Middlebury

CSCI 201: Data Structures

Fall 2025

Lecture 2T: Arrays

Announcements

Announcements

Office hours update

- Tuesday 3-4PM
- Wednesday 3-4PM
- Thursday 4-5PM

Announcements

Office hours update

- Tuesday 3-4PM
- Wednesday 3-4PM
- Thursday 4-5PM

Course assistant drop-in hours (go/cshelp)

- Sunday 6-8PM
- Monday 6-10PM
- Wednesday 6-10PM
- *All sessions in 75 Shannon room 202*

Announcements

Office hours update

- Tuesday 3-4PM
- Wednesday 3-4PM
- Thursday 4-5PM

Course assistant drop-in hours (go/cshelp)

- Sunday 6-8PM
- Monday 6-10PM
- Wednesday 6-10PM
- *All sessions in 75 Shannon room 202*

Upcoming deadlines

- Lab 1 due **today** at 5PM
- Homework 1 due Thursday at 5PM
- *Remember to complete the homework reflection survey!*

Goals for today:

Goals for today:

- Discuss guidelines for style in Java.

Goals for today:

- Discuss guidelines for style in Java.
- Do a mini-introduction to objects and use the new keyword.

Goals for today:

- Discuss guidelines for style in Java.
- Do a mini-introduction to objects and use the new keyword.
- Create and use fixed-size arrays to store many values of the same type.

Goals for today:

- Discuss guidelines for style in Java.
- Do a mini-introduction to objects and use the new keyword.
- Create and use fixed-size arrays to store many values of the same type.
- Modify objects by using a **reference** to the underlying object.

Goals for today:

- Discuss guidelines for style in Java.
- Do a mini-introduction to objects and use the new keyword.
- Create and use fixed-size arrays to store many values of the same type.
- Modify objects by using a **reference** to the underlying object.
- Create and use multi-dimensional arrays.

Goals for today:

- Discuss guidelines for style in Java.
- Do a mini-introduction to objects and use the new keyword.
- Create and use fixed-size arrays to store many values of the same type.
- Modify objects by using a **reference** to the underlying object.
- Create and use multi-dimensional arrays.
- Use the debugger to inspect the value stored in variables.

Let's set up a few style guidelines.

```
int x = 5, y = 20;
String my_name = "Mike Wazowski"; int age = 20;
for(int i = 0; i<values.length;i++){
  if(condition) {

  }
  else
  {

  }
}
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).

```
int x = 5, y = 20;
String my_name = "Mike Wazowski"; int age = 20;
for(int i = 0; i<values.length;i++){
  if(condition) {

  }
  else
  {

  }
}
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.
- Use `CamelCase` for class names, `drinkingCamelCase` for variables/methods, lowercase for single words.

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.
- Use `CamelCase` for class names, `drinkingCamelCase` for variables/methods, lowercase for single words.
- Use spaces between keywords and other control characters (`()`, `{}`) and operators (`&&`, `<`, etc.).

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.
- Use `CamelCase` for class names, `drinkingCamelCase` for variables/methods, lowercase for single words.
- Use spaces between keywords and other control characters (`()`, `{}`) and operators (`&&`, `<`, etc.).
- No space between semi-colon and **previous** character (`i = 0;`, not `i = 0 ;`).

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.
- Use `CamelCase` for class names, `drinkingCamelCase` for variables/methods, lowercase for single words.
- Use spaces between keywords and other control characters (`()`, `{}`) and operators (`&&`, `<`, etc.).
- No space between semi-colon and **previous** character (`i = 0;`, not `i = 0 ;`).
- Use a space between semi-colon and next statement (in definition of `for`-loop).

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.
- Use `CamelCase` for class names, `drinkingCamelCase` for variables/methods, lowercase for single words.
- Use spaces between keywords and other control characters (`()`, `{}`) and operators (`&&`, `<`, etc.).
- No space between semi-colon and **previous** character (`i = 0;`, not `i = 0 ;`).
- Use a space between semi-colon and next statement (in definition of `for`-loop).
- One line, one statement.

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.
- Use `CamelCase` for class names, `drinkingCamelCase` for variables/methods, lowercase for single words.
- Use spaces between keywords and other control characters (`()`, `{}`) and operators (`&&`, `<`, etc.).
- No space between semi-colon and **previous** character (`i = 0;`, not `i = 0 ;`).
- Use a space between semi-colon and next statement (in definition of `for`-loop).
- One line, one statement.

```
1  int x = 5, y = 20;
2  String my_name = "Mike Wazowski"; int age = 20;
3  for(int i = 0; i<values.length;i++){
4    if(condition) {
5
6    }
7    else
8    {
9
10   }
11 }
```

```
1  int x = 5;
2  int y = 20;
3  String myName = "Mike Wazowski";
4  int age = 20;
5  for (int i = 0; i < values.length; i++) {
6    if (condition) {
7
8    } else {
9
10   }
11 }
```

Make your code concise without sacrificing readability.

```java
public class CircularSentenceChecker {
  public static boolean isCircularSentence(String sentence) {
    sentence = sentence.toLowerCase();
    int n = sentence.length();
    for (int i = 0; i < n; i++) {
        if (sentence.charAt(i) == ' ') {
            if (sentence.charAt(i - 1) != sentence.charAt(i + 1)) {
              return false;
            }
        }
    }
    if (sentence.charAt(n - 1) == sentence.charAt(0)) {
      return true;
    } else {
      return false;
    }
  }
}
```

Think about where your `returns` are.

```java
public class CircularSentenceChecker {
  public static boolean isCircularSentence(String sentence) {
    sentence = sentence.toLowerCase();
    int n = sentence.length();
    boolean isCircular = true;
    for (int i = 0; i < n; i++) {
        if (sentence.charAt(i) == ' ') {
            if (sentence.charAt(i - 1) != sentence.charAt(i + 1)) {
              isCircular = false;
            }
        }
    }
    if (sentence.charAt(n - 1) != sentence.charAt(0)) {
      isCircular = false;
    }
    return isCircular;
  }
}
```

Exercise: find the style problems.

```
1  public class CircularSentenceChecker {
2    public static boolean isCircularSentence(String sentence) {
3      String low_sent = sentence.toLowerCase();
4      if (low_sent.charAt(0) != low_sent.charAt(low_sent.length() - 1)) {
5        return false ;
6      }
7
8      for(int i = 0;i < low_sent.length() - 2;i++) {
9        if (low_sent.charAt(i + 1)==' ' && low_sent.charAt(i) != low_sent.charAt(i + 2)){
10         return false;
11       }
12     }
13
14     return true;
15   }
16 }
```

Remember the `String` example from last class? Here's a variation.

```
1  public class StringEquals {
2    public static void main(String[] args) {
3
4      String s = "Middlebury";
5      String a = s.substring(0, 4); // "Midd"
6      String b = "Midd";
7
8      boolean sameByEqualityOperator = (a == b);
9      boolean sameByEqualsMethod = a.equals(b);
10
11     System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);
12     System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);
13   }
14 }
```

Remember the `String` example from last class? Here's a variation.

```
1  public class StringEquals {
2    public static void main(String[] args) {
3
4      String s = "Middlebury";
5      String a = s.substring(0, 4); // "Midd"
6      String b = "Midd";
7
8      boolean sameByEqualityOperator = (a == b);
9      boolean sameByEqualsMethod = a.equals(b);
10
11     System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);
12     System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);
13   }
14 }
```

Remember the `String` example from last class? Here's a variation.

```
1  public class StringEquals {
2    public static void main(String[] args) {
3
4      String s = "Middlebury";
5      String a = s.substring(0, 4); // "Midd"
6      String b = "Midd";
7
8      boolean sameByEqualityOperator = (a == b);
9      boolean sameByEqualsMethod = a.equals(b);
10
11     System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);
12     System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);
13   }
14 }
```

Remember the `String` example from last class? Here's a variation.

```
1  public class StringEquals {
2    public static void main(String[] args) {
3
4      String s = "Middlebury";
5      String a = s.substring(0, 4); // "Midd"
6      String b = "Midd";
7
8      boolean sameByEqualityOperator = (a == b);
9      boolean sameByEqualsMethod = a.equals(b);
10
11     System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);
12     System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);
13   }
14 }
```

Remember the `String` example from last class? Here's a variation.

```
1  public class StringEquals {
2    public static void main(String[] args) {
3
4      String s = "Middlebury";
5      String a = s.substring(0, 4); // "Midd"
6      String b = "Midd";
7
8      boolean sameByEqualityOperator = (a == b);
9      boolean sameByEqualsMethod = a.equals(b);
10
11     System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);
12     System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);
13   }
14 }
```

Remember the `String` example from last class? Here's a variation.

```
1  public class StringEquals {
2    public static void main(String[] args) {
3
4      String s = "Middlebury";
5      String a = s.substring(0, 4); // "Midd"
6      String b = "Midd";
7
8      boolean sameByEqualityOperator = (a == b);
9      boolean sameByEqualsMethod = a.equals(b);
10
11     System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);
12     System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);
13   }
14 }
```

Why did this happen? What is `==` actually comparing?

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

But we can also create `String`s like this:

```
1 String s = new String("Middlebury");
```

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

But we can also create `String`s like this:

```
1 String s = new String("Middlebury");
```

Because strings are *objects*.

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

But we can also create `String`s like this:

```
1 String s = new String("Middlebury");
```

Because strings are *objects*.

- Objects are created using the `new` keyword.

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

But we can also create `String`s like this:

```
1 String s = new String("Middlebury");
```

Because strings are *objects*.

- Objects are created using the `new` keyword.
- When objects are created, the resulting variable is a *reference.*

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

But we can also create `String`s like this:

```
1 String s = new String("Middlebury");
```

Because strings are *objects*.

- Objects are created using the `new` keyword.
- When objects are created, the resulting variable is a *reference.*
- A reference is really an address to a place in memory where the object is stored.

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

But we can also create `String`s like this:

```
1 String s = new String("Middlebury");
```

Because strings are *objects*.

- Objects are created using the `new` keyword.
- When objects are created, the resulting variable is a *reference.*
- A reference is really an address to a place in memory where the object is stored.
- We can pass around this address to make changes to the underlying object.

A `String` is a class. An instance of a `String` is an `Object`.

We have been creating `String`s like this (special syntax for `String`s):

```
1 String s = "Middlebury";
```

But we can also create `String`s like this:

```
1 String s = new String("Middlebury");
```

Because strings are *objects*.

- Objects are created using the `new` keyword.
- When objects are created, the resulting variable is a *reference.*
- A reference is really an address to a place in memory where the object is stored.
- We can pass around this address to make changes to the underlying object.
- The `==` operator will check if the addresses are the same.

Back to our `String` example.

```
1  public class StringEquals {
2    public static void main(String[] args) {
3
4      String a = new String("Midd");
5      String b = new String("Midd");
6
7      boolean sameByEqualityOperator = (a == b);
8      boolean sameByEqualsMethod = a.equals(b);
9
10     System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);
11     System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);
12   }
13 }
```

Introducing fixed-size arrays! Size is fixed upon creating the array.

Introducing fixed-size arrays! Size is fixed upon creating the array.

Unlike lists in Python, where the size can change.

```
1 values = []
2 for i in range(10):
3     values.append(i)
```

Introducing fixed-size arrays! Size is fixed upon creating the array.

Unlike lists in **Python**,
where the size can change.

```
1 values = []
2 for i in range(10):
3     values.append(i)
```

We'll see dynamically-sized arrays in **Java** next week.

Fixed-sized arrays can be used to store multiple values of the same type.

```
1  public class ArrayExamples {
2    public static void main(String[] args) {
3
4      int[] values1 = new int[6]; // creates an array of integers with 6 values
5      values1[0] = 311;
6      values1[1] = 312;
7      values1[2] = 315;
8      values1[3] = 318;
9      values1[4] = 333;
10     values1[5] = 467;
11
12     int[] values2 = {311, 312, 315, 318, 333, 467};
13     int[] values3 = new int[]{311, 312, 315, 318, 333, 467};
14   }
15 }
```

Fixed-sized arrays can be used to store multiple values of the same type.

```
1  public class ArrayExamples {
2    public static void main(String[] args) {
3
4      int[] values1 = new int[6]; // creates an array of integers with 6 values
5      values1[0] = 311;
6      values1[1] = 312;
7      values1[2] = 315;
8      values1[3] = 318;
9      values1[4] = 333;
10     values1[5] = 467;
11
12     int[] values2 = {311, 312, 315, 318, 333, 467};
13     int[] values3 = new int[]{311, 312, 315, 318, 333, 467};
14   }
15 }
```

Fixed-sized arrays can be used to store multiple values of the same type.

```
1  public class ArrayExamples {
2    public static void main(String[] args) {
3
4      int[] values1 = new int[6]; // creates an array of integers with 6 values
5      values1[0] = 311;
6      values1[1] = 312;
7      values1[2] = 315;
8      values1[3] = 318;
9      values1[4] = 333;
10     values1[5] = 467;
11
12     int[] values2 = {311, 312, 315, 318, 333, 467};
13     int[] values3 = new int[]{311, 312, 315, 318, 333, 467};
14   }
15 }
```

Fixed-sized arrays can be used to store multiple values of the same type.

```
1  public class ArrayExamples {
2    public static void main(String[] args) {
3
4      int[] values1 = new int[6]; // creates an array of integers with 6 values
5      values1[0] = 311;
6      values1[1] = 312;
7      values1[2] = 315;
8      values1[3] = 318;
9      values1[4] = 333;
10     values1[5] = 467;
11
12     int[] values2 = {311, 312, 315, 318, 333, 467};
13     int[] values3 = new int[]{311, 312, 315, 318, 333, 467};
14   }
15 }
```

Fixed-sized arrays can be used to store multiple values of the same type.

```
1  public class ArrayExamples {
2    public static void main(String[] args) {
3
4      int[] values1 = new int[6]; // creates an array of integers with 6 values
5      values1[0] = 311;
6      values1[1] = 312;
7      values1[2] = 315;
8      values1[3] = 318;
9      values1[4] = 333;
10     values1[5] = 467;
11
12     int[] values2 = {311, 312, 315, 318, 333, 467};
13     int[] values3 = new int[]{311, 312, 315, 318, 333, 467};
14   }
15 }
```

Fixed-sized arrays can be used to store multiple values of the same type.

```
1  public class ArrayExamples {
2    public static void main(String[] args) {
3
4      int[] values1 = new int[6]; // creates an array of integers with 6 values
5      values1[0] = 311;
6      values1[1] = 312;
7      values1[2] = 315;
8      values1[3] = 318;
9      values1[4] = 333;
10     values1[5] = 467;
11
12     int[] values2 = {311, 312, 315, 318, 333, 467};
13     int[] values3 = new int[]{311, 312, 315, 318, 333, 467};
14   }
15 }
```

Fixed-sized arrays can be used to store multiple values of the same type.

```java
public class ArrayExamples {
  public static void main(String[] args) {

    int[] values1 = new int[6]; // creates an array of integers with 6 values
    values1[0] = 311;
    values1[1] = 312;
    values1[2] = 315;
    values1[3] = 318;
    values1[4] = 333;
    values1[5] = 467;

    int[] values2 = {311, 312, 315, 318, 333, 467};
    int[] values3 = new int[]{311, 312, 315, 318, 333, 467};
  }
}
```

Retrieve the length using `.length`.

We're not calling a method! So no `()`. We're accessing a property.

Enhanced `for`-loops (or *for-each* loops)

```
1 int[] intValues = {311, 312, 315, 318, 333, 467};
2 for (int value : intValues) {
3   System.out.println("value = " + value);
4 }
5
6 String[] stringValues = {"apple", "banana", "pear", "orange"};
7 for (String value : stringValues) {
8   System.out.println("value = " + value);
9 }
```

This can be done with objects that are *iterable* (like arrays).

Enhanced `for`-loops (or *for-each* loops)

```
1 int[] intValues = {311, 312, 315, 318, 333, 467};
2 for (int value : intValues) {
3   System.out.println("value = " + value);
4 }
5
6 String[] stringValues = {"apple", "banana", "pear", "orange"};
7 for (String value : stringValues) {
8   System.out.println("value = " + value);
9 }
```

This can be done with objects that are *iterable* (like arrays).

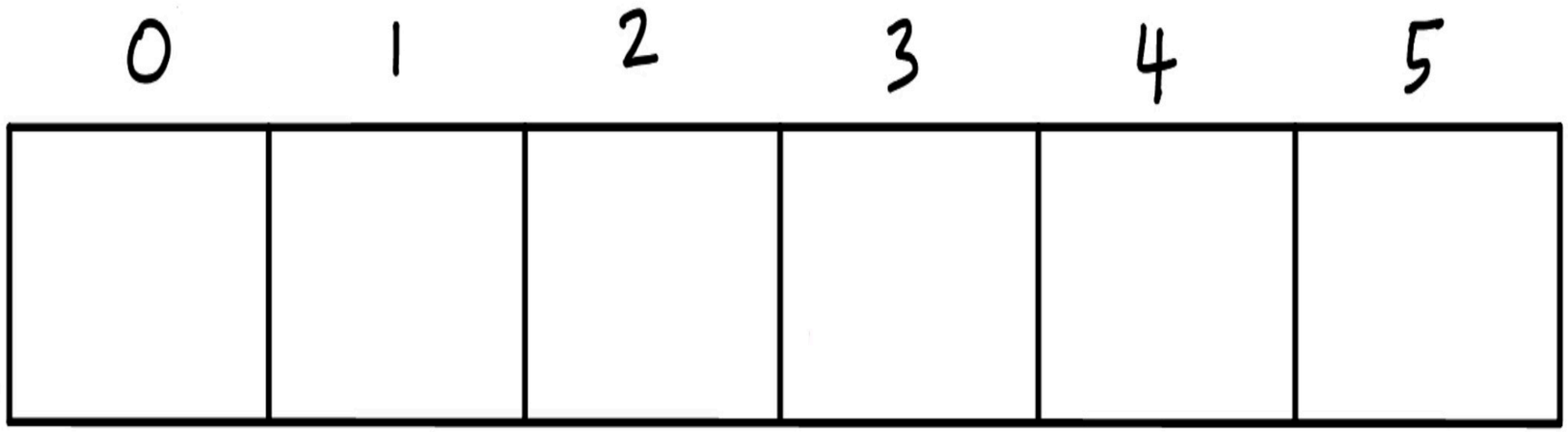
Enhanced `for`-loops (or *for-each* loops)

```
1 int[] intValues = {311, 312, 315, 318, 333, 467};
2 for (int value : intValues) {
3   System.out.println("value = " + value);
4 }
5
6 String[] stringValues = {"apple", "banana", "pear", "orange"};
7 for (String value : stringValues) {
8   System.out.println("value = " + value);
9 }
```

This can be done with objects that are *iterable* (like arrays).

The variables we declare for arrays are also references.

```
1 int[] values0 = {311, 312, 315, 318, 333, 467};
2
3 int[] values1 = values0; // values1 has the same reference as values0
4
5 values1[3] = 321; // change item in values1
6
7 System.out.println(values0[3]);
```

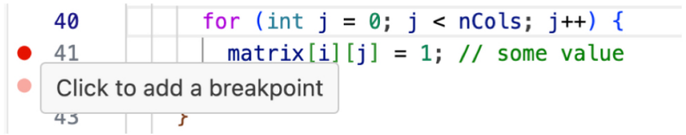


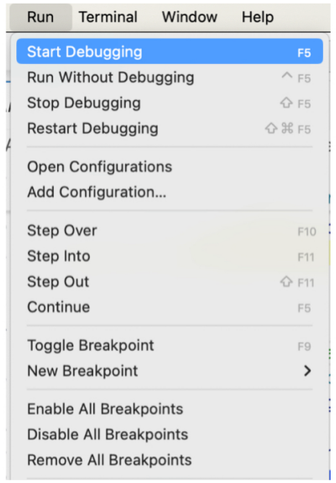
Multi-dimensional arrays: can be rectangular or jagged.

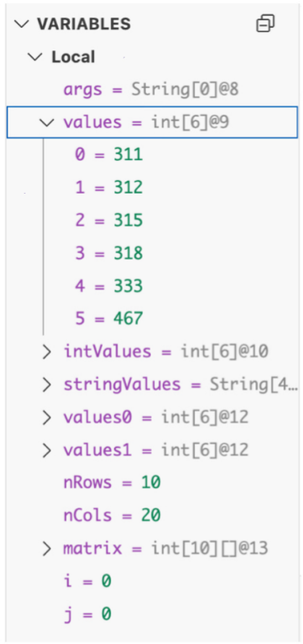
```java
// rectangular: all rows have the same size
int nRows = 10;
int nCols = 20;
int[][] matrix = new int[nRows][nCols];
for (int i = 0; i < nRows; i++) {
  for (int j = 0; j < nCols; j++) {
    matrix[i][j] = 1; // some value
  }
}

// jagged: each row has a different size
char[][] triangle = new char[nRows][]; // nRows arrays, initially null
for (int i = 0; i < triangle.length; i++) {
  triangle[i] = new char[i + 1];
  for (int j = 0; j < triangle[i].length; j++) {
    triangle[i][j] = '*';
  }
}
```

Using the VS Code debugger







Using arrays in practice

Which of the following are good use cases for fixed-size arrays?

1. Storing the state of checkers game
2. Storing information about a person like their name, birth year, and address
3. Storing a playlist of songs

Using arrays in practice

Which of the following are good use cases for fixed-size arrays?

1. Storing the state of checkers game
2. Storing information about a person like their name, birth year, and address
3. Storing a playlist of songs

Can you write a function that takes in a fixed-size array of numbers and returns a new fixed-size array with only the positive numbers in it?

Group exercises

See Practice 3 problems