Bytecode – cont’d

Last time: For **local variables,**

- class file doesn't use the names
- compiler translates names into indexes
- used only in their own class

Today: For **methods** (and also for **instance & class variables**),

- potentially used in other classes too
- remember that classes are compiled separately ➔ compiler doesn't know addresses in other classes
- hence, the .class file must keep the names = symbolic references

Runtime constant pool

- One for each class; contains read-only constants used in the class
  - E.g. numeric, string
  - Symbolic references of method & its class
  - Types (of parameters, and return values)
- Each pool entry has an index: 0, 1, 2, …
- Each frame, in its “other stuff”, includes pointer to constant pool for its class

Static method calls

(non-static in the next lecture)

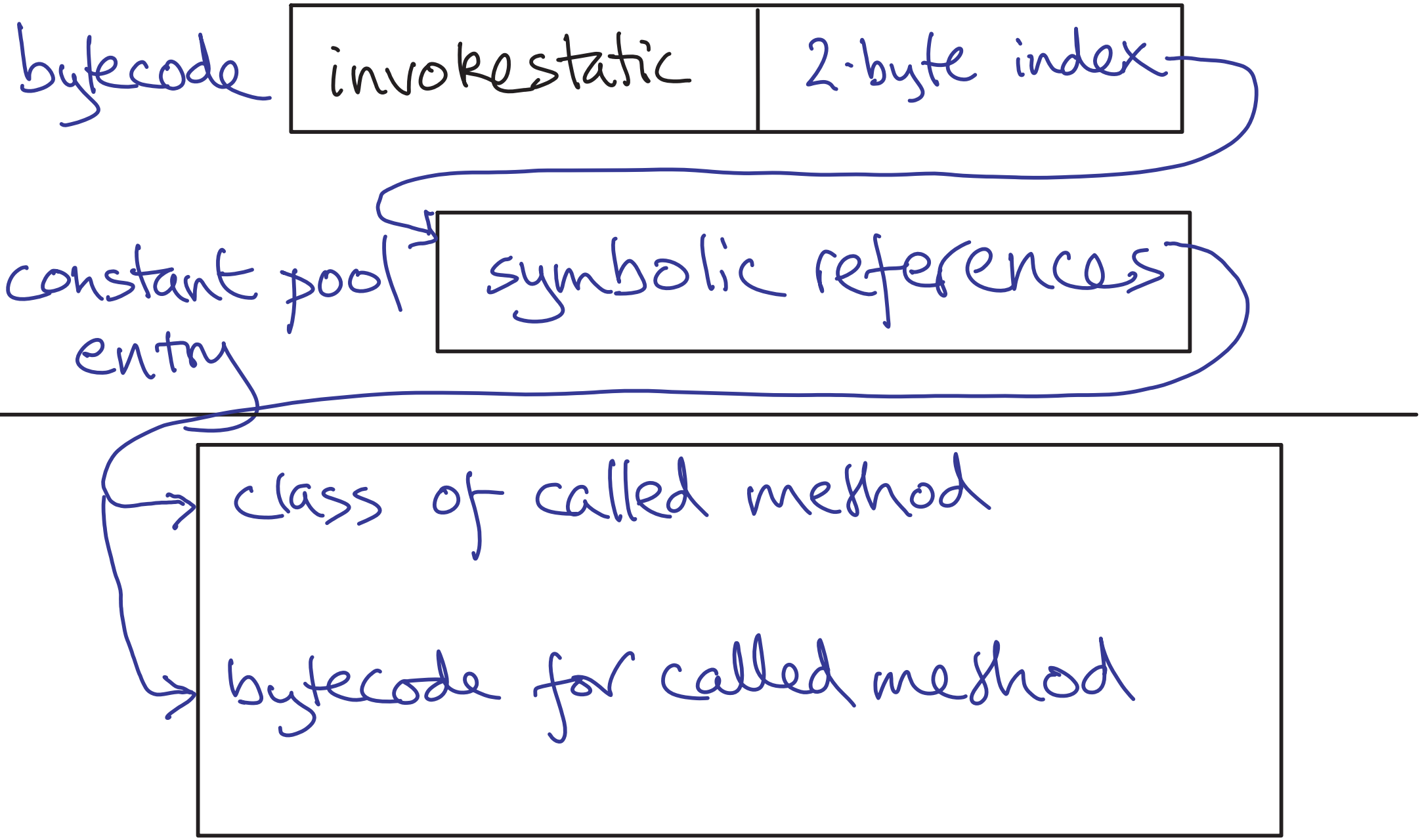
invokestatic 2-byte index

Used as index into constant pool of the class of method currently being executed

Constant pool entry is a symbolic reference to class & method

JVM uses these to find address of class, then the size needed for new frame + address of bytecode for method.

Static method calls



Method return

Static or non-static

For void methods

return

- Throw away current frame
- Carry on executing using caller's frame

(Its pc shows where it was when it made the method call.)

Method return

For methods that return a result

e.g. ireturn

int result is popped from operand stack and pushed onto caller's operand stack

Other types

lreturn freturn dreturn areturn

long float double ref

Then do the return as on previous slide.

e.g. recursive factorial

```
/**
 * Calculate factorial.
 * requires: 0 <= n
 * @param n number whose factorial is
 *     to be calculated
 * @return factorial of n
 */
public static int fact(int n){
    if (n==0){
        return 1;
    } else {
        return n*fact(n-1);
    }
}
```

$$n! = n * \overbrace{(n-1) * \cdots * 3 * 2 * 1}^{(n-1)!}$$

Recursion – When a method calls itself

- More about recursion later
  - and how to write recursive methods

For now:

Example shows how methods use frames

It also shows how frames save pc & local variables
- so recursion works OK.

Frame for fact

```
public static int
    fact(int n) {
  if (n==0) {
    return 1;
  } else {
    return n*fact(n-1);
  }
}
```

One parameter (n)

No other local variables

How big an operand stack?

Most complicated calculation

n * fact(n-1)

Reverse Polish:

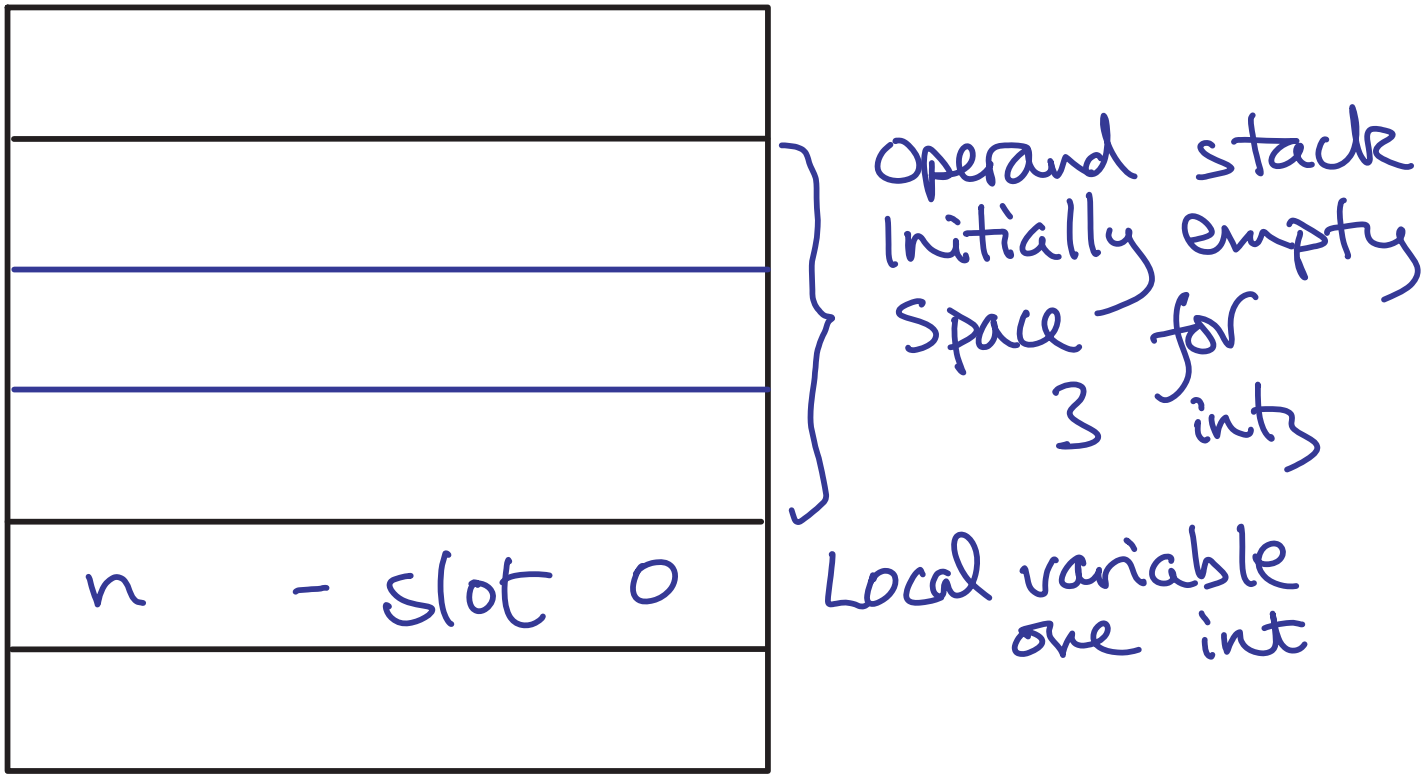
n n 1 - fact *

stack

n	n	1	n-1	fact(n-1)	n*fact(n-1)
	n	n	n	n	
		n			

max depth = 3

Frame for fact



javap disassembler

"disassembler" — converts byte code to mnemonics

Command javap can be applied to a .class file and displays its structure

e.g. maybe don't need this

```
javap -classpath . -c Factorial
```

- `-classpath .` — look in current folder
- `-c` — show bytecode
- `Factorial` — disassembles Factorial.class

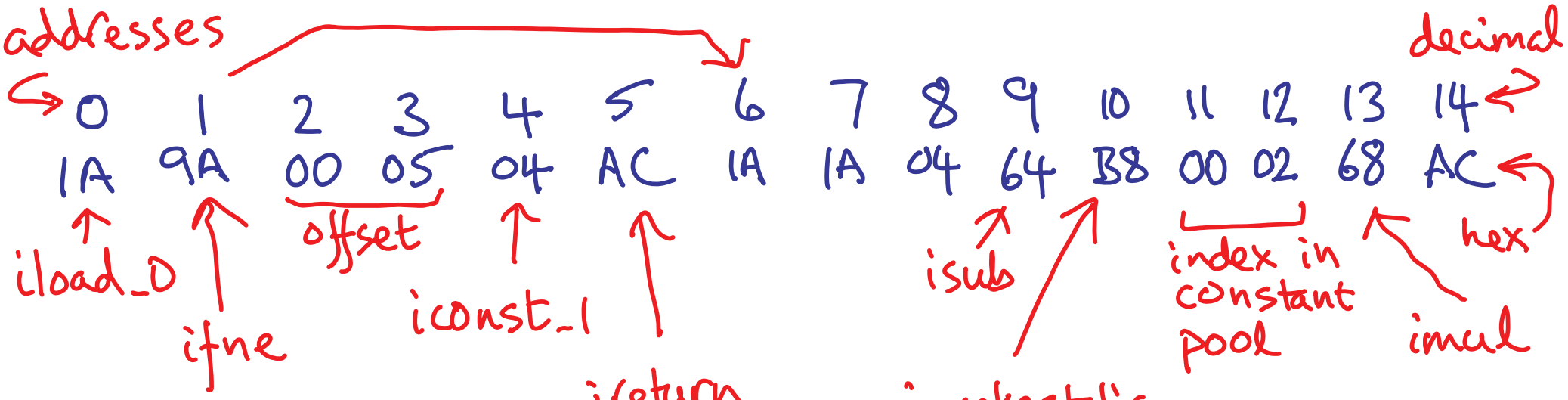
Option -l gives table of local variables

Bytecode in mnemonics

Addresses in decimal

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```

Bytecode in bytes



Conditional jump is from address 1 (ifne) to 6 (iload_0). ∴ offset = 5

Mnemonics show absolute address 6

We're not going to use these opcodes. But they are what a .class file is really made of.

What bytecode does what Java?

```
public static int fact(int);
  Code:
   0:   iload_0
   1:   ifne    6
   4:   iconst_1
   5:   ireturn
   6:   iload_0
   7:   iload_0
   8:   iconst_1
   9:   isub
   10:  invokestatic    #2; //Method fact:(I)I
   13:  imul
   14:  ireturn
```

```
public static int fact(int n){
    if (n==0){
        return 1;
    } else {
        return n*fact(n-1);
    }
}
```

What bytecode does what Java?

```
public static int fact(int);
  Code:
    0:  iload_0
    1:  ifne    6
    4:  iconst_1
    5:  ireturn
    6:  iload_0
    7:  iload_0
    8:  iconst_1
    9:  isub
    10: invokestatic    #2; //Method fact:(I)I
    13: imul
    14: ireturn
```

Why do they all start with i?

What bytecode does what Java?

```
public static int fact(int);
  Code:
    0:  iload_0
    1:  ifne  6
    4:  iconst_1
    5:  ireturn
    6:  iload_0
    7:  iload_0
    8:  iconst_1
    9:  isub
    10: invokestatic  #2; //Method fact:(I)I
    13: imul
    14: ireturn
```

— push variable 0, i.e. n

— conditional jump

— return integer

— push integer constant 1

— integer subtract

— call fact

— integer multiply

Why do they all start with i?

Mostly: shows int type for operation.

Exceptions: ifne, invokestatic

What bytecode does what Java?

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic  #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```

stack

n

empty

1

n

n, n

n, n, 1

n, n-1

n, fact(n-1)

n * fact(n-1)

What bytecode does what Java?

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic  #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```

```
public static int fact(int n)
{
    if (n==0){
        return 1;
    } else {
        return n*fact(n-1);
    }
}
```

stack

n

empty

1

n

n, n

n, n, 1

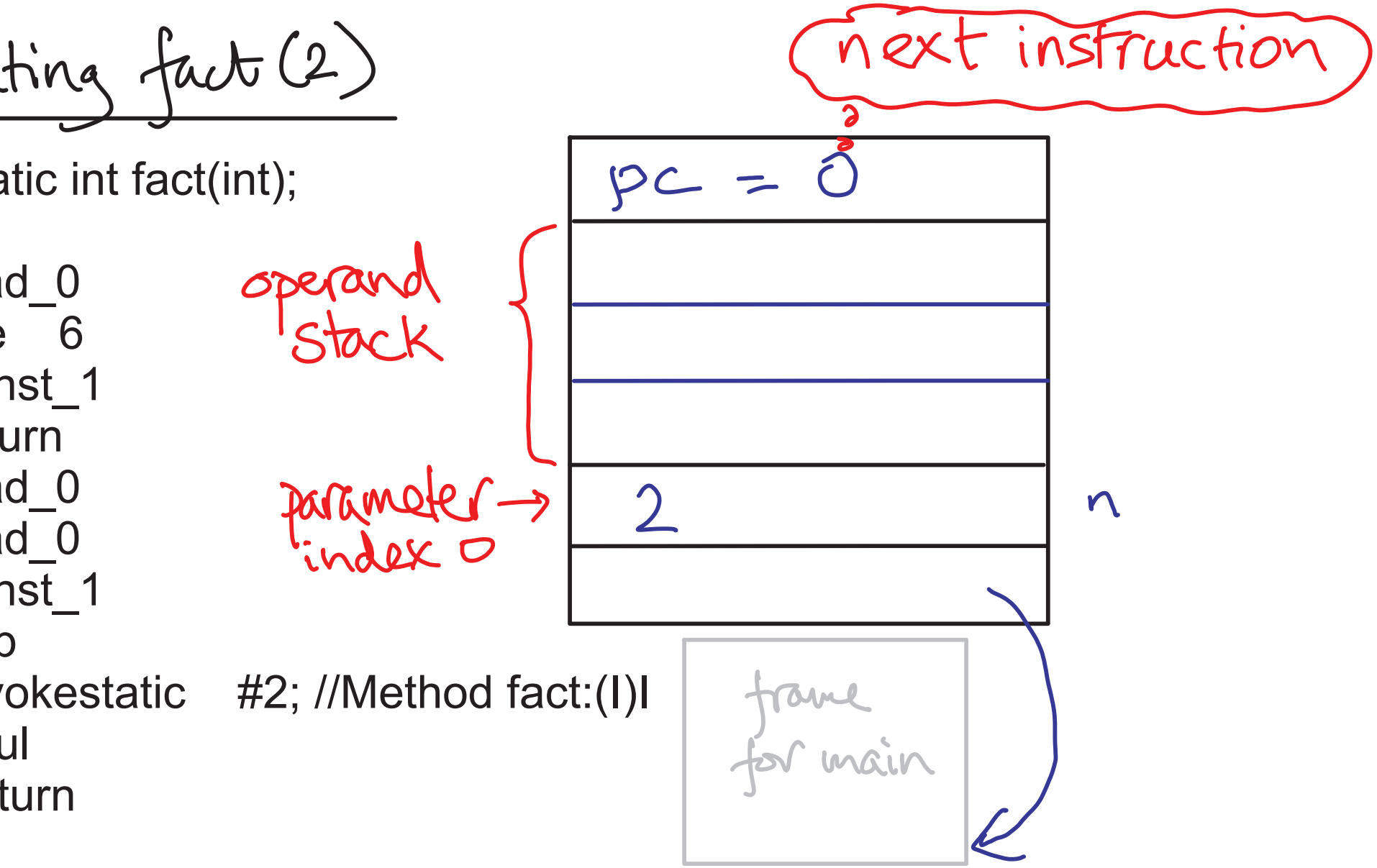
n, n-1

n * fact(n-1)

n, fact(n-1)

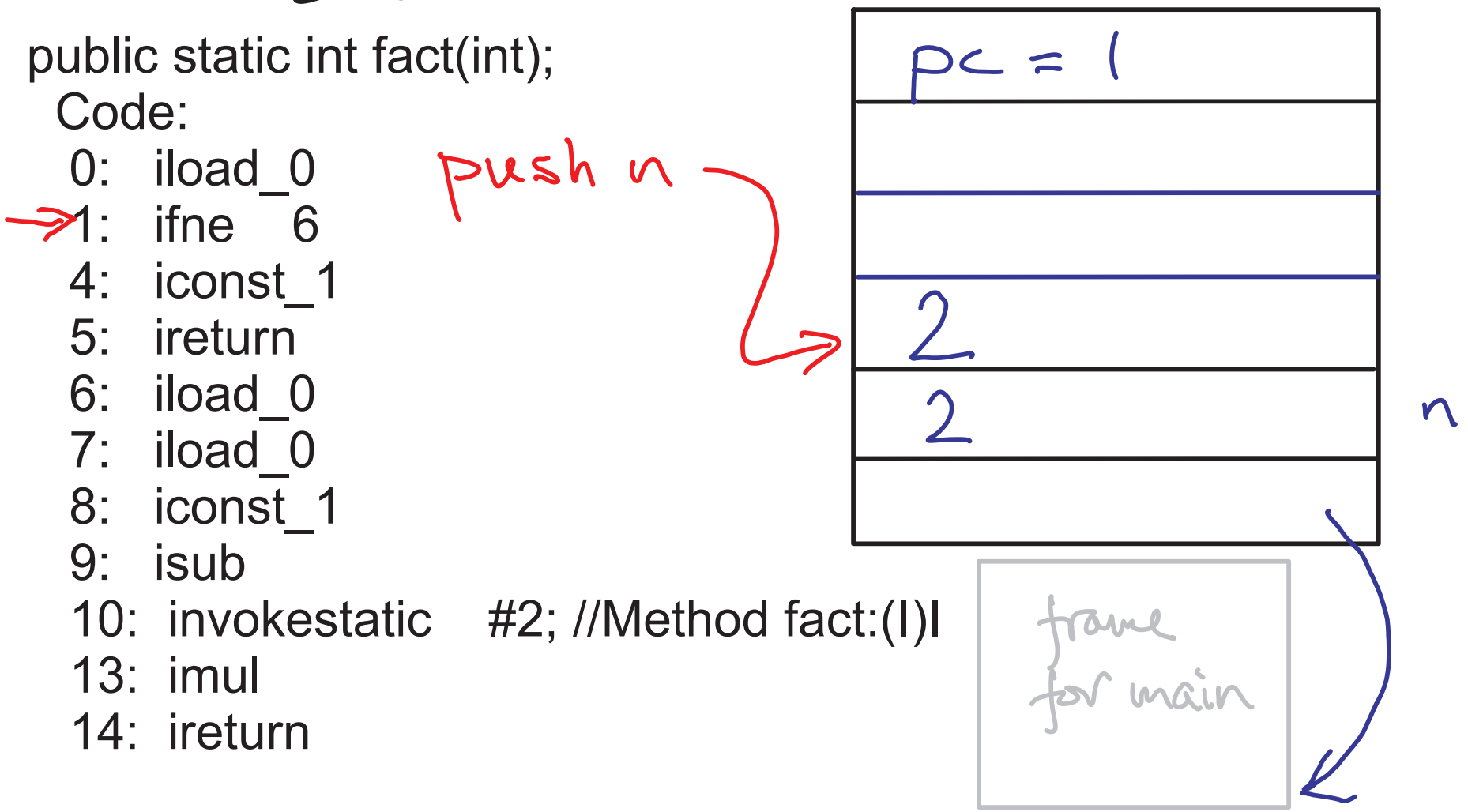
Executing fact(2)

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```



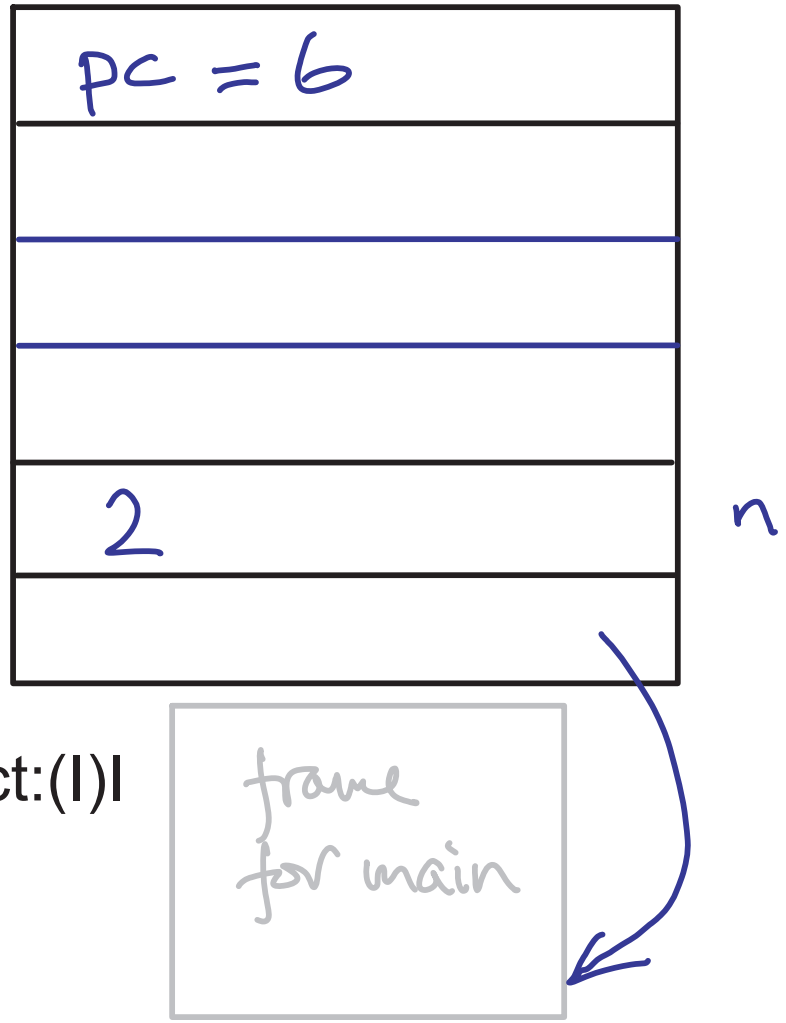
Executing fact(2)

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```



Executing fact(2)

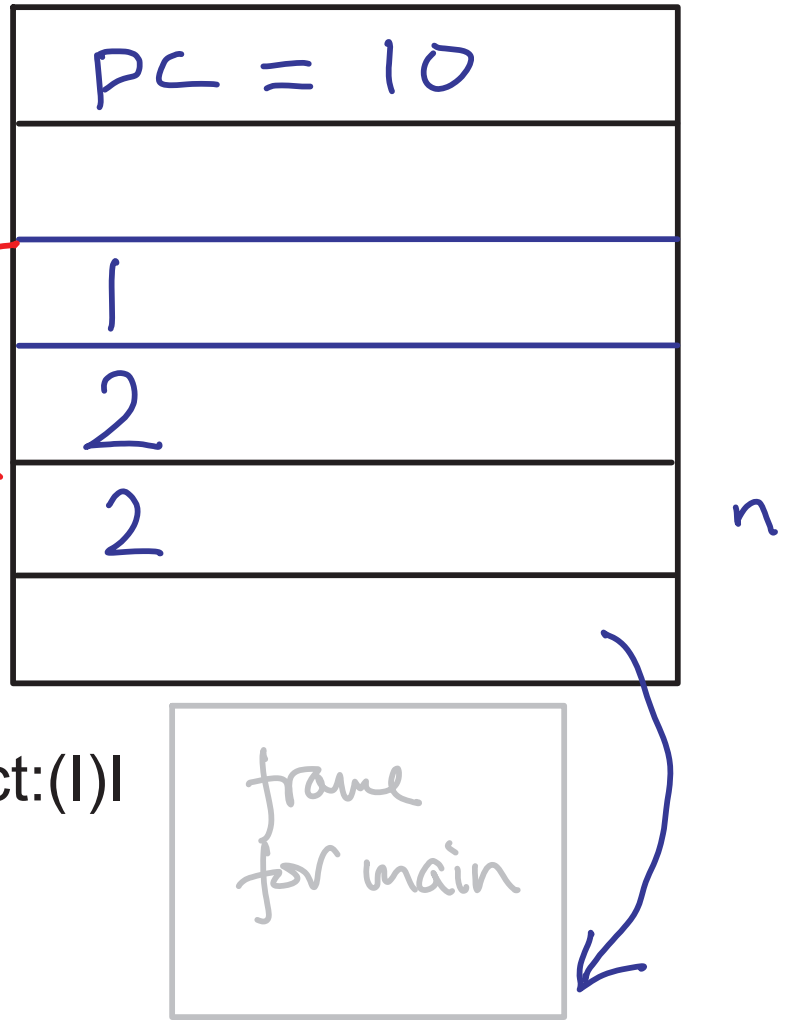
```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```



Executing fact(2)

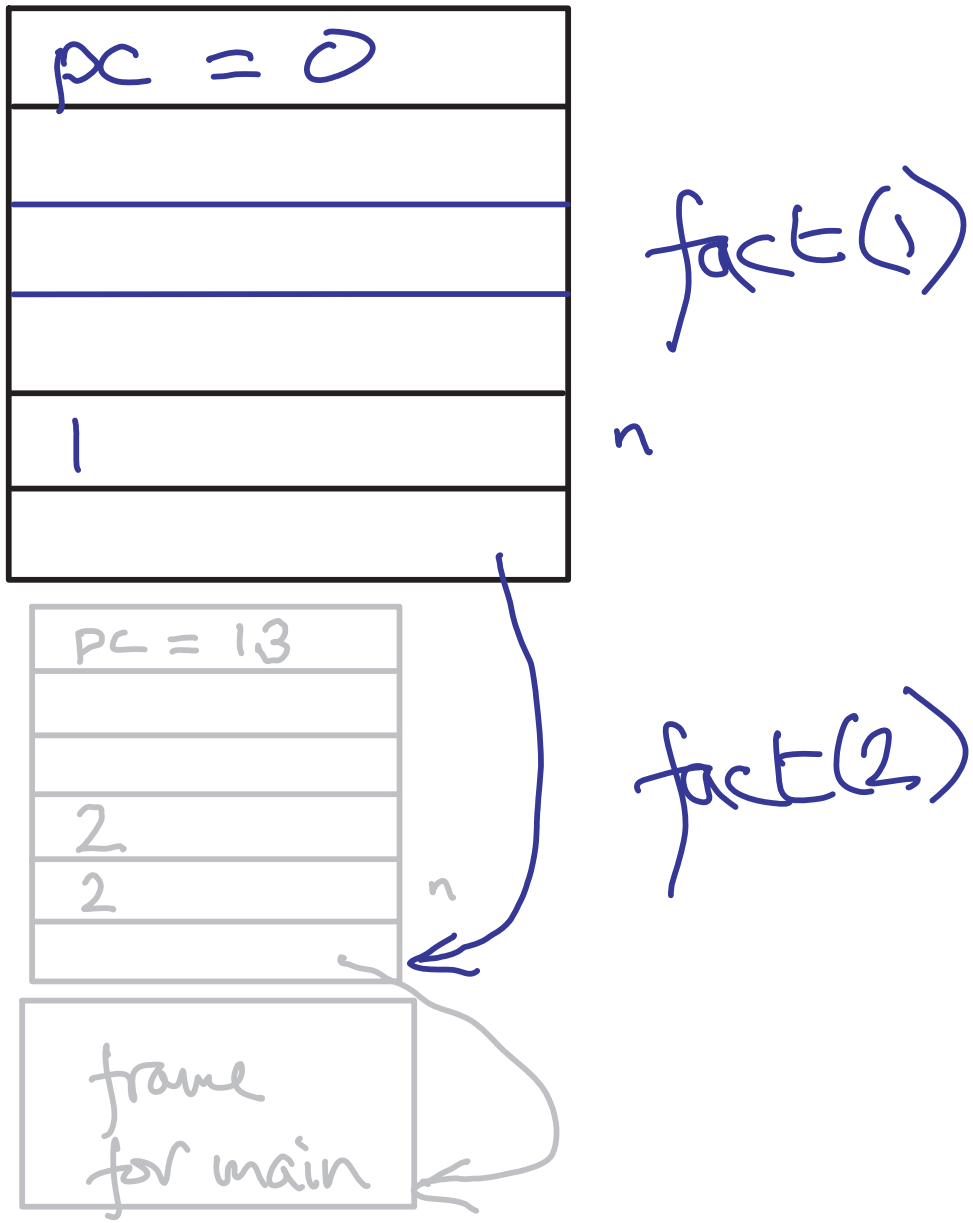
```
public static int fact(int);
  Code:
    0:  iload_0
    1:  ifne   6
    4:  iconst_1
    5:  ireturn
    6:  iload_0
    7:  iload_0
    8:  iconst_1
    9:  isub
   10:  invokestatic  #2; //Method fact:(I)I
   13:  imul
   14:  ireturn
```

2
empty
2
2, 2
2, 2, 1
2, 1



Recursive call fact(1)

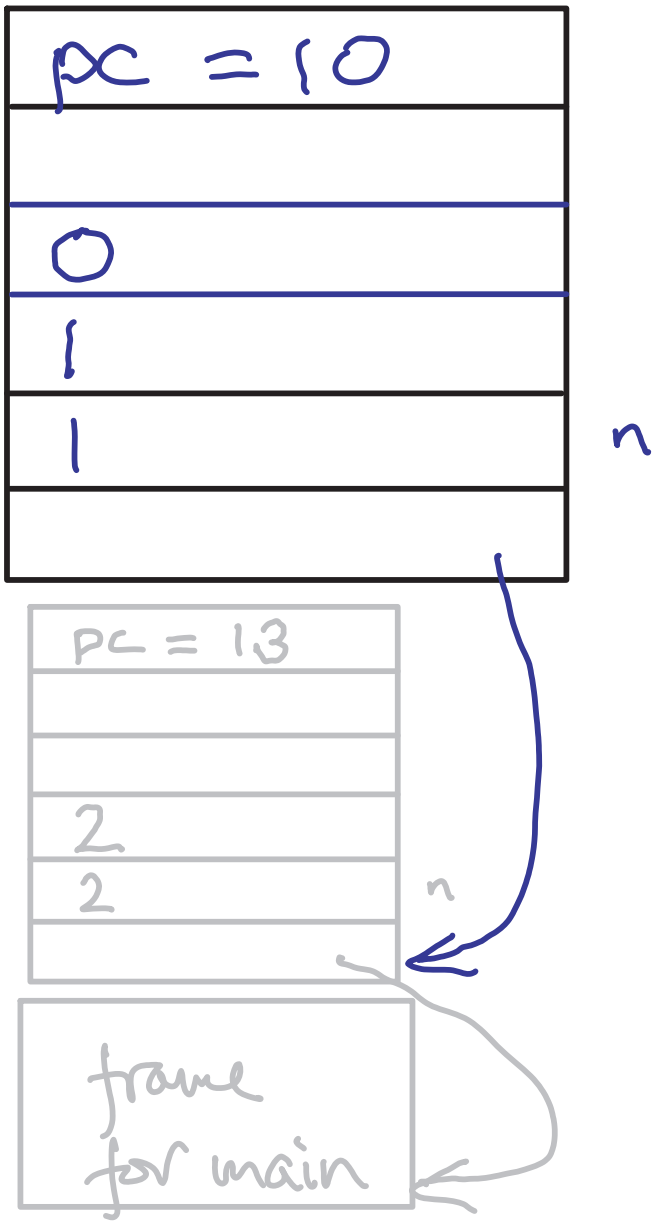
```
public static int fact(int);
  Code:
    0:  iload_0
    1:  ifne    6
    4:  iconst_1
    5:  ireturn
    6:  iload_0
    7:  iload_0
    8:  iconst_1
    9:  isub
    10: invokestatic    #2; //Method fact:(I)I
    13: imul
    14: ireturn
```



Two frames for fact:
one for each call

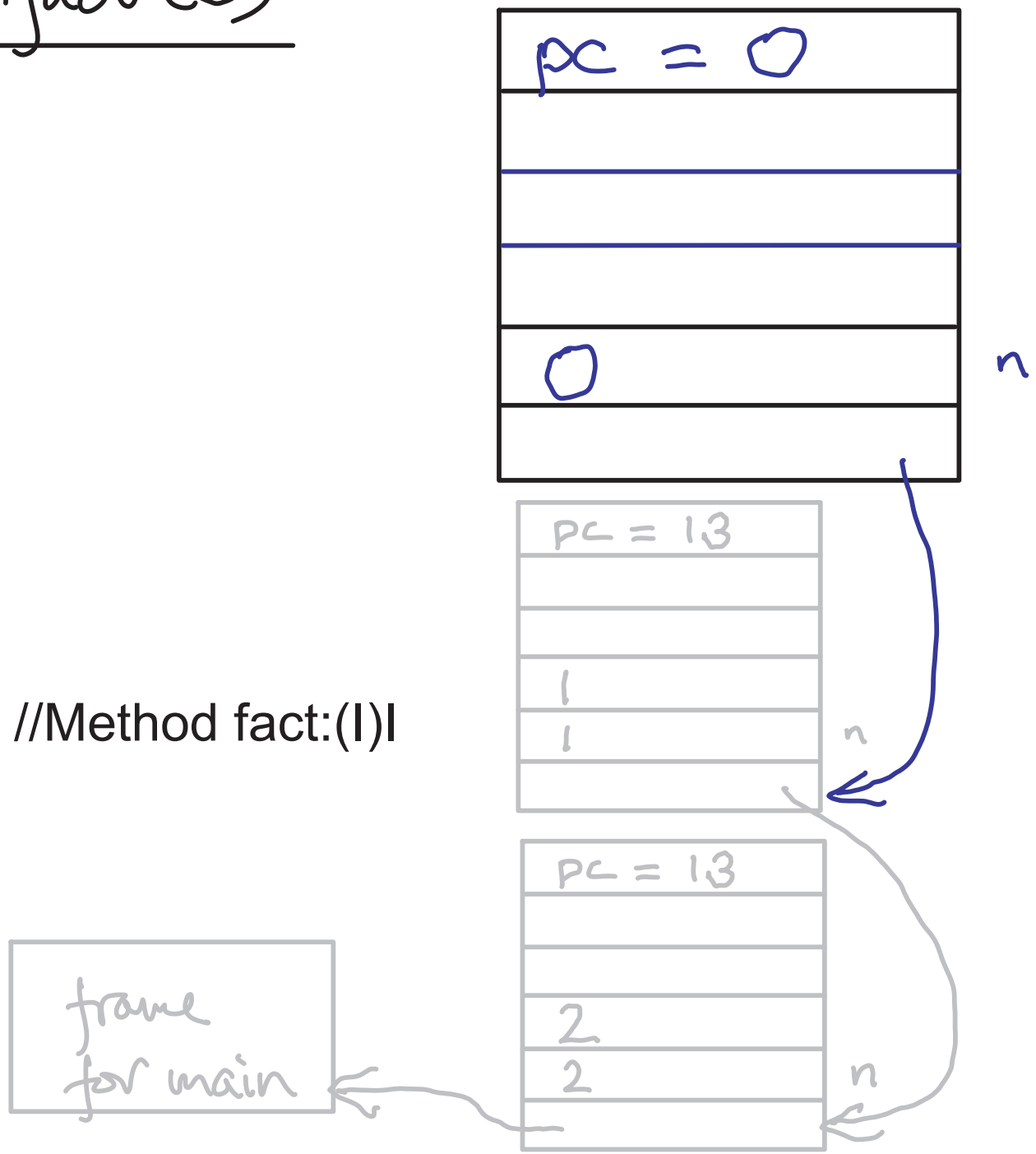
Recursive call fact(1)

```
public static int fact(int);
  Code:
    0:  iload_0          1
    1:  ifne    6        empty
    4:  iconst_1
    5:  ireturn
    6:  iload_0          1
    7:  iload_0          1, 1
    8:  iconst_1         1, 1, 1
    9:  isub             1, 0
   10:  invokestatic     #2; //Method fact:(I)I
   13:  imul
   14:  ireturn
```



Recursive call fact(0)

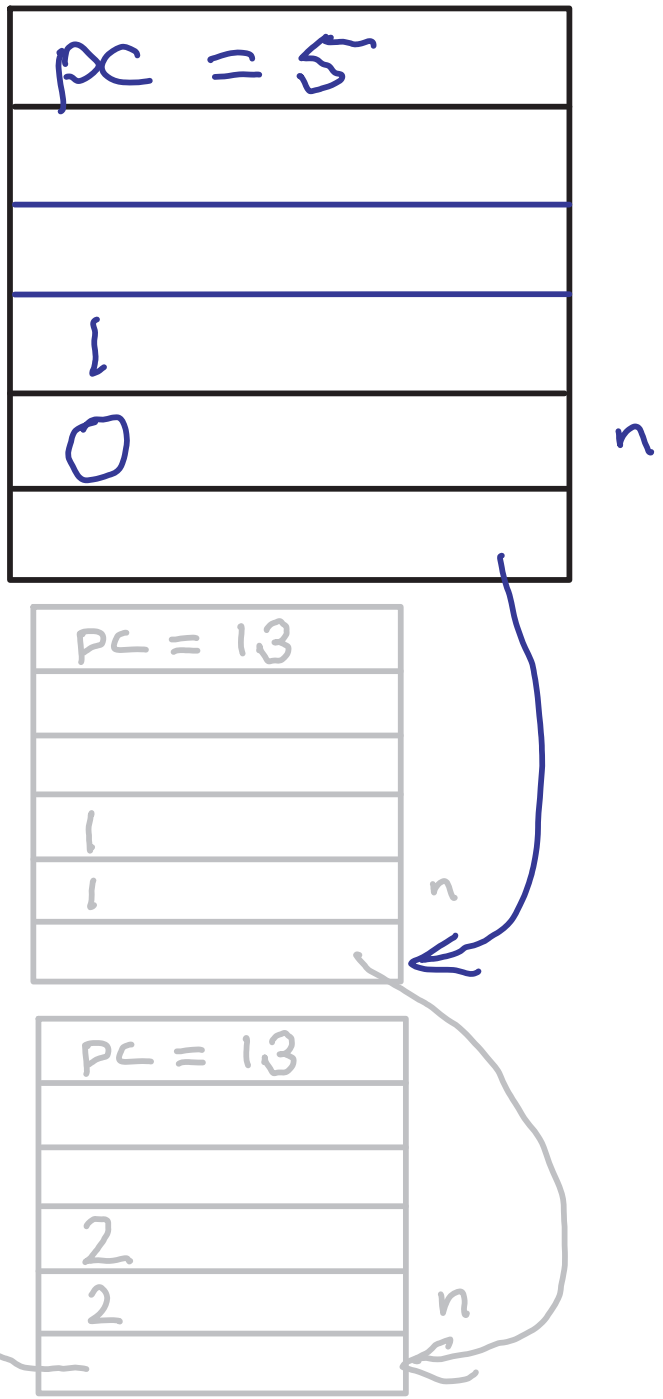
```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```



Recursive call fact(0)

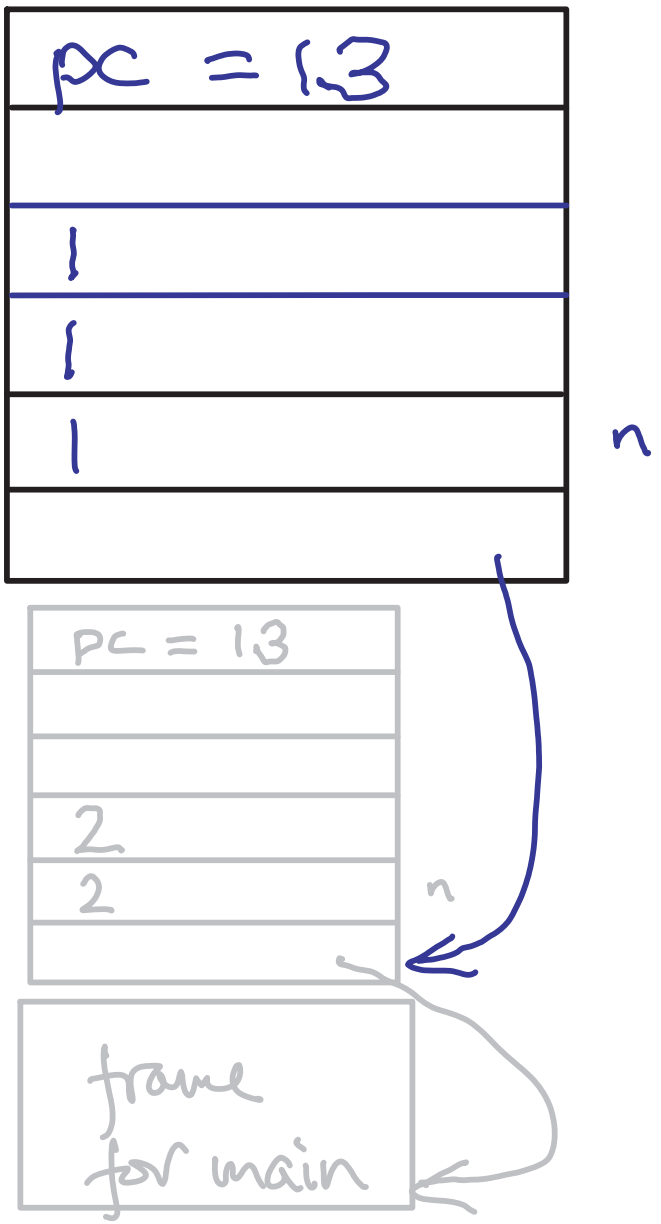
```
public static int fact(int);
  Code:
    0:  iload_0
    1:  ifne    6
    4:  iconst_1
    5:  ireturn
    6:  iload_0
    7:  iload_0
    8:  iconst_1
    9:  isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```

0
empty
1



Return from fact (0)

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```

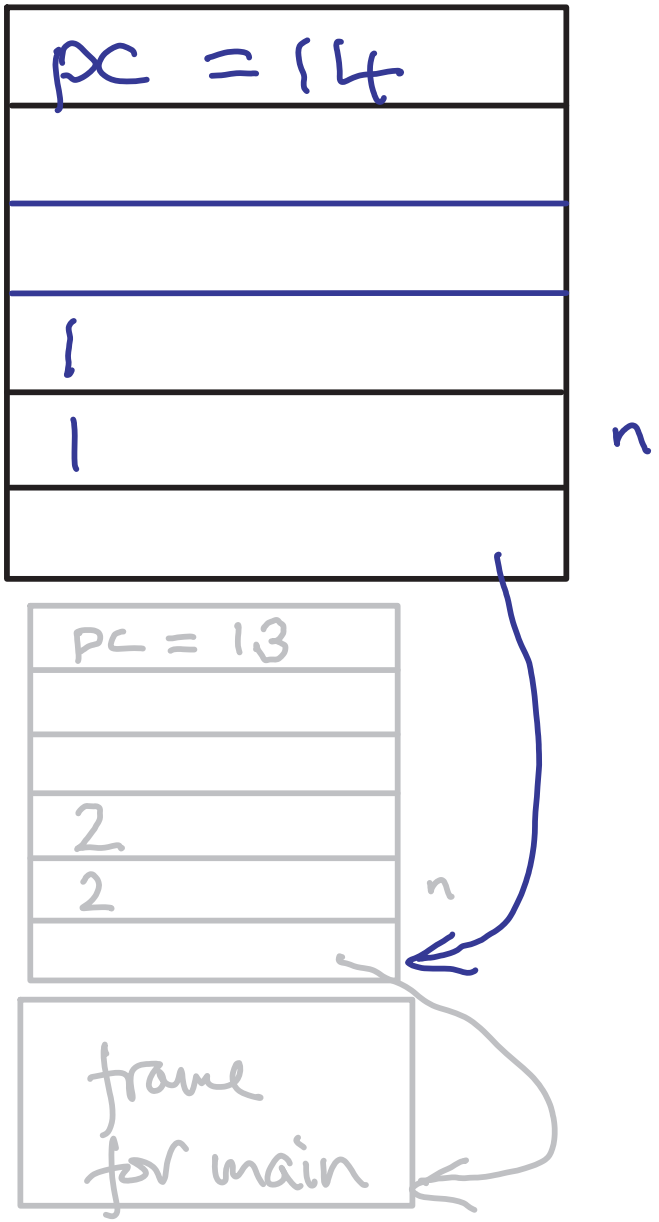


Completing fact (1)

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```

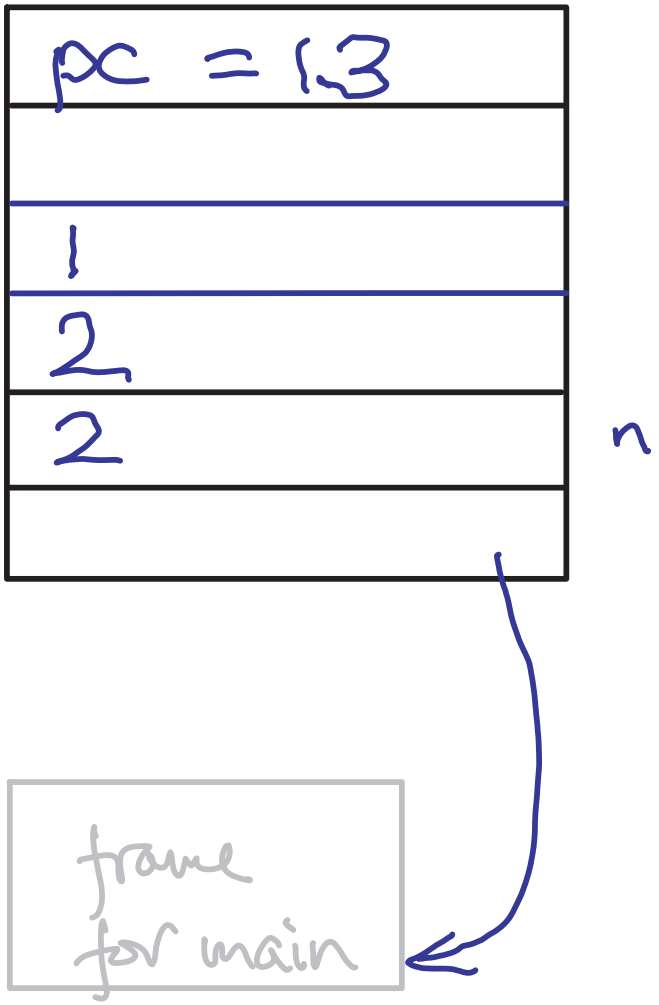
1

1,1



Return from fact (1)

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```



Completing fact (2)

```
public static int fact(int);
  Code:
    0:   iload_0
    1:   ifne    6
    4:   iconst_1
    5:   ireturn
    6:   iload_0
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fact:(I)I
    13:  imul
    14:  ireturn
```

2,1

2