

Smalltalk Implementation

Prof. Harry Porter
Portland State University

The Image

The object heap

The Virtual Machine

The underlying **system** (e.g., Mac OS X)

The ST language **interpreter**

The **object-memory manager**

Outline:

Describe a simple implementation

Representation of objects in memory

The “bytecode” representation of ST code

The bytecode interpreter

Memory management / garbage collection algorithms

Optimization Techniques

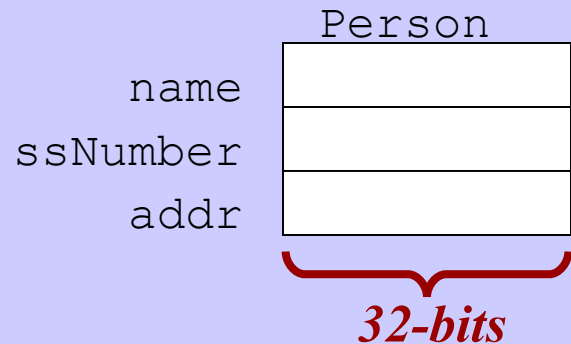
References

- Smalltalk-80: The Language and its Implementation, by Goldberg and Robson (Part IV), Addison-Wesley, 1983.
- Smalltalk-80: The Language, by Goldberg and Robson (Chapter 21), Addison-Wesley, 1989.
- Smalltalk-80: Bits of History, Words of Advice, ed. Glen Krasner, Addison-Wesley, 1983.
- Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, by David Ungar, ACM Software Engineering Notes/SIGPLAN Notices: Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, 1984.
- Efficient Implementation of the ST-80 System, by Peter L. Deutsch and Allan M. Schiffman, POPL-84, Salt Lake City, UT, 1984.
- Architecture of SOAR: Smalltalk on a RISC, by Ungar, Blau, Foley, Samples, Patterson, 11th Annual Symposium on Computer Architecture, Ann Arbor, MI, 1984.
- The Design and Evaluation of a High Performance Smalltalk System, by David M. Ungar, MIT Press, ACM Distinguished Dissertation (1986), 1987.

Representing Objects

Object = Block of memory (i.e., “struct”, “record”)

Field = Offset into record (“instance variable”)



Representing Objects

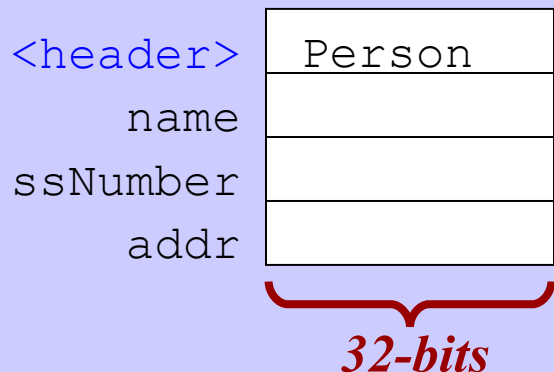
Object = Block of memory (i.e., “struct”, “record”)

Field = Offset into record (“instance variable”)

Header

A “hidden” field, included in every object.

Tells the class of the object (and other stuff).



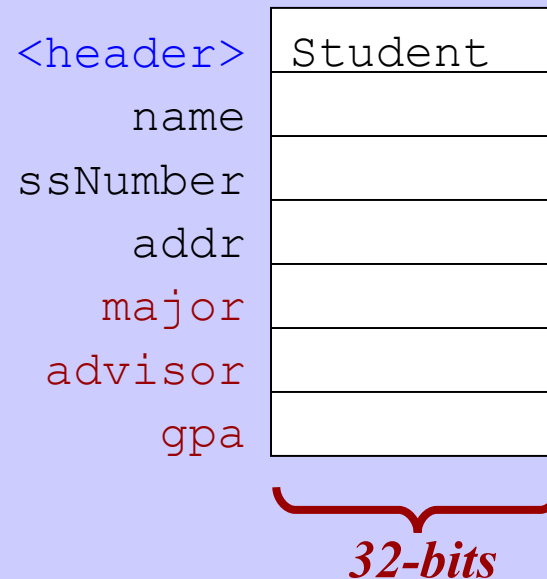
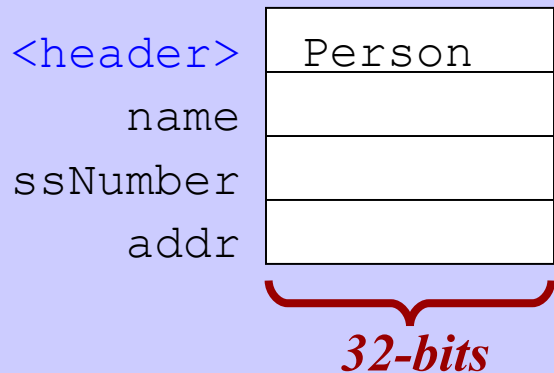
Representing Objects

Subclassing:

Existing fields in the same locations

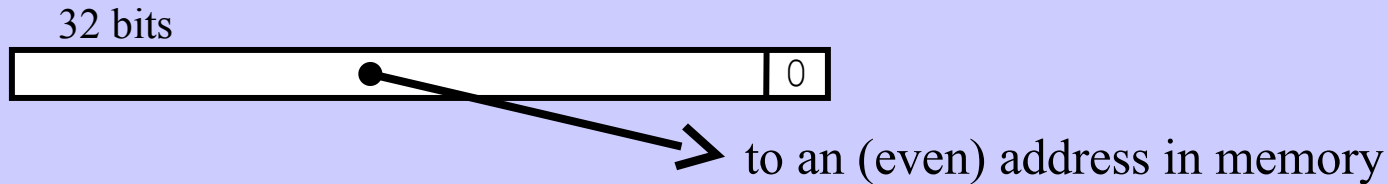
New fields added to end of record

Example: Student is a subclass of Person

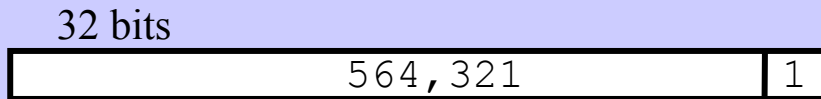


Tagged Values

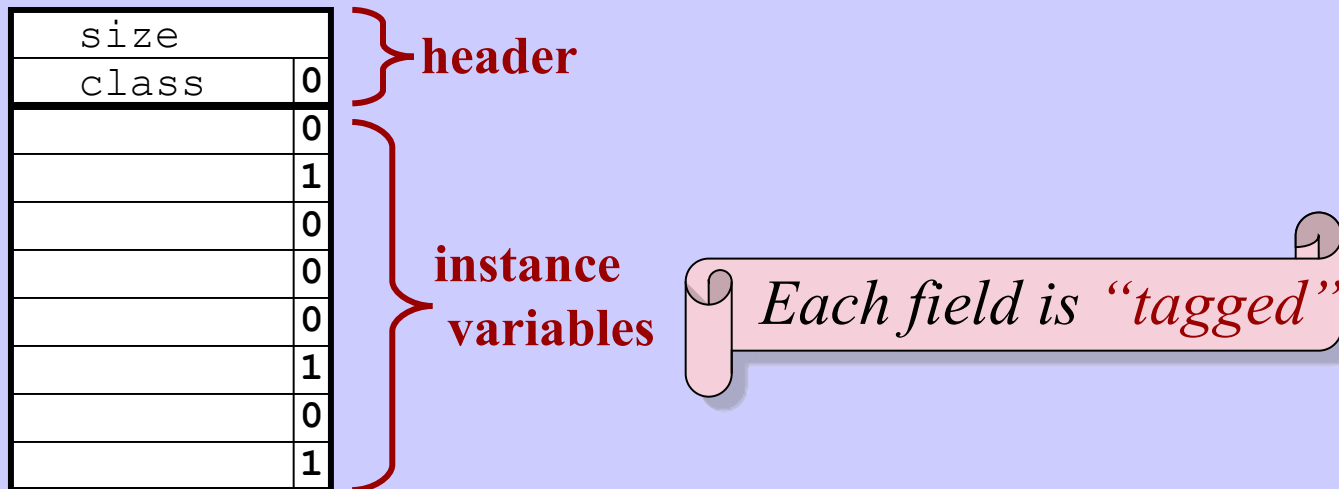
Object-Oriented Pointers (OOPs)



SmallIntegers (-1,073,741,824 .. 1,073,741,823)



Objects



Other formats for objects (containing “raw” bits)

ByteArray

size			
class			0
			0
			1
			0
01	23	A0	4F
23	CC	D6	FF
45	4A	F0	80
56	86	7F	7F
78	00	00	00

} *Header*

} *Normal Instance Variables*

WordArray

size	
class	0
	0
	1
	0
0123A04F	
23CCD6FF	
454AF080	
56867F7F	
78303132	

} *Header*

} *Normal Instance Variables*

These fields are not “tagged”

Bytecodes

The instructions of the virtual machine (VM) interpreter
The VM executes one bytecode instruction after another.

Note: “*execute*” = “*interpret*” = “*emulate*”

A real machine executes instructions.

The VM executes bytecodes.

Like machine language instructions

- Comparable level of detail
- 1 to 4 bytes long
- Tight encoding into the available bits (CISC architecture)

(Java used ST’s approach VM, bytecodes, etc.)

The Compiler

Translates methods (i.e., Strings) into instances of a class called

CompiledMethod

Contains a sequence of bytes (the “bytecodes” to execute)

The Compiler

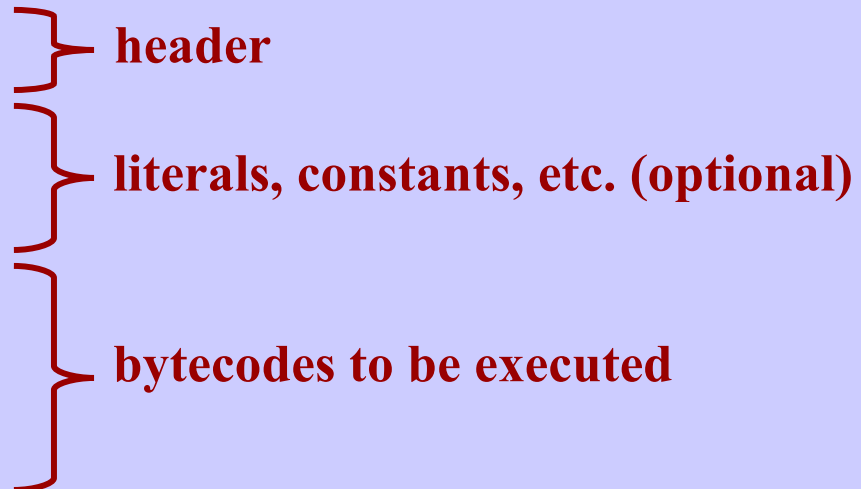
Translates methods (i.e., Strings) into instances of a class called

CompiledMethod

Contains a sequence of bytes (the “bytecodes” to execute)

CompiledMethod is subclass of *ByteArray*.

size			
class			0
			0
			1
			0
01	23	A0	4F
23	CC	D6	FF
45	4A	F0	80
56	86	7F	7F
78	00	00	00



Class Symbol

Symbols are used for method selectors.

```
'hello'           'at:put:'  
#hello #at:put:
```

Like the class *String*.

Symbol is a subclass of *String*.

Consider a string `'hello'` ... there may be many *Strings* with these chars.

Consider the symbol `#hello` ... there is only one *Symbol* with these chars.

There is a system-wide collection of all *Symbol* objects.

All *Symbol* objects are kept in this “symbol table”.

String

`'hello'` and `'hello'` may be two different objects.

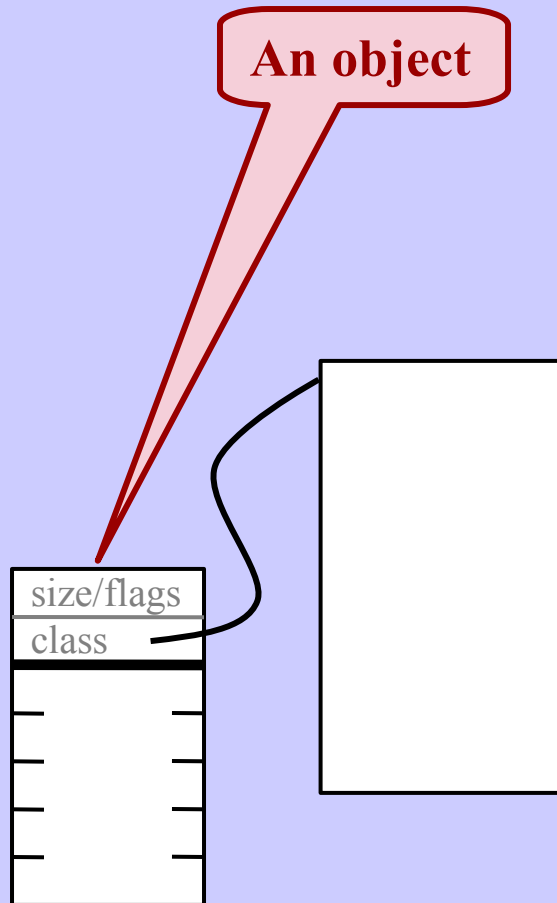
`=` will compare characters, one-by-one.

You should always use `=` to test *Strings*.

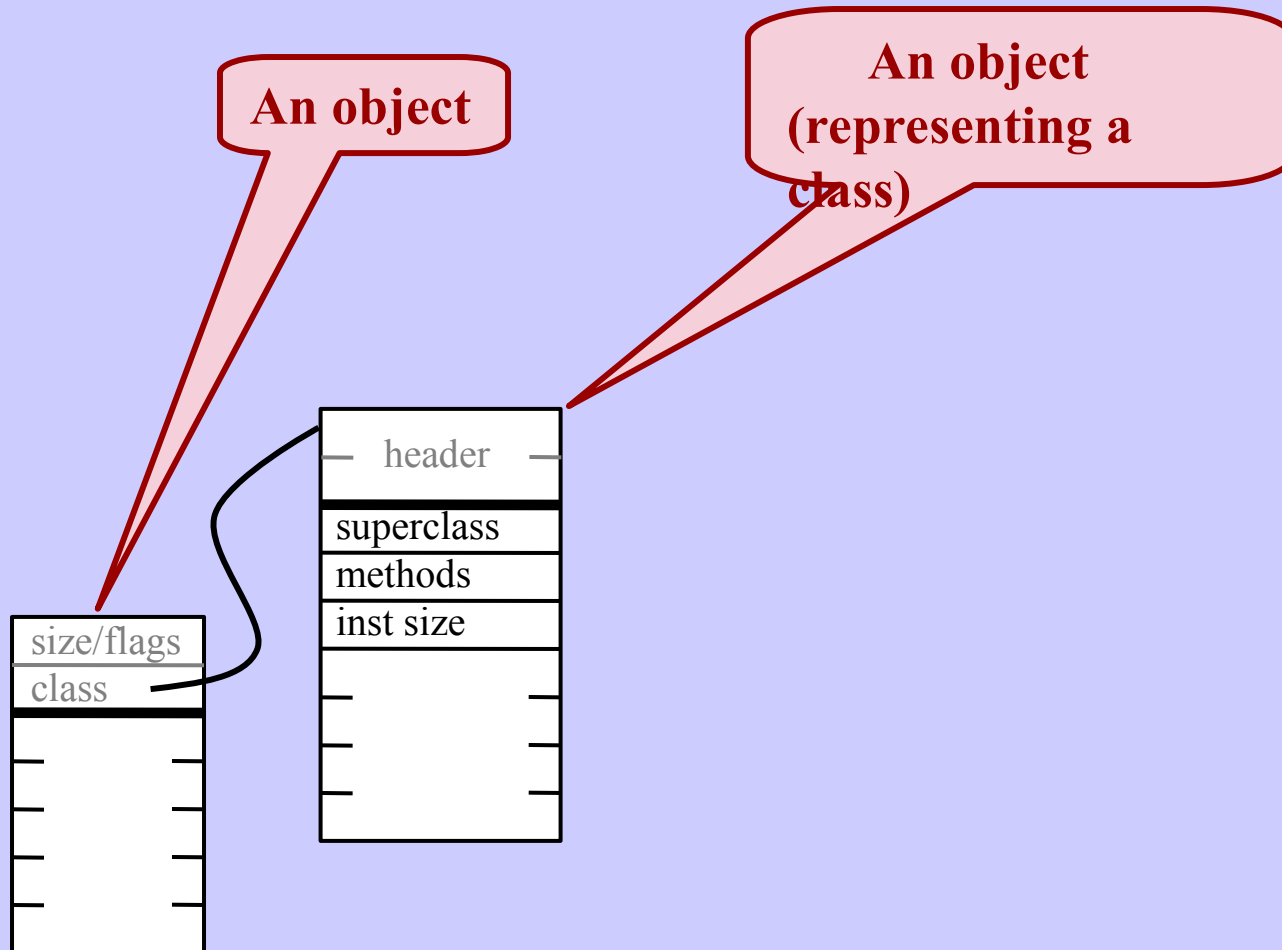
Symbol

You can always rely on `==` , which is fast!

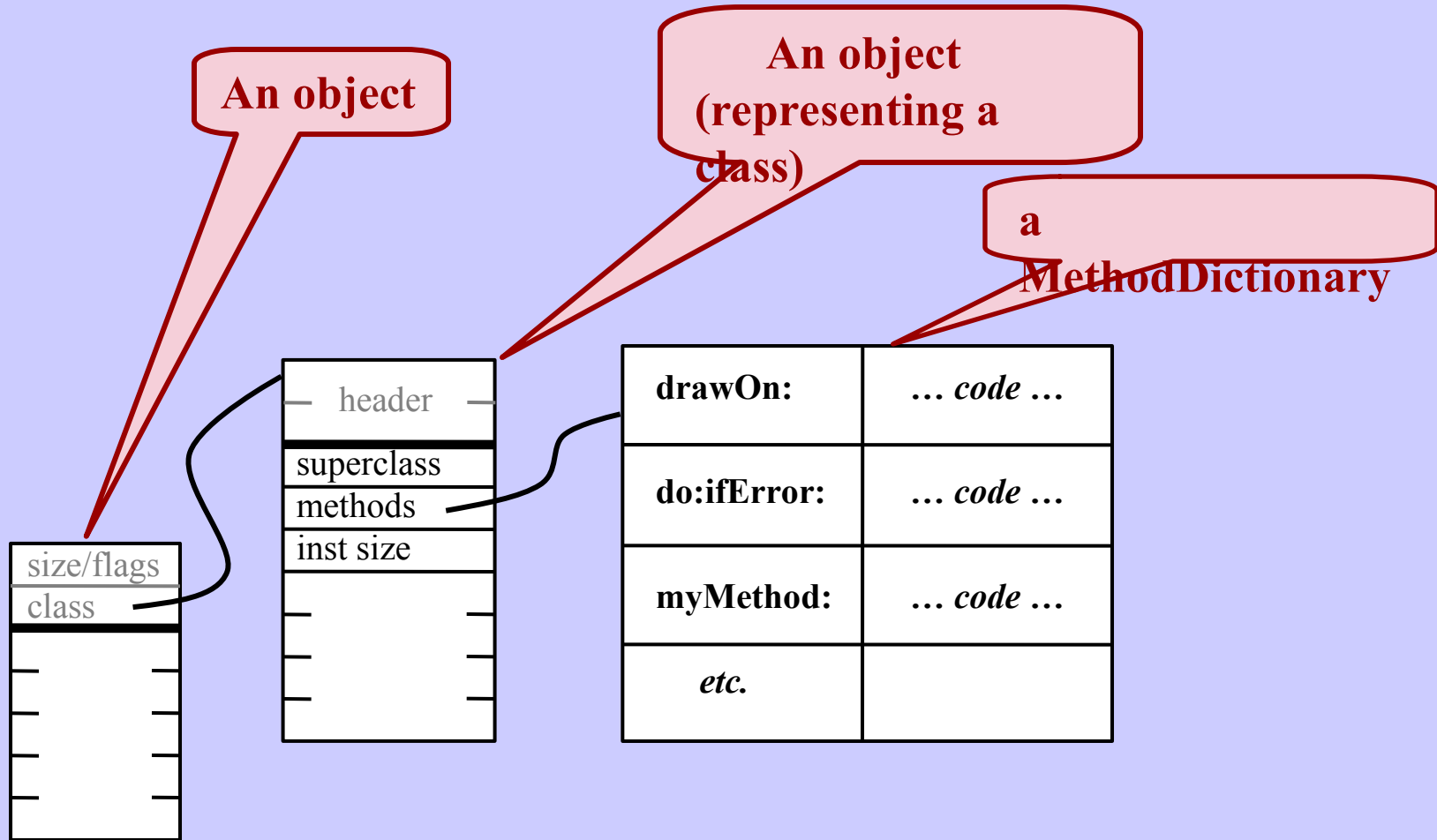
Representing Classes



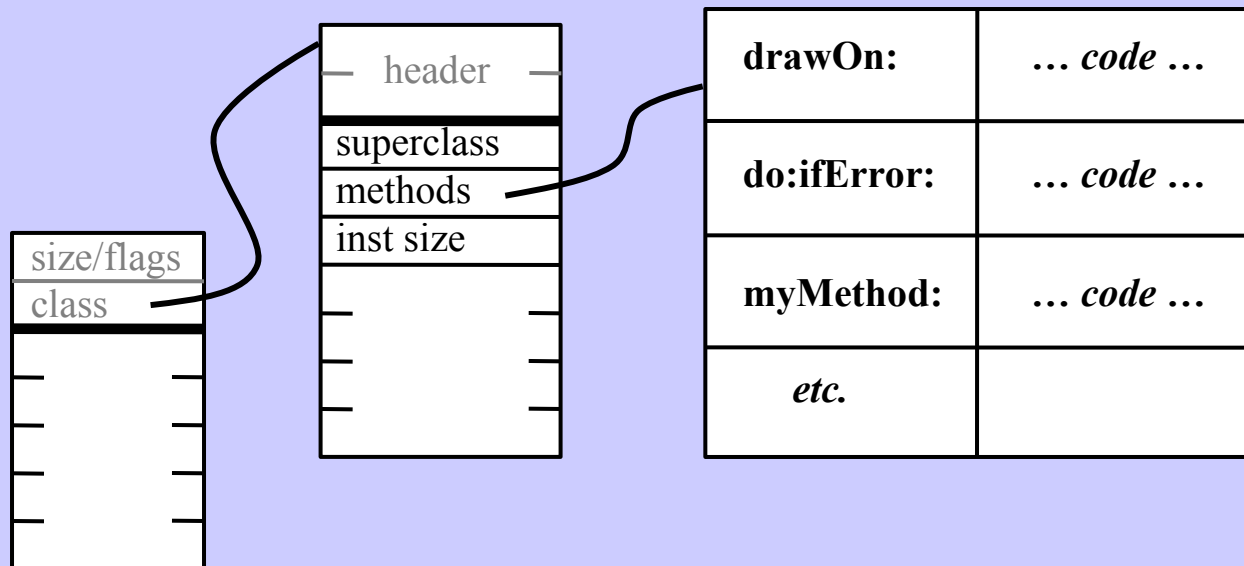
Representing Classes



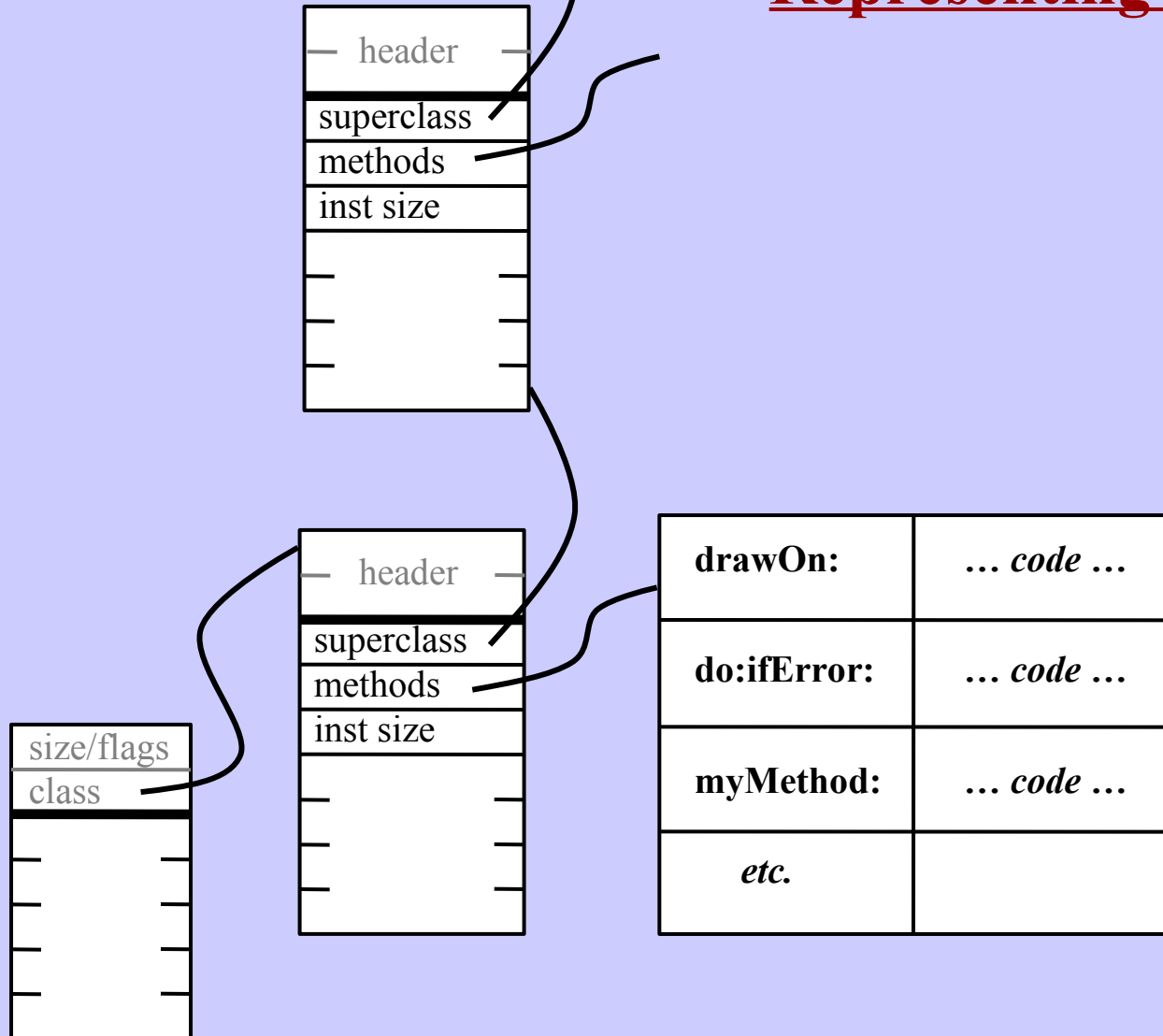
Representing Classes



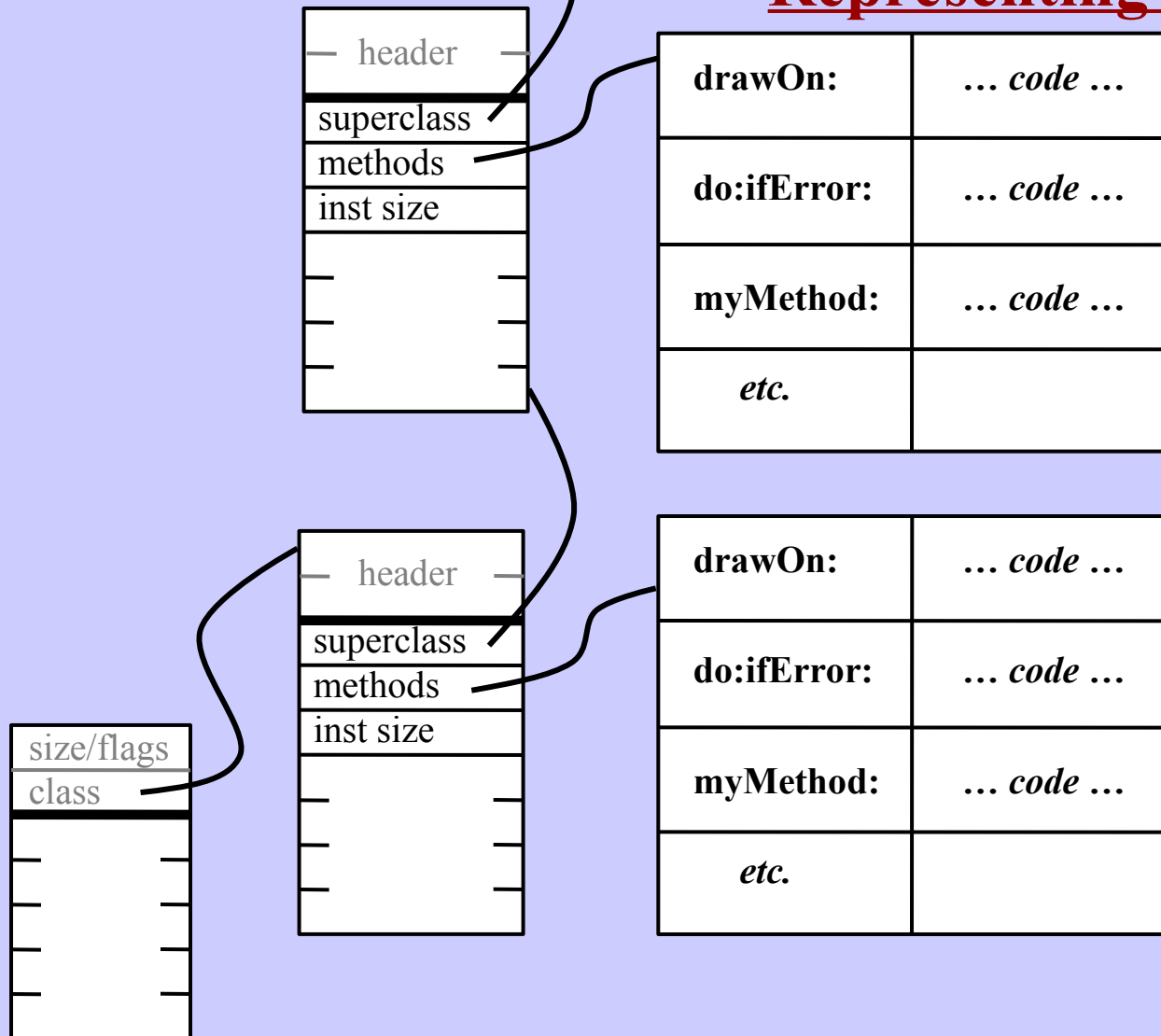
Representing Classes



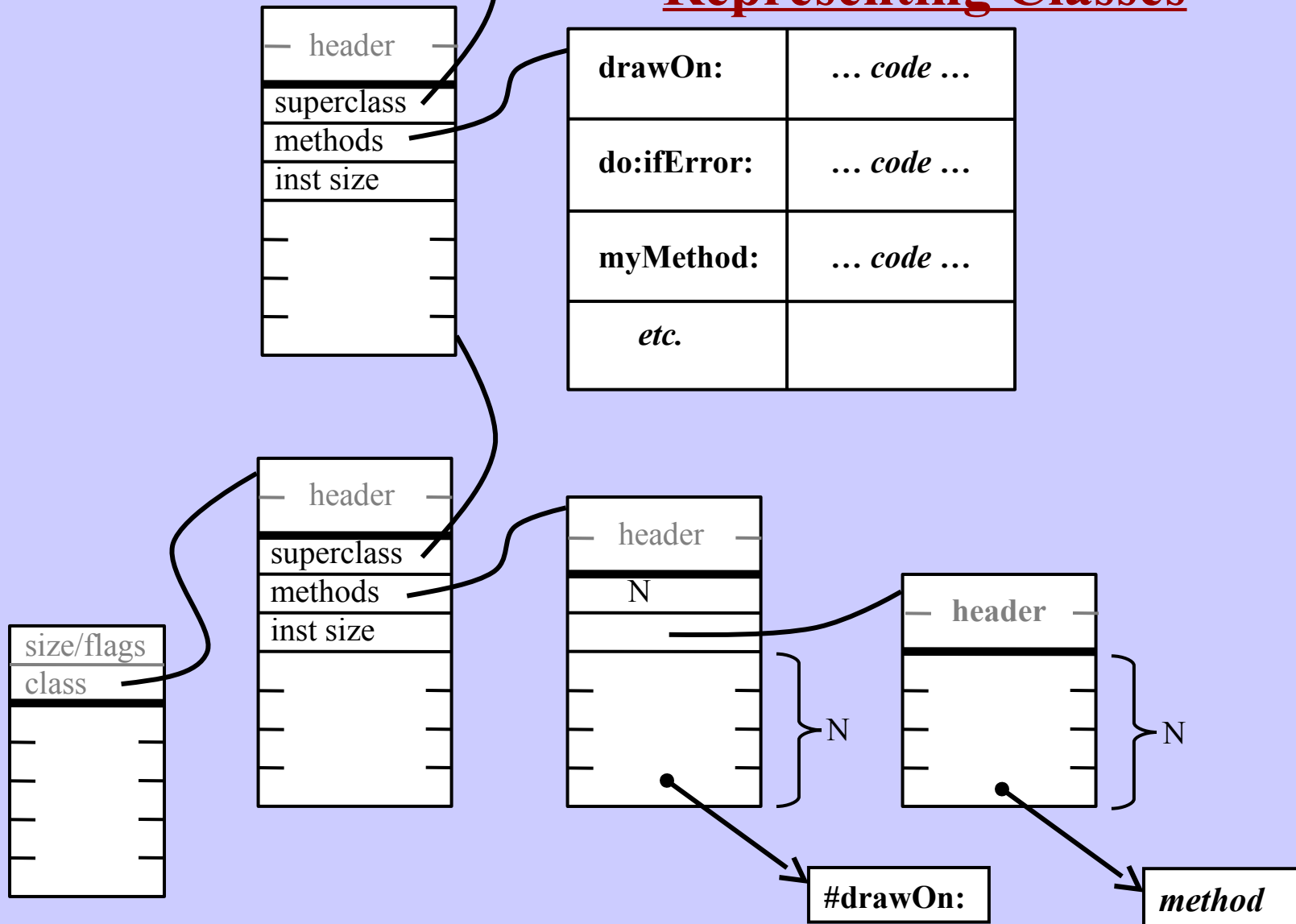
Representing Classes



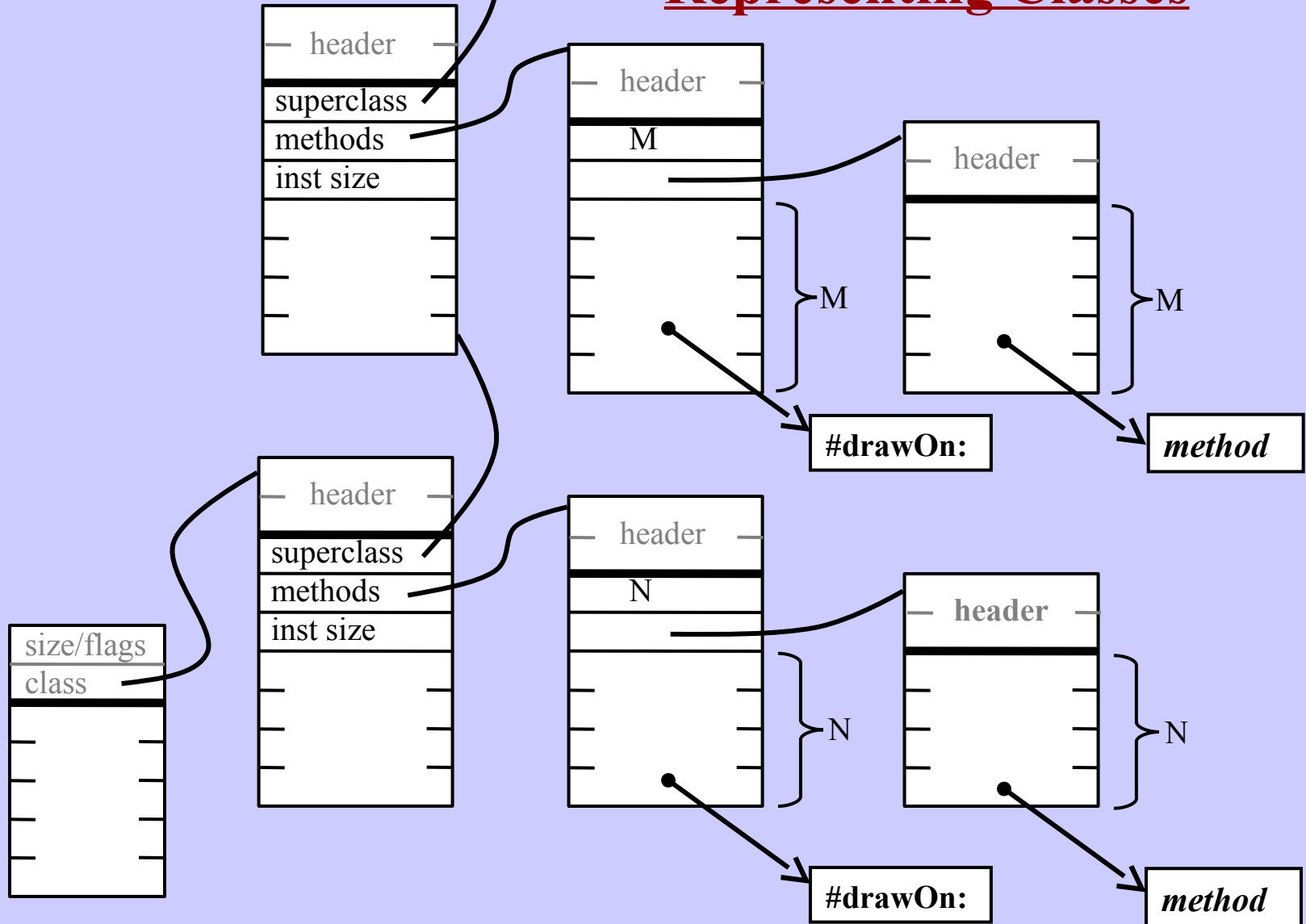
Representing Classes



Representing Classes



Representing Classes



Stack Machine Architectures

Typical instructions:

push

pop

add

call

return

jump

Example Source:

Compiler produces:

Stack Machine Architectures

Typical instructions:

```
push
pop
add
call
return
jump
```

Example Source:

```
4 + y
```

Compiler produces:

```
push 4
push y
add
```

Stack Machine Architectures

Typical instructions:

```
push  
pop  
add  
call  
return  
jump
```

Example Source:

```
x := 4 + y;
```

Compiler produces:

```
push 4  
push y  
add  
pop x
```

Stack Machine Architectures

Typical instructions:

```
push  
pop  
add  
call  
return  
jump
```

Example Source:

```
x := 4 + y;
```

Compiler produces:

```
push 4  
push y  
add  
pop x
```


Stack Machine Architectures

Typical instructions:

```
push
pop
add
call
return
jump
```

Example Source:

```
x := 4 + y * z;
```

Compiler produces:

```
push 4
push y
push z
mult
add
pop x
```

Stack Machine Architectures

Typical instructions:

```
push
pop
add
call
return
jump
```

Example Source:

```
x := 4 + y* z;
```

Compiler produces:

```
push 4
push y
push z
mult
add
pop x
```

Example Source:

```
x := 4 + foo(a,b+c) * z;
```

Compiler produces:

Stack Machine Architectures

Typical instructions:

push
pop
add
call
return
jump

Example Source:

x := 4 + **y*** z;

Compiler produces:

push 4
push y
push z
mult
add
pop x

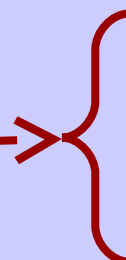
Example Source:

x := 4 + **foo(a,b+c)** * z;

Compiler produces:

push 4

push z
mult
add
pop x



Stack Machine Architectures

Typical instructions:

push
pop
add
call
return
jump

Example Source:

x := 4 + **y*** z;

Compiler produces:

push 4
push y
push z
mult
add
pop x

Example Source:

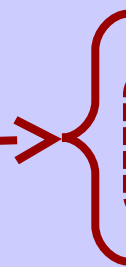
x := 4 + **foo(a,b+c)** * z;

Compiler produces:

push 4

push b
push c
add

push z
mult
add
pop x



Stack Machine Architectures

Typical instructions:

```
push
pop
add
call
return
jump
```

Example Source:

```
x := 4 + y * z;
```

Compiler produces:

```
push 4
push y
push z
mult
add
pop x
```

Example Source:

```
x := 4 + foo(a,b+c) * z;
```

Compiler produces:

```
push 4
push a
push b
push c
add
call foo
push z
mult
add
pop x
```



Stack Machine Architectures

Typical instructions:

push
pop
add
call
return
jump

Example Source:

x := 4 + **y*** z;

Compiler produces:

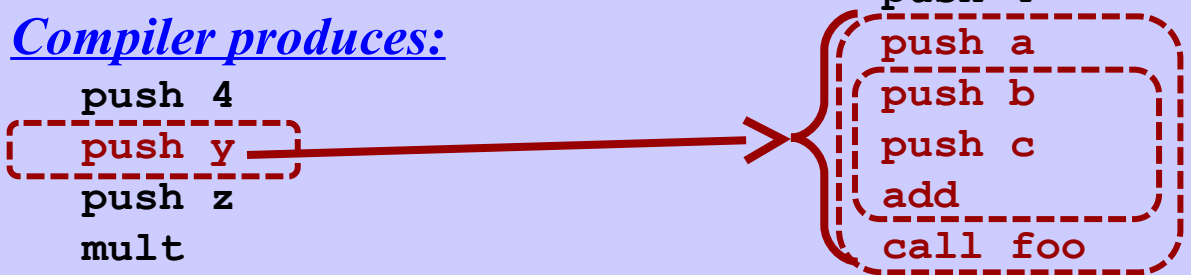
push 4
push y
push z
mult
add
pop x

Example Source:

x := 4 + (a do: b+c) * z;

Compiler produces:

push 4
push a
push b
push c
add
call foo
push z
mult
add
pop x



Stack Machine Architectures

Typical instructions:

push
pop
add
~~call~~ send
return
jump

Example Source:

x := 4 + y * z;

Compiler produces:

push 4
push y
push z
mult
add
pop x

Example Source:

x := 4 + (a do: b+c) * z;

Compiler produces:

push 4
push a
push b
push c
add
~~call fee~~ send #do:
push z
mult
add
pop x



The Virtual Machine

Typical instructions:

```
push x
pop x
sendMessage #xxx
returnTop
jump x
... etc ...
```

Each is encoded into 8-bit bytecode:

```
00  push receiver's 1st instance variable
01  push receiver's 2nd instance variable
60  pop into 1st instance variable
61  pop into 2nd instance variable
76  push constant 1
C0  send #at:
B1  send #-
7C  return top
... etc ...
```

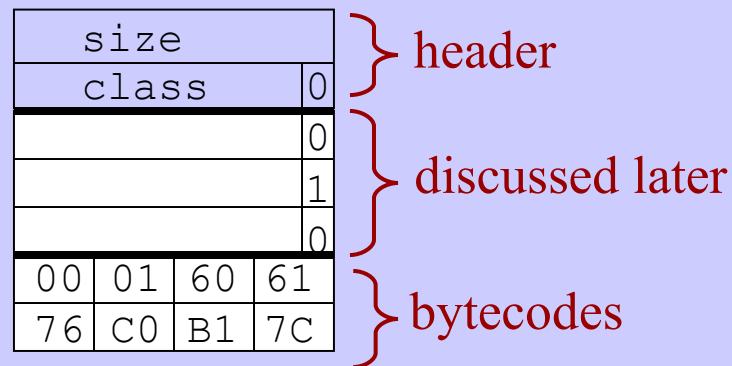

The Virtual Machine

Typical instructions:

```
push x
pop x
sendMessage #xxx
returnTop
jump x
... etc ...
```

Each is encoded into 8-bit bytecode:

```
00  push receiver's 1st instance variable
01  push receiver's 2nd instance variable
60  pop into 1st instance variable
61  pop into 2nd instance variable
76  push constant 1
C0  send #at:
B1  send #-
7C  return top
... etc ...
```



As Stored in the Object:

An Example Method

Method:

```
popLifo
| myTemp |
myTemp ← lifoArray at: lifoTop.
lifoTop ← lifoTop - 1.
^ myTemp
```

Class:

Lifo

Instance Variables:

lifoArray (1st inst var)

lifoTop (2nd inst var)

Compiled Bytecodes:

An Example Method

Method:

```
popLifo
  | myTemp |
  myTemp ← lifoArray at: lifoTop.
  lifoTop ← lifoTop - 1.
  ^ myTemp
```

Class:

Lifo

Instance Variables:

lifoArray (1st inst var)

lifoTop (2nd inst var)

Compiled Bytecodes:

```
00  Push receiver's 1st instance variable (lifoArray)
01  Push receiver's 2nd instance variable (lifoTop)
C0  Send binary message #at:
68  Pop stack into 1st temp variable (myTemp)
01  Push receiver's 2nd instance variable (lifoTop)
76  Push constant 1
B1  Send binary message #-
61  Pop stack into receiver's 2nd instance variable (lifoTop)
10  Push 1st temp variable (myTemp)
7C  Return stack top
```

Bytecodes Can Refer to Operands

Directly:

The **receiver** (self)

The **arguments** to the method

The receiver's **instance variables**

The **temporary variables** (i.e., “local” variables)

Some common **constants**:

`nil, true, false, -1, 0, 1, 2`

32 common **message selectors**:

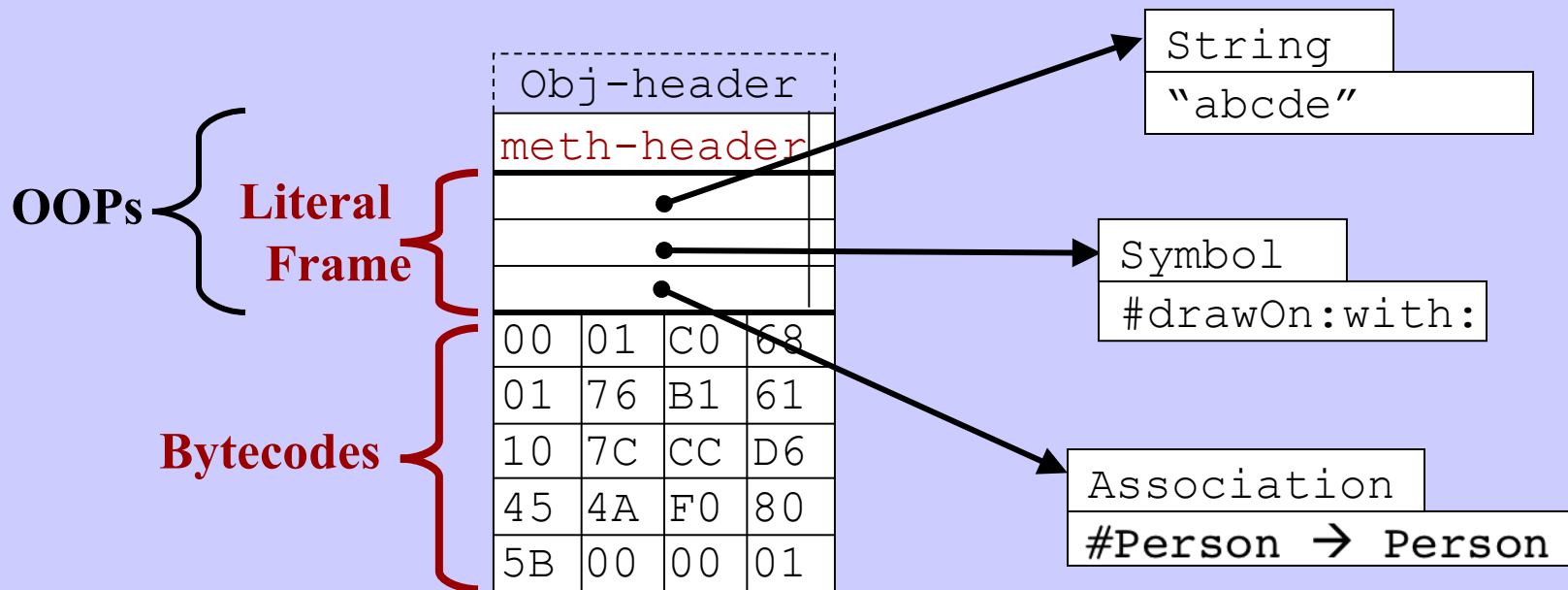
`+ - < = at: at:put: @ x y ...`

Indirectly:

Thru the “literal frame”:

- **Constants** occurring within the method (e.g., 57, \$a, ‘abc’)
- All other **message selectors**
- **Global variables** (e.g., class names)

The Format of *CompiledMethod* Objects



The *CompiledMethod* Header

- The size of the activation record (i.e., the “stack frame”)
- The number of temporary variables for this method
- Number of literals (i.e., where to find 1st bytecode)
- Additional flags:
 - Just return self
 - Just return instance variable k (where k = 0 .. 31)
 - Is this a “normal” method?
 - Number of arguments? 0 .. 4
 - An extension header word is used for all other cases
 - Number of arguments? (0 .. 31)
 - Is this a primitive method? (0 .. 255)

Message Selectors

From the bytecode, the interpreter can get

the message selector

the number of arguments

32 commonly used selectors are handled specially

+ - < = @ do: at: at:put: class ...

Two versions of the “send-message” bytecode

- Optimized encoding for the 32 common selectors

1 0 1 - - - - -

- The more general version

Longer than 1 byte

The number of **arguments**

32 common selectors → Implicit

“general” send-message bytecode → Encoded into the instruction

All Other Selectors

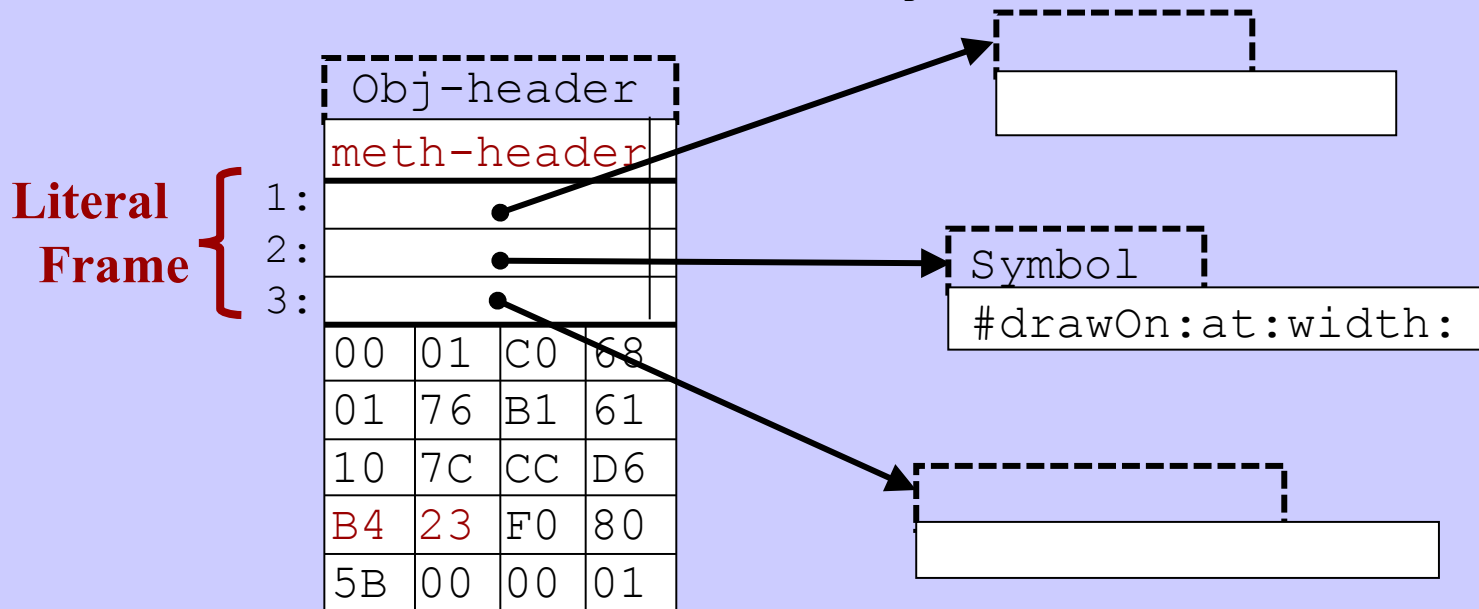
The remaining selectors are stored in the **literal** area

The bytecode for a “general” send includes:

- Which literal field points to the selector
- Number of arguments

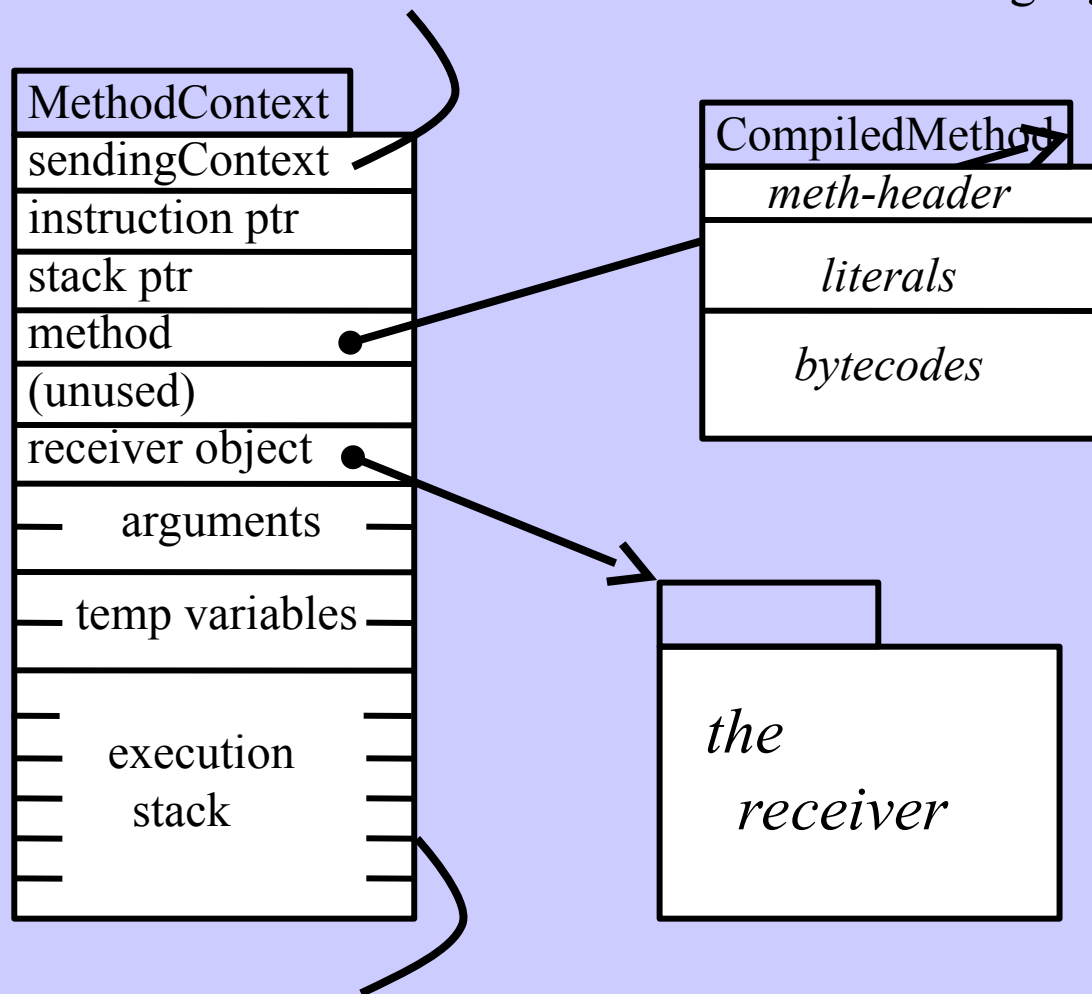
Bytecode:

B4 23 Send (literal: 2; numArgs: 3)



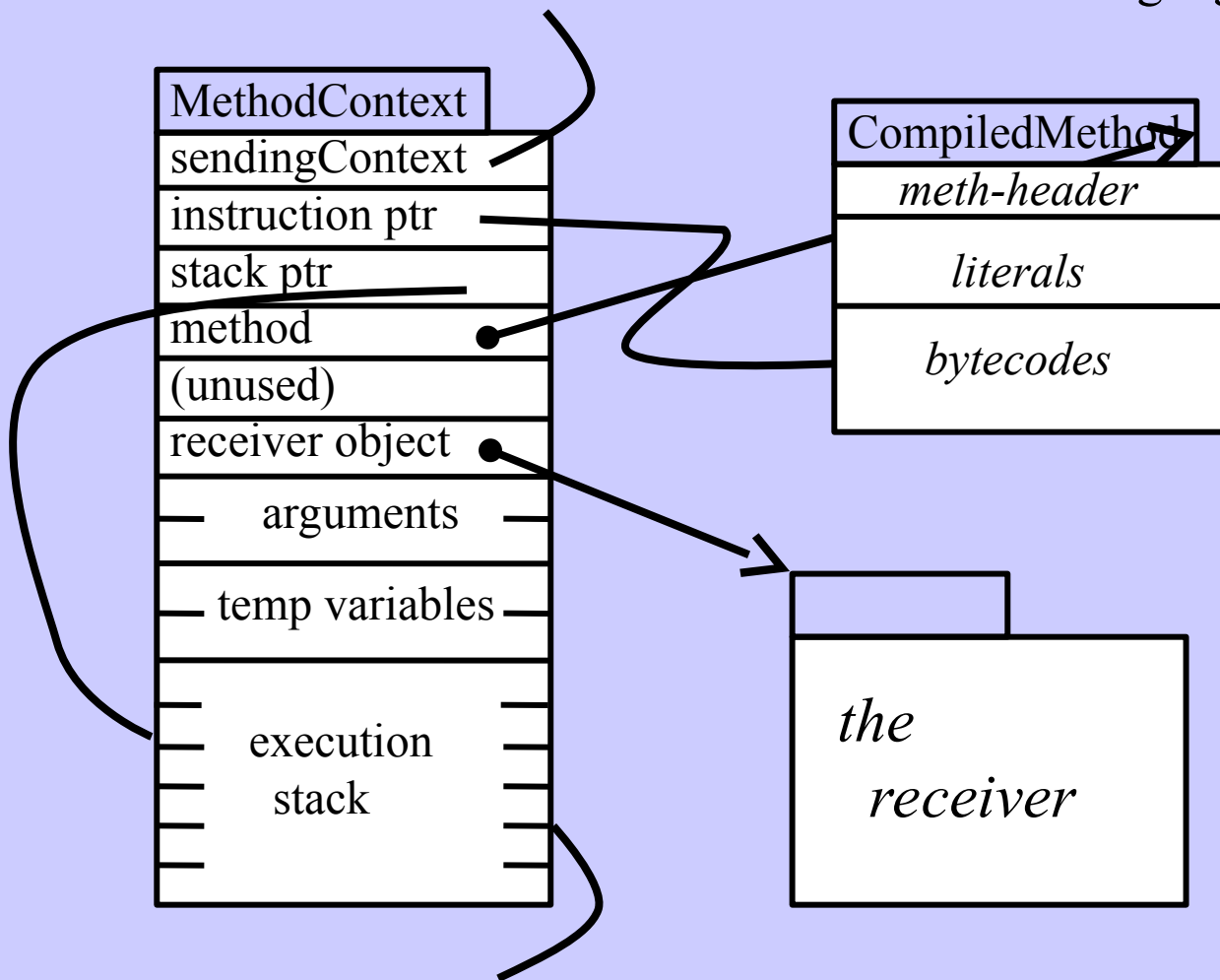
Activation Records

- When a method is called, a *MethodContext* is created.
- Like an “Activation Record” or “Frame” in traditional language



Activation Records

- When a method is called, a *MethodContext* is created.
- Like an “Activation Record” or “Frame” in traditional language



What Happens When a Message is Sent?

x at: y put: z

```
00    push x onto the stack
00    push y onto the stack
00    push z onto the stack
00    send #at:put: message (numArgs: 2)
```

(Pops recvr and args. Leave result on top of sender's stack.)

- **Find the receiver** buried underneath the args
- **Do method lookup** to obtain the *CompiledMethod* object
- **Allocate a new *MethodContext***
(The *CompiledMethod* tells how big the *MethodContext* should be)
- **Initialize the *MethodContext***
 - Pointer to receiver
 - Instruction pointer
 - Pointer to the *CompiledMethod* object
 - Pointer to the top of the stack
 - Pointer to the sending context
- **Pop the message arguments** and store into the new *MethodContext*
- **Begin executing bytecodes** in the new method, using the new *MethodContext*

MethodContexts are Objects!

Advantages

- *MethodContexts* live in the object heap
Running code can be saved in the “image” file
- ***Debugger can access them easily***
Debugging tools can be written in Smalltalk
- Blocks are represented as objects, too!
A *BlockContext* object can be passed around, stored, etc.
You can send messages to blocks (e.g., #value)

Disadvantages

MethodContexts are Objects!

Advantages

- *MethodContexts* live in the object heap
Running code can be saved in the “image” file
- ***Debugger can access them easily***
Debugging tools can be written in Smalltalk
- Blocks are represented as objects, too!
A *BlockContext* object can be passed around, stored, etc.
You can send messages to blocks (e.g., #value)

Disadvantages

- Creation overhead!
- Very short lifetimes!
→ **Big strain on the garbage collector**

MethodContexts are Objects!

Advantages

- *MethodContexts* live in the object heap
- Running code can be saved in the “image” file
- ***Debugger can access them easily***
- Debugging tools can be written in Smalltalk
- Blocks are represented as objects, too!
- A *BlockContext* object can be passed around, stored, etc.
- You can send messages to blocks (e.g., `#value`)

Disadvantages

- Creation overhead!
 - Very short lifetimes!
- ***Big strain on the garbage collector***

Conclusion:

A worthwhile abstraction
... but special optimizations are mandatory!
(A stack is really used)

PrimitiveMethods

- Some methods are implemented directly in the VM.
SmallInteger arithmetic, I/O, performance critical code, etc.
- The VM executes a native “C” function.
Normal bytecode execution does not happen.
- Primitive operations may “fail”.
e.g., the “C” code cannot handle some special cases.
The native code terminates
The method is executed, as normal.

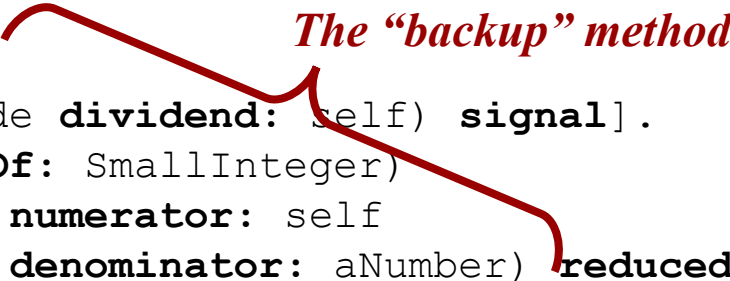
PrimitiveMethods

- Some methods are implemented directly in the VM.
SmallInteger arithmetic, I/O, performance critical code, etc.
- The VM executes a native “C” function.
Normal bytecode execution does not happen.
- Primitive operations may “fail”.
e.g., the “C” code cannot handle some special cases.
The native code terminates
The method is executed, as normal.

Example from SmallInteger:

```
/ aNumber  
  <primitive: 10>  
  aNumber isZero  
  ifTrue: [^(ZeroDivide dividend: self) signal].  
          (aNumber isMemberOf: SmallInteger)  
  ifTrue: [^(Fraction numerator: self  
                      denominator: aNumber) reduced]  
  ifFalse: [^super / aNumber]
```

The “backup” method



PrimitiveMethods – Implementation

A flag in the header of the *CompiledMethod*

- Does this method have a “primitive” implementation?
- Header includes the primitive number (0 .. 255)

The *MethodContext* is not created

Instead, a **native routine** in the VM is called.

The native routine manipulates values on the sender’s stack

- Pop arguments off the stack
- Leave the result on the stack

Problems while executing a primitive?

Primitives execution “fails”

Undo any partial execution

Execute the backup method

Create a *MethodContext*

Execute the *CompiledMethod*’s bytecodes

Blocks

Every block is an object

```
...  
b4 := [ :x :y | stmt. stmt. stmt. x+y ].  
...
```

```
...  
z := b4 value: a value: b.  
...
```

Blocks

Every block is an object

```
...  
b4 := [ :x :y | stmt. stmt. stmt. x+y ].  
...
```

```
...  
z := b4 value: a value: b.  
...
```

BlockContext

When encountered in execution, a *BlockContext* is created.
When evaluated, it's like invoking a method.

After execution, the block returns

... to the caller

```
[ :x :y | stmt. stmt. stmt. x+y ]
```

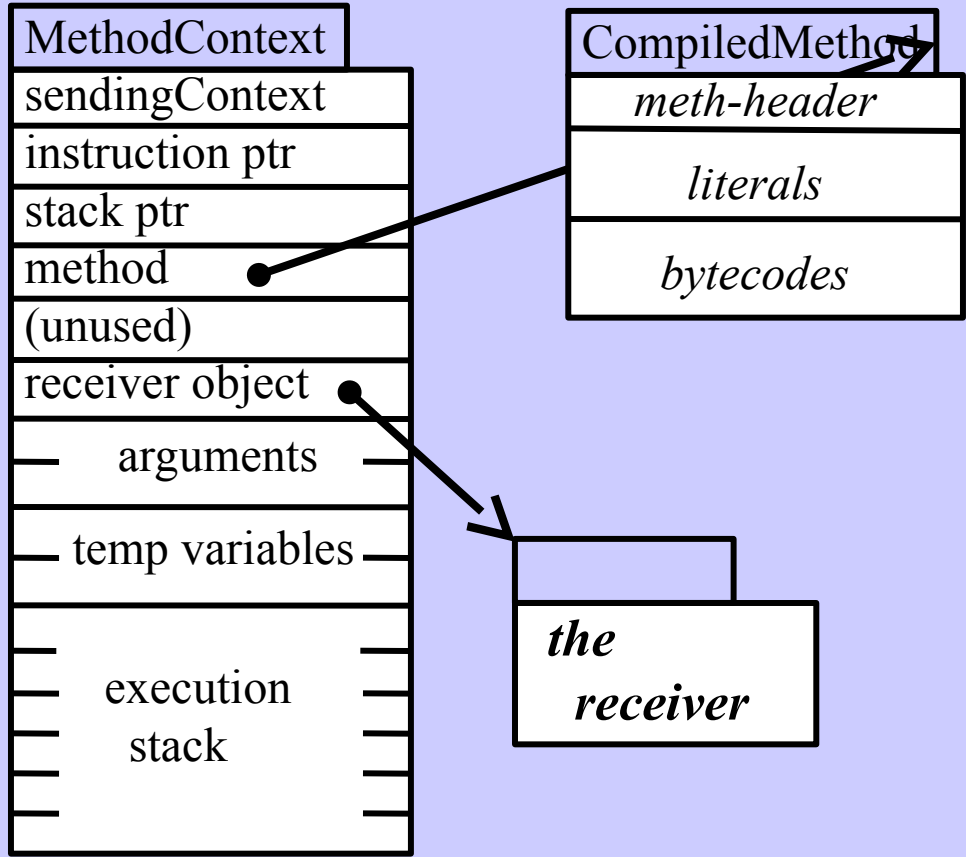
... from the method where it was created

```
[ :x :y | stmt. stmt. stmt. ^x+y ]
```

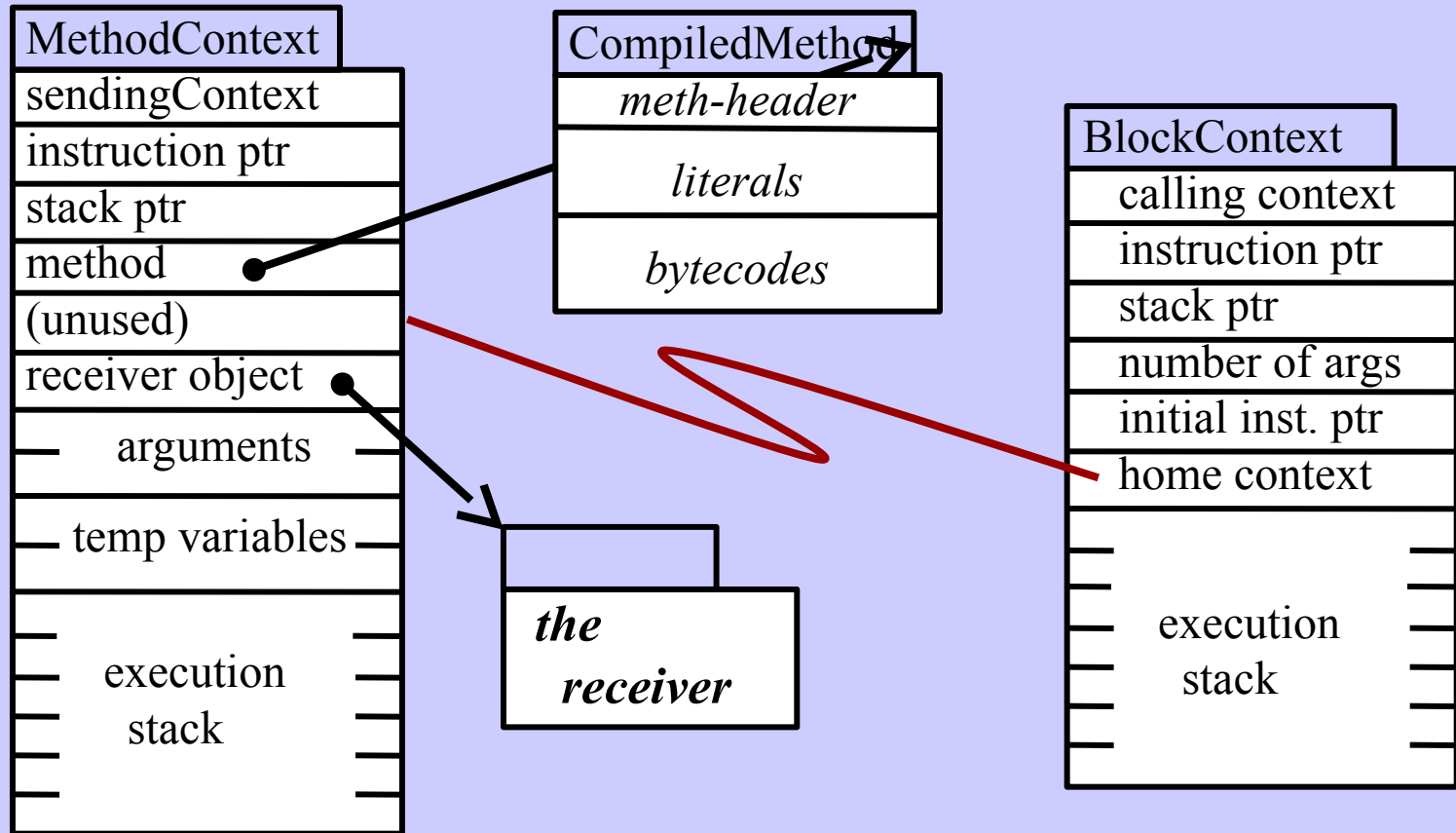
The *BlockContext* object will be garbage collected

when no longer needed (i.e., not reachable)

How are Blocks Represented?

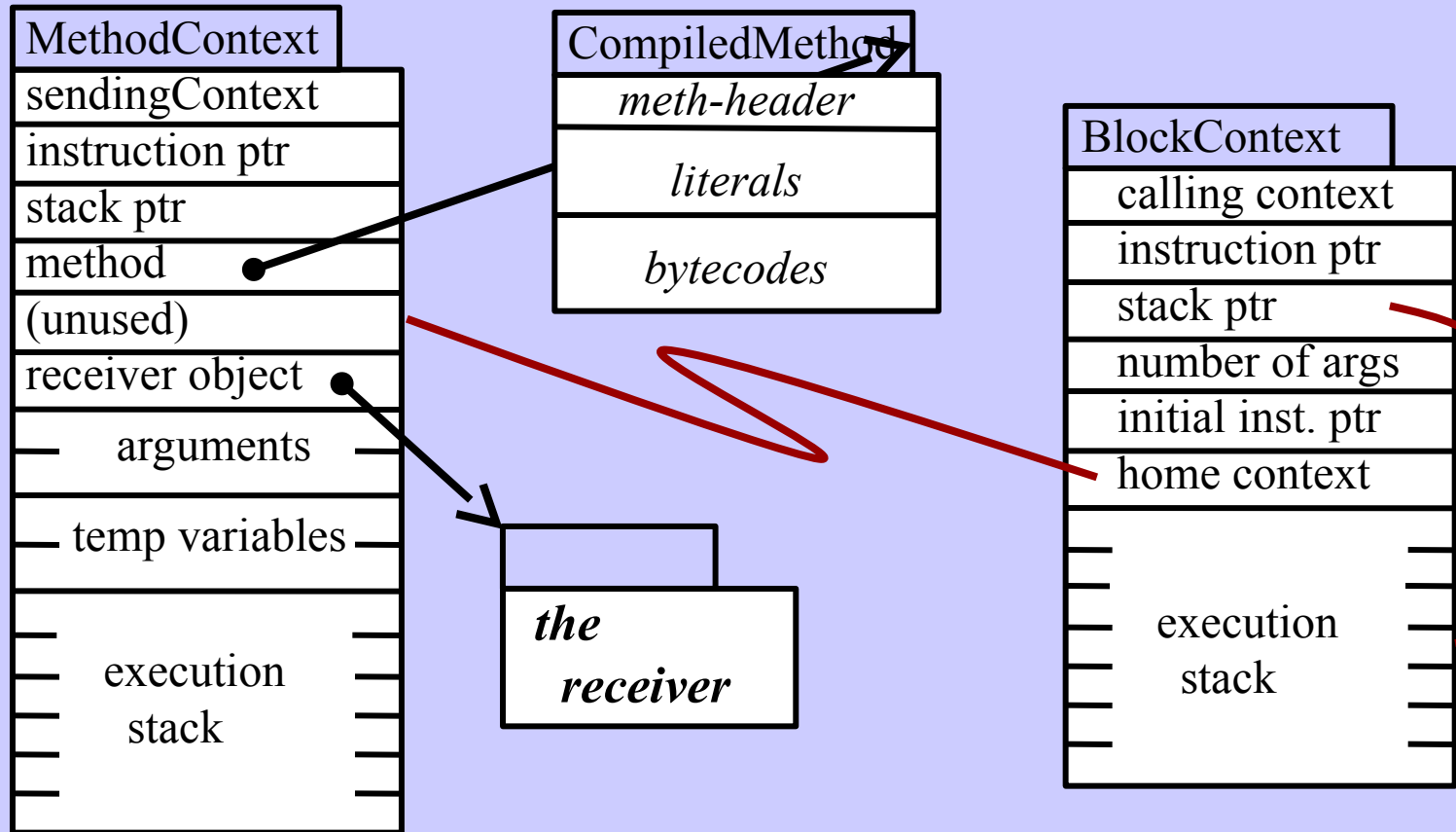


How are Blocks Represented?



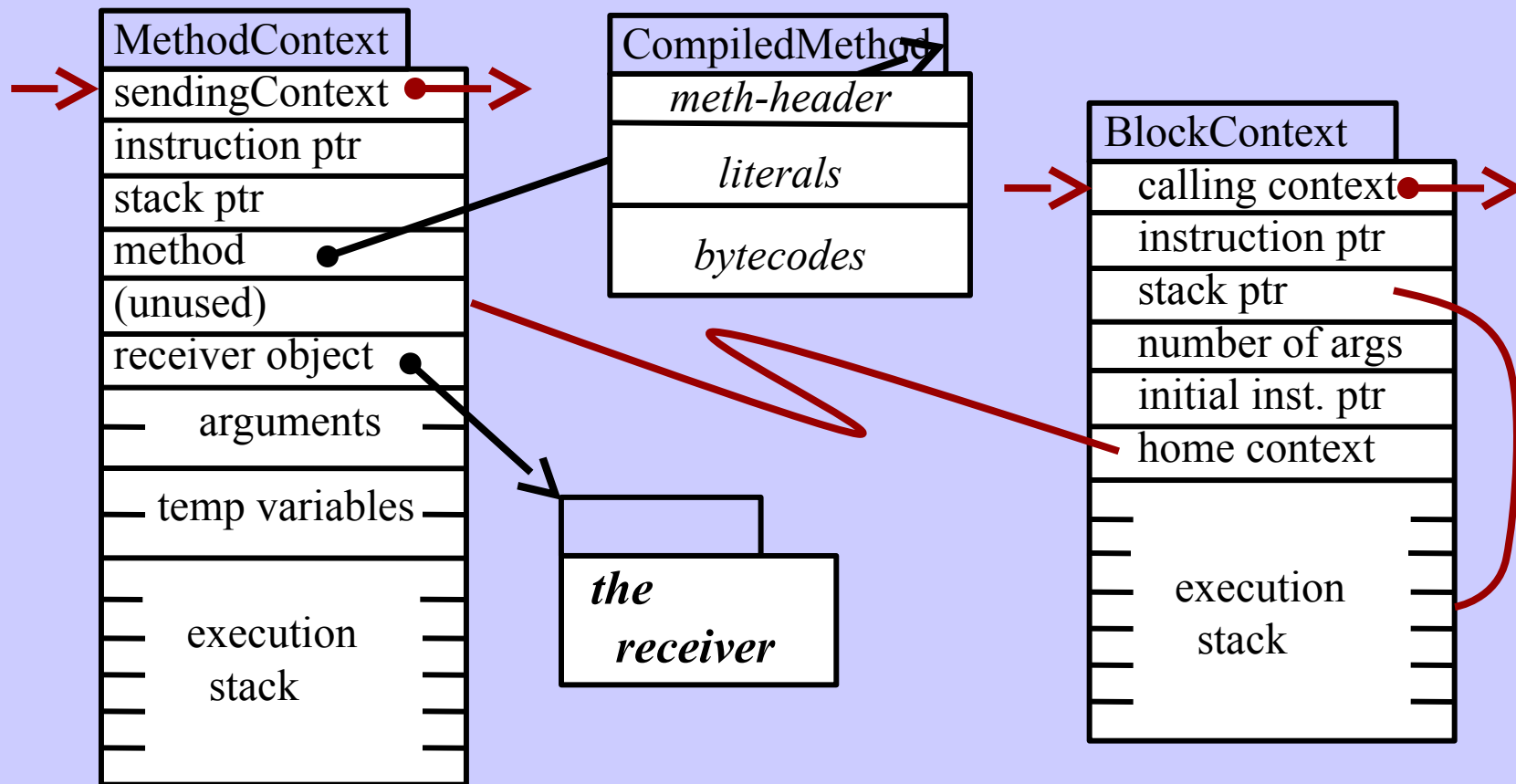
When created, the *BlockContext* has a pointer back to its *“home context”*.

How are Blocks Represented?



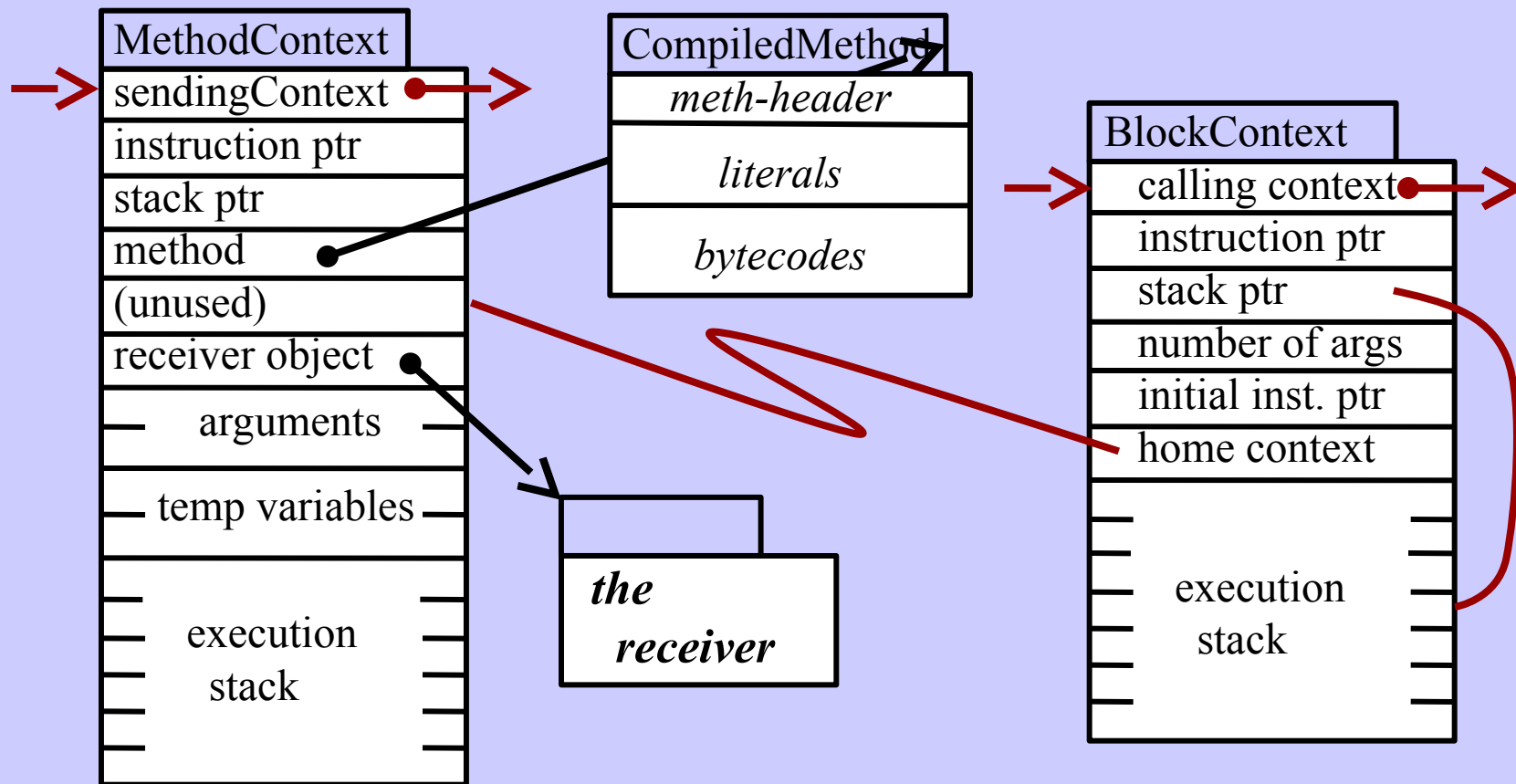
The BlockContext has its own *execution stack*

How are Blocks Represented?



When evaluated (i.e., when invoked / called)...
the **BlockContext** is added to the **“calling stack”** of frames.

How are Blocks Represented?



NOTE: The block keeps its temps and arguments in the home context.
Only one invocation active at one time; NO RECURSION!

The blockCopy: **primitive**

*Will create a block object.
Will push a ptr to it onto
stack.*

The blockCopy: primitive

```
incrAll  
  ^ self do: [ :x | x incr ]
```

*Will create a block object.
Will push a ptr to it onto
stack*

CompiledMethod:

Header:

Literals:

Bytecodes:

The blockCopy: primitive

```
incrAll  
  ^ self do: [ :x | x incr ]
```

*Will create a block object.
Will push a ptr to it onto
stack*

CompiledMethod:

Header: 1 temp variable needed (*x*)

Literals: #incr

Bytecodes:

70 Push receiver (self) onto stack

CB Send #do:

7C Return stack top

The blockCopy: primitive

```
incrAll  
  ^ self do: [ :x | x incr ]
```

*Will create a block object.
Will push a ptr to it onto
stack*

CompiledMethod:

Header: *1 temp variable needed (x)*

Literals: #incr

Bytecodes:

```
70      Push receiver (self) onto stack  
89      Push the active context onto the stack  
76      Push 1 onto the stack (num args to block)  
C8      Send #blockCopy:  
A4 04   Jump around next 4 bytes  
68      Pop stack into 1st temp variable (x)  
10      Push 1st temp var (x) onto the stack  
D0      Send #incr  
7D      Block Return (return stack top as block's result)  
CB      Send #do:  
7C      Return stack top
```

blockCopy:

Skip this slide

- A primitive method
 - Passed the number of arguments
 - Sent to the current context
 - (The “**home context**”)
- Creates a new *BlockContext* object
 - Initializes its “**HomeContext**” field
 - Initializes its “**InitialInstructionPointer**” field
 - Based on the current instruction pointer + 2
 - Pushes an OOP to the new *BlockContext* onto the current stack
- Storage for arguments to the block...
 - The block’s arguments must be allocated space somewhere.
 - They are allocated in the home context (as temp variables)
 - A block begins by popping its arguments into the home context
 - What if the method that created the block has already returned?
 - No problem; the space still exists.
 - Why will the home context not get garbage collected?

Blocks have two ways of returning

```
(x < y)
  ifTrue: [ stmt. stmt. stmt. 43 ]
  ifFalse: [ stmt. stmt. stmt. ^43 ]
```

Normal Return

How does a block return?

Pop a value off of the current stack.

Push it (the return value) onto the caller's stack.

Resume executing caller's instructions.

Return from enclosing method

Normal return from a block:

Push result onto **Calling Context**'s stack.

Resume execution using **Calling Context**'s instruction pointer.

Return from enclosing method:

Look at the **Home Context**.

Look at its **Sending Context**

Push result onto that context's stack.

Resume execution using that context.

Blocks in Smalltalk are not “Closures”

```
| fact |  
fact ← [ :n |  
  (n < 1)  
    ifTrue: [1]  
    ifFalse: [ n * (fact value: (n - 1)) ]  
].  
  
fact value: 4 → ???
```

The block invokes itself recursively.
This code will not work correctly!

In Smalltalk / Squeak:

Blocks may not be entered recursively.

Only one *BlockContext* is created.

Storage for only one copy of “n”

The interpreter will catch this.

“Attempt to evaluate a block that is already being evaluated”

Pharo Implements Blocks Differently

```
incrAll
```

```
^ self do: [ :x | x incr ]
```

CompiledMethod:

Header: *numArgs=0, numTemps=0, isPrimitive=No,*

Literals: #incr

Bytecodes:

70 Push receiver (self) onto stack

81
01
00
03

} closureNumCopied: 0, numArgs=1, next 3 bytes

10 pushTemp: 0

D0 send: #incr

7D Block Return (return stack top as block's result)

CB Send #do:

7C Return stack top

Message Sending in C++

Skip these slides

Message Sending in C++

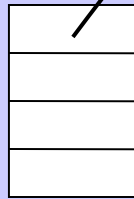
Source:

p calcBenefits: x with: y

Bytecodes:

push p
push x
push y
send 8, 2

<header>
name
ssNumber
addr



className	"Person"
size	16
8:	
12:	
16:	
20:	

calcBenefits:
...code...
return

setName:
...code...

foo:
...code...

bar:
...code...

Message Sending in C++

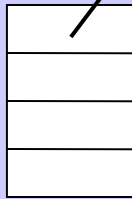
Source:

p calcBenefits: x with: y

Bytecodes:

push p
push x
push y
send 8, 2

<header>
name
ssNumber
addr



className	"Person"
size	16
8:	
12:	
16:	
20:	

calcBenefits:
...code...
return

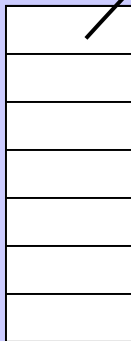
setName:
...code...

foo:
...code...

bar:
...code...

Add a subclass...

<header>
name
ssNumber
addr
major
advisor
gpa



className	"Student"
size	28
8:	
12:	
16:	
20:	
24:	
28:	
32:	

Message Sending in C++

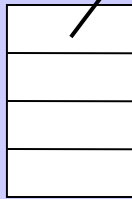
Source:

p calcBenefits: x with: y

Bytecodes:

push p
push x
push y
send 8, 2

<header>
name
ssNumber
addr



className	"Person"
size	16
8:	
12:	
16:	
20:	

calcBenefits:
...code...
return

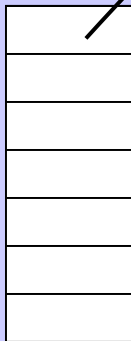
setName:
...code...

foo:
...code...

bar:
...code...

*Override some methods,
and add new ones...*

<header>
name
ssNumber
addr
major
advisor
gpa



className	"Student"
size	28
8:	
12:	
16:	
20:	
24:	
28:	
32:	

calcBenefits:
...code...

method2:
...code...

method3:
...code...

method4:
...code...

Message Sending in C++

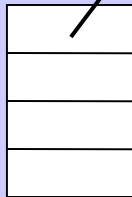
Source:

p calcBenefits: x with: y

Bytecodes:

push p
push x
push y
send 8, 2

<header>
name
ssNumber
addr



className	"Person"
size	16
8:	
12:	
16:	
20:	

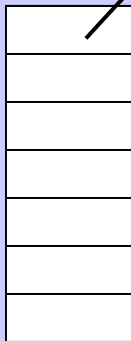
calcBenefits:
...code...
return

setName:
...code...

foo:
...code...

bar:
...code...

<header>
name
ssNumber
addr
major
advisor
gpa



className	"Student"
size	28
8:	
12:	
16:	
20:	
24:	
28:	
32:	

calcBenefits:
...code...

method2:
...code...

method3:
...code...

method4:
...code...