

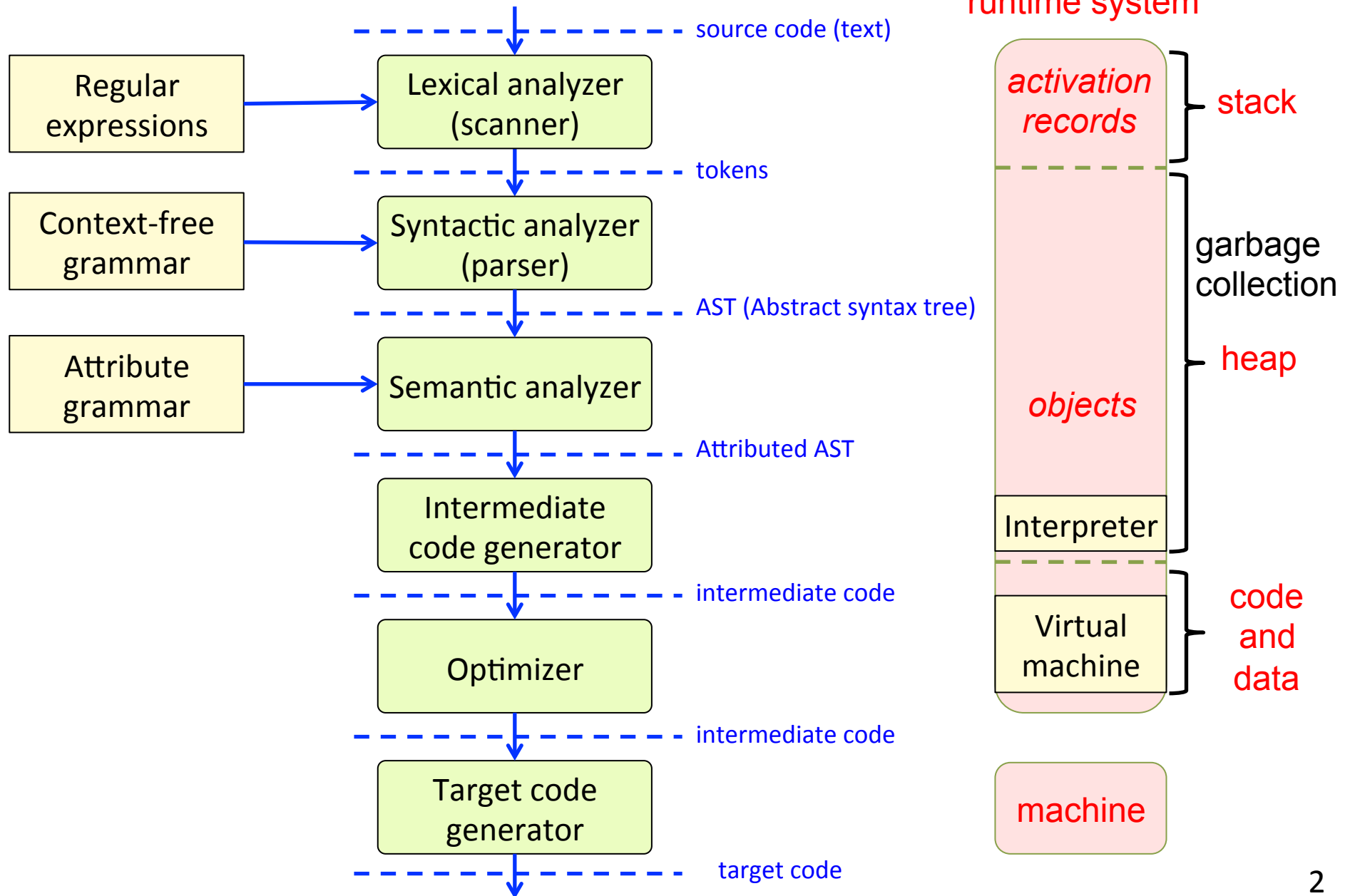
EDAN65: Compilers, Lecture 13

# Runtime systems for object-oriented languages

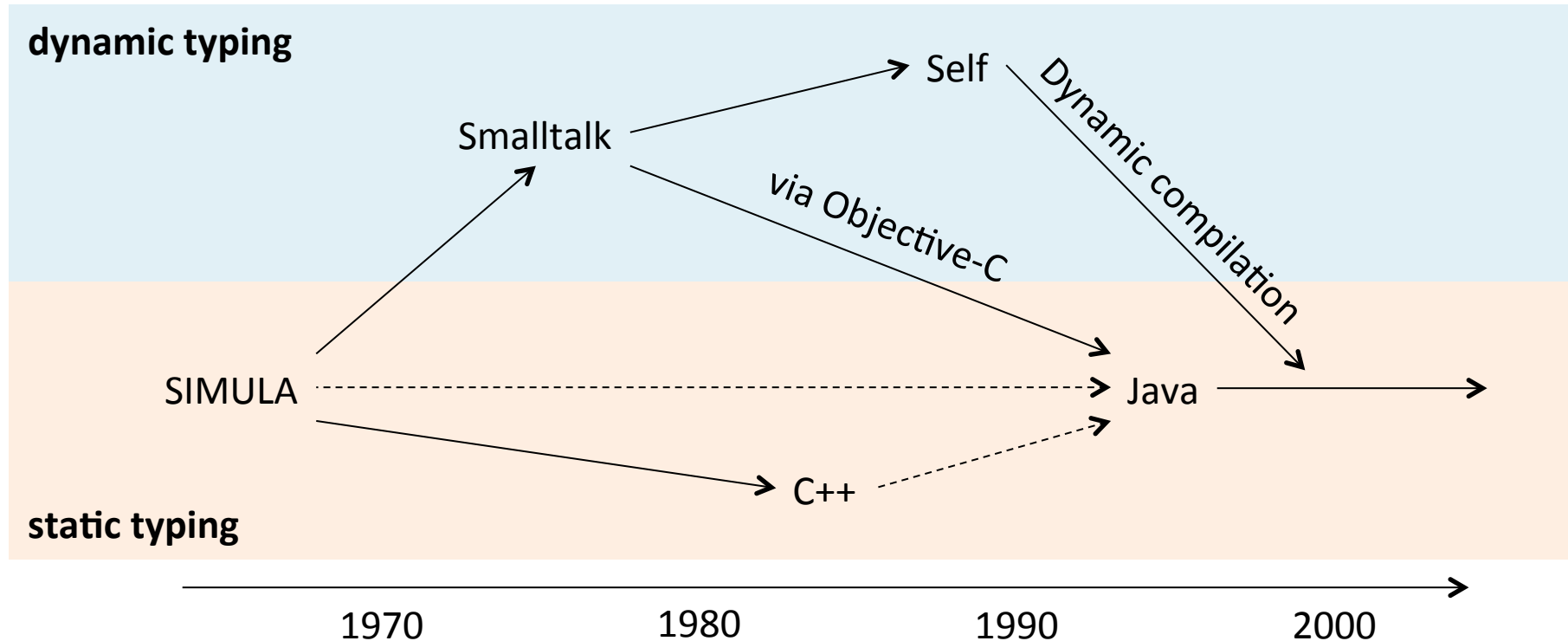
Görel Hedin

Revised: 2016-10-10

# This lecture



# Some influential OO languages



## Dynamic typing

At runtime, every object has a type.

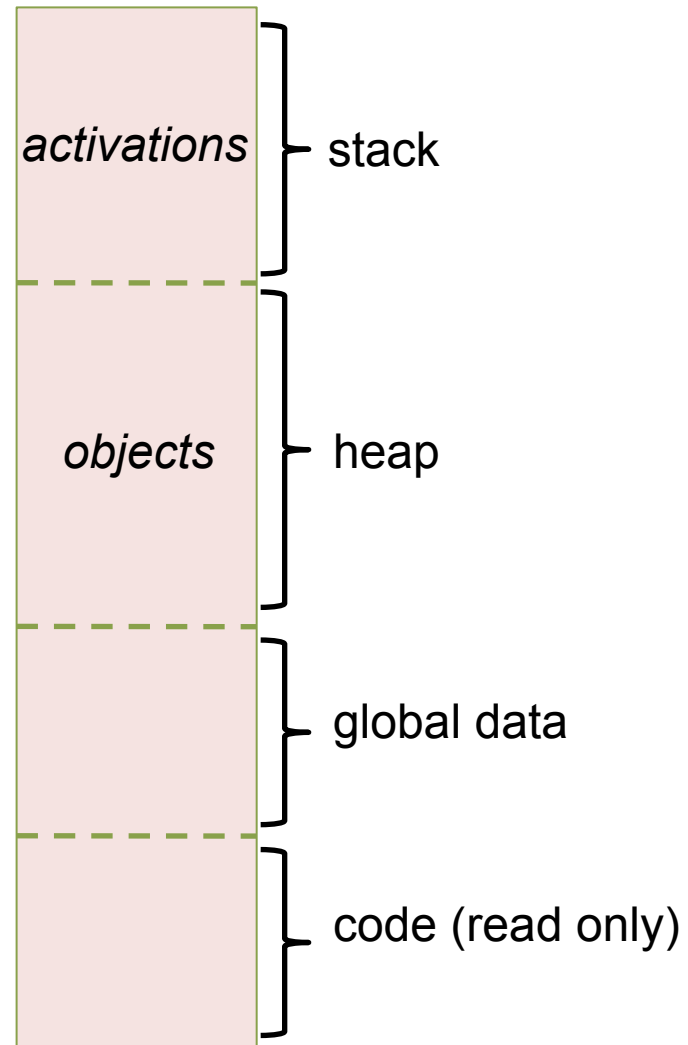
## Static typing

At runtime, every object has a type.

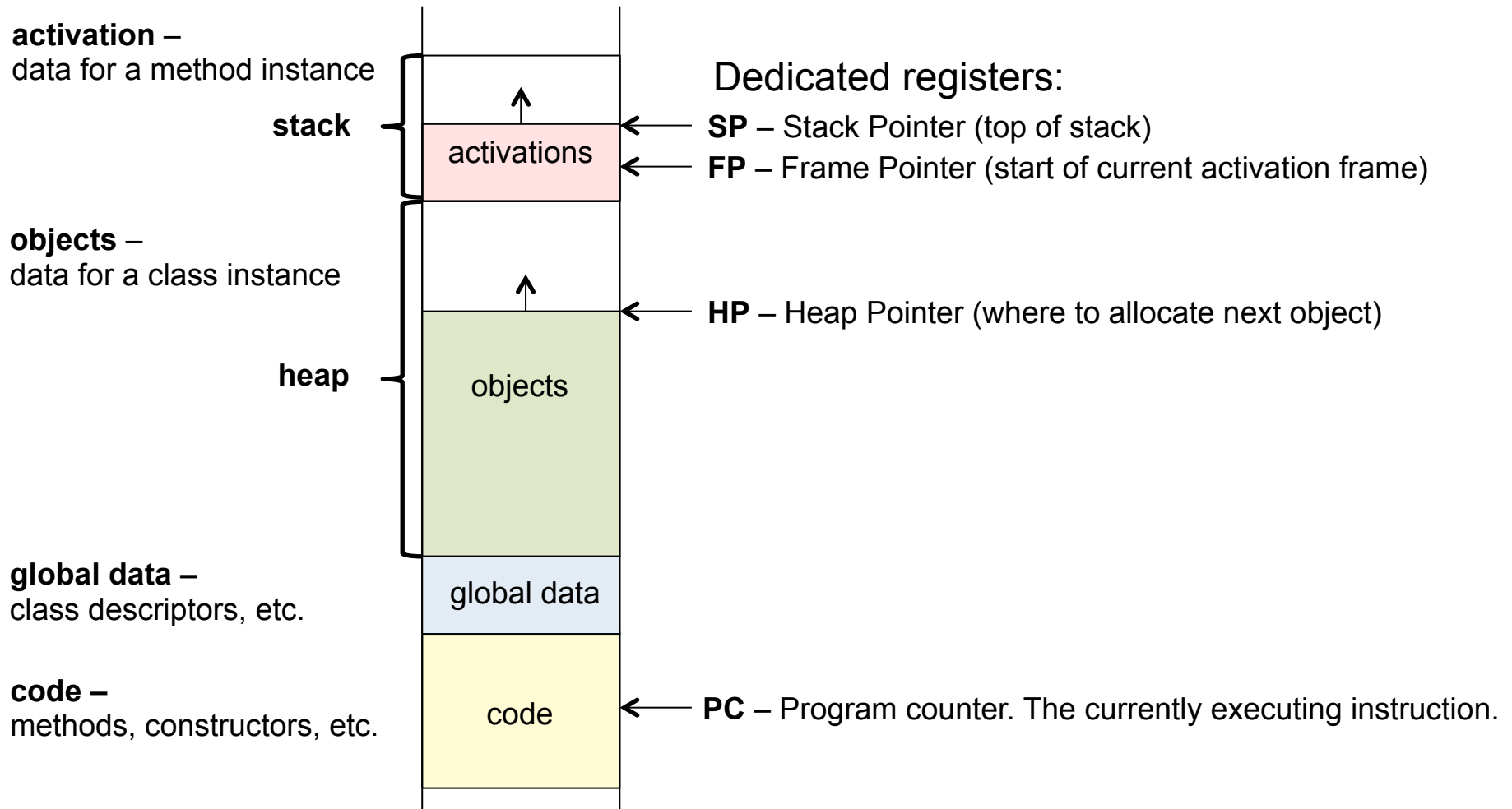
At compile-time, every variable has a type.

At runtime, the variable points to an object of at least that type.

# Example memory segments



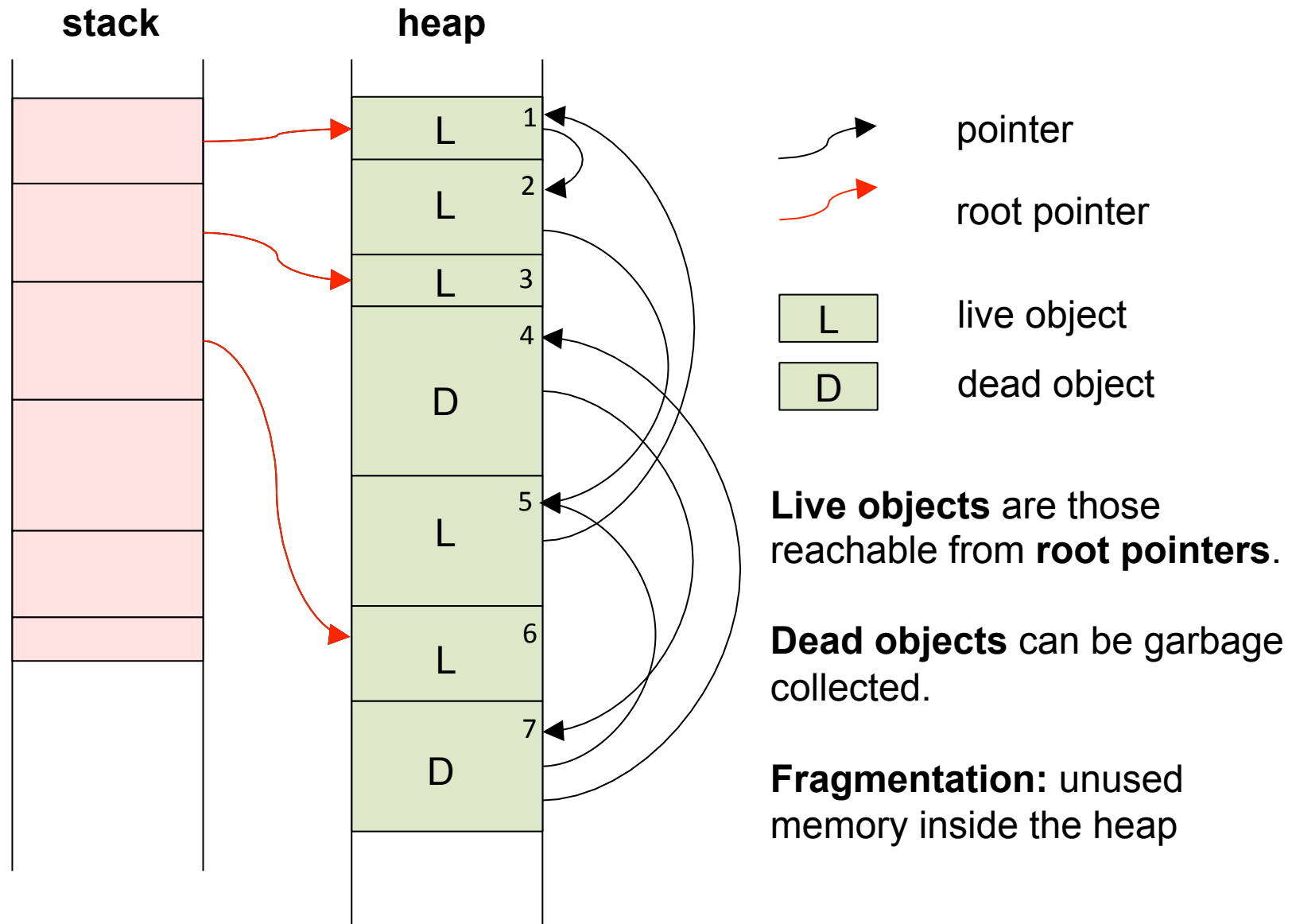
# Typical memory usage for OO languages



The figure shows typical use for statically loaded languages like Simula and C++. For languages with dynamic loading (like Java, Smalltalk, ...), class descriptors and code are placed on the heap, rather than in the global data and code segments.

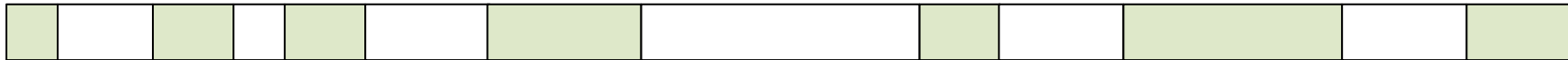
To support threads, each thread has its own stack. Typical stack size: 1 MB. Typical heap size: 80 MB.

# The heap

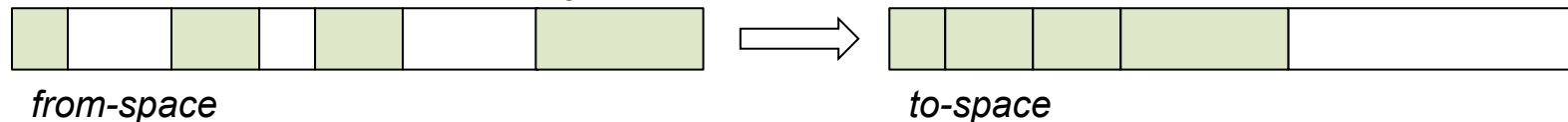


# Major garbage collection techniques

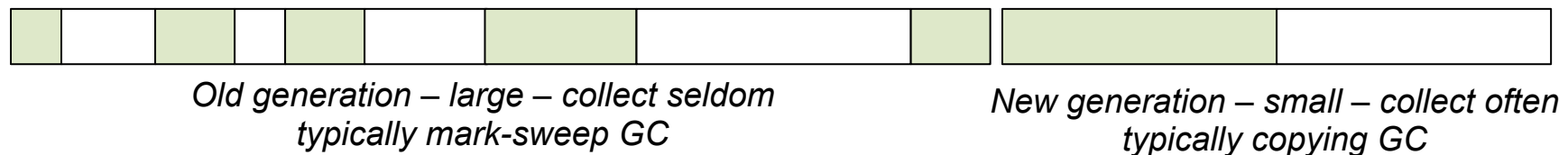
**Mark-sweep GC:** Follow all pointers and mark all live objects. Sweep heap and collect free objects. Or **compact** the heap to avoid fragmentation.



**Copying GC:** Divide heap into two spaces. Allocate new objects in *from-space*. When full, move all live objects to *to-space*. Switch *from-space* and *to-space*.

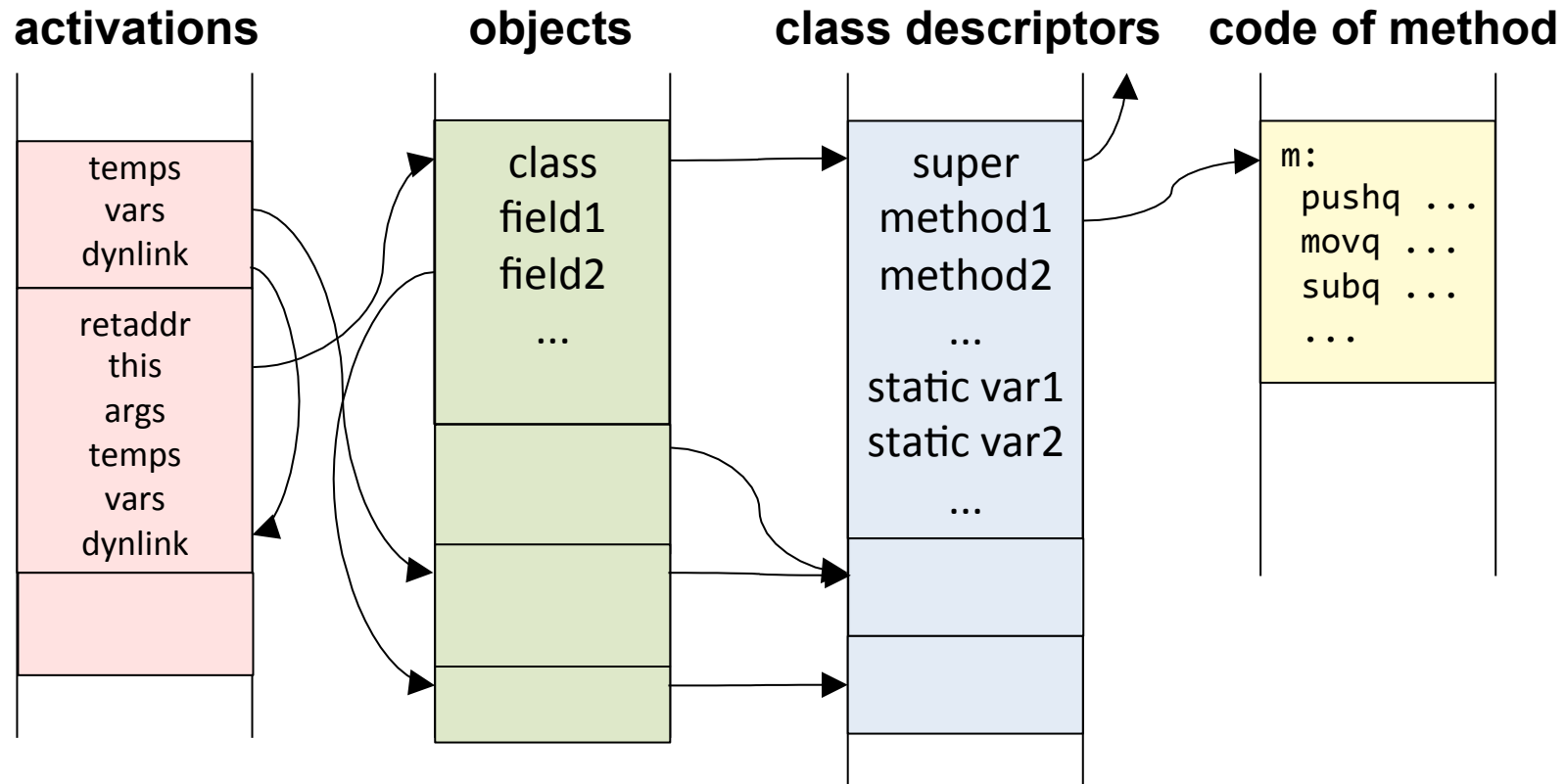


**Generational GC:** Efficient because most objects die young. Move (tenure) surviving objects to older generation.



**Reference counting:** Inefficient (overhead when reading and writing references). Deallocate when count=0. Does not handle cycles. Fragmentation problems.

# Typical runtime structures for objects



## A method activation

The "this" pointer (static link) is viewed as an extra argument.  
"This" is used to access fields and methods.  
Variables, args and temps can point to objects.

## An object

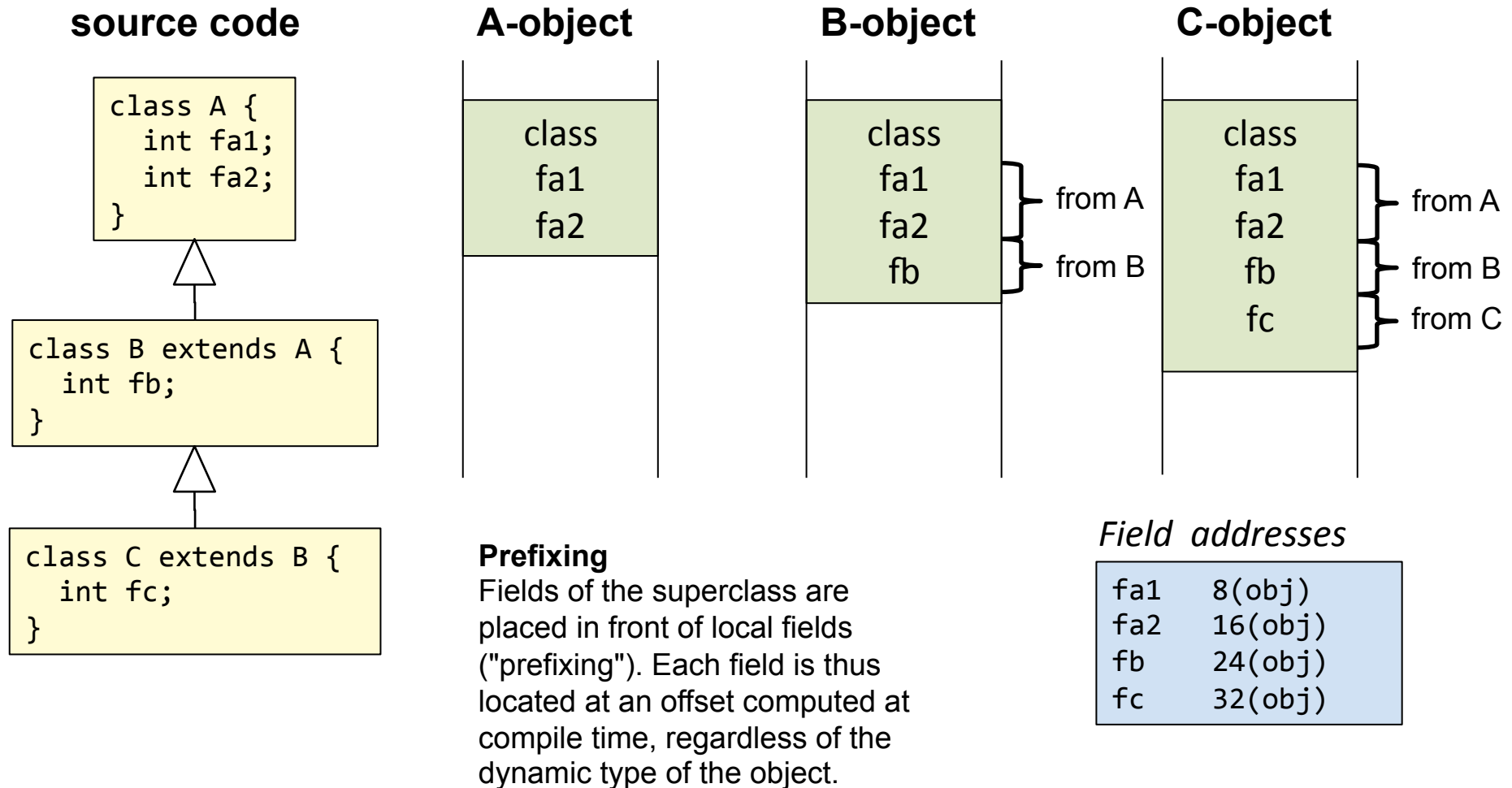
Has pointer to the class descriptor (for accessing methods).  
Can have fields that point to objects.

## A class descriptor

Can access super class through the super pointer.  
Has pointers to its methods.  
Can have static variables.



# Inheritance of fields, prefixing



# Access to fields (single inheritance)

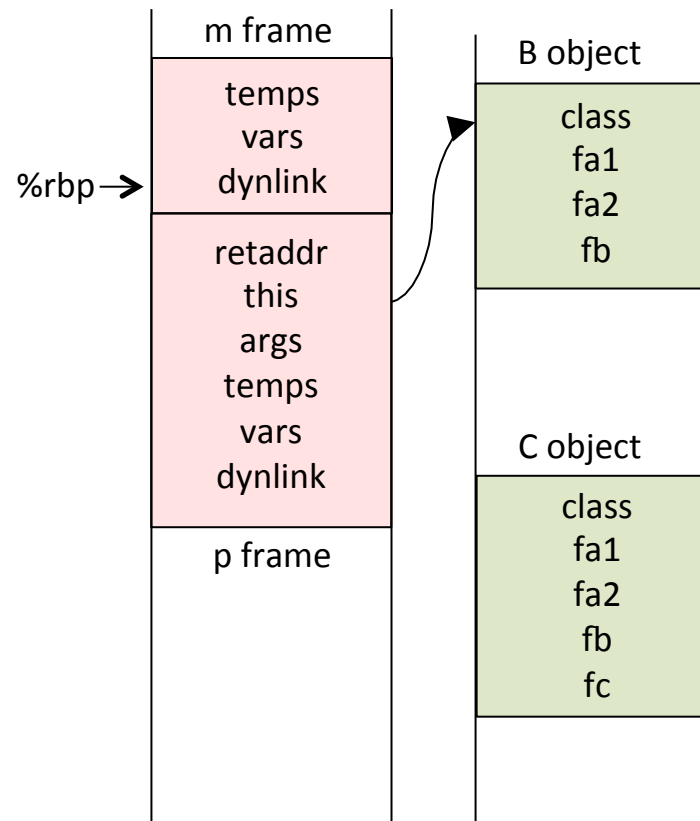
## source code

```
class A {  
  int fa1;  
  int fa2;  
  void m() {  
    fa1 = fa2;  
    ...  
  }  
}
```

```
class B extends A {  
  int fb;  
}
```

```
class C extends B {  
  int fc;  
}
```

```
void p(A r) {  
  r.m();  
}
```



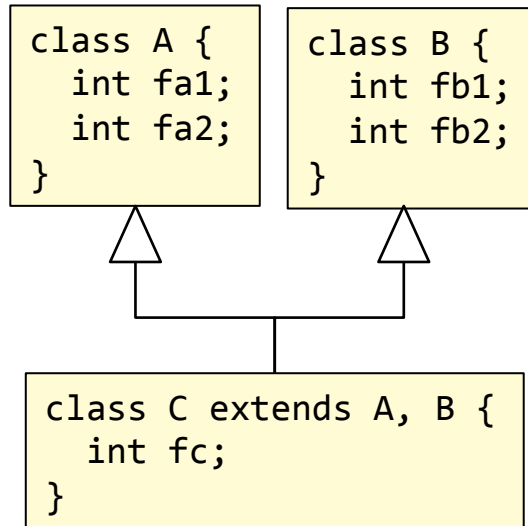
The code for **m** knows the static type of the object (**A**), but not the dynamic type (**B** or **C** in this case).

Because of prefixing, the code for **m** can access **fa1** and **fa2** through an efficient indirect access, using a fixed offset, without knowing the dynamic type of the object.

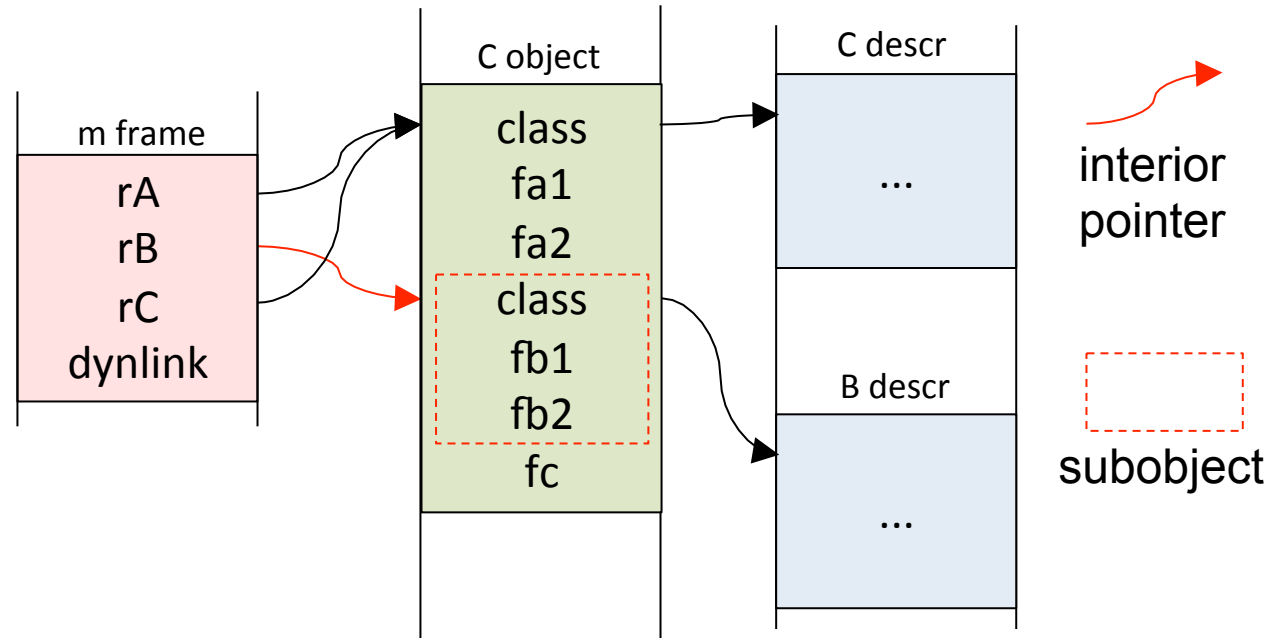
```
# Example code, assuming "this" pointer is at 16(%rbp):  
A-m:  
...  
movq 16(%rbp), %rax      # this -> rax  
movq 16(%rax), 8(%rax)   # fa2 -> fa1
```

# Access to fields (multiple inheritance, C++)

## source code



```
void m() {
  A rA = new C();
  B rB = rA;
  C rC = rA;
}
```



## Interior pointers and subobjects

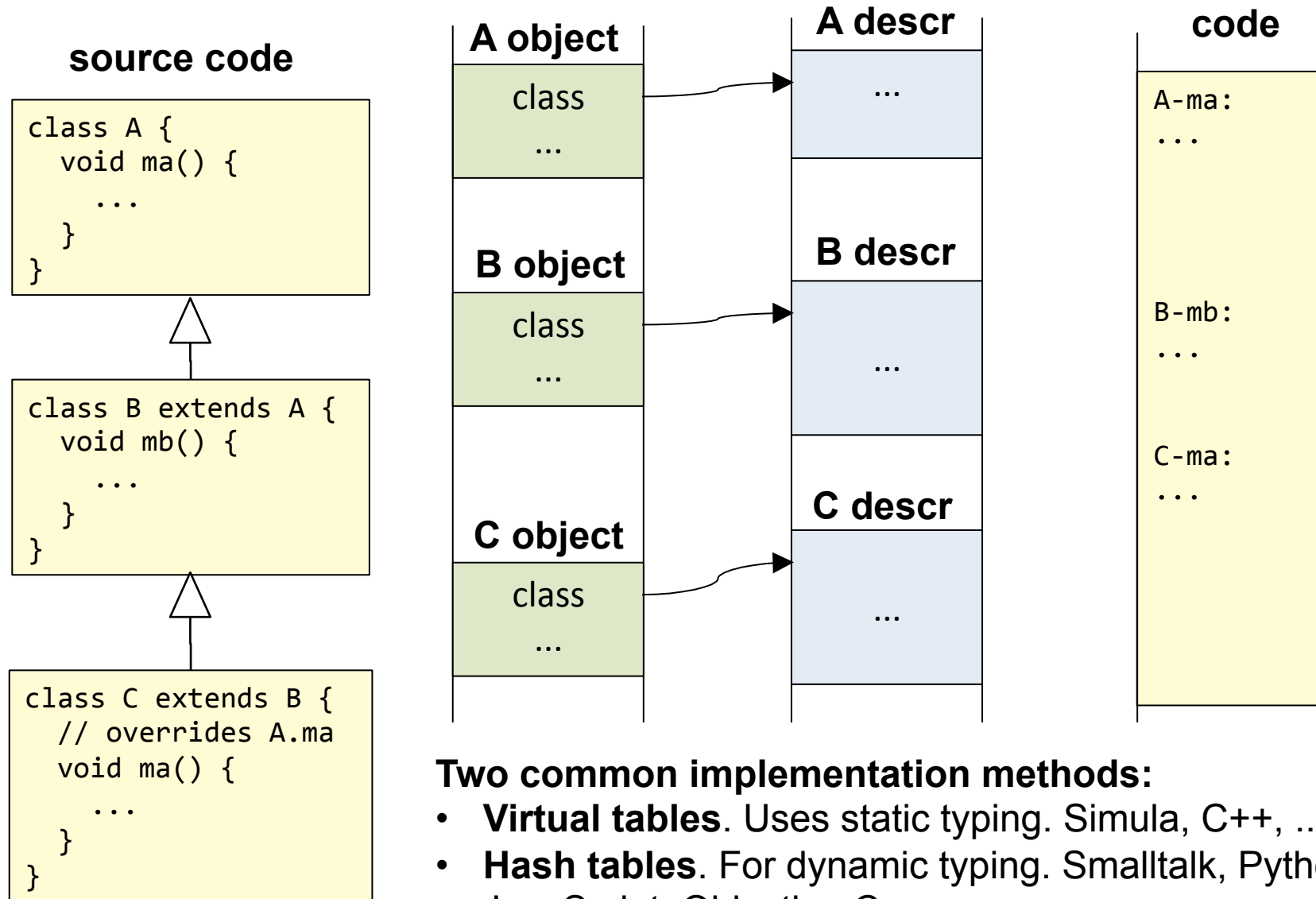
Parts of the class hierarchy are treated like single inheritance: rA and rC point to the full C object.

For remaining parts, allocate subobjects inside the main object. rB points to the *interior* of the C object, to the B subobject.

Gives problems for garbage collector:  
The GC needs to identify full objects. Solvable, but expensive.

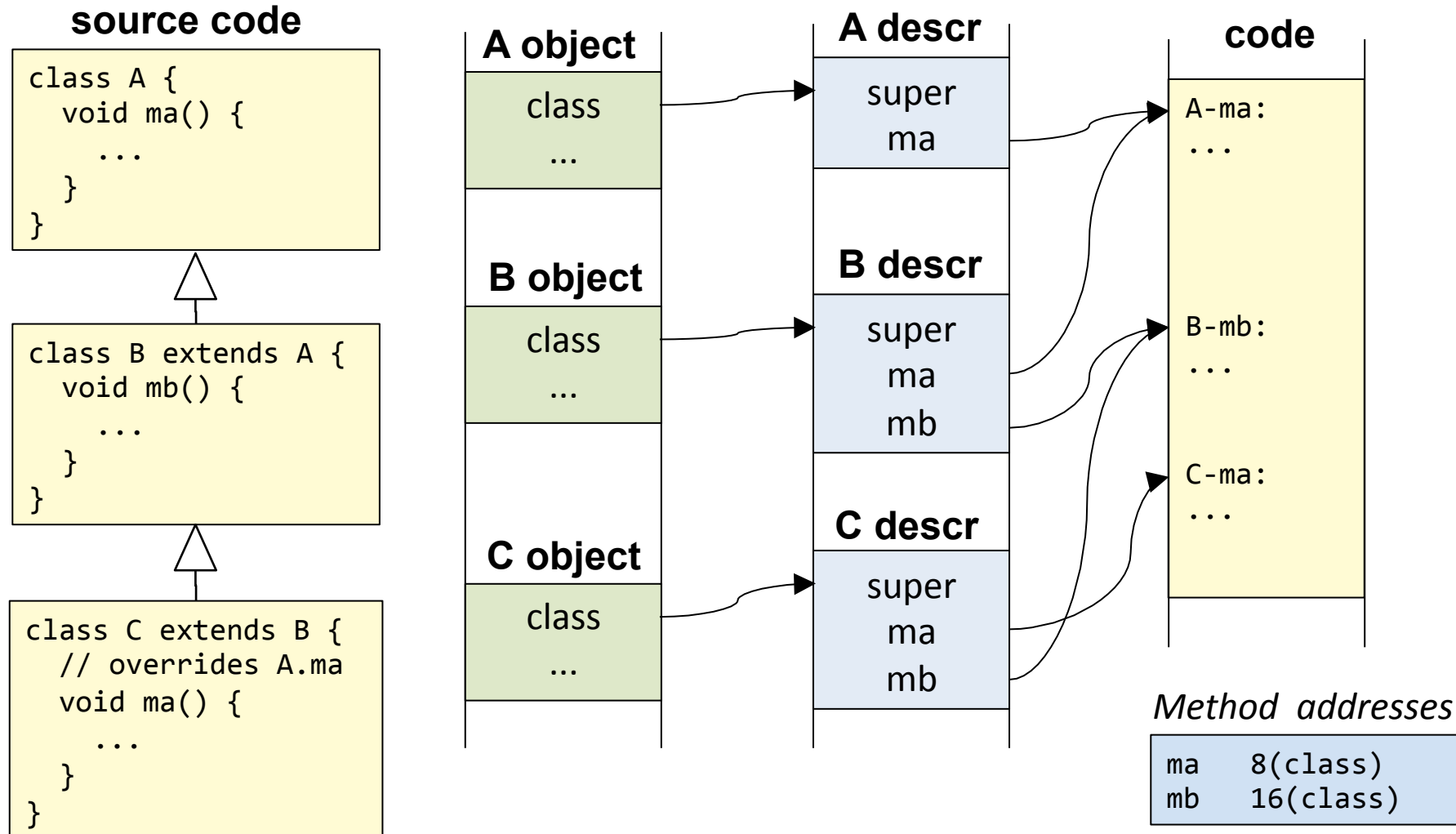
# Dynamic dispatch

(Calling methods in presence of inheritance and overriding)



# Virtual table dynamic dispatch

For statically typed languages: Simula, C++, ...



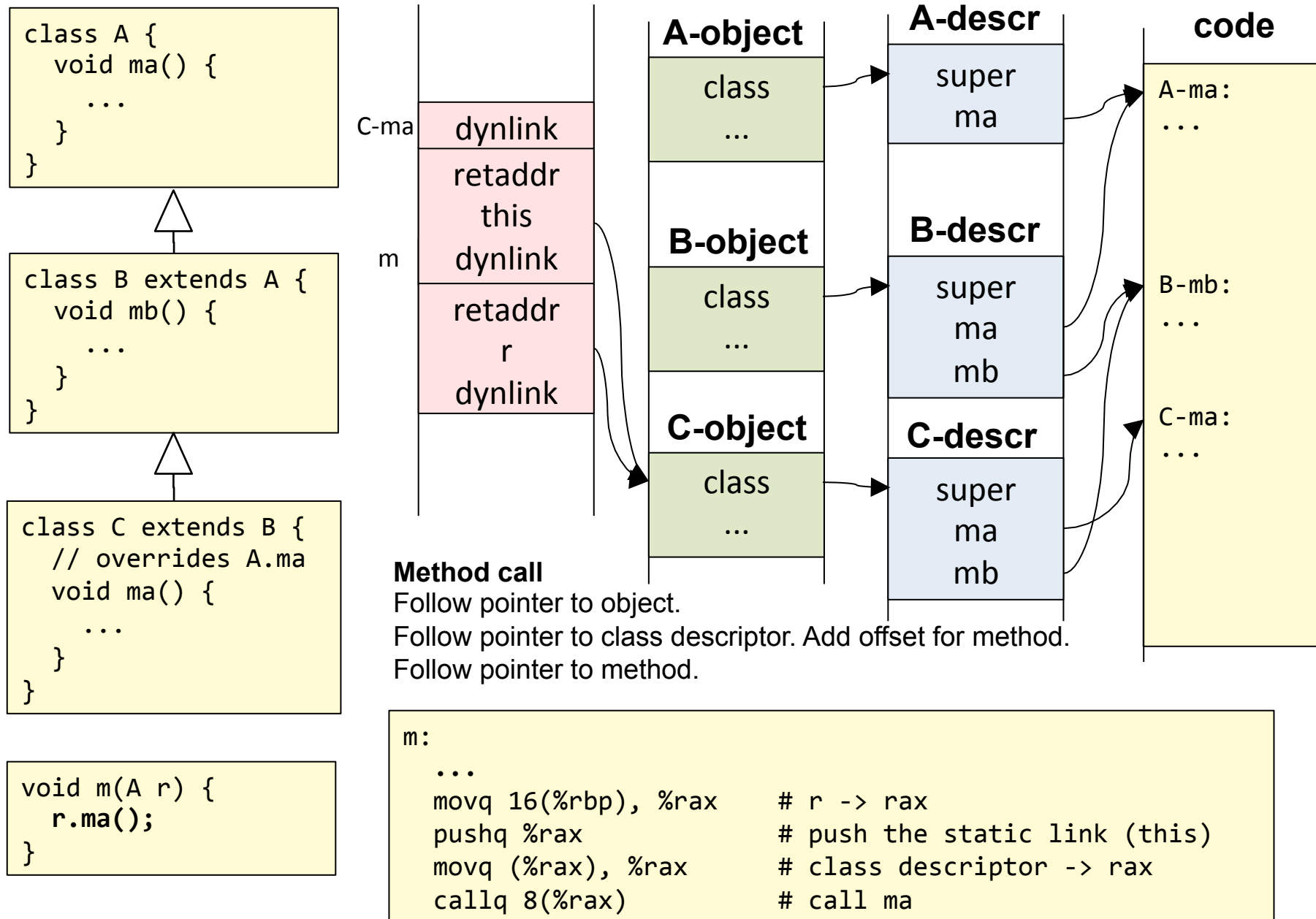
## Virtual tables

Class descriptor contains *virtual table* (often called "vtable").

Pointers to superclass methods are placed in front of locally declared methods ("prefixing").

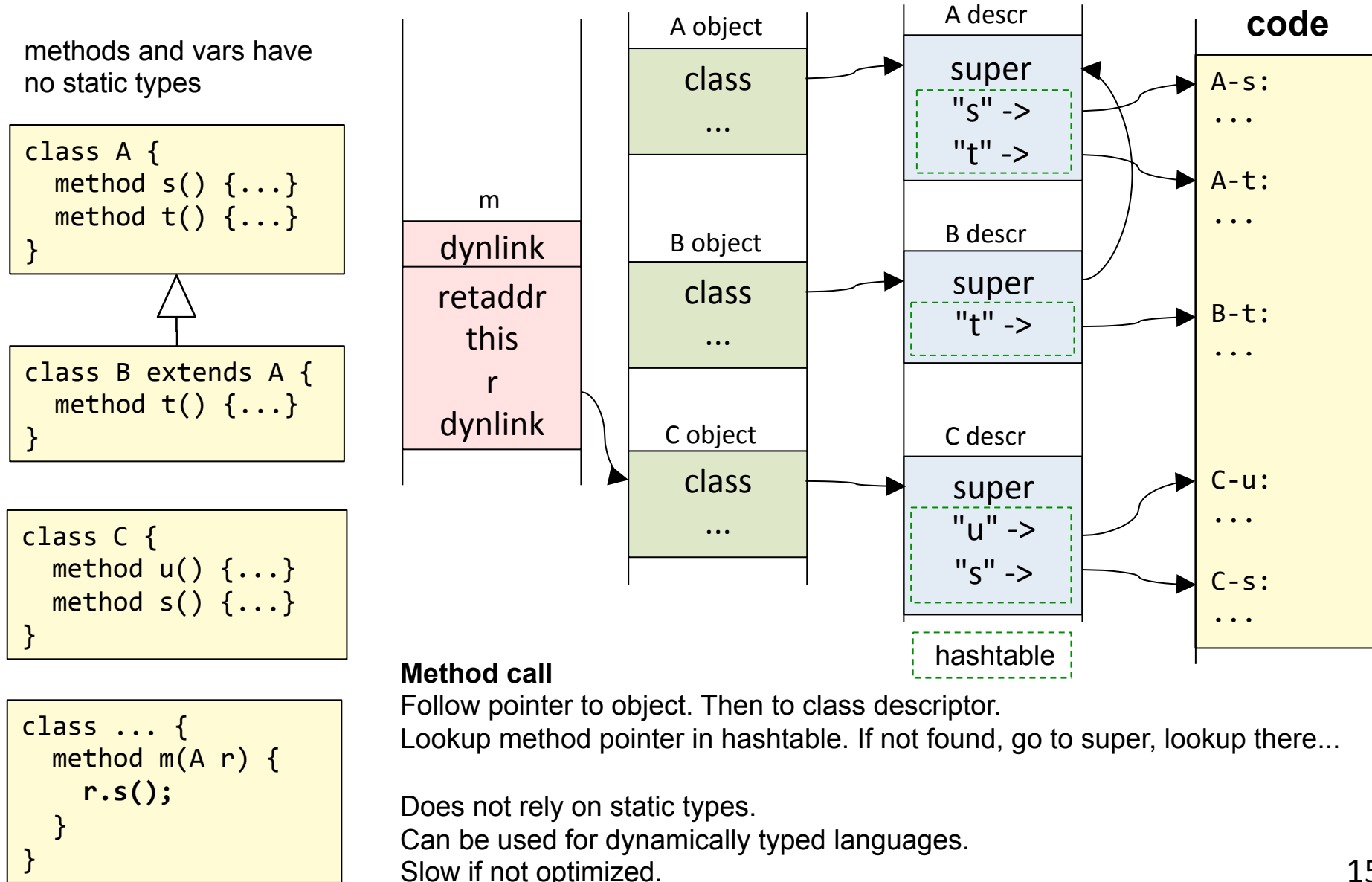
Each method pointer is located at an offset computed at compile time, using the static type.

# Calling a method via the virtual table



# Hash table dynamic dispatch

For dynamically typed languages: Smalltalk, Python, JavaScript, Objective-C, ...



# Comparison, dynamic dispatch

## **Virtual tables**

Can implement multiple inheritance by adapting prefixing, similarly to field access.

Cannot be used for dynamically typed languages.

Fast calls – only an indirect jump.

## **Hash tables**

No problem with multiple inheritance.

Can be used for dynamically typed languages.

Slow calls – need to do hash table lookup.

Both can be optimized...



# Optimization of procedural languages (C)

## **Local optimizations (within methods):**

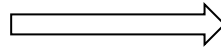
- common subexpression elimination
- constant propagation
- constant folding
- dead code elimination
- loop invariant code motion
- ...

**Inlining** (avoid calls, get more code to optimize over)

# Example local optimizations

```
a = b * c + d;  
e = f + b * c;
```

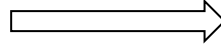
common subexpression elimination



```
t = b * c;  
a = t + d;  
e = f + t;
```

```
int a = 37;  
return a + 5;
```

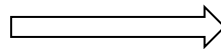
constant propagation



```
int a = 37;  
return 37 + 5;
```

```
int a = 37;  
return 37 + 5;
```

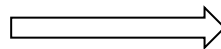
constant folding



```
int a = 37;  
return 42;
```

```
int a = 37;  
return 42;
```

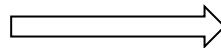
dead code elimination



```
return 42;
```

```
for (int i ...) {  
    a = b + 3;  
    x[i] = a * i;  
}
```

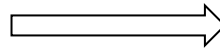
loop invariant code motion



```
a = b + 3;  
for (int i ...) {  
    x[i] = a * i;  
}
```

# Inlining

```
void f(int b) {  
    ...  
    for (int i ...) {  
        a = g(b);  
        x[i] = a * i;  
    }  
}  
  
int g(int x) {  
    return x + 3;  
}
```



```
void f(int b) {  
    ...  
    for (int i ...) {  
        a = b + 3;  
        x[i] = a * i;  
    }  
}  
  
int g(int x) {  
    return x + 3;  
}
```

After inlining, there could be more opportunities for local optimizations.

# Optimization of OO languages

## **Difficult to optimize OO with conventional techniques**

- Many small methods – not much to optimize in each
- Virtual methods difficult to inline – actual method not known until runtime

## **If methods could be inlined...**

... we could save the expensive calls

... we would get larger code chunks to optimize over

# Approaches to optimization of OO code

## **Static compilation approaches**

Analysis of complete programs: "whole world analysis"

Find methods to be inlined. Then optimize further.

Not used in practice: cannot be used with dynamic loading.

## **Dynamic compilation approaches**

Inline methods at runtime (self-modifying code)

Dynamic compilation and optimization (at runtime)

Use simple conventional optimization techniques

(must be fast enough at runtime)

Very successful in practice (Java, CLR, Javascript, ...)

Can beat optimized C for some benchmarks.

# Other important optimizations in OO

**Dynamic type tests (casts, instanceof)**

**Synchronization and thread switches**

**Garbage collection**

# Interpretation vs Compilation in Java

## **Interpreting JVM**

portable but slow

## **JIT – Just-In-Time compilation**

compile each method to machine code the first time it is executed  
requires very fast compilation – no time to optimize

## **AOT – Ahead-of-time compilation**

Generate machine code for a complete program, before execution. This is "normal" compilation, the way it is done in C, C++, ...  
Problem to use this approach for Java: cannot support dynamic loading.

## **Adaptive optimizing compiler**

Run interpreter initially to get profiling data  
Find "hot spots" which are translated to machine code, and then optimized  
May outperform AOT compilers in some cases!  
The approach used today in the SUN/Oracle JVM, called "HotSpot".

# Inline call caches

## a way to optimize method calls at runtime

### Based on hash table lookup

Do a normal (slow) lookup. The result is a method implementation, say Bus-m.

Guess that the next call will be for an object of the same type (Bus), i.e., to Bus-m.

Replace the call with a direct call to Bus-m-prologue, with the receiver as argument.

The prologue checks if the receiver is of the guessed type (Bus).

If so, continue executing Bus-m. If not, do a normal (slow) lookup.

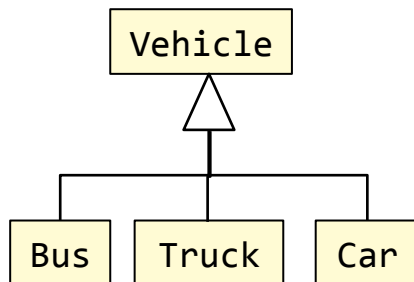
### Original calling code

```
Vehicle v = ...;
while (...) {
    v = aList.get();
    v.m();
}
```

optimize →

### Optimized calling code

```
Vehicle v = ...;
while (...) {
    v = aList.get();
    Bus-m-prologue(v);
}
```



### Called method:

```
Bus-m-prologue:
    if (receiver is not a Bus)
        receiver.m(); // Ordinary slow lookup
Bus-m:
    normal method body
    ...
```



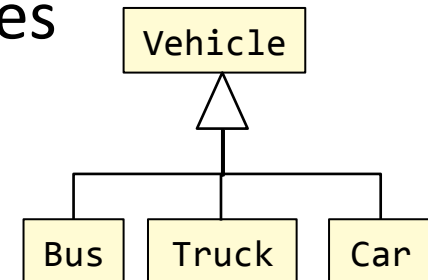
# Polymorphic inline caches (PICs)

a generalization of inline call caches

**Handle several possible object types**

Inline the prologues into the calling code.

Check for several types.



Inlined call cache

```
Vehicle v = ...;
while (...) {
    v = aList.get();
    Bus-m-prologue(v);
}
```

optimize →

Methods:

```
Bus-m-prologue:
    if (!receiver is a Bus)
        receiver.m(); // normal lookup
Bus-m:
    normal method body
    ...
```

Polymorphic inlined cache

```
Vehicle v = ...;
while (...) {
    v = aList.get();
    if (v is a Bus)
        Bus-m(v)
    else if (v is a Car)
        Car-m(v)
    else
        v.m(); // normal lookup
}
```

Methods:

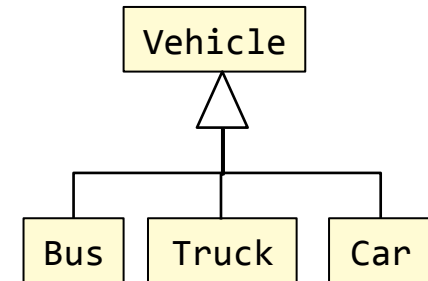
```
Bus-m:
    ...
Car-m:
    ...
```

# Inlining method bodies

Can be done after inlining calls

## Inlining method bodies

Copy the called methods into the calling code



### Polymorphic inlined cache

```
Vehicle v = ...;
while (...) {
    v = aList.get();
    if (v is a Bus)
        Bus-m(v)
    else if (v is a Car)
        Car-m(v)
    else
        v.m(); // normal lookup
}
```

### Methods:

```
Bus-m:
...
Car-m:
...
```

optimize →

### with inlined methods

```
Vehicle v = ...;
while (...) {
    v = aList.get();
    if (v is a Bus)
        ... // code for Bus-m
    else if (v is a Car)
        ... // code for Car-m
    else
        v.m(); // normal lookup
}
```

### Methods:

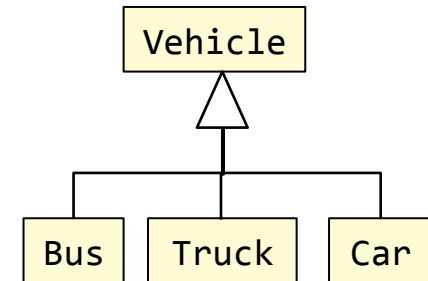
```
Bus-m:
...
Car-m:
...
```

# Further optimization

**Now there is a large code chunk at the calling site**

Ordinary local optimizations can now be done

- common subexpression elimination
- loop invariant code motion
- ...



Polymorphic inlined cache

```
Vehicle v = ...;
while (...) {
  v = aList.get();
  if (v is a Bus)
    Bus-m(v)
  else if (v is a Car)
    Car-m(v)
  else
    v.m(); // normal lookup
}
```

Methods:

```
Bus-m:
...
Car-m:
...
```

optimize →

with inlined methods

```
Vehicle v = ...;
while (...) {
  v = aList.get();
  if (v is a Bus)
    ... // code for Bus-m
  else if (v is a Car)
    ... // code for Car-m
  else
    v.m(); // normal lookup
}
```

Methods:

```
Bus-m:
...
Car-m:
...
```

# Dynamic adaptive compilation

**Keep track of execution profile**

**Add PICs dynamically**

Order cases according to frequency

Inline the called methods if sufficiently frequent

Optimize the code if sufficiently frequent

**Adapt the optimizations depending on current profile**

# Dynamic adaptive compilation

**Techniques originated in the Smalltalk and Self compiler**

**Adapted to Java in SUN/Oracle's HotSpot JVM**

Techniques originally developed for dynamically typed languages useful also for statically typed languages!  
Dynamic adaptive optimizations may outperform optimizations possible in a static compiler!

**Client vs Server compiler**

Local optimizations vs heavy inlining and other memory intensive optimizations.

**Warm-up vs. Steady state**

Slower when the program starts (warm-up). Fast after a while (steady-state).

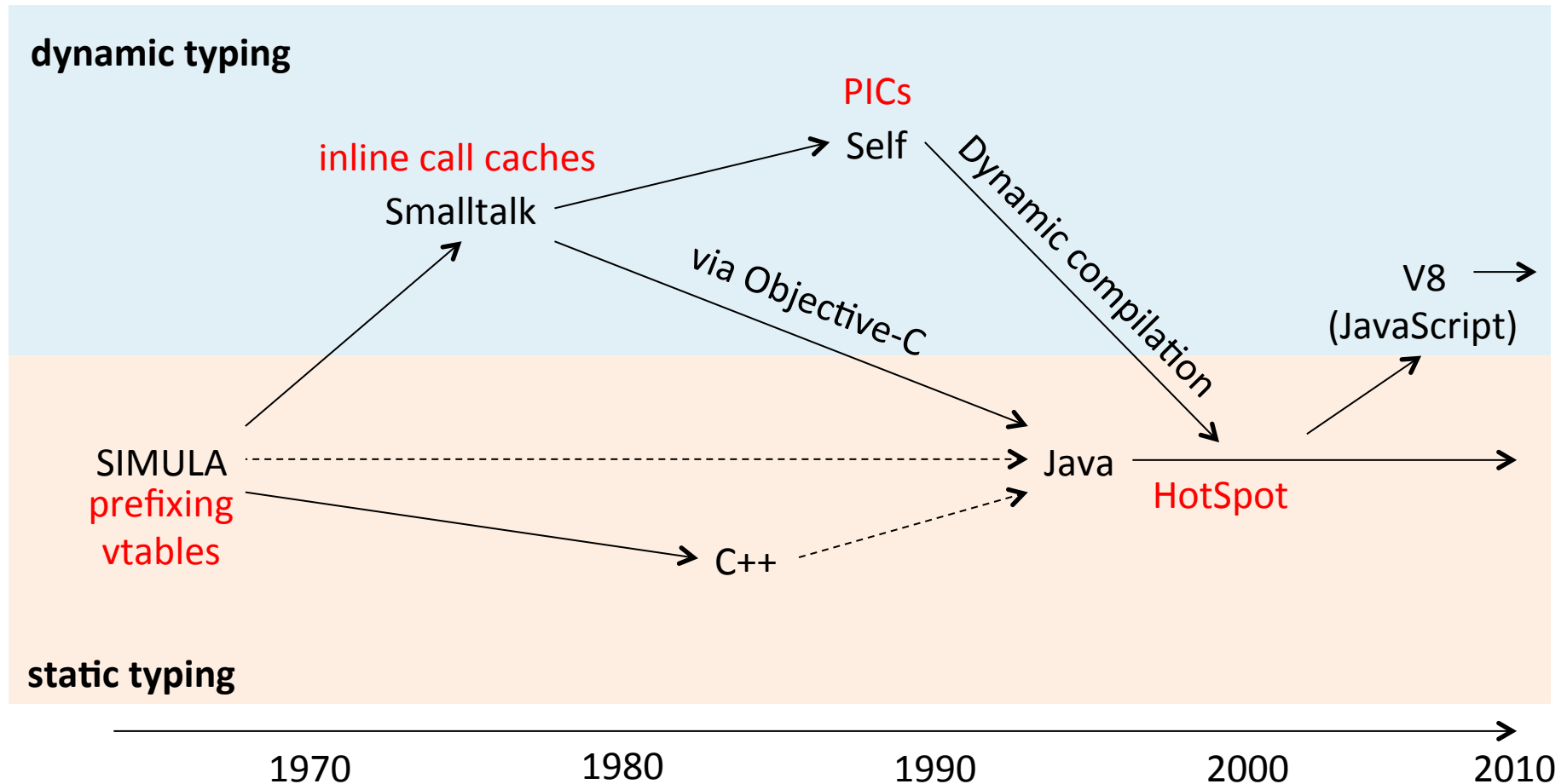
**A huge success:**

Fast execution in spite of fast compilation and dynamic loading.

Now used in other major languages like C# (CLR platform), Javascript, etc.

Many languages compile to Java Bytecode to take advantage of the HotSpot JVM.

# Major advances in OO implementation



# Summary questions

- What is the difference between dynamic and static typing?
- Is Java statically typed?
- What is a heap pointer?
- How are inherited fields represented in an object?
- What is prefixing?
- How can dynamic dispatch be implemented?
- What is a virtual table?
- Why is it not straightforward to optimize object-oriented languages?
- What is an inline call cache?
- What is a polymorphic inline cache (PIC)?
- How can code be further optimized when call caches are used?
- What is meant by dynamic adaptive compilation?