**CHAPTER 9: OBJECTS AND CLASSES**

* 1. **INTRODUCTION**

Object-oriented programming enables you to develop large-scale software and GUIs effectively.

Object-oriented programming is essentially a technology for developing reusable software.

* 1. **DEFINING CLASSES FOR OBJECTS**

A class defines the properties and behaviors for objects.

Object-oriented programming (OOP) involves programming using objects.

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

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.
* 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.

Objects of the same type are defined using a common class. A class is a template, blueprint, or contract that defines what an object’s data fields and methods will be.

An object is an instance of a class.

You can create many instances of a class.

Creating an instance is referred to as instantiation.

The terms object and instance are often interchangeable.

A Java class uses variables to define data fields and methods to define actions. In addition, a class provides methods of a special type, known as constructors, which are invoked to create a new object.

A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.

* 1. **CONSTRUCTING OBJECTS USING CONSTUCTORS**

A constructor is invoked to create an object using the new operator. Constructors are a special kind of method.

They have three peculiarities:

* 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. Constructors play the role of initializing objects.

The constructor has exactly the same name as its defining class.

To construct an object from a class, invoke a constructor of the class using the new operator, as follows:

new ClassName(arguments);

* 1. **ACCESSING OBJECTS VVIA REFERENCE VARIABLES**

An object’s data and methods can be accessed through the dot (.) operator via the object’s reference variable.

Newly created objects are allocated in the memory. They can be accessed via reference variables.

Objects are accessed via the object’s **reference variables**, which contain references to the objects.

Such variables are declared using the following syntax:

ClassName objectRefVar;

A class is essentially a programmer-defined type. A class is a reference type, which means that a variable of the class type can reference an instance of the class.

* + 1. **Accessing an Object’s Data and Methods**

In OOP terminology, an object’s member refers to its data fields and methods. After an object is created, its data can be accessed and its methods can be invoked using the **dot operator (.)** also known as the **object member access operator**:

* **objectRefVar.dataField** references a data field in the object.
* **objectRefVar.method(arguments)** invokes a method on the object.
  + 1. **Reference Data Fields and the null Value**

The data fields can be of reference types.

For example, the following Student class contains a data field name of the String type. String is a predefined Java class.

class Student {

String name;

int age;

boolean isScienceMajor;

char gender;

}

If a data field of a reference type does not reference any object, the data field holds a special Java value, **null. null** is a literal just like **true** and **false**. While **true** and **false** are Boolean literals, null is a literal for a reference type. **null** is not a Java keyword, but it is a reserved word in Java.

* + 1. **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 type of value the variable can hold.

For a variable of a primitive type, the value is of the primitive type.

For a variable of a reference type, the value is a reference to where an object is located.

* 1. **USING CLASSES FROM THE JAVA LIBRARY**

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

* + 1. **THE Date CLASS**

Java provides a system-independent encapsulation of date and time in the **java.util.Date** class.

|  |  |
| --- | --- |
| java.util.Date | |
| +Date() | Constructs a Date object for the current time. |
| +Date(elapsedTime: long) | Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT. |
| +toString(): String | Returns a string representing the date and time. |
| +getTime(): long | Returns the number of milliseconds since January 1, 1970, GMT. |
| +setTime(elapsedTime: long):void | Sets a new elapse time in the object. |

A **Date** object represents a specific date and time.

You can use the no-arg constructor in the Date class to create an instance for the current date and time, the **getTime()** method to return the elapsed time in milliseconds since January 1, 1970, GMT, and the **toString()** method to return the date and time as a string.

* + 1. **THE Random CLASS**

You have used **Math.random()** to obtain a random double value between 0.0 and 1.0 (excluding 1.0).

Another way to generate random numbers is to use the **java.util.Random** class.

|  |  |
| --- | --- |
| Java.util.Random | |
| +Random() | Constructs a Random object with the current time as its seed. |
| +Random(seed: long) | Constructs a Random object with a specified seed |
| +nextInt(): int | Returns a random int value |
| +nextInt(n: int): int | Returns a random int value between 0 and n (excluding n) |
| +nextLong(): long | Returns a random long value |
| +nextDouble(): double | Returns a random double value between 0.0 and 1.0 (excluding 1.0) |
| +nextFLoat(): float | Returns a random float value between 0.0F and 1.0F (excluding 1.0F). |
| +nextBoolean(): boolean | Returns a random boolean value |

A **Random** object can be used to generate random values.

* + 1. **THE Point2D CLASS**

Java API has a convenient Point2D class in the **javafx.geometry** package for representing a point in a two-dimensional plane. The UML diagram for the class is shown below.

|  |  |
| --- | --- |
| Javafx.geometry.Point2D | |
| +Point2D(x: double, y: double) | Constructs a Point2D x- and y-coordinates. |
| +distance(x: double, y: double): double | Returns the distance between this point and the specified point (x, y). |
| +distance(p: Point2D): double | Returns the distance between this point and the specified point p. |
| +getX(): double | Returns the x-coordinate from this point. |
| +getY(): double | Returns the y-coordinate from this point. |
| +midpoint(p: Point2D): Point2D | Returns the midpoint between this point and point p. |
| +toString(): String | Returns a string representation for the point. |

A **Point2D** object represents a point with x- and y-coordinates.

* 1. **STATIC VARIABLES, CONSTANTS, AND METHODS**

A static variable is shared by all objects of the class. A static method cannot access instance members (i.e., instance data fields and methods) of the class.

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; it is not shared among objects of the same class.

To declare a static variable or define a static method, put the modifier **static** in the variable or method declaration. The static variable **numberOfObjects** and the static method **getNumberOfObjects()** can be declared as follows:

**static int numberOfObjects;**

**static int getNumberObjects() {**

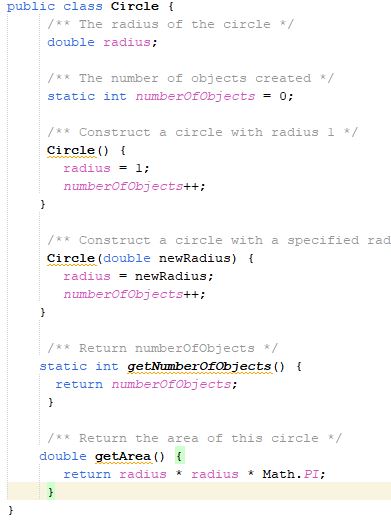
**return numberOfObjects;**

**}**

Constants in a class are shared by all objects of the class. Thus, constants should be declared as final static. For example, the constant PI in the Math class is defined as follows:

**final static double PI = 3.14159265358979323846;**

Example:



Method **getNumberOfObjects()** in Circle is a static method. All the methods in the Math class are static. The main method is static, too.

Instance methods (e.g., **getArea())** and instance data (e.g., **radius**) belong to instances and can be used only after the instances are created. They are accessed via a reference variable.

* 1. **VISIBILITY MODIFIERS**

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

You can use the public visibility modifier for classes, methods, and data fields to denote they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as **package-private or package-access**.

**Note:** Packages can be used to organize classes. To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

**package packageName;**

If a class is defined without the package statement, it is said to be placed in the default package.

The **private** modifier applies only to the members of a class.

The **public** modifier can apply to a class or members of a class. Using the modifiers public and private on local variables would cause a compile error.

**Note:** In most cases, the constructor should be **public**. However, if you want to prohibit the user from creating an instance of a class, use a **private constructor**.

* 1. **DATA FIELD ENCAPSULATION**

Making data fields private protects data and makes the class easy to maintain.

To prevent direct modifications of data fields, you should declare the data fields private, using the private modifier. This is known as **data field encapsulation**.

A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field.

To make a private data field accessible, provide a getter method to return its value.

To enable a private data field to be updated, provide a setter method to set a new value.

A getter method is also referred to as an accessor and a setter to a mutator.

A getter method has the following signature:

public returnType getPropertyName()

If the **returnType** is **boolean**, the getter method should be defined as follows by convention:

public boolean isPropertyName()

A setter method has the following signature:

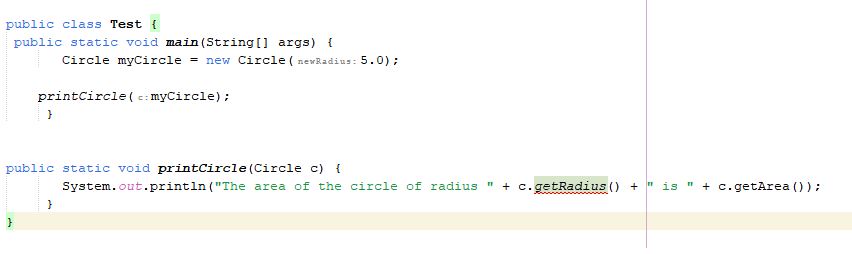
public void setPropertyName(dataType propertyValue)

* 1. **PASSING OBJECTS TO METHODS**

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

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object.

The following code passes the **myCircle** object as an argument to the **printCircle** method:

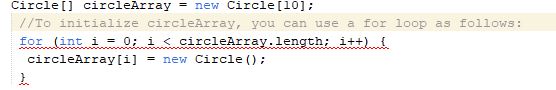


Java uses exactly one mode of passing arguments: pass-by-value.

* 1. **ARRAY OF OBJECTS**

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

You can also create arrays of objects. For example, the following statement declares and creates an array of 10 Circle objects:

****

An array of objects is actually an array of reference variables. Thus, invoking **circleArray[1]** **.getArea()** involves two levels of referencing:

* **circleArray** references the entire array, and
* **circleArray[1]** references a Circle object.

**Note:** When an array of objects is created using the new operator, each element in the array is a reference variable with a default value of null.

* 1. **IMMUTABLE OBJECTS AND CLASSES**

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

Normally, you create an object and allow its contents to be changed later. However, occasion ally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object as **immutable object** and its class as **immutable class**.

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A class with all private data fields and no mutators is not necessarily immutable.

For a class to be immutable, it must meet the following requirements:

* All data fields must be private.
* There can’t be any mutator methods for data fields.
* No accessor methods can return a reference to a data field that is mutable.
  1. **THE SCOPE OF VARIABLES**

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

Instance and static variables in a class are referred to as the class’s variables or data fields.

A variable defined inside a method is referred to as a local variable.

The scope of a class’s variables is the entire class, regardless of where the variables are declared.

A class’s variables and methods can appear in any order in the class.

* 1. **THE This REFERENCE**

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

When an instance method is called on an object, the **this** keyword is set to this object.

You can use the **this** keyword to reference the object’s instance members in the class.

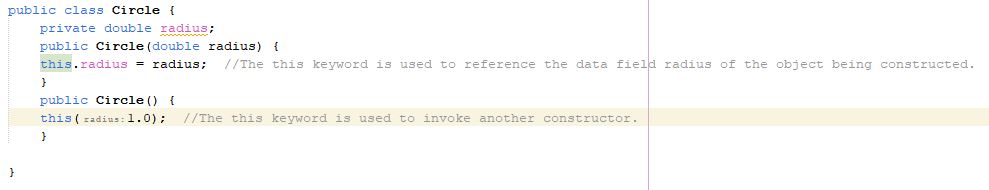
Some instructors prefer using the **this** keyword explicitly in the code, because it clearly distinguishes the instance variables from local variables. However, the **this** reference is normally omitted for brevity Nevertheless, the **this** keyword is needed to reference a data field hidden by a method or constructor parameter, or to invoke an overloaded constructor.

* + 1. **Using this to Reference Data Fields**

It is a good practice to use the data field as the parameter name in a setter method or a constructor to make the code easy to read and to avoid creating unnecessary names. In this case, you need to use the **this** keyword to reference the data field in the setter method.

* + 1. **Using this to Invoke a Constructor**

The this keyword can be used to invoke another constructor of the same class. For example, you can rewrite the Circle class as follows:

****

**Note:** Java requires that the this(arg-list) statement appear first in the constructor before any other executable statements.

**THE END!**