Spring 2019

CSE 216 : Programming Abstractions

TOPIC 2 – NAMES, SCOPES AND BINDINGS

NAMES, BINDING & SCOPES

Names:

- Function names, variable names, type names refer to memory addresses at runtime or to abstract type structures at compile time.

Binding:

- To clearly define the semantics of the program, we need to clearly identify this association between names and the objects they refer to.
- E.g. Function name is bound to its definition.
- The compiler/runtime system has to do this automatically.

Scopes:

- What are the rules that determine which names are visible in which parts of the program?

NAMES & BINDINGS

Name

A mnemonic character string representing something else (an identifier from the parser's point of view)

- `x, sin, f, prog1, null?` are names.
- `1, 2, 3, "test"` are not names.
- +, <=, … may be names if they are not built-in operations.

Binding

An association between two entities, typically between a name and the object it refers to

- Name and memory location (for a variable)
- Name and function
- Name and type

REFERENCING ENVIRONMENTS & SCOPES

Referencing environment

A complete set of bindings active at a certain point in a program

Scope of a binding

The region of a program or time interval(s) in the program's execution during which the binding is active

Scope

A maximal region of the program where no bindings are destroyed (e.g., a function body)

QUESTIONS ABOUT BINDINGS

- When is the binding established?
- How long does the binding/the bound object exist?
- Where does the bound object live?

BINDING TIMES

- Language design time: the design of specific program constructs (syntax), primitive types, and meaning (semantics), etc. are decided when the language is designed.

- Language implementation time: many issues are left to the implementer. These may include numeric precision (i.e., the number of bits), run time memory sizes, built-in run time exceptions, etc.

- Program writing time: e.g., the choice of algorithms, data structures, names.

BINDING TIMES

- Compile time (Early binding): compilers choose (i) how to map high-level constructs to machine code, and (ii) the memory layout for things used in the program.

- Link time (Early binding): the time at which multiple object codes (machine code files) and libraries are combined into one executable. For complex programs, there may be names in one module that refer to things in another module. Such bindings are done at link time.

- Load time (Early binding): the time at which the OS loads the executable into memory so that it can run.

- Run time (Early binding): many language-specific decisions may be taken during run time; the binding of values to variables may occur at run time.

Examples

LANGUAGE	FEATURE	BINDING TIME
C	*syntax:* `if (a>0) b:=a;`	language design
	reserved keywords: `main`	language design
	primitive types: `float` and `struct`	language design
	calls to static library routines: `printf`	link
	specific type of a variable	compile
Java	*reserved keywords:* `class`	language design
Any	*internal representation of literals (e.g.* `3.14` or `"foo"`*)*	language implementation
	non-static allocation of space for variables	run time

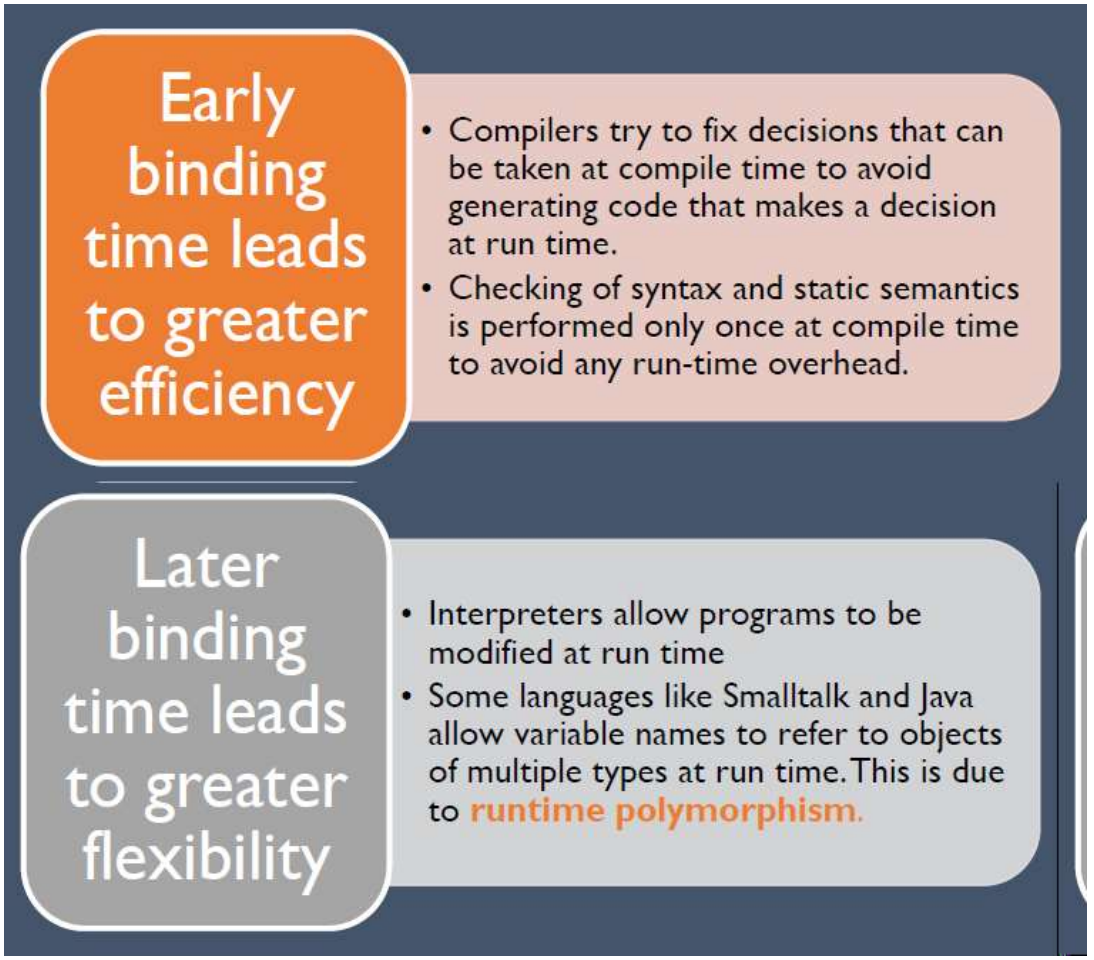
IMPORTANCE OF BINDING TIMES

Early binding (compile time, link time, load time):

- Typical in compiled languages
- Also called static binding

Late binding (run time):

- Typical in interpreted languages
- Also called dynamic binding



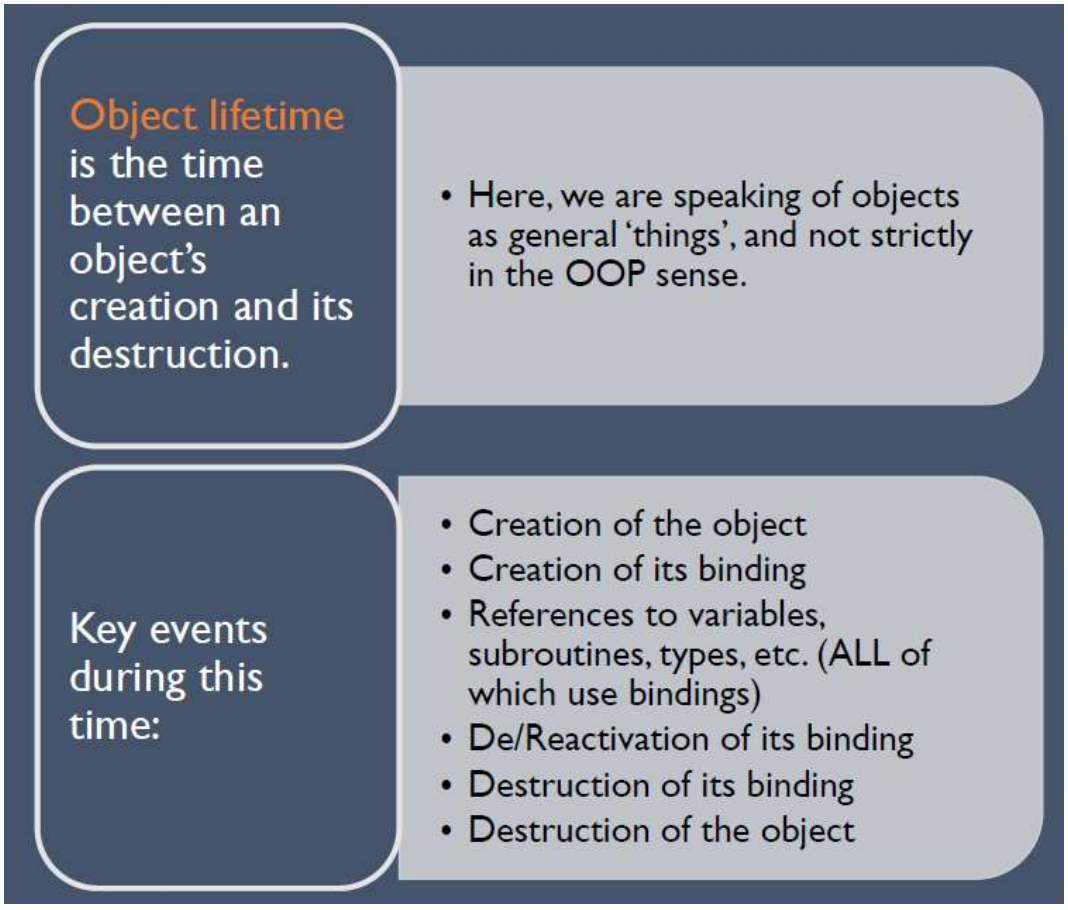
What is a run time?

Run time is a very broad term that covers the entire span from the beginning to the end of execution:

- program start-up time
- module entry time
- elaboration time (point a which a declaration is first "seen")
- procedure entry time
- block entry time
- statement execution time

Object Lifetime

- If object outlives binding it's garbage
- If binding outlives object it's a dangling reference

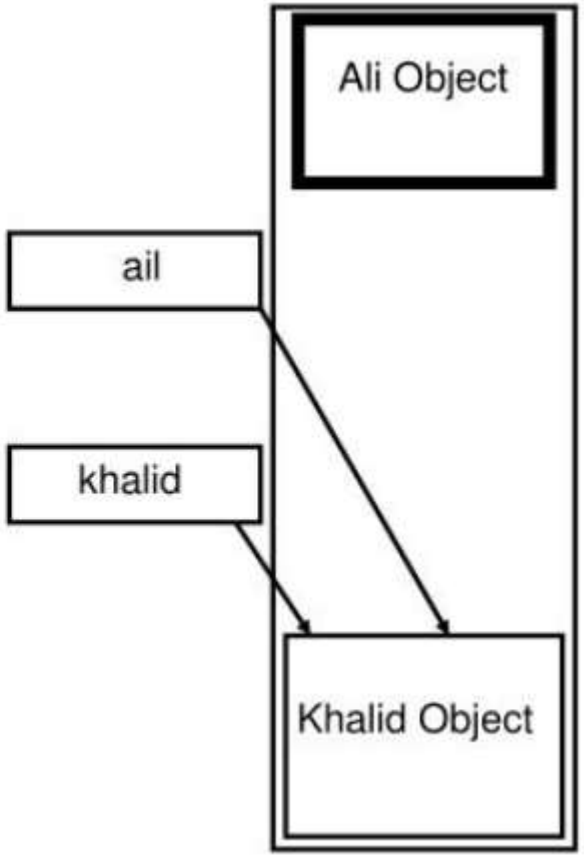


Garbage vs. Dangling Reference

Garbage: unreferenced objects

```
Student ali= new Student();
Student khalid= new Student();
ali=khalid;
```

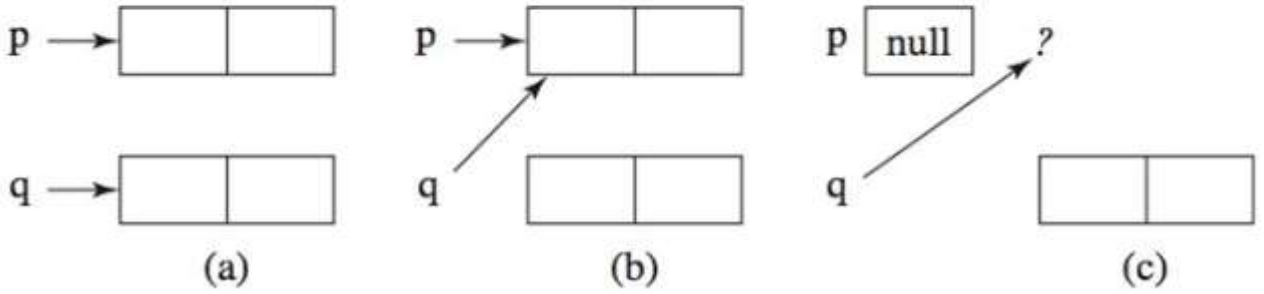
Now ali Object becomes a garbage,
It is unreferenced Object



Another Example of Garbage

```
class node {
    int value;
    node next;
}
node p, q;
```

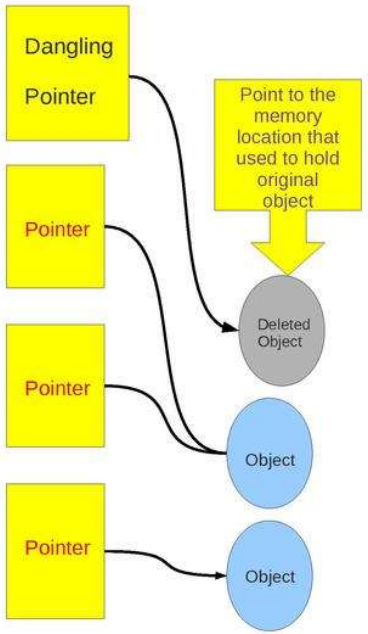
```
p = new node();
q = new node();
q = p;
delete p;
```



Garbage vs. Dangling Reference

Dangling Reference: Reference to a memory address that was originally allocated, but is now deallocated

```
int * p = new int;
delete p;

int i = *p; // error, p has been deleted!
```



Object Storage Management

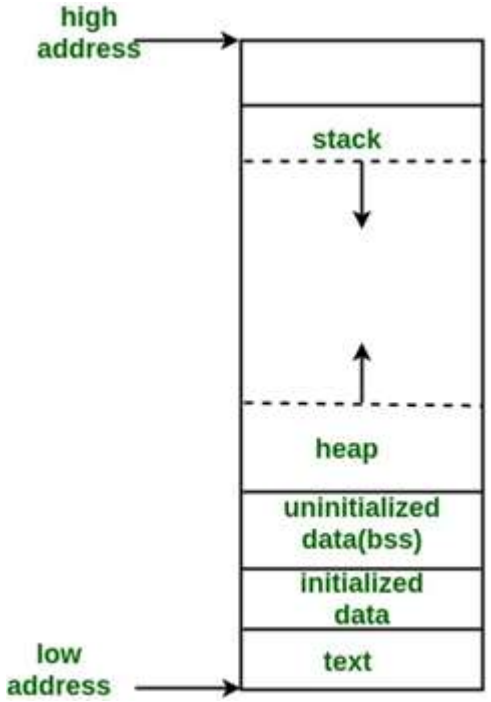
Storage Allocation mechanisms are used to manage the object's space during its lifetime:

- **Static** objects are given an absolute address that is retained throughout the program's execution
  - Global variables, subroutine code, class method code
- **Stack** objects are allocated and deallocated in last-in, first-out order, usually in connection with subroutine calls and returns
  - Subroutine arguments, local variables
- **Heap**: the objects may be allocated and deallocated at arbitrary times
  - Class instances in Java

Static Storage Allocation

- Small constants - often stored within the instruction itself
- Global variables
- static or own variables
- Explicit constants (including strings, sets, etc.), e.g., printf("hello, world\n")
- Arguments and return values
- *Temporaries* (intermediate values produced in complex calculations)

Memory Layout of C Program



Text segment: Contains executable instructions

Initialized data segment: Global variables and static variables initialized by the programmer

Unintialized data segment: Declared but not explicitly initialized variables

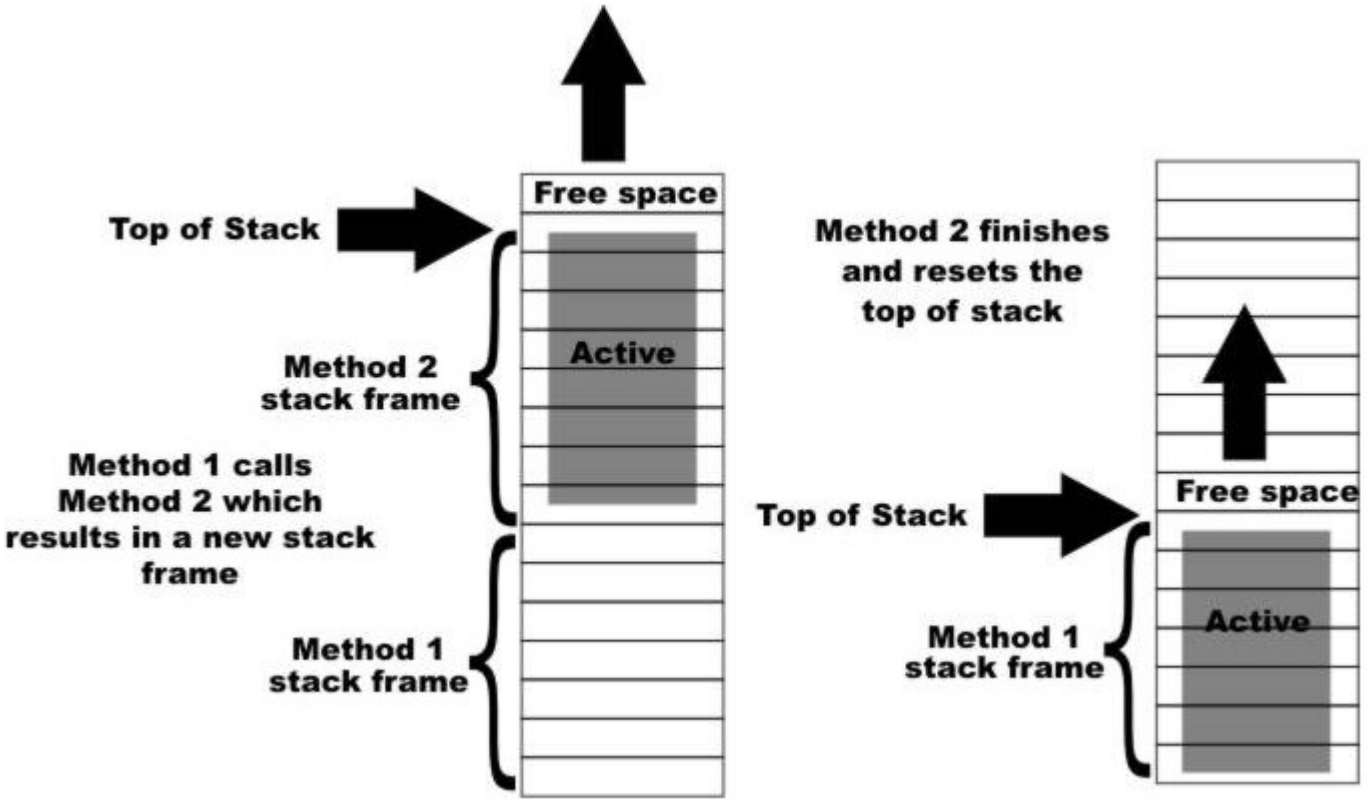
Stack: Starts from higher address and grows towards lower address. Saves information each time a function is called

Heap: Starts from lower address and grows towards higher address. Dynamic memory allocation takes place. Managed by malloc, realloc and free.

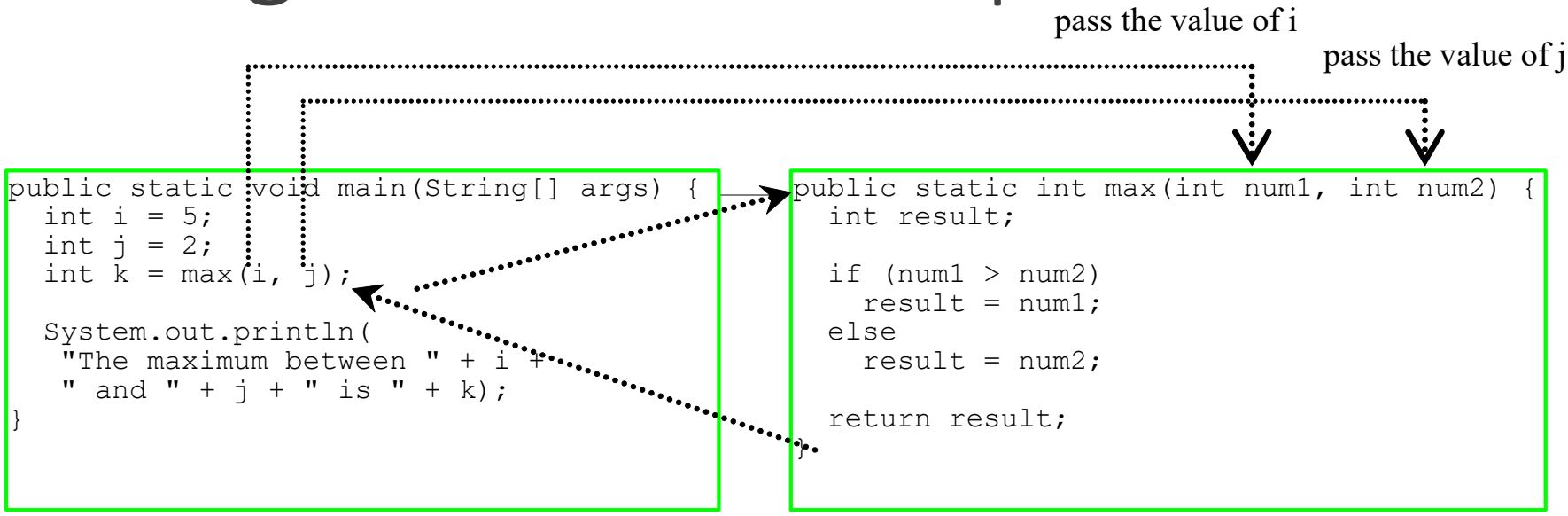
Stack Based Storage Management

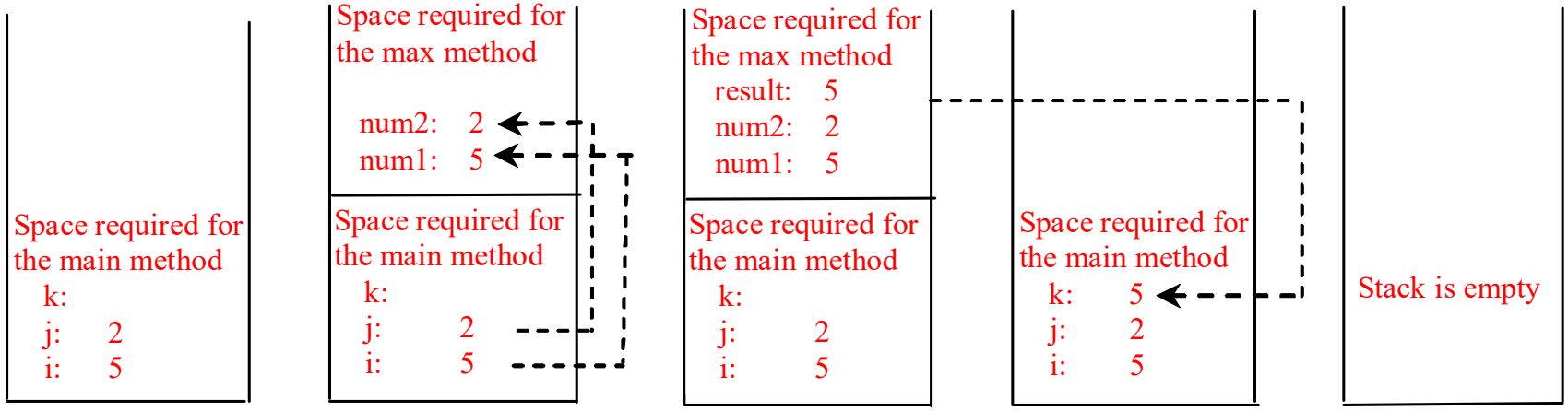
- Why a **stack**?
  - allocate space for recursive **routines**
  - the way subroutines call each other (or themselves) can be represented in a stack in a very natural way.
  - reuse space
- Each instance of a subroutine at run time has its own *frame* (or *activation record*) for:
  - parameters
  - local variables
  - return address

Stack Based Storage Management



Calling Methods Example in Java





(a) The main method is invoked.

(b) The max method is invoked.

(c) The max method is being executed.

(d) The max method is finished and the return value is sent to k.

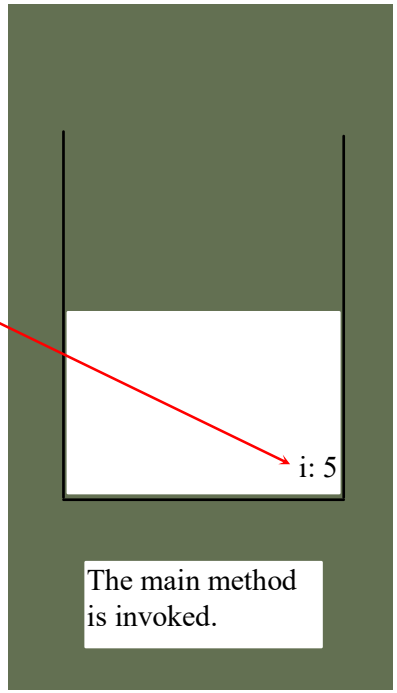
(e) The main method is finished.

Trace Call Stack

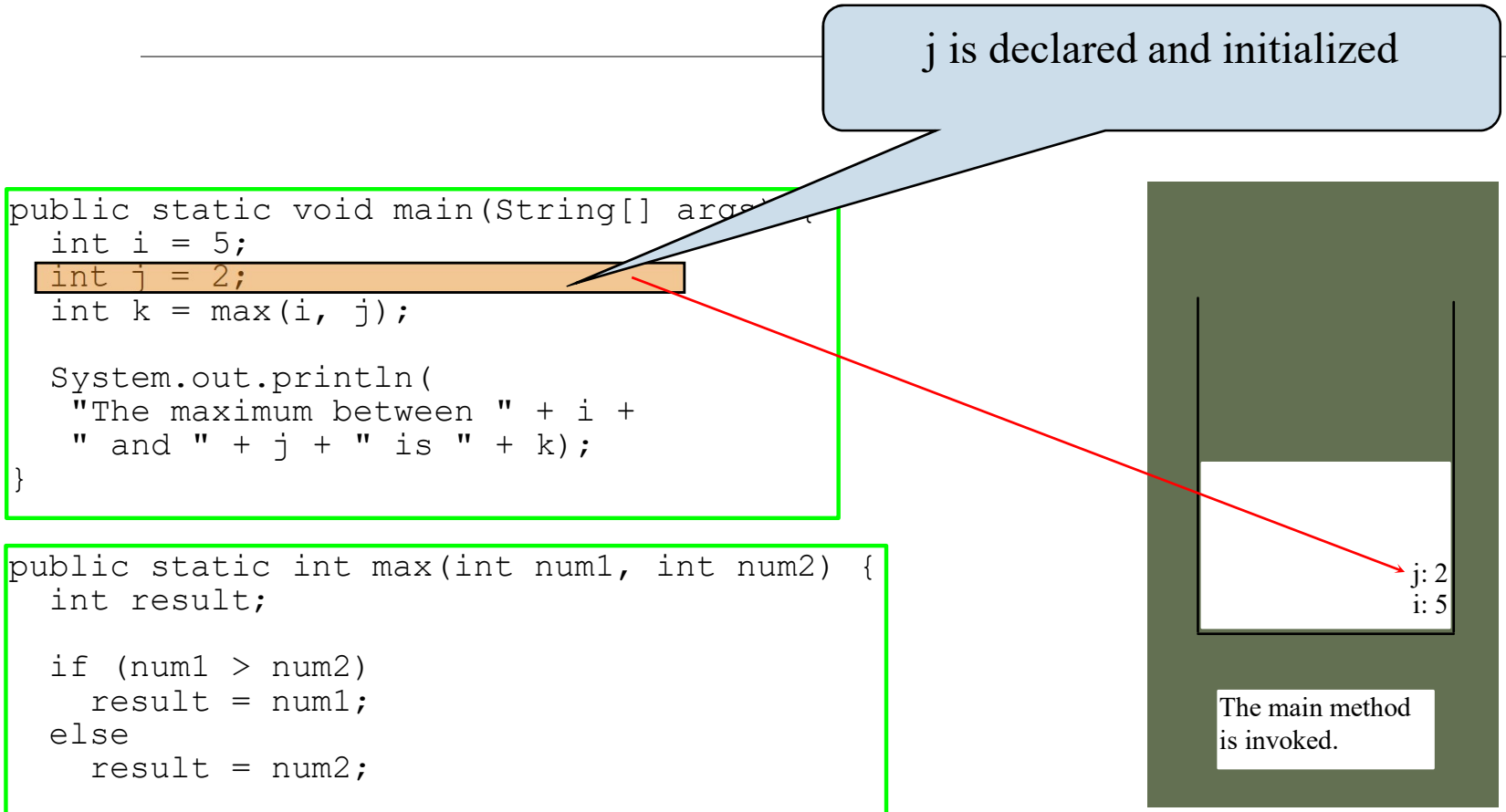
i is declared and initialized

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

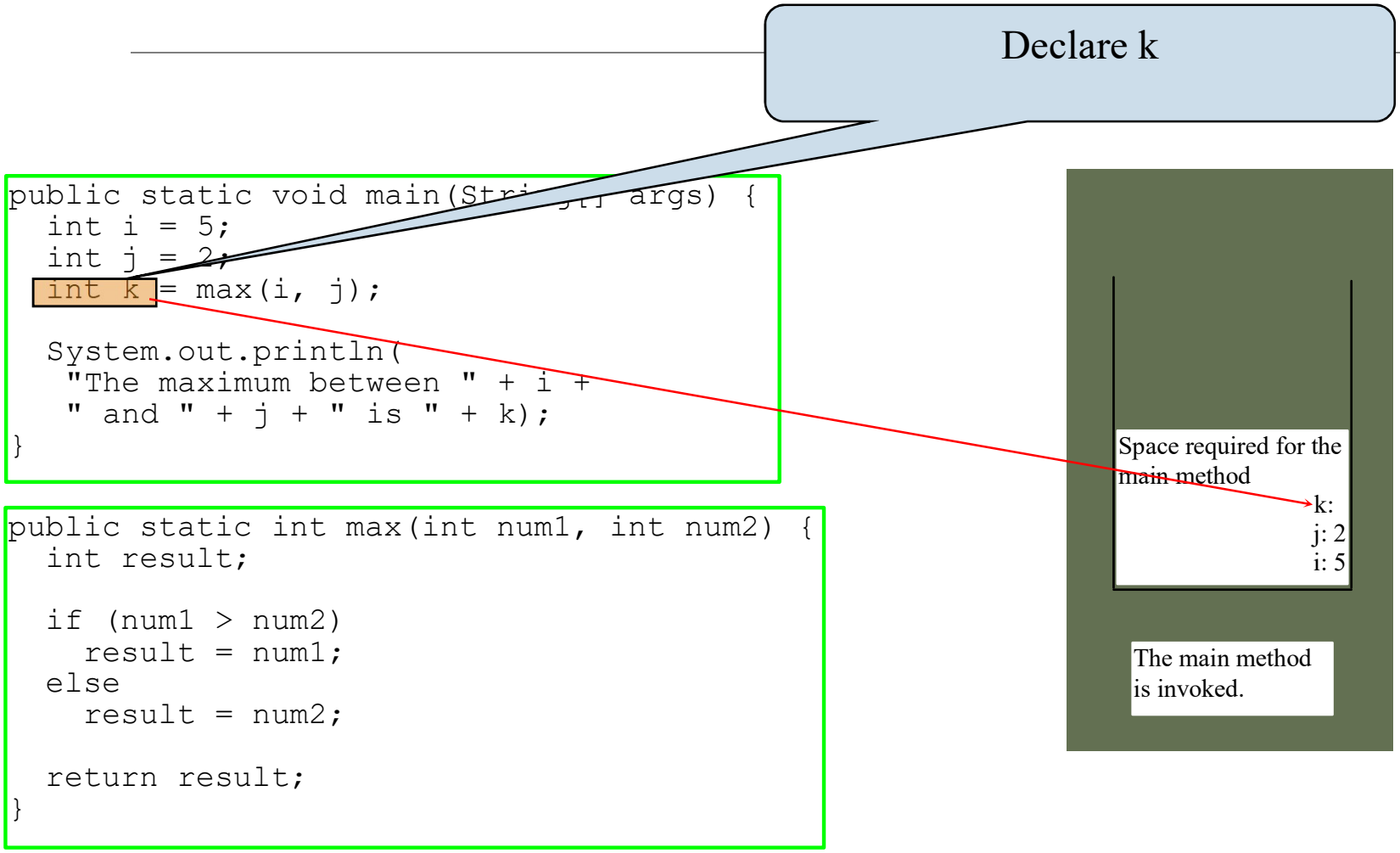
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Call Stack



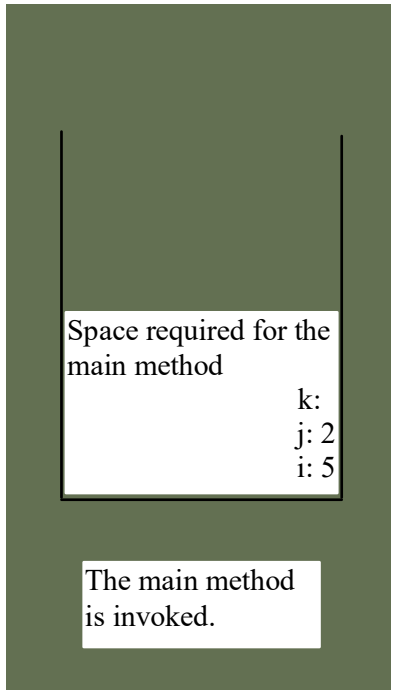
Trace Call Stack



Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

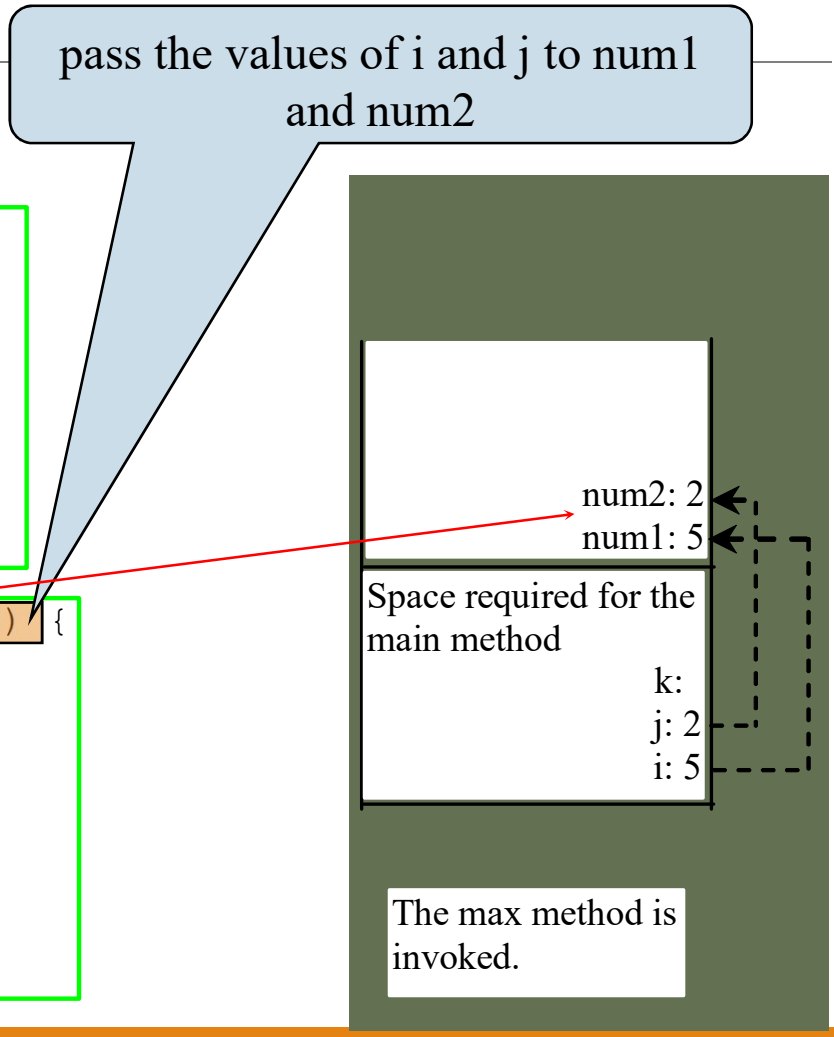
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

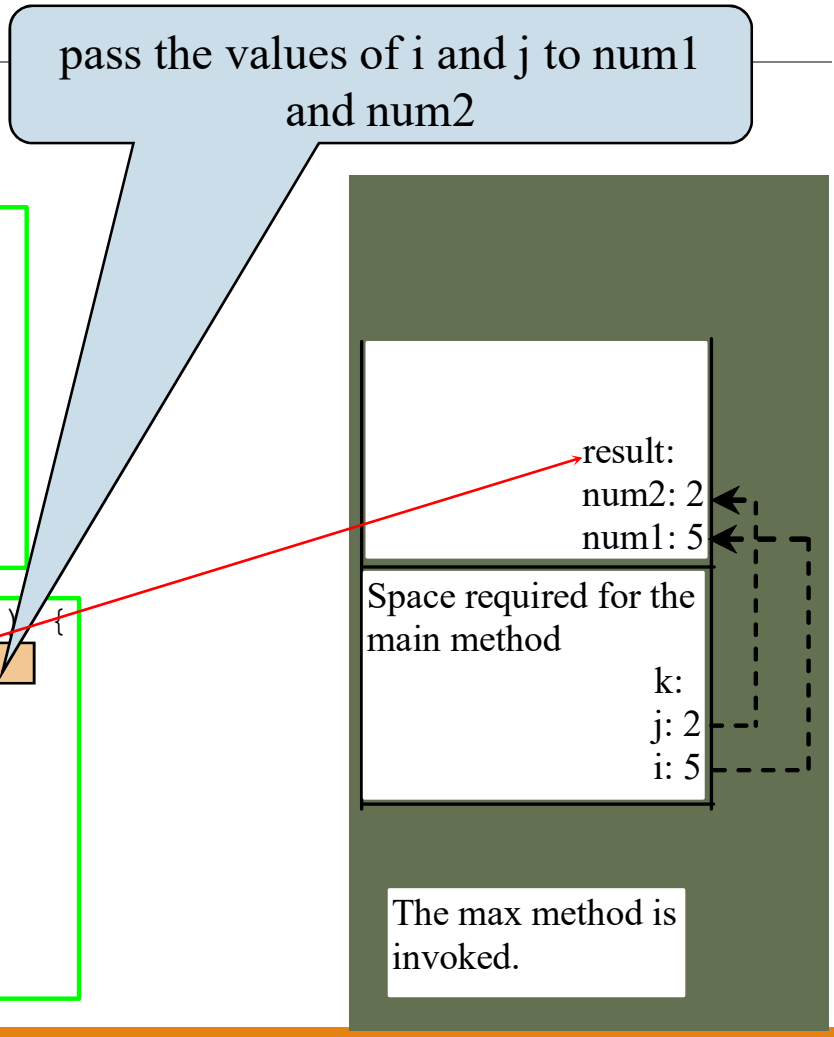
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

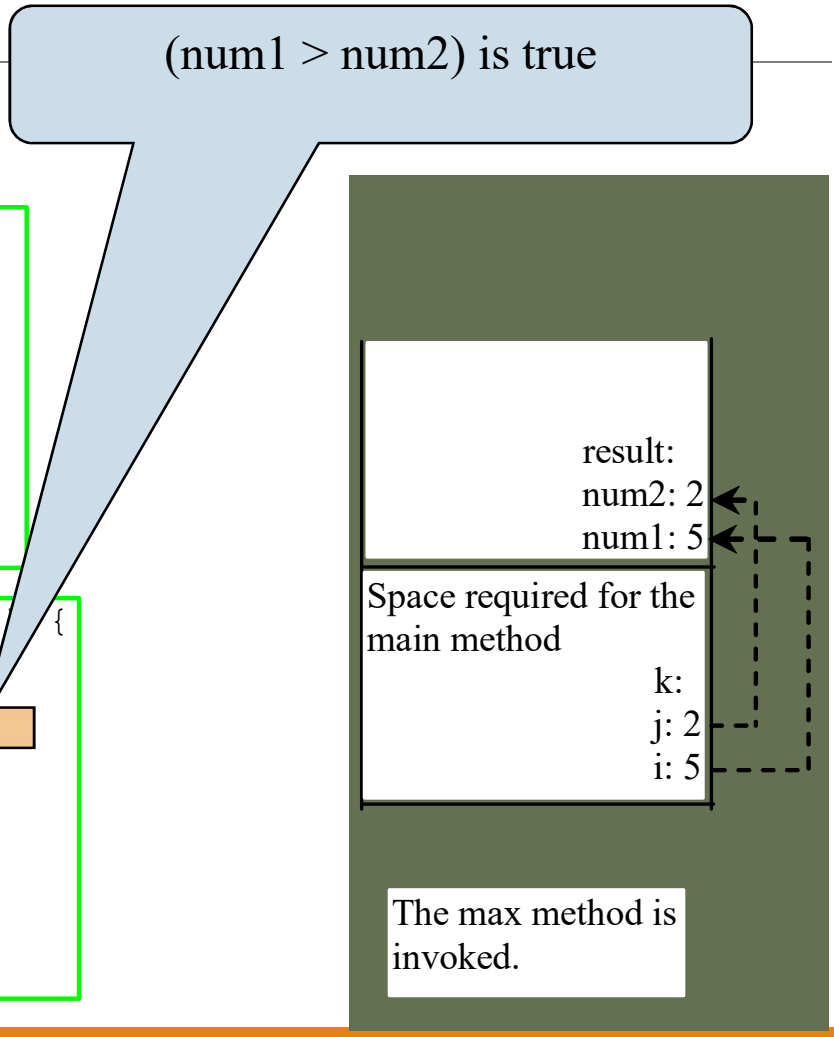
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

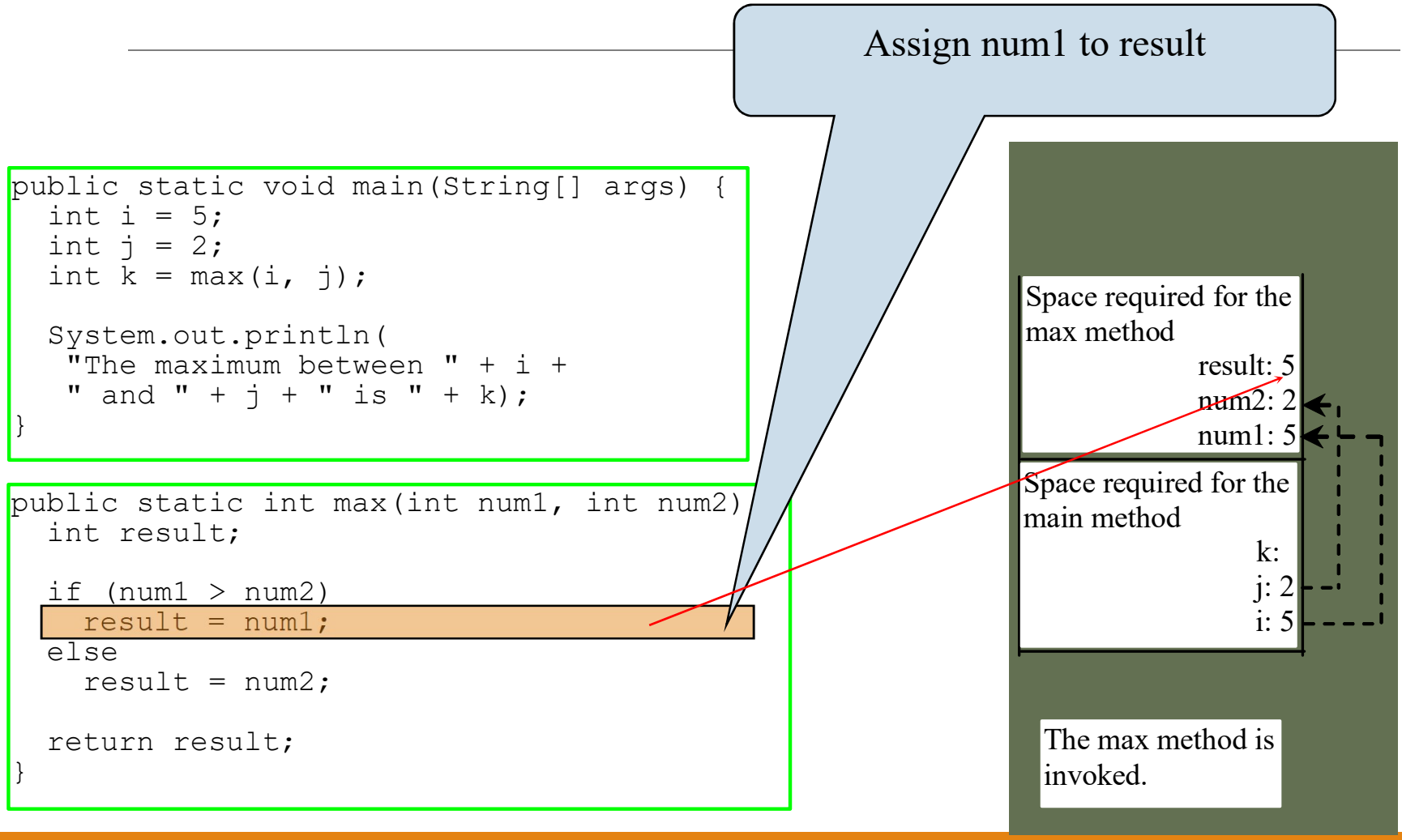
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



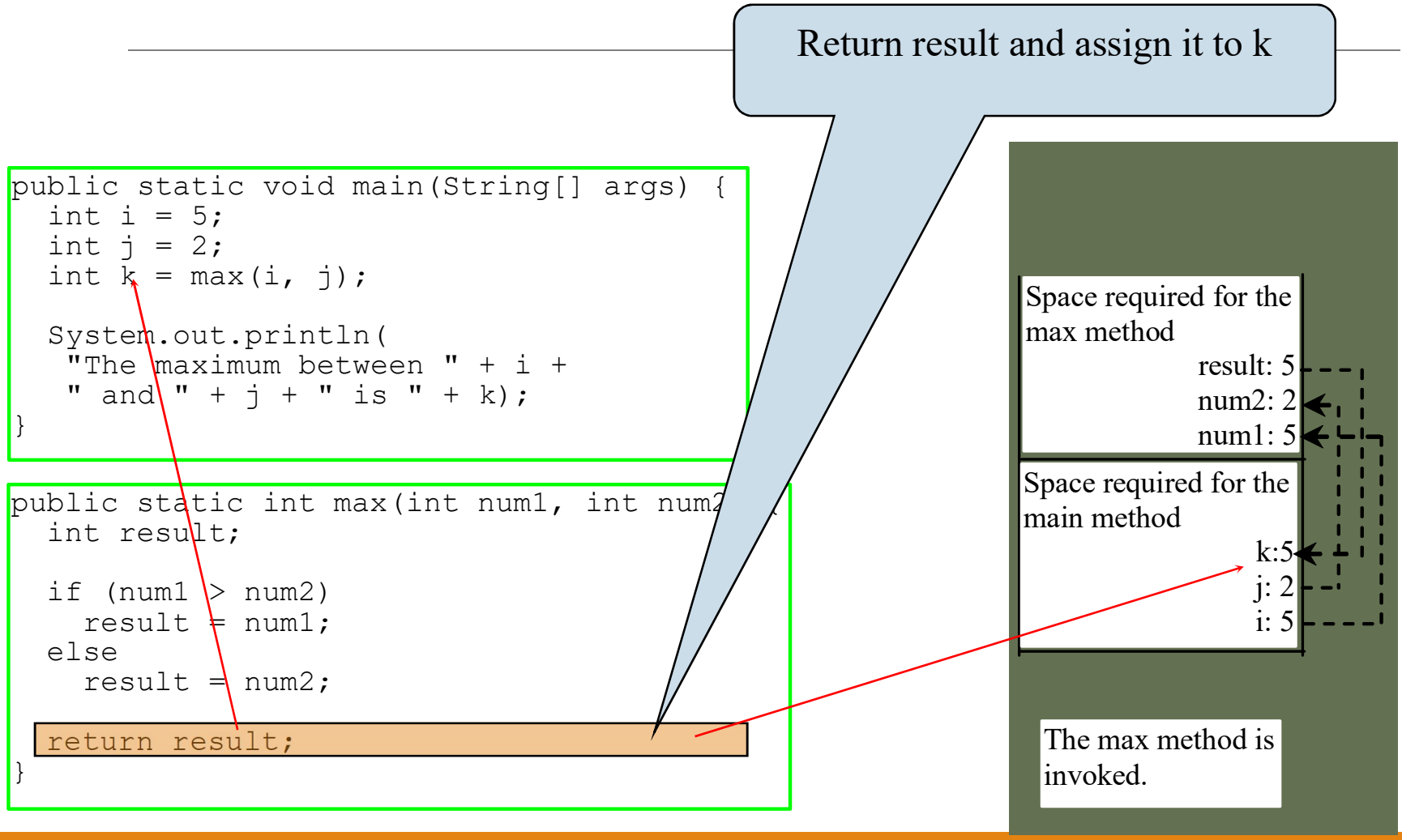
Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2)
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



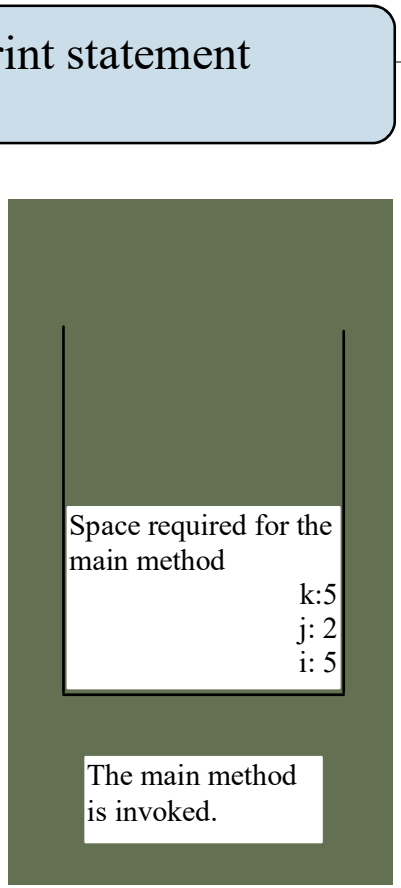
Trace Call Stack



Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Stack Based Storage Management

Stack pointers:

- The *frame pointer* (fp) register points to a known location within the frame of the current subroutine
- The *stack pointer* (sp) register points to the first unused location on the stack (or the last used location on some machines)

Stack based allocation of space

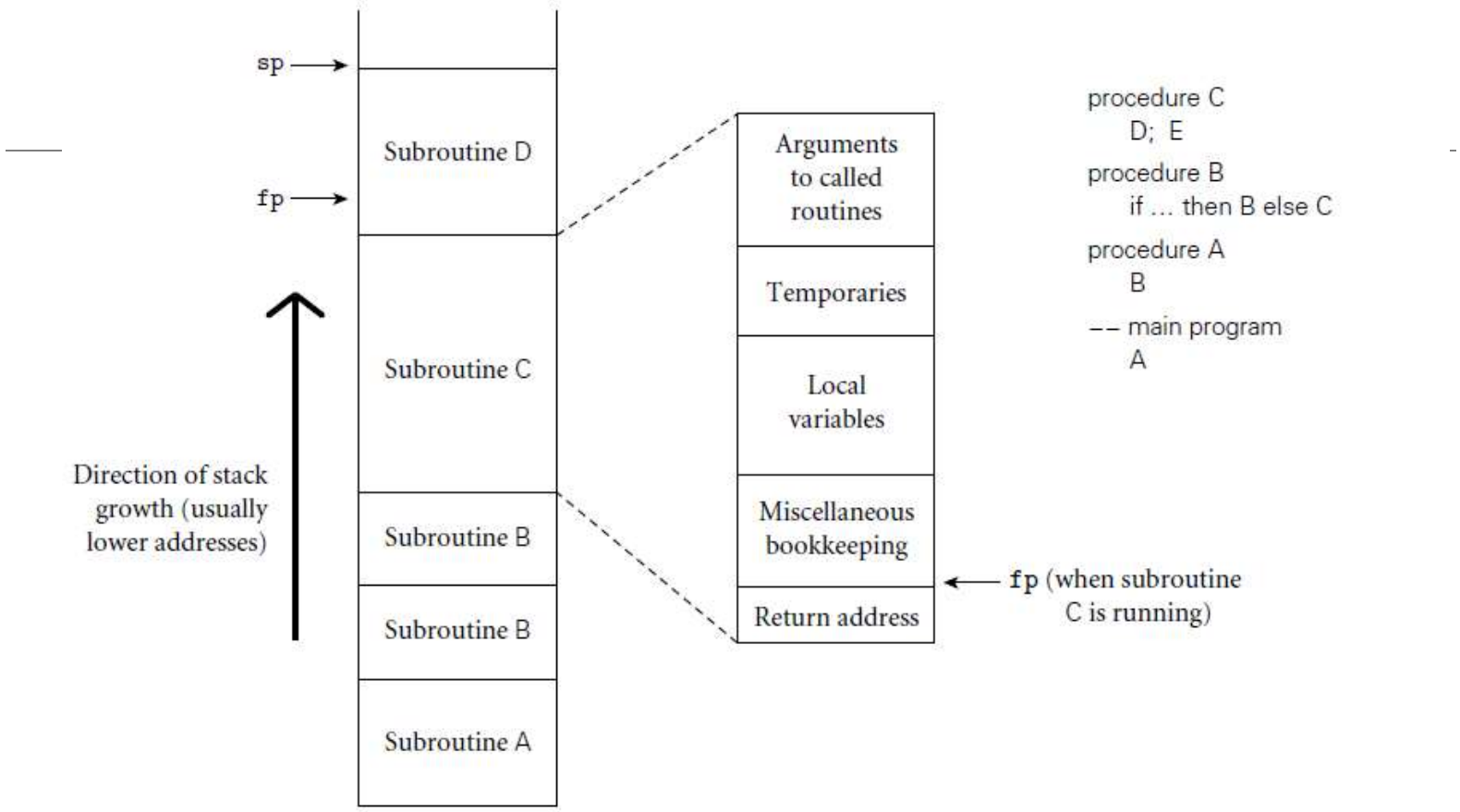
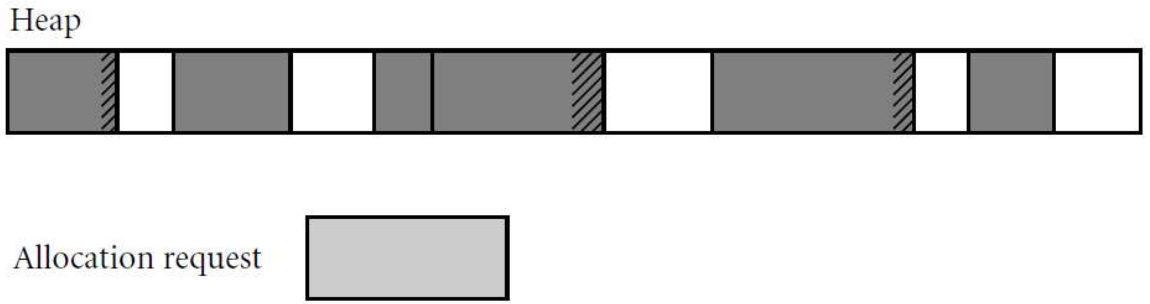


Figure 3.1 **Stack-based allocation of space for subroutines.** We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Heap-Based Storage Management

- A heap is a region of storage in which sub-blocks can be allocated and deallocated at arbitrary times
- Dynamically allocated pieces of data structures: objects, Strings, lists, and sets, whose size may change as a result of an assignment statement or other update operation
- Two concerns with heap space management: Speed and Space

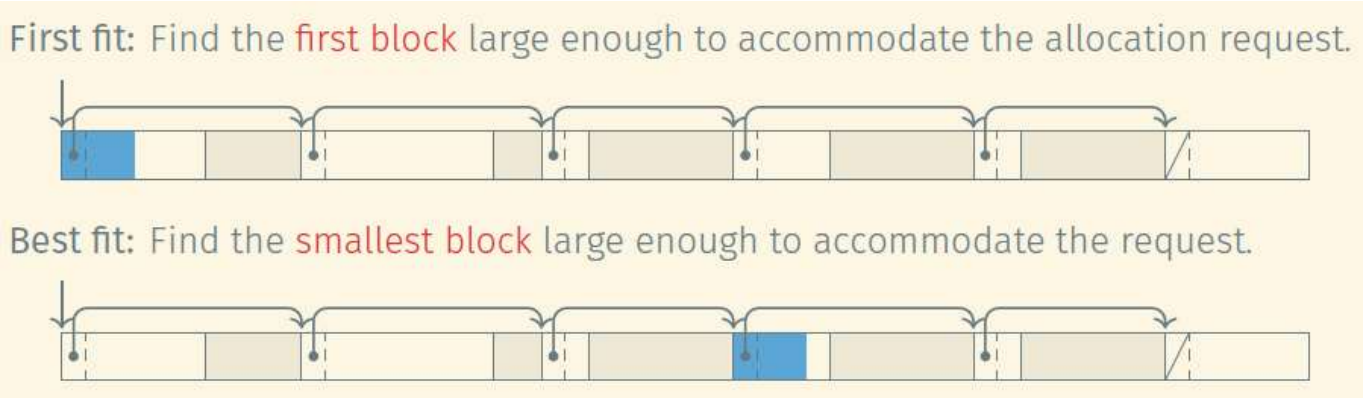


Heap Management

Free list: List of blocks of free memory

The allocation algorithm searches for a block of adequate size to accommodate the allocation request

- Find a free block that is at least as big as the requested amount of memory
- Mark requested number of bytes (plus padding) as allocated
- Return rest of the free block to free list



Heap Fragmentation

- *Internal fragmentation* occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object
  - e.g. Boolean is stored in 1 bit/1 byte
- *External fragmentation* occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks
  - There may be quite a lot of free space, but no one piece of it may be large enough to satisfy some request

Heap Compaction

To fight fragmentation, some memory management algorithms perform heap compaction once a while

Mark-Compact Algorithm

- Mark reachable objects
- Relocate the marked objects towards the beginning of the heap area