# Lecture 12: Programming (Virtual) Machines

**Eelco Visser**

**CS4200 Compiler Construction**
**TU Delft**
**November 2018**

TU Delft

# Virtual Machines

- (Virtual) machine architecture
  - In particular: Java Virtual Machine
- Machine instructions
- Encoding linguistic abstractions
- Optimizing generated code

## Linguistic Abstraction

- High-level programming languages abstract from low-level machine mechanics
- Abstractions in imperative and object-oriented languages
- Calling Conventions
- Register machines vs stack machines

# Summary

## Stack frames in the Java Virtual Machine

- parameter passing, returning results
- implementation strategies

## Stack frames in register-based machines

- registers x86 family
- manipulating stack registers
- calling conventions

## Optimizations

# Reading Material

# The Java® Virtual Machine Specification

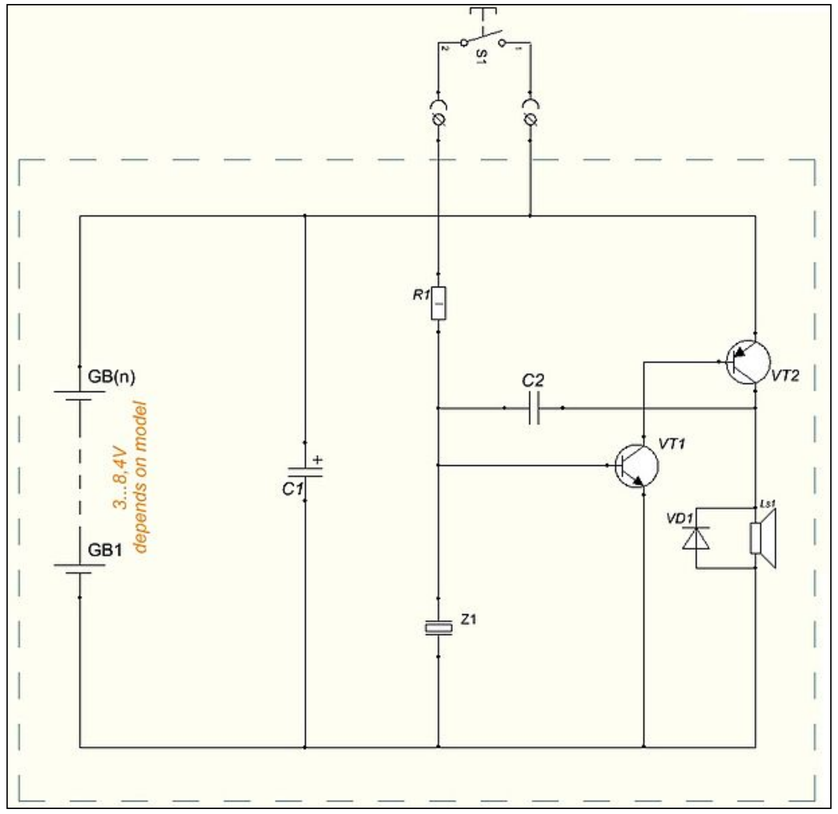
*Java SE 8 Edition*

Tim Lindholm
Frank Yellin
Gilad Bracha
Alex Buckley

2015-02-13

# Programmable Computing Machines

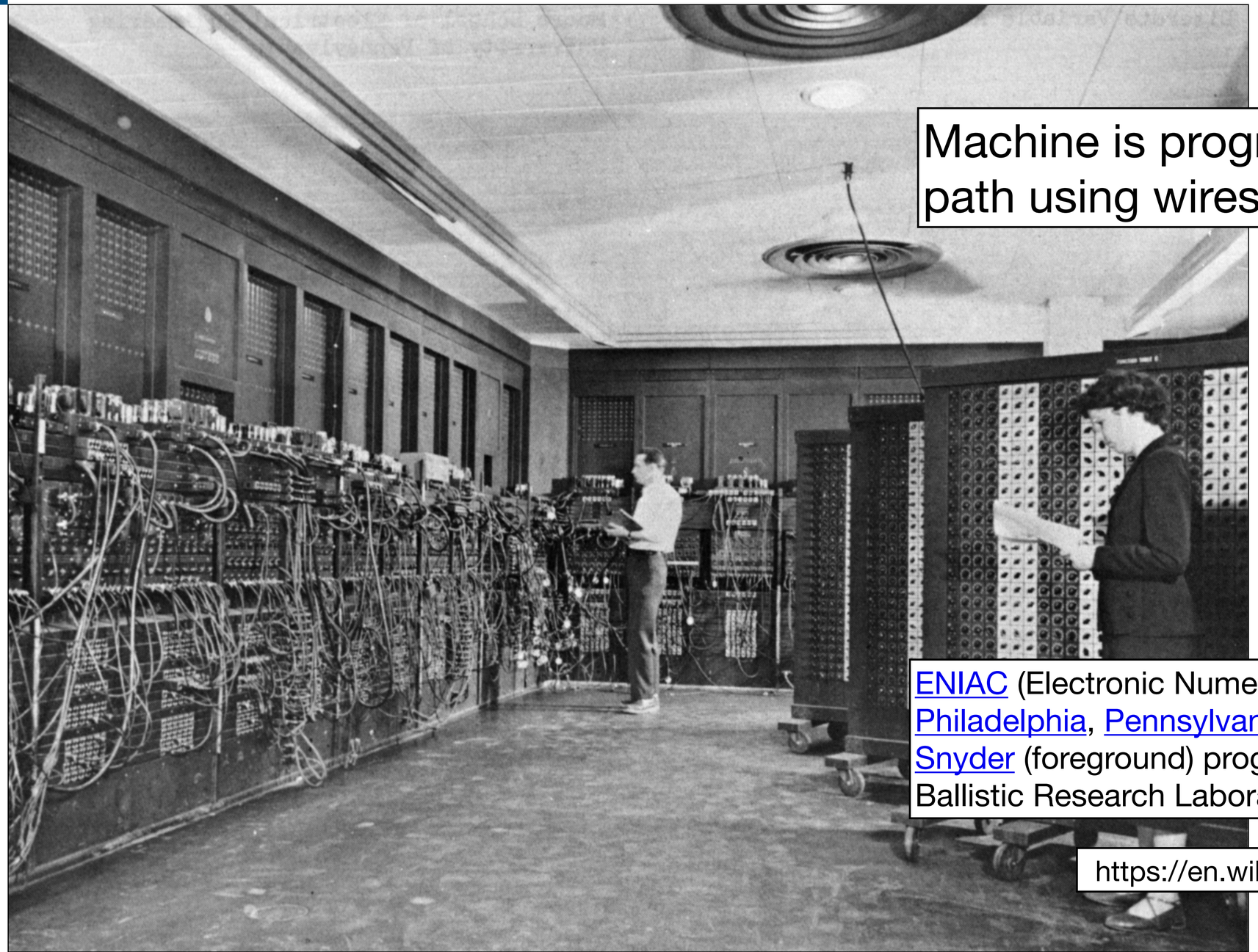
# Hard-Wired Programs




Fixed to perform one computation from input to output

https://commons.wikimedia.org/wiki/File:Wiring_diagram_of_battery-powered_doorbell.JPG

# Programmable Machines



Machine is programmed by creating data path using wires

ENIAC (Electronic Numerical Integrator And Computer) in Philadelphia, Pennsylvania. Glen Beck (background) and Betty Snyder (foreground) program the ENIAC in building 328 at the Ballistic Research Laboratory (BRL).

https://en.wikipedia.org/wiki/ENIAC#/media/File:Eniac.jpg

# Stored-Program Computer (Von Neumann Architecture)

## Central Processing Unit

- Processor registers
- Arithmetic logic unit

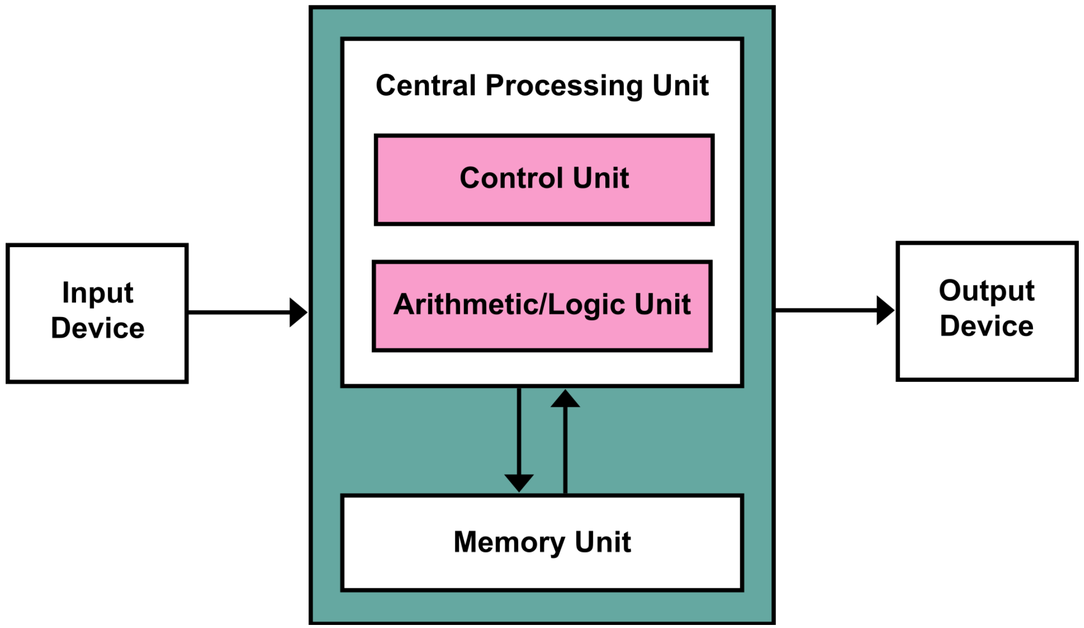
## Main Memory

- Stores data and instructions

## External Storage

- Persistent storage of data

## Input/Output




https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Von_Neumann_Architecture.svg

# State & Control

# State

## Machine state

- data stored in memory
- memory hierarchy: registers, RAM, disk, network, …

## Imperative program

- computation is series of changes to memory
- basic operations on memory (increment register)
- controlling such operations (jump, return address, …)
- control represented by state (program counter, stack, …)

# Registers: x86 Family

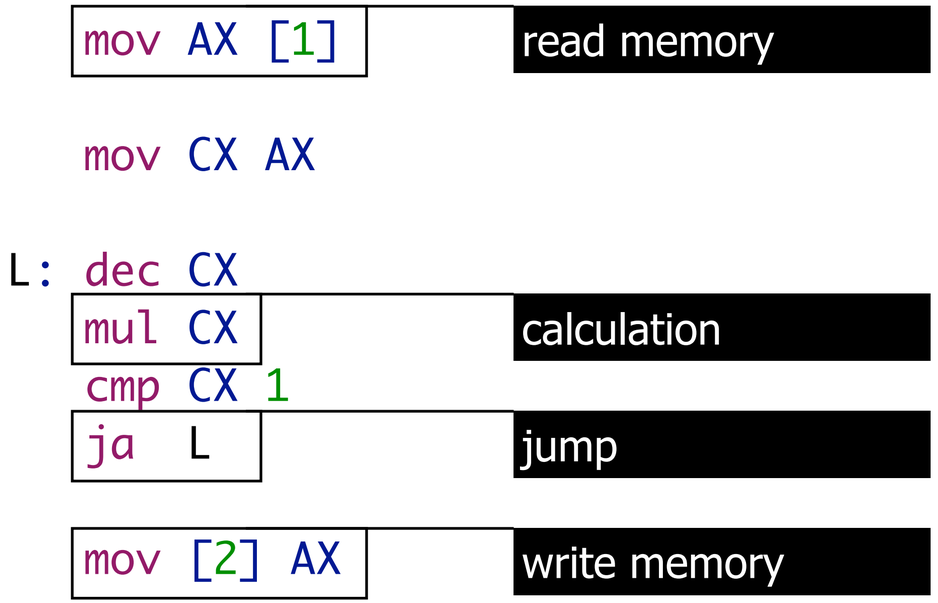
## General purpose registers

- accumulator AX - arithmetic operations
- counter CX - shift/rotate instructions, loops
- data DX - arithmetic operations, I/O
- base BX - pointer to data
- stack pointer SP, base pointer BP - top and base of stack
- source SI, destination DI - stream operations

## Special purpose registers

- segments SS, CS, DS, ES, FS, GS
- flags EFLAGS

# Example: x86 Assembler



# Example: Java Bytecode

```
.method static public m(I)I

        iload 1
        ifne else
        iconst_1
        ireturn

else:   iload 1
        dup
        iconst_1
        isub
        invokestatic Math/m(I)I
        imul
        ireturn
```

jump

read memory

calculation

# Memory & Control Abstractions

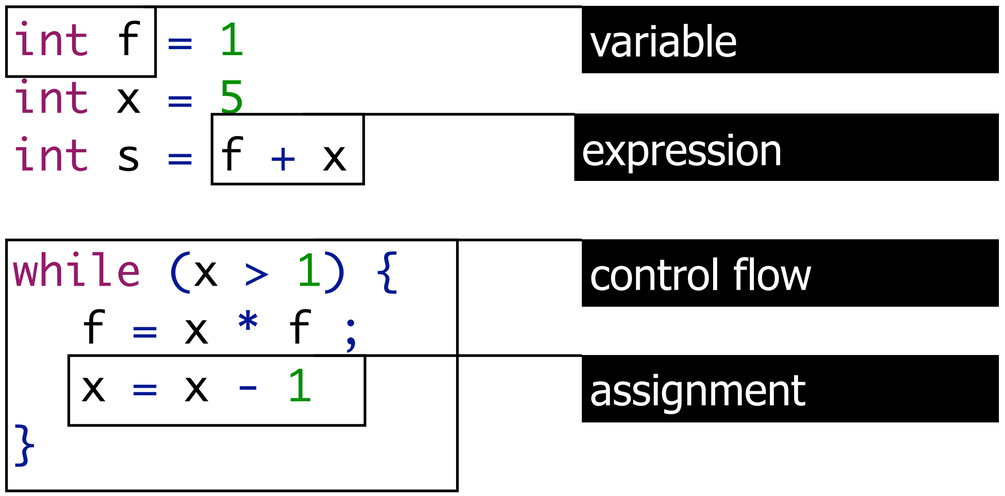
## Memory abstractions

- variables: abstract over data storage
- expressions: combine data into new data
- assignment: abstract over storage operations

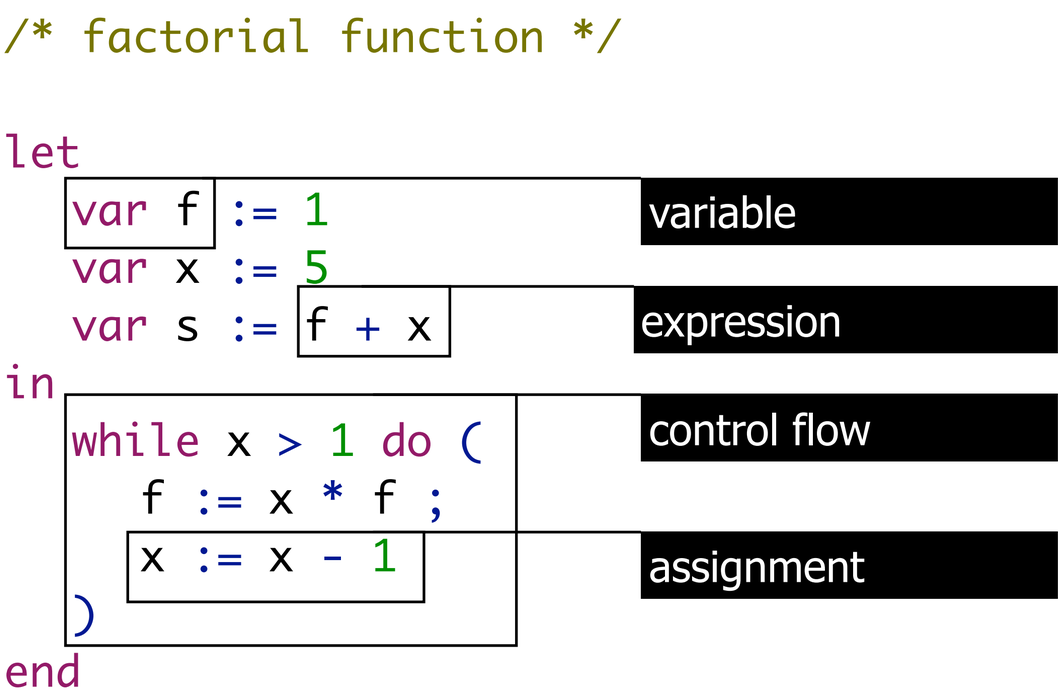
## Control-flow abstractions

- structured control-flow: abstract over unstructured jumps
- ‘go to statement considered harmful’ Edgser Dijkstra, 1968

# Example: C


int f = 1
int x = 5
int s = f + x
while (x > 1) {
f = x * f ;
x = x - 1
}
variable
expression
control flow
assignment

# Example: Tiger



# Procedures

# Procedural Abstraction

## Control-flow abstraction

- Procedure: named unit of computation
- Procedure call: jump to unit of computation and return

## Memory abstraction

- Formal parameter: the name of the parameter
- Actual parameter: value that is passed to procedure
- Local variable: temporary memory

## Recursion

- Procedure may (indirectly) call itself
- Consequence?

# Example: Procedures in C

```
#include <stio.h>

/* factorial function */
int fac( int num ) {
   if (num < 1)
      return 1;
   else
      return num * fac(num - 1) ;
}

int main() {
   int x = 10 ;

   int f = fac( x ) ;
   int x printf("%d! = %d\n", x, f);
   return 0;
}
```

formal parameter

recursive call

local variable

actual parameter

# Procedures in Tiger

```
/* factorial function */

let
   function fac( n: int ) : int =
      let
         var f := 1
      in
         if n < 1 then
            f := 1
         else
            f := (n * fac(n - 1) );
         f
      end
   var f := 0
   var x := 5
in
   f := fac( x )
end
```

formal parameter

local variable

recursive call

actual parameter

# Implementing Procedures with Stack and Stack Frames

## Stack

- temporary storage
- grows from high to low memory addresses
- starts at SS

## Stack frames

- return address
- local variables
- parameters
- stack base: BP
- stack top: SP

# Registers in x86 Family

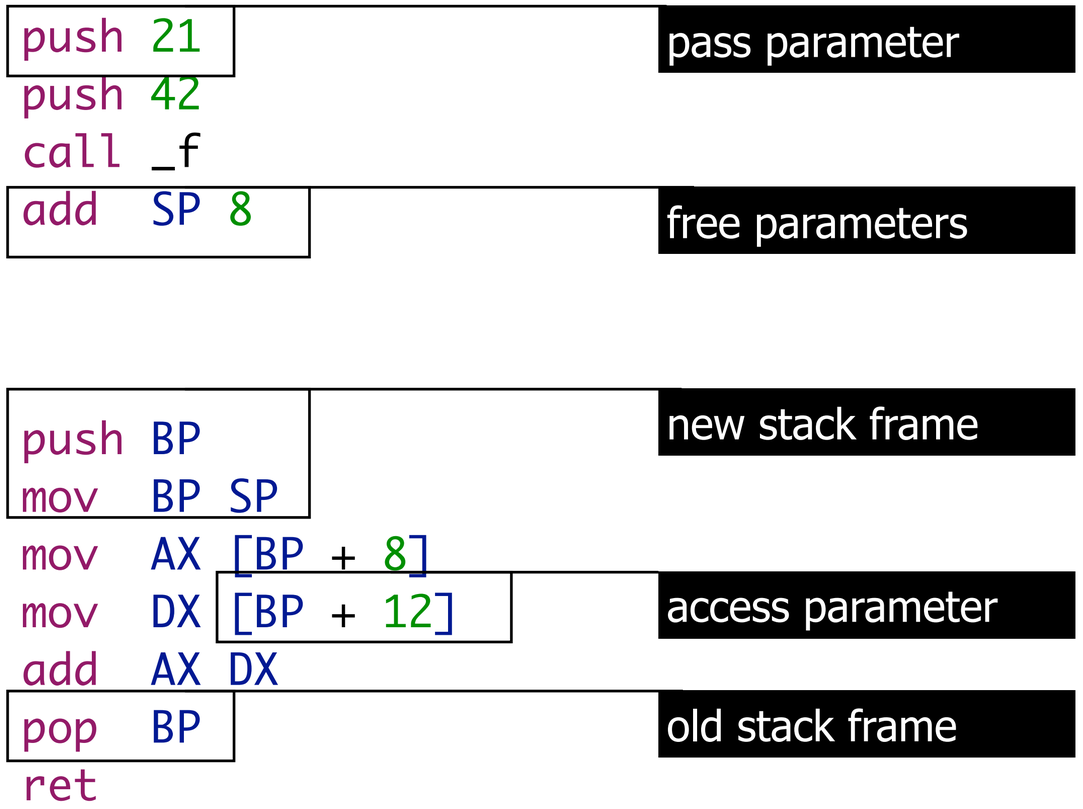
## General purpose registers

- accumulator AX - arithmetic operations
- counter CX - shift/rotate instructions, loops
- data DX - arithmetic operations, I/O
- base BX - pointer to data
- stack pointer SP, base pointer BP - top and base of stack
- source SI, destination DI - stream operations

## Special purpose registers

- segments SS, CS, DS, ES, FS, GS
- flags EFLAGS

# Example: Procedures in x86 Assembler



# Calling Conventions: CDECL

## Caller

- push parameters right-to-left on the stack
- clean-up stack after call

```
push 21
push 42
call _f
add  ESP 8
```

## Callee

- save old BP
- initialise new BP
- save registers
- return result in AX
- restore registers
- restore BP

```
push EBP
mov  EBP ESP
mov  EAX [EBP + 8]
mov  EDX [EBP + 12]
add  EAX EDX
pop  EBP
ret
```

# Calling Conventions: STDECL

## caller

- push parameters right-to-left on the stack

```
push 21
push 42
call _f@8
```

## callee

- save old BP
- initialise new BP
- save registers
- return result in AX
- restore registers
- restore BP

```
push EBP
mov  EBP ESP
mov  EAX [EBP + 8]
mov  EDX [EBP + 12]
add  EAX EDX
pop  EBP
ret  8
```

# Calling Conventions: FASTCALL

## Caller

- passes parameters in registers
- pushes additional parameters right-to-left on the stack

```
mov  ECX 21
mov  EDX 42
call @f@8
```

## Callee

- save old BP, initialise new BP
- save registers
- return result in AX
- restore registers
- restore BP
- cleans up the stack

```
push EBP
mov  EBP ESP
mov  EAX ECX
add  EAX EDX
pop  EBP
ret
```

# Calling Conventions

## Procedure declarations

- in principle: full freedom
- project constraints
- target platform constraints

## Procedure calls

- need to match procedure declarations

## Precompiled libraries

- avoid recompilation
- source code not always available

## Standardization

- compilers / high-level languages standardize use of calling conventions
- portable code: does not depend on particular calling convention

# Object-Oriented Languages

# Modularity: Objects & Messages

## Objects

- Generalization of records
- Identity
- State
- Behaviour

## Messages

- Objects send and receive messages
- Trigger behaviour
- Imperative realisation: method calls

# Modularity: Classes

## Classes

- Generalization of record types
- Characteristics of objects: attributes, fields, properties
- Behaviour of objects: methods, operations, features

## Encapsulation

- Interface exposure
- Hide attributes & methods
- Hide implementation

```
public class C {
    public int f1;
    private int f2;
    public void m1() { return; }
    private C m2(C c) { return c; }
}
```

# Inheritance vs Interfaces

## Inheritance

- Inherit attributes & methods
- Additional attributes & methods
- Override behaviour
- Nominal subtyping

## Interfaces

- Avoid multiple inheritance
- Interface: contract for attributes & methods
- Class: provide attributes & methods
- Nominal subtyping

```
public class C {
   public int f1;
   public void m1() {…}
   public void m2() {…}
}

public class D extends C {
   public int f2;
   public void m2() {…}
   public void m3() {…}
}

public interface I {
   public int f;
   public void m();
}

public class E implements I
{
   public int f;
   public void m() {…}
   public void m'() {…}
}
```

# Polymorphism

## Ad-hoc polymorphism

- Overloading
  - same method name, independent classes
  - same method name, same class, different parameter types
- Overriding
  - same method name, subclass, compatible types

## Universal polymorphism

- Subtype polymorphism
  - inheritance, interfaces
- Parametric polymorphism

# Static vs. Dynamic Dispatch

## Dispatch

- link method call to method

## Static dispatch

- type information at compile-time

## Dynamic dispatch

- type information at run-time
- single dispatch: one parameter
- multiple dispatch: more parameters

# Single Dispatch in Java

```
public class A {}
public class B extends A {}
public class C extends B {}

public class D {
   public A m(A a) { System.out.println("D.m(A a)"); return a; }
   public A m(B b) { System.out.println("D.m(B b)"); return b; }
}

public class E extends D {
   public A m(A a) { System.out.println("E.m(A a)"); return a; }
   public B m(B b) { System.out.println("E.m(B b)"); return b; }
}
```

```
A a = new A(); B b  = new B(); C c  = new C(); D d = new D(); E e  = new E();
               A ab = b;       A ac = c;                      D de = e;

d. m(a); d. m(b); d. m(ab); d. m(c); d. m(ac);
e. m(a); e. m(b); e. m(ab); e. m(c); e. m(ac);
de.m(a); de.m(b); de.m(ab); de.m(c); de.m(ac);
```

# Overriding

## Methods

- parameter types
- return type

## Covariance

- method in subclass
- return type: subtype of original return type

## Contravariance

- method in subclass
- parameter types: supertypes of original parameter types

# Overloading vs Overriding

```
public class F {
   public A m(B b) { System.out.println("F.m(B b)"); return b; }
}

public class G extends F {
   public A m(A a) { System.out.println("G.m(A a)"); return a; }
}

public class H extends F {
   public B m(B b) { System.out.println("H.m(B b)"); return b; }
}
```

```
A a = new A(); B b  = new B(); F f = new F(); G g = new G(); H h = new H();
               A ab = b;

f.m(b);
g.m(a); g.m(b); g.m(ab);
h.m(a); h.m(b); h.m(ab);
```

# Invariance

```
public class X {
   public A a;
   public A    getA()    { return a ; }
   public void setA(A a) { this.a = a ; }
}

public class Y extends X {
   public B a;
   public B    getA()    { return a ; }
   public void setA(B a) { this.a = a ; }
}
```

```
A a = new A(); B b  = new B(); X y = new Y();

y.getA(); y.setA(b); y.setA(a); y.getA();

String[] s = new String[3] ; Object[] o = s ; o[1] = new A();
```

# Summary: Abstractions

# Abstractions for Memory and Control

## Imperative languages

- subroutines, routines, procedures, functions, methods
- scoping: local variables
- declarations with parameters (formal parameters)
- calls with arguments (actual parameters)
- pass by value, pass by reference

## Machine code

- jumps: call and return
- call stack: return address, parameters, private data
- procedure prologue and epilogue

# Imperative vs Object-Oriented

## Imperative languages

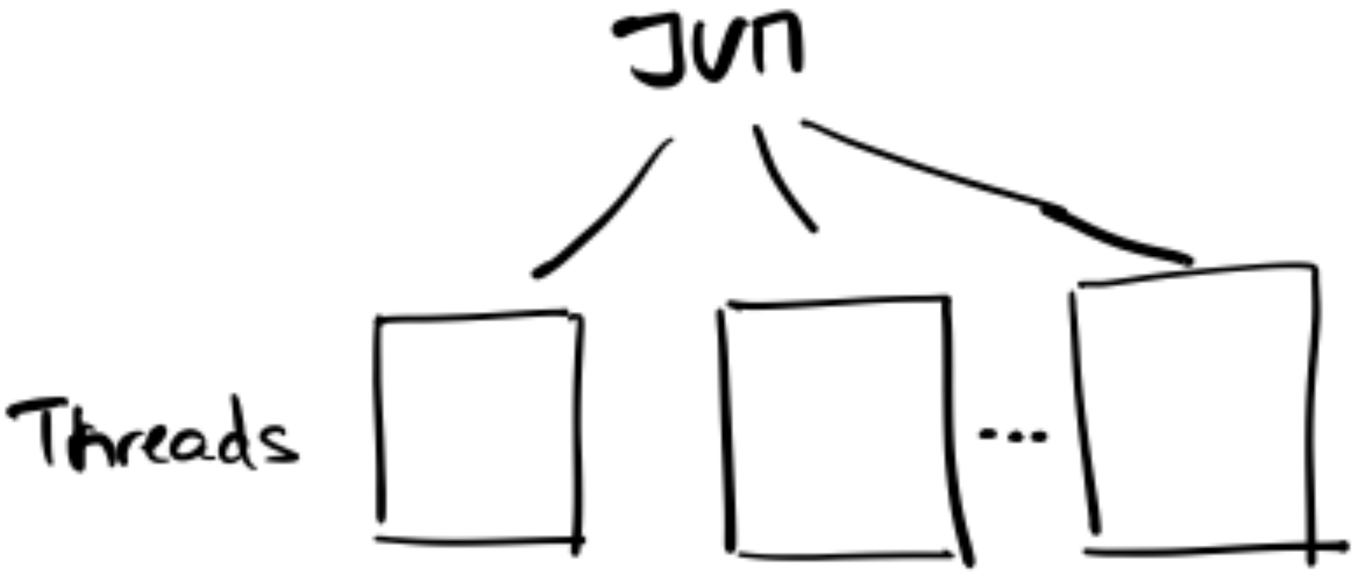
- state & statements
- abstraction over machine code
- control flow & procedures
- types

## Object-oriented languages

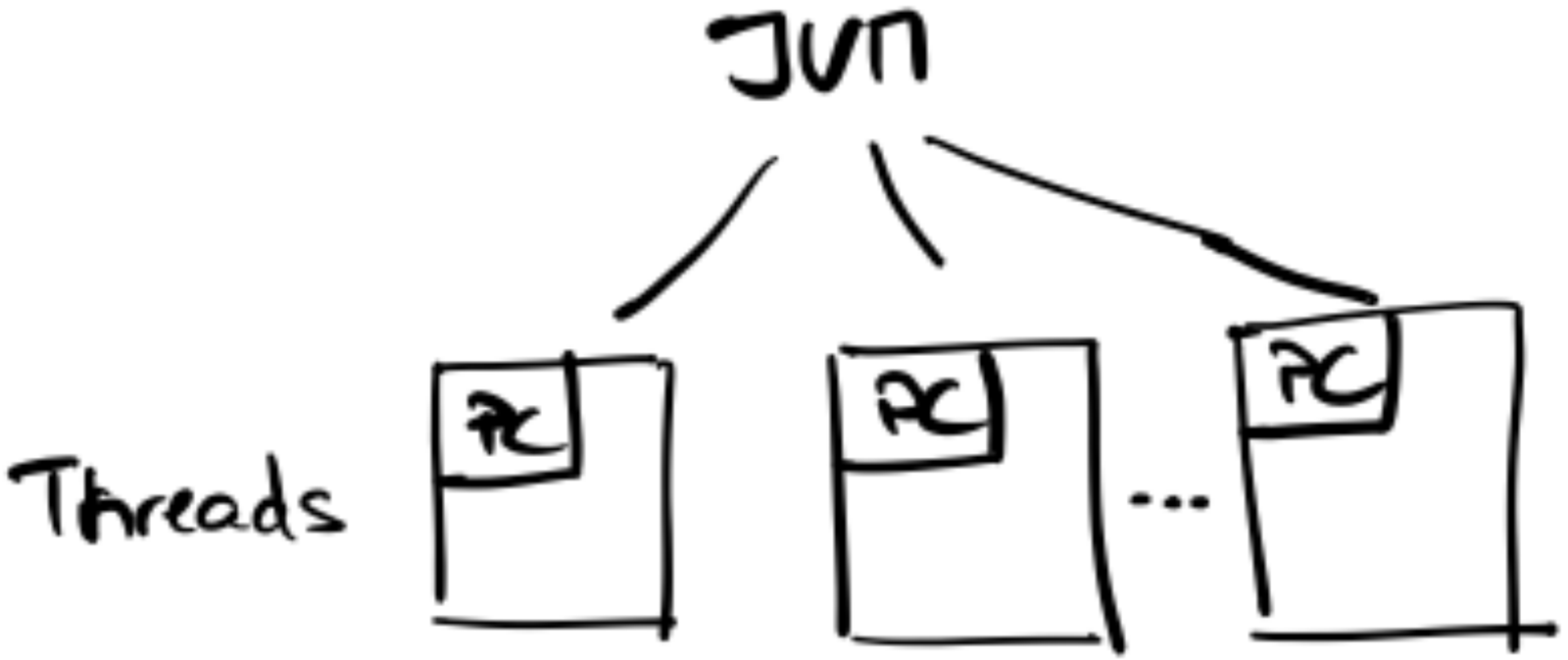
- objects & messages
- classes
- inheritance
- types

# Java Virtual Machine: Architecture

# JVM Architecture

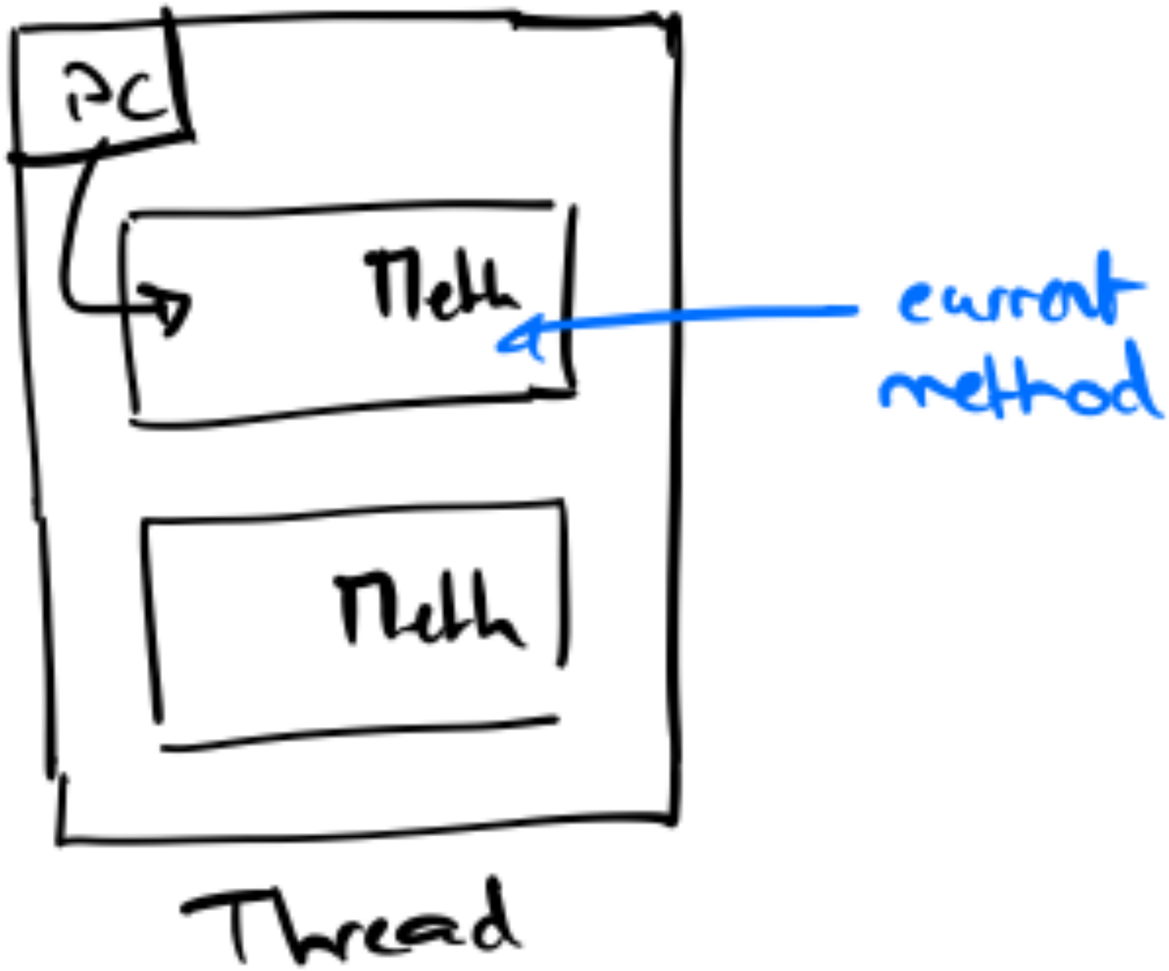


# JVM Architecture: Threads

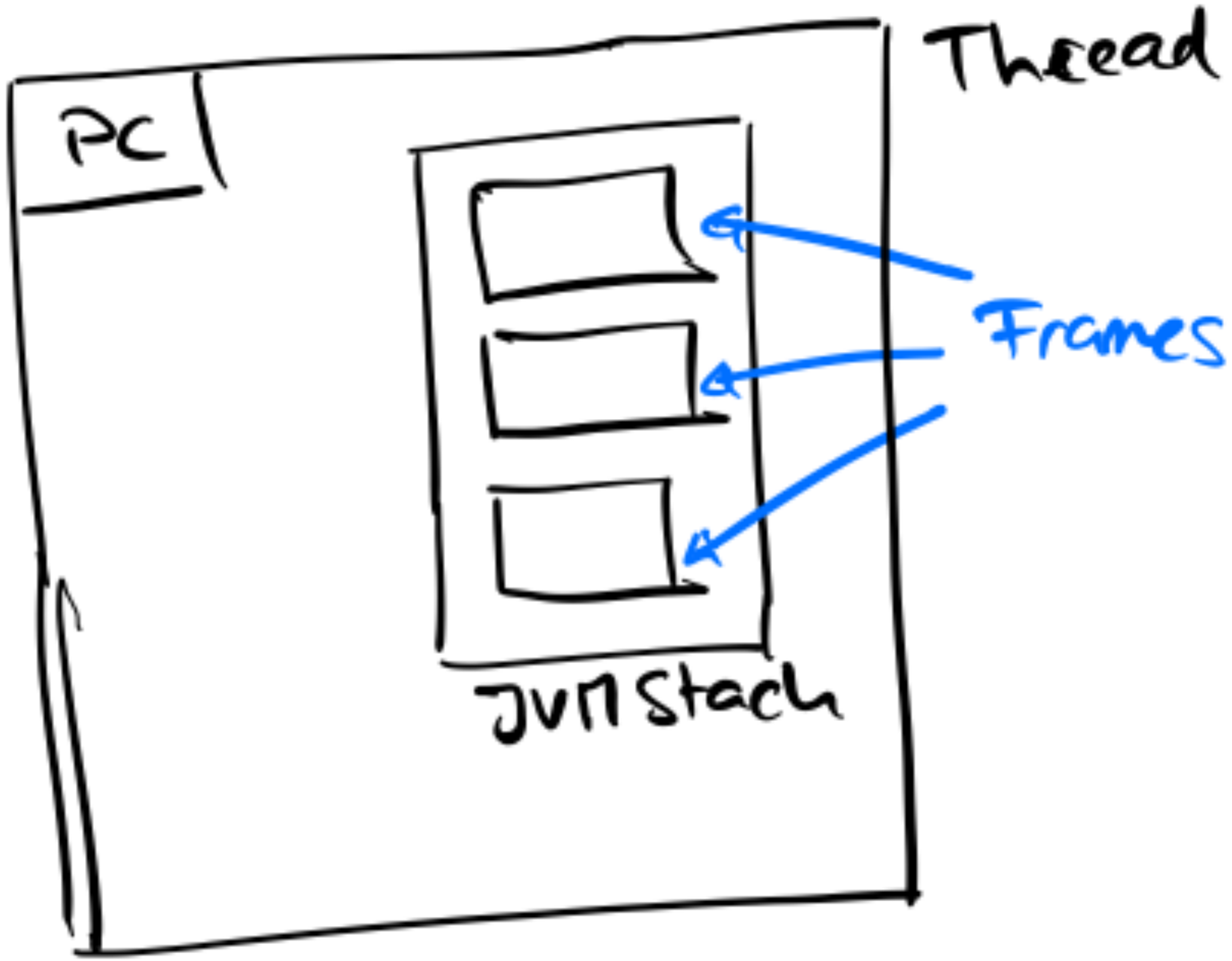



Each thread has its own program counter (PC)

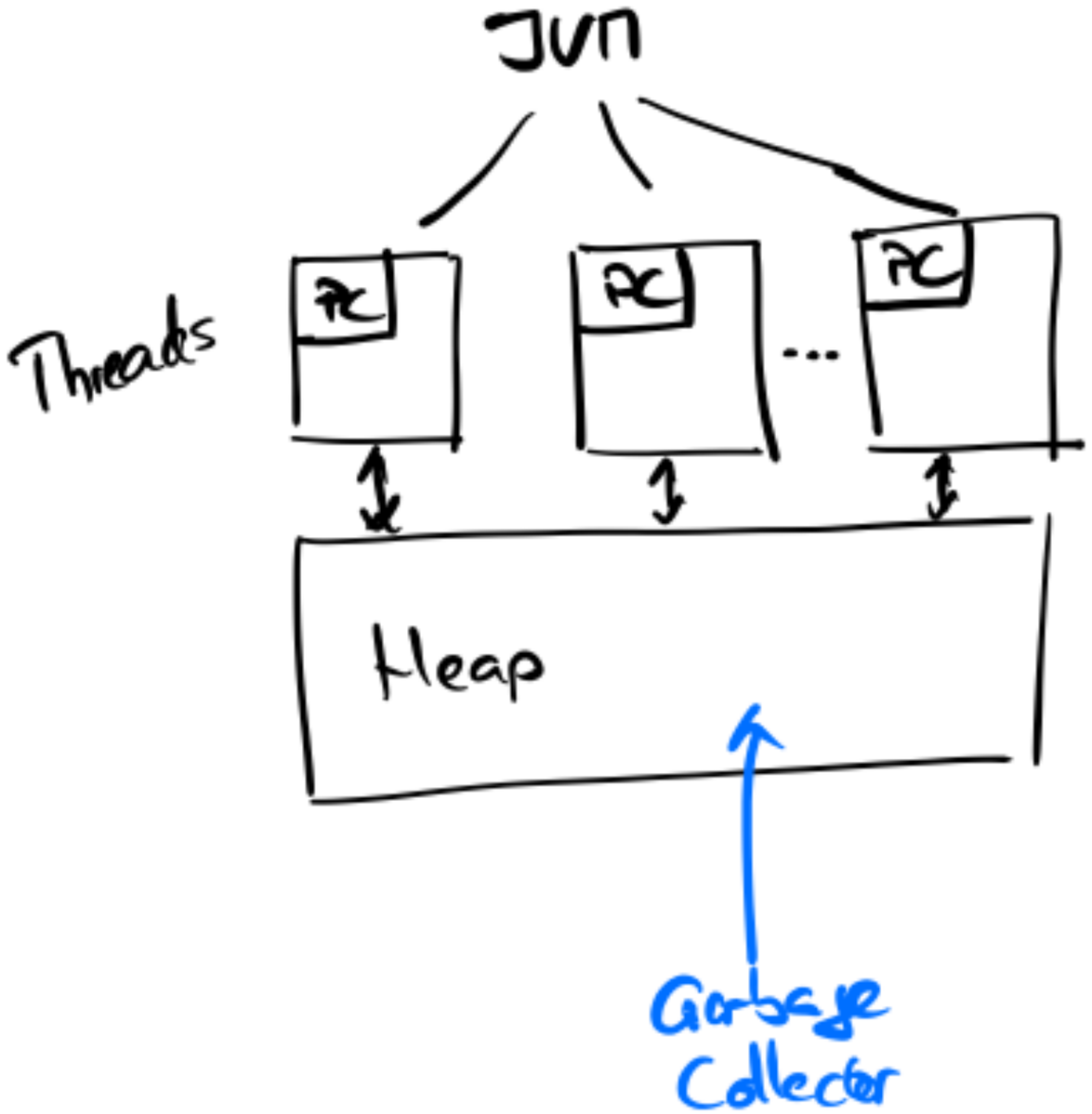
# JVM Architecture: Thread



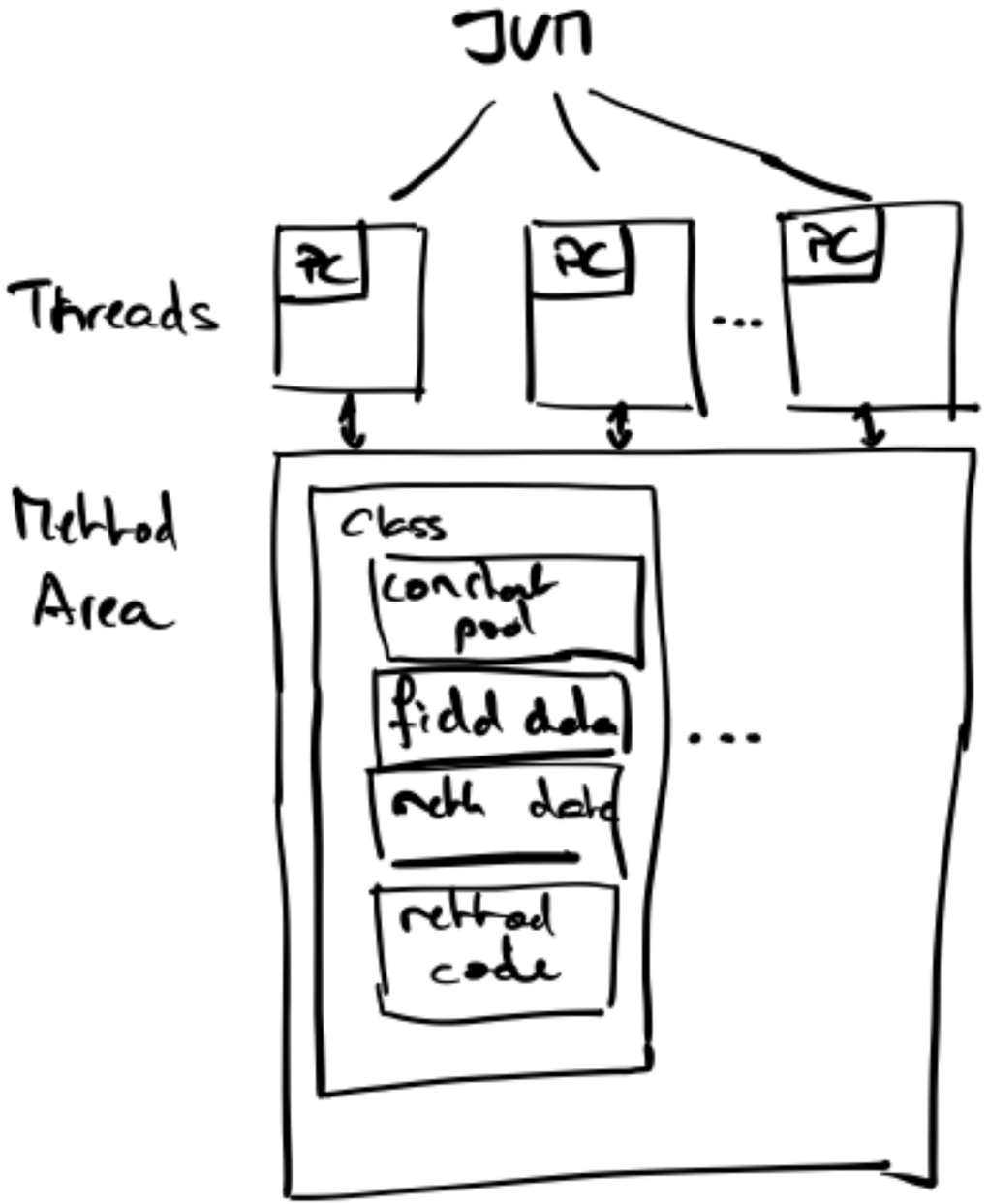
# JVM Architecture: Stack



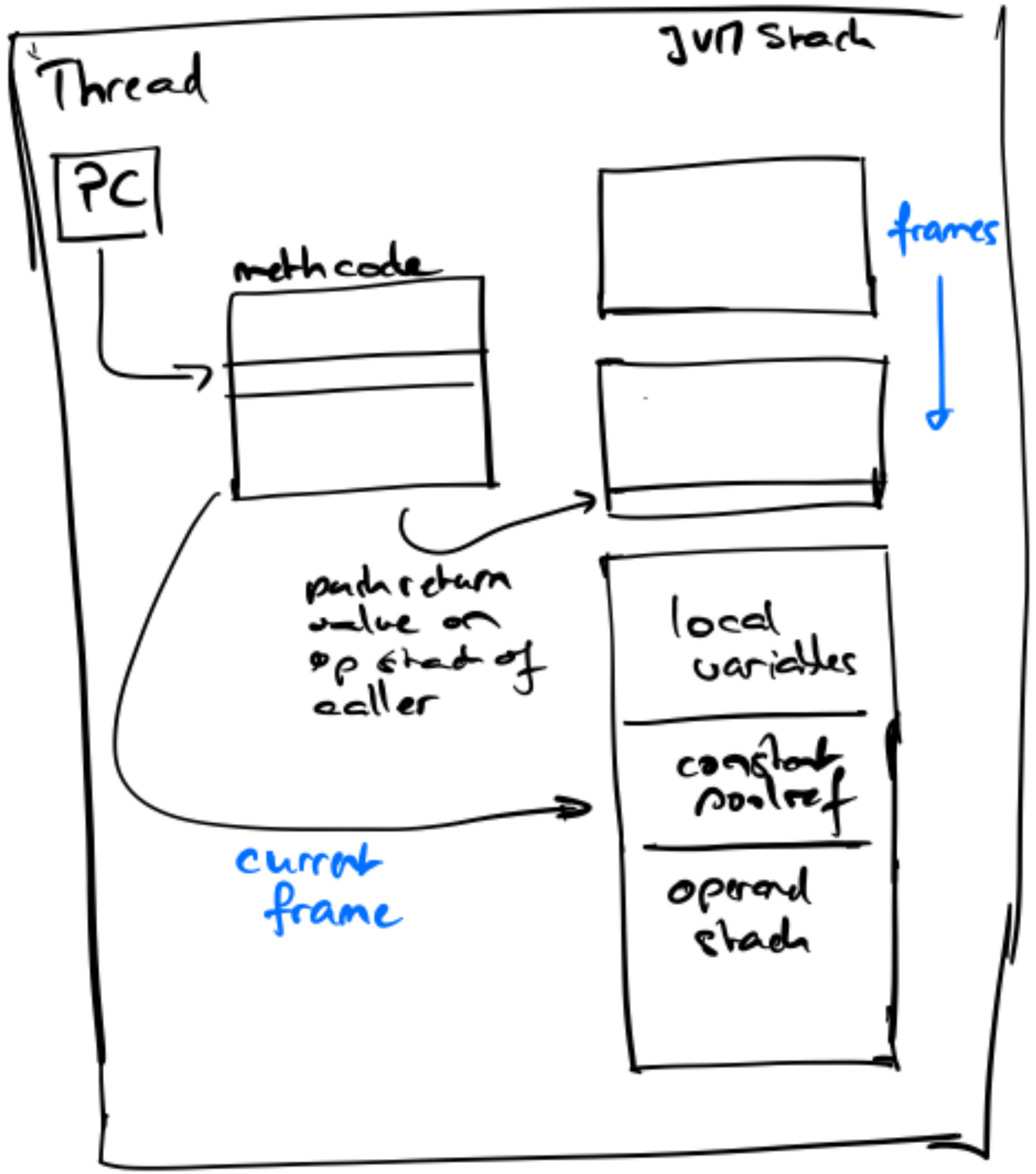
# JVM Architecture: Heap



# JVM Architecture: Method Area



# JVM Architecture: JVM Stack



# JVM: Control Flow

# Bytecode Instructions: Goto

| method area | | |
|---|---|---|
| pc: 00 | | |
| 00 | A7 | goto |
| 01 | 00 | |
| 02 | 04 | 04 |
| 03 | 00 | nop |
| 04 | A7 | goto |
| 05 | FF | |
| 06 | FF | 03 |

# Bytecode Instructions: Goto

| method area | | |
|---|---|---|
| pc: 04 | | |
| 00 | A7 | goto |
| 01 | 00 | |
| 02 | 04 | 04 |
| 03 | 00 | nop |
| 04 | A7 | goto |
| 05 | FF | |
| 06 | FF | 03 |

# Bytecode Instructions: Goto

| method area | | |
|---|---|---|
| pc: 03 | | |
| 00 | A7 | goto |
| 01 | 00 | |
| 02 | 04 | 04 |
| 03 | 00 | nop |
| 04 | A7 | goto |
| 05 | FF | |
| 06 | FF | 03 |

# Bytecode Instructions: Goto

| method area | | |
|---|---|---|
| pc: 04 | | |
| 00 | A7 | goto |
| 01 | 00 | |
| 02 | 04 | 04 |
| 03 | 00 | nop |
| 04 | A7 | goto |
| 05 | FF | |
| 06 | FF | 03 |

# JVM: Operand Stack

# Operand Stack

| method area | | |
|---|---|---|
| pc: 00 | | |
| 00 | 04 | iconst_1 |
| 01 | 05 | iconst_2 |
| 02 | 10 | bipush |
| 03 | 2A | |
| 04 | 11 | sipush |
| 05 | 43 | |
| 06 | 03 | |

| stack |
|---|
| optop: 00 |
| 00 |
| 01 |
| 02 |
| 03 |
| 04 |
| 05 |
| 06 |

# Operand Stack

| method area | |
|---|---|
| pc: 01 | |
| 00 | 04 iconst_1 |
| 01 | 05 iconst_2 |
| 02 | 10 bipush |
| 03 | 2A |
| 04 | 11 sipush |
| 05 | 43 |
| 06 | 03 |

| stack | |
|---|---|
| optop: 01 | |
| 00 | 0000 0001 |
| 01 | |
| 02 | |
| 03 | |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | |
|---|---|
| pc: 02 | |
| 00 | 04 iconst_1 |
| 01 | 05 iconst_2 |
| 02 | 10 bipush |
| 03 | 2A |
| 04 | 11 sipush |
| 05 | 43 |
| 06 | 03 |

| stack | |
|---|---|
| optop: 02 | |
| 00 | 0000 0001 |
| 01 | 0000 0002 |
| 02 | |
| 03 | |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | | |
|---|---|---|
| pc: 04 | | |
| 00 | 04 | iconst_1 |
| 01 | 05 | iconst_2 |
| 02 | 10 | bipush |
| 03 | 2A | |
| 04 | 11 | sipush |
| 05 | 43 | |
| 06 | 03 | |

| stack | |
|---|---|
| optop: 03 | |
| 00 | 0000 0001 |
| 01 | 0000 0002 |
| 02 | 0000 002A |
| 03 | |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | |
|---|---|
| pc: 07 | |
| 00 | 04 iconst_1 |
| 01 | 05 iconst_2 |
| 02 | 10 bipush |
| 03 | 2A |
| 04 | 11 sipush |
| 05 | 43 |
| 06 | 03 |

| stack | |
|---|---|
| optop: 04 | |
| 00 | 0000 0001 |
| 01 | 0000 0002 |
| 02 | 0000 002A |
| 03 | 0000 4303 |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | | |
|---|---|---|
| pc: 07 | | |
| 07 | 60 | iadd |
| 08 | 68 | imul |
| 09 | 5F | swap |
| 0A | 64 | isub |
| 0B | 9A | ifne |
| 0C | FF | |
| 0D | F5 | 00 |

| stack | |
|---|---|
| optop: 04 | |
| 00 | 0000 0001 |
| 01 | 0000 0002 |
| 02 | 0000 002A |
| 03 | 0000 4303 |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | | |
|---|---|---|
| pc: 08 | | |
| 07 | 60 | iadd |
| 08 | 68 | imul |
| 09 | 5F | swap |
| 0A | 64 | isub |
| 0B | 9A | ifne |
| 0C | FF | |
| 0D | F5 | 00 |

| stack | |
|---|---|
| optop: 03 | |
| 00 | 0000 0001 |
| 01 | 0000 0002 |
| 02 | 0000 432D |
| 03 | |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | |
|---|---|
| pc: 09 | |
| 07 | 60 iadd |
| 08 | 68 imul |
| 09 | 5F swap |
| 0A | 64 isub |
| 0B | 9A ifne |
| 0C | FF |
| 0D | F5 00 |

| stack | |
|---|---|
| optop: 02 | |
| 00 | 0000 0001 |
| 01 | 0000 865A |
| 02 | |
| 03 | |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | | |
|---|---|---|
| pc: 0A | | |
| 07 | 60 | iadd |
| 08 | 68 | imul |
| 09 | 5F | swap |
| 0A | 64 | isub |
| 0B | 9A | ifne |
| 0C | FF | |
| 0D | F5 | 00 |

| stack | |
|---|---|
| optop: 02 | |
| 00 | 0000 865A |
| 01 | 0000 0001 |
| 02 | |
| 03 | |
| 04 | |
| 05 | |
| 06 | |

# Operand Stack

| method area | | |
|---|---|---|
| pc: 0B | | |
| 07 | 60 | iadd |
| 08 | 68 | imul |
| 09 | 5F | swap |
| 0A | 64 | isub |
| 0B | 9A | ifne |
| 0C | FF | |
| 0D | F5 | 00 |

| stack | |
|---|---|
| optop: 01 | |
| 00 | 0000 8659 |
| 01 | |
| 02 | |
| 03 | |
| 04 | |
| 05 | |
| 06 | |

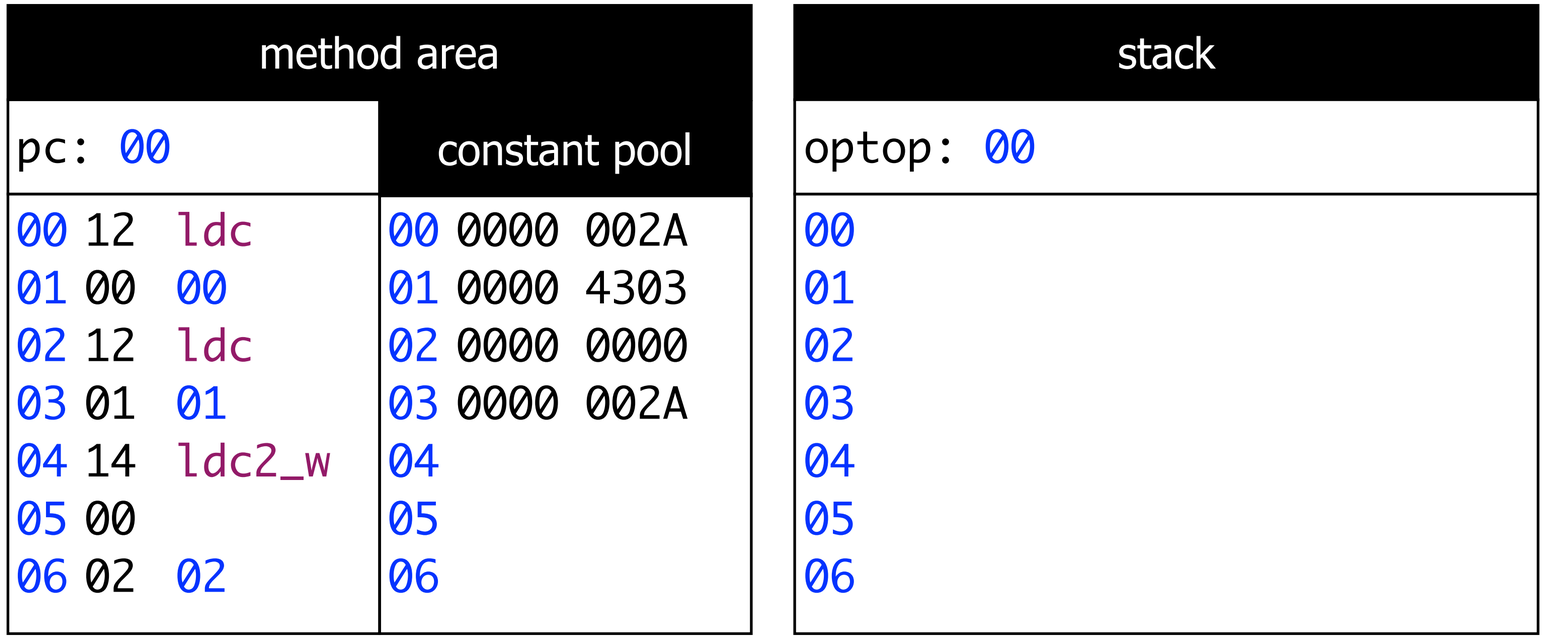
# Operand Stack

| method area | | |
|---|---|---|
| pc: 00 | | |
| 00 | 04 | iconst_1 |
| 01 | 05 | iconst_2 |
| 02 | 10 | bipush |
| 03 | 2A | |
| 04 | 11 | sipush |
| 05 | 43 | |
| 06 | 03 | |

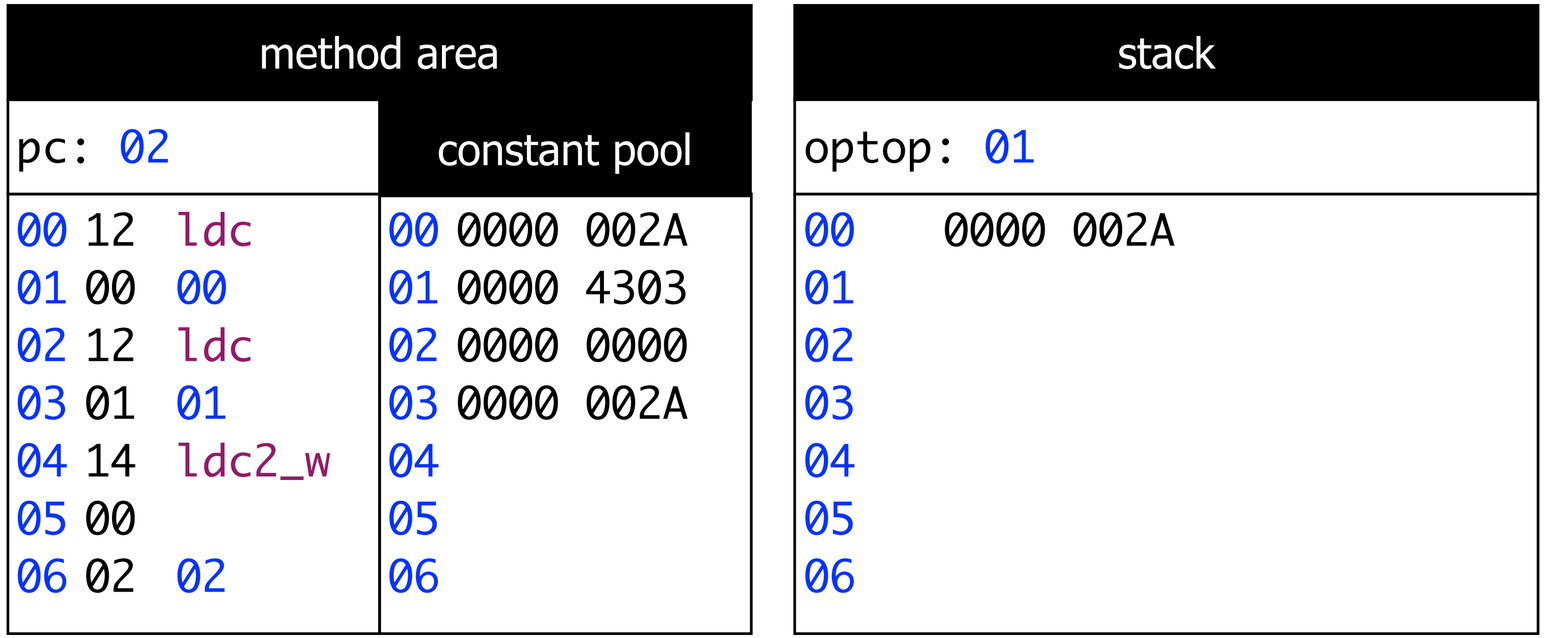
| stack |
|---|
| optop: 00 |
| 00 |
| 01 |
| 02 |
| 03 |
| 04 |
| 05 |
| 06 |

# JVM: Constant Pool

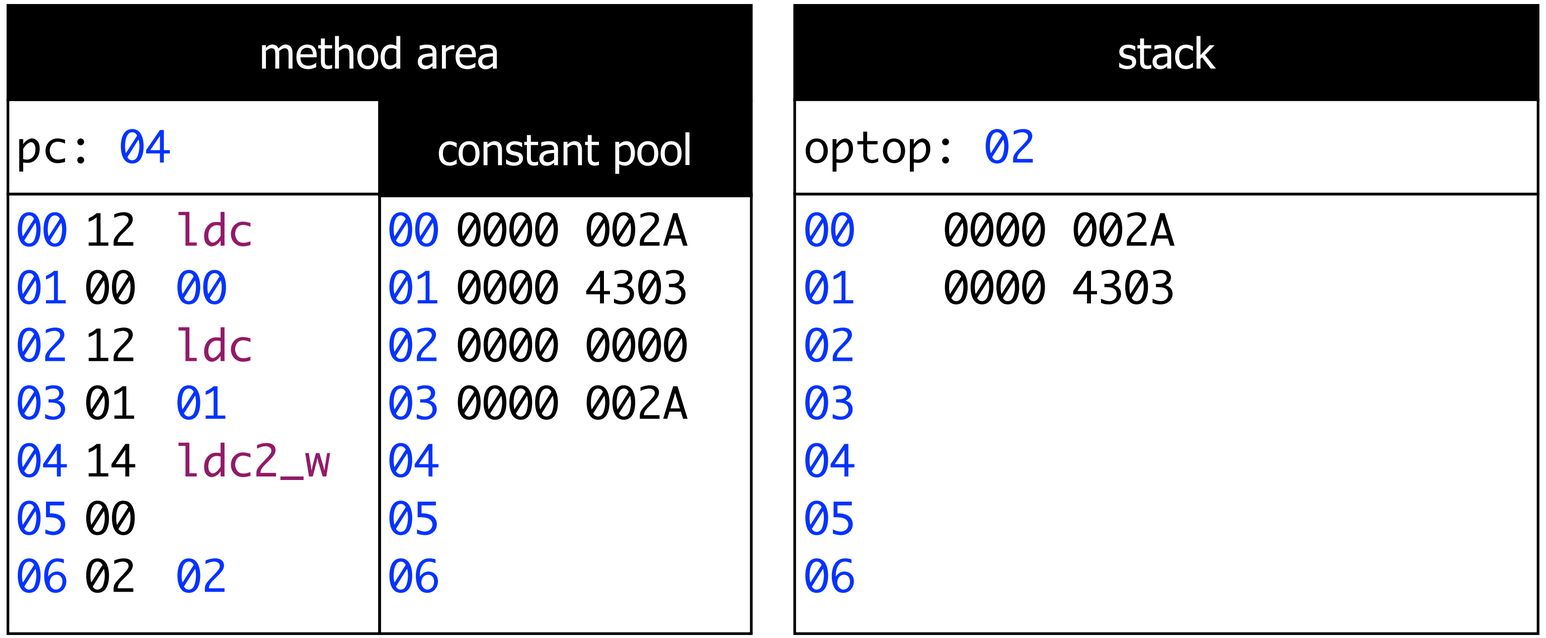
# Constant Pool



# Constant Pool



# Constant Pool



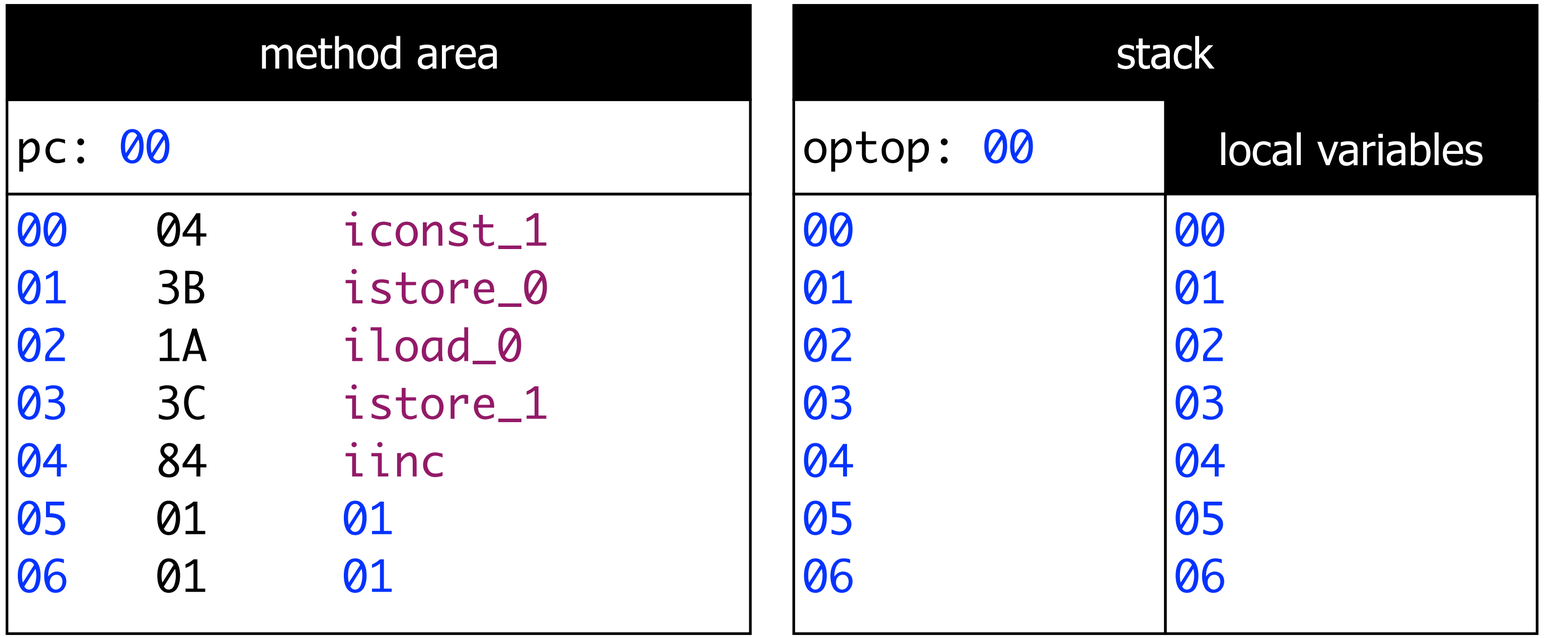
# Constant Pool

| method area | |
|---|---|
| pc: 07 | constant pool |
| 00 12 ldc | 00 0000 002A |
| 01 00 00 | 01 0000 4303 |
| 02 12 ldc | 02 0000 0000 |
| 03 01 01 | 03 0000 002A |
| 04 14 ldc2_w | 04 |
| 05 00 | 05 |
| 06 02 02 | 06 |

| stack |
|---|
| optop: 04 |
| 00 0000 002A |
| 01 0000 4303 |
| 02 0000 0000 |
| 03 0000 002A |
| 04 |
| 05 |
| 06 |

# JVM: Local Variables

# Local Variables



# Local Variables

| method area | |
|---|---|
| pc: 01 | |
| 00 | 04 iconst_1 |
| 01 | 3B istore_0 |
| 02 | 1A iload_0 |
| 03 | 3C istore_1 |
| 04 | 84 iinc |
| 05 | 01 01 |
| 06 | 01 01 |

| stack | |
|---|---|
| optop: 01 | local variables |
| 00 0000 0001 | 00 |
| 01 | 01 |
| 02 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

# Local Variables

| method area | | |
|---|---|---|
| pc: 02 | | |
| 00 | 04 | iconst_1 |
| 01 | 3B | istore_0 |
| 02 | 1A | iload_0 |
| 03 | 3C | istore_1 |
| 04 | 84 | iinc |
| 05 | 01 | 01 |
| 06 | 01 | 01 |

| stack | |
|---|---|
| optop: 00 | local variables |
| 00 | 00 0000 0001 |
| 01 | 01 |
| 02 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

# Local Variables

| method area | |
|---|---|
| pc: 03 | |
| 00 | 04 iconst_1 |
| 01 | 3B istore_0 |
| 02 | 1A iload_0 |
| 03 | 3C istore_1 |
| 04 | 84 iinc |
| 05 | 01 01 |
| 06 | 01 01 |

| stack | |
|---|---|
| optop: 01 | local variables |
| 00 0000 0001 | 00 0000 0001 |
| 01 | 01 |
| 02 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

# Local Variables

| method area | |
|---|---|
| pc: 04 | |
| 00 | 04 iconst_1 |
| 01 | 3B istore_0 |
| 02 | 1A iload_0 |
| 03 | 3C istore_1 |
| 04 | 84 iinc |
| 05 | 01 01 |
| 06 | 01 01 |

| stack | |
|---|---|
| optop: 00 | local variables |
| 00 | 00 0000 0001 |
| 01 | 01 0000 0001 |
| 02 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

# Local Variables

| method area | |
|---|---|
| pc: 07 | |
| 00 | 04 iconst_1 |
| 01 | 3B istore_0 |
| 02 | 1A iload_0 |
| 03 | 3C istore_1 |
| 04 | 84 iinc |
| 05 | 01 01 |
| 06 | 01 01 |

| stack | |
|---|---|
| optop: 00 | local variables |
| 00 | 00 0000 0001 |
| 01 | 01 0000 0002 |
| 02 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

# JVM: Heap

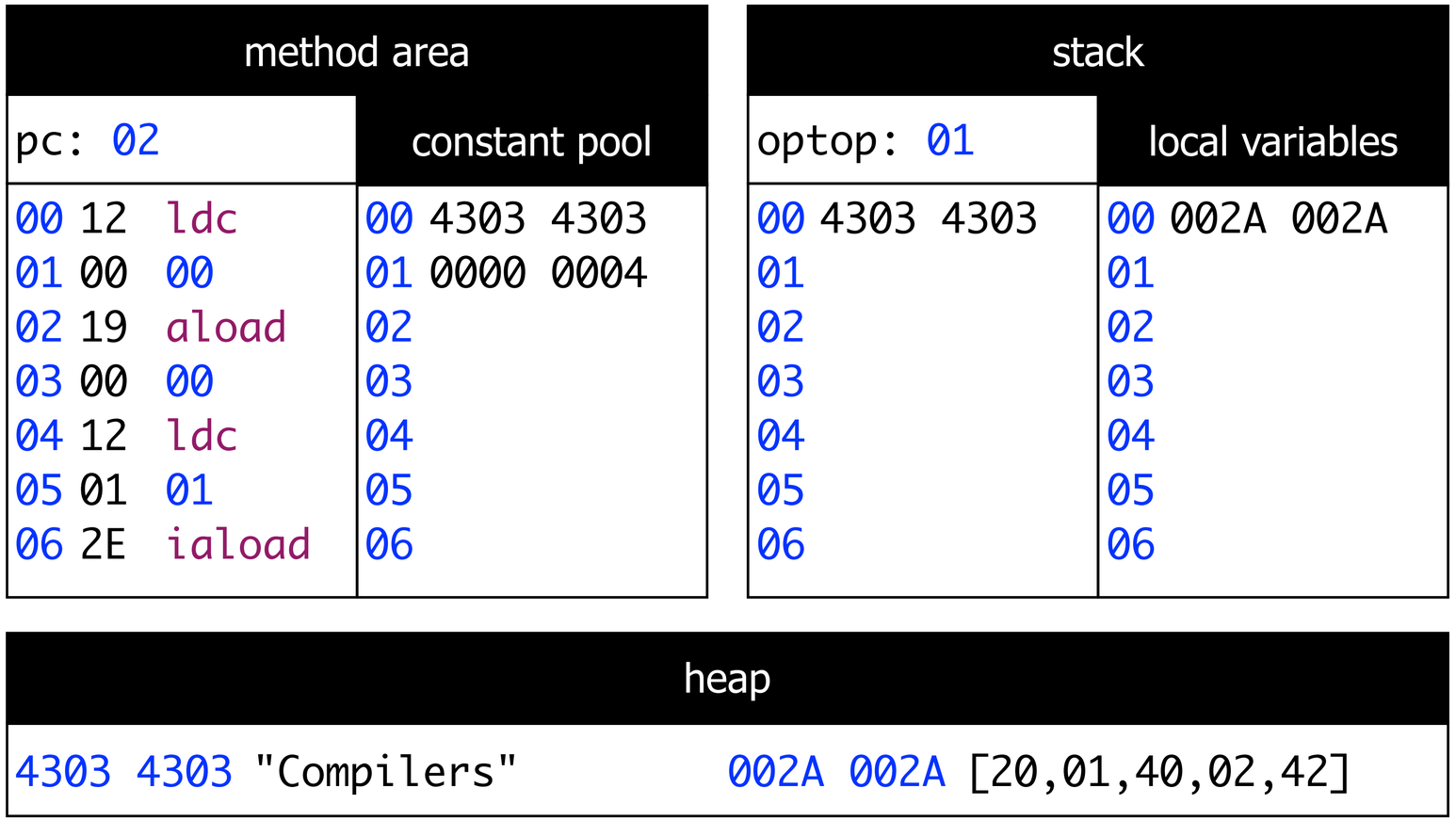
# Heap

| method area | |
|---|---|
| pc: 00 | constant pool |
| 00 12 ldc | 00 4303 4303 |
| 01 00 00 | 01 0000 0004 |
| 02 19 aload | 02 |
| 03 00 00 | 03 |
| 04 12 ldc | 04 |
| 05 01 01 | 05 |
| 06 2E iaload | 06 |

| stack | |
|---|---|
| optop: 00 | local variables |
| 00 | 00 002A 002A |
| 01 | 01 |
| 02 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

| heap |
|---|
| 4303 4303 "Compilers" 002A 002A [20,01,40,02,42] |

# Heap


method area
pc: 02
00 12 ldc
01 00 00
02 19 aload
03 00 00
04 12 ldc
05 01 01
06 2E iaload
constant pool
00 4303 4303
01 0000 0004
02
03
04
05
06
stack
optop: 01
00 4303 4303
01
02
03
04
05
06
local variables
00 002A 002A
01
02
03
04
05
06
heap
4303 4303 "Compilers"
002A 002A [20,01,40,02,42]

# Heap

**method area**

| pc: 04 | constant pool |
|---|---|
| 00 12 ldc | 00 4303 4303 |
| 01 00 00 | 01 0000 0004 |
| 02 19 aload | 02 |
| 03 00 00 | 03 |
| 04 12 ldc | 04 |
| 05 01 01 | 05 |
| 06 2E iaload | 06 |

**stack**

| optop: 02 | local variables |
|---|---|
| 00 4303 4303 | 00 002A 002A |
| 01 002A 002A | 01 |
| 02 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

**heap**

4303 4303 "Compilers"  002A 002A [20,01,40,02,42]

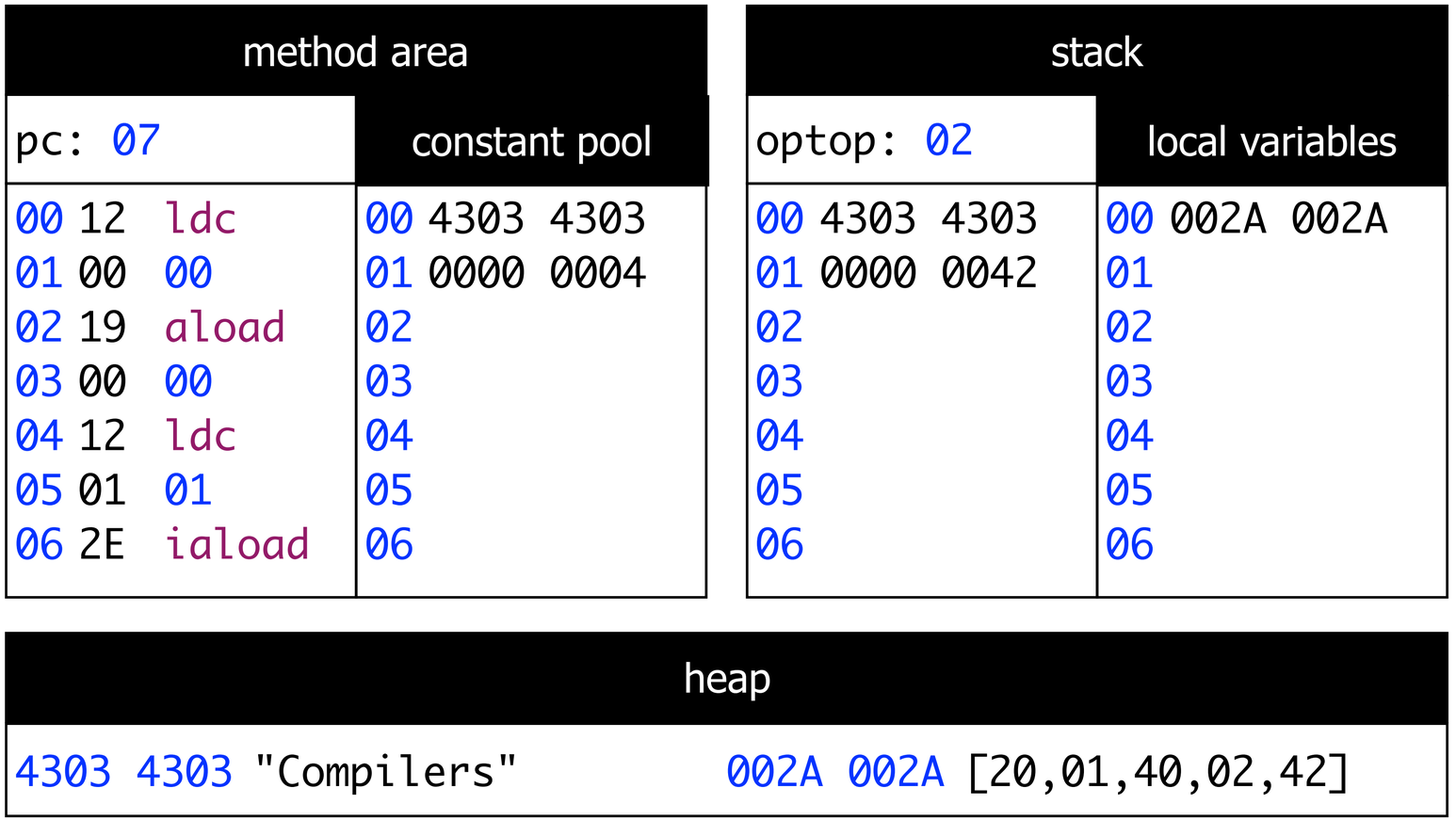
# Heap

| method area | |
|---|---|
| pc: 06 | constant pool |
| 00 12 ldc | 00 4303 4303 |
| 01 00 00 | 01 0000 0004 |
| 02 19 aload | 02 |
| 03 00 00 | 03 |
| 04 12 ldc | 04 |
| 05 01 01 | 05 |
| 06 2E iaload | 06 |

| stack | |
|---|---|
| optop: 03 | local variables |
| 00 4303 4303 | 00 002A 002A |
| 01 002A 002A | 01 |
| 02 0000 0004 | 02 |
| 03 | 03 |
| 04 | 04 |
| 05 | 05 |
| 06 | 06 |

| heap |
|---|
| 4303 4303 "Compilers" 002A 002A [20,01,40,02,42] |

# Heap


method area
pc: 07
00 12 ldc
01 00 00
02 19 aload
03 00 00
04 12 ldc
05 01 01
06 2E iaload
constant pool
00 4303 4303
01 0000 0004
02
03
04
05
06
stack
optop: 02
00 4303 4303
01 0000 0042
02
03
04
05
06
local variables
00 002A 002A
01
02
03
04
05
06
heap
4303 4303 "Compilers"
002A 002A [20,01,40,02,42]

# JVM: Stack Frames

# Static vs. Dynamic Dispatch

## Dispatch

- link method call to method

## Static dispatch

- based on type information at compile-time

## Dynamic dispatch

- based on type information at run-time
- single dispatch: one parameter
- multiple dispatch: more parameters

# Example: Static Call

```
function fac(n: int): int =
   if
      n = 0
   then
      1
   else
      n * fac(n - 1)
```

```
.class public Exp

   .method public static fac(I)I

            iload 1
            ifne else

            iconst_1
            ireturn

      else: iload 1
            dup
            iconst_1
            isub
            invokestatic Exp/fac(I)I
            imul
            ireturn
   .end method
```

# Example: Dynamic Call

```
function fac(n: int): int =
   if
      n = 0
   then
      1
   else
      n * fac(n - 1)
```

```
.class public Exp

   .method public fac(I)I

            iload 1
            ifne else

            iconst_1
            ireturn

      else: iload 0
            iload 1
            dup
            iconst_1
            isub
            invokevirtual Exp/fac(I)I
            imul
            ireturn
   .end method
```

# Code Pattern: Dynamic Method Call

## Caller

- push reference to receiver object
- push parameters left-to-right
- call method

## Virtual machine on call

- allocate space (frame data, operand stack, local variables)
- store frame data (data pointer, return address, exception table)
- store parameters as local variables
- dynamic dispatch
- point pc to method code

# Code Pattern: Return from Method Call

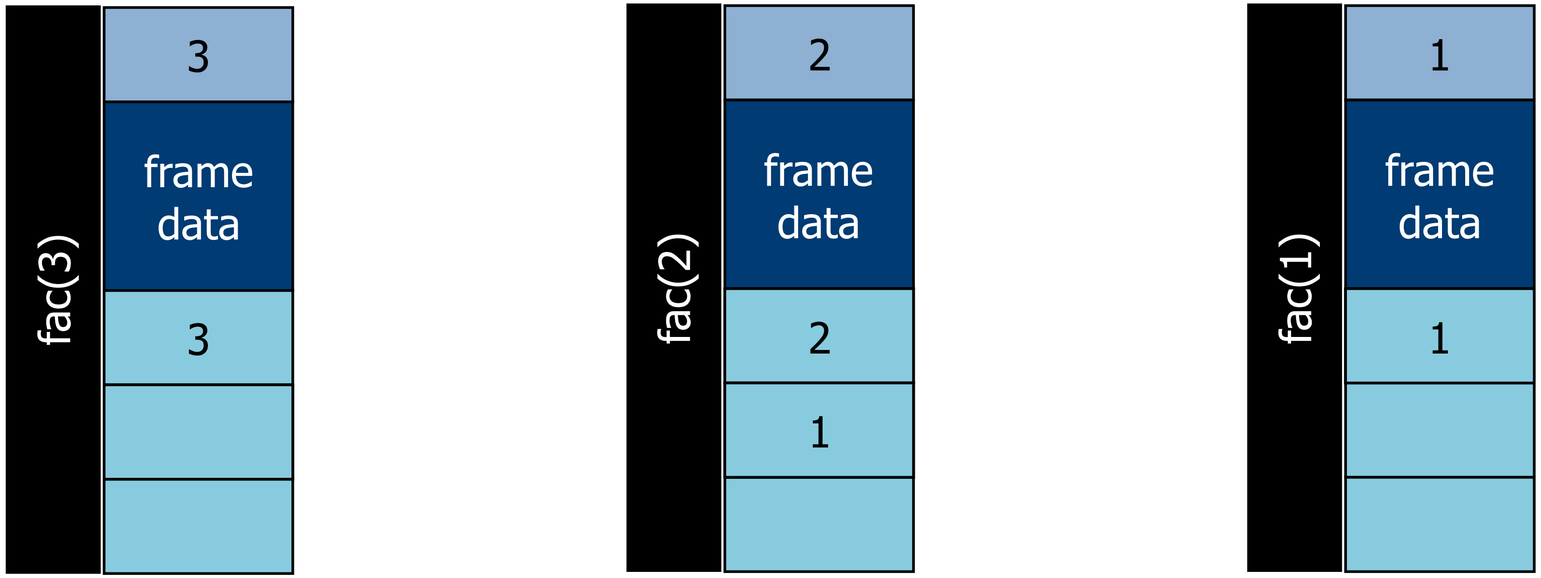
## Callee

- parameters in local variables
- leave result on operand stack
- return to caller

## Virtual machine on return

- push result on caller's operand stack
- point pc to return address
- destroy frame

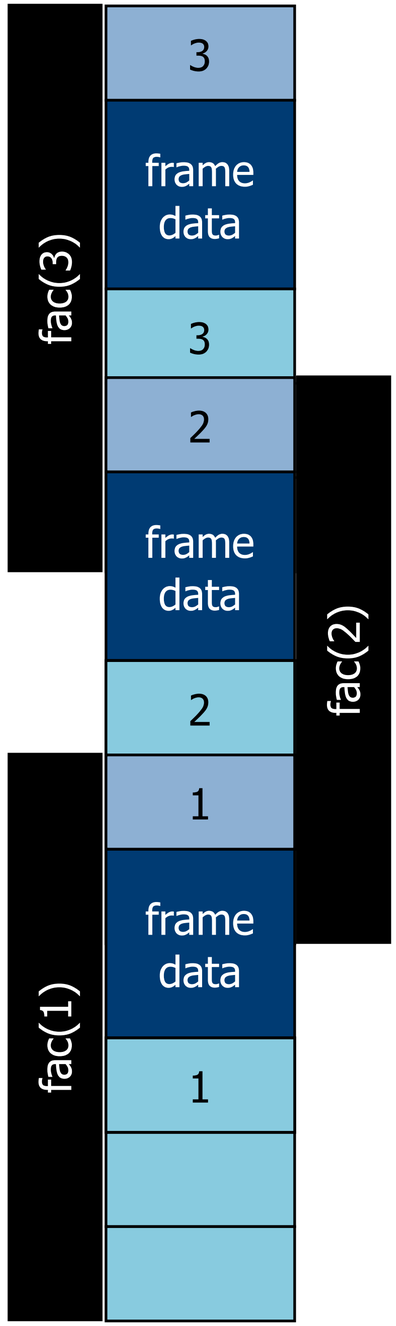
# Implementation: Heap-Based



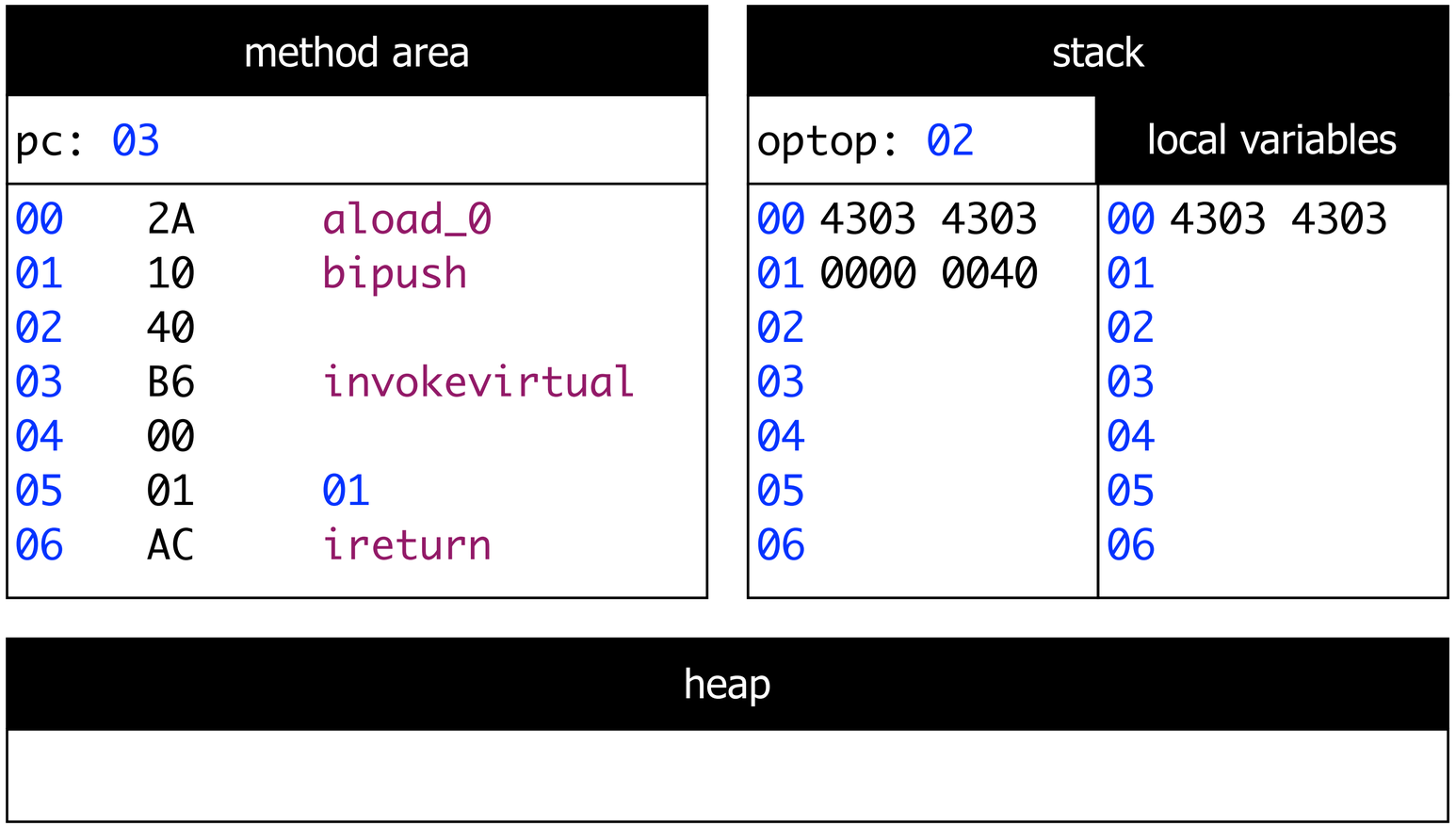

```
function fac(n: int): int =
   if
      n = 0
   then
      1
   else
      n * fac(n - 1)
```

# Implementation: Stack-Based

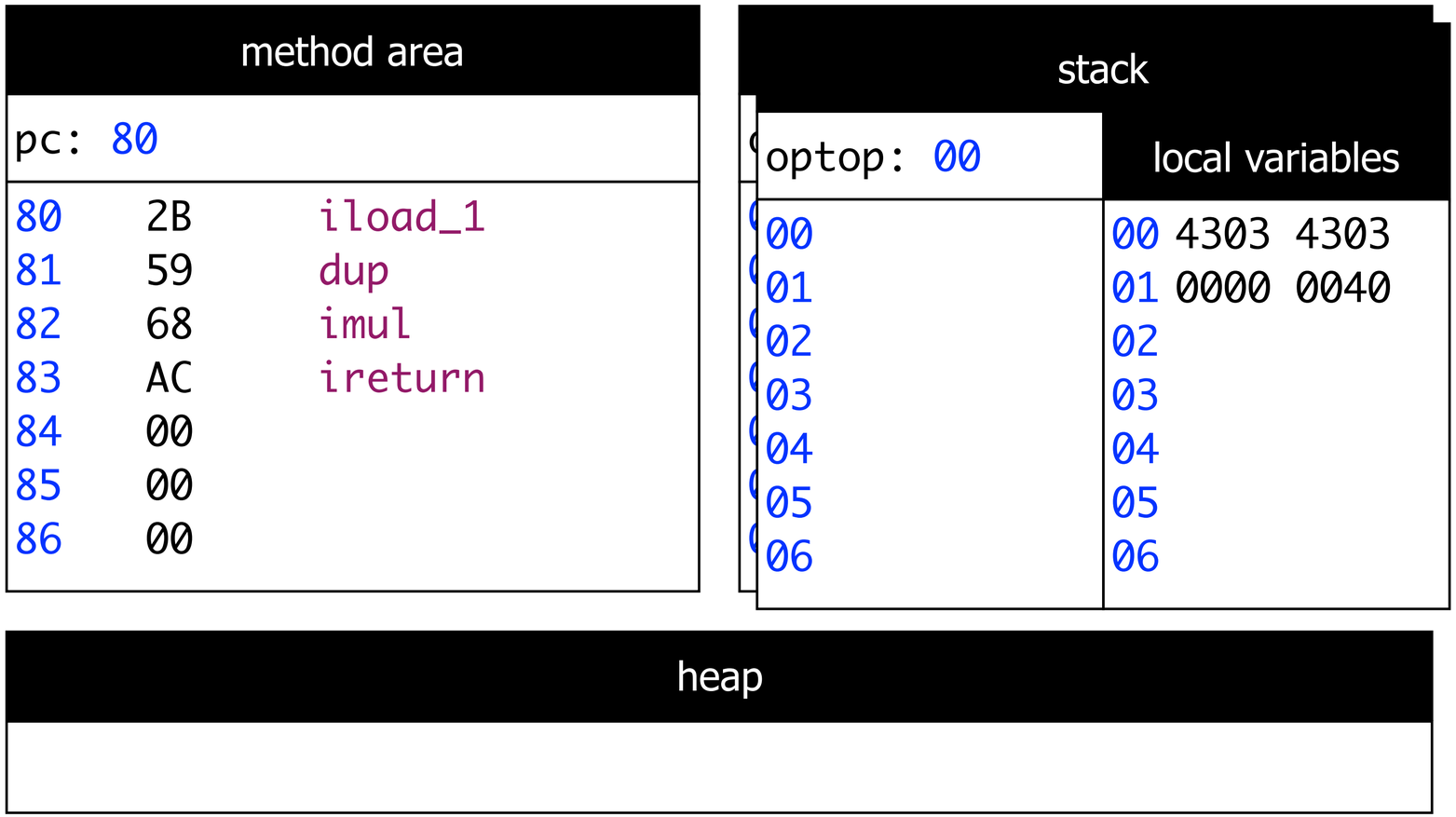
```
function fac(n: int): int =
   if
      n = 0
   then
      1
   else
      n * fac(n - 1)
```



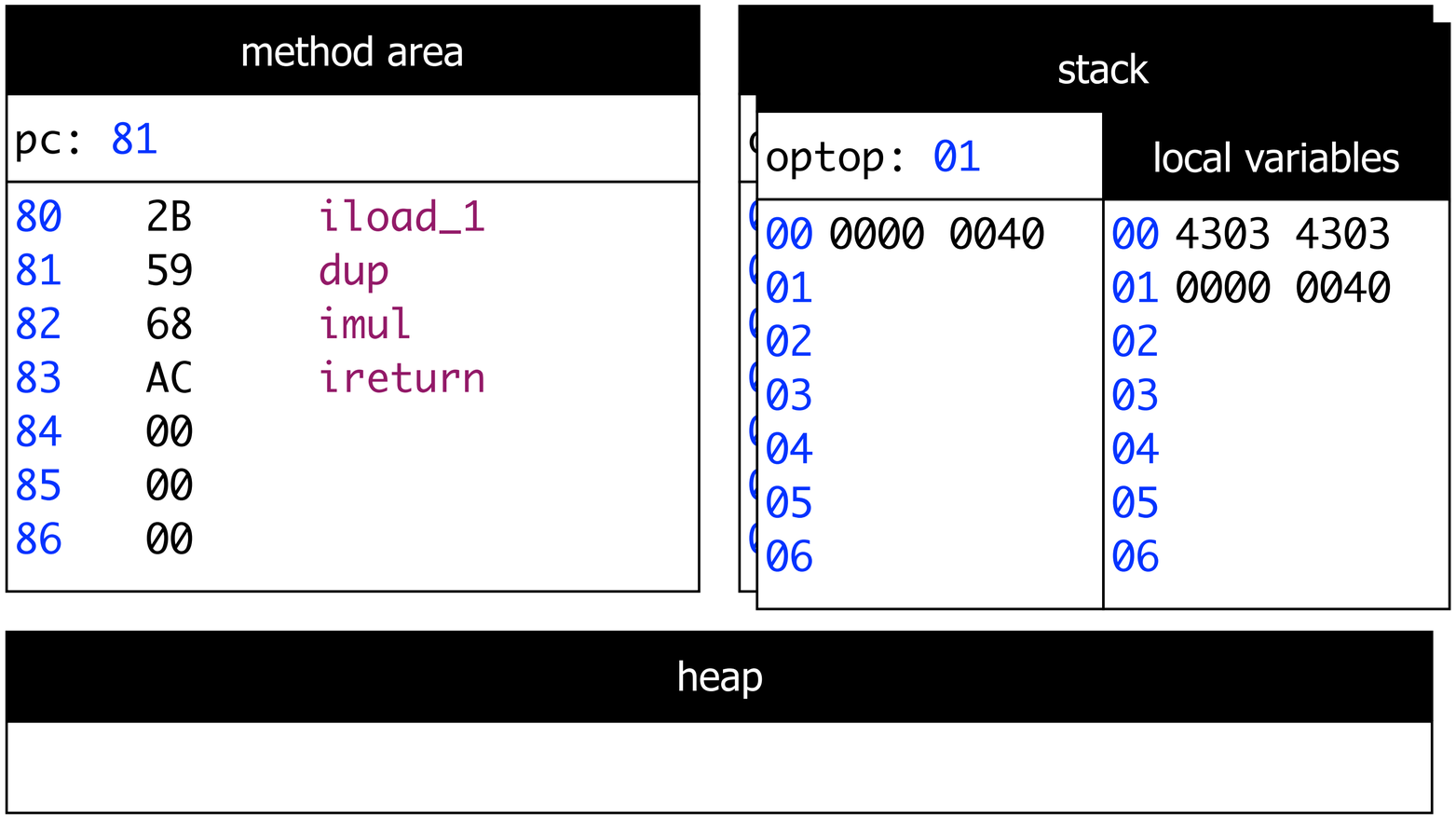
# Stack Frames



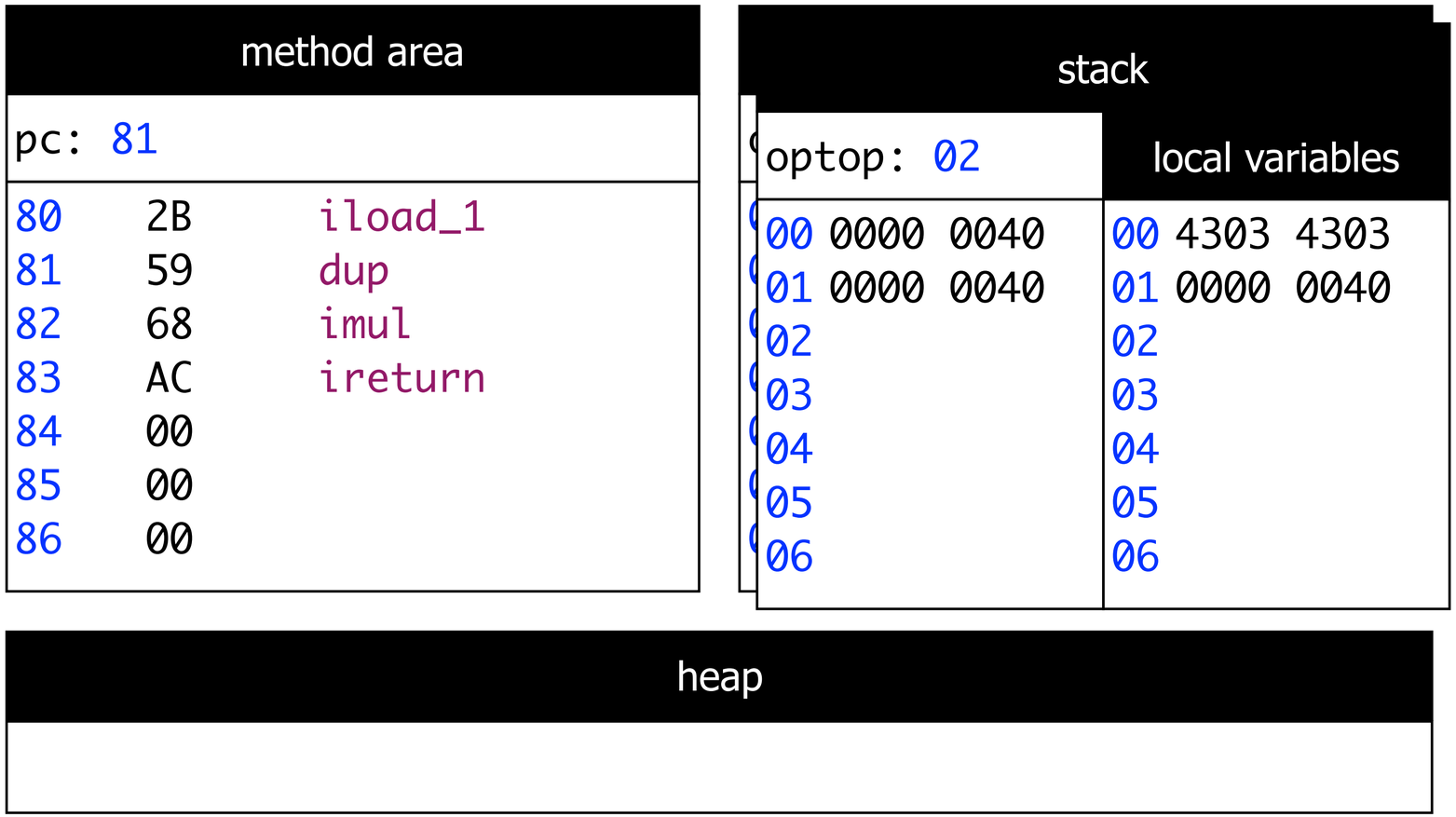
# Stack Frames



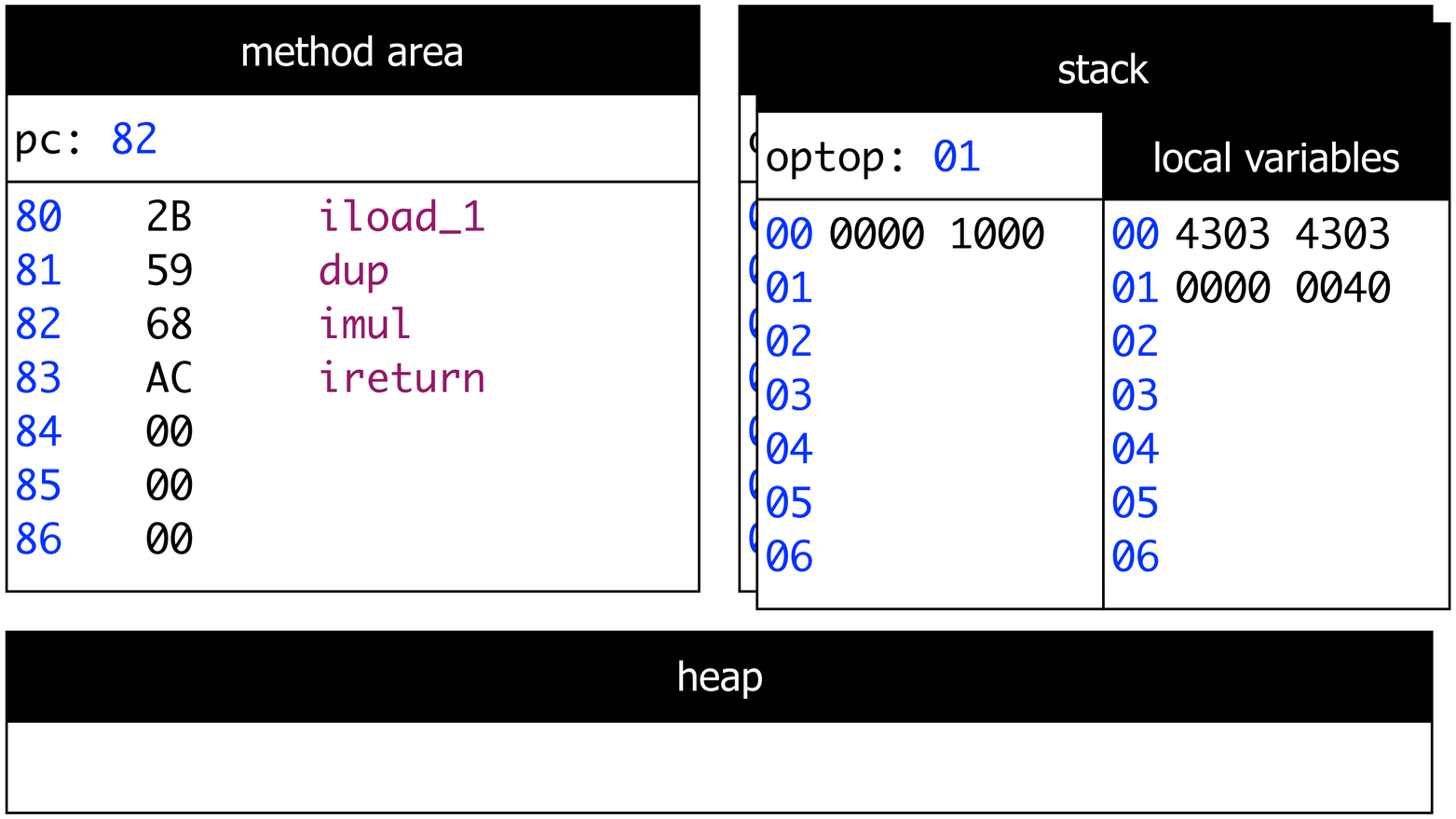
# Stack Frames



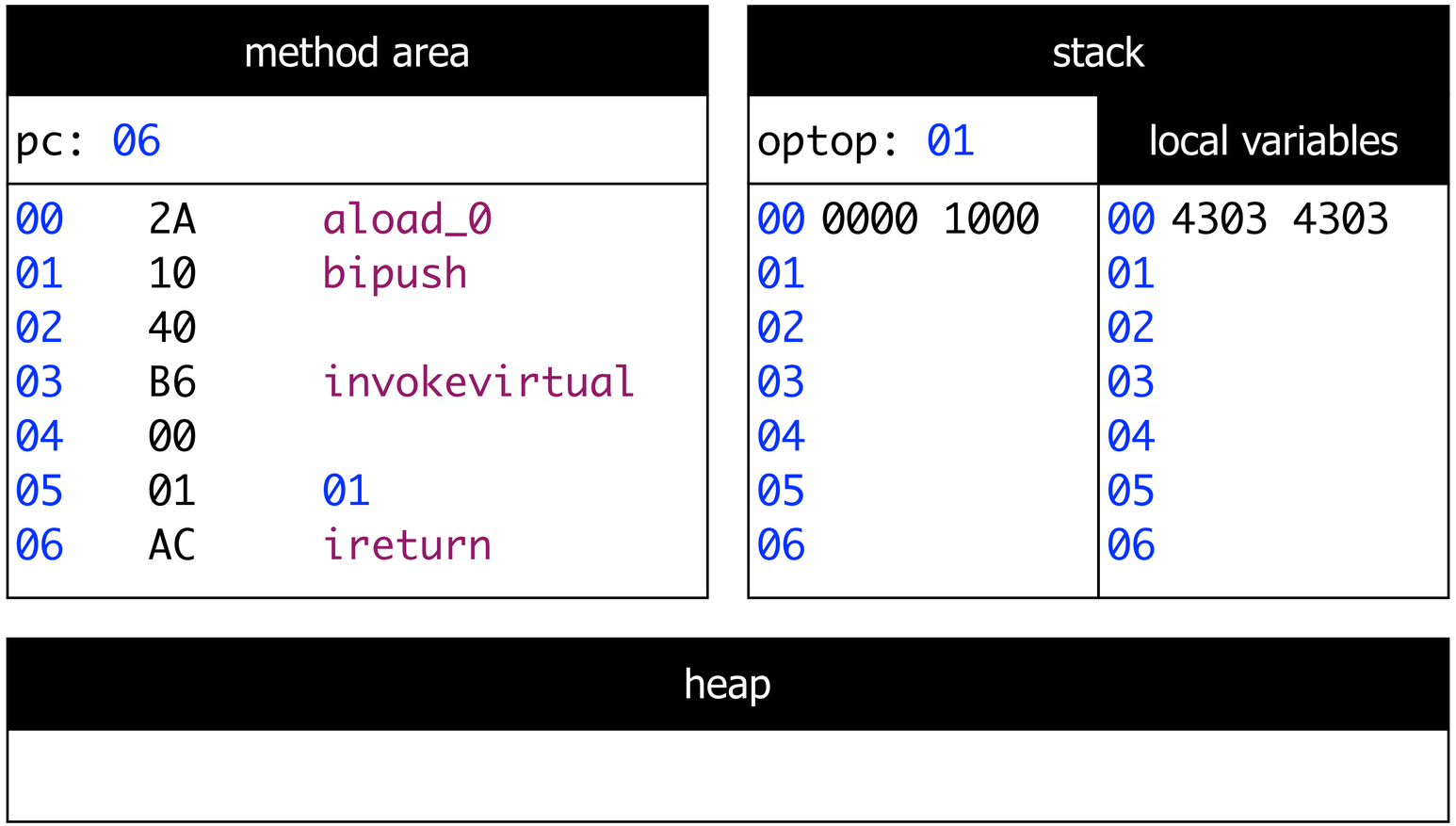
# Stack Frames



# Stack Frames



# Stack Frames



# JVM: Class Files

# Java Compiler

```
> ls

    Course.java

> javac -verbose Course.java

    [parsing started Course.java]
    [parsing completed 8ms]
    [loading java/lang/Object.class(java/lang:Object.class)]
    [checking university.Course]
    [wrote Course.class]
    [total 411ms]

> ls

    Course.class      Course.java
```

# Class Files: Format

magic number `CAFEBABE`

class file version (minor, major)

constant pool count + constant pool

access flags

this class

super class

interfaces count + interfaces

fields count + fields

methods count + methods

attribute count + attributes

# Jasmin Intermediate Language

```
.class public Exp

   .method public static fac(I)I

            iload 1
            ifne else

            iconst_1
            ireturn

      else: iload 1
            dup
            iconst_1
            isub
            invokestatic Exp/fac(I)I
            imul
            ireturn
   .end method
```

# Next: Code Generation