

# Java面向对象特性

孙聪

网络与信息安全学院

2019-09-16

# 课程内容

- Java概述
- 面向对象程序设计概念
- Java语言基础
- Java面向对象特性
- Java高级特征
- 容器类
- 常用预定义类
- 异常处理
- 输入输出
- 线程

# 提要

- ① 对象与类
- ② 访问权限控制
- ③ 对象生命周期
- ④ 类的继承与多态

# 提要

- 1 对象与类
- 2 访问权限控制
- 3 对象生命周期
- 4 类的继承与多态

# 对象与类

## 概念回顾

- 在OOP中，类是一个可扩展的程序代码模板，用于创建一类对象并为对象提供成员变量初始值和方法实现
- 类是将同一类对象都具有的数据和行为进行封装所形成的复合数据类型
  - 类是同种对象的集合与抽象
  - 定义和描述了一类对象的共同的数据特征和行为特征
  - 类是创建对象的一种模板，对象是类的具体实例

# 类定义

类定义 = 类声明 + 类体

```
类声明 { //类体  
    成员变量声明;  
    构造方法声明;  
    成员方法声明;  
    初始化程序块;  
}
```

# 类声明

类声明 { //类体

成员变量声明;  
构造方法声明;  
成员方法声明;  
初始化程序块;

}

- [public] [abstract|final] class 类名 [extends 父类名]  
[implements 接口名列表] {...}
  - public: 用来声明任意类均可访问该类
  - 缺省（非public）：只有与该类在同一包中的类可访问该类
  - 一个Java源文件中至多只能有一个public类，该类的类名应与该源文件的文件名相同

# 类声明

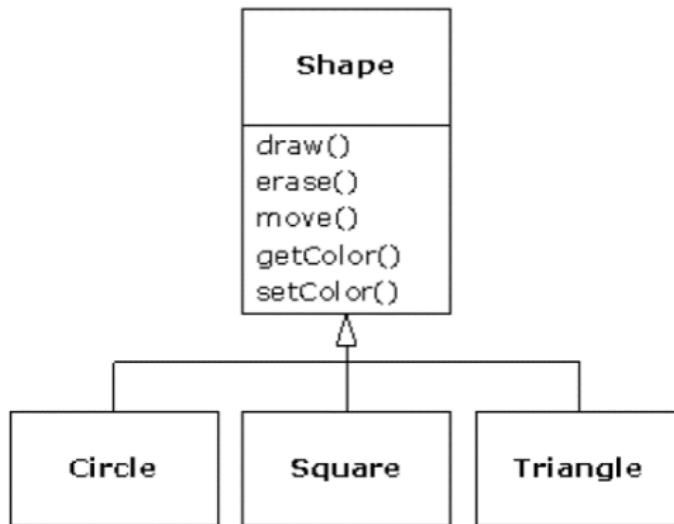
类声明 { //类体

成员变量声明;  
构造方法声明;  
成员方法声明;  
初始化程序块;

}

- [public] [abstract|final] class 类名 [extends 父类名]  
[implements 接口名列表] {...}
  - abstract: 声明该类为抽象类
  - final: 声明该类不能被继承
  - extends: 表示该类从父类继承得到。父类名可以是用户自定义的类，也可以是Java预定义的类
  - implements: 该类实现了接口名列表中的所有接口

- 根据以下继承关系图，在 MyClassDecl.java 文件中声明所有的类，并保证：
  - Shape 类不能生成任何对象实例
  - Triangle 类不能被其他类继承



# 成员变量声明

类声明 { // 类体

    成员变量声明;

    构造方法声明;

    成员方法声明;

    初始化程序块;

}

# 成员变量声明

类声明 //类体

成员变量声明；

构造方法声明；

成员方法声明；

初始化程序块；

}

- [public|protected|private] [static] [final] [transient] [volatile] 类型 变量名；
  - public|protected|private: 成员变量的访问权限
  - static: 限制该成员变量为类变量（无static: 该成员变量为实例变量）
  - final: 用于声明一个常量，该常量的值在程序中不能修改
  - transient: 声明一个暂时性变量（不会被串行化）
  - volatile: 禁止编译器通过保存多线程共享变量的多个拷贝来提高效率

# 成员变量声明

类声明 //类体

```
成员变量声明;  
构造方法声明;  
成员方法声明;  
初始化程序块;  
}
```

- 显式初始化：成员变量声明中可以包含简单的赋值表达式，这种在声明成员变量时对变量的显式赋值称为显式初始化

## 例

```
class Rectangle{  
    private int width = 0;  
    private int height = 0;  
    ...  
}
```

# 成员方法声明

类声明 //类体

    成员变量声明；

    构造方法声明；

**成员方法声明；**

    初始化程序块；

}

- [public|protected|private] [static] [final|abstract]  
[native] [synchronized] 返回类型 方法名([参数列表])  
[throws 异常类型列表] { [语句块] }

- public, protected, private, static: 与成员变量类似
- abstract: 声明该方法是抽象方法，无方法体
- final: 声明该方法不能被重写
- native: 本地方方法
- synchronized: 多线程访问共享数据
- throws: 抛出异常的列表

# 方法调用中的参数传递

- 基本要求一：实参与形参在次序和数量上匹配，并在类型上兼容

# 方法调用中的参数传递

- 基本要求一：实参与形参在次序和数量上匹配，并在类型上兼容
- 基本要求二：实参传递给形参的过程，是值传递（pass-by-value）的过程
  - 基本类型的参数：传值
  - 对象或数组等引用类型的参数：传递实参变量的值（一个指向对象或数组的引用，传递过后形参和实参均指向同一个对象或数组）

# 方法调用中的参数传递

- 基本要求一：实参与形参在次序和数量上匹配，并在类型上兼容
- 基本要求二：实参传递给形参的过程，是值传递（pass-by-value）的过程
  - 基本类型的参数：传值
  - 对象或数组等引用类型的参数：传递实参变量的值（一个指向对象或数组的引用，传递过后形参和实参均指向同一个对象或数组）
- 注意，在方法体中
  - 将形参指向另一对象，则实参变量所指向的对象不再受方法影响
  - 修改形参所指向的对象的内容，则修改在方法结束后保留下

# 方法调用中的参数传递

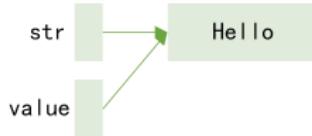
```
public class PassTest {  
    public void changeInt(int value) { // 基本类型的参数  
        value = 55;  
    }  
    public static void main(String args[]) {  
        int val;  
        PassTest pt = new PassTest();  
  
        val = 11;  
        pt.changeInt(val); // 基本类型参数的值传递  
        System.out.println("Int value is: " + val);  
    }  
}
```

# 方法调用中的参数传递

```
public class PassTest {  
    public void changeStr(String value) { // 引用类型参数  
        value = new String("different"); //方法中改变形参所指对象  
    }  
    public static void main(String args[]) {  
        String str;  
        PassTest pt = new PassTest();  
  
        str = new String("Hello");  
        pt.changeStr(str); // 引用类型参数的传递  
        System.out.println("Str value is: " + str);  
    }  
}
```

# 方法调用中的参数传递

```
public class PassTest {  
    public void changeStr(String value) { // 引用类型参数  
        value = new String("different"); // 方法中改变形参所指对象  
    }  
    public static void main(String args[]) {  
        String str;  
        PassTest pt = new PassTest();  
  
        str = new String("Hello");  
        pt.changeStr(str); // 引用类型参数的传递  
        System.out.println("Str value is: " + str);  
    }  
}
```



# 方法调用中的参数传递

```

public class PassTest {
    public void changeStr(String value) { // 引用类型参数
        value = new String("different"); // 方法中改变形参所指对象
    }
    public static void main(String args[]) {
        String str;
        PassTest pt = new PassTest();

        str = new String("Hello");
        pt.changeStr(str); // 引用类型参数的传递
        System.out.println("Str value is: " + str);
    }
}

```



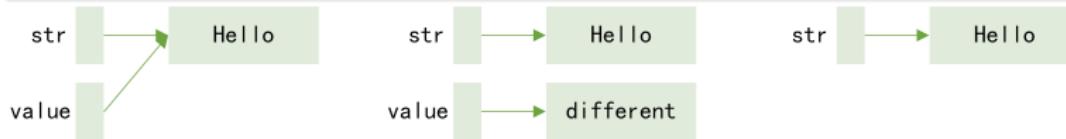
# 方法调用中的参数传递

```

public class PassTest {
    public void changeStr(String value) { // 引用类型参数
        value = new String("different"); // 方法中改变形参所指对象
    }
    public static void main(String args[]) {
        String str;
        PassTest pt = new PassTest();

        str = new String("Hello");
        pt.changeStr(str); // 引用类型参数的传递
        System.out.println("Str value is: " + str);
    }
}

```



# 方法调用中的参数传递

```
public class PassTest {  
    float ptValue;  
    public void changeObjValue(PassTest ref) { // 引用类型参数  
        ref.ptValue = 99.0f; //改变参数所指对象的成员变量值  
    }  
    public static void main(String args[]) {  
        PassTest pt = new PassTest();  
  
        pt.ptValue = 101.0f;  
        pt.changeObjValue(pt); // 引用类型参数的传递  
        System.out.println("Pt value is: " + pt.ptValue);  
    }  
}
```

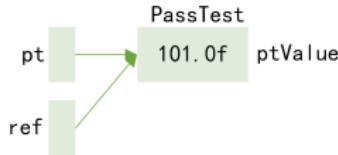
# 方法调用中的参数传递

```

public class PassTest {
    float ptValue;
    public void changeObjValue(PassTest ref) { // 引用类型参数
        ref.ptValue = 99.0f; // 改变参数所指对象的成员变量值
    }
    public static void main(String args[]) {
        PassTest pt = new PassTest();

        pt.ptValue = 101.0f;
        pt.changeObjValue(pt); // 引用类型参数的传递
        System.out.println("Pt value is: " + pt.ptValue);
    }
}

```



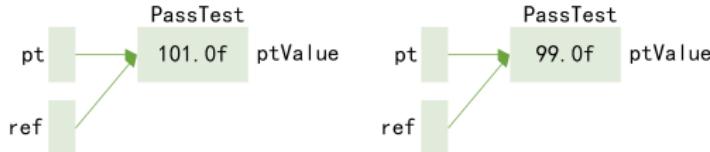
# 方法调用中的参数传递

```

public class PassTest {
    float ptValue;
    public void changeObjValue(PassTest ref) { // 引用类型参数
        ref.ptValue = 99.0f; // 改变参数所指对象的成员变量值
    }
    public static void main(String args[]) {
        PassTest pt = new PassTest();

        pt.ptValue = 101.0f;
        pt.changeObjValue(pt); // 引用类型参数的传递
        System.out.println("Pt value is: " + pt.ptValue);
    }
}

```



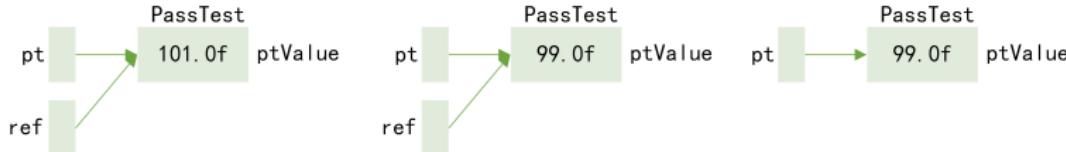
# 方法调用中的参数传递

```

public class PassTest {
    float ptValue;
    public void changeObjValue(PassTest ref) { // 引用类型参数
        ref.ptValue = 99.0f; // 改变参数所指对象的成员变量值
    }
    public static void main(String args[]) {
        PassTest pt = new PassTest();

        pt.ptValue = 101.0f;
        pt.changeObjValue(pt); // 引用类型参数的传递
        System.out.println("Pt value is: " + pt.ptValue);
    }
}

```



# 在方法声明时，还有以下问题需要注意

- 可变长参数：类型 ... 参数名
  - 方法内部将可变长参数作为数组对待
  - 可变长参数只能作为方法参数列表中最后一个参数

```
public class VariousArgs {  
    public double ratingAverage(double r, int ... points) {  
        int sum=0;  
        for(int p: points)  
            sum+=p;  
        return ((sum*r)/points.length);  
    }  
    public static void main(String[] args) {  
        VariousArgs va=new VariousArgs();  
        System.out.println( va.ratingAverage(0.5, new int[]{95, 90, 85}) );  
        System.out.println( va.ratingAverage(0.5, 94, 92, 90, 88, 86));  
    }  
}
```

# 在方法声明时，还有以下问题需要注意

- 若局部变量名与类的成员变量名相同，则类的成员变量被隐藏，需使用this将该成员变量显露出来

```
public class UnmaskField{  
    private int x = 1;  
    private int y = 1;  
    public void changeFields(int a, int b) {  
        x = a;          //x指成员变量  
        int y = b;      //局部变量y使同名的类成员变量被隐藏  
        this.y = 8;     //this.y指成员变量  
        System.out.println("x=" + x + " ; y=" + y); //局部变量y的值  
    }  
    public void PrintFields() { System.out.println("x=" + x + " ; y=" + y); }  
    public static void main(String args[]) {  
        UnmaskField uf = new UnmaskField();  
        uf.PrintFields();  
        uf.changeFields(10, 9);  
        uf.PrintFields();  
    }  
}
```

# this关键字的用法

- 在成员方法内部，this表示对“调用方法的那个对象”的引用
  - 方法形参或局部变量与成员变量同名时，成员变量被隐藏，通过“this. 成员变量名”使用成员变量
  - this可作为返回值，返回对当前对象的引用。例：  
ReturnedThis.java
- 在构造方法内部，通过this调用当前类的另一个构造方法（见“构造方法重载”）

# 构造方法声明

```
类声明 { // 类体  
    成员变量声明;  
    构造方法声明;  
    成员方法声明;  
    初始化程序块;  
}
```

- [public|protected|private] 类名 ([参数列表]) { [语句块] }

- 例：

```
class Circle{  
    private double radius;  
    public Circle(double r){ radius = r; }  
}
```

# 构造方法的特点

- 构造方法的名称必须与类名相同
- 构造方法没有返回值，这与返回值为void不同
- 构造方法在创建一个对象时调用，调用时必须使用关键字new

# 默认构造方法

- 对于没有定义构造方法的类，Java编译器会自动加入一个特殊的构造方法，该构造方法**没有参数且方法体为空**，称为默认构造方法
  - 用默认构造方法初始化对象时，对象的成员变量初始化为默认值
  - 若类中已定义了构造方法，则编译器不再向类中加入默认构造方法

```
class Bird {  
    int i;  
    // public Bird(int j){ i=j; }  
}  
  
public class DefaultConstructor {  
    public static void main(String[] args) {  
        Bird nc = new Bird(); // 默认构造方法  
    }  
}
```

## 习题3-1

- 定义EmpInfo类，要求：
  - 在 MyClassAndObj. java 文件中进行定义
  - 包含三个成员变量： name, designation, department，均为 String 类型
  - 包含非默认构造方法
  - 包含成员方法print()：用于输出成员变量内容
- 定义EmpInfo类型的对象，要求：
  - 对象变量名为 emp1, emp2
  - 设置两个对象的成员变量值分别为：
    - emp1: “Robert Java”, “Manager”, “Coffee shop”
    - emp2: “Tom Java”, “Worker”, “Coffee shop”
  - 调用 print() 查看输出

# 提要

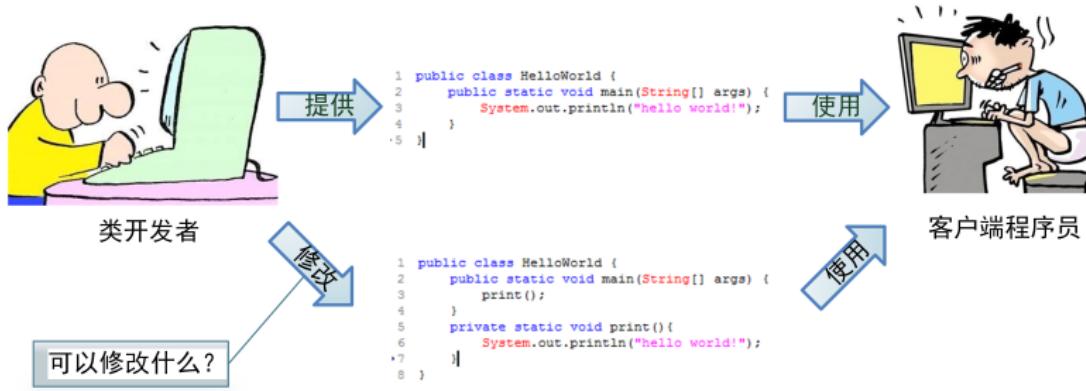
① 对象与类

② 访问权限控制

③ 对象生命周期

④ 类的继承与多态

# 封装的意义



# 访问权限控制

- 基本问题：“如何把变动的事物与保持不变的事物区分开”？
- 对于类库，要求
  - 类库开发者有权限进行修改和改进
  - 客户代码不会因为类库的改动而受到影响
- 解决方法
  - 权限修饰符：类库开发者向类外部（客户程序员）指定哪些可用，哪些不可用
  - `public | protected | private | (default/package)`

# 包 (package)

相关类与接口的一个集合，提供了类的命名空间的管理和访问保护

- 访问权限控制的**前提之一**
- 包机制的好处：
  - 能够对类和接口按照其功能进行组织
  - 每个包都提供独立的命名空间，不同包中的同名类之间不会冲突
  - 同一个包中的类之间有比较宽松的访问权限控制

# 包的定义

- package pkg1[. pkg2[. pkg3...]];

- 定义从当前源代码编译出的.class文件应放在哪个包存储路径中
- 包存储路径：“包的根路径\pkg1\...\pkgn”
- 包的根路径由CLASSPATH环境变量指出
- 包分隔符“.”相当于目录分隔符

# 包的定义

- `package pkg1[. pkg2[. pkg3…]];`
  - 定义从当前源代码编译出的.class文件应放在哪个包存储路径中
  - 包存储路径：“包的根路径\pkg1\...\pkgn”
  - 包的根路径由CLASSPATH环境变量指出
  - 包分隔符“.”相当于目录分隔符
- 注意
  - 包定义语句在每个源程序中只能有一条，即一个类只能属于一个包
  - 包定义语句必须在程序代码的第一行（除注释之外）

# 包成员

- 包成员：包中的类和接口
- 包成员的访问方法

- 第1步：`import pkg1[. pkg2[. pkg3 ...]]. (类名|*);`
- `pkg1[. pkg2[. pkg3 ...]]` 表示包层次，对应于文件目录
- “类名”指明所要引入的类
- 通配符`*`表示引入该包中的所有类
- `import`语句须在源程序所有类声明之前，`package`语句之后

```
[package ...]      //默认是package .;
[import ...]        //默认是import java.lang.*;
[类声明]
```

- 
- 第2步：使用包成员
    - 使用短名（类名）引用包成员
    - 使用长名（“包名.类名”）引用包成员（过于繁琐，  
一般仅当两个包中含有同名类时，为区分同名类才使用）
  - 例：`PackageImport.java`

# 权限修饰符

- public|protected|private|(default)

	同一个类	同一个包	子类	全局	成员变量	方法	类
private	✓				✓	✓	
default	✓	✓			✓	✓	✓
protected	✓	✓	✓		✓	✓	
public	✓	✓	✓	✓	✓	✓	✓

# 权限修饰符

- public|protected|private|(default)

	同一个类	同一个包	子类	全局	成员变量	方法	类
private	✓				✓	✓	
default	✓	✓			✓	✓	✓
protected	✓	✓	✓		✓	✓	
public	✓	✓	✓	✓	✓	✓	✓

- public权限的成员可以被任何类访问

# 权限修饰符

- public|protected|private|**(default)**

	同一个类	同一个包	子类	全局	成员变量	方法	类
private	✓				✓	✓	
default	✓	✓			✓	✓	✓
protected	✓	✓	✓		✓	✓	
public	✓	✓	✓	✓	✓	✓	✓

- default**权限的成员可以被它所在的包中的所有类访问

## pkg1/A.java

```

package pkg1;
public class A{
    public A() { System.out.println("A's constructor"); }
    void mtd() { System.out.println("A's method"); } //mtd()为包权限
}
class B{//B为包权限
    public B() { System.out.println("B's constructor"); }
}

```

## pkg2/PublicVsPackage.java

```

package pkg2; //pkg2与pkg1不是同一个包
import pkg1.*;
public class PublicVsPackage{
    public static void main(String[] args) {
        A obj=new A();
        //B obj2=new B();      //在pkg1之外不能创建B的对象
        //obj.mtd();          //在pkg1之外不能访问mtd()方法
    }
}

```

# 权限修饰符

- public|protected|private|(default)

	同一个类	同一个包	子类	全局	成员变量	方法	类
private	✓				✓	✓	
default	✓	✓			✓	✓	✓
protected	✓	✓	✓		✓	✓	
public	✓	✓	✓	✓	✓	✓	✓

- private权限的成员变量，成员方法：除包含该成员的类之外，其他类均无法访问该成员
- private权限的构造方法：其他类不能生成该类的实例对象
- 同一个类的不同对象之间可互相访问私有成员

## pkg2/PrivateVsPackage.java

```
package pkg2;
public class PrivateVsPackage{
    public static void main(String[] args) {
        C obj=new C(); //默认构造方法的权限与类的访问权限相同
        obj.mtd2();
        //obj.mtd1(); //与C在同一包中的其他类无法访问mtd1()
    }
}
class C{
    private void mtd1() {
        System.out.println("C's method 1");
    }
    void mtd2() {
        System.out.println("C's method 2");
        this.mtd1(); //同一个类内可以调用mtd1()
    }
}
```

# 权限修饰符

- public | **protected** | private | (default)

	同一个类	同一个包	子类	全局	成员变量	方法	类
private	✓				✓	✓	
default	✓	✓			✓	✓	✓
<b>protected</b>	✓	✓	✓		✓	✓	
public	✓	✓	✓	✓	✓	✓	✓

- protected**权限的成员，可以被它所在的包中的所有类访问
- protected**权限的成员，可以被它所属类的子类访问  
(不论子类是否在同一包中)

## pkg1/C.java

```
public class C{
    protected void func() { System.out.println("Protected method of C"); }
}
```

## pkg2/ProtectedVsPackageAndPublic.java

```
import pkg1.C;
public class ProtectedVsPackageAndPublic{
    public static void main(String[] args) {
        C obj=new C();
        //! obj.func(); //不是C的子类，且与C非同一个包
    }
}
class CSub extends C{ //C的子类，可以访问C的func()方法
    void mtd(C parent, CSub sub) {
        func();
        //! parent.func(); //应通过子类引用而非父类引用访问func()
        sub.func();
    }
}
```

写程序证明：protected权限包含package权限

- 同一包中的其他类是否能访问protected权限的成员？

写程序证明：protected权限大于package权限

- package权限的成员在包外子类中是否能够访问？

# 包相关的其他问题

- 引入其他类的静态成员

- 程序在使用其他类的static final的常量或static的方法时，每次引用这些常量或方法都用它们所属的类名
- 例：java.lang.Math类定义了常数PI以及静态方法cos()，需用  
`double r=Math.cos(Math.PI * theta);`

- 简化方法：`import static [静态常量名|静态方法名]*;`

- 例：

```
import static java.lang.Math.PI;
import static java.lang.Math.cos;
```

或

```
import static java.lang.Math.*;
```

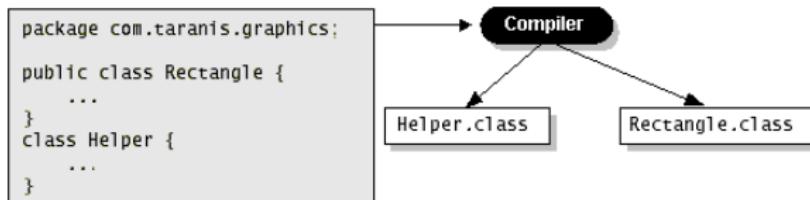
然后直接使用：

```
double r = cos (PI*theta);
```

- 例：ImportStatic.java

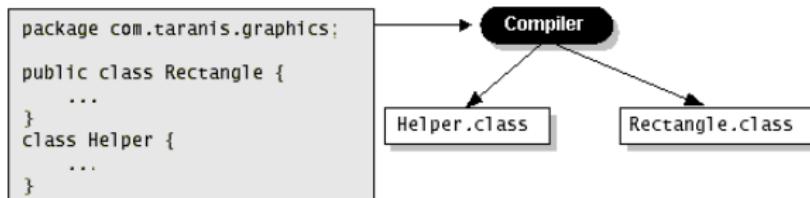
# 源代码文件(.java)与字节码文件(.class)的管理

- 每个类（而非每个.java文件）对应一个.class文件

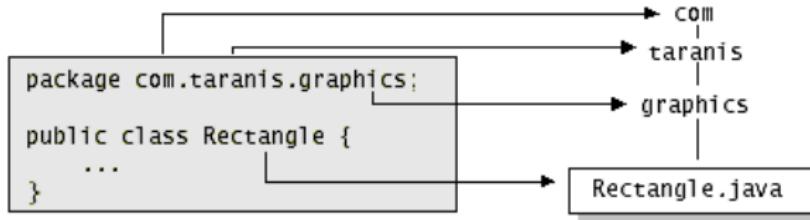


# 源代码文件(.java)与字节码文件(.class)的管理

- 每个类（而非每个.java文件）对应一个.class文件

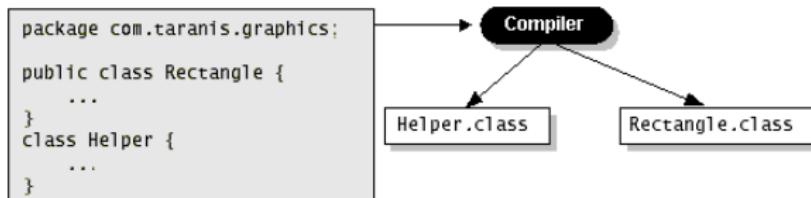


- 源文件可以按照包名指明的路径放置。如

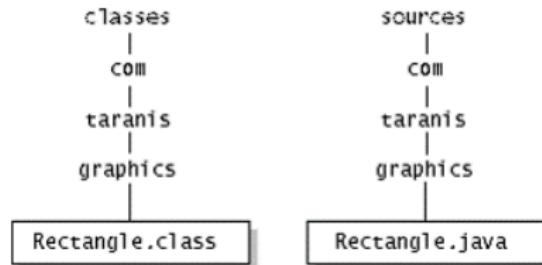


# 源代码文件(.java)与字节码文件(.class)的管理

- 每个类（而非每个.java文件）对应一个.class文件



- 可将源文件与类文件分别存放（部分IDE支持），采用如下方式



# 提要

1 对象与类

2 访问权限控制

3 对象生命周期

4 类的继承与多态

# 对象生命周期

对象的创建 → 对象的使用 → 对象的清除

# 对象生命周期

对象的创建 → 对象的使用 → 对象的清除

# 对象生命周期

对象的创建 → 对象的使用 → 对象的清除

## 步骤

- 声明对象变量：类名 变量名；
- 对象的实例化：new 类名( [参数列表] )；

## 对象实例化的过程

- 为对象分配存储空间，并用默认值对成员变量初始化
- 执行显式初始化，即执行成员变量声明时的赋值
- 执行构造方法的主体，完成对象初始化
- 返回该对象的引用

# 对象的创建

```
public class Point {  
    public int x = 2;  
    public int y;  
  
    public Point(int y) {  
        this.y = y;  
    }  
    public static void main(String[] args) {  
        Point p = new Point(22);  
    }  
}
```

# 对象的创建

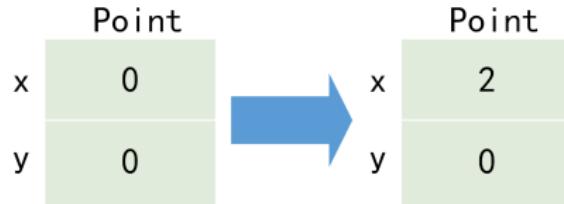
```
public class Point {  
    public int x = 2;  
    public int y;  
  
    public Point(int y) {  
        this.y = y;  
    }  
    public static void main(String[] args) {  
        Point p = new Point(22);  
    }  
}
```

Point

x	0
y	0

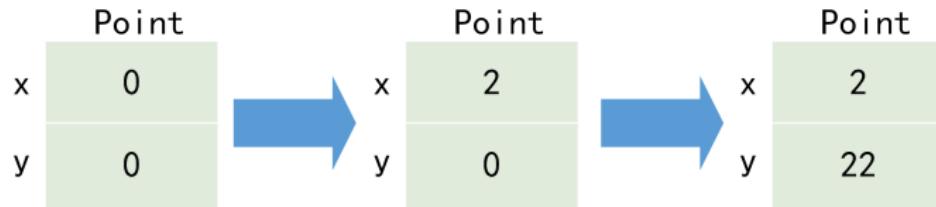
# 对象的创建

```
public class Point {  
    public int x = 2;  
    public int y;  
  
    public Point(int y) {  
        this.y = y;  
    }  
    public static void main(String[] args) {  
        Point p = new Point(22);  
    }  
}
```



# 对象的创建

```
public class Point {  
    public int x = 2;  
    public int y;  
  
    public Point(int y) {  
        this.y = y;  
    }  
    public static void main(String[] args) {  
        Point p = new Point(22);  
    }  
}
```



写出以下程序的输出

```
class Window {  
    Window(int m) { System.out.println("window " + m); }  
}  
  
class House {  
    Window w1 = new Window(1);  
    House() {  
        System.out.println("House");  
        w3 = new Window(33);  
    }  
    Window w2 = new Window(2);  
    void f() { System.out.println("f()"); }  
    Window w3 = new Window(3);  
}  
  
public class OrderOfInit {  
    public static void main(String[] args) {  
        House h = new House();  
        h.f();  
    }  
}
```

# 初始化程序块

```
类声明 { // 类体  
    成员变量声明;  
    构造方法声明;  
    成员方法声明;  
    初始化程序块;  
}
```

# 初始化程序块

```
类声明 { // 类体  
    成员变量声明;  
    构造方法声明;  
    成员方法声明;  
    初始化程序块;  
}
```

- 分类

- 实例初始化程序块
- 静态初始化程序块

- 实例初始化程序块的执行时机

- 在分配对象存储空间和使用类型默认值初始化之后
- 在构造方法执行之前
- 与显式初始化的优先级相当，先后关系依赖于实例化方法在代码中的调用顺序

```
public class InstanceInitializer {  
    Comp c1=new Comp(1);  
    public InstanceInitializer() {  
        System.out.println("Instance Initializer");  
    }  
    Comp c2;  
    public static void main(String[] args) {  
        InstanceInitializer ii=new InstanceInitializer();  
    }  
    Comp c3=new Comp(3);  
    { //实例初始化程序块  
        c2=new Comp(2);  
    }  
}  
class Comp {  
    public Comp(int i){ System.out.println("Comp(" + i + ")"); }  
}
```

# 对象生命周期

对象的创建 → 对象的使用 → 对象的清除

- 通过圆点运算符(.)
- 注意成员变量和方法的访问权限

# 对象生命周期

对象的创建 → 对象的使用 → 对象的清除

- 垃圾收集机制 (garbage collection)：  
当Java运行环境确定某个对象不再被使用时，将其删除
- 一个对象在没有引用指向它时，可作为垃圾收集
- Java运行环境中的垃圾收集器周期性地释放不用的对象占用的空间
- 垃圾收集器在对象被收集前调用对象的`finalize()`方法
- 某些情况下可通过调用`System.gc()`建议JVM执行垃圾收集

# 提要

1 对象与类

2 访问权限控制

3 对象生命周期

4 类的继承与多态

# 类的继承

- 继承：类之间的“is a”关系
- 反映出一个类（子类）是另一个类（父类）的特例，  
继承、修改和细化父类的状态和行为

```
public class Employee {           public class Manager {  
    String name;                 String name;  
    Date hireDate;               Date hireDate;  
    Date dateOfBirth;           Date dateOfBirth;  
    String jobTitle;             String jobTitle;  
    int grade;                   int grade;  
    ...                           String department;  
}                                Employee[] subordinates;  
                                  ...  
}
```

# 类的继承

- 继承是从已有的类创建新类的一种方法
  - 子类继承父类的**所有**成员变量和方法，子类中只需声明特有的东西
  - **class 子类名 extends 父类名 { ... }**

```
public class Employee {           public class Manager extends Employee{  
    String name;  
    Date hireDate;  
    Date dateOfBirth;  
    String jobTitle;  
    int grade;  
    ...  
}  
                                String department;  
                                Employee[] subordinates;  
                                ...  
}
```

# 类的继承

## 注意

- 继承了父类的变量和方法不代表在子类中一定可以访问它们：  
父类的private的成员变量和方法无法在子类中访问。  
**考虑：父类的package权限的成员呢？**
- 创建一个类总会继承其他的类
  - 显式的继承使用extends关键字
  - 如果没有显式地指明父类，则从Java的标准根类Object进行继承

# 对父类成员的访问——super关键字

- 用法1：表示“当前对象的父类对象的引用”，用来引用父类中的成员变量或方法

```
public class TestInheritance{  
    public static void main(String[] args) {  
        Rectangle rect=new Rectangle();  
        rect.newDraw();  
    }  
}  
class Shape{  
    public void draw() { System.out.println("Draw shape"); }  
}  
class Rectangle extends Shape{  
    public void draw() { System.out.println("Draw Rectangle"); }  
    public void newDraw(){  
        draw();  
        super.draw();  
    }  
}
```

# 对父类成员的访问——super关键字

- Rectangle中的draw()方法与Shape中的draw()方法是什么关系？
- 若Rectangle中没有实现自己的draw()方法，则程序输出是什么？

```
public class TestInheritance{  
    public static void main(String[] args) {  
        Rectangle rect=new Rectangle();  
        rect.newDraw();  
    }  
}  
  
class Shape{  
    public void draw() { System.out.println("Draw shape"); }  
}  
  
class Rectangle extends Shape{  
    public void draw() { System.out.println("Draw Rectangle"); }  
    public void newDraw(){  
        draw();  
        super.draw();  
    }  
}
```

# 对父类成员的访问——super关键字

- 用法2：在子类构造方法中，通过“`super([参数列表]);`”调用父类的构造方法。该语句必须出现在子类构造方法的第一行

# 子类对象的创建与初始化的一般步骤

- 子类构造方法调用父类构造方法
  - 通过显式的super()方法调用或编译器隐含插入的super()方法调用
  - 这一过程递归地进行，直到根类Object的构造方法。在这一过程中，子类对象所需的所有内存空间被分配，所有成员变量均使用默认值初始化
- 从根类Object的构造方法开始，自顶向下地对每个类依次执行如下两步：
  - 显式初始化及使用实例初始化程序块进行初始化
  - 执行构造方法的主体（不包括使用super()调用父类构造方法的语句）

```
public class ConstructSubObj{  
    public static void main(String[] args) {  
        Undergraduate ug=new Undergraduate(12345678);  
    }  
}  
  
class Person{  
    Person() { System.out.println("Person"); }  
}  
  
class Student extends Person{  
    Student(int id) { System.out.println("Student "+id); }  
}  
  
class Undergraduate extends Student{  
    Undergraduate(int id) {  
        super(id); //必须使用，因为student没有默认构造方法  
        System.out.println("Undergraduate");  
    }  
}
```

```
class Meal { Meal() { System.out.println("Meal()"); } }
class Bread { Bread() { System.out.println("Bread()"); } }
class Cheese { Cheese() { System.out.println("Cheese()"); } }
class Lettuce { Lettuce() { System.out.println("Lettuce()"); } }
class Lunch extends Meal {
    Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch {
    PortableLunch() { System.out.println("PortableLunch()"); }
}
public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() { System.out.println("Sandwich()"); }

    public static void main(String[] args) {
        new Sandwich();
    }
}
```

回顾对象的实例化过程步骤：

- 为对象分配存储空间，并用默认值对成员变量初始化
- 执行显式初始化，即执行成员变量声明时的赋值
- 执行构造方法的主体，完成对象初始化
- 返回该对象的引用

能否将其归纳到子类的实例化过程？

# 多态

- 编译时多态
  - 重载 (overloading)
- 运行时多态
  - 重写 (overriding)
  - 向上转型 (upcasting)
  - 动态绑定 (dynamic binding)

# 多态

- 编译时多态
  - 重载 (overloading)
- 运行时多态
  - 重写 (overriding)
  - 向上转型 (upcasting)
  - 动态绑定 (dynamic binding)

# 重载

一个类中定义多个名称相同但参数不同的方法

```
class Screen {  
    public void print( int i){ ... }  
    public void print( float f){ ... }  
    public void print( String str ){ ... }  
}
```

# 重载

一个类中定义多个名称相同但参数不同的方法

```
class Screen {  
    public void print( int i){ ... }  
    public void print( float f){ ... }  
    public void print( String str ){ ... }  
}
```

- 重载必须遵循的原则
  - 方法的参数表（包括参数的类型或个数）必须不同
  - 方法的返回类型、修饰符可以相同也可以不同

# 重载

## 重载方法的使用

```
public class TestOverloading{  
    public static void main(String[] args) {  
        System.out.println(false); //println(boolean b)  
        System.out.println('C'); //println(char c)  
        System.out.println(123); //println(int i)  
        System.out.println(123L); //println(long l)  
        System.out.println(12.3f); //println(float f)  
        System.out.println(12.3); //println(double d)  
        System.out.println(); //println(): 输出换行符  
        char[] cc={'a', 'b', 'c'};  
        System.out.println(cc); //println(char[] lc)  
        System.out.println("abc"); //println(String s)  
        System.out.println(new java.util.Date()); //println(Object o)  
    }  
}
```

# 重载

## 重载构造方法的使用

- 如果一个类定义了多个构造方法(目的是使对象具有不同初始值), 则可以在一个构造方法中调用另一个构造方法
- 使用this()调用当前类的构造方法
- 当一个构造方法使用this()调用另一构造方法时, 对this的调用必须位于此构造方法的起始处
- 对this()和super()的调用具有互斥性。当一个构造方法使用this()调用另一构造方法时, 构造方法起始处不再有对super()的调用

```
public class ConstructorOverloading{  
    public static void main(String[] args) {  
        Student stu=new Student();  
        System.out.println(stu.getName() + " , " + stu.getID());  
    }  
}  
  
class Student{  
    private String name;  
    private String id;  
    public Student(String nm, String id) {  
        this.name=nm;  
        this.id=id;  
    }  
    public Student(String nm) { this(nm, "00000000"); }  
    public Student() { this("Unknown"); }  
    public String getName() { return name; }  
    public String getID() { return id; }  
}
```

## 习题3-12：为什么将方法重载称为“编译时多态”？

```
public class OverloadTest{  
    public void m(String s){  
        System.out.println("m(String)");  
    }  
    public void m(Object o){  
        System.out.println("m(Object)");  
    }  
    public static void main(String[] args){  
        Object o=new String("hello");  
        OverloadTest t=new OverloadTest();  
        t.m(o);  
    }  
}
```

# 多态

- 编译时多态
  - 重载 (overloading)
- 运行时多态
  - 重写 (overriding)
  - 向上转型 (upcasting)
  - 动态绑定 (dynamic binding)

# 成员方法重写

- 在子类中修改父类方法的实现
- 方法重写的前提：存在继承关系
- 方法重写的要求
  - 不改变方法的名称、参数列表和返回值，改变方法的内部实现
  - 子类中重写方法的访问权限不能缩小
  - 子类中重写方法不能抛出新的异常

# 多态

- 编译时多态
  - 重载 (overloading)
- 运行时多态
  - 重写 (overriding)
  - 向上转型 (upcasting)
  - 动态绑定 (dynamic binding)

# 向上转型

- 子类对象既可以作为该子类的类型对待，也可以作为其父类的类型对待
- 向上转型（Upcasting）：将子类对象的引用转换成父类对象的引用
  - 如：`Employee e=new Manager();`
  - 通过该变量（如e）只能访问父类（Employee）的方法，子类（Manager）特有的部分被隐藏
  - 又例：`Triangle.java`

向上转型使得数组可以包含不同类型的对象

```
Employee[] staff = new Employee[3];
staff[0] = new Manager();
staff[1] = new Secretary();
staff[2] = new Employee();
```

# 多态

- 编译时多态
  - 重载 (overloading)
- 运行时多态
  - 重写 (overriding)
  - 向上转型 (upcasting)
  - 动态绑定 (dynamic binding)

# 动态绑定

- 绑定 (binding) : 将方法调用和方法体联系在一起
- 动态绑定: 绑定操作在程序运行时根据变量指向的对象实例的具体类型找到正确的方法体(而不是根据变量本身的类型)

# 动态绑定

- 绑定 (binding) : 将方法调用和方法体联系在一起
- 动态绑定：绑定操作在程序运行时根据变量指向的对象实例的具体类型找到正确的方法体(而不是根据变量本身的类型)

方法重写，向上转型，动态绑定的关系

- 由于存在Upcasting，父类变量可能指向父类的对象，也可能指向子类的对象
- 相应地，通过父类变量发出的方法调用，可能执行该方法在父类中的实现，也可能执行该方法在某子类中的实现，具体执行哪个实现由动态绑定决定
- 重写是运行时多态的基础，重写关系下的两个方法的实现不同才使得“多态”有意义

# 动态绑定

```
Employee e;  
...           //内部多种实现：继承、重写、向上转型  
e.getDetails(); //对外一个接口  
...
```

- 例：shapes/TestShapes.java

# 动态绑定

- Java中除了static方法和final方法 (private方法属于final方法) , 其余方法都采用动态绑定

写出以下程序的输出结果

```
public class PrivOverride {  
    private void f() { System.out.println("private f()"); }  
    public static void main(String[] args) {  
        PrivOverride po = new Derived();  
        po.f();  
    }  
}  
class Derived extends PrivOverride {  
    public int a;  
    public void f() { System.out.println("public f()"); }  
}
```

# 成员变量隐藏

- 概念：父类中成员变量被子类中同名的成员变量隐藏
- 成员变量隐藏和成员方法重写的前提
  - 父类成员能够在子类中被访问到
  - 父类中private成员，在子类中不能被隐藏（重写）

习题3-13：成员变量的隐藏是否具有“运行时多态”？

```
public class Sub extends Super {  
    String s= "sub" ;  
    public static void main(String[] args) {  
        Super sup=new Sub();  
        System.out.println(sup.s);  
    }  
}  
class Super {  
    String s= "super" ;  
}
```

```
public class NotOverriding extends Base{  
    private int i=2;  
    public static void main(String[] args) {  
        NotOverriding no=new NotOverriding();  
        no.increase();  
        System.out.println(no.i);  
        System.out.println(no.getI());  
    }  
}  
class Base{  
    private int i=100;  
    public void increase(){  
        this.i++;  
    }  
    public int getI(){  
        return this.i;  
    }  
}
```

# 向下转型与运行时类型识别

- 父类弱、子类强，父类变量不能直接按子类引用，必须强制转换才能作为子类的引用使用
- 向下转型（downcasting）：将父类类型的引用变量强制（显式）地转换为子类类型
- 如何保证向下转型的正确性？
  - 先测试父类变量引用的对象确实为子类类型，再执行转换：

```
if (supVar instanceof SomeSubClass)  
    (someSubClass) supVar;
```

- 对强制转换结果进行运行时类型识别，若结果类型与子类类型不符，则抛出ClassCastException

```

public class Downcasting {
    public static void main(String[] args) {
        BaseClass[] b={new BaseClass(), new SubClass()};
        b[0].f();
        b[1].f();
        //! b[1].g(); //b[i]为BaseClass类型，只有f()接口方法
        ((SubClass)b[1]).g();
        //! ((SubClass)b[0]).g(); //抛出运行时异常
        for(int i=0;i<b.length;i++) {
            if(b[i] instanceof SubClass)
                ((SubClass)b[i]).g();
        }
    }
}
class BaseClass{
    public void f() { System.out.println("Base. f()"); }
}
class SubClass extends BaseClass{
    public void f() { System.out.println("Sub. f()"); }
    public void g() { System.out.println("Sub. g()"); }
}

```