Classes

Lecture 6a

Topics

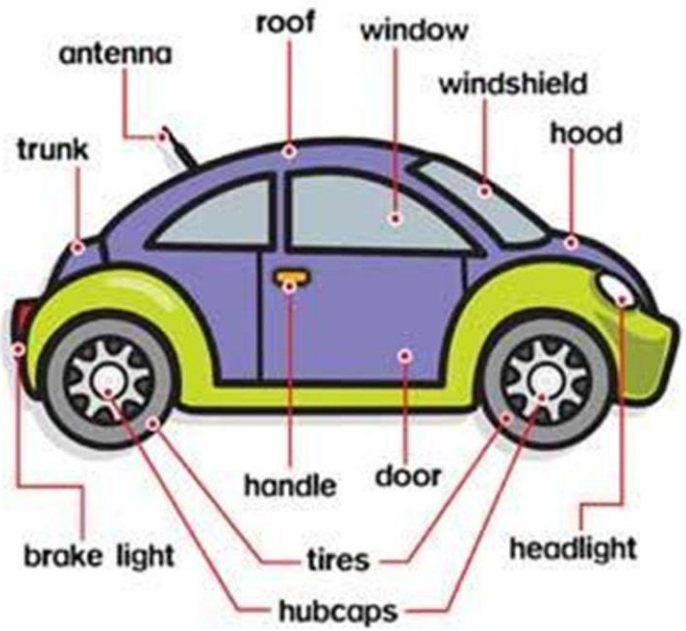
- Objects and Classes
- Writing a Simple Class, Step by Step
- Instance Fields and Methods
- Constructors
- The “this” Reference Variable
- Passing Objects as Arguments
- Overloading Methods and Constructors
- Scope of Instance Fields
- Static Class Members
- Writing an equals Method
- Methods that Copy Objects
- Aggregation
- Garbage Collection

Objects and Classes (1 of 10)

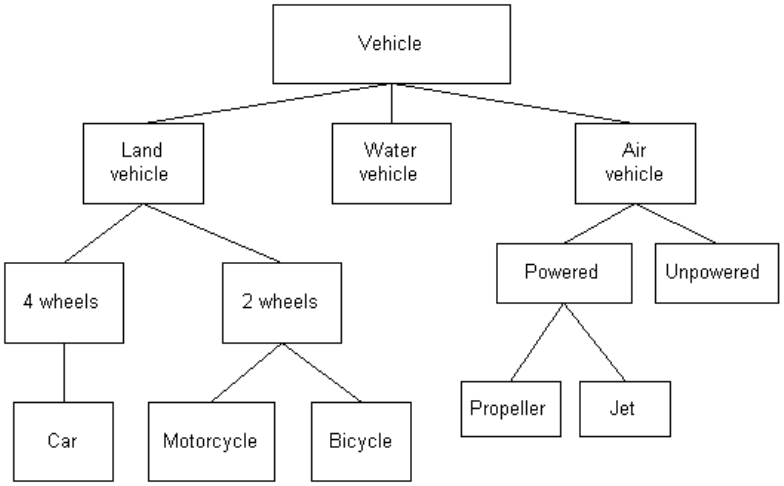
- An object exists in memory and performs a specific task.
- An object is created from a class that contains code describing the object.
- Objects have two general capabilities:
  - Objects can store data. The pieces of data stored in an object are known as *fields*.
  - Objects can perform operations. The operations that an object can perform are known as *methods*.

Objects and Classes (2 of 10)

- A car is a collection of objects



- So is a computer program! In OO languages, you create programs that are made of objects!



Objects and Classes (3 of 10)

- You have already used the following objects:
  - `String` objects, for generating strings
  - `Scanner` objects, for reading input
  - `Random` objects, for generating random numbers
- When a program needs the services of a particular type of object, it creates that object in memory, and then calls that object's methods as necessary.

Objects and Classes (4 of 10)

- Classes: Where Objects Come From
  - A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).
  - You can think of a class as a code "blueprint" that can be used to create a particular type of object.

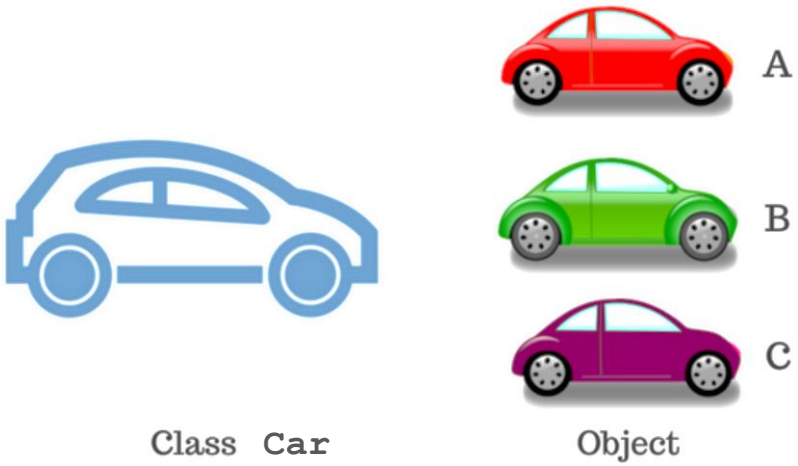
Blueprint

Class Car

Objects and Classes (5 of 10)

- When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed.

- Each object that is created from a class is called an *instance* of the class.



Instances of the class `Car`

Objects and Classes (6 of 10)

- Point to ponder #1:

  So, a class is an object?

  No.

  So, what a class is?

  A description (blueprint) of an object.

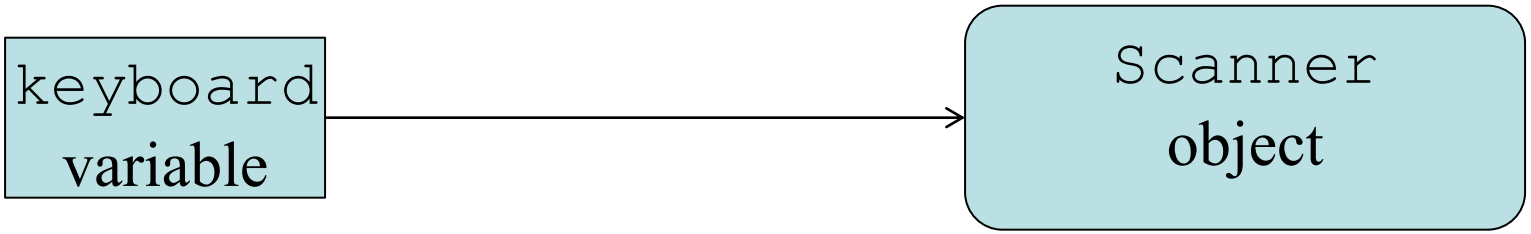
Objects and Classes (7 of 10)

Example:

This expression creates a `Scanner` object in memory.

```
Scanner keyboard = new Scanner(System.in);
```

The object's memory address is assigned to the `keyboard` variable.



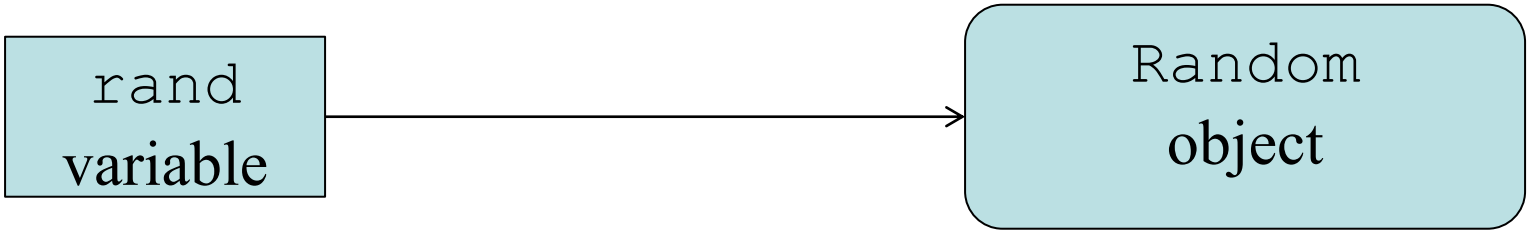
Objects and Classes (8 of 10)

Example:

This expression creates a `Random` object in memory.

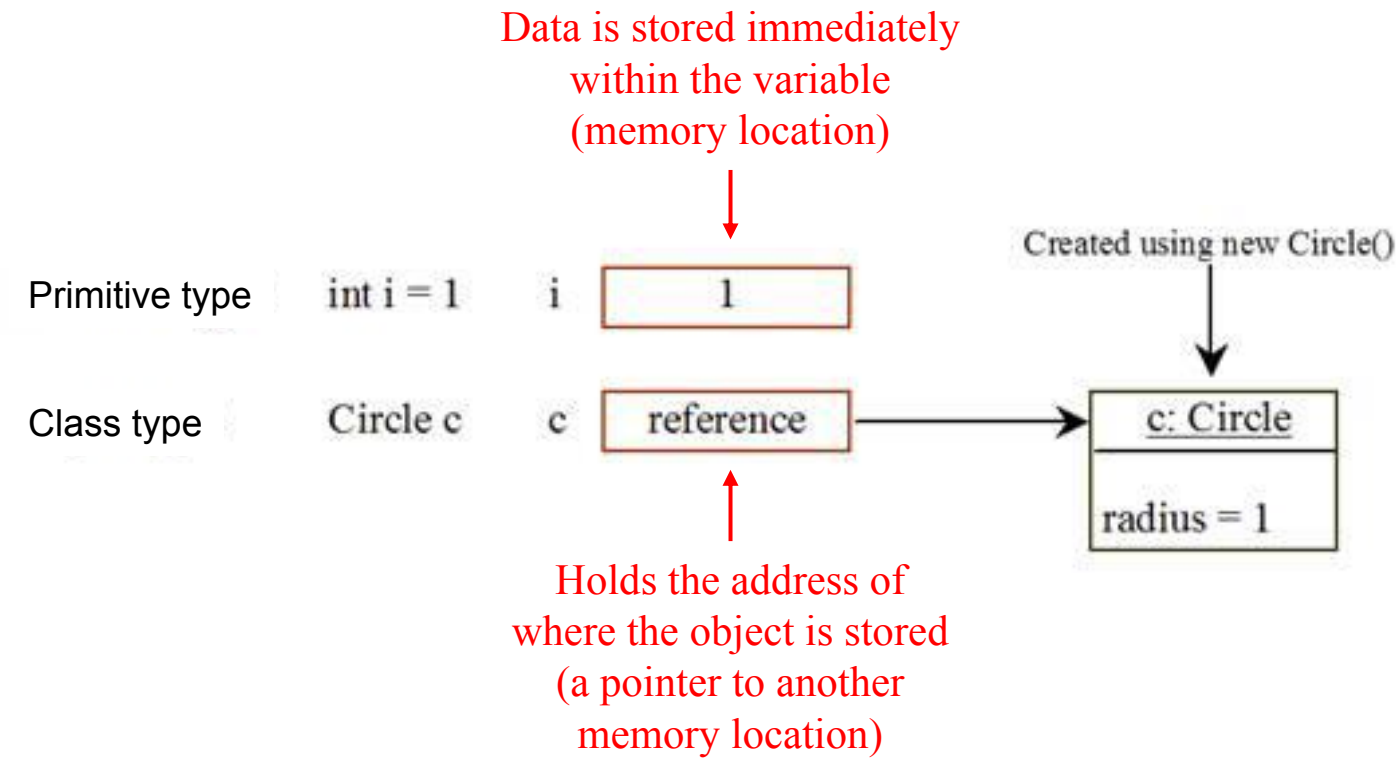
```
Random rand = new Random();
```

The object's memory address is assigned to the `rand` variable.



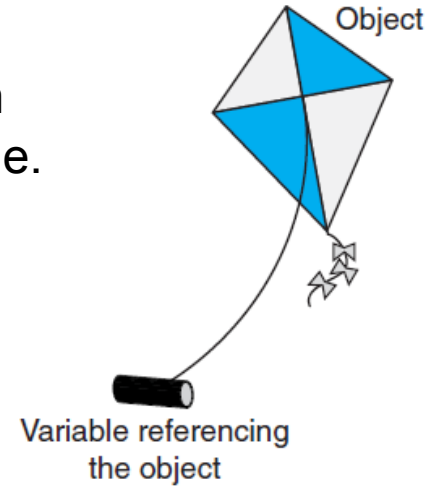
Objects and Classes (9 of 10)

- Difference between primitive and reference variables (class type variables)



Objects and Classes (10 of 10)

- Creating an object typically requires the following two steps:
  - 1. You declare a reference variable.
  - 2. You create the object in memory and assign its memory address to the reference variable.



- Ex: `Random rand = new Random();`

declares a variable named rand which can be used to reference an object of the Random type.

creates an object from the Random class and returns that object's memory address.

Writing a Class, Step by Step (1 of 2)

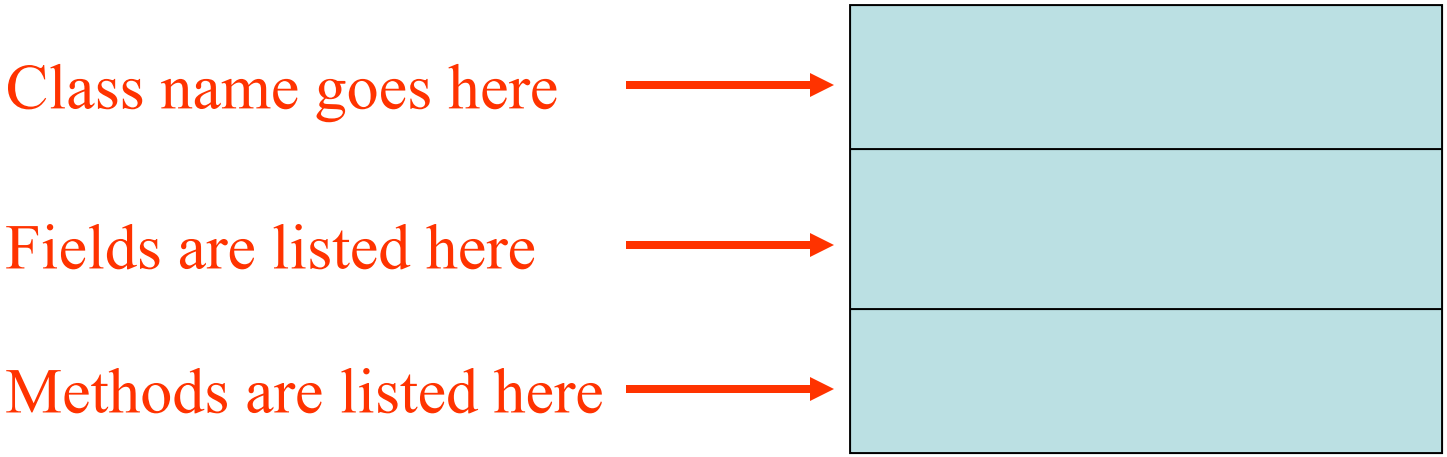
- You can write your own classes - with your own fields and methods - to create objects that you need in a program.
- For instance, a `Rectangle` object will have the following fields:
  - `length`. The length field will hold the rectangle's length.
  - `width`. The width field will hold the rectangle's width.

Writing a Class, Step by Step (2 of 2)

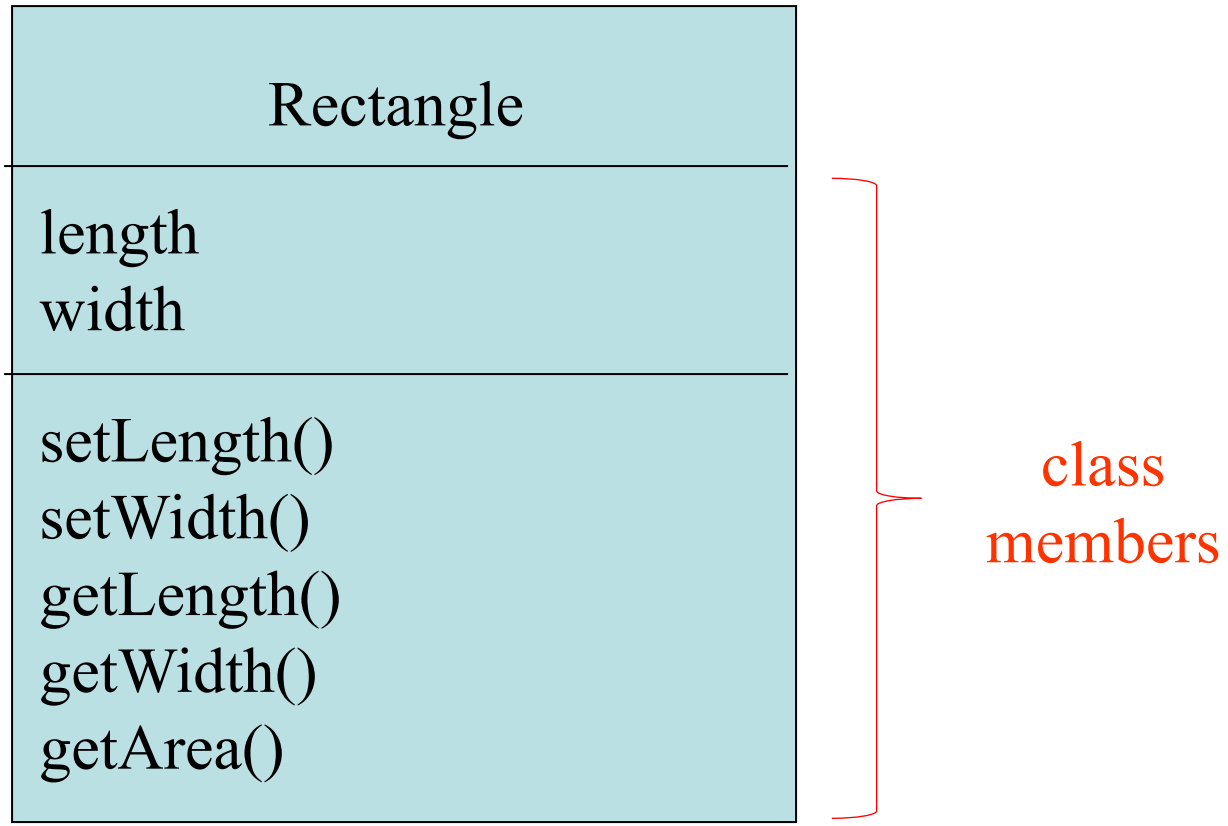
- The `Rectangle` class will also have the following methods:
  - `setLength`. The `setLength` method will store a value in an object's `length` field.
  - `setWidth`. The `setWidth` method will store a value in an object's `width` field.
  - `getLength`. The `getLength` method will return the value in an object's `length` field.
  - `getWidth`. The `getWidth` method will return the value in an object's `width` field.
  - `getArea`. The `getArea` method will return the area of the rectangle, which is the result of the object's `length` multiplied by its `width`.

UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.

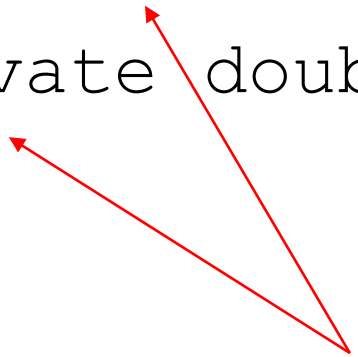


UML Diagram for `Rectangle` class



Writing the Code for the Class Fields

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

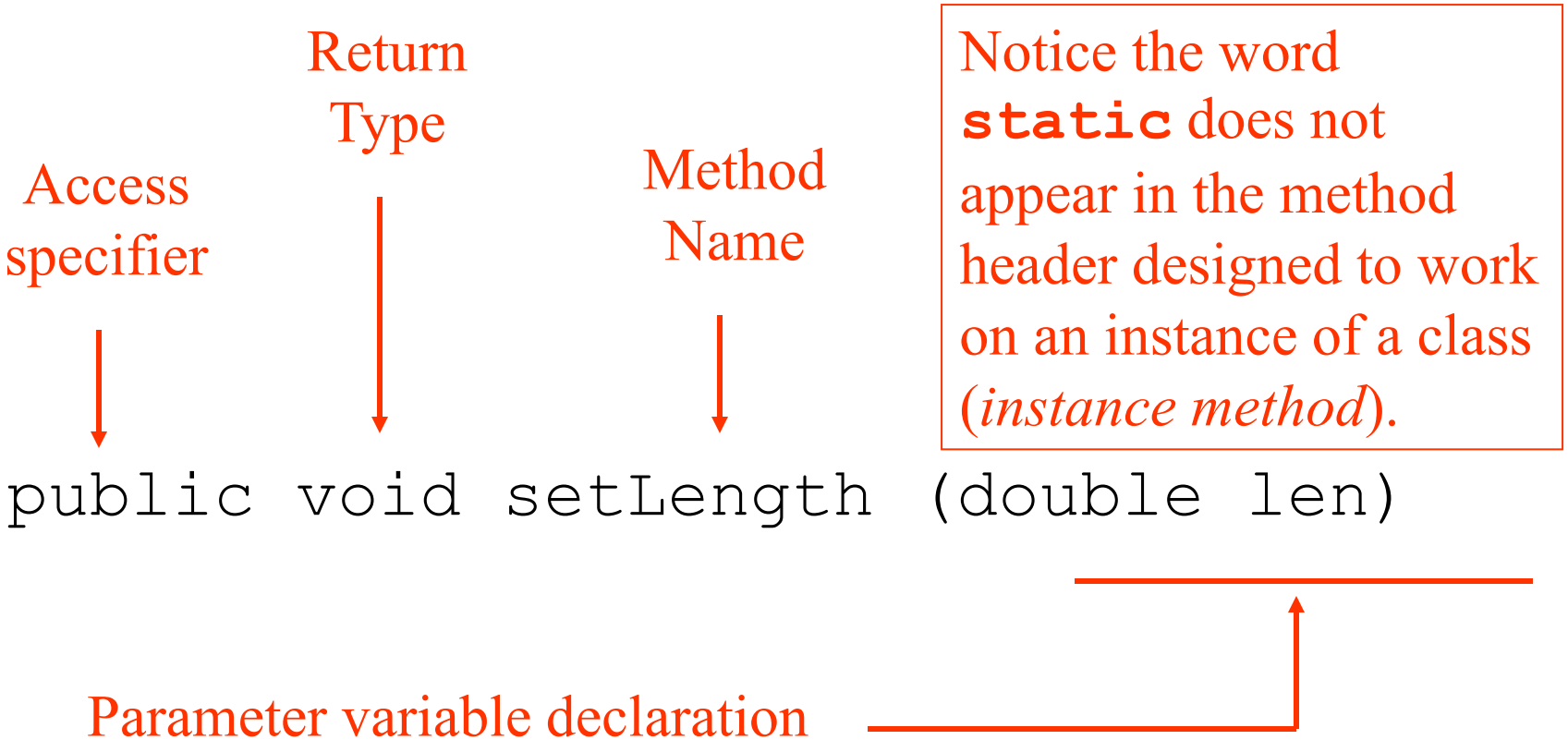


hide its data from code
outside the class.

Access Specifiers

- An access specifier is a Java keyword that indicates how a field or method (members of the class) can be accessed.
- `public`
  - When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
  - When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

Header for the `setLength` Method



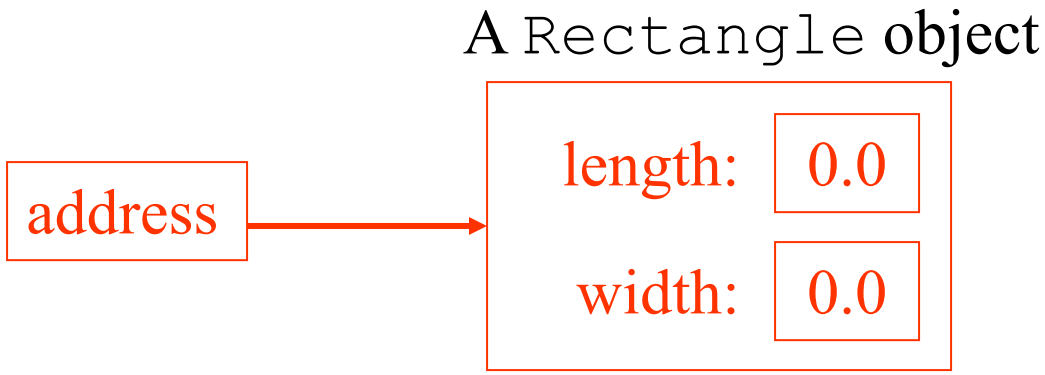
Writing and Demonstrating the `setLength` Method

```
/**
    The setLength method stores a value in the
    length field.
    @param len The value to store in length.
*/
public void setLength(double len)
{
    length = len;
}
```

Creating a `Rectangle` object (1 of 2)

```
Rectangle box = new Rectangle ();
```

The `box` variable holds the address of the Rectangle object.

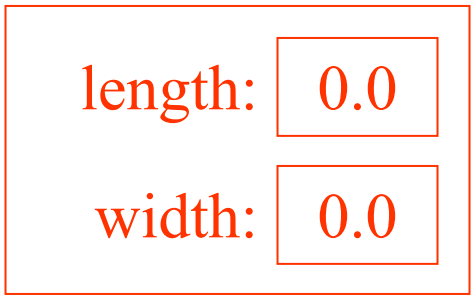


Creating a `Rectangle` object (2 of 2)

- Point to ponder #1:

  Why when we create a new instance (object) of the Rectangle class `length` and `width` are set to zero?

A `Rectangle` object

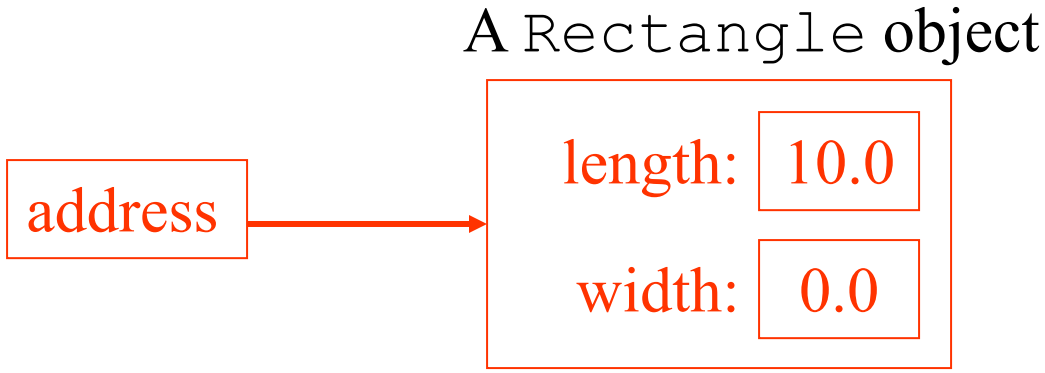


They are initialized by using the default value of the data type.

Calling the `setLength` Method

```
box.setLength(10.0);
```

The `box` variable holds the address of the `Rectangle` object.



This is the state of the box object after the `setLength` *method executes.*

Writing the `getLength` Method

```
/**
    The getLength method returns a Rectangle
    object's length.
    @return The value in the length field.
*/
public double getLength()
{
   return length;
}
```

Similarly, the `setWidth` and `getWidth` methods can be created.

Writing and Demonstrating the `getArea` Method

```
/**
   The getArea method returns a Rectangle
   object's area.
   @return The product of length times width.
*/
public double getArea()
{
   return length * width;
}
```

Examples: Rectangle.java, RectangleDemo.java

CalPolyPomona

Classes

Lecture 6b

Topics

- Objects and Classes
- Writing a Simple Class, Step by Step
- Instance Fields and Methods
- The “this” Reference Variable
- Passing Objects as Arguments
- Overloading Methods and Constructors
- Scope of Instance Fields
- Static Class Members
- Writing an equals Method
- Methods that Copy Objects
- Aggregation
- Garbage Collection

Accessor and Mutator Methods

- Because of the concept of data hiding, fields in a class are private.
- The methods that retrieve the data of fields are called ***accessors***.
- The methods that modify the data of fields are called ***mutators***.
- Each field that the programmer wishes to be viewed by other classes needs an accessor.
- Each field that the programmer wishes to be modified by other classes needs a mutator.

Accessors and Mutators

- For the `Rectangle` example, the accessors and mutators are:
  - `setLength` : sets the value of the `length` field.
    ```
    public void setLength(double len) ...
    ```
  - `setWidth` : sets the value of the `width` field.
    ```
    public void setLength(double w) ...
    ```
  - `getLength` : returns the value of the `length` field.
    ```
    public double getLength() ...
    ```
  - `getWidth` : returns the value of the `width` field.
    ```
    public double getWidth() ...
    ```
- Other names for these methods are ***getters*** and ***setters***.

Data Hiding (1 of 2)

- Data hiding is an important concept in object-oriented programming
- An object hides its internal, private fields from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and make changes to the object's internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.

Data Hiding (2 of 2)

- Data hiding is important because classes are typically used as components in large software systems, involving a team of programmers.

- Data hiding helps enforce the integrity of an object's internal data.

Stale Data (1 of 2)

- Some data is the **result of a calculation**.
- Consider the area of a rectangle.
  - *length* × *width*
- It would be impractical to use an *area* variable here.
- Data that requires the calculation of various factors has the potential to become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a method rather than store it in a variable.

Stale Data (2 of 2)

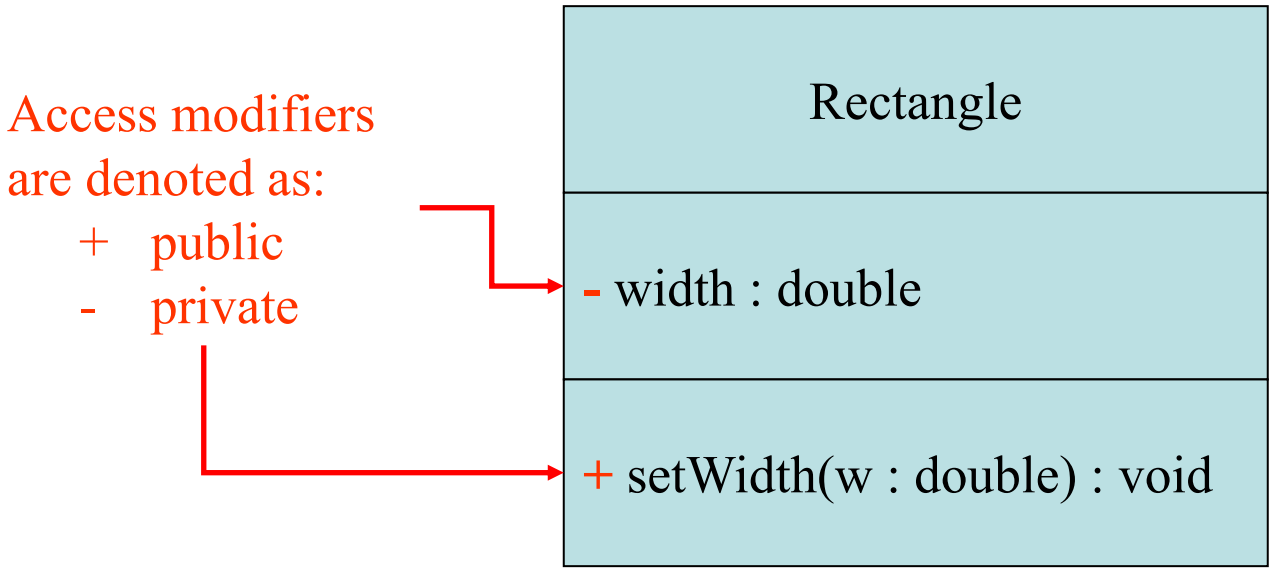
- Rather than use an `area` variable in a `Rectangle` class:

```
public double getArea()
{
 return length * width;
}
```

- This dynamically calculates the value of the rectangle's area when the method is called.
- Now, any change to the `length` or `width` variables will not leave the area of the rectangle stale.

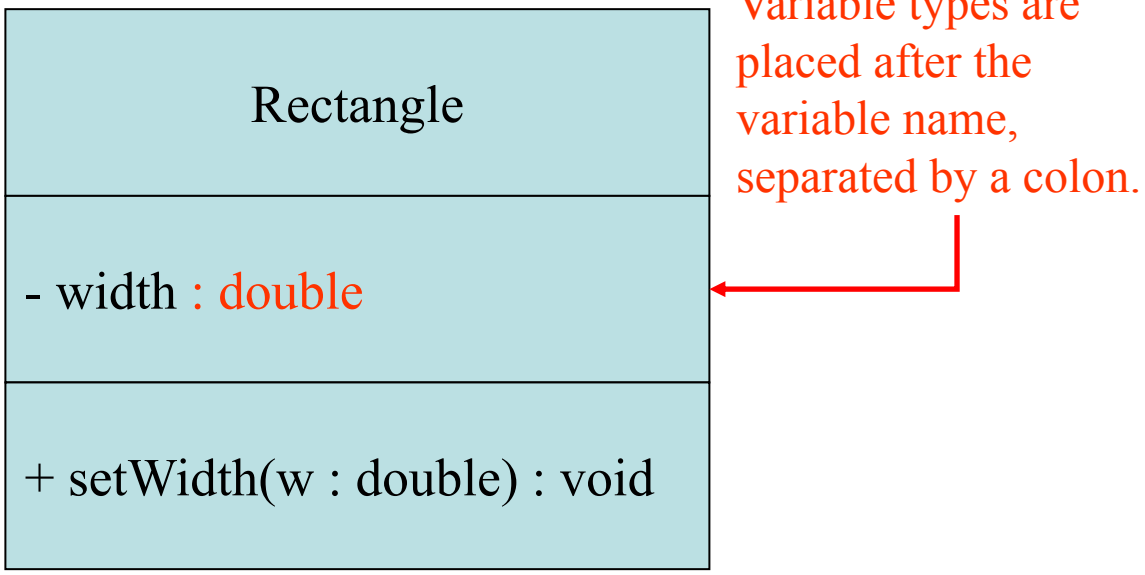
UML Data Type and Parameter Notation (1 of 4)

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



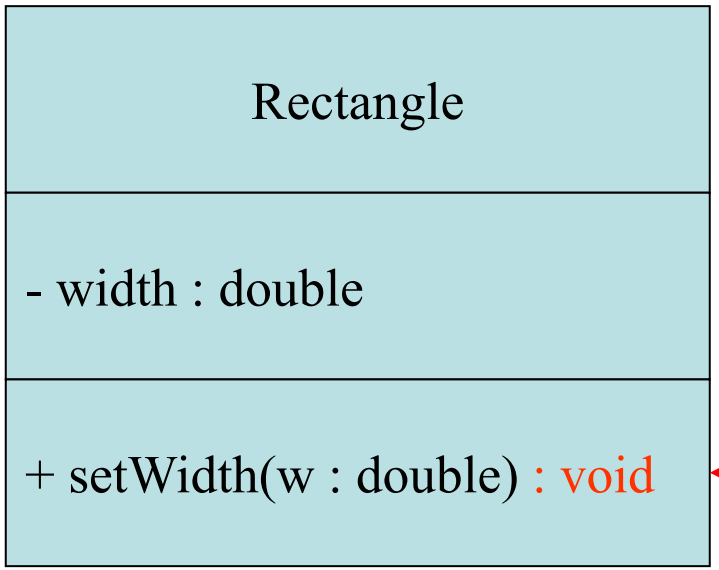
UML Data Type and Parameter Notation (2 of 4)

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



UML Data Type and Parameter Notation (3 of 4)

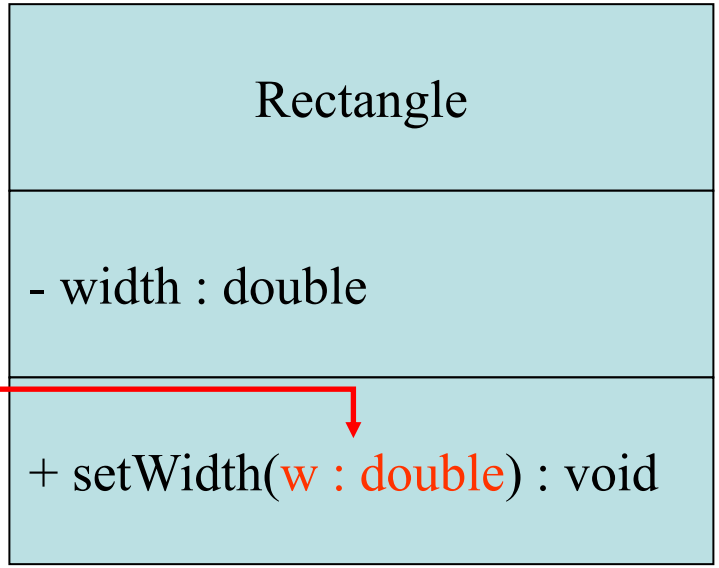
- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



UML Data Type and Parameter Notation (4 of 4)

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.

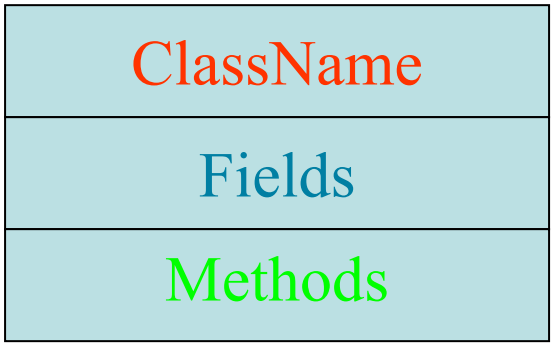
Method parameters and their data types are shown inside the parentheses.



Converting the UML Diagram to Code (1 of 3)

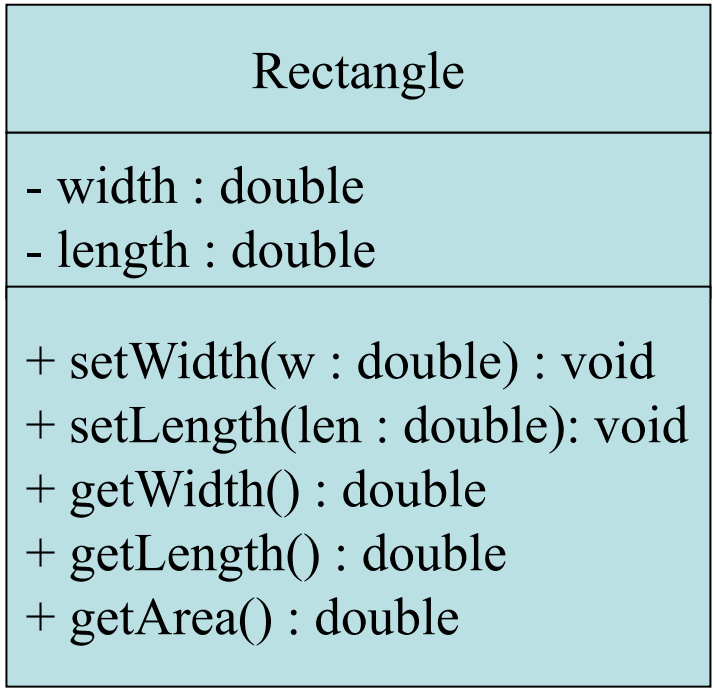
- Putting all this information together, a Java class file can be built easily using the UML diagram.
- The UML diagram parts match the Java class file structure.

```
class header
{
   Fields
   Methods
}
```



Converting the UML Diagram to Code (2 of 3)

The structure of the class can be compiled and tested without having bodies for the methods. Just be sure to put in dummy return values for methods that have a return type other than void.

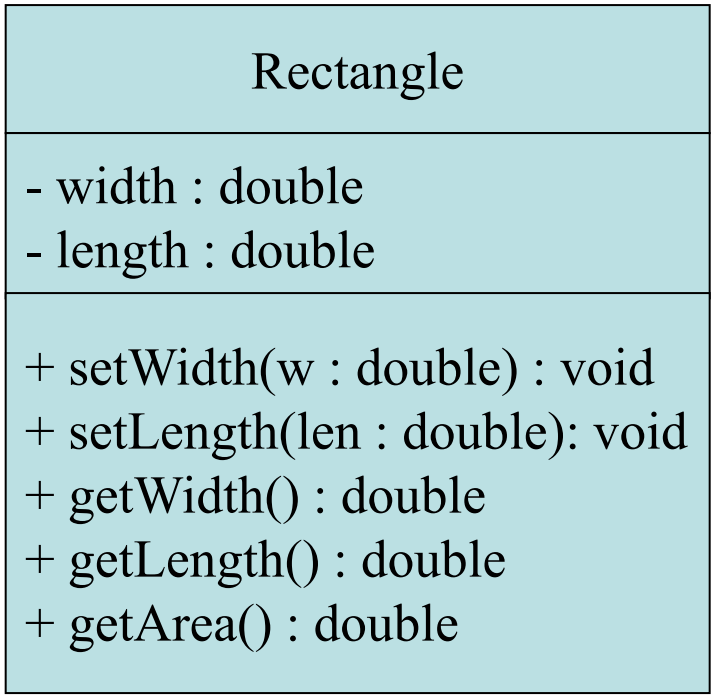



```
public class Rectangle
{
    private double width;
    private double length;

    public void setWidth(double w)
    {
    }
    public void setLength(double len)
    {
    }
    public double getWidth()
    {   return 0.0;
    }
    public double getLength()
    {   return 0.0;
    }
    public double getArea()
    {   return 0.0;
    }
}
```

Converting the UML Diagram to Code (3 of 3)

Once the class structure has been tested, the method bodies can be written and tested.




```
public class Rectangle
{
    private double width;
    private double length;

    public void setWidth(double w)
    {   width = w;
    }
    public void setLength(double len)
    {   length = len;
    }
    public double getWidth()
    {   return width;
    }
    public double getLength()
    {   return length;
    }
    public double getArea()
    {   return length * width;
    }
}
```

Class Layout Conventions

- The layout of a source code file can vary by employer or instructor.
- A common layout is:
  - Fields listed first
  - Methods listed second
    - Accessors and mutators are typically grouped.
- There are tools that can help in formatting layout to specific standards.

Instance Fields and Methods (1 of 2)

- Fields and methods that are declared as previously shown are called *instance fields* and *instance methods*.
- Objects created from a class each have their own copy of instance fields.
- Instance methods are methods that are <u>not</u> declared with a special keyword, `static`.

Instance Fields and Methods (2 of 2)

- Instance fields and instance methods require an object to be created in order to be used.
- See example: RoomAreas.java
- Note that each room represented in this example can have different dimensions.

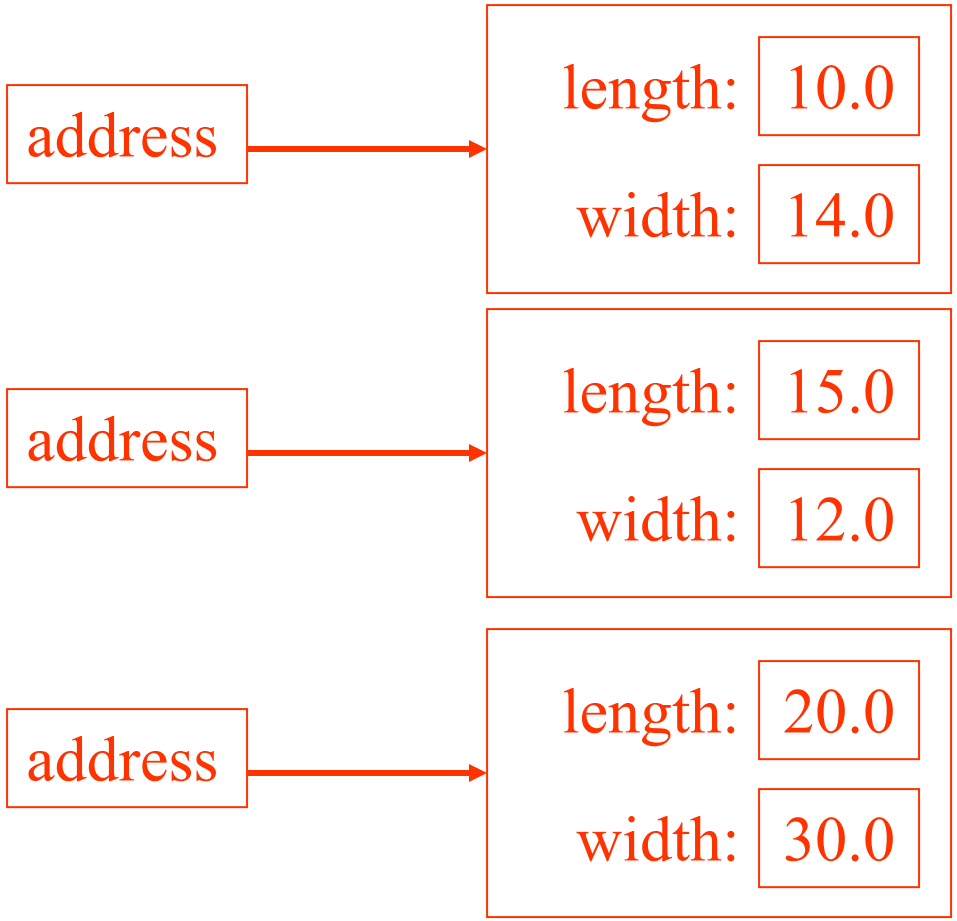
```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

States of Three Different Rectangle Objects

The `kitchen` variable holds the address of a `Rectangle` Object.

The `bedroom` variable holds the address of a `Rectangle` Object.

The `den` variable holds the address of a `Rectangle` Object.



Instance Methods x Static Methods

- Point to ponder #3:

  What is the difference between the two methods' headers below?

```
public void setLength (double len)
```

Belongs to the objects instantiated from the class. It performs an operation on instance fields.

```
public static void setLength (double len)
```

Belongs to the class. It performs an operation on static fields.

Constructors (1 of 3)

- Classes can have special methods called *constructors*.
- A constructor is a method that is automatically called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors (2 of 3)

- Point to ponder #4:

  Why are those special methods called constructors?

Because they help construct an object.

Constructors (3 of 3)

- Constructors have a few special properties that set them apart from normal methods.
  - Constructors have the same name as the class.
  - Constructors have no return type (not even `void`).
  - Constructors may not return any values.
  - Constructors are typically public.

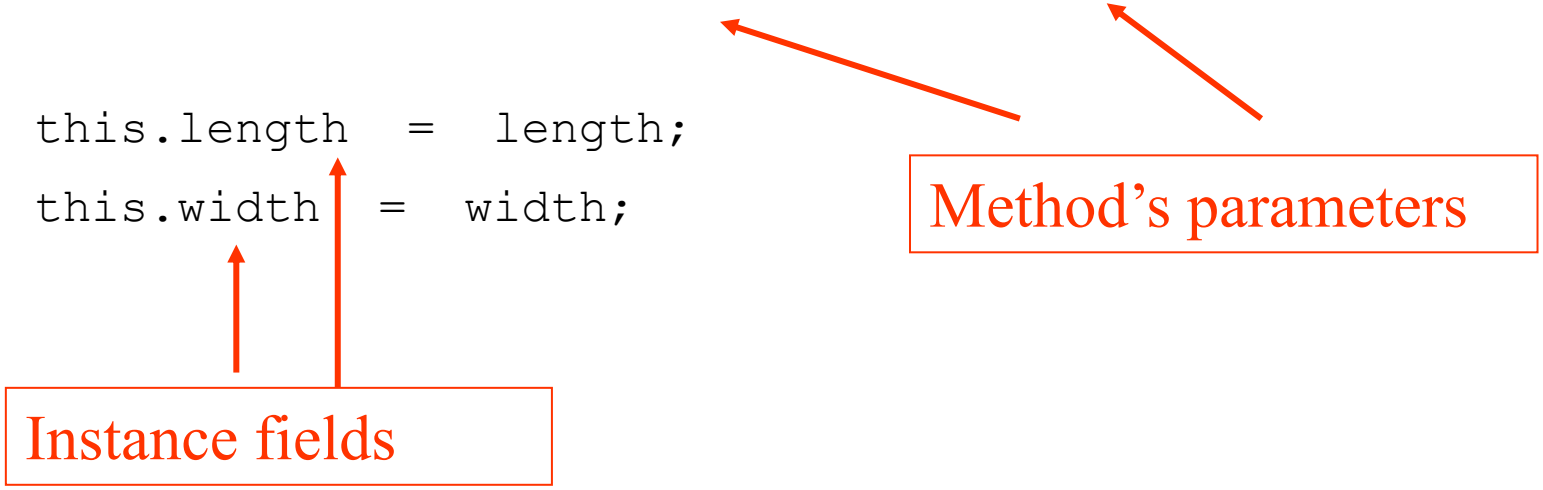
Constructor for `Rectangle` Class (1 of 3)

```
/**
   Constructor
   @param len The length of the rectangle.
   @param w The width of the rectangle.
*/
public Rectangle(double len, double w)
{
   length = len;
   width = w;
}
```

Examples: Rectangle.java, ConstructorDemo.java

Constructor for `Rectangle` Class (2 of 3)

```
public Rectangle(double length, double width)
   {
      this.length  =  length;
      this.width   =  width;
   }
```



- The this reference is simply a name of a reference variable that an object can use to refer to itself.
- The this reference contains the address of the calling object.
- The this reference can be used to allow a parameter to have the same name as an instance field.

Constructor for `Rectangle` Class (3 of 3)

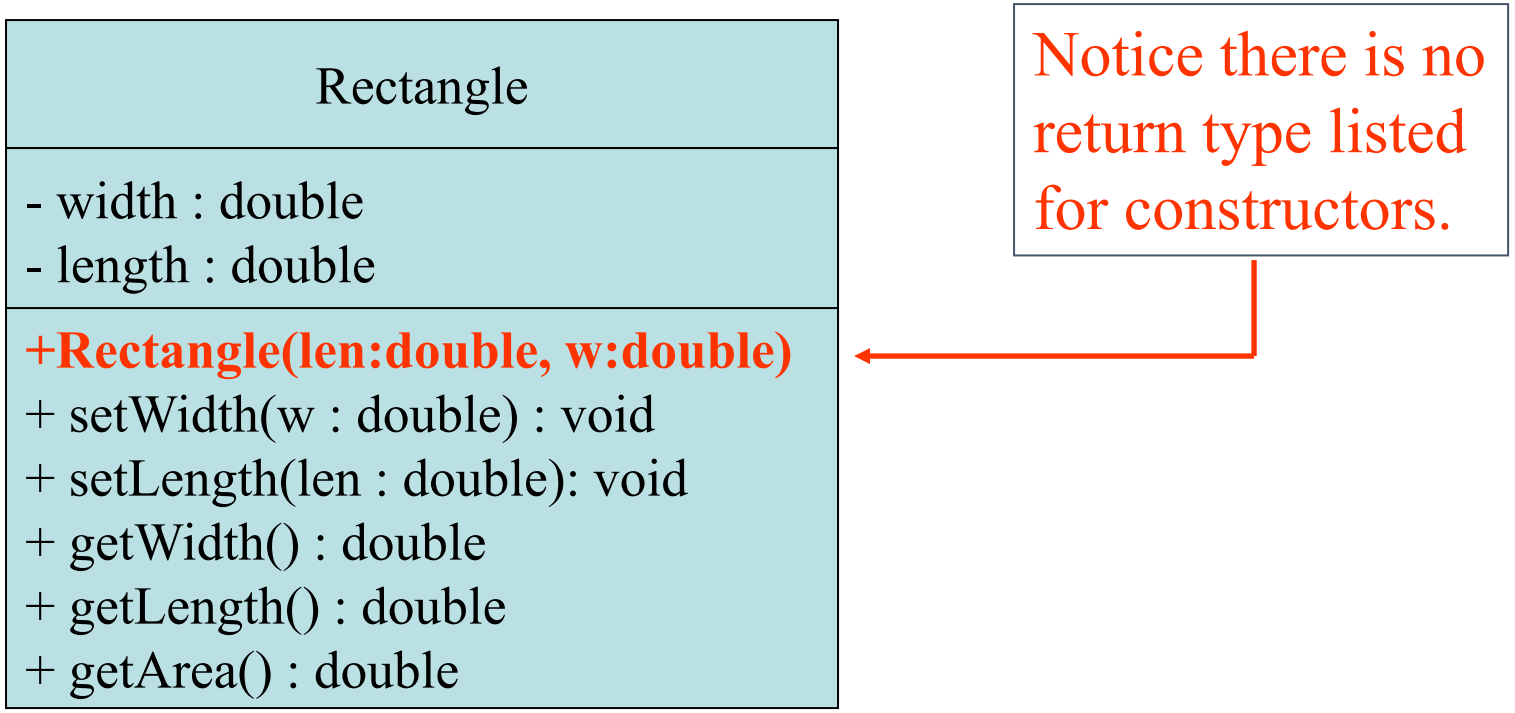
- Point to ponder #5:

  Why constructors do not have a return type, even void?

Because they are not executed by explicit method calls.

Constructors in UML

- In UML, the most common way constructors are defined is:



Uninitialized Local Reference Variables

- Reference variables can be declared without being initialized.

```
Rectangle box;
```

- This statement does not create a `Rectangle` object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

```
box = new Rectangle(7.0, 14.0);
```

- `box` will now reference a `Rectangle` object of length 7.0 and width 14.0.

The Default Constructor (1 of 2)

- When an object is created, its constructor is always called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
  - It sets all object's numeric fields to 0.
  - It sets all object's `boolean` fields to `false`.
  - It sets all object's reference variables to the special value *null*.

The Default Constructor (2 of 2)

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
- A default constructor is not provided by Java if a constructor is already written.

Writing Your Own No-Arg Constructor (1 of 2)

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
public Rectangle()
{
   length = 1.0;
   width = 1.0;
}
```

Writing Your Own No-Arg Constructor (2 of 2)

- Point to ponder #6:

  Where is the problem here?

```
Rectangle box = new Rectangle();
public Rectangle(double len, double w)
{
      length = len;
      width = w;
}
```

When we add our own constructor to the class it becomes the only constructor available. Java does not provide a default constructor.

Passing Objects as Arguments

- When you pass an object as an argument, the thing that is passed into the parameter variable is the object's memory address.
- As a result, parameter variable references the object, and the receiving method has access to the object.
- See DieArgument.java

Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method `add`

```
public int add(int num1, int num2)
{
  int sum = num1 + num2;
  return sum;
}

 public String add (String str1, String str2)
{
  String combined = str1 + str2;
  return combined;
}
```

Method Signature and Binding (1 of 2)

- A method signature consists of the method's name and the data types of the method's parameters, in the order that they appear. The return type is not part of the signature.

```
add(int, int)
add(String, String)
```

Signatures of the `add` *methods of previous slide*

- The process of matching a method call with the correct method is known as *binding*. The compiler uses the method signature to determine which version of the overloaded method to bind the call to.
- Constructors can also be overloaded, which means that a class can have more than one constructor.

Method Signature and Binding (2 of 2)

- Point to ponder #1:

  Which Java commands are valid or invalid?

```
add(3,3);
```


```
add("Hi", "3");
```


```
add(3, "3");
```


```
add(3);
```


```
add(3.0,3);
```


`Rectangle` Class Constructor Overload (1 of 2)

- Point to ponder #2:

  What are the values of length and width after box1 and box2 objects are built?

```
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```

```
public class Rectangle
{
   private double width;
   private double length;

   public Rectangle ()
      width = 1;
      length = 2;
   }

}
```

width = 1.0
length = 2.0

```
public class Rectangle
{
   private double width;
   private double length;

   public Rectangle (double width,
                                double length)
     this.width = 1 + width;
     this.length = 2 + length;
   }

}
```

width = 6.0
length = 12.0

`Rectangle` Class Constructor Overload (1 of 2)

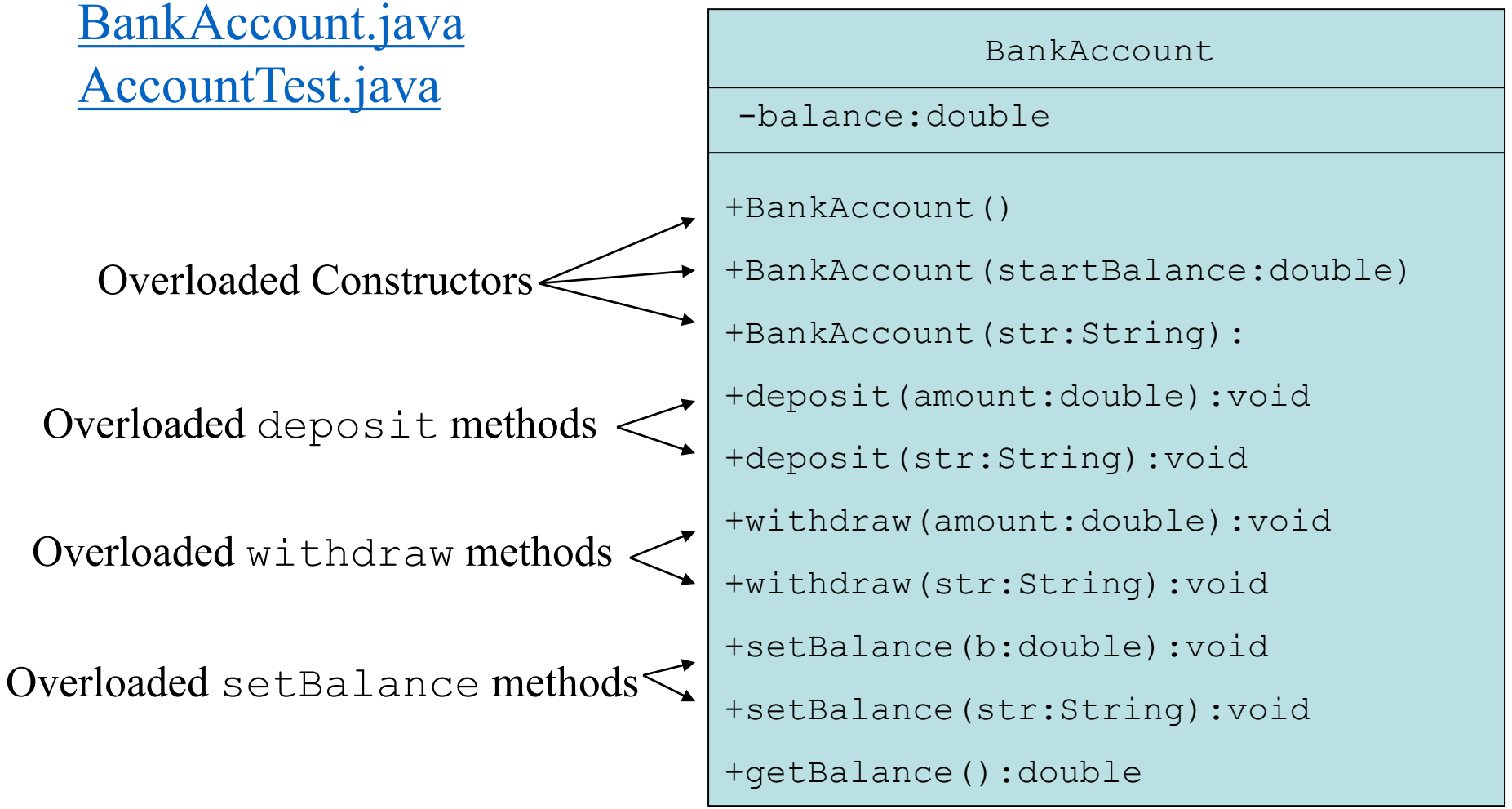
- Point to ponder #3:

  A default constructor is the same thing of a no-arg constructor?

  No, the default constructor does not include any implementation in its body. It automatically sets all object's numeric fields to 0, all object's boolean fields to false, and all object's reference variables to the special value null.

The `BankAccount` Example (1 of 2)

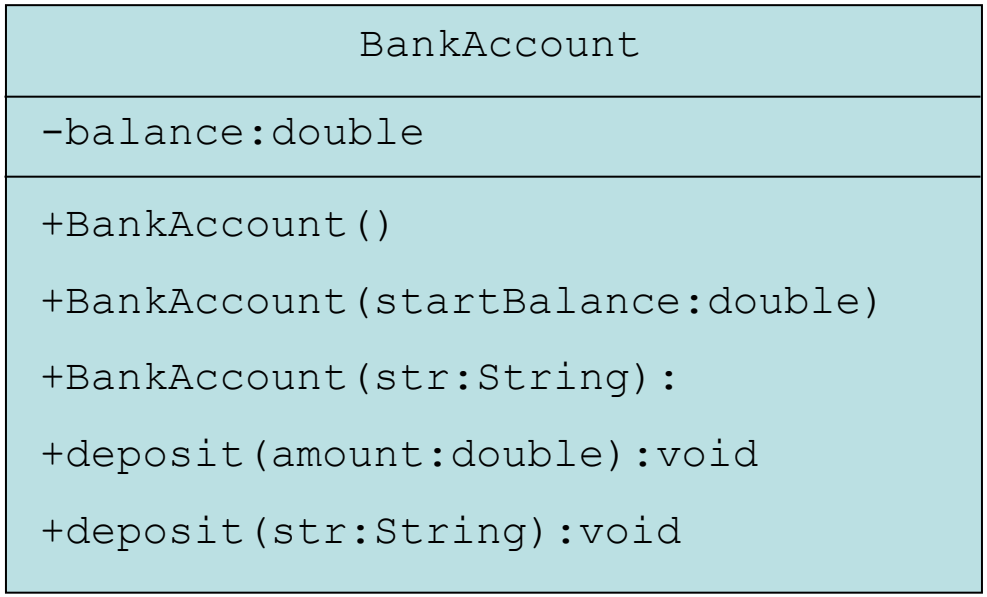
BankAccount.java
AccountTest.java



The `BankAccount` Example (2 of 2)

- Point to ponder #4:

  Why it is a good idea to overload methods?



To add flexibility to your class (API). The callers will have multiple ways to create objects or execute actions.

Scope of Instance Fields

- Variables declared as instance fields in a class can be accessed by any instance method in the same class as the field.
- If an instance field is declared with the `public` access specifier, it can also be accessed by code outside the class

❑ Point to ponder #5:

Why this approach should be avoided?

Data Hiding OO concept!

Shadowing

- A parameter variable is, in effect, a local variable.
- Within a method, variable names must be unique.
- A method may have a local variable with the same name as an instance field.
- This is called *shadowing*.
- The local variable will *hide* the value of the instance field.
- Shadowing is discouraged and local variable names should not be the same as instance field names or the `this` reference should be used.

Packages and `import` Statements

- Classes in the Java API are organized into *packages* (group of related classes).
- Explicit and Wildcard `import` statements
  - Explicit imports name a **specific** class
    - `import java.util.Scanner;`
    - `import java.util.Random;`
  - Wildcard imports name a package, followed by an `*`
    - `Import java.util.*;`
- The `java.lang` package is automatically made available to any Java class (e.g., `System`, `String`).

Some Java Standard Packages

Package	Description
java.io	Provides classes that perform various types of input and output.
java.lang	Provides general classes for the Java language. This package is automatically imported.
java.net	Provides classes for network communications.
java.security	Provides classes that implement security features.
java.sql	Provides classes for accessing databases using structured query language.
java.text	Provides various classes for formatting text.
java.util	Provides various utility classes.

The `toString` Method (1 of 3)

- The `toString` method of a class returns a string representing the state of an object (object's data).
- It can be called *explicitly*:

```
Rectangle box1 = new Rectangle (2.0, 3.0);
System.out.println(box1.toString());
```

- Or implicitly, whenever you pass an object of the class to `println` or `print`.

```
Rectangle box1 = new Rectangle 2.0, 3.0);
System.out.println(box1);
```

The `toString` Method (2 of 3)

- The `toString` method is also called implicitly whenever you concatenate an object of the class with a string.

```
Rectangle box1 = new Rectangle (2.0, 3.0);
System.out.println("The rectangle data is:\n" + box1);
```

The `toString` Method (3 of 3)

- All objects have a `toString` method that returns the class name and a hash of the memory address of the object.
- We can override the default method with our own to print out more useful information.

```
public String toString() {
   return String.format("Length: %.2f, Width is: %.2f",
                        length, width);
}
```

Stock.java
StockDemo1.java

CalPolyPomona

Classes

Lecture 6c

Topics

- Objects and Classes
- Writing a Simple Class, Step by Step
- Instance Fields and Methods
- Constructors
- The “this” Reference Variable
- Passing Objects as Arguments
- Overloading Methods and Constructors
- Scope of Instance Fields
- Static Class Members
- Writing an equals Method
- Methods that Copy Objects
- Aggregation
- Garbage Collection

Review: Instance Fields and Methods (1 of 2)

- Each instance of a class has its own copy of instance variables.
  - Example:
    - The `Rectangle` class defines a `length` and a `width` field.
    - Each instance of the `Rectangle` class can have different values stored in its `length` and `width` fields.
- Instance methods require that an instance of a class be created in order to be used.
- Instance methods typically interact with instance fields or calculate values based on those fields.

Review: Instance Fields and Methods (1 of 2)

- Point to ponder #1:

  Can instance fields and methods be used before an instance of the class is created?

  No, only after an object (instance of the class) is created.

Static Class Members (1 of 2)

- *Static fields* and *static methods* do not belong to a single instance of a class.
- In fact, an instance of the class doesn't even have to exist for values to be stored in the class's static fields.
- Likewise, static methods do not operate on the fields that belong to any instance of the class. Instead, they can operate only on static fields.

Static Class Members (2 of 2)

- To invoke a static method or use a static field, the class name, rather than the instance name, can be used.
- Example:

```
double val = Math.sqrt(25.0);
```

Static Fields (1 of 2)

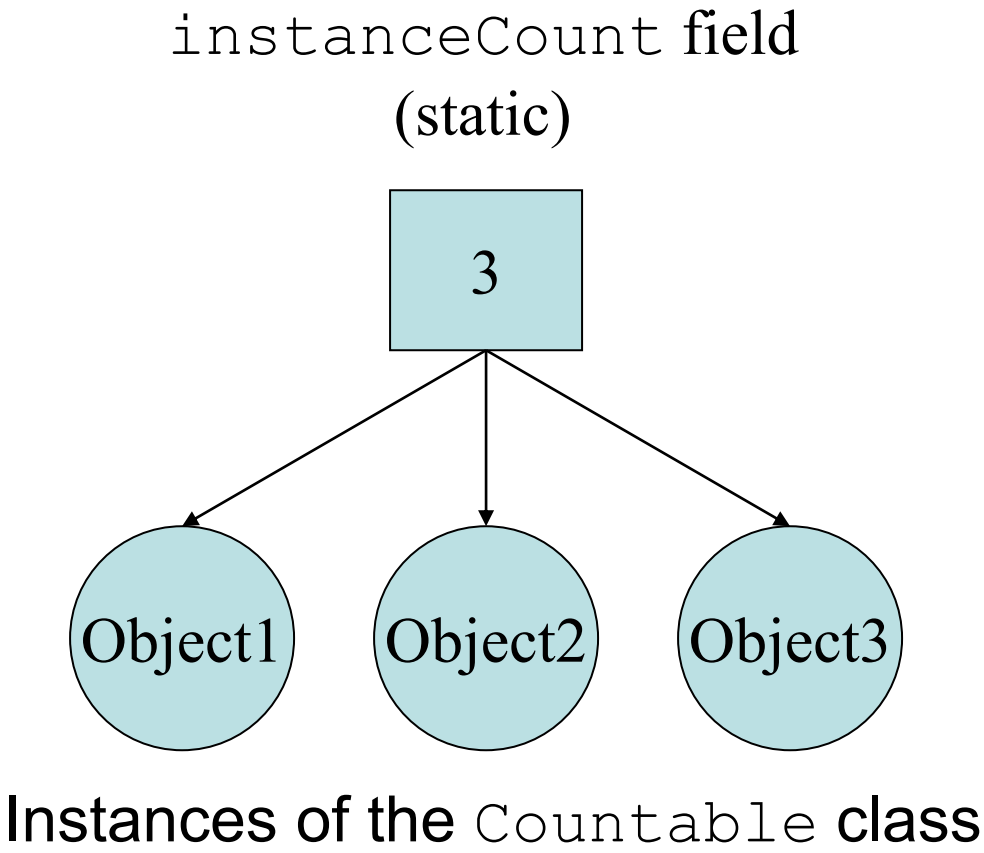
- Class fields are declared using the `static` keyword between the access specifier and the field type.

```
private static int instanceCount = 0;
```

- The field is initialized to 0 only once, regardless of the number of times the class is instantiated (a single copy of a class's static field is shared by all instances of the class)
  - Primitive static fields are initialized to 0 if no initialization is performed.

- Examples: Countable.java, StaticDemo.java

Static Fields (2 of 2)



Static Methods (1 of 2)

- Methods can also be declared static by placing the `static` keyword between the access specifier and the return type of the method.

```
public static double milesToKilometers(double miles)
                                        {…}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

```
double kilosPerMile = Metric.milesToKilometers(1.0);
```

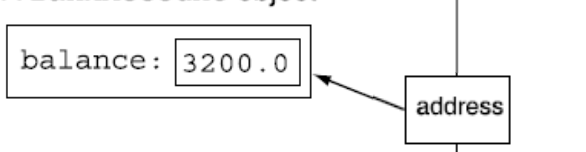
- Examples: Metric.java, MetricDemo.java

Static Methods (2 of 2)

- Static methods are convenient because they may be called at the class level.
- They are typically used to create **utility classes**, such as the `Math` class in the Java Standard Library and have no need to collect and store data.
- Static methods cannot refer to non-static members of the class. This means that any method called from a static method must also be static. It also means that if the method uses any of the class's fields, they must be static as well. Static methods DO NOT communicate with instance fields, only static fields.

Returning Objects from Methods

- As methods can be written to return an int, double, or float, they can also be written to return a reference to an object.




```
BankAccount account;
account = getAccount(3200.0);
. . .

public static BankAccount getAccount(double
                                     balance)
{
  // Create a BankAccount object and return
  // a reference to it.
  return new BankAccount(balance);
}
```

The `equals` Method (1 of 3)

- When the == operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.
- All objects have an `equals` method.
- The default operation of the `equals` method is to compare memory addresses of the objects (just like the == operator).

The `equals` Method (2 of 3)

- The `Stock` class has an `equals` method.
- If we try the following:

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = new Stock("GMX", 55.3);
if (stock1 == stock2) // This is a mistake.
   System.out.println("The objects are the same.");
else
   System.out.println("The objects are not the same.");
```

only the addresses of the objects are compared.

The `equals` Method (3 of 3)

- Instead of using the `==` operator to compare two `Stock` objects, we should use the `equals` method.

```
public boolean equals(Stock object2)
{
   boolean status;

   if(symbol.equals(Object2.symbol) && sharePrice == Object2.sharePrice)
      status = true;
   else
      status = false;
   return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.
- See example: StockCompare.java

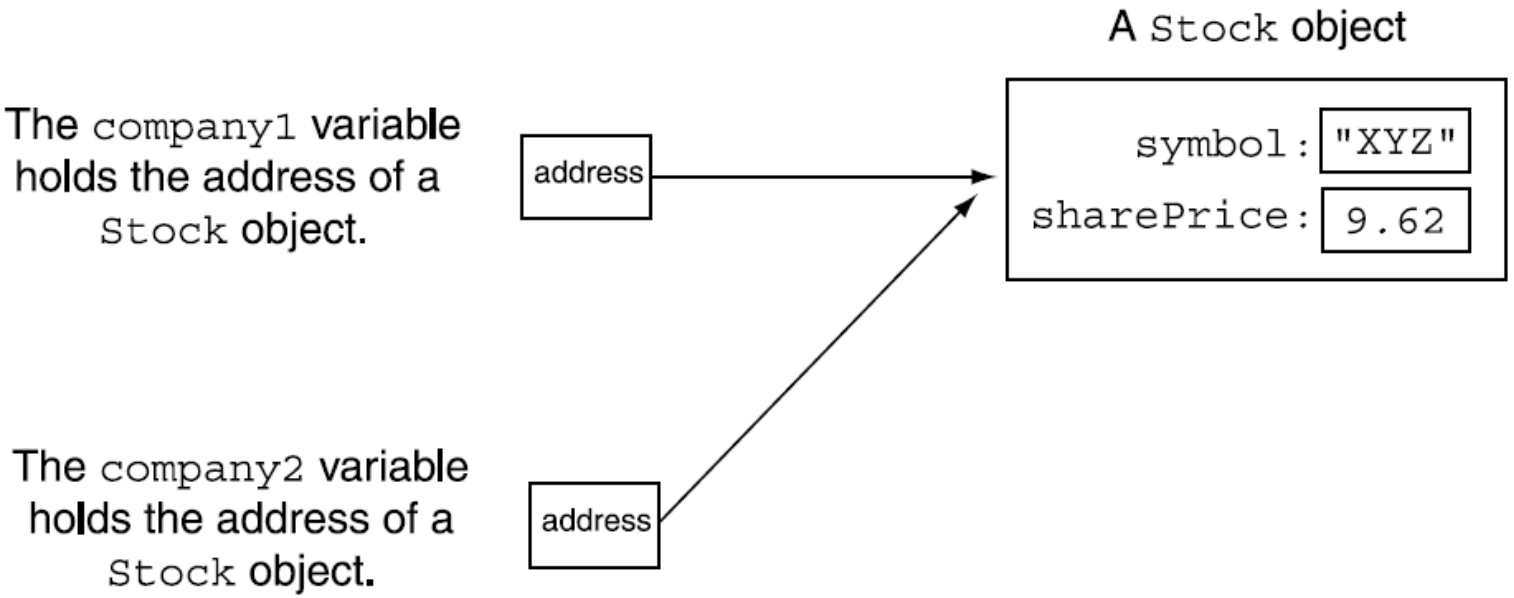
Methods that Copy Objects (1 of 2)

- There are two ways to copy an object.
  - You cannot use the assignment operator to copy reference types
  - Reference only copy
    - This is simply copying the address of an object into another reference variable.
  - Deep copy (correct)
    - This involves creating a new instance of the class and copying the values from one object into the new object.
  - Example: ObjectCopy.java

Methods that Copy Objects (2 of 2)

```
Stock company1 = new Stock("XYZ", 9.62);
Stock company2 = company1;
```

reference copy



Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object2)
{
   symbol = object2.symbol;
   sharePrice = object2.sharePrice;
}

// Create a Stock object
Stock company1 = new Stock("XYZ", 9.62);
//Create company2, a copy of company1
Stock company2 = new Stock(company1);
```

Null References

- A *null reference* is a reference variable that points to nothing.
- If a reference is null, then no operations can be performed on it.
- References can be tested to see if they point to null prior to being used.

```
if(name != null)
{
  System.out.println("Name is: " + name.toUpperCase());
}
```

- Examples: FullName.java, NameTester.java

The `this` Reference (1 of 2)

- The `this` reference is simply a name that an object can use to refer to itself.
- The `this` reference can be used to overcome shadowing and allow a parameter to have the same name as an instance field.

```
public void setFeet(int feet)
{
   this.feet = feet;
   //sets the this instance's feet field
   //equal to the parameter feet.
}
```

Local parameter variable feet

Shadowed instance variable

The `this` Reference (2 of 2)

- The `this` reference can be used to call a constructor from another constructor.

```
public Stock(String sym)
{
  this(sym, 0.0);
}
```

  - This constructor would allow an instance of the `Stock` class to be created using only the symbol name as a parameter.
  - It calls the constructor that takes the symbol and the price, using *sym* as the symbol argument and 0 as the price argument.
- Elaborate constructor chaining can be created using this technique.
- If `this` is used in a constructor, it must be the first statement in the constructor.

Garbage Collection (1 of 6)

- When objects are no longer needed, they should be destroyed.
- This frees up the memory that they consumed.
- Java handles all the memory operations for you.
- Simply set the reference to *null* and Java will reclaim the memory.

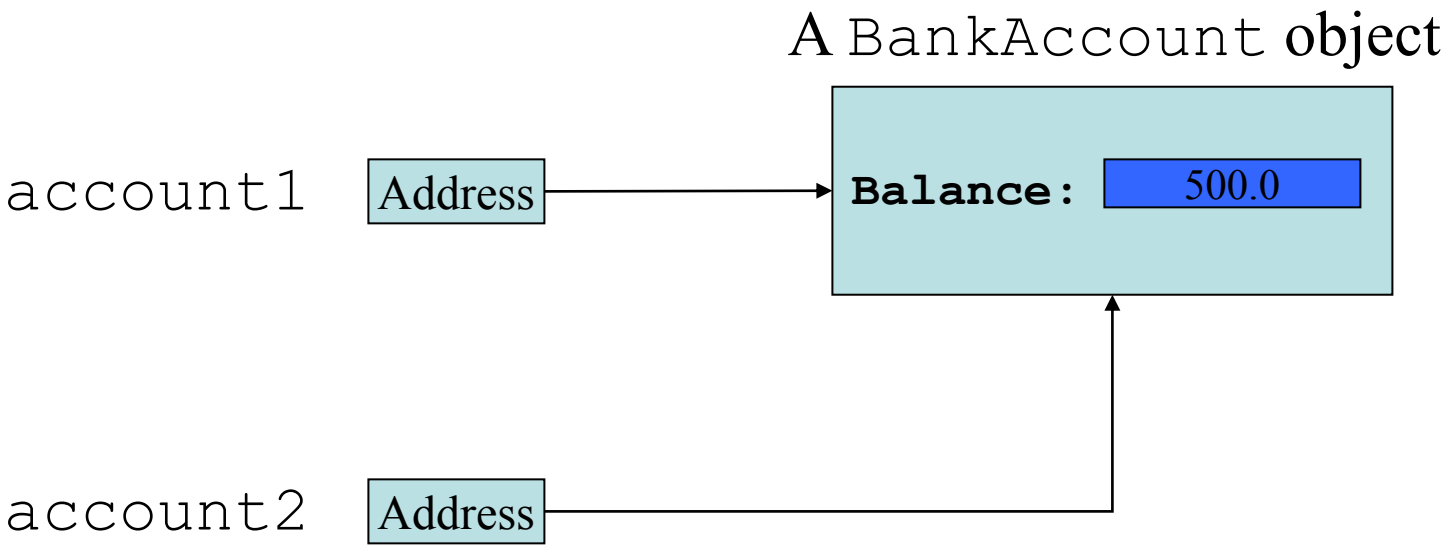
Garbage Collection (2 of 6)

- The Java Virtual Machine has a process that runs in the background that reclaims memory from released objects.
- The *garbage collector* will reclaim memory from any object that no longer has a valid reference pointing to it.

```
BankAccount account1 = new BankAccount(500.0);
BankAccount account2 = account1;
```

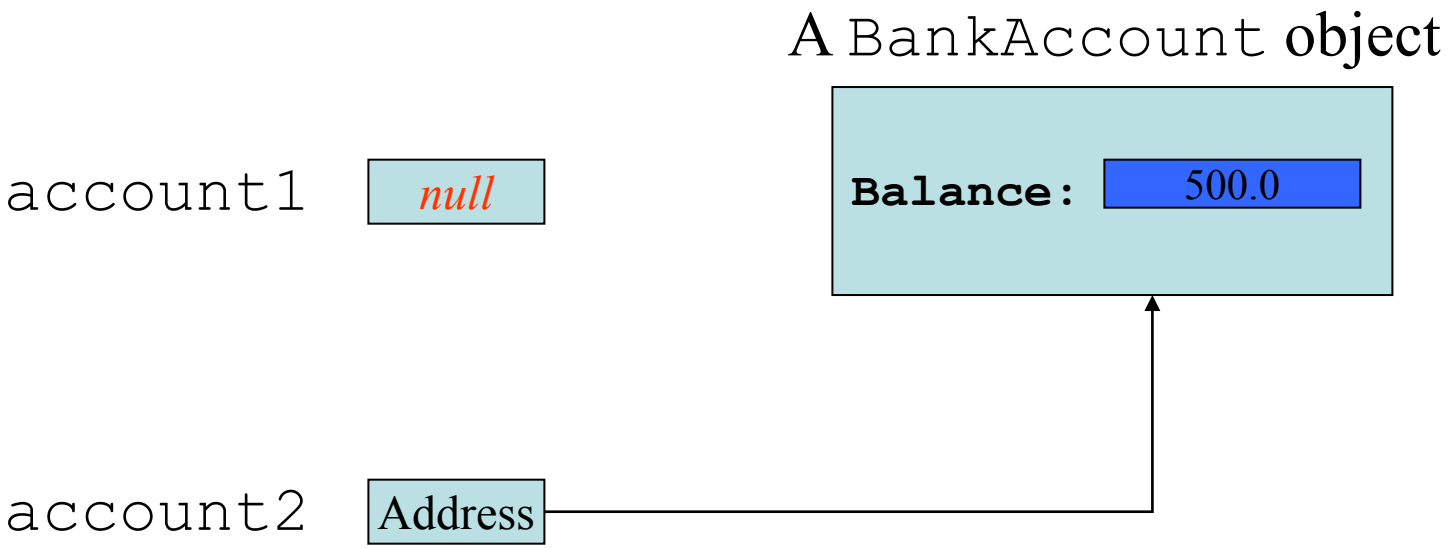
- This sets `account1` and `account2` to point to the same object.

Garbage Collection (3 of 6)



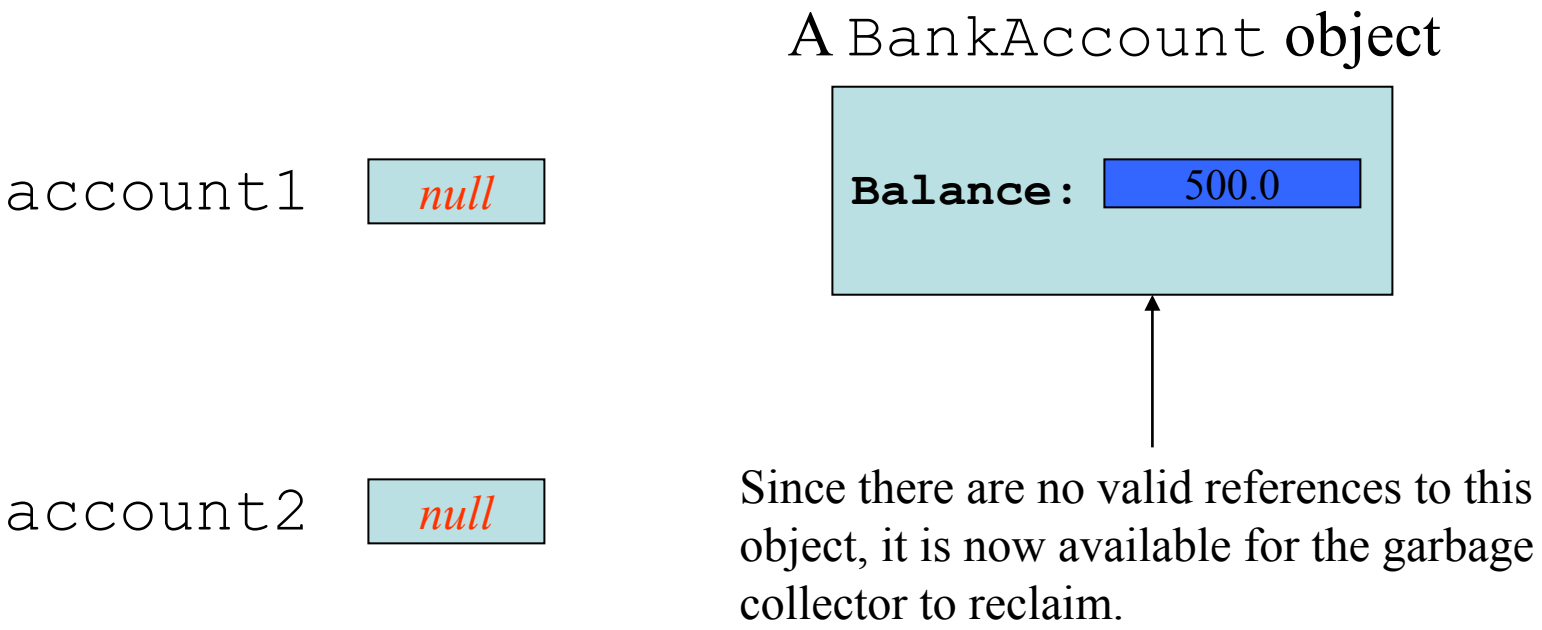
Here, both `account1` and `account2` point to the same instance of the `BankAccount` class.

Garbage Collection (4 of 6)



However, by running the statement: **`account1 = null;`**
only `account2` will be pointing to the object.

Garbage Collection (5 of 6)

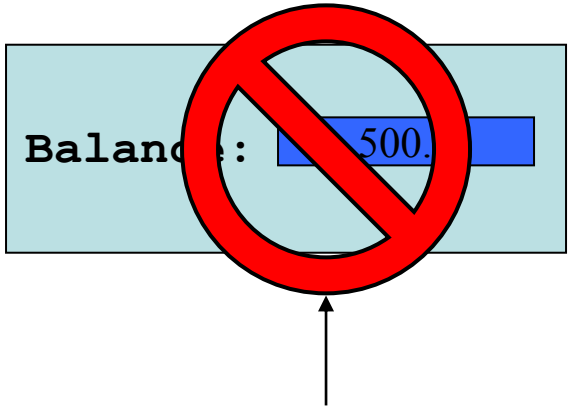


If we now run the statement: **`account2 = null;`**
neither `account1` or `account2` will be pointing to the object.

Garbage Collection (6 of 6)

`account1` *null*

`account2` *null*



The garbage collector reclaims the memory the next time it runs in the background.

The `finalize` Method

- If a method with the signature:

  ```
  public void finalize(){…}
  ```

  is included in a class, it will run just prior to the garbage collector reclaiming its memory.
- The garbage collector is a background thread that runs periodically.
- It cannot be determined when the `finalize` method will actually be run.

Aggregation

- Making an instance of one class a field in another class is called *object aggregation*.
- The word aggregate means “a whole which is made of constituent parts.”
- Aggregation creates a “has a” relationship between objects.
- Examples:
  - Instructor.java, Textbook.java, Course.java, CourseDemo.java

Aggregation in UML Diagrams