Java Collection Framework

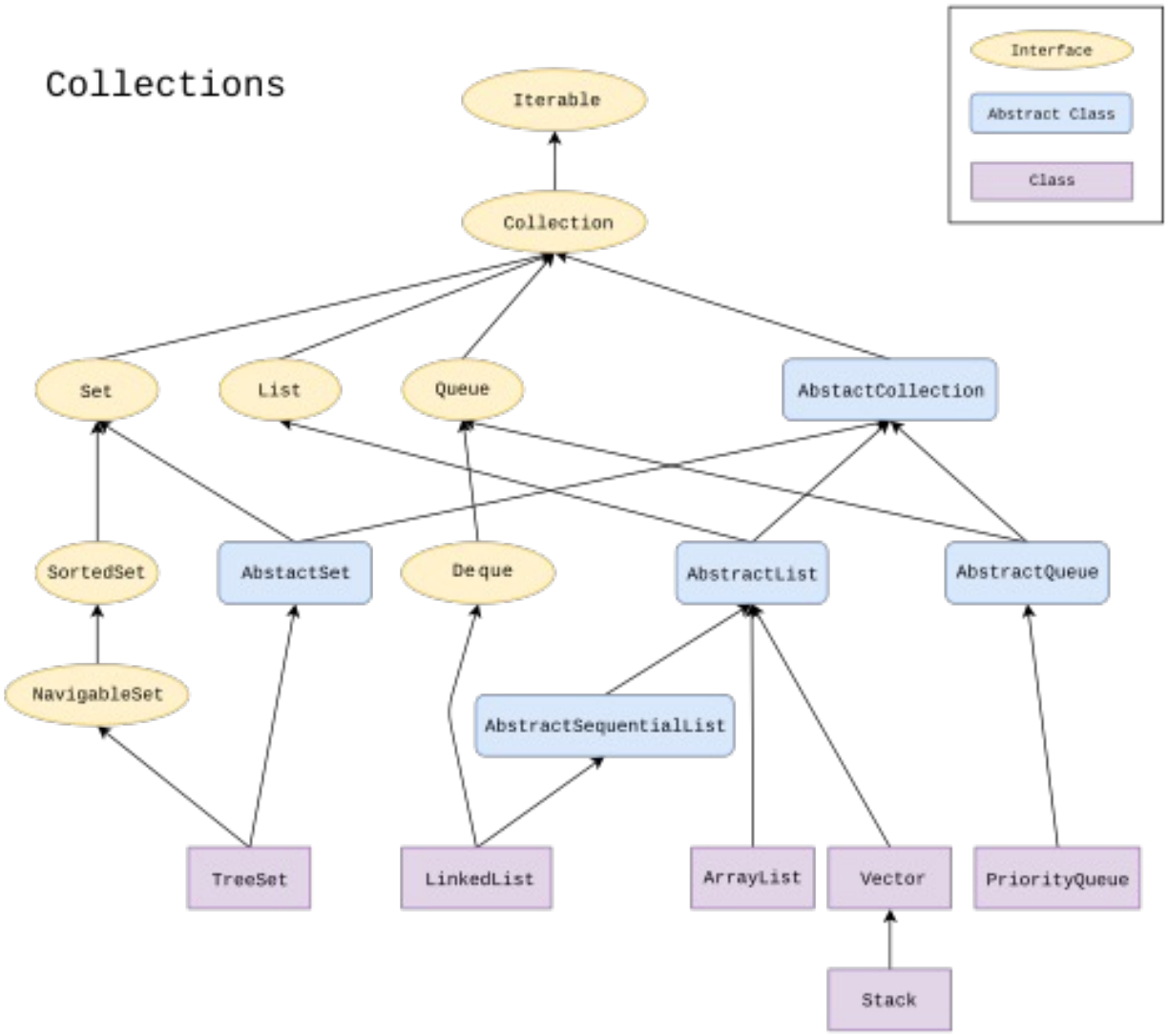


Image from https://en.wikipedia.org/wiki/Java_collections_framework, accessed September 18, 2018

Iterators

- A Java interface class to process each element of a Collection
- Methods:
  - hasNext()
  - next()
  - remove() (not always present)
  - forEachRemaining(Consumer action) (not always present)
- Specific iterators may have additional methods
  - Eg. ListIterator has hasPrevious() and previous()
- Order isn't usually guaranteed
  - Though order is usually predictable

Iterators – typical use

```
ArrayList<String> collection;
…
Iterator<String> iterate = collection.iterator();

while (iterate.hasNext()) {
        String data = iterate.next();
        // do something with ”data”
}
…
for ( Iterator<String> it2 = collection.iterator(); it2.hasNext(); ){
        String data2 = it2.next();
        // do something with “data2”
}
…
for( String data3 : collection ) {   // like an implicit iterator
        // do something with “data 3”
}
```

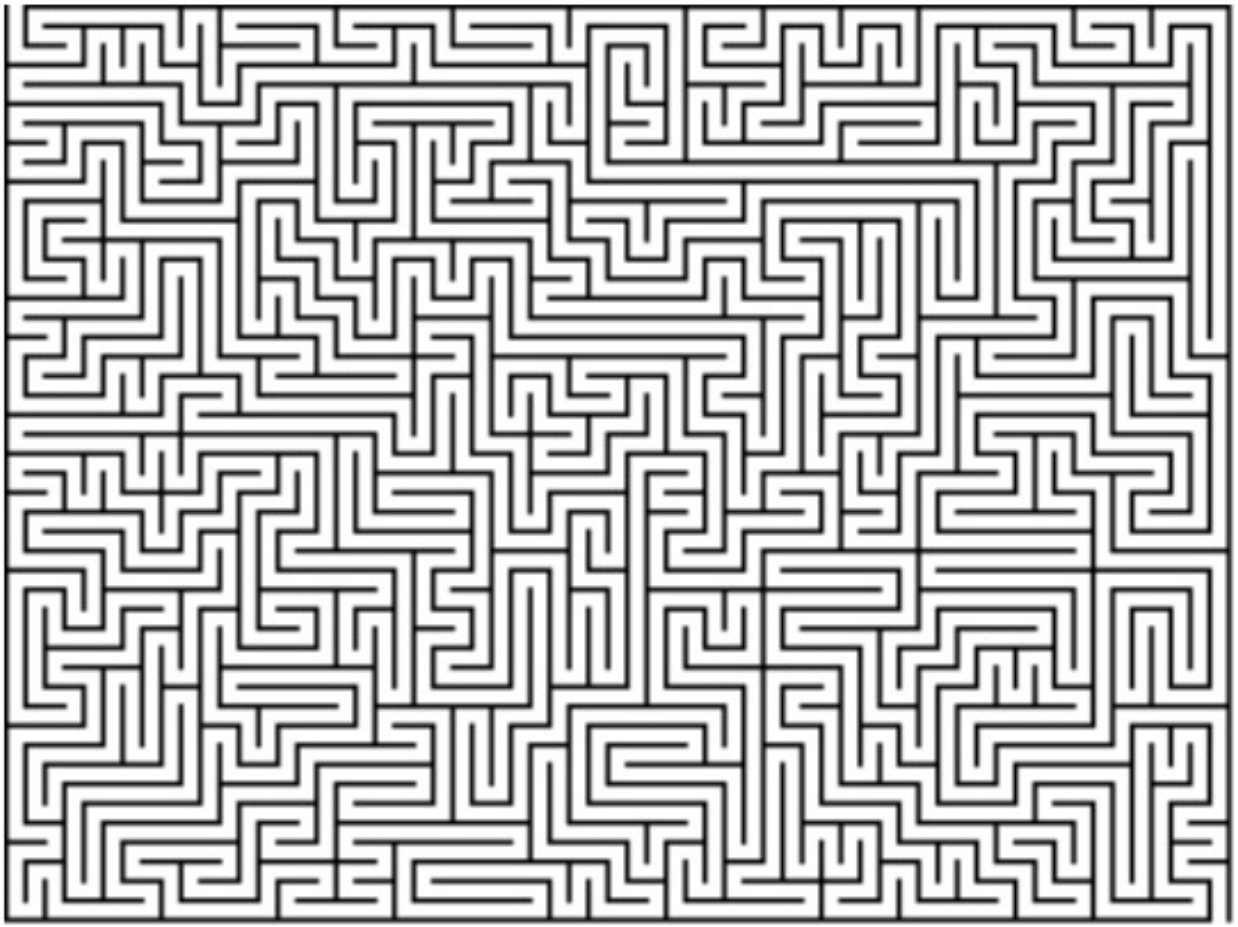
Why use iterators?

- Isolates your code from the implementation of an ADT
- Uses a common practice that others will recognize

- Limitations
  - Don't want your data structure changing under you at the same time
  - Generally uni-directional

Problem solving

- How would you approach getting a program to solve a maze?



DALHOUSIE UNIVERSITY
Inspiring Minds

Problem solving

- Wander randomly and remember your path
  - Can wander over the same space
  - Clean up the path at the end
- Use a rule to explore deterministically
  - A rule must exist that will succeed
- Send people off in parallel to each take their own path and report back
  - Needs coordination among all of the people
- Backtrack when you meet a deadend
  - Must remember past decisions so that you can change them when needed

Problem solving - recursion

- Applicable when you have a problem to solve that can be decomposed into smaller problems of the same type
  - Needs a simplest case where it can stop
  - May need to remember some past data
  - May need to recombine some of the work from the smaller problems after they are solved.
- Programming parallel to mathematical induction
- Examples?

Recursion

- Appears in programs as a method that calls itself with a smaller instance of data

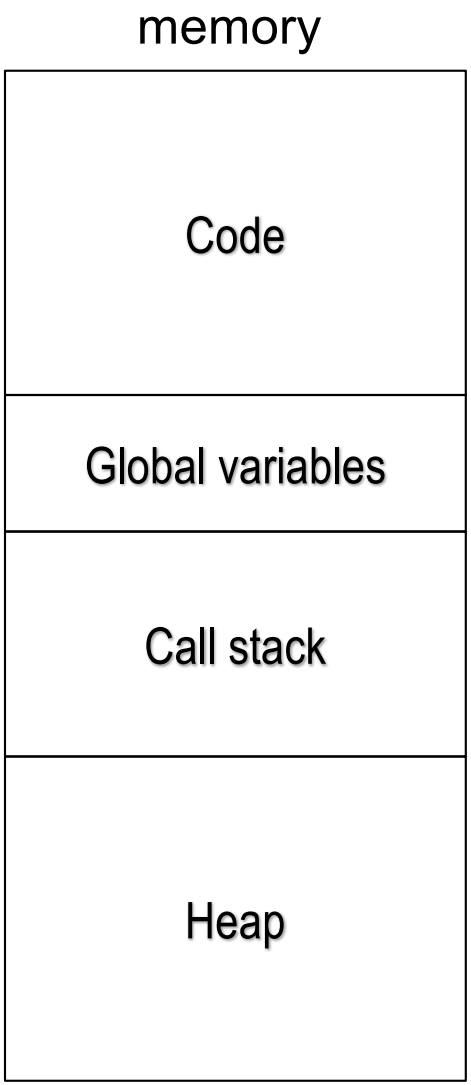
- Canonical example: Fibonacci numbers

```
int fib (int n) {
        if (n<= 0) {
                return 0;
        } else if (n<=2) {
                return 1;
        } else {
                return fib(n-1)+fib(n-2)
        }
}
```

Recursion

- The calls “remember” the past data in the call stack
- Recursive algorithms can also be solved iteratively
  - With iteration:
    - You are responsible for “remembering” all of the past data
    - You are responsible for knowing where to keep going if you backtrack

Anatomy of a process



Your compiled program

Global variable, static variables

Local variables, history of method calls

Objects created with “new”

DALHOUSIE UNIVERSITY
Inspiring Minds

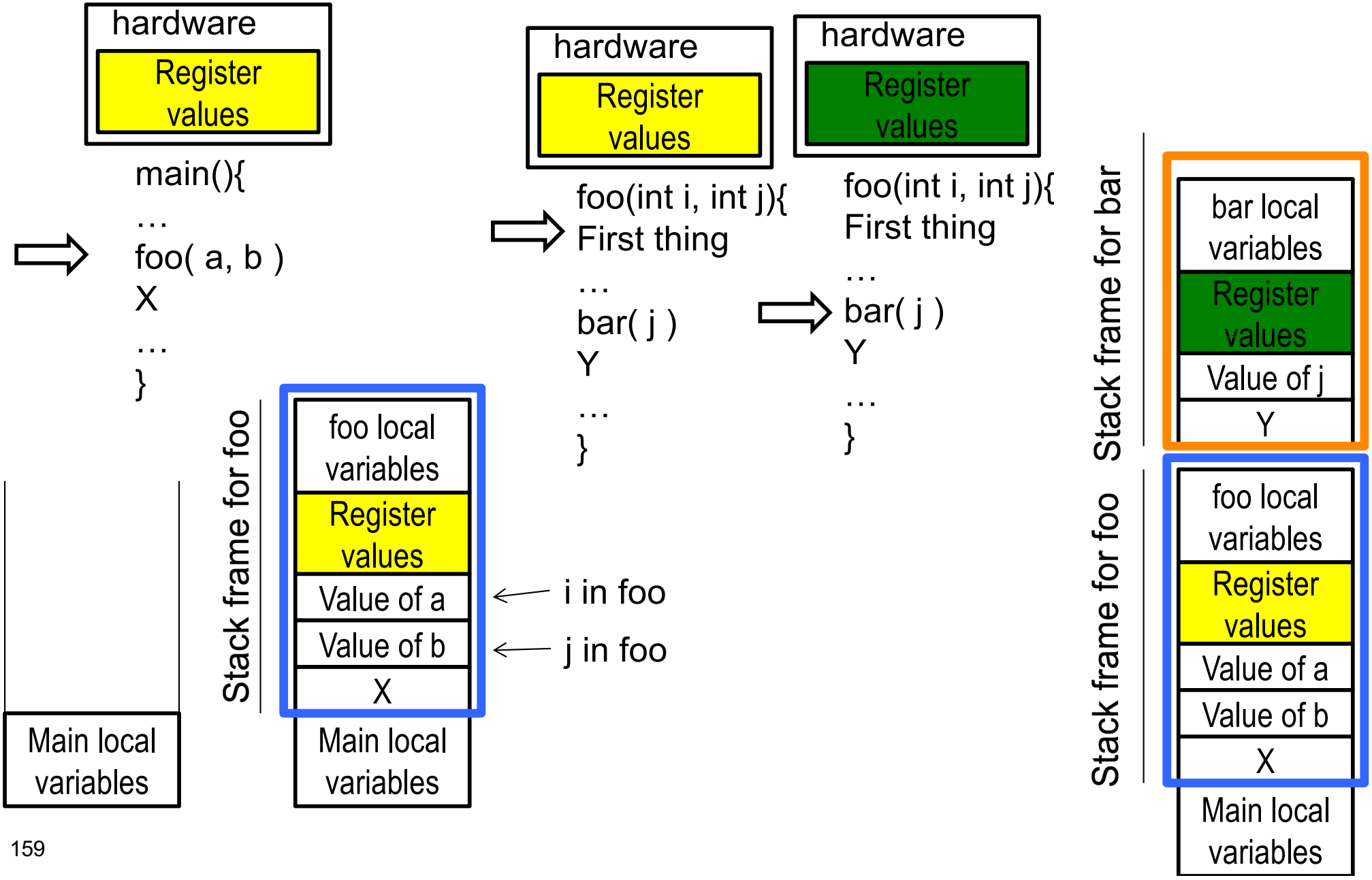
The call stack

- A stack is a last-in first-out (LIFO) data structure.
- Calling and returning from functions acts in a LIFO manner
  - We can use a stack to track which function in which we are currently executing
  - All of the data relating to the execution of a function is grouped into a stack frame
    - The data is a picture of the state of the function
    - The stack frame holds all the parts of the picture together

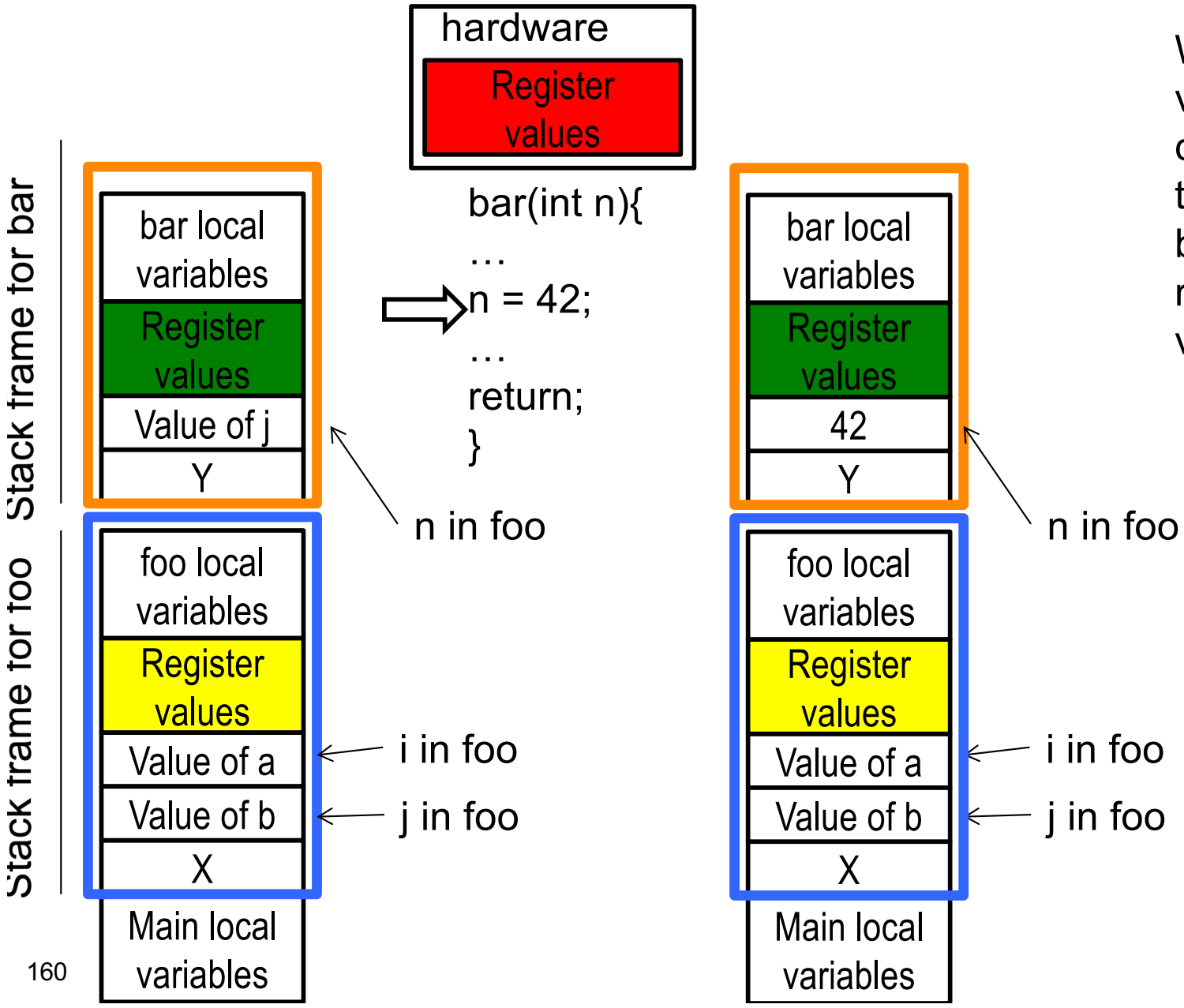
The call stack

- When we call a function, we create a stack frame for it and add the frame to the call stack
- The stack frame holds
  - The instruction to execute in the calling function when the function ends
  - Hardware information (register values) that the current function will change and need to be restored for the calling function
  - The parameters for the function
  - Space for local variables for the function
- When the function ends, the stack frame for the function is removed from the call stack, so all the information disappears

The call stack

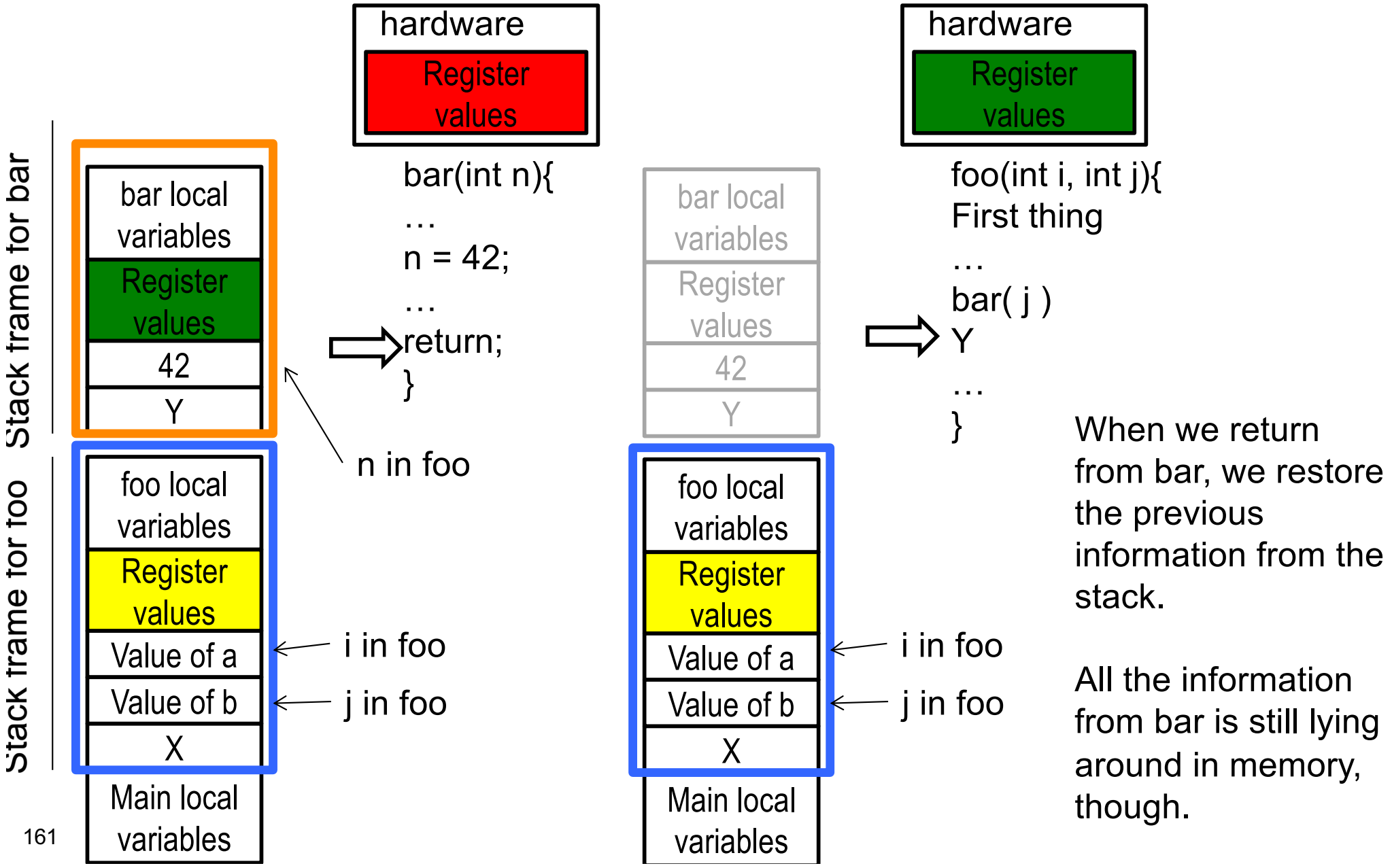


The call stack



When we set a value in bar, we change the value in the stack frame of bar. When foo runs, its copy of the value isn't changed!

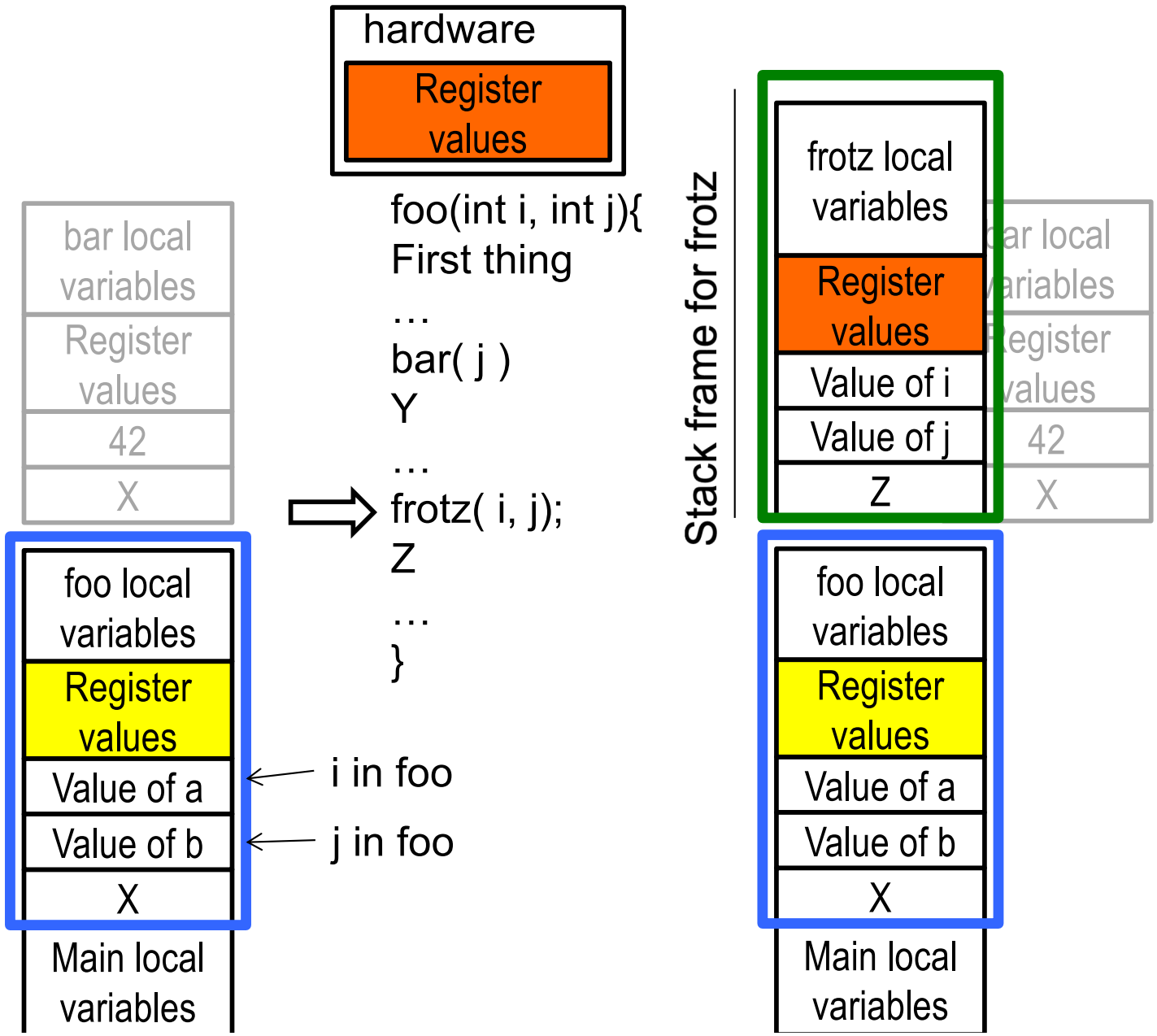
The call stack



When we return from bar, we restore the previous information from the stack.

All the information from bar is still lying around in memory, though.

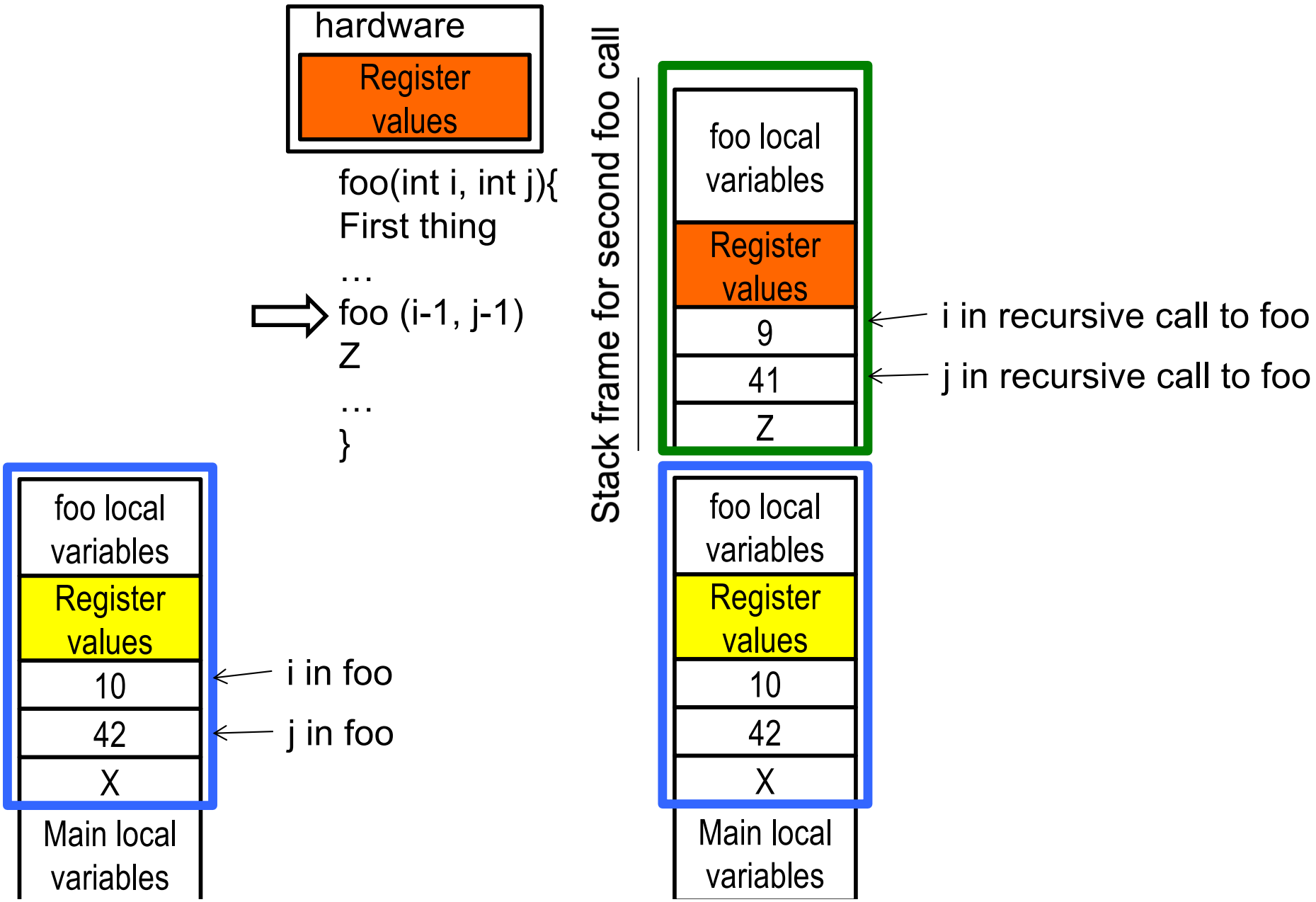
The call stack



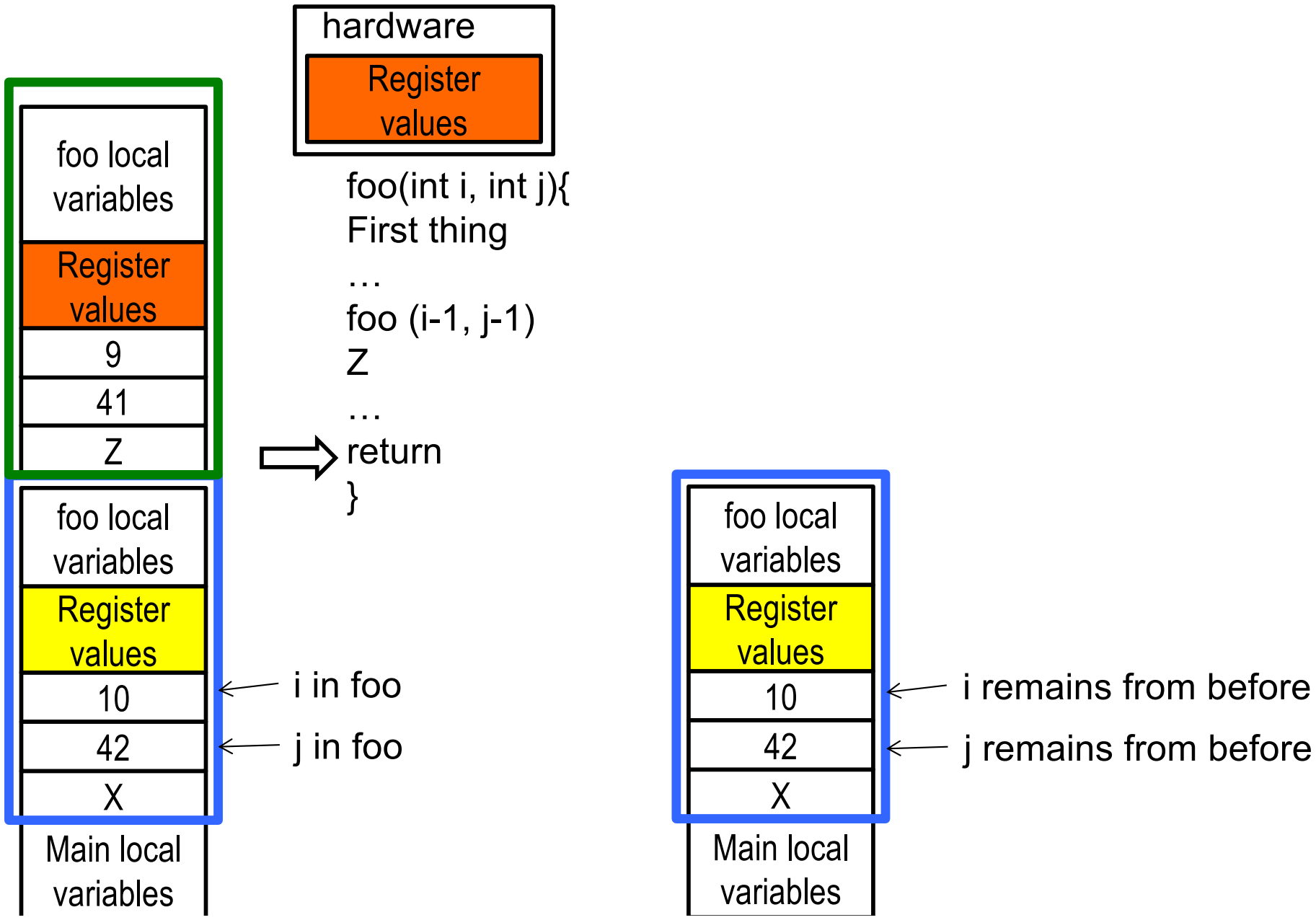
The stack frame for frotz overlaps the space previously used by bar.

All the old values for bar are still in memory, so that's what frotz sees as data unless you initialize your variables.

The call stack – recursive call



The call stack – recursive call ends



Common uses of recursion

- Binary tree traversals
- Binary search
- Divide and conquer style of problem solving
  - Mergesort, quicksort
- Backtracking
  - Lets the stack remember past decisions for you and "undo" changes to local variables when you reach a dead-end
- State space exploration
  - Easy to launch search in multiple directions
  - Want to remember the places where you have been so you don't visit them again

Sudoku