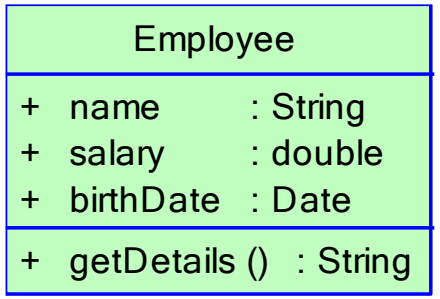
第11章 继承和多态

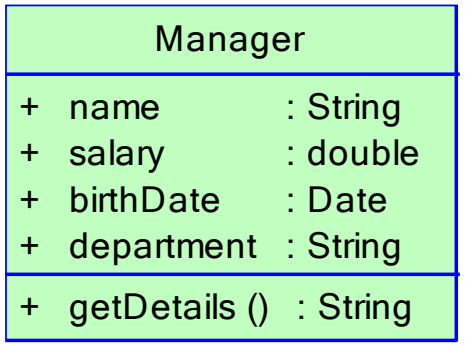
目 录

contents

11.1 类继承、子类和父类的isA关系



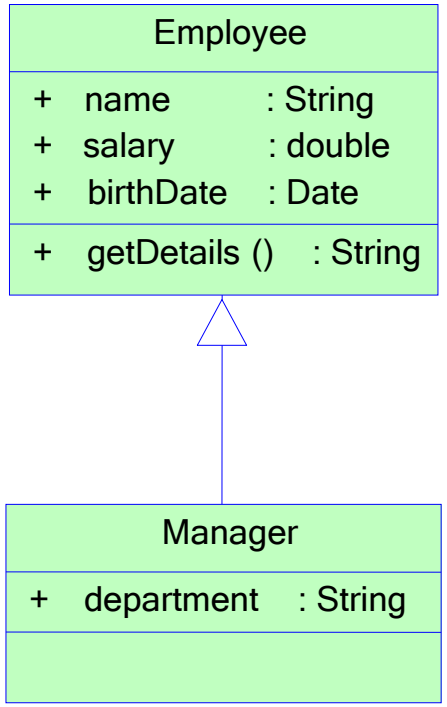

```
public class Employee {
    public String name;
    public double salary;
    public Date birthDate;
    public String getDetails() {...}
}
```




```
public class Manager {
    public String name; //同名成员
    public double salary; //同名成员
    public Date birthDate; //同名成员
    public String department;
    public String getDetails() {...}
}
```

11.1 类继承、子类和父类的isA关系

考虑到Manager只比Employee多出一个属性Department，可以利用继承机制（提供了源代码重用机制，提高了开发效率同时也提高了软件可靠性：父类的代码如果经过了可靠性测试，我们通过继承机制可放心直接使用，只需要关注子类代码的实现）



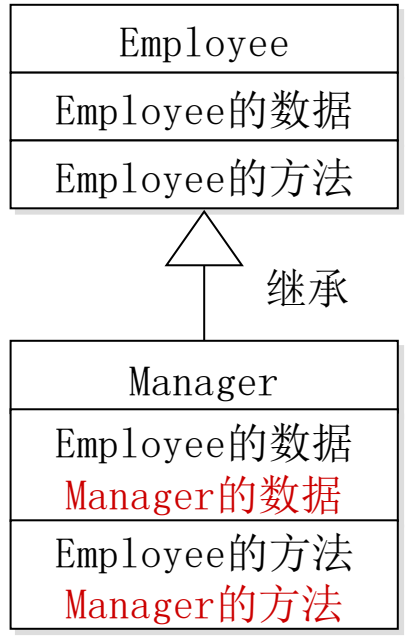

```
public class Employee {
    public String name;
    public double salary;
    public Date birthDate;
    public String getDetails() {...}
}
```

```
public class Manager extends Employee {
    public String department;
}
```

11.1 类继承、子类和父类的isA关系

- 语法

```
class ClassName extends Superclass {
        class body
} //如果父类是Object，则extends部分可省略（前面的示例代码里，每个的父类都是Object）
```

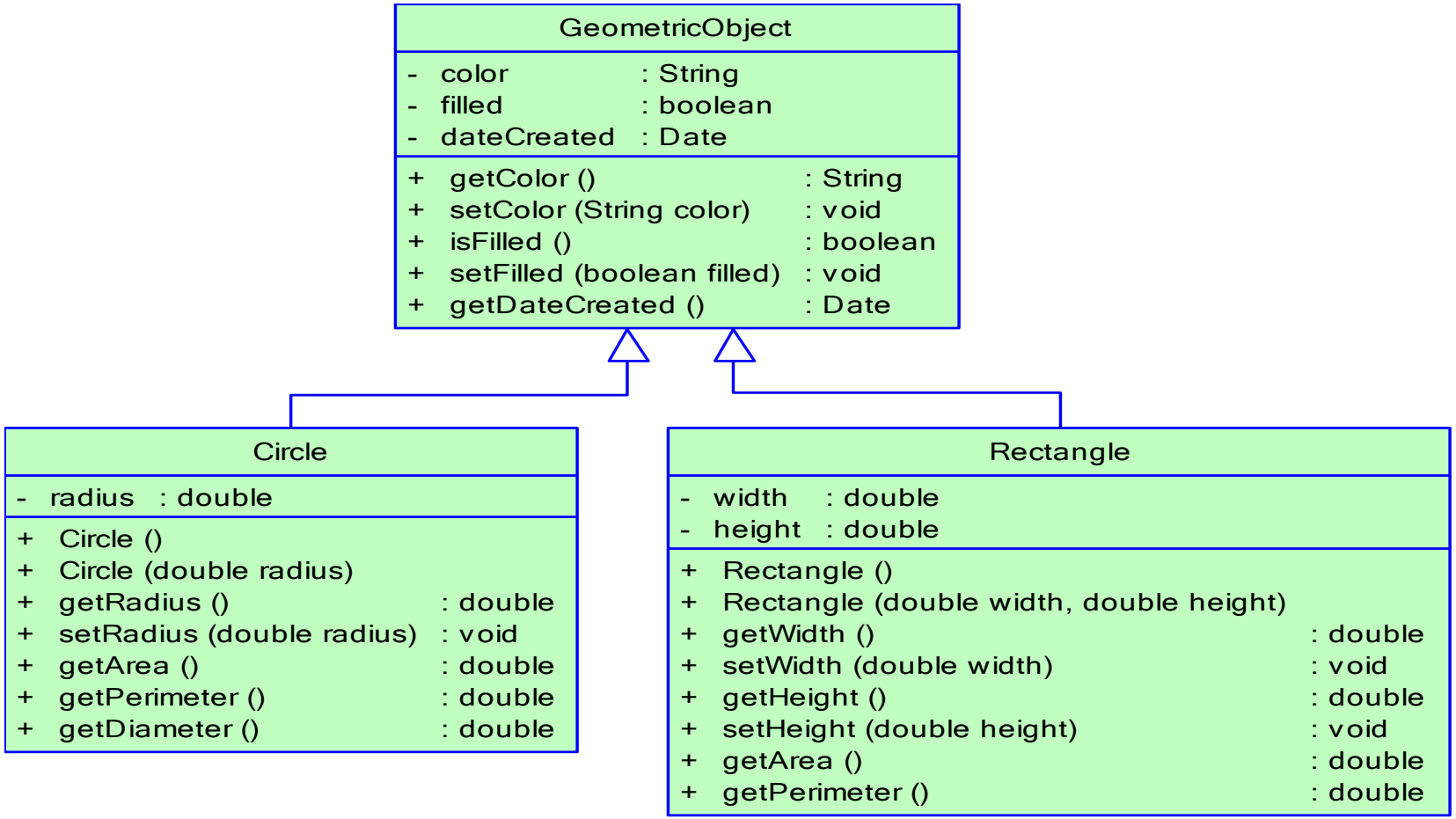


- 如果class C1 extends C2，则称C1为子类(subclass)，C2为父类(superclass)。
  - 子类继承了父类中可访问的数据和方法，子类也可添加新的数据和方法，
  - 子类不继承父类的构造函数。
  - 一个类只能有一个直接父类（Java不支持多重继承，因为Java的设计者认为没有必要）。
- Java的继承都是公有继承，因此被继承的就是父类，继承的类就是子类。因此父类的成员如果被继承到子类，访问权限不变
- 因此子类和父类是ISA关系：一个子类对象ISA父类对象

11.1 类继承、子类和父类的isA关系-设计案例

- 假设要设计模拟几何对象的类，如圆和矩形，考虑的因素有颜色，是否填充，创建日期，圆的半径，矩形的周长等
- 不要直接就定义Class Circle和Class Rectangle，先分析几何对象的共同属性和行为。共同的属性和行为有颜色、是否填充，创建时间，以及这些属性的getter和setter行为。而圆的半径、矩形的长宽不是几何图形共有的属性。
- 可以设计通用类GeometricObject来模拟共有的属性和方法。而Circle、Rectangle类通过继承GeometricObject获得共同的属性和行为，同时添加自己特有的属性和行为

11.1 类继承、子类和父类的isA关系-案例UML



说明：任何类在设计时应考虑覆盖祖先类`Object`的如下函数：`equals`, `clone`, `toString`等

11.1 类继承、子类和父类的isA关系-案例代码

```
public class GeometricObject { //等价于public class GeometricObject extends Object
    private String color = "white";
    private boolean filled;
    private Date dateCreated; //java.util.Date是JDK定义的类，表示日期和时间
    public GeometricObject() { dateCreated = new Date();}

    public String getColor() { return color; }
    public void setColor(String color) { this.color = color;}
    public boolean isFilled() { return filled; }
    public void setFilled(boolean filled) { this.filled = filled;}
    public Date getDateCreated() {return dateCreated;}

    @Override //覆盖Object类的toString()方法
    public String toString( ) { //还应考虑equals，clone
       return "created on " + dateCreated + "\n\tcolor: " + color
                            + " and filled: " + filled;
   }//toString方法应该返回一个描述当前对象的有意义的字符串
}
```

new Date()返回当前时间

这里不应该提供setDateCreated方法

11.1 类继承、子类和父类的isA关系-案例代码

```
public class Circle extends GeometricObject {
        private double radius; //新增属性
        public Circle() { }
        public Circle(double radius) { this.radius = radius; }
        public double getRadius() { return radius; }
        public void setRadius (double radius) { this.radius = radius; }

        public double getArea() {
                return radius * radius * Math.PI;
        }
        public double getDiameter() {
                return 2 * radius;
        }
        public double getPerimeter() {
                return 2 * radius * Math.PI;
        }
        //还应考虑equals，clone，toString等函数
}
```

11.1 类继承、子类和父类的isA关系-案例代码

```
public class Rectangle extends GeometricObject {
        private double width;
        private double height;

        public Rectangle() { }
        public Rectangle(double width, double height) {
                this.width = width;
                this.height = height;
        }

        public double getWidth() { return width; }
        public void setWidth (double width) { this.width = width;}
        public double getHeight() { return height;}
        public void setHeight (double height) { this.height = height;}
        public double getArea() { return width * height;}
        public double getPerimeter() {
                return 2 * (width + height);
        }
          //还应考虑equals，clone，toString等函数
}
```

11.1 类继承、子类和父类的isA关系

GeometricObject类的属性和方法

私有属性:
color
filled
dateCreated

公有方法:
getColor
setColor
IsFilled
setFilled
getDataCreated

11.1 类继承、子类和父类的isA关系

Circle类的属性和方法

GeometricObject的私有属性在子类中不可见（即不能在子类里直接访问）

但可以通过所继承的get和set方法设置和访问

自己的私有属性：
radius

父类的公有方法：
getColor
setColor
IsFilled
setFilled
getDataCreated

+从父类继承的公有方法

自己实现的公有方法：
getRadius
setRadius
getArea
getDiameter
getPerimeter

11.1 类继承、子类和父类的isA关系

Rectangle类的属性和方法

GeometricObject的私有属性在子类中不可见（即不能在子类里直接访问）

但可以通过所继承的get和set方法设置和访问

自己的私有属性：
width
height

父类的公有方法：
getColor
setColor
IsFilled
setFilled
getDataCreated

+从父类继承的公有方法

自己实现的公有方法：
getWidth
setWidth
getHeight
setHeight
getArea
getPerimeter

11.1 类继承、子类和父类的isA关系-子类对象的构造

实例初始化模块

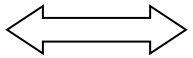
- 初始化块是Java类中可以出现的第四种成员（前三种包括属性、方法、构造函数），分为实例初始化块和静态初始化块。
- 实例初始化模块（instance initialization block, IIB）是一个用大括号括住的语句块，直接嵌套于类体中，不在方法内。
- 它的作用就像把它放在了类中每个构造方法的最开始位置。用于初始化对象。实例初始化块先于构造函数执行
- 作用：如果多个构造方法共享一段代码，并且每个构造方法不会调用其他构造方法，那么可以把这段公共代码放在初始化模块中。
- 一个类可以有多个初始化模块，模块按照在类中出现的顺序执行

11.1 类继承、子类和父类的isA关系-子类对象的构造

实例初始化模块

- 初始化模块可以简化构造方法的代码

```
public class Book{
  private static int numOfObjects;
  private String title
  private int id;
  public Book(String title){
    this.title = title;
  }
  public Book(int id){
    this.id = id
  }

  {
    numOfObjects++;
  }
}
```




```
public class Book{
  private static int numOfObjects;
  private String title
  private int id;
  public Book(String title){
    numOfObjects++;
    this.title = title;
  }
  public Book(int id){
    numOfObjects++;
    this.id = id
  }


}
```

11.1 类继承、子类和父类的isA关系-子类对象的构造

实例初始化模块

- 实例初始化模块还有个作用是可以截获异常

```
public class A{
    //下面数据成员初始化语句可能会抛出异常，所以下面语句不成立
    private InputStream fs = new FileInputStream(new File("C:\\1.txt"));
    private int i = 0;
}
```

```
public class A{
    //在实例初始化块里初始化数据成员可以截获异常
    private InputStream fs = null;
    {
      try{ fs = new FileInputStream(new File("C:\\1.txt"));}
      catch(Exception e){ …}
    }
    public A(){ … }

}
```

11.1 类继承、子类和父类的isA关系-子类对象的构造

实例初始化模块

- 实例初始化模块最重要的作用是当我们需要写一个内部匿名类时：匿名类不可能有构造函数，这时可以用实例初始化块来初始化数据成员

```
interface ISay{ public abstract void sayHello(); }
public class InstanceInitializationBlockTest {
    public static void main(String[] args){
        ISay say = new ISay()
        {    //这里定义了一个实现了ISay接口的匿名类
             //final类型变量一般情况下必须马上初始化，一种例外是：final实例变量可以在构造函数里再初始化。
             //但是匿名类又不可能有构造函数，因此只能利用实例初始化块
             private final int j;  //为了演示实例初始化块的作用，这里特意没有初始化常量j
             {
                 j = 0;  //在实例初始化块里初始化j
             }
             @Override
             public void sayHello() { System.out.println("Hello");}
        };
        say.sayHello();
    }
}
```

实例初始化模块

内部匿名类

11.1 类继承、子类和父类的isA关系-子类对象的构造

实例初始化模块

```
interface ISay{ public abstract void sayHello(); }
public class InstanceInitializationBlockTest {
    public static void main(String[] args){
        ISay say = new ISay()
        {   //这里定义了一个实现了ISay接口的匿名类
            //必须使用实例初始化块的另外一种场景：下面语句会抛出异常，所以编译错误
            //private InputStream s = new FileInputStream(new File("C:\\1.txt"));
            private InputStream s;
            {
                try {
                    s = new FileInputStream(new File("C:\\1.txt"));
                } catch (FileNotFoundException e) { e.printStackTrace(); }
            }
            @Override
            public void sayHello() { System.out.println("Hello");}
        };
        say.sayHello();
    }
}
```

实例初始化模块

内部匿名类

11.1 类继承、子类和父类的isA关系-子类对象的构造

实例初始化模块

- 实例初始化模块只有在创建类的实例时才会调用。
- 定义并初始化类的实例变量等价于实例初始化块：private int id = 0;
- 一个类可以有多个实例初始化块，对象被实例化时，模块按照在类中出现的顺序执行，构造函数最后运行。

```
public class Book{
   private int id = 0;                    //执行次序：1
   public Book(int id){                   //执行次序：4
      this.id = id
   }
   {
      //实例初始化块                      //执行次序：2
   }
   {
      //实例初始化块                      //执行次序：3
   }
}
```

11.1 类继承、子类和父类的isA关系-子类对象的构造

静态初始化模块

- 静态初始化模块是由static修饰的初始化模块{}，只能访问类的静态成员，并且在JVM的Class Loader将类装入内存时调用。（类的装入和类的实例化是两个不同步骤，首先是将类装入内存，然后再实例化类的对象）。
- 在类体里直接定义静态变量相当于静态初始化块

```
public class A{
    //类的属性和方法定义
    {
       //实例初始化模块
    }
    static {
       //静态初始化模块
    }
    public static int i = 0;//直接定义静态变量相当于静态初始化块
}
```

11.1 类继承、子类和父类的isA关系-子类对象的构造

静态初始化模块

- 一个类可以有多个静态初始化块，类被加载时，这些模块按照在类中出现的顺序执行

```
public class Book{
  private static int id = 0;          //执行次序：1
  public Book(int id){
        this.id = id
  }
  static {
     //静态初始化块                    //执行次序：2
  }
  static {
     //静态初始化块                    //执行次序：3
  }
}
```

11.1 类继承、子类和父类的isA关系-子类对象的构造

初始化模块执行顺序

- 第一次使用类时装入类
  - 如果父类没装入则首先装入父类，这是个递归的过程，直到继承链上所有祖先类全部装入
  - 装入一个类时，类的静态数据成员和静态初始化模块按它们在类中出现的顺序执行
- 实例化类的对象
  - 首先构造父类对象，这是个递归过程，直到继承链上所有祖先类的对象构造好
  - 构造一个类的对象时，按在类中出现的顺序执行实例数据成员的初始化及实例初始化模块
  - 执行构造函数函数体

11.1 类继承、子类和父类的isA关系-子类对象的构造

初始化模块执行顺序

- 如果声明类的实例变量时具有初始值，如

```
double radius = 5.0;
```

变量的初始化就像在实例初始化模块中一样，即等价于

```
double radius; { radius = 5.0; }
```

- 如果声明类的静态变量时具有初始值，如

```
static int numOfObjects = 0;
```

变量的初始化就像在静态初始化模块中一样，即等价于

```
static int numOfObjects; static{ numOfObjects = 0; }
```

11.1 类继承、子类和父类的isA关系-子类对象的构造

InitializeDemo

```
public class InitDemo{
    InitDemo(){
        new M();
    }
    public static void main(String[] args){
         System.out.println("(1) ");
         new InitDemo();
    }
    {
        System.out.println("(2) ");
    }
    static{
        System.out.println("(0) ");
    }
}
```

```
class N{
    N(){      System.out.println("(6) "); }
    {
        System.out.println("(5) ");
    }
    static {
        System.out.println("(3) ");
    }
}
class M extends N{
    M(){      System.out.println("(8) "); }
    {
        System.out.println("(7) ");
    }
    static {
        System.out.println("(4) ");
    }
}
```

11.2 super关键字

- 利用super可以显式调用父类的构造函数
  - super(parameters$_{opt}$)调用父类的的构造函数。
  - 必须是子类构造函数的第1条且仅1条语句(先构造父类)。
  - 如果子类构造函数中没有显式地调用父类的构造函数，那么将自动调用父类不带参数的构造函数。
  - 父类的构造函数在子类构造函数之前执行。
- 访问父类的成员（包括静态和实例成员）
  - super不能用于静态上下文（即静态方法和静态初始化块里不能使用super），this也不能用于静态上下文
  - super.data（如果父类属性在子类可访问，包括实例和静态）
  - super.method(parameters)（如果父类方法在子类可访问，包括实例和静态）
  - 不能使用super.super.p()这样的super链

11.2 super关键字

如果子类中没有显式地调用父类的构造函数，**也没有使用this调用类的其它构造函数**，那么将自动调用父类不带参数的构造函数，因为编译器会偷偷地在子类构造函数第一条语句前加上super();

```
public A () {

}
```

等价于

```
public A () {
   super();
}
```

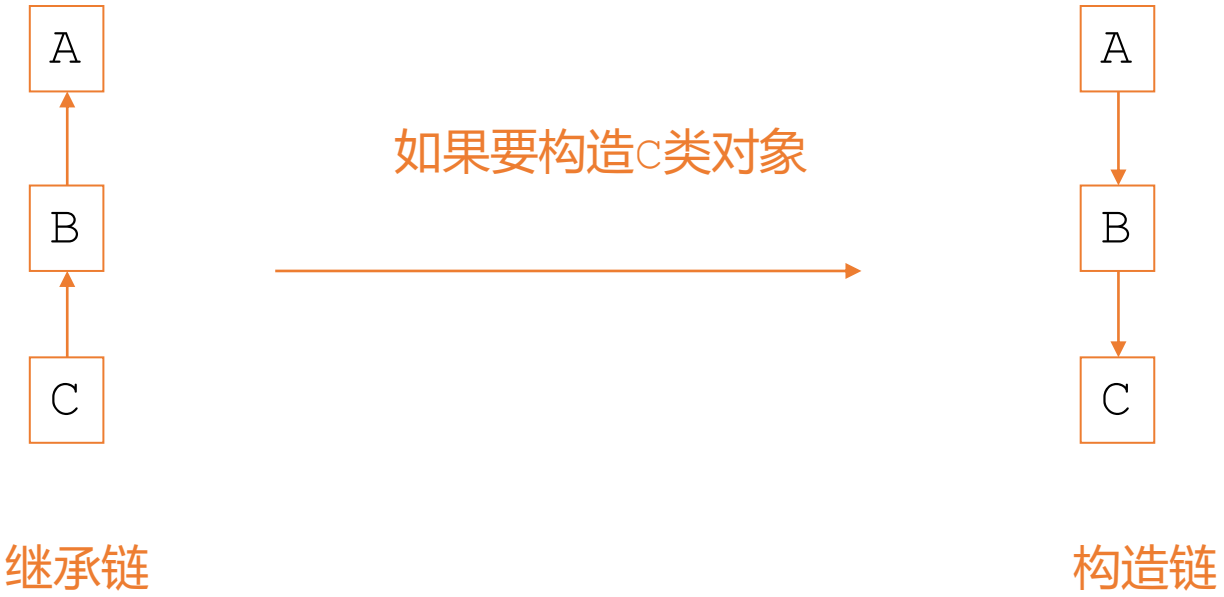
```
public A (double d)
{
//some statements

}
```

等价于

```
public A (double d)
{
   super();
//some statements
}
```

11.2 super关键字

- 在任何情况下，构造一个类的实例时，会沿着继承链调用所有父类的构造方法，这叫构造方法链。



Example: 教材11.3.2节构造方法链例子

11.2 super关键字-无参构造函数

- 如果一个类自定义了构造函数（不管有无参数），编译器不会自动加上无参构造函数。
- 如果一个类没定义任何构造函数，编译器会自动地加上无参构造函数。
- 编译器在为子类添加无参构造函数时，函数体里会用`super( )`默认调用父类的无参构造函数，如果找不到父类无参构造函数，则编译器为子类添加无参构造函数失败，编译报错。
- 如果一个类定义了带参数的构造函数，一定别忘了定义一个无参的构造函数，原因是：由于系统不会再自动加上无参构造函数，就造成该类没有无参构造函数

11.2 super关键字-无参构造函数的后果

- 如果父类没有无参构造函数，那么子类构造函数里若调用父类无参构造函数就会编译出错。

```
class Fruit {
        public Fruit(String name) {
                System.out.println("调用Fruit的构造函数");
        }
}
class Apple extends Fruit { }
//编译器为Apple提供无参构造函数时出错
//子类Apple没定义任何构造函数，故编译为子类提供无参构造函数Apple();
//提供的Apple()会调用父类无参构造函数Fruit(),因下列原因无法调用:
//父类定义了有参构造函数，所以编译没有为父类提供无参Fruit( )。
```

11.3 实例方法覆盖

- 如果子类重新定义了从父类中继承的实例方法，称为方法覆盖(method override)。
  - 仅当父类方法在子类里是可访问的，该实例方法才能被子类覆盖，即父类私有实例方法不能被子类覆盖，父类实例私有方法自动视为final的。
  - 静态方法不能被覆盖，如果静态方法在子类中重新定义，那么父类方法将被隐藏。
  - 覆盖特性：一旦父类中的实例方法被子类覆盖，同时用父类型的引用变量引用了子类对象，这时不能通过这个**父类型**引用变量去访问被覆盖的父类方法(即这时被覆盖的父类方法不可再被发现)。因为实例方法具有多态性（晚期绑定）
    - 在子类函数中可以使用super调用被覆盖的父类方法（**只有一级**）。
  - 隐藏特性：指父类的变量（实例变量、静态变量）和静态方法在子类被重新定义，但由于类的变量（实例和静态）和静态方法没有多态性，因此通过父类型引用变量访问的一定是父类变量、静态方法(即被隐藏的可再发现)。
  - 方法覆盖的哲学涵义：子对象当然可以修改父类的行为（生物进化除了遗传，还有变异）

11.3 实例方法覆盖

```
class A{
    public void m() {
        System.out.println("A's m");
    }

    public static void s() {
        System.out.println("A's s");
    }
}
```

```
class B extends A{
    //覆盖父类实例方法
    public void m() {
        System.out.println("B's m");
    }
    //隐藏父类静态方法
    public static void s() {
        System.out.println("B's s");
    }
}
```

```
public class OverrideDemo {

    public static void main(String[] args) {
        A o = new B(); //父类型变量引用子类对象
        o.m(); //由于父类实例方法m被子类覆盖，o运行时指向B类对象，由于多态性，执行的是B的m
        o.s();  //由于s是静态方法，没有多态性，编译器编译时对象o的声明类型是A，所以执行的是A的s
    }
}
```

父类型变量o引用了子类对象，通过o调用被覆盖的实例方法m时，调用的一定是子类方法，这时不可能调用到父类的方法m(父类函数不能被发现），因为多态特性。多态性使得根据new 后面的类型决定调用哪个m

静态方法和**成员变量**没有多态性，因此要根据声明类型决定调用哪个s

11.3 实例方法覆盖

```
class A{
    public void m() {
        System.out.println("A's m");
    }

    public static void s() {
        System.out.println("A's s");
    }
}
```

```
class B extends A{
    public void m() {//覆盖父类实例方法
        System.out.println("B's m");
    }
    public static void s() {//隐藏父类静态方法
        System.out.println("B's s");
    }
}
```

B：实际运行时指向的类型

A：声明类型

```
public class OverrideDemo {
    public static void main(String[] args) {
        A o = new B();
        o.m();
        o.s();
    }
}
```

绑定：找到函数入口地址的过程

引用变量o有二个类型：声明类型A，实际运行时类型B

1. 编译时看声明类型（检查语法错误）
2. 类方法、类属性、实例属性没有多态性（类私有方法，类final方法不能覆盖，也没有多态性），声明类型决定访问的实际成员，比如判断o.s()执行的是哪个函数按照o的声明类型。函数入口地址在编译时就确定（早期绑定），而编译时所有变量的类型都按声明类型。
3. 类能够覆盖的实例方法（有多态性的），运行时类型决定访问的实际方法。比如判断o.m()执行的是哪个函数按照o的实际运行类型，在运行时按照o指向的实际类型B来重新计算函数入口地址（晚期绑定。多态性），因此调用的是B的m

11.3 实例方法覆盖

```
class A{
    public void m() {
        System.out.println("A's m");
    }

    public static void s() {
        System.out.println("A's s");
    }
}
```

```
class B extends A{
    //覆盖父类实例方法
    public void m() {
        System.out.println("B's m");
    }
    //隐藏父类静态方法
    public static void s() {
        System.out.println("B's s");
    }
}
```

```
public class OverrideDemo {

    public static void main(String[] args) {
        A o = new B();
        o.m();
        o.s();
    }
}
```

B：实际运行时指向的类型

A：声明类型 → A o = new B();

因此一旦引用变量o指向了B类型对象（A o = new B()），o.m()调用的永远是B的m，再也无法通过o调用A的m，哪怕强制转换都不行：((A)o).m();调用的还是B的m
这就是前面PPT所说的不能再发现。

11.3 实例方法覆盖

```
class A{
    public void m() {
        System.out.println("A's m");
    }

    public static void s() {
        System.out.println("A's s");
    }
}

class B extends A{
    //覆盖父类实例方法
    public void m() {
        System.out.println("B's m");
    }
    //隐藏父类静态方法
    public static void s() {
        System.out.println("B's s");
    }
}

public class OverrideDemo {

    public static void main(String[] args) {
        //静态方法隐藏
        B o = new B();
        o.s(); //调用B的s，将父类A的s隐藏，即通过B类型的引用变量o是不可能调用A的s
        ((A)o).s(); //通过强制类型转换，可以调用A的s，可以找回。也可以通过类名调用来找回:A.s();
    }                    //这就是第29页PPT讲的：被隐藏的变量和静态方法可以再发现
}
```

B：实际运行时指向的类型

B：声明类型 → B o = new B();

11.3 实例方法覆盖

```
class ch11_A{
    void f(int m){System.out.println("ch11_A f(int):"+m);}
    static void s(int m){System.out.println("ch11_A s(int):"+m);}
}
class ch11_B extends ch11_A{
    void f(int m){System.out.println("ch11_B f(int):"+m);}
    static void s(int m){System.out.println("ch11_B s(int):"+m);}
    void f(byte m){System.out.println("ch11_B f(byte):"+m);}
    static void s(byte m){System.out.println("ch11_B s(byte):"+m);}
}
public class TestOverride {
    public static void main(String[] args) {
        byte t1=18,t2=15;
        ch11_A a=new ch11_B();
        a.f(t1);        a.s(t2);
        System.out.println("Next 1");
        ch11_B b=(ch11_B)a;
        b.f(t1);        b.s(t2);
        System.out.println("Next 2");
        b.f(18);        b.s(15);
    }
}
```

```
ch11_B f(int):18
ch11_A s(int):15
Next 1
ch11_B f(byte):18
ch11_B s(byte):15
Next 2
ch11_B f(int):18
ch11_B s(int):15
```

改成 byte、short、long，会有什么结果

改成 int、short、long，会有什么结果

11.3 实例方法覆盖

- 比如可以在Circle类里覆盖toString（）方法（第8页的例子基础上）

```
public String toString() {
   return "A circle " + super.toString()
        + "\n\tradius: " + radius;
}
```

- 这样做的好处是Circle对象的基本属性如color，filled，dateCreated由父类方法打印，Circle对象只负责打印新的属性值

11.4 Object中的方法

- java.lang.Object类是所有类的祖先类。如果一个类在声明时没有指定父类，那么这个类的父类是Object类。
  - 它提供方法如**toString**、**equals**、getClass、**clone**、~~finalize~~。前3个为公有，后2个为保护。getClass为final（用于泛型和反射机制，禁止覆盖）。
  - equals方法：用于测试两个对象内容是否相等。Object类的默认实现是比较两个对象引用是否引用同一个对象。因此通常子类应该覆盖该方法。
  - toString方法：返回代表这个对象的字符串。Object类的默认实现是返回由类名、@和hashCode组成，类名中包含package。

    ```
    Circle circle = new Circle();
    ```

    ```
    circle.toString();//Circle@15037e5，如果Circle没有覆盖toString
    ```
  - Object的toString方法提供的信息不是很有用。因此通常子类应该覆盖该方法，提供更有意义的信息（前一页PPT例子）。

11.4 Object中的方法

- equals用于判断一个对象同另一个对象的所有成员内容是否相等。覆盖时应考虑：
  - 对基本类型数值成员。直接使用==判断即可。
  - 对引用类型变量成员。则需要对这些变量成员调用equals判断，不能用==。
- 覆盖equals函数，最好同时覆盖hashCode()方法，该方法返回对象的hashCode值。
  - *需要对比的时候，首先用hashCode去对比，如果hashCode不一样，则表示这两个对象肯定不相等（也就是不必再用equals()去再对比了），如果hashCode相同，此时再用equals()比，如果equals()也相同，则表示这两个对象是真的相同了，这样既能大大提高了效率也保证了对比的绝对正确性！*
- 覆盖equals函数，首先用instanceof检查参数的类型是否和当前对象的类型一样。
- 例如，在Circle类中覆盖：

```
public boolean equals(Object o){
    if(o instanceof Circle)        //应先检查另一对象o的类型
        return  radius==((Circle)o).radius;
    return false;
}
```

11.4 Object中的方法

- 覆写equals()需要注意的地方：
- • 对称性：如果x.equals(y)返回是“true”，那么y.equals(x)也应该返回是“true”。
- • 反射性：x.equals(x)必须返回是“true”。
- • 类推性：如果x.equals(y)返回是“true”，而且y.equals(z)返回是“true”，那么z.equals(x)也应该返回是“true”。
- • 一致性：如果x.equals(y)返回是“true”，只要x和y内容一直不变，不管你重复x.equals(y)多少次，返回都是“true”。
- • 任何情况下，x.equals(null)，永远返回是“false”；
- • x.equals(和x不同类型的对象)永远返回是“false”。
- 覆写hashCode的原则
- • 如果两个对象相等，则两个对象的hashCode()必须相等；
- • 如果两个对象不相等，则两个对象的hashCode()尽量不要相等

11.4 Object中的方法

```
class B1 extends A1{
    int b;
    StringBuffer sb;
    S_A sa;
    String[] ss={"a","b","c"};
    StringBuffer[] bb=new StringBuffer[3];
    List<String> ls=new ArrayList<>();
    @Override
    public String toString() {
        StringBuffer sss=new StringBuffer();
        for(String s:ls){
            sss.append(s);
            sss.append(" ");
        }
        return "B1{" +"b=" + b +", sb=" + sb +
                ", sa=" + sa +", ss=" + Arrays.toString(ss) +
                ", bb=" + Arrays.toString(bb) +
                ", ls=" + sss + "} " + super.toString();
    }
}
```

1. 直接父类不是Object，使用super. toString()获得父类的信息字符串，然后和其它信息拼接。
2. 值类型属性、包装类型、String属性直接和其它信息拼接。
3. 一般类类型属性，使用对象. toString()获得信息和其它信息拼接。
4. 数组类型属性，可以使用下面的方法获得信息和其它信息拼接。
    1. Arrays.*toString*(对象)，可能不是我们需要的格式
    2. 循环数组的每一个元素，按照要求的格式拼接成一个字符串，可以使用String拼接，也可以使用StringBuffer、StringBuilder进行拼接
5. 容器类型属性，遍历容器的每一个元素（迭代模式），按照要求的格式拼接成一个字符串，可以使用String拼接，也可以使用StringBuffer、StringBuilder进行拼接。

11.4 Object中的方法

```
class B1 extends A1{
    int b;
    StringBuffer sb;
    S_A sa;
    String[] ss={"a","b","c"};
    StringBuffer[] bb=new StringBuffer[3];
    List<String> ls=new ArrayList<>();

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof B1)) return false;
        if (!super.equals(o)) return false;
        B1 b1=(B1)o;
        return b == b1.b &&
                Objects.equals(sb, b1.sb) &&
                Objects.equals(sa, b1.sa) &&
                Objects.deepEquals(ss, b1.ss) &&
                Objects.deepEquals(bb, b1.bb) &&
                Objects.equals(ls, b1.ls);
    }
}
```

1. 形参o不是当前类的类型，返回false。
2. 直接父类不是Object，super.equals(o)结果为false，直接返回false
3. 依次判断子类每一个实例属性、只要有一个为false，返回false，根据属性类型不同，按照下面的方法判断：
    1. 值类型，包装类型，可以用==，!=判断
    2. 一般类类型属性（包括String类型），使用equals判断：
        1. 对象.equals(其它对象)
        2. Objects.*equals*(对象1,对象2)
        3. StringBuffer、StringBuilder需要转换成String比较内容
    3. 数组类型属性：
        1. 对象1==对象2，为真
        2. 依次对比对象1和对象2的每一个元素，所有内容相同，结果为真（注意数组元素类型不同，判断内容相同的方法有区别）
        3. Objects.*deepEquals*(对象1,对象2)
        4. Arrays.equals(对象1,对象2), Arrays. *deepEquals*(对象1,对象2)
    4. 容器类型属性：
        1. 对象1==对象2，为真
        2. 依次对比对象1和对象2的每一个元素，所有内容相同，结果为真（注意数组元素类型不同，判断内容相同的方法有区别）
        3. Objects.*equals*(对象1,对象2)

11.4 Object中的方法

- 要实现一个类的clone方法，首先这个类需要实现Cloneable接口，否则会抛出CloneNotSupportedException异常
- Cloneable接口其实就是一个标记接口，里面没有定义任何接口方法，只是用来标记一个类是否支持克隆：没有实现该接口的类不能克隆
- 还要公有覆盖clone方法，即Object类里clone方法是保护的，子类覆盖这个方法时应该提升为public
- 方法里应实现深拷贝clone，Object的clone实现是浅拷贝。
- 克隆的深度：要克隆的对象可能包含基本类型数值成员或引用类型变量成员，对于基本类型数值成员使用=赋值即可，对于引用类型成员则需要进一步嵌套调用该成员的克隆方法进行赋值。

11.4 Object中的方法

```
//首先必须实现Cloneable接口
class A implements Cloneable{
    public static final int SIZE = 10;
    private int[] values = new int[SIZE]; //A的values成员是数组

    public int[] getValues(){
        return values;
    }
    //覆盖clone方法，提升为public，只是调用Object的的clone，
    //不修改行为
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();//调用Object的clone。//注意super.clone返回Object在运行时就是A类型
    }
}
```

Clone方法不带参数，返回Object，同时可能会抛出CloneNotSupportedException异常

```
public class CloneTest {
    public static void main(String[] args) throws CloneNotSupportedException {
        A o1 = new A();
        A o2 = (A)(o1.clone()); //clone返回Object，因此要强制类型转换
        System.out.println(o1 == o2); //false，说明clone返回的是新的引用
        System.out.println(o1.getValues() == o2.getValues()); //true 但为浅拷贝克隆
    }                                  //因为o1和o2内部values引用指向了内存里同一个数组
                                       //说明A的clone方法里，所调用的super.clone()是浅拷贝
}
```

11.4 Object中的方法

```
class A implements Cloneable{
    public static final int SIZE = 10;
    private int[] values = new int[SIZE];
    public int[] getValues(){ return values; }
    public void setValues(int[] newValues) {
        this.values = newValues;
    }
    //覆盖equals方法，比较二个A类型对象内容是否一样
    public boolean equals(Object obj) {
        if(obj instanceof A){
            A o = (A)obj;
            return java.util.Arrays.equals(this.getValues(),o.getValues());
        }
        else return false;
    }
    //覆盖clone方法，提升为public，重新实现为深拷贝
    public Object clone() throws CloneNotSupportedException {
       // A newObj = new A(); //new一个新对象，该方法不好：在有继承关系的情况下，不利于复用父类的clone方法
        A newObj = (A)super.clone(); //强烈建议这么做
        newObj.values = this.values.clone(); //数组的clone是深拷贝，如果去掉clone，则是浅拷贝
        return newObj;
    }
}
```

添加setValues方法，设置内部数组内容

覆盖equals方法，比较二个A类型对象内容

重新覆盖clone方法，实现深拷贝

11.4 Object中的方法

```
class A implements Cloneable{
    //覆盖toString方法
    public String toString() {
        StringBuffer buf = new StringBuffer();
        for(int v : values){
            buf.append(v + " ");
        }
        return buf.toString().trim(); //去掉最后多余的空格
    }
}
```

```
public class CloneTest {
    public static void main(String[] args) throws CloneNotSupportedException {
        A o1 = new A();
        o1.setValues(new int[]{1,2,3,4,5,6,7,8,9,10});
        A o2 = (A)(o1.clone());
        System.out.println(o1 == o2); //false
        System.out.println(o1.getValues() == o2.getValues()); //false 说明不是浅拷贝克隆
        System.out.println(o1.equals(o2)); //true,二个对象的内容相等,  说明是深拷贝克隆
        System.out.println(o2.toString()); //显示 1 2 3 4 5 6 7 8 9 10
    }
}
```

11.4 Object中的方法

```
public class CloneTest {
    public static void main(String[] args) throws CloneNotSupportedException {
        A o1 = new A();
        A o2 = (A)(o1.clone());
        System.out.println(o1 == o2); //false
        System.out.println(o1.getValues() == o2.getValues()); //false 说明不是是浅拷贝克隆
        System.out.println(o1.equals(o2)); //true,二个对象的内容相等,  说明是深拷贝克隆
    }
}
```

```
class B implements Cloneable{
    A a;     //引用类型成员
    int i;   //值类型

    @Override
    public Object clone() throws CloneNotSupportedException {
        //B newObj = new B(); //这样做不好
         B newObj = (B)super.clone();
        newObj.i = this.i;    //值类型成员直接=赋值
        newObj.a = (A)(this.a.clone()); //引用类型的成员不能直接赋值，必须调用clone方法，深拷贝
        return newObj;
    }
}
```

如果B类里包含了A类型成员，只要A实现了深拷贝克隆，则B可以很方便地实现深拷贝克隆。如C类里包含了B类型成员，D类型里包含了C类型成员…，以此类推，只要每个类型都实现了深拷贝克隆，那么最外层的包装类可以非常方便的实现深拷贝克隆。这就是第37页PPT里讲到的克隆的深度问题

11.4 Object中的方法

```
class B1 extends A1 implements Cloneable{
    int b;
    StringBuffer sb;
    S_A sa;
    String[] ss={"a","b","c"};
    StringBuffer[] bb=new StringBuffer[3];
    List<String> ls=new ArrayList<>();

    public Object clone() throws CloneNotSupportedException {
        B1 b1=(B1)super.clone();
        b1.b=b;
        b1.sb=new StringBuffer(sb);
        b1.sa=(S_A)sa.clone();
        b1.ss=(String[]) ss.clone();
        for(int i=0;i<ss.length;i++){b1.ss[i]=new String(ss[i]);}
        b1.bb=(StringBuffer[]) bb.clone();
        for(int i=0;i<bb.length;i++) {b1.bb[i]=new StringBuffer(bb[i]);}
        b1.ls=new ArrayList<>();
        for(String s:ls) {b1.ls.add(s);}
        return b1;
    }
}
```

二步完成：

1. 子类名 newObj=(子类名)super.clone();
   //无论父类是否支持克隆
2. 依次完成子类每一个实例属性的clone，根据实例类型不同：
   1. 值类型，用=复制克隆（可以没有，第一步其实已经完成了值类型属性的克隆）
   2. String，包装类，建议使用new新建
   3. 一般类类型，支持克隆的，调用clone方法，不支持克隆的，使用new
   4. 数组对象，先clone/new/arraycopy数组对象，或者String[] ls2=Arrays.copyOf(ls1, ls1.length);再完成每一个元素的克隆
   5. 容器类，先new获得容器对象，再完成每一个元素的克隆

11.5 多态性、动态绑定和对象的强制类型转换

- 继承关系使一个子类可以继承父类的特征（属性和方法），并附加新特征
- 子类是父类的具体化（沿着继承链从祖先类到后代类，特征越来越具体；反过来，从后代类往祖先类回溯，越来越抽象）
- 每个子类的实例都是父类的实例（子类对象ISA父类），但反过来不成立

```
Class Student extends Person{ …}
Person p = new Student();//OK
Student s = new Person();//error
```

父类引用可直接指向子类对象

- 这个特性是多态的重要基础
- 先看一个例子

11.5 多态性、动态绑定和对象的强制类型转换

多态：通过引用变量调用实例函数时，根据所引用的实际对象的类型，执行该类型的相应实例方法，从而表现出不同的行为称为多态。通过继承时覆盖父类的实例方法实现多态。多态实现的原理：在运行时根据引用变量指向对象的实际类型，重新计算调用方法的入口地址（晚期绑定）。

```
class Person{  void Greeting( ){ System.out.println("Best wish from a person!"); } }
class Employee extends Person
              { void Greeting( ){ System.out.println("Best wish from a employee!");} }
class Manager extends Employee{
               void Greeting( ){ System.out.println("Best wish from a manager!");} }

public class GreetingTest1{
    public static void main(String[] args){
       //父类引用变量可以引用本类和子类对象，p1,p2,p3的声明类型都是Person(父类型)，p2,p3执行子类对象
       Person p1= new Person( ),p2= new Employee( ),p3= new Manager( );
        p1.Greeting( ); //调用Person的Greeting()     ，由于实际指向对象类型是Person
        p2.Greeting( ); //调用Employee的Greeting()   ，由于实际指向对象类型是Employee
        p3.Greeting( ); //调用Manager的Greeting()    ，由于实际指向对象类型是Manager

    }
}
```

11.5 多态性、动态绑定和对象的强制类型转换

```
class GreetingSender{
        public void newYearGreeting (Person p){ p.Greeting(); //编译时应该是Person的Greeing}
}
public class GreetingTest1{
       public static void main(String[] args){
        GreetingSender g = new GreetingSender();
        g.newYearGreeting(new Person());       //调用Person的Greeting()
        g.newYearGreeting(new Employee());     //调用Employee的Greeting()
        g.newYearGreeting(new Manager());      //调用Manager的Greeting()
      }
}
```

以最后一条语句为例来解释多态特性：
当实参new Manager()传给形参Person p时，等价于Person p = new Manager()，因此执行p.Greeting()语句时根据形参p指向的对象的实际类型动态计算Greeting方法的入口地址，调用了Manager的Greeting()

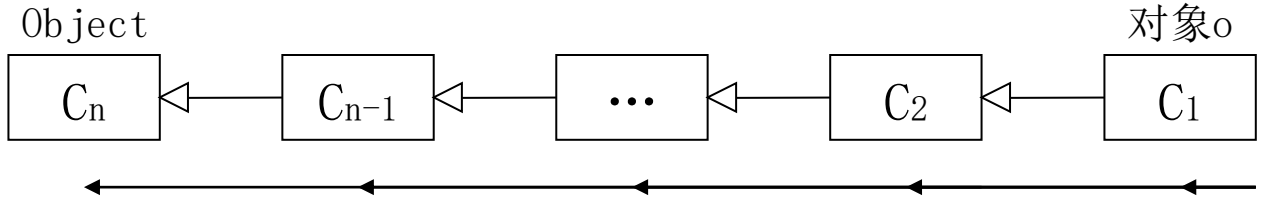
这段程序的源代码之所以体现了多态的三个重要因素：
1：不同类之间有继承关系
2：newYearGreeting方法的参数类型用的父类类型
3：newYearGreeting调用的Greeting方法都被子类用自己的行为覆盖

GreetingSender类的newYearGreeting方法的参数是Person类型，那么newYearGreeting的行为应该是Person对象的行为。
但是在实际运行时我们看到随着实参对象类型的变化，newYearGreeting方法却表现出了多种不同的行为，这种机制称为多态。
满足了这三个条件，继承链中不同子类的对象做为方法的实参去调用方法会使该方法表现出不同的行为。由于子类的实例也是父类的实例，所以用子类对象作为实参传给方法中的父类型的形参是没有问题的。

11.5 多态性、动态绑定和对象的强制类型转换

- 多态条件：父类变量可引用本类和子类对象，子类对象isA父类对象
- 当调用实例方法时，由Java虚拟机动态地决定所调用的方法，称为动态绑定(dynamic binding)或者晚期绑定或者延迟绑定(lazy binding)或者多态。

  假定对象o是类C1的实例，C1是C2的子类，C2是C3的子类，…，Cn-1是Cn的子类。也就是说，Cn是最一般的类，C1是最具体的类。在Java中，Cn是Object类。如果调用继承链里子类型C1对象o的方法p，Java虚拟机按照C1、C2、…、Cn的顺序依次查找方法p的实现。一旦找到一个实现，将停止查找，并执行找到的第一个实现（覆盖的实例函数）。

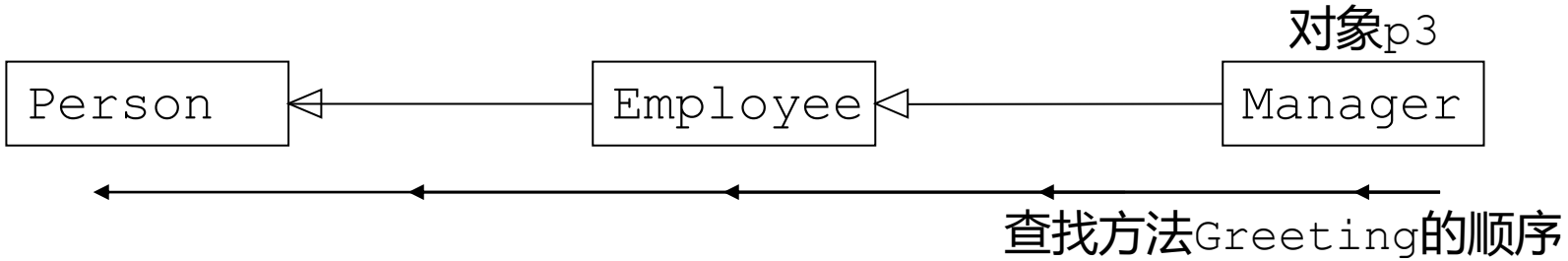


查找方法p的顺序：看C1是否覆盖p，如果已覆盖，调用C1的p;如果C1没有覆盖p，则查看C2是否覆盖，以此类推

从C1开始顺着继承链往父类查找，直到找到第一个p的实现，并调用这个p的实现

11.5 多态性、动态绑定和对象的强制类型转换

- 上例中，若p3指向Manager类的对象。当通过newYearGreeting(p3)方法去调用p3.Greeting()方法时，Java虚拟机会沿着继承链，从Manager类到父类查找Greeting的实现，结果找到Manager自己的实现



11.5 多态性、动态绑定和对象的强制类型转换

通用编程：方法参数和变量的声明类型越抽象越好，越抽象越通用

- 由于父类变量可以引用子类对象，针对父类型设计的任何代码都可以应用于子类对象。
  - `newYearGreeting(Person p){p.Greeting()}`这段代码可以应用于所有`Person`子类型对象
  - 多态性允许方法使用更通用的类作为参数类型。
  - 如果方法参数是父类，那么这个参数可以接受任何子类对象作为实参。当调用这对象的方法时，将动态绑定方法的实现。
    - `newYearGreeting(Person p)`:该方法能接受从`Person`类型开始所有子类型对象作为实参
    - `newYearGreeting(Manager p)`:该方法只能能接受从`Manager`类型开始所有子类型对象作为实参
    - 哪个更好?

11.5 多态性、动态绑定和对象的强制类型转换

通用编程：方法的参数参数，变量的类型越抽象越好，越抽象越通用

- 还是考虑上例
  - 如果没有多态机制，针对Person，Employee及Manager类，我们必须写出三个重载版本的newYearGreeting函数

```
class GreetingSender{
        public void newYearGreeting(Person p){ p.Greeting(); }
        public void newYearGreeting(Employee e){ e.Greeting(); }
        public void newYearGreeting(Manager m){ m.Greeting(); }

}
```

  - 假如我们新增加一个从Manager派生的CEO类，也实现了Greeting方法，我们需要增加一个新的newYearGreeing的重载版本。
  - 更为糟糕的是，我们需要重新编译GreetingSender类。
  - 回到第44页实现的多态版的GreetingSender类，可以适用于任何Person子类型，哪怕GreetingSender类作为商业类已经卖出去了，对后来新派生的CEO类型都可以不用重新编译地很好地工作

11.5 多态性、动态绑定和对象的强制类型转换

- 父类变量引用子类对象，可视为将子类对象转换为父类（不需强制类型转换）。
- 类型转换(type casting)可以将一个对象的类型转换成继承链中的另一种类型。
  - 从子类到父类的转换是合法的，称为隐式转换。

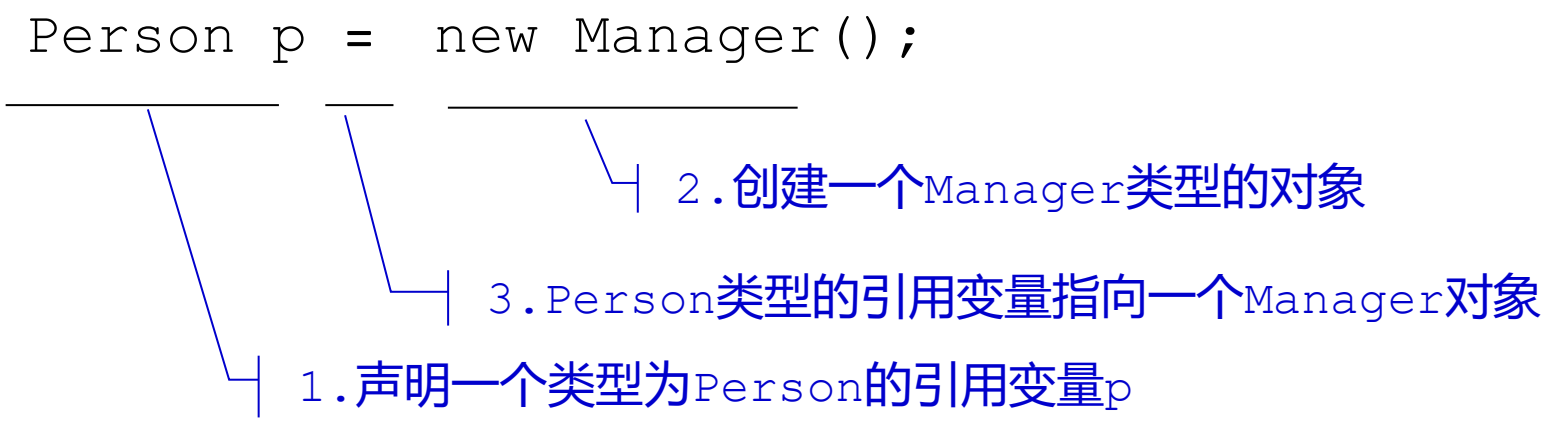
    ```
    Person p=new Manager();//将子类对象转换为父类对象
    ```
  - 从父类到子类必须显式（强制）转换。

    ```
    Manager m = p; //编译错 ，p是Person父类型，Person不一定是Manager
    Manager m = (Manager)p;//ok，但转换前没有检查
    ```
  - 从父类到子类转换必须显式转换，转换前应进行检查更安全。

    ```
    Manager m = null;
    if(p instanceof Manager) m= (Manager)p; //安全：转换前检查
    ```

11.5 多态性、动态绑定和对象的强制类型转换

- 为什么从父类到子类转换必须强制类型转换?
- 首先要理解类型检查（type checking）发生在编译时
- 然后要理解Person p = new Manager()的真正涵义



- 但是创建Manager对象并由p来引用是在运行时发生
- 因为程序还没运行，编译器无法知道p会指向什么对象，编译器在编译时只能根据变量p的声明类型（Person）来类型检查

11.5 多态性、动态绑定和对象的强制类型转换

- 当编译器检查到 `Manager m = p;` 编译器认为Person类型引用p要赋值给类型为Manager类型引用，扩展内存可能引起麻烦且不安全，因此，编译器认为类型不匹配，会报错。
- 加上强制转换 `Manager m = (Manager)p;` 意思是强烈要求编译器，把p解释成Manager类型，风险我来承担。这个时候编译器就按Manager类型来解释p
- 因此，强制类型转换意味着你自己承担风险，编译器不会再做类型检查。
- 强制类型转换的风险是：运行时如果p指向的对象不是Manager的实例时程序会出错。
- 为了避免风险，最好用instanceof来做实例类型检查。

11.5 多态性、动态绑定和对象的强制类型转换

instanceof操作符

- 可以用instanceof操作符判断一个引用指向的对象是否是一个类的实例。表达式返回boolean值。
- 语法

referenceVariable instanceof *TypeName*

- 所以上面的例子安全的写法为：

```
Person p = new Manag
if (p instanceof Man
    Manager m =  (Ma
```

```
if(p3 instanceof Manager m5){
    System.out.println("p1 is Person:"+m5.toString());
}
else{
    System.out.println("p1 is not Person");
}
```

- 意思是如果p指向的对象真的是Manager实例，再强制转换类型

11.5 多态性、动态绑定和对象的强制类型转换

- 重载发生在编译时(Compile time)，编译时编译器根据实参比对重载方法的形参找到最合适的方法。
- 多态发生在运行(Run time)时，运行时JVM根据变量所引用的对象的真正类型来找到最合适的实例方法。
- 有的书上把重载叫做“编译时多态”，或者叫“早期绑定”（早期指编译时）。
- 多态是晚期绑定（晚期指运行时）
- 绑定是指找到函数的入口地址的过程。

11.5 多态性、动态绑定和对象的强制类型转换

多态和强制类型转换例子

- 编写程序，创建两个几何对象：圆和矩形。调用displayObject来显示结果。
  - 如果对象是圆，显示半径和面积
  - 如果对象是矩形，显示面积
  - 警告：对象访问运算符(.)优先于类型转换运算符。使用括号保证在(.)运算符之前转换

```
((Circle)object).getArea() //OK
(Circle)object.getArea(); //错误
```

Example:教材第11章第9节程序清单11-7

11.6 访问控制符和修饰符final

- 类成员的访问控制符

高

成员修饰符	本类	本包	子类	它包
public	√	√	√	√
protected	√	√	●(它包)	*(非子类）
无(package)	√	√	*	*
private	√	*	*	*

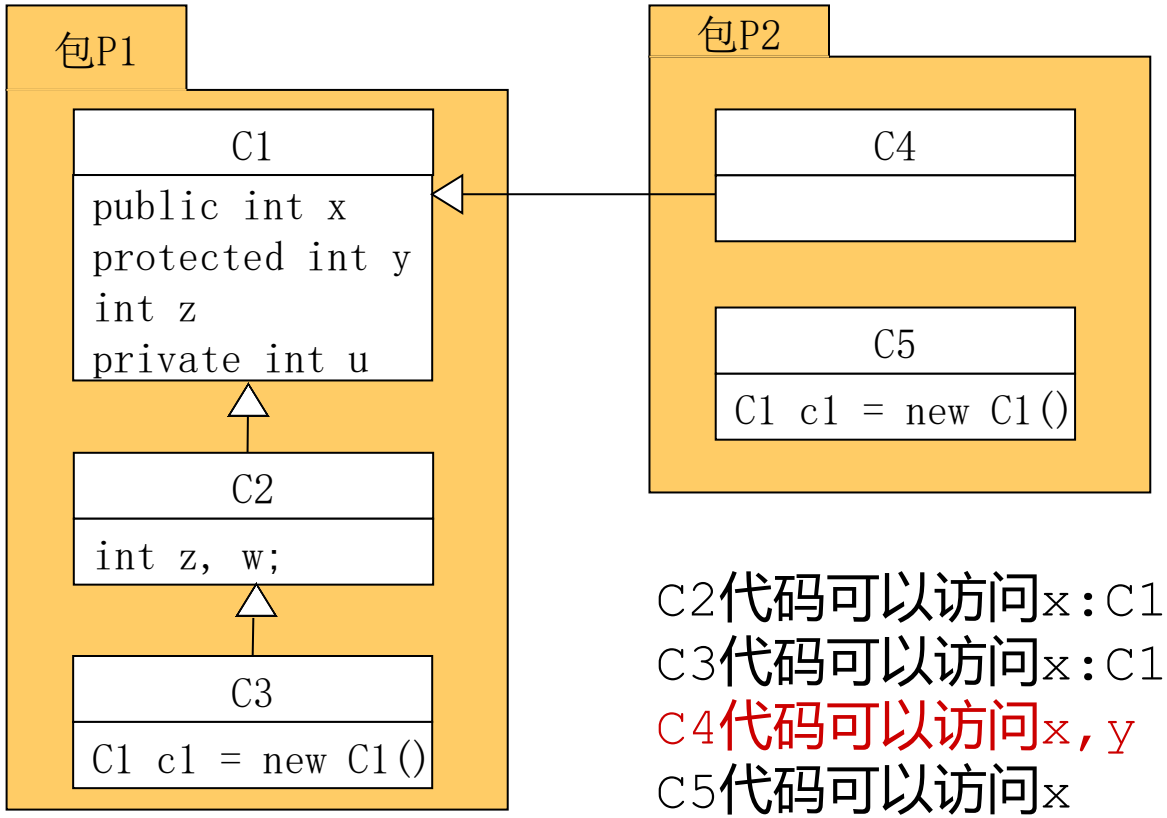
低

- 父类私有成员在子类不可见。
- 父类公有和保护成员在子类可见。
- 继承到子类后不改变父类成员的访问权限。
- 具体见第9章的例子，下一页的例子也请大家自己看

●子类类体中可以访问从父类继承来的protected成员 。但如果子类和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。

11.6 访问控制符和修饰符final

protected修饰符用于修饰数据和方法，
可以被同一个包中的任何类或不同包中的子类访问。



C2代码可以访问x:C1,y:C1,z:C1(通过super.z),w,z:C2
C3代码可以访问x:C1,y:C1,z:C2,w:C2(不能访问z:C1)
C4代码可以访问x,y
C5代码可以访问x

11.6 访问控制符和修饰符final

- final可以修饰变量、方法、类
- final修饰变量
  - final成员变量：常量，数据初始化后不能再修改。
  - final局部变量:常量，数据初始化后不能再修改。
- final修饰方法（实例方法和静态方法）：最终方法，实例方法不能被子类覆盖，静态方法不能被隐藏
  - Object类的getClass( )
- final类：最终类，不能派生子类。
  - String, StringBuffer
  - Math