03 - Object-Oriented Thinking Part 1

CS202: Introduction to Object Oriented Programming

Victor Mejia
CSULA

Objectives

- To apply class abstraction to develop software (§10.2).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.3).
- To discover the relationships between classes (§10.4).
- To design programs using the object-oriented paradigm (§§10.5–10.6).

Today's Topics

- *Quick Recap*: The this Keyword
- *Quick Recap*: Static Methods and Variables
- Passing Objects to Methods
- Class Abstraction and Encapsulation
- Thinking in Objects
- Class Relationships
  - Association: Aggregation and Composition
- Case Study: Designing the Course Class
- Case Study: Designing a Class for Stacks

Quick Recap: the this keyword

The `this` Keyword

- The **this** keyword is the name of a reference that refers to an object itself. One common use of the **this** keyword is reference a class's *hidden data fields*.
- Another common use of the **this** keyword to enable a constructor to invoke another constructor of the same class.

The `this` Keyword

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
  return this.radius * this.radius * Math.PI;
  }

  public String toString() {
    return "radius: " + this.radius
       + "area: " + this.getArea() ;
  }
}
```

(a)

Equivalent

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public String toString() {
    return "radius: " + radius
       + "area: " + getArea() ;
  }
}
```

(b)

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  ...
}
```

The **this** keyword is used to reference the hidden data field radius of the object being constructed.

The **this** keyword is used to invoke another constructor.

Quick Recap: Static Variables and Methods

Instance Variables, and Methods

Instance variables belong to a specific instance.

Instance methods are invoked by an instance of the class.

Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

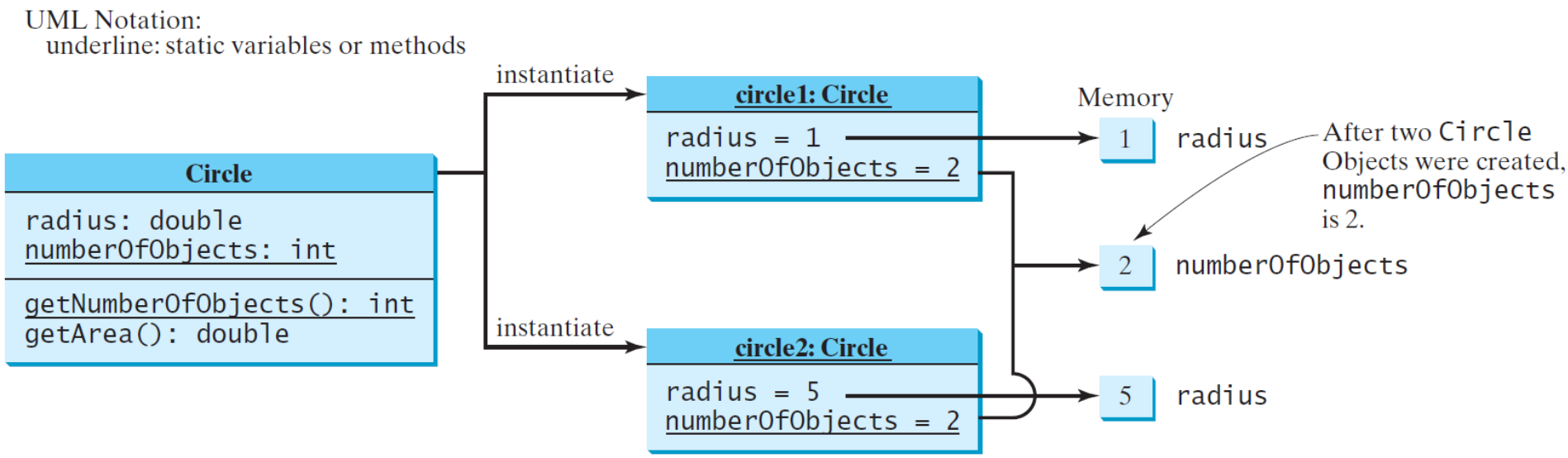
Static methods are not tied to a specific object.

Static constants (use `final`) are final variables shared by all the instances of the class.

Static Variables, Constants, and Methods

To declare static variables, constants, and methods, use the `static` modifier.

Static Variables, Constants, and Methods



Passing Objects to Methods

Passing Objects to Methods

- Passing by value for primitive type value (the value is passed to the parameter)
- Passing by value for reference type value (the value is the reference to the object)

Passing Objects to Methods

- Java is always pass by value
- in methods, the objects are **passed a copy of the object reference**

Example: Passing Objects

- Circle.java
- TestPassingObjects.java

Array of Objects

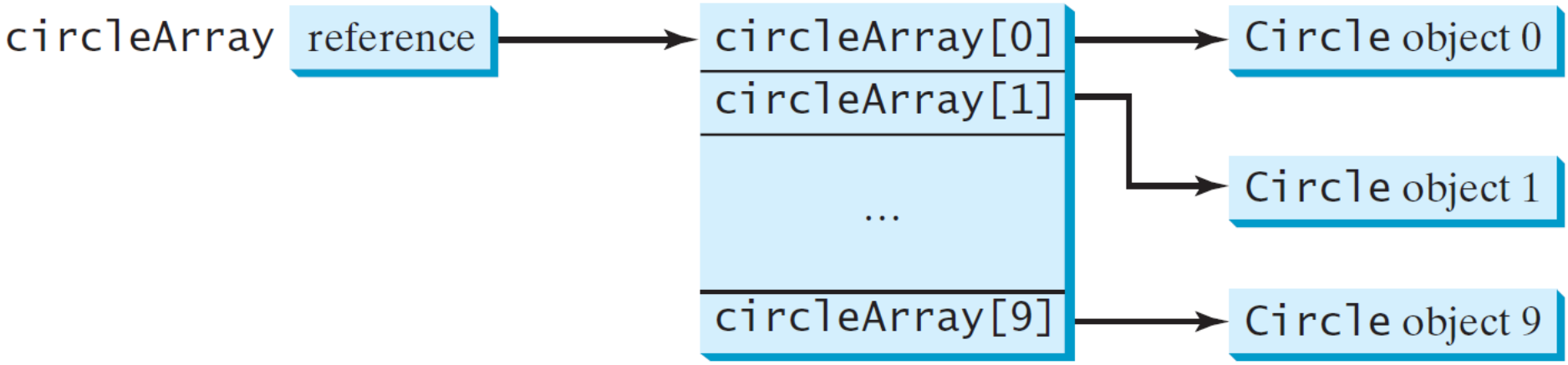
Array of Objects

```
Circle[] circleArray = new Circle[10];
```

- An array of objects is actually an *array of reference variables*.
- So invoking circleArray[1].getArea() involves two levels of referencing as shown in the next figure.
- circleArray references to the entire array. circleArray[1] references to a Circle object.

Array of Objects, cont.

```
Circle[] circleArray = new Circle[10];
```



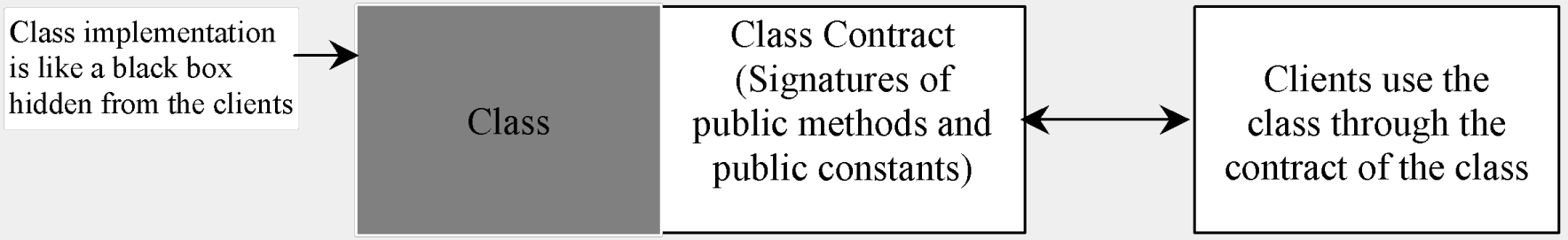
Introduction

- You see the advantages of object-oriented programming from the preceding lecture.
- This week we will learn how to solve problems using the object-oriented paradigm.

Class Abstraction and Encapsulation

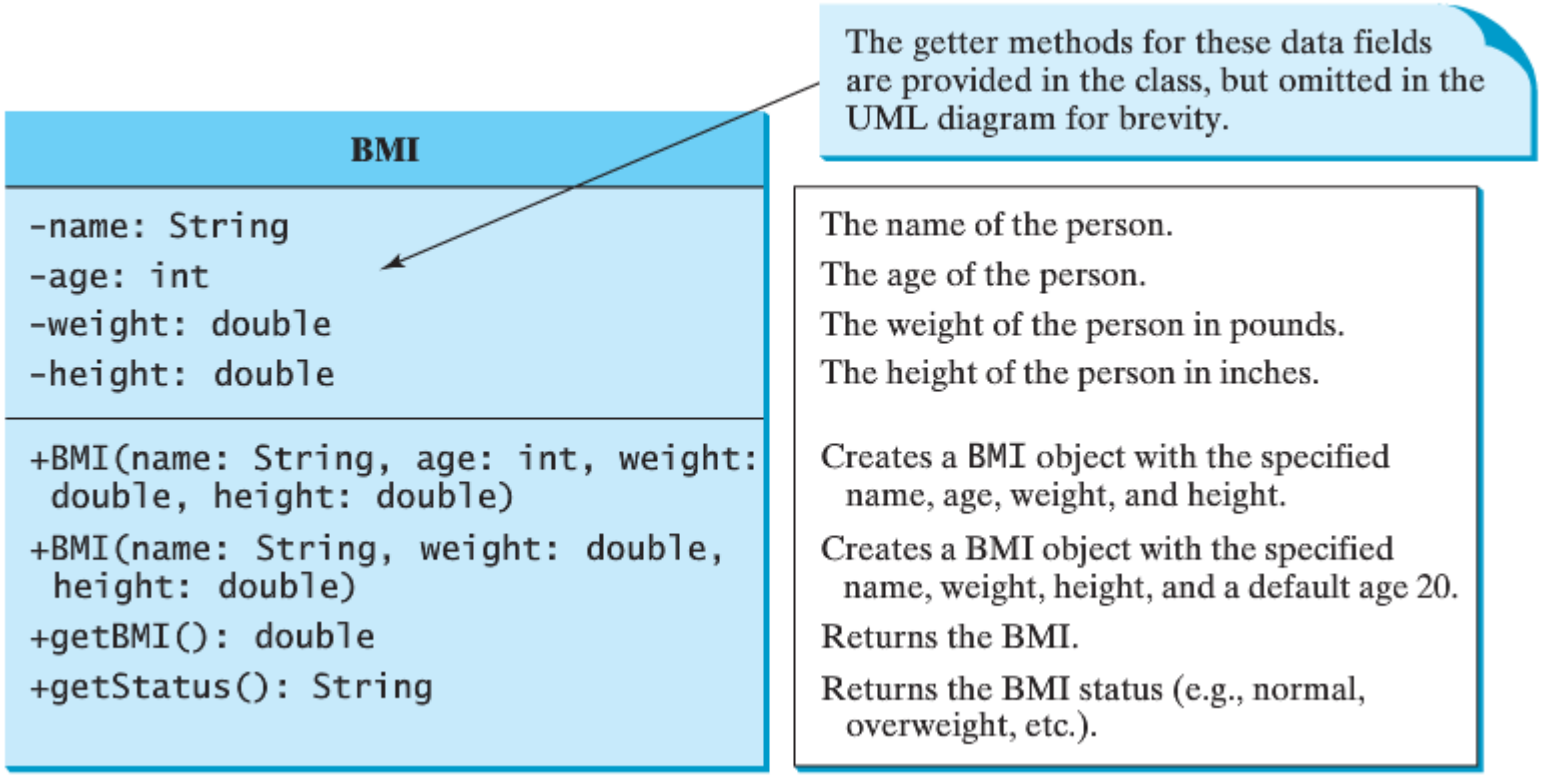
Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Thinking in Objects

The BMI Class

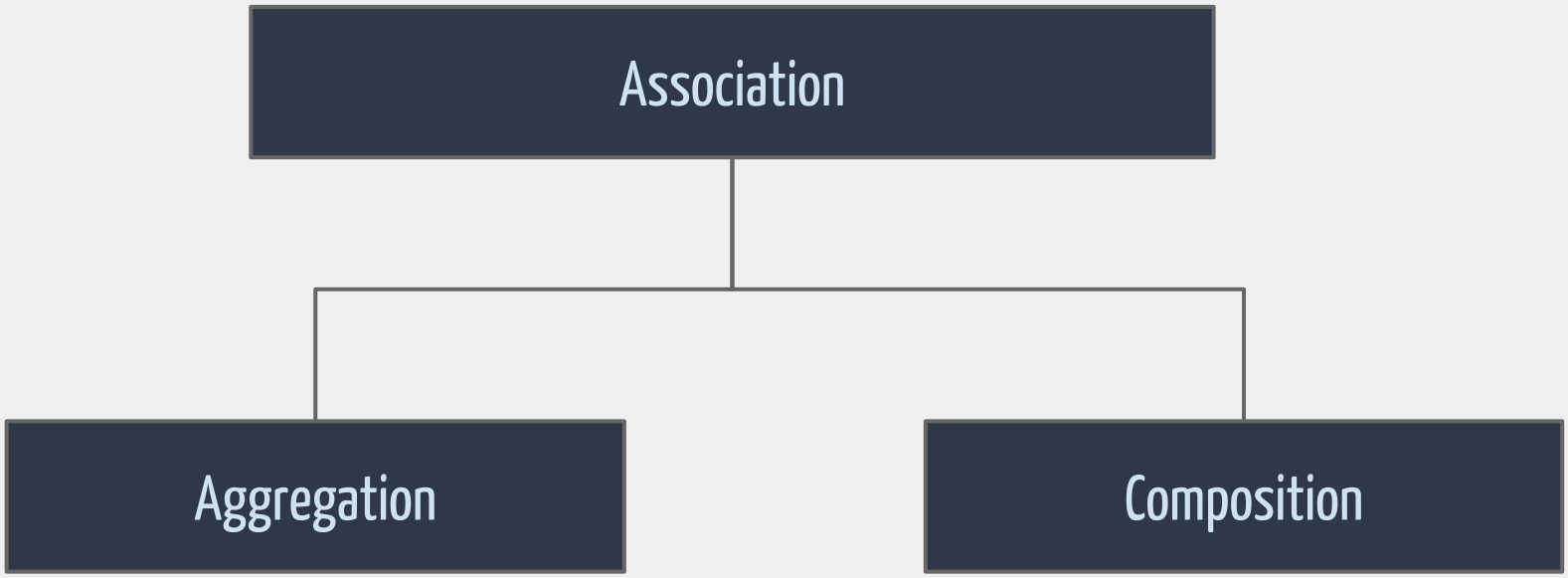


Procedural to Object-Oriented

- ComputeAndInterpretBMI.java
- BMI.java
- UseBMIClass.java

Class Relationships

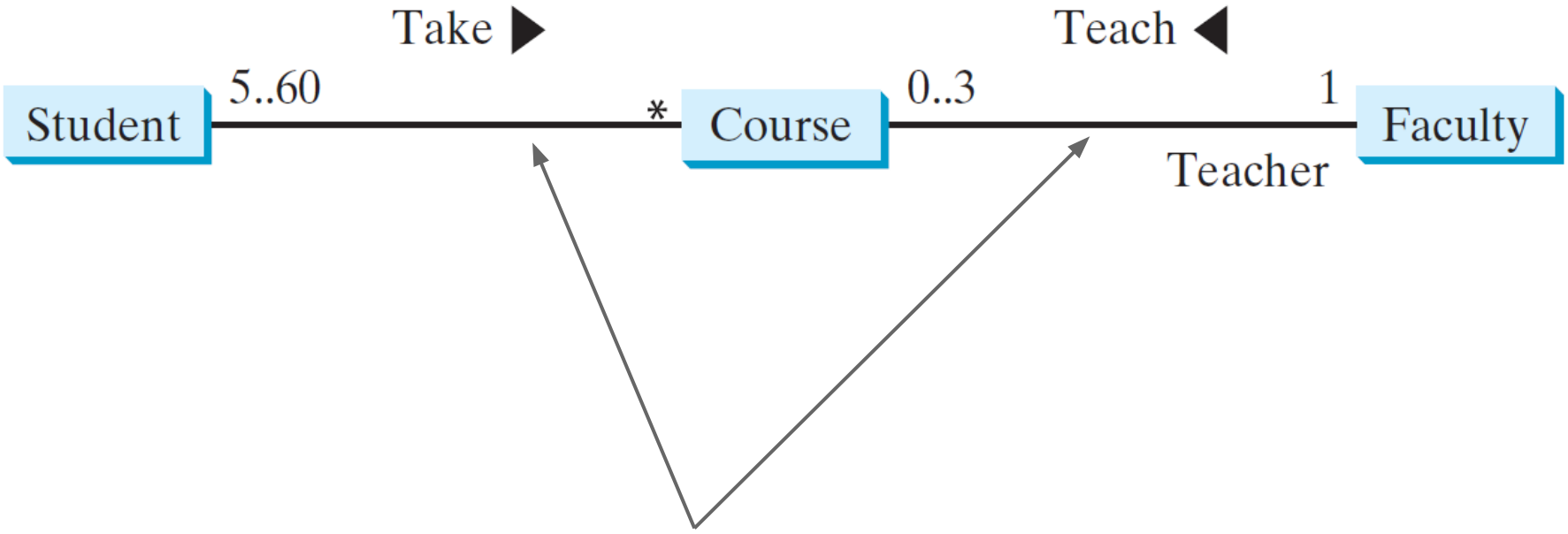
Association



Association

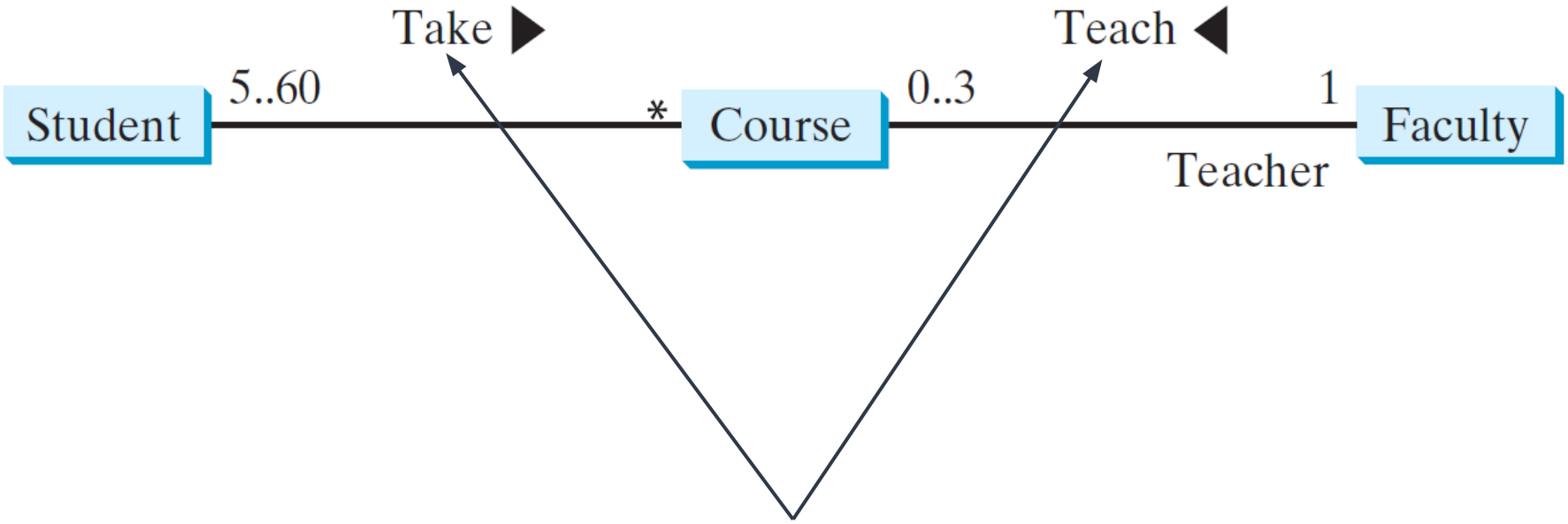
- A general binary description that describes an activity between two classes
- Student taking a course
  - association between Student and Course class
- Faculty teaching a course
  - association between Faculty and Course class

Association



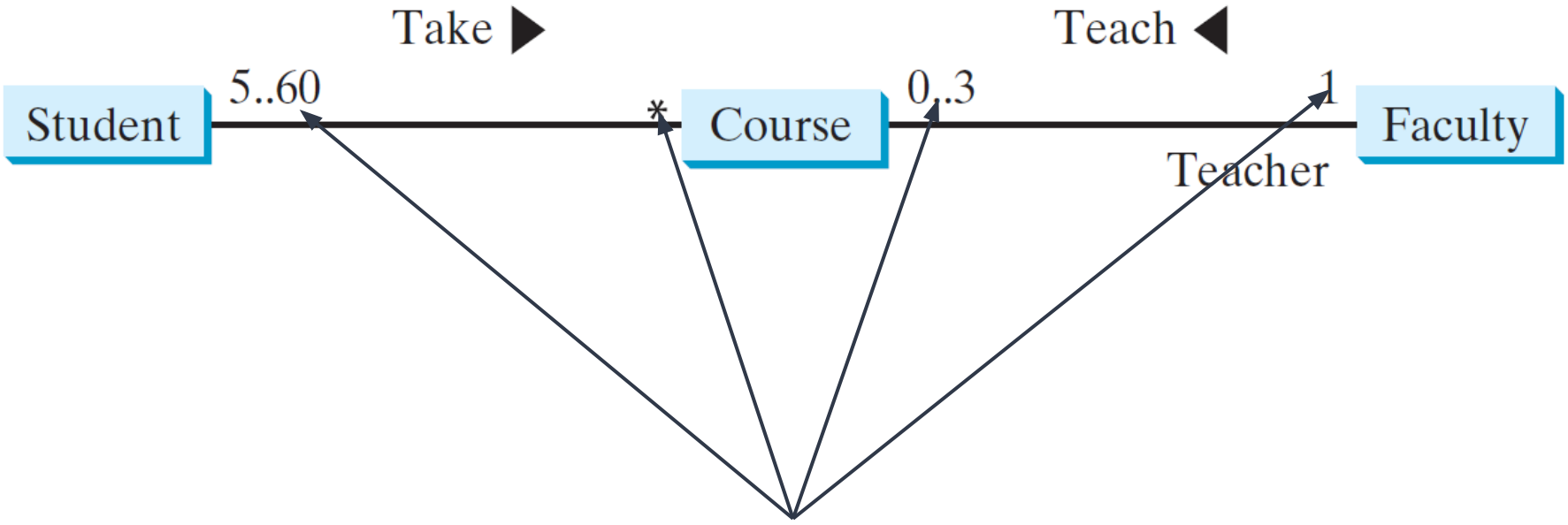
a solid line

Relationship



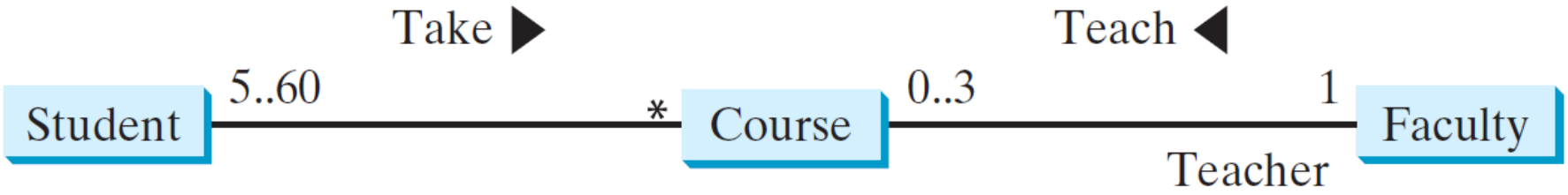
may include a small triangle indication direction of relationship

Multiplicity



- * (any number)
- m...n (range, like 2...10)
- n (number, like 5)

Association



- a student may take any number of courses
- a faculty member may teach at most three courses
- a course may have from five to sixty students
- a course is taught by only one faculty member.

Association

```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

FIGURE 10.5 The association relations are implemented using data fields and methods in classes.

Aggregation and Composition

- Aggregation models *has-a* relationships and represents an ownership relationship between two objects.
- The owner object is called an *aggregating object* and its class an *aggregating class*.
- The subject object is called an *aggregated object* and its class an *aggregated class*.

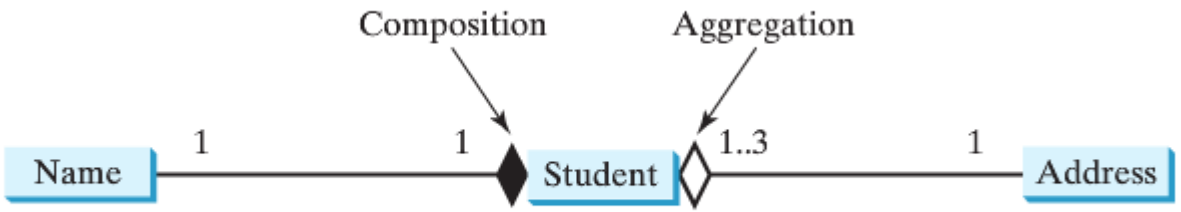
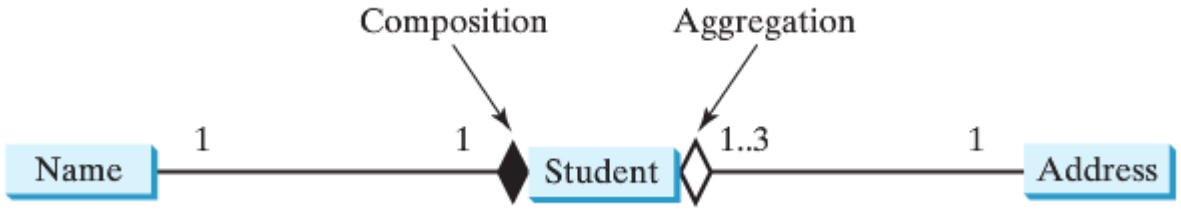


FIGURE 10.6 Each student has a name and an address.

Composition

- If an object is **exclusively owned by an aggregating object**, the relationship between the object and its aggregating object is referred to as a **composition**
  - "student has a name" is composition
  - "student has an address" is an aggregation, since an address can be shared by many students



Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.

```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

Aggregating class

```
public class Address {
  ...
}
```

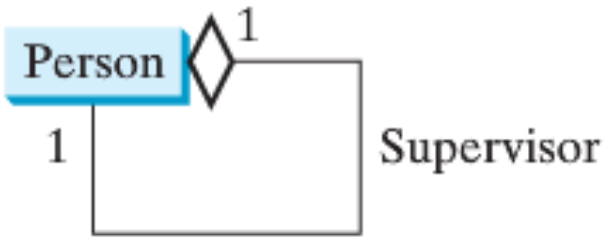
Aggregated class

Aggregation or Composition?

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both **compositions**.

Aggregation Between Same Class

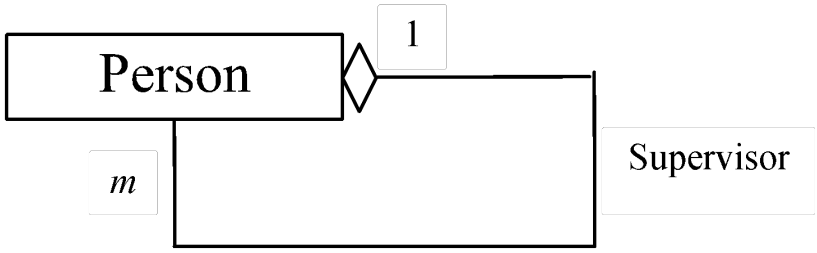
Aggregation may exist between objects of the same class. For example, a person may have a supervisor.




```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;
  ...
}
```

Aggregation Between Same Class

What happens if a person has several supervisors?

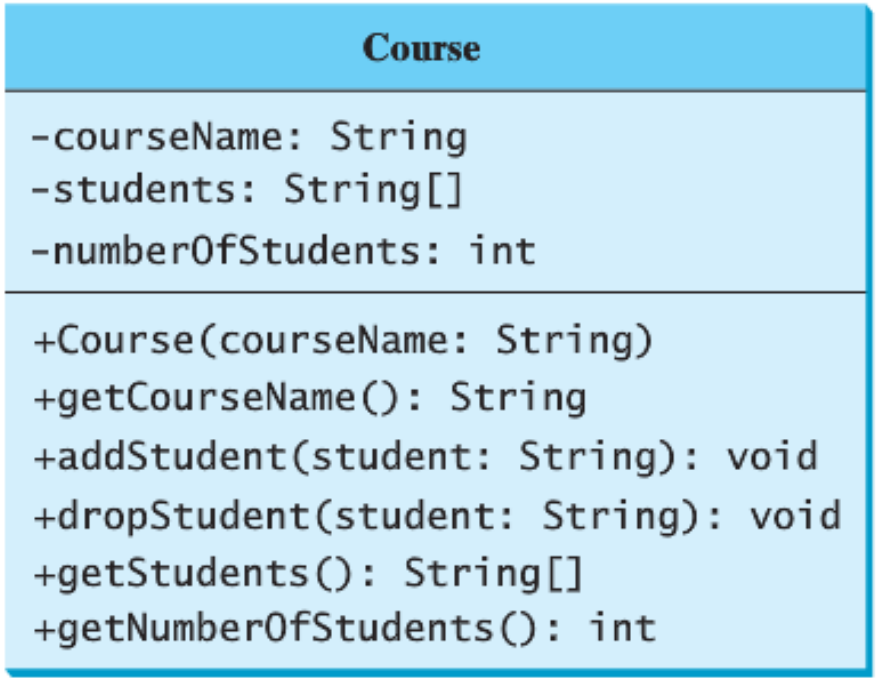



```
public class Person {
  ...
  private Person[] supervisors;

}
```

Case Study: Designing the Course Class

Example: The Course Class



The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the students for the course.

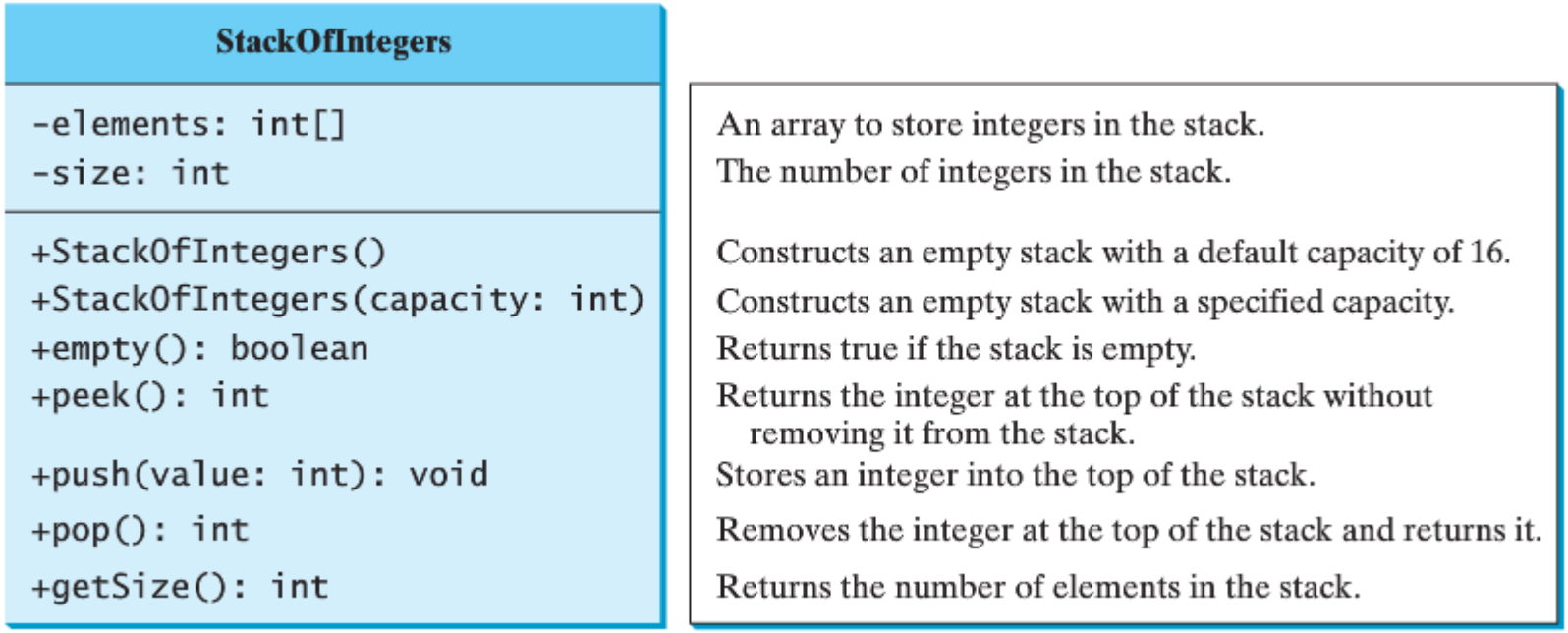
Returns the number of students for the course.

Code: The Course Class

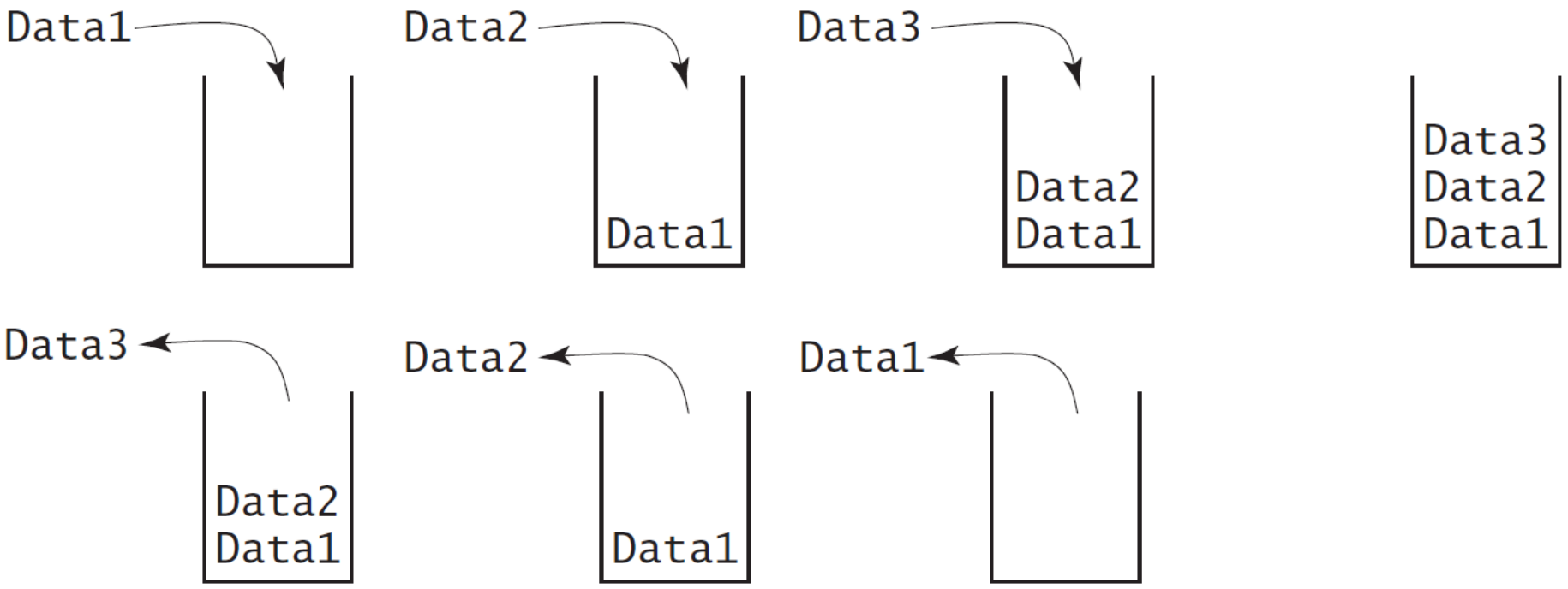
- Course.java
- TestCourse.java

Case Study: Designing a class for Stacks

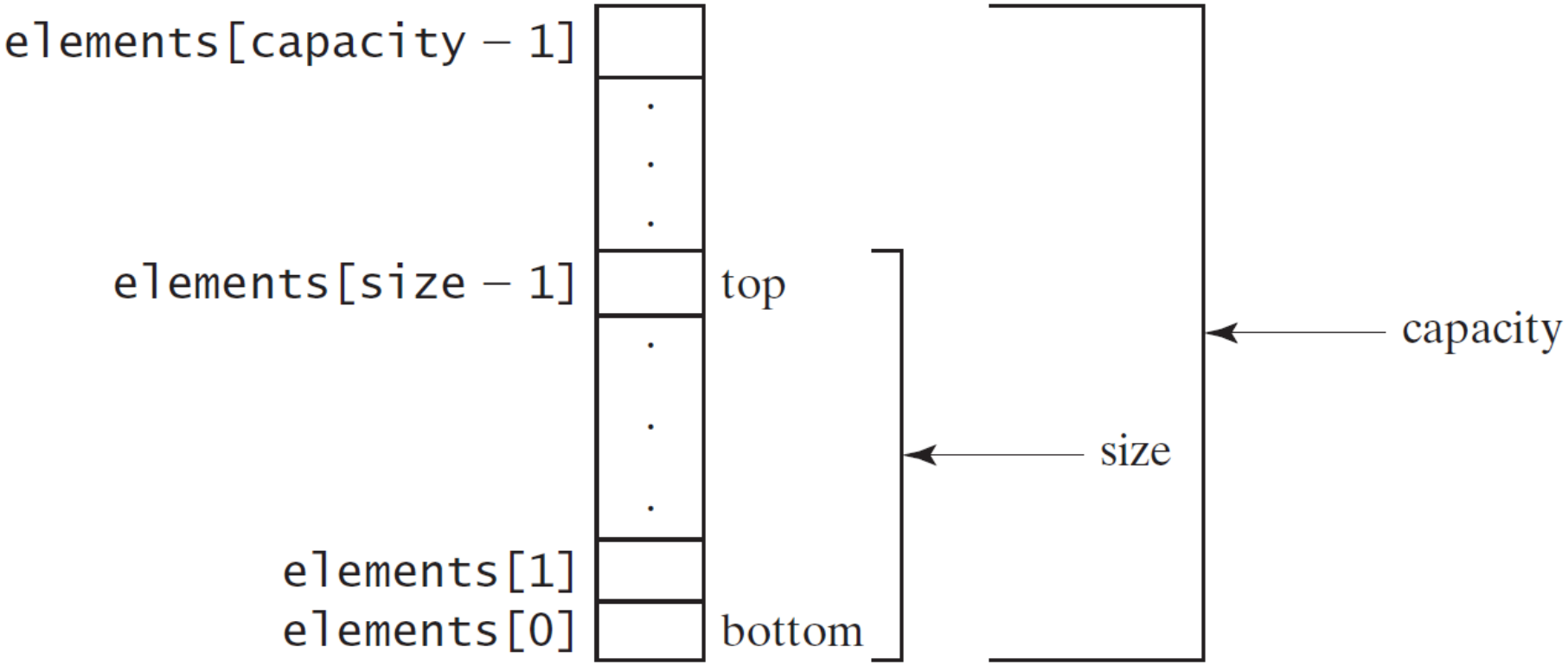
Example: The StackOfIntegers Class



Designing the StackOfIntegers Class



Implementing the StackOfIntegers Class



Code: Stack of Integers

- StackOfIntegers.java
- TestStackOfIntegers.java