CHAPTER 3 – OBJECTS AND CLASSES

CENG522

Dr. Serdar ARSLAN

Slides adapted from Starting Out with Java book slides

Objects and Classes

- An object exists in memory, and performs a specific task.
- 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

- You may have already used some objects:
  - `Scanner` objects, for reading input
  - `Random` objects, for generating random numbers
  - `PrintWriter` objects, for writing data to files
- 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

- 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.

Objects and Classes

- 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.

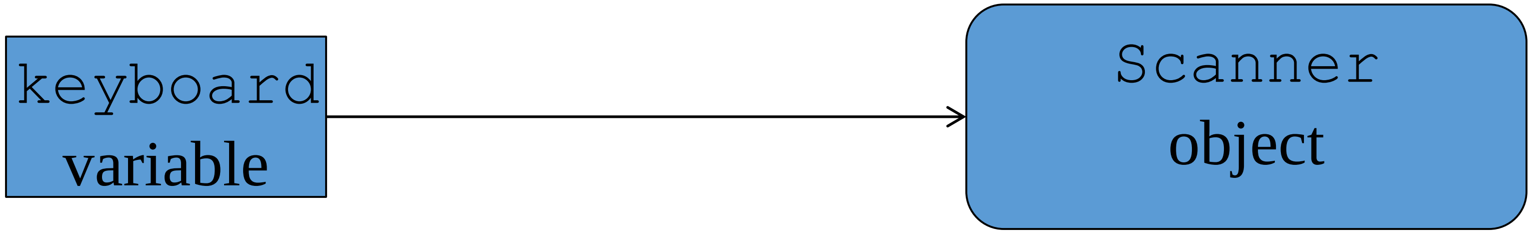
Objects and Classes

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.



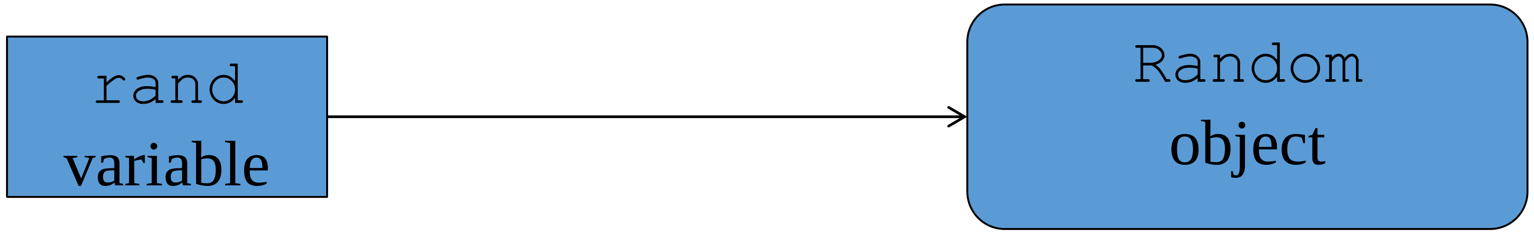
Objects and Classes

Example:

This expression creates a `Random` object in memory.

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

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



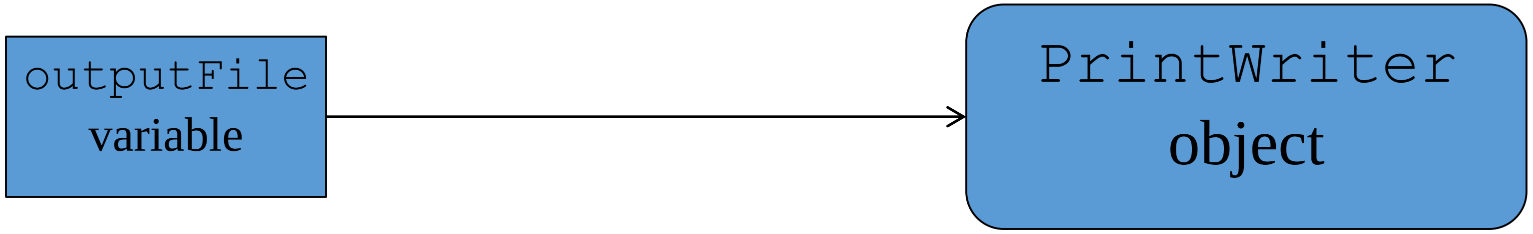
Objects and Classes

Example:

This expression creates a `PrintWriter` object in memory.

```
PrintWriter outputFile = new PrintWriter("numbers.txt");
```

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



Objects and Classes

- The Java API provides many classes
    - `Scanner`
    - `Random`
    - `PrintWriter`
    - `String`
    - `Double`
    - `ArrayList`
    - `StreamReader`
    - `Etc…`

Writing a Class, Step by Step

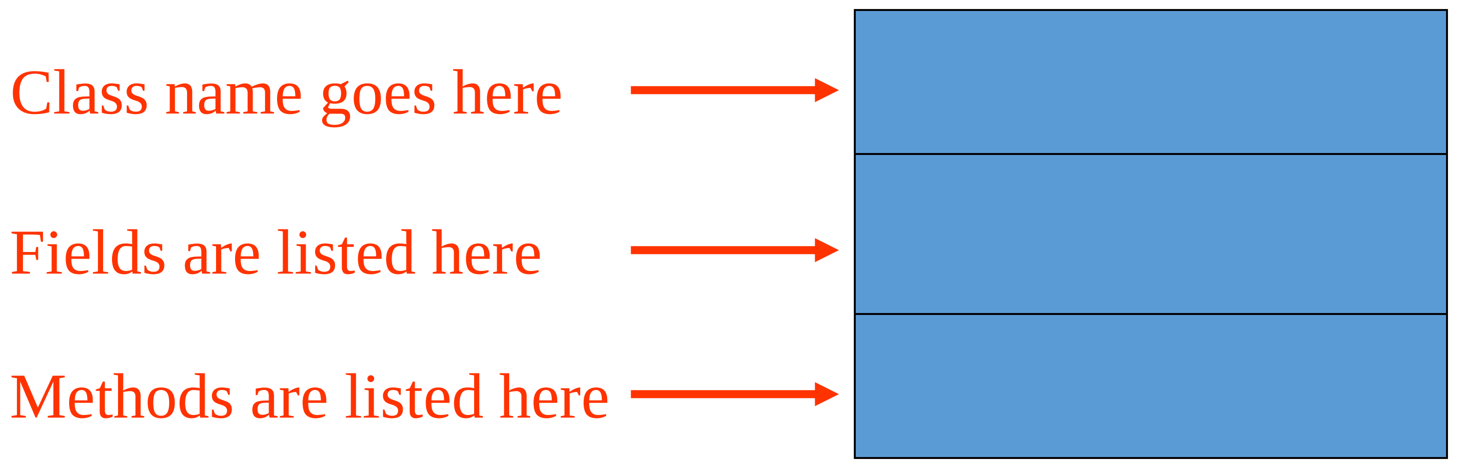
- A `Rectangle` object will have the following fields:
  - `length`. The length field will hold the rectangle’s length.
  - `width`. The width field will hold the rectangle’s width.

Writing a Class, Step by Step

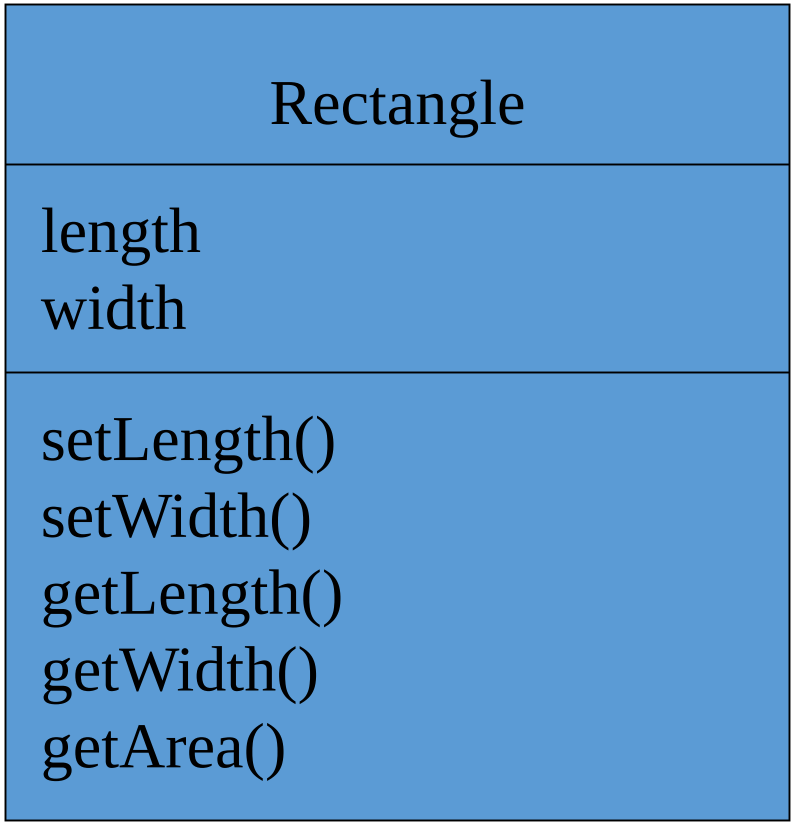
- The `Rectangle` class will also have the following methods:
  - **`setLength`**. The `setLength` method will store a value in an object's `length` field.
  - **`setWidth`**. The `setWidth` method will store a value in an object's `width` field.
  - **`getLength`**. The `getLength` method will return 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;
}
```

Access Specifiers

- An access specifier is a Java keyword that indicates how a field or method 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.

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;
 }
```

Header for the `setLength` Method

Return Type

Access specifier

Method Name

Notice the word **`static`** does not appear in the method header designed to work on an instance of a class (*instance method*).

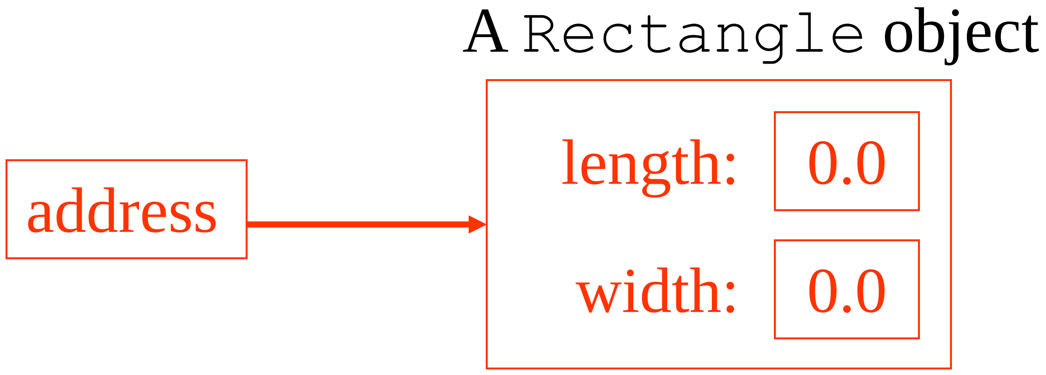
```
public void setLength (double len)
```

Parameter variable declaration

Creating a `Rectangle` object

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

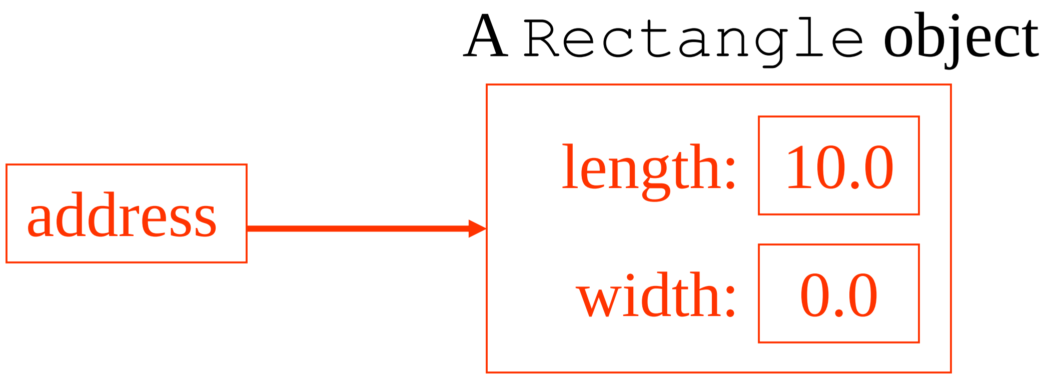
The `box` variable holds the address of the Rectangle object.



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;
 }
```

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

- 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

- 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

- 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

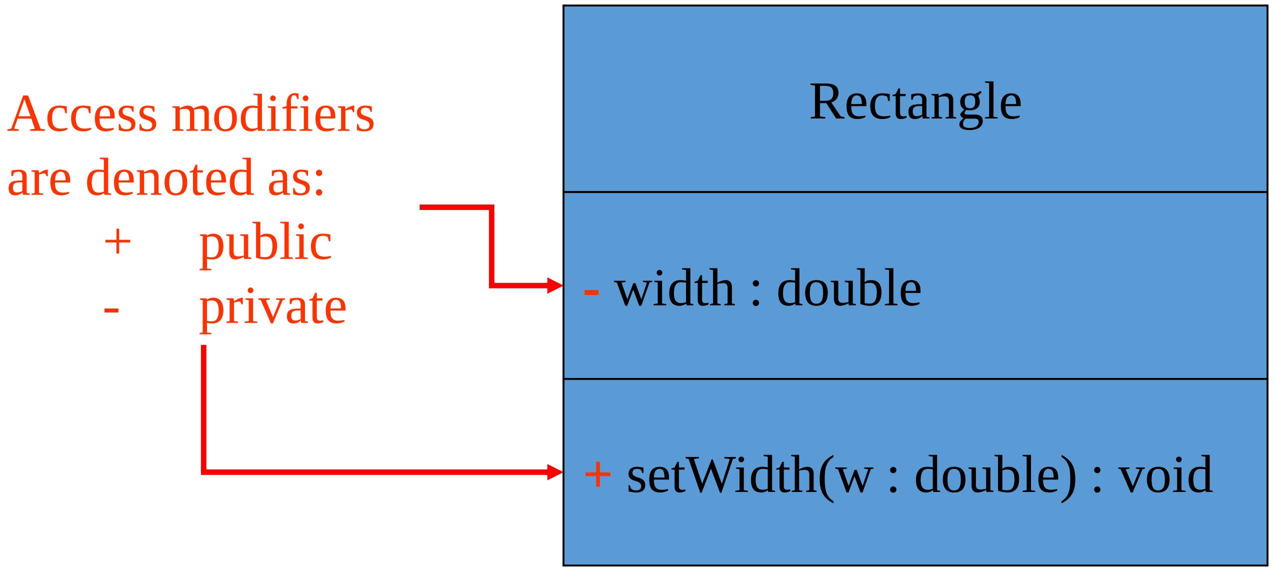
- Rather than use an `area` variable in a `Rectangle` class:

```
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.

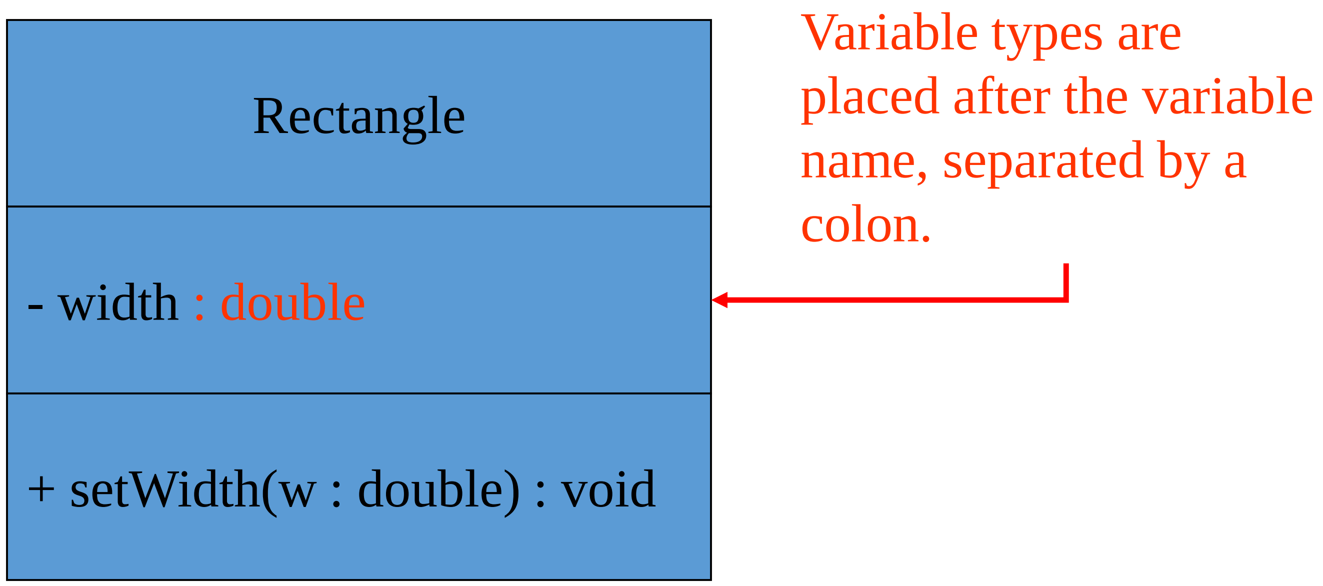
UML Data Type and Parameter Notation

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


Access modifiers are denoted as:
+ public
- private
Rectangle
- width : double
+ setWidth(w : double) : void

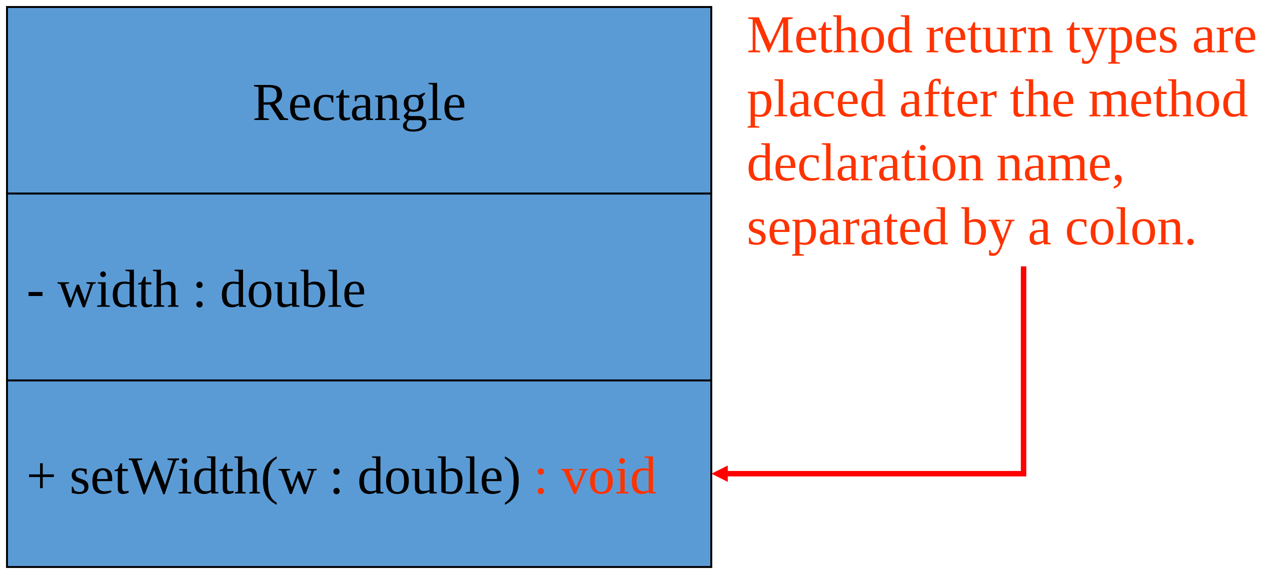
UML Data Type and Parameter Notation

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


Rectangle
- width : double
+ setWidth(w : double) : void
Variable types are placed after the variable name, separated by a colon.

UML Data Type and Parameter Notation

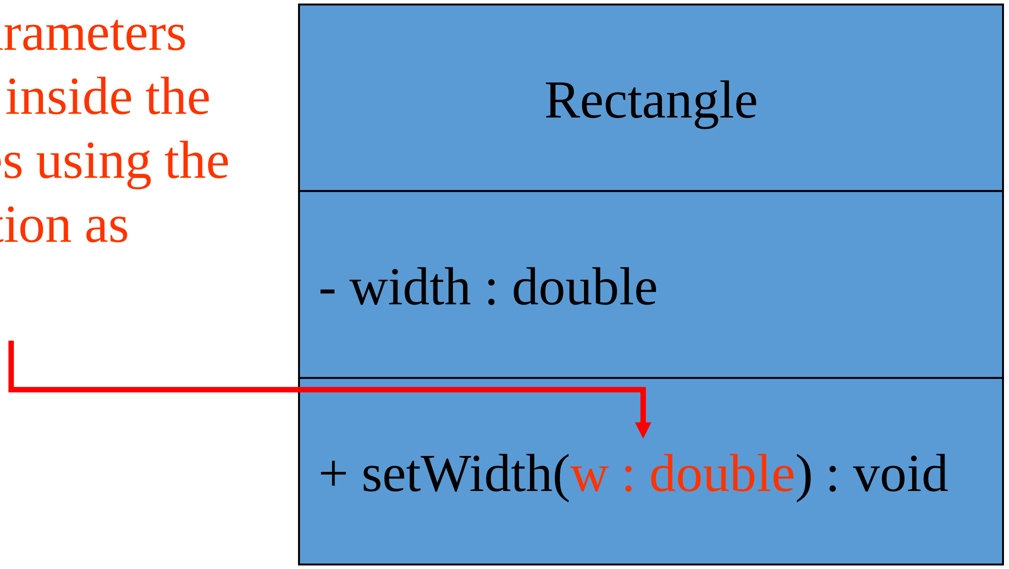
- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.


Rectangle
- width : double
+ setWidth(w : double) : void
Method return types are placed after the method declaration name, separated by a colon.

UML Data Type and Parameter Notation

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

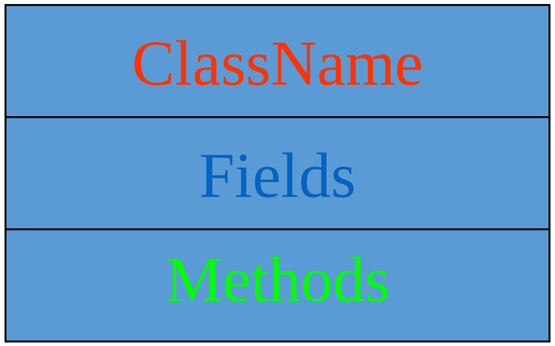
Method parameters are shown inside the parentheses using the same notation as variables.



Converting the UML Diagram to Code

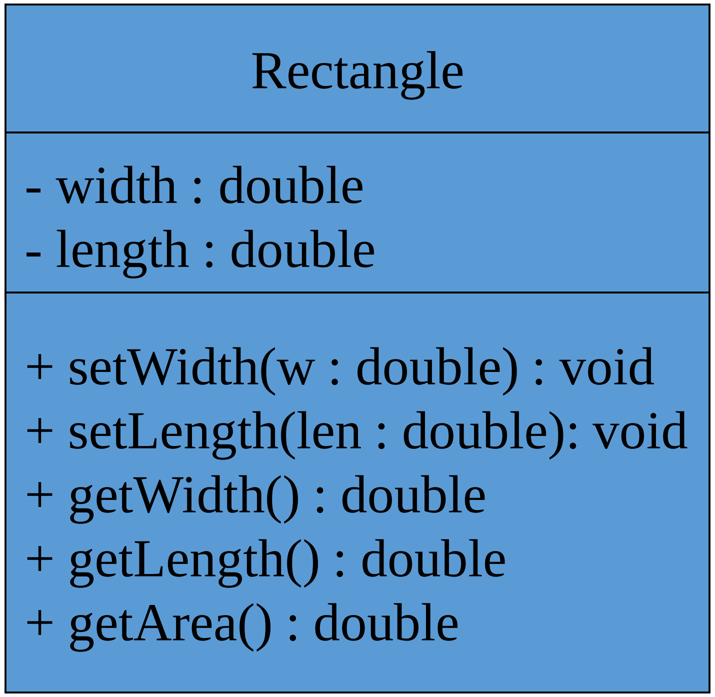
- Putting all of this information together, a Java class file can be built easily using the UML diagram.
- The UML diagram parts match the Java class file structure.

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



Converting the UML Diagram to Code

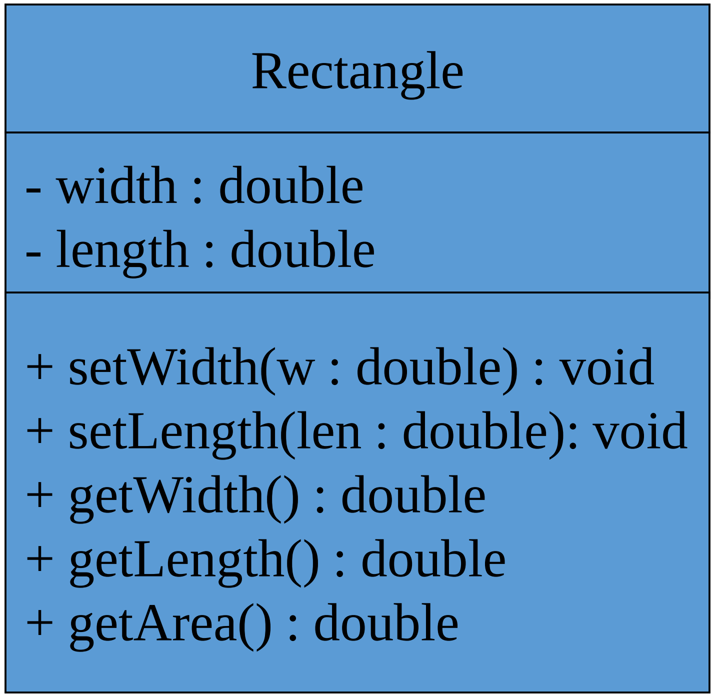
The structure of the class can be compiled and tested without having bodies for the methods. Just be sure to put in dummy return values for methods that have a return type other than void.



```
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

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

- 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

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

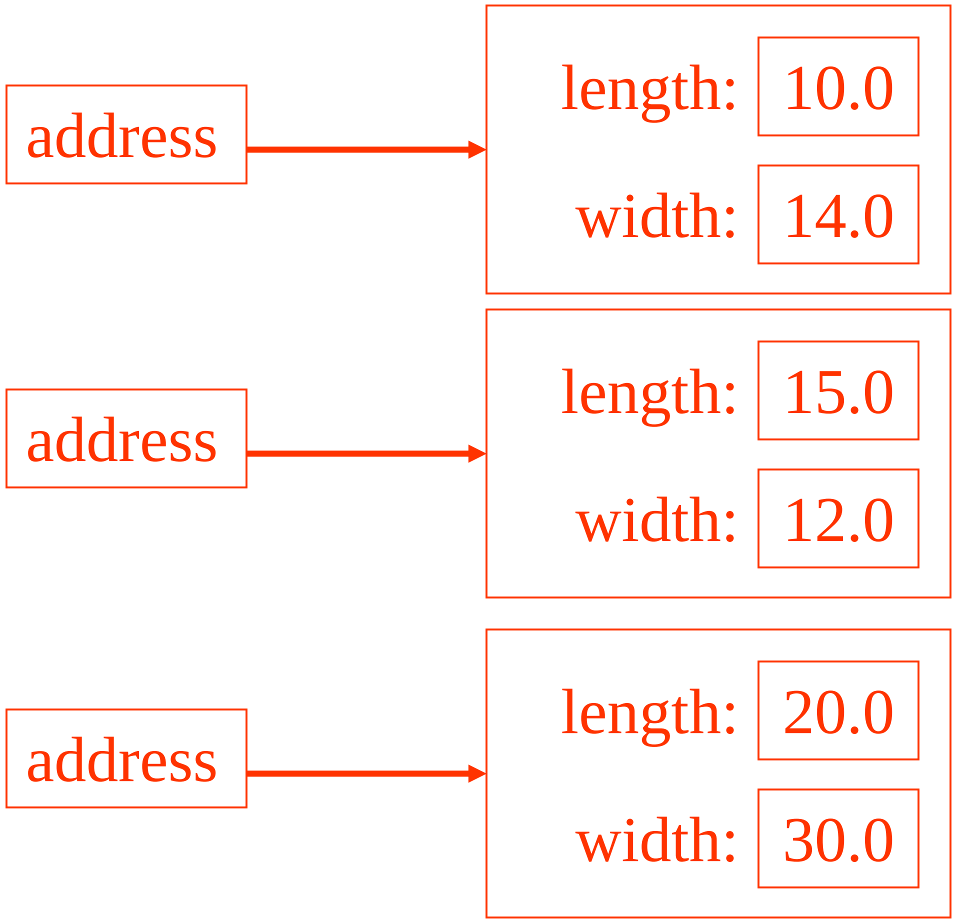
```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

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.



Constructors

- 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

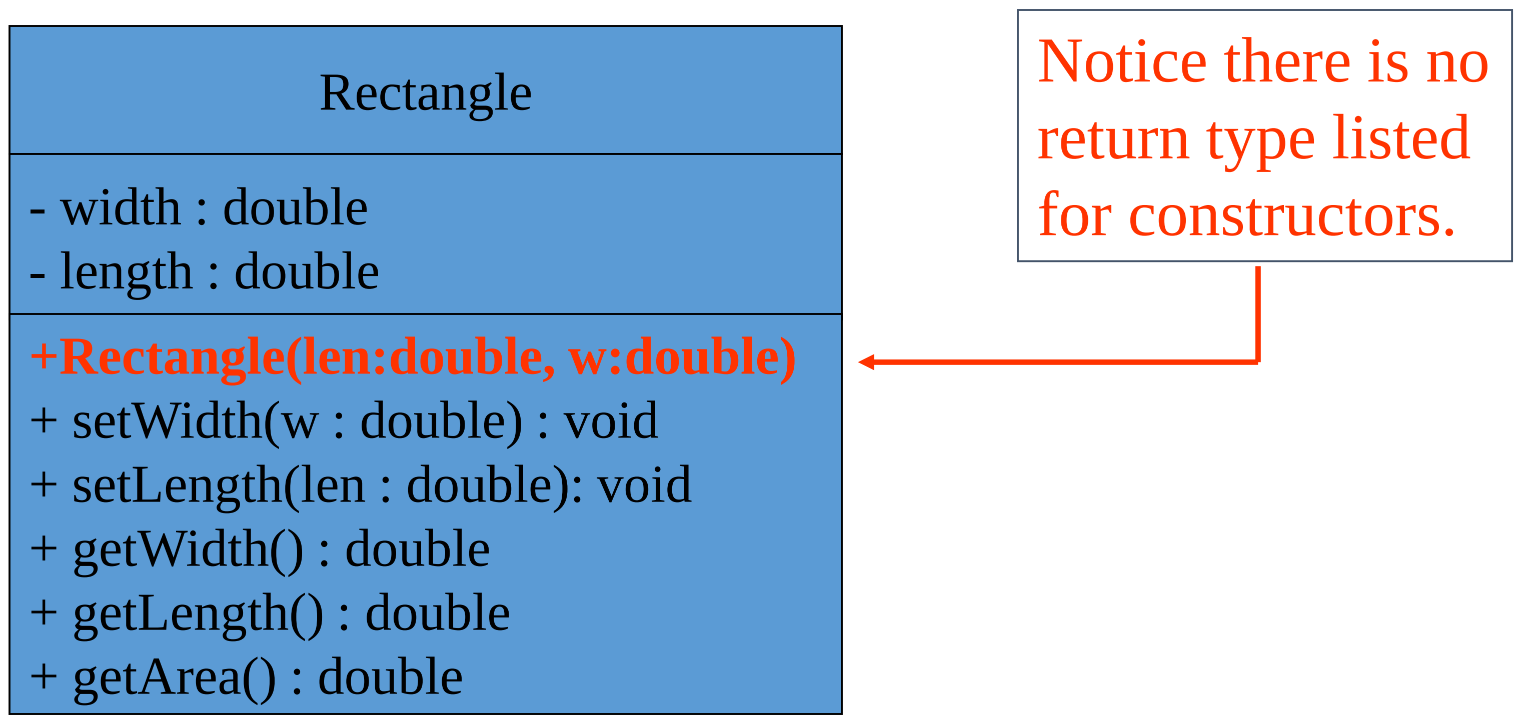
- Constructors have a few special properties that set them apart from normal methods.
  - Constructors have the same name as the class.
  - Constructors have no return type (not even `void`).
  - Constructors may not return any values.
  - Constructors are typically public.

Constructor for `Rectangle` Class

```
/**
   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;
}
```

Constructors in UML

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


Rectangle
- width : double
- length : double
+Rectangle(len:double, w:double)
+ setWidth(w : double) : void
+ setLength(len : double): void
+ getWidth() : double
+ getLength() : double
+ getArea() : double
Notice there is no return type listed for constructors.

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

- 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 of the object’s numeric fields to 0.
  - It sets all of the object’s `boolean` fields to `false`.
  - It sets all of the object’s reference variables to the special value *null*.

The Default Constructor

- 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

- 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;
}
```

The `String` Class Constructor

- One of the `String` class constructors accepts a string literal as an argument.
- This string literal is used to initialize a `String` object.
- For instance:

```
String name = new String("Michael Long");
```

The `String` Class Constructor

- This creates a new reference variable *name* that points to a `String` object that represents the name "Michael Long"
- Because they are used so often, `String` objects can be created with a shorthand:

```
String name = "Michael Long";
```

Passing Objects as Arguments

- When you pass a 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.

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

- 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 <u>not</u> 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.

`Rectangle` Class Constructor Overload

If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

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

`Rectangle` Class Constructor Overload

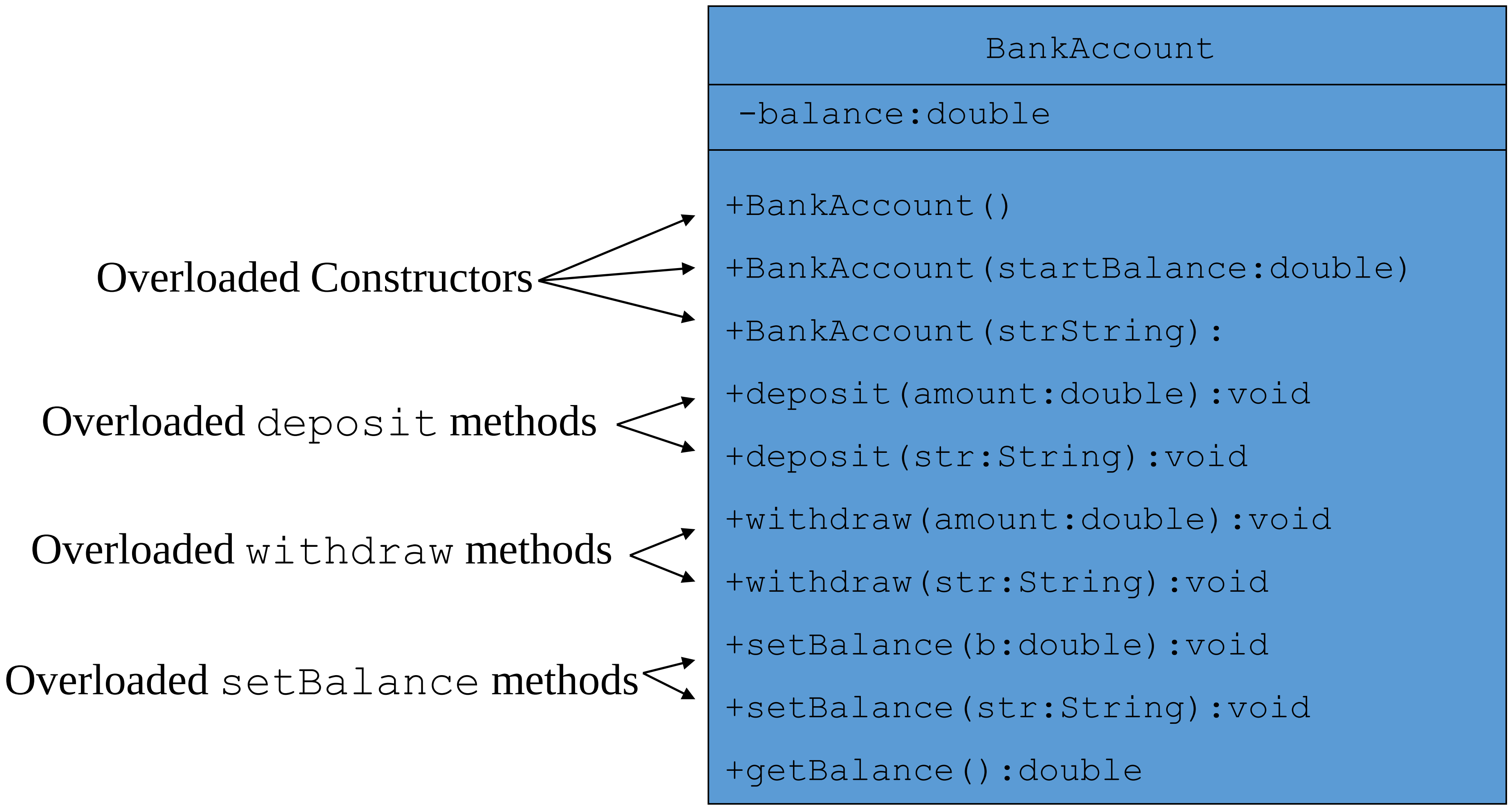
If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

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

The first call would use the no-arg constructor and `box1` would have a length of 1.0 and width of 1.0.

The second call would use the original constructor and `box2` would have a length of 5.0 and a width of 10.0.

The `BankAccount` Example


Overloaded Constructors
Overloaded deposit methods
Overloaded withdraw methods
Overloaded setBalance methods

```
BankAccount
-balance:double
+BankAccount()
+BankAccount(startBalance:double)
+BankAccount(strString):
+deposit(amount:double):void
+deposit(str:String):void
+withdraw(amount:double):void
+withdraw(str:String):void
+setBalance(b:double):void
+setBalance(str:String):void
+getBalance():double
```

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, as long as an instance of the class exists.

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.

Packages and `import` Statements

- Classes in the Java API are organized into *packages.*
- Explicit and Wildcard `import` statements
  - Explicit imports name a specific class
    - `import java.util.Scanner;`
  - Wildcard imports name a package, followed by an `*`
    - `import java.util.*;`

- The `java.lang` package is automatically made available to any Java class.

Some Java Standard Packages

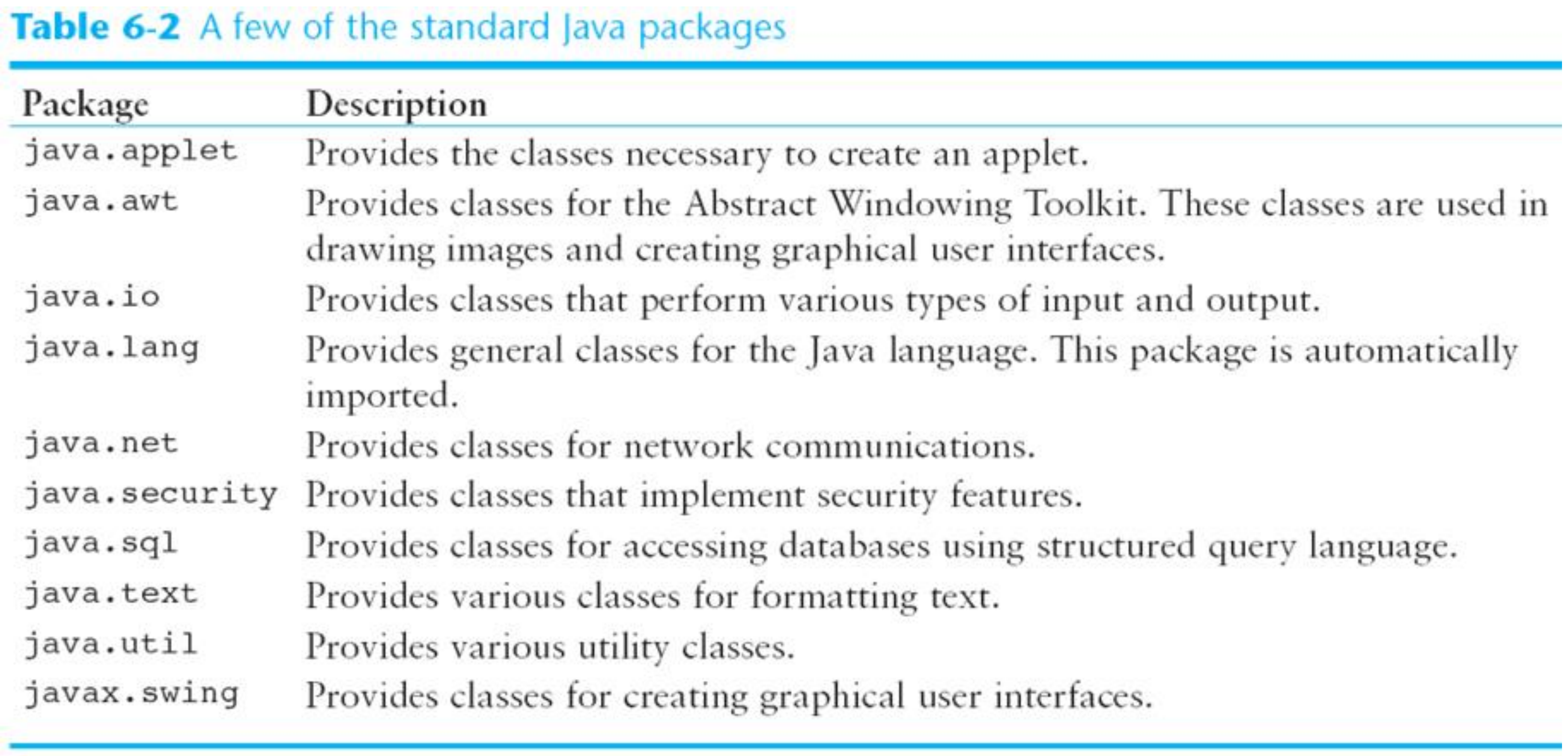


Table 6-2 A few of the standard Java packages

Package	Description
`java.applet`	Provides the classes necessary to create an applet.
`java.awt`	Provides classes for the Abstract Windowing Toolkit. These classes are used in drawing images and creating graphical user interfaces.
`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.
`javax.swing`	Provides classes for creating graphical user interfaces.

Object Oriented Design
Finding Classes and Their Responsibilities

- Finding the classes
  - Get written description of the problem domain
  - Identify all nouns, each is a potential class
  - Refine list to include only classes relevant to the problem
- Identify the responsibilities
  - Things a class is responsible for knowing
  - Things a class is responsible for doing
  - Refine list to include only classes relevant to the problem

Example

- *Joe’s Automotive Shop services foreign cars, and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer’s name, address, and telephone number. Then the manager determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.*

Example - Identifying All of the Nouns

- **Joe's Automotive Shop** services **foreign cars**, and specializes in servicing **cars** made by **Mercedes**, **Porsche**, and **BMW**.
- When a **customer** brings a **car** to the **shop**, the **manager** gets the **customer**'s **name**, **address**, and **telephone number**.
- Then the **manager** determines the **make**, **model**, and **year** of the **car**, and gives the **customer** a **service quote**.
- The **service quote** shows the **estimated parts charges**, **estimated labor charges**, **sales tax**, and **total estimated charges**.

Example - Identifying All of the Nouns

- address
- foreign cars
- Porsche
- BMW
- Joe’s Automotive
- Shop
- customer
- Mercedes
- telephone number
- Name
- year
- sales tax
- car
- make
- service quote
- cars
- manager
- shop
- estimated labor charges
- model
- total estimated charges
- estimated parts charges

Example – Refining the List of Nouns

- **Some of the nouns really mean the same thing**
  - **car**, **cars**, and **foreign cars**
    - These all refer to the general concept of a car.
  - **Joe’s Automotive Shop** and **shop**
    - Both of these refer to the company “Joe’s Automotive Shop.”

Example - Identifying All of the Nouns

- address
- ~~foreign cars~~
- Porsche
- BMW
- ~~Joe's Automotive Shop~~
- customer
- Mercedes
- telephone number
- Name
- year
- sales tax
- car
- make
- service quote
- ~~cars~~
- manager
- shop
- estimated labor charges
- model
- total estimated charges
- estimated parts charges

Example – Refining the List of Nouns

- **Some nouns might represent items that we do not need to be concerned with in order to solve the problem.**
- We can cross **shop** off the list because our application only needs to be concerned with individual service quotes. It doesn’t need to work with or determine any companywide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will not need a class for the **manager** because the problem statement does not direct us to process any information about the manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

Example - Identifying All of the Nouns

- address
- ~~foreign cars~~
- Porsche
- BMW
- ~~Joe's Automotive Shop~~
- customer
- Mercedes
- telephone number
- Name
- year
- sales tax
- car
- make
- service quote
- ~~cars~~
- ~~manager~~
- ~~shop~~
- estimated labor charges
- model
- total estimated charges
- estimated parts charges

Example – Refining the List of Nouns

- **Some of the nouns might represent objects, not classes**

Example - Identifying All of the Nouns

- address
- ~~foreign cars~~
- ~~Porsche~~
- ~~BMW~~
- ~~Joe’s Automotive Shop~~
- customer
- ~~Mercedes~~
- telephone number
- Name
- year
- sales tax
- car
- make
- service quote
- ~~cars~~
- ~~manager~~
- ~~shop~~
- estimated labor charges
- model
- total estimated charges
- estimated parts charges

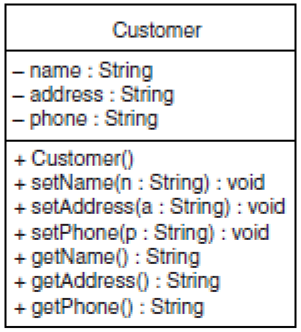
Example – Refining the List of Nouns

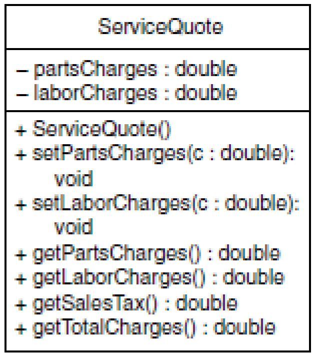
- **Some of the nouns might represent simple values that can be stored in a primitive variable and do not require a class.**
  - ~~address~~
  - ~~foreign cars~~
  - ~~Porsche~~
  - ~~BMW~~
  - ~~Joe's Automotive Shop~~
  - customer
  - ~~Mercedes~~
  - ~~telephone number~~
  - ~~Name~~
  - ~~year~~
  - ~~sales tax~~
  - car
  - ~~make~~
  - service quote
  - ~~cars~~
  - ~~manager~~
  - ~~shop~~
  - ~~estimated labor charges~~
  - model
  - ~~total estimated charges~~
  - ~~estimated parts charges~~

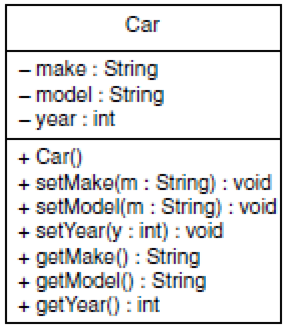
Example - Identifying a Class’s Responsibilities

- A class’s *responsibilities* are as follows:
  - The things that the class is responsible for knowing
  - The actions that the class is responsible for doing
- It is often helpful to ask the questions
  - “In the context of this problem, what must the class know? What must the class do?”

Example - Identifying a Class's Responsibilities




ServiceQuote
– partsCharges : double
– laborCharges : double
+ ServiceQuote()
+ setPartsCharges(c : double): void
+ setLaborCharges(c : double): void
+ getPartsCharges() : double
+ getLaborCharges() : double
+ getSalesTax() : double
+ getTotalCharges() : double



Review of Instance Fields and Methods

- 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.

Static Class Members

- *Static fields* and *static methods* do not belong to a single instance of a class.
- To invoke a static method or use a static field, the class name, rather than the instance name, is used.
- Example:

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

Class name

Static method

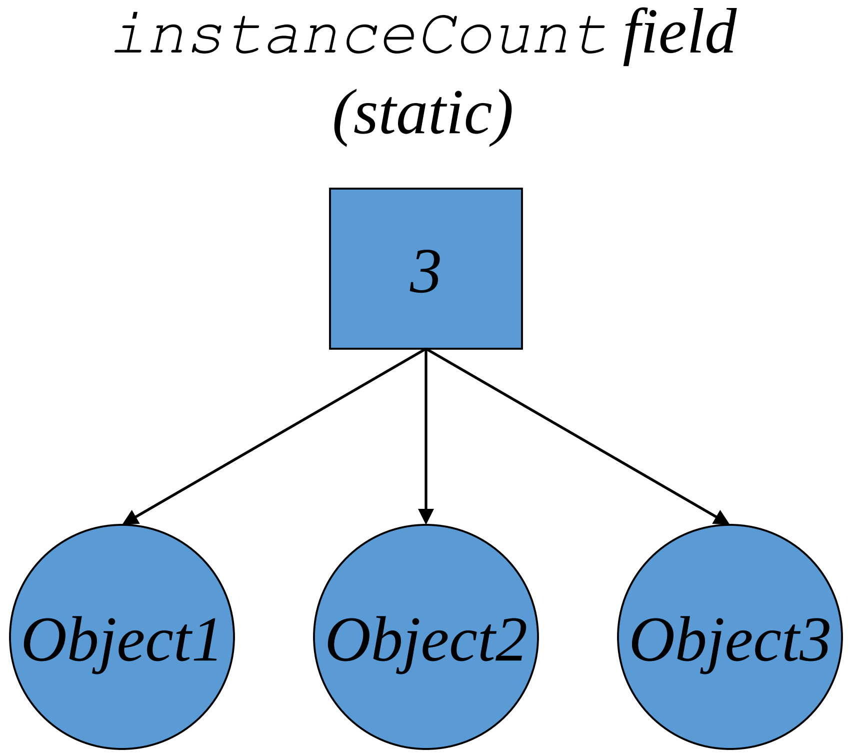
Static Fields

- 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.
  - Primitive static fields are initialized to 0 if no initialization is performed.

Static Fields



Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier 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);
```

Static Methods

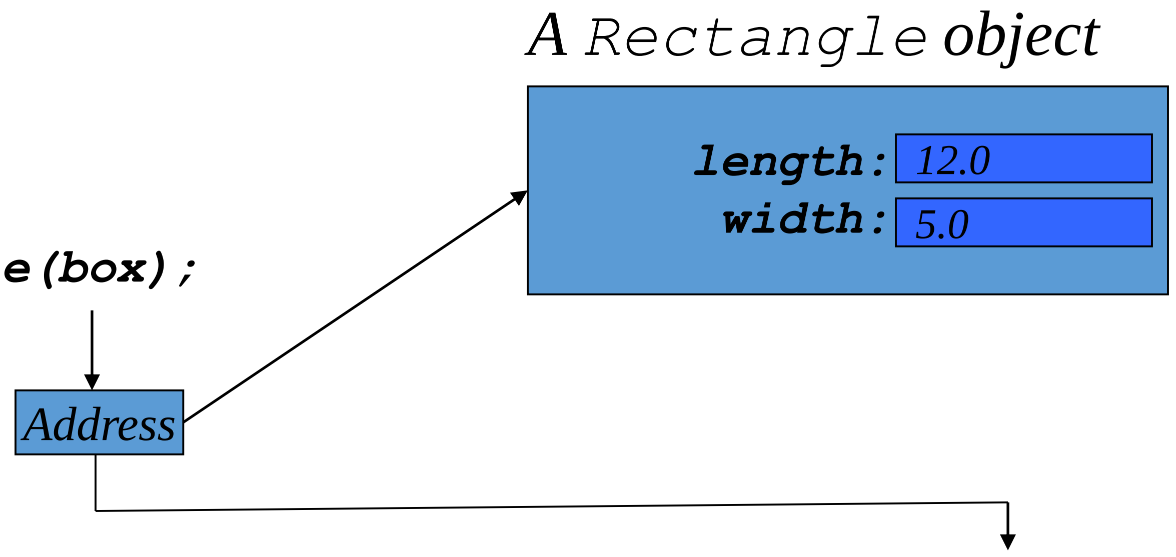
- Static methods are convenient because they may be called at the class level.
- They are typically used to create utility classes, such as the `Math` class in the Java Standard Library.
- Static methods may not communicate with instance fields, only static fields.

Passing Objects as Arguments

- Objects can be passed to methods as arguments.
- Java passes all arguments *by value*.
- When an object is passed as an argument, the value of the reference variable is passed.
- The value of the reference variable is an address or reference to the object in memory.
- A *copy* of the object is *not passed*, just a pointer to the object.
- When a method receives a reference variable as an argument, it is possible for the method to modify the contents of the object referenced by the variable.

Passing Objects as Arguments

```
displayRectangle(box);
```



```
public static void displayRectangle(Rectangle r)
{
  // Display the length and width.
  System.out.println("Length: " + r.getLength() +
                     " Width: " + r.getWidth());
}
```

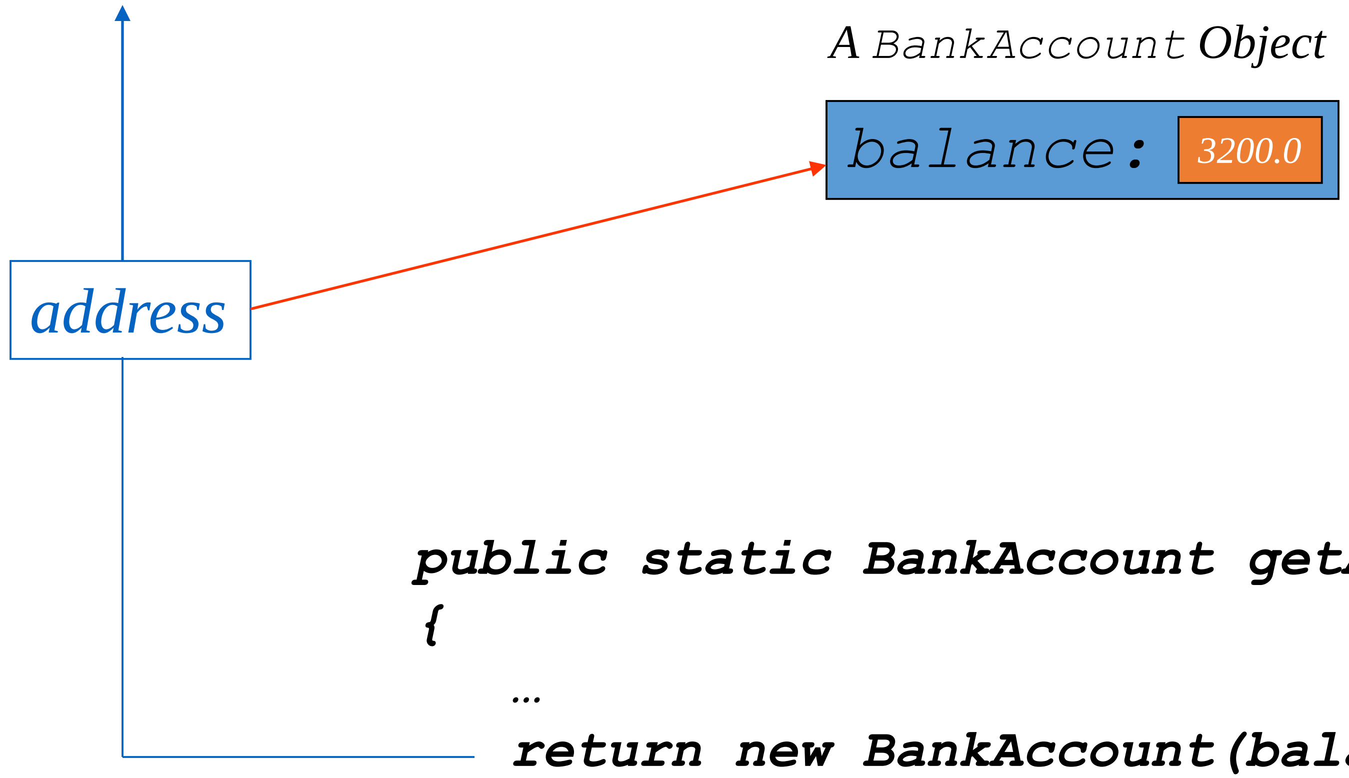
Returning Objects From Methods

- Methods are not limited to returning the primitive data types.
- Methods can return references to objects as well.
- Just as with passing arguments, a copy of the object is **not** returned, only its address.
- Method return type:

```
public static BankAccount getAccount()
{
      …
    return new BankAccount(balance);
}
```

Returning Objects from Methods

```
account = getAccount();
```



```
public static BankAccount getAccount()
{
   …
   return new BankAccount(balance);
}
```

The `toString` Method

- The `toString` method of a class can be called *explicitly*:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany.toString());
```

- However, the `toString` method does not have to be called explicitly but is called implicitly whenever you pass an object of the class to `println` or `print`.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany);
```

The `toString` method

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

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println("The stock data is:\n" +
 xyzCompany);
```

The `toString` Method

- 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.

The `equals` Method

- 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

- 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

- 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.

Methods That Copy Objects

- 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.

Copy Constructors

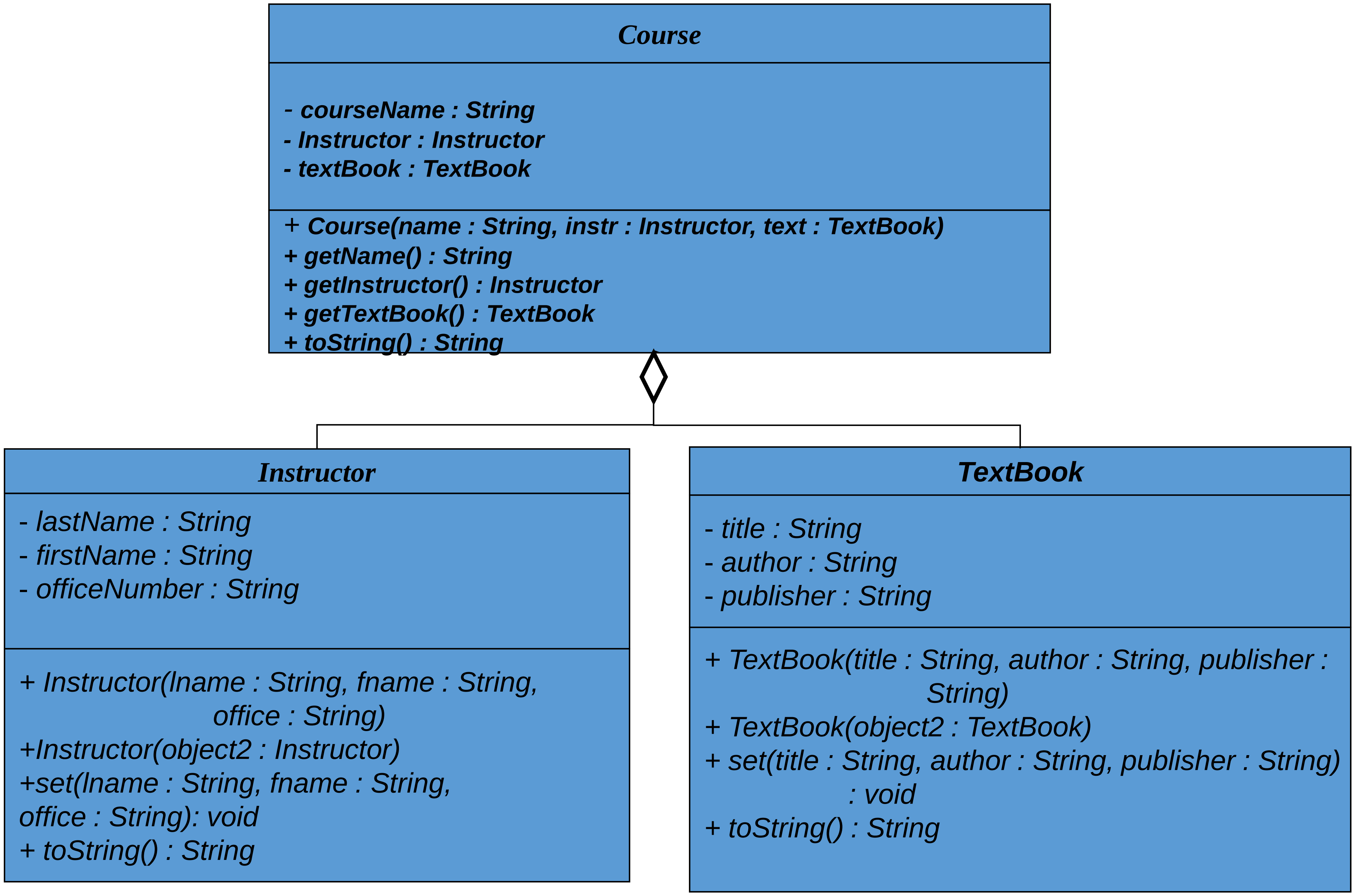
- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object 2)
{
    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);
```

Aggregation

- Creating an instance of one class as a reference in another class is called *object aggregation*.
- Aggregation creates a “has a” relationship between objects.

Aggregation in UML Diagrams


Course
- courseName : String
- Instructor : Instructor
- textBook : TextBook
+ Course(name : String, instr : Instructor, text : TextBook)
+ getName() : String
+ getInstructor() : Instructor
+ getTextBook() : TextBook
+ toString() : String
Instructor
- lastName : String
- firstName : String
- officeNumber : String
+ Instructor(lname : String, fname : String, office : String)
+Instructor(object2 : Instructor)
+set(lname : String, fname : String, office : String): void
+ toString() : String
TextBook
- title : String
- author : String
- publisher : String
+ TextBook(title : String, author : String, publisher : String)
+ TextBook(object2 : TextBook)
+ set(title : String, author : String, publisher : String) : void
+ toString() : String

Returning References to Private Fields

- Avoid returning references to private data elements.
- Returning references to private variables will allow any object that receives the reference to modify the variable.

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());
}
```

The `this` Reference

- 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

- 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.

Enumerated Types

- Known as an `enum`, requires declaration and definition like a class
- Syntax:
  `enum typeName {` *one or more enum constants* `}`
  - Definition:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
           FRIDAY, SATURDAY }
```

  - Declaration:

```
Day WorkDay; // creates a Day enum
```

  - Assignment:

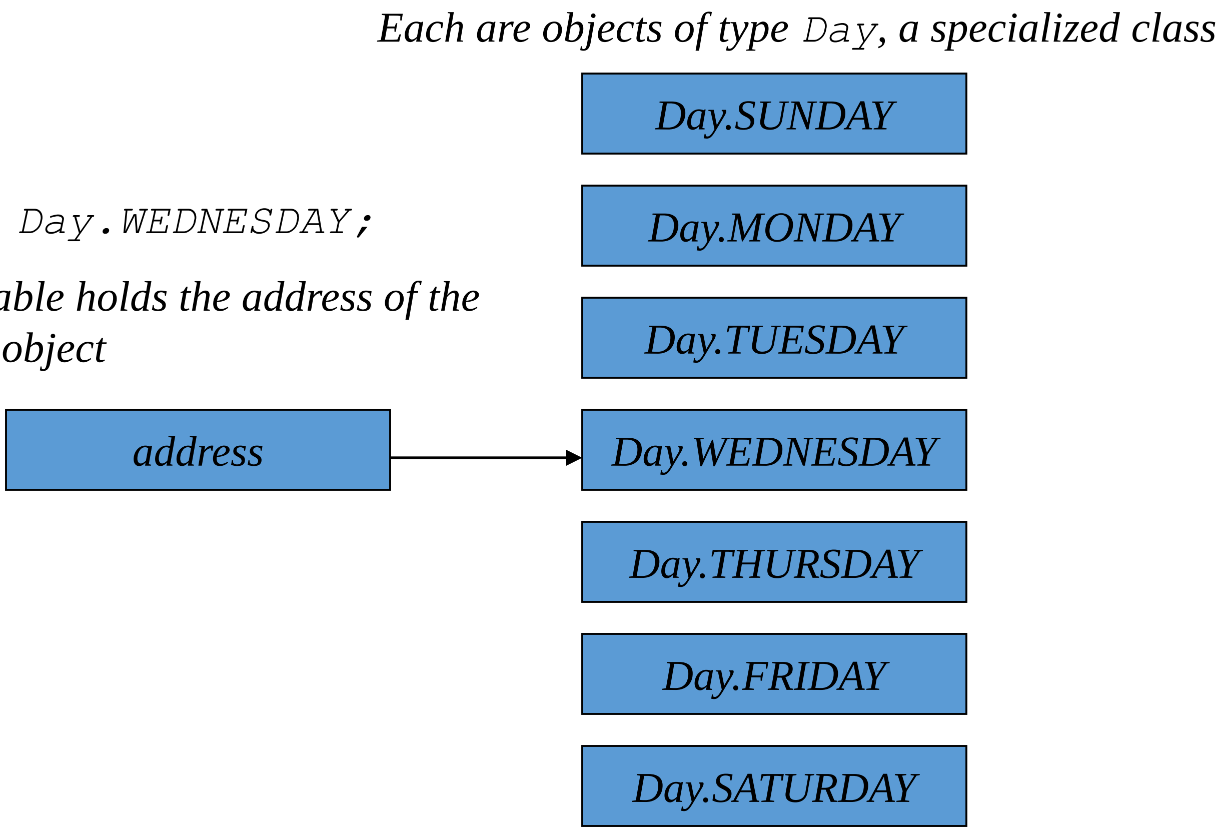
```
Day WorkDay = Day.WEDNESDAY;
```

Enumerated Types

- An `enum` is a specialized class

```
Day workDay = Day.WEDNESDAY;
```

The `workDay` *variable holds the address of the* `Day.WEDNESDAY` *object*



Enumerated Types - Methods

- `toString` – returns name of calling constant
- `ordinal` – returns the zero-based position of the constant in the enum. For example the ordinal for `Day.THURSDAY` is 4
- `equals` – accepts an object as an argument and returns true if the argument is equal to the calling enum constant
- `compareTo` - accepts an object as an argument and returns a negative integer if the calling constant's ordinal < than the argument's ordinal, a positive integer if the calling constant's ordinal > than the argument's ordinal and zero if the calling constant's ordinal == the argument's ordinal.

Enumerated Types - Switching

- Java allows you to test an enum constant with a `switch` statement.

Garbage Collection

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

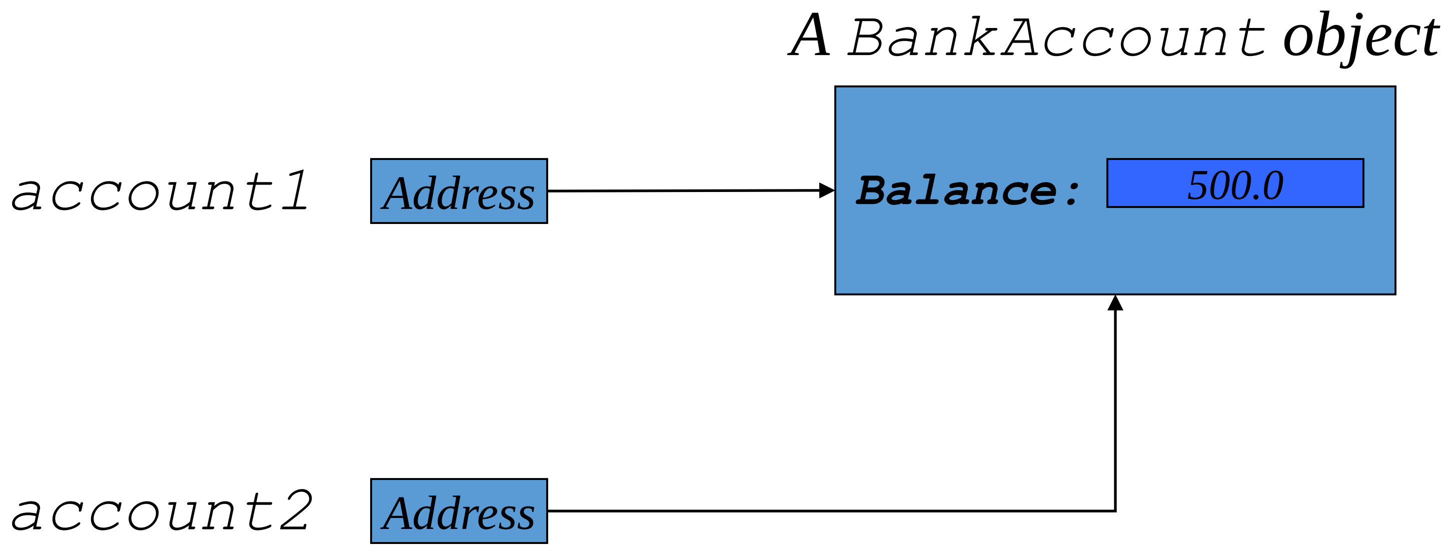
Garbage Collection

- 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;
```

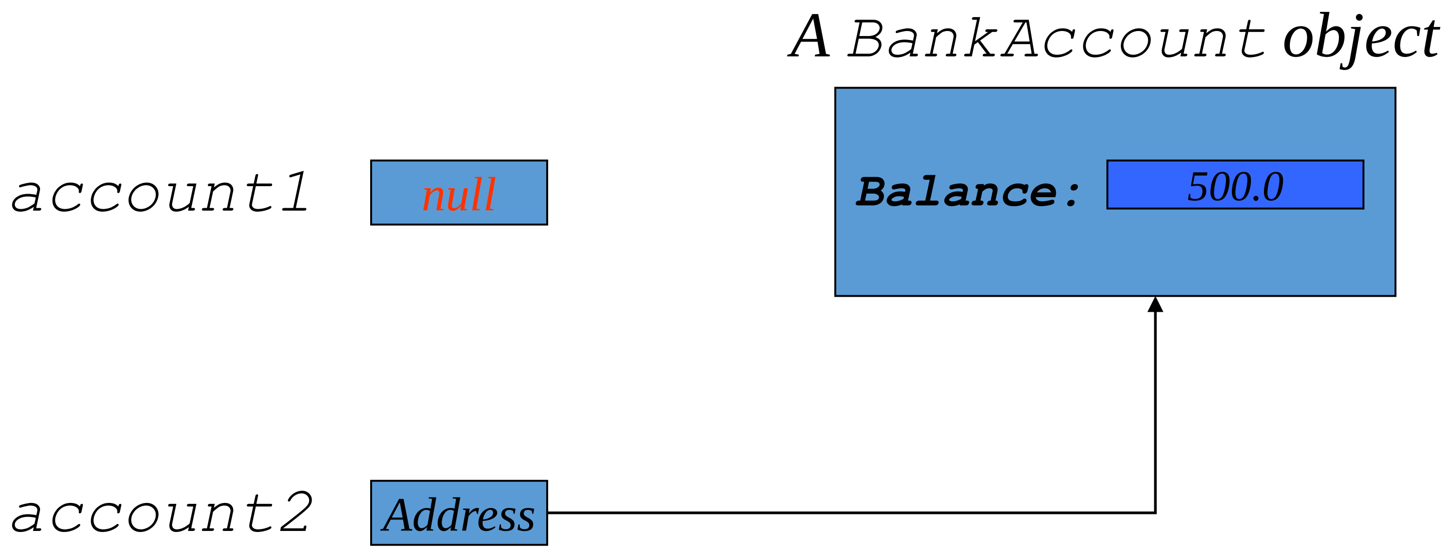
- This sets `account1` and `account2` to point to the same object.

Garbage Collection



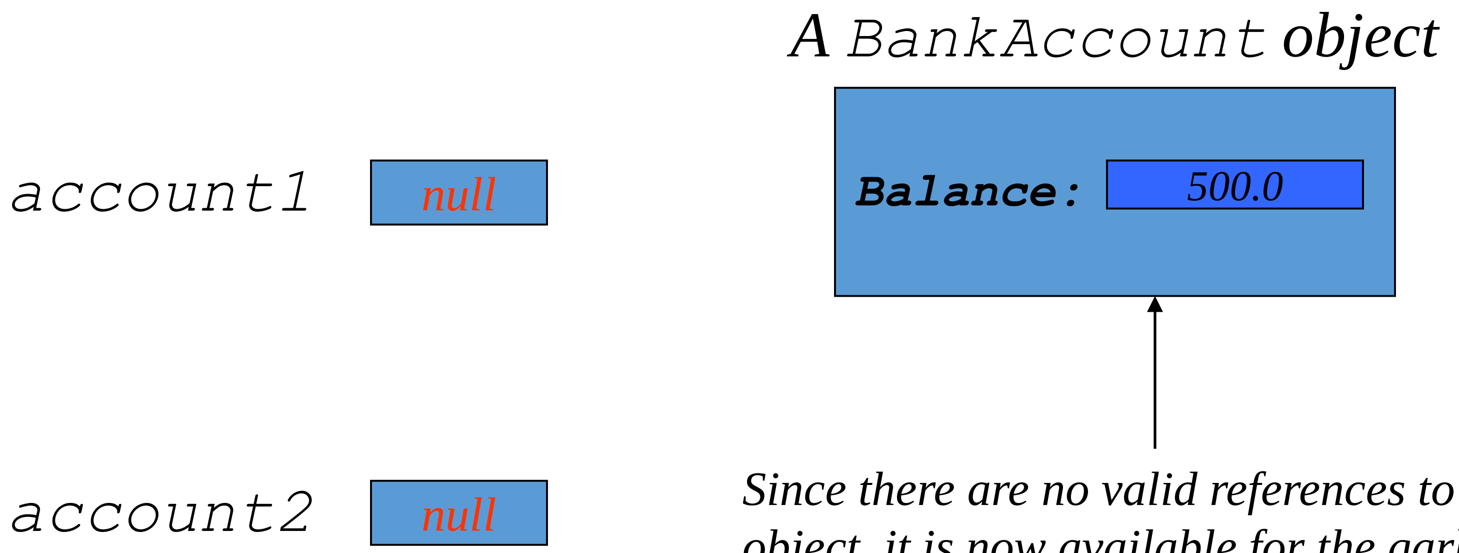
Here, both `account1` and `account2` point to the same instance of the `BankAccount` class.

Garbage Collection



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

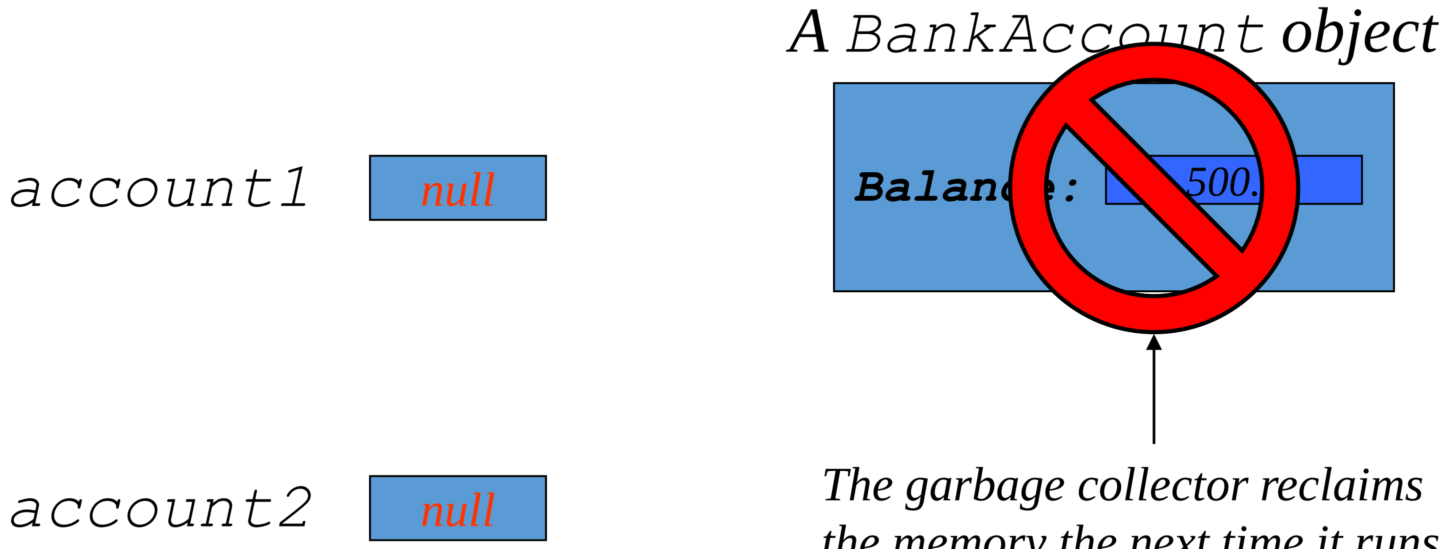
Garbage Collection



Since there are no valid references to this object, it is now available for the garbage collector to reclaim.

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

Garbage Collection



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.

Class Collaboration

- Collaboration – two classes interact with each other
- If an object is to collaborate with another object, it must know something about the second object's methods and how to call them
- If we design a class `StockPurchase` that collaborates with the `Stock` class (previously defined), we define it to create and manipulate a `Stock` object

CRC Cards

- Class, Responsibilities and Collaborations (CRC) cards are useful for determining and documenting a class’s responsibilities
  - The things a class is responsible for knowing
  - The actions a class is responsible for doing
- CRC Card Layout (Example for class Stock)

Stock	
Know stock to purchase	*Stock class*
Know number of shares	*None*
Calculate cost of purchase	*Stock class*
Etc.	*None or class name*