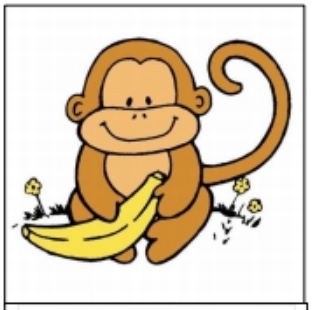
final与abstract修饰符

多态与重载

多态与重载

- 如果在方法的参数中巧用多态技术，有时可以不用方法重载
- 试想动物园中的饲养员既要给猴子投食（香蕉），还要给兔子投食（胡萝卜），给猫投食小鱼，……

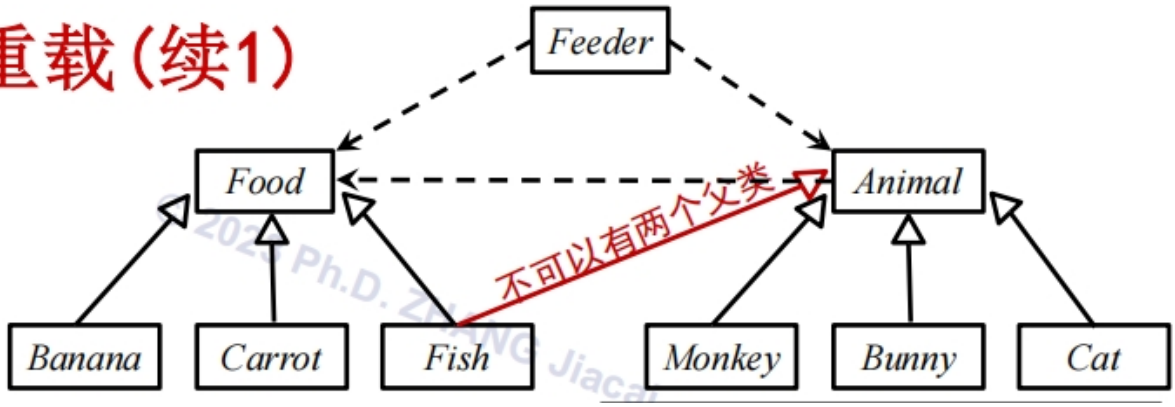


```
public class Feeder{
    public void feed(Monkey m, Banana b){}
    public void feed(Bunny b, Carrot, c){}
    public void feed(Cat c, Fish f){}
    …
}
```

© 2023 Ph.D. ZHANG Jiacai

UML图：

多态与重载(续1)



JAVA

```java
abstract class Food{}  // 定义抽象类
class Banana extends Food{}
class Carrot extends Food{}
class Fish extends Food{}

abstract class Animal{  // 定义抽象类
        abstract public void eat(Food f);  // 定义抽象方法
}

class Monkey extends Animal{
        public void eat(Food f){  // 对抽象方法的实现
                Banana banana = (Banana)f;  // 向下强制类型转换
                System.out.println("peel the banana...");
        }
}
class Bunny extends Animal{
        public void eat(Food f){
                Carrot carrot = (Carrot)f;
                System.out.println("wash the carrot...");
        }
}
class Cat extends Animal{
        public void eat(Food f){
                Fish fish = (Fish)f;
                System.out.println("grab the fish");
        }
}

class Feeder{
        public void feed(Animal a, Food f){
                a.eat(f);
        }
}
class TestFeeder{
        public static void main(String args[]){
                // 尽可能把变量声明为继承树中父类类型
                Feeder feeder = new Feeder();
                Animal animal = new Monkey();
                Food food = new Banana();
                feeder.feed(animal, food); // 执行Feeder方法

                Animal another = new Bunny();
                Food dinner = new Carrot();
                feeder.feed(another, dinner);
        }
}
```

1. **抽象类：**
    - `Food` 是一个抽象类，表示食物。

- `Animal` 也是一个抽象类，表示动物。它有一个抽象方法 `eat(Food f)`，意味着任何继承 `Animal` 的子类都需要实现这个方法。

2. **类之间的关系：**
   - `Banana`、`Carrot` 和 `Fish` 是 `Food` 的子类，表示具体的食物。
   - `Monkey`、`Bunny` 和 `Cat` 是 `Animal` 的子类，表示具体的动物。每种动物都喜欢吃一种特定的食物。
3. **重写：**
   - `Monkey`、`Bunny` 和 `Cat` 都重写了父类 `Animal` 的 `eat` 方法，实现了各自吃食物的行为。
4. **类型转换：**
   - 在每个动物的 `eat` 方法中，都有向下类型转换的操作。例如，猴子只吃香蕉，所以将传入的 `Food f` 转换为 `Banana` 类型。
5. **Feeder 类：**
   * `Feeder` 类有一个 `feed` 方法，接受一个动物和一个食物作为参数，并调用动物的 `eat` 方法来喂食动物。

这段代码展示了多态性的使用，即我们可以通过父类引用指向子类对象，并调用重写的方法。例如，在 `TestFeeder` 类中，我们可以使用 `Animal` 引用指向 `Monkey` 对象，并调用 `Monkey` 的 `eat` 方法。

> `TestFeeder` 类的 `main` 方法具体执行过程如下：
>
> 1. 创建一个 `Feeder` 对象 `feeder`。
> 2. 创建一个 `Monkey` 对象 `animal`，但请注意，这个对象的引用类型是 `Animal`，这是**多态性**的一个体现。即使我们使用 `Animal` 类型的引用来指向 `Monkey` 对象，我们仍然可以调用 `Monkey` 类中重写的方法。
> 3. 创建一个 `Banana` 对象 `food`。
> 4. 使用 `feeder` 对象的 `feed` 方法，将 `animal` 和 `food` 作为参数传入。在 `feed` 方法内部，会调用 `animal` 的 `eat` 方法。由于 `animal` 实际上是一个 `Monkey` 对象，因此会执行 `Monkey` 类中重写的 `eat` 方法。在这个方法中，会将传入的 `Food` 类型参数向下转型为 `Banana` 类型，然后输出“peel the banana…”。
> 5. 接着，创建了一个 `Bunny` 对象 `another` 和一个 `Carrot` 对象 `dinner`。
> 6. 使用 `feeder` 对象的 `feed` 方法，将 `another` 和 `dinner` 作为参数传入。类似地，会调用 `another` 的 `eat` 方法，由于 `another` 实际上是一个 `Bunny` 对象，因此会执行 `Bunny` 类中重写的 `eat` 方法。在这个方法中，会将传入的 `Food` 类型参数向下转型为 `Carrot` 类型，然后输出“wash the carrot…”。

> 总的来说，`TestFeeder` 类的 `main` 方法通过创建不同的动物和食物对象，并使用 `Feeder` 类的 `feed` 方法来测试多态性和重写方法的执行。

这里多态还是重载

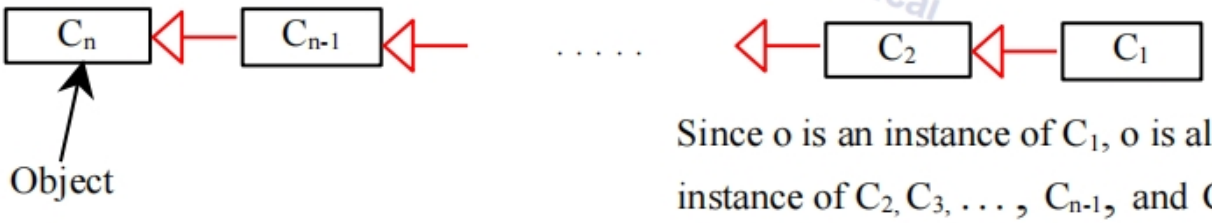
- m(x)方法可调用x对象作为实参
  - x可以是任何类型对象
  - 打印语句会自动调用x.toString()
- 这里没有重载，涉及的知识点
  - 多态，方法参数利用多态
  - 动态绑定：C1类的实例调用某个方法，将会从C1，C2，…Cn顺序寻找方法的实现：优先子类匹配

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```



Since o is an instance of C_1, o is also an instance of $C_2, C_3, \ldots, C_{n-1}$, and C_n

JAVA

```java
public class PolymorphismDemo {
        public static void main(String[] args) {
                m(new GraduateStudent());
                m(new Student());
                m(new Person());
                m(new Object());
        }
        public static void m(Object x) {
                System.out.println(x.toString());
        }
}

class GraduateStudent extends Student {}

class Student extends Person {
        public String toString() {
                return "Student";
        }
}

class Person extends Object {
        public String toString() {
                return "Person";
        }
}
```

这段代码展示了Java中的**多态性和方法覆盖**。
首先，我们来看类之间的关系：

- `GraduateStudent` 是 `Student` 的子类。
- `Student` 是 `Person` 的子类。
- `Person` 是 `Object` 的子类。
  在Java中，每个对象都有一个 `toString()` 方法，因为它继承自 `Object` 类。这个 `toString()` 方法**可以被覆盖**，正如 `Student` 和 `Person` 类中所做的那样。

现在，来看 `m` 函数。它接受一个 `Object` 类型的参数，并打印其 `toString()` 方法的返回值。

接下来，我们*看 `main` 方法中的调用：*

1. `m(new GraduateStudent());`
   - 由于 `GraduateStudent` 没有覆盖 `toString()` 方法，所以它会使用其父类 `Student` 的 `toString()` 方法。因此，输出为 "Student"。
2. `m(new Student());`
   - 这里直接调用了 `Student` 的 `toString()` 方法，所以输出为 "Student"。
3. `m(new Person());`
   - 这里直接调用了 `Person` 的 `toString()` 方法，所以输出为 "Person"。
4. `m(new Object());`
   - 这里直接调用了 `Object` 的 `toString()` 方法。默认的 `Object.toString()` 方法通常会返回对象的类名和哈希码。因此，输出可能是类似于 "PolymorphismDemo1@15db9742"的字符串，其中"PolymorphismDemo1" 是匿名子类的名称，"@" 之后的部分是哈希码。

final 修饰符

介绍

- `final`具有**“不可改变的”**的含义，它可以修饰类、成员方法和变量
  - 用`final`修饰的**类**：不能被继承，没有子类
  - 用`final`修饰的**方法**：不能被子类的方法覆盖
  - 用`final`修饰的**变量**：表示常量，只能被赋值一次
- Java中修饰符基本都*只能用于类和类成员*
  - `final`可*用于局部变量*，其余修饰符不可用于局部变量
  `public final class Company{}` 对类用`final`修饰
  `public final void getDetail(){}` 对方法用`final`修饰

final类

- Java可以将类定义为`final`，这样的类**不可以被子类继承**，这主要是要确保安全，`final`类修饰的**引用变量没有多态问题**
  - 比如*String类*，使用`String`类的地方不会用子类代替
- 在以下情况，可以考虑把类定义`final`类型，使得这个类不能被继承：
  - 出于安全的原因，类的实现细节不允许有任何改动
  - **确信这个类不会被继承和扩展**

◈ 例如JDK中的java.lang.String类被定义为final类型：

public final class String{…}

◈ 以下MyString类试图继承String类，这会导致编译错误：

public class MyString extends String{…} //编译出错

◈ String类定义的变量没有多态的问题，没有方法覆盖和动态绑定

final方法

- 使用final定义的方法，不可以在子类中被重写
- 优化程序，编译器可以知道直接调用哪个函数，而不用虚函数一样动态查找

```
public class King{
    public (final) void marry(){
        //不允许娶叶赫那拉氏的女子
        ……
    }
}

public class SubKing  extends King{
    public void marry(){
        //允许娶叶赫那拉氏的女子
        ……
    }
}
```

在这段代码中，`final` 关键字用于修饰 `marry` 方法，表示该方法在 `King` 类中是最终版本，**不能被子类重写**（override）。

final变量

- final定义的变量是*常数*，不可以修改
  - 如果final定义一个引用变量，那么变量**不可以引用其它变量**，但**可以修改引用的对象的内容**
- 例如在`java.lang.Integer`类中定义了两个常量：

```java
//表示int类型的最大值
public static final int MAX_VALUE= 2147483647;
//表示int类型的最小值
public static final int MIN_VALUE= -2147483648;
```

- final类型的成员变量**只能赋值一次**，
  - 可以*声明变量直接赋值*（显式初始化），后面不可更改， **final通常与static一起修饰成员变量**
  - 也可以*声明不指定值*（blank final变量），可以在*构造器*中赋值一次，一般不用`static`
- final类型的局部变量可以显示初始化，也可声明不赋值，但在作用域内**能且只能赋值一次**

final示例

```
public final class FinalTest {
    final static int i = 10;
    public FinalTest() { }
    public final int getNumber() {
        i = 20; //illegal
        return i;
    }
}
```

```
public class Customer{
  static final int BASE_SALARY =1500;
  private final long customerID;
  public Customer(){ customerID=createID(); }
  public long createID(){}
}
```

```
class FinalSub extends FinalTest {} //illegal

class FinalDemo {
   final int getNumber() { return 10; }
}
class FinalDemoSub extends FinalDemo {
    int getNumber() {  //illegal
        return 20;
    }
}
```

© 2023 Ph.D. ZHANG Jiacai

abstract 抽象与实现

abstract关键字

- `abstract`修饰符可用来修饰类和成员方法
- 与`final`关键字相反，`abstract`定义的类表示抽象类
  - 抽象类**必须被继承**才能实例对象
  - 抽象类可以包含*成员变量，具体方法和抽象方法*
  - 抽象类（`abstract`类）表示抽象概念，**不可以被实例为对象**
  - 比如java中的`Number`类，仅仅表示数字概念，只能实例化它的子类如`Integer、Double`等
- `abstract` 修饰的方法称为抽象方法
  - 抽象方法只有声明，没有方法体，没有实现
  - 有方法体的方法称为**具体方法**
  - `abstract` 定义的方法后面会**重写**
  - `static`和`abstract`修饰符不能在一起使用，没有静态抽象方法

``

举例：

JAVA

```
public abstract class Vehicle{
        private String name;
        public String getName(){return name;}
        public void setName(String name){this.name = name;}
        // 抽象类也可以定义具体方法，这些具体方法可以被继承
        public abstract void move(); // 定义抽象方法，不需要再添加大括号
}
public class Car extends Vehicle{
        public void move(){...} // 子类需要将抽象方法实现
}
public class Ship extends Vehicle{
        public void move(){...} // 子类需要将抽象方法实现
}

Vehicle v1=new Vehicle(); //非法，抽象类不能实例化
Vehicle v2=new Car();
Vehicle v3=new Ship();
v2. move();
v3.move();
```

抽象方法

- 抽象类*可以包含抽象方法*，为所有子类都定义了一个方法接口，抽象方法**只需声明，不需实现**
  - 子类要么也用`abstract`定义为抽象类
  - 子类要么**实现所有父类定义的抽象方法**
- 抽象类可以定义**属性，方法，构造器**
  - 抽象类不一定包含抽象方法
  - 一旦某个类包含了抽象方法，这个类必须是抽象类
  - 抽象类也可以直接调用静态变量与静态方法

JAVA

```
public abstract class Base{
        abstract void method1();
        public void method2(){;}
        private Base(){}
} // 虽然不创建对象，但可以定义构造器给子类用

class Sub1 extends Base{
        void method1(){}
        public void method2(){}
        private Sub1(){}  // 因为父类设为private，所以会报错
}

abstract class Sub2 extends Base{
        public void method2(){}
        private Sub2(){}
} // 没有实现所有抽象方法，所以仍为抽象类
```

例子2：在抽象类中直接调用静态变量

```java
abstract class Animal{
        public static int CLASS_CONST = 10;
        public void getDetail(){}
        abstract String getCategory();
}

public class Dog extends Animal{
        String getCategory(){return "Dog";}
        public static void main(String args[]){
                //Animal c=new Animal();
                System.out.println(Animal.CLASS_CONST);
        }
}
```

Q:既然子类中必须要重新实现父类中的抽象方法，父类自己不会创建对象，自己又不用这个抽象方法，父类中定义的抽象方法不会用到，那么父类又何必去定义抽象方法，为什么不直接留给子类来定义？

> 父类中定义抽象方法的原因是为了**强制子类实现某些特定的行为或功能**。通过定义抽象方法，父类可以确保子类具有相同的方法签名，从而保证子类在运行时能够正确地执行这些方法。这种机制有助于建立良好的**类层次结构和代码复用性**。
> 虽然父类自己不会创建对象，也不会直接调用这些抽象方法，但通过这些方法的定义，父类为子类提供了一个**统一的接口或行为规范**。这样，在使用子类对象时，可以确保它们具有预期的行为，而无需关心具体的子类实现细节。
> 如果父类不定义抽象方法，而是留给子类来定义，那么在使用子类对象时，就需要对每个子类的具体实现进行了解，这会增加代码的复杂性和维护难度。通过定义抽象方法，父类可以为子类提供一个清晰的、一致的接口，从而提高**代码的可读性和可维护性**。

总结

抽象方法与抽象类

```
GeometricObject[] geo = new GeometricObject[10];
```

- 抽象方法不能包含在非抽象类中
  - 如果抽象类Base的子类Sub没有实现Base中的全部抽象方法，那么Sub类也必须声明为抽象类
  - 如果非抽象类Sub继承了抽象类Base，则在Sub类中必须实现Base中全部的抽象方法，即便这些抽象方法实现后也没有调用
- 抽象类不能创建对象
  - 不能new构造器来创建对象，抽象类可以用来声明引用类型的变量
  - 但可以在抽象类中定义构造器，抽象类的构造器留给非抽象的子类调用
- 没有抽象方法的抽象类，虽然方法都实现了，但这个类是抽象类
  - 不能调用构造器生成对象
  - 这个类很可能留着给子类继承用
- 抽象类和抽象方法在UML中通常用斜体表示
- 具体类的子类可能是抽象类
  - 如Object是具体类，但其子类GeometricObject是抽象类

© 2023 PKU. ZHANG Jiacai

举例（UML图画法）

抽象类示例