Revised: 11/23/2025

Object-Oriented Programming IV

Michael C. Hackett

Associate Professor, Computer Science

COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

- Arrays of Objects
- Objects of Arrays
- Copying Objects
- Object Equality
- Static Field and Methods

Arrays of Objects

- Arrays can contain references to objects.

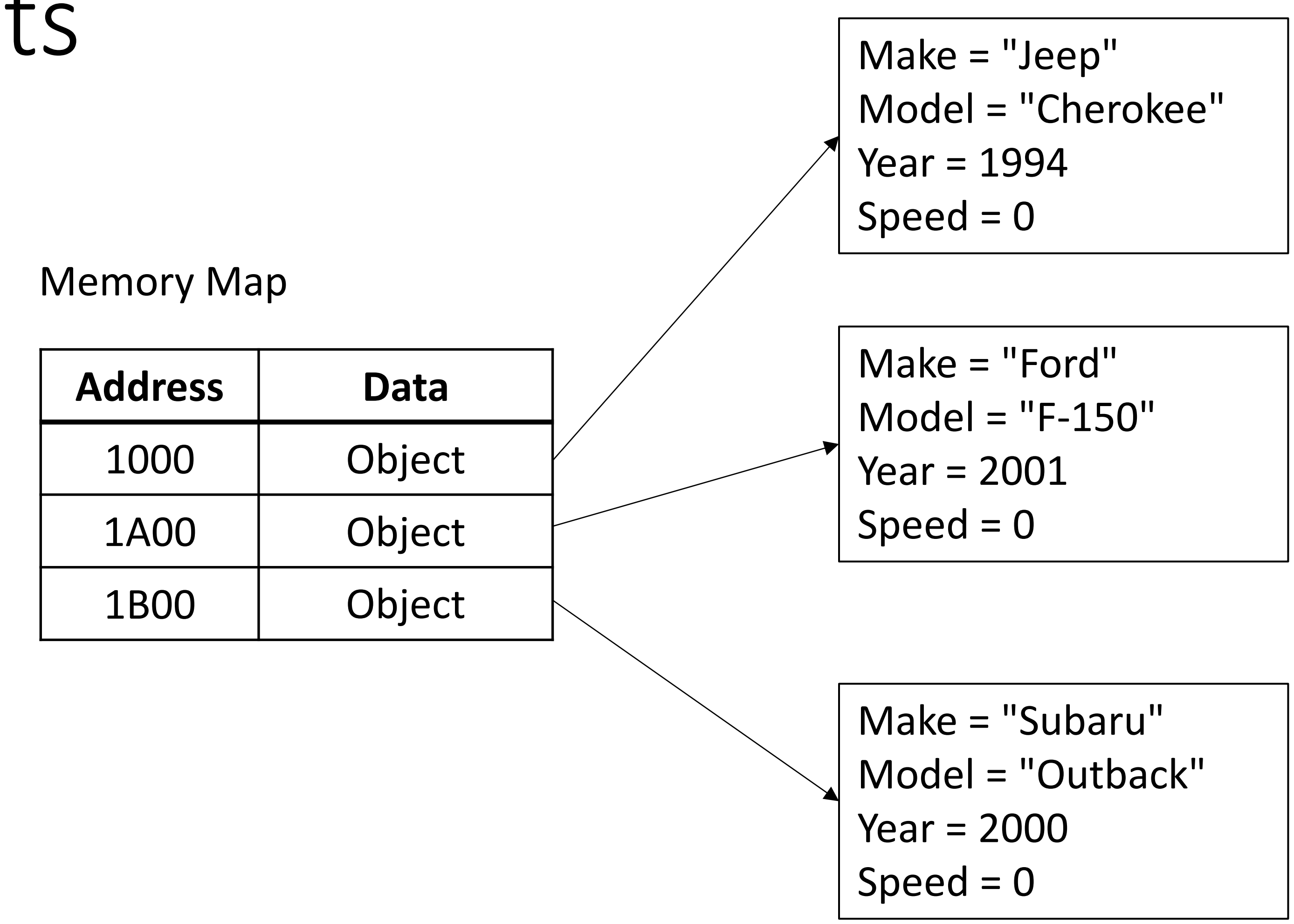
- The statements below create an array of three Car objects.

```
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);
```

Arrays of Objects

Symbol Table

Symbol	Address
`myCars`	1000



Memory Map

Address	Data
1000	Object
1A00	Object
1B00	Object

Make = "Jeep"
Model = "Cherokee"
Year = 1994
Speed = 0

Make = "Ford"
Model = "F-150"
Year = 2001
Speed = 0

Make = "Subaru"
Model = "Outback"
Year = 2000
Speed = 0

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

Arrays 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.";
    }
}
```

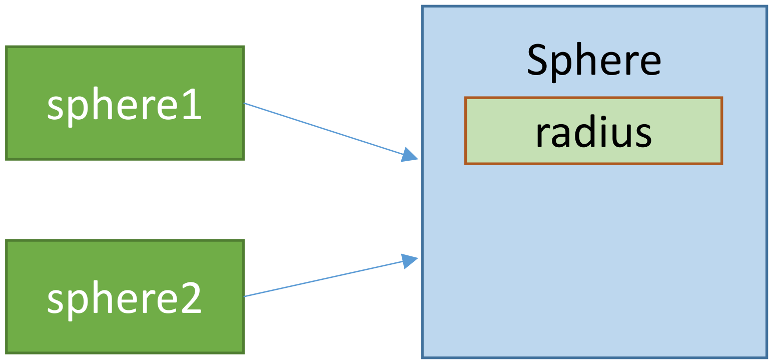
Copying Objects

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

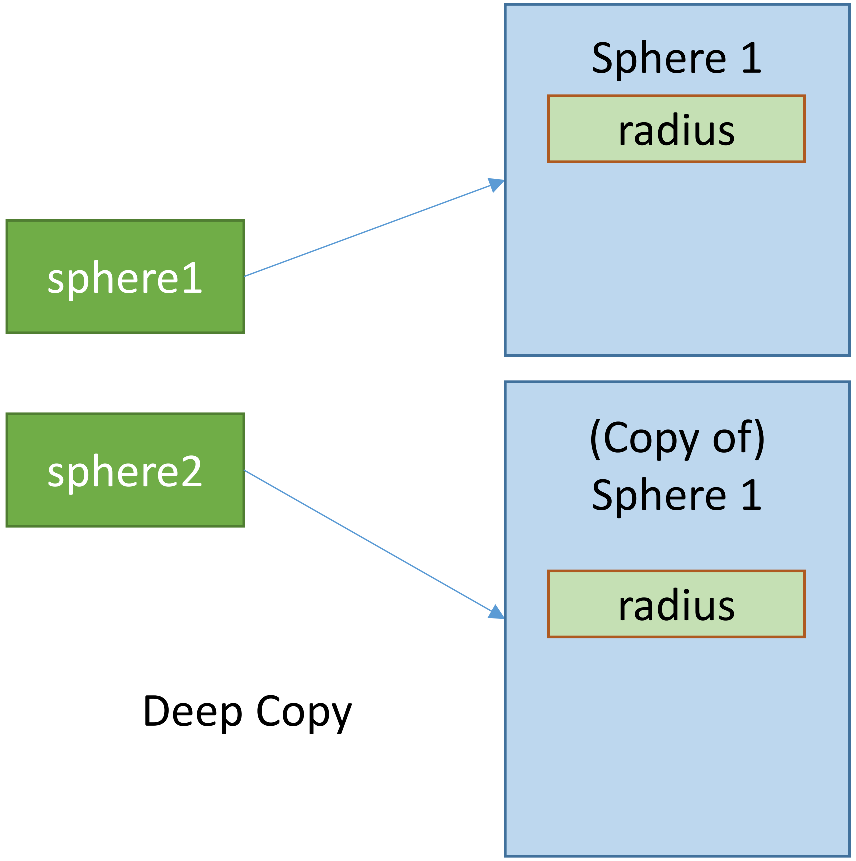
- Deep Copy: The ***data*** referenced by one variable is copied to a new location in memory, and is then referenced by a different variable.

- 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, <u>NOT their own</u>.

Copying Objects



Shallow Copy

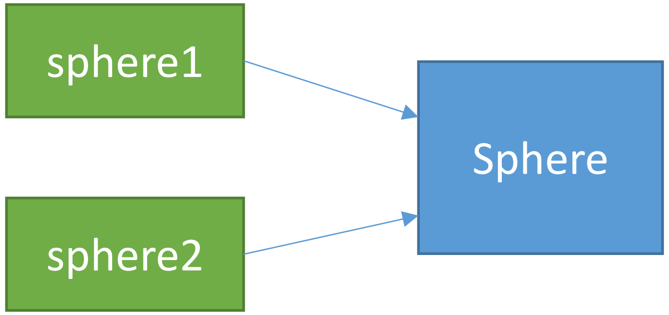


Deep Copy

Copying Objects

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

```
public static void main(String[] args) {
  Sphere sphere1 = new Sphere(5);
  Sphere sphere2 = sphere1;
  sphere2.setRadius(10);
  System.out.print("Sphere 1's radius = ");
  System.out.println(sphere1.getRadius());
}
```



```
Sphere 1’s radius = 10
```

Copying Objects

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

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

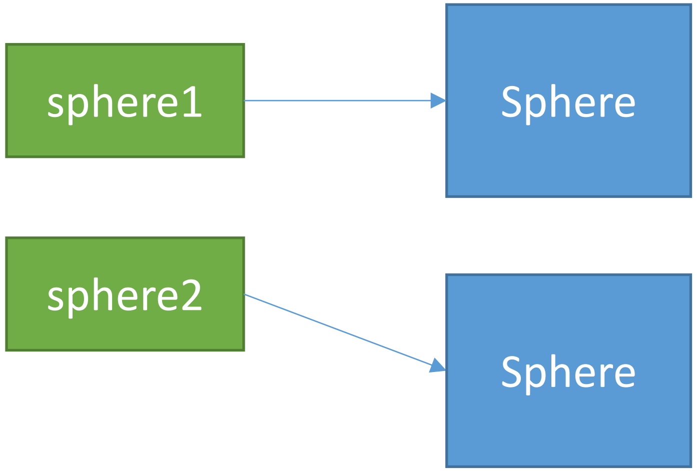
Copying Objects

- This method will return a new instance of a Sphere object, using the original instance's fields.

```
public Sphere getCopy() {
   return new Sphere(radius);
}
```

Copying Objects

```
public static void main(String[] args) {
  Sphere sphere1 = new Sphere(11);
  Sphere sphere2 = sphere1.getCopy();
  sphere2.setRadius(20);
  System.out.print("Sphere 1's radius = ");
  System.out.println(sphere1.getRadius());
  System.out.print("Sphere 2's radius = ");
  System.out.println(sphere2.getRadius());
}
```



```
Sphere 1’s radius = 11
Sphere 2’s radius = 20
```

The values are different because sphere1 and sphere2 are their own objects, not shallow copies.

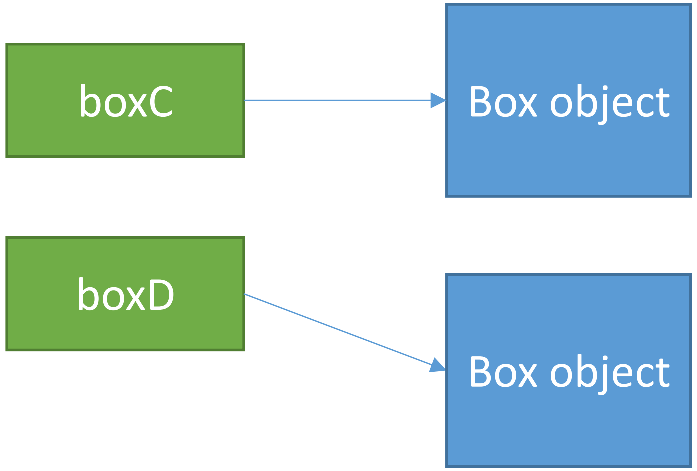
Copying Objects

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

```
public Sphere(Sphere s) {
  radius = s.getRadius();
}
```

Copying Objects

```java
public static void main(String[] args) {
  Sphere sphere1 = new Sphere(11);
  Sphere sphere2 = new Sphere(sphere1);
  sphere2.setRadius(20);
  System.out.print("Sphere 1's radius = ");
  System.out.println(sphere1.getRadius());
  System.out.print("Sphere 2's radius = ");
  System.out.println(sphere2.getRadius());
}
```



```
Sphere 1’s radius = 11
Sphere 2’s radius = 20
```

Object Equality

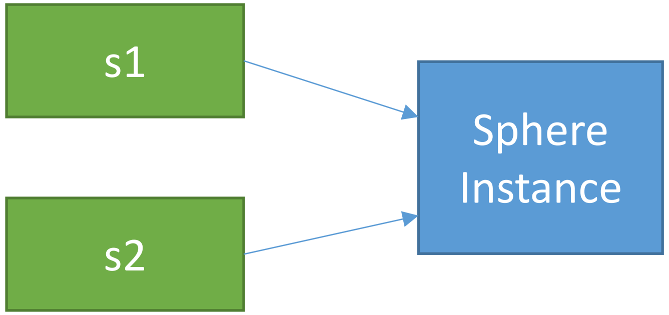
- What does it mean for two instances of an object to be "equal" to each other?
  - Do all of the fields in the two instances need to have the same values? Maybe only some fields?

- 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”)

Object Equality

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

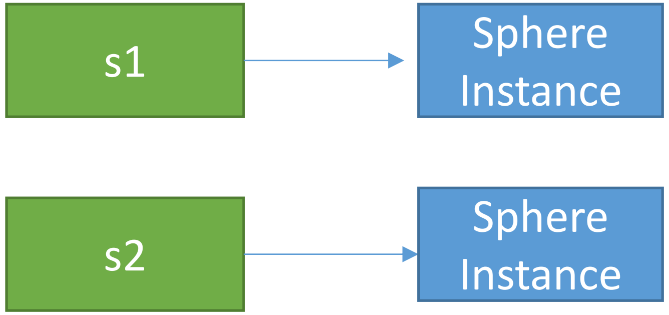
```
Sphere s1 = new Sphere(8);
Sphere s2 = s1;

if(s1 == s2) {
    System.out.println("s1 shares the same reference as s2");
}
```



Object Equality

- Even though s1 and s2's instances have the same radius, that is not what the equality operator checks for.
  - Since s1 and s2 have different references, the equality operator returns false.

```
Sphere s1 = new Sphere(8);
Sphere s2 = new Sphere(8);

if(s1 == s2) {
    System.out.println("s1 shares the same reference as s2");
}
```



Object Equality

- 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.

Object Equality

- This equals method compares the fields of the parameter Sphere object to this Sphere object’s fields.

```java
public boolean equals(Sphere otherSphere) {
  if(otherSphere.getRadius() == radius) {
    return true;
  }
  return false;
}
```

Object Equality

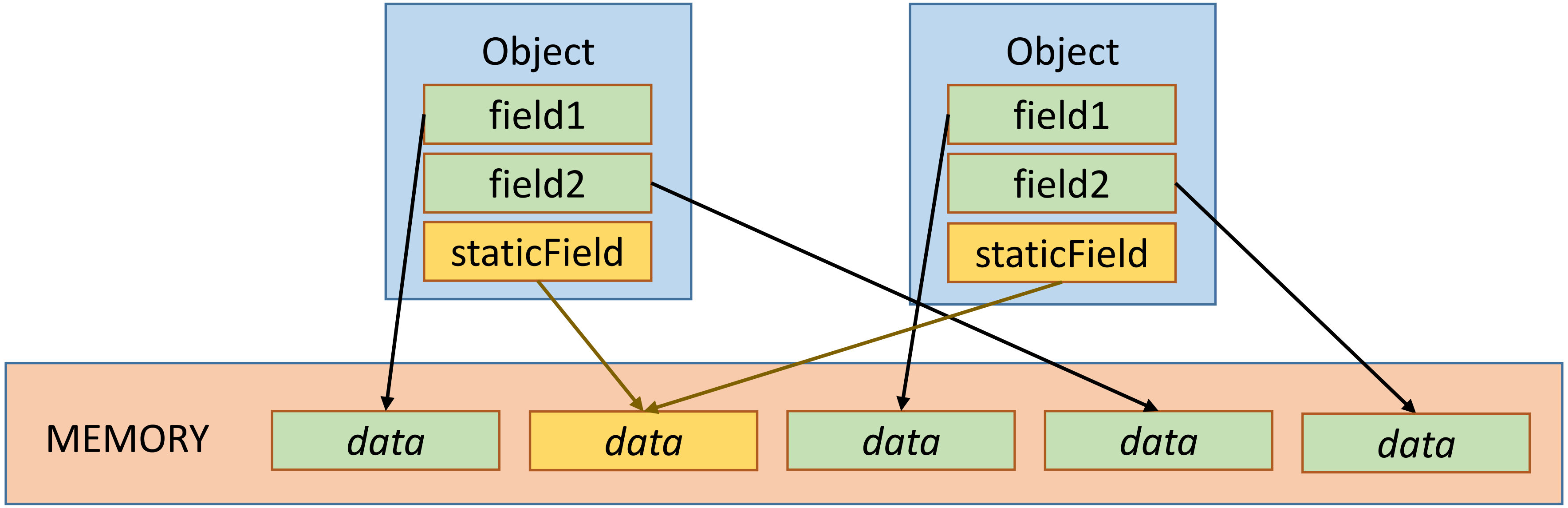
```java
Sphere s1 = new Sphere(9);
Sphere s2 = new Sphere(9);

if(s1.equals(s2)) {
  System.out.println("s1 and s2 have the same dimensions");
}
else {
  System.out.println("s1 and s2 do not have the same dimensions");
}
```

```
s1 and s2 have the same dimensions
```

```java
public boolean equals(Sphere otherSphere) {
  if(otherSphere.getRadius() == radius) {
    return true;
  }
  return false;
}
```

Static Fields

- A ***static field*** (also called a ***class field***) is a field whose reference is shared across **<u>all</u>** 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:

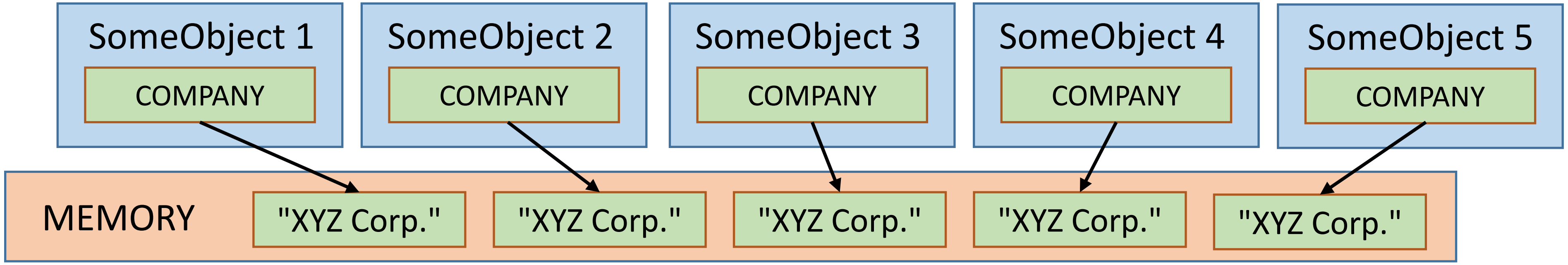
```
private static String myStaticField;
```

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() {
    ...
}
```

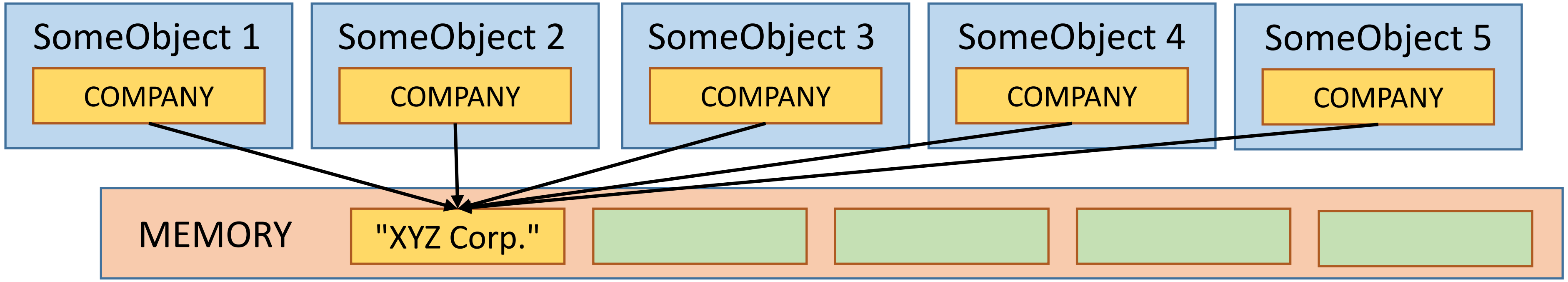
- Every time we instantiate this object, space is allotted for each object's COMPANY field.



Static Fields

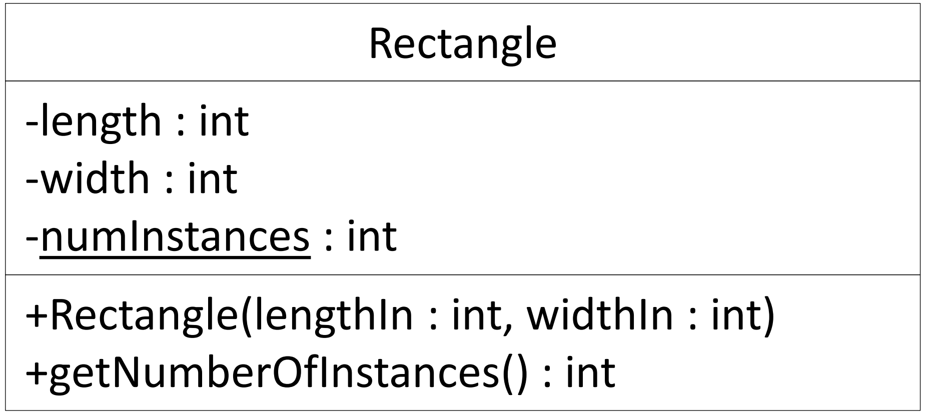
- By making the constant a static field, we can save space:

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

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



Static Members in UML

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



Rectangle

-length : int
-width : int
-numInstances : int

+Rectangle(lengthIn : int, widthIn : int)
+getNumberOfInstances() : int

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;
  ...
}
```

Will not compile

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.