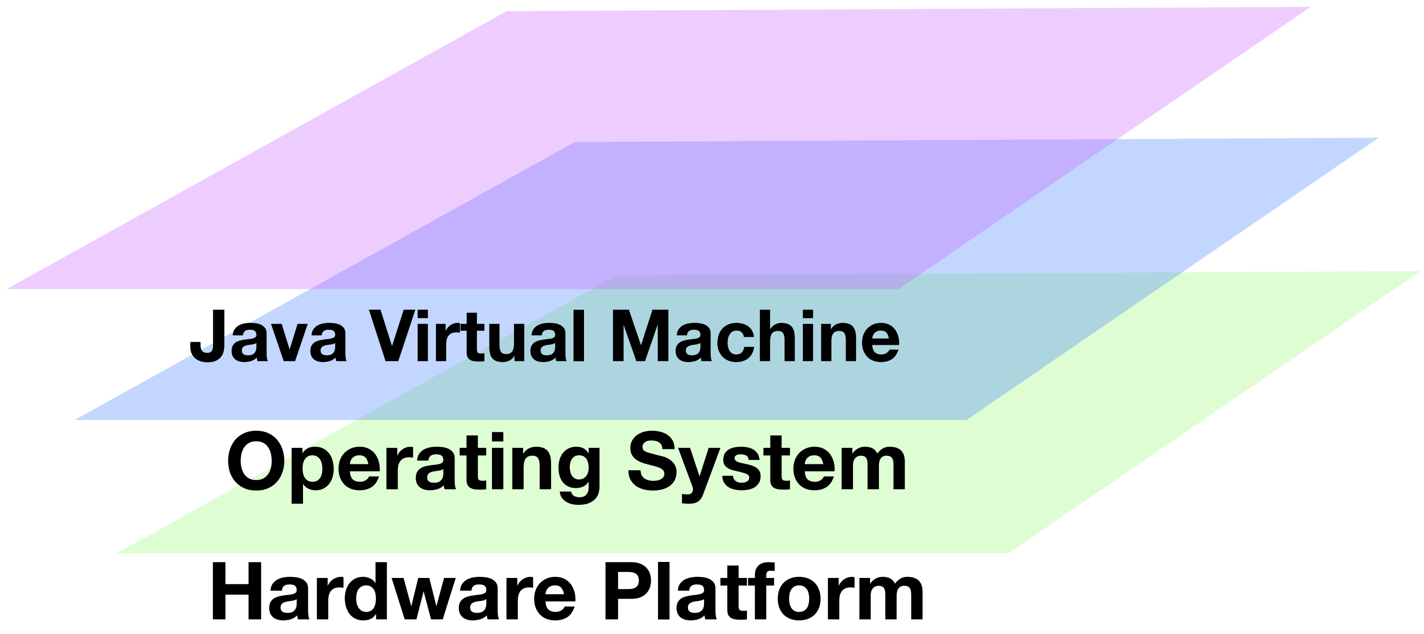
Lecture: Virtual Machines

Jan Vitek

CS

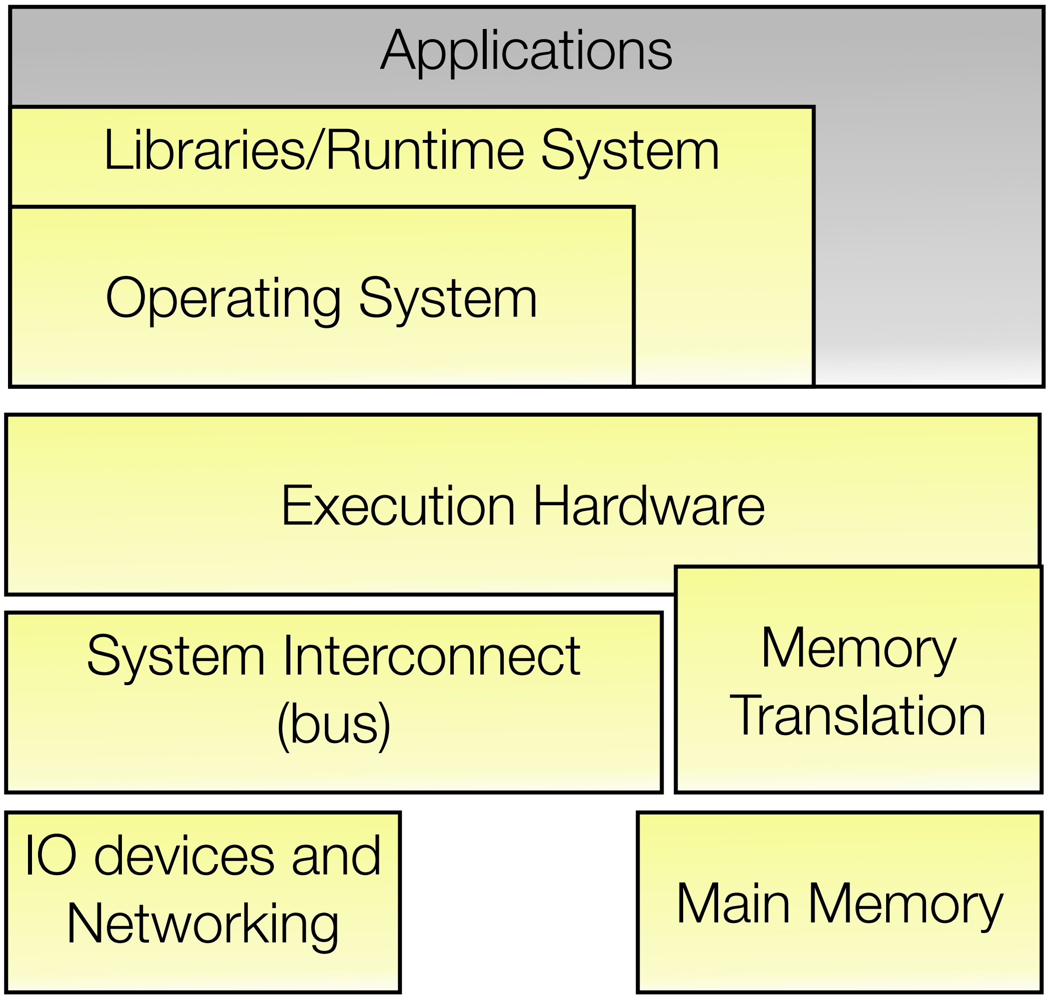
Spring 2011

Java Virtual Machine



Multiple Platforms

What is the "Machine"?



[James Smith, VEE'05]

Java

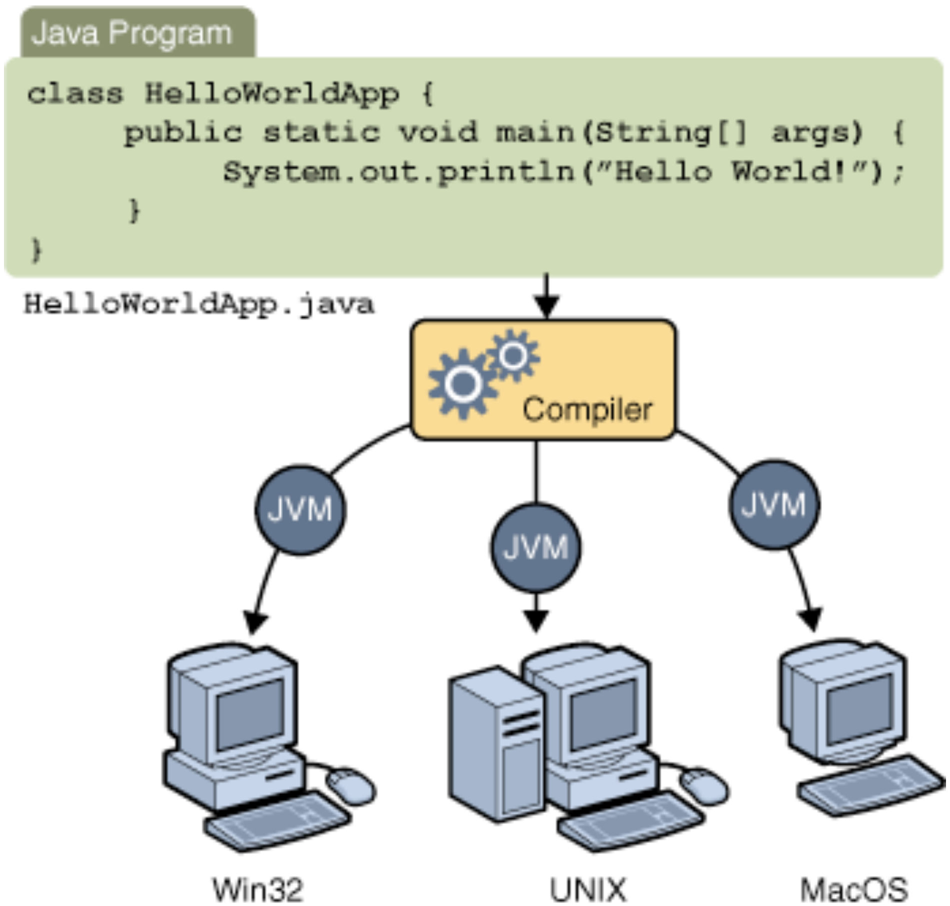
A few words of history

- ~**1985** - James Gosling designs NeWS - Network/extensible Window System for Sun Microsystems. NeWS is a portable dynamically-typed object-oriented language, with garbage collection, a portable code format, dynamic loading.

- **1992** - James Gosling designs Oak - a statically-typed object-oriented language for embedded devices. Oak has inheritance, garbage-collection, a portable intermediate representation, type-safe, but a syntax like C.

- **1995** - Java, aka Oak, is introduced. It runs on devices with >20MB of main memory.

From Source Code to Running Program



Java Virtualizes...

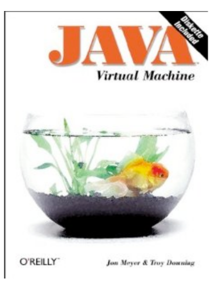
- ... locale
- ... threading
- ... endianness
- ... memory models
- ... operating system
- ... program extension
- ... data representation
- ... memory management

A Main Reference Source

The Java^{TM} Virtual Machine Specification (2nd Ed)
by Tim Lindholm & Frank Yellin
Addison-Wesley, 1999

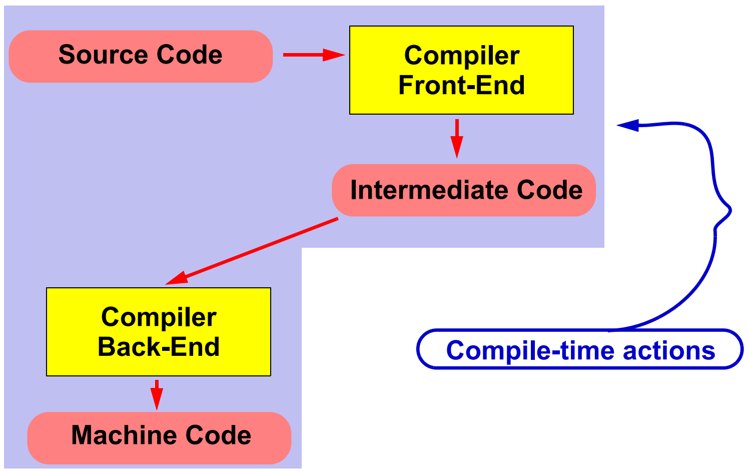
The book is on-line and available for download:

`http://java.sun.com/docs/books/vmspec/`

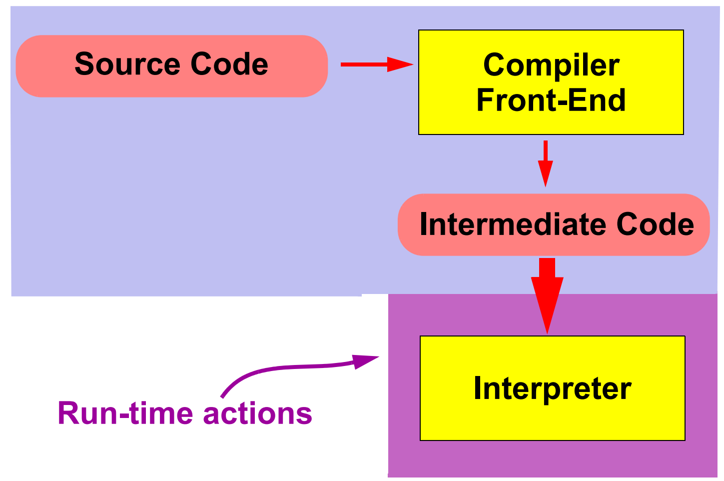


Implementing programming languages

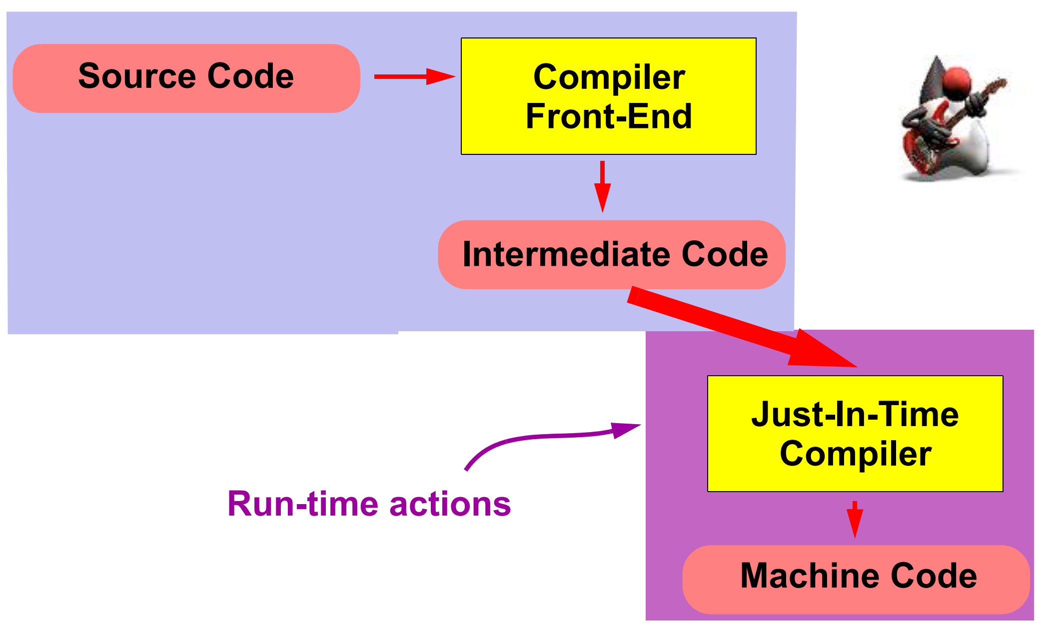
Usual Programming Language Implementation



Another Programming Language Implementation



And Another Implementation



An Overview

- Source code is translated into an intermediate representation, (IR)
- The IR can be processed in these different ways:
  1. compile-time (static) translation to machine code
  2. emulation of the IR using an interpreter
  3. run-time (dynamic) translation to machine code = JIT (Just-In-Time) compiling

What is IR?

IR is code for an idealized computer, a *virtual machine*.

A Small IR and its Interpreter (1/3)

We need a representation scheme for the bytecode. A simple one is:

- to use one byte for an opcode,
- four bytes for the operand of LDI,
- two bytes for the operands of LD, ST, JMP and JMPF.

As well as 0 for STOP, we will use this opcode numbering:

LDI	LD	ST	ADD	SUB	EQ	NE	GT	JMP	JMPF	READ	WRITE
1	2	3	4	5	6	7	8	9	10	11	12

The order of the bytes in the integer operands is important. We will use *big-endian* order.

A Small IR and its Interpreter (2/3)

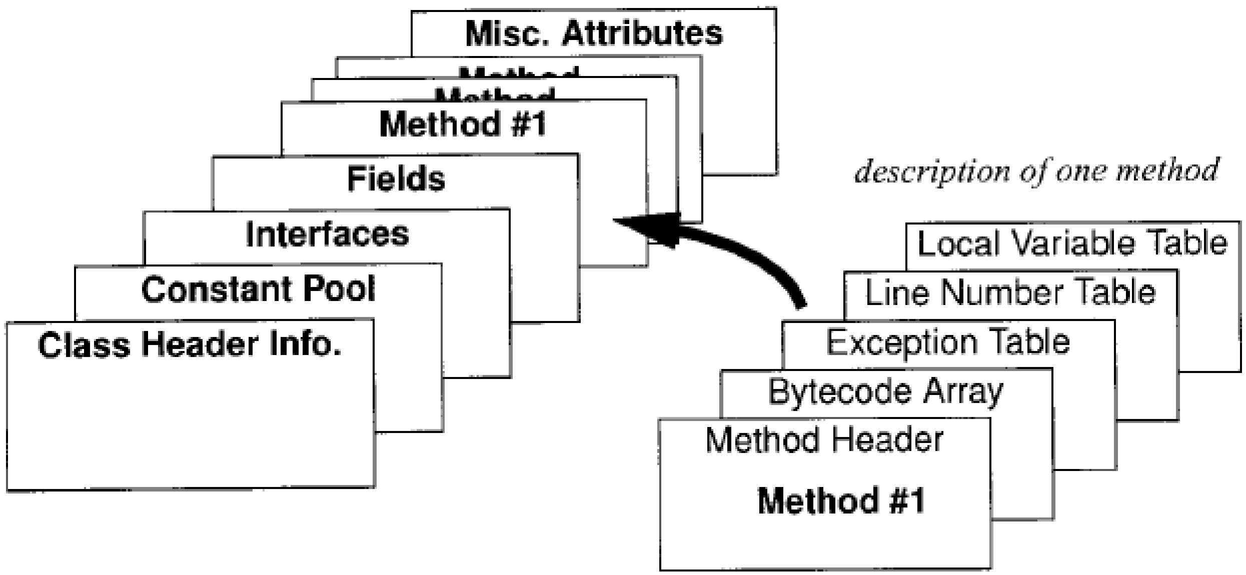
It emulates the fetch/decode/execute stages of a computer.

```
for( ; ; ) {
    opcode = code[pc++];
    switch(opcode) {
      case LDI:
          val = fetch4(pc);  pc += 4;
          push(val);
          break;
      case LD:
          num = fetch2(pc);  pc += 2;
          push( variable[num] );
          break;
      ...
      case SUB:
          right = pop();  left = pop();
          push( right-left );
      ...
```

A Small IR and its Interpreter (3/3)


```
        case JMP:
            pc = fetch2(pc);
            break;
        case JMPF:
            val = pop();
            if (val)
                pc += 2;
            else
                pc = fetch2(pc);
            break;
        ...
    }  /* end of switch */
}  /* end of for loop */
```

The Java Classfile

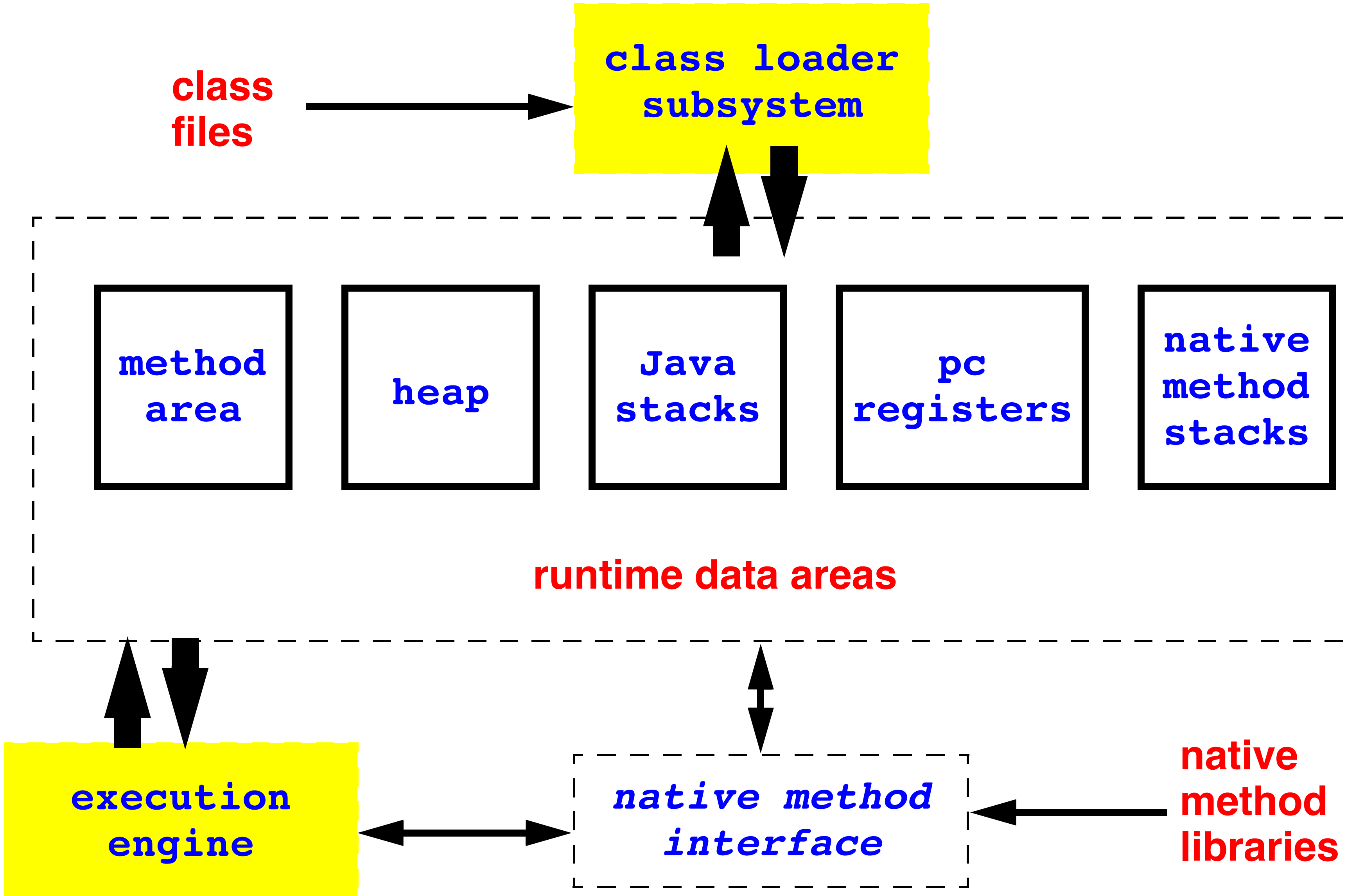


JVM Architecture

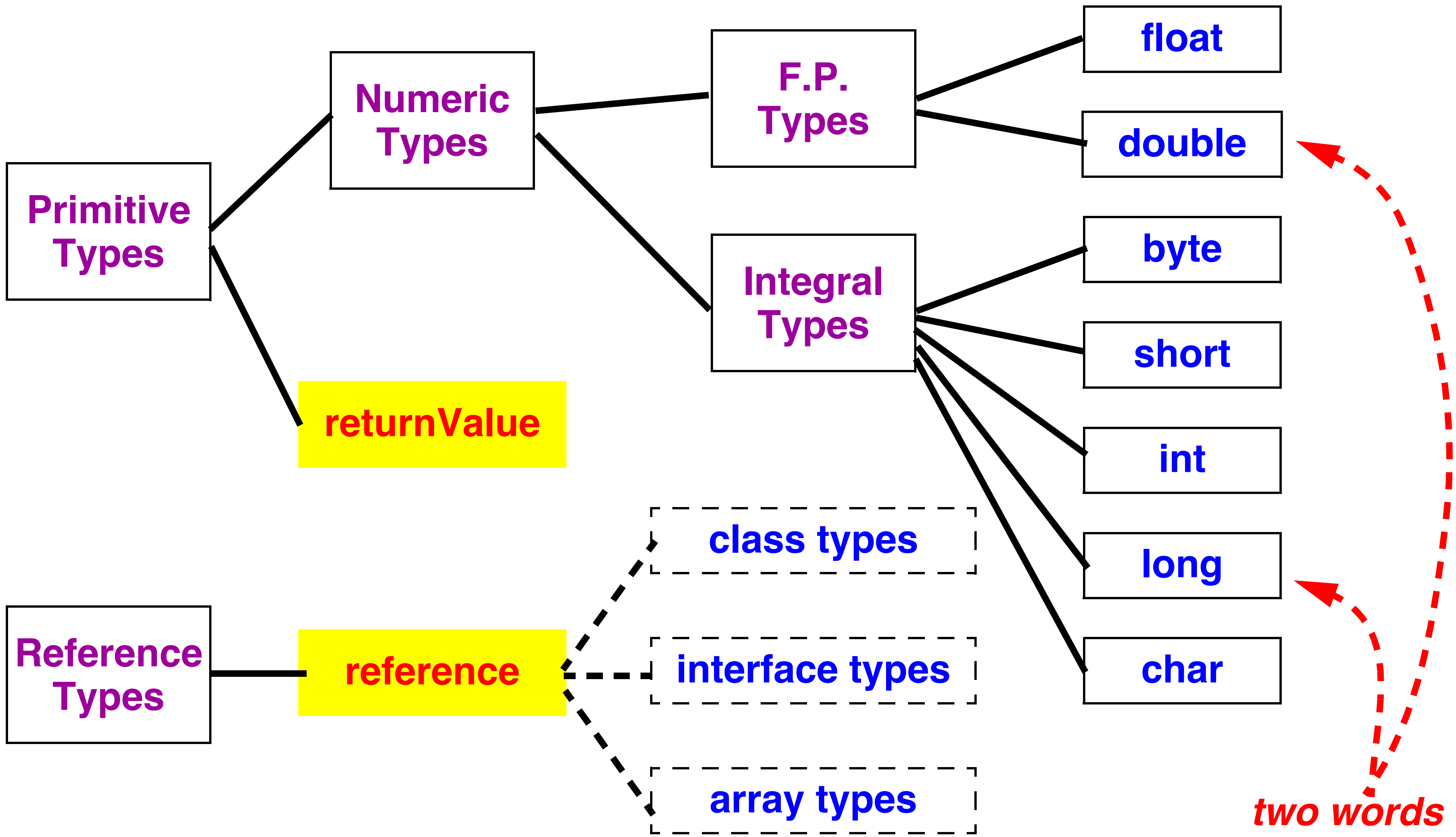
The internal runtime structure of the JVM consists of:

- One: (i.e. shared by all threads)
  - method area
  - heap
- For each thread, a:
  - program counter (pointing into the method area)
  - Java stack
  - native method stack (system dependent)

Runtime structure



Datatypes


Primitive Types
Numeric Types
F.P. Types
float
double
Integral Types
byte
short
int
long
char
returnValue
Reference Types
reference
class types
interface types
array types
two words

Java Bytecode


```
int bar(int i) {
    try {
        if (i == 3) return this.foo();
    } finally {
        this.ladida();
    }
    return i;
}
```

Region	Target
1–12	17
13–16	21

```
01  iload_1                 // Push i
02  iconst_3                // Push 3
03  if_icmpne 10            // Goto 10 if i does not equal 3
    // Then case of if statement
04  aload_0                 // Push this
05  invokevirtual foo       // Call this.foo
06  istore_2                // Save result of this.foo()
07  jsr 13                  // Do finally block before returning
08  iload_2                 // Recall result from this.foo()
09  ireturn                 // Return result of this.foo()
    // Else case of if statement
10  jsr 13                  // Do finally block before leaving try
    // Return statement following try statement
11  iload_1                 // Push i
12  ireturn                 // Return i
    // finally block
13  astore_3                // Save return address in variable 3
14  aload_0                 // Push this
15  invokevirtual ladida    // Call this.ladida()
16  ret 3                   // Return to address saved on line 13
    // Exception handler for try body
17  astore_2                // Save exception
18  jsr 13                  // Do finally block
19  aload_2                 // Recall exception
20  athrow                  // Rethrow exception
    // Exception handler for finally body
21  athrow                  // Rethrow exception
```

Virtualizing Memory

- Memory is a set of objects with fields, methods and a class + local variables of a method
- Memory is read by accessing a field or local variable
- Memory is modified by writing to a field or local variable
- Location and size of data are not exposed
- Memory allocation is done by call in new

```
public class Main {
  static public
    void main(String[] a){
     Cell c1, c2 = null;
     while (true) {
       c1 = new Cell();

       c2 = c1;
     }
  }
}
```

- Question:
  - Does `main()` terminate?

```
class Cell {Cell next; }
```

Virtualizing Memory

- Memory is a set of objects with fields, methods and a class + local variables of a method
- Memory is read by accessing a field or local variable
- Memory is modified by writing to a field or local variable
- Location and size of data are not exposed
- Memory allocation is done by call in new

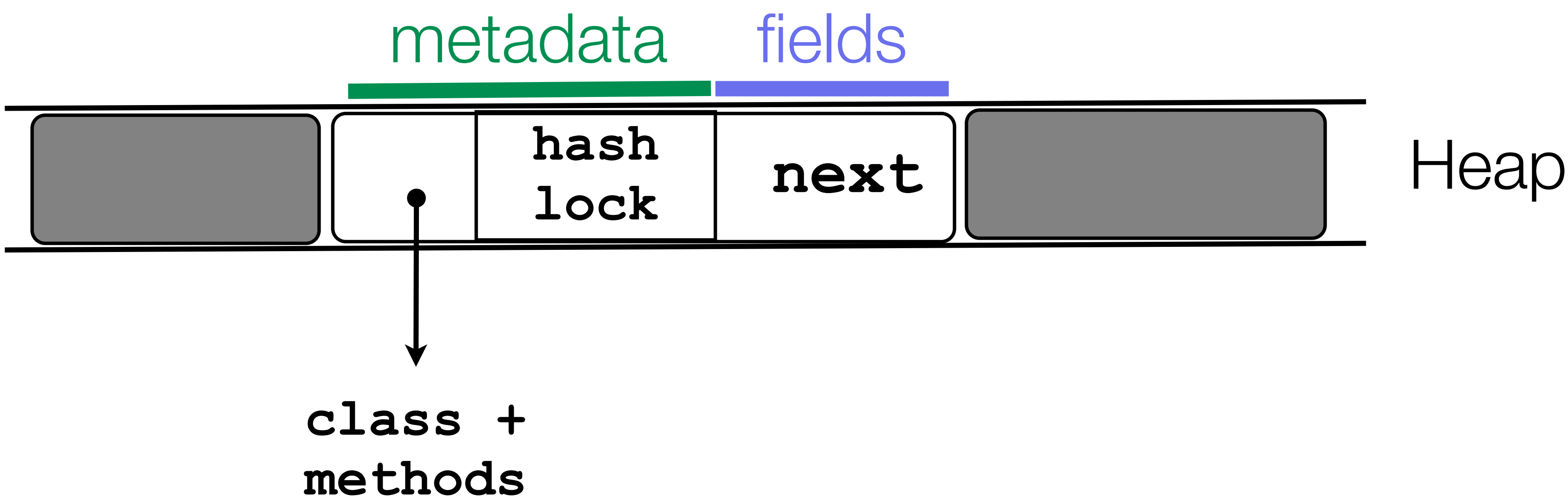
```
public class Main {
  static public
   void main(String[] a){
    Cell c1, c2 = null;
    while (true) {
      c1 = new Cell();
      c1.next = c2;
      c2 = c1;
    }
  }
}

class Cell {Cell next; }
```

- **Question:**
  - Does `main()` terminate?

Virtualizing Memory

- The semantics of `new` is as follows:
  - Allocate space for the object's fields and metadata fields
  - Initialize the metadata fields
  - Set all fields to null/zero/false
  - Invoke the user defined constructor method

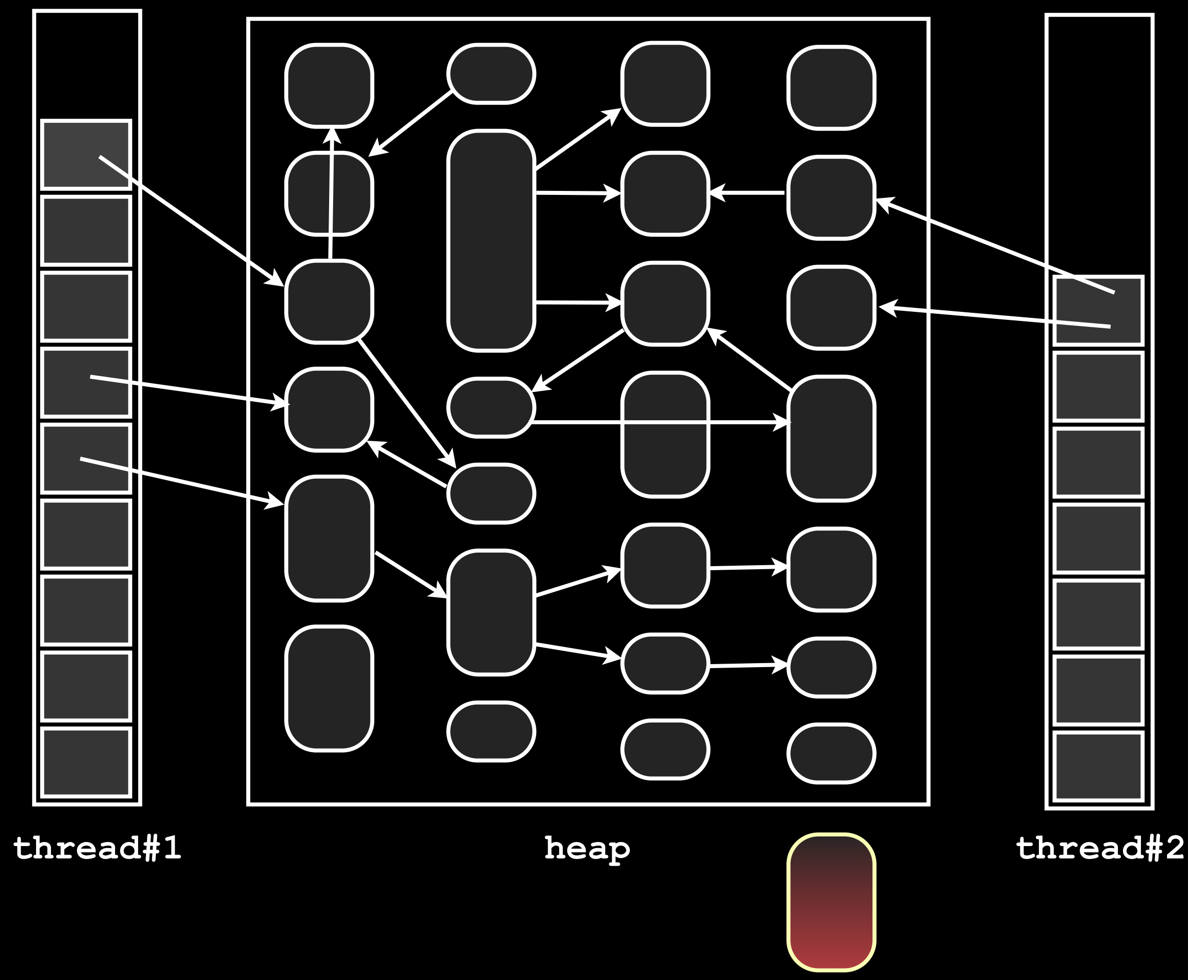


Virtualizing memory

- Garbage collection is the technology that gives the illusion of infinite resources
- Garbage collection or GC is implemented by the programming language with the help of the compiler
  - Though for a some well-behaved C programs it is possible to link a special library that provides most of the benefits of GC

  - Question:
    - *How does GC work?*

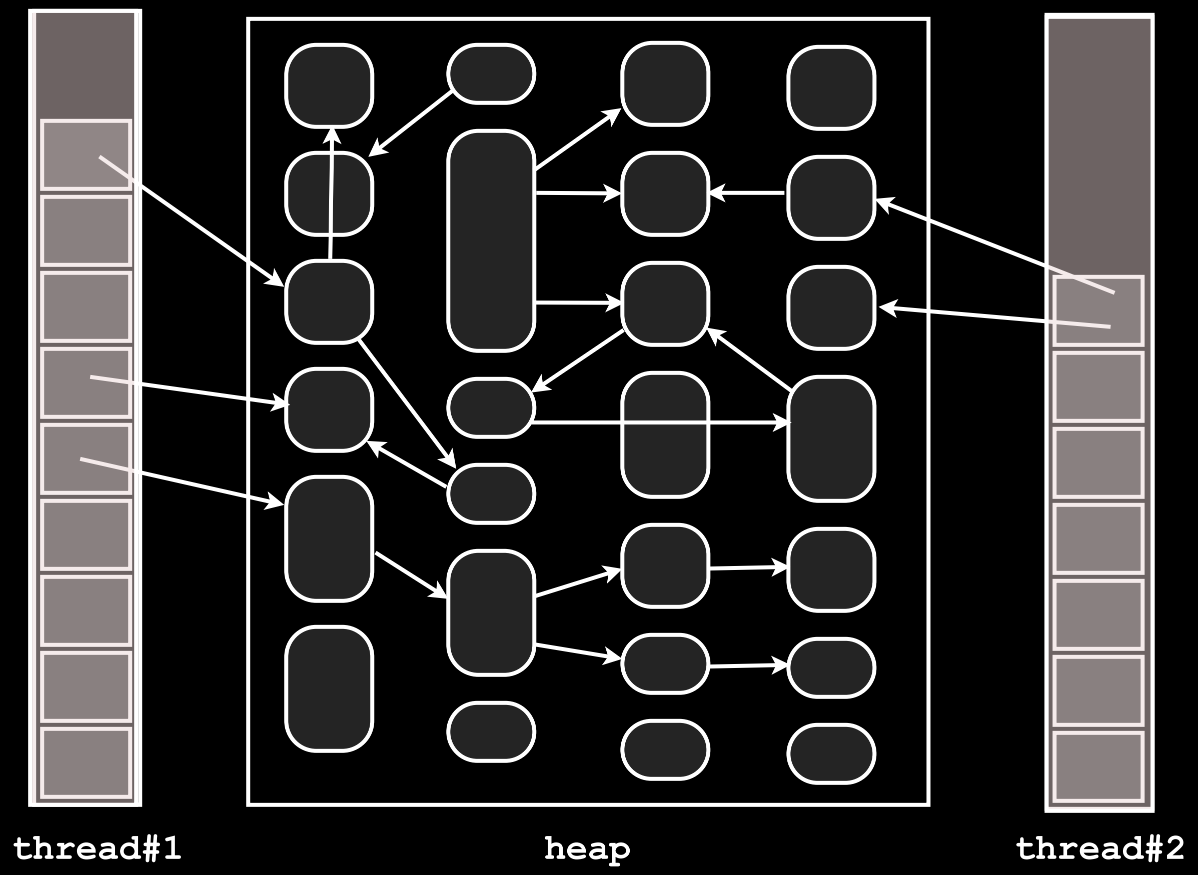
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

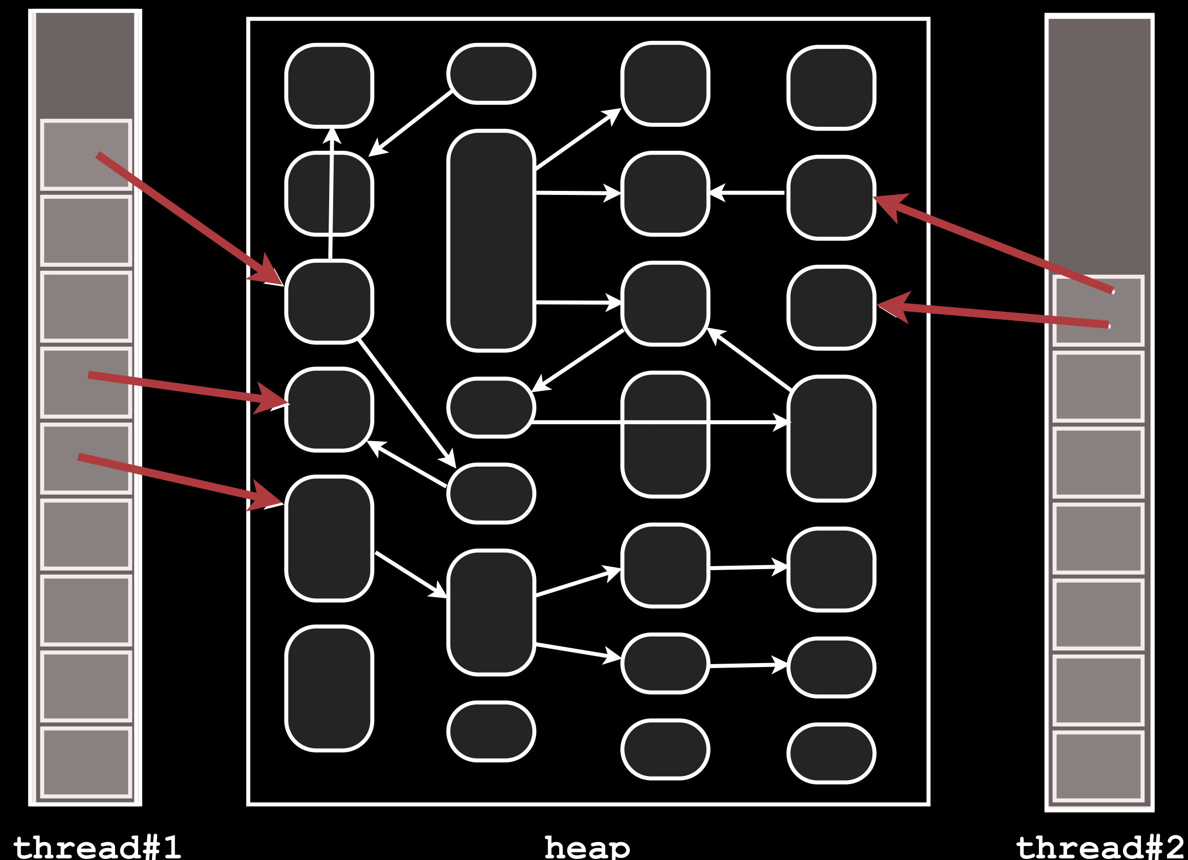
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

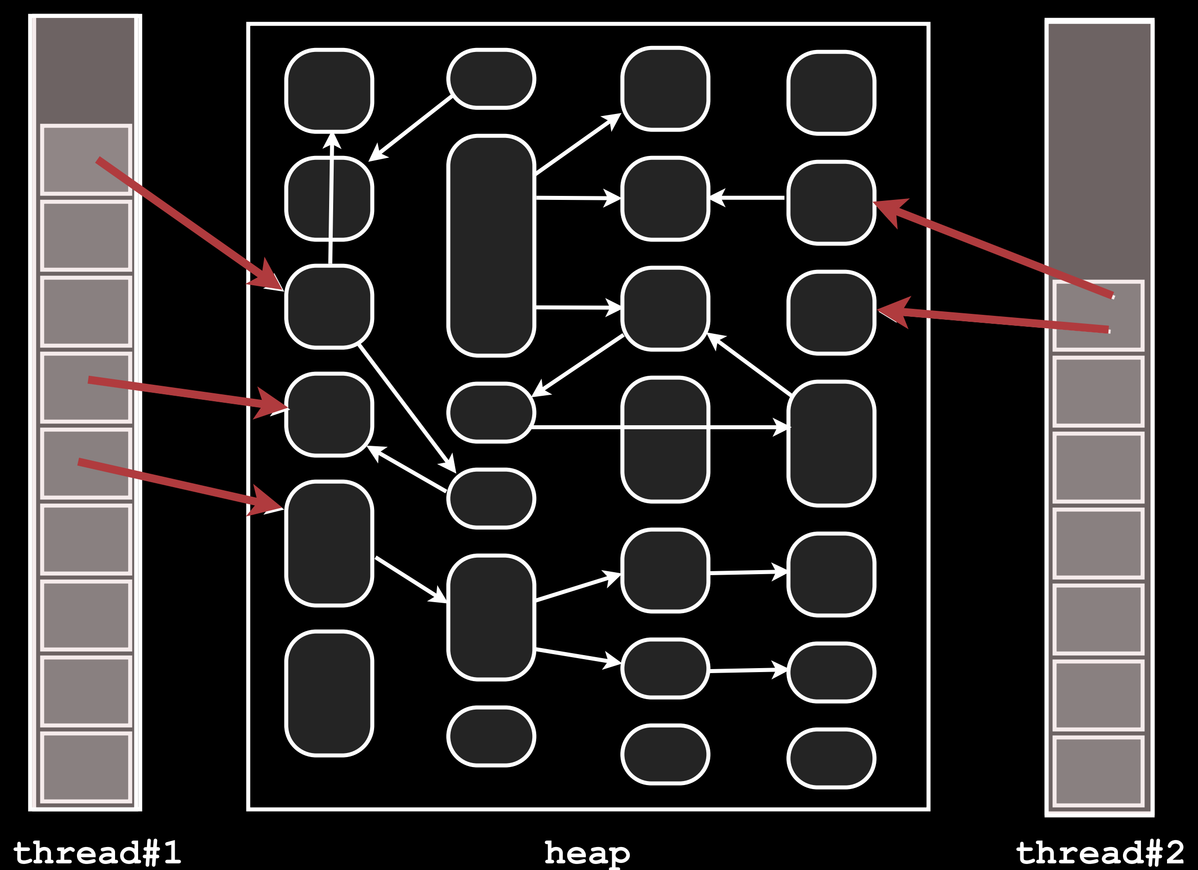
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

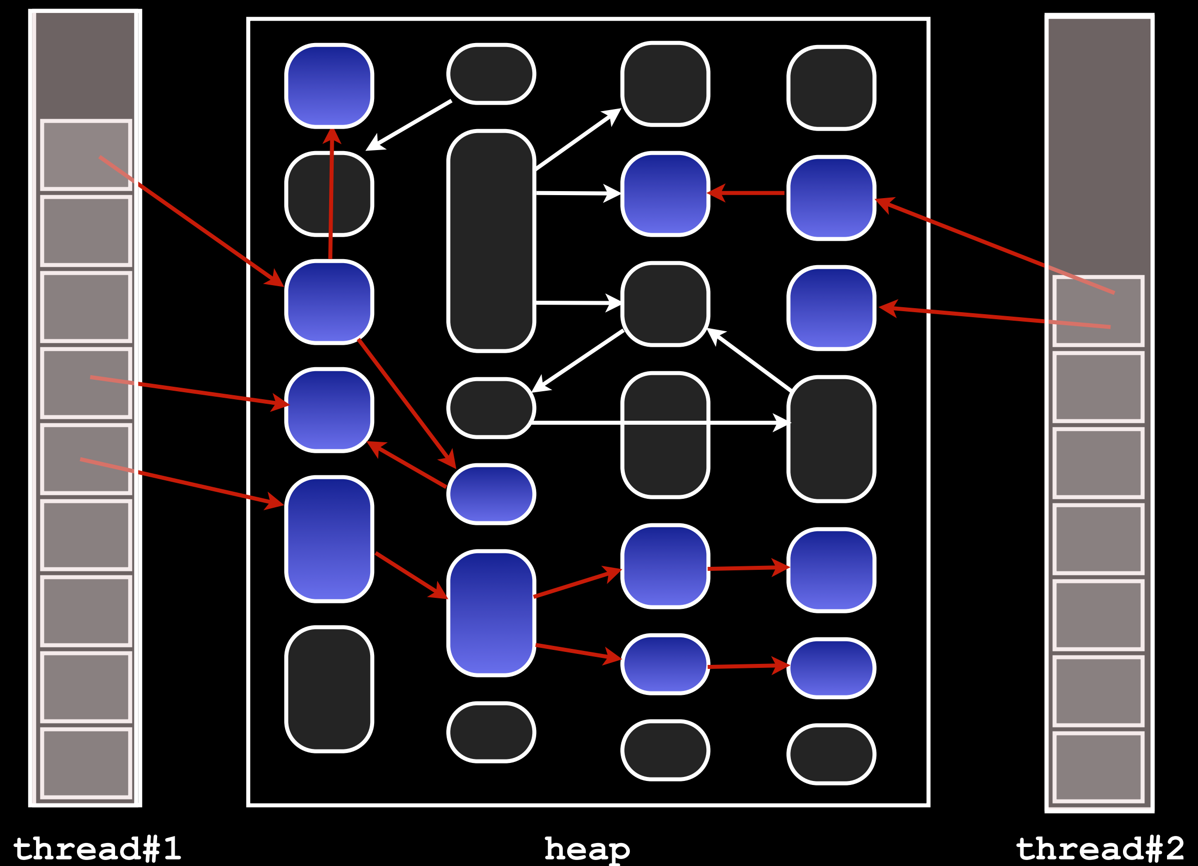
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

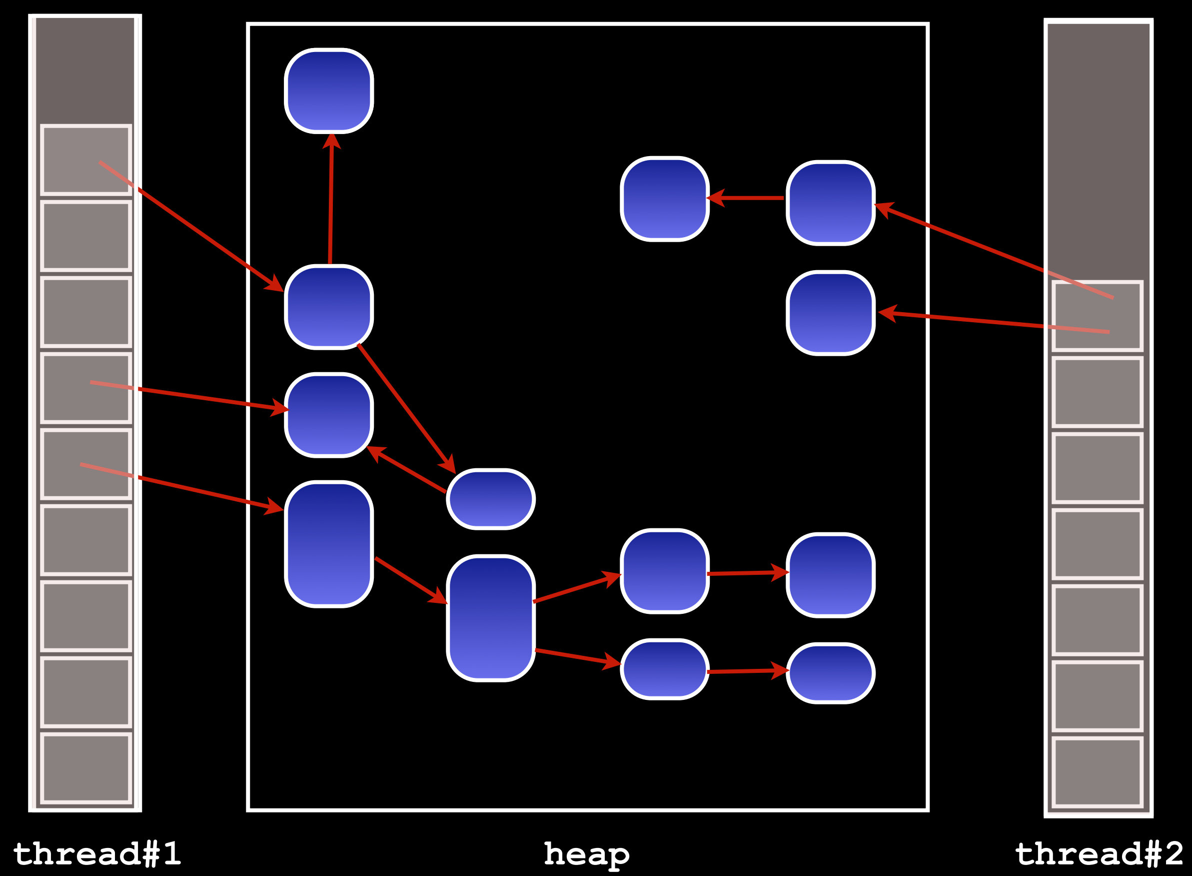
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

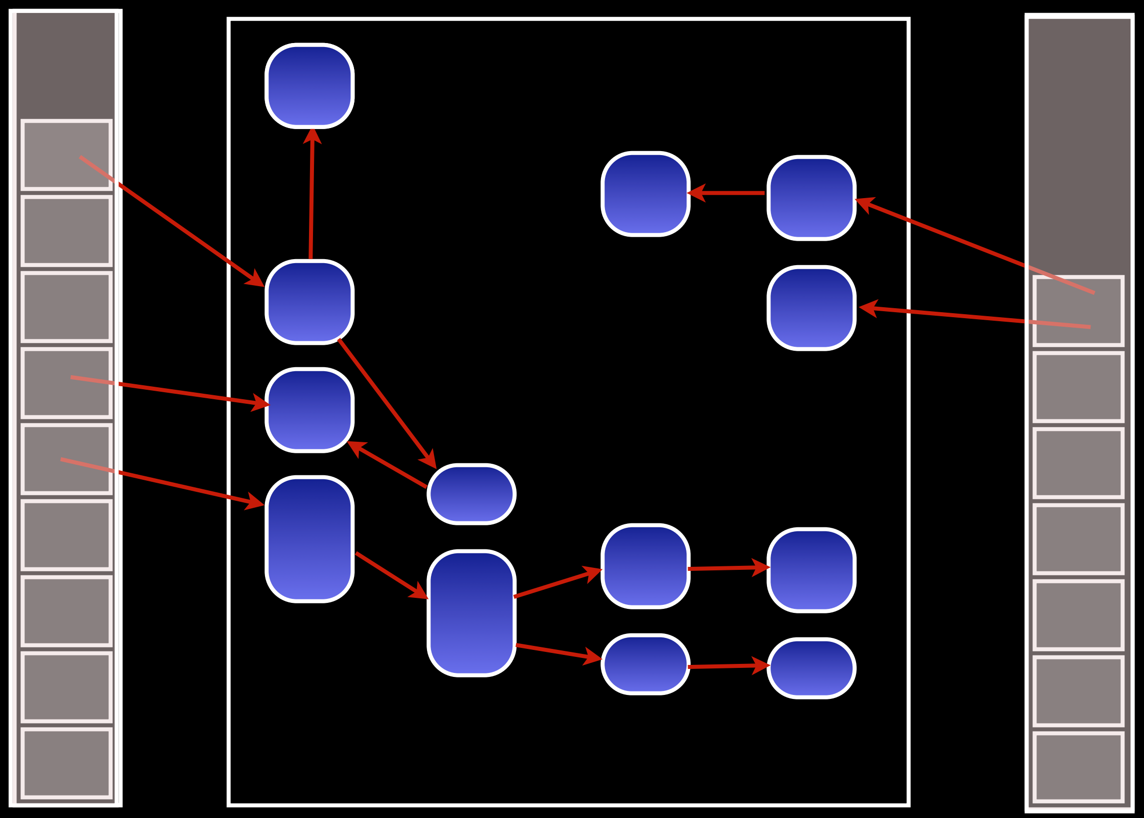
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

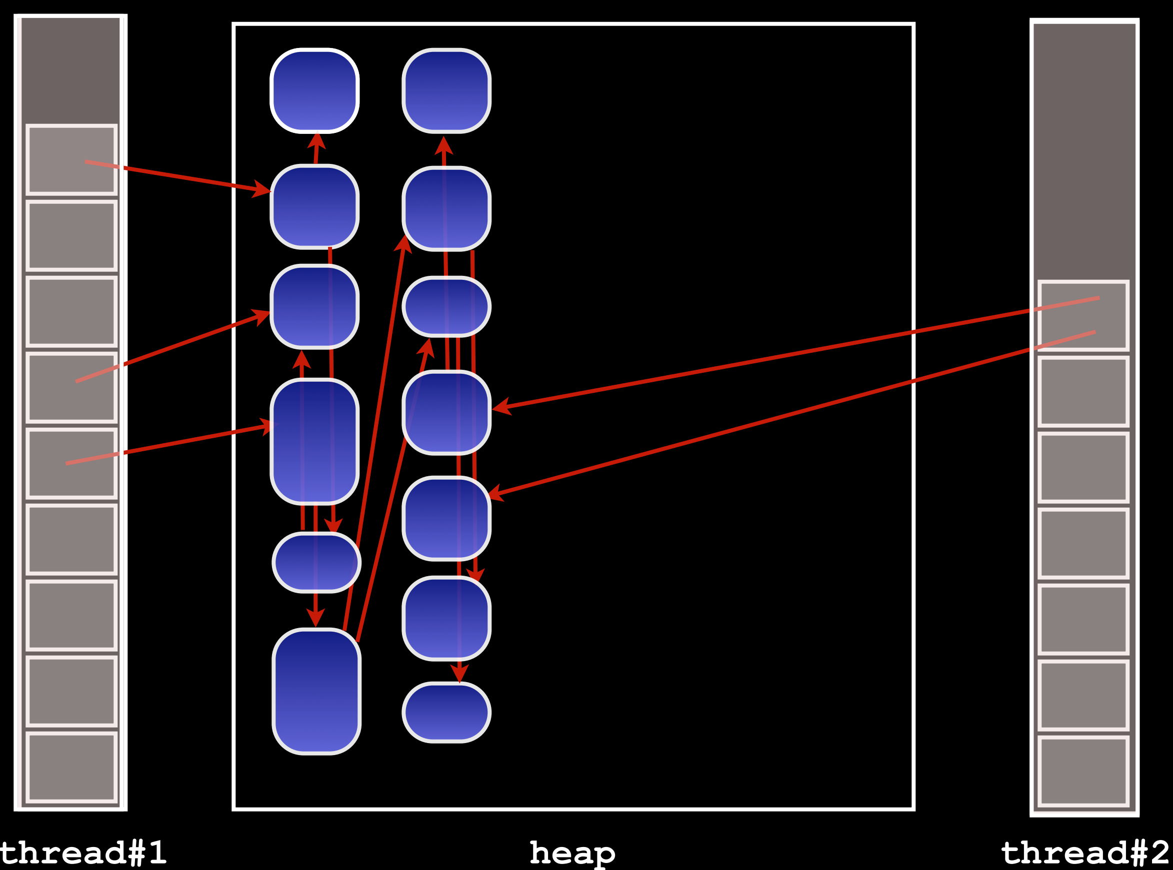
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Conclusion

- The Java Virtual Machine provides a level of abstract over the hardware and the operating system that hides their specificities
- Java source code is compiled to bytecode (javac)
- Bytecode is either interpreted or JIT-ed to execute by the JVM (java)