**JAVA基础**

# 基本数据类型的包装类

8大基本数据类型 byte short int long float double char boolean

包装类 Byte Short Integer Long Float Double Character Boolean

|  |
| --- |
| **public** **class** WrapBaseClass {  **public** **static** **void** main(String[] args) {  **int** i = 123;  // 创建引用类型变量inte，参数可传入字符串  Integer inte = **new** Integer("123");  System.***out***.println(inte);  // 将引用类型变量inte转换为基本数据类型int  **int** k = inte.intValue();  System.***out***.println(k);  // 自动装箱 基本类型的变量直接赋给对应的包装类变量  Integer in = i;  // 自动拆箱 包装类变量直接赋给对应的基本类型的变量  i = inte;    **int** l = Integer.*parseInt*("678");  Float.*parseFloat*("12.22");  Boolean.*parseBoolean*("false");    System.***out***.println(l);  // java.lang.NumberFormatException,数据格式化异常  // 注意在转换时，后面的内容一定要是可转换为前面相应的数据类型的数据，  // 否则会报如上的异常信息  // double m = Double.parseDouble("abc");  // System.out.println(m);    // nullpointexception classcastexception  // arrayindexoutofboundsexception  // NumberFormatException  // 前面大于后面返回1 相等返回0 前面小于后面返回-1  System.***out***.println(Integer.*compare*(102, 102));  }  } |

# ==与equals

* Object 类提供的toString方法总是返回该对象实现类的类名 + @ +hashCode值。
* ==和equals比较运算符：==要求两个引用变量指向同一个对象才会返回true。equals方法则允许用户提供自定义的相等规则(重写)。

Object类提供的equals方法判断两个对象相等的标准与==完全相同。因此开发者通常需要重写equals方法

|  |
| --- |
| String s1 = "abc";  String s2 = "abc";  String s3 = **new** String("abc");  String s4 = **new** String("abc");  // == 比较的是地址  System.***out***.println(s1 == s2);// true  System.***out***.println(s1 == s3);// false  System.***out***.println(s3 == s4);// false    // equals比较的是值  System.***out***.println(s1.equals(s2)); // true  System.***out***.println(s1.equals(s3)); // true  System.***out***.println(s3.equals(s4)); // true      Student stu1 = **new** Student("张三","上海市");  Student stu2 = **new** Student("张三","上海市");  System.***out***.println(stu1.equals(stu1));// true  System.***out***.println(stu1 == stu2);// false  System.***out***.println(stu1.equals(stu2));// true  C:\Users\tzhang\AppData\Local\Temp\1490071914(1).png |
| **public** **class** Student {  // public String sno;  **public** String name;  **public** String address;  // 类成员，通过类名.的方式直接访问,这种类成员被所有类共享，一次变化  // 影响所有，慎用！  **public** **static** String *gender*;  **public** Student(String name, String address) {  **super**();  **this**.name = name;  **this**.address = address;  }  @Override  **public** String toString() {  **return** "Student [name=" + name + ", address=" + address + "]";  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj) {  **return** **true**;  }  **if** (obj **instanceof** Student) {  Student stu = (Student) obj;  **if** (**this**.name.equals(stu.name) && **this**.address.equals(stu.address)) {  **return** **true**;  }  }  **return** **false**;  }  } |
| * **用static修饰的类成员属于类成员，类成员既可通过类来访问，也可以通过类的对象来访问。当通过对象来访问类属性时，系统会在底层转换为通过该类来访问类属性。** * **类成员并不是属于实例，它是属于类本身的。只要类存在，类成员就存在。** * **即使通过null对象来访问类成员，程序也不会引发NullPointerException。** * **类成员方法不能直接访问实例成员，与类的加载有关系。** |
| **public** **class** StaticDemo {  // 静态属性/变量， 单个类中使用可以根据需要而定  **private** **static** String *name*;  **static** Student *stu1*;    **private** String address;    // 静态方法  **public** **static** **void** main(String[] args) {  // static方法中访问外部成员，1通过对象访问，二将外部成员  // 使用static修饰  System.***out***.println(*name*);  StaticDemo sd = **new** StaticDemo();  System.***out***.println(sd.address);    Student.*gender* = "男";  // 访问外部类的类成员  System.***out***.println(Student.*gender*);// 男  // System.out.println(stu1.name);// nullpointexception  // 当通过对象来访问类属性时，系统会在底层转换为通过该类来访问类属性  System.***out***.println(*stu1*.*gender*);// 男    Student stu = **new** Student("z", "d");  // 公共(public)属性直接通过点访问，私有属性通过get、set方法访问  System.***out***.println(stu.name);  System.***out***.println(stu.*gender*);// 男    Student.*gender* = "女";  System.***out***.println(Student.*gender*);// 女  System.***out***.println(*stu1*.*gender*);// 女  System.***out***.println(stu.*gender*);// 女  }  } |

# 单例类

* 如果一个类始终只能创建一个对象，称为单例类。须符合以下几个条件：
  + 1.我们把该类的构造器使用Private修饰，从而把该 类的所有构造器隐藏起来。
  + 2.则需要提供一个public方法作为该类的访问点，用于创建该类的对象，且必须使用static修饰

3.该类还必须缓存已经创建的对象，必须用static修饰

|  |
| --- |
| **public** **class** SingleInstance {  // 在程序运行周期内，一个类只会被加载一次，static修饰的变量  // 随类的销毁而销毁  // 该类还必须缓存（常驻内存）已经创建的对象  **static** SingleInstance *s*;    **private** SingleInstance(){    }  // 懒汉式单例，调用时创建分配空间，在高并发情况下，线程不安全  **public** **static** SingleInstance getInstance(){  **if**(**null** == *s*) {  *s* = **new** SingleInstance();  }  **return** *s*;  }  } |
| **public** **class** EhsSingleInstance {  // 饿汉式 类加载时创建  **private** **static** EhsSingleInstance *ehs* = **new** EhsSingleInstance();    **private** EhsSingleInstance(){    }    **public** **static** EhsSingleInstance getInstance() {  **return** *ehs*;  }  } |
| **public** **class** MainSingleInstance {  **static** SingleInstance *si1*;  **public** **static** **void** main(String[] args) {  *si1* = SingleInstance.*getInstance*();  SingleInstance si2 = SingleInstance.*getInstance*();  System.***out***.println(*si1* == si2);  }  } |

# Final

* 成员变量是随类的初始化或对象初始化而初始化的。final修饰的成员变量必须由程序员指定初始值。
* 对于类属性而言，要么在静态初始化块中初始化，要么在声明该属性时初始化。
* 对于实例属性，要么在普通初始化块中指定初始值。要么在定义时、或构造器中指定初始值。
* final 修饰的类不可以被继承
* 不可变的类要满足以下几个条件：
* 1.使用private和final修饰符来修饰该类的属性
* 2.提供带参数的构造器，用于根据传入参数来初始化类里的属性
* 3.仅为该类的属性提供getter方法，不要为该类的属性提供setter方法，因为普通方法无法修改final修饰的属性
* 4.如有必要，重写Object类中hashCode 和equals

1、final类  
        final类不能被继承，因此final类的成员方法没有机会被覆盖，默认都是final的。在设计类时候，如果这个类不需要有子类，类的实现细节不允许改变，并且确信这个类不会载被扩展，那么就设计为final类。

非final类中的final方法，可被继承

2、final方法  
        如果一个类不允许其子类覆盖（重写）某个方法，则可以把这个方法声明为final方法。  
        使用final方法的原因有二：  
        第一、把方法锁定，防止任何继承类修改它的意义和实现。  
        第二、高效。编译器在遇到调用final方法时候会转入内嵌机制，大大提高执行效率。

**一、final**  
        根据程序上下文环境，Java关键字final有“这是无法改变的”或者“终态的”含义，它可以修饰非抽象类、非抽象类成员方法和变量。你可能出于两种理解而需要阻止改变：设计或效率。

        final类不能被继承，没有子类，final类中的方法默认是final的。  
        final方法不能被子类的方法覆盖，但可以被继承。  
        final成员变量表示常量，只能被赋值一次，赋值后值不再改变。  
        final不能用于修饰构造方法。  
        注意：父类的private成员方法是不能被子类方法覆盖的，因此private类型的方法默认是final类型的。

|  |
| --- |
| **public** **class** FinalDemo {    // fianl修饰的变量一旦赋值后，就不可以改变  **public** **static** **final** **int** ***fVal*** = 10;  **public** **static** **int** *val* = 10;    **public** **static** **void** main(String[] args) {  *val* = 20;  //报错，试图改变一个使用final修饰的变量  // fVal = 20;  //报错，试图改变一个使用final修饰的变量  // Student.race = "dd";    **final** Student stu1 = **new** Student("张三", "上海市");  Student stu2 = **new** Student("李四", "苏州市");  // 赋值会报错，fianl修饰不可改变  // stu1 = stu2;  // final修饰的对象，是引用（指向地址）不能变，但是其属性值是可变的  stu1.name = "王五";  stu1.address = "南京市";    *test*("赵六");  }    **public** **static** **void** test(**final** String str) {  /\*  \* final参数可以获取到调用时传过来的实参，但是获取后就不能改变他的值  \*/  // str = "王五";  System.***out***.println(str);  }  } |

Final类没有子类

final方法可以被继承

# 抽象类

* 抽象方法和类都必须使用abstract来修饰，有抽象方法的类只能定义成抽象类，抽象类里也可以没有抽象方法。

抽象类不能被实例化，可以通过其子类给他赋值，普通类里有的抽象类里也有，定义抽象方法只需在普通方法上增加abstract修饰符，并把普通方法的方法体（也就是方法后花括号括起来的部分）全部去掉，并在方法后增加分号即可

public abstract void test();

抽象类是不完整的，它只能用作基类。在面向对象方法中，抽象类主要用来进行类型隐藏和充当全局变量的角色(子类公有的属性)

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。抽象类往往用来表征我们在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同(具体的事物)，但是本质上相同的具体概念的抽象

比如，在一个图形编辑软件的分析设计过程中，就会发现问题领域存在着圆、三角形这样一些具体概念，它们是不同的，但是它们又都属于形状这样一个概念，形状这个概念在问题领域并不是直接存在的，它就是一个抽象概念。而正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的

在面向对象方法中，抽象类主要用来进行类型隐藏。构造出一个固定的一组行为的抽象描述，但是这组行为却能够有任意个可能的具体实现方式。这个抽象描述就是抽象类，而这一组任意个可能的具体实现则表现为所有可能的派生类。模块可以操作一个抽象体。由于模块依赖于一个固定的抽象体，因此它可以是不允许修改的；同时，通过从这个抽象体派生，也可扩展此模块的行为功能。

* 抽象类的特征：有得有失，得到了新能力，可以拥有抽象方法；失去了创建对象的能力。

抽象类代表了一种未完成的类设计（抽象的概念，不能完全表示一个实体），它体现的是一种模板

|  |
| --- |
| **import** java.sql.Connection;  /\*\*  \* 包含抽象发方法的类，一定是抽象类  \* 抽象类 = 普通类（该有的抽象类都有） + 抽象方法(可有可无)  \* **@author** tzhang  \*/  **public** **abstract** **class** DealData {  **public** String name;    **public** DealData(){  }  // 抽象类的构造器不是用来创建实例的，而是用来初始化里面属性的  **public** DealData(String name) {  **this**.name = name;  }  // 抽象类可以包含抽象方法，不能实例化  // 类方法 类名.方法名调用  // public static void t() {  //  // }  // 实例方法/普通方法 实例.方法名调用  **public** **void** test(){  System.***out***.println("test");  }  // 抽象方法，没有实现，子类中实现，抽象类是用来被继承的,抽象方法可以被继承  /\*\*  \* 获取数据库连接信息  \* **@return**  \* 数据库连接  \*/  **public** **abstract** Connection getConnection();  } |
| **import** java.sql.Connection;  /\*\*  \* 包含抽象发方法的类，一定是抽象类  \* **@author** tzhang  \*  \*/  **public** **class** MysqlDb **extends** DealData {  **public** MysqlDb() {  **super**();  // **TODO** Auto-generated constructor stub  }  **public** MysqlDb(String name) {  **super**(name);    }  /\*\*  \* 采用Mysql设置参数的方式获取连接  \*/  @Override  **public** Connection getConnection() {  System.***out***.println("mysql connection");  **return** **null**;  }  } |
| **import** java.sql.Connection;  **public** **class** OracleDb **extends** DealData {  /\*\*  \* 采用oracle设置参数的方式获取连接  \*/  @Override  **public** Connection getConnection() {  System.***out***.println("oracle connection");  **return** **null**;  }  } |
| **import** java.sql.Connection;  **public** **class** Main {  // 父类变量，指向子类实例，当调用父类变量中的方法时，  // 实际上是执行变量指向的子类中的方法实现  **public** **static** **void** main(String[] args) {  // 一开始底层数据库使用这个Mysql数据库  // DealData mysqlDb = new SqlServerDb();  // mysqlDb.getConnection();    // SqlServer数据库  // DealData sqlServerDb = new SqlServerDb();  // sqlServerDb.getConnection();    // 用oracle数据库  // DealData oracleDb = new OracleDb();  // oracleDb.getConnection();    // 匿名内部类（了解）,存在一个没有名字的类，他继承自DealData，  // dealData就相当于是匿名内部类的一个实例  DealData dealData = **new** DealData() {  @Override  **public** Connection getConnection() {  // **TODO** Auto-generated method stub  **return** **null**;  }  };  }  } |

# 接口

* 接口定义的是多个类共同的行为规范，这些行为是与外部交流的通道，这就意味着接口里通常是定义一组公用的方法。
* 接口体现了规范(接口)与实现分离的设计。
  + 常量都是：public static final修饰
  + 方法都是：public abstract 修饰

接口的继承和类继承不一样，接口完全支持多继承，子接口扩展某个父接口将会获得父接口的所有抽像方法，常量属性，内部类和枚举类定义

* 接口可以用于声明引用类型的变量，但接口不能用于创建实例。
* 当使用接口来声明引用类型的变量时，这个引用类型的变量必须引用到其实现类的对象。
* 一个类可以实现一个或多个接口，继承使用extends关键字，实现接口则使用implements关键字。
* 一个类实现了一个或多个接口之后，这个类必须完全实现这些接口里所定义的全部抽象方法（也就是重写这些抽象方法）；

否则，该类将保留从父接口那里继承到的抽象方法，该类也必须定义成抽象类

* 一个类实现接口，但不实现里面的抽象方法，那么这个类怎么改？

把他改成抽象类就行，为什么？

* 组装一个电脑-》步骤-》我来实现这些步骤

|  |
| --- |
| **public** **interface** IComputer {  **void** fisrt();  **void** second();  **void** third();  } |
| **public** **class** Hp **implements** IComputer {  @Override  **public** **void** fisrt() {  System.***out***.println("惠普装主版");    }  @Override  **public** **void** second() {  System.***out***.println("惠普装散热器");    }  @Override  **public** **void** third() {  System.***out***.println("惠普连接外部配件");  }  } |
| **public** **class** Lenovo **implements** IComputer{  @Override  **public** **void** fisrt() {  System.***out***.println("联想装主版");    }  @Override  **public** **void** second() {  System.***out***.println("联想装散热器");    }  @Override  **public** **void** third() {  System.***out***.println("联想连接外部配件");    }  } |
| /\*\*  \* 对于一个类来讲，继承是单继承的（类），实现时多实现的（接口）  \* 对于一个接口来讲，继承是多继承的，接口只能继承接口  \* 接口继承会讲其父接口的常量属性和方法都继承过来  \* **@author** tzhang  \*  \*/  **public** **class** Main {  **public** **static** **void** main(String[] args) {  // 接口变量指向其实现类的实例，当调用其方法时，实际是调用其实现类对应的方法  IComputer lenovo = **new** Lenovo();  lenovo.fisrt();  lenovo.second();  lenovo.third();    IComputer hp = **new** Hp();  hp.fisrt();  hp.second();  hp.third();    // 匿名内部类，创建了一个没有名字的内部类，他实现了这个IComputer接口  // ic想相当于是匿名内部类的一个实例  IComputer ic = **new** IComputer() {    @Override  **public** **void** third() {  // **TODO** Auto-generated method stub    }    @Override  **public** **void** second() {  // **TODO** Auto-generated method stub    }    @Override  **public** **void** fisrt() {  // **TODO** Auto-generated method stub    }  };  }  } |

### 接口与抽象类的相似性

* 接口和抽象类都不能被实例化，它们都位于继承树的顶端，用于被其他类实现和继承。
* 接口和抽象类都可以包含抽象方法，实现接口或继承抽象类的普通子类都必须实现这些抽象方法。

### 接口与抽象类的区别

* 接口里只能包含抽象方法，不能包含已经提供实现的方法；抽象类则完全可以包含普通方法。
* 接口里不能定义静态方法；抽象类里可以定义静态方法。
* 接口里只能定义静态常量属性（默认public static final修饰，必须要给初始值），不能定义普通属性；抽象类里则既可以定义普通属性，也可以定义静态常量属性。
* 接口不包含构造器；抽象类里可以包含构造器，抽象类里的构造器并不是用于创建对象，而让其子类调用这些构造器来完成属于抽象类的数据初始化操作。
* 接口里不能包含初始化块，但抽象类则完全可以包含初始化块。
* 一个类最多只能有一个直接父类，包括抽象类；但一个类可以直接实现多个接口，通过实现多个接口可以弥补Java单继承的不足。

## 拓展：

自学简单工厂模式、命令模式等设计模式