The JVM bytecode ISA

© 2015 Oracle, Mario Wolczko

Constant pool:

① memory efficiency: do not need to store constants multiple times in the bytecode.

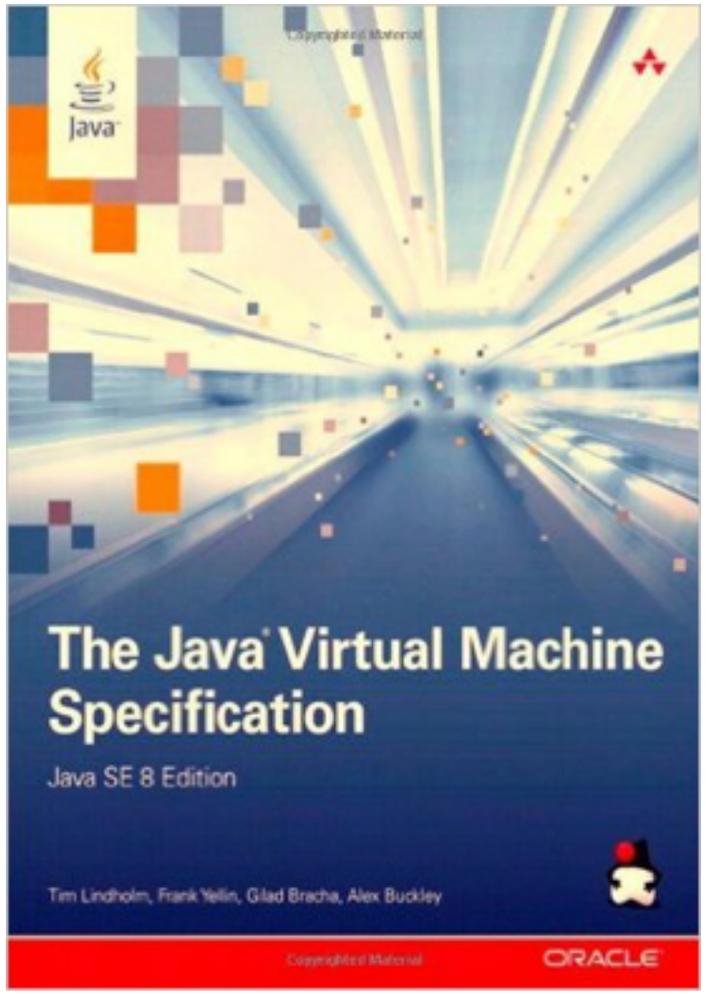
② type safety & verification: store type information along with constants.

③ symbolic references: classes, methods, Fields

- Let's take a look through the JVM bytecode set.

- The entire set is too big to examine in detail; we will point out some salient features and choices.

- The quoted material is taken from the Java SE 8 JVM spec (see bibliography).



Bytecode categorization

- Arithmetic
- Memory (locals, fields, statics, arrays): access and object creation, synchronization
- Stack manipulation
- Type checking and conversion
- Control (comparisons and branches, method call and return, exceptions)

2.11.3 Arithmetic Instructions

- Add: *iadd*, *ladd*, *fadd*, *dadd*.
- Subtract: *isub*, *lsub*, *fsub*, *dsub*.
- Multiply: *imul*, *lmul*, *fmul*, *dmul*.
- Divide: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Remainder: *irem*, *lrem*, *frem*, *drem*.
- Negate: *ineg*, *lneg*, *fneg*, *dneg*.
- Shift: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
- Bitwise OR: *ior*, *lor*.
- Bitwise AND: *iand*, *land*.
- Bitwise exclusive OR: *ixor*, *lxor*.
- Local variable increment: *iinc*.
- Comparison: *dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

2.11.3 Arithmetic Instructions

- Add: *iadd*, *ladd*, *fadd*, *dadd*.
- Subtract: *isub*, *lsub*, *fsub*, *dsub*.
- Multiply: *imul*, *lmul*, *fmul*, *dmul*.
- Divide: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Remainder: *irem*, *lrem*, *frem*, *drem*.
- Negate: *ineg*, *lneg*, *fneg*, *dneg*.
- Shift: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
- Bitwise OR: *ior*, *lor*.
- Bitwise AND: *iand*, *land*.
- Bitwise exclusive OR: *ixor*, *lxor*.
- Local variable increment: *iinc*.
- Comparison: *dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

NaN handling variants

iadd

Operation	Add `int`
Format	*iadd*
Forms	*iadd* = 96 (0x60)
Operand Stack	..., *value1*, *value2* → ..., *result*
Description	Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack. The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values. Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

iadd

type is part of the opcode

Operation Add `float`

Format

fadd

Forms *fadd* = 98 (0x62)

Operand Stack ..., *value1* , *value2* →
..., *result*

Description Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and undergo value set conversion (§2.8.3), resulting in *value1*' and *value2*'. The `float` *result* is *value1*' + *value2*'. The *result* is pushed onto the operand stack.

The result of an *fadd* instruction is governed by the rules of IEEE arithmetic:

- If either *value1*' or *value2*' is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If

continued…

ladd

ladd

Operation Add `long`

Format

ladd

Forms *ladd* = 97 (0x61)

Operand Stack ..., *value1*, *value2* →
..., *result*

Description Both *value1* and *value2* must be of type `long`. The values are popped from the operand stack. The `long` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

more on stack layout later…

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `long`. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *ladd* instruction never throws a run-time exception.

Operation Add `double`

Format

dadd

Forms *dadd* = 99 (0x63)

Operand Stack
..., *value1* , *value2* →
..., *result*

Description Both *value1* and *value2* must be of type `double`. The values are popped from the operand stack and undergo value set conversion (§2.8.3), resulting in *value1'* and *value2'*. The `double` *result* is *value1'* + *value2'*. The *result* is pushed onto the operand stack.

The result of a *dadd* instruction is governed by the rules of IEEE arithmetic:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If

2.11.2 Load and Store Instructions

The load and store instructions transfer values between the local variables (§2.6.1) and the operand stack (§2.6.2) of a Java Virtual Machine frame (§2.6):

- Load a local variable onto the operand stack: *iload*, *iload_<n>*, *lload*, *lload_<n>*, *fload*, *fload_<n>*, *dload*, *dload_<n>*, *aload*, *aload_<n>*.
- Store a value from the operand stack into a local variable: *istore*, *istore_<n>*, *lstore*, *lstore_<n>*, *fstore*, *fstore_<n>*, *dstore*, *dstore_<n>*, *astore*, *astore_<n>*.
- Load a constant on to the operand stack: *bipush*, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *iconst_<i>*, *lconst_<l>*, *fconst_<f>*, *dconst_<d>*.
- Gain access to more local variables using a wider index, or to a larger immediate operand: *wide*.

aload

Operation Load `reference` from local variable

Format

aload
index

Forms *aload* = 25 (0x19)

Operand Stack ... →
..., *objectref*

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction (§*astore*) is intentional.

The *aload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

aload

objects are always accessed via *references* — width unspecified

locals are indexed by an 8-bit int

aload

Operation Load `reference` from local variable

Format

aload
index

Forms *aload* = 25 (0x19)

Operand Stack ... →
..., *objectref*

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction (§*astore*) is intentional.

The *aload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

aload

objects are always accessed via *references* — width unspecified

locals are indexed by an 8-bit int

more later!

wide *wide*

Operation Extend local variable index by additional bytes

Format 1

wide
<opcode>
indexbyte1
indexbyte2

→ load / store operation.

where *<opcode>* is one of *iload*, *fload*, *aload*, *lload*, *dload*, *istore*, *fstore*, *astore*, *lstore*, *dstore*, or *ret*

Format 2

wide
iinc
indexbyte1
indexbyte2
constbyte1
constbyte2

Forms *wide* = 196 (0xc4)

Operand Stack Same as modified instruction

Description

The *wide* instruction modifies the behavior of another instruction. It takes one of two formats, depending on the instruction being modified. The first form of the *wide* instruction modifies one of the instructions *iload*, *fload*, *aload*, *lload*, *dload*, *istore*, *fstore*, *astore*, *lstore*, *dstore*, or *ret* (§*iload*, §*fload*, §*aload*, §*lload*, §*dload*, §*istore*, §*fstore*, §*astore*, §*lstore*, §*dstore*, §*ret*). The second form applies only to the *iinc* instruction (§*iinc*).

In either case, the *wide* opcode itself is followed in the compiled code by the opcode of the instruction *wide* modifies. In either form, two unsigned bytes *indexbyte1* and *indexbyte2* follow the modified opcode and are assembled into a 16-bit unsigned index to a local variable in the current frame (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The calculated index must be an index into the local variable array of the current frame. Where the *wide* instruction modifies an *lload*, *dload*, *lstore*, or *dstore* instruction, the index following the calculated index (index + 1) must also be an index into the local variable array. In the second form, two immediate unsigned bytes *constbyte1* and *constbyte2* follow *indexbyte1* and *indexbyte2* in the code stream. Those bytes are also assembled into a signed 16-bit constant, where the constant is (*constbyte1* << 8) | *constbyte2*.

The widened bytecode operates as normal, except for the use of the wider index and, in the case of the second form, the larger increment range.

Notes

Although we say that *wide* "modifies the behavior of another instruction," the *wide* instruction effectively treats the bytes constituting the modified instruction as operands, denaturing the embedded instruction in the process. In the case of a modified *iinc* instruction, one of the logical operands of the *iinc* is not even at the normal offset from the opcode. The embedded instruction must never be executed directly; its opcode must never be the target of any control transfer instruction.

lstore_<n>

lstore_<n>

Operation Store `long` into local variable

Format

lstore_<n>

Forms

lstore_0 = 63 (0x3f)

lstore_1 = 64 (0x40)

lstore_2 = 65 (0x41)

lstore_3 = 66 (0x42)

longs and doubles take *two* local variables

Operand Stack ..., *value* →

...

Description Both <*n*> and <*n*>+1 must be indices into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at <*n*> and <*n*>+1 are set to *value*.

Notes Each of the *lstore_<n>* instructions is the same as *lstore* with an *index* of <*n*>, except that the operand <*n*> is implicit.

ldc *ldc*

Operation Push item from run-time constant pool

Format

ldc
index

Forms *ldc* = 18 (0x12)

Operand Stack ... →
..., *value*

Description The *index* is an unsigned byte that must be a valid index into the run-time constant pool of the current class (§2.6). The run-time constant pool entry at *index* either must be a run-time constant of type `int` or `float`, or a `reference` to a string literal, or a symbolic reference to a class, method type, or method handle (§5.1).

If the run-time constant pool entry is a run-time constant of type `int` or `float`, the numeric *value* of that run-time constant is pushed onto the operand stack as an `int` or `float`, respectively.

Otherwise, if the run-time constant pool entry is a `reference` to an instance of class `String` representing a string literal (§5.1), then a `reference` to that instance, *value*, is pushed onto the operand stack.

Otherwise, if the run-time constant pool entry is a symbolic reference to a class (§5.1), then the named class is resolved (§5.4.3.1) and a `reference` to the `Class` object representing that class, *value*, is pushed onto the operand stack.

Otherwise, the run-time constant pool entry must be a symbolic reference to a method type or a method handle (§5.1). The method type or method handle is resolved (§5.4.3.5) and a `reference` to the resulting instance of `java.lang.invoke.MethodType` or `java.lang.invoke.MethodHandle`, *value*, is pushed onto the operand stack.

Linking Exceptions

During resolution of a symbolic reference to a class, any of the exceptions pertaining to class resolution (§5.4.3.1) can be thrown.

During resolution of a symbolic reference to a method type or method handle, any of the exception pertaining to method type or method handle resolution (§5.4.3.5) can be thrown.

Notes

The *ldc* instruction can only be used to push a value of type `float` taken from the float value set (§2.3.2) because a constant of type `float` in the constant pool (§4.4.4) must be taken from the float value set.

2.11.5 Object Creation and Manipulation

- Create a new class instance: *new*.
- Create a new array: *newarray*, *anewarray*, *multianewarray*.
- Access fields of classes (`static` fields, known as class variables) and fields of class instances (non-`static` fields, known as instance variables): *getstatic*, *putstatic*, *getfield*, *putfield*.
- Load an array component onto the operand stack: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Store a value from the operand stack as an array component: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- Get the length of array: *arraylength*.
- Check properties of class instances or arrays: *instanceof*, *checkcast*.

2.11.6 Operand Stack Management Instructions

A number of instructions are provided for the direct manipulation of the operand stack: *pop*, *pop2*, *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2*, *dup2_x2*, *swap*.

dup_x1: { before: ....., value2. value1
after: ....., value1, value2, value1.

dup_x2 { before: ....., value3, value2, value1
after: ....., value1, value3, value2, value1

new *new*

Operation Create new object

Format

new
indexbyte1
indexbyte2

Forms *new* = 187 (0xbb)

Operand Stack ... → ..., *objectref*

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in a class type. Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (§2.3, §2.4). The *objectref*, a `reference` to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

Operation Create new array

Format

newarray
atype

Forms *newarray* = 188 (0xbc)

Operand Stack ..., *count* →
..., *arrayref*

Description The *count* must be of type `int`. It is popped off the operand stack. The *count* represents the number of elements in the array to be created.

The *atype* is a code that indicates the type of array to create. It must take one of the following values:

not part of the opcode

Table 6.5.newarray-A. Array type codes

Array Type	*atype*
`T_BOOLEAN`	4
`T_CHAR`	5
`T_FLOAT`	6
`T_DOUBLE`	7
`T_BYTE`	8
`T_SHORT`	9
`T_INT`	10
`T_LONG`	11

A new array whose components are of type *atype* and of length *count* is allocated from the garbage-collected heap. A `reference` *arrayref* to this new array object is pushed into the operand stack. Each of the elements of the new array is initialized to the default initial value (§2.3, §2.4) for the element type of the array type.

Run-time Exception

If *count* is less than zero, *newarray* throws a `NegativeArraySizeException`.

Notes

In Oracle's Java Virtual Machine implementation, arrays of type `boolean` (*atype* is `T_BOOLEAN`) are stored as arrays of 8-bit values and are manipulated using the *baload* and *bastore* instructions (§*baload*, §*bastore*) which also access arrays of type `byte`. Other implementations may implement packed `boolean` arrays; the *baload* and *bastore* instructions must still be used to access those arrays.

byte

getfield *getfield*

Operation Fetch field from object

Format

getfield
indexbyte1
indexbyte2

Forms *getfield* = 180 (0xb4)

Operand Stack ..., *objectref* →
..., *value*

Description The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2). The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.

The type of *objectref* must not be an array type. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

fields are accessed by name

Operation Set static field in class

Format

putstatic
indexbyte1
indexbyte2

Forms *putstatic* = 179 (0xb3)

Operand Stack ..., *value* →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.2).

On successful resolution of the field, the class or interface that declared the resolved field is initialized (§5.5) if that class or interface has not already been initialized.

type not part of instruction, but part of value and descriptor

The type of a *value* stored by a *putstatic* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class, and the instruction must occur in the `<clinit>` method of the current class (§2.9).

2.11.5 Object Creation and Manipulation

- Create a new class instance: *new*.
- Create a new array: *newarray*, *anewarray*, *multianewarray*.
- Access fields of classes (`static` fields, known as class variables) and fields of class instances (non-`static` fields, known as instance variables): *getstatic*, *putstatic*, *getfield*, *putfield*.
- Load an array component onto the operand stack: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Store a value from the operand stack as an array component: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- Get the length of array: *arraylength*.
- Check properties of class instances or arrays: *instanceof*, *checkcast*.

2.11.6 Operand Stack Management Instructions

A number of instructions are provided for the direct manipulation of the operand stack: *pop*, *pop2*, *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2*, *dup2_x2*, *swap*.

baload

Operation Load `byte` or `boolean` from array

Format *baload*

Forms *baload* = 51 (0x33)

Operand Stack ..., *arrayref*, *index* → ..., *value*

baload

types in opcodes…sort of

The *baload* instruction is used to load values from both `byte` and `boolean` arrays. In Oracle's Java Virtual Machine implementation, `boolean` arrays - that is, arrays of type `T_BOOLEAN` (§2.2, §*newarray*) - are implemented as arrays of 8-bit values. Other implementations may implement packed `boolean` arrays; the *baload* instruction of such implementations must be used to access those arrays.

caload

Operation Load `char` from array

Format *caload*

Forms *caload* = 52 (0x34)

Operand Stack ..., *arrayref*, *index* → ..., *value*

- `char`, whose values are 16-bit unsigned integers representing Unicode code points in the Basic Multilingual Plane, encoded with UTF-16, and whose default value is the null code point (`'\u0000'`)

instanceof

instanceof

Operation Determine if object is of given type

Format

instanceof
indexbyte1
indexbyte2

Forms *instanceof* = 193 (0xc1)

Operand Stack ..., *objectref* →
..., *result*

Otherwise, the named class, array, or interface type is resolved (§5.4.3.1). If *objectref* is an instance of the resolved class or array or implements the resolved interface, the *instanceof* instruction pushes an `int` *result* of 1 as an `int` on the operand stack; otherwise, it pushes an `int` *result* of 0.

Notes The *instanceof* instruction is very similar to the *checkcast* instruction (§*checkcast*). It differs in its treatment of `null`, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

checkcast *checkcast*

Operation Check whether object is of given type

Format

checkcast
indexbyte1
indexbyte2

Forms *checkcast* = 192 (0xc0)

Operand Stack ..., *objectref* →
..., *objectref*

Description The *objectref* must be of type `reference`. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a class, array, or interface type.

If *objectref* is `null`, then the operand stack is unchanged.

Otherwise, the named class, array, or interface type is resolved (§5.4.3.1). If *objectref* can be cast to the resolved class, array, or interface type, the operand stack is unchanged; otherwise, the *checkcast* instruction throws a `ClassCastException`.

2.11.5 Object Creation and Manipulation

- Create a new class instance: *new*.
- Create a new array: *newarray*, *anewarray*, *multianewarray*.
- Access fields of classes (`static` fields, known as class variables) and fields of class instances (non-`static` fields, known as instance variables): *getstatic*, *putstatic*, *getfield*, *putfield*.
- Load an array component onto the operand stack: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Store a value from the operand stack as an array component: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- Get the length of array: *arraylength*.
- Check properties of class instances or arrays: *instanceof*, *checkcast*.

2.11.6 Operand Stack Management Instructions

A number of instructions are provided for the direct manipulation of the operand stack: *pop*, *pop2*, *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2*, *dup2_x2*, *swap*.

dup on category 2

6 flavors of dup!

Operation Pop the top operand stack value

Format

pop

Forms *pop* = 87 (0x57)

Operand Stack ..., *value* →
...

Description Pop the top value from the operand stack.

The *pop* instruction must not be used unless *value* is a value of a category 1 computational type (§2.11.1).

dup

Operation	Duplicate the top operand stack value
Format	*dup*
Forms	*dup* = 89 (0x59)
Operand Stack	..., *value* → ..., *value*, *value*
Description	Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

The *dup* instruction must not be used unless *value* is a value of a category 1 computational type (§2.11.1).

dup

type-agnostic … sort of

Both have the same meaning, but with different representation.

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

Actual type	Computational type	Category
`boolean`	`int`	1
`byte`	`int`	1
`char`	`int`	1
`short`	`int`	1
`int`	`int`	1
`float`	`float`	1
`reference`	`reference`	1
`returnAddress`	`returnAddress`	1
`long`	`long`	2
`double`	`double`	2

T → 型

integer to some other type →

Table 2.11.1-A. Type support in the Java Virtual Machine instruction set

opcode	byte	short	int	long	float	double	char	reference
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			*iushr*	*lushr*				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

dup2 *dup2*

Operation Duplicate the top one or two operand stack values

Format

dup2

Forms *dup2* = 92 (0x5c)

Operand Stack

Form 1:

..., *value2*, *value1* →

..., *value2*, *value1*, *value2*, *value1*

where both *value1* and *value2* are values of a category 1 computational type (§2.11.1).

Form 2:

..., *value* →

..., *value*, *value*

where *value* is a value of a category 2 computational type (§2.11.1).

determined by the top operand's type of the stack.

Description Duplicate the top one or two values on the operand stack and push the duplicated value or values back onto the operand stack in the original order.

2.11.4 Type Conversion Instructions

- `int` to `long`, `float`, or `double`
- `long` to `float` or `double`
- `float` to `double`

The widening numeric conversion instructions are *i2l*, *i2f*, *i2d*, *l2f*, *l2d*, and *f2d*.

- `int` to `byte`, `short`, or `char`
- `long` to `int`
- `float` to `int` or `long`
- `double` to `int`, `long`, or `float`

The narrowing numeric conversion instructions are *i2b*, *i2c*, *i2s*, *l2i*, *f2i*, *f2l*, *d2i*, *d2l*, and *d2f*. A narrowing numeric conversion can result in a value of different sign, a different order of magnitude, or both; it may thereby lose precision.

Operation Convert `int` to `long`

Format

i2l

Forms *i2l* = 133 (0x85)

Operand Stack ..., *value* →
..., *result*

Description The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack and sign-extended to a `long` *result*. That *result* is pushed onto the operand stack.

Notes The *i2l* instruction performs a widening primitive conversion (JLS §5.1.2). Because all values of type `int` are exactly representable by type `long`, the conversion is exact.

2.11.7 Control Transfer Instructions

- Conditional branch: *ifeq*, *ifne*, *iflt*, *ifle*, *ifgt*, *ifge*, *ifnull*, *ifnonnull*, *if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmple*, *if_icmpgt* *if_icmpge*, *if_acmpeq*, *if_acmpne*.
- Compound conditional branch: *tableswitch*, *lookupswitch*.
- Unconditional branch: *goto*, *goto_w*, *jsr*, *jsr_w*, *ret*.

only integer and reference compare-and-branch

if<cond> *if<cond>*

Operation Branch if `int` comparison with zero succeeds

Format

if<cond>
branchbyte1
branchbyte2

Forms

ifeq = 153 (0x99)

ifne = 154 (0x9a)

iflt = 155 (0x9b)

ifge = 156 (0x9c)

ifgt = 157 (0x9d)

ifle = 158 (0x9e)

Operand Stack

..., *value* →

...

Description The *value* must be of type `int`. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *ifeq* succeeds if and only if *value* = 0
- *ifne* succeeds if and only if *value* ≠ 0
- *iflt* succeeds if and only if *value* < 0
- *ifle* succeeds if and only if *value* ≤ 0
- *ifgt* succeeds if and only if *value* > 0
- *ifge* succeeds if and only if *value* ≥ 0

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if<cond>* instruction.

if_icmp<cond>

if_icmp<cond>

Operation Branch if `int` comparison succeeds

Format

if_icmp<cond>
branchbyte1
branchbyte2

Forms

if_icmpeq = 159 (0x9f)

if_icmpne = 160 (0xa0)

if_icmplt = 161 (0xa1)

if_icmpge = 162 (0xa2)

if_icmpgt = 163 (0xa3)

if_icmple = 164 (0xa4)

Operand Stack ..., *value1*, *value2* →

...

Description Both *value1* and *value2* must be of type `int`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparison are as follows:

- *if_icmpeq* succeeds if and only if *value1* = *value2*
- *if_icmpne* succeeds if and only if *value1* ≠ *value2*
- *if_icmplt* succeeds if and only if *value1* < *value2*
- *if_icmple* succeeds if and only if *value1* ≤ *value2*
- *if_icmpgt* succeeds if and only if *value1* > *value2*
- *if_icmpge* succeeds if and only if *value1* ≥ *value2*

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if_icmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_icmp<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if_icmp<cond>* instruction.

Operation Branch if `reference` comparison succeeds

Format

if_acmp<cond>
branchbyte1
branchbyte2

Forms *if_acmpeq* = 165 (0xa5)

if_acmpne = 166 (0xa6)

Operand Stack ..., *value1*, *value2* →

...

Description Both *value1* and *value2* must be of type `reference`. They are both popped from the operand stack and compared. The results of the comparison are as follows:

- *if_acmpeq* succeeds if and only if *value1* = *value2*
- *if_acmpne* succeeds if and only if *value1* ≠ *value2*

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if_acmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_acmp<cond>* instruction.

Otherwise, if the comparison fails, execution proceeds at the address of the instruction following this *if_acmp<cond>* instruction.

tableswitch

Operation Access jump table by index and jump

Format

tableswitch
<0-3 byte pad>
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
lowbyte1
lowbyte2
lowbyte3
lowbyte4
highbyte1
highbyte2
highbyte3
highbyte4
jump offsets...

Forms *tableswitch* = 170 (0xaa)

Operand Stack ..., *index* →
...

No actual component in hardware.

tableswitch

Work on continuous interval!

If work on discrete value,

should consider lookupswitch.

low - high

Description

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

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

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

Notes

The alignment required of the 4-byte operands of the *tableswitch* instruction guarantees 4-byte alignment of those operands if and only if the method that contains the *tableswitch* starts on a 4-byte boundary.

cf. lookupswitch

goto

goto

Operation Branch always

Format

goto
branchbyte1
branchbyte2

Forms *goto* = 167 (0xa7)

Operand Stack No change

Description The unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

2.11.8 Method Invocation and Return Instructions

- *invokevirtual* invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the normal method dispatch in the Java programming language.
- *invokeinterface* invokes an interface method, searching the methods implemented by the particular run-time object to find the appropriate method.
- *invokespecial* invokes an instance method requiring special handling, whether an instance initialization method (§2.9), a `private` method, or a superclass method.
- *invokestatic* invokes a class (`static`) method in a named class.
- *invokedynamic* invokes the method which is the target of the call site object bound to the *invokedynamic* instruction. The call site object was bound to a specific lexical occurrence of the *invokedynamic* instruction by the Java Virtual Machine as a result of running a bootstrap method before the first execution of the instruction. Therefore, each occurrence of an *invokedynamic* instruction has a unique linkage state, unlike the other instructions which invoke methods.

more later

The method return instructions, which are distinguished by return type, are *ireturn* (used to return values of type `boolean`, `byte`, `char`, `short`, or `int`), *lreturn*, *freturn*, *dreturn*, and *areturn*. In addition, the *return* instruction is used to return from methods declared to be void, instance initialization methods, and class or interface initialization methods.

```
class Foo { void m() { … } }
new Foo().m(); // invokevirtual
```

invokevirtual ***invokevirtual***

Operation Invoke instance method; dispatch based on class

Format

invokevirtual
indexbyte1
indexbyte2

Forms *invokevirtual* = 182 (0xb6)

Operand Stack ..., *objectref*, [*arg1*, [*arg2* ...]] → ...

5 more pages of detail elided…

Virtual Method Table Dispatch

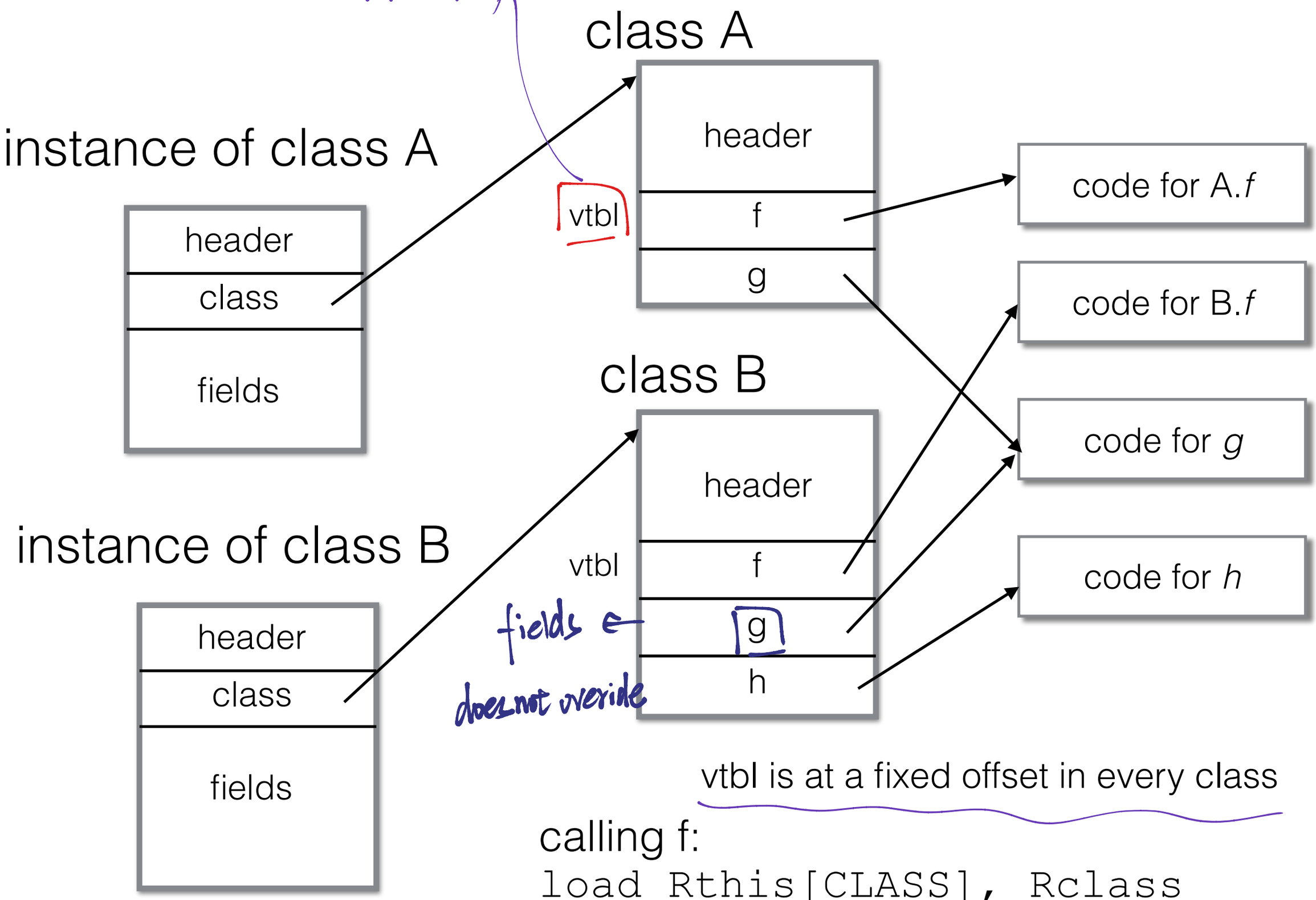
- The standard technique for invoking virtual methods in interpreters and compilers

- Single-inheritance, static typing

```
class A {
 void f(..) {…}
 int g(…) {…f(…);} }

class B extends A {
 void f(..) {…} // overridden
 String h(…) {…} } // added
```

entry point for each method.



vtbl is at a fixed offset in every class

calling f:

```
load Rthis[CLASS], Rclass
load Rclass[VTBL+f], Rmethod
call Rmethod
```

faster than iterative lookup

table

class Foo { static void m() { … } }
Foo.m(); // invokestatic

invokestatic

invokestatic

Operation Invoke a class (`static`) method

Format

invokestatic
indexbyte1
indexbyte2

Forms *invokestatic* = 184 (0xb8)

Operand Stack ..., [*arg1*, [*arg2* ...]] → ...

no *this* passed

Target method is determined statically — no dynamic lookup required

```
interface F { void m(); }
class Foo implements F { void m() { … } }
aFoo.m(); // invokeinterface
```

invokeinterface ***invokeinterface***

Operation Invoke interface method

Format

invokeinterface
indexbyte1
indexbyte2
count
0

redundant. historical remaining.

Forms *invokeinterface* = 185 (0xb9)

Operand Stack ..., *objectref*, [*arg1*, [*arg2* ...]] →
...

Interface dispatch

- to be covered later, in compilation

```
class A { void m() {…} }
class B extends A {
  void m() { …super.m(); // invokespecial
… } }
```

invokespecial ***invokespecial***

Operation Invoke instance method; special handling for superclass, private, and instance initialization method invocations

Format

invokespecial
indexbyte1
indexbyte2

Forms *invokespecial* = 183 (0xb7)

Operand Stack ..., *objectref*, [*arg1*, [*arg2* ...]] →
...

Notes The difference between the *invokespecial* instruction and the *invokevirtual* instruction (§*invokevirtual*) is that *invokevirtual* invokes a method based on the class of the object. The *invokespecial* instruction is used to invoke instance initialization methods (§2.9) as well as `private` methods and methods of a superclass of the current class.

private / protected method.

return *return*

Operation Return `void` from method

Format

return

Forms *return* = 177 (0xb1)

Operand ... →

Stack [empty]

Description The current method must have return type `void`. If the current method is a `synchronized` method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread. If no exception is thrown, any values on the operand stack of the current frame (§2.6) are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

2.11.10 Synchronization

Method-level synchronization is performed implicitly, as part of method invocation and return (§2.11.8). A `synchronized` method is distinguished in the run-time constant pool's `method_info` structure (§4.6) by the `ACC_SYNCHRONIZED` flag, which is checked by the method invocation instructions. When invoking a method

Synchronization of sequences of instructions is typically used to encode the `synchronized` block of the Java programming language. The Java Virtual Machine supplies the *monitorenter* and *monitorexit* instructions to support such language constructs. Proper implementation of `synchronized` blocks requires cooperation

Operation Enter monitor for object

Format

monitorenter

Forms *monitorenter* = 194 (0xc2)

Operand Stack ..., *objectref* →

...

Description The *objectref* must be of type `reference`.

Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes *monitorenter* attempts to gain ownership of the monitor associated with *objectref*, as follows:

- If the entry count of the monitor associated with *objectref* is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.
- If the thread already owns the monitor associated with *objectref*, it reenters the monitor, incrementing its entry count.
- If another thread already owns the monitor associated with *objectref*, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

Notes

A *monitorenter* instruction may be used with one or more *monitorexit* instructions (§*monitorexit*) to implement a `synchronized` statement in the Java programming language (§3.14). The *monitorenter* and *monitorexit* instructions are not used in the implementation of `synchronized` methods, although they can be used to provide equivalent locking semantics. Monitor entry on invocation of a `synchronized` method, and monitor exit on its return, are handled implicitly by the Java Virtual Machine's method invocation and return instructions, as if *monitorenter* and *monitorexit* were used.

The association of a monitor with an object may be managed in various ways that are beyond the scope of this specification. For instance, the monitor may be allocated and deallocated at the same time as the object. Alternatively, it may be dynamically allocated at the time when a thread attempts to gain exclusive access to the object and freed at some later time when no thread remains in the monitor for the object.

The synchronization constructs of the Java programming language require support for operations on monitors besides entry and exit. These include waiting on a monitor (`Object.wait`) and notifying other threads waiting on a monitor (`Object.notifyAll` and `Object.notify`). These operations are supported in the standard package `java.lang` supplied with the Java Virtual Machine. No explicit support for these operations appears in the instruction set of the Java Virtual Machine.

Operation Exit monitor for object

Format

monitorexit

Forms *monitorexit* = 195 (0xc3)

Operand Stack ..., *objectref* →

...

Description The *objectref* must be of type `reference`.

The thread that executes *monitorexit* must be the owner of the monitor associated with the instance referenced by *objectref*.

The thread decrements the entry count of the monitor associated with *objectref*. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

Run-time Exceptions If *objectref* is `null`, *monitorexit* throws a `NullPointerException`.

Otherwise, if the thread that executes *monitorexit* is not the owner of the monitor associated with the instance referenced by *objectref*, *monitorexit* throws an `IllegalMonitorStateException`.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the second of those rules is violated by the execution of this *monitorexit* instruction, then *monitorexit* throws an `IllegalMonitorStateException`.

Exceptions

2.11.9 Throwing Exceptions

An exception is thrown programmatically using the *athrow* instruction. Exceptions can also be thrown by various Java Virtual Machine instructions if they detect an abnormal condition.

- recall the forms:

```
void catchOne() {
    try {
        tryItOut();  // will call cantBeZero()
    } catch (TestExc e) {
        handleExc(e);
    }
}

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

The compiled form of exceptions

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

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

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

```
Method void catchOne()
0   aload_0            // Beginning of try block
1   invokevirtual #6   // Method Example.tryItOut()V
4   return             // End of try block; normal return
5   astore_1           // Store thrown value in local var 1
6   aload_0            // Push this
7   aload_1            // Push thrown value
8   invokevirtual #5   // Invoke handler method:
                       // Example.handleExc(LTestExc;)V
11  return             // Return after handling TestExc
Exception table:
From    To      Target      Type
0       4       5           Class TestExc
```

The compiled form of exceptions

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

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

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

```
Method void catchOne()
0   aload_0              // Beginning of try block
1   invokevirtual #6     // Method Example.tryItOut()V
4   return               // End of try block; normal return
5   astore_1             // Store thrown value in local var 1
6   aload_0              // Push this
7   aload_1              // Push thrown value
8   invokevirtual #5     // Invoke handler method:
                         // Example.handleExc(LTestExc;)V
11  return               // Return after handling TestExc
Exception table:
From    To      Target      Type
0       4       5           Class TestExc
```

The compiled form of exceptions

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

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

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

```
Method void catchOne()
0   aload_0            // Beginning of try block
1   invokevirtual #6   // Method Example.tryItOut()V
4   return             // End of try block; normal return
5   astore_1           // Store thrown value in local var 1
6   aload_0            // Push this
7   aload_1            // Push thrown value
8   invokevirtual #5   // Invoke handler method:
                       // Example.handleExc(LTestExc;)V
11  return             // Return after handling TestExc
Exception table:
From    To    Target    Type
0       4     5         Class TestExc
```

The compiled form of exceptions

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

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

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

```
Method void catchOne()
0   aload_0            // Beginning of try block
1   invokevirtual #6   // Method Example.tryItOut()V
4   return             // End of try block; normal return
5   astore_1           // Store thrown value in local var 1
6   aload_0            // Push this
7   aload_1            // Push thrown value
8   invokevirtual #5   // Invoke handler method:
                       // Example.handleExc(LTestExc;)V
11  return             // Return after handling TestExc
Exception table:
From    To    Target    Type
0       4     5         Class TestExc
```

The compiled form of exceptions

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

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

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

```
Method void catchOne()
0   aload_0            // Beginning of try block
1   invokevirtual #6   // Method Example.tryItOut()V
4   return             // End of try block; normal return
5   astore_1           // Store thrown value in local var 1
6   aload_0            // Push this
7   aload_1            // Push thrown value
8   invokevirtual #5   // Invoke handler method:
                       // Example.handleExc(LTestExc;)V
11  return             // Return after handling TestExc
Exception table:
From    To    Target    Type
0       4     5         Class TestExc
```

The compiled form of exceptions

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

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

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

```
Method void catchOne()
0   aload_0            // Beginning of try block
1   invokevirtual #6   // Method Example.tryItOut()V
4   return             // End of try block; normal return
5   astore_1           // Store thrown value in local var 1
6   aload_0            // Push this
7   aload_1            // Push thrown value
8   invokevirtual #5   // Invoke handler method:
                       // Example.handleExc(LTestExc;)V
11  return             // Return after handling TestExc
Exception table:
From    To    Target    Type
0       4     5         Class TestExc
```

Operation Throw exception or error

Format

athrow

Forms *athrow* = 191 (0xbf)

Operand Stack ..., *objectref* →
objectref

Description The *objectref* must be of type `reference` and must refer to an object that is an instance of class `Throwable` or of a subclass of `Throwable`. It is popped from the operand stack. The *objectref* is then thrown by searching the current method (§2.6) for the first exception handler that matches the class of *objectref*, as given by the algorithm in §2.10.

If an exception handler that matches *objectref* is found, it contains the location of the code intended to handle this exception. The `pc` register is reset to that location, the operand stack of the current frame is cleared, *objectref* is pushed back onto the operand stack, and execution continues.

If no matching exception handler is found in the current frame, that frame is popped. If the current frame represents an invocation of a `synchronized` method, the monitor entered or reentered on invocation of the method is exited as if by execution of a *monitorexit* instruction (§*monitorexit*). Finally, the frame of its invoker is reinstated, if such a frame exists, and the *objectref* is rethrown. If no such frame exists, the current thread exits.

The operand stack diagram for the *athrow* instruction may be misleading: If a handler for this exception is matched in the current method, the *athrow* instruction discards all the values on the operand stack, then pushes the thrown object onto the operand stack. However, if no handler is matched in the current method and the exception is thrown farther up the method invocation chain, then the operand stack of the method (if any) that handles the exception is cleared and *objectref* is pushed onto that empty operand stack. All intervening frames from the method that threw the exception up to, but not including, the method that handles the exception are discarded.

Last, but not least, *finally*

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

- The *finally* clause is invoked regardless of whether the *try* cause completed successfully, or whether it was aborted due to an exception.

JSR and RET, a cautionary tale in 2 acts
Act One: the conquest of space

2.11.7 Control Transfer Instructions

- Unconditional branch: *goto*, *goto_w*, *jsr*, *jsr_w*, *ret*.

- Used to call the *finally* block in a try-catch-finally

  1. From non-excepting code (after *try*), and

  2. After an exception is thrown and caught (after *try* and *catch*)

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

```
Method void tryFinally()
0   aload_0              // Beginning of try block
1   invokevirtual #6     // Method Example.tryItOut()V
4   jsr 14               // Call finally block
7   return               // End of try block
8   astore_1             // Beginning of handler for any throw
9   jsr 14               // Call finally block
12  aload_1              // Push thrown value
13  athrow               // ...and rethrow value to the invoker
14  astore_2             // Beginning of finally block
15  aload_0              // Push this
16  invokevirtual #5     // Method Example.wrapItUp()V
19  ret 2                // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

```
Method void tryFinally()
0   aload_0              // Beginning of try block
1   invokevirtual #6     // Method Example.tryItOut()V
4   jsr 14               // Call finally block
7   return               // End of try block
8   astore_1             // Beginning of handler for any throw
9   jsr 14               // Call finally block
12  aload_1              // Push thrown value
13  athrow               // ...and rethrow value to the invoker
14  astore_2             // Beginning of finally block
15  aload_0              // Push this
16  invokevirtual #5     // Method Example.wrapItUp()V
19  ret 2                // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

Just linkage

```
Method void tryFinally()
0   aload_0              // Beginning of try block
1   invokevirtual #6     // Method Example.tryItOut()V
4   jsr 14               // Call finally block
7   return               // End of try block
8   astore_1             // Beginning of handler for any throw
9   jsr 14               // Call finally block
12  aload_1              // Push thrown value
13  athrow               // ...and rethrow value to the invoker
14  astore_2             // Beginning of finally block
15  aload_0              // Push this
16  invokevirtual #5     // Method Example.wrapItUp()V
19  ret 2                // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

```
Method void tryFinally()
0   aload_0              // Beginning of try block
1   invokevirtual #6     // Method Example.tryItOut()V
4   jsr 14               // Call finally block
7   return               // End of try block
8   astore_1             // Beginning of handler for any throw
9   jsr 14               // Call finally block
12  aload_1              // Push thrown value
13  athrow               // ...and rethrow value to the invoker
14  astore_2             // Beginning of finally block
15  aload_0              // Push this
16  invokevirtual #5     // Method Example.wrapItUp()V
19  ret 2                // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

```
Method void tryFinally()
0   aload_0              // Beginning of try block
1   invokevirtual #6     // Method Example.tryItOut()V
4   jsr 14               // Call finally block
7   return               // End of try block
8   astore_1             // Beginning of handler for any throw
9   jsr 14               // Call finally block
12  aload_1              // Push thrown value
13  athrow               // ...and rethrow value to the invoker
14  astore_2             // Beginning of finally block
15  aload_0              // Push this
16  invokevirtual #5     // Method Example.wrapItUp()V
19  ret 2                // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

```
Method void tryFinally()
0   aload_0             // Beginning of try block
1   invokevirtual #6    // Method Example.tryItOut()V
4   jsr 14              // Call finally block
7   return              // End of try block
8   astore_1            // Beginning of handler for any throw
9   jsr 14              // Call finally block
12  aload_1             // Push thrown value
13  athrow              // ...and rethrow value to the invoker
14  astore_2            // Beginning of finally block
15  aload_0             // Push this
16  invokevirtual #5    // Method Example.wrapItUp()V
19  ret 2               // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

```
Method void tryFinally()
0   aload_0             // Beginning of try block
1   invokevirtual #6    // Method Example.tryItOut()V
4   jsr 14              // Call finally block
7   return              // End of try block
8   astore_1            // Beginning of handler for any throw
9   jsr 14              // Call finally block
12  aload_1             // Push thrown value
13  athrow              // ...and rethrow value to the invoker
14  astore_2            // Beginning of finally block
15  aload_0             // Push this
16  invokevirtual #5    // Method Example.wrapItUp()V
19  ret 2               // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

Finally, compiled using JSR and RET

```
void tryFinally() {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

```
Method void tryFinally()
0   aload_0             // Beginning of try block
1   invokevirtual #6    // Method Example.tryItOut()V
4   jsr 14              // Call finally block
7   return              // End of try block
8   astore_1            // Beginning of handler for any throw
9   jsr 14              // Call finally block
12  aload_1             // Push thrown value
13  athrow              // ...and rethrow value to the invoker
14  astore_2            // Beginning of finally block
15  aload_0             // Push this
16  invokevirtual #5    // Method Example.wrapItUp()V
19  ret 2               // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

Why do this? To save duplicating the *finally* code.

jsr *jsr*

Operation Jump subroutine

subroutines *within* methods!

Format

jsr
branchbyte1
branchbyte2

Forms *jsr* = 168 (0xa8)

Operand Stack ... →
..., *address*

Description The *address* of the opcode of the instruction immediately following this *jsr* instruction is pushed onto the operand stack as a value of type `returnAddress`. The unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of this *jsr* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr* instruction.

Notes Note that *jsr* pushes the address onto the operand stack and *ret* (§*ret*) gets it out of a local variable. This asymmetry is intentional.

In Oracle's implementation of a compiler for the Java programming language prior to Java SE 6, the *jsr* instruction was used with the *ret* instruction in the implementation of the `finally` clause (§3.13, §4.10.2.5).

Operation Return from subroutine

Format

ret
index

Forms *ret* = 169 (0xa9)

Operand Stack No change

Description The *index* is an unsigned byte between 0 and 255, inclusive. The local variable at *index* in the current frame (§2.6) must contain a value of type `returnAddress`. The contents of the local variable are written into the Java Virtual Machine's `pc` register, and execution continues there.

Notes

Note that *jsr* (§*jsr*) pushes the address onto the operand stack and *ret* gets it out of a local variable. This asymmetry is intentional.

In Oracle's implementation of a compiler for the Java programming language prior to Java SE 6, the *ret* instruction was used with the *jsr* and *jsr_w* instructions (§*jsr*, §*jsr_w*) in the implementation of the `finally` clause (§3.13, §4.10.2.5).

The *ret* instruction should not be confused with the *return* instruction (§*return*). A *return* instruction returns control from a method to its invoker, without passing any value back to the invoker.

The *ret* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

- *jsr* and *ret* were abandoned in Java SE 6 (class version 50) and now the *finally* code is duplicated.

- We shall see how *jar* and *ret* complicate GC when we cover memory management.

The interaction of exceptions and synchronization

The Java Virtual Machine supports exceptions thrown within `synchronized` methods and `synchronized` statements differently:

- Monitor exit on normal `synchronized` method completion is handled by the Java Virtual Machine's return instructions. Monitor exit on abrupt `synchronized` method completion is handled implicitly by the Java Virtual Machine's *athrow* instruction.
- When an exception is thrown from within a `synchronized` statement, exit from the monitor entered prior to the execution of the `synchronized` statement is achieved using the Java Virtual Machine's exception handling mechanism (§3.14).

Examining bytecode

- Use *javap* from the JDK to “disassemble” methods from a class into their constituent byte codes:

```
javap -c Foo
```

- A variety of open source bytecode assemblers (incl. Java libraries) are available, should you want to create your own bytecode “by hand”.

JVM: Class file verification

- When a class is loaded, the file is first checked to be in the right format; this includes checking for valid names in descriptors, and a valid constant pool.
- Additional checks ensure that more static constraints are met (e.g., branches go to a valid bytecode)
- Finally, all operations are type checked using either type inference or embedded stack type maps.

Types in the JVM

A single local variable can hold a value of type `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference`, or `returnAddress`. A pair of local variables can hold a value of type `long` or `double`.

2.3.3 The `returnAddress` Type and Values

The `returnAddress` type is used by the Java Virtual Machine's *jsr*, *ret*, and *jsr_w* instructions (§*jsr*, §*ret*, §*jsr_w*). The values of the `returnAddress` type are pointers to the opcodes of Java Virtual Machine instructions. Unlike the numeric primitive types, the `returnAddress` type does not correspond to any Java programming language type and cannot be modified by the running program.

Detour: abstract interpretation

Abstract interpretation is the process of interpreting a program using an abstracted (i.e., detail-omitting) representation of program state.

Example uses:

- Calculating the types of items on the stack (e.g., to verify type correctness of bytecode)

Example: type-checking the expression language bytecode

We'll consider how to type-check the bytecode introduced in the last section. Let's additionally assume each variable can contain an int or a float, typed statically, FORTRAN-style (*i*—*n* are ints).

On the next slide we sketch an abstract bytecode interpreter, which considers only types (int and float).

```
case o(LITINT): push(INT); inc_pc(5); continue;
case o(LITFLT): push(FLT); inc_pc(5); continue;
case o(GETINT): if ('i'>L(*pc)+'a' || 'n'<L(*pc)+'a') error();
                push(INT); ++pc; continue;
case o(PUTINT): if (stack_empty()) error();
                if (tos() != INT) error();
                if (var not an int var) error();
                pop(); ++pc; continue;
case o(GETFLT): // similar to GETINT
case o(PUTFLT): // similar to PUTINT
case o(ADDINT): if (stack_depth()<2) error();
                if (tos()!=INT || nos()!=INT) error();
                pop(); ++pc; continue;
// ditto other INT binary operators
case o(ADDFLT): // similar to ADDINT
case o(I2F):    if (stack_empty()) error();
                if (tos()!=INT) error();
                tos()= FLT; ++pc; continue;
case o(END):    if (!stack_empty()) error(); // or whatever
```

Abstract interpretation of types in the presence of loops and branches

- Type state at the merge of two branches must be identical
  - Type state at the end of a loop must match type state at the beginning of the loop
- The abstract interpreter is run over each basic block, starting with the function entry

Type checking in the JVM verifier

- The original scheme in the JVM (§4.10.2) used abstract interpretation to infer the type state of the stack and locals.
  - This was complex and potentially slow and bulky (undesirable on space-constrained devices)
- Since class file version 50, a different technique has been used: stack maps, embedded as attributes in the class file (`StackMapTable`).

Pre-computed stack type maps

- *javac* (or whatever generated the bytecode) can model the stack type state, and emits a stack type map for each instruction.
- The JVM can then easily verify in a single pass that the map accurately corresponds to the behavior of the instructions.
- The precise rules are specified (in Prolog) in §4.10.1.

Miscellaneous topics

- Class loading and linking, class loaders, unloading, resolution, consistency
  - Chapter 5 of the spec., Chapter 13 of the language spec.
- “jar” files — just a way of packaging lots of class files
- Initializers and constructors
- Finalization — covered later, in GC
- Threads, atomicity, synchronization implementation
  - deserving a section of its own, later
- “quick” bytecodes — anachronism, of historical interest only

Wrap up

- Wide variety of typing techniques employed
- Verification used to guarantee properties of code, to ensure safety of JVM
- Stack representation: values or 32-bit words?
- Java semantics pervade the bytecodes

Exercises

- Read Chapter 3 of the JVM spec, *Compiling for the Java Virtual Machine*, and [Gosling 95].

- Sketch the design of an alternate bytecode set for Java attempting to be more orthogonal and polymorphic, e.g., have single *add*, *sub*, …, *load*, *store*, opcodes. Try to make the set minimal, without regard to performance. What are the necessary conditions for this to be viable? How many opcodes can you save? What would you do with the savings?
  Prepare a ~10-minute presentation and send it to Patrick by the end of next Thursday. We will select one or more to be presented next week. (PDF, PPT, or Keynote) It's OK to work in pairs or groups of 3 (be sure to put all names on the presentation).