Java学习笔记

> 经验总结

Java 简介部分

Java是怎么运行的，exe文件呢?

> 💡 Java固有“**一次编译到处运行**”的特点，其本质上便是Java内部的虚拟机的效果，在Java文件的编译中，会根据主文件(.java)生成一个.class文件，这个.class文件仅可在Java自带的虚拟机KVM中运行，正是因为如此，Java才有着良好的跨平台性能，特点。

Package包管理

Java项目的结构：

一般来说，Java项目中包含这样几个文件(夹)

out文件夹/src文件夹/.idea配置文件夹/项目配置文件

> out文件夹：包含编译完成后的自解码文件
>
> src文件夹：主类的文件夹，一般开发都在src文件夹下进行

src文件夹下有个com.company.XXX文件夹，下可以进行`package`的引入

```Java
package com.company.XXX;
```

Java第一个程序的讲解

Java

```java
package com.google.demo;    //用于定位自己的 Main 主类

//主类
public static void Main{

    //main 函数，程序入口
    public static void main(string[] args){
        System.out.println("hello");

    }
    public static void sum(int number_a,int number_b){
        return result=number_a+number_b;
    }
}
```

> 骚操作：在IDEA里，直接输入sout即可生成`System.out.println("hello");`输入soutv即可生成与最近的变量名相同的`System.out.println(XXX);`

注释与文档

在Java中，注释的方式与C语言中相同，都是`//`注释一行`/**/`注释一段，但在我们写文档时，我们可以使用`/**`+`Enter`的方式，直接生成对函数的注解，例如：

Java

```java
//求和函数
public static int sum(int number_a,int number_b){
    return number_a+number_b;
}


/**
 *该方法传递两个int类型的参数，用来求和
 * @param number_a 第一个和数
 * @param number_b 第二个和数
 * @return 返回求和后的结果
 */
public static int sum(int number_a,int number_b){
    return number_a+number_b;
}
```

> 在书写文档时，也可以用这种方式，对函数进行注解，再利用javadoc命令，对Java函数生成文档。
>
> 在写中文文档时，有时会出现中文乱码的情况，这时需要在后面加上`-encoding utf-8`

字符串操作

关于字符串的操作，比较复杂，具体可以参考JDK文档Overview (Java SE 9 & JDK 9) (oracle.com)

这里仅举几个例子

Java

```
string str_1 = "hello"
string str_2 = "World"

System.out.println(len(str_1));
//output:5
string str_3=str_1.contact(str_2);
System.out.println(str_3);
//output:HelloWorld
string str_4=str_1+str_2;
System.out.println(str_4);
//output:HelloWorld
```

> 使用`.contact()`和`+`连接字符串

字符串结束符那些事

在C语言中，我们在定义一个字符串的时候，在结尾必须要加一个`'\0'`或`null`，但在java中，我们不必对字符串的结尾过多关注，在Java的String对象中，官方已经定义好了结尾的字符。

---

import导包与API文档

在java中，依然存在许多函数的包，与python一样，可以通过`import`的方式导入包

- 有一些包中的函数是不需要导入的，使用同一个`package`下的包不需要`import`
- `java.lang.*`是java默认自带的，优先级最高

Java中的数组

在C语言中，我们定义数组可以按以下方式

C

```
int a[]={1,2,3,,4,5,65,6,12};
```

但在Java中，我们定义一个数组，常常使用以下方式：

Java

```java
int[] a={1,2,1,3,21,3412,3,12,41,12};
//或者是
int[] a=new int[5];     //new方式相当于C语言中的malloc函数，新分配一个内存空间
```

遍历数组：

Java

```java
int[] arr_1={1,2,3,4,45,12,3,21,3,24,12,34141};

for (int i = 0; i < arr_1.length; i++) {     //length表示一个变量参数
    System.out.println(arr_1[i]);
}

//JDK5后出现的增强for数组
for (int e: arr_1) {
    System.out.println(e);
}
```

Arrays的应用

此类包含用于操作数组（例如**排序和搜索**）的各种方法。此类还包含一个允许将数组视为列表的静态代理。

对应Arrays的操作：Arrays (Java SE 9 & JDK 9) (oracle.com)

一些简单的应用:

	A	B
1	public static int binarySearch(Object[] a, Object key)	用二分查找算法在给定数组中搜索给定值的对象(Byte,Int,(排序好的。如果**查找值**包含在数组中，则返回搜索键的**索引**
2	public static boolean equals(long[] a, long[] a2)	如果两个指定的 long 型数组彼此相等，则返回 true。如果 并且两个数组中的**所有相应元素对都是相等**的，则认为这两 返回一个**布尔变量**，如果两个数组以相同顺序包含相同的元 样的方法适用于所有的其他基本数据类型（Byte，short，
3	public static void fill(int[] a, int val)	将指定的 **int** 值分配给指定 **int** 型数组指定范围中的每个元 他基本数据类型（Byte，short，Int等）。
4	public static void sort(Object[] a)	对指定对象数组根据其元素的自然顺序进行**升序排列**。同样 数据类型（Byte，short，Int等）。

方法的重载

方法名相同，方法参数类型与参数个数不一定相同，返回值也可不一样

在java中，我们可以只用一个方法名，从而实现多种不同的方法的编写，例如：

C++

```
// Java中的求和函数
// int类型
public static int sum(int number_a,int number_b){
    return number_a+number_b;
}

//double类型
public static double sum(double x,double y){
    return x+y;
}

//调用时，都可以实用sum函数
int x=7,y=0;
int sum=sum(x,y);

double x_1=2.3 , y_1=7.8;
double sumDouble=sum(x_1,y_1);
```

> Java的返回值的隐式类型转换

Java的面向对象思想

POP向OOP的思维转换

POP语言，类似于C语言，整体思维为线性的，主要步骤为：

- 思考一个过程
- 第一步做什么
- 第二步做什么
- ……
- 最后一步做什么

POP语言的主要思想的缺陷：走一步看一步，目标不明确，不适用于大众。

OOP语言的思想：对过程的要求不高，明确目标，

Java面向对象实战

面向对象语言的最重要的特点就是引入了类这一结构，接下来以狗为例，阐述类的定义

我们设计一个狗的管理系统，在这个管理系统里，每个狗单独定义为一个类，这个类包含了狗的所有属性，比如，年龄，品种，名字等多种属性。

Java

```java
public class Dogs{
    string name;        //名字
    string variety;     //品种
    int age;            //年龄

void eat(){
    System.out.println("狗吃饭");
    }

void sleep(){
    System.out.println(name+"狗睡觉");
    }

}
```

接下来我们要新建(new)一个对象

TypeScript

```typescript
// 在Main.java中，调用对象

public static void Main(String[] args) {
    Dogs zhangDogs = new Dogs();     //new相当于C语言中的malloc函数，为对象分配空间
    zhangDogs.name="Jerry";
    zhangDogs.age=2;
    zhangDogs.variety="拉布拉多";//给对象zhangDogs中的成员变量初始化

    zhangDogs.sleep();//调用zhangDogs类中的sleep()方法
}
```

成员变量，行为，this

成员变量：类中定义的变量称为成员变量。

行为：类中定义的方法称为行为

this关键字：指代所调用的对象，从而可以一次性传出多个参数

注销账户与空指针异常

注销账户：

```java
zhangDogs=null;     //直接将所定义的对象指向NULL即可
```

空指针异常：

当我们注销账户后，仍然访问这个类时

```java
zhangDogs=null;     //直接将所定义的对象指向NULL即可
System.out.println(zhangDogs.name);//即会出现空指针异常
```

OOP封装

在我们的类里，如果成员变量定义为public类型的，就会出现安全隐患：

```java
//在初始化时：
zhangDogs.age=-100;//初始化出现异常，但编译通过
```

这样就会出现-100岁的狗>_<

并且会造成用户的权限无限大，用户可以为所欲为(我支付宝里面想有多少钱就改成多少钱)（乐.jpg）

但我们肯定不能在每次定义的时候都做个if判断，那么这时候，我们就需要在类中进行封装

Java

```java
//在定义一个Java类时，我们将所有的成员变量定义为 private 类型：
public class Dogs{
    private string name;        //名字
    private string variety;     //品种
    private int age;            //年龄

void eat(){
    System.out.println("狗吃饭");
    }

void sleep(){
    System.out.println(name+"狗睡觉");
    }

}
```

这样就能避免用户可以随意更改类中的成员变量

但当我们想要直接访问Dogs类中的成员变量时，就会报错：

Java

```java
System.out.println(zhangDogs.name);
//直接会报错
```

这时我们要如何访问并修改成员变量呢?

Java

```java
//我们需要定义一组getset方法来对类中的成员变量进行访问
public String getName(){
    return this.name
}

public String setName(String Name){
    this.name=Name;
}
```

> 在大型项目中，我们常使用直接生成的方法来生成一组get/set方法

在lombok库中，我们可以直接在类的定义之前加上`@Getter,@Setter`注解，从而对每个成员变量都相当于生成了一组`get/set`方法

ToString()方法

当我们格式化输出类中的成员变量时。我们不能直接输出

```Java
System.out.println(zhangDog);
```

这样输出的形式是这样：

```Apache
zhangDogs = com.company.Dogs@511baa65
```

这玩意看不明白>_<

这时，我们就可以在类中写一个toString方法来在输出时格式化输出类：

```TypeScript
@Override
public String toString() {
    return "Dogs{" +
            "name='" + name + '\'' +
            ", Age=" + Age +
            ", variety='" + variety + '\''
            }';
```

这样，当我们在需要格式化输出类Dog中的全部成员变量时，就会得到如下结果

```Bash
zhangDogs = Dogs{name='Jerry', Age=0, variety='拉布拉多'}
```

> 我们也可以使用lombok中的@ToString注解，就可以不用专门写一个`toString`函数了

构造方法

构造方法是一个特殊的方法，在定义一个类的同时，构造方法就会调用，常常用来初始化类中的成员变量

首先定义一个构造方法：

TypeScript

```
//这是一个有参构造器
public Dogs(String name, int age, String variety, String food) {
    this.name = name;
    this.Age = age;
    this.variety = variety;
    this.food = food;
}
//这是一个无参构造器
public Dogs() {
}
```

我们可以发现，构造方法有两类：有参和无参，为什么要定义一个无参构造器呢?

在我们定义一个类时，系统会默认生成一个无参的构造器，但当我们定义了一个有参构造器时，这个默认的无参构造器便会被清除，这时，便需要我们手动定义一个无参的构造器了

构造方法的重载和this关键字详解

在定义构造方法时，我们会发现，同一个类下的不同构造函数之间的关系为重载关系，当我们调用不同的构造方法时，也会实现不同的初始化方式.

在类的定义中，我们可以使用`this`关键字，来指定使用类中的成员变量或是行为，这样一种方法在类的内部也可以使用，比如:

TypeScript

```
public class Dogs {

    private String name;
    private int Age;
    private String variety;
    private String food;
    //构造方法
    public Dogs() {
    }

    public Dogs(String name, int age, String variety, String food) {
        this.name = name;
        this.Age = age;
        this.variety = variety;
        this.food = food;
    }
    //行为
    public void sleep(){
        System.out.println(name+"在睡觉");
    };

    public void eat(){
        System.out.println(name+"在吃饭");
        this.eat();
    };
```

垃圾回收机制

在前面说到对账户的注销可以利用`zhangDog=null;`来实现，但Java中所拥有的垃圾回收机制可以很好地帮助我们实现对内存占用的回收，一般为自动触发，但在有需要用手动的情况下，我们也可以使用`System.gc()`语句来手动实现对内存占用的回收

静态变量与静态方法

静态变量：

在类中，我们可以定义一个`public static AAA XXX=BBB;`这里的AAA是数据类型，XXX为变量名，BBB为所赋值。

这样的一个变量称之为静态变量，我们不能通过这个类定义的Object来访问静态变量，我们只能直接用类名去访问，例如：

Java

```java
//在定义一个Java类时，我们将所有的成员变量定义为 private 类型：
public class Dogs{
    private string name;        //名字
    private string variety;     //品种
    private int age;            //年龄
    public static String plot="南瓜小区";

void eat(){
    System.out.println("狗吃饭");
    }

void sleep(){
    System.out.println(name+"狗睡觉");
    }

}

//访问时必须使用 Dogs.plot 才有效
System.out.println(Dogs.plot);
```

output：

Java

```
南瓜小区
```

静态方法：

同静态变量一样，我们也可以定义一个静态的方法，和静态变量一样，在此不做过多赘述。

Private static

在我们定义一个public类型的静态变量时，我们就需要承受用户对成员变量的直接访问，这也会造成巨大的安全隐患，因此，推荐使用`private`关键字来定义静态变量。

Java

```java
private static String plot="南瓜小区";
```

但是当我们要去访问时，也就需要使用`get/set`方法，这时，我们可以使用如下方法：

Java

```java
public static String getPlot(){
    return plot;
}
```

外部调用这个`Plot`时，就可以使用这个函数，就不需要通过`this`传递参数了

Static单例模式

static可以完全脱离对象使用，我们也可以在类的里面定义一个静态的成员类,这时我们定义的类只能创建一次，且不提供new的方式来创建一个类。

Java

```java
public class Earth{
    private static Earth earthInstances = new Earth();

    private Earth(){
    }

    public static Earth getInstances(){
        return earthInstances;
    }

}


//在外部调用时，我们可以以下方法进行调用：
Earth earthInstances=Earth.getInstances();
```

内部类

内部类是指在一个类中定义的另外一个类，可以通过`exp1.exp2`来对内部类进行访问并定义，不常用，在此不做过多赘述

面向对象的Java

面向对象的几个基本特征：

可继承性，多态性，封装性

继承

面向对象中，要定义一个类的时候，常利用**继承**，继承可以使得子类具有父类的属性和方法或者重新定义、追加属性和方法等。我们常使用 `extends` 关键字指定所继承的类：

TypeScript

```
//假设在一个动物园管理系统中
//定义一个父类Animals
public class Animals{
    String name;
    int ages;
    String foods;

    public void eat(){
        System.out.print("在吃饭");
    }

    public void bark(){
        System.out.print("动物叫");
    }

    //构造函数
    public Animals(String name, int ages, String foods) {
    this.name = name;
    this.ages = ages;
    this.foods = foods;
    }
}

//类 Tiger 继承自Animals
public class Tiger extends Animals{
    public Tiger(String name, int ages, String foods) {
    super(name, ages, foods);
    }
}
```

> 在上述这样一个 `Tiger` 子类中，已经包含了 `Animals` 父类中的全部变量与方法，我们可以直接对 `Tiger` 的 `name` ， `foods` 属性进行访问。

Java

```java
//Main类中
public class Main {
    public void main() {
        Tiger tiger = new Tiger("jerry", 2, "meat");
        System.out.println("Tigers.foods = " + tiger.foods);
    }
}

//OutPut:
Tigers.foods = meat
```

对于一个继承自父类的子类对象，其内部已然包含了父类的全部属性和方法，要在子类中访问调用父类中的变量或方法时，常使用`super`关键字，这种情况在多层继承中用的尤其多

注：父类的private成员方法是不能被子类方法覆盖的。

重写

当一个子类继承父类的时候，我们往往需要重新给子类写一个方法，例如，上述`Animals`父类中的`bark`方法，直接调用则输出"动物叫"，但我们需要搞清楚是什么动物叫，或者哪一只动物叫，这时，我们就需要对子类进行重写

TypeScript

```typescript
@Override
public void bark(){
    System.out.println(name + "叫");
}
```

输出：

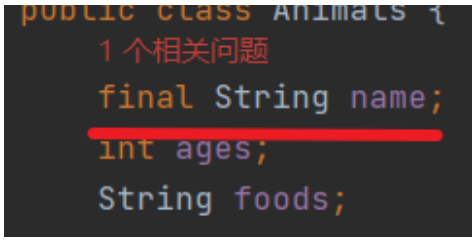
TypeScript

```
jerry叫
```

Final关键字详解

当我们定义一个类时，加上`Final`关键字，说明所定义的这个类是一个最终类，**不可被继承**，在上面的例子里面，我们可以给最终的`Tiger`类加上一个`final`关键字，说明这是一个最终类。

同样，我们也可以对变量进行`final`的修饰，说明这个变量不可被重新赋值，例如：



就会报错：

final修饰的变量有三种：静态变量、实例变量和局部变量，分别表示三种类型的常量。

抽象类与接口思想引入

Question 1:

在上面的例子里面，我们定义了一个`Animals`类，然后我们定义了很多动物的类，这些类全都继承于`Animals`类，`Animals`只用于被继承，这样，我们如果new一个`Animals`类，那这样是无意义的，会造成资源的浪费.

Question 2:

在这个`Animals`类中，我们定义了一个`bark`方法，来对动物的叫声音进行描述，我们对动物园内已有的动物已经定义好了一些类，但当我们新引入了一个动物，还没来得及对其进行写一个类的时候，这时，如果我们直接调用`bark`方法，就会出现”动物叫“这一个令人疑惑的结果。

抽象类

上面两个问题促使我们思考，能不能对类进行改变，使里面的一些属性，方法被抽象成一个统一的属性，例如，Tiger的名字"jerry"，年龄"2"，食物"meat"是一个实例，但可以将其抽象成 名字，年龄，食物等抽象的概念，这样，我们就得到了一个类，**这个类是对于继承于此类的子类的一个抽象描述**，这个类就称为**抽象类，**在上面的例子中，`Animals`便是一个抽象

我们可以通过`abstract`关键字来修饰一个抽象类

抽象类的规范：

- 抽象类不可被新建(new)
- 一个类只能继承一个抽象类
- 在抽象类中，如果某一个方法是抽象的，其他方法可以不是抽象的。
- 继承一个抽象类时，抽象类中的全部方法都必须重写。

> 抽象类的定义比较灵活，我们可以在一个抽象类中定义非抽象方法或属性，也可以将全部方法和属性定义为抽象的，这完全取决于需求。

例如；

Java

```java
public abstract class Animals{
    String name;
    int ages;
    String foods;

    public void eat(){
        System.out.print("在吃饭");
    }

    public abstract void bark();

    //构造函数
    public Animals(String name, int ages, String foods) {
    this.name = name;
    this.ages = ages;
    this.foods = foods;
    }
}


public final class Dogs extends Animals {

    public Dogs(String name, int ages, String foods) {
        super(name, ages, foods);
    }
    @Override
    public void bark(){
        System.out.println(name + "叫");
    }


}
public class Dogs extends Animals {
    public Dogs(String name, int ages, String foods) {
        super(name, ages, foods);
    }
    @Override
    public void bark(){
        System.out.println(name + "叫");
    }
}
//这里的Dogs继承了抽象类Animals
public final class Dogs extends Animals {
    public Dogs(String name, int ages, String foods) {
        super(name, ages, foods);
    }
```

```
47      @Override
48      public void bark(){
49          System.out.println(name + "叫");
50      }
51  }
```

接口

上面说到了抽象类的定义，这里再次引入一个结构：接口

接口可以理解成一个仅含有方法(行为) 的抽象类，接口里面的所有方法都必须是抽象的，而且接口在被继承时，不使用`extends`关键字，而是`implements`关键字，例如：

Java

```
1   public interface Animals_action{
2       public void eat();
3       public void bark();
4   
5   }
6   
7   //实现:
8   public class Dogs_action implements Animals_action{
9       String name;
10      int ages;
11      String meat;
12  
13      @Override
14      public void bark(){
15          System.out.println(name + "叫");
16      }
17  
18      @Override
19      public void eat(){
20          System.out.println(name + "在吃饭");
21      }
22  }
```

抽象类与接口的区别

抽象类	接口
抽象类中可以包含非抽象方法与属性	接口中的全部方法都必须是抽象的
抽象类中可以包含变量	接口中只能包含抽象方法

抽象类可以被继承	接口只能被实现(implements)
抽象类中的变量是普通变量。	接口里定义的变量只能是公共的静态的常量
抽象类是重构的结果	接口是设计的结果
抽象类主要用来抽象类别	接口主要用来抽象功能
抽象类不能被实例化(new)	接口可以被实例化

多态

多态的基础是继承

在上面的动物园系统中，我们定义了一个类`Animals`这个类主要用于继承，

TypeScript

```
public class Animals {
    final String name;
    int ages;
    String foods;

    public void eat(){
        System.out.print("在吃饭");
    }

    public void bark(){
        System.out.println("动物叫");
    }

    public void TreatMents(){
        System.out.println("需要去医务室");
    }

    //构造函数
    public Animals(String name, int ages, String foods) {
        this.name = name;
        this.ages = ages;
        this.foods = foods;
    }
}
```

当我们在定义一个Dogs类时，我们有可能需要这样定义：

TypeScript

```
public void main() {
    Animals Dogs = new Animals("Jerry",2,"meat");
    Dogs.eat();
}
```

多态指的是：不同的子类在继承父类后分别都重写覆盖了父类的方法，即父类同一个方法，在继承的子类中表现出不同的形式，子类可以使用父类的方法，也可以使用自己的方法

例如：

Java

```
//在上述的动物园案例中，子类Tiger需要使用父类Animals的医疗(TreatMents)方法

public class Tiger extends Animals{
    public void TreatMents(){
        System.out.println("老虎需要去医务室");
    }

    //老虎吼叫
    public void TigerBellow(){
        System.out.println("老虎在吼叫");
    }
}

//调用特殊方法
//在主类中：（向上转型）
Animals tiger = new Tiger();
tiger.TreatMents();
//输出:
老虎需要去医务室

//但当我们需要对老虎的吼叫进行描述时，我们必须将其转换过来
Tiger real_tiger = (Tiger)tiger;
real_tiger.TigerBellow()

//输出:
老虎在吼叫
```

匿名内部类

在对接口的方法进行调用的时候，可以在类中直接new 一个接口类

Java

```java
public interface Human{
    public void eat();
    public void sleep();
}

//主类中:
public class main{
    new Human() {
        @Override
        public void eat() {
            System.out.println("中国人吃中国菜");
        }

        @Override
        public void sleep() {

        }
    };
}
```

权限修饰符

修饰符	当前类	同一包内	子孙类(同一包)	子孙类(不同包)	其他包
`public`	Y	Y	Y	Y	Y
`protected`	Y	Y	Y	Y/N（说明）	N
`default`	Y	Y	Y	N	N
`private`	Y	N	N	N	N

Object超类

Java中所有的类的父类都是Object

在我们写一个`ToString()`方法时，我们常会在前面发现一个`@Override`注解，这是因为`Object`类中含有一个`ToString()`方法，需要重写

恭喜！Java入门完成！

下一站------Java API！