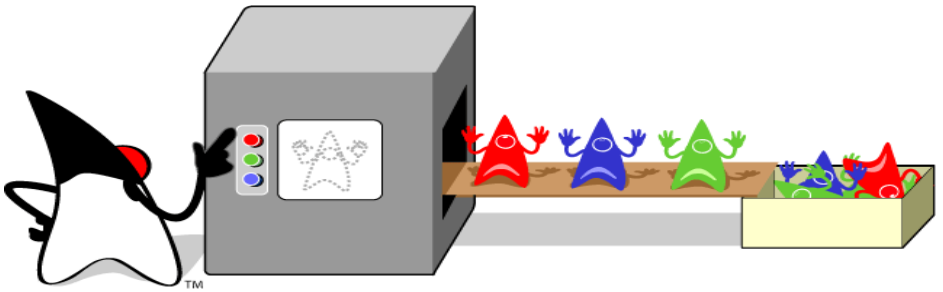
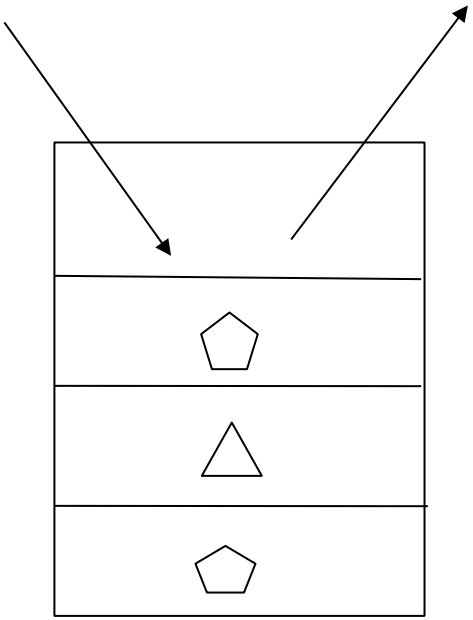
Chap 5 Java类的语法结构和体系结构

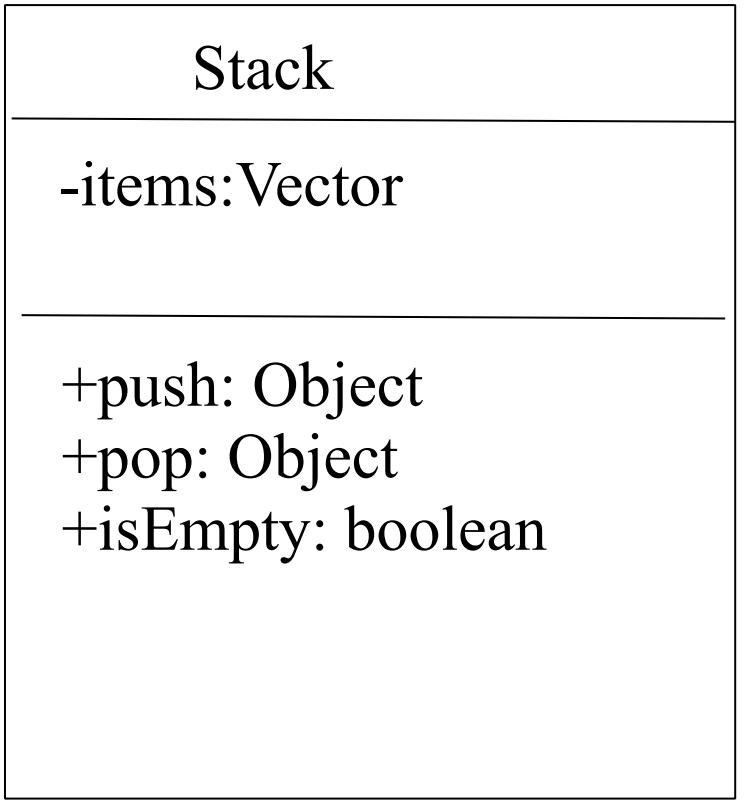


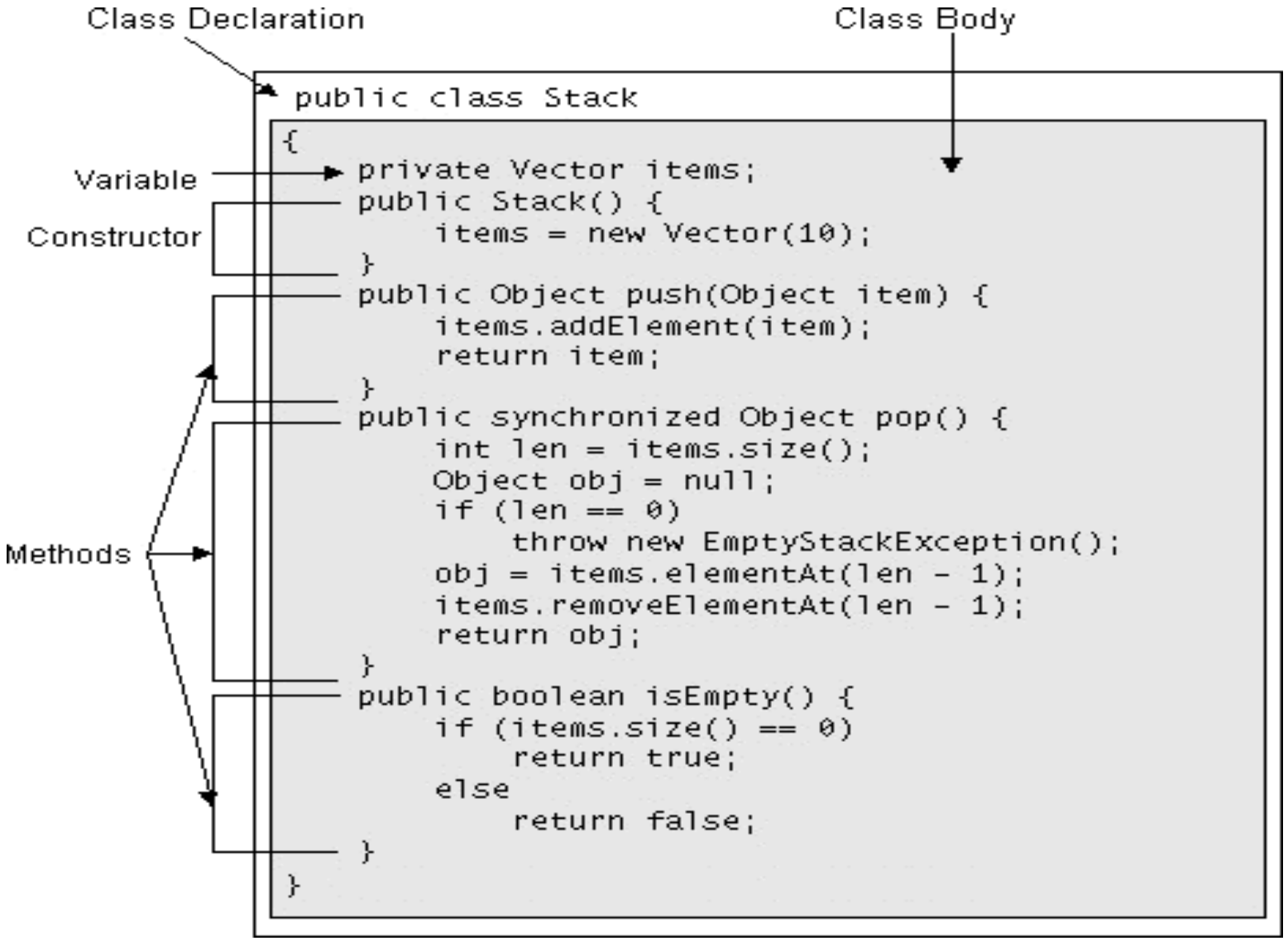
5.1 Java类的语法结构

举例: 堆栈的实现



堆栈




Class Declaration
Class Body
Variable
Constructor
Methods
public class Stack
{
private Vector items;
public Stack() {
items = new Vector(10);
}
public Object push(Object item) {
items.addElement(item);
return item;
}
public synchronized Object pop() {
int len = items.size();
Object obj = null;
if (len == 0)
throw new EmptyStackException();
obj = items.elementAt(len - 1);
items.removeElementAt(len - 1);
return obj;
}
public boolean isEmpty() {
if (items.size() == 0)
return true;
else
return false;
}
}

类的定义由下面两个部分组成:

*类的声明，确定类的名称.

* 类体，包含提供由类得到的对象的生命周期的全部代码 (构造方法，成员变量和成员方法等).

5.1.1声明一个类

public	Class is publicly accessible.
abstract	Class cannot be instantiated.
final	Class cannot be subclassed.
class NameOfClass	***Name of the Class.***
extends *Super*	Superclass of the class.
implements *Interfaces*	Interfaces implemented by the class.
{	
ClassBody	
}	

类的声明的各个部分和用途
类的访问级别: **public**, 无显式修饰符

5.1.2 声明成员变量

accessLevel	Indicates the access level for this member.
static	Declares a class member.
final	Indicates that it is constant.
transient	This variable is transient.
volatile	This variable is volatile.
type name	The type and name of the variable.

成员变量声明的各个部分以及用途.

访问级别: private, 无显式修饰符, protected, public

举例: private Vector items;

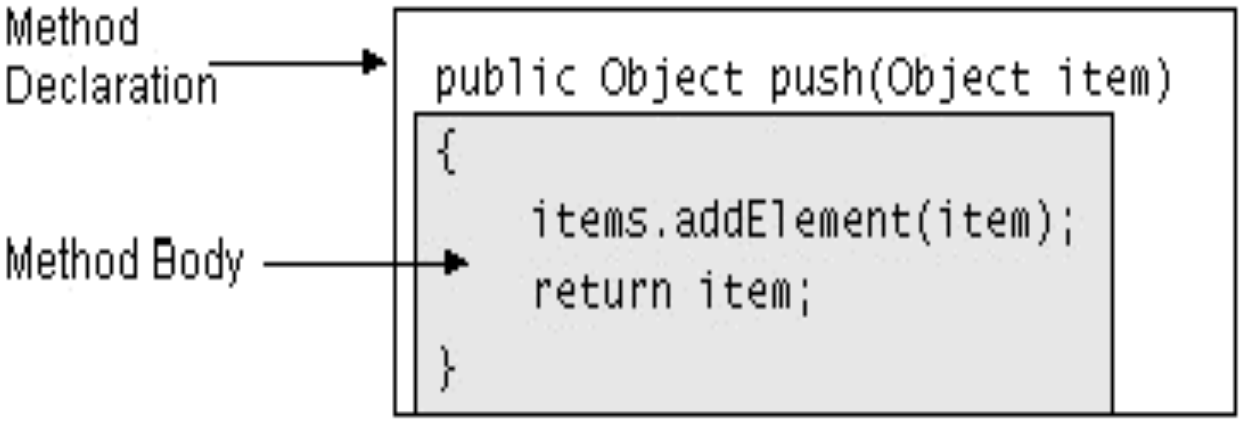
5.1.3 定义方法

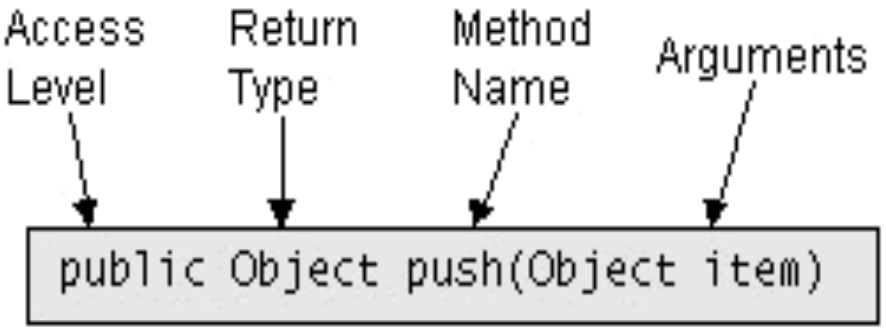
accessLevel	Access level for this method.
static	This is a class method.
abstract	This method is not implemented.
final	Method cannot be overridden.
native	Method implemented in another language.
synchronized	Method requires a monitor to run.
returnType methodName	The return type and method name.
(*paramList*)	The list of arguments.
throws *exceptions*	The exceptions thrown by this method.

方法声明的各个部分和用途
访问级别: private, 无显式修饰符, protected, public

举例:

push 方法


Method
Declaration
Method Body
public Object push(Object item)
{
items.addElement(item);
return item;
}


Access
Level
Return
Type
Method
Name
Arguments
public Object push(Object item)

- 方法名的相关问题

* 除构造方法外，方法名不能与类名相同。

*与父类中方法有相同方法名，参数结构和返回类型的方法将覆盖/重设 (override)或屏蔽/隐藏(hide)父类中的方法。

* Java支持方法名的重载(overload),即多个方法可以有相同的方法名，但方法的参数类型以及其数量应各不相同。

重载举例：

```
class DataArtist {
    void draw(String s) {
        . . .
    }
    void draw(int i) {
        . . .
    }
    void draw(float f) {
        . . .
    }
}
```

5.1.4 类的构造方法

• 构造方法(constructor) 用于对由类创建的对象进行初始化，构造方法的名称应该与类的名称相同。
举例：Stack 类中的构造方法

```
public Stack() {
    items = new Vector(10);
}
```

注意! 构造方法没有返回类型，它由new操作符调用并返回新创建的对象.

•**Java**支持构造方法的重载，一个类可以有多个构造方法，这些构造方法的名称相同但参数表不同.

举例:

```
public Stack() {
    items = new Vector(10);
}

public Stack(int initialSize) {
    items = new Vector(initialSize);
}
-----------------------------------
new Stack(10);
new Stack();
```

- 如果一个类中没有定义构造方法，**Java**运行环境将为其提供一个默认的构造方法，但这个构造方法不做任何具体的初始化工作.

- 通过设定构造方法的访问权限**(private, protected, public,** 无显式修饰符)，可以指定哪些对象可以创建你的类的实例.

5.1.5 在方法和构造方法中传递参数

参数表是用“，”间隔起来的类型 / 名称对。在方法体中，可以简单的使用参数名称来引用参数的数值。

•参数名称

参数名称可以和类的成员变量的名称相同.在这种情形称为参数隐藏成员变量，通常用于构造方法中。 此时在方法中访问的变量是参数而不是成员变量。使用**this**(当前对象)的引用才能访问到成员变量。

```
class Circle {
    int x, y, radius;
    public Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```

- 参数类型

基本型变量类型

举例:

```
double computePayment(double loanAmt, double rate,
                              double futureValue, int numPeriods) {
   double I, partial1, denominator, answer;
   I = rate / 100.0;
   partial1 = Math.pow((1 + I), (0.0 - numPeriods));
   denominator = (1 - partial1) / I;
   answer = ((-1 * loanAmt) / denominator)
         - ((futureValue * partial1) / denominator);
   return answer;
}
```

调用:

```
…computePayment(100.23, 30.50,  3000.12, 6);
```

- 参数类型

引用型变量类型

举例:

```
static Polygon polygonFrom(Point[] listOfPoints) {
    . . .
}
 // Point 是一个位于x,y坐标点的类
// Polygon 是一个表示多边形的类
```

- 参数类型

引用型变量类型

Point类：

```
public class Point {
    public int x = 0;
    public int y = 0;
    //构造方法
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- 参数类型

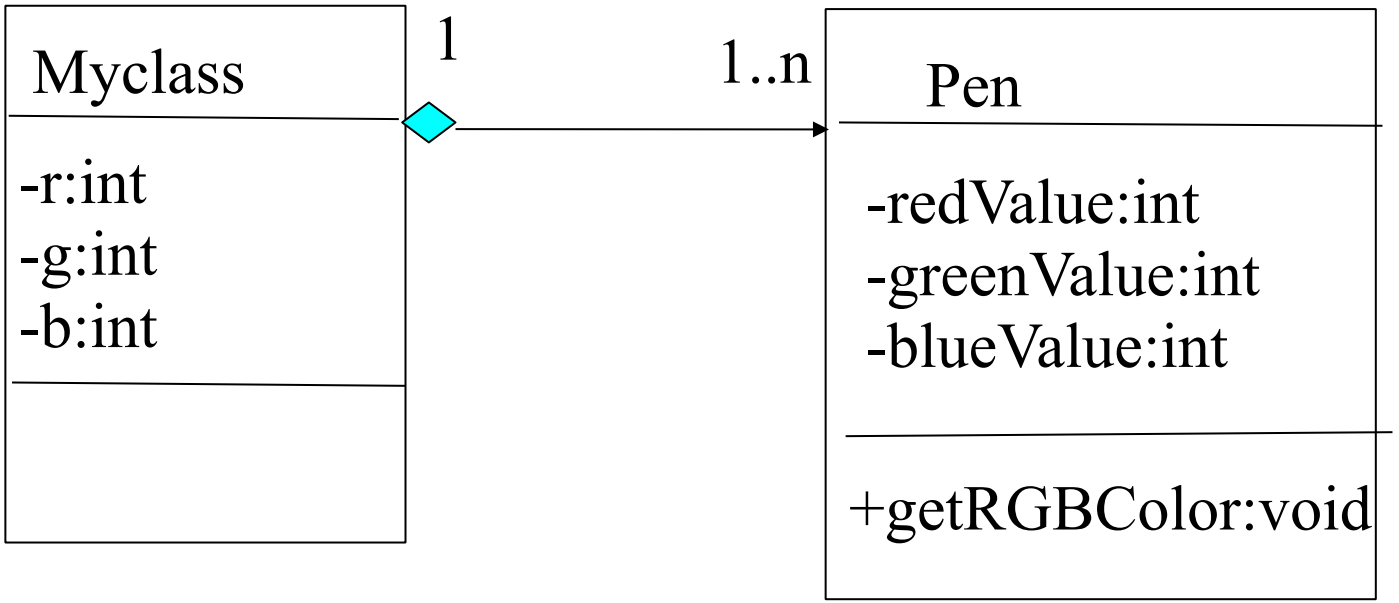
引用型变量类型

调用:

```
Point[] myPointsLists = new Point[3];
myPointsLists[0] = new Point(1,2);
myPointsLists[1] = new Point(7,8);
myPointsLists[2] = new Point(4,5);

…polygonFrom(myPointsLists);
```

举例:

假定有在一个图形应用程序中有若干Java语句用于从Pen对象中获取当前的颜色:


Myclass
-r:int
-g:int
-b:int
1
1..n
Pen
-redValue:int
-greenValue:int
-blueValue:int
+getRGBColor:void

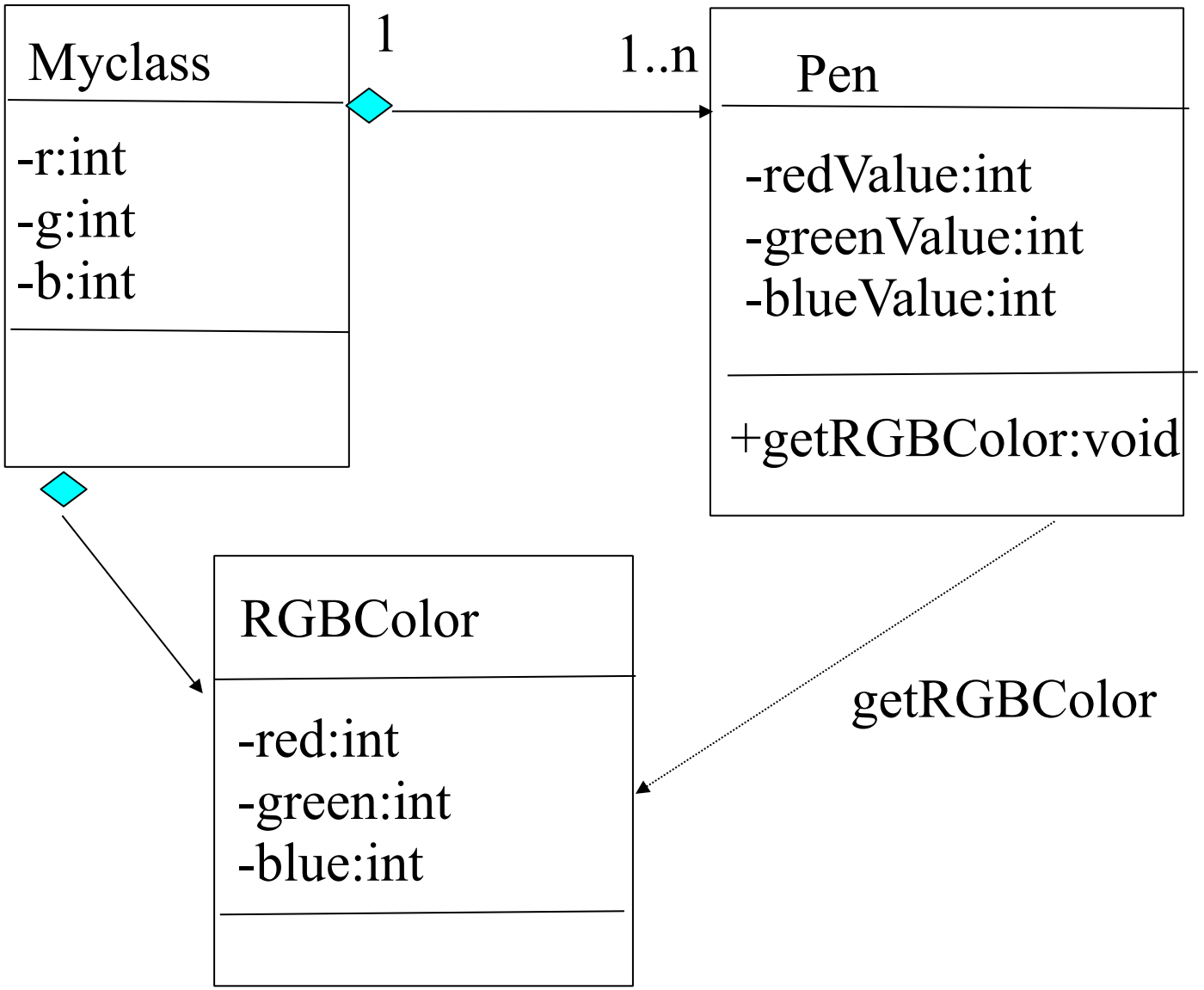
```
class Pen {
   int redValue, greenValue, blueValue;
   void getRGBColor(int red, int green, int
blue) {
      // red, green, and blue have been created
      // and their values are -1
      red = redValue;
      green = greenValue;
      blue = blueValue;
   }
}

    . . .

    int r = -1, g = -1, b = -1;
    pen.getRGBColor(r, g, b);
    System.out.println("red = " + r +
                       ", green = " + g +
                       ", blue = " + b);
```

Java方法将r,g,b变量的变量值(-1)而不是对其的引用传递给getRGBColor方法，相当于getRGBColor(-1, -1, -1).

对变量的本地拷贝的改变不会影响调用时使用的参数。

可以看到方法并未象设想那样去工作，当控制走向**println**语句时，**getRGBColor**的参数**(red,green,blue)**将不再存在，**r,g,b**的值仍然是**-1.**

传值给程序员带来了安全性的保证，也就是，即使不小心，方法也不会修改一个在作用范围之外的变量的值。但另一方面，我们常常需要一个方法能修改参数的值，如例子中的getRGBColor，如何实现这样的操作呢?


Myclass
-r:int
-g:int
-b:int
1
1..n
Pen
-redValue:int
-greenValue:int
-blueValue:int
+getRGBColor:void
RGBColor
-red:int
-green:int
-blue:int
getRGBColor

引入一个新的对象RGBColor,来保存red,green,blue值:

```
class RGBColor {
    public int red, green, blue;
}
```

改写 Pen 类中的getRGBColor方法 如下:

```
class Pen {
    int redValue, greenValue, blueValue;
    void getRGBColor(RGBColor aColor) {
        aColor.red = redValue;
        aColor.green = greenValue;
        aColor.blue = blueValue;
    }
}
```

改写调用过程:

```
. . .
RGBColor penColor = new RGBColor();
pen.getRGBColor(penColor);
System.out.println("red = " + penColor.red +
                   ", green = " + penColor.green +
                   ", blue = " + penColor.blue);
. . .
```

5.1.6 方法的返回值

没有声明为void的方法应该包含return语句，其返回的数值的类型必须与方法的返回类型一致。

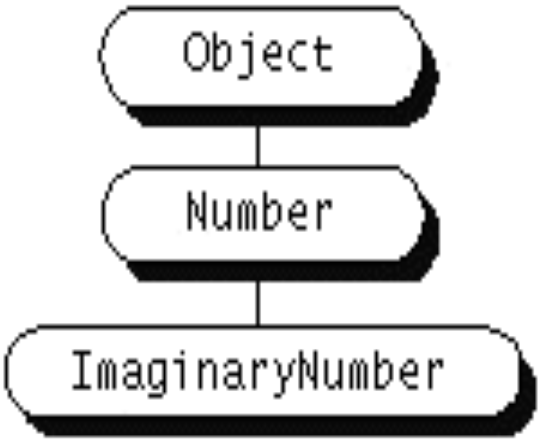
举例:

```
public boolean isEmpty() {
    if (items.size() == 0)
        return true;
    else
        return false;
} //返回基本类型
```

```
public synchronized Object pop() {
    int len = items.size();
    Object obj = null;
    if (len == 0)
        throw new EmptyStackException();
    obj = items.elementAt(len - 1);
    items.removeElementAt(len - 1);
    return obj;
} // 返回引用类型
```

注意！当方法返回一个对象时，该对象的类必须与返回类型的类完全相同或者是其子类。

举例:




```
public Number returnANumber() {
    . . .
}
```

returnANumber 方法可以返回一个ImaginaryNumber 而不能返回一个Object.

5.1.7使用this关键字

通常可以在一个对象的方法体中直接访问对象的成员变量，但当方法的参数名与成员变量名相同时，需要使用this来对成员变量进行访问.

举例:

```
class HSBColor {
    int hue, saturation, brightness;
    HSBColor (int hue, int saturation, int brightness) {
        this.hue = hue;
        this.saturation = saturation;
        this.brightness = brightness;
    }
```

另外，在同一个类中，也可以使用**this**来调用其他构造方法。

```
public class Rectangle{
  private int x,y;
  private int width, height;
  public Rectangle(){
  this(0, 0, 0, 0);
  }
  public Rectangle( int width, int height){
  this(0, 0, width, height);
  }
```

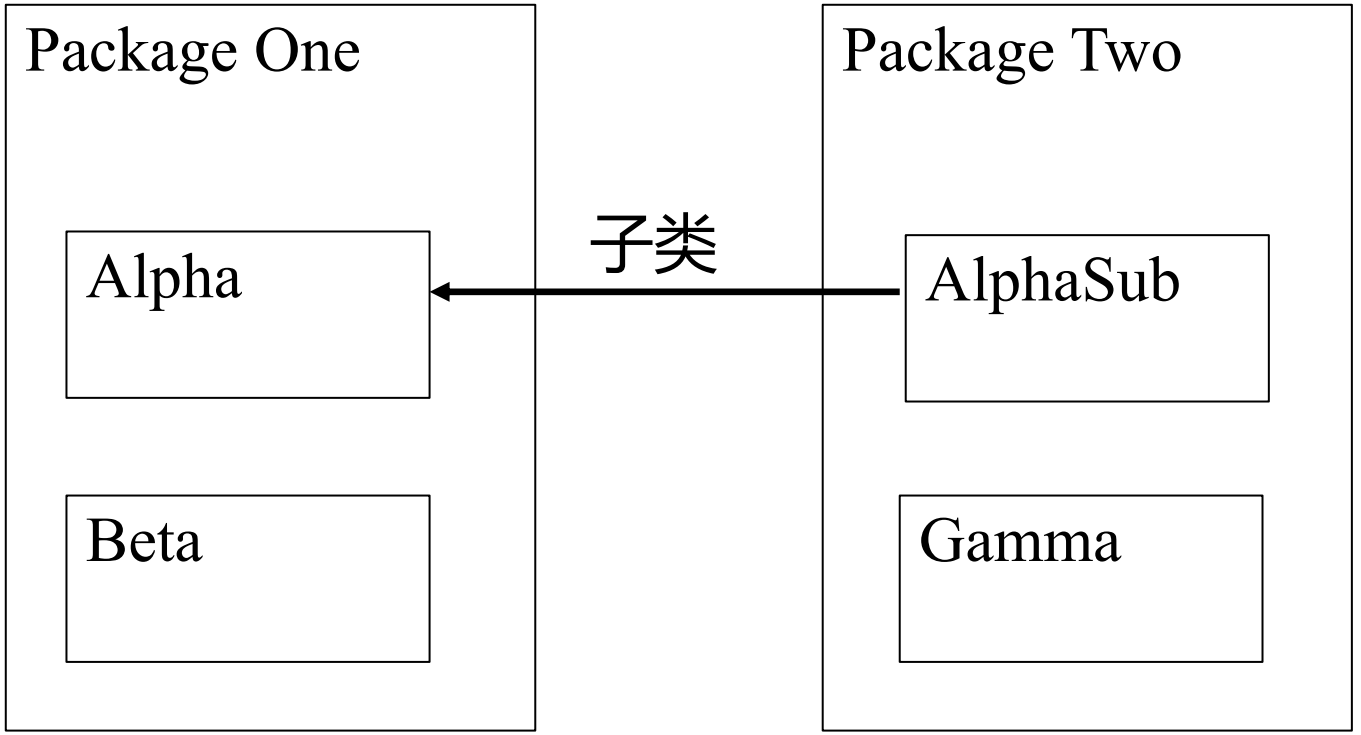
```
public Rectangle( int x, int y,int width, int height){
  this.x = x; this.y =y; this.width = width; this.height =height;
  }
  .......
}
```

5.1.8 对类的成员的访问控制

在java 中，可以使用访问限定字(access specifier),也称为访问级别修饰符来对类中的变量或方法进行访问控制。

Specifier(修饰符)	**class (类)**	**package(包)**	**subclass(子类)**	**world(所有)**
private	√			
无显式修饰符	√	√		
protected	√	√	√	
public	√	√	√	√

举例



Alpha类成员的访问范围

Specifier(修饰符)	Alpha	Beta	AlphaSub	Gamma
private	√			
无显式修饰符	√	√		
protected	√	√	√	
public	√	√	√	√

5.1.9 理解实例的成员和类的成员

实例变量和类变量(类变量用static声明).

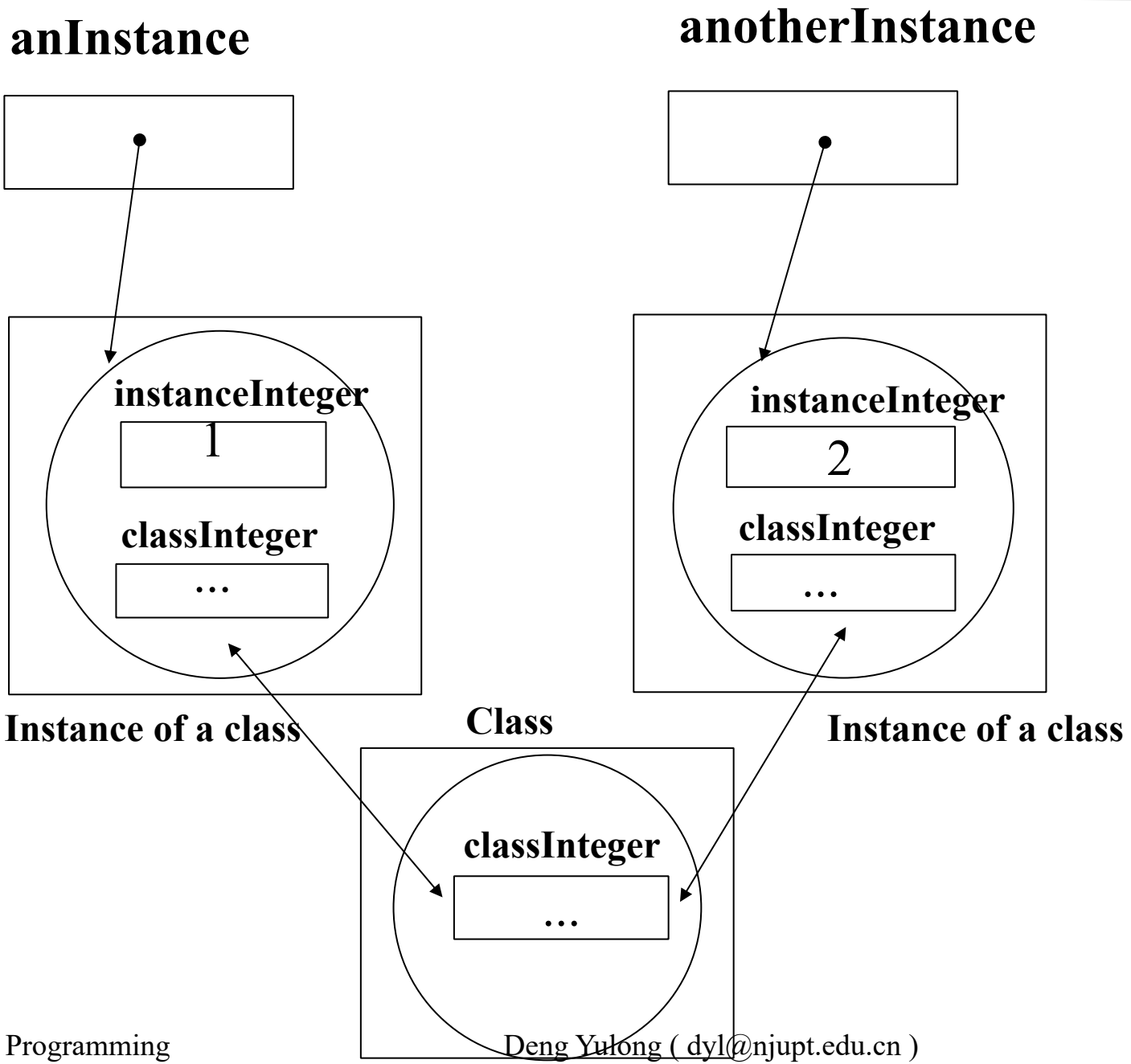
Java运行环境为类的每一个实例创建一份实例变量的拷贝，但无论该类创建了多少实例，只为类分配一次类变量。

举例:

```
public class AClass {

   public int instanceInteger = 0;
   public int instanceMethod() {
      return instanceInteger;
   }
   public static int classInteger = 0;
   public static int classMethod() {
      return classInteger;
   }
   public static void main(String[] args) {
      AClass anInstance = new AClass();
      AClass anotherInstance = new AClass();
```

```
//通过实例引用实例变量和实例方法
anInstance.instanceInteger = 1;
anotherInstance.instanceInteger = 2;
System.out.println(anInstance.instanceMethod());
System.out.println(anotherInstance.instanceMethod());

//在类的方法中直接引用实例变量或方法则非法
//System.out.println(instanceMethod());    //illegal
//System.out.println(instanceInteger);     //illegal

//通过类引用类变量或方法
AClass.classInteger = 7;
System.out.println(classMethod());
```

```
    //通过实例引用类变量或方法
        System.out.println(anInstance.classMethod());

         //Instances share class variables
        anInstance.classInteger = 9;
        System.out.println(anInstance.classMethod());
        System.out.println(anotherInstance.classMethod());
    }
}
```


anInstance
anotherInstance
instanceInteger
1
classInteger
…
instanceInteger
2
classInteger
…
Instance of a class
Class
Instance of a class
classInteger
…

5.1.10 对实例的成员变量和类变量进行初始化

在类的定义中可以在声明类变量和实例变量的时候对其进行初始化:

```
class BedAndBreakfast {
    static final int MAX_CAPACITY = 10;
    boolean full = false;
}
```

注意! 直接初始化受到的一些限制条件如下.

* 初始化仅能采用赋值语句的形式进行.
* 初始化不能调用任何可能抛出指定异常(checked exception)的方法.
* 如果初始化调用了抛出实时异常(runtime exception)的方法,

则不能进行错误恢复(error recovery).

初始化的另一种方式

在初始化类变量时，可以将初始化代码放在静态初始化体中。

在初始化实例变量时，可以将初始化代码放在构造方法中。

- 使用静态初始化体

举例：

```
import java.util.ResourceBundle;

class Errors {
    static ResourceBundle errorStrings;
    static {
        try {
            errorStrings = ResourceBundle.
                                getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            // error recovery code here
        }
    }
}
```

•初始化实例变量

举例:

```
// 其中errorStrings 为实例变量
import java.util.ResourceBundle;

class Errors {
   ResourceBundle errorStrings;

 Errors() {
       try {
          errorStrings = ResourceBundle.
                         getBundle("ErrorStrings");
       } catch (java.util.MissingResourceException e) {
          // error recovery code here
       }
    }
}
```

举例：

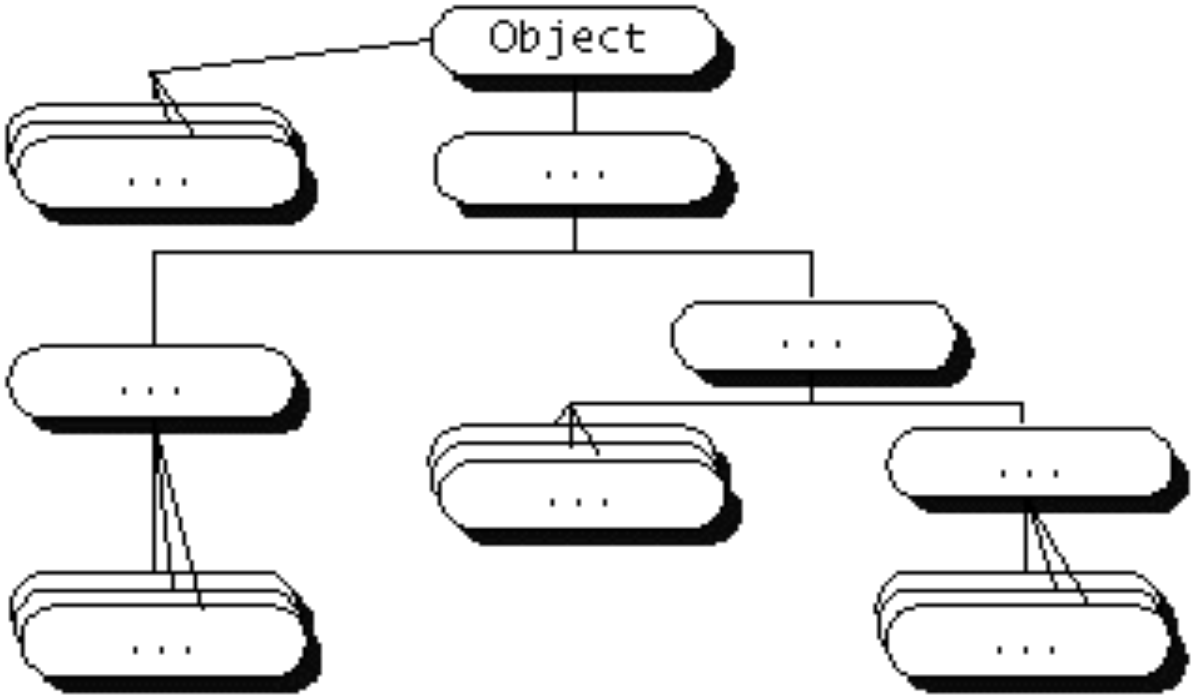
```
public class ExGui{
  private int k;
  public static void main(String argus[]){
  k = 0;  // error occured
 }
}
```

//如何改写代码以便能正确初始化变量k？

```
// 以下代码是否正确？

public class ExGui{
   private static int k;
   public static void main(String argus[]){
   k = 0;
  }
 }
```

5.2 继承和方法覆盖及屏蔽（重设及隐藏）

Java中所有的类都派生自Object 类



子类继承了所有的父类的状态和行为。

5.2.1 方法的覆盖（重设）和屏蔽（隐藏）(overriding 和 hiding)

子类中的实例方法如果与父类中的实例方法有相同的方法名，
参数表以及返回类型，则称为子类的方法覆盖（重设）了父类中的方法。
子类中的类方法(静态方法)如果与父类中的类方法(静态方法)有相同
的方法名，参数表以及返回类型，则称为子类的类方法(静态方法)
屏蔽(隐藏)了父类中的类方法(静态方法) 。

举例: 覆盖父类 Object中的toString方法

```
public class MyClass extends Object{
   private int anInt = 4;
   //覆盖Object类中的toString方法
   public String toString(){
   return "Instance of MyClass. anInt = "+ anInt;
   }
}
```

```
public class MyClass extends Object{
   private int anInt = 4;
 //覆盖Object类中的toString方法
   public String toString(){
   return "Instance of MyClass. anInt = "+ anInt;
   }
  //重载 toString 方法提供其他功能
  public String toString(String prefix){
  return prefix + ":" + toString();
  }
}
```

注意!

*** 覆盖的方法的访问范围不能小于被覆盖的方法.**
***覆盖的方法抛出的异常不能多于被覆盖的方法.**
*** 子类中不能覆盖父类中声明为final的方法.**
*** 子类中必须覆盖父类中声明为abstract的方法，否则子类仍为抽象的.**
*** 子类中不能覆盖在父类中声明为static的方法.如果子类中定义了一个与父类中的类方法同名且同参数的方法，则称子类中的方法屏蔽了父类中的方法。**

举例:

```
public class parent{
  public void method(){
  }
}

public class Child extends Parent{
 private void method(){
 }
}
```

```
public class UseBoth{
  public void othermethod(){
  Parent p1 = new Parent ();
  Parent p2 = new Child();
  p1.method();
  p2.method(); // error occurred
 }
}
```

举例：

```
public class Planet {
  public static void hide(){
   System.out.println(“The hide method in Planet”);
   }
  public void override(){
    System.out.println(“The override method in Planet”);
   }
 }
```

```
public class Earth extends Planet{
    public static void hide(){
     System.out.println(“The hide method in Earth”);
    }
   public void override(){
     System.out.println("The override method in Earth");
    }
```

```
Public static void main(String [] args){
  Earth myEarth = new Earth();
  Palent myPlaent = (Planet)myEarth;
  myPlanet.hide();
  myPlanet.override();
 }
}
```

注意!
运行环境将调用编译时刻引用指向的类方法
运行环境将调用运行时刻引用指向的实例方法

5.2.2 成员变量屏蔽 (隐藏)

定义在子类中的变量如果与父类中的变量同名，即使类型不同，
也会屏蔽(隐藏)父类中定义的变量（注意!不建议这种做法）

5.2.3 使用Super关键字

在子类中，可以使用super关键字引用被屏蔽的成员变量或是被覆盖的方法.

举例:

```
//Superclass.java

public class Superclass{
    public boolean aVariable;
    public void aMethod(){
        aVariable = true;
  }
}
```

```
//Subclass.java
public class Subclass extends Superclass{
   public boolean aVariable; // 隐藏父类中的aVariable
   public void aMethod(){    // 覆盖父类中的aMethod
        aVariable = false;
        super.aMethod();
        System.out.println(aVariable);
        System.out.println(super.aVariable);
   }
   public static void main(String[] args){
      Subclass sc = new Subclass();
      sc.aMethod();
    }
}
```

也可以在构造方法中使用super调用父类中的构造方法.
注意!
***super调用必须是子类中构造方法的第1条语句.**
***如果子类对父类的构造方法没有进行任何的调用(包含super这种方式)，则运行环境将会调用父类默认的构造方法。**

举例:

```
public class Employee{
 String name;
 public Employee(String n){
    name = n;
  }
}
public class Manager extends Employee{
 String department;
 public Manager(String s, String d){
  super(s);
  department = d;
 }
}
```

5.2.4 Object类的常用方法

在java系统中每个类都是Object类的直接或间接的子类.

Object 类提供的方法:

(可被覆盖)

* clone
* equals/hashCode
* finalize
* toString

(不可被覆盖)

* getClass
* notify
* notifyAll
* wait

- **clone方法**

*** 可以使用clone方法从已经存在的对象创建出一个对象.**
aCloneableObject.clone();

***在类的声明时加入对Cloneable接口的实现即可将类变成可以被克隆的类.**

注意!

不能使用new以及构造方法来创建克隆对象而应该调用super.clone() 来进行这一操作.

举例：(克隆一个堆栈)

```
public class Stack implements Cloneable
{
   private Vector items;
   // code for Stack's methods and constructor not shown
   protected Object clone() {
      try {
         Stack s = (Stack)super.clone();   // clone the stack
         s.items = (Vector)items.clone();  // clone the vector
         return s;                         // return the clone
      } catch (CloneNotSupportedException e) {
         // this shouldn't happen because Stack is Cloneable
         throw new InternalError();
      }
   }
}
```

- **equals and hashCode** 方法

*** equals**方法比较两个对象是否相等并返回**true**或**false.**
*** hashCode**方法返回对象在哈希表中的散列值**.**
*** 覆盖equals**方法时必须同时覆盖**hashCode**方法**.**
*****必须保证同样的对象返回同样的哈希值**.**

举例:

```
Integer one = new Integer(1), anotherOne = new Integer(1);

if (one.equals(anotherOne))
    System.out.println("objects are equal");
```

•finalize 方法

finalize方法用于将一个对象在被垃圾回收前清除掉.一般情况下不需要对其进行覆盖。

•toString 方法

toString方法返回表示对象的一个字符串.

举例: System.out.println(Thread.currentThread().toString());

• getClass 方法

*** getClass方法用于返回一个对象的类的实时表示，它是一个final方法并返回一个类对象(Class object).**

举例:

```
void PrintClassName(Object obj) {
   System.out.println("The Object's class is " +
                      obj.getClass().getName());
}
```

5.2.5 终止类和终止方法(Final Classes and Methods)

* 声明为**final**的类不能派生子类.

定义一个终止类的目的在于禁止派生以增强系统安全性或获取一个更好的面向对象设计方案

```
final class ChessAlgorithm {
    . . .
}
```

* 声明为final的方法不能被子类覆盖

```
class ChessAlgorithm {
    . . .
    final void nextMove(ChessPiece pieceMoved,
                        BoardLocation newLocation)   {   ...   }
    . . .}
```

注意! 标注为static或private的方法自动设定为final方法.

*** 声名为final的变量即为常数.**

注意! 如果将一个引用类型的变量标注为final，则将无法改变该变量的引用指向，但可以改变其指向对象的内容。

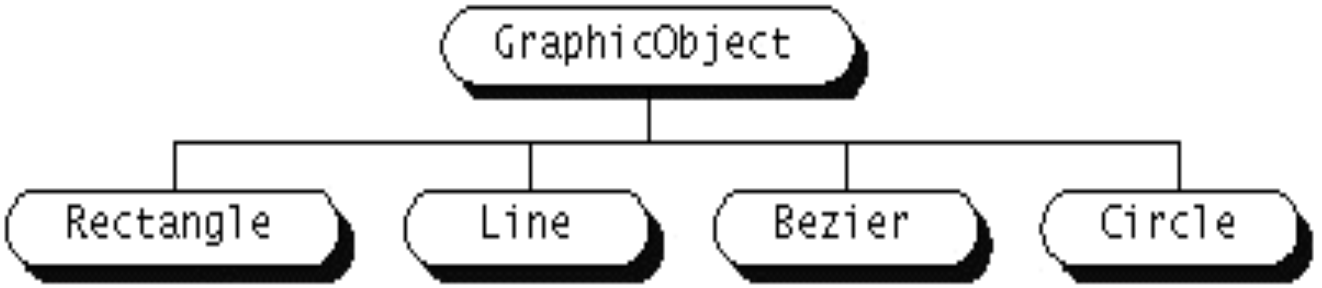
5.2.6 抽象类和抽象方法 (Abstract Classes and Methods)

*抽象类是只能被派生的类而本身不能被实例化.

```
abstract class Number {
    . . .
}
```

*抽象方法是一个没有定义任何实现的方法.抽象类可以包含抽象方法并将其部分或全部的具体实现交给子类完成。

举例:



Rectangle,Line,bezier,Circle 类从同样一个父类GraphicObject继承状态和行为

```
abstract class GraphicObject {
    int x, y;
    . . .
    void moveTo(int newX, int newY) {
        . . .
    }
    abstract void draw();
}
```

```
class Circle extends GraphicObject {
    void draw() {
        . . .
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        . . .
    }
}
```

注意!

* 不要求抽象类一定要包含抽象方法，它也可以包含非抽象的方法以及变量
* 抽象类的子类必须提供父类中所有的抽象方法的实现，否则它仍然是一个抽象类.
* 除了间接地创建抽象类的子类的实例外，不能创建抽象类的实例
* 可以声明指向抽象类的引用型变量

5.3 嵌套类(Nested Classes)

嵌套类是指一个类是另一个类的成员.

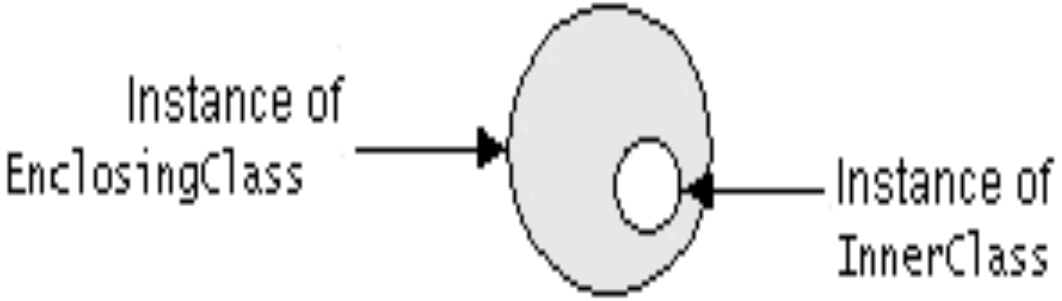
```
class EnclosingClass{
    . . .
    class ANestedClass {
        . . .
    }
}
```

静态嵌套类和内部类(非静态嵌套类)

```
class EnclosingClass{
    . . .
    static class AStaticNestedClass {
        . . .
    }
    class InnerClass {
        . . .
    }
}
.
```

注意!

* 静态嵌套类不能直接使用外部类的实例变量和方法，而应该通过对象引用进行这一操作
* 内部类的实例存在于其外部类的实例中，内部类可以直接访问其外部类中的实例变量或方法.
*内部类也可以定义为抽象类.
*内部类可以定义为被其他内部类实现的接口.
* 内部类可以声明为**private**或**protected.**
* 内部类不能声明任何静态成员



内部类的实例仅存活于外部内的实例中

举例:

java.util.Iterator 接口包含
public boolean hasNext();
public Object next();
public void remove();

例如：可以这样使用:

```
while(hasNext()){
  next();
   ...
}
```

注意： Iterator接口使用时需要满足排斥性，即不允许2个或2个以上的同时操作。

```
public class Stack {
    private Vector items;
    ...//code for Stack's methods and constructors not shown...
    public Iterator iterator() {
        return new StackIterator();
    }
```

```
class StackIterator implements Iterator {
        int currentItem = items.size() - 1;
        public boolean hasNext() { ...
                }
        public Object next() { ...
          }
        public  void remove(){ ...
         }
    }
}
```

也可以定义一个无需命名的内部类，称为匿名类

```
public class Stack {
    private Vector items;
    ...//code for Stack's methods and constructors not shown...
    public Iterator iterator() {
        return new Iterator(){
```

```
        int currentItem = items.size() - 1;
        public boolean hasNext() { ...
                }
        public Object next() { ...
          }
        public  void remove(){ ...
         }
    }
  }
}
```