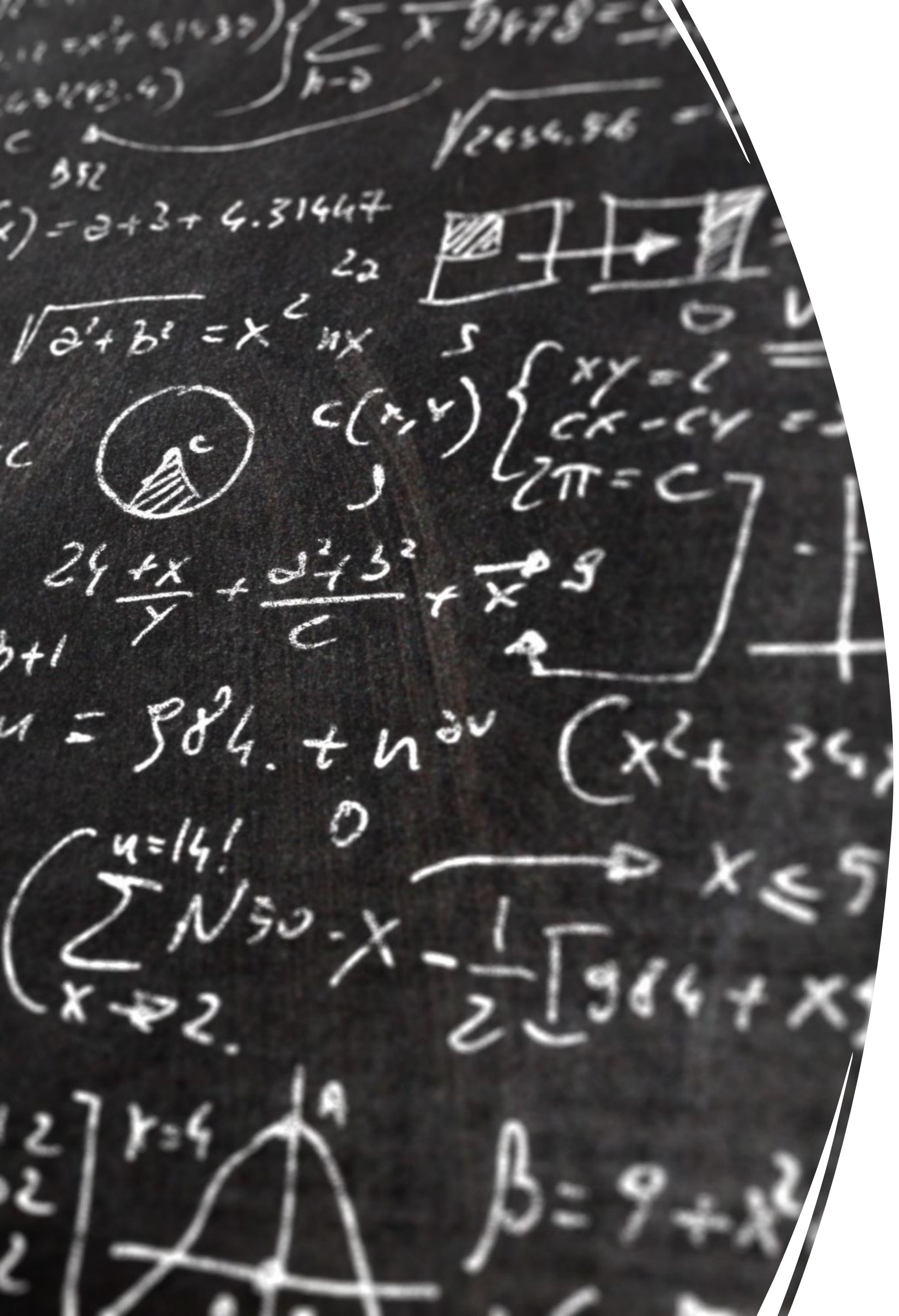


CS 113 – Computer Science I

Lecture 5 – Methods I

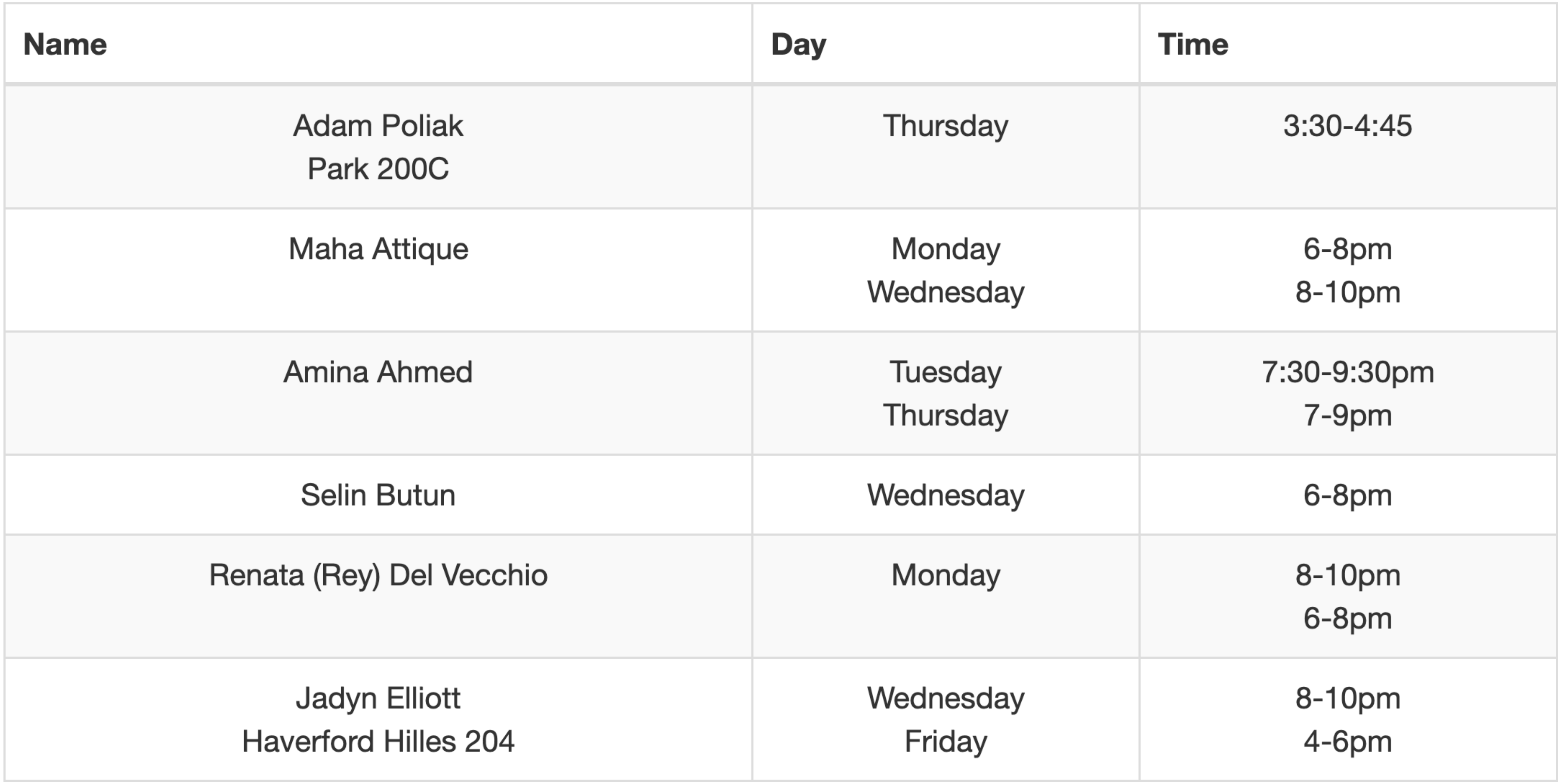
Adam Poliak

01/31/2023

Announcements (1/2)

- Assignment 01 – due 02/01
  - Released today
- Assignment 02 – due 02/08
  - Released tonight or tomorrow
- Great participation on Piazza!

Announcements (2/2) – Office Hours



Name	Day	Time
Adam Poliak Park 200C	Thursday	3:30-4:45
Maha Attique	Monday Wednesday	6-8pm 8-10pm
Amina Ahmed	Tuesday Thursday	7:30-9:30pm 7-9pm
Selin Butun	Wednesday	6-8pm
Renata (Rey) Del Vecchio	Monday	8-10pm 6-8pm
Jadyn Elliott Haverford Hilles 204	Wednesday Friday	8-10pm 4-6pm

Agenda

- Announcements
- Recap
- Methods

Recap - Methods

- Abstractions
- Re-usable portions of code
- Anatomy of a method:
  - Name
  - Parameters
  - Body
  - Return Type
- Signature (everything but body)
- Parameters vs arguments

Executing a function: steps

1. When you encounter a function, pause!
2. Create a *frame* to hold the function’s state
3. Copy argument values
4. Execute the function, line by line. Continue until
    1. you hit a return statement
    2. you run out of statements
5. Send back return value (can be nothing if function is *void*)
6. Delete the function’s frame
7. Resume original function

Agenda

- Announcements
- Recap
- Methods

What is different about these methods?

```java
// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (double), the area as width * height
// side effects: none
public static double area(double width, double height) {
    return width * height;
}
```

```java
// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (none)
// Side effect: prints the area to the console
public static void area(double width, double height) {
     double a = width * height;
     System.out.println("Area is "+ a);
}
```

Warning: don’t confuse printing with returning

```java
// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (double), the area as width * height
// side effects: none
public static double area(double width, double height) {
    return width * height;
}
```

```java
// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (none)
// Side effect: prints the area to the console
public static void area(double width, double height) {
     double a = width * height;
     System.out.println("Area is "+ a);
}
```

Benefits of methods

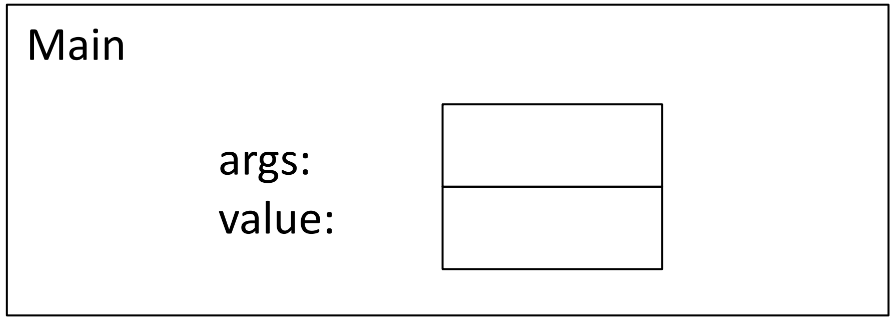
- Split large problems into small problems
- Easier to maintain code/cleaner code
  - Only need to fix mistakes
  - DRY: Don’t repeat yourself
- Implement once, re-use in different programs
- Abstract details so user doesn’t need to worry about details

Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```

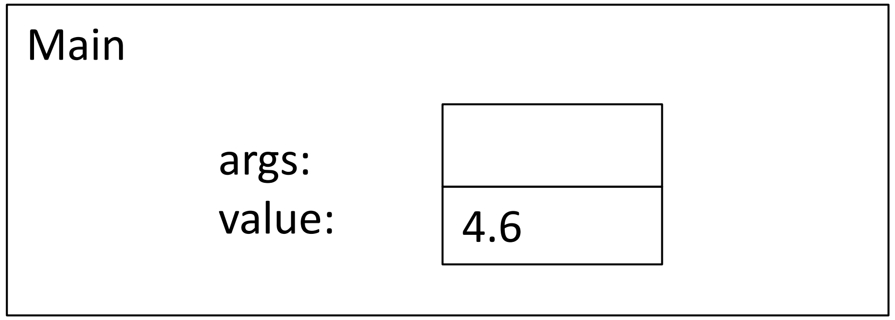
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



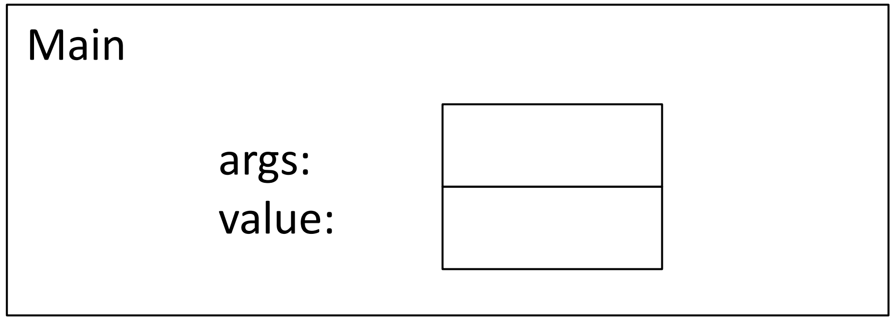
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



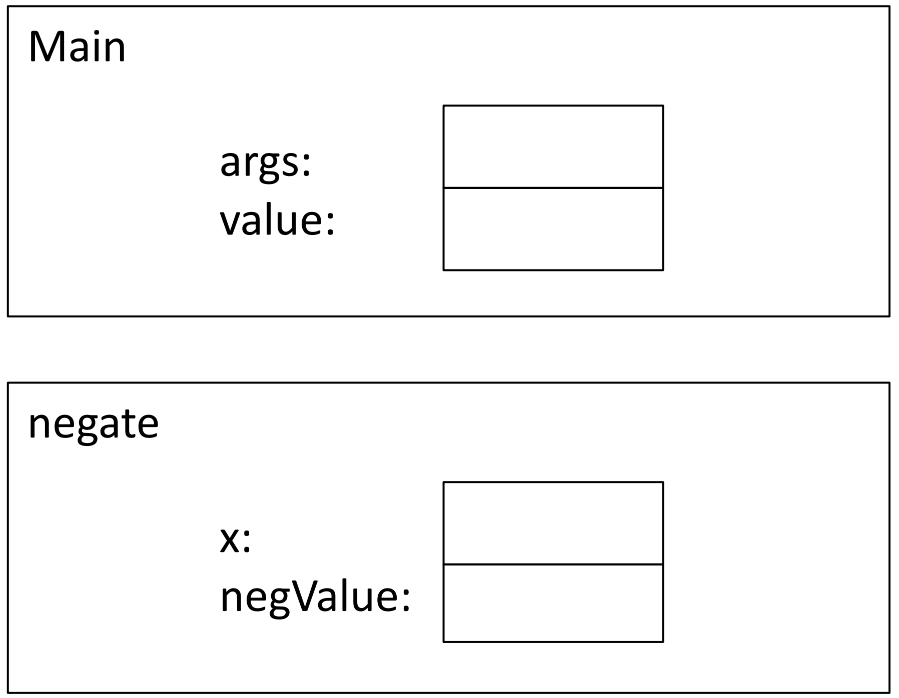
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



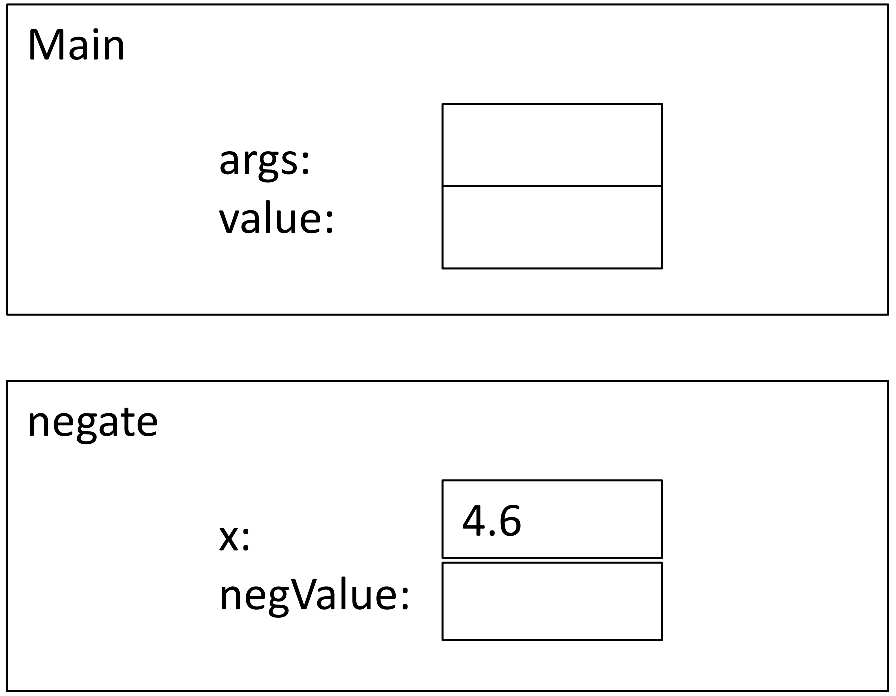
Exercise: Draw stack diagram

```java
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



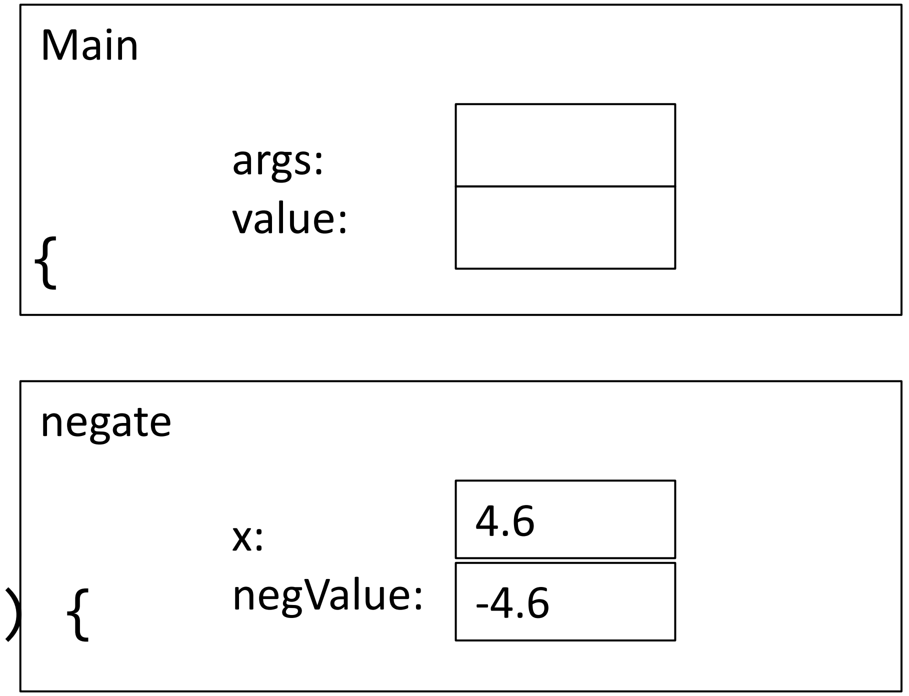
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



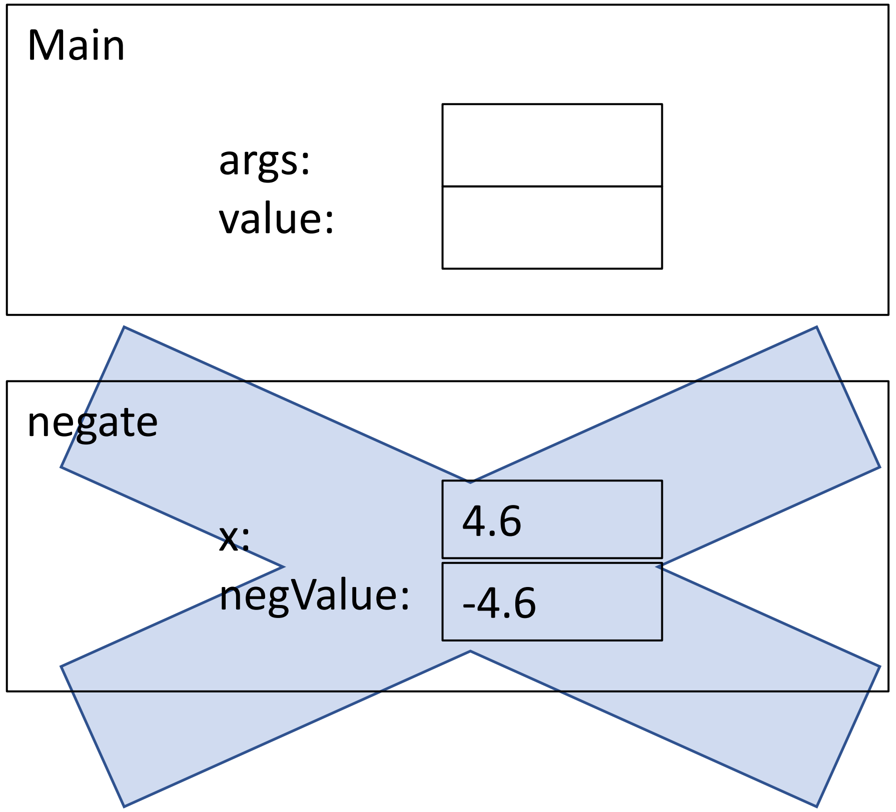
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



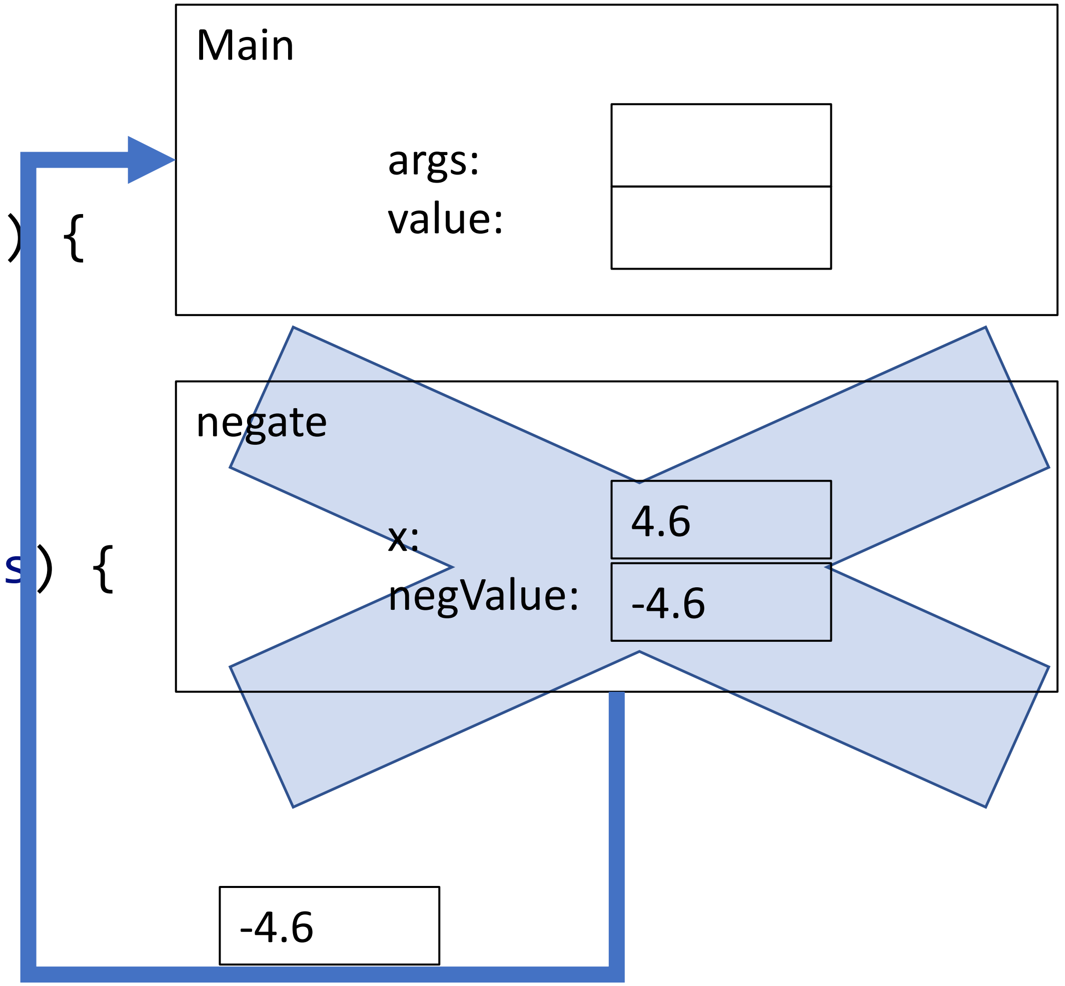
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



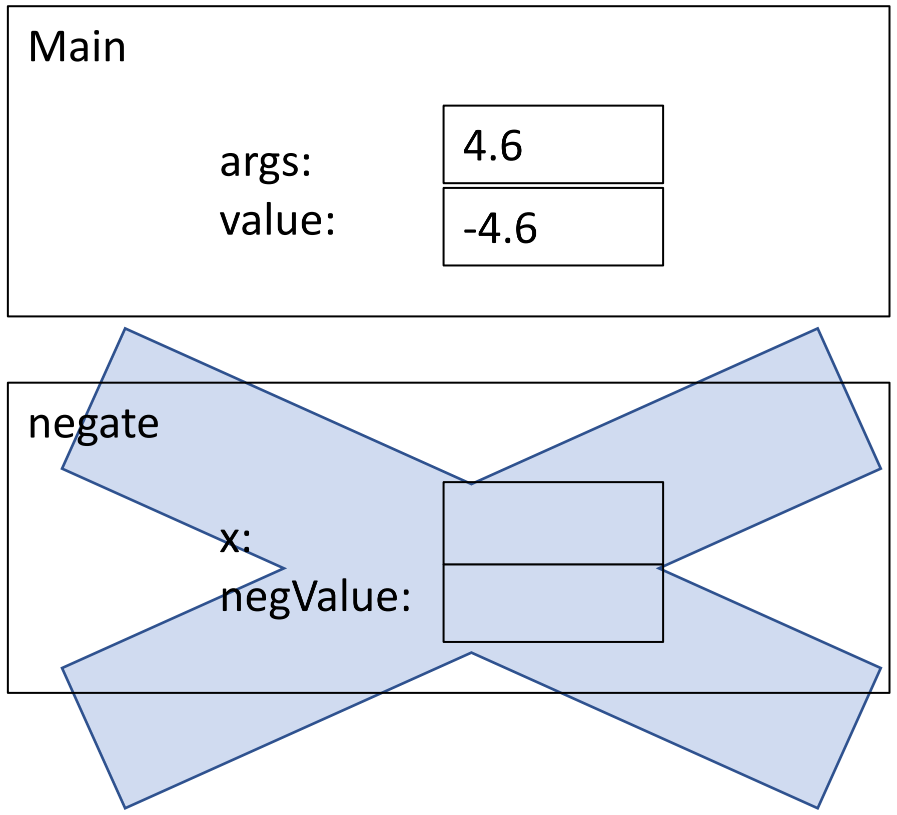
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



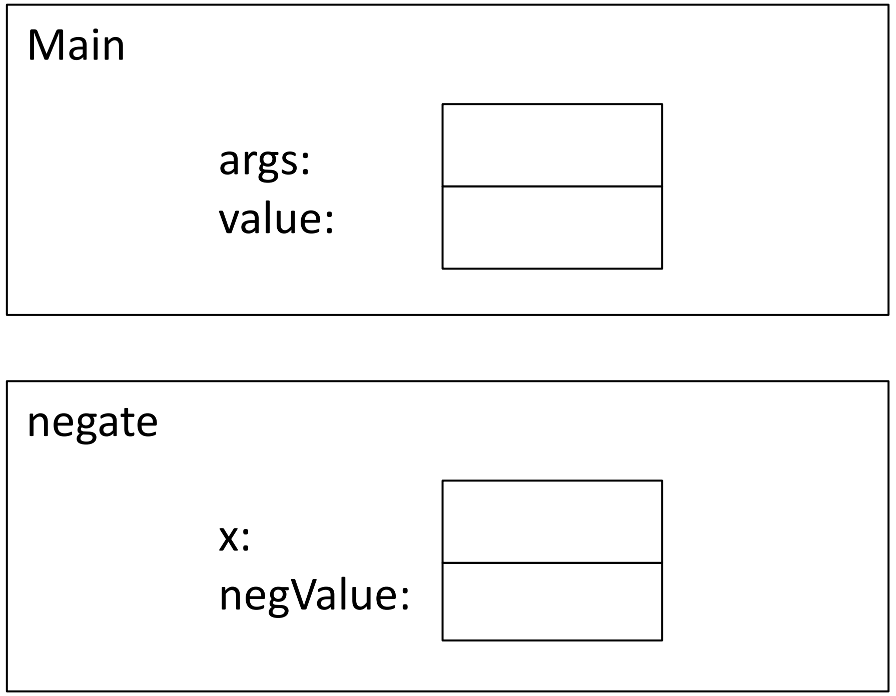
Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = 4.6;
        value = negate(value);
    }
}
```



Exercise: Draw stack diagram

```
public class Negate {

    public static double negate(double x) {
        double negValue = -1 * x
        return negValue;
    }

    public static void main(String[] args) {
        double value = -5.4;
        value = negate(value);
    }
}
```



Scope

- area of a program where a variable can be used
- Stack diagram’s helpful for identifying scope
- Online demo with pythontutor.com: https://pythontutor.com/java.html#mode=edit

What variables are in scope in area()? in main()?

Scope

```
public class Area {

    public static double area(double width, double height) {
        float result = width * height;
        return result;
    }

    public static void main(String[] args) {

        double size = area(10.0, 5);
        System.out.printf("Area is %d\n”, size);
    }
}
```

Method specifications

Idea: “contract” between the function user and the method implementation

Inputs and their types

Return type

Description of how function behaves, including special cases

A **side effect** refers to changes the method makes that last after the method returns (e.g. printing to the console is a side effect)

The **method signature** includes just the inputs and outputs of the function

Why have method specifications?

- Make the behavior of method clear
- Enable user to use method without having to look at the implementation

Method Specifications

```
/**
 * Returns a random real number from a Gaussian distribution with
 *  mean &mu and standard deviation &sigma
 *
 * @param mu the mean
 * @param sigma the std
 * @ return a real number distributed according to the Gaussian distribution
 * /
public static double gaussian(double mu, double sigma) {
	return mu + sigma * gaussian();
}
```

Unit testing

Verify that method is implemented correctly

Call the method with different inputs and check the results

In a library, we can use the main method to test methods

Top down design

1. Identify features of the program
    1. List them out!
2. Identify verbs and nouns in feature list
    1. Verbs: functions
    2. Nouns: objects/variables
3. Sketch major steps – how features should fit together
    1. Algorithm!
4. Write program skeleton
    1. Include function **stubs** (placeholders for our functions)
    2. Function **stub:** empty function with parameters and return type
5. Implement and test function stubs one at a time