Applied Static Analysis 2016

Dr. Michael Eichberg (Organizer)
Johannes Lerch, Ben Hermann, Sebastian Proksch, Karim Ali Ph.D.

Software Technology Group

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

3

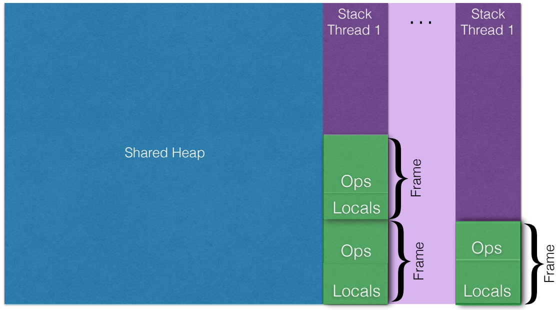
“boolean” only has special support w.r.t. the creation of arrays.

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)

4

Structure of the Java Virtual Machine (JVM)



5

The size of the operand stack and the size of the locals depends on the called method.

Category two values require two slots on the operand stack / two locals.

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)

6

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`)

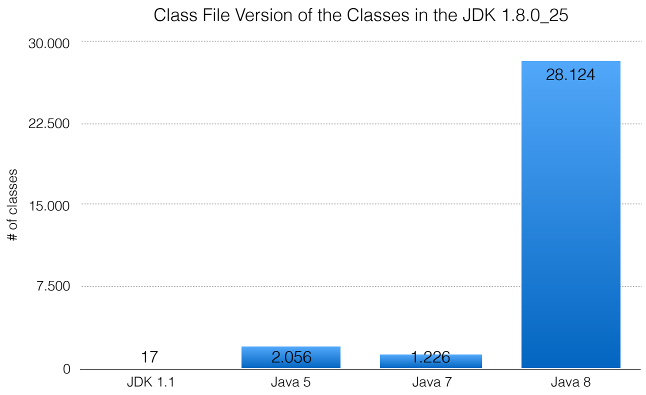
7

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`)

8

Obsolete Code in the JDK



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

11

The parameters are stored in the locals when the method is invoked. Recall that double and long values need two slots.
(In case of instance methods, the implicit `this` parameter is stored in local variable 0.)

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

12

Notice the compilation of the if statement. It is common that in the bytecode the if operator is the inverse one, because if the condition evaluates to true, we then perform the jump (to the else branch) while in the source code, we simply fall through in case the condition evaluates to true.

Java Bytecode - Loops

```java
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

13

When we have a loop, we will always find an unconditional goto in the bytecode. Here, the if condition is actually the same as in the source code!

By moving the test to the end, we have no goto instruction while executing the nth iteration (n > 1).

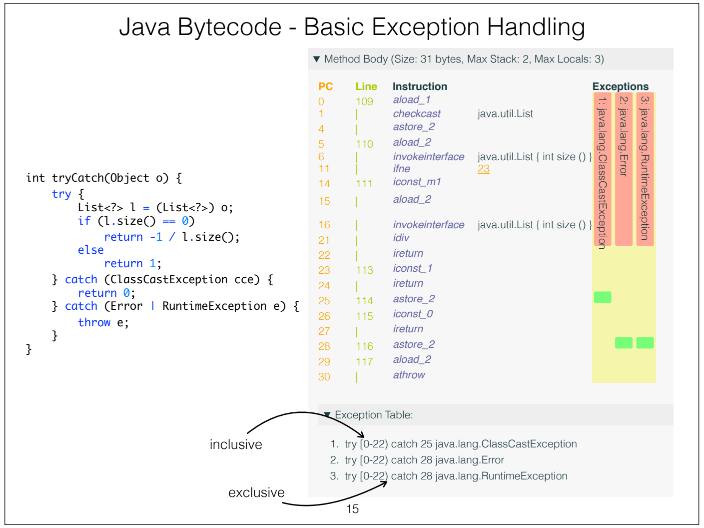
Java Bytecode - Object Creation

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

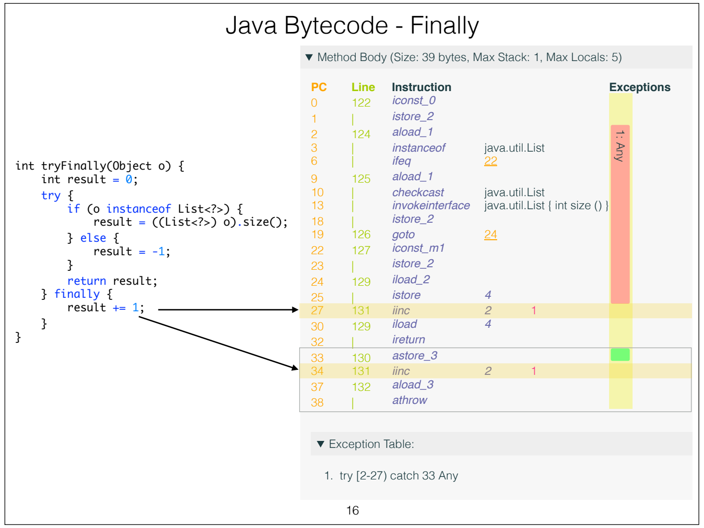
PC	Instruction	Code
0	getstatic TACDemo { boolean $assertionsDisabled }	assert(i > 0)
3	ifne →18	
6	iload_0	
7	ifgt →18	
10	new java.lang.AssertionError	
13	dup	
14	invokespecial java.lang.AssertionError { void <init> () }	
17	athrow	
18	iload_0	Math.log10(i)
19	i2d	
20	invokestatic java.lang.Math {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

14

Initialization of an object is done in two steps. First allocation of the memory (using new) and then calling the constructor (<init>).
Assertions are compiled to a field access instruction to check if assertions are enabled. Afterwards (conditionally), the assertion is checked.



At runtime the first handler that can handle an exception will be invoked.



The finally block is generally included twice - independent of its size! Once, for the case if no exception is thrown and once for the case when an exception is thrown.

The stream is closed in the regular case, but also if an exception occurs - even if the exception occurs while closing the stream!

A try-with-resources statement can handle the case where the resource is “null”.

Exception handlers:

try [13-17) catch 26 Any

try [25-26) catch 26 Any

try [4-37) catch 37 Any

Java Bytecode - Synchronization

```java
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		

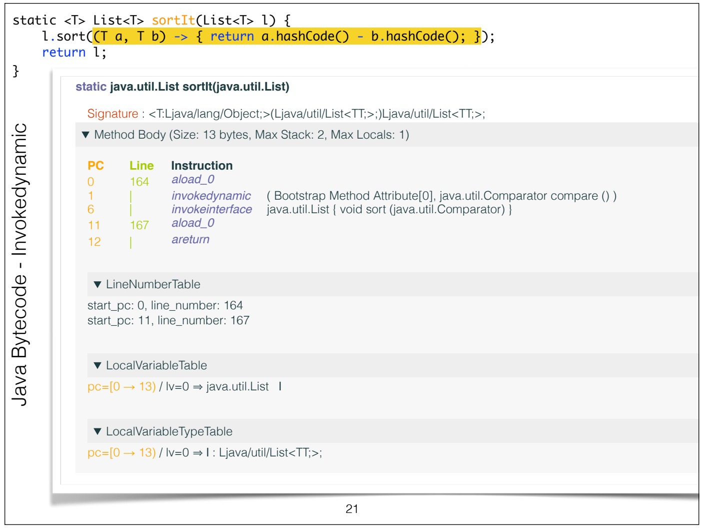
19

`synchronized` blocks are translated to monitorenter/monitorexit(+) instructions.
`synchronized` blocks will always result in a try-finally handler to ensure that the lock is released even in case of an exception!

Java Bytecode - Invokedynamic

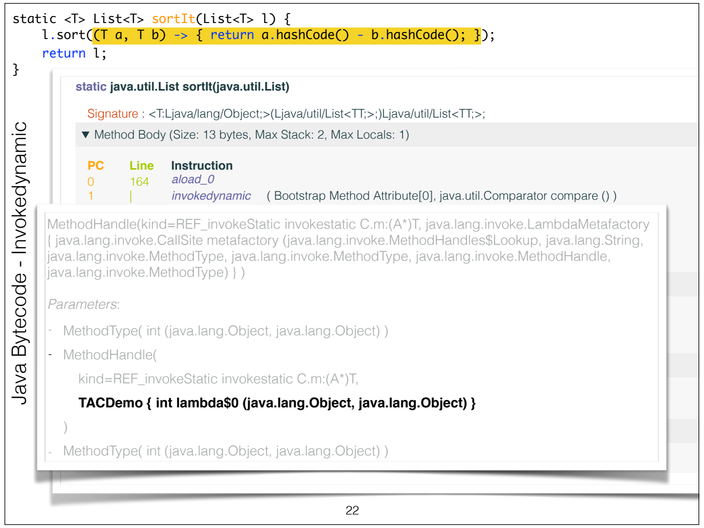
```java
static <T> List<T> sortIt(List<T> l) {
    l.sort(
      (T a, T b) -> { return a.hashCode() - b.hashCode(); }
    );
    return l;
}
```

20

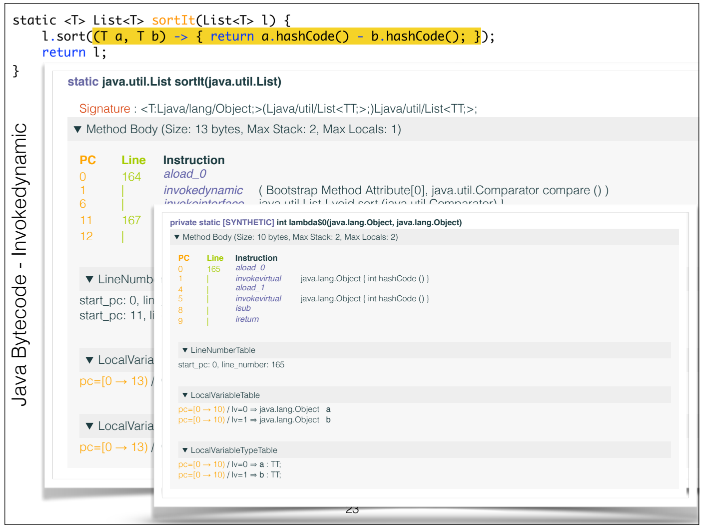


Lambda expression in Java source code are compile using invoke dynamic instructions.

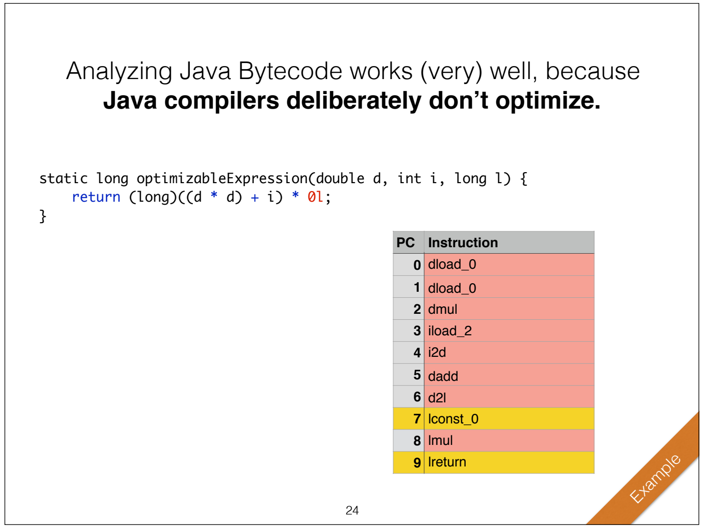
Some information is optional; e.g., the line number table and also the local variable (type) tables. However, the signature is practically not optional for libraries.



The Bootstrap Method Attribute [0].



The “lambda method”.



The result of the expression is always “0” and this could be easily detected, but the compiler won’t optimize it. However, the JVM will (most likely) do it.

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

25

Example

Here, the second iinc instruction is useless and this could be easily detected, but it is still not optimized.

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

26

Example

~~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

Example

Summary

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 exit.
- The catch block is not immediately available.

29

Additional differences compared to Java are: String concatenation is compiled using StringBuilders and Varargs are always stored 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 bin_op z` or `x = 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.*

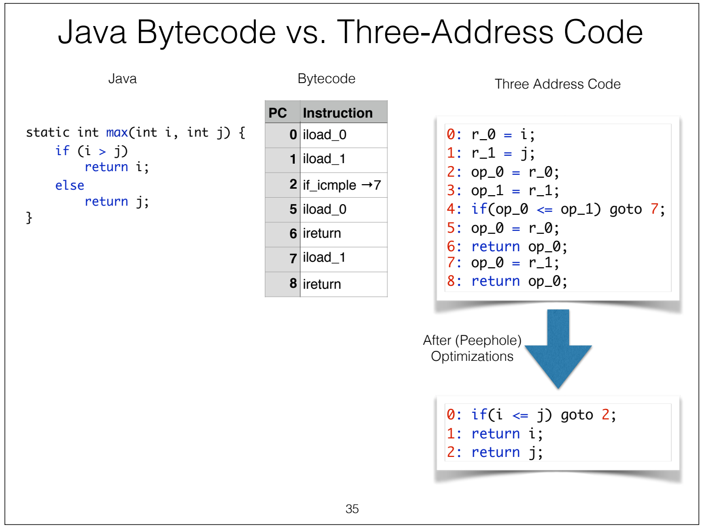
Converting Java Bytecode to Three-Address Code

- 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_1 = r_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

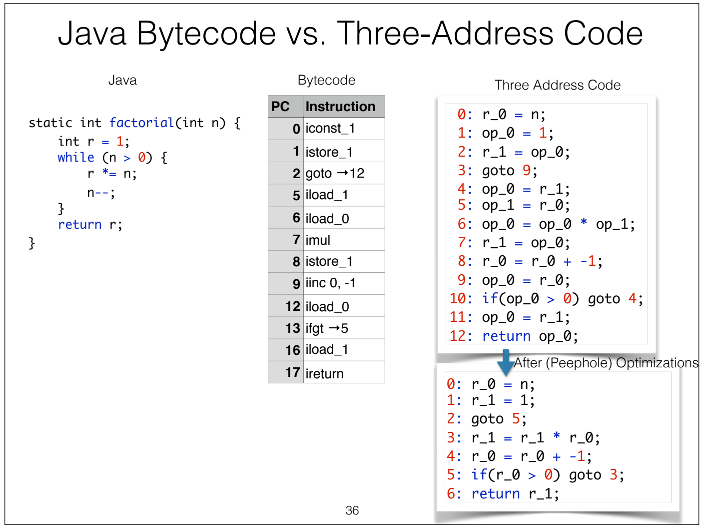
```
static int numberOfDigits(int i) {
    return ((int) Math.floor(Math.log10(i))) + 1;
}
```

PC	Instruction	Stack Layout (before execution)	Three-Address Code
	initialization		`r_0 = i; // parameter`
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;`



Peephole optimizations use a “sliding window” over the cfg’s basic blocks (discussed later) to perform, e.g., the following optimizations:

- elimination of redundant loads and stores
- constant folding
- constant propagation
- common subexpression elimination
- copy propagation
- strength reduction (`x * 2 => x +x`; `x / 2 = x >> 1`)
- elimination of useless instructions (`y = x * 0 => y = 0`; **may cause false negatives**!)
- *exploitation of the instruction set (not really possible for Java bytecode)*



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

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

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:      ValExpr,
        condition: RelationalOperator,
        right:     ValExpr
)

BinaryExpr(
        pc:   PC,
        cTpe: ComputationalType,
        op:   BinaryArithmeticOperator,
        left: ValExpr, right: ValExpr
)

PrefixExpr(
        pc:      PC,
        cTpe:    ComputationalType,
        op:      UnaryArithmeticOperator,
        operand: ValExpr
)

PrimitiveTypecastExpr(
        pc:         PC,
        targetTpe:  BaseType,
        operand:    ValExpr
)

New(pc: PC, tpe: ObjectType)

NewArray(
        pc:     PC,
        counts: List[ValExpr],
        tpe:    ArrayType
)
ArrayLoad(pc: PC, index: Var, arrayRef: Var)
ArrayLength(pc: PC, arrayRef: Var)
```

ValExpr

```
Param(cTpe: ComputationalType, name: String)
SimpleVar(id: Int, cTpe: ComputationalType)

IntConst(pc: PC, value: Int)
LongConst(pc: PC, value: Long)
FloatConst(pc: PC, value: Float)
DoubleConst(pc: PC, value: Double)
StringConst(pc: PC, value: String)
ClassConst(pc: PC, value: ReferenceType)

NullExpr(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

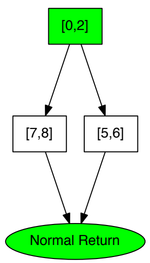
Java

```
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

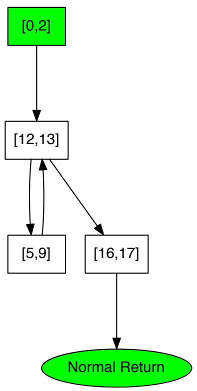
Java

```
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

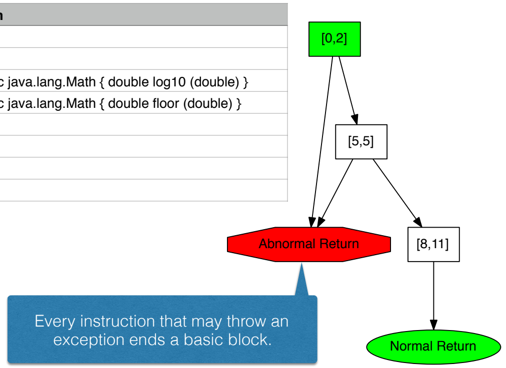


Java Bytecode

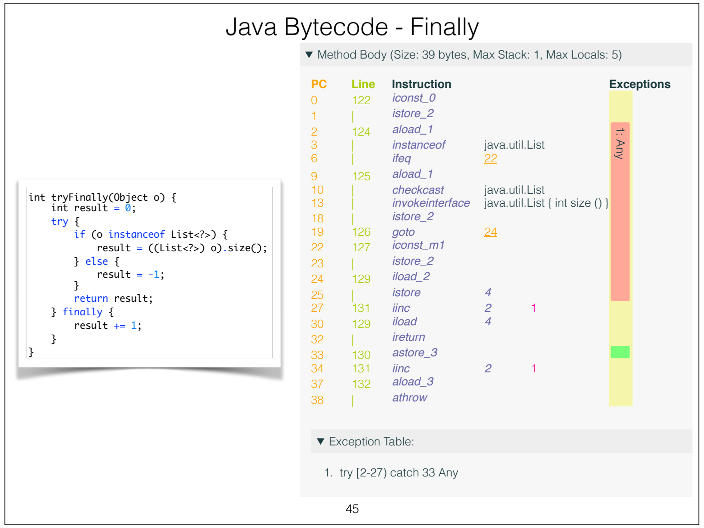
```
static int baseNumberOfDigits(int i) {
    return ((int) Math.floor(Math.log10(i))) + 1;
}
```

CFG

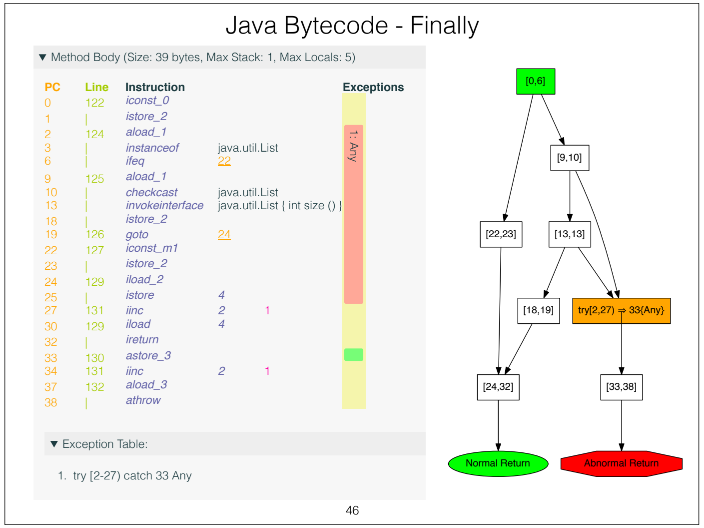
PC	Instruction
0	iload_0
1	i2d
2	invokestatic java.lang.Math { double log10 (double) }
5	invokestatic java.lang.Math { double floor (double) }
8	d2i
9	iconst_1
10	iadd
11	ireturn



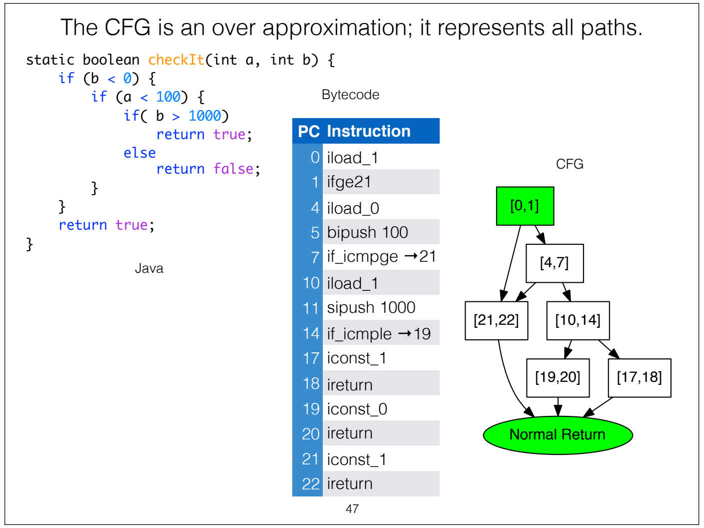
In general, the basic blocks in Java bytecode are rather small; many bytecode instruction can throw an exception.



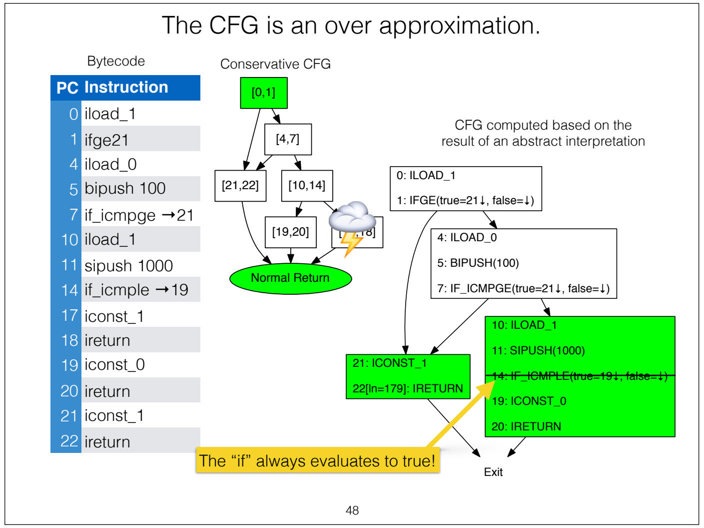
When we reach an exception handler the operand stack always only contains the thrown exception.



The method call (List.size) now goes to the exception handler.



When we perform a decent data-flow analysis, we can determine that the third if condition will always evaluate to false and, hence, the edge to the basic block starting with instruction 17 is dead.



We can now conclude that the instructions 17 and 18 are dead.

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`.
  (Every node dominates itself.)
- In a dominator tree the initial node is the *unique* root node and each node `d` only dominates its descendants.
- Dominator information is useful in identifying loops; by identifying the header and the back edge.

49

(Hence, every node dominates itself.)

Loops - CFG and Dominator Tree

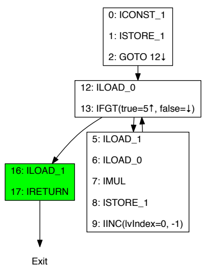
Java

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

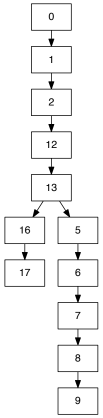
Bytecode

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)



50

Given the dominator tree it is then possible to compute various other helpful data structure: Reverse Dominator Tree, Dominance Frontier, Control Dependence Graph…

Post-Dominator Tree

- The post-dominator tree is the dominator tree computed using the *reverse* control flow graph.
- The reverse control flow graph is the control flow graph where all edges are reversed and — *if necessary* — a new artificial unique root node is added.
- The post-dominator tree is used, e.g., to determine control dependency.

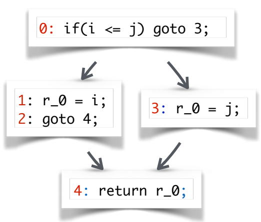
51

Post-Dominator Tree

Java

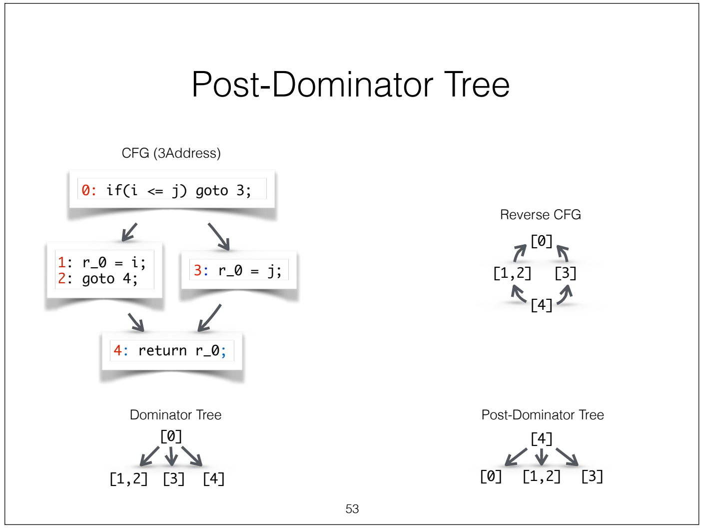
```
static int max(int i, int j) {
    int max;
    if (i > j)
        max = i;
    else
        max = j;
    return max;
}
```

CFG (3Address)



52

(Hence, every node dominates itself.)



(Hence, every node dominates itself.)

Control-Dependence

- An instruction/statement is control dependent on a predicate if the value of the predicate controls the execution of the instruction.

- Let G be a control flow graph; Let X and Y be nodes in G; Y is control dependent on X iff
  - there exists a directed path P from X to Y with any Z in P \ {X,Y} post-dominated by Y
  - X is not post-dominated by Y

54

(Paper: Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9, 3 (July 1987), 319-349. DOI=http://dx.doi.org/10.1145/24039.24041)

First rule in other words: starting with X it is not possible to bypass Y. Second rule in other words: X will not be executed anyway.

Control-Dependence

- An instruction/statement is control dependent on a predicate if the value of the predicate controls the execution of the instruction.

CFG (3Address)

```
0: if(i <= j) goto 3;

1: r_0 = i;
2: goto 4;

3: r_0 = j;

4: return r_0;
```

Post Dominator Tree

```
      [4]
[0]  [1,2]  [3]
```

Is [3] control dependent on [0]? Yes
Is [1,2] control dependent on [0]? Yes
Is [4] control dependent on [0]?
Is [4] control dependent on [1,2]?
Is [4] control dependent on [3]?
} No; [4] post dominates all other nodes

55

Identification of Lazily Initialized Fields

(,e.g. to facilitate the identification of immutable classes)

Control-Dependence

```
class S(final val i: Int, final val j : Int) {

    private[this] var hash: Int = _

    override def hashCode(): Int = {
        if (hash == 0) {
            hash = i*31+j
        }
        hash
    }
}
```

hash is updated if and only if hash still has the default value; after that hash is never updated again (unless the computation's result was the default value).

The update is thread-safe.

56

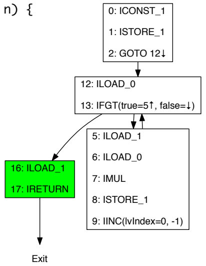
This is a common scenario; e.g., used by java.lang.String.

Def-Use Dependencies

Java

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

CFG



Def-Use Information

PC	Instruction	Used By	Uses
0	iconst_1	{7,17}	
1	istore_1	N/A	
2	goto →12		
5	iload_1		
6	iload_0		
7	imul	{7,17}	1. Operand {-1,9} 2. Operand {0,7}
8	istore_1		
9	iinc (lv=0,val=-1)	{7,9,13}	
12	iload_0		
13	ifgt →5		{-1,9}
16	iload_1		
17	ireturn		{0,7}

Here "-1" is used to indicate a usage of the first parameter.

Loops - Def-Use Dependencies

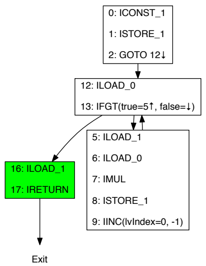
Java

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

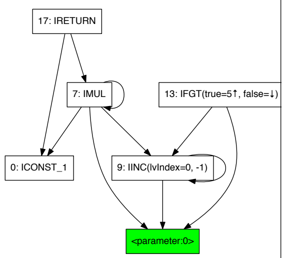
Bytecode

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



Implicit 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.

59

Here, “static” refers to the definition site - it may be in a loop and may be executed multiple times.

SSA is an improvement of def-use chains.

Java Bytecode vs. Three-Address Code vs. SSA

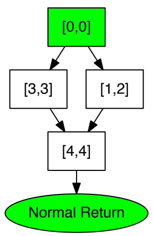
Java

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

Three Address Code

```
0: if(i <= j) goto 3;
1: r_0 = i;
2: goto 4;
3: r_0 = j;
4: return r_0;
```

CFG



SSA

```
0: if(i <= j) goto 3;
```

```
3: t_1 = j;
```

```
1: t_2 = i;
2: goto 4;
```

```
t_3 = Φ(t_1,t_2)
4: return t_3;
```

60

Java Bytecode vs. Three-Address Code vs. SSA

Java

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

```
6: return r_7;
```

61

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 - *basically always* the case that the class hierarchy contains "holes" (is not upwards closed). 😭

62