

Chapter 9 Objects and Classes

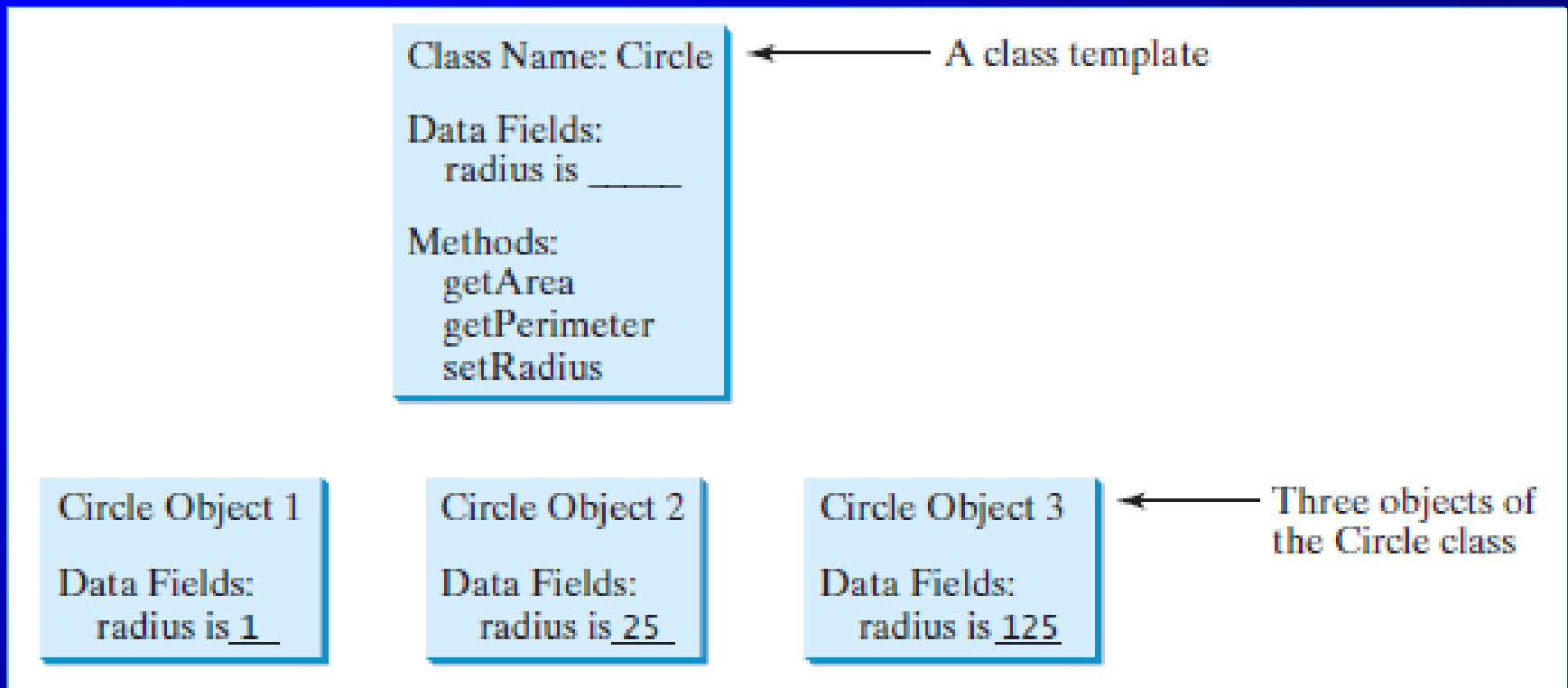


面向对象编程的基本概念

- 面向对象编程（Object-oriented programming, OOP）是以对象为基本单位来搭建程序的编程方法。术语对象（*object*）指的是在真实世界中能够被明确区分的实体（*entity*），例如学生，桌子，圆，按钮。
- 对象具有唯一的标识，状态和行为。对象状态由一组数据域 *data fields*（或属性 *properties*）构成，对象的行为 *behavior* 则是由一组方法构成。



举一个圆对象作为例子



一个对象有状态和行为。如上图，状态就是半径，行为则有三个：求面积，求周长，设置半径。作为类模版Circle而言，它只是一个概念，仅作为定义使用。下面三个才是真正的对象。

类

- Java用class来定义对象的概念，是同类型对象的总称。它本身不是实体，只是一个笼统的概念。例如学生这个概念，只有具体到某一个学生上，才能称为实体。
- Java使用成员变量来定义对象的数据域，使用成员方法来定义对象的行为。
- 此外，Java的类还提供了一些特殊的方法，称为构造方法，用来创建具体的对象实体。



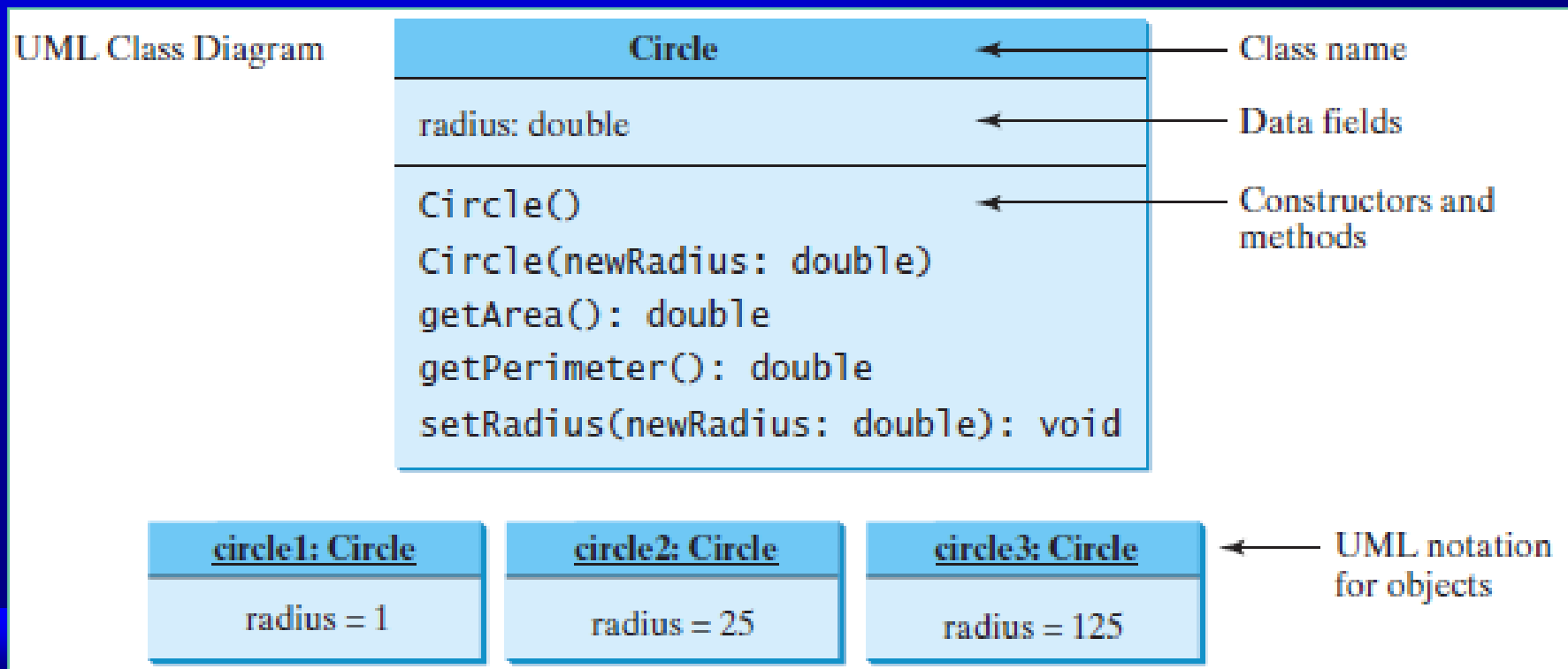
圆对象的Java编程实现

```
class Circle {  
    /** The radius of this circle */  
    double radius = 1;  ← Data field  
  
    /** Construct a circle object */  
    Circle() {  
    }  
  
    /** Construct a circle object */  
    Circle(double newRadius) {  
        radius = newRadius;  
    }  
    ← Constructors  
  
    /** Return the area of this circle */  
    double getArea() {  
        return radius * radius * Math.PI;  
    }  
  
    /** Return the perimeter of this circle */  
    double getPerimeter() {  
        return 2 * radius * Math.PI;  
    }  
    ← Method  
  
    /** Set new radius for this circle */  
    double setRadius(double newRadius) {  
        radius = newRadius;  
    }  
}
```



用Unified Modeling Language (UML)表示Circle类

- ☞ UML是一种面向对象的建模语言，它运用统一的、标准化的标记和定义实现对软件系统进行面向对象的描述和建模。



例题：定义一个类，并创建对象

- 👉 学习目标：如何创建一个对象，访问对象的数据，使用对象的方法。
- 👉 先看源代码：



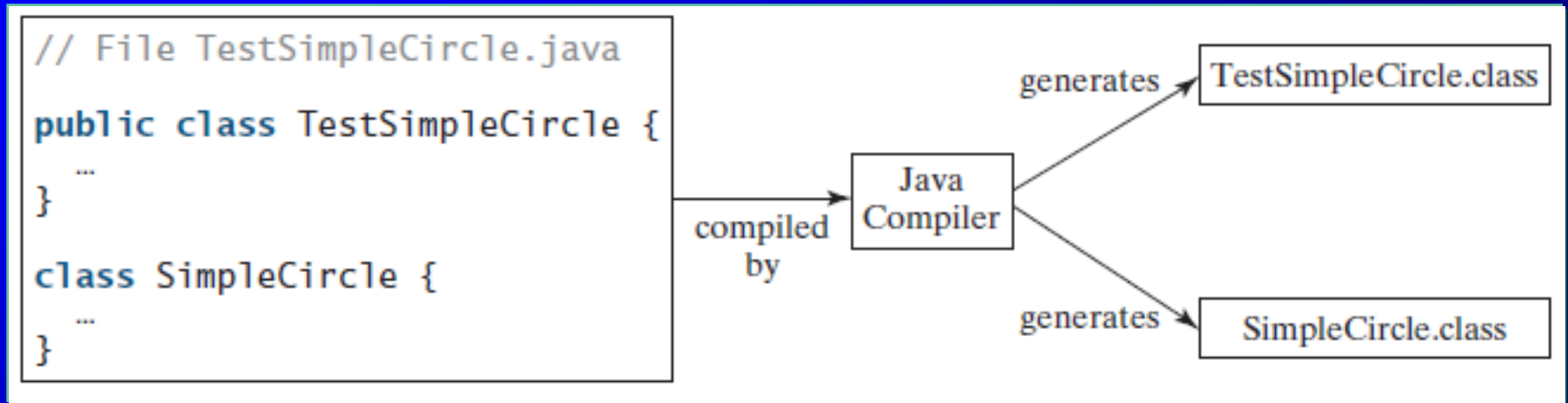
```
1 public class TestSimpleCircle {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a circle with radius 1
5         SimpleCircle circle1 = new SimpleCircle();
6         System.out.println("The area of the circle of radius "
7             + circle1.radius + " is " + circle1.getArea());
8
9         // Create a circle with radius 25
10        SimpleCircle circle2 = new SimpleCircle(25);
11        System.out.println("The area of the circle of radius "
12            + circle2.radius + " is " + circle2.getArea());
13
14        // Create a circle with radius 125
15        SimpleCircle circle3 = new SimpleCircle(125);
16        System.out.println("The area of the circle of radius "
17            + circle3.radius + " is " + circle3.getArea());
18
19        // Modify circle radius
20        circle2.radius = 100; // or circle2.setRadius(100)
21        System.out.println("The area of the circle of radius "
22            + circle2.radius + " is " + circle2.getArea());
23    }
```



```
24 }
25
26 // Define the circle class with two constructors
27 class SimpleCircle {
28     double radius;
29
30     /** Construct a circle with radius 1 */
31     SimpleCircle() {
32         radius = 1;
33     }
34
35     /** Construct a circle with a specified radius */
36     SimpleCircle(double newRadius) {
37         radius = newRadius;
38     }
39
40     /** Return the area of this circle */
41     double getArea() {
42         return radius * radius * Math.PI;
43     }
44
45     /** Return the perimeter of this circle */
46     double getPerimeter() {
47         return 2 * radius * Math.PI;
48     }
49
50     /** Set a new radius for this circle */
51     void setRadius(double newRadius) {
52         radius = newRadius;
53     }
54 }
```



➡ 程序编译后的结果是：



➡ 程序的执行结果是：

```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```

几点说明

- ☞ Java可以在一个源文件中放置多个类，例如上述例子就有两个类。但是只能有一个类是public修饰的，这个类叫主类，main函数总是被放在这个类中。
- ☞ TestSimpleCircle类是一个测试类，它的的功能是用于测试SimpleCircle类，所以它本身除了main函数之外，没有再定义自己的成员变量和方法。
- ☞ 其实一个类也可以自己带一个main函数来测试自己，例如改写上一个例子为：

```

1 public class SimpleCircle {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a circle with radius 1
5         SimpleCircle circle1 = new SimpleCircle();
6         System.out.println("The area of the circle of radius "
7             + circle1.radius + " is " + circle1.getArea());
8
9         // Create a circle with radius 25
10        SimpleCircle circle2 = new SimpleCircle(25);
11        System.out.println("The area of the circle of radius "
12            + circle2.radius + " is " + circle2.getArea());
13
14        // Create a circle with radius 125
15        SimpleCircle circle3 = new SimpleCircle(125);
16        System.out.println("The area of the circle of radius "
17            + circle3.radius + " is " + circle3.getArea());
18
19        // Modify circle radius
20        circle2.radius = 100;
21        System.out.println("The area of the circle of radius "
22            + circle2.radius + " is " + circle2.getArea());
23    }
24
25    double radius;
26
27    /** Construct a circle with radius 1 */
28    SimpleCircle() {
29        radius = 1;
30    }

```

 左边的代码不全，不过已经可以看到改动的思路：先将SimpleCircle类设为主类，然后再补一个main方法，当然，源文件需要重命名。

几个要点复习

☞ 创建一个对象，用new关键字，例如：

- SimpleCircle circle1 = **new** SimpleCircle();
- SimpleCircle circle2 = **new** SimpleCircle(25);

☞ 创建对象之后，可以使用.访问对象成员和对象方法，例如：

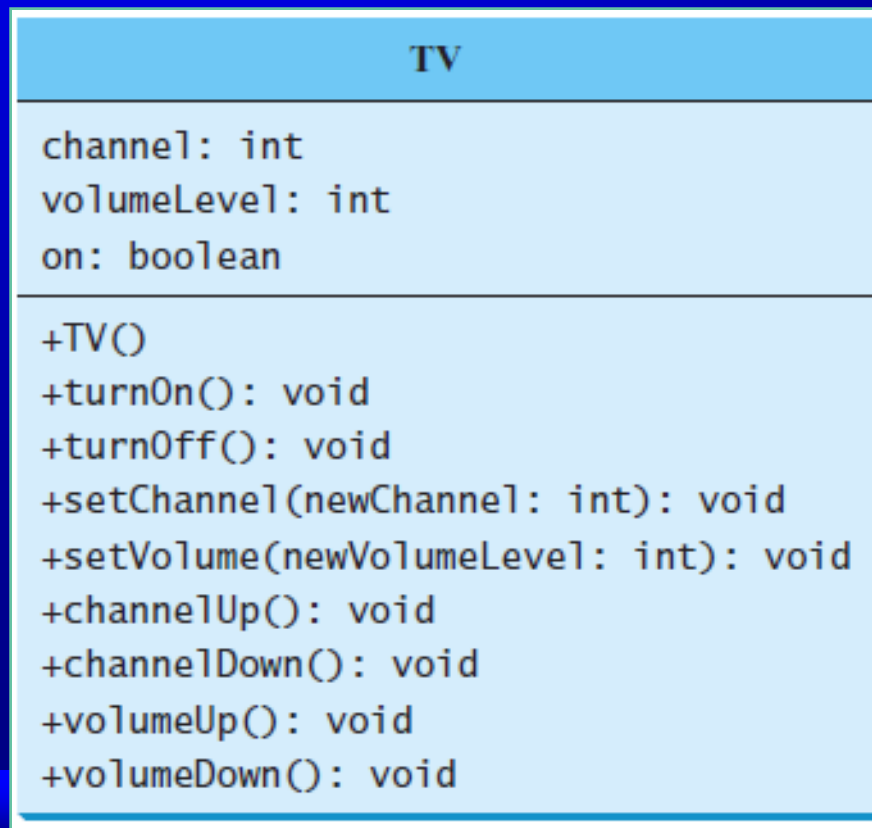
- circle1.radius
- circle2.getArea()

☞ 对象必须**先创建后使用**，没有经过new的对象是空对象null，无法使用。



再来个例子巩固一下

👉 下面是一个电视类的UML图:



```
1 public class TV {
2     int channel = 1; // Default channel is 1
3     int volumeLevel = 1; // Default volume level is 1
4     boolean on = false; // TV is off
5
6     public TV() {
7     }
8
9     public void turnOn() {
10         on = true;
11     }
12
13     public void turnOff() {
14         on = false;
15     }
16
17     public void setChannel(int newChannel) {
18         if (on && newChannel >= 1 && newChannel <= 120)
19             channel = newChannel;
20     }
21 }
```

```
22     public void setVolume(int newVolumeLevel) {
23         if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
24             volumeLevel = newVolumeLevel;
25     }
26
27     public void channelUp() {
28         if (on && channel < 120)
29             channel++;
30     }
31
32     public void channelDown() {
33         if (on && channel > 1)
34             channel--;
35     }
36
37     public void volumeUp() {
38         if (on && volumeLevel < 7)
39             volumeLevel++;
40     }
41
42     public void volumeDown() {
43         if (on && volumeLevel > 1)
44             volumeLevel--;
45     }
46 }
```


测试类及运行结果

```
1 public class TestTV {  
2     public static void main(String[] args) {  
3         TV tv1 = new TV();  
4         tv1.turnOn();  
5         tv1.setChannel(30);  
6         tv1.setVolume(3);  
7  
8         TV tv2 = new TV();  
9         tv2.turnOn();  
10        tv2.channelUp();  
11        tv2.channelUp();  
12        tv2.volumeUp();  
13  
14        System.out.println("tv1's channel is " + tv1.channel  
15            + " and volume level is " + tv1.volumeLevel);  
16        System.out.println("tv2's channel is " + tv2.channel  
17            + " and volume level is " + tv2.volumeLevel);  
18    }  
19 }
```

tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2

构造方法

```
Circle() {  
}
```

构造方法是一种特殊的方法，当对象被创建的时候，构造方法会被自动调用。

```
Circle(double newRadius) {  
    radius = newRadius;  
}
```



构造方法的写法

- 构造方法的名字必须和类的名字完全相同，包括大小写。
- 构造方法不能有返回类型，甚至连void都不能写，所以不可能出现return语句。
- 构造方法会在new的时候被自动调用，所以它特别适合用来做对象初始化之类的工作。
- 构造方法经常被重载，所以new的时候可以利用传入的参数，调用不同的构造方法。

利用构造方法创建对象

```
new ClassName();
```

例如:

```
new Circle();
```

```
new Circle(5.0);
```



默认构造方法

如果一个类没有显式定义任何构造方法，Java 会隐含定义一个**没有形参，方法体为空**的构造方法。这个方法称为默认构造方法。例如下面这个类，框出来的部分，如果你不写，Java 也会默认偷偷补上（当然不会改动到源代码）：

```
class Circle {  
    Circle() {  
    }  
}
```



对象引用的变量声明

为了引用一个对象，必须将对象赋值给一个引用变量。声明一个对象的引用很简单，语法如下：

```
ClassName objectRefVar;
```

例如：

```
Circle myCircle;
```

这个用法其实和以前的变量声明没有区别，例如以前是这样声明一个整型变量：`int i;`

只不过当变量类型变为对象之后，我们把这样的变量叫做**引用变量**，以和普通变量相区别。

声明并创建对象

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

将对象的引用
赋值给变量

创建对象

例如:

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



访问对象

☞ 访问对象的数据:

```
objectRefVar.data
```

例如: `myCircle.radius`

☞ 调用对象的方法:

```
objectRefVar.methodName(arguments)
```

例如: `myCircle.getArea()`



单步执行一下

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

声明对象引用变量

myCircle

myCircle

null



单步执行一下

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle null

: Circle
radius: 5.0

创建一个对象



单步执行一下

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

将对象的引用赋值给 myCircle

myCircle

引用值

: Circle

radius: 5.0



单步执行一下

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle

引用值

: Circle

radius: 5.0

yourCircle

null

声明yourCircle

单步执行一下

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle

引用值

: Circle

radius: 5.0

yourCircle

null

创建一个新
Circle对象

: Circle

radius: 0.0

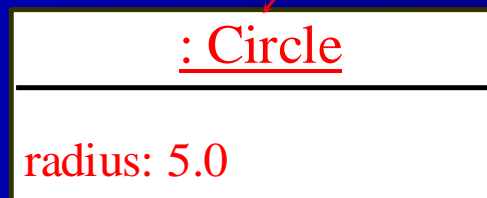
单步执行一下

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

```
Circle yourCircle = new Circle();
```

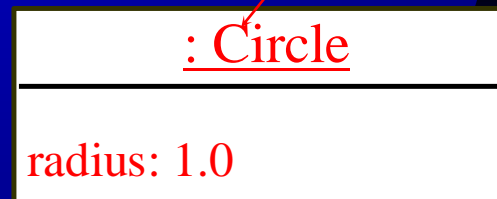
```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle 引用值

将对象的引用赋值给yourCircle



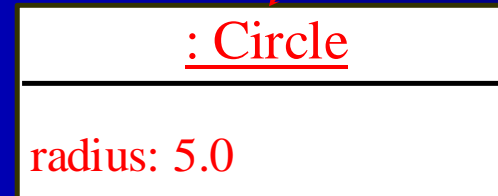
单步执行一下

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

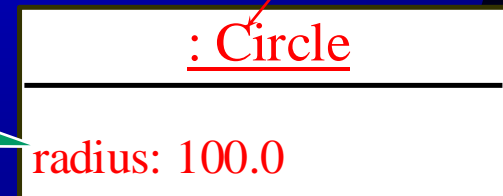
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle reference value



修改yourCircle的
radius成员变量

注意

我们一直都是这样调用数学库的方法：

Math.methodName(arguments) (如, Math.pow(3, 2.5))

那么，能否直接使用Circle1.getArea() 来调用 getArea() 方法？答案是：**不行**。原因在于，在本章之前，我们定义的所有方法，前面都有一个修饰词static，这就是所谓的**静态方法**。显然这里的 getArea() 是一个非静态的方法，所以只能通过对象的引用来调用：

objectRefVar.methodName(arguments) (如, myCircle.getArea())

关于静态变量，静态常量和静态方法的更多讨论，后续章节我们再展开。



数据域

数据域也可以有引用类型，例如，下面这个Student类就包含一个name的成员，它是String类型。

String其实是一个类，所以name在这里的确切表述，是一个**引用变量**，当然你简单把它看成变量也行。

所以，可以看出类的成员是没什么限制的，可以是普通变量，也可以是某个类的引用，甚至是它自己的引用。

```
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'  
}
```

null

如果一个数据域是引用类型，并且未被初始化，则Java会自动为将其初始化为空值 **null**，表示该引用尚未指向任何一个具体对象。



数据域的默认值

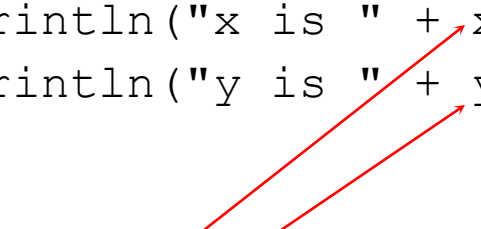
- ☞ 数据域都有默认值，引用类型的默认值是`null`，数值类型的默认值为`0`，布尔类型默认值为`false`，字符类型默认值为`'\u0000'`。
- ☞ 作为对比，Java对于局部变量或者临时变量都没有默认值，所以这些变量在被赋值之前都是**随机值**。

```
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);  
    }  
}
```

例题

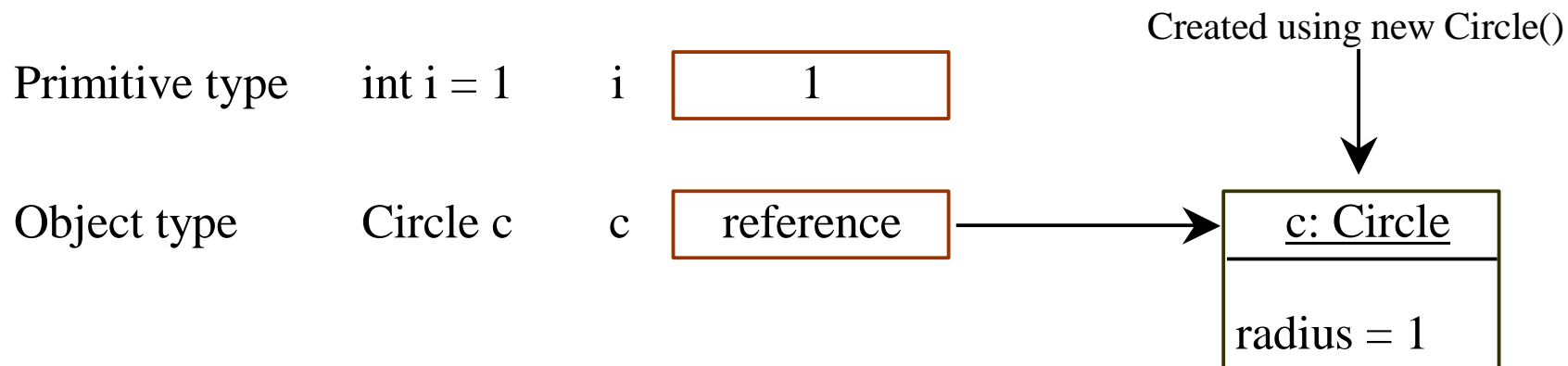
- Java的局部变量是没有默认值的。试图直接使用未初始化的局部变量，将直接导致编译错误。

```
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);  
    }  
}
```



编译直接报错：变量未初始化！

基本数据类型的变量和引用类型的变量



基本数据类型的变量和引用类型的变量，执行=时的行为不同

Primitive type assignment $i = j$

Before:

i 1

j 2

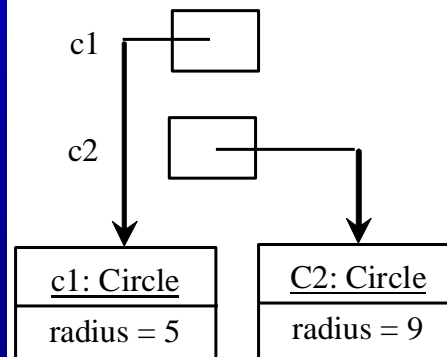
After:

i 2

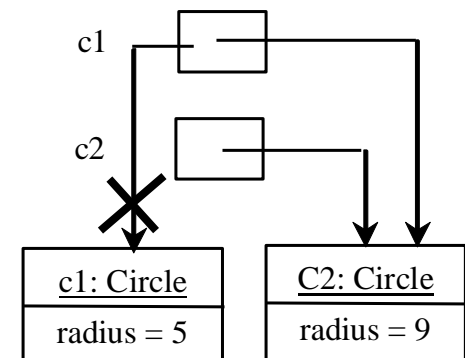
j 2

Object type assignment $c1 = c2$

Before:



After:



垃圾回收

➡ 前一页PPT中，执行 $c1 = c2$ 之后， $c1$ 和 $c2$ 指向（或者说引用）了同一个对象。这个操作导致原先 $c1$ 指向的那个对象没有被引用（即，再没有变量指向它），这时候这个对象已经无法使用，成为垃圾（garbage）。这些垃圾会占用掉一定的内存，不过它们会被JVM自动回收，所以不需要你做额外的处理。

小提示

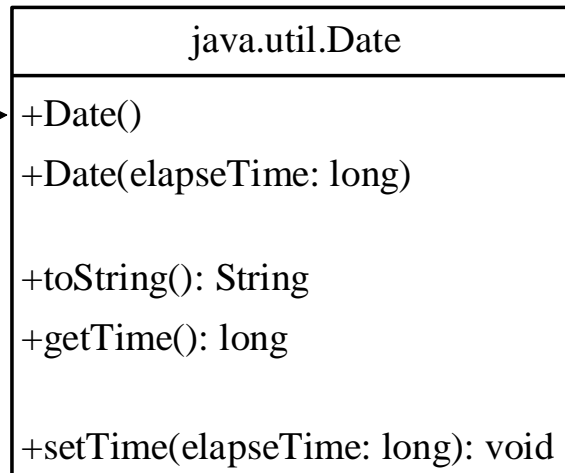
- ➡ 如果你确定不再需要一个对象，并且想要清掉这个对象所占用的内存空间，你可以简单的用null给引用这个对象的变量赋值，这样JVM就知道这个对象需要被回收。



日期类Date

☞ Java提供了一个处理日期和时间的日期类 java.util.Date 。下面是这个类的UML图。

The + sign indicates
public modifier



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.

一个例子

要打印当前时间，可以用下面的代码：

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

输出的格式类似这样：

Mon May 27 17:55:46 CST 2013



随机数类Random

除了Math.random(), Java还提供了一个随机数生成类java.util.Random。下面是这个类的UML图。

java.util.Random

```
+Random()  
+Random(seed: long)  
+nextInt(): int  
+nextInt(n: int): int  
+nextLong(): long  
+nextDouble(): double  
+nextFloat(): float  
+nextBoolean(): boolean
```

Constructs a Random object with the current time as its seed.

Constructs a Random object with a specified seed.

Returns a random int value.

Returns a random int value between 0 and n (excluding n).

Returns a random long value.

Returns a random double value between 0.0 and 1.0 (excluding 1.0).

Returns a random float value between 0.0F and 1.0F (excluding 1.0F).

Returns a random boolean value.

相同的种子点，将会生成相同的随机序列

```
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) + " ");
```



2D坐标类Point2D

☞ `javafx.geometry.Point2D`类，用来表示平面上的一个点(x, y)。下面是这个类的UML图。

`javafx.geometry.Point2D`

```
+Point2D(x: double, y: double)
+distance(x: double, y: double): double
+distance(p: Point2D): double
+getX(): double
+getY(): double
+toString(): String
```

```
Constructs a Point2D object with the specified x- and y-coordinates.
Returns the distance between this point and the specified point (x, y).
Returns the distance between this point and the specified point p.
Returns the x-coordinate from this point.
Returns the y-coordinate from this point.
Returns a string representation for the point.
```



一个例子

```
1  import java.util.Scanner;
2  import javafx.geometry.Point2D;
3
4  public class TestPoint2D {
5      public static void main(String[] args) {
6          Scanner input = new Scanner(System.in);
7
8          System.out.print("Enter point1's x-, y-coordinates: ");
9          double x1 = input.nextDouble();
10         double y1 = input.nextDouble();
11         System.out.print("Enter point2's x-, y-coordinates: ");
12         double x2 = input.nextDouble();
13         double y2 = input.nextDouble();
14
15         Point2D p1 = new Point2D(x1, y1);
16         Point2D p2 = new Point2D(x2, y2);
17         System.out.println("p1 is " + p1.toString());
18         System.out.println("p2 is " + p2.toString());
19         System.out.println("The distance between p1 and p2 is " +
20             p1.distance(p2));
21     }
22 }
```

实例变量和方法

实例变量属于某个特定的实例。不同实例的实例变量毫无关联。例如：

```
Circle c1 = new Circle(); Circle c2 = new Circle(5);
```

这两个圆的radius是实例变量，因此各自的半径是独立的。c1和c2的半径毫无关联。

实例方法需要借助某个类的实例才能被调用。

什么是实例变量和实例方法？答案很简单，类的成员中，凡是没有用static修饰的都算。



静态变量，常量和方法

静态变量和静态常量，是所有同一个类的实例共享的。

静态方法没有绑定在某个特定对象上。也就是说，它也是所有实例共享的。

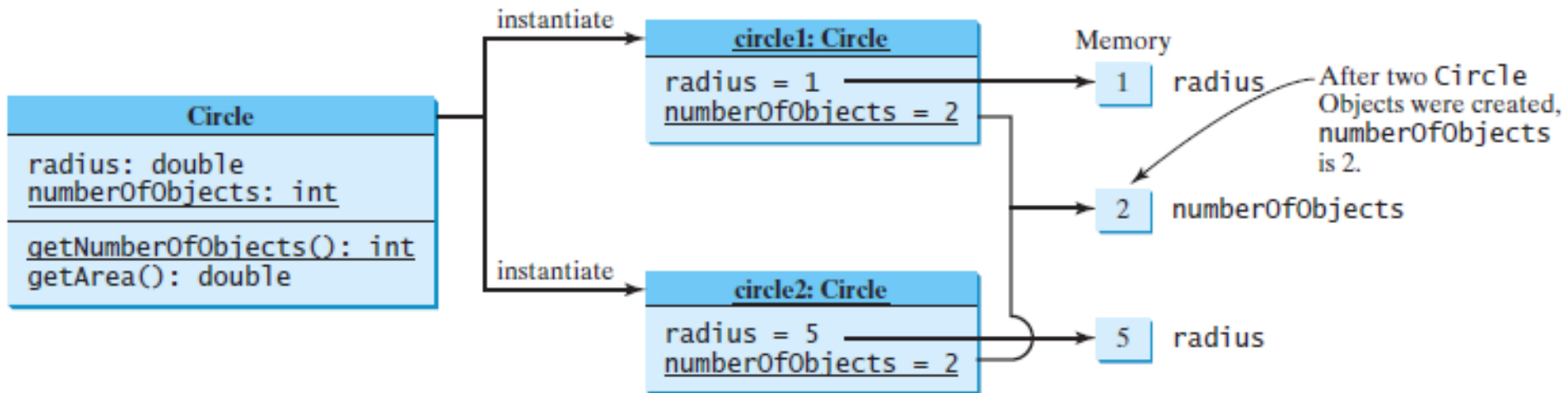
如何区分成员是否是静态的？答案是，用static修饰的就是。



静态变量，常量和方法图示

- UML图例说明： 有下划线的成员表示静态成员。可以看出， `numberOfObjects` 是多个实例共享的， `radius` 是独立的。

UML Notation:
underline: static variables or methods



实例变量的例子

在Circle的基础上，添加一个静态成员，体会一下有无static的不同之处。



```
public class CircleWithStaticMembers {  
    double radius;  
    static int numberOfObjects = 0;  
  
    CircleWithStaticMembers() {  
        radius = 1;  
        numberOfObjects++;  
    }  
    CircleWithStaticMembers(double newRadius) {  
        radius = newRadius;  
        numberOfObjects++;  
    }  
  
    static int getNumberOfObjects() {  
        return numberOfObjects;  
    }  
    double getArea() {  
        return radius * radius * Math.PI;  
    }  
}
```

```
public class TestCircleWithStaticMembers {
    public static void main(String[] args) {
        System.out.println("Before creating objects");
        System.out.println("The number of Circle objects is " +
            CircleWithStaticMembers.numberOfObjects);
        CircleWithStaticMembers c1 = new CircleWithStaticMembers();
        System.out.println("\nAfter creating c1");
        System.out.println("c1: radius (" + c1.radius +
            ") and number of Circle objects (" +
            c1.numberOfObjects+ ")");

        CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);
        c1.radius = 9;
        System.out.println("\nAfter creating c2 and modifying c1");
        System.out.println("c1: radius (" + c1.radius +
            ") and number of Circle objects (" +
            c1.numberOfObjects + ")");
        System.out.println("c2: radius (" + c2.radius +
            ") and number of Circle objects (" +
            c2.numberOfObjects+ ")");
    }
}
```

运行结果

```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

☞ 上述程序的结果说明，静态成员是共享的，其余成员则是独立的。



包（package）

- ➡ 包的作用类似于C的函数库，但是C的函数库很容易出现重名的问题，包在一定程度上解决了这个问题
- ➡ 一个包通常包含很多个功能相近的类。



创建自己的包

➡ 创建包

- `package` 包名; //此句必须是源文件的第一条语句

➡ 例如:

- `package com.example.graphics;`
- `package oop;`

➡ 一个Java源文件必定属于某个包。如果你没有使用`package`语句，这个源文件的所有类，会被自动归入一个匿名包。



包的目录映射

- ➡ 包名和目录名有一个映射规则，所以源代码必须存放在指定位置才行。假设Rectangle.java代码第一行是：
 - package com.example.graphics;
- ➡ 那么Rectangle.java的存放目录必须是：
 -\com\example\graphics\Rectangle.java
- ➡ 前面....可以是任意目录，因为package只规定了相对目录，因此只要保持\com\example\graphics\结构就行。



包的编译、运行

- ✎ 假设源代码Test.java的第一行是：package oop; 此时，Test.java必须存放在oop目录下。
- ✎ 编译这个文件，可以在oop的同一级目录运行：
 - **javac oop/Test.java**
顺利的话，oop目录下会有Test.class生成。当然，也可以在oop目录下运行**javac Test.java**
- ✎ 运行这个程序，一定要在oop的同一级目录运行（否则会出现**错误：找不到或无法加载主类**）：
 - **java oop.Test**
- ✎ 友情提醒：使用IDE开发，不需要考虑上述问题

包的命名

- ☞ Java建议包的名字取成域名的逆序，例如cn.edu.xmu，这是为了避免命名重复。
- ☞ 包之间没有嵌套关系，例如java.awt和java.awt.geom是两个完全独立的包。
- ☞ 其实上面已经提到，包名中的.最后会被映射成文件目录。因此，包java.awt是由java/awt下的java文件编译而来，包java.awt.geom则是由java/awt/geom下的文件编译而来，这是两个不同的文件夹，所以源文件之间不存在包含关系。



包的使用

➡ 导入包有两种格式

- `import 包名.类名;` //导入这个包的特定类
- `import 包名.*;` //导入这个包的所有类

➡ 假设你想使用`java.awt.event`包中的`ActionEvent`类，可以选择以下两种方式导入：

- `import java.awt.event.ActionEvent;`
- `import java.awt.event.*;`

➡ 导入后，可以这样使用`ActionEvent`类：

- `ActionEvent myEvent = new ActionEvent();`

➡ 其实不导入一个类也是可以用的，就是写起来麻烦：

- `java.awt.event.ActionEvent myEvent = new java.awt.event.ActionEvent();`



可见性修饰符

默认情况下，类的变量和方法，可以被在同一个包（`package`）中的任意类访问。

☞ `public`

类的成员能被任意包的任意类访问。

☞ `private`

类的成员仅能被类自身内部的方法所访问。如果一个属性被`private`修饰，外部的类想要访问这个属性，就只能通过`get`和`set`方法（如果这个类有提供的话）



package p1;

```
public class C1 {  
    public int x;  
    int y;  
    private int z;  
  
    public void m1() {  
    }  
    void m2() {  
    }  
    private void m3() {  
    }  
}
```

```
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();  
    }  
}
```

package p1;

```
class C1 {  
    ...  
}
```

```
public class C2 {  
    can access C1  
}
```

package p2;

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

private修饰符将访问范围限制到类自身内部；默认的（没有加任何修饰符的）情况下，访问范围是同一个包；如果是**public**，则访问范围不再局限于包（相当于没有任何限制）。

为什么需要将数据域设置为 private?

- ➡ 可以保护数据。
- ➡ 可以让类容易维护。

这就是面向对象的第一个特性：**封装性**



传递对象到方法中

- ➡ 对基本数据类型而言，传递的是值(执行形参=实参的操作)
- ➡ 对引用类型而言，传递的是对象的引用 (执行形参=实参的操作)
- ➡ 也就是说，其实Java参数传递只有一种，就是值传递（形参=实参）。只不过在上面两种情况下，对=的处理有所不同。



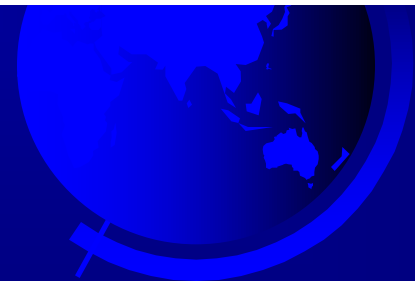
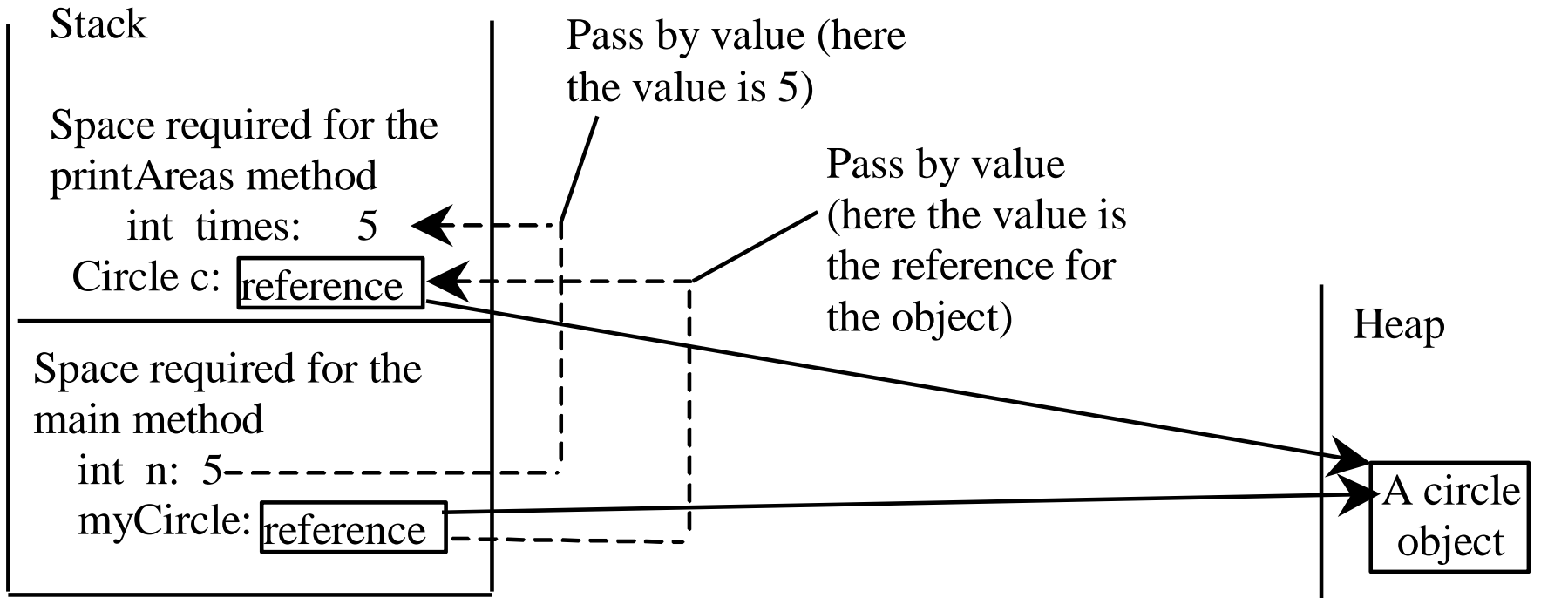
```

1 public class TestPassObject {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a Circle object with radius 1
5         CircleWithPrivateDataFields myCircle =
6             new CircleWithPrivateDataFields(1);
7
8         // Print areas for radius 1, 2, 3, 4, and 5.
9         int n = 5;
10        printAreas(myCircle, n);
11
12        // See myCircle.radius and times
13        System.out.println("\n" + "Radius is " + myCircle.getRadius());
14        System.out.println("n is " + n);
15    }
16
17    /** Print a table of areas for radius */
18    public static void printAreas(
19        CircleWithPrivateDataFields c, int times) {
20        System.out.println("Radius \t\tArea");
21        while (times >= 1) {
22            System.out.println(c.getRadius() + "\t\t" + c.getArea());
23            c.setRadius(c.getRadius() + 1);
24            times--;
25        }
26    }
27 }

```

Radius	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	29.274333882308138
4.0	50.26548245743669
5.0	79.53981633974483
Radius is 6.0	
n is 5	

传递对象到方法中



对象数组

```
Circle[] circleArray = new Circle[10];
```

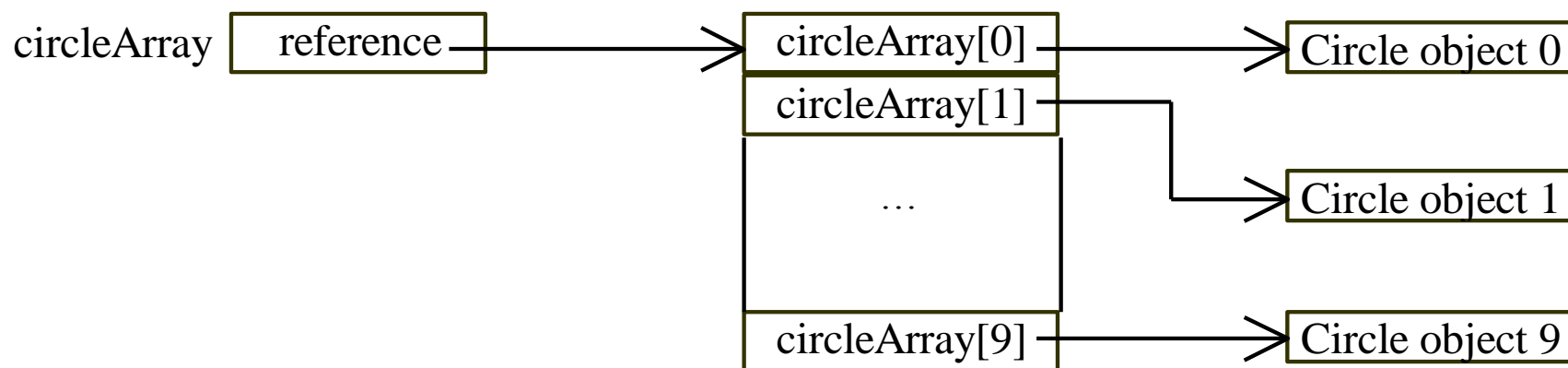
对象数组的实质，是一堆的引用变量构成的。例如，调用`circleArray[1].getArea()`，实际上通过引用操作了两次对象，一次是下标，表示取到数组对象的第二个元素，该元素是个`Circle`的引用，然后通过这个引用，再调用`getArea`方法。



对象数组图示

```
Circle[] circleArray = new Circle[10];
```

注意这行代码执行后， `circleArray[0]...`
`circleArray[9]` 这10个元素都是`null`，因为它们还没有指向某个对象。

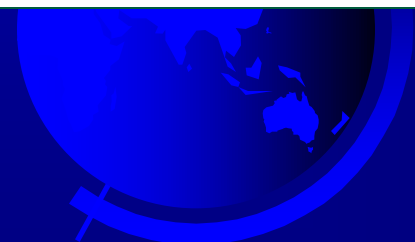


一个很长的例子—1/4

```
1 public class TotalArea {
2     /** Main method */
3     public static void main(String[] args) {
4         // Declare circleArray
5         CircleWithPrivateDataFields[] circleArray;
6
7         // Create circleArray
8         circleArray = createCircleArray();
9
10        // Print circleArray and total areas of the circles
11        printCircleArray(circleArray);
12    }
13
14    /** Create an array of Circle objects */
15    public static CircleWithPrivateDataFields[] createCircleArray() {
16        CircleWithPrivateDataFields[] circleArray =
17            new CircleWithPrivateDataFields[5];
18
19        for (int i = 0; i < circleArray.length; i++) {
20            circleArray[i] =
21                new CircleWithPrivateDataFields(Math.random() * 100);
22        }
23
24        // Return Circle array
25        return circleArray;
26    }
```

一个很长的例子—2/4

```
27
28  /** Print an array of circles and their total area */
29  public static void printCircleArray(
30      CircleWithPrivateDataFields[] circleArray) {
31      System.out.printf("%-30s%-15s\n", "Radius", "Area");
32      for (int i = 0; i < circleArray.length; i++) {
33          System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
34              circleArray[i].getArea());
35      }
36
37      System.out.println("-----");
38
39      // Compute and display the result
40      System.out.printf("%-30s%-15f\n", "The total area of circles is",
41          sum(circleArray) );
```



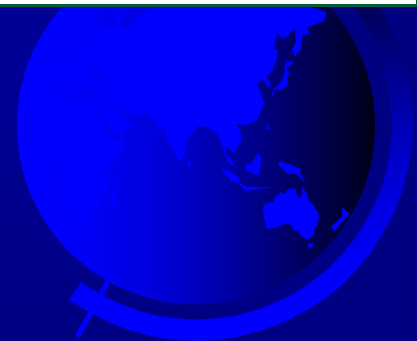
一个很长的例子—3/4

```
42     }
43
44     /** Add circle areas */
45     public static double sum(CircleWithPrivateDataFields[] circleArray) {
46         // Initialize sum
47         double sum = 0;
48
49         // Add areas to sum
50         for (int i = 0; i < circleArray.length; i++)
51             sum += circleArray[i].getArea();
52
53         return sum;
54     }
55 }
```

一个很长的例子—4/4

Radius	Area
70.577708	15648.941866
44.152266	6124.291736
24.867853	1942.792644
5.680718	101.380949
36.734246	4239.280350

The total area of circles is 28056.687544



不可修改的对象和类

➡ 如果希望对象的所有成员都是不可修改的，例如：

```
1 public class Student {  
2     private int id;  
3     private String name;  
4     private java.util.Date dateCreated;  
5  
6     public Student(int ssn, String newName) {  
7         id = ssn;  
8         name = newName;  
9         dateCreated = new java.util.Date();  
10    }  
11  
12    public int getId() {  
13        return id;  
14    }  
15  
16    public String getName() {  
17        return name;  
18    }  
19  
20    public java.util.Date getDateCreated() {  
21        return dateCreated;  
22    }  
23 }
```

➡ 然而按照上述写法，dateCreated有可能被外部类黑掉。



一个例子

☞ 下面这段代码可以把dateCreated成员黑掉。

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student(111223333, "John");  
        java.util.Date dateCreated = student.getDateCreated();  
        dateCreated.setTime(200000); // Now dateCreated field is changed!  
    }  
}
```

☞ 所以，对需要保护的成员，忠告是：

- 所有成员都需要设置为private;
- 不对外提供能够修改成员的方法;
- 所有对外的方法，都不能返回成员的引用。



变量作用域

- ➡ 实例变量和静态变量的作用范围都是整个类，无论它们在类内部的何处声明。当然为了可读性，一般统一放在类的最前或者最后；
- ➡ 局部变量的作用范围，是从声明的地方开始，到包含它的最近的右括号}为止。局部变量在使用前，必须显式初始化，否则会导致编译错误。



this关键字

- ☞ this关键字表示一个引用，这个引用指向**任何对象自己**。this最常用的地方在于通过它访问一个类的隐藏数据域（*hidden data fields*）。
- ☞ this的另外一个常用之处，是在一个类的某个构造方法中，通过它来调用同一个类的另外一个构造方法。



访问隐藏数据域

```
public class Foo {  
    private int i = 5;  
    private static double k = 0;  
  
    void setI(int i) {  
        this.i = i;  
    }  
  
    static void setK(double k) {  
        Foo.k = k;  
    }  
}
```

Suppose that f1 and f2 are two objects of Foo.

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



调用重载的另一个构造方法

```
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 END

