Chapter 9 Objects and Classes

Objectives

- To describe objects and classes, and use classes to model objects (§9.2).
- To use UML graphical notation to describe classes and objects (§9.2).
- To demonstrate how to define classes and create objects (§9.3).
- To create objects using constructors (§9.4).
- To access objects via object reference variables (§9.5).
- To define a reference variable using a reference type (§9.5.1).
- To access an object’s data and methods using the object member access operator (·) (§9.5.2).
- To define data fields of reference types and assign default values for an object’s data fields (§9.5.3).
- To distinguish between object reference variables and primitive data type variables (§9.5.4).
- To use the Java library classes `Date`, `Random`, and `Point2D` (§9.6).
- To distinguish between instance and static variables and methods (§9.7).
- To define private data fields with appropriate `get` and `set` methods (§9.8).
- To encapsulate data fields to make classes easy to maintain (§9.9).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§9.10).
- To store and process objects in arrays (§9.11).
- To create immutable objects from immutable classes to protect the contents of objects (§9.12).
- To determine the scope of variables in the context of a class (§9.13).
- To use the keyword `this` to refer to the calling object itself (§9.14).
Object-Oriented Programming

Object-oriented programming (OOP) involves programming using objects

An object represents an entity in the real world that can be distinctly identified

Examples: A student, a desk, a circle, and even a loan can all be viewed as objects

Each object has a unique identity, state, and behaviors:

- **State** of an object consists of a set of *data fields* (also known as *properties*) with their current values
- **Behavior** of an object is defined by a set of *methods*

Classes

*Classes* are constructs that define objects of the same type

- A Java *class* uses *variables* to define data fields and *methods* to define behaviors
- Additionally, a *class* provides a special type of methods, known as *constructors*, which are invoked to construct objects from that *class*
A Java Class

class SimpleCircle {
    /** The radius of this circle */
    double radius = 1.0;

    /** Construct a circle object */
    Circle() {
    }

    /** Construct a circle object */
    Circle(double newRadius) {
        radius = newRadius;
    }

    /** Return the area of this circle */
    double getArea() {
        return radius * radius * 3.14159;
    }
}

UML Notation for Classes & Objects

UML Class Diagram

SimpleCircle

radius: double
Circle()
Circle(newRadius: double)
getArea(): double
getPerimeter(): double
setRadius(newRadius: double): void

circle1: SimpleCircle
radius = 1.0

circle2: SimpleCircle
radius = 25

circle3: SimpleCircle
radius = 125

UML notation for objects
Example 1: Defining Classes & Creating Objects

**Objective:** Demonstrate creating objects, accessing data, and invoking methods of the `SimpleCircle` class

```
TestSimpleCircle Run
```

Example 2: Defining Classes & Creating Objects

```
TV
channel: int
volumeLevel: int
on: boolean

+TV()
+turnOn(): void
+turnOff(): void
+setChannel(newChannel: int): void
+setVolume(newVolumeLevel: int): void
+channelUp(): void
+channelDown(): void
+volumeUp(): void
+volumeDown(): void
```

The current channel (1 to 120) of this TV.
The current volume level (1 to 7) of this TV.
Indicates whether this TV is on/off.

Constructs a default TV object.
Turns on this TV.
Turns off this TV.
Sets a new channel for this TV.
Sets a new volume level for this TV.
Increases the channel number by 1.
Decreases the channel number by 1.
Increases the volume level by 1.
Decreases the volume level by 1.
# Constructors

<table>
<thead>
<tr>
<th>What are they?</th>
<th>They are a special kind of methods that are invoked to construct and initialize objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do we define them?</td>
<td>They must have the same name as the class itself. They do not have a return type - not even <code>void</code></td>
</tr>
</tbody>
</table>
<pre><code>| `Circle()` { } |
| `Circle(double newRadius) { radius = newRadius; }` |
</code></pre>
<p>| How do we invoke them? | They are invoked using the <code>new</code> operator |
| <code>new Circle();</code> |
| <code>new Circle(5.0);</code> |</p>

## Default & No-Args Constructors

- A constructor without parameters is called a **no-args constructor**
  
  **Example:** `Circle() { }`

- A class may be **defined without constructors**

- In that case, a no-arg constructor with an empty body is **implicitly defined** in the class

- This constructor, called the **default constructor**, is provided automatically **only if no constructors are explicitly defined** in the class
Declaring Object Reference Variables

- To reference an object, assign the object to a reference variable

- To declare a reference variable, use the syntax:

  ```
  ClassName objectRefVar;
  ```

  **Example:**
  ```
  Circle myCircle;
  ```

Declaring & Constructing Objects in a Single Step

  ```
  ClassName objectRefVar = new ClassName();
  ```

  **Example:**
  ```
  Circle myCircle = new Circle();
  ```
Accessing Object’s Members

Referencing the object’s data:

objectRefVar.data

Example: myCircle.radius

Invoking the object’s method:

objectRefVar.methodName(arguments)

Example: myCircle.getArea()
Trace Code, cont.

Circle myCircle = new Circle(5.0);
myCircle no value

Circle yourCircle = new Circle();

myCircle reference value

Assign object reference to myCircle
Create a circle

Create a circle

: Circle
radius: 5.0
Trace Code, cont.

Circle myCircle = new Circle(5.0);
Circle yourCircle = new Circle();
yourCircle.radius = 100;

myCircle reference value

Circle
radius: 5.0

yourCircle no value

Declare yourCircle

Create a new Circle object

Circle
radius: 1.0
Trace Code, cont.

Circle myCircle = new Circle(5.0);
Circle yourCircle = new Circle();
yourCircle.radius = 100;

Assign object reference to yourCircle

Change radius in yourCircle
Caution

Recall that, to invoke a method in the Math class, we use

\[ \text{Math.methodName(arguments)} \] (e.g., Math.pow(3, 2.5))

Can we invoke \text{getArea()} using \text{SimpleCircle.getArea()}?

The answer is \text{no}. All the methods used before this chapter were static methods, which are defined using the \text{static} keyword. However, \text{getArea()} is not static. It must be invoked from an object using

\[ \text{objectRefVar.methodName(arguments)} \]

Example: \text{myCircle.getArea()}

Reference Data Fields

The data fields can be of primitive type or reference type

\textbf{Example:} The following \textbf{Student} class contains a data field \texttt{name} of the \texttt{String} type, which is a reference type

```java
public class Student {
    String name; // name has default value null
    int age; // age has default value 0
    boolean isMathMajor; // isMathMajor default value is false
    char gender; // gender has default value '\u0000'
}
```
Default Value for a Data Field

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>0</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td>'\u0000'</td>
</tr>
<tr>
<td>Reference</td>
<td>null</td>
</tr>
</tbody>
</table>

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.

Example:

```java
public class Test {
    public static void main(String[] args) {
        Student student = new Student();
        System.out.println("name? " + student.name);
        System.out.println("age? " + student.age);
        System.out.println("isMathMajor? " + student.isMathMajor);
        System.out.println("gender? " + student.gender);
    }
}
```

Example: Local Variables inside Methods

Java assigns no default value to local variables inside methods

```java
public class Test {
    public static void main(String[] args) {
        int x; // x has no default value
        String y; // y has no default value
        System.out.println("x is " + x);
        System.out.println("y is " + y);
    }
}
```

Compile-time error: variable not initialized
Differences between Variables of Primitive Data Types & Object Types

Primitive type:

```
int i = 1
```

Object type:

```
Circle c
```

Created using `new Circle()`:

```
c: Circle
radius = 1
```

Copying Variables of Primitive Data Types & Object Types

**Primitive type assignment** `i = j`

Before:

```
i 1
j 2
```

After:

```
i 2
j 2
```

**Object type assignment** `c1 = c2`

Before:

```
c1: Circle
radius = 5
```
```
c2: Circle
radius = 9
```

After:

```
c1: Circle
radius = 5
```
```
c2: Circle
radius = 9
```
Garbage Collection

As shown in the previous figure, after the assignment statement \( c1 = c2 \), \( c1 \) points to the same object referenced by \( c2 \). The object previously referenced by \( c1 \) is no longer referenced. That object is now garbage.

Garbage is automatically collected by JVM.

TIP: If you know that an object is no longer needed, you can explicitly assign \texttt{null} to its reference variable. The JVM will automatically collect the space if the object is not referenced by any variable.

Instance and Static Variables, Constants & Methods

<table>
<thead>
<tr>
<th>Instance</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>belong to a specific instance</td>
</tr>
<tr>
<td>Methods</td>
<td>are invoked by an instance of a class</td>
</tr>
<tr>
<td>Constants</td>
<td>are final variables shared by all the instances of the class</td>
</tr>
</tbody>
</table>
Example: Using Instance & Class Variables & Methods

Objective: Demonstrate the roles of instance and class variables and their uses. This example adds a class variable `numberOfObjects` to track the number of Circle objects created.
Visibility Modifiers & Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package

**public:** The class, data, or method is visible to any class in any package

**private:** The data or methods can be accessed only by the declaring class

The `get()` and `set()` methods (called accessor and mutator methods, respectively) are used to read and modify private properties

Visibility Modifiers Example #1

```
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();
    }
}
```

**private** modifier restricts access to within a class

**default** modifier restricts access to within a package

**public** modifier enables unrestricted access
Visibility Modifiers Example #2

```java
package p1;
class C1 {
    ...
}

package p1:
public class C2 {
    can access C1
}

package p2:
public class C3 {
    cannot access C1;
    can access C2;
}
```

default modifier restricts access to within a *package*
public modifier enables *unrestricted* access

---

**NOTE**

- An object cannot access its private members, as shown in (b)
- It is OK, however, if the object is declared in its own class, as shown in (a)

```java
public class C {
    private boolean x;
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x);
        System.out.println(c.convert());
    }

    // (a) This is okay because object c is used inside the class C.
    private int convert() {
        return x ? 1 : -1;
    }
}

public class Test {
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x);
        System.out.println(c.convert());
    }
}
```

(b) This is wrong because `x` and `convert` are private in class `C`. 
Why Data Fields Should Be *private*

1. To protect data

2. To make code easy to maintain

The practice of preventing direct access to data from outside the class is called *data field encapsulation*.

---

Data Field Encapsulation

For *data field encapsulation*, we make the *data fields* in a class *private* and provide access to them via *public methods*.

<table>
<thead>
<tr>
<th>Circle</th>
<th>The radius of this circle (default: 1.0). The number of circle objects created.</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Circle()</td>
<td>Constructs a default circle object.</td>
</tr>
<tr>
<td>+Circle(radius: double)</td>
<td>Constructs a circle object with the specified radius.</td>
</tr>
<tr>
<td>+getRadius(): double</td>
<td>Returns the radius of this circle.</td>
</tr>
<tr>
<td>+setRadius(radius: double): void</td>
<td>Sets a new radius for this circle.</td>
</tr>
<tr>
<td>+getNumberOfObjects(): int</td>
<td>Returns the number of circle objects created.</td>
</tr>
<tr>
<td>+getArea(): double</td>
<td>Returns the area of this circle.</td>
</tr>
</tbody>
</table>

The - sign indicates private modifier.
Passing Objects to Methods

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

Passing Objects to Methods, cont.

![Diagram showing stack and heap for passing objects to methods.]

- Stack:
  - Activation record for the printArea method:
    - int times: 5
    - Circle c: reference
  - Activation record for the main method:
    - int n: 5
    - myCircle: reference

- Heap:
  - A Circle object

Pass-by-value (here the value is 5)
Pass-by-value (here the reference for the object)
Array of Objects

Circle[] circleArray = new Circle[10];

An array of objects is actually an array of reference variables. So invoking circleArray[1].getArea() involves two levels of referencing as shown in the next figure:

circleArray references to the entire array

circleArray[1] references to a Circle object

Array of Objects, cont.

Circle[] circleArray = new Circle[10];

circleArray reference → circleArray[0] → Circle object 0

circleArray[1] → Circle object 1
...

circleArray[9] → Circle object 9

Example: Summarizing the areas of the circles
Immutable Objects and Classes

If the contents of an object cannot be changed once the object is created, the object is called an immutable object and its class is called an immutable class.

Example: If you delete the set method in the Circle class in Listing 8.10, the class would be immutable because radius is private and cannot be changed without a set() method.

A class with all private data fields and without mutators is not necessarily immutable. For example, the following class Student has all private data fields and no mutators, but it is mutable.

Example

```java
public class Student {
    private int id;
    private BirthDate birthDate;
    public Student(int ssn, int year, int month, int day) {
        id = ssn;
        birthDate = new BirthDate(year, month, day);
    }
    public int getId() {
        return id;
    }
    public BirthDate getBirthDate() {
        return birthDate;
    }
}

public class BirthDate {
    private int year;
    private int month;
    private int day;
    public BirthDate(int newYear, int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }
    public void setYear(int newYear) {
        year = newYear;
    }
}

public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}```
What Class is Immutable?

For a class to be immutable, it must:

- mark all data fields private
- provide no mutator methods and no accessor methods that would return a reference to a mutable data field object

Scope of Variables

The scope of a variable is the region of code within which a variable is visible

- The scope of instance and static variables is the entire class. They can be declared anywhere inside a class
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable
  - A local variable must be initialized explicitly before it can be used
The **this** Keyword

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

**Example:** Reference the Hidden Data Fields

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

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

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

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

```java
F f1 = new F(); F f2 = new 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
```
**Example: Calling Overloaded Constructor**

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

    public Circle(double radius) {
        this.radius = radius;
        // this must be explicitly used to reference the data field radius of the object being constructed
    }

    public Circle() {
        this(1.0); // this is used to invoke another constructor
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
        // Every instance variable belongs to an instance represented by this, which is normally omitted
    }
}
```

---

**The Date Class**

- Java provides a system-independent encapsulation of date and time in the `java.util.Date` class
- You can use the `Date` class to create an instance for the current date and time and use its `toString` method to return the date and time as a string

```
java.util.Date
  +Date()
  +Date(elapseTime: long)
  +toString(): String
  +getTime(): long
  +setTime(elapseTime: long): void

- Constructs a Date object for the current time.
- Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
- Returns a string representing the date and time.
- Returns the number of milliseconds since January 1, 1970, GMT.
- Sets a new elapse time in the object.
```
**Date Class Example**

The following code

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

displays a string like:

```
Sun Mar 09 13:50:19 EST 2003
```

**The Random Class**

We have used `Math.random()` to obtain a random double value between 0.0 and 1.0 (excluding 1.0). A more useful random number generator is provided in the `java.util.Random` class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Random()</code></td>
<td>Constructs a Random object with the current time as its seed.</td>
</tr>
<tr>
<td><code>Random(seed: long)</code></td>
<td>Constructs a Random object with a specified seed.</td>
</tr>
<tr>
<td><code>nextInt(): int</code></td>
<td>Returns a random int value.</td>
</tr>
<tr>
<td><code>nextInt(n: int): int</code></td>
<td>Returns a random int value between 0 and n (exclusive).</td>
</tr>
<tr>
<td><code>nextLong(): long</code></td>
<td>Returns a random long value.</td>
</tr>
<tr>
<td><code>nextDouble(): double</code></td>
<td>Returns a random double value between 0.0 and 1.0 (exclusive).</td>
</tr>
<tr>
<td><code>nextFloat(): float</code></td>
<td>Returns a random float value between 0.0F and 1.0F (exclusive).</td>
</tr>
<tr>
<td><code>nextBoolean(): boolean</code></td>
<td>Returns a random boolean value.</td>
</tr>
</tbody>
</table>
Random Class Example

If two Random objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two Random objects with the same seed 3.

```java
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
    System.out.print(random1.nextInt(1000) + " ");
Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
    System.out.print(random2.nextInt(1000) + " ");
```

From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961

The Point2D Class

Java API has a convenient Point2D class in the javafx.geometry package for representing a point in a two-dimensional plane.