Dynamic Vs. Static memory allocation

Sometimes we create data structures that are “fixed” and don’t need to grow or shrink(Static Allocation). At other times, we want to increase and decrease the size of our data structures to accommodate changing needs(Dynamic Allocation). In the following difference between static and dynamic allocation.

Static Allocation (fixed in size)	Dynamic Allocation (change in size)
Done at compile time. Global variables: variables declared Advantage: efficient execution time. Disadvantage If we declare more static data space than we need, we waste space. If we declare less static space than we need, we are out of luck.	Done at run time. Data structures can grow and shrink to fit changing data requirements. We can allocate (create) additional storage whenever we need them. Advantage: we can always have exactly the amount of space required - no more, no less.

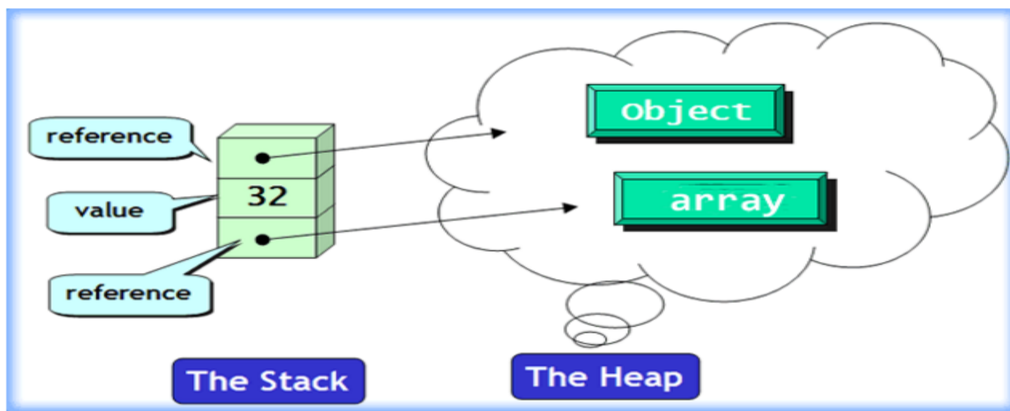
A program is assigned three independent portions of memory, as address space:

• **Code area** - where code to be executed is stored

• **Dynamic Memory Area as known Heap** to store variables and objects allocated dynamically accessed

• **Execution Stack** - to perform computation, store local variables and perform function call management accessed with a **LIFO** policy.

Difference between Stack memory and Heap Memory



Stack	Heap
1) Used to store local variables,address of the objects and method calls.	1) Used to store Object values in java.
2) Data in the stack dies or gets deleted once the block of code or method call terminates.	2) Data in the heap dies till the object gets garbage collected.
3) Stack is faster when compared to heap	3) Heap is slower when compared to stack

Activation Records(AR)

Each time a method is called, all the information needed for the method execution is put on the stack. That information is called the **activation record (AR)** of the method call. When the call is completed the corresponding AR is destroyed. Activation records are organized *from bottom to top* in memory diagram.

When recall a program, the storage partition looks like this:

When a when program's main method is started, the partition looks like this:

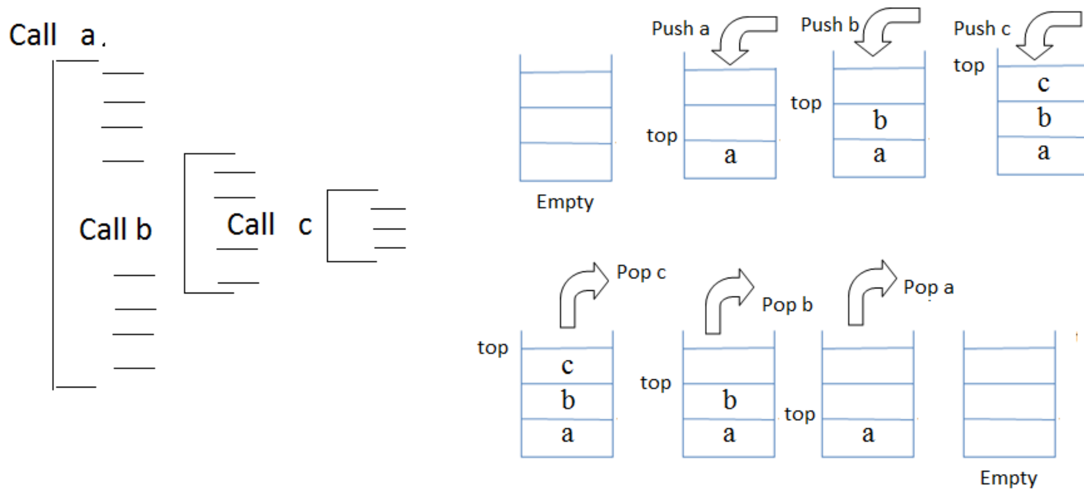
Activation record for main holds cells for main's **local variables** and the **return address** to which the JVM should jump when main finishes. Say that main calls a method, p. Then, an activation record holding p's local variables and return address is pushed onto the stack:

If p calls q, the same happens:

When q finishes, its activation record is popped, and the configuration reverts to this figure:

When p finishes, its record is popped, and main can finish.

☒ **Use Stack to Execute call methods**



Recursion

A ***recursive method*** is a method that contains a statement (or statements) that makes a call to itself.

How do I write a recursive function?

any recursive method will include the following three basic elements:

1. A test to stop or continue the recursion.
2. An end case that terminates the recursion.
3. A recursive call(s) that continues the recursion.

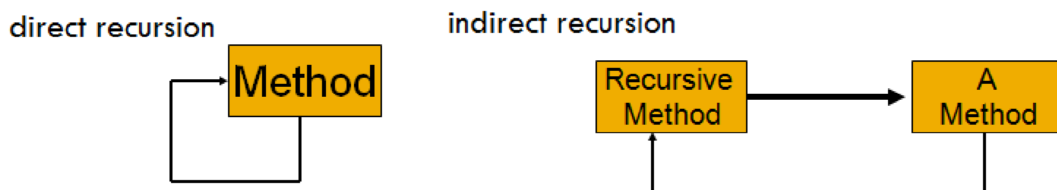
Example: recursive factorial method.

```
public int factorial(int N)
{
   if (N == 1) {   <---- Test to stop or continue.
      return 1;    <---- End case: recursion stops.
   }

   else {
      return N * factorial(N-1);
   }              ^-- Recursive case: recursion
}                     continues with a recursive call.
```

In Java, each time a method (recursive or otherwise) is called, a structure known as an ***activation record*** or ***activation frame*** is created to store information about the progress of that invocation of the method. This frame stores the parameters and local variables specific to a given call of the method, and information about which command in the body of the method is currently executing.

Q: **Who can be called a recursive method?**

Ans.: A recursive method can be called either **directly** recursion, or **indirectly** recursion through another method, which may in turn make a call back to the recursive method.



Recursion vs. iteration

- ☐ Both iteration and recursion are based on a control structure.
  - Iteration uses a repetition structure (such as for, while or do/while)
  - Recursion uses a selection structure (such as if, if/else or switch).
- ☐ Both iteration and recursion involve repetition.
  - Iteration explicitly uses a repetition structure.
  - Recursion achieves repetition through repeated method calls.
- ☐ Iteration and recursion each involve a termination test.
- ☐ Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

What happens when a function b method called?

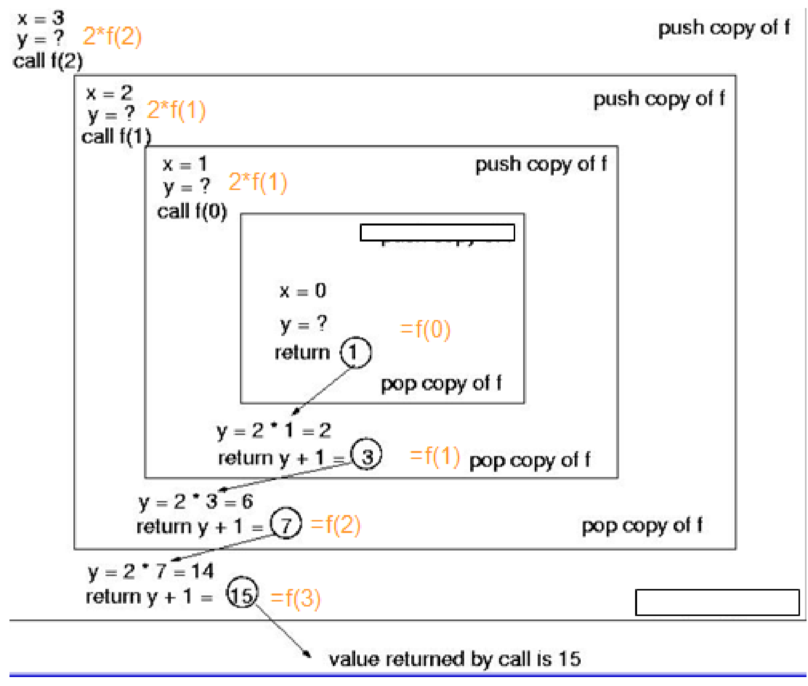
```
int a(int w) {
  return w+w; }

 int  b(int x) {
  int  z,y;
     …… // other statements
  z = a(x) + y;
 return z; }
```

Ans.:

1. Since it needs to come back to function ***b*** later, it needs to store everything about function **b** that is going to need (**x, y, z**, and the place to start executing upon return, then An **activation** record is stored into a stack .
2. Then, **x** from **a** is bounded to **w** from **b**
3. Control is transferred to function **a**
4. After function **a** is executed, the activation record is popped out of the stack
5. All the old values of the parameters and variables in function **b** are restored and the return value of function **a** replaces **a(x)** in the assignment statement

What happens when a recursive function f(3) is called?

```
int  f(int x) {
   int  y;
   if  (x==0)     return 1;
   else {
     y = 2 * f(x-1);
     return  y+1; }
}
```

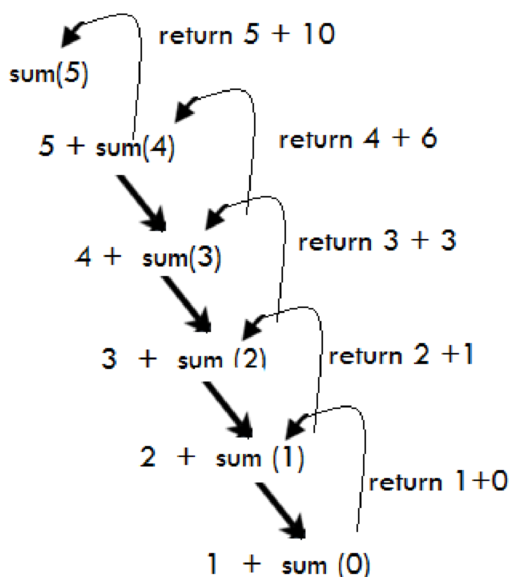


Example: Write a function int sum (int n) that computes the sum of numbers from 1 to n in two cases : use loop and recursively

```
int sum (int n){
   int s = 0;
   for (int i=0; i<n; i++)
      s += i;
   return s;}
```

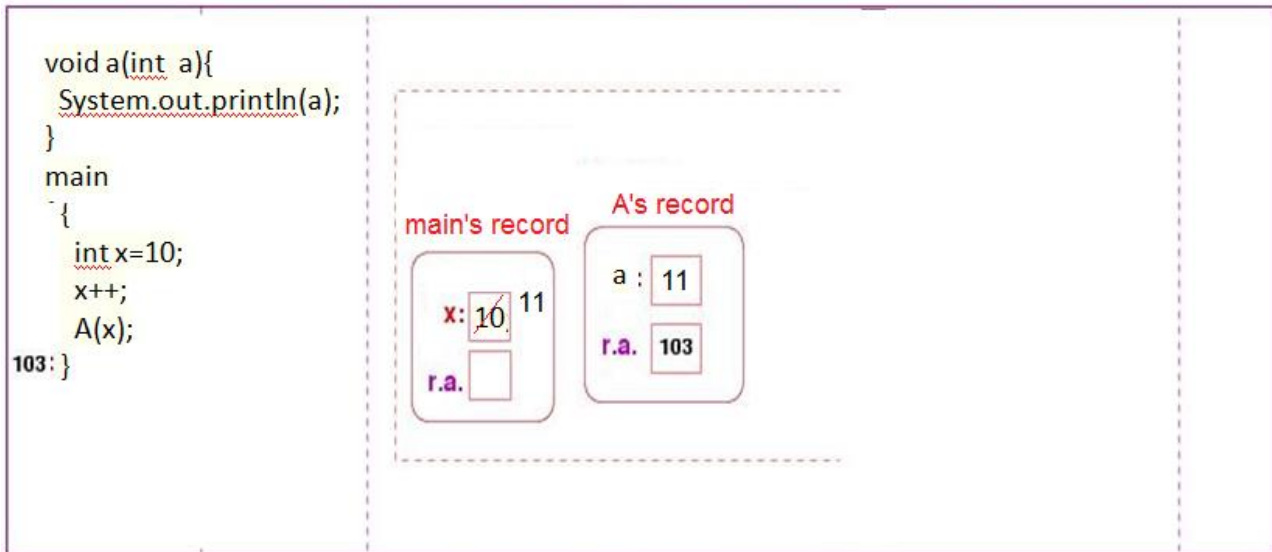
```
int sum (int n){
   int s;
   if ( n == 0 ) return 0;
   else          s = n + sum(n-1)
   return s;}
```

How dose it work?

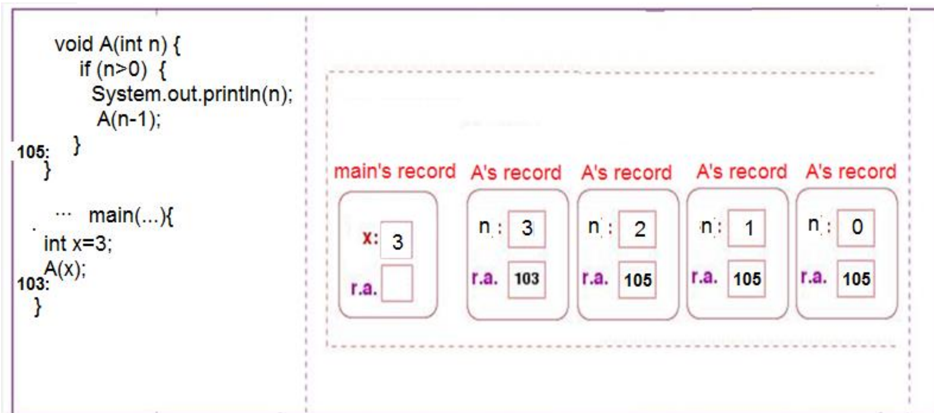


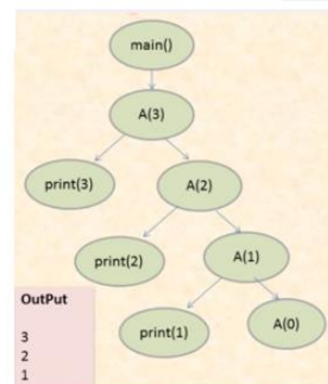
☒ Use stack to execute recursion method

Example1:

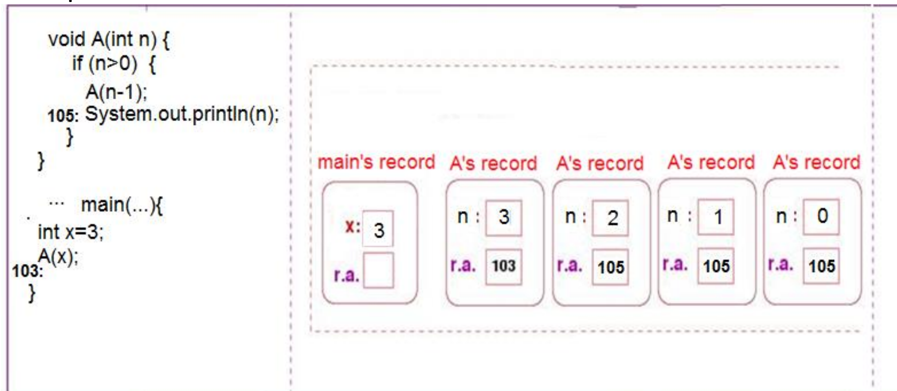


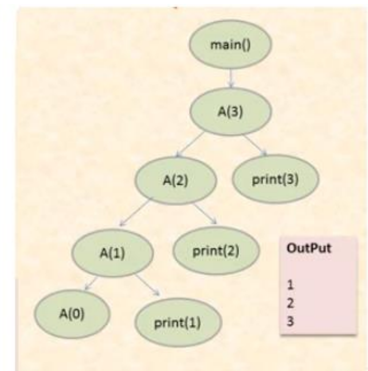
Example2:





Example3:





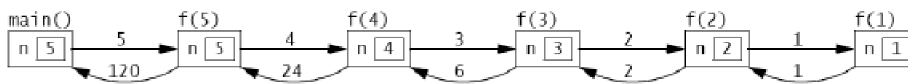
Example4: The *factorial* function is defined mathematically by:

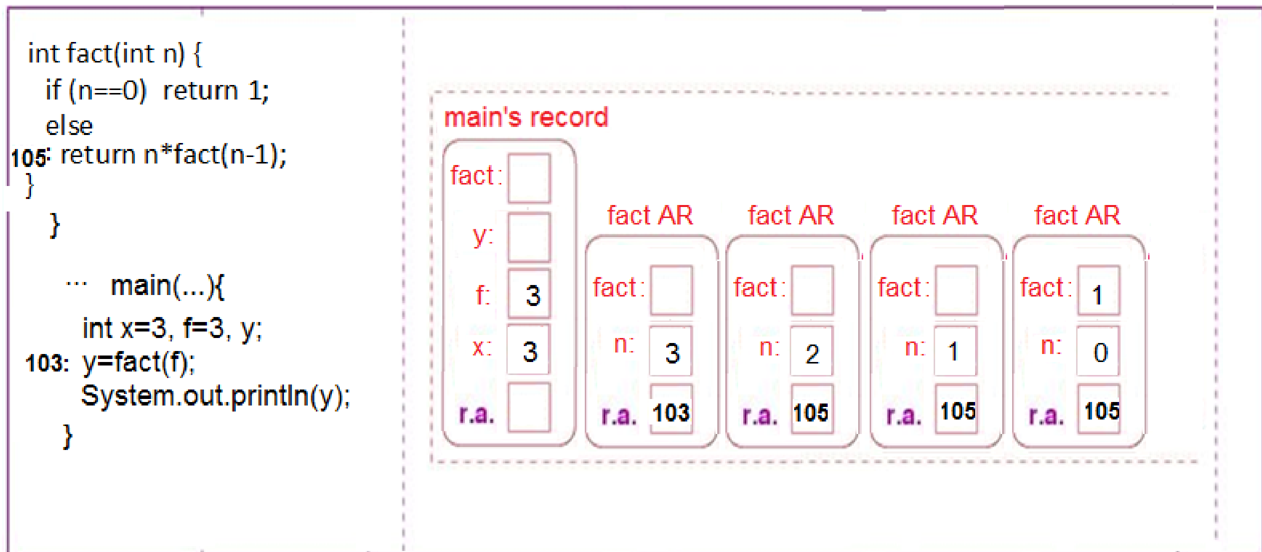
$$n! = \begin{cases} 1, \text{if } n = 0 \\ n(n-1)!, \text{if } n > 0 \end{cases}$$

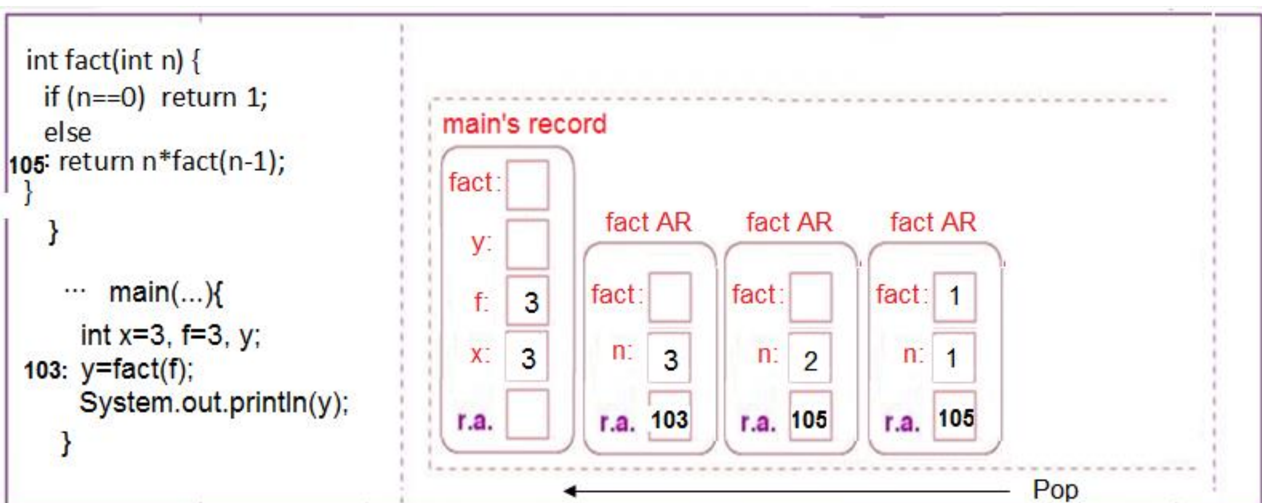
Programming definition of factorial:

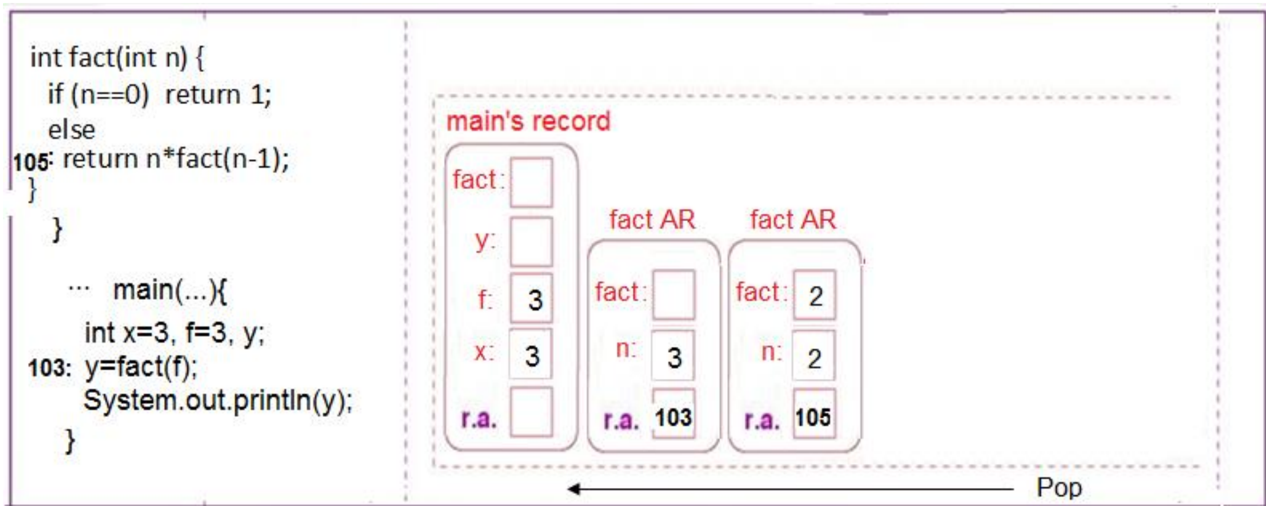
```
int fact(int n) {
  if (n==0)        return 1;    // basis
  else             return n*fact(n-1); // recursive part
}
```

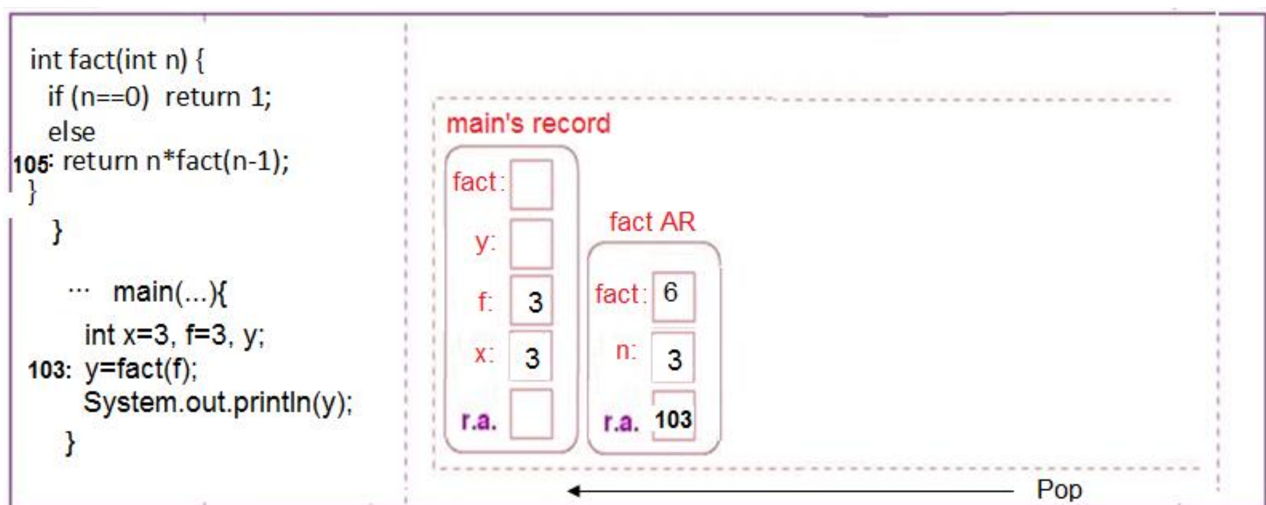
Trace on call fact(5);

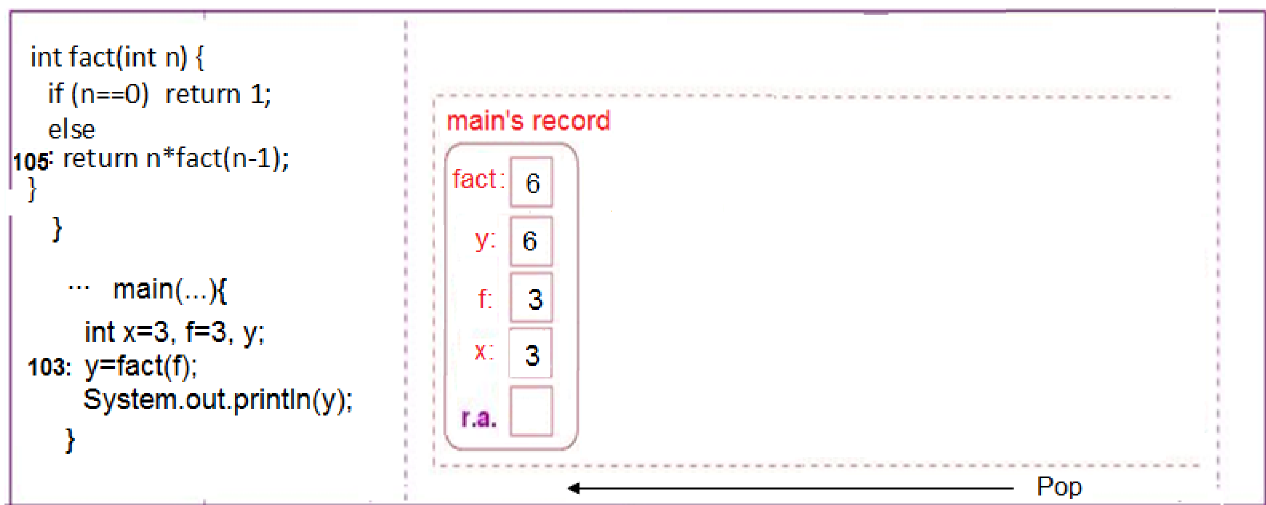












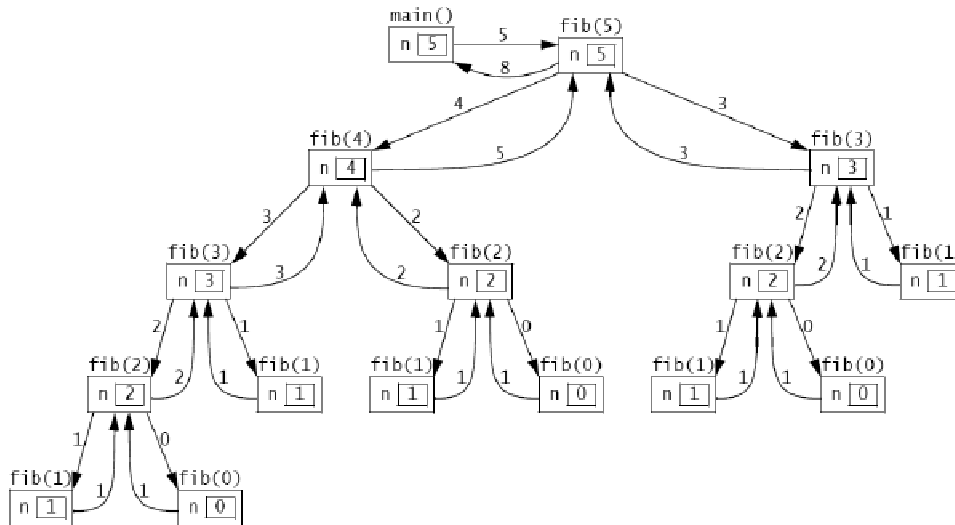
Example5: The *Fibonacci numbers* are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …. Each number after the second is the sum of the two preceding numbers. The *Fibonacci* function is defined mathematically by:

$$F_n = \begin{cases} 0, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ F_{n-1} + F_{n-2}, \text{ if } n > 1 \end{cases}$$

Programming definition of Fibonacci:

```
int fib(int n) {
   if (n < 2)      return n;     // basis
   else            return fib(n-1) + fib(n-2);     // recursive part
}
```

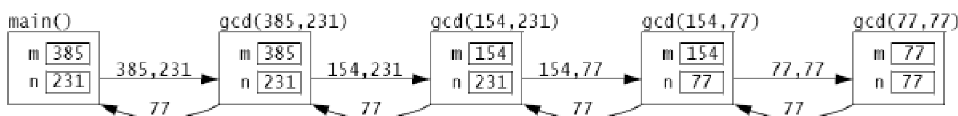
Trace on call fib(5);



Example 6. To find gcd for two number, the following algorithm simply subtracts the smaller number from the larger. This is done recursively by calling either gcd(m,n m) or gcd(m n,n):
Programming definition of gcd:

```
int gcd(int m, int n) {
  if (m==n)        return n;                  // basis
  else if (m<n)    return gcd(m,n-m);  // recursion
  else             return gcd(m-n,n);  // recursion
 }
```

Trace on the call gcd(385, 231).



Example7: The *Power* function is defined mathematically by:

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

Programming definition of factorial:

```
double power(double x, int n) {
    if (n == 0)        return 1;
    else               return x * power(x, n−1);
}
```

Trace on call Power(3,2)?

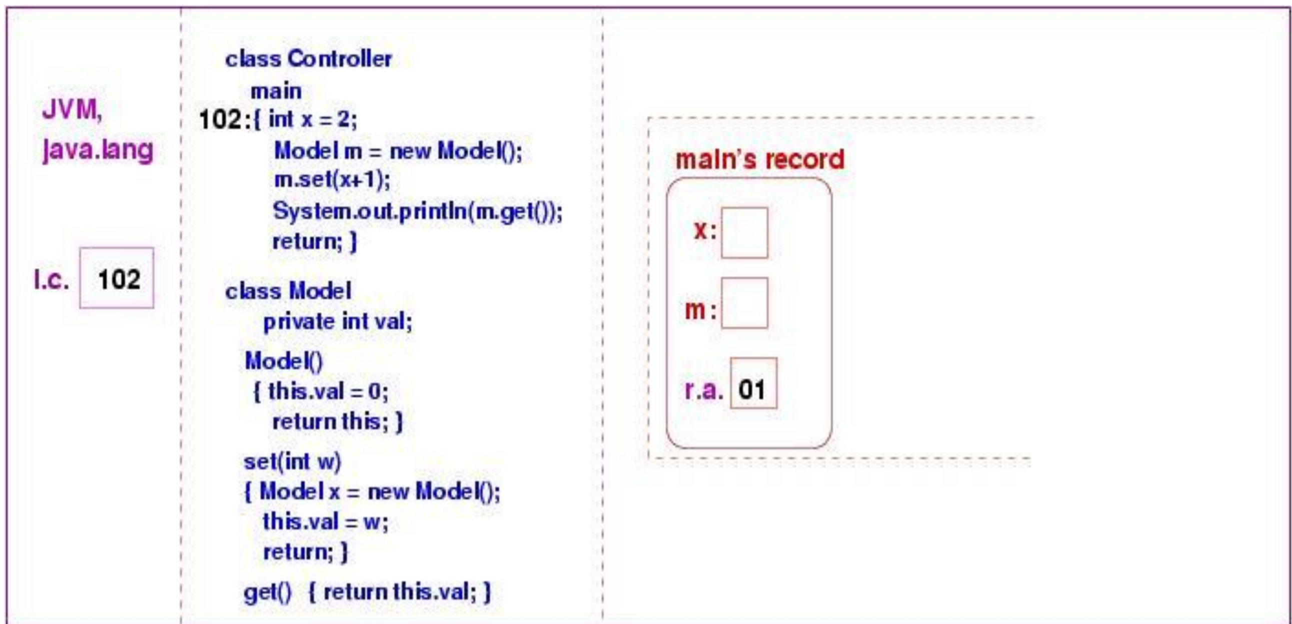
Problems:

1. Write a recursive function that returns the sum of the squares of *n* positive integers.
2. Write a recursive function that returns the sum of *n* powers of a base *b*.
3. Write and a recursive function that returns the sum of *n* elements of an array.
4. Write a recursive function that returns the maximum among *n* elements of an array.
5. Write a recursive boolean function that determines whether a string is a palindrome. (A *palindrome* is a string of characters that is the same as the string obtained from it by reversing its letters.)

Example : Execute program with objects

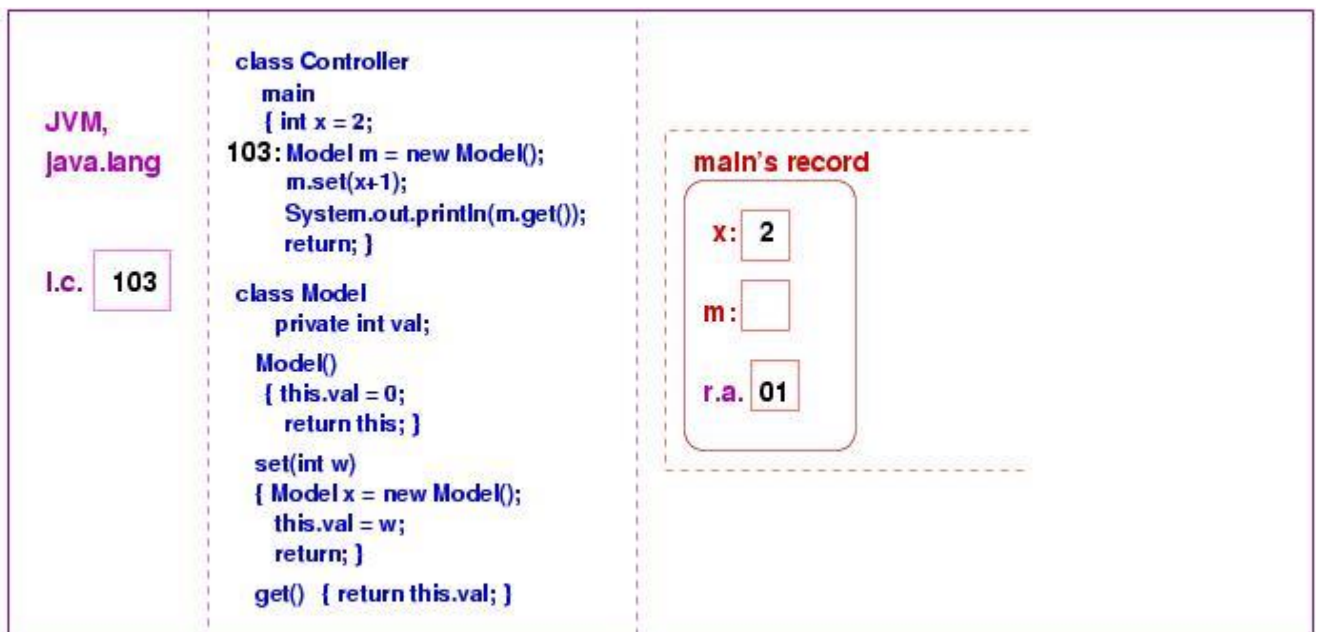
```
public class Controller {
  public static void main(...){
     int x = 2;
     Model m = new Model();
     m.set(x+1);
     System.out.println( m.get() ); }
}

public class Model{
   private int val;

   public Model() {
      val = 0;
    }

   public void set(int w){
     Model x = new Model();
     val = w;
    }

   public int get() {
     return val; }
    }
```

Main's record holds cells for its two local variables, x and m, and the first instruction that executes, at address 102, saves 2 in x's cell:



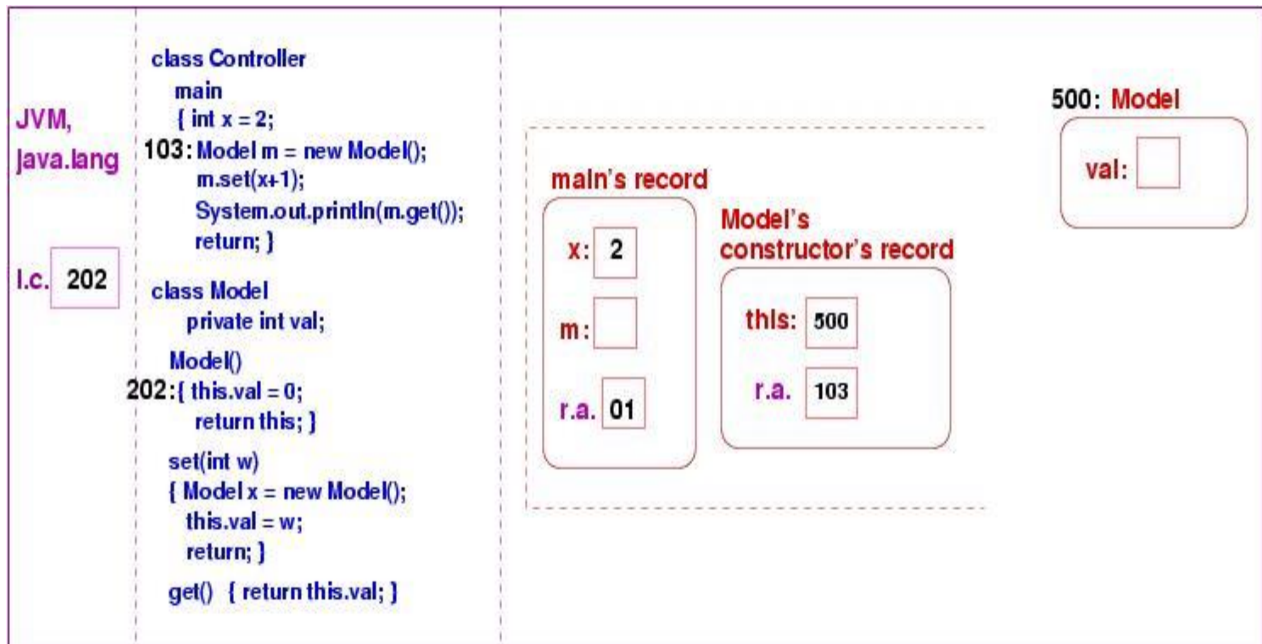
The next instruction, at address 103, constructs a new Model object. Several steps must be performed:

1. A Model object is constructed in heap storage, and a cell for its private variable, val, is allocated therein.
2. The object's constructor method is immediately executed.
3. The address of the newly constructed object is returned as the result of executing the constructor method.

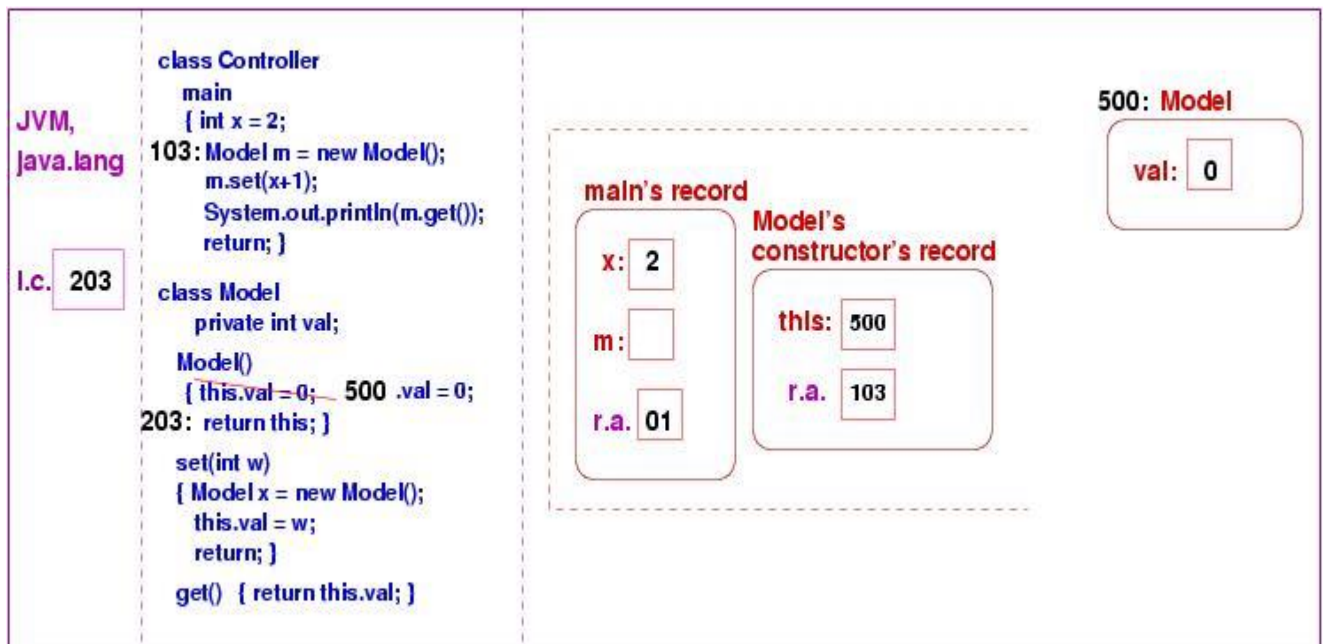


The picture below shows Step 1: The object is constructed, a cell is allocated for its variable, val, and its constructor method is started. (The instruction counter in the JVM is reset to 202, the first instruction in the constructor method.)

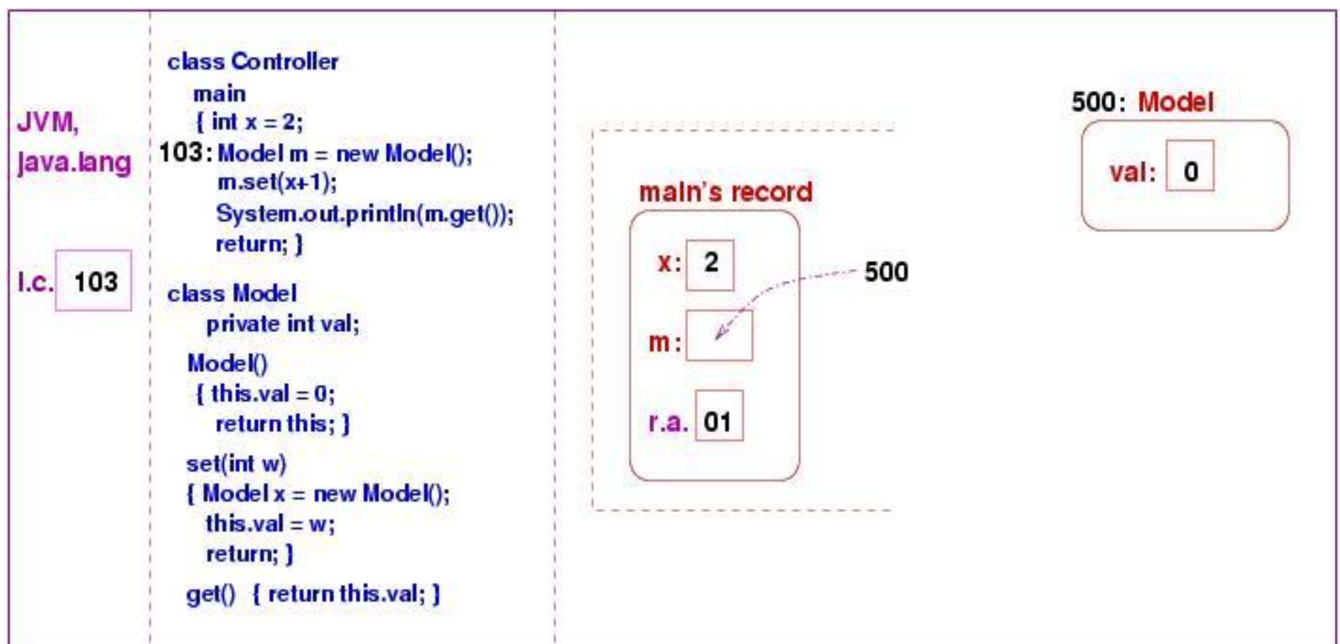
The constructor method's activation record is pushed onto the stack. It holds cells for the method's local variables, and it holds the return address back to main's code.



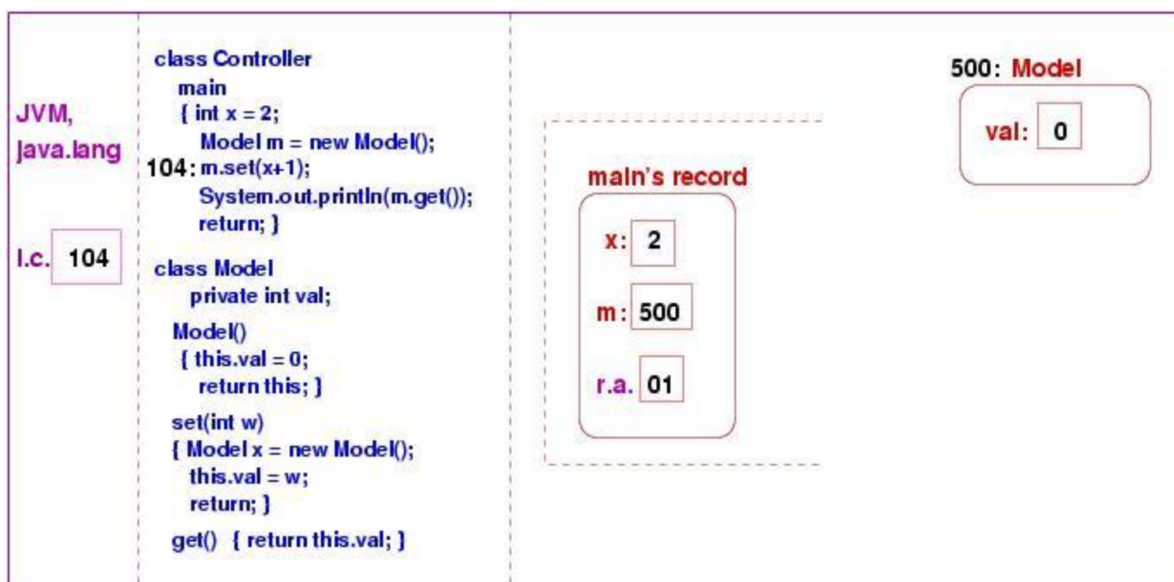
The val variable within the object at address 500 that must be assigned zero:



The **return this** instruction causes the value in the return-address cell to be copied into the instruction counter and also the value in this's cell to be returned to its destination in main.



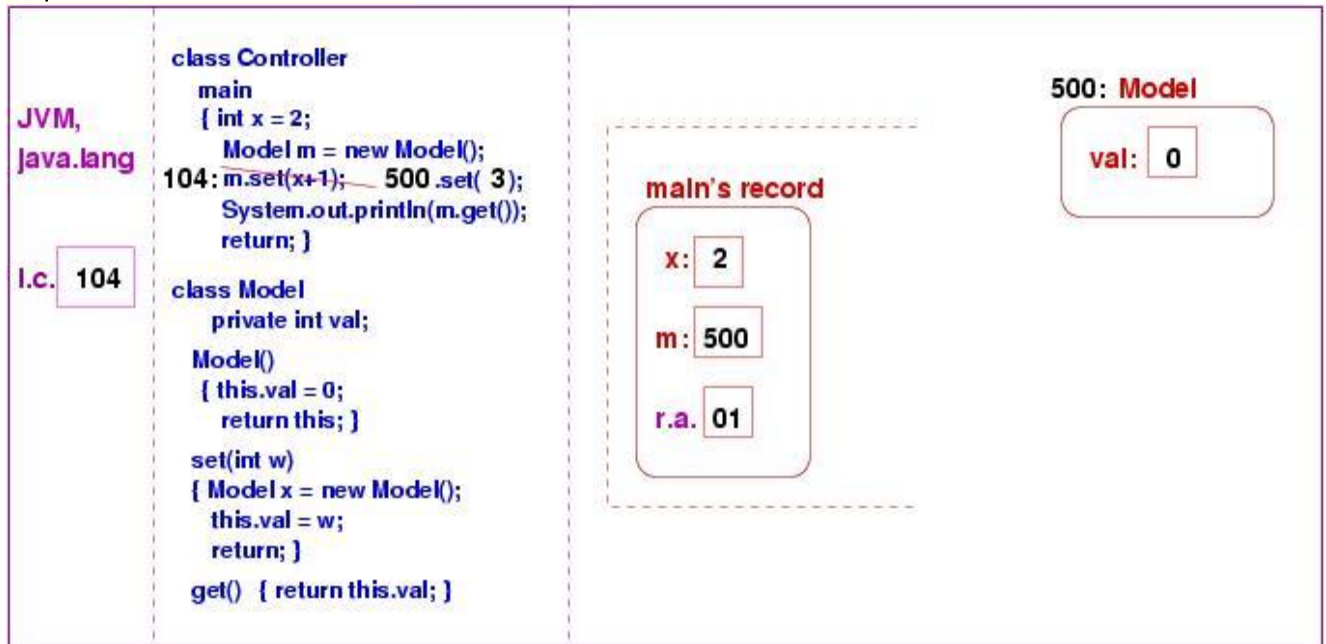
The activation record for the constructor method is erased. Now, the assignment to m is completed:



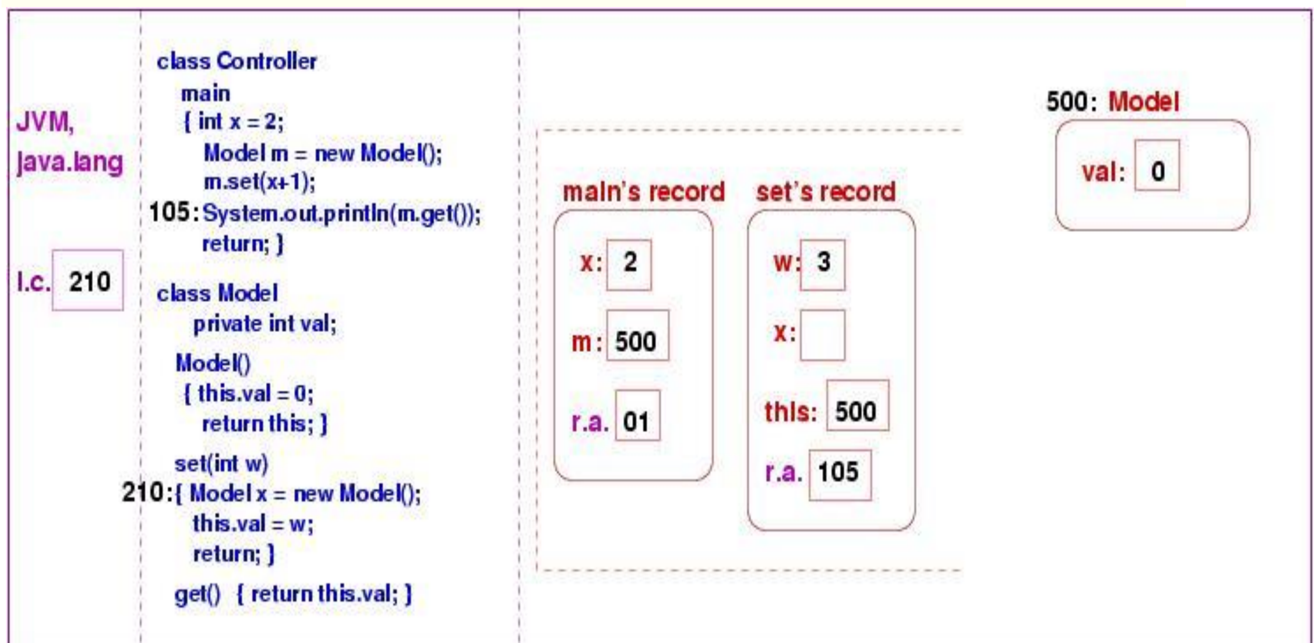
Next, to execute the method invocation, m.set(x+1). The evaluation of the invocation proceeds from left to right:

1. determine the target object's (m's) address. (Here, it is 500.)
2. determine the value of the argument (actual parameter). Here, it is 3.
3. start the invoked method.

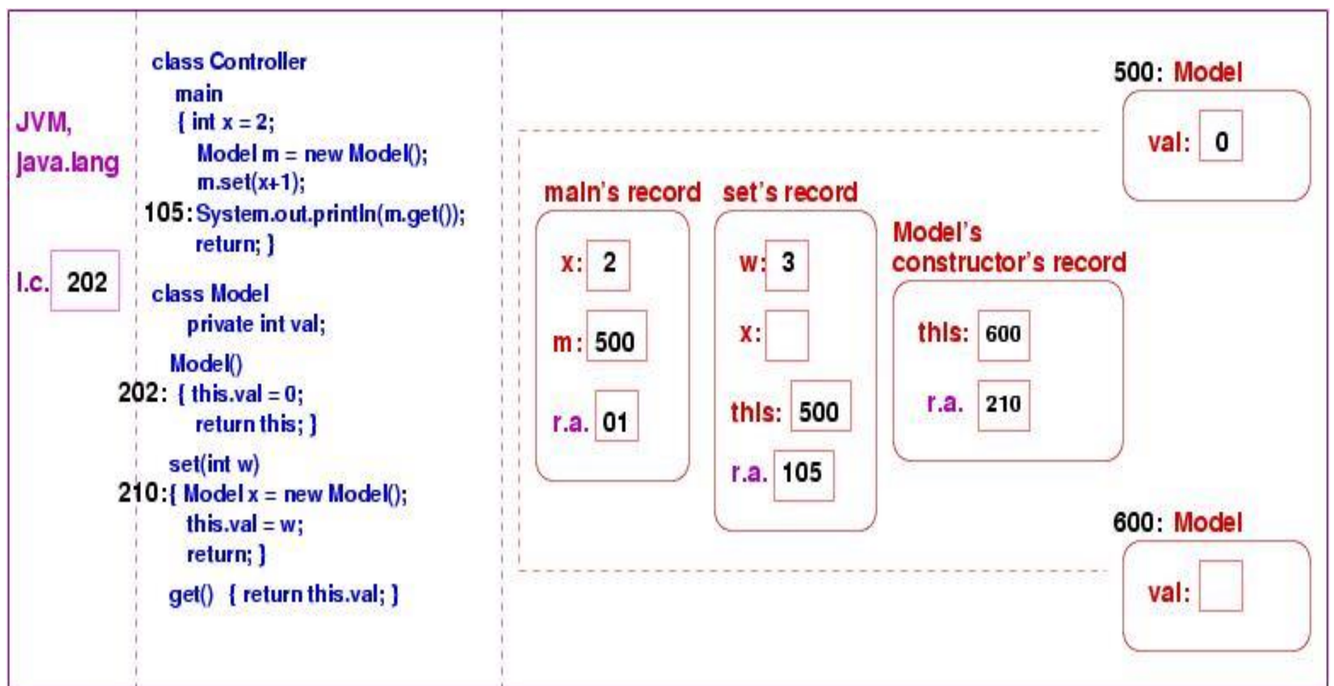
Steps 1 and 2 are shown below:



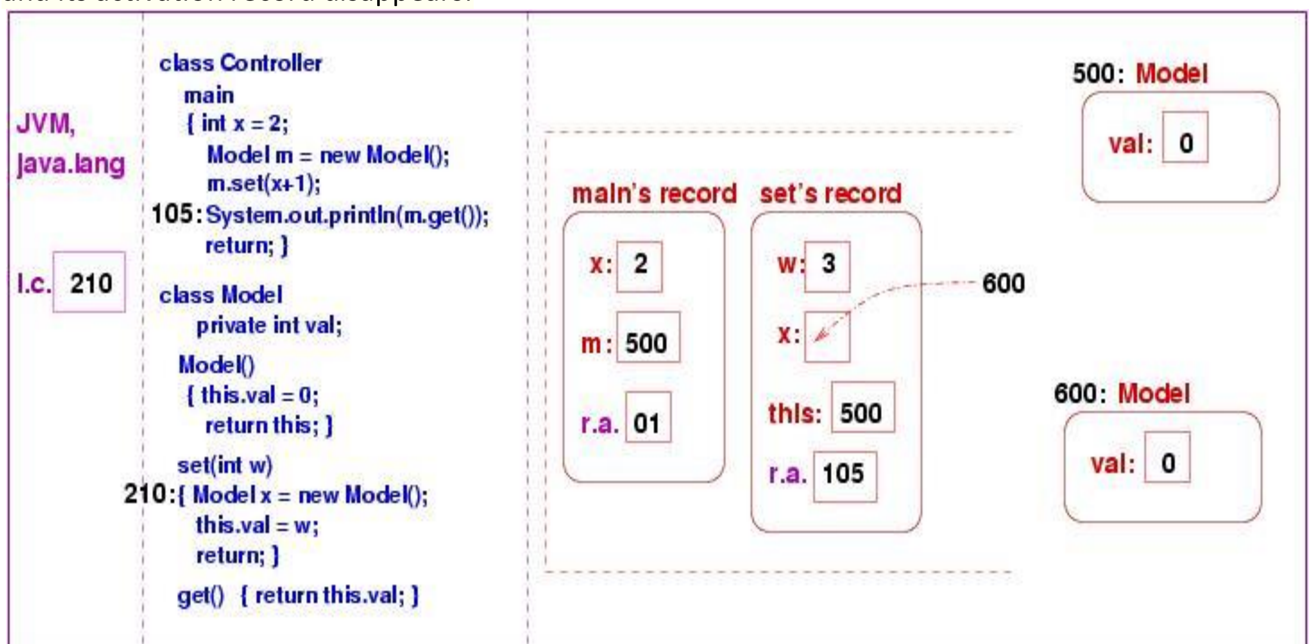
At Step 3 an activation record for method set is pushed, where its this variable is initialized to 500. Notice that its argument is saved in the variable for formal parameter, w. And, the return address is saved. The instruction counter is reset to the first instruction within the invoked method:



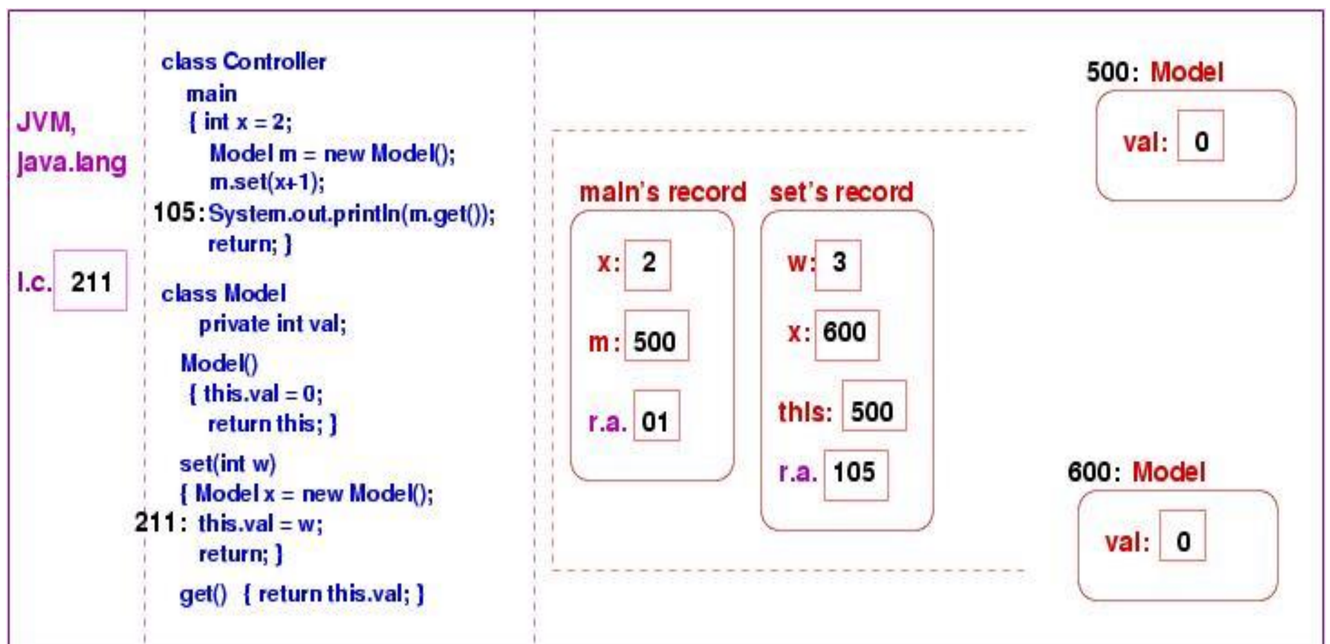
The next instruction constructs a second Model object, and the steps seen a moment ago are repeated. We see a new object constructed and fresh activation record pushed for Model's constructor method:



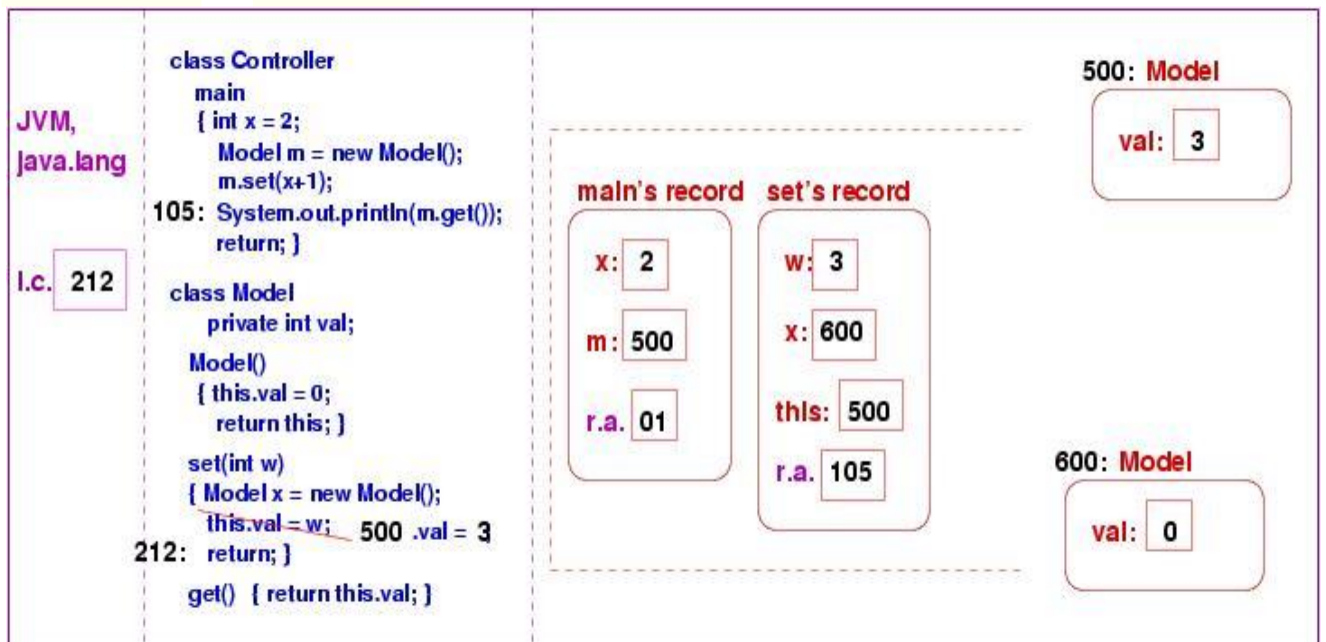
The constructor executes as seen before, using the same instructions as before, but since this's cell holds 600, the new object is correctly initialized. The constructor returns its address to its caller, and its activation record disappears:



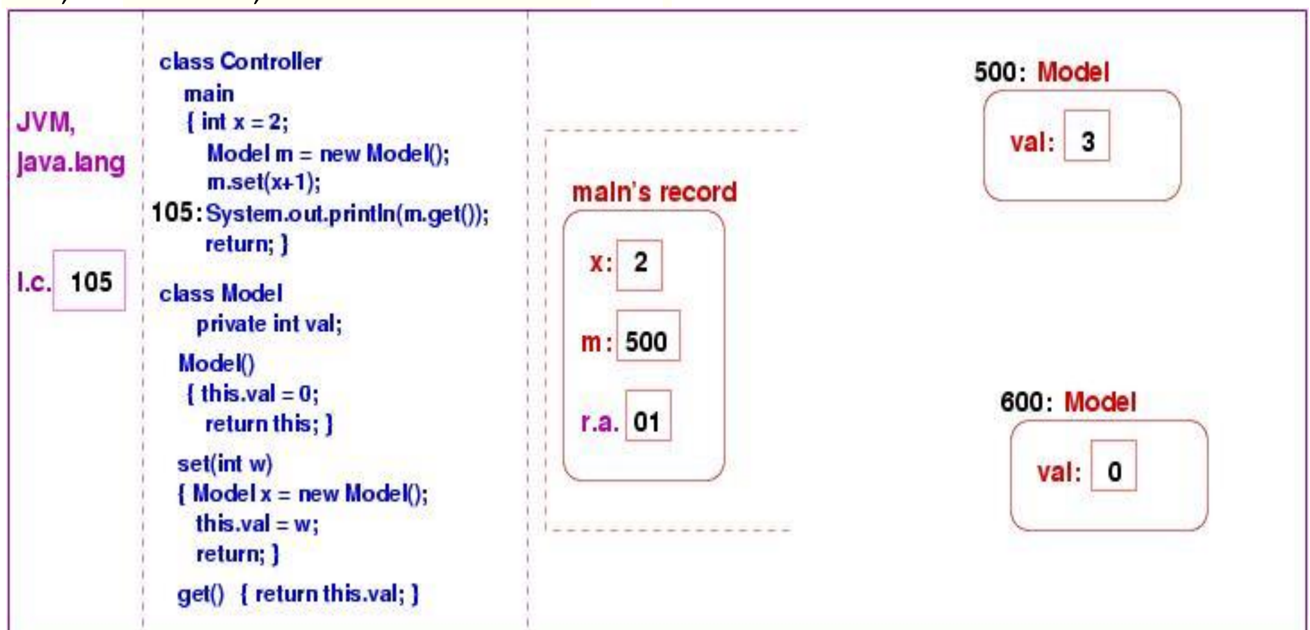
Because the execution has returned to the code for set, and because the topmost activation record holds the variables for set, variable x is correctly assigned:



and the correct val field is reset:



Now, set is finished, and execution returns to main:



Notice that the object at address 600 still rests in storage, even though it is impossible for the application to reference it; the object is *garbage*, and at a later point in the execution, the *garbage collector* program within the JVM will examine all of storage and erase all such unreachable objects.

The next instruction, System.out.println(m.get()), an invocation of get whose this cell holds 500. Once get returns 3, the println method is invoked.