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

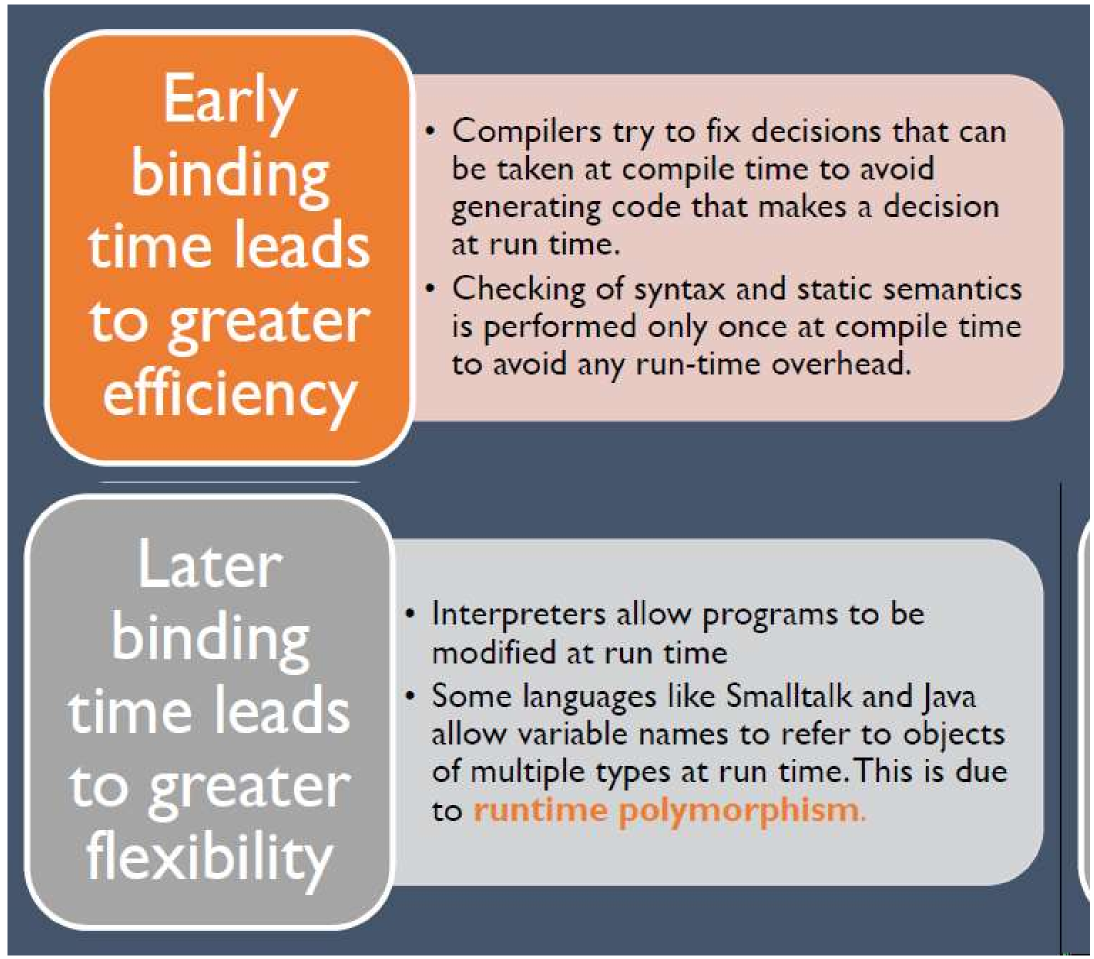
Examples

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

Early Binding Example

```
class SimpleCalculator
{
    int add(int a, int b)
    {
        return a+b;
    }
    int  add(int a, int b, int c)
    {
        return a+b+c;
    }
}
public class Demo
{
   public static void main(String args[])
   {
                        SimpleCalculator obj = new SimpleCalculator();
       System.out.println(obj.add(10, 20));
       System.out.println(obj.add(10, 20, 30));
   }
}

//Output: 30, 60
```

Late Binding Example

```
class X
{
   public void methodA() // Base class method
   {
      System.out.println ("hello, I'm methodA of class X");
   }
}
class Y extends X
{
   public void methodA() // Derived Class method
   {
      System.out.println ("hello, I'm methodA of class Y");
   }
}
public class Z
{
  public static void main (String args []) {
     X obj1 = new X(); // Reference and object X
     X obj2 = new Y(); // X reference but Y object
     obj1.methodA();
     obj2.methodA();
  }
}
//hello, I'm methodA of class X
//hello, I'm methodA of class Y
```

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

Object lifetime is the time between an object's creation and its destruction.

- Here, we are speaking of objects as general 'things', and not strictly in the OOP sense.

Key events during this time:

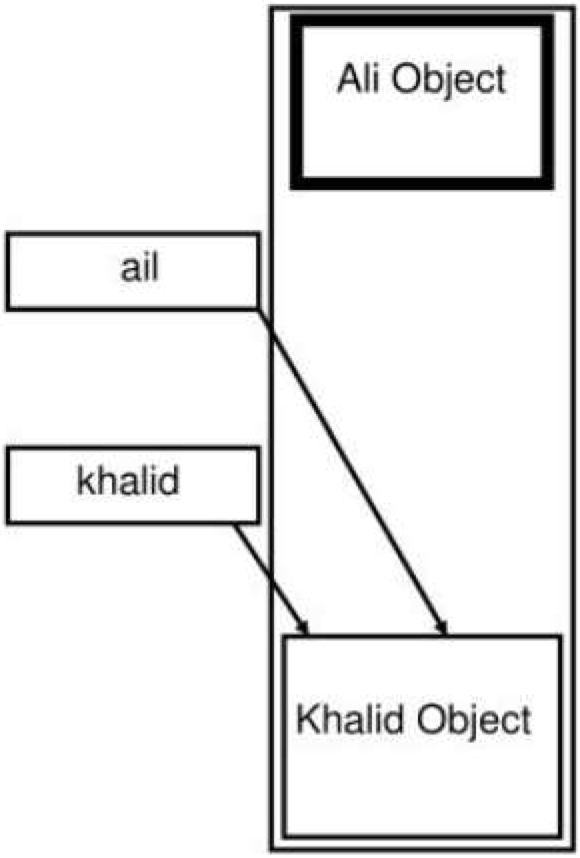
- Creation of the object
- Creation of its binding
- References to variables, subroutines, types, etc. (ALL of which use bindings)
- De/Reactivation of its binding
- Destruction of its binding
- Destruction of the object

Garbage vs. Dangling Reference

Garbage: unreferenced objects

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

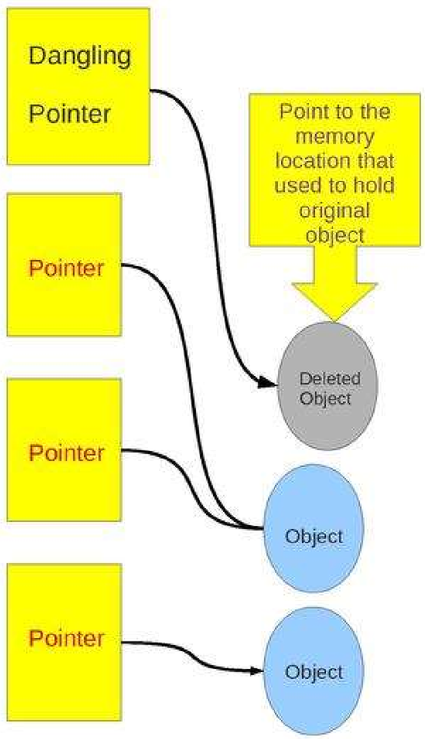
Now ali Object becomes a garbage,
It is unreferenced Object



Garbage vs. Dangling Reference

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

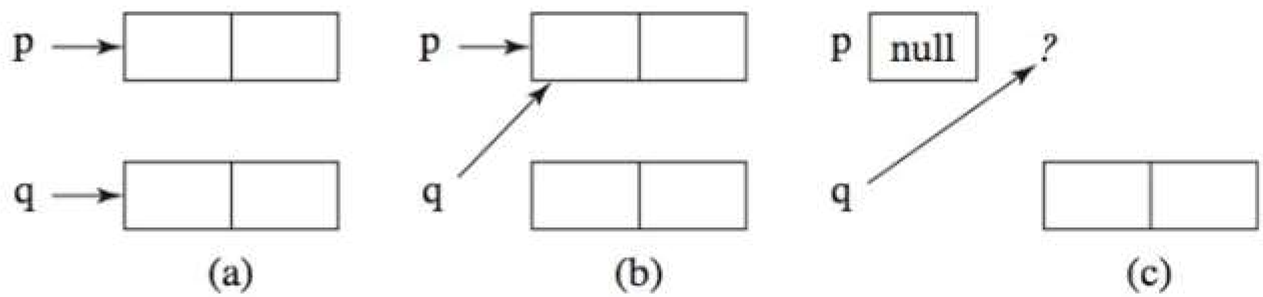
```
int * p = new int;
delete p;

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



Example of Dangling Reference

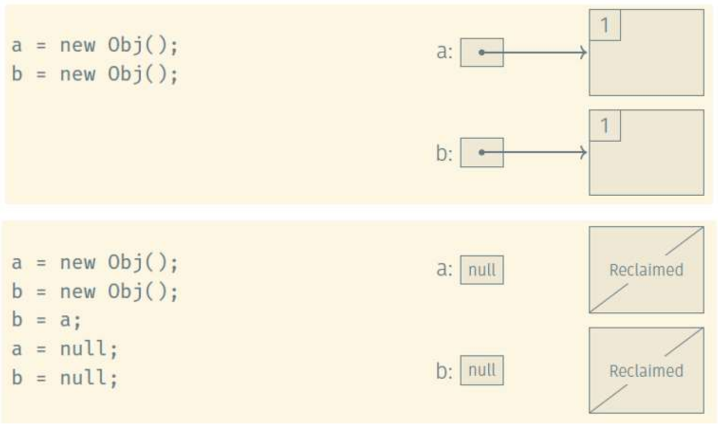
```
class node {
    int value;
    node next;
}
node p, q;
```

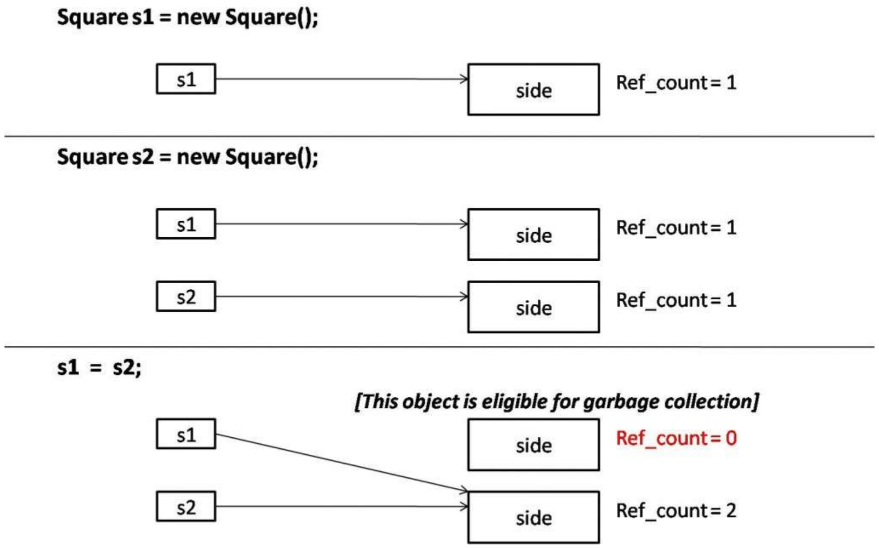
```
p = new node();
q = new node();
q = p;
delete p;
```



Garbage Collection

- 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.
- Reference Counting Algorithm





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

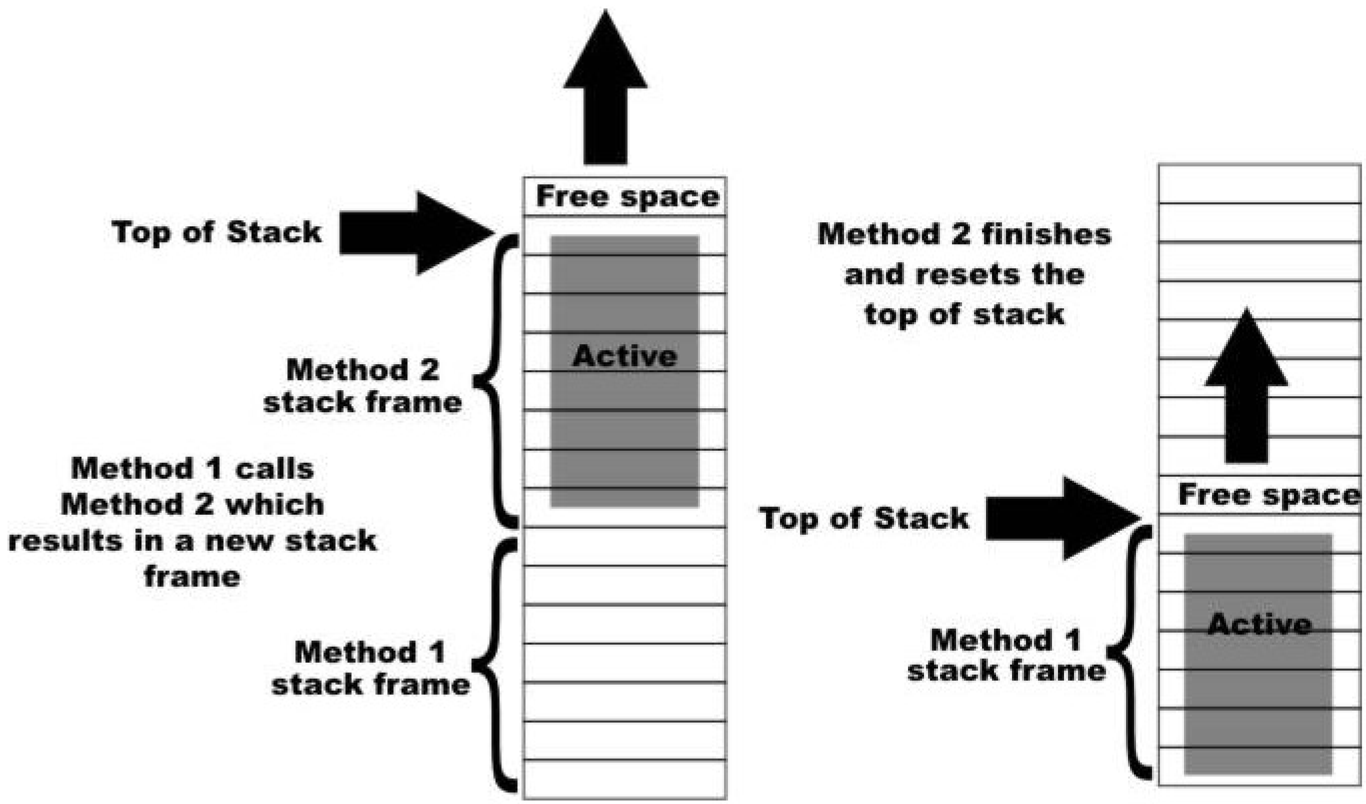
Static Storage Allocation

- 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)

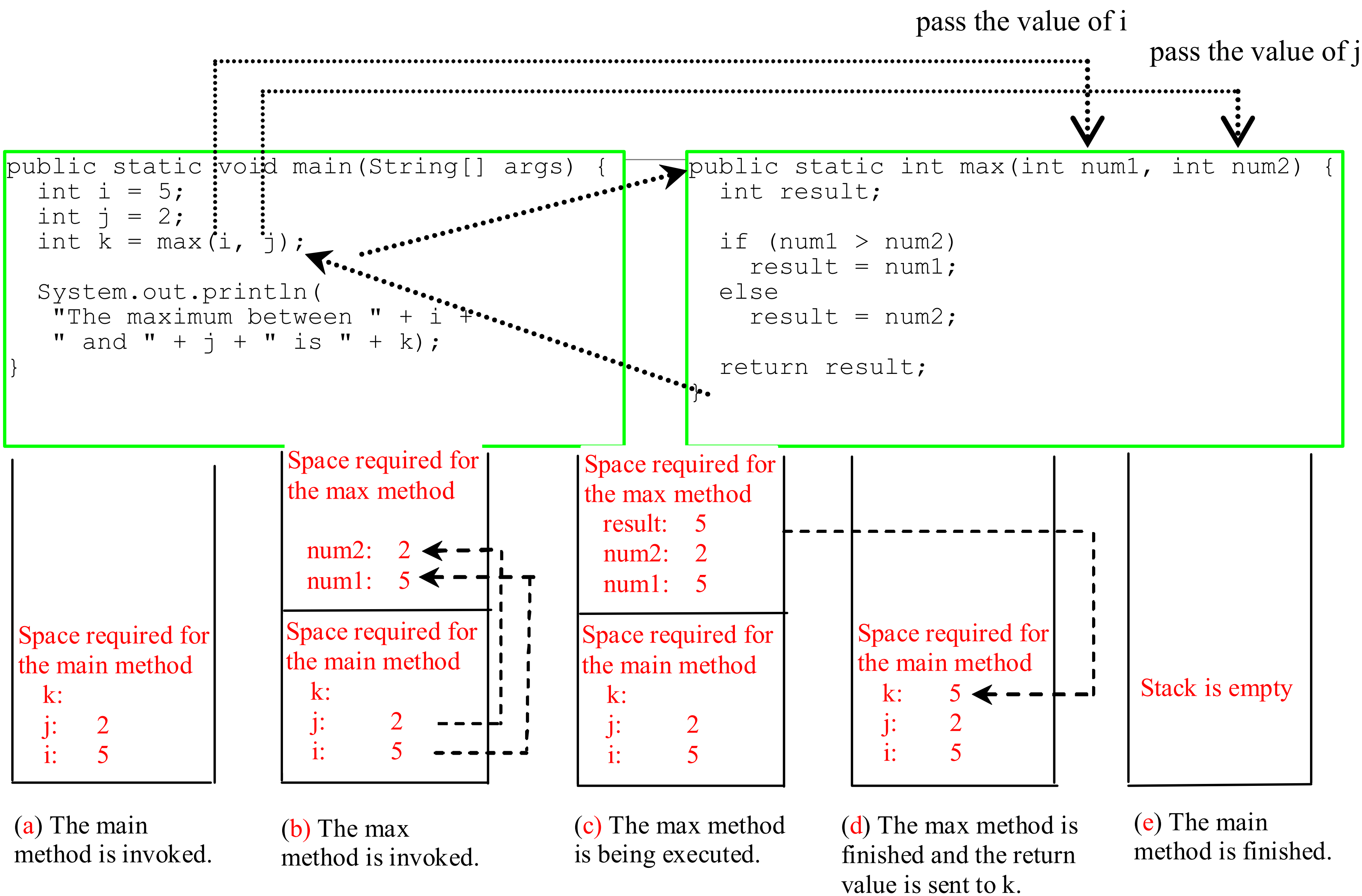
Stack Based Storage Management

- 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

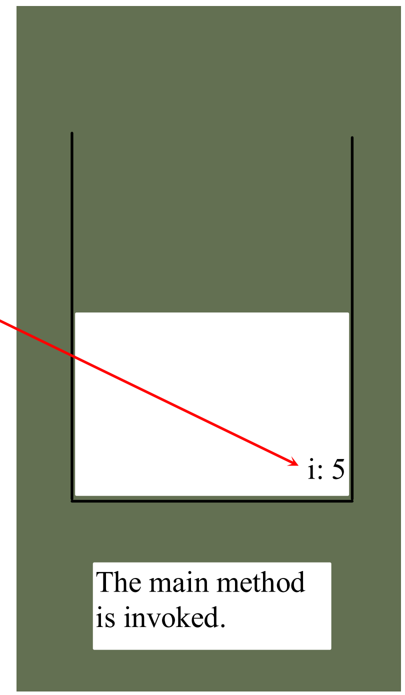


Trace Call Stack

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;
}
```

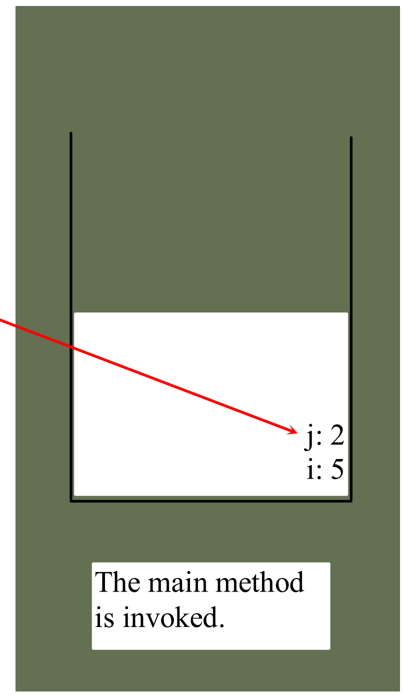


Trace Call Stack

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;
}
```

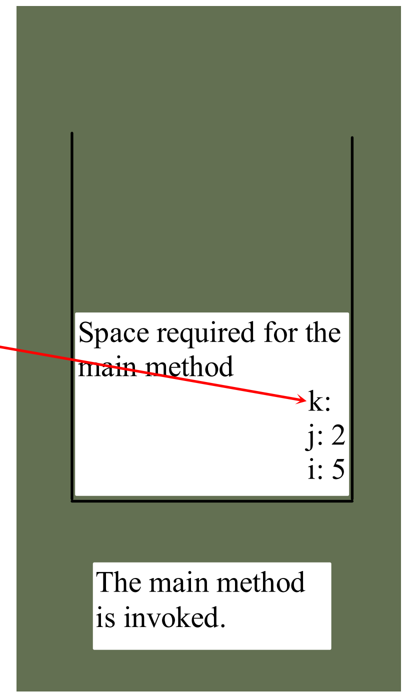


Trace Call Stack

Declare k

```
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;
}
```

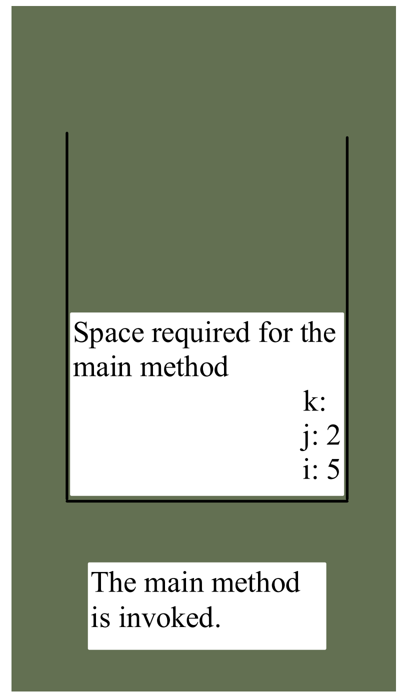


Trace Call Stack

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;
}
```

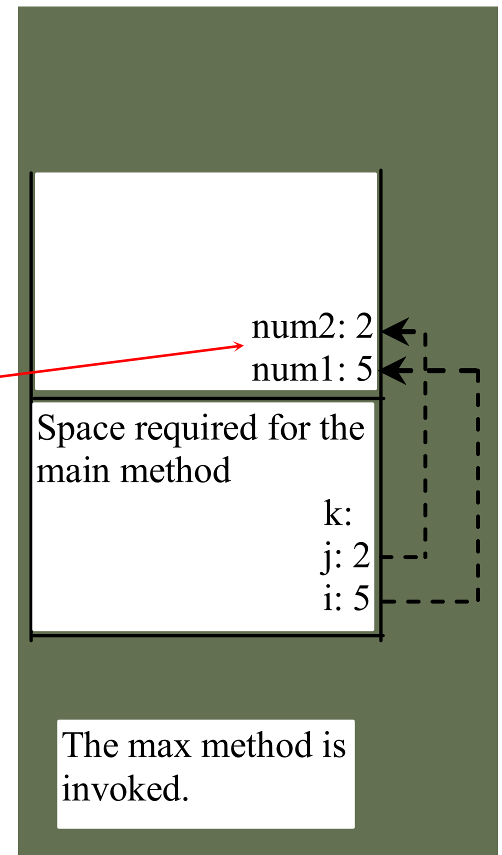


Trace Call Stack

pass the values of i and j to num1 and num2

```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;
}
```

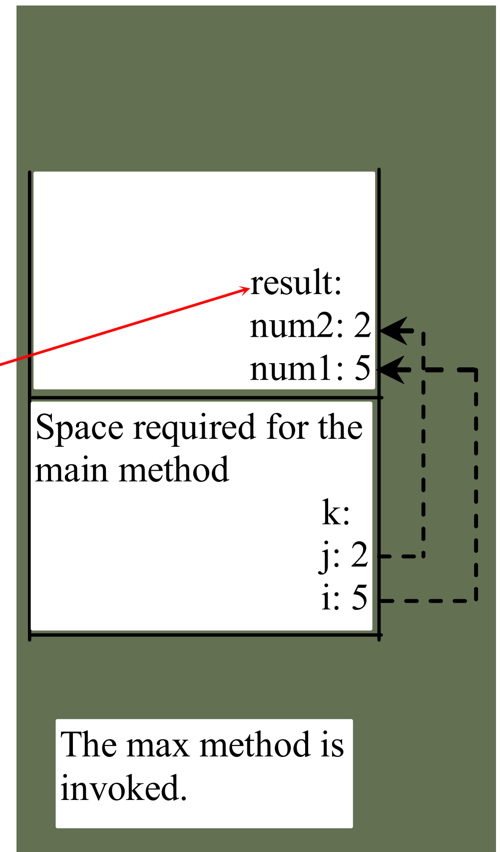


Trace Call Stack

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;
}
```

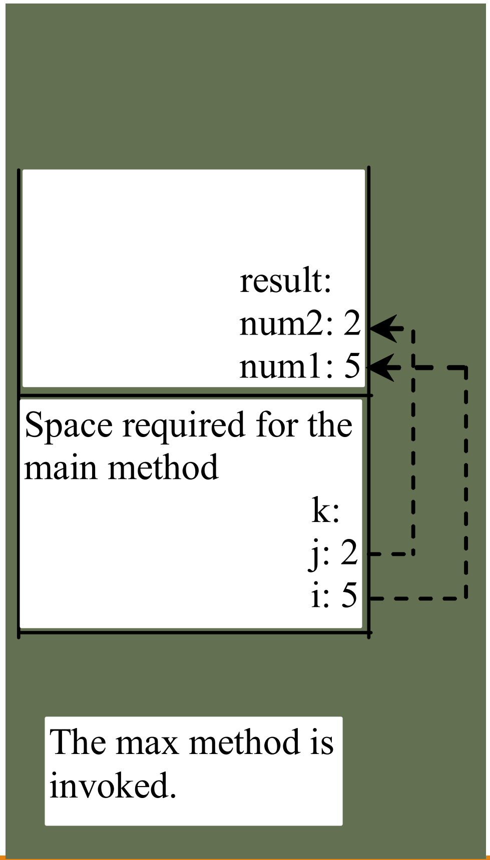


Trace Call Stack

(num1 > num2) is true

```
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;
}
```

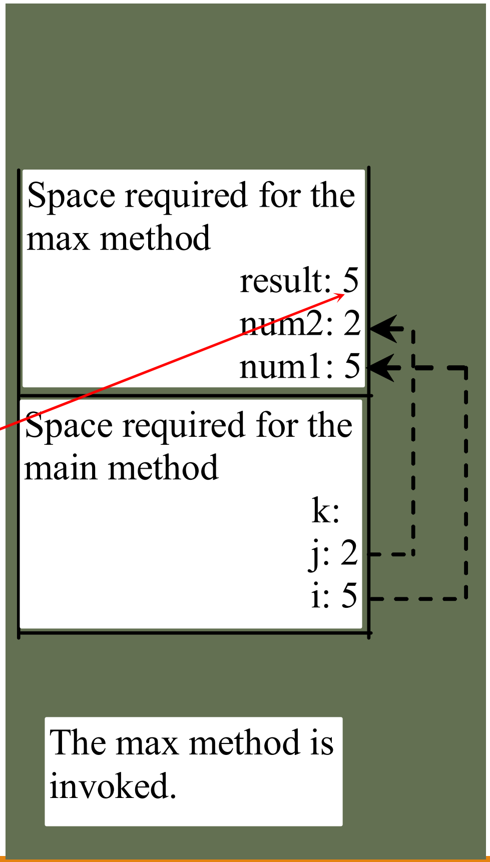


Trace Call Stack

```
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;
}
```

Assign num1 to result

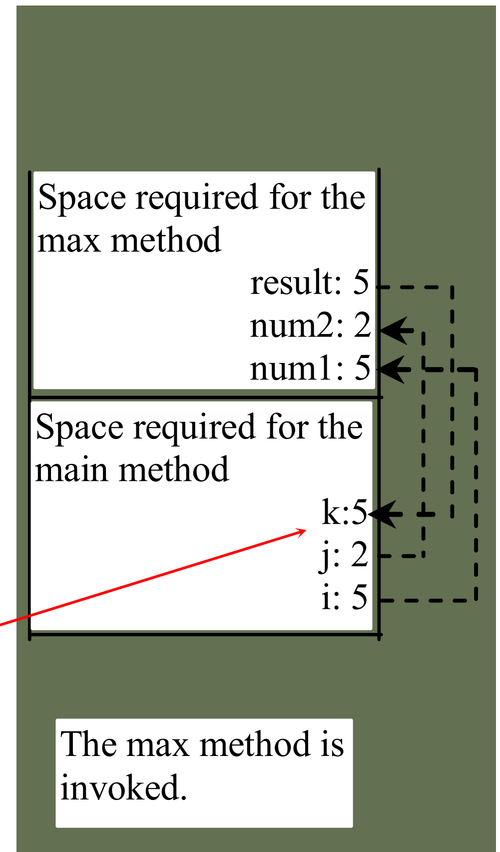


Trace Call Stack

Return result and assign it to k

```
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;
}
```

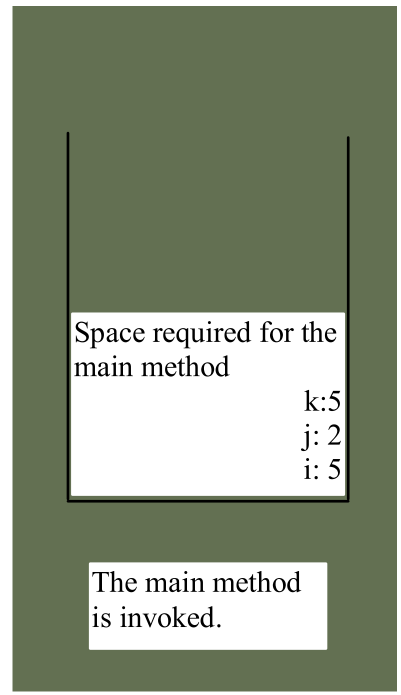


Trace Call Stack

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;
}
```



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

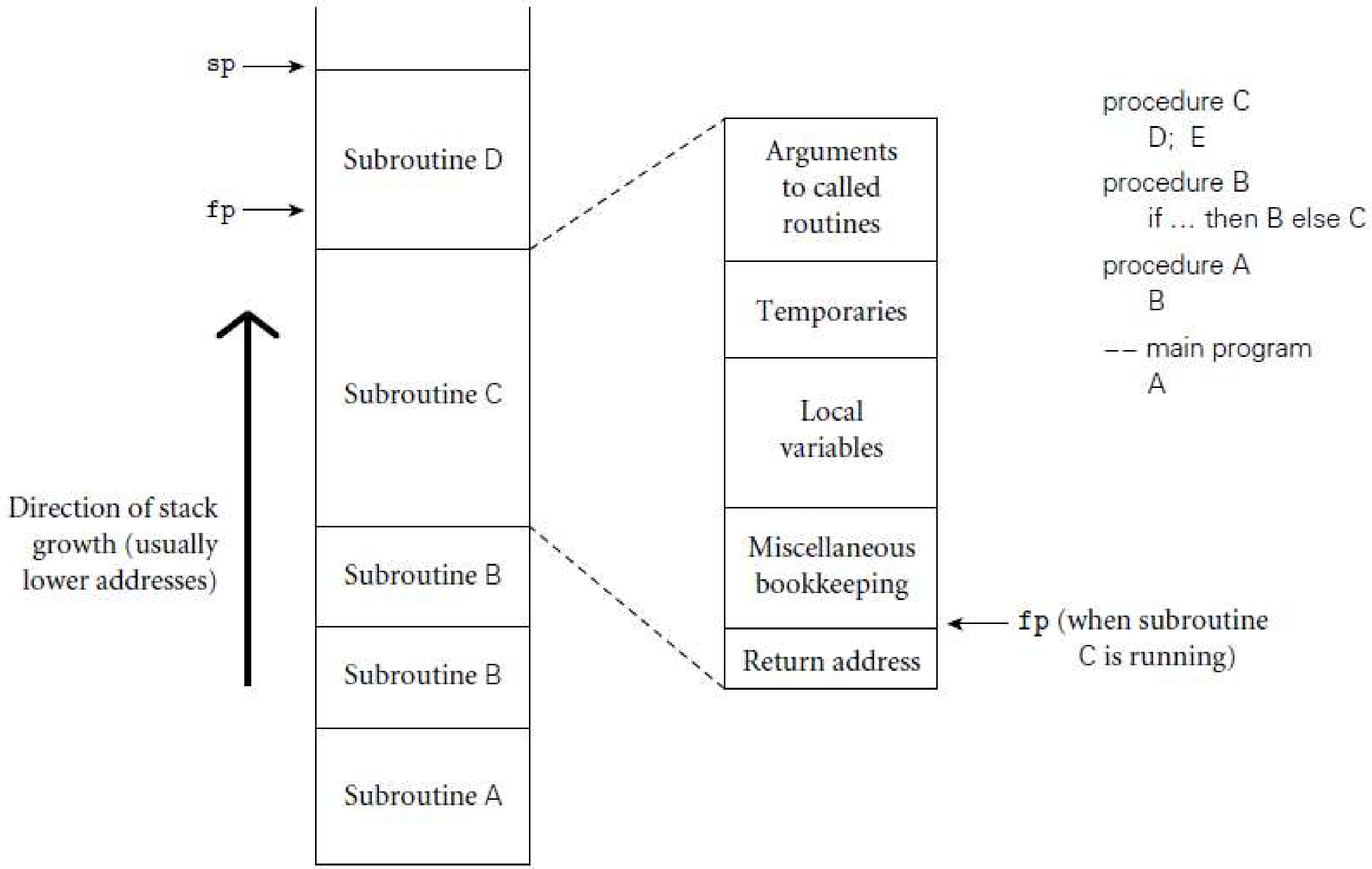
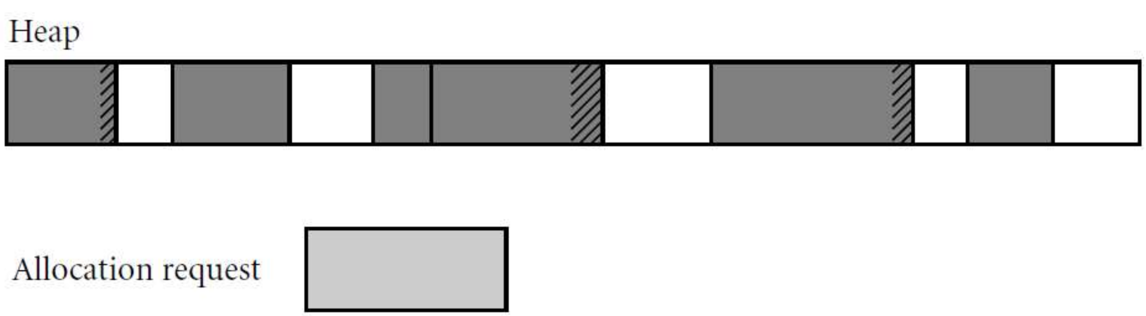


Figure 3.1 Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously 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

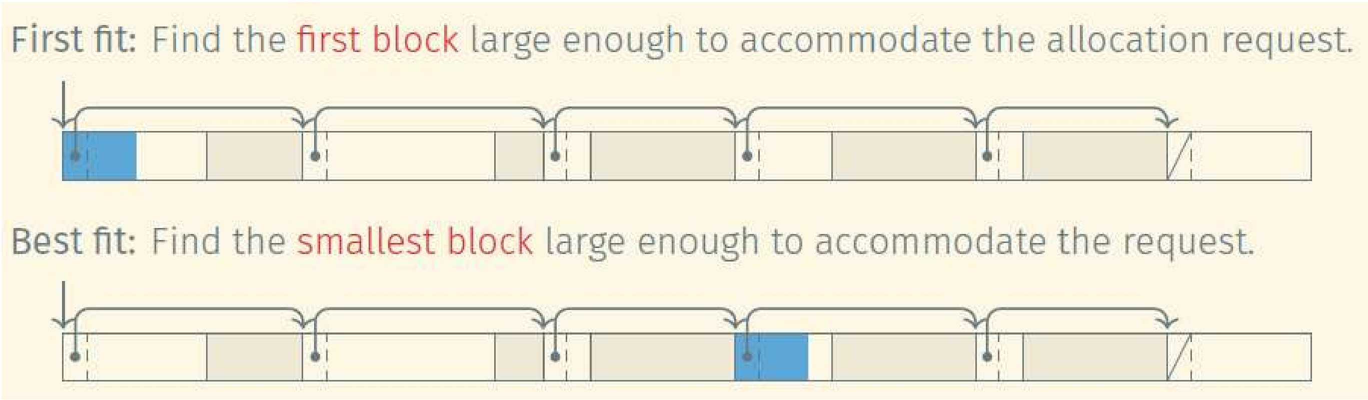


Heap Management

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

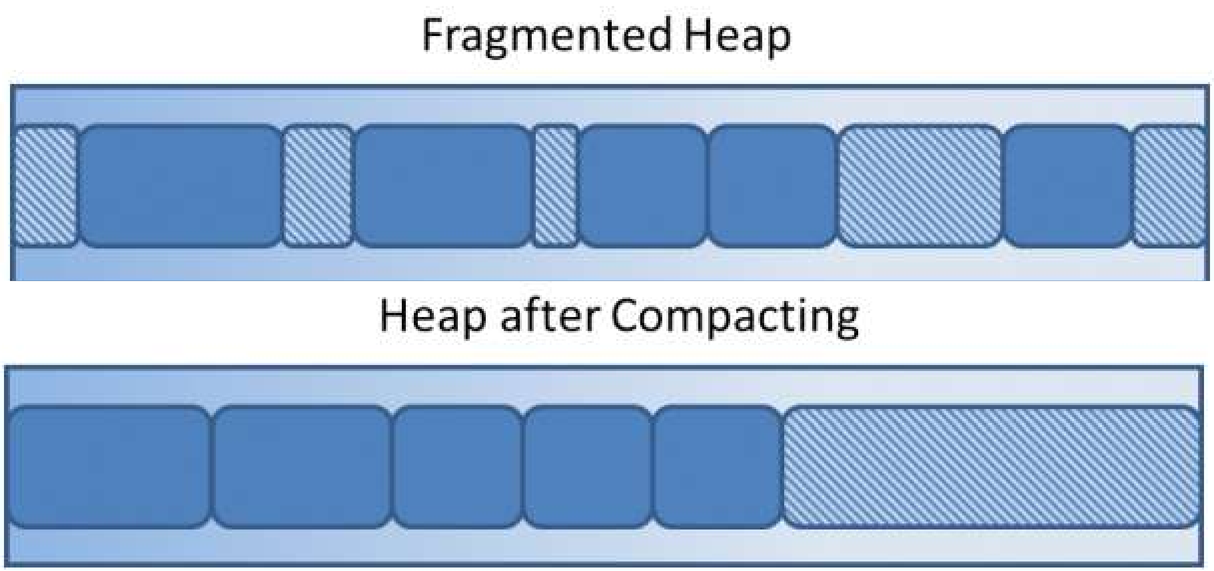
- *Internal fragmentation* occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object
  - e.g. Boolean is stored in 1 bit/1 byte
- *External fragmentation* occurs when the remaining, unused space is composed of multiple blocks
  - There may be quite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

Heap Compaction

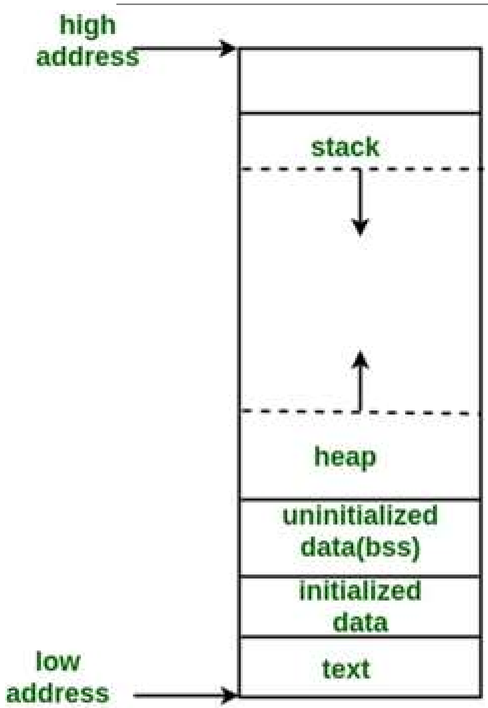
To fight fragmentation, some memory management algorithms perform heap compaction once a while

Mark-Compact Algorithm

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



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.

Example: https://www.geeksforgeeks.org/memory-layout-of-c-program/

C Program without any variables

```c
#include <stdio.h>

int main(void)
{
    return 0;
}
```

The size command reports the sizes (in bytes) of the text, data, and bss segments.

```
[narendra@CentOS]$ size memory-layout
text        data        bss        dec        hex     filename
960         248          8         1216       4c0     memory-layout
```

C Program with a global variable

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    return 0;
}
```

The size command reports the sizes (in bytes) of the text, data, and bss segments. (bss - block started by symbol)

```
[narendra@CentOS]$ size memory-layout
text       data       bss       dec       hex    filename
 960        248        12       1220      4c4    memory-layout
```

C Program - Add a static variable

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss
    return 0;
}
```

The size command reports the sizes (in bytes) of the text, data, and bss segments.

```
[narendra@CentOS]$ size memory-layout
text        data        bss        dec        hex    filename
 960         248         16       1224        4c8    memory-layout
```

C Program - Initialize a static variable

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i = 100; /* Initialized static variable
                              stored in DS*/
    return 0;
}
```

The size command reports the sizes (in bytes) of the text, data, and bss segments.

```
[narendra@CentOS]$ size memory-layout
text        data        bss        dec        hex     filename
960          252         12       1224        4c8     memory-layout
```

C Program - Initialize a global variable

```c
#include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored
    in DS*/
    return 0;
}
```

The size command reports the sizes (in bytes) of the text, data, and bss segments.

```
[narendra@CentOS]$ size memory-layout
text        data        bss        dec        hex     filename
960          256          8       1224        4c8     memory-layout
```

Scoping

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

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

Lexical (static) scoping

- Binding based on nesting of blocks
- Can be determined at compile time

Dynamic scoping

- Binding depends on flow of execution at runtime
- Can only be determined at runtime

Scope Rules

Scoping rule example:

- 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

Static 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 is used in languages like C and Java

Java Example 1

```
/* Java Program Example - Java Variables Scope */
public class JavaProgram
{
    public static void main(String args[])
    {

        int x;     //known to all code within main

        x = 10;
        if(x == 10)
        {
            int y = 20;    //known only to this block

            /* x and y both known here */
            System.out.println("x : " + x + "\ny : " + y);
            x = y * 2;
        }
        // y = 100;   //error! y not known here

        /* x is still known here */
        System.out.println("x is " +x);

    }
}
```

Java Example 2

```
/* Java Program Example - Demonstrate lifetime of a variable - Java Scope
Rules */

public class JavaProgram
{
    public static void main(String args[])
    {

        int x;

        for(x=0; x<5; x++)
        {
            int y = -1;    //y is initialized each time block is entered
            System.out.println("y is : " +y);    //this always prints -1

            y = 100;
            System.out.println("y is now : " +y);
        }

    }
}
```

C Example

```c
// A C program to demonstrate static scoping.
#include<stdio.h>
int x = 10;

// Called by g()
int f()
{
   return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
   int x = 20;
   return f();
}

int main()
{
  printf("%d", g());
  printf("\n");
  return 0;
}
```

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
```

Dynamic Scoping

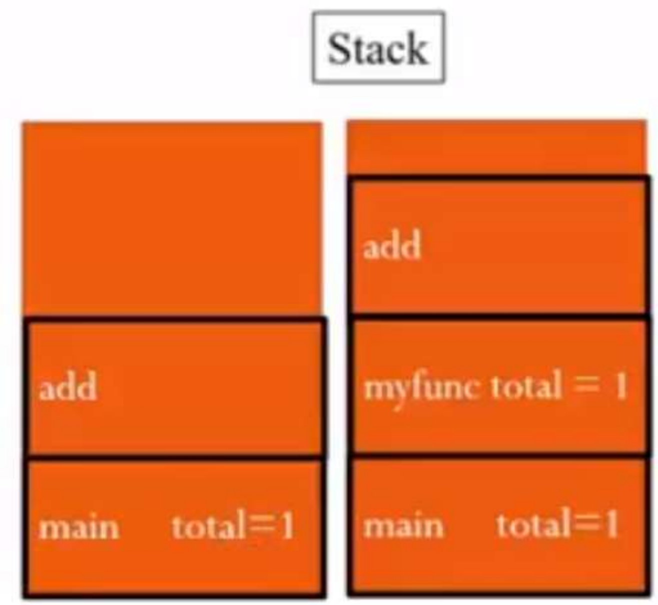
Dynamic scope rules: bindings depend on the current state of program execution:

- 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

Example

```
int x = 10;

// Called by g()
int f()
{
   return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
   int x = 20;
   return f();
}

main()
{
  printf(g());
}
```

Output with static scoping:

Output with dynamic scoping:

Perl Example I

```
# Lexical and dynamic scopes in Perl;
# Perl's keyword "my" defines a statically scoped local variable;
#the keyword "local" defines dynamically scoped local variable.

$a = 0;
sub foo {
  return $a;
}
sub staticScope {
  my $a = 1; # lexical (static)
  return foo();
}

print staticScope()."\n"; # 0

$b = 0;
sub bar {
  return $b;
}
sub dynamicScope {
  local $b = 1;
  return bar();
}
print dynamicScope()."\n"; # 1
```

Perl Example II

```perl
# A perl code to demonstrate dynamic scoping
$x = 10;
sub f
{
   return $x;
}
sub g
{
   # Since local is used, x uses
   # dynamic scoping.
   local $x = 20;

   return f();
}
print g()."\n";
```

Overloading

- Same name, more than one meaning
- Some overloading happens in almost all languages
  - integer +   vs.   real +   vs. String concatenation
- 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;
  }
}
```