Lecture 03: Object Oriented Programming

CS 0445: Data Structures

Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Sep 5, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

Objects and Classes

- An object belongs to a class, which defines its data type
- A class specifies …
  - Kind of data objects of that class have
  - What actions the objects can take
  - How they accomplish these actions
- Object Oriented Programming
  - A world consisting of objects that interact with one another by means of actions

Objects and Classes

An outline of a class and …

Class Name: Automobile

Data:

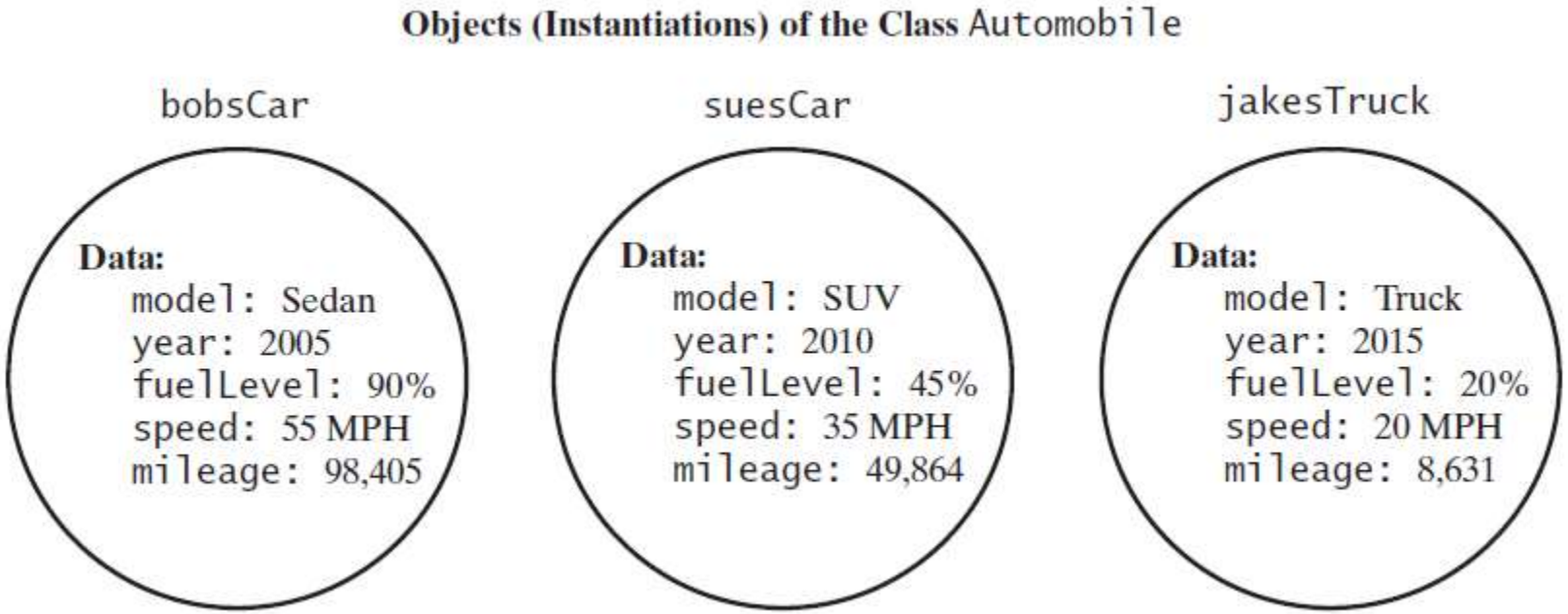
- model
- year
- fuelLevel
- speed
- mileage

Methods (actions):

- goForward
- goBackward
- accelerate
- decelerate
- getFuelLevel
- getSpeed
- getMileage

Objects and Classes

- … three of its instances



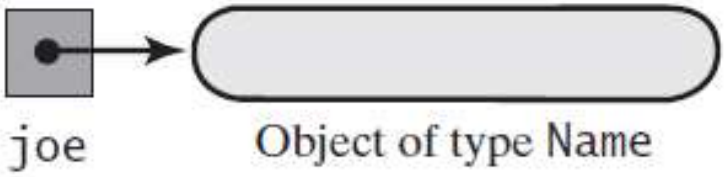
Using the Methods in a Java Class

- A program component that uses a class is called a client of the class
- The **new** operator creates an instance of a class
  - By invoking a special method within the class
  - Known as a constructor
- Variable **joe** references memory location where object is stored

```
Name joe = new Name();
```

Using the Methods in a Java Class

A variable that references an object



Using the Methods in a Java Class

- Class should have methods that give capability to set data values

```
joe.setFirst("Joseph");
joe.setLast("Brown");
```

  - Void methods, they do not return a value.
- Class needs methods to retrieve values

```
String hisName = joe.getFirst();
```

  - Valued methods, return a value

References and Aliases

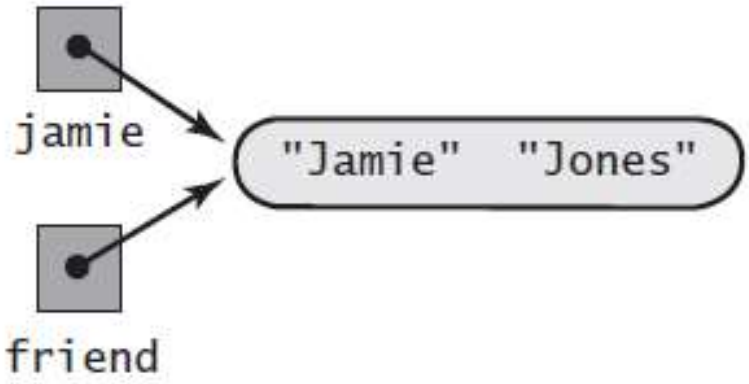
- A reference variable contains the address in memory of an actual object.
- Consider:

```
Name jamie = new Name();
jamie.setFirst("Jamie");
jamie.setLast("Jones");
Name friend = jamie;
```

- Variables **jamie** and **friend** reference the same instance of **Name**

References and Aliases

Aliases (handles) of an object



Defining a Java Class

- The Java class Name that represents a person's name.
  - Store a class definition in a file
  - File name is the name of the class followed by .java.

```
public class Name
{
   private String first; // First name
   private String last;  // Last name
   < Definitions of methods are here >
   . . .
} // end Name
```

Defining a Java Class

- **public** means no restrictions on where class is used
- Strings **first** and **last** are class's data fields
- **private** means only methods within class can refer to the data fields

```
public class Name
{
   private String first; // First name
   private String last;  // Last name
   < Definitions of methods are here >
   . . .
} // end Name
```

Defining a Java Class

- **private** fields accessed by
  - Accessor methods (get)
  - Mutator methods (set)

```
public class Name
{
   private String first; // First name
   private String last;  // Last name
   < Definitions of methods are here >
   . . .
} // end Name
```

Method Definitions

- General form:

```
access-modifier  use-modifier  return-type  method-name(parameter-list)
{
   method-body
}
```

  - *use modifier* is optional and in most cases omitted
  - *return type*, (for a valued method), data type of the value method returns
  - *parameters* specify values, objects that are inputs

Method Definitions

- Examples of **get** and **set** methods

```
public String getFirst()  <-- Header
{
   return first;          } Body
} // end getFirst
```

```
public void setFirst(String firstName)
{
   first = firstName;
} // end setFirst
```

- Possible to reference class data field **first** with **this.first**
  - **this** references "this" instance of the **Name** object

Arguments and Parameters

- Consider:

```
Name joe = new Name();
joe.setFirst("Joseph");
joe.setLast("Brown");
```

- Strings "Joseph" and "Brown" are the arguments.
  - Correspond to the parameters of the method definition
- Method invocation must provide exactly as many arguments as parameters as method definition

Passing Arguments

- Method cannot change the value of an argument that has a primitive data type
  - Mechanism is described as *call-by-value.*
- When parameter has class type, corresponding argument in method invocation must be object of that class type
  - Parameter is initialized to the memory address of that object
  - Method can change the data in the object

A Definition of the Class Name

```
1  public class Name
2  {
3     private String first; // First name
4     private String last;  // Last name
5
6     public Name()
7     {
8     } // end default constructor
9
10    public Name(String firstName, String lastName)
11    {
12       first = firstName;
13       last = lastName;
14    } // end constructor
15
16    public void setName(String firstName, String lastName)
17    {
18       setFirst(firstName);
19       setLast(lastName);
20    } // end setName
21
22    public String getName()
```

A Definition of the Class Name

```
21
22      public String getName()
23      {
24         return toString();
25      } // end getName
26
27      public void setFirst(String firstName)
28      {
29         first = firstName;
30      } // end setFirst
31
32      public String getFirst()
33      {
34         return first;
35      } // end getFirst
36
37      public void setLast(String lastName)
38      {
39         last = lastName;
40      } // end setLast
41
```

A Definition of the Class Name

```
40      } // end setLast
41
42      public String getLast()
43      {
44         return last;
45      } // end getLast
46
47      public void giveLastNameTo(Name aName)
48      {
49         aName.setLast(last);
50      } // end giveLastNameTo
51
52      public String toString()
53      {
54         return first + " " + last;
55      } // end toString
56   } // end Name
```

Constructors

- Constructor allocates memory for object, initializes the data fields
- Constructor has certain special properties
  - Same name as the class
  - No return type, not even **void**
  - Any number of parameters, including no parameters
- Constructor without parameters called the default constructor

Constructors

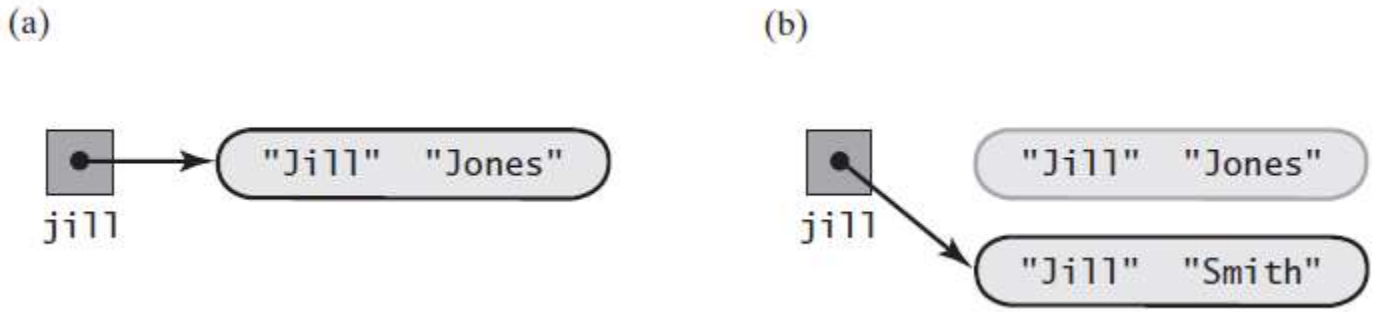
- Consider these two statements:

```
Name jill = new Name("Jill", "Jones");
jill = new Name("Jill", "Smith");
```

  - Second statement allocates new memory, with **jill** pointing to it
  - Previous memory location “lost”
  - System periodically deallocates, returns to O.S.

Constructors

An object (a) after its initial creation;
(b) after its reference is lost



The Method **toString**

- Method **toString** in class **Name** returns a string that is person's full name
  - Java will invoke it automatically when you write

```
System.out.println(jill);
```

- Providing a class with a method **toString** is a good idea in general

Methods That Call Other Methods

- Can use the reserved word **this** to call a constructor
  - From within the body of another constructor.

```
public Name()
{
   this("", "");
} // end default constructor
```

  - Revision of default constructor to initialize **first** and **last**, by calling the second constructor

Static Fields and Methods

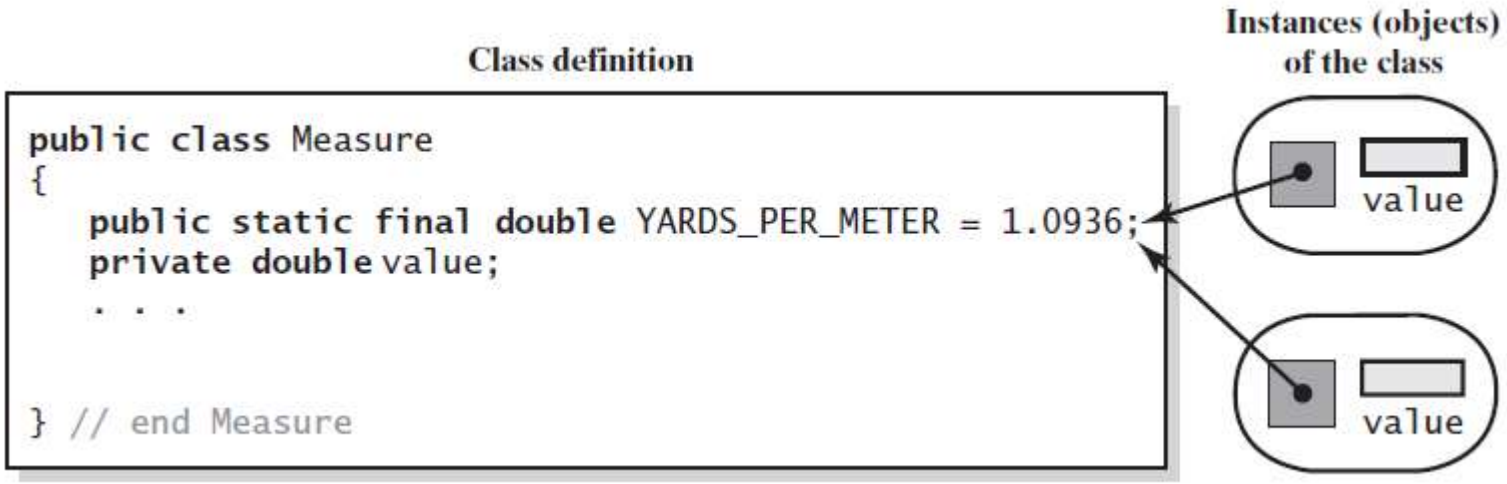
- Sometimes you need a data field that does not belong to any one object
  - Such a data field is called a *static field*

```
private static int numberOfInvocations = 0;
```

- Objects can use static field to communicate with each other
  - Or to perform some joint action.

Static Fields and Methods

A static field **YARDS_PER_METER** versus a nonstatic field value. Objects of the class **Measure** all reference the same static field but have their own copy of value



Static Fields and Methods

- Static method: a method that does not belong to an object of any kind.
  - Still a member of a class
  - Use the class name instead of an object name to invoke the method
- Methods from class **Math**

```
int maximum = Math.max(2, 3);
double root = Math.sqrt(4.2);
```

Overloading Methods

- Methods within same class can have same name,

```
public void setName(String firstName, String lastName)
public void setName(Name otherName)
```

  - As long as they do not have identical parameters

Packages

- Using several related classes is more convenient if …
  - You group them together within a Java package
- To identify a class as part of a particular package
  - Begin the file that contains the class with a statement like **package myStuff;**
  - Then place all of the files within one directory or folder, give it same name as the package.

Packages

- To use a package in your program …
  - Begin the program with a statement such as **import myStuff.*;**
- Asterisk makes all public classes within package available to the program

The Java Class Library

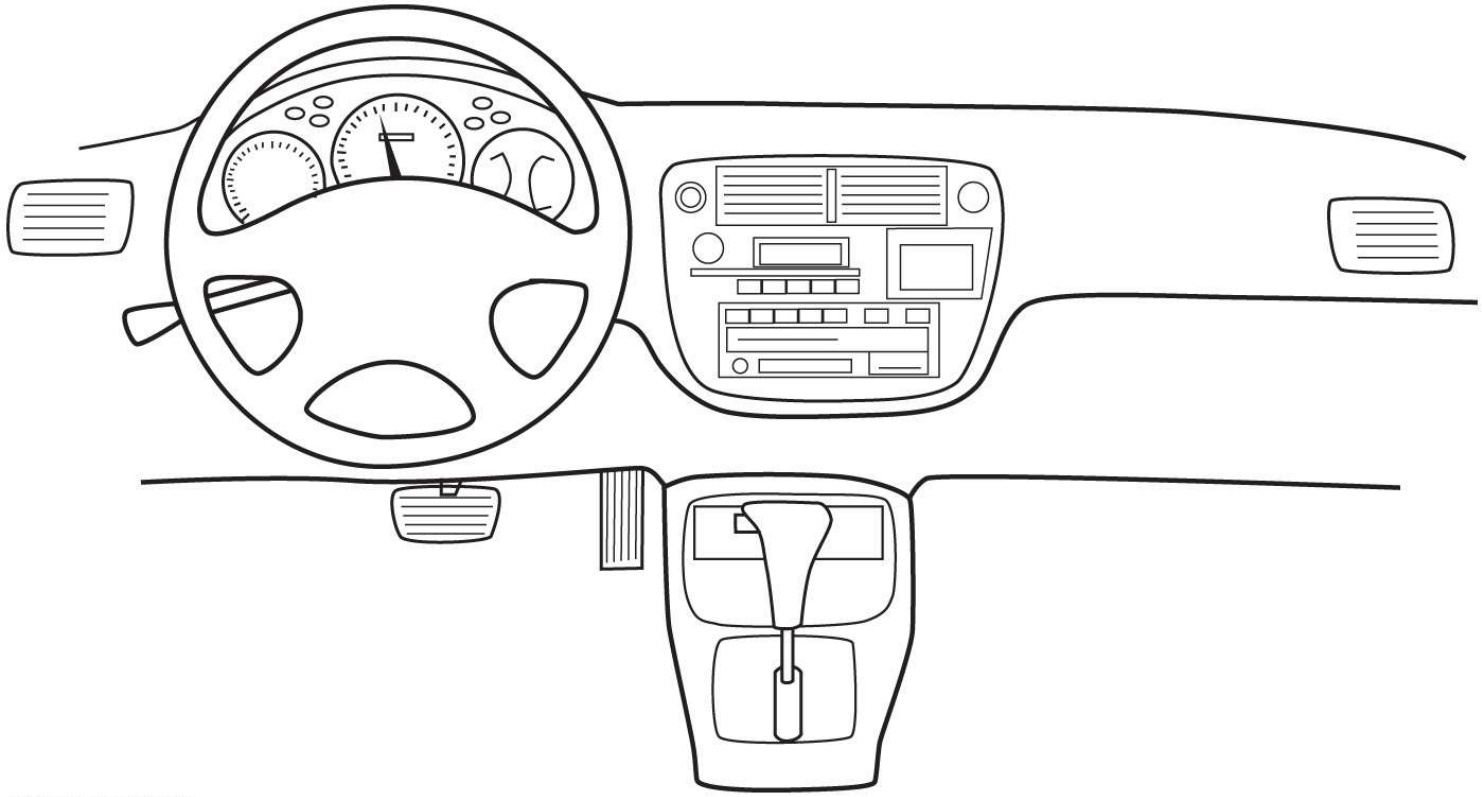
- Java comes with a collection of many classes you can use
  - This collection of classes is known as **the Java Class Library**
  - Sometimes as the **Java Application Programming Interface**

Encapsulation

- Information hiding
- Enclose data and methods within a class
- Hide implementation details
- Programmer receives only enough information to be able to use the class

Encapsulation

- An automobile's controls are visible to the driver, but its inner workings are hidden

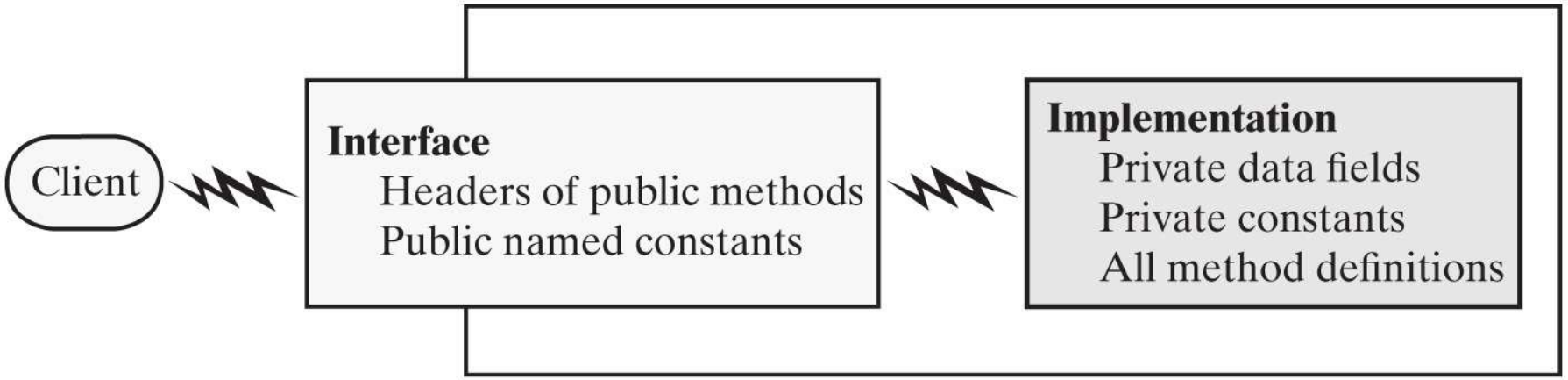


Abstraction

- Focus on what instead of how
  - What needs to be done?
  - For the moment ignore how it will be done.
- Divide class into two parts
  - Client interface
  - Implementation

Abstraction

- An interface provides well-regulated communication between a hidden implementation and a client



Specifying Methods

- Preconditions
  - What must be true before method executes
  - Implies responsibility for client
- Postconditions
  - Statement of what is true after method executes
- Use assertions
  - In comments or with assert statement

Java Interfaces

- Program component that declares a number of public methods
  - Should include comments to inform programmer
  - Any data fields here should be public, final, static

Interface `Measurable`

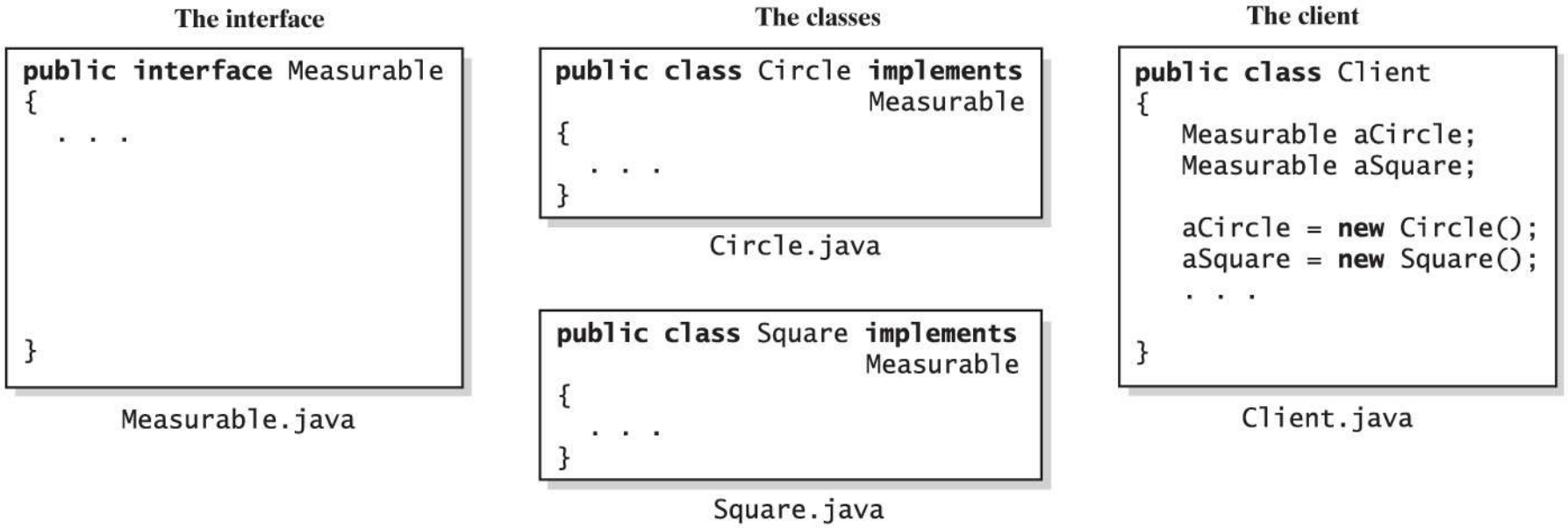
```
/**
   An interface for methods that return
   the perimeter and area of an object.
*/
public interface Measurable
{
   /** Gets the perimeter.
      @return  The perimeter. */
   public double getPerimeter();

   /** Gets the area.
      @return  The area. */
   public double getArea();
} // end Measurable
```

Interface NameMeasurable

```
/** An interface for a class of names. */
public interface NameInterface
{
  /** Sets the first and last names.
      @param firstName  A string that is the desired first name.
      @param lastName   A string that is the desired last name. */
  public void setName(String firstName, String lastName);

  /** Gets the full name.
      @return  A string containing the first and last names. */
  public String getName();

  public void setFirst(String firstName);
  public String getFirst();

  public void setLast(String lastName);
  public String getLast();

  public void giveLastNameTo(NameInterface aName);
  public String toString();
} // end NameInterface
```

Implementing an Interface

- The files for an interface, a class that implements the interface, and the client



Implementing an Interface

- A way for programmer to guarantee a class has certain methods
- Several classes can implement the same interface
- A class can implement more than one interface

Interface as a Data Type

- You can use a Java interface as you would a data type
- Indicates variable can invoke certain set of methods and only those methods.
- An interface type is a reference type
- An interface can be used to derive another interface by using inheritance

Interface vs. Abstract Class

- Purpose of interface similar to that of abstract class
  - But an interface is not a class
- Use an abstract class …
  - If you want to provide a method definition
  - Or declare a private data field that your classes will have in common
- A class can implement several interfaces but can extend only one abstract class.

Named Constants Within an Interface

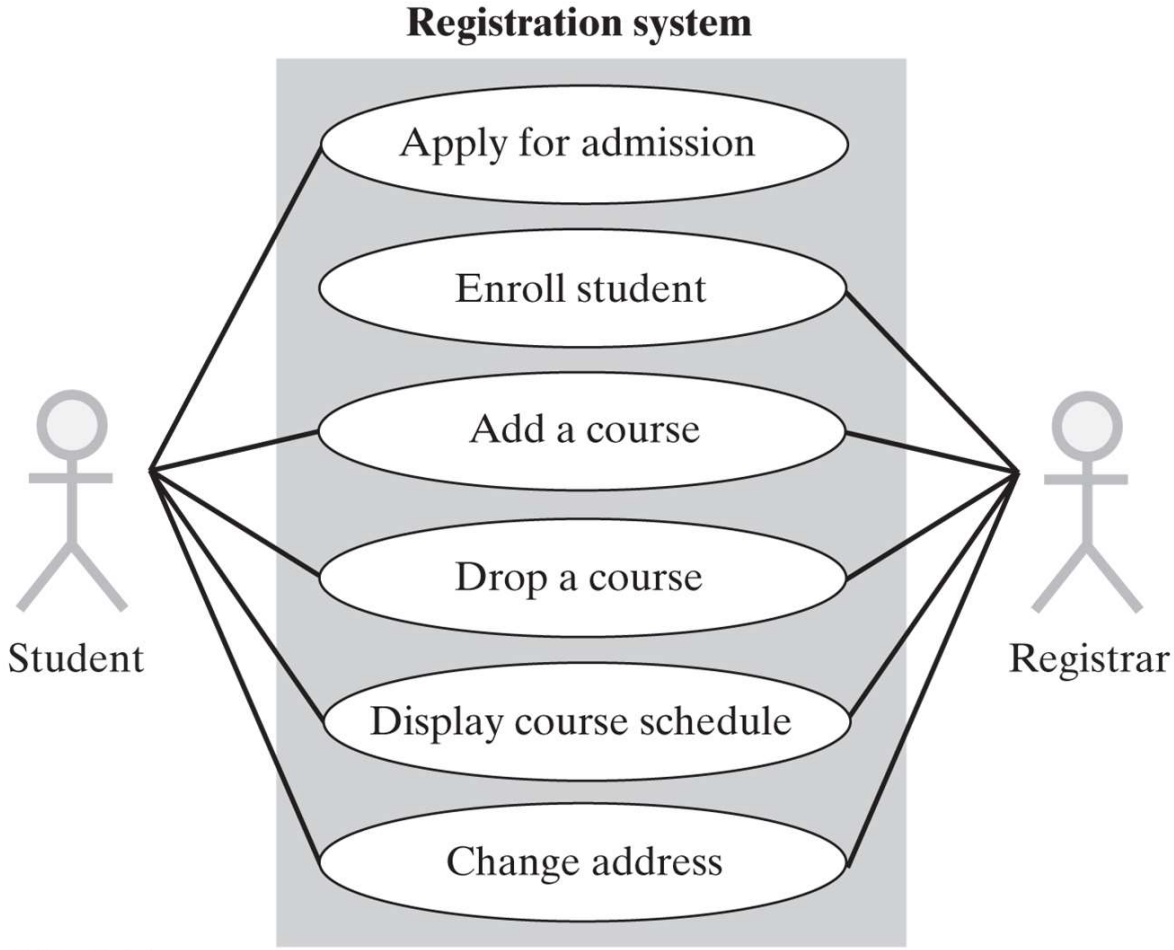
- An interface can contain named constants,
  - Public data fields that you initialize and declare as final.
- Options:
  - Define the constants in an interface that the classes implement
  - Define your constants in a separate class instead of an interface

Choosing Classes

- Consider a registration system for your school …
- Issues:
  - Who, what will use the system?
  - What can each actor do with the system?
  - Which scenarios involve common goals?

Choosing Classes

- A use case diagram for a registration system



Identifying Classes

- A description of a use case for adding a course

System: Registration
Use case: Add a course
Actor: Student
Steps:

1. **Student enters identifying data.**
2. **System confirms eligibility to register.**
   a. **If ineligible to register, ask student to enter identification data again.**
   b. **Student chooses a particular section of a course from a list of course offerin**
   c. **System confirms availability of the course.**
   d. **If course is closed, allow student to return to Step 3 or quit.**
   e. **System adds course to student's schedule.**
   f. **System displays student's revised schedule of courses.**

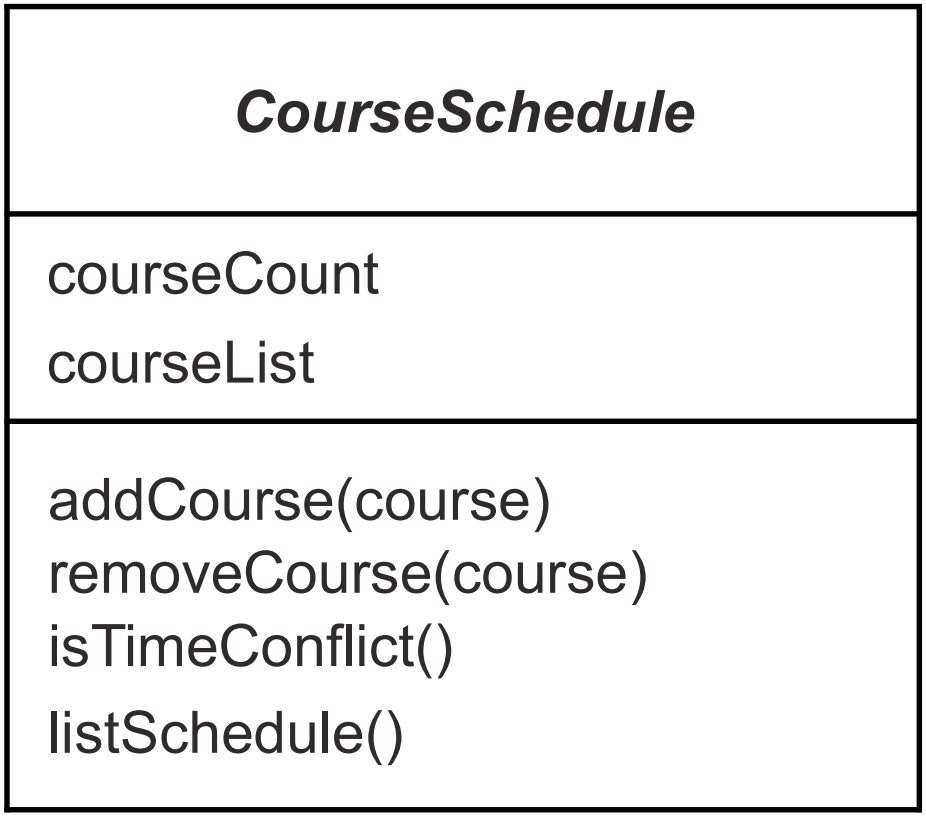
CSC Card Example

- A class-responsibility-collaboration (CRC) card

CourseSchedule
Responsibilities
Add a course
Remove a course
Check for time conflict
List course schedule
Collaborations
Course
Student

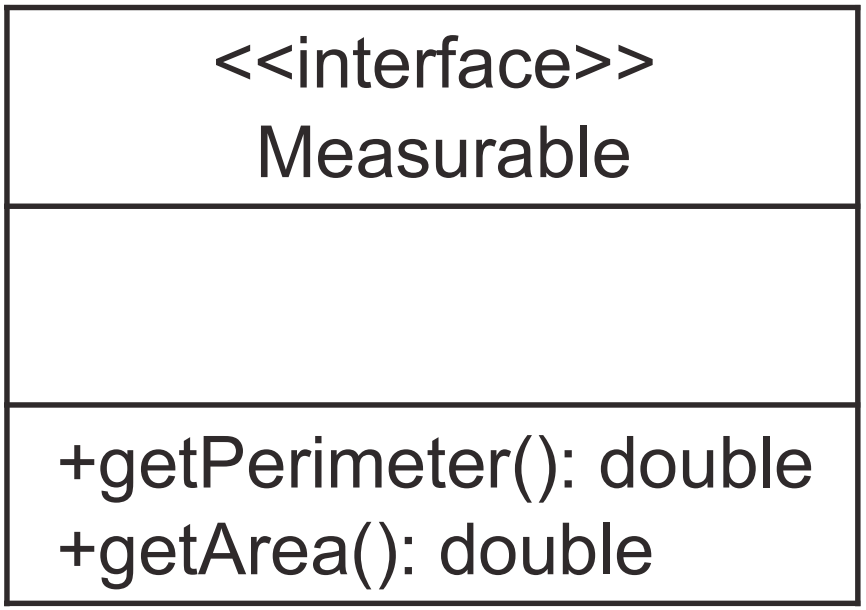
Unified Modeling Language Class

- A class representation that can be a part of a class diagram



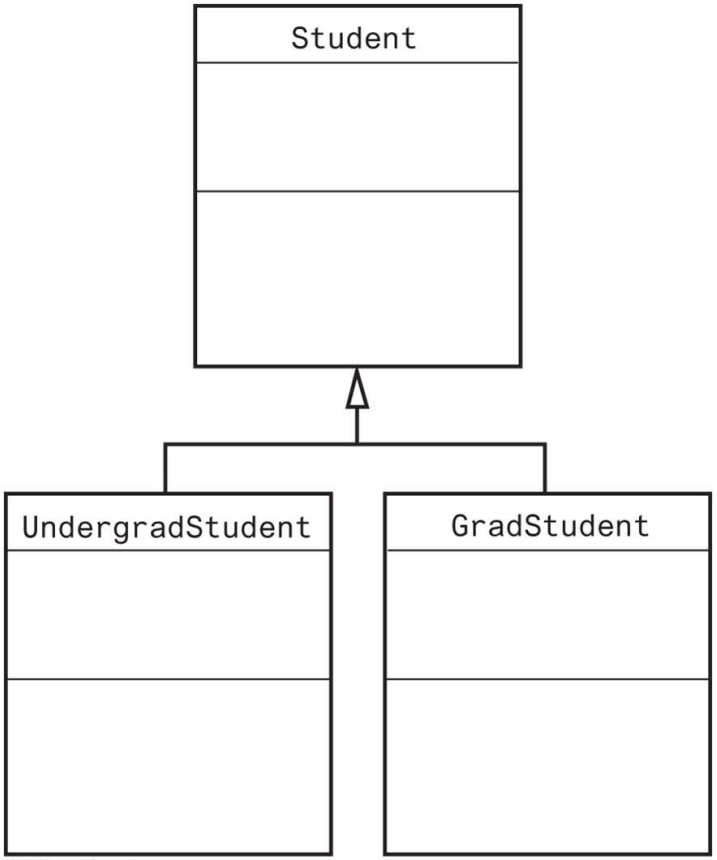
UML Interface Example

- UML notation for the interface `Measurable`



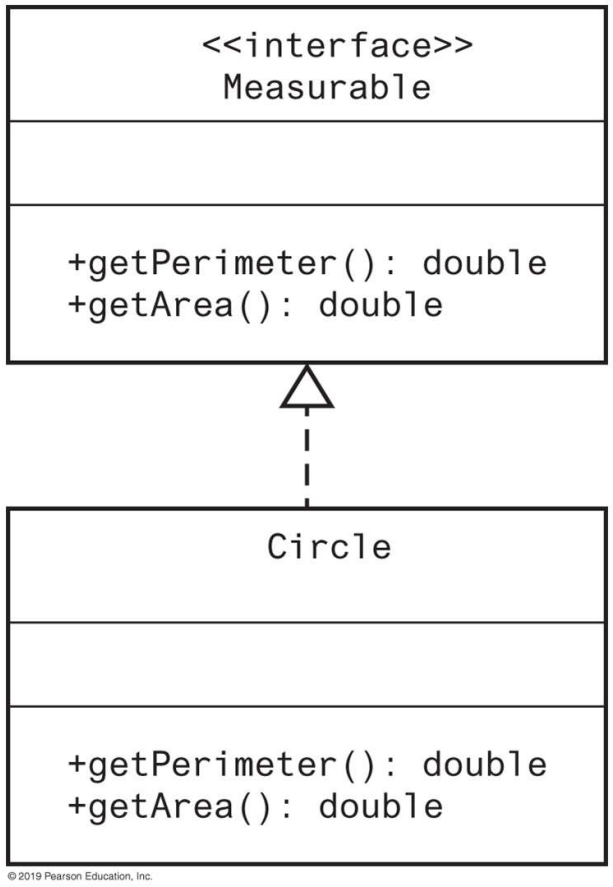
UML Class Hierarchy

- A class diagram showing the base class `Student` and two subclasses



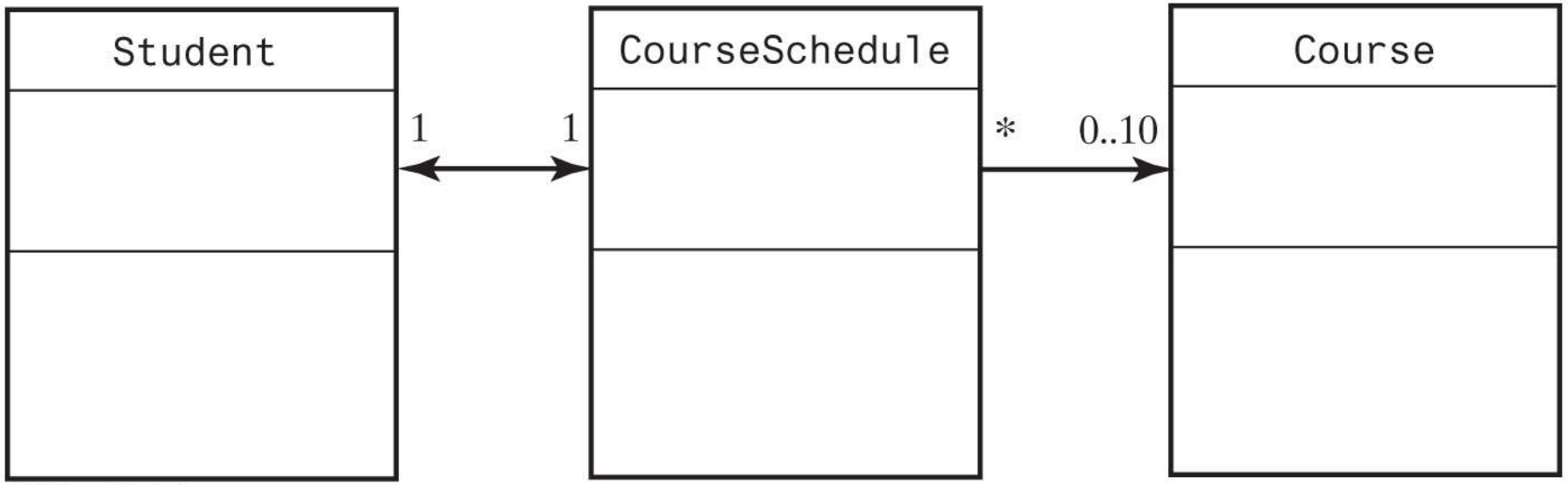
UML Interface Implementation

- A class diagram showing the class `Circle` that implements the interface `Measurable`



UML Class Associations

- Part of a UML class diagram with associations



Reusing Classes

- Not all programs designed and written “from scratch”
- Actually, most software created by combining
  - Already existing components with
  - New components
- Saves time and money
- Reused components are already tested