CSCI 1102 Computer Science 2

Meeting 9: Thursday 2/25/2021
Memory Organization

Mostly Random Access Memory (RAM)	Ephemeral Memory
Flash Drive	Persistent Storage

RAM: Contiguously allocated bytes – each byte has a numerical address

Addresses are natural numbers

An 8-bit pattern

32-bit words – 4 consecutive bytes

Address	
0x0000	0x 03 02 01 00
0x0004	0x 07 06 05 04
0x0008	0x 0B 0A 09 08
0x000C	0x 0F 0E 0D 0C

64-bit words – 8 consecutive bytes

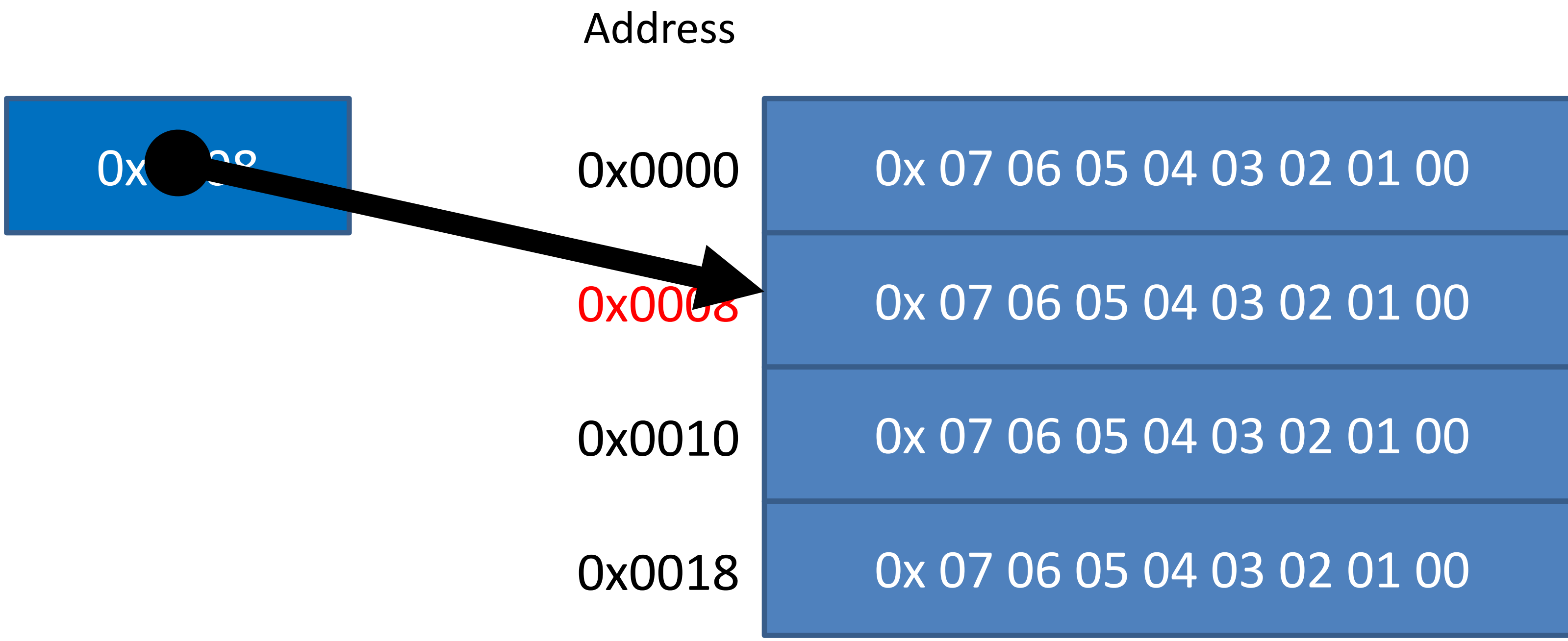
Address	
0x0000	0x 07 06 05 04 03 02 01 00
0x0008	0x 07 06 05 04 03 02 01 00
0x0010	0x 07 06 05 04 03 02 01 00
0x0018	0x 07 06 05 04 03 02 01 00

Variables often hold Addresses

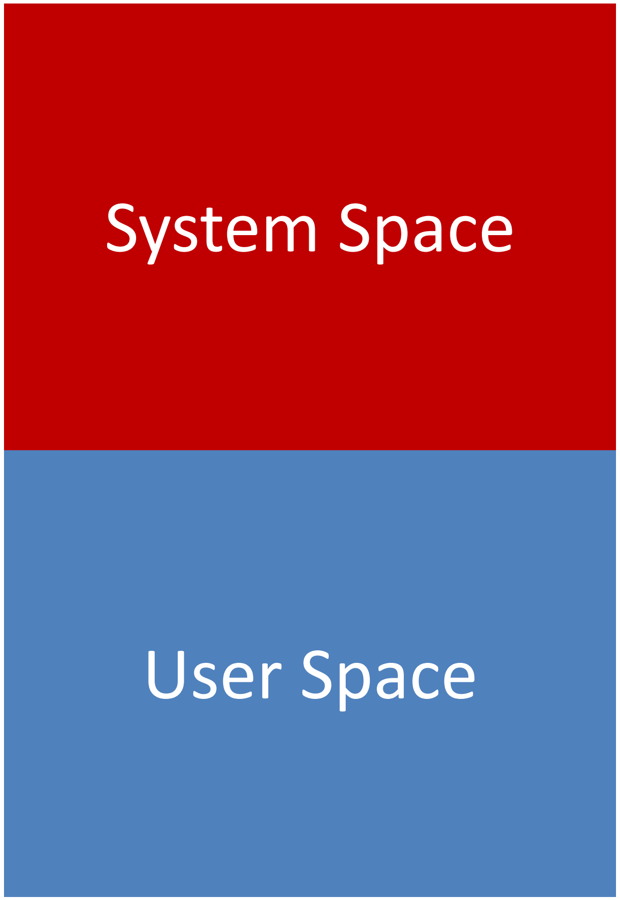
0x0008

Address	
0x0000	0x 07 06 05 04 03 02 01 00
0x0008	0x 07 06 05 04 03 02 01 00
0x0010	0x 07 06 05 04 03 02 01 00
0x0018	0x 07 06 05 04 03 02 01 00

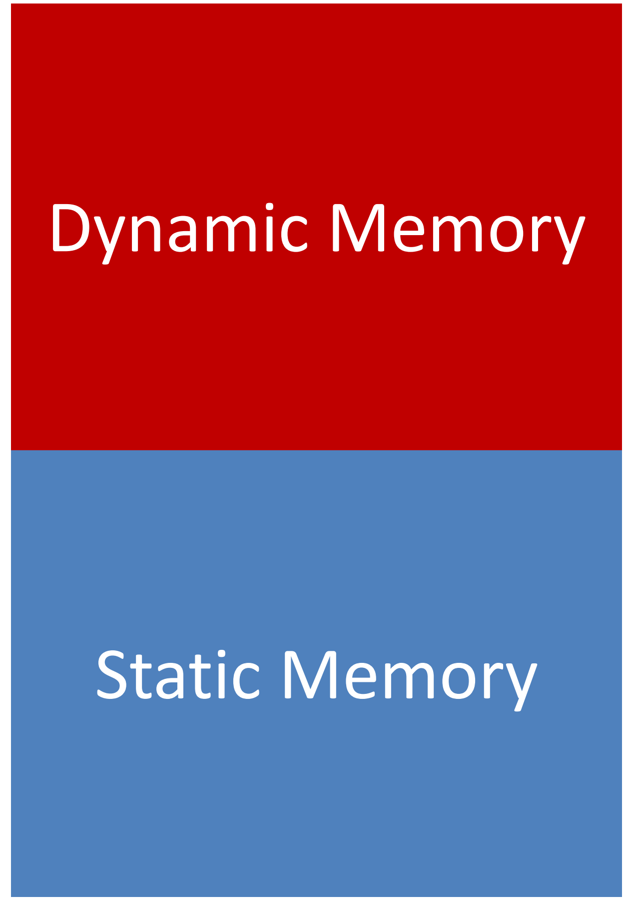
In diagrams, addresses are usually depicted abstractly as arrows, often called pointers

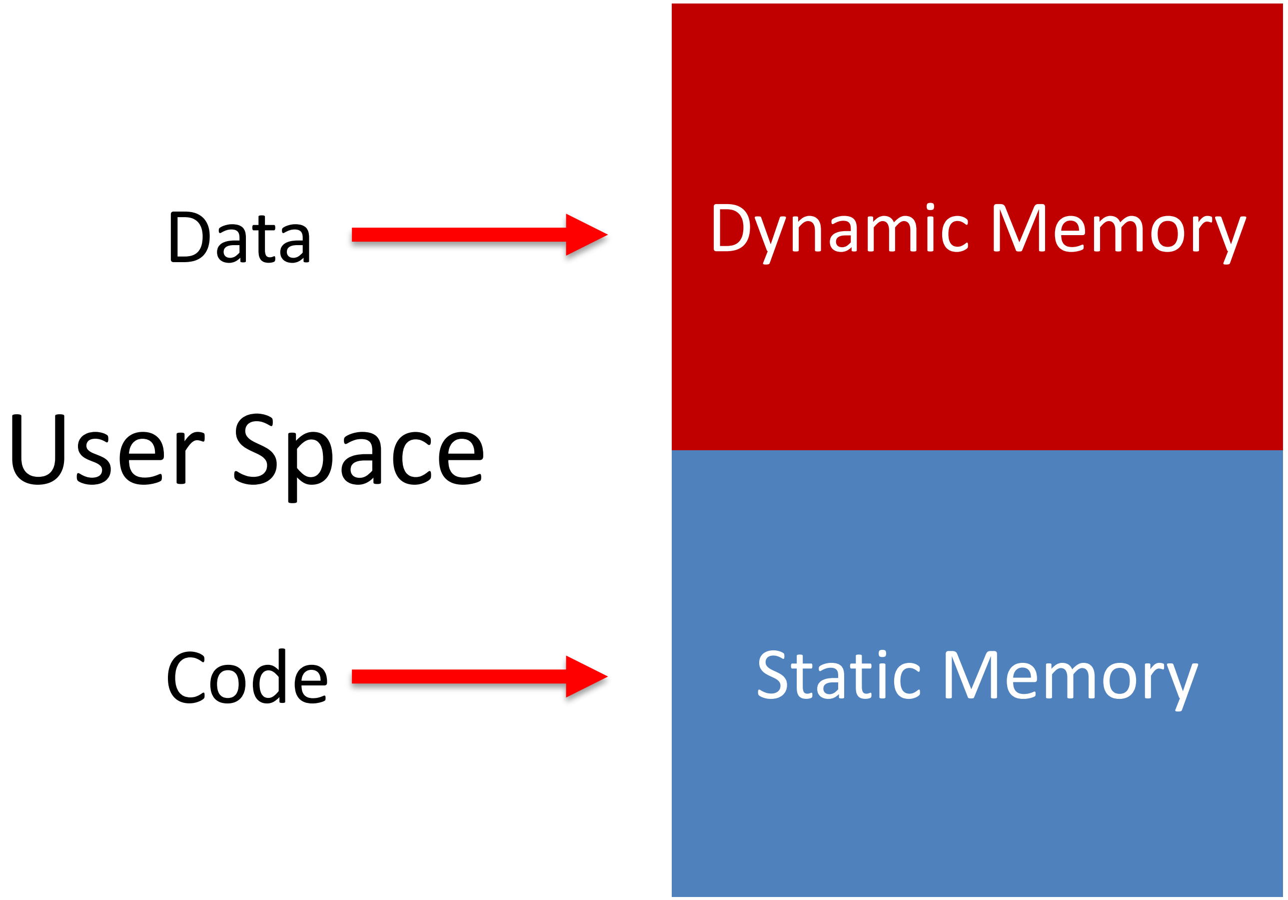


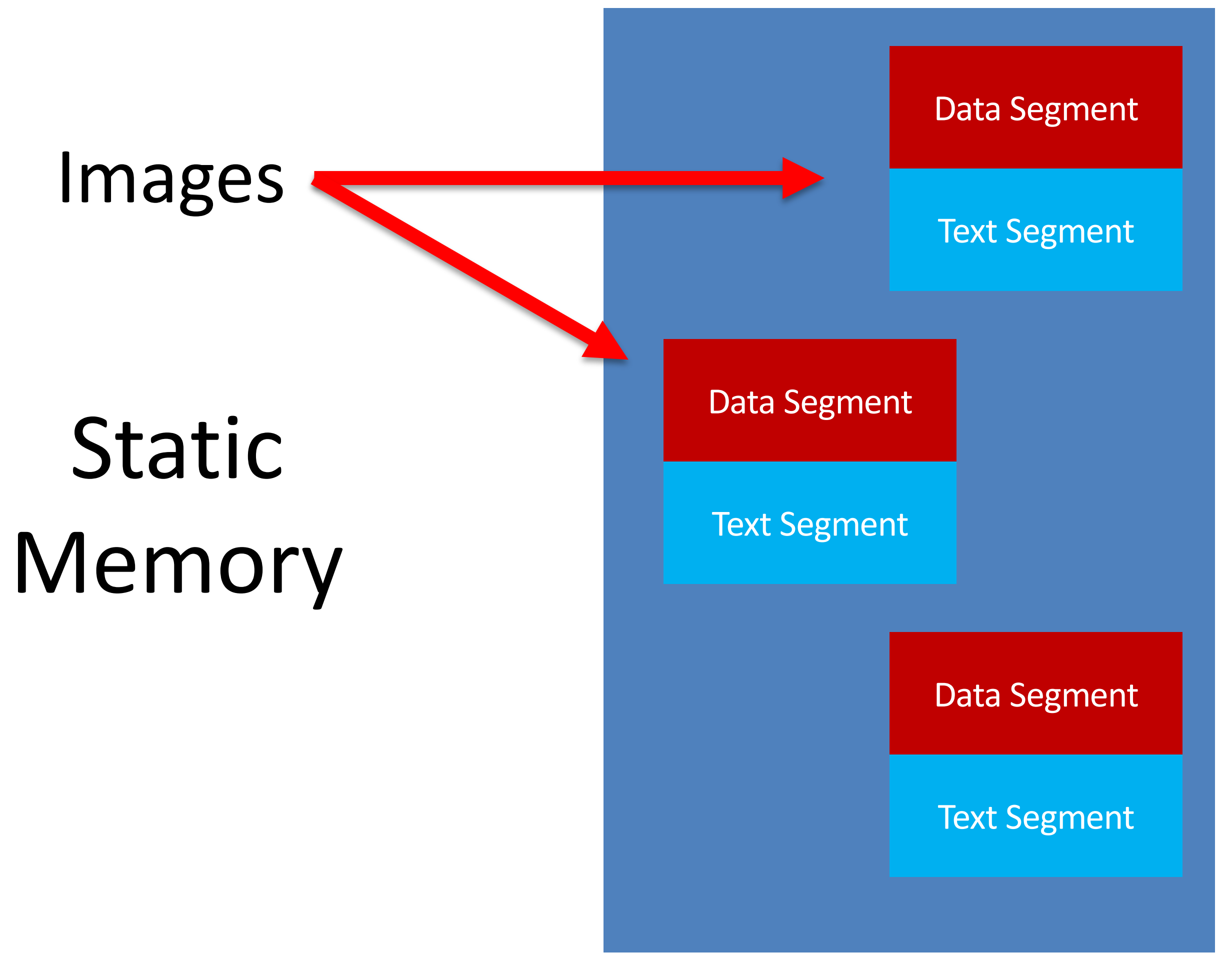
Ephemeral Memory



User Space

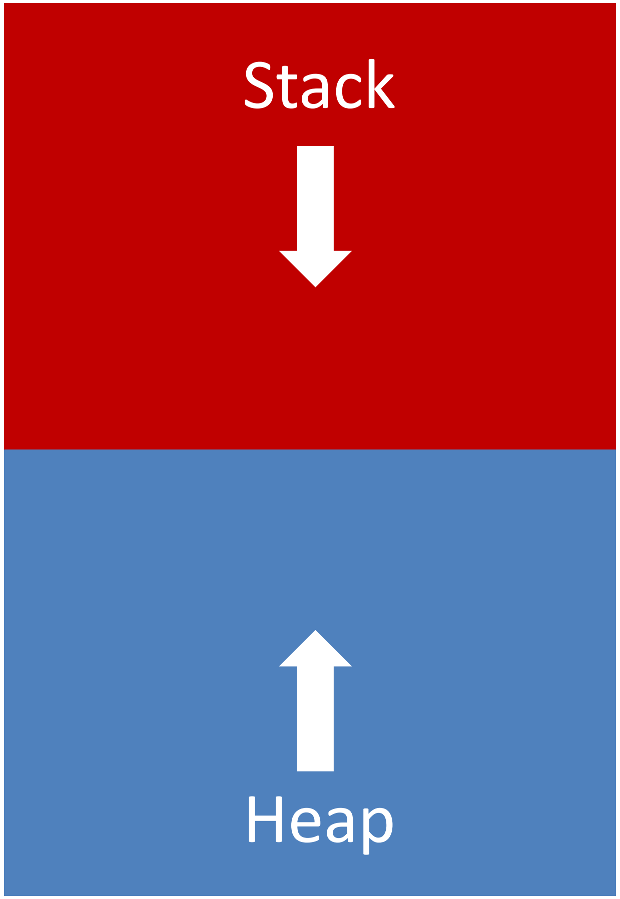



Data
Dynamic Memory
User Space
Code
Static Memory

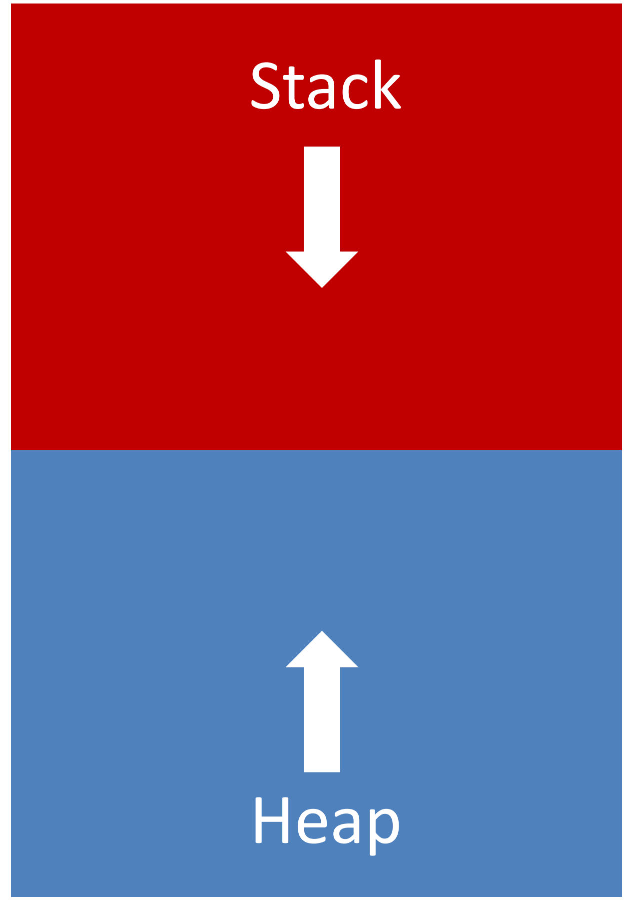

Images
Static
Memory
Data Segment
Text Segment
Data Segment
Text Segment
Data Segment
Text Segment

All function/method definitions can be understood as images residing in static memory

Dynamic Memory



Storage for function variables



Dynamic Memory

Storage for large values (e.g., arrays) & long-living values

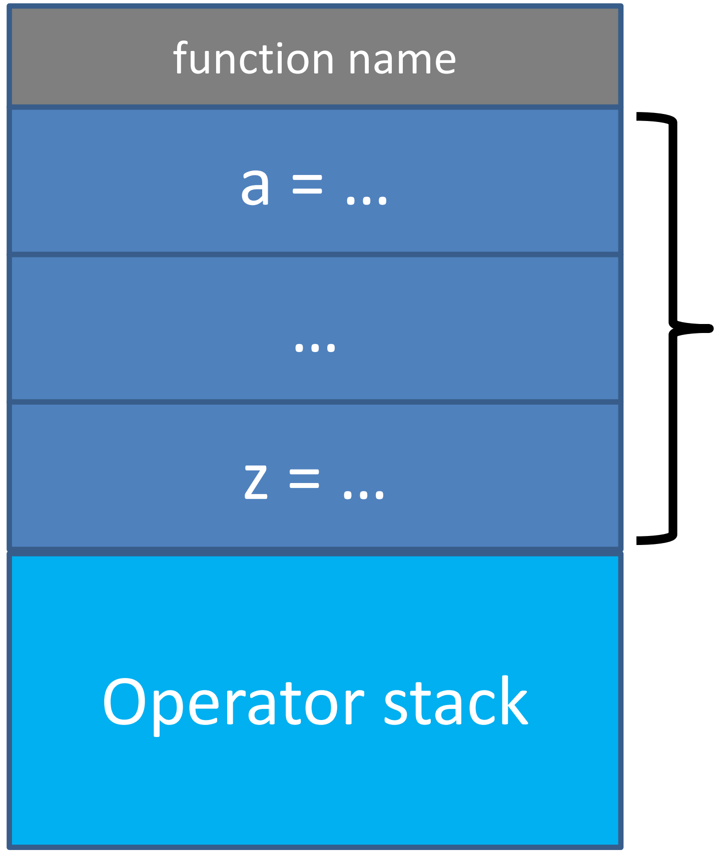
Managing Dynamic Memory

- The stack: the ~~compiler generates code that~~ JVM manages the allocation and deallocation of call frames on the call stack

- A call frame is placed on top of the stack on function call and removed on function return

- The heap: managed by a run-time support routine called a garbage collector.

The Call Stack: Call Frames/Activation Records



Slots for input parameters & local variables.

A fixed-size stack for expression evaluation

Body of function evaluated with respect to a call frame.

The Size of a Call Frame's Operator Stack is Fixed

a + (b + (c + (d + e)))

a b c d e + + + +

e

d

c

b

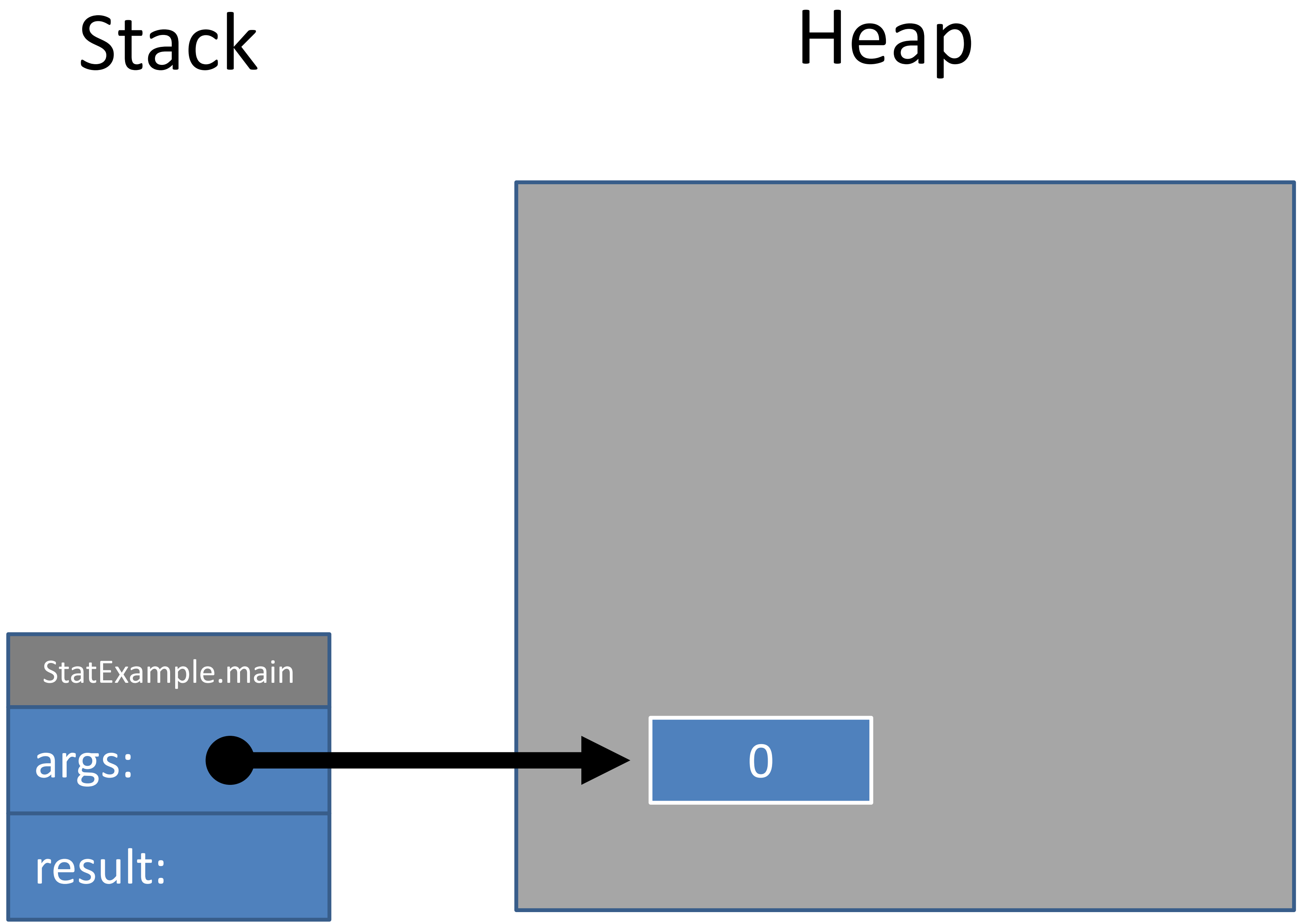
a

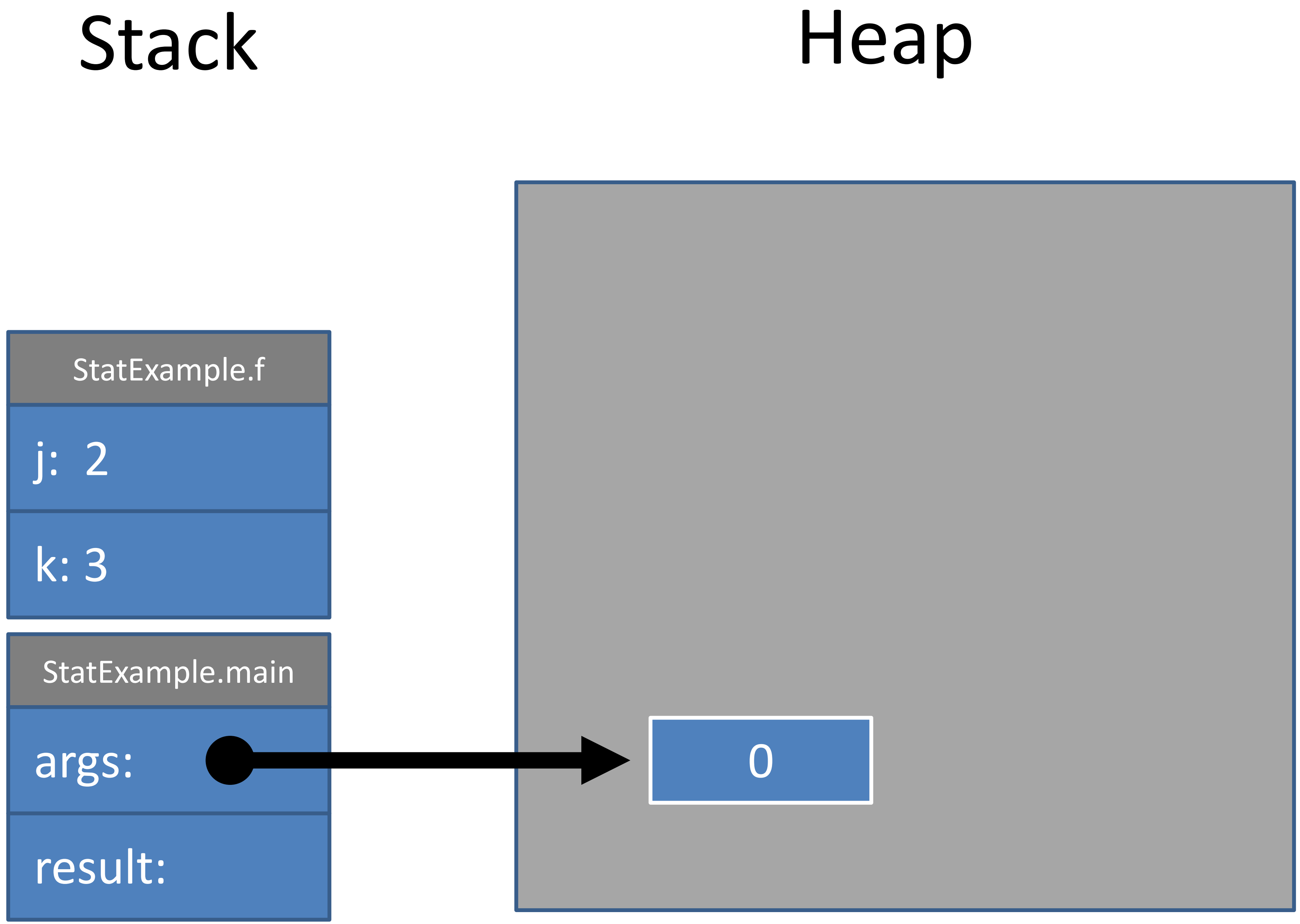
Static Function Example

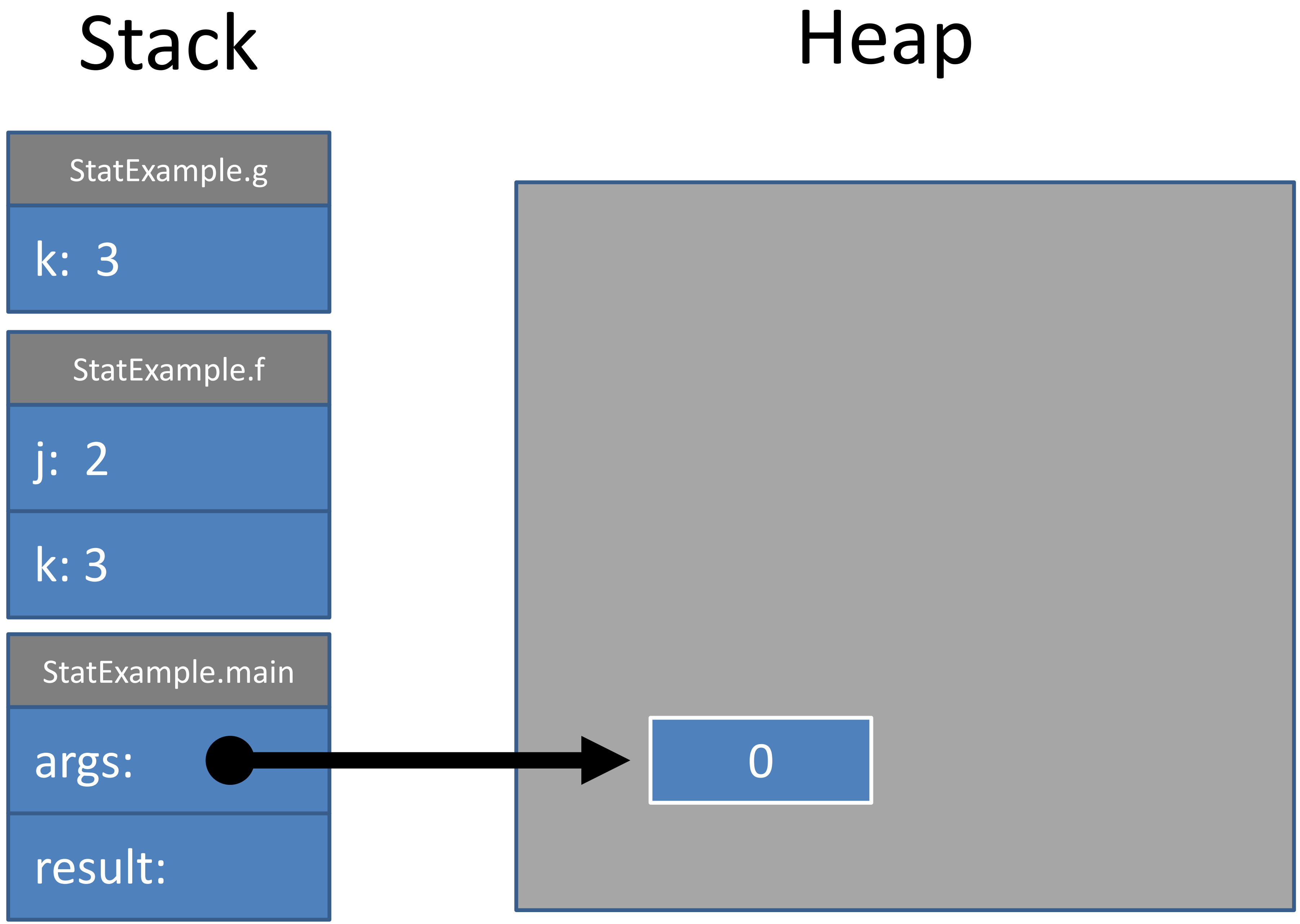
```java

public class StatExample {

  public static int g(int k) {
    return k * 2;
  }

  public static int f(int j, int k) {
    return j * g(k);
  }

  public static void main(String[] args) {
    int result = f(2, 3);
    System.out.format("The answer is %d.\n", result);
  }
}
```

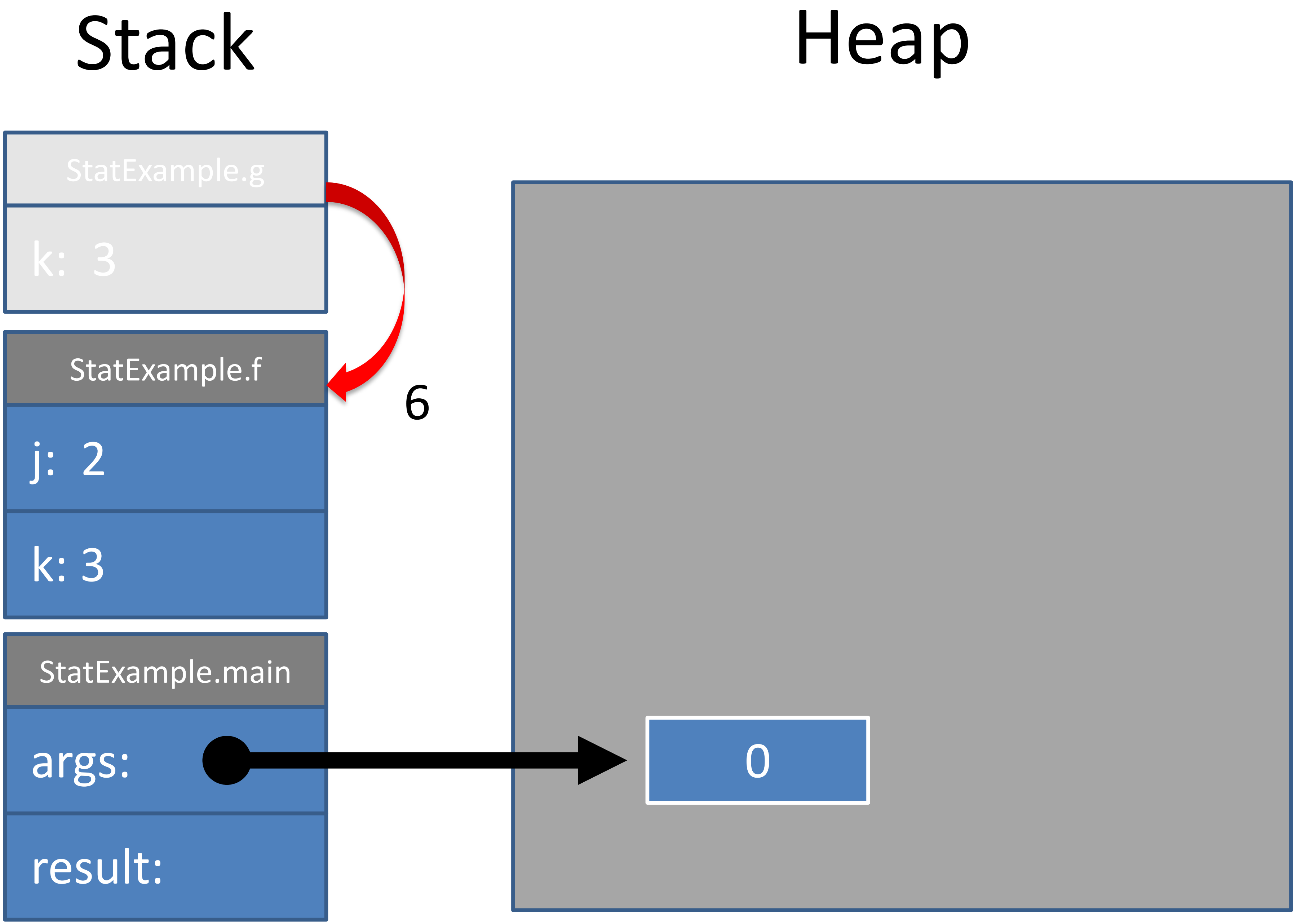
Stack

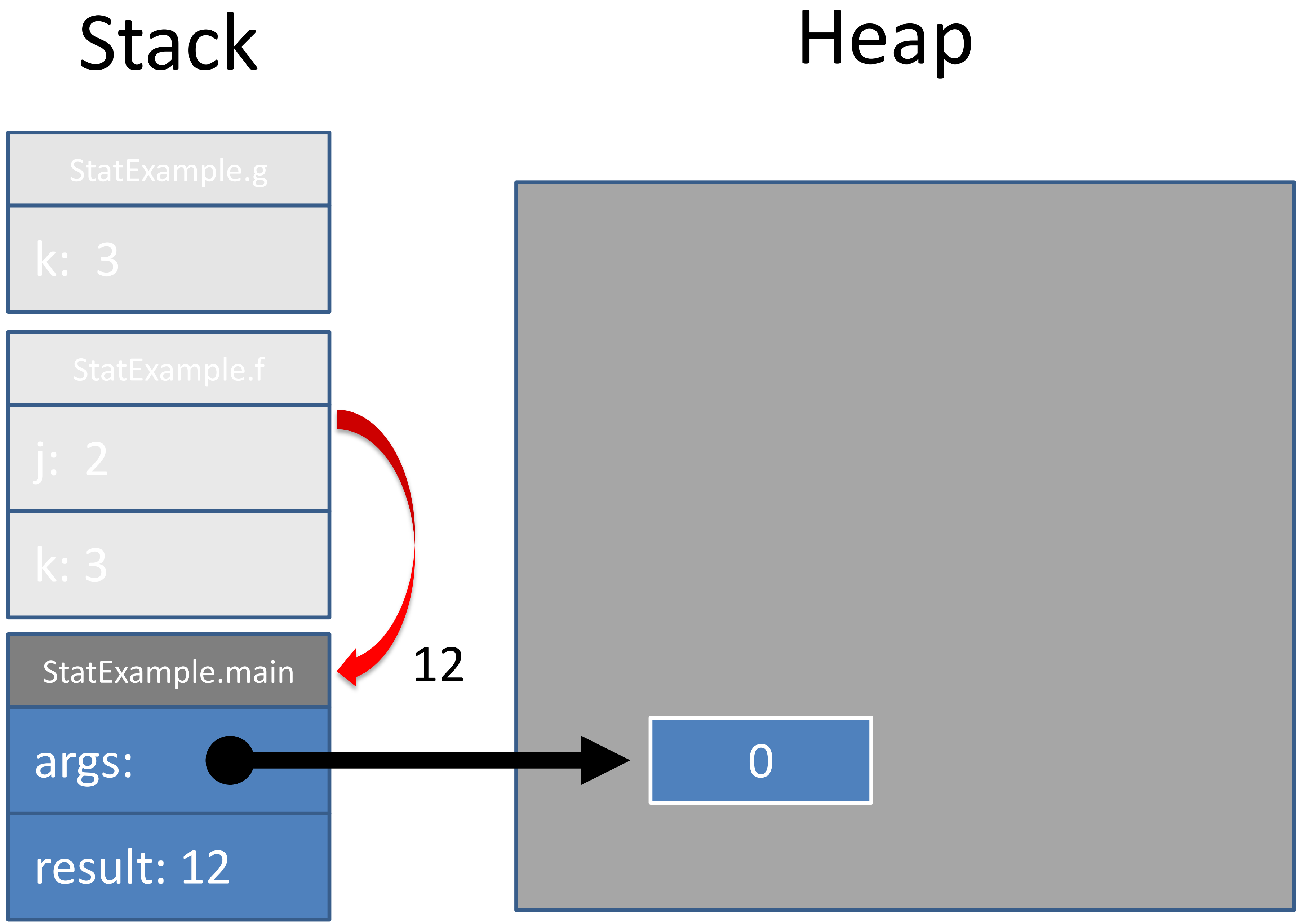
Heap

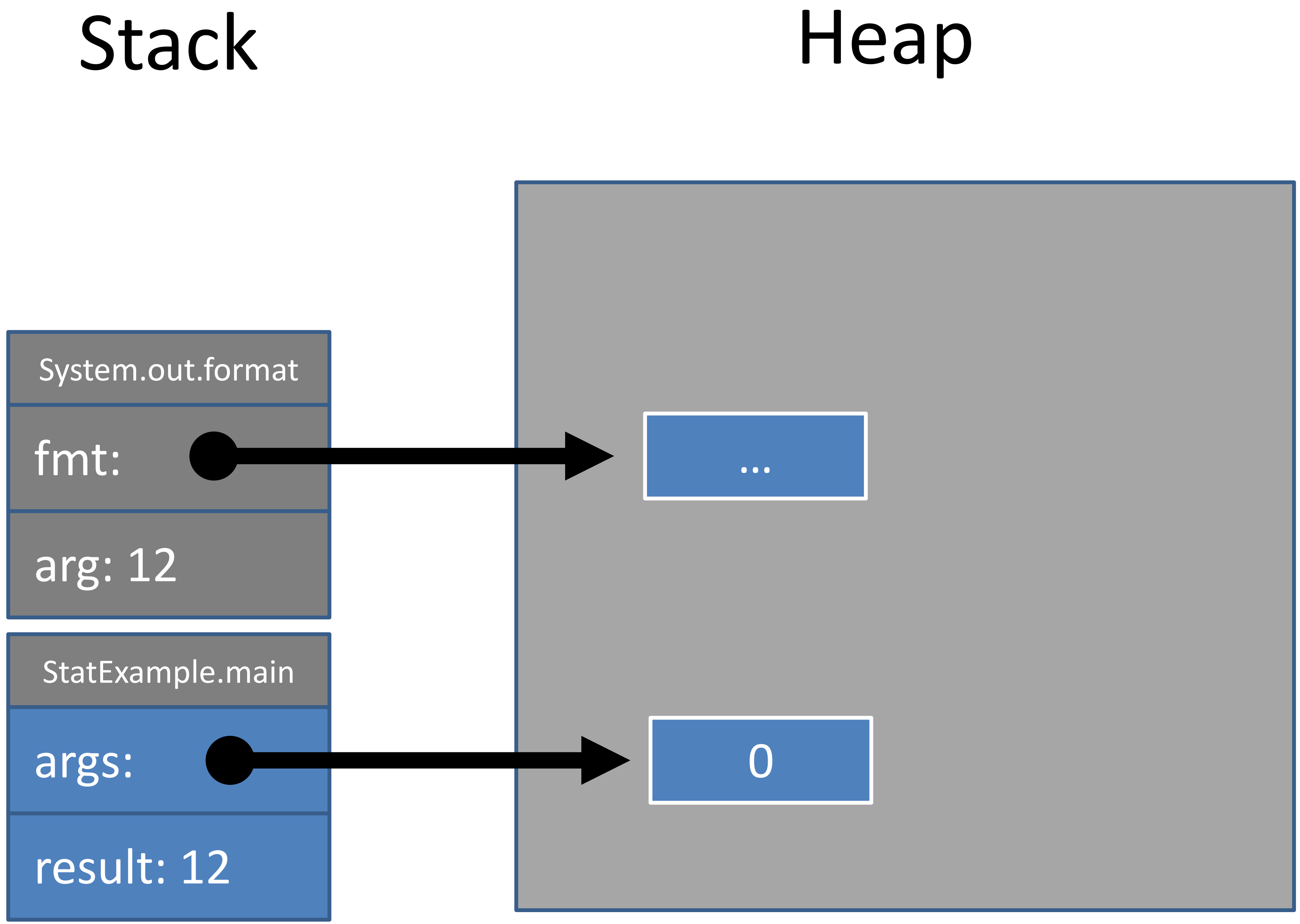

Stack
Heap
StatExample.main
args:
result:
0


Stack
Heap
StatExample.f
j: 2
k: 3
StatExample.main
args:
result:
0

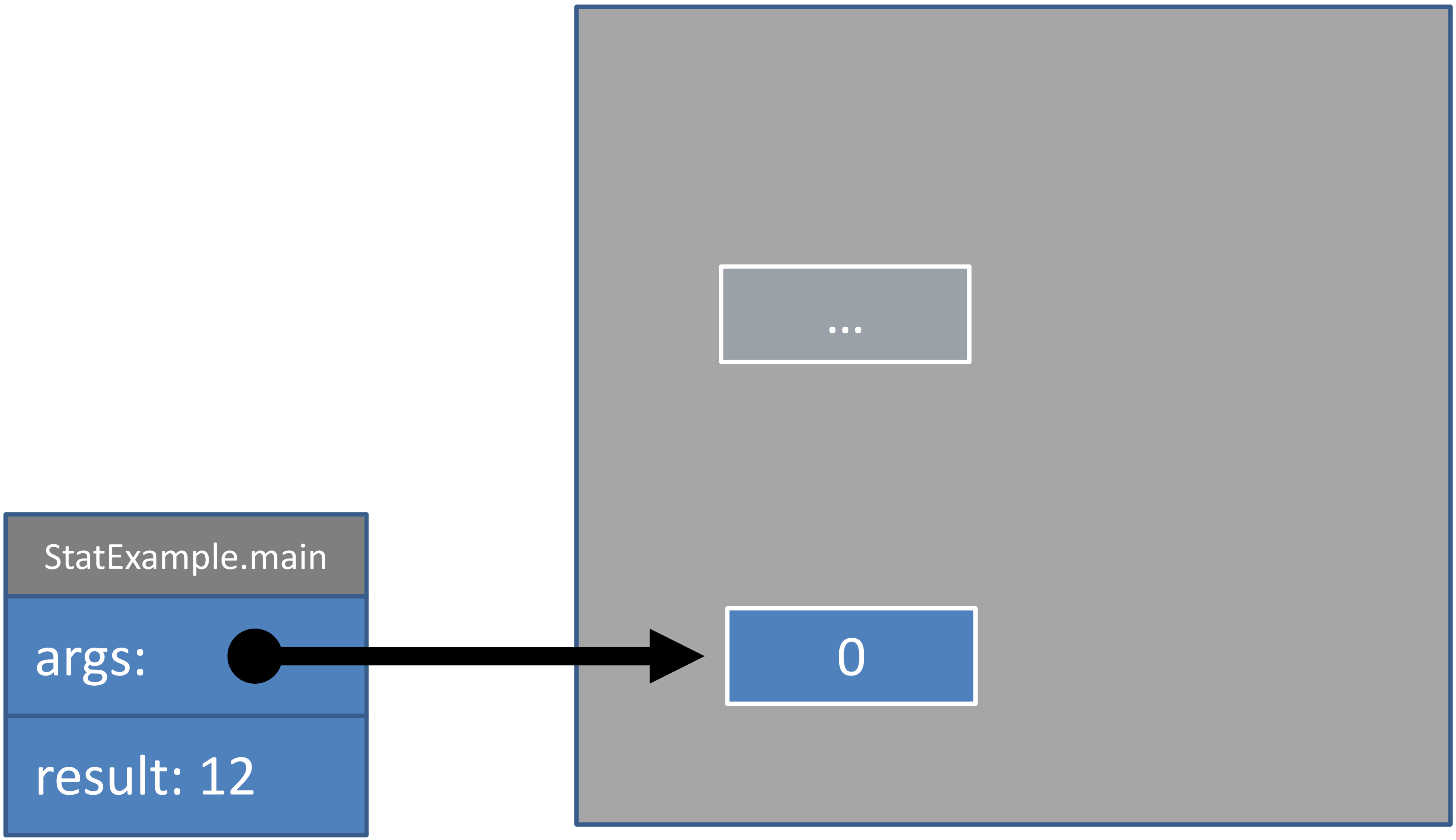

Stack
Heap
StatExample.g
k: 3
StatExample.f
j: 2
k: 3
StatExample.main
args:
result:
0


Stack
Heap
StatExample.g
k: 3
StatExample.f
6
j: 2
k: 3
StatExample.main
args:
result:
0


Stack
Heap
StatExample.g
k: 3
StatExample.f
j: 2
k: 3
StatExample.main
args:
result: 12
12
0

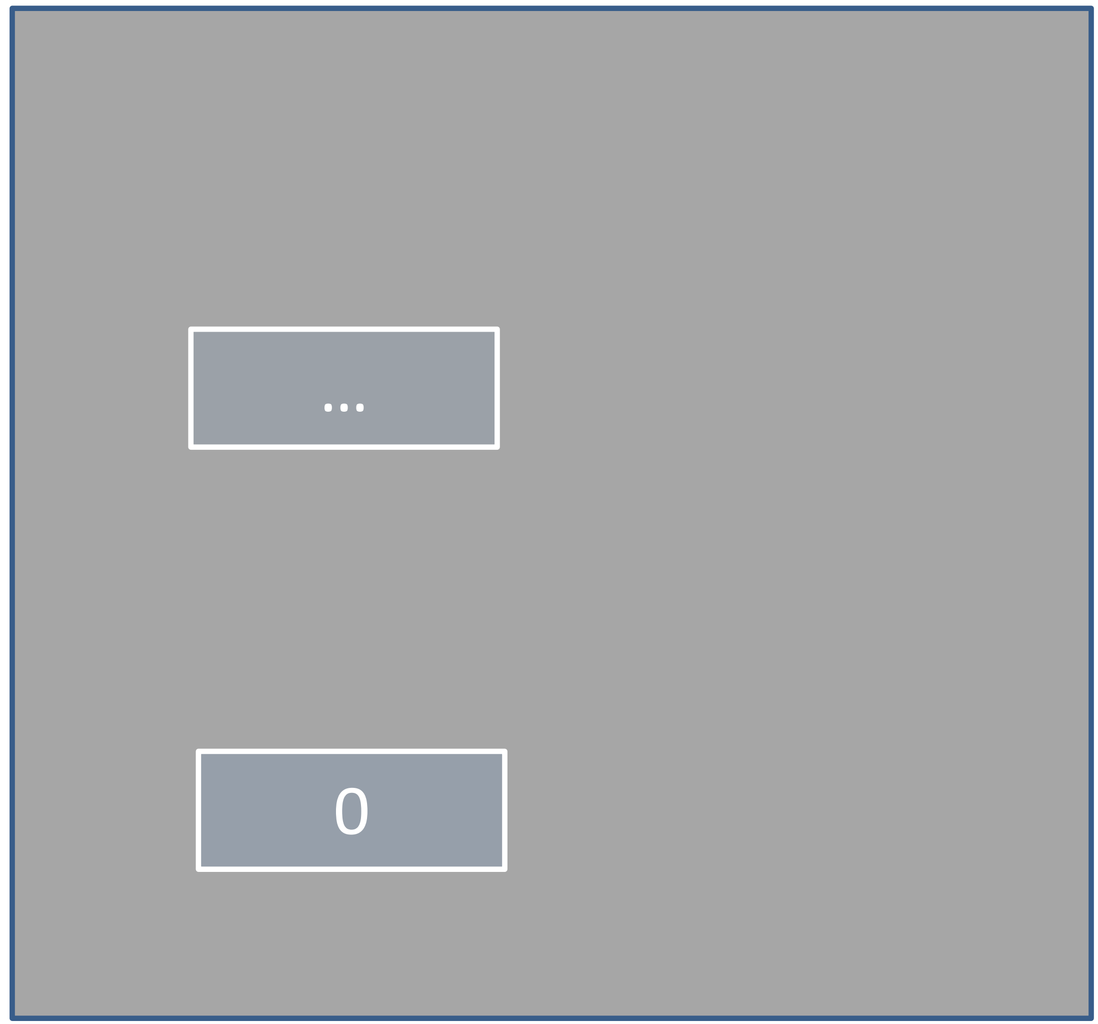

Stack
Heap
System.out.format
fmt:
arg: 12
StatExample.main
args:
result: 12
...
0

Stack Heap



Stack

Heap

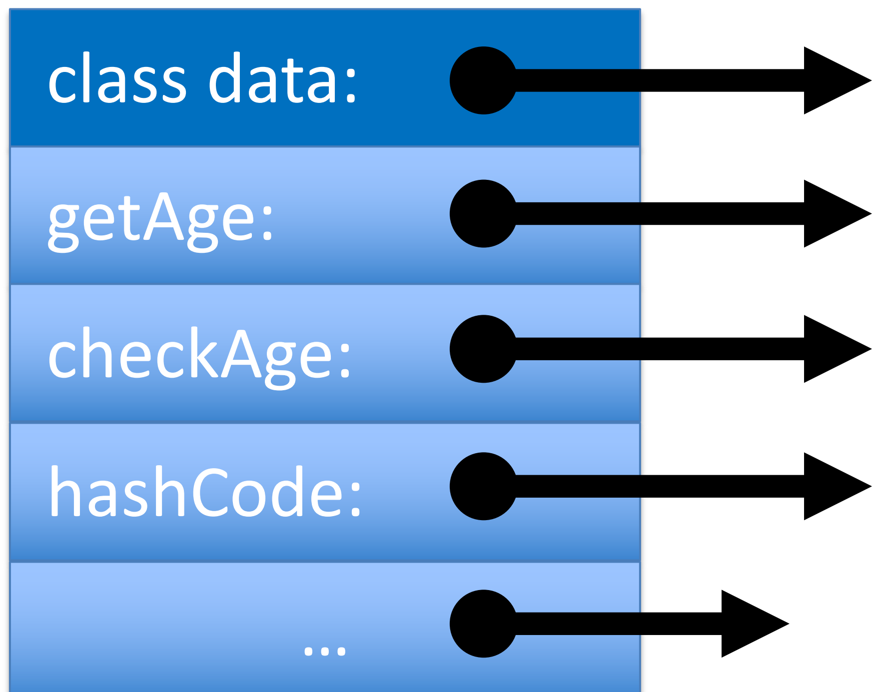


```java

public class DynExample {

  private int age;

  public DynExample(int age) { this.age = age; }

  private int getAge() { return this.age; }

  private void checkAge() {
    if (this.age < 0 || this.age > 120)
      throw new RuntimeException("Bad age");
  }

  public static void main(String[] args) {
    DynExample de = new DynExample(21);
    de.checkAge();
    System.out.format("Age is %d.\n", de.getAge());
  }
}
```

Dynamic Function Example

Every Class has a *Dispatch Table*

```
public static void main(String[] args) {

  DynExample de = new DynExample(21);
```



Info on **getAge** function

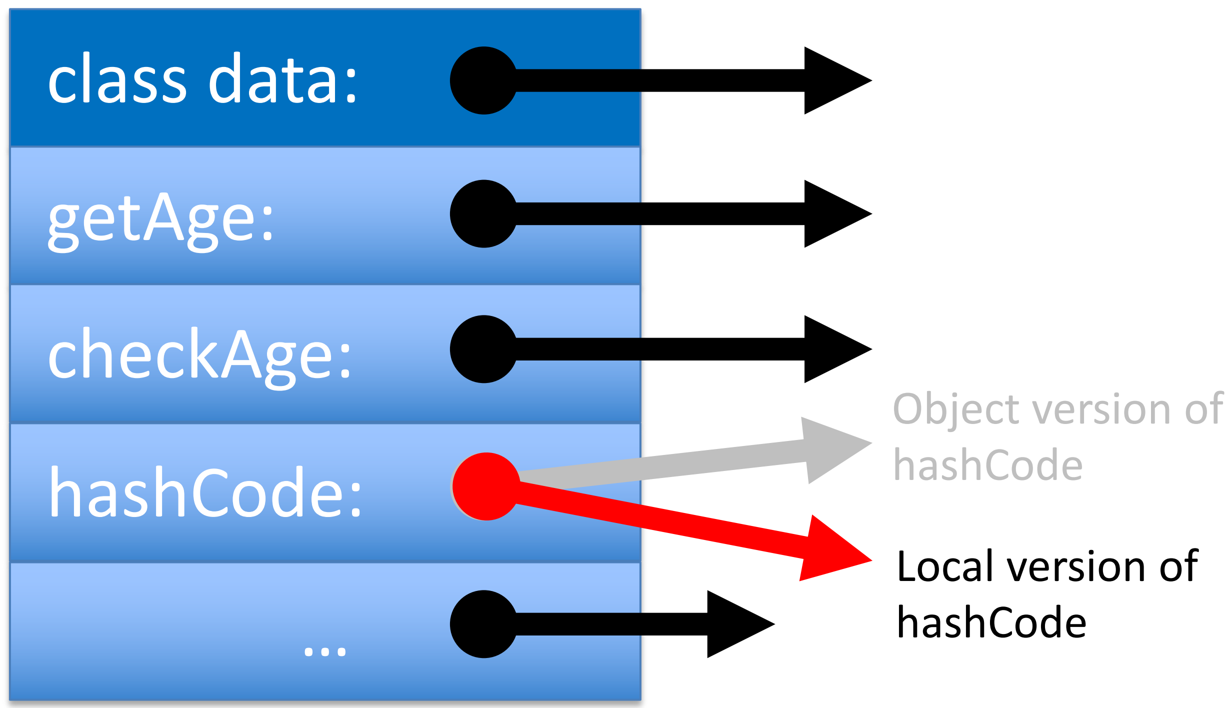
Info on **hashCode** function inherited from Object

hashCode etc inherited from Object

Dispatch Tables Support Inheritance and Override

```java
public static void main(String[] args) {

  DynExample de = new DynExample(21);
```

Override e.g., hashCode? Replace the arrow in the table



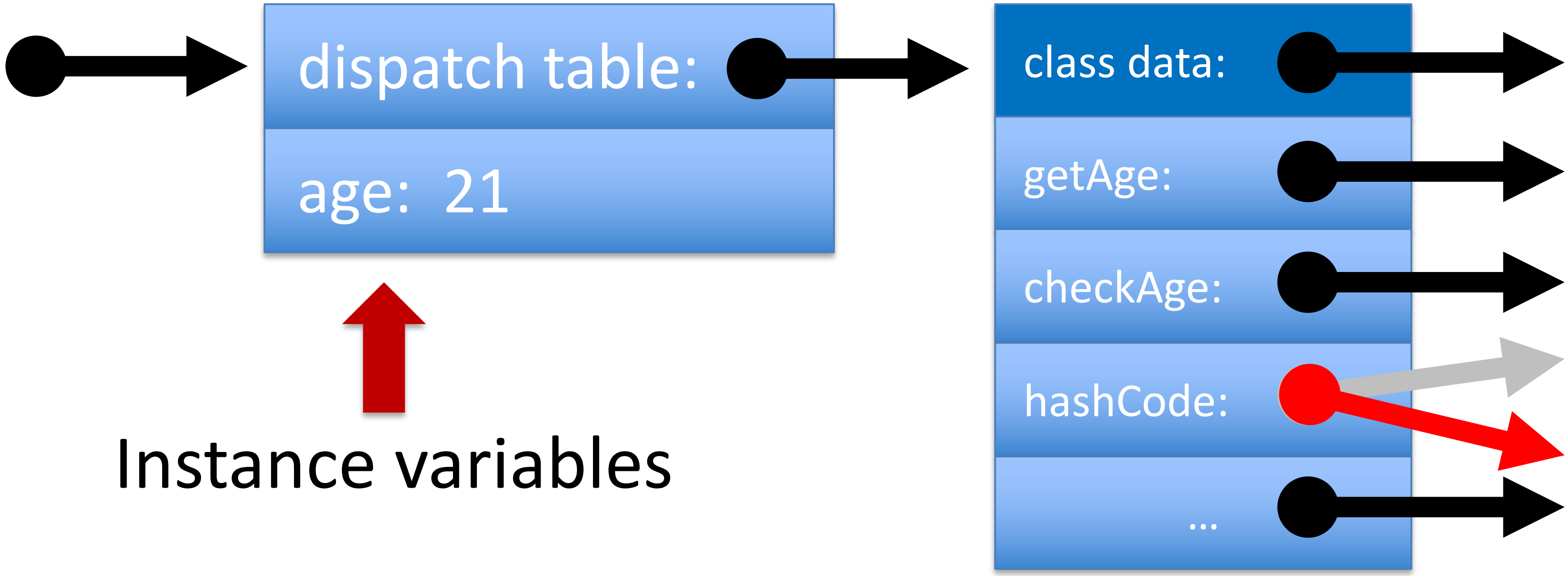
Dispatch Tables

- All objects created with new can share the same dispatch table

- Dispatch tables are *fixed*, they can be stored in static memory or in the heap.

Object Representation in the Heap

```
public static void main(String[] args) {

    DynExample de = new DynExample(21);
```



Message-Passing Style

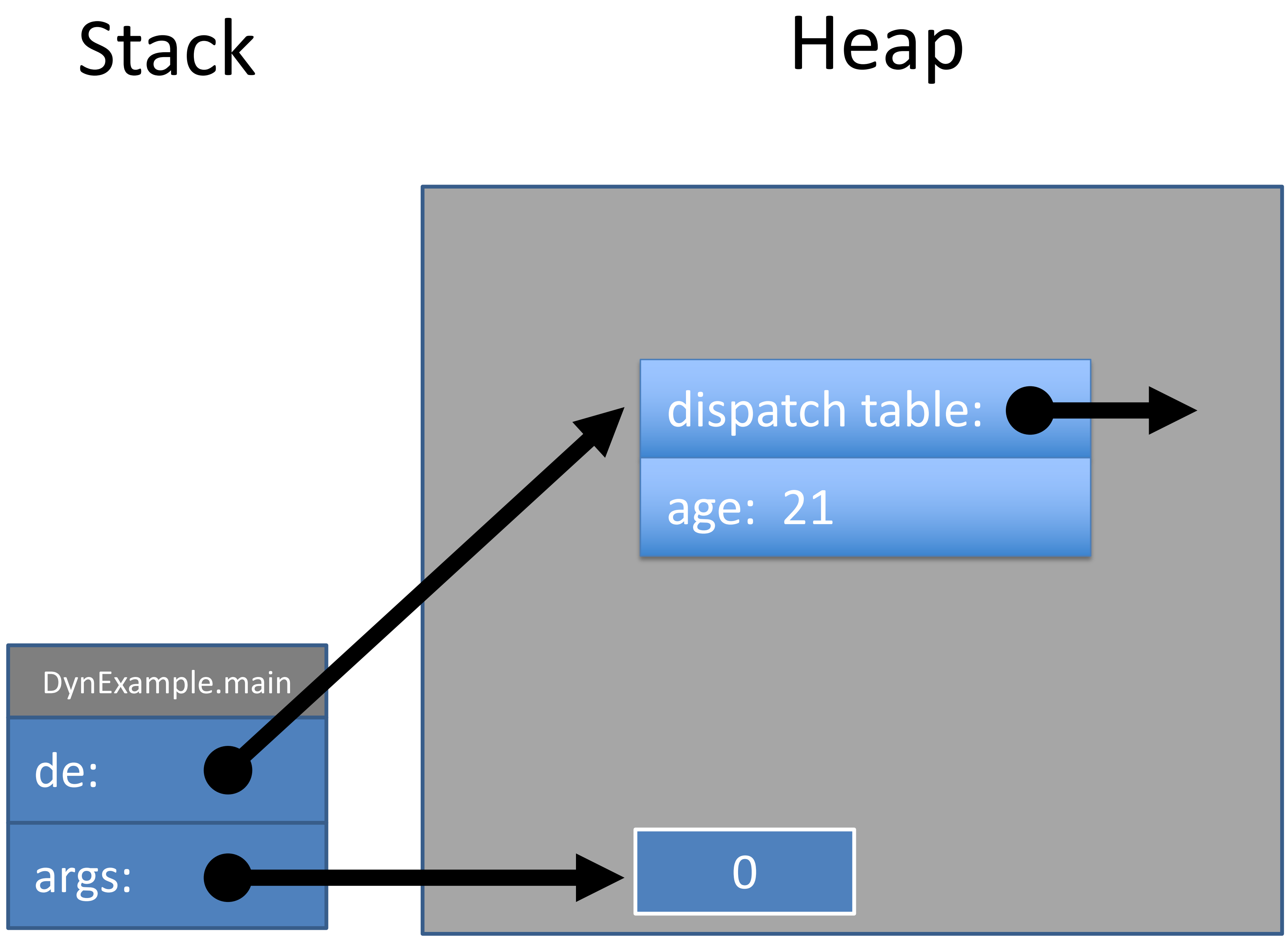
- Dynamic functions use message-passing style;

move(point, dx, dy)

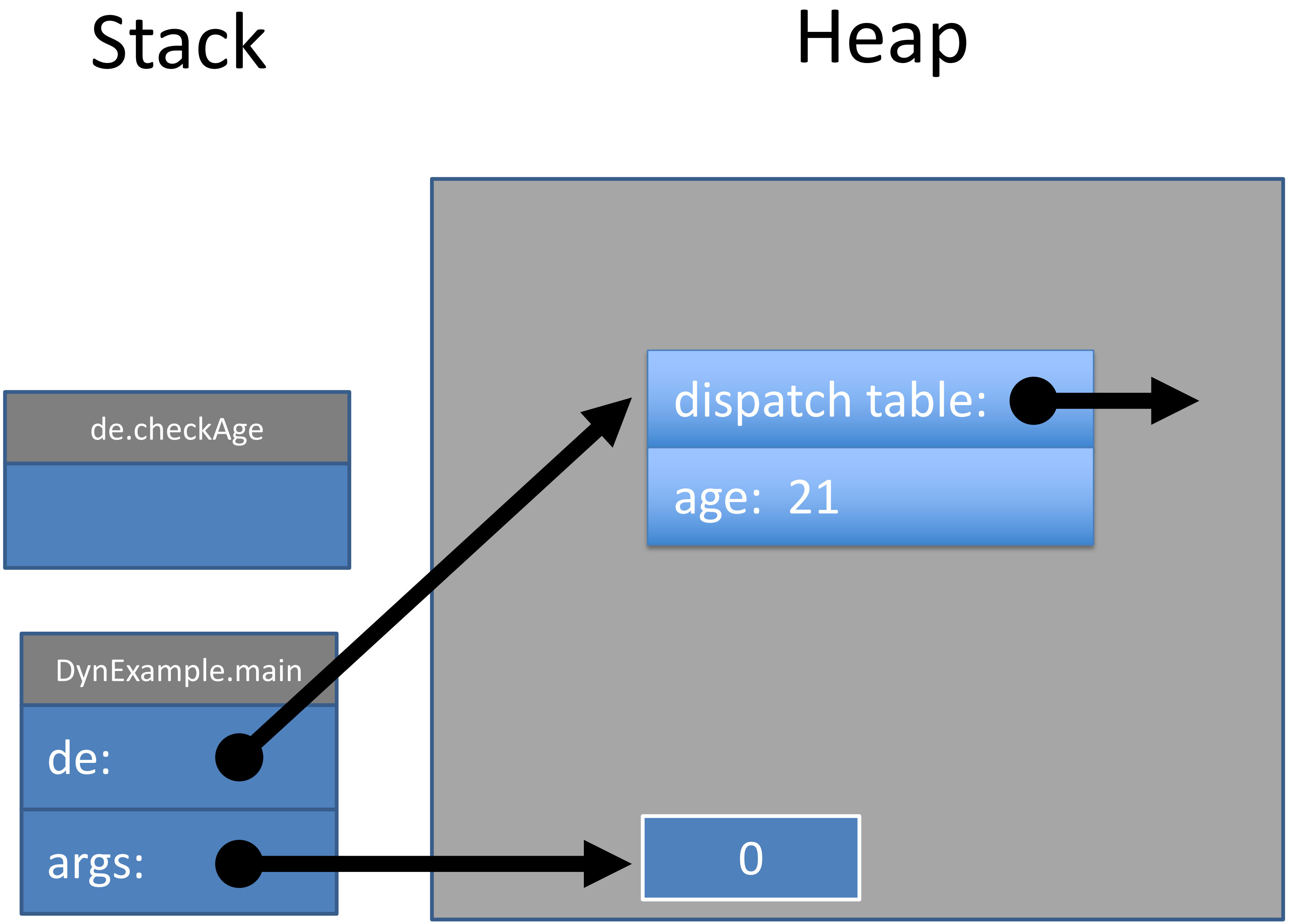
point.move(dx, dy)

- Message-passing style is orthogonal to inheritance

Message-Passing style Plumbing



Stack
Heap
DynExample.main
de:
args:
dispatch table:
age: 21
0


Stack
Heap
de.checkAge
DynExample.main
de:
args:
dispatch table:
age: 21
0

How can de.checkAge access the instance variable age?

Replace all calls of dynamic functions

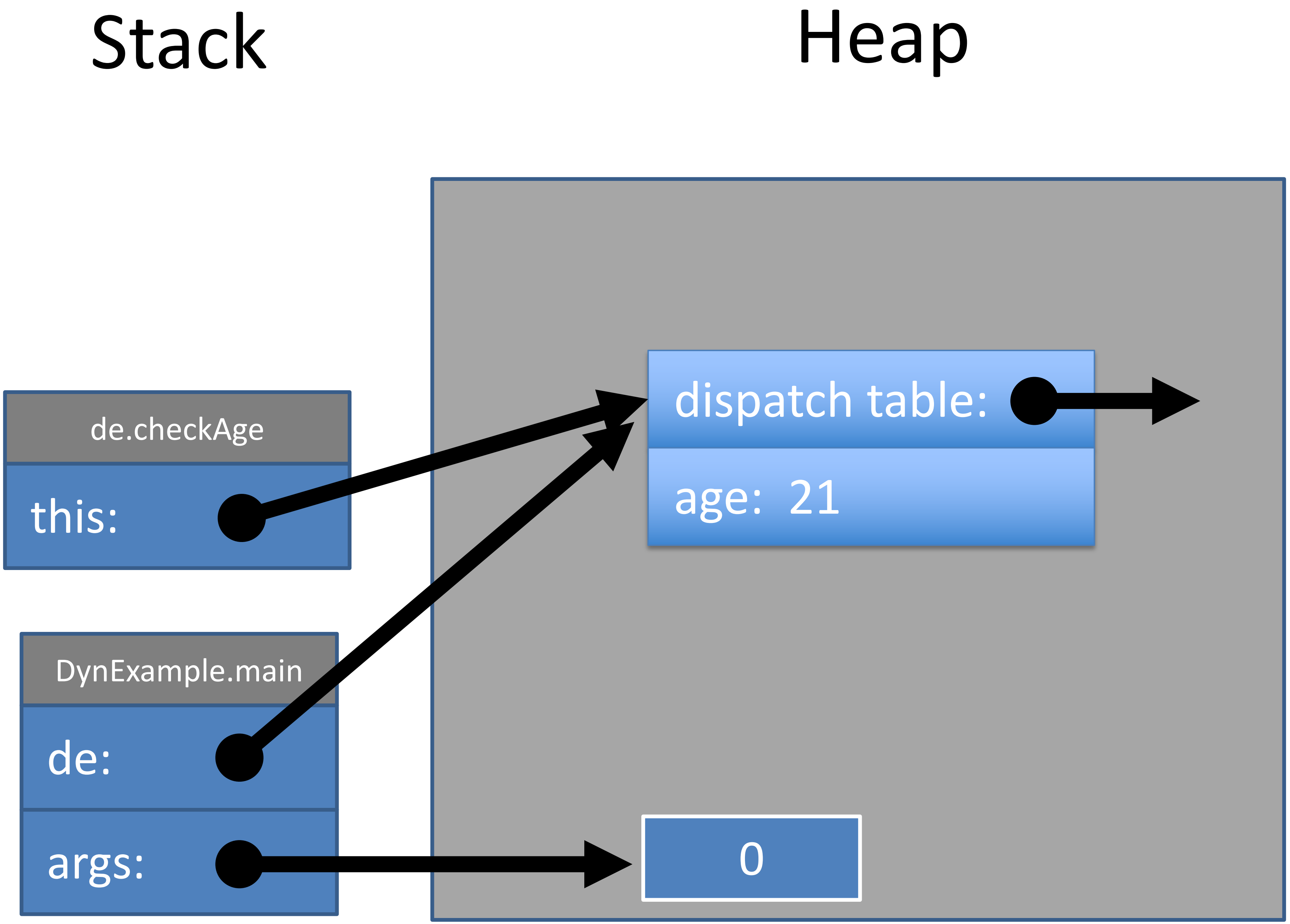
de.checkAge()

de.checkAge(de)

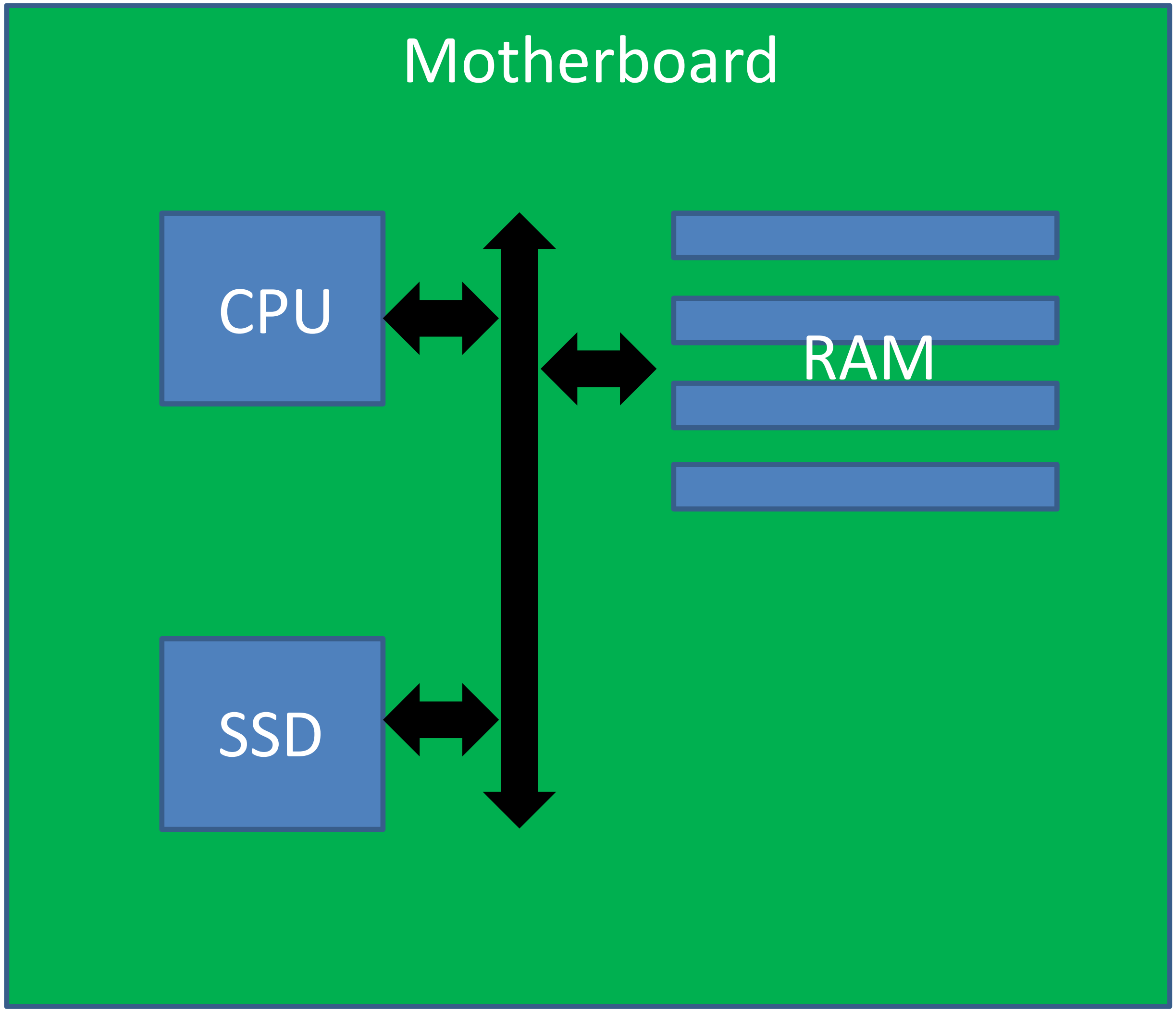
Replace all defs of dynamic functions

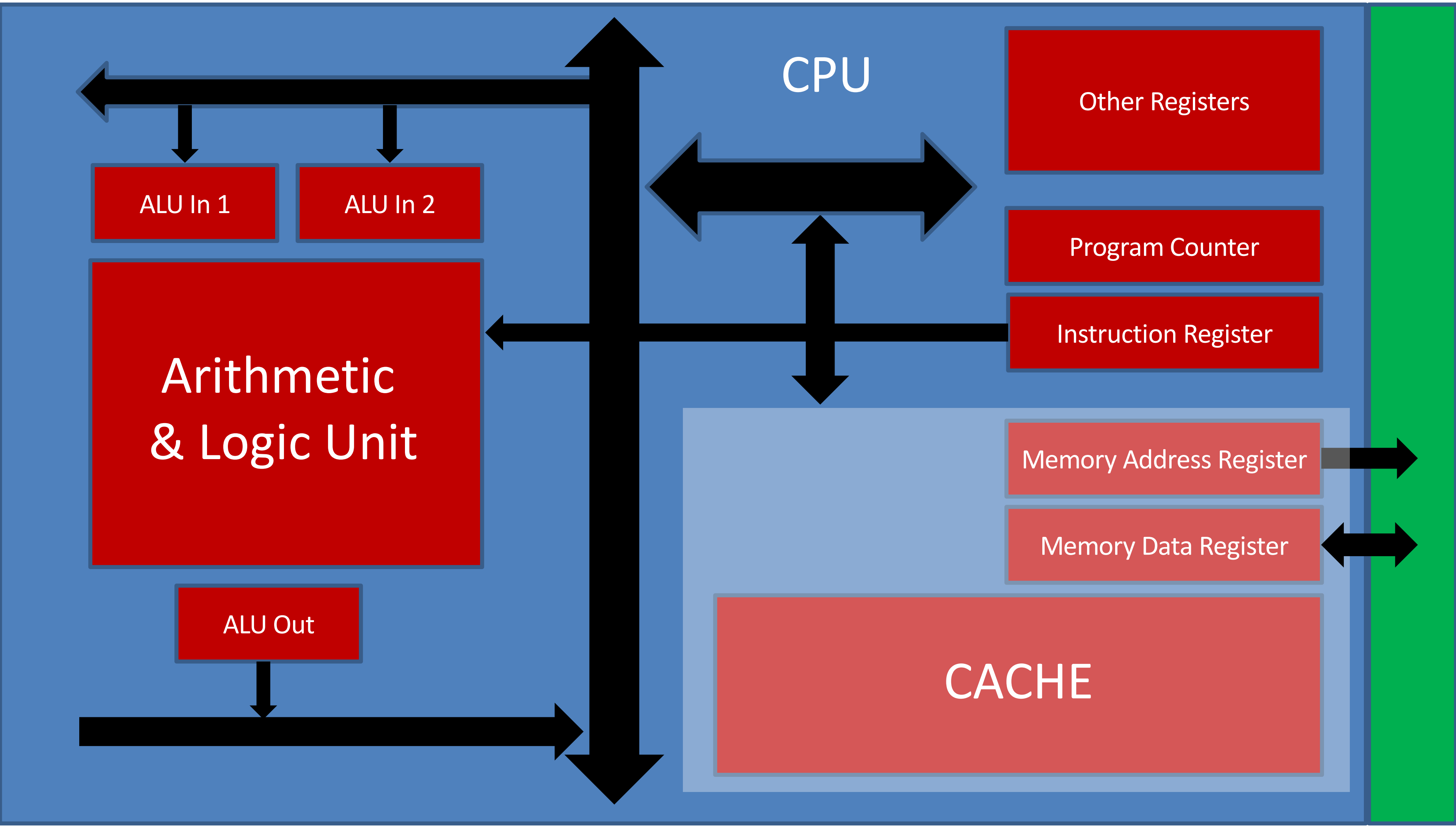
```
void checkAge() { … }
```

```
void checkAge(DynExample this) { … }
```


Stack
Heap
de.checkAge
this:
DynExample.main
de:
args:
dispatch table:
age: 21
0

Locality


Motherboard
CPU
SSD
RAM


CPU
Other Registers
ALU In 1
ALU In 2
Program Counter
Instruction Register
Arithmetic
& Logic Unit
Memory Address Register
Memory Data Register
ALU Out
CACHE

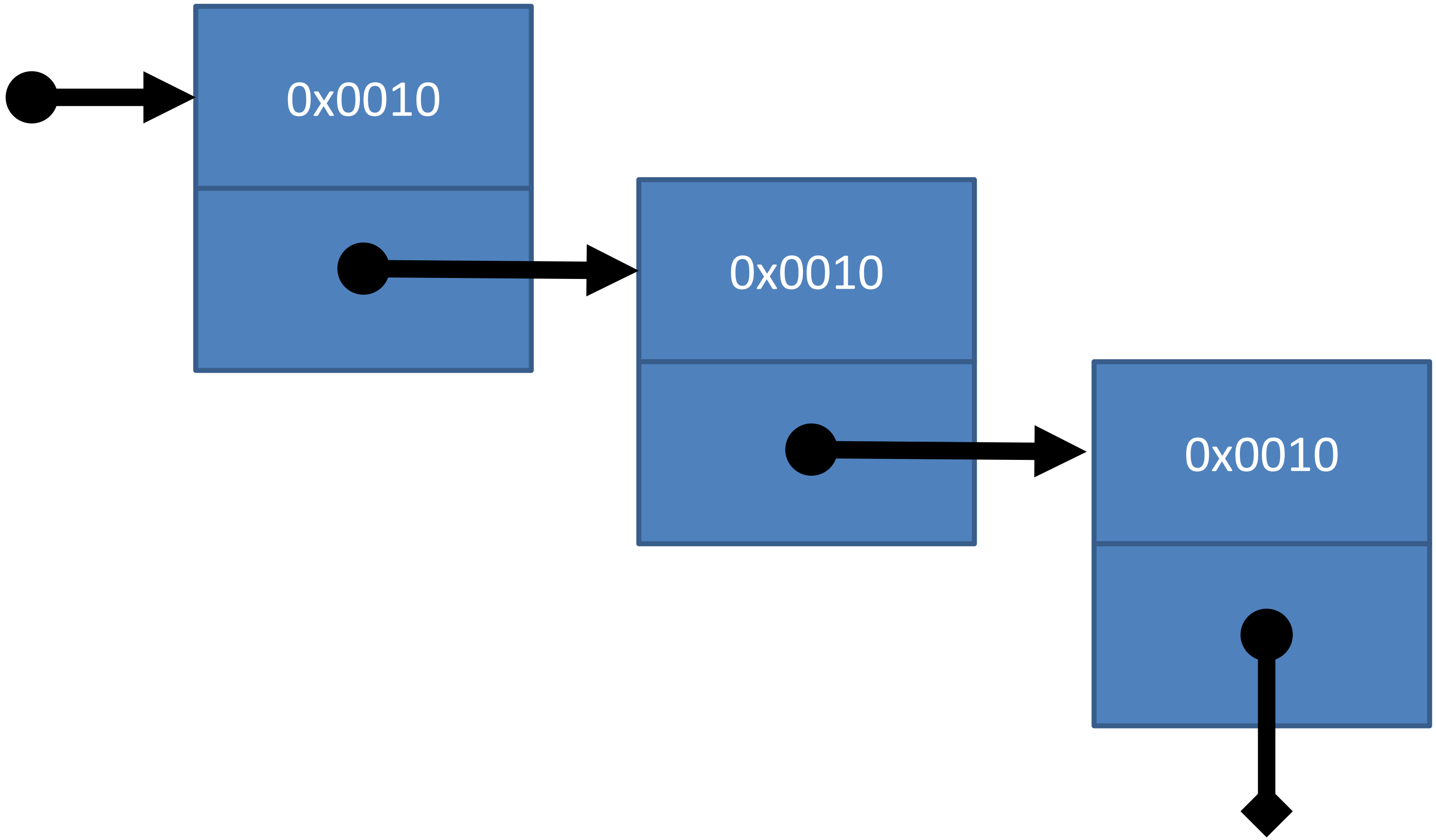
Latency numbers every programmer should know

```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns
Mutex lock/unlock ........................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

Linked Data Structures

- Pro: unlimited size with no linear resizing cost;
- Con: Poor locality

Block Storage & Linked Storage



Block Storage & Linked Storage