

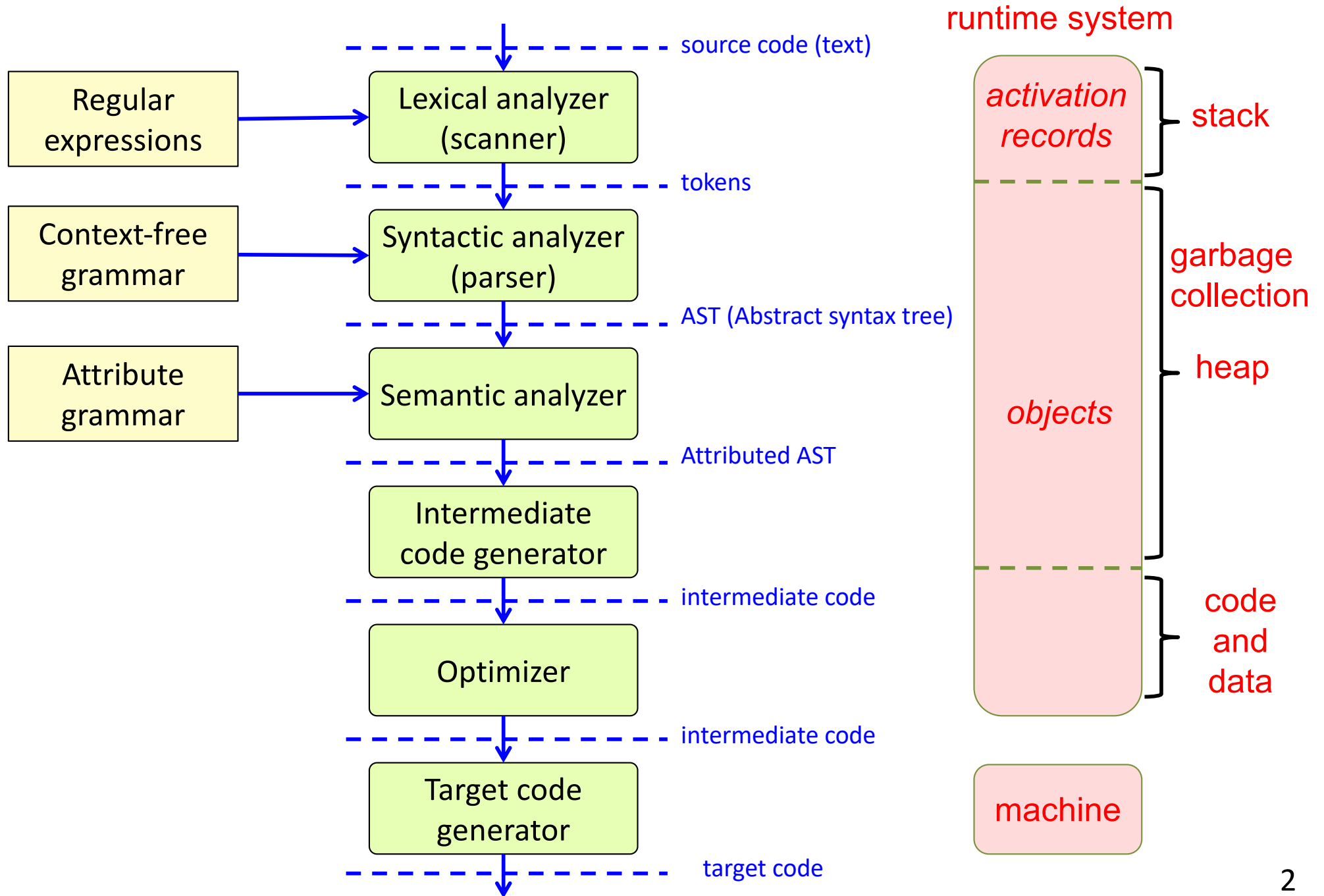
EDAN65: Compilers, Lecture 12

Runtime systems for object-oriented languages

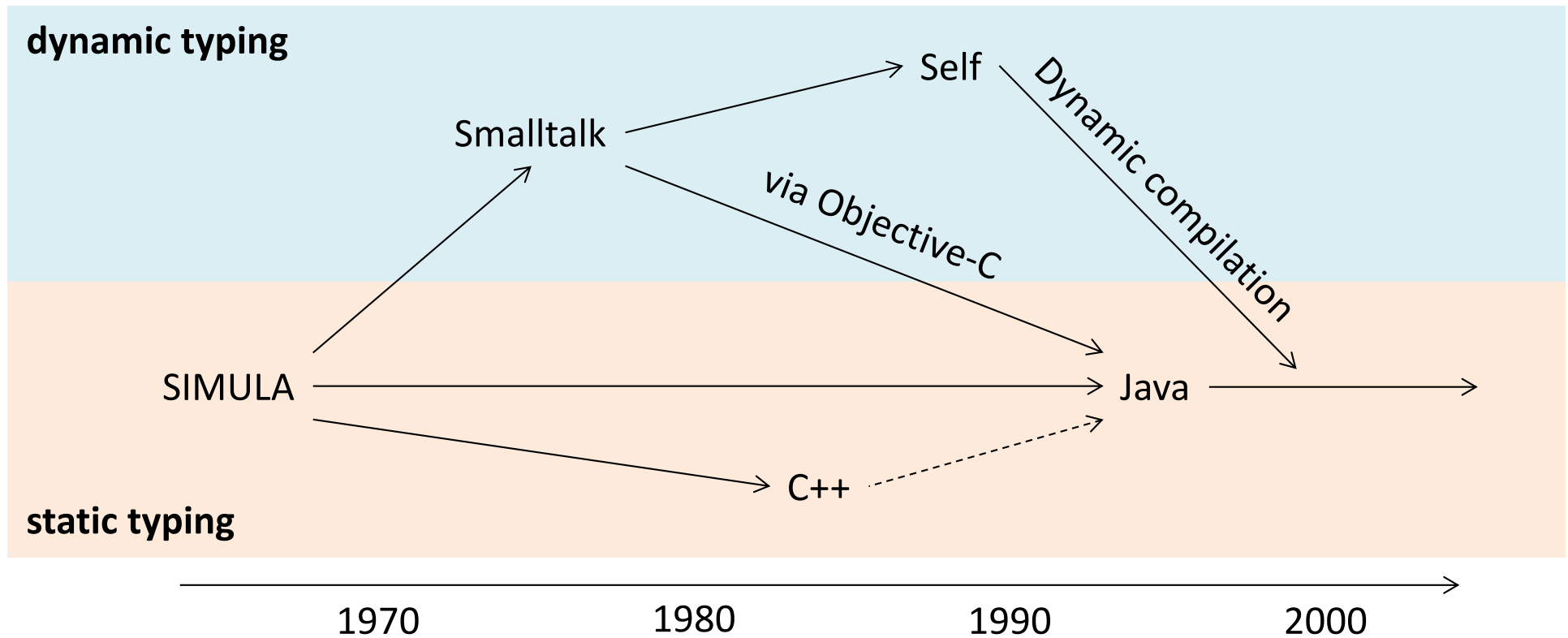
Görel Hedin

Revised: 2023-10-02

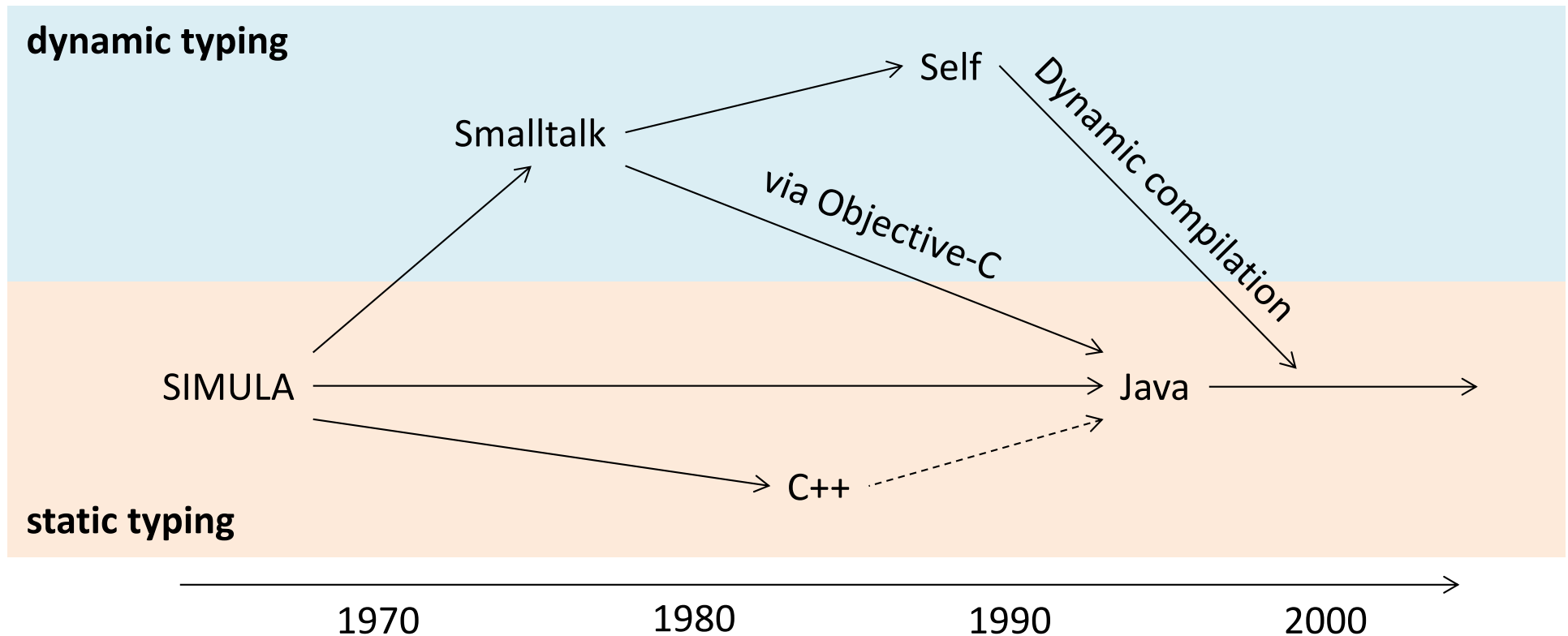
This lecture



Some influential OO languages



Some influential OO languages



Type

A *type* is a set of values or objects

At runtime, every object has a type (the class used for creating the object).

Dynamic typing

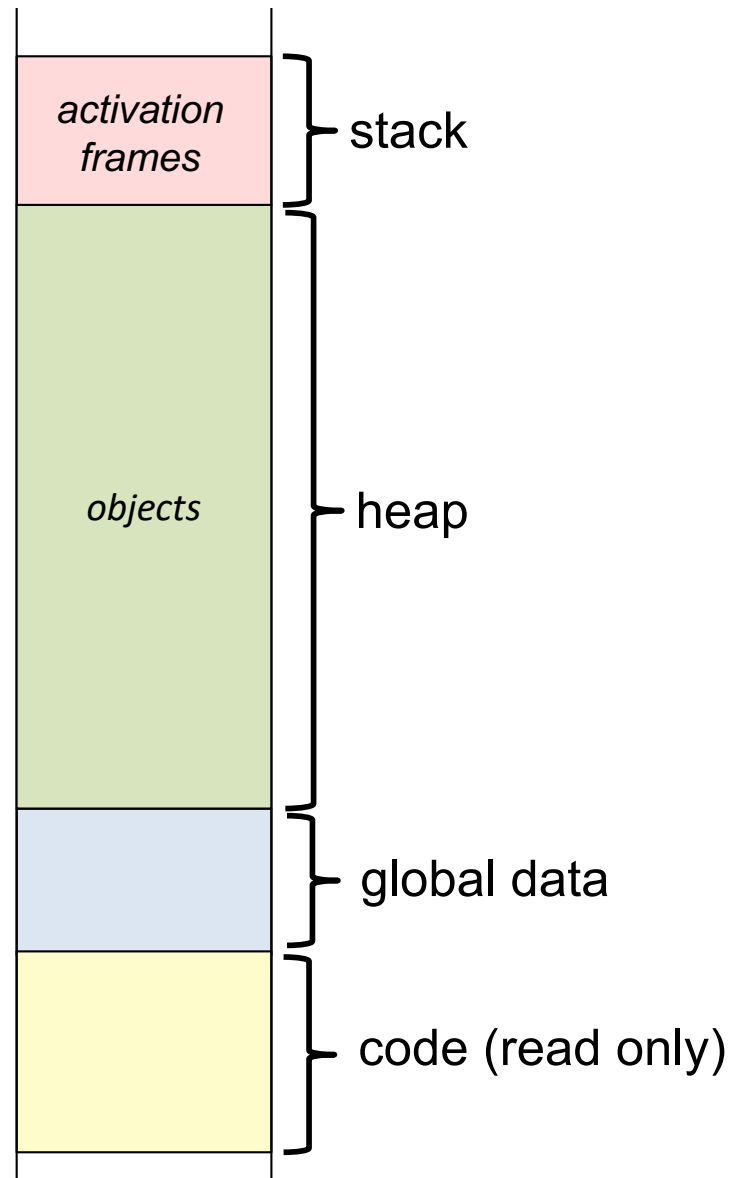
There are no types for things at compile-time. Only for objects at runtime.

Static typing

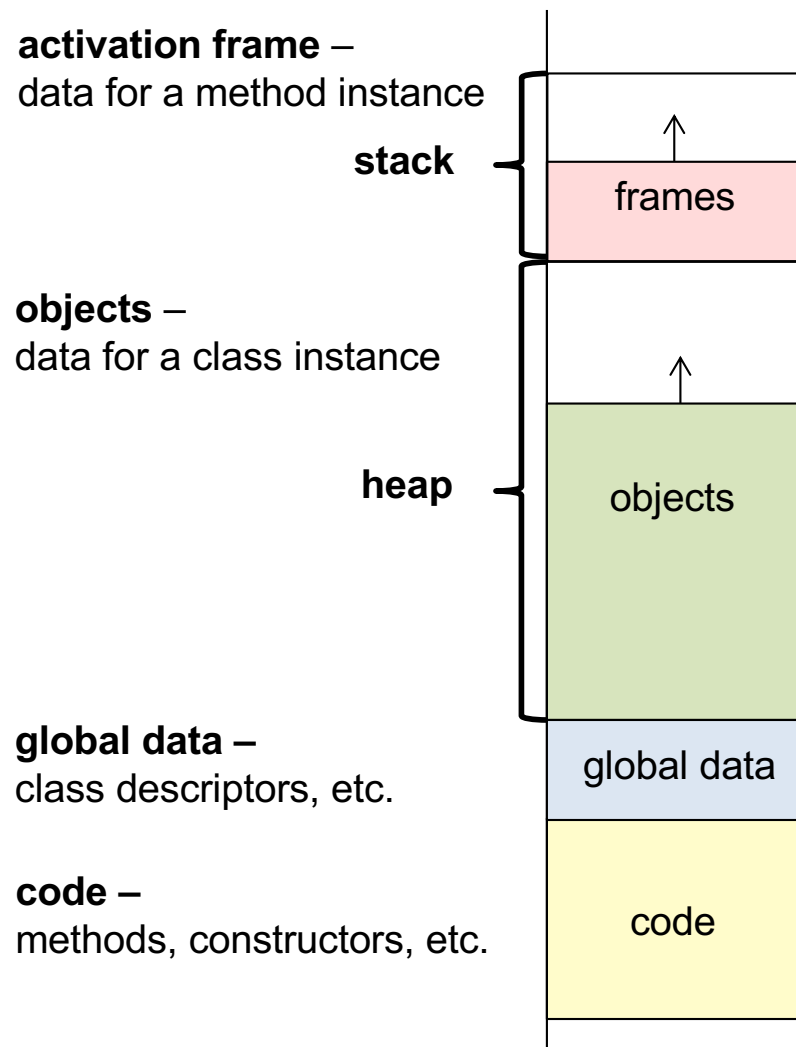
Variables have types at compile-time.

At runtime, the variable points to an object of *at least* that type (that type, or a subtype).

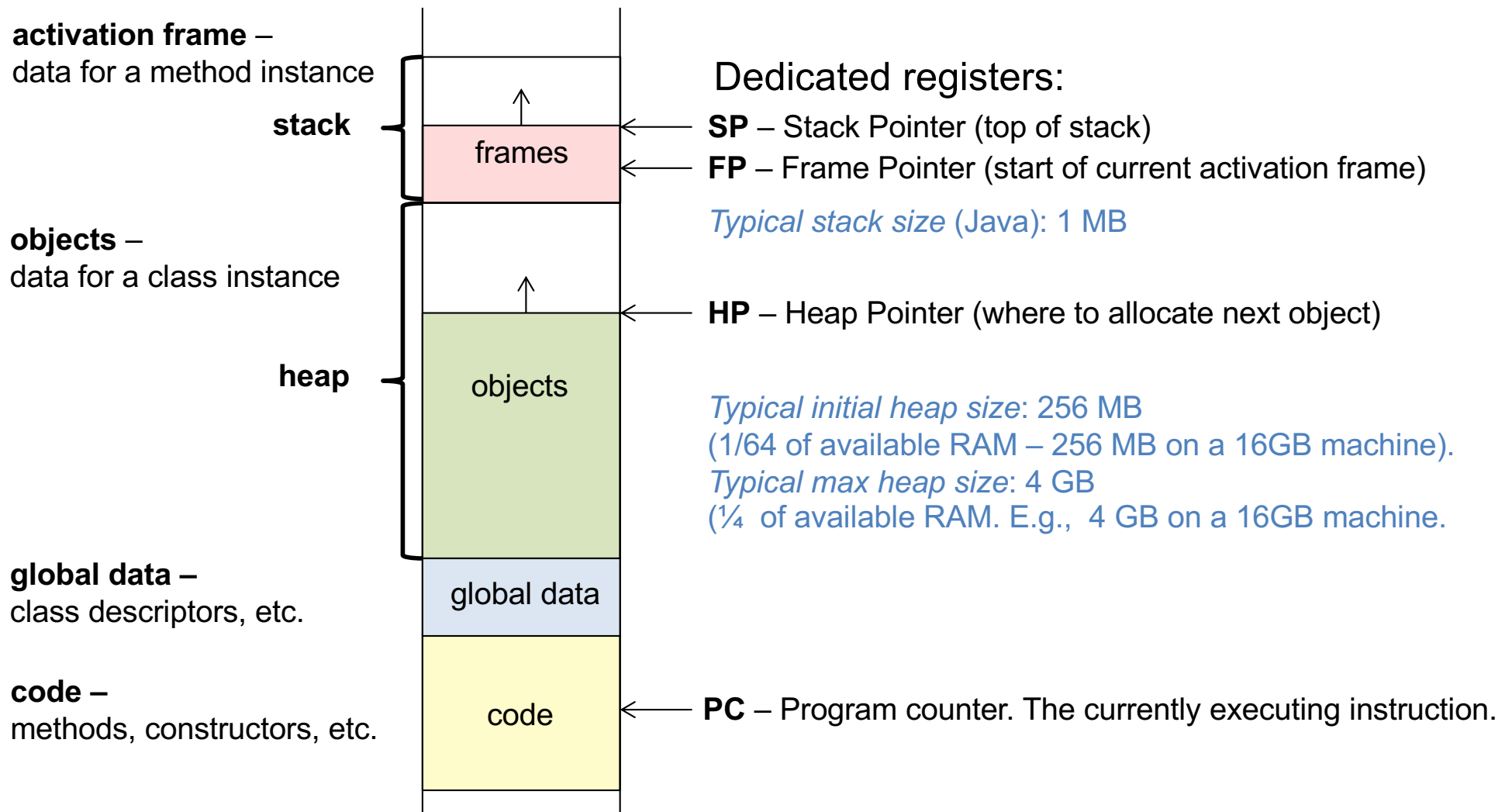
Example memory segments



Typical memory usage for OO languages



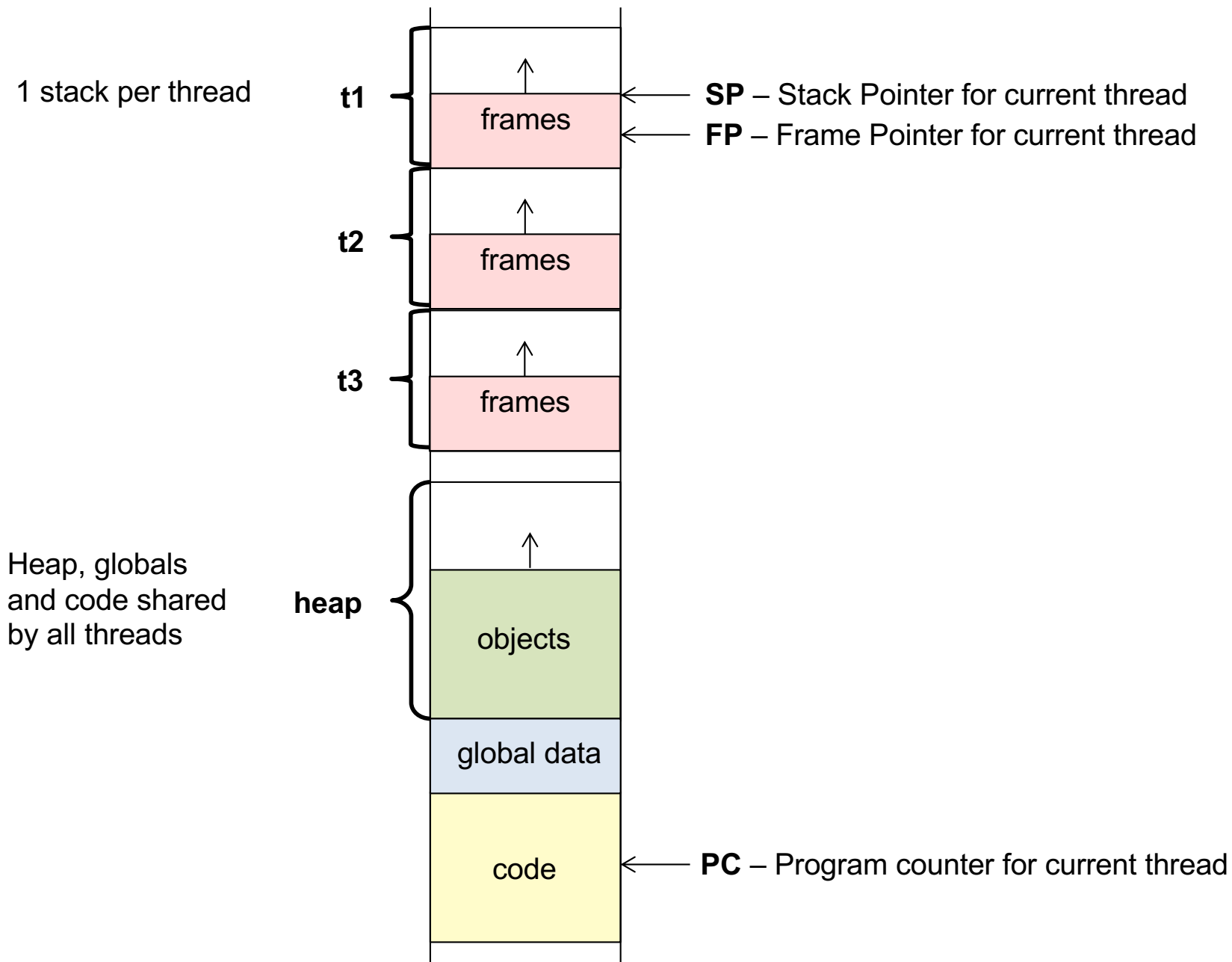
Typical memory usage for OO languages



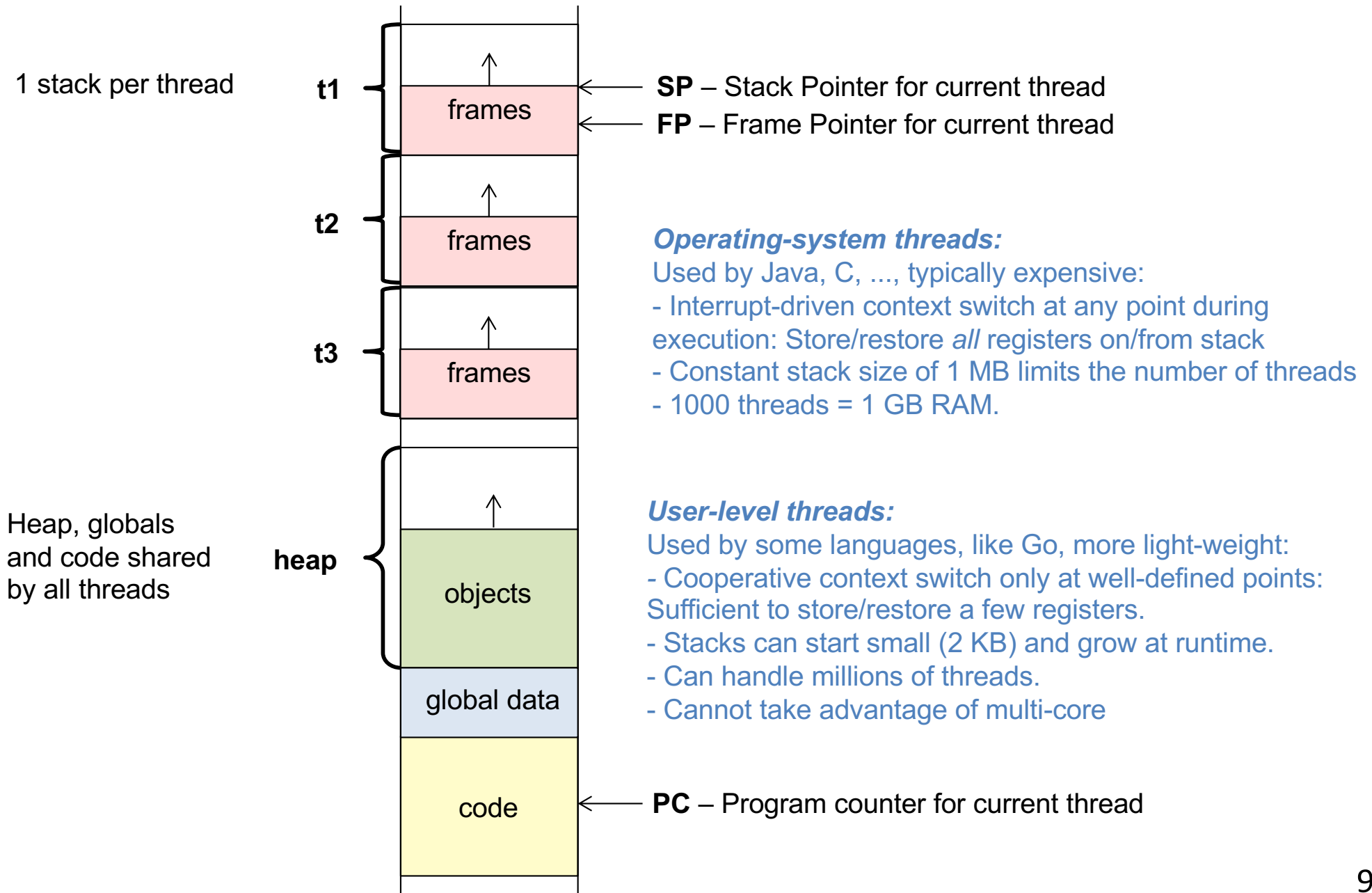
Dynamic class loading (like in Java):

*Class descriptors and JIT-compiled bytecode placed on the heap instead of in the data/code segments.
(The code for the jvm itself is placed in the code segment)*

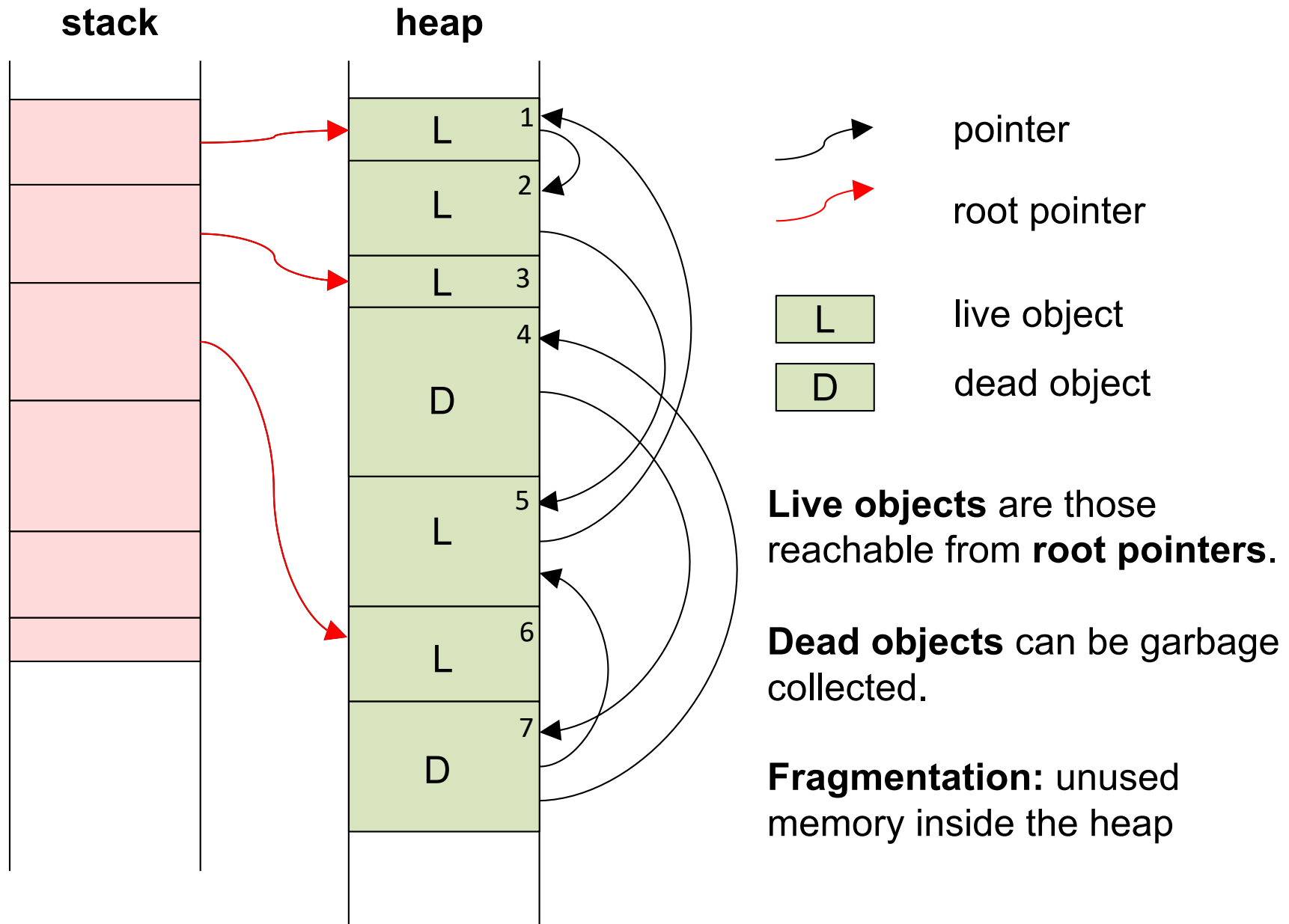
Handling threads



Handling threads



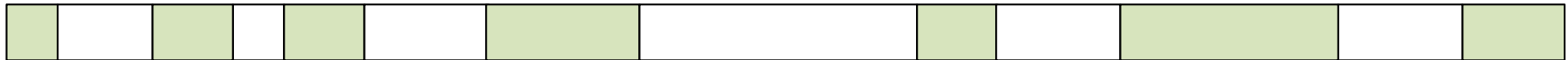
The heap



Major garbage collection techniques

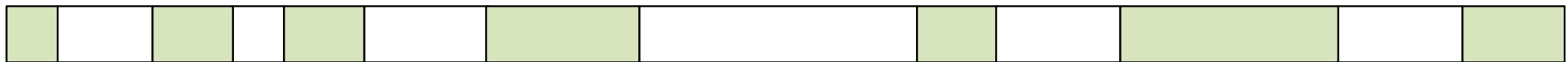
Major garbage collection techniques

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

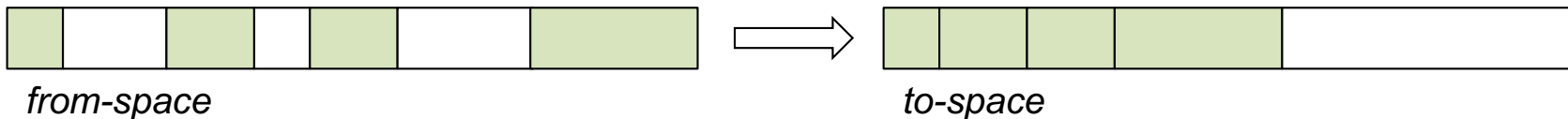


Major garbage collection techniques

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

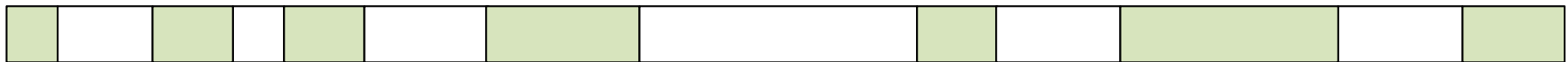


Copying GC: Divide heap into two spaces. Allocate new objects in *from-space*. When full, move all live objects to *to-space*. Flip *from-space* and *to-space*.

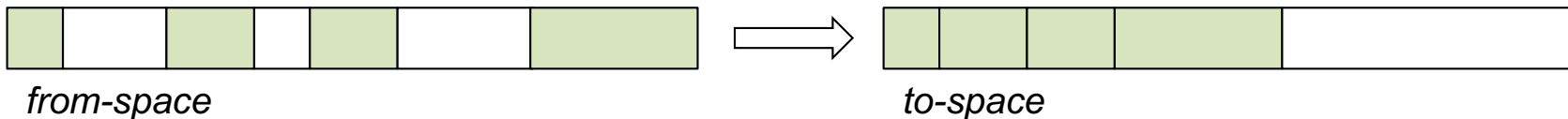


Major garbage collection techniques

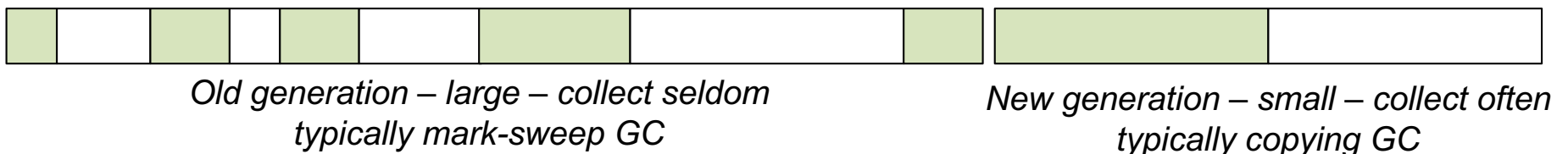
Mark-sweep GC: Follow all pointers and mark all live objects. Sweep heap and collect free objects. Or **compact** the heap to avoid fragmentation.



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

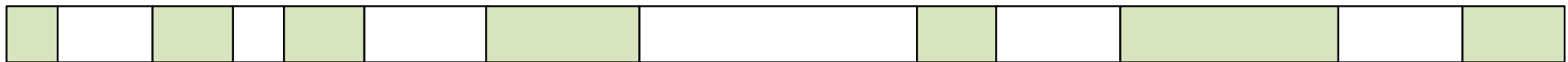


Generational GC: Efficient because most objects die young. Move (tenure) surviving objects to older generation.

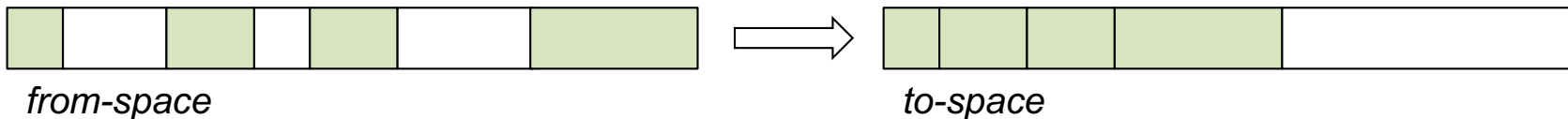


Major garbage collection techniques

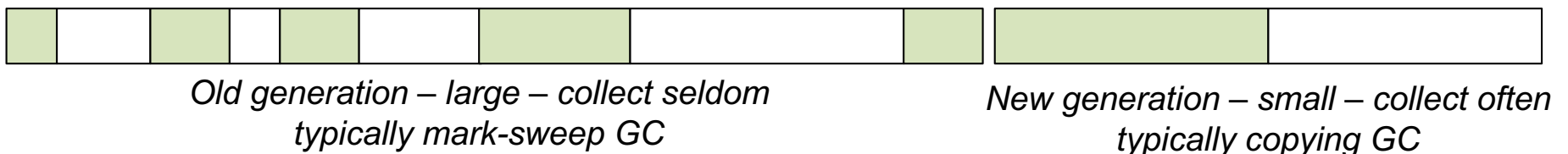
Mark-sweep GC: Follow all pointers and mark all live objects. Sweep heap and collect free objects. Or **compact** the heap to avoid fragmentation.



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



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

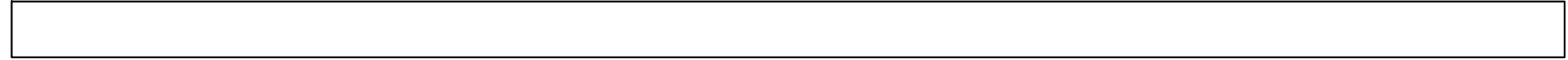


Reference counting: Inefficient (overhead when reading and writing references). Deallocate when count=0. Does not handle cycles. Fragmentation problems.

Mark-Sweep GC

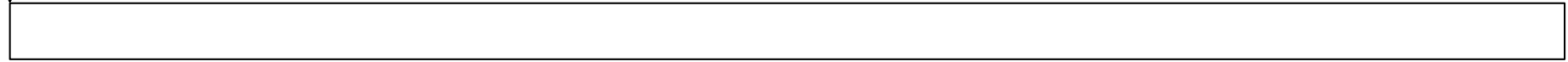
Mark-Sweep-Compact GC

HP 1. The program starts. The heap is empty.



Mark-Sweep-Compact GC

HP 1. The program starts. The heap is empty.

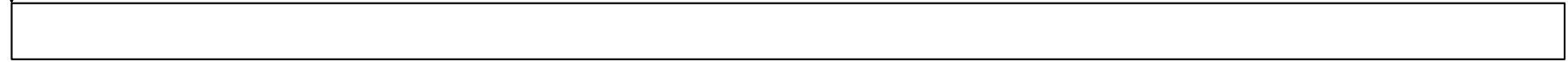


2. The program runs and allocates new objects until the heap is full. Program stops. HP



Mark-Sweep-Compact GC

HP 1. The program starts. The heap is empty.



2. The program runs and allocates new objects until the heap is full. Program stops. HP

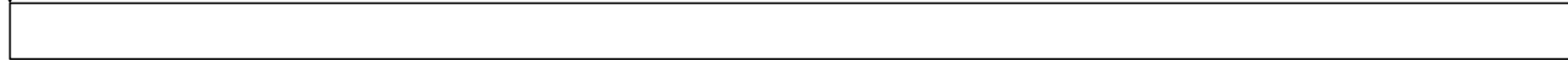


3. Mark phase: The GC follows pointers from the roots and marks all live objects. HP



Mark-Sweep-Compact GC

HP 1. The program starts. The heap is empty.



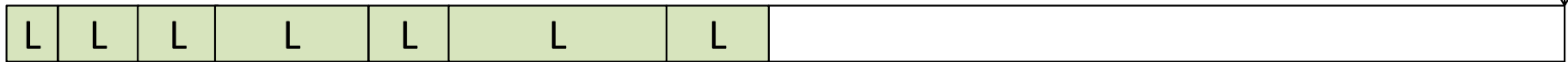
2. The program runs and allocates new objects until the heap is full. Program stops. HP



3. Mark phase: The GC follows pointers from the roots and marks all live objects. HP

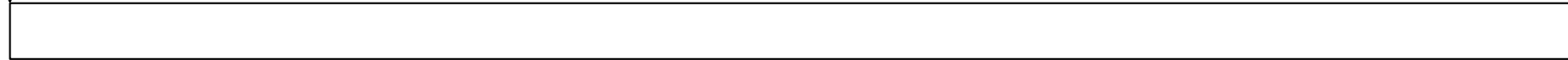


4. Sweep and compact phase: The GC scans the heap to compact live objects. HP



Mark-Sweep-Compact GC

HP 1. The program starts. The heap is empty.



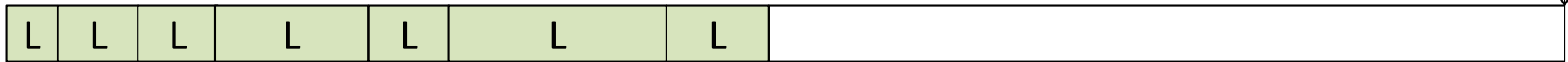
2. The program runs and allocates new objects until the heap is full. Program stops. HP



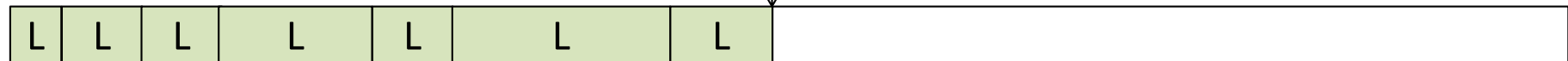
3. Mark phase: The GC follows pointers from the roots and marks all live objects. HP



4. Sweep and compact phase: The GC scans the heap to compact live objects. HP

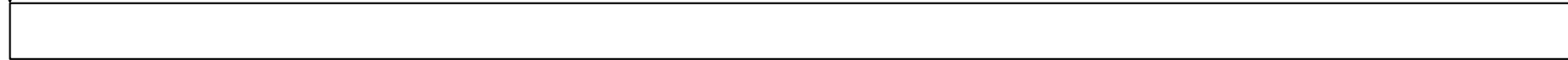


5. The program continues to run. HP



Mark-Sweep-Compact GC

HP 1. The program starts. The heap is empty.



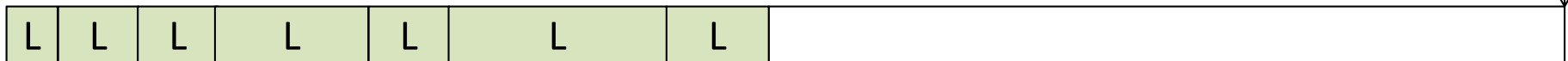
2. The program runs and allocates new objects until the heap is full. Program stops. HP



3. Mark phase: The GC follows pointers from the roots and marks all live objects. HP



4. Sweep and compact phase: The GC scans the heap to compact live objects. HP



5. The program continues to run. HP

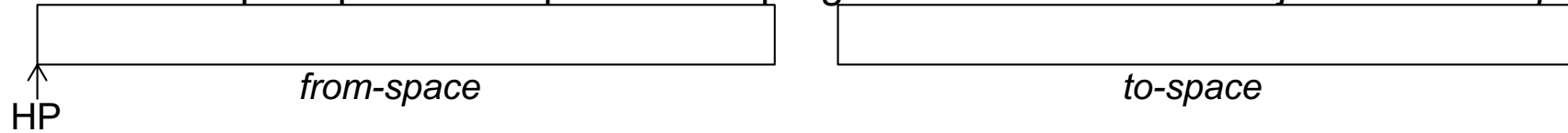


- + Avoids fragmentation.
- Long pause at GC ("stop the world")

Copying GC

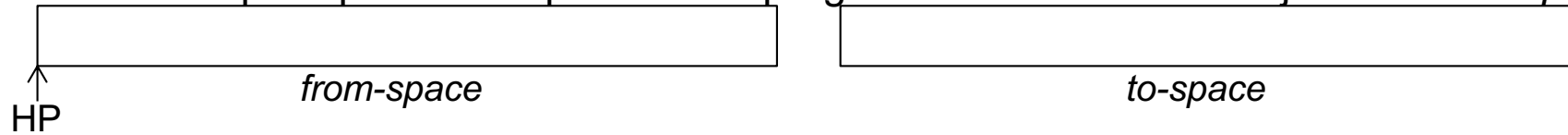
Copying GC

1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.

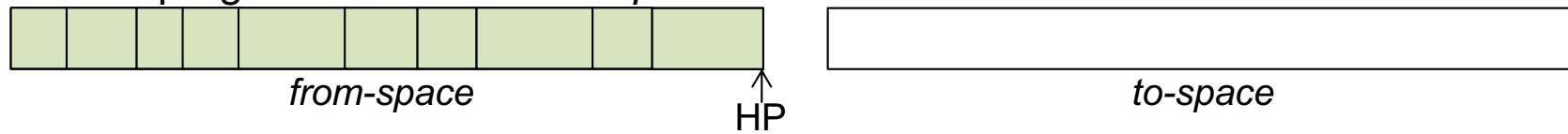


Copying GC

1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.

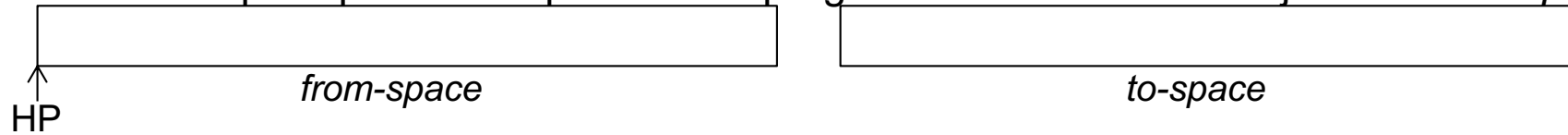


2. The program runs until *from-space* is full.

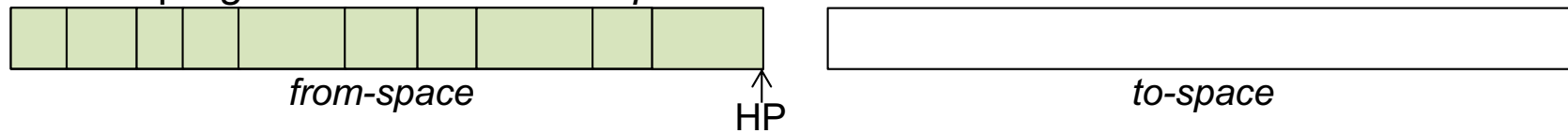


Copying GC

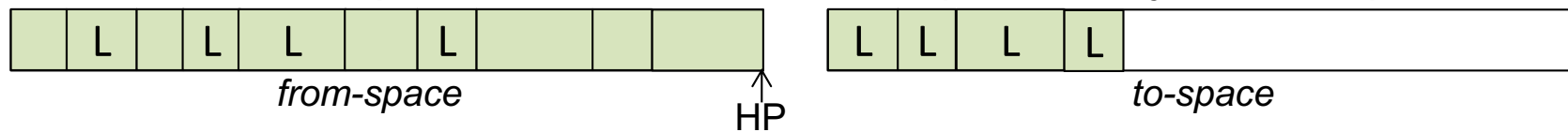
1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.



2. The program runs until *from-space* is full.

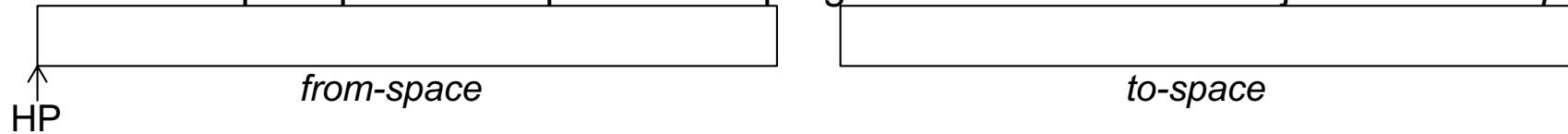


3. The GC follows pointers from the roots and copies live objects to *to-space*.

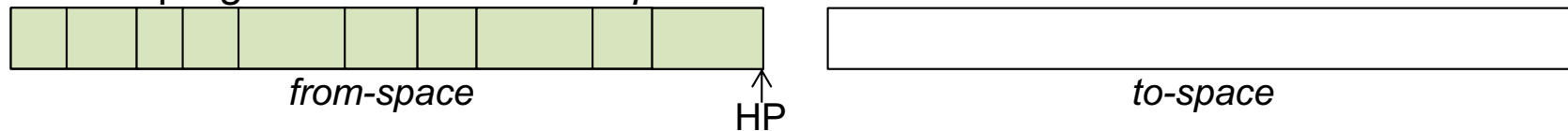


Copying GC

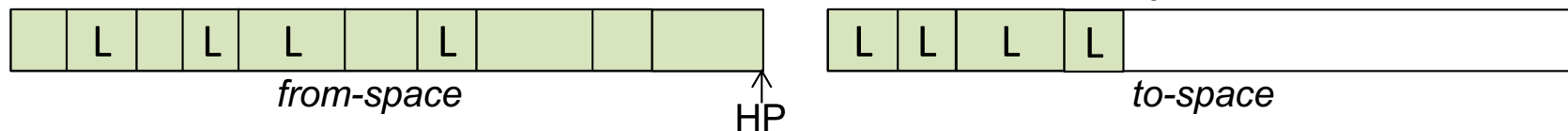
1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.



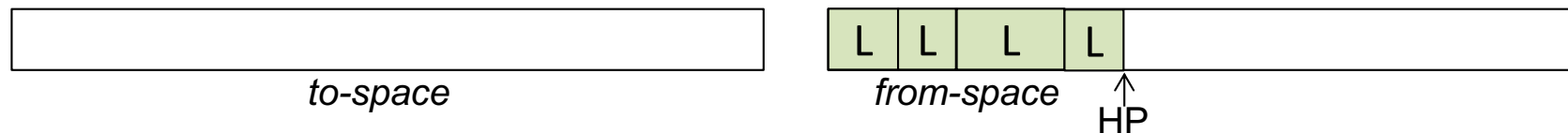
2. The program runs until *from-space* is full.



3. The GC follows pointers from the roots and copies live objects to *to-space*.

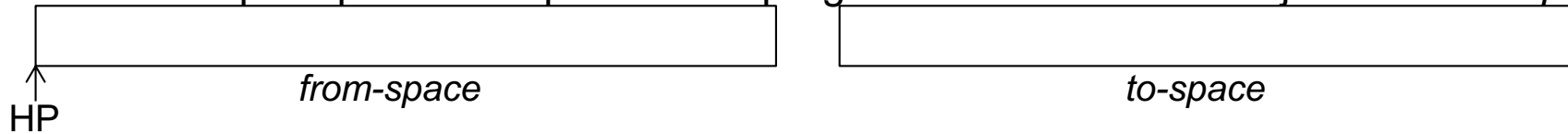


4. Flip *to-space* and *from-space*

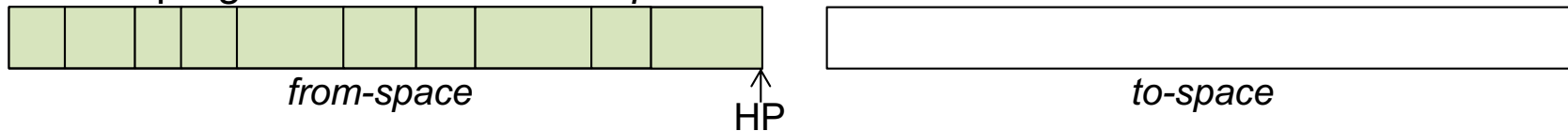


Copying GC

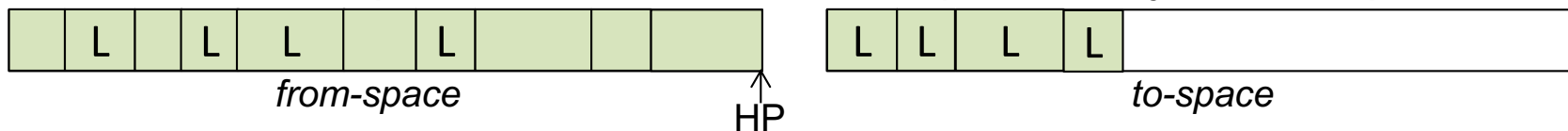
1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.



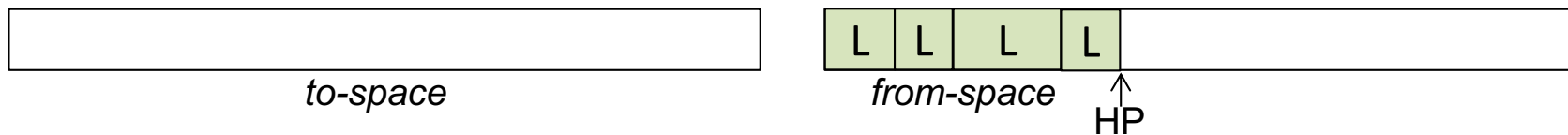
2. The program runs until *from-space* is full.



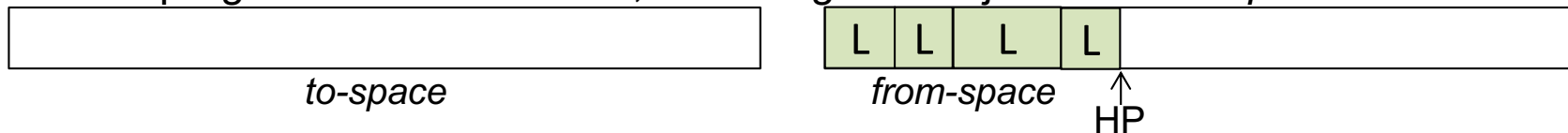
3. The GC follows pointers from the roots and copies live objects to *to-space*.



4. Flip *to-space* and *from-space*

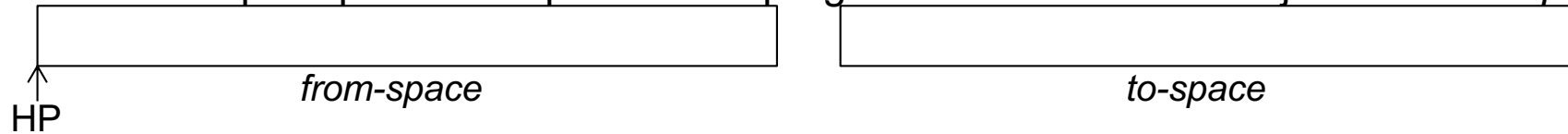


5. The program continues to run, allocating new objects in *from-space*.

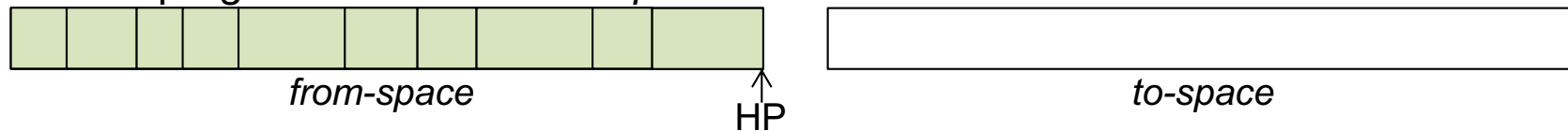


Copying GC

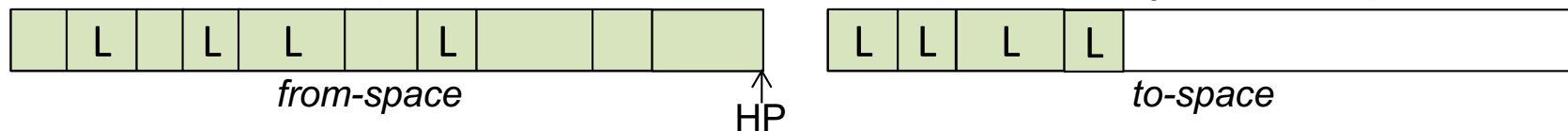
1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.



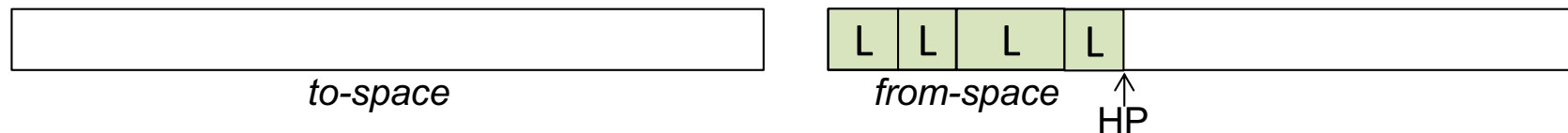
2. The program runs until *from-space* is full.



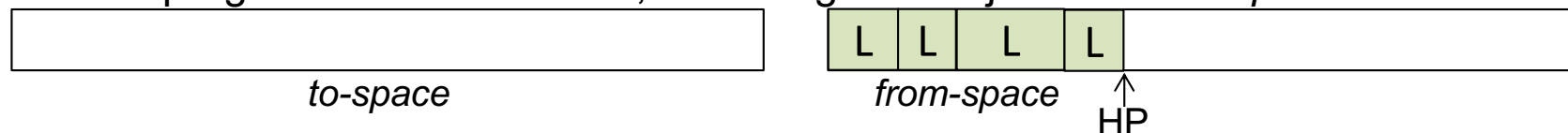
3. The GC follows pointers from the roots and copies live objects to *to-space*.



4. Flip *to-space* and *from-space*



5. The program continues to run, allocating new objects in *from-space*.



- + Efficient – looks only at live objects (which are few). Avoids fragmentation.
- Fairly long pauses at GC ("stop the world"). Uses twice the amount of memory.

Generational GC

Generational GC

1. The heap is split into a large old and a small new generation.



old generation



new generation

Generational GC

1. The heap is split into a large old and a small new generation.



old generation

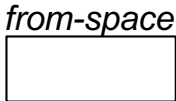


new generation

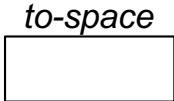
2. Typical algorithms.



old generation, mark-sweep



from-space

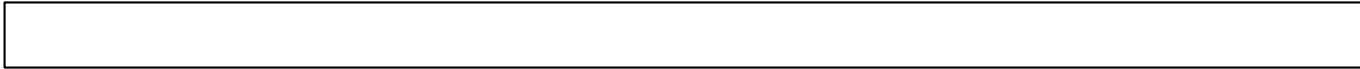


to-space

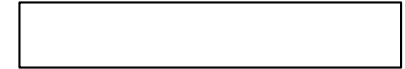
*new generation,
copying*

Generational GC

1. The heap is split into a large old and a small new generation.

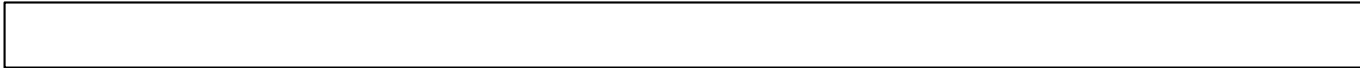


old generation

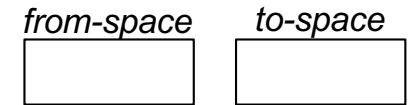


new generation

2. Typical algorithms.



old generation, mark-sweep



new generation, copying

3. The program runs, allocating objects in from-space in new gen.



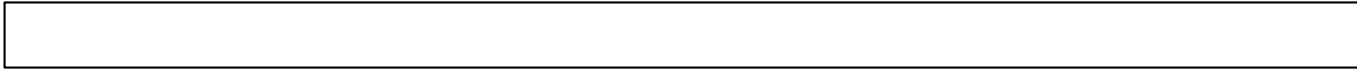
old generation



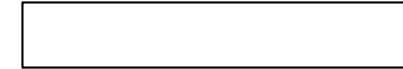
new generation

Generational GC

1. The heap is split into a large old and a small new generation.



old generation

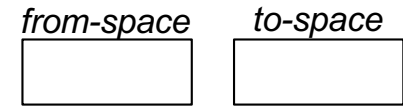


new generation

2. Typical algorithms.

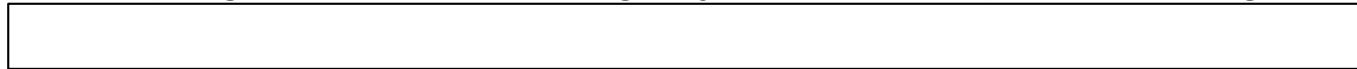


old generation, mark-sweep



new generation, copying

3. The program runs, allocating objects in from-space in new gen.



old generation

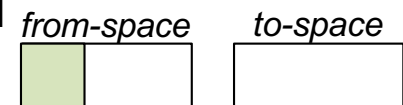


new generation

4. Objects surviving a few GCs in new gen are *tenured* – moved to old



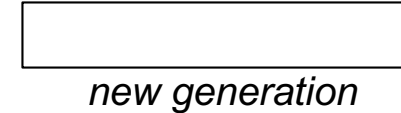
old generation



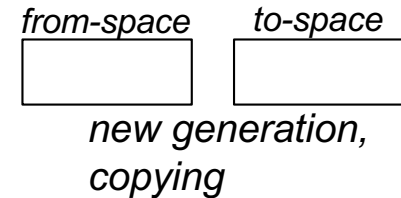
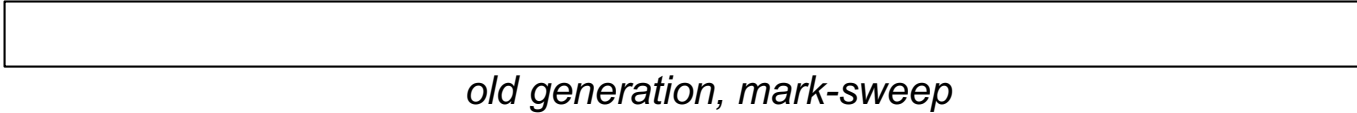
new generation

Generational GC

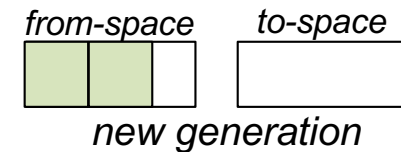
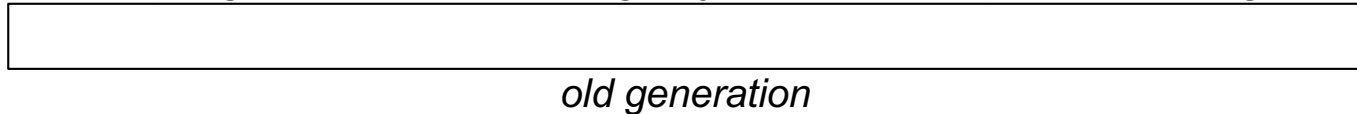
1. The heap is split into a large old and a small new generation.



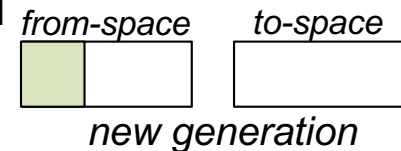
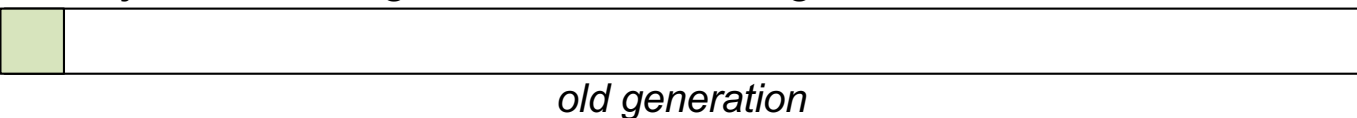
2. Typical algorithms.



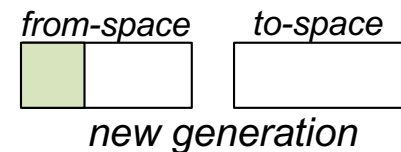
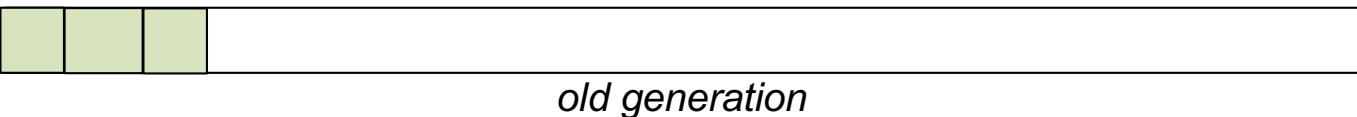
3. The program runs, allocating objects in from-space in new gen.



4. Objects surviving a few GCs in new gen are *tenured* – moved to old

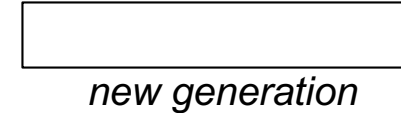


5. Most objects die very young. Few survive to be tenured.

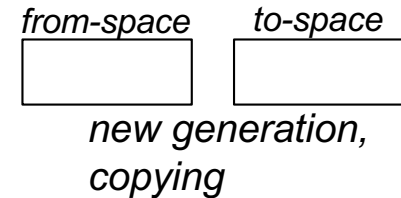
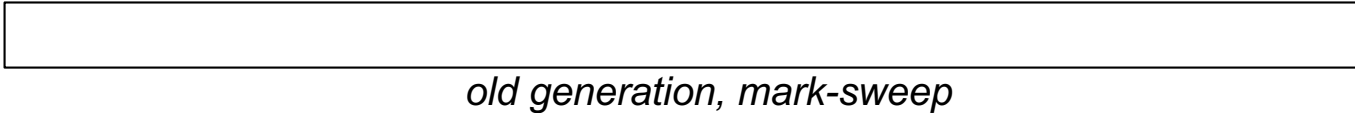


Generational GC

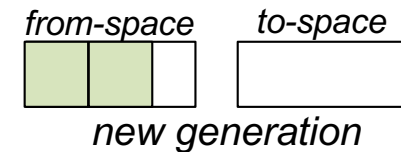
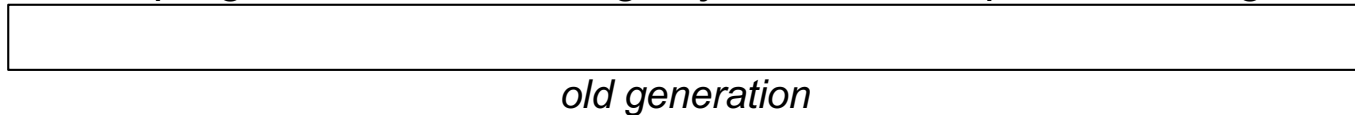
1. The heap is split into a large old and a small new generation.



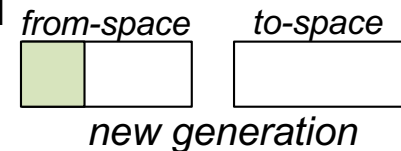
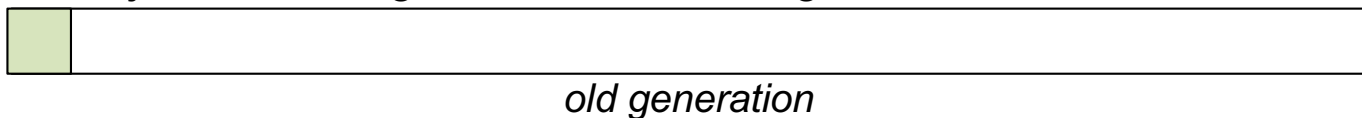
2. Typical algorithms.



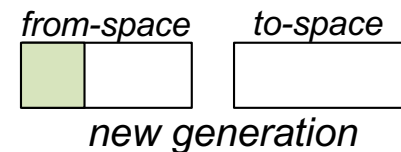
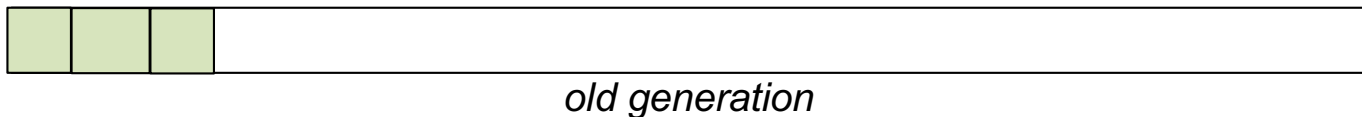
3. The program runs, allocating objects in from-space in new gen.



4. Objects surviving a few GCs in new gen are *tenured* – moved to old



5. Most objects die very young. Few survive to be tenured.



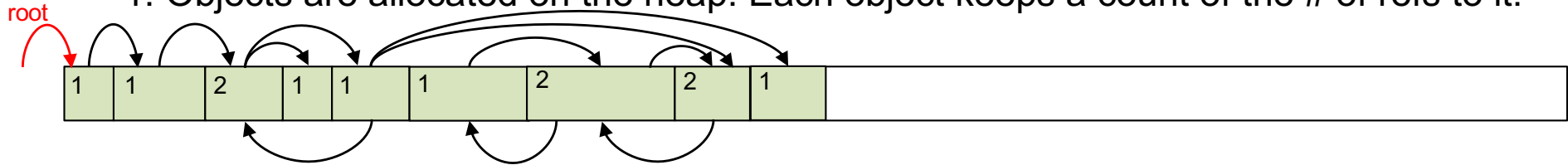
+ Efficient. Short pauses. GC in new generation is quick (small area).

+ Old generation grows very slowly. Avoids fragmentation.

Reference-counting GC

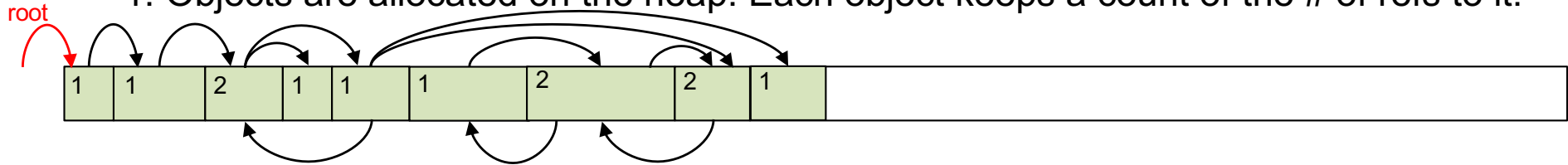
Reference-counting GC

1. Objects are allocated on the heap. Each object keeps a count of the # of refs to it.

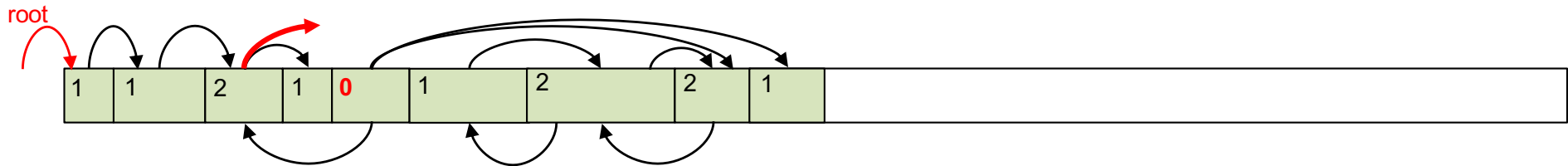


Reference-counting GC

1. Objects are allocated on the heap. Each object keeps a count of the # of refs to it.

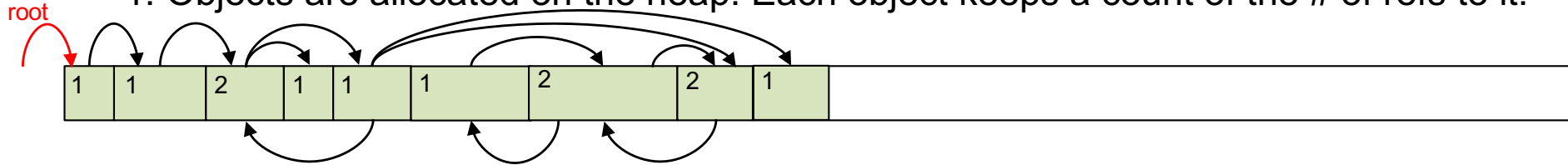


2. When a reference is changed, counts are updated.

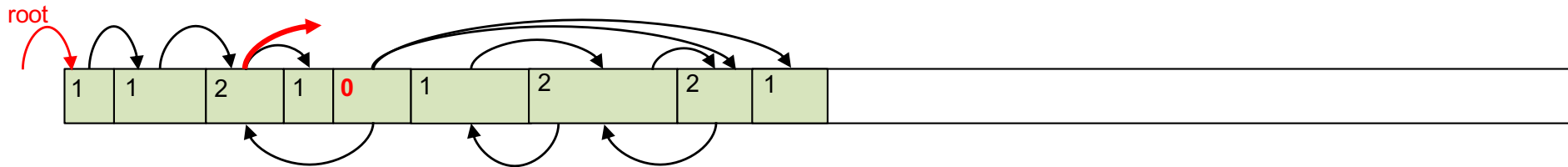


Reference-counting GC

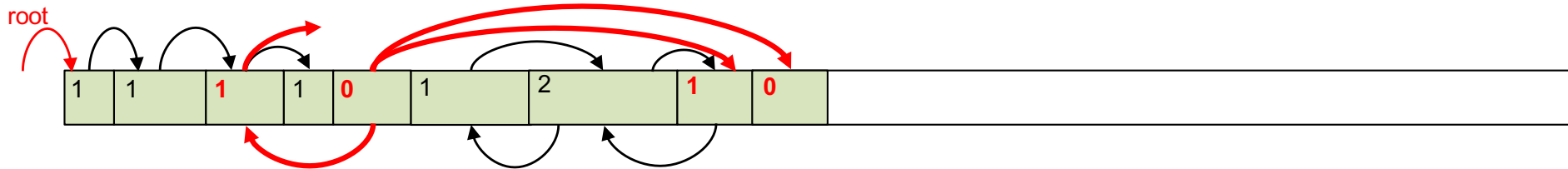
1. Objects are allocated on the heap. Each object keeps a count of the # of refs to it.



2. When a reference is changed, counts are updated.

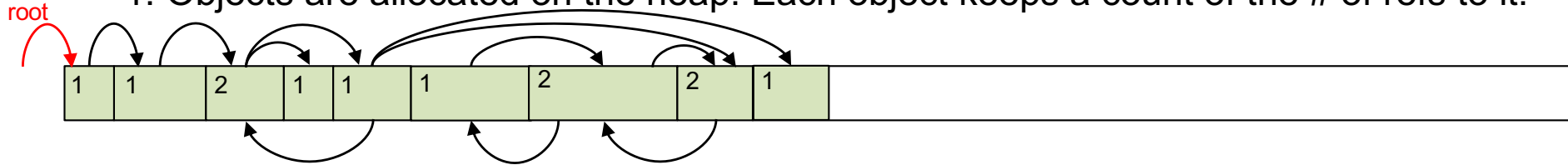


3. When a count goes to zero, the object is deallocated. Its references are followed, and counts are decremented, and may go to zero. The process continues recursively.

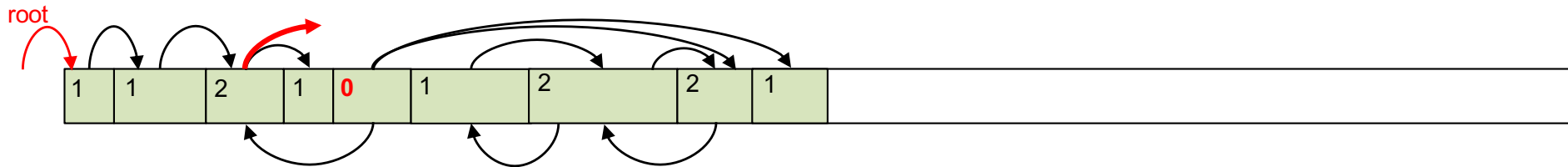


Reference-counting GC

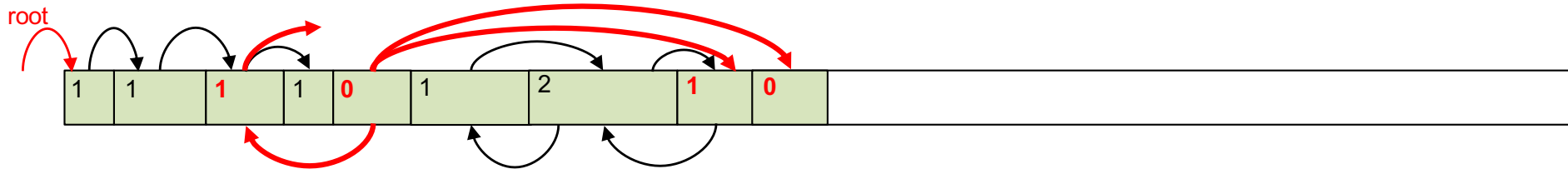
1. Objects are allocated on the heap. Each object keeps a count of the # of refs to it.



2. When a reference is changed, counts are updated.



3. When a count goes to zero, the object is deallocated. Its references are followed, and counts are decremented, and may go to zero. The process continues recursively.

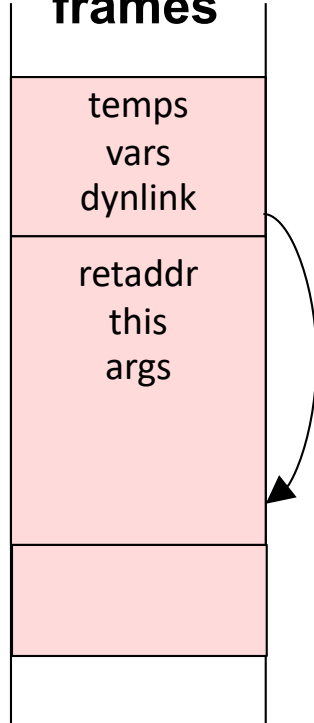


- + Short pauses: GC is incremental (a little work is done at each assignment)
- Inefficient (because work is done at each assignment)
- Cyclic structures are not garbage collected.
- No compaction – the heap becomes fragmented.

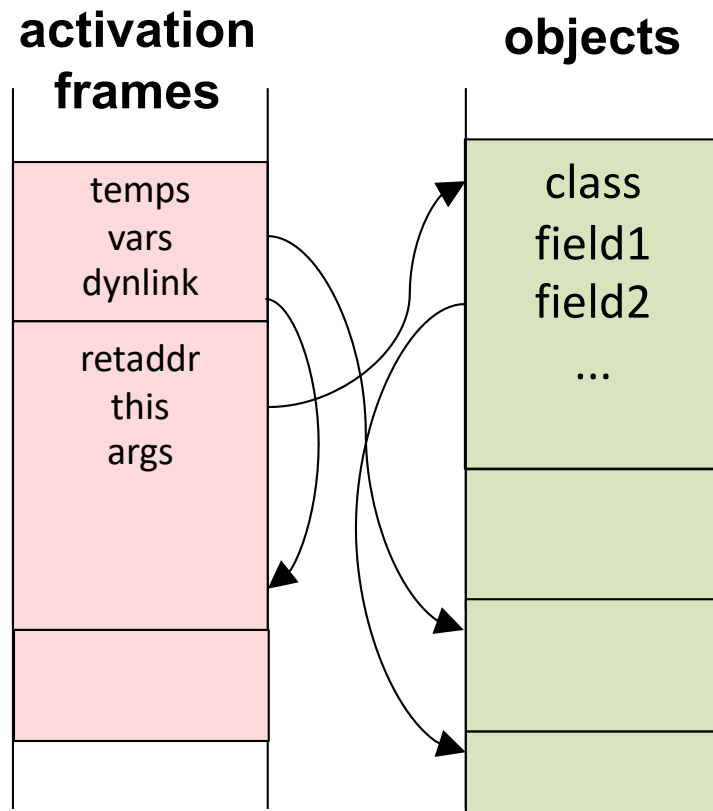
Fields and dynamic dispatch in OO

Typical runtime structures for objects

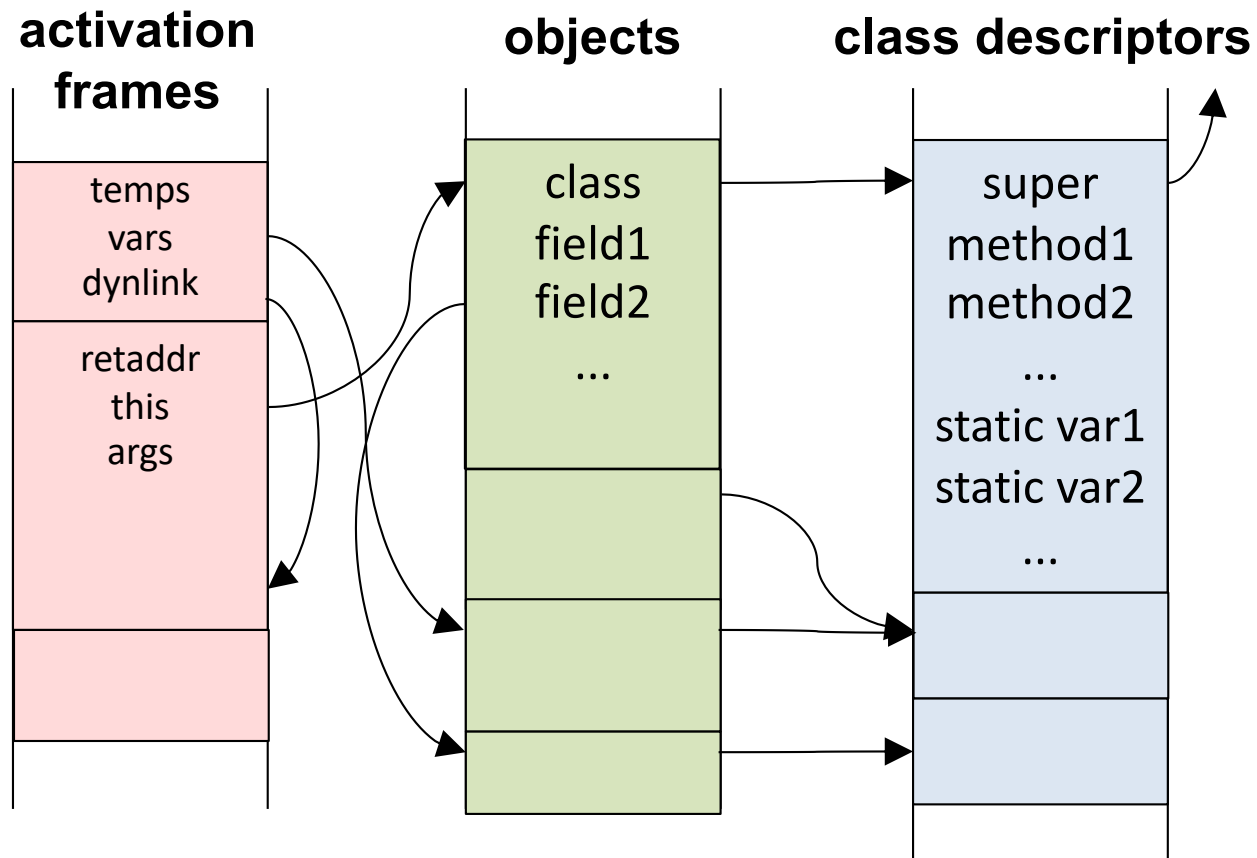
**activation
frames**



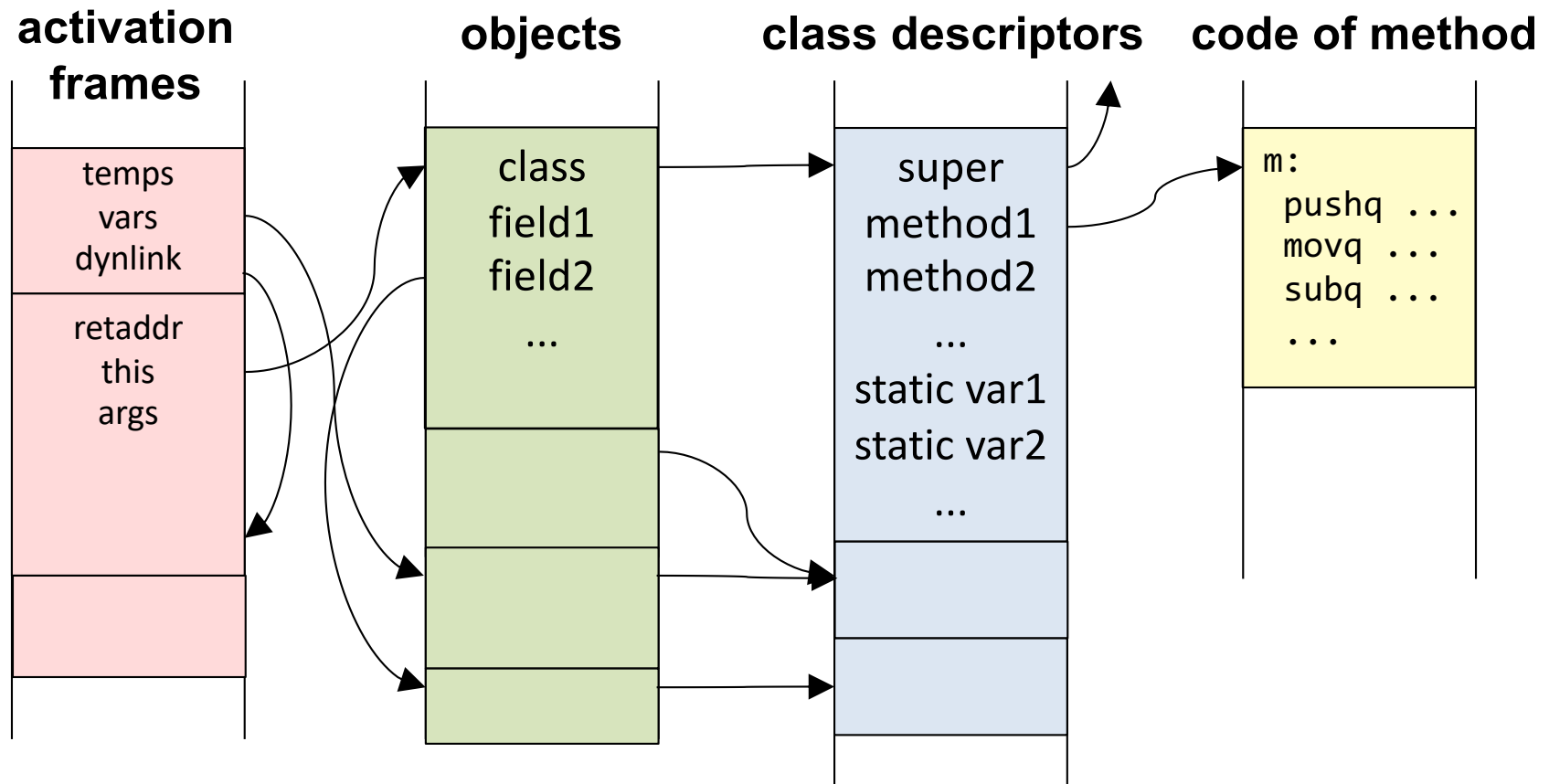
Typical runtime structures for objects



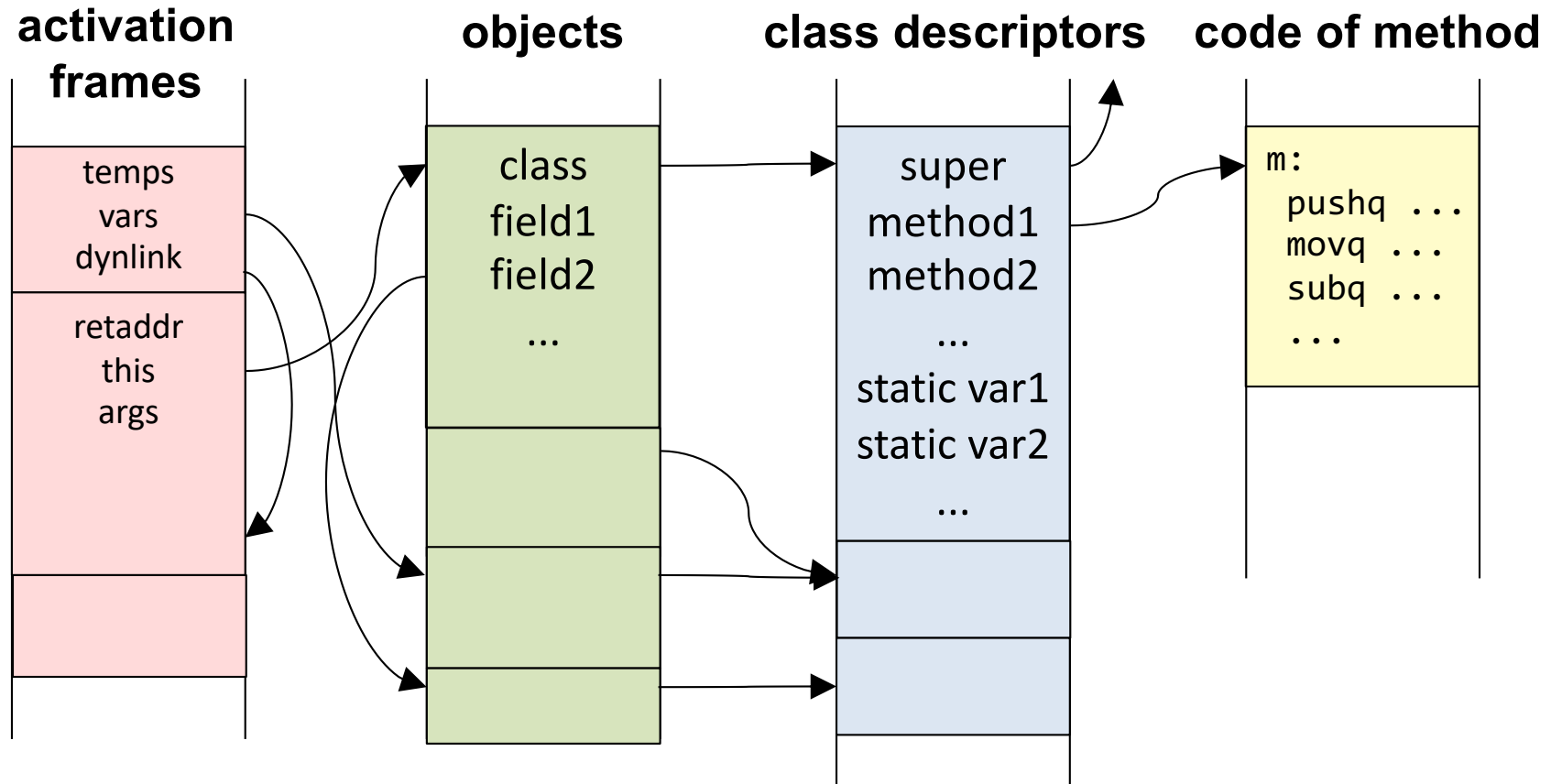
Typical runtime structures for objects



Typical runtime structures for objects



Typical runtime structures for objects



A method activation has a "this" pointer

- viewed as an extra 0th argument
- analogous to a static link
- used for accessing fields and methods

Note that vars, temps, and other args may also point to objects (GC roots).

An object has

- a pointer to the class descriptor (for accessing methods).
- fields that may point to objects.

A class descriptor has

data common to all objects of that class:

- a pointer to the superclass descriptor
- pointers to its methods
- static variables

Fields

Inheritance of fields, prefixing

source code

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

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

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

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

```
class  
fa1  
fa2
```

B-object

```
class  
fa1  
fa2  
fb
```

from A

from B

C-object

```
class  
fa1  
fa2  
fb  
fc
```

from A

from B

from C

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)

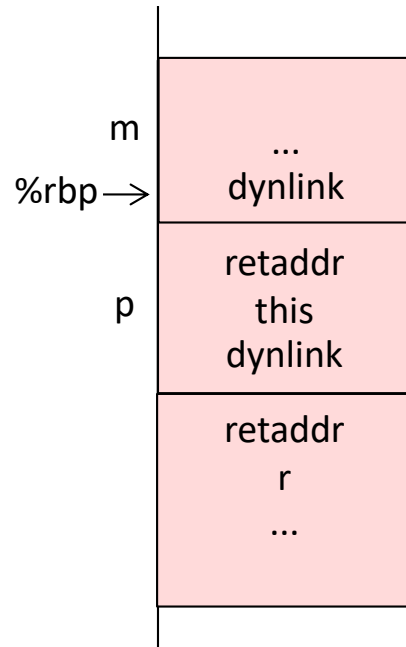
source code

```
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();  
}
```



Access to fields (single inheritance)

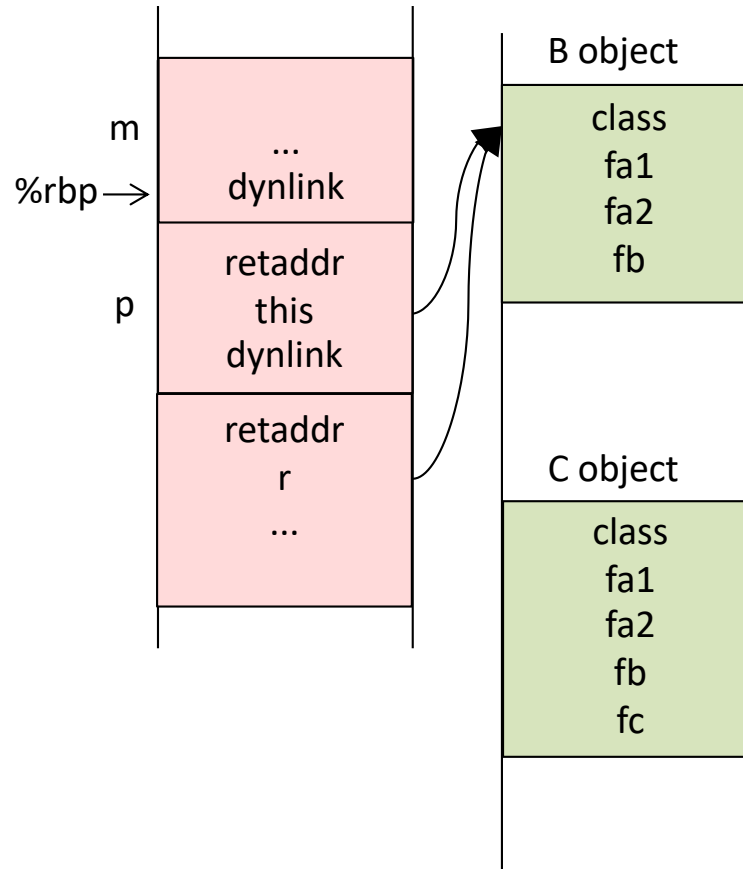
source code

```
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:
```

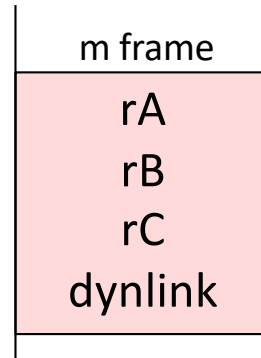
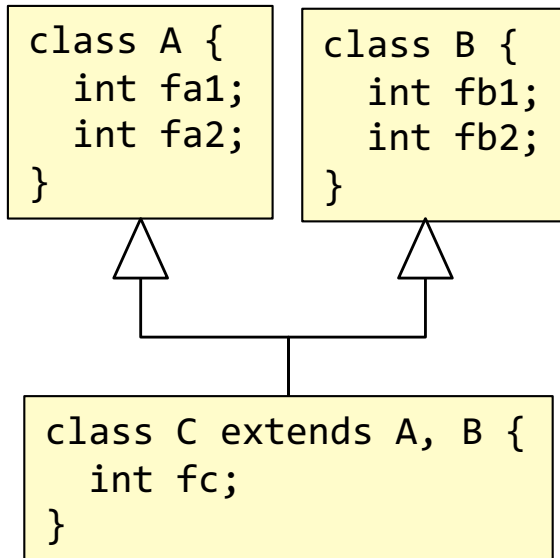
```
...
```

```
movq 16(%rbp), %rax    # this -> rax
```

```
movq 16(%rax), 8(%rax) # fa2 -> fa1
```

Access to fields (multiple inheritance, C++)

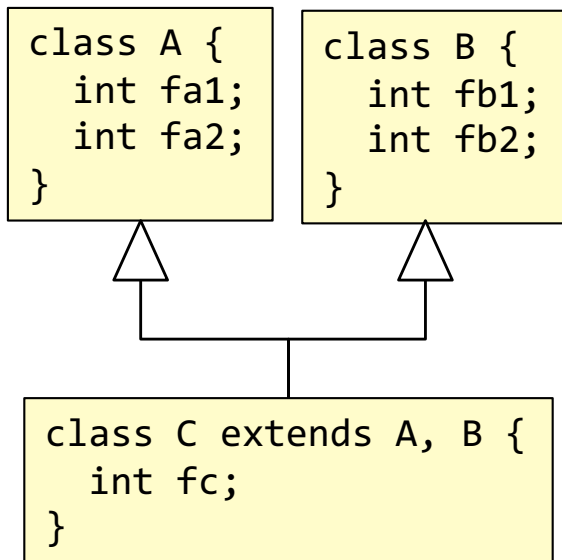
source code



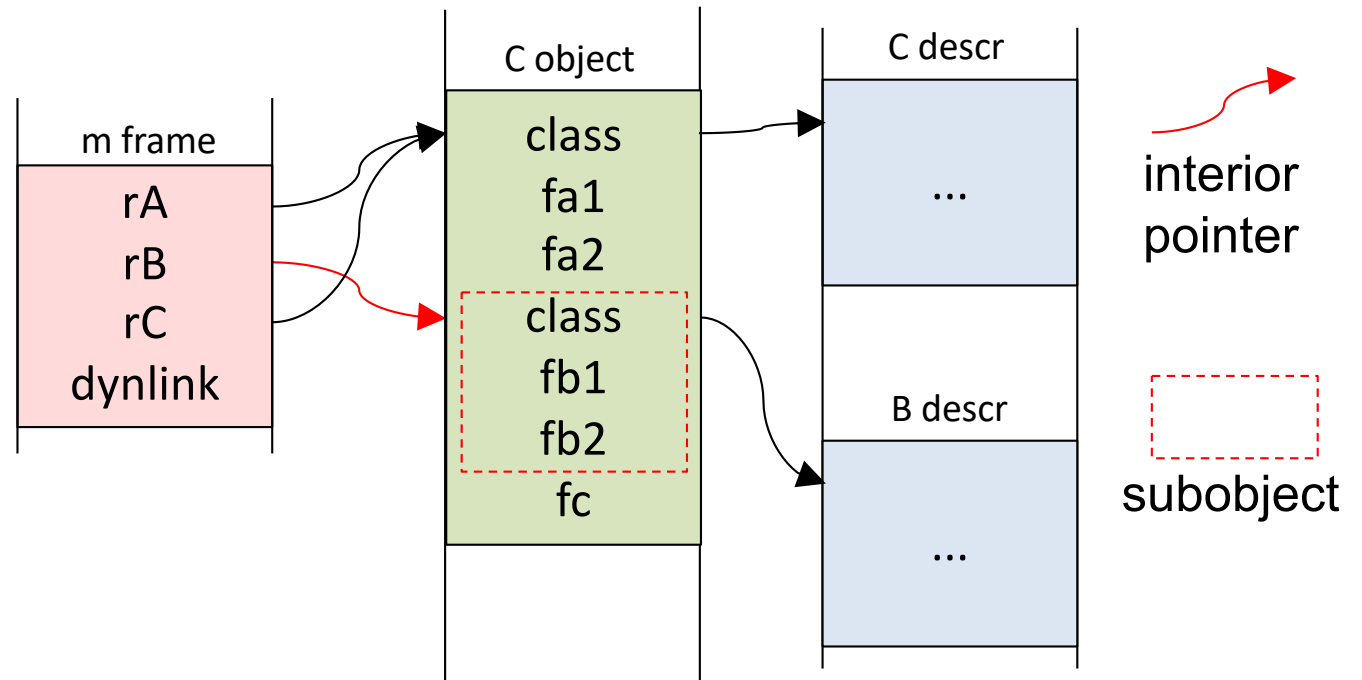
```
void m() {  
    C rC = new C();  
    B rB = rC;  
    A rA = rC;  
}
```

Access to fields (multiple inheritance, C++)

source code

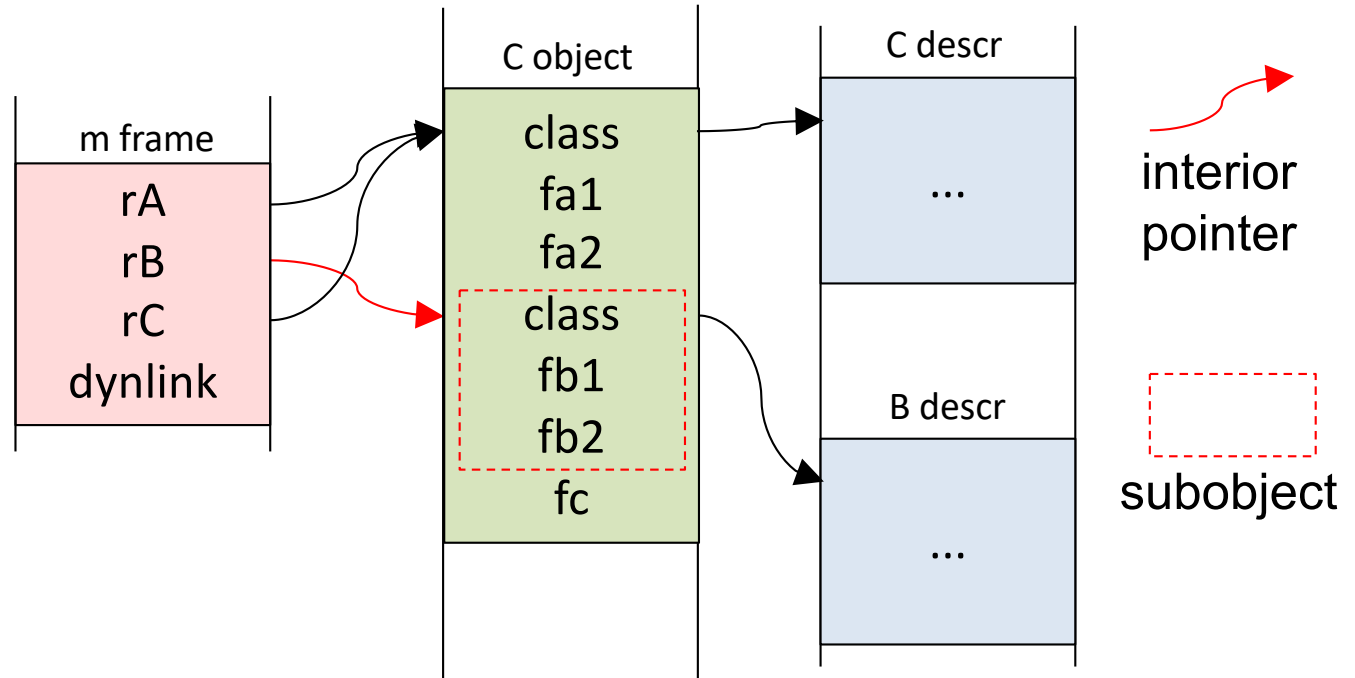
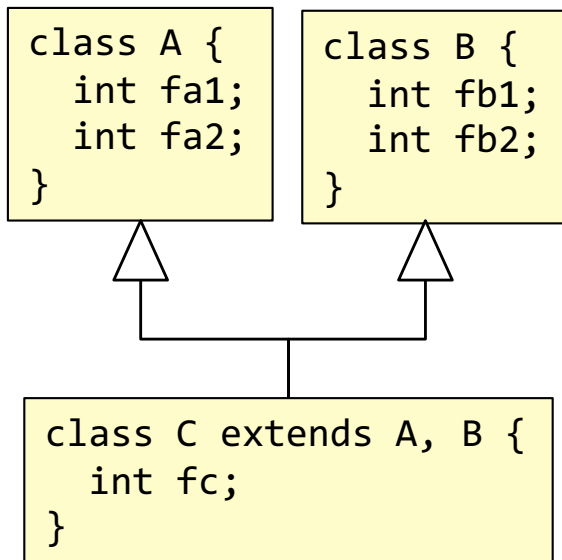


```
void m() {
  C rC = new C();
  B rB = rC;
  A rA = rC;
}
```



Access to fields (multiple inheritance, C++)

source code



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

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

Dynamic dispatch

(Calling methods in presence of inheritance and overriding)

source code

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



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



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

code

A-ma:

...

B-mb:

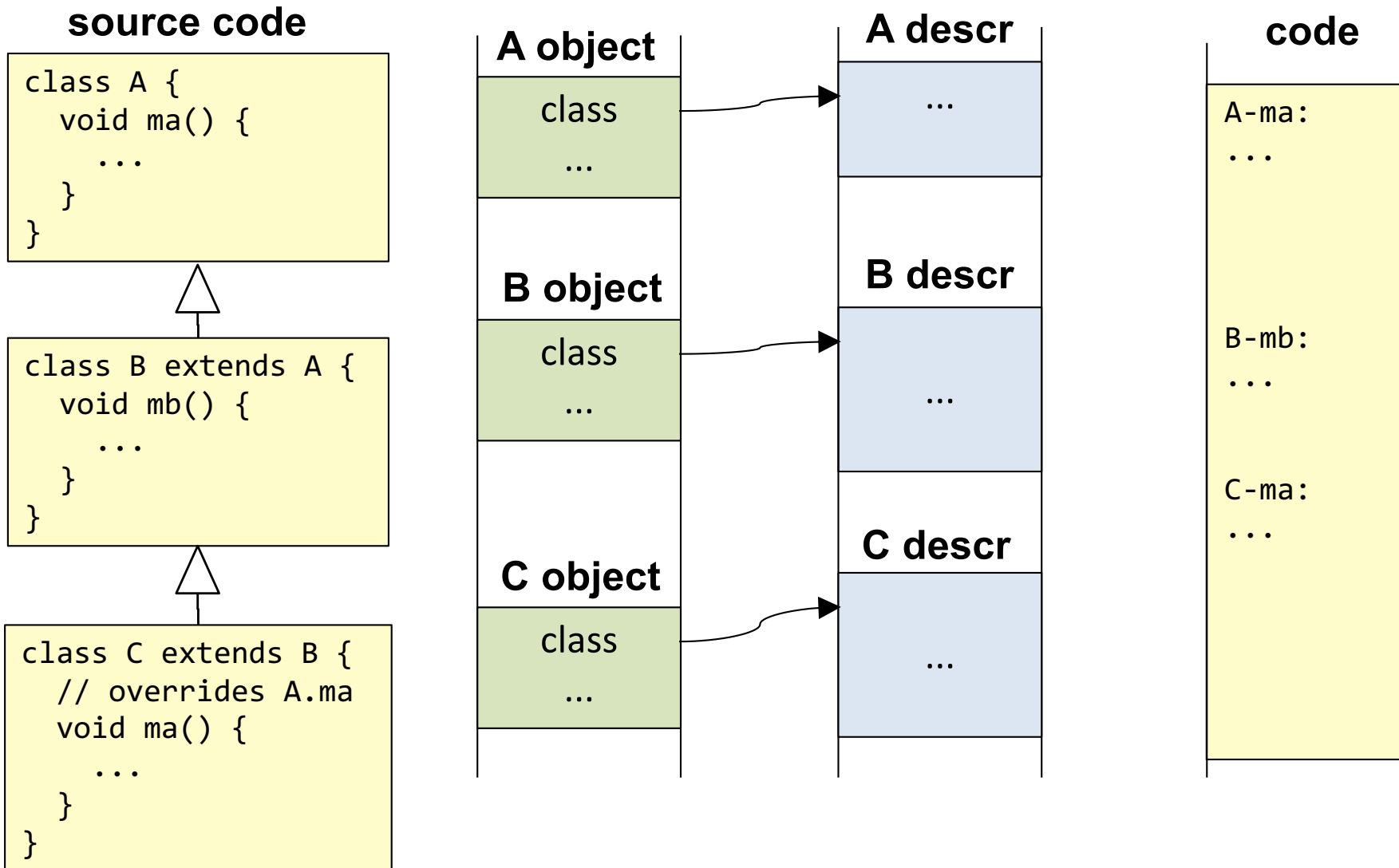
...

C-ma:

...

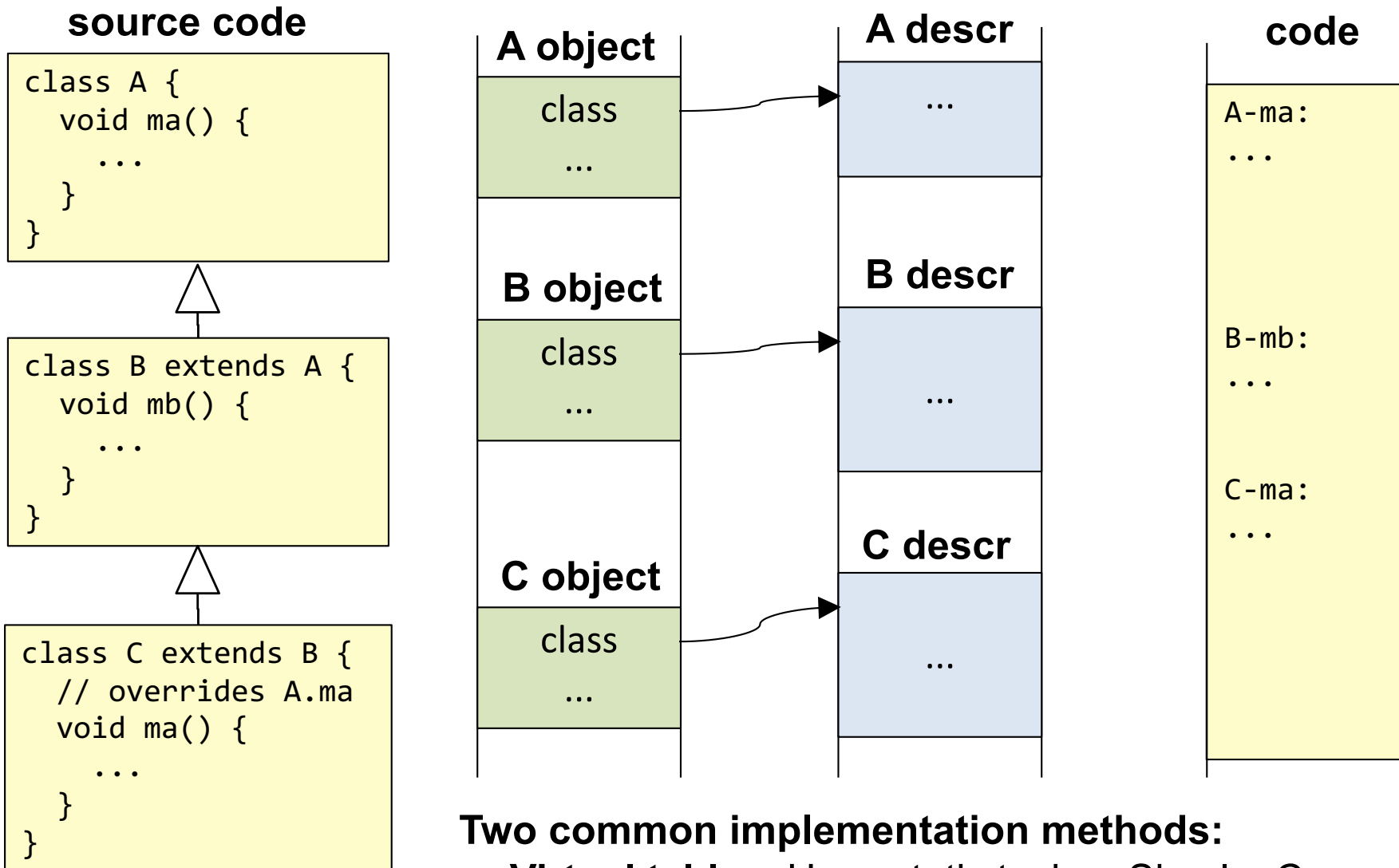
Dynamic dispatch

(Calling methods in presence of inheritance and overriding)



Dynamic dispatch

(Calling methods in presence of inheritance and overriding)

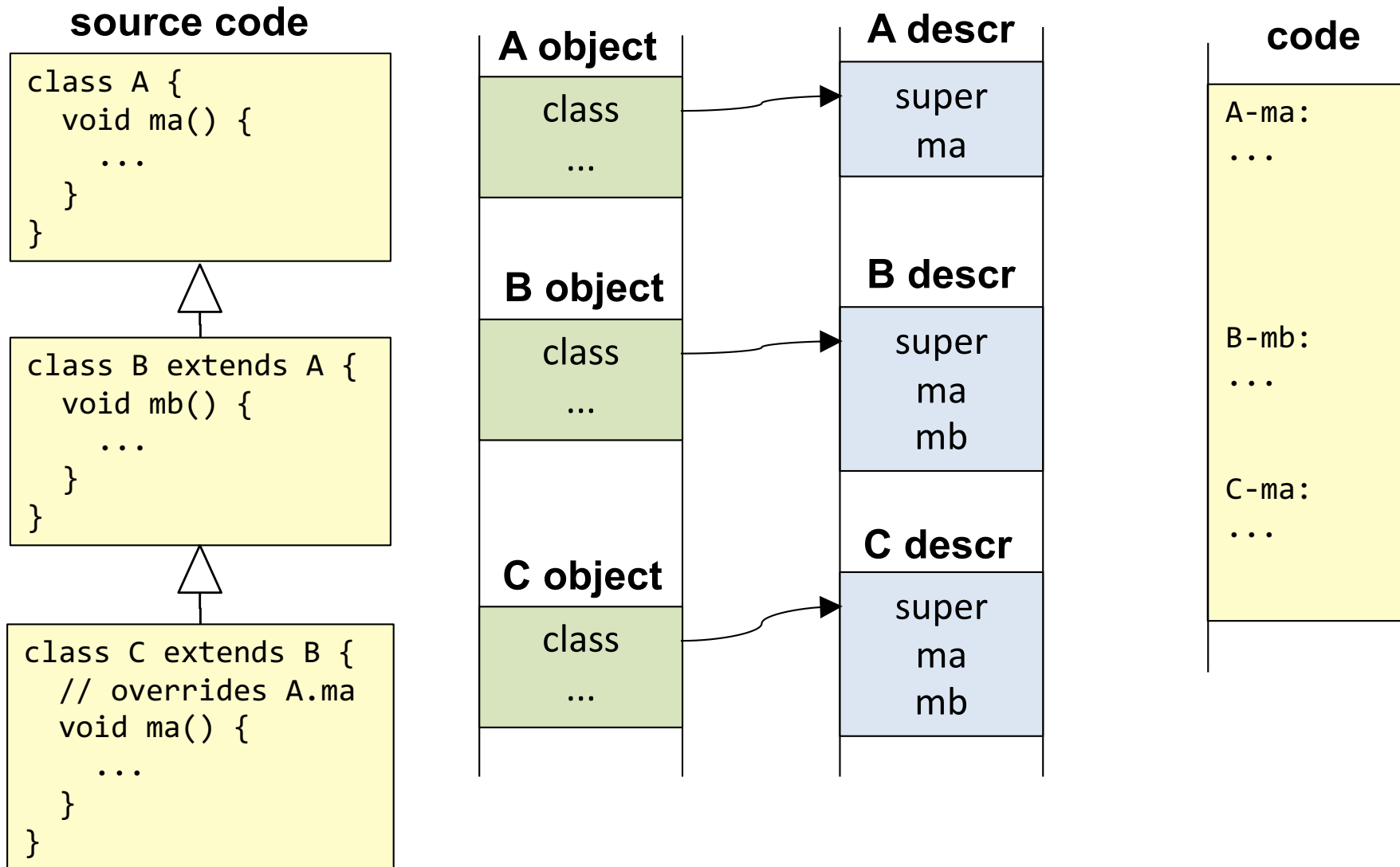


Two common implementation methods:

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

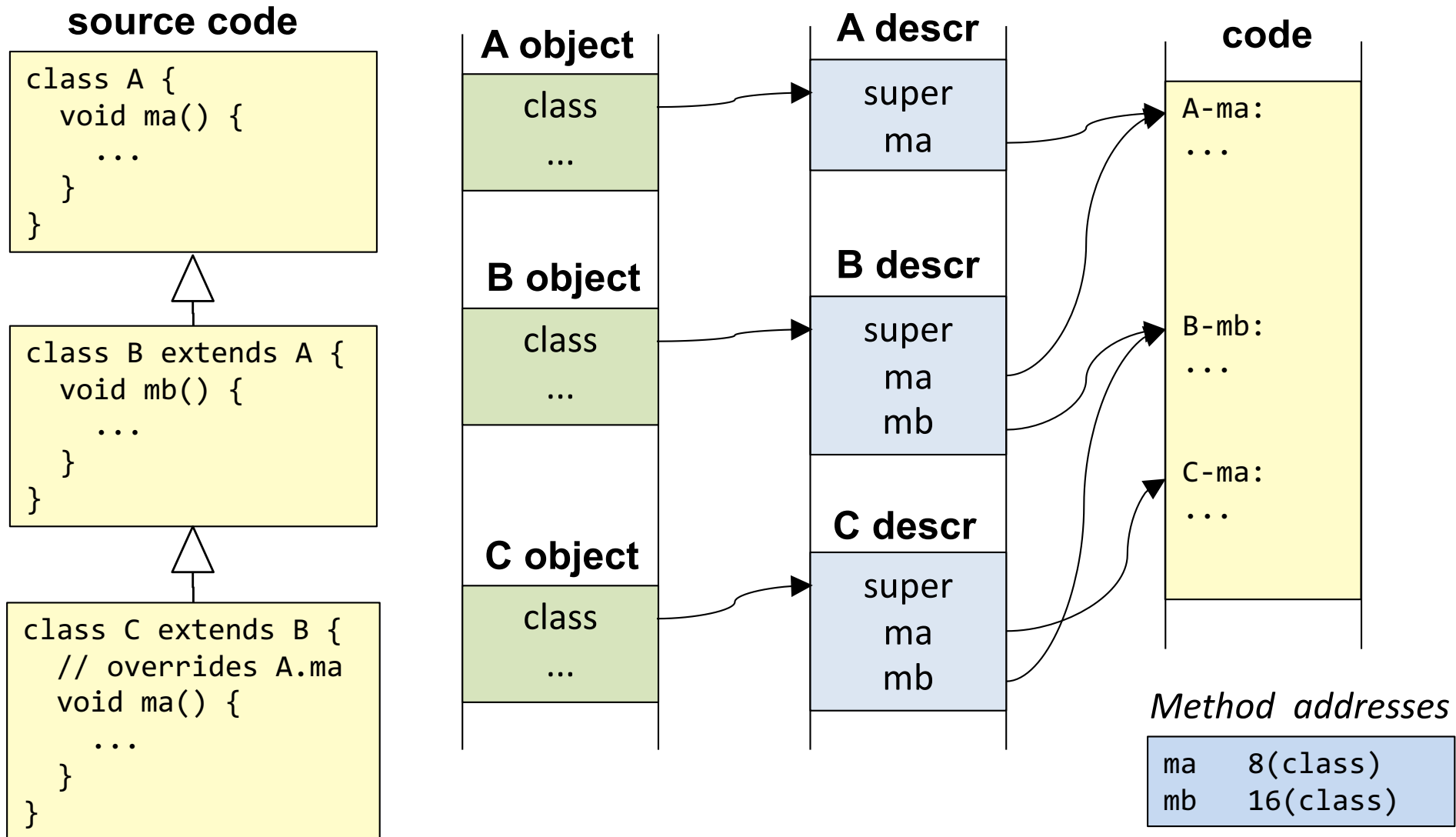
Virtual table dynamic dispatch

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



Virtual table dynamic dispatch

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



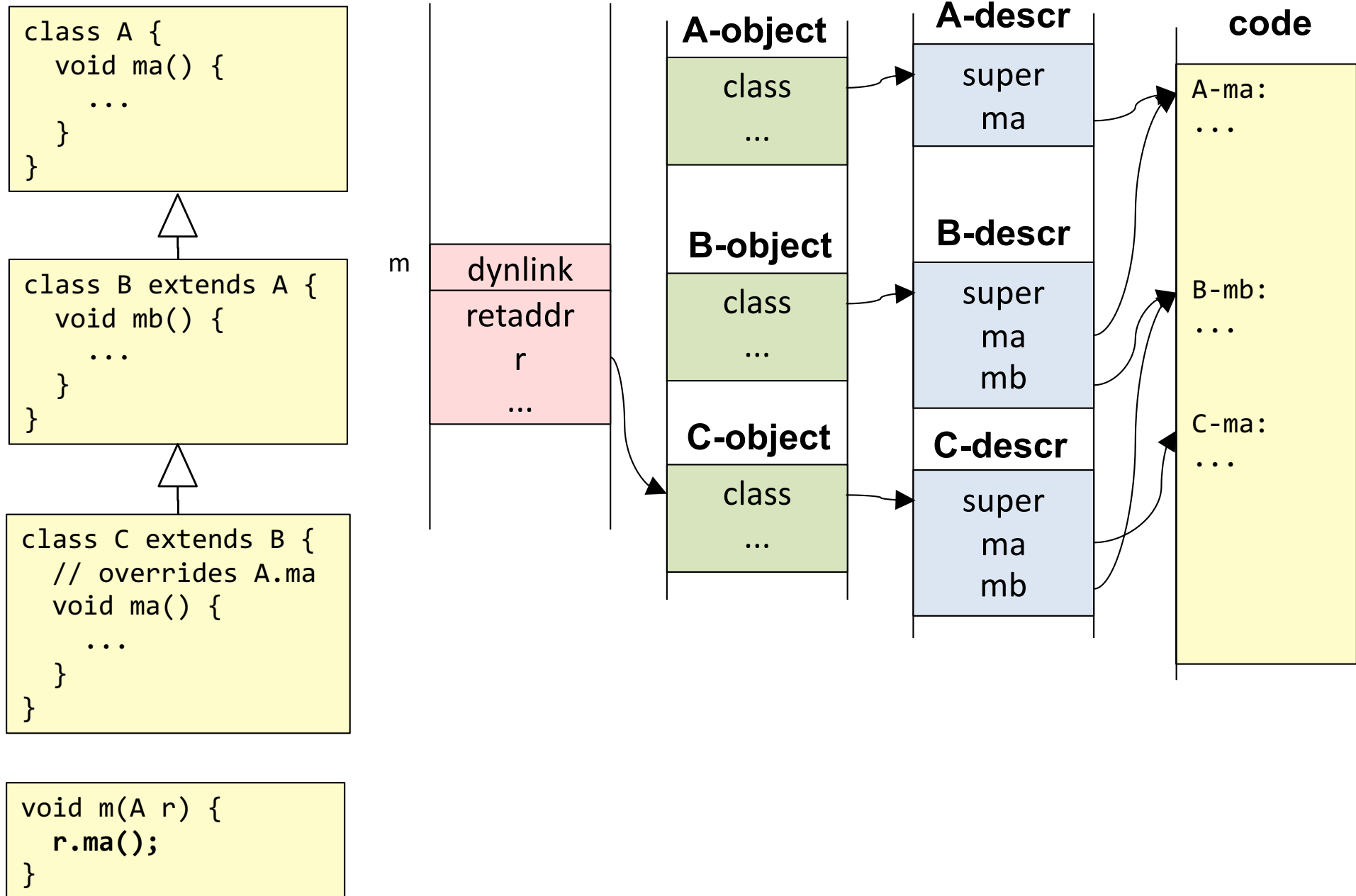
Virtual tables

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



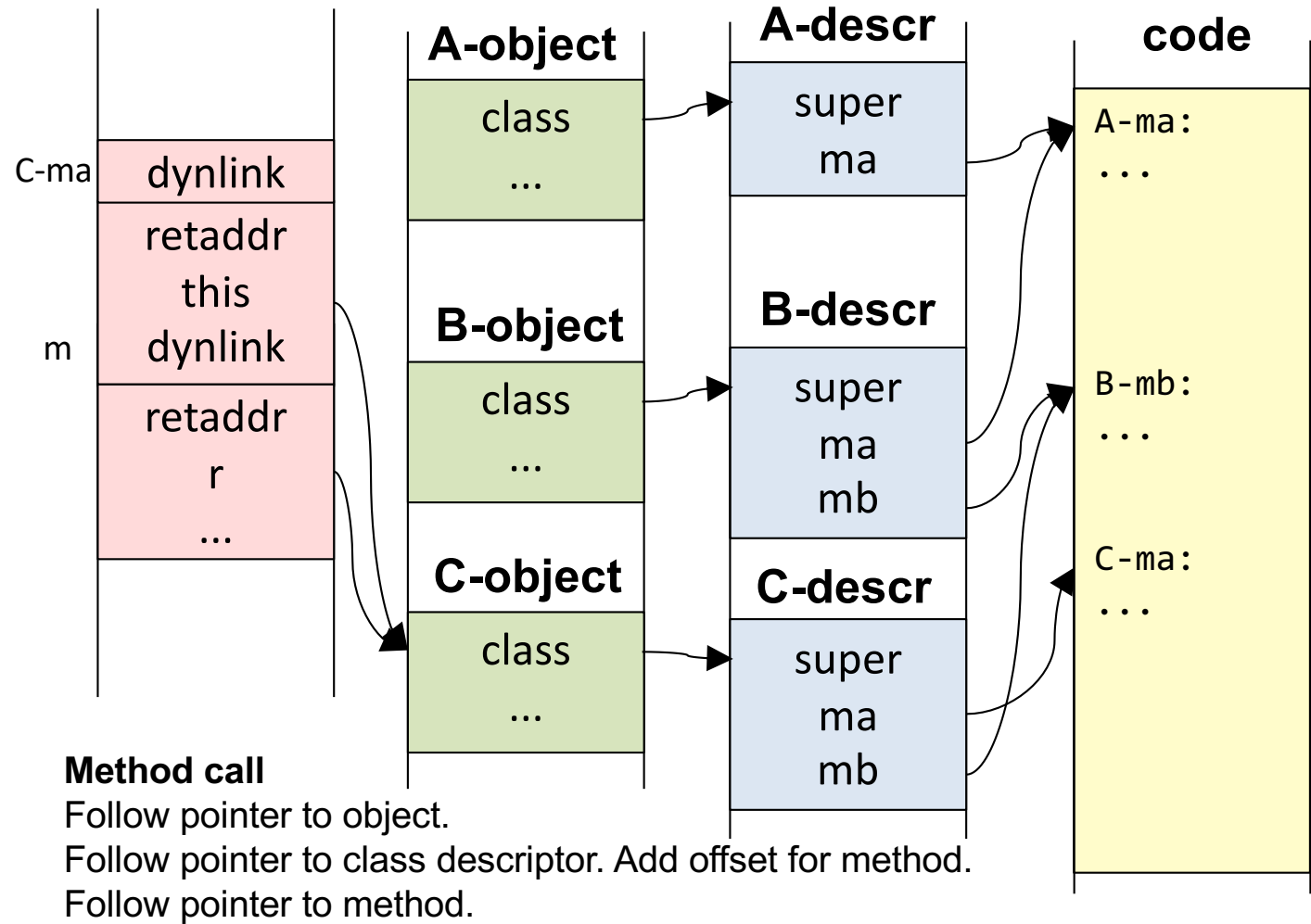
Calling a method via the virtual table

```
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();
}
```



```
m:
  ...
  movq 16(%rbp), %rax    # r -> rax
  pushq %rax            # push the static link to ma (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

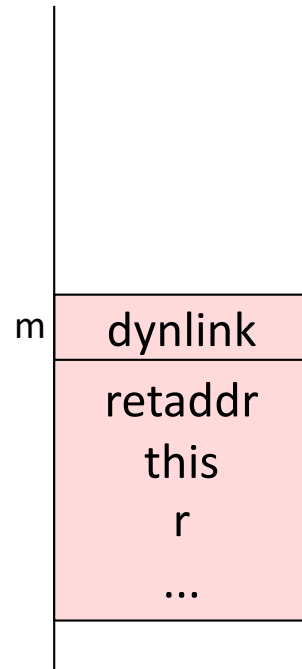
```
class A {  
  method s() {...}  
  method t() {...}  
}
```



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

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

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



code

A-s:

...

A-t:

...

B-t:

...

C-u:

...

C-s:

...

Hash table dynamic dispatch

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

methods and vars have
no static types

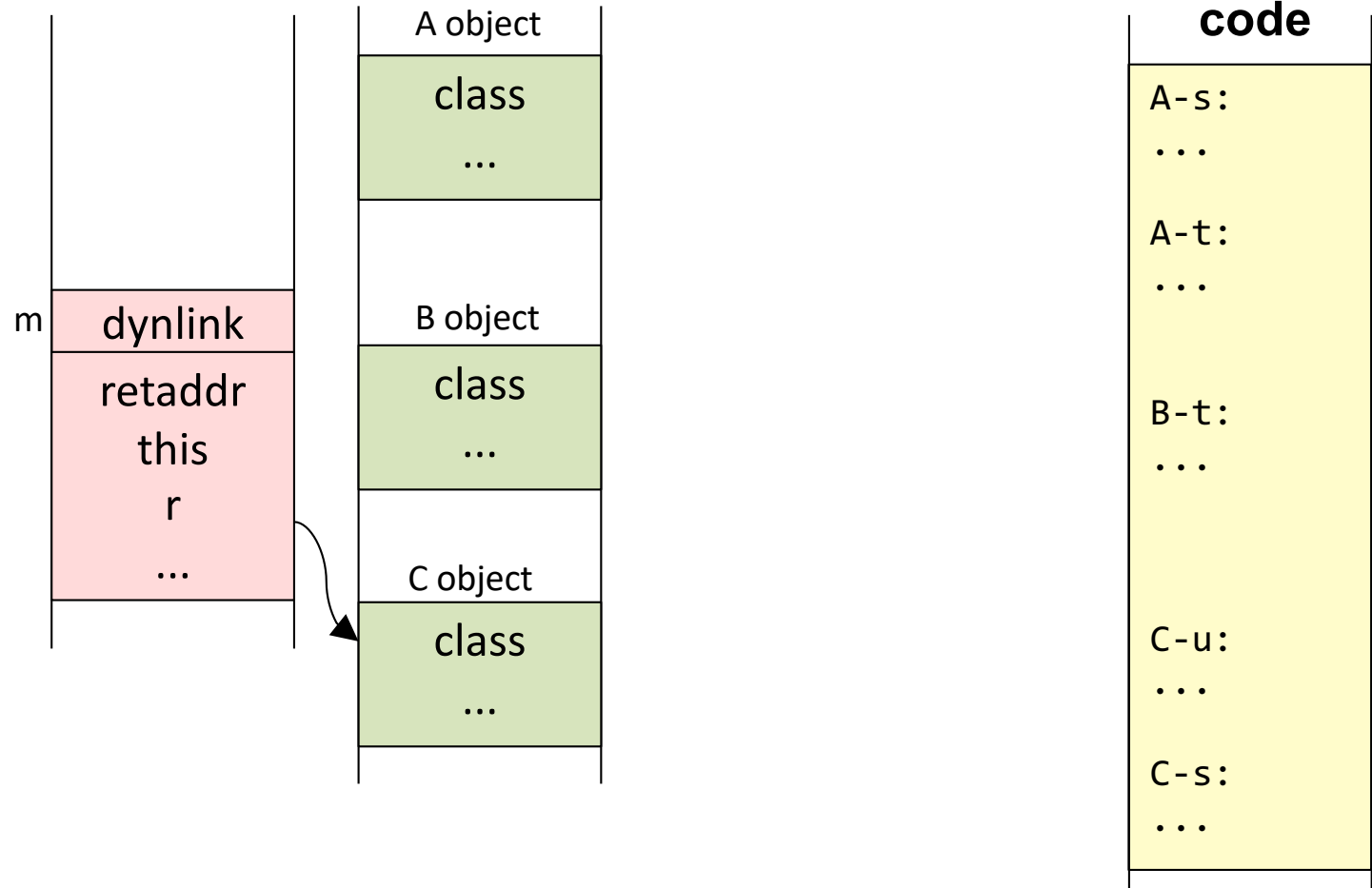
```
class A {  
  method s() {...}  
  method t() {...}  
}
```



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

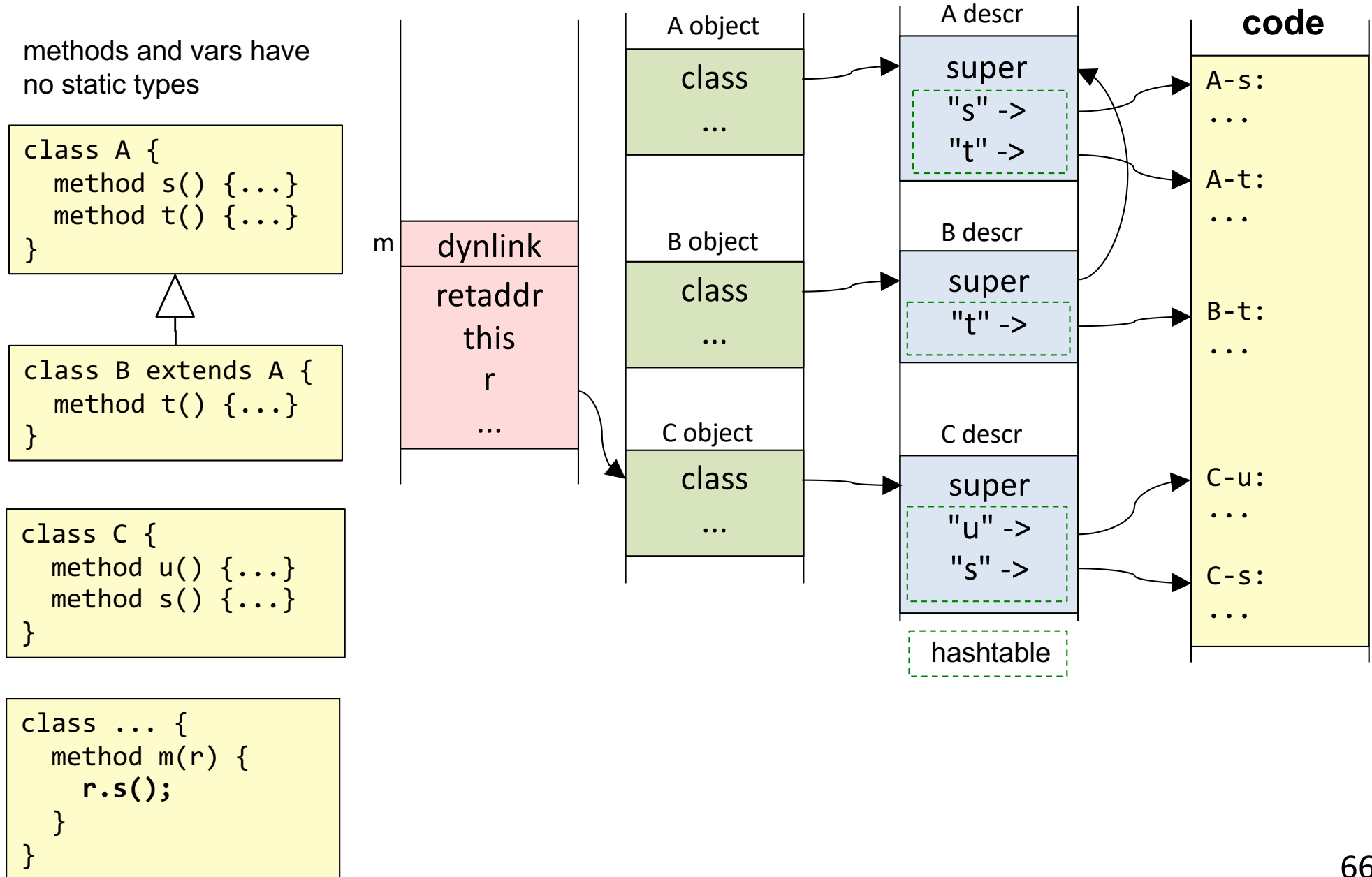
```
class C {  
  method u() {...}  
  method s() {...}  
}
```

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



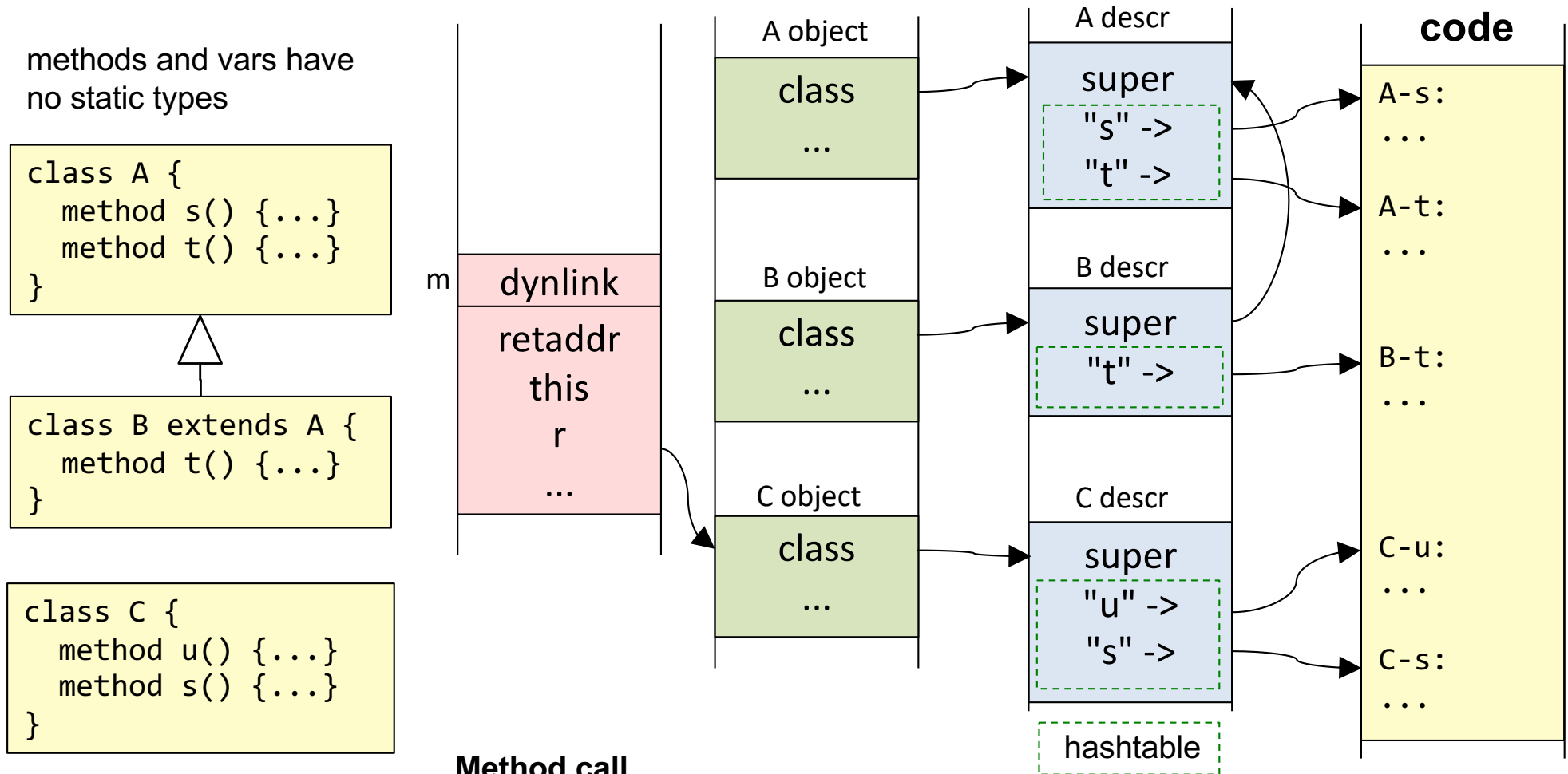
Hash table dynamic dispatch

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



Hash table dynamic dispatch

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



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)

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)

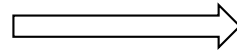
Example local optimizations

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

Example local optimizations

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

common subexpression elimination

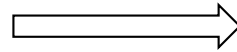


```
t = b * c;  
a = t + d;  
e = f + t;
```


Example local optimizations

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

common subexpression elimination



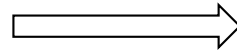
```
t = b * c;  
a = t + d;  
e = f + t;
```

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

Example local optimizations

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

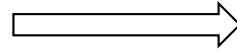
common subexpression elimination



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

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

constant propagation



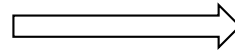
```
int a = 37;  
return 37 + 5;
```

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

Example local optimizations

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

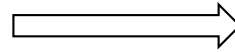
common subexpression elimination



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

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

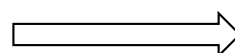
constant propagation



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

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

constant folding



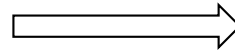
```
int a = 37;  
return 42;
```

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

Example local optimizations

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

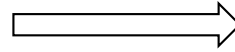
common subexpression elimination



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

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

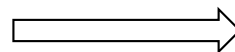
constant propagation



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

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

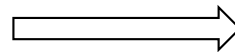
constant folding



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

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

dead code elimination



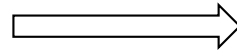
```
return 42;
```

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

Example local optimizations

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

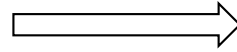
common subexpression elimination



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

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

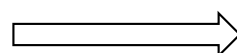
constant propagation



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

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

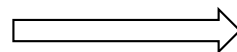
constant folding



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

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

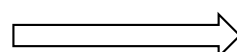
dead code elimination



```
return 42;
```

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

loop invariant code motion



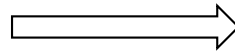
```
a = b + 3;  
for (int i ...) {  
    x[i] = a * i;  
}
```

Inlining

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

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

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

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.

Available as an experimental option for Java 9, but removed in Java 16.

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.
Available as an experimental option for Java 9, but removed in Java 16.

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!

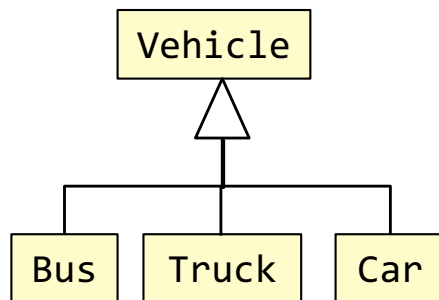
The approach used today in the SUN/Oracle JVM, called "HotSpot".

Inline call caches

a way to optimize method calls at runtime

Original calling code

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



Inline call caches

a way to optimize method calls at runtime

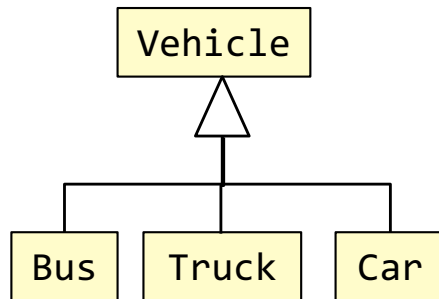
Original calling code

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

optimize →

Optimized calling code

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



Inline call caches

a way to optimize method calls at runtime

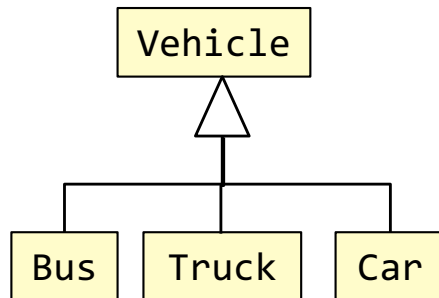
Original calling code

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

optimize →

Optimized calling code

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



Called method:

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


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 Car-m.

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

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

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

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

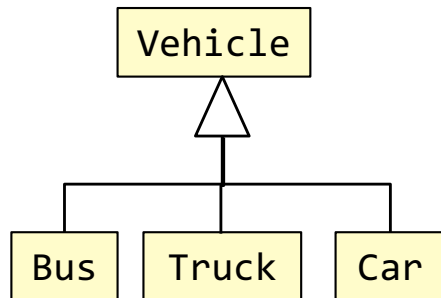
Original calling code

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

optimize →

Optimized calling code

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



Called method:

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

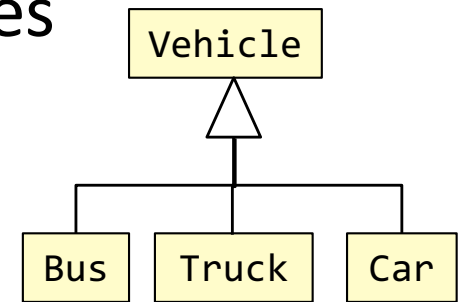
Polymorphic inline caches (PICs)

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();
    Car-m-prologue(v);
}
```

Methods:

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

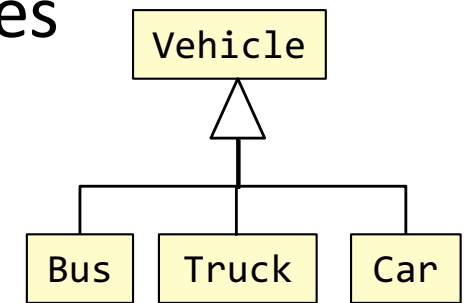
Polymorphic inline caches (PICs)

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();
  Car-m-prologue(v);
}
```

optimize

Polymorphic inlined cache

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

Methods:

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

Methods:

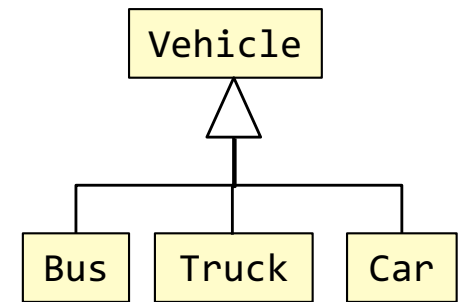
```
Car-m:
  ...
Truck-m:
  ...
```

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 Car)
    Car-m(v)
  else if (v is a Truck)
    Truck-m(v)
  else
    v.m(); // normal lookup
}
```

Methods:

Car-m:

...

Truck-m:

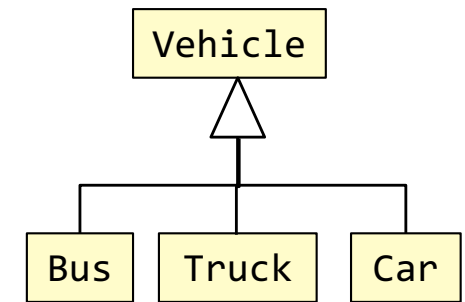
...

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 Car)
    Car-m(v)
  else if (v is a Truck)
    Truck-m(v)
  else
    v.m(); // normal lookup
}
```

Methods:

```
Car-m:
...
Truck-m:
...
```

optimize →

with inlined methods

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

Methods:

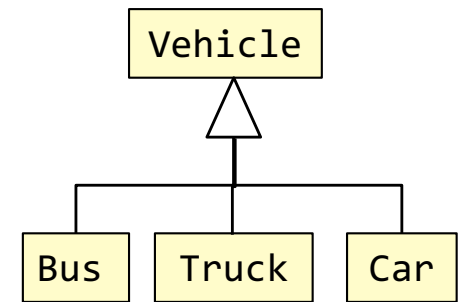
```
Car-m:
...
Truck-m:
...
```

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 Car)
    Car-m(v)
  else if (v is a Truck)
    Truck-m(v)
  else
    v.m(); // normal lookup
}
```

Methods:

```
Car-m:
...
Truck-m:
...
```

optimize →

with inlined methods

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

Methods:

```
Car-m:
...
Truck-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 VM

Local optimizations vs heavy inlining and other memory intensive optimizations.
For modern 64-bit machines, there is only a Server version available.

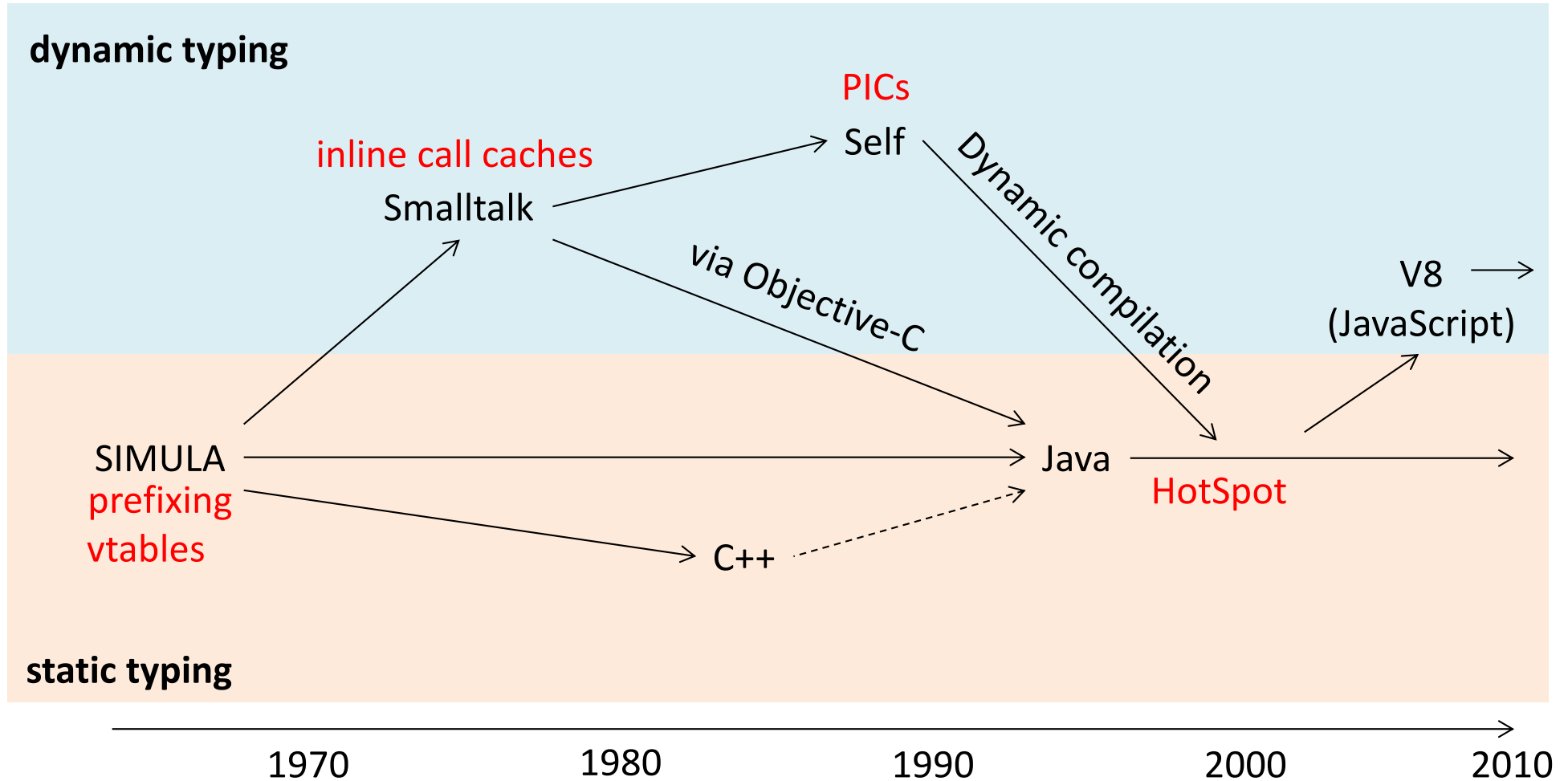
Warm-up vs. Steady state

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?