
  Several chapters available online at
  http://www.artima.com/insidejvm/ed2/

Jasmin Assembly Language

- SUN Microsystems has not defined an assembly format for Java bytecode.
- Jasmin is a Java assembler which has been installed on the elc1 server.
- Jasmin instructions have 1-to-1 correspondence to Java bytecode instructions.
  - Operation codes (op-codes) represented by mnemonics
  - fields written in symbolic form
  - Local variables are encoded by indices (integers)
- Examples (to be discussed with our JVM introduction)

Encoding of Jasmin assembly instructions, in general:

ASCII string “bipush 20” → <opcode> <operand>

- covers less space than ASCII string
- easier to decode (see Slide #7)

Example gcd.java

```
//Java source code, to compute the greatest
//common divisor:

public class gcd {
   static int gcd(int a, int b) {
      while (b != 0) {
        if (a > b)
          a = a - b;
        else
          b = b - a;
      }
      return a;
   }

   public static void main(String argv[]) {
      int i = 2;
      int j = 4;
      System.out.println(gcd(i,j));
   }
}
```

Example gcd.j (part 1)

```
; gcd.j

; Generated by ClassFileAnalyzer (Can)
; Analyzer and Disassembler for Java class files
; (Jasmin syntax 2, http://jasmin.sourceforge.net)
;
; ClassFileAnalyzer, version 0.7.0

.bytecode 50.0
.source gcd.java
.class public gcd
.super java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1
    .var 0 is this Lgcd; from Label0 to Label1

    Label0:
    .line 1
       0: aload_0
       1: invokespecial java/lang/Object/<init>()V
    Label1:
       4: return
.end method
```

The jasmin assembler and disassembler are installed on the elc1 server. To disassemble a class file:

javac gcd.java

classfileanalyzer gcd.class > gcd.j

To assemble:

jasmin gcd.j

(will again produce gcd.class)

Example gcd.j (part 2)

```
.method static gcd(II)I
    .limit stack 2
    .limit locals 2
    .var 0 is arg0 I from Label2 to Label5
    .var 1 is arg1 I from Label2 to Label5
    Label2:
    .line 3
       0: iload_1         // index of b
       1: ifeq Label0
    .line 4
       4: iload_0
       5: iload_1
       6: if_icmple Label1
    .line 5
       9: iload_0         // index of a
      10: iload_1
      11: isub
      12: istore_0
      13: goto Label2
    Label1:
    .line 7
      16: iload_1
      17: iload_0
      18: isub
      19: istore_1
      20: goto Label2
    Label0:
    .line 9
      23: iload_0
    Label5:
      24: ireturn
.end method
```

```
1  public class gcd {
2    static int gcd(int a, int b) {
3      while (b != 0) {
4        if (a > b)
5          a = a - b;
6        else
7          b = b - a;
8      }
9      return a;
10 }
```

Example gcd.j (part 3)

```
.method public static main([Ljava/lang/String;)V
    .limit stack 3
    .limit locals 3
    .var 0 is arg0 [Ljava/lang/String; from Label0 to Label1

    Label0:
    .line 13
       0: iconst_2
       1: istore_1   // index of i
    .line 14
       2: iconst_4
       3: istore_2   // index of j
    .line 15
       4: getstatic java.lang.System.out Ljava/io/PrintStream;
       7: iload_1
       8: iload_2
       9: invokestatic gcd/gcd(II)I
      12: invokevirtual java/io/PrintStream/println(I)V
    Label1:

.line 16
      15: return
.end method
```

```
12 public static void main(String argv[]) {
13       int i = 2;
14       int j = 4;
15       System.out.println(gcd(i,j));
16 }
```

Example gcd.class

```
ca fe ba be 00 00 00 32 00 1f 0a 00 06 00 12 09
00 13 00 14 0a 00 05 00 15 0a 00 16 00 17 07 00
0b 07 00 18 01 00 06 3c 69 6e 69 74 3e 01 00 03
28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e
65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00 03 67
63 64 01 00 05 28 49 49 29 49 01 00 0d 53 74 61
63 6b 4d 61 70 54 61 62 6c 65 01 00 04 6d 61 69
6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67
2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 75
72 63 65 46 69 6c 65 01 00 08 67 63 64 2e 6a 61
76 61 0c 00 07 00 08 07 00 19 0c 00 1a 00 1b 0c
00 0b 00 0c 07 00 1c 0c 00 1d 00 1e 01 00 10 6a
61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 01
00 10 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74
65 6d 01 00 03 6f 75 74 01 00 15 4c 6a 61 76 61
2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b
01 00 13 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74
53 74 72 65 61 6d 01 00 07 70 72 69 6e 74 6c 6e
01 00 04 28 49 29 56 00 21 00 05 00 06 00 00 00
00 00 03 00 01 00 07 00 08 00 01 00 09 00 00 00
1d 00 01 00 01 00 00 00 05 2a b7 00 01 b1 00 00
00 01 00 0a 00 00 00 06 00 01 00 00 00 01 00 08
00 0b 00 0c 00 01 00 09 00 00 00 4c 00 02 00 02
00 00 00 19 1b 99 00 16 1a 1b a4 00 0a 1a 1b 64
3b a7 ff f3 1b 1a 64 3c a7 ff ec 1a ac 00 00 00
02 00 0a 00 00 00 16 00 05 00 00 00 03 00 04 00
04 00 09 00 05 00 10 00 07 00 17 00 09 00 0d 00
00 00 05 00 03 00 0f 06 00 09 00 0e 00 0f 00 01
00 09 00 00 00 34 00 03 00 03 00 00 00 10 05 3c
07 3d b2 00 02 1b 1c b8 00 03 b6 00 04 b1 00 00
00 01 00 0a 00 00 00 12 00 04 00 00 00 0d 00 02
00 0e 00 04 00 0f 00 0f 00 10 00 01 00 10 00 00
00 02 00 11
```

Output of
od –An –tx1 gcd.class

Lecture 4: The Java Virtual Machine (JVM)

1. JVM Primer ✓

2. MiniC code generation ✓

   Jasmin assembly code is our target language ✓

3. JVM
   - Data types
   - Operand Stack
   - Local variables
     - local variable array & indices
   - Instructions
     - Jasmin instructions
   - Parameter passing
     - Jasmin method invocations

JVM Data Types

Data type	Value Range	Descriptor
boolean	{0, 1}	Z
byte	8 bit signed 2's complement, (-2^7 to 2^7 -1)	B
short	16 bit signed 2's complement, (-2^{15} to 2^{15} -1)	S
int	32 bit signed 2's complement, (-2^{31} to 2^{31} -1)	I
long	64 bit signed 2's complement, (-2^{63} to 2^{63} -1)	L
char	16 bit unsigned Unicode (0 to 2^{16}-1)	C
float	32-bit IEEE 754-single precision	F
double	64-bit IEEE 754-double precision	D
reference	32 bit unsigned reference (0 to 2^{32}-1)	see next slide

- MiniC types are mapped 1-to-1 to Java's primitive types.
  - int→int, bool→boolean, float→float, string→Java.Lang.String
- Java's boolean, byte, char and short are all implemented as int, but arrays of these types may be stored in arrays of less than 32 bits.

JVM Data Types (cont.)

Data type	Descriptor
class name	class-name
interface name	interface-name
array reference	[[...[component-type
void	V

A semi-colon ";" marks the end of a class or interface descriptor.

- Class and interface names are qualified names with "." replaced by "/"
- The number of brackets "[" is equal to the number of dimensions of an array.

Data type	Descriptor
class java.lang.Object	java/lang/Object
class java.lang.String	java/lang/String
reference to instance of class java.lang.Object	Ljava/lang/Object;
String[]	[Ljava/lang/String;
int[]	[I
float [] []	[[F

- See $4.3.2 of the JVM Spec for a formal definition.

Boolean, Byte, Short and Char represented as Int

```
.method public static main([Ljava/lang/String;)V
    .limit stack 1
    .limit locals 5
    .var 0 is arg0 [Ljava/lang/String; from Label0 to Label1


    Label0:
    .line 3
       0: iconst_1
       1: istore_1

    .line 4
       2: iconst_1
       3: istore_2

    .line 5
       4: iconst_2
       5: istore_3

    .line 6
       6: bipush 116
       8: istore 4

    Label1:
.line 7
      10: return

.end method
```

```
1 public class IntTypes {
2   public static void main(String argv[]) {
3     boolean z = true;
4     byte b = 1;
5     short s = 2;
6     char c = 't';
7   }
8 }
```

Java's boolean, byte, short and char are all implemented as int.

Printing Data Type Descriptors

```
public class Desc {
  public static void main(String argv[]) {

    Object o = new Object();
    int [] i = new int[10];
    float [] [] f = new float[10][10];
    String s1 = "Hello Compiler!";
    String [] s2 = { "Hello", "Compilerwriter!" };

    System.out.println("The class name of Object is: " + o.getClass());
    System.out.println("The class name of int[] is: " + i.getClass());
    System.out.println("The class name of float[][] is: " + f.getClass());
    System.out.println("The class name of String is: " + s1.getClass());
    System.out.println("The class name of String[]: " + s2.getClass());
  }

}
```

Generated output:

```
The class name of Object is: class java.lang.Object
The class name of int[] is: class [I
The class name of float[][] is: class [[F
The class name of String is: class java.lang.String
The class name of String[]: class [Ljava.lang.String;
```

Method Descriptors

A method descriptor specifies a method's return type and the number and types of its arguments.

Format: (ParameterType*) ReturnType

Examples:

Method Declaration	Method Descriptor
int gcd(int i, int j)	(II)I
void main(String argv[])	([Ljava/lang/String;)V
char foo (float f, String)	(FLjava/lang/String;)C

See $4.3.3 of the JVM Spec for more information.

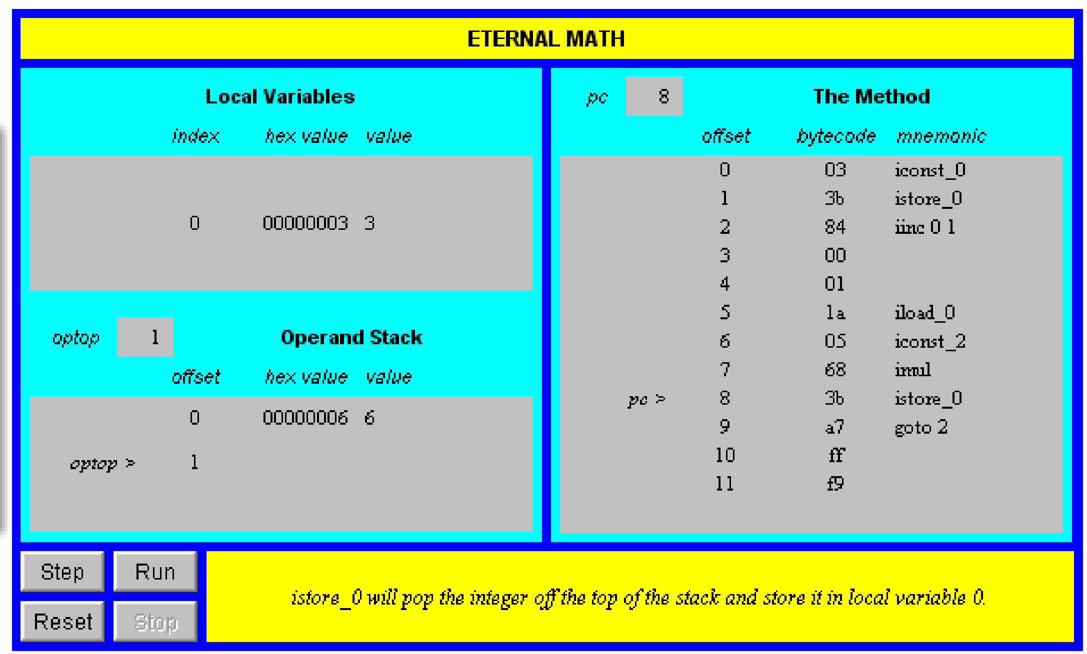
Method + Local Variable Array + Operand Stack

Whenever a Java method is called, the JVM creates

1) an operand stack
2) a local variable array for the method to execute.

Run the simulation at http://www.artima.com/insidejvm/applets/EternalMath.html

```
class Act {

 public static void doMathForever() {
     int i = 0;
     for (;;) {
       i += 1;
       i *= 2;
     }
 }
}
```



The Operand Stack

- Accessed by pushing and popping values
  - storing operands and receiving the operand’s results
  - passing arguments to a method
  - receiving the result returned by a called method
- A new operand stack is created every time a method is called.
- This unified view is one of the main reasons why code generation for stack-based machines is easier than for register-based machines.

Java Class Methods and Instance Methods

The Java programming language provides two kinds of methods:

1. Class or static methods
   - declared using the keyword "**static**".
   - do not require an object instance to be called
     → cannot access instance variables of an object.
   - invoked via the class name: **SomeClass.foo();**
   - like a procedure call, bind at compile-time

```
class SomeClass {
    public static void foo() {}
}
```

2. Instance Methods
   - declared without keyword "**static**"
   - require an object instance to be called
     → can access instance variables of an object.
   - invoked via the object:
     SomeClass x = new SomeClass(); x.foo();
   - bind (dispatch) at run-time
     → need the "this" pointer as the method's implicit first argument to point to the object's instance (to figure out the target method of the dispatching call, see also slides on the visitor design pattern).

```
class SomeClass {
    public void foo() {}
}
```

Further details: http://java.sun.com/docs/books/tutorial/java/javaOO/classvars.html

The Local Variable Array

- A new local variable array is created each time a method is called.
- Local variables are addressed using indices
  - smallest index: 0
- Instance methods
  - slot 0 allocated to this-pointer
  - actual parameters (if any) given consecutive indices, starting from 1
  - Indices allocated to the other local variables in any order
- Class methods
  - actual parameters (if any) given consecutive indices, starting from 0
  - Indices allocated to the other local variables in any order
- One slot can hold a value of type boolean, byte, char, short, int, float, or reference.
- One pair of slots can hold a `long` or a `double`.

Local Variable Indices: Class Methods

```
.method public static foo()V
    .limit stack 2
    .limit locals 4


    .line 2
       0: iconst_1
       1: istore_0

    .line 3
       2: iconst_2
       3: istore_1

    .line 4
       4: iconst_3
       5: istore_2

    .line 5
       6: iload_0
       7: iload_1
       8: iadd
       9: iload_2
      10: iadd
      11: istore_3

    .line 6
      12: return

.end method
```

```
1 public static void foo() {
2     int i1 = 1;  // index 0
3     int i2 = 2;  // index 1
4     int i3 = 3;  // index 2
5     int i = i1 + i2 + i3; // index 3
6 }
```

Local Variable Indices: Instance Methods

```
.method public foo1()V
    .limit stack 2
    .limit locals 5
    .var 0 is this LDesc; from Label0 to Label1

    Label0:
    .line 24
       0: iconst_1
       1: istore_1 // i1 has index 1 now!
    .line 25
       2: iconst_2
       3: istore_2
    .line 26
       4: iconst_3
       5: istore_3
    .line 27
       6: iload_1
       7: iload_2
       8: iadd
       9: iload_3
      10: iadd
      11: istore 4
    Label1:
.line 28
      13: return
.end method
```

```
public void foo1() { // "this" given index 0
    int i1 = 1;  // index 1
    int i2 = 2;  // index 2
    int i3 = 3;  // index 3
    int i = i1 + i2 + i3; // index 4
 }
```

Lecture 4: The Java Virtual Machine (JVM)

1. JVM Primer ✓

2. MiniC code generation ✓

   Jasmin assembly code is our target language ✓

3. JVM
   - Data types ✓
   - Operand Stack ✓
   - Local variables ✓
     - local variable array & indices ✓
   - Instructions
     - Jasmin instructions
   - Parameter passing
     - Jasmin method invocations

Jasmin (or JVM) instructions

1. Arithmetic instructions
2. Load/Store instructions
3. Transfer of control instructions
4. Type conversion instructions
5. Operand stack management instructions
6. Object creation and manipulation
7. Method invocation instructions
8. Throwing exceptions
9. Implementing finally (exceptions)
10. Synchronization

We won't cover the greyed-out topics, because we don't need them for our MiniC implementation on top of the JVM.

The interested reader is referred to the "Reference" slide at the beginning of this presentation.

Arithmetic Instructions ($3.11.3, JVM Spec)

- add: iadd, fadd
- subtract: isub, fsub
- multiply: imul, fmul
- divide: idiv, fdiv
- negative: ineg, fneg
- comparison: fcmpg, fcmpl
- ...

Load/Store Instructions ($3.11.2, JVM Spec)

- Loading a local variable onto the operand stack:
```
iload <index #nr>, iload_0, ..., iload_3
fload <index #nr>, fload_0, ..., fload_3
```
  ...
- Storing a value from the operand stack into a local variable:
```
istore <index #nr>, istore_0, ..., istore_3
fstore <index #nr>, fstore_0, ..., fstore_3
```
  ...
- Load a constant onto the operand stack:
```
bipush, sipush, ldc, iconst_0, iconst_1, ...,
iconst_5, fconst_0, ...
```

Transfer of control instructions ($3.11.7, JVM Spec)

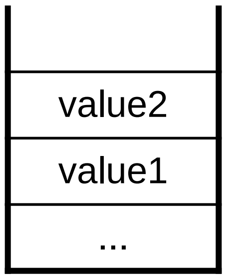
- Unconditional:
  `goto`

- Conditional jumps on **int:**
  - comparing the topmost stack element against 0:
    `ifeq` (==0), `ifne` (!=0), `ifle` (<=0), `iflt`, `ifge`, `ifgt`
  - comparing the two topmost stack elements:
    `if_icmpeq`, `if_icmpne`, `if_icmple`, `if_icmplt`,
    `if_icmpge`, `if_icmpgt`
    ...

- Conditional jumps on **float:**
  `fcmpg,fcmpl`
  and then
  `ifeq, ifne, ifle, iflt, ifge, ifgt`

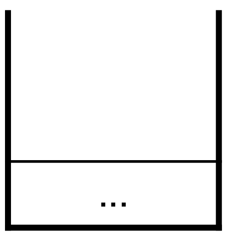
if_cmpge label

- Operand stack before compare:




```
;; Jasmin code:
   ...
if_cmpge label
   ... ;; false
   ... ;; "fall through"
label:
   ... ;; true
   ... ;; "branch taken"
```

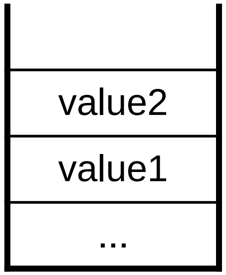
- Operand stack after compare:



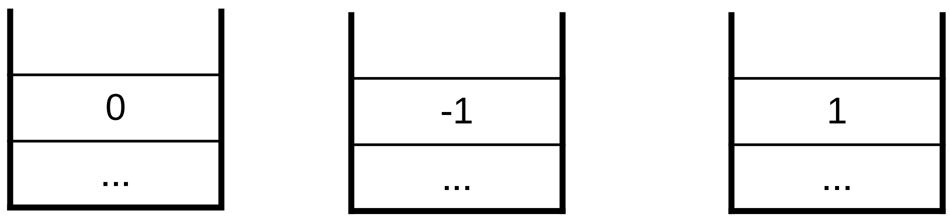
1. Pop the two int values and compare them
2. If value1 >= value2, jump to label, otherwise continue execution at the instruction following **`if_cmpge`**.

fcmpg and fcmpl

- Operand stack before comparison:



- Operand stack after comparison:



value1 == value2   value1 < value2   value1 > value2

- If value1 or value2 is NaN (not-a-number), fcmp* comparison is false
  - In the NaN case, fcmpg pushes 1 and fcmpl pushes -1
  - javac compiler selects fcmpg or fcmpl, depending on the required value for a ``false'' comparison.

```
if (x<y)...
```

```
fcmpg
ifge l_false
... ; "true" branch
l_false:
```

```
if (x>y)...
```

```
fcmpl
ifle l_false
... ; "true" br.
l_false:
```

Type Conversion Instructions

```
int i = 1;    // index 1
float f = i; // index 2
```

```
;; Jasmin code:
iconst_1
istore_1
iload_1
i2f
fstore 2
```

- Only i2f is used in the MiniC compiler.
- i2c, i2b, f2i, aso not used.

Method Invocation Instructions

- Method calls:
  - invokestatic
  - invokevirtual
  - invokespecial (also known as invoke**non**virtual)
    - the instance initialization method <init>
    - a private method of <this>
    - a method in a super-class of this
  - invokeinterface
    - possibly more run-time overhead, because of multiple inheritance

- Method returns:
  - return
  - ireturn
  - freturn

The Syntax for Method Invocation Instructions

- Invokestatic/virtual/special:

  invoke* method-spec

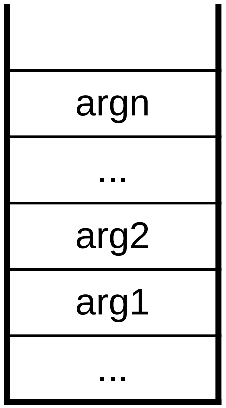
  where method-spec consists of a classname, a field name and a descriptor.

- Invokeinterface not used in our MiniC compiler implementation.

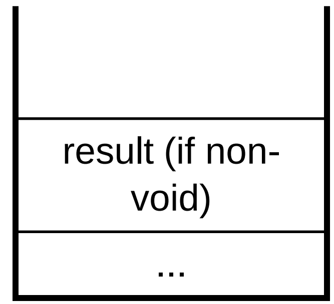
Method Invocation (cont.)

- Invokestatic:

operand stack
before call:

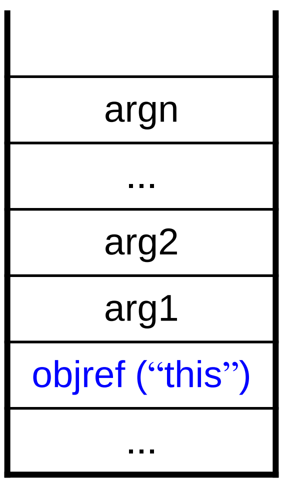


operand stack
after call:

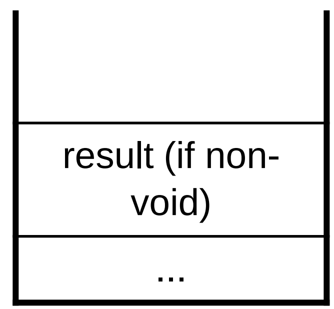


- Invokevirtual and invokespecial:

operand stack
before call:



operand stack
after call:



Static Method Invocation

```
public class Met1 {
   static int add(int i1, int i2) {
     return i1 + i2;
   }

   public static void main(String argv[]) {
     add(1,2);    // no object reference needed!
   }
}
```

```
.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 1
  .line 7
    0: iconst_1
    1: iconst_2
    2: invokestatic Met1/add(II)I    ;; no "this" pointer passed to add
    5: pop                           ;; unused result of add() discarded
  .line 8
    6: return
.end method
```

Instance Method Invocation

```java
public class Met2 {
   int add(int i1, int i2) {
     return i1 + i2;
   }

   public static void main(String argv[]) {
     Met2 m = new Met2();
     m.add(1,2);
   }
}
```

```
.method public static main([Ljava/lang/String;)V
    .limit stack 3
    .limit locals 2
    .var 0 is arg0 [Ljava/lang/String; from Label0 to Label1

    Label0:
    .line 7
       0: new Met2        ;; allocate Met2 object on heap, returns "this" on stack
       3: dup             ;; duplicate "this" on stack
       4: invokespecial Met2/<init>()V         ;; call the Met2 constructor
       7: astore_1                             ;; store "this" in var 1 slot

    .line 8
       8: aload_1                              ;; push "this" on stack
       9: iconst_1                             ;; push value 1 on stack
      10: iconst_2                             ;; push value 2 on stack
      11: invokevirtual Met2/add(II)I          ;; call add() method
      14: pop                                  ;; discard unused return value

    Label1:
    .line 9
      15: return
.end method
```

Accessing Static Fields

```
public class Field {

   static int i; // class variable
                 // (static field)

   public static void main(String argv[]) {
      i = i + 1;
   }
}
```

```
.class public Field
.super java/lang/Object

.field static i I          ;; static field

.method public static main([Ljava/lang/String;)V
       ...
       0: getstatic Field.i I
       3: iconst_1
       4: iadd
       5: putstatic Field.i I
       8: return

.end method
```

Syntax:

- **`getstatic field-spec type-descriptor`**

  where **`field-spec`** consists of a **classname** followed by a **field-name**.

Reading Materials

- Try out the tools mentioned in this lecture!
  - Available on the server.
  - You can install them on your PC as well.
- The JVM Spec
  - Chapter 3 (on instructions)
  - Chapter 7 (more examples on compiling for the JVM)
- “Inside the JVM” book, Chapter 5

Lecture 4: The Java Virtual Machine (JVM)

1. JVM Primer ✓

2. MiniC code generation ✓

   Jasmin assembly code is our target language ✓

3. JVM
   - Data types ✓
   - Operand Stack ✓
   - Local variables ✓
     - local variable array & indices ✓
   - Instructions ✓
     - Jasmin instructions ✓
   - Parameter passing ✓
     - Jasmin method invocations ✓