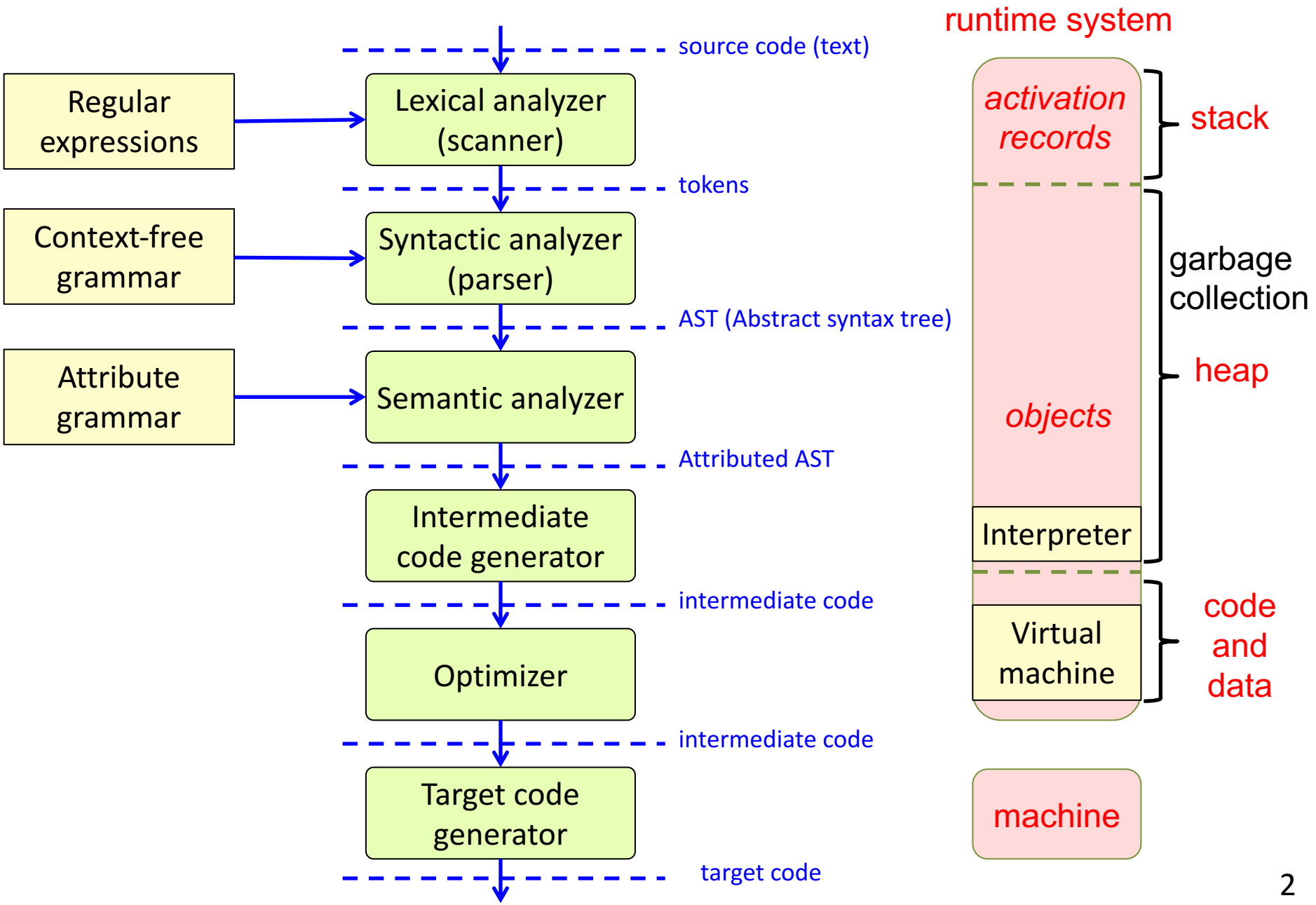
EDAN65: Compilers, Lecture 13

Runtime systems for object-oriented languages

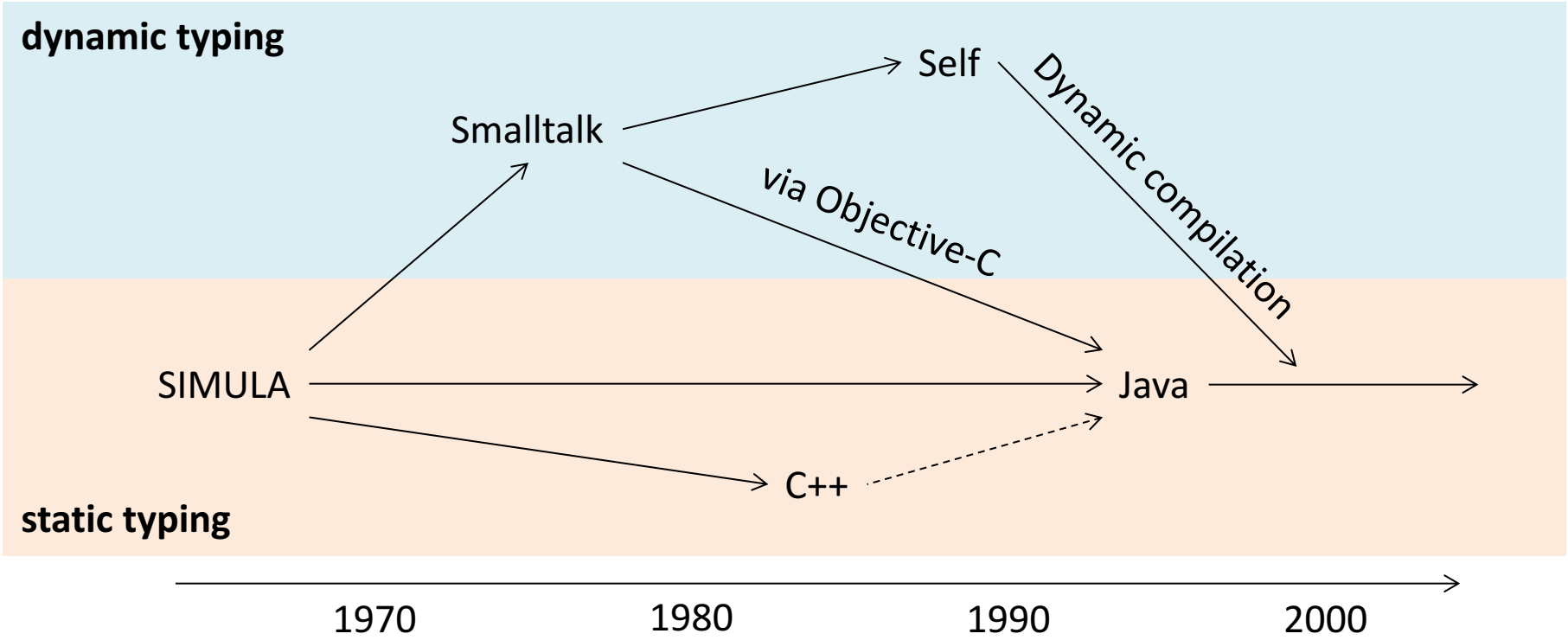
Görel Hedin

Revised: 2017-10-08

This lecture



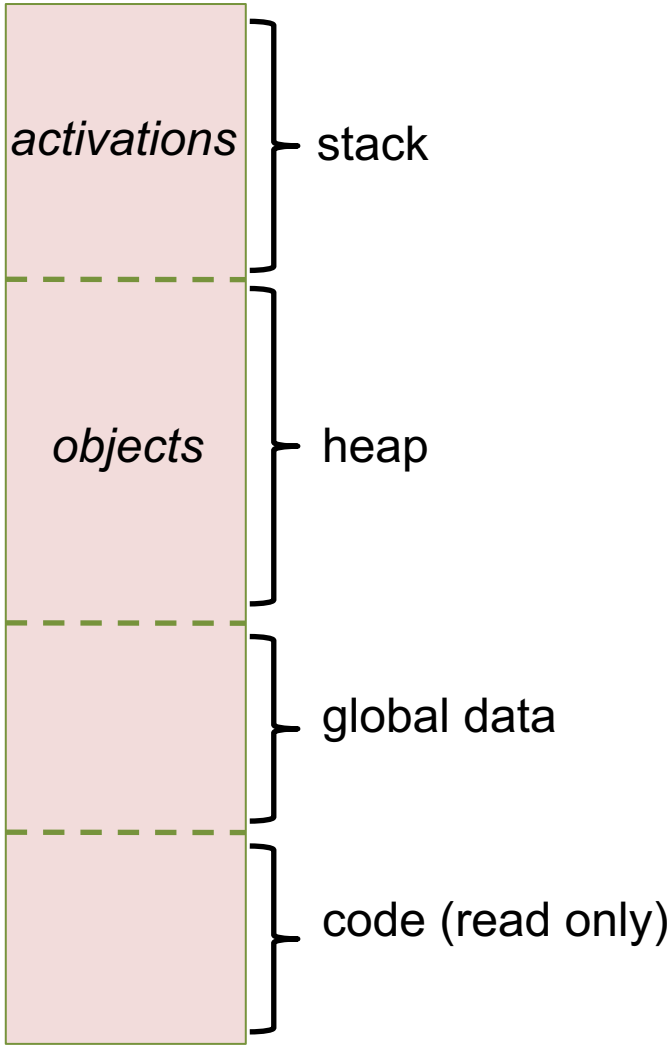
Some influential OO languages



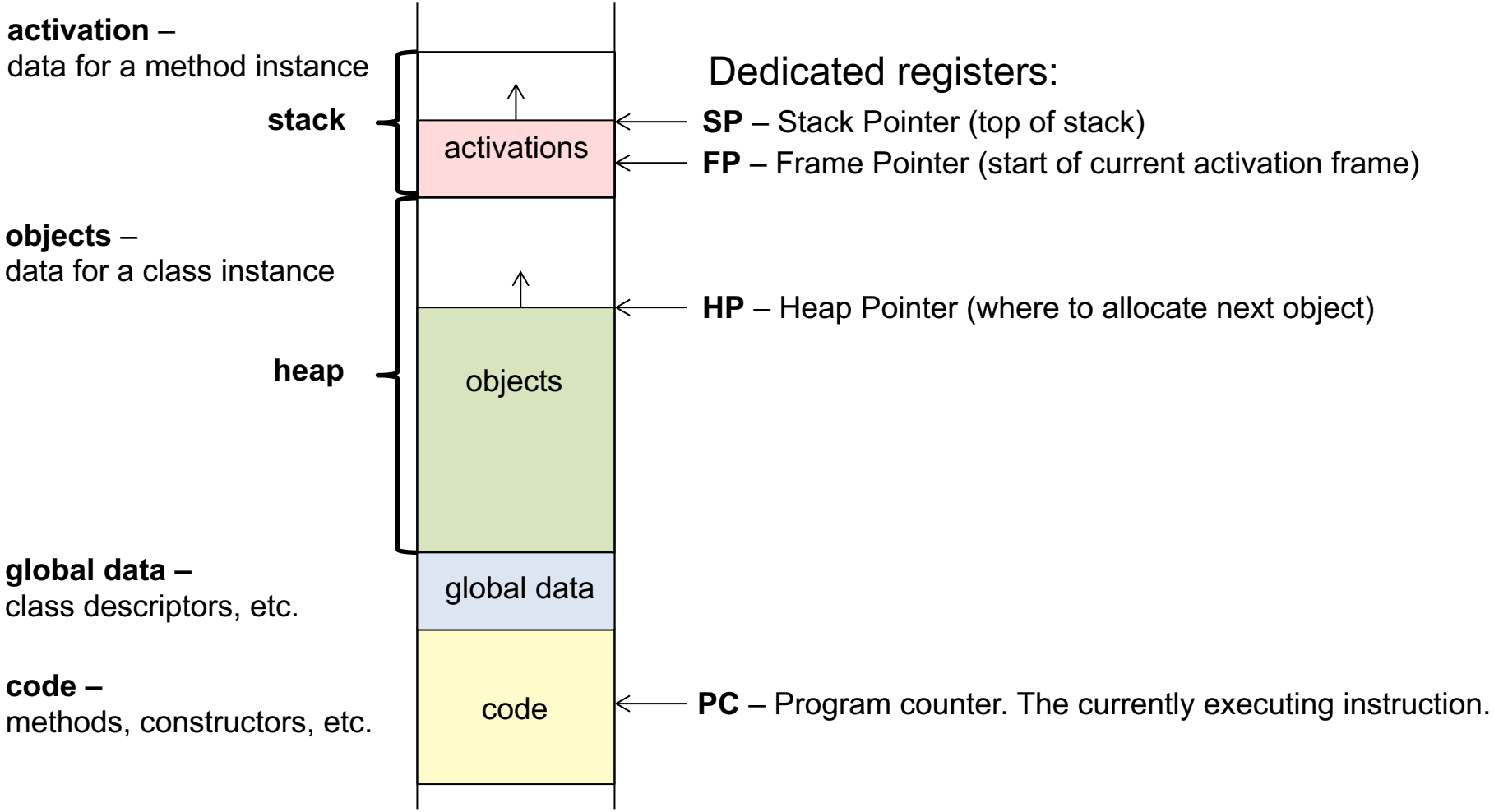
Dynamic typing
At runtime, every object has a type.

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

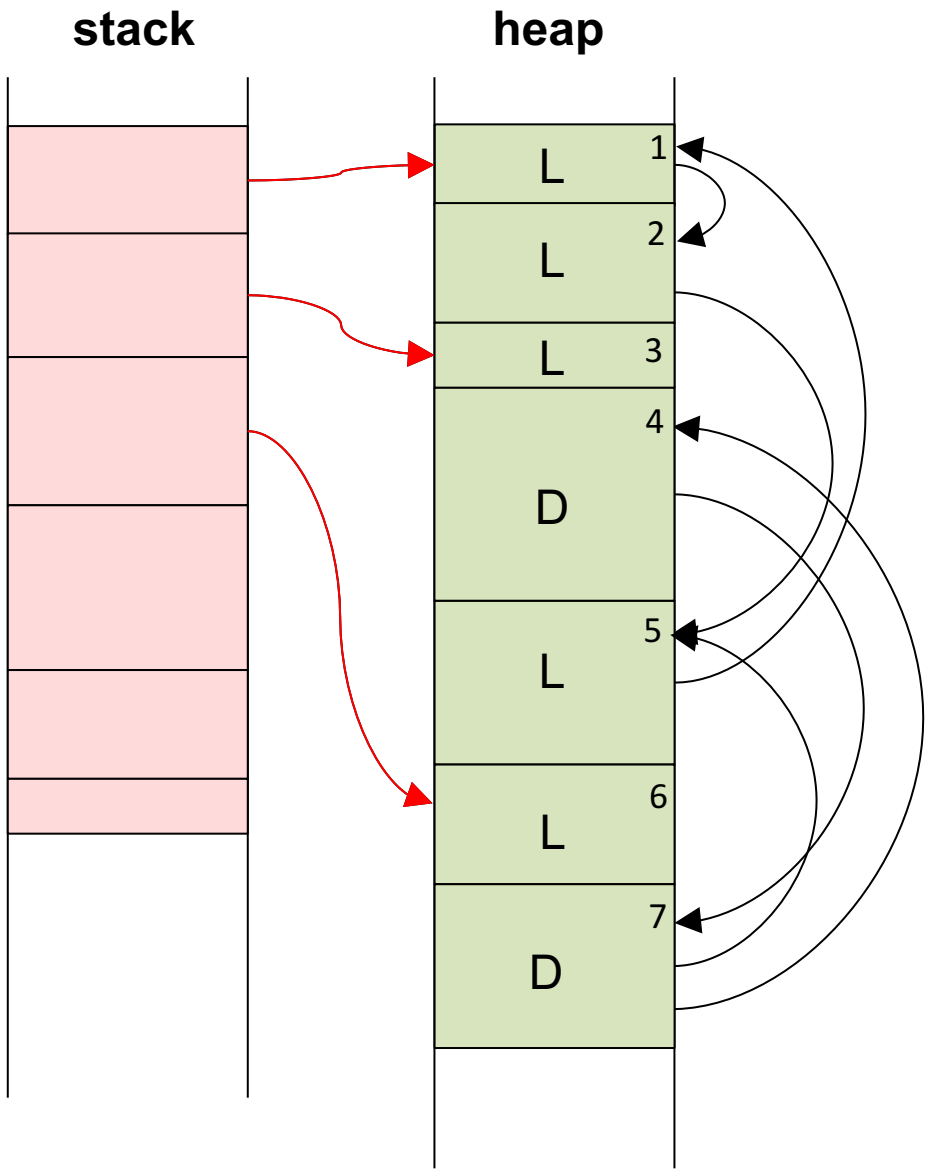


Typical memory usage for OO languages



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



pointer

root pointer

L live object

D dead object

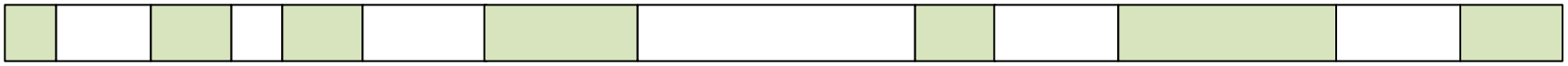
Live objects are those reachable from **root pointers**.

Dead objects can be garbage collected.

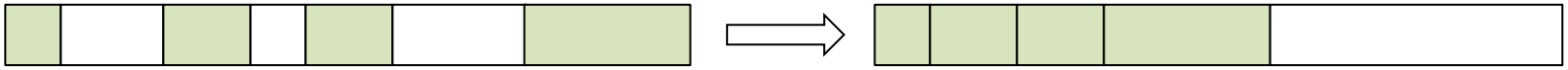
Fragmentation: unused memory inside the heap

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.

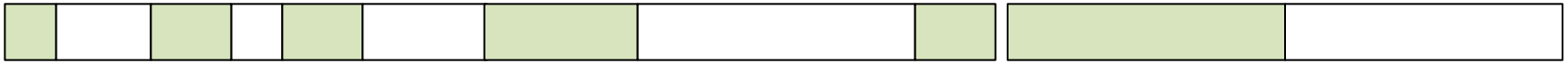


Copying GC: Divide heap into two spaces. Allocate new objects in *from-space*.
When full, move all live objects to *to-space*. Switch *from-space* and *to-space*.



from-space *to-space*

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

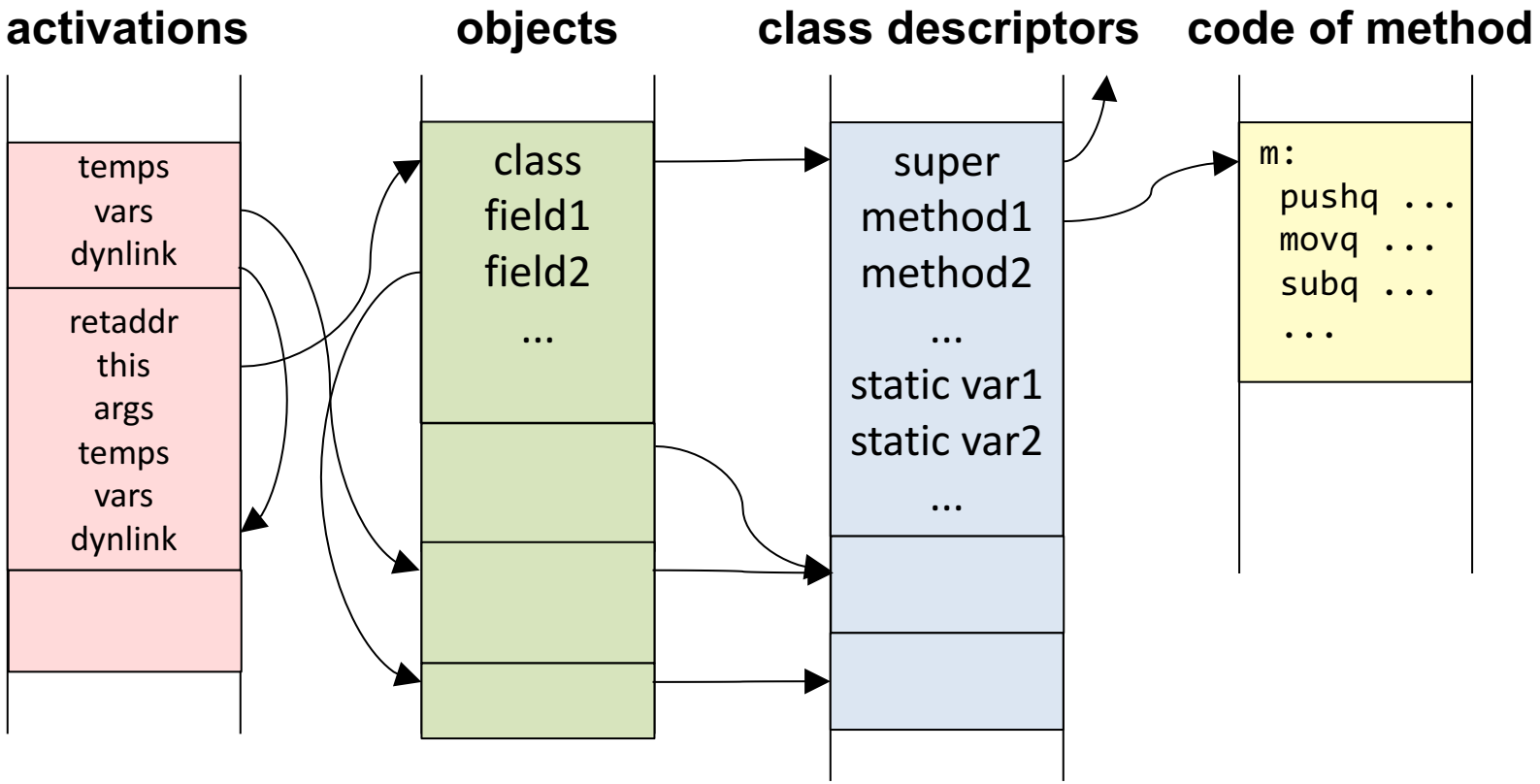


Old generation – large – collect seldom
typically mark-sweep GC

New generation – small – collect often
typically copying GC

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

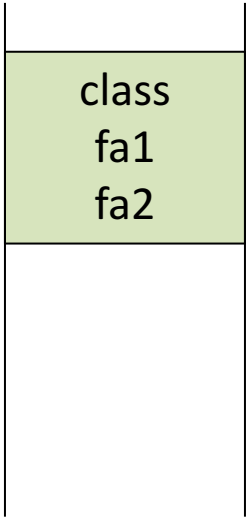
source code

```
class A {
  int fa1;
  int fa2;
}
```

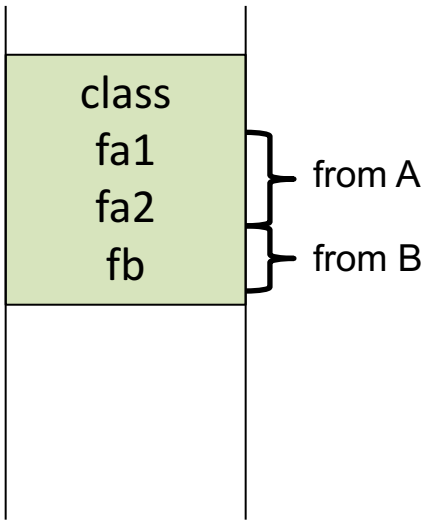
```
class B extends A {
  int fb;
}
```

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

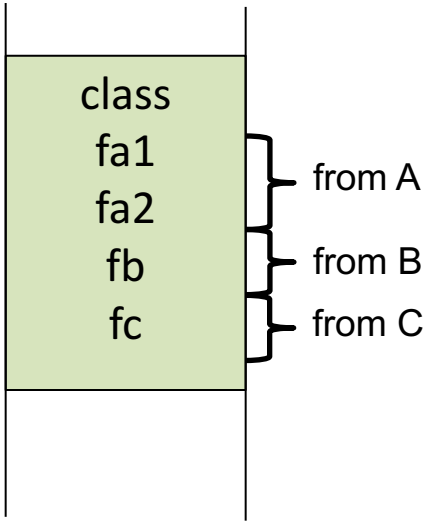
A-object



B-object



C-object



Prefixing
Fields of the superclass are placed in front of local fields ("prefixing"). Each field is thus located at an offset computed at compile time, regardless of the dynamic type of the object.

Field addresses

```
fa1   8(obj)
fa2   16(obj)
fb    24(obj)
fc    32(obj)
```

Access to fields (single inheritance)

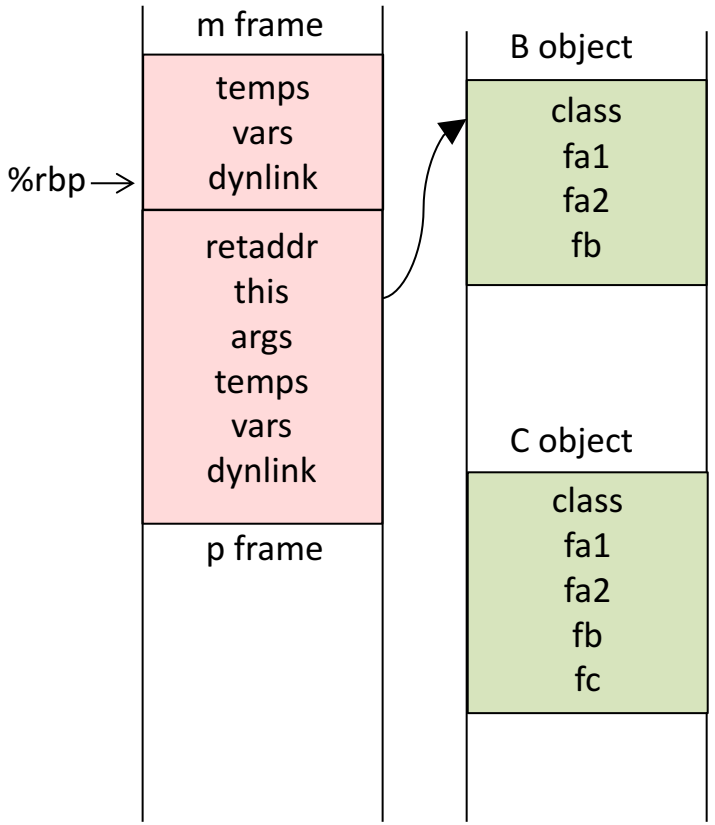
source code

```
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
```

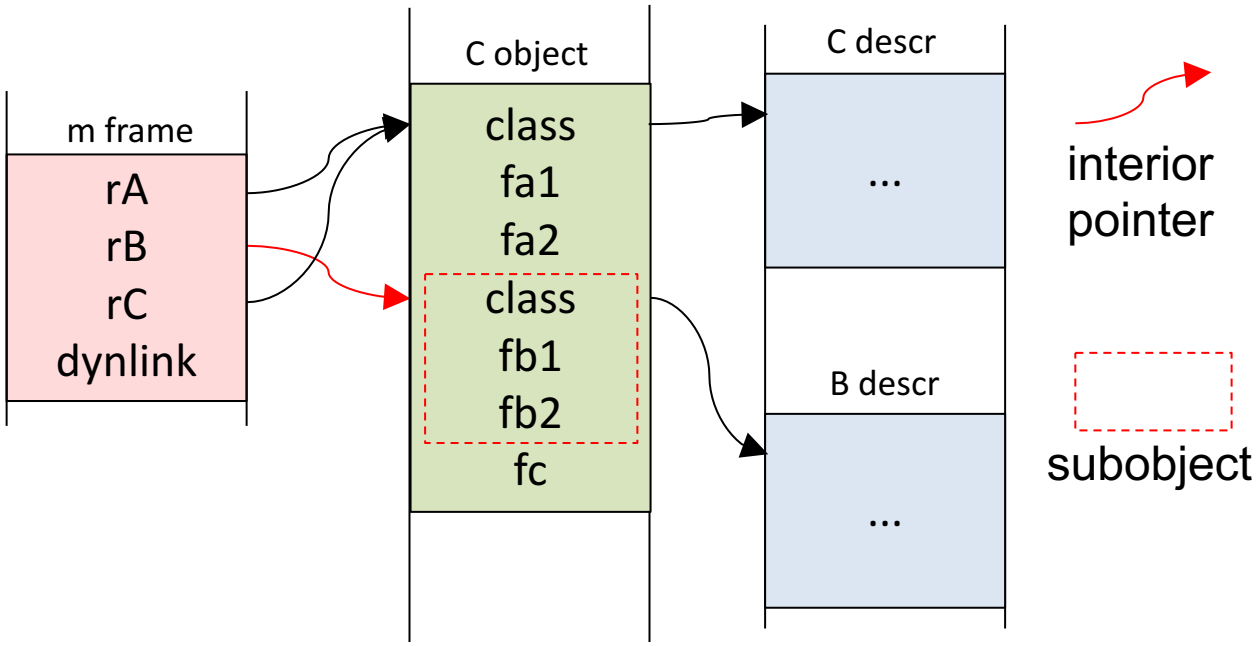
Access to fields (multiple inheritance, C++)

source code

```
class A {
  int fa1;
  int fa2;
}
```

```
class B {
  int fb1;
  int fb2;
}
```

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




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

Interior pointers and subobjects

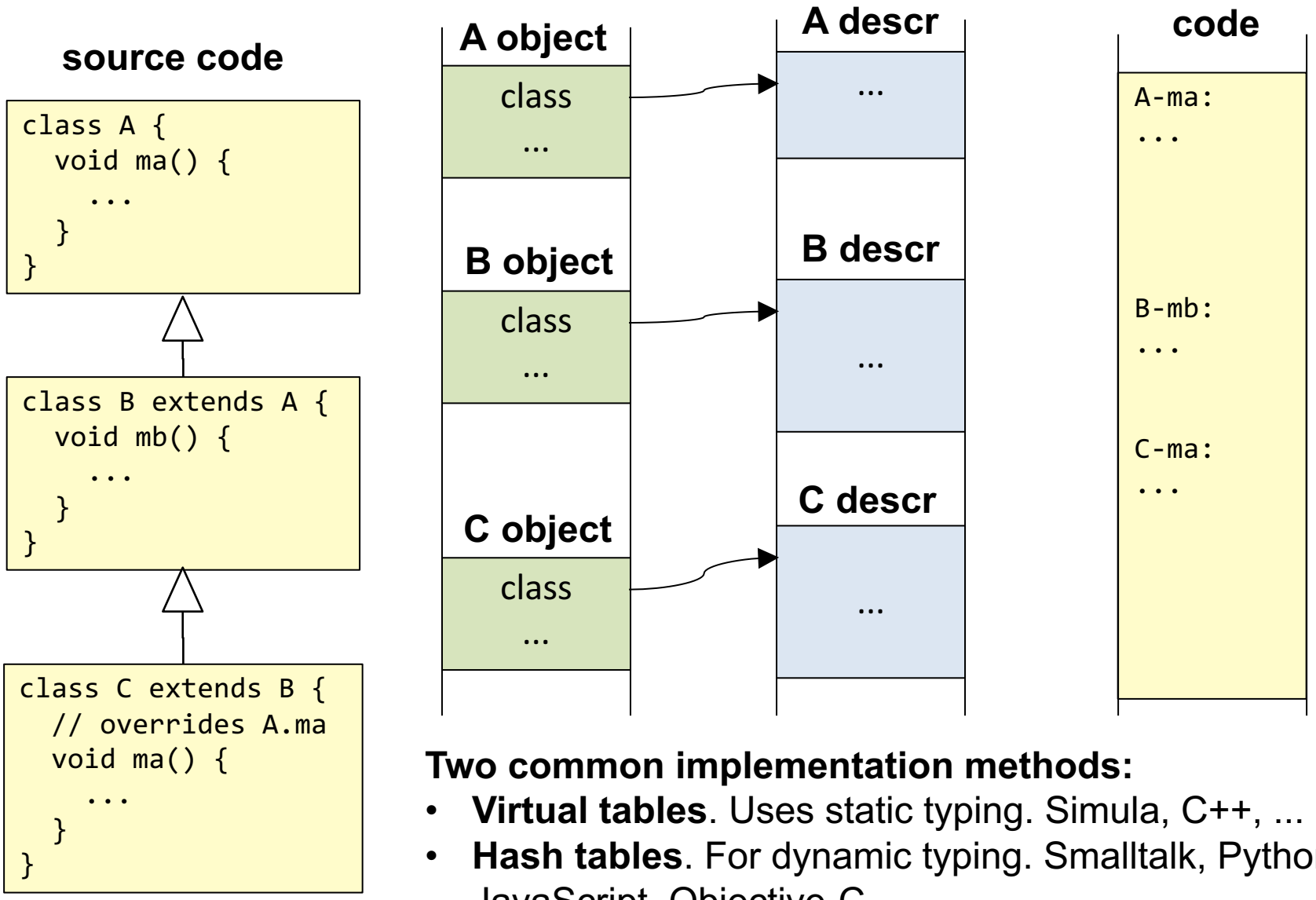
Parts of the class hiearchy are treated like single inheritance:
rA and rC point to the full C object.

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)

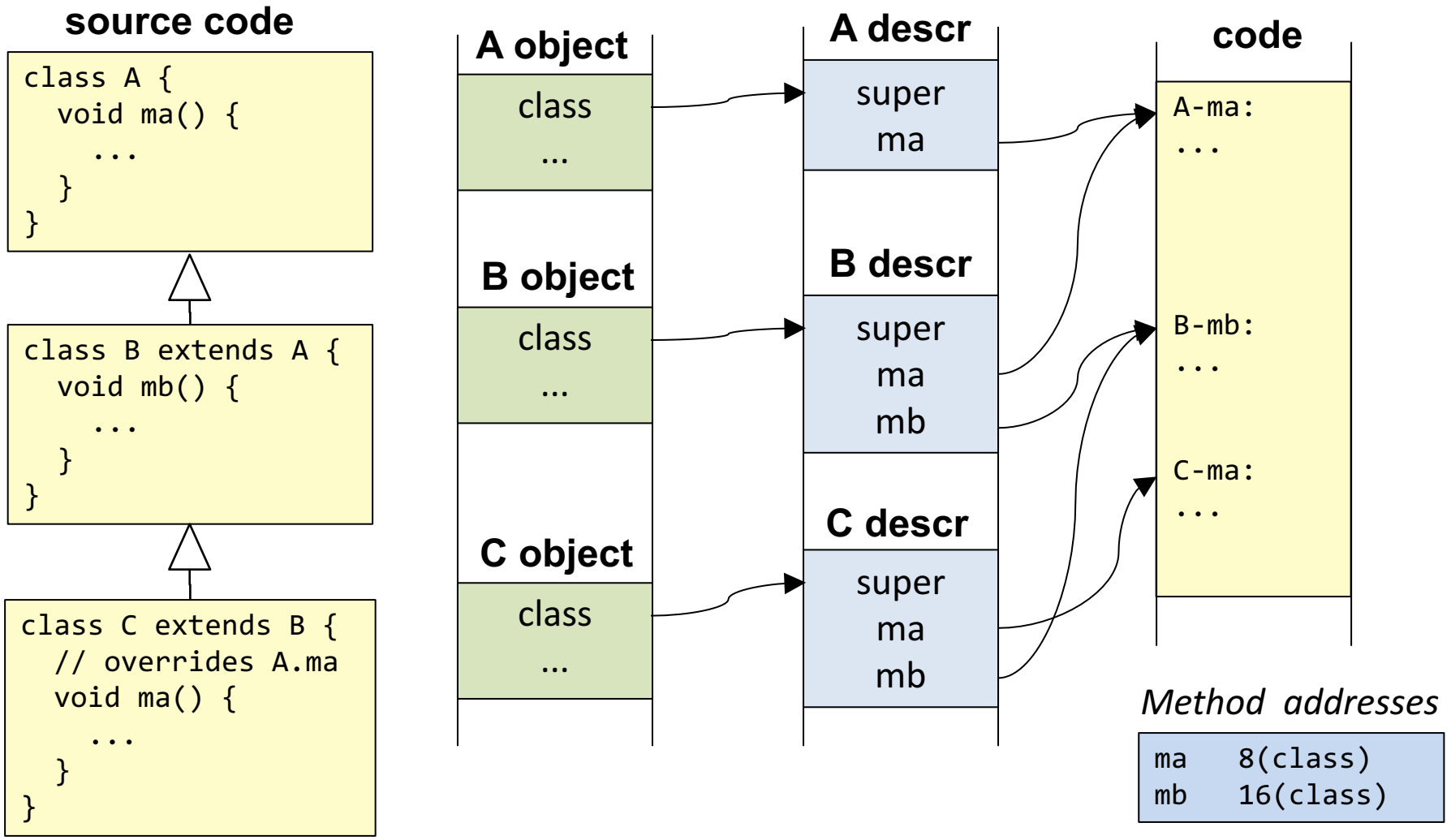


Two common implementation methods:

- **Virtual tables**. Uses static typing. Simula, C++, ...
- **Hash tables**. For dynamic typing. Smalltalk, Python, JavaScript, Objective-C, ...

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.

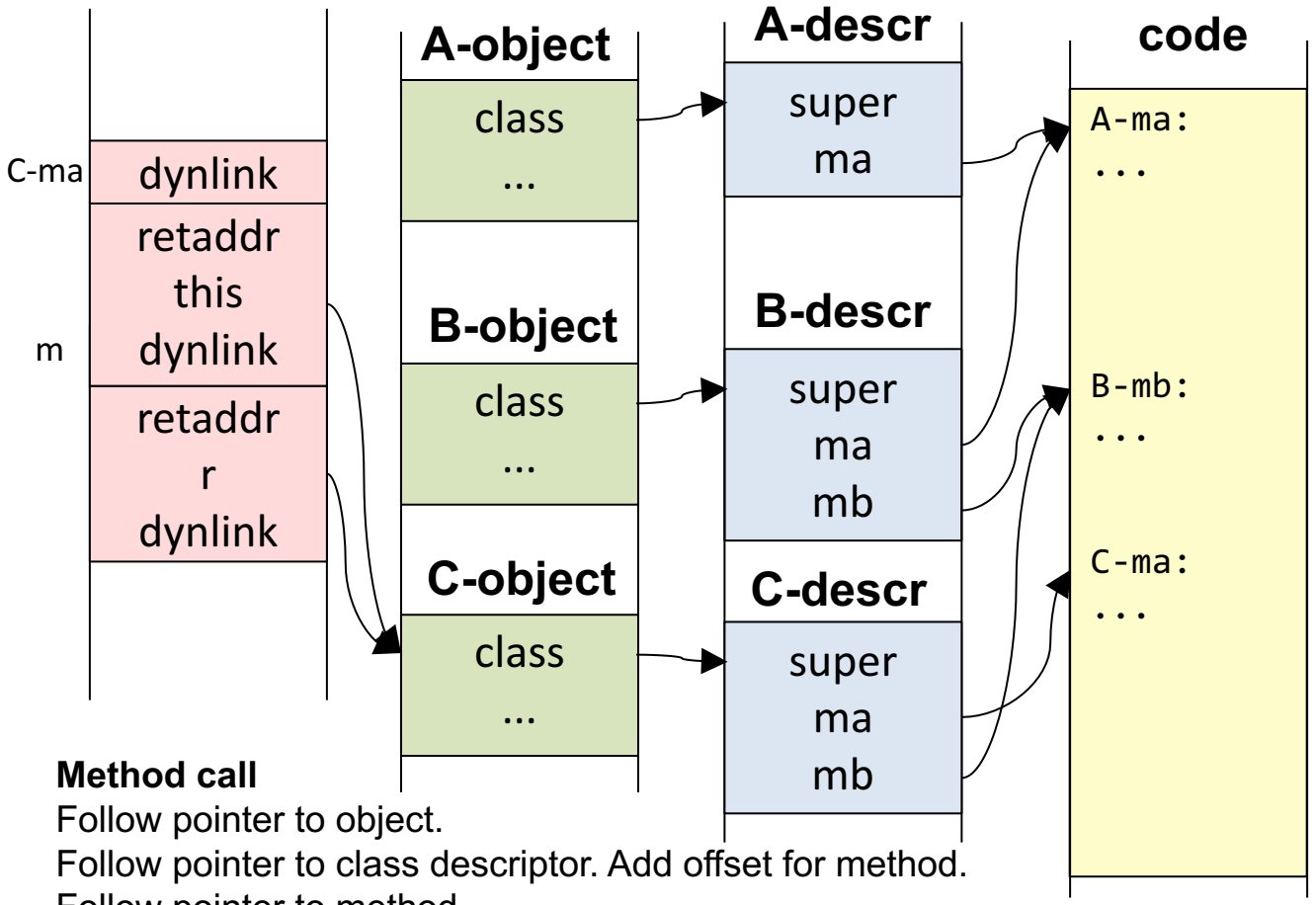
Calling a method via the virtual table

```
class A {
  void ma() {
    ...
  }
}
```

```
class B extends A {
  void mb() {
    ...
  }
}
```

```
class C extends B {
  // overrides A.ma
  void ma() {
    ...
  }
}
```

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



Method call
Follow pointer to object.
Follow pointer to class descriptor. Add offset for method.
Follow pointer to method.

```
m:
  ...
  movq 16(%rbp), %rax     # r -> rax
  pushq %rax              # push the static link (this)
  movq (%rax), %rax       # class descriptor -> rax
  callq 8(%rax)           # call ma
```

Hash table dynamic dispatch

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

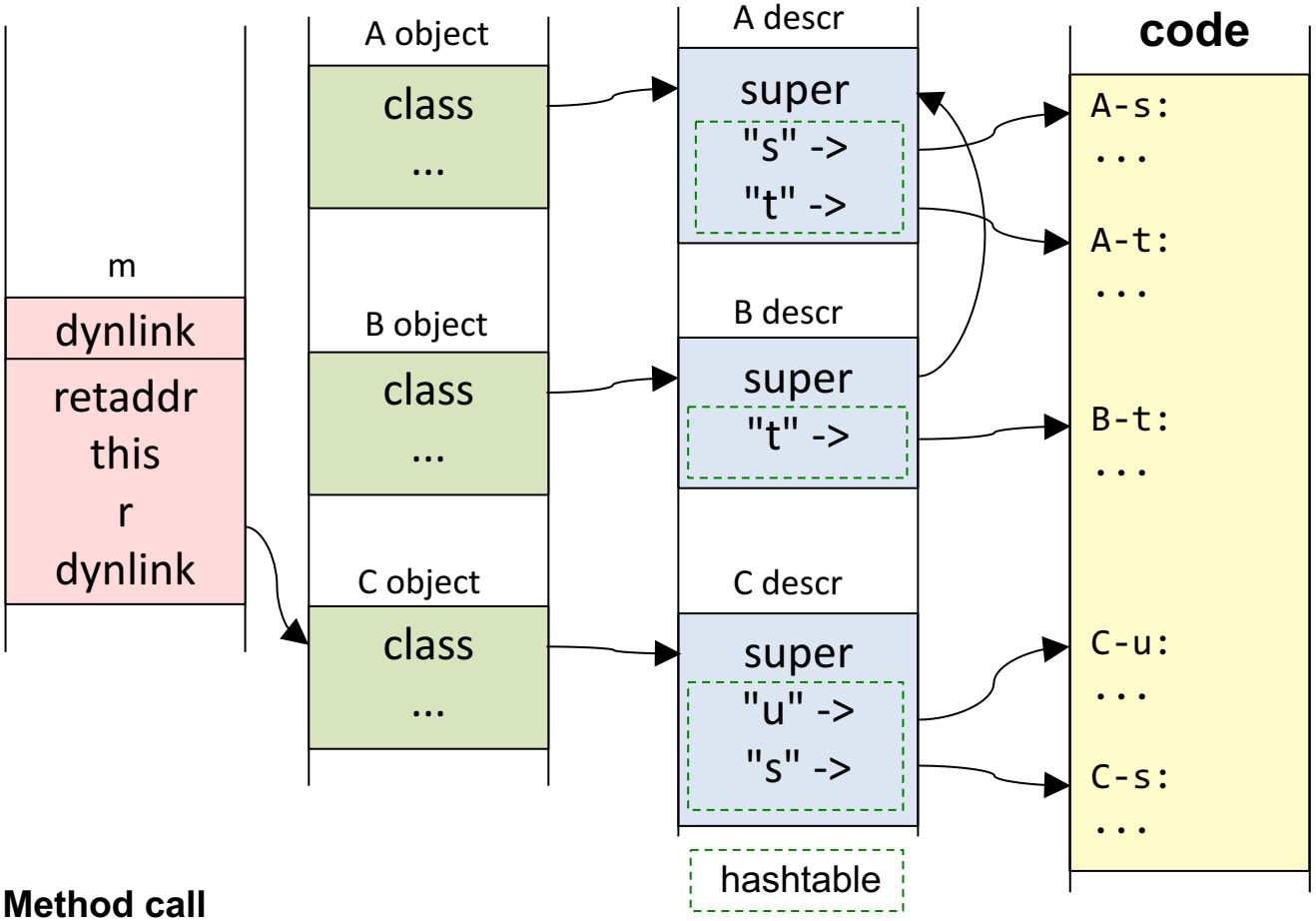
methods and vars have
no static types

```
class A {
  method s() {...}
  method t() {...}
}
```

```
class B extends A {
  method t() {...}
}
```

```
class C {
  method u() {...}
  method s() {...}
}
```

```
class ... {
  method m(r) {
    r.s();
  }
}
```



Method call

Follow pointer to object. Then to class descriptor.
Lookup method pointer in hashtable. If not found, go to super, lookup there...

Does not rely on static types.
Can be used for dynamically typed languages.
Slow if not optimized.

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):

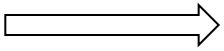
- common subexpression elimination
- constant propagation
- constant folding
- dead code elimination
- loop invariant code motion
- ...

Inlining (replace call by method body, get more code to optimize over)

Example local optimizations

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

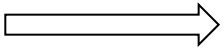
common subexpression elimination



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

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

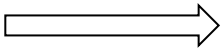
constant propagation



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

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

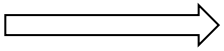
constant folding



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

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

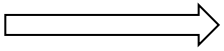
dead code elimination



```
return 42;
```

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

loop invariant code motion

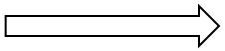


```
a = b + 3;
for (int i ...) {
  x[i] = a * i;
}
```

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.
Drawback: does not support dynamic loading.
Will be available as an option in Java 9.

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 mechanisms valuable to optimize 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: does not support dynamic loading.
But will be available as an option for Java 9.

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.

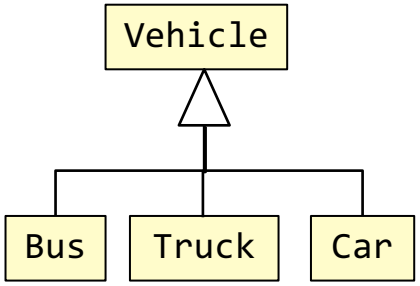
Original calling code

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

optimize →

Optimized calling code

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

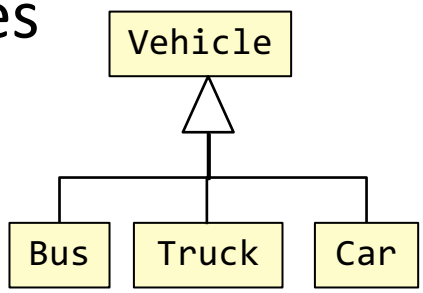


Called method:

```
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

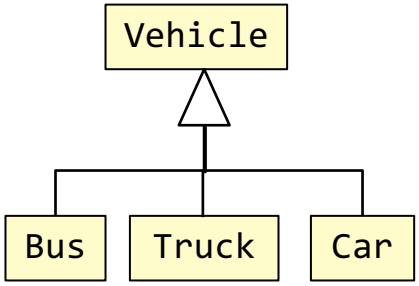
```
Vehicle v = ...;
while (...) {
  v = aList.get();
  if (v is a Bus)
    Bus-m(v)
  else if (v is a Car)
    Car-m(v)
  else
    v.m(); // normal lookup
}
```

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 →

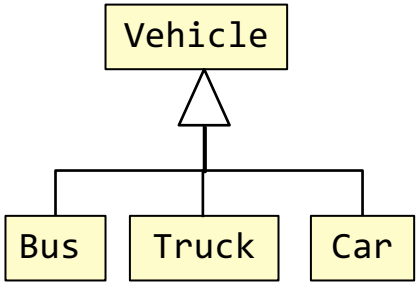
with inlined methods

```
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.

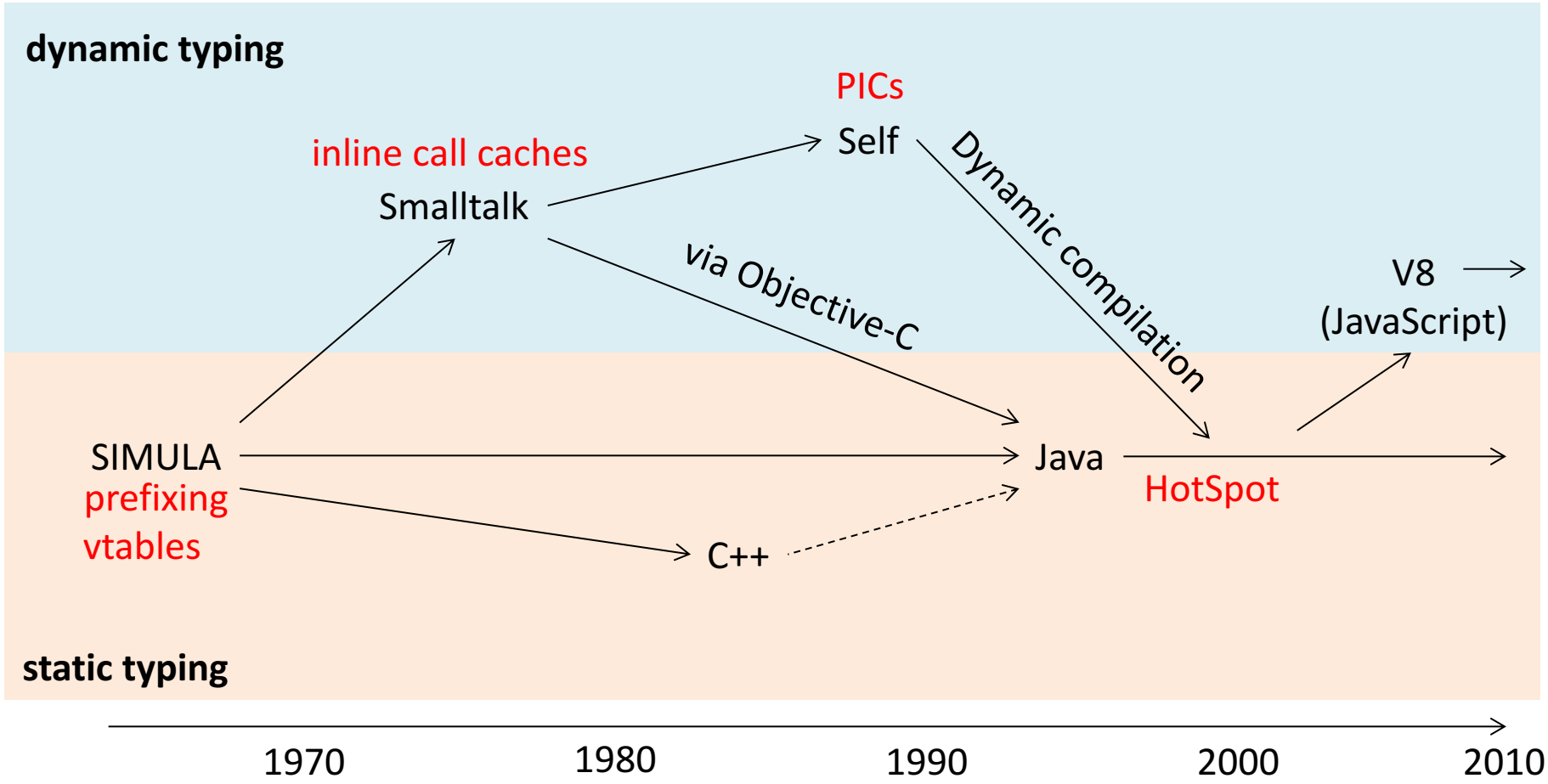
Warm-up vs. Steady state

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?