Code Compiled with javac

```
static int k = 0;
static boolean action(int si,
                      boolean ob,
                      int sm, int pr) {
          if (sm + 2*pr > 10 &&
          !(si <= 5 && ob)) {
             k++; return true;
          } else {
             return false;
          } }
```

```
0:   iload_2
1:   iconst_2
2:   iload_3
3:   imul
4:   iadd
5:   bipush  10
7:   if_icmple      29
10:  iload_0
11:  iconst_5
12:  if_icmpgt      19
15:  iload_1
16:  ifne    29
19:  getstatic      #2; //Field k
22:  iconst_1
23:  iadd
24:  putstatic      #2; //Field k
27:  iconst_1
28:  ireturn
29:  iconst_0
30:  ireturn
```

Compared to our current translation:

if 'sm+2*pr > 10' false, immediately ireturns

if 'si > 5' is true, immediately goes to 'then' part

no intermediate result for if condition - do branches directly

negation sign eliminated and pushed through

only one iconst_0 and one iconst_1

Translate This While Loop using Rules that Explicitly Put Booleans on Stack

```
static void count(int from,
                  int to,
                  int step) {
  int counter = from;
  while (counter < to) {
    counter = counter + step;
  }
}
```

```
nbegin: iload #counter
        iload #to
        if_icmplt ntrue
        iconst_0
        goto nafter
ntrue:  iconst_1
nafter: ifeq nexit
        iload #counter
        iload #step
        iadd
        istore #counter
        goto nbegin
nexit:
```

Towards More Efficient Translation Compiling by Passing Destinations

Macro ‘branch’ instruction

Introduce an imaginary big instruction

branch(c,nTrue,nFalse)

Here

c is a potentially *complex* Java boolean expression,
that is the main reason why branch is not a real instruction

nTrue is label to jump to when c evaluates to true

nFalse is label to jump to when c evaluates to false

no “fall through” – always jumps (symmetrical)

We show how to:

- use **branch** to compile if, while, etc.
- expand **branch** recursively into concrete bytecodes

Using **branch** in Compilation

```
[ if (c) t else e ] =
           branch(c,nTrue,nFalse)
nTrue: [ t ]
           goto nAfter
nFalse:  [ e ]
nAfter:
```

```
[ while (c) s ] =
test:   branch(c,body,exit)
body: [ s ]
        goto test
exit:
```

Decomposing **branch**

branch(**!**c,nThen,nElse) =

  branch(c,nElse,nThen)

branch(c1 **&&** c2,nThen,nElse) =

  branch(c1,nNext,nElse)

nNext: **branch**(c2,nThen,nElse)

branch(c1 **||** c2,nThen,nElse) =

  branch(c1,nThen,nNext)

nNext: **branch**(c2,nThen,nElse)

branch(**true**,nThen,nElse) =

  goto nThen

branch(**false**,nThen,nElse) =

  goto nElse

boolean var b with slot N

branch(b,nThen,nElse) =

  iload_N

  ifeq nElse

  goto nThen

Compiling Relations

branch(e1 **R** e2,nThen,nElse) =

[e1 **]**

[e2 **]**

if_icmpR nThen

goto nElse

R can be <,>,==,!=,<=,>=,...

Putting boolean variable on the stack

Consider storing x = c

where x,c are boolean and c has **&&, ||**

How to put result of **branch** on stack to allow **istore**?

[b = c] = **branch**(c,nThen,nElse)

```
nThen: iconst_1
       goto nAfter
nElse: iconst_0
nAfter: istore #b
```

Compare Two Translations of This While Loop

```
while (counter < to) {
   counter = counter + step;
 }
```

old one:

```
nbegin:  iload #counter
         iload #to
         if_icmplt ntrue
         iconst_0
         goto nafter
ntrue:   iconst_1
nafter:  ifeq nexit
         iload #counter
         iload #step
         iadd
         istore #counter
         goto nbegin
nexit:
```

new one:

```
test:    iload #counter
         iload #to
         if_icmplt body
         goto exit
body:    iload #counter
         iload #step
         iadd
         istore #counter
         goto test
exit:
```

Complex Boolean Expression: Example

Generate code for this:

```
if ((x < y)&& !((y < z) && ok))
  return
else
  y = y + 1
```

```
        branch(x<y,n1,else)
n1:     branch(y<z,n2,then)
n2:     branch(ok,else,then)
then:   return
        goto after
else:   iload #y
        iconst_1
        istore #y
after:
```

This would be much longer with old translation.

Implementing **branch**

- Option 1: emit code using **branch**, then rewrite
- Option 2: **branch** is a just a function in the compiler that expands into instructions

$$\textbf{branch}(c,nTrue,nFalse)$$
$$\downarrow$$

```
def compileBranch(c:Expression,
    nTrue : Label, nFalse : Label) : List[Bytecode] =
{ … }
```

The function takes two destination labels.

More Complex Control Flow

Destination Parameters in Compilation

- To compilation functions **[…]** pass a **label** to which instructions should jump **after** they finish.
  - No fall-through

```
[ x = e ] after =                  // new parameter 'after'
   [ e ]
   istore #x
   goto after                      // at the end jump to it


[ s1 ; s2 ] after =
                  [ s1 ] freshL
   freshL:        [ s2 ] after
```

we could have any junk in here because ([s1] freshL) ends in a jump

Translation of **if**, **while**, **return** with one 'after' parameter

[if (c) t **else** e **]** after =

branch(c,nTrue,nFalse)

nTrue: **[** t **]** after

nFalse: **[** e **]** after

[while (c) s **]** after =

test: **branch**(c,body,after)

body: **[** s **]** test

[return exp **]** after =

[exp **]**

ireturn

Generated Code for Example

[if (x < y) **return; else** y = 2; **]** after =

```
         iload #x
         iload #y
         if_icmp_lt nTrue
         goto nFalse
nTrue:   return
nFalse:  iconst_2
         istore #y
         goto after
```

Note: no **goto** after **return** because

- translation of 'if' does not generate goto as it did before, since it passes it to the translation of the body
- translation of 'return' knows it can ignore the 'after' parameter

break statement

A common way to exit from a loop is to use a 'break' statement e.g.

```
while (true) {
  code1
  if (cond) break
  cond2
}
```

Consider a language that has expressions, assignments, the {…} blocks, 'if' statements, while, and a 'break' statement.
The **'break' statement exits the innermost loop and can appear inside arbitrarily complex blocks and if conditions**.
How would translation scheme for such construct look like?

Two Destination Parameters

```
[ s1 ; s2 ] after brk =
                [ s1 ] freshL brk
  freshL:       [ s2 ] after brk

[ x = e ] after brk =
  [ e ]
  istore #x
  goto after
```

```
[ return exp ] after =
   [ exp ]
   ireturn
[ break ] after brk =
   goto brk
[ while (c) s ] after brk =
test:   branch(c,body,after)
body:  [ s ] test after
```

$\uparrow$

this is where the second parameter gets bound to the exit of the loop

if with two parameters

[if (c) t **else** e **]** after brk =

branch(c,nTrue,nFalse)

nTrue: **[** t **]** after brk

nFalse: **[** e **]** after brk

break and **continue** statements? Three parameters!

[**break**] after brk **cont** =
  goto brk

[**continue**] after brk **cont** =
  goto **cont**

[**while** (c) s] after brk **cont** =
test: **branch**(c,body,after)
body: [s] test after **test**

Some High-Level Instructions for JVM

Method Calls

Invoking methods (arguments pushed onto stack)

invokestatic

invokevirtual

Returning value from methods:

ireturn – take integer from stack and return it

areturn – take reference from stack and return it

return – return from a method returning ‘**void**’

invokestatic

invokestatic
indexbyte1
indexbyte2

..., [arg1, [arg2 ...]] → ...

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 method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The resolved method must not be an instance initialization method (§2.9) or the class or interface initialization method (§2.9). It must be static, and therefore cannot be abstract.

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

The operand stack must contain nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved method.

If the method is synchronized, the monitor associated with the resolved Class object is entered or reentered as if by execution of a monitorenter instruction (§monitorenter) in the current thread.

If the method is not native, the **nargs argument values are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The nargs argument values are consecutively made the values of local variables of the new frame, with arg1 in local variable 0 (or, if arg1 is of type long or double, in local variables 0 and 1) and so on.** Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java Virtual Machine, that is done. The nargs argument values are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

If the native method is synchronized, the monitor associated with the resolved Class object is updated and possibly exited as if by execution of a monitorexit instruction (§monitorexit) in the current thread.

If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

invokevirtual

invokevirtual

indexbyte1

indexbyte2

..., objectref, [arg1, [arg2 ...]] →...

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 method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The resolved method must not be an instance initialization method (§2.9) or the class or interface initialization method (§2.9). Finally, if the resolved method is protected (§4.6), and it is a member of a superclass of the current class, and the method 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.

If the resolved method is not signature polymorphic (§2.9), then the invokevirtual instruction proceeds as follows.

Let C be the class of objectref. The actual method to be invoked is selected by the following lookup procedure:

If C contains a declaration for an instance method m that overrides (§5.4.5) the resolved method, then m is the method to be invoked, and the lookup procedure terminates.

Otherwise, if C has a superclass, this same lookup procedure is performed recursively using the direct superclass of C; the method to be invoked is the result of the recursive invocation of this lookup procedure.

Otherwise, an AbstractMethodError is raised.

The objectref must be followed on the operand stack by nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is synchronized, the monitor associated with objectref is entered or reentered as if by execution of a monitorenter instruction (§monitorenter) in the current thread.

If the method is not native, the **nargs argument values and objectref are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The objectref and the argument values are consecutively made the values of local variables of the new frame, with objectref in local variable 0, arg1 in local variable 1 (or, if arg1 is of type long or double, in local variables 1 and 2), and so on.** Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

...

Translating Method Calls: Example

[x = **objExpr.myMethodName(e1,e2)** **]** =

[objExpr]

[e1]

[e2]

invokevirtual #13

istore #x

constant pool area:

0: "hello, world"

1:

...

13: className.myMethodName/(II)I

...

Rule for Method Call Translation

[objExpr.myMethodName(e1,...,en) **]** =

[objExpr **]**

[e1 **]**

...

[en **]**

invokevirtual #constantPoolAddr

Objects and References

ifnull label - consume top-of-stack reference and jump if it is null

ifnonnull label - consume top-of-stack reference, jump if *not* null

new #className - create fresh object of class pointed to by the offset
#className in the constant pool
(does not invoke any constructors)

getfield #field – consume object reference from stack,
obj.field then dereference the field of that object given
by (field,class) stored in the #field pointer in the constant pool
and put the value of the field on the stack

putfield #field - consume an object reference obj and a value v
obj.field= v from the stack and store v it in the #field of obj

"If the field descriptor type is boolean, byte, char, short, or int, then the value must be an int."

Array Manipulation

a = reference - “address” arrays

i = int arrays (and some other int-like value types)

Selected array manipulation operations:

- **newarray**, **anewarray**, **multianewarray** – allocate an array object and put a reference to it on the stack
- **aaload**, **iaload** – take: a reference to array and index from stack load the value from array and push it onto the stack
- **aastore**, **iastore** – take: a reference to array, an index, a value from stack, store the value into the array index
- **arraylength** – retrieve length of the array

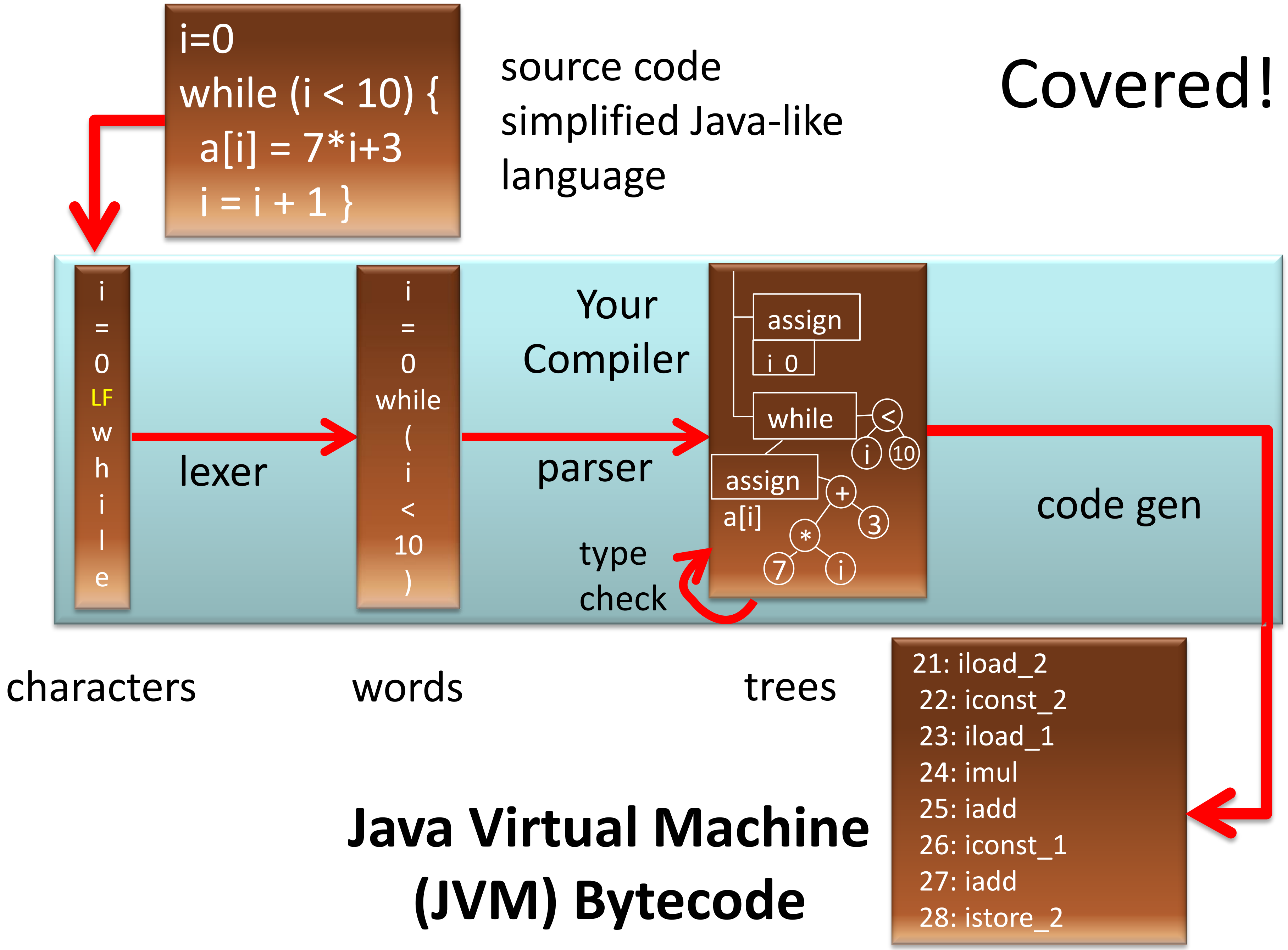
Java arrays store the size of the array and its time, which enables run-time checking of array bounds and object types.

There are Floating Point Operations…

- fadd
- faload (for floating point arrays)
- fastore (for floating point arrays)
- fcmp<op>
- fconst_<f>
- fdiv
- fload
- fload_<n>
- fmul
- fneg
- frem
- freturn
- fstore
- fstore_<n>
- fsub

When needed,
READ THE JVM Spec ☺

Covered!



```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

```
21: iload_2
 22: iconst_2
 23: iload_1
 24: imul
 25: iadd
 26: iconst_1
 27: iadd
 28: istore_2
```

Java Virtual Machine (JVM) Bytecode