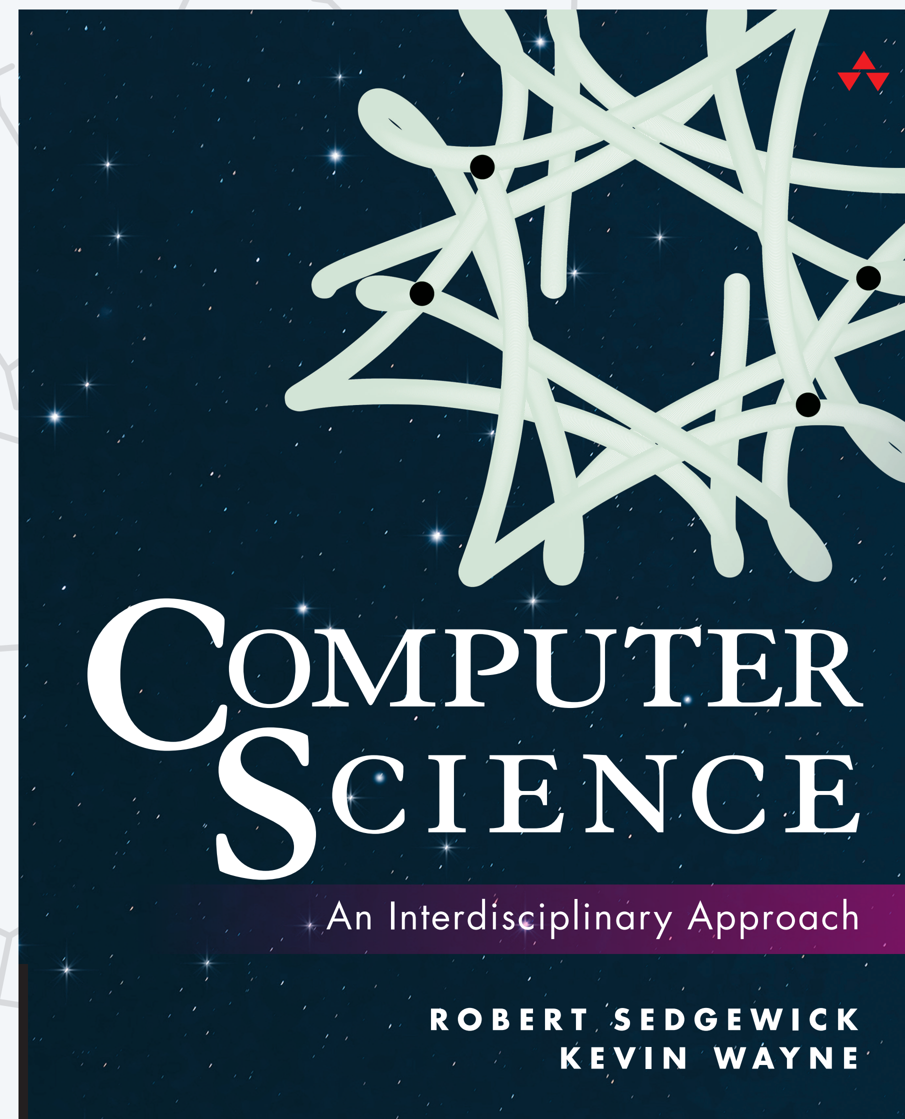
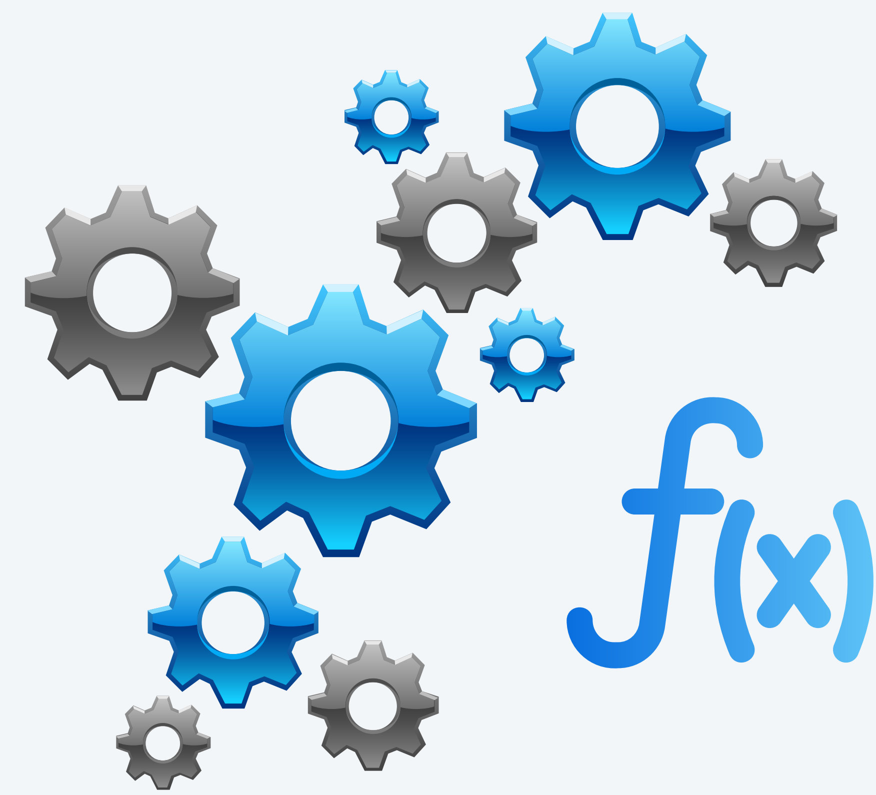


https://introcs.cs.princeton.edu

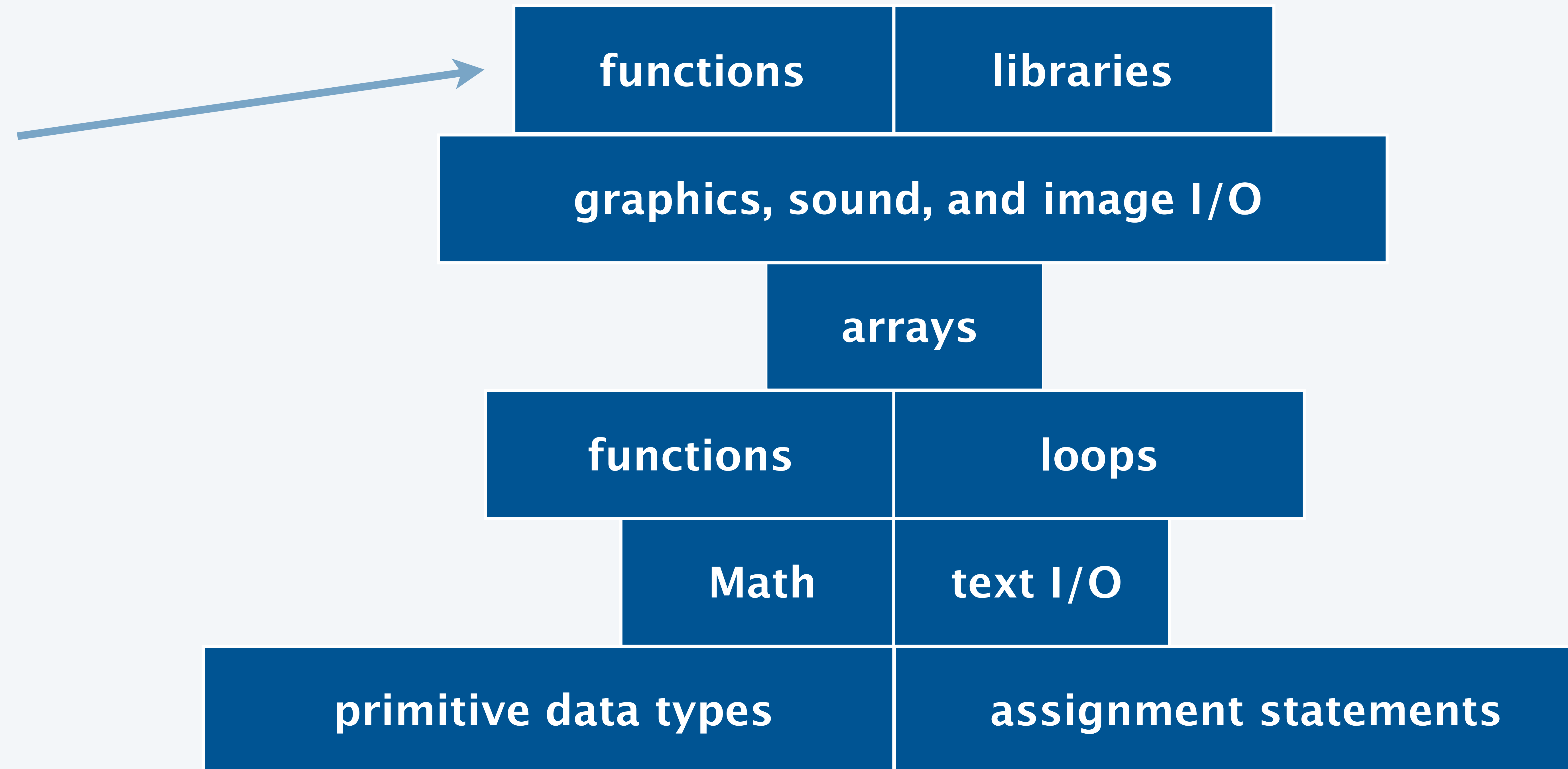
2.1 FUNCTIONS

- *under-the-hood*
- *recursion*
- *what next?*

Basic building blocks for programming

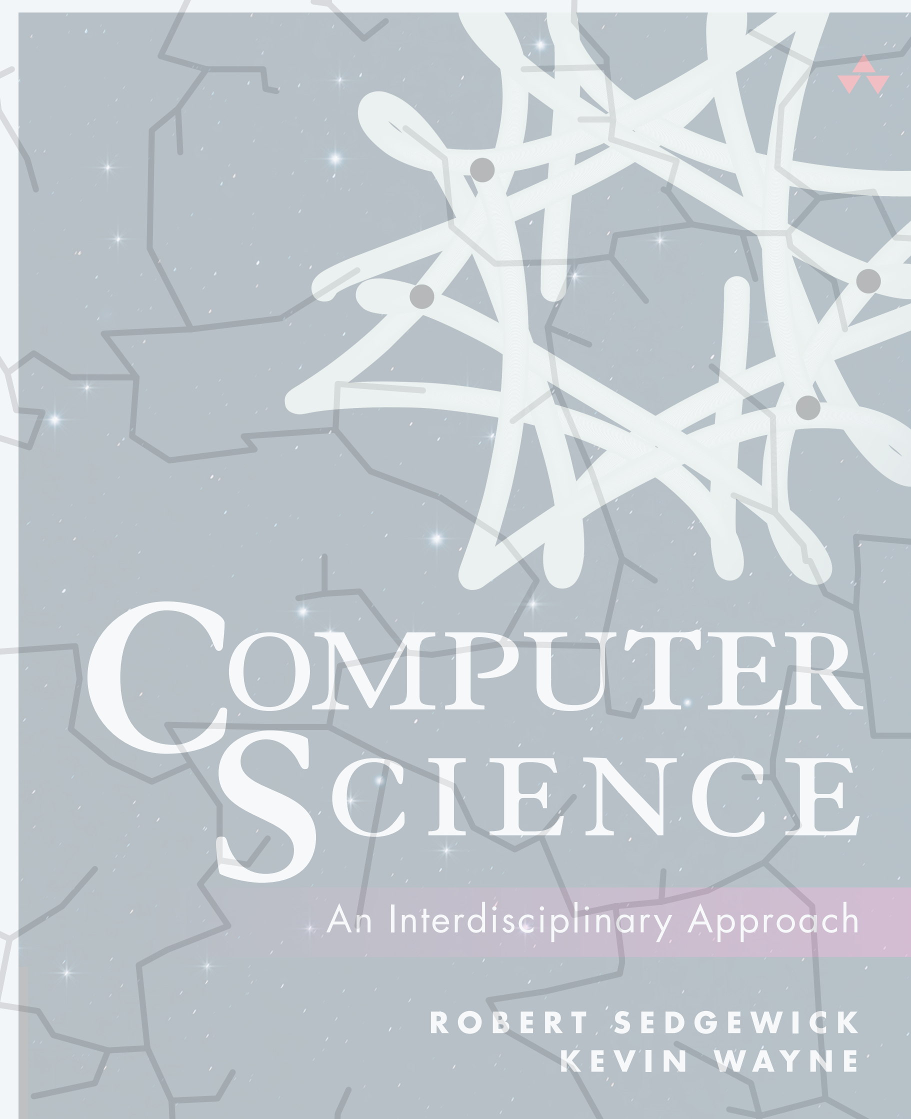


divide a program into functions



Review

1) What are functions and why should you use them?
2) What are two ways functions interact with outside world?
3) What is an API?



https://introcs.cs.princeton.edu

2.1 Functions

- *under-the-hood*
- *debugger*
- *what next?*

Parameters vs. arguments

- Parameters – set of variables functions take in (a contract)
- Arguments – actual values passed in by user of function at any given time
- Example 1: NJ DMV requires various documents to prove residency (parameters)
  - You must submit your specific documents (arguments) to get your license in return
- Example 2: Tigerfile this week requires `Synth.java` and `MySound.java` (parameters)
  - You submit your versions of `Synth.java` and `MySound.java` (arguments) to get a grade in return

Pass by value

- We grade copies of your files – this is pass by value!
- Pass by value evaluates each argument expression to produce a value.
- Java assigns (copies) each value to the corresponding parameter variable.

Pass by value

- Do not get confused by parameter/argument naming
- Variables in a function != variables in calling function

```
public static void main(String[] args) {
    int a = 5;
    int b = 4;

    int sum = addNum(a, 4 * b);
    ...
}
```

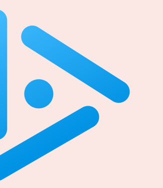
```
public static int addNum(int a, int b)
{
    b = a + b;
    return b;
}
```

Pass by value

- Do not get confused by parameter/argument naming
- Variables in a function != variables in calling function

```
public static void main(String[] args) {
   int a = 5;
   int b = 4;

   int sum = addNum(a, 4 * b);
   ...
}
```

```
public static int addNum(int x, int y)
{
    y = x + y;
    return y;
}
```



What does the following program print?

A. -126

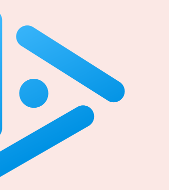
B. 126

C. Compile-time error.

D. Run-time error.

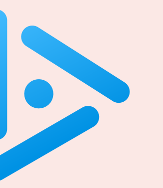
```
public class Mystery {

   public static void negate(int a) {
      a = -a;
   }

   public static void main(String[] args) {
      int a = 126;
      negate(a);
      StdOut.println(a);
   }
}
```

Functions: quiz 1



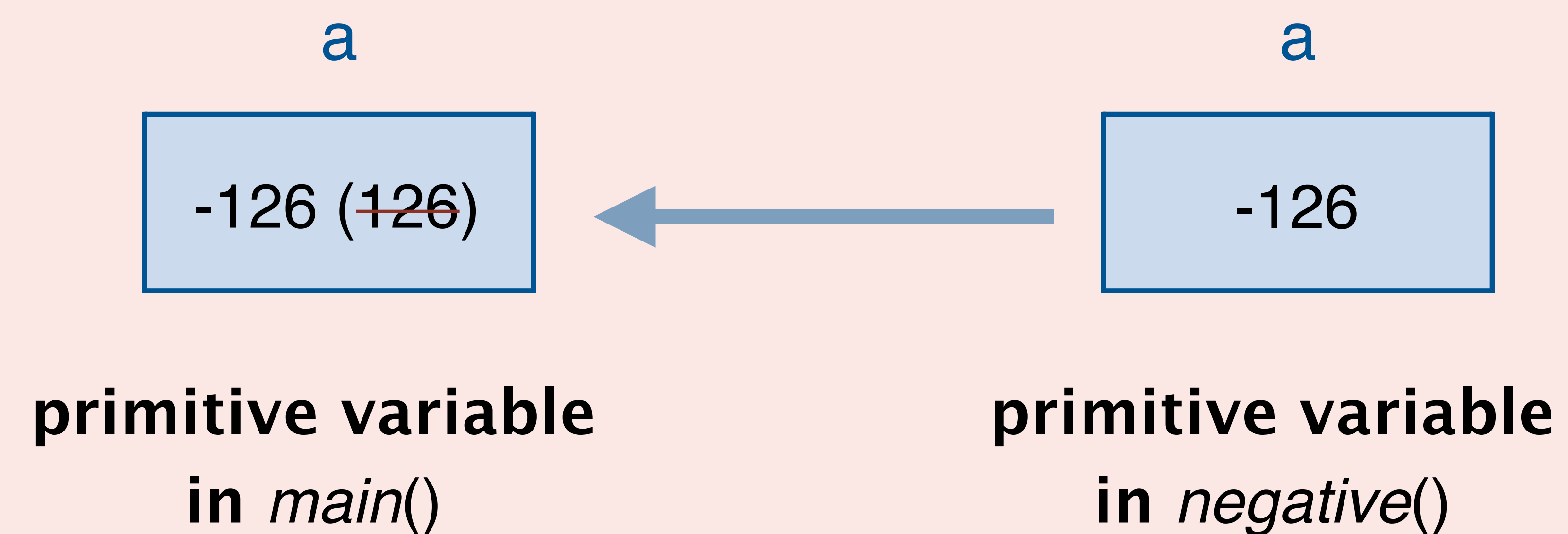
How would we negate original a?

```
public class Mystery {

   public static void negate(int a) {
      a = -a;
   }

   public static void main(String[] args) {
      int a = 126;
      negate(a);
      StdOut.println(a);
   }
}
```



How would we negate original a?

```
public class Mystery {

   public static void negate(int a) {
      a = -a;
      return a;
   }

   public static void main(String[] args) {
      int a = 126;
      negate(a);
      a = negate(a);
      StdOut.println(a);
   }
```

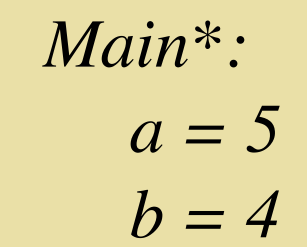


Function call stack

- Each function call requires memory for new arguments and variables
- Call stack is how this function memory is arranged
- Stacks are Last In, First Out (LIFO)

Visualizing call stack

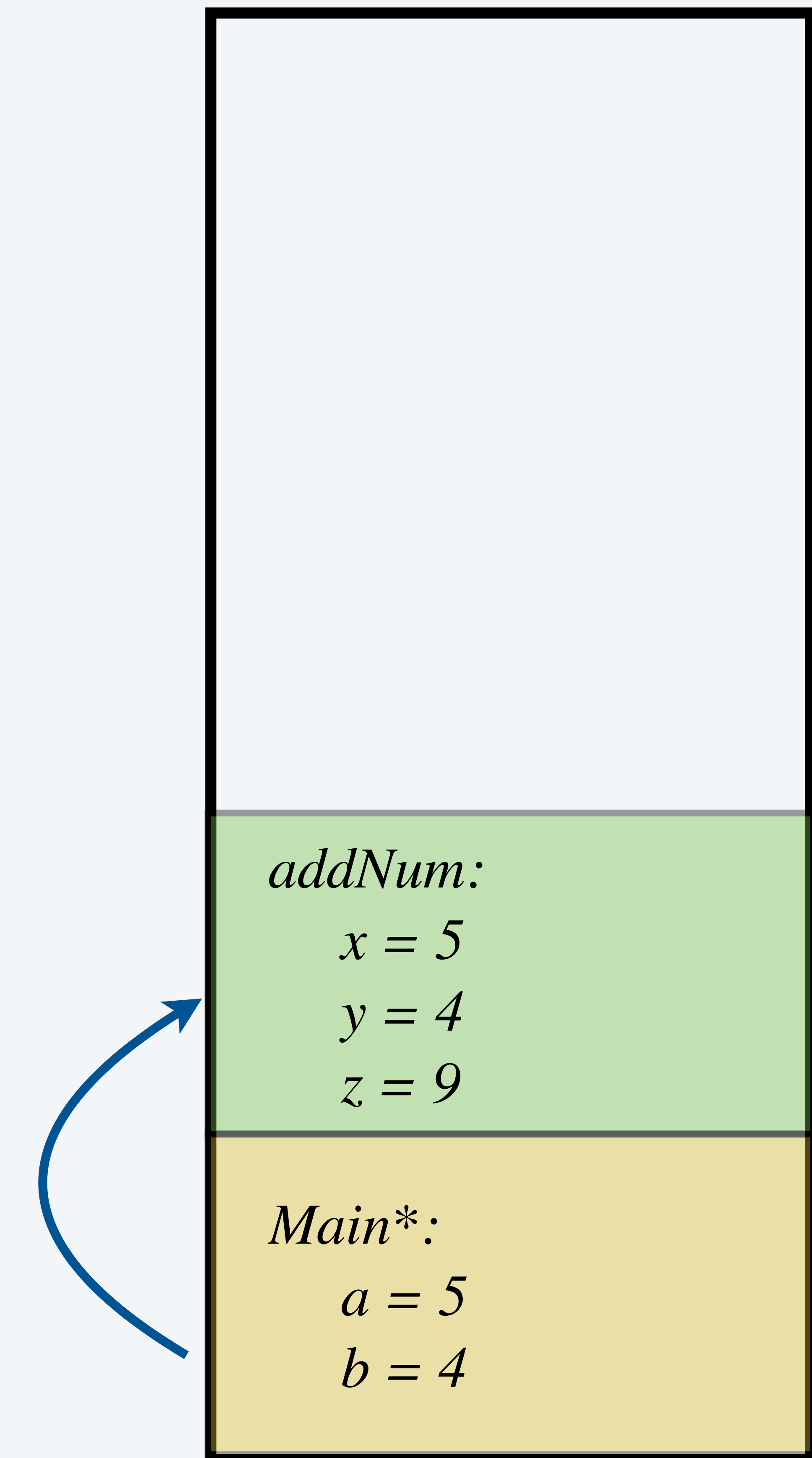
```java
public static int addNum(int x, int y) {
    int z = x + y;
    return z;

public static void main(String[] args) {
    int a = 5;
    int b = 4;

    a = addNum(a, b);
}
```



* ignore args[] for now

Visualizing call stack

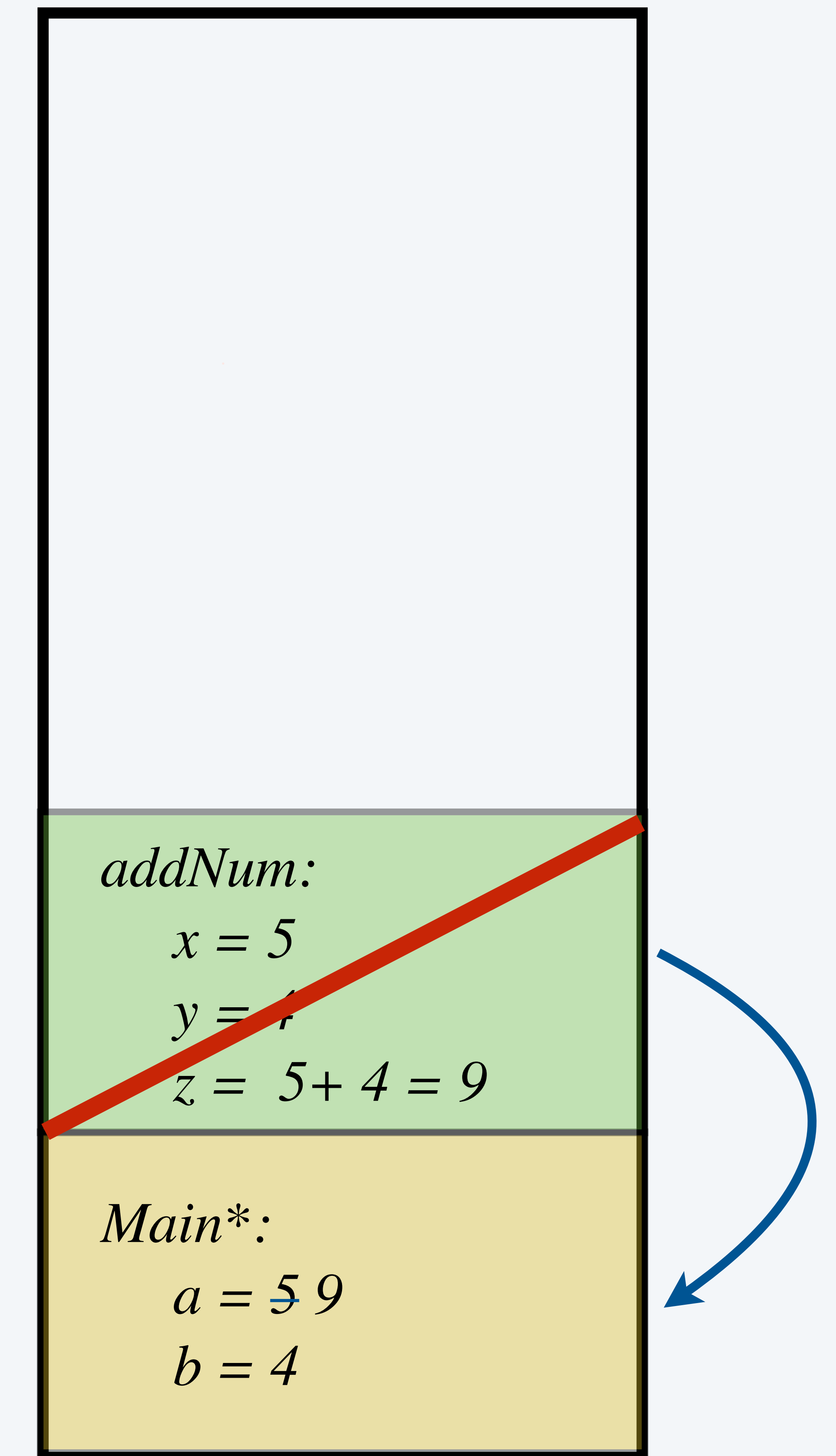
```
public static int addNum(int x, int y) {
   int z = x + y;
   return z;

public static void main(String[] args) {
   int a = 5;
   int b = 4;

   a = addNum(a, b);
}
```



* ignore args[] for now

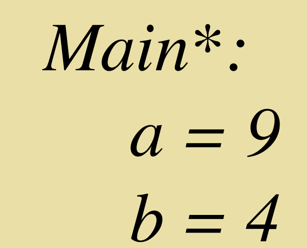
Visualizing call stack

```
public static int addNum(int x, int y) {
   int z = x + y;
   return z;

public static void main(String[] args) {
   int a = 5;
   int b = 4;

   a = addNum(a, b);
}
```



Visualizing call stack

```java
public static int addNum(int x, int y) {
   int z = x + y;
   return z;

public static void main(String[] args) {
   int a = 5;
   int b = 4;

   a = addNum(a, b);
}
```

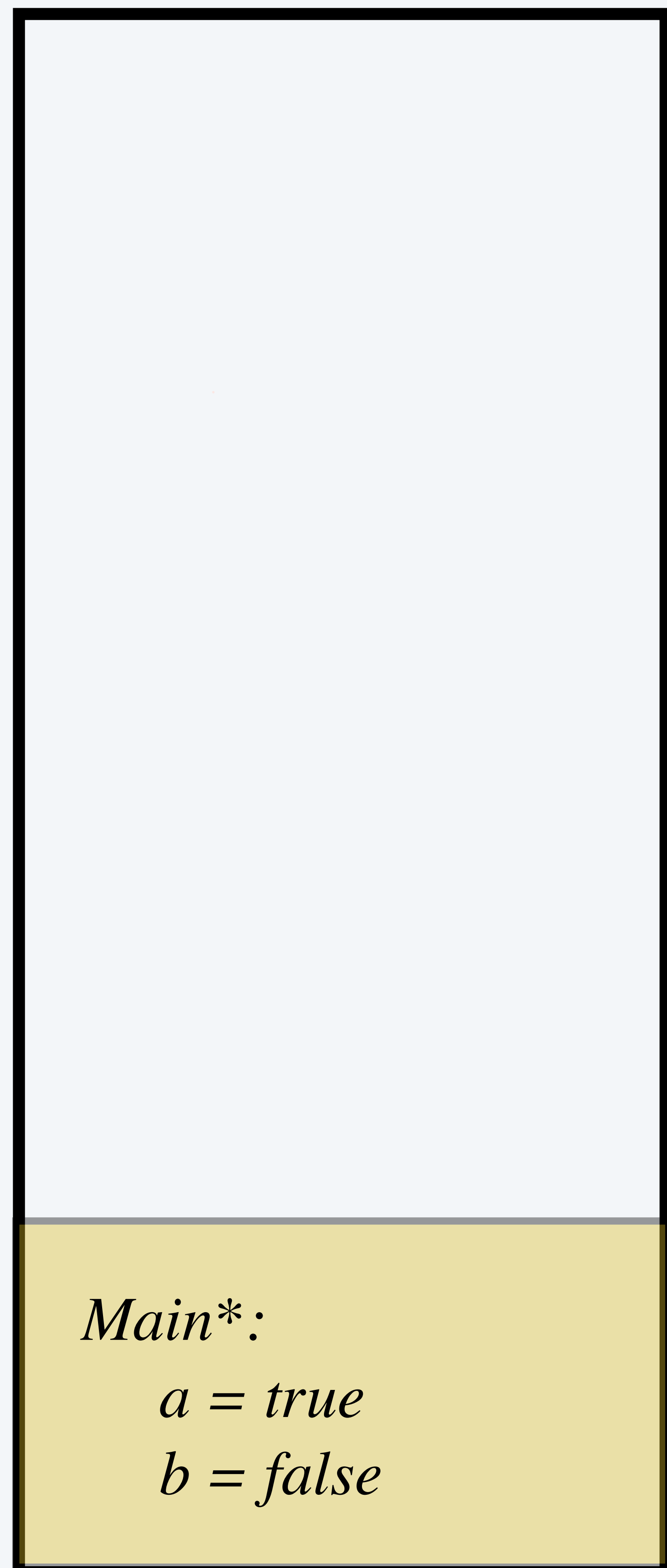


** ignore args[] for now*

Visualizing call stack

- Recreate `!(a && b)` without && and !

```
public static boolean not(boolean x){
   if (x == false)
        return true;
   return false;
}
public static boolean notAnd(boolean x, boolean y) {
   if (x == y)
        return not(true);
   return not(false);
}
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = notAnd(a, b);
}
```

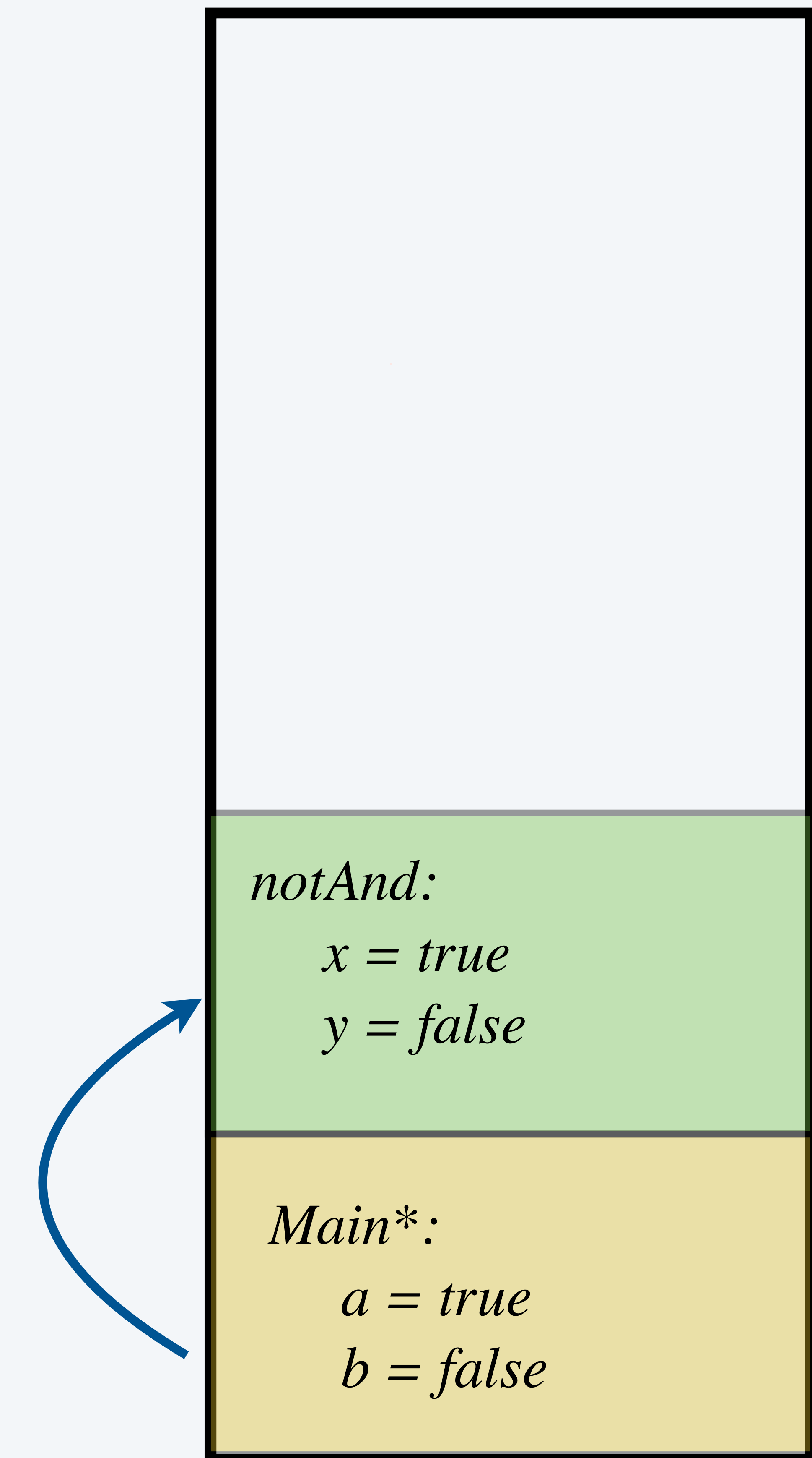


* ignore args[] for now

Visualizing call stack

- Recreate `!(a && b)` without `&&` and `!`

```
public static boolean not(boolean x){
   if (x == false)
        return true;
   return false;
}
public static boolean notAnd(boolean x, boolean y) {
   if (x == y)
        return not(true);
   return not(false);
}
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = notAnd(a, b);
}
```

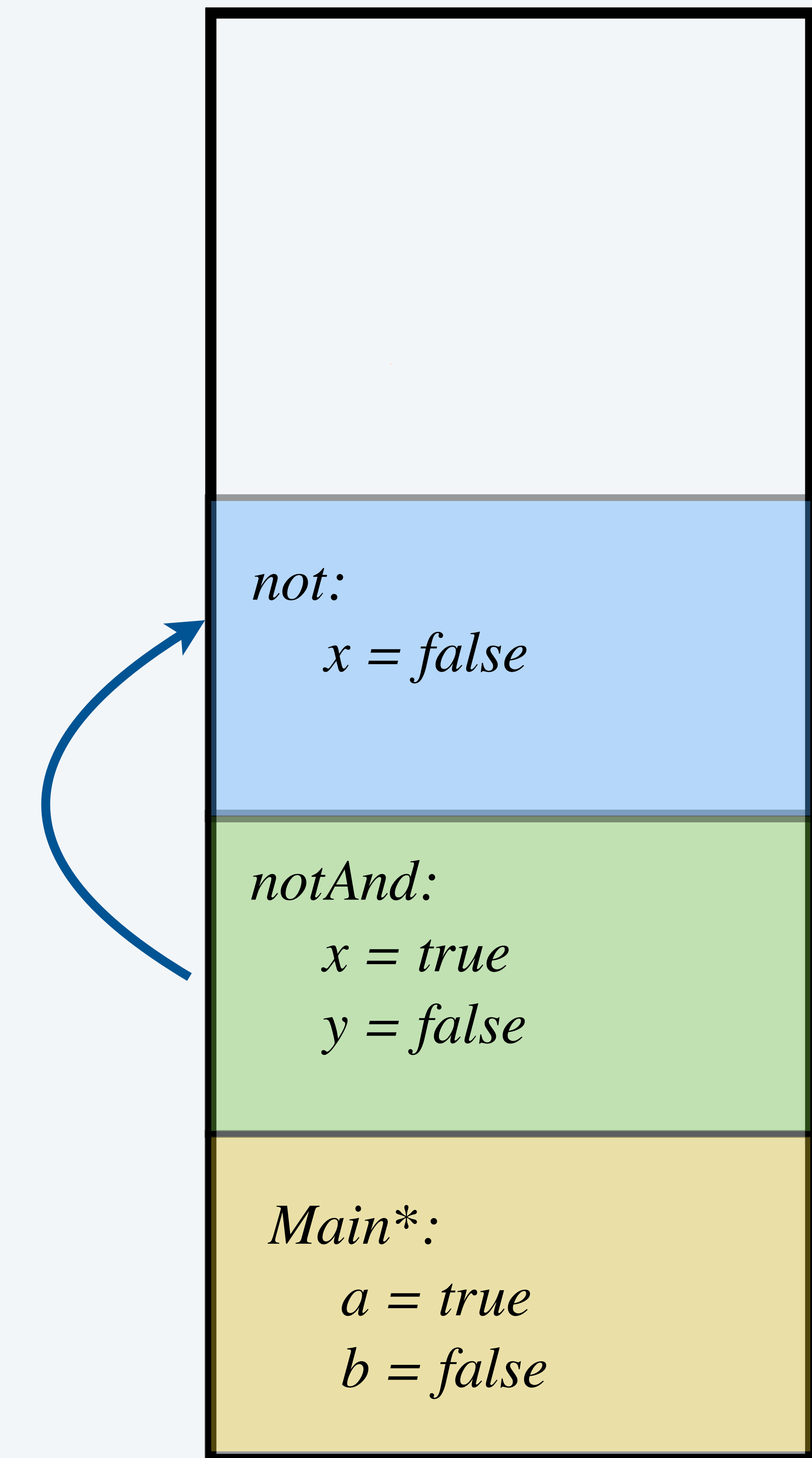


* ignore args[] for now

Visualizing call stack

- Recreate `!(a && b)` without && and !

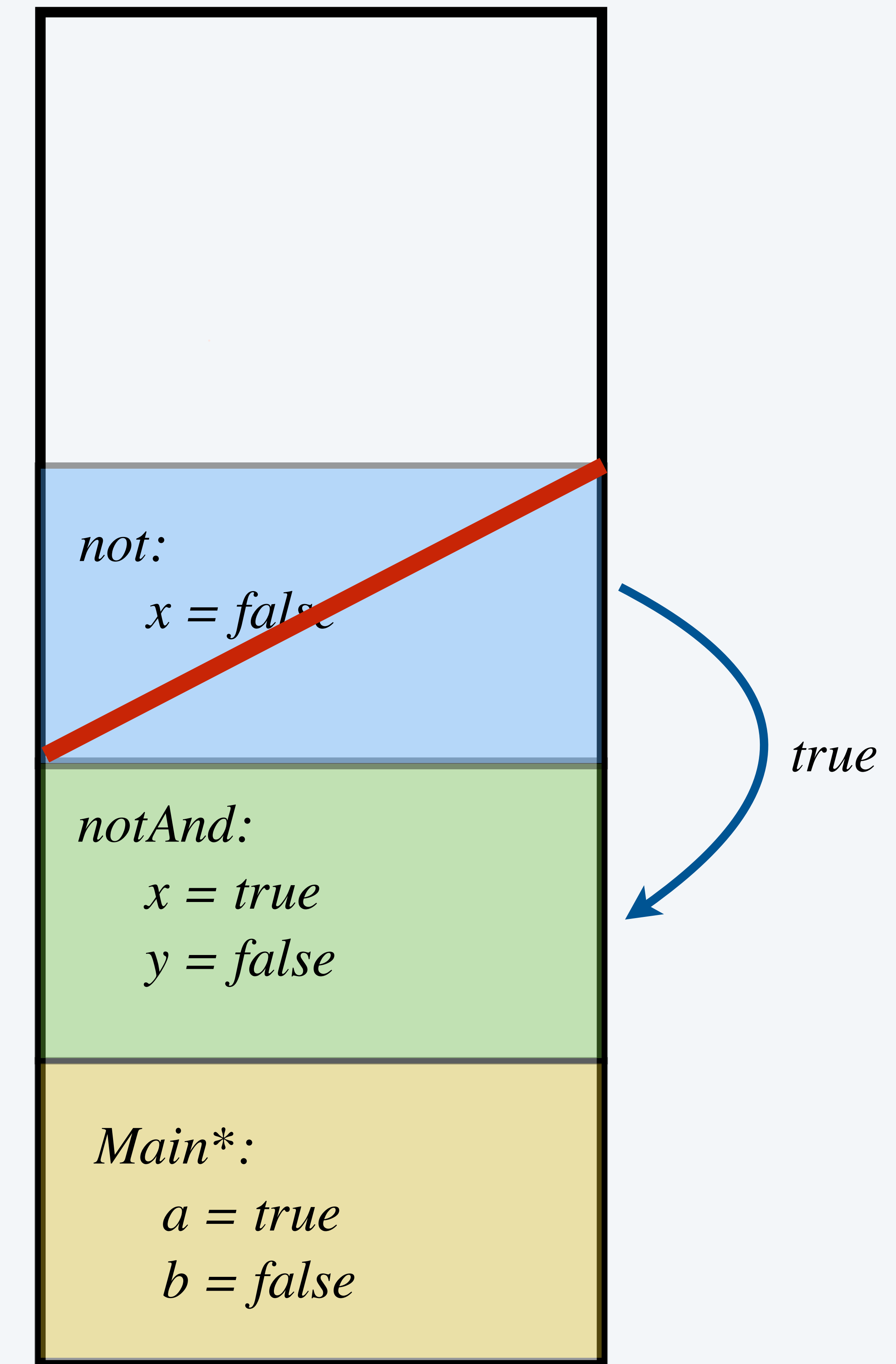
```
public static boolean not(boolean x){
   if (x == false)
        return true;
   return false;
}
public static boolean notAnd(boolean x, boolean y) {
   if (x == y)
        return not(true);
   return not(false);
}
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = notAnd(a, b);
}
```



Visualizing call stack

- Recreate `!(a && b)` without && and !

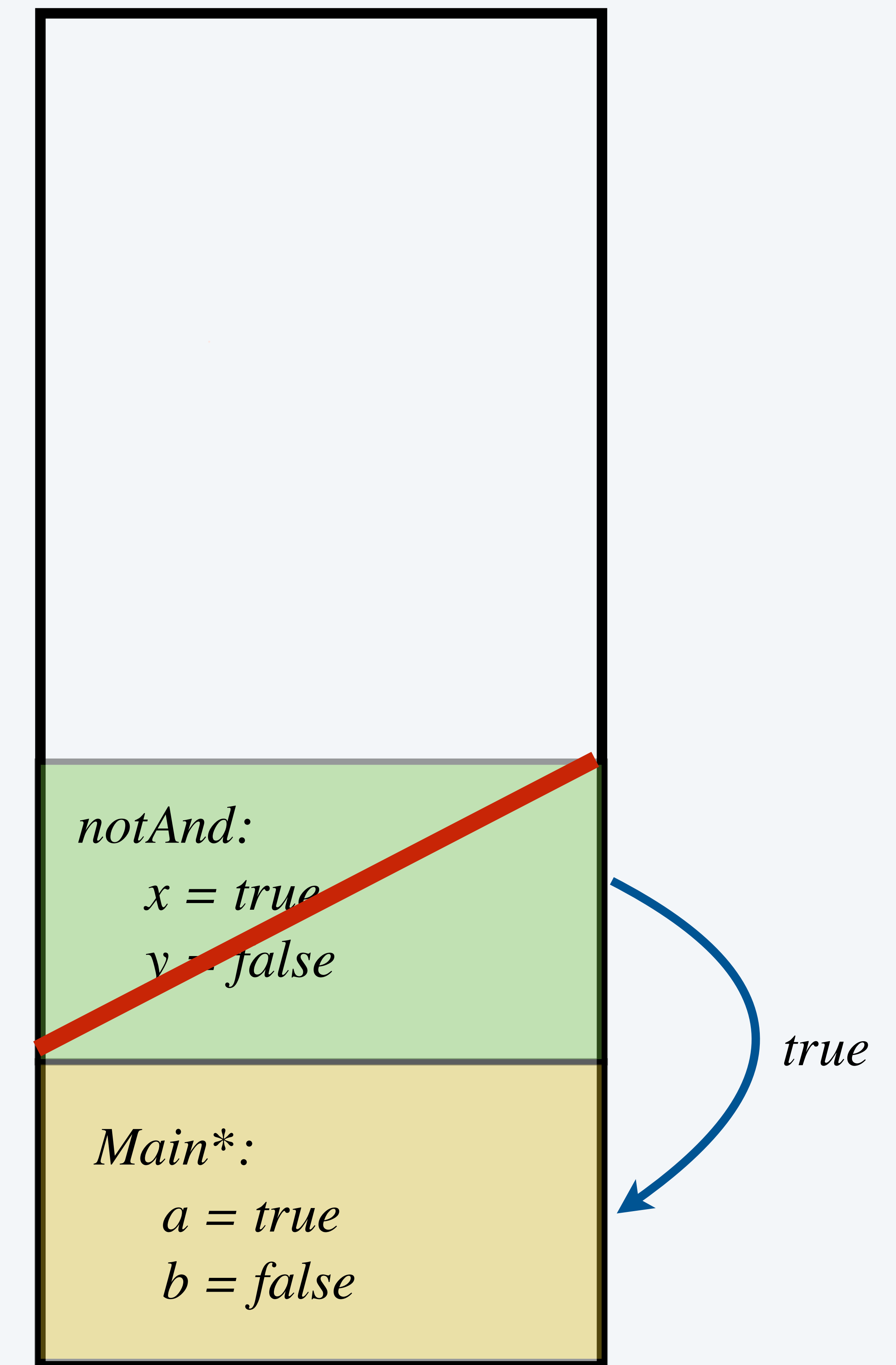
```
public static boolean not(boolean x){
   if (x == false)
        return true;
   return false;
}
public static boolean notAnd(boolean x, boolean y) {
   if (x == y)
        return not(true);
   return not(false);
}
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = notAnd(a, b);
}
```



Visualizing call stack

- Recreate `!(a && b)` without && and !

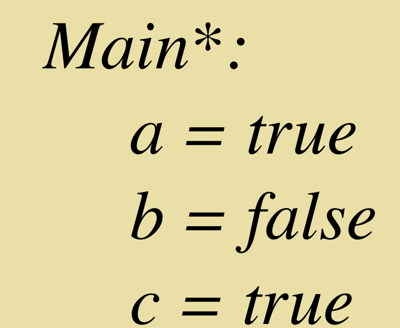
```
public static boolean not(boolean x){
   if (x == false)
         return true;
   return false;
}
public static boolean notAnd(boolean x, boolean y) {
   if (x == y)
         return not(true);
   return not(false);
}
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = notAnd(a, b);
}
```



Visualizing call stack

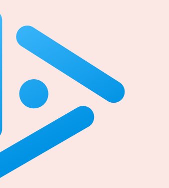
- Recreate `!(a && b)` without && and !

```
public static boolean not(boolean x){
   if (x == false)
        return true;
   return false;
}
public static boolean notAnd(boolean x, boolean y) {
   if (x == y)
        return not(true);
   return not(false);
}
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = notAnd(a, b);
}
```



* ignore args[] for now

Arrays 2: quiz 1 revisited



What does the following code fragment print?

A. 0 1 2 0 1 2

B. 0 1 2 1 2 6

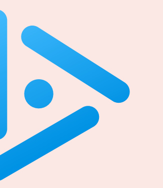
C. 1 2 6 0 1 2

D. 1 2 6 0 1 2

E. 1 2 6 1 2 6

```
int[] a = { 1, 2, 6 };
int[] b = new int[a.length];

b = a;
for (int i = 0; i < b.length; i++) {
    b[i] = i;
}

for (int i = 0; i < a.length; i++) {
    System.out.print(a[i] + " ");
}

for (int i = 0; i < b.length; i++) {
    System.out.print(b[i] + " ");
}
```

Arrays 2: quiz 1 revisited



What does the following code fragment print?

- **A.** 0 1 2 0 1 2
- **B.** 0 1 2 1 2 6
- **C.** 1 2 6 0 1 2
- **D.** 1 2 6 0 1 2
- **E.** 1 2 6 1 2 6

"b" now references same array "a" references

```
int[] a = { 1, 2, 6 };
int[] b = new int[a.length];

b = a;
for (int i = 0; i < b.length; i++) {
    b[i] = i;
}

for (int i = 0; i < a.length; i++) {
    System.out.print(a[i] + " ");
}

for (int i = 0; i < b.length; i++) {
    System.out.print(b[i] + " ");
}
```

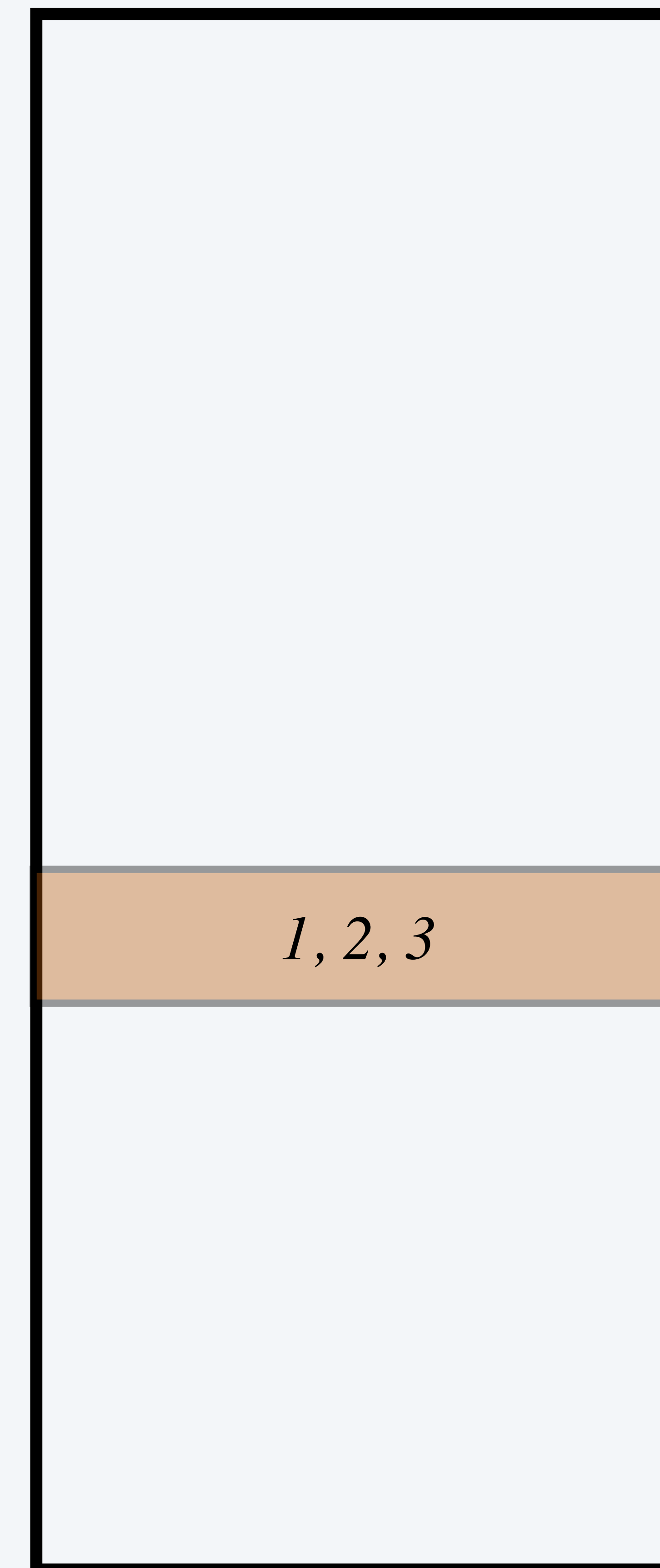
Array references and pass by value

- Array variables are references to memory that is actual array
- Java is pass by value language
- What happens if you pass array variable to another function?
  - The reference *is* the value

Visualizing call stack

- Heap – memory where “new” memory lives

```
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
      doubleNums[i] = x[i] * 2;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   double(numbers);
}
```



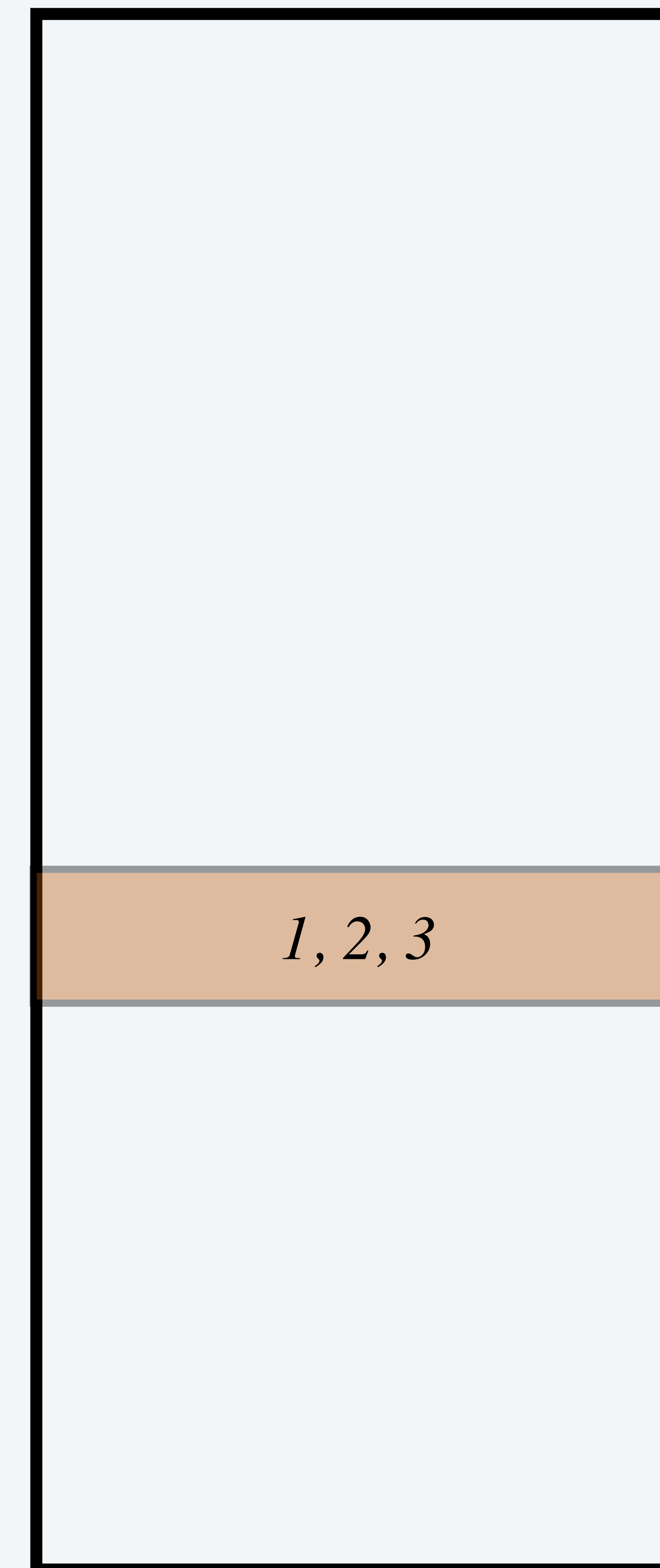
* ignore args[]

Visualizing call stack

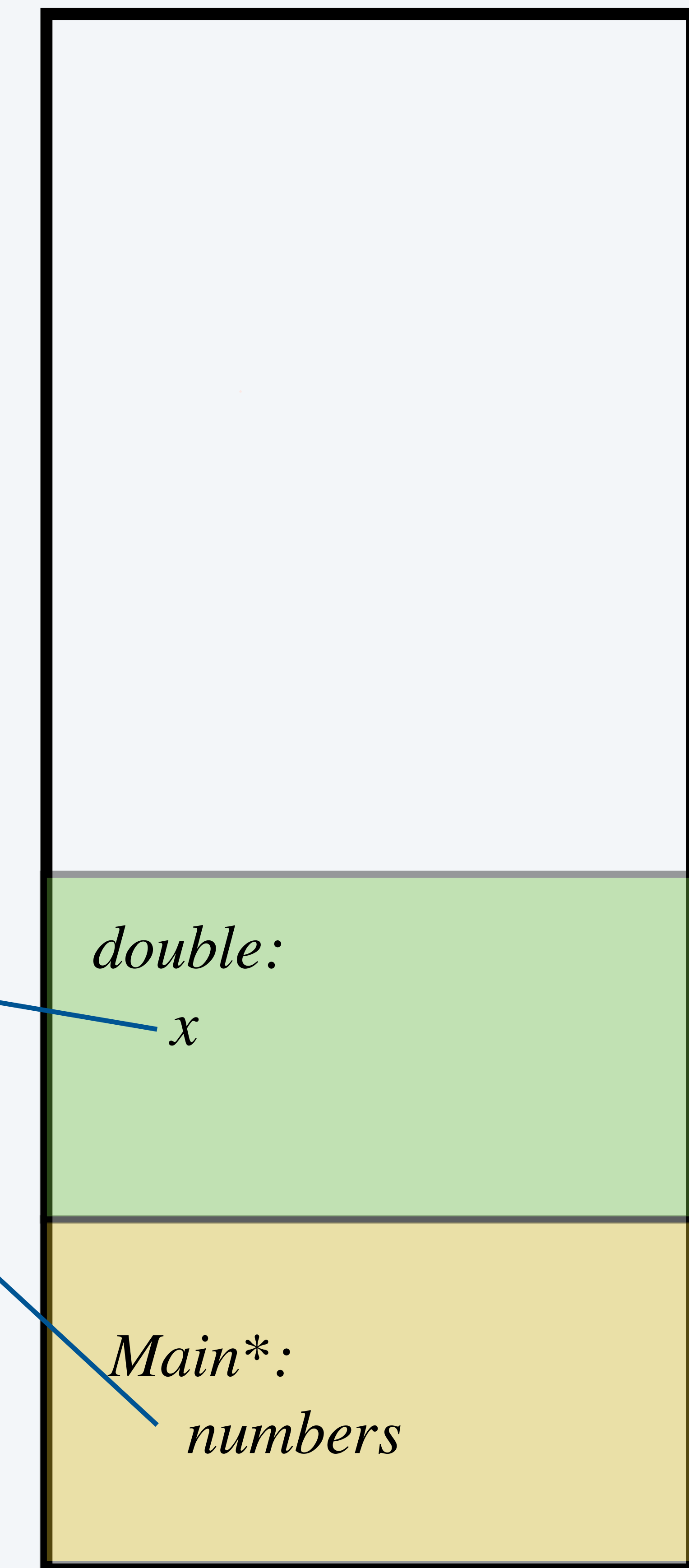
- Heap – memory where “new” memory lives
- Pass by value means x is a copy of reference

```java
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
       doubleNums[i] = x[i] * 2;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   double(numbers);
}
```

Heap



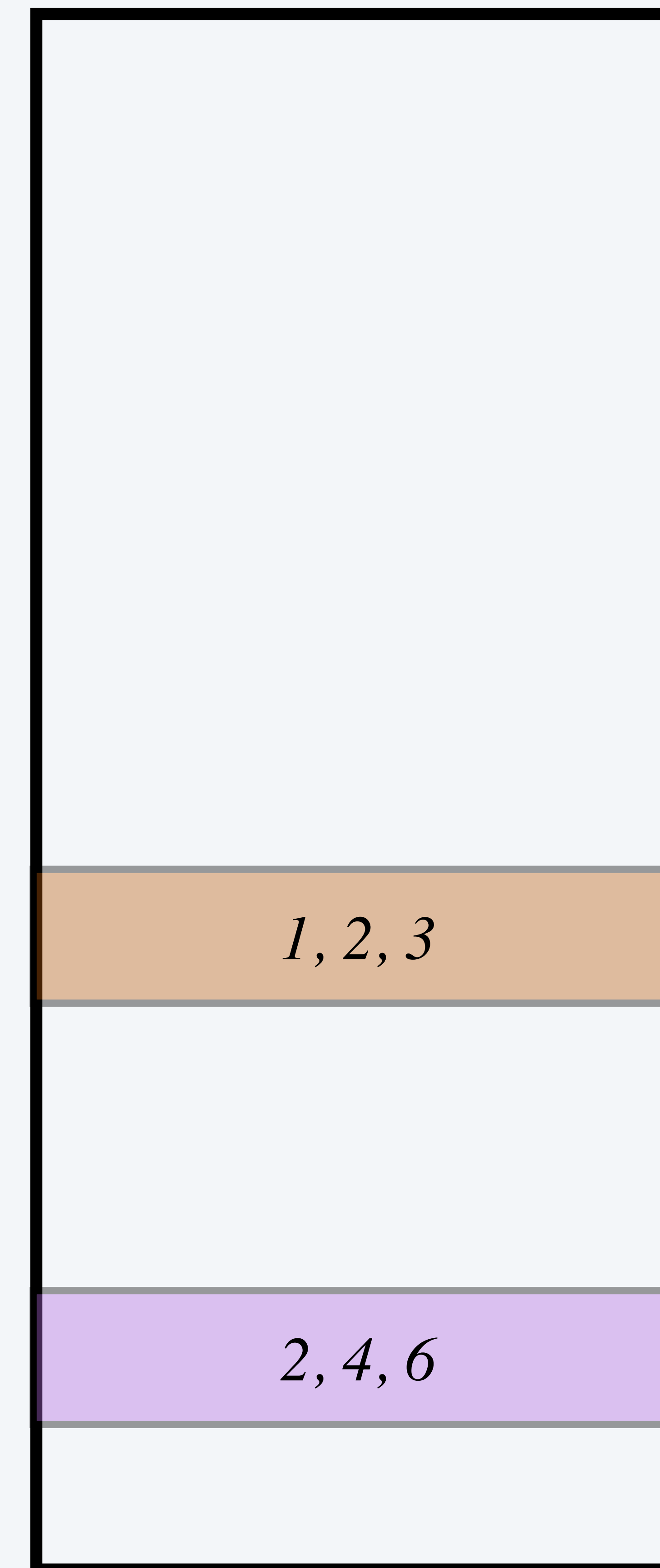
Stack

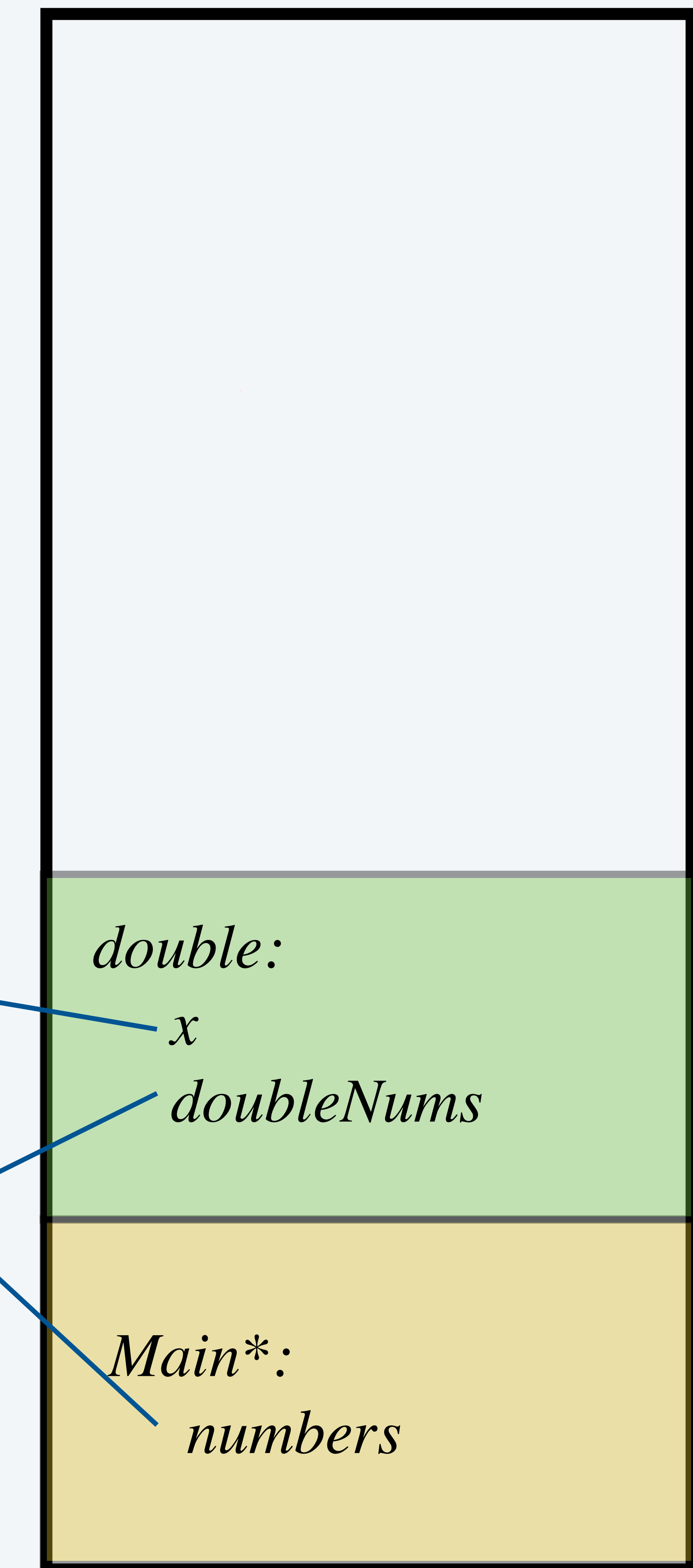


* ignore args[]

Visualizing call stack

```
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
      doubleNums[i] = x[i] * 2;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   double(numbers);
}
```

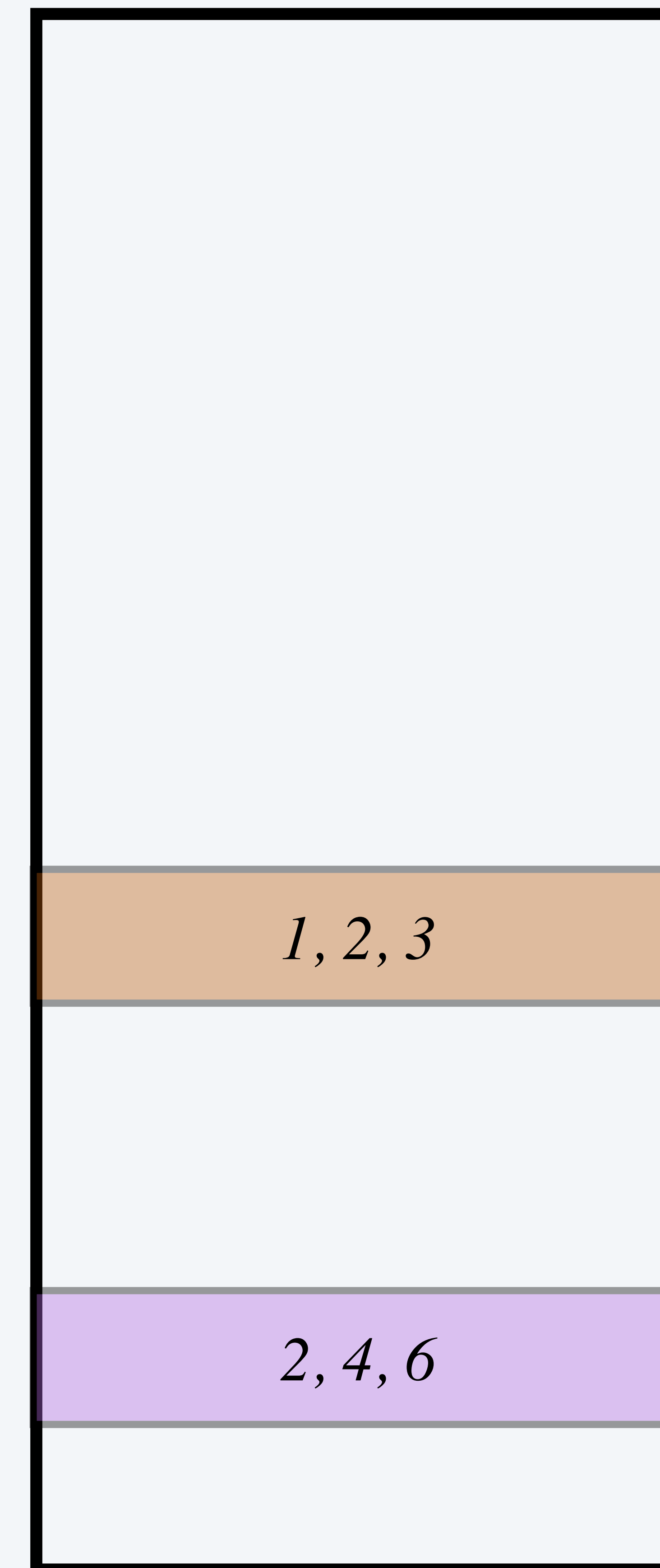


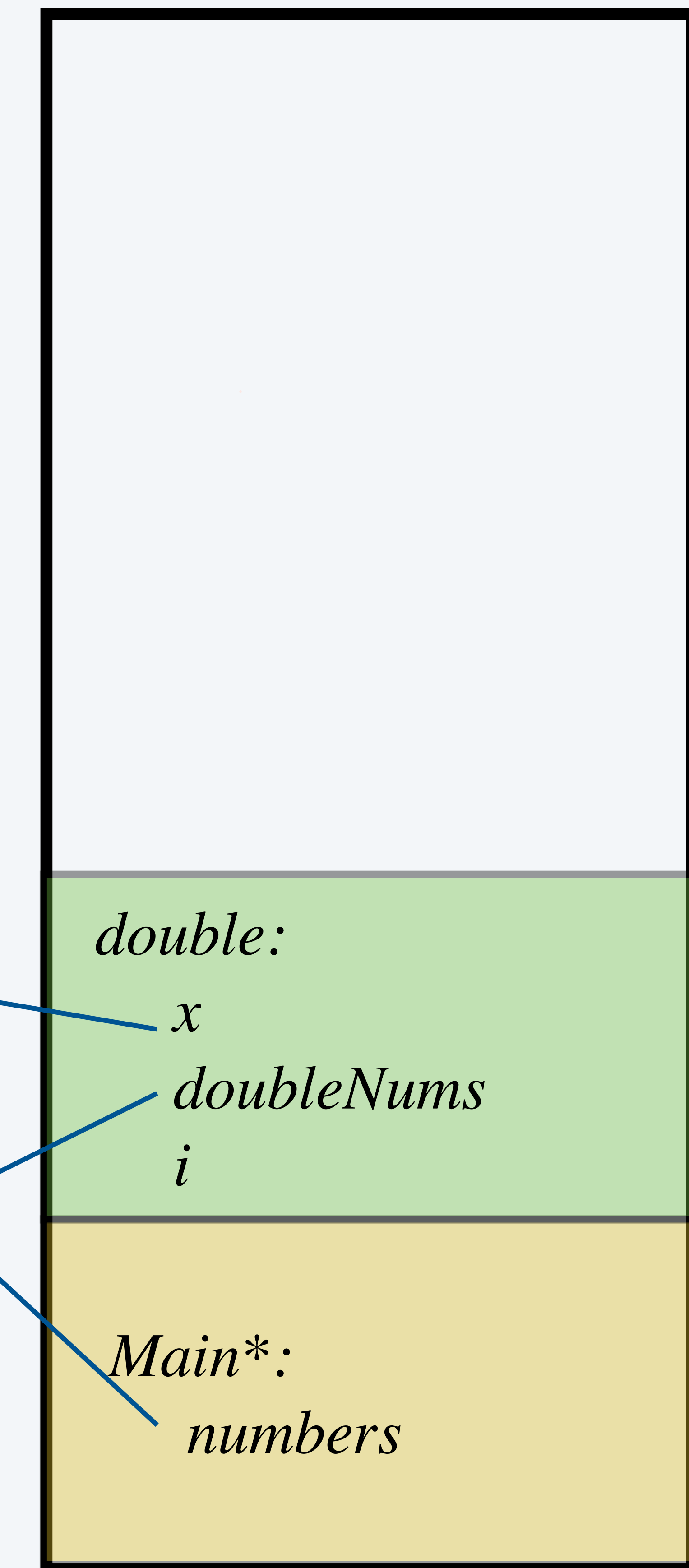


* ignore args[]

Visualizing call stack

```
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
      doubleNums[i] = x[i] * 2;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   double(numbers);
}
```



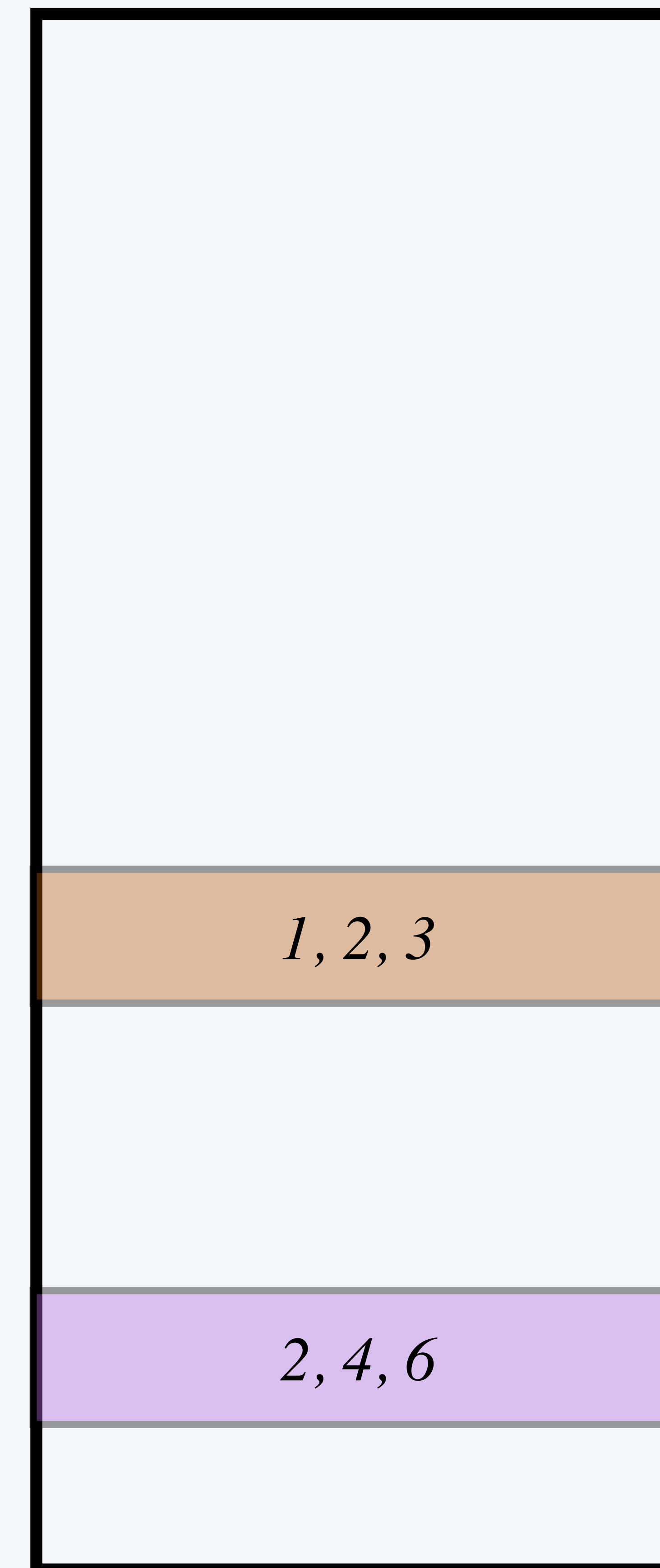


* ignore args[]

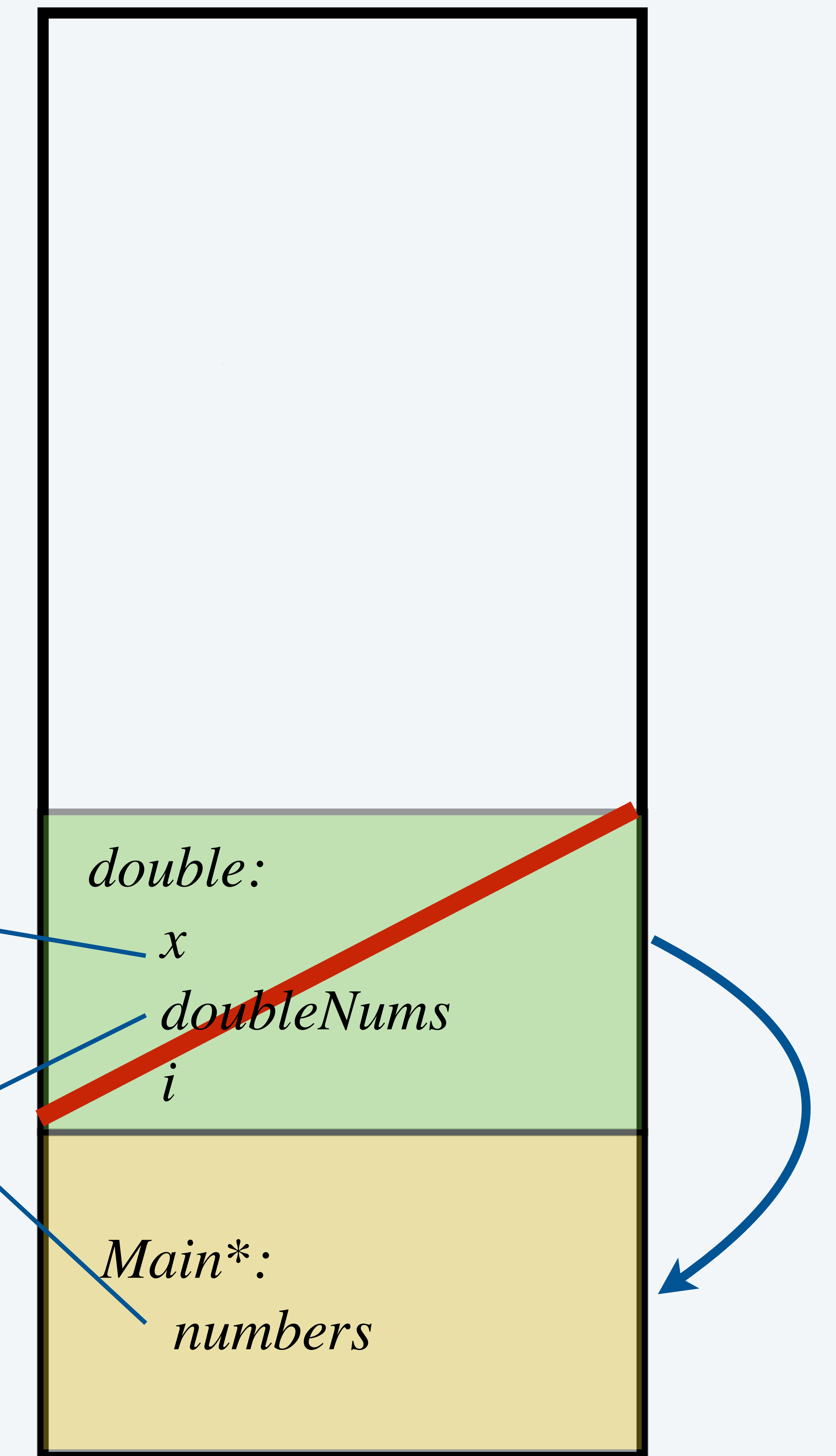
Visualizing call stack

```
public static void double(int[] x) {
    int[] doubleNums = new int[x.length];
    for (int i = 0; i < x.length; i++) {
        doubleNums[i] = x[i] * 2;
}
public static void main(String[] args) {
    int[] numbers = { 1, 2, 3 };
    double(numbers);
}
```

Heap



Stack

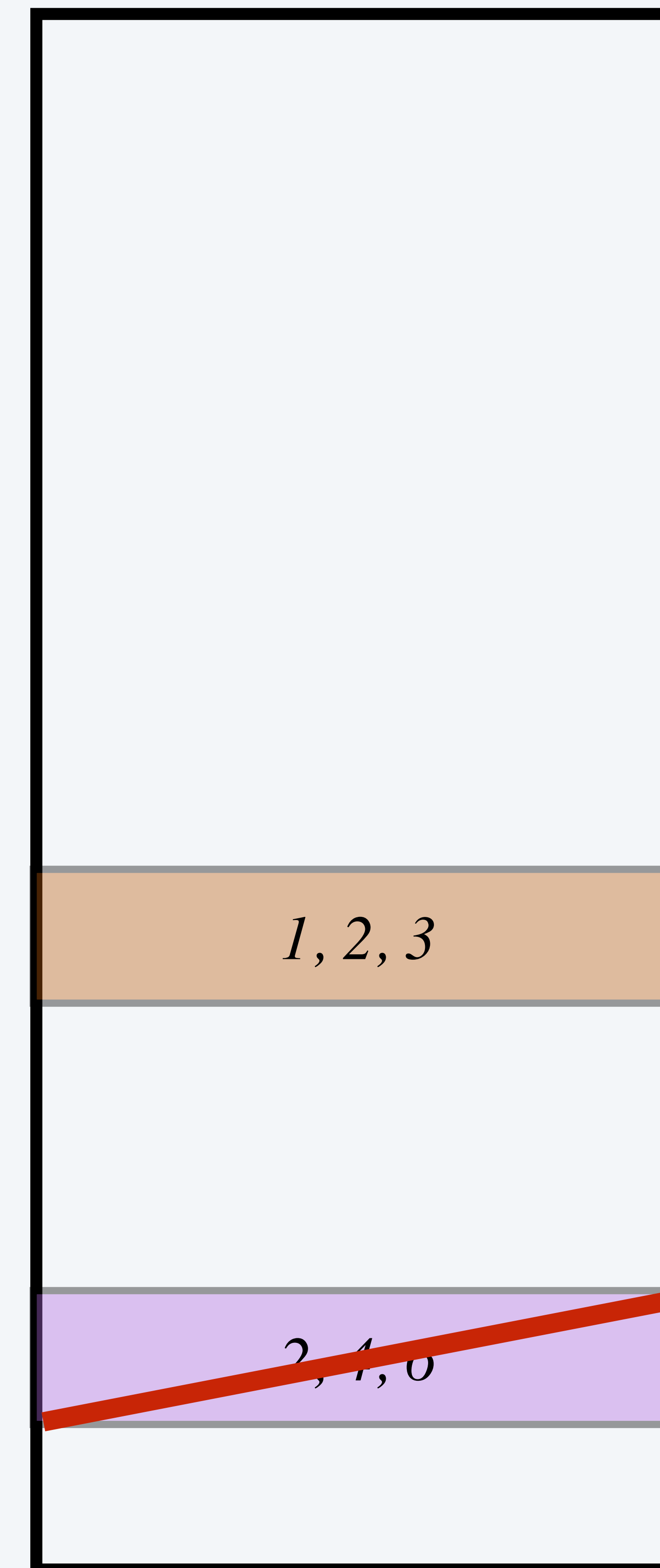


* ignore args[]

Visualizing call stack

```
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
      doubleNums[i] = x[i] * 2;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   double(numbers);
}
```

Heap



Stack

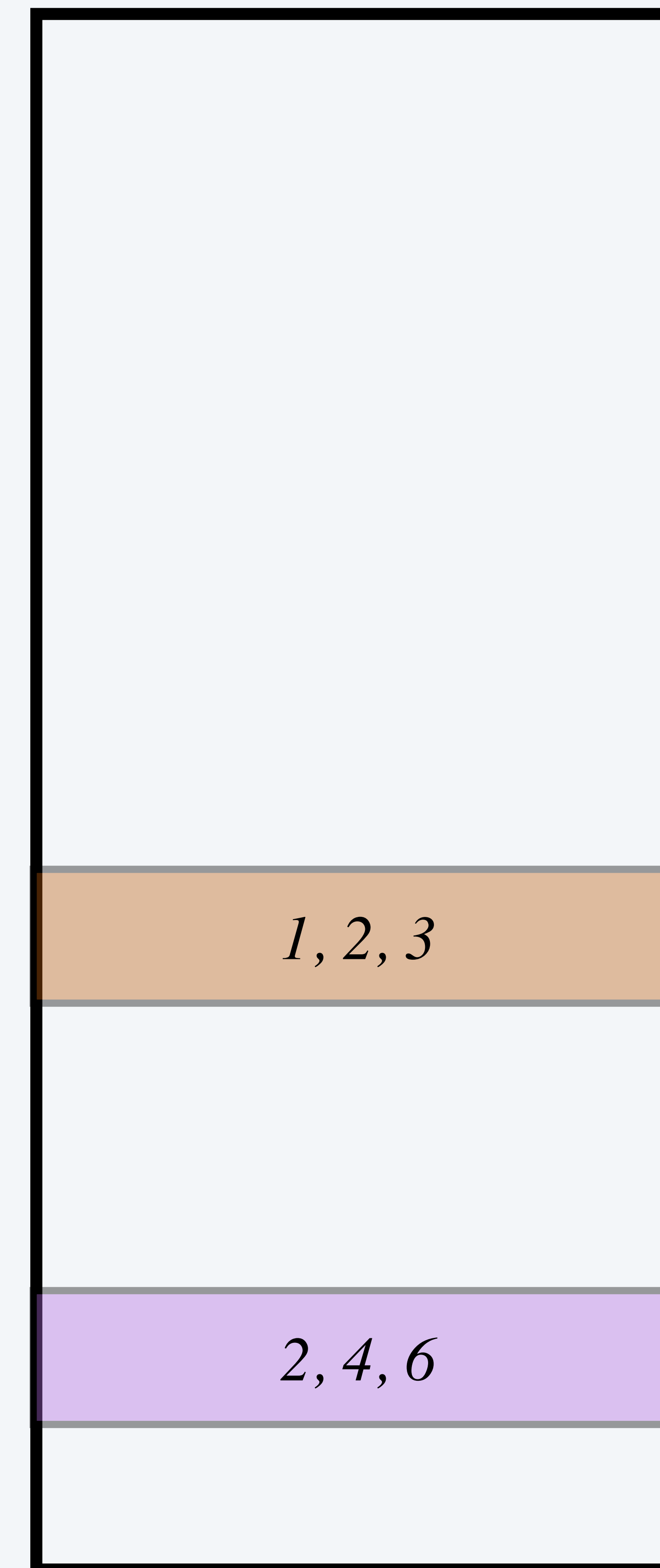
* ignore args[]

Visualizing call stack

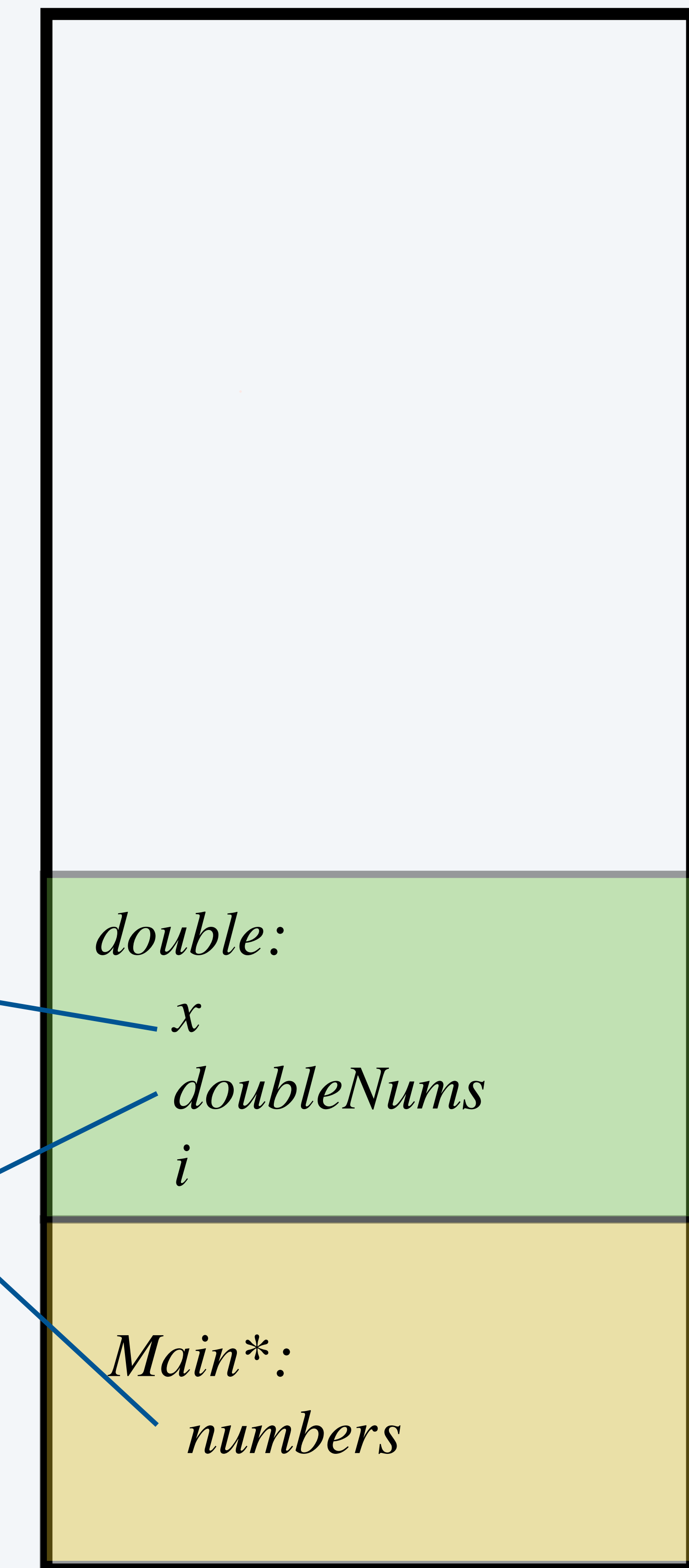
- To fix: either return new array

```
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
       doubleNums[i] = x[i] * 2;
   return doubleNums;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   numbers = double(numbers);
}
```

Heap



Stack



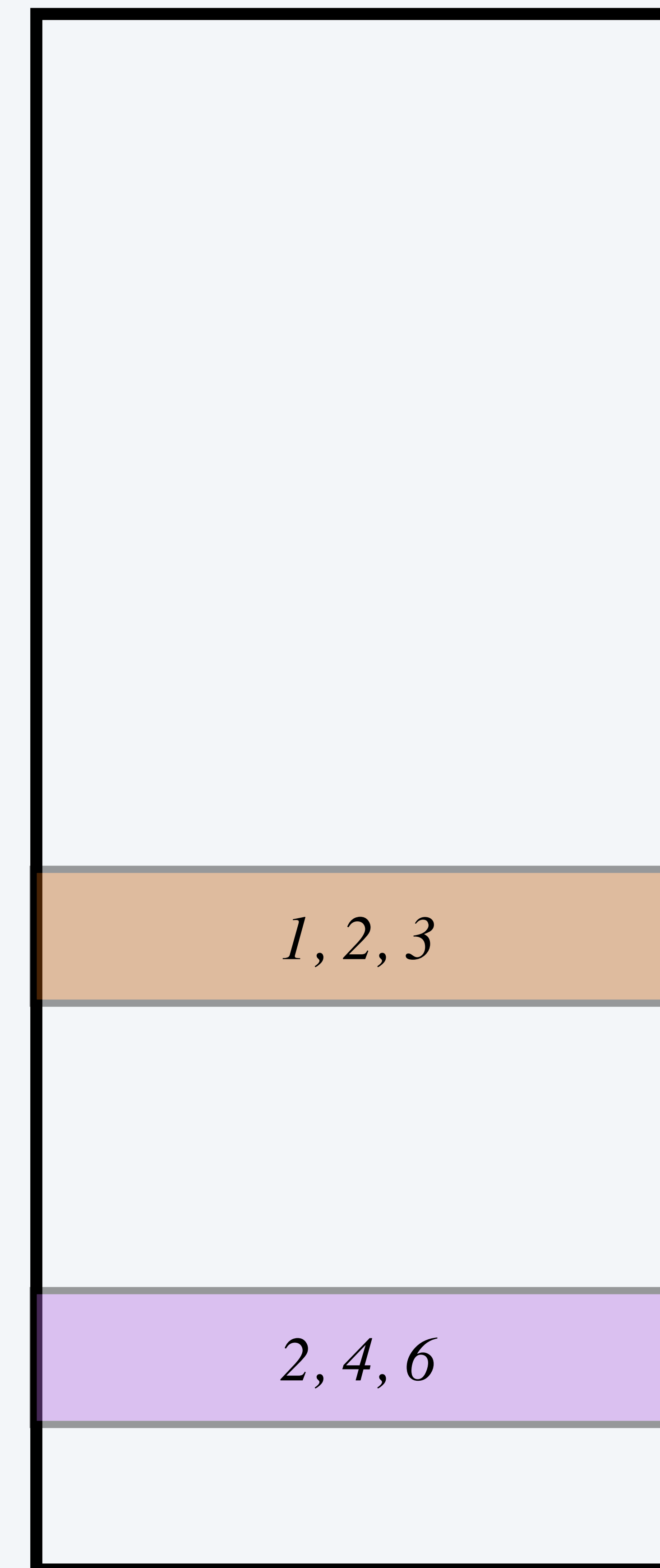
* ignore args[]

Visualizing call stack

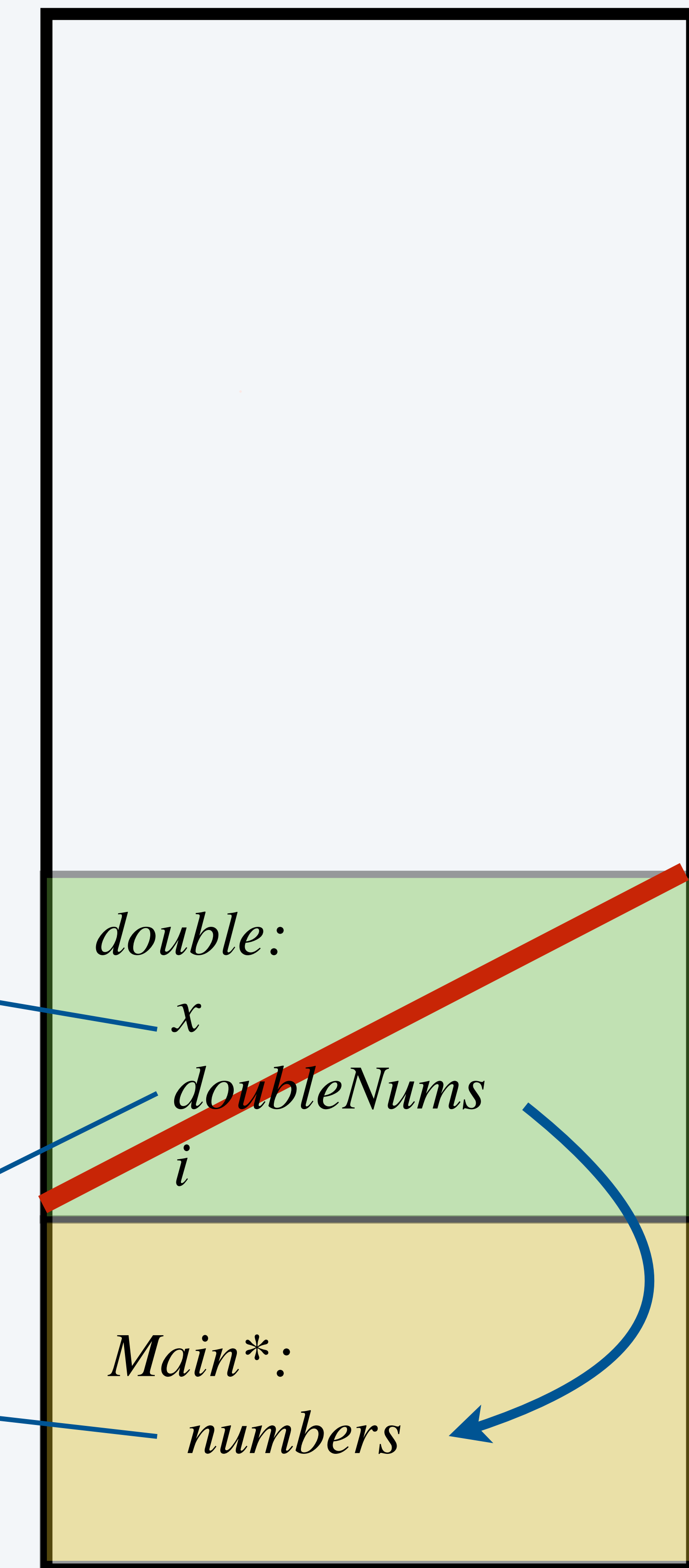
- To fix: either return new array

```
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
      doubleNums[i] = x[i] * 2;
   return doubleNums;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   numbers = double(numbers);
}
```

Heap



Stack



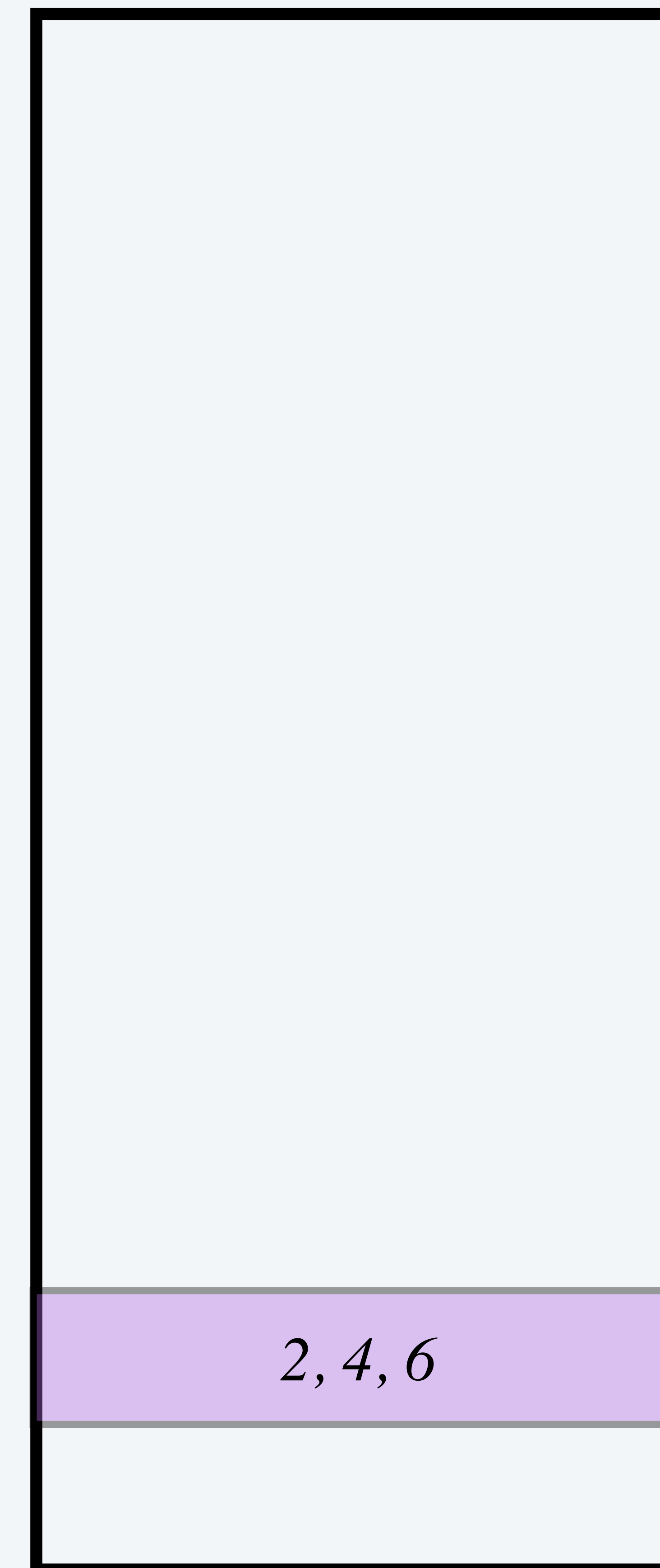
* ignore args[]

Visualizing call stack

- To fix: either return new array

```
public static void double(int[] x) {
   int[] doubleNums = new int[x.length];
   for (int i = 0; i < x.length; i++) {
      doubleNums[i] = x[i] * 2;
   return doubleNums;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   numbers = double(numbers);
}
```

Heap



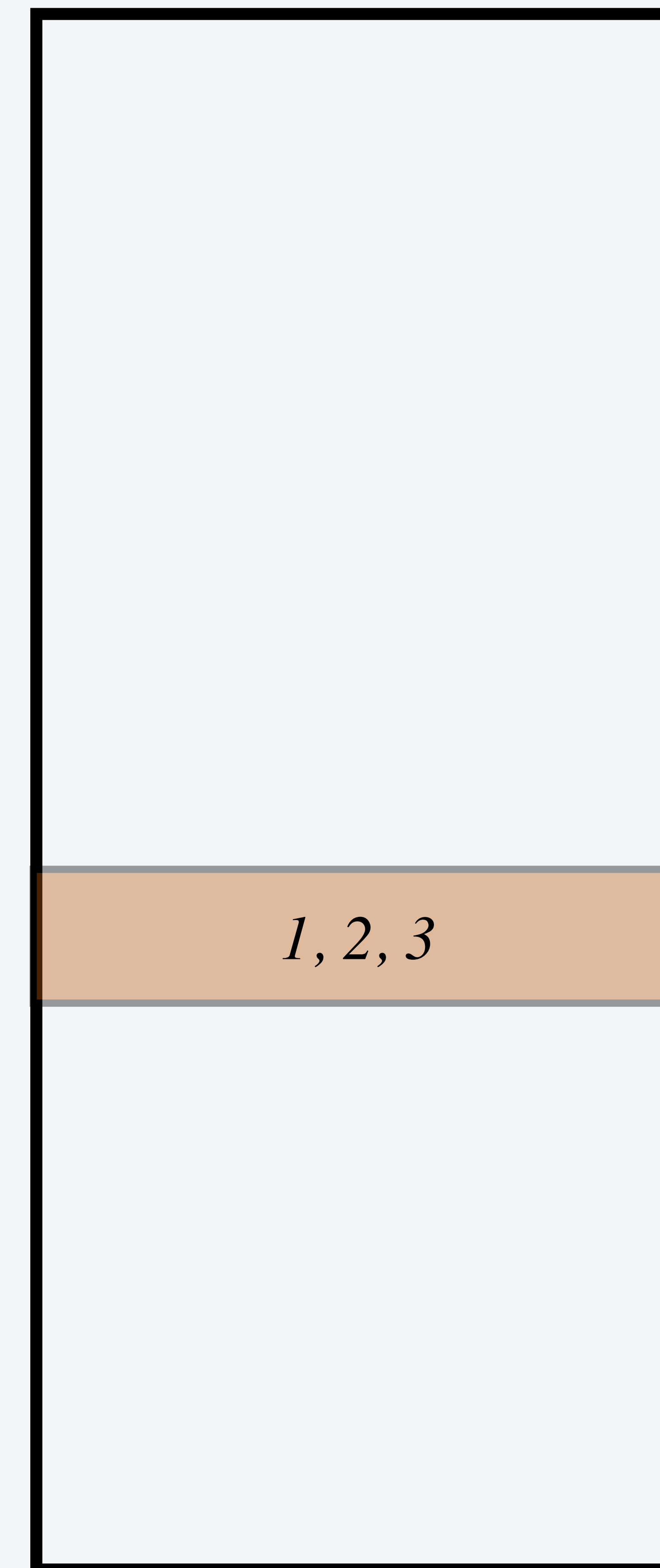
Stack

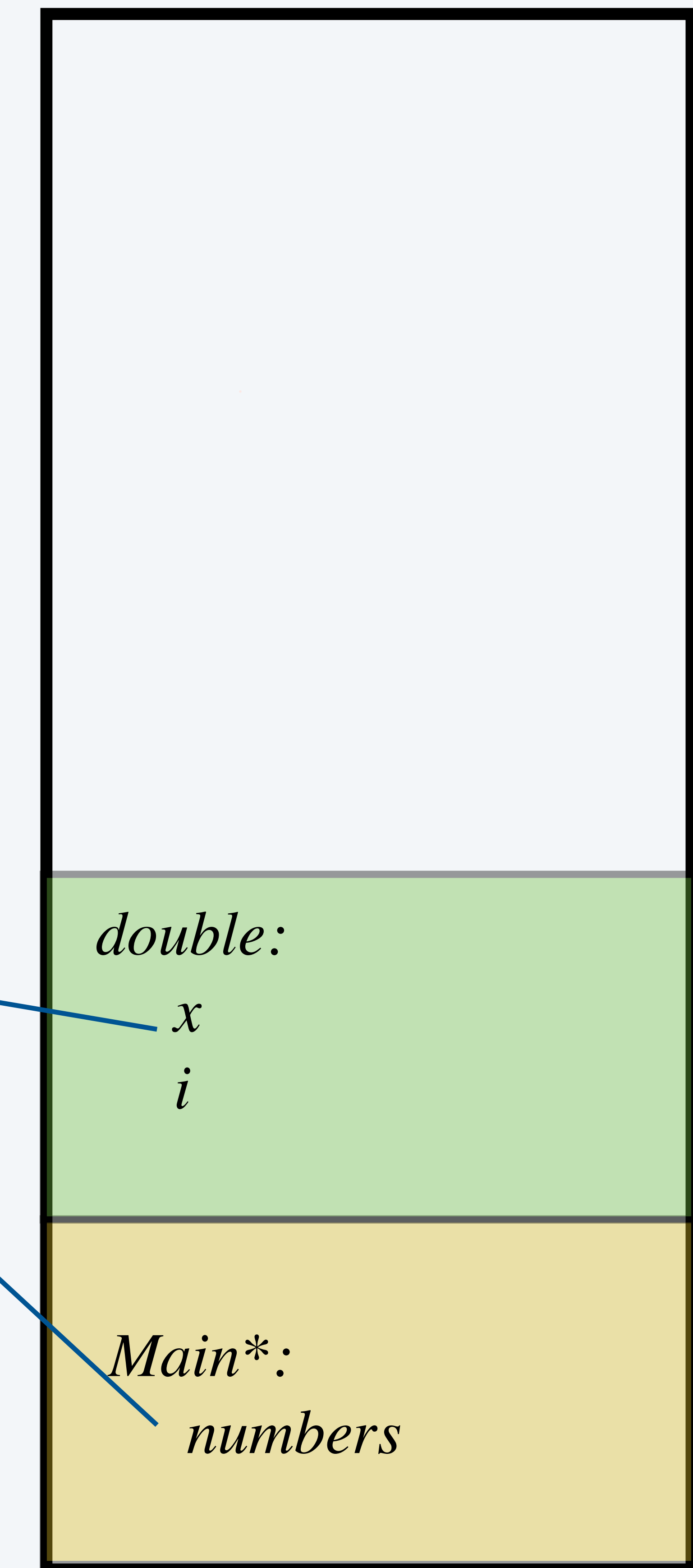
* ignore args[]

Visualizing call stack

- To fix: either return new array or modify in place

```
public static void double(int[] x) {
    for (int i = 0; i < x.length; i++) {
        x[i] = x[i] * 2;
}
public static void main(String[] args) {
    int[] numbers = { 1, 2, 3 };
    double(numbers);
}
```





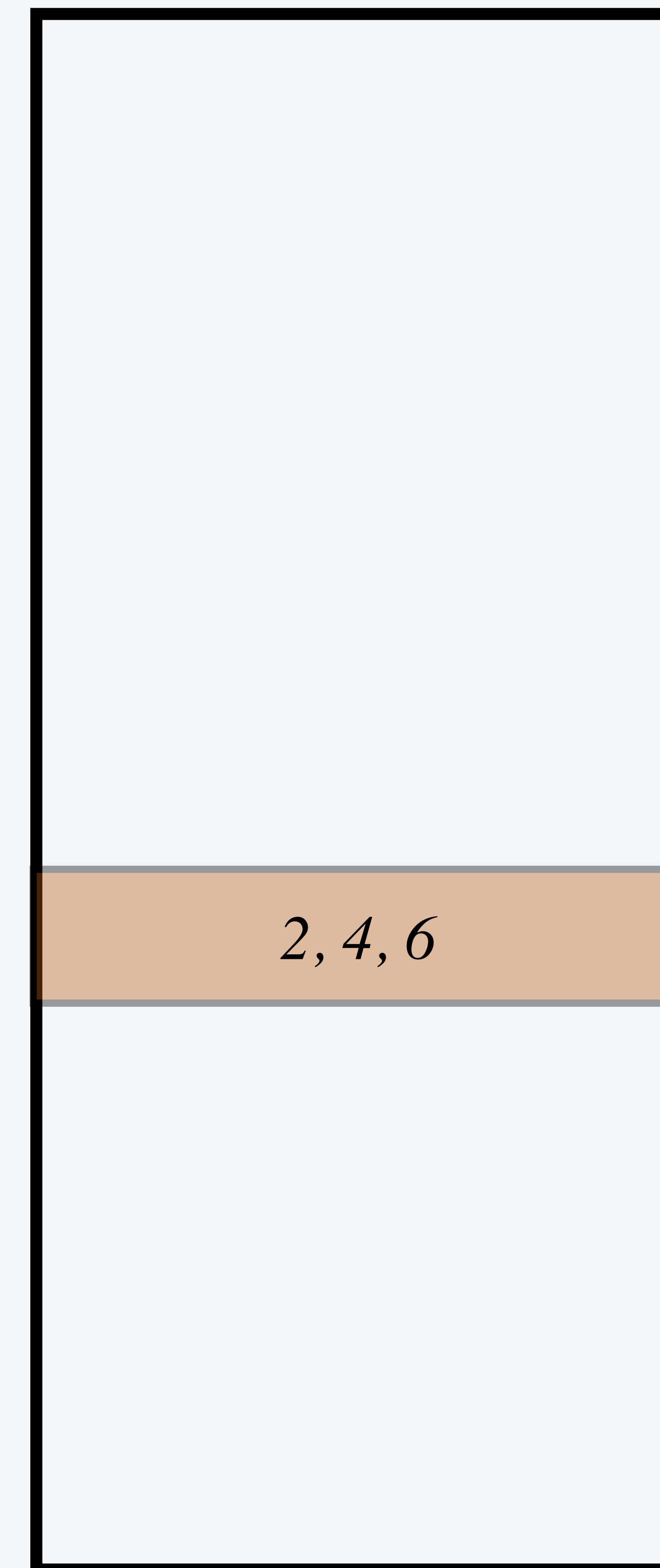
* ignore args[]

Visualizing call stack

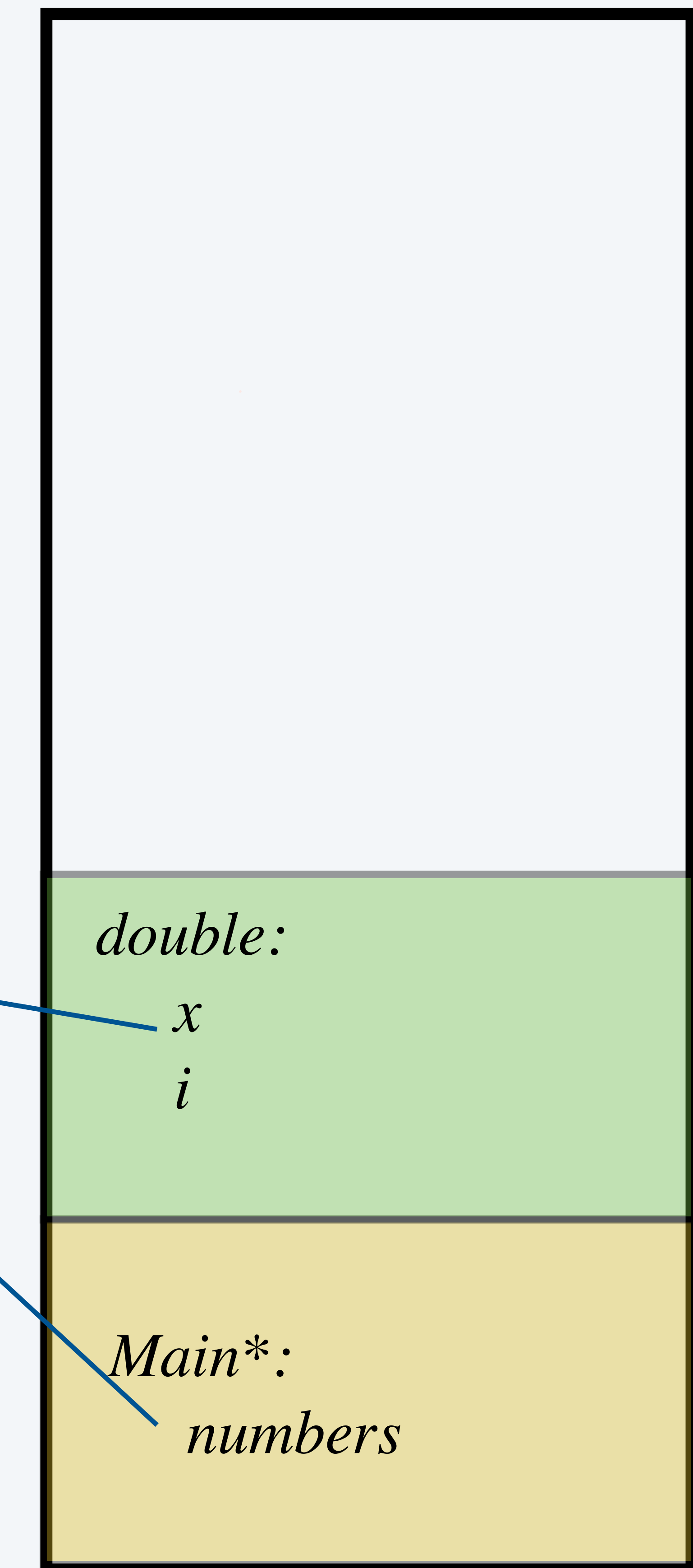
- To fix: either return new array or modify in place

```
public static void double(int[] x) {
    for (int i = 0; i < x.length; i++) {
        x[i] = x[i] * 2;
}
public static void main(String[] args) {
    int[] numbers = { 1, 2, 3 };
    double(numbers);
}
```

Heap



Stack

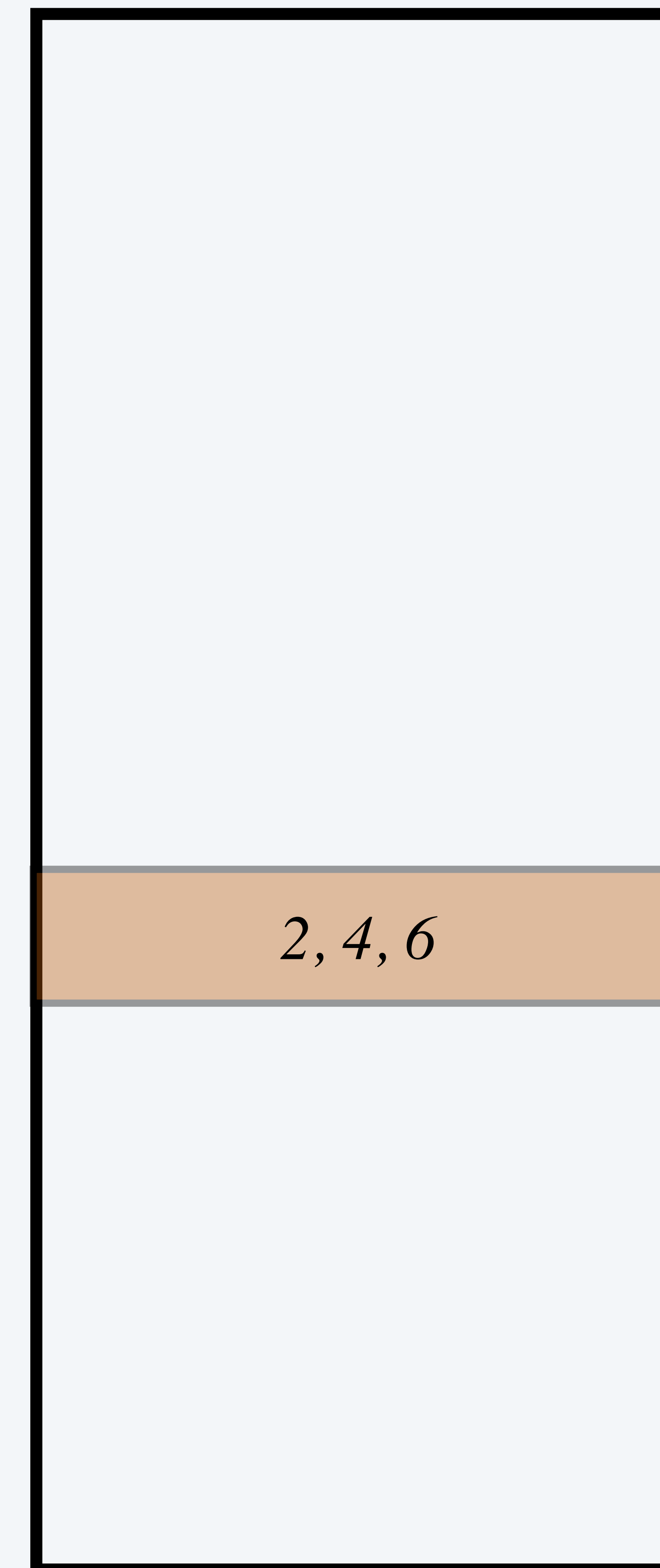


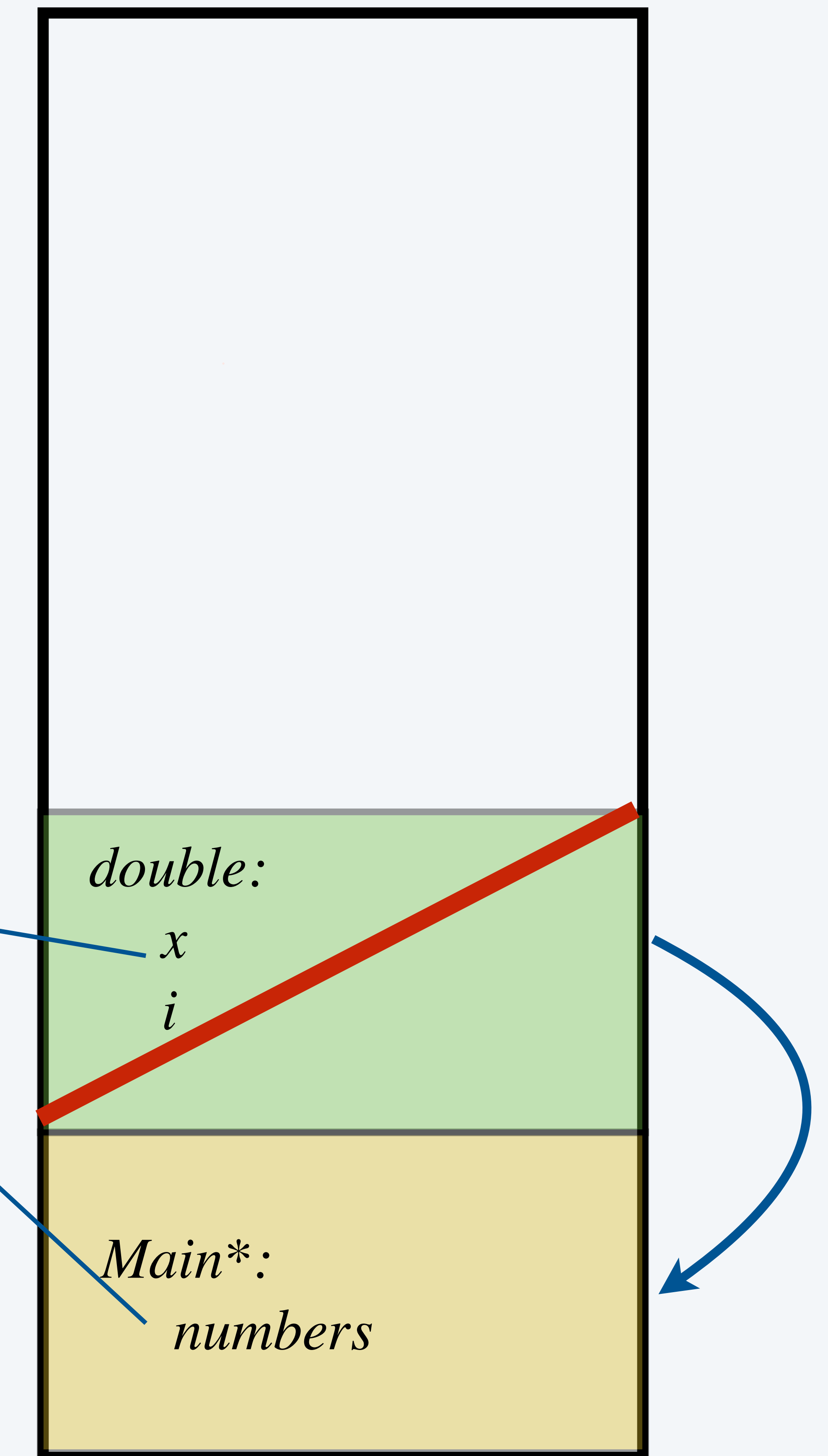
* ignore args[]

Visualizing call stack

- To fix: either return new array or modify in place

```
public static void double(int[] x) {
   for (int i = 0; i < x.length; i++) {
       x[i] = x[i] * 2;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   double(numbers);
}
```





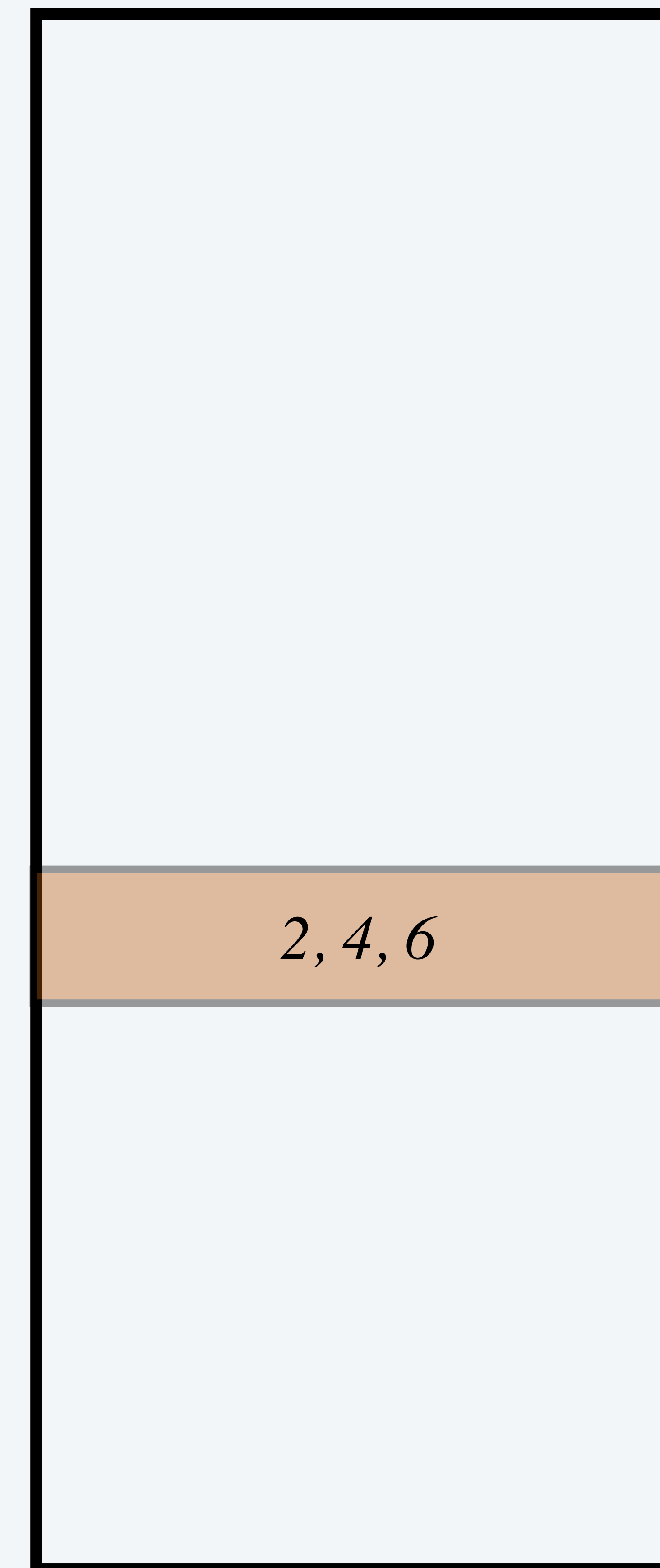
* ignore args[]

Visualizing call stack

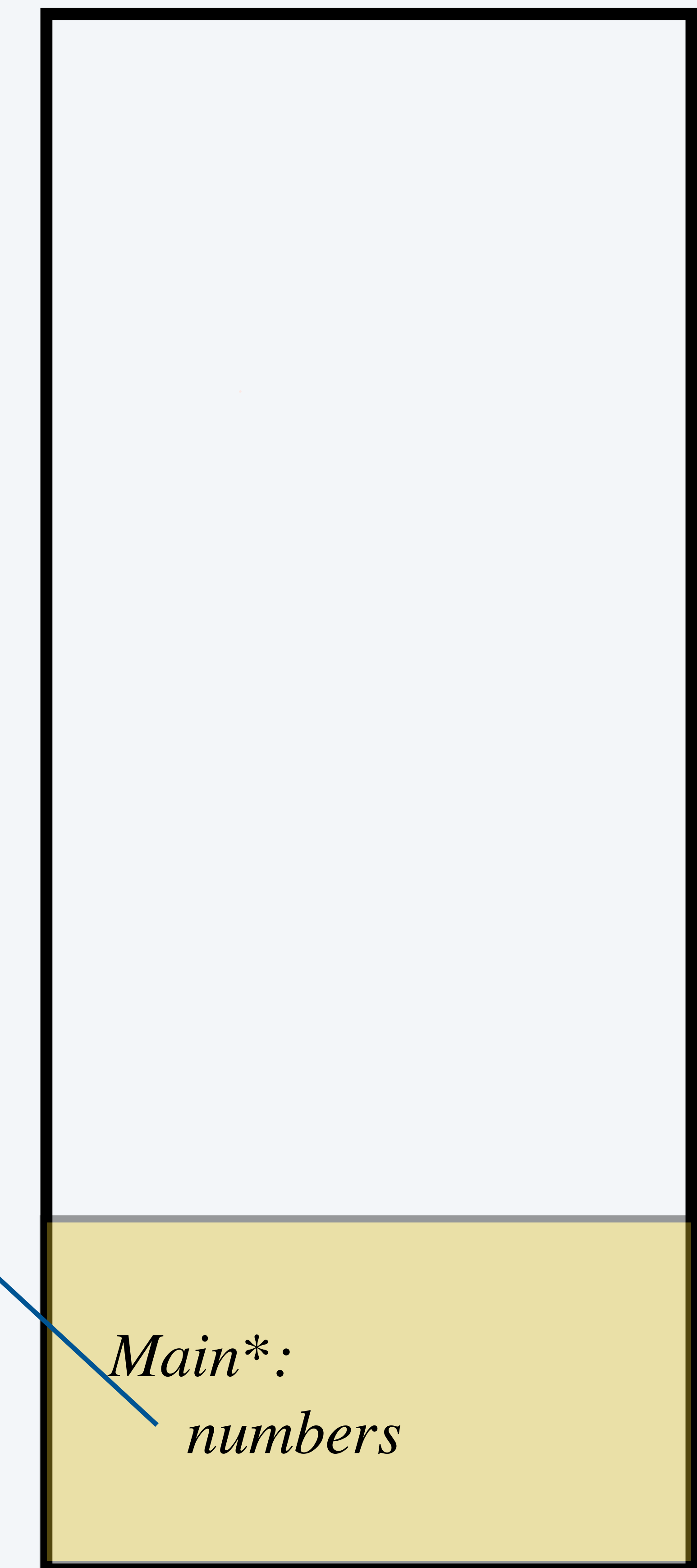
- To fix: either return new array or modify in place

```
public static void double(int[] x) {
   for (int i = 0; i < x.length; i++) {
       x[i] = x[i] * 2;
}
public static void main(String[] args) {
   int[] numbers = { 1, 2, 3 };
   double(numbers);
}
```

Heap



Stack



* ignore args[]

Visualizing call stack

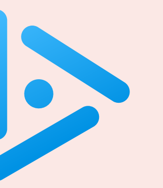
- To fix: either return new array or modify in place

Return new array

```
public static double[] double(int[] x) {
    int[] doubleNums = new int[x.length];
    for (int i = 0; i < x.length; i++) {
        doubleNums[i] = x[i] * 2;
    return doubleNums;
}
public static void main(String[] args) {
    int[] numbers = { 1, 2, 3 };
    numbers = double(numbers);
}
```

Modify in place

```
public static void double(int[] x) {
    for (int i = 0; i < x.length; i++) {
        x[i] = x[i] * 2;
}
public static void main(String[] args) {
    int[] numbers = { 1, 2, 3 };
    double(numbers);
}
```



What does the following program print?

A. 12

B. -12

C. Compile-time error.

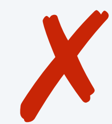
D. Run-time error.

```
public class AnotherMystery {

   public static void negate(int[] b) {
      for (int i = 0; i < b.length; i++)
         b[i] = -b[i];
   }

   public static void main(String[] args) {
      int[] a = { 12, 6 };
      negate(a);
      StdOut.println(a[0]);
   }
}
```

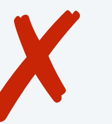
Copying an array

Beware of common bugs!

```
public static int[] copy(int[] a) {
   return a;
}
```



```
public static void copy(int[] a, int[] b) {
   b = a;
}
```



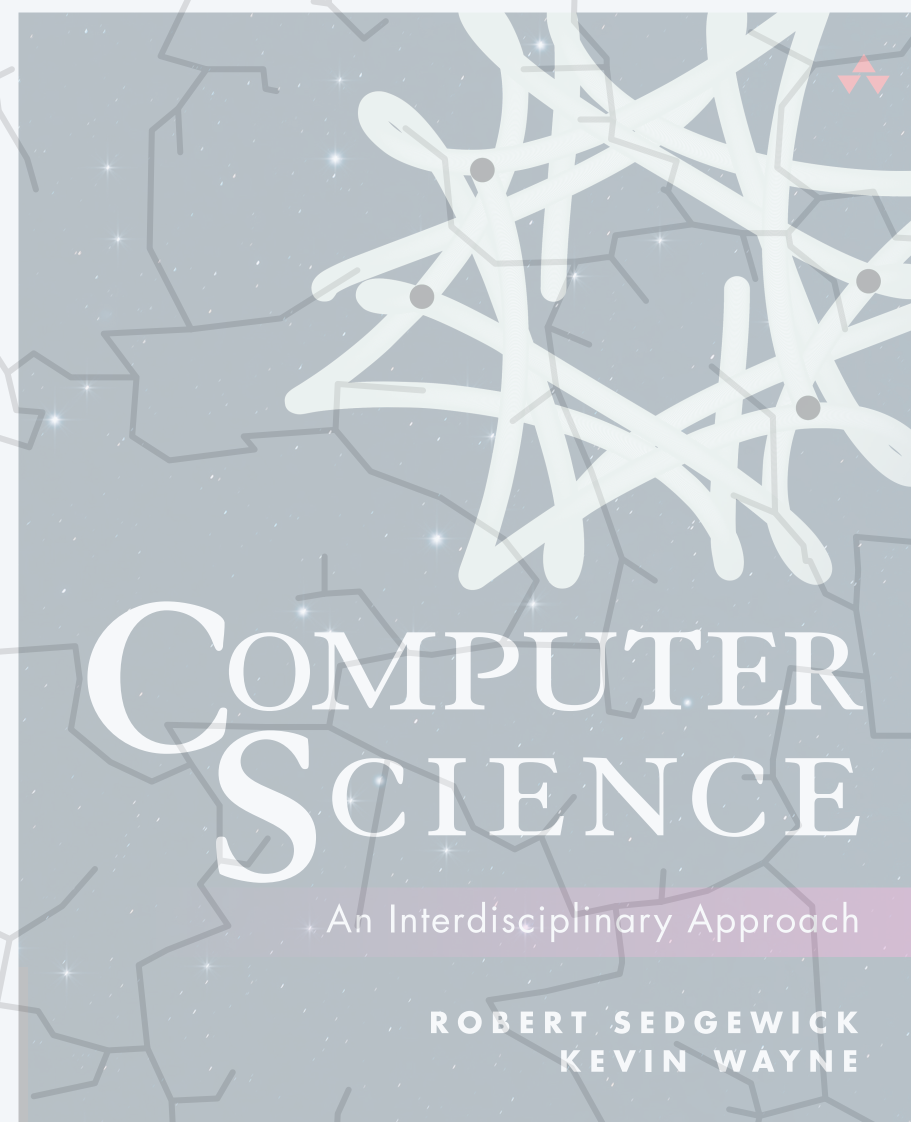
```
public static int[] copy(int[] a) {
   int[] b = new int[a.length];
   for (int i = 0; i < a.length; i++)
      b[i] = a[i];
   return a;
}
```

```
public static void copy(int[] a, int[] b) {
   for (int i = 0; i < a.length; i++)
      b[i] = a[i];
}
```

if calling code ran b = new int[a.length]

```
public static int[] copy(int[] a) {
   int[] b = new int[a.length];
   for (int i = 0; i < a.length; i++)
      b[i] = a[i];
   return b;
}
```

```
public static void copy(int[] a, int[] b) {
   b = new int[a.length];
   for (int i = 0; i < a.length; i++)
      b[i] = a[i];
}
```

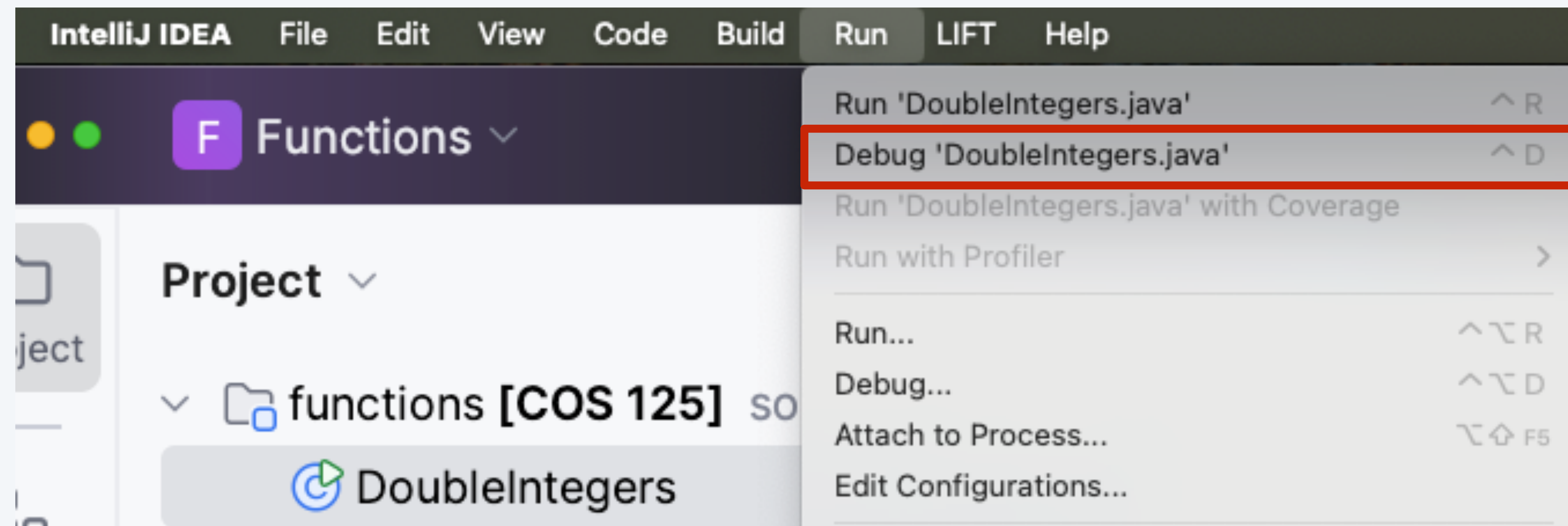


2.1 FUNCTIONS

- *under-the-hood*
- *debugger*
- *what next?*

Debugger

- Print statements are helpful for finding simple bugs
- Debugger can be used for more complex bugs
- Shows entire state of program including call stack and variable values



Debugger

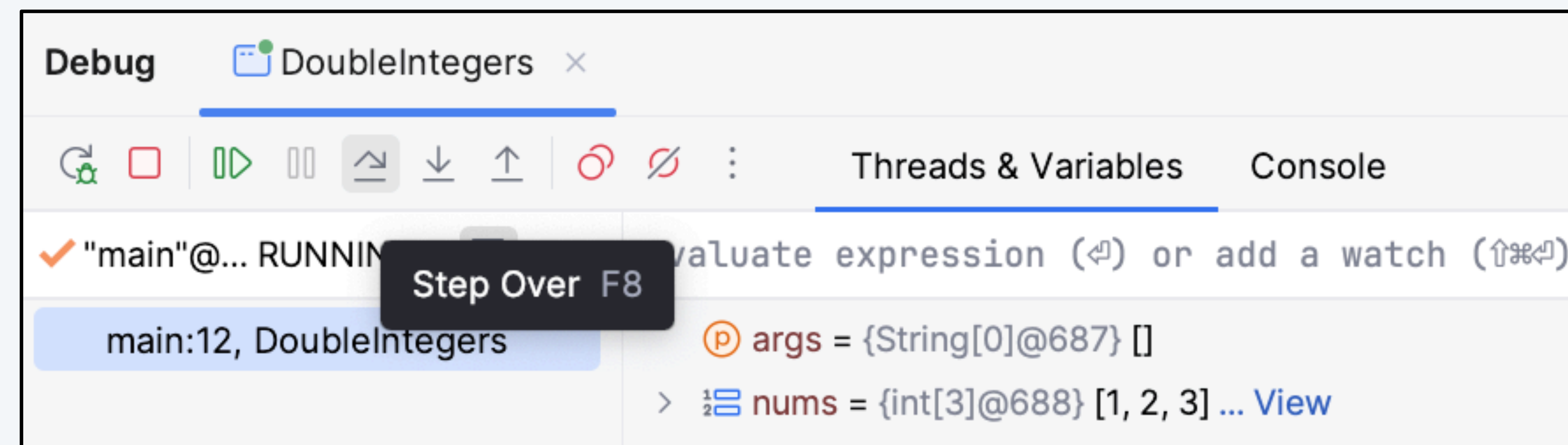
- Breakpoint – line of code where debugger stops and waits for command
  - Click on line number to set breakpoint
  - Then run debugger

```
10      public static void main(String[] args) {   args: []
            int[] nums = { 1, 2, 3 };
12          nums = doubleInts(nums);
13      }
14  }
```

Debugger

- Once at breakpoint, choose from different options
  - Step over – move to next line (skips function call)
  - Step into – enter function
  - Step out – finishes current function and returns to parent function

```
10      public static void main(String[] args) {   args: []
11          int[] nums = { 1, 2, 3 };   nums: [1, 2, 3]
12          nums = doubleInts(nums);   nums: [1, 2, 3]
13      }
14  }
```



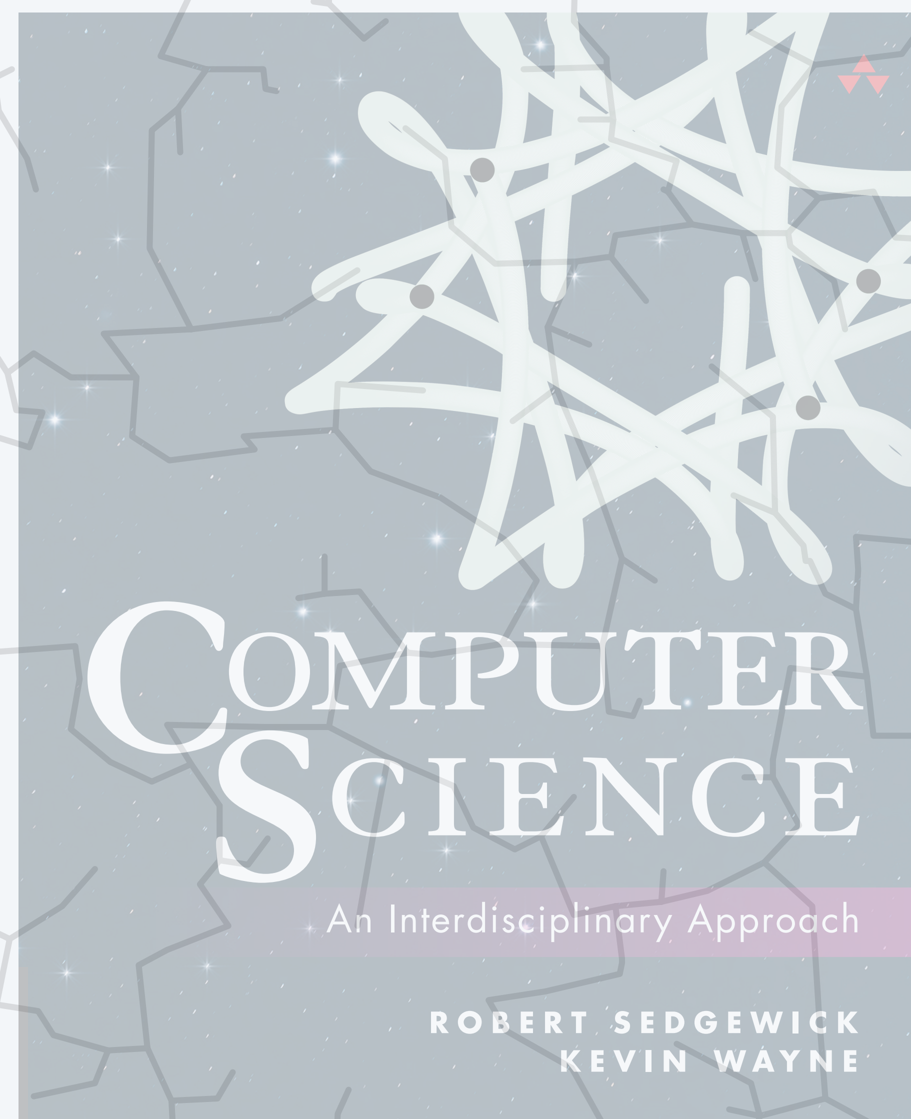
Debugger

- As you move through program, call stack on left
- Local variables on right

```
1  public class DoubleIntegers {
2      public static int[] doubleInts(int[] nums) {   nums: [1, 2, 3]
3          int[] doubles = new int[nums.length];   nums: [1, 2, 3]
4          for (int i = 0; i < nums.length; i++) {
5              doubles[i] = nums[i] * 2;
6          }
7          return doubles;
8      }
```


Call stack

Local variables at start of doubleInts()



https://introcs.cs.princeton.edu

2.1 FUNCTIONS

- *under-the-hood*
- *debugger*
- *what next?*

Next in COS 126 - Recursion

Recursion is when something is specified in terms of itself. ← *self-reference*

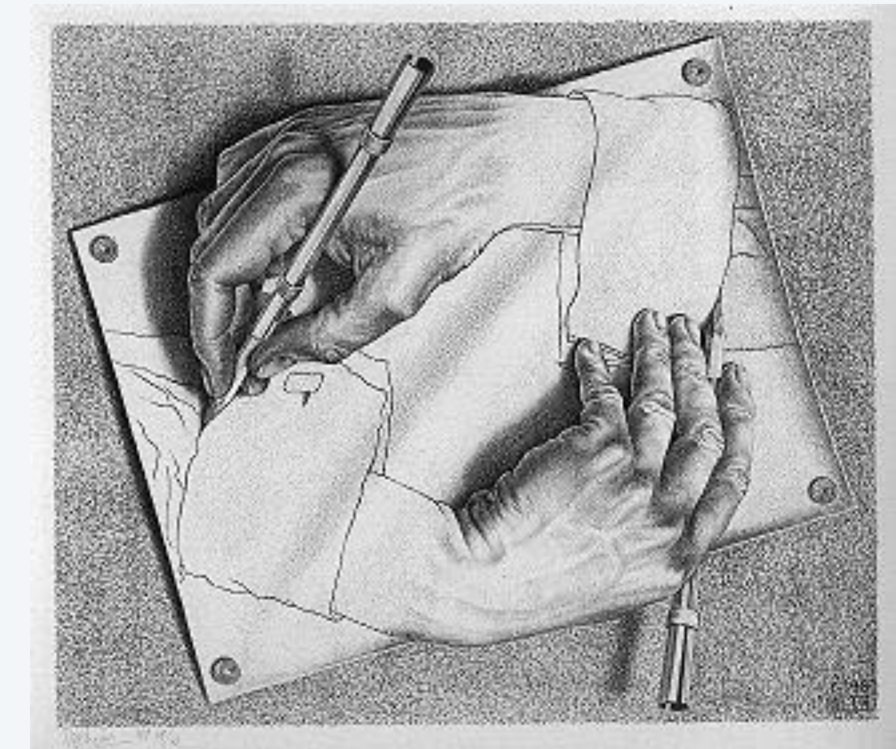
Why learn recursion?

- Powerful programming paradigm.
- Insight into the nature of computation and math. ← *proofs by induction, incompleteness theorems*

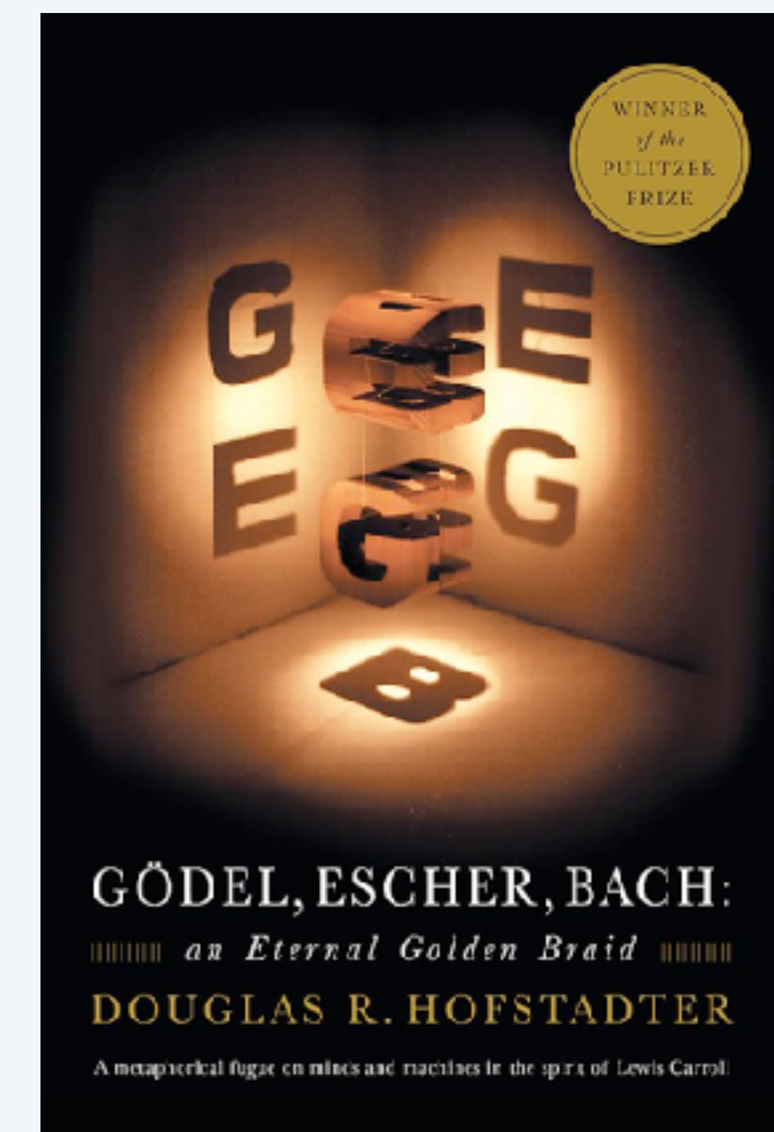
Many computational artifacts are naturally self-referential.

- File system with folders containing folders.
- Binary trees.
- Divide-and-conquer algorithms

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



Drawing Hands,
by M. C. Escher



Object-oriented programming

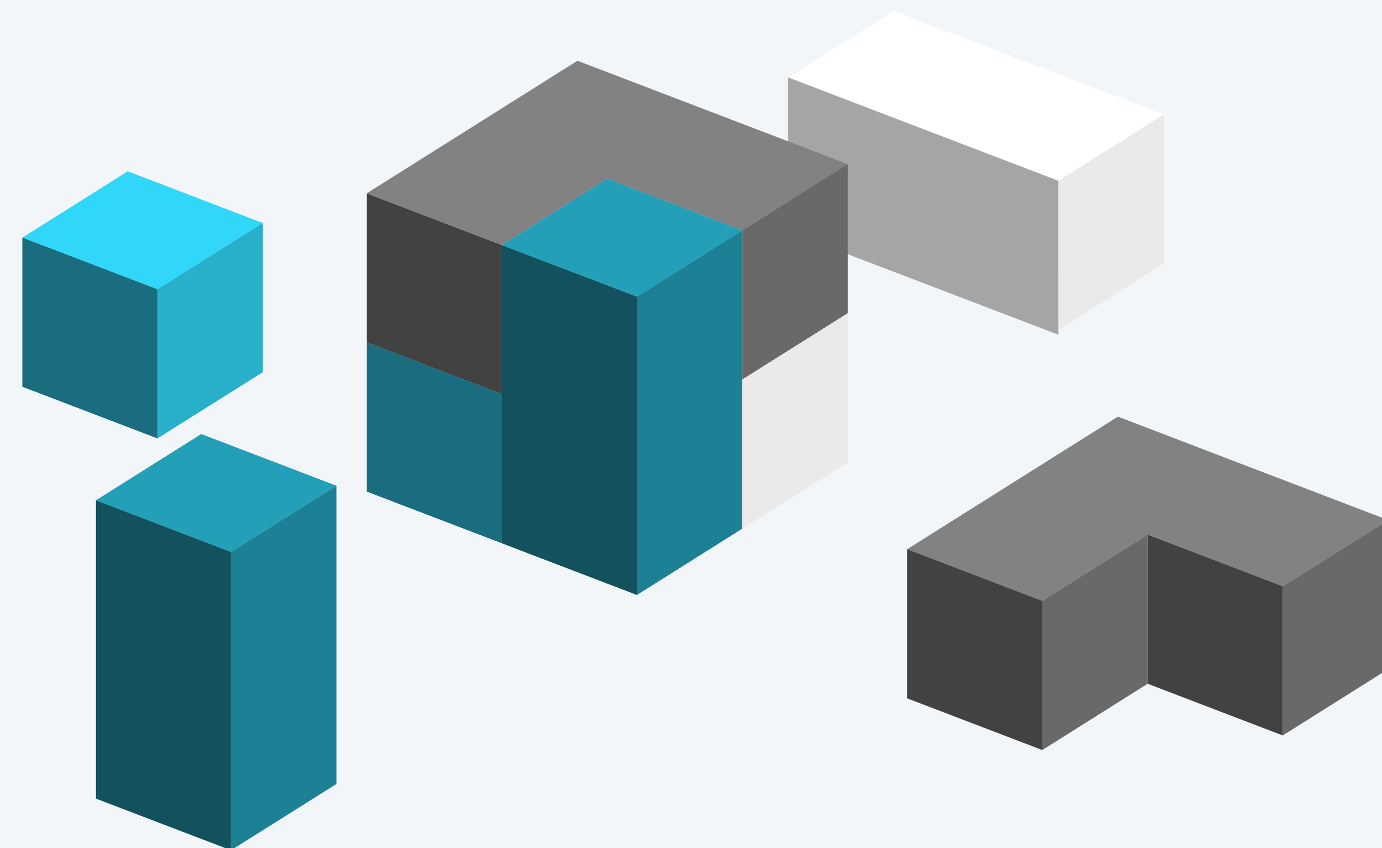
Data type. A set of values and a set of operations on those values.

Java class. Java's mechanism for defining a new data type.

Object. An instance of a data type that has

- State: value from its data type.
- Behavior: actions defined by the data type's operations.
- Identity: unique identifier (e.g. memory address).

```
public class Triangle{
   public static void main(String[] args) {
      Triangle equilateral = new Triangle(4, 4, 4);
      double area = equilateral.Area();
      double perimeter = equilateral.Perimeter();
   }
}
```



Theory of computing

Scenario 1. You just wrote a program that solves Problem A. You're feeling proud (as you should), and think your program is the best.

Can you prove it's the best solution for Problem A?

Scenario 2. You spent hours and hours trying to solve Problem B, but didn't get there. You're smart and know it — so Problem B looks like the issue.

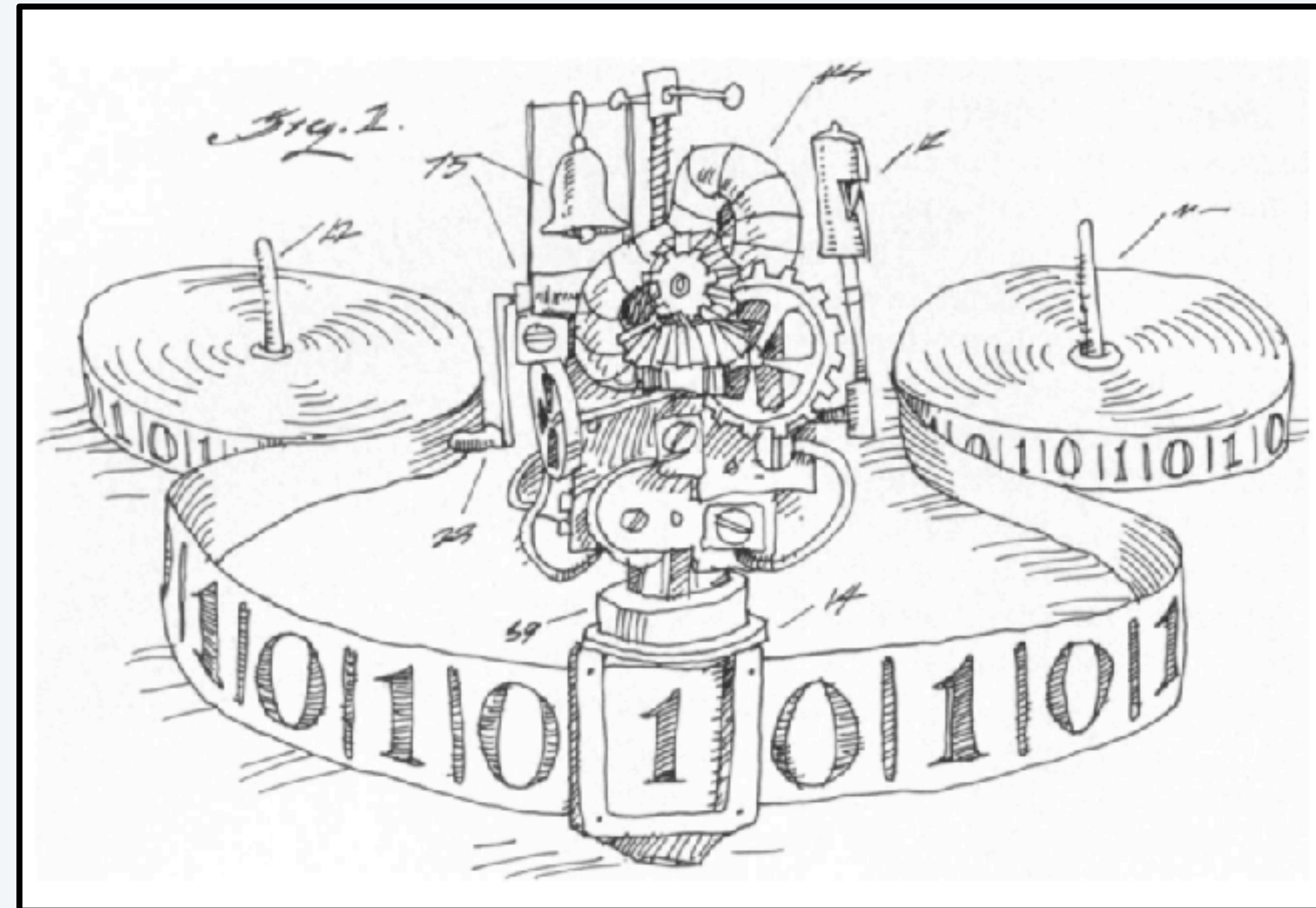
Can you prove Problem B is really hard to solve?

Fundamental questions

Q1. What is an algorithm?

Q2. What is an efficient algorithm?

Q3. Which problems can be solved efficiently?



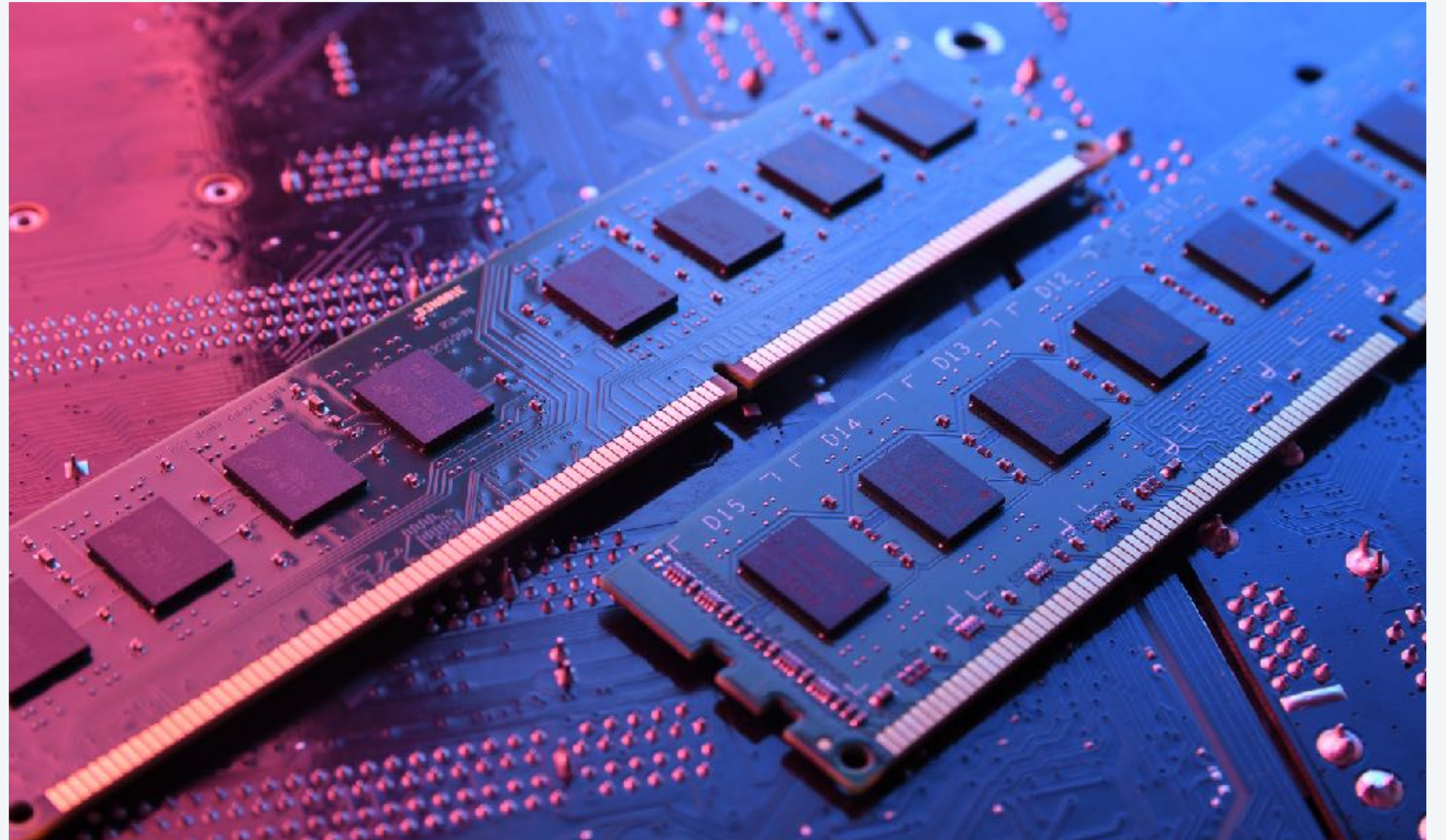
A Turing machine

Systems-level questions

Q1. How does the OS manage computer memory?

Q2. How does programming language compilation work?

Q3. How do virtual machines work?



What we hope you take away from COS 125

- More than just how to use Java
- Programming basics that transfer to most other languages
- Meta-skills
  - Problem-solving
  - Thinking logically
  - Technical communication through written code and verbal explanations

Final exam

Day: August 13th

Place: Jadwin Hall 343

Time: 1:30pm to 2:50pm

Mix of multiple choice and handwritten questions

Closed book, but can bring “cheatsheet:”

- 8.5-by-11 paper, one side, in your own handwriting.

Good luck!

Credits

media	source	license
Gears	Adobe Stock	education license
Function Gradient	Adobe Stock	education license
Function Machine	Wvbailey	public domain
Gödel, Escher, Bach cover	Amazon	
Drawing Hands	Wikipedia	
Happy Programmer	Adobe Stock	education license
Frustrated Programmer	Adobe Stock	
Cartoon of Turing Machine	Tom Dunne	
Pancakes	Adobe Stock	education license
RAM memory	Adobe Stock	education license

Lecture Slides © Copyright 2025 Robert Sedgewick, Kevin Wayne and Marcel Dall'Agnol