Code Generation for the Java Virtual Machine

Dr. Dan Resler
Dept. of Computer Science
Virginia Commonwealth University

March 2010

Outline

Reference Book

Java Virtual Machine by Meyer/Downing
(O'Reilly: 1997) (out-of-print)
ISBN-13: 978-1565-92194-8

The Java Virtual Machine (JVM)

- a **virtual stack machine** – implemented in software
- designed to easily and naturally implement the Java programming language
- executes **Java bytecode**
  - each opcode is one byte – hence
  - compact representation (16 Kb is enough for the most complex applications)
  - easy to verify (type safety, no pointer arithmetic, no illegal operations, no stack overflow/underflow)
  - type safe (type checking, type conversion, type inference, instanceof, checkcast)
  - operations are specifically designed for the Java programming language

The Java Virtual Machine (JVM)

- a **virtual stack machine** – implemented in software
- designed to easily and naturally implement the Java programming language
- executes **Java bytecode**
  - each opcode single byte in length
  - some opcodes are multi-byte (the opcode is followed by operands)
  - instructions are tagged with type information (e.g. iadd, fadd, dadd, aadd)
  - type information is used by the JVM to enforce type safety
  - JVM bytecode is platform independent

The Java Virtual Machine (JVM)

- a **virtual stack machine** – implemented in software
- designed to easily and naturally implement the Java programming language
- executes **Java bytecode**
  - each opcode single byte in length
  - some require parameters (therefore are multi-byte instructions)
  - instructions fall into broad groups: load & store, arithmetic & logic, type conversion, object creation & manipulation, operand stack management, control transfer, method invocation & return

The Java Virtual Machine (JVM)

- a **virtual stack machine** – implemented in software
- designed to easily and naturally implement the Java programming language
- executes **Java bytecode**
  - each opcode single byte in length
  - some require parameters (therefore are multi-byte instructions)
  - instructions fall into broad groups: load & store, arithmetic & logic, type conversion, object creation & manipulation, operand stack management, control transfer, method invocation & return

The Java Virtual Machine (JVM)

- a **virtual stack machine** – implemented in software
- designed to easily and naturally implement the Java programming language
- executes **Java bytecode**
  - each opcode single byte in length
  - some require parameters (therefore are multi-byte instructions)
  - instructions fall into broad groups: load & store, arithmetic & logic, type conversion, object creation & manipulation, operand stack management, control transfer, method invocation & return

The Java Virtual Machine (JVM)

- a **virtual stack machine** – implemented in software
- designed to easily and naturally implement the Java programming language
- executes **Java bytecode**
  - each opcode single byte in length
  - some require parameters (therefore are multi-byte instructions)
  - instructions fall into broad groups: load & store, arithmetic & logic, type conversion, object creation & manipulation, operand stack management, control transfer, method invocation & return

Jasmin

- will present Jasmin JVM assembly-language
- Jasmin is an assembler for the JVM
- It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the JVM instruction set, and converts them into binary Java class files, suitable for loading by a Java runtime system
- home page: `http://jasmin.sourceforge.net/`

Jasmin

- will present Jasmin JVM assembly-language
- Jasmin is an assembler for the JVM
- It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the JVM instruction set, and converts them into binary Java class files, suitable for loading by a Java runtime system
- home page: `http://jasmin.sourceforge.net/`

Jasmin

- will present Jasmin JVM assembly-language
- Jasmin is an assembler for the JVM
- It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the JVM instruction set, and converts them into binary Java class files, suitable for loading by a Java runtime system
- home page: `http://jasmin.sourceforge.net/`

Jasmin

- will present Jasmin JVM assembly-language
- Jasmin is an assembler for the JVM
- It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the JVM instruction set, and converts them into binary Java class files, suitable for loading by a Java runtime system
- home page: `http://jasmin.sourceforge.net/`

The JVM Stack Machine

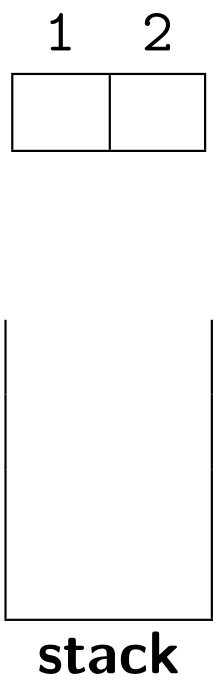
```
i = 10;
k = i + (i * 5) - 2;
```

The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

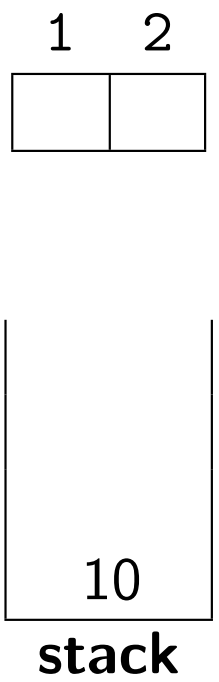


The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

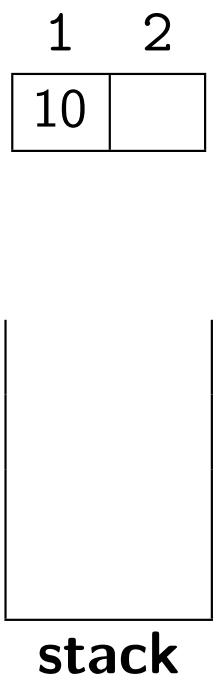


The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

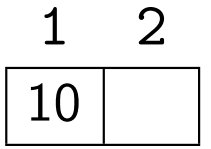


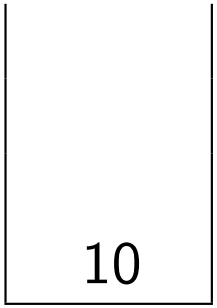
The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```





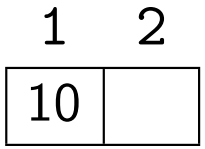
stack

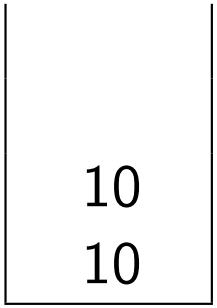
The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```





stack

The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

1	2
10	

5
10
10

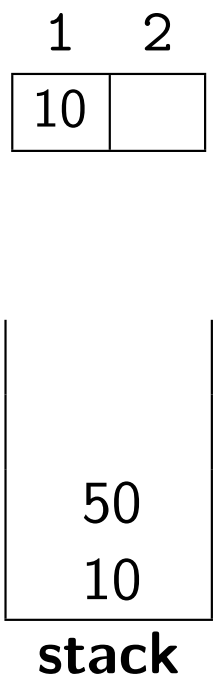
stack

The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

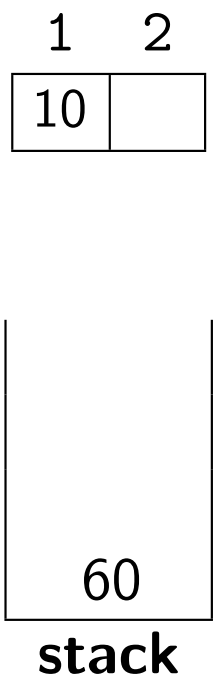


The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

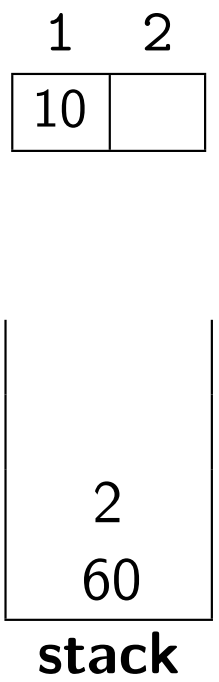


The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

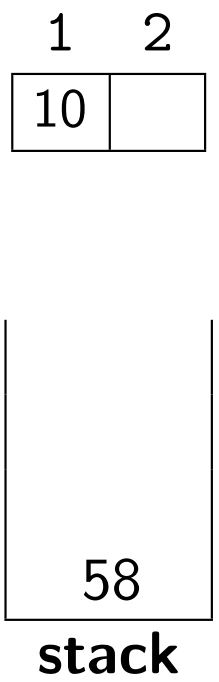


The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

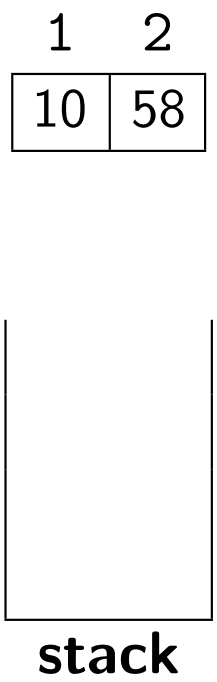


The JVM Stack Machine

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

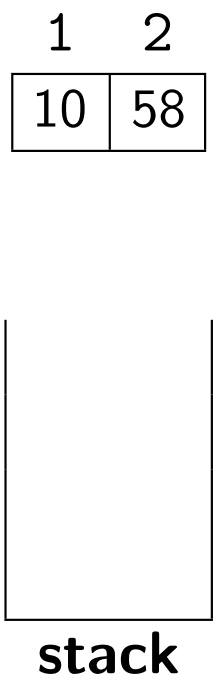


The JVM Stack Machine

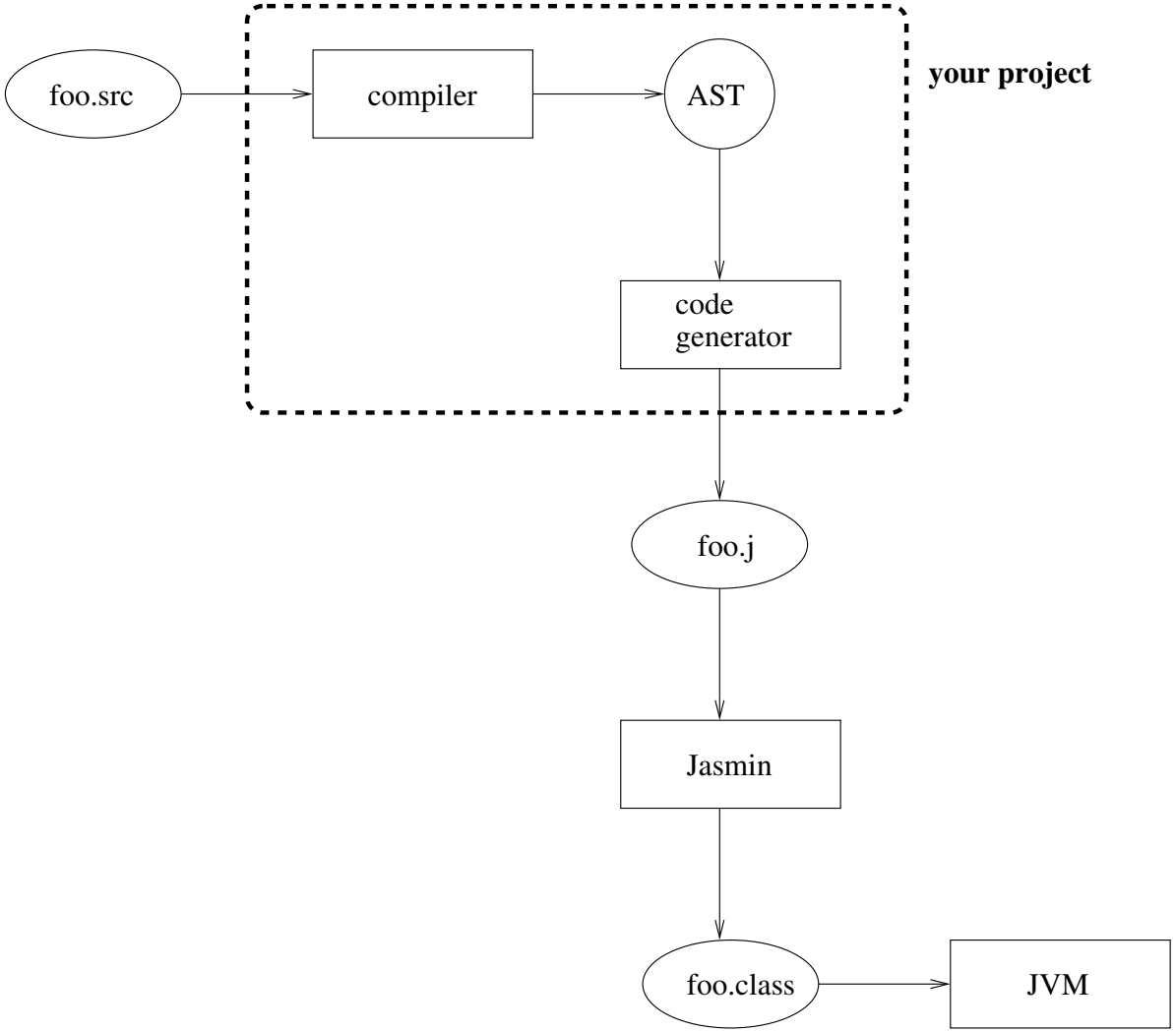
```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```



Source Code to Execution



Outline

Storage Allocation

- assign addresses (offsets) for all variables
- count number of local variables in each scope
- compute stack space required (more on this later)
- information can be stored in symbol table or computed 'on the fly'

Storage Allocation

- assign addresses (offsets) for all variables
- count number of local variables in each scope
- compute stack space required (more on this later)
- information can be stored in symbol table or computed 'on the fly'

Storage Allocation

- assign addresses (offsets) for all variables
- count number of local variables in each scope
- compute stack space required (more on this later)
- information can be stored in symbol table or computed 'on the fly'

Storage Allocation

- assign addresses (offsets) for all variables
- count number of local variables in each scope
- compute stack space required (more on this later)
- information can be stored in symbol table or computed 'on the fly'

Code Generation

- will traverse the AST outputting bytecode asm instructions
- create one class per instruction; use `toString()` method to produce asm listing
- more on this later

Code Generation

- will traverse the AST outputting bytecode asm instructions
- create one class per instruction; use `toString()` method to produce asm listing
- more on this later

Code Generation

- will traverse the AST outputting bytecode asm instructions
- create one class per instruction; use `toString()` method to produce asm listing
- more on this later

Outline

Runtime Data Areas

pc register program counter, one per thread (essentially the only register in the JVM)

Java stack stores stack frames, one per thread

heap stores all dynamically created objects & arrays, one heap per JVM, uses garbage collection

method area stores compiled code, one per JVM

constant pool stores constants including program literals and classes, methods, objects

native stacks stacks for methods written in other languages

Runtime Data Areas

pc register program counter, one per thread (essentially the only register in the JVM)

Java stack stores stack frames, one per thread

heap stores all dynamically created objects & arrays, one heap per JVM, uses garbage collection

method area stores compiled code, one per JVM

constant pool stores constants including program literals and classes, methods, objects

native stacks stacks for methods written in other languages

Runtime Data Areas

pc register program counter, one per thread (essentially the only register in the JVM)

Java stack stores stack frames, one per thread

heap stores all dynamically created objects & arrays, one heap per JVM, uses garbage collection

method area stores compiled code, one per JVM

constant pool stores constants including program literals and classes, methods, objects

native stacks stacks for methods written in other languages

Runtime Data Areas

pc register program counter, one per thread (essentially the only register in the JVM)

Java stack stores stack frames, one per thread

heap stores all dynamically created objects & arrays, one heap per JVM, uses garbage collection

method area stores compiled code, one per JVM

constant pool stores constants including program literals and classes, methods, objects

native stacks stacks for methods written in other languages

Runtime Data Areas

pc register program counter, one per thread (essentially the only register in the JVM)

Java stack stores stack frames, one per thread

heap stores all dynamically created objects & arrays, one heap per JVM, uses garbage collection

method area stores compiled code, one per JVM

constant pool stores constants including program literals and classes, methods, objects

native stacks stacks for methods written in other languages

Runtime Data Areas

pc register	program counter, one per thread (essentially the only register in the JVM)
Java stack	stores stack frames, one per thread
heap	stores all dynamically created objects & arrays, one heap per JVM, uses garbage collection
method area	stores compiled code, one per JVM
constant pool	stores constants including program literals and classes, methods, objects
native stacks	stacks for methods written in other languages

Stack Frame

- We talk of the current ...

  frame stack frame for the executing method
  method executing method
  class class containing executing method

- **local variables**: "array" of 32-bit words that hold local (method) variables; accessed by word index (i.e. the variable *n* is thought of being stored at *variables*[*n*]); long/double allocated 2 words
- **operand stack**: holds operands & results for instructions; also used to pass method arguments & store returned values; each method has its own, isolated operand stack

Stack Frame

- We talk of the current ...

  frame stack frame for the executing method
  method executing method
  class class containing executing method

- **local variables**: "array" of 32-bit words that hold local (method) variables; accessed by word index (i.e. the variable *n* is thought of being stored at *variables*[*n*]); long/double allocated 2 words
- **operand stack**: holds operands & results for instructions; also used to pass method arguments & store returned values; each method has its own, isolated operand stack

Stack Frame

- We talk of the current ...

  frame stack frame for the executing method
  method executing method
  class class containing executing method

- **local variables**: "array" of 32-bit words that hold local (method) variables; accessed by word index (i.e. the variable *n* is thought of being stored at *variables*[*n*]); long/double allocated 2 words
- **operand stack**: holds operands & results for instructions; also used to pass method arguments & store returned values; each method has its own, isolated operand stack

Stack Frame

- We talk of the current ...

frame	stack frame for the executing method
method	executing method
class	class containing executing method

- **local variables**: "array" of 32-bit words that hold local (method) variables; accessed by word index (i.e. the variable *n* is thought of being stored at *variables*[*n*]); long/double allocated 2 words
- **operand stack**: holds operands & results for instructions; also used to pass method arguments & store returned values; each method has its own, isolated operand stack

Stack Frame

- We talk of the current ...

frame	stack frame for the executing method
method	executing method
class	class containing executing method

- **local variables**: "array" of 32-bit words that hold local (method) variables; accessed by word index (i.e. the variable *n* is thought of being stored at *variables*[*n*]); long/double allocated 2 words
- **operand stack**: holds operands & results for instructions; also used to pass method arguments & store returned values; each method has its own, isolated operand stack

Stack Frame

- We talk of the current ...

frame	stack frame for the executing method
method	executing method
class	class containing executing method

- **local variables**: "array" of 32-bit words that hold local (method) variables; accessed by word index (i.e. the variable n is thought of being stored at $variables[n]$); long/double allocated 2 words
- **operand stack**: holds operands & results for instructions; also used to pass method arguments & store returned values; each method has its own, isolated operand stack

Outline

HelloWorld.j

```
public class HelloWorld {
  public static void main(String [] args) {
   System.out.println("hello world!");
  }
}
```

```
.source HelloWorld.java
.class  public HelloWorld
.super  java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "hello world!"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    return
.end method
```

Primitive Data Types

Types	Size (bits)	Descriptor
byte	8	B
char	16	C
short	16	S
int	32	I
long	64	J
float	32	F
double	64	D

- booleans are represented as integers (with the optional descriptor Z)
- word size: 32 bits (i.e. size of local variables & operand stack)
- `void` type descriptor: V

Primitive Data Types

Types	Size (bits)	Descriptor
byte	8	B
char	16	C
short	16	S
int	32	I
long	64	J
float	32	F
double	64	D

- booleans are represented as integers (with the optional descriptor Z)
- word size: 32 bits (i.e. size of local variables & operand stack)
- `void` type descriptor: V

Primitive Data Types

Types	Size (bits)	Descriptor
byte	8	B
char	16	C
short	16	S
int	32	I
long	64	J
float	32	F
double	64	D

- booleans are represented as integers (with the optional descriptor Z)
- word size: 32 bits (i.e. size of local variables & operand stack)
- `void` type descriptor: V

Primitive Data Types

Types	Size (bits)	Descriptor
byte	8	B
char	16	C
short	16	S
int	32	I
long	64	J
float	32	F
double	64	D

- booleans are represented as integers (with the optional descriptor Z)
- word size: 32 bits (i.e. size of local variables & operand stack)
- `void` type descriptor: V

Structured Types

Type	Form	Examples	equiv. in Java
Object	`L`*classname*`;`	`Ljava/lang/Object;`	java.lang.Object
		`Ljava/lang/String;`	java.lang.String
		`LHelloWorld;`	HelloWorld
Array	`[`*type*	`[I`	`int []`
		`[F`	`float []`
		`[[D`	`double [][]`
		`[Ljava/lang/String;`	`String []`

Method Type Descriptors

- Syntax: *(<parameter descriptor>)<return descriptor>*
- Examples:
  - in Java: `void main(String[]args)`
    in JVM: `main([Ljava/lang/String;)V`
  - in Java: `int[][] m(int i, double[] d, int[] i1)`
    in JVM: `m(I[D[I)[[I`

Method Type Descriptors

- Syntax: *(<parameter descriptor>)<return descriptor>*
- Examples:

in Java: `void main(String[]args)`
in JVM: `main([Ljava/lang/String;)V`

in Java: `Object m1(int i, double d, String s)`
in JVM: `m1(IDLjava/lang/String;)Ljava/lang/Object;`

Method Type Descriptors

- Syntax: *(<parameter descriptor>)<return descriptor>*
- Examples:

  in Java: `void main(String[]args)`
  in JVM: `main([Ljava/lang/String;)V`

  in Java: `Object m1(int i, double d, String s)`
  in JVM: `m1(IDLjava/lang/String;)Ljava/lang/Object;`

Method Type Descriptors

- Syntax: *(<parameter descriptor>)<return descriptor>*
- Examples:

  in Java: `void main(String[]args)`
  in JVM: `main([Ljava/lang/String;)V`

  in Java: `Object m1(int i, double d, String s)`
  in JVM: `m1(IDLjava/lang/String;)Ljava/lang/Object;`

Method Type Descriptors

- Syntax: *(<parameter descriptor>)<return descriptor>*
- Examples:

in Java: `void main(String[]args)`
in JVM: `main([Ljava/lang/String;)V`

in Java: `Object m1(int i, double d, String s)`
in JVM: `m1(IDLjava/lang/String;)Ljava/lang/Object;`

Method Type Descriptors

- Syntax: *(<parameter descriptor>)<return descriptor>*
- Examples:

in Java: `void main(String[]args)`
in JVM: `main([Ljava/lang/String;)V`

in Java: `Object m1(int i, double d, String s)`
in JVM: `m1(IDLjava/lang/String;)Ljava/lang/Object;`

Instruction Set

- over 160 instructions (opcode occupies one byte ∴ 256 possible opcodes)
- not orthogonal – operations provided for one type not necessarily provided for all types (there are not enough opcodes to offer the same support for all of Java's types)

  (cf. http://en.wikipedia.org/wiki/Orthogonal_instruction_set)
- Prefix codes for instructions:

Type	Code
int	i
long	l
float	f
double	d
byte	b
char	c
short	s
reference	a

Instruction Set

- over 160 instructions (opcode occupies one byte ∴ 256 possible opcodes)
- not orthogonal – operations provided for one type not necessarily provided for all types (there are not enough opcodes to offer the same support for all of Java's types)

  (cf. `http://en.wikipedia.org/wiki/Orthogonal_instruction_set`)
- Prefix codes for instructions:

Type	Code
int	`i`
long	`l`
float	`f`
double	`d`
byte	`b`
char	`c`
short	`s`
reference	`a`

Instruction Set

- over 160 instructions (opcode occupies one byte ∴ 256 possible opcodes)
- not orthogonal – operations provided for one type not necessarily provided for all types (there are not enough opcodes to offer the same support for all of Java's types)

  (cf. `http://en.wikipedia.org/wiki/Orthogonal_instruction_set`)
- Prefix codes for instructions:

Type	**Code**
int	`i`
long	`l`
float	`f`
double	`d`
byte	`b`
char	`c`
short	`s`
reference	`a`

Instruction Set

	int	long	float	double	byte	char	short	reference
?2c	✓							
?2d	✓	✓	✓					
?2i		✓	✓	✓				
?2f	✓	✓		✓				
?2l	✓		✓	✓				
?2s	✓							
?add	✓	✓	✓	✓				
?aload	✓	✓	✓	✓	✓	✓	✓	✓
?and	✓	✓						
?astore	✓	✓	✓	✓	✓	✓	✓	✓
?cmp		✓						
?cmp{g|1}			✓	✓				
?const_<n>	✓	✓	✓	✓				✓

Instruction Set

	int	long	float	double	byte	char	short	reference
?div	✓	✓	✓	✓				
?inc	✓							
?ipush					✓		✓	
?load	✓	✓	✓	✓				
?mul	✓	✓	✓	✓				
?neg	✓	✓	✓	✓				
?newarray								✓
?or	✓	✓						
?rem	✓	✓	✓	✓				
?return	✓	✓	✓	✓				✓
?shl	✓	✓						
?shr	✓	✓						
?store	✓	✓	✓	✓				✓
?sub	✓	✓	✓	✓				
?throw								✓
?ushr	✓	✓						
?xor	✓	✓						

Relational Expressions

- Example:

```
a = 10;
x = a <= 100;
```

- One solution – code for 'a <= 100':

```
        iload_1            ; load a
        bipush 100         ; push 100 onto stack
        if_icmpgt label_0  ; jump if 1st integer > 2nd
        iconst_1           ; push 1 onto stack
        goto label_1       ; jump to label_1
label_0:
        iconst_0           ; push 0 onto stack
label_1:
```

Relational Expressions

- Example:

```
a = 10;
x = a <= 100;
```

☜ what should this evaluate to?

- One solution – code for 'a <= 100':

```
        iload_1            ; load a
        bipush 100         ; push 100 onto stack
        if_icmpgt label_0  ; jump if 1st integer > 2nd
        iconst_1           ; push 1 onto stack
        goto label_1       ; jump to label_1
label_0:
        iconst_0           ; push 0 onto stack
label_1:
```

Relational Expressions

- Example:

```
a = 10;
x = a <= 100;
```

☞ what should this evaluate to? $x \leftarrow 1$

- One solution – code for 'a <= 100':

```
        iload_1            ; load a
        bipush 100         ; push 100 onto stack
        if_icmpgt label_0  ; jump if 1st integer > 2nd
        iconst_1           ; push 1 onto stack
        goto label_1       ; jump to label_1
label_0:
        iconst_0           ; push 0 onto stack
label_1:
```

Relational Expressions

- Example:

```
a = 10;
x = a <= 100;
```

☞ what should this evaluate to? $x \leftarrow 1$

- One solution – code for 'a <= 100':

```
          iload_1            ; load a
          bipush 100         ; push 100 onto stack
          if_icmpgt label_0  ; jump if 1st integer > 2nd
          iconst_1           ; push 1 onto stack
          goto label_1       ; jump to label_1
label_0:
          iconst_0           ; push 0 onto stack
label_1:
```

Flow-of-Control Instructions

Name	Description	# of operands
ifeq	jump if zero	pop one operand
ifnull	jump if null	
iflt	jump if $<$ zero	
ifle	jump if $<=$ zero	
ifne	jump if non-zero	
ifnonnull	jump if non-null	
ifgt	jump if $>$ zero	
ifge	jump if $>=$ zero	
if_icmpeq	jump if two integers are equal	pop two operands
if_icmpne	jump if two integers are not equal	
if_icmplt	jump if one integer is $<$ to another	
if_icmpgt	jump if one integer is $>$ to another	
if_icmple	jump if one integer is $<=$ to another	
if_icmpge	jump if one integer is $>=$ to another	

The Stack

`bipush` – push one-byte signed integer

- Syntax: `bipush` <*n*>
  where $-128 <= n <= 127$
- Stack:

before	after
...	<*n*>
...	...

The Stack

`iconst_<n>` – push integer constant 0, 1, 2, 3, 4, or 5

- Syntax:
  `iconst_0` or `iconst_1` or `iconst_2` or `iconst_3` or `iconst_4` or `iconst_5`
- Stack:

before	after
...	<*n*>
...	...

The Stack

`ldc` <*value*> – push single-word constant onto stack

- Syntax: `ldc` <*value*>
  where <*value*> is an int, a float, or a literal string
- Examples:

```
ldc "hello world"
ldc 298
ldc 1.5
```

- Stack:

before	after
...	<*value*>
...	...

The Stack

pop – discard top word on stack

- Syntax: pop
- Stack:

before	**after**
item	...
...	...

The Stack

`dup` – duplicate top single-word item on stack

- Syntax: `dup`
- Stack:

before	**after**
item	item
...	item

Local Variables

`iload` – retrieve integer from local variable

- Syntax: `iload` <*varnum*>
- Stack:

before	**after**
...	int-value
...	...

Local Variables

`iload_<n>` – retrieve integer from local variable <n>

- Syntax:
  `iload_0` or `iload_1` or `iload_2` or `iload_3`
- Stack:

before	after
...	int-value
...	...

Local Variables

`istore` – store integer in local variable

- Syntax: `istore` <*varnum*>
- Stack:

before	**after**
int-value	...
...	...

Local Variables

`istore_<n>` – store integer in local variable <n>

- Syntax:
  `istore_0` or `istore_1` or `istore_2` or `istore_3`
- Stack:

before	**after**
int-value	...
...	...

Objects

`new` – create an object

- Syntax: `new` <*class*>
- Example: `new java/lang/StringBuffer`
- Stack:

before	after
...	object_ref
...	...

- Before the new object can be used one of its <init> methods must be called:

```
new java/lang/StringBuffer
dup
invokespecial java/lang/StringBuffer/<init>()V
astore_1 ; assign object reference to local var
```

Objects

`getfield` – get value of object field

- Syntax: `getfield` <*field−spec*> <*descriptor*>
- Example: `getfield MyClass/field1 I`
- Stack:

before	**after**
object_ref	value
...	...

Objects

`putfield` – set value of object field

- Syntax: `putfield` <*field – spec*> <*descriptor*>
- Example: `putfield MyClass/field1 I`
- Stack:

before	**after**
value	...
object_ref	...

Objects

`getstatic` – get value of static field

- Syntax: `getstatic` <*field−spec*> <*descriptor*>
- Example:
  `getstatic java/lang/System/out Ljava/io/Printstream;`
- Stack:

before	**after**
...	value
...	...

Objects

`putstatic` – set value of object field

- Syntax: `putstatic` <*field−spec*> <*descriptor*>
- Example: `putstatic MyClass/staticfield1 I`
- Stack:

before	**after**
value	...
...	...

Integer Arithmetic

`iadd, isub, imul, idiv` – add/subtract/multiple/divide integers

- Syntax: `iadd` or `isub` or `imul` or `idiv`
- Stack:

before	**after**
value1	result
value2	...

- for `isub`: result = value2 − value1
  for `idiv`: result = value2 div value1

Integer Arithmetic

`iadd, isub, imul, idiv` – add/subtract/multiple/divide integers

- Syntax: `iadd` or `isub` or `imul` or `idiv`
- Stack:

before	after
value1	result
value2	...

☞ top of stack

- for `isub`: result = value2 − value1
  for `idiv`: result = value2 div value1

Integer Arithmetic

`iadd`, `isub`, `imul`, `idiv` – add/subtract/multiple/divide integers

- Syntax: `iadd` or `isub` or `imul` or `idiv`
- Stack:

before	after
value1	result
value2	...

☞ top of stack

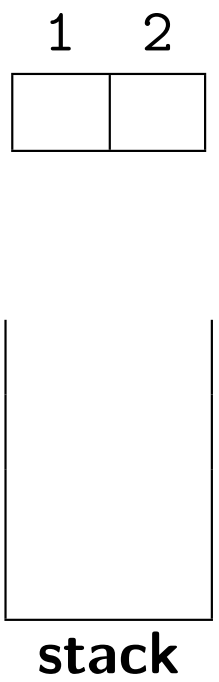
- for `isub`: result = value2 − value1
  for `idiv`: result = value2 div value1

Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

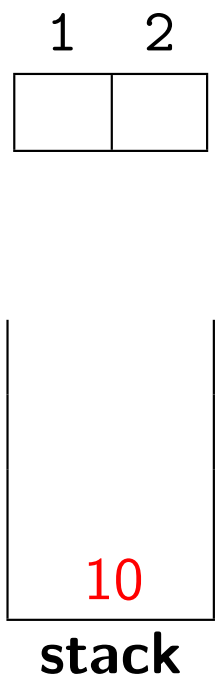


Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

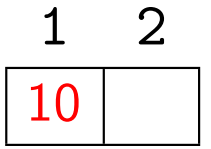


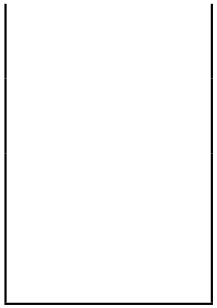
Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```



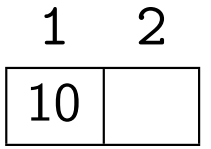


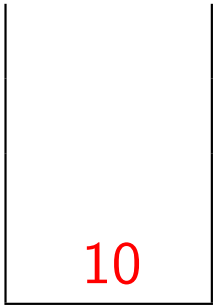
Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```





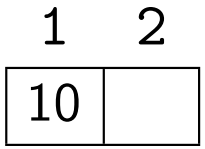
stack

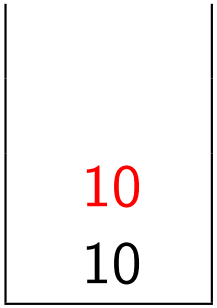
Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```





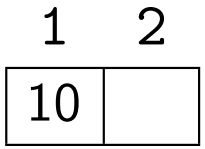
stack

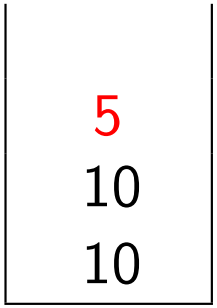
Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```





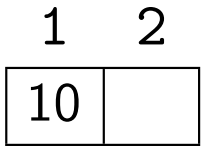
stack

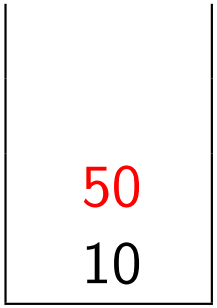
Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```





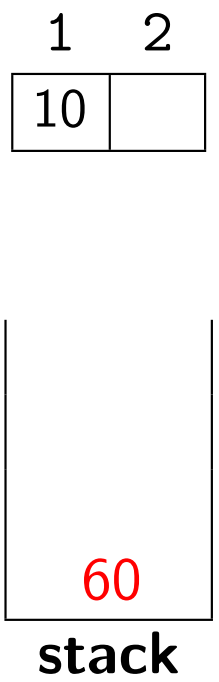
stack

Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

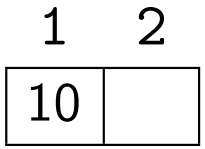


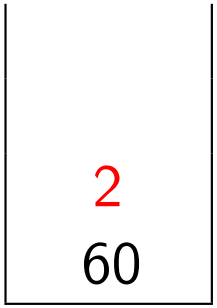
Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```





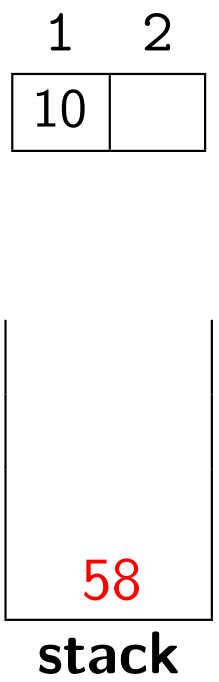
stack

Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

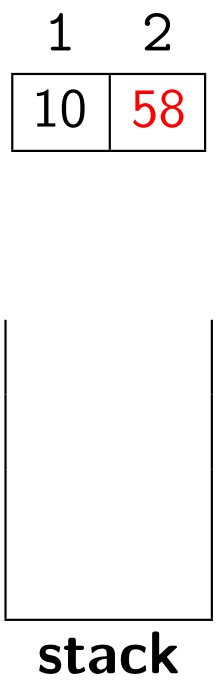


Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```

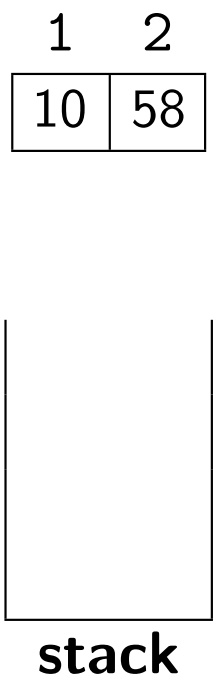


Integer Arithmetic Example (again)

```
i = 10;
k = i + (i * 5) - 2;
```

⇓

```
bipush 10
istore_1 // i
iload_1
iload_1
iconst_5
imul
iadd
iconst_2
isub
istore_2 // k
```



Conditional Branches

`ifeq` – jump if zero

- Syntax: `ifeq` <*label*>
- Example: `ifeq label_end`
- Stack:

before	**after**
int_value	...
...	...

- target of label defined as <*label*>:
- also `ifne`, `ifnull`, `ifnonnull`, `iflt`, `ifle`, `ifgt`, `ifge`

Conditional Branches

`if_icmpeq` – jump if two integers are equal

- Syntax: `if_icmpeq` <*label*>
- Example: `if_icmpeq label_end`
- Stack:

before	**after**
int_value	...
int_value	...

- target of label defined as <*label*>:
- also `if_icmpne`

Conditional Branches

`if_icmpgt` – jump if one integer is greater than another

- Syntax: `if_icmpgt` <*label*>
- Example: `if_icmpgt label_end`
- Stack:

before	after
int_value1	...
int_value2	...

☜ top of stack

- target of label defined as <*label*>:
- jump if value2 > value1
- also `if_icmplt`, `if_icmple`, `if_icmpge`

Unconditional Branches

`goto` – branch to address

- Syntax: `goto` <*label*>
- Example: `goto label_end`
- Stack:

before	**after**
...	...
...	...

- target of label defined as <*label*>:

Method Invocation

`invokevirtual` – call an instance method

- Syntax: `invokevirtual` <*method−spec*>
- Example:

  `invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z`
- parameters are pushed on stack in left-to-right order
- Stack:

before	**after**
argN	[result] ☜ top of stack
...	...
arg2	...
arg1	...
object_ref	...
...	...

Method Invocation

`invokespecial` – invoke constructor & private methods

- Syntax: `invokespecial` <*method−spec*>
- Example:

  ```
  invokespecial java/lang/StringBuffer/<init>()V
  ```

- parameters are pushed on stack in left-to-right order
- Stack:

before	**after**
argN	[result] ☜ top of stack
...	...
arg2	...
arg1	...
object_ref	...
...	...

Method Invocation

`invokestatic` – call a class (static) method

- Syntax: `invokestatic` <*method – spec*>
- Example:

  `invokestatic java/lang/System/exit(I)V`
- parameters are pushed on stack in left-to-right order
- Stack:

before	**after**
argN	[result] ☜ top of stack
...	...
arg2	...
arg1	...
...	...

Method Return

`return` – return from method

- Syntax: `return`
- simple return with no result
- all items on the current method's operand stack are discarded
- Stack:

before	**after**
...	...

Method Return

`ireturn` – return from method with integer result

- Syntax: `ireturn`
- pops an integer value and pushes it onto the operand stack of the invoker
- all items on the current method's operand stack are discarded
- Stack:

before	**after**
int_value	n/a

Overview

- Directive statements are used to give Jasmin meta-level information
- Syntax: .<*name*>
- Examples:

```
.catch        .line
.class        .method
.end          .source
.field        .super
.implements   .throws
.interface    .var
.limit
```

Overview

- Directive statements are used to give Jasmin meta-level information
- Syntax: .<*name*>
- Examples:

```
.catch        .line
.class        .method
.end          .source
.field        .super
.implements   .throws
.interface    .var
.limit
```

Overview

- Directive statements are used to give Jasmin meta-level information
- Syntax: .<*name*>
- Examples:

```
.catch        .line
.class        .method
.end          .source
.field        .super
.implements   .throws
.interface    .var
.limit
```

Outline

Example #1

```
.source Example1.java
.class  public Example1
.super  java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1
.line 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 3
.line 3
    iconst_2
    istore_1
    iconst_5
    istore_2
.line 4
    iload_1
    iload_2
    if_icmple Label1
.line 5
```

```
1 - public class Example1 {
2 -   public static void main(String [] args) {
3 -     int a = 2, b = 5;
4 -     if (a > b)
5 -       System.out.print("a ");
6 -     else
7 -       System.out.print("b ");
8 -     System.out.println("is bigger");
9 -   }
10- }
```

Example #1

```
...
4-   if (a > b)
5-     System.out.print("a ");
6 -   else
7 -     System.out.print("b ");
8 -   System.out.println("is bigger");
9 - }
...
```

```
.line 5
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "a "
  invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
  goto Label2
.line 7
Label1:
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "b "
  invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
.line 8
Label2:
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "is bigger"
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
.line 9
  return
.end method
```

Example #2

```
.source Example2.java
.class  public Example2
.super  java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1
.line 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 4
.line 3
    iconst_5
    istore_1
    iconst_1
    istore_2
    iconst_0
    istore_3
.line 4
```

```
1 - public class Example2 {
2 -  public static void main(String [] args) {
3 -    int n = 5, i = 1, sum = 0;
4 -    while (i <= n) {
5 -      sum += i;
6 -      i++;
7 -    }
8 -    System.out.println(sum);
9 -  }
10- }
```

Example #2

```
.line 4
Label1:
    iload_2
    iload_1
    if_icmpgt Label2
.line 5
    iload_3
    iload_2
    iadd
    istore_3
.line 6
    iinc 2 1
    goto Label1
.line 8
Label2:
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload_3
    invokevirtual java/io/PrintStream/println(I)V
.line 9
    return
.end method
```

```
1 - public class Example2 {
2 -  public static void main(String [] args) {
3 -    int n = 5, i = 1, sum = 0;
4 -    while (i <= n) {
5 -      sum += i;
6 -      i++;
7 -    }
8 -    System.out.println(sum);
9 -  }
10- }
```

Example #3

```
.source Example3.java
.class  public Example3
.super  java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1
.line 3
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 4
    .limit locals 4
.line 6
    new java/util/Scanner
    dup
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
    astore_1
.line 8
```

```
1 - import java.util.Scanner;
2 -
3 - public class Example3 {
4 -
5 -  public static void main(String [] args) {
6 -   Scanner s = new Scanner(System.in);
7 -   int a = s.nextInt(), b = s.nextInt();
8 -   System.out.println("Max:  " + max(a,b));
9 -  }
10-
11-  private static int max(int x, int y) {
12-   return (x > y ?  x :  y);
13-  }
14-
15- }
```

Example #3

```
...
5 -  public static void main(String [] args) {
6 -   Scanner s = new Scanner(System.in);
7 -   int a = s.nextInt(), b = s.nextInt();
8 -   System.out.println("Max:  " + max(a,b));
9 -  }
10-
...
```

```
.line 8
  aload_1
  invokevirtual java/util/Scanner/nextInt()I
  istore_2
  aload_1
  invokevirtual java/util/Scanner/nextInt()I
  istore_3
.line 9
  getstatic java/lang/System/out Ljava/io/PrintStream;
  new java/lang/StringBuilder
  dup
  invokespecial java/lang/StringBuilder/<init>()V
  ldc "Max: "
  invokevirtual java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/StringBuilder;
  iload_2
  iload_3
  invokestatic Example3/max(II)I
  invokevirtual java/lang/StringBuilder/append(I)Ljava/lang/StringBuilder;
  invokevirtual java/lang/StringBuilder/toString()Ljava/lang/String;
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
.line 10
  return
.end method
```

Example #3

```
.method private static max(II)I
    .limit stack 2
    .limit locals 2
.line 13
    iload_0
    iload_1
    if_icmple Label1
    iload_0
    goto Label2
Label1:
    iload_1
Label2:
    ireturn
.end method
```

```
1 - import java.util.Scanner;
2 -
3 - public class Example3 {
4 -
5 -  public static void main(String [] args) {
6 -   Scanner s = new Scanner(System.in);
7 -   int a = s.nextInt(), b = s.nextInt();
8 -   System.out.println("Max:  " + max(a,b));
9 -  }
10-
11-  private static int max(int x, int y) {
12-   return (x > y ?  x :  y);
13-  }
14-
15-  }
```

Outline