

Object Oriented Programming

Abstraction and Encapsulation

Michael C. Hackett
Computer Science Department

Community
College
of Philadelphia

Lecture Topics

- Basics of Object Oriented Design
 - Objects
 - Classes
- Instance Variables/Fields
- Constructors
- Access Modifiers
- Methods
 - Mutator Methods
 - Accessor Methods
 - Utility Methods
- Class Diagrams
- Overloaded Methods
- String Representation
- Copying Objects
- Object Equality
- Static Fields and Methods

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

What is Object Oriented Programming?

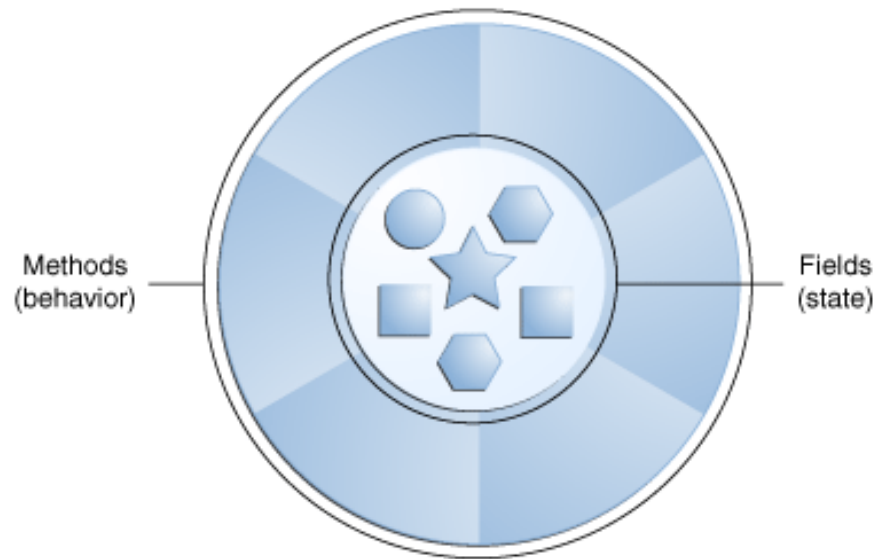
- A programming paradigm where software is written so that a program functions as a system of objects.
 - A different way of designing software.
- The objects interact with each other to complete the program's tasks.
- Software objects are coded to contain information about themselves and allow interaction with other objects.

What are Objects?

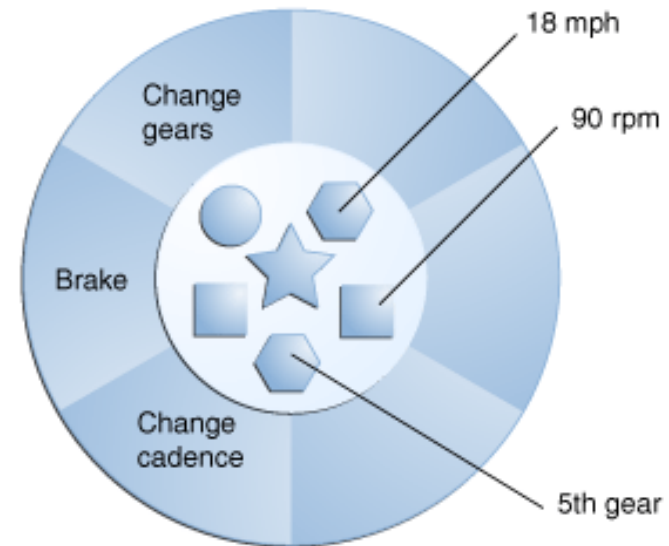
- A software object is *conceptually* similar to real world objects.
- Real world objects all have two characteristics:
 - They have ***attributes***- properties that make something unique.
 - A bicycle's attributes could be its speed, color, tire size, etc.
 - They have ***behaviors***- actions that it can do.
 - A bicycle's behaviors could be pedaling, braking, changing gear, etc.
- When we model a software object, it too has attributes and behaviors.
 - Objects store their attributes in variables referred to as ***fields***.
 - Objects expose their behaviors as ***functions***.

A Bicycle Object

Software Object



Bicycle Object



Why apply Object Oriented Programming?

- Modularity (Abstraction)
 - The code for an object is written and maintained separately from the code of other objects.
- Information Hiding (Encapsulation)
 - By interacting with the object's methods, the details of its internal implementation is hidden.
- Reusability
 - You can create multiple instances of an object.

Abstraction

- ***Abstraction*** is an OOD principle that software objects are able to function as individual entities.
- Using fields, the object can hold information about itself.
- Using methods, the object can perform various actions and operations and communicate with other objects.

Encapsulation

- ***Encapsulation*** is an OOD principle that suggests we design objects so that all relevant data (attributes) and behaviors (methods) are together.
 - The object also controls how its information is seen or changed.
- Only attributes and behaviors relevant to the object should be in the object.
 - Other data or functions not related to the object's use should be placed in other objects.
 - For example, it wouldn't make sense for a Bicycle object to have a fuel level attribute (but would, perhaps, make sense in a Moped object.)

What are Classes?

- A ***class*** is the blueprint from which objects are created.
- It is the *source code* of the object.
 - The object is the idea, the code in the class is the implementation of the idea.
- When a new software object is created from a class, this is referred to as instantiation.
 - *Creating a new instance of an object.*
 - *Instantiating an object.*

Classes

- ***Class Declaration*** (or ***Class Header***) for an object named “Pyramid” is shown below.

```
class Pyramid {
```

```
}
```



Everything between these braces is the **class body**

The diagram consists of two blue arrows originating from a single point on the right. One arrow points to the opening curly brace '{' of the class declaration, and the other points to the closing curly brace '}'.

Classes

- We will create a Pyramid class throughout the rest of this lecture.
- After completing our Pyramid class, we will instantiate a Pyramid object in a separate program.
- We have been using classes already.
 - We have only seen the use of a *main* method, which is not required in every class.
 - A main method allows the JVM to know where to start executing a Java program.

Instance Variables

- An ***instance variable*** is a field that is accessible by the methods of the class.
 - It is declared inside the class, but outside of any method.
- The data stored in an instance variable is unique to each instance of an object.

Declaring Instance Variables

```
class Pyramid {  
  
    int width;  
    int length;  
    int height;  
    String color;  
  
}
```

- Declared just as we have seen before, but not within any particular method.
- Can be initialized at declaration or in a method.

Creating an Instance of an Object

- In a second class (and in a separate source code file) named PyramidTest, we will ***instantiate*** a Pyramid object in its main method (shown below).
 - ***Instantiation*** is the term used when you create an ***instance*** of an object.

```
class PyramidTest {  
  
    public static void main(String[] args) {  
        Pyramid example = new Pyramid();  
    }  
}
```

The Default Constructor

- Objects are instantiated using a special method called a **constructor**.
- Constructors are used to “set up” or *construct* an instance of an object.
- When there are no constructors present in a class, the compiler automatically adds a **default constructor**.
 - This guarantees every object has a constructor.

```
class Pyramid {  
  
    public static void main(String[] args) {  
        Pyramid example = new Pyramid();  
    }  
}
```


Accessing an Instance's Fields

- We can access an instance's fields using dot notation.

```
class PyramidTest {  
  
    public static void main(String[] args) {  
        Pyramid example = new Pyramid();  
  
        //Sets the object's width attribute to 3  
        example.width = 3;  
    }  
}
```

Accessing an Instance's Fields

- We can access an instance's fields using dot notation.

```
class PyramidTest {  
  
    public static void main(String[] args) {  
        Pyramid example = new Pyramid();  
  
        //Sets the object's width attribute to 3  
        example.width = 3;  
        System.out.println("The pyramid's width is " + example.width);  
    }  
}
```

The pyramid's width is 3

Constructors

- Constructors are a special type of method/behavior that prepares the instance of the object.
- Constructors almost always have the same name as the class file.
 - Not always the case as a single source code file can contain multiple classes.
- Classes may have multiple constructors.
 - This is referred to as ***overloading*** or having ***overloaded constructors***.

The No Argument (No-Arg) Constructor

```
class Pyramid {  
  
    int width;  
    int length;  
    int height;  
    String color;  
  

```

```
    Pyramid() {  
        width = 1;  
        length = 1;  
        height = 5;  
        color = "White";  
    }  
  

```

```
}
```

- Replaces the default constructor added by the compiler.
 - A class only has a default constructor when the class has no constructors defined.
- The code in the no-arg constructor's body will be executed when the no-arg constructor is called.
- The names of any constructors must always match the class name.

Without the No-Arg Constructor

```
class PyramidTest {  
  
    public static void main(String[] args) {  
        Pyramid example = new Pyramid();  
  
        System.out.println("The pyramid's width is " + example.width);  
        System.out.println("The pyramid's length is " + example.length);  
        System.out.println("The pyramid's height is " + example.height);  
        System.out.println("The pyramid's color is " + example.color);  
    }  
}
```

```
The pyramid's width is 0  
The pyramid's length is 0  
The pyramid's height is 0  
The pyramid's color is null
```

With the No-Arg Constructor defined

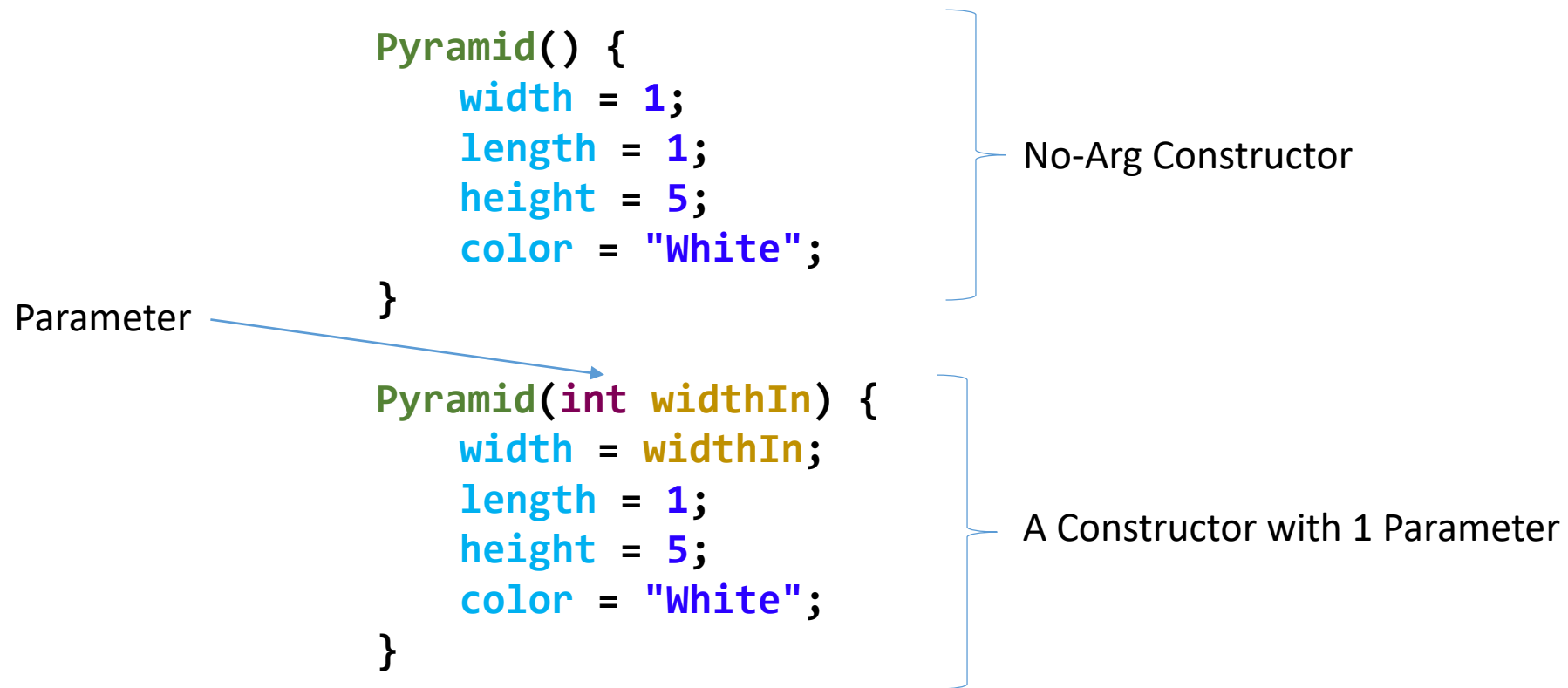
```
class PyramidTest {  
  
    public static void main(String[] args) {  
        Pyramid example = new Pyramid();  
  
        System.out.println("The pyramid's width is " + example.width);  
        System.out.println("The pyramid's length is " + example.length);  
        System.out.println("The pyramid's height is " + example.height);  
        System.out.println("The pyramid's color is " + example.color);  
    }  
}
```

```
The pyramid's width is 1  
The pyramid's length is 1  
The pyramid's height is 5  
The pyramid's color is white
```

Passing Values into a Constructor

- In many cases, you will want to pass data into a constructor to set the values of the fields.
- ***Parameters*** are variables that represent data that is given (or *passed*) to a constructor.
 - Data given to the constructor are called ***arguments***.
- If a constructor declares a parameter list, a value for each parameter must be present.
 - The list must include the data type for each parameter.

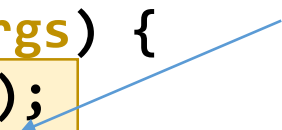
Constructors



Constructors

- Now, we have a second way to instantiate an instance of a Pyramid object.

```
class PyramidTest {  
  
    public static void main(String[] args) {  
        Pyramid example1 = new Pyramid();  
        Pyramid example2 = new Pyramid(5);  
  
        System.out.println("The first pyramid's width is " + example1.width);  
        System.out.println("The second pyramid's width is " + example2.width);  
    }  
}
```



```
The first pyramid's width is 1  
The second pyramid's width is 5
```

Constructors

```
Pyramid() {  
    width = 1;  
    length = 1;  
    height = 5;  
    color = "White";  
}
```

No-Arg Constructor

```
Pyramid(int widthIn) {  
    width = widthIn;  
    length = 1;  
    height = 5;  
    color = "White";  
}
```

A Constructor with 1 Parameter

```
Pyramid(int widthIn, int lengthIn) {  
    width = widthIn;  
    length = lengthIn;  
    height = 5;  
    color = "White";  
}
```

A Constructor with 2 Parameters

Constructors

```
Pyramid(int widthIn, int lengthIn, int heightIn) {  
    width = widthIn;  
    length = lengthIn;  
    height = heightIn;  
    color = "White";  
}
```

A Constructor with
3 Parameters

```
Pyramid(int widthIn, int lengthIn, int heightIn, String colorIn) {  
    width = widthIn;  
    length = lengthIn;  
    height = heightIn;  
    color = colorIn;  
}
```

A Constructor with
4 Parameters

Constructors

- Now, there are 5 different ways to instantiate an instance of a Pyramid object.

```
class PyramidTest {  
  
    public static void main(String[] args) {  
        Pyramid example1 = new Pyramid();  
        Pyramid example2 = new Pyramid(5);  
        Pyramid example3 = new Pyramid(7, 7, 15, "Blue");  
  
        System.out.println("The third pyramid's height is " + example3.height);  
        System.out.println("The third pyramid's color is " + example3.color);  
    }  
}
```

```
The third pyramid's height is 15  
The third pyramid's color is Blue
```

Constructor Signatures

- There is no limit to the number of constructors in a class.
- However, each constructor must have a unique ***signature***.
 - A constructor signature consists of its name and parameter list data types.


```
Pyramid()  
Pyramid(int widthIn)  
Pyramid(int widthIn, int lengthIn)  
Pyramid(int widthIn, int lengthIn, int heightIn)  
Pyramid(int widthIn, int lengthIn, int heightIn, String colorIn)
```


Signatures:

```
Pyramid()  
Pyramid(int)  
Pyramid(int, int)  
Pyramid(int, int, int)  
Pyramid(int, int, int, String)
```

Constructor Signatures

- If our Pyramid class had the following constructors, it would not compile.
 - Their signatures are not unique.

```
Pyramid(int widthIn) {  Pyramid(int)
    width = widthIn;
    length = 1;
    height = 5;
    color = "White";
}
```

```
Pyramid(int lengthIn) {  Pyramid(int)
    width = 1;
    length = lengthIn;
    height = 5;
    color = "White";
}
```

`Pyramid example = new Pyramid(5);`

There's no way to know which constructor to use.

Access Modifiers

- Access Modifiers specify how classes, fields, and methods can be accessed by other objects.
 - This limits other objects from making changes to the data in the fields or using methods that pertain only to the object's internal implementation.

public

- Modifier that allows a field or method to be accessible to all other objects.

private

- Modifier that does not allow a field or method to be accessible to other objects.

Private Fields

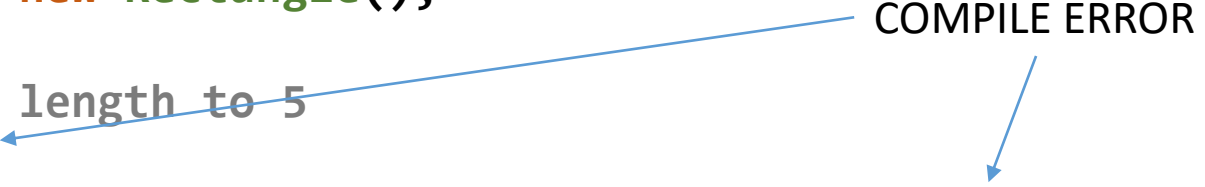
```
public class Rectangle {  
  
    private int length;  
    private int width;  
  
}
```

- This Rectangle class has two private fields.
- These fields cannot be accessed anywhere except from within the Rectangle class.

Private Fields

```
public class RectangleTest {  
  
    public static void main(String[] args) {  
        Rectangle example = new Rectangle();  
  
        //Sets the object's length to 5  
        example.length = 5;  
        System.out.println("The third pyramid's height is " + example.length);  
    }  
}
```

COMPILE ERROR



- A private field is not accessible.
- The only way to change or retrieve the value of a field would be to use a *method*.

Methods

- A ***method*** is a collection of statements that operates on an object's internal data.
 - After the method finishes executing its code, it may or may not return data.
- The code in a method runs when the method is *called*.

Methods

- Most methods in an object are either a mutator method or an accessor method.
- A **mutator method** changes (or “sets”) data in an object.
 - Colloquially called a “setter” method.
- An **accessor method** retrieves (or “gets”) data from an object.
 - Colloquially called a “getter” method.
- The use of mutators and accessors allows the object to specify how (and if) its private data is accessed or changed.

Mutator Method (Length)

Diagram illustrating the components of the method signature:

- Access Modifier (Optional): `public`
- Return Type: `void`
- Method Name: `setLength`
- Parameter: `(int lengthIn)`

```
public void setLength(int lengthIn) {  
    length = lengthIn;  
}
```

- This method will change the current value of the object's length field.
- A method that does not return data must have a **void** return type.
 - Mutators typically do not return data. They “set” data- they don’t “get” data.

Accessor Method (Length)

Access Modifier (Optional) Return Type Method Name Parameter List

```
public int getLength() {  
    return length;  
}
```

- This method will return the current value of our gear instance variable.
- A ***return statement*** indicates the value that is returned by the method.
- The type of data returned must match the method's return type.
 - Since this method's return type is int, the method can only return an int value when called.

Using Method Calls to set/get an Instance's Data

```
public class RectangleTest {  
  
    public static void main(String[] args) {  
        Rectangle example = new Rectangle();  
  
        example.setLength(7);  
        System.out.println("The rectangle's length is " + example.getLength());  
    }  
}
```

Argument

Parameter

The rectangle's length is 7

```
public void setLength(int lengthIn) {  
    length = lengthIn;  
}  
  
public int getLength() {  
    return length;  
}
```

Accessor and Mutator Method (Width)

```
public void setWidth(int widthIn) {  
    width = widthIn;  
}
```

```
public int getWidth() {  
    return width;  
}
```

Using Method Calls to set/get an Instance's Data

```
public class RectangleTest {  
  
    public static void main(String[] args) {  
        Rectangle example = new Rectangle();  
  
        example.setLength(5);  
        example.setWidth(10);  
        System.out.println("The rectangle's length is " + example.getLength());  
        System.out.println("The rectangle's width is " + example.getWidth());  
    }  
  
}
```

```
The rectangle's length is 5  
The rectangle's width is 10
```


Checking Data

- Using mutators to change the state of the object gives the object control of its state.
- For example, we can add checks to the length and width field mutators to check for negative distances or distances of zero.
 - The object (through its methods) can decide what happens.

Checking Data

- Additional code can be added to the mutator to check the argument passed to it.
 - Now, a zero or negative argument will cause the length field to be assigned 1.

```
public void setLength(int lengthIn) {  
    if(lengthIn <= 0) {  
        length = 1;  
    }  
    else {  
        length = lengthIn;  
    }  
}
```

Checking Data

- A similar check can be added in the mutator for the width field.

```
public void setWidth(int widthIn) {  
    if(widthIn <= 0) {  
        width = 1;  
    }  
    else {  
        width = widthIn;  
    }  
}
```

Using Method Calls to set/get an Instance's Data

```
public class RectangleTest {  
  
    public static void main(String[] args) {  
        Rectangle example = new Rectangle();  
  
        example.setLength(-66);  
        example.setWidth(0);  
        System.out.println("The rectangle's length is " + example.getLength());  
        System.out.println("The rectangle's width is " + example.getWidth());  
    }  
  
}
```

The rectangle's length is 1
The rectangle's width is 1

It's now impossible for a Rectangle object's length or width fields to be zero or negative.

Utility Methods

- Another type of method is a **utility method**.
 - These are methods that aren't wholly concerned with setting or getting data to/from an object, but perform some other operation or function.
 - A utility function may or may not return data.
- A Rectangle object might have a utility method to calculate the area or perimeter of the shape.

Utility Method (Area)

- This calculateArea method will calculate and return the area of the shape.

```
public int calculateArea() {  
    int area = length * width;  
    return area;  
}
```

Utility Method (Perimeter)

- This calculatePerimeter method will calculate and return the perimeter of the shape.

```
public int calculatePerimeter() {  
    int perimeter = 2 * (length + width);  
    return perimeter;  
}
```

Utility Methods

```
public class RectangleTest {  
  
    public static void main(String[] args) {  
        Rectangle example = new Rectangle();  
  
        example.setLength(6);  
        example.setWidth(5);  
        int calc1 = example.calculateArea();  
        int calc2 = example.calculatePerimeter();  
        System.out.println("The rectangle's area is " + calc1);  
        System.out.println("The rectangle's perimeter is " + calc2);  
    }  
}
```

The rectangle's area is 30
The rectangle's perimeter is 22

Method Naming

- By convention (in Java), method names are in camel-case.
- A method's name should always describe what it does.
 - Accessor methods normally start with “get”.
 - For example: getValue, getName
 - Mutator methods normally start with “set”.
 - For example: setValue, setName
 - Utility functions will vary in name.
 - For example: fixHeaders, validateData, updateValues, toSquareFeet
 - Methods that return a boolean normally start with “is” or “has”.
 - For example: isFinished, isClosed, hasCorrectData

Method Access

- Accessor and mutator methods are normally public.
 - Other objects need to call these methods to interact with the object.
- Utility methods may or may not be public.
 - It depends on the object's design.
 - Make the method private if you don't want other objects to call it.

Method Signatures

- Like constructors, methods have signatures.
- A method signature consists of its name and parameter list data types.
- All methods in a class must have unique signatures.

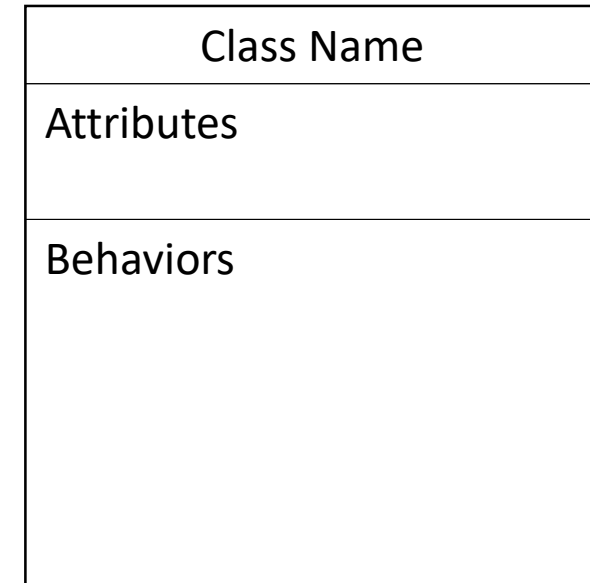
`public void exampleMethod(int arg1, String arg2)` —————→ `exampleMethod(int, String)`

`public int exampleMethod2()` —————→ `exampleMethod2()`

`public String exampleMethod3(String[] arg1)` —————→ `exampleMethod3(String[])`

Class Diagrams

- Unified Modeling Language provides a set of standard diagrams for graphically depicting an object oriented system.
- In UML, each class is shown as a box, with three sections:
 - The Class Name
 - Class Attributes (Fields/Variables)
 - Class Behaviors (Constructors and Methods)

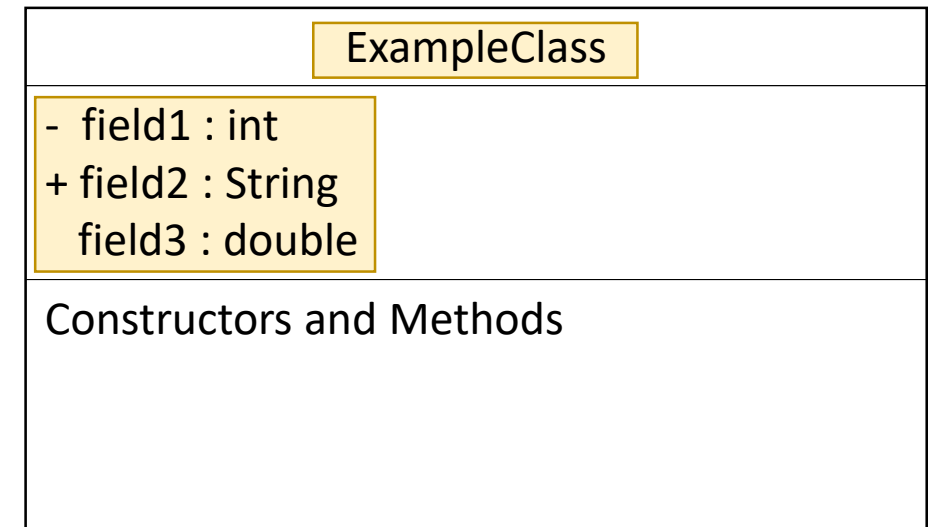


Class Diagrams

- When displaying fields (and parameter names in methods) in a class diagram, the format to use is:
 - name : type
- Access specifier symbols:
 - + public fields/methods
 - - private fields/methods
 - None (or ~) No access specifier.

Class Diagram (Fields)

```
class ExampleClass {  
    private int field1;  
    public String field2;  
    double field3;  
  
    public ExampleClass(int argIn) {  
        //Constructor code  
    }  
  
    public int method1(double arg1In, int arg2In) {  
        //method1 code  
    }  
  
    private void method2() {  
        //method2 code  
    }  
}
```



Class Diagrams (Constructors and Methods)

- When displaying a constructor in a class diagram, the format to use is:
 - `name(arg : type, ...)`
- When displaying a method in a class diagram, the format to use is:
 - `name(arg : type, ...) : returnType`
- Access modifier symbols used are the same as for fields.

Class Diagram (Constructors and Methods)

```
class ExampleClass {  
    private int field1;  
    public String field2;  
    double field3;
```

```
    public ExampleClass(int argIn) {  
        //Constructor code  
    }  
  
    int method1(double arg1In, int arg2In) {  
        //method1 code  
    }  
  
    private void method2() {  
        //method2 code  
    }  
}
```

ExampleClass
- field1 : int + field2 : String field3 : double
+ ExampleClass(argIn : int) method1(arg1In : double, arg2In : int) : int - method2() : void

```
}
```


Overloaded Methods

- A method is ***overloaded*** when two or more methods share the same name but have *different parameter types/lists*.
- No limit to the number of times a method can be overloaded.

```
public void setLength(int lengthIn) {  
    if(lengthIn <= 0) {  
        length = 1;  
    }  
    else {  
        length = lengthIn;  
    }  
}
```

```
public void setLength(double lengthIn) {  
    if(lengthIn <= 0) {  
        length = 1;  
    }  
    else {  
        length = (int)Math.round(lengthIn);  
    }  
}
```

Overloaded Methods

- Overloaded methods give the appearance of one method that handles multiple types of arguments.
 - Overloaded methods are also known as ***polymorphic methods*** for this reason.

```
class RectangleTest {  
  
    public static void main(String[] args) {  
        Rectangle example = new Rectangle();  
        example.setLength(7);  
        example.setLength(4.3); //Changes the length to 4.3 (rounded to 4)  
    }  
}
```

Overloaded Methods

- It's common practice to chain overloaded methods together, just as we chain constructors.

```
public void setLength(int lengthIn) {  
    if(lengthIn <= 0) {  
        length = 1;  
    }  
    else {  
        length = lengthIn;  
    }  
}  
  
public void setLength(double lengthIn) {  
    setLength((int)Math.round(lengthIn));  
}  
  
public void setLength(String lengthIn) {  
    setLength(Integer.parseInt(lengthIn));  
}
```

String Representation

- 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
```

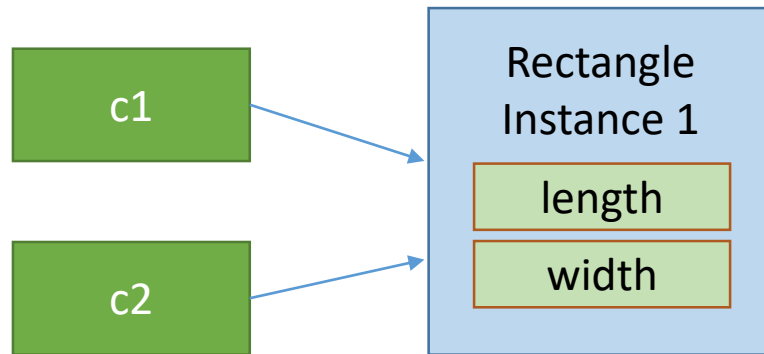
```
Area: 986.96...
```

```
Circumference: 62.83...
```

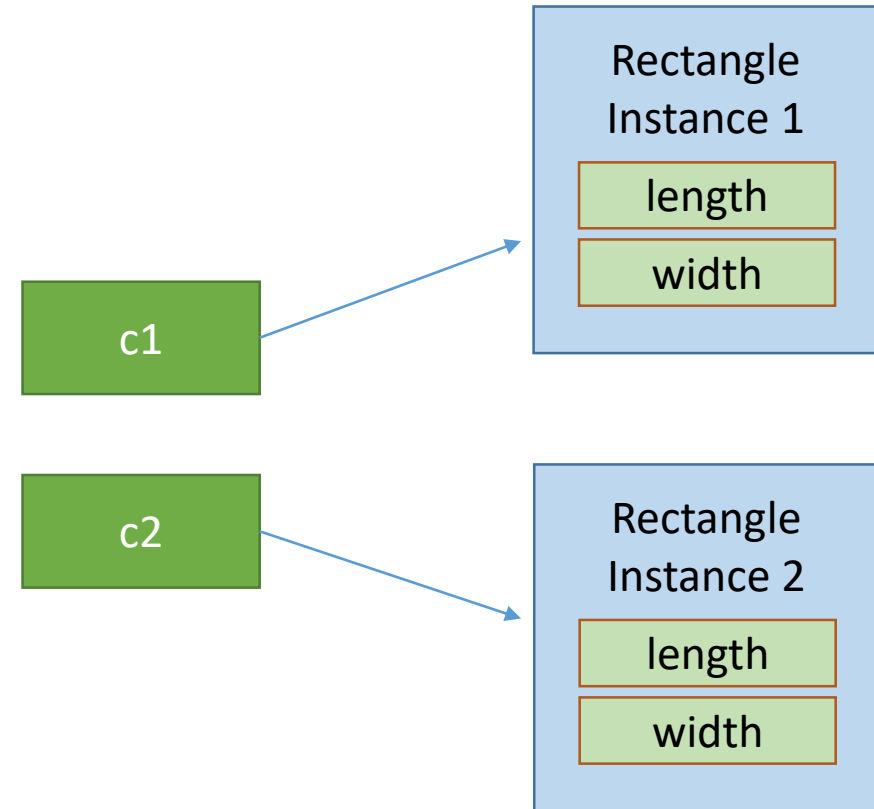

Shallow Copy vs Deep Copy

- 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



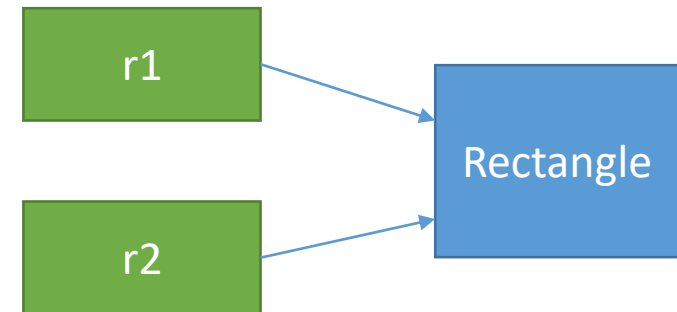
Deep Copy

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) {
```

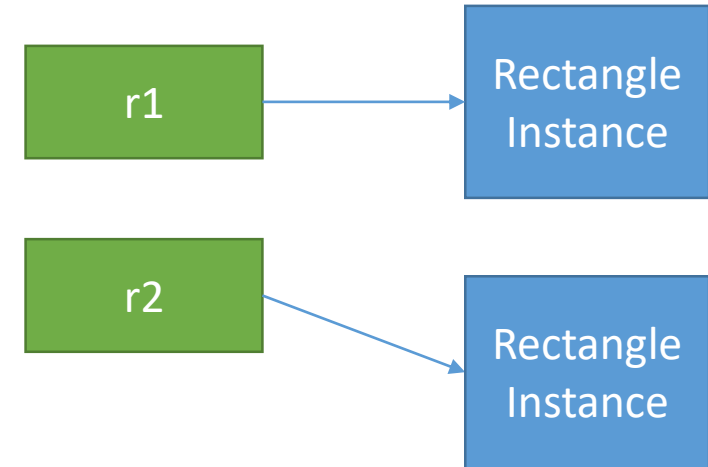
```
    Rectangle r1 = new Rectangle(11, 5);  
    Rectangle r2 = r1.clone();  
    r1.setLength(20);  
    System.out.print("r1's length = ");  
    System.out.println(r1.getLength());  
    System.out.print("r2's length = ");  
    System.out.println(r2.getLength());
```

```
}
```

r1's length = 20

r2's length = 11

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();  
}
```

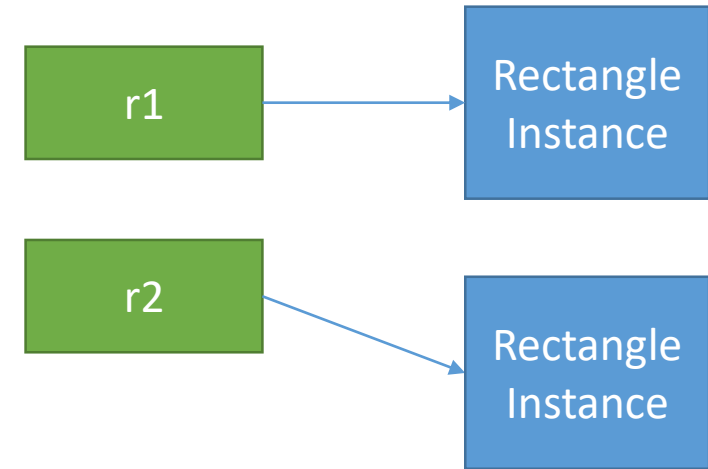
Copy Constructor

```
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();  
}
```

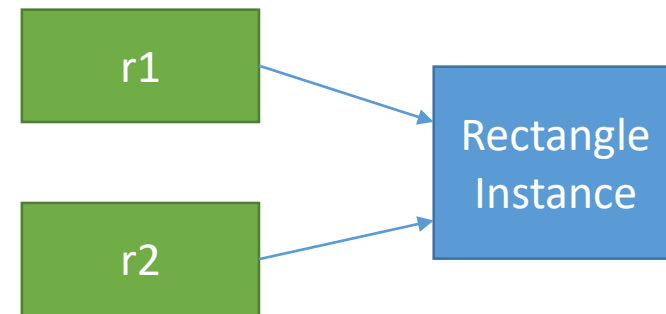

Equality of Instances

- 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.

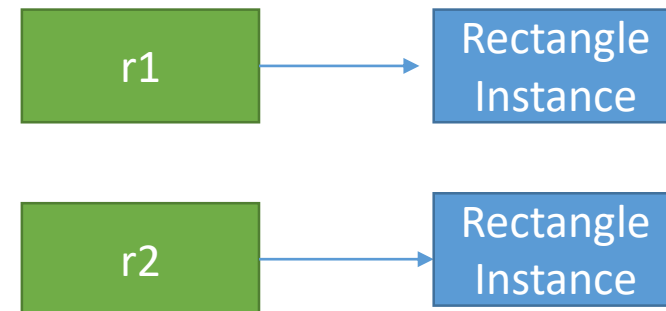
```
Rectangle r1 = new Rectangle(8, 9);  
Rectangle r2 = r1;  
    true  
if(r1 == r2) {  
    System.out.println("r1 shares the same reference as r2");  
}
```



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);

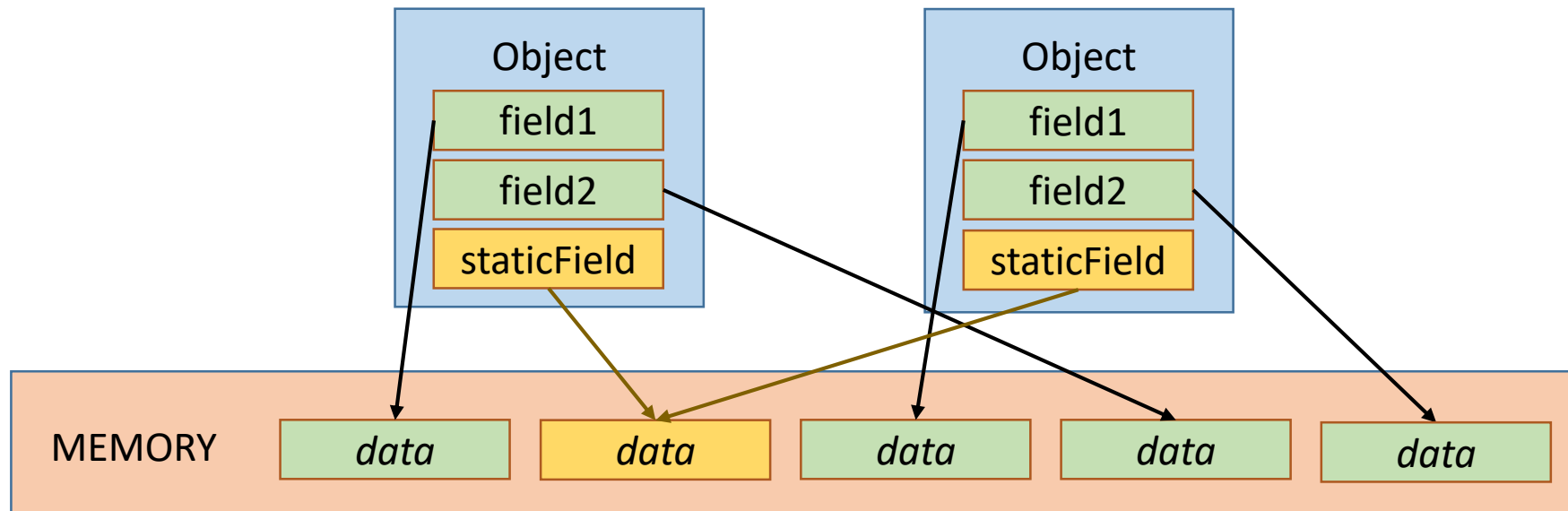
    true
if(r1.equals(r2)) {
    System.out.println("r1 and r2 have the same dimensions");
}
else {
    System.out.println("r1 and r2 do not have the same dimensions");
}
```

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;

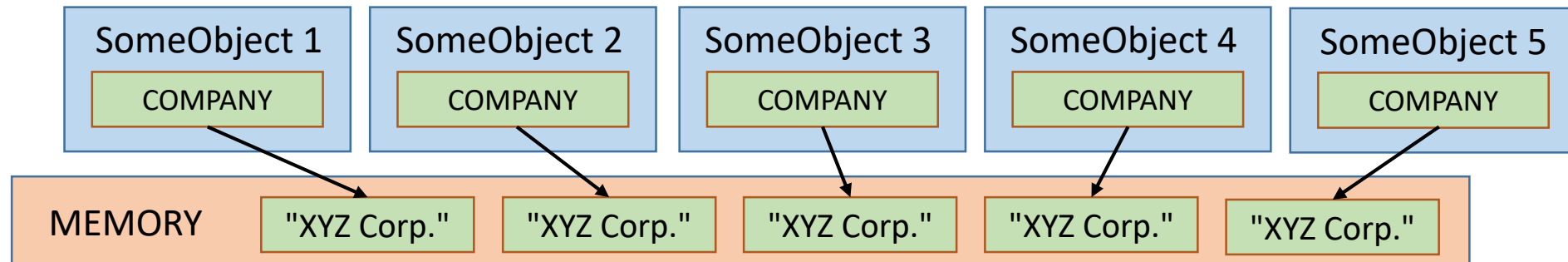
Static Fields

- 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.

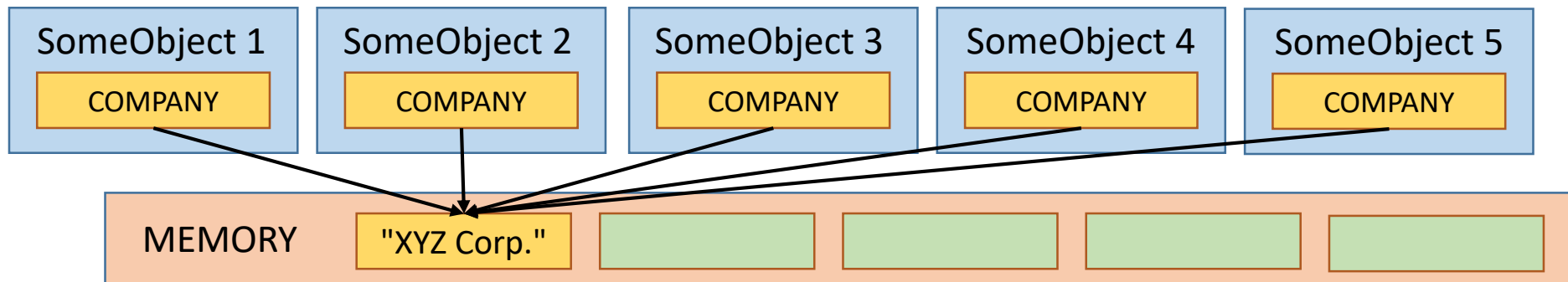


Static Fields

- 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;  
    ...  
}
```

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));  
}
```

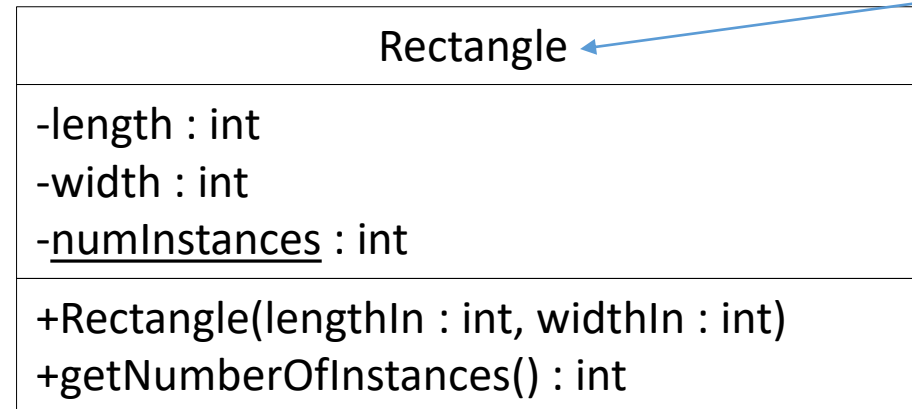
```
public static void main(String[] args) {  
    System.out.print("The sum of 5 and 6 is ");  
    System.out.println(Calculate.add(5, 6));  
}
```

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.

Static Members in UML

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



(As shown on Slide 84)