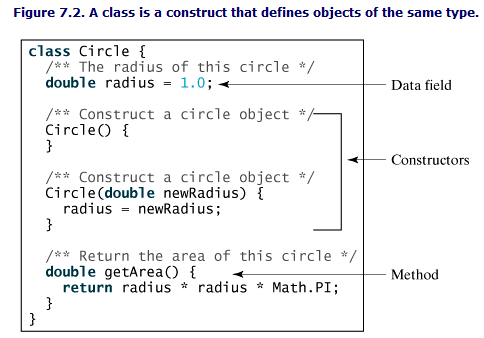
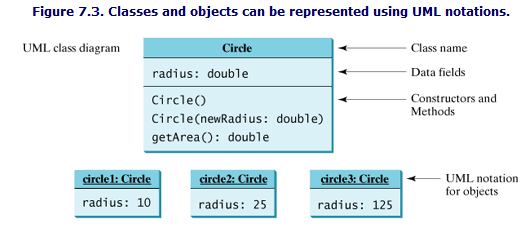
Part 2: Object-Oriented Programming

* You learned how to write simple Java applications using
  + primitive data types,
  + control statements,
  + methods, and
  + arrays, all of which are features commonly available in procedural programming languages.
* Java, however, is an object-oriented programming language that uses
  + abstraction,
  + encapsulation,
  + inheritance, and
  + polymorphism
    - to provide
      * great flexibility,
      * modularity, and
      * reusability for developing software.
* Now you will learn how to define, extend, and work with classes and their objects.

**7.2. Defining Classes for Objects**

* *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 is represented by data fields (also known as properties) with their current values.
  + The behavior of an object is defined by a set of methods.
* A circle object, for example, has
  + a data field, radius, which is the property that characterizes a circle.
  + One behavior of a circle is that its area can be computed using the method getArea().
* Objects of the same type are defined using a common class.
  + A class is a template or blueprint that defines what an object's data 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 behaviors.
* Additionally, a class provides methods of a special type, known as 
  + *constructors*,
* which are invoked when a new object is created.
* A constructor is a special kind of method.
* A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects

****

* The Circle class is different from all of the other classes you have seen thus far.
* It does not have a main method and therefore cannot be run;
* it is merely a definition used to declare and create Circle objects.
* For convenience, the class that contains the main method will be referred to as the main class
* The illustration of class templates and objects can be standardized using UML (Unified Modeling Language) notations.
* This notation, is called a UML class diagram, or simply a class diagram.
* For more information on UML, see [www.rational.com/uml/](http://www.rational.com/uml/).
* ****

**7.3. Constructors**

* The constructor has exactly the same name as the defining class.
* Like regular methods, constructors can be overloaded (i.e., multiple constructors with the same name but different signatures),
  + making it easy to construct objects with different initial data values.
* To construct an object from a class, invoke a constructor of the class using the new operator, as follows:

new ClassName(arguments);

* For example,
  + new Circle() creates an object of the Circle class using the first constructor defined in the Circle class, and
  + new Circle(5) creates an object using the second constructor defined in the Circle class.
* A class normally provides a constructor without arguments (e.g., Circle()).
  + Such a constructor is called a no-arg or no-argument constructor.
* A class may be declared 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 declared in the class.
* Constructors are a **special kind of method**, with three differences:
  + 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.

### 7.4. Accessing Objects via Reference Variables

* Newly created objects are allocated in the memory. How can they be accessed?

#### 7.4.1. Reference Variables and Reference Types

* Objects are accessed via object *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

ClassName objectRefVar;

* A class defines a type, known as a *reference type*.
* Any variable of the class type can reference to an instance of the class.
* The following statement declares the variable myCircle to be of the Circle type:

Circle myCircle;

* The variable myCircle can reference a Circle object. The next statement creates an object and assigns its reference to myCircle.

myCircle = new Circle();

You can write one statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable.

ClassName objectRefVar = new ClassName();

example:

Circle myCircle = new Circle();

The variable myCircle holds a reference to a Circle object.

Note

* Arrays are treated as objects in Java.
* Arrays are created using the new operator.
* An array variable is actually a variable that contains a reference to an array

#### 7.4.2. Accessing an Object's Data and Methods

After an object is created,

* its data can be accessed and
* its methods invoked using
  + the *dot operator (.*), also known as
  + the object member access operator:

For example,

* myCircle.radius references the radius in myCircle, and
* myCircle.getArea() invokes the getArea method on myCircle.
* The data field radius is referred to as **an instance variable** because it is dependent on a specific instance.
* For the same reason, the method getArea is referred to as an ***instance method***, because you can only invoke it on a specific instance.
* Most of the time, you create an object and assign it to a variable. Later you can use the variable to reference the object.
* Occasionally, an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable, as shown below:

new Circle();

or

System.out.println("Area is " + new Circle(5).getArea());

* The former statement creates a Circle object.
* The latter statement creates a Circle object and invokes its getArea method to return its area.
* An object created in this way is known as an **anonymous object.**

7.4.3. Example: Declaring Classes and Creating Objects

**Listing 7.1. TestCircle1.java**

**1 public class TestCircle1**

**2 {**

**3 public static void main(String[] args)**

**{**

**4 // Create a circle with radius 5.0**

**5 Circle1 myCircle = new Circle1(5.0);**

**6 System.out.println("The area of the circle of radius "**

**7 + myCircle.radius + " is " + myCircle.getArea());**

**9 // Create a circle with radius 1**

**10 Circle1 yourCircle = new Circle1();**

**11 System.out.println("The area of the circle of radius "**

**12 + yourCircle.radius + " is " + yourCircle.getArea());**

**15 yourCircle.radius = 100;**

**16 System.out.println("The area of the circle of radius "**

**17 + yourCircle.radius + " is " + yourCircle.getArea());**

**18 }**

**19 }**

**21 // Define the circle class with two constructors**

**21 class Circle1**

**22 {**

**23 double radius;**

**25 /\*\* Construct a circle with radius 1 \*/**

**26 Circle1()**

**26 {**

**27 radius = 1.0;**

**28 }**

**30 /\*\* Construct a circle with a specified radius \*/**

**31 Circle1(double newRadius)**

**{**

**32 radius = newRadius;**

**33 }**

**34**

**35 /\*\* Return the area of this circle \*/**

**36 double getArea()**

**{**

**37 return radius \* radius \* Math.PI;**

**38 }**

**39 }**

* The program contains two classes. The first class, TestCircle1, is the main class. Its sole purpose is to test the second class, Circle1. Every time you run the program, the JVM invokes the main method in the main class.
* You can put the two classes into one file, **but only one class in the file can be a public class**.
* Furthermore, the public class must have the **same name as the file name** and the **main method must be in a public class**. Therefore, the file name is TestCircle1.java if the TestCircle1 and Circle1 classes are both in the same file.
* The main class contains the main method (line 3) that creates two objects.
  + The constructor Circle1(5.0) was used to create myCircle with a radius of 5.0 (line 5), and
  + The constructor Circle1() was used to create yourCircle with a radius of 1.0 (line 10).
* These two objects (referenced by myCircle and yourCircle) have different data but share the same methods. Therefore, you can compute their respective areas by using the getArea() method.
* There are many ways to write Java programs. For instance, you can combine the two classes in the example into one
* **Listing 7.2. Circle1.java**

1 public class Circle1

{

3 public static void main(String[] args)

{

5 Circle1 myCircle = new Circle1(5.0);

6 System.out.println("The area of the circle of radius "

7 + myCircle.radius + " is " + myCircle.getArea());

10 Circle1 yourCircle = new Circle1();

11 System.out.println("The area of the circle of radius "

12 + yourCircle.radius + " is " + yourCircle.getArea());

14 // Modify circle radius

15 yourCircle.radius = 100;

16 System.out.println("The area of the circle of radius "

17 + yourCircle.radius + " is " + yourCircle.getArea());

18 }

20 double radius;

23 Circle1()

{

24 radius = 1.0;

25 }

28 Circle1(double newRadius)

{

29 radius = newRadius;

30 }

33 double getArea()

{

34 return radius \* radius \* Math.PI;

35 }

36 }

This demonstrates that you can test a class by simply adding a main method in the same class.

* Recall that you use
  + Math.methodName(arguments) (e.g., Math.pow(3, 2.5)) to invoke a method in the Math class.
  + Can you invoke
    - getArea() using Circle1.getArea()?

The answer is no.

All the methods in the **Math class are static methods**, which are defined using the static keyword. However, **getArea()** **is an instance method**, and thus **non-static**. It **must be invoked from an object** using objectRefVar.methodName(arguments)

(e.g., myCircle.getArea()).

#### 7.4.4. 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; // 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'

}

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

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.

The following code displays the default values of data fields name, age, isScienceMajor, and gender for a Student object:

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);

}

}

The following code has a compilation error because local variables x and y are not initialized:

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);

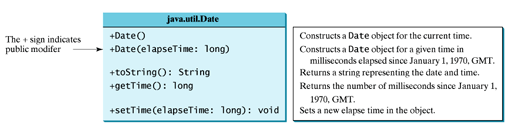
System.out.println("y is " + y);

}

}

**Classes in the Java library**

**7.5.1. The Date Class of the java.util package**



java.util.Date date = new jave.util.Date();

System.out.println("The elapsed time since Jan 1, 1970 is " +

date.getTime() + " milliseconds");

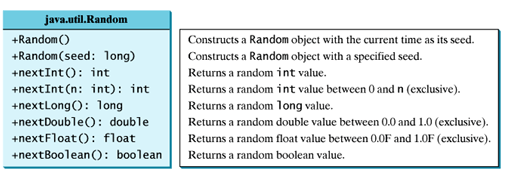
System.out.println(date.toString());

displays the output like this:

The elapse time since Jan 1, 1970 is 1100547210284 milliseconds

Mon Nov 15 14:33:30 EST 2004

7.5.2. The Random Class of the java.util package

****

If two Random objects have the same seed, they will generate identical sequences of numbers. For example:

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) + " ");

The code generates the same sequence of random int values:

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

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

### 7.6. Static Variables, Constants, and Methods

An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. For example, suppose that you create the following objects:

Circle circle1 = new Circle();

Circle circle2 = new Circle(5);

The radius in circle1 is independent of the radius in circle2, and is stored in a different memory location. Changes made to circle1's radius do not affect circle2's radius, and vice versa.

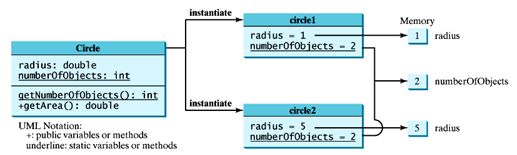
* If you want all the instances of a class to share data, use *static variables*.
* Static variables store values for the variables in a common memory location.
* Java supports static methods as well as static variables. 
  + *Static methods* can be called without creating an instance of the class.
* Let us modify the Circle class by adding a static variable numberOfObjects to count the number of circle objects created.
  + When the first object of this class is created, numberOfObjects is 1.
  + When the second object is created, numberOfObjects becomes 2.

**Listing 7.3. Circle2.java**

|  |
| --- |
| 1 public class Circle2 {  2 /\*\* The radius of the circle \*/  3 double radius;  5 /\*\* The number of the objects created \*/  6 static int numberOfObjects = 0;  8 /\*\* Construct a circle with radius 1 \*/  9 Circle2() {  10 radius = 1.0;  11 numberOfObjects++;  12 }  14 /\*\* Construct a circle with a specified radius \*/  15 Circle2(double newRadius) {  16 radius = newRadius;  17 numberOfObjects++;  18 }  20 /\*\* Return numberOfObjects \*/  21 static int getNumberOfObjects() {  22 return numberOfObjects;  23 }  25 /\*\* Return the area of this circle \*/  26 double getArea() {  27 return radius \* radius \* Math.PI;  28 }  29 } |

The Circle class defines

* the instance variable radius and
* the static variable numberOfObjects,
* the instance methods getRadius, setRadius, and getArea, and
* the static method getNumberOfObjects. (Note that static variables and functions are underlined in the UML class diagram.)

****

* Other examples of static methods are
  + showMessageDialog and showInputDialog in the JOptionPane class, and
  + all the methods in the Math class.
* Static methods (e.g., getNumberOfObjects()) and static data (e.g., numberOfObjects) can be accessed from
  + a reference variable or
  + from their class name.

The program in [Listing 7.4](mk:@MSITStore:F:\Java\Java%20Reference\Prentice.Hall.Introduction.to.Java.Programming.Comprehensive.Version.6th.Edition.ebook-LRN(1).chm::/7.6.%20Static%20Variables,%20Constants,%20and%20Methods.htm#ch07ex04) demonstrates how to use instance and static variables and methods

##### Listing 7.4. TestCircle2.java

|  |
| --- |
| 1 public class TestCircle2 {  2 /\*\* Main method \*/  3 public static void main(String[] args) {  4 // Create c1  5 Circle2 c1 = new Circle2();  7 // Display c1 BEFORE c2 is created  8 System.out.println("Before creating c2");  9 System.out.println("c1 is : radius (" + c1.radius +  10 ") and number of Circle objects (" +  11 c1.numberOfObjects + ")");  13 // Create c2  14 Circle2 c2 = new Circle2(5);  16 // Change the radius in c1  17 c1.radius = 9;  19 // Display c1 and c2 AFTER c2 was created  20 System.out.println("\nAfter creating c2 and modifying " +  21 "c1's radius to 9");  22 System.out.println("c1 is : radius (" + c1.radius +  23 ") and number of Circle objects (" +  24 c1.numberOfObjects + ")");  25 System.out.println("c2 is : radius (" + c2.radius +  26 ") and number of Circle objects (" +  27 c2.numberOfObjects +")");  28 }  29 } |

Note that:

* c1.numberOfObjects and c2.numberOfObjects could be replaced by Circle2.numberOfObjects. This improves readability.
* You can also replace Circle2.numberOfObjects by Circle2.getNumberOfObjects().
* The imported data and methods can be referenced or called without specifying a class.
  + For example, you can use PI (instead of Math.PI),
  + and random() (instead of Math.random()), if you have the following import statement in the class:

import static java.lang.Math.\*;

* Note Again that
  + **Static variables and methods** can be used from instance or static methods in the class
  + But: instance variables and methods can only be used from instance methods, **not from static methods**

|  |
| --- |
| Thus the code given below would be wrong. |

public class Foo {

int i = 5;

static int k = 2;

public **static** void main(String[] args)

{

**int j = i; // Wrong because i is an instance variable**

**m1(); // Wrong because m1() is an instance method**

}

public void m1()

**{**

**// Correct since instance and static**

**variables and methods**

**// can be used in an instance method**

**i = i + k + m2(i, k);**

**}**

public static int m2(int i, int j)

{

return (int)(Math.pow(i, j));

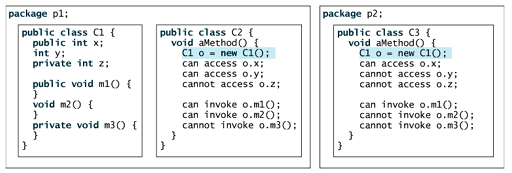
}

}

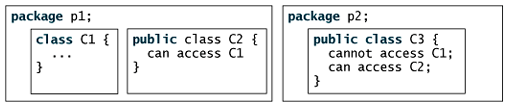
### 7.7. Visibility Modifiers

Java provides several modifiers that control access to data fields, methods, and classes.

* public makes classes, methods, and data fields accessible from any class.
* private makes methods and data fields accessible only from within its own class.
* If public or private is not 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*.



If a class is not declared public, it can only be accessed within the same package



* Visibility modifiers are used only for the members of the class, not local variables inside the methods

### 7.8. Data Field Encapsulation

The data fields radius and numberOfObjects in the Circle2 can be modified directly (e.g., myCircle.radius = 5 or Circle2.numberOfObjects = 10).

This is not a good practice for two reasons:

* First, data may be tampered.
  + For example, numberOfObjects is to count the number of objects created, but it may be set to an arbitrary value (e.g., Circle2. numberOfObjects = 10).
* Second, it makes the class difficult to maintain and vulnerable to bugs.
  + Suppose you want to modify the Circle2 class to ensure that the radius is non-negative after other programs have already used the class.
    - So, you have to change not only the Circle2 class, but also the programs that use the Circle2 class.
* To prevent direct modifications of properties, you should declare the field private, using the private modifier.
  + This is known as *data field encapsulation*.

But when a client needs to retrieve and modify a data field,

* make a private data field accessible by providing a get method to return the value of the data field.
* To enable a private data field to be updated, provide a set method to set a new value.

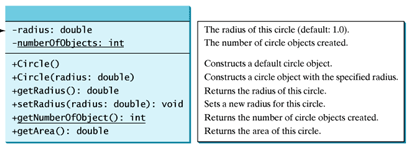
A get method has the following signature:

public returnType getPropertyName()

A set method has the following signature:

public void setPropertyName(dataType propertyValue)

Let us create a new circle class with a private data field radius and its associated accessor and mutator methods.



##### Listing 7.5. Circle3.java

|  |
| --- |
| 1 public class Circle3 {  2 /\*\* The radius of the circle \*/  3 private double radius = 1;  5 /\*\* The number of the objects created \*/  6 private static int numberOfObjects = 0;  8 /\*\* Construct a circle with radius 1 \*/  9 public Circle3() {  10 numberOfObjects++;  11 }  13 /\*\* Construct a circle with a specified radius \*/  14 public Circle3(double newRadius) {  15 radius = newRadius;  16 numberOfObjects++;  17 }  19 /\*\* Return radius \*/  20 public double getRadius() {  21 return radius;  22 }  24 /\*\* Set a new radius \*/  25 public void setRadius(double newRadius) {  26 radius = (newRadius >= 0) ? newRadius : 0;  27 }  29 /\*\* Return numberOfObjects \*/  30 public static int getNumberOfObjects() {  31 return numberOfObjects;  32 }  34 /\*\* Return the area of this circle \*/  35 public double getArea() {  36 return radius \* radius \* Math.PI;  37 }  38 } |

* Since these methods are the only ways to read and modify radius, you have total control over how the radius property is accessed. If you have to change the implementation of these methods, you need not change the client programs. This makes the class easy to maintain.

Here is a client program that uses the Circle3 class to create a Circle3 object and modifies the radius using the setRadius method.

|  |
| --- |
| 1 // TestCircle3.java: Demonstrate private modifier |
| 2 public class TestCircle3 { |
| 3 /\*\* Main method \*/ |
| 4 public static void main(String[] args) { |
| 5 // Create a Circle with radius 5.0 |
| 6 Circle myCircle = new Circle(5.0); |
| 7 System.out.println("The area of the circle of radius " |
| 8 + myCircle.getRadius() + " is " + myCircle.getArea()); |
| 9 |
| 10 // Increase myCircle's radius by 10% |
| 11 myCircle.setRadius(myCircle.getRadius() \* 1.1); |
| 12 System.out.println("The area of the circle of radius " |
| 13 + myCircle.getRadius() + " is " + myCircle.getArea()); |
| 14 } |
| 15 } |

**Can you use myCircle.radius in the client program?no**

Since numberOfObjects is private, it cannot be modified. This prevents tampering. For example, the user cannot set numberOfObjects to 100. The only way to make it 100 is to create one hundred objects of the Circle3 class.

### 7.10. Passing Objects to Methods

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:

public class TestPassObject {

public static void main(String[] args) {

Circle3 myCircle = new Circle3(5.0);

printCircle(myCircle);

}

public static void printCircle(Circle3 c) {

System.out.println("The area of the circle of radius "

+ c.getRadius() + " is " + c.getArea());

}

}

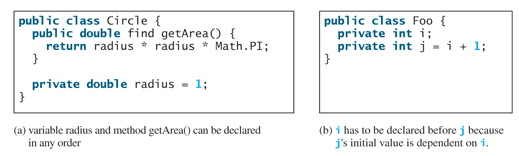
Let us demonstrate the difference between passing a primitive type value and passing a reference value with the program in [Listing 7.6](mk:@MSITStore:F:\Java\Java%20Reference\Prentice.Hall.Introduction.to.Java.Programming.Comprehensive.Version.6th.Edition.ebook-LRN(1).chm::/7.10.%20Passing%20Objects%20to%20Methods.htm#ch07ex06).

##### Listing 7.6. TestPassObject.java

|  |
| --- |
| 1 public class TestPassObject {  2 /\*\* Main method \*/  3 public static void main(String[] args) {  4 // Create a Circle object with radius 1  5 Circle3 myCircle = new Circle3(1);  6  7 // Print areas for radius 1, 2, 3, 4, and 5.  8 int n = 5;  9 printAreas(myCircle , n);  10  11 // See myCircle.radius and times  12 System.out.println("\n" + "Radius is " + myCircle.getRadius());  13 System.out.println("n is " + n);  14 }  15    16 /\*\* Print a table of areas for radius \*/  17 public static void printAreas(Circle3 c, int times) {  18 System.out.println("Radius \t\tArea");  19 while (times >= 1) {  20 System.out.println(c.getRadius() + "\t\t" + c.getArea());  21 c.setRadius(c.getRadius() + 1);  22 times——;  23 }  24 }  25 } |

### 7.11. The Scope of Variables

* 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 be declared in any order in the class.
* The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first, as shown below.



* You can declare a class's variable only once, but you can declare the same variable name in a method many times in different non-nesting blocks.
* If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is **hidden**.
* For example, in the following program, x is defined as an instance variable and as a local variable in the method.

class Foo {

int x = 0; // instance variable

int y = 0;

Foo() {

}

void p() {

int x = 1; // local variable

System.out.println("x = " + x);

System.out.println("y = " + y);

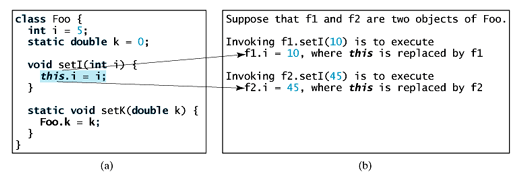
}

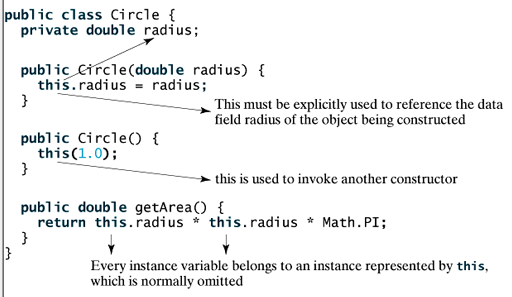
}

What is the printout for f.p(), where f is an instance of Foo? The printout for f.p() is 1 for x and 0 for y. Here is why:

* x is declared as a data field with the initial value of 0 in the class, but is also defined in the method p() with an initial value of 1. The latter x is referenced in the System.out.println statement.
* y is declared outside the method p(), but is accessible inside it.

**7.12. The this Keyword**

* Sometimes you need to reference a class's hidden variable in a method.
  + For example, a property name is often used as the parameter name in a set method for the property.
  + In this case, you need to reference the hidden property name in the method in order to set a new value to it.
  + A hidden static variable can be accessed simply by using the ClassName.StaticVariable reference.
  + A hidden instance variable however, can be accessed by using the keyword **this**
* 
* The line this.i = i means "assign the value of parameter i to the data field i of the calling object."
* The keyword this serves as a proxy for the object that invokes the instance method setI
* The line Foo.k = k means that the value in parameter k is assigned to the static data field k of the class, which is shared by all the objects of the class.
* The keyword this can also be used inside a constructor to invoke another constructor of the same class. For example, you can redefine the Circle class as follows

****

The line this(1.0) invokes the constructor with a double value argument in the class.

### 7.13. Array of Objects

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

Circle[] circleArray = new Circle[10];

To initialize the circleArray, you can use a for loop like this one:

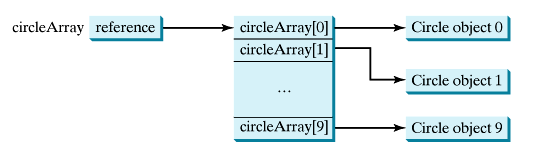
for (int i = 0; i < circleArray.length; i++) {

circleArray[i] = new Circle();

}

An array of objects is actually an array of reference variables. So invoking circleArray[1].getArea() involves two levels of referencing, as shown below.

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

****

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

[Listing 7.7](mk:@MSITStore:F:\Java\Java%20Reference\Prentice.Hall.Introduction.to.Java.Programming.Comprehensive.Version.6th.Edition.ebook-LRN(1).chm::/7.13.%20Array%20of%20Objects.htm#ch07ex07) demonstrates how to use an array of objects

|  |
| --- |
| **1 public class TotalArea {**  **2 /\*\* Main method \*/**  **3 public static void main(String[] args) {**    **5 Circle3[] circleArray;**  **8 circleArray = ; createCircleArray()**    **11 printCircleArray(circleArray);**  **12 }**  **15 public static Circle3[] createCircleArray() {**  **16 Circle3[] circleArray = new Circle3[10];**  **17**  **18 for (int i = 0; i < circleArray.length; i++) {**  **19 circleArray[i] = new Circle3(Math.random() \* 100);**  **20 }**  **23 return circleArray;**  **24 }**  **25**  **27 public static void printCircleArray**  **28 (Circle3[] circleArray) {**  **29 System.out.println("Radius\t\t\t\t" + "Area");**  **30 for (int i = 0; i < circleArray.length; i++) {**  **31 System.out.print(circleArray[i].getRadius() + "\t\t" +**  **32 circleArray[i].getArea() + '\n');**  **33 }**  **34**  **35 System.out.println("–––––––––––––—");**  **38 System.out.println("The total areas of circles is \t" +**  **39 sum(circleArray));**  **40 }**  **41**  **43 public static double sum(Circle3[] circleArray) {**  **45 double sum = 0;**  **48 for (int i = 0; i < circleArray.length; i++)**  **49 sum += circleArray[i].getArea();**  **51 return sum;**  **52 }**  **53 }** |

### 7.14. Class Abstraction and Encapsulation

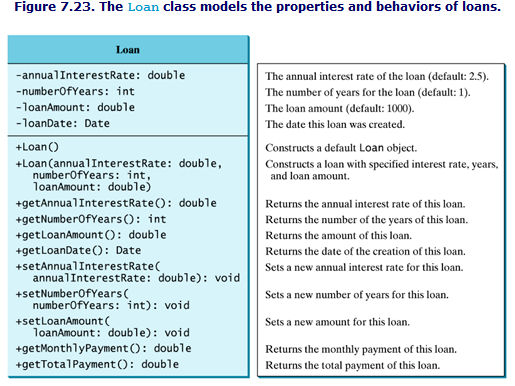
* Java provides many levels of abstraction.
* *Class abstraction* is the separation of class implementation from the use of a class.
* The creator of a class provides a description of the class and lets the user know how the class can be used.
* The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*.
* The user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is known as *class encapsulation*.



The internal implementation is encapsulated and hidden from you. Eg. You can build a computer without knowing how a component is implemented

* Another Example:
  + Consider getting a loan.
  + A specific loan can be viewed as an object of a Loan class.
    - Interest rate, loan amount, and loan period are its data properties, and
    - computing monthly payment and total payment are its methods.
  + When you buy a car,
    - a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period.
    - You can then use the methods to find the monthly payment and total payment of your loan.
    - As a user of the Loan class, you don't need to know how these methods are implemented

**7.15. Case Study: The Loan Class: First Create The Class Contract**

****

##### Listing 7.8. TestLoanClass.java

|  |
| --- |
| 1 import javax.swing.JOptionPane;  3 public class TestLoanClass {  5 public static void main(String[] args) {  7 String annualInterestRateString = JOptionPane.showInputDialog(  8 "Enter yearly interest rate, for example 8.25:");  11 double annualInterestRate =  12 Double.parseDouble(annualInterestRateString);  15 String numberOfYearsString = JOptionPane.showInputDialog(  16 "Enter number of years as an integer, \nfor example 5:");  19 int numberOfYears = Integer.parseInt(numberOfYearsString);  22 String loanString = JOptionPane.showInputDialog(  23 "Enter loan amount, for example 120000.95:");  26 double loanAmount = Double.parseDouble(loanString);  28 // Create Loan object  29 Loan loan =  30 new Loan(annualInterestRate, numberOfYears, loanAmount);  32 // Format to keep two digits after the decimal point  33 double monthlyPayment =  34 (int)(loan.getMonthlyPayment() \* 100) / 100.0;  35 double totalPayment =  36 (int)(loan.getTotalPayment() \* 100) / 100.0;  38 // Display results  39 String output = "The loan was created on " +  40 loan.getLoanDate().toString() + "\nThe monthly payment is " +  41 monthlyPayment + "\nThe total payment is " + totalPayment;  42 JOptionPane.showMessageDialog(null, output);  43 }  44 } |

##### Listing 7.9. Loan.java

|  |
| --- |
| **1 public class Loan {**  **2 private double annualInterestRate;**  **3 private int numberOfYears;**  **4 private double loanAmount;**  **5 private java.util.Date loanDate;**  **8 public Loan()**  **{**  **9 this(7.5, 30, 100000);**  **10 }**  **15 public Loan(double annualInterestRate, int numberOfYears,**  **16 double loanAmount)**  **{**  **17 this.annualInterestRate = annualInterestRate;**  **18 this.numberOfYears = numberOfYears;**  **19 this.loanAmount = loanAmount;**  **20 loanDate = new java.util.Date();**  **21 }**  **24 public double getAnnualInterestRate()**  **{**  **25 return annualInterestRate;**  **26 }**  **29 public void setAnnualInterestRate(double annualInterestRate)**  **{**  **30 this.annualInterestRate = annualInterestRate;**  **31 }**  **34 public int getNumberOfYears()**  **{**  **35 return numberOfYears;**  **36 }**  **39 public void setNumberOfYears(int numberOfYears)**  **{**  **40 this.numberOfYears = numberOfYears;**  **41 }**  **44 public double getLoanAmount()**  **{**  **45 return loanAmount;**  **46 }**  **49 public void setLoanAmount(double loanAmount)**  **{**  **50 this.loanAmount = loanAmount;**  **51 }**  **54 public double getMonthlyPayment()**  **{**  **55 double monthlyInterestRate = annualInterestRate / 1200;**  **56 return loanAmount \* monthlyInterestRate / (1 -**  **57 (Math.pow(1 / (1 + monthlyInterestRate), numberOfYears \***  **12)));**  **58 }**  **61 public double getTotalPayment()**  **{**  **62 return getMonthlyPayment() \* numberOfYears \* 12;**  **63 }**  **66 public java.util.Date getLoanDate()**  **{**  **67 return loanDate;**  **68 }**  **69 }** |