# 1数据类型

## 1.1基本数据类型

byte short int long char float double boolean

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 基本类型 | 默认值 | 存储 | 封装类 | 数据范围 |
| byte | 0 | 1字节 | Byte | -128~127 |
| short | 0 | 2字节 | Shrot | -32768~32767 |
| Int | 0 | 4字节 | Integer | -21亿~21亿 |
| long | 0L(0l) | 8字节 | Long | -2^63~2^63-1 |
| char | ‘\u0000’ | 2字节 | Character | 0~65535 |
| float | 0.0F(0.0f) | 4字节 | Float | 32位EEE754单精度 |
| double | 0.0(0.0d()D) | 8字节 | Double | 64位EEE754双精度 |
| boolean | false | 1位 | Boolean | true/false |

## 1.2引用数据类型

接口 类 数组 String等

## 1.3数据类型转换

通常在进行整形数据处理时，会存在数据溢出，即数据过大或太小产生的错误数据的情况。这时候便需要进行数据类型转换。

描述小数默认类型为double类型，需要定义为float类型的时候需要在数字后加上f或者F如：1.2f。

1. 范围⼩的数据类型可以⾃动变为数据范围⼤的数据类型（在数学计算时）

2. 数据范围⼤的数据类型只有强制转换才能转为数据类型⼩的数据类型。

然后是字符类型，字符类型可以和int型的数据相互转换，但需要注意char类型中的‘0’~‘9’和数字0~9所代表的范围并不同。

最后是布尔类型，在java中的布尔类型与其他语言不同，在其他语言中用0代表false!0表示true，但是在java中并没有这种表达方式，中能用true和false表示。

String作为一种引用类型，其实是一个类，在后面的总结中会详细介绍。

# 2运算符

## 2.1基本运算符

+ - \* / %

这里需要注意的是前置还是后置的问题，后置先运算，后进行自增自减。前置相反。

### 2.2三目运算符

（运算符，结果为boolean类型）？结果1 : 结果2;

例如: 1 > 2 ? x- y: y - x;

等价于

If(1>2){

x-y;

}else{

y-x;

}

### 2.3关系运算符

> < = <= >= == :返回类型均为boolean类型（注意在比较引用类型的时候不可以用==，而应该使用.equals方法，这个在后面会详细介绍）

### 2.4逻辑运算符

(&和&&为与) (|和||为或) (!为非)

其中|| 和 &&代表短路或和短路与，在进行逻辑判断的时候都运用短路与和或。

### 2.5位运算符

& | ~ ^分别代表按位与，按位或，按位取反，以及按位异或

&:同为1才为1，其余为0

|：有一个1就为1，反之为0

~：1变为0，0变为1

^: 相同为0，不同为1

# 3 Java中的逻辑控制

## 3.1 分支结构

If(布尔表达式){

true：表达式1;

}else{

false：表达式2;

}

可以一个if对应多个else代表不同分支，也支持嵌套。

switch(数字|枚举|字符|字符串){

case 内容1 : {

内容满⾜时执⾏语句;

[break;]

} case 内容2 : {

内容满⾜时执⾏语句;

[break;]

} ... default:{

内容都不满⾜时执⾏语句;

[break;]

}

}

需要注意，break通常不能省略。

## 3.2 循环结构

**while循环**

white（循环条件）{

循环体；

更新循环变量；

}

**do...while()循环**

do{

循环体；

更新循环变量；

}while(循环条件)

**for()循环**

for（循环变量初始值;循环终止条件;循环标量更新）{

循环体;

}

## 3.3 循环控制

break 直接结束循环循环。

continue 跳出本次循环，进行下一次循环条件判断。

# 4 方法的定义和使用

## 4.1 方法的定义

方法是一段可以被重复调用代码，格式如下：

访问权限修饰符 (static) (final)返回值类型 方法名（参数列表）{

方法体代码;

return 返回值;

}

## 4.2 方法的重载

在同一个类中方法名相同，参数列表不同（顺序或个数）的方法，需要注意，返回值类型不做要求，也就是说，不能存在只有返回值类型不同的两个方法。

## 4.3 方法的递归

递归就是方法自己调用自己，需要有一个递归的结束条件，而且每次递归的参数需要做一定的调整，保证能越来越趋近于结束条件。所有的分治思想，都可以采用递归的方式去实现。

## 4.4 方法的调用

一般的方法都是需要通过实例化对象来调用的，但是加了static修饰的方法，不依赖于对象，可以直接通过类名调用。具体的会在面向对象的时候详细介绍。

# 5 面向对象

在《java核心技术 卷一》这本书中对面向对象是这样定义的：面向对象的程序是由对象组成的， 每个对象包含对用户公开的特定功能部分和隐藏的实 现部分。程序中的很多对象来自标准库，还有一些是自定义的。究竟是自己构造对象， 还是 从外界购买对象完全取决于开发项目的预算和时间。但是， 从根本上说， 只要对象能够满足 要求，就不必关心其功能的具体实现过程。在 OOP 中， 不必关心对象的具体实现，只要能 够满足用户的需求即可。

所以我们了解到在面向对象中引入了类与对象的概念。

## 5.1 类

类是构造对象的模板，在类中定义了对象的属性以及行为。我们可以将类想象成打造铁器的模具，它规定了打造出来的铁器的形状与大小。而对象就是打造出来的铁器。打造铁器的这个过程，我们将其称为类的实例化。在平常开发中我们几乎所有的代码都在某个类的内部，在java的类库中提供了很多的类，为我们日常的开发提供了很多帮助，但是这些类想要达到我们的目的是远远不够的，所以我们还要自己创造更多的类，来达到我们想要的结果。

封装是面向对象有的一个重要概念。从形式上看， 封装不过是将数据和行为组合在一个包中， 并对对象的使用者隐藏了数据的实现方式。对象 中的数据称为实例代码块, 操纵数据的过程称为方法， 对于每个特定的 类实例（对象）都有一组特定的实例域值。这些值的集合就是这个对象的当前状态。 无论何时，只要向对象发送一个消息，它的状态就有可能发生改变。 实现封装的关键在于绝对不能让类中的方法直接地访问其他类的实例域。程序仅通过对 象的方法与对象数据进行交互。封装给对象赋予了“ 黑盒” 特征， 这是提高重用性和可靠性的关键。 这意味着一个类可以全面地改变存储数据的方式，只要仍旧使用同样的方法操作数据， 其他对象就不会知道或介意所发生的变化。

## 5.2 对象

同一个类的所有实例化对象具有类似的属性和行为，这些行为需要对象调用方法来实现，我们可以用一个类来描述一个对象，但是对象却不能表示一个类，这就相当于现实中，可以把桃子说成是水果，但是却不能说水果就是桃子。这里的水果就相当于一个类，而桃子便是类的实例化对象。

我们下来就用一段代码来表示对象的实例化过程化，以及调用成员方法的过程。

class Student{

private String name;//学生的姓名，private表示私有

private int age;//学生的年龄

private String school;//学校名称

public Student(){

//构造函数

}

public Student(String name,int age, String school){

//构造函数的重载

//this 表示当前对象的引用，可以调用本类属性和方法 this.name=name;//this 调用本类属性

this.age=age; this.school=school; this.read;//this调用本类方法 } public void read(){

//成员方法

System.out.println("读书");

}

}

在类的实例化中，我们只需要使用new关键字来进行实例化，这一过程包括两个步骤：1.开辟内存空间（new关键字实现）

1. 调用合适的构造方法。代码如下：（省略主类代码）：

Student s1=new Student();//(调用不带参数的构造方法)

Student s2=new Student(“李雷”,25,”清华大学”);//(调用带有参数的方法)

s1.read;

s2.read;//调用类的成员方法，即对象的行为。

## 5.3 继承

OOP 的另一个原则会让用户自定义 Java 类变得轻而易举，这就是：可以通过扩展一个 类来建立另外一个新的类。事实上， 在 Java 中， 所有的类都源自于一个“ 神通广大的超类”， 它就Object。而这个过程就是面向对象的另一个重要特征：继承。利用继承，我们可以基于已存在的类构造一个新类。继承已存在的类就是复用（继承）这些类的方法和域。在此基础上，还可以添加一些新的方法和域， 以满足新的需求。这是 Java 程序设计中的一项核心技术。

在java中一个类只能继承自一个类，也就是单继承，需要实现多继承，我们需要借助接口来实现。我们下面来看一段继承的代码;

class Person{

private String name;

private int age;

static {//静态代码块，在类加载时就就执行了，并且只执行一次

}

public Person(String name,int age){//调用合适的构造函数

this.name=name;//this 关键字表示当前对象的引用

this.age=age;

}

public void setName(){//提供一系列的get set方法

this.name=name;

}

}

class Student extends Person{

private String school;//对父类的属性进行扩充

static{ //静态代码块

}

public Student(String name, int age,String school){

super(name,age);//利用super调用父类构造函数。必须写在第一行。

this.school=school;

}

public void func(){ //子类的成员方法

}

}

在这段代码中我们可以发现，继承只继承了属性和成员方法，并没有继承父类的构造方法。并且在子类中可以用super 关键字来调用父类的属性和方法。注意 用final修饰的类不可以被继承。

## 5.4 多态

多态是面向对象的第三个重要的特征，多态的体现主要体现在方法的多态和对象的多态。

方法的多态：方法的重载与覆写都是多态的体现。

对象的多态：在抽象类和接口中，一个类实现了抽象类的抽象方法，后者接口，在进行实例化的时候，发生向上转型。（前提条件是发生了方法的覆写），代码示例如下：

class Person{

private String name;

private int age;

static {//静态代码块，在类加载时就就执行了，并且只执行一次

}

public Person(String name,int age){//调用合适的构造函数

this.name=name;//this 关键字表示当前对象的引用

this.age=age;

}

public void setName(){//提供一系列的get set方法

this.name=name;

}

}

class Student extends Person{

private String school;//对父类的属性进行扩充

static{ //静态代码块

}

public Student(String name, int age,String school){

super(name,age);//利用super调用父类构造函数。必须写在第一行。

this.school=school;

}

public void func(){ //子类的成员方法

}

}

多态是面向对象编程的一个重要特征，其本质是在方法区存放的方法表中，在向上转型中，子类的方法覆盖了父类的方法，从而最终的结果为调用的子类的方法。

## 5.5 代码执行顺序

在面向对象的过程中，发生了继承关系，往往代码的执行顺序会变得比较复杂，下面我们来用一段代码来分析一下，在这个过程中代码的执行顺序究竟是什么样的。

public class Parent {

private int age;

private String name;

public Parent(){

System.out.println("parent.init()");

}

{

System.out.println("parent.instance");

}

static {

System.out.println("parent.static");

}

public void func(){

System.out.println("parent.func()");

}

}

public class Son extends Parent{

private String school;

public Son(){

System.out.println("Son.init()");

}

{

System.out.println("Son.instance()");

}

static{

System.out.println("Son.static");

}

public void func(){

System.out.println("Son.func()");

}

}

public class TestDemo {

public static void main(String[] args) {

Son son=new Son();

Parent parent=new Son();

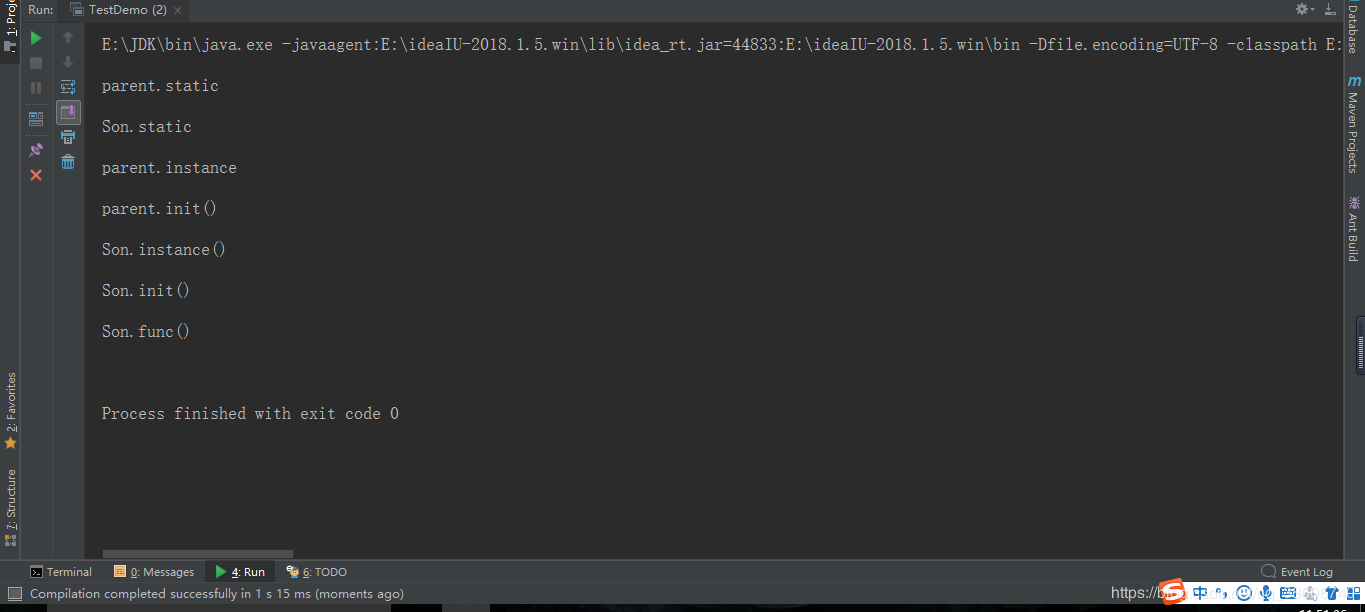
parent.func();

son.func();

}

}

这段代码的运行结果如下图，



从这个运行结果我们可以看出来首先执行父类的静态代码块，然后子类静态代码块，接下来是父类属性已经构造方法，最后是子类属性和构造方法。成员方法看调用顺序。

## 5.6 内部类

内部类顾名思义就是一个类的内部嵌套另外一个类，利用内部类可以方便操作外部类的私有成员，但是却会破坏程序的结构，使程序的逻辑显得比较混乱。

下面我们便来看一下内部类的具体分类。

### 5.6.1 成员内部类

成员内部类就是在一个类的内部嵌套一个普通类，需要注意成员内部类定义静态成员变量的时候必须背final所修饰。具体代码如下：

//代码示例：

class OuterClass{

private int age;

private String name;

public static int data=11;

public OuterClass(){

System.out.println("外部类构造函数");

}

class Inter{

//实例内部类定义静态成员变量必须使用final修饰

public static final int age=10;

public Inter(){

System.out.println("内部类构造函数");

}

public Inter(int age,String name){

//实例内部了消耗了额外的内存空间，因为实例内部类具有外部类的this引用

OuterClass.this.age=age; OuterClass.this.name=name;

}

public void print(){

System.out.println("内部类成员方法");

}

}

public void print(){

System.out.println("外部类成员方法");

}

}

public class TestDemo1 {

public static void main(String[] args) {

//实例内部类的实例化需要借助外部类对象

OuterClass.Inter inter=new OuterClass().new Inter(); inter.print();

}

}

### 5.6.2 静态内部类

静态内部类，静态内部类就是外部类嵌套了的被static修饰的类。由于被static所修饰，所以静态内部类的实例化可以不需要借助外部类对象来创建。代码如下：

class OuterClass{

private int age;

private String name;

public static int data=11;

public OuterClass(){

System.out.println("外部类构造函数");

}

static class Inter{

public static int age=10;//静态内部类定义静态成员变量不需要使用final修饰

public Inter(){

System.out.println("内部类构造函数");

}

public void print(){

System.out.println("外部类成员方法");

}

}

public void print(){

System.out.println("内部部类成员方法");

}

}

public class TestDemo1 {

public static void main(String[] args) {

//静态内部类的实例化

OuterClass.Inter inter = new OuterClass.Inter(); inter.print();

}

}

### 5.6.3 方法内部类

⽅法内部类定义在外部类的⽅法中，局部内部类和成员内部类基本⼀致，只是它们的作⽤域不同，⽅法 内部类只能在该⽅法中被使⽤，出了该⽅法就会失效。 对于这个类的使⽤主要是应⽤与解决⽐较复杂的问题，想创建⼀个类来辅助我们的解决⽅案，到那时⼜不希望这个类是公共可⽤的，所以就产⽣了局部内部类也就是方法内部类。需要注意方法内部类不允许使用任何访问修饰符，代码如下：

class OuterClass{

private int age;

private String name;

public static int data=11;

public OuterClass(){

System.out.println("外部类构造函数");

}

public void print(){

class Inter{//本地内部类，不允许被修饰符修饰

public static final int age=10;

public Inter(){

System.out.println("内部类构造函数");

}

}

new Inter();

}

}

public class TestDemo1 {

public static void main(String[] args) {

OuterClass outerClass=new OuterClass();

outerClass.print();

}}

### 5.6.4 匿名内部类

匿名内部类其实就是⼀个没有名字的⽅法内部类，所以它符合⽅法内部类的所有约束.除此之外，还有⼀

些地⽅需要注意:

1. 匿名内部类是没有访问修饰符的。

2. 匿名内部类必须继承⼀个抽象类或者实现⼀个接⼝

3. 匿名内部类中不能存在任何静态成员或⽅法

4. 匿名内部类是没有构造⽅法的，因为它没有类名。

5. 与局部内部相同匿名内部类也可以引⽤⽅法形参。此形参也必须声明为 final

代码如下：

interface Myinterface{

void print();

}

class OuterClass{

private int age;

public String name;

public void Myinterface(){

new Myinterface() {//不允许使用访问修饰符修饰

@Override

public void print() {

System.out.println("匿名内部类实现接口");

}

}.print();

}

}

public class TestDemo1 {

public static void main(String[] args) {

OuterClass outerClass=new OuterClass();

outerClass.Myinterface();

}

}

### 5.6.5 内部类的执行顺序

内部类的使用往往使得程序的结构变得混乱，所以我们需要看一下程序的执行顺序，我们借助代码来看一下。

public class Outer {

private int age;

private String name;

public Outer(){

System.out.println("Outer.init()");

}

{

System.out.println("Outer.instance()");

}

static {

System.out.println("Outer.static()");

}

public void func(){

System.out.println(name+age);

}

class Inter{

public static final int age2=10;

private String name2;

public Inter(){

System.out.println("Inter.init()");

}

{

System.out.println("Inter.instance()");

}

public void func(){

System.out.println(name+age);

}

}

}

public class TestDemo {

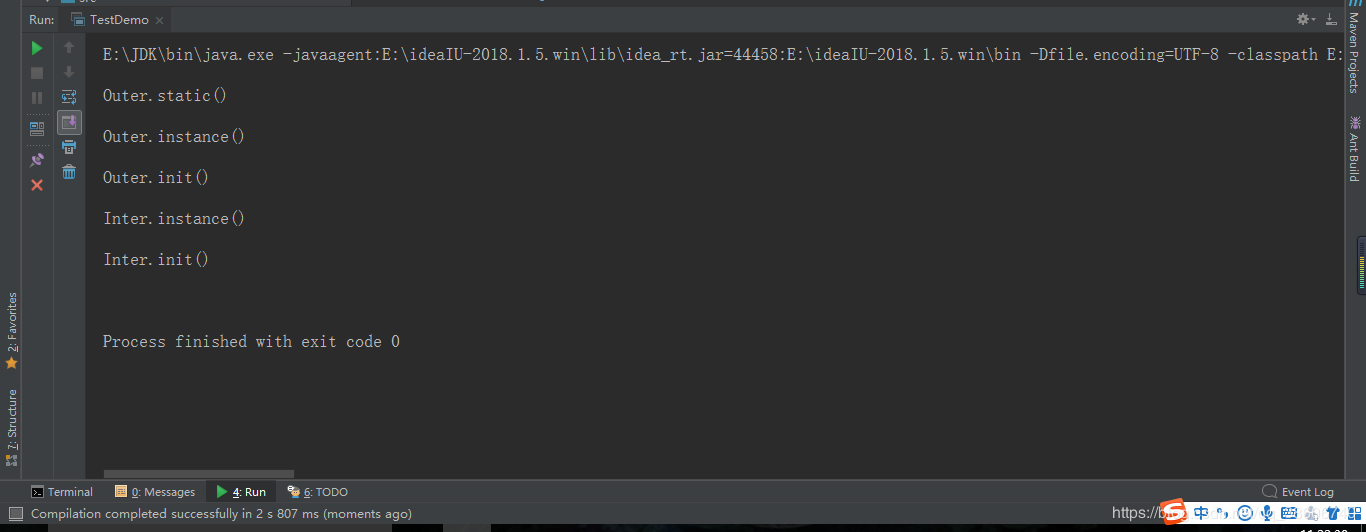
public static void main(String[] args) {

Outer.Inter inter=new Outer().new Inter();

}

}

运行结果：



从运行结果我们可以看出，首先执行的是外部类的静态代码块，外部类实例代码块，外部类构造函数，然后是内部类实例代码块，内部类构造函数。最后还有内部类的成员方法，代码中并没有加入。需要注意的是，实例内部类也可以定义静态的成员属性，不过需要用final修饰，代表这个属性在编译时期就已经确定下来了，不能再修改。静态内部类大体和实例内部类相似，不过内部类要调用外部类成员，需要提供有外部类引用的构造函数。

# 6 java中的数组

## 6.1 数组的定义

Java中的数组与C语言中的类似，都是一组相关类型的变量集合，并且这一系列变量可以按统一的方式操作。

首先是数组的定义，

语法格式是：类型[] 数组名称=new 类型[数组长度];

//代码示例：

int[] arr=new int[10];//定义一个长度为10的数组，下标从0-9.

arr[0]=1;//动态初始化。

arr[1]=2;

int[] arr2={0,1,2};//静态初始化，在Java中未初始化的数组默认填充为0

注意在使用时不能发生数组越界，即使用的空间超过了数组所开辟的空间。同时也必须为数组开辟空间，即实例化之后才可以使用。数组在内存中的存储是，数组的引用即首元素的地址存放在栈上，数组元素存放在堆上，同一块堆可以被不同的引用指向（这也是浅拷贝的基础）

代码示例：

int[]arr1=null;

int[] arr2=null;

arr1[0]=1;

arr1[1]=2;

arr1[2]=3;

arr2=arr1;

arr2[0]=30;

最后的结果是arr1[0]的值为30，因为将arr1赋给arr2即他两个指向同一个数组，arr2元素改变arr1中的元素也相应的发生了变化。java中的二维数组与C语言中的不同，java中二维数组是不规则的。代码示例：

int[2][3]arr={{1,2}{1,2,3}}

这个数组在java中指的是有两个一维数组，每个一维数组最长为三个元素，但不一定是三个。数组也可以通过方法进行处理。代码示例：

class ArrDemo{

public static void showArray(int []arr){//方法接收数组

for(int i=0;i<arr.length;i++){//arr.length 返回数组的长度

System.out.println(arr[i]);

}

}

public static int[] init(){//方法返回数组

int[]arr1={1,2,3,4};

return arr;

}

}

```

## 6.2 java对数组的支持

首先是数组的四种拷贝方式

（1）利用for循环拷贝，这种方式和C语言中相同，不再赘述。

（2）Arrays.copy,利用Arrays类中的copy方法实现对数组的拷贝，代码示例：

int[]array={1,2,3,4,5,6,7,8,9}

int[]array2=Arrays.copyof(array,array.length);//参数，（原数组，拷贝长度）

（3）System.arrayCopy,System类中的arrayCopy方法，用native修饰（这种拷贝速度最快）代码示例

System.arrayCopy(array,0,array2,0,array.length);

（4）clone代码示例

array2=array.clone();

这四种拷贝方式早拷贝基本类型的时候，都是深拷贝，只有在引用类型的时候是浅拷贝。

然后是Arrays类中对数组的操作支持

Arrays.binarySearch(arr,0,arr.length,5); /\*二分查找法查找数组中的元素,如果找到

返回元素下标，如果没找到就以负数返回该值将要插入的位置，参数分别为，需要查找的数组

查找的起始位置，查找的结束位置，查找的值\*/

arr2=Arrays.copyOf(arr,4); //拷贝数组为一个新长度，截取或者用0填充

arr2=Arrays.copyOfRange(arr,0,3); //拷贝数组的指定范围

Arrays.equals(arr,arr2); //判断两个数组深层是否相等，返回布尔类型

Arrays.fill(arr2,2,5,4); /\*将指定的值填充到数组的指定位置，参数分别为，被填充

数组，其实填充位置，终止填充位置（不包括），填充值\*/

Arrays.sout(arr); //对arr数组进行升序排序

System.out.println(Arrays.toString(arr)) //将arr中的内容以字符串的形式返回。

# 7 抽象类与接口

## 7.1 抽象类的定义和使用。

抽象类只是在普通类的基础上扩充了⼀些抽象⽅法⽽已，所谓的抽象⽅法指的是只声明⽽未实现的⽅法 （即没有⽅法体）。所有抽象⽅法要求使⽤**abstract**关键字来定义，并且抽象⽅法所在的类也⼀定要使 ⽤**abstract。代码示例：**

abstract class Person{

private String name ; // 属性

public String getName(){ // 普通⽅法

return this.name;

}

public void setName(String name){

this.name = name ;

}

// {}为⽅法体，所有抽象⽅法上不包含⽅法体

public abstract void getPersonInfo() ; //抽象⽅法

}

在使用抽象类的时候，需要注意，抽象类中并不一定有抽象方法，但抽象方法必定是在抽象类或者接口里面的。再定义一个抽象类时候，这个抽象类必须有子类，在一个非抽象子类中，必须要实现当前抽象类的抽象方法，这也就规定了抽象类在定义时不允许使用final修饰。抽象类本身不能实例化，但可以借助子类利用多态性实现实例化。同时抽象类由于被继承，所以其子类同样遵循继承原则。

## 7.2 接口的定义与使用

抽象类与普通类相⽐最⼤的特点是约定了⼦类的实现要求，但是抽象类存在单继承局限。如果要约定⼦ 类的实现要求并避免单继承局限就需要使⽤接⼝。 在开发过程之中：接⼝优先（在⼀个操作既可以使⽤抽象类⼜可以使⽤接⼝的时候，优先考虑使 ⽤接⼝）

代码如下：

interface IMessage{

public static final String MSG = "Hello World" ; // 全局常量

public abstract void print() ; // 抽象⽅法

}

⼦类如果要想使⽤接⼝，那么就必须使⽤implements关键字来实现接⼝，同时，⼀个⼦类可以实现多 个接⼝，【可以使⽤接⼝来实现多继承的概念】对于接⼝的⼦类（不是抽象类）必须覆写接⼝中的全部抽象方法。

代码如下：

interface IMessage{

public static final String MSG = "I am a biter" ; // 全局常量

public abstract void print() ; // 抽象⽅法

}

interface INews {

public abstract String getNews() ;

}

class MessageImpl implements IMessage,INews {

public void print() {

System.out.println(IMessage.MSG) ;

}

public String getNews(){

return IMessage.MSG ; // 访问常量都建议加上类名称

}

}

public class Test{

public static void main(String[] args) {

IMessage m = new MessageImpl() ; //⼦类向上转型,为⽗接⼝实例化对象

m.print() ; // 调⽤被⼦类覆写过的⽅法

INews n = (INews) m ;

System.out.println(n.getNews()) ;

}

在这里就是多态性的一个体现，实际产生的对象是 new产生的实例化对象，而不是前面的那个。这就好比说，水果 桃子=new 桃子。最终产生的对象是桃子一样。在接口中，所有的属性和方法都是public的，所以weight保持代码的简洁性，我们在使用接口时，通常不写public。

## 7.3 接口和抽象类的区别。

|  |  |  |  |
| --- | --- | --- | --- |
| No | 区别 | 抽象类 | 接口 |
| 1 | 结构组成 | 普通类+抽象方法 | 抽象方法+全局变量 |
| 2 | 权限 | 各种权限 | public |
| 3 | 子类使用 | extends | implements |
| 4 | 关系 | 可以实现多个接口 | 不能继承抽象类，可以继承多个接口 |
| 5 | 子类限制 | 单继承 | 利用接口实现多继承 |

## 7.4 设计模式

### 7.4.1 模块设计模式

模块设计模式是抽象类的一个实际运用场景，比如说平常煮茶时的几个步骤。

1. 将⽔煮沸

2. ⽤沸⽔浸泡茶叶

3. 把茶倒进杯⼦

4. 加佐料

u示例代码：

class Tea {

void prepareRecipe() {

boilWater();

steepTeaBag();

pourInCp();

addLemon();

}

将⽔煮沸

public void boilWater() {

System.out.println("Boiling Water");

}

冲泡茶

public void steepTeaBag() {

System.out.println("Steeping the tea");

}

把茶倒进杯⼦中

public void pourInCup() {

System.out.println("Pouring into cup");

}

加佐料

public void addSeasoning() {

System.out.println("Adding Seasoning");

}

}

我们可以发现，其实在冲泡奶茶咖啡等饮料时其实都是使用的同样的操作，所以我们便可以将这作为一个模板，来实现不同的饮料制作。

代码如下：

\* 基类声明为抽象类的原因是

\* 其⼦类必须实现其操作

\*/

abstract class AbstractClass {

/\*\*

\* 模板⽅法，被声明为final以免⼦类改变这个算法的顺序

\*/

final void templateMethod() {

}

/\*\*

\* 具体操作延迟到⼦类中实现

\*/

abstract void primitiveOperation1();

abstract void primitiveOperation2();

/\*\*

\* 具体操作且共⽤的⽅法定义在超类中，可以被模板⽅法或⼦类直接使⽤

\*/

final void concreteOperation() {

// 实现

}

/\*\*

\* 钩⼦⽅法是⼀类"默认不做事的⽅法"

\* ⼦类可以视情况决定要不要覆盖它们。

\*/

void hook() {

// 钩⼦⽅法

}

}

扩展上述类，引⼊"钩⼦"⽅法

超类实现:

abstract class CaffeineBeverage {

final void prepareRecipe() {

boilWater();

brew();

pourInCup();

// 如果顾客想要饮料我们才调⽤加料⽅法

if (customerWantsCondiments()){

addCondiments();

}

}

abstract void brew(); abstract void addCondiments();

void boilWater() {

System.out.println("Boiling water");

}

void pourInCup() {

System.out.println("Pouring into cup");

}

/\*\*

\* 钩⼦⽅法

\* 超类中通常是默认实现

\* ⼦类可以选择性的覆写此⽅法

\* @return

\*/

boolean customerWantsCondiments() {

return true;

}

}

⼦类实现:

class Tea extends CaffeineBeverage {

void brew() {

System.out.println("Steeping the tea");

}

void addCondiments() {

System.out.println("Adding Lemon");

}

}

class Coffee extends CaffeineBeverage {

void brew() {

System.out.println("Dripping Coffee through filter");

}

void addCondiments() {

System.out.println("Adding Sugar and Milk");

}

/\*\*

\* ⼦类覆写了钩⼦函数，实现⾃定义功能

\* @return

\*/

public boolean customerWantsCondiments() {

String answer = getUserInput();

if (answer.equals("y")) {

return true; }else {

return false;

}

}

private String getUserInput() {

String answer = null;

System.out.println("您想要在咖啡中加⼊⽜奶或糖吗 (y/n)?");

Scanner scanner = new Scanner(System.in);

answer = scanner.nextLine();

return answer;

}

}

测试类:

public class Test {

public static void main(String[] args) {

CaffeineBeverage tea = new Tea();

CaffeineBeverage coffee = new Coffee();

System.out.println("\nMaking tea...");

tea.prepareRecipe();

System.out.println("\nMaking Coffee");

coffee.prepareRecipe();

}

通过这个例子，我们可以发现，其实很多具有相同操作的过程，都可以使用模板设计模式来使这些代码变得更加简单一些。

### 7.4.2 简单工厂模式

简单工厂模式。有一个抽象的产品的接口，需要具体的产品类去实现这个接口，然后需要有一个工厂类去产生具体产品。最后用客户端去实现买电脑。

代码如下：

public interface Computer {//首先定义一个电脑接口

void print();

}

class Macbook implements Computer{//具体产品类

@Override

public void print() {//MacBook去实现Computer接口

System.out.println("MacBook");

}

}

class SurfaceBook implements Computer{//具体产品类

@Override

public void print() {//SurfaceBook去实现Computer接口

System.out.println("SurfaceBook");

}

}

class ComputeFactory {//产品工厂类

public static Computer getInstance(String type){//返回一种具体产品

Computer computer=null;

switch (type){

case "Mac":

computer=new Macbook();break;//生产Macbook

case "Sur":

computer=new SurfaceBook();break;//生产SurfaceBook

default:

System.out.println("请输入型号");break;//其他型号

}

return computer;//返回生产的电脑

}

}

import java.util.Scanner;//导入Scanner类

public class Client {//客户端类

public static void Buy(Computer computer){//接收电脑型号买电脑

computer.print();

}

public static void main(String[] args) {

Scanner scanner=new Scanner(System.in);

String type=scanner.nextLine();//用户输入需要购买的型号

Computer computer=ComputeFactory.getInstance(type);//根据用户输入型号生产电脑

Buy(computer);调用买电脑方法，传过去用户需要购买的型号买到电脑。

}

}

简单工厂模式的优点顾名思义就是简单，缺点是，在工厂中添加其他型号电脑时违反了OCP封装原则。

### 7.4.3 工厂方法模式

工厂方法模式。有一个产品接口，需要具体的产品实现产品接口，有一个工厂接口，需要具体的工厂实现工厂接口。同样用代码来详解。

public interface Computer {//产品接口

void print();

}

public interface ComputerFactory {//工厂接口

Computer CreatComputer();//生产一种具体产品

}

public class MacBook implements Computer{//具体产品类MacBook

@Override

public void print() {//实现产品接口

System.out.println("MacBook");

}

}

public class SurfaceBook implements Computer {//具体产品类SurfaceBook

@Override

public void print() {//实现产品接口

System.out.println("SurfaceBook");

}

}

public class MacFactory implements ComputerFactory {//具体工厂类MacFactory

@Override

public Computer CreatComputer() {//实现工厂接口

return new MacBook();//生产MacBook

}

}

public class SurfaceFactory implements ComputerFactory {//具体工厂类SurfaceFactory

@Override

public Computer CreatComputer() {

return new SurfaceBook();//生产SurfaceBook

}

}

public class Client {//客户端类，实现购买电脑

public void BuyComputer(Computer computer){//根据型号购买电脑

computer.print();

}

public static void main(String[] args) {

Client client=new Client();

ComputerFactory computerFactory=new SurfaceFactory();//生产SurfaceBook

client.BuyComputer(computerFactory.CreatComputer());//购买生产出的电脑

}

}

工厂方法类的优点是降低了代码的耦合度，实现了开放封闭原则，添加新的产品类的时候不需要修改原有的代码，缺点同样也很明显，代码量增多，并且在添加抽象产品的时候，比如Mac添加苹果手机，同样会违反OCP封装原则。

### 7.4.4 抽象工厂模式

public interface Computer {//抽象产品接口电脑

void print();

}

public interface OpreatSystem {//抽象产品接口操作系统

void print();

}

public interface Factory {//抽象工厂接口

Computer createComputer();//生产电脑

OpreatSystem createOpreatSystem();//安装电脑操作系统

}

class MacBook implements Computer {//具体产品类MacBook

@Override

public void print() {

System.out.println("MacBook");

}

}

class SurfaceBook implements Computer {//具体产品类SurfaceBook

@Override

public void print() {

System.out.println("SurfaceBook");

}

}

class MacFactory implements Factory {//具体工厂类Mac工厂

@Override

public Computer createComputer() {

return new MacBook();//生产MacBook

}

@Override

public OpreatSystem createOpreatSystem() {

return new MacOs();//安装MacOs系统

}

}

class SurfaceFactory implements Factory {//具体工厂Surface工厂

@Override

public Computer createComputer() {

return new SurfaceBook();//生产SurfaceBook

}

@Override

public OpreatSystem createOpreatSystem() {

return new Win10();//安装Win10系统

}

}

public class Client {

public void BuyComputer(Computer computer){

computer.print();//购买电脑的型号

}

public void Os(OpreatSystem opreatSystem){

opreatSystem.print();//购买电脑的操作系统

}

public static void main(String[] args) {

Client client=new Client();

Factory factory=new MacFactory();//使用MacFactory

client.BuyComputer(factory.createComputer());//买MacFactory生产出的电脑

client.Os(factory.createOpreatSystem());//购买的电脑操作系统是

}

}

抽象工厂模式总体上和工厂方法模式差不多，是对工厂方法模式的扩充，抽象工厂接口可以灵活扩充，缺点也同工厂方法模式，并且代码量更大，在扩展具体产品时同样会违反OCP封装原则。

### 7.4.5 代理设计模式

interface ISubject {

public void buyComputer() ; // 核⼼功能是买电脑

}

class RealSubject implements ISubject {

public void buyComputer() {

System.out.println("买⼀台外星⼈电脑") ;

}

}

class ProxySubject implements ISubject {

private ISubject subject ; // 真正的操作业务

public ProxySubject(ISubject subject) {

this.subject = subject ;

}

public void produceComputer() {

System.out.println("1.⽣产外星⼈电脑") ;

}

public void afterSale() {

System.out.println("3.外星⼈电脑售后团队") ;

}

public void buyComputer() {

this.produceComputer() ; // 真实操作前的准备

this.subject.buyComputer() ; // 调⽤真实业务

this.afterSale() ; // 操作后的收尾

}

}

class Factory {

public static ISubject getInstance(){

return new ProxySubject(new RealSubject()) ;

}

}

public class Test{ public static void main(String[] args) {

ISubject subject = Factory.getInstance() ;

subject.buyComputer() ;

}

代理模式的本质：所有的真实业务操作都会有⼀个与之辅助的⼯具类（功能类）共同完成。