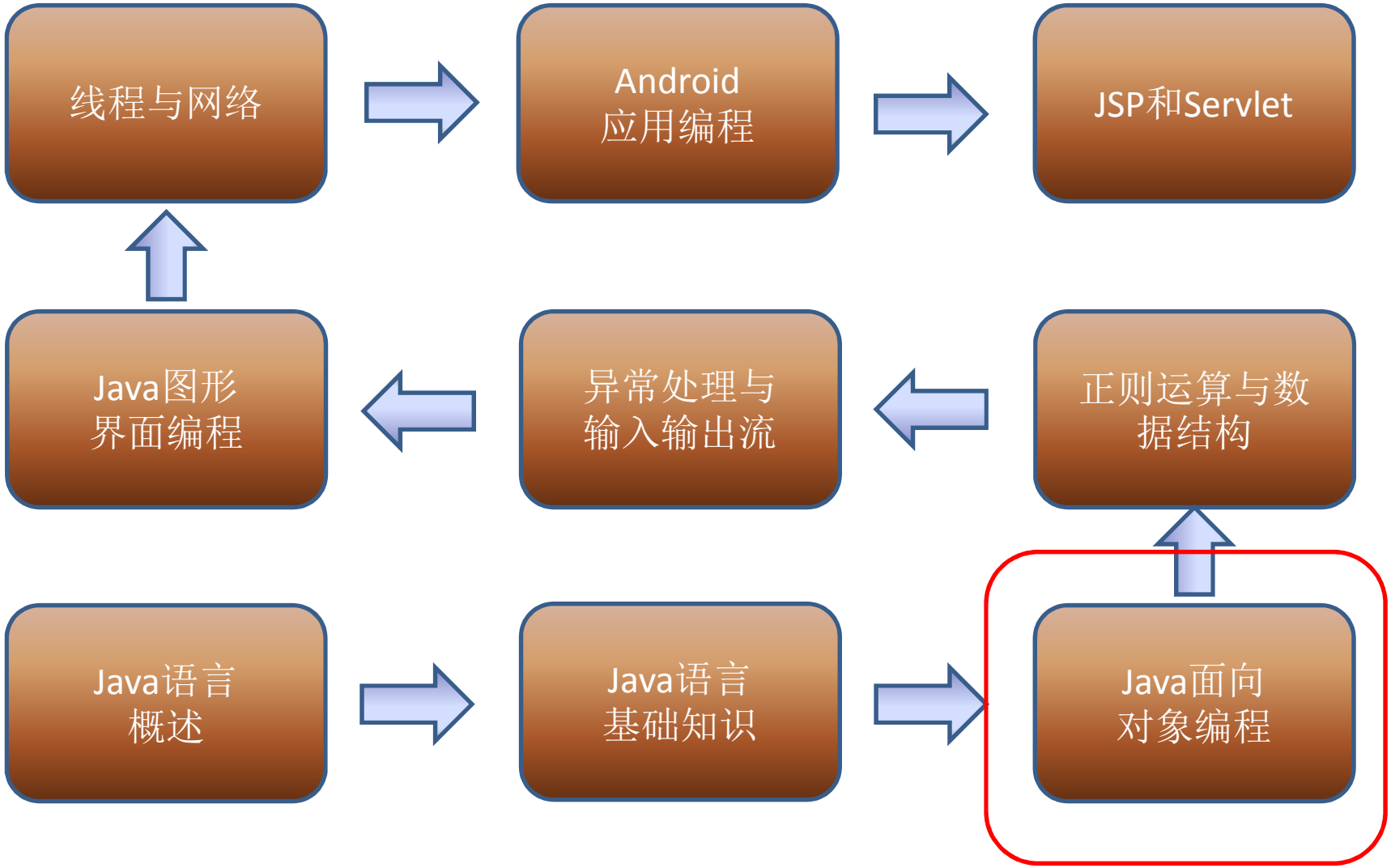

# 第三讲 Java的面向对象编程

# 课程内容安排


线程与网络
Android
应用编程
JSP和Servlet
Java图形
界面编程
异常处理与
输入输出流
正则运算与数
据结构
Java语言
概述
Java语言
基础知识
Java面向
对象编程

# 课前思考

- 面向对象有哪几个特性？
- 能否在一个类的内部定义另外一个类？
- 怎样让不同的类都拥有相同的方法？


# 学习目标

- 掌握Java中面向对象的特点和用法
- 掌握接口的概念和用法

# 面向对象的程序

面向对象的程序是由多个对象组成的系统，每个对象描述了一定的数据结构并完成一定的功能，对象之间通过互相通信（方法调用）来完成之间的协作。

# 什么 是面向对象?

- 80年代初的定义:

  面向对象是一种新兴的程序设计方法,或者是一种新的程序设计规范(paradigm),其基本思想是使用对象、类、继承、封装、消息等基本概念来进行程序设计。

- 其应用领域不仅仅是软件，还有计算机体系结构和人工智能等

# 基本思想

- 从现实世界中客观存在的事物（即对象）出发来构造软件系统，并且在系统构造中尽可能运用人类的自然思维方式。
- 开发一个软件是为了解决某些问题，这些问题所涉及的业务范围称作该软件的问题域。
- 软件开发是一种逻辑思维活动，其方法不应该是一种超越人类日常的思维方式。

# 类的基本概念

- 把众多的事物归纳、划分成一些类是人类在认识客观世界时经常采用的思维方法。分类的原则是抽象。
- 类的定义：类是具有相同属性和服务的一组对象的集合，它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和服务两个主要部分。

# 消息

- 消息就是向对象发出的服务请求，它应该包含下述信息：提供服务的对象标识、服务标识、输入信息和回答信息。

```
public class HelloWorldApp{
 public static void main(String args[]) {
     System.out.println("hello");
 }
}
```

- 服务通常被称为方法或函数。

# 主动对象

- 主动对象是一组属性和一组服务的封装体，其中至少有一个服务不需要接收消息就能主动执行（称作主动服务）。
- Java中拥有main 方法的对象就是主动对象。

# 面向对象的程序设计方法

- OOA－Object Oriented Analysis
- OOD－Object Oriented Design
- OOI －Object Oriented Implementation

# 类

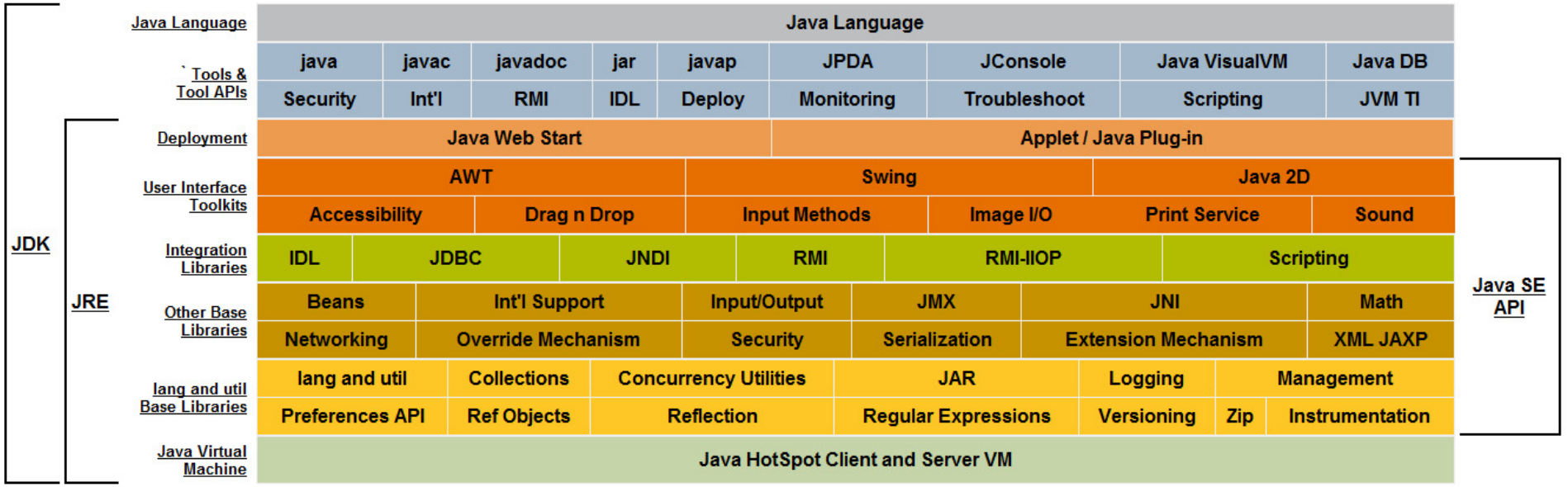
- 包括类声明和类体
- 类声明：

  [public][abstract|final] class className
  [extends superclassName]

  [implements interfaceNameList]

  {……}

# Java类库


JDK
JRE
Java Language
Java Language
` Tools & Tool APIs
java
javac
javadoc
jar
javap
JPDA
JConsole
Java VisualVM
Java DB
Security
Int'l
RMI
IDL
Deploy
Monitoring
Troubleshoot
Scripting
JVM TI
Deployment
Java Web Start
Applet / Java Plug-in
User Interface Toolkits
AWT
Swing
Java 2D
Accessibility
Drag n Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Scripting
Other Base Libraries
Beans
Int'l Support
Input/Output
JMX
JNI
Math
Networking
Override Mechanism
Security
Serialization
Extension Mechanism
XML JAXP
lang and util Base Libraries
lang and util
Collections
Concurrency Utilities
JAR
Logging
Management
Preferences API
Ref Objects
Reflection
Regular Expressions
Versioning
Zip
Instrumentation
Java Virtual Machine
Java HotSpot Client and Server VM
Java SE API

# 类java.lang.Object


- 该类定义了一些所有对象最基本的状态
- hashCode(): 对象的哈希代码
- equals() :比较两个对象(引用)是否相同
- getClass(): 返回对象所对应的类
- toString(): 将对象用字符串表示
- finalize(): 垃圾回收时调用
- notify(), notifyAll(), wait() :
- clone() : 克隆对象


```
class Cat {
    int hunger;
    void eat() {}
    void sleep() {}
    public static void main(String args[]){
            Cat c1=new Cat();
            Cat c2=new Cat();
            System.out.println(c1.hashCode());
            System.out.println(c2.hashCode());
            System.out.println(c1.toString());
            System.out.println(c2.toString());
            System.out.println(c1.equals(c2));
            System.out.println(c1.getClass());
            System.out.println(c2.getClass());
    }
}
```

# 类体


- 类体定义如下：

```
class className
{     [public | protected | private ] [static]
      [final]  [transient] [volatile] type
      variableName;   //成员变量
      [public | protected | private ] [static]
      [final | abstract] [native] [synchronized]
      returnType methodName([paramList])
      [throws exceptionList]
       {statements}          //成员方法
}
```

# 成员变量

[public | protected | private ] [static]

[final] [transient] [volatile] type

variableName; //成员变量

static: 静态变量（类变量）；相对于实例变量

final: 常量

transient: 暂时性变量，用于对象存档

volatile: 贡献变量，用于并发线程的共享

# 成员方法


[public | protected | private ] [static]
[final | abstract] [native] [synchronized]
returnType methodName([paramList])
[throws exceptionList]　　//方法声明
{statements}　　//方法体

- 方法声明

  static:　类方法，可通过类名直接调用
  abstract: 抽象方法，没有方法体
  final:　方法不能被重写
  native:　集成其它语言的代码
  synchronized:　控制多个并发线程的访问

# 成员方法(值参传递)


- 参数的值参传递:

```
public class PassTest{
    float ptValue;
    public static void main(String args[]) {
        int val;
        PassTest pt=new PassTest();
        val=11;
        System.out.println("Original Int  Value is:" +val);
```

# 成员方法(值参传递)


```
    pt.changeInt(val);
    System.out.println("Int Value after Change is:" +val);
    pt.ptValue=101f;
    System.out.println("Original ptValue is:" +pt.ptValue);
    pt.changeObjValue(pt);
    System.out.println("ptValue after Change is:" +pt.ptValue);
}
```

# 成员方法(值参传递)

```
  public void changeInt(int value){
      value=55;
  }
  public void changeObjValue(PassTest ref){
      ref.ptValue=99f;
  }
}
```

# 成员方法(值参传递)


- 运行结果

```
c:\>java PassTest
Original Int Value is : 11
Int Value after Change is: 11
Original ptValue is: 101.0
ptValue after Change is : 99.0
```

# 成员方法(方法体)

- 方法体包括局部变量的声明以及所有合法的Java语句。
- 局部变量的作用域在该方法内部。
- 若局部变量与类的成员变量同名，则类的成员变量被隐藏。

# 成员方法(方法体)

```
class Variable{
    int x=0,y=0,z=0;    //类的成员变量
    void init(int x,int y) {
    this.x=x; this.y=y;
    int z=5;                //局部变量
    System.out.println("** in init**");
    System.out.("x="+x+"  y="+y+"  z="+z)
   }
}
```

# 成员方法(方法体)


```
public class VariableTest{
    public static void main(String args[]){
        Variable v=new Variable();
        System.out.println(“**before init**”);
        System.out.println(“x=”+v.x+  “y=”+v.y+“z=”+v.z);
        v.init(20,30);
        System.out.println(“**after init**”);
        System.out.println(“x=”+v.x+  “y=”+v.y+“z=”+v.z);
    }
}
```

# 成员方法(方法体)

运行结果

```
c:\>java VariableTest
 **before init**
  x=0  y=0   z=0
 ** in init **
   x=20  y=30  z=5
  **after init**
   x=20  y=30 z=0
```

# 成员方法(方法体)


- this-----用在一个方法中引用当前对象，它的值是调用该方法的对象。
- 返回值须与返回类型一致，或是其子类
- 返回类型是接口时，返回值必须实现该接口。

# 方法重载(Method Overloading)


- 方法重载指多个方法享有相同的方法名
- 区别在于：参数类型不同，或个数不同；
- 参数类型的区分度要够，例如不能是同一简单类型
- 返回类型不能用来区分重载的方法。

```
class MethodOverloading{
    void receive(int i){
        System.out.println("Receive one int data");
        System.out.println("i="+i);
    }
    void receive(int x, int y) {
        System.out.println("Receive two int data");
        System.out.println("x="+x+"   y="+y);
    }
    void  receive(double d) {
        System.out.println("Receive one double data");
        System.out.println("d="+d);
    }
    void  receive(String s) {
        System.out.println("Receive a string");
        System.out.println("s="+s);
    }
}
```

# 方法重载


```
public class MethodOverloadingTest{
    public static void main(String args[]) {
        MethodOverloading mo=new
                MethodOverloading();
        mo.receive(1);
        mo.receive(2,3);
        mo.receive(12.56);
        mo.receive("very interesting, isn't it?");
    }
}
```

# 方法重载


- 运行结果：

```
c:\>java MethodOverloadingTest
Receive one int data
i=1
Receive two int datas
x=2   y=3
Receive one double data
d=12.56
Receive a string
s=very interesting, isn't it?
```


# 构造方法

- 构造方法具有和类名相同的名称，而且不返回任何数据类型
- 构造方法只能由new运算符调用
- 利用构造方法进行初始化
- 重载经常用于构造方法

# 构造方法

```
class Point{
   int x,y;
   Point(){
        x=0; y=0;
   }
   Point(int x, int y){
        this.x=x; this.y=y;
  }
}
```

# 对象


- 类实例化可生成对象
- 对象通过消息传递来进行交互
- 消息传递即激活指定的某个对象的方法以改变其状态或让它产生一定的行为；表现为调用该对象的某个方法。

```
public class HelloWorldApp{
    public static void main(String args[]) {
        System.out.println("hello");
    }
}
```

# 对象的生命周期

- 生成
- 使用
- 清除

# 对象的生成


- 包括声明、实例化和初始化
- 格式：

  type objectName=new type([paramlist]);

(1)声明：type objectName

声明并不为对象分配内存空间，而只是分配一个引用空间；

(2)实例化：运算符new为对象分配内存空间，它调用对象的构造方法，返回引用；一个类的不同对象分别占据不同的内存空间。

# 对象的生成

(3)生成：执行构造方法，进行初始化；根据参数不同调用相应的构造方法。

- 对象的引用指向一个中间的数据结构，它存储有关数据类型的信息以及当前对象所在的堆的地址，而对于对象所在的实际的内存地址是不可操作的，这就保证了安全性。

# 对象的使用

- 通过运算符“.”可以实现对变量的访问和方法的调用。
- 设定访问权限来限制其它对象对它的访问

# 调用对象的变量

- 格式： objectReference.variable
- objectReference是一个已生成的对象，也可以是能生成对象的表达式
- 例： p.x= 10;

  tx=new Point( ).x;

# 调用对象的方法


- 格式：
  objectReference.methodName([paramlist]);
- 例如： p.move(30,20);
  new Point( ).move(30,20);
- 使用方法的返回值

```
if (p.equals(20,30))  {
      …    //statements when equal
    } else
      …    //statements when unequal
}
```

# 对象的清除

- 当不存在对一个对象的引用时，该对象成为一个无用对象。
- System.gc( );

# 面向对象特性

- 封装性
- 继承性
- 多态性

# 封装性


- 类定义：

```
class className
   {     [public | protected | private ] [static]
          [final]  [transient] [volatile] type
          variableName;   //成员变量
          [public | protected | private ] [static]
          [final | abstract] [native] [synchronized]
          returnType methodName([paramList])
          [throws exceptionList]
           {statements}          //成员方法
   }
```

# 成员变量和成员方法的访问权限

| | 同一个类 | 同一个包 | 不同包中的子类 | 不同包非子类 |
|---|---|---|---|---|
| private | * | | | |
| default | * | * | | |
| protected | * | * | * | |
| public | * | * | * | * |

# 使类和成员的可访问能力最小化


- 一个设计良好的模块会隐藏所有的实现细节，包括内部数据和其他细节。信息隐藏可以有效地解除各个模块之间的耦合关系
- 尽可能使每一个类或者成员不被外界访问。public类需要永远支持它，以保持兼容性。非public类实际上是包私有的（default），可以对它进行修改、替换、或者去除。
- public类应用尽可能少地包含public的成员域（field）。


# 继承性

- 通过继承实现代码复用
- 根类： java.lang.Object
- 父类包括直接或者间接被继承的类
- Java不支持多重继承
- 子类可以重写父类的方法，及命名与父类同名的成员变量

# 创建子类


- 格式：

```
class SubClass extends SuperClass {
    …
}
```

# 成员变量的隐藏和方法的重写


```
class SuperClass{
    int x;  …
    void setX( ){   x=0; }  …
}
class SubClass extends SuperClass{
    int x;   //hide x in SuperClass
    …
    void  setX( ) {  //override method setX()
            x=5;  }    ….
 }
```

# 重写

- 子类中重写的方法和父类中被重写的方法要具有相同的名字，相同的参数表和相同的返回类型

# super

- super用来引用当前对象的父类

(1) 访问父类被隐藏的成员变量，如：

```
super.variable;
```

(2)调用父类中被重写的方法，如：

```
super.Method([paramlist]);
```

(3)调用父类的构造函数，如：

```
super([paramlist]);
```


```
class SuperClass{
    int x;
    SuperClass( ) {
    x=3;
    System.out.println("in SuperClass : x=" +x);
 }
  void doSomething( ) {
    System.out.println("in SuperClass.doSomething()");
  }
}
```


```
class SubClass extends SuperClass {
    int x;
    SubClass( ) {
        super( );    //call constructor of superclass
        x=5;       //super( ) 要放在方法中的第一句
        System.out.println("in SubClass  :x="+x);
    }
void doSomething( ) {
    super.doSomething( ); //call method of superclass
    System.out.println("in SubClass.doSomething()");
```


```
 System.out.println("super.x="+super.x+"sub.x="+x);
 }
}
public class Inheritance {
    public static void main(String args[]) {
    SubClass subC=new SubClass();
    subC.doSomething();
   }
}
```


- 运行结果

c:\> *java Inheritance*

in SuperClass: x=3

in SubClass:  x=5

in SuperClass.doSomething()

in SubClass.doSomething()

super.x=3  sub.x=5

# 多态性

- 静态多态性（编译时多态）
  由方法重载实现
- 动态多态性（运行时多态）
  由方法重写实现
  调用规则：根据实例的类型

# 重写方法的调用规则


```
class A{
    void callme( ) {
        System.out.println("Inside A's callme() method");}
  }
class B extends A{
  void callme( ) {
      System.out.println("Inside B's callme() Method");}
 }
```


```
public class Dispatch{
    public static void main(String args[]) {
        A a=new B();
        a.callme( );
     }
}
```

# 重写方法的调用规则


- 运行结果

c:\> *java Dispatch*

*Inside B's callme() method*

# 方法重写应遵循的原则

(1)重写后的方法不能比被重写的方法有更严格的访问权限

(2)重写后的方法不能比被重写的方法产生更多的例外

# 方法重写应遵循的原则


```
class Parent{
    public void function( ){ }  }
class Child extends Parent{
    private void function( ) {}   }
public class OverriddenTest{
    public static void main(String args[])  {
        Parent p1=new Parent();
        Parent p2=new Child();
        p1.function();
        p2.function();  }
 }
```

# 对象状态的确定（instanceof)


- Manager和Contractor都是Employee的子类

```
public void method(Employee e) {
      if  (e instanceof  Manager)
            …
        }  else if ( e instanceof Contractor)  {
              …  }
              else  {
                    …
              }
}
```

# final关键字


(1) final修饰变量,则成为常量

final type variableName;

修饰成员变量时，定义时同时给出初始值，而修饰局部变量时不做要求。

(2)final修饰方法，则该方法不能被子类重写

```
final returnType methodName(paramList){
  …
}
```


# final关键字

(3)final修饰类，则类不能被继承

```
final class finalClassName{
    …
}
```

# 实例成员和类成员


- 类成员的格式：

```
static type classVar;
static  returnType classMethod({paramlist]) {
    …
}
```

# 实例变量和类变量

- 每个对象的实例变量都分配内存
- 通过对象来访问实例变量
- 所有实例对象共享同一个类变量。
- 类变量可通过类名直接访问，无需先生成一个实例对象
- 也可以通过实例对象访问类变量

# 实例方法和类方法


- 类方法不能访问实例变量，只能访问类变量
- 类方法可以由类名直接调用
- 类方法中不能使用this或super

# 实例方法和类方法

```
class Member {
   static int classVar;
   int instanceVar;
   static void setClassVar(int i)  {
       classVar=i;
  // instanceVar=i;    //can't access nostatic member
   }
```


```
static int getClassVar() {
    return classVar;
}
void setInstanceVar(int i){
    classVar=i;   //can access static member
    instanceVar=i;
 }
 int getInstanceVar( ) {
    return instanceVar;
 }
}
```

```
public class MemberTest{
    public static void main(String args[]) {
        Member m1=new Member();
        Member m2=new Member();
        m1.setClassVar(1);
        m2.setClassVar(2);
    System.out.println("m1.classVar="+m1.getCalssVar()+
        "m2.ClassVar="+m2.getClassVar());
        m1.setInstanceVar(11);
        m2.setInstanceVar(22);
System.out.println("m1.InstanceVar="+m1.getInstanceVar()+
    "m2.InstanceVar="+m2.InstanceVar());
  }
}
```

# 实例方法和类方法


- 运行结果

c:\> *java memberTest*

m1.classVar=2　　m2.classVar=2

m1.InstanceVar=11　m2.InstanceVar=22

# Inner Class

- 内部类:在一个类的内部嵌套定义类

(1)是其它类的成员

(2)在一个语句块的内部定义

(3)在表达式内部匿名定义

- Inner Class 一般用来生成事件适配器,用于事件处理 。


```
class Outer{
    private int size;/* 声明类Outer的成员变量，其初值默认为0*/
    public class Inner{ //声明内部类
        public void doStuff(){ //内部类的方法
            size++; //存取其外部类的成员变量
        }
    }
```


```
    public void testInner(){ //类Outer的实例成员方法
        Inner i=new Inner(); //建立内部类Inner的对象i
        i.doStuff(); //通过i调用内部类Inner的成员方法
        System.out.println(size);
    }
        public static void main(String args []){
        Outer o=new Outer();
        o.testInner();
    }
}
```

# 运行结果

1

# 抽象类


- 抽象类是用abstract来修饰的类，例如

```
public abstract class GraphicObject {
        int x,y;
        abstract void draw();
        abstract void moveTo(double deltaX, double deltaY);
}
```

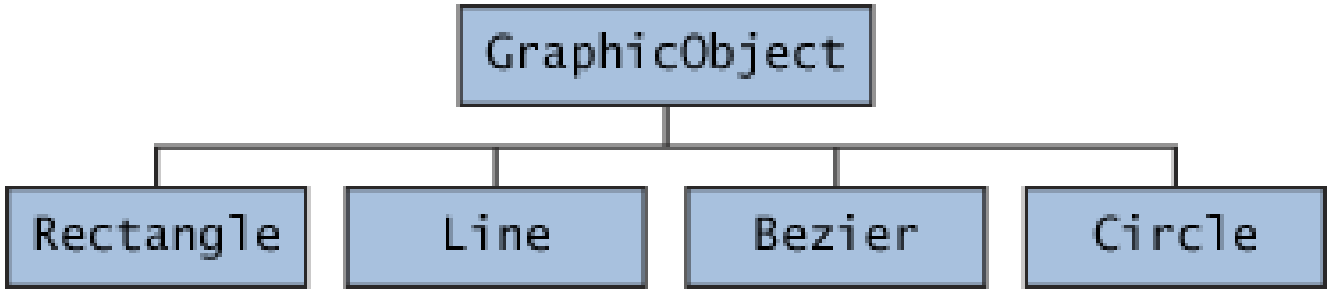
- 抽象方法是没有实现（方法体）的方法，用abstract来修饰，例如

```
        abstract void draw();
```


- 抽象类不能被实例化，一般通过子类继承抽象类，并重写所有的抽象方法；然后可以实例化子类
- 若类中包含了抽象方法，则该类必须被定义为抽象类

# 抽象类的作用

- 定义一些类的共同父类




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

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

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


# 接口

- 接口是抽象类，只包含常量和方法的定义，而没有变量和方法的实现

# 接口

- 用处

(1)通过接口实现不相关类的相同行为,而无需考虑这些类之间的关系.

(2)通过接口指明多个类需要实现的方法

(3)通过接口了解对象的交互界面,而无需了解对象所对应的类

# 接口定义


- 包括接口声明和接口体
- 完整的接口声明:

[public] interface interfaceName[extends listOfSuperInterface] { … }

- 接口体包括常量定义和方法定义

常量定义: type NAME=value; 该常量被实现该接口的多个类共享; 缺省的具有public ,final, static的属性.

方法体定义:(缺省的具有 public和abstract属性)

returnType methodName([paramlist])

# 接口定义

- 例

```
interface MouseListener extends EventListener{
    void mouseClicked(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
}
```

# 接口的定义


- 利用关键字extends，可以把多个接口组合成为一个接口。


```
interface A{
    int a=1;
    void showa();
}

interface B{
    int b=2;
    void showb();
}
```


```
interface C{
    int c=3;
    void showc();
}

interface D extends A,B,C{
    int d=4;
    void showd();
}
```


```
class ABCDE implements D{
    int e=5;
    public void showa(){
        System.out.println("a="+a);
    }
    public void showb(){
        System.out.println("b="+b);
    }
    public void showc(){
        System.out.println("c="+c);
    }
```

```
    public void showd(){
        System.out.println("d="+d);
    }

    /** 本类中定义showe()方法 */
    public void showe(){
        System.out.println("e="+e);
    }
}
```


```
public class ABCDETest{
    public static void main(String a[]){
        D abcde=new ABCDE();
        abcde.showa();
        abcde.showb();
        abcde.showc();
        abcde.showd();
    }
}
```


# 接口的实现

- 在类的声明中用implements子句来表示一个类实现了某个接口
- 一个类可以实现多个接口,在implements子句中用逗号分开
- 必须实现接口中定义的所有方法


```
interface Shape{
    int position_x=50,position_y=50; //中心
    double PI=3.14159;
    void draw(); //绘制图形
}

interface Area{
    double area();//计算图形面积
}
```


```
class Square implements Shape,Area{ //正方形类
    private int length;
    void setSize(int l){ //给定边长
            length=l;
    }
    public void draw(){
            System.out.println("This is a square.");
            /*以(position_x,position_y)为中心，画出一个
    边长是length的正方形*/
   }
    public double area(){ //求正方形面积
            return length*length;
    }
}
```

```
class Circle implements Shape,Area{
    private int radius;
     void setSize(int r){ //给定半径
         radius=r;
     }
     public void draw(){
        System.out.println("This is a circle.");
         /*以(position_x,position_y)为中心，画出一个半径是radius的圆*/
      }
    public double area(){ //求圆的面积
         return PI*radius*radius;
    }
}
```

```
class Trigon implements Shape,Area{
    private int bottom;
    private int highness;
    void setSize(int b,int h){ //给定底和高
        bottom=b;
        highness=h;
    }
    public void draw(){
        System.out.println("This is a trigon.");
        //以(position_x,position_y)为重心，画出一个
        //底为bottom，高为highness的三角形
    }
    public double area(){ //求三角形的面积
        return 0.5*bottom*highness;
    }
}
```

```
public class DemoOfSimpleFigure{
    public static void main(String args[]){
        Square sq=new Square();
        Circle ci=new Circle();
        Trigon tr=new Trigon();
        sq.setSize(5);
        sq.draw();
        System.out.println("Area of the square is "+sq.area());
        ci.setSize(2);
        ci.draw();
        System.out.println("Area of the circle is "+ci.area());
        tr.setSize(5,7);
        tr.draw();
        System.out.println("Area of the trigon is "+tr.area());
    }
}
```

# 运行结果

This is a square.

Area of the square is 25.0

This is a circle.

Area of the circle is 12.56636

This is a trigon.

Area of the trigon is 17.5

# 接口类型的使用


- 作为一种引用类型来使用
- 任何实现该接口的类的实例都可以存储在该接口类型的变量中
- 通过这些变量可以访问类所实现的接口中的方法

# Multi-Inheritance by Interface: An example


```
interface ISpeaker {
    public void speak( );
}
```

```
class People{
    …
}
```

```
class IOSApp{
    …
}
```

```
class Ape{
    …
}
```

# Implementing Interface

```
class  Lecturer extends People implements ISpeaker {
        public void speak(){
            System.out.println("Let's learn Java!");
        }
}
```

```
class  EvolutionApe extends Ape implements ISpeaker {
        public void speak(){
            System.out.println("Caesar is home!");
        }
}
```

```
class  Siri extends IOSApp implements ISpeaker {
        public void speak(){
            System.out.println("I found 50 beautiful girls, 3 of them are fairly close to
you.");
        }
}
```

# Using interface

```
public static void main (String[] args) {
    Set<Speaker> speakers = getAllSpeakers();
    for (Speaker s : speakers) {
        s.speak();
    }
}
```


# 本讲小结

- 类与对象
- Object类和Java类库结构
- 方法重写和方法重载
- 类变量和类方法
- 成员变量和成员方法的访问权限
- 内部类
- 抽象类
- 接口


# 谢谢！