جـامعة الجـلالة
GALALA UNIVERSITY

CSE110 Principles of Programming

Lecture 8: Object Oriented Programming (A second look)

Professor Shaker El-Sappagh

Shaker.elsappagh@gu.edu.eg

Fall 2023

Chapter Topics

- Static Class Members
- Passing Objects as Arguments to Methods
- Returning Objects from Methods
- The `toString` method
- Writing an `equals` Method
- Methods that Copy Objects
- Aggregation
- The `this` Reference Variable
- Enumerated Types
- Garbage Collection
- Focus on Object-Oriented Design: Class Collaboration

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);
```

Static Fields

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

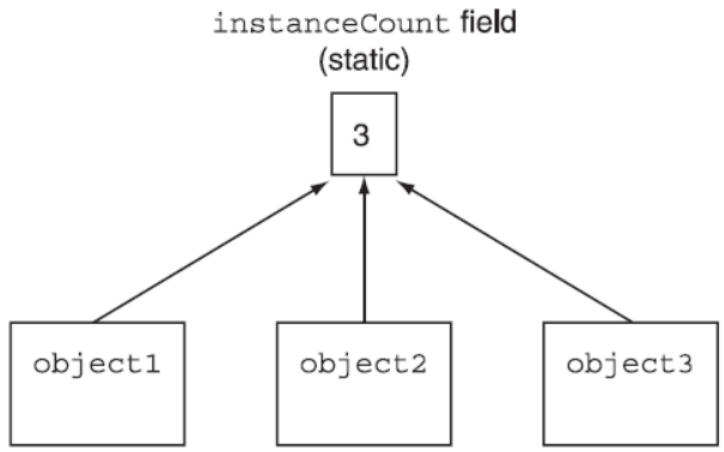
```
private static int instanceCount = 0;
```

- 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.

```
public class Countable
{
   private static int instanceCount = 0;

   /**
      The constructor increments the static
      field instanceCount. This keeps track
      of the number of instances of this
      class that are created.
   */

   public Countable()
   {
```

```
        instanceCount++;
    }

    /**
        The getInstanceCount method returns
        the number of instances of this class
        that have been created.
        @return The value in the instanceCount field.
    */

    public int getInstanceCount()
    {
        return instanceCount;
    }
}
```

```
public class StaticDemo
{
   public static void main(String[] args)
   {
      int objectCount;
      Countable object1 = new Countable();
      Countable object2 = new Countable();
      Countable object3 = new Countable();

      // Get the number of instances from
      // the class's static field.
      objectCount = object1.getInstanceCount();
      System.out.println(objectCount +
                         " instances of the class " +
                         "were created.");
   }
}
```



Instances of the `Countable` class

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);
```

```
public class Metric
{
    public static double milesToKilometers(double m)
    {
        return m * 1.609;
    }

    /**
        The kilometersToMiles method converts
        a distance in kilometers to miles.
        @param k The distance in kilometers.
        @return The distance in miles.
    */

    public static double kilometersToMiles(double k)
    {
        return k / 1.609;
    }
}
```

```
public class MetricDemo
{
   public static void main(String[] args)
   {
      String input; // To hold input
      double miles; // A distance in miles
      double kilos; // A distance in kilometers

      // Get a distance in miles.
      input = JOptionPane.showInputDialog("Enter " +
                              "a distance in miles.");
      miles = Double.parseDouble(input);

      // Convert the distance to kilometers.
      kilos = Metric.milesToKilometers(miles);
      JOptionPane.showMessageDialog(null,
        String.format("%,.2f miles equals %,.2f kilometers.",
                      miles, kilos));

      // Get a distance in kilometers.
      input = JOptionPane.showInputDialog("Enter " +
                       "a distance in kilometers: ");
      kilos = Double.parseDouble(input);

      // Convert the distance to kilometers.
      miles = Metric.kilometersToMiles(kilos);
      JOptionPane.showMessageDialog(null,
        String.format("%,.2f kilometers equals %,.2f miles.",
                      kilos, miles));
```

Static Methods

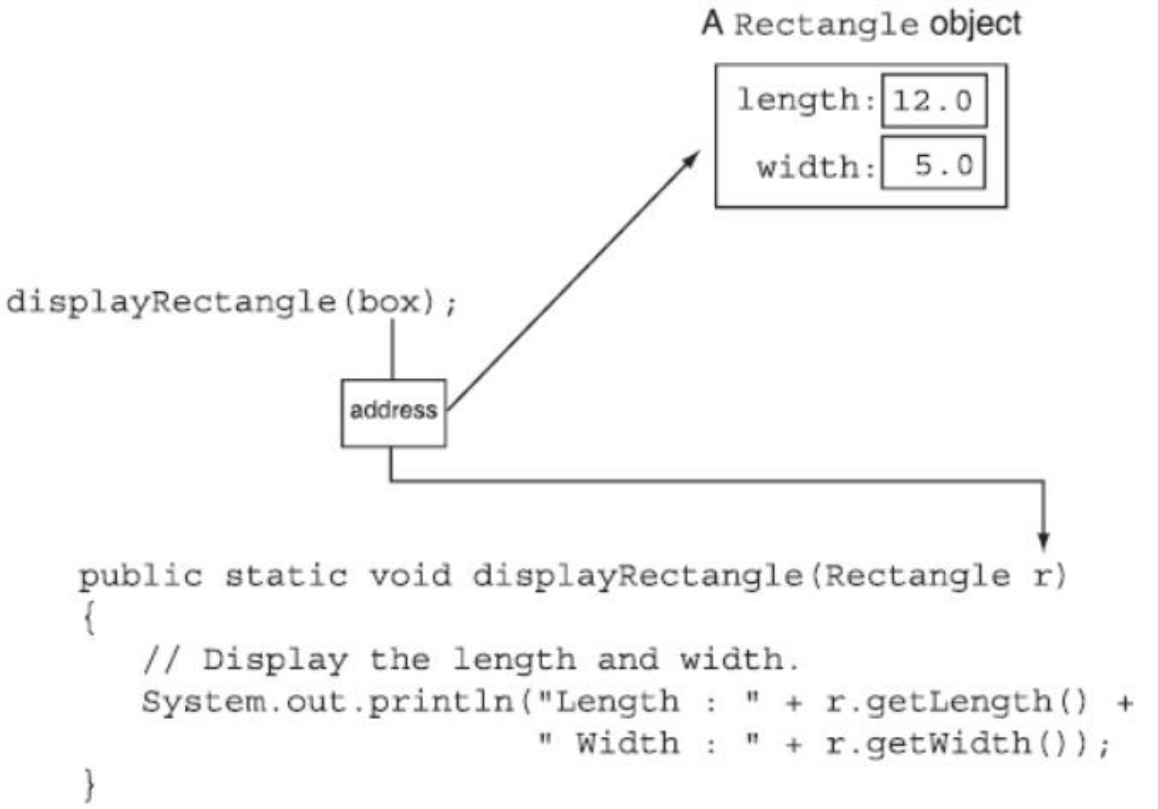
- Static methods are convenient because they may be called at the **class level**.
- They are typically used to create **utility classes**, such as the `Math` class in the Java Standard Library.
- Static methods **may not communicate** with instance fields, only static fields.

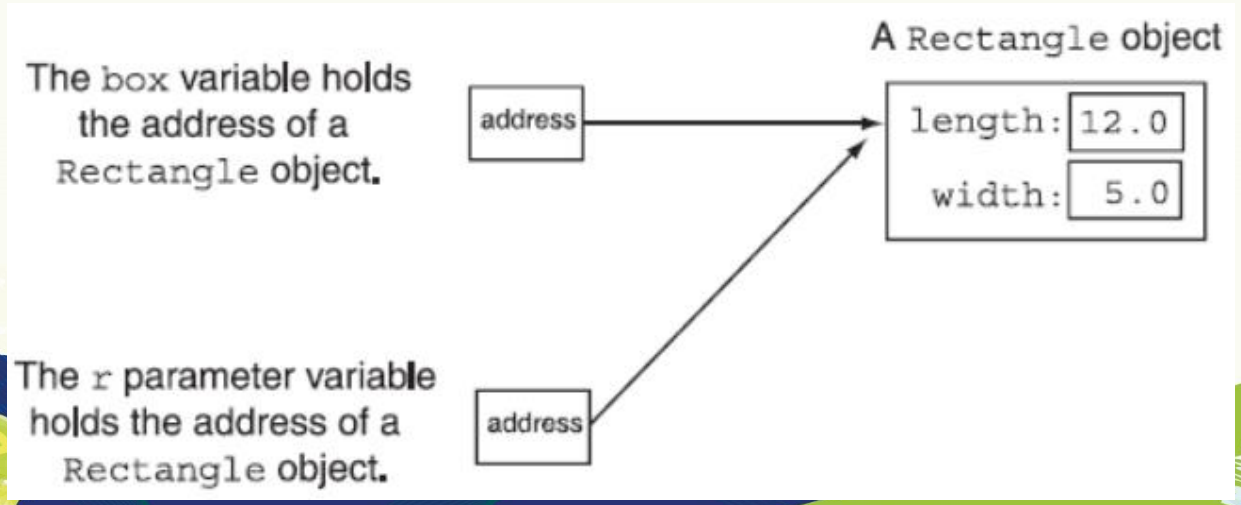
Passing Objects as Arguments

- 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.

```
public class PassObject
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle(12.0, 5.0);

      // Pass a reference to the object to
      // the displayRectangle method.
      displayRectangle(box);
   }

   /**
      The displayRectangle method displays the
      length and width of a rectangle.
      @param r A reference to a Rectangle
      object.
   */

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

Passing Objects as Arguments





```
public class PassObject2
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle(12.0, 5.0);

      // Display the object's contents.
      System.out.println("Contents of the box object:");
      System.out.println("Length : " + box.getLength() +
                         " Width : " + box.getWidth());

      // Pass a reference to the object to the
      // changeRectangle method.
      changeRectangle(box);

      // Display the object's contents again.
      System.out.println("\nNow the contents of the " +
                         "box object are:");
      System.out.println("Length : " + box.getLength() +
                         " Width : " + box.getWidth());
   }

   /**
      The changeRectangle method sets a Rectangle
      object's length and width to 0.
      @param r The Rectangle object to change.
   */

   public static void changeRectangle(Rectangle r)
   {
      r.setLength(0.0);
      r.setWidth(0.0);
   }
}
```

Program Output

```
Contents of the box object:
Length : 12.0 Width : 5.0

Now the contents of the box object are:
Length : 0.0 Width : 0.0
```

Returning Objects From Methods

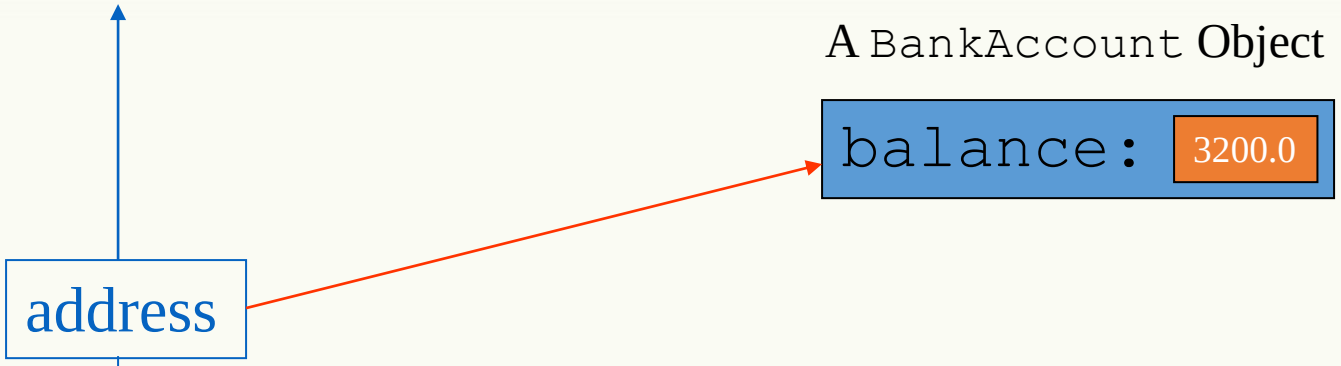
- Methods are not limited to returning the primitive data types.
- Methods can **return references** to objects as well.
- Just as with passing arguments, a copy of the object is **not** returned, **only its address**.
- Method return type:

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

```
public class ReturnObject
{
   public static void main(String[] args)
   {
      BankAccount account;

      // Get a reference to a BankAccount object.
      account = getAccount();

      // Display the account's balance.
      JOptionPane.showMessageDialog(null,
                "The account has a balance of $" +
                account.getBalance());
       System.exit(0);
    }

    public static BankAccount getAccount()
    {
       String input;         // To hold input
       double balance;       // Account balance

       // Get the balance from the user.
       input = JOptionPane.showInputDialog("Enter " +
                                  "the account balance.");
       balance = Double.parseDouble(input);

       // Create a BankAccount object and return
       // a reference to it.
       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);
```

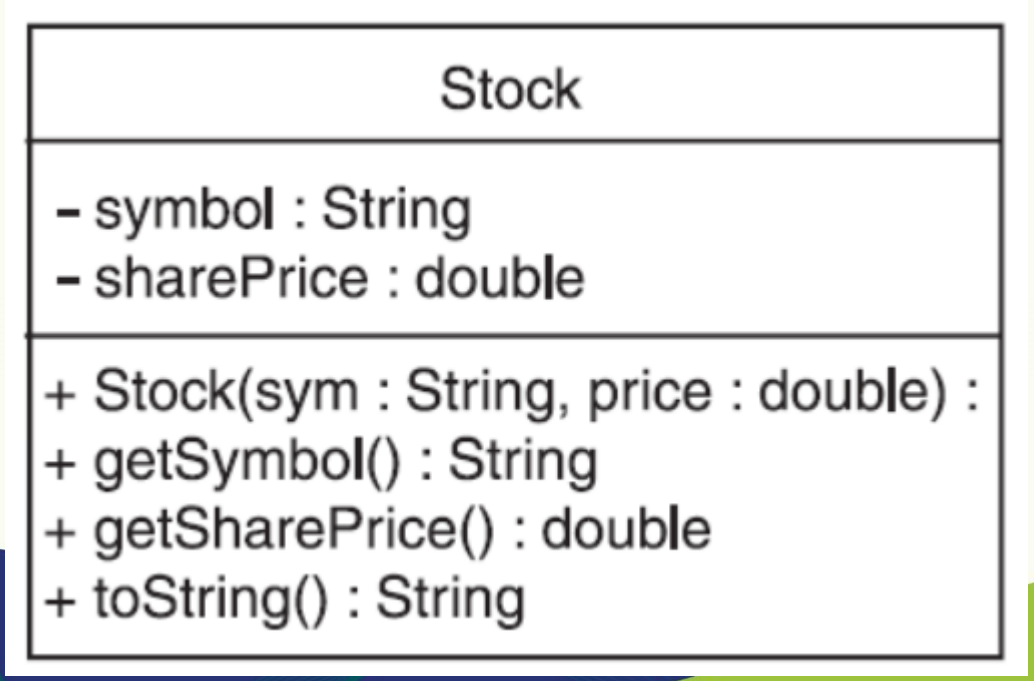
The `toString` method

- 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.



```
public class Stock
{
   private String symbol;         // Trading symbol of stock
   private double sharePrice;     // Current price per share
   public Stock(String sym, double price)
   {
      symbol = sym;
      sharePrice = price;
   }

   /**
      getSymbol method
      @return The stock's trading symbol.
   */

   public String getSymbol()
   {
      return symbol;
   }

   /**
      getSharePrice method
      @return The stock's share price
   */

   public double getSharePrice()
   {
      return sharePrice;
   }

   public String toString()
   {
      // Create a string describing the stock.
      String str = "Trading symbol: " + symbol +
                   "\nShare price: " + sharePrice;

      // Return the string.
      return str;
   }
}
```

```
public class StockDemo1
{
   public static void main(String[] args)
   {
      // Create a Stock object for the XYZ Company.
      // The trading symbol is XYZ and the current
      // price per share is $9.62.
      Stock xyzCompany = new Stock ("XYZ", 9.62);

      // Display the object's values.
      System.out.println(xyzCompany);
   }
}
```

Program Output

```
Trading symbol: XYZ
Share price: 9.62
```

The `equals` Method

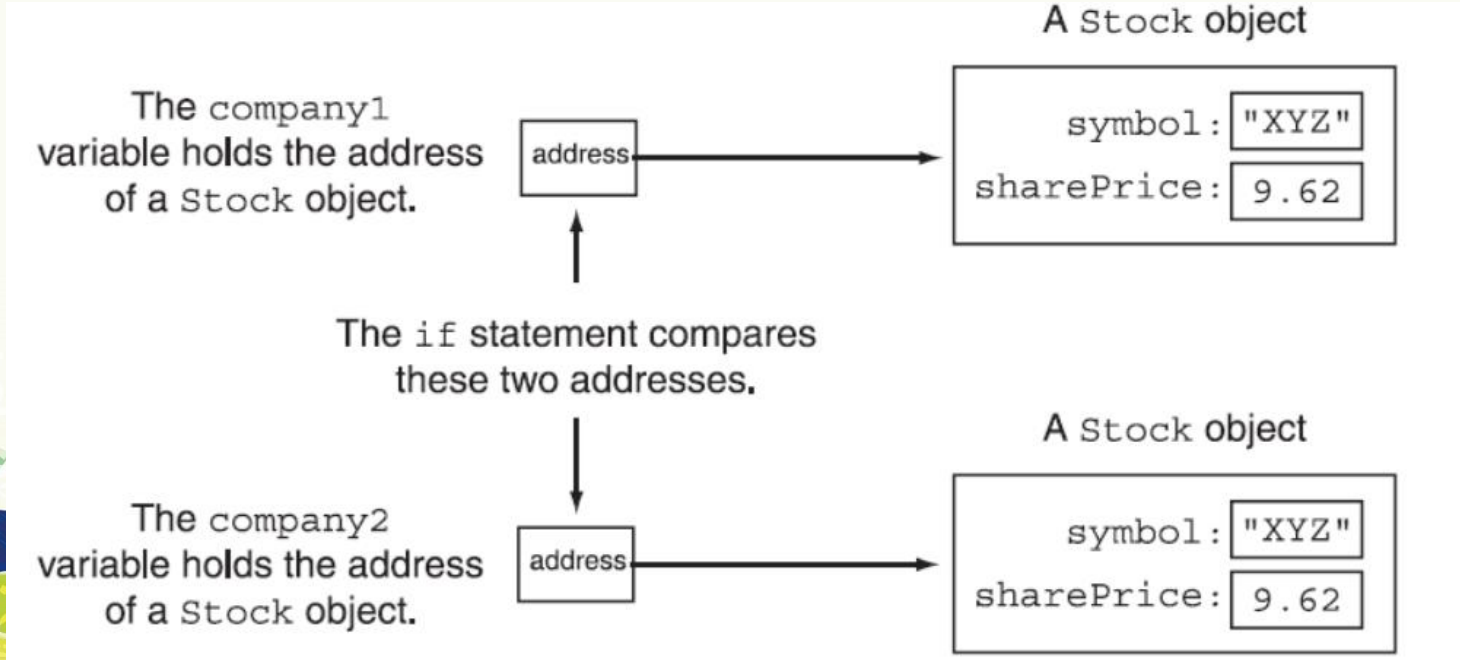
- When the == operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.
- All objects have an `equals` method.
- The default operation of the `equals` method is to compare memory addresses of the objects (just like the ==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

```
public class StockCompare
{
   public static void main(String[] args)
   {
      // Create two Stock objects with the same values.
      Stock company1 = new Stock("XYZ", 9.62);
      Stock company2 = new Stock("XYZ", 9.62);

      // Use the equals method to compare the objects.
      if (company1.equals(company2))
         System.out.println("Both objects are the same.");
      else
         System.out.println("The objects are different.");
   }
}
```

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.

```
public Stock copy()
{
    // Create a new Stock object and initialize it
    // with the same data held by the calling object.
    Stock copyObject = new Stock(symbol, sharePrice);

    // Return a reference to the new object.
    return copyObject;
}
```

```
public class ObjectCopy
{
   public static void main(String[] args)
   {
      // Create a Stock object.
      Stock company1 = new Stock("XYZ", 9.62);

      // Declare a Stock variable
      Stock company2;

      // Make company2 reference a copy of the object
      // referenced by company1.
      company2 = company1.copy();

      // Display the contents of both objects.
      System.out.println("Company 1:\n" + company1);
      System.out.println();
      System.out.println("Company 2:\n" + company2);

      // Confirm that we actually have two objects.
      if (company1 == company2)
      {
         System.out.println("The company1 and company2 " +
                  "variables reference the same object.");
      }
      else
      {
         System.out.println("The company1 and company2 " +
                "variables reference different objects.");
      }
   }
}
```

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);
```

Reusability

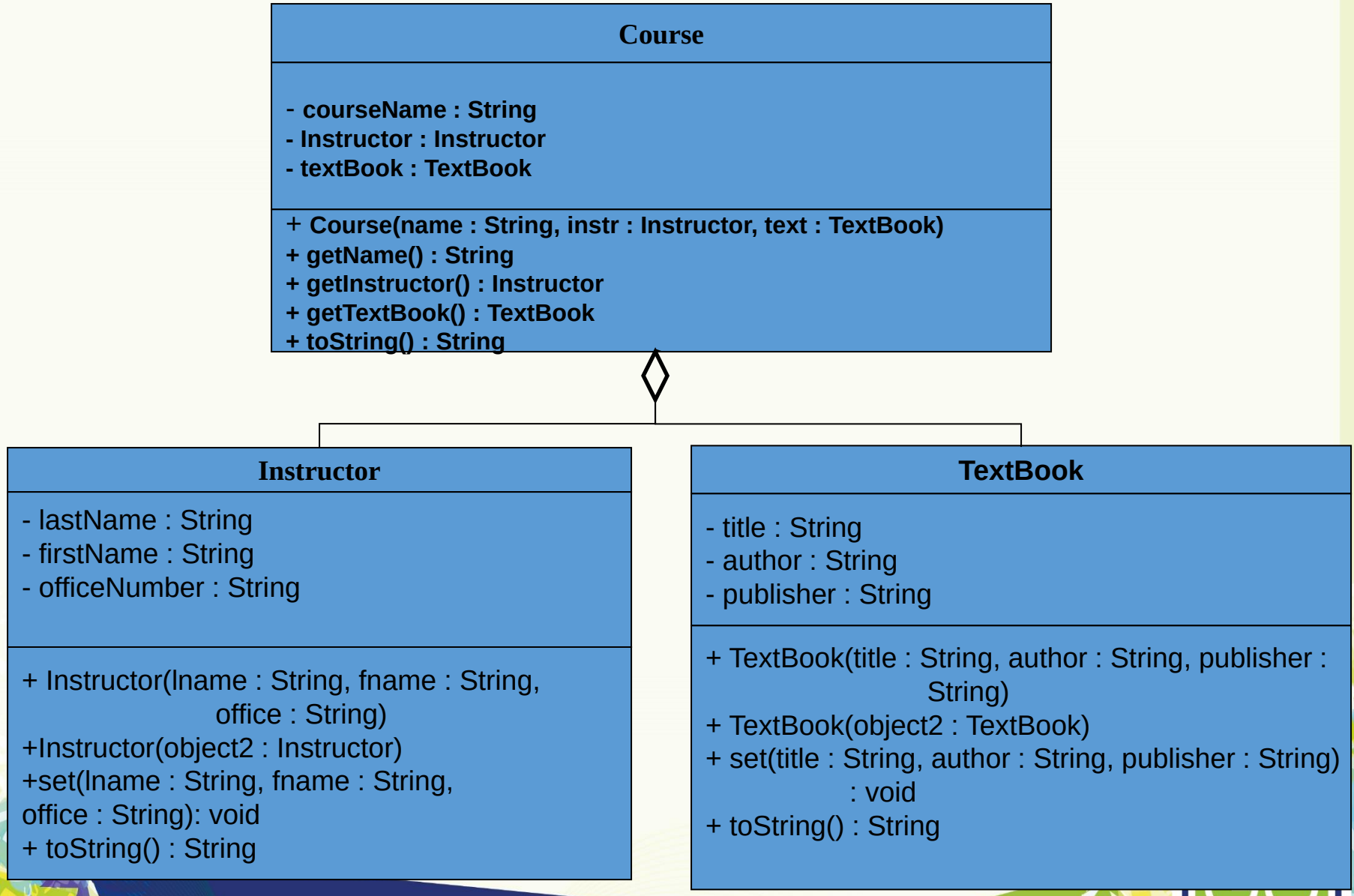
OOP supports two types of reusability:

1. Is a relationship (inheritance)
2. Has a relationship (Composition or Aggregation)

Composition or 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



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());
}
```

```
public class FullName
{
   private String lastName;          // Last name
   private String firstName;         // First name
   private String middleName;        // Middle name

   /**
      The setLastName method sets the lastName field.
      @param str The String to set lastName to.
   */

   public void setLastName(String str)
   {
      lastName = str;
   }
 public void setFirstName(String str)
 {
    firstName = str;
 }
```

```
public void setMiddleName(String str)
{
    middleName = str;
}
public int getLength()
{
    return lastName.length() + firstName.length()
           + middleName.length();
  public String toString()
  {
     return firstName + " " + middleName + " "
            + lastName;
  }
}
```

```
public class NameTester
{
   public static void main(String[] args)
   {
      int len;  // To hold the name length
       FullName name = new FullName();

       // Get the length of the full name.
       len = name.getLength();
   }
}
```

This program will crash when you run it because the getLength method is called before the name object's fields are made to reference String objects.

One way to prevent the program from crashing is to use if statements in the getLength method to determine whether any of the fields are set to null.

```
public int getLength()
{
   int len = 0;

   if (lastName != null)
      len += lastName.length();

   if (firstName != null)
      len += firstName.length();

   if (middleName != null)
      len += middleName.length();

   return len;
}
```

Another way to handle this problem is to write a no-arg constructor that assigns values to the reference fields. Here is an example:

```
public FullName()
{
    lastName = "";
    firstName = "";
    middleName = "";
}
```

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.
}
```

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.
- **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;
```

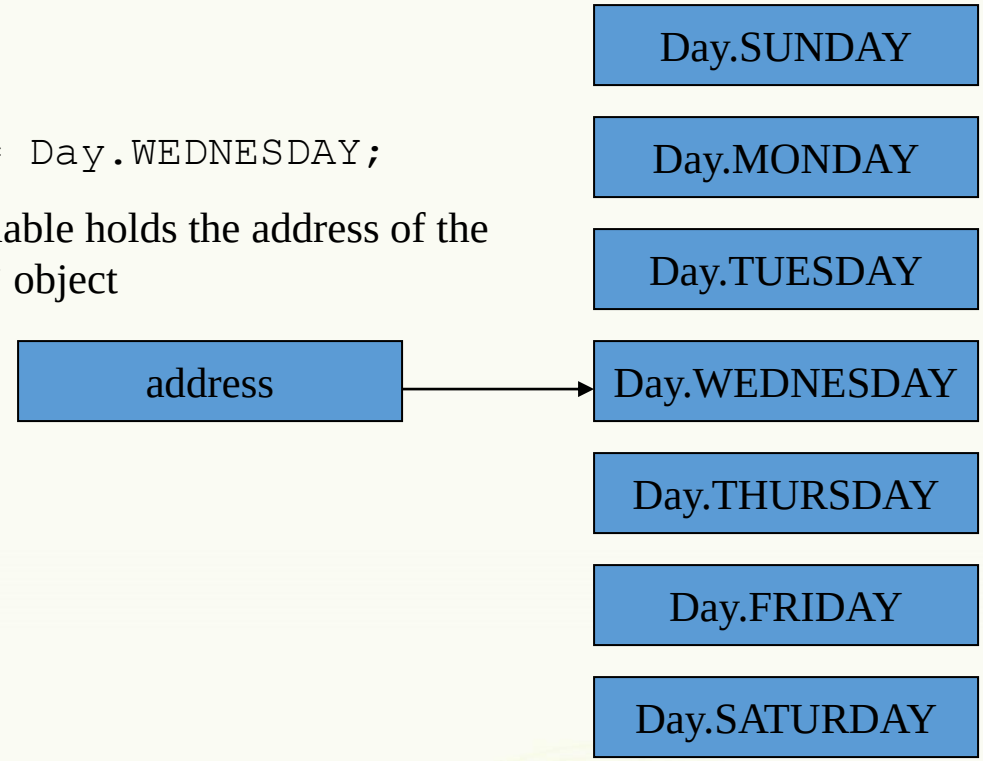
Enumerated Types

- An `enum` is a specialized class

Each are objects of type `Day`, a specialized class

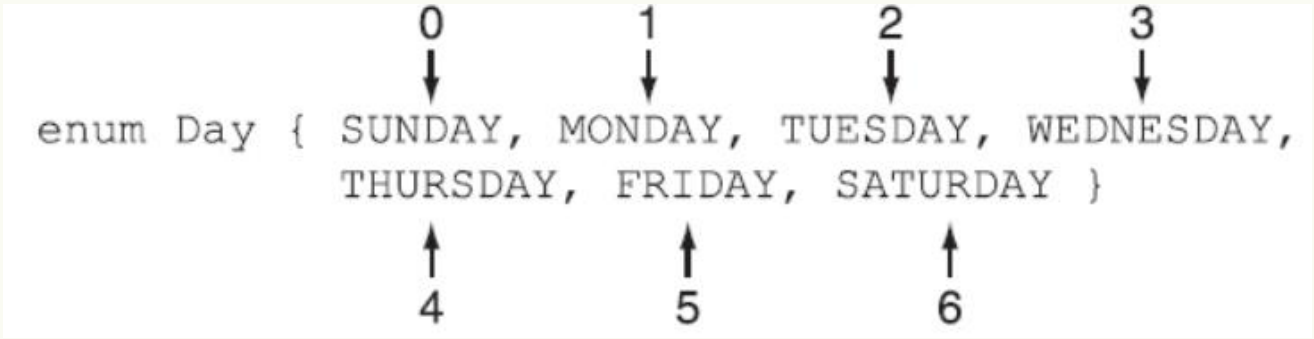
```
Day workDay = Day.WEDNESDAY;
```

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.

```
public class EnumDemo
{
   // Declare the Day enumerated type.
   enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
              THURSDAY, FRIDAY, SATURDAY }

   public static void main(String[] args)
   {
      // Declare a Day variable and assign it a value.
      Day workDay = Day.WEDNESDAY;

      // The following statement displays WEDNESDAY.
      System.out.println(workDay);

      // The following statement displays the ordinal
      // value for Day.SUNDAY, which is 0.
      System.out.println("The ordinal value for " +
                         Day.SUNDAY + " is " +
                         Day.SUNDAY.ordinal());

      // The following statement displays the ordinal
      // value for Day.SATURDAY, which is 6.
      System.out.println("The ordinal value for " +
                         Day.SATURDAY + " is " +
                         Day.SATURDAY.ordinal());
```

```
        // The following statement compares two enum constants.
        if (Day.FRIDAY.compareTo(Day.MONDAY) > 0)
            System.out.println(Day.FRIDAY + " is greater than " +
                               Day.MONDAY);
        else
            System.out.println(Day.FRIDAY + " is NOT greater than " +
                               Day.MONDAY);
    }
}
```

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 the memory operations for you.
- Simply set the reference to *null* and Java will reclaim the memory.

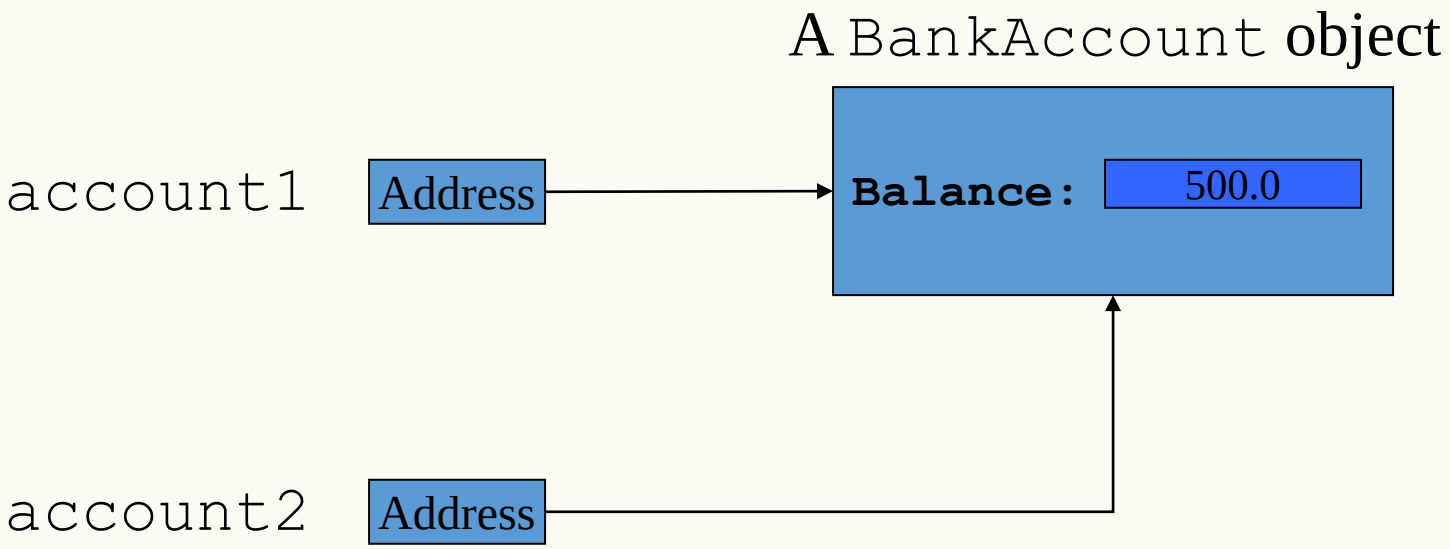
Garbage Collection

- 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;
```

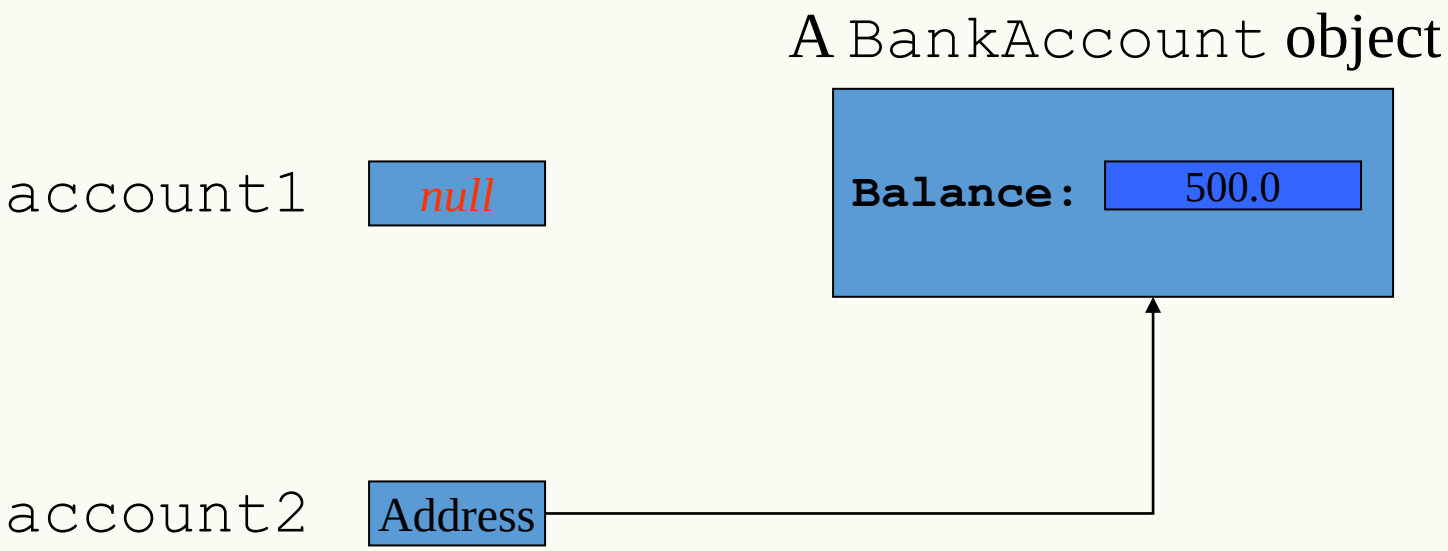
- This sets `account1` and `account2` to point to the same object.

Garbage Collection



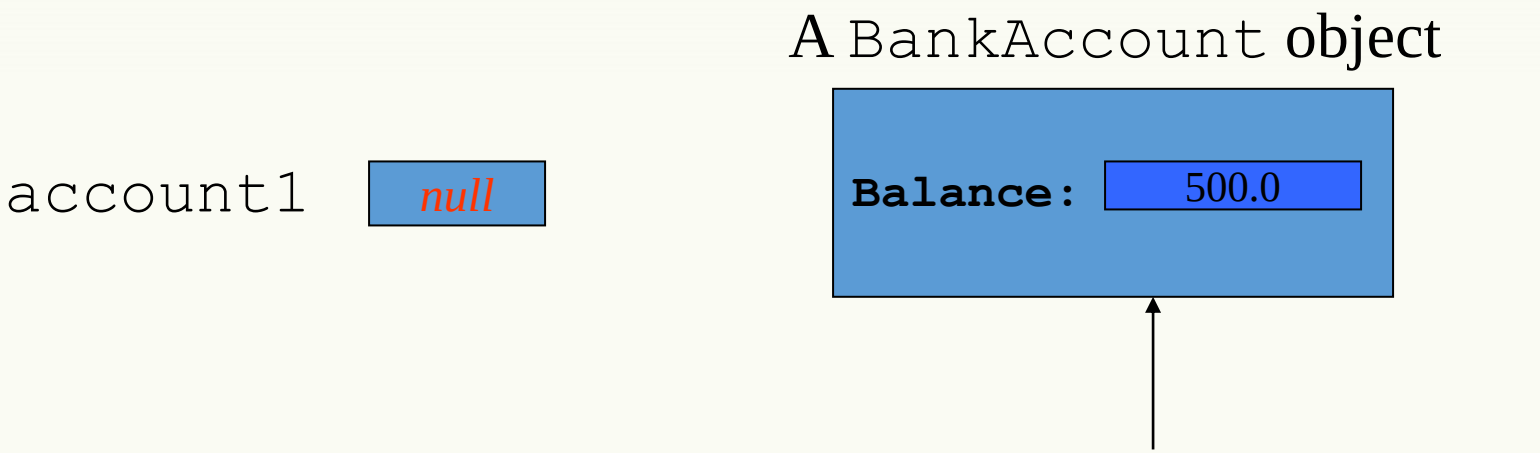
Here, both `account1` and `account2` point to the same instance of the `BankAccount` class.

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.

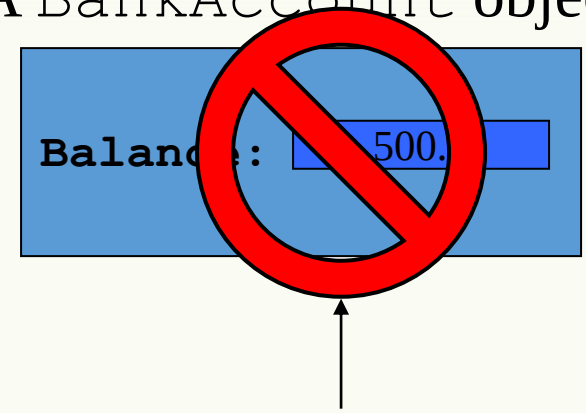
If we now run the statement: `account2 = null;`
neither `account1` or `account2` will be pointing to the object.

Garbage Collection

`account1` *null*

`account2` *null*

A `BankAccount` object



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

The `finalize` Method

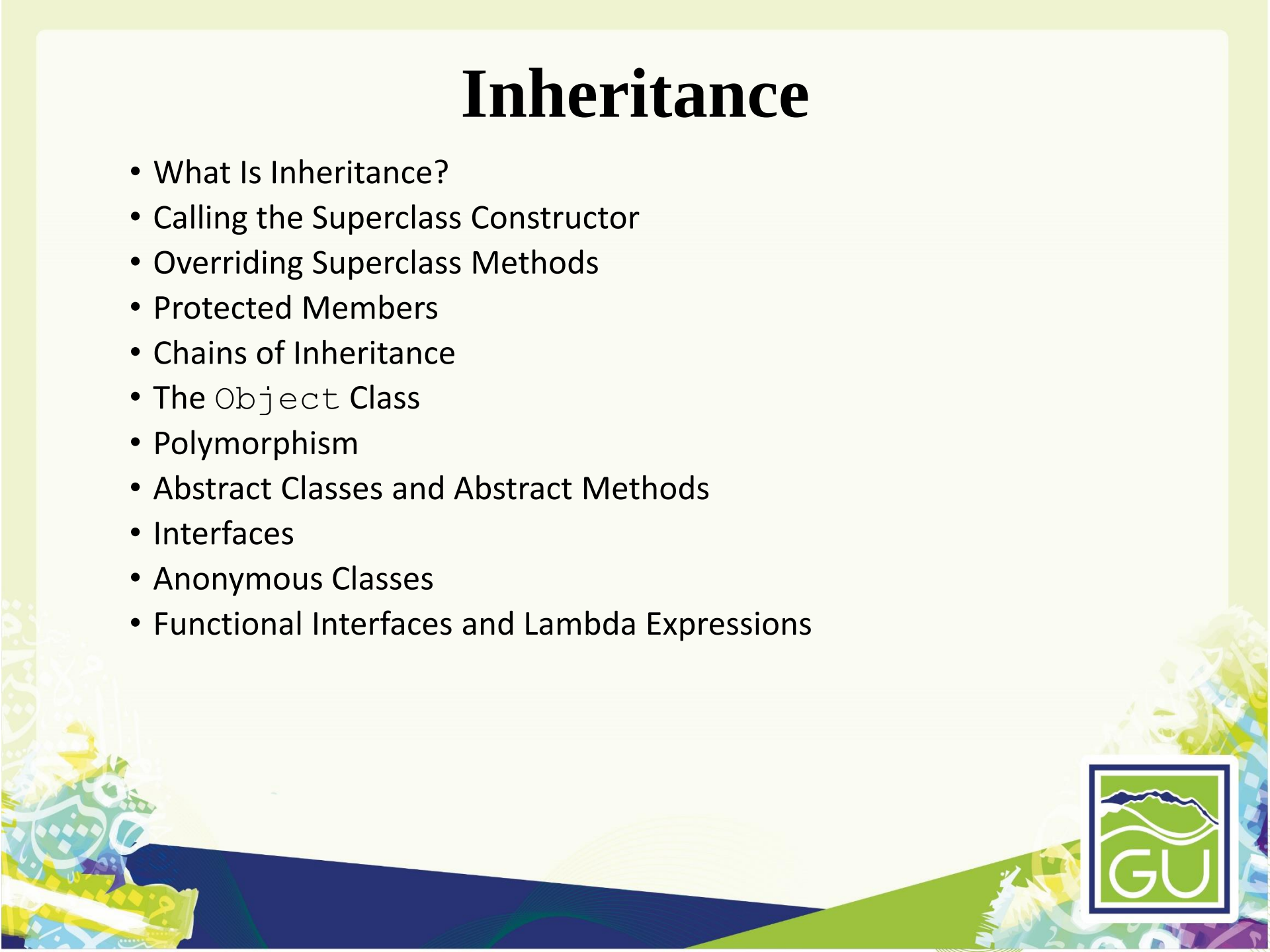
- If a method with the signature:

  ```
  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.

Inheritance

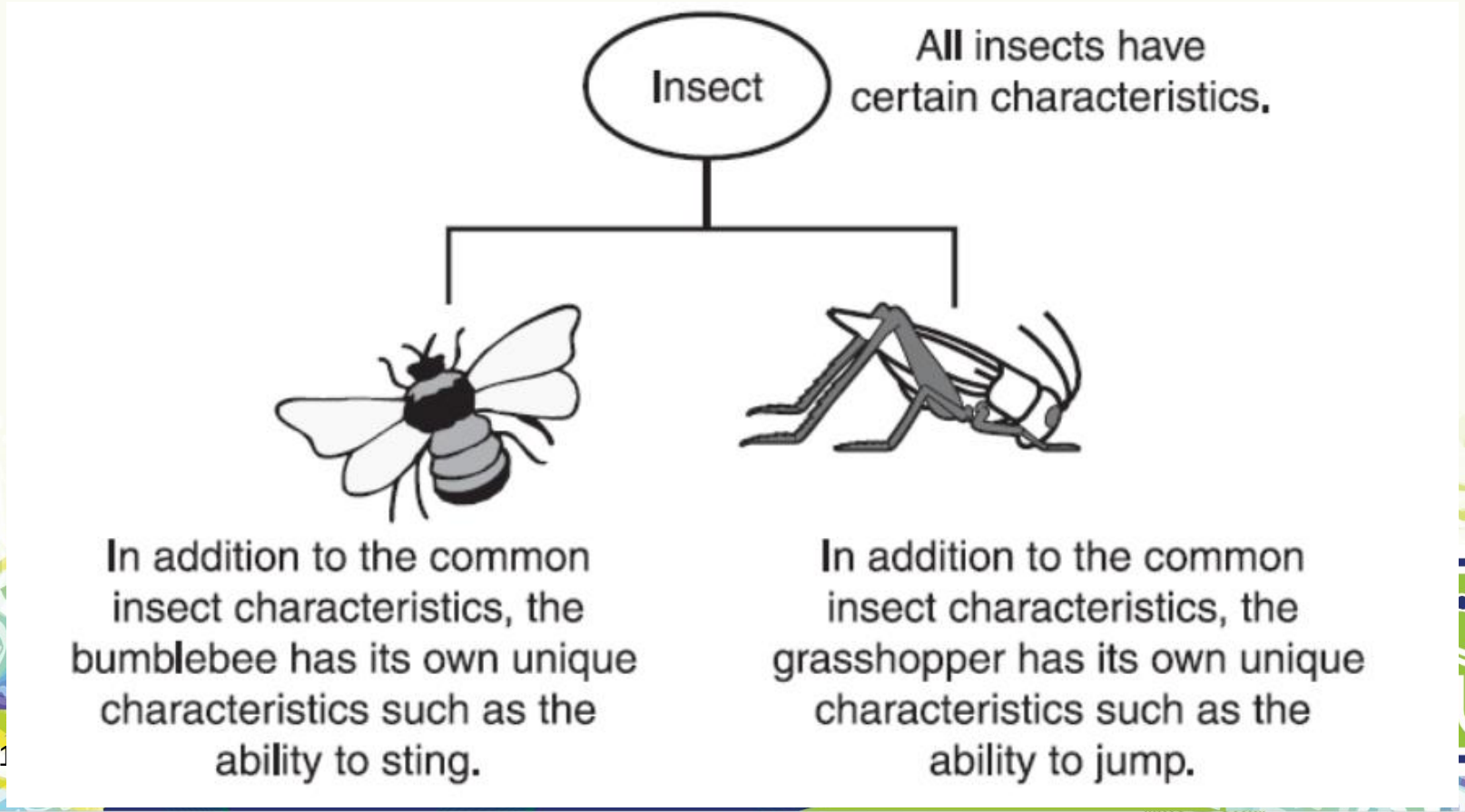
- What Is Inheritance?
- Calling the Superclass Constructor
- Overriding Superclass Methods
- Protected Members
- Chains of Inheritance
- The `Object` Class
- Polymorphism
- Abstract Classes and Abstract Methods
- Interfaces
- Anonymous Classes
- Functional Interfaces and Lambda Expressions



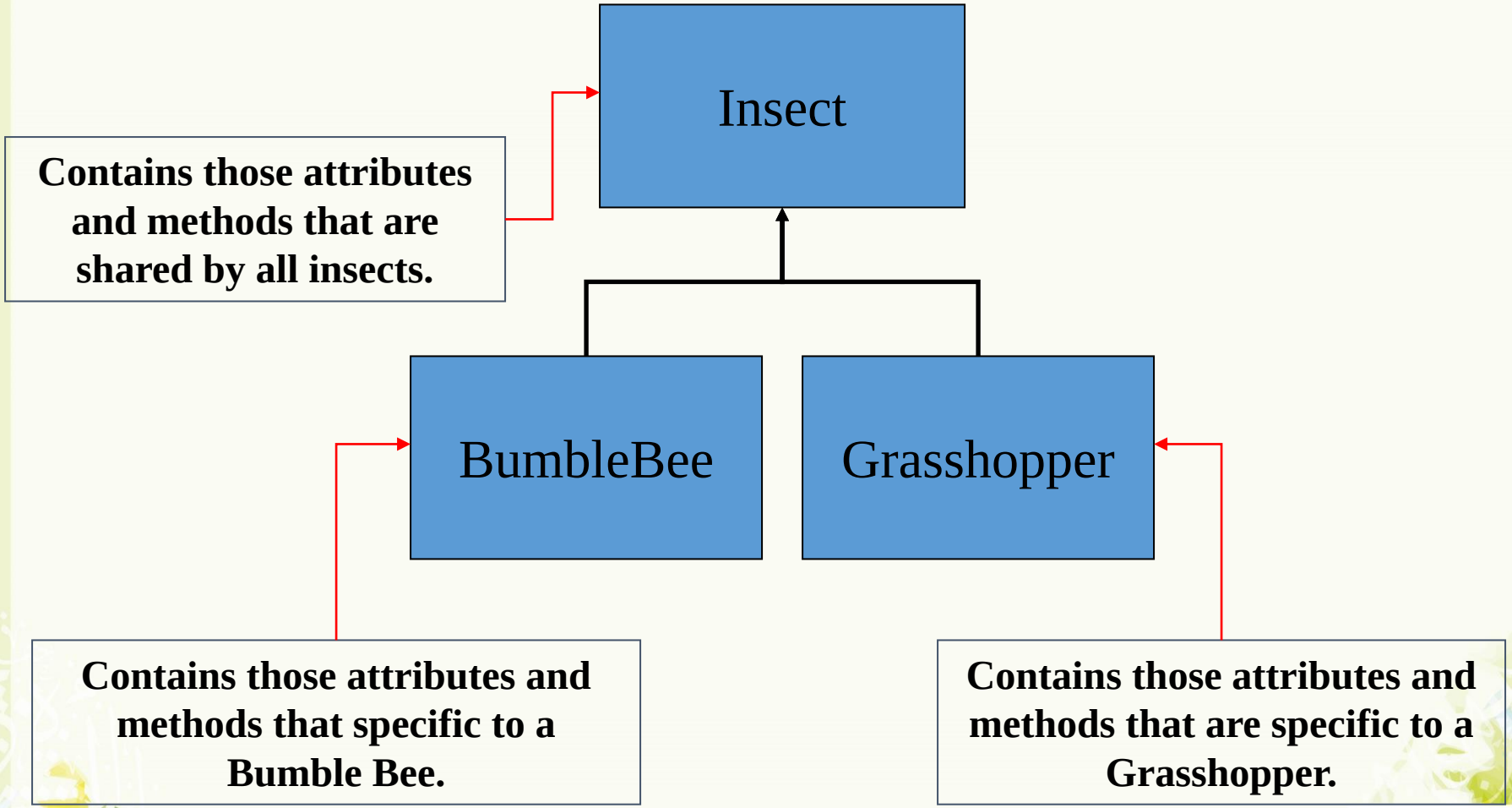
What is Inheritance?

Generalization vs. Specialization

- Real-life objects are typically specialized versions of other more general objects.
- The term “insect” describes a very general type of creature with numerous characteristics.



Inheritance



The “is a” Relationship

- The relationship between a superclass and an inherited class is called an “is a” relationship.
  - A grasshopper “is a” insect.
  - A poodle “is a” dog.
  - A car “is a” vehicle.
- A specialized object has:
  - all of the characteristics of the general object, plus
  - additional characteristics that make it special.
- In object-oriented programming, *inheritance* is used to create an “is a” relationship among classes.

The “is a” Relationship

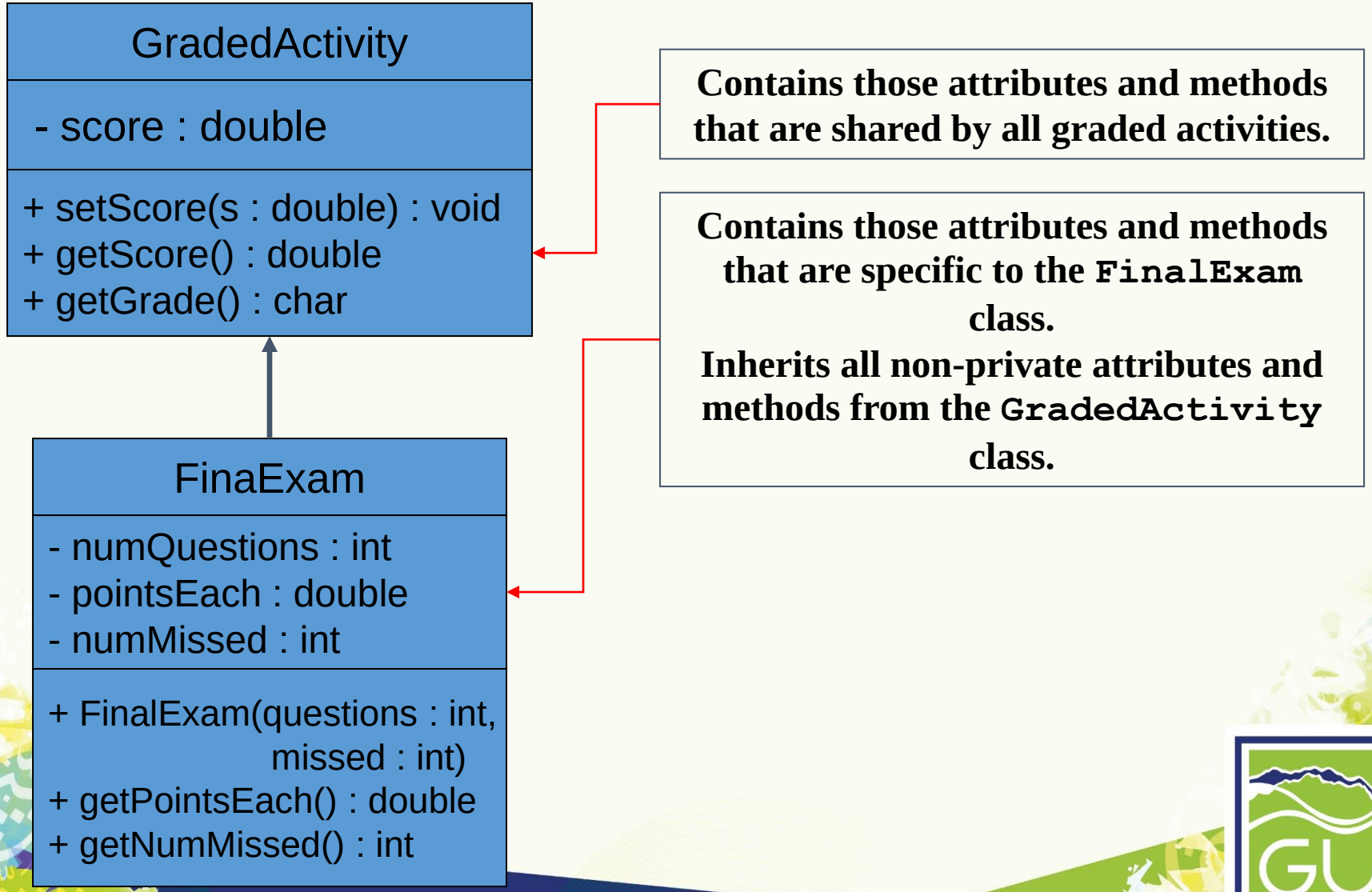
- We can *extend* the capabilities of a class.
- Inheritance involves a superclass and a subclass.
  - The *superclass* is the general class and
  - the *subclass* is the specialized class.
- The subclass is based on, or extended from, the superclass.
  - Superclasses are also called *base classes*, and
  - subclasses are also called *derived classes.*
- The relationship of classes can be thought of as *parent classes* and *child classes*.

Inheritance

- The subclass inherits fields and methods from the superclass without any of them being rewritten.
- New fields and methods may be added to the subclass.
- The Java keyword, *extends*, is used on the class header to define the subclass.

```
public class FinalExam extends GradedActivity
```

The `GradedActivity` Example



Inheritance, Fields and Methods

- Members of the superclass that are marked *private*:
  - are not inherited by the subclass,
  - exist in memory when the object of the subclass is created
  - may only be accessed from the subclass by public methods of the superclass.
- Members of the superclass that are marked *public*:
  - are inherited by the subclass, and
  - may be directly accessed from the subclass.

Inheritance, Fields and Methods

- When an instance of the subclass is created, the non-private methods of the superclass are available through the subclass object.

```
FinalExam exam = new FinalExam();
exam.setScore(85.0);
System.out.println("Score = "
                   + exam.getScore());
```

- Non-private methods and fields of the superclass are available in the subclass.

```
setScore(newScore);
```

Inheritance and Constructors

- Constructors are not inherited.
- When a subclass is instantiated, the superclass default constructor is executed first.

```
public class SuperClass1
{
    /**
       Constructor
    */

    public SuperClass1()
    {
       System.out.println("This is the " +
                  "superclass constructor.");
    }
}
```

```
public class SubClass1 extends SuperClass1
{
    /**
       Constructor
    */

    public SubClass1()
    {
       System.out.println("This is the " +
                  "subclass constructor.");
    }
}
```

```
public class ConstructorDemo1
{
    public static void main(String[] args)
    {
        SubClass1 obj = new SubClass1();
    }
}
```

Program Output

```
This is the superclass constructor.
This is the subclass constructor.
```

In an inheritance relationship, the subclass inherits members from the superclass, not the other way around. This means it is not possible for a superclass to call a subclass's method.

The Superclass's Constructor

- The `super` keyword refers to an object's superclass.
- The superclass constructor can be explicitly called from the subclass by using the `super` keyword.
- If a parameterized constructor is defined in the superclass,
  - the superclass **must** provide a no-arg constructor, or
    - subclasses must provide a constructor, and subclasses must call a superclass constructor.
- Calls to a superclass constructor must be **the first java statement** in the subclass constructors.

```
public class SuperClass2
{
   /**
      Constructor #1
   */

   public SuperClass2()
   {
    System.out.println("This is the superclass " +
                       "no-arg constructor.");
  }

  /**
     Constructor #2
  */

  public SuperClass2(int arg)
  {
     System.out.println("The following argument " +
                        "was passed to the superclass " +
                        "constructor: " + arg);
  }
}
```

Program Output

```
The following argument was passed to the superclass constructor: 10
This is the subclass constructor.
```

```
public class SubClass2 extends SuperClass2
{
    /**
        Constructor
    */

    public SubClass2()
    {
       super(10);
       System.out.println("This is the " +
                          "subclass constructor.");
    }
}
```

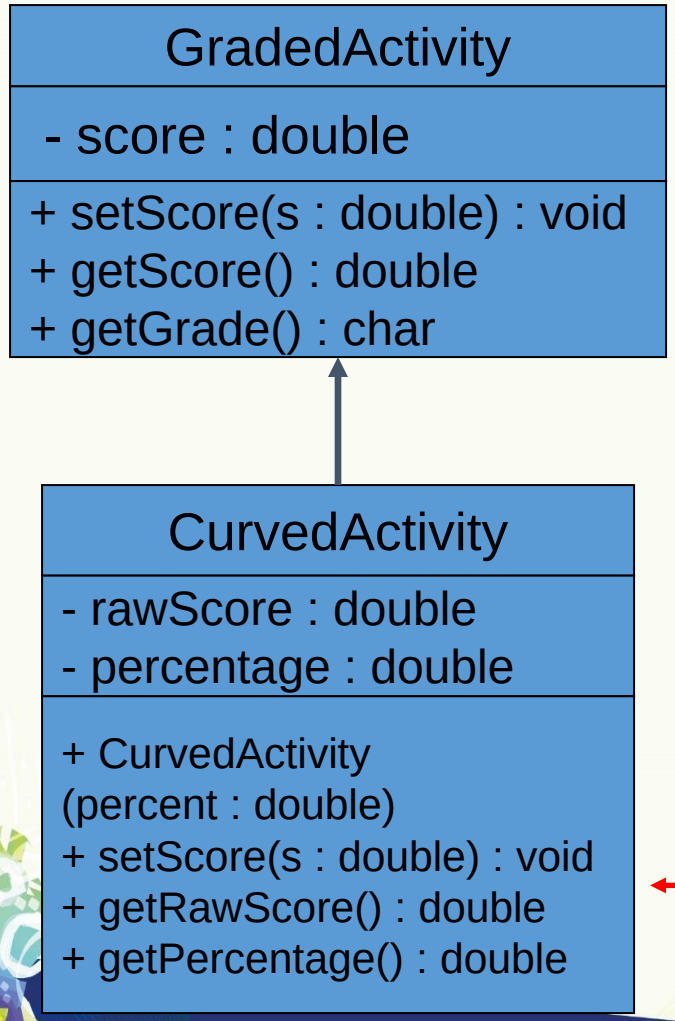
Summary of Constructor Issues in Inheritance

- The superclass constructor always executes **before** the subclass constructor.
- You can write a super statement that **calls** a superclass constructor, but **only** in the subclass's constructor. You cannot call the superclass constructor from any other method.
- If a super statement that calls a superclass constructor appears in a subclass **constructor**, it must be the **first statement**.
- If a subclass constructor does not explicitly call a superclass constructor, Java will **automatically** call super() just before the code in the subclass's constructor executes.
- If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it **must** call one of the constructors that the superclass does have.

Overriding Superclass Methods

- A subclass may have a method with the same signature as a superclass method.
- The subclass method overrides the superclass method.
- This is known as *method overriding*.

Overriding Superclass Methods



This method is a more specialized version of the `setScore` method in the superclass, `GradedActivity`.

Overriding Superclass Methods

- Recall that a method's *signature* consists of:
  - the method's name
  - the data types method's parameters in the order that they appear.
- A subclass method that overrides a superclass method must have the same signature as the superclass method.
- An object of the subclass invokes the subclass's version of the method, not the superclass's.
- The `@Override` annotation should be used just before the subclass method declaration.
  - This causes the compiler to display a error message if the method fails to correctly override a method in the superclass.

Overriding Superclass Methods

- An subclass method can call the overridden superclass method via the super keyword.

```
super.setScore(rawScore * percentage);
```

- There is a distinction between overloading a method and overriding a method.
- Overloading is when a method has the same name as one or more other methods, but with a different signature.
- When a method overrides another method, however, they both have the same signature.

Overriding Superclass Methods

- Both overloading and overriding can take place in an inheritance relationship.
- Overriding can only take place in an inheritance relationship.

```
public class SuperClass3
{
   /**
      This method displays an int.
      @param arg An int.
   */

   public void showValue(int arg)
   {
      System.out.println("SUPERCLASS: " +
                         "The int argument was " + arg);
   }

   /**
      This method displays a String.
      @param arg A String.
   */

   public void showValue(String arg)
   {
      System.out.println("SUPERCLASS: " +
                         "The String argument was " + arg);
   }
}
```

```
public class SubClass3 extends SuperClass3
{
   /**
      This method overrides one of the superclass methods.
      @param arg An int.
   */

   @Override
   public void showValue(int arg)
   {
      System.out.println("SUBCLASS: " +
            "The int argument was " + arg);
   }

   /**
      This method overloads the superclass methods.
      @param arg A double.
   */

   public void showValue(double arg)
   {
      System.out.println("SUBCLASS: " +
            "The double argument was " + arg);
   }
}
```

```
public class ShowValueDemo
{
   public static void main(String[] args)
   {
      // Create a SubClass3 object.
      SubClass3 myObject = new SubClass3();

      myObject.showValue(10);          // Pass an int.
      myObject.showValue(1.2);         // Pass a double.
      myObject.showValue("Hello");     // Pass a String.
   }
}
```

Program Output

```
SUBCLASS: The int argument was 10
SUBCLASS: The double argument was 1.2
SUPERCLASS: The String argument was Hello
```

Preventing a Method from Being Overridden

- The `final` modifier will prevent the overriding of a superclass method in a subclass.

```
public final void message()
```

- If a subclass attempts to override a final method, the compiler generates an error.
- This ensures that a particular superclass method is used by subclasses rather than a modified version of it.

Protected Members

- Protected members of class:
  - may be accessed by methods in a subclass, and
  - by methods in the same package as the class.
- Java provides a third access specification, `protected`.
- A *protected* member's access is somewhere between *private* and *public*.
- Example:
  - GradedActivity2.java
  - FinalExam2.java
  - ProtectedDemo.java

```
public class GradedActivity2
{
   protected double score;  // Numeric score

   /**
      The setScore method sets the score field.
      @param s The value to store in score.
   */

   public void setScore(double s)
   {
      score = s;
   }

   /**
      The getScore method returns the score.
      @return The value stored in the score field.
   */

   public double getScore()
   {
      return score;
```

```
}

/**
   The getGrade method returns a letter grade
   determined from the score field.
   @return The letter grade.
*/

public char getGrade()
{
   char letterGrade;

   if (score >= 90)
      letterGrade = 'A';
   else if (score >= 80)
      letterGrade = 'B';
   else if (score >= 70)
      letterGrade = 'C';
   else if (score >= 60)
      letterGrade = 'D';
   else
      letterGrade = 'F';

   return letterGrade;
}
}
```

```
public class FinalExam2 extends GradedActivity2
{
   private int numQuestions;     // Number of questions
 private double pointsEach;    // Points for each question
 private int numMissed;        // Number of questions missed
 public FinalExam2(int questions, int missed)
 {
    double numericScore;         // To hold a numeric score

    // Set the numQuestions and numMissed fields.
    numQuestions = questions;
    numMissed = missed;

    // Calculate the points for each question and
    // the numeric score for this exam.
    pointsEach = 100.0 / questions;
    numericScore = 100.0 - (missed * pointsEach);

    // Call the inherited setScore method to
    // set the numeric score.
    setScore(numericScore);

    // Adjust the score.
    adjustScore();
 }
```

```
public double getPointsEach()
{
   return pointsEach;
}
public int getNumMissed()
{
   return numMissed;
}
  private void adjustScore()
  {
     double fraction;

     // Get the fractional part of the score.
     fraction = score - (int) score;

     // If the fractional part is .5 or greater,
     // round the score up to the next whole number.
     if (fraction >= 0.5)
        score = score + (1.0 - fraction);
  }
}
```

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the FinalExam2 class,
   which extends the GradedActivity2 class.
*/

public class ProtectedDemo
{
   public static void main(String[] args)
   {
      String input;         // To hold input
      int questions;        // Number of questions
      int missed;           // Number of questions missed
```

```
        // Get the number of questions on the exam.
        input = JOptionPane.showInputDialog("How many " +
                        "questions are on the final exam?");
        questions = Integer.parseInt(input);

        // Get the number of questions the student missed.
        input = JOptionPane.showInputDialog("How many " +
                        "questions did the student miss?");
        missed = Integer.parseInt(input);

        // Create a FinalExam object.
        FinalExam2 exam = new FinalExam2(questions, missed);

        // Display the test results.
        JOptionPane.showMessageDialog(null,
             "Each question counts " + exam.getPointsEach() +
             " points.\nThe exam score is " +
             exam.getScore() + "\nThe exam grade is " +
             exam.getGrade());

        System.exit(0);
    }
}
```

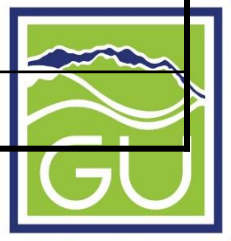
Protected Members

- Using `protected` instead of `private` makes some tasks easier.
- However, any class that is derived from the class, or is in the same package, has unrestricted access to the protected member.
- It is always better to make all fields `private` and then provide `public` methods for accessing those fields.
- **If no access specifier for a class member is provided, the class member is given *package access* by default.**
- Any method in the same package may access the member.

Access Specifiers

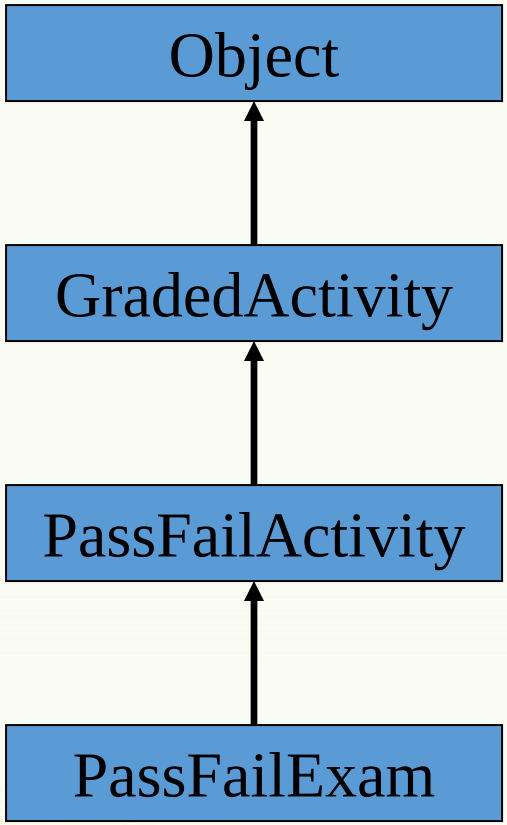
Access Modifier	Accessible to a subclass inside the same package?	Accessible to all other classes inside the same package?
default (no modifier)	Yes	Yes
Public	Yes	Yes
Protected	Yes	Yes
Private	No	No

Access Modifier	Accessible to a subclass outside the package?	Accessible to all other classes outside the package?
default (no modifier)	No	No
Public	Yes	Yes
Protected	Yes	No
Private	No	No



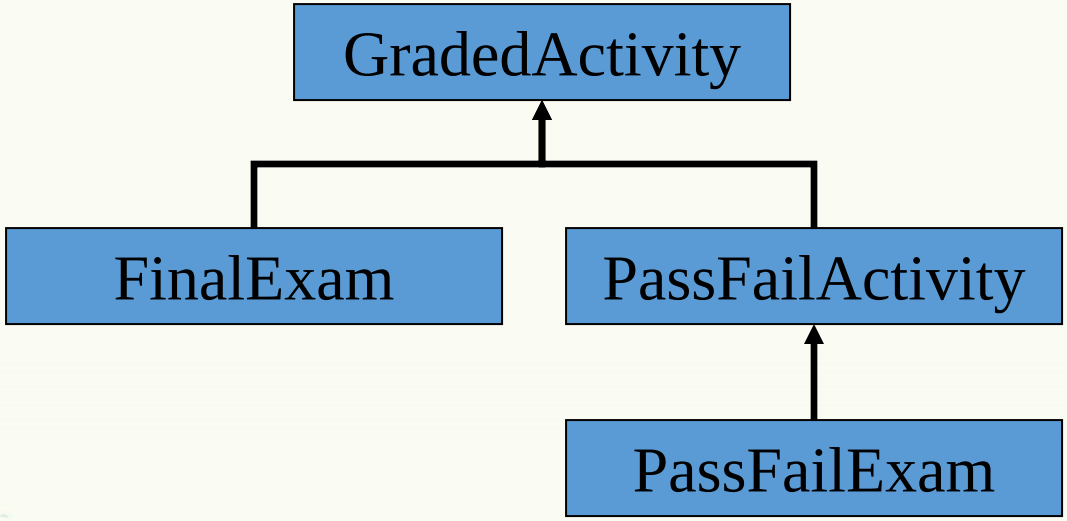
Chains of Inheritance

- A superclass can also be derived from another class.



Chains of Inheritance

- Classes often are depicted graphically in a *class hierarchy*.
- A class hierarchy shows the inheritance relationships between classes.



The `Object` Class

- All Java classes are directly or indirectly derived from a class named `Object`.
- `Object` is in the `java.lang` package.
- Any class that does not specify the `extends` keyword is automatically derived from the `Object` class.

```
public class MyClass
{
   // This class is derived from Object.
}
```

- Ultimately, every class is derived from the `Object` class.

The `Object` Class

- Because every class is directly or indirectly derived from the `Object` class:
  - every class inherits the `Object` class's members.
    - example: `toString` and `equals`.
- In the `Object` class, the `toString` method returns a string containing the object's class name and a hash of its memory address.
- The `equals` method accepts the address of an object as its argument and returns true if it is the same as the calling object's address.
- Example: ObjectMethods.java

Polymorphism

- Out of scope of the course.

Interfaces

- Out of scope of the course.

Research project

1. Select a specific domain like hospital, restaurant, pharmacy, university, airport, company, or etc.
2. Make abstraction of the domain in the form of the possible classes that will be in this domain. Design the classes using UML.
3. For each class, collect the private data and the public behavior of it and provide an implementation for the body of each class.
4. Provide a little application of the usage of these classes.

Thank you.