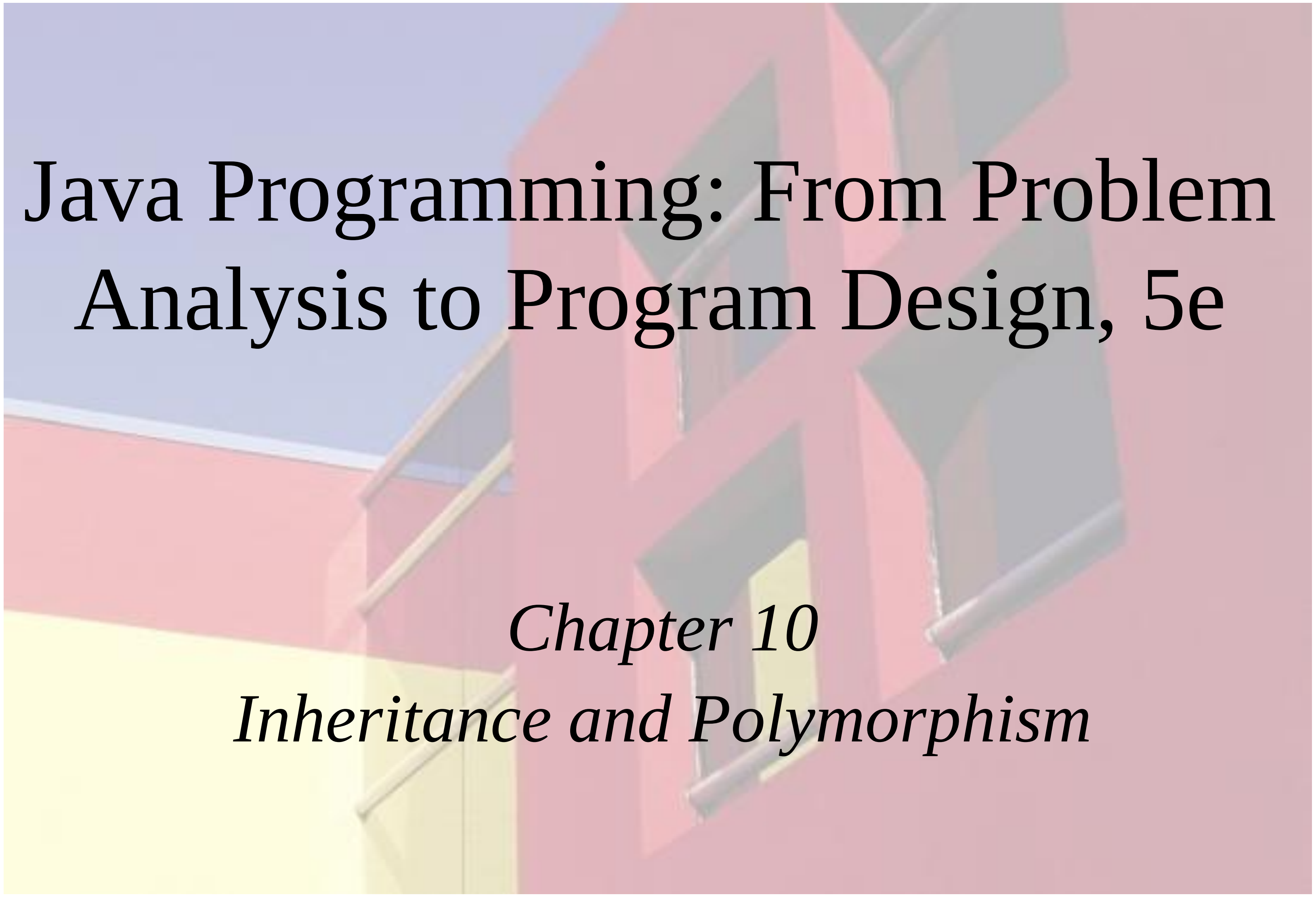


Java Programming: From Problem Analysis to Program Design, 5e

Chapter 10

Inheritance and Polymorphism

Chapter Objectives

- Learn about inheritance
- Learn about subclasses and superclasses
- Explore how to override the methods of a superclass
- Examine how constructors of superclasses and subclasses work

Chapter Objectives (continued)

- Learn about polymorphism
- Examine abstract classes
- Become aware of interfaces
- Learn about composition

Inheritance

- "is-a" relationship
- Single inheritance
  - Subclass is derived from one existing class (superclass)
- Multiple inheritance
  - Subclass is derived from more than one superclass
  - Not supported by Java
  - In Java, a class can only extend the definition of one class

Inheritance (continued)

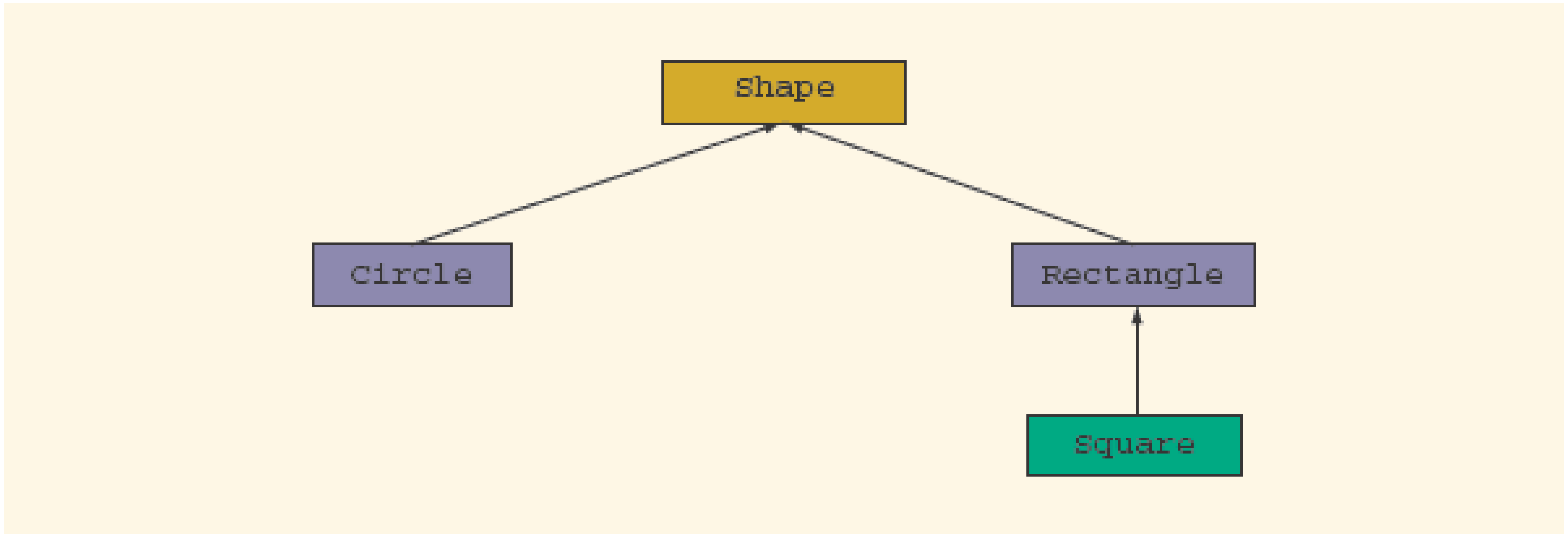


FIGURE 10-1 Inheritance hierarchy

```
modifier(s) class ClassName extends ExistingClassName
                                           modifier(s)
{
    memberList
}
```

Inheritance: `class Circle` Derived from `class Shape`

```
public class Circle extends Shape
{
          .
          .
          .
}
```

Inheritance Rules

1. The `private` members of the superclass are private to the superclass
2. The subclass can directly access the `public` members of the superclass
3. The subclass can include additional data and/or method members

Inheritance Rules (continued)

4. The subclass can override, that is, redefine the `public` methods of the superclass
   - However, this redefinition applies only to the objects of the subclass, not to the objects of the superclass
5. All data members of the superclass are also data members of the subclass
   - Similarly, the methods of the superclass (unless overridden) are also the methods of the subclass
   - Remember Rule 1 when accessing a member of the superclass in the subclass

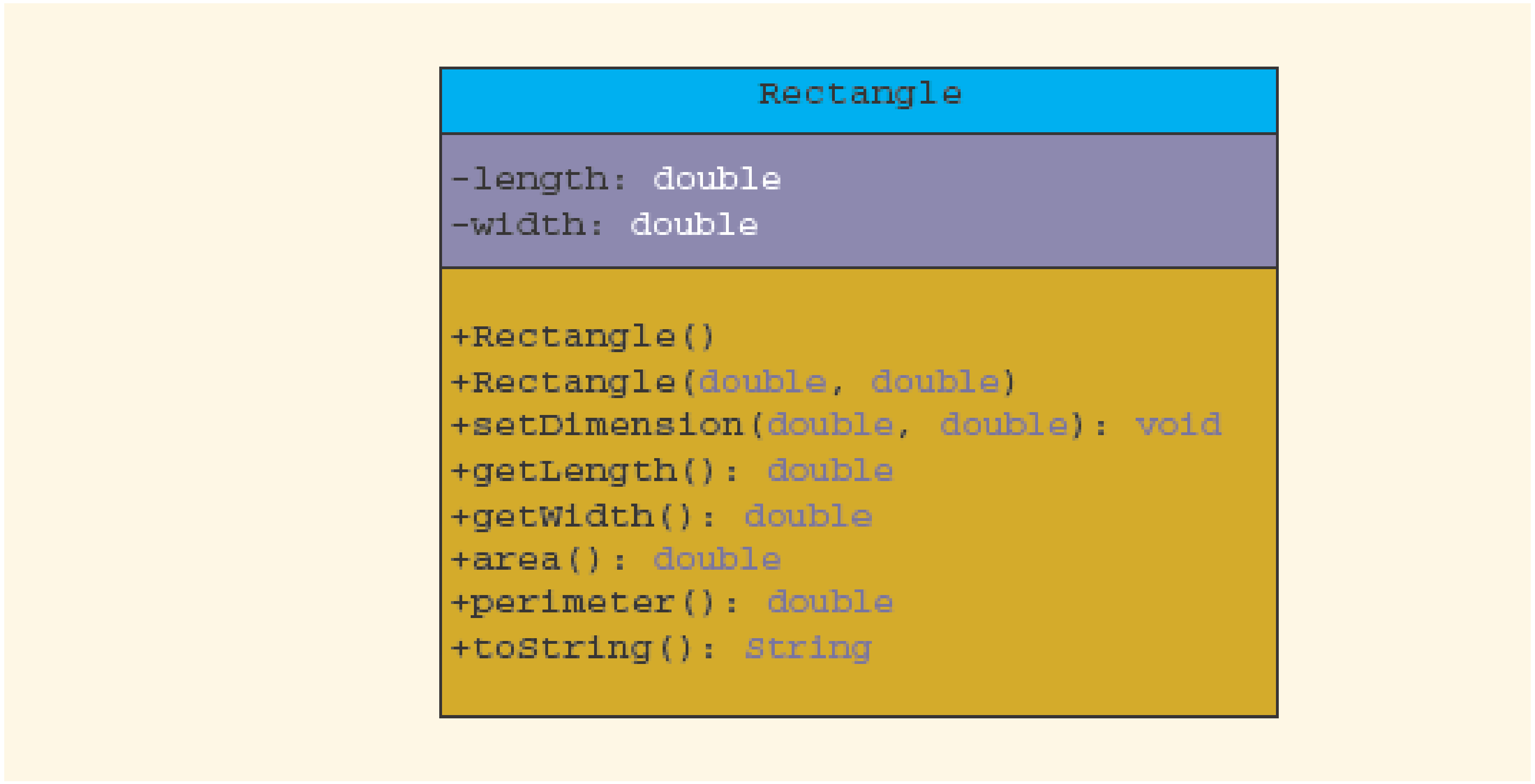
Inheritance (continued)

- To write a method's definition of a subclass, specify a call to the public method of the superclass
  - If subclass overrides `public` method of superclass, specify call to `public` method of superclass:

    `super.MethodName(parameter list)`
  - If subclass does not override `public` method of superclass, specify call to `public` method of superclass:

    `MethodName(parameter list)`

UML Class Diagram: `class Rectangle`



```
Rectangle
-length: double
-width: double
+Rectangle()
+Rectangle(double, double)
+setDimension(double, double): void
+getLength(): double
+getWidth(): double
+area(): double
+perimeter(): double
+toString(): String
```

FIGURE 10-2 UML class diagram of the `class Rectangle`

UML Class Diagram: class Box

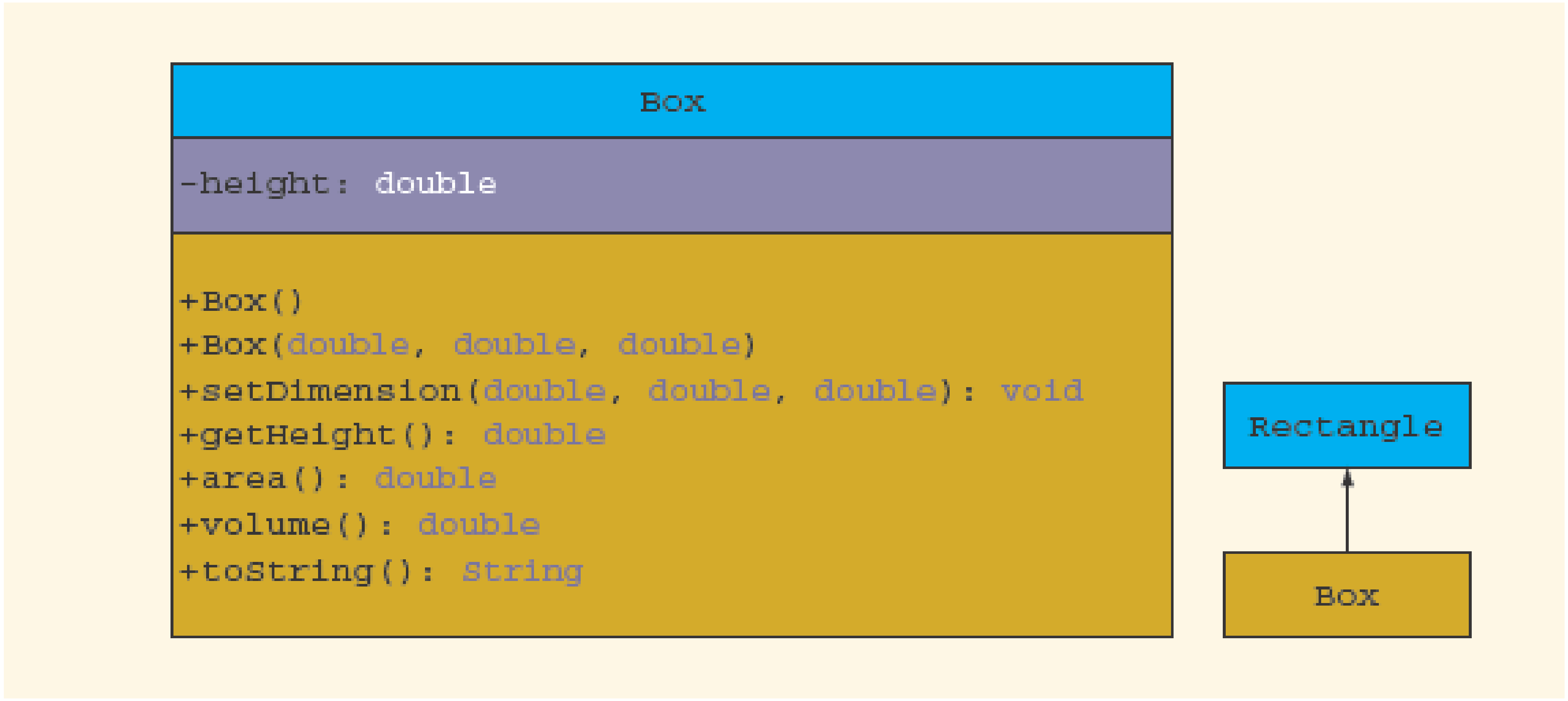


FIGURE 10-3 UML class diagram of the class Box and the inheritance hierarchy

class Box

```
public String toString()
{
    return super.toString()    //retrieve length and width
           + "; Height = " + height;
}
public void setDimension(double l, double w, double h)
{
    super.setDimension(l, w);
    if (h >= 0)
        height = h;
    else
        height = 0;
}
public double area()
{
    return  2 * (getLength() * getWidth()
                 + getLength() * height
                 + getWidth() * height);
}
```

Defining Constructors of the Subclass

- Call to constructor of superclass
  - Must be first statement
  - Specified by: super parameter list

```
public Box()
{
    super();
    height = 0;
}

public Box(double l, double w, double h)
{
    super(l, w);
    height = h;
}
```

Objects `myRectangle` and `myBox`

```
Rectangle myRectangle = new Rectangle(5, 3);
Box myBox = new Box(6, 5, 4);
```

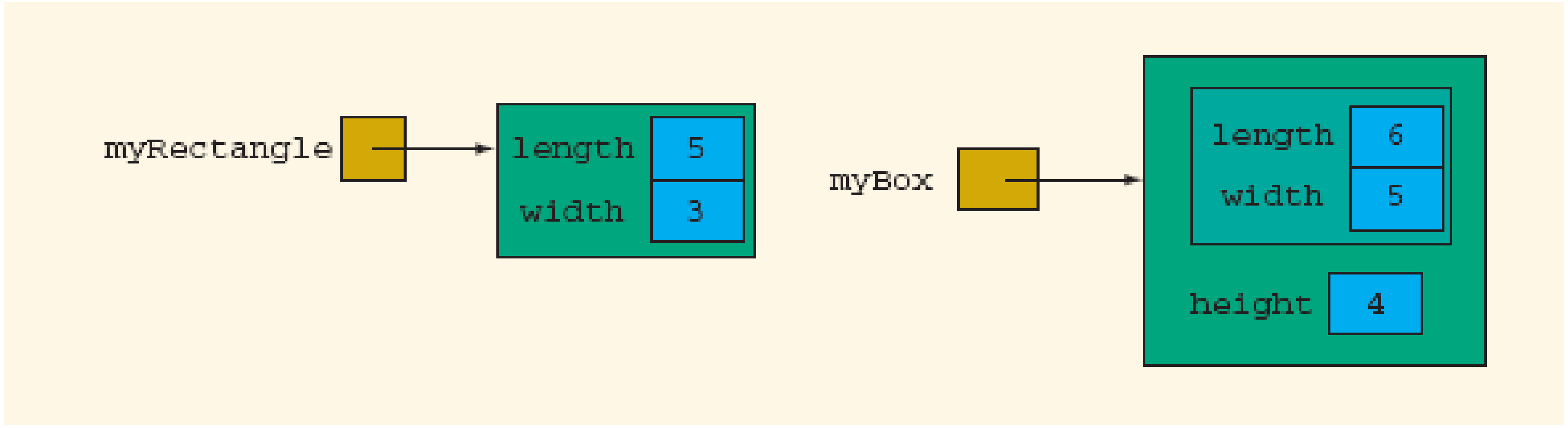


FIGURE 10-4 Objects myRectangle and myBox

- **A call to a constructor of a superclass is specified in the definition of a constructor of the subclass**
- **When a subclass constructor executes, first a constructor of the superclass executes to initialize the data members inherited from the superclass and then the constructor of the subclass executes to initialize the data members declared by the subclass**
- **First the constructor of the `class Rectangle` executes to initialize the instance variables `length` and `width` and then the constructor of the `class Box` executes to initialize the instance variable `height`**

Shadowing Variables

- Suppose that the `class` `SubClass` is derived from the `class` `SuperClass` and `SuperClass` has a variable named `temp`
- You can declare a variable named `temp` in the `class` `SubClass`; in this case, the variable `temp` of `SubClass` is called a shadowing variable
- The concept of a shadowing variable is similar to the concept of overriding a method, but it causes confusion
- The `SubClass` is derived from `SuperClass`, so it inherits the variable `temp` of `SuperClass`
- Because a variable named `temp` is already available in `SubClass`, there is seldom if ever any reason to override it
- It is poor programming practice to override a variable in the `SubClass`

Shadowing Variables (continued)

- Anyone reading code with a shadowed variable will have two different declarations of a variable seeming to apply to the shadowed variable of the `SubClass`
- This causes confusion and should be avoided; in general, you should avoid shadowing variables

UML Class Diagram

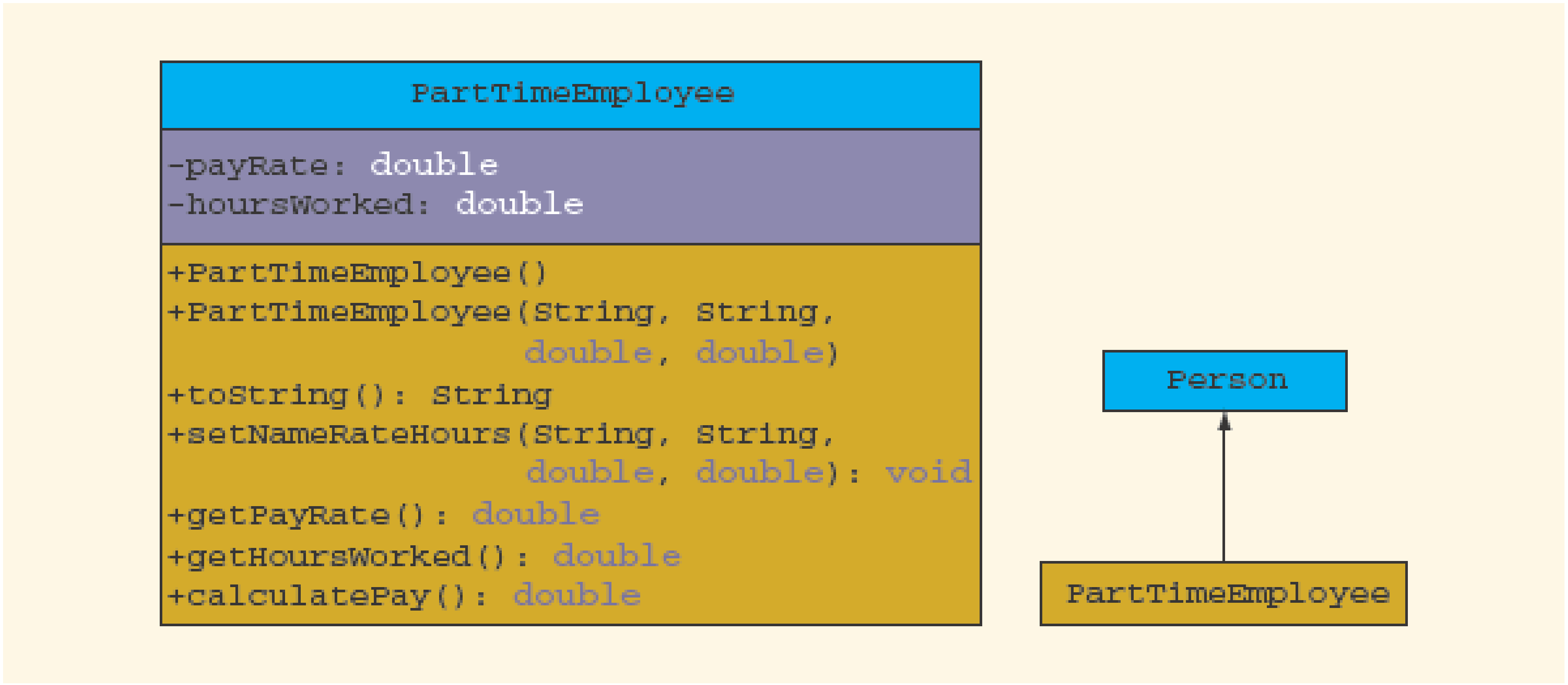


FIGURE 10-5 UML class diagram of the **class** PartTimeEmployee and the inheritance hierarchy

Protected Members of a Class

- The `private` members of a class are `private` to the class and cannot be directly accessed outside the class
- Only methods of that class can access the `private` members directly; as discussed previously, the subclass cannot access the `private` members of the superclass directly
- If you make a `private` member `public`, then anyone can access that member
- If a member of a superclass needs to be (directly) accessed in a subclass and yet still prevent its direct access outside the class, you must declare that member using the modifier `protected`
- The accessibility of a `protected` member of a class falls between `public` and `private`
- A subclass can directly access the `protected` member of a superclass

Protected Access vs. Package Access

- Typically a member of a class is declared with the modifier public, private, or protected to give appropriate access to that member
- You can also declare a member without any of these modifiers
- If a class member is declared without any of the modifiers public, private, or public, then the Java system gives to that member the default package access; that is, that member can be directly accessed in any class contained in that package
- There is a subtle difference between package access and protected accesses of a member

Protected Access vs. Package Access (continued)

- If a member of a class has a package access, that member can be directly accessed in any class contained in that package, but not in any class that is not contained in that package even if a subclass is derived from the class containing that member and the subclass is not contained in that package
- On the other hand, if a member of a class is protected, that member can be directly accessed in any subclass even if the subclass is contained in a different package

The `class Object`

- Directly or indirectly becomes the superclass of every class in Java
- `public` members of `class Object` can be overridden/invoked by object of any class type

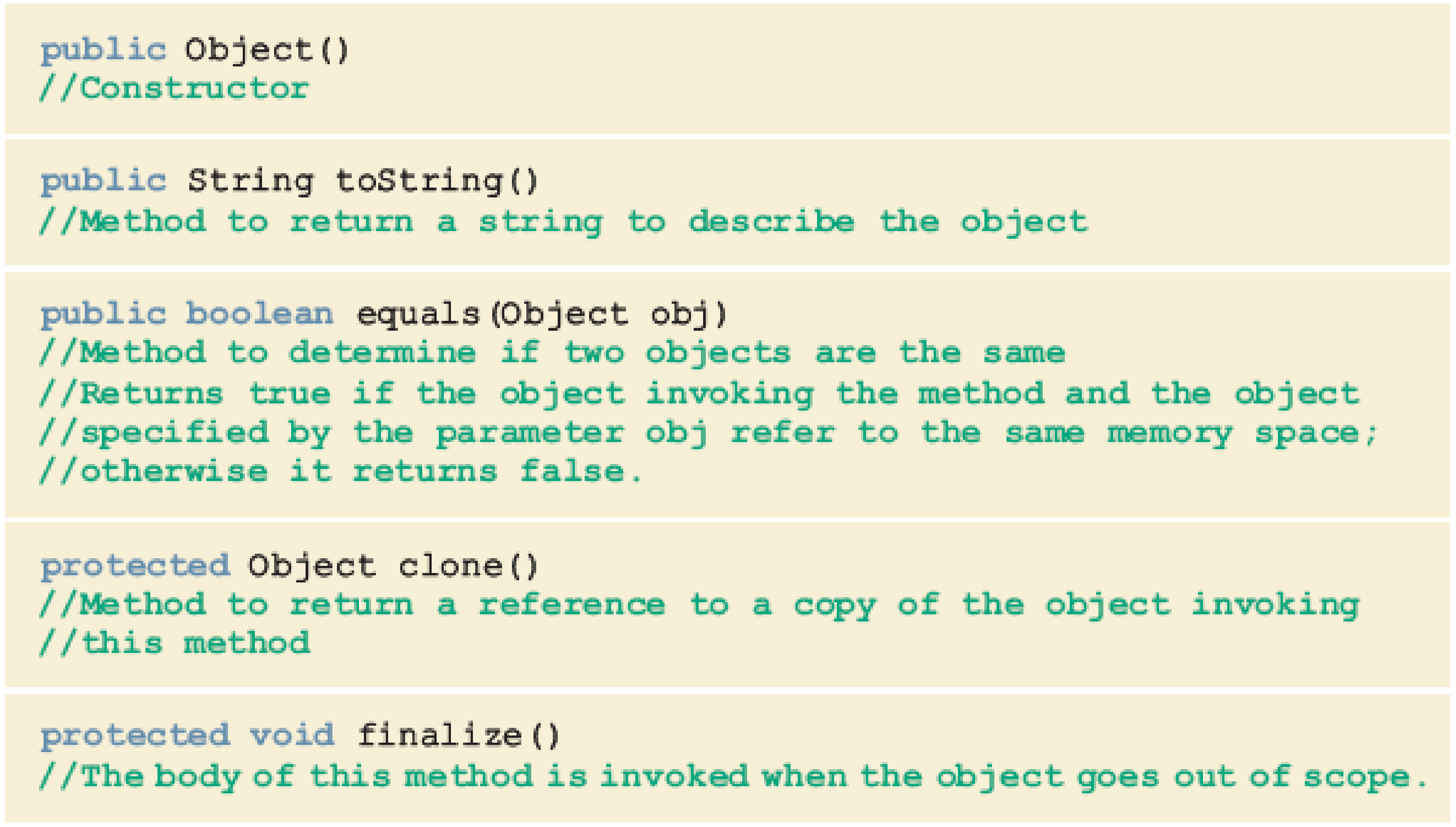
The `class Object`: Equivalent Definition of a Class

```
public class Clock
{
    //Declare instance variables as given in Chapter 8
   //Definition of instance methods as given in Chapter 8
   //...
}

public class Clock extends Object
{
  //Declare instance variables as given in Chapter 8
   //Definition of instance methods as given in Chapter 8
  //...
}
```

Some Constructors and Methods of the `class Object`

TABLE 10-1 Constructors and Methods of the `class Object`



```
public Object()
//Constructor
```

```
public String toString()
//Method to return a string to describe the object
```

```
public boolean equals(Object obj)
//Method to determine if two objects are the same
//Returns true if the object invoking the method and the object
//specified by the parameter obj refer to the same memory space;
//otherwise it returns false.
```

```
protected Object clone()
//Method to return a reference to a copy of the object invoking
//this method
```

```
protected void finalize()
//The body of this method is invoked when the object goes out of scope.
```

Hierarchy of Java Stream Classes

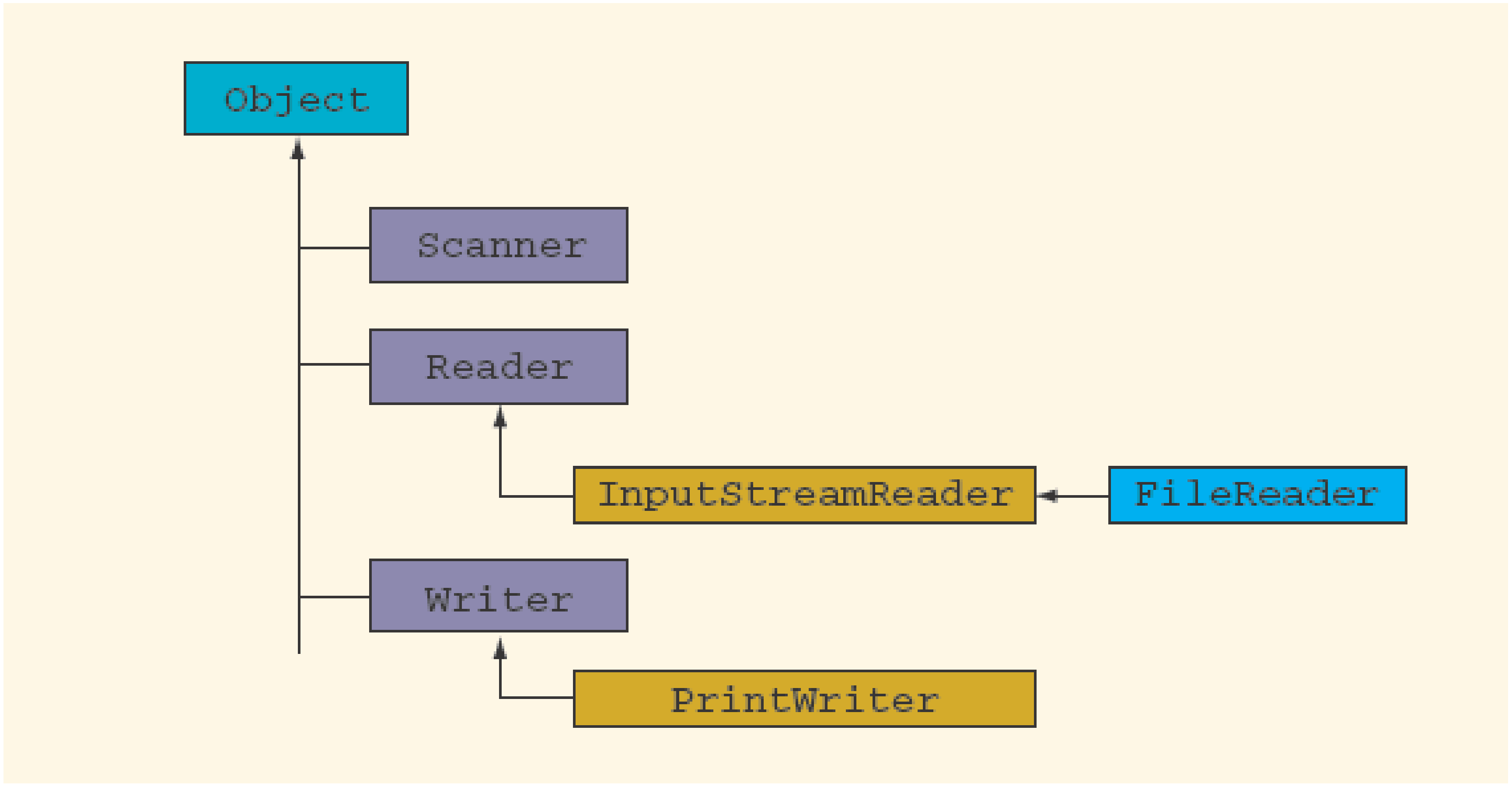


FIGURE 10-6 Java stream classes hierarchy

Polymorphism

- Java allows us to treat an object of a subclass as an object of its superclass
  - In other words, a reference variable of a superclass type can point to an object of its subclass

```
Person name, nameRef;
PartTimeEmployee employee, employeeRef;
name = new Person("John", "Blair");
employee = new PartTimeEmployee("Susan", "Johnson",
                                12.50, 45);
nameRef = employee;
System.out.println("nameRef: " + nameRef);
```

```
nameRef: Susan Johnson wages are: $562.5
```

Polymorphism (continued)

- Late binding or dynamic binding (run-time binding)
  - Method executed determined at execution time, not compile time
- The term polymorphism means assigning multiple meanings to the same method name
- In Java, polymorphism is implemented using late binding
- The reference variable `name` or `nameRef` can point to any object of the `class Person` or the `class PartTimeEmployee`

Polymorphism (continued)

- These reference variables have many forms; that is, they are polymorphic reference variables
  - They can refer to objects of their own class or objects of the classes inherited from their class
- You can declare a method of a class final using the key word `final`; for example, the following method is final:

```
public final void doSomeThing()
{
      //...
}
```

Polymorphism (continued)

- If a method of a `class` is declared `final`, it cannot be overridden with a new definition in a derived class
- In a similar manner, you can also declare a `class` final using the keyword `final`
- If a `class` is declared `final`, then no other class can be derived from this class
- Java does not use late binding for methods that are `private`, marked `final`, or `static`

Polymorphism (continued)

- You cannot automatically make reference variable of subclass type point to object of its superclass
- Suppose that `supRef` is a reference variable of a superclass type; moreover, suppose that `supRef` points to an object of its subclass
- You can use an appropriate cast operator on `supRef` and make a reference variable of the subclass point to the object

Polymorphism (continued)

- On the other hand, if `supRef` does not point to a subclass object and you use a cast operator on `supRef` to make a reference variable of the subclass point to the object, then Java will throw a `ClassCastException`, indicating that the class cast is not allowed

Polymorphism (continued)

- Operator `instanceof`: determines whether reference variable that points to object is of particular class type

```
p instanceof BoxShape
```

- This expression evaluates to `true` if `p` points to an object of the `class BoxShape`; otherwise it evaluates to `false`

Abstract Methods

- Abstract method: method that has only the heading with no body
  - Must be declared abstract

```
public void abstract print();
public abstract object larger(object,
                              object);
void abstract insert(int insertItem);
```

Abstract Classes

- Abstract class: class that is declared with the reserved word `abstract` in its heading
  - An abstract class can contain instance variables, constructors, finalizer, and nonabstract methods
  - An abstract class can contain abstract method(s)
  - If a class contains an abstract method, then the class must be declared abstract
  - You cannot instantiate an object of an abstract class type; you can only declare a reference variable of an abstract class type
  - You can instantiate an object of a subclass of an abstract class, but only if the subclass gives the definitions of *all* the abstract methods of the superclass

Abstract Class Example

```
public abstract class AbstractClassExample
{
    protected int x;

    public void abstract print();

    public void setX(int a)
    {
         x = a;
    }

    public AbstractClassExample()
    {
        x = 0;
    }
}
```

Interfaces

- Definition: class that contains only abstract methods and/or named constants
- How Java implements multiple inheritance
- To be able to handle a variety of events, Java allows a class to implement more than one interface

Some Interface Definitions

```
public interface WindowListener
{
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}

public interface ActionListener
{
    public void actionPerformed(ActionEvent e);
}
```

Polymorphism via Interfaces

- An interface can be used in the implementation of abstract data types
- Define an interface that contains the method's headings and/or named constants
- You can define the class that implements the interface
- The user can look at the interface and see what operations are implemented by the class
- Just as you can create polymorphic references to classes in an inheritance hierarchy, you can also create polymorphic references using interfaces
- You can use an interface name as the type of a reference variable, and the reference variable can point to any object of any class that implements the interface
- Because an interface contains only method headings and/or named constants, *you cannot create an object of an interface*

Composition (Aggregation)

- Another way to relate two classes
- One or more members of a class are objects of another class type
- “has-a” relation between classes
  - e.g., “every person has a date of birth”

Composition (Aggregation) Example

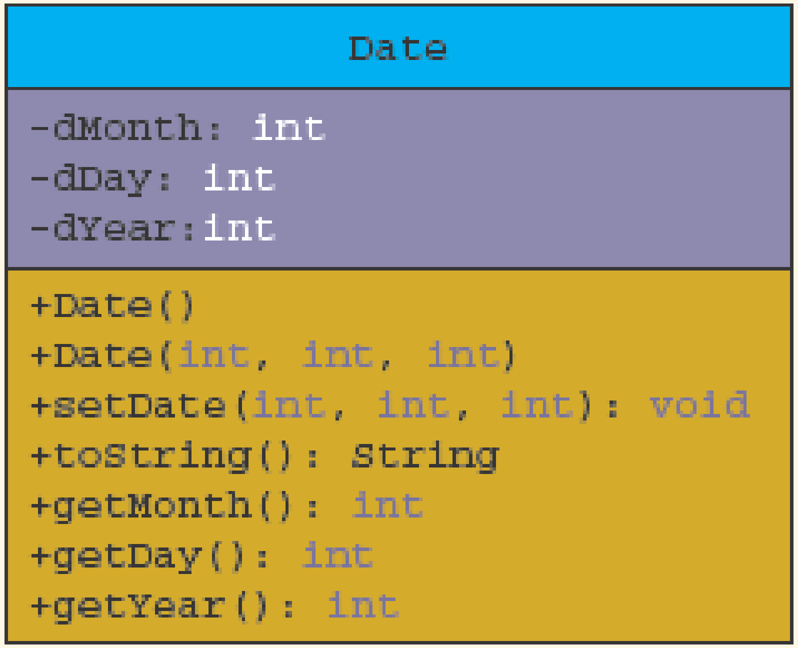


FIGURE 10-16 UML class diagram of the `class` Date

Composition (Aggregation) Example (continued)

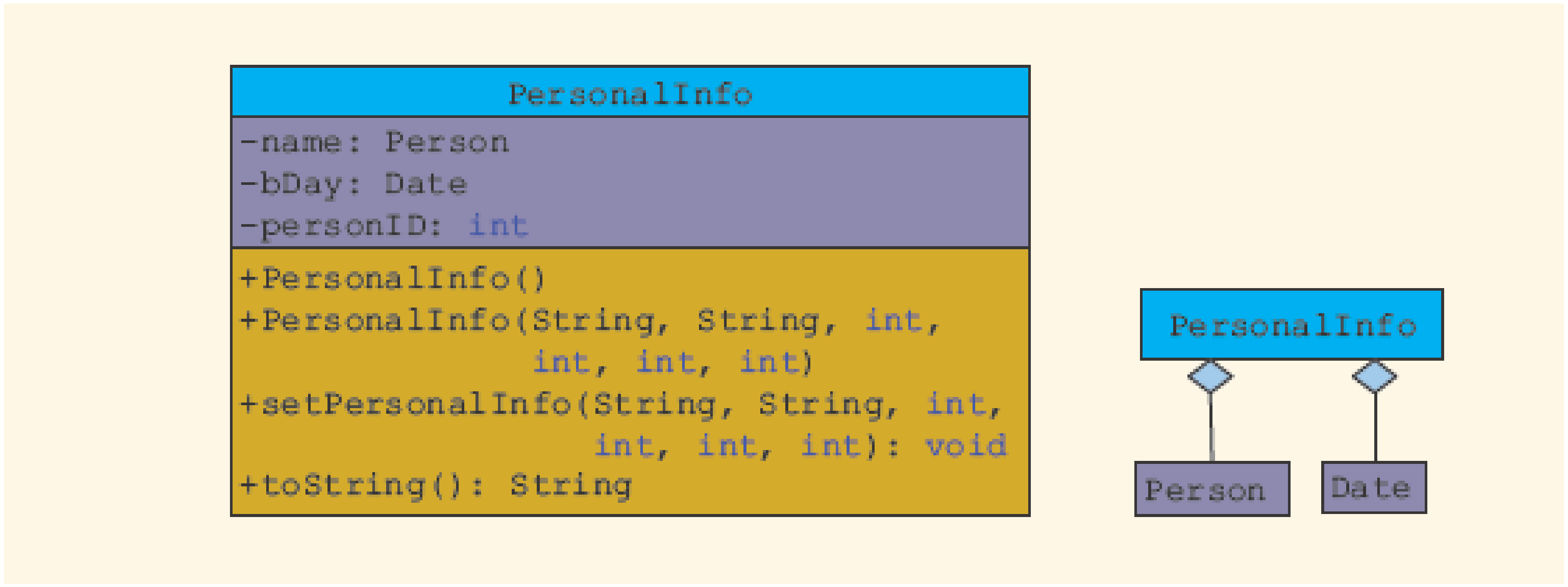

PersonalInfo
-name: Person
-bDay: Date
-personID: int
+PersonalInfo()
+PersonalInfo(String, String, int, int, int, int)
+setPersonalInfo(String, String, int, int, int, int): void
+toString(): String
PersonalInfo
Person
Date

FIGURE 10-17 UML class diagram of the **class** PersonalInfo

Programming Example: Grade Report

- Components: student, course
- Operations on course
  - Set course information
  - Print course information
  - Show credit hours
  - Show course number

Components Course and Student

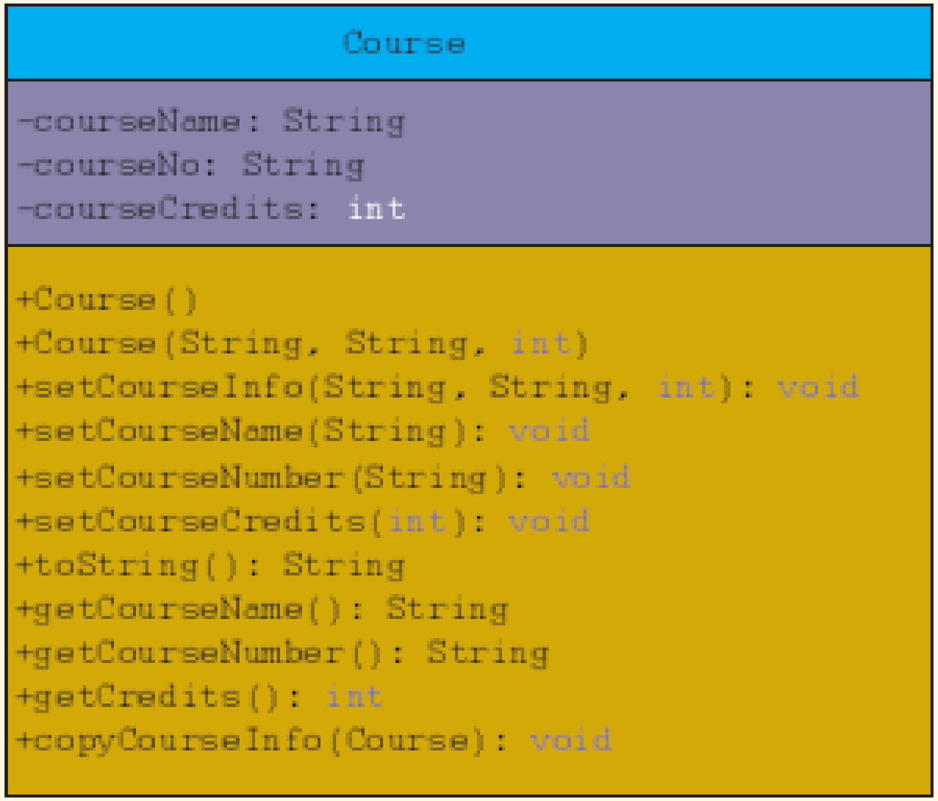
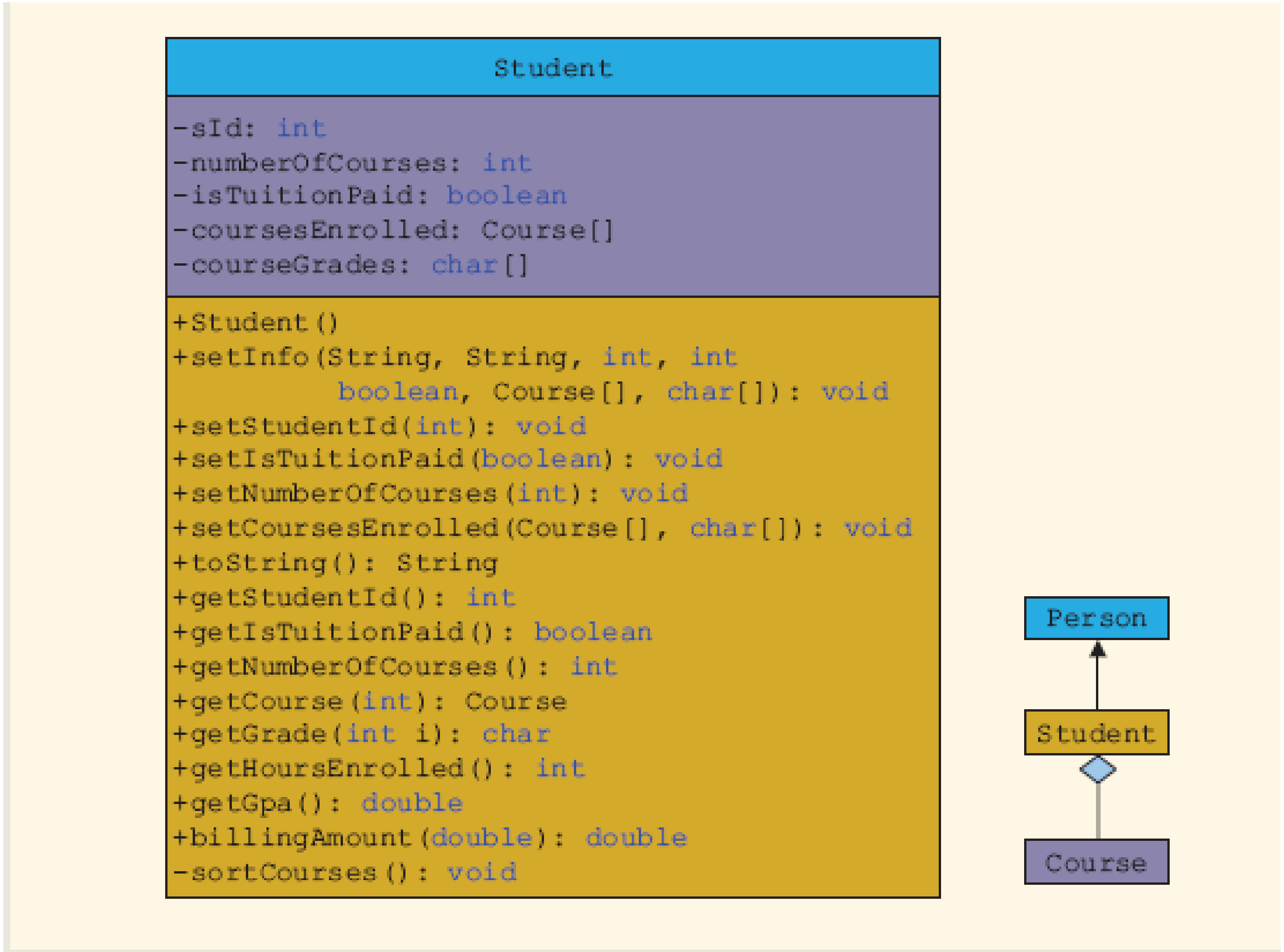

Course
-courseName: String
-courseNo: String
-courseCredits: int
+Course()
+Course(String, String, int)
+setCourseInfo(String, String, int): void
+setCourseName(String): void
+setCourseNumber(String): void
+setCourseCredits(int): void
+toString(): String
+getCourseName(): String
+getCourseNumber(): String
+getCredits(): int
+copyCourseInfo(Course): void

FIGURE 10-18 UML class diagram of the class Course

Components Course and Student (continued)



Student
-sId: int
-numberOfCourses: int
-isTuitionPaid: boolean
-coursesEnrolled: Course[]
-courseGrades: char[]
+Student()
+setInfo(String, String, int, int
boolean, Course[], char[]): void
+setStudentId(int): void
+setIsTuitionPaid(boolean): void
+setNumberOfCourses(int): void
+setCoursesEnrolled(Course[], char[]): void
+toString(): String
+getStudentId(): int
+getIsTuitionPaid(): boolean
+getNumberOfCourses(): int
+getCourse(int): Course
+getGrade(int i): char
+getHoursEnrolled(): int
+getGpa(): double
+billingAmount(double): double
-sortCourses(): void
Person
Student
Course

FIGURE 10-19 UML class diagram of the class Student

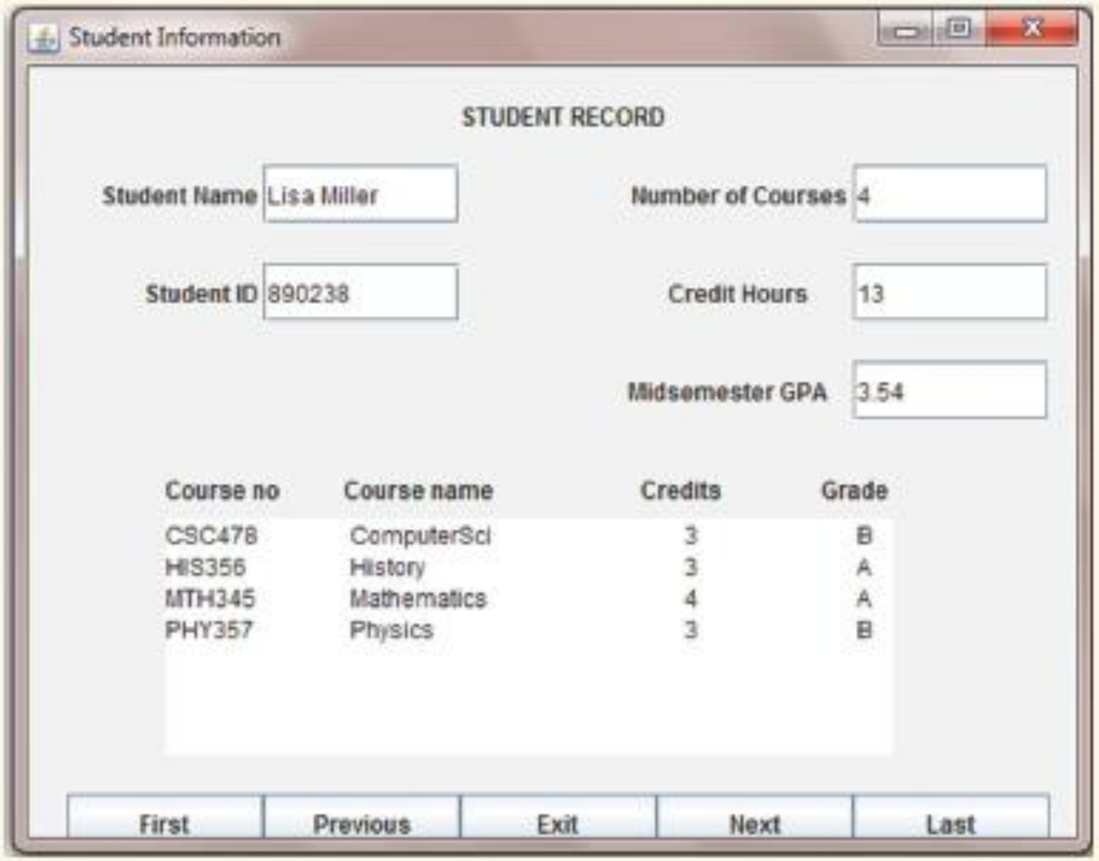
Programming Example: Grade Report

- Operations on student
  - Set student information
  - Print student information
  - Calculate number of credit hours taken
  - Calculate GPA
  - Calculate billing amount
  - Sort the courses according to the course number

Programming Example: Grade Report (continued)

- Main algorithm
  - Declare variables
  - Open input file
  - Open output file
  - Get number of students registered and tuition rate
  - Load students’ data
  - Print grade reports

Sample Output: Grade Report Program



Student Name	Lisa Miller	Number of Courses	4
Student ID	890238	Credit Hours	13
		Midsemester GPA	3.54

```
Course no     Course name     Credits     Grade
CSC478        ComputerSci        3          B
HIS356        History            3          A
MTH345        Mathematics        4          A
PHY357        Physics            3          B
```

FIGURE 10-20 GUI to show a student's record

Chapter Summary

- Inheritance
  - Single and multiple
  - Rules
  - Uses
  - Superclasses/subclasses (objects)
  - Overriding/overloading methods
  - Constructors
- The `class Object`

Chapter Summary (continued)

- Java stream classes
- Polymorphism
- Abstract methods
- Abstract classes
- Interfaces
- Composition