

# Java 类



- **Java 的类**
  - 系统定义的类（Java APIs）
  - 用户程序自定义的类
- 类的定义
- 对象
- 类成员的修饰符



- **Java 类库**

- Java.lang

- Java.io

- Java.util

- Java.awt

- Java.awt.image

- Java.awt.peer

- Java.applet

- Java.net

- Java.corba

- Java.lang.reflect

- Java.rmi

- Java.security

- Java.util.zip

- Java.awt.datatransfer

- Java.awt.event

- Java.sql



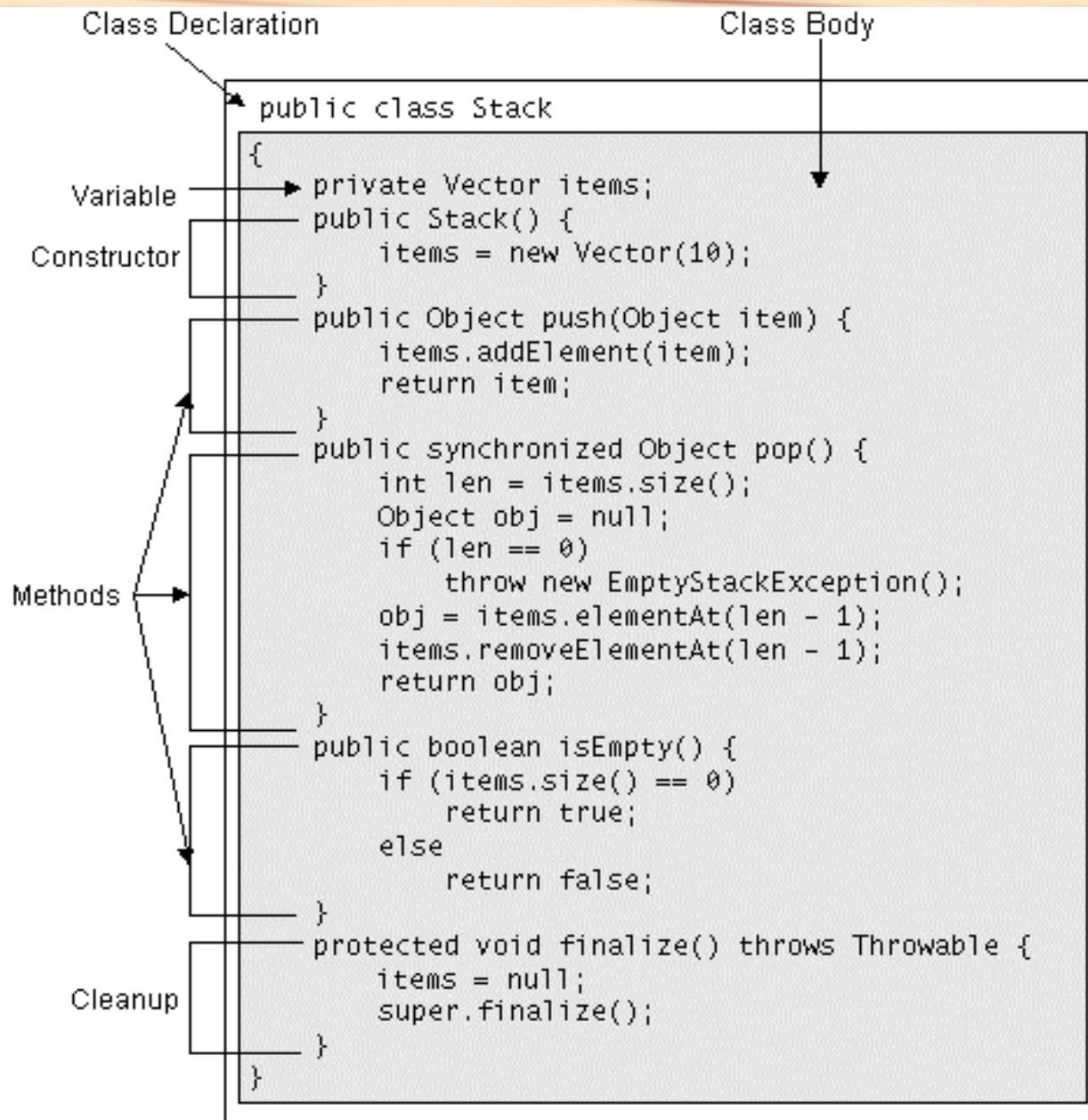
- 继承系统类
- 创建系统类的对象
- 直接使用系统类
- 如何使用?——import

例如:

```
import java.awt.*;
```

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





```
Class PhoneCard {  
    long cardNumber;  
    private int password;  
    double balance;  
    String connectNumber;  
    boolean connected;  
    boolean performConnection(long cn, int pw) {  
        if (cn == cardNumber && pw == password) {  
            connected = true;  
            return; }  
        else  
            { connected = false;  
                return false;}  
    }
```



```
double getBalance() {  
    if (connected)  
        return balance;  
    else  
        return -1;  
}
```

```
void performDial() {  
    if (connected)  
        balance -=0.5;  
}  
}
```



- 定义

[类修饰符] class 类名 [extends 超类名] [implements 接口名{,接口名}]  
{ 类体 }

- 修饰符

- 访问控制符
- 抽象类(abstract)
- 最终类(final)





- 对象变量的声明  
type objectName;
- 对象的创建：
  - New运算符: new ( object type and arguments)  
*PhoneCard myCard = new PhoneCard();*
  - 系统自动为对象分配内存空间
- 对象的初始化：
  - 采用默认的初始化值；
  - 声名变量时初始化；
  - 使用“初始化块”
  - 使用构造函数初始化



- 与方法相似，但不同之处如下：
  - 构造函数名与类名相同；
  - 构造函数没有返回类型；
  - 构造函数的作用是完成对对象的初始化工作；
  - 构造函数一般不能由编程人员显式地直接调用；
  - 在创建一个类的新对象的同时，系统会自动调用该类的构造函数为新对象初始化
- 初始化过程
  - 采用默认的初始化值(0, \u0000, false, null)
  - 声名变量时初始化
  - 使用“初始化块”
  - 构造函数



```
class Menu {  
    int i = 99;  
    Depth o = new Depth();  
    boolean b = true;  
    int j = f(i);  
    ...    }  
}
```

New an object

Call a method

```
Class classname {  
    int a; //下面是一个初始化块  
    { a = 0;  
        ...  
    } ...  
}
```

[返回](#)



1. 在一个类里变量初始化顺序由定义顺序决定的，在任何方法（包括构造器）之前得到初始化
2. 首先初始化static对象（如果它们尚未由前一次对象创建过程初始化），然后初始化非static对象

例：创建一个对象的过程：

- 类型为A的一个对象首次创建或A类的static成员首次访问时，JAVA解释器必须找到A.class
- A.class的所有初始化模块都会运行（static 初始化仅发生一次）
- 创建一个new A()时，首先在内存堆中为其分配足够空间；然后清零，将A中所有基本类型设为默认值
- 显式初始化、执行构造器

[返回](#)



```
Class Card{  
    Tag t1 = new Tag(1);  
    Card() {  
        System.out.println("Card()");  
        t3 = new Tag(33);    }  
    Tag t2 = new Tag(2);  
    void f() {System.out.println("f()"); }  
    Tag t3 = new Tag(3);  
}  
...  
Card t = new Card();  
t.f();
```

Tag(1)

Tag(2)

Tag(3)

Card()

Tag(33)

f()

[返回](#)

```
Class Bowl {  
    Bowl(int maker) {  
        System.out.println("Bowl(" + maker + ")");  
    }  
    void f(int maker) {  
        System.out.println("f(" + maker + ")");  
    }  
}
```

```
Class Table {  
    static Bowl b1 = new Bowl(1);  
    Table() {  
        System.out.println("Table()");  
        b2.f(1);  
    }  
    void f2(int maker) {  
        System.out.println("f2(" + maker + ")");  
    }  
    static Bowl b2 = new Bowl(2);  
}
```



```
Class Cupboard {  
    Bowl b3 = new Bowl(3);  
    static Bowl b4 = new Bowl(4);  
    Cupboard() {  
        System.out.println("Cupboard()");  
        b4.f(2);  
    }  
    void f3(int maker) {  
        System.out.println("f3(" + maker + ")");  
    }  
    static Bowl b5 = new Bowl(5);  
}
```



```

Public class StaticInitialization {
    public static void main(String[] args) {
        System.out.println("Creating new Cupboard() in main");
        new Cupboard() ;
        System.out.println("Creating new Cupboard() in main");
        new Cupboard() ;
        t2.f2(1);  t3.f3(1); }
    static Table t2 = new Table();
    static Cupboard t3 = new Cupboard() ; }

```

Bowl(1)	Bowl(2)	Table()	f(1)	Bowl(4)	Bowl(5)	Bowl(3)
Cupboard()	f(2)	Creating new Cupboard() in main				Bowl(3)
Cupboard()	f(2)	Creating new Cupboard() in main				Bowl(3)
Cupboard()	f(2)	f2(1)	f3(1)			

[返回](#)




**PhoneCard(long cn, int pw, double b, String s)**

```
{  cardNumber = cn;
    password = pw;
    if (b > 0)
        balance = b;
    else
        System.exit(1);
    connectNumber = s;
    connected = false;
}
```

- **PhoneCard newCard =  
new PhoneCard(12345678,1234,50.0,"300");**

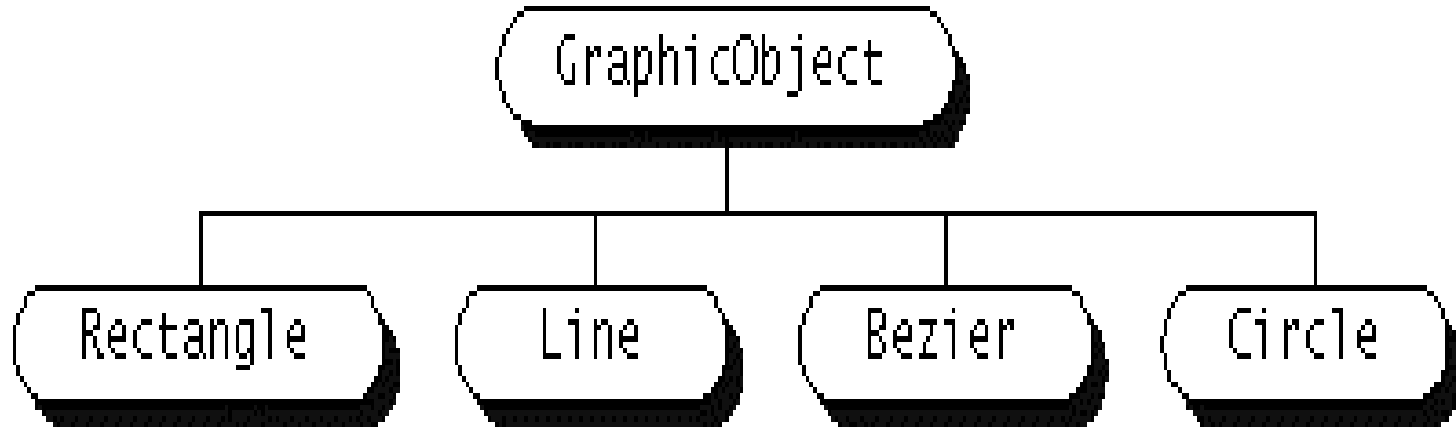


- 没有参数的构造函数称为no-arg”无参构造函数”
- 如果一个类中没有任何形式的构造函数，Java语言将提供一个不做任何事情或缺省的无参构造函数。这种构造函数只有在没有其它构造函数时才自动提供
- 如果你既需要一个无参构造函数，又需要一个或几个带参数的构造函数，则必须显式提供一个无参构造函数



- 凡是用abstract修饰符修饰的类称为抽象类，抽象类就是没有具体对象的概念类
- 不能创建一个抽象类的对象
- 抽象类仅定义了部分实现的类，而留待扩展类去提供这些方法的部分或全部的进一步实现
- 由于抽象类是它的所有子类的公共属性的集合，所以可以利用这些公共属性来提高开发和维护程序的效率





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

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



```
abstract class PhoneCard  
  
{    double balance;  
  
    abstract boolean performDial();  
  
    double getBalance()  
        { return balance; }  
  
}
```



- 如果一个类被**final**修饰符所修饰和限定，说明该类不可能有子类
- 被定义为**final**的类通常是一些有固定作用、用来完成某种标准功能的类，如Java系统定义好的用来实现网络功能的**Socket**类、**InetAddress**类等都是**final**类
- 安全性与性能优化
  - `abstract final class A{ ...} ?`



- 实例变量

- 非静态变量
- 表示每个对象的状态
- 最终变量 ( `final variable` ) : 用来声明一个常量, 必须被初始化、不能被修改

- 类变量

- 静态域: 由该类的所有对象共享
- `final static variable`



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





```
Class PhoneCard200 {  
    static String connectNumber = “200”;  
    static double additoryFee;  
    long cardNumber;  
    int password;  boolean connected;  double balance; }  
  
...  
PhoneCard200 my200_1 = new PhoneCard200();  
PhoneCard200 my200_2 = new PhoneCard200();  
My200_1.additoryFee = 0.1;  
System.out.println(“第二张200卡的附加费” + my200_2.  
    additoryFee);  
System.out.println(“200卡的附加费” +  PhoneCard200.  
    additoryFee);
```



- 静态初始化块是由关键字static引导的一对大括号括起来的语句块，它的作用与类的构造函数有些相似，但有三点不同
  - 构造函数是对每个新创建的对象初始化，而静态初始化块是对类自身进行初始化
  - 构造函数是在用new运算符产生新对象时由系统自动执行，而静态初始化块则是在它所属的类加载入内存时由系统调用执行
  - 不同于构造函数，静态初始化块不是方法

```
static
{
    nextCardNumber = 2001800001;
}
```



- 一个类的域如果被声明为**final**，则它的取值在程序的整个执行过程中都不会改变

- 注意

- 最终域用来定义常量的数据类型

- 最终域必须初始化

- 最终域在整个程序的运行过程中不能被改变

- 因为所有类对象的常量成员，其数值都固定一致，为了节省空间，常量通常声明为**static**

- 例如：

```
static final String connectNumber = "200";
```



```
Class A {  
    int i = 1;  
    public void g(); }  
Class B { final int j = 9;  
    public static final int k = 20;  
    final int m = (int)(Math.random()*20)  
    final A a1 = new A();  
    final A a2; ...  
    void q(final A x) { //JDK1.1  
        x = new A();    x.g(); }  
    public static void main(String[] args){  
        B b = new B();  
        b.a1.i++;  
        b.a1 = new A(); } ...}
```

No initializer

X is final

Can't change  
reference

✓

✓

✓

✓

X

X

✓

X



- 如果一个域被volatile修饰符所修饰，说明这个域可能同时被几个线程所控制和修改
- Exmample

```
currentValue = 5; // volatile currentValue = 5;  
for(;;){ display.showValue(currentValue);  
Thread.sleep(1000); }
```



- 修饰符1 修饰符2... 返回类型 方法名(形式参数表)

**throw [ 异常列表]**

**{ 方法体语句 ; }**

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



- 在一个类外或一个方法内定义一个方法是错误的
- 所有在方法内定义的变量或参数都是局部变量
- 参数的个数是不能改变的
- 对每个参数需要说明其类型

*Wrong: float x,y*

*Right: float x, float y*



- 方法调用的格式

**reference. method( actual-parameter-list )**

- 返回语句return

- 按值调用或引用

- 按值调用: 实现值的传递, 方法中的形式参数接受实际参数的值, 但并不能改变实际参数的值

- Example

- Simple Type Parameters
- Reference Type Parameters
- Example





```
public static void halveIt(double arg ) {
```

```
    arg /=2.0;
```

```
    System.out.println("halved:arg = " + arg);
```

```
}
```

```
.....
```

```
double one = 1.0;
```

```
System.out.println("before:one=" + one);
```

**one=1**

```
halveIt(one);
```

**arg=0.5**

```
System.out.println("after:one=" + one);
```

**one=1**

[返回](#)



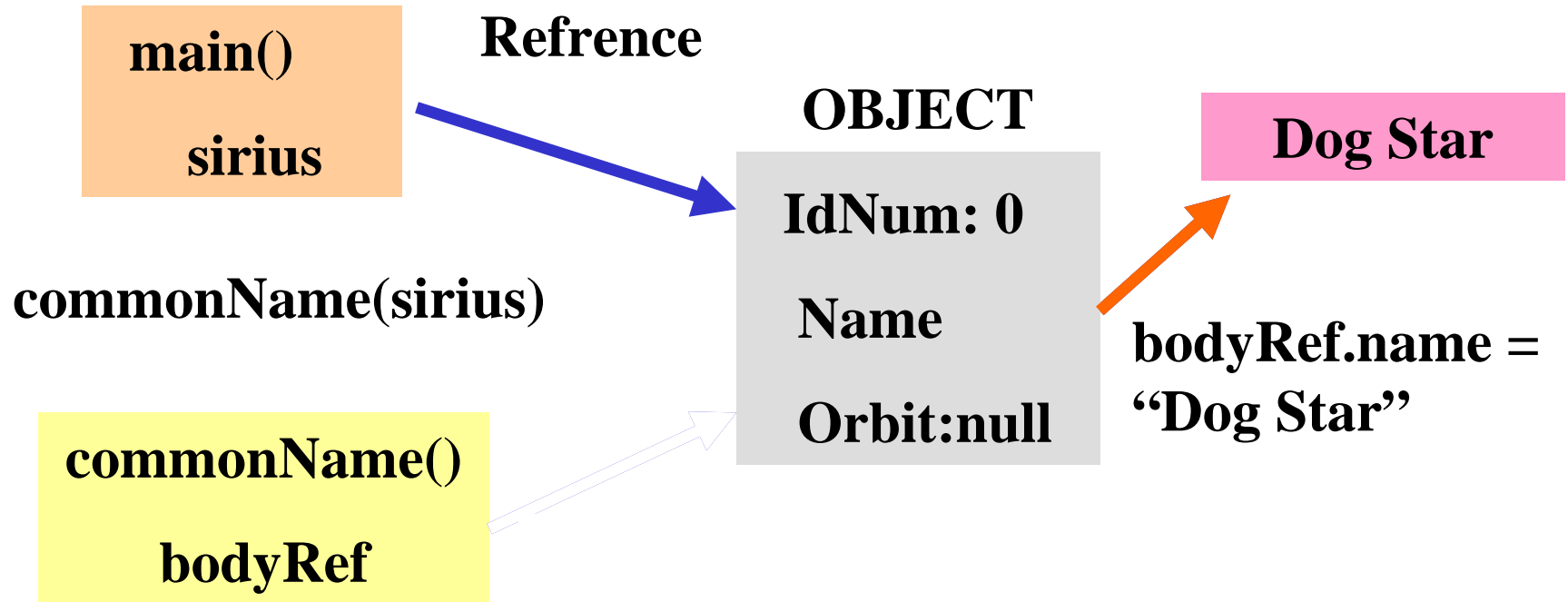
```
public static void commonName(Body bodyRef) {
    bodyRef.name = "Dog Star"; name changed
    bodyRef = null;                No effect on sirius
}

.....
```

```
Body sirius = new Body("Sirius",null);
System.out.println("before:"+sirius); before: 0(Sirius)
commonName(sirius);
System.out.println("after:"+sirius); after: 0(Dog Star)
```



**Body sirius = new Body(“Sirius”,null)**



**public static void commonName(Body bodyRef)**

**bodyRef = null**

[返回](#)



- 引用

```
class Date {
    private int year; private int month; private int day;
    int getYear(){return year;}
    int getMonth(){return month;}
    int getDay(){return day;}
    void setDate(int y, int m, int d) {
        year = y; month = m; day = d;    }
    void showDate() {
        System.out.println(year + "," + month + "," +
+day);    }
}
```



```

public class ChangeDate{
    void exchangeDate(Date x, Date y){
        Date temp = new Date();
        temp.setDate(x.getYear(), x.getMonth(), x.getDay());
        x.setDate(y.getYear(), y.getMonth(), y.getDay());
        y.setDate(temp.getYear(), temp.getMonth(), temp.getDay());
        d1.setDate(1999,12,31); d2.setDate(2000,1,1);
        d1.showDate();      d2.showDate();
        ChangeDate ex = new ChangeDate();
        ex.exchangeDate(d1,d2);
        System.out.println("d1,d2 be excahnge");
        d1.showDate();
        d2.showDate();      }
    }

```

1999,12,31

2000,1,1

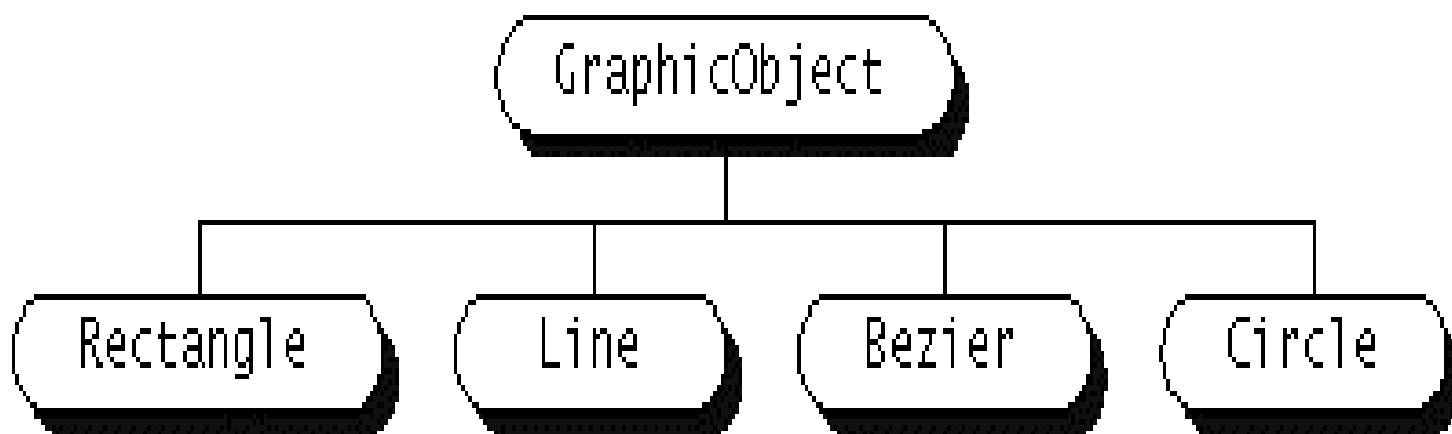
d1,d2 be  
exchanged

2000,1,1

1999,12,31

[返回](#)

- 修饰符abstract修饰的抽象方法是一种仅有方法头，没有具体的方法体和操作实现的方法
- 所有的抽象方法都必须存在于抽象类之中



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

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

- **static**修饰符修饰的方法是属于整个类的方法，其含义如下：

- 调用该方法时，应该使用类名做前缀，而不是某一个具体的对象名

- static**方法是属于整个类的，它在内存中的代码段将随着类的定义而分配和装载，不被任何一个对象专有

- static**方法只能处理**static**域

```
static void setAdditory(double newAdd)
{
    if (newAdd > 0)
        additoryFee = newAdd;
}
```



- **Final**修饰符所修饰的类方法，是功能和内部语句不能被更改的最终方法，即不能被当前类的子类重新定义的方法

- **注意**

- 所有已被**private**修饰符限定为私有的方法被缺省地认为是**final**

- 包含在**final**类中能方法被缺省地认为是**final**





## • 访问控制符

访问控制符是一组限定类、域或方法是否可以被程序里的其他部分访问和调用的修饰符

## • 访问控制符类型

modifier	class	subclass	package	world
<b>private</b>	<b>X</b>			
<b>protected</b>	<b>X</b>	<b>X*</b>	<b>X</b>	
<b>public</b>	<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>
<b>package</b>	<b>X</b>		<b>X</b>	



- public
  - 任何地方都能访问
  - 可以被子类继承
  - 公共的接口
- private
  - 只能在该类中被访问
- protected
  - 可以被子类访问或继承；可以在同一个包中被访问
- package:
  - 在同一个包中被访问



```
package Greek;  
  
public class Alpha {  
    public int iampublic;  
    public void publicMethod()  
        { System.out.println("publicMethod"); }  
}
```

```
import Greek.*;  
package Roman;  
class Beta {  
    void accessMethod() {  
        Alpha a = new Alpha();  
        a.iampublic = 10; //?  
        a.publicMethod(); //?  
    }  
}
```

legal

legal

[返回](#)



```
class Alpha {  
    private int iamprivate;  
    private void privateMethod()  
    { System.out.println("privateMethod"); }  
}
```

```
class Beta {  
    void accessMethod() {  
        Alpha a = new Alpha();  
        a.iamprivate = 10;    // ?  
        a.privateMethod();  // ?    }  
    }
```

**Beta.java:9: Variable iamprivate in class Alpha not accessible from class Beta. a.iamprivate = 10; // illegal**

**^1 error**

**Beta.java:12: No method matching privateMethod() found in class Alpha. a.privateMethod(); // illegal**

**1 error**

[返回](#)

```
package Greek;  
class Alpha {  
    protected int iamprotected;  
    protected void protectedMethod()  
        { System.out.println("protectedMethod"); }  
}
```

subclass lives in a different package

```
import Greek.*;  
package Latin;  
class Delta extends Alpha {  
    void accessMethod(Alpha a, Delta d) {  
        a.iamprotected = 10; // illegal  
        d.iamprotected = 10; // legal  
        a.protectedMethod(); // illegal  
        d.protectedMethod(); // legal    }  
}
```

in the same package

```
package Greek;  
class Gamma {  
    void accessMethod() {  
        Alpha a = new Alpha();  
        a.iamprotected = 10; // legal  
        a.protectedMethod(); // legal  
    }  
}
```



```
package Greek;  
class Alpha {  
    int iampackage;  
    void packageMethod()  
    {    System.out.println("packageMethod");    }  
}
```

```
package Greek;  
class Beta {  
    void accessMethod() {  
        Alpha a = new Alpha();  
        a.iampackage = 10;    // ?  
        a.packageMethod();    ///?  
    }  
}
```

legal

legal

[返回](#)



- **P86 : 4-7、 4-8、 4-9、 4-11**

