Names, Scopes, and Bindings

CSE 307 – Principles of Programming Languages

Stony Brook University

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

Names, Scopes, and Bindings

- *Names* are identifiers (mnemonic character strings used to represent something in the program - instead of low-level concepts like addresses):
  - A name also represents 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
  - E.g. function name is bound to its definition.

Names, Scopes, and Bindings

- The textual region of the program in which a binding is active is its *scope*.
  - E.g. Java blocks
- The complete set of bindings in effect at a given point in a program is known as the current *referencing environment*.

Bindings

- ***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 such as int, other aspects of language semantics
    - language implementation time: precision (number of bits) of the fundamental types such as int, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow

Bindings

- 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), resolves intermodule references
- load time: choice of physical addresses for virtual addresses such as names

Bindings

- ***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
- The terms *STATIC* and *DYNAMIC* are generally used to refer to things bound before run time and at run time.

Bindings

- In general, later binding times are associated
with greater flexibility (e.g. assigning a keyword during language design provides less flexibility at a later time)
- Early binding times are associated with greater efficiency
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times

Lifetime and Storage Management

- Bindings key events:
  - creation of objects
  - creation of bindings
  - references to variables (which use 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

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:
  - 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.*
  - *Temporaries* (intermediate values produced in complex calculations)
  - *Other bookkeeping information* such as 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* (or *activation record*) for:
    - parameters
    - local variables
    - return address

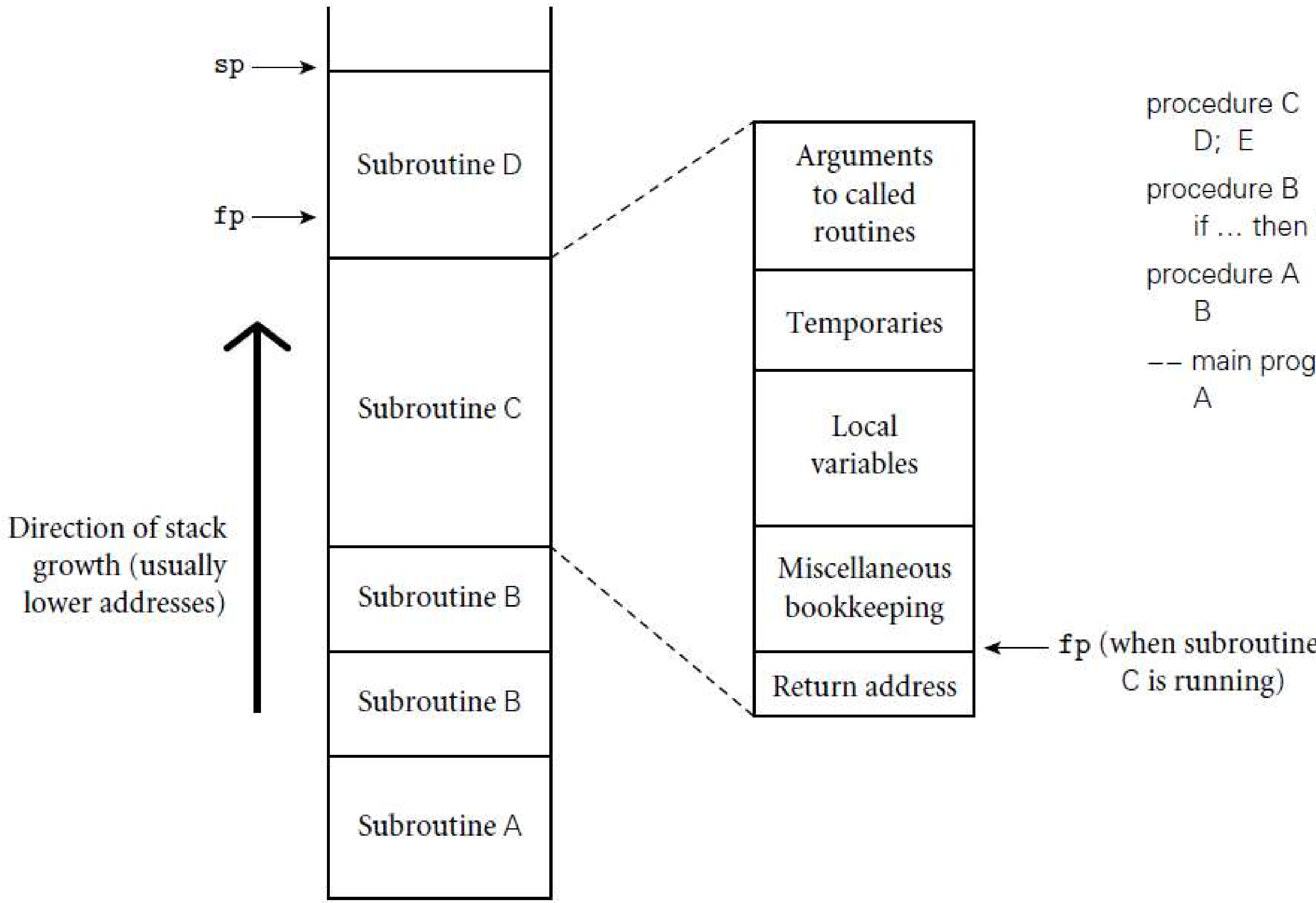
Lifetime and Storage Management

- Maintenance of the stack is the responsibility of the subroutine *calling sequence* (the code executed by the caller immediately before and after the call), which includes: 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 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)

Lifetime and Storage Management




```
procedure C
    D;  E
procedure B
    if … then B else C
procedure A
    B
-- main program
    A
```

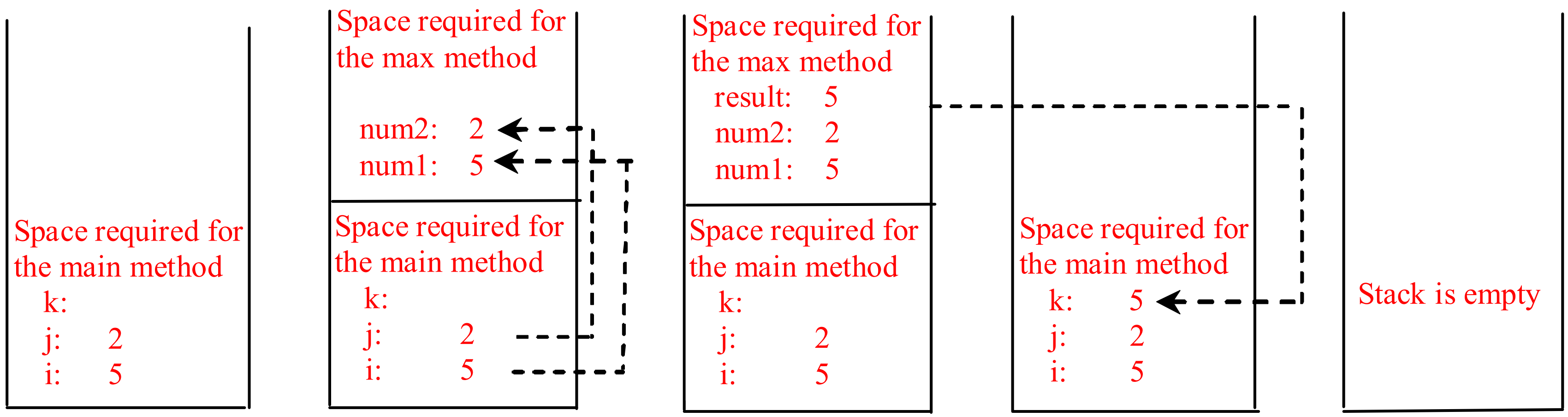
Calling Methods Example in Java

pass the value of i

pass the value of j

```java
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);
}
```

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

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

  return result;
}
```



(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.

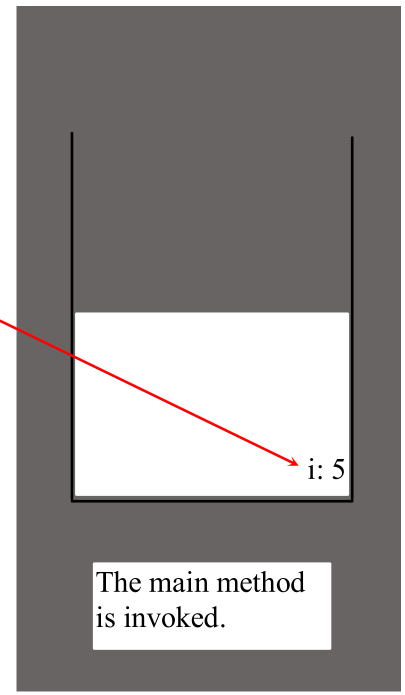
(e) The main method is finished.

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;
}
```

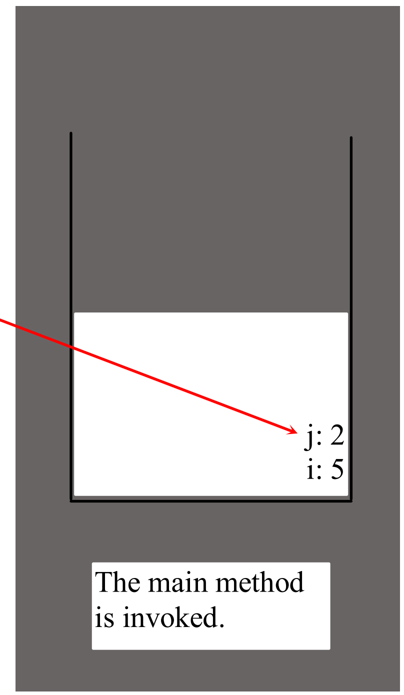


Trace Call Stack

j 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;
}
```

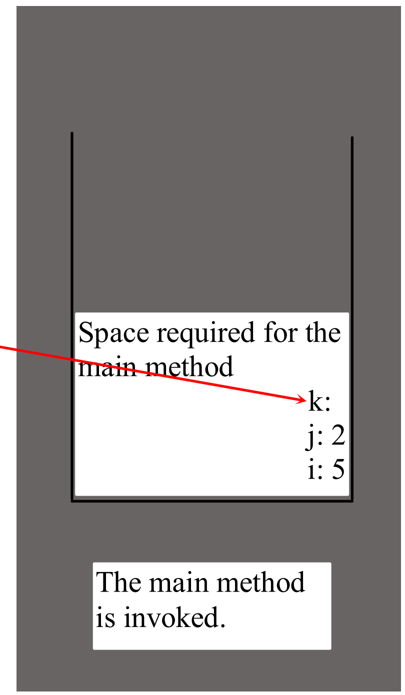


Trace Call Stack

Declare k

```java
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);
}
```

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

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

  return result;
}
```

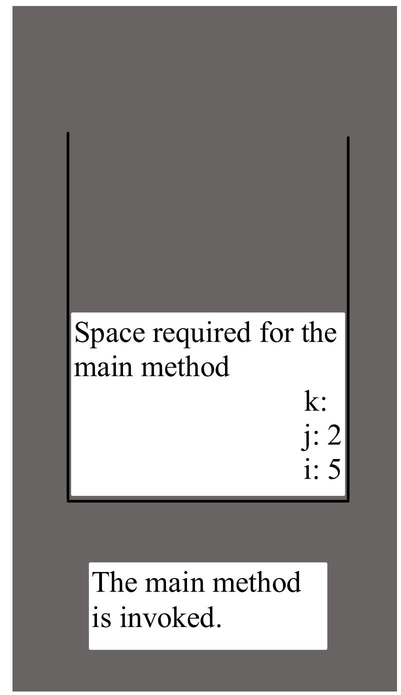


Trace Call Stack

Invoke max(i, j)

```
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;
}
```

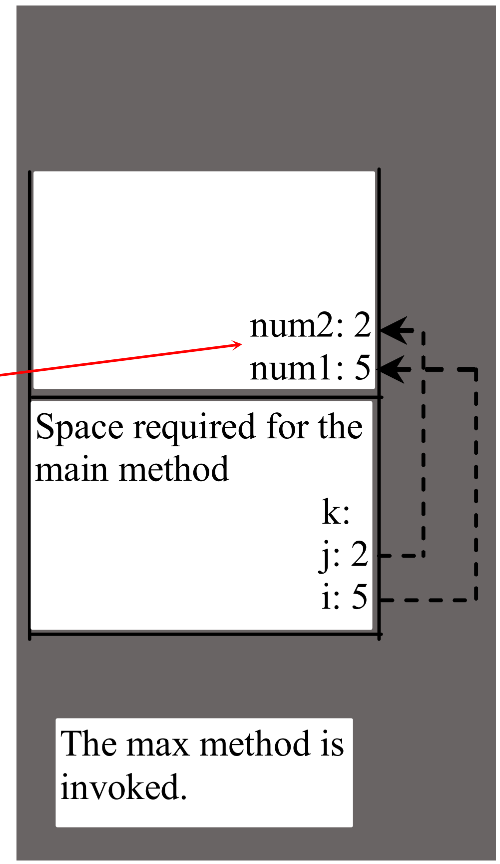


Trace Call Stack

pass the values of i and j to num1 and num2

```
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;
}
```

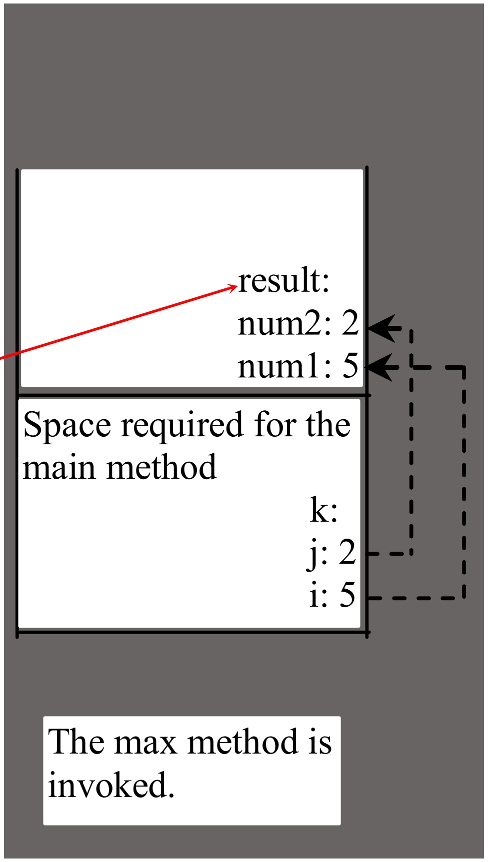


Trace Call Stack

pass the values of i and j to num1 and num2

```
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;
}
```

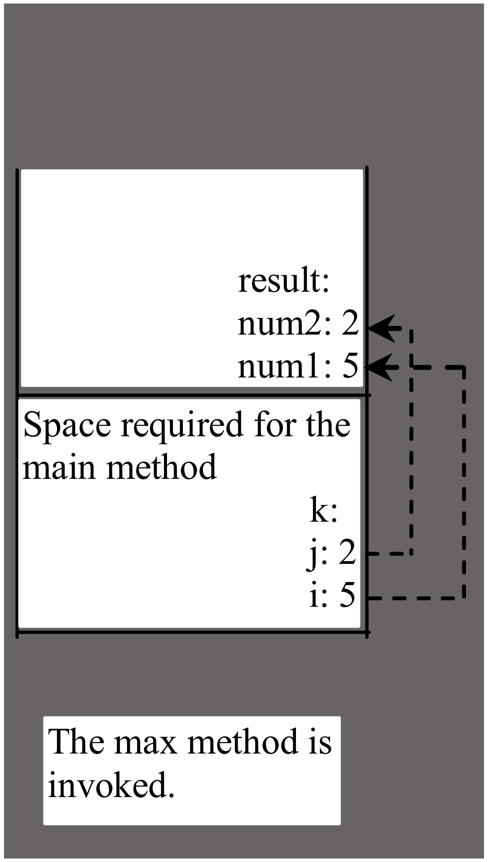


Trace Call Stack

```
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;
}
```

(num1 > num2) is true

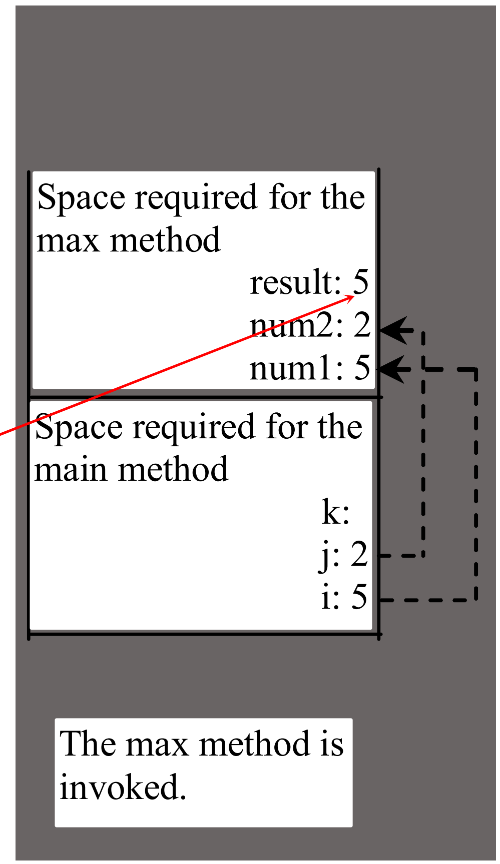


Trace Call Stack

Assign num1 to result

```
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);
}
```

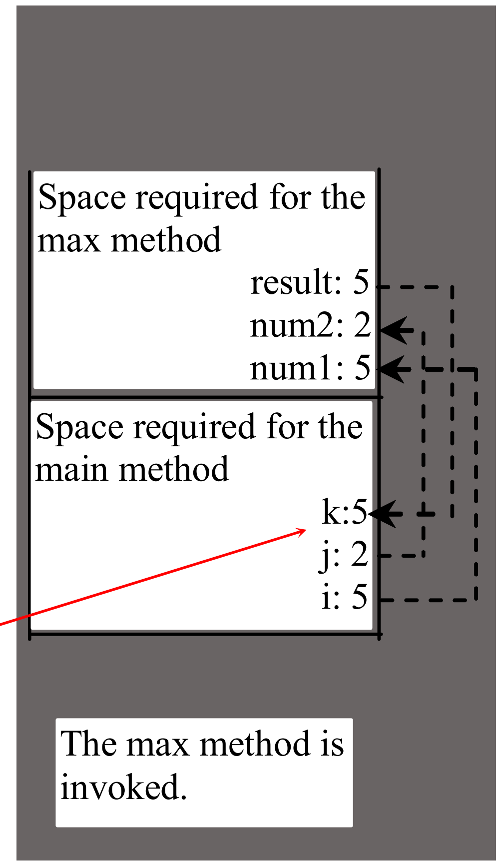
```
public static int max(int num1, int num2)
  int result;

  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;
}
```

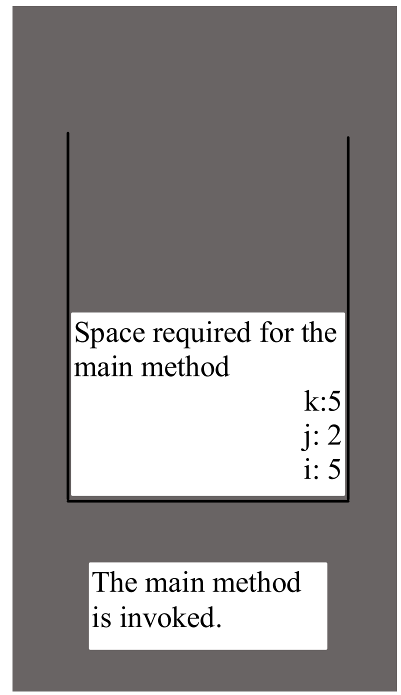


Trace Call Stack

Execute print statement

```
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;
}
```

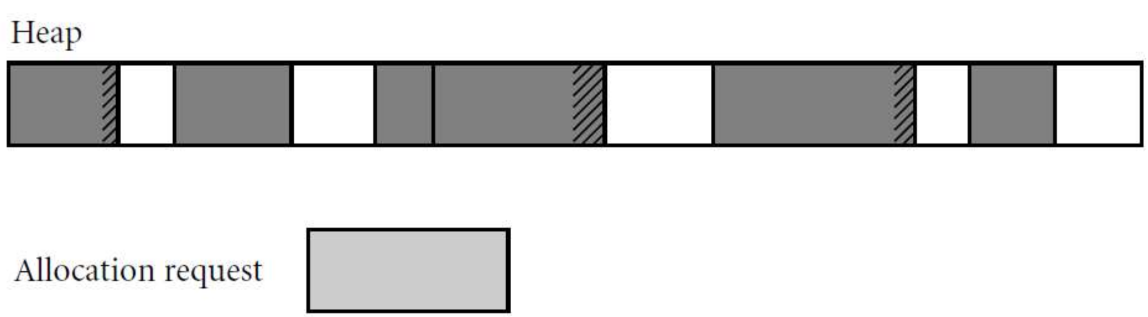


Lifetime and Storage Management

- Heap-Based Allocation
  - ***Heap*** is for dynamic allocation
  - 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

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 – 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

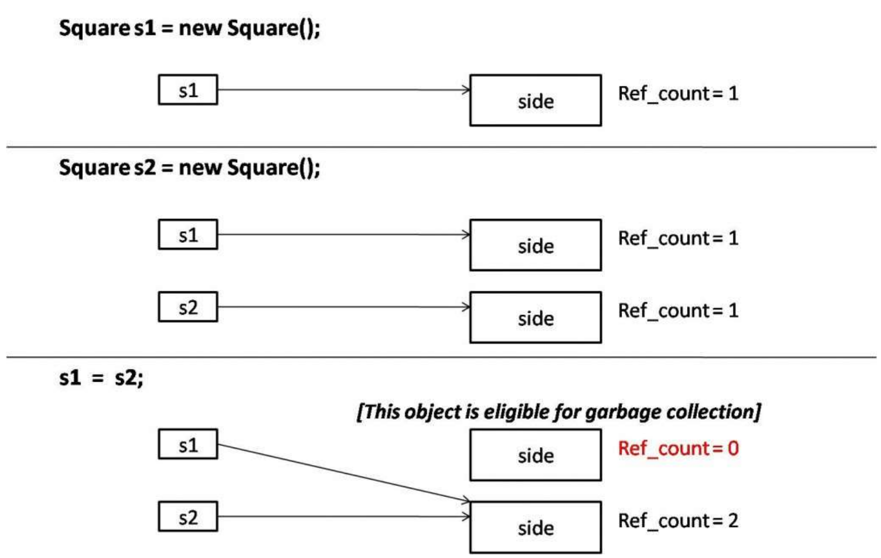


Lifetime and Storage Management

- 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



Mark & Sweep: https://i.stack.imgur.com/W5pX6.gif

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.
- The scope of a binding is determined *statically* or *dynamically*
- Scoping rule example 1: Declaration before use
  - Can a name be used before it is declared?
    - Java local vars: NO
    - Java class static variables and methods: YES

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

Python global

```
# Here, we're creating a variable 'x', in the __main__ scope.
x = 'None!'
def func_A():
  # The below declaration lets the function know that we mean the global 'x' when we refer to that variable, not any local one
  global x
  x = 'A'
  return x
def func_B():
  # Here, we are somewhat mislead.  We're actually involving two different
  #  variables named 'x'.  One is local to func_B, the other is global.
  # By calling func_A(), we do two things: we're reassigning the value
  #  of the GLOBAL x as part of func_A, and then taking that same value
  #  since it's returned by func_A, and assigning it to a LOCAL variable
  #  named 'x'.
  x = func_A() # look at this as: x_local = func_A()
  # Here, we're assigning the value of 'B' to the LOCAL x.
  x = 'B' # look at this as: x_local = 'B'
  return x # look at this as: return x_local
```

Scope Rules

- In ***static scope rules,*** bindings are defined by the **physical (lexical) structure of the program**
- ***Static scoping*** (also called *lexical scoping*) rule examples:
  - one big scope – one big segment of memory (old Basic),
  - scope of a function (variables live through a function execution - Java)
  - block scope (a local var. is available in the block in which is defined)
  - nested subroutines (have access to the variables defined in the parent)
  - if a variable is active in one or more scopes, then the closest nested scope rule applies
- Lexical/static scoping was used for ALGOL and has been picked up in most other languages since then: like Pascal, C, Java

Scope Rules

- *ELABORATION* = process of creating bindings when entering a subroutine scope
- 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
- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated

Scope Holes

In static scoping, "inner" declarations hide, or **shadow** outer declarations of the same identifier, causing a **hole** in the scope of the outer identifier's binding. But when we're in the scope of the inner binding, can we still *see* the outer binding? Sometimes, we can:

In C++:

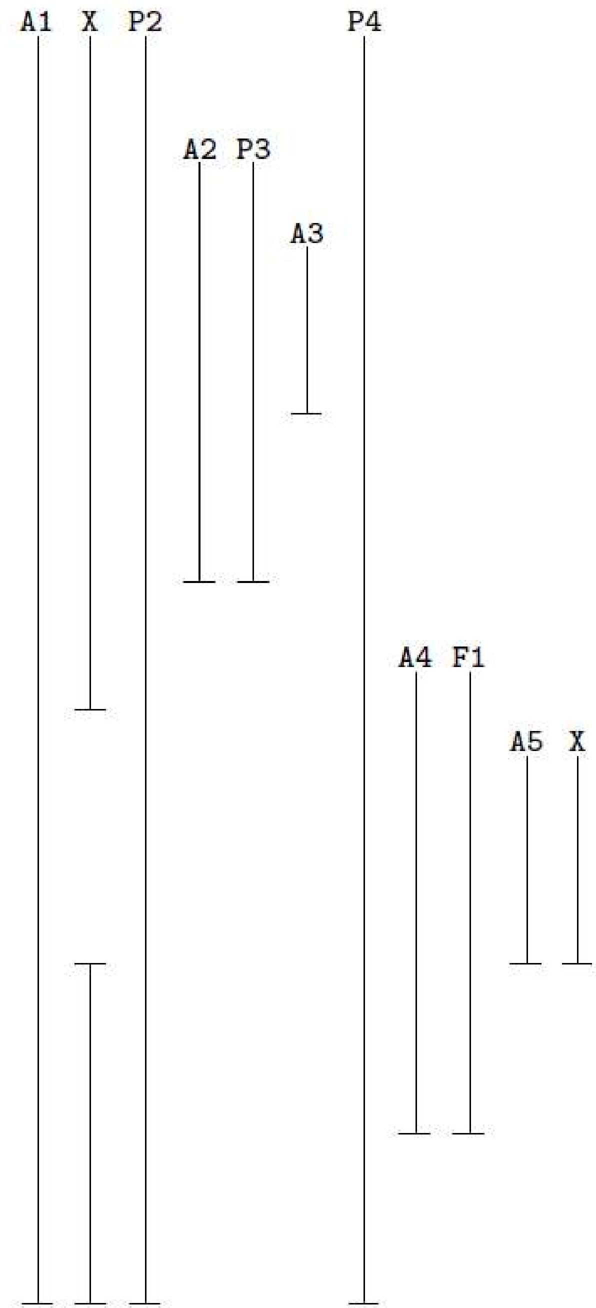
```
int x = 1;
namespace N {
  int x = 2;
  class C {
    int x = 3;
    void f() {int x = 4; cout << ::x << N::x << this->x << x << '\n';
  }
}
```

Static Scoping

- With *STATIC (LEXICAL) SCOPE RULES* (e.g., C, Java and python), a scope is defined in terms of the physical (lexical) structure of the program:
  - 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, Pascal, Java and python included, employ static scope rules
- **Nested blocks:**
  - The classical example of static scope rules is the *most closely nested rule* used in block structured languages (started with Pascal):
- **Classes** (in 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
```



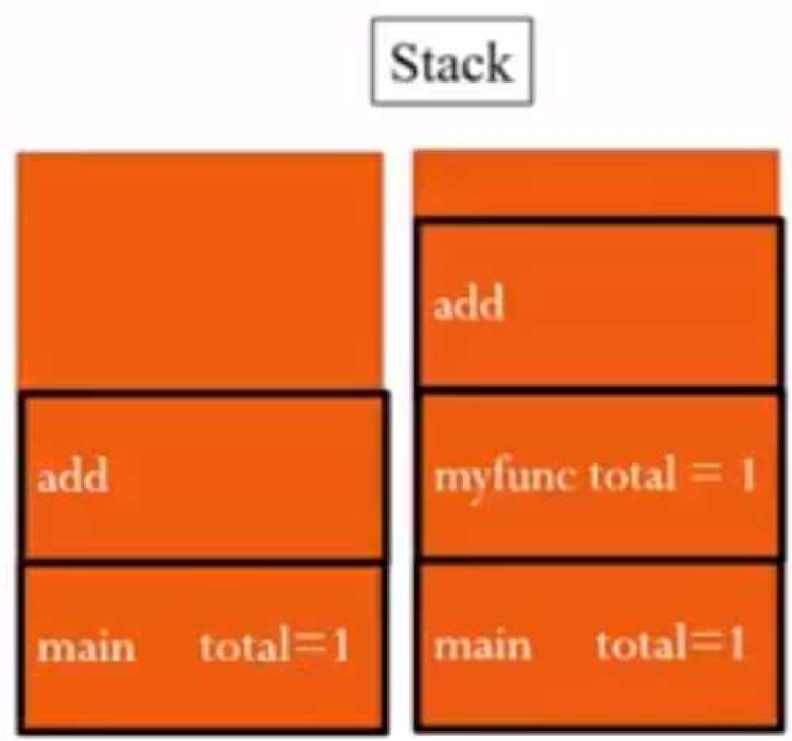
Dynamic Scoping

- ***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
```



prints 1 (add dynamically binds to total in myfunc)

Dynamic Scoping of bindings

- Dynamic scope rules are usually encountered in interpreted languages
  - **Lisp, Perl, Ruby**
    - 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
- A common use of dynamic scope rules is to provide implicit parameters to subroutines

The Meaning of Names within a Scope

- Overloading:
  - same name, more than one meaning
  - some overloading happens in almost all languages
    - integer + vs. real + vs. String concatenation
    - read and write in Pascal are overloaded based on the number and types of parameters
    - 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;
  }
}
```

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

reference to max is ambiguous
both method max(int,double) in AmbiguousOverloading and method max(double,int) in AmbiguousOverloading match
----
(Alt-Enter shows hints)

```java
    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
    - 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 Comparable)
```

      - generic functions - a syntactic template that can be instantiated in more than one way at compile or even run time
        - via macro processors 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.

Output:

Student

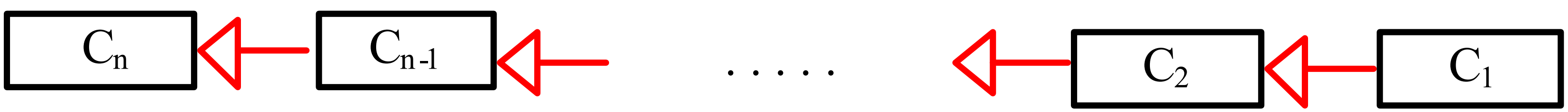
Student

Person

java.lang.Object@15db9742

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

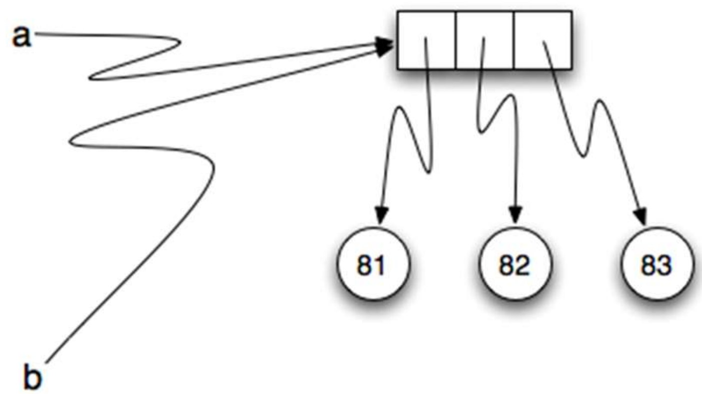
Method Matching vs. Binding

- 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

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

```
a = [81, 82, 83]
b = [81, 82, 83]

print(a == b)
print(a is b)

b = a
print(a == b)

b[0] = 5
print(a)
```



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++)`
    - scope capture: temporary `var` used inside macro has same name as a real `var` – for example: `t` exists outside

```
#define SWAP(a,b) {t = (a); (a) = (b); (b) = t;}
```

      - Scheme and Common Lisp hygienic macros rename variables

Multiple Side Effects

In C, preprocessor macros can have unexpected effects because their arguments can be evaluated multiple times. For example, the following code:

```
#define MAX(a,b) ((a)>(b) ? (a) : (b))
int i = 5, j = MAX(i++, 0);
```

becomes:

```
int i = 5, j = ((i++)>(0) ? (i++) : (0));
```

and the variable `i` will have the value 7—not 6 as expected—because the macro's arguments are repeated in the macro definition.

The Hygiene Problem

In programming languages that have non-hygienic macro systems, it is possible for existing variable bindings to be hidden from a macro by variable bindings that are created during its expansion. In C, this problem can be illustrated by the following fragment:

```
#define INCI(i) do { int a=0; ++i; } while(0)
int main(void)
{
    int a = 4, b = 8;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

Running the above through the C preprocessor produces:

```
int main(void)
{
    int a = 4, b = 8;
    do { int a=0; ++a; } while(0);
    do { int a=0; ++b; } while(0);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

The variable a declared in the top scope is shadowed by the a variable in the macro, which introduces a new scope. As a result, it is never altered by the execution of the program, as the output of the compiled program shows:

```
a is now 4, b is now 9
```

The Hygiene Problem

The simplest solution is to give the macros variables names that do not conflict with any variable in the current program:

```
#define INCI(i) do { int INCIa=0; ++i; } while(0)
int main(void)
{
    int a = 4, b = 8;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

Until a variable named `INCIa` is created, this solution produces the correct output:

```
a is now 5, b is now 9
```