# 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();

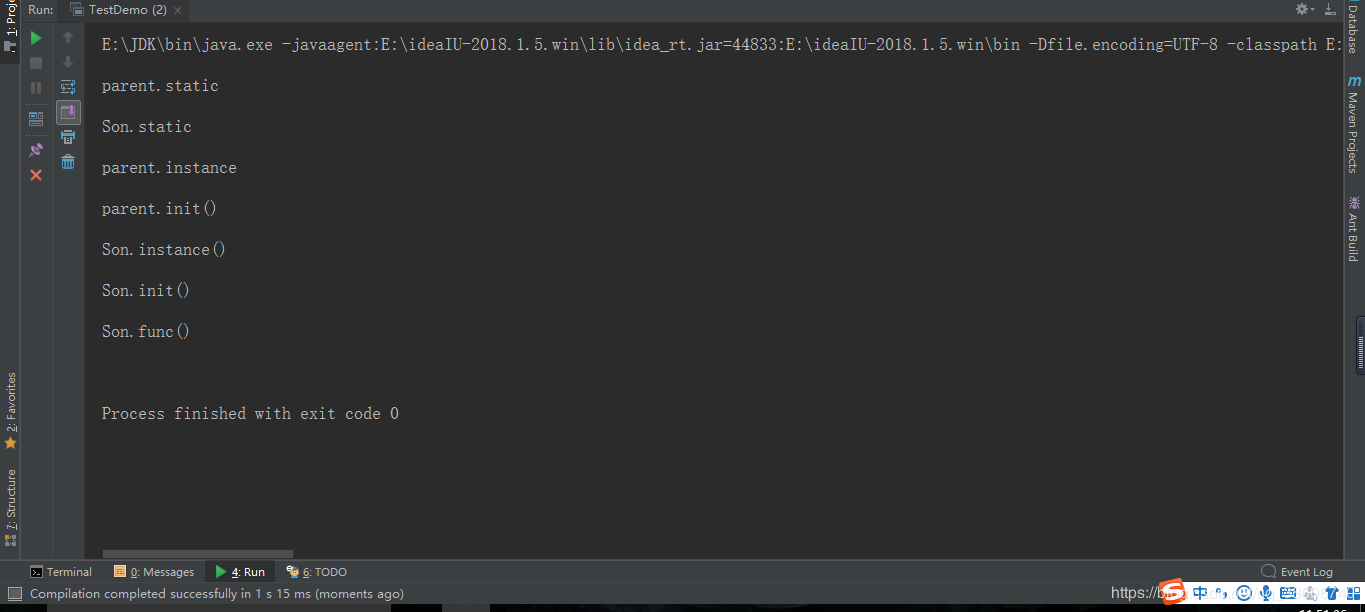
parent.func();

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