Names, Scopes, and Bindings

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

Name, Scope, and Binding

- *Names* are identifiers (mnemonic character strings used to represent something in the program instead of low-level concepts like addresses):
  - A name can also represent an *abstraction* of a complicated program fragment (e.g., name of a method (*control abstraction*), class *(data abstraction),* module).
  - Some symbols (like '+') can also be names
- A *binding* is an association between two things, such as a name and the thing it names
  - In general, *binding time* refers to the notion of resolving any design decision in a language implementation

Name, Scope, and Binding

- The *scope* of a binding is the part of the program (textually)in which the binding is active.
- The complete set of bindings in effect at a given point in a program is known as the current *referencing environment*.

Binding

- *Binding Time* is the point at which a binding is created or, more generally, the point at which any implementation decision is made.
- There are many times when decision about the binding are taken:
  - language design time: the control flow constructs, the set of fundamental (primitive) types, the available constructors for creating complex types, and many other aspects of language semantics are chosen when the language is designed
  - language implementation time: precision (number of bits) of the fundamental types, the coupling of I/O to the operating system's notion of files, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow.

Binding

- program writing time: programmers choose algorithms and names
- compile time: compilers plan for data layout (the mapping of high-level constructs to machine code, including the layout of statically defined data in memory)
- link time: layout of whole program in memory (virtual addresses are chosen at link time), the linker chooses the overall layout of the modules with respect to one another, and resolves intermodule references
- load time: choice of physical addresses (the processor's memory management hardware translates virtual addresses into physical addresses during each individual instruction at run time)

Binding

- Run time decisions about binding:
  - value/variable bindings, sizes of strings
  - execution flow

    - Note: *run time* is a very broad term that covers the entire span from the beginning to the end of execution. It subsumes:
      - 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
    - The terms *STATIC* and *DYNAMIC* are generally used to refer to things bound before run time and at run time.

Binding

- In general, **early binding times are associated with greater efficiency**
- **Later binding times are associated with greater flexibility**

- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times

- Some languages try to do both (e.g., Java JVM)

Lifetime and Storage Management

- Key events:
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects

Lifetime and Storage Management

- The period of time between the creation and the destruction of a name-to-object binding is called the binding's *lifetime* :
  - If object outlives binding it's *garbage*
  - If binding outlives object it's a *dangling reference*, e.g., if an object created via the C++ new operator is passed as a & parameter and then deallocated (delete-ed) before the subroutine returns.
- The textual region of the program in which the binding is active is its *scope*

Lifetime and Storage Management

- *Storage Allocation* mechanisms are used to manage the object's space:
  - Static: the objects are given an absolute address that is retained throughout the program's execution
  - Stack: the objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.
  - Heap: the objects may be allocated and deallocated at arbitrary times (require a complex storage management mechanism).

Lifetime and Storage Management

- **Static allocation** for:
  - code (and small constants - often stored within the instruction itself)
  - globals
  - static or own variables
  - explicit constants (including strings, sets, etc.), e.g., printf("hello, world\n") (called *manifest constants* or *compile-time constants).*
  - *Arguments and return values:* Fortran (Fortran didn't have subroutine recursion) and Basics (Basic didn't have function-level scopes).
  - *Temporaries* (intermediate values produced in complex calculations)
  - *Bookkeeping information* (subroutine's return address, a reference to the stack frame of the caller (the *dynamic link*), additional saved registers, debugging information

Lifetime and Storage Management

- **Stack**:
  - Why a **stack**?
    - allocate space for recursive routines
    - reuse space
  - Each instance of a subroutine at run time has its own *frame* (an *activation record*):
    - parameters
    - local variables
    - temporaries

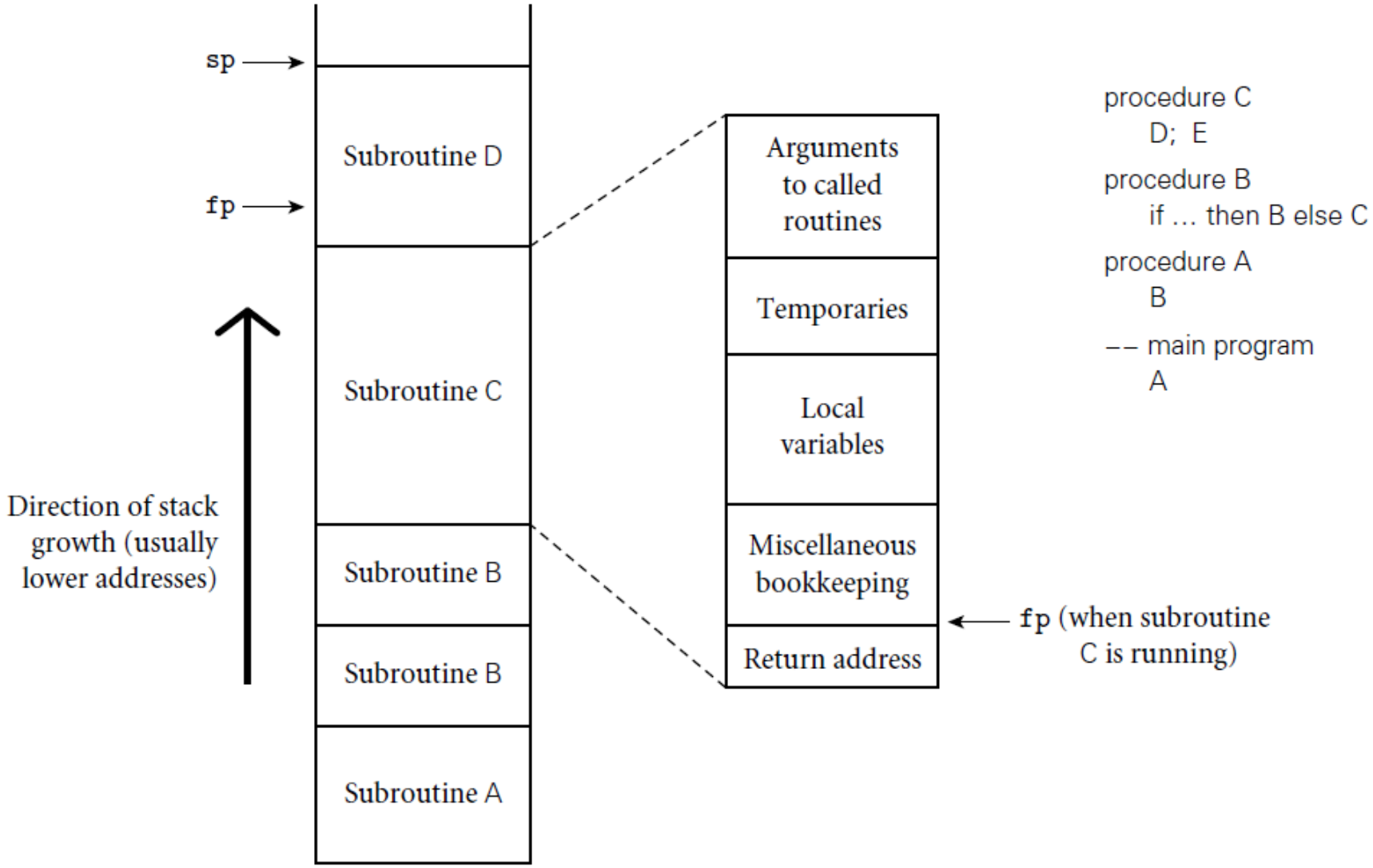
Lifetime and Storage Management

- Maintenance of the stack is the responsibility of the subroutine *calling sequence* (the code executed by the caller immediately before and after the call), the *prologue* (code executed at the beginning) and *epilogue* (code executed at the end) of the subroutine itself.

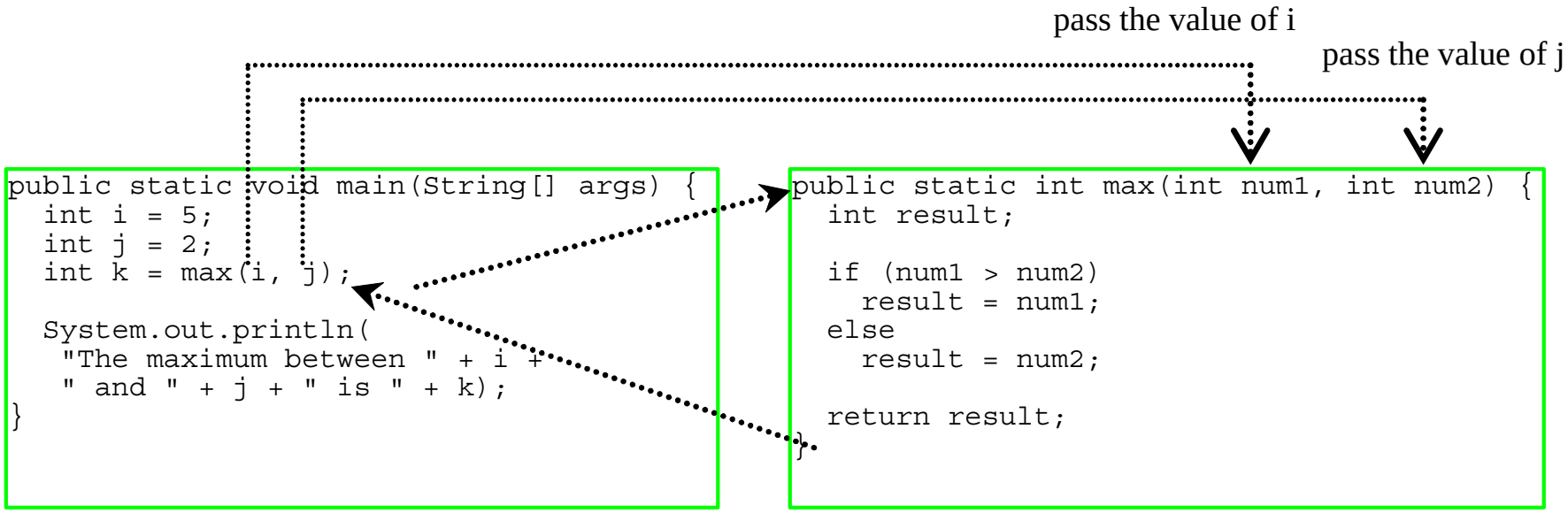
Lifetime and Storage Management

- Stack pointers:
  - The *frame pointer* (fp) register always points to a known location within the frame of the current subroutine.
    - fp points to the parameters (above the return address) of this call
  - The *stack pointer* (sp) register points to the first unused location on the stack (or the last used location on some machines)
    - sp would point to where arguments would be for next call
    - Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time.

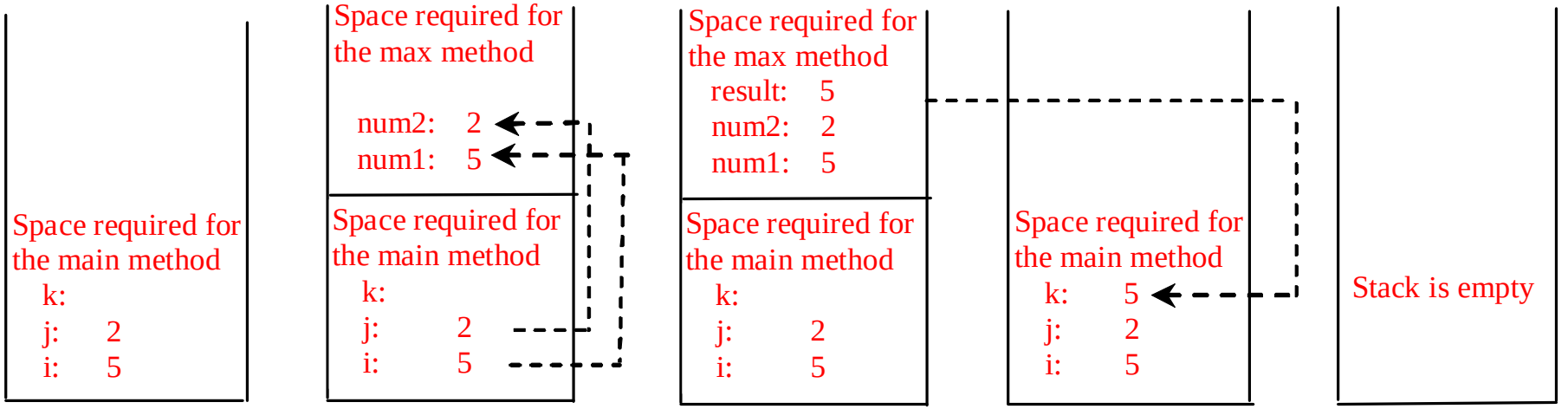
Lifetime and Storage Management



Calling Methods Example in Java



Call Stacks



(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

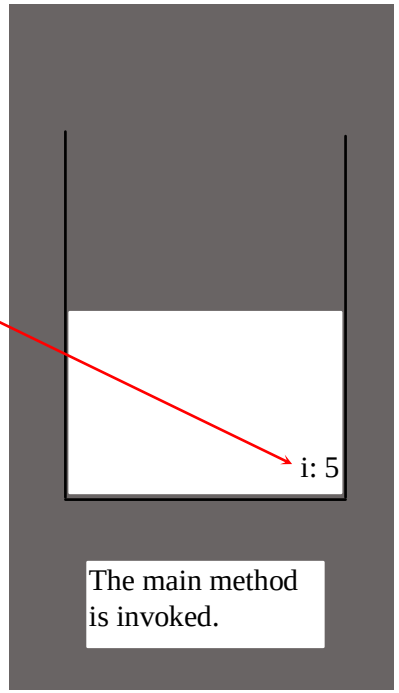
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  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

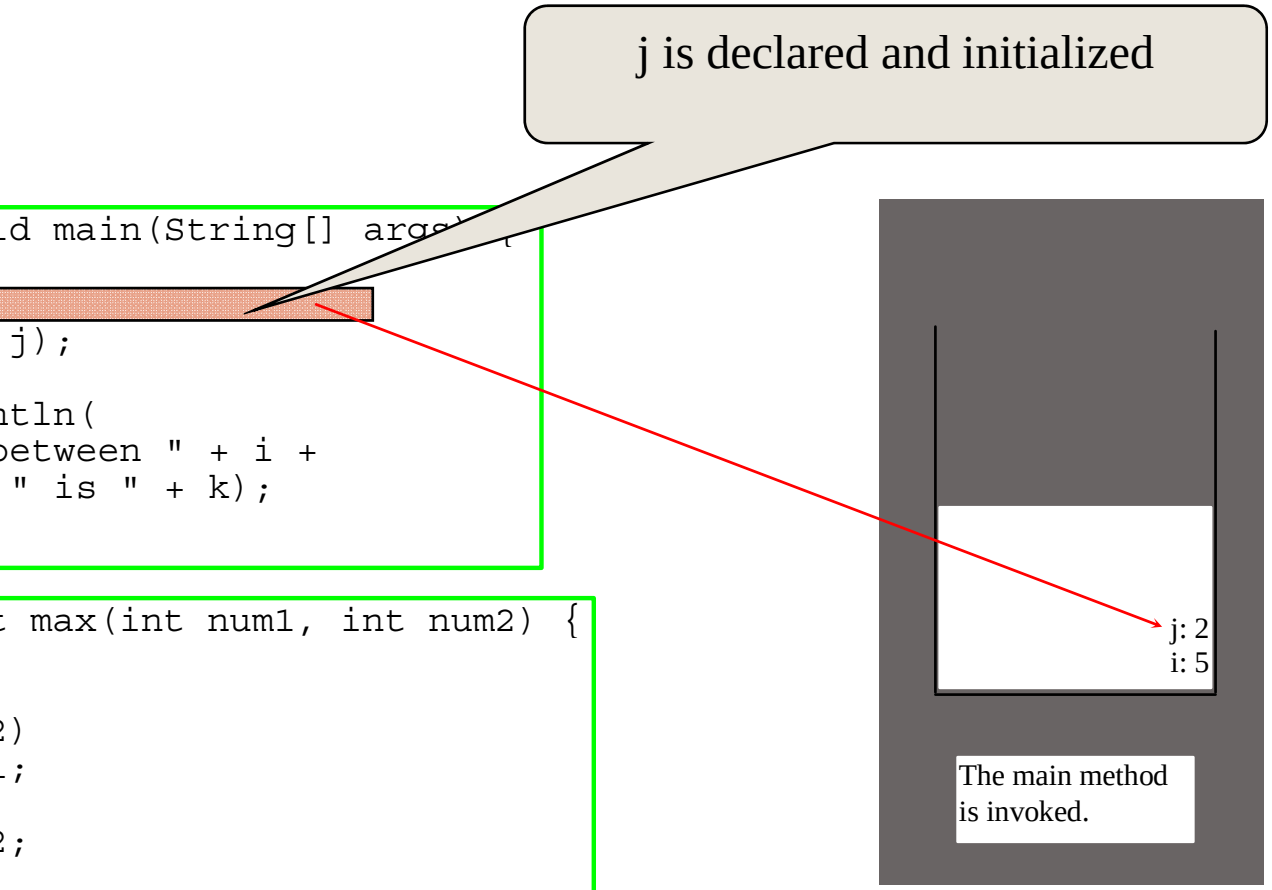
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  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

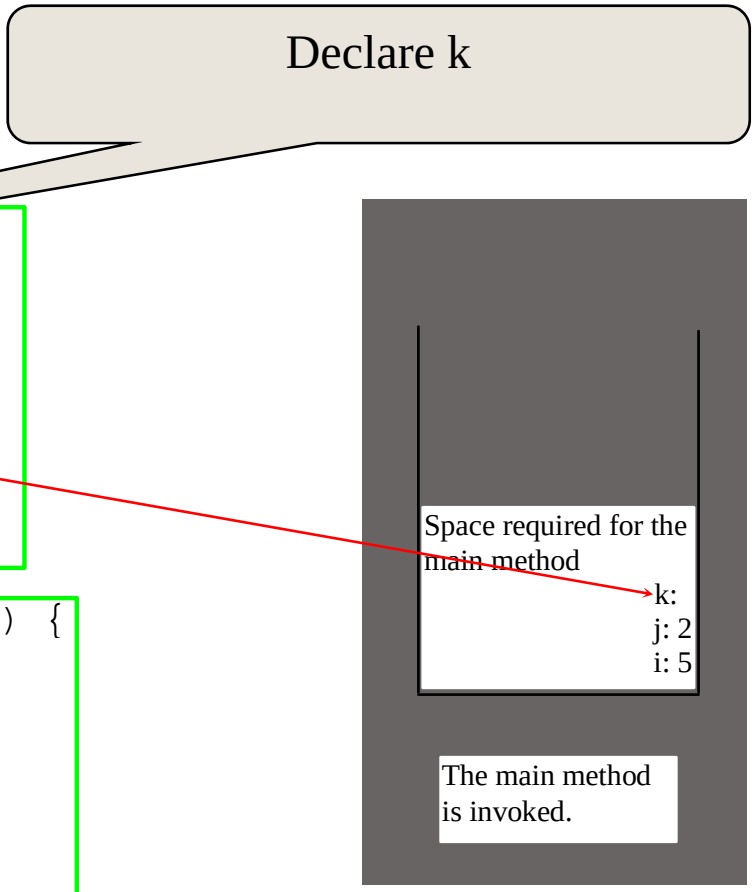
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

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

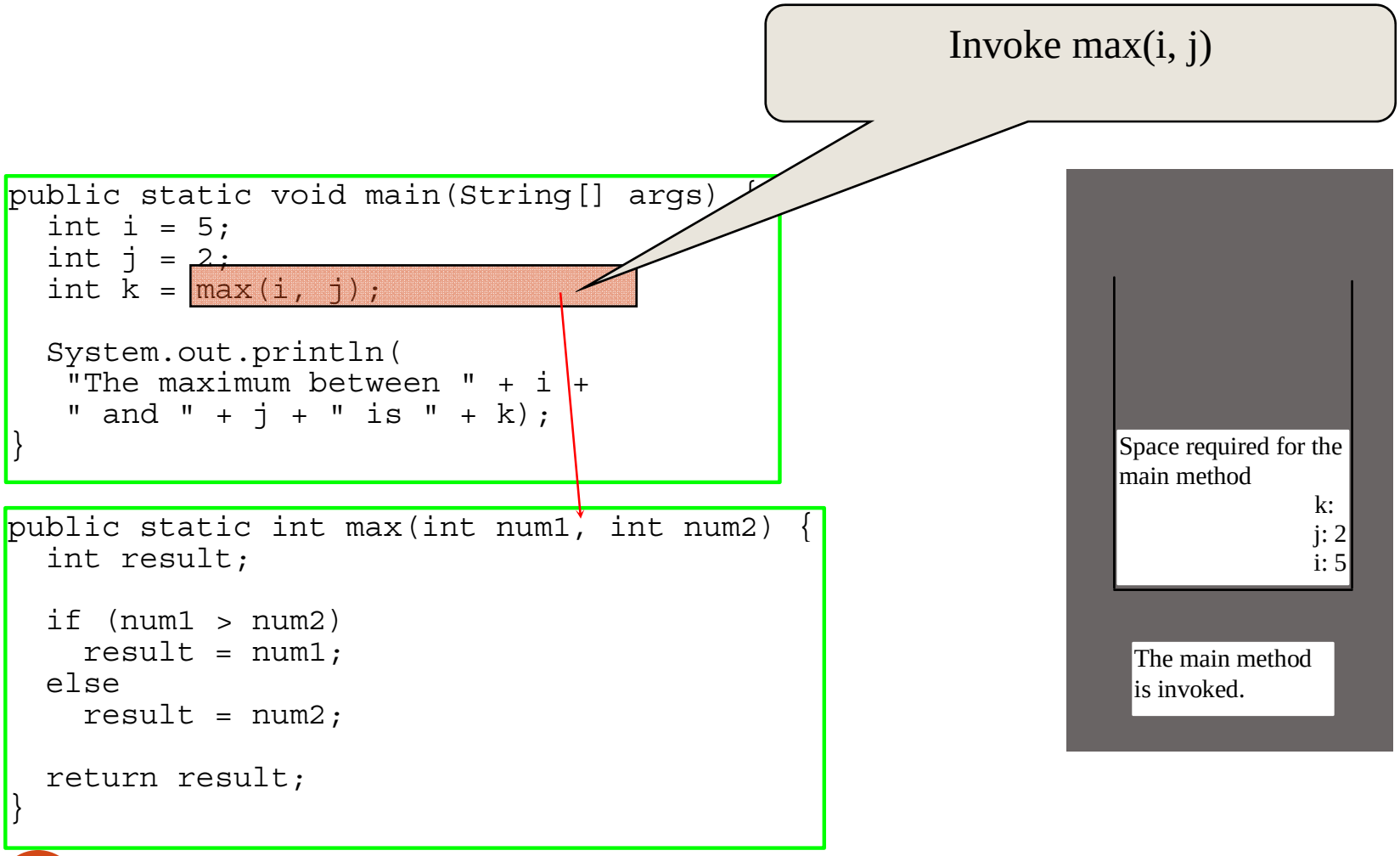
```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```



Trace Call Stack



Trace Call Stack

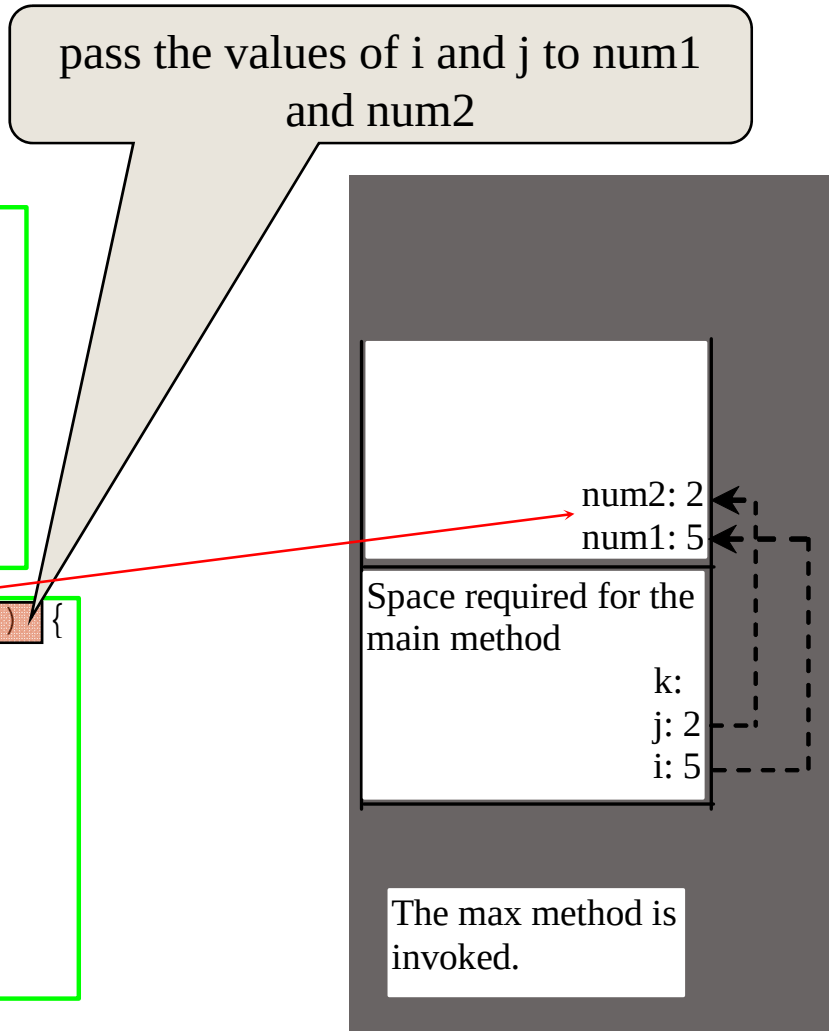
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  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;
}
```



Trace Call Stack

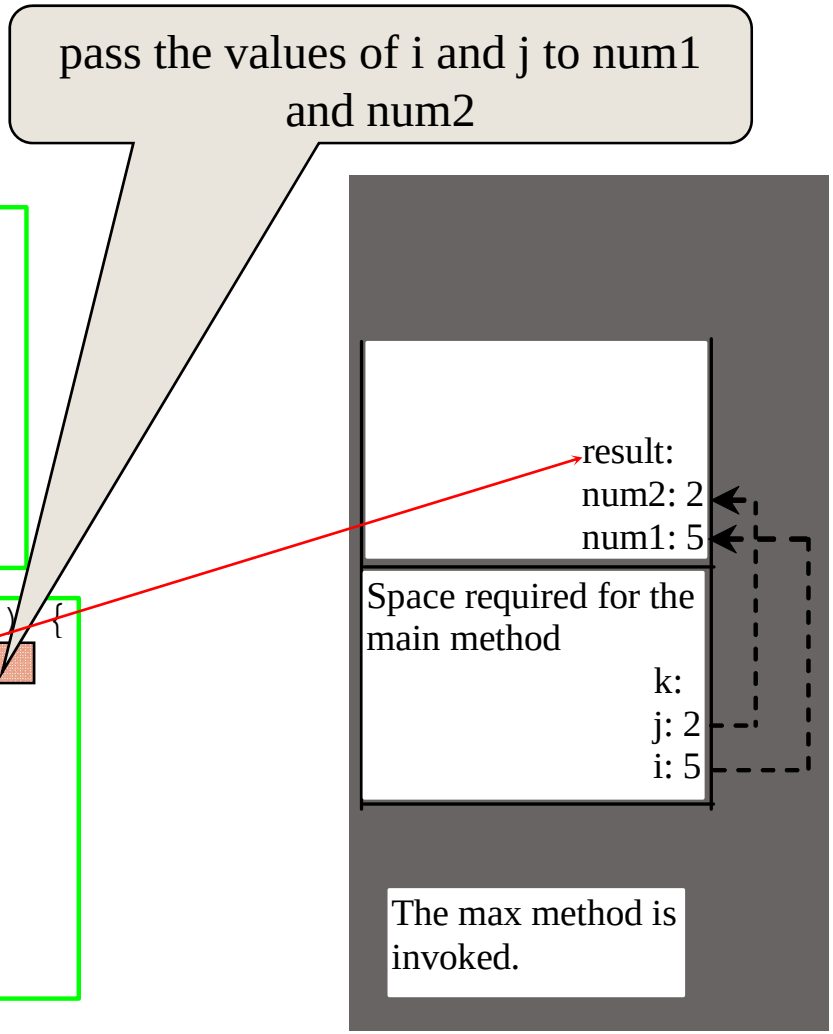
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  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

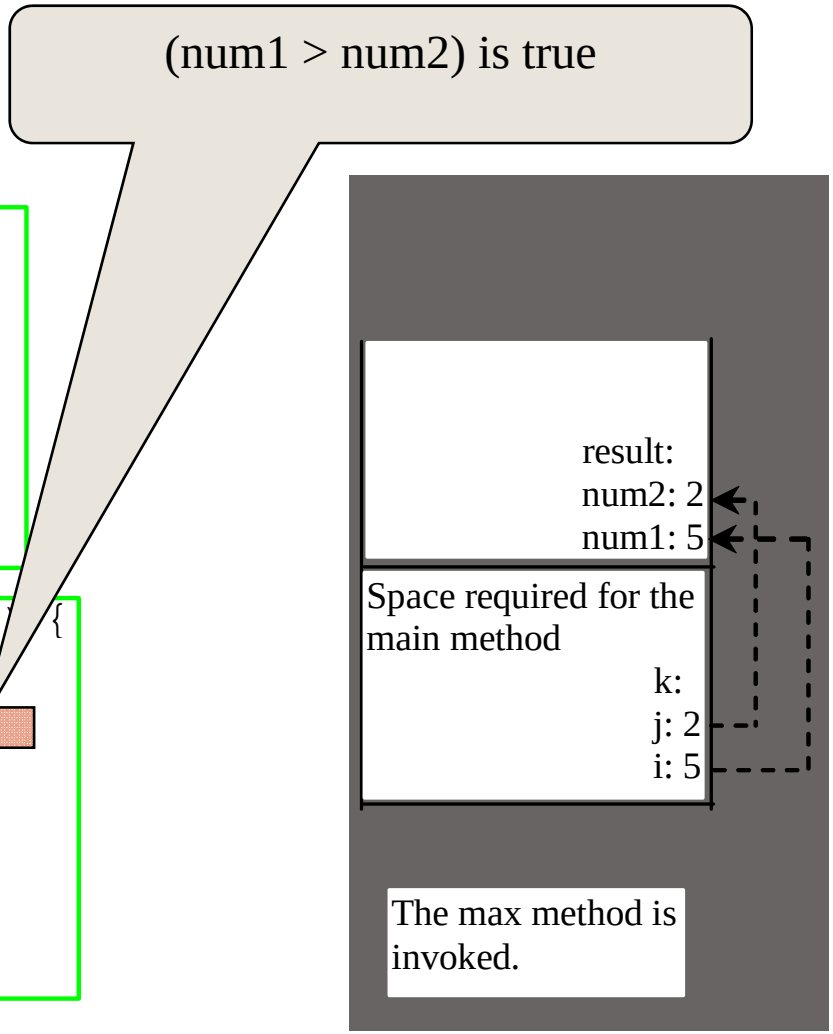
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  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

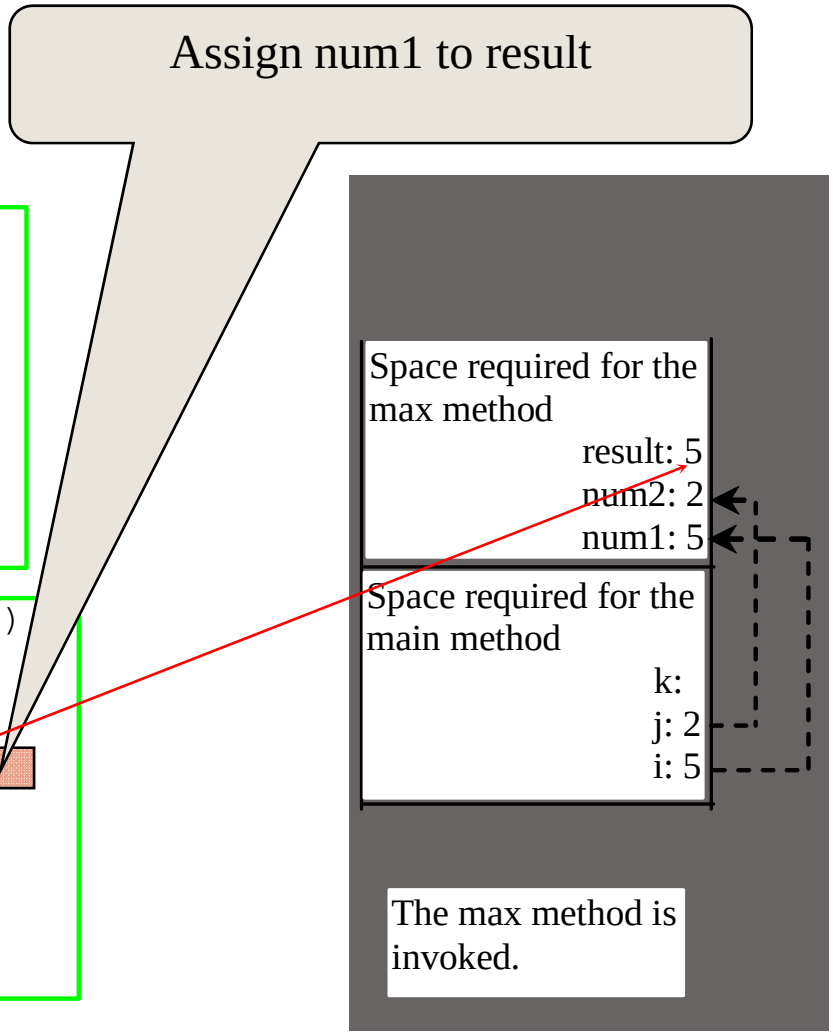
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

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

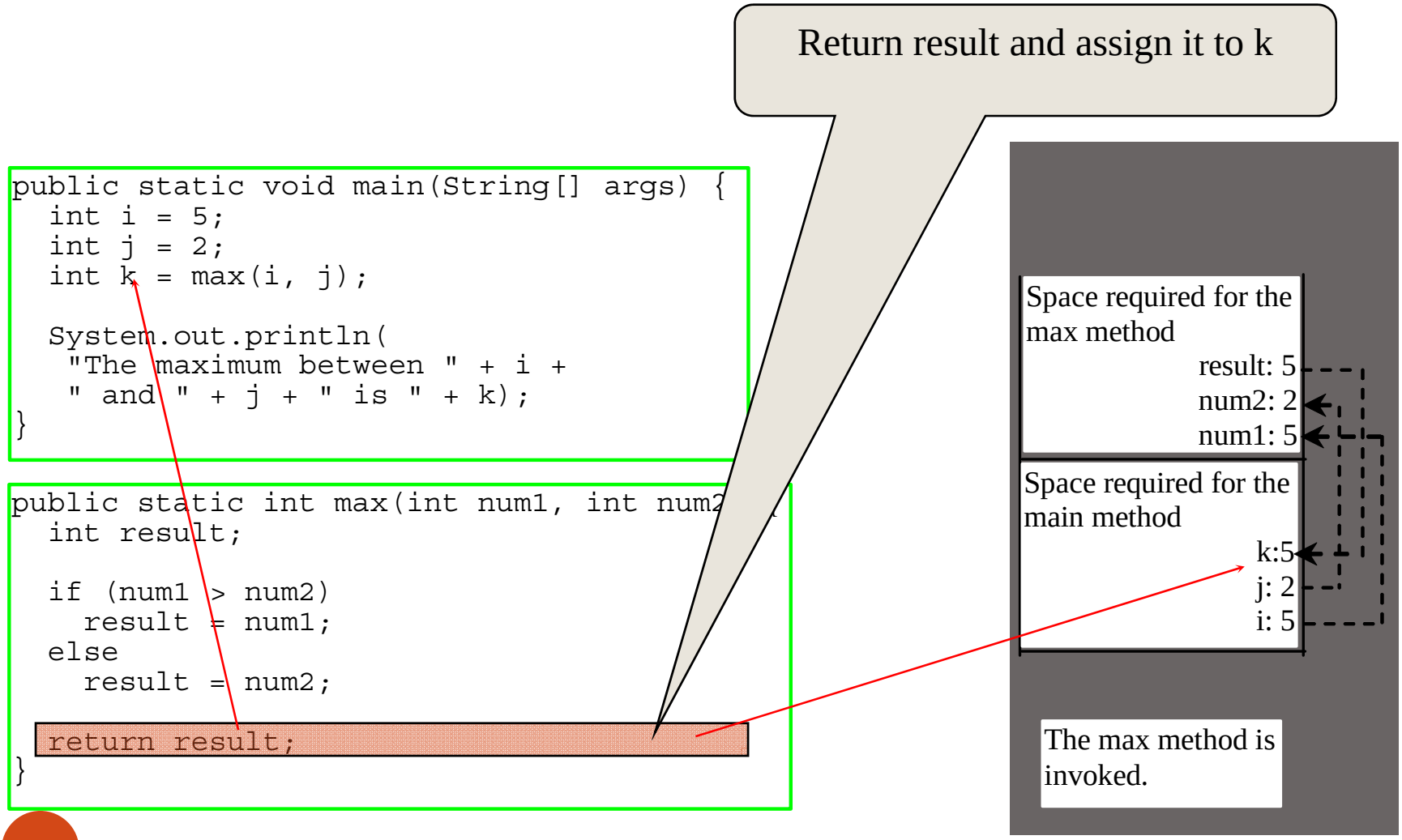
```
public static int max(int num1, int num2)
  int result;

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

  return result;
}
```



Trace Call Stack



Trace Call Stack

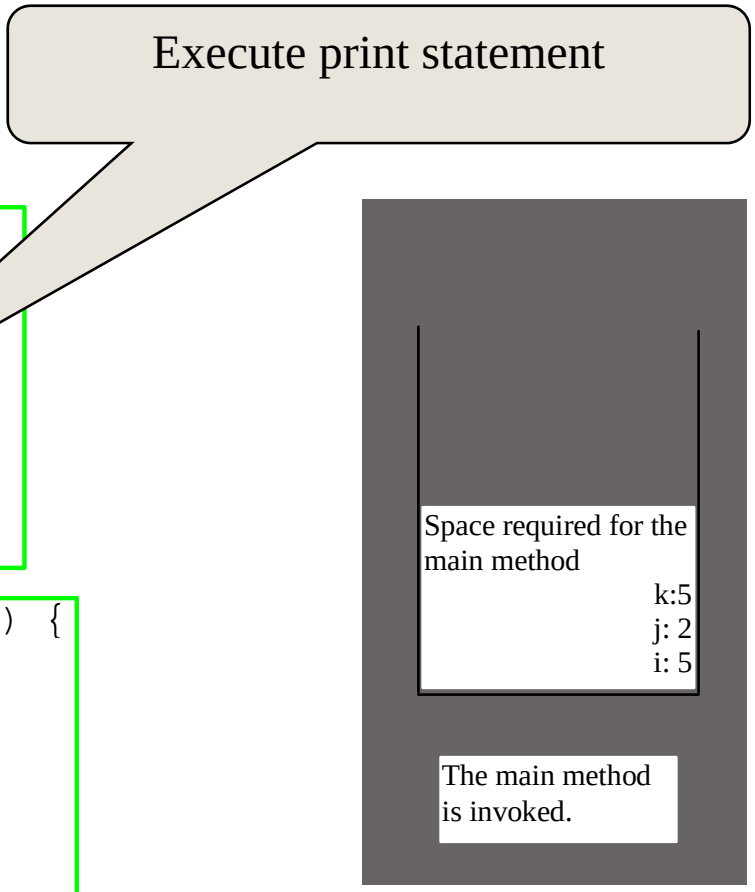
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  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;
}
```



Lifetime and Storage Management

- Reminder: maintenance of stack is responsibility of calling sequence and subroutine prolog and epilog.
- Optimizations:
  - space is saved by putting as much in the prolog and epilog as possible
  - time may be saved by
    - putting stuff in the caller instead

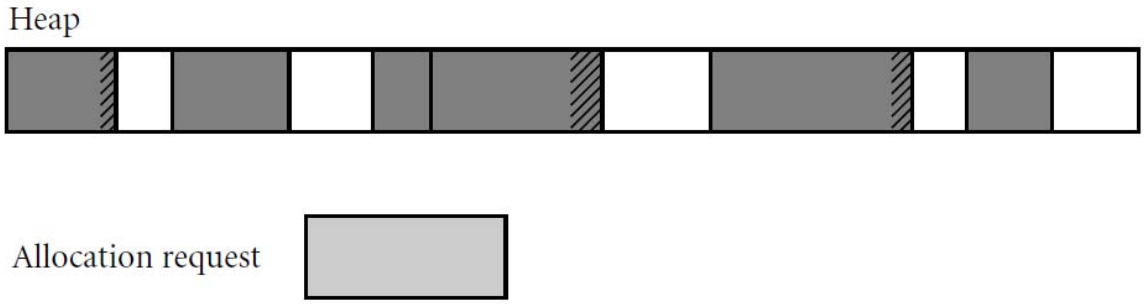
      or
    - combining what's known in both places (interprocedural optimization)
    - Unfolding subroutines (e.g., Prolog/Datalog unfolding optimizations)
- Note: one cannot return references to objects on the stack
  - Rookie mistake in C: the lifetime is limited to function scope.

Lifetime and Storage Management

- Heap for dynamic allocation:
  - A heap is a region of storage in which subblocks 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.

Lifetime and Storage Management

- *Fragmentation*:
  - *Internal fragmentation* occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object.
  - *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.



Lifetime and Storage Management

- The storage-management algorithm maintains a single linked list, the *free list*, of heap blocks not currently in use.
  - The *first fit* algorithm selects the first block on the list that is large enough to satisfy a request.
  - The *best fit* algorithm searches the entire list to find the smallest block that is large enough to satisfy the request.
- Common mechanisms for dynamic pool adjustment:
  - The *buddy system*: the standard block sizes are powers of two.
  - The *Fibonacci heap:* the standard block sizes are the Fibonacci numbers.
- *Compacting the heap* moves already-allocated blocks to free large blocks of space.

Lifetime and Storage Management

- *Garbage Collection* (GC):
  - In languages that deallocation of objects is not explicit.
    - Manual deallocation errors are among the most common and costly bugs in real-world programs.
  - Objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable.
    - Costly.
  - Methodologies: reference counting, Mark/Sweep, Copying, Generational GC.
    - Stop-the-world vs. incremental vs. concurrent

Scope Rules

- The binding *scope* is the textual region of the program in which a binding is active.
  - A scope is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted.
- Scoping rule example 1: Declaration before use:
  - Can a name be used before it is declared?
    - Java local vars: NO
    - Java class properties and methods: YES
- The scope of a binding is determined *statically* or *dynamically*.

Scope Rules

- Scoping rule example 2:
  - Two uses of a given name. Do they refer to the same binding?

```
a = 1
    ... def f():
    a = 2
    b = a
```

  - the scoping rules determine the scope

Scope Rules

- The key idea in **static scope rules** is that bindings are defined by the **physical (lexical) structure of the program**.

- Static scoping examples:
  - one big scope (old Basic),
  - scope of a function (variables live through a function execution)
  - block scope,
  - nested subroutines,
  - if a variable is active in one or more scopes, then the closest nested scope rule applies

Scope Rules

- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope), and
  - make references to variables.

  - On subroutine exit:
    - destroy bindings for local variables
    - reactivate bindings for global variables that were deactivated

Scope Rules

- *ELABORATION* = process of creating bindings when entering a scope:
  - storage may be allocated,
  - tasks started,
  - even exceptions propagated as a result of the elaboration of declarations.

Static Scoping

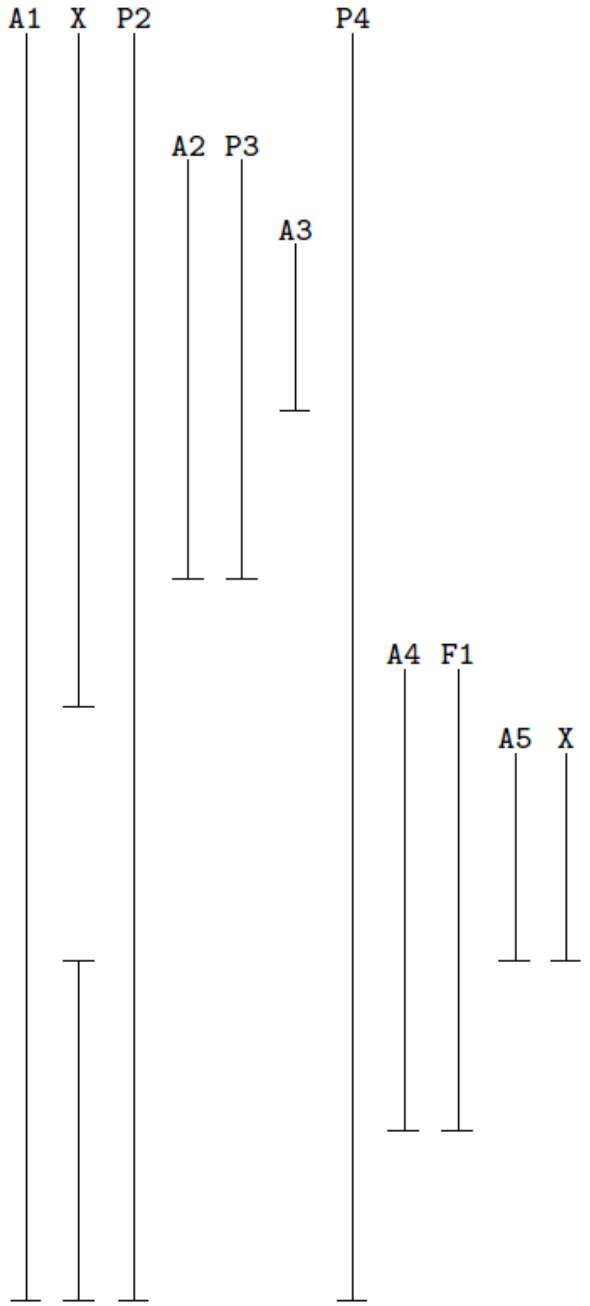
- With *STATIC (LEXICAL) SCOPE RULES* (e.g., C), a scope is defined in terms of the physical (lexical) structure of the program:
  - The determination of scopes can be made by the compiler
  - All bindings for identifiers can be resolved by examining the program
  - Typically, we choose the most recent, active binding made at compile time
    - Most compiled languages, C and Pascal included, employ static scope rules

Static Scoping

- **Nested blocks:**
- The classical example of static scope rules is the most closely nested rule used in block structured languages such as Pascal:
  - An identifier is known in the scope in which it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope (the original identifier is *hidden*)
  - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found
- **Classes** (in abstraction and object-oriented languages) have even more sophisticated (static) scope rules

Pascal:

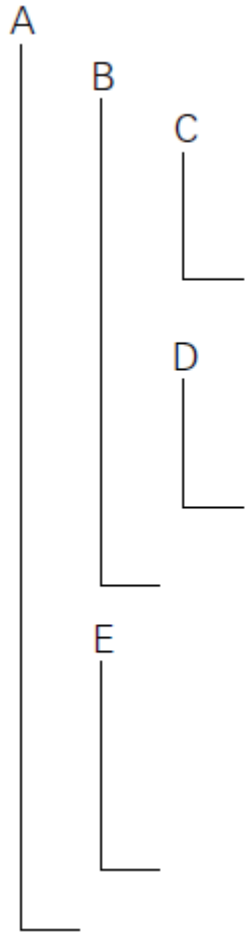
```
procedure P1(A1 : T1);
var X : real;
    ...
    procedure P2(A2 : T2);
        ...
        procedure P3(A3 : T3);
        ...
        begin
            ...      (* body of P3 *)
        end;
        ...
    begin
        ...          (* body of P2 *)
    end;
    ...
    procedure P4(A4 : T4);
        ...
        function F1(A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...      (* body of F1 *)
        end;
        ...
    begin
        ...          (* body of P4 *)
    end;
    ...
begin
    ...              (* body of P1 *)
end
```



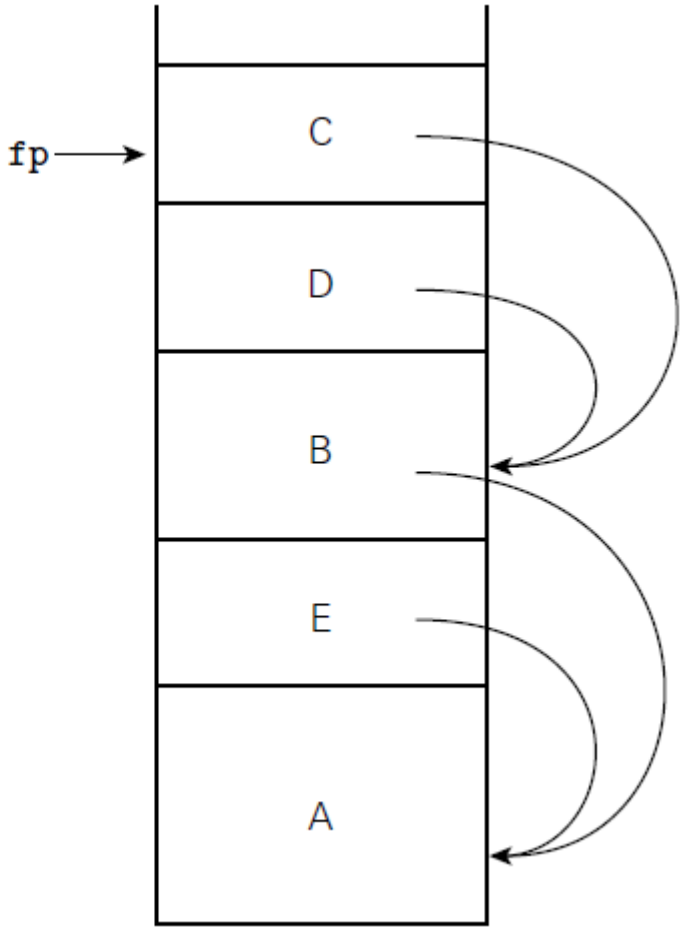
Static Scoping

- The simplest way in which to find the frames of surrounding scopes is to maintain a *static link* in each frame that points to the "parent" frame (i.e., the frame of the most recent invocation of the lexically surrounding subroutine).
  - If a subroutine is declared at the outermost nesting level of the program, then its frame will have a null static link at run time.
  - If a subroutine is nested k levels deep, then its frame's static link, and those of its parent, grandparent, and so on, will form a *static chain* of length k at run time.

Nesting of subroutines:



During run time (with links):



Static Scoping

- **Declaration Order:**
  - Several early languages, including Algol 60 and Lisp, required that all declarations appear at the beginning of their scope.
  - Pascal modified the requirement to say that names must just be declared before they are used.
    - However, it still uses a whole-block scope (also implemented in C#)
      - Example in C#: defining a variable in a block makes every external declaration hidden, so the M=N assignment generates a compiling error:

```
class A {
    const int N = 10;
    void foo() {
        const int M = N;    // uses inner N before it is declared
        const int N = 20;
```

Dynamic Scoping

- **Dynamic scope rules**: bindings depend on the current state of program execution:
  - They cannot always be resolved by examining the program because they are dependent on calling sequences
  - The binding might depend on how a function is called
  - To resolve a reference, we use the most recent, active binding made at run time

Dynamic Scoping of bindings

- Example:

```
var total = 0
def add():
    total += 1
def myfunc():
   var total = 0
   add()
add()
myfunc()
print total
```

- very easy to implement (stack of names), but often hard to make efficient.

Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
  - Such languages do not always have type checking of at compile time because type determination isn't always possible when dynamic scope rules are in effect

Dynamic Scoping

Example typing in SML:

```
- fun concat(x,y) = if x=[] then y
= else hd(x)::concat(tl(x),y);
val concat = fn : ''a list * ''a list -> ''a list
```

```
- concat([1,2],[3,4,5]);
      val it = [1,2,3,4,5] : int list
```

Binding of Reference Environments

- A reference environment = all the bindings active at a given time.
- When we take a reference to a function, we need to decide which reference environment we want to use!!!
  - Deep binding binds the environment at the time the procedure is passed as an argument.
  - Shallow binding binds the environment at the time the procedure is actually called.

Binding of Reference Environments

```
x: integer := 1
y: integer := 2

procedure add
  x := x + y

procedure second(P:procedure)
  x:integer := 2
  P()

procedure first
  y:integer := 3
  second(add)

first()
write_integer(x)
```

- dynamic scoping with deep binding: when add is passed into second the environment of add is x = 1, y = 3 and the x is the global x so it writes 4 into the global x, which is the one picked up by the write_integer.

- shallow binding just traverses up until it finds the nearest variable that corresponds to the name (increments the local x), so the answer would be 1.

Closure

- Reference environment + Subroutine = closure.

```
def foo():
    a = 100
    def bar():
        return a
    a += 1
    return bar
f = foo()
print f()
```

Output: 101

Implementing closures means we may need to keep around foo's frame even after foo quits.

- heap allocation to keep around objects pointed to by foo.

- In languages without closures, we can fake them with objects.

Scope Rules Example: Static vs. Dynamic

```
program scopes (input, output );
var a : integer;
procedure first();
        begin a := 1; end;
procedure second();
        var a : integer;
        begin first(); end;
begin
        a := 2;
        if read_integer() > 0 second();
        else first();
        write(a);
end.
```

- Program output depends on both scope rules and, in the case of dynamic scoping, a value read at run time.
- If static scoping is in effect, this program prints a 1.
- If dynamic scoping is in effect, the output depends on the value read at line 8 at run time: if the input is positive, the program prints a 2; otherwise it prints a 1.
- The assignment to the variable a at line 4 refers either to the global variable declared at line 2 or to the local variable declared at line 6.

Scope Rules Example: Static vs. Dynamic

- Static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable a.
- Dynamic scope rules, on the other hand, require that we choose the most recent, active binding at run time:
  - Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines
  - This is generally considered bad programming practice nowadays (except in functional languages)
    - Alternative mechanisms exist
      - static variables that can be modified by auxiliary routines
      - default and optional parameters

Scope Rules Example: Static vs. Dynamic

- At run time we create a binding for a when we enter the main program.
- Then we create another binding for a when we enter procedure second
  - This is the most recent, active binding when procedure first is executed
  - Thus, we modify the variable local to procedure second, not the global variable
  - However, we write the global variable because the variable a local to procedure second is no longer active

The Meaning of Names within a Scope

- Aliasing:
  - Two names point to the same object.
  - Makes program hard to understand.
  - Makes program slow to compile.

The Meaning of Names within a Scope

- Aliasing example: passing a variable by reference to a subroutine that also accesses that variable directly:

```
double sum, sum_of_squares;
...
void accumulate(double& x) // x is passed by reference
{
   sum += x;
   sum_of_squares += x * x;
}
...
accumulate(sum);
```

sum is passed as an argument to accumulate, so, sum and x will be aliases for one another

The Meaning of Names within a Scope

- Aliasing (cont.):
  - What are aliases good for?
    - space saving - modern data allocation methods are better
    - multiple representations - unions are better
    - linked data structures - legit

The Meaning of Names within a Scope

- Overloading:
  - same name, more than one meaning.
  - some overloading happens in almost all languages
    - integer + vs. real +
    - read and write in Pascal
      - Methods are overloaded based on the number and types of parameters (depends on language).
    - function return in Pascal
  - some languages get into overloading in a big way
    - Java
    - C++

Overloading&Ambiguous Invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }
  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

The Meaning of Names within a Scope

- It's worth distinguishing between some closely related concepts
  - overloaded functions - two different things with the same name; in C++
    - overload norm
    - int norm (int a){return a>0 ? a : -a;)
    - complex norm (complex c) { // …
  - polymorphic functions -- one thing that works in more then one way
    - Overriding in OO programming, and
    - Generic programming: function min (A : array of integer); …
      - generic functions - a syntactic template that can be instantiated in more than one way at compile time
        - via macro processors in C++ (built-in in C++)

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}
class GraduateStudent
        extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type – can be invoked with any object

Polymorphism: an object of a subtype can be used wherever its supertype value is required

***Dynamic binding*:** the Java Virtual Machine determines dynamically at runtime which implementation is used by the method

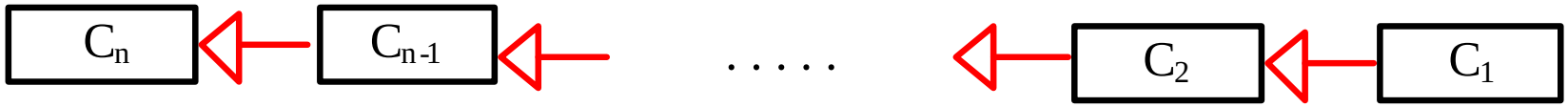
When the method m(Object x) is executed, the argument x's toString method is invoked.

Method Matching vs. Binding

- The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time
- The Java Virtual Machine dynamically binds the implementation of the method at runtime

Dynamic Binding

- Suppose an object o is an instance of classes C_1, C_2, …, C_{n-1}, and C_n
  - C_1 is a subclass of C_2, C_2 is a subclass of C_3, …, and C_{n-1} is a subclass of C_n
  - C_n is the most general class, and C_1 is the most specific class
  - If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, …, C_{n-1} and C_n, in this order, until it is found, the search stops and the first-found implementation is invoked



Since o is an instance of C_1, o is also an instance of C_2, C_3, …, C_{n-1}, and C_n

Dynamic Binding

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}
class GraduateStudent extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}
class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Output:

Student

Student

Person

java.lang.Object@12345678

Binding of Referencing Environments

- Accessing variables with dynamic scope:
  - (1) keep a stack (association list) of all active variables
    - When you need to find a variable, hunt down from top of stack
    - This is equivalent to searching the activation records on the dynamic chain

Binding of Referencing Environments

- Accessing variables with dynamic scope:
  - (2) keep a central table with one slot for every variable name
    - If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
    - Otherwise, you'll need a hash function or something to do lookup
    - Every subroutine changes the table entries for its locals at entry and exit.

Binding of Referencing Environments

- (1) gives you slow access but fast calls
- (2) gives you slow calls but fast access
- In effect, variable lookup in a dynamically-scoped language corresponds to symbol table lookup in a statically-scoped language
- Because static scope rules tend to be more complicated, however, the data structure and lookup algorithm also have to be more complicated

Separate Compilation

- Separately-compiled files in C provide a sort of poor person's modules:
  - Rules for how variables work with separate compilation are messy
  - Language has been jerry-rigged to match the behavior of the linker
  - Static on a function or variable outside a function means it is usable only in the current source file
    - This static is a different notion from the static variables inside a function
  - Extern on a variable or function means that it is declared in another source file
  - Functions headers without bodies are extern by default
  - Extern declarations are interpreted as forward declarations if a later declaration overrides them

Separate Compilation

- Separately-compiled files in C (continued)
  - Variables or functions (with bodies) that don't say static or extern are either global or common (a Fortran term)
    - Functions and variables that are given initial values are global
    - Variables that are not given initial values are common
  - Matching common declarations in different files refer to the same variable
    - They also refer to the same variable as a matching global declaration

Modules (AKA packages)

- Break program up into parts, which need to be explicitly imported.

- We only need to agree on the meaning of names when our code interacts.

Macros

- Macros are a way of assigning a name to some syntax.
  - C: Textual substitution.

    #define MAX(x, y) (x > y ? x : y)
    - benefit: shorter code, no stack, can choose not to execute some of the code
  - Problems with macros:
    - multiple side effects: MAX(a++, b++)
    - binding: MAX(1 | a, b) - we have to change our macro to

      #define MAX(x, y) ((x) > (y) ? (x) : (y))
    - scope capture: temp var used inside macro has same name as real var
  - Scheme and Common Lisp hygenic macros:
    - quote variables,
    - rename variables,
    - potentially have multiple side effects.

Example Problem

```
var total = 0
def a():
   total += 1
def b(f):
   var total = 0
   f()
   print "B", total
def c():
   var total = 0
   b(a)
   print "C", total
c()
print "T", total
```

	static	dynamic shallow	dynamic deep
B	0	1	0
C	0	0	1
T	1	0	0

Scheme

- **Local Bindings:** creates a new location for each id, and places the values into the locations. It then evaluates the bodys, in which the ids are bound.

```
> (let ((x 5)) x)
5
> (let ((x 5))
    (let ((x 2)
          (y x))
      (list y x)))
'(5 2)
> (let fac ([n 10])
    (if (zero? n)
        1
        (* n (fac (sub1 n)))))
3628800
```

Scheme

- `let*` evaluates the value-expressions one by one, creating a location for each id as soon as the value is available. The ids are bound in the remaining value-expressions as well as the bodys, and the ids need not be distinct; later bindings shadow earlier bindings.

```
> (let* ((x 1)
         (y (+ x 1)))
    (list y x))
'(2 1)
```

Scheme

- `letrec:` the locations for all ids are created first and filled with #<undefined>, all ids are bound in all value-expressions as well as the bodys, and each id is set immediately after the corresponding value-expression is evaluated.
  - allow recursion in the definition of the variables bound locally

```
> (letrec ([is-even? (lambda (n)
                       (or (zero? n)
                           (is-odd? (sub1 n))))]
         [is-odd? (lambda (n)
                    (and (not (zero? n))
                         (is-even? (sub1 n))))])
    (is-odd? 11))
#t
```