Chapter 11 继承和多态

11.1 类继承、子类和父类的ISA关系

11.1.1 基本语法

```
class Class2 extends Class1{
    // Class Body
}
```

Class1称为父类（superclass），Class2称为子类（subclass）。

11.1.2 继承特性

- 子类继承父类中可访问的数据和方法，子类也可以添加新的数据和方法
- 子类不继承父类的构造函数
- 一个类只能由一个直接父类
- 父类的成员继承到子类，访问权限不变
- 父类的私有属性在子类中不可见，但是可以通过所继承的get和set方法访问和设置。

11.1.3 初始化块

初始化块分为实例初始化块和静态初始化块，是java类中可以出现的第四种成员（前三种：方法，属性，构造函数）。

实例初始化块(instance initialization block)

- 一个用大括号括住的语句块，直接嵌于类中，不在方法内
- 它的作用就像把它放在了类的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，还可以截获异常
- 一般来说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，初始化时在类中按顺序执行

```
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
  }
}
```

静态初始化块

- 由static修饰的初始化模块
- 只能访问类的静态成员
- 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）
- 一个类可以有多个静态初始化模块，类被加载时，这些模块按照在类中的顺序执行

```
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
  }
}
```

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

```
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");
    }
}

// 输出： 0 1 2 3 4 5 6 7 8
```

11.2 Super关键字

- 利用super关键字可以显示调用父类的构造函数
  - 调用方法：`super(parameters);`
  - 必须是子类构造函数的第一条语句且仅一条语句
  - 如果子类构造函数中没有显示调用父类构造函数，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模块中均不能使用，this也是
  - super.dataName 可以访问父类中非私有的属性
  - super.method(paramaters) 可以调用父类中的方法
  - 不能链式调用，即不能`super.super.data`

11.3 方法覆盖

子类重新定义了从父类中继承的**实例方法**称为方法覆盖（override），方法覆盖有以下约束和特点：

- 父类的方法必须是可以访问的，即私有方法不能被覆盖
- 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
- 一旦父类中的方法被子类覆盖，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。**
- 在子类函数中可以使用super调用被覆盖的父类方法
- 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)。

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

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(){
        System.out.println("B's m");
    }

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

详解：引用变量o有二个类型：声明类型A，实际运行时类型B。

判断o.s()执行的是哪个函数按照o的声明类型，因为静态函数s没有多态性，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。

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。

11.4 Object类

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）。

实现equals

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。

实现clone

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
- **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（按成员赋值）。
- 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。

11.5 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，每一个子类的实例都是父类的实例，但是反过来不成立。

```
Class Student extends Person{ …}
Person p = new Student();//OK 父类引用可直接指向子类对象
Student s = new Person();//error
```

11.5.1 多态

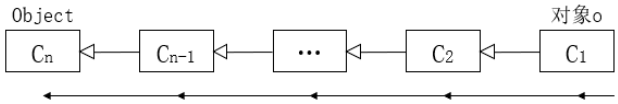
多态指的是通过引用变量调用实例函数时，根据所引用的实际对象的类型，执行该类型的相应实例方法，从而表现出不同的行为称为多态。通过继承时覆盖父类的实例方法实现多态。多态实现的原理：在运行时根据引用变量指向对象的实际类型，重新计算调用方法的入口地址（晚期绑定）。

```
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
指向子类对象
        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

    }
}
```

11.5.2 动态绑定

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

11.5.3 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`

- 从父类到子类必须强制类型转换

  `Manager m = (Manager)p; // Manager m = p 编译会报错`

- 从父类到子类转换必须显示转换，转换前应进行检查更加安全

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

11.5.4 summary

- 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。