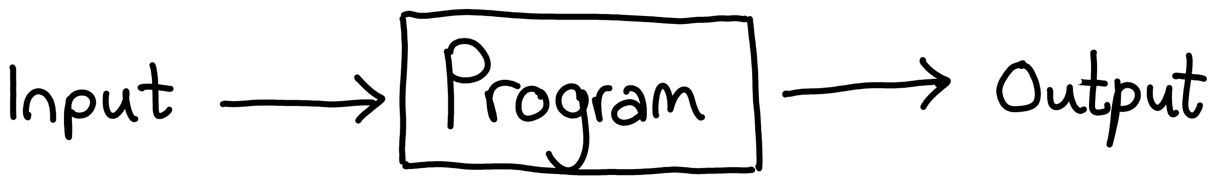
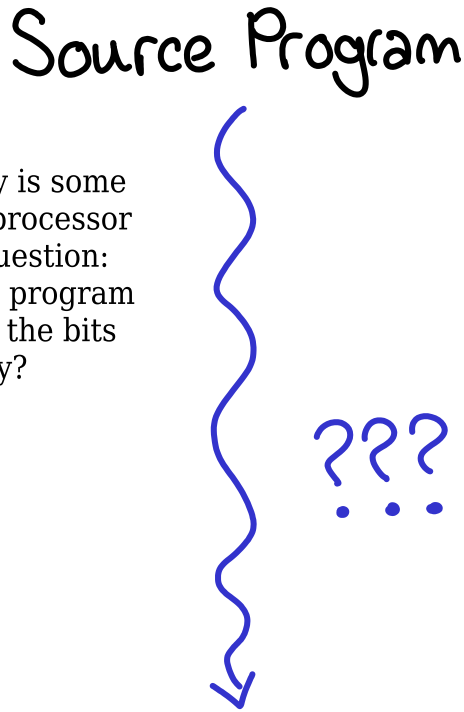
Implementation

Edward Z. Yang

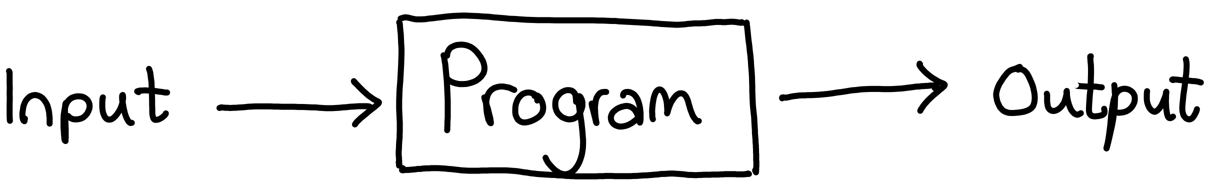
(intro)

The most basic picture of computation is that we have some program, which takes some input and produces some output.





The program, however, usually is some sort of native format which a processor can run, which presents the question: how do we transform a source program (of human readable code) into the bits and bytes of machine assembly?



Source Program

$\downarrow$

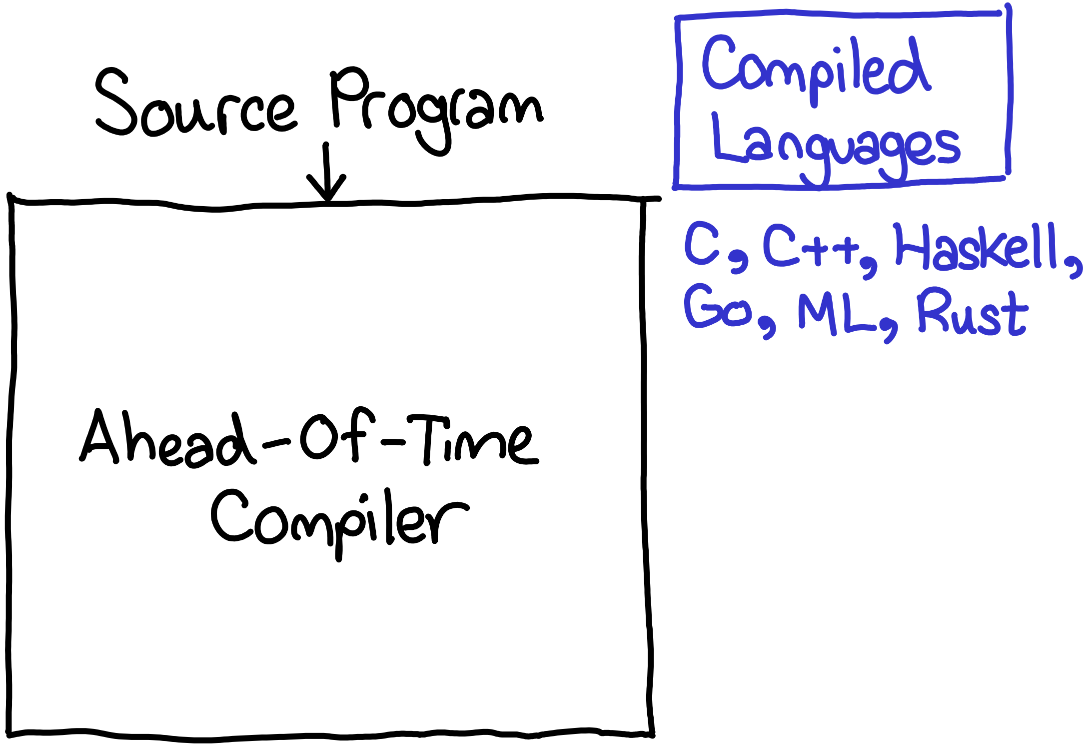
Interpreted Languages

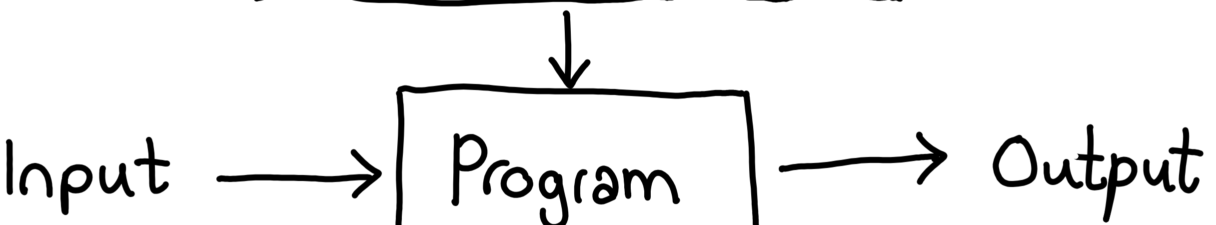
Python, Ruby, PHP, Perl, MATLAB...

One way to do this is with an "interpreter", which takes the source program as an additional input and executes the program by interpreting the code as a universal machine. This strategy is quite common, and it is probably the easiest way to write an implementation of a programming language. It's also quite possibly the very slowest way to implement a language.

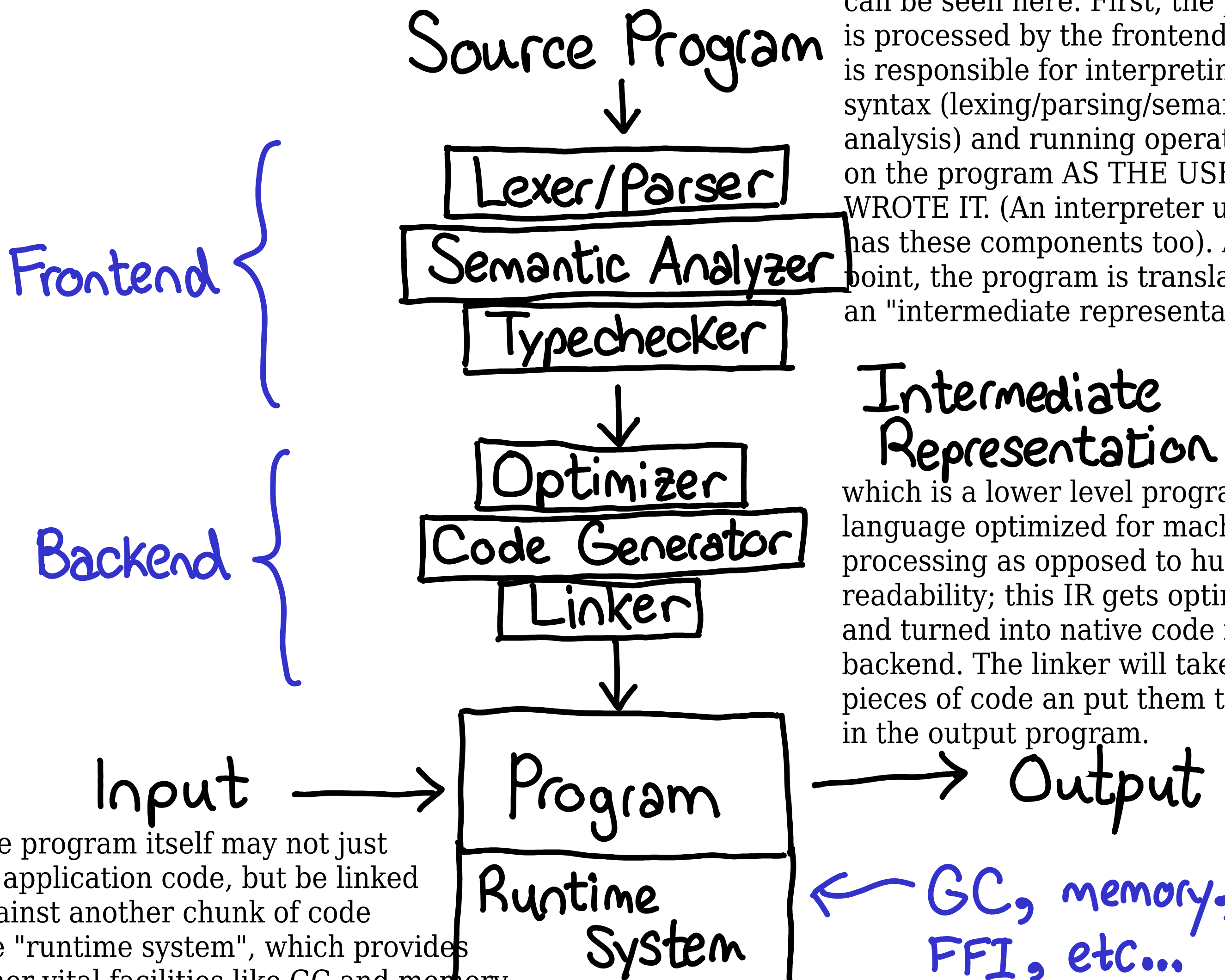
Input $\rightarrow$ $\boxed{\text{Interpreter}}$ $\rightarrow$ Output

Another strategy is to take the source program and "compile" it into a native code program that can run the code. These compilers are often referred to as "Ahead-Of-Time" (AOT) compilers because the compilation occurs BEFORE we actually need to run the program.





The basic architecture of a compiler can be seen here. First, the program is processed by the frontend, which is responsible for interpreting the syntax (lexing/parsing/semantic analysis) and running operations on the program AS THE USER WROTE IT. (An interpreter usually has these components too). At some point, the program is translated into an "intermediate representation"

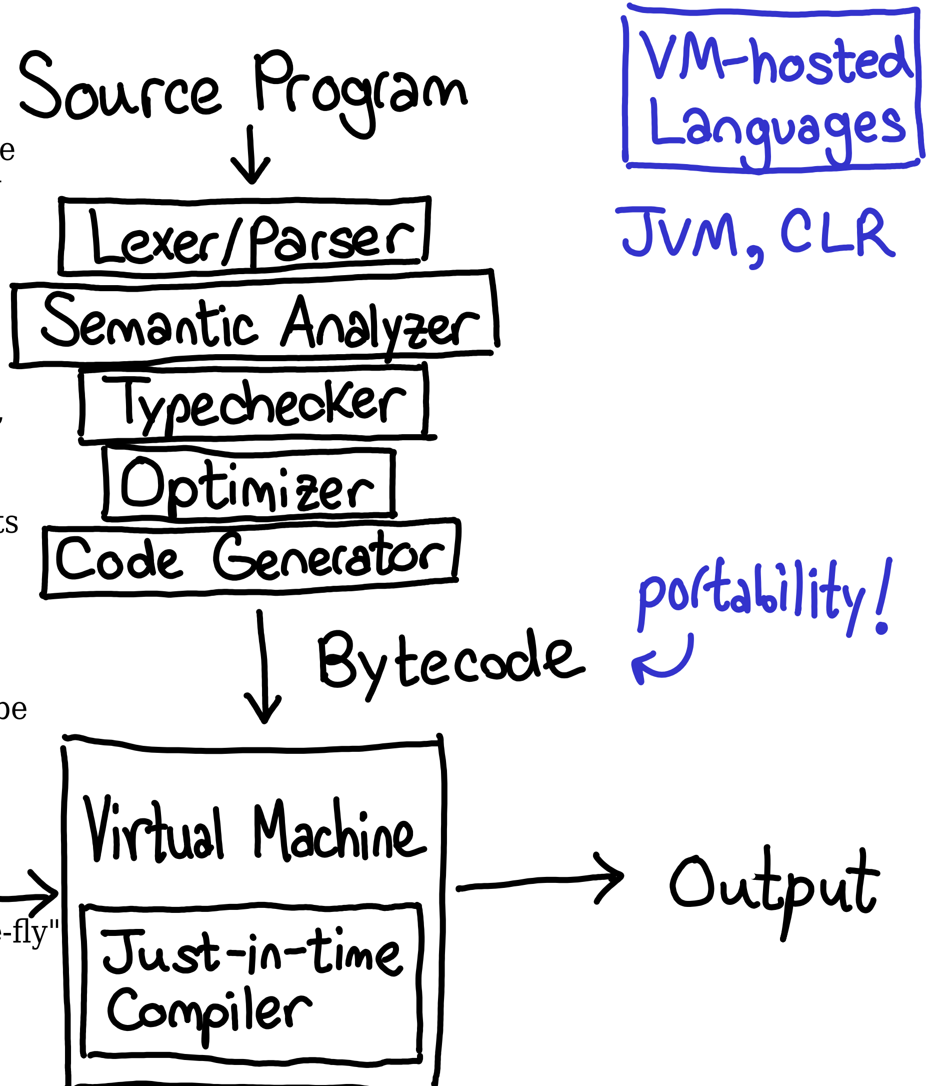


which is a lower level programming language optimized for machine processing as opposed to human readability; this IR gets optimized and turned into native code in the backend. The linker will take various pieces of code an put them together in the output program.

The program itself may not just be application code, but be linked against another chunk of code the "runtime system", which provides other vital facilities like GC and memory.

Another variation on the compilation scheme is to compile to a bytecode, instead of native code. Native code is architecture specific, so you have to compile a program separately for every architecture you want to support. A bytecode format is usually some language which is very similar to assembly (low level), but is also portable. The compiler can generate bytecode, which is then fed to a virtual machine (really a glorified interpreter) which now interprets the bytecode stream: because bytecodes are relatively simple, this operation can be fast.

Further performance gains can be had if the virtual machine is equipped with a just-in-time compiler, which can compile bytecodes to native code "on-the-fly" as the program is executing.

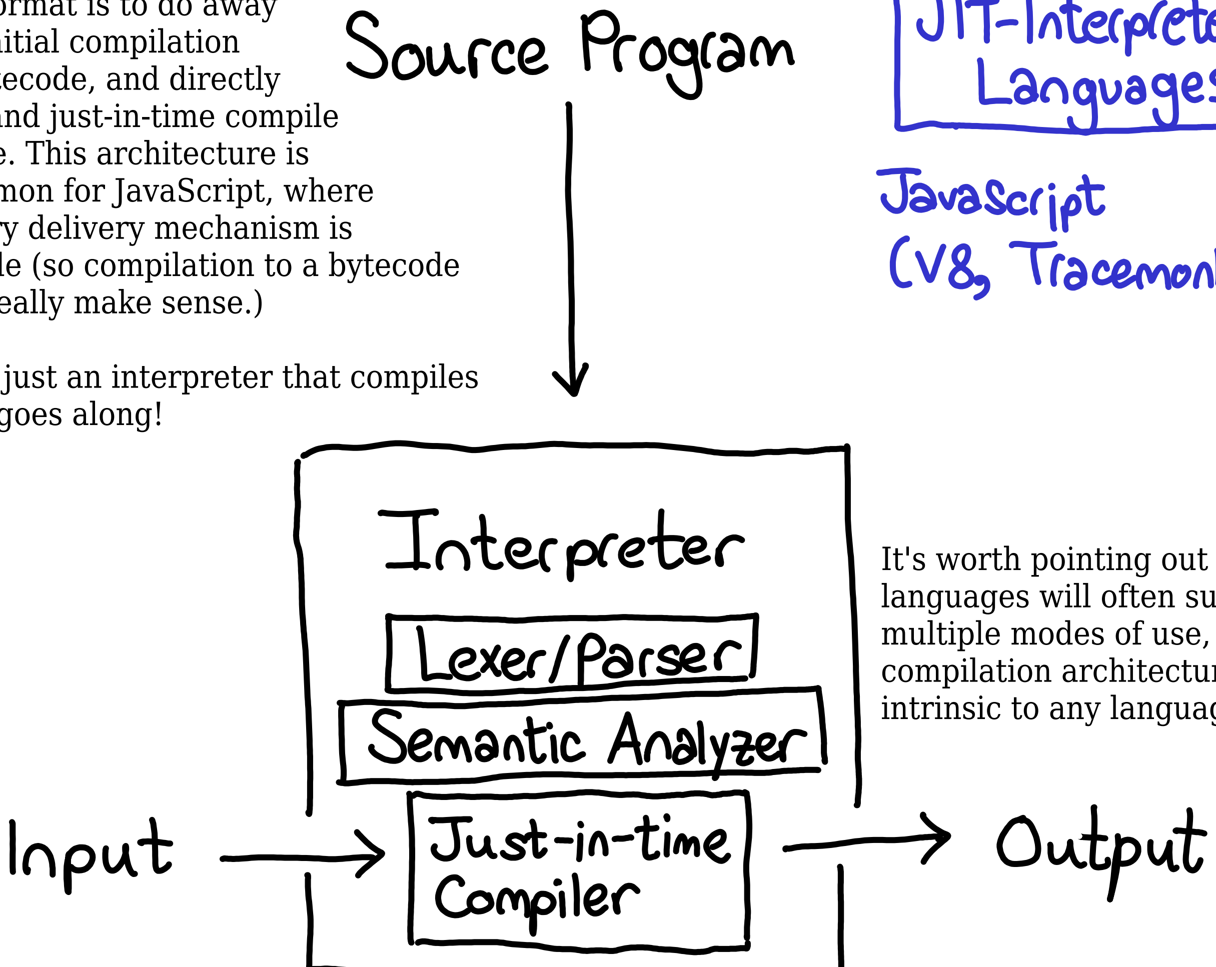


JIT-Interpreted Languages

JavaScript
(V8, Tracemonkey)

Another twist on the virtual machine format is to do away with the initial compilation step to bytecode, and directly interpret and just-in-time compile a language. This architecture is quite common for JavaScript, where the primary delivery mechanism is source code (so compilation to a bytecode wouldn't really make sense.)

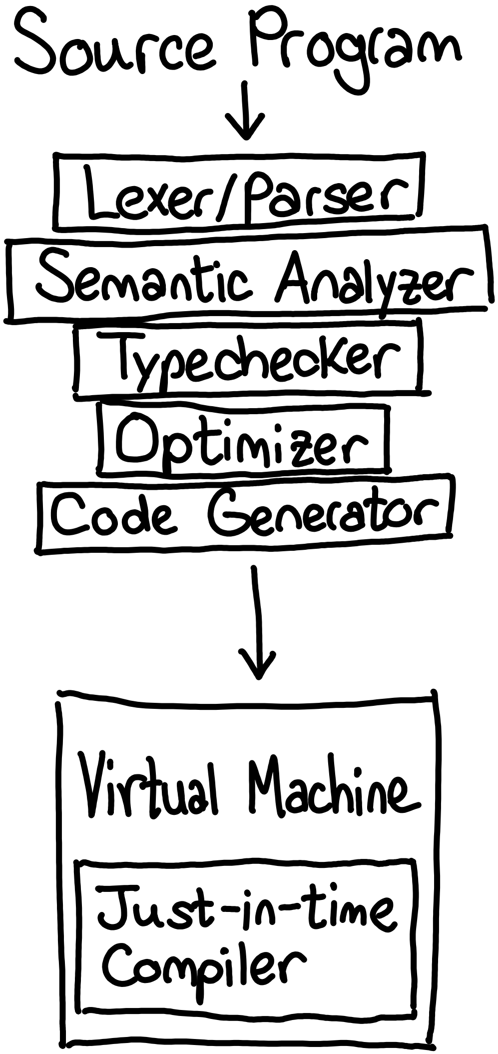
Really, it's just an interpreter that compiles code as it goes along!

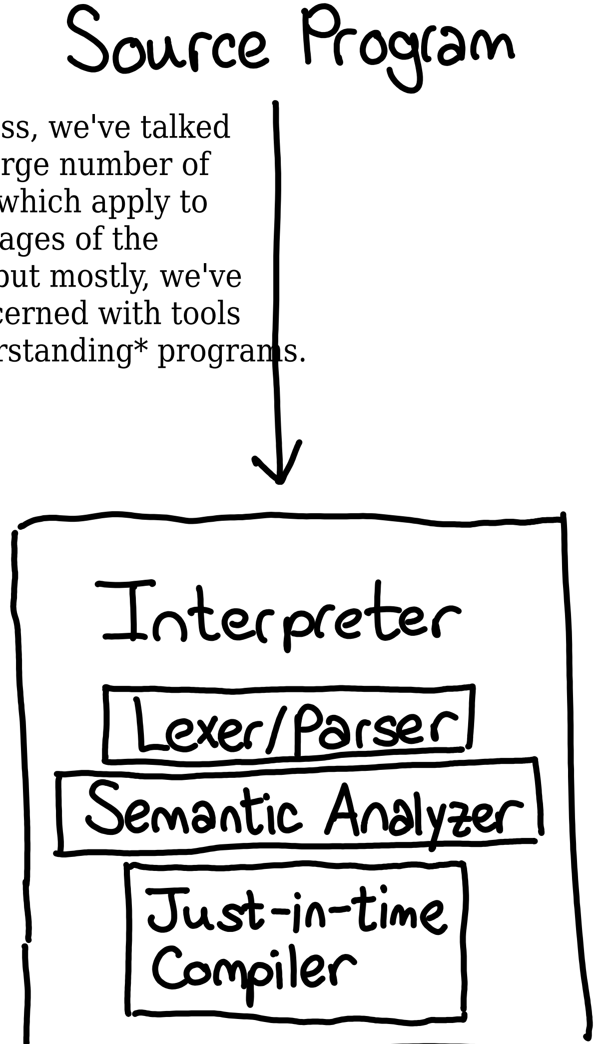


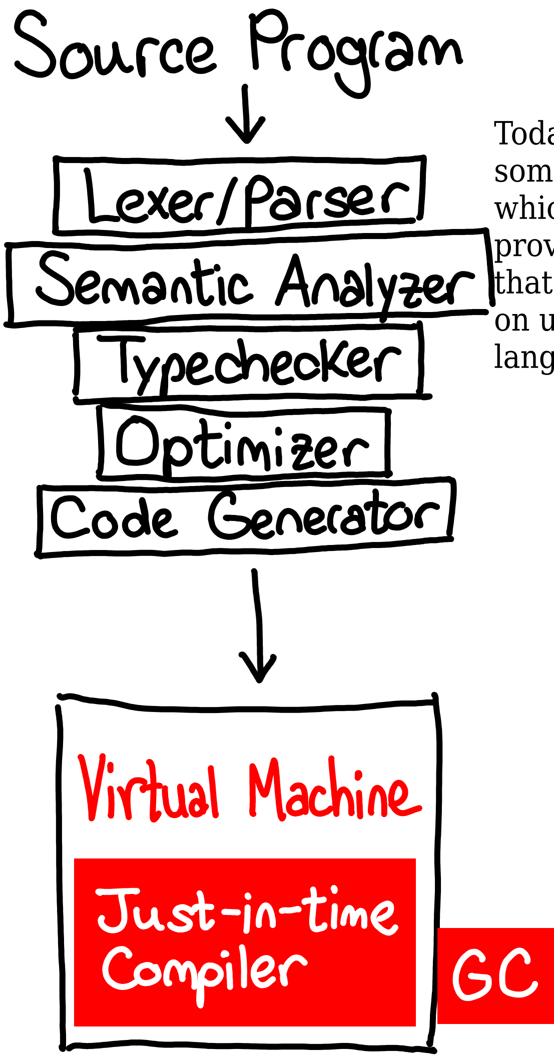
It's worth pointing out that languages will often support multiple modes of use, so the compilation architecture is not intrinsic to any language.

Languages often support multiple implementation

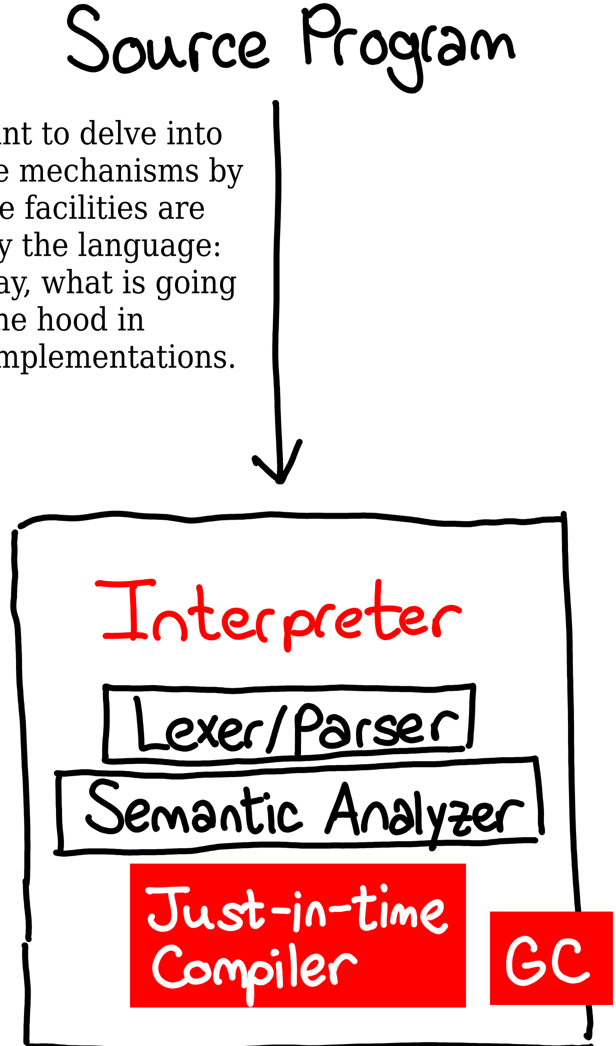
In this class, we've talked about a large number of concepts which apply to various stages of the pipeline; but mostly, we've been concerned with tools for *understanding* programs.







Today, I want to delve into some of the mechanisms by which those facilities are provided by the language: that is to say, what is going on under the hood in language implementations.



- Garbage Collection

- Dynamic Dispatch in a JIT

compare with C++ vtables

There are two primary topics I want to cover (unfortunately, they're a little disjoint from each other). First, I want to talk about garbage collection, which provides one of the most important abstractions that a language with a runtime will give you: the illusion of infinite memory. Second, I want to talk about how one goes about implementing dynamic dispatch when you have the ability to just-in-time compile code. In particular, I want to demonstrate how the tradeoffs are different as opposed to C++, where no such dynamic code (re)writing is possible.

I COULD BE BOUNDED IN A NUTSHELL
AND CALL MYSELF THE KING OF INFINITE SPACE

HAMLET ACT II SCENE II

Lambda Calculus

Activation-record Model

One of the themes in this course has been the idea of using abstract models to reason about our programs. Thus, we've talked about programs in terms of the substitution model in lambda calculus, or the activation-record model in the presence of mutation, and tried to avoid making reference to the underlying machine architecture or memory hierachy.

x86 machine architecture

Memory Hierarchy

INFINITE MEMORY



One interesting implication about this, is that in all of the models we have talked about, we have had some sort of assumption of "infinite memory": that it doesn't matter how big a lambda term gets or how many activation records were allocated; somehow, there'd always be enough space. Of course, in reality, there isn't, and we have to be economical in our use of memory, lest we run out of it. So in low level languages like C, memory must be manually managed; specifically, you should free memory when you're done with it.

FINITE MEMORY

INFINITE MEMORY

GARBAGE COLLECTION

Garbage collection is the abstraction barrier by which we can give programmers the illusion of infinite memory. The idea is that if a user allocated some data which now, provably, will never be used again, we can reclaim that memory and use it for something else.

FINITE MEMORY

& reuse space which provably
will never be used again

Managed Memory

Really, there are two abstractions involved here.

Memory Management

Managed memory is the abstraction that gives us the illusion of infinite memory. We can allocate objects and don't have to worry about freeing them. The price we pay is we must respect pointers as opaque types which cannot be synthesized.

allocate

Interface for allocating objects.
Pointers are opaque.

Garbage collection / Reference counting

Managed memory is built on top of a more low-level memory API which is based on allocating and freeing explicit chunks of memory. This API is provided by your operating system and hardware.

malloc
free

Interface for explicitly allocating/deallocating finite memory

Operating System & Hardware

Garbage Collection

This is not really a compiler implementors course, so we're just going to survey three of the most important ideas for how GC is implemented. These are real algorithms and the way they are implemented in languages is not too far from the descriptions here.

- Reference Counting

  ARC, Perl, PHP, Python

  The Cycle Problem

- Tracing Collection

  Java, Haskell, ML, Lisp, Go, JavaScript

  Mark and Sweep

  Copying Collection

The basic goal of GC is this:

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

The biggest question of GC is, "How do I know when I'm done using an object?"

IDEAL

Object has no causal influence on future program execution

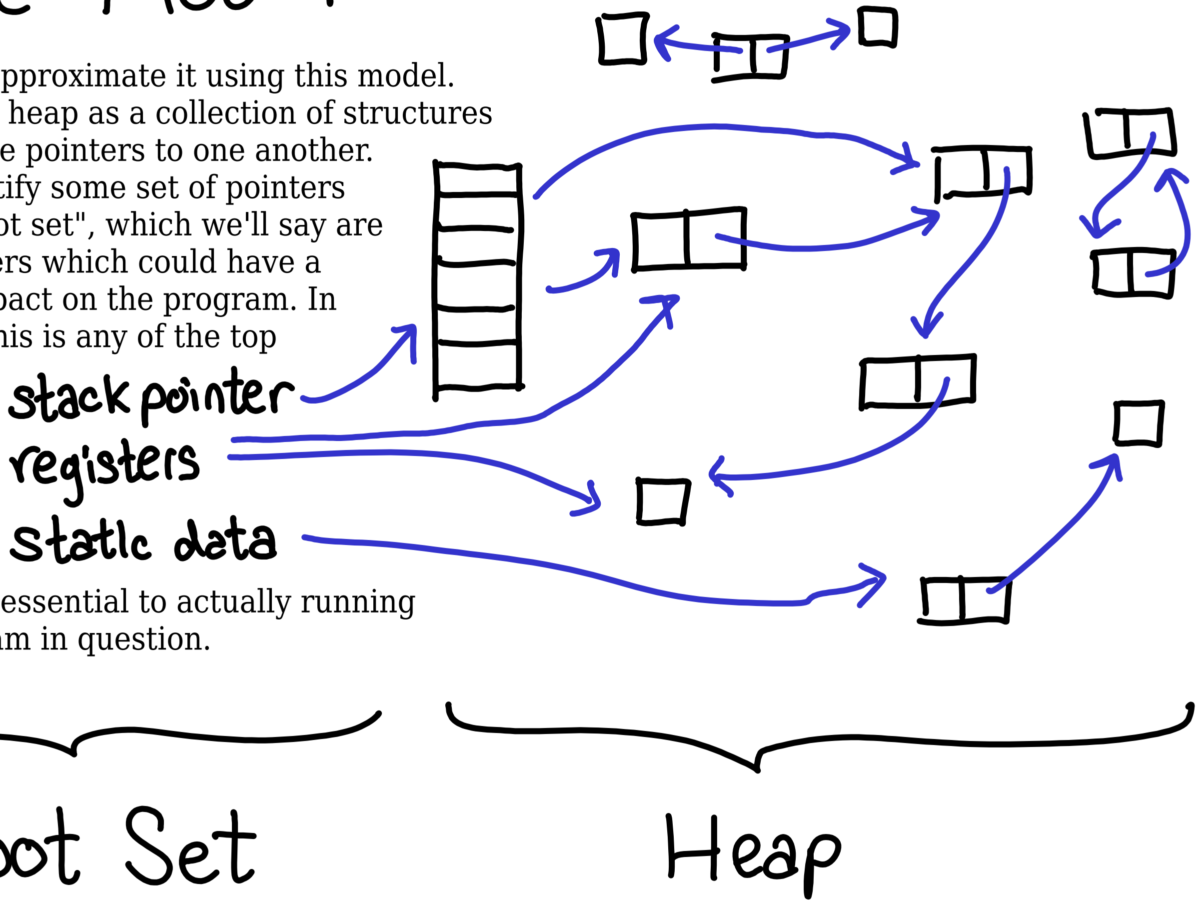
When I am done using an object,

free its memory

We won't be able to achieve this ideal, however...

The Model

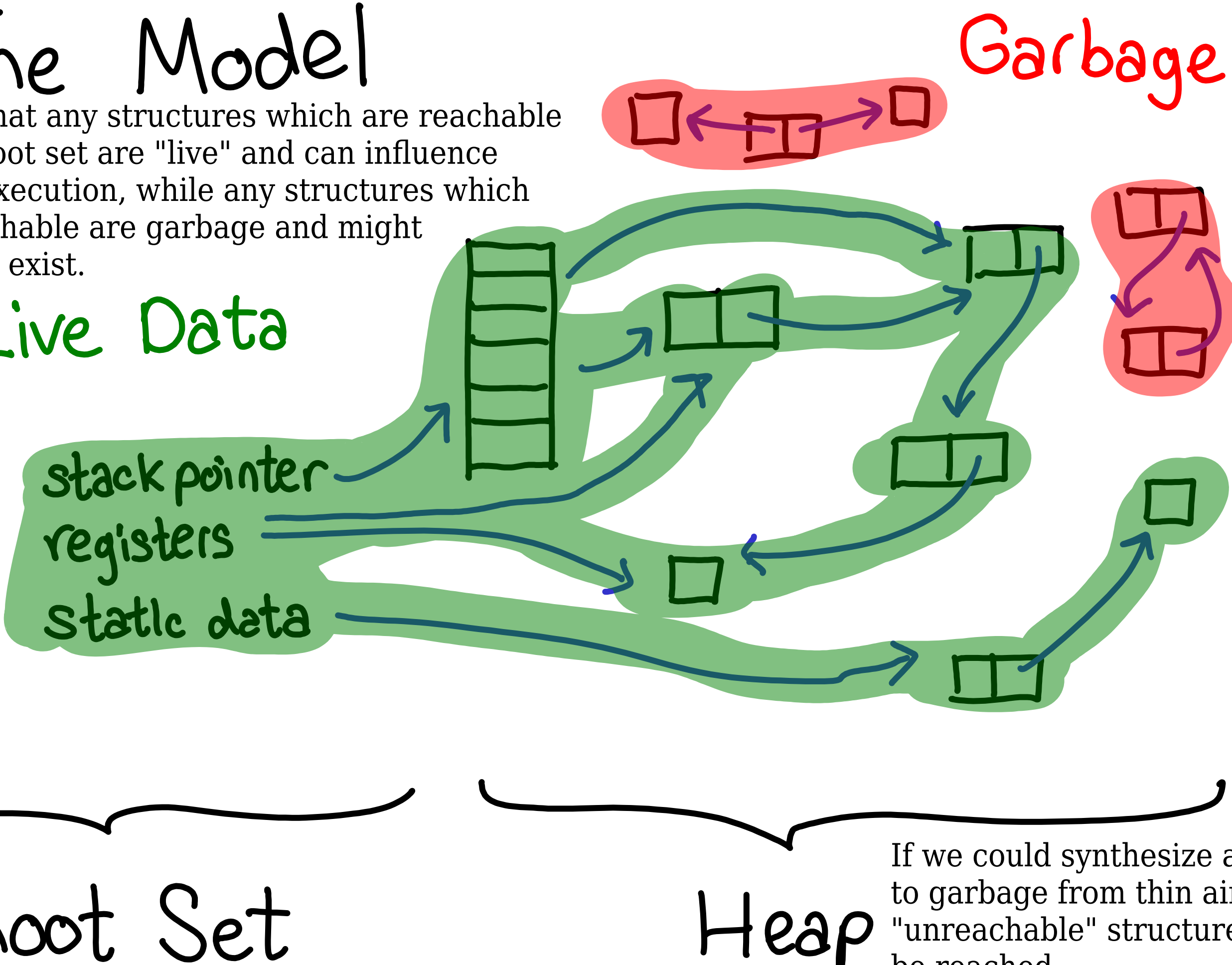
So, we'll approximate it using this model. Model the heap as a collection of structures which have pointers to one another. We'll identify some set of pointers as the "root set", which we'll say are any pointers which could have a causal impact on the program. In practice this is any of the top



level data essential to actually running the program in question.

The Model

We'll say that any structures which are reachable from the root set are "live" and can influence program execution, while any structures which are unreachable are garbage and might as well not exist.

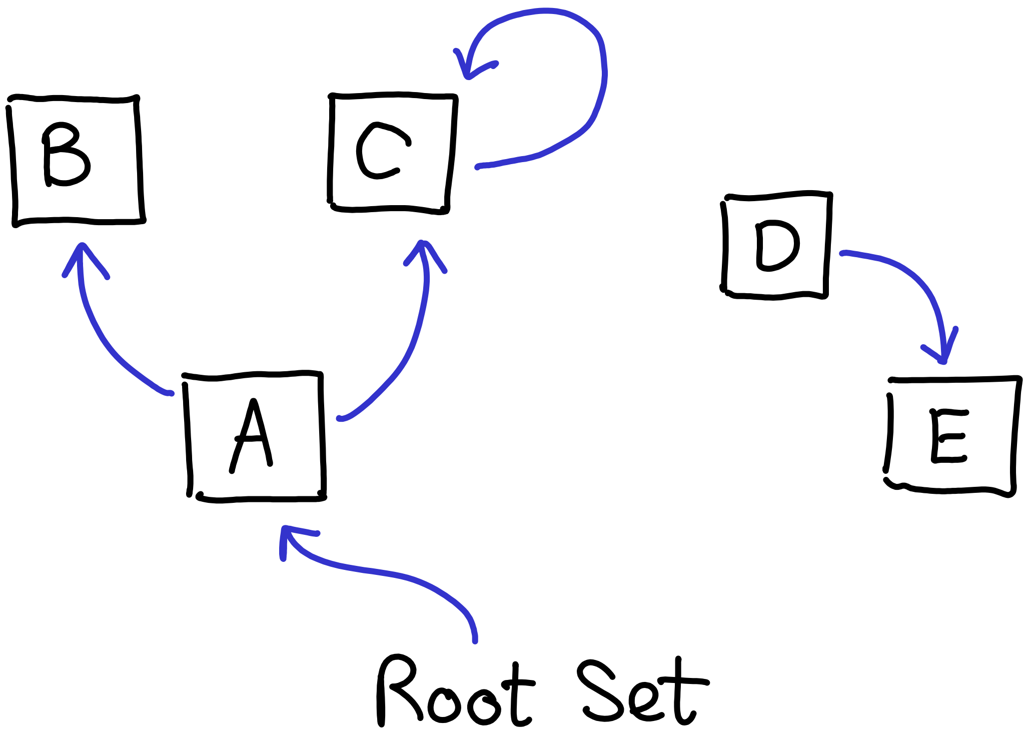


If we could synthesize a pointer to garbage from thin air, then "unreachable" structures could be reached.

Why must pointer arithmetic be disallowed?

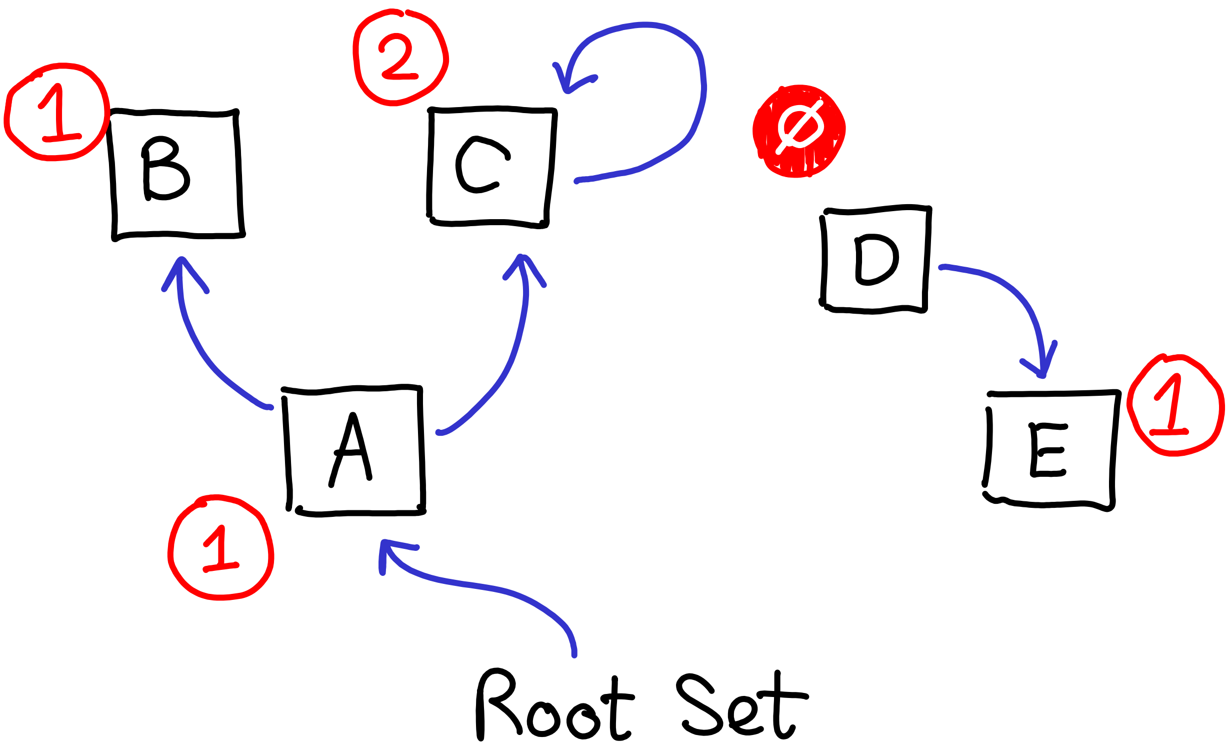
Reference counting

Count the number of incoming references



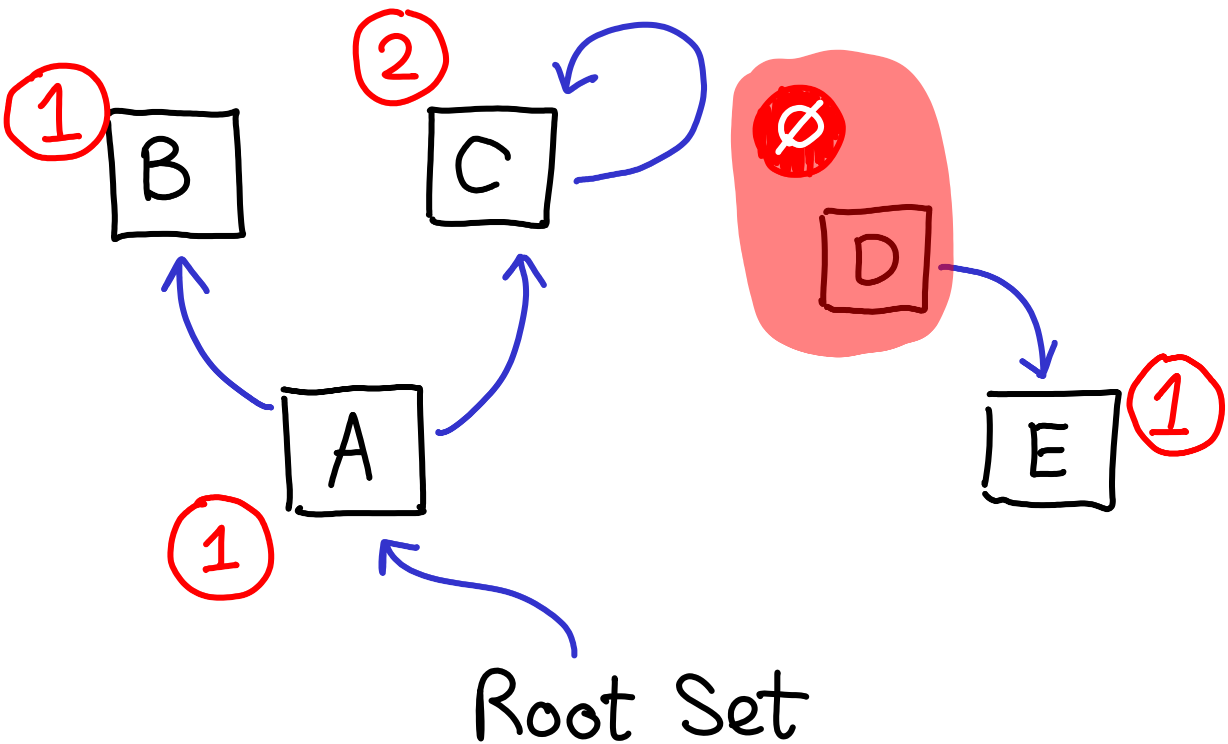
Reference counting

Count the number of <u>incoming references</u>



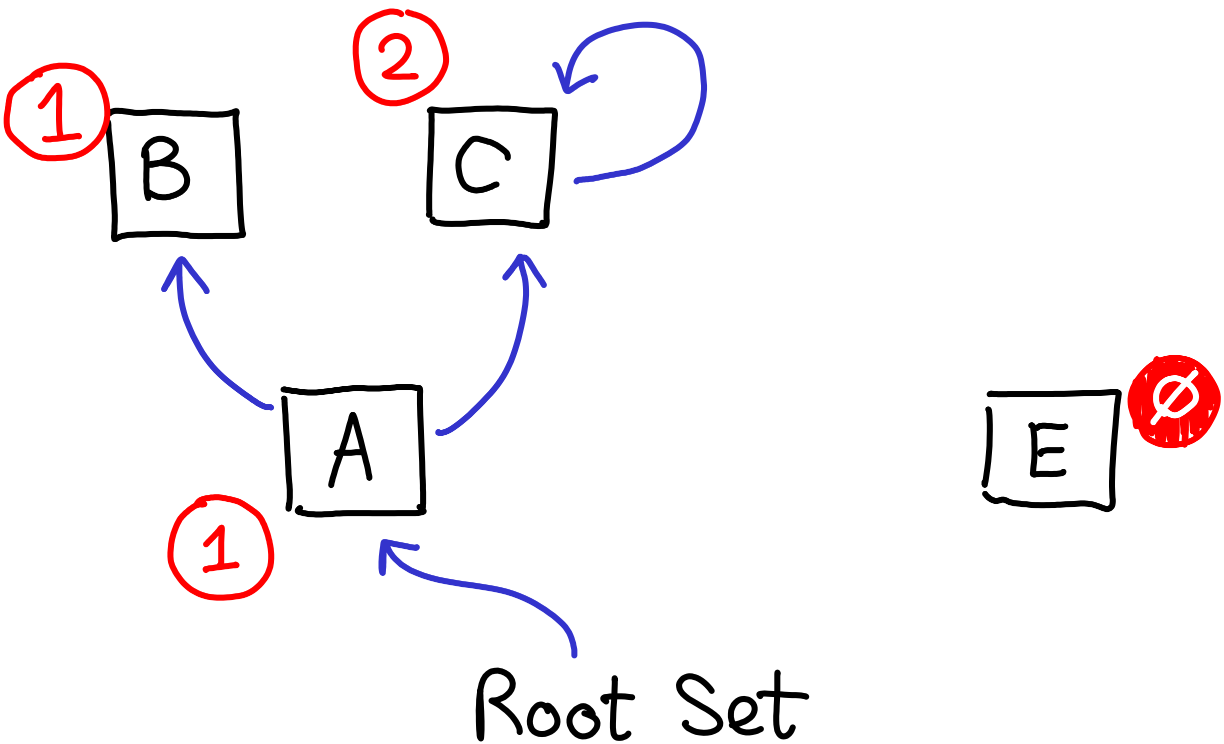
Reference counting

Count the number of incoming references



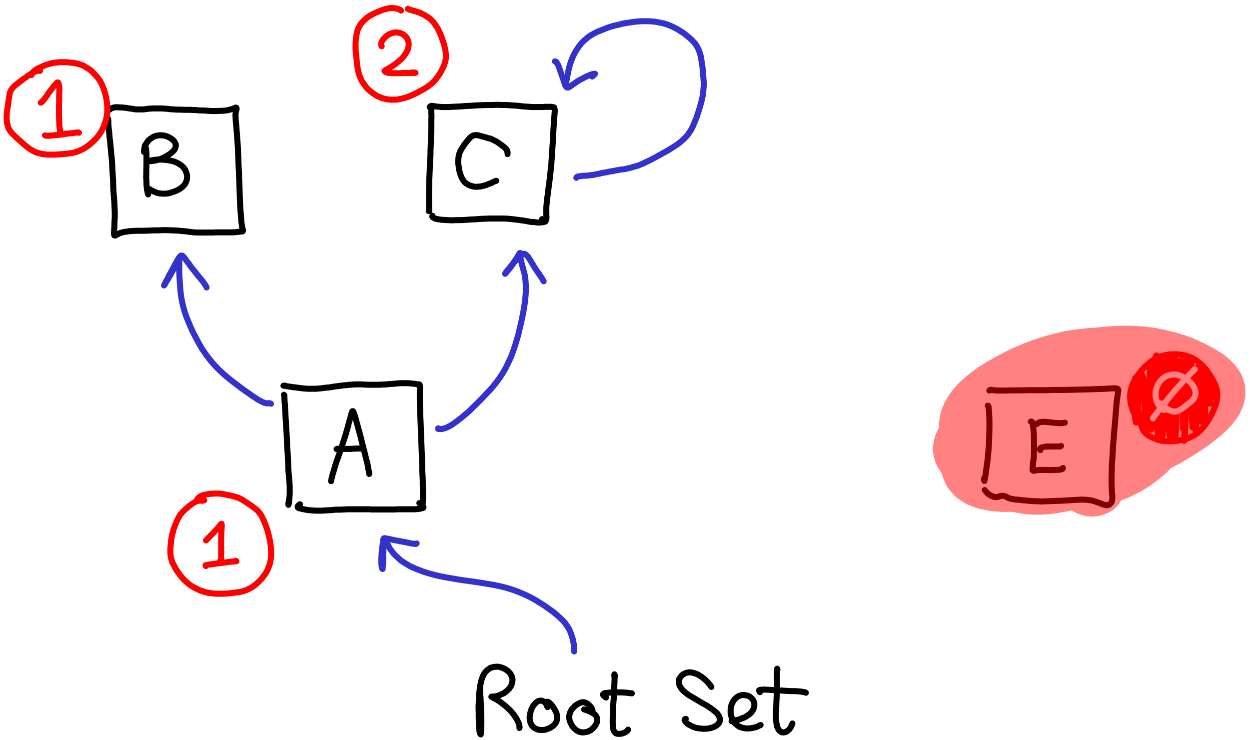
Reference counting

Count the number of <u>incoming references</u>



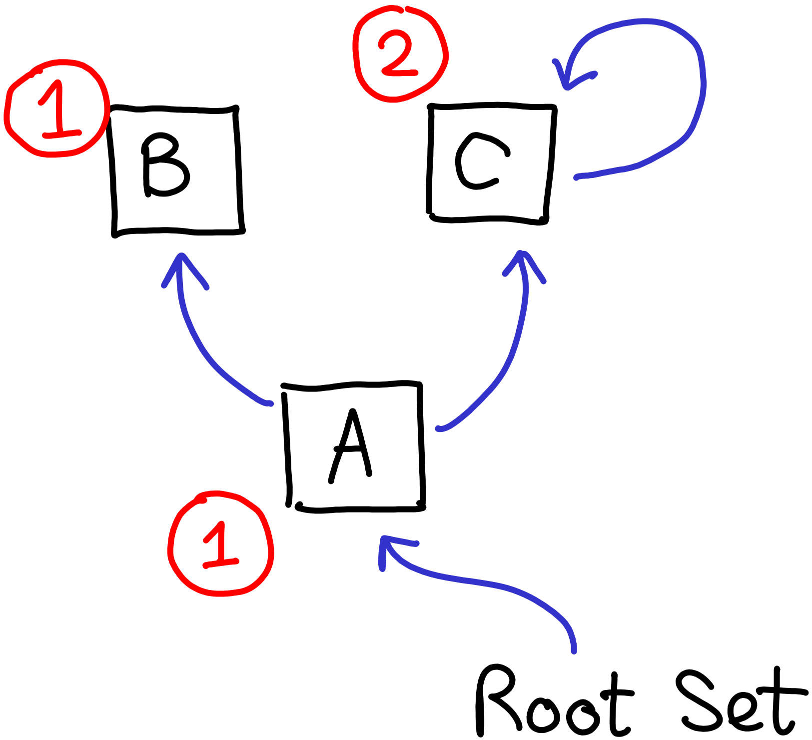
Reference counting

Count the number of <u>incoming references</u>



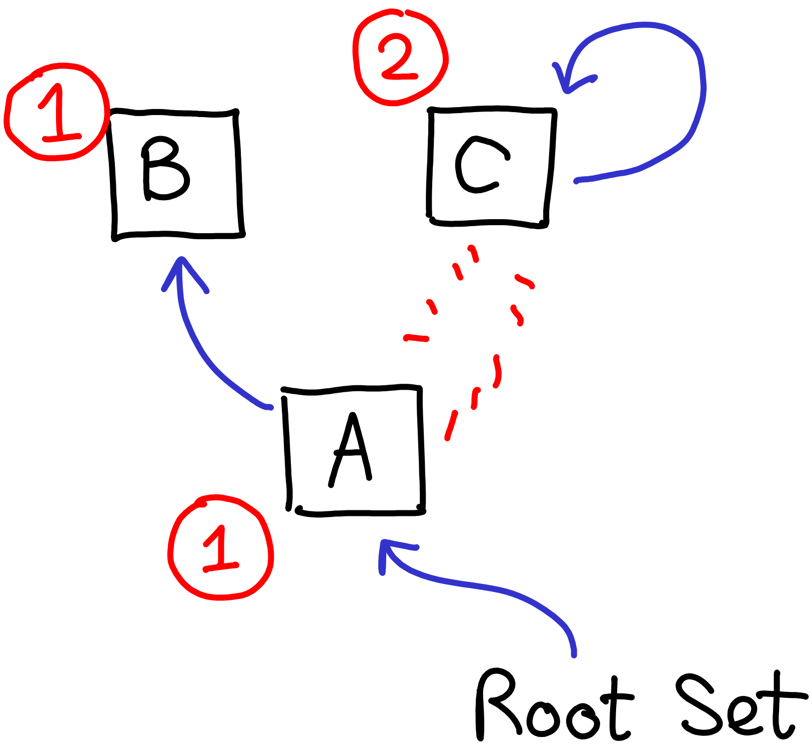
Reference counting

Count the number of <u>incoming references</u>



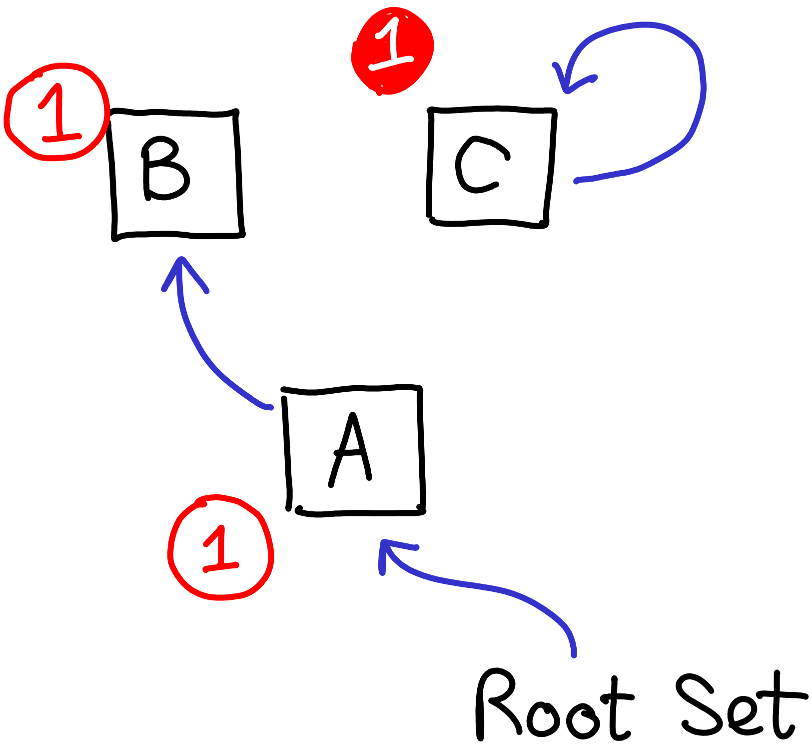
Reference counting

Count the number of <u>incoming references</u>



Reference counting

Count the number of incoming references



Reference counting

Count the number of incoming references

✓ Very easy to implement

✓ Objects immediately deallocated

This also means that finalizers (snippets of code that run when an object becomes dead) are much easier to implement with refcounts.

✗ Cycles never die! (cycle-breaking)

Refers to the practice of manually breaking cycles to ensure memory gets deallocated.

✗ Storing & updating counts is costly

Update a count for every pointer manipulation!

✗ Sychronizing updates

And it gets worse when things are multithreaded, because now updates to the counter have to be synchronized. (Notice that this has nothing to do with whether the object itself is synchronized; incoming references can be from disjoint objects.)

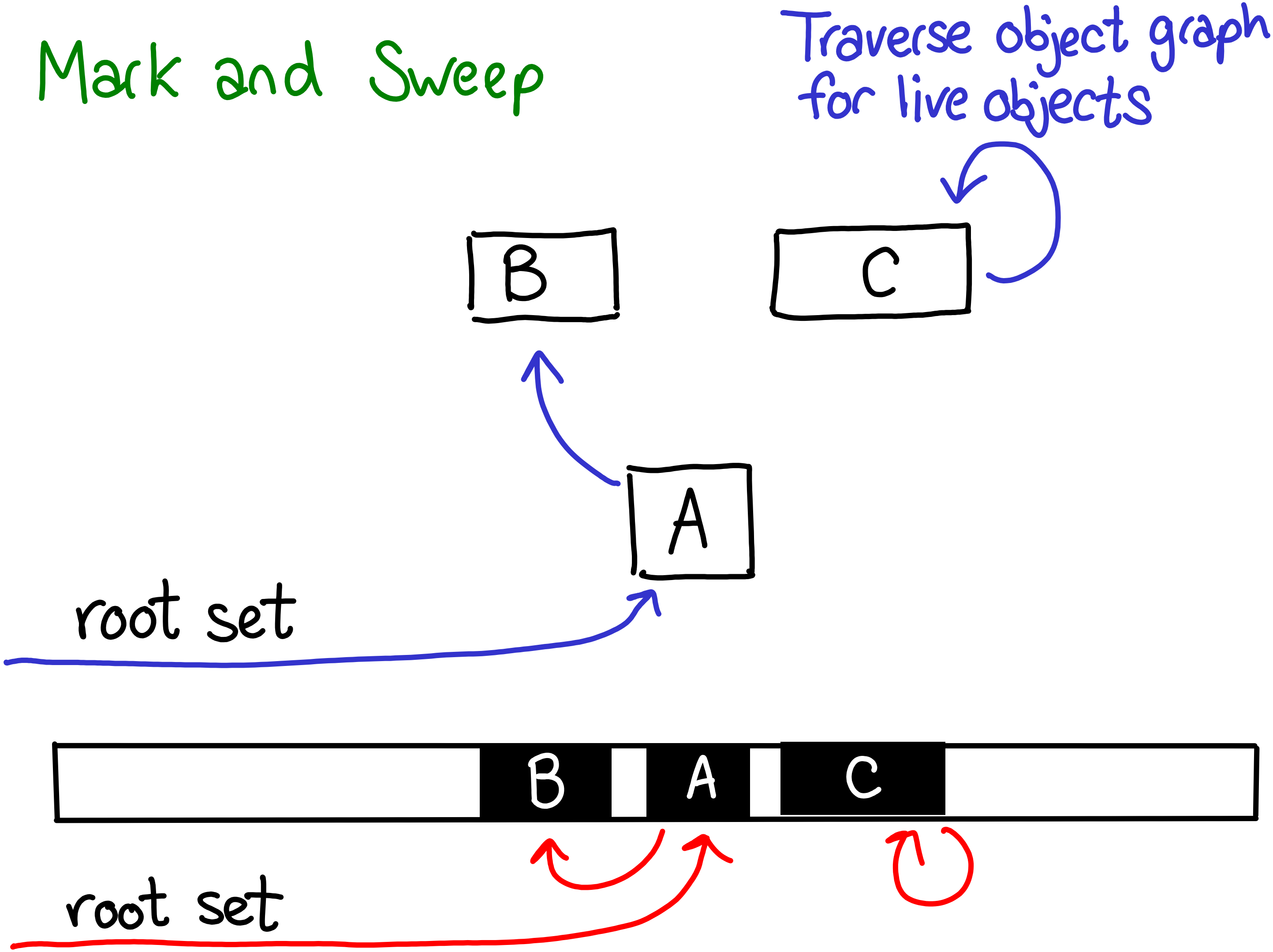
Reference counting

Count the number of <u>incoming references</u>

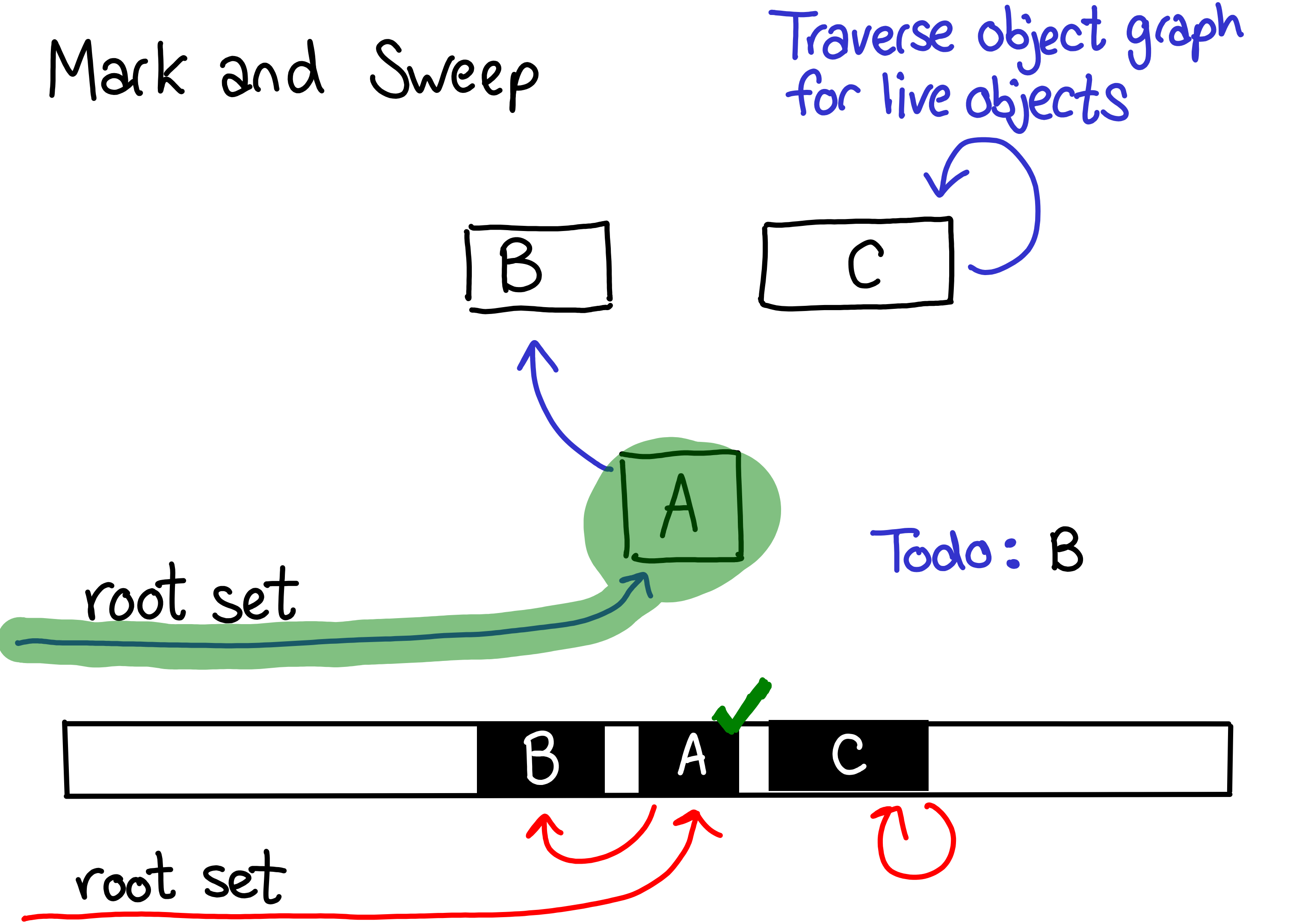
- ✓ Very easy to implement
- ✓ Objects immediately deallocated
- ✗ Cycles never die! (cycle-breaking)
- ✗ Storing & updating counts is costly
- ✗ Sychronizing updates

So the next scheme I'm going to describe will fix these problems.

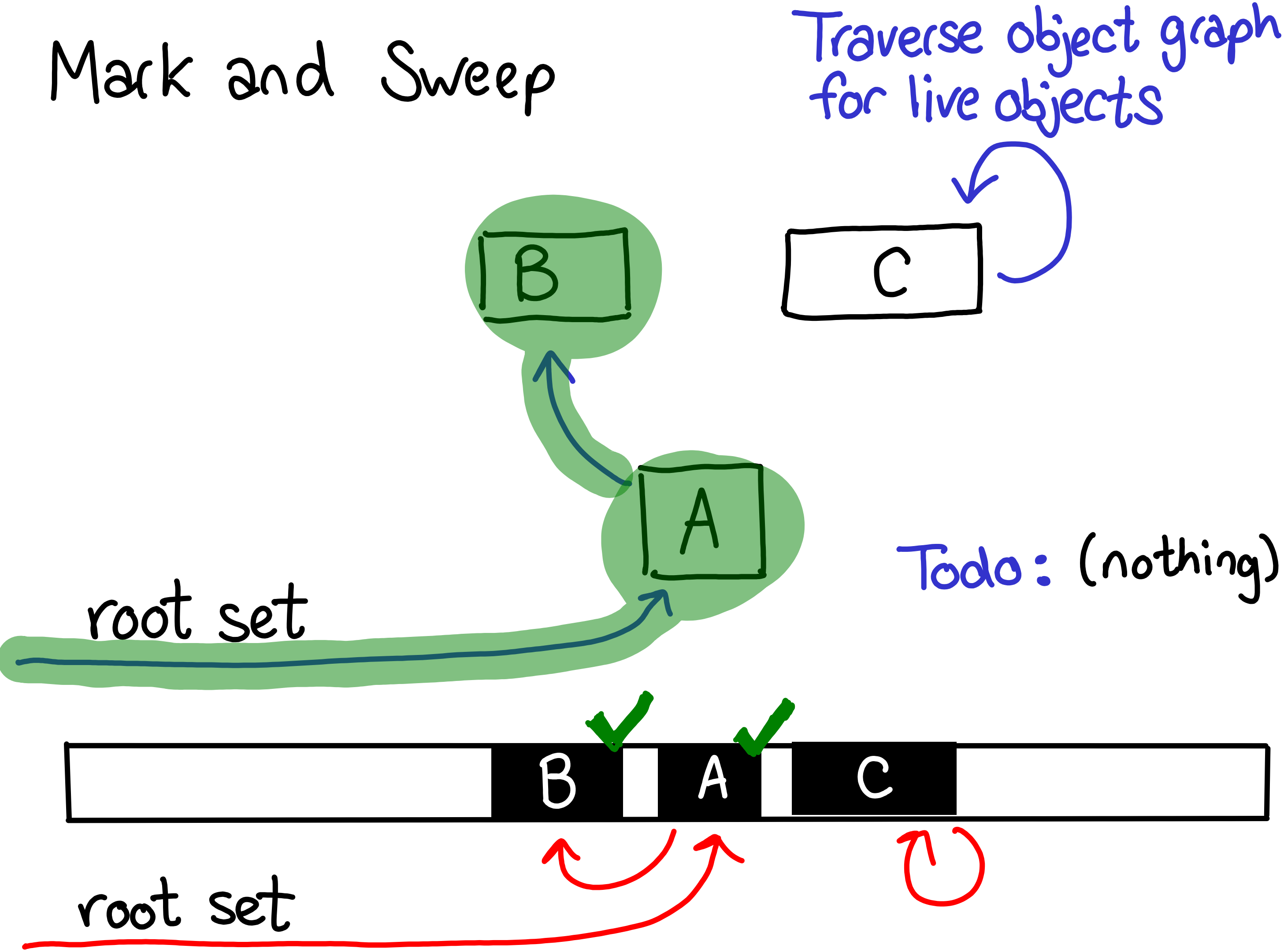
Mark and Sweep



Mark and Sweep

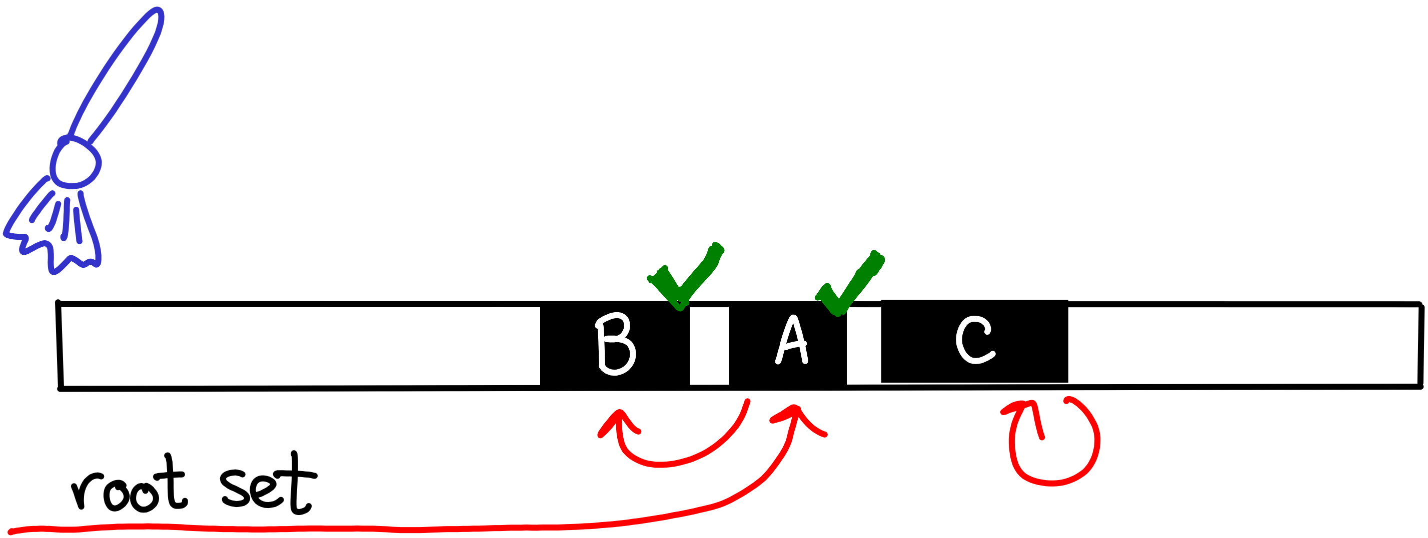


Mark and Sweep



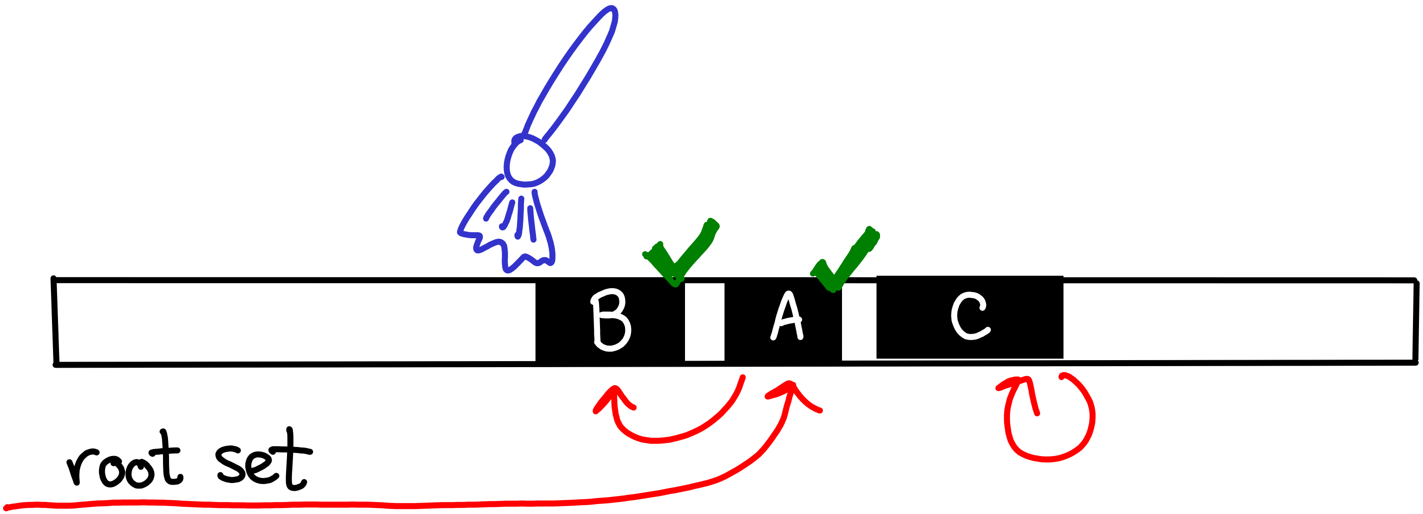
Mark and Sweep

Sweep memory for dead objects



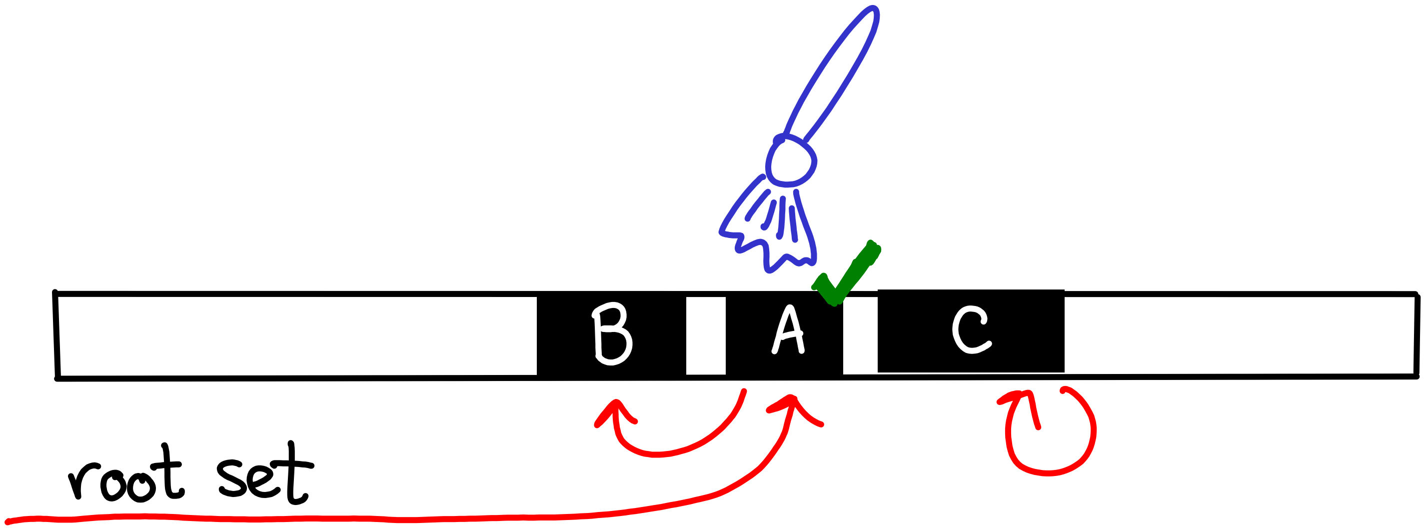
Mark and Sweep

Sweep memory for dead objects



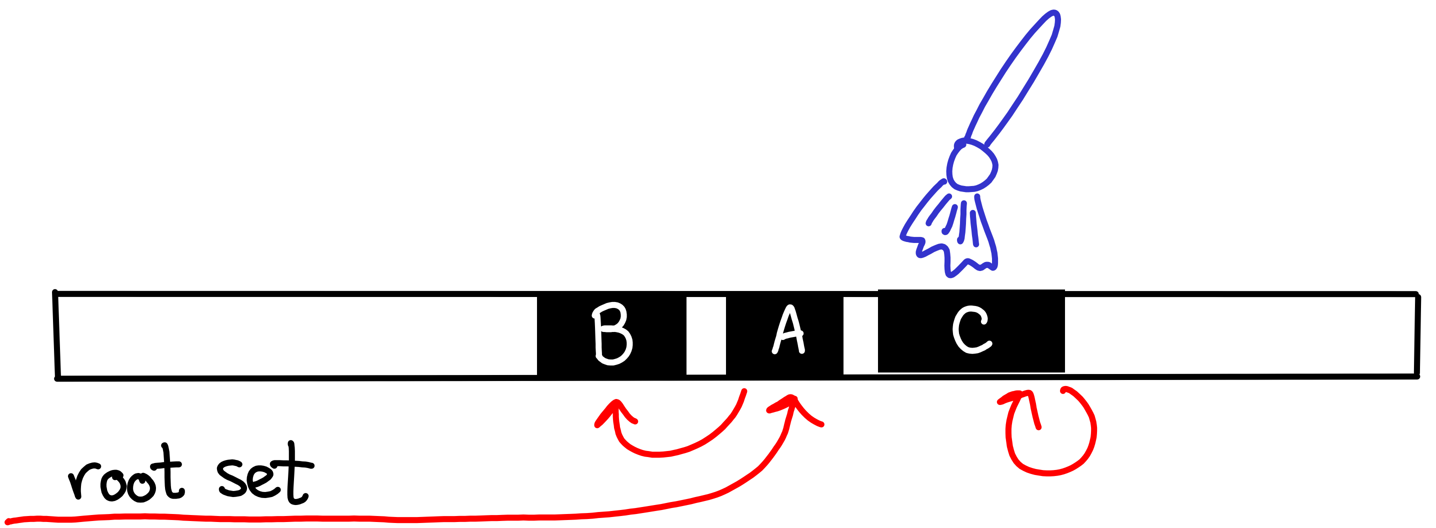
Mark and Sweep

Sweep memory for dead objects



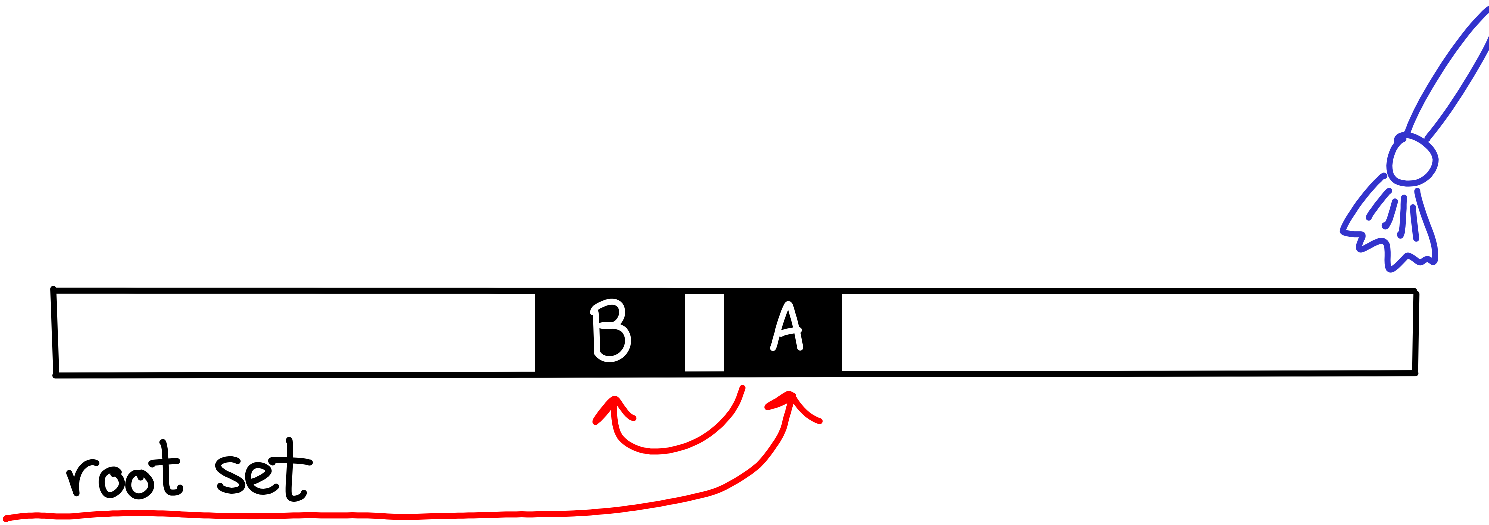
Mark and Sweep

Sweep memory for dead objects



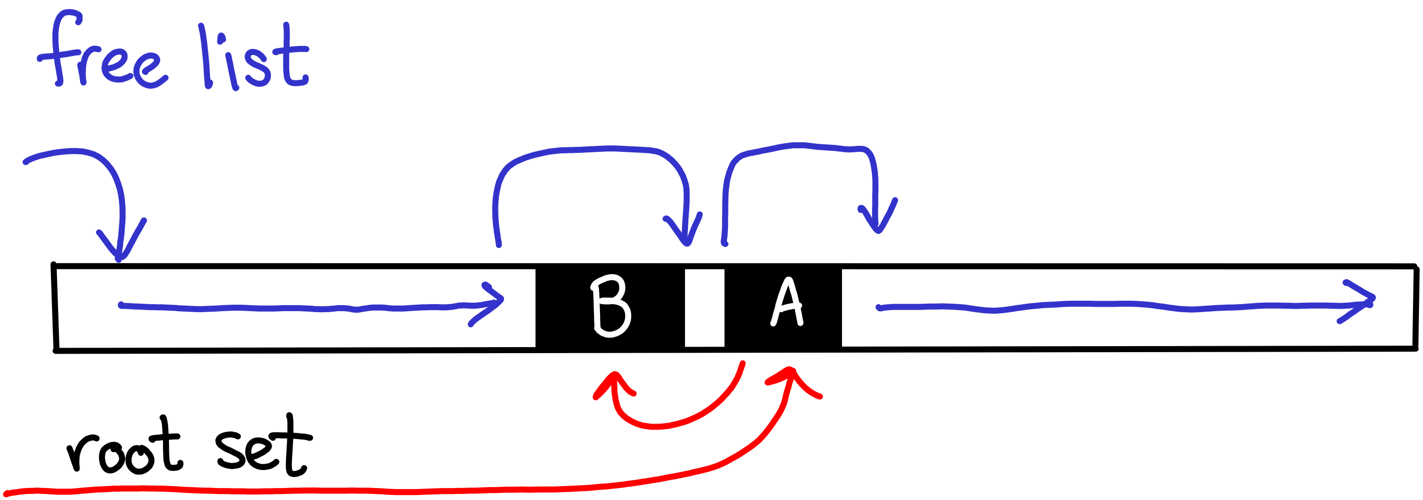
Mark and Sweep

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
- ⚠ Naively needs to traverse entire heap
- ⚠ Naively leads to fragmentation (can compact)
- ✗ Needs to store a mark bit
- ✗ Needs to maintain TODO list
- ✗ Stop-the-world GC (could refcounting pause?)

Baker's algorithm can be used to only traverse the LIVE data.

Mark and Sweep

Traverse object graph for live objects

Sweep memory for dead objects

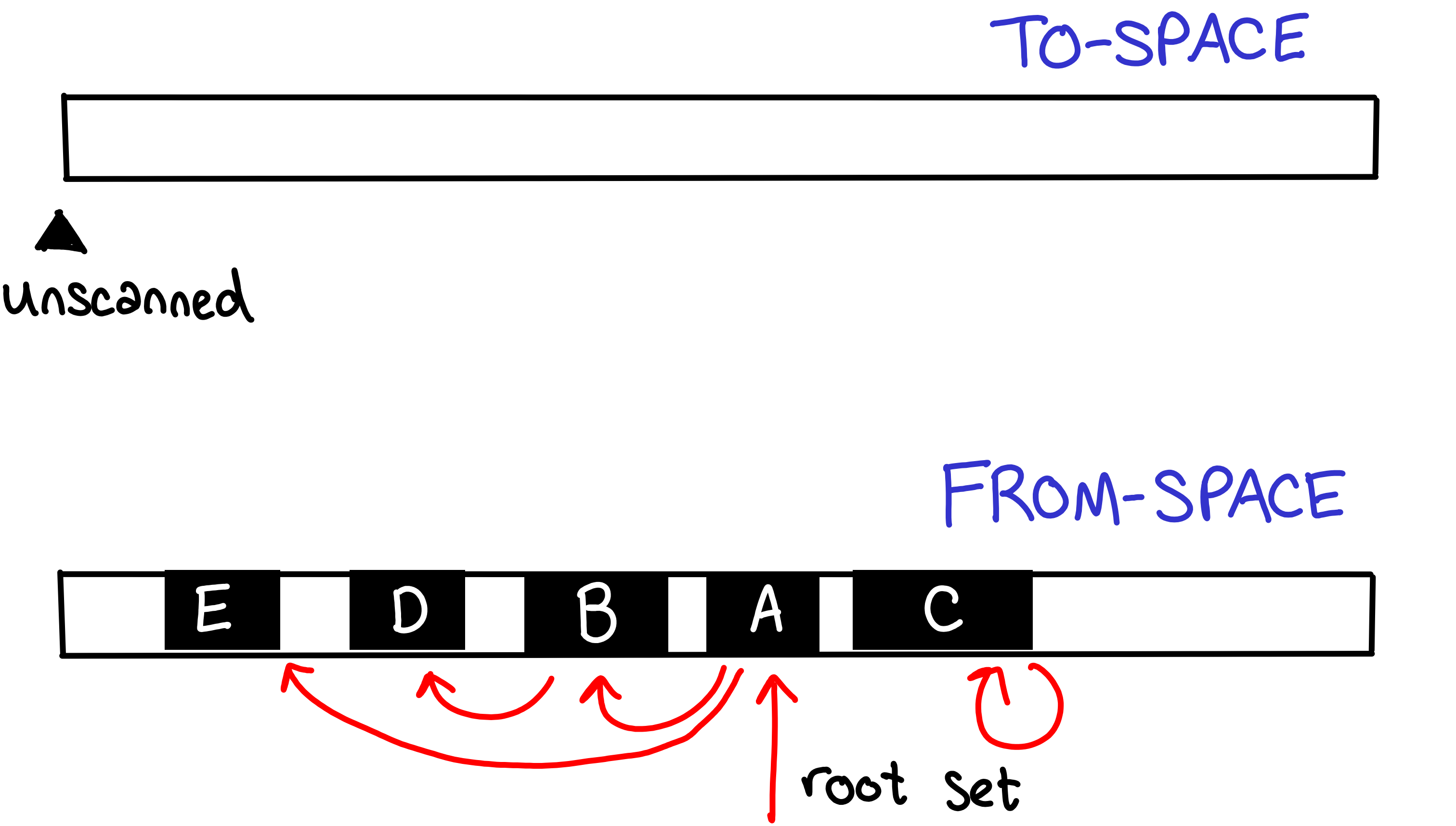
- ✓ Cycles are handled
- ✓ No extra bookkeeping
- ⚠ Naively needs to traverse entire heap
- ⚠ Naively leads to fragmentation (can compact)
- ✗ Needs to store a mark bit
- ✗ Needs to maintain TODO list

We can fix these problems, but avoiding stop-the-world is quite difficult (a research problem, even.)

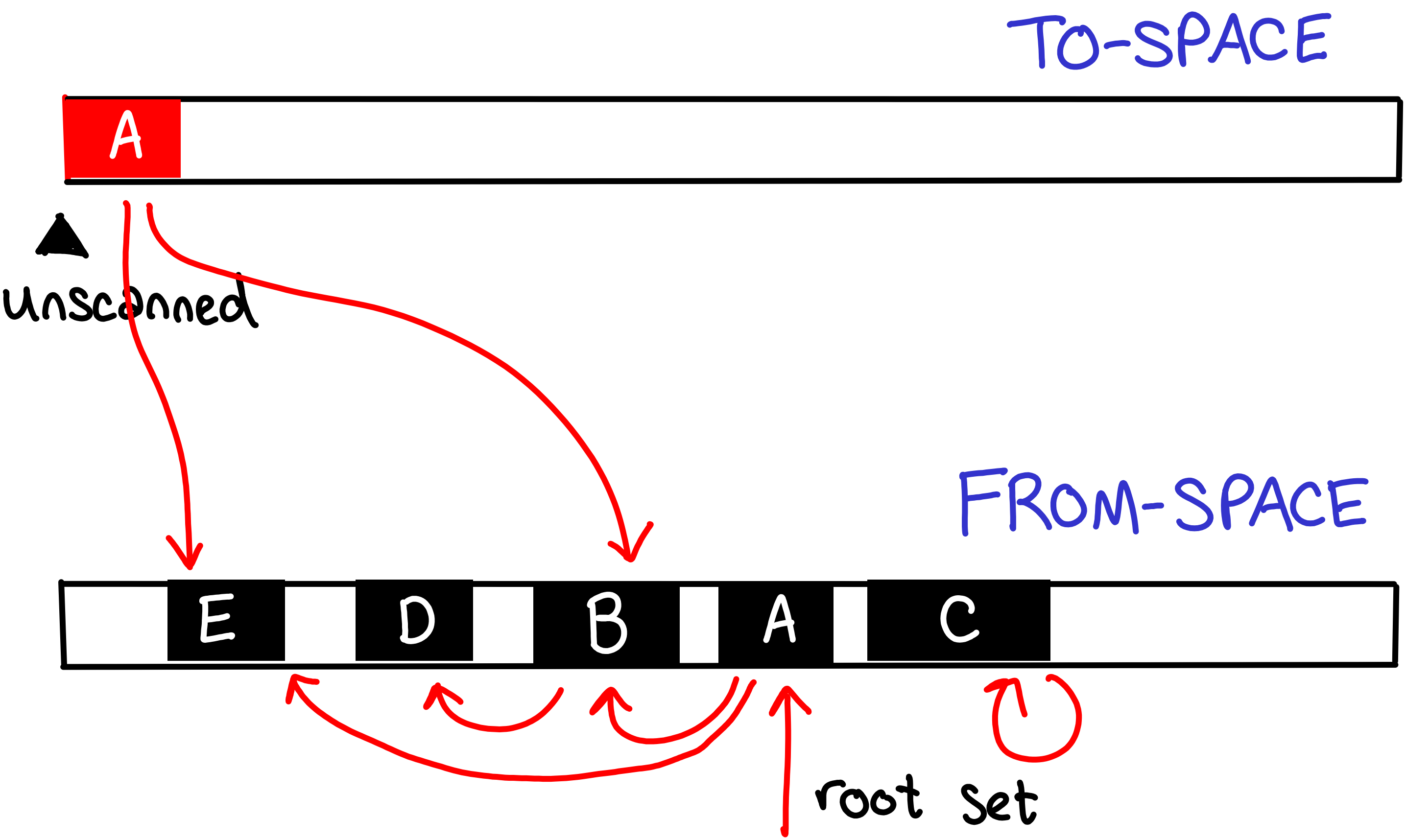
- ✗ Stop-the-world GC

(could refcounting pause?)

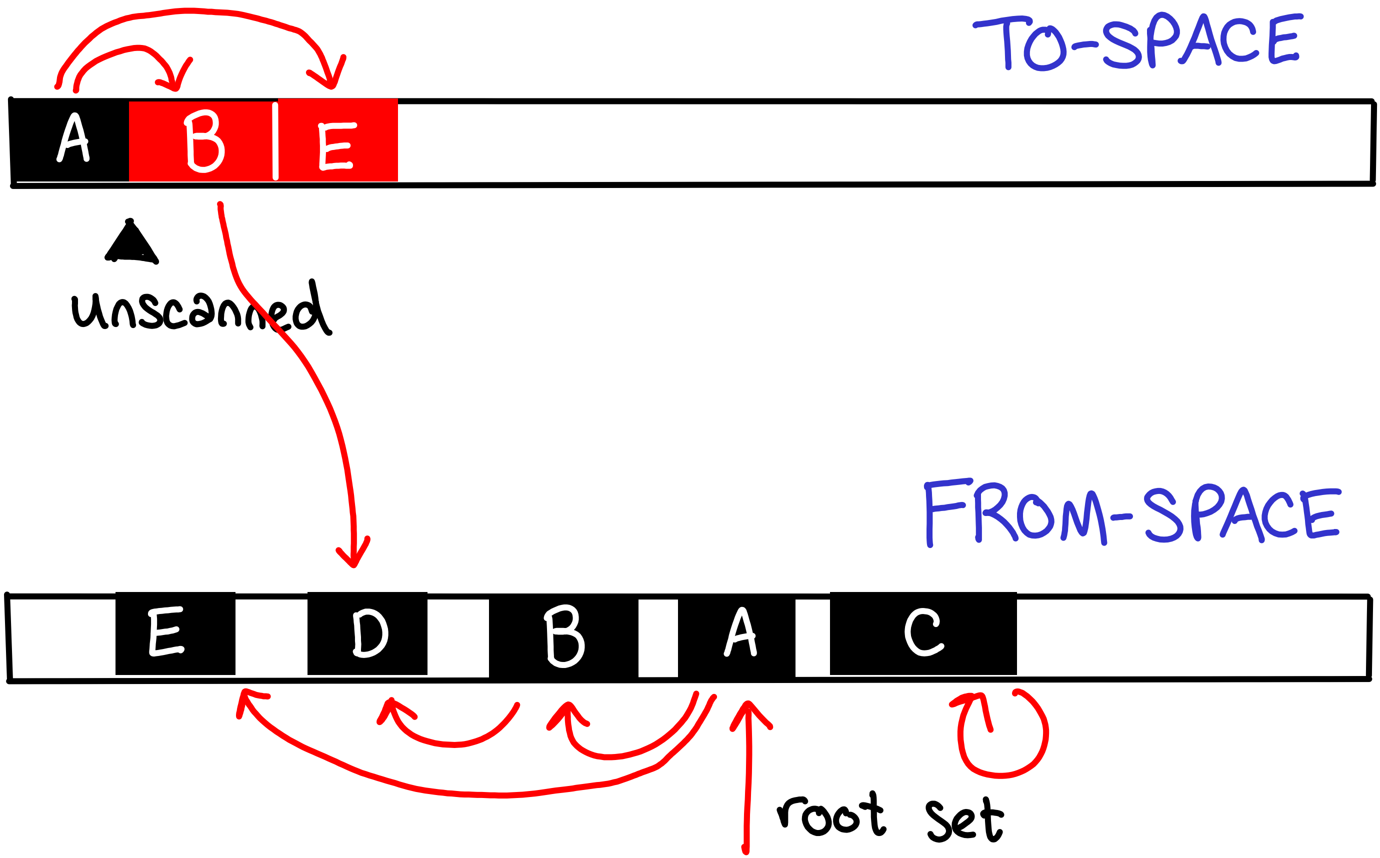
Copying Collection



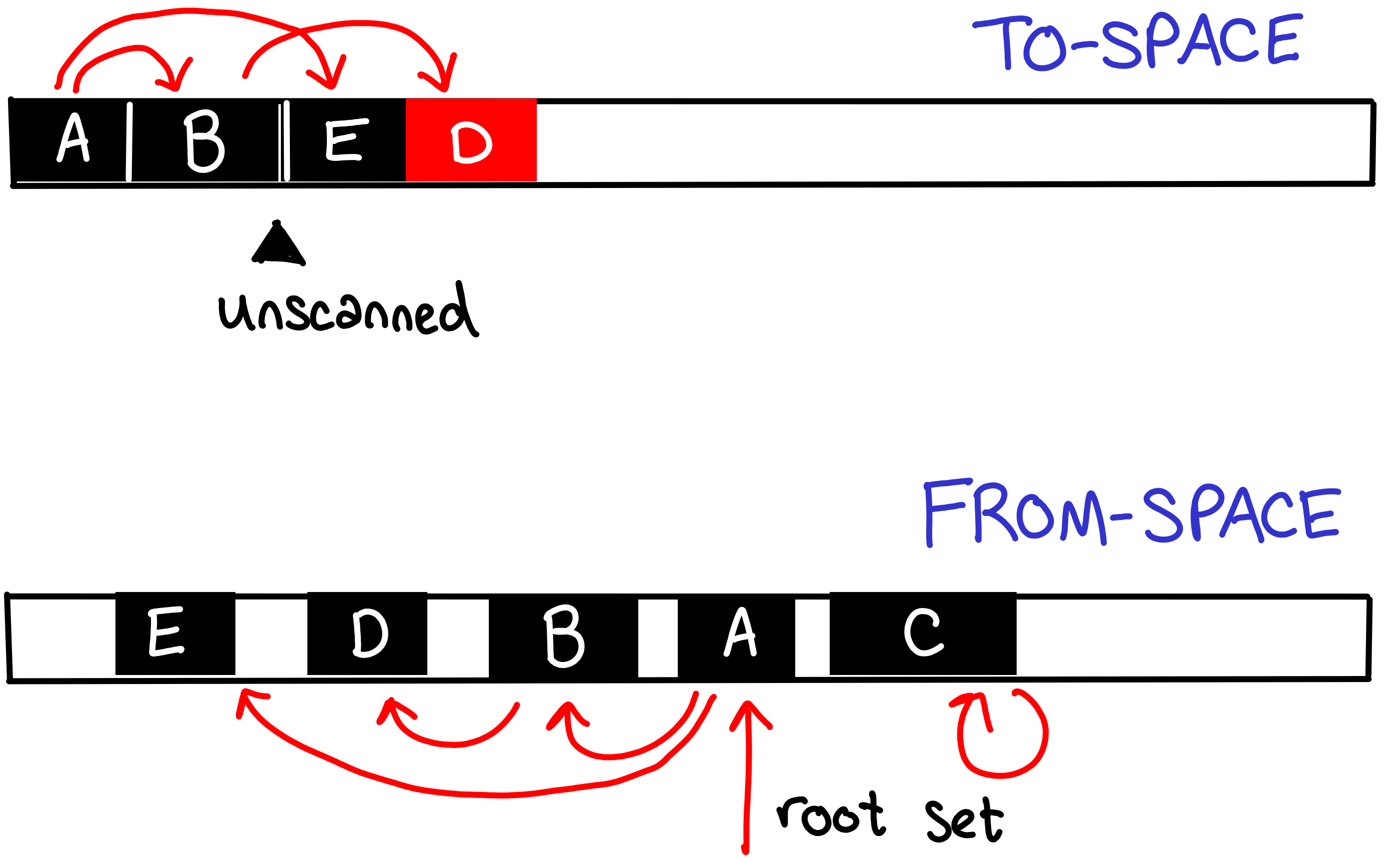
Copying Collection



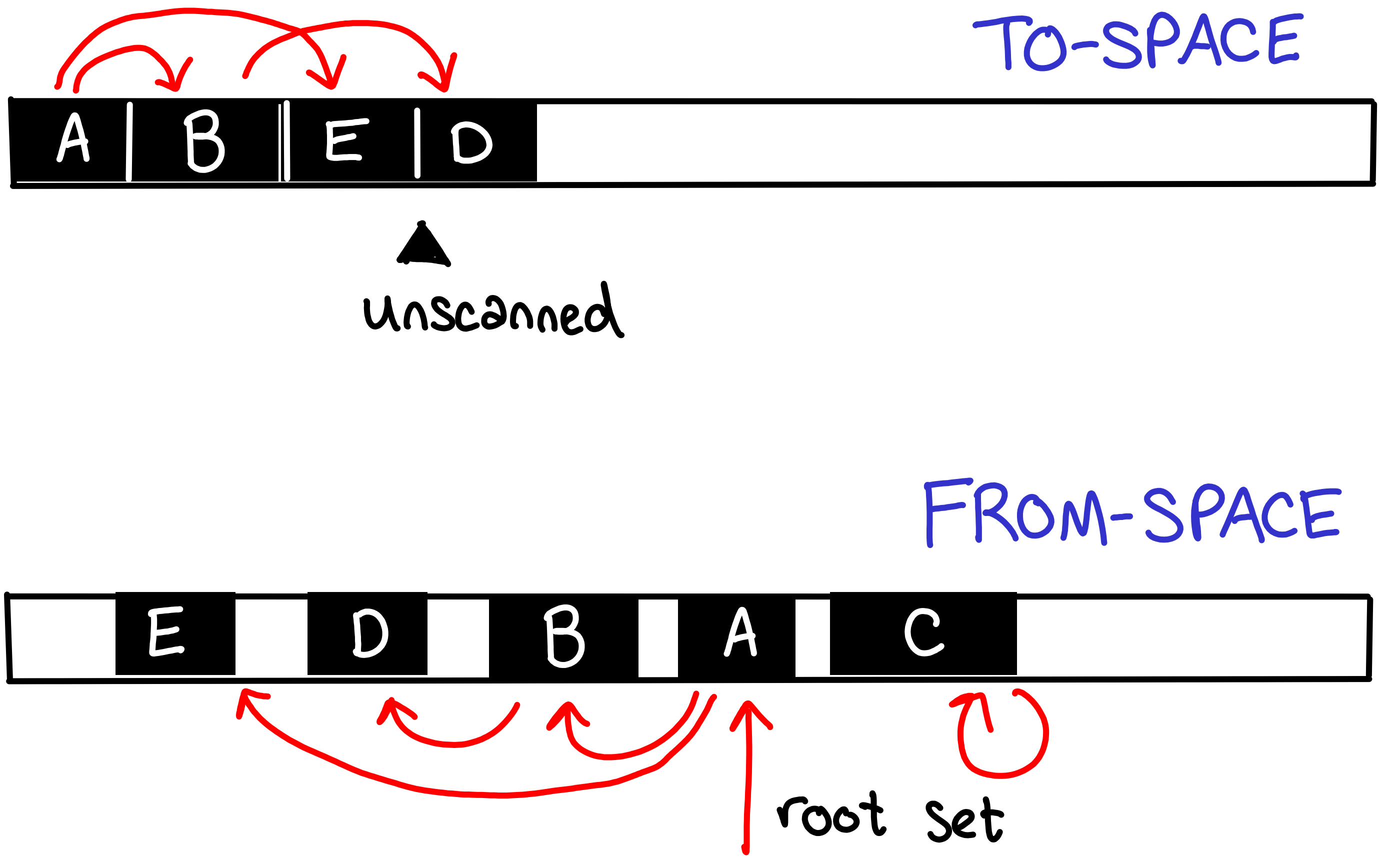
Copying Collection



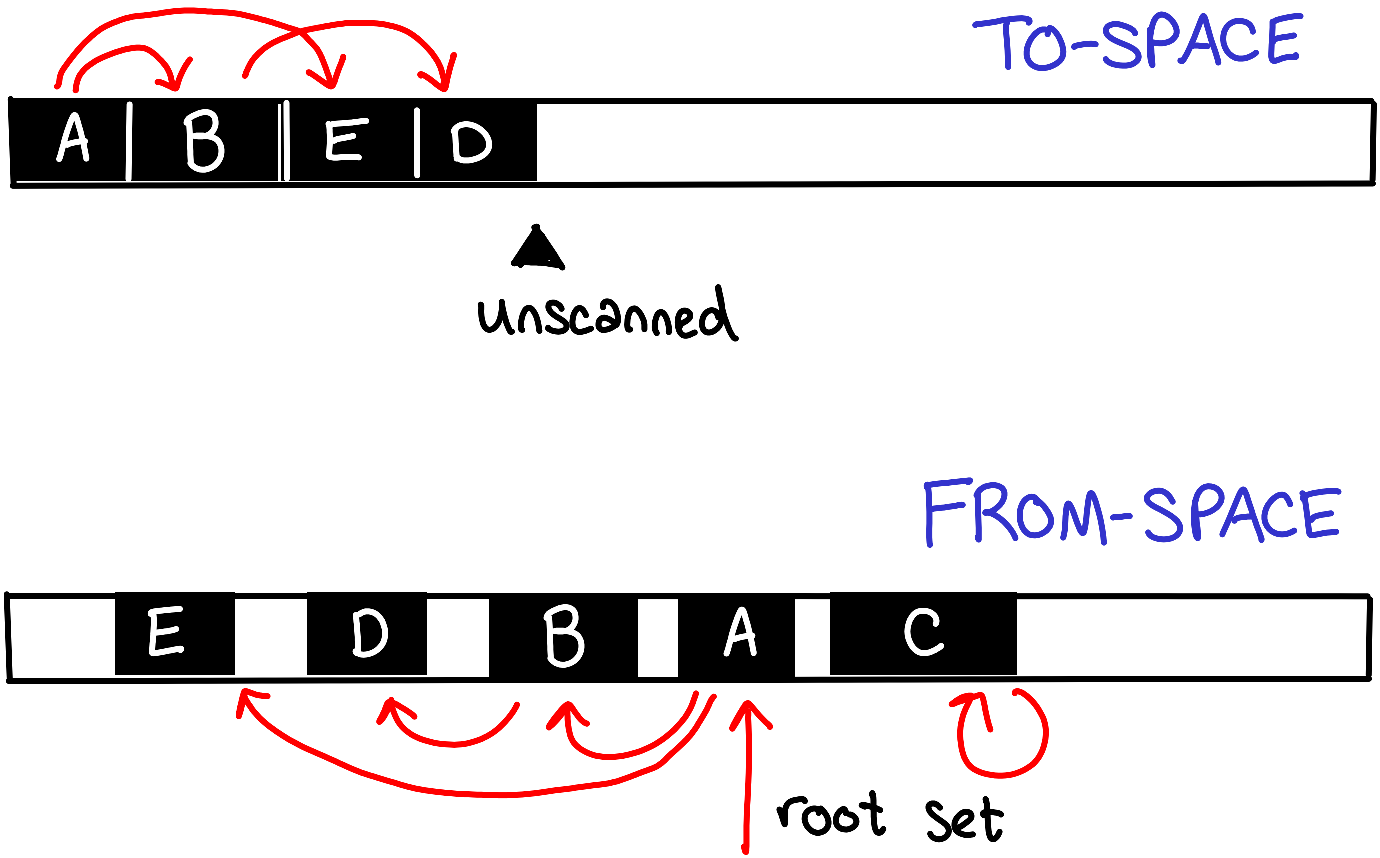
Copying Collection



Copying Collection



Copying Collection



Copying Collection

- ✓ Compacts data (better locality)
- ✓ Constant space bookkeeping
- ✗ Needs x2 available space
- ✗ (Still) Stop-the-world GC

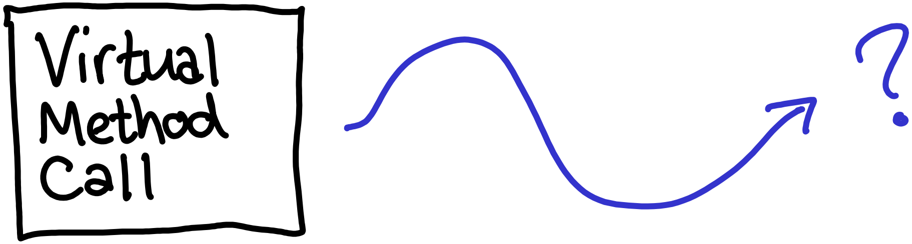
Summary: Garbage Collection

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

Why is generational garbage collection difficult? Mutation!
The assumption is that you don't need to scan old generations
when you are collecting younger ones, because old objects
can't point to young ones. This doesn't hold if you have mutation.

Dynamic Dispatch

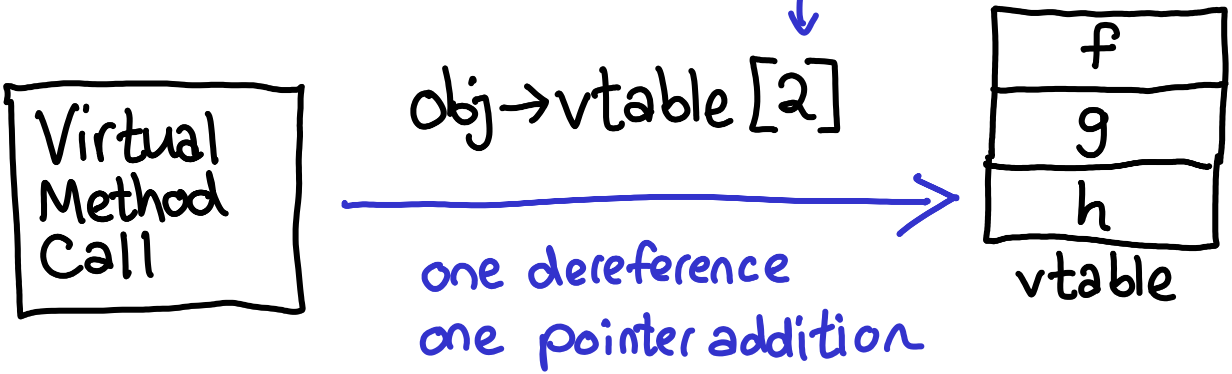
Recap: C++ Virtual Tables



Remember the control flow lectures; how do we know where to go when we make a virtual method call?

Recap: C++ Virtual Tables

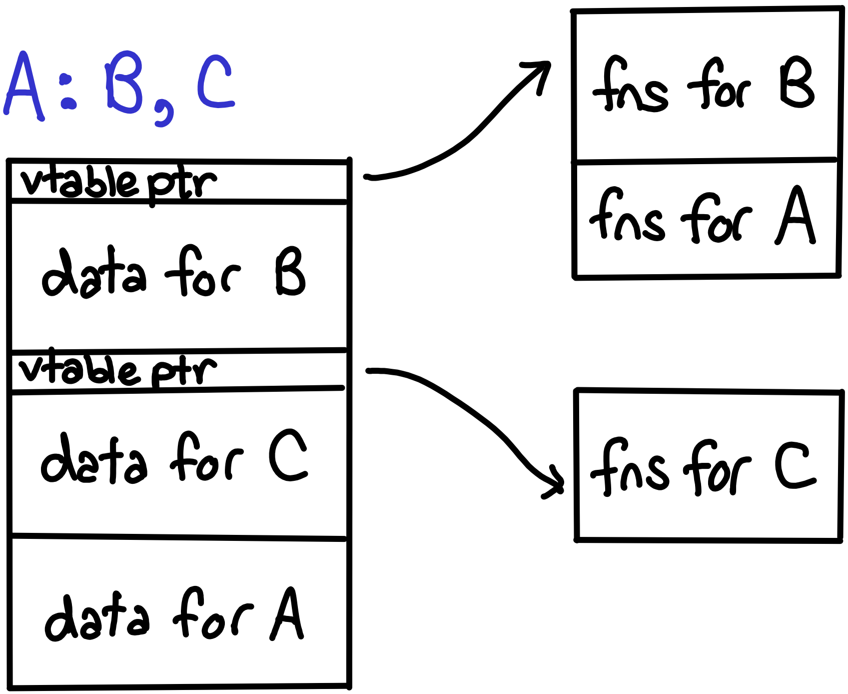
The answer was, you looked up a function pointer in the vtable, and jumped to that location. And recall in our discussion about vtables, the entire process was optimized to ensure that this function call could be done as quickly as possible.



C++ goal: Make virtual dispatch <u>as efficient as possible</u>

Recap: C++ Virtual Tables

class A: B, C



And there was even a fancy scheme for dealing with multiple inheritance by "mimicking" the expected layout at every possible subtype for the object.

C++ doesn't have interfaces!

Consequence: Multiple inheritance, but no interfaces

Recap: C++ Virtual Tables

We could say the motivating problem is how you can quickly call a virtual function, even though you don't know WHERE it might be stored in a class. Naively, you have to do some dictionary lookup.

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

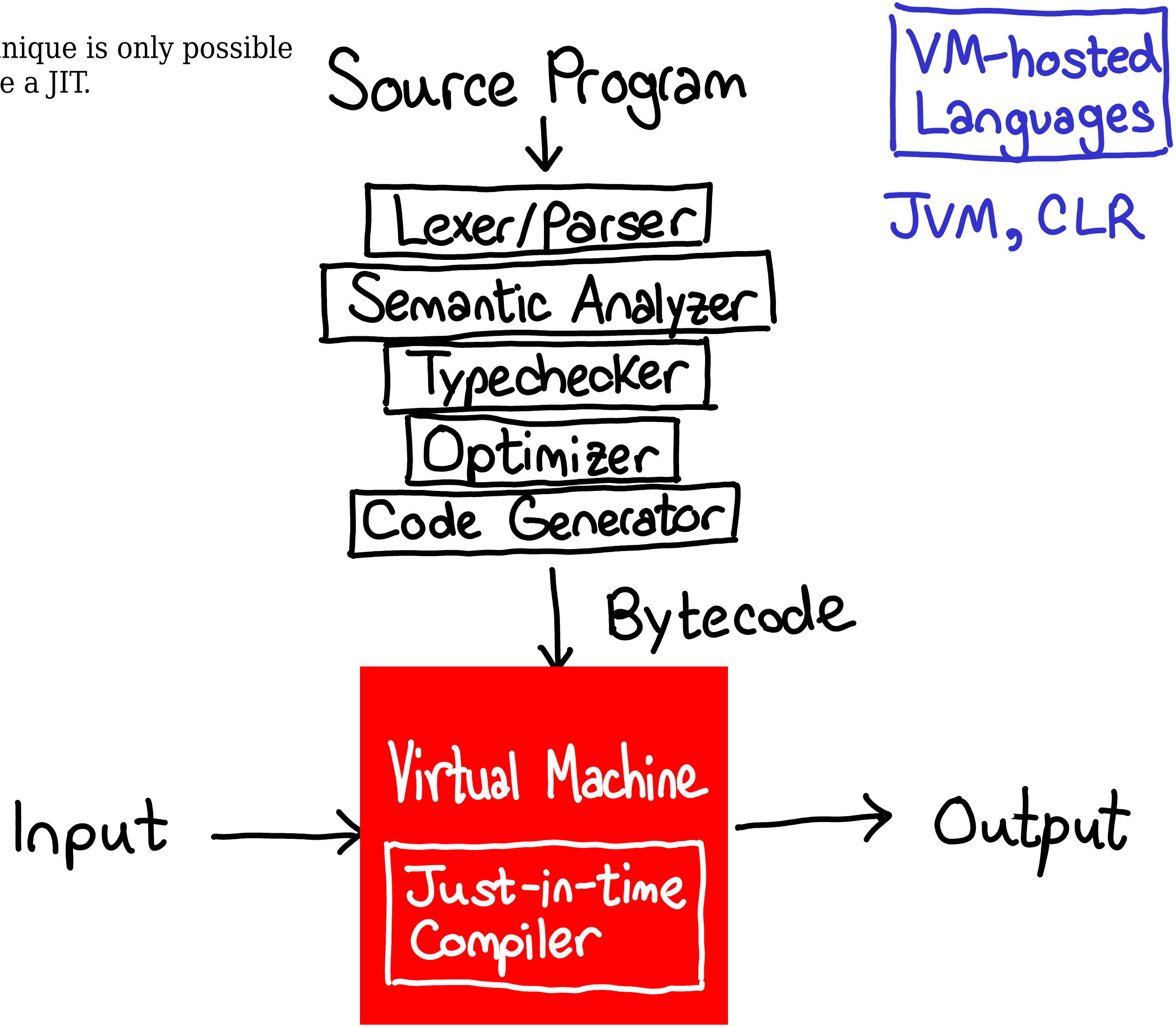
Motivating Problem

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

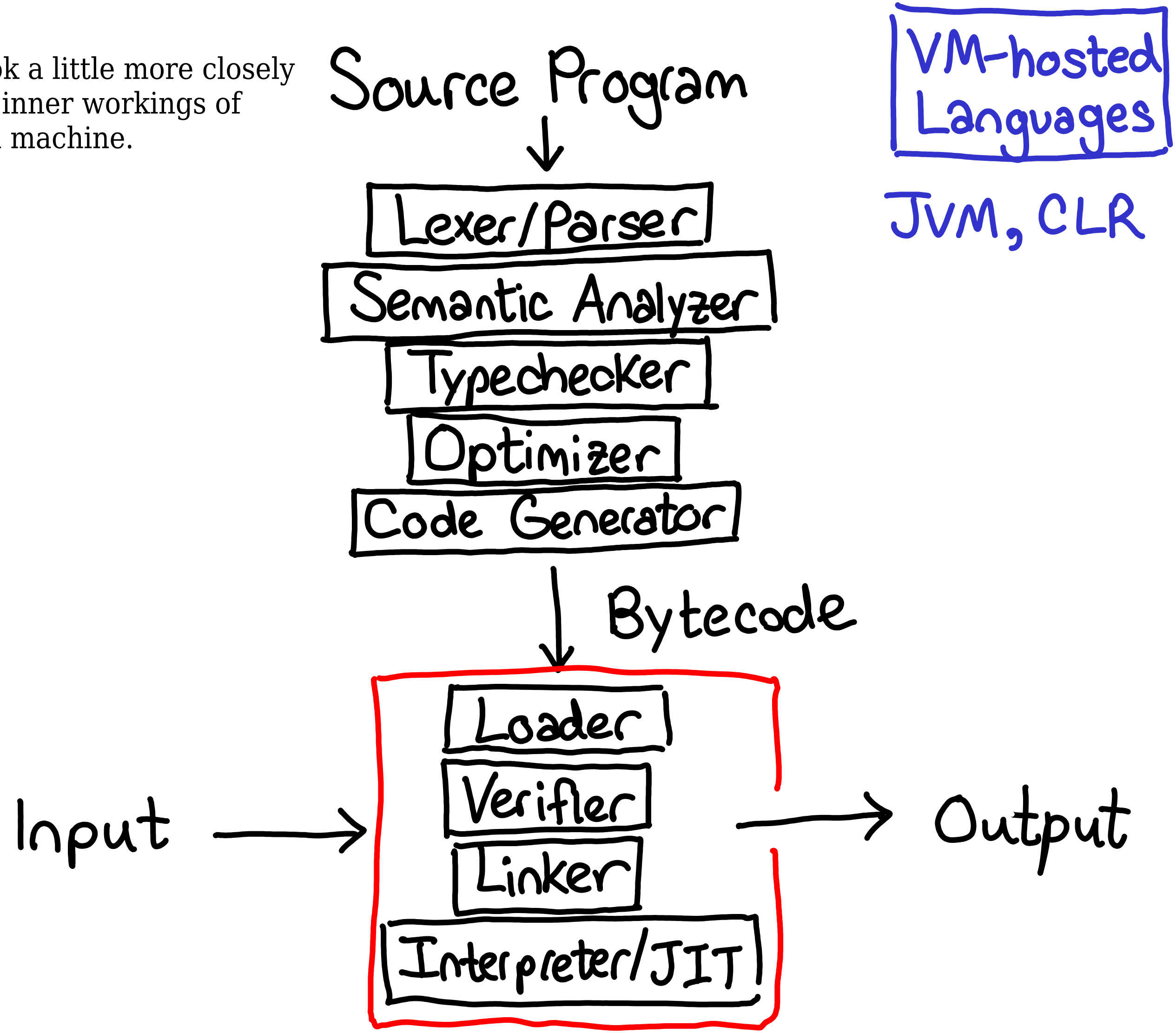
```
class B {
    virtual void g();
    virtual void f();
}
```

Today: Do a dictionary lookup and cache it

This technique is only possible if we have a JIT.



Let's look a little more closely into the inner workings of a virtual machine.



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

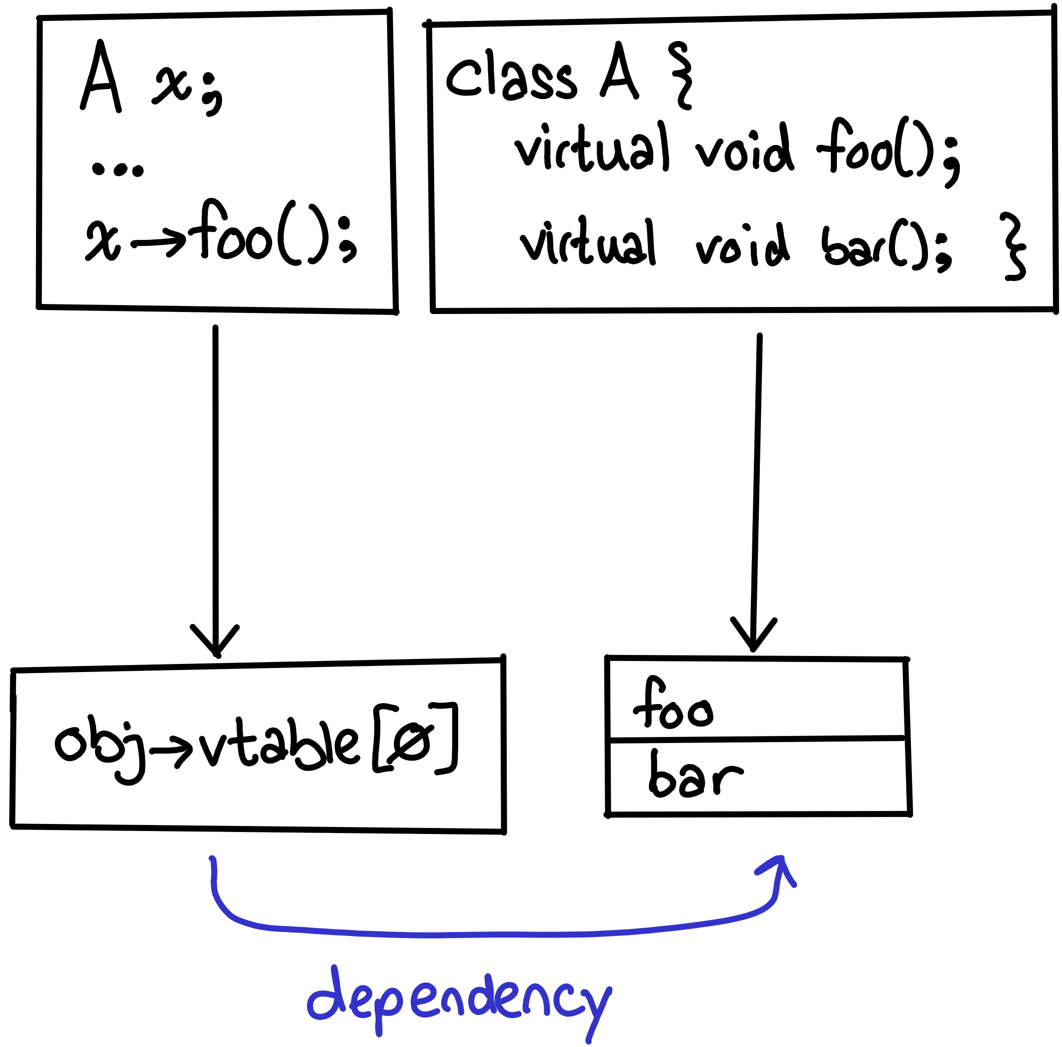
3. invokedynamic — polymorphic inline caches

(or Smalltalk or Self)

invokevirtual

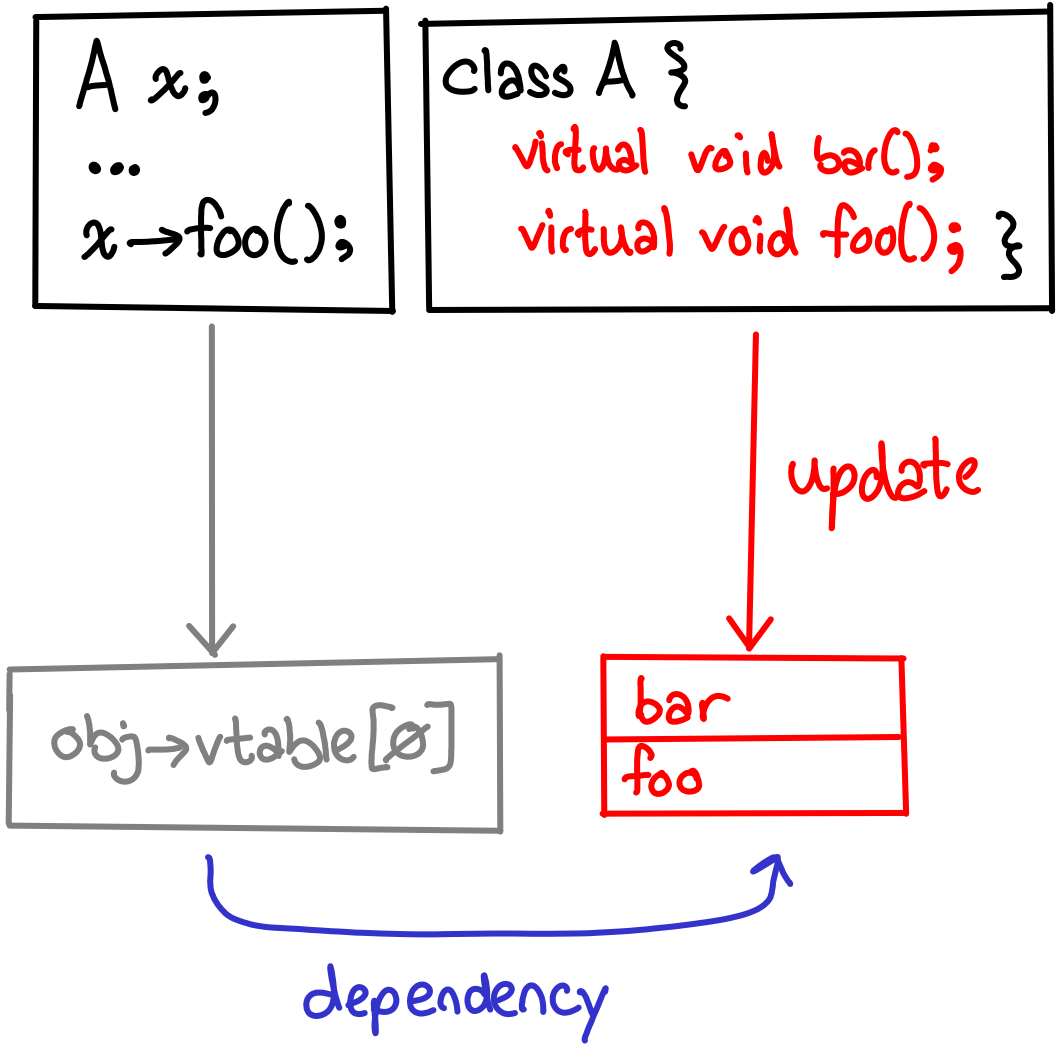
```
A x;
...
x->foo();
```

invokevirtual

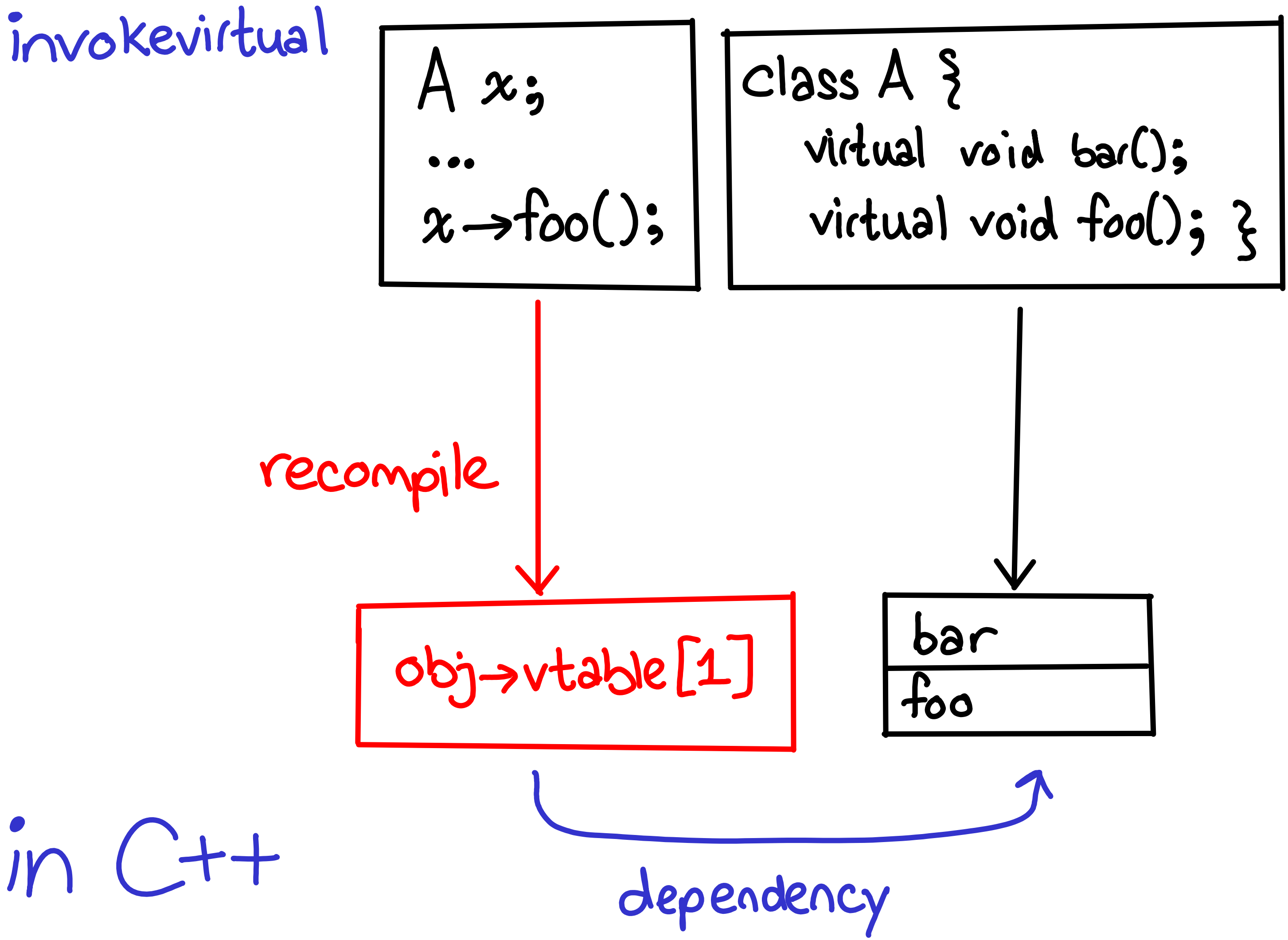


in C++

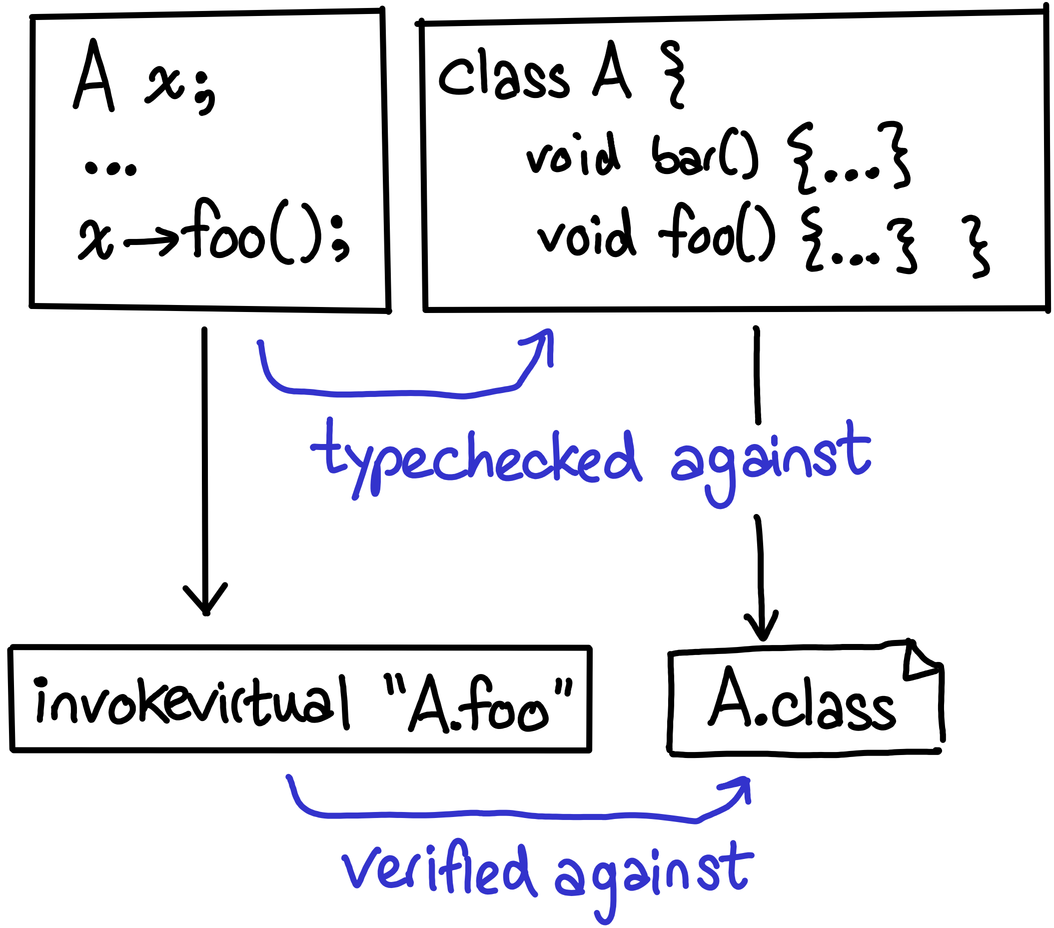
invokevirtual



in C++

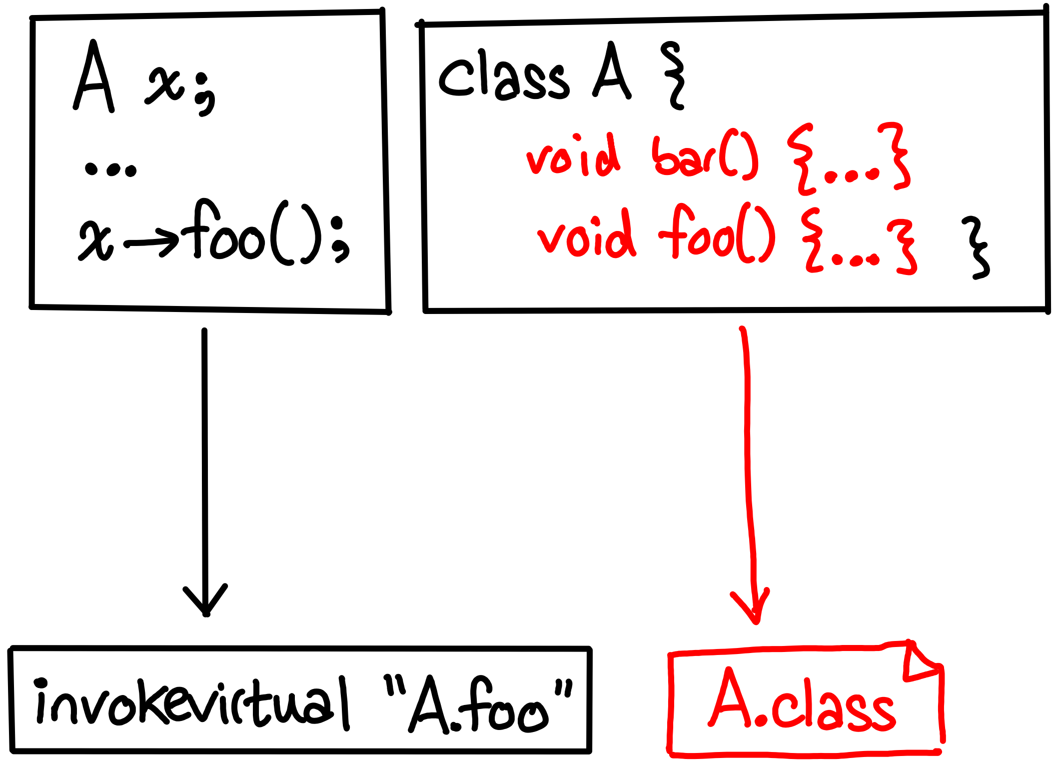

invokevirtual
A x;
...
x→foo();
class A {
virtual void bar();
virtual void foo(); }
recompile
obj→vtable[1]
bar
foo
in C++
dependency

invokevirtual



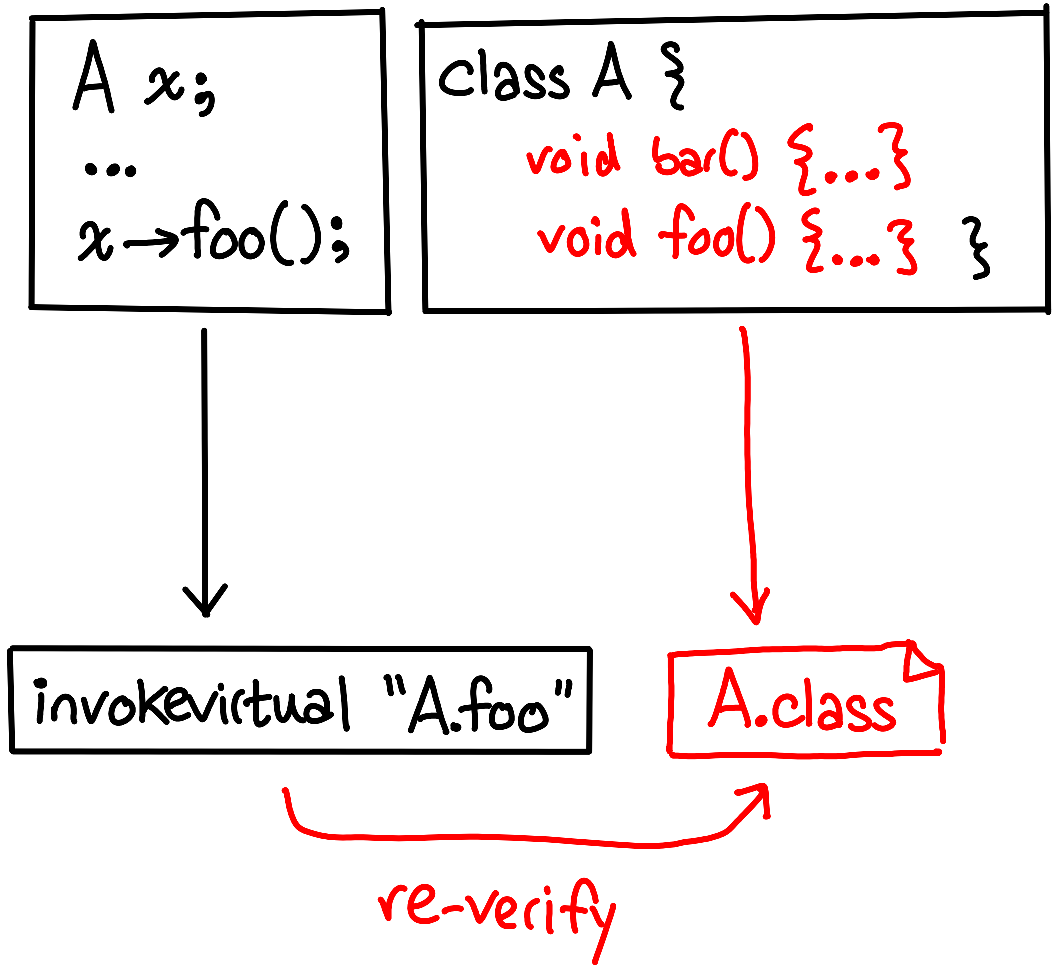
in Java

invokevirtual



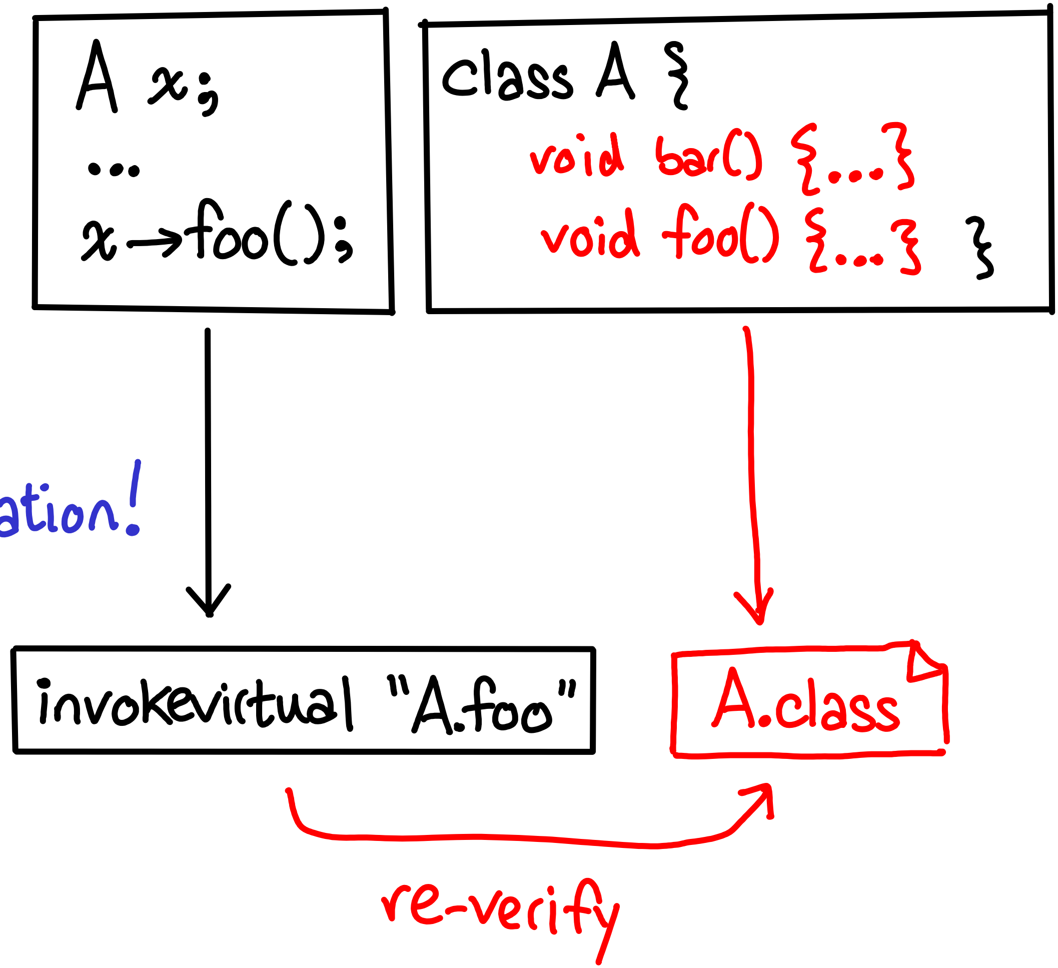
in Java

invokevirtual



in Java

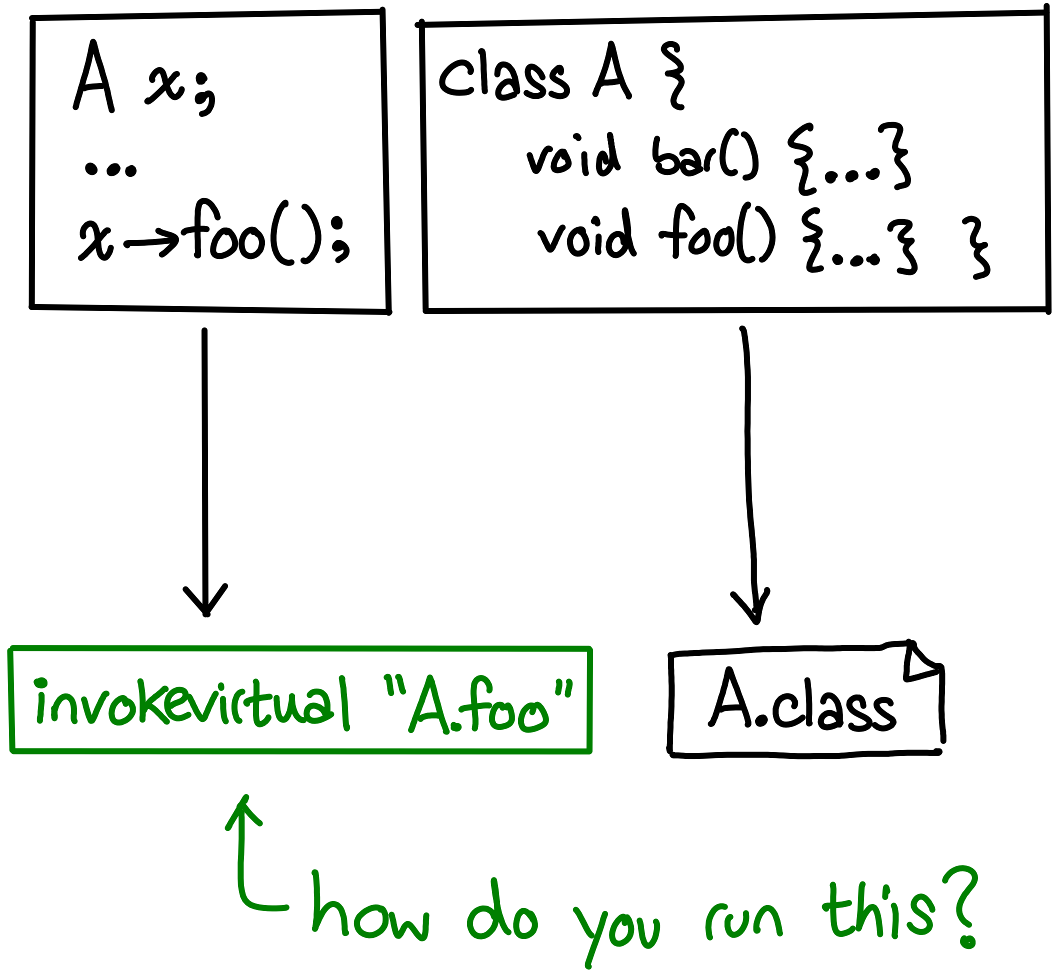
invokevirtual



but no recompilation!

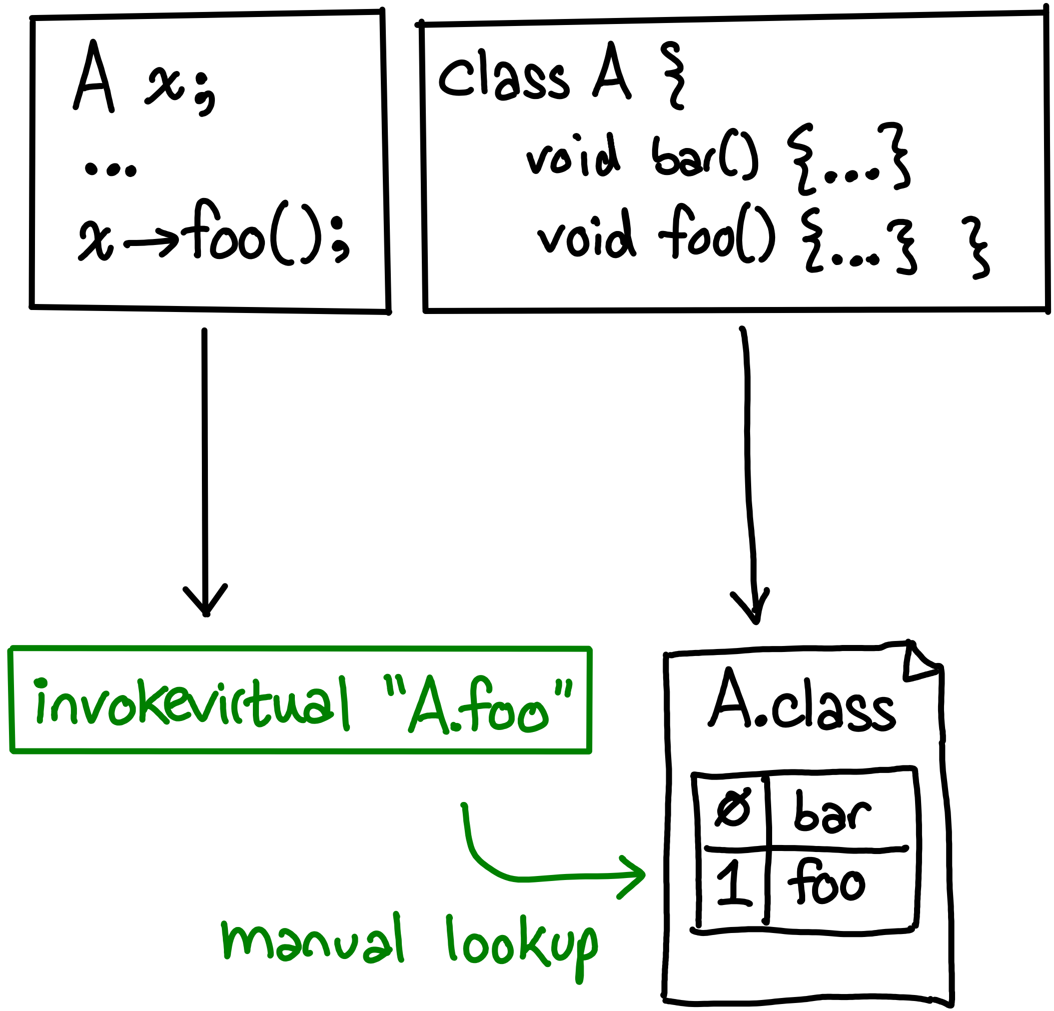
in Java

invokevirtual



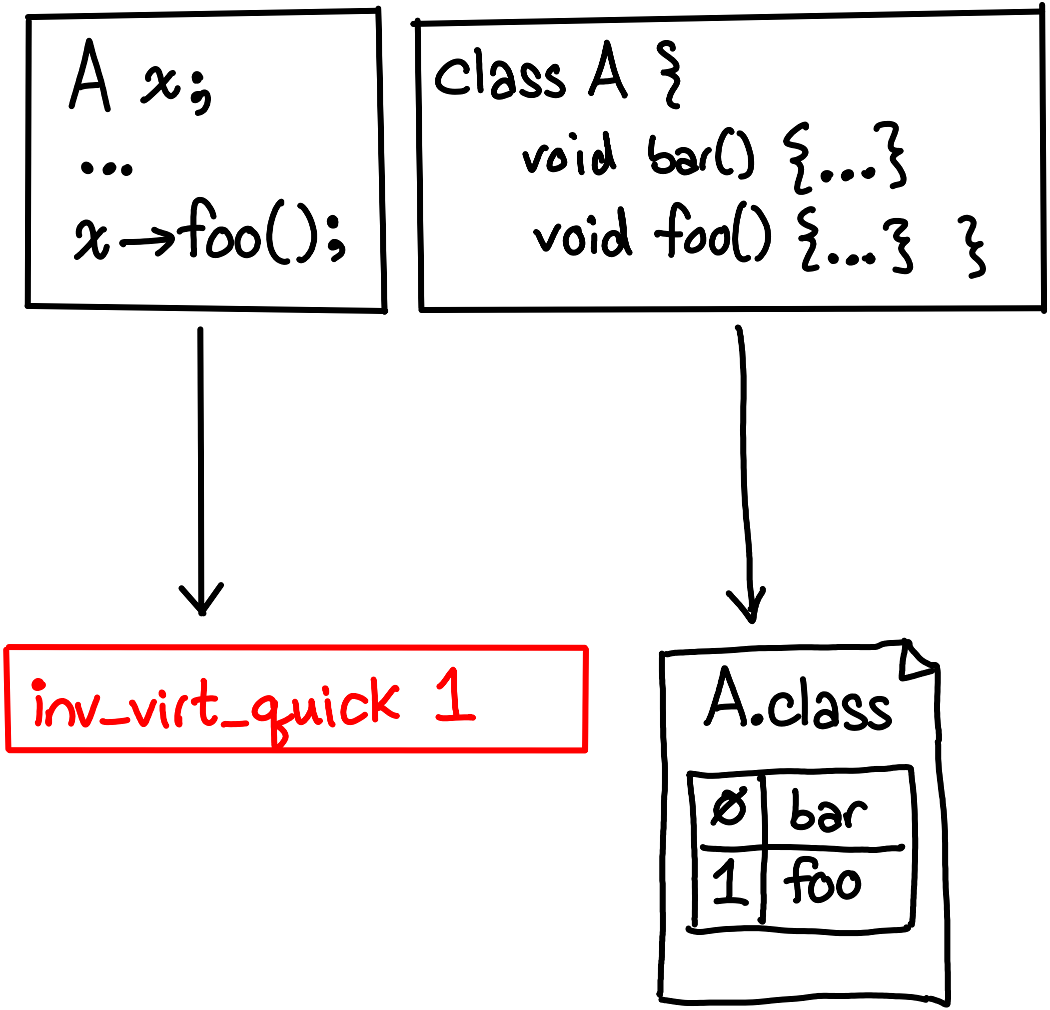
in Java

invokevirtual



in Java

invokevirtual



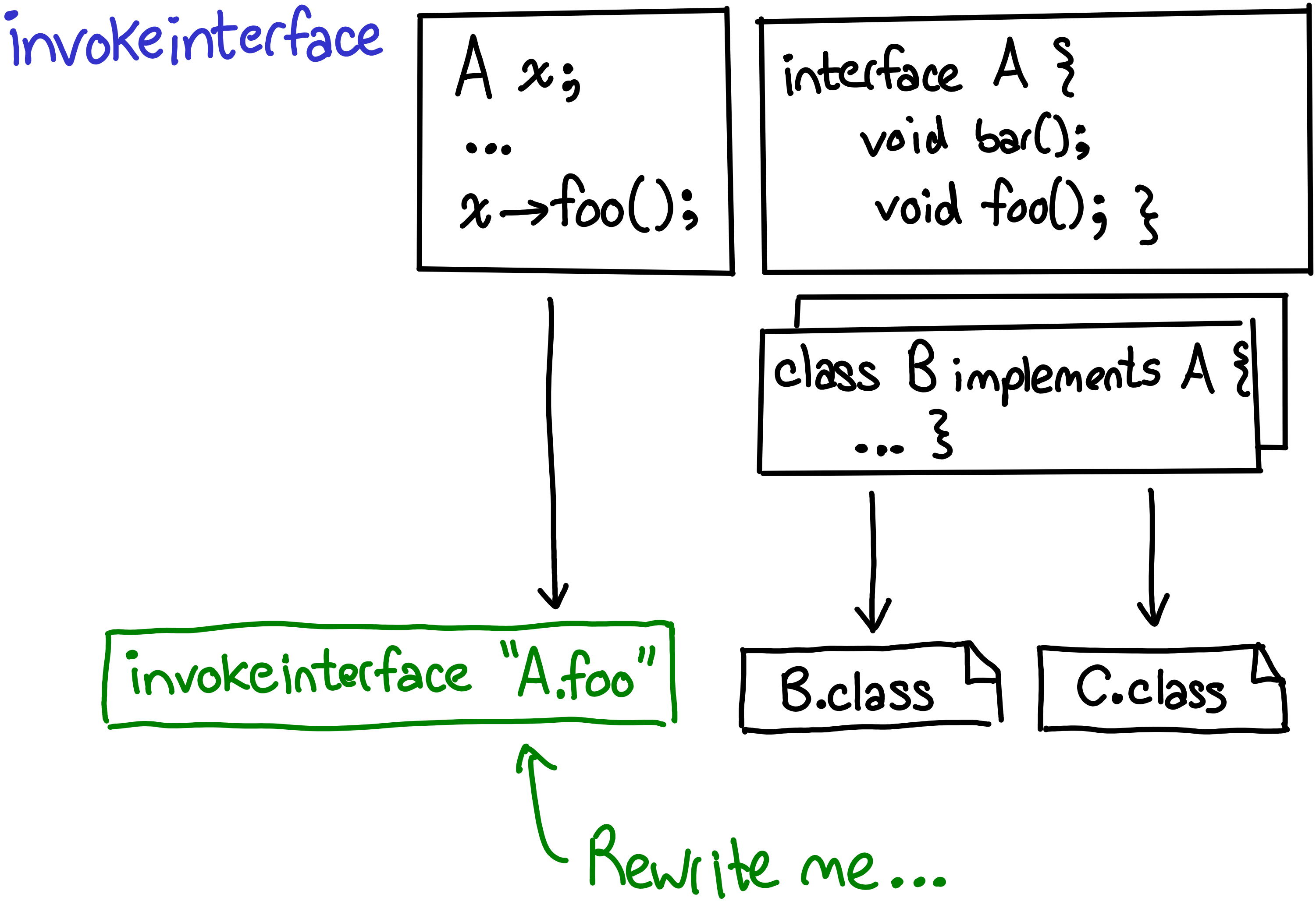
in Java

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

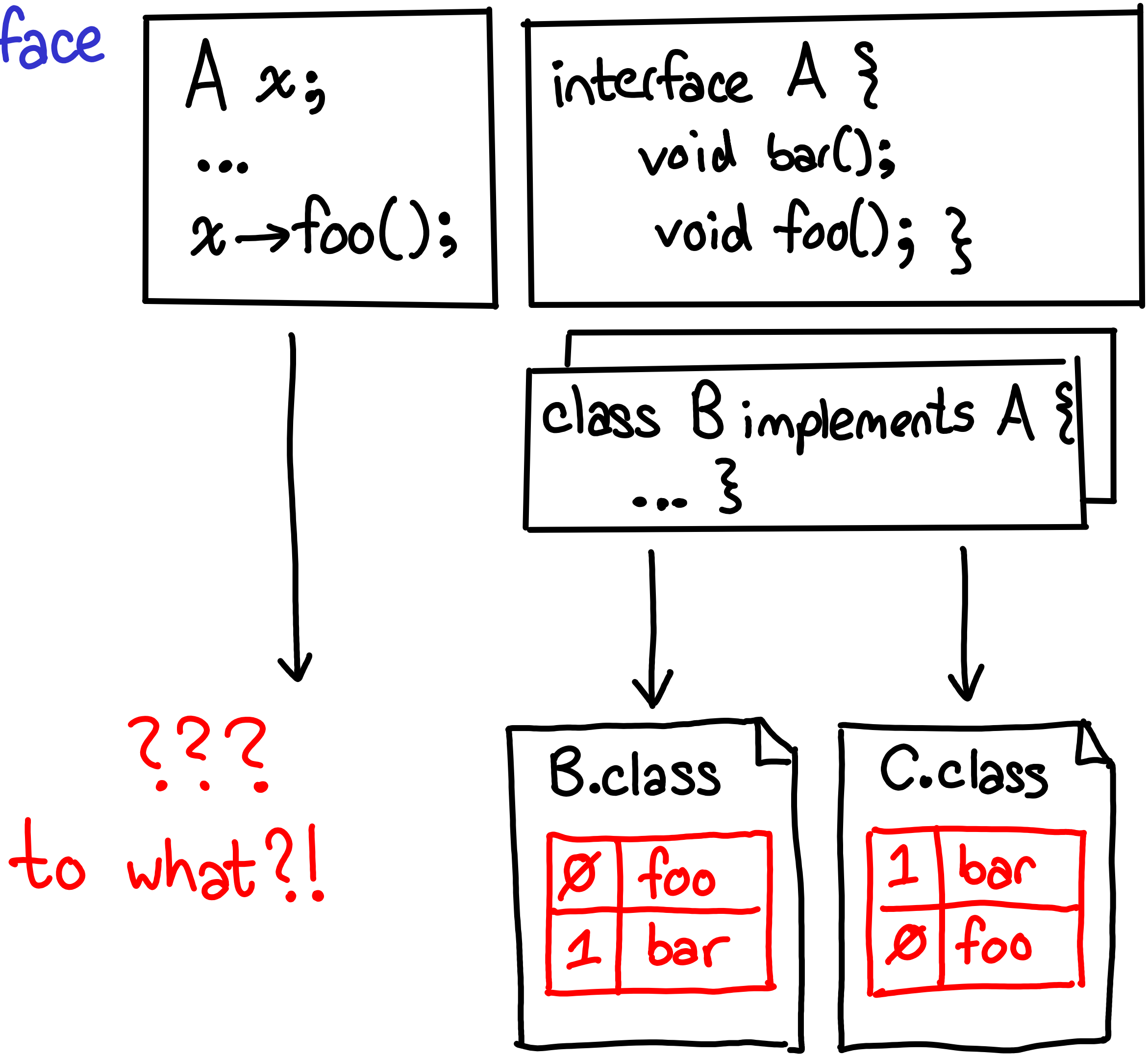
invokevirtual "A.foo" → inv_virt_quick 1

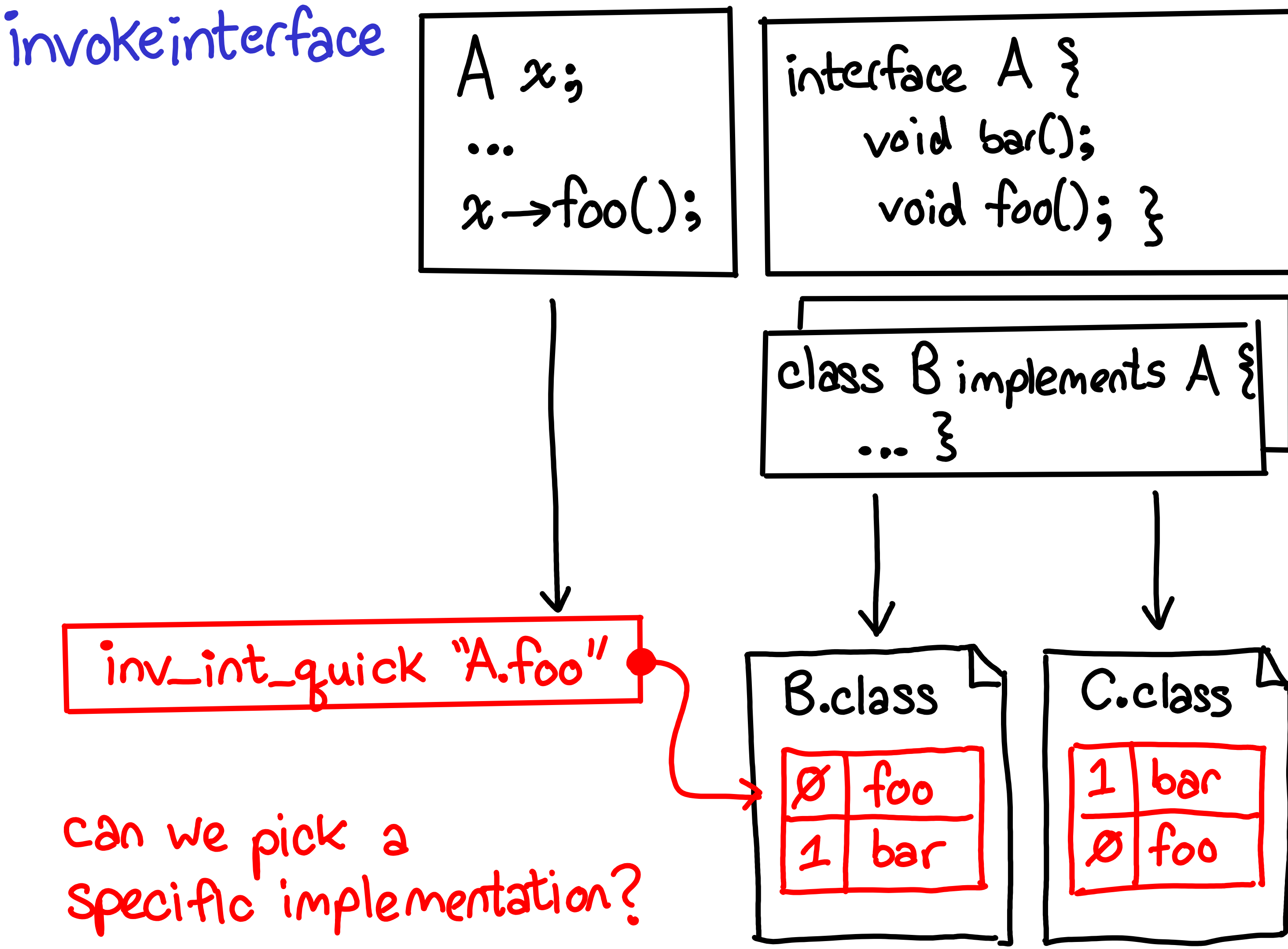
fast, C++-like machine code

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




invokeinterface
A x;
...
x→foo();
interface A {
void bar();
void foo(); }
class B implements A {
... }
inv_int_quick "A.foo"
B.class
Ø foo
1 bar
C.class
1 bar
Ø foo
can we pick a
specific implementation?

invokeinterface

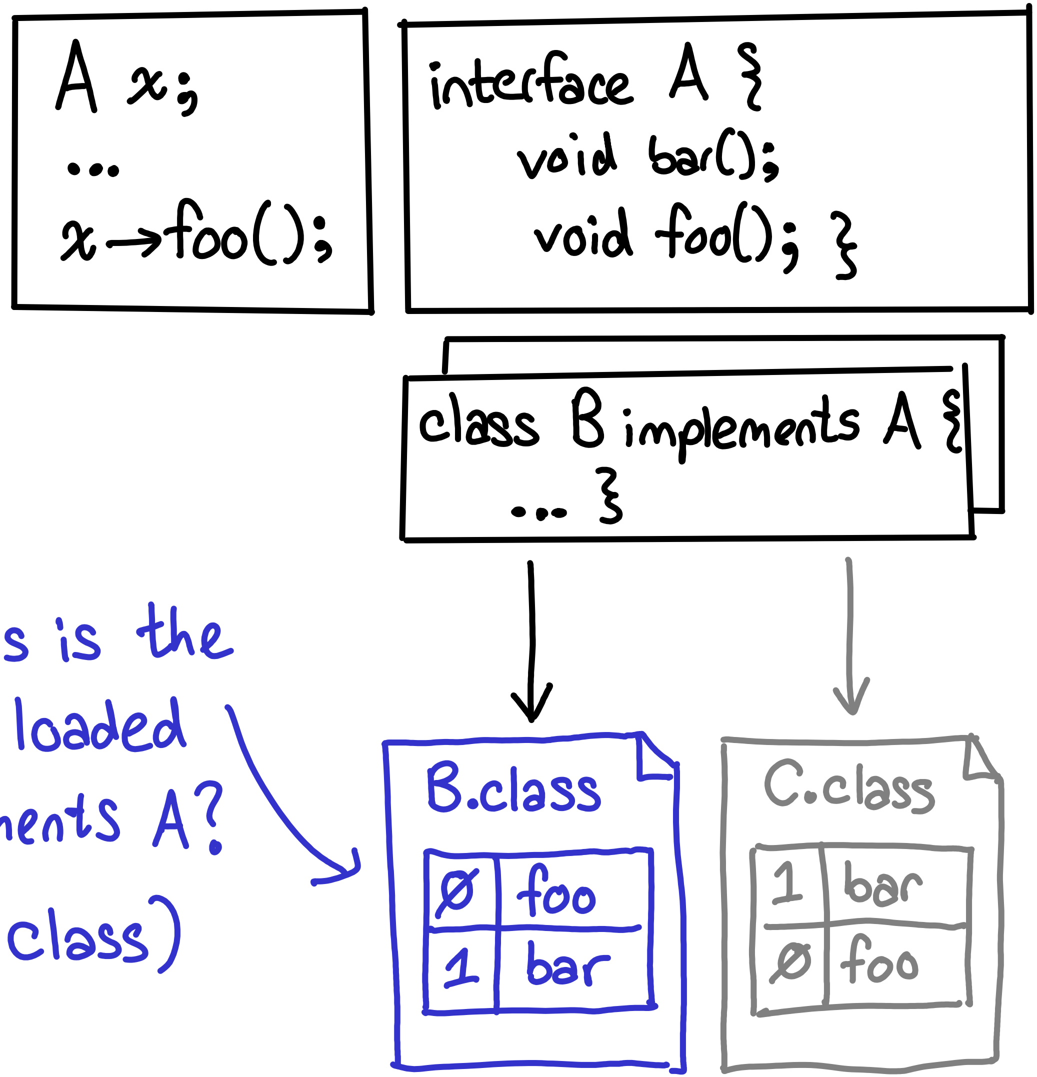
cache

```
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 <u>inline cache</u>

invokeinterface



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 <u>always</u> be B,
<u>omit conditional</u>

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

originated 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)