Java bytecode instruction listings

From Wikipedia, the free encyclopedia

Main article: Java bytecode

This is a list of the instructions that make up the Java bytecode, an abstract machine language that is ultimately executed by the Java virtual machine. The Java bytecode is generated by language compilers targeting the Java Platform, most notably the Java programming language.

Mnemonic	**Opcode *(in hex)***	**Other bytes**	**Stack [before]→ [after]**	**Description**
aaload	32		arrayref, index → value	load onto the stack a reference from an array
aastore	53		arrayref, index, value →	store into a reference in an array
aconst_null	01		→ null	push a *null* reference onto the stack
aload	19	1: index	→ objectref	load a reference onto the stack from a local variable *#index*
aload_0	2a		→ objectref	load a reference onto the stack from local variable 0
aload_1	2b		→ objectref	load a reference onto the stack from local variable 1
aload_2	2c		→ objectref	load a reference onto the stack from local variable 2
aload_3	2d		→ objectref	load a reference onto the stack from local variable 3
anewarray	bd	2: indexbyte1, indexbyte2	count → arrayref	create a new array of references of length *count* and component type identified by the class reference *index* (*indexbyte1 << 8 + indexbyte2*) in the constant pool
areturn	b0		objectref → [empty]	return a reference from a method
arraylength	be		arrayref → length	get the length of an array
astore	3a	1: index	objectref →	store a reference into a local variable *#index*
astore_0	4b		objectref →	store a reference into local variable 0
astore_1	4c		objectref →	store a reference into local variable 1
astore_2	4d		objectref →	store a reference into local variable 2
astore_3	4e		objectref →	store a reference into local variable 3
athrow	bf		objectref → [empty], objectref	throws an error or exception (notice that the rest of the stack is cleared, leaving only a reference to the Throwable)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
baload	33		arrayref, index → value	load a byte or Boolean value from an array
bastore	54		arrayref, index, value →	store a byte or Boolean value into an array
bipush	10	1: byte	→ value	push a *byte* onto the stack as an integer *value*
caload	34		arrayref, index → value	load a char from an array
castore	55		arrayref, index, value →	store a char into an array
checkcast	c0	2: indexbyte1, indexbyte2	objectref → objectref	checks whether an *objectref* is of a certain type, the class reference of which is in the constant pool at *index* (*indexbyte1 << 8 + indexbyte2*)
d2f	90		value → result	convert a double to a float
d2i	8e		value → result	convert a double to an int
d2l	8f		value → result	convert a double to a long
dadd	63		value1, value2 → result	add two doubles
daload	31		arrayref, index → value	load a double from an array
dastore	52		arrayref, index, value →	store a double into an array
dcmpg	98		value1, value2 → result	compare two doubles
dcmpl	97		value1, value2 → result	compare two doubles
dconst_0	0e		→ 0.0	push the constant *0.0* onto the stack
dconst_1	0f		→ 1.0	push the constant *1.0* onto the stack
ddiv	6f		value1, value2 → result	divide two doubles
dload	18	1: index	→ value	load a double *value* from a local variable *#index*
dload_0	26		→ value	load a double from local variable 0
dload_1	27		→ value	load a double from local variable 1
dload_2	28		→ value	load a double from local variable 2

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
dload_3	29		→ value	load a double from local variable 3
dmul	6b		value1, value2 → result	multiply two doubles
dneg	77		value → result	negate a double
drem	73		value1, value2 → result	get the remainder from a division between two doubles
dreturn	af		value → [empty]	return a double from a method
dstore	39	1: index	value →	store a double *value* into a local variable *#index*
dstore_0	47		value →	store a double into local variable 0
dstore_1	48		value →	store a double into local variable 1
dstore_2	49		value →	store a double into local variable 2
dstore_3	4a		value →	store a double into local variable 3
dsub	67		value1, value2 → result	subtract a double from another
dup	59		value → value, value	duplicate the value on top of the stack
dup_x1	5a		value2, value1 → value1, value2, value1	insert a copy of the top value into the stack two values from the top. value1 and value2 must not be of the type double or long.
dup_x2	5b		value3, value2, value1 → value1, value3, value2, value1	insert a copy of the top value into the stack two (if value2 is double or long it takes up the entry of value3, too) or three values (if value2 is neither double nor long) from the top
dup2	5c		{value2, value1} → {value2, value1}, {value2, value1}	duplicate top two stack words (two values, if value1 is not double nor long; a single value, if value1 is double or long)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
dup2_x1	5d		value3, {value2, value1} → {value2, value1}, value3, {value2, value1}	duplicate two words and insert beneath third word (see explanation above)
dup2_x2	5e		{value4, value3}, {value2, value1} → {value2, value1}, {value4, value3}, {value2, value1}	duplicate two words and insert beneath fourth word
f2d	8d		value → result	convert a float to a double
f2i	8b		value → result	convert a float to an int
f2l	8c		value → result	convert a float to a long
fadd	62		value1, value2 → result	add two floats
faload	30		arrayref, index → value	load a float from an array
fastore	51		arrayref, index, value →	store a float in an array
fcmpg	96		value1, value2 → result	compare two floats
fcmpl	95		value1, value2 → result	compare two floats
fconst_0	0b		→ 0.0f	push *0.0f* on the stack
fconst_1	0c		→ 1.0f	push *1.0f* on the stack
fconst_2	0d		→ 2.0f	push *2.0f* on the stack
fdiv	6e		value1, value2 → result	divide two floats
fload	17	1: index	→ value	load a float *value* from a local variable *#index*
fload_0	22		→ value	load a float *value* from local variable 0
fload_1	23		→ value	load a float *value* from local variable 1
fload_2	24		→ value	load a float *value* from local variable 2
fload_3	25		→ value	load a float *value* from local variable 3
fmul	6a		value1, value2 → result	multiply two floats
fneg	76		value → result	negate a float

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
frem	72		value1, value2 → result	get the remainder from a division between two floats
freturn	ae		value → [empty]	return a float
fstore	38	1: index	value →	store a float *value* into a local variable *#index*
fstore_0	43		value →	store a float *value* into local variable 0
fstore_1	44		value →	store a float *value* into local variable 1
fstore_2	45		value →	store a float *value* into local variable 2
fstore_3	46		value →	store a float *value* into local variable 3
fsub	66		value1, value2 → result	subtract two floats
getfield	b4	2: index1, index2	objectref → value	get a field *value* of an object *objectref*, where the field is identified by field reference in the constant pool *index* (*index1 << 8 + index2*)
getstatic	b2	2: index1, index2	→ value	get a static field *value* of a class, where the field is identified by field reference in the constant pool *index* (*index1 << 8 + index2*)
goto	a7	2: branchbyte1, branchbyte2	[no change]	goes to another instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
goto_w	c8	4: branchbyte1, branchbyte2, branchbyte3, branchbyte4	[no change]	goes to another instruction at *branchoffset* (signed int constructed from unsigned bytes *branchbyte1 << 24* + branchbyte2 << 16 + *branchbyte3 << 8 + branchbyte4*)
i2b	91		value → result	convert an int into a byte
i2c	92		value → result	convert an int into a character

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
i2d	87		value → result	convert an int into a double
i2f	86		value → result	convert an int into a float
i2l	85		value → result	convert an int into a long
i2s	93		value → result	convert an int into a short
iadd	60		value1, value2 → result	add two ints
iaload	2e		arrayref, index → value	load an int from an array
iand	7e		value1, value2 → result	perform a bitwise and on two integers
iastore	4f		arrayref, index, value →	store an int into an array
iconst_m1	02		→ -1	load the int value -1 onto the stack
iconst_0	03		→ 0	load the int value 0 onto the stack
iconst_1	04		→ 1	load the int value 1 onto the stack
iconst_2	05		→ 2	load the int value 2 onto the stack
iconst_3	06		→ 3	load the int value 3 onto the stack
iconst_4	07		→ 4	load the int value 4 onto the stack
iconst_5	08		→ 5	load the int value 5 onto the stack
idiv	6c		value1, value2 → result	divide two integers
if_acmpeq	a5	2: branchbyte1, branchbyte2	value1, value2 →	if references are equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_acmpne	a6	2: branchbyte1, branchbyte2	value1, value2 →	if references are not equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
if_icmpeq	9f	2: branchbyte1, branchbyte2	value1, value2 →	if ints are equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_icmpne	a0	2: branchbyte1, branchbyte2	value1, value2 →	if ints are not equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_icmplt	a1	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is less than *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_icmpge	a2	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is greater than or equal to *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_icmpgt	a3	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is greater than *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_icmple	a4	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is less than or equal to *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifeq	99	2: branchbyte1, branchbyte2	value →	if *value* is 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
ifne	9a	2: branchbyte1, branchbyte2	value →	if *value* is not 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
iflt	9b	2: branchbyte1, branchbyte2	value →	if *value* is less than 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifge	9c	2: branchbyte1, branchbyte2	value →	if *value* is greater than or equal to 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifgt	9d	2: branchbyte1, branchbyte2	value →	if *value* is greater than 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifle	9e	2: branchbyte1, branchbyte2	value →	if *value* is less than or equal to 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifnonnull	c7	2: branchbyte1, branchbyte2	value →	if *value* is not null, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifnull	c6	2: branchbyte1, branchbyte2	value →	if *value* is null, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
iinc	84	2: index, const	[No change]	increment local variable *#index* by signed byte *const*

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
iload	15	1: index	→ value	load an int *value* from a local variable *#index*
iload_0	1a		→ value	load an int *value* from local variable 0
iload_1	1b		→ value	load an int *value* from local variable 1
iload_2	1c		→ value	load an int *value* from local variable 2
iload_3	1d		→ value	load an int *value* from local variable 3
imul	68		value1, value2 → result	multiply two integers
ineg	74		value → result	negate int
instanceof	c1	2: indexbyte1, indexbyte2	objectref → result	determines if an object *objectref* is of a given type, identified by class reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
invokedynamic	ba	4: indexbyte1, indexbyte2, 0, 0	[arg1, [arg2 ...]] →	invokes a dynamic method identified by method reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
invokeinterface	b9	4: indexbyte1, indexbyte2, count, 0	objectref, [arg1, arg2, ...] →	invokes an interface method on object *objectref*, where the interface method is identified by method reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
invokespecial	b7	2: indexbyte1, indexbyte2	objectref, [arg1, arg2, ...] →	invoke instance method on object *objectref*, where the method is identified by method reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
invokestatic	b8	2: indexbyte1, indexbyte2	[arg1, arg2, ...] →	invoke a static method, where the method is identified by method reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
invokevirtual	b6	2: indexbyte1, indexbyte2	objectref, [arg1, arg2, ...] →	invoke virtual method on object *objectref*, where the method is identified by method reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
ior	80		value1, value2 → result	bitwise int or
irem	70		value1, value2 → result	logical int remainder
ireturn	ac		value → [empty]	return an integer from a method
ishl	78		value1, value2 → result	int shift left
ishr	7a		value1, value2 → result	int arithmetic shift right
istore	36	1: index	value →	store int *value* into variable *#index*
istore_0	3b		value →	store int *value* into variable 0
istore_1	3c		value →	store int *value* into variable 1
istore_2	3d		value →	store int *value* into variable 2
istore_3	3e		value →	store int *value* into variable 3
isub	64		value1, value2 → result	int subtract
iushr	7c		value1, value2 → result	int logical shift right
ixor	82		value1, value2 → result	int xor
jsr	a8	2: branchbyte1, branchbyte2	→ address	jump to subroutine at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*) and place the return address on the stack

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
jsr_w	c9	4: branchbyte1, branchbyte2, branchbyte3, branchbyte4	→ address	jump to subroutine at *branchoffset* (signed int constructed from unsigned bytes *branchbyte1 << 24 + branchbyte2 << 16 + branchbyte3 << 8 + branchbyte4*) and place the return address on the stack
l2d	8a		value → result	convert a long to a double
l2f	89		value → result	convert a long to a float
l2i	88		value → result	convert a long to a int
ladd	61		value1, value2 → result	add two longs
laload	2f		arrayref, index → value	load a long from an array
land	7f		value1, value2 → result	bitwise and of two longs
lastore	50		arrayref, index, value →	store a long to an array
lcmp	94		value1, value2 → result	compare two longs values
lconst_0	09		→ 0L	push the long 0 onto the stack
lconst_1	0a		→ 1L	push the long 1 onto the stack
ldc	12	1: index	→ value	push a constant *#index* from a constant pool (String, int or float) onto the stack
ldc_w	13	2: indexbyte1, indexbyte2	→ value	push a constant *#index* from a constant pool (String, int or float) onto the stack (wide *index* is constructed as *indexbyte1 << 8 + indexbyte2*)
ldc2_w	14	2: indexbyte1, indexbyte2	→ value	push a constant *#index* from a constant pool (double or long) onto the stack (wide *index* is constructed as *indexbyte1 << 8 + indexbyte2*)
ldiv	6d		value1, value2 → result	divide two longs

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
lload	16	1: index	→ value	load a long value from a local variable *#index*
lload_0	1e		→ value	load a long value from a local variable 0
lload_1	1f		→ value	load a long value from a local variable 1
lload_2	20		→ value	load a long value from a local variable 2
lload_3	21		→ value	load a long value from a local variable 3
lmul	69		value1, value2 → result	multiply two longs
lneg	75		value → result	negate a long
lookupswitch	ab	4+: <0-3 bytes padding>, defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, npairs1, npairs2, npairs3, npairs4, match-offset pairs...	key →	a target address is looked up from a table using a key and execution continues from the instruction at that address
lor	81		value1, value2 → result	bitwise or of two longs
lrem	71		value1, value2 → result	remainder of division of two longs
lreturn	ad		value → [empty]	return a long value
lshl	79		value1, value2 → result	bitwise shift left of a long *value1* by *value2* positions
lshr	7b		value1, value2 → result	bitwise shift right of a long *value1* by *value2* positions
lstore	37	1: index	value →	store a long *value* in a local variable *#index*
lstore_0	3f		value →	store a long *value* in a local variable 0
lstore_1	40		value →	store a long *value* in a local variable 1
lstore_2	41		value →	store a long *value* in a local variable 2
lstore_3	42		value →	store a long *value* in a local variable 3

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→ [after]	Description
lsub	65		value1, value2 → result	subtract two longs
lushr	7d		value1, value2 → result	bitwise shift right of a long *value1* by *value2* positions, unsigned
lxor	83		value1, value2 → result	bitwise exclusive or of two longs
monitorenter	c2		objectref →	enter monitor for object ("grab the lock" - start of synchronized() section)
monitorexit	c3		objectref →	exit monitor for object ("release the lock" - end of synchronized() section)
multianewarray	c5	3: indexbyte1, indexbyte2, dimensions	count1, [count2,...] → arrayref	create a new array of *dimensions* dimensions with elements of type identified by class reference in constant pool *index* (*indexbyte1* << *8* + *indexbyte2*); the sizes of each dimension is identified by *count1*, [*count2*, etc.]
new	bb	2: indexbyte1, indexbyte2	→ objectref	create new object of type identified by class reference in constant pool *index* (*indexbyte1* << *8* + *indexbyte2*)
newarray	bc	1: atype	count → arrayref	create new array with *count* elements of primitive type identified by *atype*
nop	00		[No change]	perform no operation
pop	57		value →	discard the top value on the stack
pop2	58		{value2, value1} →	discard the top two values on the stack (or one value, if it is a double or long)
putfield	b5	2: indexbyte1, indexbyte2	objectref, value →	set field to *value* in an object *objectref*, where the field is identified by a field reference *index* in constant pool (*indexbyte1* << *8* + *indexbyte2*)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
putstatic	b3	2: indexbyte1, indexbyte2	value →	set static field to *value* in a class, where the field is identified by a field reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
ret	a9	1: index	[No change]	continue execution from address taken from a local variable *#index* (the asymmetry with jsr is intentional)
return	b1		→ [empty]	return void from method
saload	35		arrayref, index → value	load short from array
sastore	56		arrayref, index, value →	store short to array
sipush	11	2: byte1, byte2	→ value	push a short onto the stack
swap	5f		value2, value1 → value1, value2	swaps two top words on the stack (note that value1 and value2 must not be double or long)
tableswitch	aa	4+: [0-3 bytes padding], defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, lowbyte1, lowbyte2, lowbyte3, lowbyte4, highbyte1, highbyte2, highbyte3, highbyte4, jump offsets...	index →	continue execution from an address in the table at offset *index*
wide	c4	3/5: opcode, indexbyte1, indexbyte2 or iinc, indexbyte1, indexbyte2, countbyte1, countbyte2	[same as for corresponding instructions]	execute *opcode*, where *opcode* is either iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore, or ret, but assume the *index* is 16 bit; or execute iinc, where the *index* is 16 bits and the constant to increment by is a signed 16 bit short
breakpoint	ca			reserved for breakpoints in Java debuggers; should not appear in any class file
impdep1	fe			reserved for implementation-dependent operations within debuggers; should not appear in any class file
impdep2	ff			reserved for implementation-dependent operations within debuggers; should not appear in any class file
(no name)	cb-fd			these values are currently unassigned for opcodes and are reserved for future use

See also

- Java bytecode, a general description of the java bytecode within the context of the JVM
- ARM9E, a CPU family with direct Java bytecode execution ability
- Common Intermediate Language (CIL), a similar bytecode specification that runs on the CLR of the .NET Framework.
- C to Java Virtual Machine compilers

External links

- Sun's Java Virtual Machine Specification (http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Java_bytecode_instruction_listings&oldid=489706690"
Categories: Java platform | Instruction set listings

- This page was last modified on 29 April 2012 at 00:21.
-