ESc101 : Fundamental of Computing

I Semester 2008-09

Lecture 30

- String (Comparisons)
- Details of method invocation
- Recursion

Comparing two String objects

`str.compareTo(anotherString)`

returns **int** which is less than or equal to or greater than 0 if the String str is less than or equal to or greater than string anotherString.

Note that the strings are compared lexicographically (like in dictionary). So

- "Z" is *greater* "ABC"
- "Y" is *less* than "YA";
- "abc" is *greater* than "aabcdef"

What if there are special characters ?

Comparing two string obects

General algorithm :

Let **s** and **t** are two strings to be compared.

- start scanning **s** and **t** from left to right character by character until you find a mismatch. The order (less than or greater than) between the two strings is determined by the comparison between the **unicode value** of the two characters.

Consider the following program

```
public static int g(int j)
{  int i; i= j*2;
   return i;
}
public static int f(int i)
{  int j = g(i)*i;
   return j;
}
public static void main(String args[])
{ double d; int k=2;
  int i = f(k);
  System.out.println(i);
  ...
}
```

what is the output ?

Consider the following program

```
public static int g(int j)
{  int i; i= j*2;
   return i;
}
public static int f(int i)
{  int j = g(i)*i;
   return j;
}
public static void main(String args[])
{ double d; int k=2;
  int i = f(k);
  System.out.println(i);
  ...
}
```

output = **8**

What about the following program ?

```
public static int f(int i)
{ if(i<=1)  return 1;
  else return f(i-1)*i;
}
public static void main(String args[])
{  int i = 3;
   int j = f(i);
   System.out.println(j);
}
```

What about the following program ?

```
public static int f(int i)
{ if(i<=1)  return 1;
  else return f(i-1)*i;
}
public static void main(String args[])
{  int i = 3;
   int j = f(i);
   System.out.println(j);
}
```

Not clear ??

We need to know the invocation of methods in detail

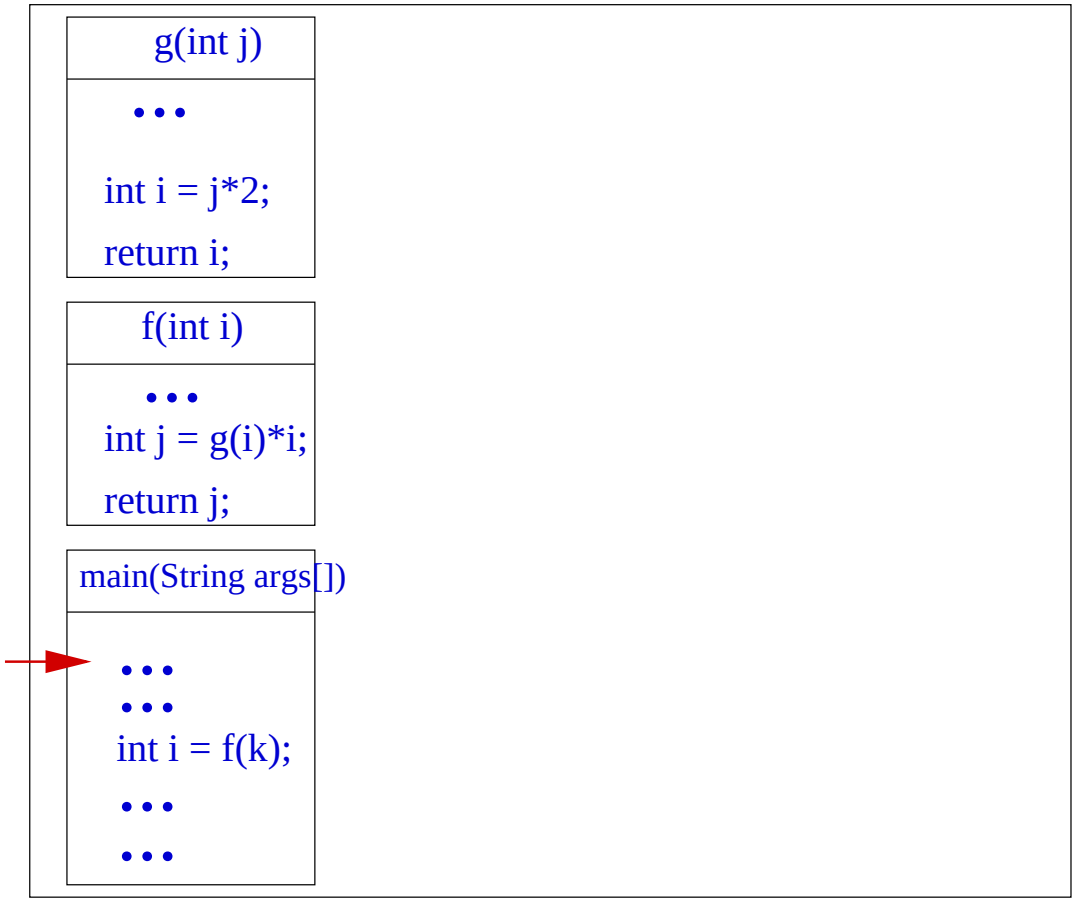
Suppose **M1** invokes **M2**

- How do we keep track of the scope of variables of M1 and M2 at any stage?
- How is it determined where to go at the end of execution of a M2 ?
- How is it ensured that on return from a method **M2** we are in the same state of **M1** in which we left **M1** ?

Let us consider the execution of the first program

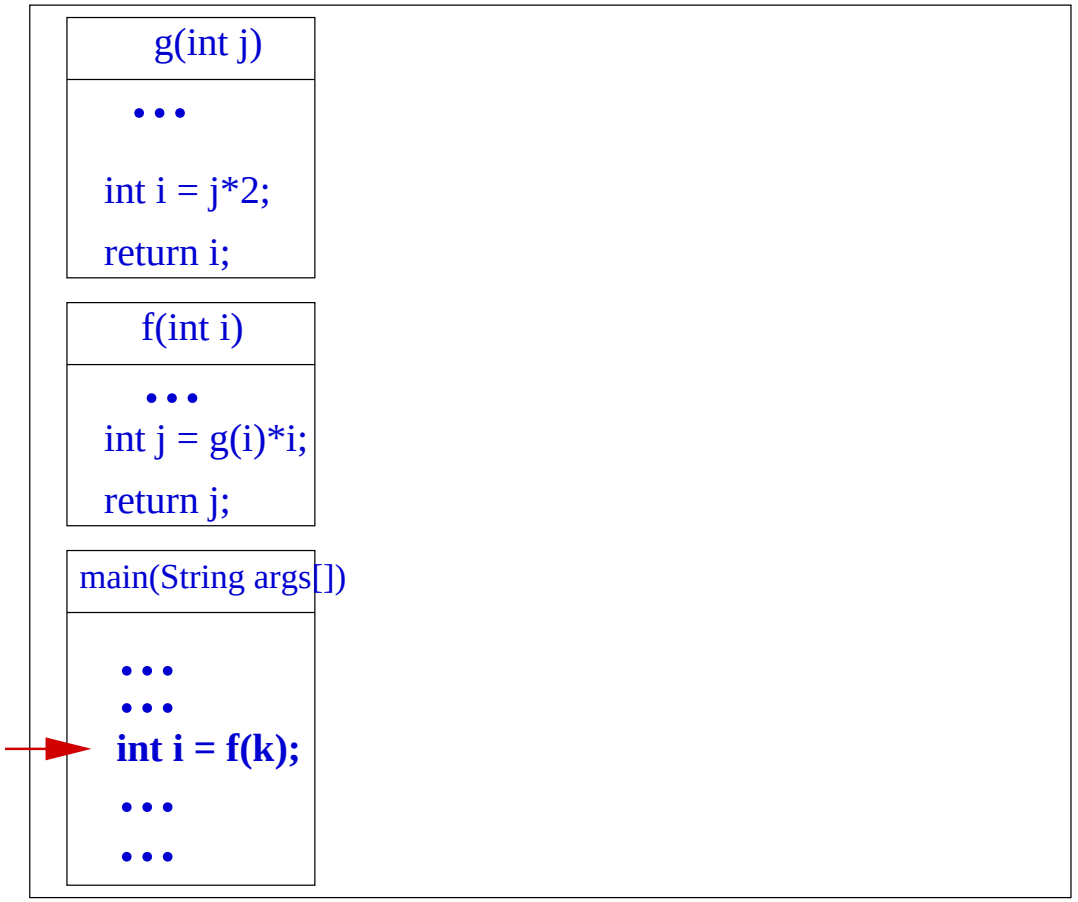
```
public static int g(int j)
{  int i; i= j*2;
   return i;
}
public static int f(int i)
{  int j = g(i)*i;
   return j;
}
public static void main(String args[])
{ double d; int k=2;
  int i = f(k);
  System.out.println(i);
  ...
}
```

Execution starts from `main`



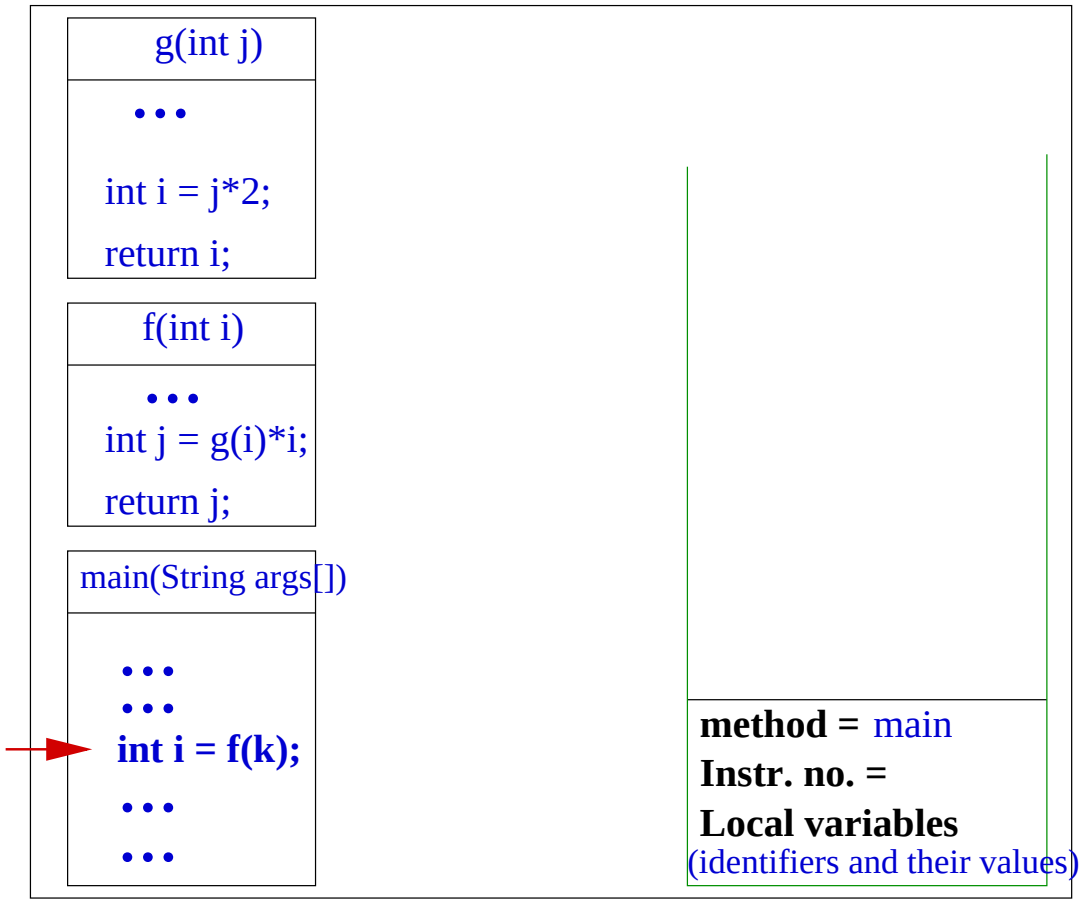
Memory

What actually happens on reaching `int i = f(k)`



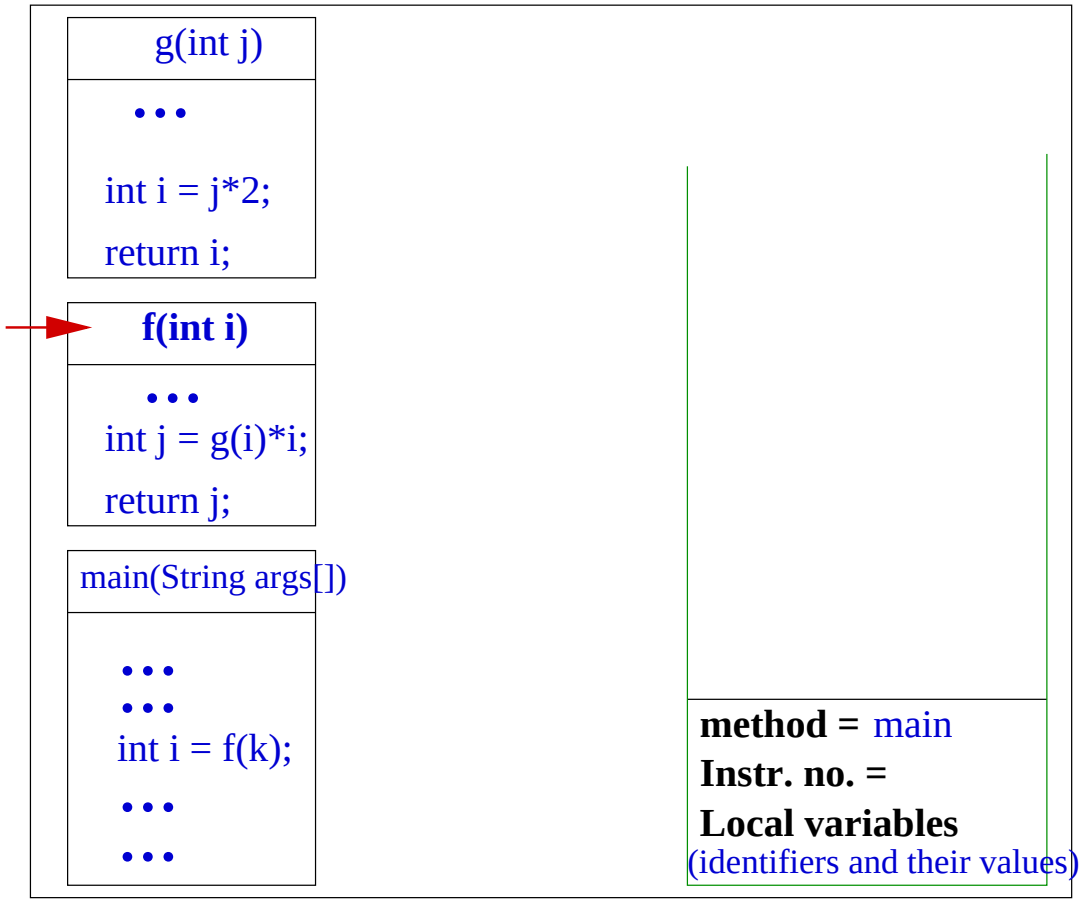
Memory

We keep some records



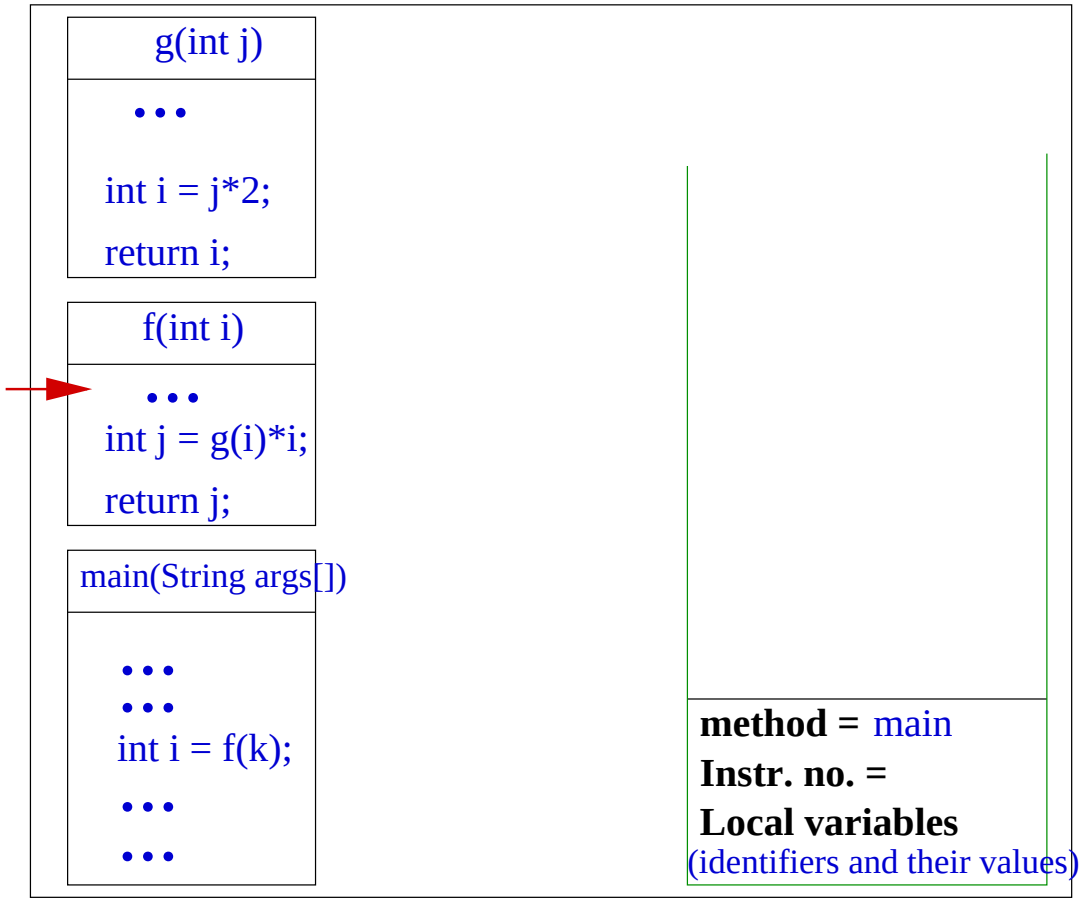
Memory

Then we start excuting `f( )`



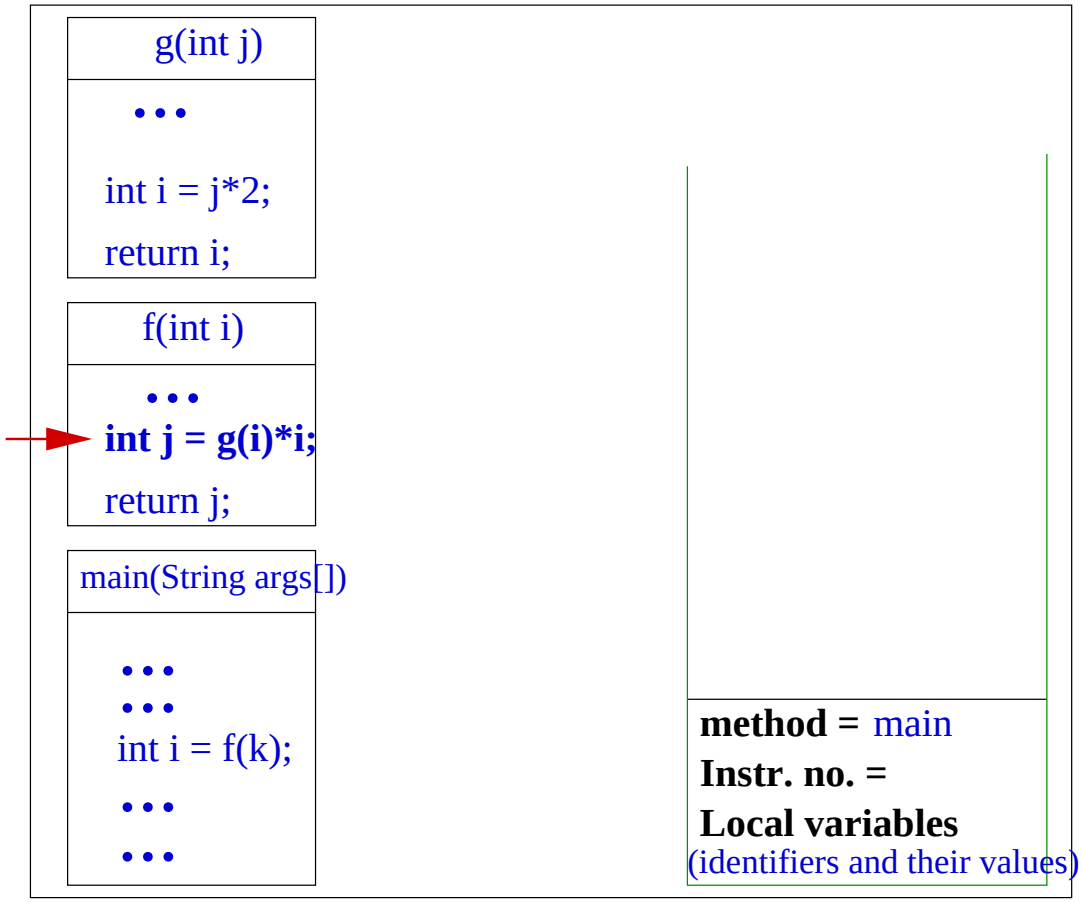
Memory

Execution of `f( )` proceeds from instr. #1



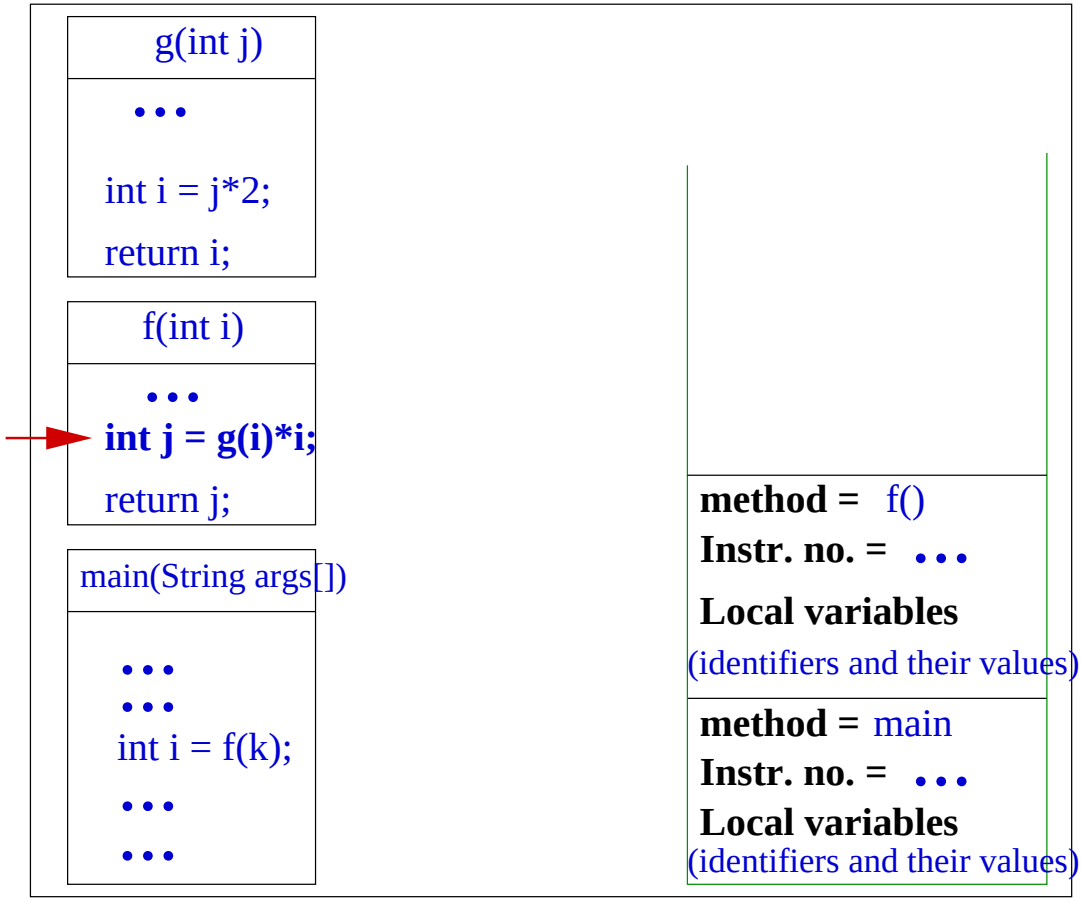
Memory

What happens on reaching `int j = g(i)*i ?`



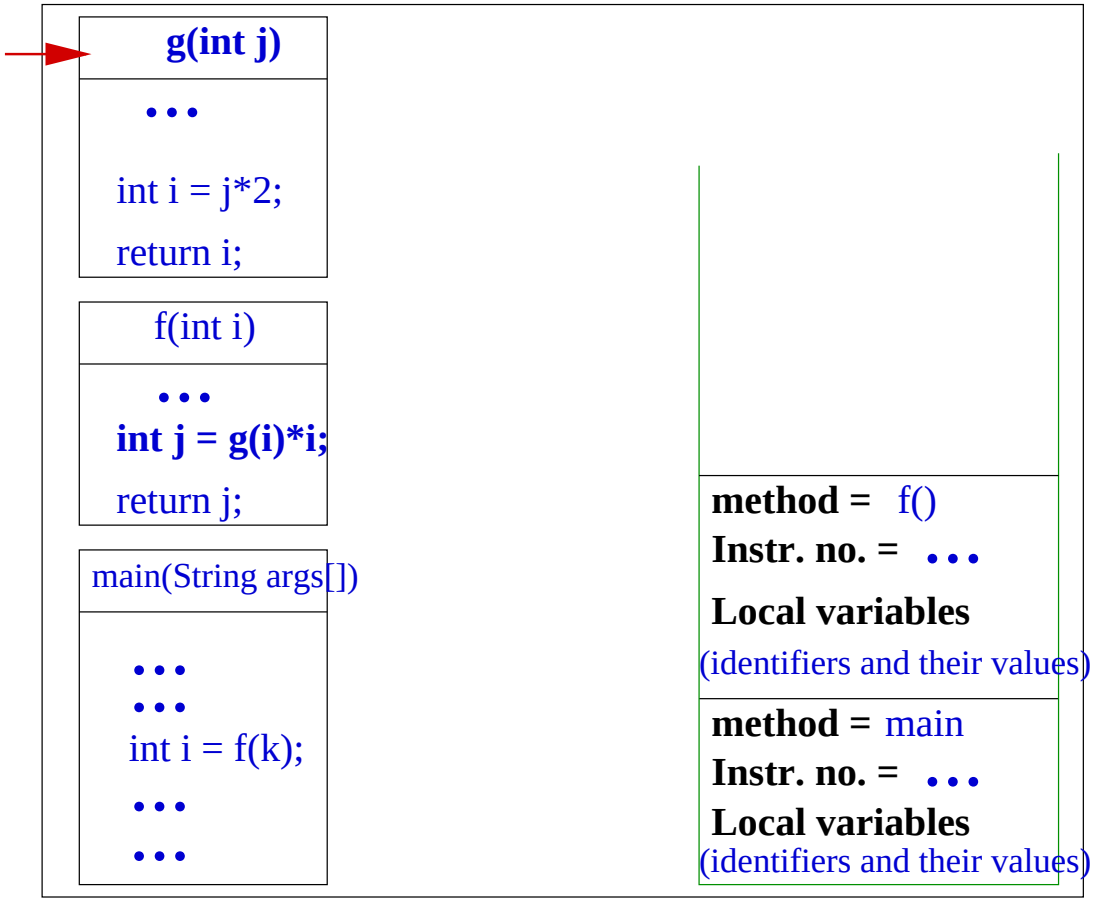
Memory

We keep few more records on the stack

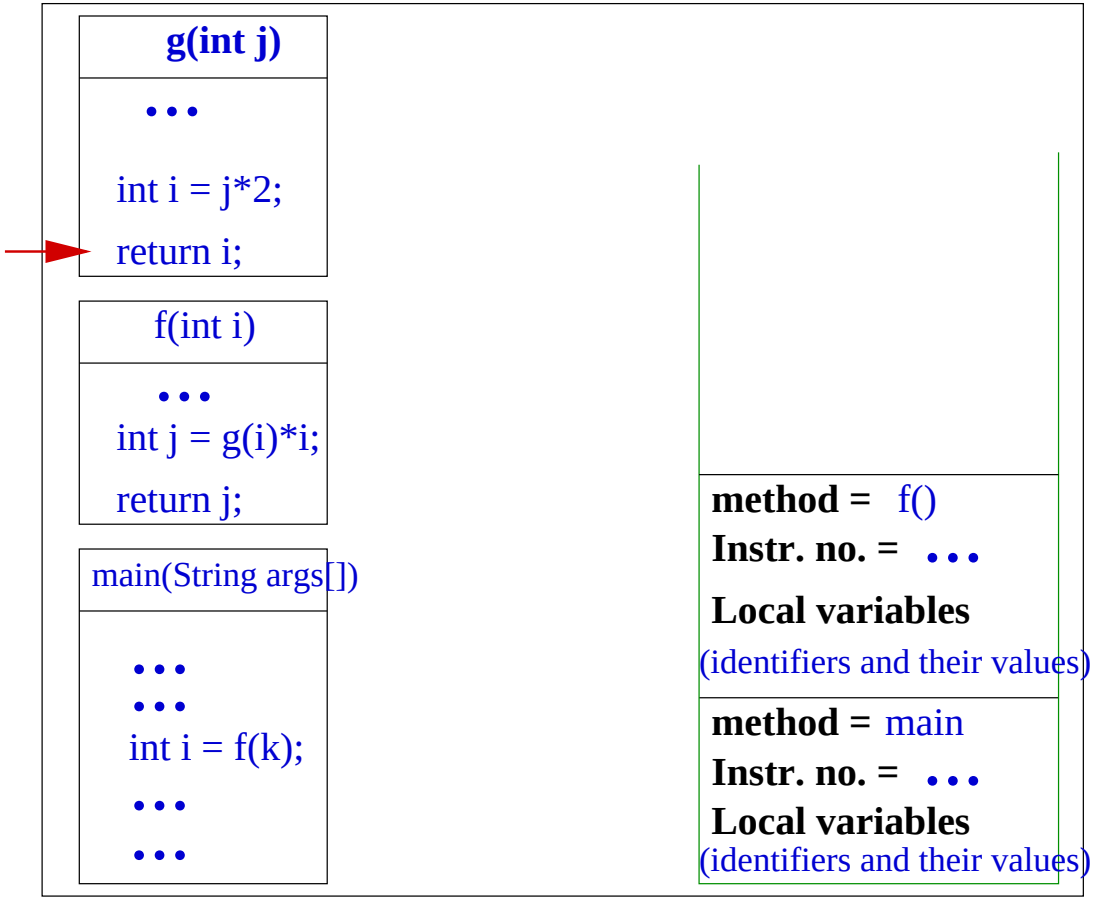


Memory

We start executing g( )

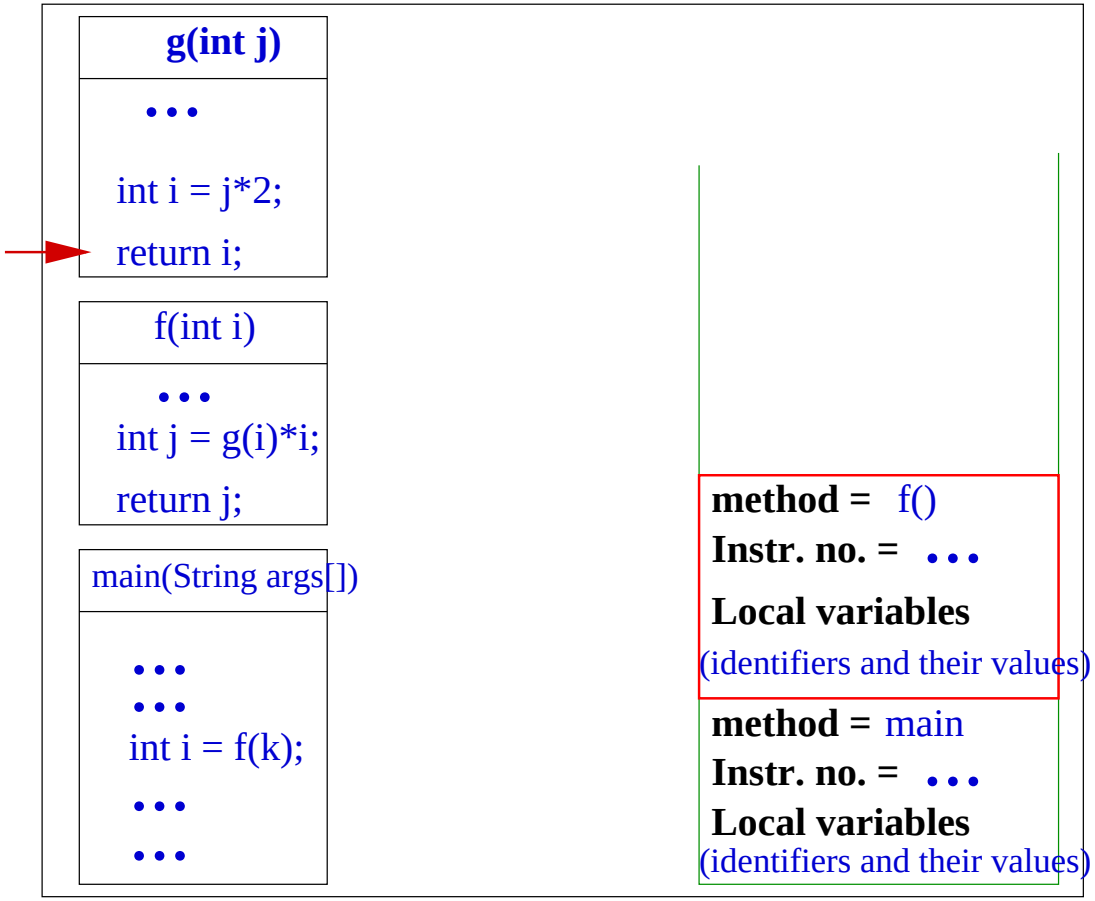


Memory


g(int j)
• • •
int i = j*2;
return i;
f(int i)
• • •
int j = g(i)*i;
return j;
main(String args[])
• • •
• • •
int i = f(k);
• • •
• • •
method = f()
Instr. no. = • • •
Local variables
(identifiers and their values)
method = main
Instr. no. = • • •
Local variables
(identifiers and their values)

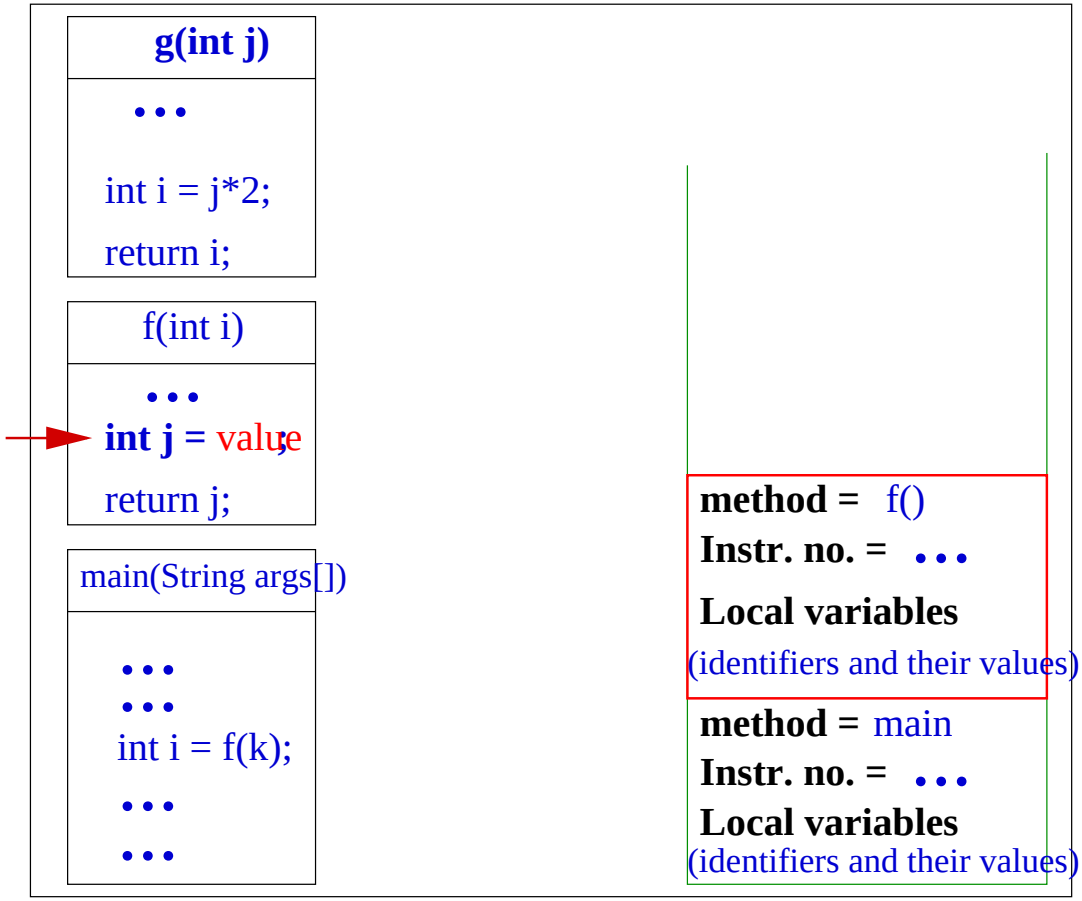
Memory

What happens when we reach end of g() ?



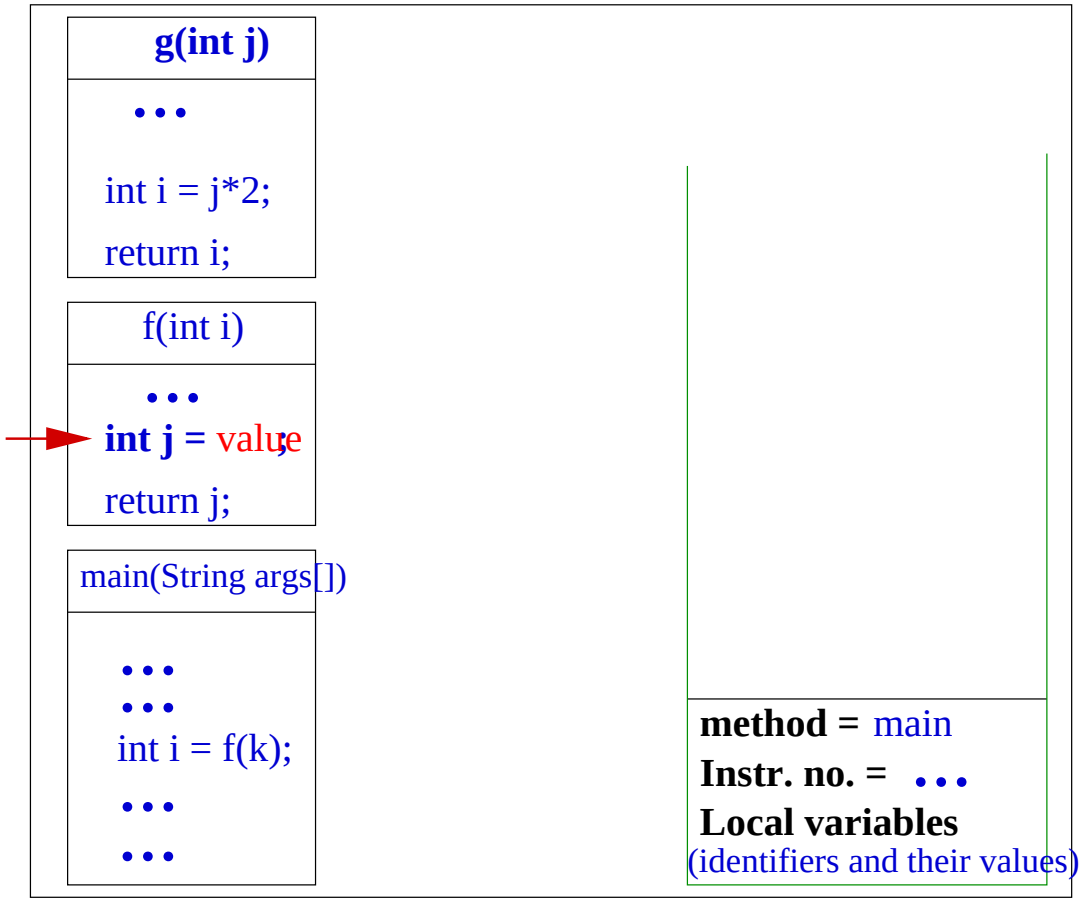
Memory

We explore the top entry of stack



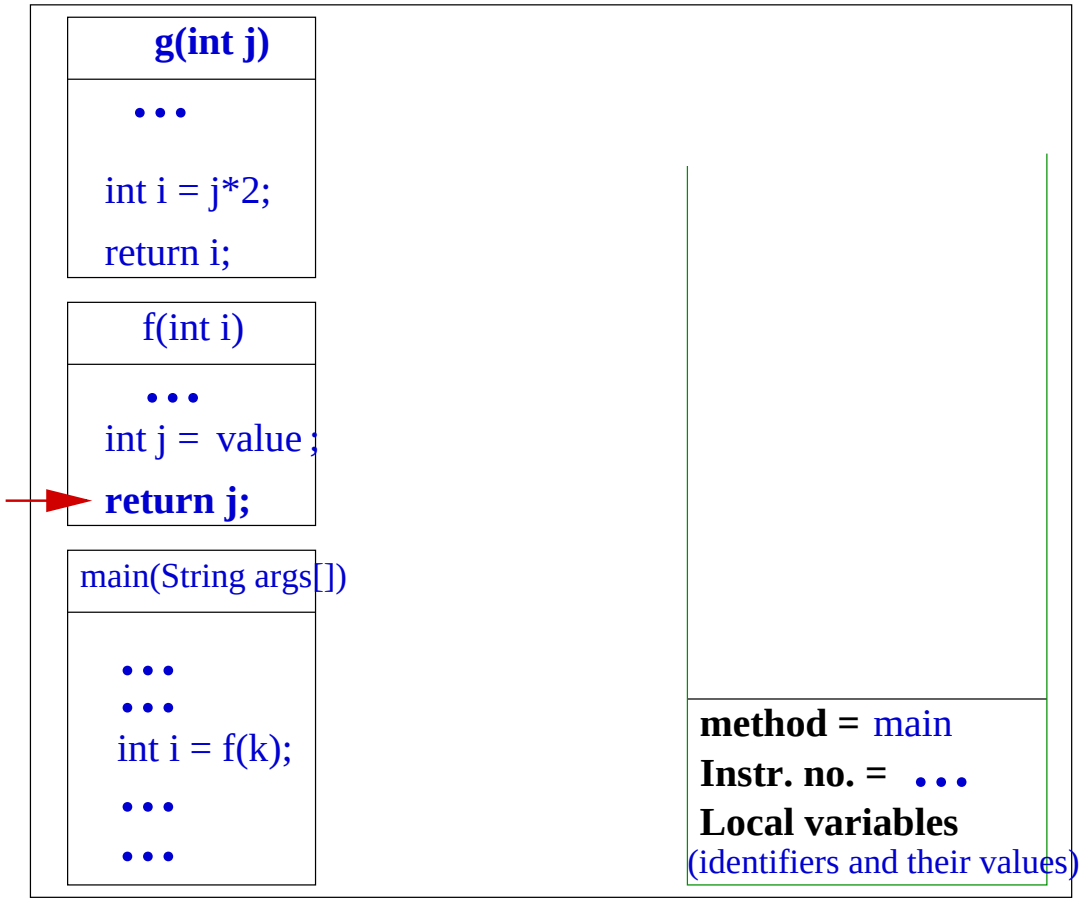
Memory

We copy the value returned by `g` to `j`



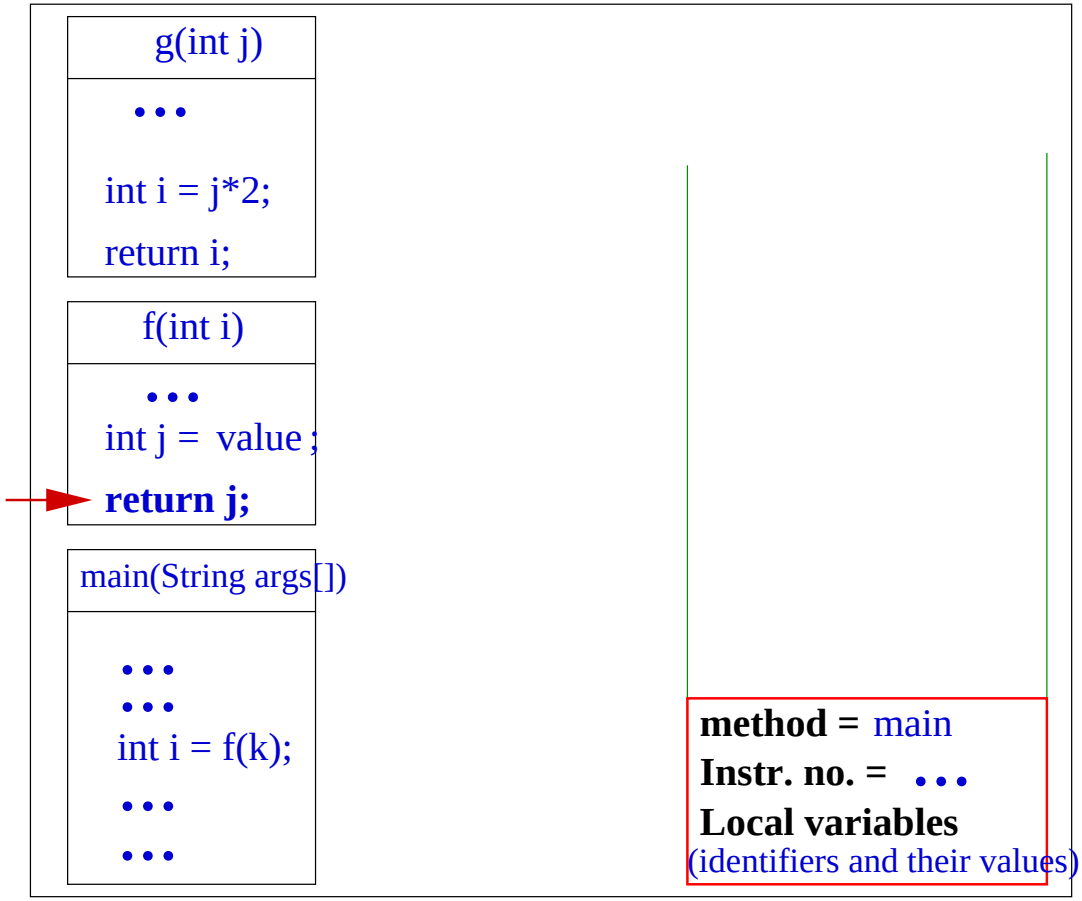
Memory

What happens when we reach the end of `f` ?



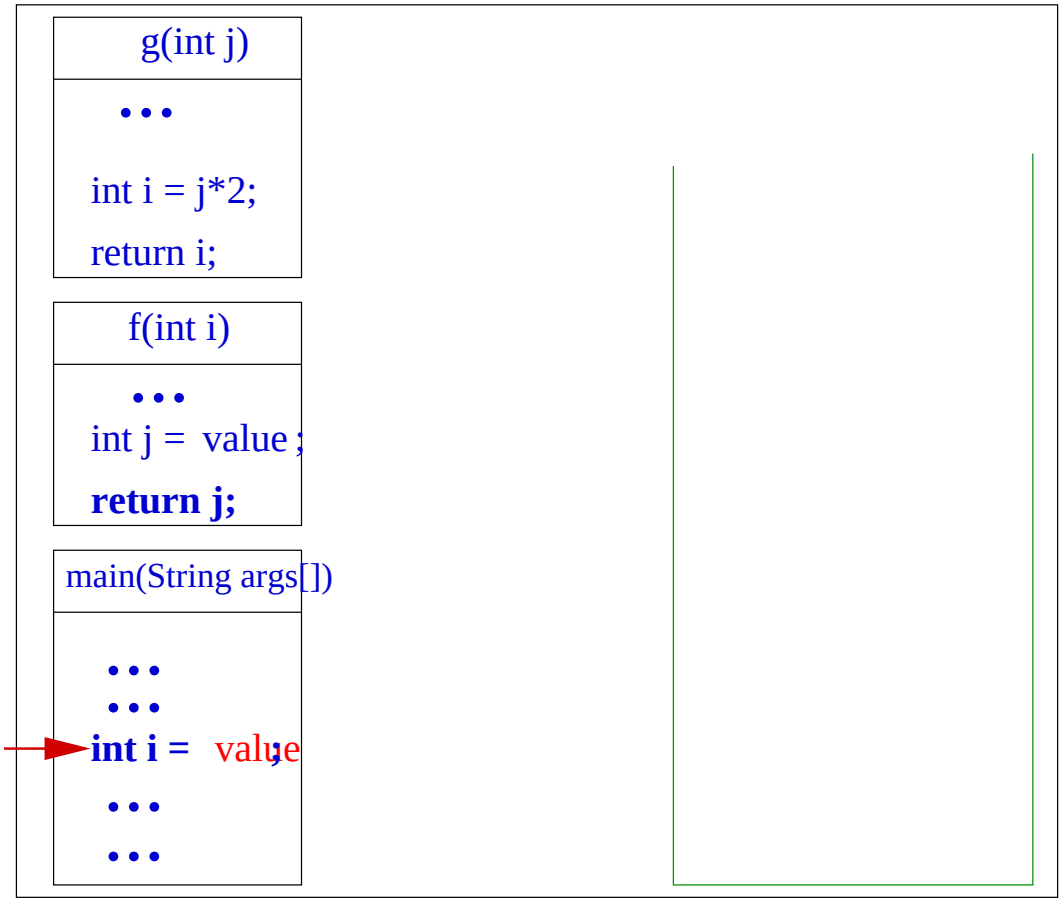
Memory

We explore the top entry of the stack



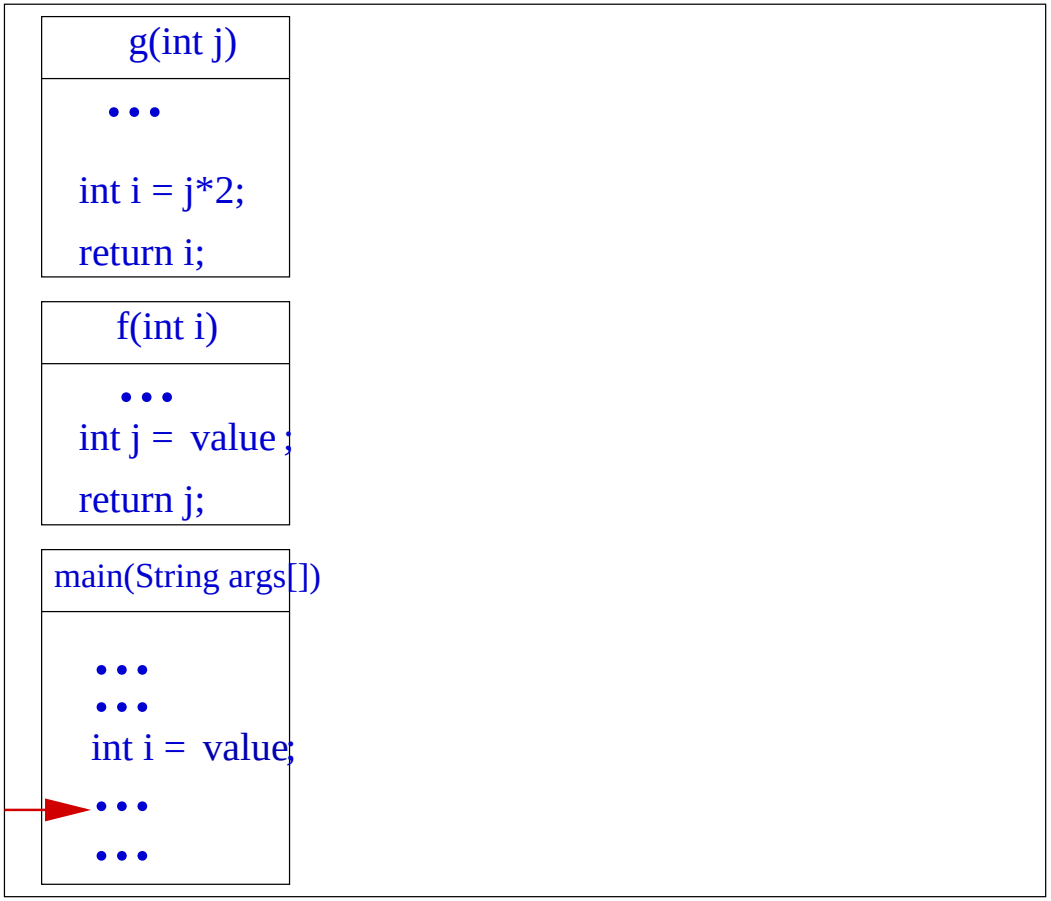
Memory

We copy the value returned by `f( )` to i



Memory

The execution proceeds as usual



Memory

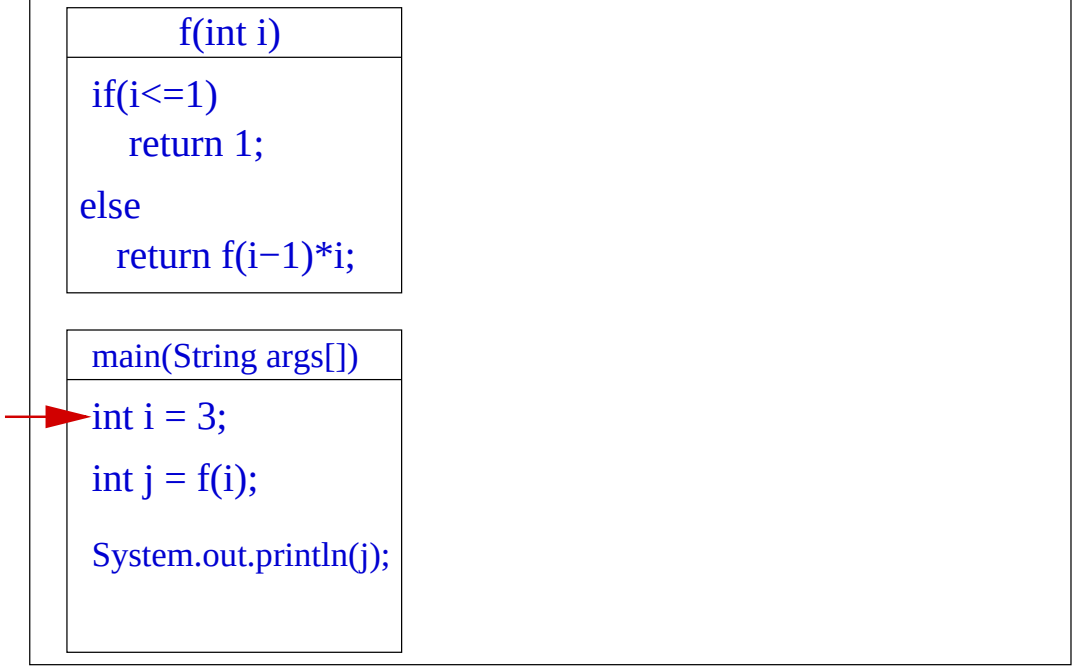
Having got the complete picture of method invocation ...

Let us consider our second program.

Everything is quite systematic now

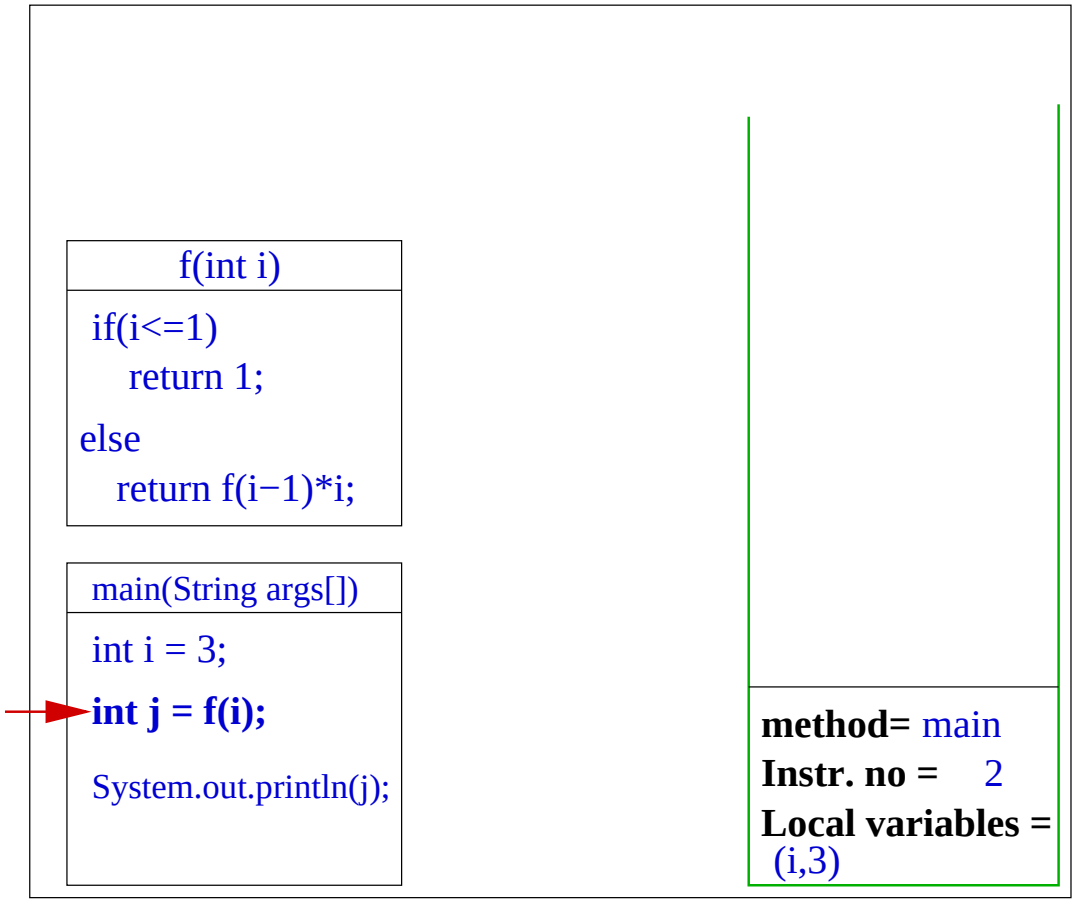
```
public static int f(int i)
{ if(i<=1)  return 1;
  else
  return f(i-1)*i;
}
public static void main(String args[])
{  int i = 3;
   int j = f(i);
   System.out.println(j);
}
```

Execution starts from Ist instruction of main()



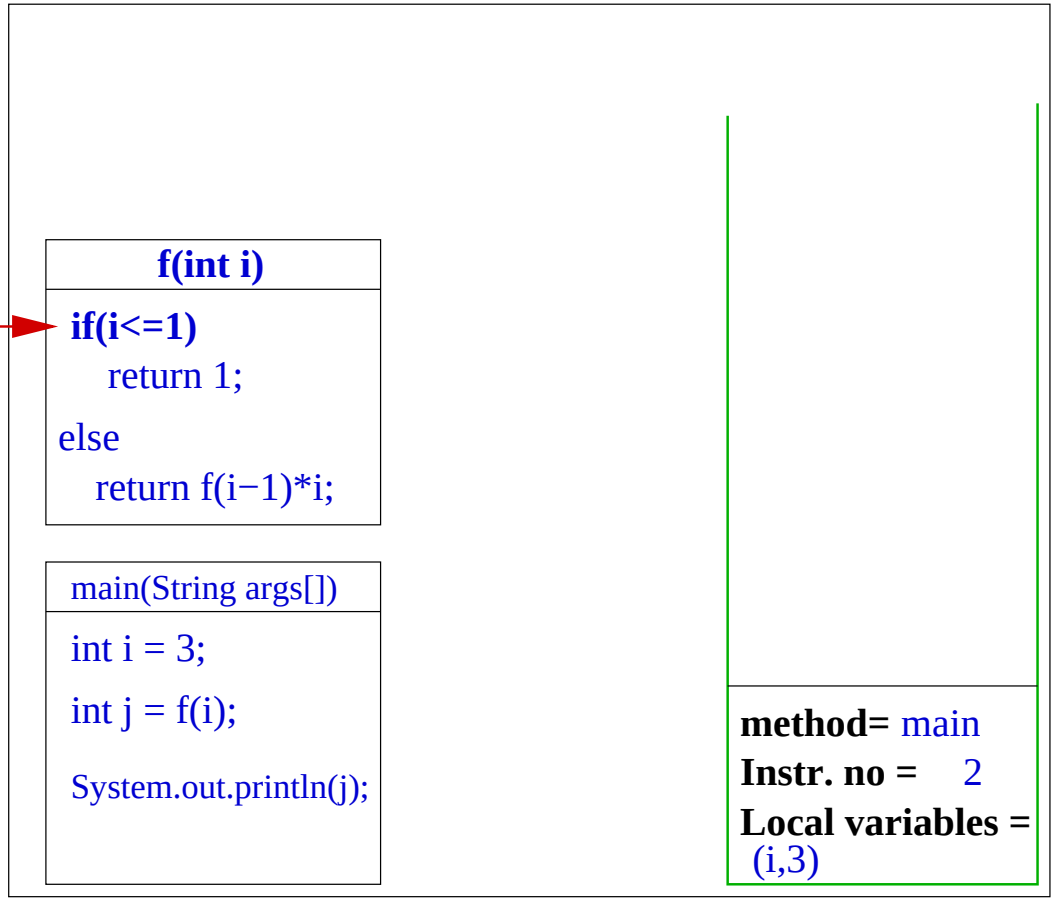
Memory

When we reach instruction `int j = f(i)`



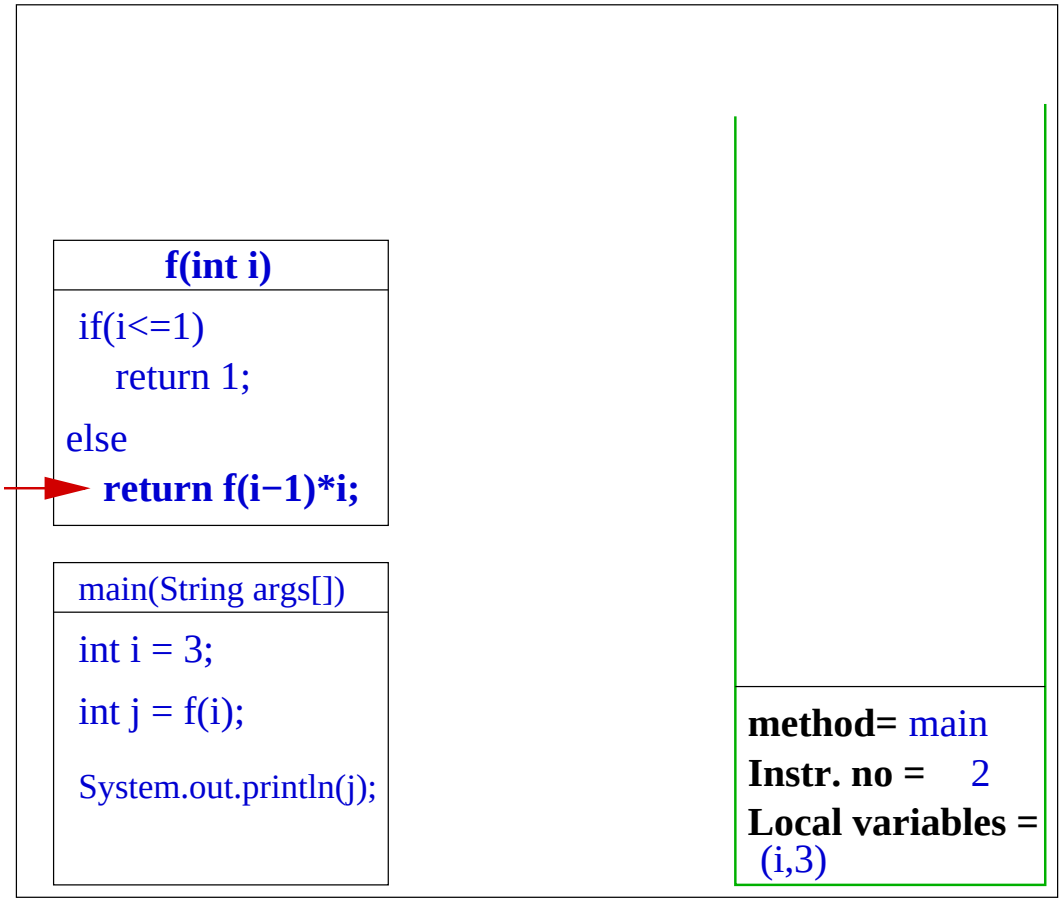
Memory

We keep record of main and invoke f(3)



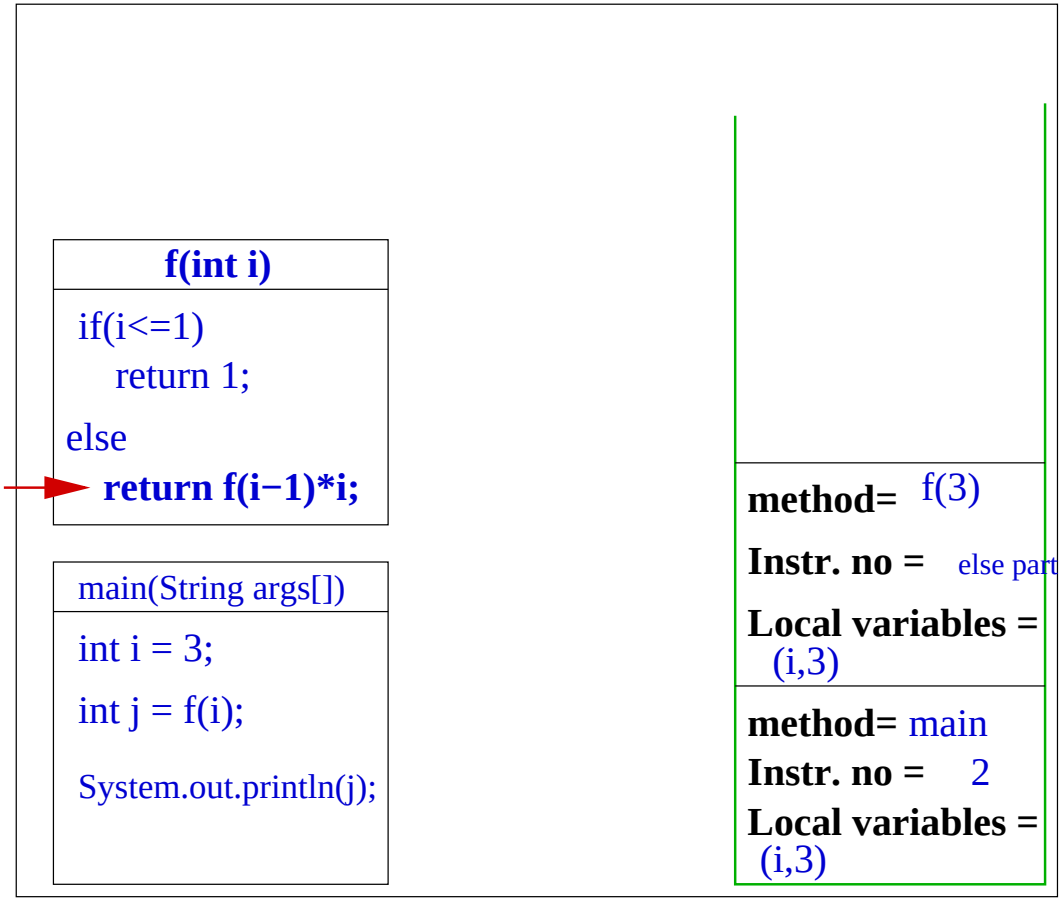
Memory

Since $i > 1$, we reach else part of Ist instr.



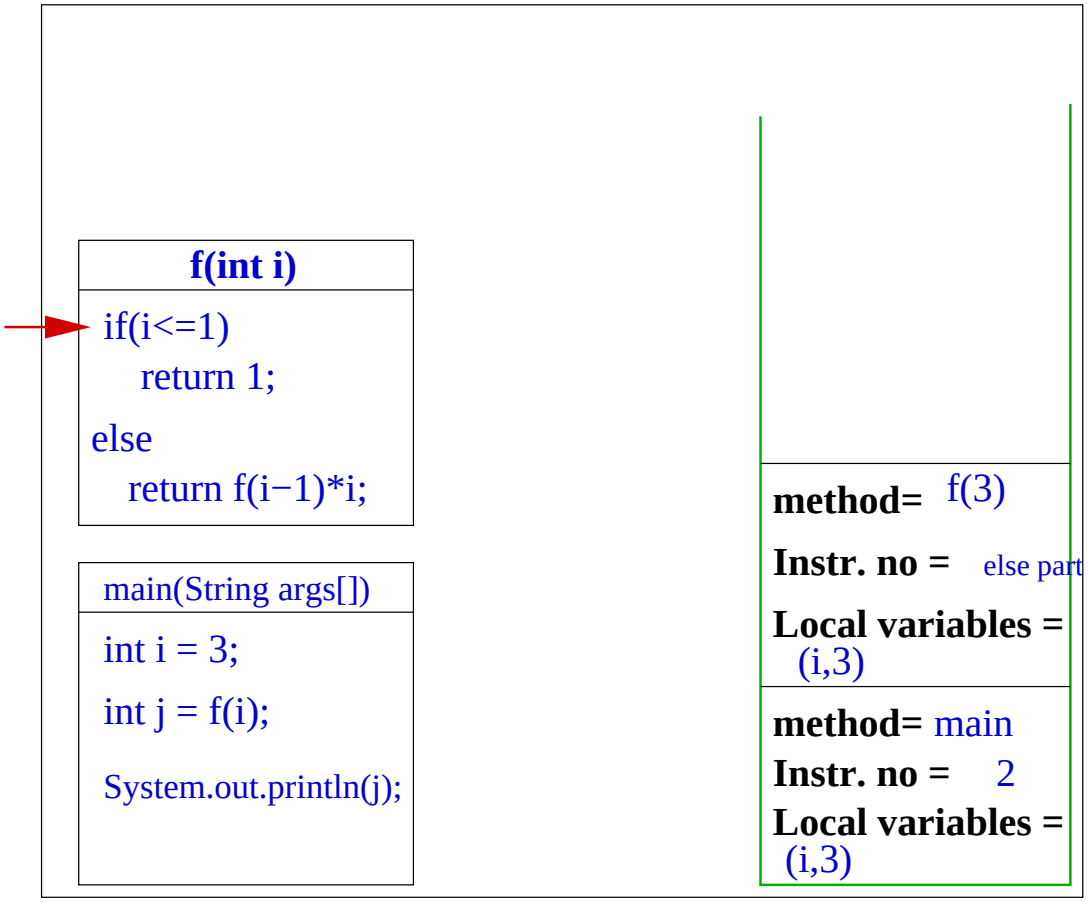
Memory

We keep record of current f(3) and invoke f(2)



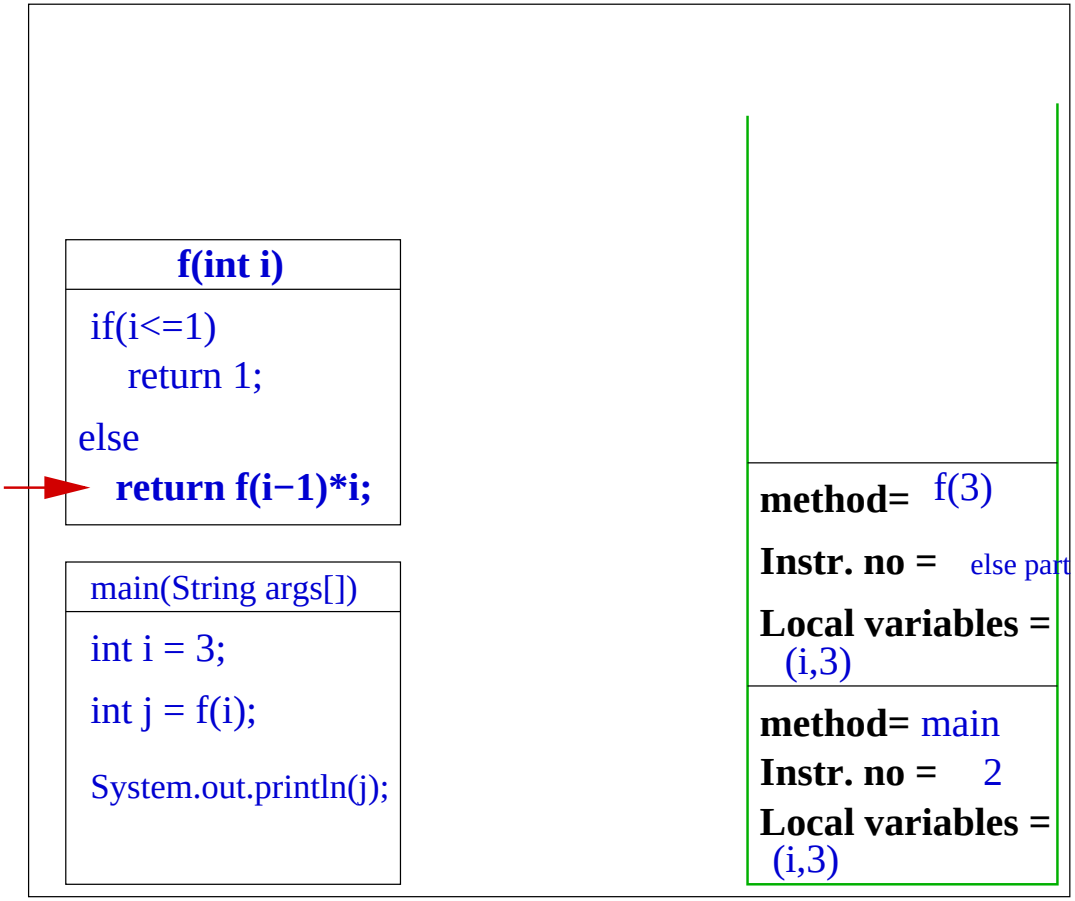
Memory

Since $i > 1$ still, we reach else part of Ist instr.



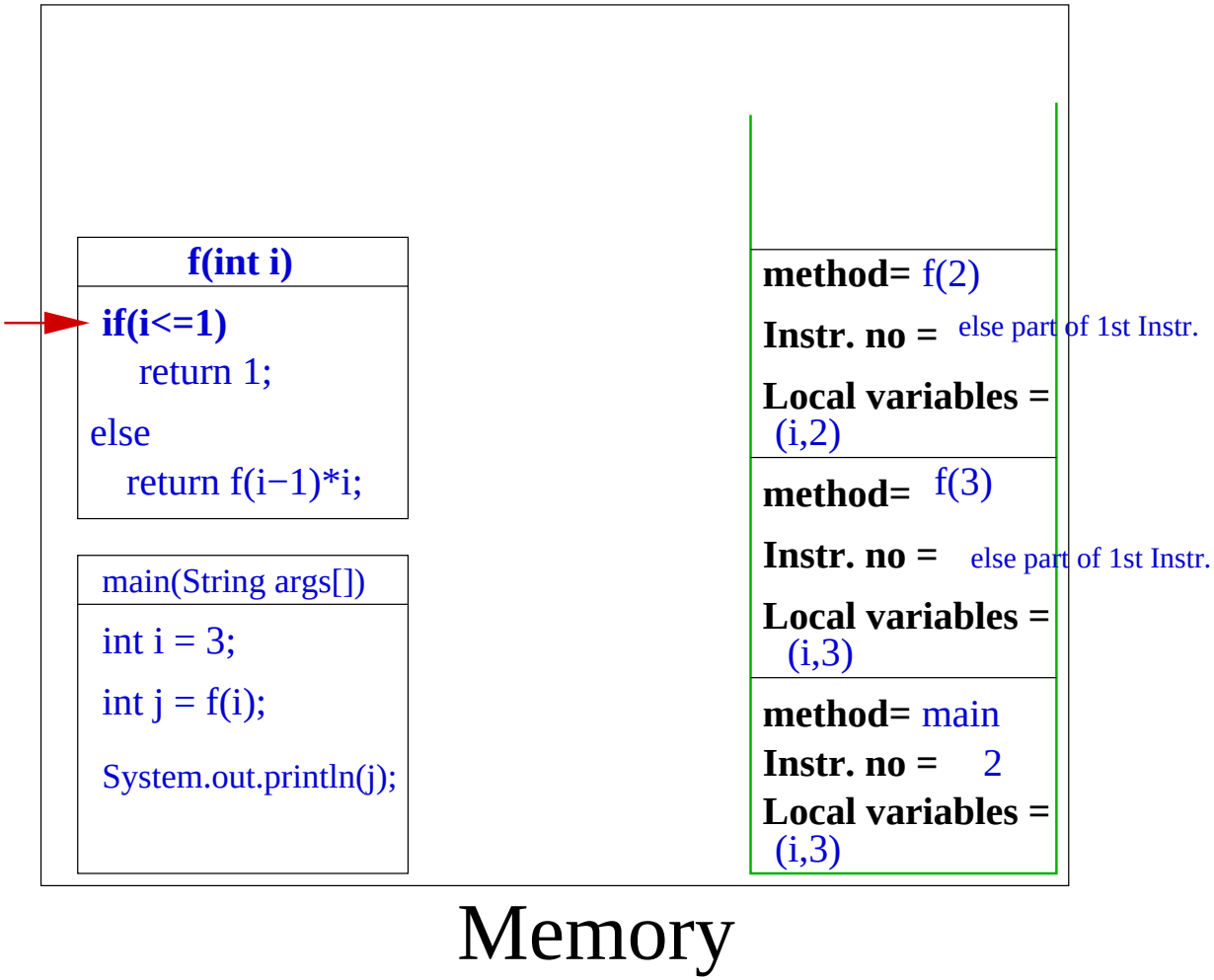
Memory

We keep record of current f(2) and invoke f(1)



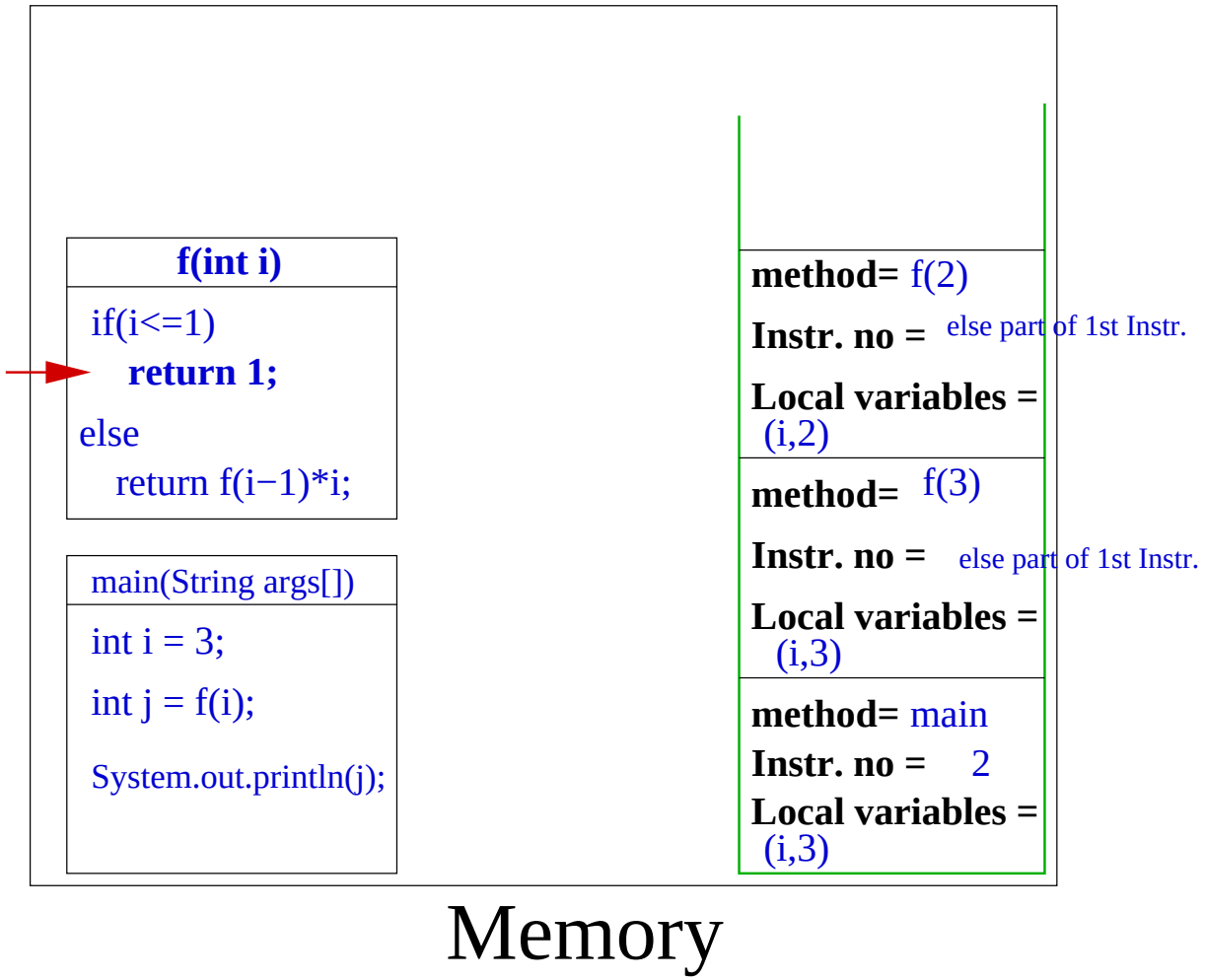
Memory

since i==1, we execute the instruction return 1



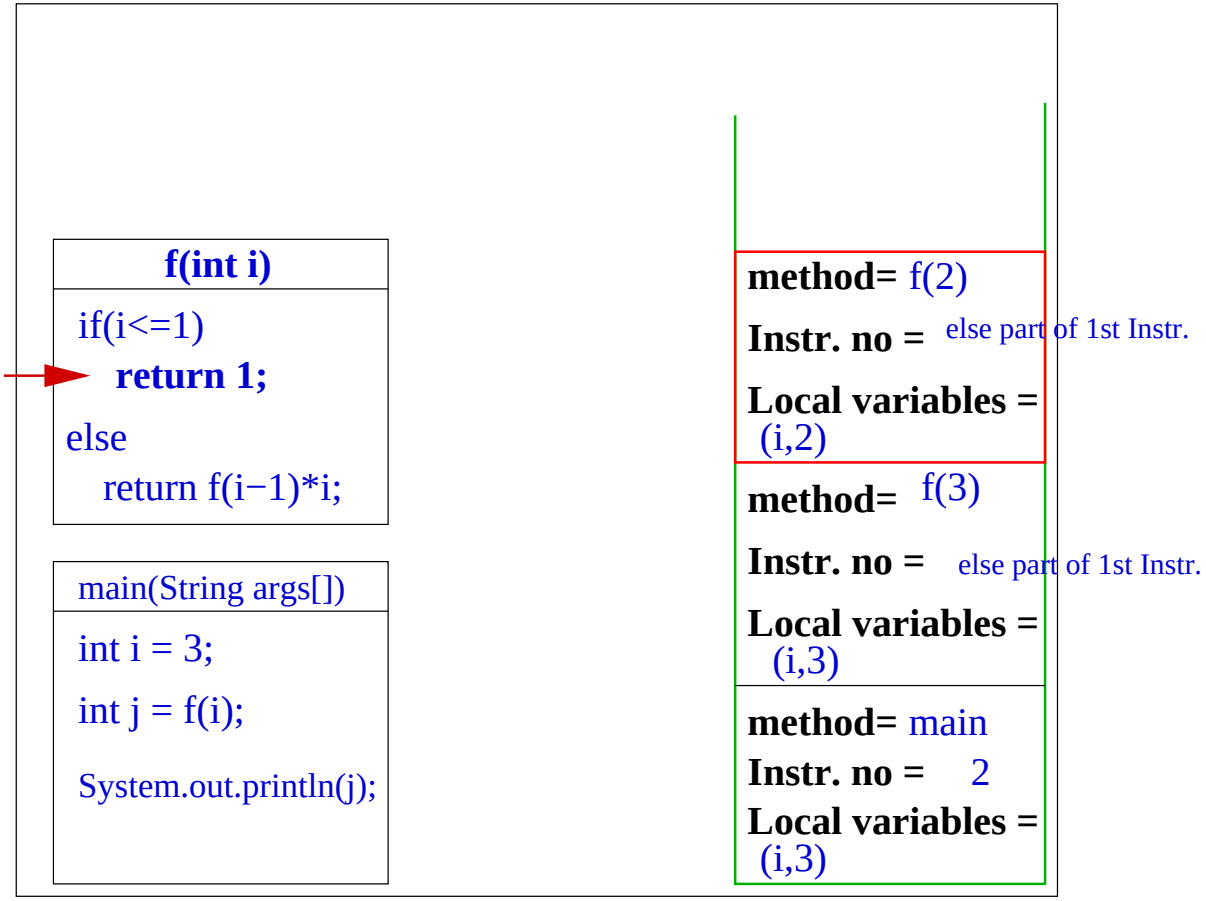
Memory

Where do we return now ?



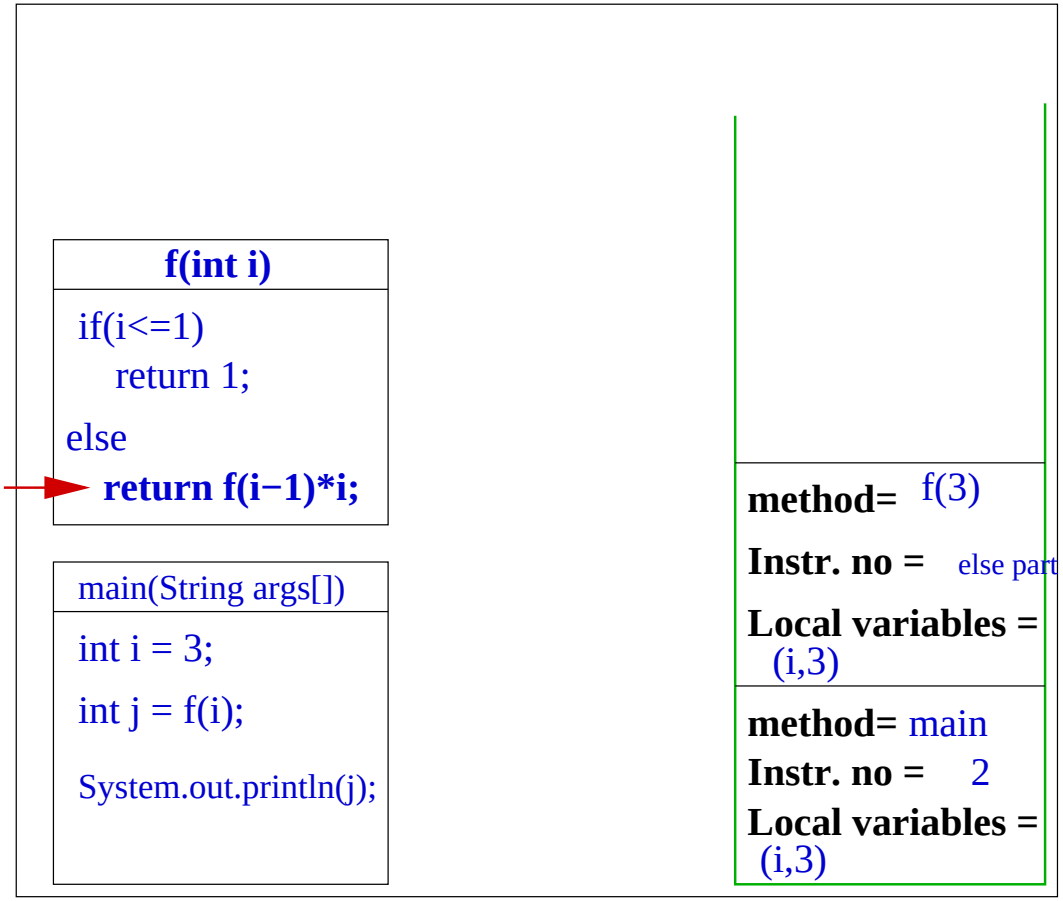
Memory

We return to the method on top of stack : f(2) but which instr. ?



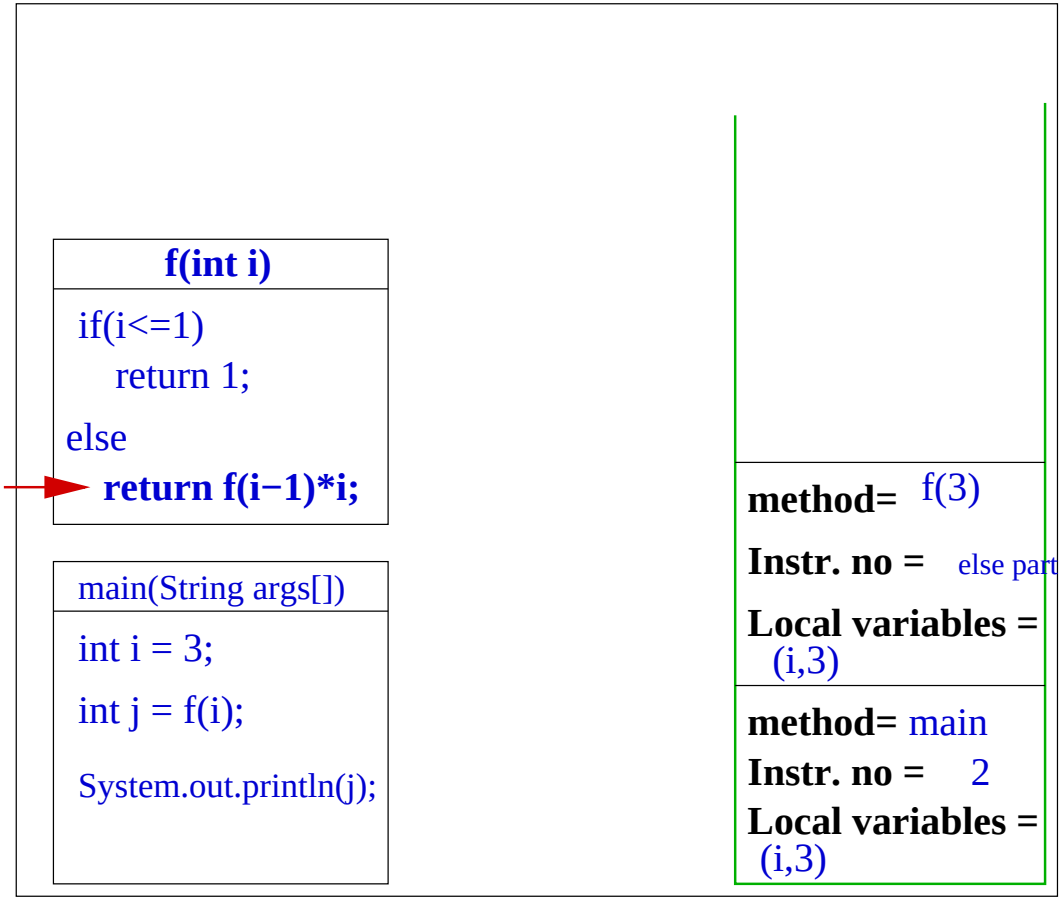
Memory

We return to the else part of the Ist instruction of f(2)



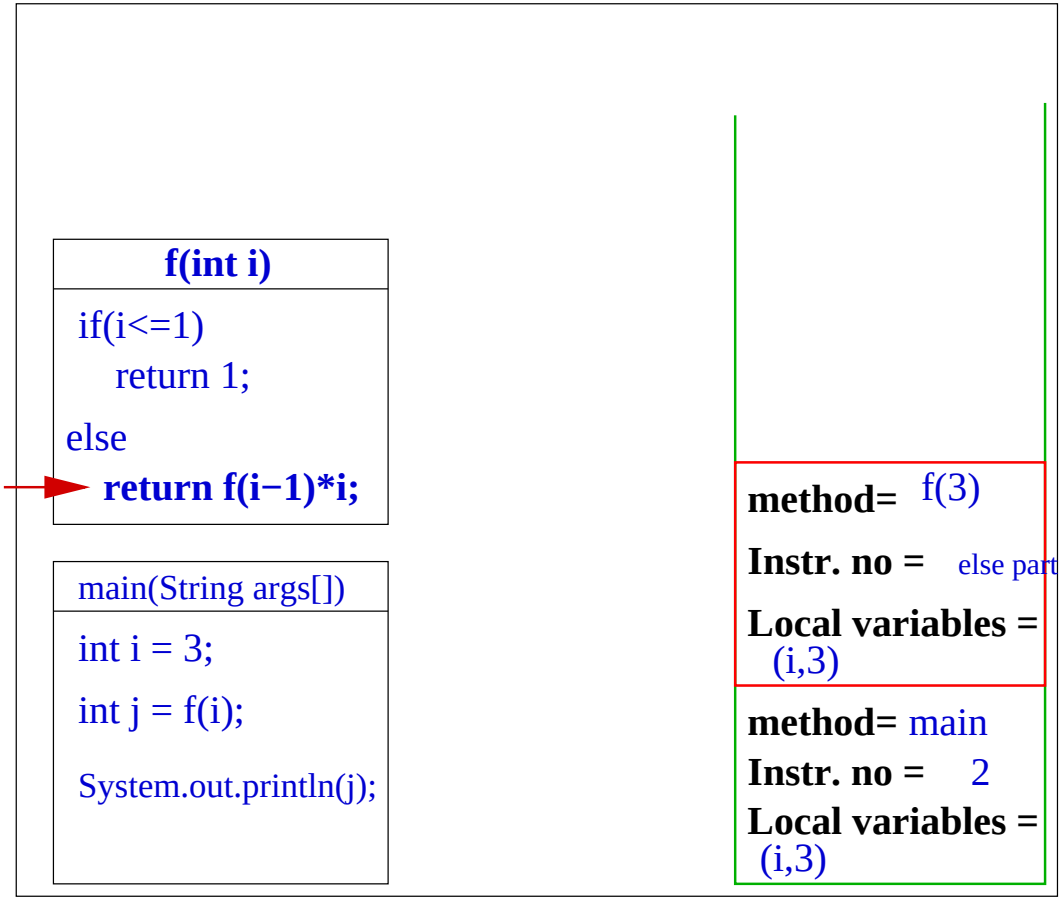
Memory

Where do we return from here ?



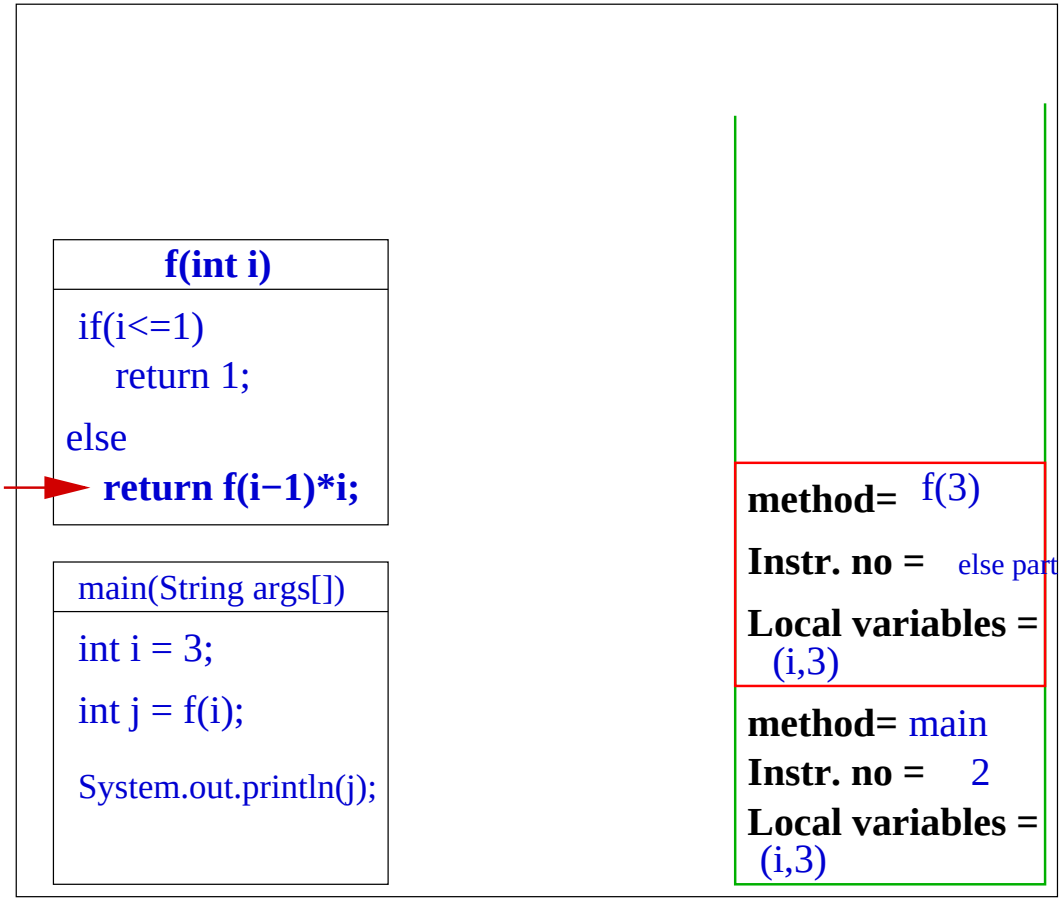
Memory

Where do we return from here ?



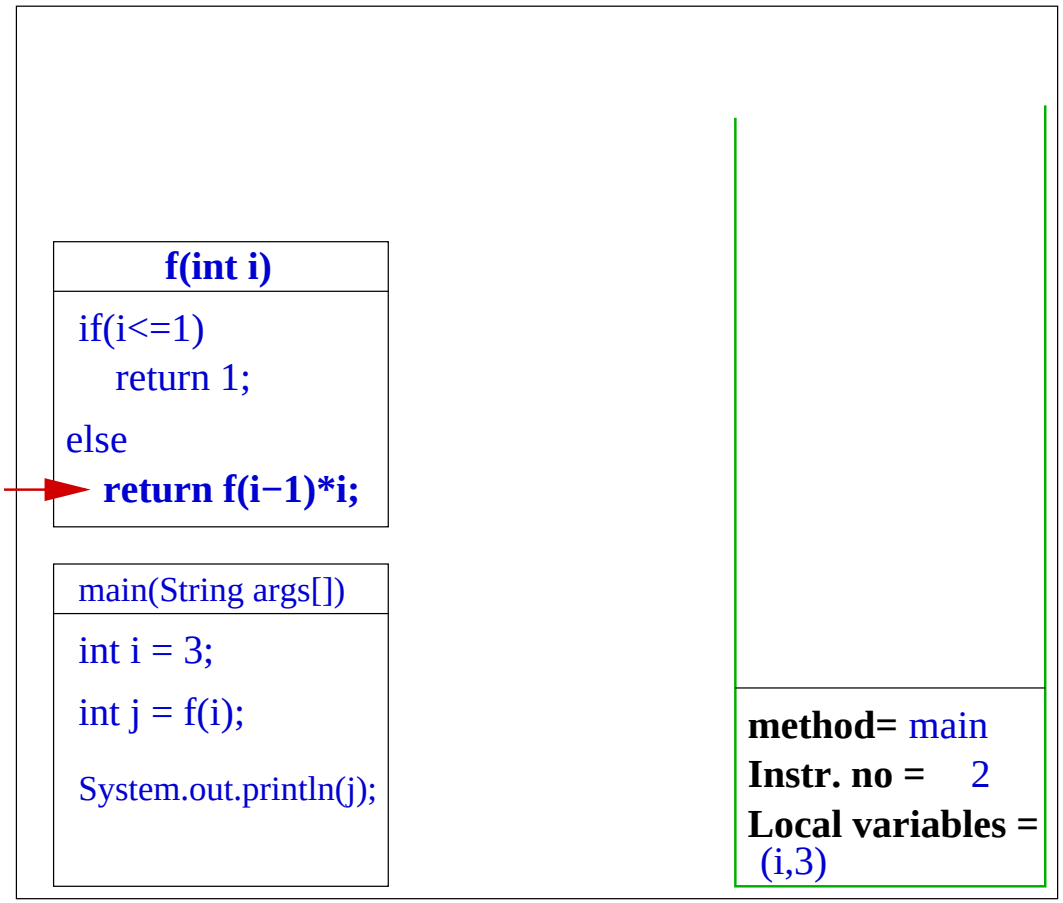
Memory

We return to the method on top of stack : f(3)



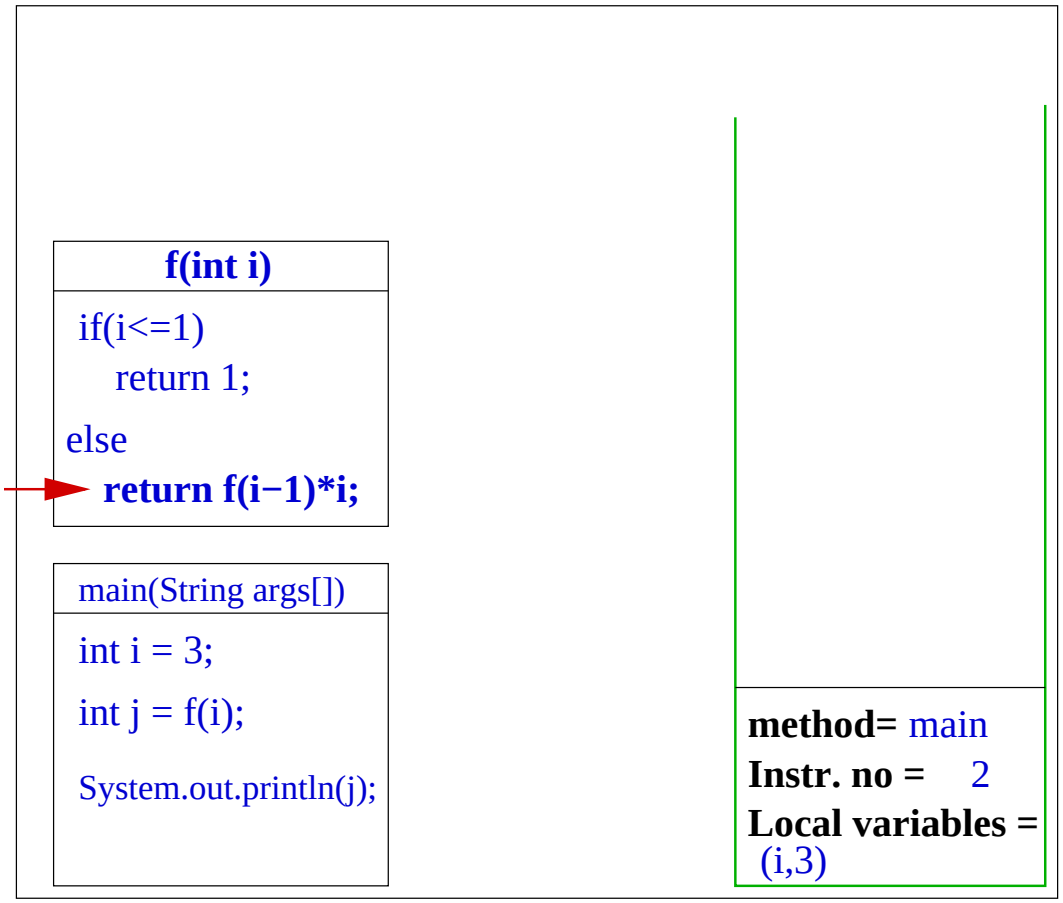
Memory

We are at else part of instruction of method :f(3)



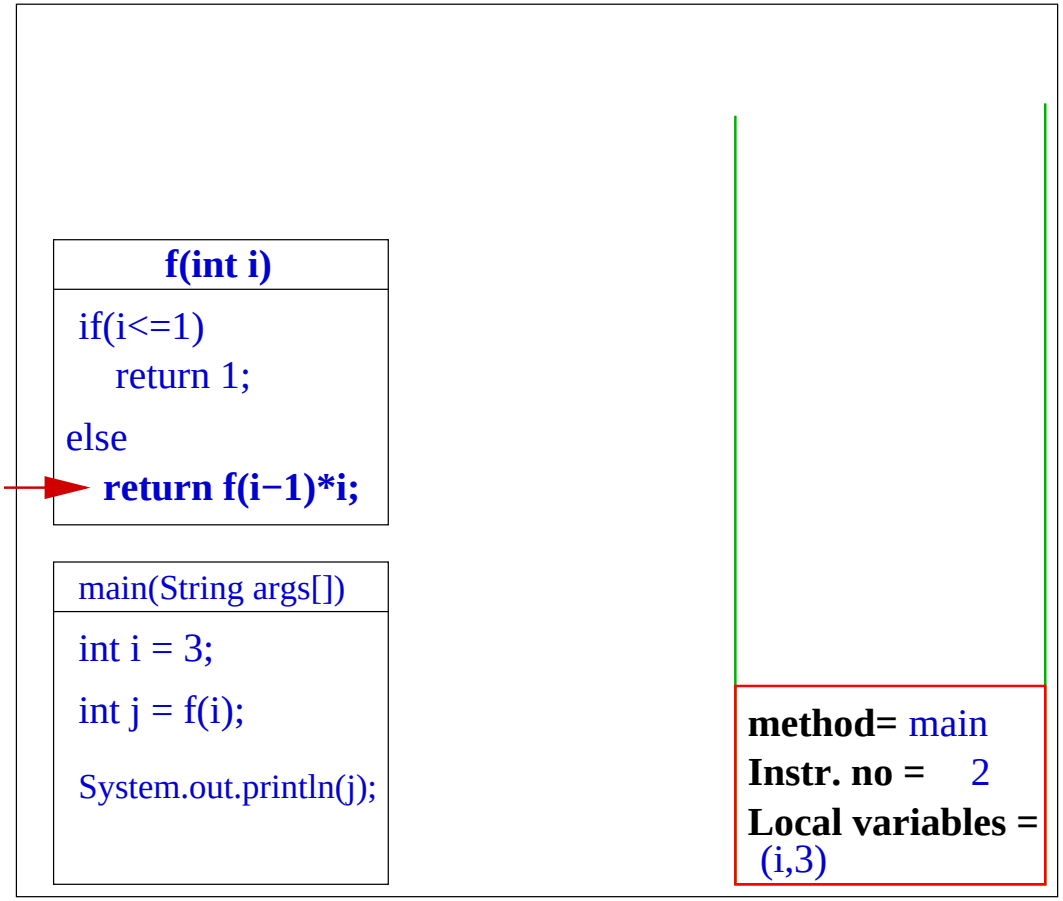
Memory

Where do we return from here ?



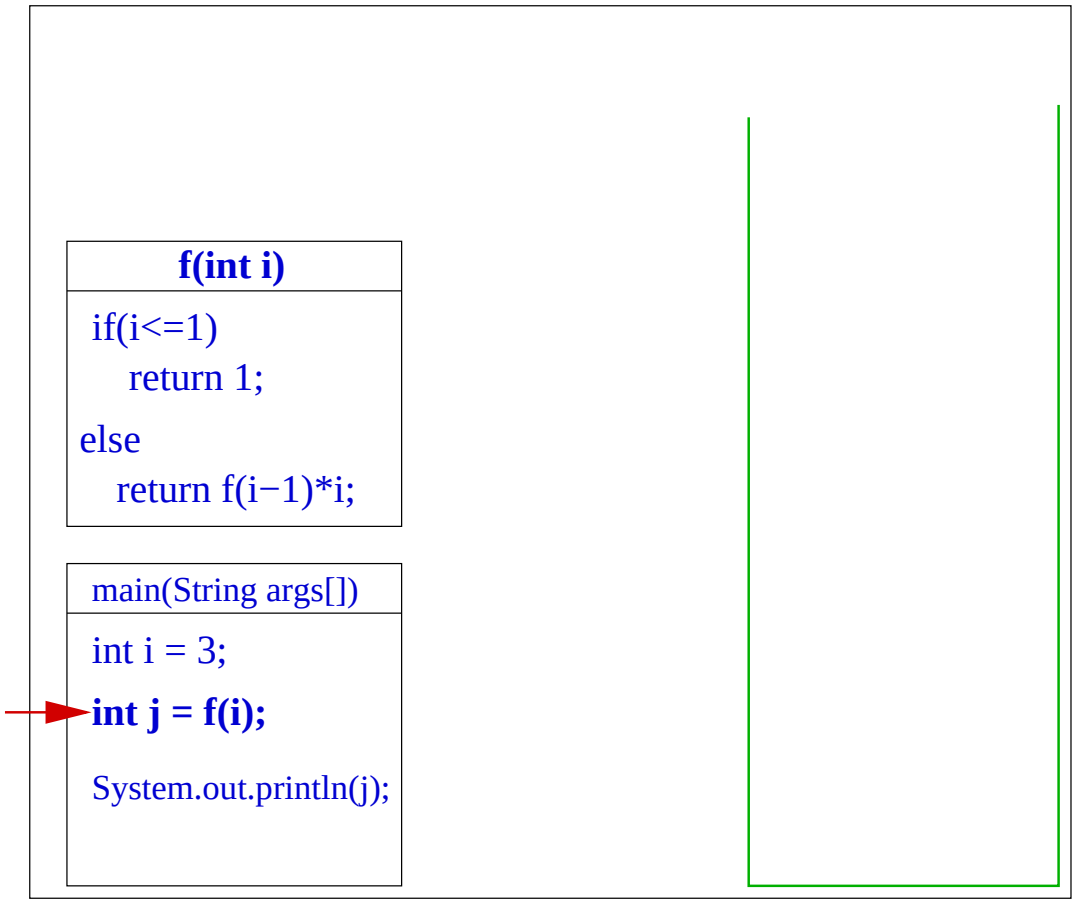
Memory

To the method on top of the stack : main()



Memory

and now the usual execution in the main



Recursion

There are many functions which can be defined inductively :

- fibonacci numbers :

$$f(0) = 0, f(1) = 1;$$

for $i > 1$ $f(i) = f(i-1) + f(i-2);$

- factorial(n) :

$$factorial(0) = 1;$$

for $i > 0$, $factorial(i) = i * factorial(i-1)$

Recursion

There are some functions whose definition itself is inductive (recursive) in nature.

- Permutation of string of n charcters
- Combinations ...
- Powerset of a set S : the set of all subsets of a the set S

Recursion

There are some functions whose definition itself is inductive (recursive) in nature.

- Permutation of string of n charcters
- Combinations ...
- Powerset of a set S : the set of all subsets of a the set S

They have a very simple recursive solution