School of Science & Engineering
Department of CSE
Canadian University of Bangladesh

Lecture-9: **Objects & Classes (Part III)**

Instructor: Prof. Miftahur Rahman, Ph.D.
Semester: Summer 2024

Object-Oriented Problem Solving

Objects & Classes (Part III)

Based on Chapters 9 & 10 of "Introduction to Java Programming" by Y. Daniel Liang.

Outline

- Array of Objects (9.11)
- Immutable Objects and Classes (9.12)
- The *this* reference (9.14)
- Method Abstraction and Stepwise Refinement (6.11)
- Class Abstraction and Encapsulation (10.2)
- Thinking in Objects (10.3)
- Class Relationships (10.4)
- Processing Primitive Data Type Values as Objects. (10.7 & 10.8)
- The BigInteger and BigDecimal Classes (10.9)

Array of Objects

- An array can hold objects as well as primitive type values.
- The following statement declares and creates an array of ten *Circle* objects:

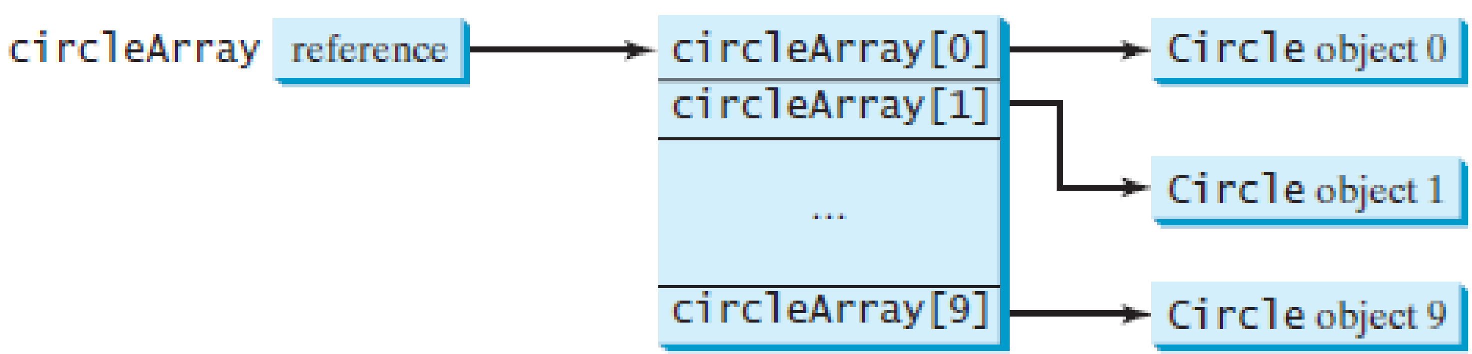
```
Circle[] circleArray = new Circle[10];
```

- To initialize *circleArray*, you can use a *for* loop like this one:

```
for (int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle();
}
```

Array of Objects (Cont.)

- An array of objects is actually an array of reference variables. So, invoking *circleArray[1].getArea()* involves two levels of referencing.
  - *circleArray* references the entire array;
  - *circleArray[1]* references a **Circle** object.
- When an array of objects is created using the *new* operator, each element in the array is a reference variable with a default value of *null*.



Array of Objects (Example)

```java
public class TotalArea {
  /** Main method */
  public static void main(String[] args) {
    // Declare circleArray
    CircleWithPrivateDataFields[] circleArray;

    // Create circleArray
    circleArray = createCircleArray();

    // Print circleArray and total areas of the circles
    printCircleArray(circleArray);
  }

  /** Create an array of Circle objects */
  public static CircleWithPrivateDataFields[] createCircleArray() {
    CircleWithPrivateDataFields[] circleArray =
      new CircleWithPrivateDataFields[5];

    for (int i = 0; i < circleArray.length; i++) {
      circleArray[i] =
        new CircleWithPrivateDataFields(Math.random() * 100);
    }

    // Return Circle array
    return circleArray;
  }

```

Array of Objects (Example Cont.)

```java
  /** Print an array of circles and their total area */
  public static void printCircleArray(
      CircleWithPrivateDataFields[] circleArray) {
    System.out.printf("%-30s%-15s\n", "Radius", "Area");
    for (int i = 0; i < circleArray.length; i++) {
      System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
        circleArray[i].getArea());
    }

    System.out.println("-----------------------------------------");

    // Compute and display the result
    System.out.printf("%-30s%-15f\n", "The total area of circles is",
     sum(circleArray) );
  }
```

Array of Objects (Example Cont.)

```java

  /** Add circle areas */
  public static double sum(CircleWithPrivateDataFields[] circleArray) {
    // Initialize sum
    double sum = 0;

    // Add areas to sum
    for (int i = 0; i < circleArray.length; i++)
      sum += circleArray[i].getArea();

    return sum;
  }
}
```

```
// LISTING 9.11 TotalArea.java
public class TotalArea {
/** Main method */
public static void main(String[] args) {
// Declare circleArray
CircleWithPrivateDataFields[] circleArray;
// Create circleArray
circleArray = createCircleArray();

// Print circleArray and total areas of the circles
printCircleArray(circleArray);
}
```

```
/** Create an array of Circle objects */
public static CircleWithPrivateDataFields[]
createCircleArray() {
CircleWithPrivateDataFields[] circleArray = new
CircleWithPrivateDataFields[5];

for (int i = 0; i < circleArray.length; i++) {
circleArray[i] = new
CircleWithPrivateDataFields(Math.random() * 100);
}
// Return Circle array
return circleArray;
}
```

```
/** Print an array of circles and their total area */
public static void printCircleArray(
CircleWithPrivateDataFields[] circleArray) {
System.out.printf("%-30s%-15s\n", "Radius", "Area");
for (int i = 0; i < circleArray.length; i++) {
System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
circleArray[i].getArea());
}
System.out.println("—————————————————————————————————————————-");
// Compute and display the result
System.out.printf("%-30s%-15f\n", "The total area of circles is",
sum(circleArray) );
}
```

```
/** Add circle areas */
public static double
sum(CircleWithPrivateDataFields[] circleArray) {
// Initialize sum
double sum = 0;

// Add areas to sum
for (int i = 0; i < circleArray.length; i++)
sum += circleArray[i].getArea();

return sum;
}
}
```

Array of Objects (Example Output)

```
Radius                         Area
70.577708                      15649.941866
44.152266                      6124.291736
24.867853                      1942.792644
 5.680718                      101.380949
36.734246                      4239.280350
-----------------------------------------------
The total area of circles is 28056.687544
```

Immutable Objects and Classes

- Normally, you create an object and allow its contents to be changed later.
- However, occasionally it is desirable to create an object <u>whose contents cannot be changed once the object has been created</u>.
  - Such an object is called *immutable object* and its class is called *immutable class*.

Immutable Objects and Classes (Cont.)

- For a class to be immutable, it must meet the following requirements:
  - All data fields must be private.
  - There can’t be any mutator methods for data fields.
  - No accessor methods can return a reference to a data field that is mutable.

Immutable Objects and Classes (Example)

```
public class Student{
    private int id;
    private String name;
    private double [] grades = new double[3];

    public Student (int ssn, String newName){
        id = ssn;
        name = newName;
    }

    public int getId(){ return id; }

    public String getName(){ return name; }

    public double [] getGrades(){
        return grades;
    }
}
```

This method actually returns a reference to the array *grades*, which means it can be changed once returned.

Immutable Objects and Classes: Example (Cont.)

```
public class test {
    public static void main(String [] args){
        Student student = new Student (112233,  “John”);
        double [] G = student.getGrades();
        G[0] = 90.0;
        G[1] = 95.5;
        G[2] = 92.9;
    }
}
```

The *this* Reference

- The keyword *this* refers to the object itself.
- The *this* *keyword* is the name of a reference that an object can use to refer to itself

```java
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

```java
public class Circle {
  private double radius;

  ...

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

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

(b)

Using *this* to Reference Hidden Data Fields

- The *this* keyword can be used to reference a class's hidden data fields.
- A hidden *static variable* can be accessed simply by using the *ClassName.staticVariable*.
- A hidden *instance variable* can be accessed by using the keyword *this*.

Using *this* to Reference Hidden Data Fields: Example

```
public class F {
  private int i = 5;
  private static double k = 0;

  public void setI(int i) {
    this.i = i;
  }

  public static void setK(double k) {
    F.k = k;
  }

  // Other methods omitted
}
```

```
Suppose that f1 and f2 are two objects of F.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2

Invoking F.setK(33) is to execute
   F.k = 33. setK is a static method
```

Using *this* to Invoke a Constructor

- The *this* keyword can be used to invoke another constructor of the same class.

```
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.

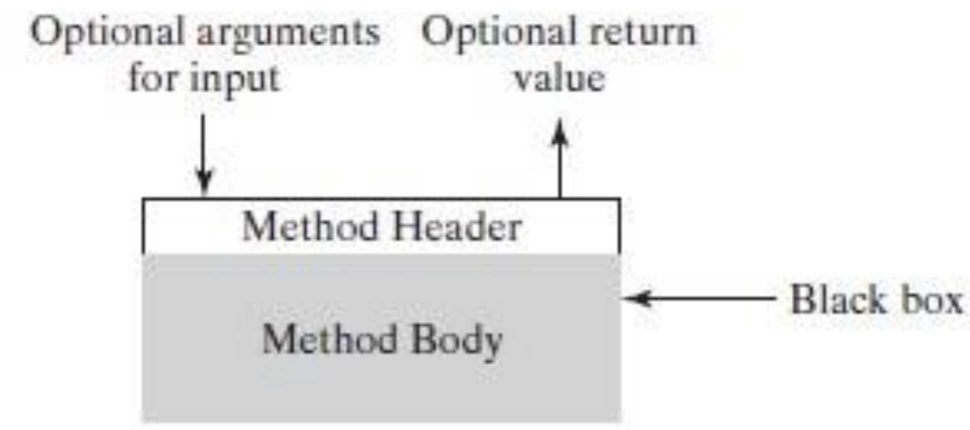
Using *this* to Invoke a Constructor Notes

- Java requires that the *this(arg-list)* statement appear first in the constructor before any other executable statements.
- If a class has multiple constructors, it is better to implement them using *this(arg-list)* as much as possible.
  - In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using *this(arg-list)*.
  - This syntax often simplifies coding and makes the class easier to read and to maintain.

Method Abstraction and Stepwise Refinement

- The key to developing software is to apply the concept of abstraction.
- *Method abstraction* is achieved by separating the use of a method from its implementation.
  - The client can use a method without knowing how it is implemented.
  - The details of the implementation are encapsulated in the method and hidden from the client who invokes the method.
  - This is also known as *information hiding* or *encapsulation.*
- If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature.

Method Abstraction and Stepwise Refinement (Cont.)



- You have already used the *System.out.print* method to display a string and the *max* method to find the maximum number.
- You know how to write the code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented.

Method Abstraction and Stepwise Refinement (Cont.)

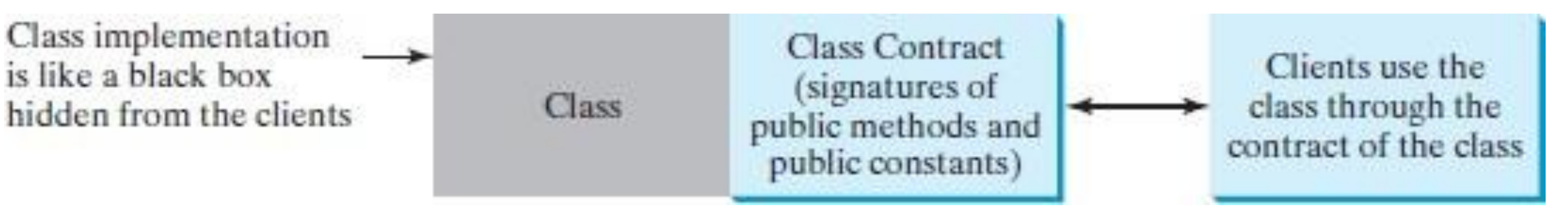
- The concept of method *abstraction* can be applied to the process of developing programs.
- When writing a large program, you can use the *divide-and-conquer* strategy, also known as *stepwise refinement*, to decompose it into subproblems.
  - The subproblems can be further decomposed into smaller, more manageable problems.

Class Abstraction and Encapsulation

- *Class abstraction* separates class implementation from how the class is used.
  - The creator of a class describes the functions of the class and lets the user know how the class can be used.
  - The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*.

Class Abstraction and Encapsulation (Cont.)

- The user of the class does not need to know how the class is implemented.
  - The details of implementation are encapsulated and hidden from the user.
  - This is called *class encapsulation*.
  - For this reason, a class is also known as an *abstract data type (ADT)*.



Thinking in Objects

- The procedural paradigm focuses on designing methods.
- The object-oriented paradigm couples data and methods together into objects.
  - Software design using the object-oriented paradigm focuses on objects and operations on objects.
  - The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.
- In procedural programming, data and operations on the data are separate, and this methodology requires passing data to methods.
- Object-oriented programming places data and the operations that pertain to them in an object.

```java
// LISTING 3.4 ComputeAndInterpretBMI.java

import java.util.Scanner;
public class ComputeAndInterpretBMI {
public static void main(String[] args) {
Scanner input = new Scanner(System.in);

// Prompt the user to enter weight in pounds
System.out.print("Enter weight in pounds: ");
double weight = input.nextDouble();

// Prompt the user to enter height in inches
System.out.print("Enter height in inches: ");
double height = input.nextDouble();

final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
final double METERS_PER_INCH = 0.0254; // Constant
```

```
// Compute BMI
double weightInKilograms = weight * KILOGRAMS_PER_POUND;
double heightInMeters = height * METERS_PER_INCH;
double bmi = weightInKilograms / (heightInMeters * heightInMeters);

// Display result
System.out.println("BMI is " + bmi);
if (bmi < 18.5)
System.out.println("Underweight");
else if (bmi < 25)
System.out.println("Normal");
else if (bmi < 30)
System.out.println("Overweight");
else
System.out.println("Obese");
}
}
```

```
Enter weight in pounds: 146 ↵Enter
Enter height in inches: 70 ↵Enter
BMI is 20.948603801493316
Normal
```

line#	weight	height	weightInKilograms	heightInMeters	bmi	output
9	146					
13		70				
19			66.22448602			
20				1.778		
21					20.9486	
25						BMI is 20.95
31						Normal

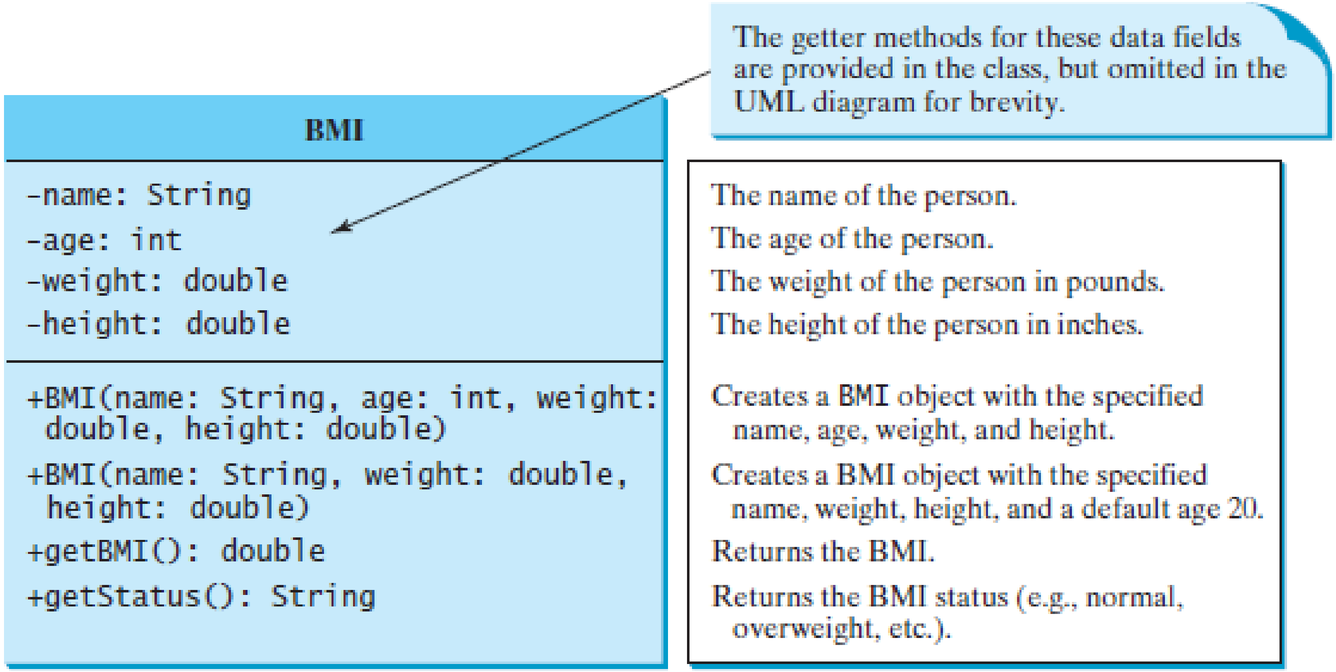
Thinking in Objects (Cont.)

- The code cannot be reused in other programs, because the code is in the *main* method.
- To make it reusable, define a static method to compute body mass index as follows:

  public static double getBMI(double weight, double height)

- This method is useful for computing body mass index for a specified weight and height.
- However, it has limitations:
  - Suppose you need to associate the weight and height with a person's name and birth date.
  - You could declare separate variables to store these values, but these values would not be tightly coupled.
- The ideal way to couple them is to create an object that contains them all.
- Since these values are tied to individual objects, they should be stored in instance data fields.

Thinking in Objects (Cont.)



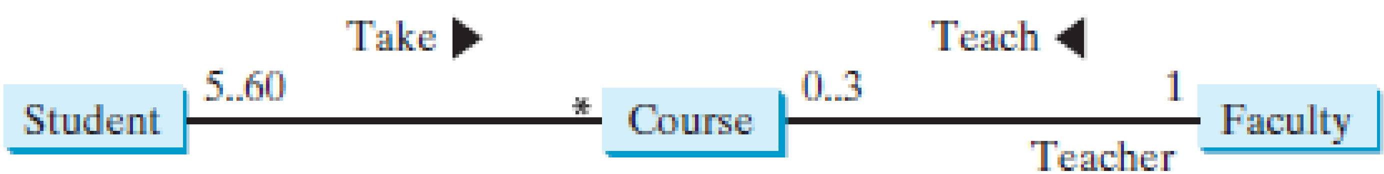

```java
public class UseBMIClass {
  public static void main(String[] args) {
    BMI bmi1 = new BMI("Kim Yang", 18, 145, 70);
    System.out.println("The BMI for " + bmi1.getName() + " is "
      + bmi1.getBMI() + " " + bmi1.getStatus());

    BMI bmi2 = new BMI("Susan King", 215, 70);
    System.out.println("The BMI for " + bmi2.getName() + " is "
      + bmi2.getBMI() + " " + bmi2.getStatus());
  }
}
```

Class Relationships

- To design classes, you need to explore the relationships among classes.
- The common relationships among classes are:
  - Association,
  - Aggregation and Composition, and
  - Inheritance.

Class Relationships
Association

- *Association* is a general binary relationship that describes an activity between two classes.
- For example:
  - A student taking a course is an association between the *Student* class and the *Course* class.
  - A faculty member teaching a course is an association between the *Faculty* class and the *Course* class.



Class Relationships Association (Cont.)

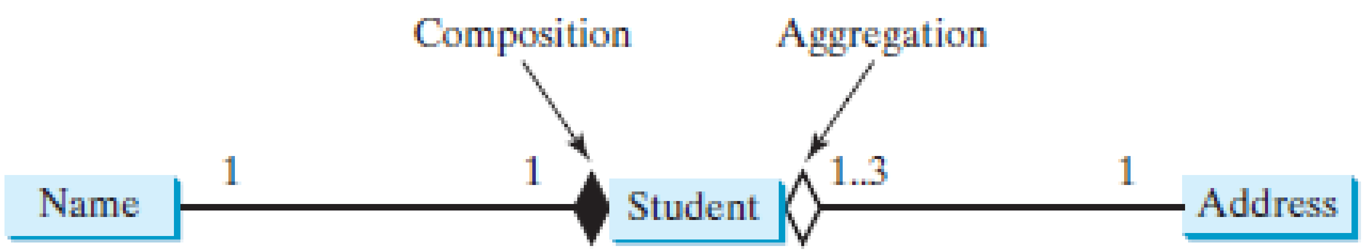
```java
public class Student {
  private Course[]
    courseList;

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

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

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

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

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

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

Class Relationships
Aggregation and Composition

- *Aggregation* is a special form of association that represents an ownership relationship between two objects.
- Aggregation models *has-a* relationships.
- An object can be owned by several other aggregating objects.
- 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*.
- For example, “a student has a name” is a composition relationship between the *Student* class and the *Name* class, whereas “a student has an address” is an aggregation relationship between the *Student* class and the *Address* class, since an address can be shared by several students.



Class Relationships
Aggregation and Composition (Cont.)

- An *aggregation* relationship is usually represented as a data field in the aggregating class.
- Since aggregation and composition relationships are represented using classes in the same way, we will not differentiate them and call both compositions for simplicity.

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

Aggregated class

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

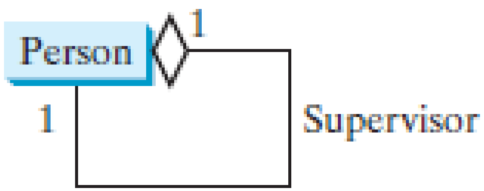
Aggregating class

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

Aggregated class

Class Relationships
Aggregation and Composition (Cont.)

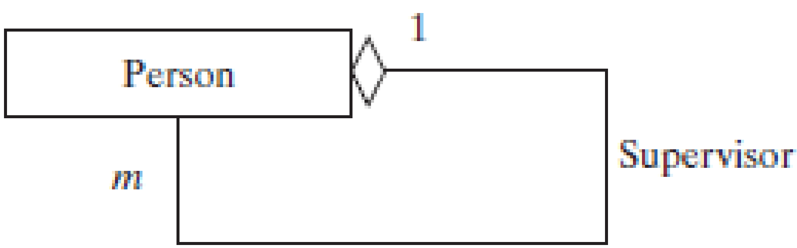
- Aggregation may exist between objects of the same class.
- In the relationship "a person has a supervisor," a supervisor can be represented as a data field in the *Person* class.




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

  ...
}
```

Class Relationships
Aggregation and Composition (Cont.)

- If a person can have several supervisors, you may use an array to store supervisors.



(a)

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

(b)

Processing Primitive Data Type Values as Objects

- Owing to performance considerations, primitive data type values are not objects in Java.
  - Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects.
- However, many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or *wrap*, a primitive data type into an object
- Wrapping *int* into the *Integer* class, wrapping *double* into the *Double* class, and wrapping *char* into the *Character* class.
- By using a wrapper class, you can process primitive data type values as objects.
  - Java provides *Boolean*, *Character*, *Double*, *Float*, *Byte*, *Short*, Integer, and *Long* wrapper classes in the *java.lang* package for primitive data types.

Processing Primitive Data Type Values as Objects (Cont.)

java.lang.Integer
-value: int +MAX_VALUE: int +MIN_VALUE: int
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Integer): int +toString(): String +valueOf(s: String): Integer +valueOf(s: String, radix: int): Integer +parseInt(s: String): int +parseInt(s: String, radix: int): int

java.lang.Double
-value: double +MAX_VALUE: double +MIN_VALUE: double
+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Double): int +toString(): String +valueOf(s: String): Double +valueOf(s: String, radix: int): Double +parseDouble(s: String): double +parseDouble(s: String, radix: int): double

Processing Primitive Data Type Values as Objects (Cont.)

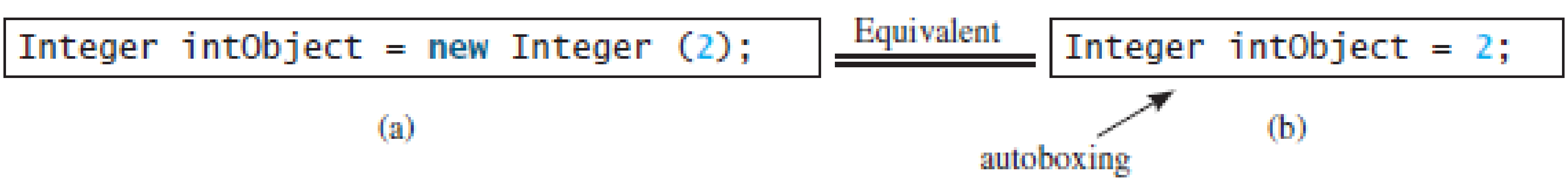
- You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value.
  - For example, *new Double(5.0)*, *new Double("5.0")*, *new Integer(5)*, and *new Integer("5")*.
- The wrapper classes do not have no-arg constructors.
- The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.
- Each numeric wrapper class has the constants *MAX_VALUE* and *MIN_VALUE*.
- Each numeric wrapper class contains the methods *doubleValue()*, *floatValue()*, *intValue()*, *longValue()*, and *shortValue()* for returning a *double*, *float*, *int*, *long*, or short value for the wrapper object.
  - *new Double(12.4).intValue()* returns *12*;
  - *new Integer(12).doubleValue()* returns *12.0*;

Processing Primitive Data Type Values as Objects (Cont.)

- The numeric wrapper classes have the static method, *valueOf (String s)*.
  - This method creates a new object initialized to the value represented by the specified string.
  - *Double doubleObject = Double.valueOf(**"12.4"**);*
  - *Integer integerObject = Integer.valueOf(**"12"**);*
- The static method *parseInt* is used to parse a numeric string into an *int* value and the *parseDouble* method in the *Double* class to parse a numeric string into a *double* value.
- Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on **10** (decimal) or any specified radix (e.g., **2** for binary, **8** for octal, and **16** for hexadecimal).
  - *Integer.parseInt("11", 2)* returns **3**;
  - *Integer.parseInt("12", 8)* returns **10**;
  - *Integer.parseInt("13", 10)* returns **13**;
  - *Integer.parseInt("1A", 16)* returns **26**;

Processing Primitive Data Type Values as Objects (Cont.)

- Converting a primitive value to a wrapper object is called *boxing*.
- The reverse conversion is called *unboxing*.
- Java allows primitive types and wrapper classes to be converted automatically.
  - The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value.
  - This is called *autoboxing* and *autounboxing*.



The *BigInteger* and *BigDecimal* Classes

- The *BigInteger* and BigDecimal classes can be used to represent integers or decimal numbers of any size and precision.
  - If you need to compute with very large integers or high-precision floating-point values, you can use the *BigInteger* and *BigDecimal* classes in the *java.math* package.
  - Both are *immutable*.
- You can use *new BigInteger(String)* and *new BigDecimal(String)* to create an instance of *BigInteger* and *BigDecimal*.
- You can use the *add*, *subtract*, *multiply*, *divide*, and *remainder* methods to perform arithmetic operations.
- The largest integer of the *long* type is *Long.MAX_VALUE* (i.e., **9223372036854775807**). An instance of *BigInteger* can represent an integer of any size.

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

The *BigInteger* and *BigDecimal* Classes (Cont.)

- There is no limit to the precision of a *BigDecimal* object.
- The *divide* method may throw an *ArithmeticException* if the result cannot be terminated.
- However, you can use the overloaded *divide(BigDecimal d, int scale, int roundingMode)* method to specify a scale and a rounding mode to avoid this exception, where *scale* is the maximum number of digits after the decimal point.
- For example, the following code creates two *BigDecimal* objects and performs division with scale **20** and rounding mode *BigDecimal.ROUND_UP*.
  - The output is **0.33333333333333333334**.

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The *BigInteger* and *BigDecimal* Classes (Cont.)

```
// LISTING 10.9 LargeFactorial.java
import java.math.*;
public class LargeFactorial {
public static void main(String[] args) {
System.out.println("50! is \n" + factorial(50));
}
public static BigInteger factorial(long n) {
BigInteger result = BigInteger.ONE;
for (int i = 1; i <= n; i++)
result = result.multiply(new BigInteger(i + ""));

return result;
}
}
```

```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```