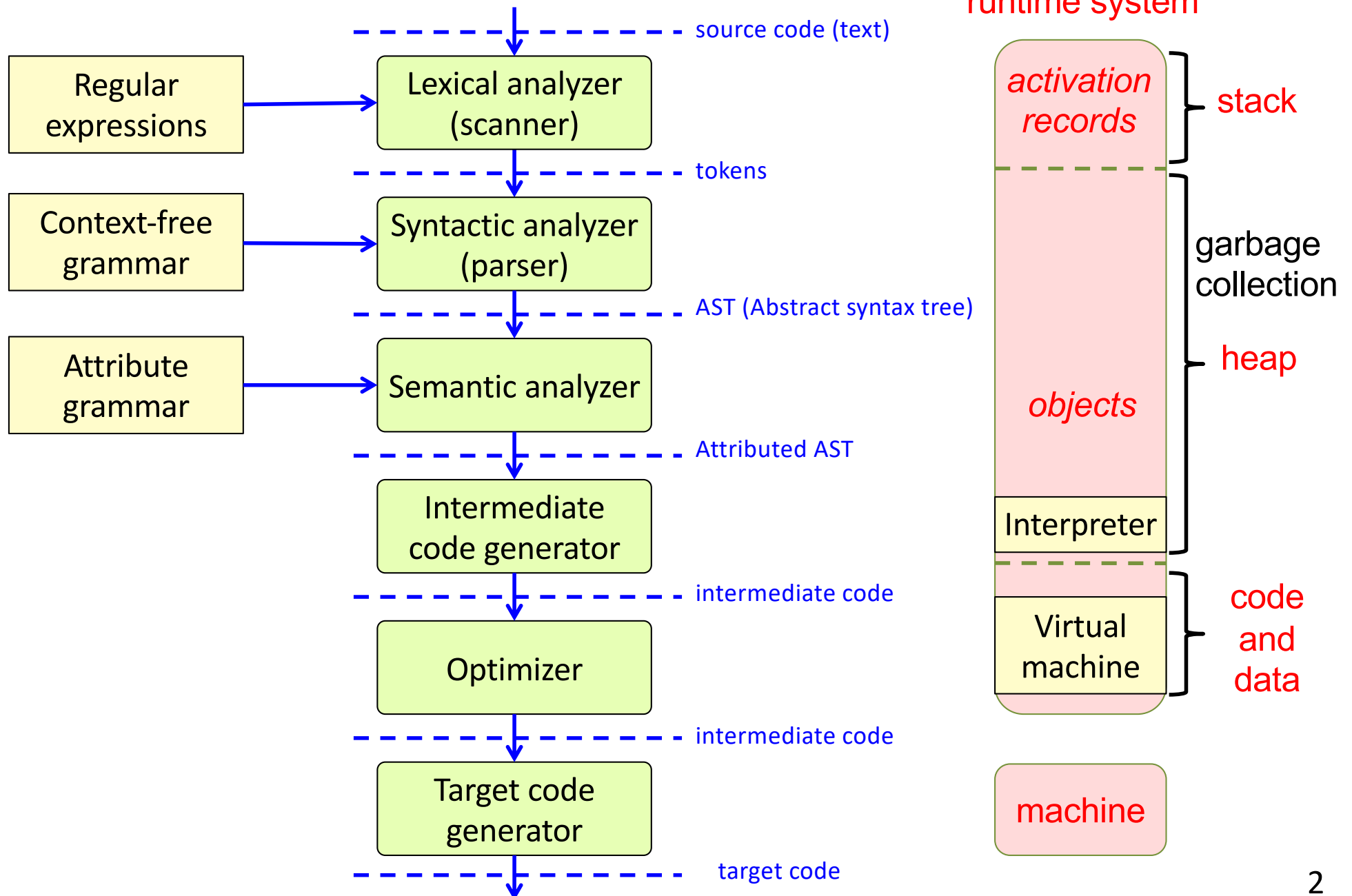EDAN65: Compilers, Lecture 12

Runtime systems for object-oriented languages
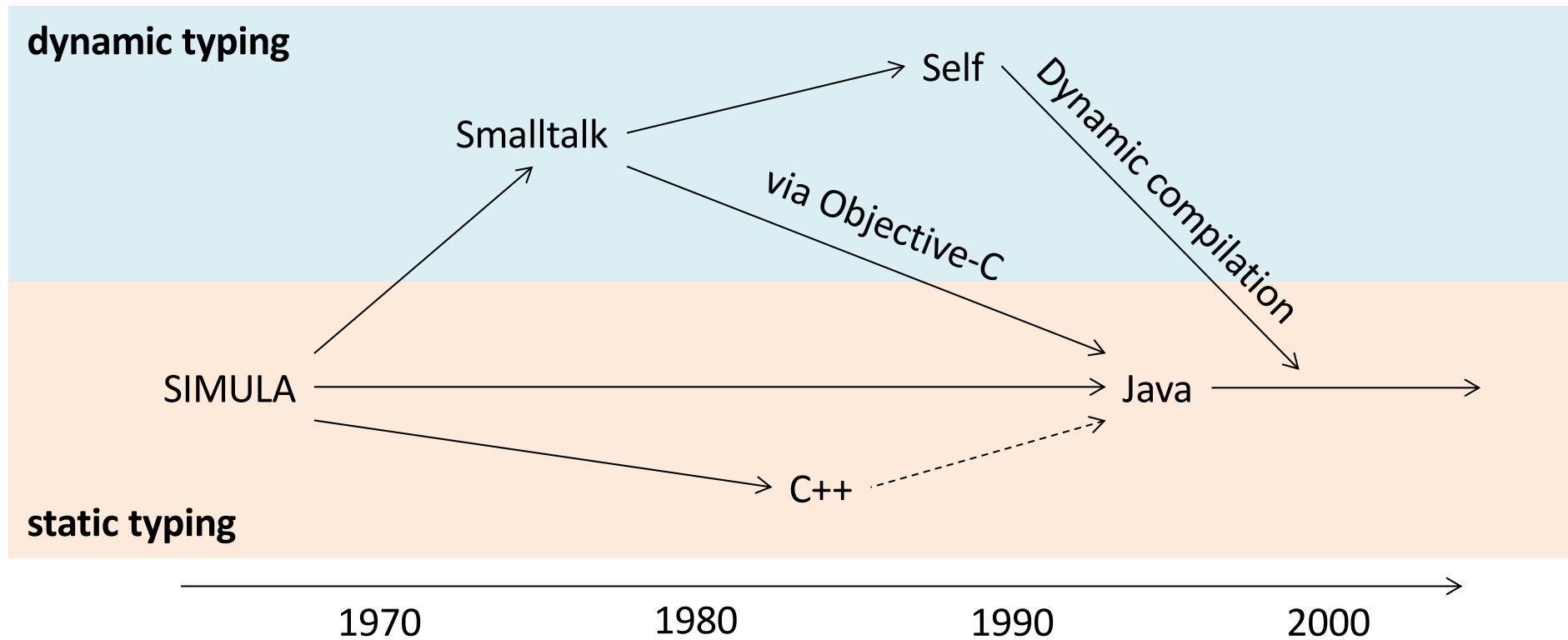
Görel Hedin

Revised: 2020-10-05
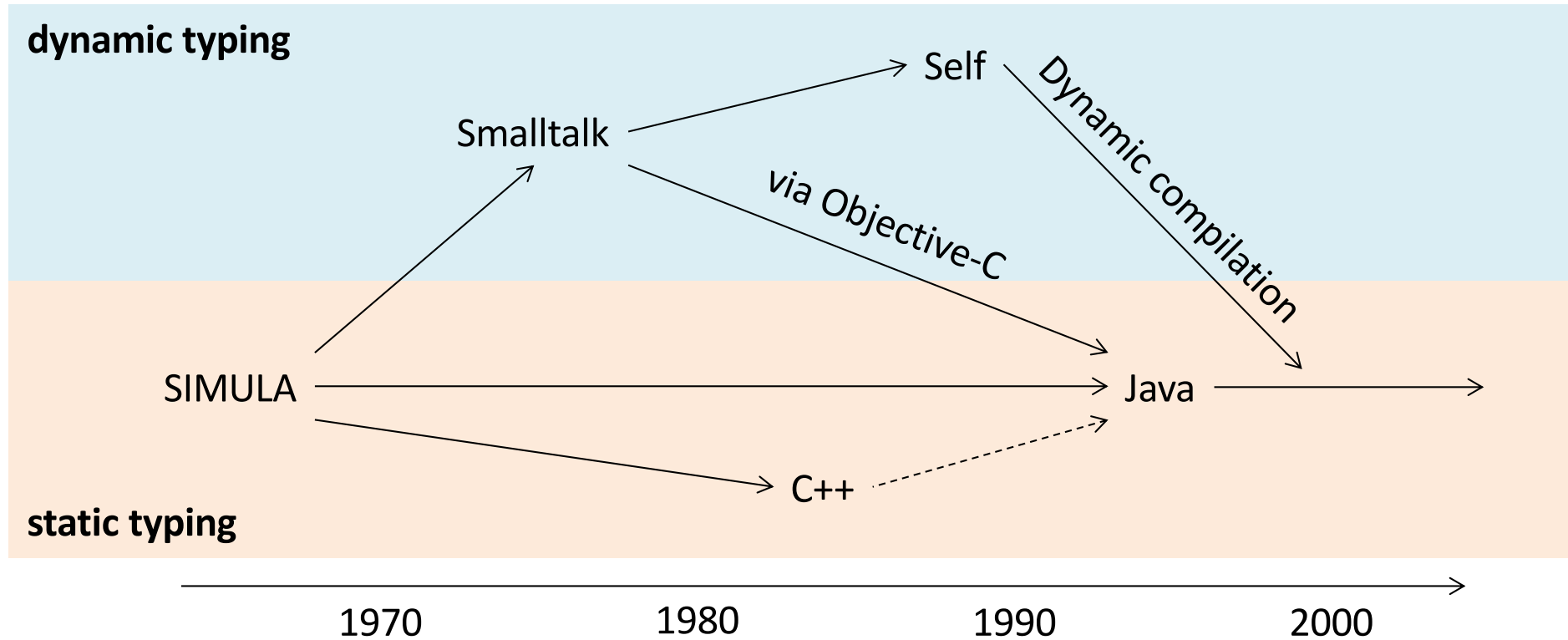
This lecture

source code (text)

Regular expressions	→	Lexical analyzer (scanner)

tokens

Context-free grammar	→	Syntactic analyzer (parser)

AST (Abstract syntax tree)

Attribute grammar	→	Semantic analyzer

Attributed AST

Intermediate code generator

intermediate code

Optimizer

intermediate code

Target code generator

target code

runtime system

activation records — stack

garbage collection

objects — heap

Interpreter

Virtual machine — code and data

machine

2

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).
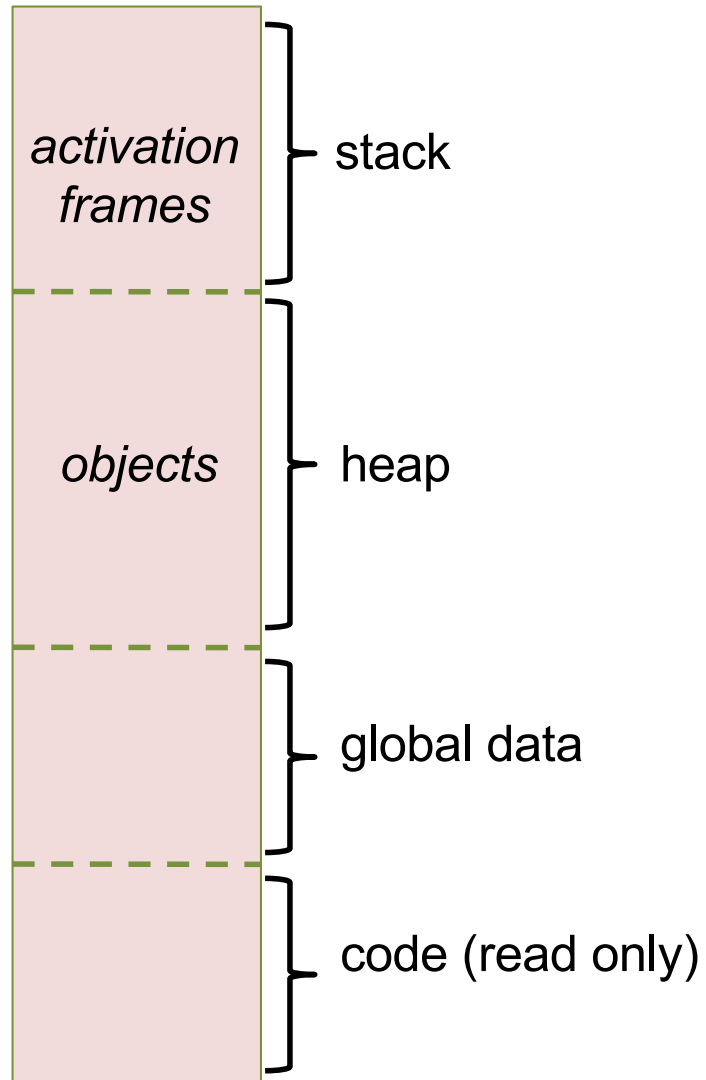
Dynamic typing
There are no types for things at compile-time. Only for objects at runtime.
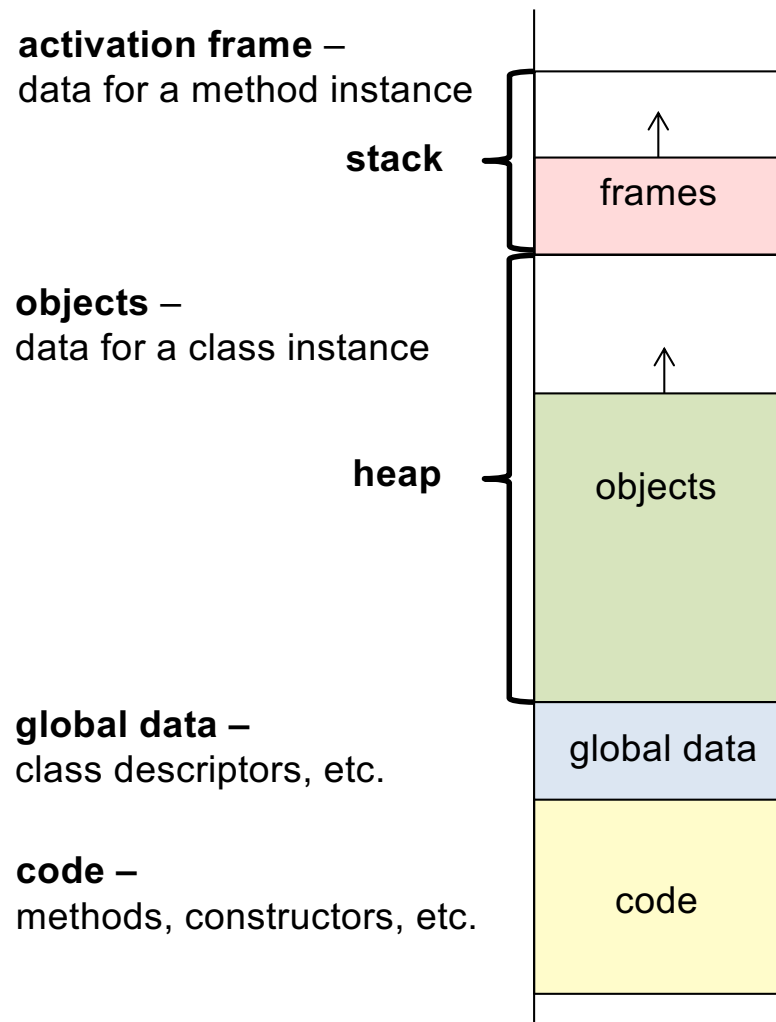
Static typing
Variables have types at compile-time.
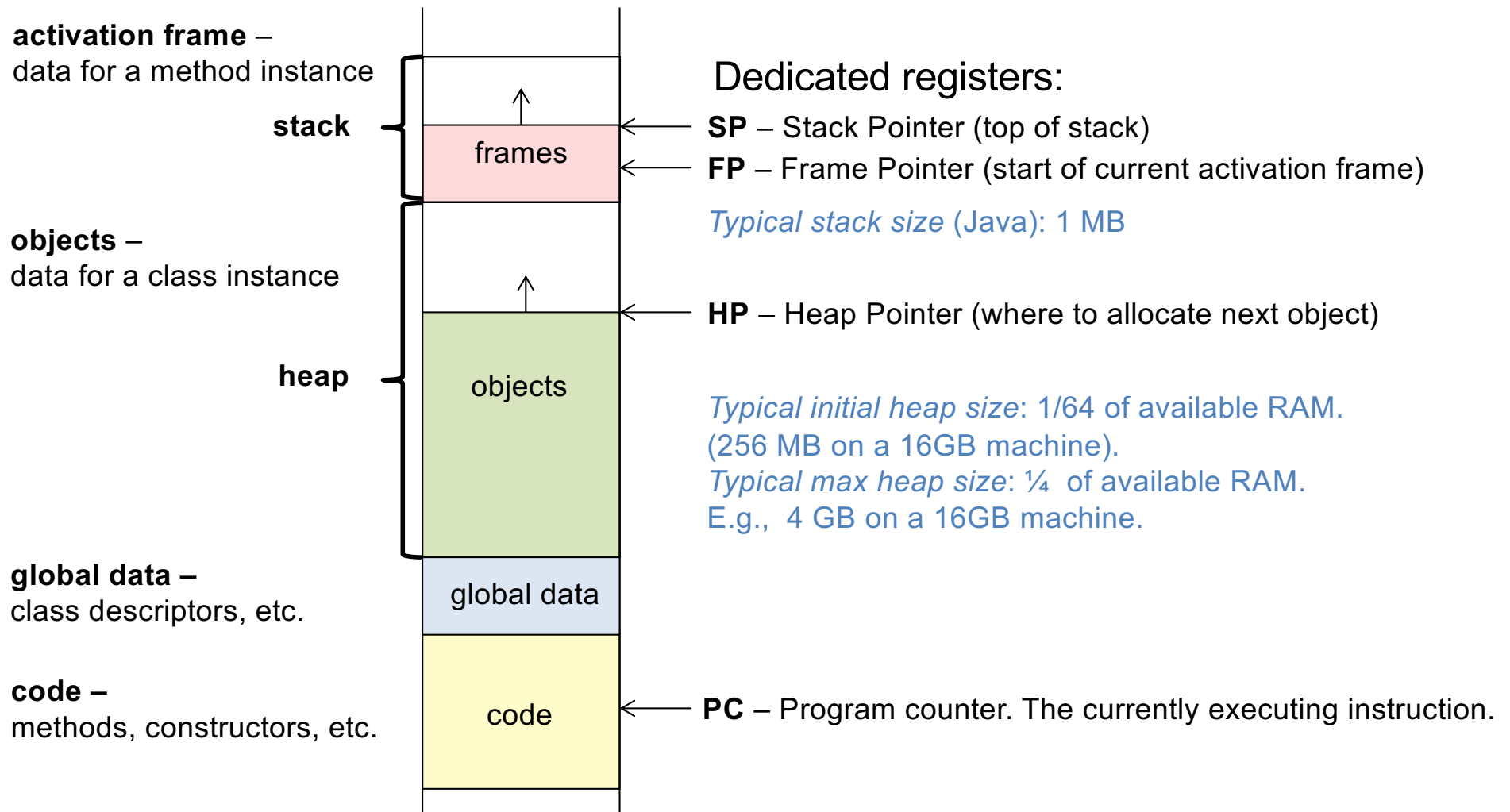At runtime, the variable points to an object of *at least* that type (that type, or a subtype).

4

Example memory segments
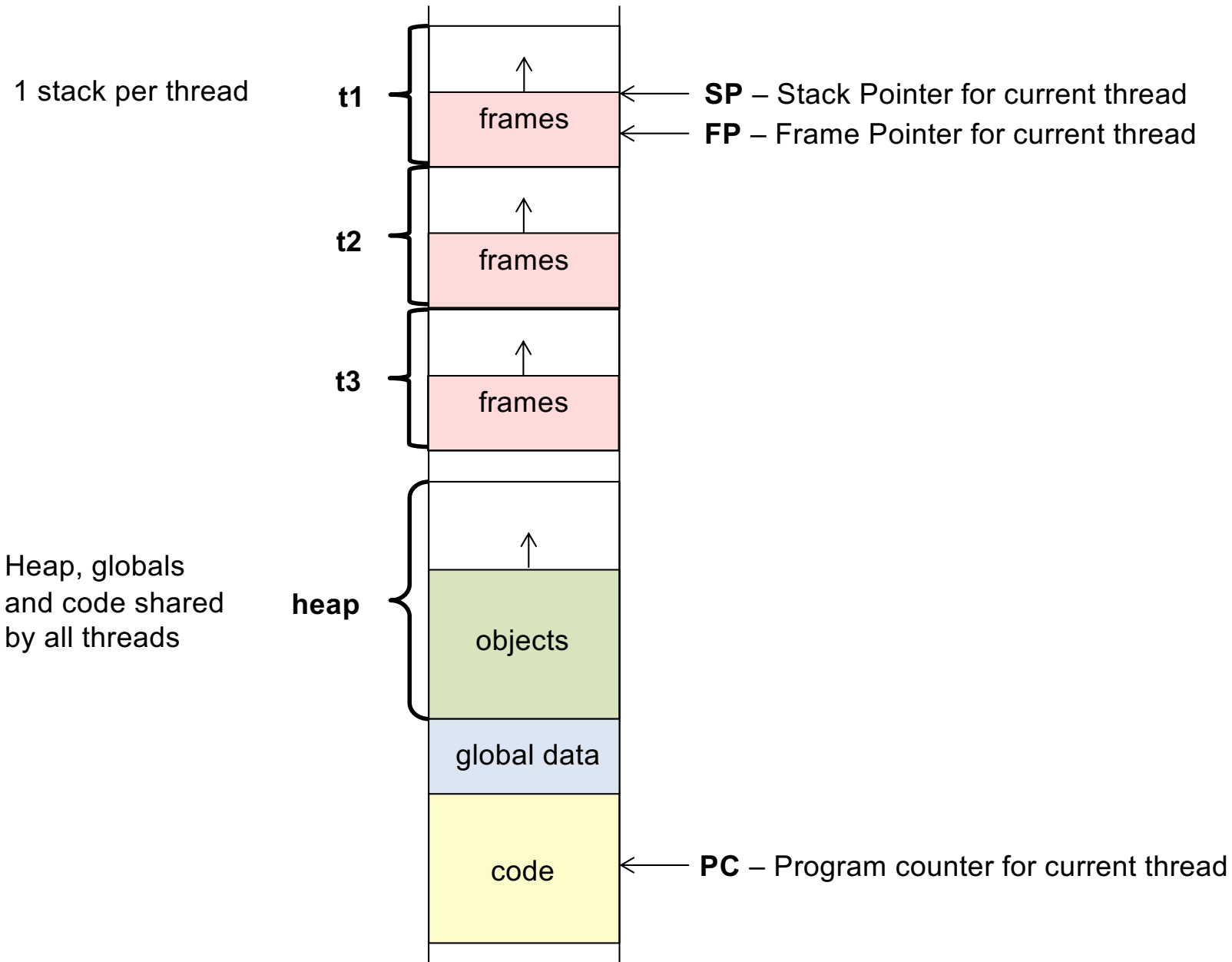
Typical memory usage for OO languages

activation frame –
data for a method instance

stack

frames

objects –
data for a class instance

heap

objects

global data –
class descriptors, etc.

global data

code –
methods, constructors, etc.

code

6

Typical memory usage for OO languages

activation frame –
data for a method instance

stack

objects –
data for a class instance

heap

global data –
class descriptors, etc.

code –
methods, constructors, etc.

frames

objects

global data

code

Dedicated registers:

SP – Stack Pointer (top of stack)
FP – Frame Pointer (start of current activation frame)

Typical stack size (Java): 1 MB

HP – Heap Pointer (where to allocate next object)

Typical initial heap size: 1/64 of available RAM.
(256 MB on a 16GB machine).
Typical max heap size: ¼ of available RAM.
E.g., 4 GB on a 16GB machine.

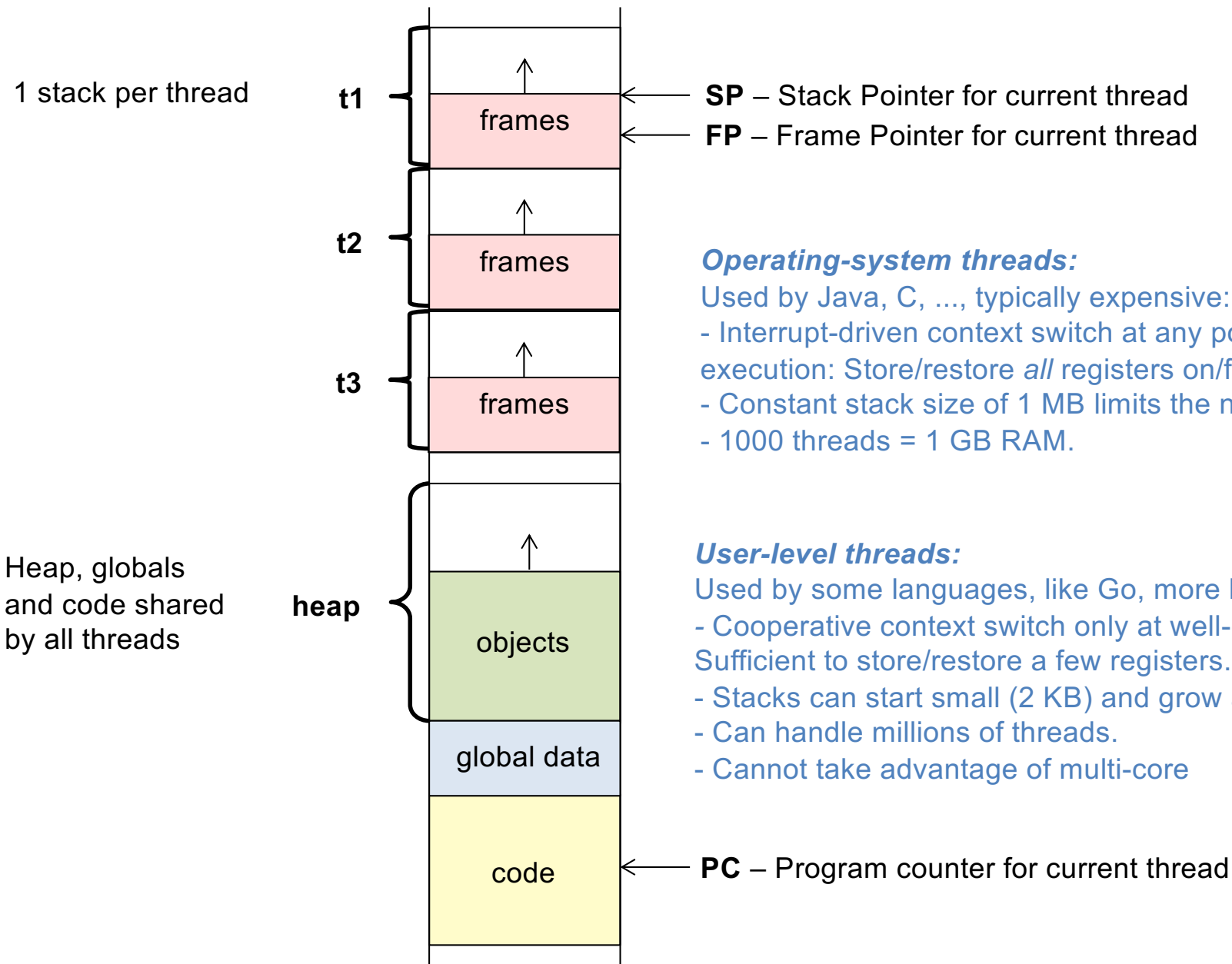**PC** – Program counter. The currently executing instruction.

Dynamic class loading (like in Java):
Would put class descriptors and code in heap objects instead of in separate data/code segments.
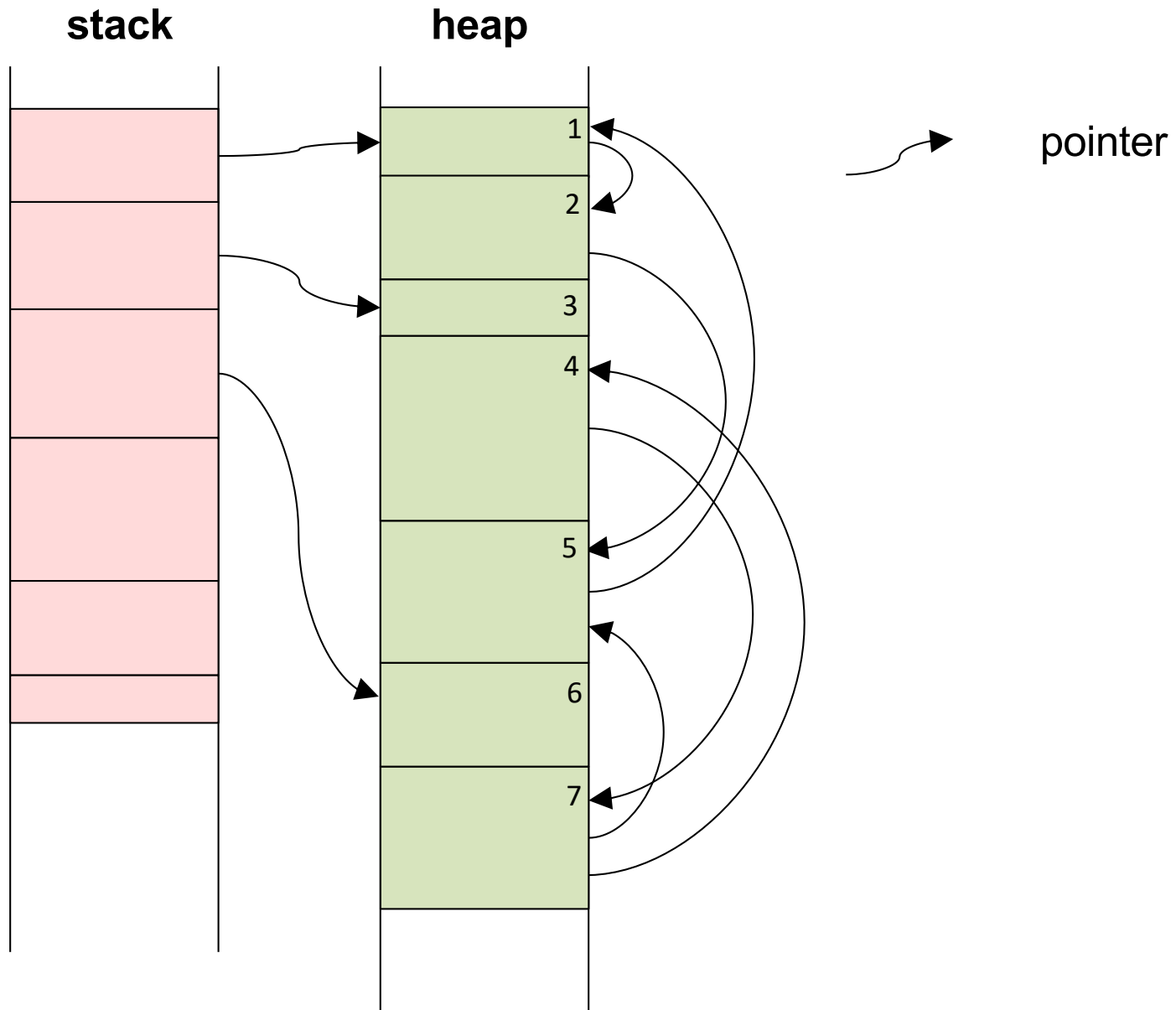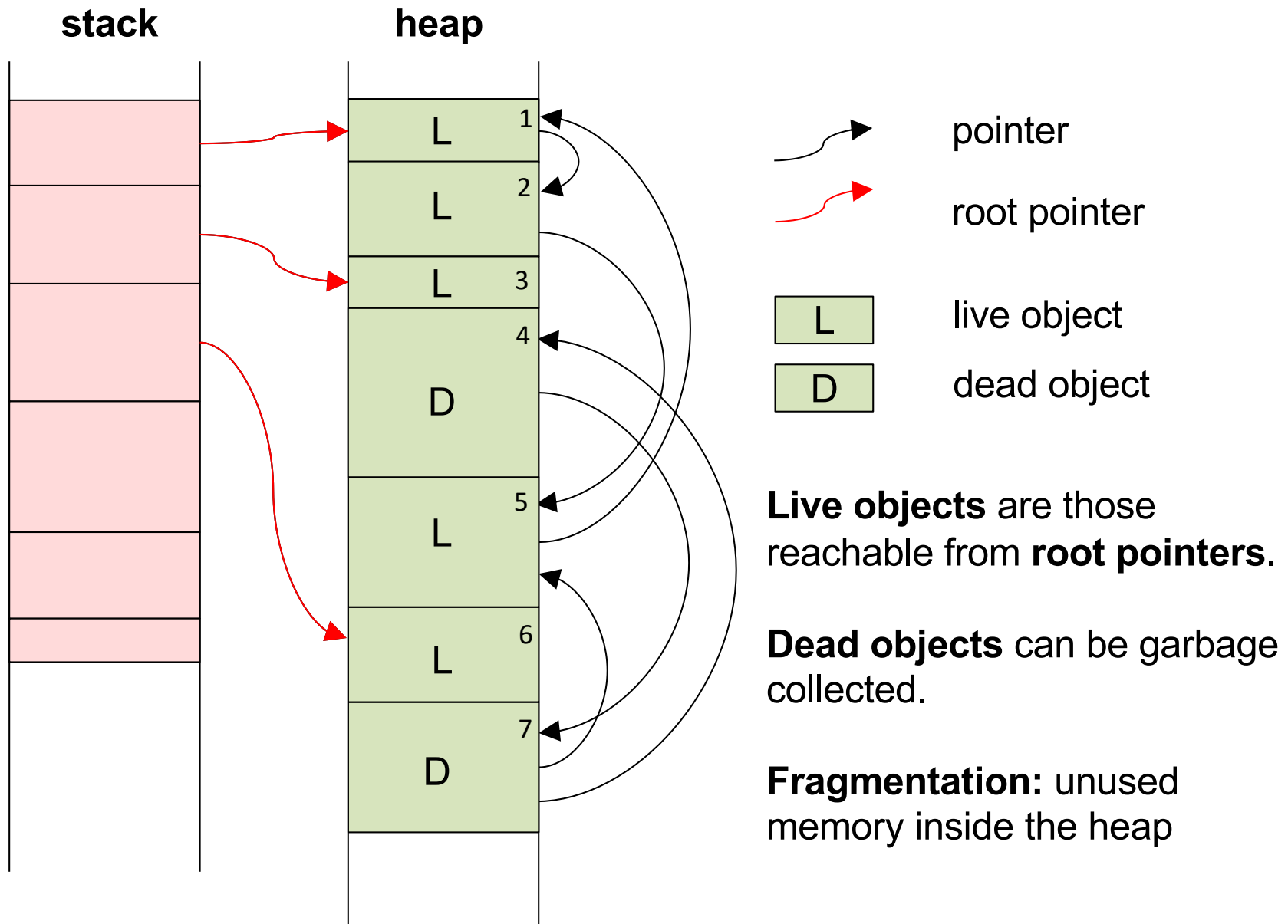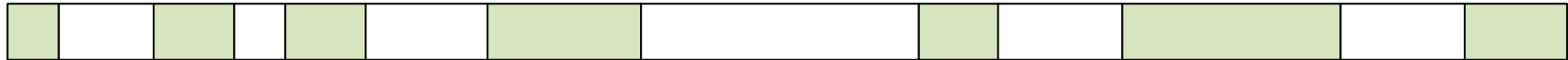
7

Handling threads

1 stack per thread

t1

frames

← **SP** – Stack Pointer for current thread

← **FP** – Frame Pointer for current thread

t2

frames

t3

frames

Heap, globals
and code shared
by all threads

heap

objects

global data

code

← **PC** – Program counter for current thread

Handling threads

1 stack per thread

t1
frames

SP – Stack Pointer for current thread
FP – Frame Pointer for current thread

t2
frames

t3
frames

Operating-system threads:
Used by Java, C, ..., typically expensive:
- Interrupt-driven context switch at any point during execution: Store/restore *all* registers on/from stack
- Constant stack size of 1 MB limits the number of threads
- 1000 threads = 1 GB RAM.

Heap, globals and code shared by all threads

heap
objects

global data

User-level threads:
Used by some languages, like Go, more light-weight:
- Cooperative context switch only at well-defined points: Sufficient to store/restore a few registers.
- Stacks can start small (2 KB) and grow at runtime.
- Can handle millions of threads.
- Cannot take advantage of multi-core

code

PC – Program counter for current thread

9

The heap

stack **heap**

1

2

3

4

5

6

7

pointer

The heap

stack   **heap**

→ pointer

→ root pointer

L live object

D dead object

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

Dead objects can be garbage collected.
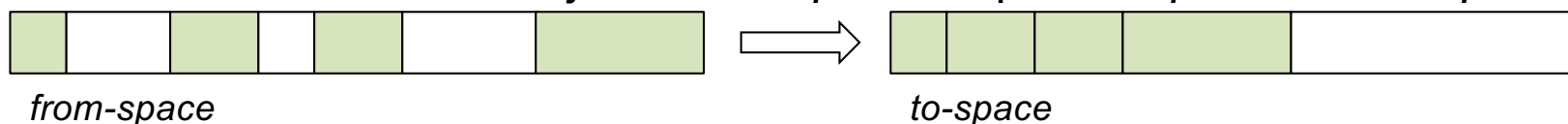
Fragmentation: unused memory inside the heap

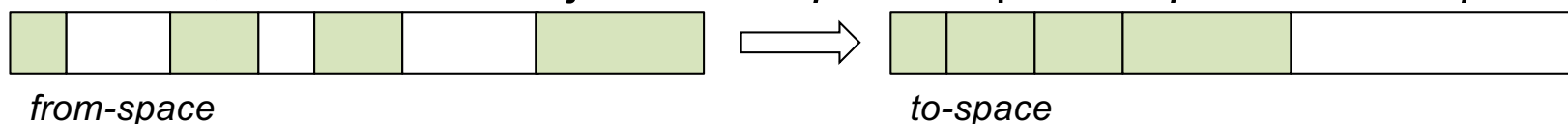# Major garbage collection techniques

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
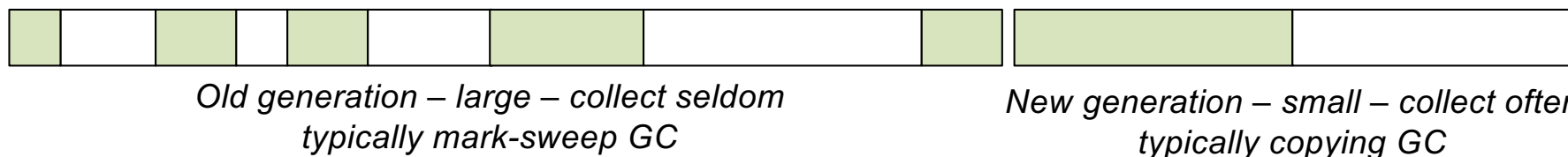
Mark-sweep GC: Follow all pointers and mark all live objects.
Sweep heap and collect free objects. Or **compact** the heap to avoid fragmentation.
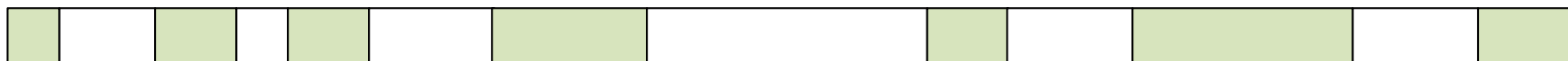
Copying GC: Divide heap into two spaces. Allocate new objects in *from-space*.
When full, move all live objects to *to-space*. Flip *from-space* and *to-space*.

from-space   *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*.

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

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*.

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.

16

Mark-Sweep GC

Mark-Sweep-Compact GC
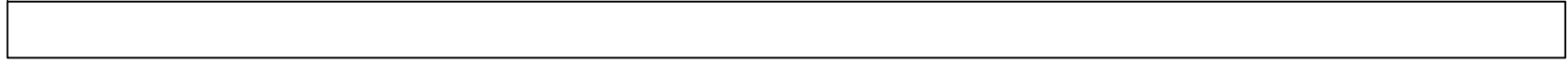
HP

1. The program starts. The heap is empty.

Mark-Sweep-Compact GC

HP

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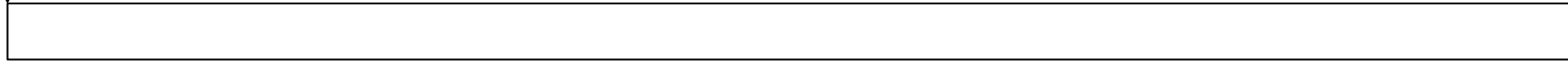
2. The program runs and allocates new objects until the heap is full. Program stops.   HP

Mark-Sweep-Compact GC
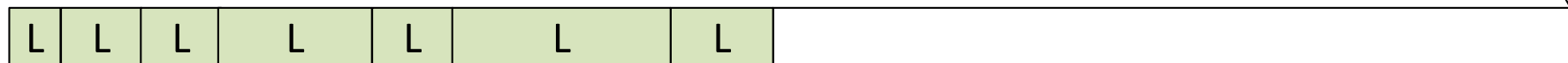
HP

1. The program starts. The heap is empty.

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

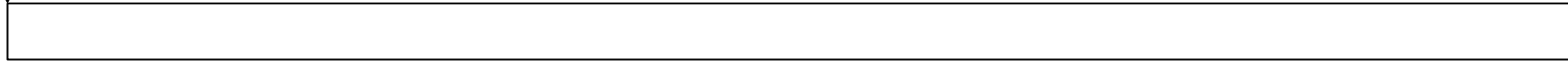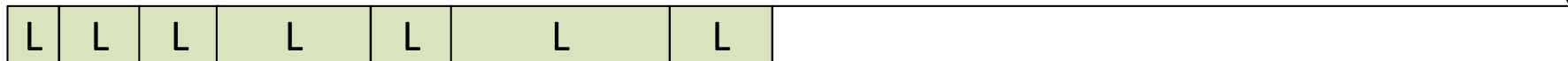3. Mark phase: The GC follows pointers from the roots and marks all live objects. HP

| L | | L | L | | L | | L | | L | | L |

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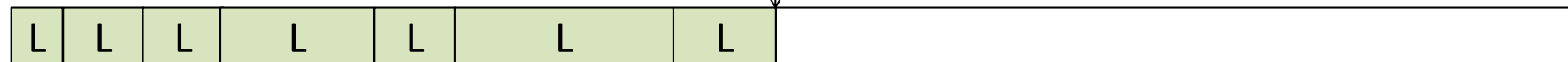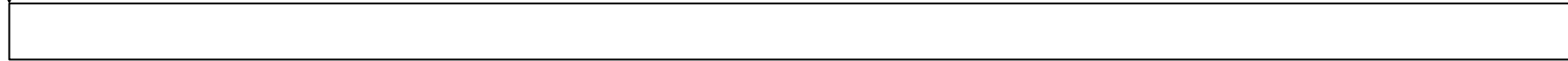
3. Mark phase: The GC follows pointers from the roots and marks all live objects. HP

| L | | L | L | | L | | | L | | L | | L |

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

| L | L | L | L | L | L | L |

Mark-Sweep-Compact GC

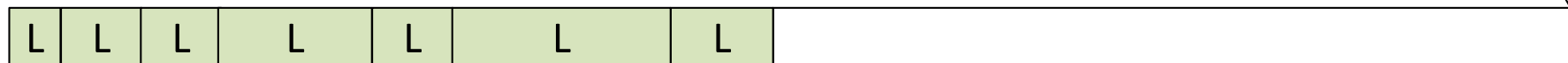HP 1. The program starts. The heap is empty.

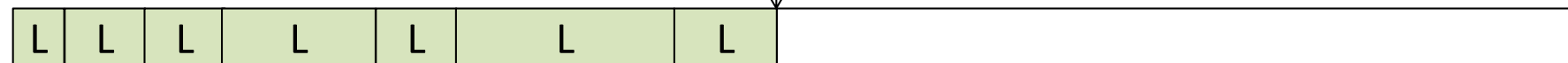2. The program runs and allocates new objects until the heap is full. Program stops. HP

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

| L | | L | L | | L | | | L | | L | | L |

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

| L | L | L | L | L | L | L |

5. The program continues to run. HP

| L | L | L | L | L | L | L |

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

| L | | L | L | | L | | L | L | | L |

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

| L | L | L | L | L | L | L |

5. The program continues to run.   HP

| L | L | L | L | L | L | L |

+ 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*.
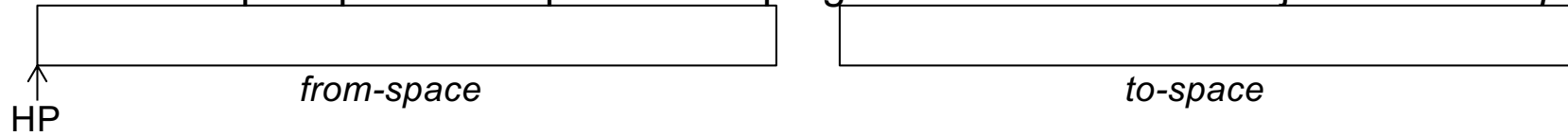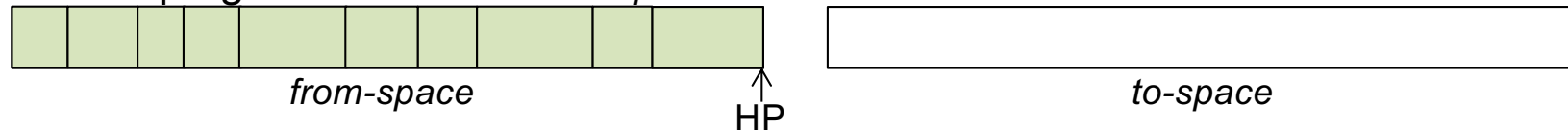
from-space   *to-space*

HP

Copying GC

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

from-space

HP

to-space

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

from-space
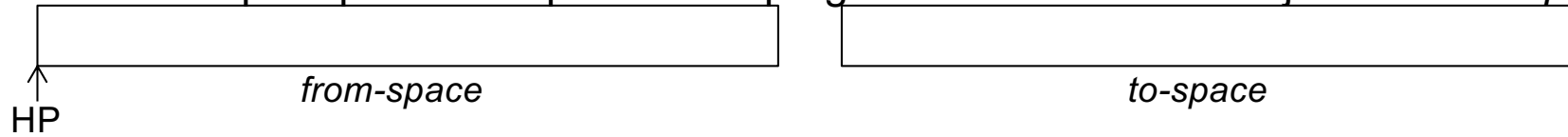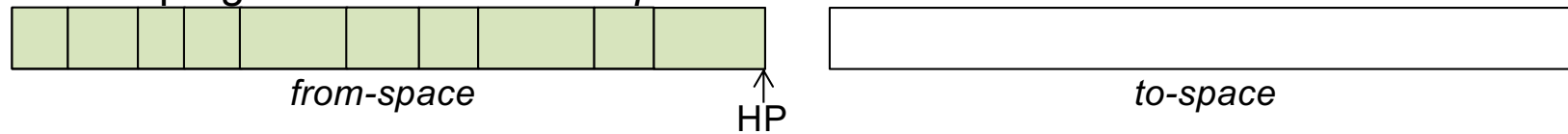
HP

to-space

Copying GC

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

from-space

HP

to-space

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

from-space

HP

to-space

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

	L		L	L		L			

from-space

HP

L	L	L	L	

to-space

Copying GC

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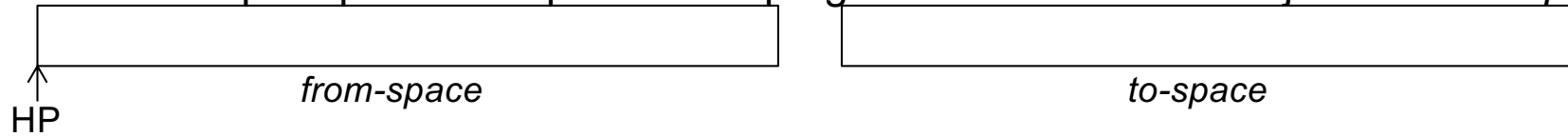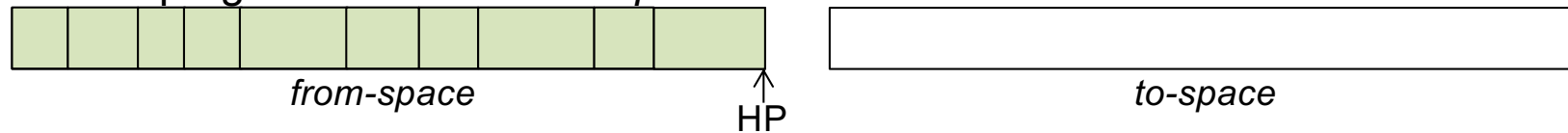
from-space

to-space

HP

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

from-space

to-space

HP

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

| L | | L | L | | L | | | | L | L | L | L |

from-space

to-space

HP

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

to-space

| L | L | L | L |

from-space

HP

Copying GC

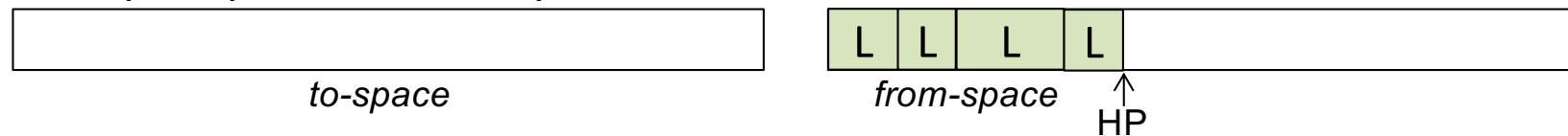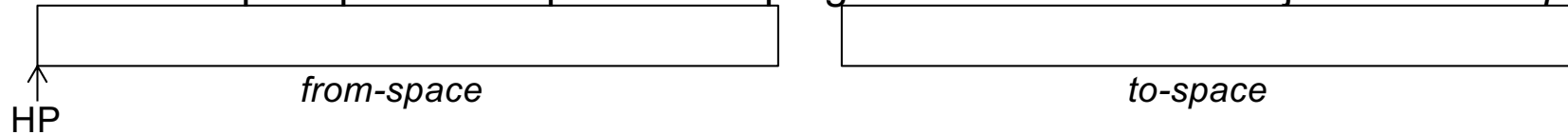1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.

from-space

to-space

HP

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

from-space

to-space

HP

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

| | L | | L | L | | L | | | |

from-space

| L | L | L | L | |

to-space

HP

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

to-space

| L | L | L | L | |

from-space

HP

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

to-space

| L | L | L | L | |

from-space

HP
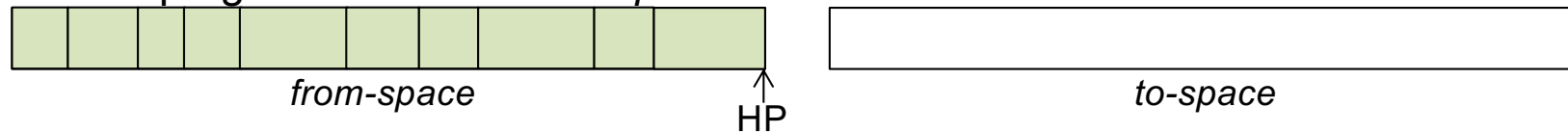
Copying GC

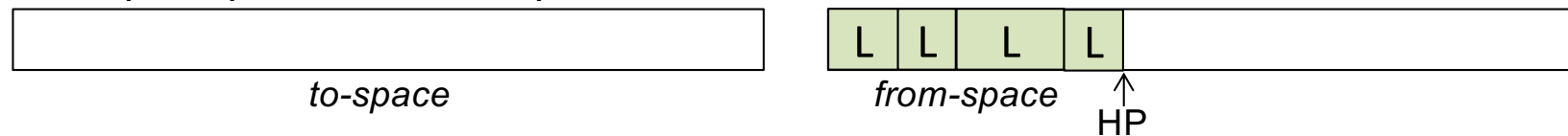1. The heap is split in two spaces. The program starts. Allocates objects in *from-space*.
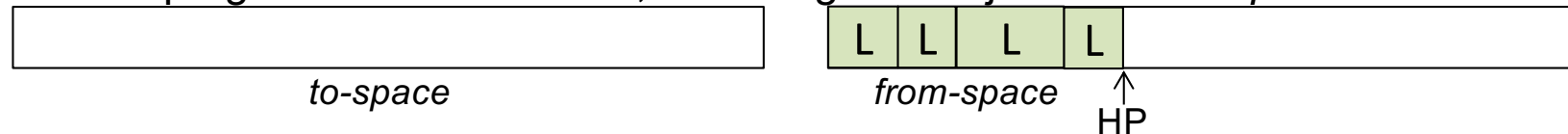

↑
HP
from-space   *to-space*

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

from-space   ↑   *to-space*
                         HP

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

| L | | L | L | | L | | |

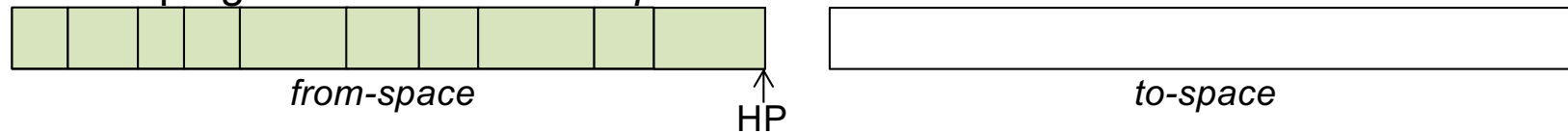*from-space*   ↑   *to-space*
                         HP

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

to-space   *from-space* ↑
                                        HP

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

to-space   *from-space* ↑
                                        HP

\+ Efficient – looks only at live objects (which are few). Avoids fragmentation.
\- Fairly long pauses at GC ("stop the world"). Uses a lot 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.

from-space   *to-space*

old generation, mark-sweep   *new generation, copying*

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

from-space   *to-space*

old generation   *new generation*

Generational GC

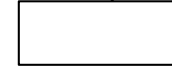1. The heap is split into a large old and a small new generation.

old generation   *new generation*

2. Typical algorithms.

from-space   *to-space*

old generation, mark-sweep   *new generation, copying*

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

from-space   *to-space*

old generation   *new generation*

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

from-space   *to-space*

old generation   *new generation*

Generational GC

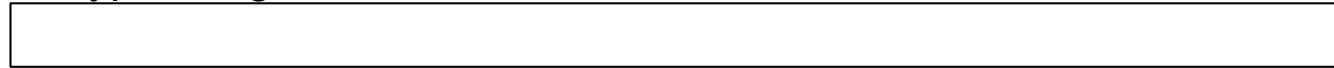1. The heap is split into a large old and a small new generation.
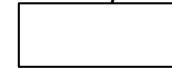
old generation

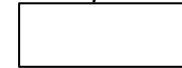*new generation*

2. Typical algorithms.
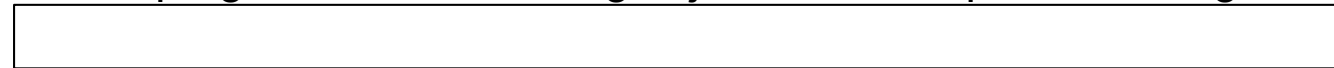
old generation, mark-sweep
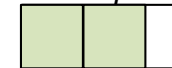
from-space *to-space*

new generation, copying

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

old generation

from-space *to-space*
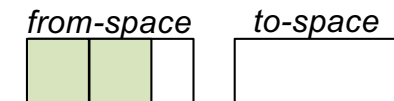
new generation

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

old generation
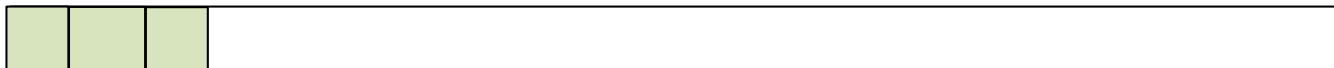
from-space *to-space*

new generation

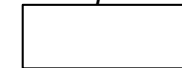5. Most objects die very young. Few survive to be tenured.

old generation
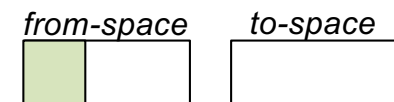
from-space *to-space*

new generation

36

Generational GC

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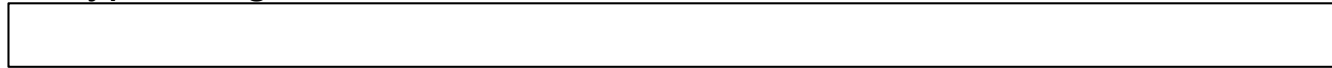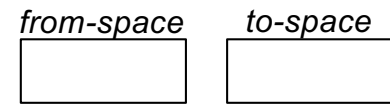
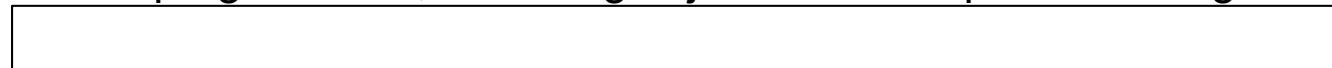old generation   new generation

2. Typical algorithms.

from-space   to-space

old generation, mark-sweep   new generation, copying

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

from-space   to-space

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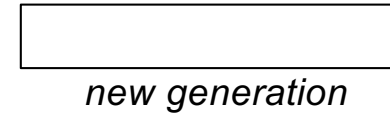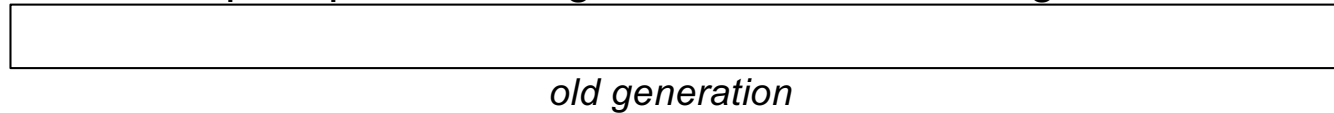
4. Objects surviving a few GCs in new gen are *tenured* – moved to old
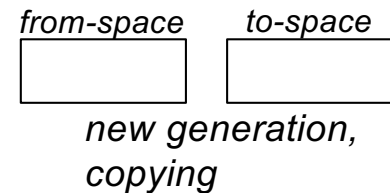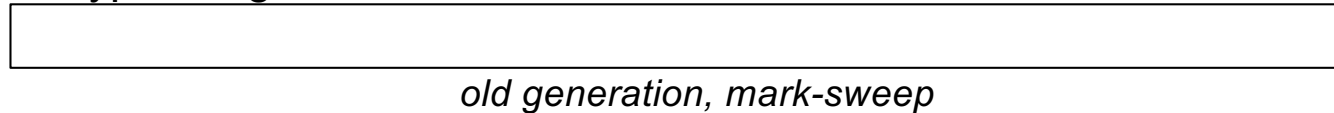
from-space   to-space

old generation   new generation

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

from-space   to-space
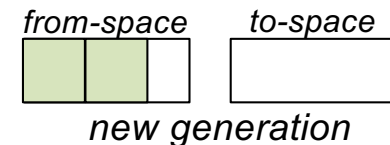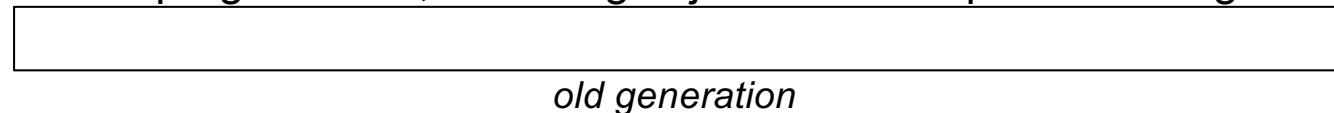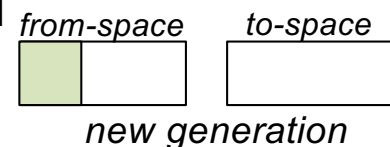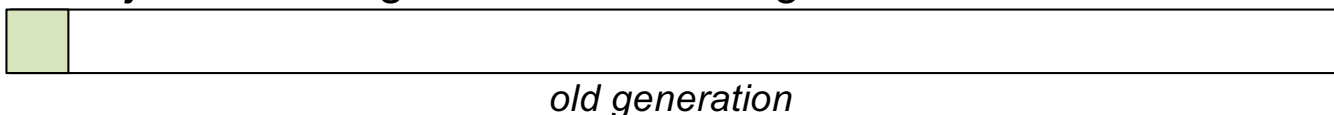
old generation   new generation
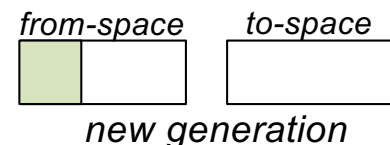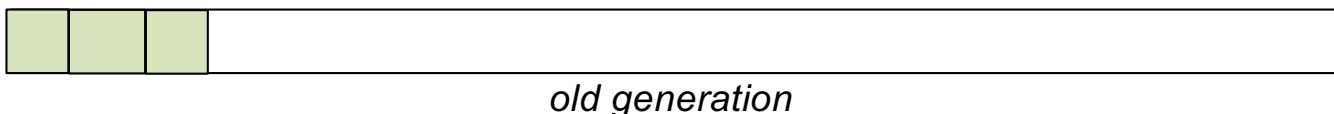
\+ GC in new generation is quick (small area). Usually only short pauses.
\+ Old generation grows very slowly. Avoids fragmentation.

Reference-counting GC

Reference-counting GC

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

root

| 1 | 1 | 2 | 1 | 1 | 1 | 2 | 2 | 1 | | |

Reference-counting GC

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

root

| 1 | 1 | 2 | 1 | 1 | 1 | 2 | 2 | 1 | |

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

root

| 1 | 1 | 2 | 1 | 0 | 1 | 2 | 2 | 1 | |
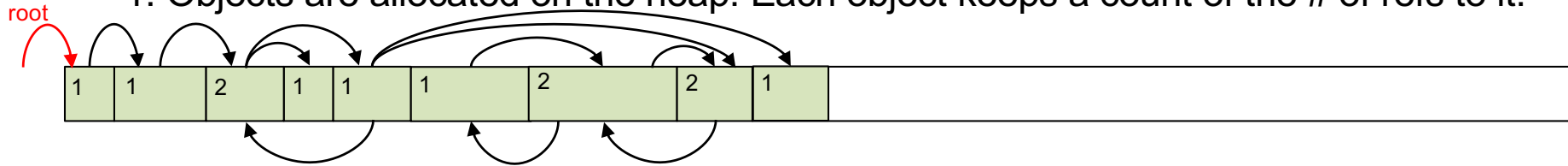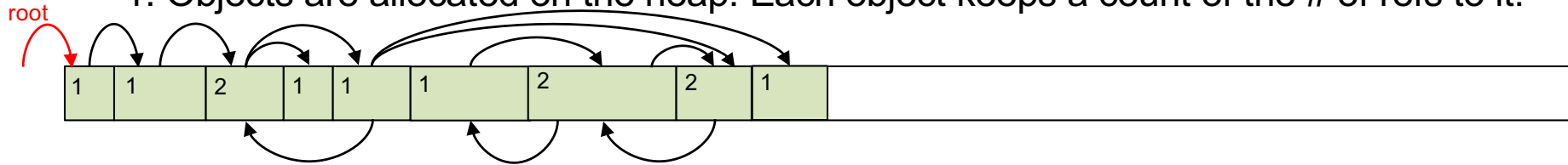
Reference-counting GC

1. Objects are allocated on the heap. Each object keeps a count of the # of refs to it.
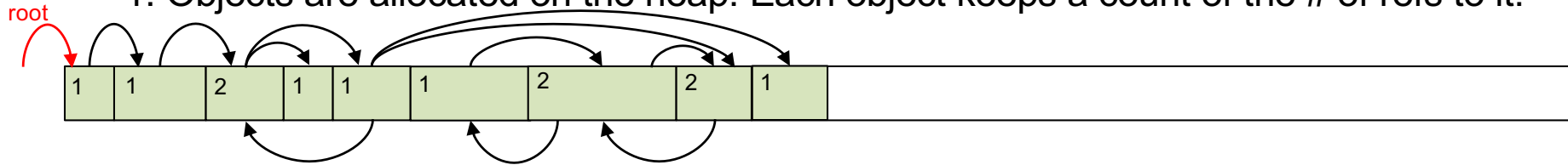
2. When a reference is changed, counts are updated.
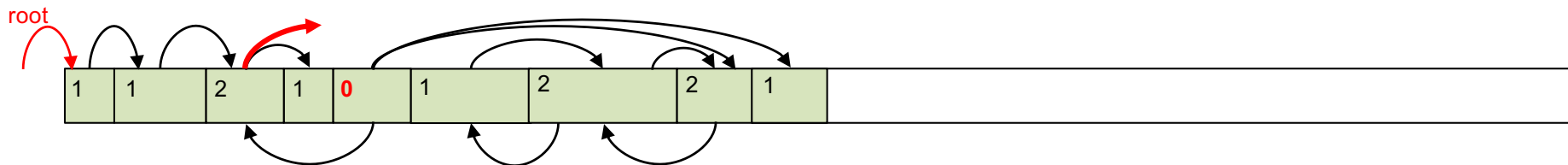
3. When a count goes to zero, the object is deallocated. Its references are followed, and counts are decremented, and may go to zero. The process continues recursively.

Reference-counting GC

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

root

| 1 | 1 | 2 | 1 | 1 | 1 | 2 | 2 | 1 | | |

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

root

| 1 | 1 | 2 | 1 | 0 | 1 | 2 | 2 | 1 | | |

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.

root

| 1 | 1 | 1 | 1 | 0 | 1 | 2 | 1 | 0 | | |
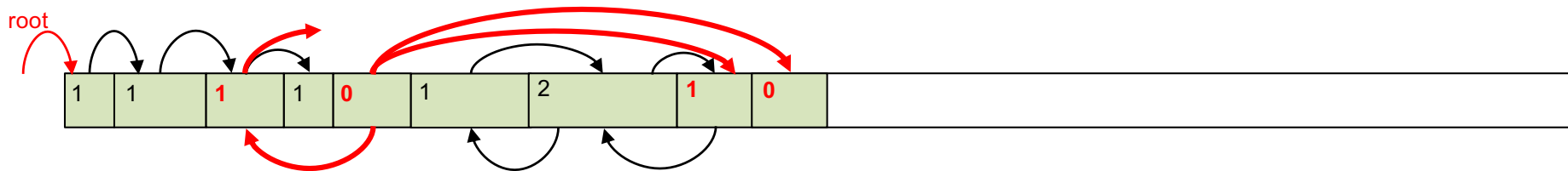
+ GC is incremental (a little work is done at each assignment)
- Very inefficient (because work is done at each assignment)
- Cyclic structures are not garbage collected.
- No compaction – the heap becomes fragmented.

Fields and dynamic dispatch in OO

Typical runtime structures for objects

activation frames

temps vars dynlink
retaddr this args temps vars dynlink

Typical runtime structures for objects

activation frames

objects

temps vars dynlink
retaddr this args temps vars dynlink

class field1 field2 ...

Typical runtime structures for objects

activation frames   **objects**   **class descriptors**

temps
vars
dynlink

retaddr
this
args
temps
vars
dynlink

class
field1
field2
...

super
method1
method2
...
static var1
static var2
...

Typical runtime structures for objects

activation frames   **objects**   **class descriptors**   **code of method**

activation frames	objects	class descriptors	code of method

temps
vars
dynlink
	class
	field1
	field2
	...
		super
		method1
		method2
		...
		static var1
		static var2
		...
			m:
			  pushq ...
			  movq ...
			  subq ...
			  ...

retaddr this args temps vars dynlink			

Typical runtime structures for objects

activation frames   **objects**   **class descriptors**   **code of method**

activation frames
temps vars dynlink
retaddr this args temps vars dynlink

objects
class field1 field2 ...

class descriptors
super method1 method2 ... static var1 static var2 ...

```
m:
  pushq ...
  movq ...
  subq ...
  ...
```

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.

48

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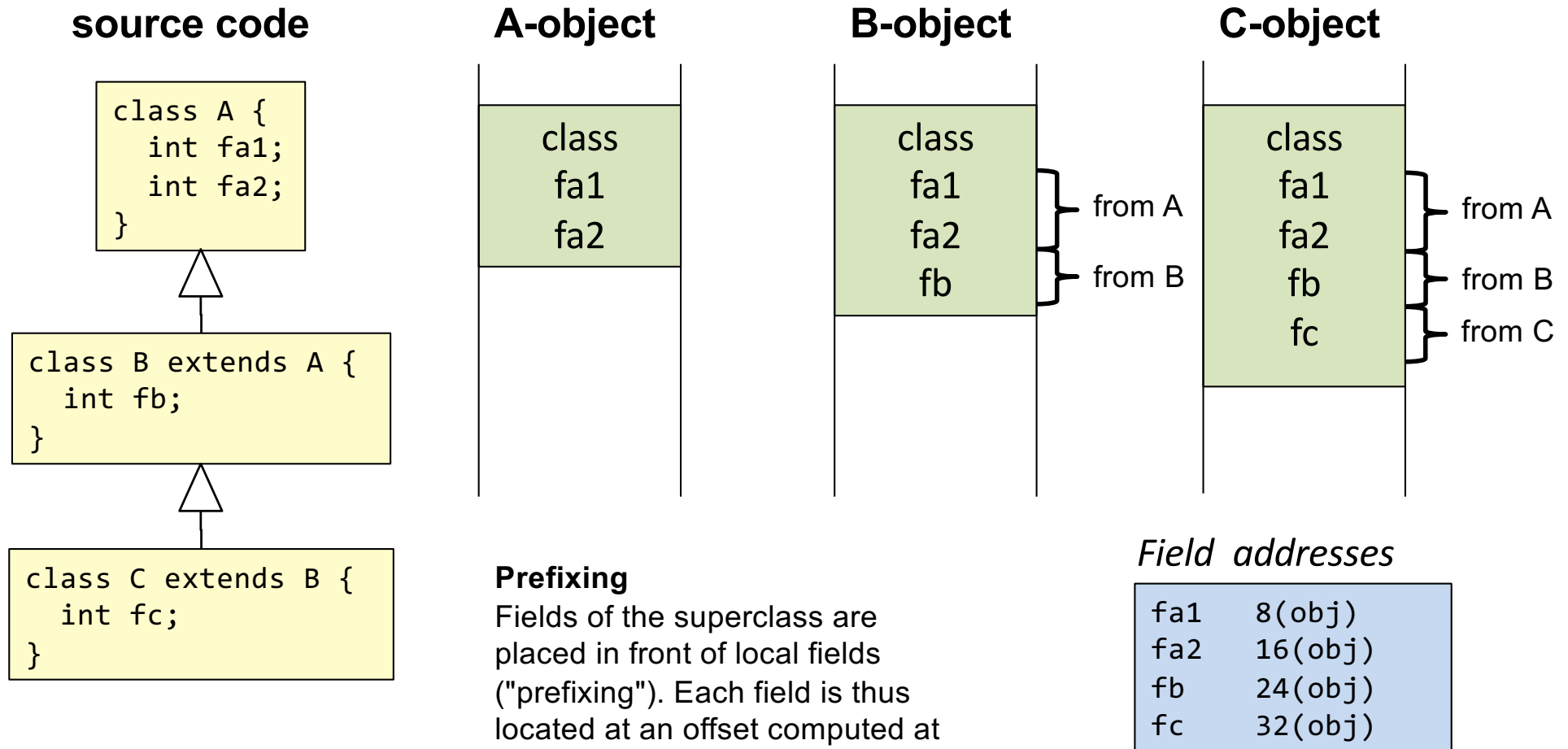
Prefixing
Fields of the superclass are placed in front of local fields ("prefixing"). Each field is thus located at an offset computed at compile time, regardless of the dynamic type of the object.

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();
}
```

m frame

| temps |
| vars |
| %rbp → | dynlink |

| retaddr |
| this |
| args |
| temps |
| vars |
| dynlink |

p frame

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();
}
```

m frame

| temps |
| vars |
| dynlink |

%rbp →

| retaddr |
| this |
| args |
| temps |
| vars |
| dynlink |

p frame
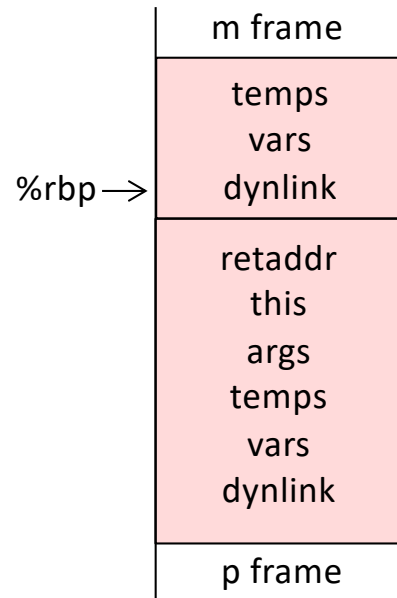
B object

| class |
| fa1 |
| fa2 |
| fb |

C object

| class |
| fa1 |
| fa2 |
| fb |
| fc |
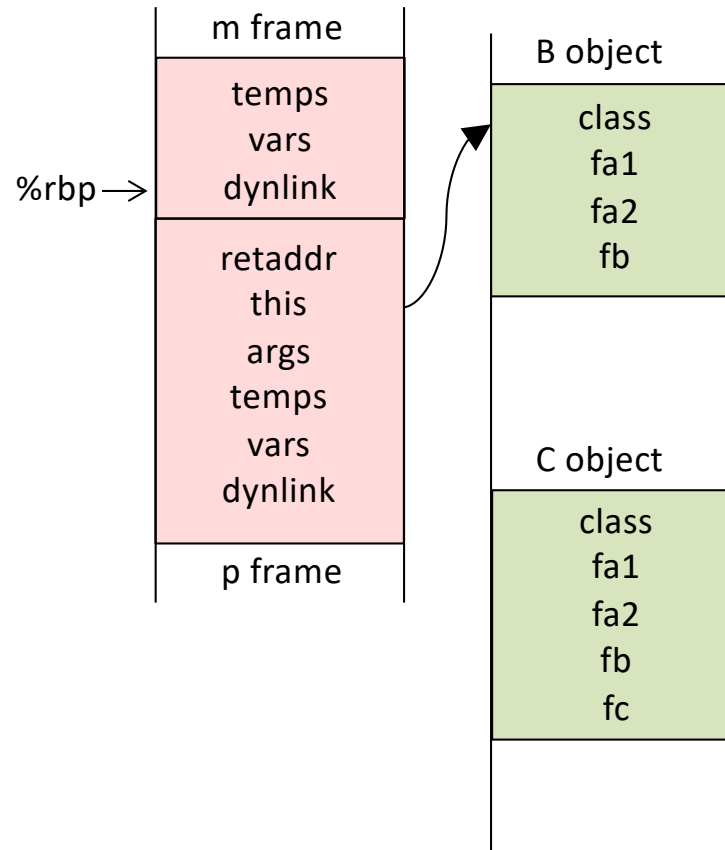
The code for m knows the static type of the object (A), but not the dynamic type (B or C in this case).

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
```
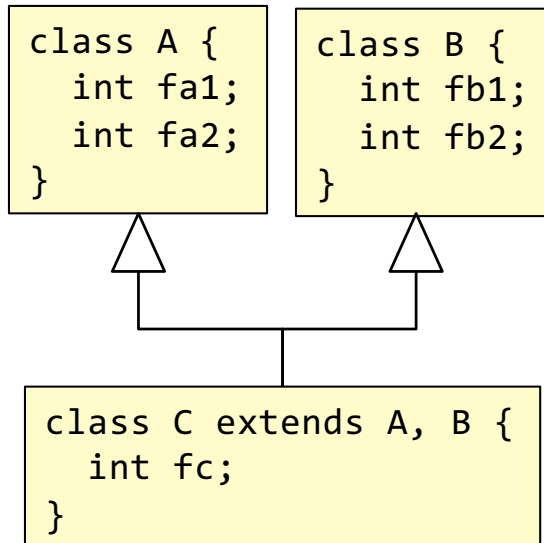
53

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

m frame
rA
rB
rC
dynlink

54

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

m frame

rA
rB
rC
dynlink

C object

class
fa1
fa2
class
fb1
fb2
fc

C descr

...

B descr

...

interior pointer

subobject

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


m frame
rA
rB
rC
dynlink

C object
class
fa1
fa2
class
fb1
fb2
fc

C descr
...

B descr
...
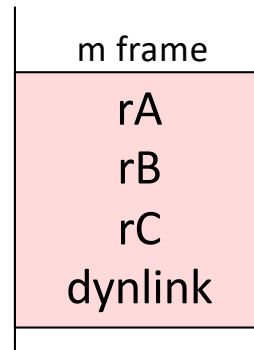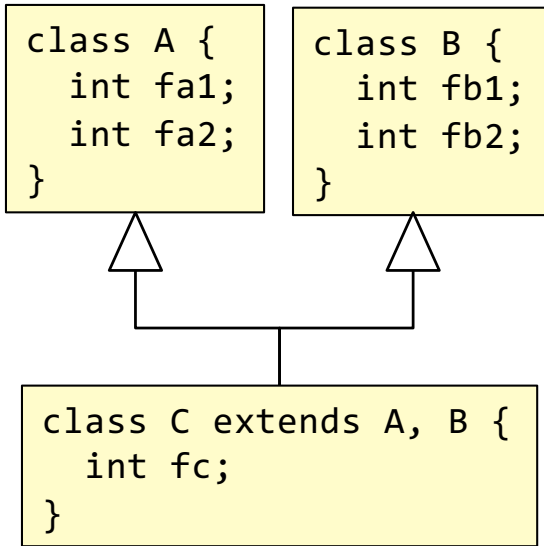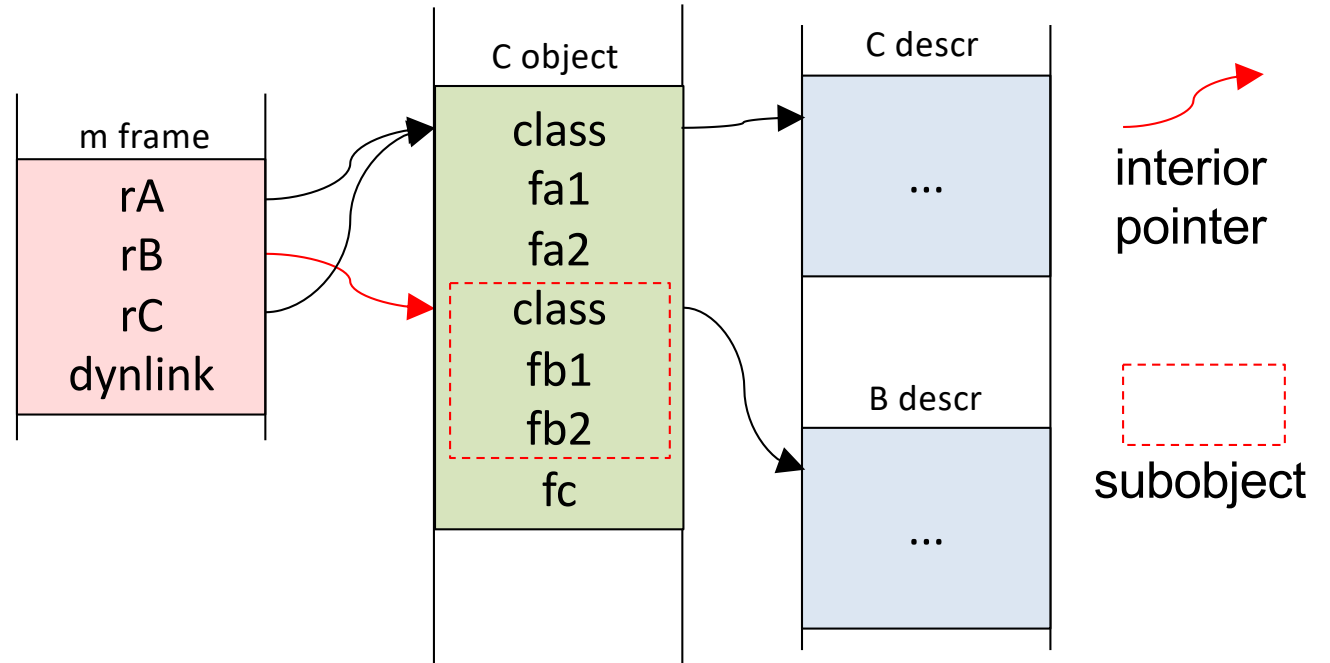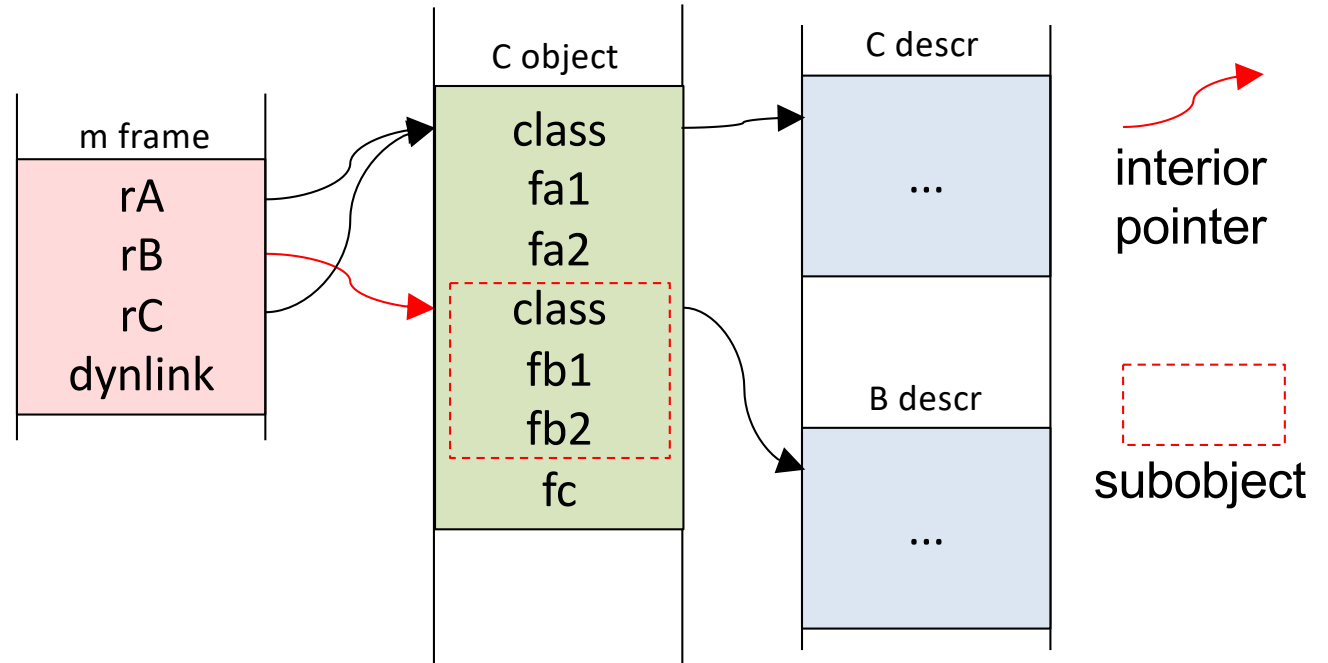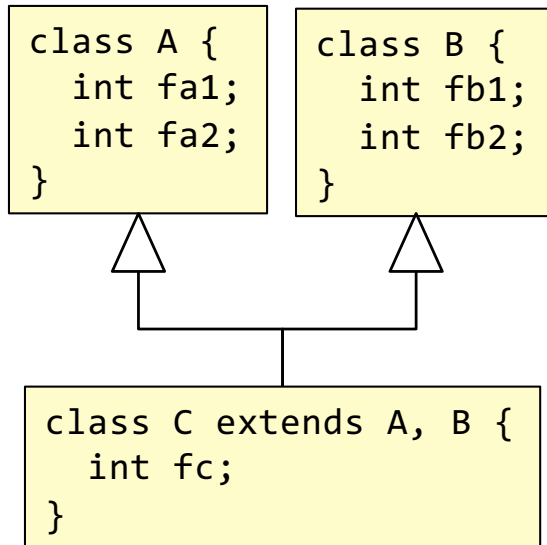
interior pointer

subobject

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.

56

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)

source code

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

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

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

A object

| class ... |

B object

| class ... |

C object

| class ... |

A descr

| ... |

B descr

| ... |

C descr

| ... |

code

```
A-ma:
  ...


B-mb:
  ...


C-ma:
  ...
```

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() {
    ...
  }
}
```

A object

| class ... |

B object

| class ... |

C object

| class ... |

A descr

| ... |

B descr

| ... |

C descr

| ... |

code

```
A-ma:
...



B-mb:
...



C-ma:
...
```

Two common implementation methods:
- **Virtual tables**. Uses static typing. Simula, C++, ...
- **Hash tables**. For dynamic typing. Smalltalk, Python, JavaScript, Objective-C, ...

60

Virtual table dynamic dispatch

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

source code

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

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

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

A object

| class ... |

B object

| class ... |

C object

| class ... |

A descr

| super |
| ma |

B descr

| super |
| ma |
| mb |

C descr

| super |
| ma |
| mb |

code

```
A-ma:
...



B-mb:
...



C-ma:
...
```

Virtual table dynamic dispatch

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

source code

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

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

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

A object

| class ... |

B object

| class ... |

C object

| class ... |

A descr

| super |
| ma |

B descr

| super |
| ma |
| mb |

C descr

| super |
| ma |
| mb |

code

```
A-ma:
...
```

```
B-mb:
...
```

```
C-ma:
...
```

Method addresses

| ma | 8(class) |
| mb | 16(class) |
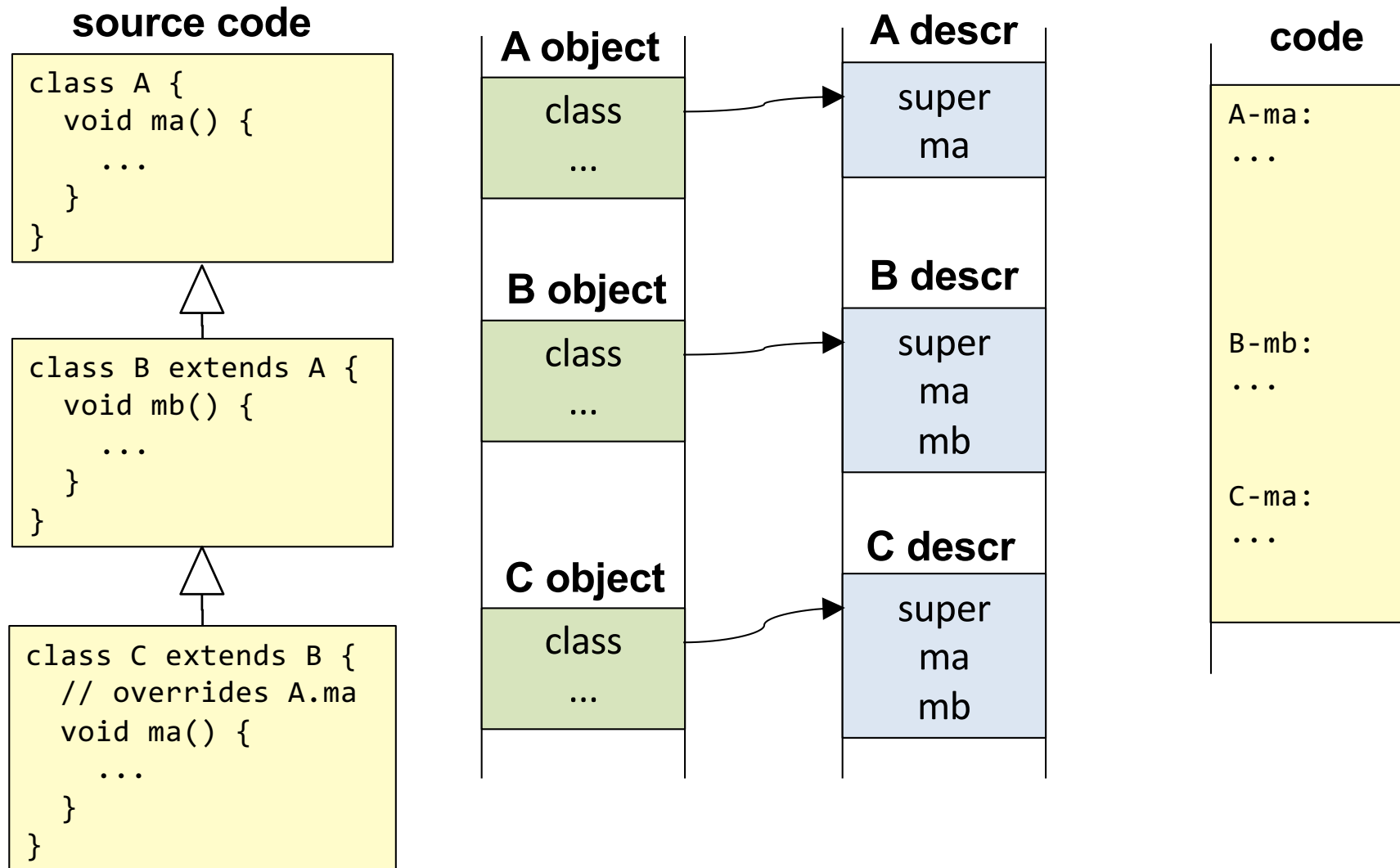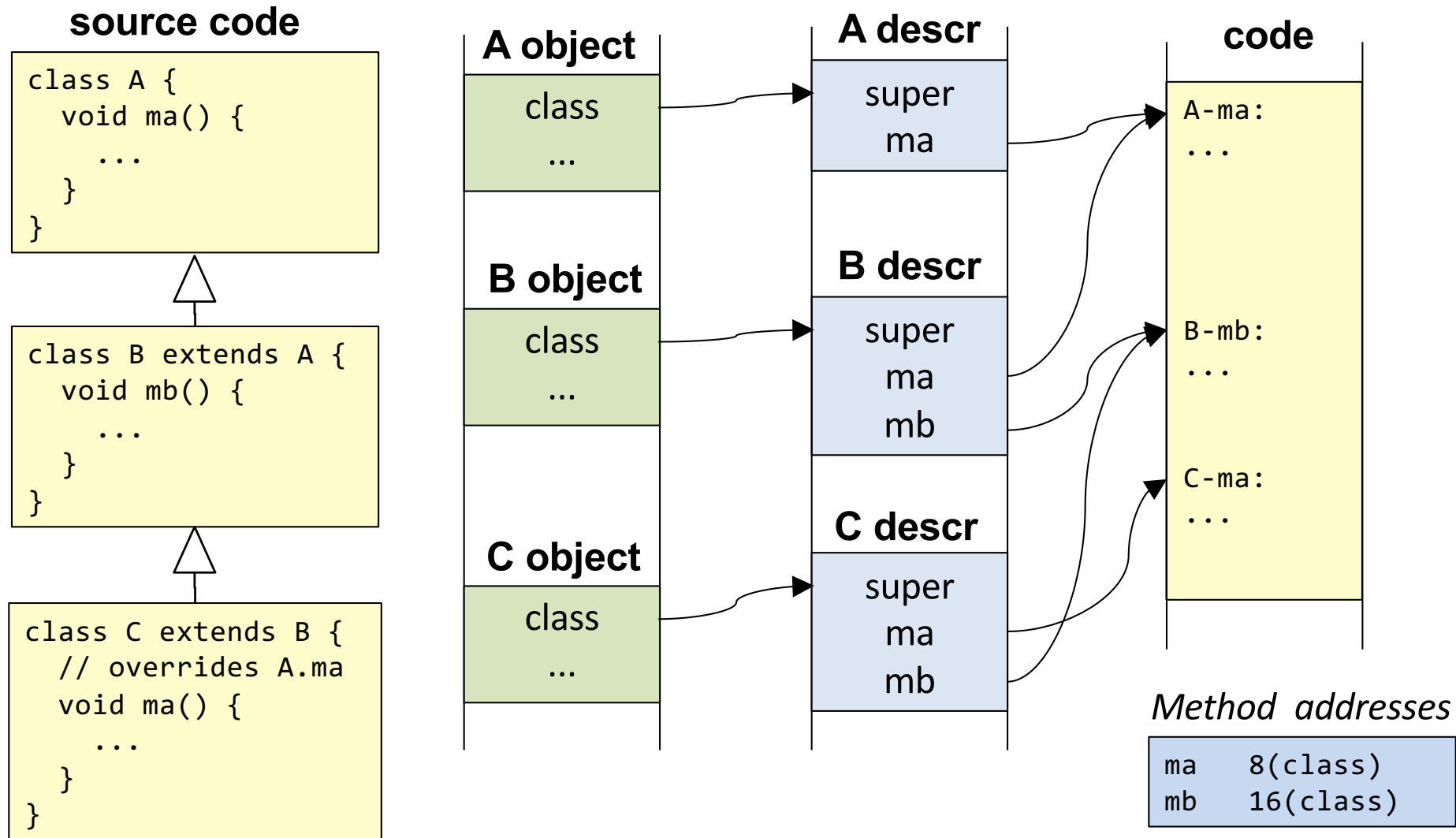
Virtual tables
Class descriptor contains *virtual table* (often called "vtable").
Pointers to superclass methods are placed in front of locally declared methods ("prefixing").
Each method pointer is located at an offset computed at compile time, using the static type.
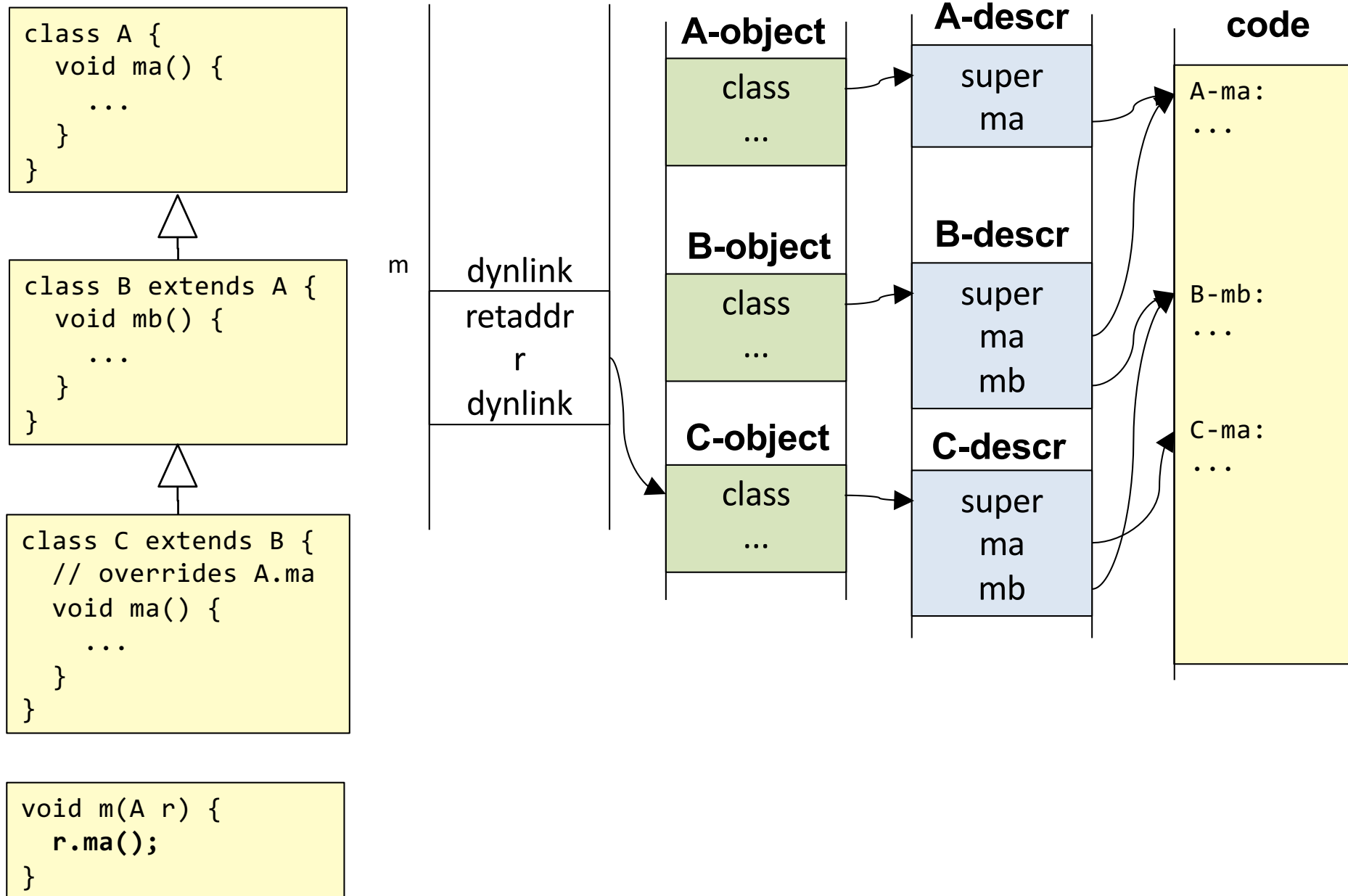
62

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();
}
```
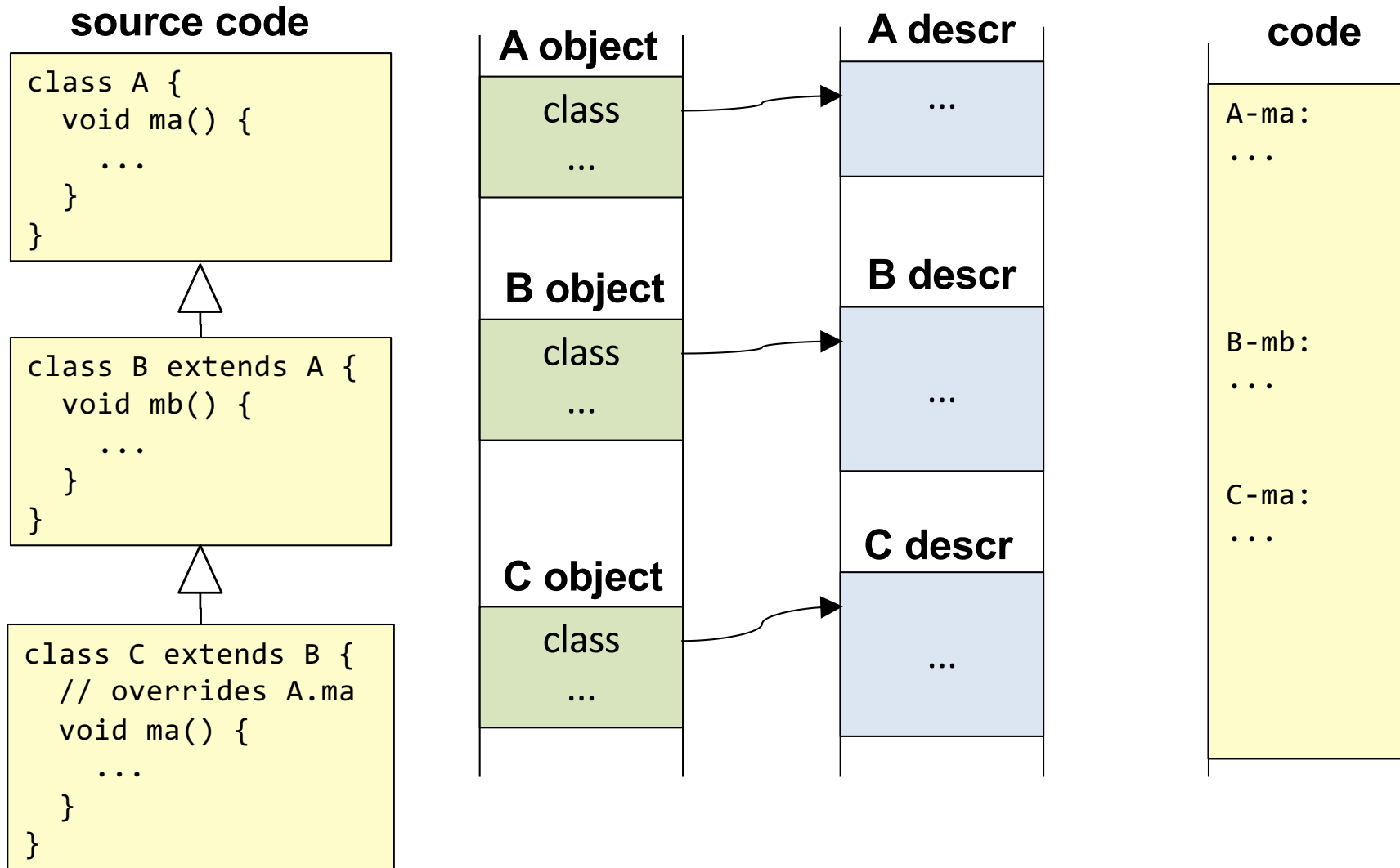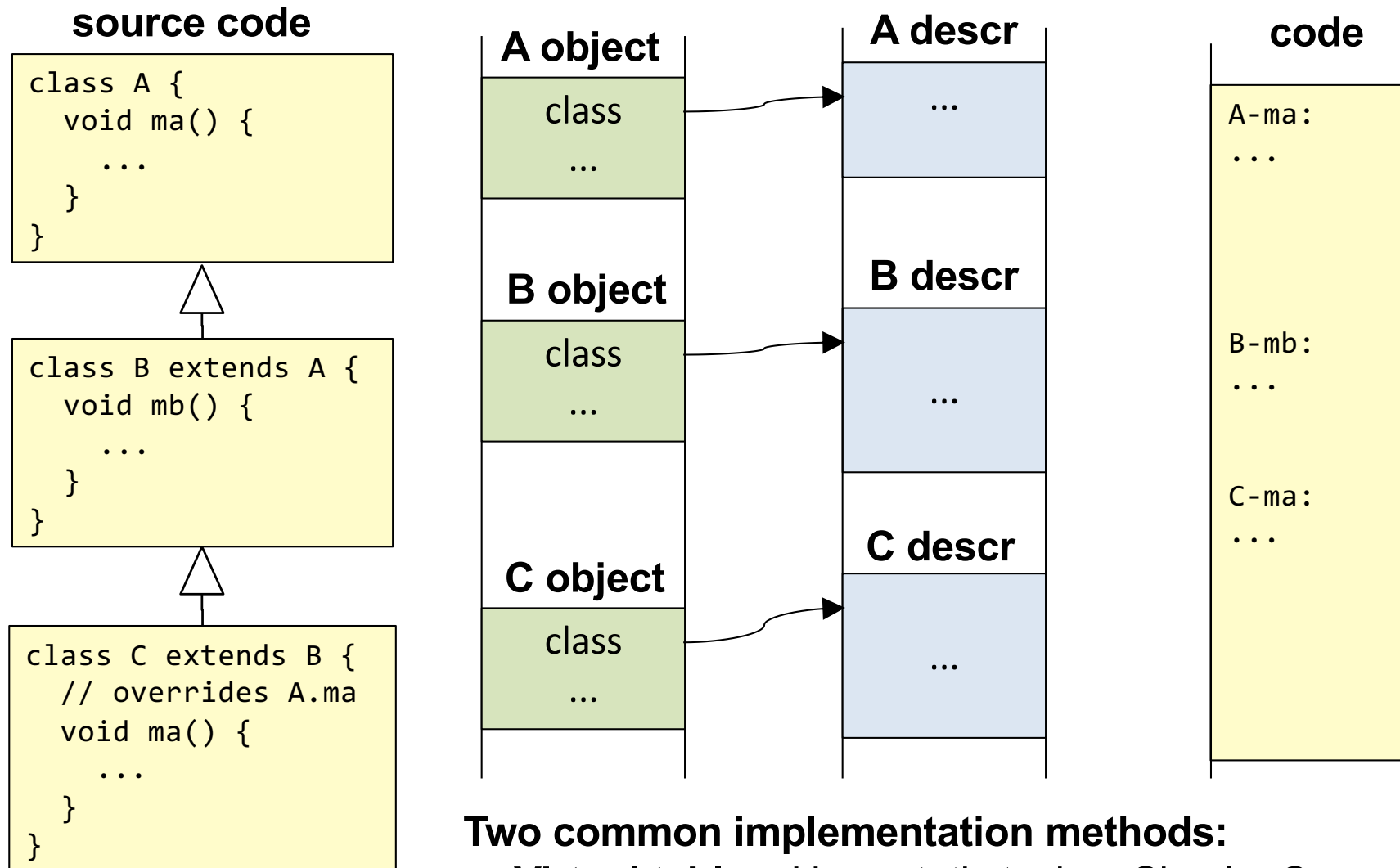
m

| dynlink |
| retaddr r |
| dynlink |

A-object

| class ... |

B-object

| class ... |

C-object

| class ... |

A-descr

| super |
| ma |

B-descr

| super |
| ma |
| mb |

C-descr

| super |
| ma |
| mb |

code

```
A-ma:
...


B-mb:
...


C-ma:
...
```

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();
}
```

C-ma | dynlink

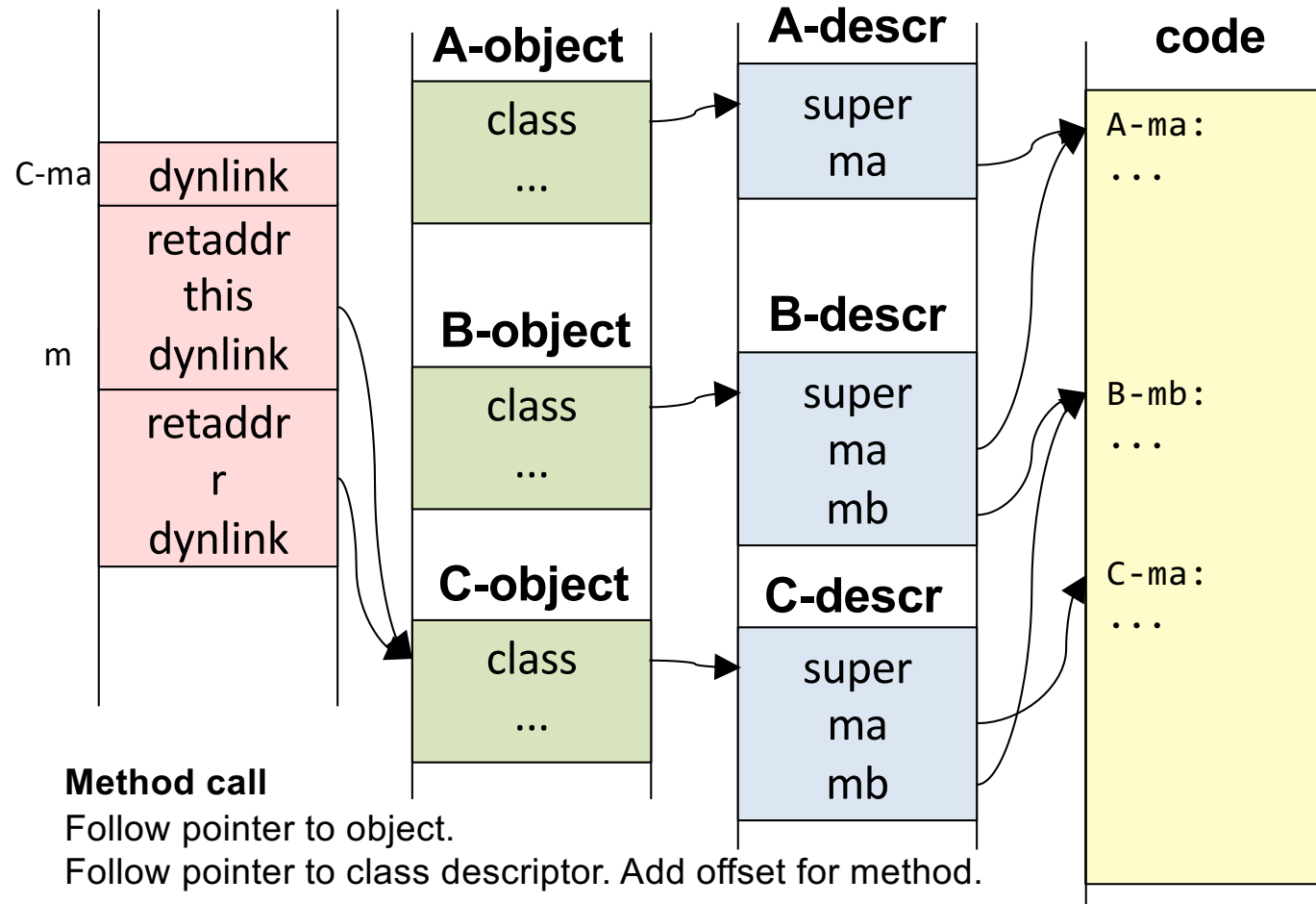m | retaddr
this
dynlink

retaddr
r
dynlink

A-object
class
...

B-object
class
...

C-object
class
...

A-descr
super
ma

B-descr
super
ma
mb

C-descr
super
ma
mb

code
A-ma:
...

B-mb:
...

C-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
```

64

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();
  }
}
```

m | dynlink
  | retaddr
  | this
  | r
  | dynlink
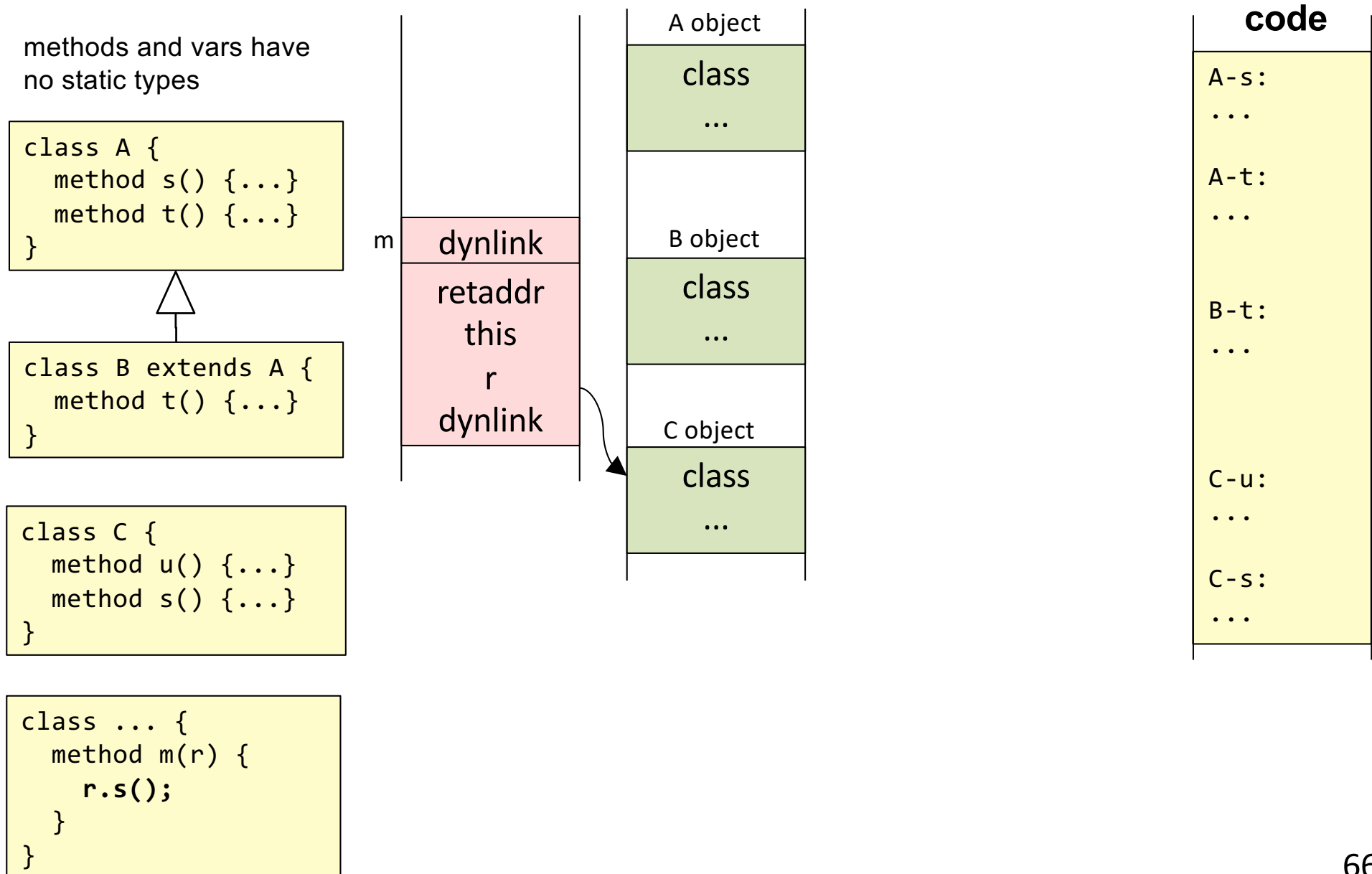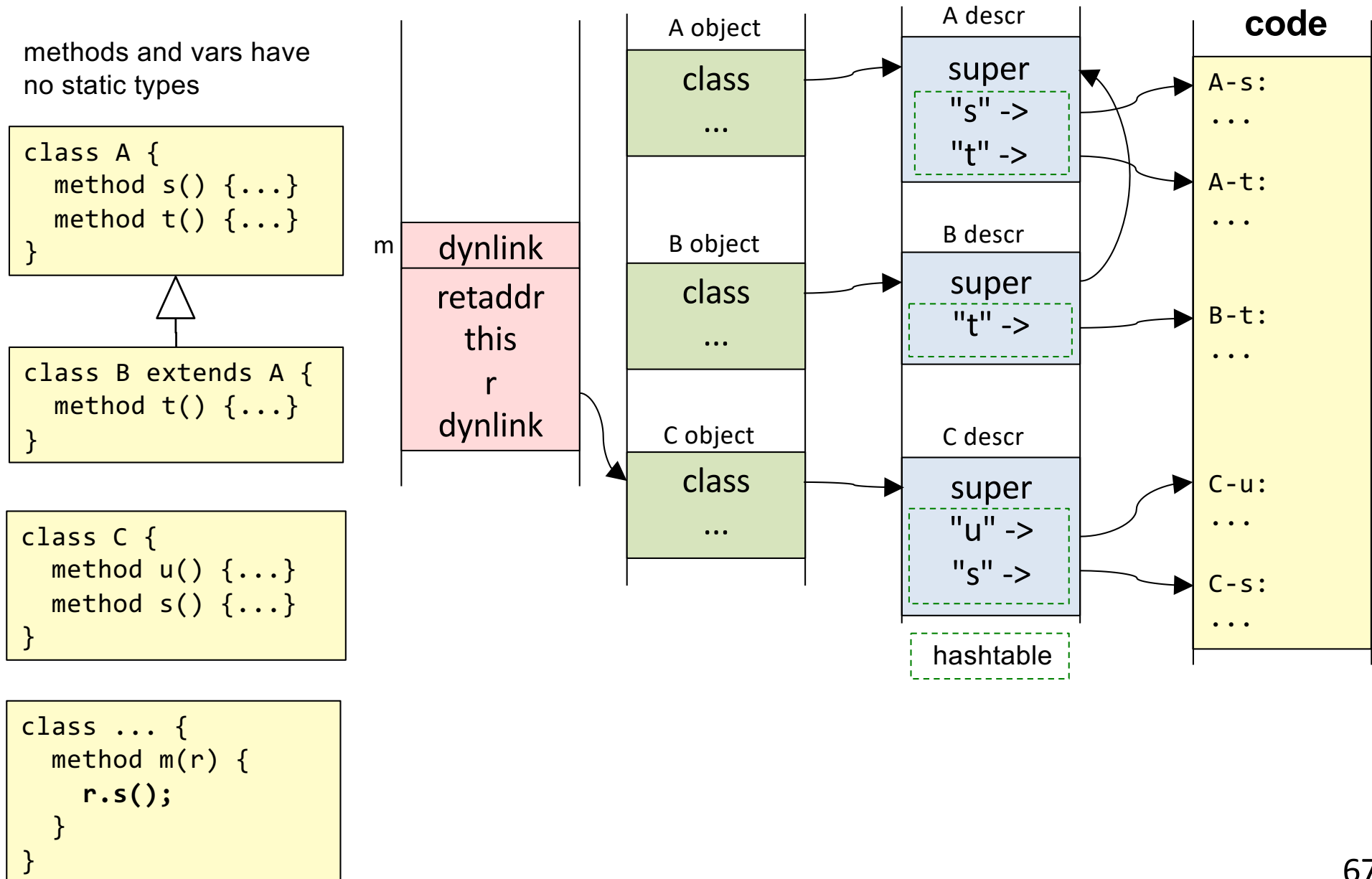
code

A-s:
...

A-t:
...

B-t:
...

C-u:
...

C-s:
...

65

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();
  }
}
```

m

dynlink
retaddr
this
r
dynlink

A object

class
...

B object

class
...

C object

class
...

code

A-s:
...

A-t:
...

B-t:
...

C-u:
...

C-s:
...

66

Hash table dynamic dispatch

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

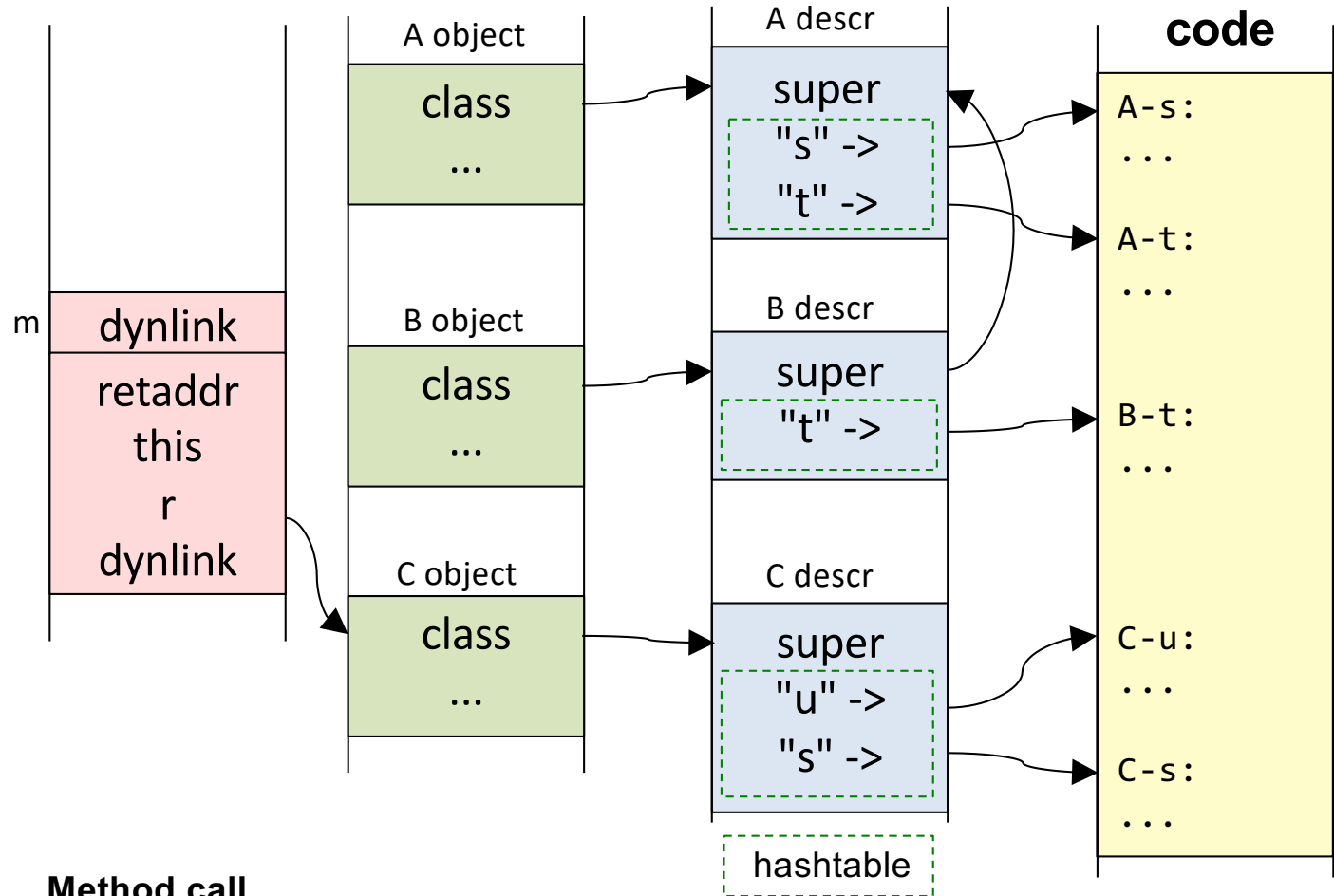methods and vars have
no static types

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

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

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

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

m | dynlink
| retaddr
| this
| r
| dynlink

A object
class
...

B object
class
...

C object
class
...

A descr
super
"s" ->
"t" ->

B descr
super
"t" ->

C descr
super
"u" ->
"s" ->

hashtable

code

A-s:
...

A-t:
...

B-t:
...

C-u:
...

C-s:
...

67

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

m

dynlink
retaddr
this
r
dynlink

A object

class
...

B object

class
...

C object

class
...

A descr

super
"s" ->
"t" ->

B descr

super
"t" ->

C descr

super
"u" ->
"s" ->

hashtable

A-s:
...

A-t:
...

B-t:
...

C-u:
...

C-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.

68

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

common subexpression elimination

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

$\Longrightarrow$

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

Example local optimizations

common subexpression elimination

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

$\Longrightarrow$

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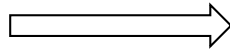
```
t = b * c;
a = t + d;
e = f + t;
```

```
int a = 37;
return a + 5;
```
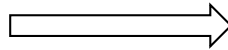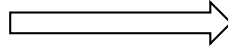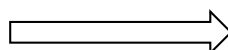
constant propagation ⟹

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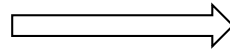
```
int a = 37;
return 37 + 5;
```

Example local optimizations

common subexpression elimination

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

⟹

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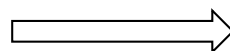
constant propagation

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

⟹

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

constant folding

```
int a = 37;
return 37 + 5;
```
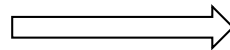
⟹
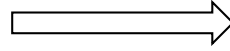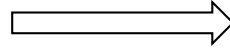
```
int a = 37;
return 42;
```

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

Example local optimizations

common subexpression elimination

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

$\Longrightarrow$

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

constant propagation

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

$\Longrightarrow$

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

constant folding

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

$\Longrightarrow$

```
int a = 37;
return 42;
```
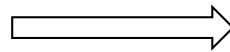
dead code elimination

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

$\Longrightarrow$
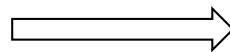
```
return 42;
```

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

Example local optimizations

common subexpression elimination

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

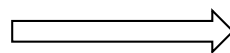$\Longrightarrow$

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

constant propagation

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

$\Longrightarrow$

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

constant folding

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

$\Longrightarrow$

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

dead code elimination

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

$\Longrightarrow$

```
return 42;
```

loop invariant code motion

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

$\Longrightarrow$

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

78
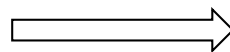
Inlining

```
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;
}
```

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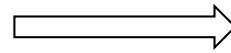
```
void f(int b) {
  ...
  for (int i ...) {
    a = b + 3;
    x[i] = a * i;
  }
}

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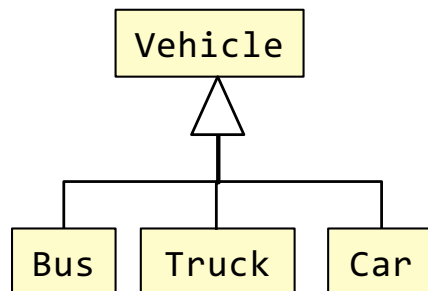The approach used today in the SUN/Oracle JVM, called "HotSpot".

Inline call caches
a way to optimize method calls at runtime

Original calling code

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

Vehicle

Bus   Truck   Car

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

```
          Vehicle
             △
      ┌──────┼──────┐
    Bus    Truck   Car
```

Inline call caches
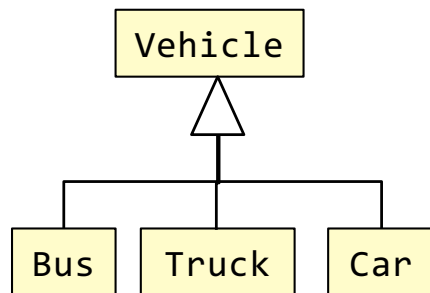## a way to optimize method calls at runtime

Original calling code

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

optimize →

Optimized calling code

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

```
        Vehicle
           △
    ┌──────┼──────┐
  Bus    Truck   Car
```

Called method:

```
Bus-m-prologue:
  if (receiver is not a Bus)
    receiver.m(); // Ordinary slow lookup
Bus-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 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).
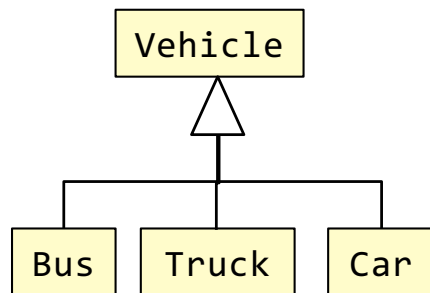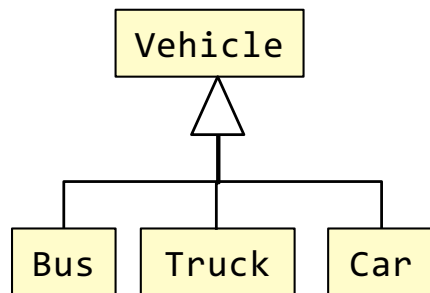If so, continue executing Bus-m. If not, do a normal (slow) lookup.

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);
}
```

Vehicle
  △
  |
Bus   Truck   Car

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.

```
            Vehicle
               △
        ┌──────┼──────┐
      Bus    Truck    Car
```

Inlined call cache

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

Methods:

```
Bus-m-prologue:
  if (!receiver is a Bus)
    receiver.m(); // normal lookup
Bus-m:
  normal method body
  ...
```
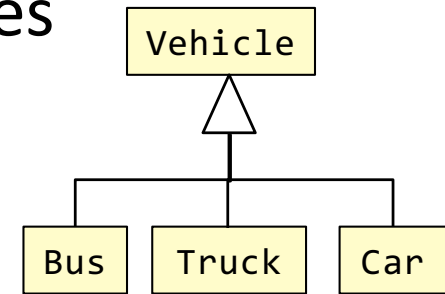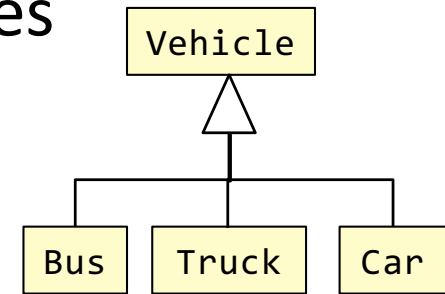
Polymorphic inline caches (PICs)
a generalization of inline call caches

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
}
```
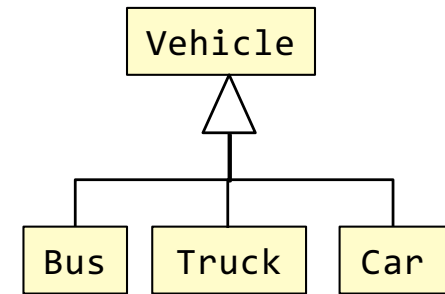
Methods:

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

Inlining method bodies
Can be done after inlining calls

Inlining method bodies
Copy the called methods into the calling code

```
           Vehicle
              △
              |
    ┌─────────┼─────────┐
  Bus       Truck      Car
```

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

Vehicle

Bus   Truck   Car

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
}
```
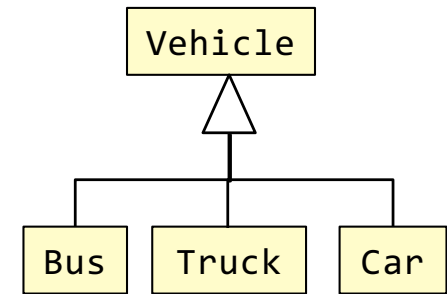
optimize →

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:
  ...
```

Methods:

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

Further optimization

Vehicle

Bus | Truck | Car

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
}
```

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
}
```
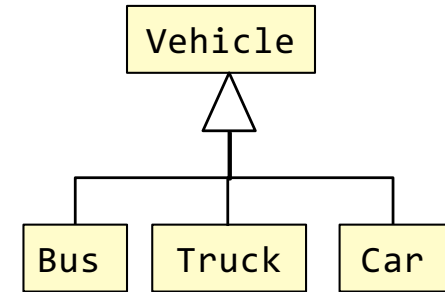
Methods:

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

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

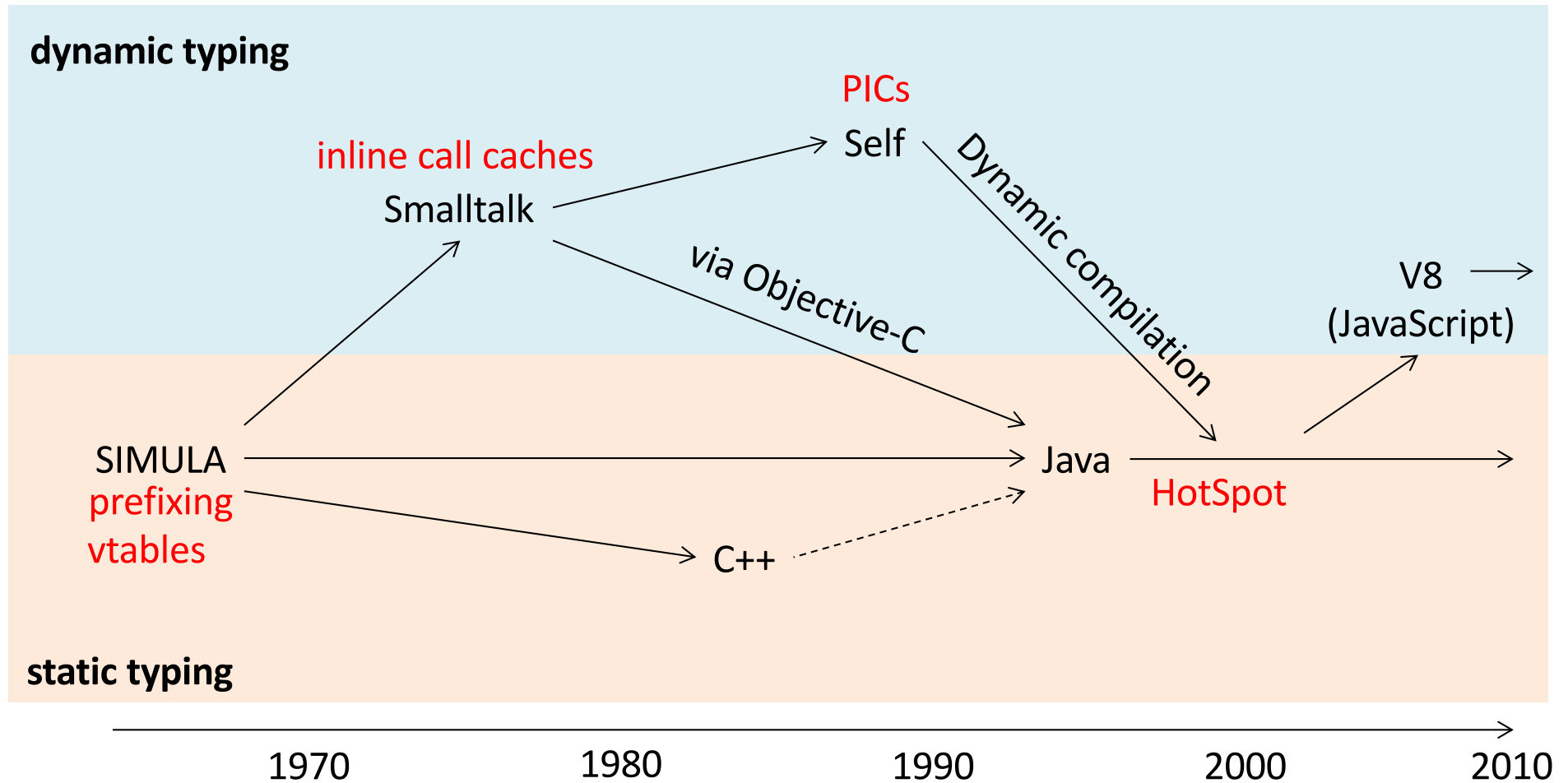Slower when the program starts (warm-up). Fast after a while (steady-state).

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?