Inheritance and Polymorphism

https://www.tutorialspoint.com/compile_java_online.php

Go to the Execute tab to run online

Review: Classes

- User-defined data types
  - Defined using the “class” keyword
  - Each class has associated
    - Data members (any object type)
    - Methods that operate on the data
- New instances of the class are declared using the “new” keyword
- “Static” members/methods have only one copy, regardless of how many instances are created

Example: Shared Functionality

```
public class Student {
   String name;
   char gender;
   Date birthday;
   Vector<Grade> grades;

   double getGPA() {
      …
   }

   int getAge(Date today) {
      …
   }
}
```

```
public class Professor {
   String name;
   char gender;
   Date birthday;
   Vector<Paper> papers;

   int getCiteCount() {
      …
   }

   int getAge(Date today) {
      …
   }
}
```

```
public class Person {
   String name;
   char gender;
   Date birthday;

   int getAge(Date today) {
      …
   }
}
```

```
public class Student
      extends Person {

   Vector<Grade> grades;

   double getGPA() {
      …
   }
}
```

```
public class Professor
      extends Person {

   Vector<Paper> papers;

   int getCiteCount() {
      …
   }
}
```

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
  - A class can only extend the definition of one class

Inheritance (continued)

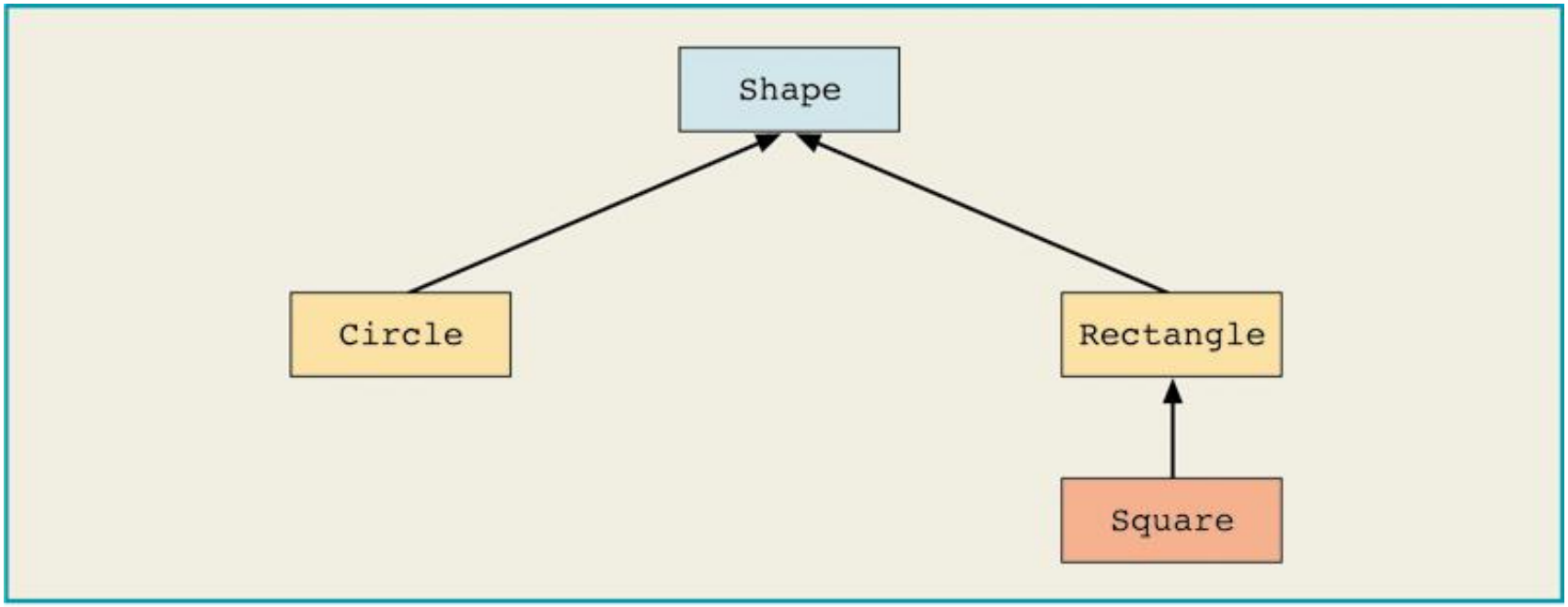

Figure 11-1 Inheritance hierarchy

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

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

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

Inheritance

- Allow us to specify *relationships between types*
  - Abstraction, generalization, specification
  - The “is-a” relationship
  - Examples?
- Why is this useful in programming?
  - Allows for code reuse
  - More intuitive/expressive code

Code Reuse

- General functionality can be written once and applied to **any** subclass
- Subclasses can specialize by adding members and methods, or overriding functions

Inheritance: Adding Functionality

- Subclasses have ***all*** of the data members and methods of the superclass
- Subclasses can add to the superclass
  - Additional data members
  - Additional methods
- Subclasses are more specific and have more functionality
- Superclasses capture generic functionality common across many types of objects

```
public class Person {
   String name;
   char gender;
   Date birthday;

   int getAge(Date today) {
      …
   }
}
```

```
public class Student
      extends Person {

   Vector<Grade> grades;

   double getGPA() {
      …
   }
}
```

```
public class Professor
      extends Person {

   Vector<Paper> papers;

   int getCiteCount() {
      …
   }
}
```

Brainstorming

- What are some other examples of possible inheritance hierarchies?
  - Person -> student, faculty…
  - Shape -> circle, triangle, rectangle…
  - Other examples???

UML Diagram: `Rectangle`

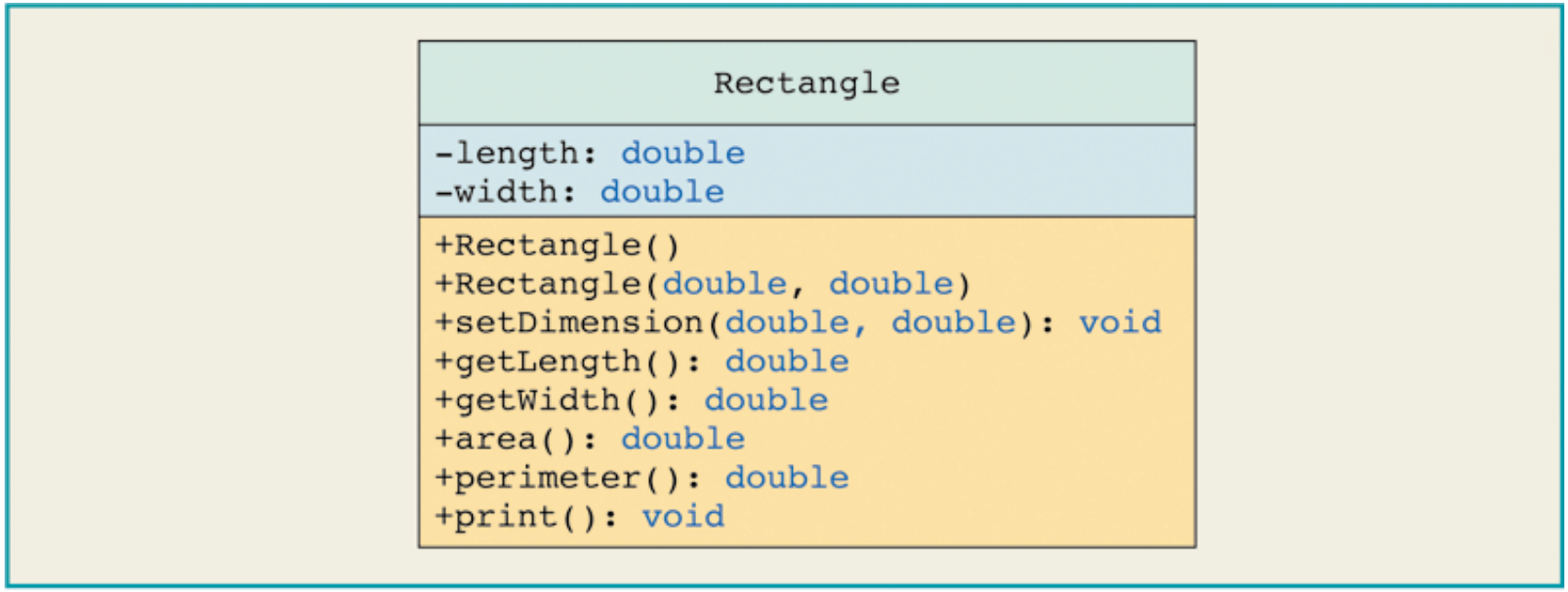

Figure 11-2 UML class diagram of the `class Rectangle`

What if we want to implement a 3d box object?

Objects `myRectangle` and `myBox`

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

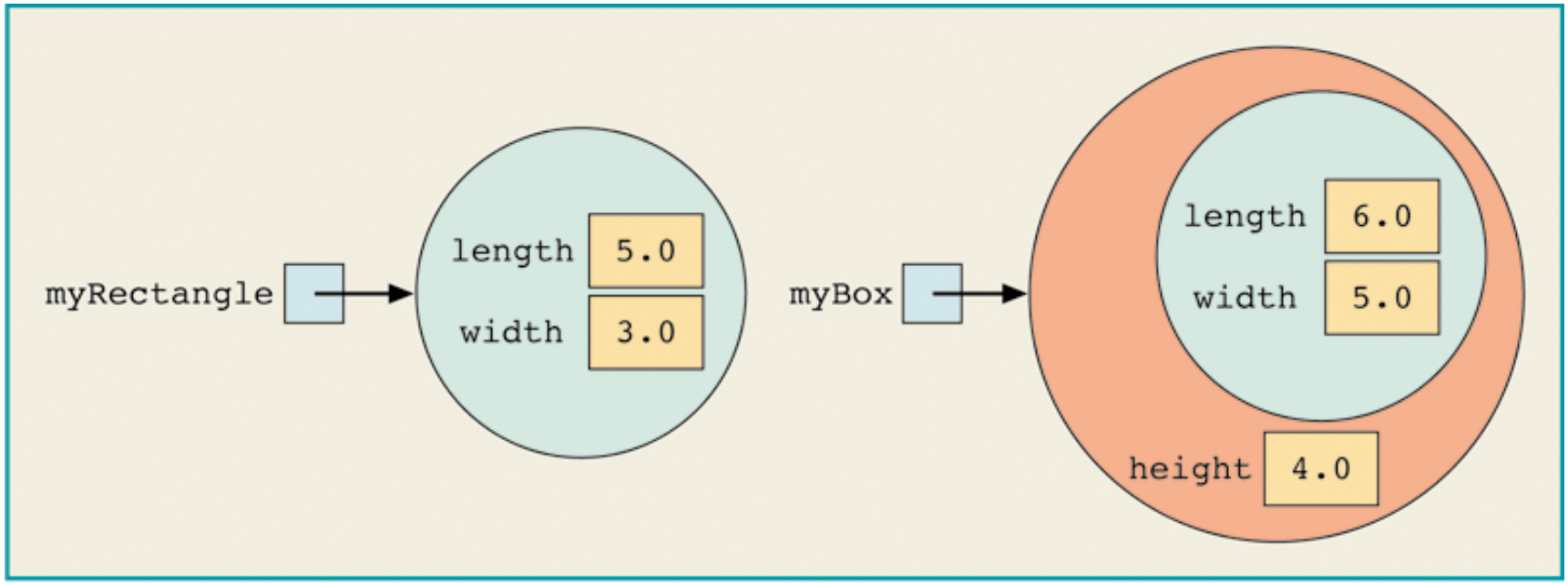

Figure 11-4 Objects `myRectangle` and `myBox`

UML Class Diagram: `class Box`

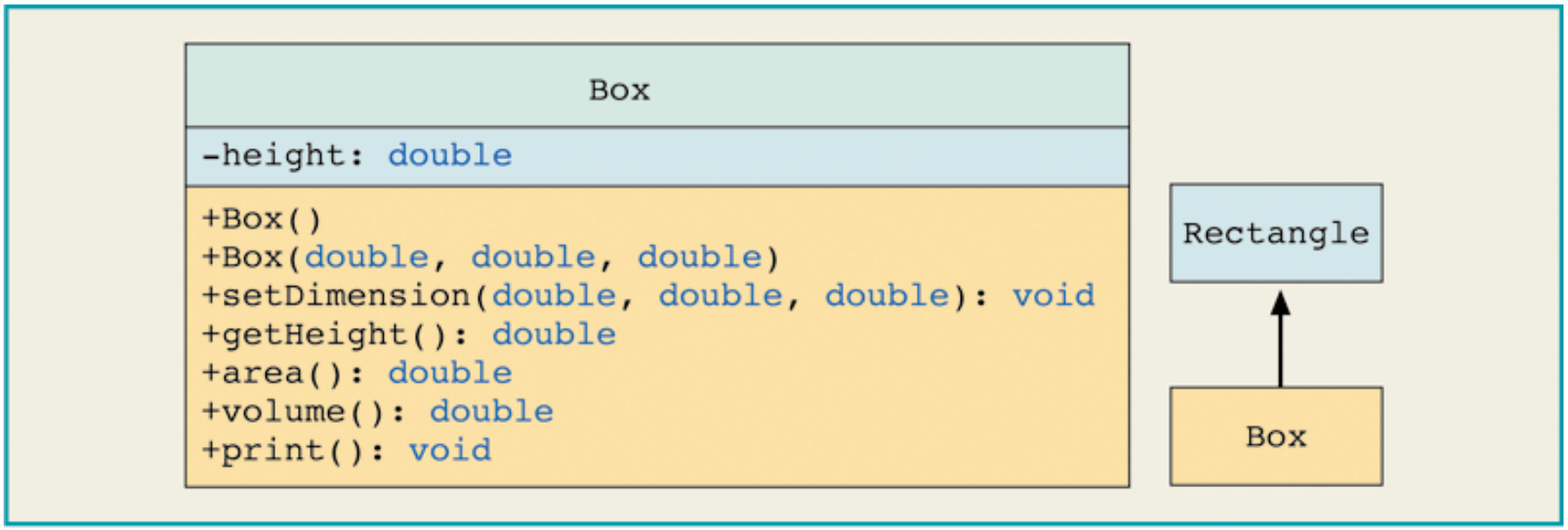

Figure 11-3 UML class diagram of the `class` Box and the inheritance hierarchy

Both a Rectangle and a Box have a surface area, but they are computed differently

Overriding Methods

- A subclass can override (redefine) the methods of the superclass
  - Objects of the subclass type will use the new method
  - Objects of the superclass type will use the original

class Rectangle

```
public double area()
{
    return  getLength() * getWidth();
}
```

class Box

```
public double area()
{
    return  2 * (getLength() * getWidth()
                 + getLength() * height
                 + getWidth() * height);
}
```

final Methods

- Can declare a method of a class final using the keyword `final`

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

- If a method of a `class` is declared `final`, it cannot be overridden with a new definition in a derived class

Calling methods of the superclass

- 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)
    ```

class Box

```
public void setDimension(double l, double w, double h)
{
     super.setDimension(l, w);
     if (h >= 0)
          height = h;
     else
          height = 0;
}}
```

Box overloads the method setDimension
(Different parameters)

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;
}
```

Access Control

- Access control keywords define which classes can access classes, methods, and members

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
none	Y	Y	N	N
private	Y	N	N	N

Polymorphism

- Can treat an object of a subclass as an object of its superclass
  - 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

- Late binding or dynamic binding (run-time binding):
  - Method to be executed is determined at execution time, not compile time
- Polymorphism: to assign multiple meanings to the same method name
- Implemented using late binding

Polymorphism (continued)

- The reference variable `name` or `nameRef` can point to any object of the `class Person` or the `class PartTimeEmployee`
- These reference variables have many forms, that is, they are polymorphic reference variables
- They can refer to objects of their own class or to objects of the classes inherited from their class

Polymorphism and References

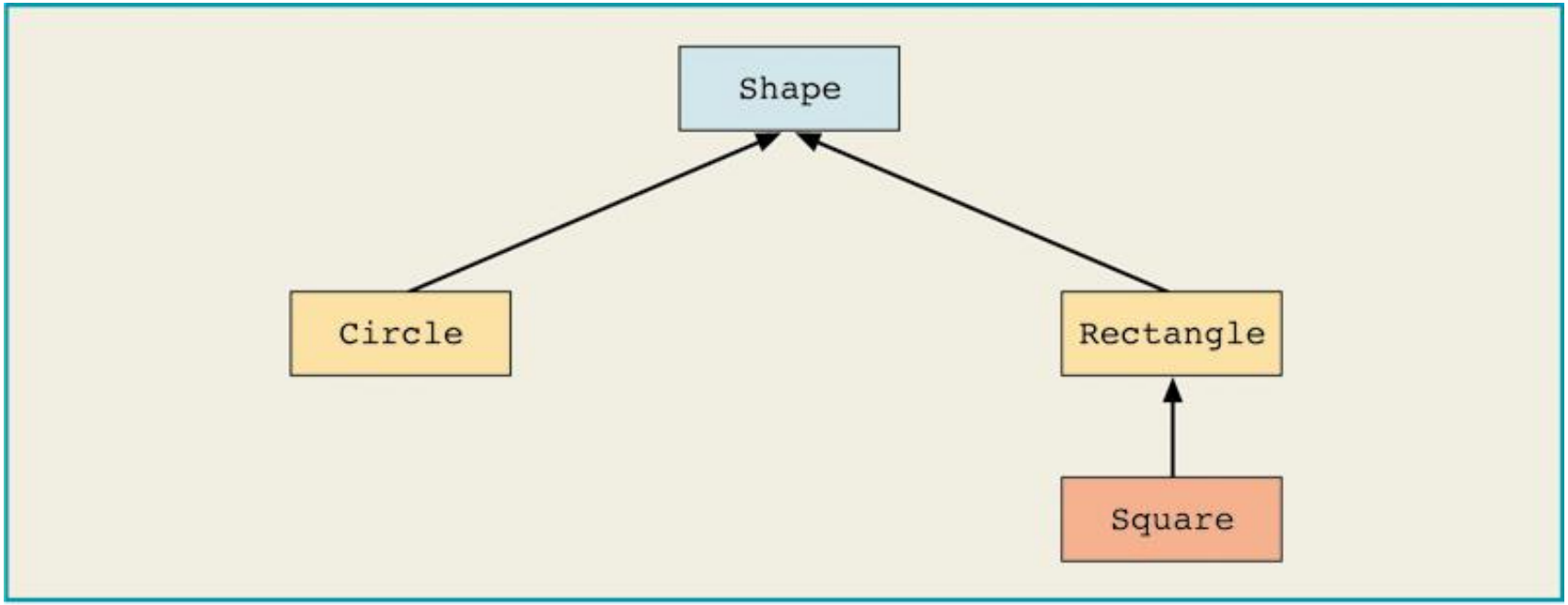

Figure 11-1 Inheritance hierarchy

```
Shape myShape = new Circle();          // allowed
Shape myShape2  = new Rectangle();     // allowed
Rectangle myRectangle = new Shame();   // NOT allowed
```

Polymorphism (continued)

- 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`
  - `Why not?`

Casting

- 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 and `supRef` points to an object of its subclass:
  - Can use a cast operator on `supRef` and make a reference variable of the subclass point to the object
  - 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 a reference variable that points to an object is of a particular class type
- This expression evaluates to `true` if `p` points to an object of the `class` `BoxShape`; otherwise it evaluates to `false`:

```
p instanceof BoxShape
```

Abstract Methods

- A method that has only the heading with no body
- Must be implemented in a subclass
- Must be declared abstract

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

Abstract Classes

- A class that is declared with the reserved word `abstract` in its heading
- An abstract class can contain instance variables, constructors, finalizers, and non-abstract methods
- An abstract class can contain abstract methods

Abstract Classes (continued)

- If a class contains an abstract method, the class must be declared abstract
- You cannot instantiate an object of an abstract class type; 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

- A 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

Composition

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

Composition Example

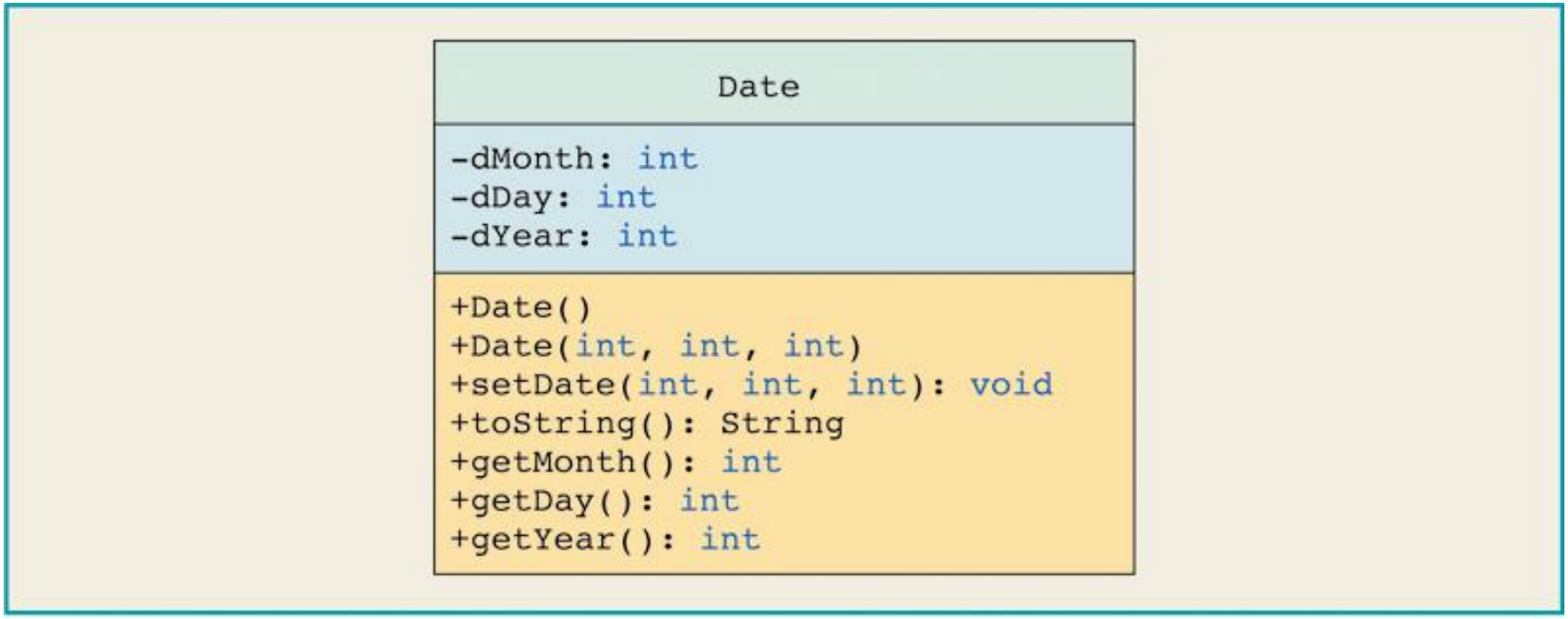

Figure 11-9 UML class diagram of the `class` `Date`

Composition Example (continued)

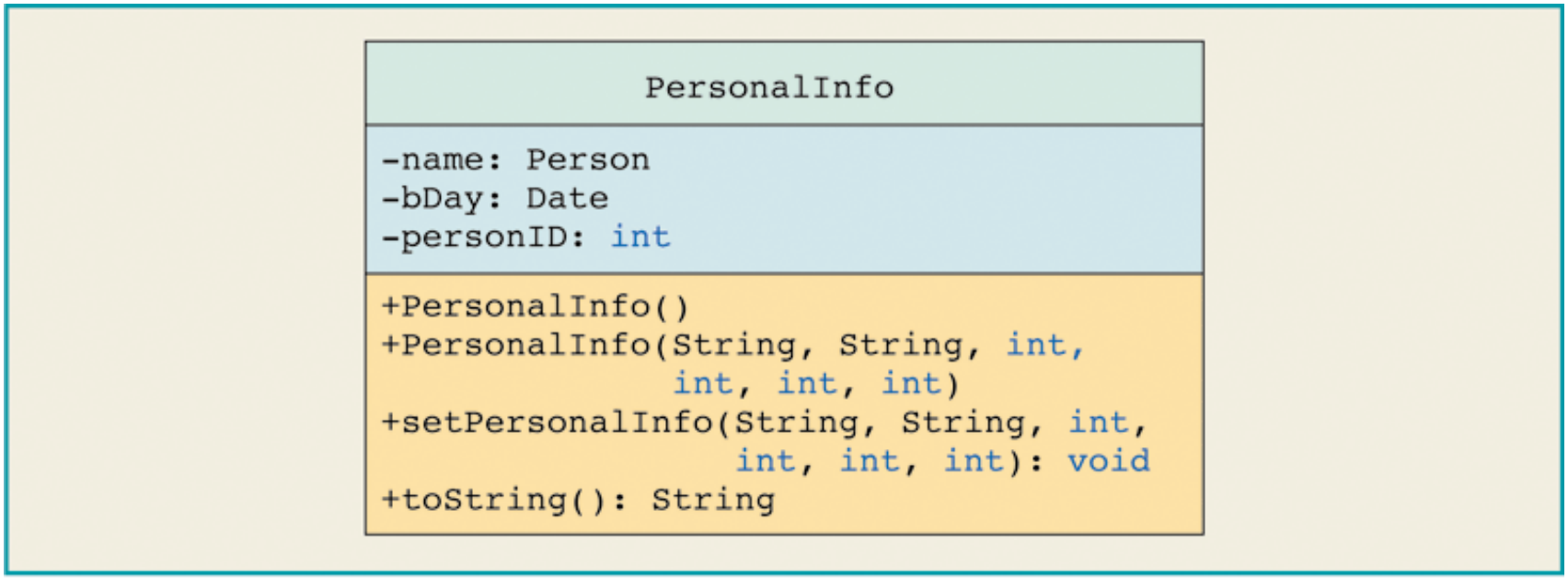

Figure 11-10 UML class diagram of the `class` `PersonalInfo`