

Applied Static Analysis 2016

Dr. Michael Eichberg (Organizer)

Johannes Lerch, Ben Hermann, Sebastian Proksch, Karim Ali Ph.D.



Java Bytecode

A HARDWARE- AND OPERATING SYSTEM-
INDEPENDENT BINARY FORMAT, KNOWN AS
THE CLASS FILE FORMAT.

The Java® Virtual Machine Specification

Java SE 8 Edition

Specification: JSR-000337 Java® SE 8 Release Contents Specification ("Specification") Version: 8

Status: Proposed Final Draft

Release: January 2014

Copyright © 1997, 2014, Oracle America, Inc. and/or its affiliates. All rights reserved. 500 Oracle Parkway, Redwood City, California 94065, U.S.A.

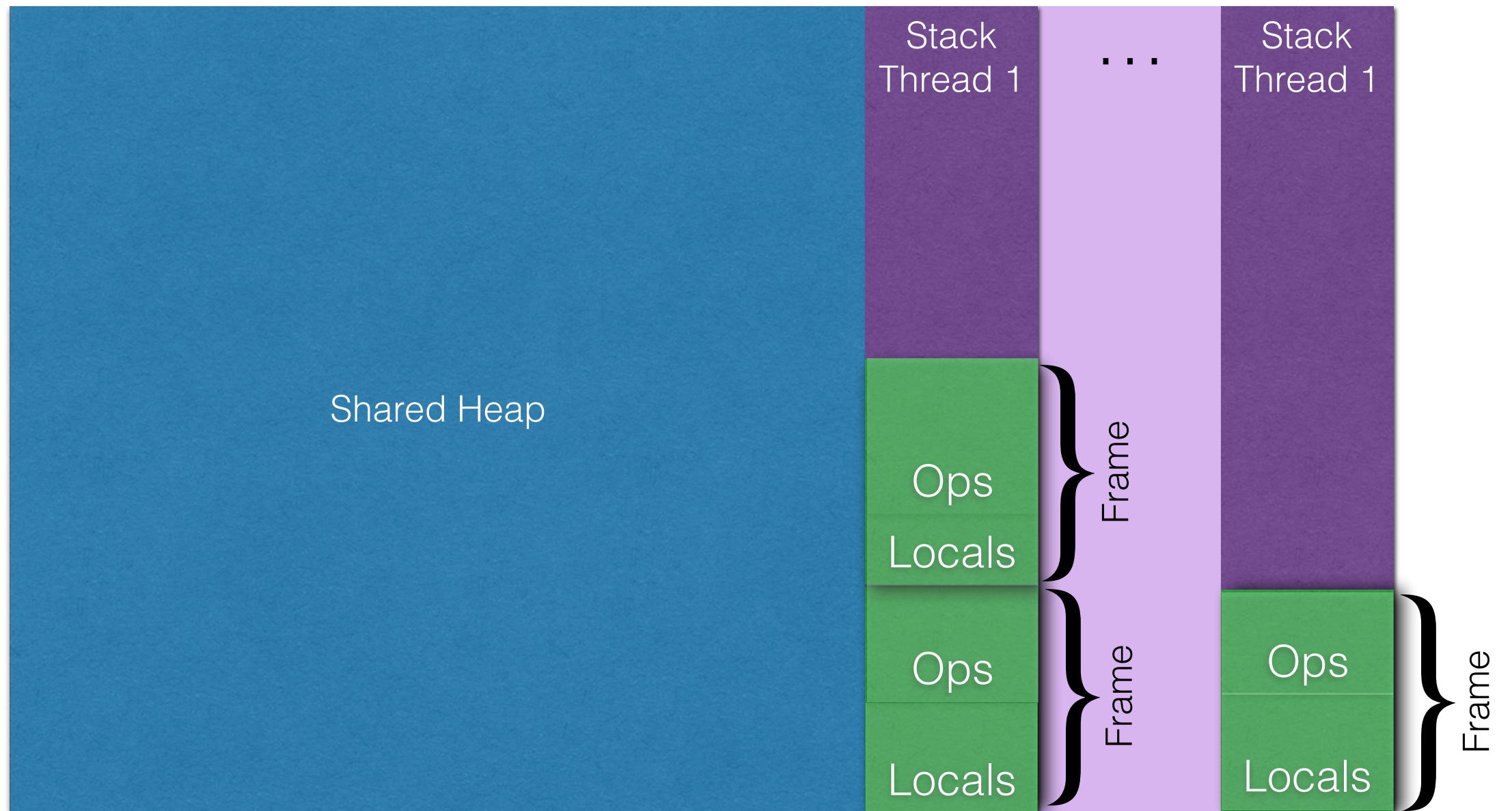
Structure of the Java Virtual Machine (JVM)

- Data types:
 - Primitive Types:
 - `boolean`, `byte`, `short`, `int`, `char` *(computational type int; cat. **1**)*
 - `long`, *(computational type long; cat. **2**)*
 - `float`, *(computational type float; cat. **1**)*
 - `double`, *(computational type long; cat. **2**)*
 - `return address` *(computational type return address; cat. **1**)*
 - Reference Types: *(computational type reference value; cat. **1**)*
 - `class`,
 - `array`,
 - `interface types`

Structure of the Java Virtual Machine (JVM)

- Run-time Data Areas
 - the **pc** (program counter) register contains the address of the instruction that is currently executed by a thread; each thread has its own pc
 - each JVM thread has a **private stack** which holds local variables and partial results
 - the **heap** which is shared among all threads
 - **frames** are allocated from a JVM thread's private stack when a method is invoked; each frame has its own array of **local variables** and **operand stack**
 - local variables are indexed
 - *a single local variable* can hold a value of type **boolean**, **byte**, **char**, **short**, **int**, **float**, **reference**, or **return address** (computational type category 1)
 - *a pair of local variables* can hold a value of type **long** or **double** (computational type category 2)
 - the **operand stack** is empty at creation time; an entry can hold any value
 - the **local variables** contains the parameters (including the implicit **this** parameter in local variable 0)

Structure of the Java Virtual Machine (JVM)



Structure of the Java Virtual Machine (JVM)

- Special Methods
 - the name of instance initialization methods (Java constructors) is “<init>”
 - the name of the class or interface initialization method (Java static initializer) is “<clinit>”
- Exceptions are instance of the class **Throwable** or one of its subclasses; exceptions are thrown if:
 - an **athrow** instruction was executed
 - an abnormal execution condition occurred (e.g., division by zero)

Structure of the Java Virtual Machine (JVM)

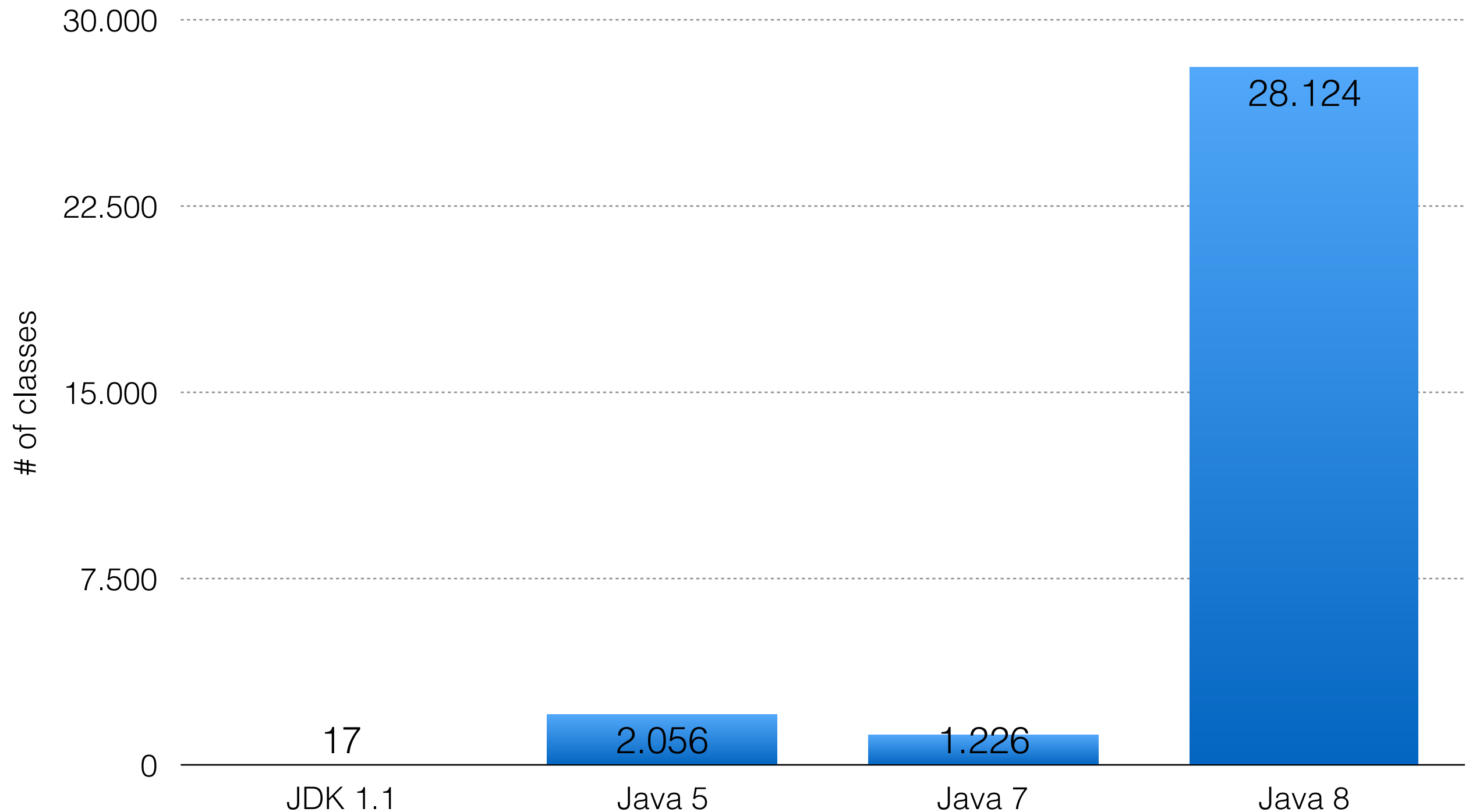
- Instruction Set Summary
 - an instruction consists of *a one-byte opcode* specifying the operation and zero or more operands (arguments to the operation)
 - most instructions *encode type information in their name*; in particular those operating on primitive types (e.g., **iadd**, **fadd**, **dadd**)
 - some are generic and are only restricted by the computational type category of the values (e.g. **swap**, **dup2**)

Structure of the Java Virtual Machine (JVM)

- Categories of Instructions
 - Load and store instructions (e.g., `aload_0`, `istore(x)`)
(Except of the load and store instructions, the only other instruction that manipulates a local variable is `iinc`.)
 - Arithmetic instructions (e.g., `iadd`, `iushr`)
 - (Primitive/Base) Type conversion instructions (e.g., `i2d`, `l2d`, `l2i`)
 - Object/Array creation and manipulation (e.g., `new`, `checkcast`)
 - (Generic) Operand Stack Management Instructions (e.g., `dup`)
 - Control Transfer Instructions (e.g., `itlt`, `if_icmplt`, `goto`, `jsr`, `ret`)
(Some are further modified using the wide modifier.)
obsolete since Java 5
 - Method Invocation and Return instructions (e.g., `invokespecial`, `return`)
 - Throwing Exceptions (`athrow`)
 - Synchronization (`monitorenter`, `monitorexit`)

Obsolete Code in the JDK

Class File Version of the Classes in the JDK 1.8.0_25



Structure of the Java Virtual Machine (JVM)

- The maximum length of a method is 65536.
(This is a frequent issue with generated code.)
- A method can have only 65536 local variables.
(The maximum number of locals in the JDK is 142.)
(The maximum number of locals in OPAL is/was 1136.)
- The maximum stack size of a single method is 65536.
(The maximum stack size of any method in the JDK is 42.)

Java Bytecode

```
static int max(int i, int j) {  
    if (i > j)  
        return i;  
    else  
        return j;  
}
```

PC	Instruction
0	iload_0
1	iload_1
2	if_icmple →7
5	iload_0
6	ireturn
7	iload_1
8	ireturn

Java Bytecode

```
static int max(int i, int j) {  
    if (i > j)  
        return i;  
    else  
        return j;  
}
```

PC	Instruction	Operand Stack	Registers/Local Vars.
0	iload_0		0: an int 1: an int
1	iload_1	an int	0: an int 1: an int
2	if_icmple →7	an int an int	0: an int 1: an int
5	iload_0		0: int ∈ [-2147483647,MAX] 1: int ∈ [MIN,2147483646]
6	ireturn	int ∈ [-2147483647,MAX]	0: int ∈ [-2147483647,MAX] 1: int ∈ [MIN,2147483646]
7	iload_1		0: an int 1: an int
8	ireturn	an int	0: an int 1: an int

Java Bytecode - Loops

```
static int factorial(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

PC	Instruction
0	iconst_1
1	istore_1
2	goto →12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc (lv=0,val=-1)
12	iload_0
13	ifgt →5
16	iload_1
17	ireturn

Java Bytecode - Object Creation

```
static int numberOfDigits(int i) {  
    assert (i > 0);  
    return ((int) Math.floor(Math.log10(i))) + 1;  
}
```

PC	Instruction	Code
0	getstatic TACDemo { boolean \$assertionsDisabled }	assert(i > 0)
3	ifne →18	
6	iload_0	
7	ifgt →18	
10	new java.lang.AssertionError	
13	dup	
14	invokespecial java.lang.AssertionError { void <init> () }	
17	athrow	Math.log10(i)
18	iload_0	
19	i2d	
20	invokestatic java.lang.Math { double log10 (double) }	
23	invokestatic java.lang.Math { double floor (double) }	Math.floor(...)
26	d2i	(int) <i>"Typecast"</i>
27	iconst_1	1
28	iadd	+
29	ireturn	return

Java Bytecode - Basic Exception Handling

```
int tryCatch(Object o) {  
    try {  
        List<?> l = (List<?>) o;  
        if (l.size() == 0)  
            return -1 / l.size();  
        else  
            return 1;  
    } catch (ClassCastException cce) {  
        return 0;  
    } catch (Error | RuntimeException e) {  
        throw e;  
    }  
}
```

▼ Method Body (Size: 31 bytes, Max Stack: 2, Max Locals: 3)

PC	Line	Instruction	Exceptions
0	109	<i>aload_1</i>	1: java.lang.ClassCastException 2: java.lang.Error 3: java.lang.RuntimeException
1		<i>checkcast</i> java.util.List	
4		<i>astore_2</i>	
5	110	<i>aload_2</i>	1: java.lang.ClassCastException 2: java.lang.Error 3: java.lang.RuntimeException
6		<i>invokeinterface</i> java.util.List { int size () }	
11		<i>ifne</i> <u>23</u>	
14	111	<i>iconst_m1</i>	1: java.lang.ClassCastException 2: java.lang.Error 3: java.lang.RuntimeException
15		<i>aload_2</i>	
16		<i>invokeinterface</i> java.util.List { int size () }	
21		<i>idiv</i>	1: java.lang.ClassCastException 2: java.lang.Error 3: java.lang.RuntimeException
22		<i>ireturn</i>	
23	113	<i>iconst_1</i>	
24		<i>ireturn</i>	1: java.lang.ClassCastException 2: java.lang.Error 3: java.lang.RuntimeException
25	114	<i>astore_2</i>	
26	115	<i>iconst_0</i>	
27		<i>ireturn</i>	1: java.lang.ClassCastException 2: java.lang.Error 3: java.lang.RuntimeException
28	116	<i>astore_2</i>	
29	117	<i>aload_2</i>	
30		<i>athrow</i>	

▼ Exception Table:

1. try [0-22) catch 25 java.lang.ClassCastException
2. try [0-22) catch 28 java.lang.Error
3. try [0-22) catch 28 java.lang.RuntimeException

inclusive

exclusive

15

Java Bytecode - Finally

```
int tryFinally(Object o) {  
    int result = 0;  
    try {  
        if (o instanceof List<?>) {  
            result = ((List<?>) o).size();  
        } else {  
            result = -1;  
        }  
        return result;  
    } finally {  
        result += 1;  
    }  
}
```

▼ Method Body (Size: 39 bytes, Max Stack: 1, Max Locals: 5)

PC	Line	Instruction	Exceptions
0	122	<i>iconst_0</i>	
1		<i>istore_2</i>	
2	124	<i>aload_1</i>	1: Any
3		<i>instanceof</i>	
6		<i>ifeq</i>	
9	125	<i>aload_1</i>	
10		<i>checkcast</i>	
13		<i>invokeinterface</i>	java.util.List { int size () }
18		<i>istore_2</i>	
19	126	<i>goto</i>	24
22	127	<i>iconst_m1</i>	
23		<i>istore_2</i>	
24	129	<i>iload_2</i>	
25		<i>istore</i>	4
27	131	<i>iinc</i>	2 1
30	129	<i>iload</i>	4
32		<i>ireturn</i>	
33	130	<i>astore_3</i>	
34	131	<i>iinc</i>	2 1
37	132	<i>aload_3</i>	
38		<i>athrow</i>	

▼ Exception Table:

1. try [2-27) catch 33 Any


```
static int readFirstByte(String path) throws Exception {  
    try (FileReader r = new FileReader(path)) {  
        return r.read();  
    }  
}
```

▼ Method Body (Size: 59 bytes, Max Stack: 3, Max Locals: 4)

PC	Line	Instruction	Exceptions
0	138	aconst_null	3: Any
1		astore_1	
2		aconst_null	
3		astore_2	1: Any
4		new java.io.FileReader	
7		dup	
8		aload_0	2: Any
9		invokespecial java.io.FileReader { void <init> (java.lang.String) }	
12		astore_3	
13	139	aload_3	1: Any
14		invokevirtual java.io.FileReader { int read () }	
17	140	aload_3	
18		ifnull 25	2: Any
21		aload_3	
22		invokevirtual java.io.FileReader { void close () }	
25	139	ireturn	2: Any
26		astore_1	
27	140	aload_3	
28		ifnull 35	2: Any
31		aload_3	
32		invokevirtual java.io.FileReader { void close () }	
35		aload_1	2: Any
36		athrow	
37		astore_2	
38		aload_1	2: Any
39		ifnonnull 47	
42		aload_2	
43		astore_1	2: Any
44		goto 57	
47		aload_1	
48		aload_2	2: Any
49		if_acmpeq 57	
52		aload_1	
53		aload_2	2: Any
54		invokevirtual java.lang.Throwable { void addSuppressed (java.lang.Throwable) }	
57		aload_1	
58		athrow	2: Any

```
static int readFirstByte(String path) throws Exception {  
    try (FileReader r = new FileReader(path)) {  
        return r.read();  
    }  
}
```

PC	Line	Instruction	Exceptions
0	138	<i>aconst_null</i>	1: Any
1		<i>astore_1</i>	
2		<i>aconst_null</i>	
3		<i>astore_2</i>	
4		<i>new</i> java.io.FileReader	
7		<i>dup</i>	
8		<i>aload_0</i>	
9		<i>invokespecial</i> java.io.FileReader { void <init> (java.lang.String) }	
12		<i>astore_3</i>	
13	139	<i>aload_3</i>	
14		<i>invokevirtual</i> java.io.FileReader { int read () }	2: Any
17	140	<i>aload_3</i>	
18		<i>ifnull</i> 25	
21		<i>aload_3</i>	
22		<i>invokevirtual</i> java.io.FileReader { void close () }	
25	139	<i>ireturn</i>	
26		<i>astore_1</i>	
27	140	<i>aload_3</i>	
28		<i>ifnull</i> 35	
31		<i>aload_3</i>	
32		<i>invokevirtual</i> java.io.FileReader { void close () }	3: Any
35		<i>aload_1</i>	
36		<i>athrow</i>	
37		<i>astore_2</i>	
38		<i>aload_1</i>	
39		<i>ifnonnull</i> 47	

Java Bytecode - Synchronization

```
public class TACDemo {  
  
    private static volatile TACDemo instance;  
  
    static TACDemo getInstance() {  
        TACDemo instance = TACDemo.instance;  
        // thread-safe double checked locking  
        if (instance == null) {  
            synchronized (TACDemo.class) {  
                instance = TACDemo.instance;  
                if (instance == null) {  
                    instance = new TACDemo();  
                    TACDemo.instance = instance;  
                }  
            }  
        }  
        return instance;  
    }  
}
```

PC	Instruction	Exception Handlers	
0	getstatic TACDemo { TACDemo instance }		
3	astore_0		
4	aload_0		
5	ifnonnull →41		
8	ldc TACDemo.class		
10	dup		
11	astore_1		
12	monitorenter		
13	getstatic TACDemo { TACDemo instance }	1: Any	
16	astore_0		
17	aload_0		
18	ifnonnull →33		
21	new TACDemo		
24	dup		
25	invokespecial TACDemo { void <init> () }		
28	astore_0		
29	aload_0		
30	putstatic TACDemo { TACDemo instance }		
33	aload_1		
34	monitorexit		
35	goto →41		
38	aload_1		2: Any
39	monitorexit		
40	athrow		
41	aload_0		
42	areturn		

```
static <T> List<T> sortIt(List<T> l) {  
    l.sort(  
        (T a, T b) -> { return a.hashCode() - b.hashCode(); }  
    );  
    return l;  
}
```

```

static <T> List<T> sortIt(List<T> l) {
    l.sort((T a, T b) -> { return a.hashCode() - b.hashCode(); });
    return l;
}

```

static java.util.List sortIt(java.util.List)

Signature : <T:Ljava/lang/Object;>(Ljava/util/List<TT;>;)Ljava/util/List<TT;>;

▼ Method Body (Size: 13 bytes, Max Stack: 2, Max Locals: 1)

PC	Line	Instruction
0	164	<i>aload_0</i>
1		<i>invokedynamic</i> (Bootstrap Method Attribute[0], java.util.Comparator compare ())
6		<i>invokeinterface</i> java.util.List { void sort (java.util.Comparator) }
11	167	<i>aload_0</i>
12		<i>areturn</i>

▼ LineNumberTable

start_pc: 0, line_number: 164
start_pc: 11, line_number: 167

▼ LocalVariableTable

pc=[0 → 13) / lv=0 ⇒ java.util.List l

▼ LocalVariableTypeTable

pc=[0 → 13) / lv=0 ⇒ l : Ljava/util/List<TT;>;

```
static <T> List<T> sortIt(List<T> l) {
    l.sort((T a, T b) -> { return a.hashCode() - b.hashCode(); });
    return l;
}
```

static java.util.List sortIt(java.util.List)

Signature : <T:Ljava/lang/Object;>(Ljava/util/List<TT;>;)Ljava/util/List<TT;>;

▼ Method Body (Size: 13 bytes, Max Stack: 2, Max Locals: 1)

PC	Line	Instruction
0	164	<i>aload_0</i>
1		<i>invokedynamic</i> (Bootstrap Method Attribute[0], java.util.Comparator compare ())

MethodHandle(kind=REF_invokeStatic invokestatic C.m:(A*)T, java.lang.invoke.LambdaMetafactory { java.lang.invoke.CallSite metafactory (java.lang.invoke.MethodHandles\$Lookup, java.lang.String, java.lang.invoke.MethodType, java.lang.invoke.MethodType, java.lang.invoke.MethodHandle, java.lang.invoke.MethodType) })

Parameters:

- MethodType(int (java.lang.Object, java.lang.Object))
- MethodHandle(
 - kind=REF_invokeStatic invokestatic C.m:(A*)T,
 - TACDemo { int lambda\$0 (java.lang.Object, java.lang.Object) }**
)
- MethodType(int (java.lang.Object, java.lang.Object))

```
static <T> List<T> sortIt(List<T> l) {  
    l.sort((T a, T b) -> { return a.hashCode() - b.hashCode(); });  
    return l;  
}
```

static java.util.List sortIt(java.util.List)

Signature : <T:Ljava/lang/Object;>(Ljava/util/List<TT;>;)Ljava/util/List<TT;>;

▼ Method Body (Size: 13 bytes, Max Stack: 2, Max Locals: 1)

PC	Line	Instruction
0	164	<i>aload_0</i>
1		<i>invokedynamic</i> (Bootstrap Method Attribute[0], java.util.Comparator compare ())
6		<i>invokeinterface</i> java.util.List (void sort (java.util.Comparator))
11	167	
12		

▼ LineNumberTable

start_pc: 0, line_number: 164
start_pc: 11, line_number: 167

▼ LocalVariableTable

pc=[0 → 13) /

▼ LocalVariableTypeTable

pc=[0 → 13) /

private static [SYNTHETIC] int lambda\$0(java.lang.Object, java.lang.Object)

▼ Method Body (Size: 10 bytes, Max Stack: 2, Max Locals: 2)

PC	Line	Instruction
0	165	<i>aload_0</i>
1		<i>invokevirtual</i> java.lang.Object { int hashCode () }
4		<i>aload_1</i>
5		<i>invokevirtual</i> java.lang.Object { int hashCode () }
8		<i>isub</i>
9		<i>ireturn</i>

▼ LineNumberTable

start_pc: 0, line_number: 165

▼ LocalVariableTable

pc=[0 → 10) / lv=0 ⇒ java.lang.Object a
pc=[0 → 10) / lv=1 ⇒ java.lang.Object b

▼ LocalVariableTypeTable

pc=[0 → 10) / lv=0 ⇒ a : TT;
pc=[0 → 10) / lv=1 ⇒ b : TT;

Analyzing Java Bytecode works (very) well, because
Java compilers deliberately don't optimize.

```
static long optimizableExpression(double d, int i, long l) {  
    return (long)((d * d) + i) * 0l;  
}
```

PC	Instruction
0	dload_0
1	dload_0
2	dmul
3	iload_2
4	i2d
5	dadd
6	d2l
7	lconst_0
8	lmul
9	lreturn

Analyzing Java Bytecode works (very) well, because
Java compilers deliberately don't optimize.

```
static void optimizableIndexInc(int[] is, int i) {  
    is[i++] = 0;  
    is[i++] = 1;  
}
```

PC	Instruction
0	aload_0
1	iload_1
2	iinc (lv=1,val=1)
5	iconst_0
6	iastore
7	aload_0
8	iload_1
9	iinc (lv=1,val=1)
12	iconst_1
13	iastore
14	return

Analyzing Java Bytecode works (very) well, because
Java compilers deliberately don't optimize.

```
int always9() {  
    int i = 3;  
    int j = 3;  
    return i * j;  
}
```

PC	Instruction
0	iconst_3
1	istore_1
2	iconst_3
3	istore_2
4	iload_1
5	iload_2
6	imul
7	ireturn

Analyzing Java Bytecode works (very) well, because
Java compilers deliberately don't optimize.

**Java compilers perform constant propagation for final local
variables and final fields (primitive values and Strings).
Expressions are evaluated if all parameters are primitive constants.**

```
int always6() {  
    final int i = 3;  
    final int j = 2;  
    return i * j;  
}
```

PC	Instruction
0	iconst_3
1	istore_1
2	iconst_2
3	istore_2
4	bipush 6
6	ireturn

```
int always8() {  
    return 4 * 2;  
}
```

PC	Instruction
0	bipush 8
2	ireturn

```
double eXpi() { return Math.E * Math.PI; }
```

PC	Instruction
0	ldc2_w 8.539734222673566d
3	dreturn

Sources of Potentially Dead Code Created by Java Compilers

- Finally blocks are generally included twice.
- Switches always have default branches.
- Constant expressions are evaluated.
- Constant propagation for final (local) variables/final fields is performed. (This includes primitive types and “Strings”).

Other Peculiarities

- Types are represented using binary notation. In binary notation packages are separated using “/”: e.g., java/lang/Object.
- The JVM has no “negate” instruction. A negation in Java (!b) is compiled to an if instruction followed by a push of the corresponding value.
- The JVM has no direct support for shortcut-evaluation (&&, ||).
- (Reliably) identifying anonymous inner classes is broken for older class files.
- The switch instructions are four byte aligned.
- The instruction set is not orthogonal; i.e., to achieve a certain effect many instructions exist.
- The catch block is not immediately available.
- String concatenation is compiled using StringBuilders.
- Varargs are put in arrays.

Three-Address Code

Three-Address Code (TAC)

- Three-address code is a sequence of statements with the general form:
`x = y op z`
- where x,y and z are (local variable) names, constants (in case of y and z) or compiler-generated temporaries
- The name was chosen, because “most” statements use three addresses: two for the *operators* and *one to store the result*

General Types of Three-Address Statements

- **Assignment** statements $x = y \text{ bin_op } z$ or $x = \text{unary_op } z$
- **Copy** statements $x = y$
- **Unconditional jumps**: `goto l` (and `jsr l`, `ret` in case of Java bytecode)
- **Conditional jumps**: `if (x rel_op y) goto l` (else fall through), `switch`
- **Method call and return**: `invoke(m, params)`, `return x`
- **Array access**: `a[i]` or `a[i] = x`
- *More IR specific types.*

Converting Java Bytecode to Three-Address Code

- Core Idea:
 - Compute for each instruction the current stack layout by following the control flow; i.e., compute the types of values found on the stack before the instruction is evaluated.
(The JVM specification guarantees that the operand stack always has the same layout independent of the taken path.)
 - Assign each local variable to a variable where the name is based on the local variable index.
E.g., an `iinc(lv=1, val=1)` instruction is transformed into the three address code: $r_{\underline{1}} = r_{\underline{1}} + 1$
 - Assign each variable on the operand stack to a corresponding local variable with an index based on the position on the stack.
E.g., if the operand stack is empty and we push the constant 1, then the three address code would be: $op_0 = 1$; if we would then push another value 2 then the code would be: $op_1 = 2$ and an addition of the two values would be: $op_0 = op_0 + op_1$

Converting Java Bytecode to Three-Address Code

```
static int numberOfDigits(int i) {  
    return ((int) Math.floor(Math.log10(i))) + 1;  
}
```

PC	Instruction	Stack Layout (before execution)	Three-Address Code
	<i>initialization</i>		<i>r_0 = i; // parameter</i>
0	iload_0		op_0 = r_0;
1	i2d	0: Integer Value	op_0 = (double) op_0;
2	invokestatic java.lang.Math.log10 (double):double	0: Double Value	op_0 = Math.log10(op_0);
5	invokestatic java.lang.Math.floor(double):double	0: Double Value	op_0 = Math.floor(op_0);
8	d2i	0: Double Value	op_0 = (int) op_0;
9	iconst_1	0: Integer Value	op_1 = 1;
10	iadd	1: Integer Value 0: Integer Value	op_0 = op_0 + op_1;
11	ireturn	0: Integer Value	return op_0;

Java Bytecode vs. Three-Address Code

Java

```
static int max(int i, int j) {  
    if (i > j)  
        return i;  
    else  
        return j;  
}
```

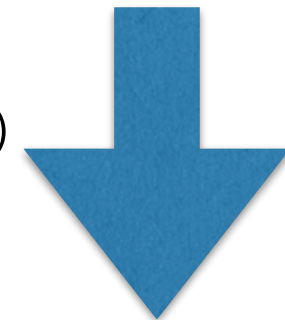
Bytecode

PC	Instruction
0	iload_0
1	iload_1
2	if_icmple →7
5	iload_0
6	ireturn
7	iload_1
8	ireturn

Three Address Code

```
0: r_0 = i;  
1: r_1 = j;  
2: op_0 = r_0;  
3: op_1 = r_1;  
4: if(op_0 <= op_1) goto 7;  
5: op_0 = r_0;  
6: return op_0;  
7: op_0 = r_1;  
8: return op_0;
```

After (Peephole)
Optimizations



```
0: if(i <= j) goto 2;  
1: return i;  
2: return j;
```

Java Bytecode vs. Three-Address Code

Java

```
static int factorial(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

Bytecode

PC	Instruction
0	iconst_1
1	istore_1
2	goto →12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc 0, -1
12	iload_0
13	ifgt →5
16	iload_1
17	ireturn

Three Address Code

```
0: r_0 = n;  
1: op_0 = 1;  
2: r_1 = op_0;  
3: goto 9;  
4: op_0 = r_1;  
5: op_1 = r_0;  
6: op_0 = op_0 * op_1;  
7: r_1 = op_0;  
8: r_0 = r_0 + -1;  
9: op_0 = r_0;  
10: if(op_0 > 0) goto 4;  
11: op_0 = r_1;  
12: return op_0;
```



After (Peephole) Optimizations

```
0: r_0 = n;  
1: r_1 = 1;  
2: goto 5;  
3: r_1 = r_1 * r_0;  
4: r_0 = r_0 + -1;  
5: if(r_0 > 0) goto 3;  
6: return r_1;
```

A Three-Address Code for Java Bytecode

Basic Statements

```
ASSIGNMENT(  
    pc:          PC,  
    targetVar:   VAR,  
    expr:        EXPR  
)  
  
GOTO(pc: PC, target: Int)  
  
JUMPTOSUBROUTINE(pc: PC, target: Int)  
RET(pc: PC, returnAddressVar: VAR)  
  
NOP(pc: PC)  
  
IF(  
    pc:          PC,  
    left:        VALEXPR,  
    condition:   RelationalOperator,  
    right:       VALEXPR,  
    target:      Int  
)  
SWITCH(  
    pc:          PC,  
    defaultTarget: PC,  
    index:       VALEXPR,  
    npairs:      IndexedSeq[(Int, PC)]  
)  
  
RETURNVALUE(pc: PC, expr: VALEXPR)  
RETURN(pc: PC)  
  
ARRAYSTORE(  
    pc:          PC,  
    arrayRef:    VAR,  
    index:       VALEXPR,  
    value:       VALEXPR  
)  
  
THROW(pc: PC, exception: VAR)  
  
MONITORENTER(pc: PC, objRef: VAR)  
MONITOREXIT(pc: PC, objRef: VAR)
```

A Three-Address Code for Java Bytecode

Field Access and Method Call Statements

```
PUTSTATIC(  
    pc:           PC,  
    declaringClass: ObjectType,  
    name:         String,  
    value:        VAL_EXPR  
)  
PUTFIELD(  
    pc:           PC,  
    declaringClass: ObjectType,  
    name:         String,  
    objRef:       VAR,  
    value:        VAL_EXPR  
)
```

```
NONVIRTUALMETHODCALL(  
    pc:           PC,  
    declaringClass: ReferenceType,  
    name:         String,  
    descriptor:   MethodDescriptor,  
    receiver:     VAR,  
    params:       List[VAL_EXPR]  
)  
VIRTUALMETHODCALL(  
    pc:           PC,  
    declaringClass: ReferenceType,  
    name:         String,  
    descriptor:   MethodDescriptor,  
    receiver:     VAR,  
    params:       List[VAL_EXPR]  
)  
STATICMETHODCALL(  
    pc:           PC,  
    declaringClass: ReferenceType,  
    name:         String,  
    descriptor:   MethodDescriptor,  
    params:       List[VAL_EXPR]  
)
```

A Three-Address Code for Java Bytecode

Expressions

INSTANCEOF(pc: PC, value: VAR, t: ReferenceType)

CHECKCAST(pc: PC, value: VAR, t: ReferenceType)

COMPARE(

pc: PC,
left: VAL_EXPR,
condition: RelationalOperator,
right: VAL_EXPR

)

BINARY_EXPR(

pc: PC,
cTpe: ComputationalType,
op: BinaryArithmeticOperator,
left: VAL_EXPR, right: VAL_EXPR

)

PREFIX_EXPR(

pc: PC,
cTpe: ComputationalType,
op: UnaryArithmeticOperator,
operand: VAL_EXPR

)

PRIMITIVE_TYPECAST_EXPR(

pc: PC,
targetTpe: BaseType,
operand: VAL_EXPR

)

NEW(pc: PC, tpe: ObjectType)

NEW_ARRAY(

pc: PC,
counts: List[VAL_EXPR],
tpe: ArrayType

)

ARRAY_LOAD(pc: PC, index: Var, arrayRef: Var)

ARRAY_LENGTH(pc: PC, arrayRef: Var)

VAL_EXPR

PARAM(cTpe: ComputationalType, name: String)

SIMPLE_VAR(id: Int, cTpe: ComputationalType)

INT_CONST(pc: PC, value: Int)

LONG_CONST(pc: PC, value: Long)

FLOAT_CONST(pc: PC, value: Float)

DOUBLE_CONST(pc: PC, value: Double)

STRING_CONST(pc: PC, value: String)

CLASS_CONST(pc: PC, value: ReferenceType)

NULL_EXPR(pc: PC)

A Three-Address Code for Java Bytecode

Expressions

```
GETFIELD(  
    pc:          PC,  
    declaringClass: ObjectType,  
    name:        String,  
    objRef:      ValExpr  
)  
GETSTATIC(  
    pc:          PC,  
    declaringClass: ObjectType,  
    name:        String  
)  
INVOKEDYNAMIC(  
    pc:          PC,  
    bootstrapMethod: BootstrapMethod,  
    name:        String,  
    descriptor:  MethodDescriptor,  
    params:      List[Expr]  
)  
METHODTYPECONST(pc: PC, desc: MethodDescriptor)  
METHODHANDLECONST(pc: PC, desc: MethodHandle)
```

```
NONVIRTUALFUNCTIONCALL(  
    pc:          PC,  
    declaringClass: ReferenceType,  
    name:        String,  
    descriptor:  MethodDescriptor,  
    receiver:    Expr,  
    params:      List[Expr]  
)  
VIRTUALFUNCTIONCALL(  
    pc:          PC,  
    declaringClass: ReferenceType,  
    name:        String,  
    descriptor:  MethodDescriptor,  
    receiver:    Expr,  
    params:      List[Expr]  
)  
STATICFUNCTIONCALL(  
    pc:          PC,  
    declaringClass: ReferenceType,  
    name:        String,  
    descriptor:  MethodDescriptor,  
    params:      List[Expr]  
)
```


Control-Flow Graph

- The control-flow graph (CFG) represents the control flow of a single method.
- Each node represents a **basic block**. A basic block is a maximal-length sequence of statements without jumps in and out (and no exceptions are thrown by intermediate instructions).
- The arcs represent the inter-node control flow.

Control-Flow Graph

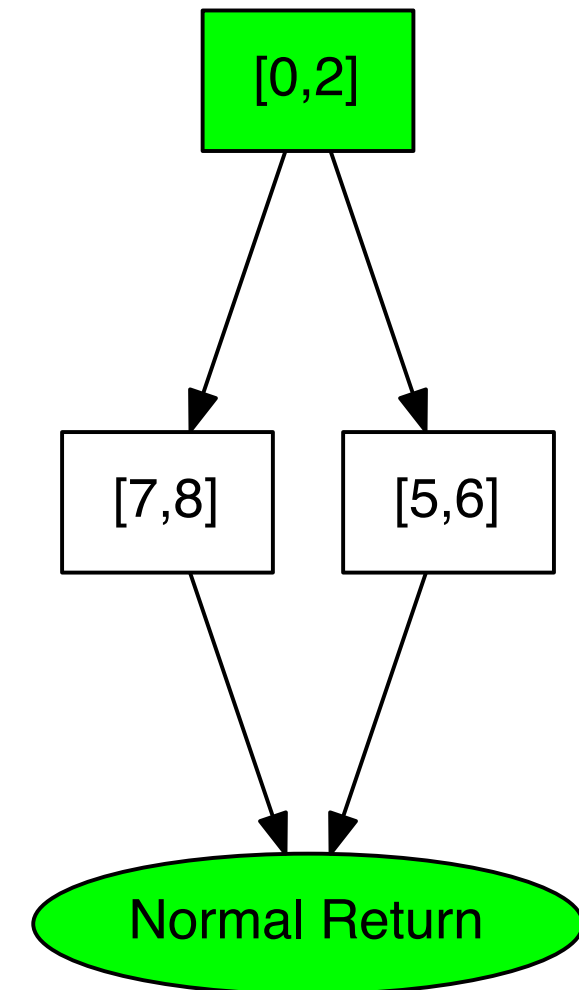
Java

```
static int max(int i, int j) {  
    if (i > j)  
        return i;  
    else  
        return j;  
}
```

Bytecode

PC	Instruction
0	iload_0
1	iload_1
2	if_icmple →7
5	iload_0
6	ireturn
7	iload_1
8	ireturn

CFG



Loops - Control-Flow Graph

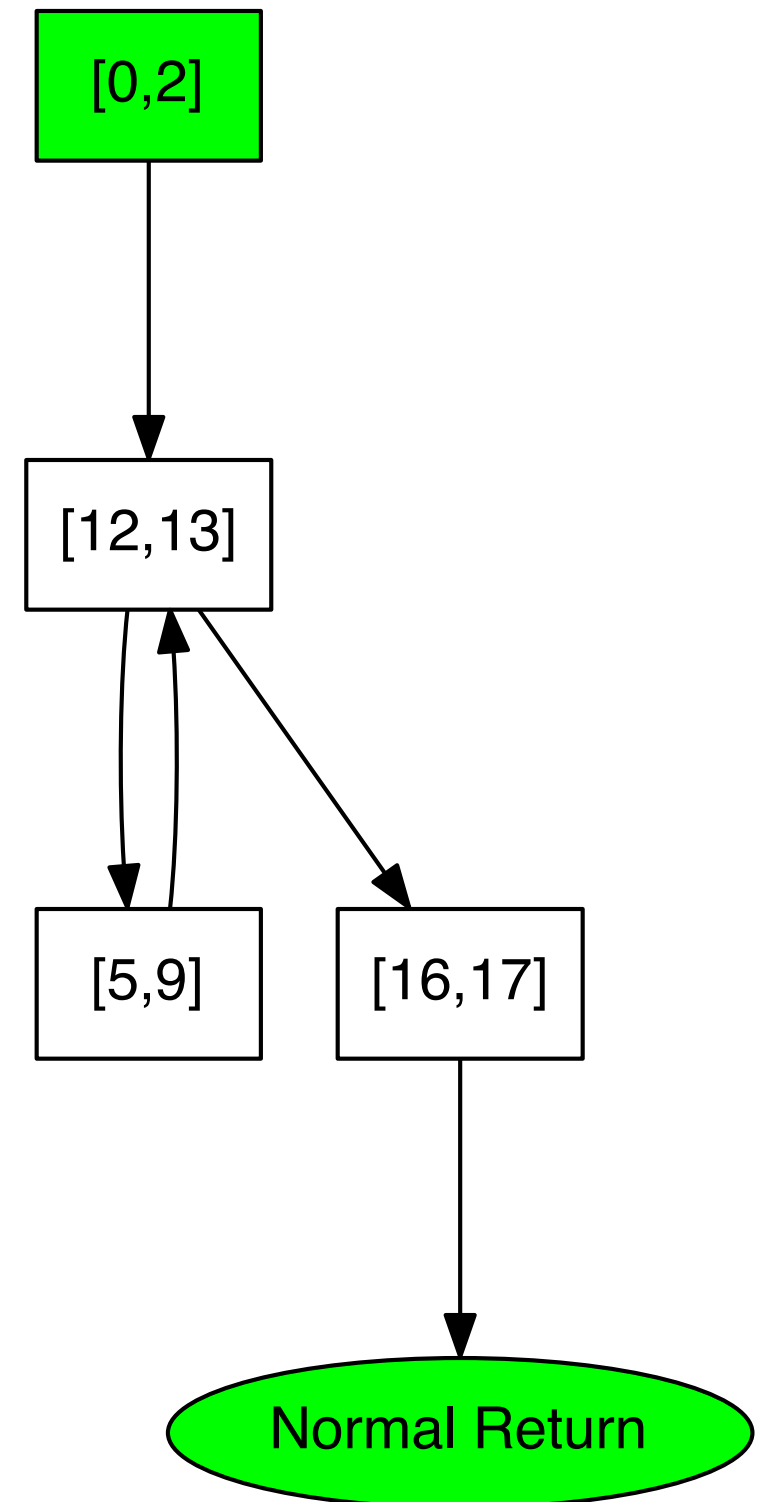
Java

```
static int factorial(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

Bytecode

PC	Instruction
0	iconst_1
1	istore_1
2	goto →12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc (lv=0,val=-1)
12	iload_0
13	ifgt →5
16	iload_1
17	ireturn

CFG

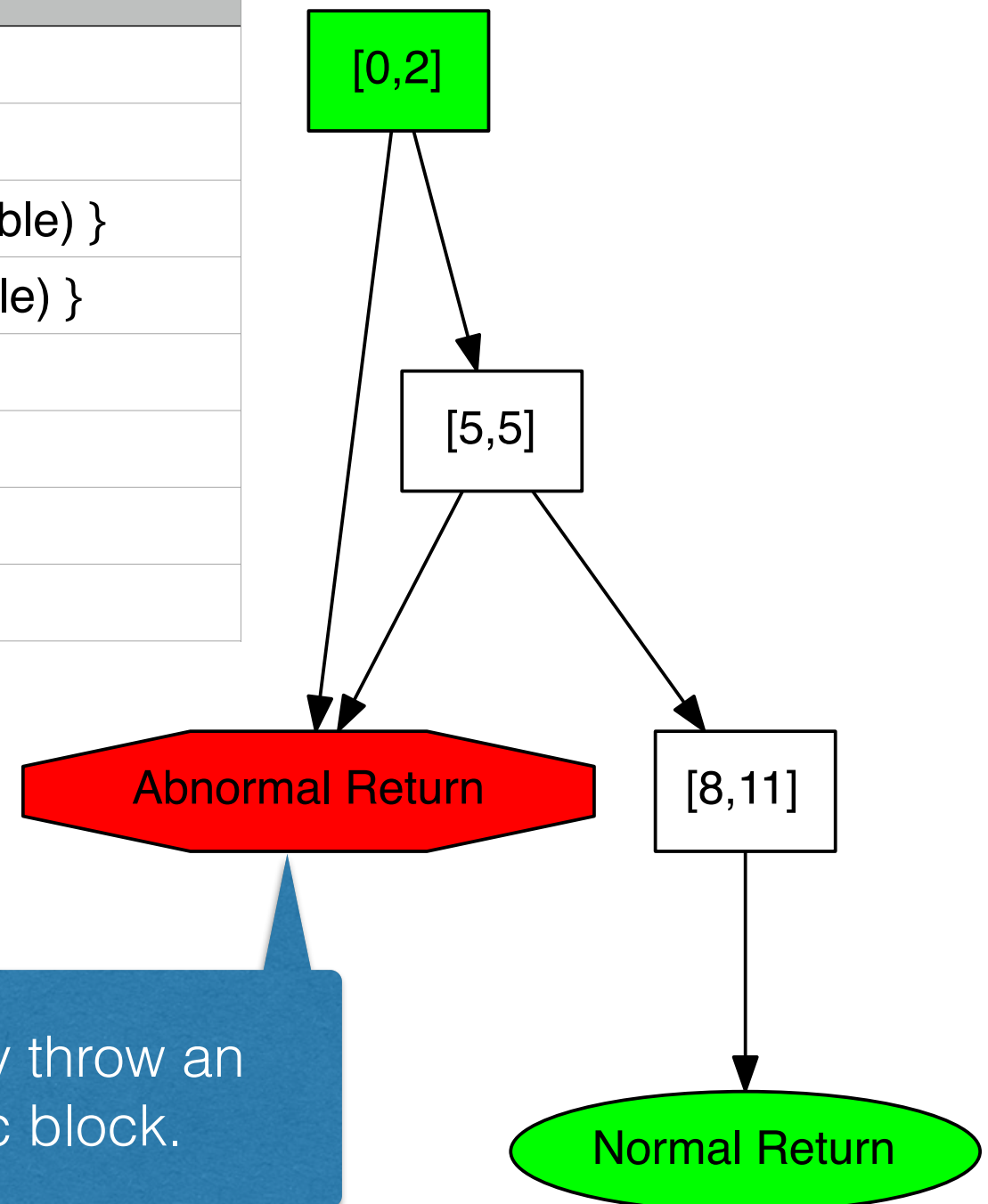


Java Bytecode

```
static int baseNumberOfDigits(int i) {  
    return ((int) Math.floor(Math.log10(i))) + 1;  
}
```

CFG

PC	Instruction
0	iload_0
1	i2d
2	invokestatic java.lang.Math { double log10 (double) }
5	invokestatic java.lang.Math { double floor (double) }
8	d2i
9	iconst_1
10	iadd
11	ireturn



Every instruction that may throw an exception ends a basic block.

Java Bytecode - Finally

```
int tryFinally(Object o) {
    int result = 0;
    try {
        if (o instanceof List<?>) {
            result = ((List<?>) o).size();
        } else {
            result = -1;
        }
        return result;
    } finally {
        result += 1;
    }
}
```

▼ Method Body (Size: 39 bytes, Max Stack: 1, Max Locals: 5)

PC	Line	Instruction	Exceptions	
0	122	<i>iconst_0</i>		
1		<i>istore_2</i>		
2	124	<i>aload_1</i>		1: Any
3		<i>instanceof</i>	java.util.List	
6		<i>ifeq</i>	<u>22</u>	
9	125	<i>aload_1</i>		
10		<i>checkcast</i>	java.util.List	
13		<i>invokeinterface</i>	java.util.List { int size () }	
18		<i>istore_2</i>		
19	126	<i>goto</i>	<u>24</u>	
22	127	<i>iconst_m1</i>		
23		<i>istore_2</i>		
24	129	<i>iload_2</i>		
25		<i>istore</i>	4	
27	131	<i>iinc</i>	2	1
30	129	<i>iload</i>	4	
32		<i>ireturn</i>		
33	130	<i>astore_3</i>		
34	131	<i>iinc</i>	2	1
37	132	<i>aload_3</i>		
38		<i>athrow</i>		

▼ Exception Table:

1. try [2-27) catch 33 Any

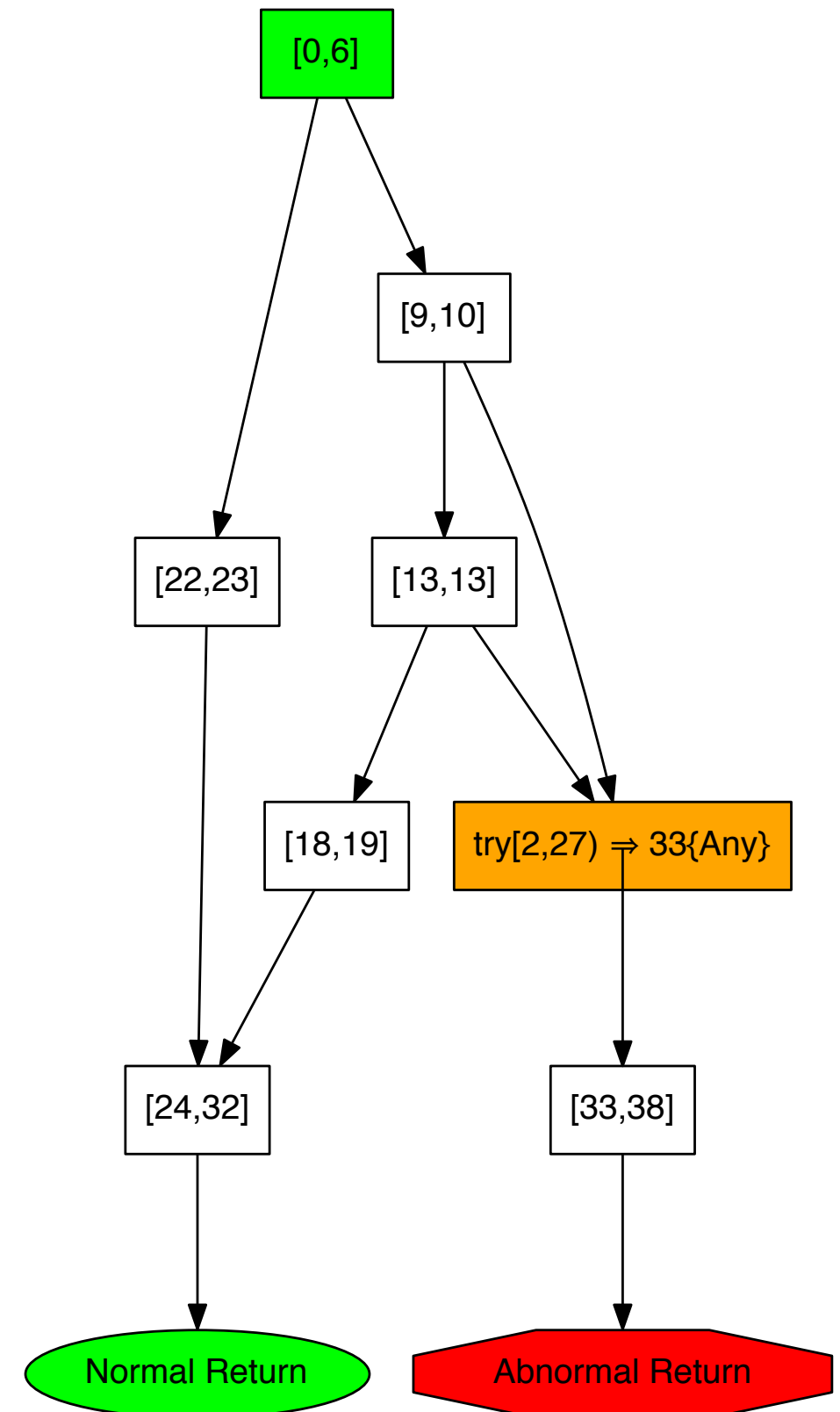
Java Bytecode - Finally

▼ Method Body (Size: 39 bytes, Max Stack: 1, Max Locals: 5)

PC	Line	Instruction	Exceptions
0	122	<i>iconst_0</i>	
1		<i>istore_2</i>	
2	124	<i>aload_1</i>	
3		<i>instanceof</i>	java.util.List
6		<i>ifeq</i>	<u>22</u>
9	125	<i>aload_1</i>	
10		<i>checkcast</i>	java.util.List
13		<i>invokeinterface</i>	java.util.List { int size () }
18		<i>istore_2</i>	
19	126	<i>goto</i>	<u>24</u>
22	127	<i>iconst_m1</i>	
23		<i>istore_2</i>	
24	129	<i>iload_2</i>	
25		<i>istore</i>	4
27	131	<i>iinc</i>	2 1
30	129	<i>iload</i>	4
32		<i>ireturn</i>	
33	130	<i>astore_3</i>	
34	131	<i>iinc</i>	2 1
37	132	<i>aload_3</i>	
38		<i>athrow</i>	

▼ Exception Table:

1. try [2-27) catch 33 Any



The CFG is an over approximation; it represents all paths.

```
static boolean checkIt(int a, int b) {  
    if (b < 0) {  
        if (a < 100) {  
            if( b > 1000)  
                return true;  
            else  
                return false;  
        }  
    }  
    return true;  
}
```

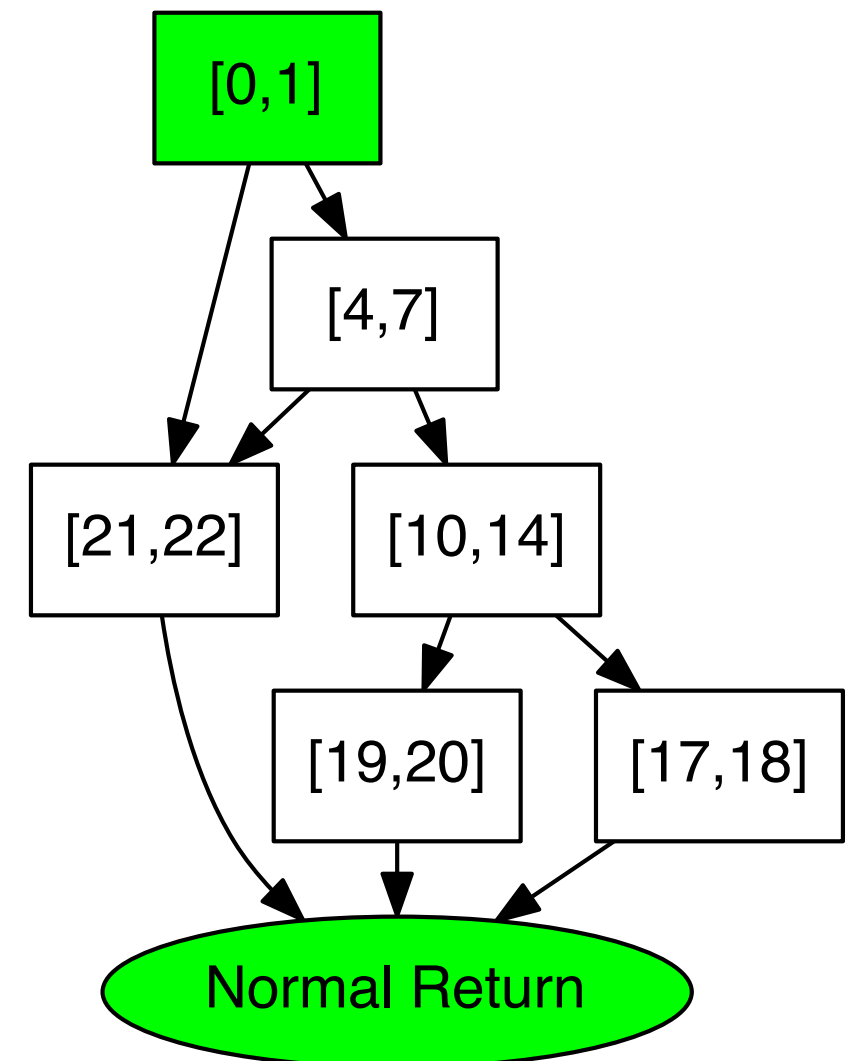
Java

Bytecode

PC Instruction

0	iload_1
1	ifge21
4	iload_0
5	bipush 100
7	if_icmpge →21
10	iload_1
11	sipush 1000
14	if_icmple →19
17	iconst_1
18	ireturn
19	iconst_0
20	ireturn
21	iconst_1
22	ireturn

CFG



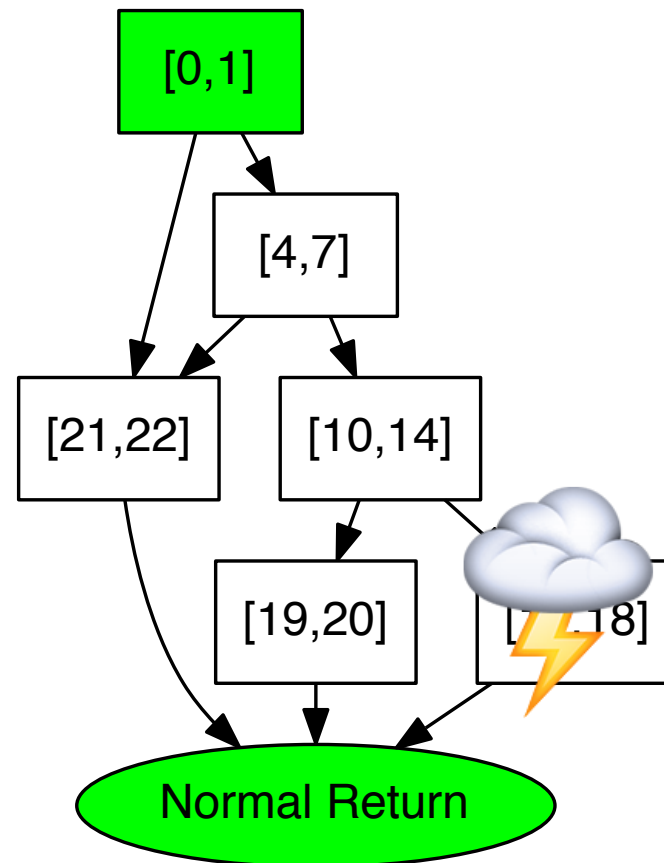
The CFG is an over approximation.

Bytecode

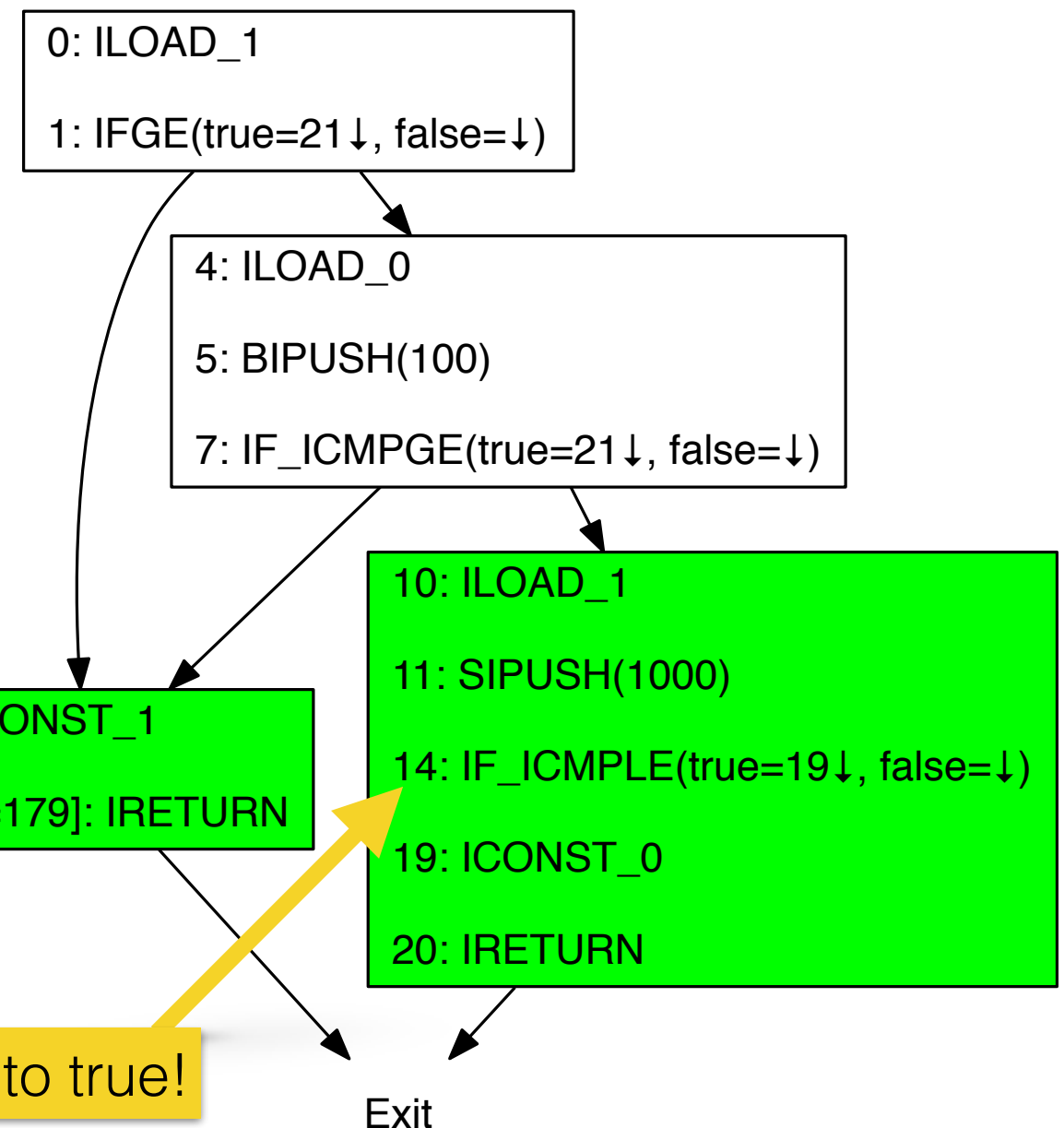
PC Instruction

0	iload_1
1	ifge21
4	iload_0
5	bipush 100
7	if_icmpge → 21
10	iload_1
11	sipush 1000
14	if_icmple → 19
17	iconst_1
18	ireturn
19	iconst_0
20	ireturn
21	iconst_1
22	ireturn

Conservative CFG



CFG computed based on the result of an abstract interpretation



The "if" always evaluates to true!

Dominator Tree

- A node d of a (control-) flow graph dominates node n , if every path from the *initial node* of the flow graph to n goes through d .
- In a dominator tree the initial node is the root and each node d only dominates its descendants.
- Dominator information is useful in identifying loops; by identify the header and the back edge.

Loops - CFG and Dominator Tree

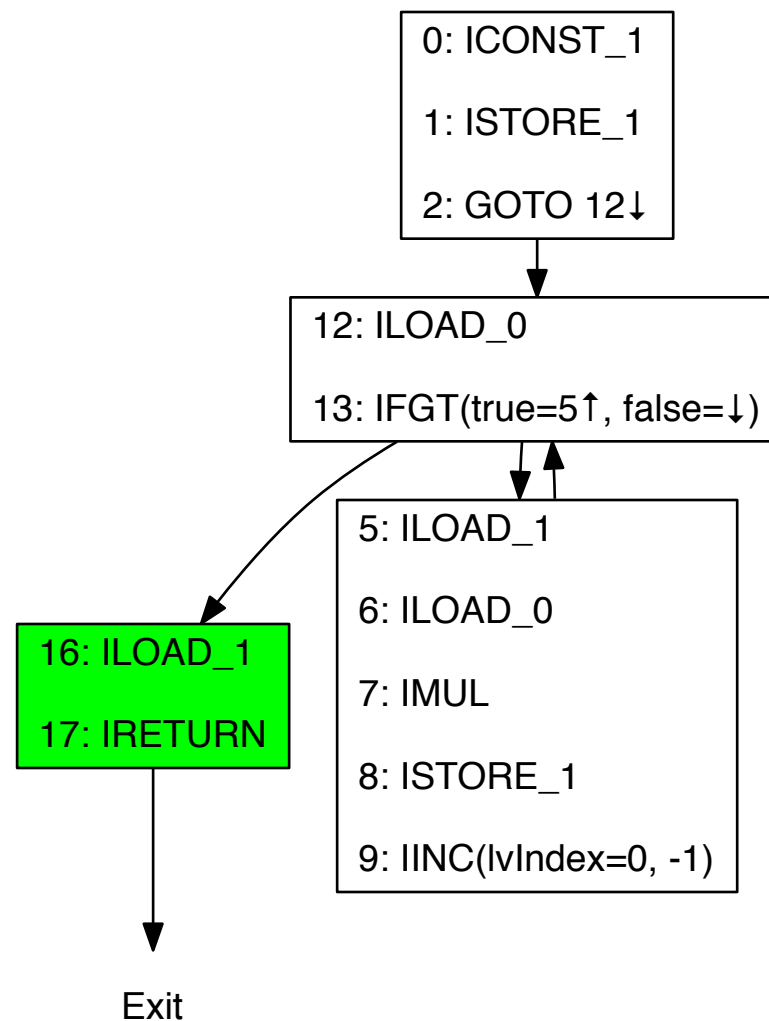
Java

```
static int factorial(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

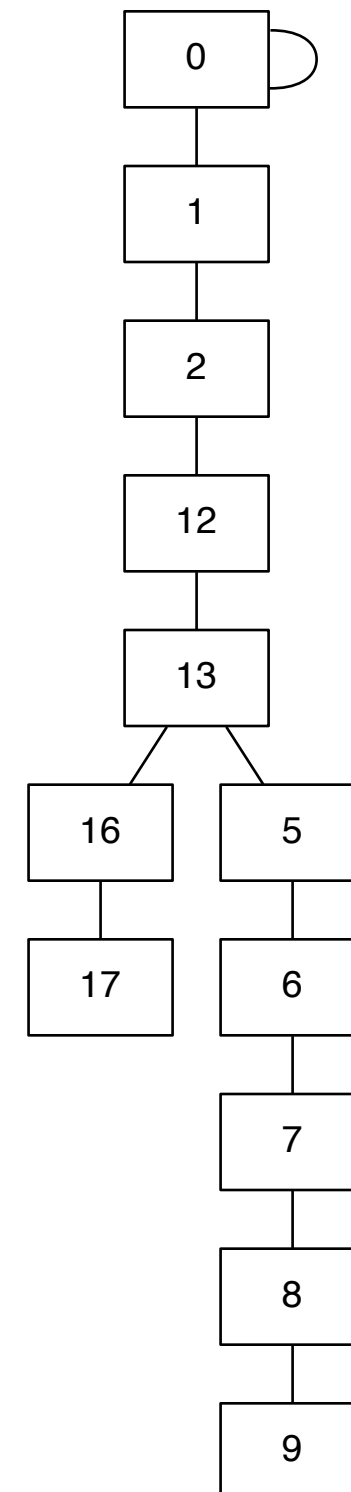
Bytecode

PC	Instruction
0	iconst_1
1	istore_1
2	goto →12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc (lv=0,val=-1)
12	iload_0
13	ifgt →5
16	iload_1
17	ireturn

CFG



Dominator Tree
(Instruction Level)



Loops - Def-Use Dependencies

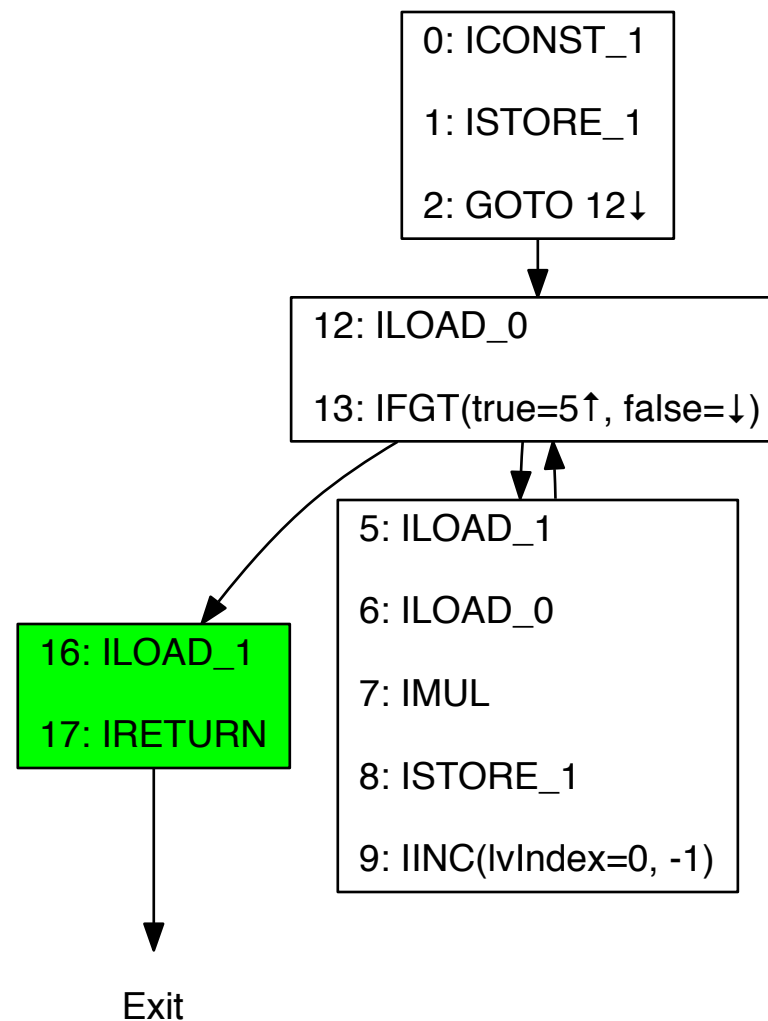
Java

```
static int factorial(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

Bytecode

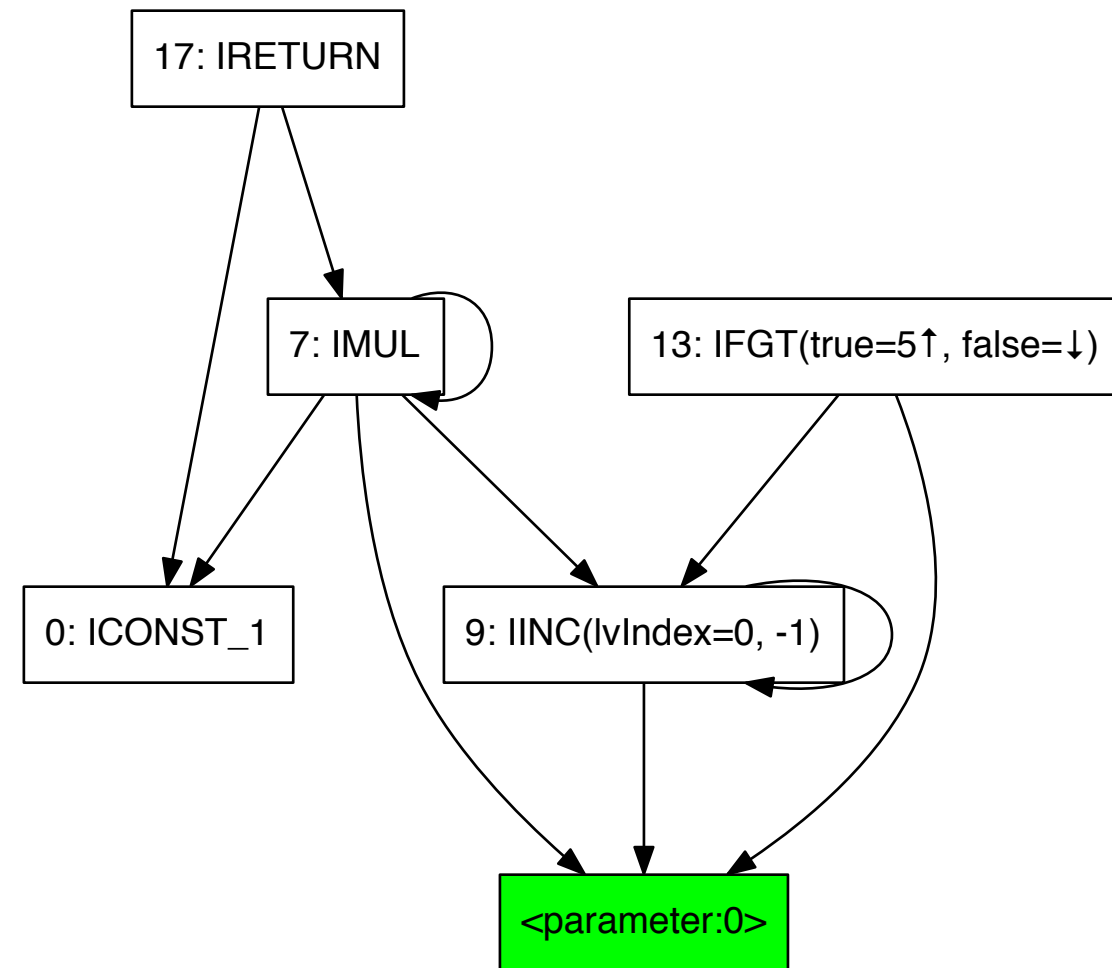
PC	Instruction
0	iconst_1
1	istore_1
2	goto →12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc (lv=0,val=-1)
12	iload_0
13	ifgt →5
16	iload_1
17	ireturn

CFG



Def-Use dependencies at the level of instructions.

Here, an instruction depends on another instruction if it uses the result of the other instruction.



SSA Code

(STATIC SINGLE ASSIGNMENT (FORM))

- Motivation: Many analyses require def-use information. I.e., the information where a used local variable is defined or vice versa.
- In SSA form each variable has only one static definition-site in the program text.
- When two control-flow paths merge, a selector function ϕ is used that initializes the variable based on the control flow that was taken.
- SSA form facilitates data-flow analyses; e.g., facilitates the elimination of redundant loads and computations *across* basic block boundaries.

Java Bytecode vs. Three-Address Code vs. SSA

Java

```
static int factorial(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

Three Address Code

```
0: r_0 = n;  
1: r_1 = 1;  
2: goto 5;  
3: r_1 = r_1 * r_0;  
4: r_0 = r_0 + -1;  
5: if(r_0 > 0) goto 3;  
6: return r_1;
```

SSA

```
0: r_0 = n;  
1: r_1 = 1;  
2: goto 5;
```

```
3: r_2 = r_7 * r_6;  
4: r_3 = r_6 - 1;
```

```
r_6 =  $\phi$ (r_0, r_3)  
r_7 =  $\phi$ (r_1, r_2)  
5: if(r_6 > 0) goto 3;
```

```
6: return r_7;
```

Class Hierarchy

- A core data-structure which encodes the project's class hierarchy and which offers query functionality to get a type's supertypes and subtypes.
- In Java `java.lang.Object` is the super type of all types (including interface types). I.e., every interface (in the byte code explicitly) inherits from `Object`.
- In practice it is - when you want to analyze libraries - always the case that the class hierarchy contains "holes" (is not upwards closed).

