Revised: 4/21/2020

Objects and Classes II

Copying Objects, Arrays of Objects, and Objects of Arrays

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- String Representation
- Object Equality
- Copying Instances
- Static Fields and Methods
- An Array of Objects
- An Object of Arrays
- Arrays and Methods
  - Arrays as Method Arguments
  - Methods that return Arrays
  - Variable-Length Arguments

Colors/Fonts

- Local Variable Names – Brown
- Primitive data types – Fuchsia
- Literals – Blue
- Keywords – Orange
- Object names – Green
- Operators/Punctuation – Black
- Field Names – Lt Blue
- Method Names – Purple
- Parameter Names – Gold
- Comments – Gray
- Package Names – Pink

Source Code – **Consolas**
Output – Courier New

String Represenation

- An object's **toString()** method normally returns a String that describes the current state of an object.
  - It may include some or all of the current values of the object's fields.
  - Can be useful for debugging

- All objects have a toString() method, even if the method is not defined in the class.

An object **without** a toString() method defined

```
public class Circle {

    private int radius;
    private double area;
    private double circumference;

    public Circle(int radiusIn) {
        radius = radiusIn;
        area = Math.pow(Math.PI * radius, 2);
        circumference = 2 * Math.PI * radius;
    }

}
```

```
public class CircleTest {

  public static void main(String[] args) {
    Circle example = new Circle(10);
    String output = example.toString();
    System.out.println(output);
  }

}
```

```
package.Circle@659e0bfd
```

Name of the package

Name of the object

Memory address

An object **with** a toString() method defined

```
public class Circle {

    private int radius;
    private double area;
    private double circumference;

    public Circle(int radiusIn) {
        radius = radiusIn;
        area = Math.pow(Math.PI * radius, 2);
        circumference = 2 * Math.PI * radius;
    }

  public String toString() {
    return "Radius: " + radius +
           "\nArea: " + area +
           "\nCircumference: " + circumference;
  }

}
```

```
public class CircleTest {

  public static void main(String[] args) {
    Circle example = new Circle(10);
    String output = example.toString();
    System.out.println(output);
  }

}
```

```
Radius: 10
Area: 986.96…
Circumference: 62.83…
```

Creating a toString() method

- When an object has a toString() method, it is implicitly called when concatenating.

```
Circle example = new Circle(10);

String output = "Circle Information:\n" + example;

System.out.println(output);
```

```
Circle Information:
Radius: 10
Area: 986.96…
Circumference: 62.83…
```

Creating a toString() method

- It will also be implicitly called when passed to System.out.println (or print, or printf)

```
Circle example = new Circle(10);

System.out.println(example);
```

```
Radius: 10
Area: 986.96…
Circumference: 62.83…
```

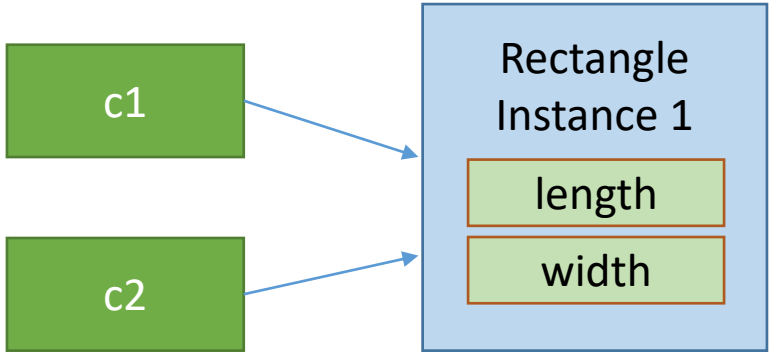
Shallow Copy vs Deep Copy

- There are two ways to create a copy of an instance.

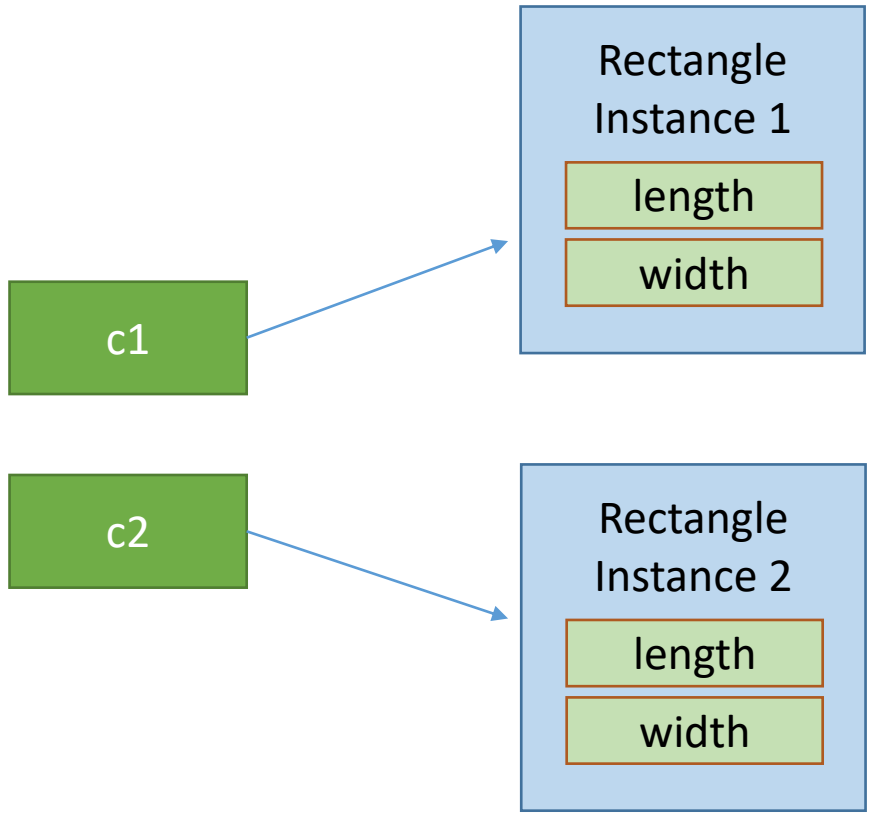
- Deep Copy: The ***data*** referenced by one variable is copied to a new location in memory, and is then referenced by a different variable.

- Shallow Copy: The ***reference*** to data at a location in memory is copied from one variable to a different variable. In essence, both variables reference the same data/object in memory, NOT their own.

Copying Instances



Shallow Copy



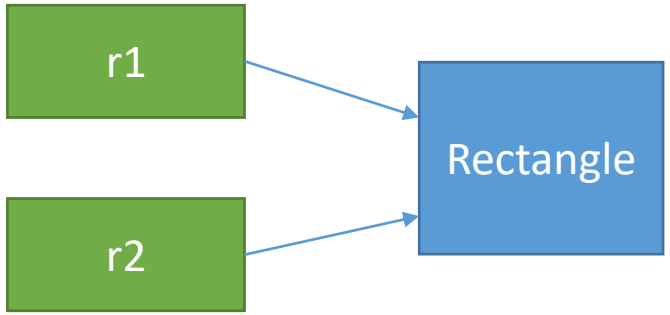
Deep Copy

Shallow Copying Instances

- To shallow copy an instance, simply use the assignment operator =
- Remember, the shallow copy is not a new instance.
  - The new variable will point to the same instance in memory.

```
public static void main(String[] args) {
  Rectangle r1 = new Rectangle(8, 9);
  Rectangle r2 = r1;
  r2.setLength(10);
  System.out.print("r1's length is ");
  System.out.println(r1.getLength());
}
```

```
r1's length is 10
```



Deep Copying Instances

- A deep copy gives us an entirely new instance with the current state of the instance we wish to copy.
  - All fields of the new instance should have the same values as the original instance.

- There are a number of techniques to deep copy instances, but we will look at two:
  - A method that returns a new instance with all of the new instance's fields set to the same values as the original instance.
  - A copy constructor.

A simple clone method

- This method in the Rectangle class would return a new instance of a Rectangle object, using this instance's data/fields.

```
public Rectangle clone() {
   return new Rectangle(length, width);
}
```

Deep Copying Instances

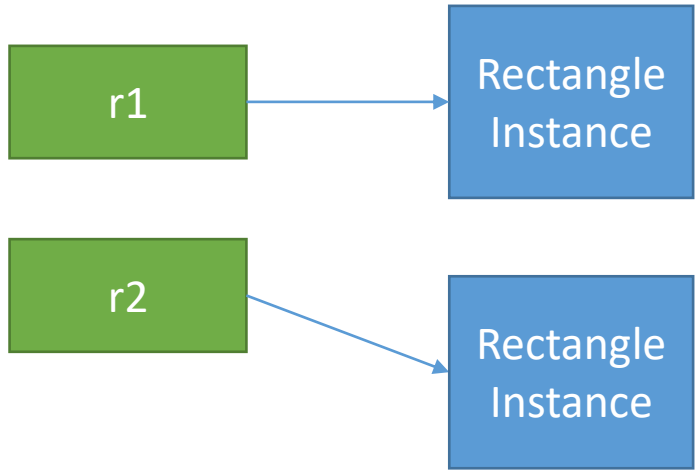
```
public static void main(String[] args) {

  Rectangle r1 = new Rectangle(11, 5);
  Rectangle r2 = r1.clone();
  r1.setLength(20);
  System.out.print("r1's length = ");
  System.out.println(r1.getLength());
  System.out.print("r2's length = ");
  System.out.println(r2.getLength());

}
```

```
r1's length = 11
r2's length = 20
```

The values are different because r1 and r2 are their own instances, not shallow copies.




```
public Rectangle clone() {
    return new Rectangle(length, width);
}
```

Copy Constructor

- A ***copy constructor*** is a constructor that takes an object of its own type as its argument.
  - It uses the data of that object to set its own fields.

```
public Rectangle(Rectangle rectangleIn) {
  length = rectangleIn.getLength();
  width = rectangleIn.getWidth();
}
```

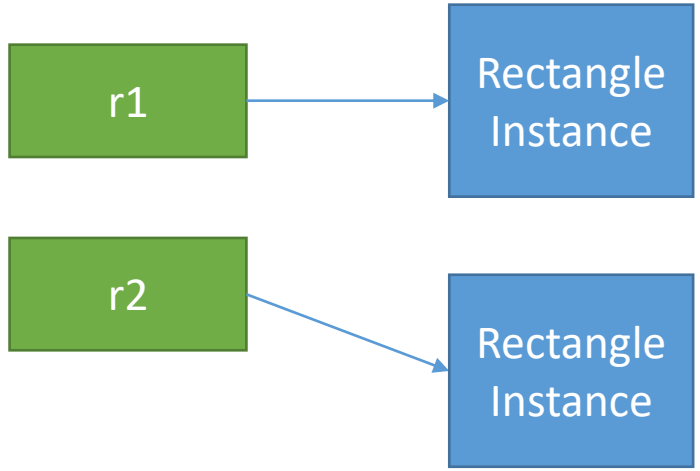
Copy Constructor

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

  Rectangle r1 = new Rectangle(5, 4);
  Rectangle r2 = new Rectangle(r1);
  r2.setLength(20);
  System.out.print("r1's length = ");
  System.out.println(r1.getLength());
  System.out.print("r2's length = ");
  System.out.println(r2.getLength());

}
```

```
r1's length = 11
r2's length = 20
```




```
public Rectangle(Rectangle rectangleIn) {
  length = rectangleIn.getLength();
  width = rectangleIn.getWidth();
}
```

Equality of Instances

- What does it mean for two instances of an object to be "equal" to each other?
  - Do all of the fields in the two instances need to have the same values? Maybe only some fields?

- Two ways to test equality of instances:
  - If two different variables reference the same instance (ie. they are shallow copies)
  - If two different instances, referenced by two different variables, contain the same data (or however you define “equal”)

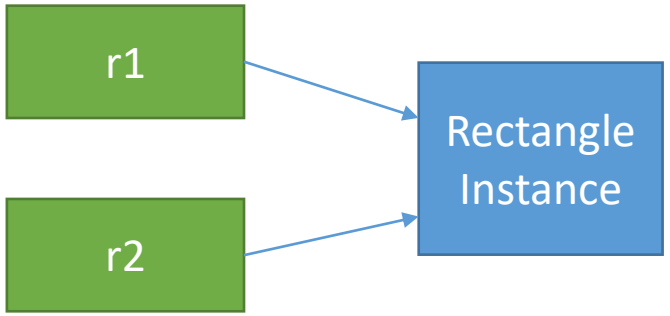
Testing the Equality of Instances

- Using the equality operator == will only tell us if the two variables being compared reference the same instance.

```
Rectangle r1 = new Rectangle(8, 9);
Rectangle r2 = r1;
```

true

```
if(r1 == r2) {
    System.out.println("r1 shares the same reference as r2");
}
```



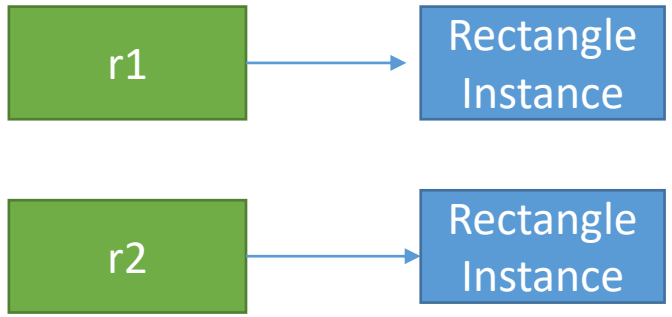
Testing the Equality of Instances

- Even though r1 and r2's instances have the same dimensions below, that is not what the equality operator checks for.
  - Since r1 and r2 have different references, the equality operator returns false.

```
Rectangle r1 = new Rectangle(8, 9);
Rectangle r2 = new Rectangle(8, 9);
```

false

```
if(r1 == r2) {
    System.out.println("r1 shares the same reference as r2");
}
```



Testing the Equality of Instances

- Every object is different, so there can be no one-size-fits-all solution.
- To determine if two instances of the same type are "equal", you will need to decide what makes two objects equal and create a method to compare them.

Writing an equals method

- As an example, we could add the method below to the Rectangle class from earlier in the lecture.
  - We would also now need (at least) getter methods for the length and width fields.
- This equals method compares the fields of the parameter Rectangle object to this Rectangle object's fields.

```
public boolean equals(Rectangle otherRectangle) {
  if(otherRectangle.getLength() == length &&
     otherRectangle.getWidth() == width) {
    return true;
  }
  return false;
}
```

Writing an equals method

```
Rectangle r1 = new Rectangle(9, 12);
Rectangle r2 = new Rectangle(9, 12);
```

true

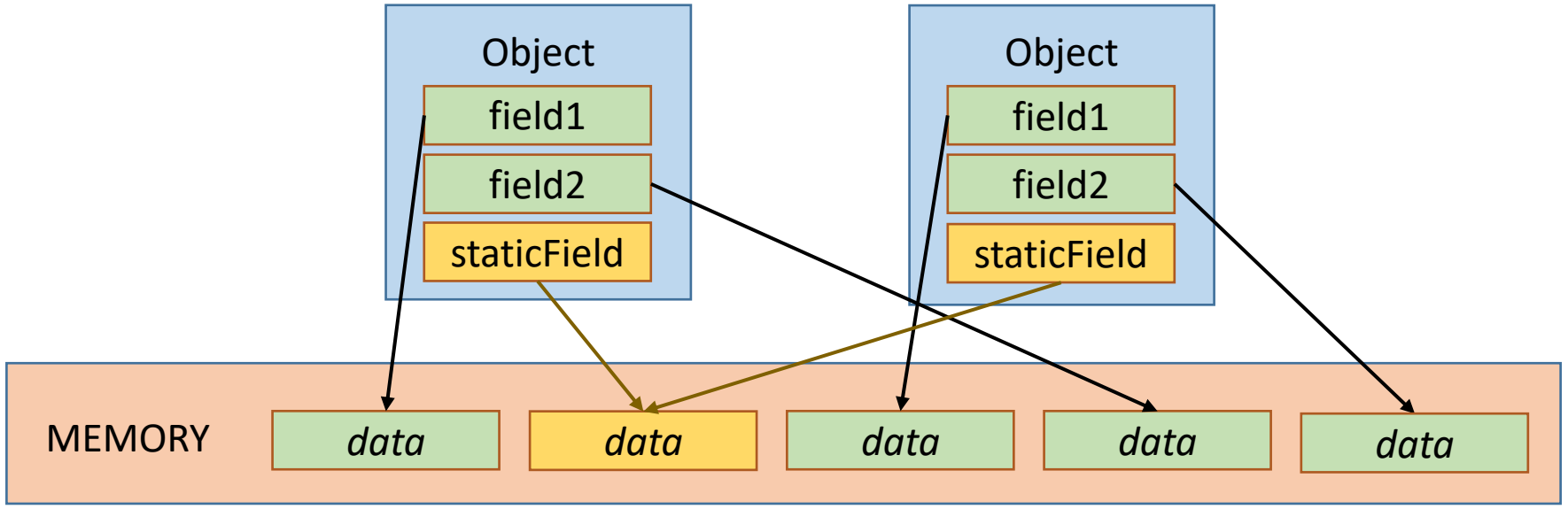
```
if(r1.equals(r2)) {
  System.out.println("r1 and r2 have the same dimensions");
}
else {
  System.out.println("r1 and r2 do not have the same dimensions");
}
```

```
r1 and r2 have the same dimensions
```

```
public boolean equals(Rectangle otherRectangle) {
  if(otherRectangle.getLength() == length &&
     otherRectangle.getWidth() == width) {
    return true;
  }
  return false;
}
```

Static Fields

- A ***static field*** (also called a ***class field***) is a field whose reference is shared across **all** instances of the object.
  - Unlike instance fields, which have unique references.



Declaring a Static Field Variable

- Place the **`static`** keyword before the field's data type.

***MODIFIERS* `static` *TYPE VARIABLE*;**

Example: `private static String myStaticField;`

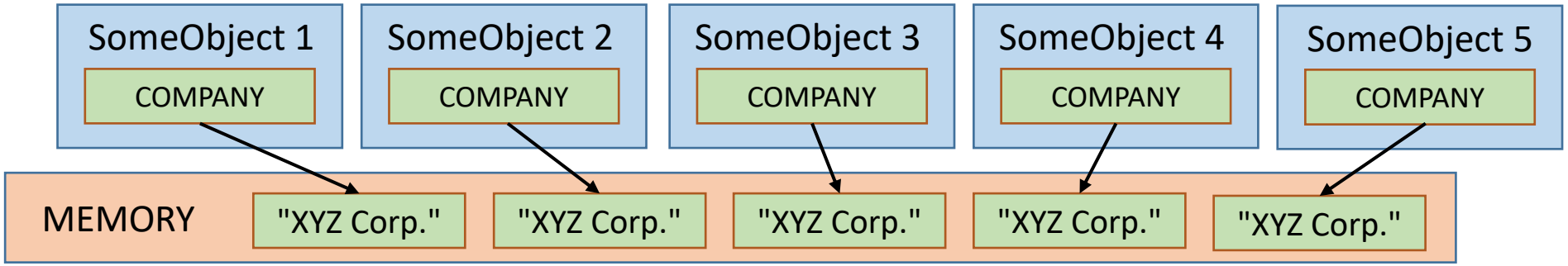
Static Fields

- The most common use of a static field is for any fields that are constant.
  - Imagine a class with a constant instance field:

```
public final String COMPANY = "XYZ Corp.";

public SomeObject() {
    ...
}
```

- Every time we instantiate this object, space is allotted for each object's COMPANY field.

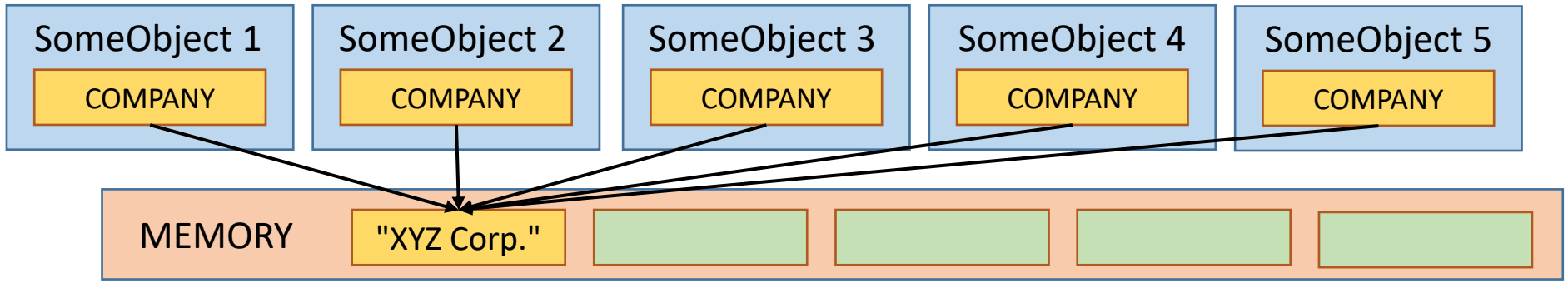


Static Fields

- By making the constant a static field, we can save space:

```
public final static String COMPANY = "XYZ Corp.";

public SomeObject() {
    ...
}
```



Static Fields

- Another common use of static fields is to count how many instances of an object has been created:

```
private int length;
private int width;
private static int numInstances = 0;

public Rectangle(int lengthIn, int widthIn) {
    length = lengthIn;
    width = widthIn;
    numInstances += 1;
}
```

Every time a new Rectangle object is instantiated, the constructor increases the value of numInstances by 1.

Static Fields

```
private int length;
private int width;
private static int numInstances = 0;

public Rectangle(int lengthIn, int widthIn) {
    length = lengthIn;
    width = widthIn;
    numInstances += 1;
}

public int getNumberOfInstances() {
    return numInstances;
}
```

Since the numInstances variable is static, the same value will be referenced for any instance of a Rectangle object.

Static Fields

```
public static void main(String[] args) {
    Rectangle r1 = new Rectangle(8, 9);
    System.out.println("Total instances = " + r1.getNumberOfInstances());
    Rectangle r2 = new Rectangle(10, 3);
    System.out.println("Total instances = " + r1.getNumberOfInstances());
}
```

```
Total instances = 1
Total instances = 2
```

- Every println statement uses "r1"
  - Notice we haven't done anything to r1 besides call a getter method.
- Every constructor call incremented the value of the numInstances field, which shares the same reference across all instances.

Using Static Fields Within a Class

- Static variables have class scope.
  - They can be used by all methods and constructors in the class, just like any instance variable.

```
private double nonStaticExample;
private static int staticExample;

public SomeObject() {
    staticExample += 1;
    nonStaticExample += 2.5;
    ...
}

public double getSumOfValues() {
    return staticExample + nonStaticExample;
}
```

Using Static Fields Within a Class

- However, local variables cannot be static.

```
public void exampleMethod() {
  static int value;
  ...
}
```

Static Methods

- A ***static method*** is a method that can be called without having an instance of the object.
  - When you get the square root of a number using the Math class, notice how you don't have to instantiate a Math object to do so. The sqrt method, like all methods in the Math class, are static.

```
Math.sqrt(16.0);
```

  - IMPORTANT: Since static methods can be called without an instance of the object, the body of a static method cannot use its object's instance fields.

Declaring a Static Method

- Place the **`static`** keyword before the method's return type.

```
MODIFIERS static TYPE NAME(PARAMETERS) {
```

Example:

```
public static String myStaticMethod() {
```

Static Methods

- The add method in the Calculate class below can be called with or without an instance of the object.

```
public class Calculate {

  public static int add(int operand1, int operand2) {
    return operand1 + operand2;
  }

}
```

```
public static void main(String[] args) {
  Calculate calc = new Calculate();
  System.out.print("The sum of 5 and 6 is ");
  System.out.println(calc.add(5, 6));
}
```

```
public static void main(String[] args) {
  System.out.print("The sum of 5 and 6 is ");
  System.out.println(Calculate.add(5, 6));
}
```

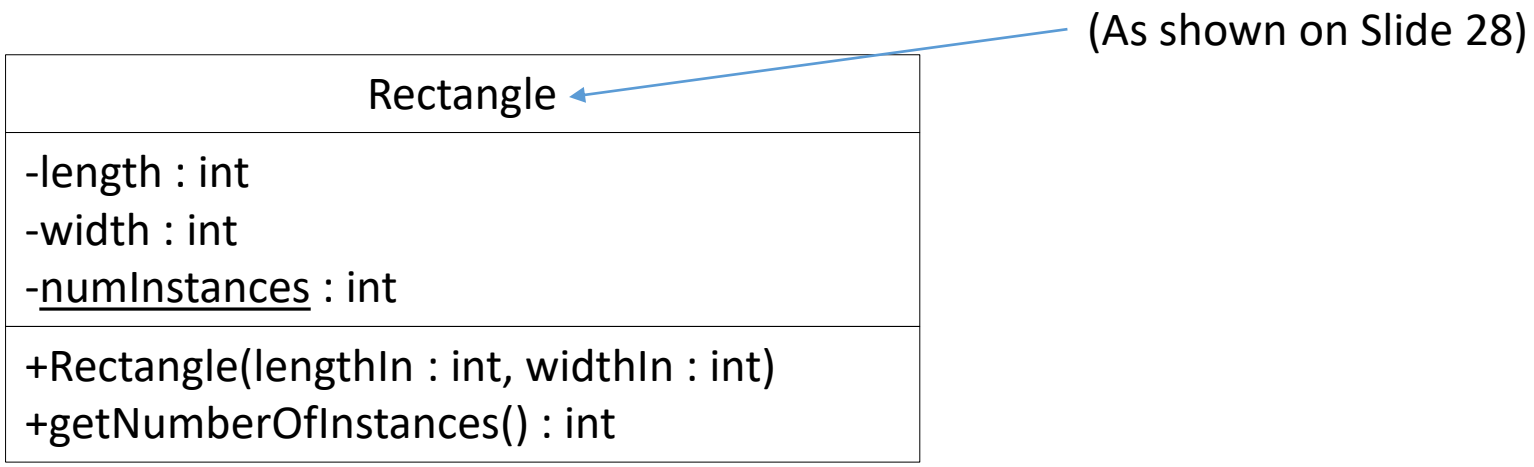
Static Methods vs Non-Static Methods

- Static methods
  - **Can** contain local variables of any data or object type.
  - **Can** use the static fields of its class.
  - **Cannot** use the instance fields of its class.
  - **Can** call any static methods in the same class.
  - **Cannot** call any non-static methods in the same class.
    - As non-static methods may rely on using instance fields.
  - **Can** be called without an instance of the class.

- Non-static methods
  - **Can** contain local variables of any data or object type.
  - **Can** use any (static or non-static) fields of its class.
  - **Can** call any method (static or non-static) in the same class.
  - **Cannot** be called without an instance of the class.

Static Members in UML

- In UML Class Diagrams, static members are underlined.
  - Only the name; not type, return type, or parameters.



Array of Objects

- Arrays can contain references to objects.

- The statements below create an array of three Car objects.

```
Car[] myCars = new Car[3];

myCars[0] = new Car("Jeep", "Cherokee", 1994);
myCars[1] = new Car("Ford", "F-150", 2001);
myCars[2] = new Car("Subaru", "Outback", 2000);
```

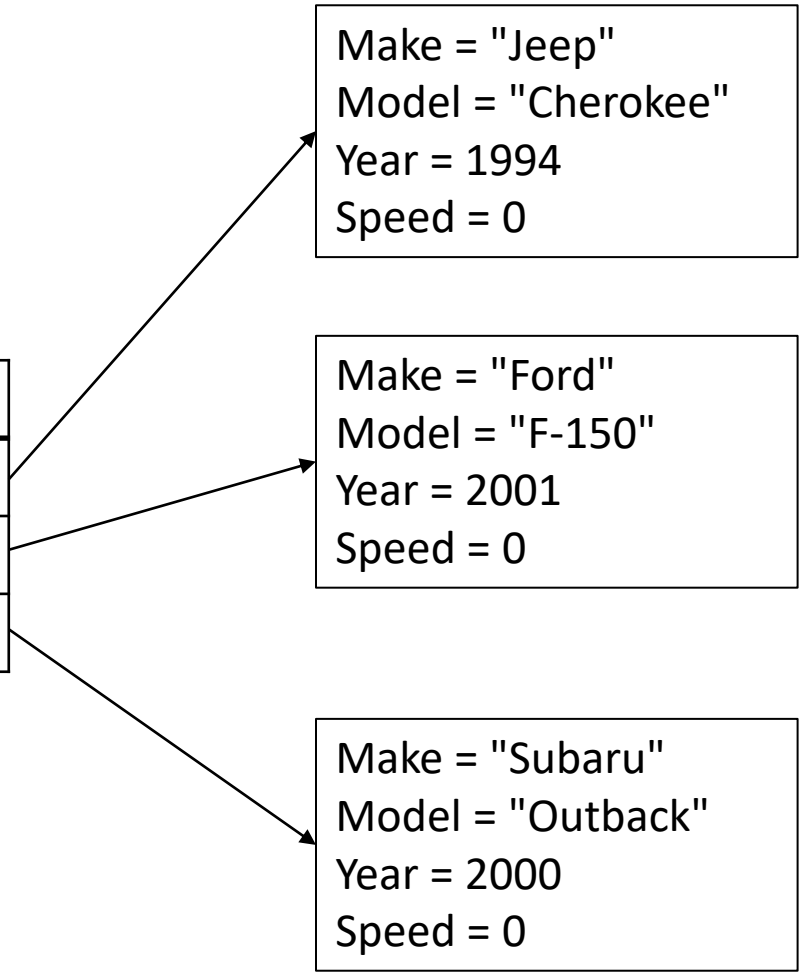
Array of Objects

Symbol Table

Symbol	Address
`myCars`	1000

Memory Map

Address	Data
1000	Object
1A00	Object
1B00	Object



(The memory addresses shown are hypothetical/for illustration purposes.)

Array of Objects

```
Car[] myCars = new Car[3];

myCars[0] = new Car("Jeep", "Cherokee", 1994);
myCars[1] = new Car("Ford", "F-150", 2001);
myCars[2] = new Car("Subaru", "Outback", 2000);

System.out.println(myCars[1].getMake());
System.out.println(myCars[2].getYear());
System.out.println(myCars[0].getModel());
```

```
Ford
2000
Cherokee
```

Object of Arrays

- An object can have a field that is an array.
  - If private, the object will have complete control over what data is added to or retrieved from the array.

```
public class ParkingLot {

    private Car[] carLot;

    public ParkingLot(int sizeIn) {
        carLot = new Car[sizeIn];
    }

}
```

Object of Arrays

- This method would control adding Car objects to the ParkingLot object's carLot field.
  - Checks that a Car object isn't already parked in the desired space.

```
public void addCar(Car carIn, int spaceIn) {
    if(carLot[spaceIn] == null) {
        carLot[spaceIn] = carIn;
    }
}
```

Object of Arrays

- This method would control removing Car objects to the ParkingLot object's carLot field.
  - Checks that a Car object is parked in that space.

```
public Car removeCar(int spaceIn) {
    if(carLot[spaceIn] == null) {
        return null;
    }
    else {
        Car temp = carLot[spaceIn];
        carLot[spaceIn] = null;
        return temp;
    }
}
```

Object of Arrays

- This method would get the String version of a Car object in the ParkingLot object's carLot field.
  - Wouldn't remove the Car object.

```
public String getCarInfo(int spaceIn) {
    if(carLot[spaceIn] != null) {
        return carLot[spaceIn].toString();
    }
    else {
        return "No car parked in this space.";
    }
}
```

Arrays as Method Arguments

- An array can be passed to a method as an argument.

- Must match the array type specified as the parameter.

```
public int sum(int[] numbers)
```

Arrays as Method Arguments

```
public int sum(int[] numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

```
int[] threeNums = {4, 5, 6};
sum(threeNums);
```

Would return 15.

Arrays as Method Arguments

- Arrays are always passed to a method **by reference**.

- **Pass by reference**- The reference to data is passed to the method.
  - Arrays and Objects are always passed by reference in Java.

- **Pass by value**- The data is passed to the method.
  - Primitive data are always passed by value in Java.

Passing by Value

```
public void demoMethod(int number) {
    number = 0;
}
```

Changes the number parameter, not x.

```
int x = 5;
demoMethod(x);
```

Passes x's value as the argument.

Passing by Reference

```
public void demoMethod(int[] array) {
    array[1]= 0;
}
```

Changes the threeNums array.

```
int[] threeNums = {4, 5, 6};
demoMethod(threeNums);
```

Passes threeNums's reference as the argument.

Variable Length Arguments

- ***Variable Length Arguments*** (or ***varargs***) allow a method to accept an undetermined number of parameters/arguments.

```
public int sum(int... numbers)
```

- The varargs must all be of the correct type.

- The varargs will be treated as an array inside the method.
  - Varargs *are* arrays, just not declared as such.

Variable Length Arguments

```
public int sum(int... numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

`sum(4, 5, 6);`	Valid. Would return 15.
`sum(2, 3);`	Valid. Would return 5.
`sum(7, 8.5);`	Not valid.

Variable Length Arguments

```
public int sum(int... numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

```
int[] myOriginalArray = {3, 5, 7, 9};

sum(myOriginalArray);
```

You can pass an array to a vararg. The sum method would return 24 in this example.

Variable Length Arguments

- No additional parameters can follow a vararg.

```
public int doMath(int... numbers, String operationType) {
```

INVALID

- Although, there can be any number of normal parameters preceding it.

```
public int doMath(String operationType, int... numbers) {
```

VALID

Variable Length Arguments

```
public int doMath(String operationType, int... numbers) {
    int answer = 0;
    if(operationType.equals("+")) {
        for(int number : numbers) {
            answer += number;
        }
    } else if(operationType.equals("*")) {
        answer = 1;
        for(int number : numbers) {
            answer *= number;
        }
    }
    return answer;
}
```

```
doMath("+", 4, 5, 6);
doMath("*", 7, 3);
```

Valid. Would return 15.
Valid. Would return 21.

Variable Length Arguments

```
public int doMath(String operationType, int... numbers) {
    int answer = 0;
    if(operationType.equals("+")) {
        for(int number : numbers) {
            answer += number;
        }
    } else if(operationType.equals("*")) {
        answer = 1;
        for(int number : numbers) {
            answer *= number;
        }
    }
    return answer;
}
```

```
int[] threeNums = {4, 5, 6};
doMath("+", threeNums);
```

Valid. Would return 15.

Returning an Array from a Method

- An array can be returned by a method.
  - Be sure the method's return type is an array.

```
public int[] getNumbers() {
   int[] threeNums = {4, 5, 6};
   return threeNums;
}
```