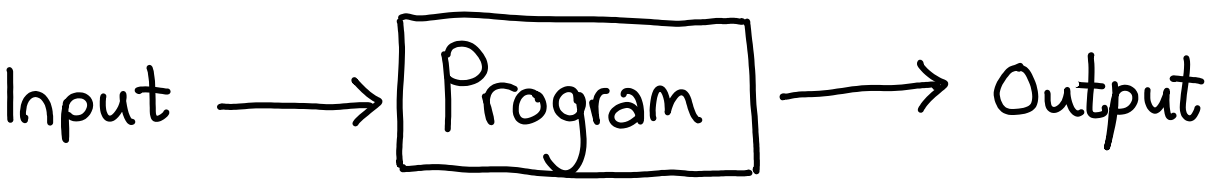
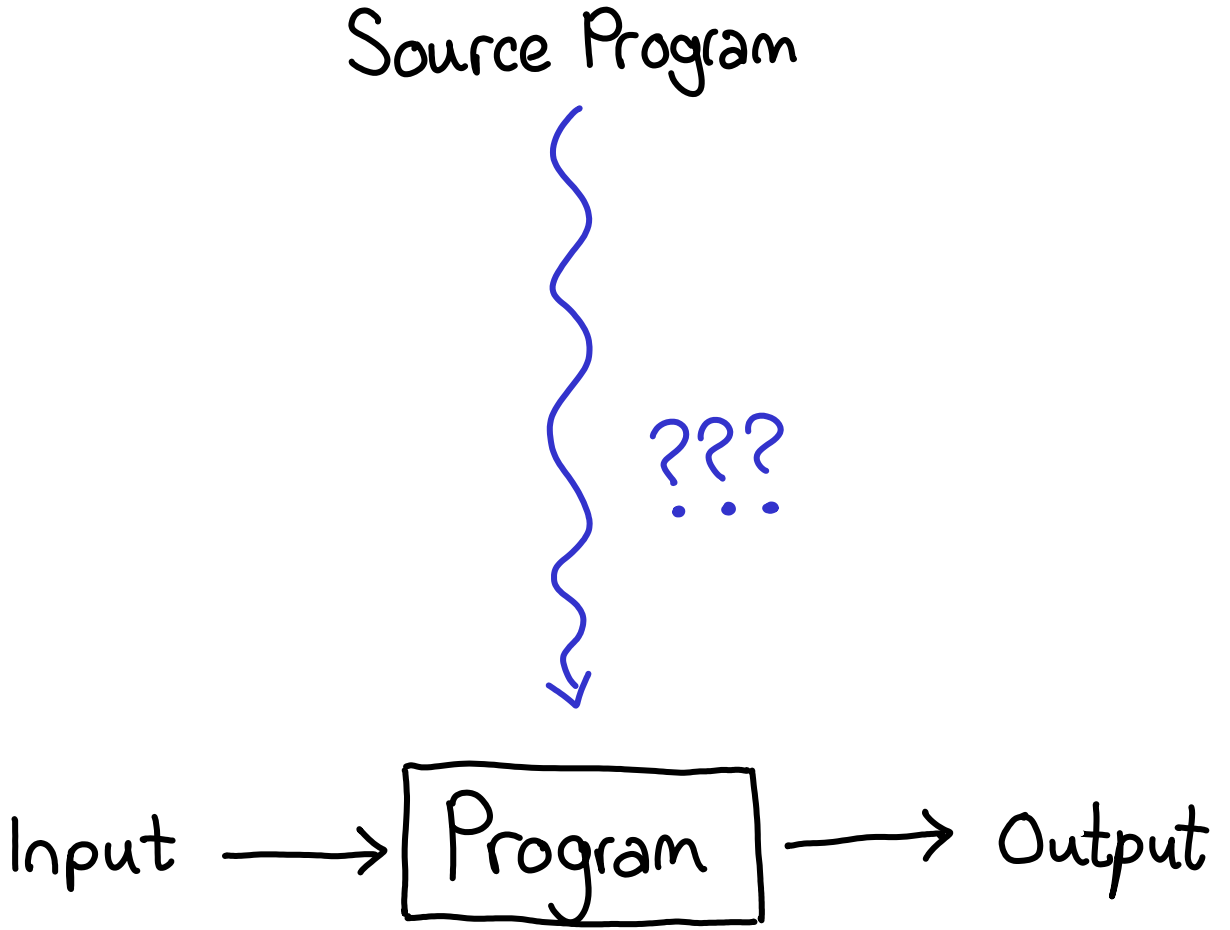
Implementation

Edward Z. Yang

(intro)


Input
Program
Output

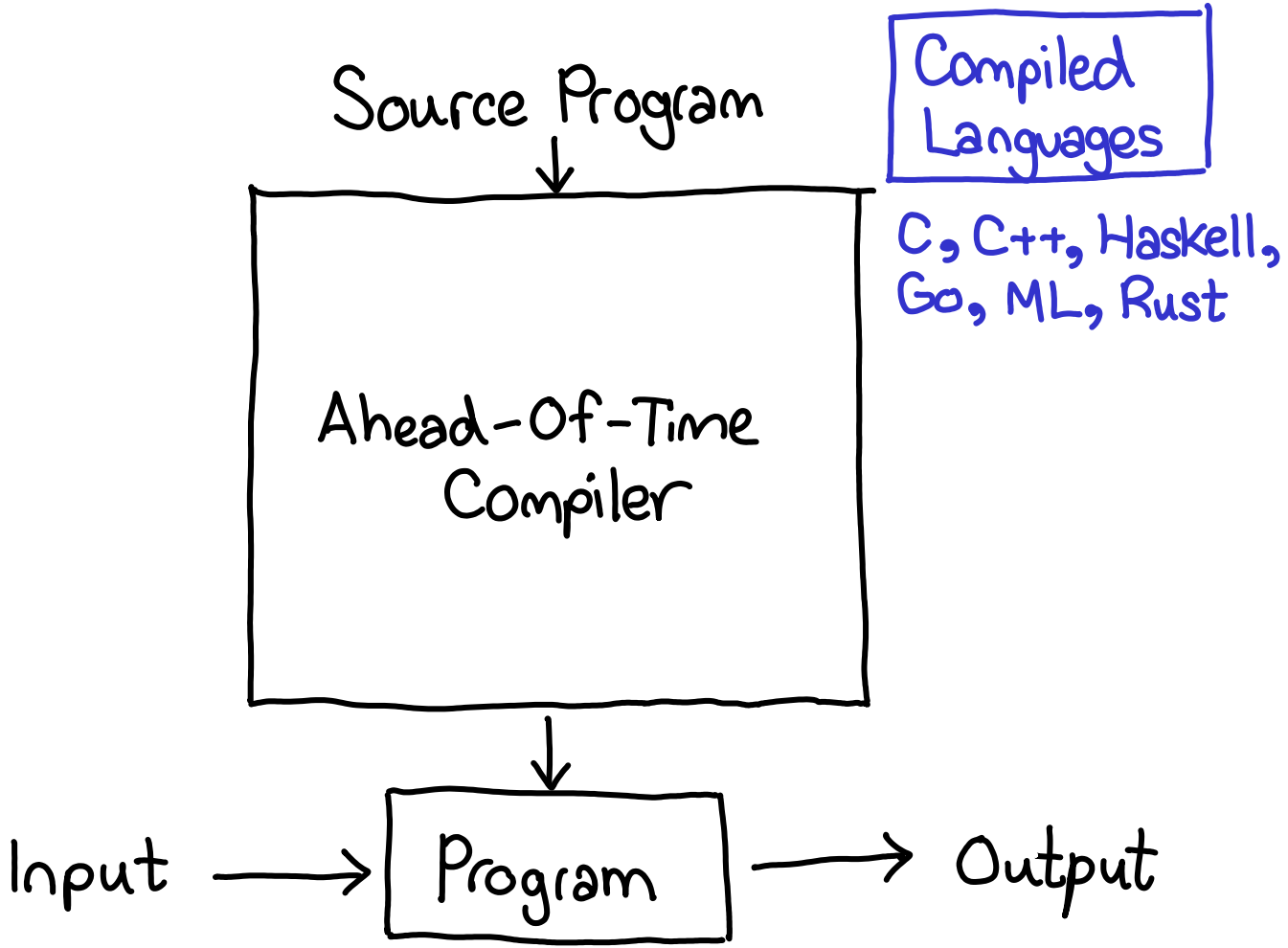

Source Program
???
Input
Program
Output

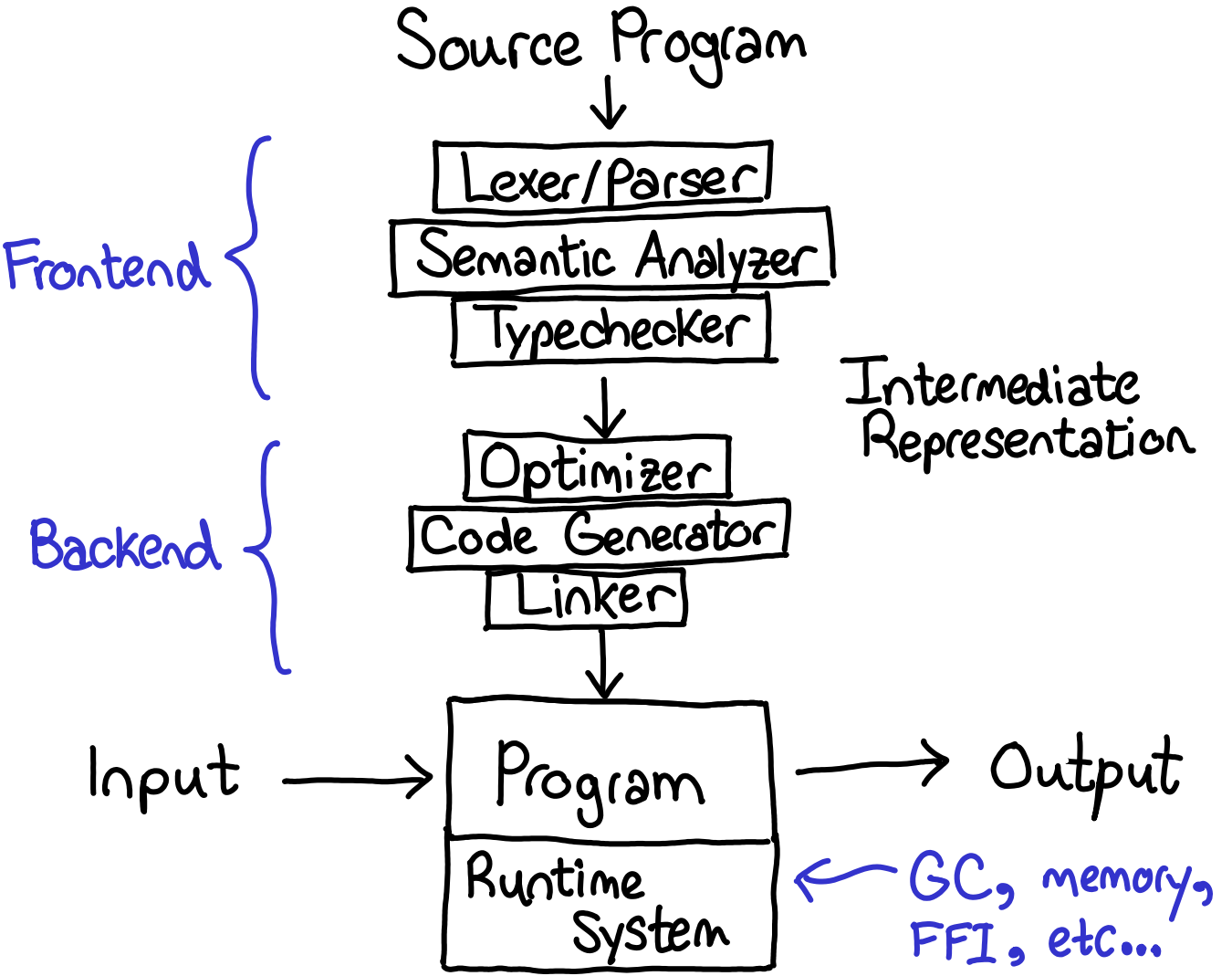
Source Program

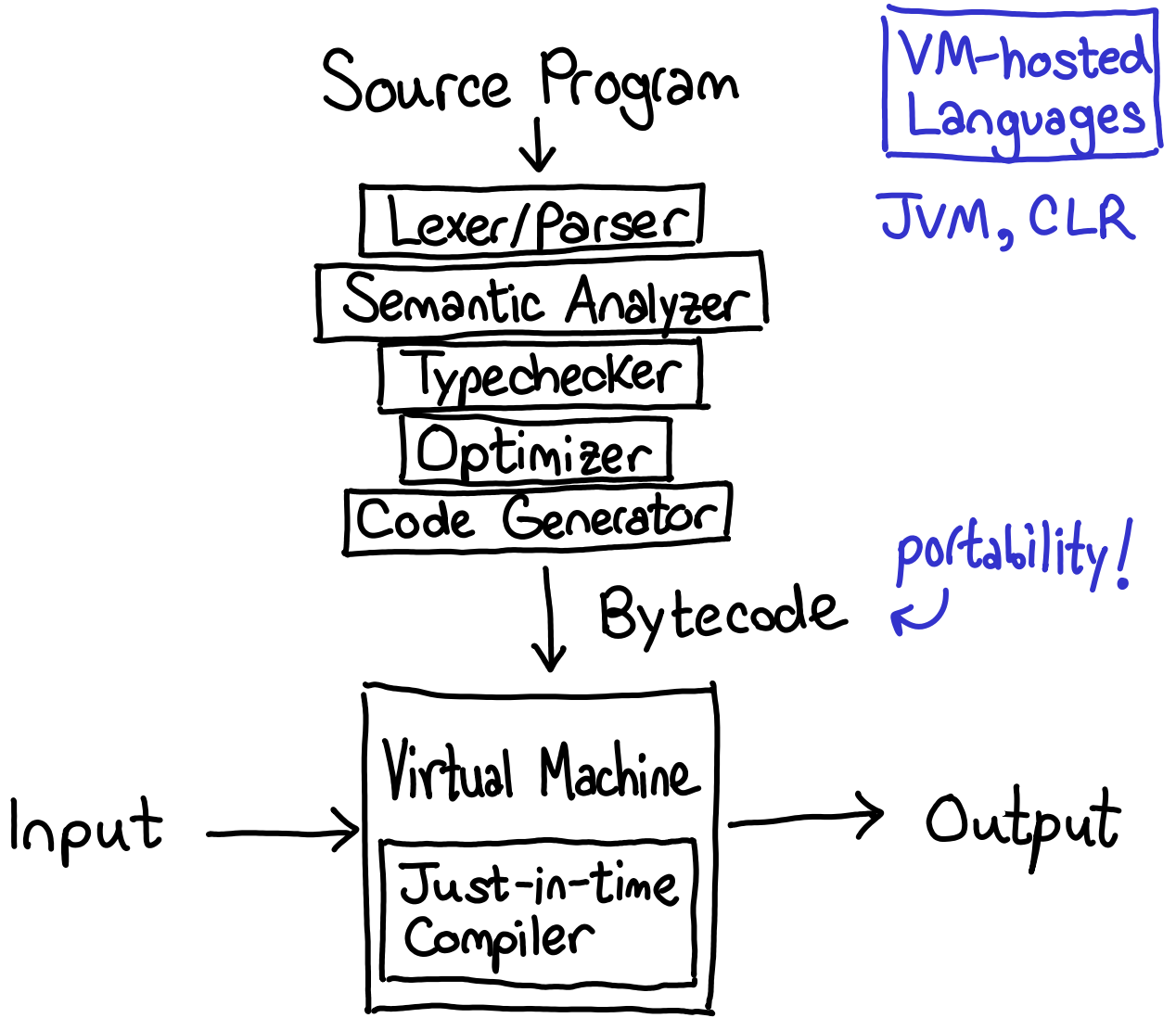
Input → Interpreter → Output

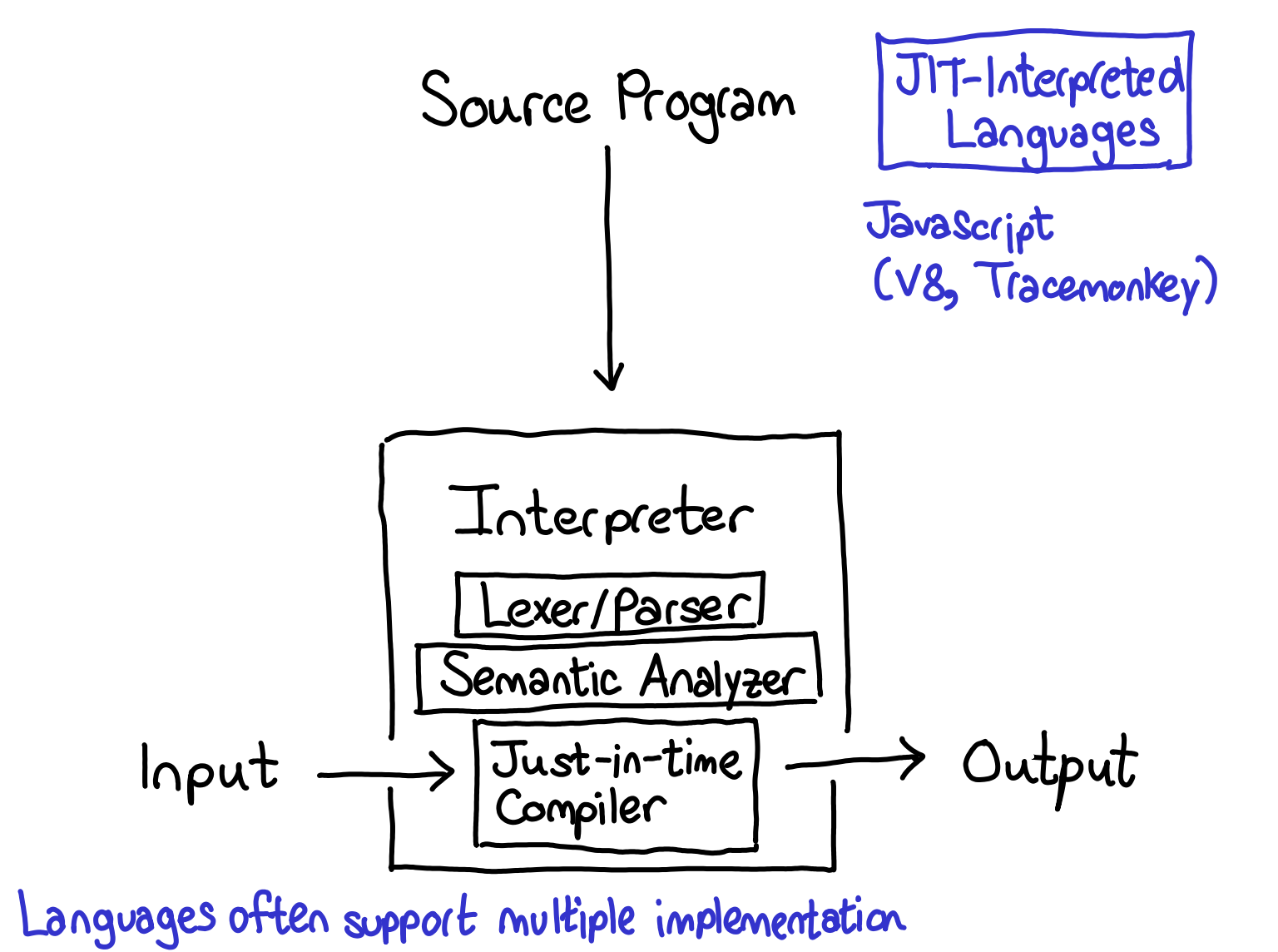
Interpreted Languages

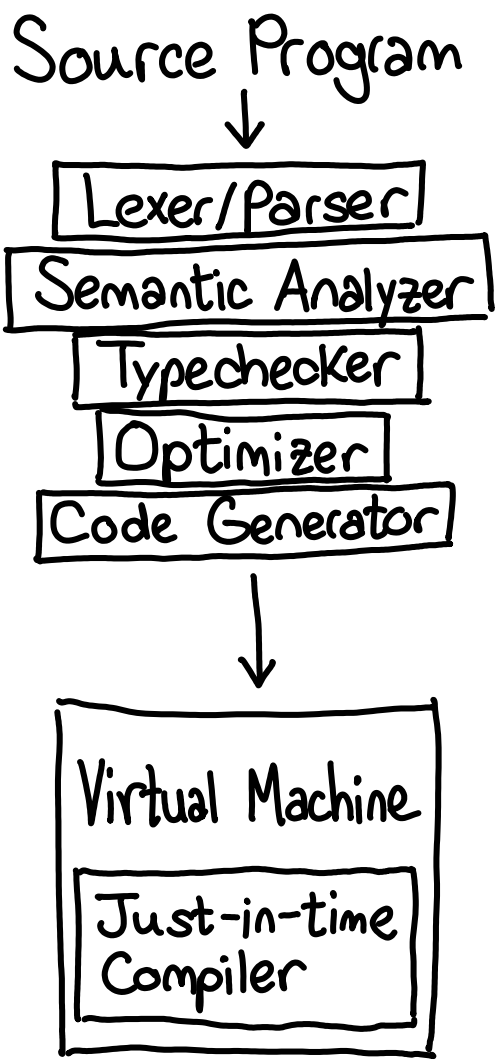
Python, Ruby, PHP, Perl, MATLAB...

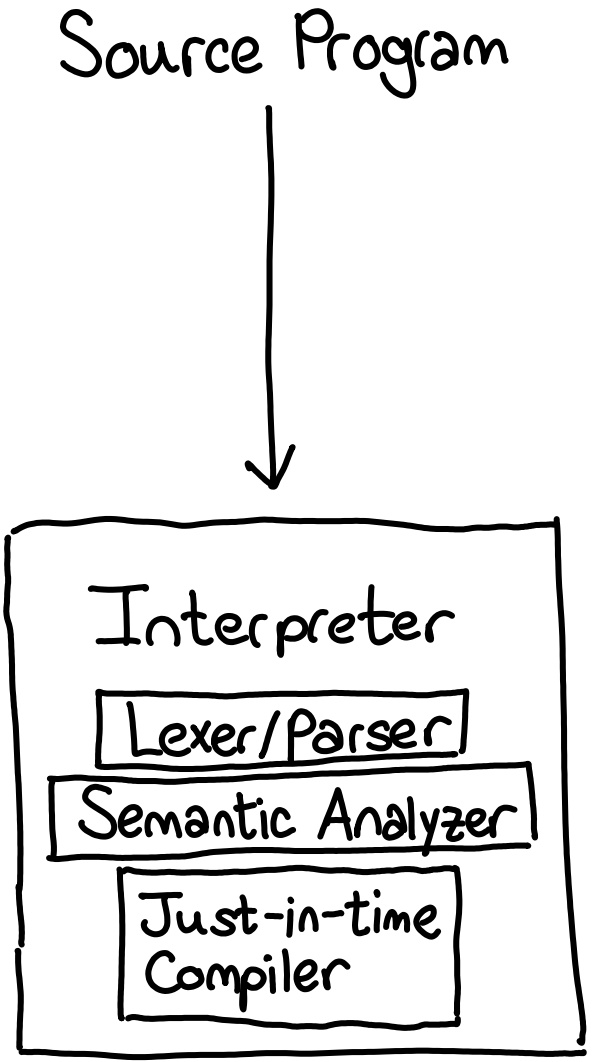

Source Program
Compiled Languages
C, C++, Haskell, Go, ML, Rust
Ahead-Of-Time Compiler
Input
Program
Output

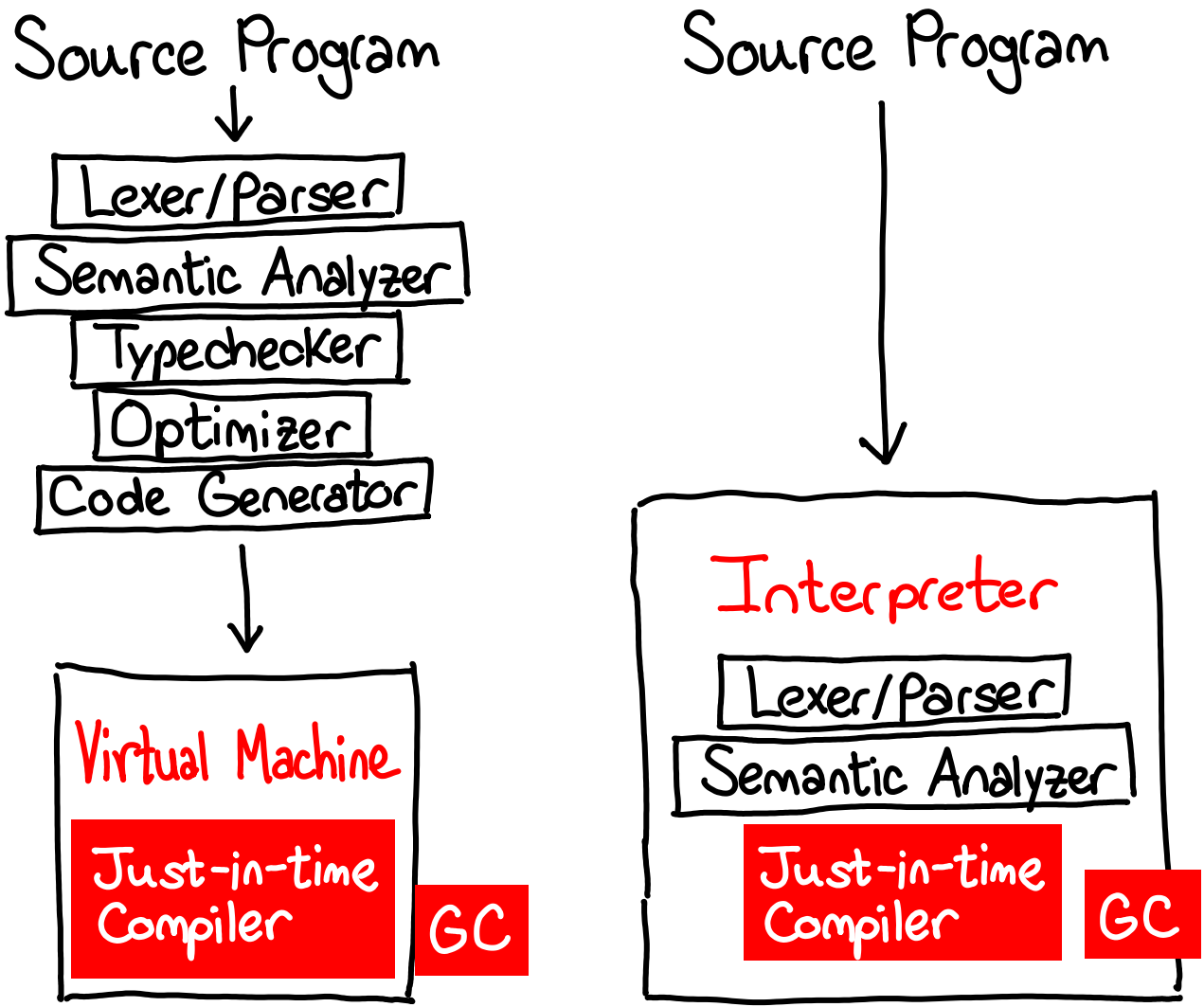

Source Program
Lexer/Parser
Semantic Analyzer
Typechecker
Frontend
Intermediate Representation
Optimizer
Code Generator
Linker
Backend
Input
Program
Output
Runtime System
GC, memory, FFI, etc...


Source Program
Lexer/Parser
Semantic Analyzer
Typechecker
Optimizer
Code Generator
Bytecode
Virtual Machine
Just-in-time Compiler
Input
Output
VM-hosted Languages
JVM, CLR
portability!


Source Program
JIT-Interpreted Languages
Javascript (V8, Tracemonkey)
Interpreter
Lexer/Parser
Semantic Analyzer
Input
Just-in-time Compiler
Output
Languages often support multiple implementation


Source Program
Lexer/Parser
Semantic Analyzer
Typechecker
Optimizer
Code Generator
Virtual Machine
Just-in-time Compiler

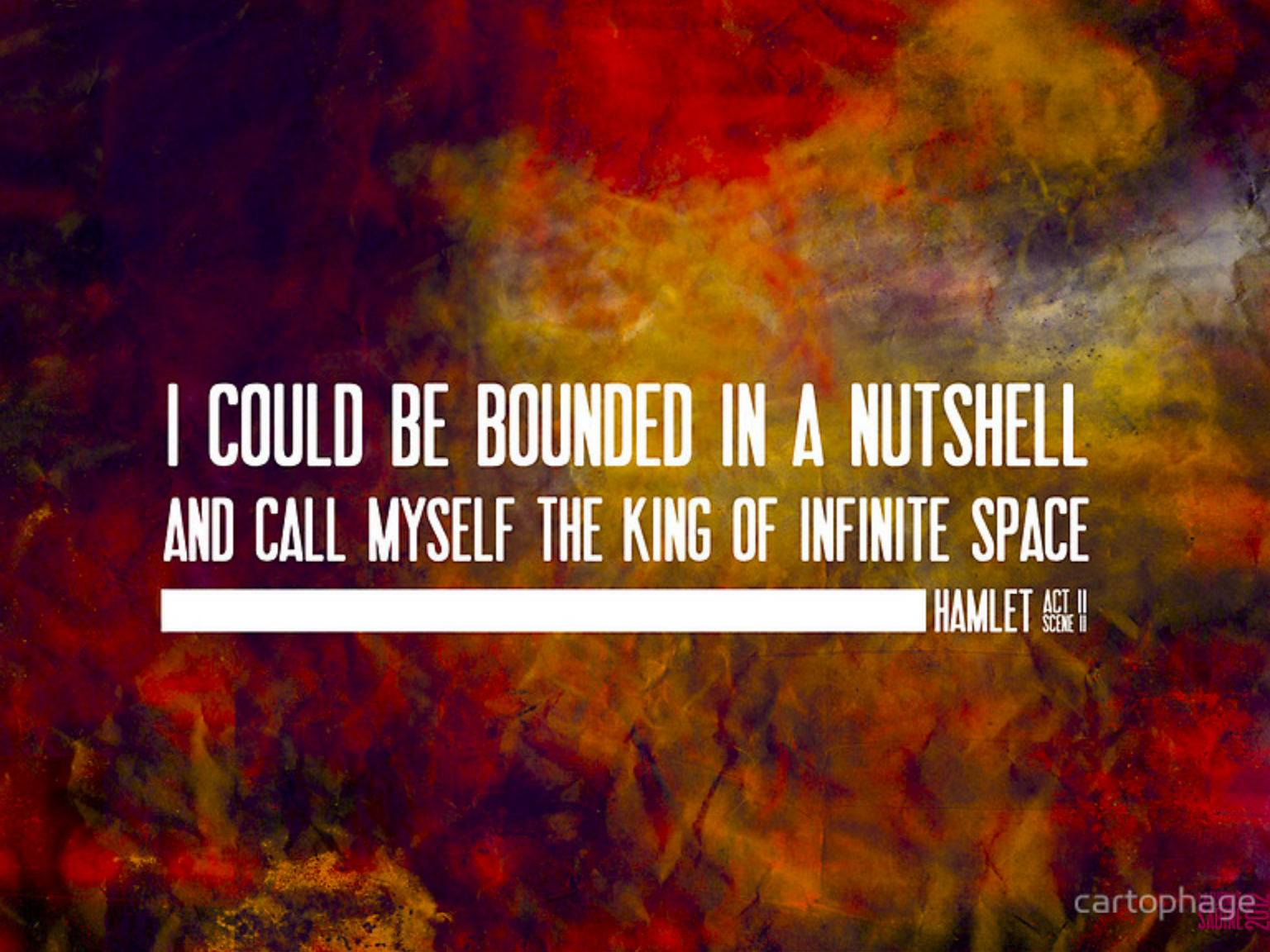

Source Program
Interpreter
Lexer/Parser
Semantic Analyzer
Just-in-time Compiler


Source Program
Lexer/Parser
Semantic Analyzer
Typechecker
Optimizer
Code Generator
Virtual Machine
Just-in-time Compiler
GC
Source Program
Interpreter
Lexer/Parser
Semantic Analyzer
Just-in-time Compiler
GC

- Garbage Collection

- Dynamic Dispatch in a JIT

  compare with C++ vtables


I COULD BE BOUNDED IN A NUTSHELL
AND CALL MYSELF THE KING OF INFINITE SPACE
HAMLET ACT II SCENE II
cartophage

Lambda Calculus

Activation-record Model

x86 machine architecture

Memory Hierarchy

INFINITE MEMORY

FINITE MEMORY

INFINITE MEMORY

GARBAGE COLLECTION

FINITE MEMORY

& reuse space which provably will never be used again

Managed Memory

Memory Management

allocate

Interface for allocating objects.
Pointers are opaque.

Garbage collection / Reference counting

malloc
free

Interface for explicitly allocating/
deallocating finite memory

Operating System & Hardware

Garbage Collection

- Reference Counting
  - ARC, Perl, PHP, Python
  - The Cycle Problem

- Tracing Collection
  - Java, Haskell, ML, Lisp, Go, JavaScript
    - Mark and Sweep
    - Copying Collection

When I am done using an object, free its memory

How do I know this?

When I am done using an object,

free its memory

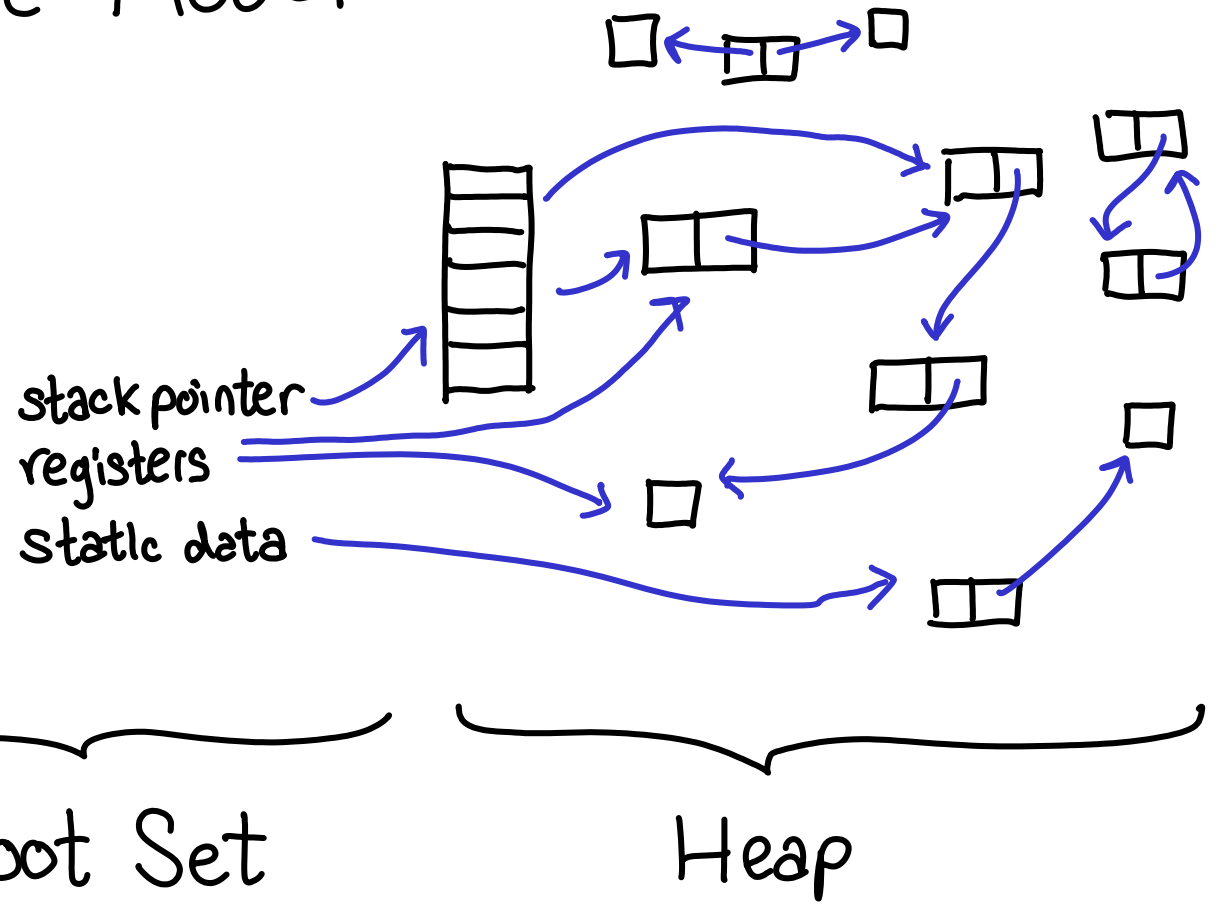
IDEAL

Object has no causal influence on future program execution

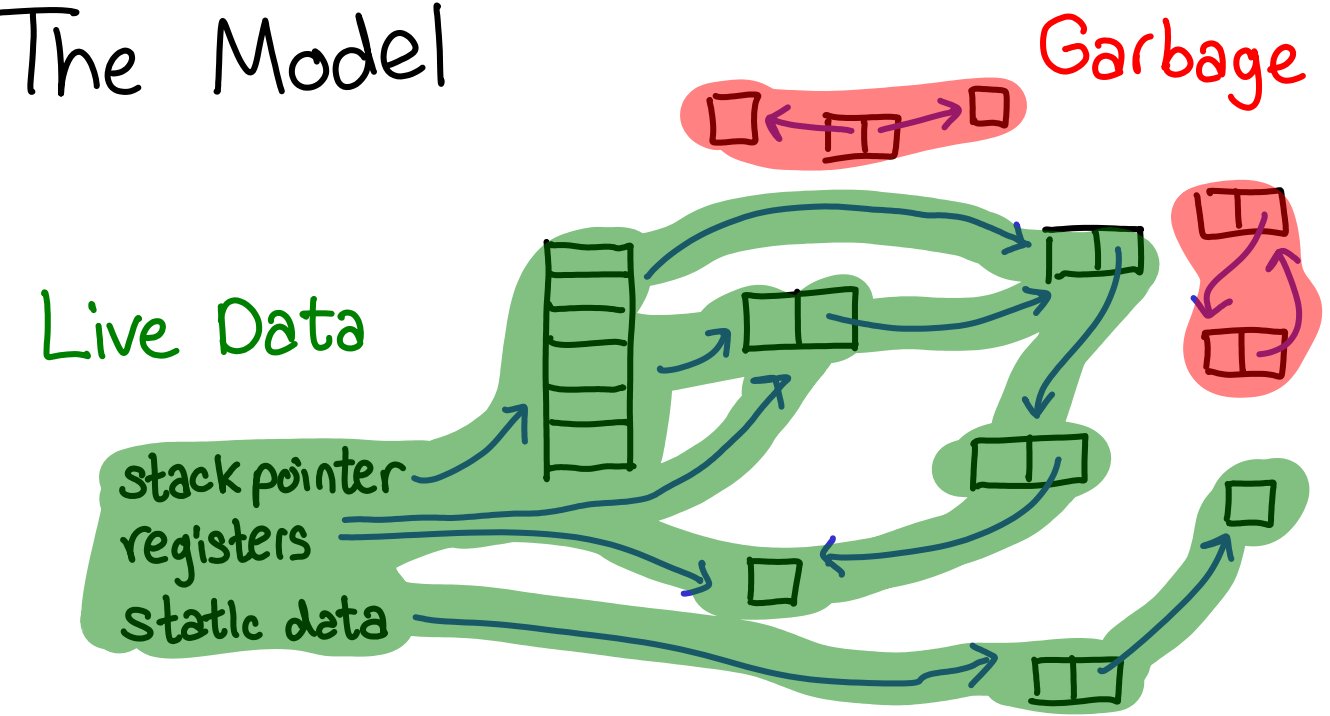
When I am done using an object,

free its memory

The Model



The Model



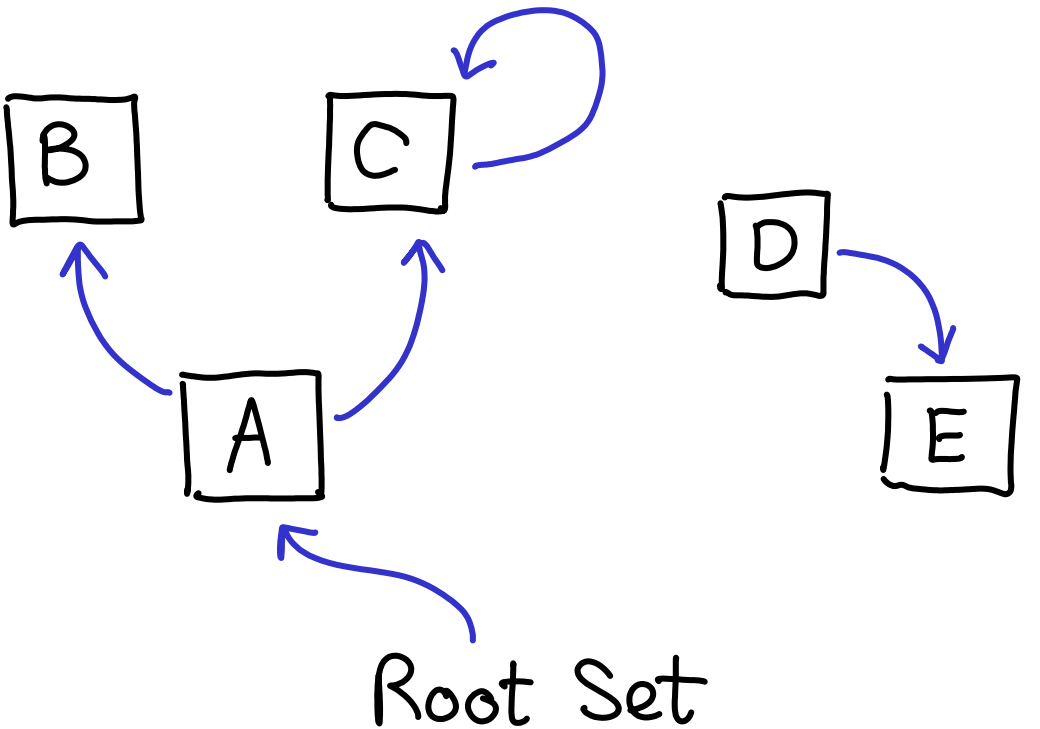
Root Set

Heap

Why must pointer arithmetic be disallowed?

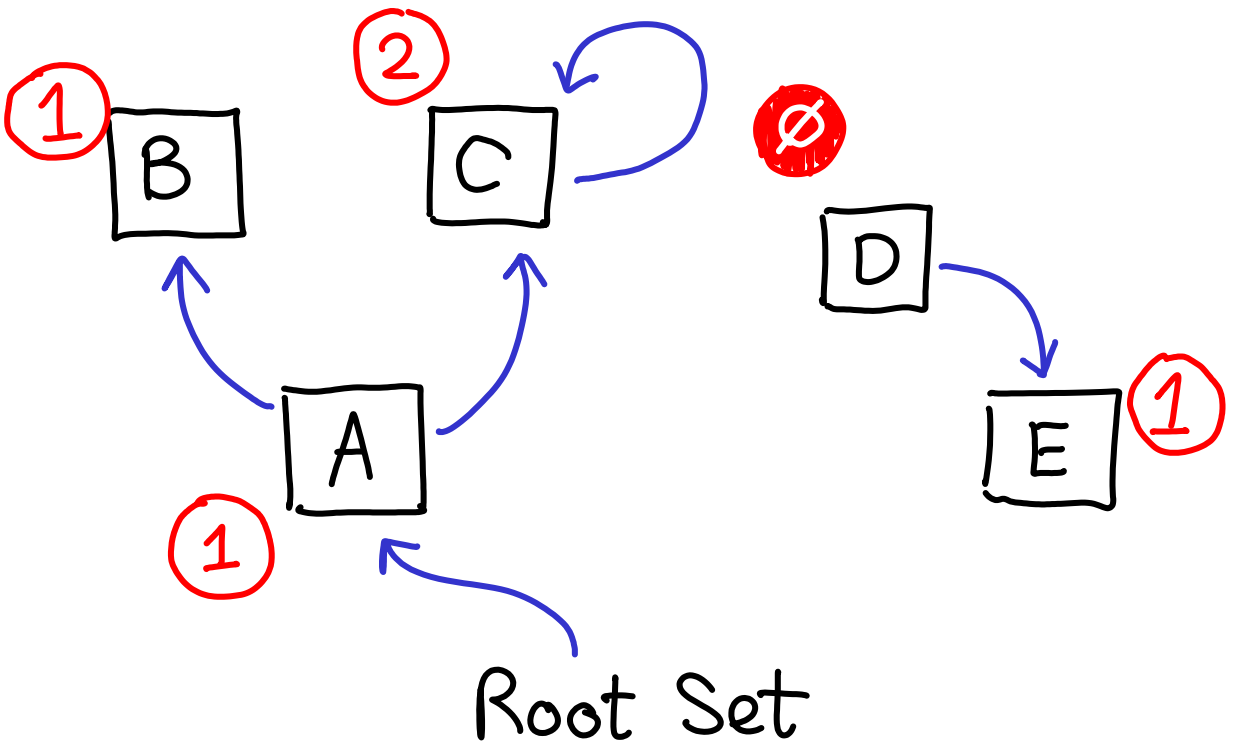
Reference counting

Count the number of incoming references



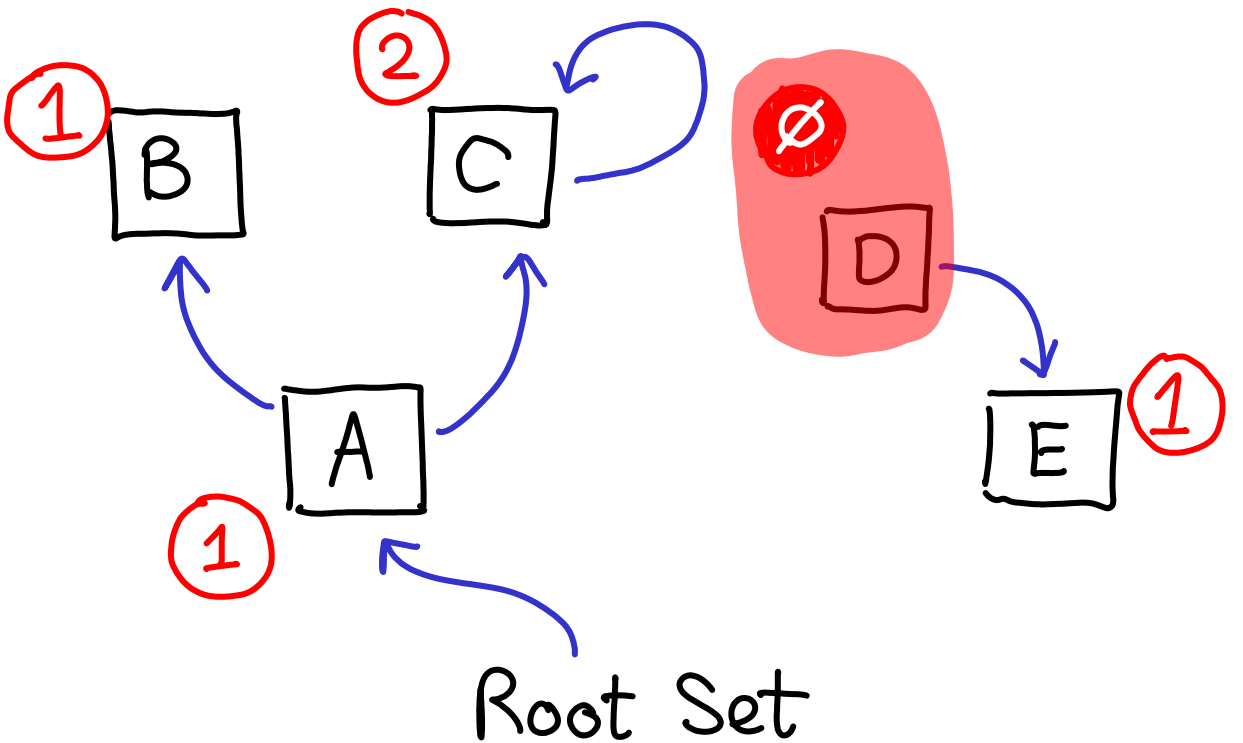
Reference counting

Count the number of incoming references



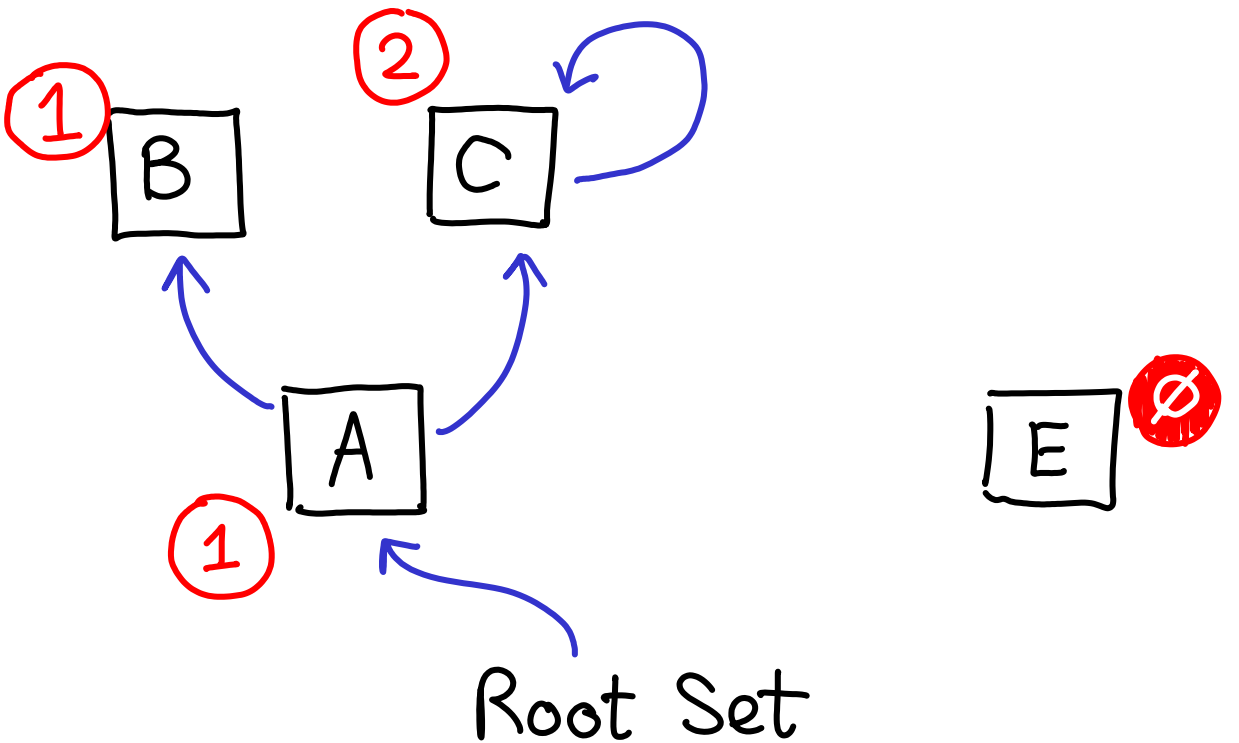
Reference counting

Count the number of incoming references



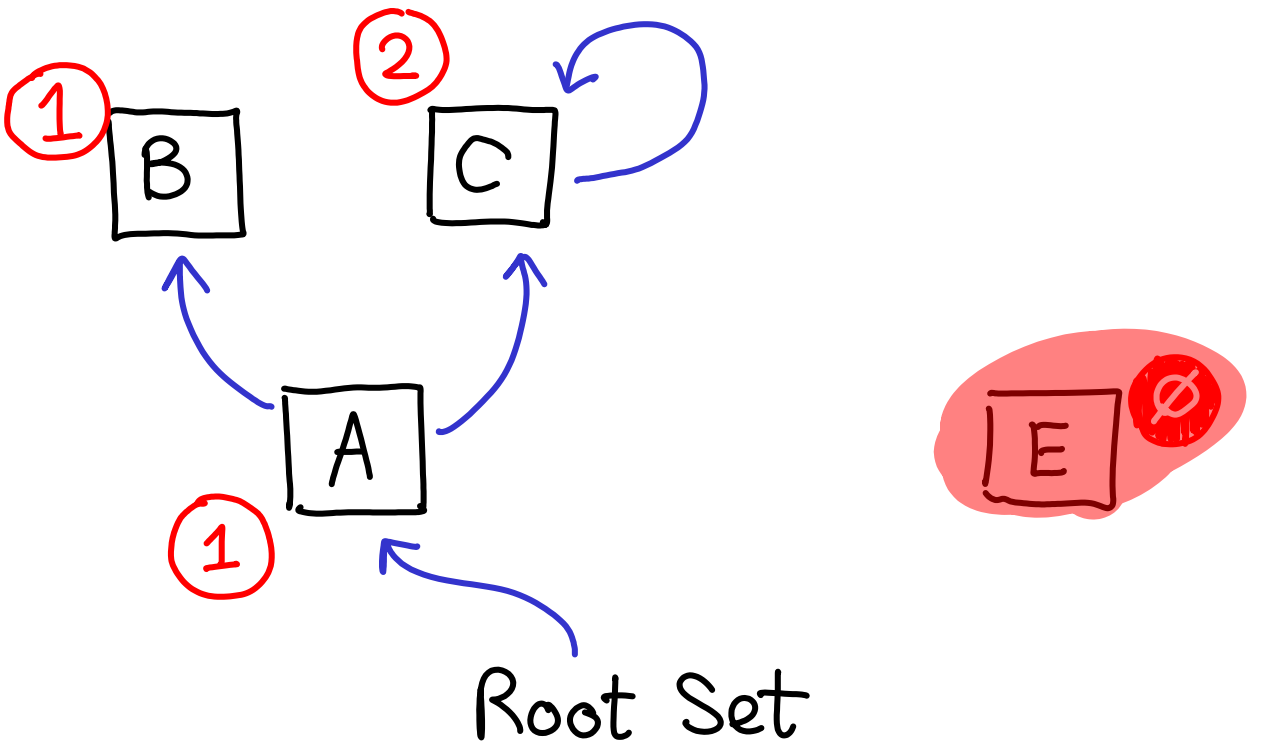
Reference counting

Count the number of incoming references



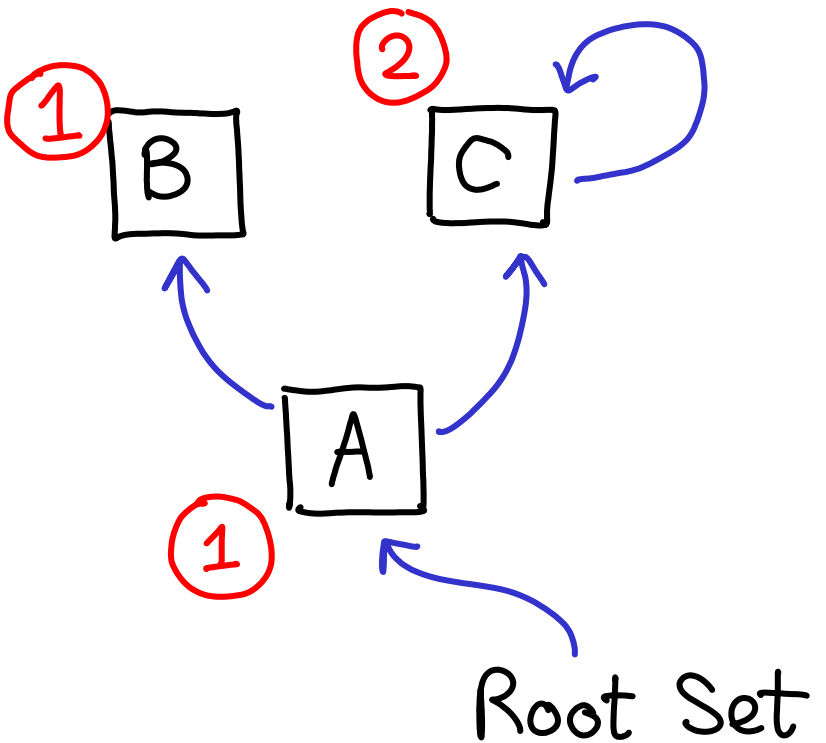
Reference counting

Count the number of incoming references



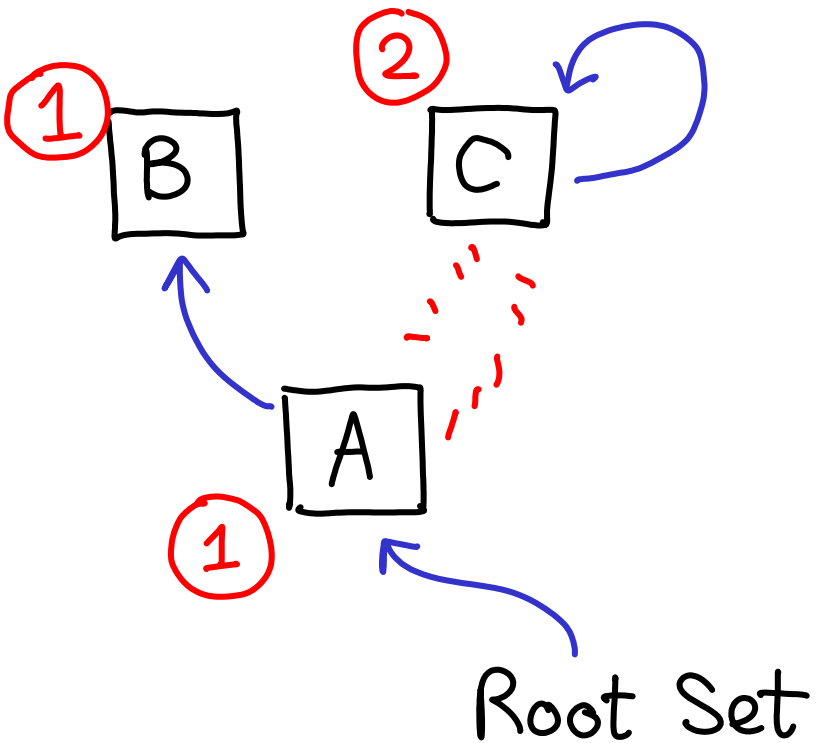
Reference counting

Count the number of incoming references



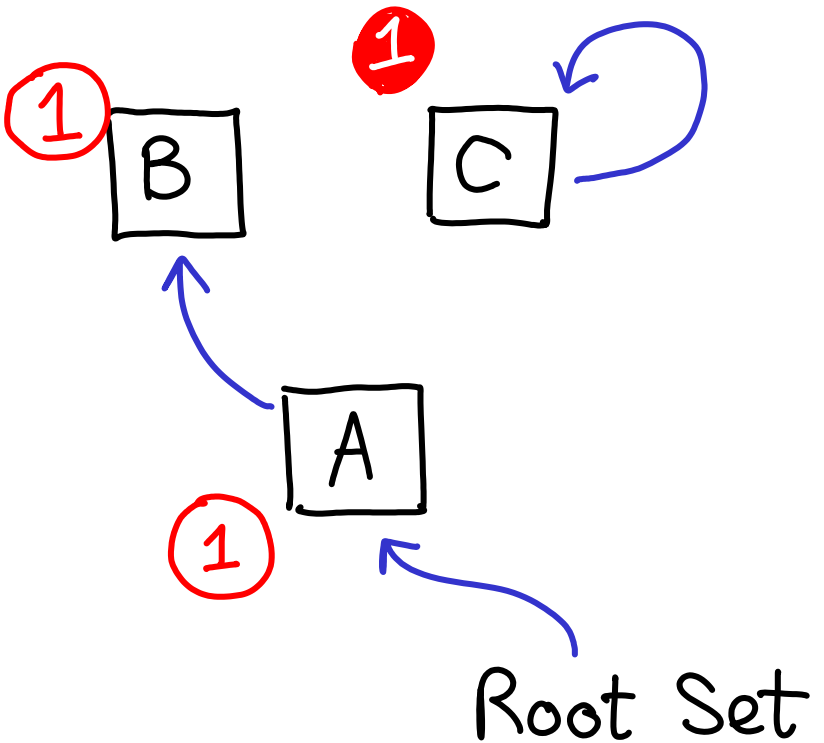
Reference counting

Count the number of incoming references



Reference counting

Count the number of incoming references



Reference counting

Count the number of incoming references

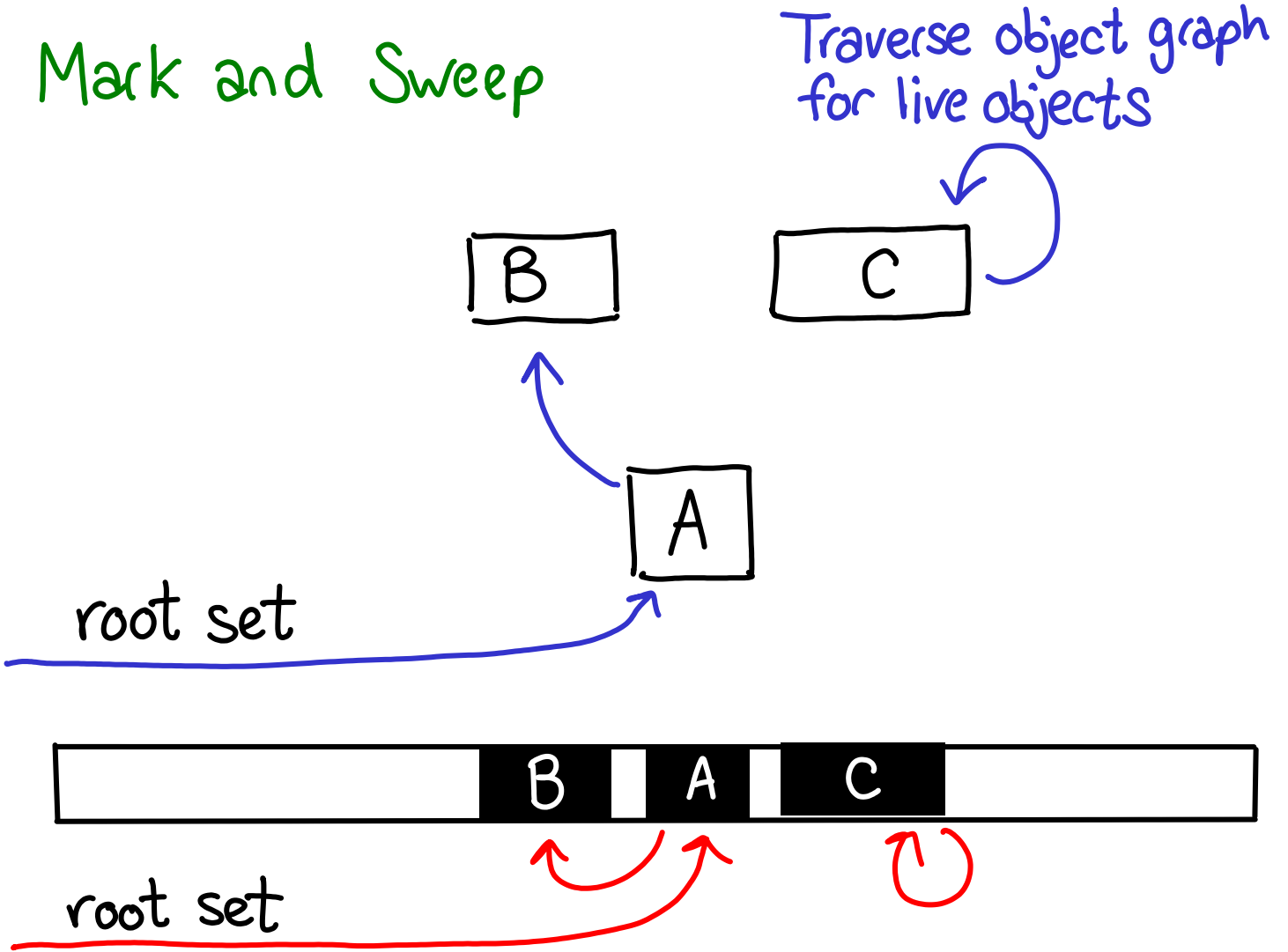
- ✓ Very easy to implement
- ✓ Objects immediately deallocated
- ✗ Cycles never die! (cycle-breaking)
- ✗ Storing & updating counts is costly
- ✗ Sychronizing updates

Reference counting

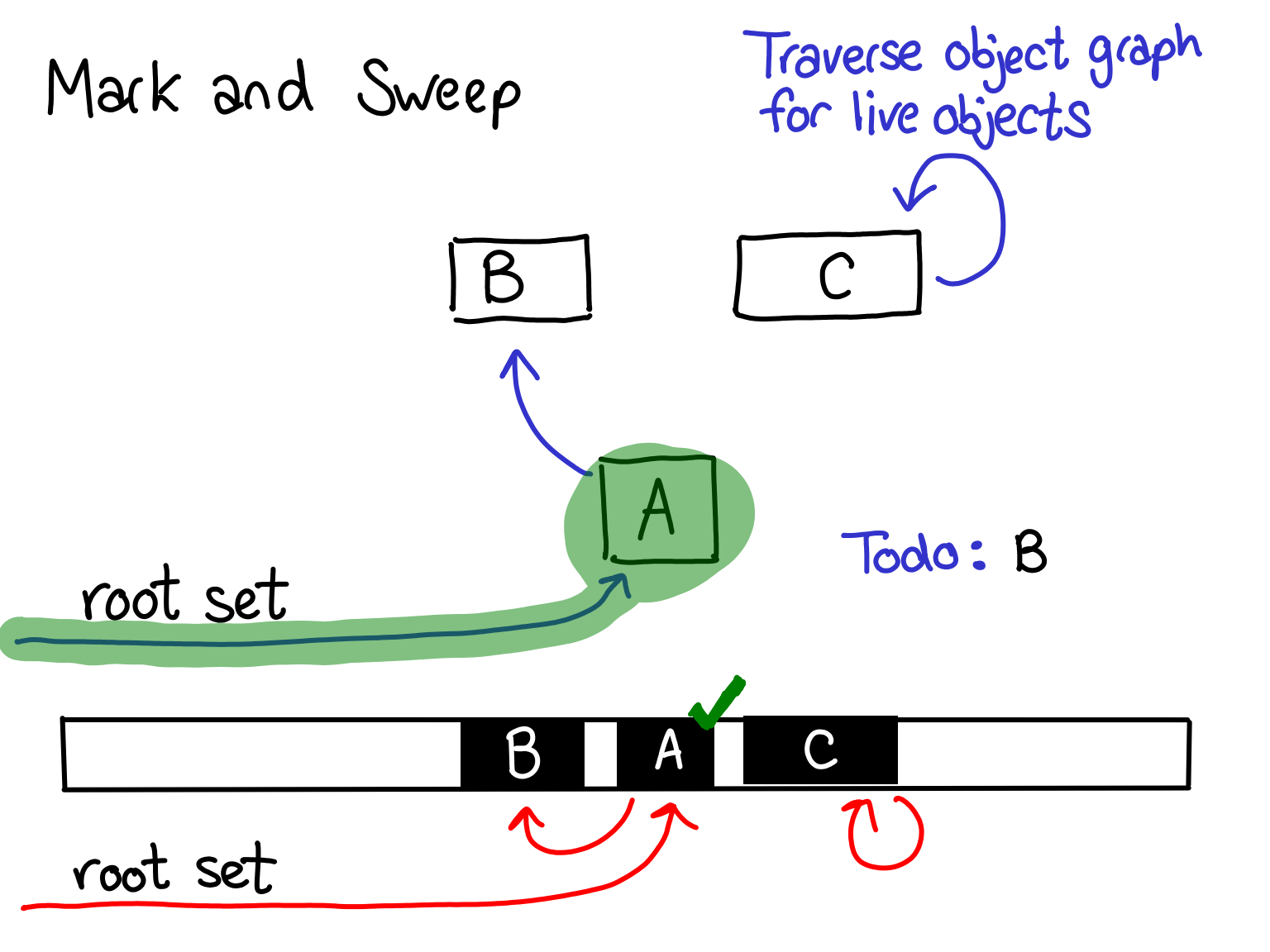
Count the number of incoming references

- ✓ Very easy to implement
- ✓ Objects immediately deallocated
- ✗ Cycles never die! (cycle-breaking)
- ✗ Storing & updating counts is costly
- ✗ Sychronizing updates

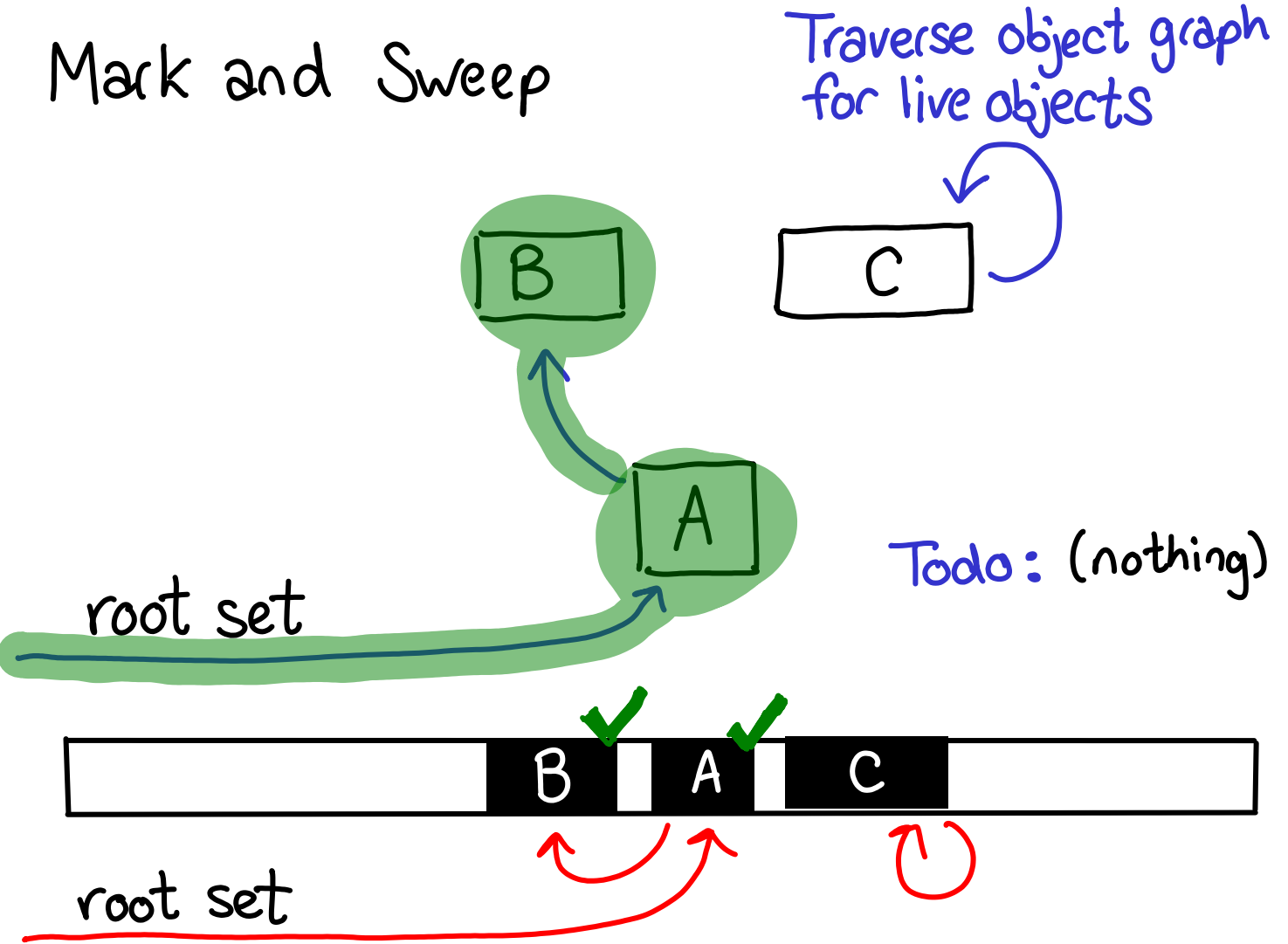
Mark and Sweep



Mark and Sweep

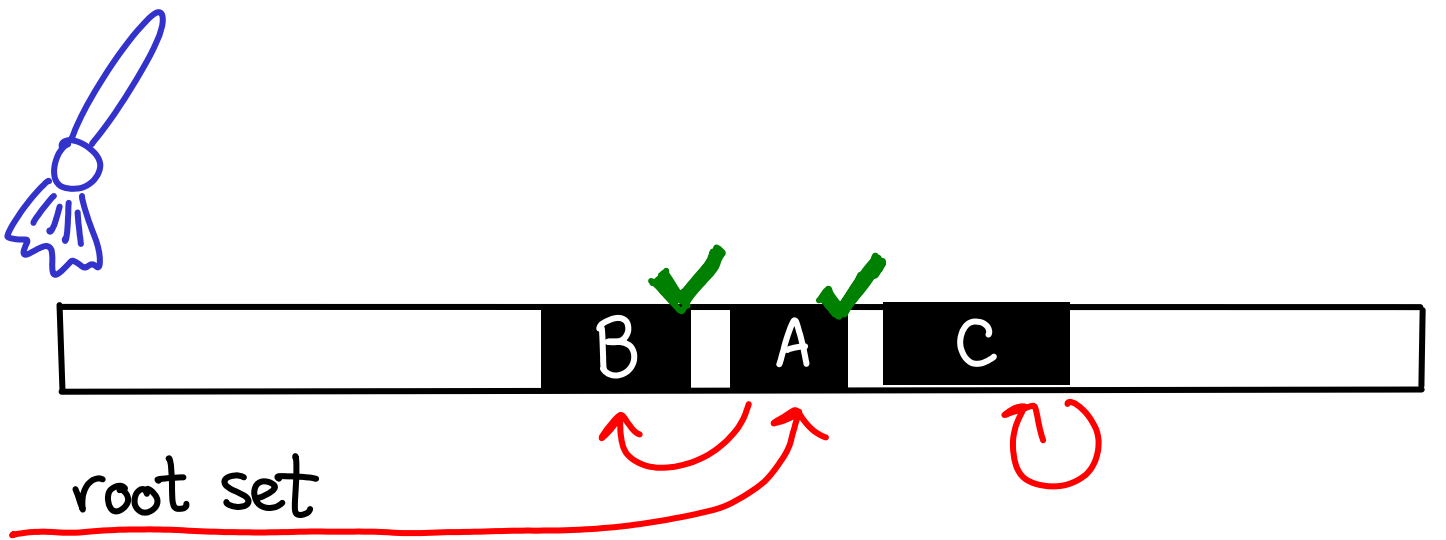


Mark and Sweep



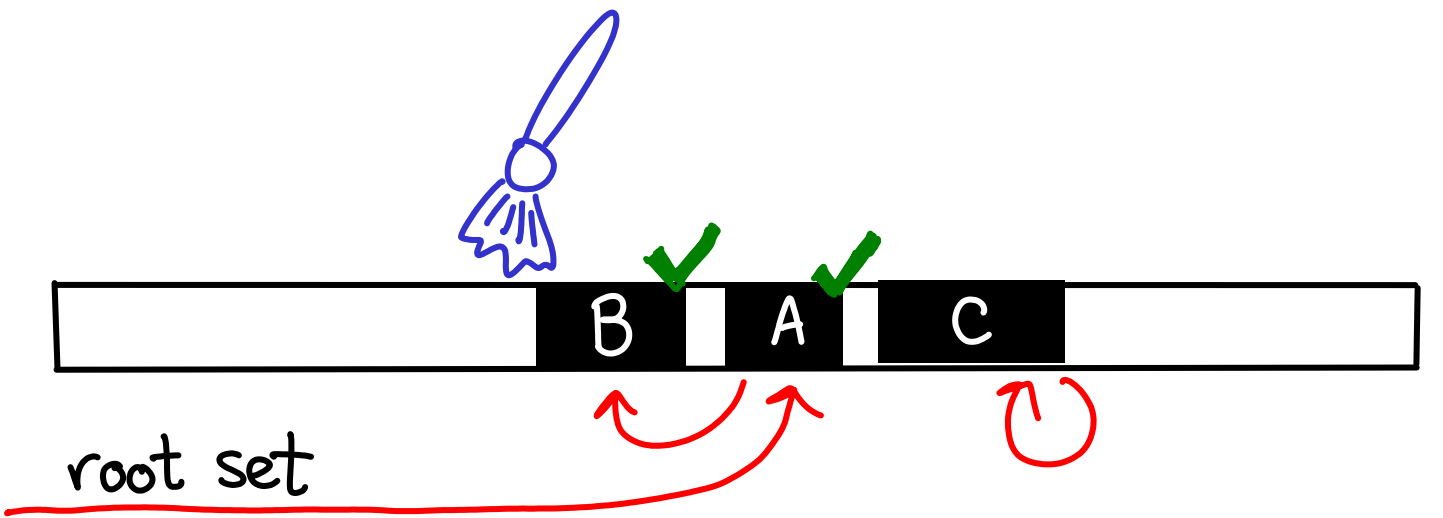
Mark and Sweep

Sweep memory for dead objects



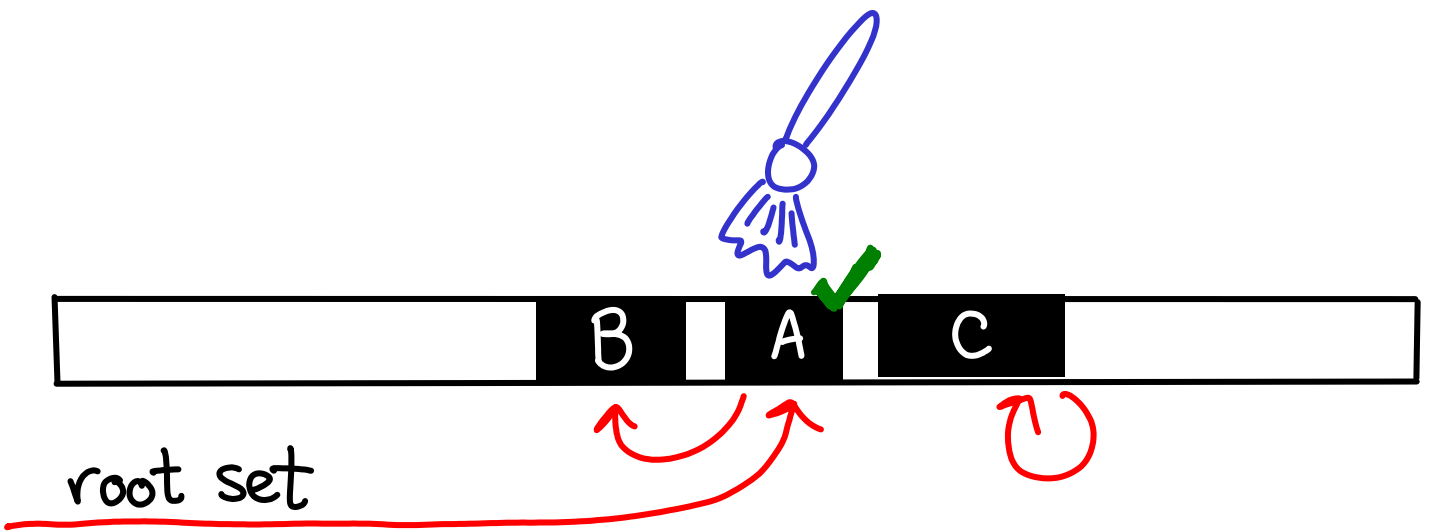
Mark and Sweep

Sweep memory for dead objects



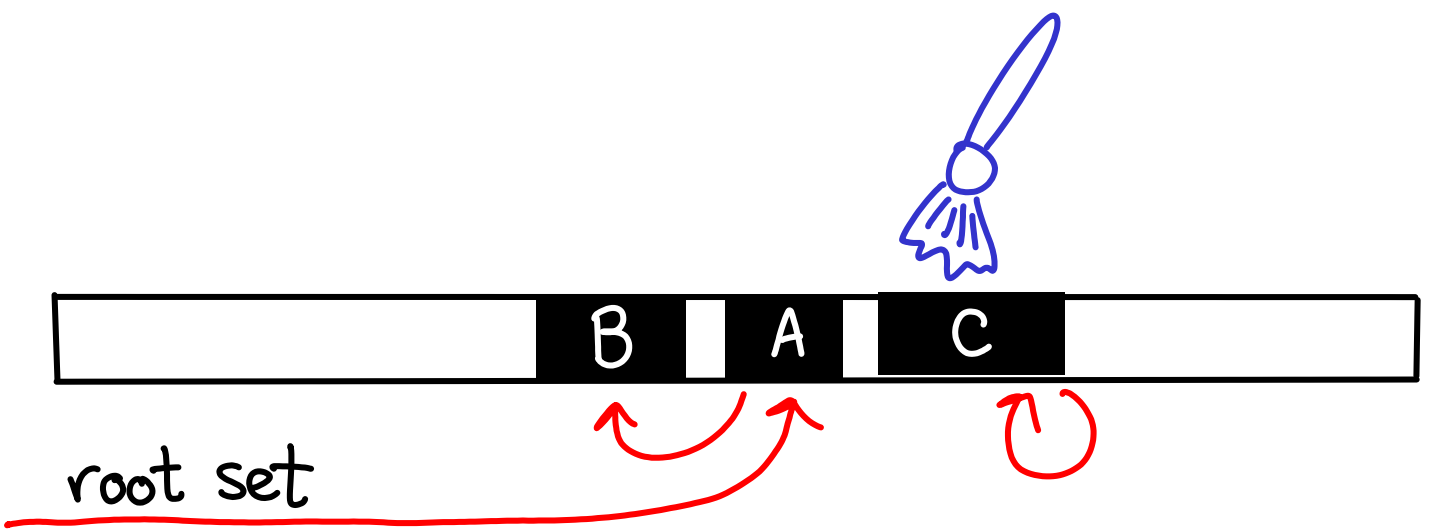
Mark and Sweep

Sweep memory for dead objects



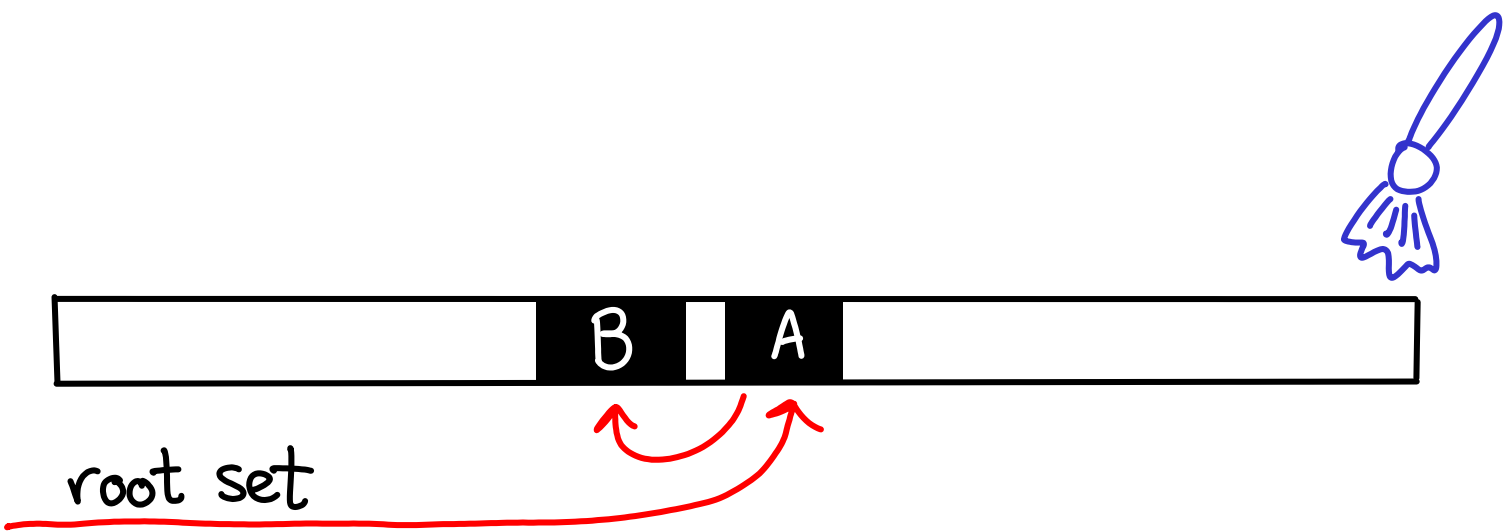
Mark and Sweep

Sweep memory for dead objects



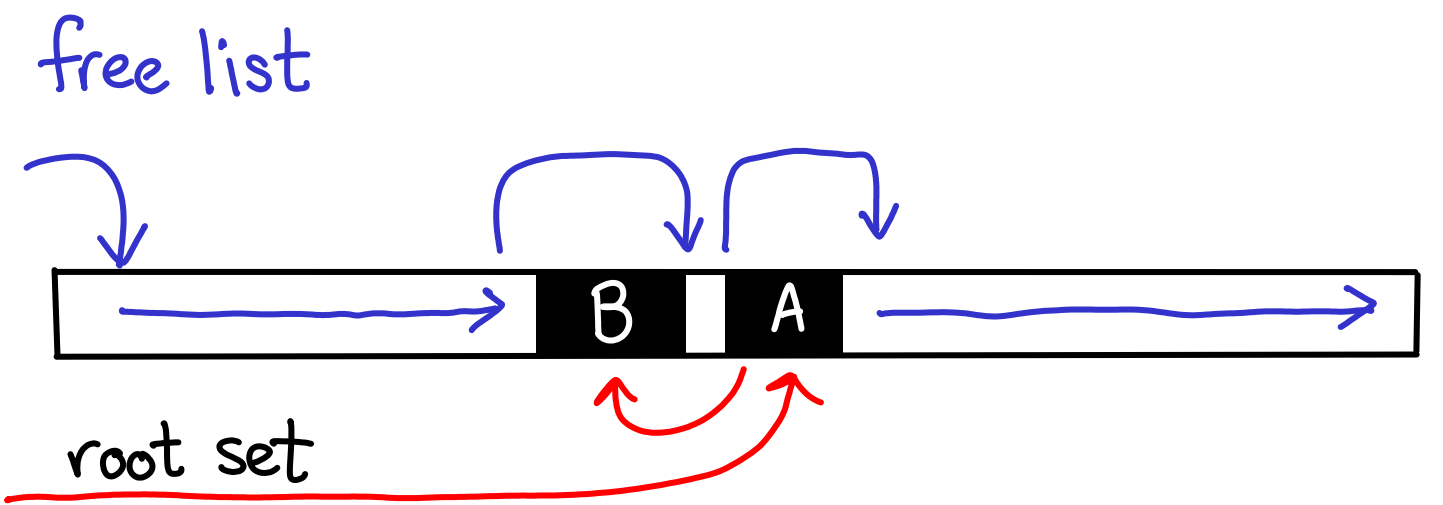
Mark and Sweep

Sweep memory for dead objects



Mark and Sweep

Sweep memory for dead objects



Mark and Sweep

Traverse object graph for live objects

Sweep memory for dead objects

✓ Cycles are handled

✓ No extra bookkeeping

⚠ Naïvely needs to traverse entire heap

⚠ Naïvely leads to fragmentation (can compact)

✗ Needs to store a mark bit

✗ Needs to maintain TODO list

✗ Stop-the-world GC (could refcounting pause?)

Mark and Sweep

Traverse object graph for live objects

Sweep memory for dead objects

✓ Cycles are handled

✓ No extra bookkeeping

⚠ Naïvely needs to traverse entire heap

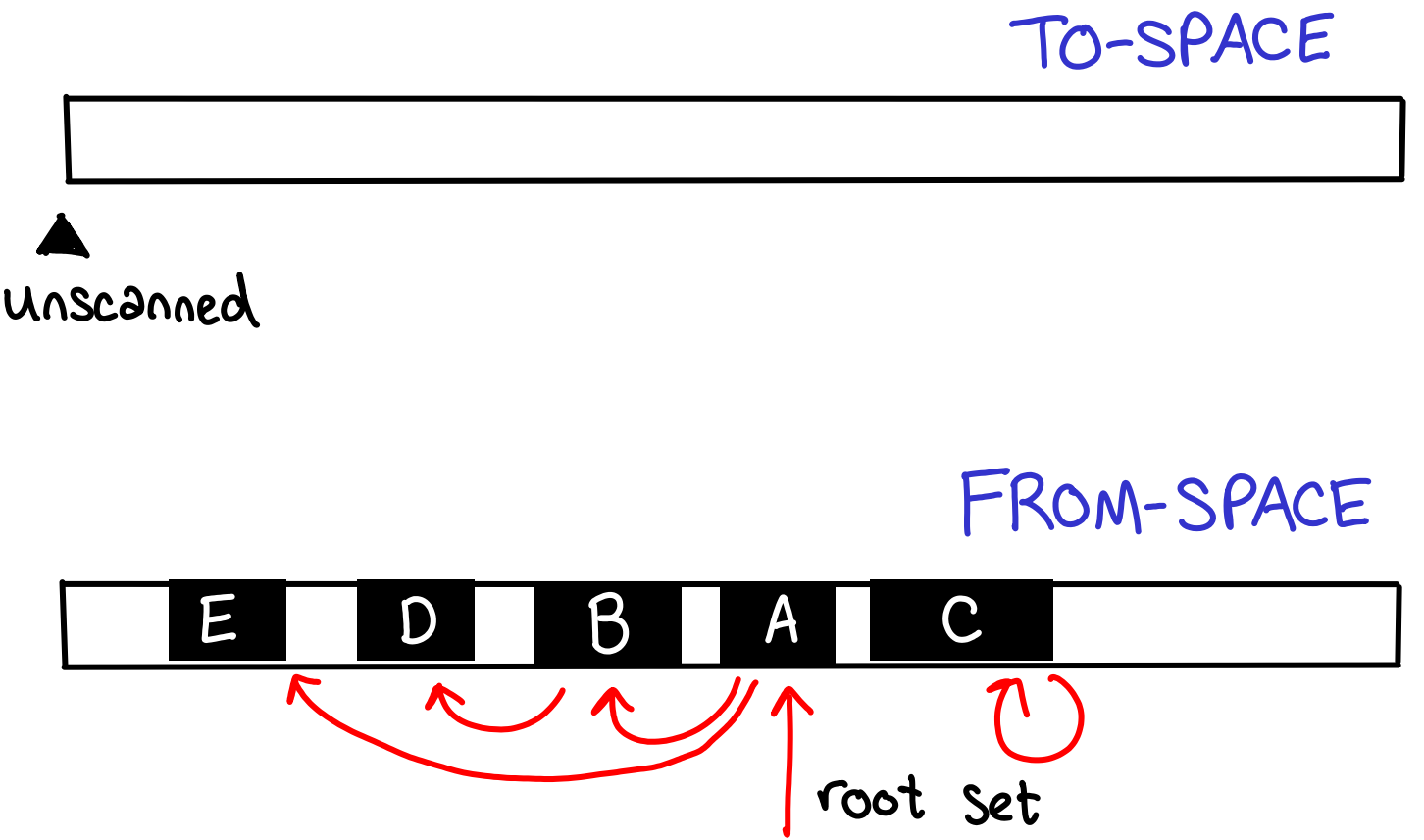
⚠ Naïvely leads to fragmentation (can compact)

✗ Needs to store a mark bit

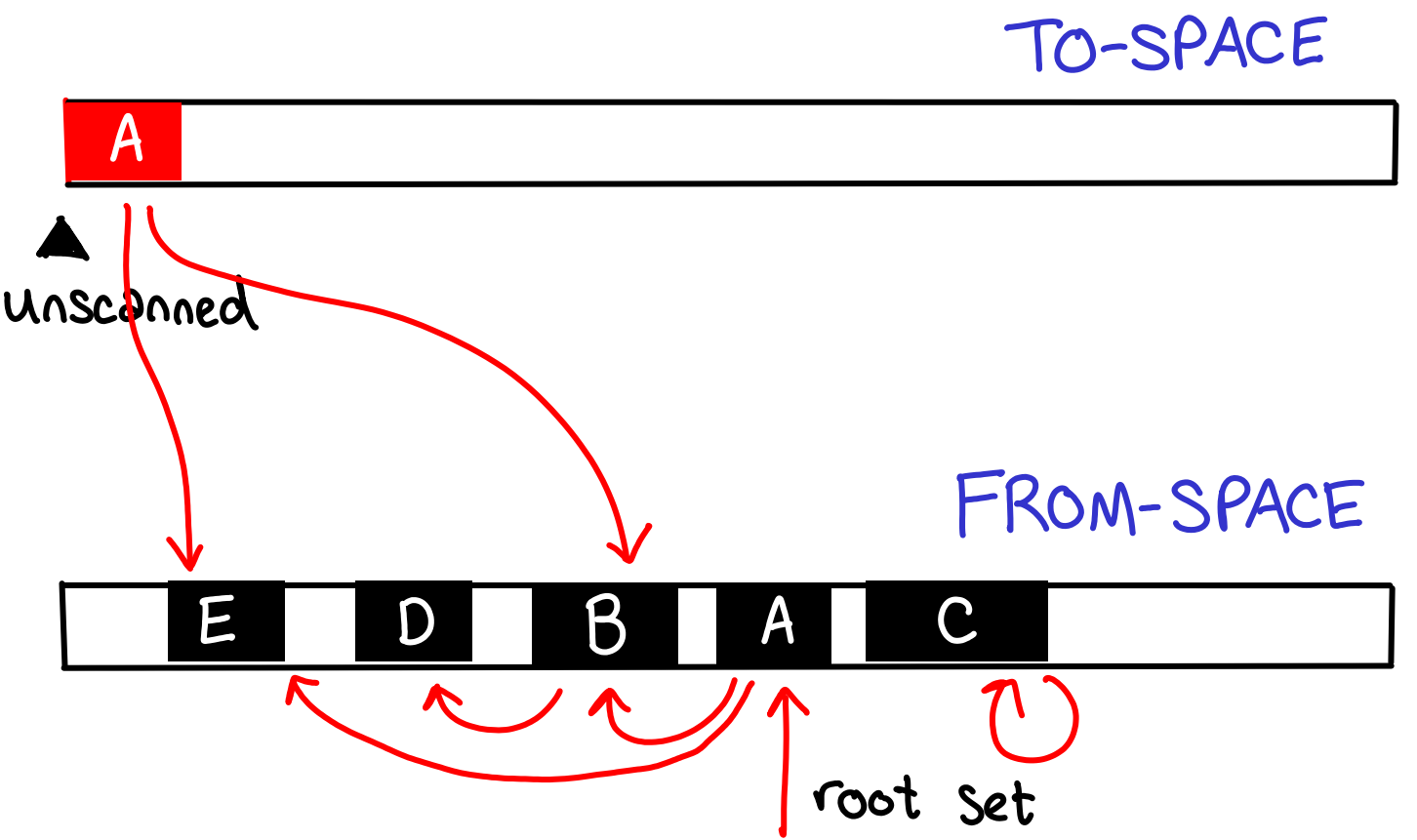
✗ Needs to maintain TODO list

✗ Stop-the-world GC (could refcounting pause?)

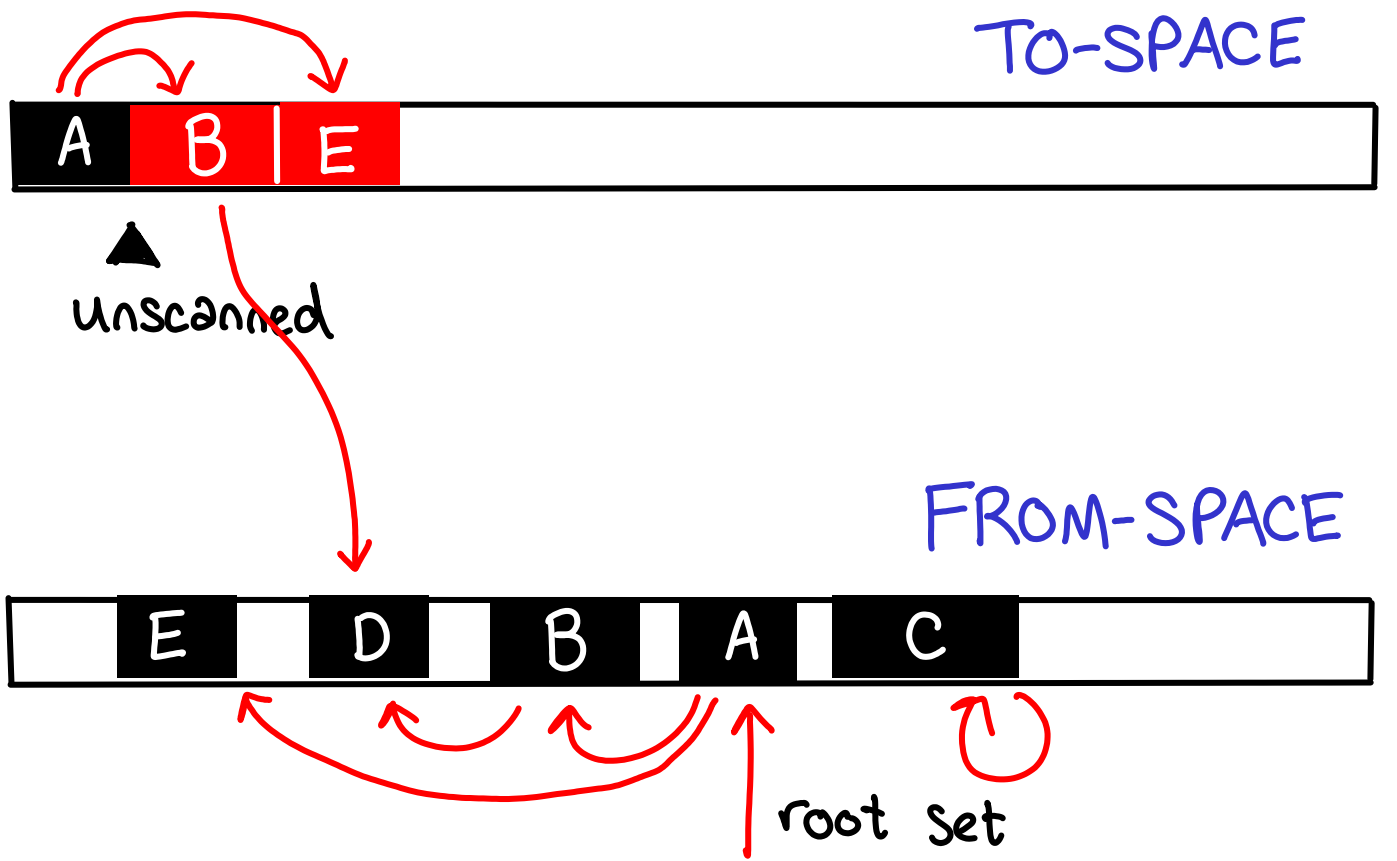
Copying Collection



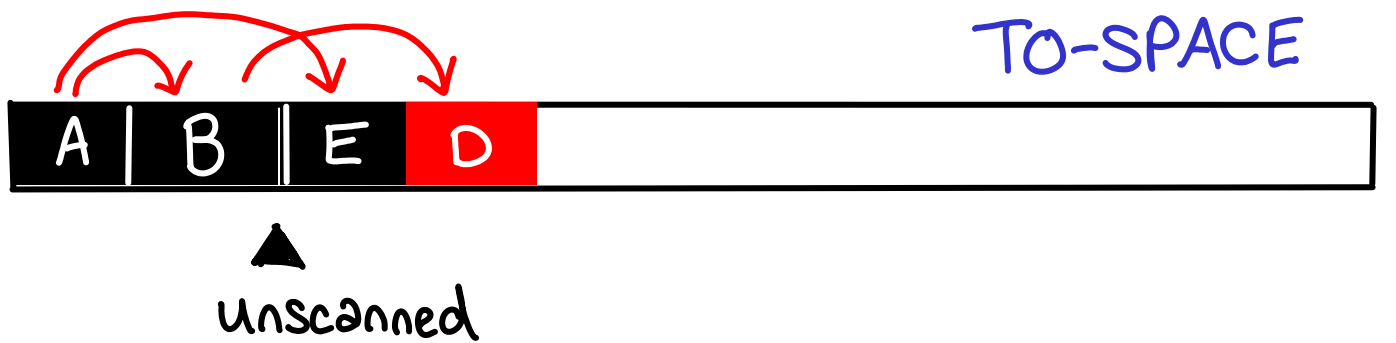
Copying Collection

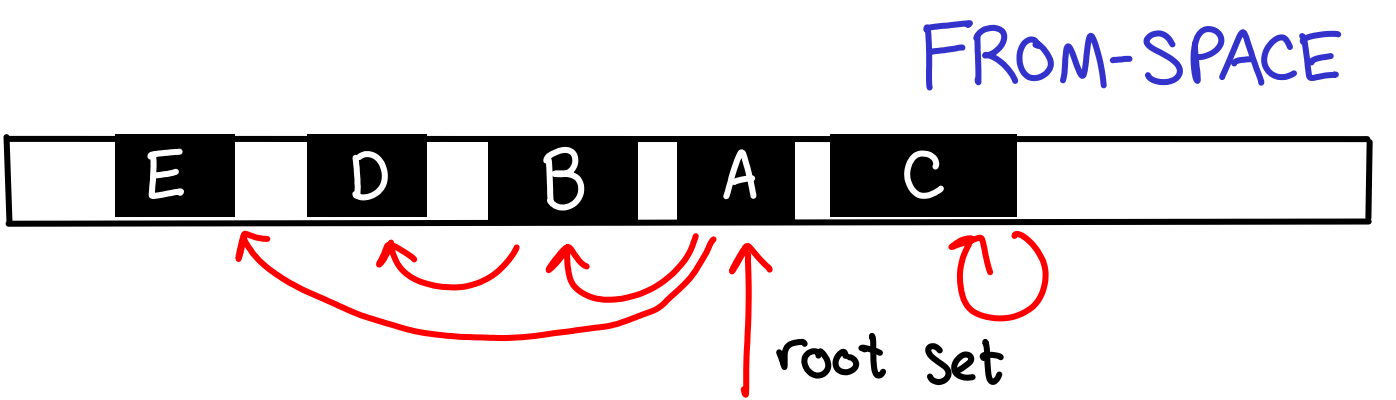


Copying Collection

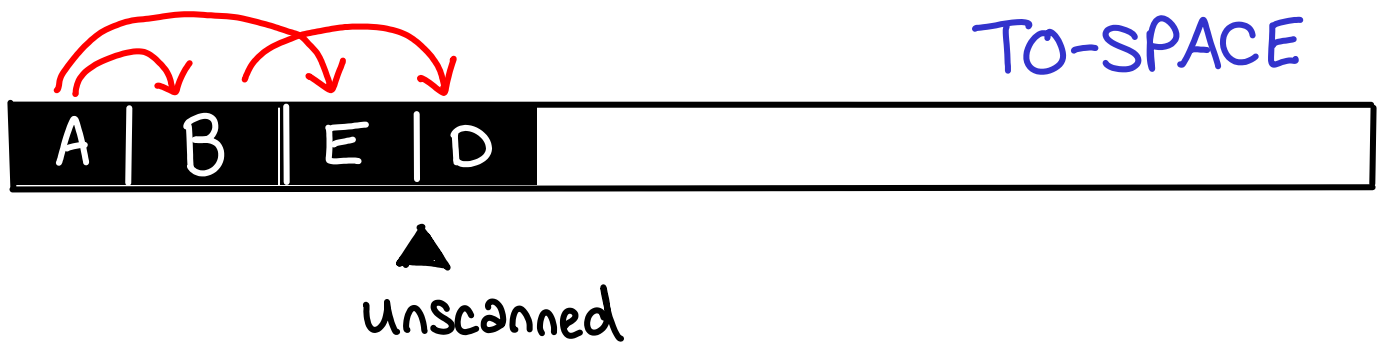


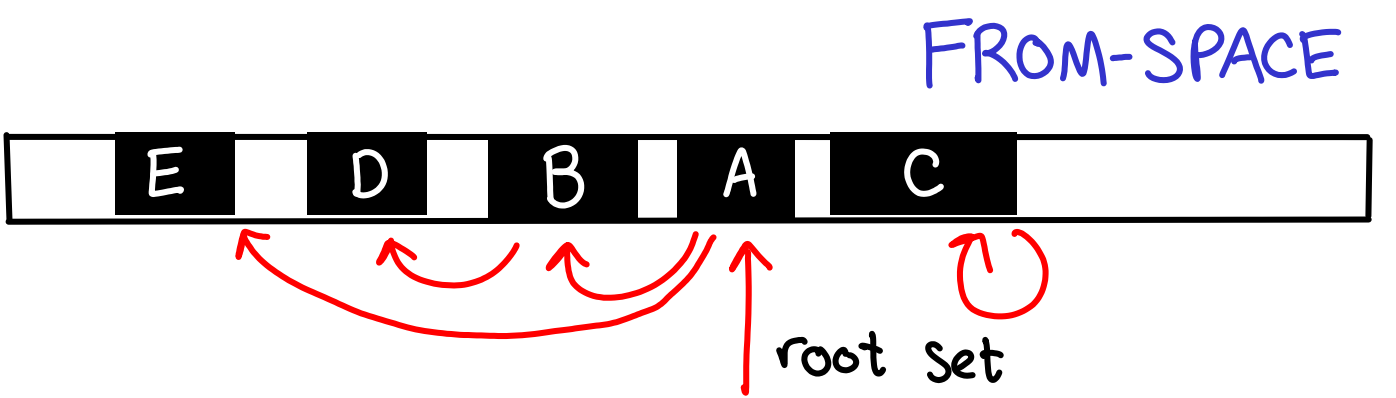
Copying Collection



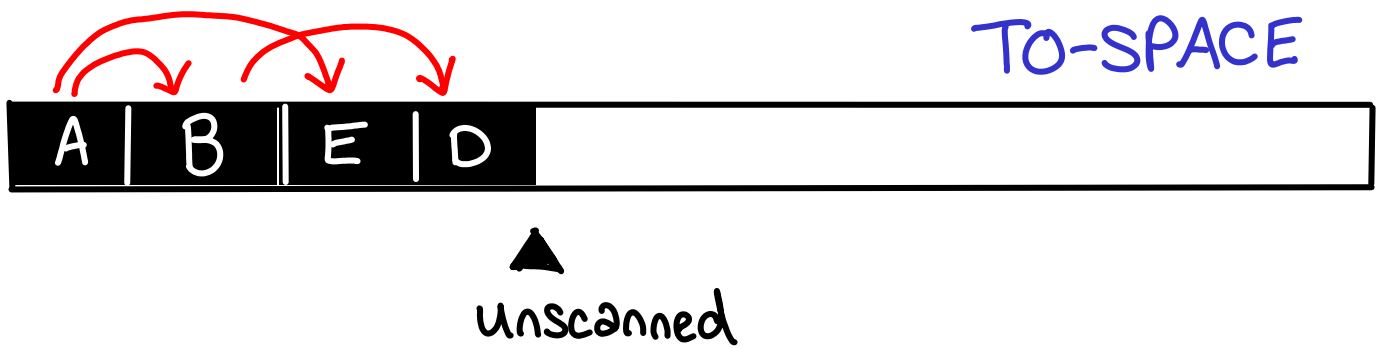


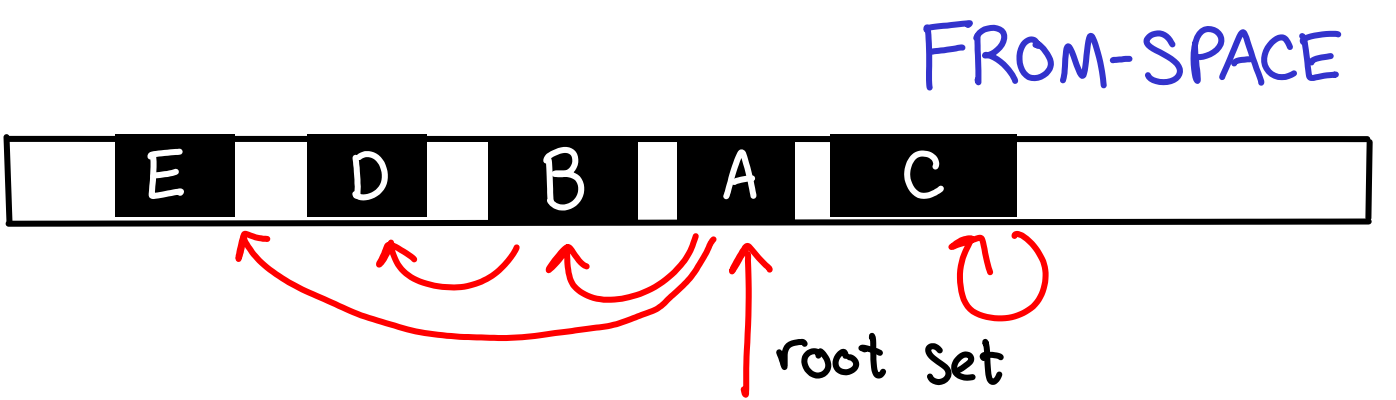
Copying Collection





Copying Collection





Copying Collection

✓ Compacts data (better locality)

✓ Constant space bookkeeping

✗ Needs x2 available space

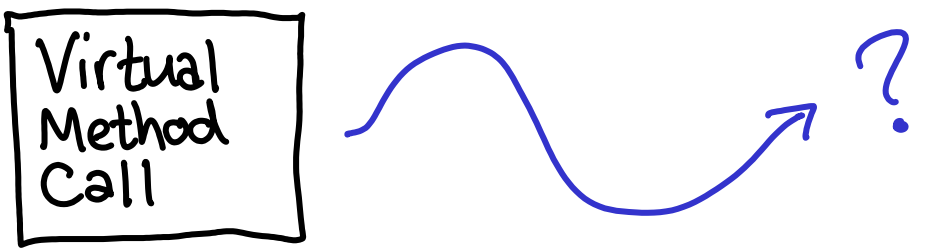
✗ (Still) Stop-the-world GC

Summary: Garbage Collection

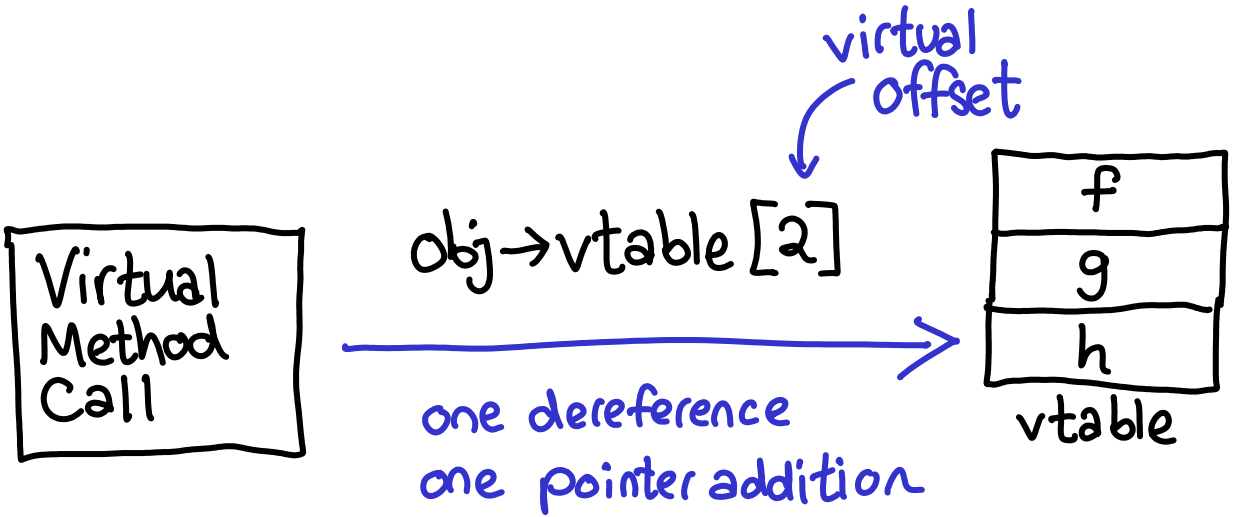
- Provide the ILLUSION of infinite memory
- Liveness based on reachability
- Generational GC (it's hard!)

Dynamic Dispatch

Recap: C++ Virtual Tables



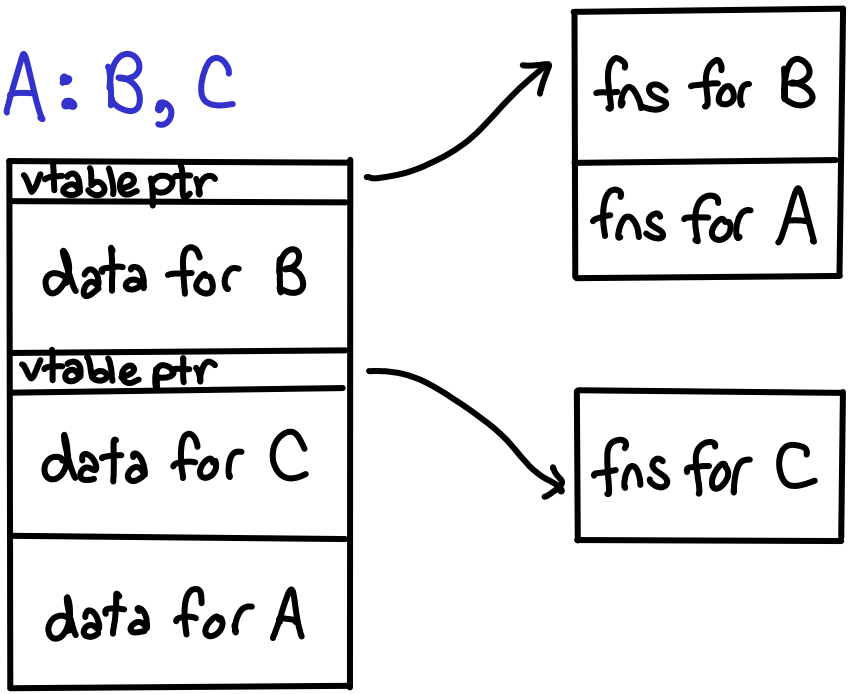
Recap: C++ Virtual Tables



C++ goal: Make virtual dispatch as efficient as possible

Recap: C++ Virtual Tables

class A: B, C



Consequence: Multiple inheritance, but no interfaces

Recap: C++ Virtual Tables

Motivating Problem

```
class A {
    virtual void f();
    virtual void g();
}
```

```
class B {
    virtual void g();
    virtual void f();
}
```

Naive solution: Do a dictionary lookup

Recap: C++ Virtual Tables

Motivating Problem

```
class A {
    virtual void f();
    virtual void g();
}
```

```
class B {
    virtual void g();
    virtual void f();
}
```

C++ says: Do both layouts

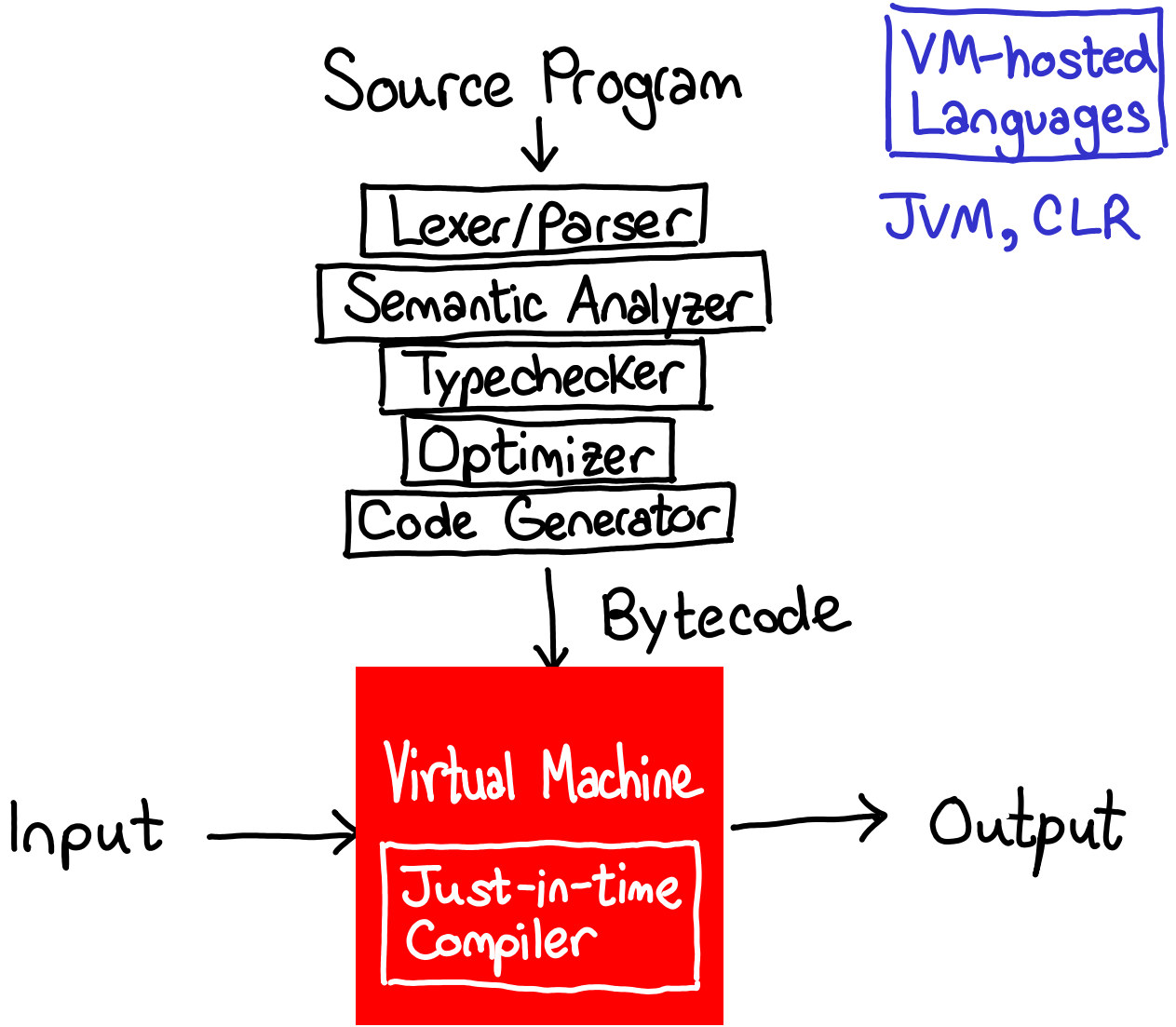
Recap: C++ Virtual Tables

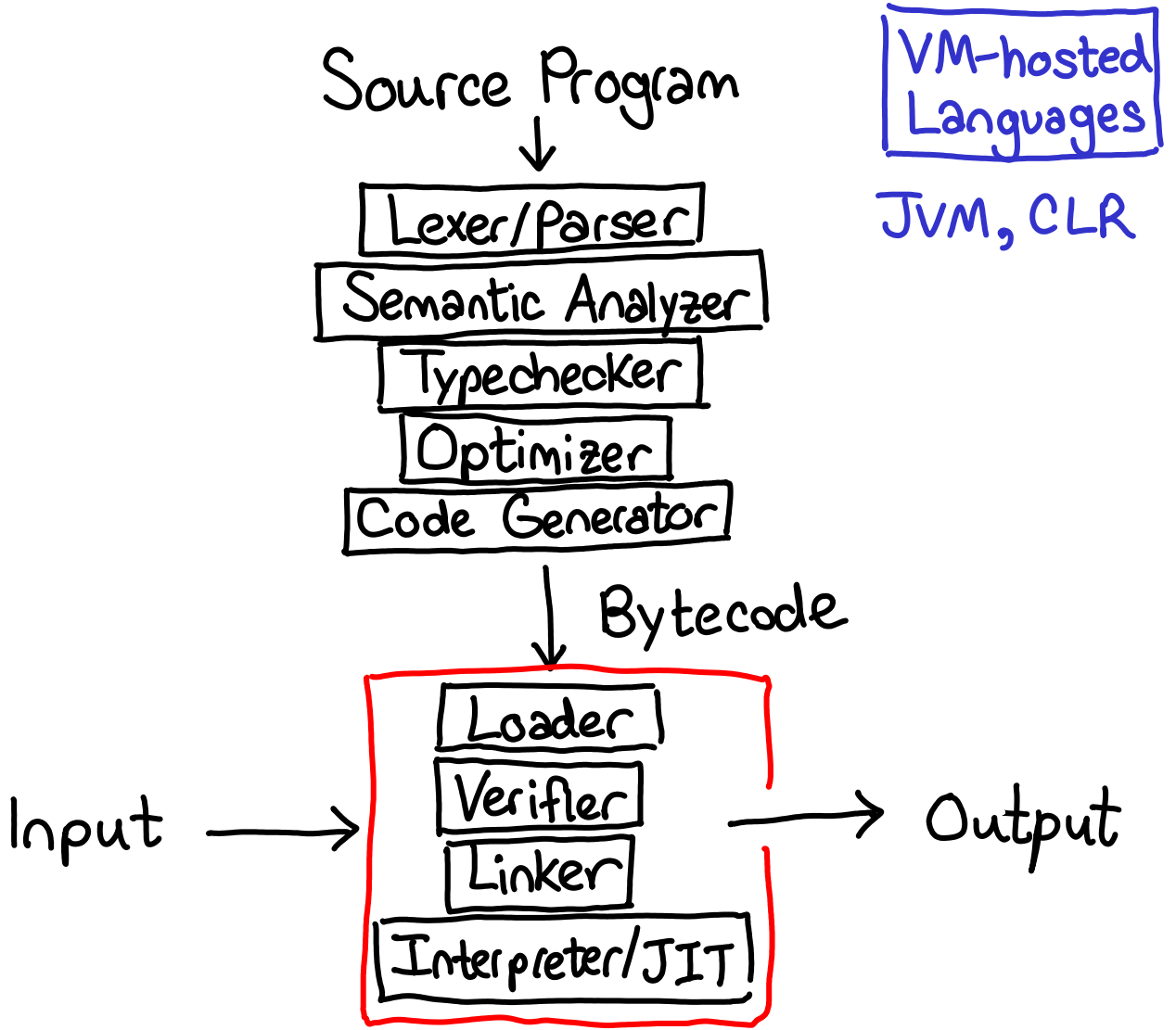
Motivating Problem

```
class A {
    virtual void f();
    virtual void g();
}
```

```
class B {
    virtual void g();
    virtual void f();
}
```

Today: Do a dictionary lookup and cache it


Source Program
Lexer/Parser
Semantic Analyzer
Typechecker
Optimizer
Code Generator
Bytecode
Virtual Machine
Just-in-time Compiler
Input
Output
VM-hosted Languages
JVM, CLR


Source Program
Lexer/Parser
Semantic Analyzer
Typechecker
Optimizer
Code Generator
Bytecode
Loader
Verifier
Linker
Interpreter/JIT
Input
Output
VM-hosted Languages
JVM, CLR

Briefly:

JVM

Loader

On-demand class loading
Search FS for object
Can override default class loader

Verifier

Check if bytecode is valid { valid opcode, valid jump targets, well-typed }

Linker

Add class/interface to runtime
Initialize static fields
Resolve names

Interpreter/JIT

Runtime checks (e.g. bounds checks)

Briefly:

JVM

Bytecode is for a stack machine

```
class A {
    int i;
    void f(int val) { i = val+1 ;}
}
```

```
aload 0  ; object ref this
iload 1  ; int val
iconst 1
iadd     ; add val+1
putfield #4 <Field int i>
return
```

Dynamic Dispatch in the JVM

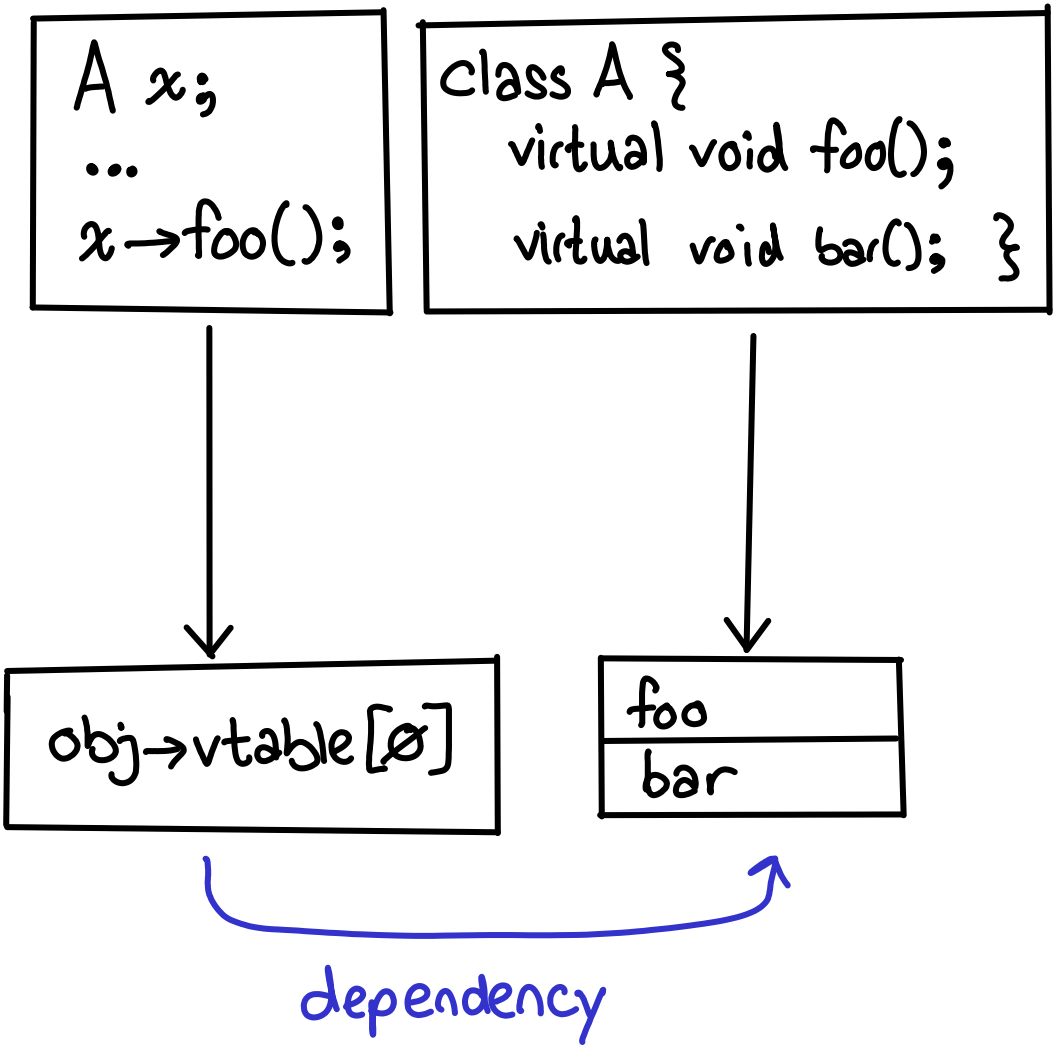
1. invokevirtual — bytecode rewriting
2. invokeinterface — inline caches
3. invokedynamic — polymorphic inline caches

(or Smalltalk or Self)

invokevirtual

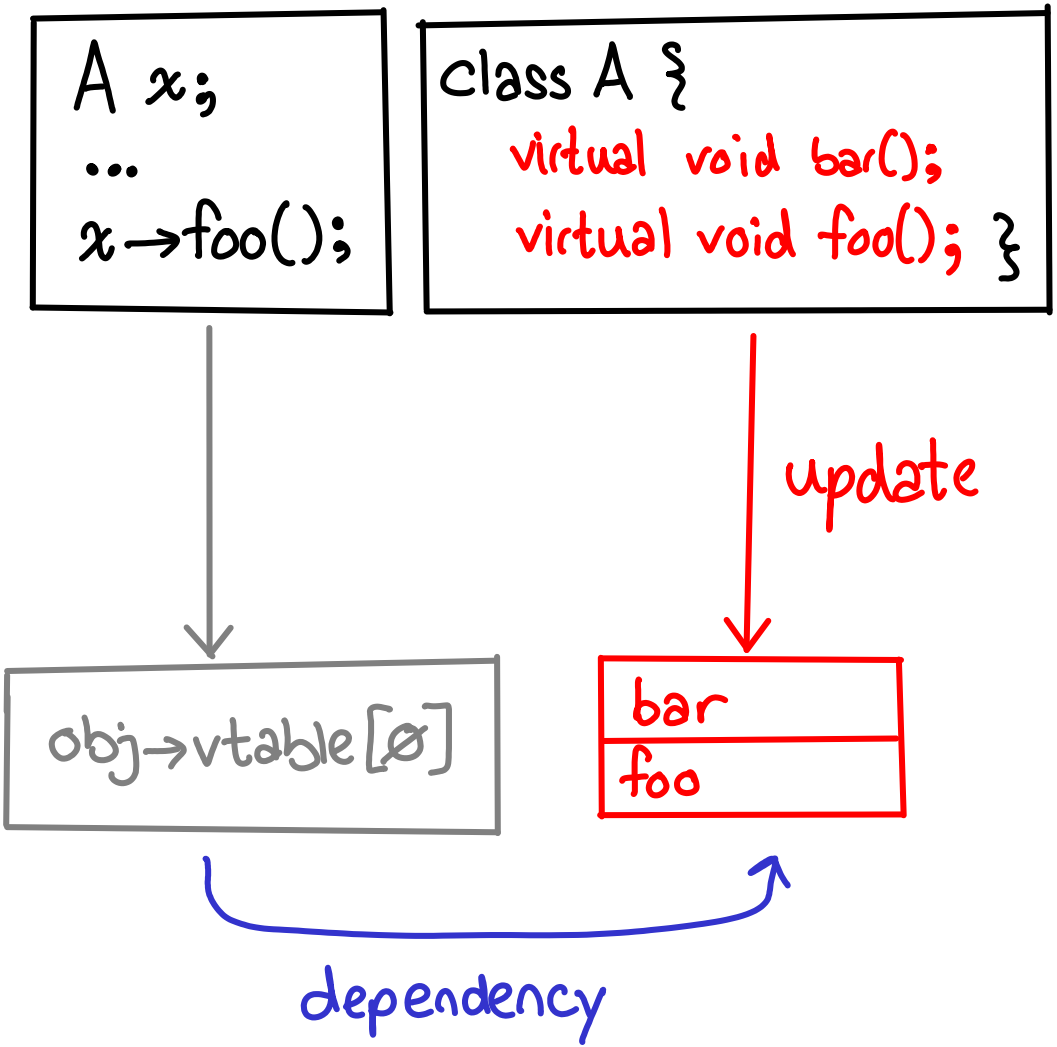
```
A x;
...
x->foo();
```

invokevirtual

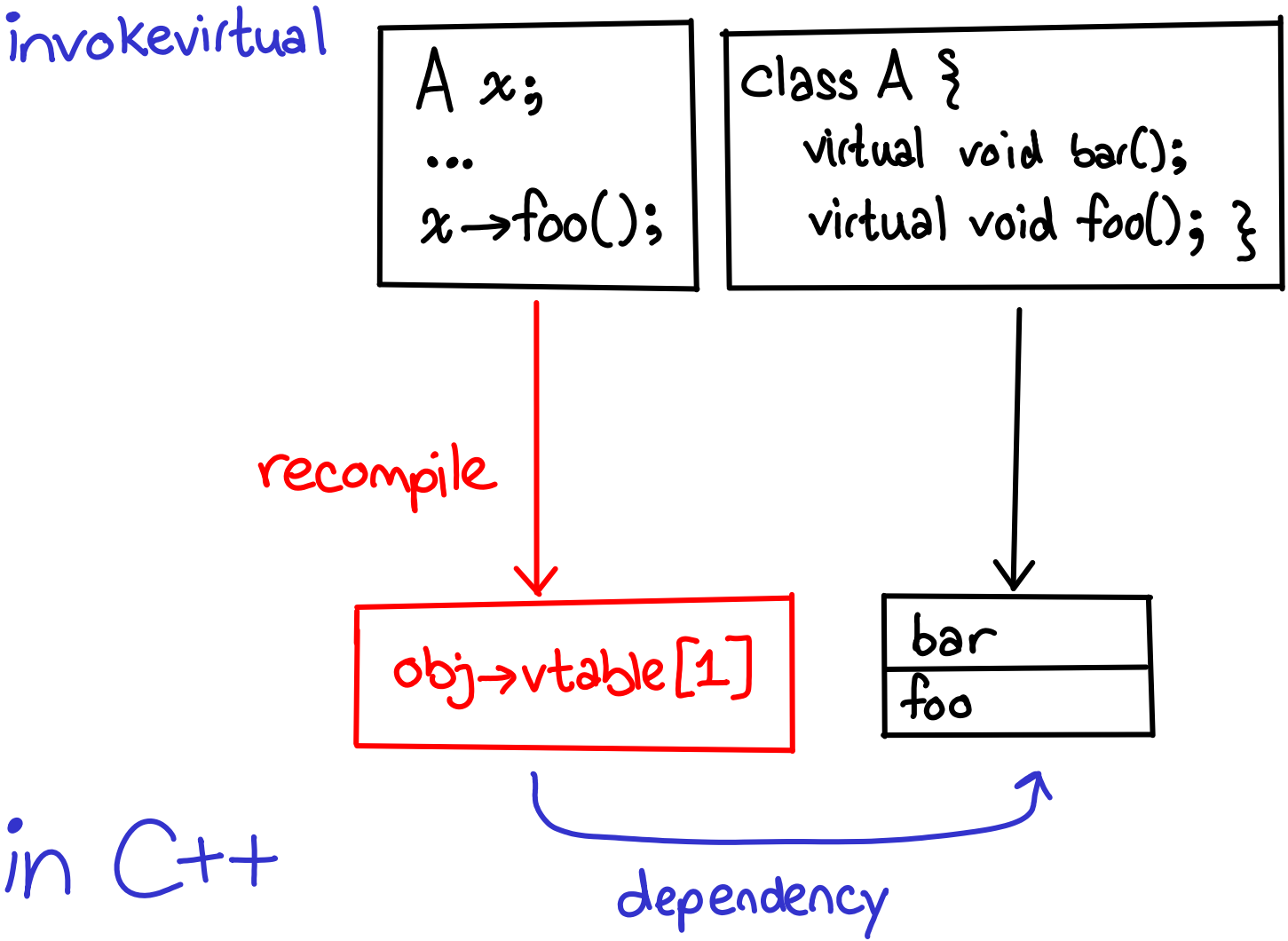


in C++

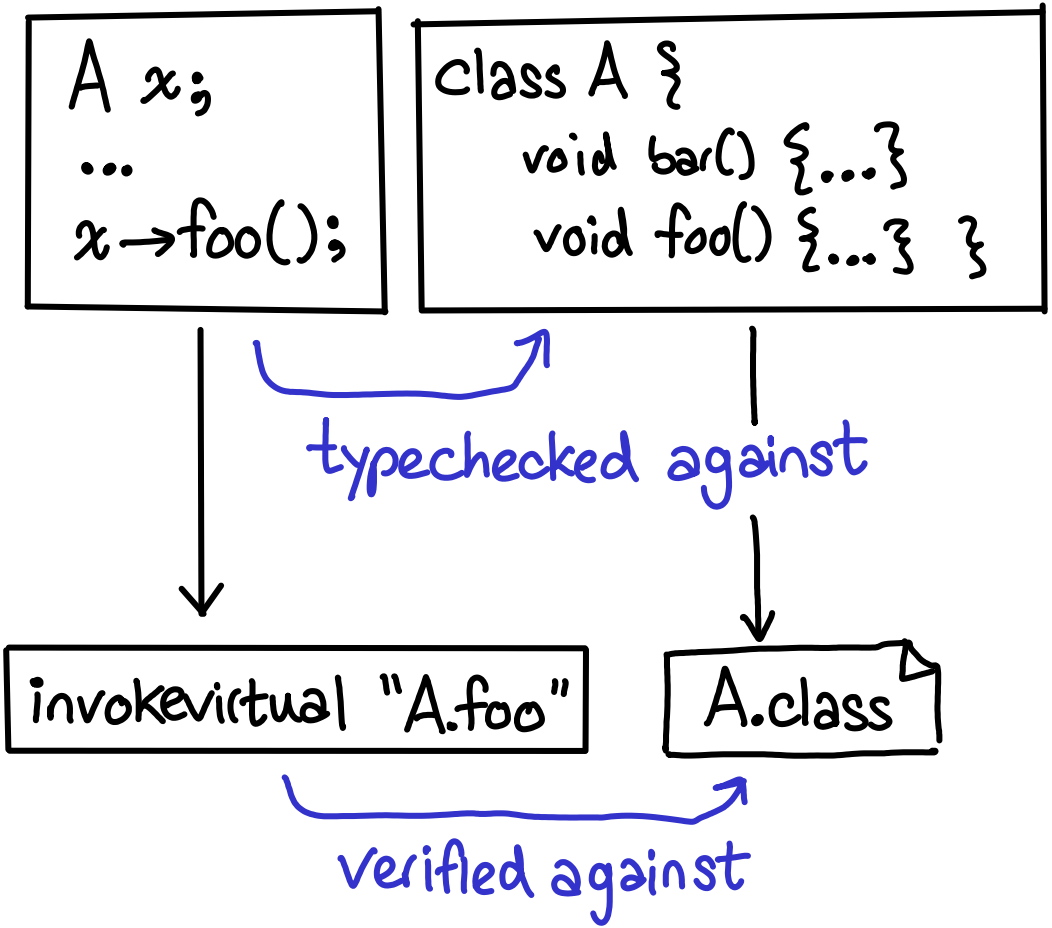
invokevirtual



in C++

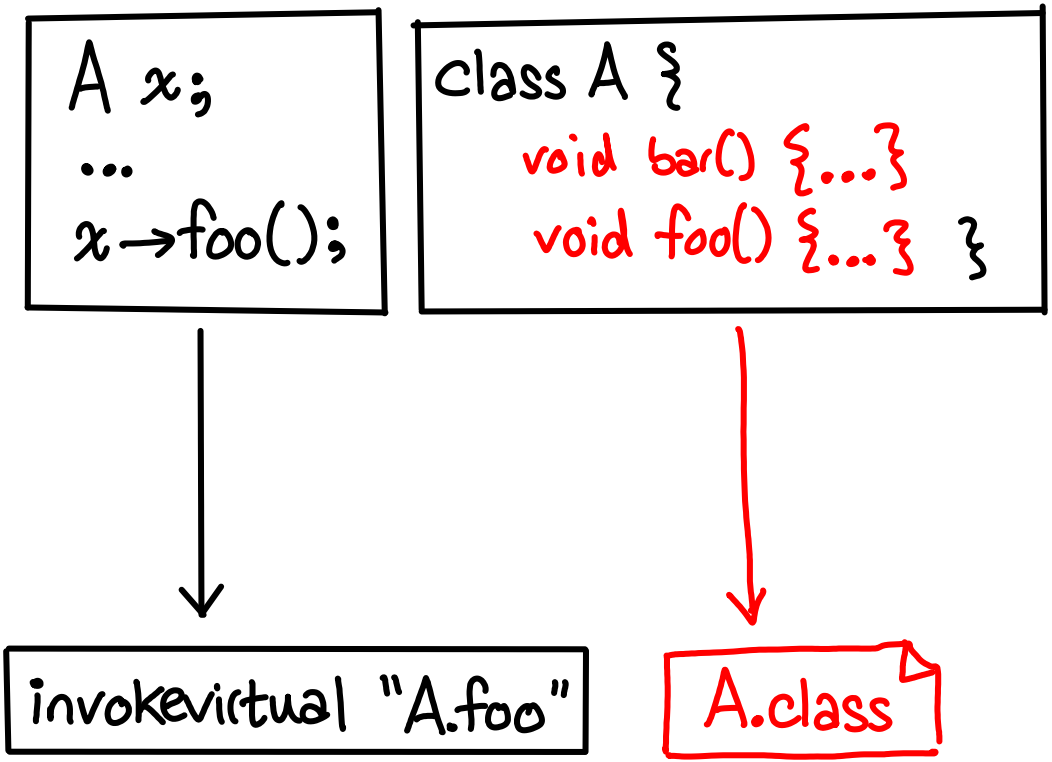

invokevirtual
A x;
...
x->foo();
class A {
virtual void bar();
virtual void foo(); }
recompile
obj->vtable[1]
bar
foo
in C++
dependency

invokevirtual



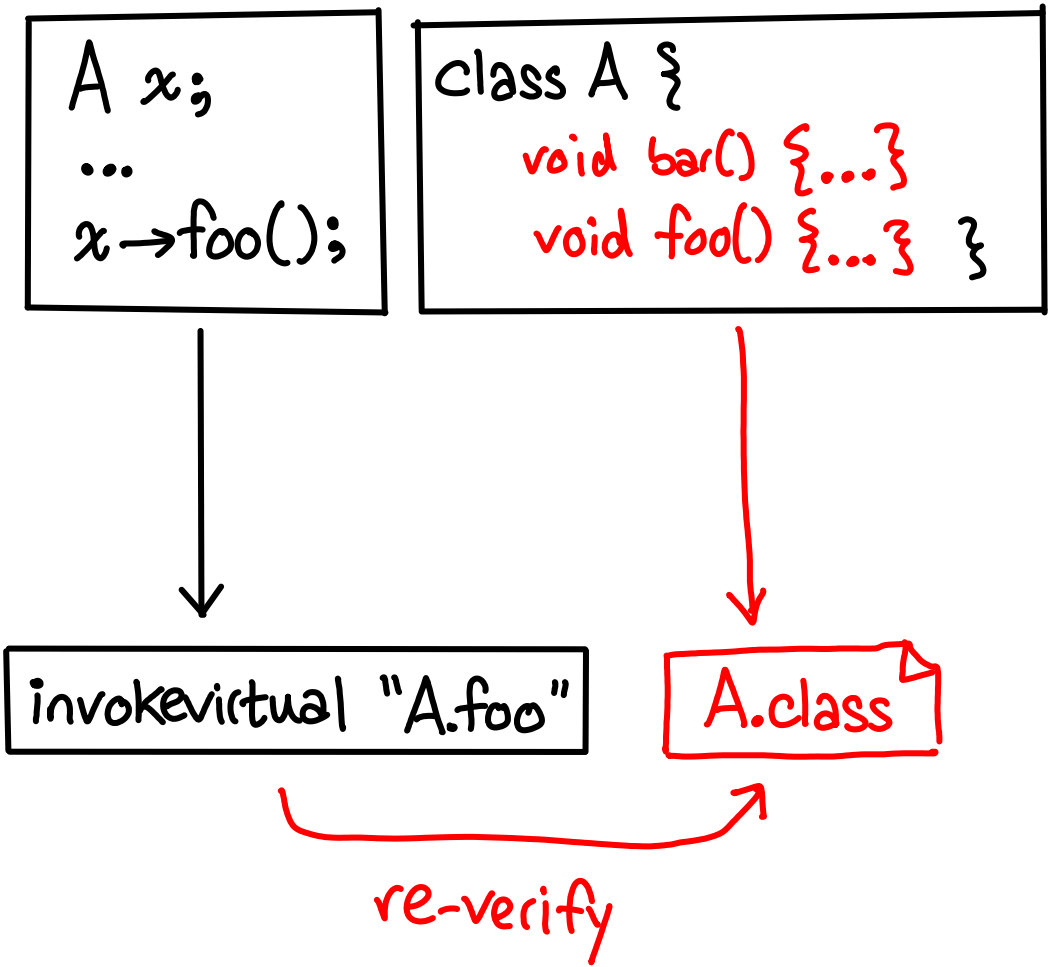
in Java

invokevirtual



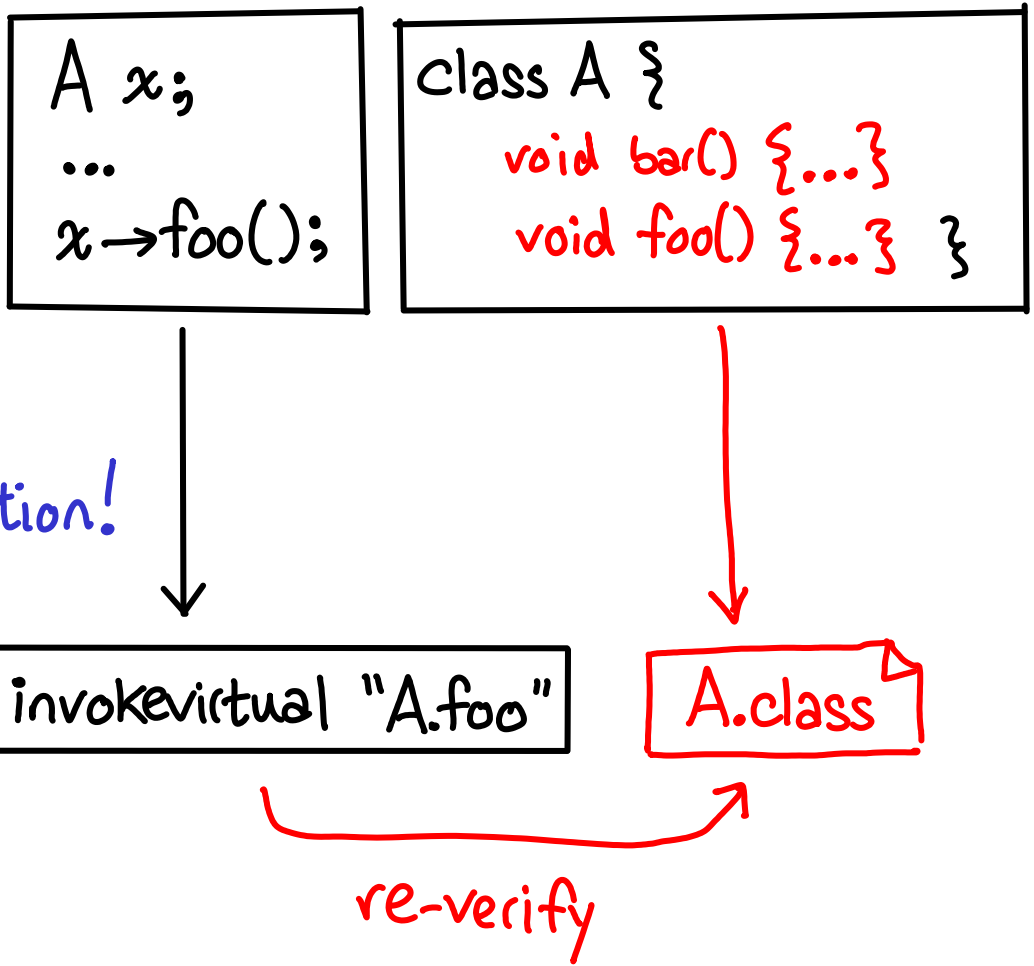
in Java

invokevirtual



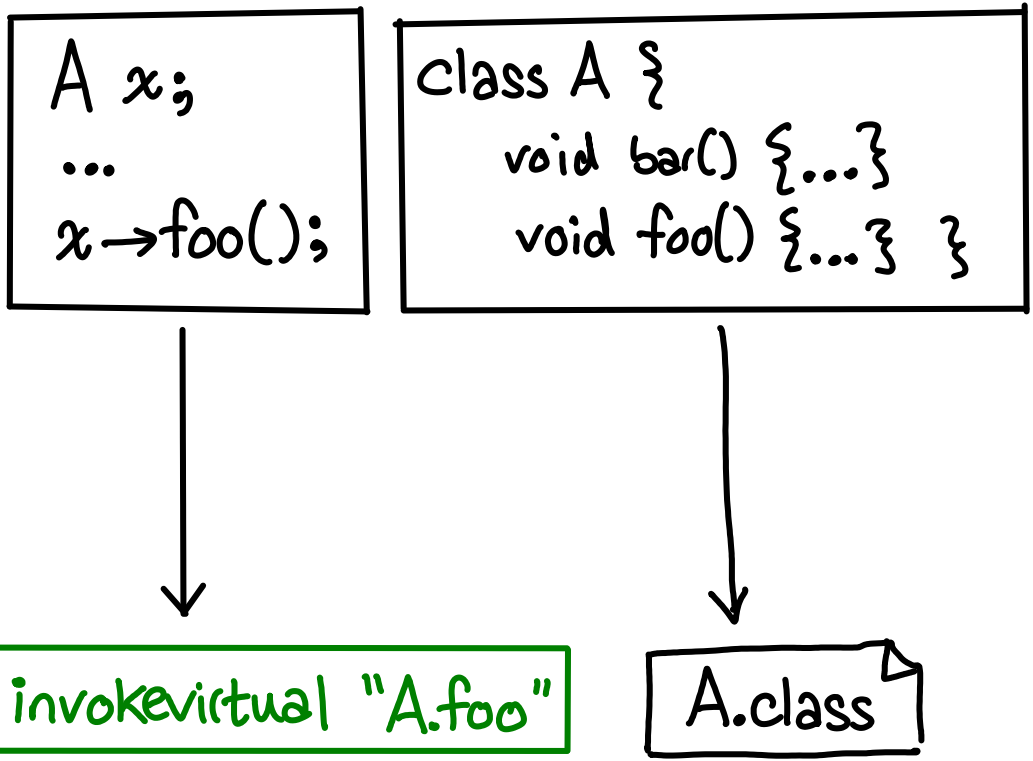
in Java

invokevirtual



in Java

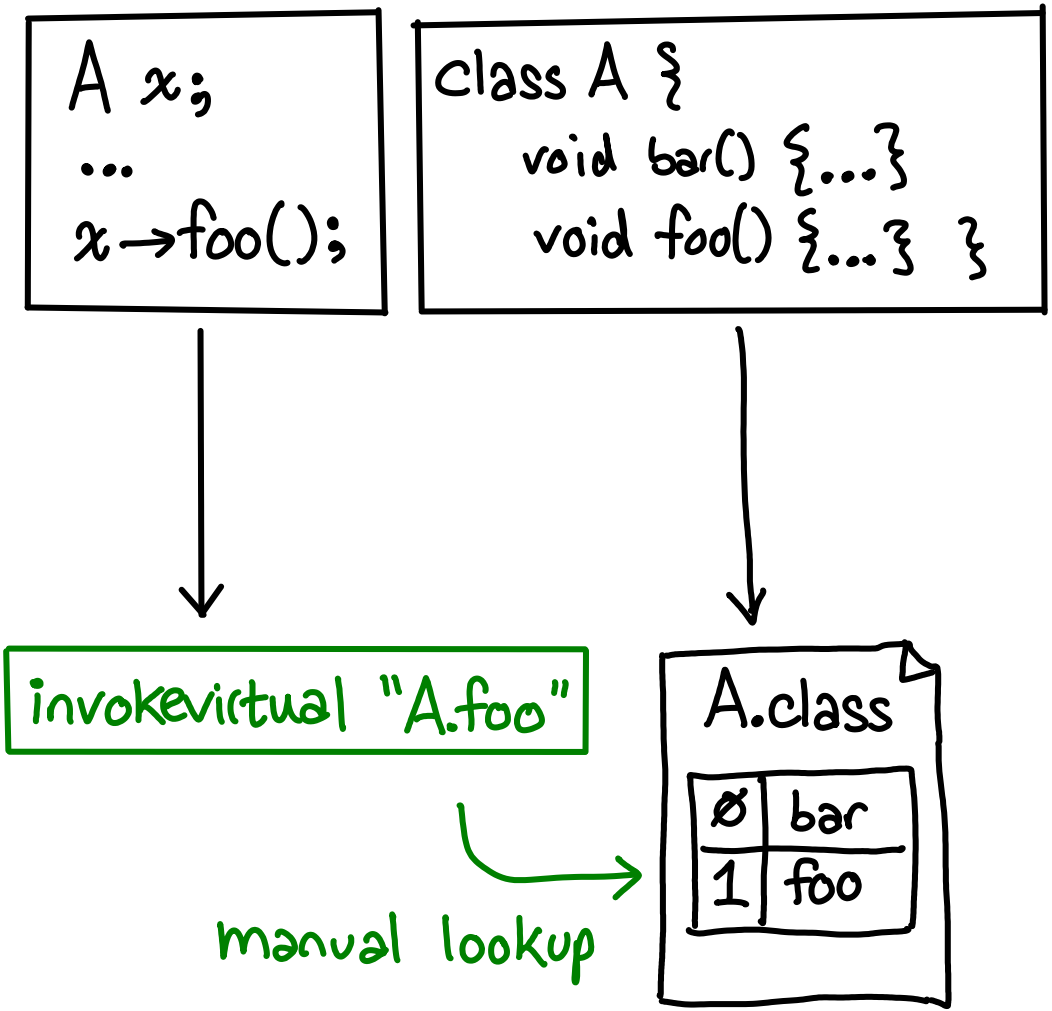
invokevirtual



how do you run this?

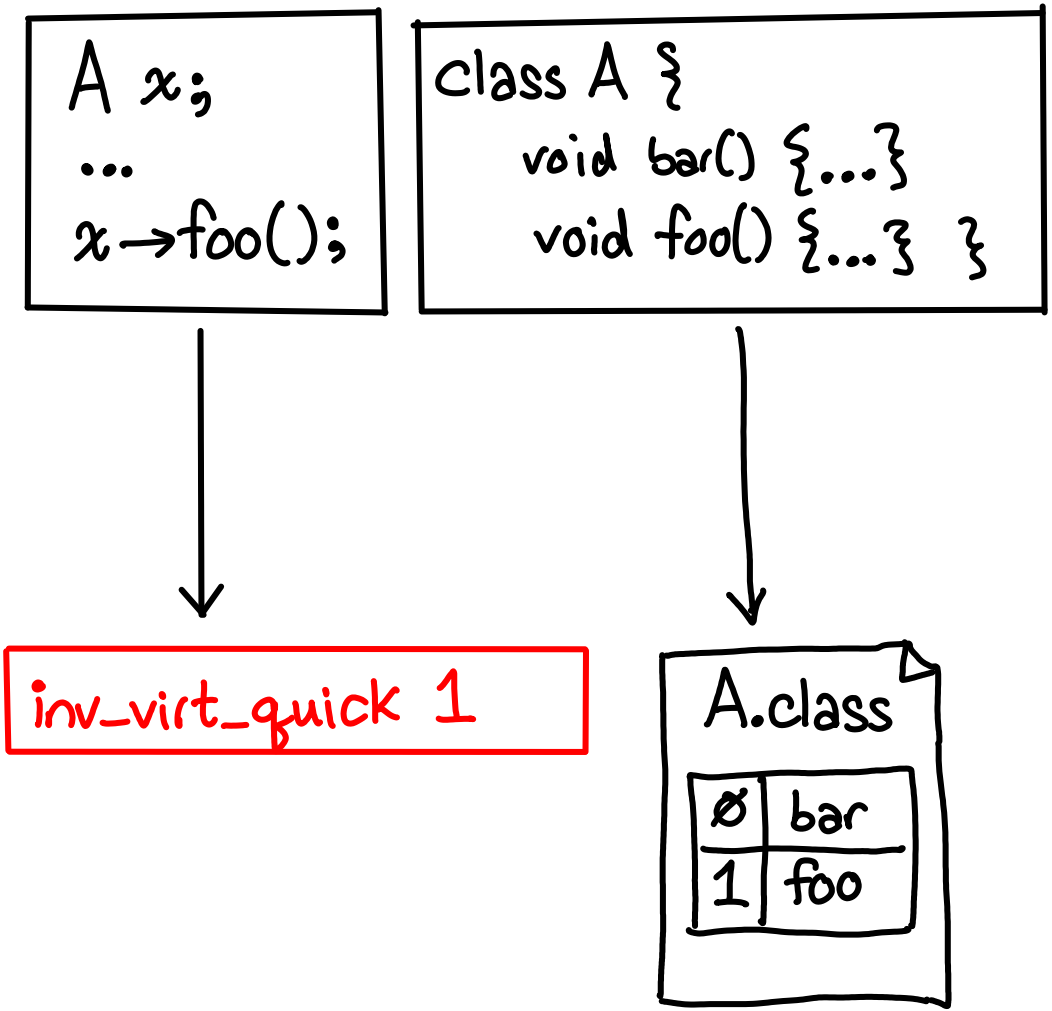
in Java

invokevirtual



in Java

invokevirtual



in Java

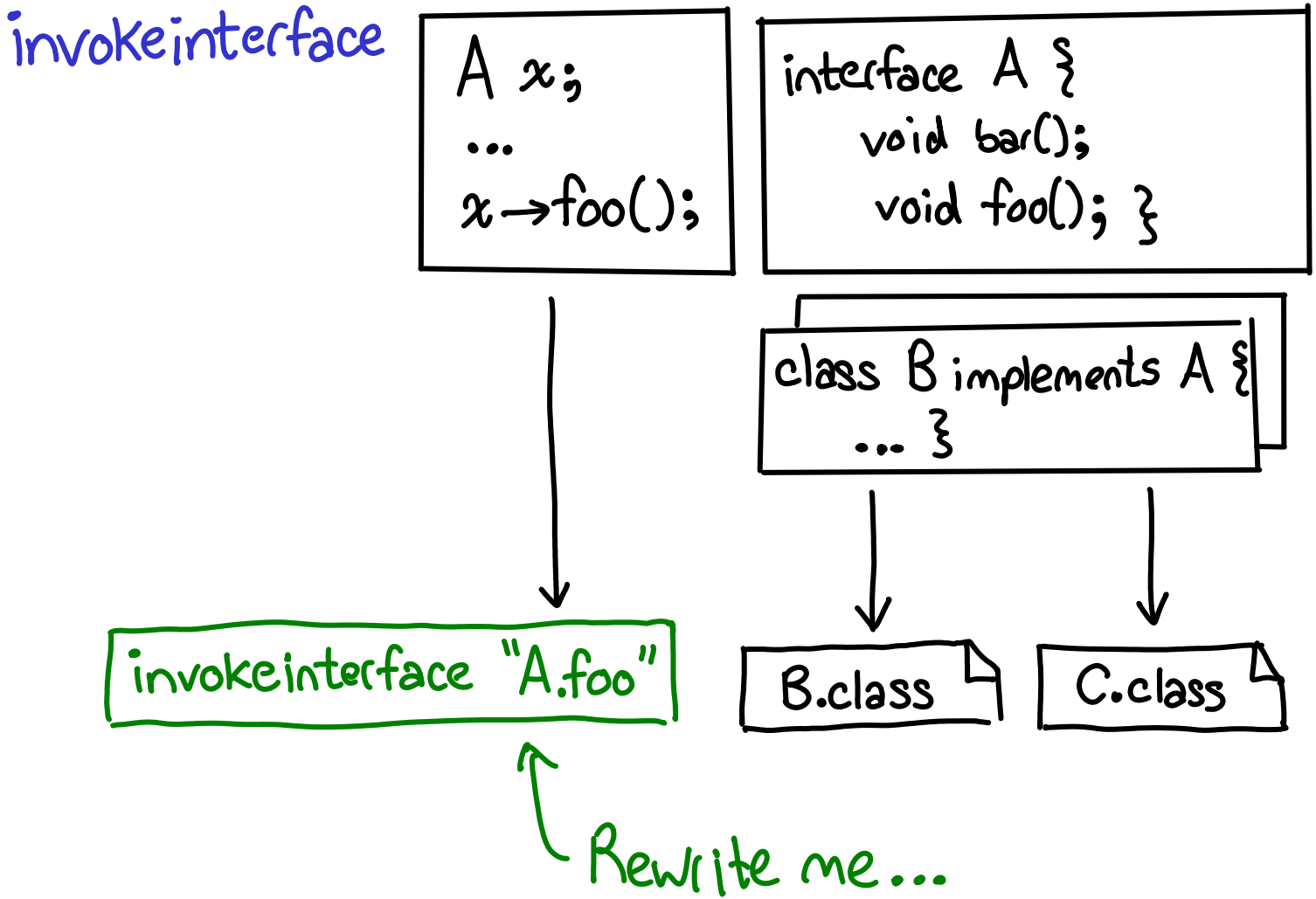
Big Idea #1: Rewrite code to make it more efficient

invokevirtual "A.foo" ⟶ inv_virt_quick 1

⇊

fast, C++-like machine code

What about Interfaces?


invokeinterface
A x;
...
x→foo();
interface A {
void bar();
void foo(); }
class B implements A {
... }
invokeinterface "A.foo"
B.class
C.class
Rewrite me...

invokeinterface

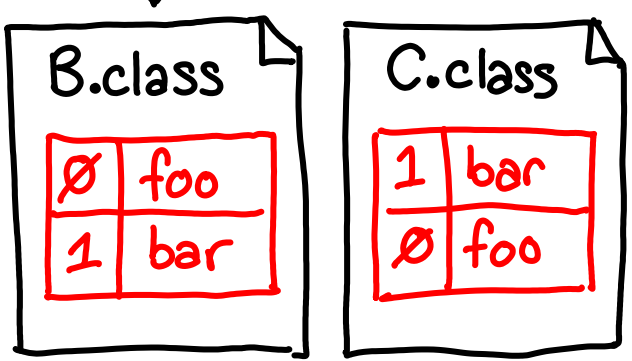
```
A x;
...
x->foo();
```

```
interface A {
    void bar();
    void foo(); }
```

```
class B implements A {
    ... }
```

???

to what?!



invokeinterface

```
A x;
...
x->foo();
```

```
interface A {
    void bar();
    void foo(); }
```

```
class B implements A {
    ... }
```

inv_int_quick "A.foo"

B.class	
0	foo
1	bar

C.class	
1	bar
0	foo

can we pick a specific implementation?

invokeinterface

cache

inv_int_quick "A.foo" <B.foo addr>

```
        if (this.class == B) {
fastpath:   directly invoke <B.foo addr>
        } else {
slowpath:   invoke interface "A.foo"
        }
```

Big Idea #2: A cache lookup can be built into the rewritten code, an inline cache

invokeinterface

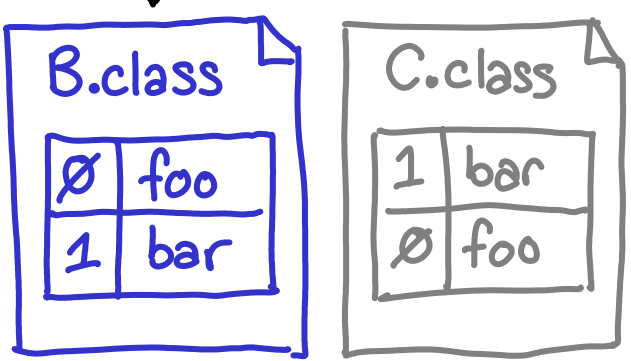
```
A x;
...
x->foo();
```

```
interface A {
    void bar();
    void foo(); }
```

```
class B implements A {
    ... }
```

What if this is the ONLY class loaded which implements A?

(Singleton class)



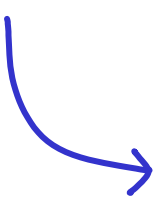
invokeinterface

inv_int_quicker <B.foo addr>

fastpath: directly invoke <B.foo addr>

call on A will always be B,
omit conditional

Corollary: Rewritten code does not have to be fully general, if you invalidate it when necessary.



Class Hierarchy Analysis

invokedynamic

In dynamic languages, usually have <10 distinct underlying types

Big Idea #3: Cache them all!

invokedynamic : Polymorphic Inline Cache

slow: invoke dynamic lookup handler

invokedynamic : Polymorphic Inline Cache

```
        if (this.class ==A) {
            directly invoke <A handler>
        } else {
slow:       invoke dynamic lookup handler
        }
```

invokedynamic : Polymorphic Inline Cache

```
        if (this.class == A) {
            directly invoke <A handler>
        if (this.class == B) {
            directly invoke <B handler>
        } else {
slow:       invoke dynamic lookup handler
        }
```

invokedynamic : Polymorphic Inline Cache

```
if (this.class == A) {
    directly invoke <A handler>
if (this.class == B) {
    directly invoke <B handler>
if (this.class == C) {
    directly invoke <C handler>
} else {
slow:   invoke dynamic lookup handler
}
```

invokedynamic : Polymorphic Inline Cache

```
if (this.class == A) {
    directly invoke <A handler>
if (this.class == B) {
    directly invoke <B handler>
if (this.class == C) {
    directly invoke <C handler>
} else {
slow:   invoke dynamic lookup handler
}
```

order by freq.

invokedynamic : Polymorphic Inline Cache

orlginated in Self

dynamically typed OOP language

PICs $\longmapsto$ 37% perf improvement

Summary:

JIT codegen = Flexibility

allows Java/JavaScript engines to avoid paying too much for indirection

(end)