Chapter 8 Objects and Classes
Motivations

After learning the preceding chapters, you are capable of solving many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large scale software systems. Suppose you want to develop a graphical user interface as shown below. How do you program it?
Objectives

- To describe objects and classes, and use classes to model objects (§8.2).
- To use UML graphical notation to describe classes and objects (§8.2).
- To demonstrate how to define classes and create objects (§8.3).
- To create objects using constructors (§8.4).
- To access objects via object reference variables (§8.5).
- To define a reference variable using a reference type (§8.5.1).
- To access an object’s data and methods using the object member access operator (.) (§8.5.2).
- To define data fields of reference types and assign default values for an object’s data fields (§8.5.3).
- To distinguish between object reference variables and primitive data type variables (§8.5.4).
- To use the Java library classes **Date**, **Random**, and **JFrame** (§8.6).
- To distinguish between instance and static variables and methods (§8.7).
- To define private data fields with appropriate **get** and **set** methods (§8.8).
- To encapsulate data fields to make classes easy to maintain (§8.9).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§8.10).
- To store and process objects in arrays (§8.11).
Object-oriented programming (OOP) involves programming using objects. An object represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors. The state of an object consists of a set of data fields (also known as properties) with their current values. The behavior of an object is defined by a set of methods.
An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.
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 the class.
class Circle {
    /** 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 Class Diagram

<table>
<thead>
<tr>
<th>Circle</th>
<th>Class name</th>
<th>UML notation for objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius: double</td>
<td>Data fields</td>
<td></td>
</tr>
<tr>
<td>Circle()</td>
<td>Constructors and methods</td>
<td></td>
</tr>
<tr>
<td>Circle(newRadius: double)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getArea(): double</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>circle1: Circle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>radius = 1.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>circle2: Circle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>radius = 25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>circle3: Circle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>radius = 125</td>
<td></td>
</tr>
</tbody>
</table>
Example: Defining Classes and Creating Objects

Objective: Demonstrate creating objects, accessing data, and using methods.

TestSimpleCircle  Run
Example: Defining Classes and Creating Objects

Objective: Demonstrate creating objects, accessing data, and using methods.

TV
TestTV
Run
Constructors

Constructors are a special kind of methods that are invoked to construct objects.

```java
Circle() {
}

Circle(double newRadius) {
    radius = newRadius;
}
```
Constructors, cont.

A constructor with no parameters is referred to as a no-arg constructor.

- Constructors 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. Constructors play the role of initializing objects.
Creating Objects Using Constructors

new ClassName();

Example:
new Circle();

new Circle(5.0);
Default Constructor

A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor, called a 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:

```java
ClassName objectRefVar;
```

Example:
```java
Circle myCircle;
```
Declaring/Creating Objects in a Single Step

ClassName objectRefVar = new ClassName();

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

- Referencing the object’s data:
  \[ \text{objectRefVar.data} \]
  e.g., \text{myCircle.radius} \\

- Invoking the object’s method:
  \[ \text{objectRefVar.methodName(arguments)} \]
  e.g., \text{myCircle.getArea()}
Trace Code

```java
Circle myCircle = new Circle(5.0);

SCircle yourCircle = new Circle();
yourCircle.radius = 100;
```
Circle myCircle = new Circle(5.0);
Circle yourCircle = new Circle();
yourCircle.radius = 100;
Trace Code, cont.

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

Assign object reference to myCircle

myCircle

reference value

Circle

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

myCircle reference value

: Circle
radius: 5.0

yourCircle no value

: Circle
radius: 1.0

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

Assign object reference to yourCircle

myCircle reference value

yourCircle reference value

: Circle
radius: 5.0

: Circle
radius: 1.0
Circle myCircle = new Circle(5.0);
Circle yourCircle = new Circle();

yourCircle.radius = 100;

myCircle reference value

Circle
radius: 5.0

yourCircle reference value

Circle
radius: 100.0

Change radius in yourCircle
Caution

Recall that you use
\[
\text{Math.methodName(arguments)} \quad (\text{e.g., Math.pow(3, 2.5)})
\]

to invoke a method in the \text{Math} class. Can you invoke \text{getArea()} using \text{Circle1.getArea()}? The answer is no. All the methods used before this chapter are static methods, which are defined using the \text{static} keyword. However, \text{getArea()} is non-static. It must be invoked from an object using

\[
\text{objectRefVar.methodName(arguments)} \quad (\text{e.g., myCircle.getArea()}).
\]

More explanations will be given in the section on “Static Variables, Constants, and Methods.”
Reference Data Fields

The data fields can be of reference types. For example, the following `Student` class contains a data field `name` of the `String` type.

```java
public class Student {
    String name; // name has default value null
    int age; // age has default value 0
    boolean isScienceMajor; // isScienceMajor has default value false
    char gender; // c has default value '\u0000'
}
```
The null Value

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.
Default Value for a Data Field

The default value of a data field is null for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. However, Java assigns no default value to a local variable inside a method.

```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("isScienceMajor? " + student.isScienceMajor);
        System.out.println("gender? " + student.gender);
    }
}
```
Example

Java assigns no default value to a local variable inside a method.

```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);
    }
}
```

Compilation error: variables not initialized
Differences between Variables of Primitive Data Types and Object Types

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>int i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object type</th>
<th>Circle c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>reference</td>
</tr>
</tbody>
</table>

Created using new Circle()

- c: Circle
- radius = 1
Copying Variables of Primitive Data Types and Object Types

**Primitive type assignment** $i = j$

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$i$</td>
</tr>
<tr>
<td>$j$</td>
<td>$j$</td>
</tr>
</tbody>
</table>

Before: $i = 1$, $j = 2$

After: $i = 2$, $j = 2$

**Object type assignment** $c1 = c2$

```
Before:
c1   c2
| c1: Circle | C2: Circle |
| radius = 5 | radius = 9 |
```

After:
```
c1   c2
| c1: Circle | C2: Circle |
| radius = 5 | radius = 9 |
```
Garbage Collection

As shown in the previous figure, after the assignment statement \( c_1 = c_2 \), \( c_1 \) points to the same object referenced by \( c_2 \). The object previously referenced by \( c_1 \) is no longer referenced. This object is known as garbage. Garbage is automatically collected by JVM.
Garbage Collection, cont

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

<table>
<thead>
<tr>
<th>java.util.Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Date()</td>
</tr>
<tr>
<td>+Date(elapseTime: long)</td>
</tr>
<tr>
<td>+toString(): String</td>
</tr>
<tr>
<td>+getTime(): long</td>
</tr>
<tr>
<td>+setTime(elapseTime: long): void</td>
</tr>
</tbody>
</table>

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.

The + sign indicates public modifier
The Date Class Example

For example, the following code

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

# The Random Class

You 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><code>java.util.Random</code></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>
The 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
Displaying GUI Components

When you develop programs to create graphical user interfaces, you will use Java classes such as JFrame, JButton, JRadioButton, JComboBox, and JList to create frames, buttons, radio buttons, combo boxes, lists, and so on. Here is an example that creates two windows using the JFrame class.

TestFrame

Run
```java
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);

JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```
Trace Code

JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
 JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
 JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);

JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);

JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);

Set visible property
Adding GUI Components to Window

You can add graphical user interface components, such as buttons, labels, text fields, combo boxes, lists, and menus, to the window. The components are defined using classes. Here is an example to create buttons, labels, text fields, check boxes, radio buttons, and combo boxes.
Instance Variables, and Methods

Instance variables belong to a specific instance.

Instance methods are invoked by an instance of the class.
Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

Static methods are not tied to a specific object.

Static constants are final variables shared by all the instances of the class.
To declare static variables, constants, and methods, use the static modifier.
Static Variables, Constants, and Methods, cont.

UML Notation:
+: public variables or methods
underline: static variables or methods

Circle
radius: double
numberOfObjects: int

getNumberOfObjects(): int
+getArea(): double

After two Circle objects were created, numberOfObjects is 2.
Example of Using Instance and Class Variables and Method

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.

CircleWithStaticMembers

TestCircleWithStaticMembers
Visibility Modifiers and 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 are used to read and modify private properties.
The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the 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 F {
    private boolean x;

    public static void main(String[] args) {
        F f = new F();
        System.out.println(f.x);
        System.out.println(f.convert());
    }

    private int convert(boolean b) {
        return x ? 1 : -1;
    }
}

(a) This is OK because object f is used inside the F class

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

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

To protect data.

To make class easy to maintain.
## Access Levels

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>default modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Example of Data Field Encapsulation

<table>
<thead>
<tr>
<th>Circle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-radius: double</td>
<td>The radius of this circle (default: 1.0).</td>
</tr>
<tr>
<td>-numberOfObjects: int</td>
<td>The number of circle objects created.</td>
</tr>
<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>+getNumberOfObject(): 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)

TestPassObject

Run
Passing Objects to Methods, cont.

Space required for the main method
int n: 5
myCircle: reference

Space required for the printAreas method
int times: 5
Circle c: reference

Pass by value (here the value is 5)
Pass by value (here the value is the reference for the object)

A circle object

Heap

Stack
Array of Objects

```java
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[0]
circleArray[1]
...  
circleArray[9]  
```
Array of Objects, cont.

Summarizing the areas of the circles

TotalArea

Run