Unit 01:
Introduction to Java

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Unit 01 Overview

- Related Reading:
  - Textbook pages 4-11, 15, 26-28
- Learning Objectives: (You should be able to…)
  - Write, compile, and execute a Java program with all of the functionality of a program you could write in C/Python (depending on your pre-requisite course)
  - Specific Java examples we will focus on include:
    - variables
    - functions
    - if/switch statements
    - loops
    - arrays

Our First Java Program

- Output `"Welcome to csc115"` when the program is executed

- In C:

```c
int main(void) {
   printf("Welcome to csc115\n");
   return 0;
}
```

- In Python:

```python
def main():
   print("Welcome to csc115")

main()
```

- Similarities

We have written a main function in both versions (but in C it is automatically called when the program executes)

We are calling *printf*/*print* and specifying what we want to output to the user when the program is executed

Our First Java Program

- Output `"Welcome to csc115"` when the program is executed

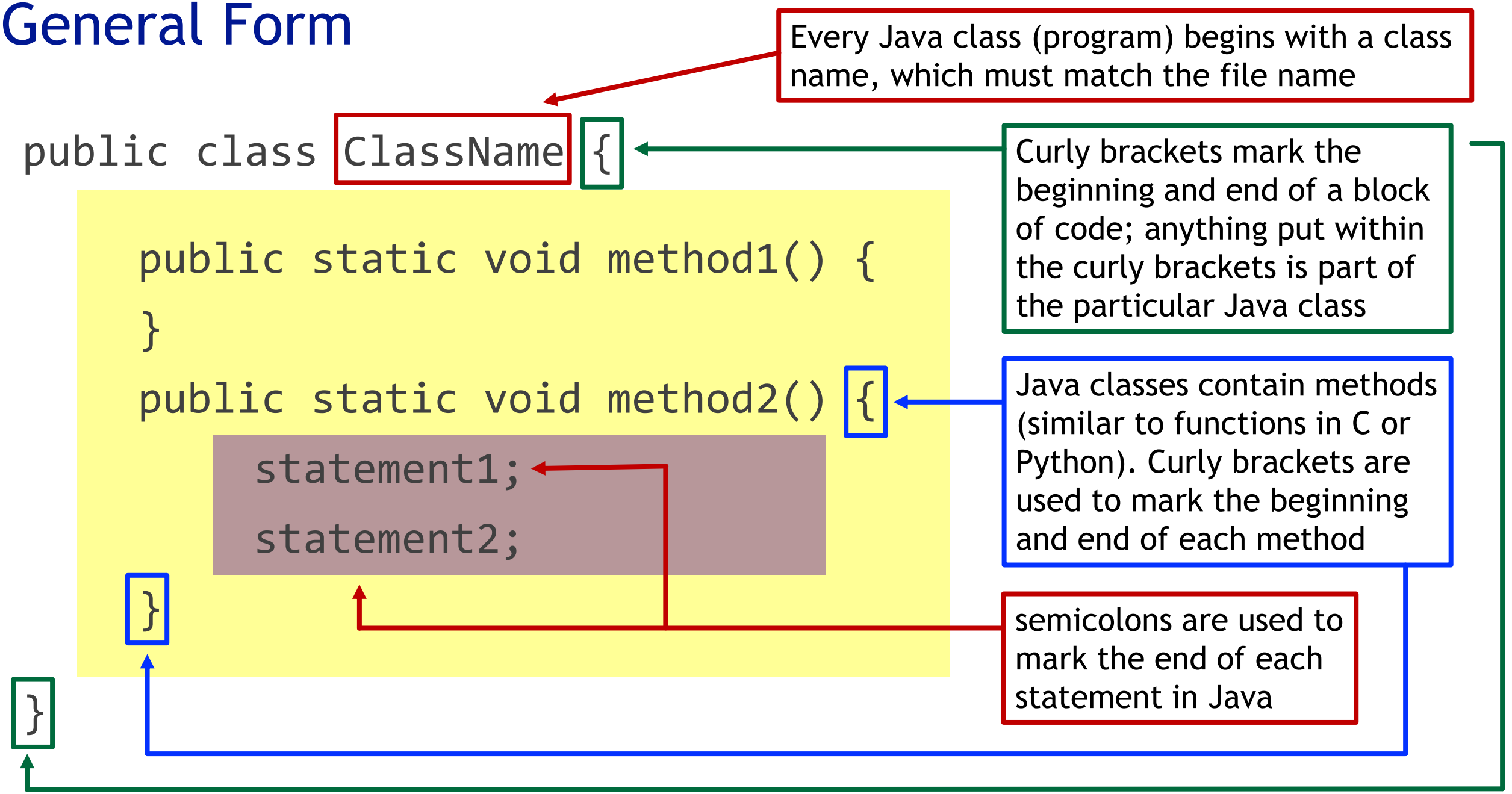
- In Java:

```
public class MyFirstJavaProgram {
   public static void main(String[] args) {
            System.out.println("Welcome to csc115");
   }
}
```

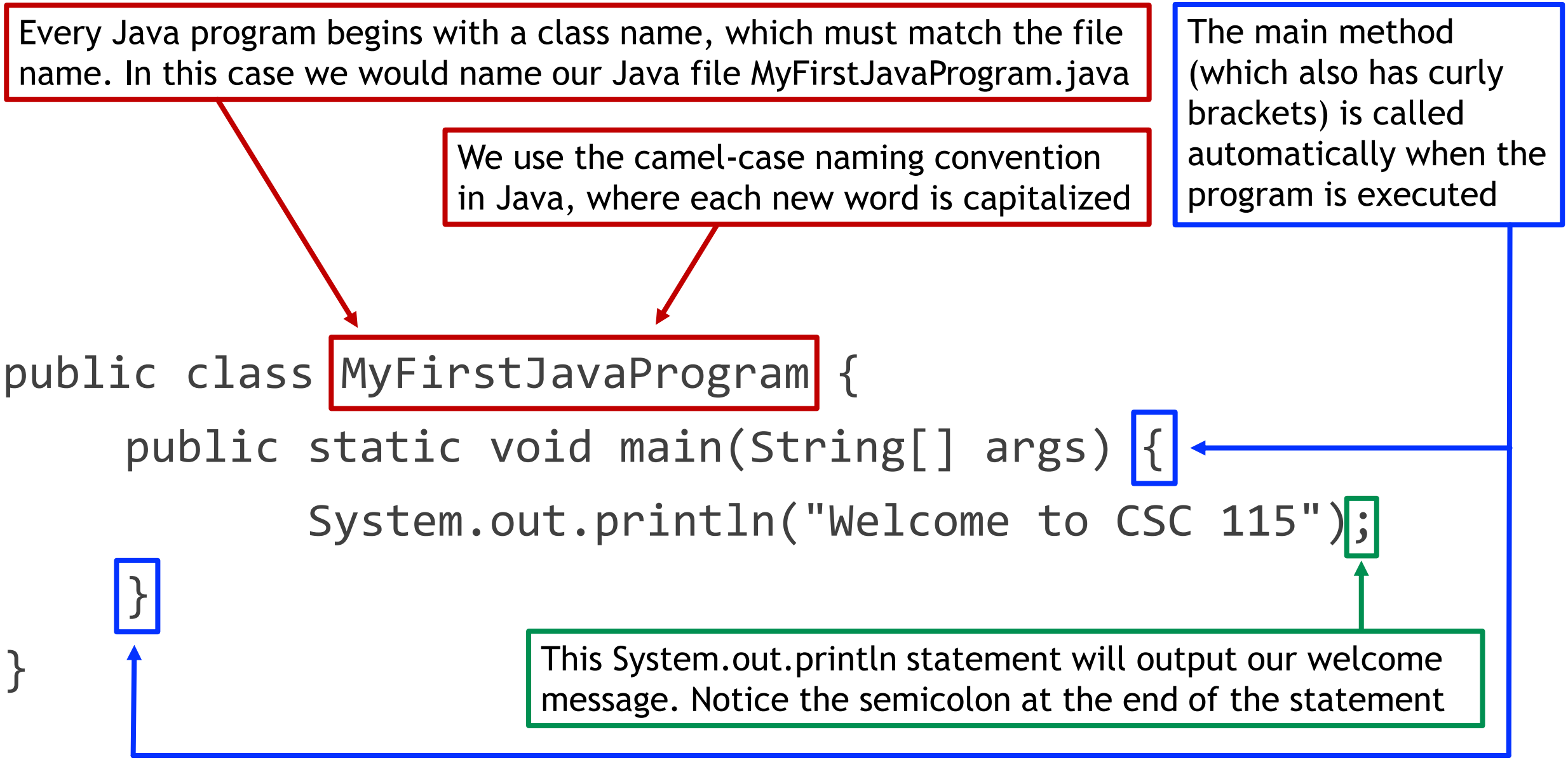
- Similarities
  - Also has a main and a print…

but there are also quite a few differences, so let's look at this piece-by-piece

General Form



Specific Example



Running a Java Program

- In this course we will compile and execute our Java programs using a command-line interface:
  - Command prompt (cmd) in Windows
  - Terminal in Mac OSX

- You can use any editor you wish, here are some recommendations:
  - Notepad++ (Windows)
  - Sublime (Windows/Mac)
  - Atom (Windows/Mac)

Demo

- There are lecture videos which overview the installation process, using the command line to compile and execute a Java program, and writing a first Java program:
  - Installation (Windows, Mac)
  - Command-line
  - First Java Program

- A short list of useful commands:
  - change directory: cd <dir_name> (cd .. goes to parent directory)
  - list directory contents: ls (Mac), dir (windows)
  - compile a Java program: javac <file_name>.java
  - execute a Java program: java <file_name> (with no .java at the end)

Recap: Quick (but important) Java Points

- What you have previously referred to as functions we will call methods in Java (we will explore why later in the course)
- To execute a Java program there are two phases:
  - compilation: determines if there are any errors in the syntax, and generates a machine executable file if no errors are found
  - execution: run the specified Java program. The main method is called automatically; no other methods will execute unless called
- Lines of code in Java typically end with curly brackets or a semicolon
  - curly brackets { } are used to mark the beginning and end of a block of code (code inside a method, or to be repeated in a loop, or in an if-statement)
  - semicolons ; are used to mark the end of each statement in Java (a line of code that performs a task)

Comments

- Similar to how we used comments in Python and C, we can use comments in Java:

```
// This is a single-line comment


/*
   Multi-line comments are also possible.
   Remember to the close the multi-line comment so your
   whole code is not commented out!
*/
```

Primitive Types

These are the primitive data types we will use for variables in Java

- From the textbook:

Category	Data Type	Wrapper Class
Boolean	boolean	Boolean
Character	char	Character
Integer	byte	Byte
	short	Short
	int	Integer
	long	Long
Floating point	float	Float
	double	Double

Typically we will use **int** for whole numbers, and **double** for decimal numbers

Later, we will explore some of the added features the Wrappers classes provide

Variables

- Represent a named memory location
- Memory location contains a value of the primitive type
- General form:

Declaration:

```
type name = value;
```

Update:

```
name = newValue;
```

- Specific examples:

```
int num = 5;     // num is 5
double gradeAverage = 86.4;
```

```
num = num + 3;   // num is now 8
```

Short-form variable operations

Short form	Equivalent	Description
`x += y;`	`x = x + y;`	set the value of variable x to be x + y (increase the value of x by y)
`x -= y;`	`x = x - y;`	subtract y from the value of variable x
`x *= y;`	`x = x * y;`	multiply the variable x by y
`x /= y;`	`x = x / y;`	divide the variable x by y
`x %= y;`	`x = x % y;`	set the value of variable x to be the remainder of dividing x by y
`x++;`	`x = x + 1;`	increment the value of x by 1
`x--;`	`x = x - 1;`	decrement the value of x by 1

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

- Output:

```

```

Memory:

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

- Output:

```

```

Memory:

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 5

- Output:

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 5

- Output:

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 5

b: 7.41

- Output:

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 5

b: 7.41

- Output:

```
5
```

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 5

b: 7.41

- Output:

```
5
7.41
```

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 12

b: 7.41

- Output:

```
5
7.41
```

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 12

b: 6.41

- Output:

```
5
7.41
```

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 12

b: 6.41

- Output:

```
5
7.41
12
```

Variables: Example

- Code:

```
public class VariablesExample {
	public static void main(String[] args) {
		int a = 5;
		double b = 7.41;
		System.out.println(a);
		System.out.println(b);
		a += 7;
		b--;
		System.out.println(a);
		System.out.println(b);
	}
}
```

Memory:

a: 12

b: 6.41

- Output:

```
5
7.41
12
6.41
```

Casting

- Allows us to convert a value from one type to another

- Implicit casting:
  - Converting a variable to another of higher precision
  - Example:

```
int i = 4;          // i is 4
double j = i;       // j is 4.0
```

- Explicit casting:
  - Forcing the conversion of a variable to a specified type
  - Example:

```
double x = 4.95;    // x is 4.95
int y = (int) x;    // y is 4
```

Beyond Primitive Types

- Variables can also store references to (the address of) where larger objects are stored in memory
- This is how Strings, Arrays, and Objects are stored in memory

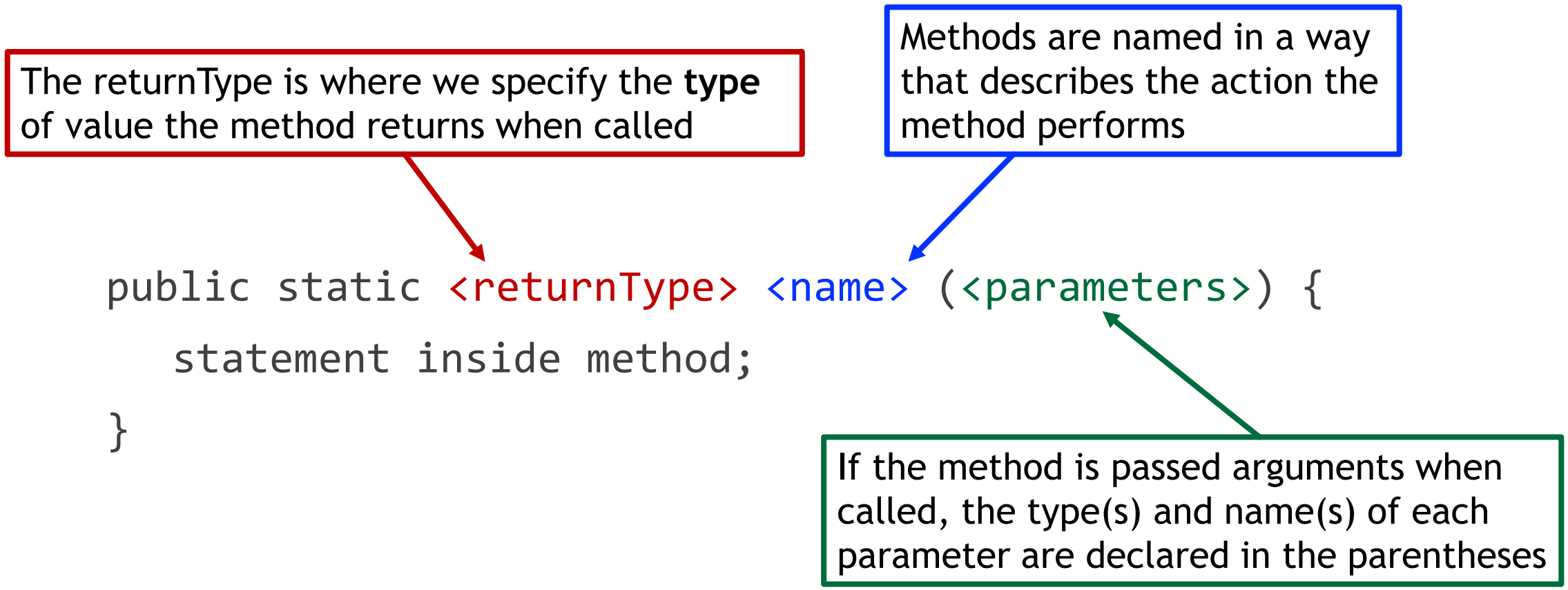
- Examples:

```
String myName = "Anthony";
Student s1 = new Student("Ali", "v00123456");
```

More on this next week!

Methods

- Methods are blocks of code declared within a class that perform a single action (similar to functions in Python and C)
- General form:



Methods

- Methods are blocks of code declared within a class that perform a single action (similar to functions in Python and C)
- Specific example:

Since this method returns the sum of three numbers, the **return type** is **int**

An appropriate name for this method might be sumNumbers, as the method calculates and returns the sum of the three numbers given

```
public static int sumNumbers (int a, int b, int c) {
    return (a + b + c);
}
```

This method is called with three **int** values:

```
int result = sumNumbers(7, 2, 4);
```

if statements

- Specify a block of code to be executed if a condition is true.
- Optionally:
  - **`else if`**: specify a new condition to test, if the first condition is false
  - **`else`**: specify a block of code to be executed if all prior conditions are false
- General form:

```
if (condition1) {
    …
}
```

if statements

- Specify a block of code to be executed if a condition is true.
- Optionally:
  - **`else if`**: specify a new condition to test, if the first condition is false
  - **`else`**: specify a block of code to be executed if all prior conditions are false
- General form:

```
if (condition1) {
    …
} else if (condition2) {
    …
} else if (condition3) {
    …
}
```

if statements

- Specify a block of code to be executed if a condition is true.
- Optionally:
  - **`else if`**: specify a new condition to test, if the first condition is false
  - **`else`**: specify a block of code to be executed if all prior conditions are false
- General form:

```
if (condition1) {
    …
} else if (condition2) {
    …
} else if (condition3) {
    …
} else {

}
```

if statements

- Specify a block of code to be executed if a condition is true.
- Optionally:
  - **`else if`**: specify a new condition to test, if the first condition is false
  - **`else`**: specify a block of code to be executed if all prior conditions are false
- General form:

```
if (condition1) {
    …
} else {
    …
}
```

if statements

- Specify a block of code to be executed if a condition is true.
- Optionally:
  - **`else if`**: specify a new condition to test, if the first condition is false
  - **`else`**: specify a block of code to be executed if all prior conditions are false

- Always starts with an `if`…
  - followed by as many additional `else if`s as are required
  - followed by a single `else`, if necessary

Operators for conditions

Operator	Meaning	Example	Result
>	greater than	x > y	True if x is greater than y, False otherwise
<	less than	x < y	True is x is less than y, False otherwise
>=	greater than or equal to	x >= y	True if x is greater than or equal to y, False otherwise
<=	less than or equal to	x <= y	True if x is less than or equal to y, False otherwise
==	equal to	x == y	True if x is equal to y, False otherwise
!=	not equal to	x != y	True if x is not equal to y, False otherwise
&&	and	x < y && x < z	True if x is less than y AND x is less than z
||	or	x < y || x == z	True if x is less than y OR if x is equal to z

if statements

- Specific example:

```
if (grade >= 90 && grade <= 100) {
      System.out.println("Excellent!");
} else if (grade > 75) {
      System.out.println("Great work");
} else if (grade > 60) {
      System.out.println("Good job");
} else if (grade >= 50){
      System.out.println("Okay...");
} else {
       System.out.println("Oh no");
}
```

Each condition is checked, one at a time, until a condition evaluates to true.

Only one code block is ever executed (the one for which the condition was true)

If no conditions evaluate to true, then the else code block is executed

```
grade = 66;
```

switch statements

- `switch` statements provide the same functionality as nested `else if` statements, but with different syntax:

```
if (childAge == 0) {
        label = "Infant";
} else if (childAge == 1) {
        label = "Baby";
} else if (childAge == 2) {
        label = "Toddler";
} else if (childAge == 3 || childAge == 4) {
        label = "Child";
} else {
        label = "Grown-up";
}
```

```
switch(childAge) {
        case 0:
                label = "Infant";
                break;
        case 1:
                label = "Baby";
                break;
        case 2:
                label = "Toddler";
                break;
        case 3: case 4:
                label = "Child";
                break;
        default:
                label = "Grown-up";
}
```

Repetition structures

- Repetition structures, or loops, allow us to repeat a block of code
- There are three types of loops we'll use in Java:
  - `while`
    - continue to execute a block of code as long as a specified condition is met
    - useful because sometimes we don't know how long a condition will be true for
  - `do-while`
    - ***always*** execute the block of code once
    - and continue to execute the block of code as long as a specified condition is met
  - `for`
    - repeat the execution of a block of code a specified number of times

The `while` loop

- General form: (syntax is very similar to a if-statement)

```
while(condition) {
   statement;
}
```

continue to execute a block of code as long as a specified condition is met

- Specific example:

```
int numLives = 3;
while(numLives > 0) {
   numLives = playLevel(numLives); //assume this is game level
}
System.out.println("Game over!");
```

The do while loop

- General form:

```
do {
   statement;
} while(condition);
```

always execute the block of code once and continue to execute the block of code as long as a specified condition is met

- Specific example:

```
int entry = 0
do {
   entry = getUserEntry(); //assume this method gets user input
   System.out.println("You entered " + entry);
} while (entry > 0 && entry < 10);
```

The for loop

repeat the execution of a block of code a specified number of times

- General form:

```
for (initialization; condition; increment) {
   statement;
}
```

- Let's look at the three parts inside the for loop parentheses:
  - initialize: - this happens once (usually is simply initializing a variable)
    - example: `int x = 5;`
  - condition: - determines whether or not to execute the block of code
    - example: `x < 10` (similar to the condition in an if-statement)
  - increment: - this code runs every time after the code block is executed
    - example: `x++;`

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("i is " + i);
}
```

Memory:

i: 1

- Code trace:
  - initialize step happens once

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("i is " + i);
}
```

Memory:

i: 1

- Code trace:
  - evaluate condition each time to determine whether or not to execute the code in the for loop body

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
    System.out.println("i is " + i);
}
```

Memory:

i: 1

Output:

```
i is 1
```

- Code trace:
  - evaluate condition each time to determine whether or not to execute the code in the for loop body

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("i is " + i);
}
```

- Code trace:
  - increment step happens every time after the code block has completed execution

Memory:

i: 2

Output:

```
i is 1
```

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("i is " + i);
}
```

- Code trace:
  - evaluate condition each time to determine whether or not to execute the code in the for loop body

Memory:

i: 2

Output:

```
i is 1
```

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
    System.out.println("i is " + i);
}
```

Memory:

i: 2

- Code trace:
  - evaluate condition each time to determine whether or not to execute the code in the for loop body

Output:

```
i is 1
i is 2
```

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("i is " + i);
}
```

- Code trace:
  - evaluate condition each time to determine whether or not to execute the code in the for loop body

Memory:

i: 5

Output:

```
i is 1
i is 2
i is 3
i is 4
i is 5
```

The for loop

repeat the execution of a block of code a specified number of times

- Specific example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("i is " + i);
}
System.out.println("done");
```

- Code trace:
  - after the code block is executed with `i = 5`, the incrementation step will set `i` to 6
  - break out of the for loop and executed whatever code follows it

Memory:

i: 6

Output:

```
i is 1
i is 2
i is 3
i is 4
i is 5
done
```

Arrays

- Arrays are used to store multiple values in a single variable
- They allow us to work with a collection of items

Array Declaration

- General form:

```
type[] name = {<comma separated values};
```

If we know what values we want to put in our array, we list them (separated by commas) when we declare the array

```
type[] name = new type[<size>];
```

Otherwise, we can just declare a new array and specify the length of the array (how many elements it can hold)

Array Declaration

- Specific examples:

create an array to store integers named intArray with initial values 6, 1, 2, and 5

```
int[] intArray = {6, 1, 2, 5};
double[] nums = {8.93, 3.14};
```

create an array to store integers named array with space to hold 5 integers (all locations are initialized to 0 for now)

```
int[] array = new int[5];
double[] grades = new double[200];
```

Working with Arrays

- Square brackets [] are used to specify which index within the collection of items is to be accessed.
  - The first index in an array is index 0

```
int[] numsArray = {7, 9, 6, 4};
int x = numsArray[0];                    //sets the x variable to 7
System.out.println(numsArray[2]);   //outputs 6

numsArray[3] += numsArray[0];
//adds the value at numsArray[0] onto the value at index 3
//numsArray[3] is originally 4, and is set to 4+7 = 11
//the array now has the following contents: {7, 9, 6, 11}
```

Working with Arrays

- We can use the **`length`** property to determine how many elements an array has:

```
int[] numsArray = {7, 9, 6, 4};
System.out.println(numsArray.length);    //outputs 4
```

- When used in combination with a for loop we can easily visit each element in an array:

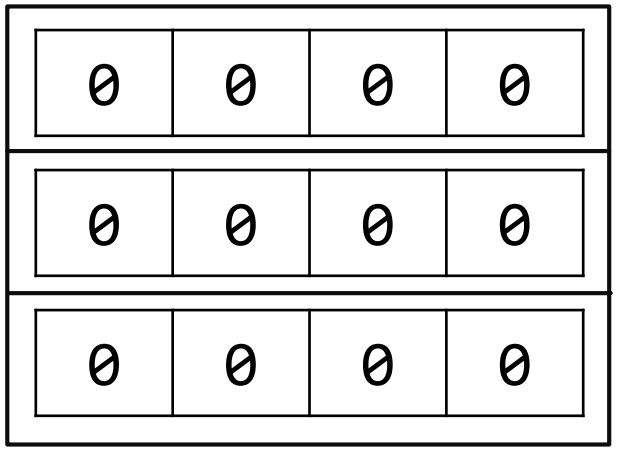
```
for (int i = 0; i < numsArray.length; i++) {
   System.out.println(numsArray[i]);
}
```

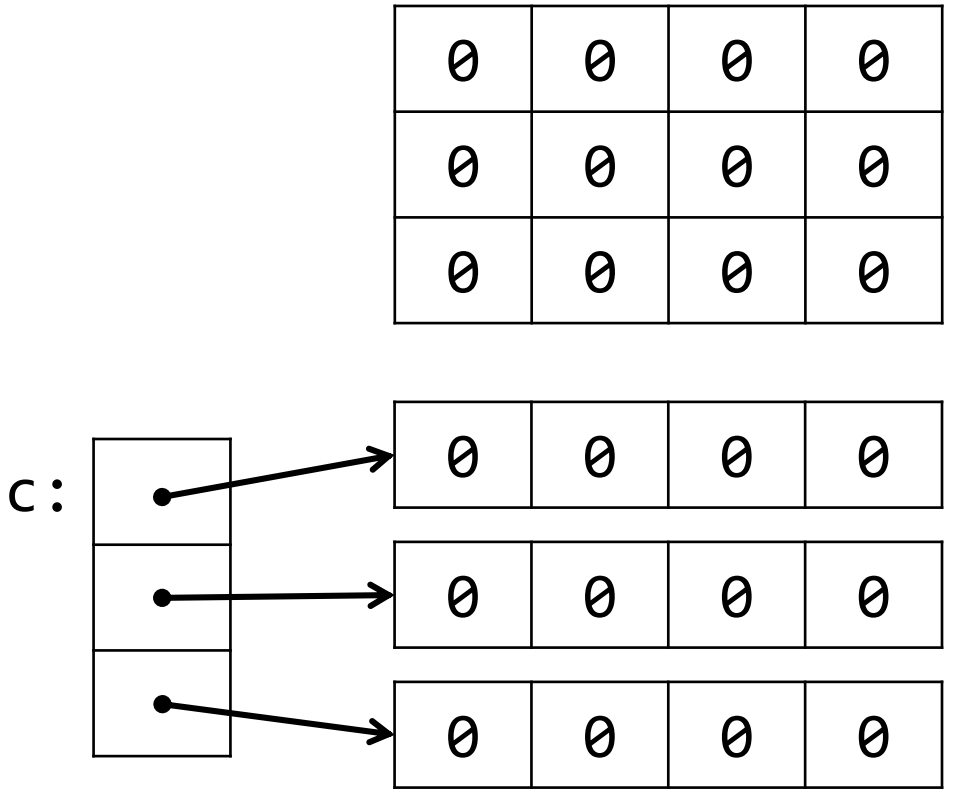
```
//output:
// 7
// 9
// 6
// 4
```

Multi-dimensional Arrays

- Later in the course, we will see that we can store different types of data inside each index in an array
- We can also store an array inside each index of an array, for example:

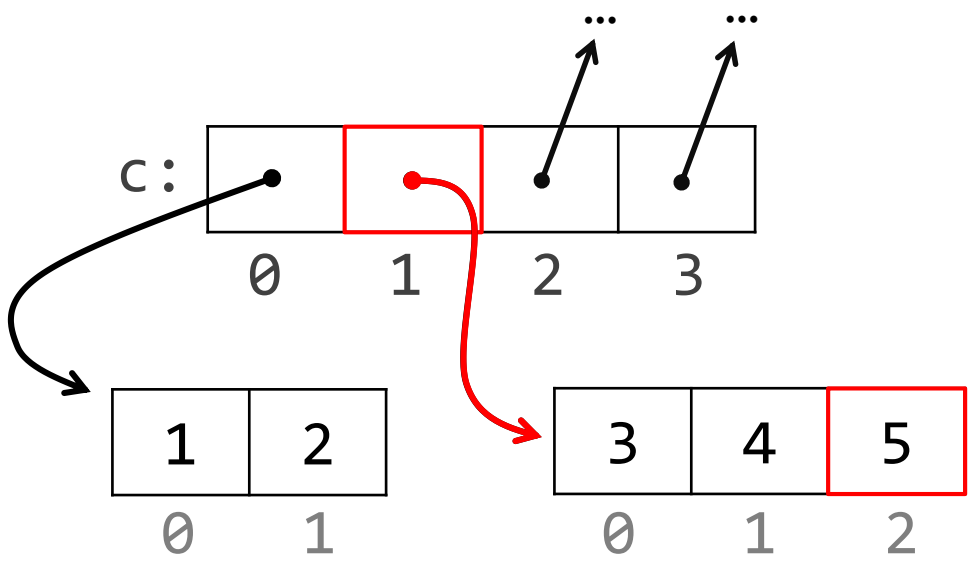
```
int c[][] = new int[3][4];
```

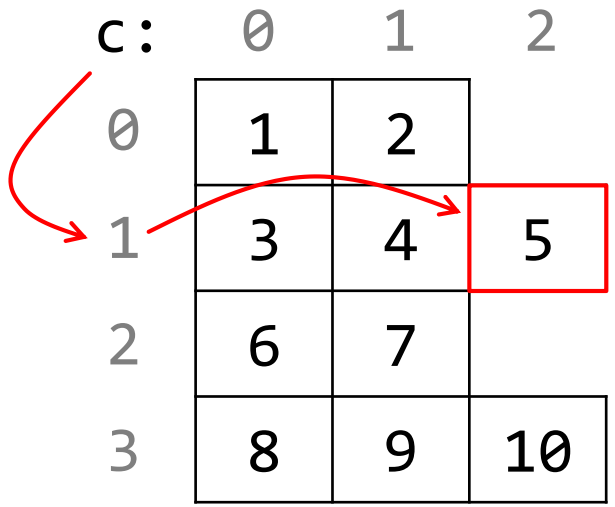




Multi-dimensional Arrays

```
int c[][] = {{1, 2}, {3, 4, 5}, {6, 7}, {8, 9, 10}};
```






```
System.out.println(c[1][2]); // Outputs 5
```

Errors and Debugging

- There are three kinds of errors we will see when writing code in Java:
  - Syntax errors
  - Runtime errors
  - Logic errors

Syntax errors

- When we compile our code, syntax errors are reported by the compiler
- If a syntax error occurs, the compiler does not create an new executable, so we will not be able to run our programs until we have fixed all syntax errors in our code

```
1 public class ErrorsExample {
2     public static void main(String[] args) {
3         int myNum = 5
4         System.out.println(myNum);
5     }
6 }
```

```
C:\Windows\System32\cmd.exe
lectures\01-java>javac ErrorsExample.java
ErrorsExample.java:3: error: ';' expected
                int myNum = 5
                            ^
1 error
```

Syntax errors

- When we compile our code, syntax errors are reported by the compiler
- If a syntax error occurs, the compiler does not create an new executable, so we will not be able to run our programs until we have fixed all syntax errors in our code

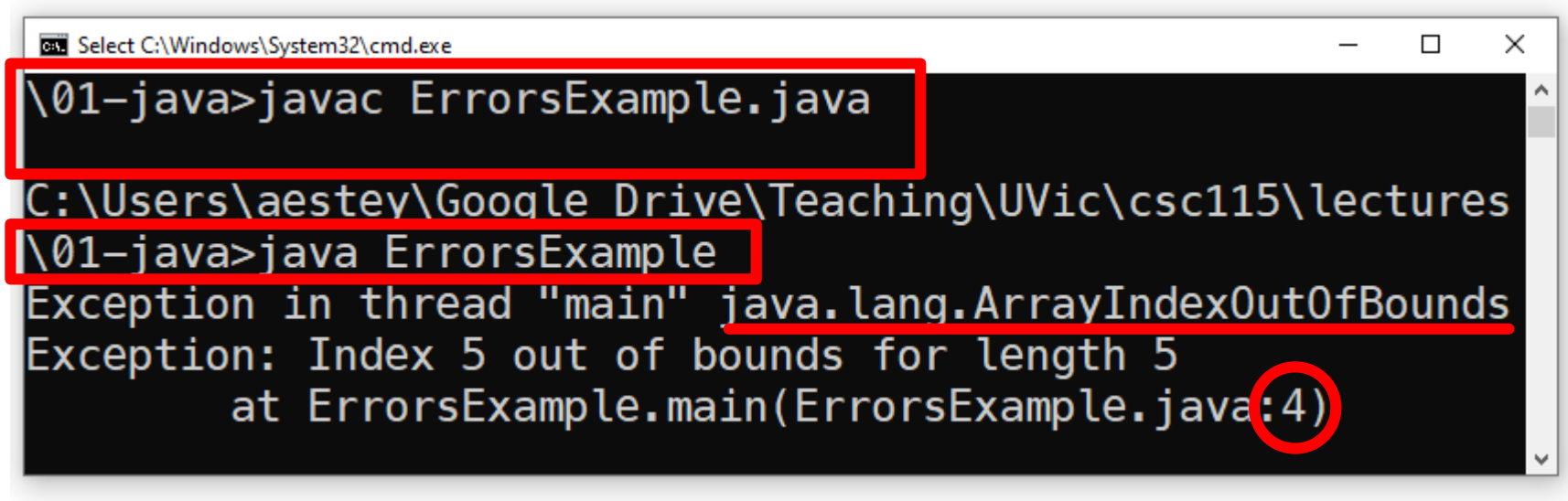
```
1  public class ErrorsExample {
2      public static void main(String[] args) {
3          int myNum = 5;
4          System.out.println(myNum);
5
6          mynum += 3;
7          System.out.println(myNum);
8      }
9  }
```

```
C:\Windows\System32\cmd.exe
\lectures\01-java>javac ErrorsExample.java
ErrorsExample.java:6: error: cannot find symbol
                mynum += 3;
                ^
  symbol:   variable mynum
  location: class ErrorsExample
```

Runtime errors

- Sometimes errors occur while our program is running; these are called runtime errors
- There is nothing wrong with the syntax, but the way we are trying to use a variable or object is incorrect

```
1  public class ErrorsExample {
2      public static void main(String[] args) {
3          int[] arr = new int[5];
4          arr[5] = 3;
5      }
6  }
```



Logic errors

- Logic errors occur when a programs successfully compiles and executes, but the result is incorrect (it does the wrong thing)
- Typically we write tests to detect logic errors. After running the tests, logic errors can be identified and fixed based on the failing tests