Compiling Java:

Can we have the elegance of Java and the speed of C?

Laurie J. Hendren

... and the OPC Java gang ...

Lars Ræder Clausen
Ulrik Pagh Schultz
Kim Burgaard
Niels Damgaard
Ulrik Skyt
Jan Møller
Erik Martino
Thorbjørn Bonbæk

- NO . . . not really.
- Why not?
- Can we make Java “fast enough”?

Outline

- Background
  - Java, the language
  - Java bytecode and virtual machine
  - interpreter vs. JIT compilation
- Optimizing Java
  - where/what to optimize?
  - three approaches to optimizing Java bytecode
  - where to go from here

According to Sun, Java is:

- simple,
- object-oriented,
- distributed,
- interpreted,
- robust,
- secure,
- architecture neutral,
- portable,
- high-performance,
- multithreaded, and
- dynamic.

A small "functional" example

$$\boxed{\begin{array}{l}\textbf{let}\\ \quad f = \lambda x.x + 1\\ \quad g = \lambda y.y + 2\\ \textbf{in}\\ \quad g(f(13))\\ \textbf{end}\end{array}}$$

```
class likeLambda {

  private static int f (int x)
    { return(x + 1); }

  private static int g (int y)
    { return(y + 2); }

  public static void main(String[] args)
    { System.out.println(g(f(13))); }

}
```

A simple, C-like example

```
class LikeC {

  private final static int SIZE = 100;

  static void initArray(int[] x)
    { for (int i=0; i < x.length;  i++)
          x[i] = 0;
    }

  public static void main(String[] args)
    { int a[] = new int[SIZE];

      initArray(a);
      System.out.println("Array Initialized");
    }
}
```

The equivalent C code

```
#define SIZE 100

void initArray(int x[])
{ int i;
  for (i=0; i < SIZE;  i++)
    x[i] = 0;
}

void main(int argc, char *argv[])
{ int a[SIZE];

  initArray(a);
  printf("Array Initialized\n");
}
```

Object-oriented Lists

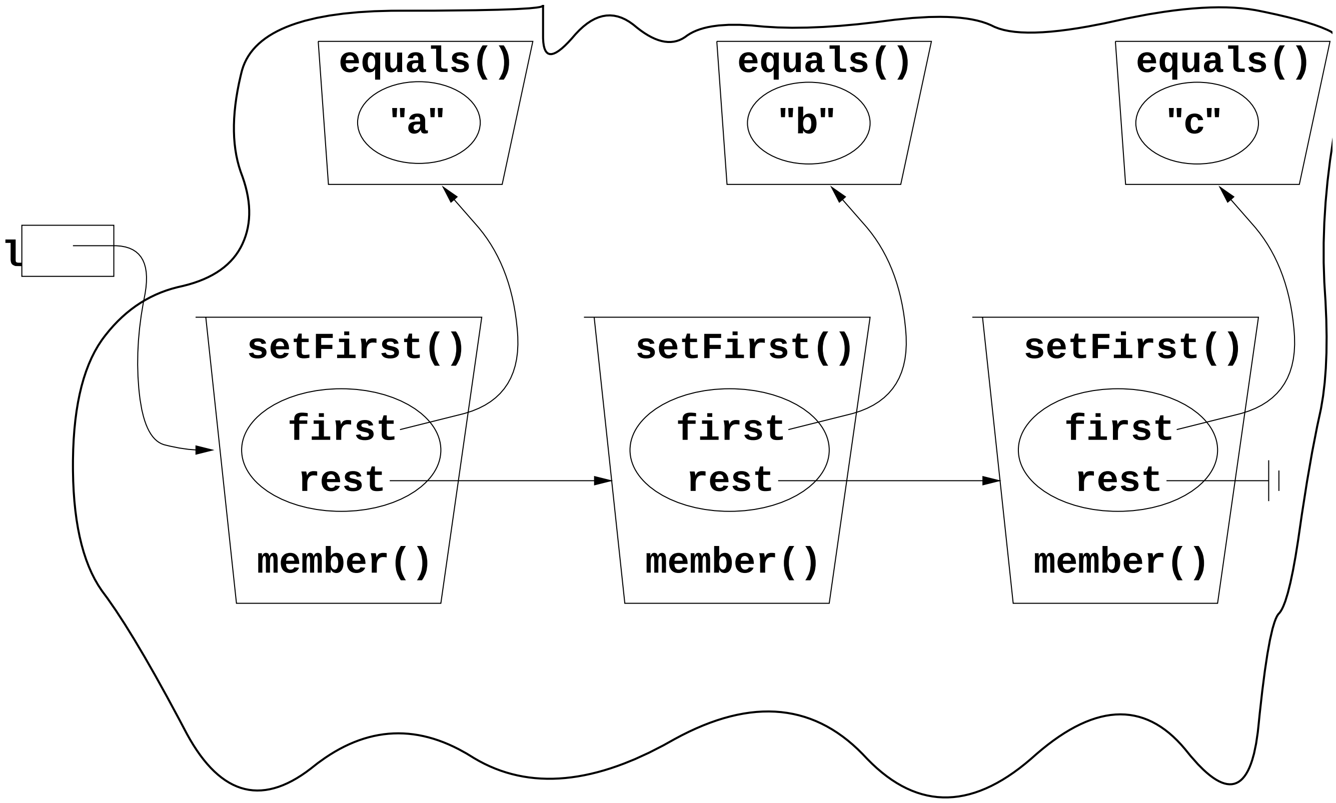
```
public class Cons {
  /* Fields */
  private Object first;
  private Cons rest;

  /* Constructor */
  public Cons(Object first, Cons rest)
    { this.first = first;
      this.rest = rest;
    }

  /* Methods */
  public void setFirst(Object newfirst)
    { first = newfirst; }

  public boolean member(Object item)
    { if (first.equals(item))
        return(true);
      else if (rest == null)
        return(false);
      else
        return(rest.member(item));
    }
}
```

Example List



Using Lists

```
class UseCons {

  public static void main(String[] args)
    { Cons l =
       new Cons("a",
                new Cons("b",
                          new Cons("c",
                                    null)));

      System.out.println("c member? " +
                              l.member("c"));
      System.out.println("z member? " +
                              l.member("z"));
    }
}
```

```
>javac UseCons.java
>java UseCons
c member? true
z member? false
```

Statements and Expressions in Java

- Statements
  - empty statement, expression statement
  - blocks `{ ... }`
  - if, switch, while, do-while, for
  - break, continue, return
  - throw, try-catch-finally
  - synchronized
  - no goto, (break, continue use labels)

- Expressions
  - pretty much like C
  - no dereference `*p`, `p->f`
  - no `&`

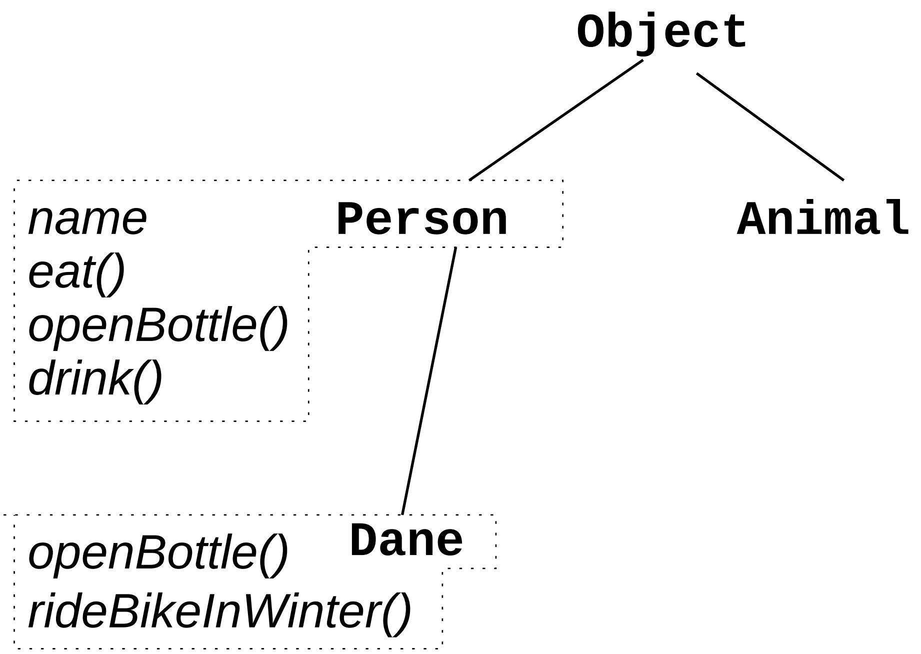
Types in Java

- Primitive Types
  - boolean: `true` and `false`
  - char: `\u0000` to `\uFFFF` (Unicode)
  - byte: $-128 \ldots 127$
  - short: $-2^{15} \ldots 2^{15} - 1$
  - int: $-2^{31} \ldots 2^{31} - 1$
  - long: $-2^{63} \ldots 2^{63} - 1$
  - float and double: per IEEE 754
- Class types
- Array types
- Interface types

Type Checking

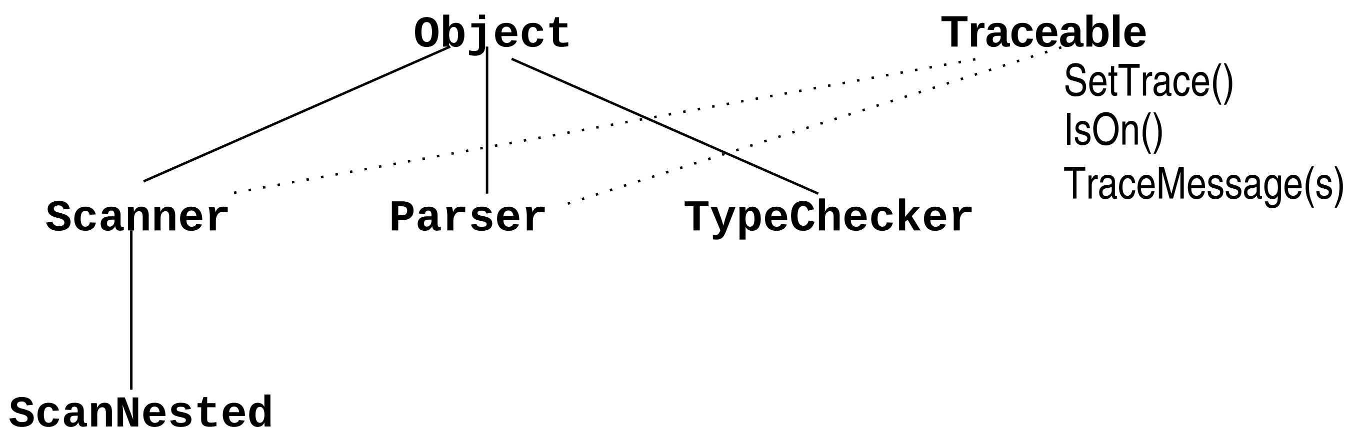
- Java is strongly-typed.
- Java uses the name of a class as its type.
- Given a type of class `C`, any instance of class `C` or a subclass of `C` is a permitted value.
- If `S` is a subclass of `T`, then `S[]` is a subclass of `T[]`.
- Checked "down-casting" and `instance_of`.
- Some type-checking must be done at run-time.

Single Inheritance in Java




```
void HaveMeal(Person p)
{  p.eat();
   if (p.openBottle()) p.drink();
}
```

Interfaces in Java

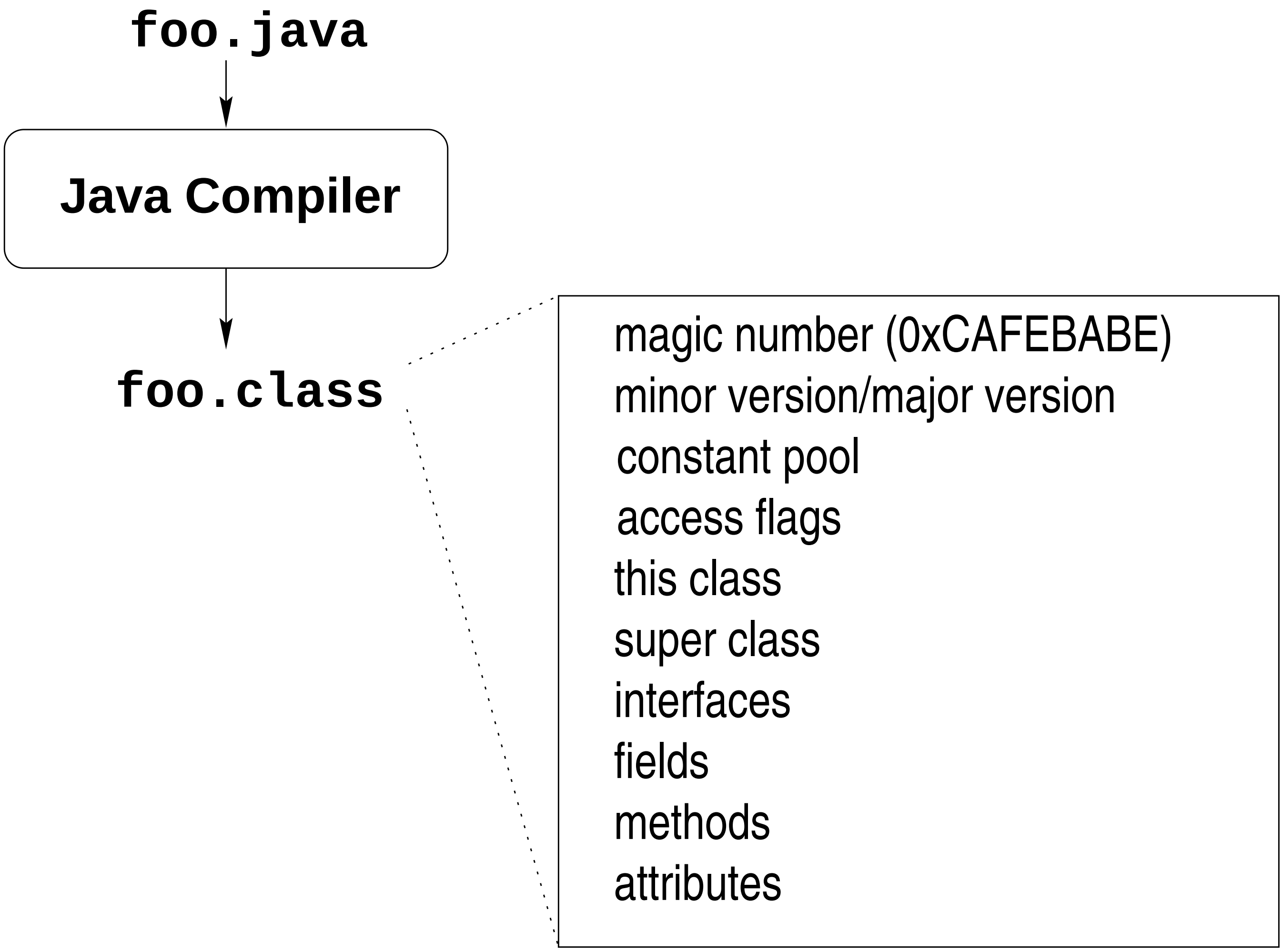



```
void DoTrace(Traceable t, String s)
  { if (t.IsOn())
        System.out.println(t.TraceMessage(s));
   }
```

Java has many libraries (packages)

`java.lang:`	basic classes, threads, class loaders ...
`java.util:`	vectors, stacks, hash tables ...
`java.io:`	byte stream I/O, formatted I/O, random access files ...
`java.net:`	networking
`java.applet:`	applet class
`java.awt:`	support for widows and images (**a**bstract **w**indow **t**oolkit)

Compiling Java to Class Files



Compiling and Decompiling Class Files

Avg.java

```
class Avg {

float average(float x, float y, float z)
  { return( (x+y+z)/3.0f ); }

}
```

javac Avg.java

Avg.class

```
...
Method float average(float,float,float)
   0 fload_1
   1 fload_2
   2 fadd
   3 fload_3
   4 fadd
   5 ldc #1 <Real 3.000000>
   7 fdiv
   8 freturn
```

mocha.Decompile Avg.class

Avg.mocha

```
class Avg {

float average(float f1, float f2, float f3)
  { return (f1 + f2 + f3) / 3; }

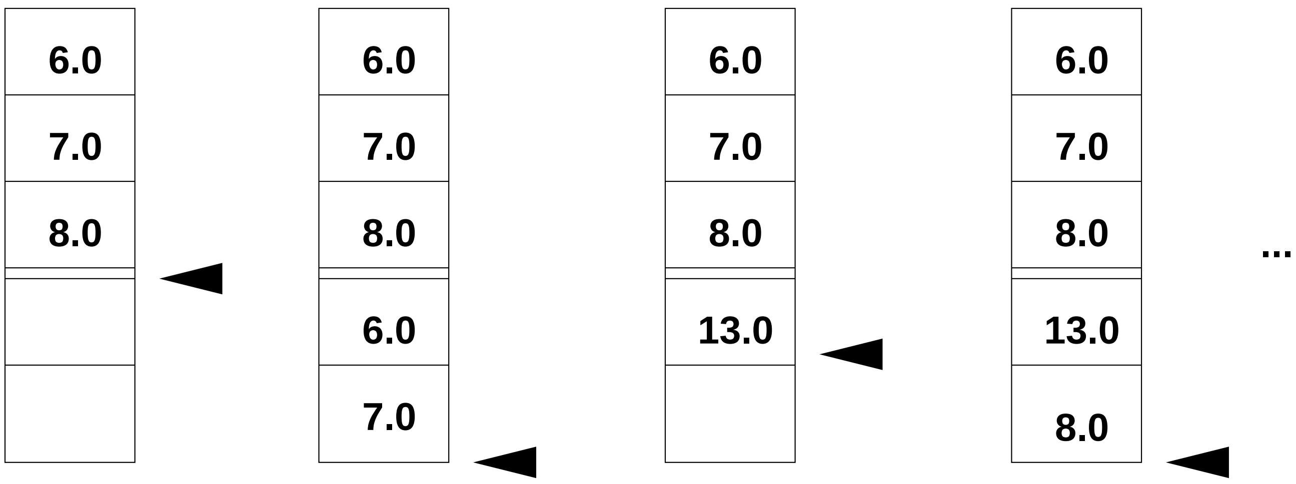
}
```

Java Virtual Machine

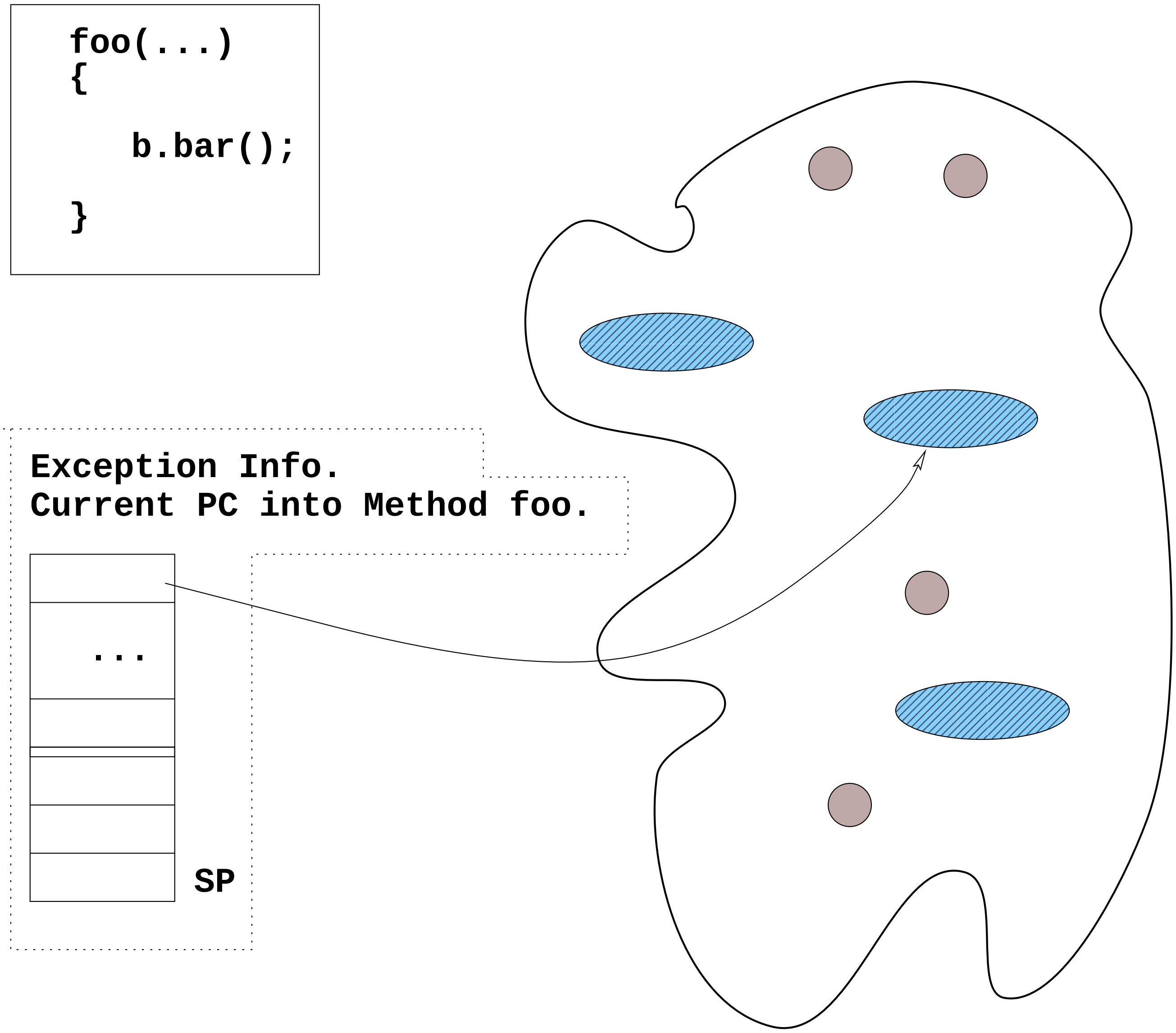
Avg.class

```
...
Method float average(float,float,float)
   0 fload_1
   1 fload_2
   2 fadd
   3 fload_3
   4 fadd
   5 ldc #1 <Real 3.000000>
   7 fdiv
   8 freturn
```

Suppose we invoke average with: average(6.0,7.0,8.0)



General Execution Model



Bytecode Verification

- Bytecode cannot be trusted to be well-formed and well-behaved.

- Before executing any bytecode that is received over the network, it should be verified.

- Verification is performed partly at class loading time, and partly at run-time.

- At load time, dataflow analysis is used to approximate the number and type of values in locals and on the stack.

Interesting Properties of Verified Bytecode

- Each instruction must be executed with the correct number and types of arguments on the stack, and in locals (on all execution paths).

- At any program point p, the stack is the same size along all execution paths.

- No local variable can be accessed before it has been assigned a value.

Interpreting Java

- When a method is invoked, a `classloader` must find the correct class, and check that it contains an appropriate method. This may require traversing up the inheritance hierarchy. If the class is not found, then an exception is raised.
- If the method has not yet been loaded, then it may be verified (remote classes). If it fails verification, an exception is raised.
- After loading and verification, either:
  - parameters are placed in the appropriate locals, and interpretation of the method body begins; or
  - the bytecode for the method is translated to native code (only for the first invocation), the appropriate parameters are loaded and control is given to the native code.

Interpreting bytecode

```
pc = 0;
while(true)
  {  npc = pc + instr_length(code[pc]);
     switch (opcode(code[pc]))
       {  ILOAD_1: push(local[1]);
                   break;
          ILOAD:   push(local[code[pc+1]]);
                   break;
          ISTORE:  t = pop();
                   local[code[pc+1]] = t;
                   break;
          IADD:    t1 = pop();  t2 = pop();
                   push(t1 + t2);
                   break;
          IFEQ:    t = pop();
                   if (t == 0) npc = code[pc+1];
                   break;
          ...
       }
     pc = npc;
  }
```

Code generated using kaffe (JIT-sparc)

```
Translating Avg.average(FFF)F (normal) 0x0
      * save sp,-128,sp
  SPILL st i1,[fp+72]          ;  0 fload_0
 RELOAD ldf f0,[fp+72]         ;
      0 fmovs f0,f2            ;
  SPILL st i2,[fp+76]          ;  1 fload_1
 RELOAD ldf f4,[fp+76]         ;
      1 fmovs f4,f6            ;
      2 fadds f2,f6,f2         ;  2 fadd
  SPILL st i3,[fp+80]          ;  3 fload_2
 RELOAD ldf f8,[fp+80]         ;
      3 fmovs f8,f6            ;
      4 fadds f2,f6,f2         ;  4 fadd
  SPILL stf f6,[fp+-36]        ;  5 ldc #1 <3.0>
      5 sethi 0x40400000,l0    ;
  SPILL st l0,[fp+-36]         ;  7 fdiv
 RELOAD ldf f10,[fp+-36]       ;
      7 fdivs f2,f10,f2        ;
      8 fmovs f2,f0            ;  8 freturn
      8 ret                    ;
      8 restore                ;
```

Times for Sun Interpreter vs. Kaffe-JIT
(gorm: SPARCSTATION, 80 Mhz)

	java (ms)	kaffe (ms)	java/ kaffe
local variable assgn.	0.48	0.037	13
instance variable assign.	1.20	0.049	24
array element assign.	1.20	0.18	6.7
int increment	0.31	0.036	8.6
object creation	9.4	32.0	.29
method call	2.3	0.68	3.4
synchronous call	17	3	5.4
math function (`abs`)	4.6	2.9	1.6
equiv. code (`abs`)	2.3	2.2	1.0

Times for Sun Interpreter vs. Kaffe-JIT
(gorm: SPARCSTATION, 80 Mhz)

	java		kaffe	
	(ms)	LVA	(ms)	LVA
local variable assgn.	0.48	1	0.037	1
instance var. assign.	1.20	2.5	0.049	1.3
array element assign.	1.20	2.5	0.18	4.9
int increment	0.31	.65	0.036	1
object creation	9.4	20	32.0	865
method call	2.3	4.8	0.68	18
synchronous call	17	35	3	81
math func. (`abs`)	4.6	9.6	2.9	78
equiv. code (`abs`)	2.3	4.8	2.2	56

Java vs. Kaffe on real programs

- Running `javac`, a java program, to compile the jBYTEMARKs benchmark:

	java (sec)	kaffe (sec)
computation	16.5	3.6
garbage collection	3.5	12.3
JIT compilation	0.0	0.6
TOTAL	20.0	16.5

- All aspects of the compilation and execution platform must be optimized.

Java/Kaffe vs. C on computation-intensive programs

- Running a prime number sieve program using C and java.

	iterations/10 sec
gcc	1042
gcc -O4	2903
java	76
kaffe	783

- Naive JIT can't compete against optimizing C compilers on computation-intensive programs.

What's so great about optimizing C compilers?

How about optimizing this loop?

```
for (i=0; i < SIZE;  i++)
   x[i] = 0;
```

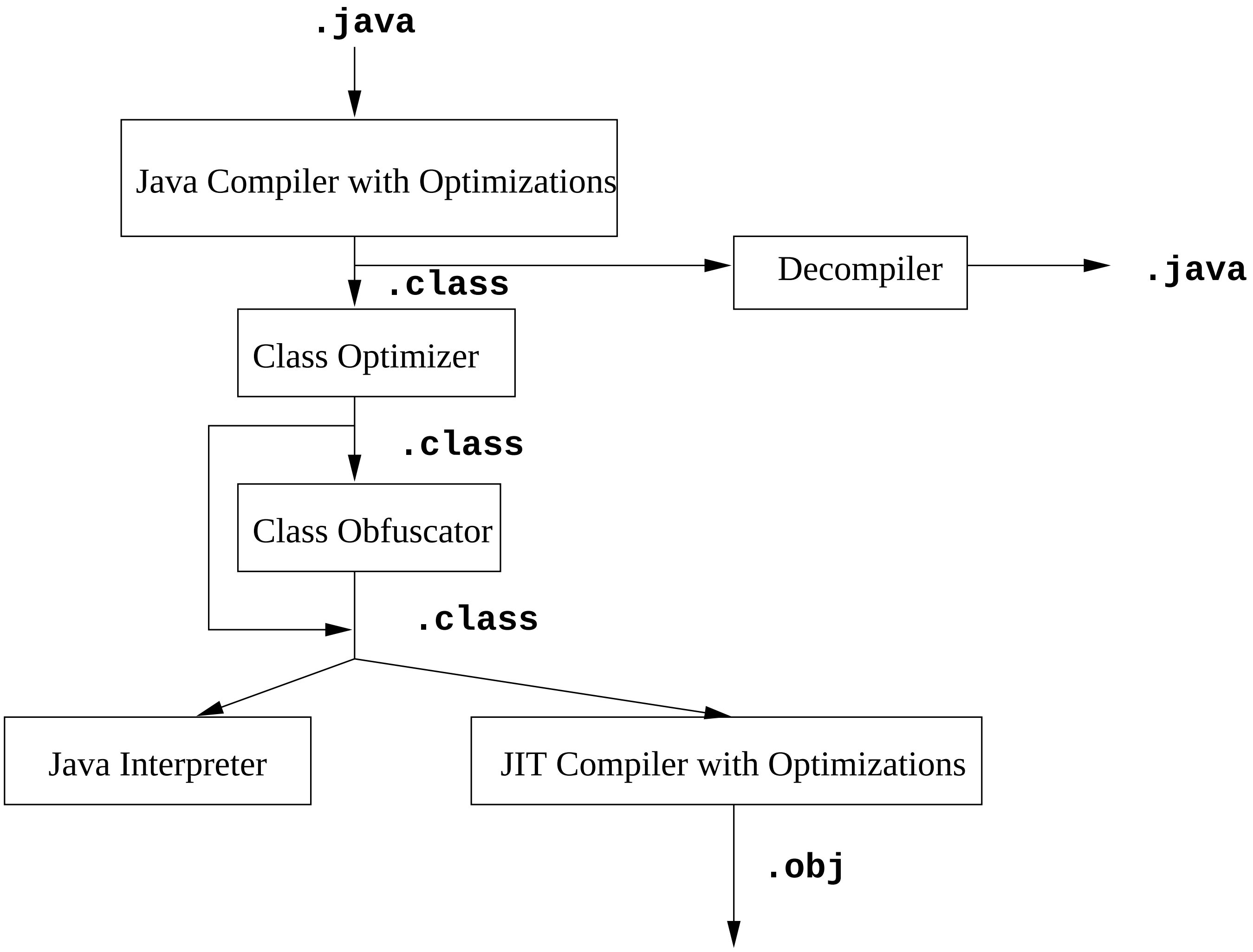
Unoptimized

```
.LL2:
  ld [%fp-20],%o0
  cmp %o0,99
  ble .LL5
  nop
  b .LL3
  nop
.LL5:
  ld [%fp-20],%o0
  mov %o0,%o1
  sll %o1,2,%o0
  ld [%fp+68],%o1
  add %o0,%o1,%o0
  st %g0,[%o0]
.LL4:
  ld [%fp-20],%o1
  add %o1,1,%o0
  mov %o0,%o1
  st %o1,[%fp-20]
  b .LL2
  nop
.LL3
```

Optimized with -O4

```
  mov 396,%g2
.LL13:
  addcc %g2,-4,%g2
  bpos,a .LL13
  st %g0,[%g2+%o0]
```

Optimizing Java - where to do it?



Optimizing bytecode (.class $\rightarrow$ .class)

(Experience with OPC projects.)

- Peephole Transformations
- Local Variable / Stack Variable Optimization
- Object/Method Optimization (purity analysis)

Example Peephole Optimizations

(*Jan Møller*, *Erik Martino* , and *Thorbjørn Bonbæk*)

- ```
  /* x = exp; return(x); */
  istore x
  iload  x        ======>   ireturn
  ireturn
  ```

- ```
  /* x = x + y; */
  iload  x
  bipush y        ======>   iinc x y
  iadd
  istore x
  ```

- ```
  /* x + y + x */
  iload_1                   iload_1
  iload_2        =======>   dup
  iadd                      iload_2
  iload_1                   iadd
  iadd                      iadd
  ```
  ```

Analysis and Transformation of Locals/Stack Variables

(*Kim Burgaard*, *Niels Damgaard* and *Ulrik Skyt*)

- Set up a dataflow analysis for reaching definitions and constant estimation by approximating the values of all locals and stack variables.

- Use the analysis to do constant propagation and dead-code removal.

Analysis and transformation of 3-addr code

```
f()
{ int x,y,z;

  x = 1;          {(x,1)}
  y = 2;          {(x,1),(y,2)}
  z = x + y;      {(x,1),(y,2),(z,3)}
  return(z);
}
```

```
f()
{ int x,y,z;

  x = 1;
  y = 2;
  z = 3;
  return(3);
}
```

```
f()
{ int x,y,z;




  return(3);
}
```

Analysis and transformation of stack code

```
iconst_1   {(s0,1)}
istore_0   {(l0,1)}
iconst_2   {(l0,1),(s0,2)}
istore_1   {(l0,l),(l1,2)}
iload_0    {(l0,1),(l1,2),(s0,1)}
iload_1    {(l0,1),(l1,2),(s0,1),(s1,2)}
iadd       {(l0,1),(l1,2),(s0,3)}
istore_2   {(l0,1),(l1,2),(l3,3)}
iload_2    {(l0,1),(l1,2),(l3,3),(s0,3)}
ireturn
```

```
iconst_1
istore_0
iconst_2
istore_1
iconst_1
iconst_2
pop
pop
iconst_3
istore_2
iconst_3
ireturn
```

```
iconst_3
ireturn
```

Analysis and transformations with side-effects

(*Lars Ræder Clausen* and *Ulrik Pagh Schultz*)

- Need to define and approximate the purity of each instruction, method, and loop.

- A computation C is:

  Pure: if C neither reads, nor writes a heap location.

  Semipure: if C reads, but does not write a heap location.

  Impure: if C writes a heap location.

Example of Purity

```
class Example {

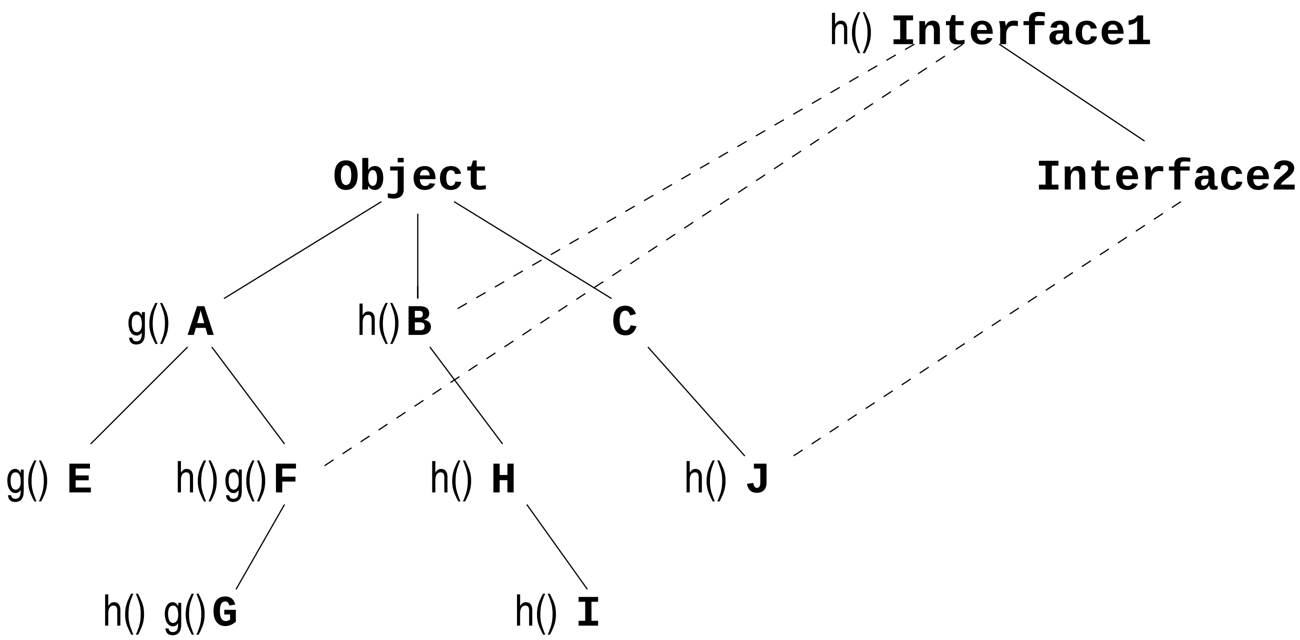
  private int g;

  int inc(int x)  /* pure */
   { return(x+1); }

  int inc_by_g(int x) /* semipure */
    { return(x+g); }

  void inc_g ( )  /* impure */
    { g++; };
}
```

Purity and the Inheritance Hierarchy



Some Experimental Results

Purity of Methods

	# . meth.	Simple Purity % pure	Simple Purity % semi	Advanced Purity % pure	Advanced Purity % semi
cream	334	4.2	6.9	5.7	9.0
java/util	174	1.7	9.2	2.9	9.2
sun/tools/java	477	2.1	14.3	2.3	23.3
jBYTEmarks	182	4.9	12.1	5.5	15.9

% of Code that is Dead

	# Instr.	Simple Purity	Advanced Purity
cream	12777	.40	.40
java/util	3894	.85	1.1
sun/tools/java	12928	.34	4.1
jBYTEmarks	14941	1.2	1.2

Lessons Learned

- Making a high-performance Java system requires a lot of engineering to optimize all facets of the system including: object allocation/garbage collection, speed of the JIT compiler, quality of of the JIT code produced, ...

- It will be hard to really compete with C on computation-intensive computations.

More Lessons Learned

Analysis and transformation of the bytecode is possible, however:

- Optimizing only the local/stack variables will probably lead to limited speedups.

- Optimizing method invocations/field references is difficult due to dynamic linking in Java. Good static results will probably only be possible by optimizing "complete applications/packages". (Precompute dataflow information, class hierarchy for packages).

- To match the dynamic nature of Java, some energy should be put into dynamic compilation and conditionally-optimized code.

What's next?

- Side-effect analysis, starting with purity analysis.
  - conservative methods that do not need the class hierarchy
  - full-blown methods using precomputed summaries of packages
- Array range checking optimizations, insert info into bytecode attributes.