

第9章 对象和类

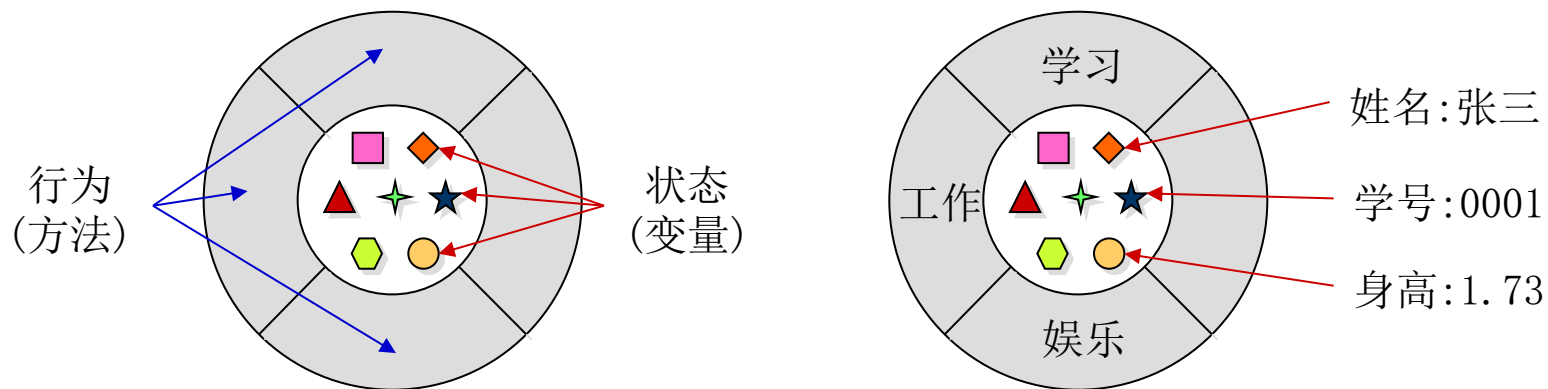
目录

contents

- ◆ 9.1类和对象的定义及UML表示
- ◆ 9.2定义类并用NEW创建其对象
- ◆ 9.3理解构造函数的作用
- ◆ 9.4理解对象访问、向方法传递对象引用
- ◆ 9.5实例(或静态)的变量、常量和方法
- ◆ 9.6可见性修饰符
- ◆ 9.7变量的作用域和访问优先级
- ◆ 9.8this引用

9.1 类和对象的UML表示

- C面向过程(或函数)设计，而Java面向对象设计。
- 对象(object)是现实世界中可识别(不一定可见)的实体，对象具有状态和行为。其状态是其属性的当前值，其行为是一系列方法，这些方法可改变对象的状态。对象：学生、按钮、政府等。

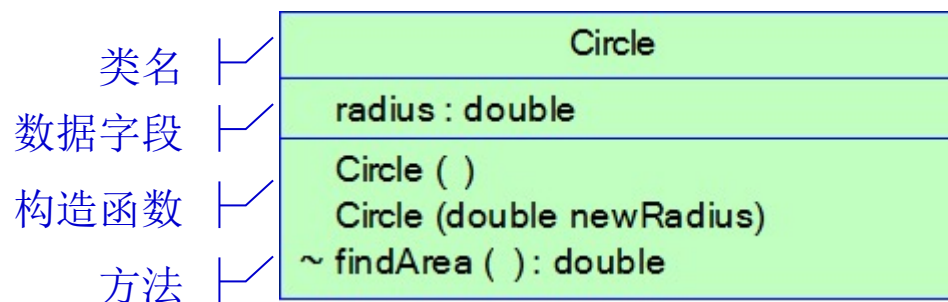


9.1 类和对象的UML表示

- 类(class)定义或封装同类对象共有的属性和方法，即将同类型对象共有的属性和行为抽象出来形成类的定义。
 - 例如要开发学生管理系统，根据应用需求，我们发现所有学生的以下共有属性和行为需要管理
 - 属性：学号、姓名、性别、所在学院、年级、班级
 - 行为：考试、上课、完成作业
 - 因此形成类的定义：Class Student { ... }, 属性作为数据成员，行为作为方法成员
- 同一类型的对象有相同的属性和方法，但每个对象的属性值不同。
- 类(类型简称)是对象的模板、蓝图。**对象是类的实例。**
 - 当定义好类Student，可以用类型Student去实例化不同对象代表不同学生
 - `Student s = new Student(...)`

9.1 类和对象的UML表示

类的UML表示



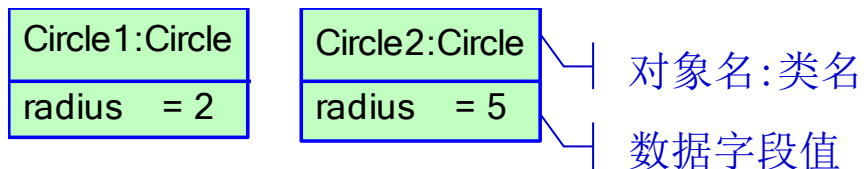
UML：广泛应用的面向对象设计的建模工具，但独立于任何具体程序设计语言。作为一种建模语言，UML有**严格的语法和语义规范**。对于复杂系统，先用UML建模，再编写代码。UML工具会自动把模型编译成Java(C++) 源码（方法体是空的）

UML采用一组**图形符号**来描述软件模型，这些图形符号简单、直观和**规范**。所描述的软件模型，可以直观地理解和阅读，由于具有规范性，所以能够保证模型的准确、一致。

成员访问权限：公有public用+表示，保护protected用#表示，私有private用-表示，包级用~表示或默认无表示

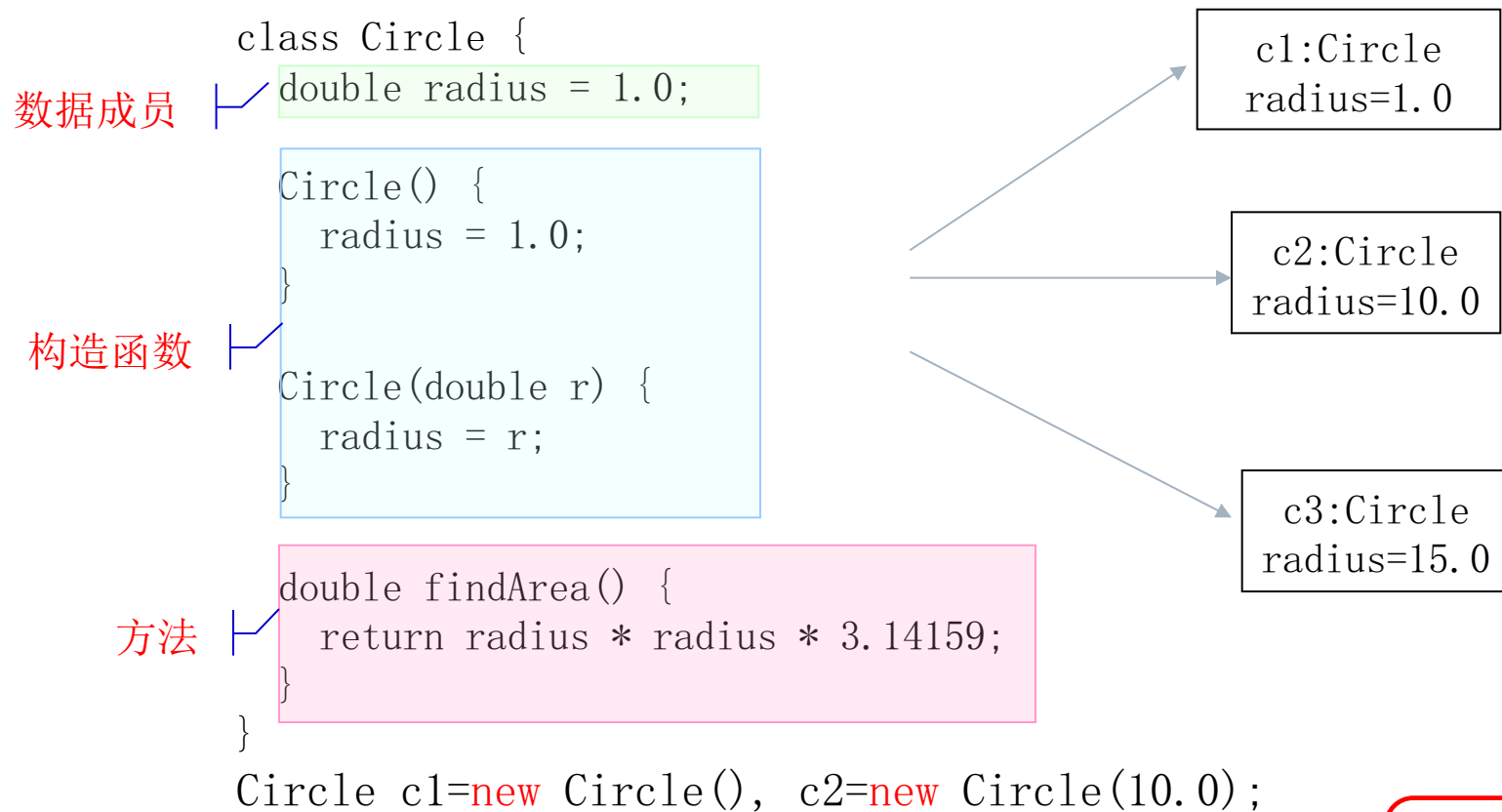
包级即可以被同一个package的代码访问的成员。Java无friend，无析构函数，垃圾自动回收

对象的UML表示



9.2 定义类并用new创建其对象

□ 圆类及其3个对象：数据字段即圆类属性。



□ 说明：Java无struct和union。

new会自动调用构造函数，根据实参确定调用哪个构造函数

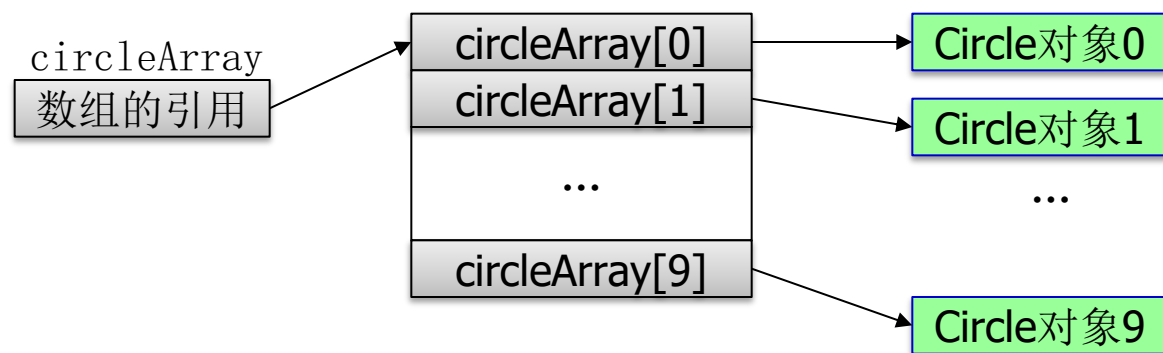
9.2 定义类并用new创建其对象

- 与基本数据类型一样，可声明并用new创建对象数组。

```
int[] a=new int[10]; //所有元素缺省初值=0
```

- 当创建对象数组时，**数组元素**的缺省初值为null。

```
Circle[] circleArray = new Circle[10]; //这时没有构造Circle对象，只是构造数组
for(int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle( );    //这时才构造Circle对象，可使用有参构造函数
}
```



9.3 构造函数(constructor)

- ❑ 无返回类型，名字同类名，用于初始化对象。
- ❑ 注意JAVA如果定义void className(...), 被认为是普通方法
- ❑ 只在new时被自动执行。
- ❑ 必须是实例方法（无static），可为公有、保护、私有和包级权限。
- ❑ 类的变量为引用(相当于C指针)，指向实例化好的对象。
Circle c2=new Circle(5.0); //调用时必须要有括弧, 可带参初始化
- ❑ 缺省构造函数(同C++)
 - 如果类未定义任何构造函数，编译器会自动提供一个不带参数的默认构造函数。
 - 如果已自定义构造函数，则不会提供默认构造函数。
- ❑ Java没有析构函数，但垃圾自动回收之前会自动调用finalize(), 可以覆盖定义该函数（但是finalize调用时机程序员无法控制）。

9.3 构造函数(constructor)

```
public class ConstructorTest {  
    //构造函数前面不能有void  
    public ConstructorTest() {  
        System.out.println("constructor");  
    }  
  
    //如果和类名同名函数前面加了void(可返回任何类型), 编译器看成是普通函数, 这和C++不一样  
    public void ConstructorTest() {  
        System.out.println("normal instance method return void");  
    }  
    public double ConstructorTest(double d) {  
        System.out.println("normal method return double");  
        return d;  
    }  
    public static void main(String ... args){  
        //先调用构造, 再调用void ConstructorTest()  
        new ConstructorTest().ConstructorTest();  
    }  
}
```


9.4 理解对象访问、向方法传递对象引用

- ❑ 访问对象：通过对象引用访问。JVM维护每个对象的引用计数器，只要引用计数器为0，该对象会由JVM自动回收。通过对象引用，可以
 - ❑ 访问对象的实例变量(非静态数据字段)：c2.radius。
 - ❑ 调用对象的实例方法：c2.findArea()。通过c2调用实例方法时，c2引用会传给实例方法里的this引用。
 - ❑ 也可访问静态成员和静态方法（不推荐。推荐用类名）
- ❑ 在**实例**方法中有个this引用，代表当前对象(引用当前对象：相当于指针)，因此在实例方法里，可以用this引用访问当前对象成员
 - this.radius
 - this.findArea()；
 - 在构造函数中调用构造函数，须防止递归调用
 - 不能对this进行赋值
- ❑ 匿名对象也可访问实例(或静态)成员：

```
new Circle( ).radius=2;
```

9.4 理解对象访问、向方法传递对象引用

```
public class Circle {
    double radius = 1.0;

    Circle() {
        radius = 1.0;
    }

    Circle(double r) {
        this.radius = r;
    }

    double findArea() {
        return radius * radius * Math.PI;
    }

    public void setRadius(double newRadius){
        this.radius = newRadius;
    }
}
```

```
public class TestSimpleCircle {
    public static void main(String[] args){
        Circle c1 = new Circle();
        System.out.println("Area = " + c1.findArea() +
            ", radius = " + c1.radius);

        Circle c2 = new Circle(10.0);
        System.out.println("Area = " + c2.findArea() +
            ", radius = " + c2.radius);

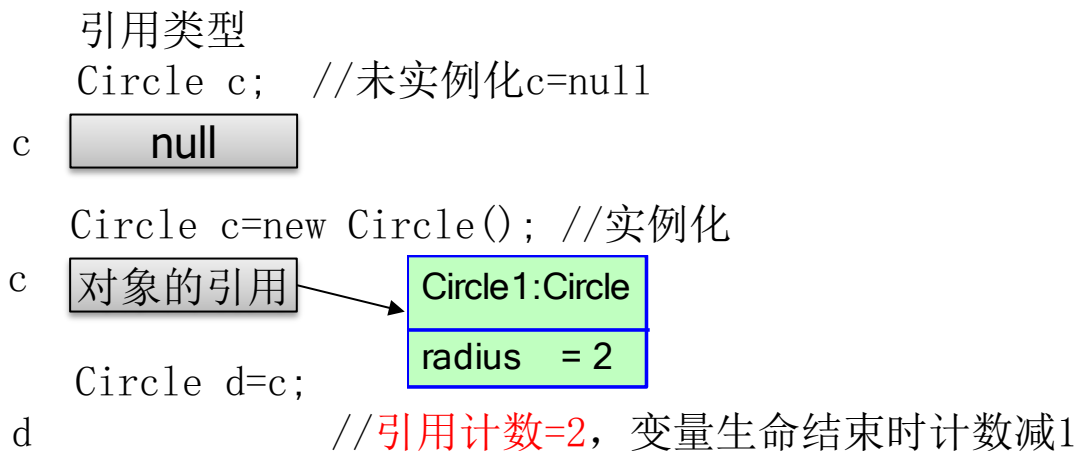
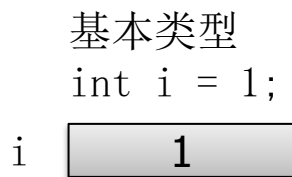
        //modify radius
        c2.setRadius(20.0);
        System.out.println("Area = " + c2.findArea() +
            ", radius = " + c2.radius);
    }
}
```

9.4 理解对象访问、向方法传递对象引用

- 编写程序，定义Circle类，创建Circle对象。
 - 创建一个半径为1的圆。
 - 创建一个半径为25的圆。
 - 创建一个半径为125的圆。
 - 显示三个圆的半径和面积。
 - 将第二个圆的半径改为100，显示其半径和面积。

9.4 理解对象访问、向方法传递对象引用

- 与基本数据类型变量不同：引用变量表示数据的内存单元地址或存储位置。
 - 基本类型变量存储的是基本类型的值。
 - 数组和类是引用类型变量。它引用了内存里的数组或对象。每个对象（数组）有引用计数。
 - 引用类型变量存储的是对象的引用。当变量未引用任何对象或未实例化时，它是值为`null`。



- 一个对象的引用计数=0时被自动回收。

9.4 理解对象访问、向方法传递对象引用

- 对象作为方法参数时与传递数组一样，传递对象实际是传递对象的引用。
 - 基本数据类型传递的是实际值的拷贝，传值后形参和实参不再相关：修改形参的值，不影响实参。
 - 引用类型变量传递的是对象的引用，通过形参修改对象object，将改变实参引用的对象object。



- Java无类似C++的&或C#的ref来修饰方法参数，只能靠形参的声明类型来区分是传值还是传引用，因此一定要注意区分。

9.4 理解对象访问、向方法传递对象引用

包 (package)

- 包是一组相关的类和接口的集合。将类和接口分装在不同的包中，可以避免重名类的冲突，更有效地管理众多的类和接口。因此package就是C++里的namespace
- 包的定义通过关键字package来实现的，package语句的一般形式：
`package 包名;`
- package语句必须出现在.java文件第一行，前面不能有注释行也不能有空白行，该.java文件里定义的所有内容（类、接口、枚举）都属于package所定义的包里。如果.java文件第一行没有package语句，则该文件定义的所有内容位于default包（缺省名字空间），但不推荐。
- 不同.java文件里的内容都可以属于同一个包，只要它们第一条package语句的包名相同

9.4 理解对象访问、向方法传递对象引用

包 (package)

这二个文件里定义的所有内容都属于同一个包hust.cs.java.ch9.因此包是逻辑上的结构，可跨越多个物理的.java文件。

A.java文件

```
package hust.cs.java.ch9;  
  
public class A{  
  
}  
  
其它定义...
```

B.java文件

```
package hust.cs.java.ch9;  
  
public class B{  
  
}  
  
其它定义...
```

9.4 理解对象访问、向方法传递对象引用

包 (package)

- package本质上就是C++里的namespace，因此
 - 在同一个package里不能定义同名的标识符（类名，接口名，枚举名）。例如一个类名和一个接口名不能相同
- 如果要使用其它包里标识符，有二个办法：
 - 用完全限定名，例如要调用java.util包里的Arrays类的sort方法：
`java.util.Arrays.sort(list);`
 - 在package语句后面，先引入要使用其它包里的标识符，再使用：
`import java.util.Arrays; //或者： import java.util.*;`
`Arrays.sort(list);`
- import语句可以有多条，分别引入多个包里的名字

9.4 理解对象访问、向方法传递对象引用

包 (package)

□ 使用二种import的区别

- 单类型导入(single type import) : 导入包里一个具体的标识符, 如

```
import java.util.Arrays;
```

- 按需类型导入(type import on demand) : 并非导入一个包里的所有类, 只是按需导入

```
import java.util.*;
```

□ 这二种导入的区别类似C++里二种使用名字空间方式的区别

- 单类型导入 : 把导入的标识符引入到当前.java文件, 因此当前文件里不能定义同名的标识符, 类似C++里 using nm::id; 把名字空间nm的名字id引入到当前代码处
- 按需导入 : 不是把包里的标识符都引入到当前.java文件, 只是使包里名字都可见, 使得我们要使用引入包里的名字时可以不使用完全限定名, 因此在当前.java文件里可以定义与引入包里同名的标识符。但二义性只有当名字被使用时才被检测到。类似于C++里的using nm;

9.4 理解对象访问、向方法传递对象引用

包 (package)

```
package p1;

public class A {

}
```

```
package p2;

//单类型导入，把p1.A引入到当前域
import p1.A;

//这个时候当前文件里不能定义A，
//下面语句编译报错
public class A {

}
```

```
package p2;

import p1.*;    //按需导入，没有马上把p1.A引入到当前域

//因此当前文件里可以定义A
public class A {
    public static void main(String[] args){
        A a1 = new A();    //这时A是p2.A
        System.out.println(a1 instanceof p2.A); //true

        //当前域已经定义了A，因此要想使用package p1里的A，
        //只能用完全限定名
        p1.A a2 = new p1.A();

    }
}
```

如果出现了名字冲突，要用完全限定名消除冲突

9.4 理解对象访问、向方法传递对象引用

包 (package)

```
package p1;  
  
public class A {  
  
}
```

```
package p2;  
  
public class A {  
  
}
```

```
package p3;  
//可以按需导入, 没有马上把p1.A, p2.A引入到当前域  
//因此下面二个import不会报错  
import p1.*;  
import p2.*;  
  
public class B {  
    //当名字被使用时二义性才被检测  
    A a; //报错, Reference to A is a ambiguous, p1.A and p2.A match;  
    p1.A a1; //这时只能用完全限定名  
    p2.A a2;  
}
```

```
package p3;  
  
import p1.A;  
import p2.A; //报错, p1.A is already defined in a single type import  
  
public class B {  
  
}
```

9.4 理解对象访问、向方法传递对象引用

包 (package)

- 包除了起到名字空间的作用外，还有个很重要的作用：提供了package一级的访问权限控制（在Java里，成员访问控制权限除了公有、保护、私有，还多了包一级的访问控制；类的访问控制除了public外，也多了包一级的访问控制）
- 包的命名习惯: 将Internet域名作为包名（但级别顺序相反），这样的好处是避免包名的重复
 - org.apache.tools.zip
 - cn.edu.hust.cs.javacourse.ch1
 - 如果所有程序员都遵循这种包命名的约定，包名重复的可能性就非常小
- 注意包名和实际工程目录之间的对应关系，在第一章里已经详细介绍

9.4 理解对象访问、向方法传递对象引用

数据成员的封装

- ❑ 面向对象的封装性要求最好把实例成员变量设为私有的或保护的
- ❑ 同时为私有、保护的实例成员变量提供公有的get和set方法。get和set方法遵循JavaBean的命名规范
- ❑ 设成员为DateType propertyName。
 - ❑ get用于获取成员值: `public DateType getPropertyName();`
 - ❑ set用于设置成员值: `public void setPropertyName(DateType value)`

```
class Circle{  
    private double radius=1.0;           //数据成员设为私有  
    public Circle( ){ radius=1.0; }  
    public double getRadius( ){ return radius; }  
    public void setRadius(double r){ radius=r; }  
}
```

如果是公有，就无法防止类的使用者写出
`o.radius = -100.0;`这样的语句

9.5 实例(或静态)的变量、常量和方法

```
class Circle {  
    private double radius;  
    /** 私有静态变量，记录当前内存里被实例化的Circle对象个数*/  
    private static int numberOfObjects = 0;  
  
    public Circle() { radius = 1.0; numberOfObjects++; }  
    public Circle(double newRadius) { radius = newRadius; numberOfObjects++; }  
  
    public double getRadius() {return radius;}  
    public void setRadius(double newRadius) { radius = newRadius;}  
    /** 公有静态方法，获取私有静态变量内容*/  
    public static int getNumberOfObjects() {return numberOfObjects;}  
  
    /** Return the area of this circle */  
    public double findArea() { return radius * radius * Math.PI; }  
    @Override  
    public void finalize() throws Throwable {  
        numberOfObjects--; //对象被析构时，计数器减1  
        super.finalize();  
    }  
}
```

在每个重载的构造函数里计数器+1

覆盖从Object继承的finalize方法，该方法在对象被回收时调用，方法里对象计数器-1。注意该方法调用时机不可控制。 @Override是注解(annotation)告诉编译器这里是覆盖父类的方法。

9.5 实例(或静态)的变量、常量和方法

```
class Circle {  
    ... //其他代码省略  
    @Override  
    public void finalize() throws Throwable {  
        numberOfObjects--; //对象被析构时，计数器减1  
        super.finalize();  
    }  
}
```

@Override可以不加，但是使用@Override注解有如下好处：

- 1: 可以当注释用，方便阅读；
- 2: 编译器可以给你验证@Override下面的方法名是否是父类中所有的，如果没有则报错。
例如，如果没写@Override，而下面的方法名又写错了，这时你的编译器是可以编译通过的，因为编译器以为这个方法是你的子类中自己增加的方法。

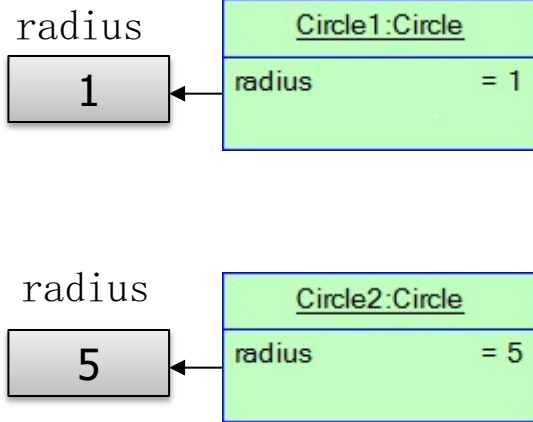
Java注解为 Java 代码提供元数据。注解可以指示编译器做些额外的动作，甚至可以自定义Java注解让编译器执行自定义的动作。Java提供了Annotation API让我们自定义注解。

9.5 实例(或静态)的变量、常量和方法

- ❑ 实例变量(instance variable):未用static修饰的成员变量, 属于类的具体实例(对象), 只能通过对象访问, 如“对象名.变量名”。
- ❑ 静态变量(static variable)是用static修饰的变量, 被类的所有实例(对象)共享, 也称类变量。可以通过对象或类名访问, 提倡“类名.变量名”访问。

Circle	
- radius	: double
- <u>numberOfObject</u>	: int
+ getRadius ()	: double
+ setRadius (double radius)	: void
+ <u>getNumberOfObject ()</u>	: int
+ findArea() ()	: double

- 表示private
+ 表示public
下划线 表示static



实例变量是作为对象内存的一部分存在

二个对象的radius:
实例变量存储于不同
对象, 彼此不影响

numberOfObject=2

静态变量是单独的内存单元, 与对象内存
分开

9.5 实例(或静态)的变量、常量和方法

- ❑ 实例常量是没有用static修饰的final变量。
- ❑ 静态常量是用static修饰的final变量。Math类中的静态常量PI定义为：

```
public static final double PI = 3.14159265358979323846;
```
- ❑ 所有常量可按需指定访问权限，不能用等号赋值修改。
- ❑ 由于它们不能被修改，故通常定义为public。
- ❑ final也可以修饰方法
 - ❑ final修饰实例方法时，表示该方法不能被子类覆盖(Override)。非final实例方法可以被子类覆盖（见继承）
 - ❑ final修饰静态方法时，表示该方法不能被子类隐藏(Hiding)。非final静态方法可以被子类隐藏。
- ❑ 构造函数不能为final的。

9.5 实例(或静态)的变量、常量和方法

方法重载 (Overload)、方法覆盖(Override)、方法隐藏(Hiding)

- 方法重载：同一个类中、或者父类子类中的多个方法具有相同的名字，但这些方法**具有不同的参数列表**（不含返回类型，即无法以返回类型作为方法重载的区分标准）
- 方法覆盖和方法隐藏：发生在父类和子类之间，前提是继承。子类中定义的方法与父类中的方法具有**相同的方法名字、相同的参数列表、相同的返回类型**（也允许子类中方法的返回类型是父类中方法返回类型的子类）
 - 方法覆盖：实例方法
 - 方法隐藏：静态方法

9.5 实例(或静态)的变量、常量和方法

方法重载 (Overload)、方法覆盖(Override)、方法隐藏(Hiding)

```
public class A {  
    public void m(int x, int y) {}  
    public void m(double x, double y) {}
```

//下面语句报错m(int,int)已经定义, 重载函数不能通过返回类型区分

```
//    public int m(int x, int y) { return 0;};  
}
```

```
class B extends A{ //B继承了A
```

```
    public void m(float x, float y) { } //重载了父类的m(int,int)和m(double,double)
```

```
    public void m(int x, int y) {} //覆盖了父类的void m(int,int), 注意连返回类型都必须一致
```

//注意下面这个语句报错, 既不是覆盖 (与父类的void m(int,int)返回类型不一样)

// 也不是合法的重载 (和父类的m(int,int)参数完全一样, 只是返回类型不一致)

```
//    public int m(int x, int y) {} //错误
```

//子类定义了新的重载函数int m()

```
public int m(){return 0;};
```

```
}
```

9.5 实例(或静态)的变量、常量和方法

方法重载 (Overload)、方法覆盖(Override)、方法隐藏(Hiding)

```
class A{
    public void m1(){ }
    public final void m2() { }

    public static void m3() { }
    public final static void m4() { }
}

class B extends A{
    //覆盖父类A的void m1()
    public void m1(){ }

    //下面语句报错，不能覆盖父类的final 方法
    //    public void m2(){ }

    public static void m3() { } //隐藏了父类的static void m3()
    //下面语句报错，父类final 静态方法不能被子类隐藏
    //    public static void m4() { }
}
```

9.5 实例(或静态)的变量、常量和方法

- ❑ 静态方法(static method)是用static修饰的方法。
- ❑ 构造函数不能用static修饰，静态函数无this引用。
- ❑ 每个程序必须有public static void main(String[])方法。
 - JOptionPane.showMessageDialog
 - JOptionPane.showInputDialog
 - Math.random
- ❑ 静态方法可以通过对象或类名调用。
- ❑ 静态方法内部只能访问类的静态成员（因为实例成员必须有实例才存在，当通过类名调用静态方法时，可能该类还没有一个实例）
- ❑ 静态方法没有多态性。

9.6 可见性修饰符

- **类访问控制符**：public和包级(默认)；**类的成员访问控制符**：private、protected、public和包级(默认)
- **Java继承时无继承控制**(见继承，即都是公有继承，和C++不同)，故**父类成员继承到派生类时访问权限保持不变**(除了私有)。
- **成员访问控制符的作用**：
 - private：只能被当前类定义的函数访问。
 - 包级：无修饰符的成员，只能被同一包中的类访问。
 - protected：子类、同一包中的类的函数可以访问。
 - public：所有类的函数都可以访问。

访问权限	本类	本包	子类	它包
public	√	√	√	√
protected	√	√	√	X
包级(默认)	√	√	X	X
private	√	X	X	X

9.6 可见性修饰符

类的成员访问控制符

```
package p1;
```

```
public class C1{//在C1.java
    public int x=1;
    int y=2;//包级
    protected int u=3,w=4;
    private int z;
    public void m1() {
        int i = x = u;
        int j = y = w;
        int k = z;
        m2();
        m3();
    }
    void m2() { } //包级
    private void m3() { }
}
```

```
public class C2 extends C1{
    //在C2.java
    int u=5; //包级
    void aMethod() {
        C1 o = new C1( );//ok, C1是public
        int i = o.x;//ok, x是public
        int j = o.y;//ok, y(包级), 可在同一包内访问
        int h = o.u;//ok, u(保护)可在同一包内访问
        i=u+super.u;//ok, 本类u及super.u(父类保护)
        int k = o.z;//error, z是私有的
        o.m1(); //ok, m1是public
        o.m2(); //ok, m2无访问修饰, 可在同一包内访问
        o.m3(); //error, m3是私有的
    }
}
```

9.6 可见性修饰符

类的成员访问控制符

```
package p2;
```

```
public class C3 extends C1{ //C3.java
    int u=5;
    void aMethod() {
        C1 o =new C1( );//ok, C1是public
        int i = o.x; //ok, x 是public
        int j = o.y;//error, y(包级), 不能在包内访问
        int h = o.u;//error, u(保护, 当前对象非o子类对象), 不能在包内访问
        i=u+super.u;//ok, 本类u及super.u(保护, 当前对象是父类super的子类对象)
        int k = o.z;//error, z是私有的
        o.m1(); //ok, m1是public
        o.m2(); //error, m2(包级), 不能在包内访问
        o.m3(); //error, m3是私有的
    }
}
```

- * 子类类体中可以访问从父类继承来的**protected**成员。但如果子类与父类不在同一个包里，子类里不能访问另外父类实例（非继承）的**protected**成员。

9.6 可见性修饰符

类的访问控制符

```
package p1;
```

```
//C1无访问修饰符，只能  
//在同一包内被访问  
class C1{  
    ...  
}
```

```
public class C2{  
    //可访问同一包的C1类  
    C1 c; //OK  
}
```

```
package p2;
```

```
public class C3{  
    //不可访问包package p1中的C1类  
    C1 c; //error  
    //可访问包package p1中的C2类 (public)  
    C2 c; //OK  
}
```

9.6 可见性修饰符

访问控制针对的是类型而不是对象级别

```
public class Foo{
    private boolean x;

    public void m(){
        Foo foo = new Foo();

        //因为对象foo在Foo类内使用，所以可以访问私有成员x，并不是只能访问this.x
        boolean b = foo.x //ok
    }
}
```

```
public class Test{
    public static void main(String[] args){
        Foo foo = new Foo();

        //因为对象foo在Foo类外使用，所以不可以访问foo的私有成员x
        boolean b = foo.x //error
    }
}
```

9.6 可见性修饰符

- 大多数情况下，构造函数应该是公有的
- 有些特殊场合，可能会防止用户创建类的实例，这可以通过将构造函数声明为私有的来实现。
 - 例如，包java.lang中的Math类的构造函数为私有的，所有的数据域和方法都是静态的，可以通过类名直接访问而不能实例化Math对象。

```
private Math () { }
```

9.7 类成员变量的作用域和访问优先级

- 类的成员变量(实例变量和静态变量)的作用域是整个类，与声明的位置无关。
- 如果一个成员变量的初始化依赖于另一个变量，则另一个变量必须在前面声明。

```
public class Foo {  
    int i;//成员变量默认初始化，new后成员默认值为0或null，函数局部变量须初始化  
    int j = i + 1;  
    int f( ){ int i=0; return i+this.i; } //局部变量i会优先访问  
} //作用域越小，被访问的优先级越高
```

- 如函数的局部变量i与类的成员变量i名称相同，那么优先访问局部变量i，成员变量i被隐藏(可用this.i或类名.i发现)。

9.8 this引用

- this引用指向调用某个方法的当前对象
 - 在实例方法中，实例变量被同名局部变量或方法形参**隐藏**，可以通过 `this.instanceVariable` 访问实例变量。
- 调用当前类的其它构造函数，需防止递归调用。
 - `this(actualParameterListopt)`
 - 必须是构造函数的第1条语句。

```
class Foo{  
    int i = 5;  
    static double k = 0.0;  
    void setI(int i){  
        this.i = i;  
    }  
    static void setK(int k){  
        Foo.k = k;  
    }  
}
```

```
Foo f1 = new Foo();  
Foo f2 = new Foo();
```

```
f1.setI(10); //这时this引用f1  
f2.setI(45); //这时this引用f2
```

说明：成员变量 `i`、`k` 的初始值可被构造函数 `Foo()` 修改，但编译提供的默认构造函数 `F00()` 什么也没有做。