**今天完成的事情：**

**第一部分：JAVA基础知识**

**内容预览：**

* JAVA基本数据类型
* 变量类型
* 修饰符
* 运算符

1. **JAVA基本数据类型**

变量就是申请内存来存储值

JAVA分为两大数据类型：

* 内置数据类型
* 引用数据类型

1. **内置数据类型**

**JAVA提供八大基本类型**

* 六种数据类型（4个整数型、两个浮点型）
* 一种字符类型
* 一种布尔型

**1# byte型**

byte 数据类型是8位、有符号的，以二进制补码表示的整数

默认值是 0

**2# short型**

short 数据类型是 16 位、有符号的以二进制补码表示的整数

默认值是 0

**3# int型**

int 数据类型是32位、有符号的以二进制补码表示的整数

默认值是 0

**4# long型**

long 数据类型是 64 位、有符号的以二进制补码表示的整数

默认值是 0L

**5# float型**

float 数据类型是单精度、32位、符合IEEE 754标准的浮点数

默认值是 0.0f

**6# double型**

double 数据类型是双精度、64 位、符合IEEE 754标准的浮点数

默认值是 0.0d

**7# booleam型**

boolean数据类型表示一位的信息

只有两个取值：true 和 false

默认值是 false

**8# char型**

char类型是一个单一的 16 位 Unicode 字符

**\*9# void型**

void类型也是一种基本数据类型，代表空

void类型也是JAVA的关键字，代表空

1. **引用数据类型**

在Java中，引用类型的变量非常类似于C/C++的指针

* 引用类型指向一个对象，指向对象的变量是引用变量

类似：A=a(int b);

A指向对象a（int b），既A为引用变量（类似指针，保存地址）

* 这些变量在声明时被指定为一个特定的类型，如Employee、Puppy等
* 对象、数组都是引用数据类型
* 所有引用类型的默认值都是null
* 一个引用变量可以用来引用任何与之兼容的类型

例如：Site site = new Site("Runoob")

site就是新建对象Site（“Runoob”）的引用变量（指针）

引用的就是对象Site保存的字符串类型

1. **JAVA常量**

JAVA 中使用 final 关键字来修饰常量，声明方式和变量类似

**常量通常使用大写字母表示**

例如：final double PI = 3.1415927

使用常量的时候，前缀 0 表示 8 进制，而前缀 0x 代表 16 进制

例如：int DECIMAL = 100;

例如：int OCTAL = 0144

例如：int HEXA = 0x64

**支持的转义字符序列**：



1. **自动类型转换**

在Java项目的实际开发和应用中，常常需要用到将对象转为String这一基本功能

在此，将对数据类型转换进行详解

**数据类型特点：**

* 整型、实型（常量）、字符型数据可以混合运算
* 运算中，不同类型的数据先转化为同一类型，然后进行运算

转换从低级到高级：

**byte & short & char—> int —> long—> float —> double**

数据类型转换必须满足如下规则：

* **不能对boolean类型进行类型转换**
* **不能把对象类型转换成不相关类的对象**
* **在把容量大的类型转换为容量小的类型时必须使用强制类型转换**
* **转换过程中可能导致溢出或损失精度**

**例如：**

**int i =128**

**byte b = (byte) i**

**因为byte类型是8位，最大值为127，所以当int强制转换为byte类型时，值128时候就会导致溢出**

* **浮点数到整数的转换是通过舍弃小数得到，而不是四舍五入**

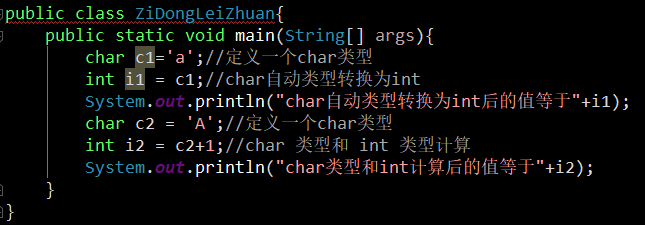
**例如：**

**(int)23.7 == 23**

**(int)-45.89f == -45**

**自动类型转换**

必须满足转换前的数据类型的位数要低于转换后的数据类型，例如：



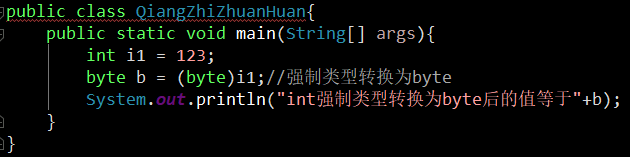
1. **强制类型转换：**

* **条件是转换的数据类型必须是兼容的**
* **格式：**

**( type ) value**

type是要强制类型转换后的数据类型

**实例：**



1. **JAVA变量类型**

在Java语言中，所有的变量在使用前必须声明

声明变量的基本格式如下：

type identifier [ = value][, identifier [= value] ....]

说明：type为Java数据类型，identifier是变量名

Java语言支持的变量类型有：

* 类变量

独立于方法之外的变量，用 static 修饰

* 实例变量（成员变量）

独立于方法之外的变量，没有 static 修饰

* 局部变量

方法中的变量

1. **局部变量**

* 局部变量声明在方法、构造方法或者语句块中
* 局部变量在被执行的时候创建，当它们执行完成后，变量将会被销毁
* 局部变量没有默认值，所以局部变量被声明后，必须经过初始化，才可使用

1. **实例变量（成员变量）**

* 实例变量声明在一个类中，但在方法、构造方法和语句块之外
* 当一个对象被实例化之后，每个实例变量的值就变成实例的参数
* 实例变量在对象创建的时候创建，在对象被销毁的时候销毁
* 实例变量的值应该至少被一个方法、构造方法或者语句块引用，使得外部能够通过这些方式获取实例变量信息
* 实例变量具有默认值。

数值型变量的默认值是0，布尔型变量的默认值是false，引用类型变量的默认值是null

变量的值可以在声明时指定，也可以在构造方法中指定

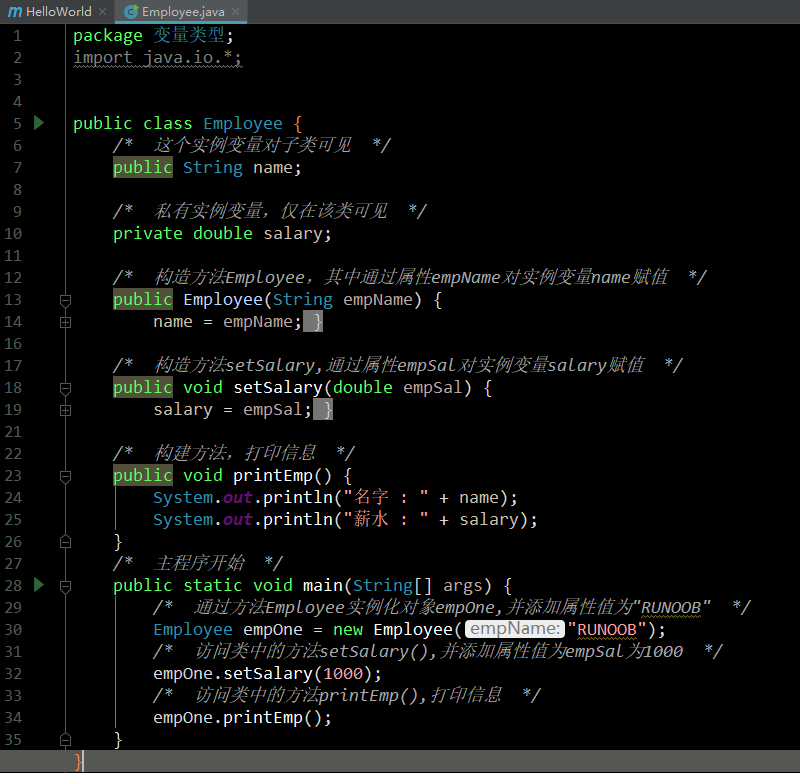
* 实例变量可以直接通过变量名访问

在静态方法以及其他类中，就应该使用完全限定名：

对象名.变量名 的方式访问

实例：Employee.java代码

// 添加了自己的理解注释 //



1. **类变量**

* 类变量也称为静态变量，在类中以static关键字声明，但必须在方法之外
* 静态变量经常被声明为常量，很少单独使用static声明变量
* 静态变量除了被声明为常量外很少使用

常量声明分为public/private，final和static四种类型。

常量初始化后不可改变

* 与实例变量具有相似的可见性。

但为了对类的使用者可见，大多数静态变量声明为public类型

* 无论一个类创建了多少个对象，类只拥有类变量的一份拷贝
* 静态变量在第一次被访问时创建，在程序结束时销毁
* 默认值和实例变量相似

此外，静态变量还可以在静态语句块中初始化

* 静态变量可以通过：类名.变量名的方式访问
* 类变量被声明为public static final类型时，类变量名称一般使用大写字母

1. **JAVA修饰符**

修饰符用来定义类、方法或者变量，通常放在语句的最前端

Java语言提供了很多修饰符，主要分为以下两类：

* 访问修饰符
* 非访问修饰符

1. **访问控制修饰符**

JAVA中，可以使用访问控制符来保护对类、变量、方法、构造方法的访问

Java 支持 4 种不同的访问权限：

* default

使用对象：类、接口、变量、方法

在同一包内可见，不使用任何修饰符（既缺省）

* private

使用对象：变量、方法

注意：不能修饰类（外部类）

在同一类内可见

* public

使用对象：类、接口、变量、方法

对所有类可见

* protected

使用对象：变量、方法

注意：不能修饰类（外部类）

对同一包内的类和所有子类可见

1. **非访问修饰符**

为了实现一些其他的功能，Java 也提供了许多非访问修饰符

**非访问修饰符大致分为五类**：

* static 修饰符

用来修饰类方法和类变量

* final 修饰符

用来修饰类、方法和变量

final 修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的

* abstract 修饰符

用来创建抽象类和抽象方法

* synchronized 和 volatile 修饰符

主要用于线程的编程

**1# static 修饰符**

* **静态变量**

static 关键字用来声明独立于对象的静态变量

静态变量也被称为类变量。局部变量不能被声明为 static 变量

* **静态方法**

static 关键字用来声明独立于对象的静态方法

静态方法从参数列表得到数据，然后计算这些数据

静态方法不能使用类的非静态变量

**2# final 修饰符**

* **final变量**

final变量一旦赋值后，不能被重新赋值

final 修饰的实例变量必须显式指定初始值

final 修饰符通常和 static 修饰符一起使用来创建类常量

* **final方法**

声明 final 方法的主要目的是防止该方法的内容被修改

类中的 final 方法可以被子类继承，但是不能被子类修改

* final类

final 类不能被继承，没有类能够继承 final 类的任何特性

**3# abstract修饰符**

* **抽象类**

声明抽象类的唯一目的是为了将来对该类进行扩充

抽象类可以包含抽象方法和非抽象方法

抽象类不能用来实例化对象

一个类不能同时被abstract和final修饰

* 抽象方法

抽象方法是一种没有任何内容的方法，该方法的的具体实现由子类提供

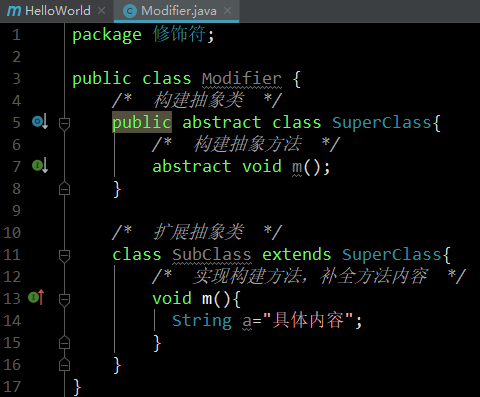
继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类

抽象方法不能被声明成 final 和 static

抽象方法的声明以分号结尾，例如：public abstract sample()

示例：

添加了自己的理解注释



**第二部分：修真任务**

**任务预览：**

* 依赖注入

1. **依赖注入（Inversion of Control）**

依赖注入是指在启动Spring容器加载Bean配置的时候，完成对变量的赋值行为

依赖注入是一种设计思想，不是编程技术

IoC是Spring的核心

IoC在加载的时候会扫描XML文件里面的Bean的相关配置，然后为这些Bean进行实例化(创建Bean)，所谓注入就是在创建的过程中完成对成员变量的赋值

IoC的一个重要功能是在系统运行中，动态的向某个对象提供它所需要的其他对象。

而实现这个功能的机制，是依靠JAVA的反射特征（reflection），反射特征允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性。

Spring所倡导的开发模式，是将所有类在Spring容器中登记，然后开发过程中，告诉Spring自己的需求，Spring会在适当的时候自动供给需求，并完成与其他程序的对接。

因此，所有的类的创造、销毁都由Spring来控制，既由Sping来控制对象的生存周期。

IoC，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系

**IoC有以下几个思考要点：**

1. **谁控制谁**

传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC拥有专门一个容器（构件）来创建这些对象

既由IoC来控制对象的创建

1. **控制了什么**

主要控制了外部资源的获取（不只是对象，也包括文件等）

1. **为何反转（有反转就有正传）**

正转：传统应用程序是由我们在对象中主动控制来获取对象的依赖（依赖调用）

反转：IoC是由容器来帮忙创建或注入依赖对象（调用对象）

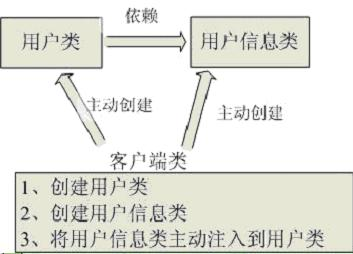
1. **哪些方面反转**

一切操作由容器来执行，容器帮助我们查找及注入依赖对象，而对象只是被动地接受依赖对象

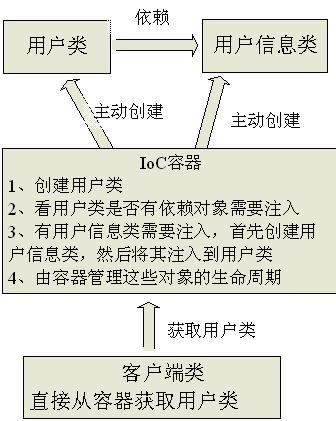
操作方式上进行了反转

图例说明：

1. 传统方式



1. IoC/DI容器操作



1. **依赖注入的形式**

**依赖注入通常有三种方式：**

* 设值注入

设值注入是指IoC容器通过成员变量的setter方法来注入被依赖对象

* 构造注入
* 接口注入

1. **设值注入**

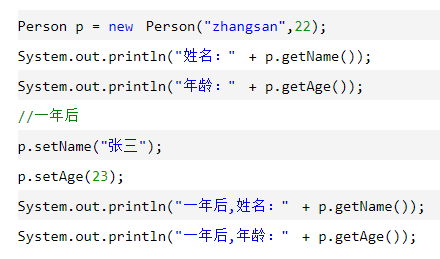
设值注入就是通过构建方法setter，在需要的地方直接引用

例如：

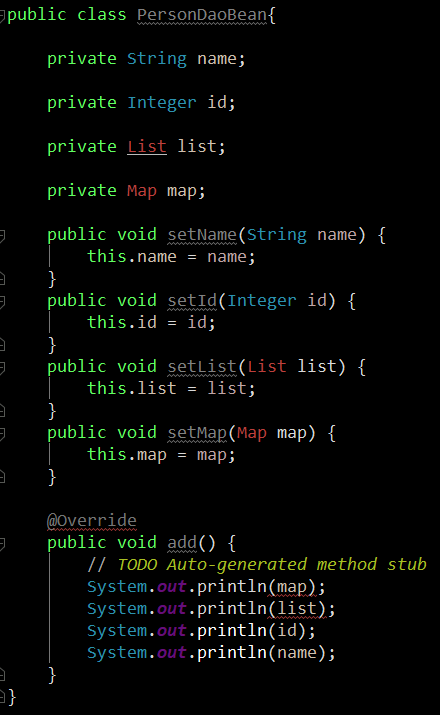
开始构建对象p（“zhangsan”，22）

在一年后需要修改对象属性

则直接调用.setter方法（既.setXxx方法）来修改



完整示例代码如下：



**this关键字**

以上示例中出现this关键字，在此做出解释

This关键字可以理解成为一个指针，指向对象本身

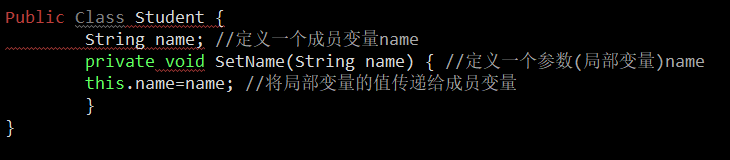
上图为例，this.name既为PersonDaoBean.name

在方法内部同理，指向对象本身

**this 的用法在 Java 中大体可以分为3种：**

1. 调用本类中的属性，也就是类中的成员变量（实例变量）

示例：



This.name即为Student.name指向对象名

1. 调用本类中的其他方法
2. 调用本类中的其他构造方法，调用时要放在构造方法的首行
3. 构造注入
4. 接口注入

**明天计划完成的事情：**

1. IoC依赖注入的学习（主要任务）
2. JAVA基础知识的学习

**遇到的问题：**

1. 一直不理解在C里构建的函数到了JAVA里面怎么叫方法

答案：原来是两种不同类型的编程思想

1. C语言面向过程编程，其中对机器的操作都是通过数据库中的函数来进行
2. JAVA/C++语言面向对象编程，在对对象的操作过程中，所执行的操作都可称呼为构建对象的行为方法，所以简称方法，但是两者含义相同
3. 对于设值注入的setter和getter方式不是很理解

答案：搜索相关资料理解了setter方式只是一种注入的编程思路

通过构建包含setXxx（）方法和getXxx（）方法的类来直接引用

达成解耦的目的，使代码复杂度降低，也减少了发生异常的可能性

**收获：**

在项目代码阅览过程中，经常碰到不熟悉的语法知识，导致对项目的理解没法进行下去，最近就一直在搜索资料，补习JAVA基础知识。随着对JAVA基础知识掌握的增加，对JAVA项目代码理解也有了一些长进，能够理解各个部分的含义。说明基础知识的补习很有意义。

另一部分修真任务，随着对Spring框架核心机制的理解，也掌握了一些新知识，应该对Spring框架的应用有一些帮助。

打好基础才好走路，很有收获。