Chapter 4
Introduction to object-oriented programming

OBJECTS AND CLASSES

1 Introduction

Introduce object-oriented programming, which you can use to develop GUI and large-scale software systems.

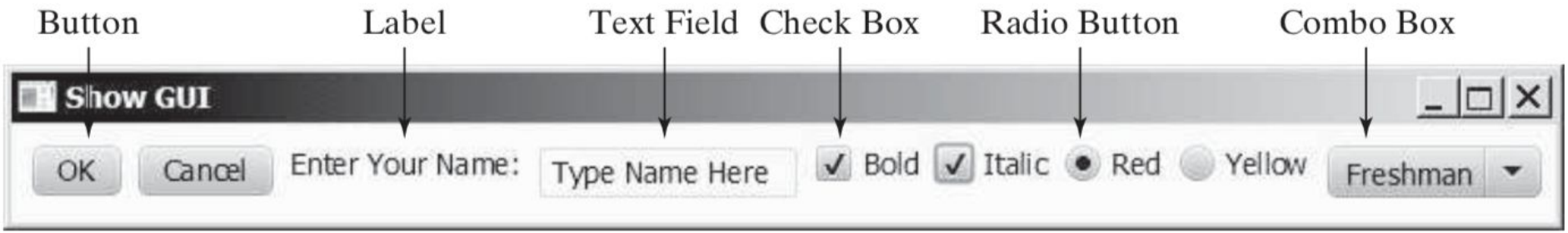


FIGURE 9.1 The GUI objects are created from classes.

2 Defining Classes for Objects

- Object-oriented programming (OOP) involves programming using objects.
- An object represents an entity in the real world, e.g., student, a desk, a circle, a button, …
- An object has a unique identity, state, and behavior.
  - The state of an object (also known as its properties or attributes) is represented by data fields with their current values.
    - A circle object, for example, has a data field radius, which is the property that characterizes a circle.
  - The behavior of an object (also known as its actions) is defined by methods. To invoke a method on an object is to ask the object to perform an action.
    - You may define methods named getArea() and getPerimeter() for circle objects.
    - A circle object may invoke getArea() to return its area and getPerimeter() to return its perimeter.

2 Defining Classes for Objects

- Objects of the same type are defined using a common class.
- A class is a template that defines what an object's data fields and methods will be.
- An object is an instance of a class.

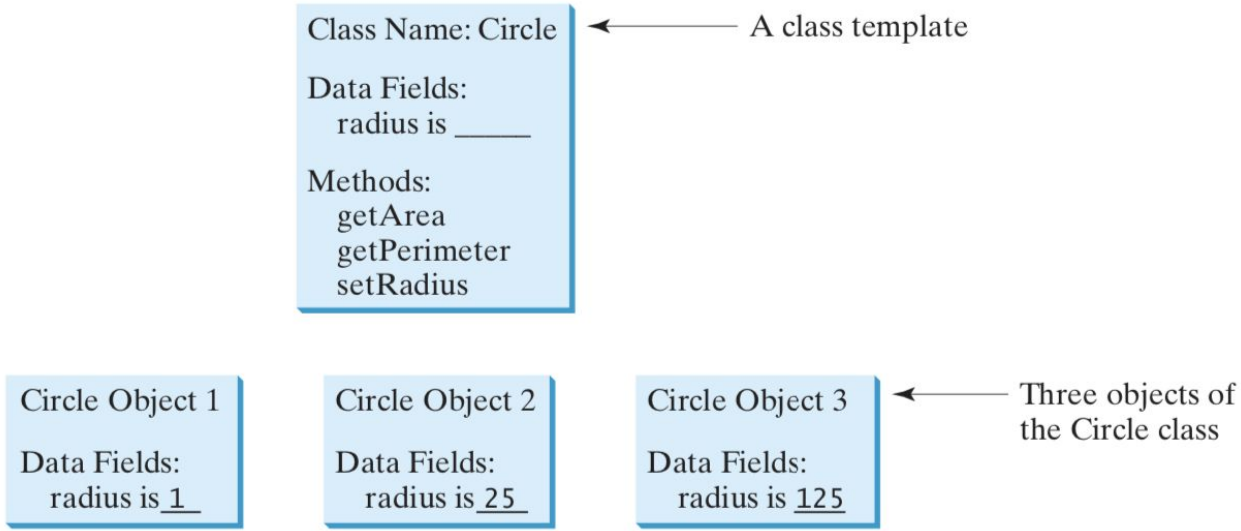


FIGURE 9.2 A class is a template for creating objects.

2 Defining Classes for Objects

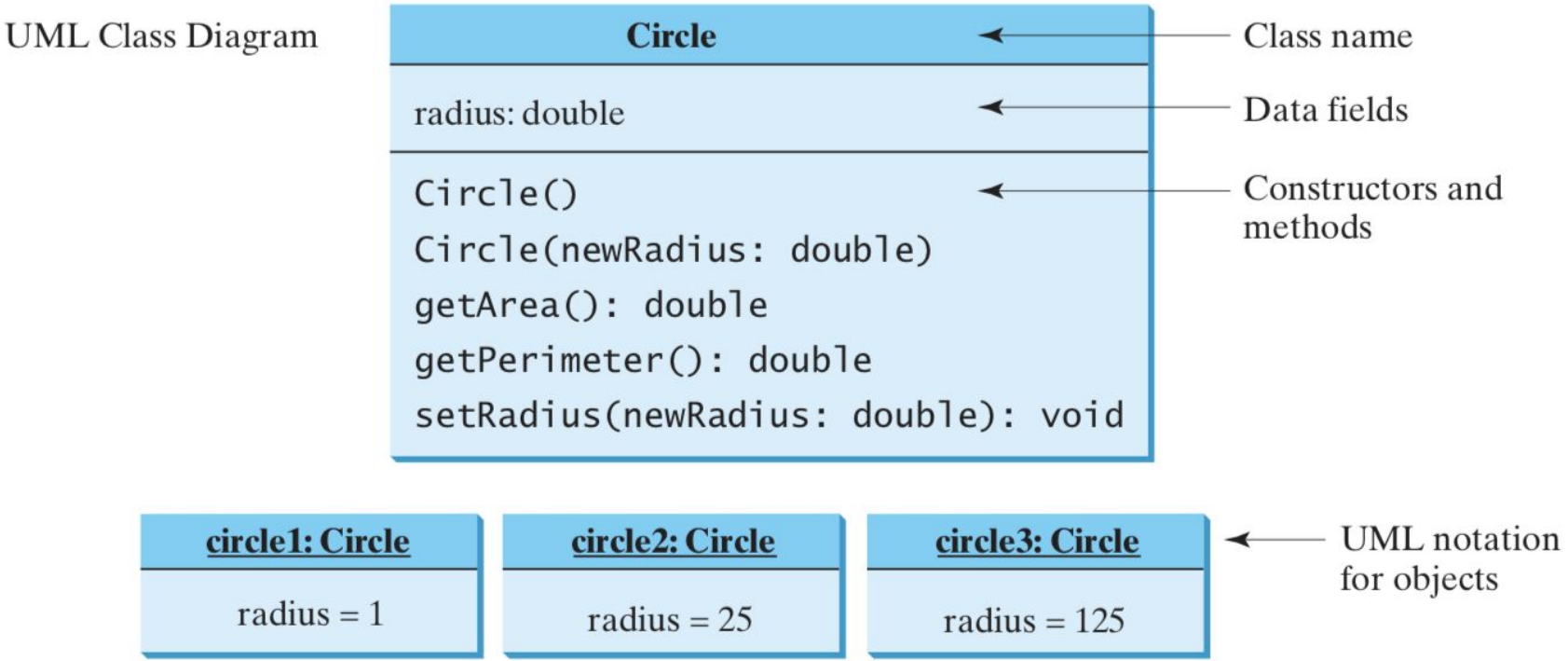


FIGURE 9.4 Classes and objects can be represented using UML notation.

2 Defining Classes for Objects

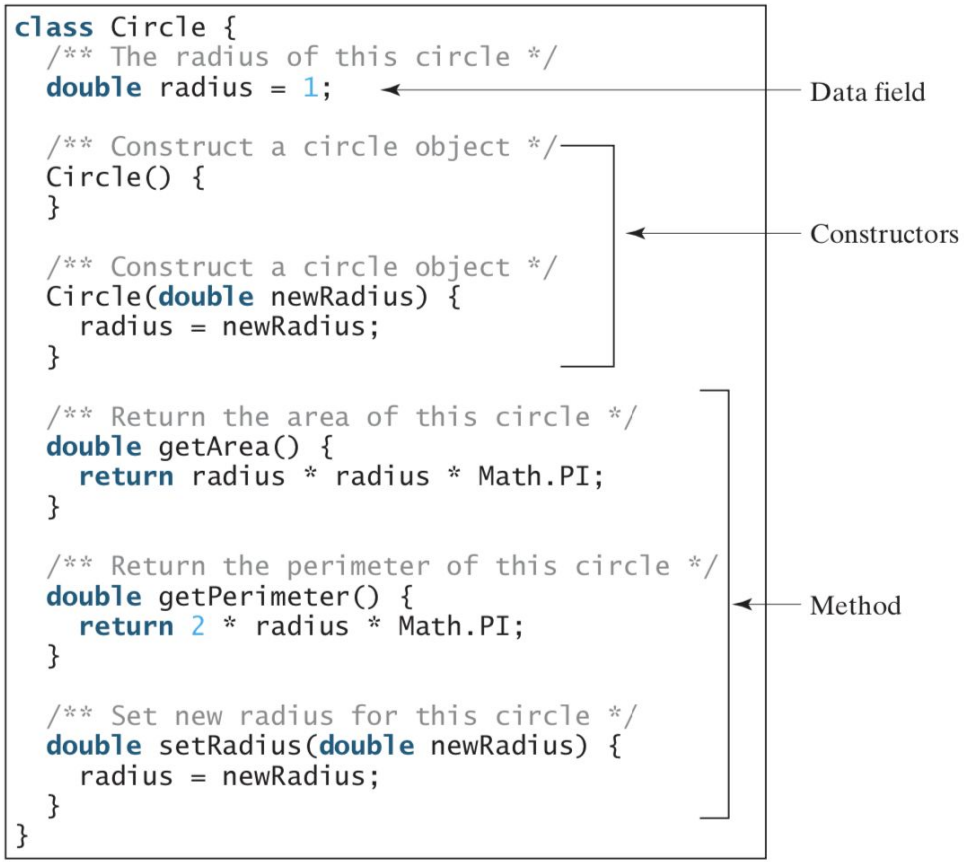
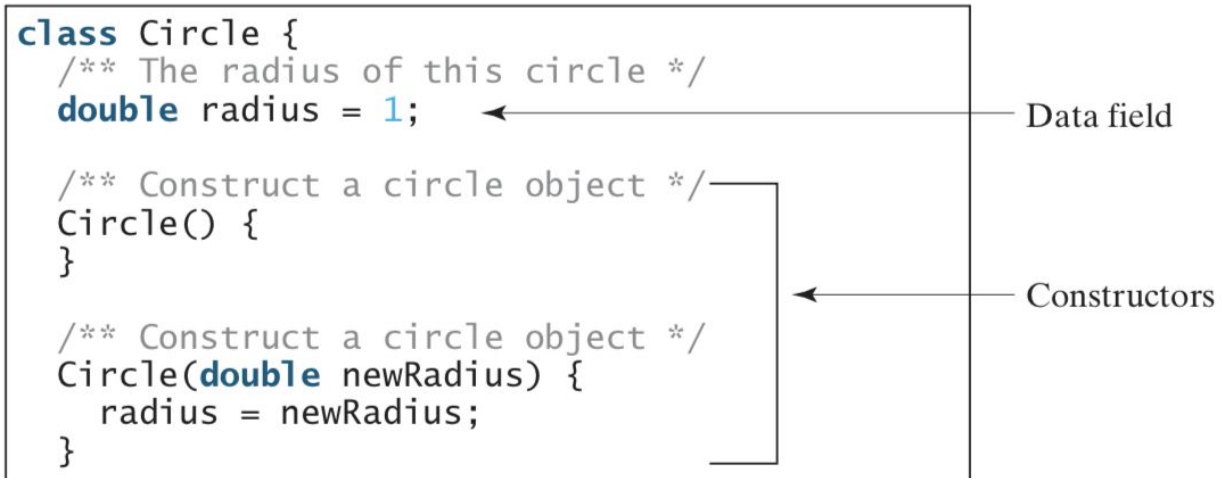


FIGURE 9.3 A class is a construct that defines objects of the same type.

3 Constructing Objects Using Constructors

Constructors are a special kind of method. Constructors play the role of initializing objects.

- A constructor must have the same name as the class itself.
- Constructors do not have a return type—not even *void*.
- Constructors are invoked using the *new* operator when an object is created.



4 Accessing Objects via Reference Variables

- A class is a programmer-defined type. A class is a reference type.
- Objects are accessed via their reference variables

```
Circle myCircle = new Circle();
```

The variable myCircle holds a reference to a Circle object.

4 Accessing Objects via Reference Variables

Accessing an Object's Data and Methods

After an object is created, its data can be accessed and its methods can be invoked using the dot operator (.)

- objectRefVar.dataField references a data field in the object.
- objectRefVar.method(arguments) invokes a method on the object.

```
Circle myCircle = new Circle();
myCircle.radius
myCircle.getArea()
```

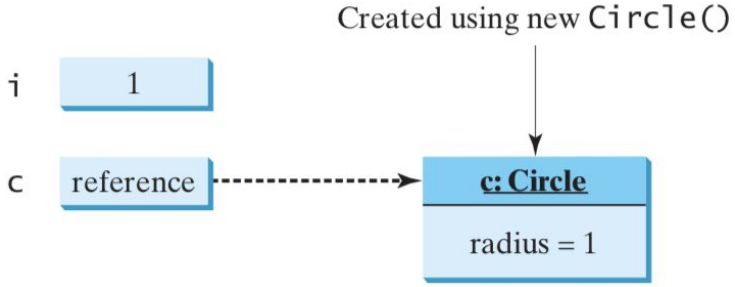
4 Accessing Objects via Reference Variables

Differences between Variables of Primitive Types and Reference Types

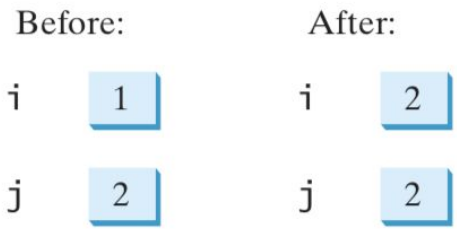
- Every variable represents a memory location that holds a value.
- When you declare a variable, you are telling the compiler what data type of variable.
- When you assign one variable to another,
  - For a variable of a primitive type, the real value of one variable is assigned to the other variable.
  - For a variable of a reference type, the reference of one variable is assigned to the other variable.

4 Accessing Objects via Reference Variables

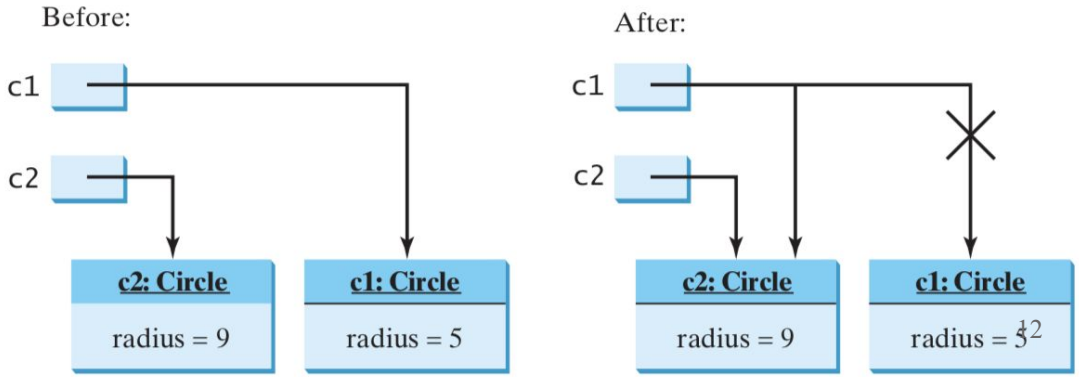
```
int i=1;
Circle c = new Circle(1);
```




```
int i=1;
int j=2;
i=j
```




```
Circle c1 = new Circle(5);
Circle c2 = new Circle(9);
c1=c2;
```



5 Using Classes from the Java Library

The Java API contains a rich set of classes for developing Java programs.

- The Date Class
- The Random Class
- The Point2D Class
- …

5 Using Classes from the Java Library

The Date Class

```
java.util.Date date = new java.util.Date();
System.out.println(date.toString());
```

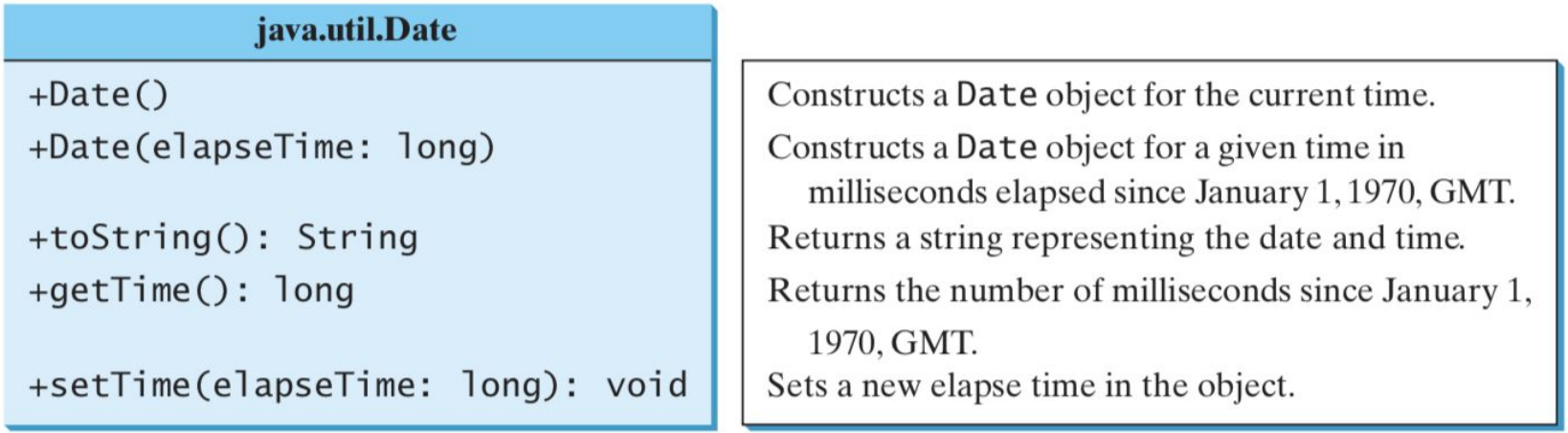


FIGURE 9.10 A `Date` object represents a specific date and time.

5 Using Classes from the Java Library

The Random Class

```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
        System.out.print(random1.nextInt(1000) + " ");
```

java.util.Random
+Random()
+Random(seed: long)
+nextInt(): int
+nextInt(n: int): int
+nextLong(): long
+nextDouble(): double
+nextFloat(): float
+nextBoolean(): boolean

Constructs a `Random` object with the current time as its seed.
Constructs a `Random` object with a specified seed.
Returns a random `int` value.
Returns a random `int` value between `0` and `n` (excluding `n`).
Returns a random `long` value.
Returns a random `double` value between `0.0` and `1.0` (excluding `1.0`).
Returns a random `float` value between `0.0F` and `1.0F` (excluding `1.0F`).
Returns a random `boolean` value.

FIGURE 9.11 A `Random` object can be used to generate random values.

6 Static Variables, Constants, and Methods

- The data field *radius* in the circle class is known as an instance variable.
  An instance variable is tied to a specific instance of the class.

  Circle circle1 = new Circle();
  Circle circle2 = new Circle(5);
  Radius of circle1 is independent of radius of circle2.

- If you want all the instances of a class to share data, use *static variables*, also known as *class variables*.

- Static methods can be called without creating an instance of the class.

6 Static Variables, Constants, and Methods

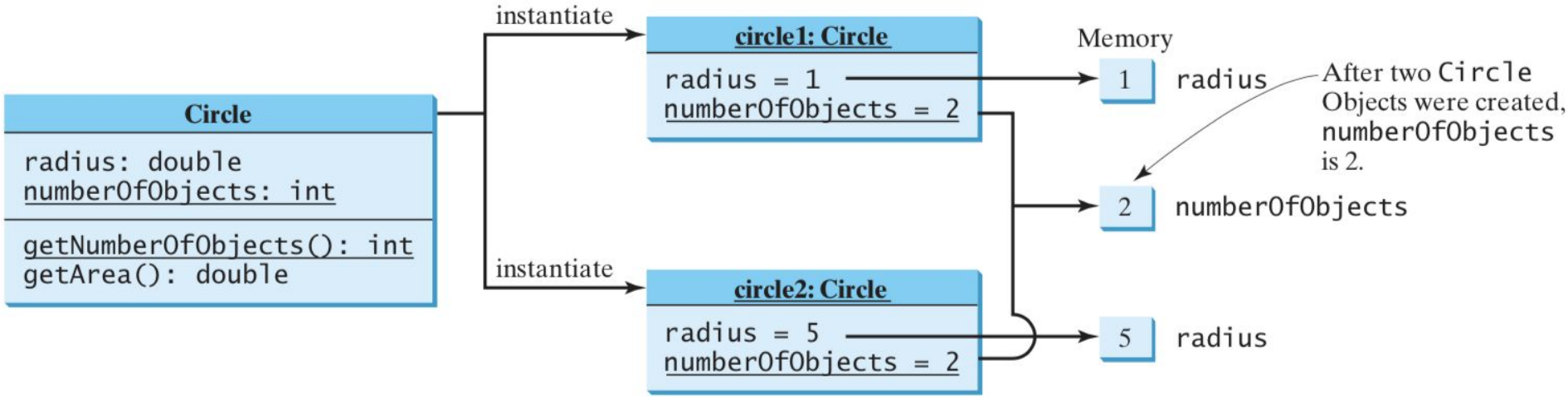
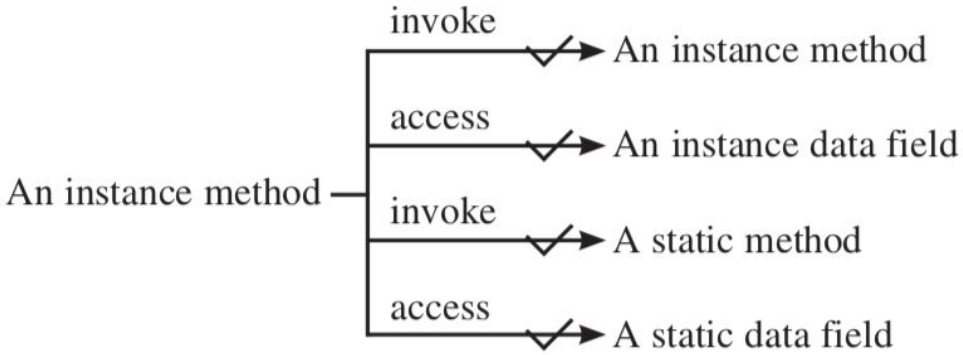
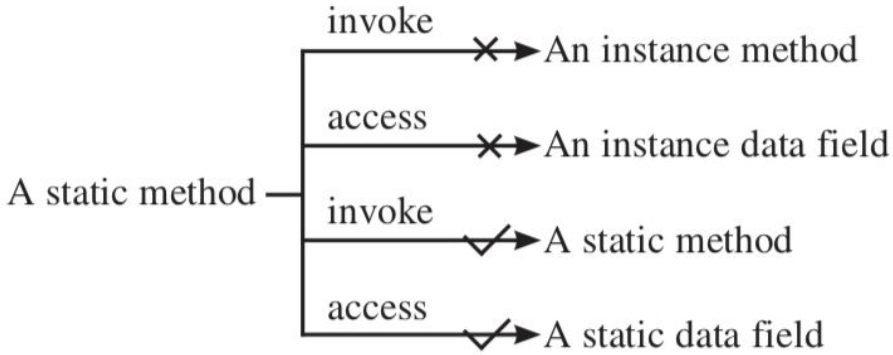


FIGURE 9.13 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

6 Static Variables, Constants, and Methods

Use ClassName.methodName(arguments) to invoke a static method and ClassName.staticVariable to access a static variable.





6 Static Variables, Constants, and Methods

LISTING 9.6 CircleWithStaticMembers.java

```
1  public class CircleWithStaticMembers {
2    /** The radius of the circle */
3    double radius;
4
5    /** The number of objects created */
6    static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    CircleWithStaticMembers() {
10     radius = 1;
11     numberOfObjects++;
12   }
13
14   /** Construct a circle with a specified radius */
15   CircleWithStaticMembers(double newRadius) {
16     radius = newRadius;
17     numberOfObjects++;
18   }
19
20   /** Return numberOfObjects */
21   static int getNumberOfObjects() {
22     return numberOfObjects;
23   }
24
25   /** Return the area of this circle */
26   double getArea() {
27     return radius * radius * Math.PI;
28   }
29 }
```

LISTING 9.7 TestCircleWithStaticMembers.java

```
1  public class TestCircleWithStaticMembers {
2    /** Main method */
3    public static void main(String[] args) {
4      System.out.println("Before creating objects");
5      System.out.println("The number of Circle objects is " +
6        CircleWithStaticMembers.numberOfObjects);
7
8      // Create c1
9      CircleWithStaticMembers c1 = new CircleWithStaticMembers();
10
11     // Display c1 BEFORE c2 is created
12     System.out.println("\nAfter creating c1");
13     System.out.println("c1: radius (" + c1.radius +
14       ") and number of Circle objects (" +
15       c1.numberOfObjects + ")");
16
17     // Create c2
18     CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);
19
20     // Modify c1
21     c1.radius = 9;
22
23     // Display c1 and c2 AFTER c2 was created
24     System.out.println("\nAfter creating c2 and modifying c1");
25     System.out.println("c1: radius (" + c1.radius +
26       ") and number of Circle objects (" +
27       c1.numberOfObjects + ")");
28     System.out.println("c2: radius (" + c2.radius +
29       ") and number of Circle objects (" +
30       c2.numberOfObjects + ")");
31   }
32 }
```

7 Visibility Modifiers

Visibility modifiers can be used to specify the visibility of a class and its members.

- public: accessible to any other classes
- no visibility modifier: accessible to any class in the same package
- private: accessible only from within its own class

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

FIGURE 9.14 The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

7 Visibility Modifiers

If a class is not defined as public, it can be accessed only within the same package.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

FIGURE 9.15 A nonpublic class has package-access.

8 Data Field Encapsulation

- Making data fields private protects data and makes the class easy to maintain.
- If a client often needs to retrieve and modify a data field
  - provide a getter method (or accessor) to return its value.
  - provide a setter method (or mutator) to set a new value.

getter method

public returnType getPropertyName()

If the returnType is boolean:

public boolean isPropertyName()

setter method

public void setPropertyName(dataType propertyValue)

8 Data Field Encapsulation

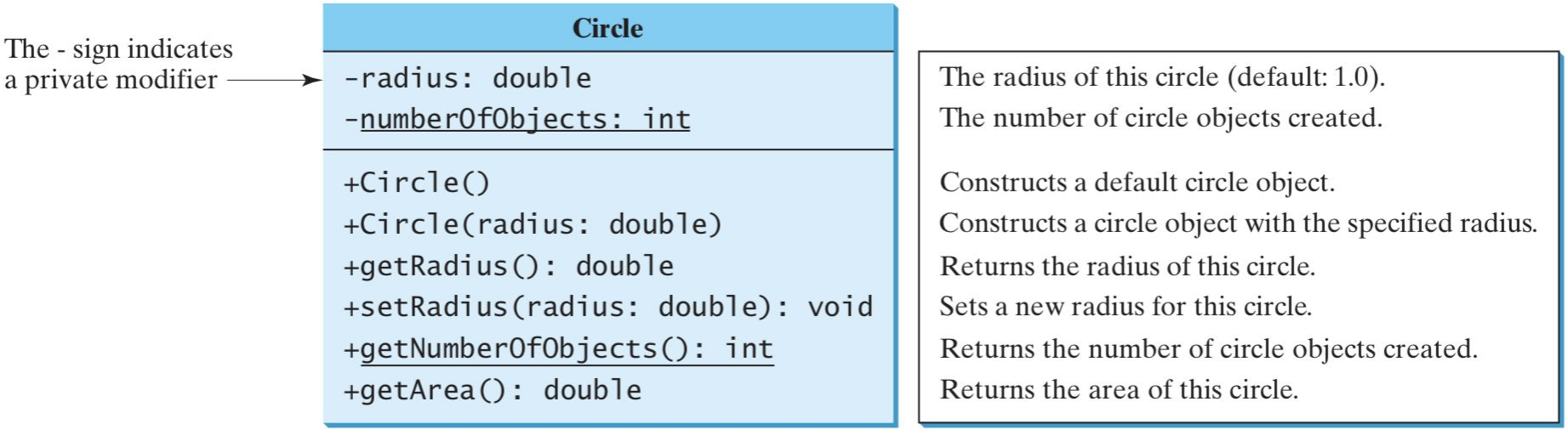


FIGURE 9.17 The `Circle` class encapsulates circle properties and provides getter/setter and other methods.

9 Passing Objects to Methods

Passing an object to a method is to pass the reference of the object.

LISTING 9.10 TestPassObject.java

```
1  public class TestPassObject {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create a Circle object with radius 1
5      CircleWithPrivateDataFields myCircle =
6        new CircleWithPrivateDataFields(1);
7
8      // Print areas for radius 1, 2, 3, 4, and 5.
9      int n = 5;
10     printAreas(myCircle, n);
11
12     // See myCircle.radius and times
13     System.out.println("\n" + "Radius is " + myCircle.getRadius());
14     System.out.println("n is " + n);
15   }
16
17   /** Print a table of areas for radius */
18   public static void printAreas(
19       CircleWithPrivateDataFields c, int times) {
20     System.out.println("Radius \t\tArea");
21     while (times >= 1) {
22       System.out.println(c.getRadius() + "\t\t" + c.getArea());
23       c.setRadius(c.getRadius() + 1);
24       times--;
25     }
26   }
27 }
```

10 Array of Objects

An array can hold objects as well as primitive type values.

```
Circle[] circleArray = new Circle[10];
for (int i = 0; i < circleArray.length; i++) {
        circleArray[i] = new Circle();
}
```

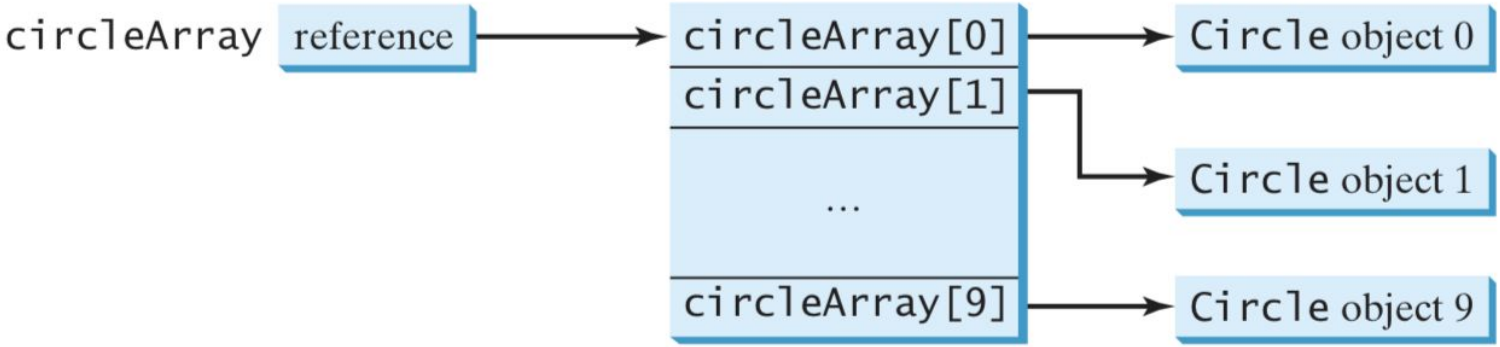


FIGURE 9.19 In an array of objects, an element of the array contains a reference to an object.

11 Immutable Objects and Classes

- You can define immutable classes to create immutable objects.
- The contents of immutable objects cannot be changed once the objects has been created.

```
1  public class Student {
2    private int id;
3    private String name;
4    private java.util.Date dateCreated;
5
6    public Student(int ssn, String newName) {
7      id = ssn;
8      name = newName;
9      dateCreated = new java.util.Date();
10   }
11
12   public int getId() {
13     return id;
14   }
15
16   public String getName() {
17     return name;
18   }
19
20   public java.util.Date getDateCreated() {
21     return dateCreated;
22   }
23 }
```

12 The Scope of Variables

- The scope of instance and static variables is the entire class, regardless of where the variables are declared.
- The exception is when a data field is initialized based on a reference to another data field.

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) The variable `radius` and method `findArea()` can be declared in any order.

```
public class F {
  private int i ;
  private int j = i + 1;
}
```

(b) `i` has to be declared before `j` because `j`'s initial value is dependent on `i`.

FIGURE 9.20 Members of a class can be declared in any order, with one exception.

13 The this Reference

- The keyword *this* refers to the object itself.
- It can also be used inside a constructor to invoke another constructor of the same class.

```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)

OBJECT-ORIENTED THINKING

1 Introduction

1 Introduction

- This chapter shows how procedural and object-oriented programming differ.
- You see the benefits of object-oriented programming and learn to use it effectively.

2 Class Abstraction and Encapsulation

- Class contract: the collection of methods and fields that are accessible from outside the class
- Class abstraction: separates class implementation from how the class is used.
  - The user of the class use the class through the contract of the class and does not need to know how the class is implemented.
- Class encapsulation
  - The details of implementation are encapsulated and hidden from the user.

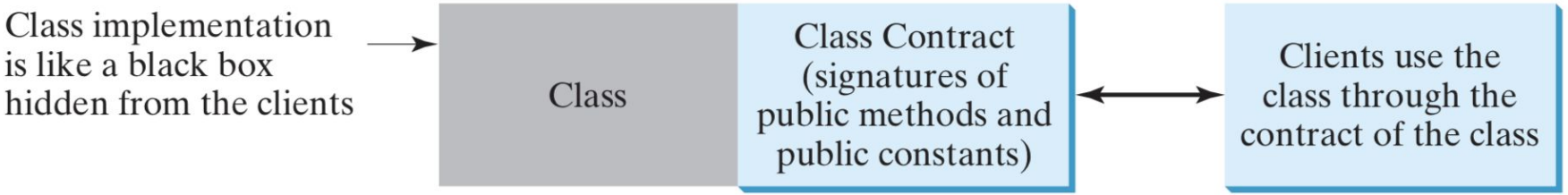


FIGURE 10.1 Class abstraction separates class implementation from the use of the class.

2 Class Abstraction and Encapsulation

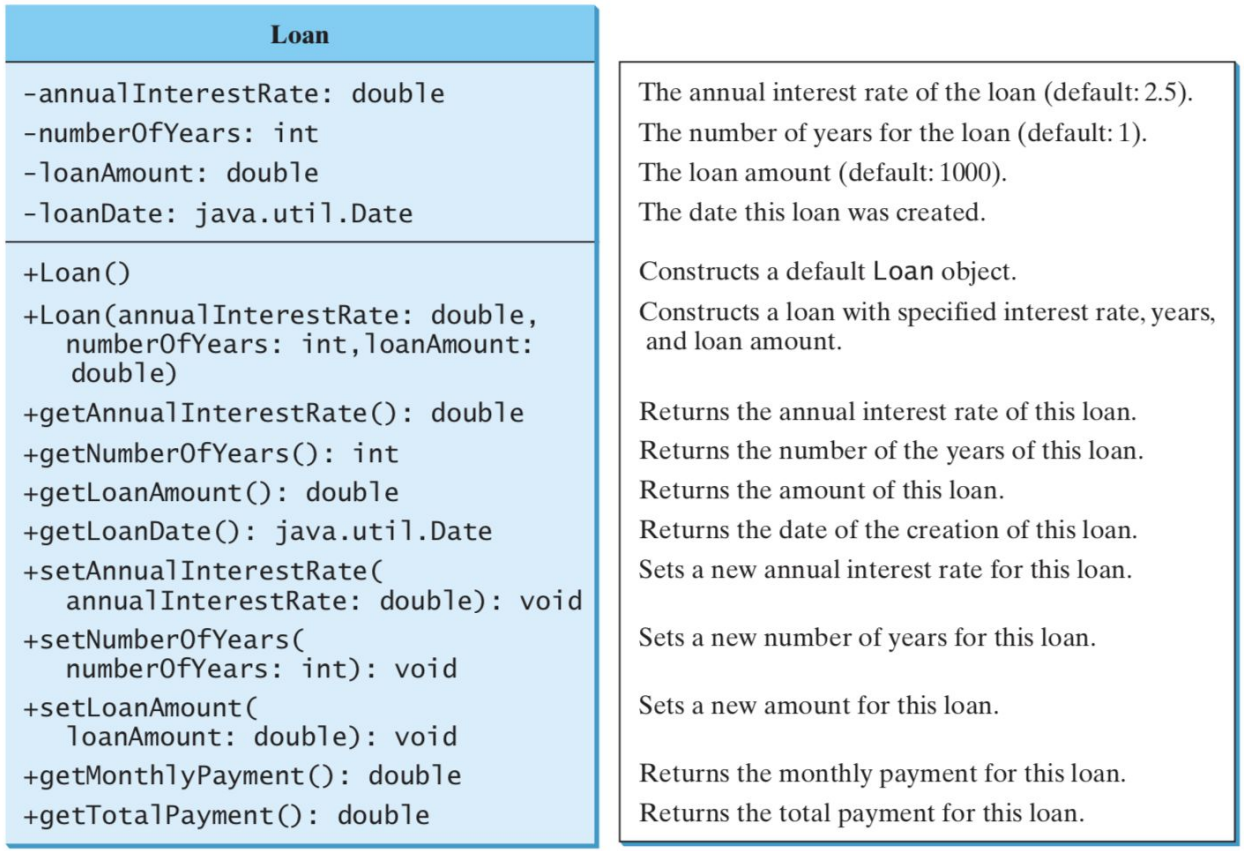


FIGURE 10.2 The Loan class models the properties and behaviors of loans.

2 Clas

```
public class TestLoanClass {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Enter yearly interest rate
    System.out.print(
      "Enter annual interest rate, for example, 8.25: ");
    double annualInterestRate = input.nextDouble();

    // Enter number of years
    System.out.print("Enter number of years as an integer: ");
    int numberOfYears = input.nextInt();

    // Enter loan amount
    System.out.print("Enter loan amount, for example, 120000.95: ");
    double loanAmount =  input.nextDouble();

    // Create Loan object
    Loan loan =
      new Loan(annualInterestRate, numberOfYears, loanAmount);

    // Display loan date, monthly payment, and total payment
    System.out.printf("The loan was created on %s\n" +
      "The monthly payment is %.2f\nThe total payment is %.2f\n",
      loan.getLoanDate().toString(), loan.getMonthlyPayment(),
      loan.getTotalPayment());
  }
}
```

3 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.

4 Class Relationships

To design classes, you need to explore the relationships among classes.
The common relationships among classes are association, aggregation, composition, and inheritance.

4 Class Relationships

4.1 Association

Association is a general binary relationship that describes an activity between two classes.

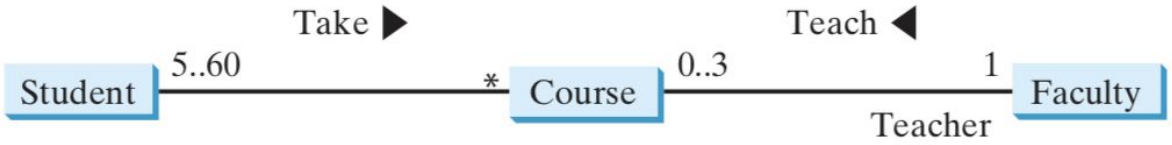
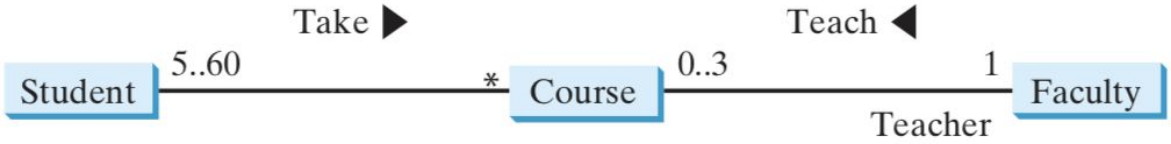


FIGURE 10.4 This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.

4 Class Relationships

4.1 Association

Association is a general binary relationship that describes an activity between two classes.




```
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.

4 Class Relationships

4.2 Aggregation and Composition

- Aggregation is a special form of association that represents an ownership relationship between two objects.
- Aggregation models has-a relationships.
- 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.

4 Class Relationships

4.2 Aggregation and Composition

- "a student has a name" is a composition relationship between the Student class and the Name class
- "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.

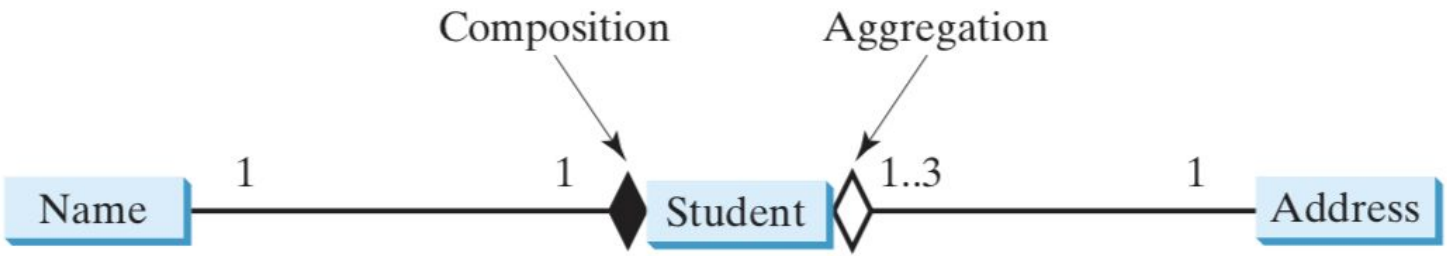
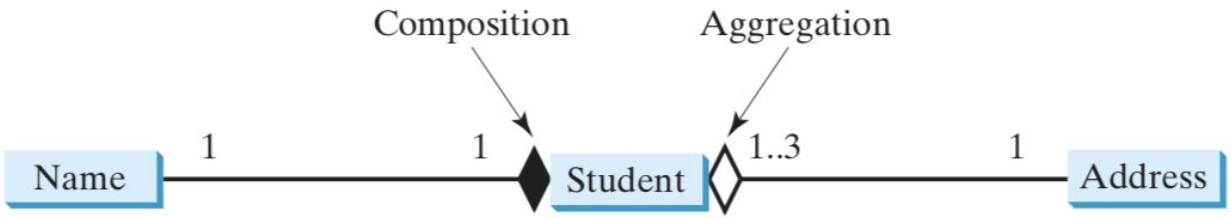


FIGURE 10.6 Each student has a name and an address.

4 Class Relationships

4.2 Aggregation and Composition




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

Aggregated class

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

Aggregating class

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

Aggregated class

4 Class Relationships

4.2 Aggregation and Composition

Aggregation may exist between objects of the same class.

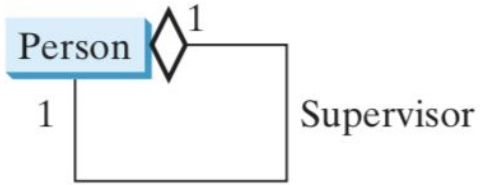


FIGURE 10.8 A person may have a supervisor.

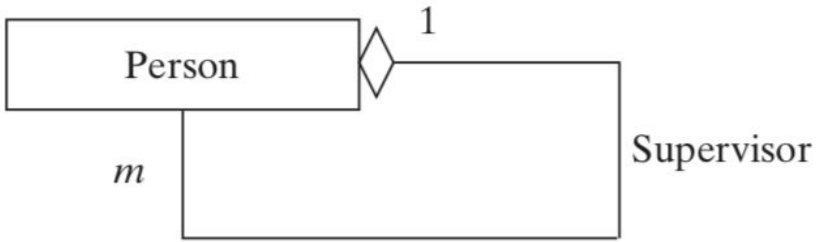
```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;

  ...
}
```

4 Class Relationships

4.2 Aggregation and Composition

Aggregation may exist between objects of the same class.



(a)

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

(b)

FIGURE 10.9 A person can have several supervisors.

5 Processing Primitive Data Type Values as Objects

- A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.
  - Primitive type : boolean, character, double, float, byte, short, integer, long
  - Wrapper class: Boolean, Character, Double, Float, Byte, Short, Integer, Long

5 Processing Primitive Data Type Values as Objects

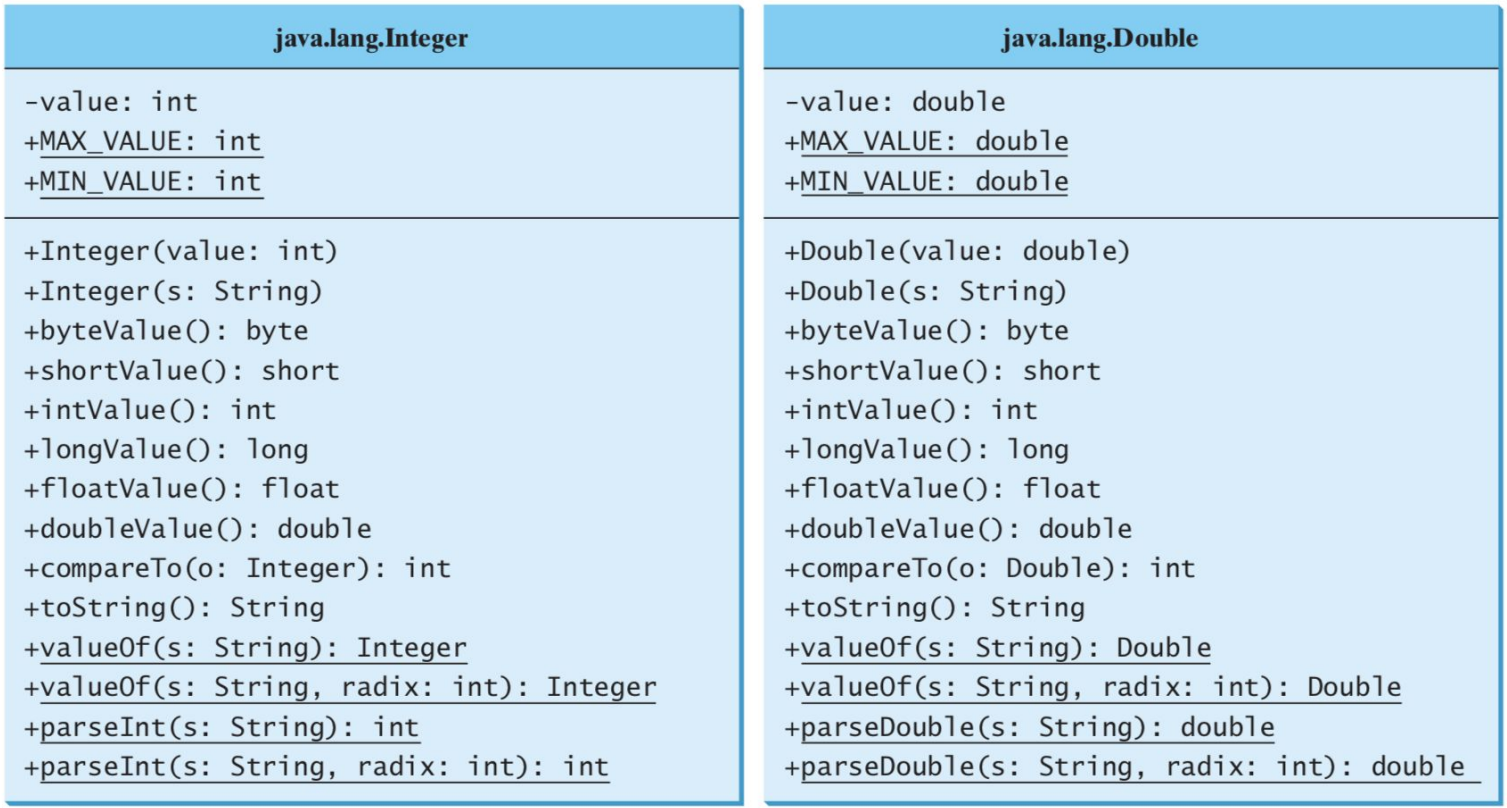


FIGURE 10.14 The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

5 Processing Primitive Data Type Values as Objects

System.out.println("The maximum integer is " + Integer.MAX_VALUE);

new Double(12.4).intValue() returns 12;

new Double(12.4).compareTo(new Double(12.3)) returns 1;

new Double(12.3).compareTo(new Double(12.3)) returns 0;

new Double(12.3).compareTo(new Double(12.51)) returns -1;

Double doubleObject = Double.valueOf("12.4");

Integer.parseInt("11", 2) returns 3;

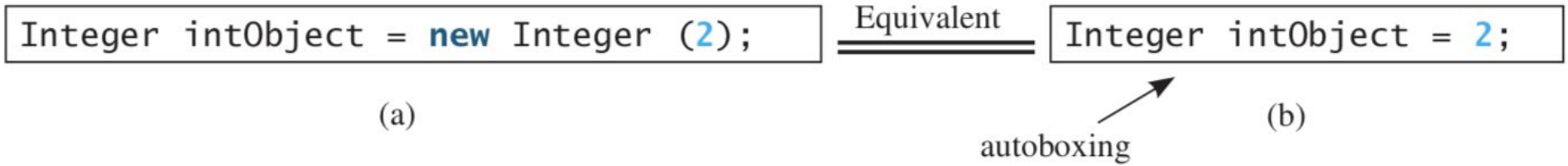
Integer.parseInt("12", 8) returns 10;

Integer.parseInt("13", 10) returns 13;

Integer.parseInt("1A", 16) returns 26;

6 Automatic Conversion between Primitive Types and Wrapper Class Types

- A primitive type value can be automatically converted to an object using a wrapper class (boxing) and vice versa, depending on the context.
- The reverse conversion is called unboxing.



7 The BigInteger and BigDecimal Classes

- 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.
- An instance of BigInteger can represent an integer of any size.

```
BigInteger a = new BigInteger("92233720368547758O7");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

8 The String Class

- Constructing a String

  String message = new String("Welcome to Java");
- A String object is immutable: Its content cannot be changed once the string is created.

  String s = "Java";

  s = "HTML";

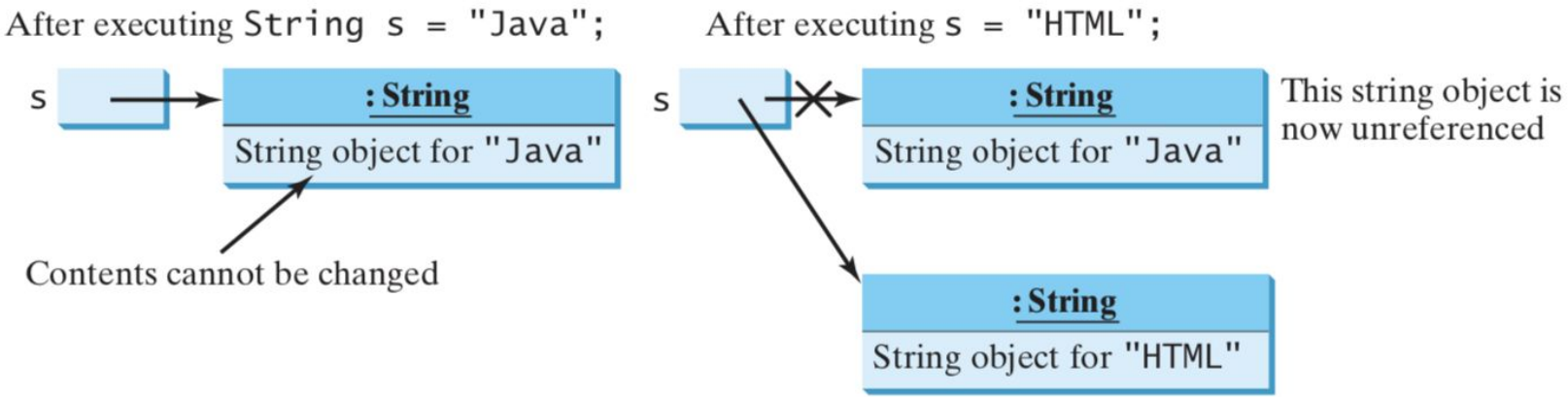


FIGURE 10.15 Strings are immutable; once created, their contents cannot be changed.

8 The String Class

- Interned string

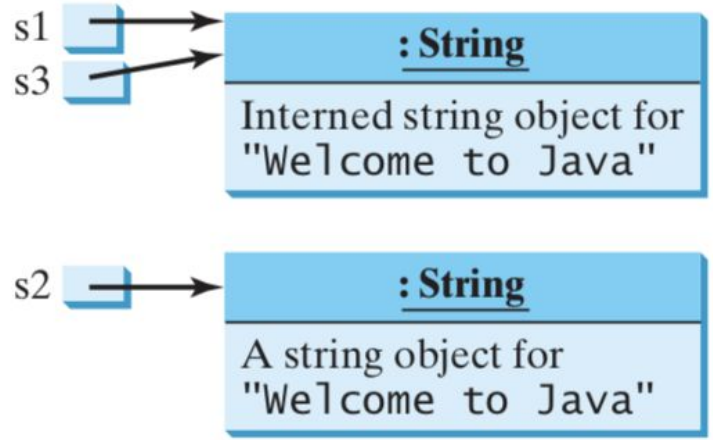
JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory.

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```



8 The String Class

Replacing and Splitting Strings

"Welcome".replace('e', 'A') returns a new string, WAlcomA.

"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.

"Welcome".replace("e", "AB") returns a new string, WABlcomAB.

"Welcome".replace("el", "AB") returns a new string, WABcome.

java.lang.String	
`+replace(oldChar: char, newChar: char): String`	Returns a new string that replaces all matching characters in this string with the new character.
`+replaceFirst(oldString: String, newString: String): String`	Returns a new string that replaces the first matching substring in this string with the new substring.
`+replaceAll(oldString: String, newString: String): String`	Returns a new string that replaces all matching substrings in this string with the new substring.
`+split(delimiter: String): String[]`	Returns an array of strings consisting of the substrings split by the delimiter.

FIGURE 10.16 The `String` class contains the methods for replacing and splitting strings.

8 The String Class

Matching, Replacing and Splitting by Patterns

- A regular expression (abbreviated regex) is a string that describes a pattern for matching a set of strings.
- You can match, replace, or split a string by specifying a pattern.

8 The String Class

Matching, Replacing and Splitting by Patterns

- "Java.*": It describes a string pattern that begins with Java followed by any zero or more characters

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

- \\d represents a single digit, and \\d{3} represents three digits

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

- returns a new string that replaces $, +, or # in a+b$#c with the string NNN

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

8 The String Class

Matching, Replacing and Splitting by Patterns

- Split the string into an array of strings delimited by punctuation marks

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");
for (int i = 0; i < tokens.length; i++)
    System.out.println(tokens[i]);
```

8 The String Class

Converting Characters and Numeric Values to Strings

String.valueOf(5.44) return "5.44"

java.lang.String	
+valueOf(c: char): String	Returns a string consisting of the character c.
+valueOf(data: char[]): String	Returns a string consisting of the characters in the array.
+valueOf(d: double): String	Returns a string representing the double value.
+valueOf(f: float): String	Returns a string representing the float value.
+valueOf(i: int): String	Returns a string representing the int value.
+valueOf(l: long): String	Returns a string representing the long value.
+valueOf(b: boolean): String	Returns a string representing the boolean value.

FIGURE 10.17 The `String` class contains the static methods for creating strings from primitive type values.

8 The String Class

Formatting Strings

String.format(format, item1, item2, ..., itemk)

This method is similar to the *printf* method except that the *format* method returns a formatted string, whereas the printf method displays a formatted string.

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

```
□□45.56□□□□14AB□□
```

9 The StringBuilder and StringBuffer Classes

- The StringBuilder and StringBuffer classes are similar to the String class except that the String class is immutable.
- The StringBuilder class is similar to StringBuffer except that the methods for modifying the buffer in StringBuffer are synchronized, which means that only one task is allowed to execute the methods.
- The constructors and methods in StringBuffer and StringBuilder are almost the same.

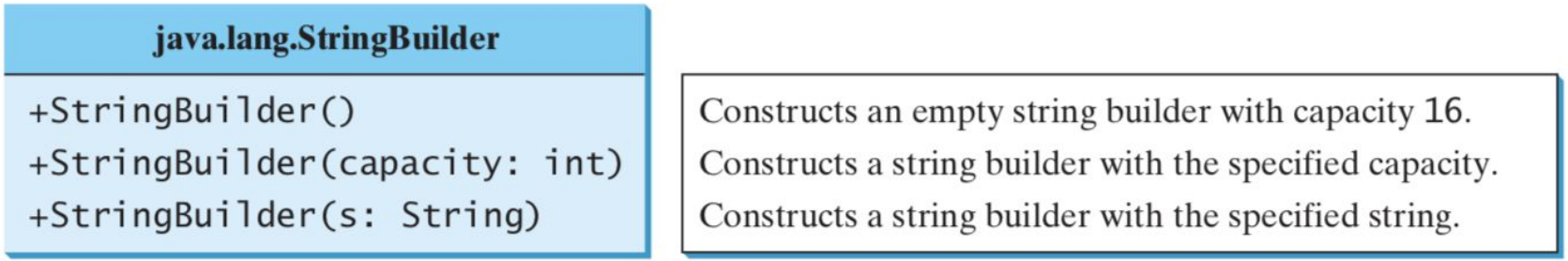


FIGURE 10.18 The `StringBuilder` class contains the constructors for creating instances of `StringBuilder`.

9 The StringBuilder and StringBuffer Classes

Modifying Strings in the StringBuilder

java.lang.StringBuilder	
`+append(data: char[]): StringBuilder`	Appends a `char` array into this string builder.
`+append(data: char[], offset: int, len: int): StringBuilder`	Appends a subarray in `data` into this string builder.
`+append(v: aPrimitiveType): StringBuilder`	Appends a primitive type value as a string to this builder.
`+append(s: String): StringBuilder`	Appends a string to this string builder.
`+delete(startIndex: int, endIndex: int): StringBuilder`	Deletes characters from `startIndex` to `endIndex-1`.
`+deleteCharAt(index: int): StringBuilder`	Deletes a character at the specified index.
`+insert(index: int, data: char[], offset: int, len: int): StringBuilder`	Inserts a subarray of the data in the array into the builder at the specified index.
`+insert(offset: int, data: char[]): StringBuilder`	Inserts data into this builder at the position offset.
`+insert(offset: int, b: aPrimitiveType): StringBuilder`	Inserts a value converted to a string into this builder.
`+insert(offset: int, s: String): StringBuilder`	Inserts a string into this builder at the position offset.
`+replace(startIndex: int, endIndex: int, s: String): StringBuilder`	Replaces the characters in this builder from `startIndex` to `endIndex-1` with the specified string.
`+reverse(): StringBuilder`	Reverses the characters in the builder.
`+setCharAt(index: int, ch: char): void`	Sets a new character at the specified index in this builder.

FIGURE 10.19 The `StringBuilder` class contains the methods for modifying string builders.

9 The StringBuilder and StringBuffer Classes

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java"); // Welcome to Java

stringBuilder.insert(11, "HTML and "); // Welcome to HTML and Java
```

stringBuilder.delete(8, 11) changes the builder to Welcome Java.
stringBuilder.deleteCharAt(8) changes the builder to Welcome o Java.
stringBuilder.reverse() changes the builder to avaJ ot emocleW.
stringBuilder.replace(11, 15, "HTML") changes the builder to Welcome to HTML.
stringBuilder.setCharAt(0, 'w') sets the builder to welcome to Java.

9 The StringBuilder and StringBuffer Classes

The toString, capacity, length, setLength, and charAt Methods

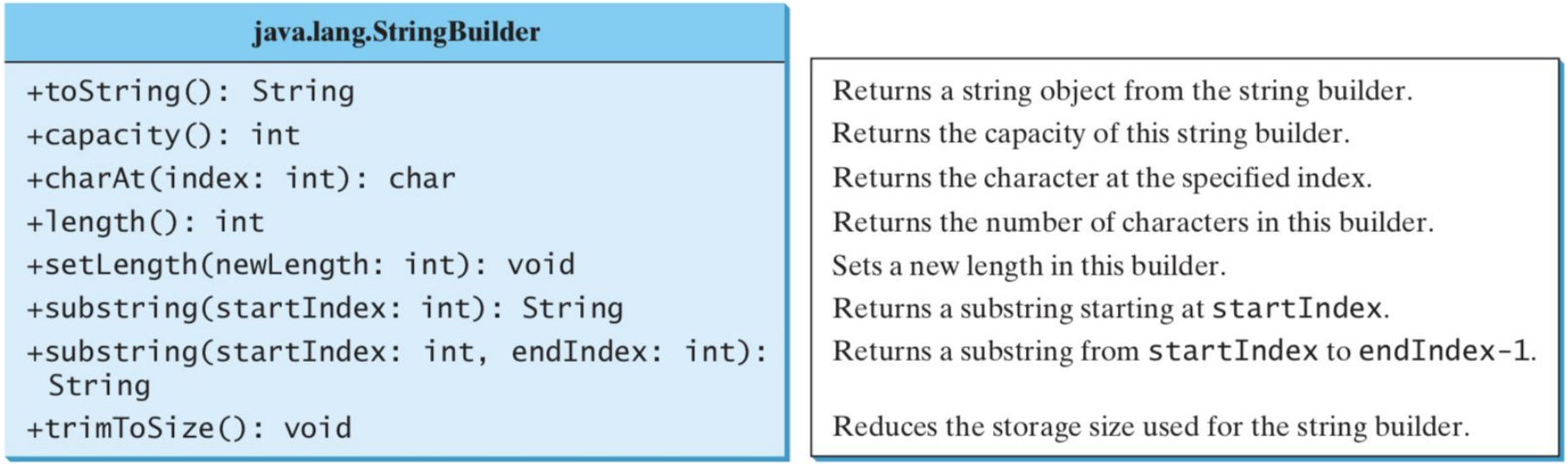


FIGURE 10.20 The `StringBuilder` class contains the methods for modifying string builders.