Recursion

RECURSION

Recursion

- An algorithm or solution that is defined in terms of itself.
- A recursive method calls invokes itself.

Recursive Solutions

- Recursion is an approach to solving problems that breaks a problem into an *identical* but *smaller* problems.
  - You continue breaking the problem into smaller problems until you reach the smallest problem possible.
  - In this smallest problem, the answer is obvious or trivial.
  - You then use this answer to “build back up” and solve the previous problems until you solve the original problem.

- If a problem is easy, solve it now.
- If the problem is hard, solve a small piece of it now, then make it smaller and solve the rest later.
- Often, you combined all the solutions to get the final answer.

Elements of Recursive Methods

- A base case
  - Something that defines when the recursion ends
- A recursive case
  - Something that solves a smaller part of the problem
  - Often solves some part now and then calls itself to solve the rest later.
  - Must eventually advance towards the base case!

Elements of Recursive Methods

- In other words, all recursive methods must:
  - Make the problem smaller
  - Know when to stop

The Base Case

- The non-recursive part of a recursive definition is called the *base case*
- Without a base case, there would be no way to terminate the recursion, creating *infinite recursion*
  - This is similar to an infinite loop
- All recursive definitions must have one or more base cases

Example: Blastoff

- Review the Blastoff example in IntroductoryRecursionExamples.java.

Factorial

- Factorial: the product of an integer and all positive integers below it
  - n!
- Example: 4! = 4 * 3 * 2 * 1 = 24
- Example: 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720
- Example: 1! = 1

- Note: 0! = 1 (this is just be definition!)

Recursively Defining Factorial

- Let's look again at the first few values:
- 1! = 1
- 2! = 2 * 1 = 2
- 3! = 3 * 2 * 1 = 6
- 4! = 4 * 3 * 2 * 1 = 24
- 5! = 5 * 4 * 3 * 2 * 1 = 120

Recursively Defining Factorial

- 1! = 1
- 2! = 2 * 1 = 2
- 3! = 3 * 2 * 1 = 6
- 4! = 4 * 3 * 2 * 1 = 24
- 5! = 5 * 4 * 3 * 2 * 1 = 120

Recursively Defining Factorial

- $1! = 1$
- $2! = 2 * 1! = 2$
- $3! = 3 * 2! = 6$
- $4! = 4 * 3! = 24$
- $5! = 5 * 4! = 120$

Recursively Defining Factorial

- 1! = 1
- 2! = 2 * 1! = 2
- 3! = 3 * 2! = 6
- 4! = 4 * 3! = 24
- 5! = 5 * 4! = 120

- n! = n * (n-1)!

Recursively Defining Factorial

- n! = n * (n-1)!

- But when do we stop?!
- Base case: 1! = 1

Recursively Defining Factorial

- $n! = n * (n-1)!$
- $1! = 1$

- Factorial is defined in terms of factorial- this is what makes it recursive.
- There is a base case that tells us when to stop.
  - The recursive case moves towards the base case.
- The recursive case combines a part of the current solution to the future solution.

Example: Factorial

- Review the Factorial example in IntroductoryRecursionExamples.java.

RECURSION IN JAVA

Recursive Methods

- A recursive method invokes itself.
- A recursive method includes:
  - one or more base cases
  - one or more recursive cases that advance towards the base case
  - a conditional to determine which case you're in!

- Recursive methods can be void or can return a value.

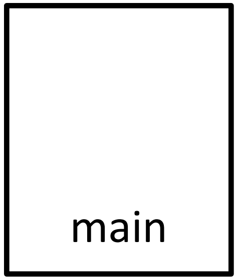
Method Control in Java

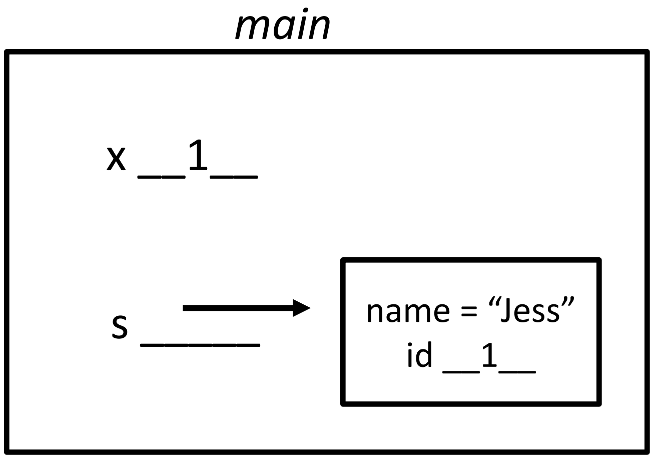
- When a method is invoked, the current method is paused and control passes to the invoked method.
  - When that method finishes, control returns back to the original method.
  - That method picks up where it left off.
- Each call to a method sets up a new execution environment
  - New parameters
    - But be careful about objects! Remember that Java is pass by value. The value of an object is a memory location/reference. So passing around objects does not make copies! It results in aliases.
  - New local variables

Example: Method Trace

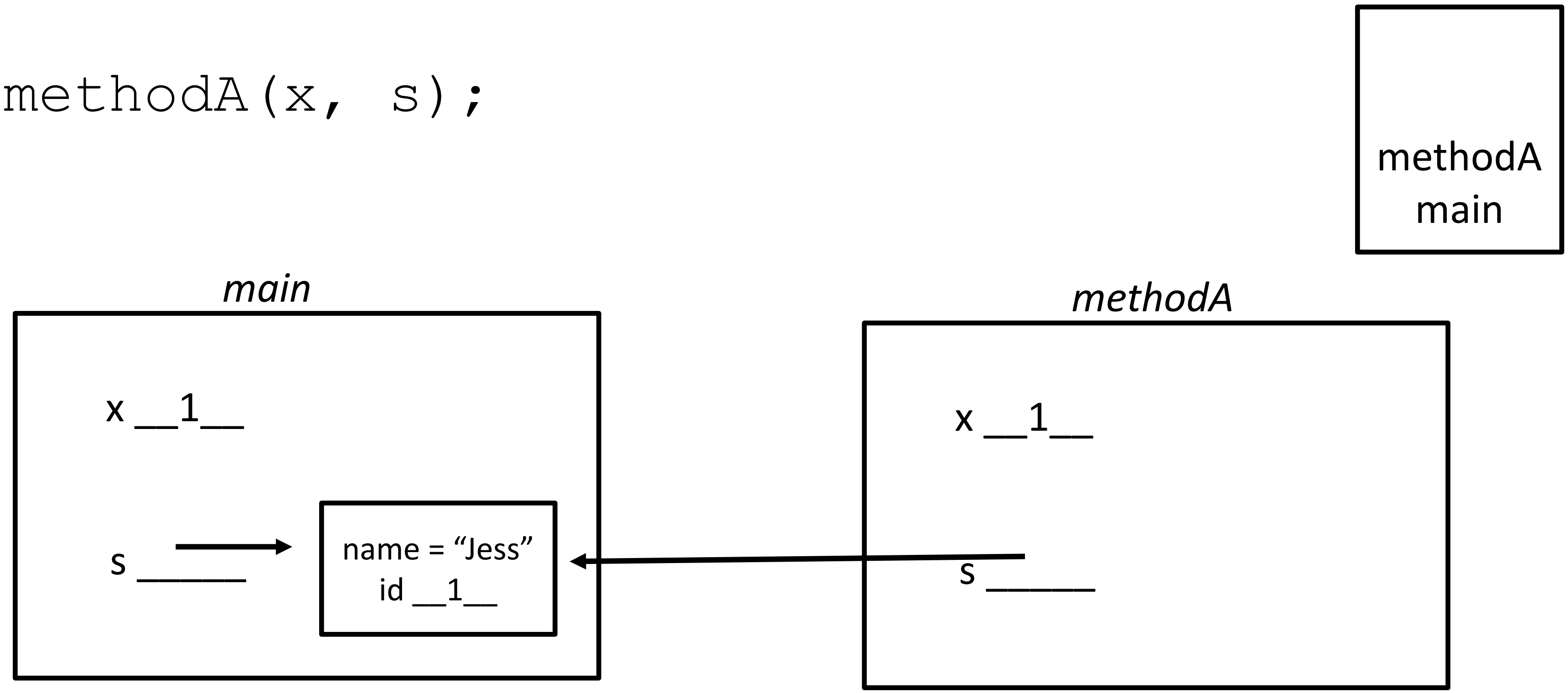
- Review the MethodTrace example.
  - This shows how methods are called and parameters are passed.
  - This example does **not** use recursion!

```
int x = 1;
Student s = new Student("Jess", 1);
```

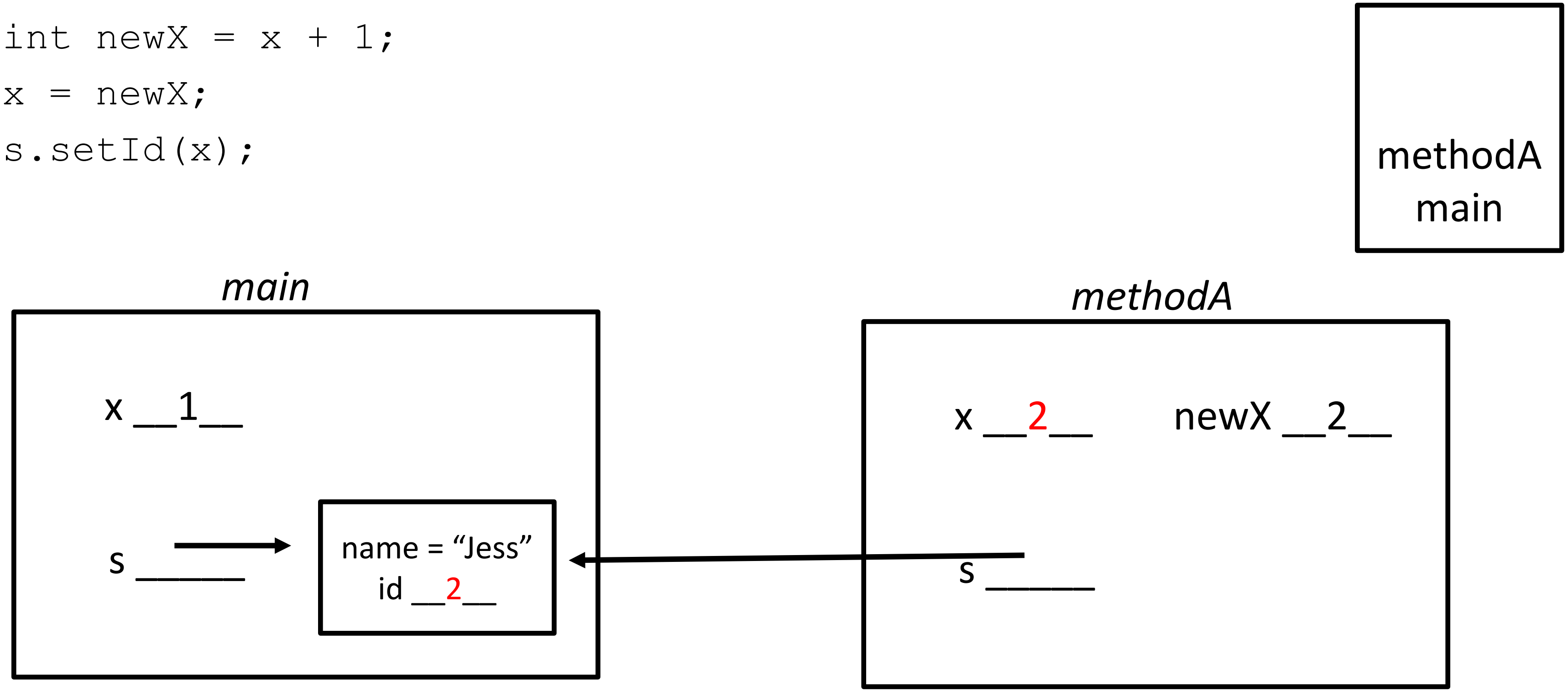




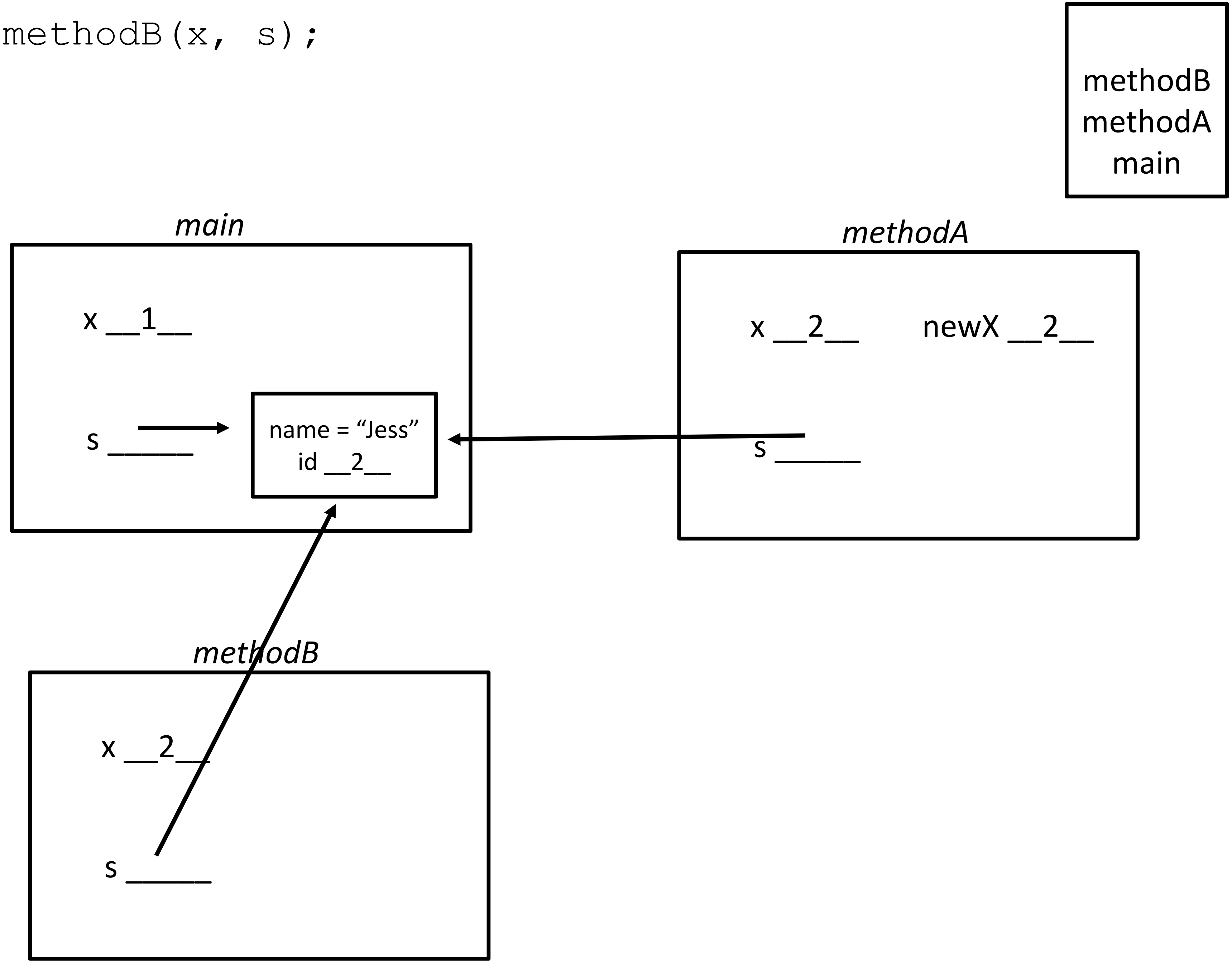
```
methodA(x, s);
```



```
int newX = x + 1;
x = newX;
s.setId(x);
```

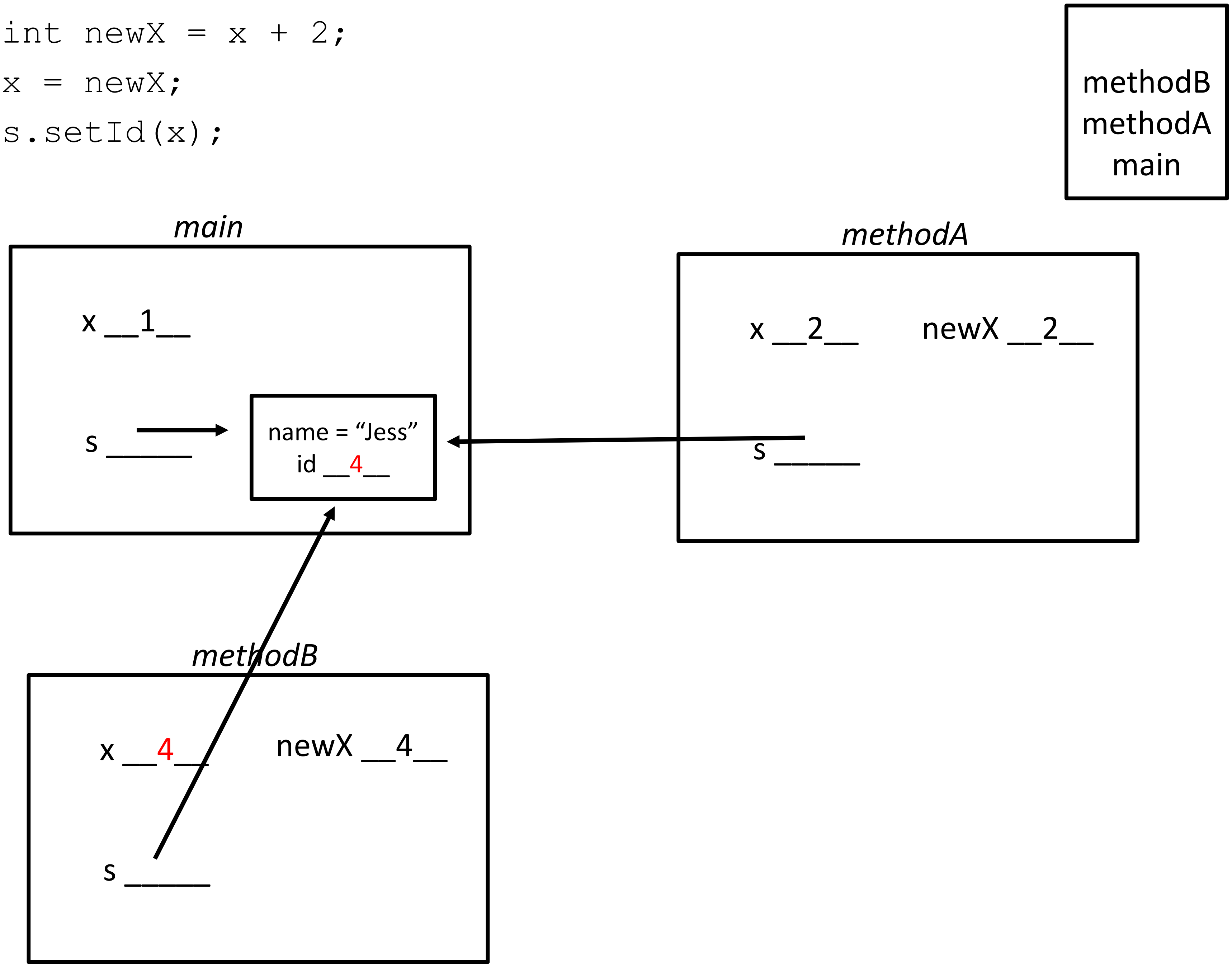


```
methodB(x, s);
```



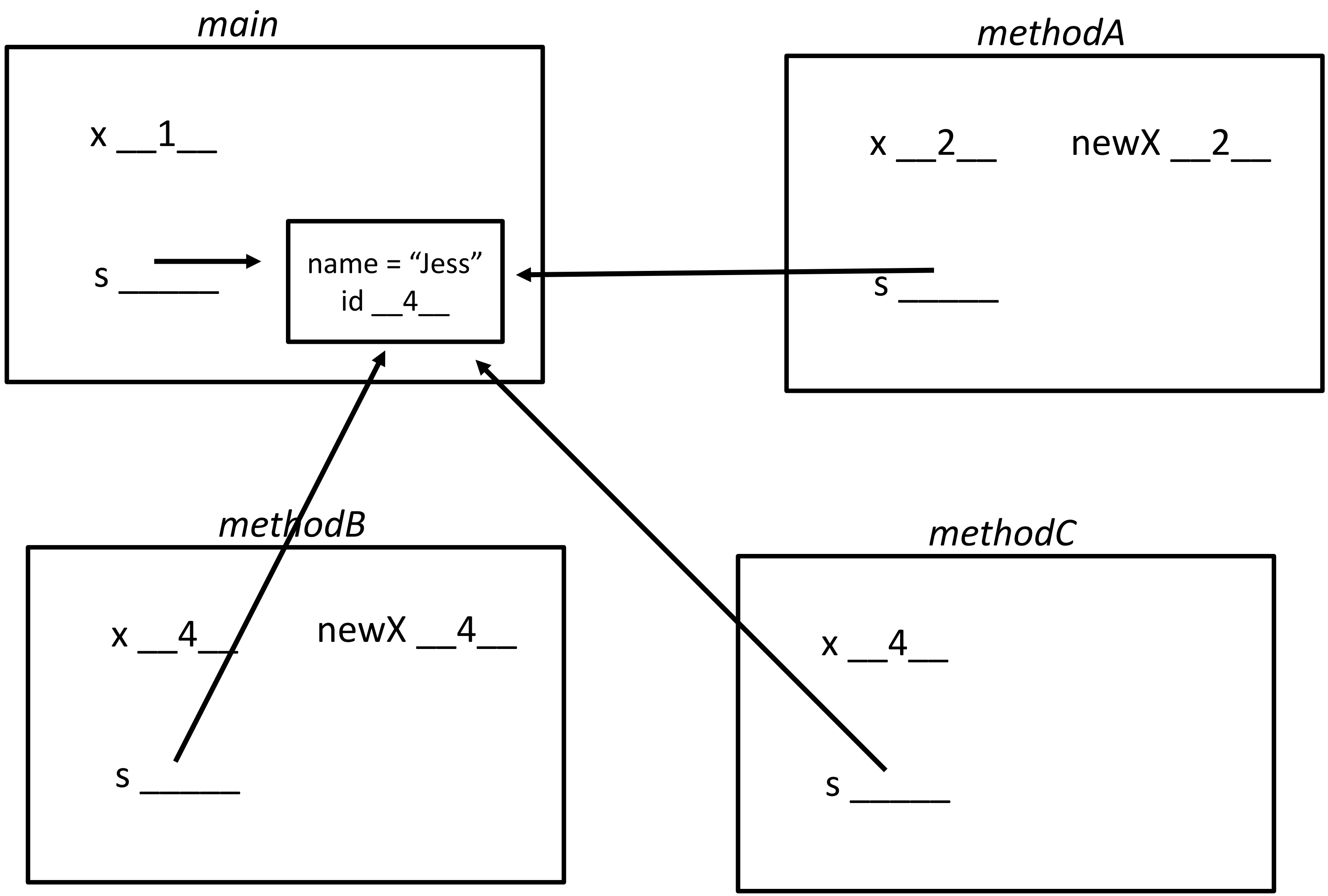
methodB
methodA
main
main
x __1__
s ______
name = "Jess"
id __2__
methodA
x __2__
newX __2__
s ______
methodB
x __2__
s ______

```
int newX = x + 2;
x = newX;
s.setId(x);
```



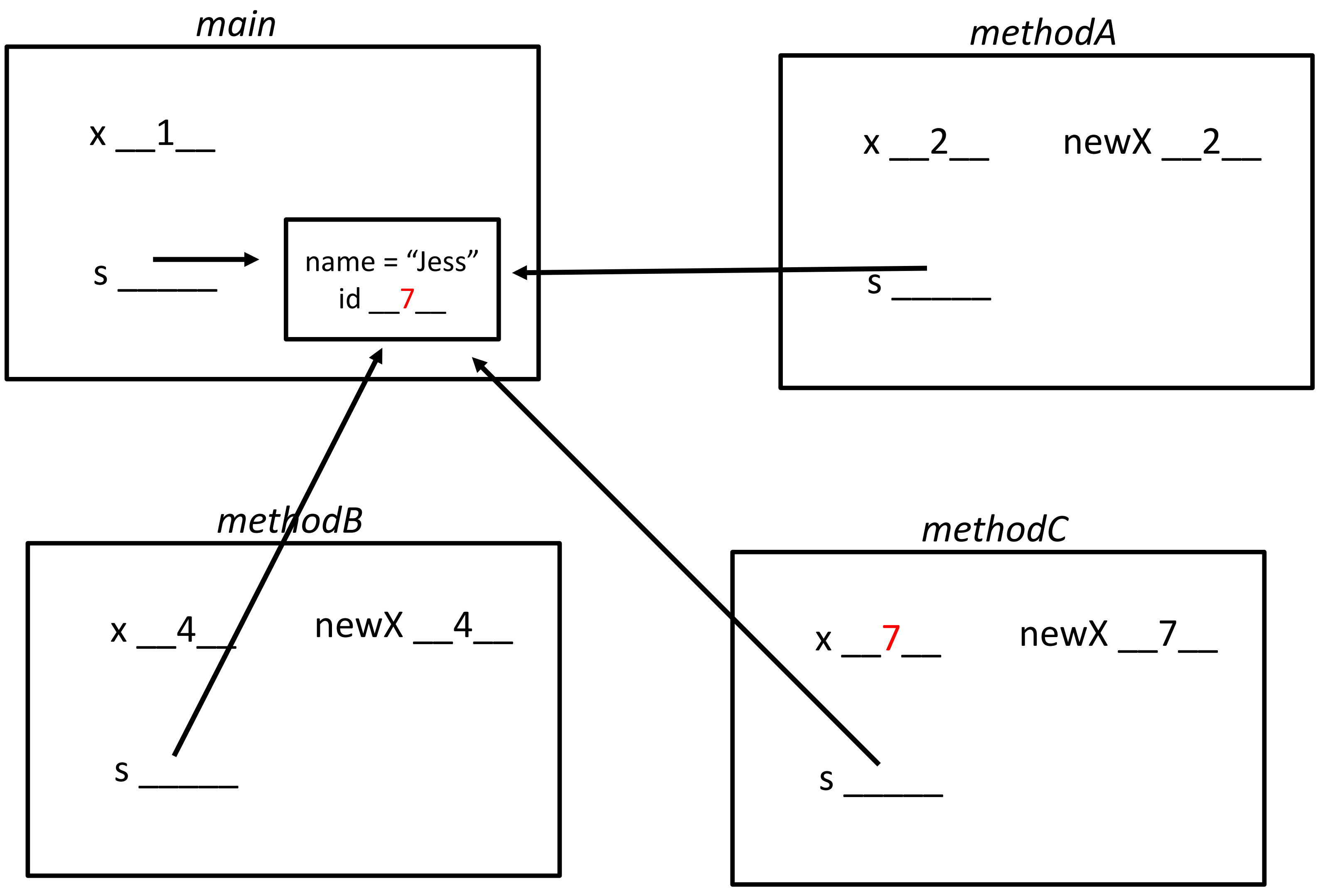
```
methodC(x, s);
```

methodC
methodB
methodA
main

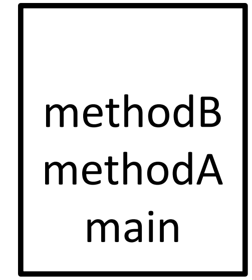


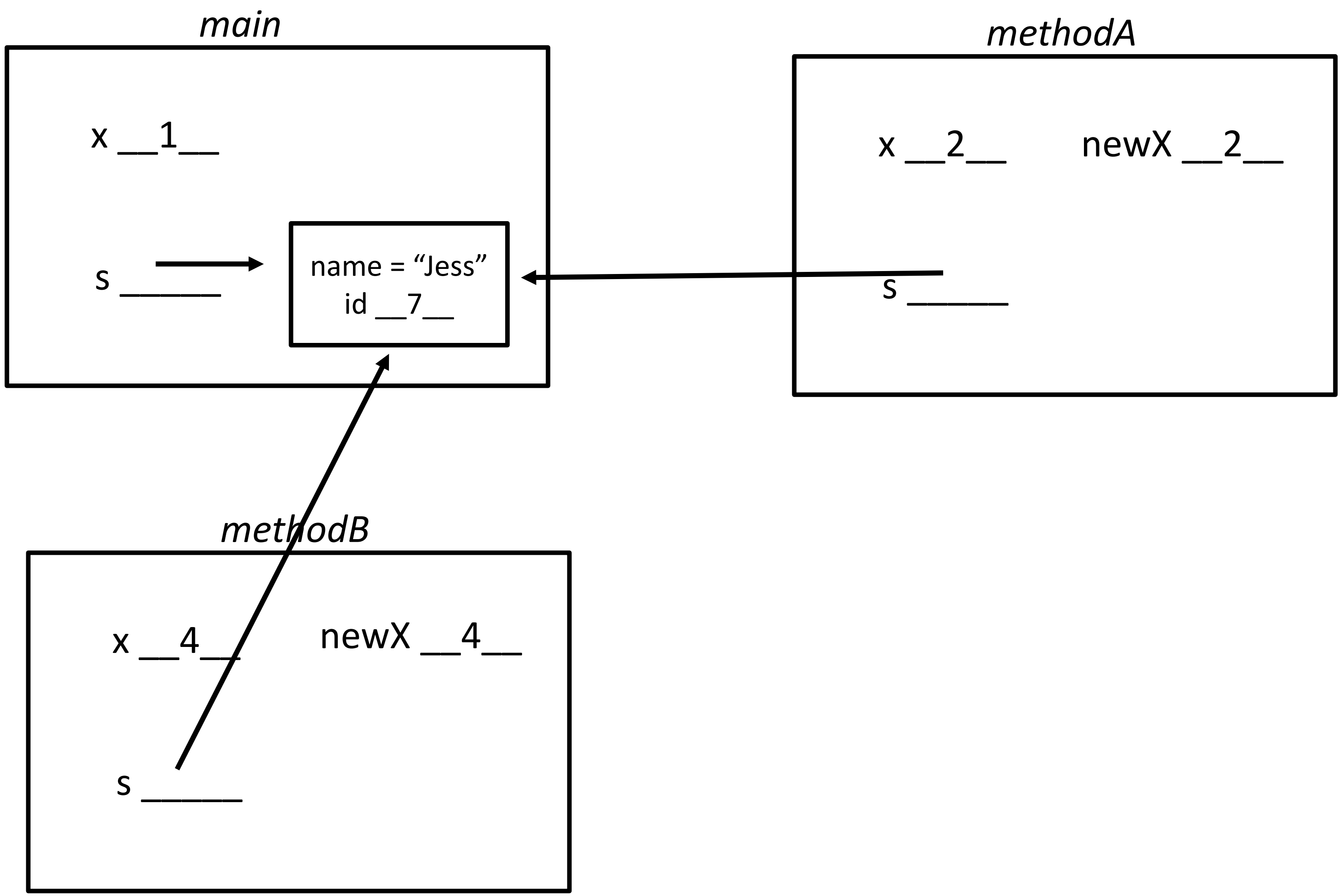
```
int newX = x + 3;
x = newX;
s.setId(x);
```

methodC
methodB
methodA
main



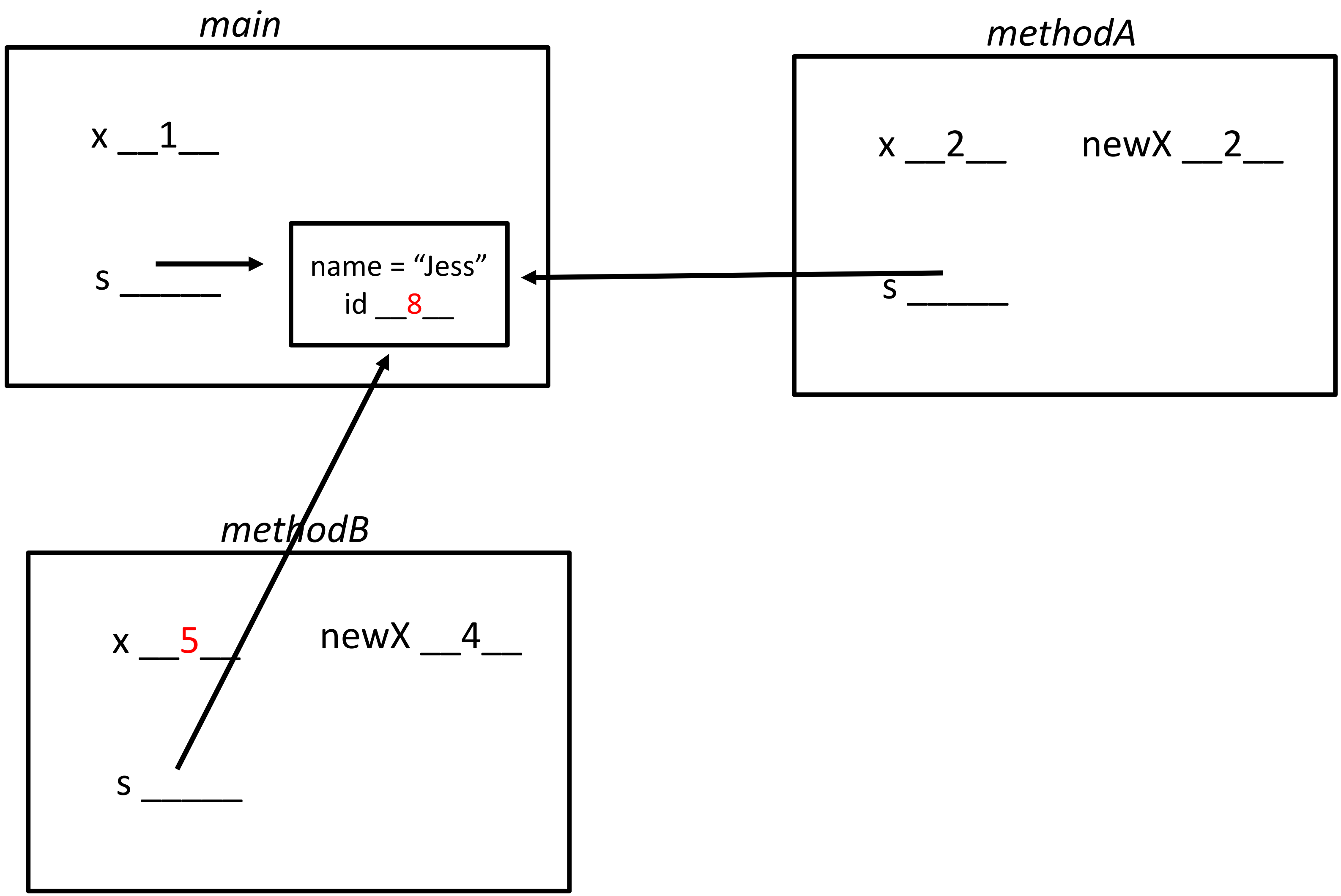
methodC ends, control returns to methodB



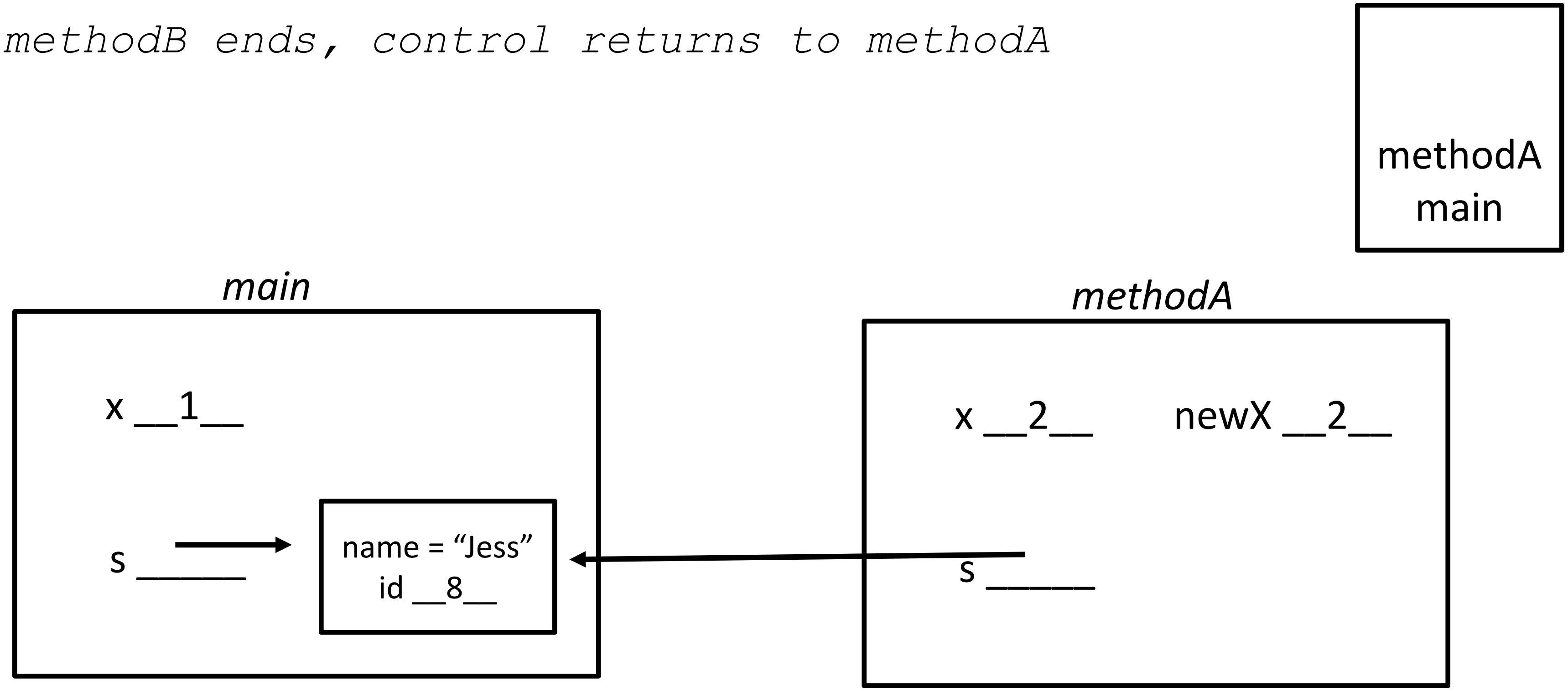


```
x = x + 1;
s.setId(s.getId()+1);
```

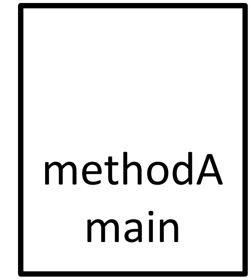
methodB
methodA
main

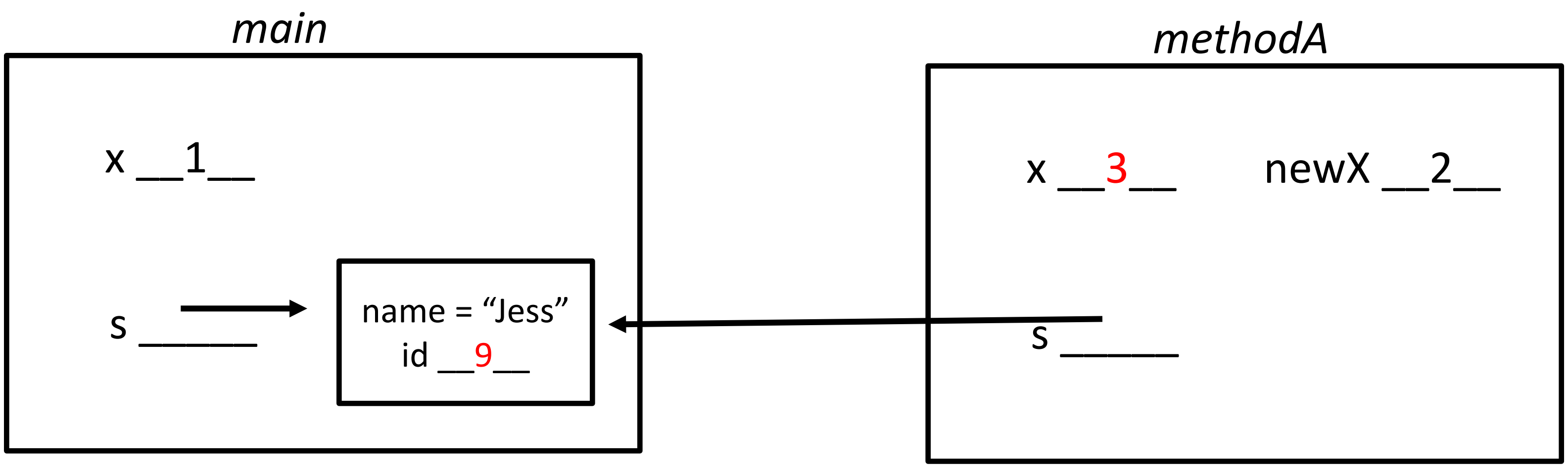


methodB ends, control returns to methodA

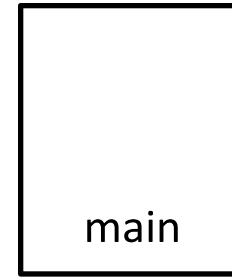


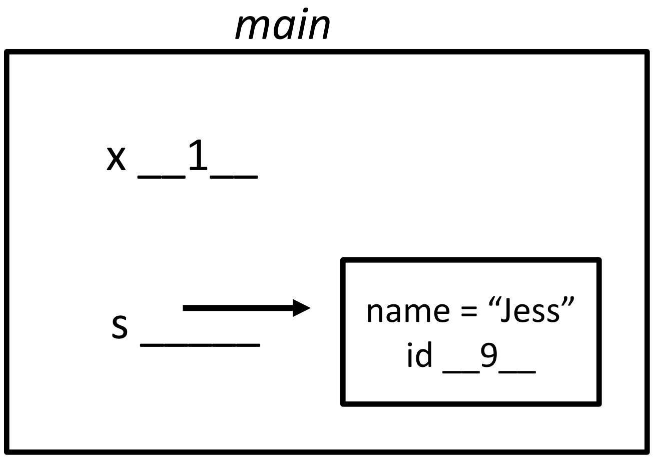
```
x = x + 1;
s.setId(s.getId()+1);
```





methodA ends, control returns to main





Recursive Method Control in Java

- The same method control rules apply!
- The only difference is that a method is invoking itself, rather than some other method.

- When a method is invoked, the current method is paused and control passes to the invoked method.
  - When that method finishes, control returns back to the original method.
  - That method picks up where it left off.
- Each call to a method sets up a new execution environment
  - New parameters
  - New local variables

Recursive Method Control in Java

- A method pauses the current execution to call itself. When control returns, you pick back up where you stopped.
- Each recursive call sets up a brand new *activation record*.
  - New parameters
  - New local variables

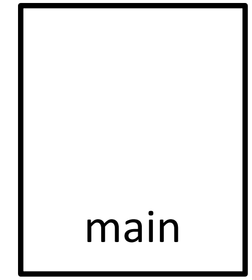
Recursive Helper Methods

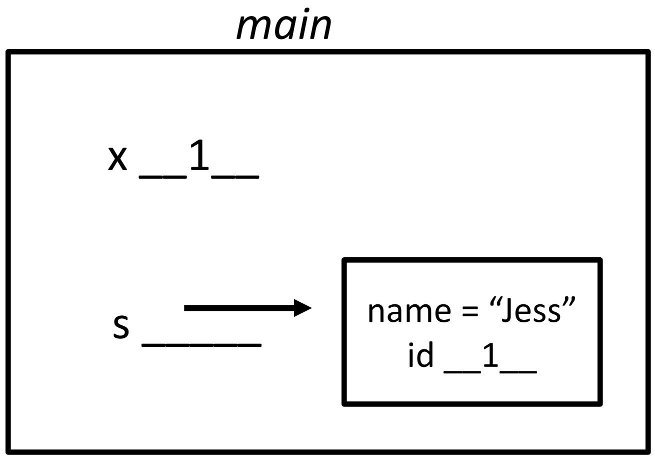
- Often, a recursive method will need additional parameters to keep track of where it is in the recursion.
- This can be done with a *helper method*.
- The helper method is invoked by the original method. The helper method is really the recursive method- it invokes itself.

Example: Recursive Method Trace

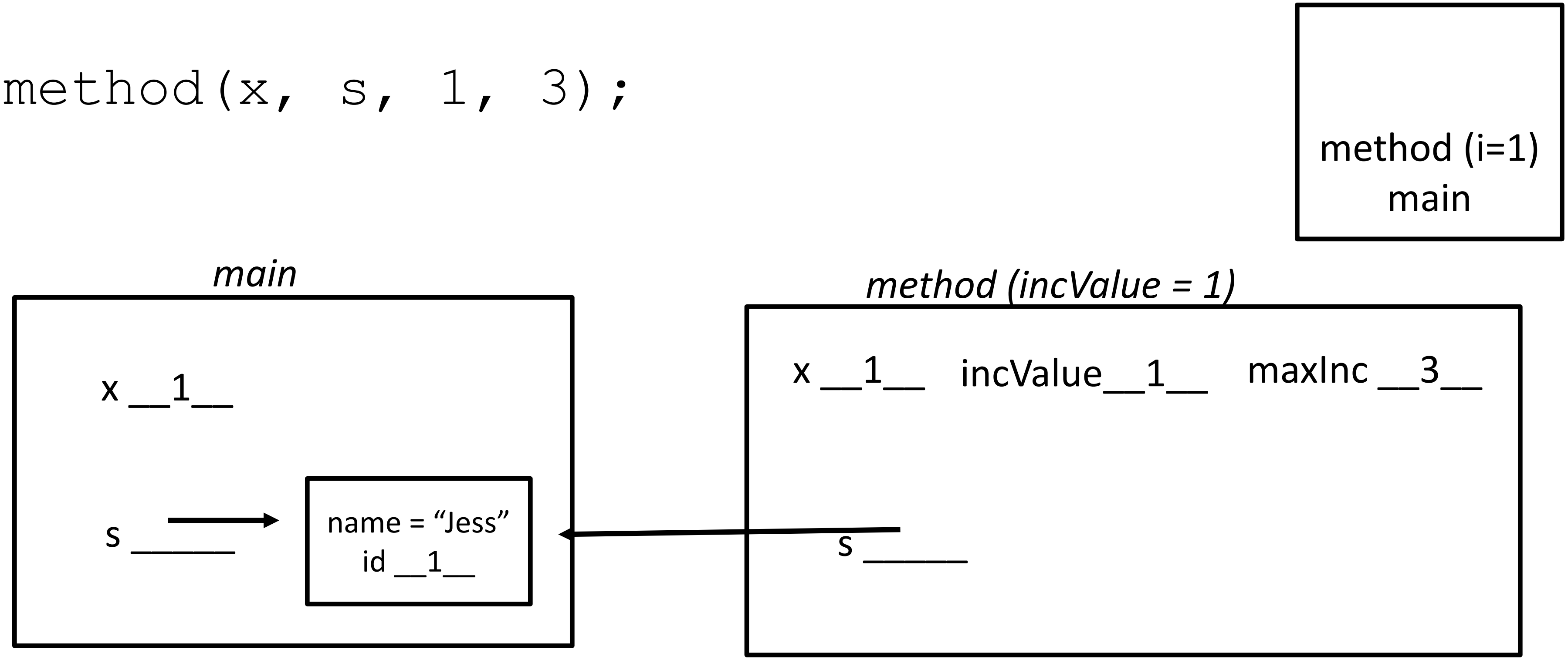
- Review the RecursiveMethodTrace example.
  - This shows how methods are called and parameters are passed when using recursion.

```
int x = 1;
Student s = new Student("Jess", 1);
```

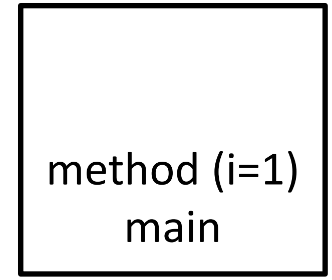


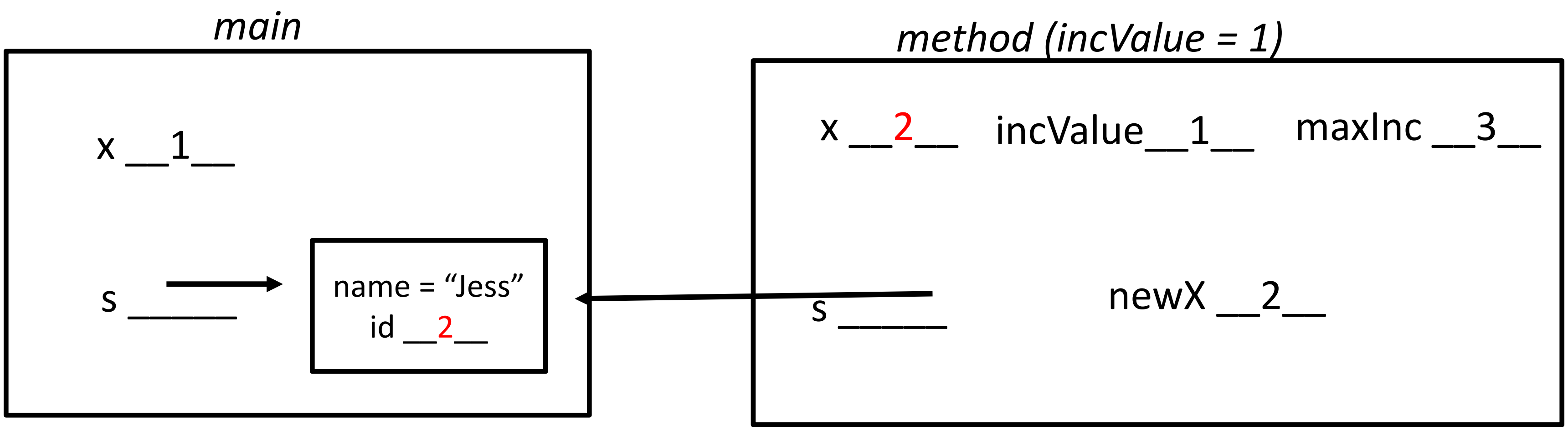


```
method(x, s, 1, 3);
```


method (i=1)
main
main
x __1__
s ______
name = “Jess”
id __1__
method (incValue = 1)
x __1__
incValue__1__
maxInc __3__
s ______

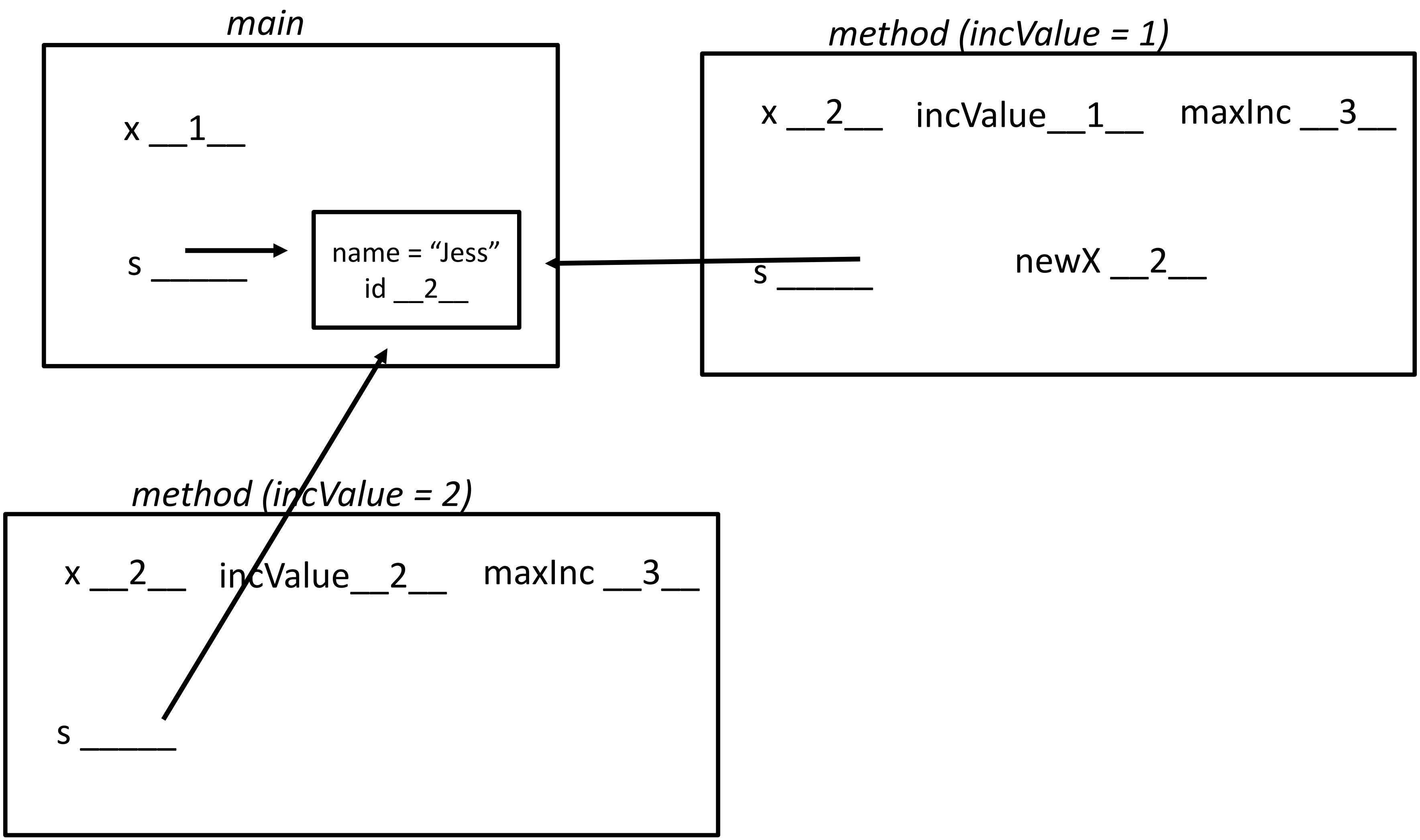
```
if (incValue <= maxInc) {
   int newX = x + incValue;
   x = newX;
   s.setId(x);
```



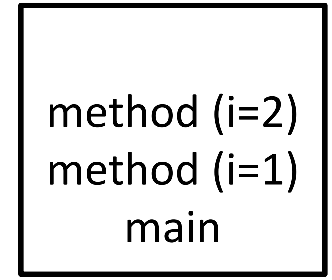


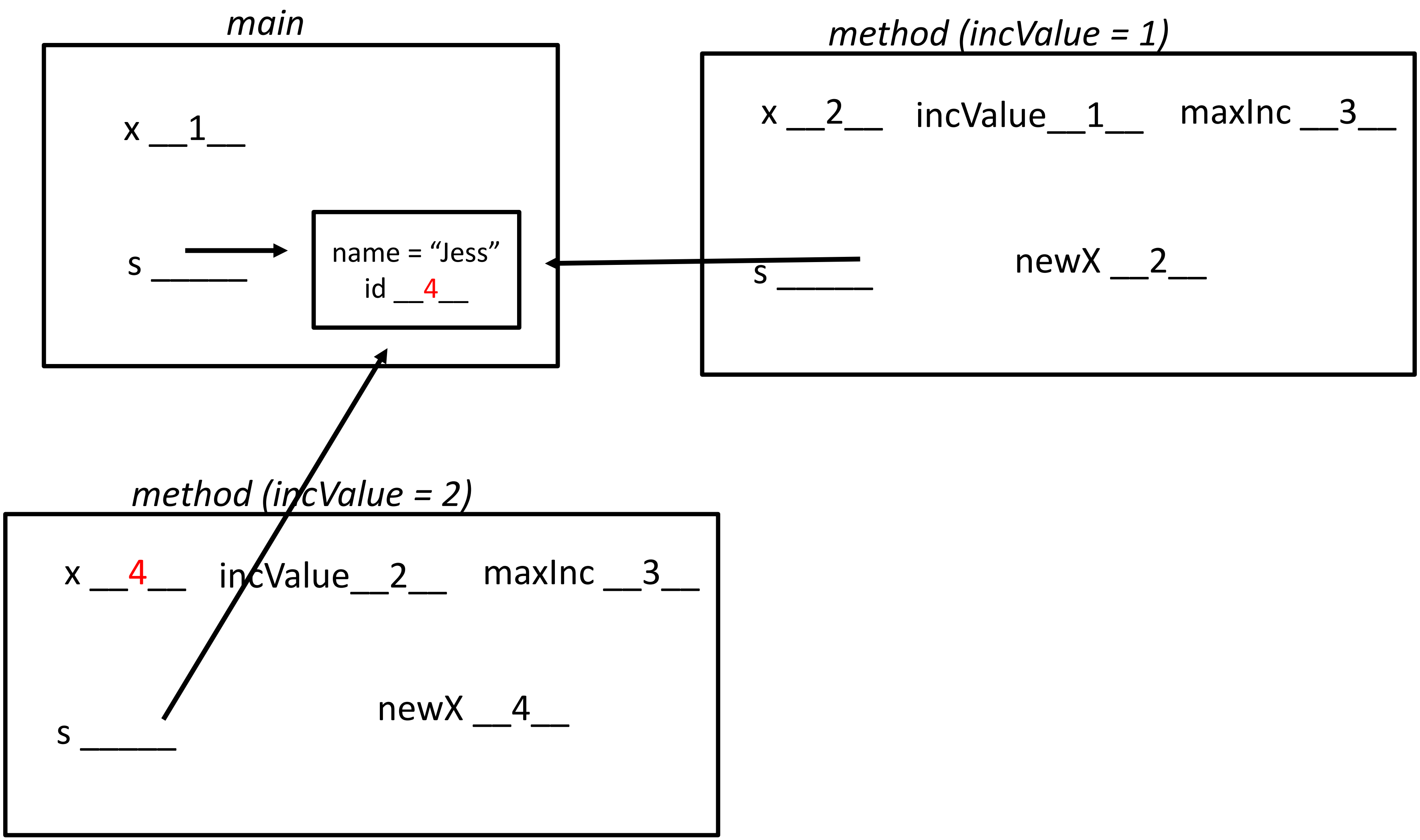
```
method(x, s, incValue + 1, maxInc);
```

method (i=2)
method (i=1)
main

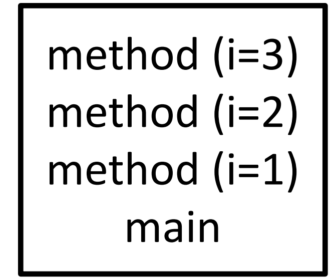


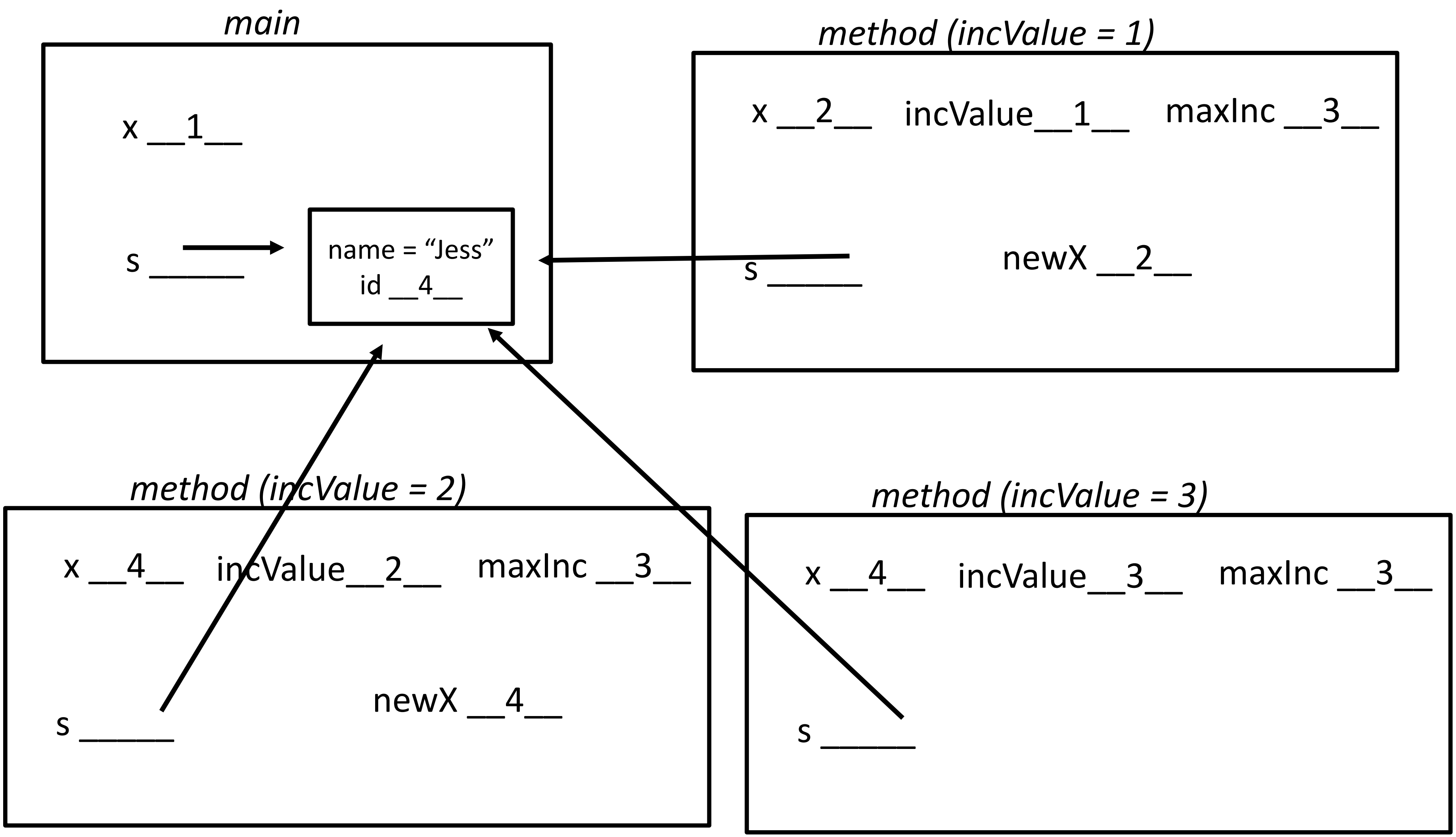
```
if (incValue <= maxInc) {
    int newX = x + incValue;
    x = newX;
    s.setId(x);
```





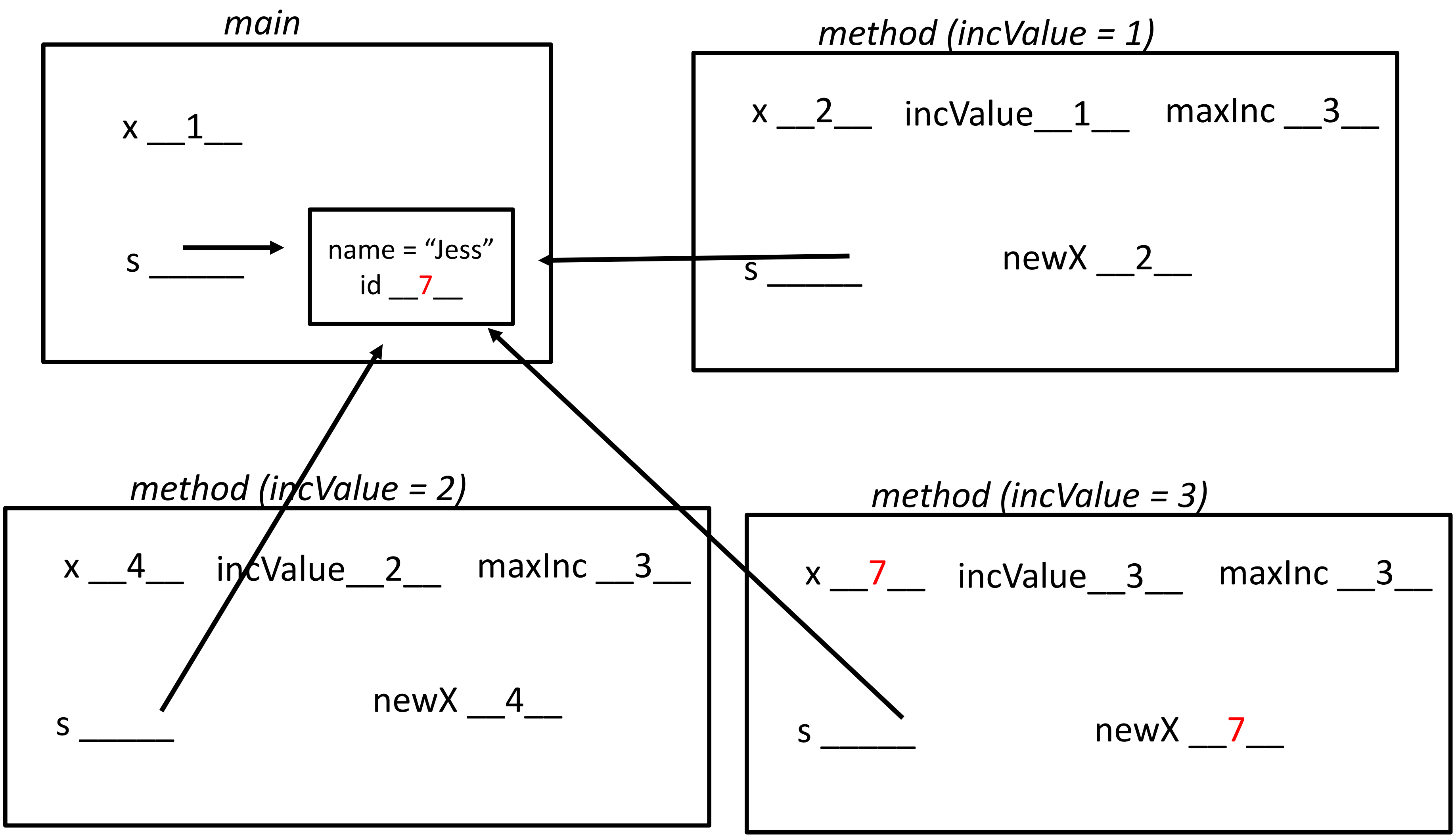
```
method(x, s, incValue + 1, maxInc);
```



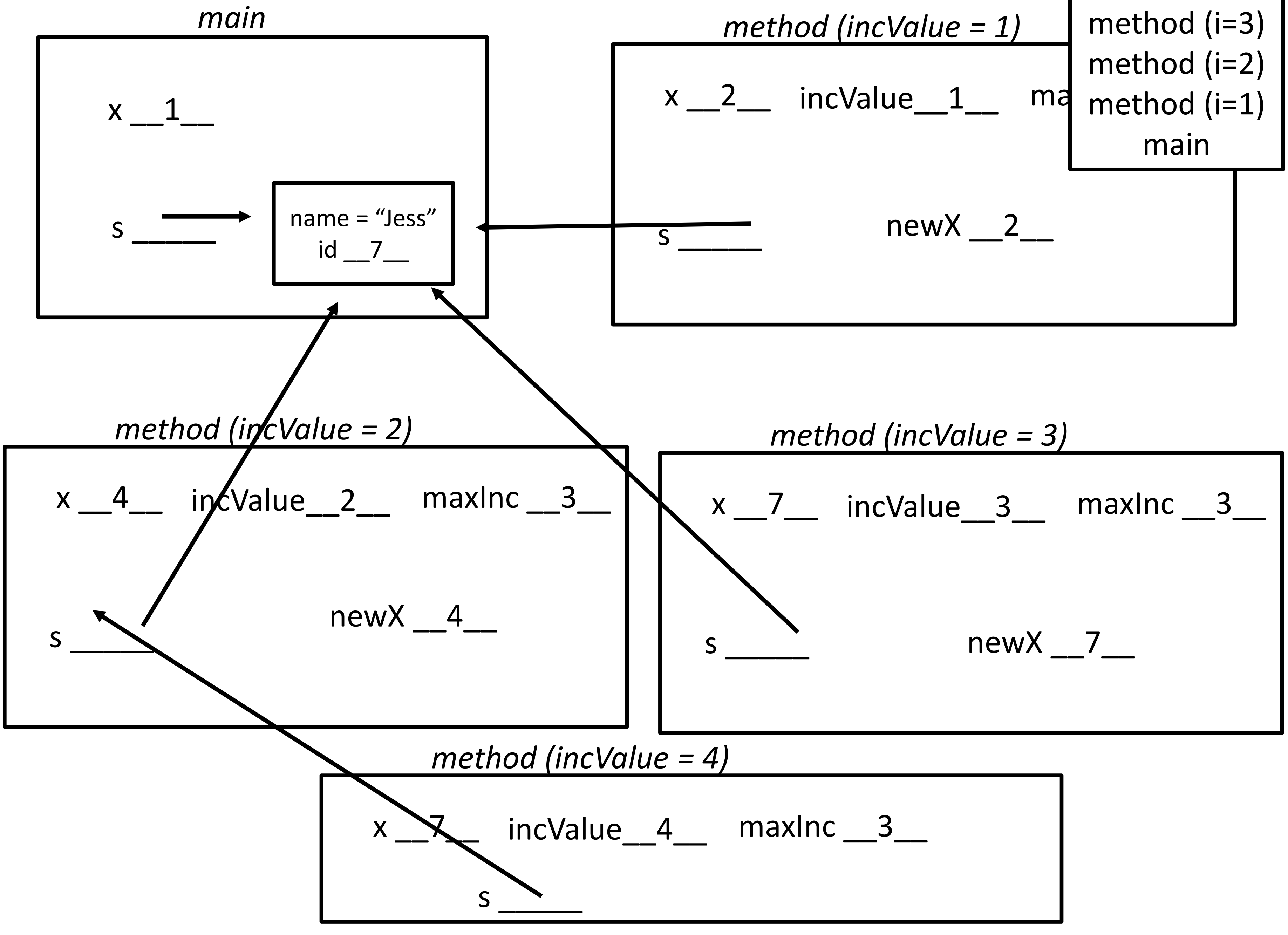


```
if (incValue <= maxInc) {
   int newX = x + incValue;
   x = newX;
   s.setId(x);
```

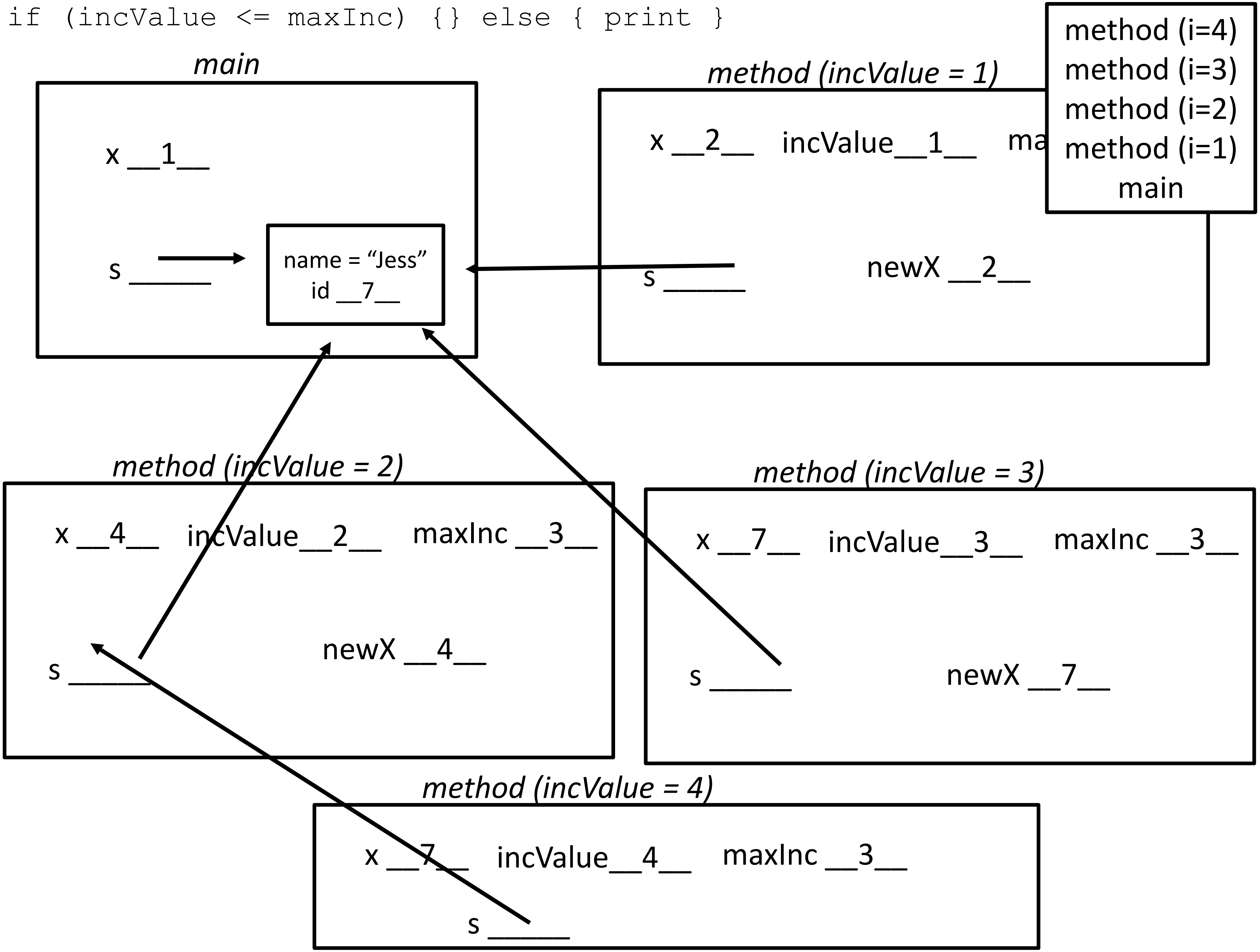
method (i=3)
method (i=2)
method (i=1)
main



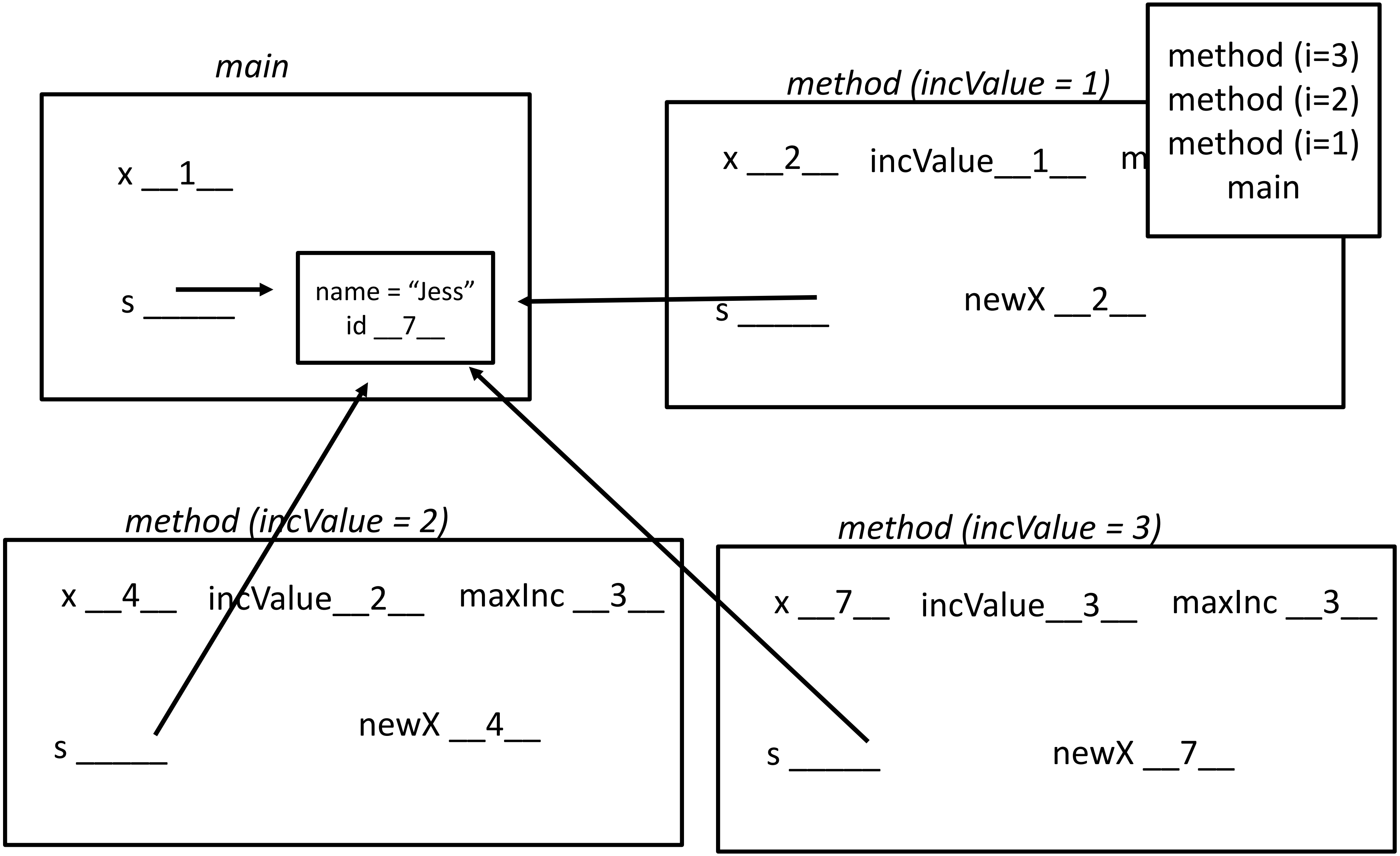
```
method(x, s, incValue + 1, maxInc);
```



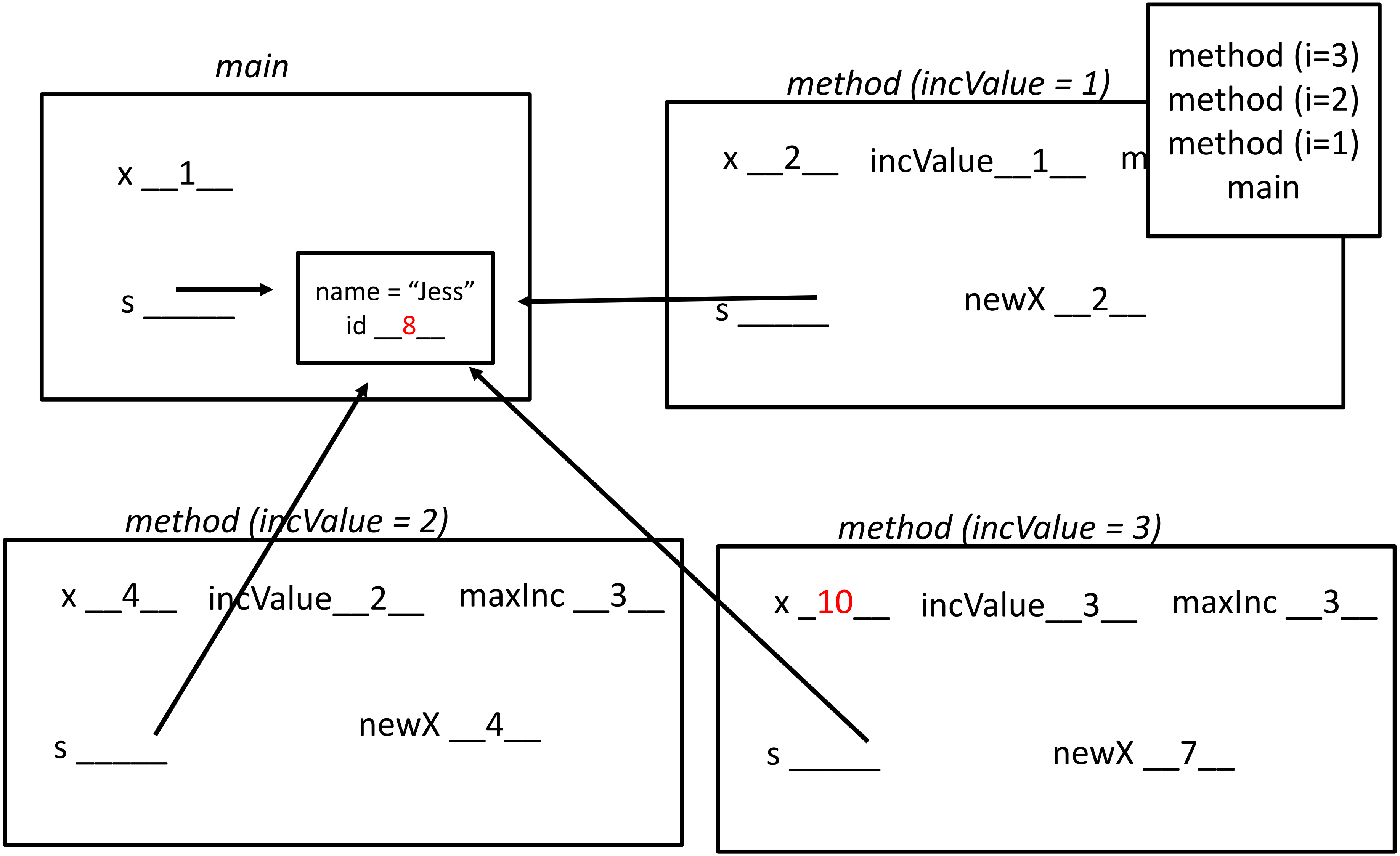
```
if (incValue <= maxInc) {} else { print }
```



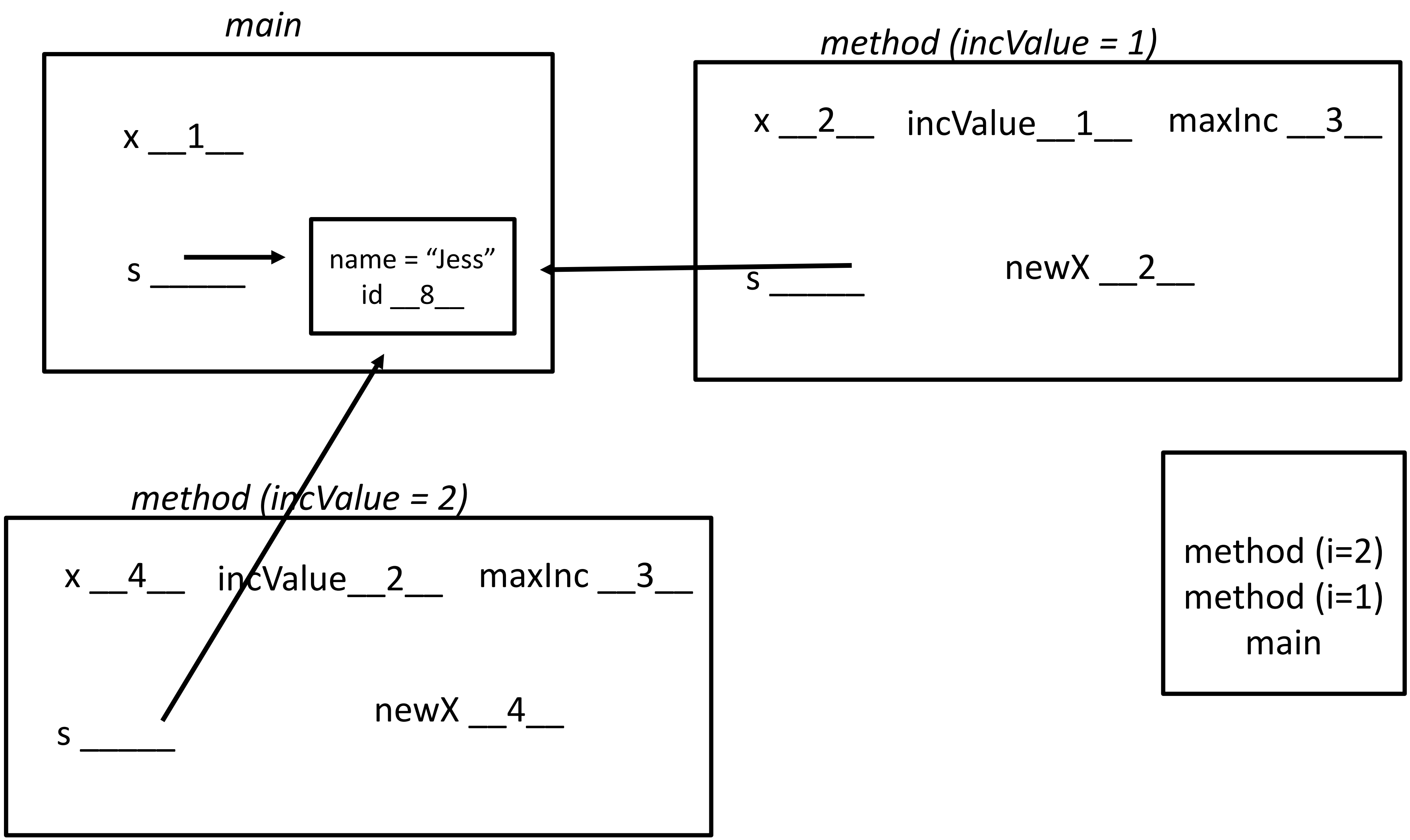
method/incValue=4 is done, activation record is popped from stack, local variables and parameters are garbage collected; control returns to method/incValue=3



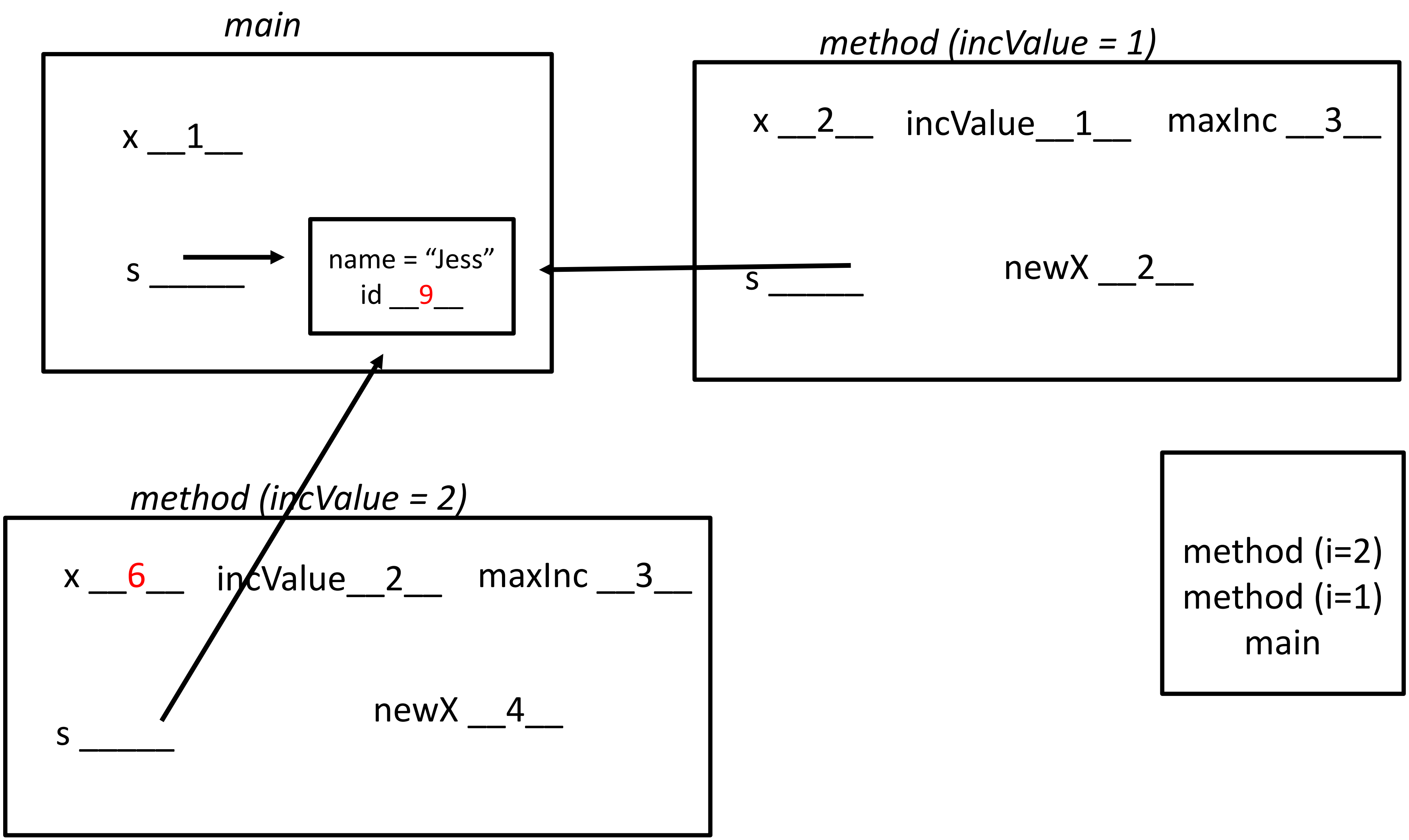
```
x = x + incValue;
s.setId(s.getId()+1);
```



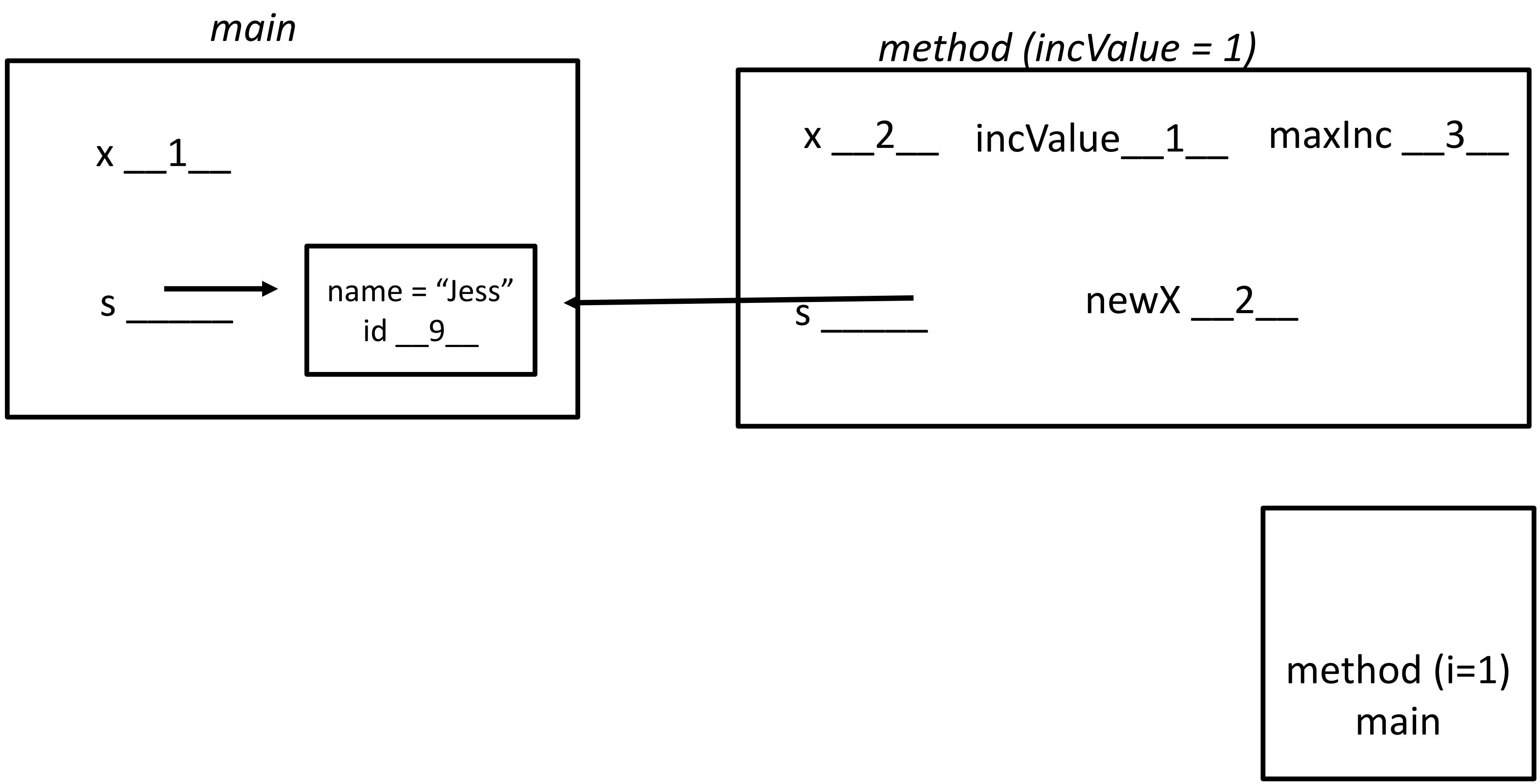
```
method/incValue=3 is done, activation record is popped from
stack, local variables and parameters are garbage collected;
control returns to method/incValue=2
```



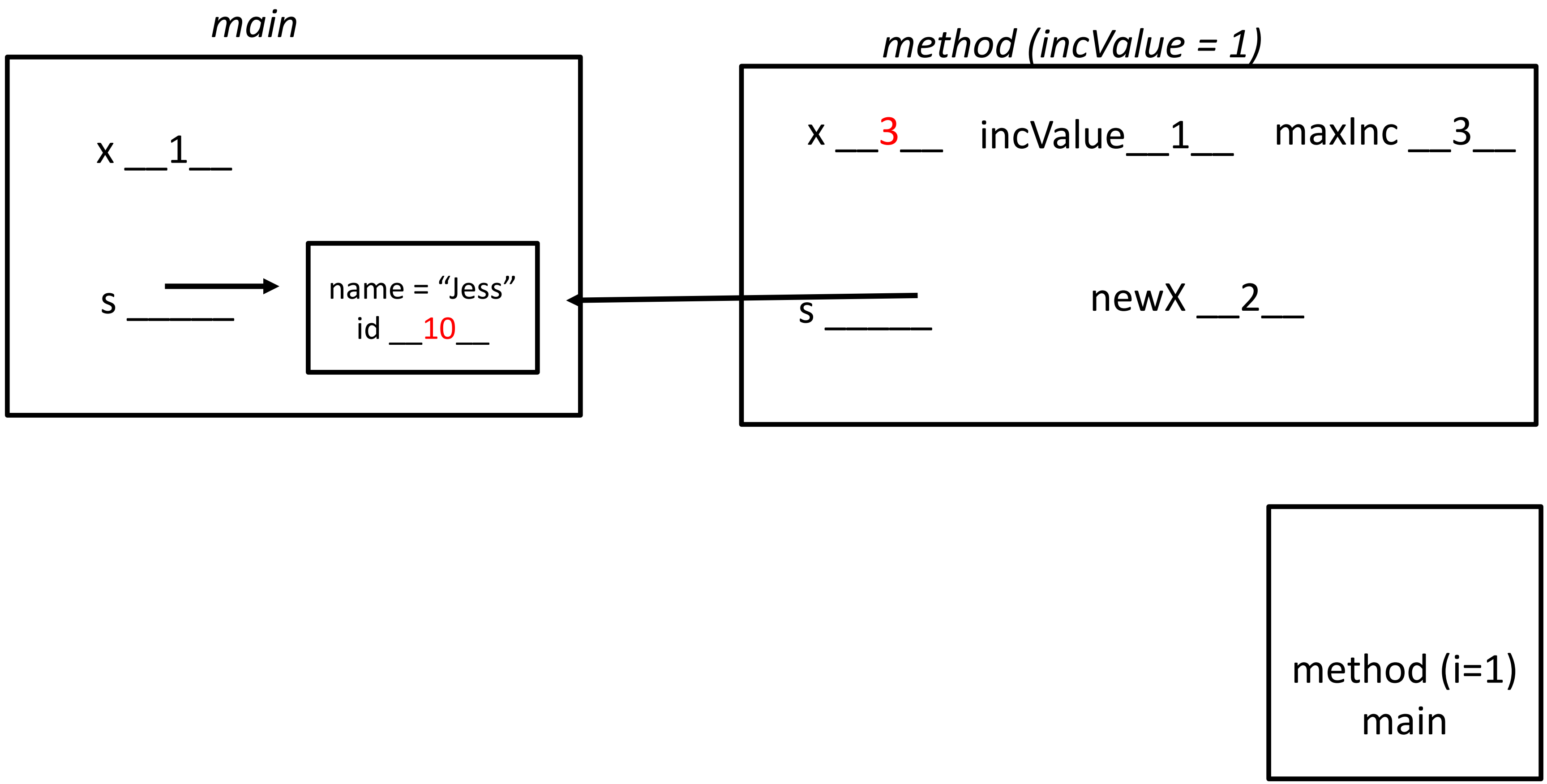
```
x = x + incValue;
s.setId(s.getId()+1);
```



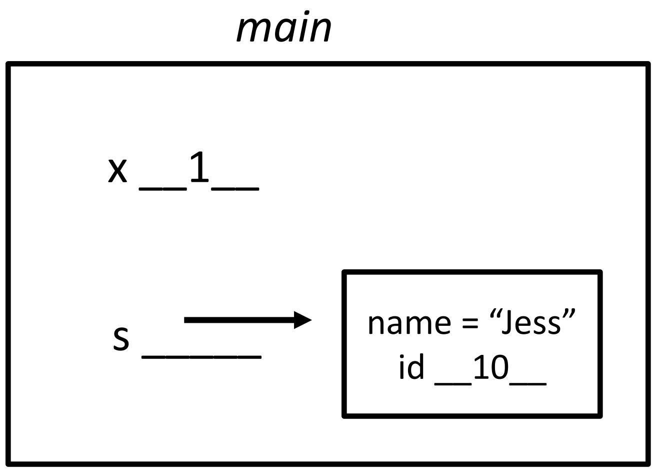
method/incValue=2 is done, activation record is popped from stack, local variables and parameters are garbage collected; control returns to method/incValue=1

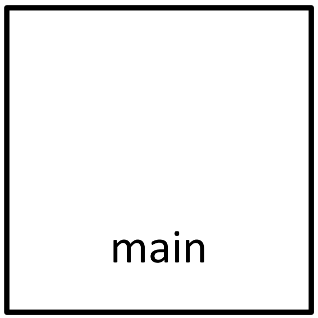


```
x = x + incValue;
s.setId(s.getId()+1);
```



```
method/incValue=1 is done, activation record is popped from
stack, local variables and parameters are garbage collected;
control returns to main; the last print happens before the
program ends
```





RECURSIVE VOID METHODS

Examples: void Methods with Strings, Arrays, and Lists

- Write a recursive method to print each character of a String.
- Write a recursive method to double the elements in an array.
- Write a recursive method to double the elements in a list.

RECURSIVE VALUED METHODS

Recursive Methods

- Recursive methods can be void or valued.

- For valued methods, it is critical to **link together** the recursive method calls.
- This is what connects or "builds up" the solution.

- You can do this with local variables or multiple return statements.

Recursive Valued Methods

- You must either:
  a) **return** the value of the recursive method call or
  b) **update** a local variable with the value of the recursive method call (and then return that local variable)
- If you don't do this step, the recursive calls are not linked together and your method will not work!
- This is a very common mistake to make!

Recursive Valued Methods

- You should **NEVER** have a call to a recursive valued method on its own.

Examples

- Write a recursive method to sum up all the numbers from 1 to some number.
- Write a recursive method to read input within a specified range and return that input.
- Write a method to return the number of times a character appears in a string.
- Write a method to sum all the values in an array.

Practice- Tracing Recursion

```
System.out.println(recMethod1(5, 1));

public int recMethod1(int x, int y) {
   if (x == y)
      return 0;
   else
      return recMethod1(x-1, y) + 1;
}
```

Practice- Tracing Recursion

```
public int recFactorial1(int x) {
    System.out.print(x);
    if (x > 1)
        return x * recFactorial1(x - 1);
    else
        return 1;
}

public int recFactorial2(int x) {
    int fac;
    if (x > 1) {
        fac = x * recFactorial2(x - 1);
        System.out.print(x);
    } else {
        fac = 1;
    }
    return fac;
}
```

Practice- Tracing Recursion

```
int[] a = {3, 2, 1, 2, 3};
System.out.println(recMethod2(a, 2, 0));
System.out.println(recMethod2(a, 2, 2));

public int recMethod2(int[] arr, int b, int c) {
    if (c < arr.length) {
        if (arr[c] != b)
            return recMethod2(arr, b, c + 1);
        else
            return 1 + recMethod2(arr, b, c + 1);
    } else {
        return 0;
    }
}
```

RECURSION AND ITERATION

Recursion and Iteration

- Any problem that can be solved with recursion can be solved with iteration.
- And vice versa.

Recursion vs. Iteration

- Just because you *can* use recursion to solve a problem, doesn't mean you *should*
- For example, the summing 1 to N problem could be implemented easily with iteration

```
int result = 0;
for(int i=1; i<=n; i++){
   result += i;
}
```

- However, for some problems, recursion provides a solution that is easier to understand

Recursion vs. Iteration

- Whether to use recursion or iteration is an important design decision
- Things to consider:
  - How clear is the solution?
  - How easy is the solution to program and test?
  - Is the solution re-using information in the best way?
  - What is the efficiency of the solution?
  - What language are you using?

Recursion and Iteration

- Write iterative solutions to some of the previous recursive examples.

Recursion and Iteration

- For many (perhaps all?!) of these examples, recursion is not needed. The iterative solution is clear, easy to write, and efficient.
- There are cases in the real world, however, where the recursive solution is much easier to write/understand.
  - These often involve more complex data structures (such as trees, graphs, etc.).

Fibonacci Numbers

- The Fibonacci numbers are used in many areas of math and science and are seen in patterns that appear in nature.
- The Fibonacci sequence begins with 0 and 1 and then continues as the sum of the preceding two numbers:

- 0 1 1 2 3 5 8 13 21 34 55 …

Fibonacci Numbers

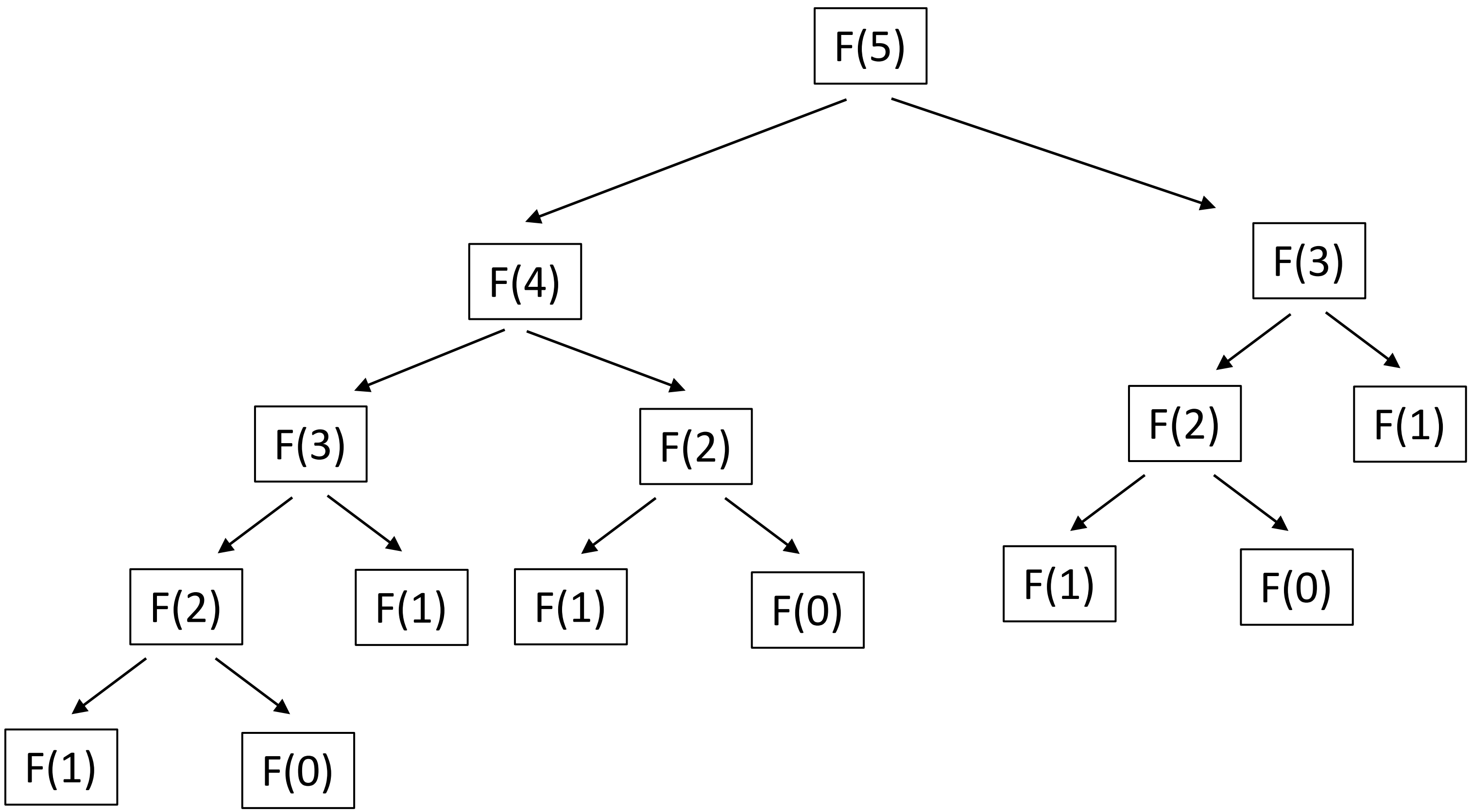
- These numbers are naturally defined recursively.

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$

Fibonacci: Recursive Solution

```
public static int fibonacciRecusive(int n) {
  if (n < 2) {
     return n;
  } else {
     return fibonacciRecusive(n-1) +
            fibonacciRecusive(n-2);
  }
}
```

Fibonacci: Recursive Call Trace



Fibonacci: Iterative Solution

```
public static int fibonacciIterative (int n) {
   int sum1 = 0, sum2 = 1;

   for(int i=0; i<n; i++) {
        int temp= sum1;
        sum1 = sum2;
        sum2 = temp + sum2;
   }
   return sum1;
}
```

Programming Fibonacci

- The recursive solution is easier to understand and is based on the mathematical definition of the algorithm.
- However, the recursive solution is $O(2^n)$ and re-calculates things many times.
- The iterative solution is linear, it will only execute n times.
  - The iterative solution is much more efficient.

Recursion vs. Iteration

- Fibonacci is an example where the recursive solution is much easier to understand, but it is **much** less efficient.
- For some problems, the recursive solution is easier to understand and will also be equally (or more) efficient.
- In Java, you always have to consider the runtime stack and the possibility for stack overflow error.

Do not mix iteration and recursion!

- A final note: do **not** mix iteration with recursion by putting the recursive call inside a loop (such as below).
- It leads to bad things... Use one or the other only!

```
public void badBadThings(int param) {
    while(condition) {
        badBadThings(param-1);
    }
} // ACK NO! DON'T DO THIS!
```

MORE EXAMPLES

More Recursion Examples

- Review the folder/file example.

More Recursion Examples

- Recursion is a natural fit with nodes.
  - Do something with the current node.
  - Pass the next node in the chain on to the recursive solution.

- Write a recursive method to print a linked chain.
  - What does the iterative solution look like?
- Write a recursive method to print a linked chain in reverse.
  - What does the iterative solution look like?

More Recursion Examples

- Add a recursive method to the AList class.
- Add a recursive method to the LList class.
- Write a recursive method that uses ListInterface from the client perspective.