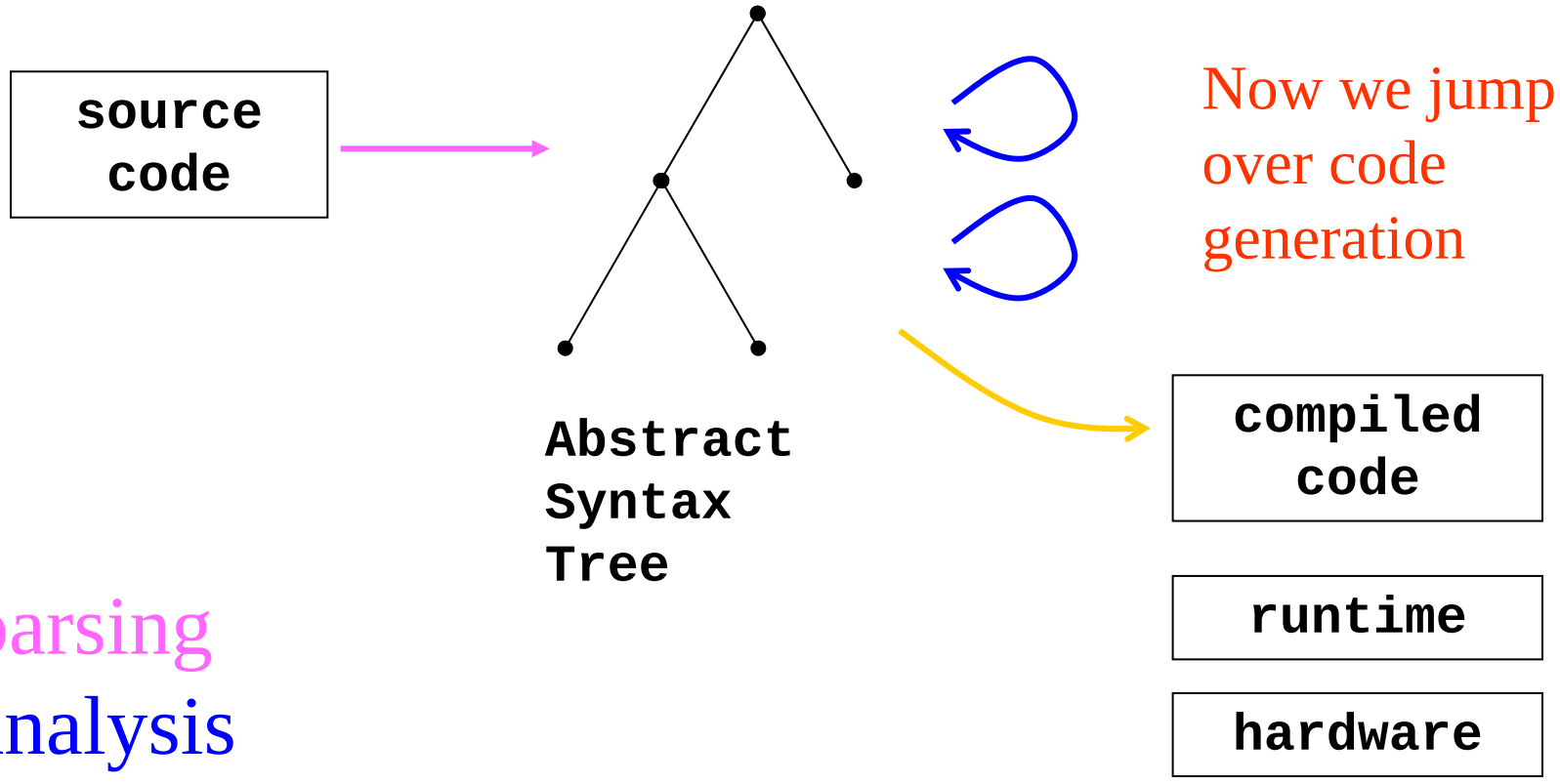
Project

- there are a couple of 3 person teams
  - regroup
  - or see me
  - or forever hold your peace
- a new drop with new type checking is coming
  - using it is optional

Compiler Architecture



Overview of Runtime Lectures

A compiler translates a program from a high-level language into an corresponding program in a low-level language.

The low level program must correspond to the high-level program.
=> High-level concepts must be modeled in terms of the low-level machine.

These lectures are not about compilation itself
instead are about compiled code
and how to use compiled code to implement high level constructs

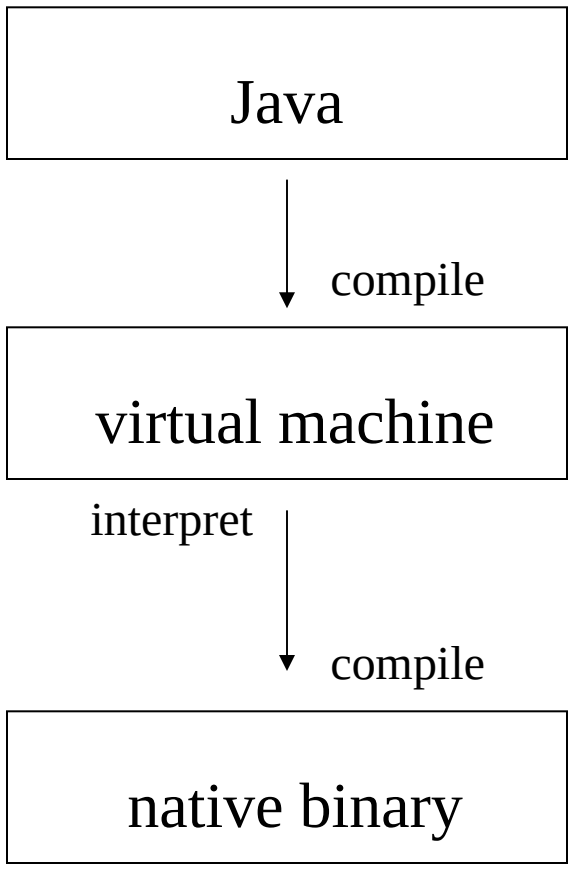
=> We need to know this before we can talk about code generation.

Overview of Runtime Lectures

- **Expression Evaluation:** How to organize computing the values of expressions (taking care of intermediate results)
    - even x + y * z is surprisingly ‘high-level’
- **Routines:** How to implement procedures, operators (and how to pass their parameters and return values)
- **Data Representation:** how to represent values of the source language on the target machine.
- **Storage Allocation**: How to organize storage for variables (considering different lifetimes of global, local and heap variables)

- A runtime design involves all of these, in delicate balance.
  → there is no forward reference free order to present this

Different Levels of Machines

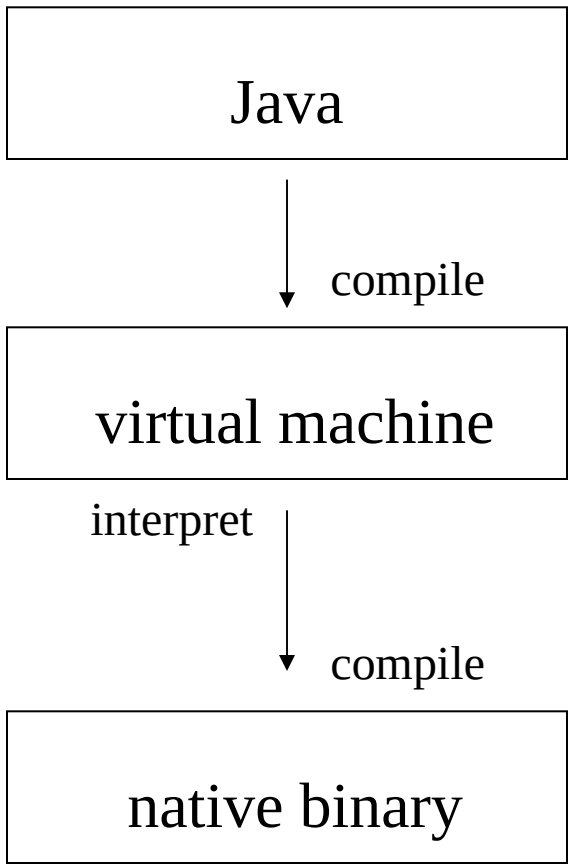


`portable, objects, type checking, modules, lexical scoping, gc, tree structured code…`

`???`

`machine-specific, flat code, flat memory`

Different Levels of Machines



`portable, objects, type checking, modules, lexical scoping, gc, tree structured code…`

`JVM: GC, stack machine, flat code, type safe`

`machine-specific, flat code, flat memory`

Virtual Machine Code

- Examples:
  - P-code for early Pascal interpreters
  - Smalltalk virtual machines
  - Lisp Machine instruction set
  - Java byte codes, CLR byte codes
- Advantages
  - easy to generate byte code
  - code is architecture independent
  - code can be more compact
- Disadvantage
  - true interpretation is slower

Compiling Virtual Machine Code

- Example:
  - gcc works in two steps
    - first compiles to RTL
      then compiles RTL to native code
- Advantages:
  - exposes key properties of underlying architecture
  - abstracts over true details of underlying architecture
  - facilitates writing code generators for multiple target machines
- Disadvantages
  - code bloat
  - have to write code generator for each target architecture

Interpreting and Compiling Byte Code

- Modern virtual machines do both
- Interpret byte code
- Selectively compile byte-code to native machine code
  - aka Just-in Time (JIT) compiler

- Attempt to get best of both worlds
  - compact code and fast code
- Requires subtle decisions about when to compile
- Originally Smalltalk VM, later Self, now all VMs
  - the Sun “HotSpot VM” uses “just-in time” compiling

Java Virtual Machine

- Has:
  - garbage-collected memory
  - registers
  - condition codes
  - execution unit

JVM memory

- stack
  - method (procedure) call frames
- heap
  - used for dynamically allocated memory (objects and arrays)
- constant pool
  - shared constant data
- code segment
  - JVM instructions of all loaded classes

JVM registers

- no general purpose registers
- byte code has limited control over registers
- stack pointer (sp)
  - points to top of stack
- local stack pointer (lsp)
  - points to location in current frame
  - used for intermediate values in this stack frame
- program counter (pc)
  - current instruction

JVM condition codes

- stores result of last instruction that sets condition codes
  - used for branching

JVM execution unit

- reads instruction at PC
  - decodes and executes instruction
- state of machine may change
  - memory, registers, condition codes
- pc is incremented after executing instruction
- method calls and branches explicitly change pc

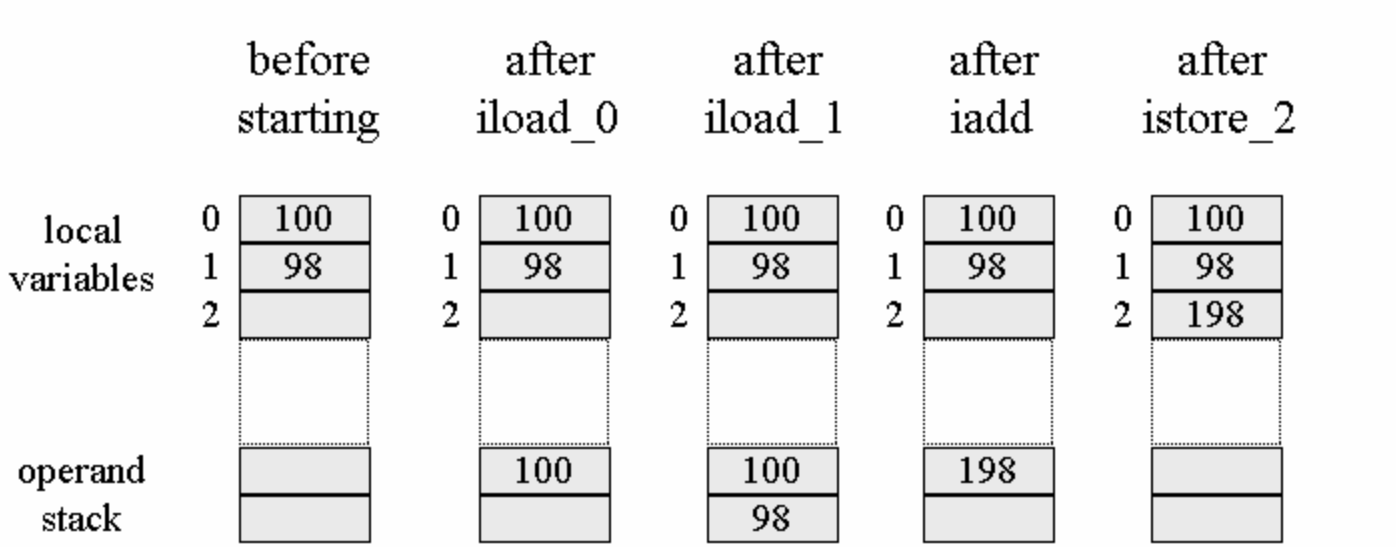
JVM stack frames

- locals
  - reference to current object (this)
  - method arguments
  - local variables
- local stack for intermediate results
  - returned from calls and operations
  - ready to be passed to other calls and operations
  - or ready to be stored in locals
- number of locals and size of local stack fixed at compile time

picture from Artima.com

```
static void foo(int x, int y) {
  int c = x + y;
  ...
}
```

```
iload_0  // push the int in local variable 0
iload_1  // push the int in local variable 1
iadd     // pop two ints, add them, push result
istore_2 // pop int, store into local variable 2
```



iadd Instruction (from http://mrl.nyu.edu/~meyer/jvmref/)

(also see http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions2.doc.html)

add two integers

- **Stack**
  - **Before** **After**
    value1 result
    value2 ...
    ...
- **Description**

  Pops two integers from the operand stack, adds them, and pushes the integer result back onto the stack. On overflow, iadd produces a result whose low order bits are correct, but whose sign bit may be incorrect.
- **Example**

```
bipush 5    ; push first int
bipush 4    ; push second int
iadd        ; add integers
            ; the top of the stack now
            ; contains the integer 9
```

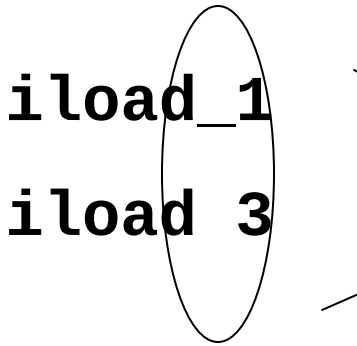
Simple Java Method

```
public int abs(int x) {
if( x < 0 )
  return x * -1;
else
  return x;
```

```
.method public abs(I)I // (int)->int
.limit stack 2
.limit locals 2
                  // --locals--   --stack--
 iload_1          //  [      ]    [        ]
 ifge Label1      //  [      ]    [        ]
 iload_1          //  [      ]    [        ]
 iconst_m1        //  [      ]    [        ]
 imul             //  [      ]    [        ]
 ireturn          //  [      ]    [        ]
Label1:
 iload_1          //
 ireturn          //
.end method
```

Simple Java Method

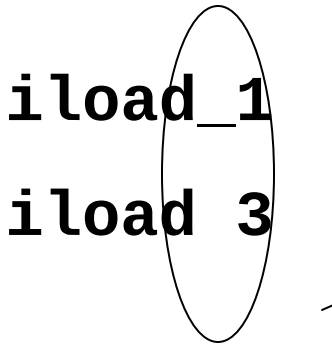
```
public int abs(int x) {
if( x < 0 )
  return x * -1;
else
  return x;
```




```
.method public abs(I)I // (int)->int
.limit stack 2
.limit locals 2
                  // --locals--   --stack--
 iload_1          //  [      ]    [        ]
 ifge Label1      //  [      ]    [        ]
 iload_1          //  [      ]    [        ]
 iconst_m1        //  [      ]    [        ]
 imul             //  [      ]    [        ]
 ireturn          //  [      ]    [        ]
Label1:
 iload_1          //
 ireturn          //
.end method
```

Simple Java Method

```
public int abs(int x) {
if( x < 0 )
  return x * -1;
else
  return x;
```




```
.method public abs(I)I // (int)->int
.limit stack 2
.limit locals 2
                  // --locals--   --stack--
 iload_1          //  [ o -3 ]    [ -3   * ]
 ifge Label1      //  [ o -3 ]    [  *   * ]
 iload_1          //  [ o -3 ]    [ -3   * ]
 iconst_m1        //  [ o -3 ]    [ -3  -1 ]
 imul             //  [ o -3 ]    [ 3    * ]
 ireturn          //  [ o -3 ]    ----------
Label1:
 iload_1          //
 ireturn          //
.end method
```

Sketch of Interpreter

```
pc = code.start;
while( true ) {
  npc = pc + instructionLength(code[pc]);
  switch( opcode(code[pc]) ) {
    case ILOAD_1: push(local[1]);
                  break;
    case ILOAD:   push(local[code[pc+1]]);
                  break;
    case ISTORE:  t = pop();
                  local[code[pc+1]] = t;
                  break;
    case IADD:



    case IFEQ:    t = pop();
                  if( t == 0 ) npc = code[pc+1];
                  break;
    ...
  }
  pc = npc;
}
```

Unary arithmetic operations:

```
ineg        [...:i] -> [...:-i]
i2c         [...:i] -> [...:i%65536]
```

Binary arithmetic operations:

```
iadd        [...:i1:i2] -> [...:i1+i2]
isub        [...:i1:i2] -> [...:i1-i2]
imul        [...:i1:i2] -> [...:i1*i2]
idiv        [...:i1:i2] -> [...:i1/i2]
irem        [...:i1:t2] -> [...:i1%i2]
```

Direct operations:

```
iinc k a    [...]  -> [...]
            local[k]=local[k]+a
```

what happens to stack

Nullary branch operations:

```
goto L          [...]  -> [...]
                branch always
```

Unary branch operations:

```
ifeq L          [...:i] -> [...]
                branch if i == 0
ifne L          [...:i] -> [...]
                branch if i != 0

ifnull L        [...:o] -> [...]
                branch if o == null
ifnonnull L     [...:o] -> [...]
                branch if o != null
```

Constant loading operations:

```
iconst_0        [...]  -> [...:0]
iconst_1        [...]  -> [...:1]
iconst_2        [...]  -> [...:2]
iconst_3        [...]  -> [...:3]
iconst_4        [...]  -> [...:4]
iconst_5        [...]  -> [...:5]

aconst_null     [...]  -> [...:null]

ldc_int i       [...]  -> [...:i]
ldc_string s    [...]  -> [...:String(s)]
```

Locals operations:

```
iload k          [...]  -> [...:local[k]]
istore k         [...:i] -> [...]
                 local[k]=i

aload k          [...]  -> [...:local[k]]
astore k         [...:o] -> [...]
                 local[k]=o
```

Field operations:

```
getfield f sig   [...:o] -> [...:o.f]
putfield f sig   [...:o:v] -> [...]
                 o.f=v
```

Stack operations:

```
dup             [...:v1] -> [...:v1:v1]
pop             [...:v1] -> [...]
swap            [...:v1:v2] -> [...:v2:v1]
nop             [...]  -> [...]
```

Class operations:

```
new C              [...]  -> [...:o]

instance_of C      [...:o] -> [...:i]
                   if (o==null) i=0
                   else i=(C<=type(o))

checkcast C        [...:o] -> [...:o]
                   if (o!=null && !C<=type(o))
                   throw ClassCastException
```

Method operations:

```
invokevirtual m sig
      [...:o:a1:...:an] -> [...]

entry=lookup(o.class);
block=select(entry,,m,sig);
push stack frame of size
      block.locals+block.stacksize;
local[0]=o;
local[1]=a1;
...
local[n]=an;
pc=block.code;
```

Method operations:

```
invokenonvirtual m sig
     [...:o:a1:...:an] -> [...]

entry=lookup(o.class);
block=entry.index(m,sig);
push stack frame of size
     block.locals+block.stacksize;
local[0]=o;
local[1]=a1;
...
local[n]=an;
pc=block.code;
```

Overview of Runtime Lectures

- **Expression Evaluation:**
  - organize computing values of expressions
  - taking care of intermediate results
- **Routines:**
  - implement procedures, operators
  - pass parameters and return values
- **Data Representation:**
  - represent values of source language
- **Storage Allocation**:
  - organize storage for variables (considering different lifetimes of global, local and heap variables)

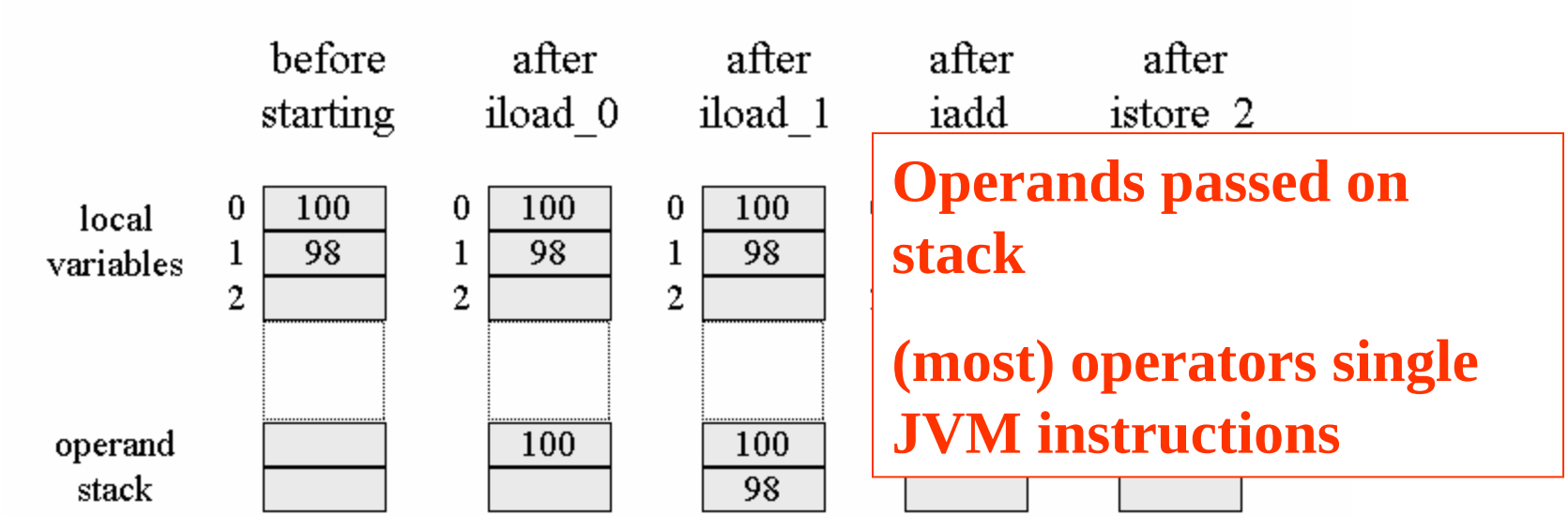
- JVM is
  - a stack machine
  - typed instructions
  - garbage collected object-level memory

How does this answer these?

picture from Artima.com

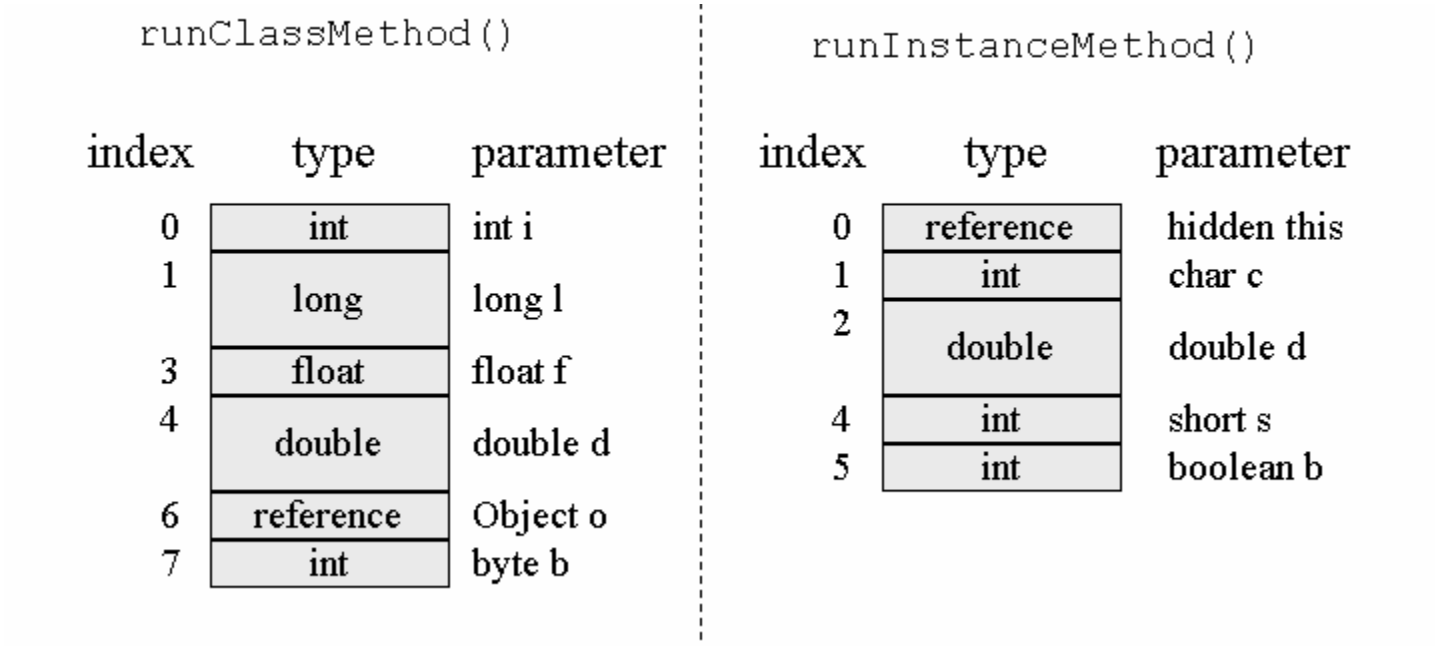
```
static void foo(int x, int y) {
  int c = x + y;
  ...
}
```

```
iload_0  // push the int in local variable 0
iload_1  // push the int in local variable 1
iadd     // pop two ints, add them, push result
istore_2 // pop int, store into local variable 2
```



picture from Artima.com

```
public static int runClassMethod(int i, long l,
            float f, double d, Object o, byte b) {
  return 0;
}

public int runInstanceMethod(char c, double d,
short s, boolean b) {
  return 0;
}
```



compiler MAY share space for locals…

```
public static void runtwoLoops() {
  for (int i = 0; i < 10; ++i) {
    System.out.println(i);
  }

  for (int j = 9; j >= 0; --j) {
    System.out.println(j);
  }
}
```

Class operations:

```
new C              [...]  -> [...:o]

instance_of C      [...:o] -> [...:i]
                   if (o==null) i=0
                   else i=(C<=type(o))

checkcast C        [...:o] -> [...:o]
                   if (o!=null && !C<=type(o))
                   throw ClassCastException
```

Method operations:

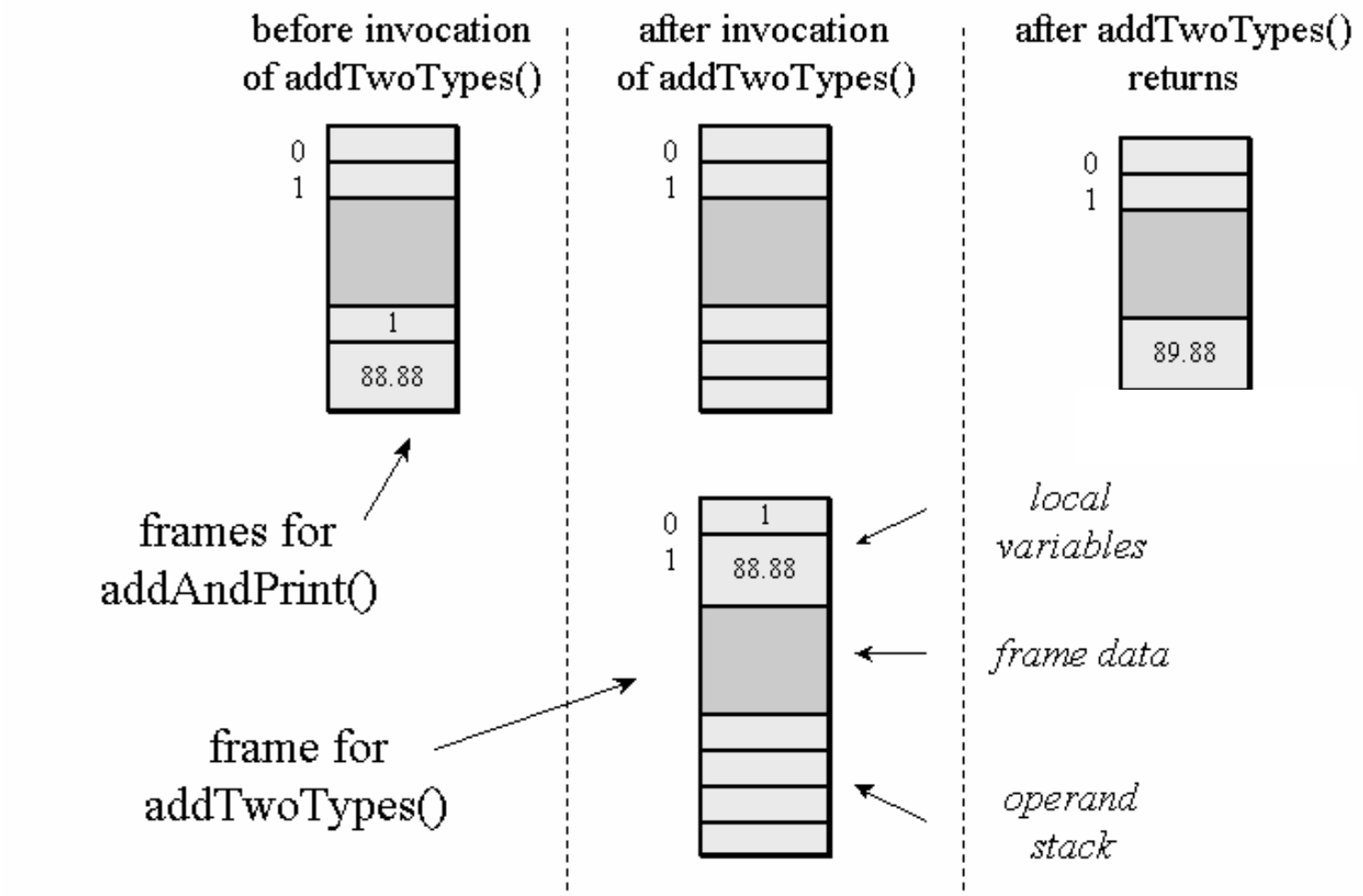
```
invokevirtual m sig
      [...:o:a1:...:an] -> [...]

entry=lookup(o.class);
block=select(entry,,m,sig);
push stack frame of size
      block.locals+block.stacksize;
local[0]=o;
local[1]=a1;
...
local[n]=an;
pc=block.code;
```

Method operations:

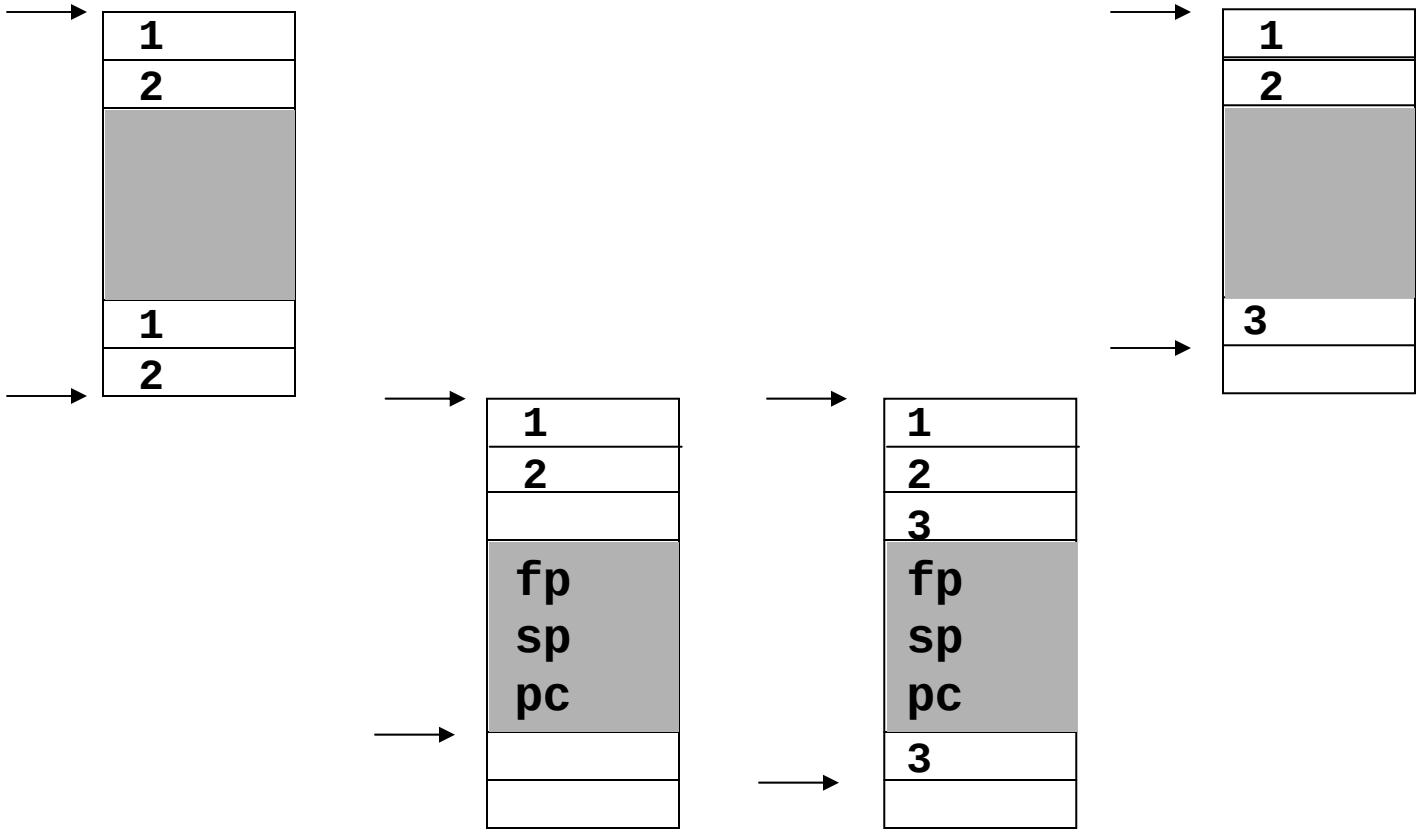
```
invokenonvirtual m sig
     [...:o:a1:...:an] -> [...]

entry=lookup(o.class);
block=entry.index(m,sig);
push stack frame of size
     block.locals+block.stacksize;
local[0]=o;
local[1]=a1;
...
local[n]=an;
pc=block.code;
```

```
public static void addAndPrint() {
  double result = addTwoTypes(1, 88.88);
  System.out.println(result);
}
public static double addTwoTypes(int i, double d) {
  return i + d;
}
```

picture from Artima.com



```
public static void foo() {
 int x = 1, y = 2;
 System.out.println(bar(x, y));
}
public static int bar(int i, int j) {
  int result = i + j;
  return result;
}
```



Sketch of Interpreter

```
int fp, sp, pc;
invoke(..find the main routine..)
while( true ) {
  npc = pc + instructionLength(code[pc]);
  switch( opcode(code[pc]) ) {
    case ILOAD_1: push(local[1]);
                  break;
    case ILOAD:   push(local[code[pc+1]]);
                  break;
    case ISTORE:  t = pop();
                  local[code[pc+1]] = t;
                  break;
    case IADD:



    case IFEQ:    t = pop();
                  if( t == 0 ) npc = code[pc+1];
                  break;
    ...
  }
  pc = npc;
}
```

Overview of Runtime Lectures

- **Expression Evaluation:**
  - organize computing values of expressions
  - taking care of intermediate results
- **Routines:**
  - implement procedures, operators
  - pass parameters and return values
- **Data Representation:**
  - represent values of source language
- **Storage Allocation**:
  - organize storage for variables (considering different lifetimes of global, local and heap variables)

- JVM is
  - a stack machine
  - typed instructions
  - garbage collected object-level memory

Register Machines

```
x + a * b

iload_1          ld [fp-32], r0
iload_2          ld [fp-64], r1
iload_3
imult            mul(r0, r1, r0)
iadd             ld [fp-0], r1
                 add(r0, r1, r1)
```

fixed set of general-purpose registers
compiled code reads/writes to them explicitly
registers are simple and fast
- **a push requires write, read/+/write**

opportunity to call *routines* in registers
- **still call procedures on a stack**

Heap Storage Allocation

RECAP: we have mentioned 2 storage allocation models so far:

- heap allocation
  - objects
  - exist “forever”
- stack allocation
  - local variables and arguments for procedures (methods)
  - lifetime follows procedure activation

NEXT:

Heap allocation – allocation of indefinite lifetime values
(C malloc, C++ / Pascal / Java new operation).

Example: (Java)

```
int[] nums = new int[computedNeededSize];
```

Heap Storage Allocation

Explicit versus Implicit Deallocation

In explicit memory management, the program must explicitly call an operation to release memory back to the memory management system.

In implicit memory management, heap memory is reclaimed automatically by a *garbage collector*.

Examples:

- Implicit: Java, Scheme
- Explicit: Pascal and C

  To free heap memory a specific operation must be called.

  Pascal ==> `dispose`

  C ==> `free`

Heap Storage Allocation

Memory managers (both implicit as well explicit) have been studied extensively.
=> algorithms for fast allocation / deallocation of memory and efficient representation (low memory overhead for memory management administration).

=> There are many complicated, sophisticated and interesting algorithms. We could dedicate an entire course to this topic alone!

We will look at memory management only superficially. Maybe at the end of the course we might cover some algorithms in detail.

Now we will look at superficially at the kind of algorithms/issues are associated with explicit and implicit memory management.

Where to put the heap?

The heap is an area of memory which is dynamically allocated.
Like a stack, it may grow and shrink during runtime.
(Unlike a stack it is not a LIFO => more complicated to manage)

In a typical programming language implementation we will have both heap-allocated and stack allocated memory coexisting.
Q: How do we allocate memory for both

A simple approach is to divide the available memory at the start of the program into two areas: stack and heap.
Another question then arises =>
- How do we decide what portion to allocate for stack vs. heap ?

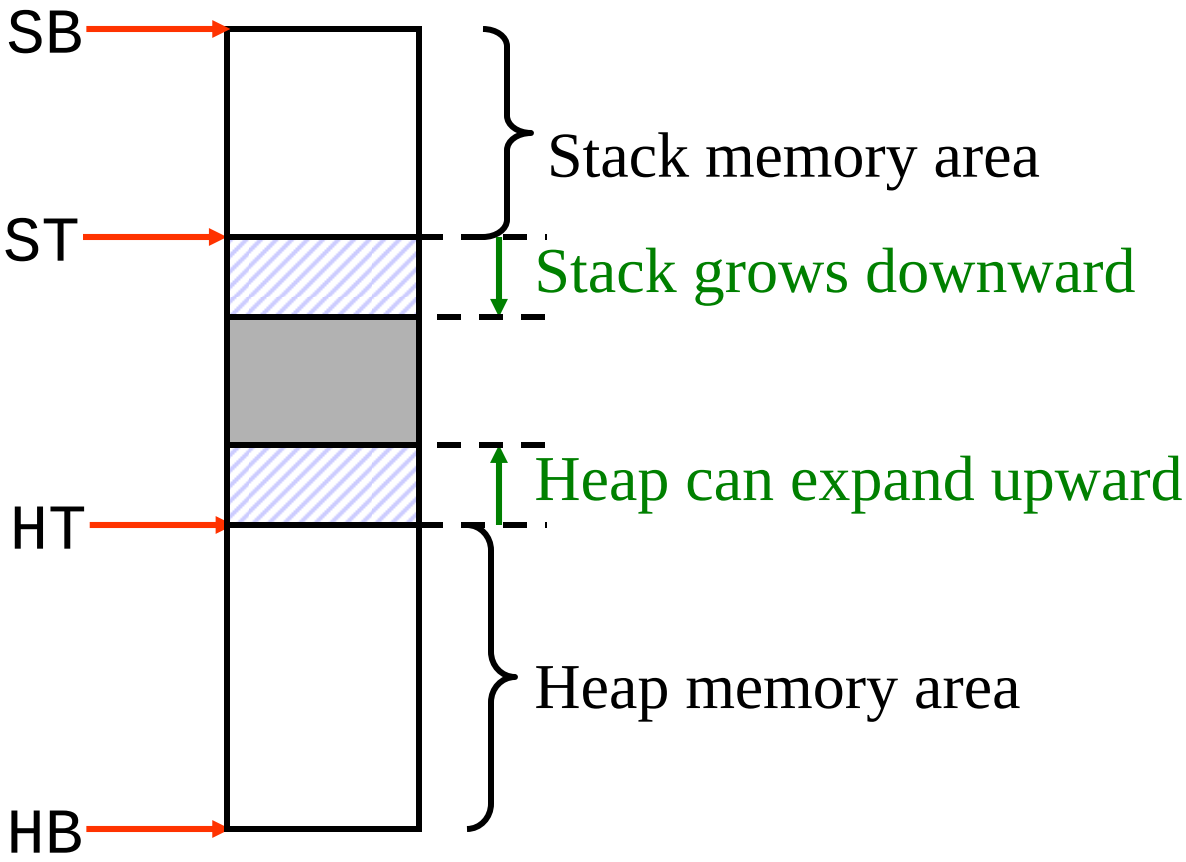
=> Issue: if one of the areas is full, then even though we still have more memory (in the other area) we will get out-of-memory errors.

Q: Isn't there a better way?

Where to put the heap?

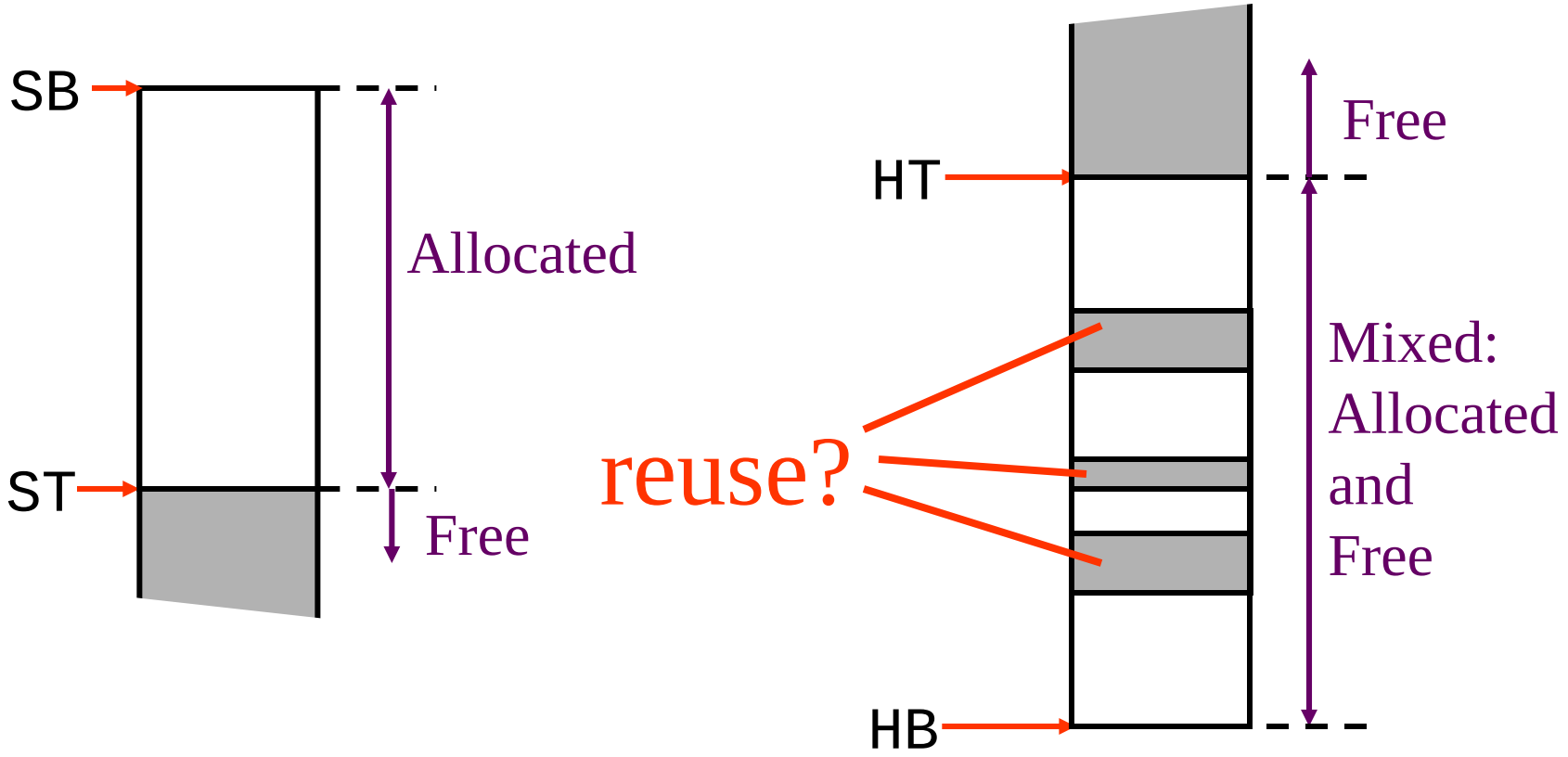
Q: Isn't there a better way?

A: Yes, there is an often used "trick": let both stack and heap share same memory area, but grow towards each other from opposite ends.



How to keep track of free memory?

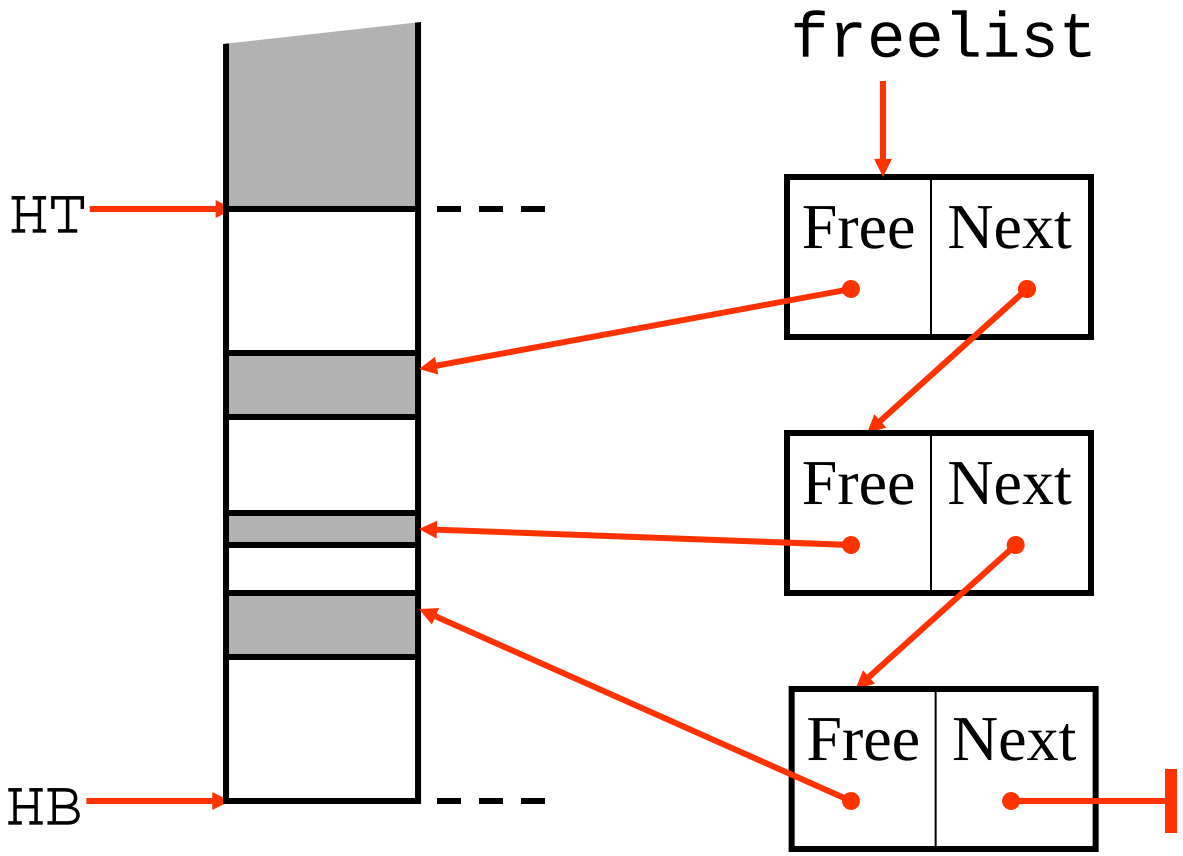
Stack is LIFO allocation => ST moves up/down everything above ST is in use/allocated. Below is free memory. This is easy! But …
Heap is not LIFO, how to manage free space in the "middle" of the heap?



How to keep track of free memory?

How to manage free space in the “middle” of the heap?

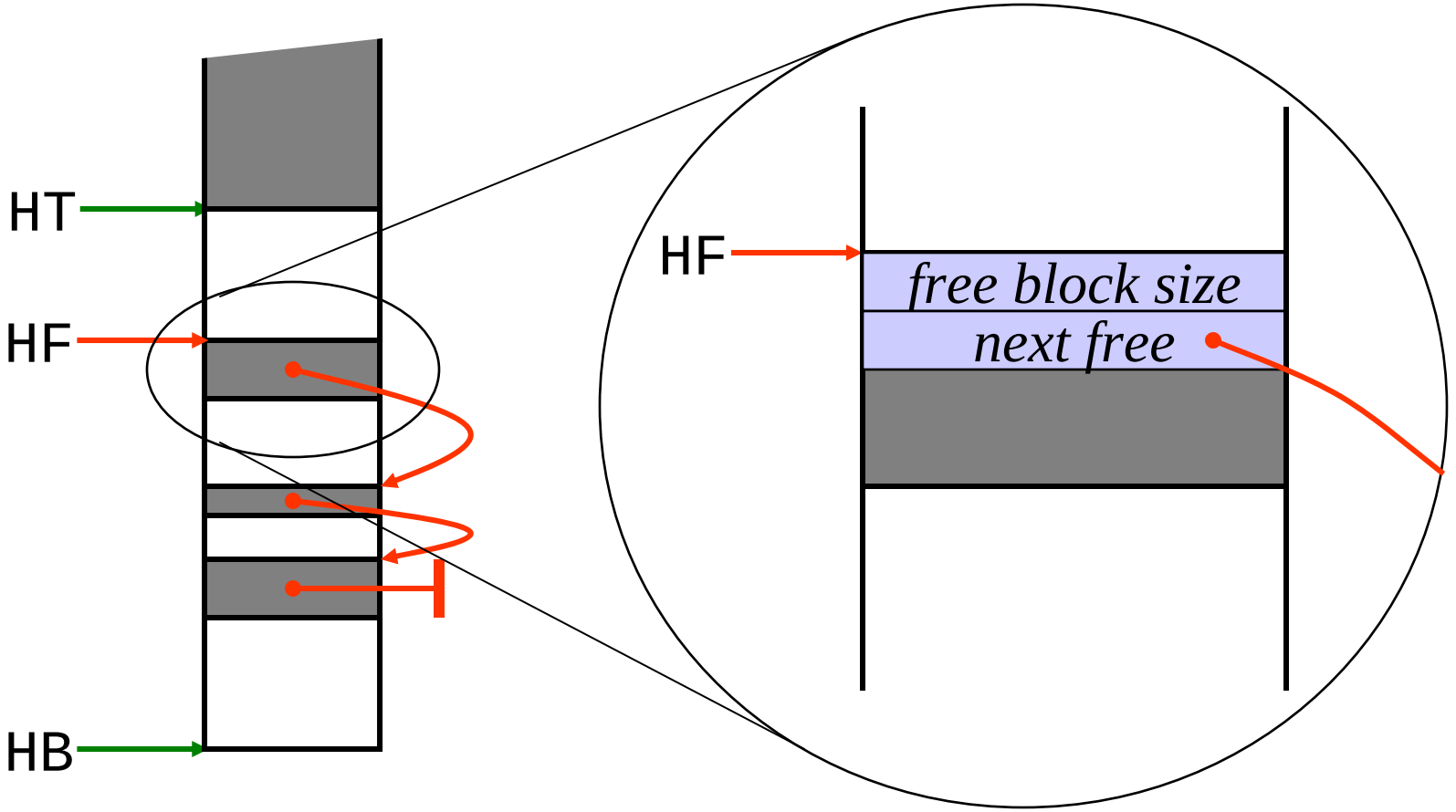
=> keep track of free blocks in a data structure: the “free list”. For example we could use a linked list pointing to free blocks.



Where should free list go?

Thread free-list through the blocks

Q: Where do we find the memory to store a freelist data structure?
A: Since the free blocks are not used for anything by the program => memory manager can use them for storing the freelist itself.



Simple Explicit Memory Manager Algorithm

Our memory manager is intended for a programming language with **explicit deallocation** of memory.

Q: What operations does the memory management library provide?

```
Pointer malloc(int size);
   ask memory manager to find and allocate a
   block of size.  Returns a pointer to
   beginning of that block.
void free(Pointer toFree,int size);
   called when a block toFree is released by
   the program and returned to the memory
   manager.
```

Simple Explicit Memory Manager Algorithm

```
Pointer malloc(int size) {
    block = search the freelist until a block of at least size is found.
    if (block is found) and (block.size == size) {
            delete block from free list.
            return pointer to start of block.
    else if (block found) and (block.size>size) {
            delete block from free list.
            insert remainder of block into free list
            return pointer to start of block.
    else  // no block found
            try to expand heap by size
            if (expansion succeeded) then
                return HT;
            else out of memory error
}
```

Simple Explicit Memory Manager Algorithm

```
void free(Pointer toFree,int size) {
   insert block(toFree,size) into the freelist
}
```

A bit more detail:

```
void free(Pointer toFree,int size) {
    toFree[0] = size; // size at offset 0
    toFree[1] = HF;   // next at offset 1
    HF = toFree;
}
```

This algorithm is rather simplistic and has some rather big problems.

Q: What are those problems? How could we fix them?

Fragmentation

One of the biggest problems with the simplistic algorithm is memory fragmentation.

- blocks are split to create smaller blocks to fit requested size.
- blocks are never merged

$\Rightarrow$blocks will get smaller and smaller.

$\Rightarrow$will run out of memory without really being out.

Better algorithm merges released blocks with neighboring blocks that are free.

=> less fragmentation

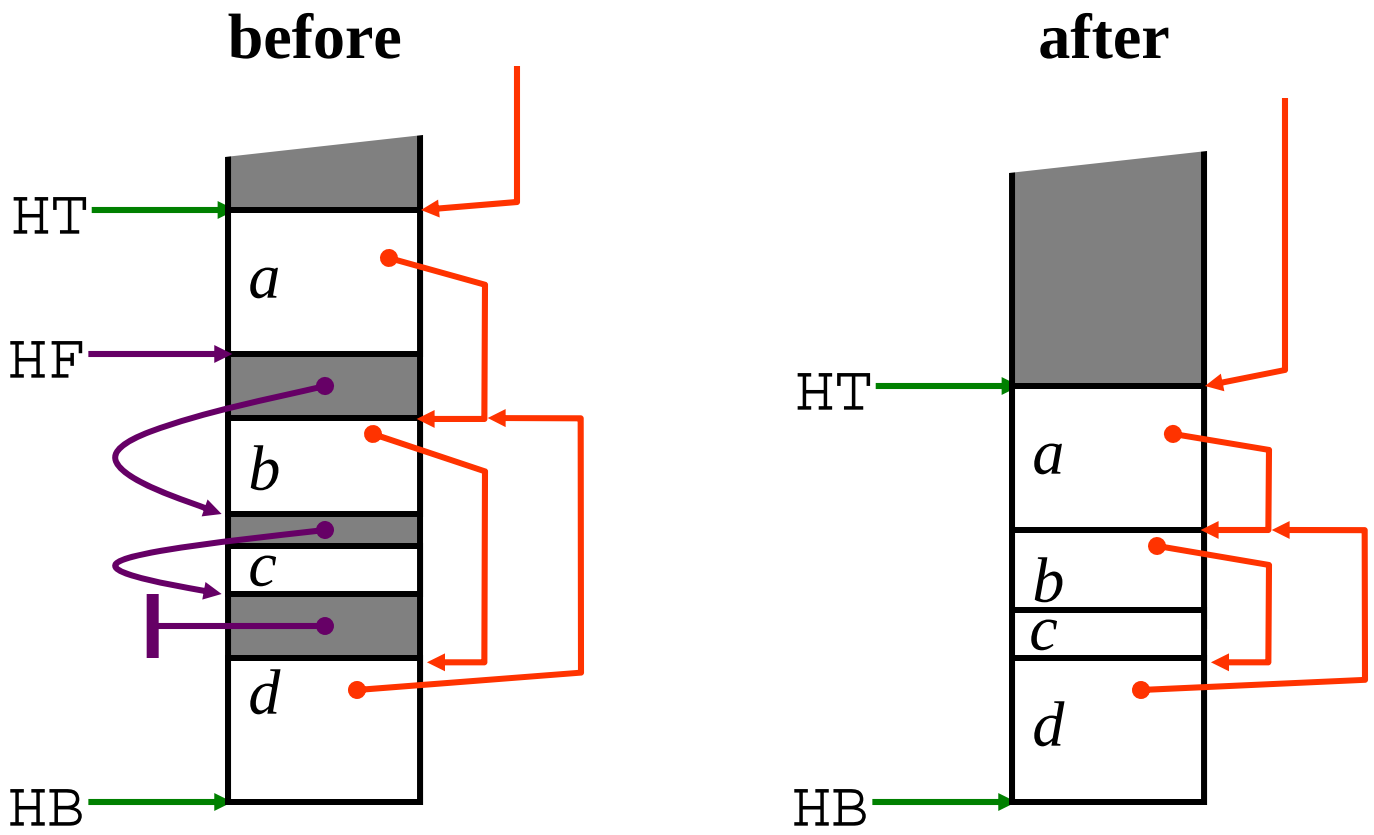
Q1: Analyze impact on the performance (time taken for) the allocation and release of blocks.

Q2: Does this solve the fragmentation problem completely?

Heap Compaction

To fight fragmentation, some memory management algorithms perform **"heap compaction"** once in a while.

Q: Can compaction be done for all programming languages?



Complication with Explicit Memory Allocation

Problems with explicit memory allocation algorithms:

Two common, nasty kinds of bugs in programs using explicit memory management are

- Memory leaks
- Dangling pointers

Q1: What are memory leaks and dangling pointers?
Q2: Is it true that there can be no memory leaks in a garbage-collected language?
Q3: Is it true that there can be no dangling pointer in a garbage-collected language.
Q4: What is a weak reference?

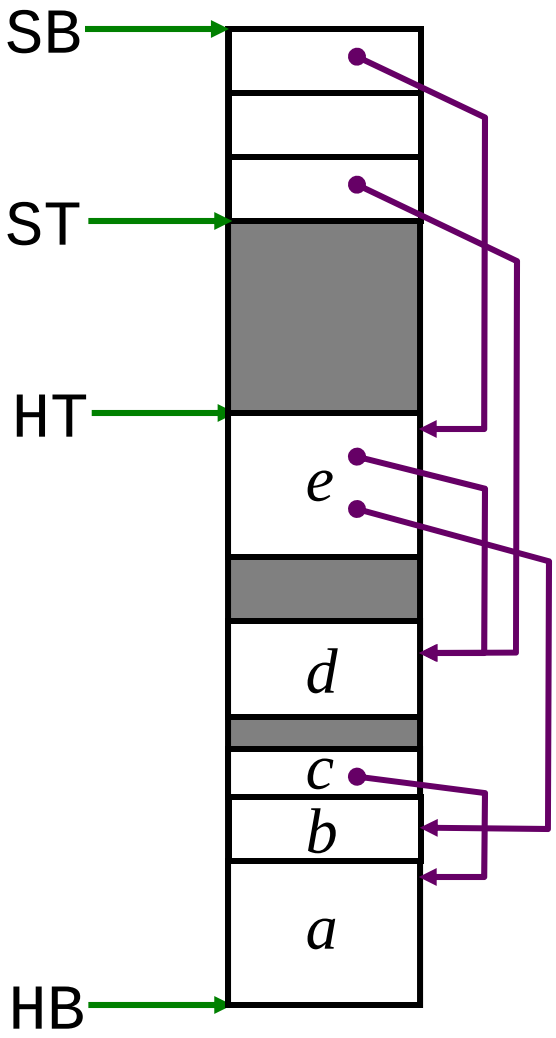
Automatic Storage Deallocation

Everybody probably knows what a garbage collector is.

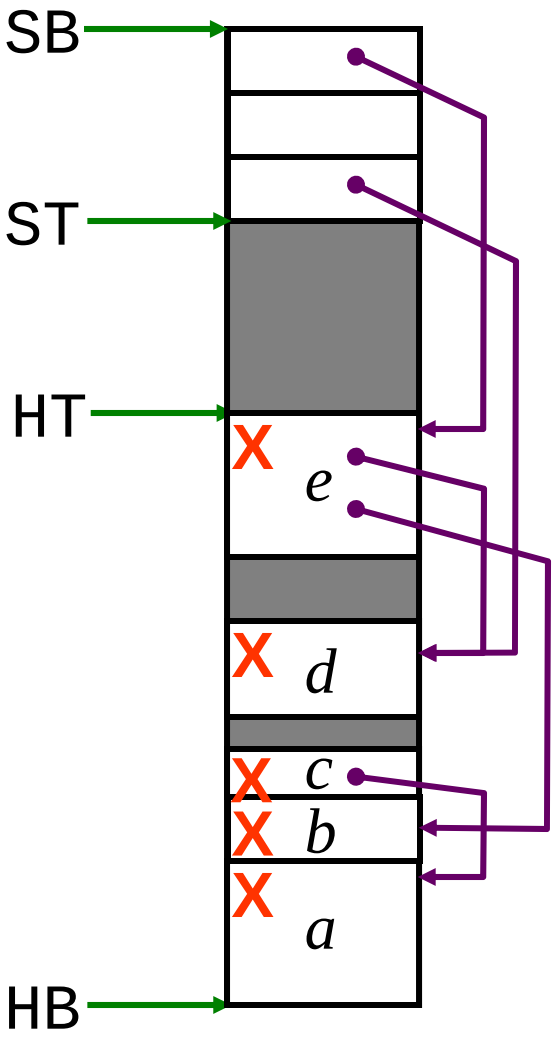
Find data (objects) which are no longer referenced.
Reclaim that memory.

Mark and Sweep Garbage Collection

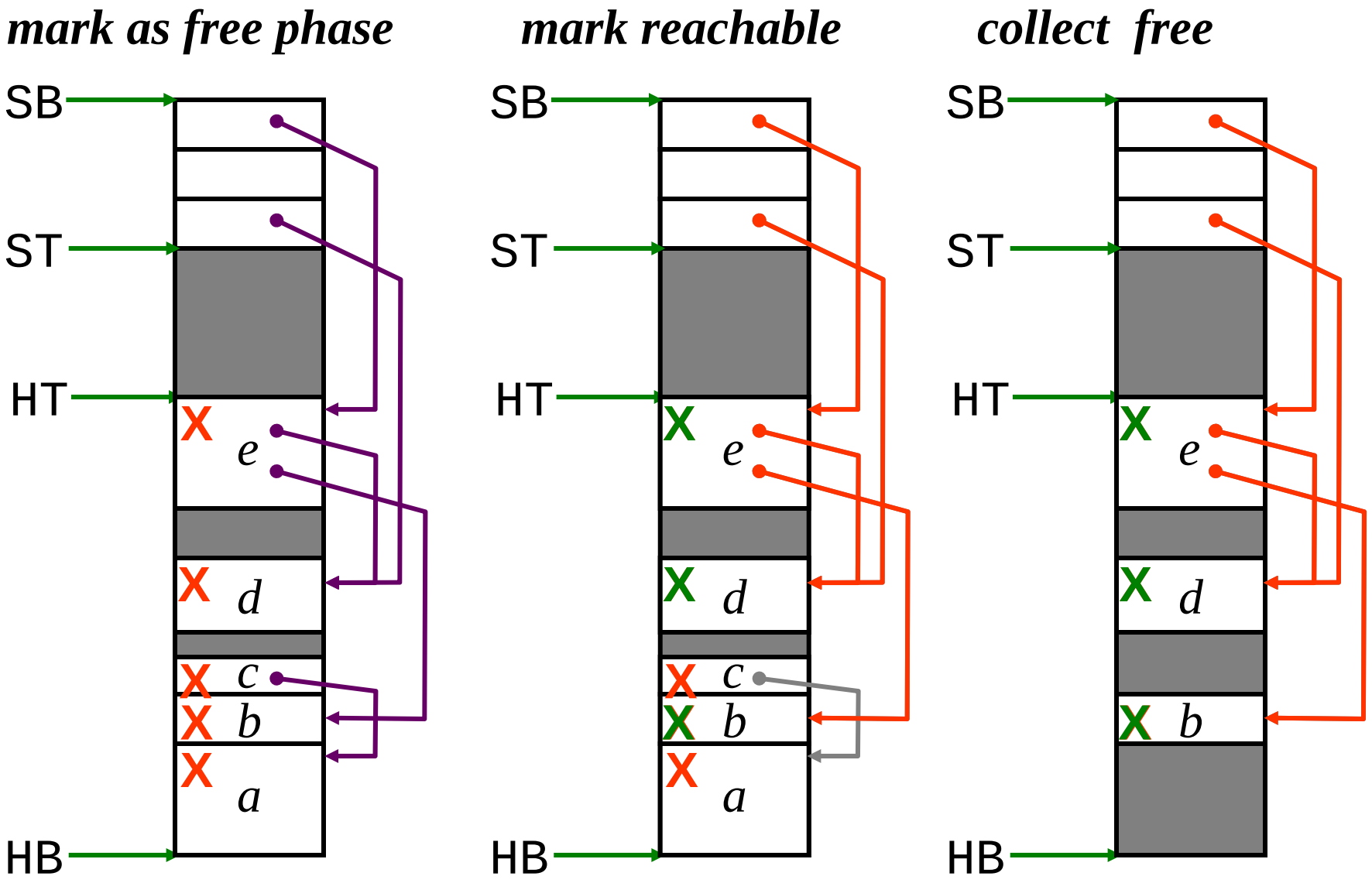
before gc



mark as free phase



Mark and Sweep Garbage Collection



Mark and Sweep Garbage Collection

Algorithm pseudo code:

```
void garbageCollect() {
    mark all heap variables as free
    for each frame in the stack
        scan(frame)
    for each heapvar (still) marked as free
        add heapvar to freelist
}
void scan(region) {
    for each pointer p in region
        if p points to region marked as free then
            mark region at p as reachable
            scan(region at p )
}
```

Q: This algorithm is recursive. What do you think about that?

Other Garbage Collection Schemes

- Incremental
  - do a little at a time
- Generational
  - young “objects” a more likely to be garbage
- Copying
  - move (and compact) objects during collection
- Program directed
  - programmer suggests what pattern object lifetime will follow