
جـامعة الجـلالة
GALALA UNIVERSITY

# CSE110 Principles of Programming

## Lecture 8: Object Oriented Programming (A first look)

Professor Shaker El-Sappagh

Shaker.elsappagh@gu.edu.eg

Fall 2023

# Chapter Topics

- Objects and Classes
- Writing a Simple Class, Step by Step
- Instance Fields and Methods
- Constructors
- Passing Objects as Arguments
- Overloading Methods and Constructors
- Scope of Instance Fields
- Packages and `import` Statements

# Think about different environments

1. Hospital
2. School
3. University
4. Airport
5. Forest
6. Pharmacy
7. …

# OOP principles

1. Abstraction
2. Encapsulation
3. Information hiding and interfaces
4. Polymorphism
5. Inheritance
6. …

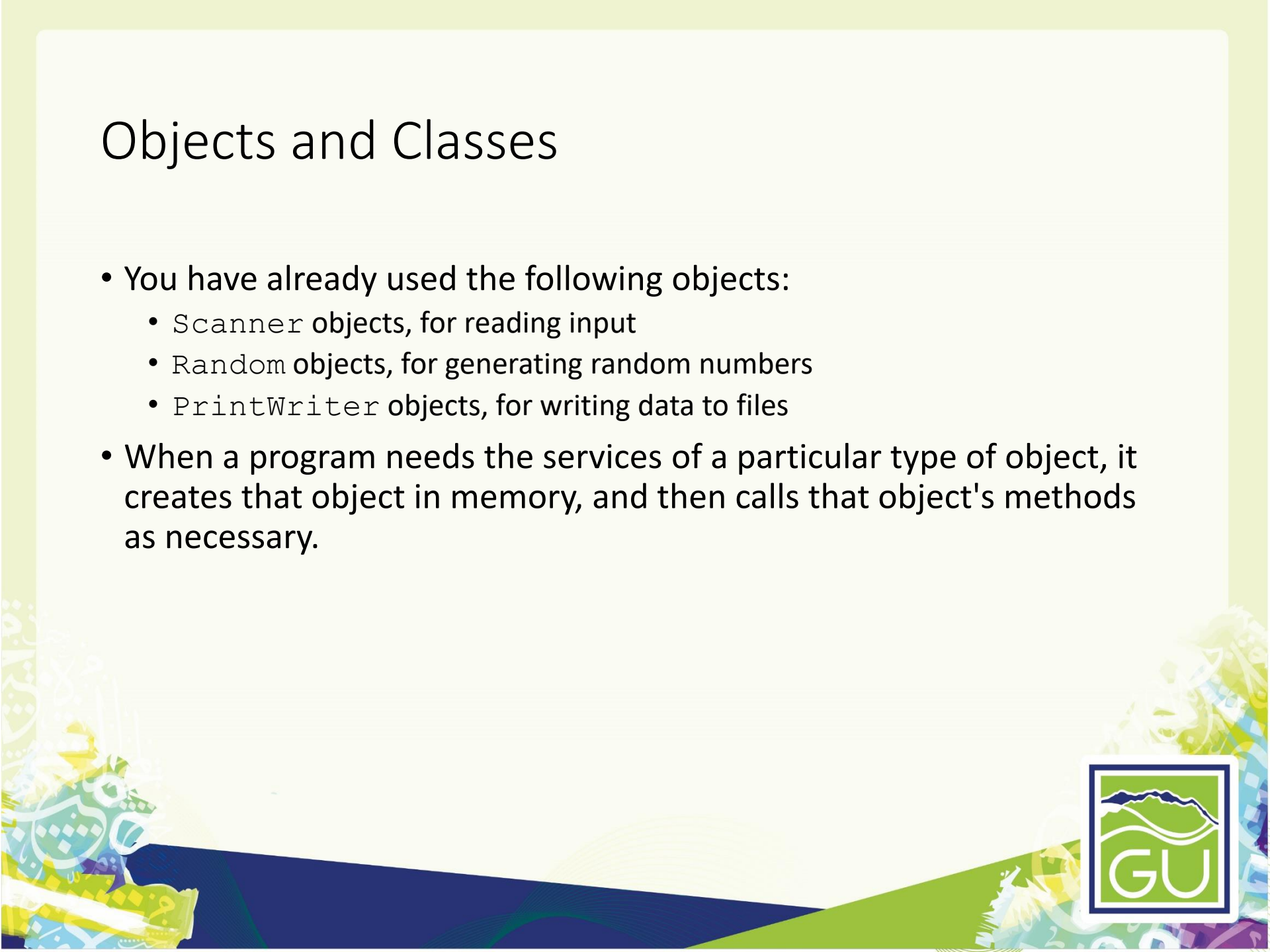
*You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its methods. The implementation is hidden.*

# 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*.

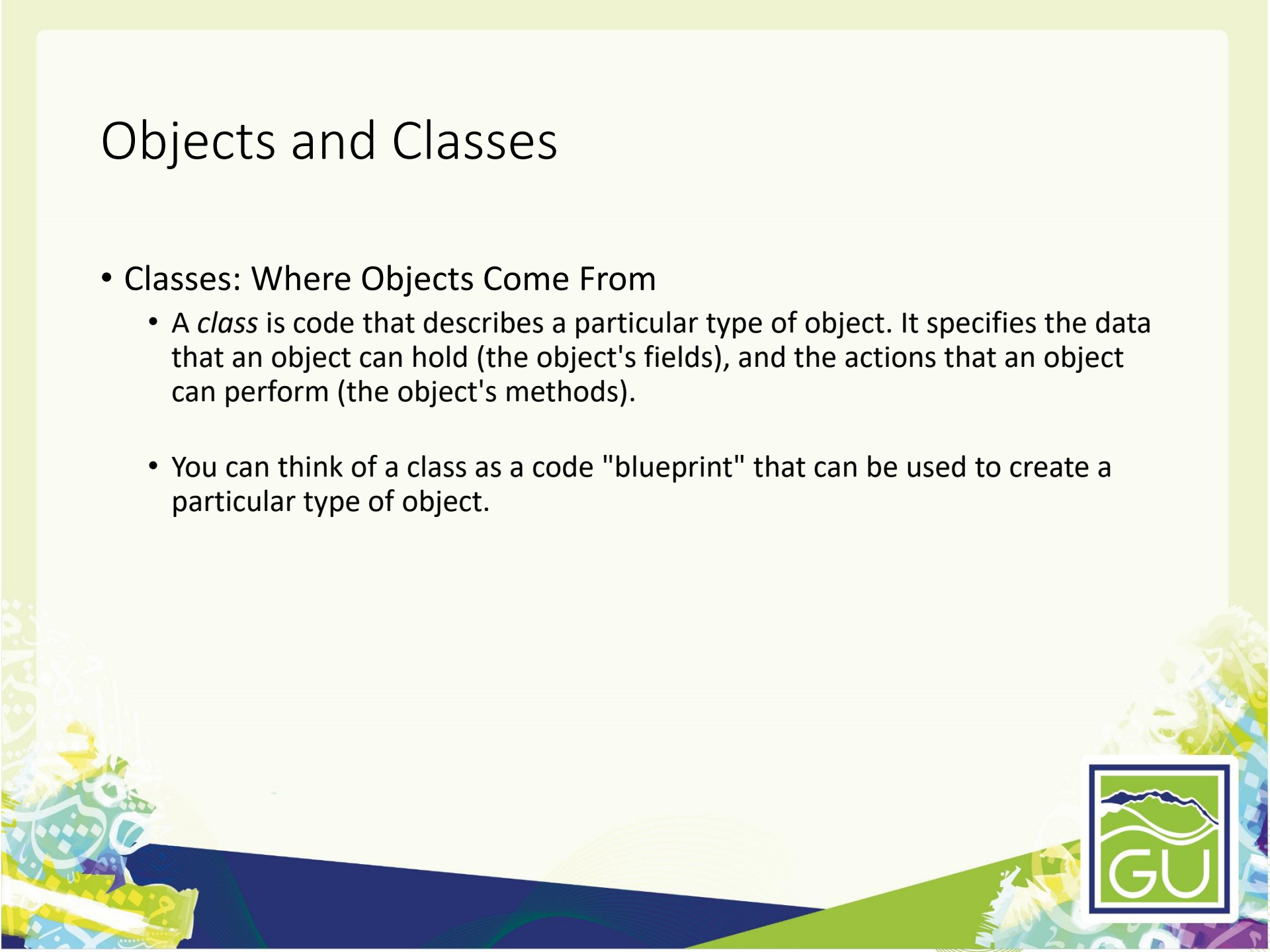
# Objects and Classes

- You have already used the following 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.



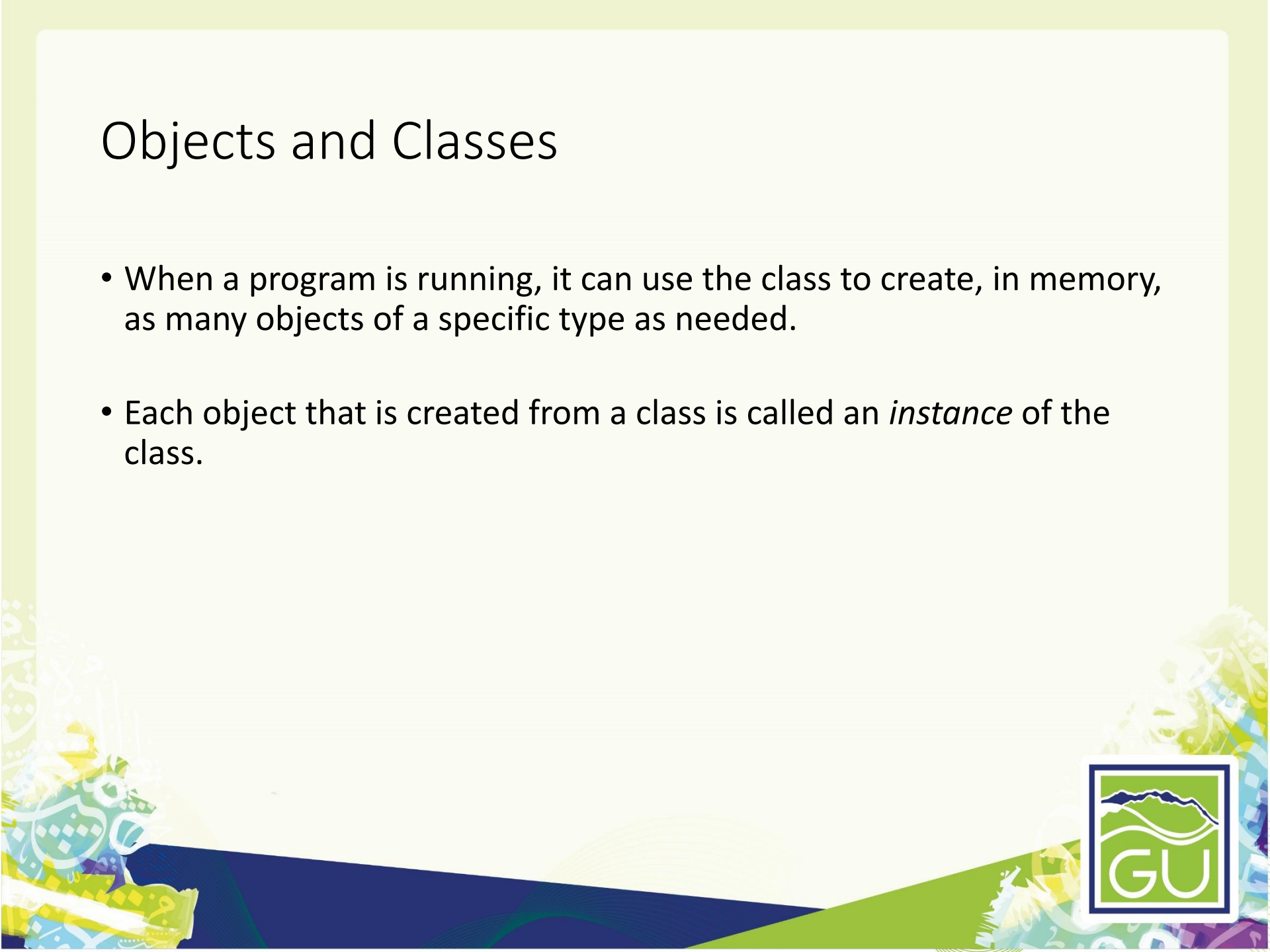
# Objects and Classes

- 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.

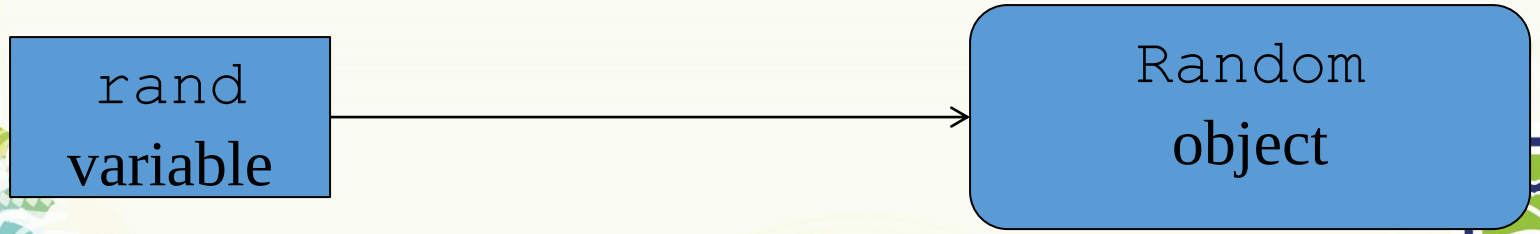
# Objects and Classes

Example:

This expression creates a `Random` object in memory.

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

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

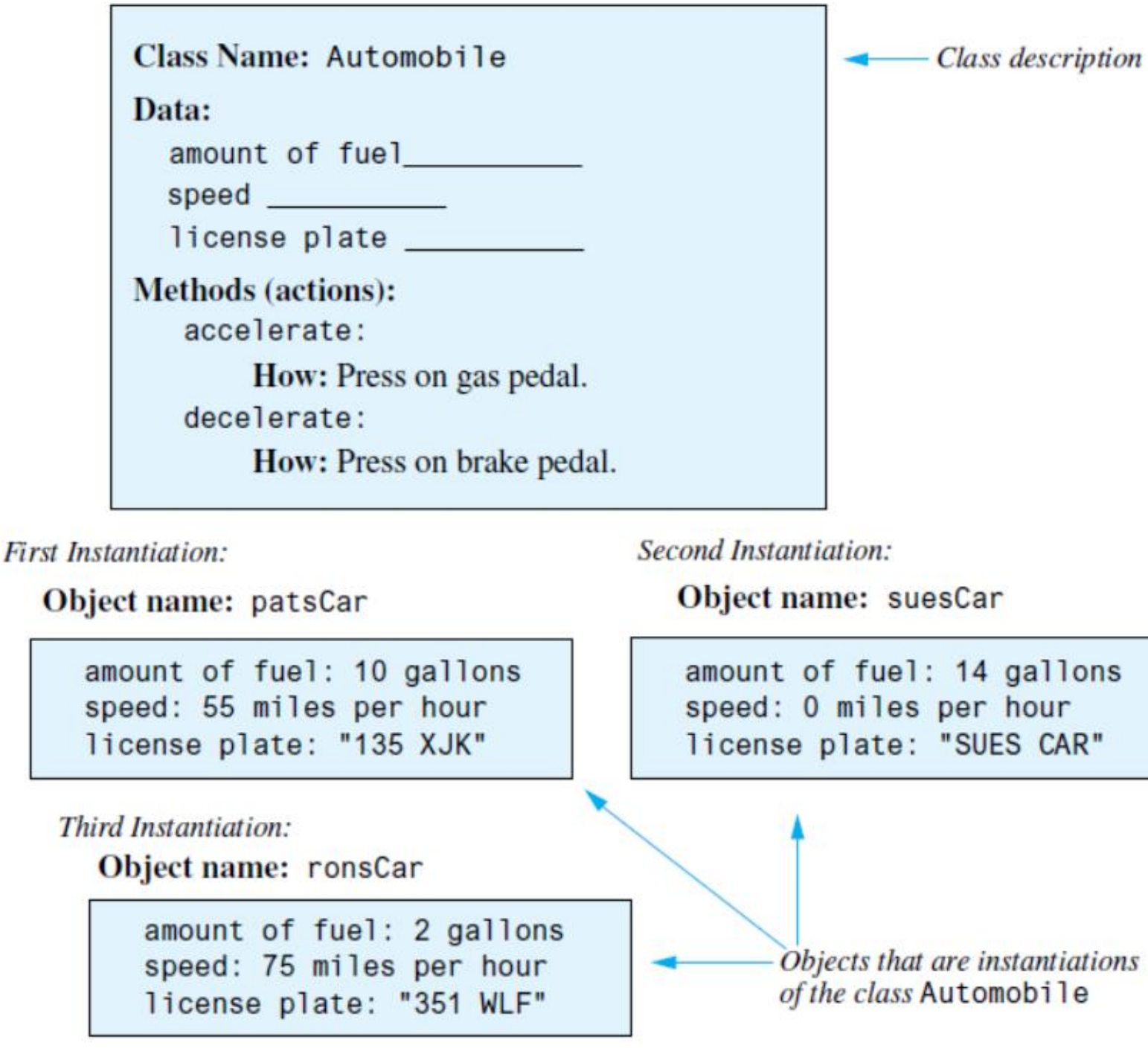


# Objects and Classes

- The Java API provides many classes
  - So far, the classes that you have created objects from are provided by the Java API.
  - Examples:
    - `Scanner`
    - `Random`
    - `PrintWriter`

```
import java.util.Scanner; // Needed for the Scanner class
import java.util.Random;  // Needed for the Random class
import java.io.*;         // Needed for file I/O classes

/**
  This program writes random numbers to a file.
*/

public class ObjectDemo
{
   public static void main(String[] args) throws IOException
   {
      int maxNumbers;   // Max number of random numbers
      int number;       // To hold a random number

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create a Random object to generate random numbers.
      Random rand = new Random();
```

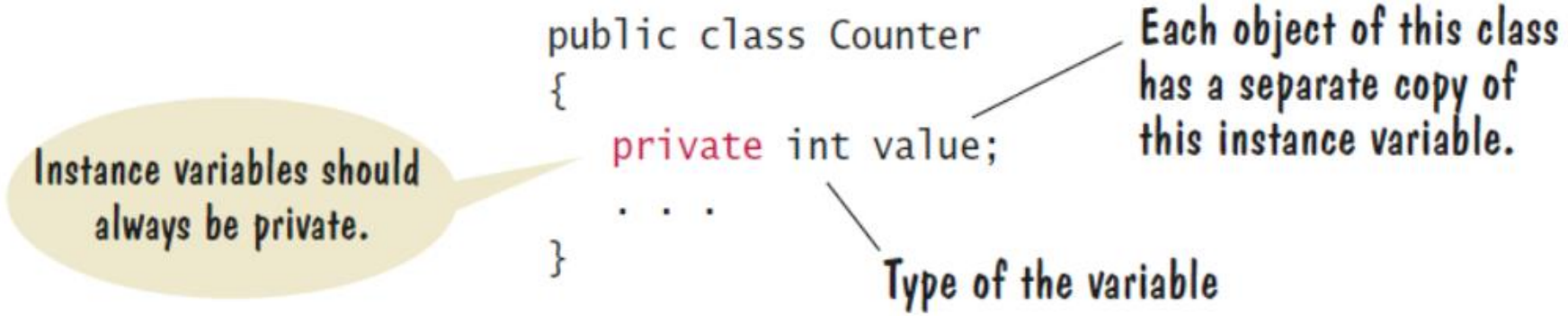
```
// Create a PrintWriter object to open the file.
PrintWriter outputFile = new PrintWriter("numbers.txt");

// Get the number of random numbers to write.
System.out.print("How many random numbers should I write? ");
maxNumbers = keyboard.nextInt();
      // Write the random numbers to the file.
      for (int count = 0; count < maxNumbers; count++)
      {
         // Generate a random integer.
         number = rand.nextInt();

         // Write the random integer to the file.
         outputFile.println(number);
      }

      // Close the file.
      outputFile.close();
      System.out.println("Done");
   }
}
```


Class Name: Automobile
Data:
amount of fuel
speed
license plate
Methods (actions):
accelerate:
How: Press on gas pedal.
decelerate:
How: Press on brake pedal.
Class description
First Instantiation:
Object name: patsCar
amount of fuel: 10 gallons
speed: 55 miles per hour
license plate: "135 XJK"
Second Instantiation:
Object name: suesCar
amount of fuel: 14 gallons
speed: 0 miles per hour
license plate: "SUES CAR"
Third Instantiation:
Object name: ronsCar
amount of fuel: 2 gallons
speed: 75 miles per hour
license plate: "351 WLF"
Objects that are instantiations of the class Automobile

# Writing a Class, Step by Step

*Syntax*

```
public class ClassName
{
   private typeName variableName;
   . . .
}
```




- Each object of a class has its own set of instance variables.
- An instance method can access the instance variables of the object on which it acts.

# Writing a Class, Step by Step

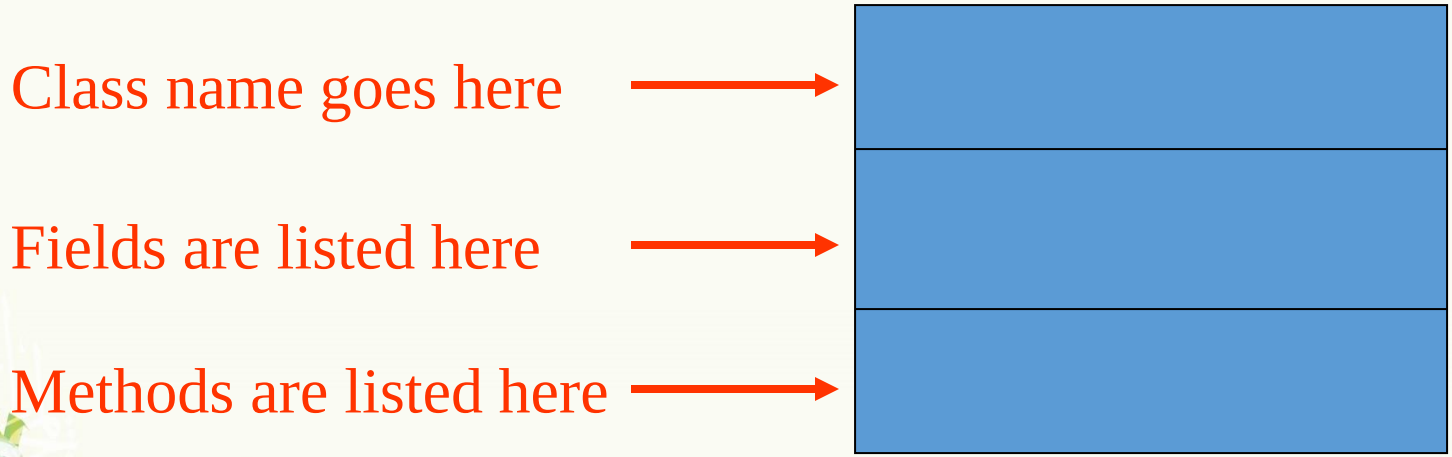
- A `Rectangle` object will have the following fields:
  - `length`. The length field will hold the rectangle's length.
  - `width`. The width field will hold the rectangle's width.

# Writing a Class, Step by Step

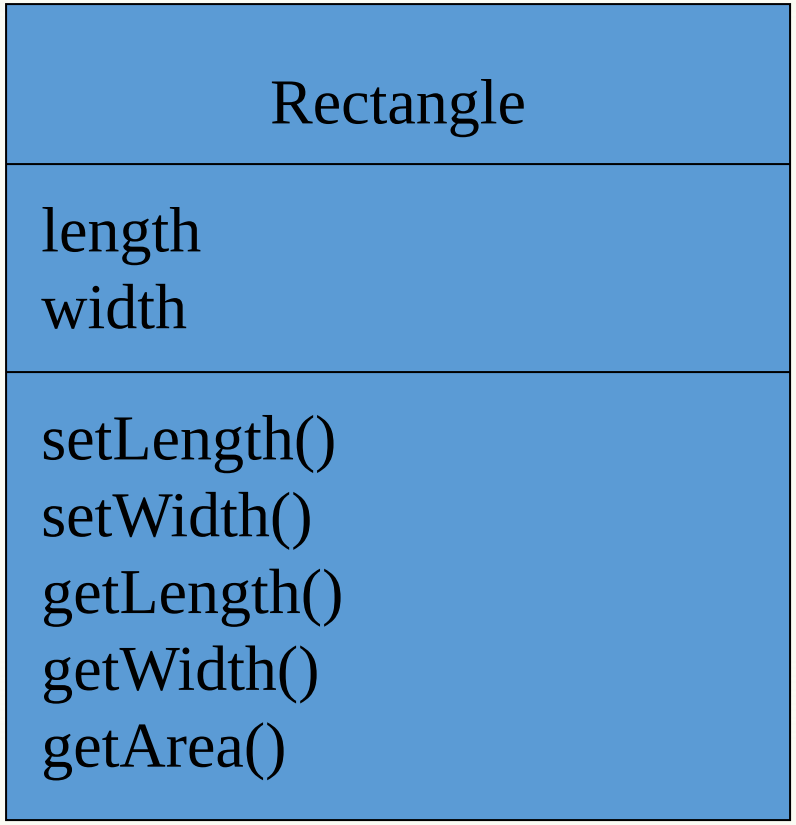
- The `Rectangle` class will also have the following methods:
  - **`setLength`**. The `setLength` method will store a value in an object's `length` field.
  - **`setWidth`**. The `setWidth` method will store a value in an object's `width` field.
  - **`getLength`**. The `getLength` method will return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

# Methods

*Syntax*

```
modifiers returnType methodName(parameterType parameterName, . . . )
{
   method body
}
```

```
public class CashRegister
{
   . . .
   public void addItem(double price)
   {
      itemCount++;
      totalPrice = totalPrice + price;
   }
   . . .
}
```

Explicit parameter

Instance variables of the implicit parameter

# Header for the `setLength` Method

Access specifier

Return Type

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

# 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;
}
```

Examples: Rectangle.java, LengthDemo.java

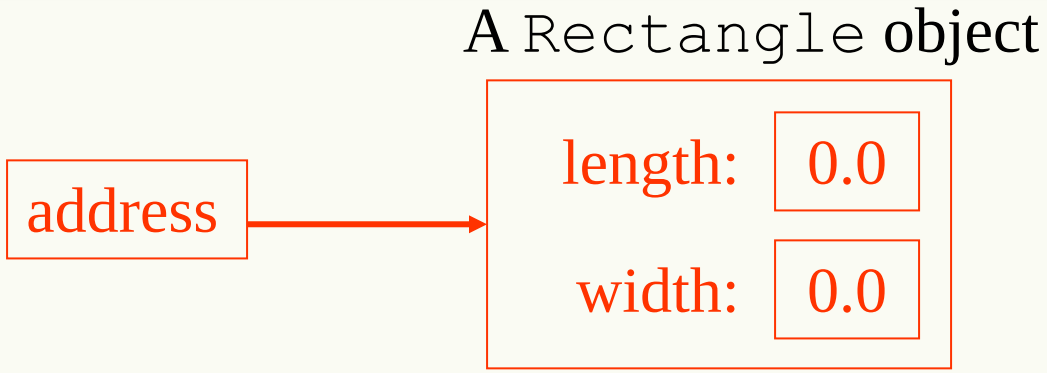
```
public class Rectangle
{
   private double length;
   private double width;

   /**
      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;
   }
}
```

```
public class LengthDemo
{
    public static void main(String[] args)
    {
        // Create a Rectangle object and assign its
        // address to the box variable.
       Rectangle box = new Rectangle();

       // Indicate what we are doing.
       System.out.println("Sending the value 10.0 " +
                          "to the setLength method.");

       // Call the box object's setLength method.
       box.setLength(10.0);

       // Indicate we are done.
       System.out.println("Done.");
    }
}
```

# Creating a `Rectangle` object

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

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



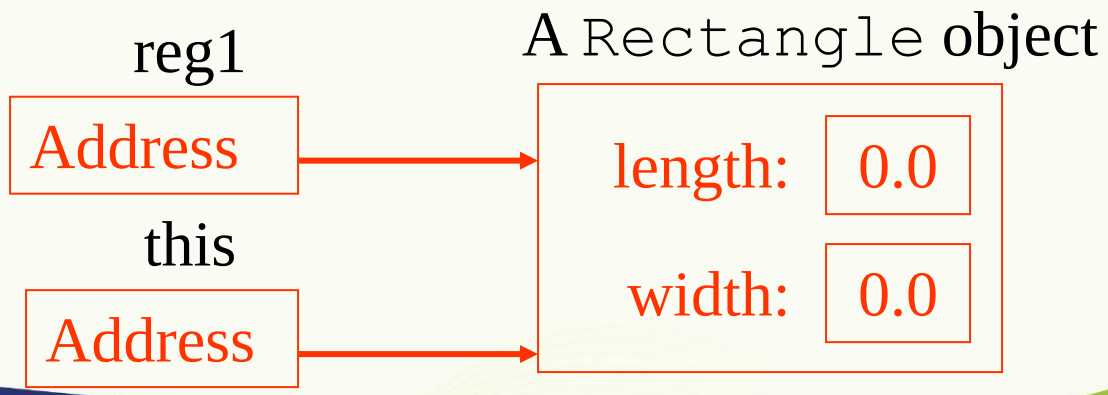
# Creating a `Rectangle` object

- **Shared References:** You can have two (or more) object variables that store references to the same object.
- **The null Reference:** An object reference can have the special value null if it refers to no object at all. It is common to use the null value to indicate that a value has never been set. For example,

  **String str = null;**
- **The this Reference:** Every instance method receives the implicit parameter in a variable called this. For example, consider the method call

  ```
  reg1.setLength(2.95);
  ```

  When the method is called, the parameter variable this refers to the same object as reg1.

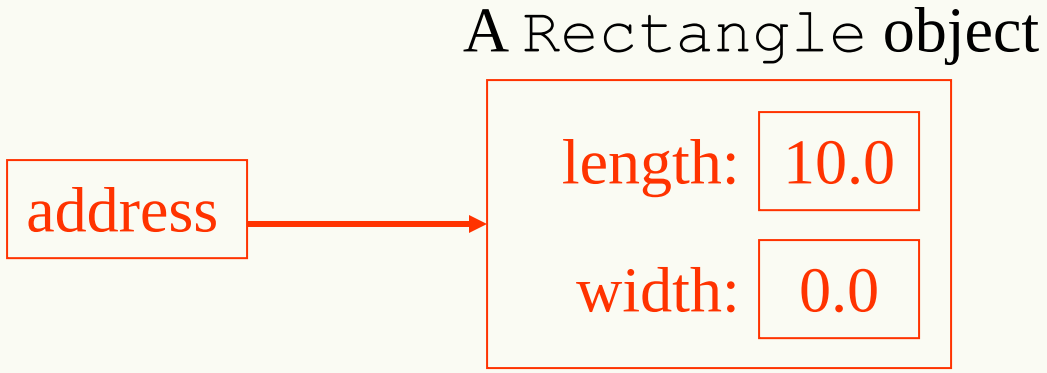


```
public class Student
{
    private int id;
    private String name;

    public Student(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
}
```

# 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.

Examples: Rectangle.java, LengthWidthDemo.java

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

   /**
      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;
   }

   /**
      The setWidth method stores a value in the
      width field.
      @param w The value to store in width.
   */

   public void setWidth(double w)
   {
      width = w;
   }
}
```

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

      // Call the object's setLength method, passing 10.0
      // as an argument.
      box.setLength(10.0);

      // Call the object's setWidth method, passing 20.0
      // as an argument.
      box.setWidth(20.0);

      // Display the object's length and width.
      System.out.println("The box's length is " +
                         box.getLength());
      System.out.println("The box's width is " +
                         box.getWidth());
   }
}
```

# 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

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

   /**
      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;
   }

   /**
      The setWidth method stores a value in the
      width field.
      @param w The value to store in width.
   */

   public void setWidth(double w)
   {
      width = w;
   }

   /**
      The getLength method returns a Rectangle
      object's length.
      @return The value in the length field.
   */

   public double getLength()
   {
      return length;
   }

   /**
      The getWidth method returns a Rectangle
      object's width.
      @return The value in the width field.
   */

   public double getWidth()
   {
      return width;
   }

   /**
      The getArea method returns a Rectangle
      object's area.
      @return The product of length times width.
   */

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

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

     // Set length to 10.0 and width to 20.0.
     box.setLength(10.0);
     box.setWidth(20.0);

     // Display the length.
     System.out.println("The box's length is " +
                        box.getLength());

     // Display the width.
     System.out.println("The box's width is " +
                        box.getWidth());

     // Display the area.
     System.out.println("The box's area is " +
                        box.getArea());
   }
}
```

# Accessor and Mutator Methods

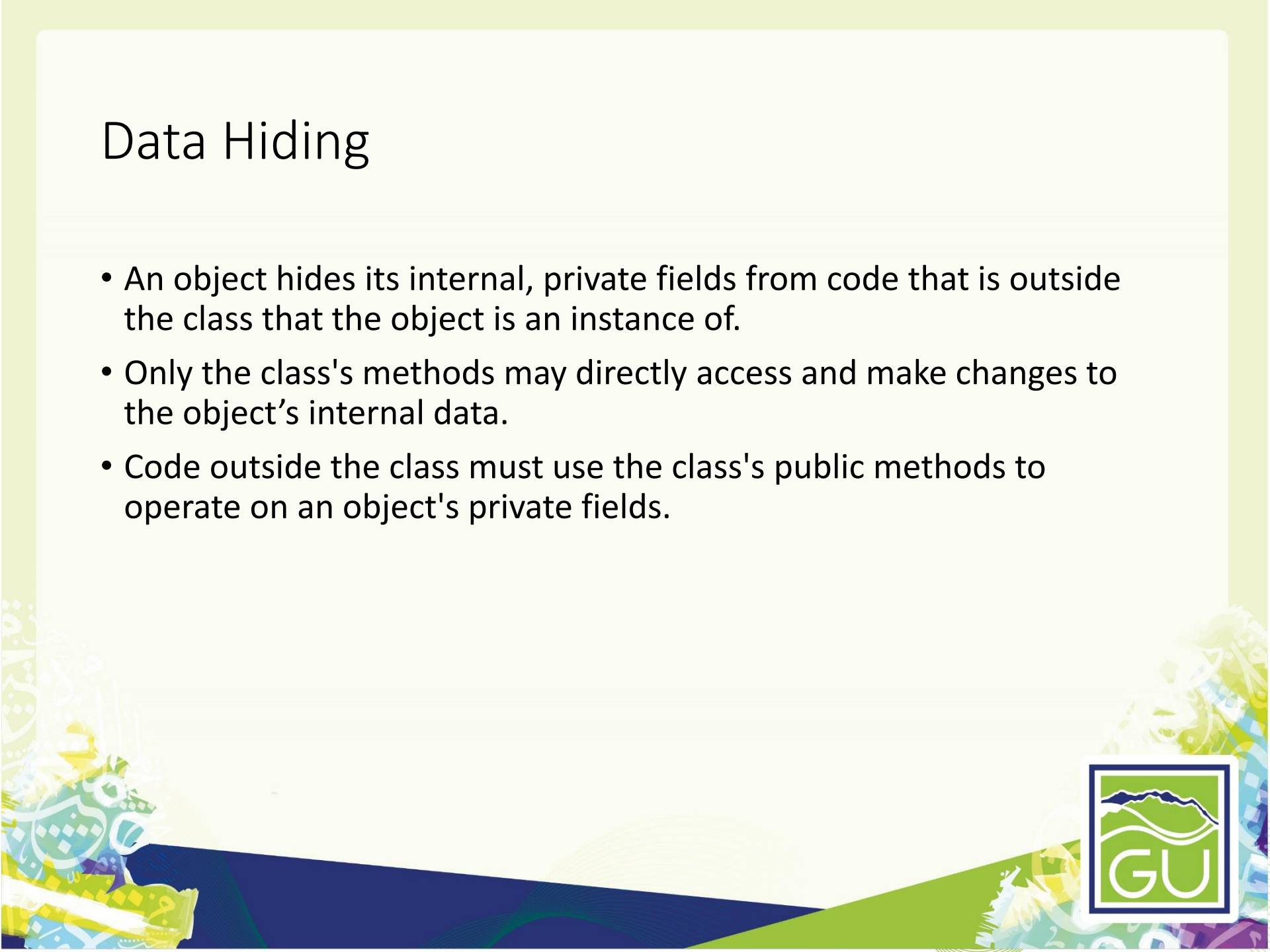
- Because of the concept of data hiding, fields in a class are private.
- The methods that retrieve the data of fields are called *accessors*.
- The methods that modify the data of fields are called *mutators*.
- Each field that the programmer wishes to be viewed by other classes needs an accessor.
-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

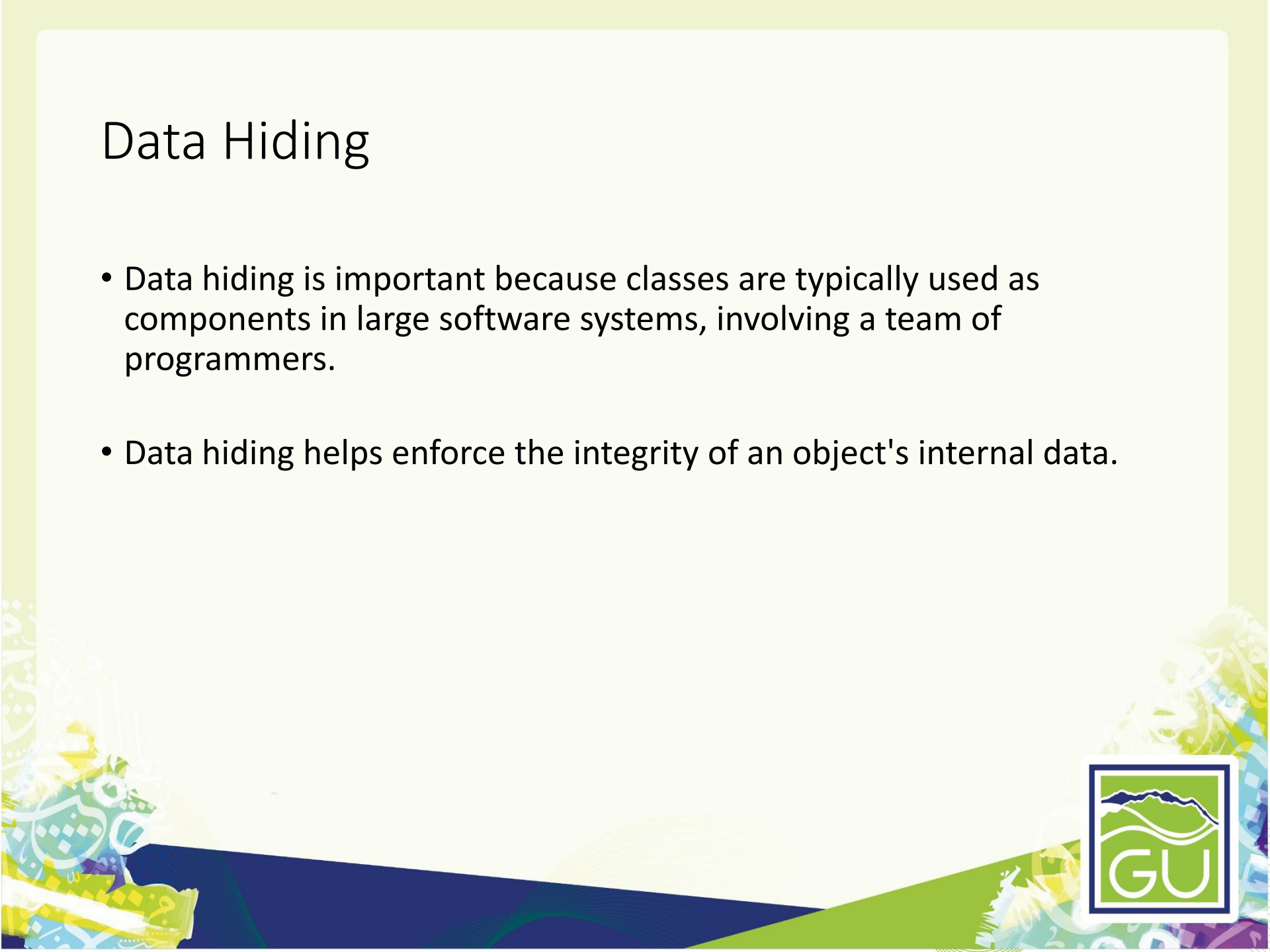
# Data Hiding

- 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

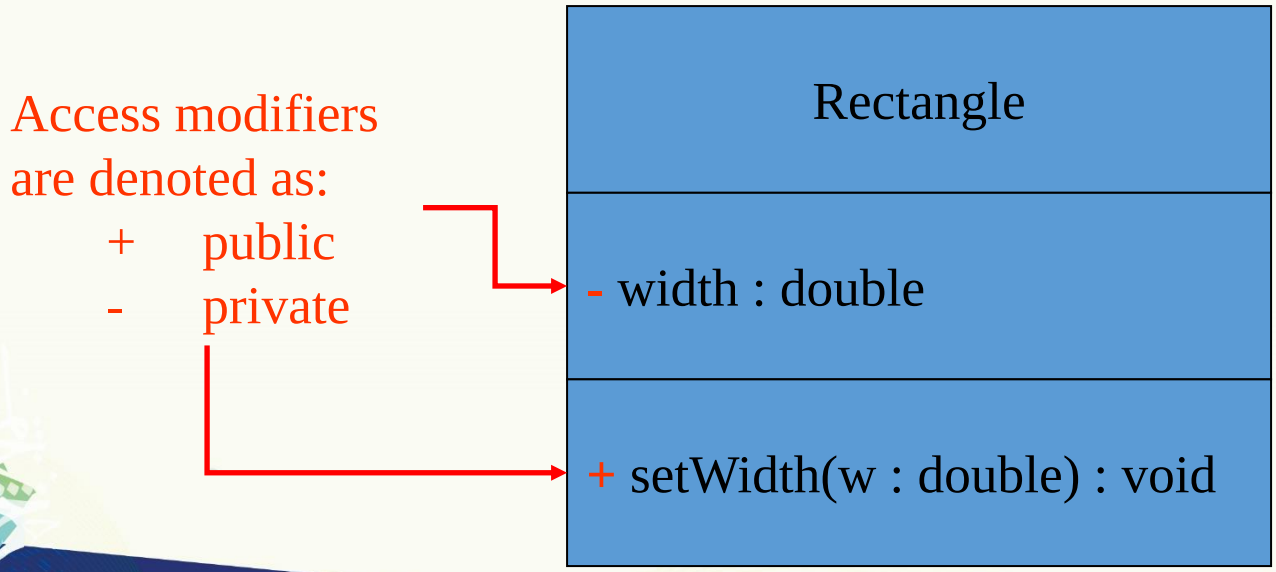
- Rather than use an `area` variable in a `Rectangle` class:

```
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.

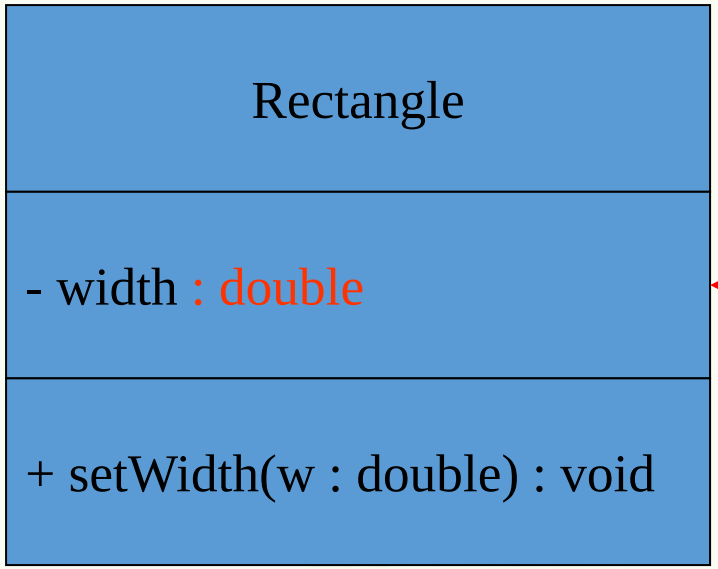
# UML Data Type and Parameter Notation

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



# UML Data Type and Parameter Notation

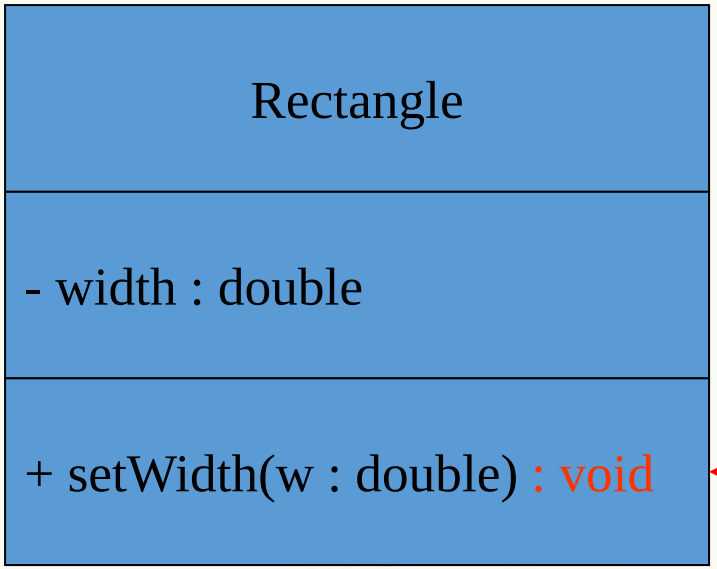
- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.




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.

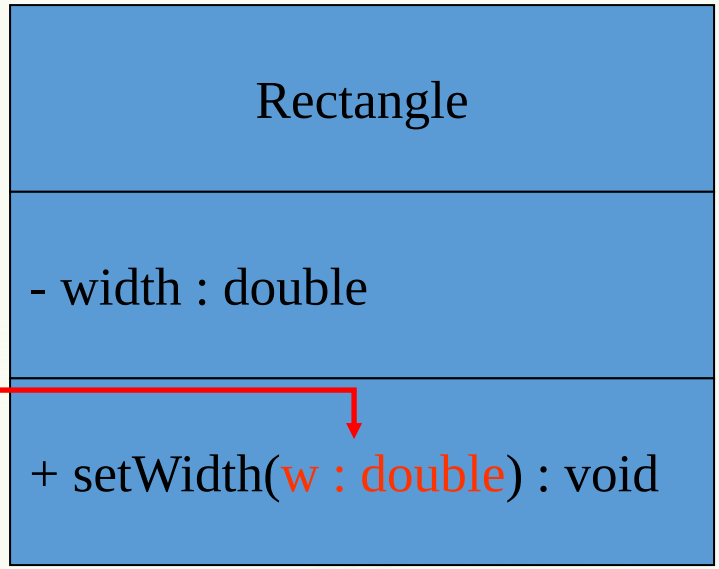



Method return types are placed after the method declaration name, separated by a colon.

# UML Data Type and Parameter Notation

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

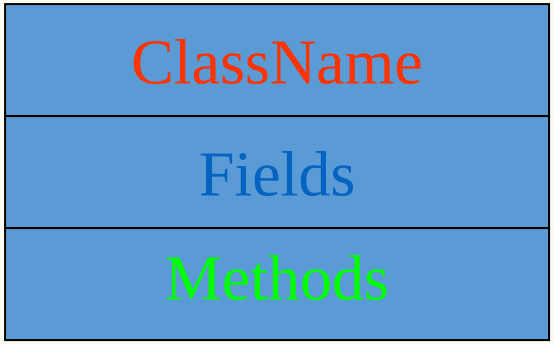
Method parameters are shown inside the parentheses using the same notation as variables.



# Converting the UML Diagram to Code

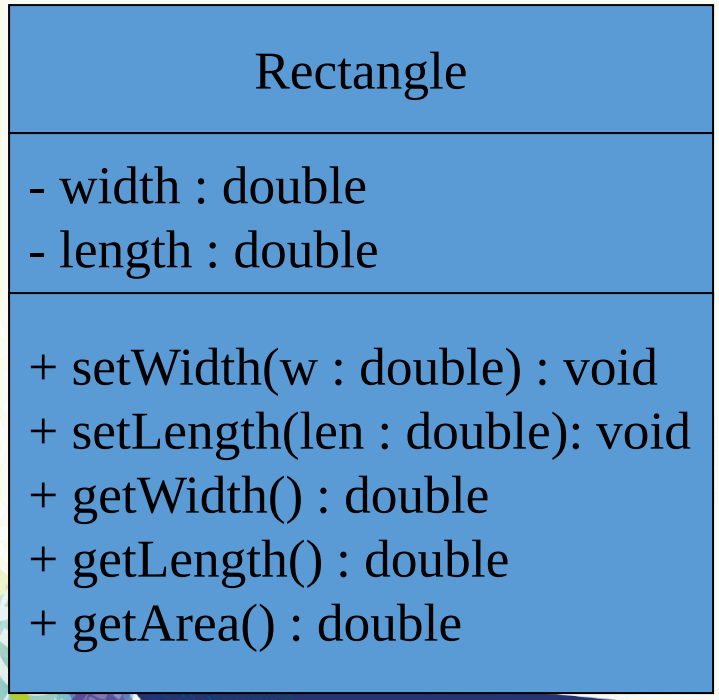
- Putting all of this information together, a Java class file can be built easily using the UML diagram.
- The UML diagram parts match the Java class file structure.

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



# Converting the UML Diagram to Code

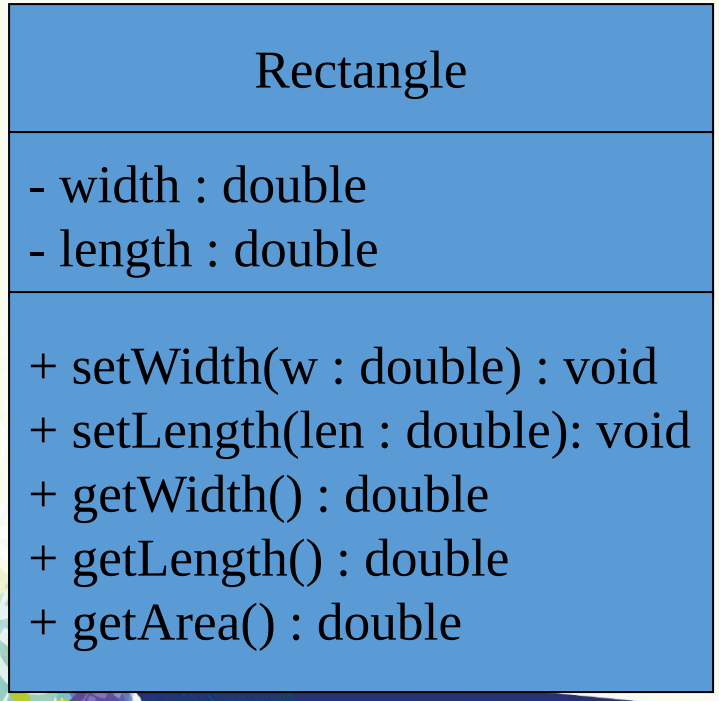
The structure of the class can be compiled and tested without having bodies for the methods. Just be sure to put in dummy return values for methods that have a return type other than void.




```
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 not 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.
- See example: RoomAreas.java
- Note that each room represented in this example can have different dimensions.

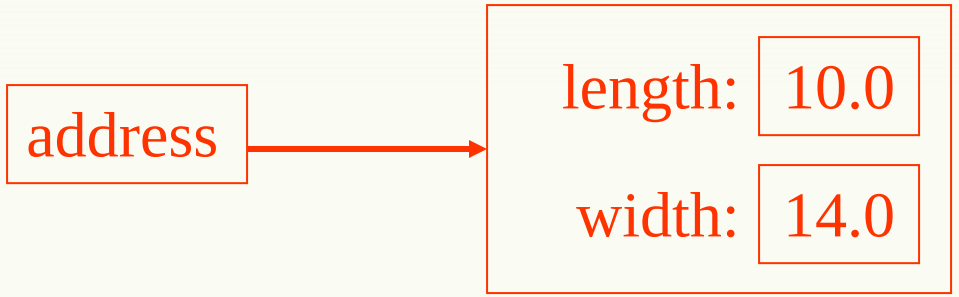
```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

```
import javax.swing.JOptionPane;

/**
   This program creates three instances of the
   Rectangle class.
*/

public class RoomAreas
{
   public static void main(String[] args)
   {
      double number;        // To hold a number
      double totalArea;     // The total area
      String input;         // To hold user input

      // Create three Rectangle objects.
      Rectangle kitchen = new Rectangle();
      Rectangle bedroom = new Rectangle();
      Rectangle den = new Rectangle();

      // Get and store the dimensions of the kitchen.
      input = JOptionPane.showInputDialog("What is the " +
                                          "kitchen's length?");
      number = Double.parseDouble(input);
      kitchen.setLength(number);
      input = JOptionPane.showInputDialog("What is the " +
                                          "kitchen's width?");
      number = Double.parseDouble(input);
      kitchen.setWidth(number);

      // Get and store the dimensions of the bedroom.
      input = JOptionPane.showInputDialog("What is the " +
                                          "bedroom's length?");
      number = Double.parseDouble(input);
      bedroom.setLength(number);
```

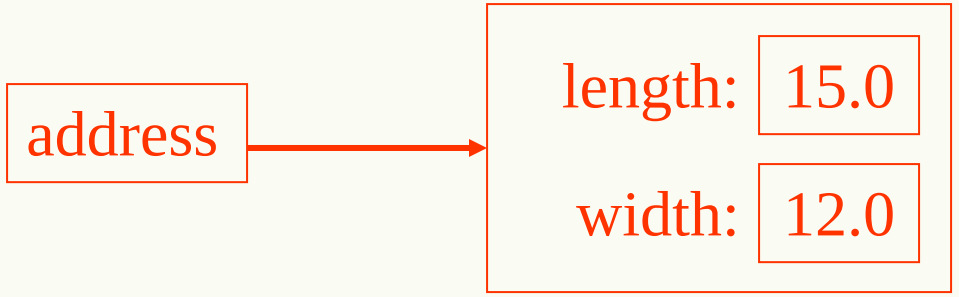
```
        input = JOptionPane.showInputDialog("What is the " +
                                            "bedroom's width?");
        number = Double.parseDouble(input);
        bedroom.setWidth(number);

        // Get and store the dimensions of the den.
        input = JOptionPane.showInputDialog("What is the " +
                                            "den's length?");
        number = Double.parseDouble(input);
        den.setLength(number);
        input = JOptionPane.showInputDialog("What is the " +
                                           "den's width?");
        number = Double.parseDouble(input);
        den.setWidth(number);

        // Calculate the total area of the rooms.
        totalArea = kitchen.getArea() + bedroom.getArea()
                    + den.getArea();

        // Display the total area of the rooms.
        JOptionPane.showMessageDialog(null, "The total area " +
                                      "of the rooms is " + totalArea);

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

# States of Three Different Rectangle Objects

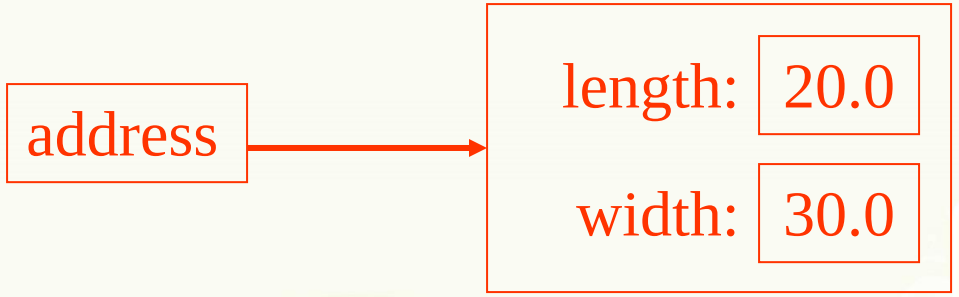
The `kitchen` variable holds the address of a `Rectangle` Object.




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;
}
```

Examples: Rectangle.java, ConstructorDemo.java

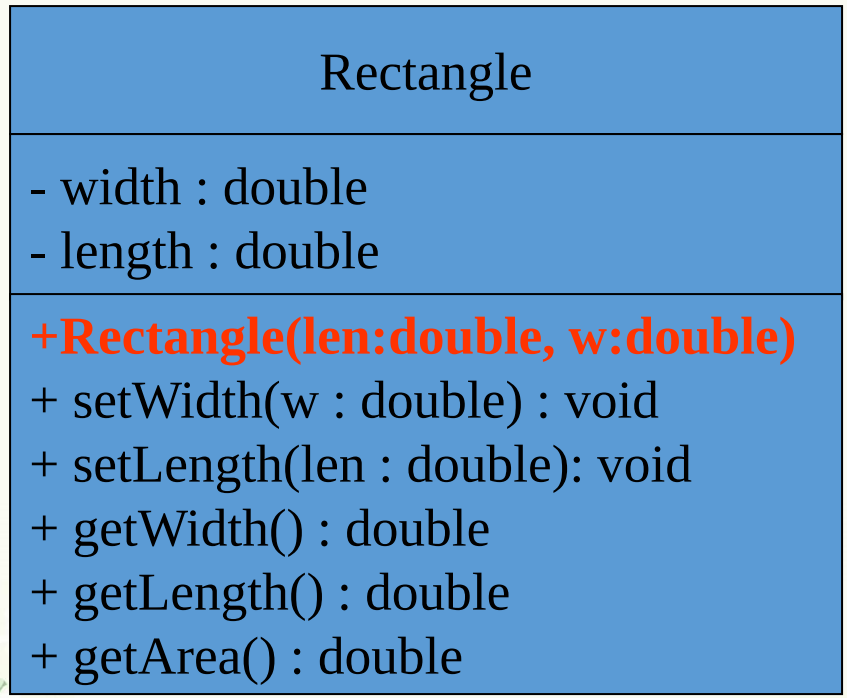
```
/**
   Rectangle class, phase 5
*/

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

   /**
      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;
   }
```

*The remainder of the class has not changed, and is not shown.*

```
/**
   This program demonstrates the Rectangle class's
   constructor.
*/

public class ConstructorDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object, passing 5.0 and
      // 15.0 as arguments to the constructor.
      Rectangle box = new Rectangle(5.0, 15.0);

      // Display the length.
      System.out.println("The box's length is " +
                         box.getLength());

      // Display the width.
      System.out.println("The box's width is " +
                         box.getWidth());

      // Display the area.
      System.out.println("The box's area is " +
                         box.getArea());
   }
}
```

# Constructors in UML

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




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";
```

# Calling One Constructor from Another

```
public class BankAccount
{
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   public BankAccount()
   {
      this(0);
   }
   . . .
}
```

# 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.
- See DieArgument.java

```
public class DieArgument
{
   public static void main(String[] args)
   {
      final int SIX_SIDES = 6;
      final int TWENTY_SIDES = 20;

      // Create a 6-sided die.
      Die sixDie = new Die(SIX_SIDES);

      // Create a 20-sided die.
      Die twentyDie = new Die(TWENTY_SIDES);

      // Roll the dice.
      rollDie(sixDie);
      rollDie(twentyDie);
   }

   public static void rollDie(Die d)
   {
      System.out.println("Rolling a " + d.getSides() +
                         " sided die.");

      // Roll the die.
      d.roll();

      // Display the die's value.
      System.out.println("The die's value: " + d.getValue());
   }
}
```

# 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 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.

# `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?

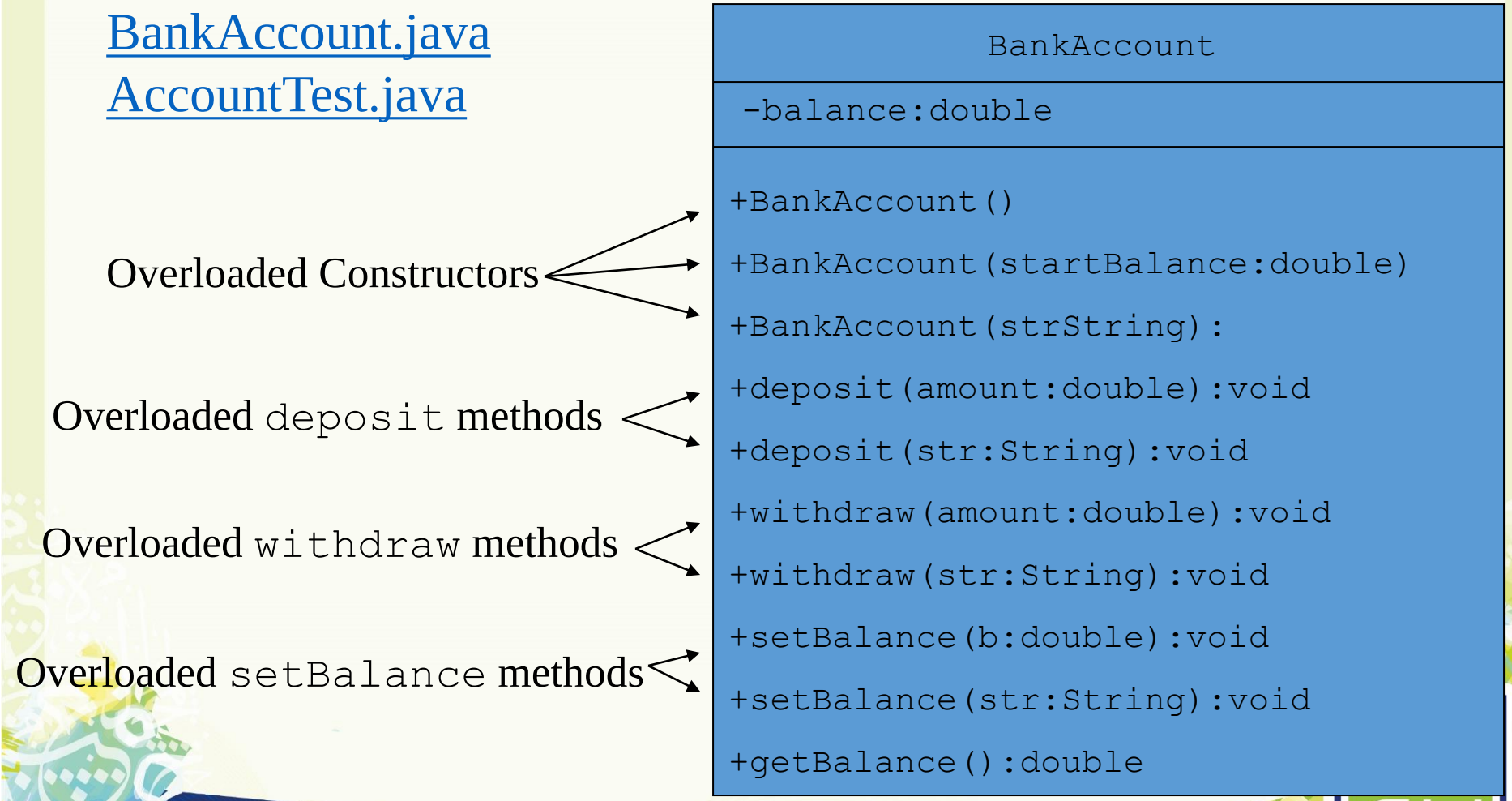
```
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```

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

BankAccount.java
AccountTest.java



```
public class BankAccount
{
   private double balance;      // Account balance

   /**
      This constructor sets the starting balance
      at 0.0.
   */

   public BankAccount()
   {
      balance = 0.0;
   }

   /**
      This constructor sets the starting balance
      to the value passed as an argument.
      @param startBalance The starting balance.
   */

   public BankAccount(double startBalance)
   {
      balance = startBalance;
   }

   /**
      This constructor sets the starting balance
      to the value in the String argument.
      @param str The starting balance, as a String.
   */

   public BankAccount(String str)
   {
      balance = Double.parseDouble(str);
   }
```

```
/**
   The deposit method makes a deposit into
   the account.
   @param amount The amount to add to the
                 balance field.
*/

public void deposit(double amount)
{
   balance += amount;
}

/**
   The deposit method makes a deposit into
   the account.
   @param str The amount to add to the
              balance field, as a String.
*/

public void deposit(String str)
{
   balance += Double.parseDouble(str);
}

/**
   The withdraw method withdraws an amount
   from the account.
   @param amount The amount to subtract from
                 the balance field.
*/

public void withdraw(double amount)
{
   balance -= amount;
}
```

```
/**
   The withdraw method withdraws an amount
   from the account.
   @param str The amount to subtract from
              the balance field, as a String.
*/

public void withdraw(String str)
{
   balance -= Double.parseDouble(str);
}
```

```
   /**
      The setBalance method sets the account balance.
      @param b The value to store in the balance field.
   */

   public void setBalance(double b)
   {
      balance = b;
   }

   /**
      The setBalance method sets the account balance.
      @param str The value, as a String, to store in
                 the balance field.
   */

   public void setBalance(String str)
   {
      balance = Double.parseDouble(str);
    }

   /**
      The getBalance method returns the
      account balance.
      @return The value in the balance field.
   */

   public double getBalance()
   {
      return balance;
   }
}
```

# Account test

```
public class AccountTest
{
   public static void main(String[] args)
   {
      String input;     // To hold user input

      // Get the starting balance.
      input = JOptionPane.showInputDialog(
                "What is your account's starting balance?");

      // Create a BankAccount object.
      BankAccount account = new BankAccount(input);

      // Get the amount of pay.
      input = JOptionPane.showInputDialog(
                "How much were you paid this month?");

      // Deposit the user's pay into the account.
      account.deposit(input);

      // Display the new balance.
      JOptionPane.showMessageDialog(null,
         String.format("Your pay has been deposited.\n" +
                       "Your current balance is $%,.2f",
                       account.getBalance()));

      // Withdraw some cash from the account.
      input = JOptionPane.showInputDialog(
                "How much would you like to withdraw?");
      account.withdraw(input);

      // Display the new balance
      JOptionPane.showMessageDialog(null,
         String.format("Now your balance is $%,.2f",
                       account.getBalance()));

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

# 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. |

## Organizing Related Classes into Packages

To put one of your classes in a package, you must place a line
package *packageName*;
as the first instruction in the source file containing the class. A package name consists of one or more identifiers separated by periods.

let's put the Financial class into a package named com.horstmann.bigjava

The `Financial.java` file must start as follows:

```
package com.horstmann.bigjava;
public class Financial
{
   . . .
}
```

# Importing Packages

- If you want to use a class from a package, you can refer to it by its full name (package name plus class name). For example, java.util.Scanner refers to the Scanner class in the java.util package:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Naturally, that is somewhat inconvenient. For that reason, you usually import a name with an import statement:

```
import java.util.Scanner;
```

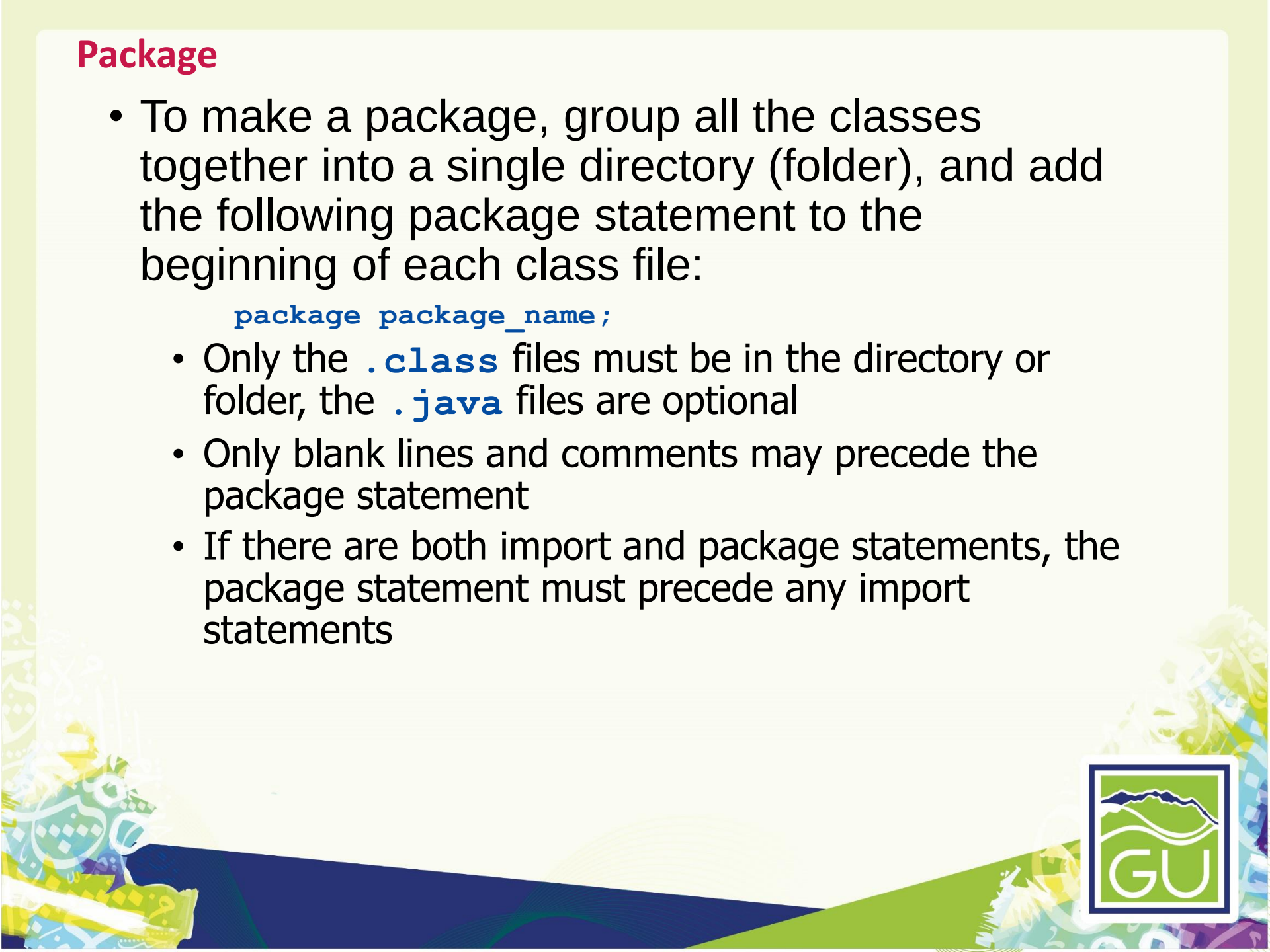
- Then you can refer to the class as Scanner without the package prefix.
- You can import *all classes* of a package with an import statement that ends in .*.
- For example, you can use the statement import java.util.*; to import all classes from the java.util package.
- That statement lets you refer to classes like Scanner or Random without a java.util prefix.

## Package

- To make a package, group all the classes together into a single directory (folder), and add the following package statement to the beginning of each class file:

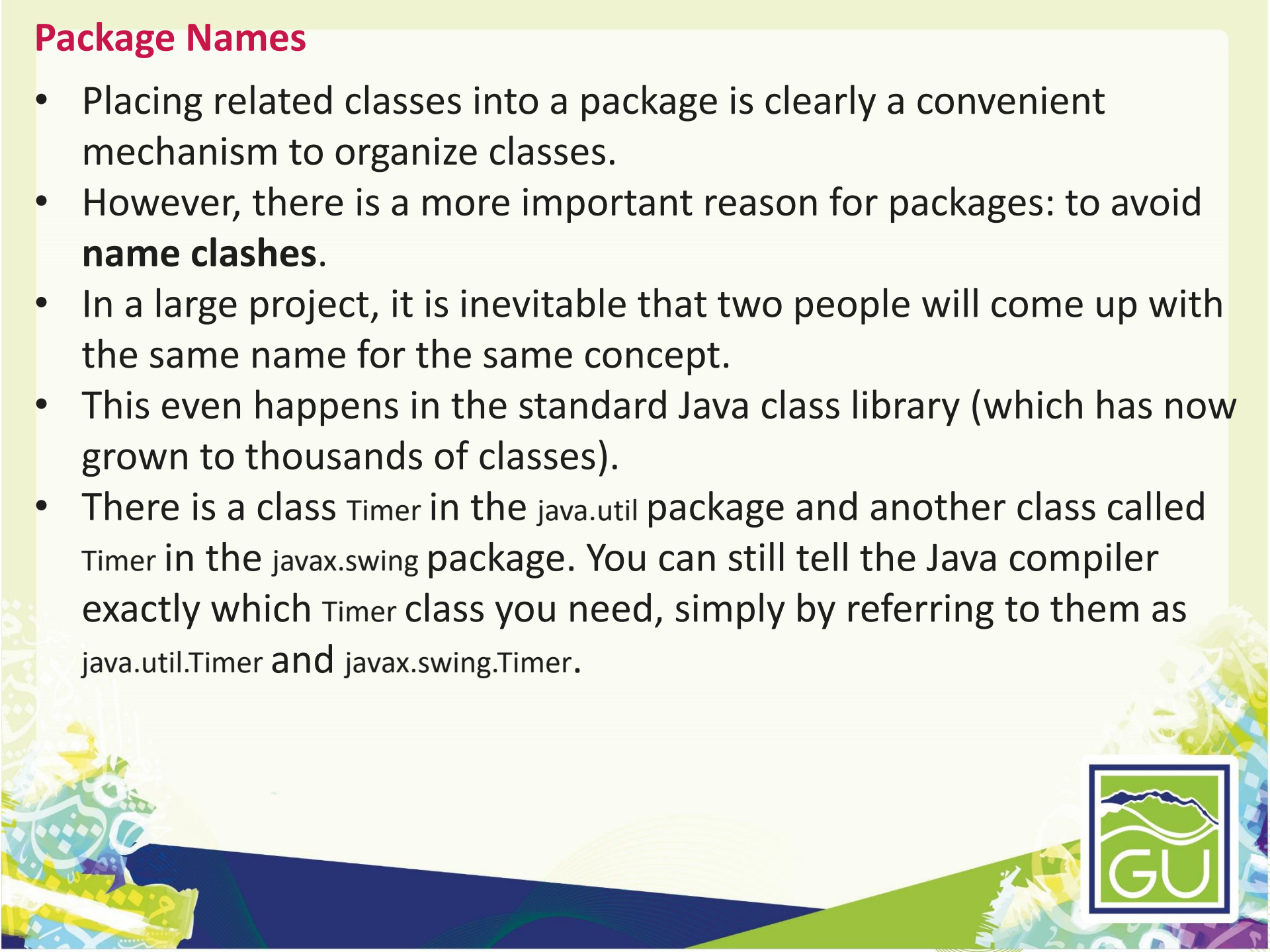
```
package package_name;
```

  - Only the `.class` files must be in the directory or folder, the `.java` files are optional
  - Only blank lines and comments may precede the package statement
  - If there are both import and package statements, the package statement must precede any import statements



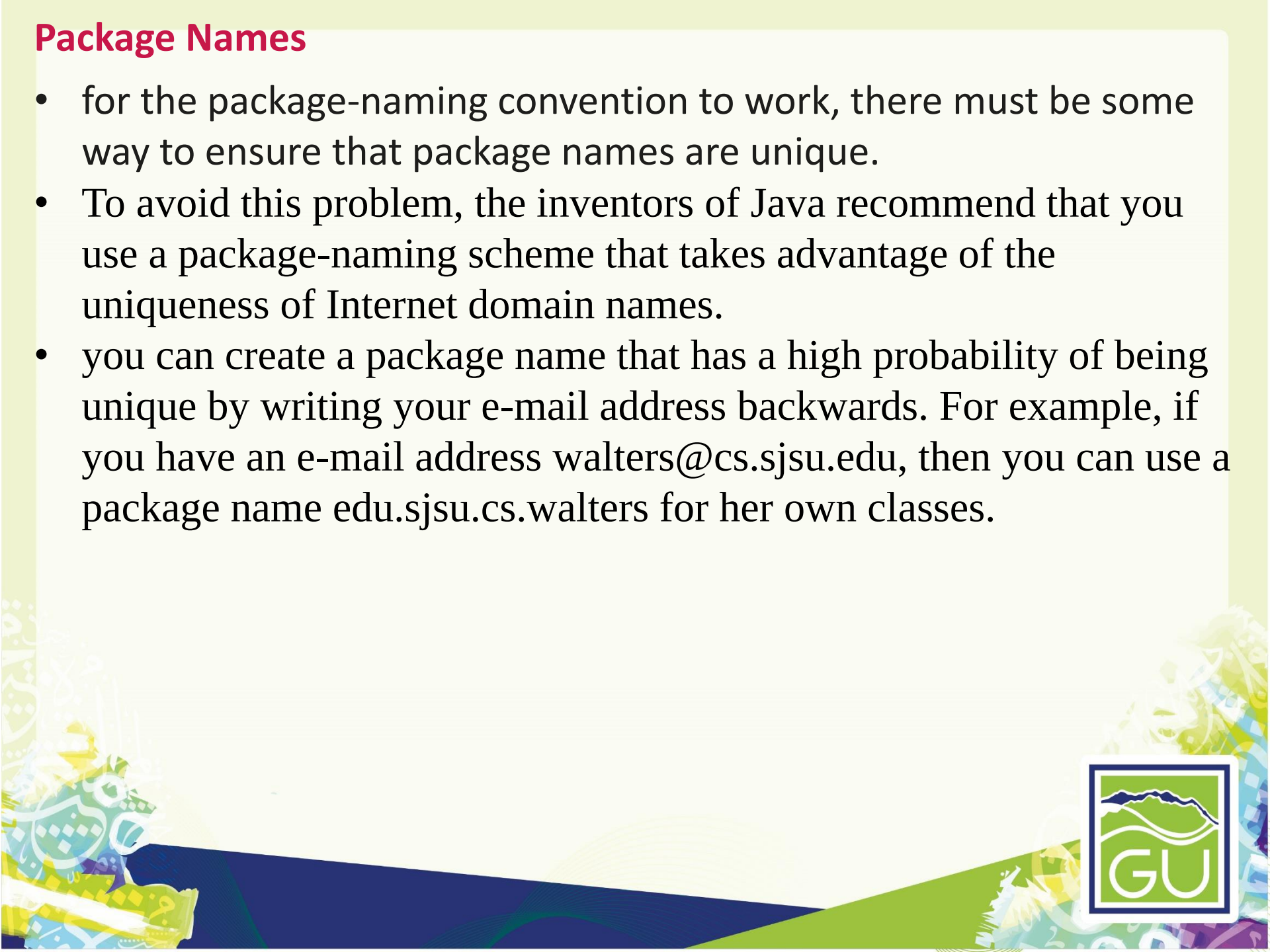
## Package Names

- Placing related classes into a package is clearly a convenient mechanism to organize classes.
- However, there is a more important reason for packages: to avoid **name clashes**.
- In a large project, it is inevitable that two people will come up with the same name for the same concept.
- This even happens in the standard Java class library (which has now grown to thousands of classes).
- There is a class Timer in the java.util package and another class called Timer in the javax.swing package. You can still tell the Java compiler exactly which Timer class you need, simply by referring to them as java.util.Timer and javax.swing.Timer.



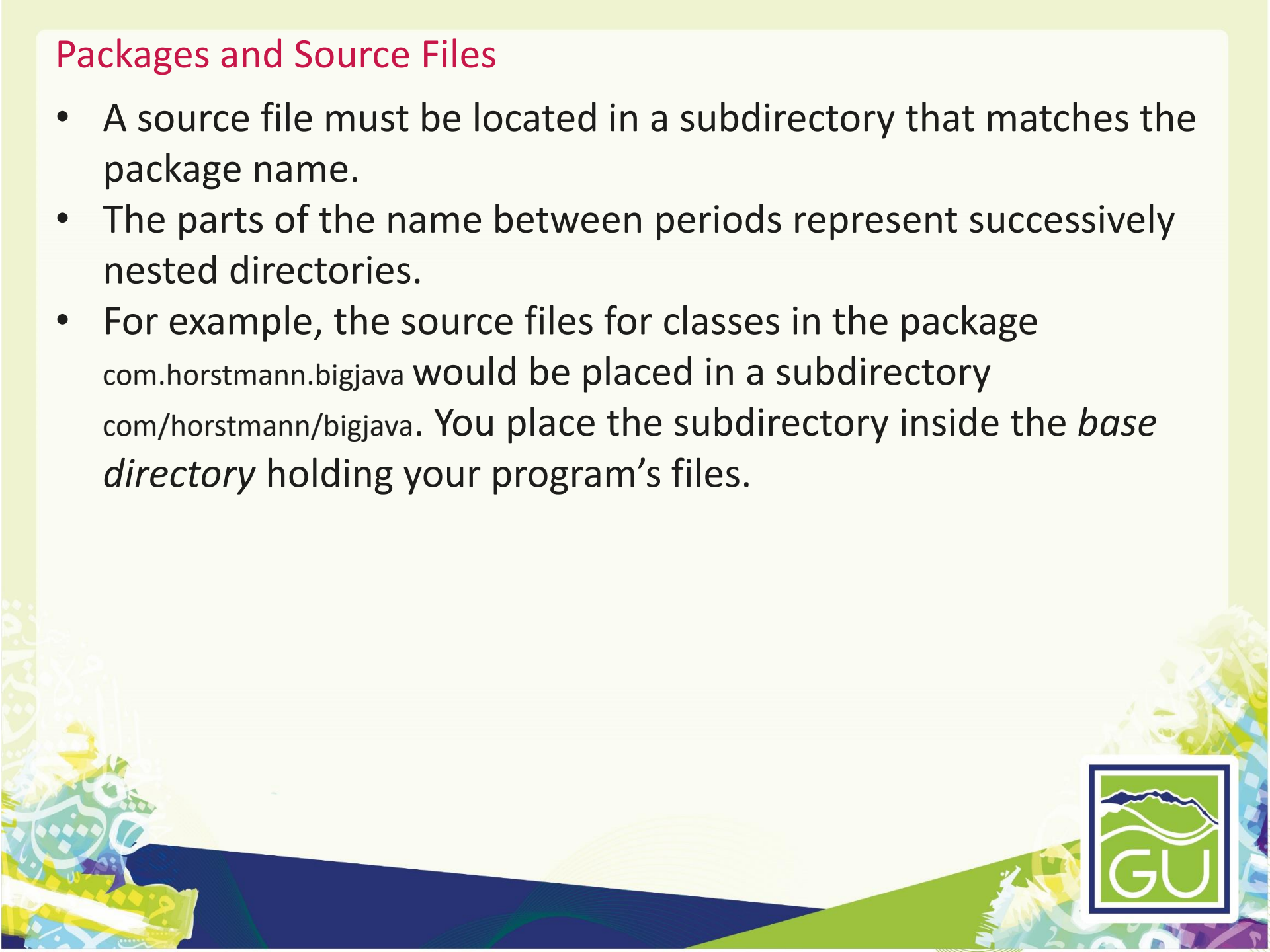
## Package Names

- for the package-naming convention to work, there must be some way to ensure that package names are unique.
- To avoid this problem, the inventors of Java recommend that you use a package-naming scheme that takes advantage of the uniqueness of Internet domain names.
- you can create a package name that has a high probability of being unique by writing your e-mail address backwards. For example, if you have an e-mail address walters@cs.sjsu.edu, then you can use a package name edu.sjsu.cs.walters for her own classes.



## Packages and Source Files

- A source file must be located in a subdirectory that matches the package name.
- The parts of the name between periods represent successively nested directories.
- For example, the source files for classes in the package com.horstmann.bigjava would be placed in a subdirectory com/horstmann/bigjava. You place the subdirectory inside the *base directory* holding your program's files.



# Creating Packages

```
package com.companyName.departmentName.carProject;

public class Engine {

private int engineCapacity;

private int engineSerialNumber;


public Engine(int engineCapacity, int engineSerialNumber) {

this.engineCapacity = engineCapacity;

this.engineSerialNumber = engineSerialNumber;

}


public int getEngineCapacity() {

return engineCapacity;

}


public int getEngineSerialNumber() {

return engineSerialNumber;

}

}
```

Package name

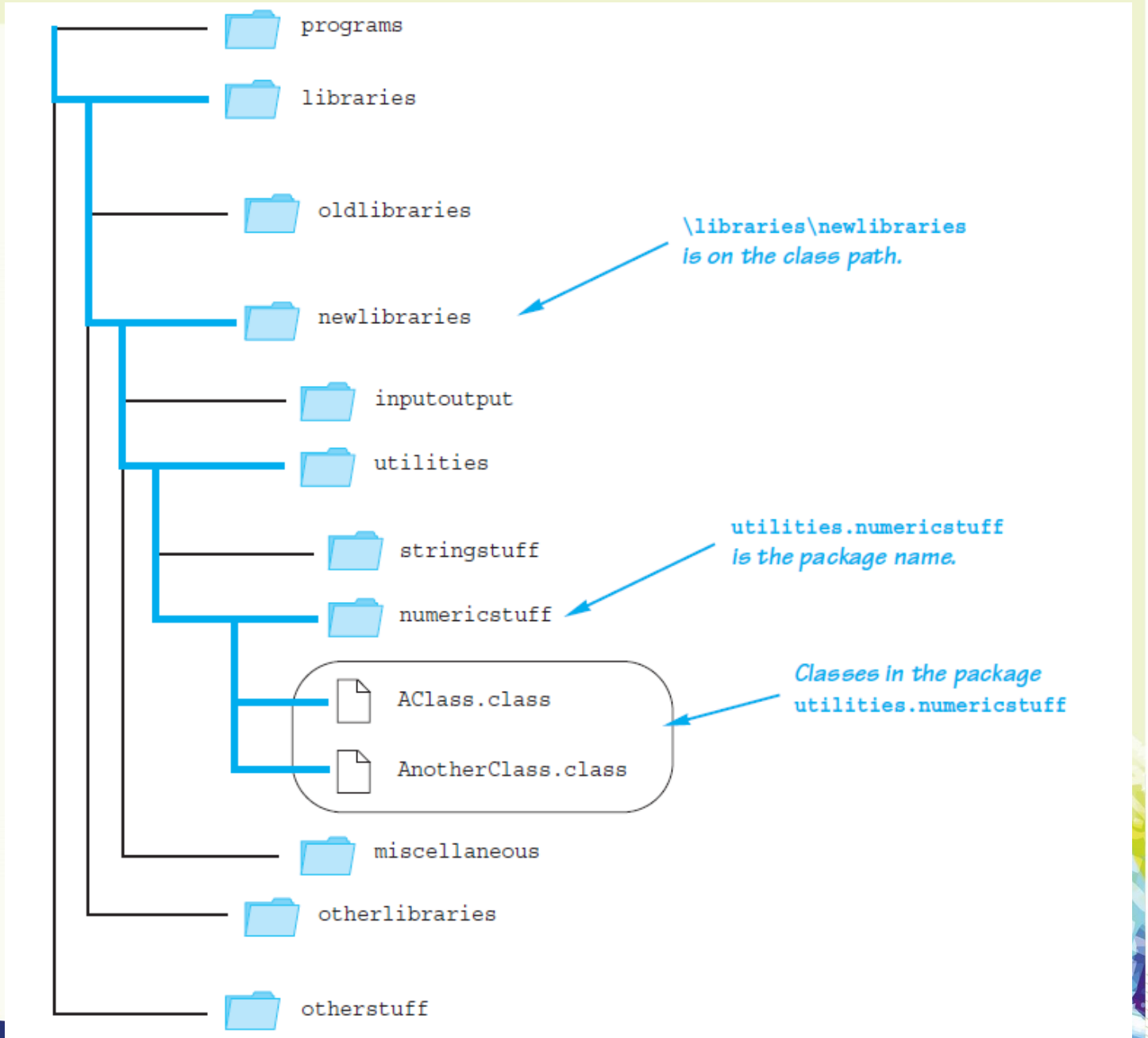
```
package com.companyName.departmentName.carProject;


class Car {

private String make;

private int year;

private Engine engine;

public Car(String make, int year, int engineCapacity, int
engineSerialNumber) {

this.make=make;

this.year=year;

engine = new Engine(engineCapacity, engineSerialNumber);

}

public String getMake() {

return make;

}

public int getYear() {

return year;

}

public int getEngineSerialNumber() {

return engine.getEngineSerialNumber();

}

public int getEngineCapacity() {

return engine.getEngineCapacity();

}

}
```

Package name

# Package Names and Directories

- A package name is the path name for the directory or subdirectories that contain the package classes
- Java needs two things to find the directory for a package: the name of the package and the value of the **CLASSPATH** variable
  - The **CLASSPATH** environment variable is similar to the **PATH** variable, and is set in the same way for a given operating system
  - The **CLASSPATH** variable is set equal to the list of directories (including the current directory, " **.** ") in which Java will look for packages on a particular computer
  - Java searches this list of directories in order, and uses the first directory on the list in which the package is found

# A Package Name



# Subdirectories Are Not Automatically Imported

- When a package is stored in a subdirectory of the directory containing another package, importing the enclosing package does not import the subdirectory package
- The import statement:

```
import utilities.numericstuff.*;
```

  imports the `utilities.numericstuff` package only
- The import statements:

```
import utilities.numericstuff.*;
import utilities.numericstuff.statistical.*;
```

  import both the `utilities.numericstuff` and `utilities.numericstuff.statistical` packages

# The Default Package

- All the classes in the current directory belong to an unnamed package called the *default package*
- As long as the current directory ( **.** ) is part of the **CLASSPATH** variable, all the classes in the default package are automatically available to a program

# 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

Thank you.