1 类与对象

1.1 面向对象概述

面向对象是一种现在最为流行的程序设计方法，几乎现在的所有应用都以面向对象为主了，最早的面向对象的概念实际上是由IBM提出的，在70年代的Smaltalk语言之中进行了应用，后来根据面向对象的设计思路，才形成C++，而由C++产生了Java这门面向对象的编程语言。

但是在面向对象设计之前，广泛采用的是面向过程，面向过程只是针对于自己来解决问题。面向过程的操作是以程序的基本功能实现为主，实现之后就完成了，也不考虑修改的可能性，面向对象，更多的是要进行子模块化的设计，每一个模块都需要单独存在，并且可以被重复利用，所以，面向对象的开发更像是一个具备标准的开发模式。

在面向对象定义之中，也规定了一些基本的特征：

1. 封装：保护内部的操作不被破坏；
2. 继承：在原本的基础之上继续进行扩充；
3. 多态：在一个指定的范围之内进行概念的转换。

对于面向对象的开发来讲也分为三个过程：OOA（面向对象分析）、OOD（面向对象设计）、OOP（面向对象编程）。

1.2 类与对象的基本概念

类与对象时整个面向对象中最基础的组成单元。

- 类：是抽象的概念集合，表示的是一个共性的产物，类之中定义的是属性和行为（方法）；
- 对象：对象是一种个性的表示，表示一个独立的个体，每个对象拥有自己独立的属性，依靠属性来区分不同对象。

可以一句话来总结出类和对象的区别：**类是对象的抽象，对象是类的实例。**类只有通过对象才可以使用，而在开发之中应该先产生类，之后再产生对象。类不能直接使用，对象是可以直接使用的。

1.3 类与对象的定义和使用

在Java中定义类，使用关键字class完成。语法如下：

```
class 类名称
{
    属性（变量）；
    行为（方法）；
    构造函数;
}
```

类中包含3要素：

1. 属性：区分个体对象差异性；
2. 方法：解决重复性代码问题；
3. 构造函数：初始化属性。

类名的命名规则：

1. 只能是字母，数字，下划线和美元符号（$）；
2. 首字母只能是字母，下划线和美元符号，不能是数字；
3. 不能是关键字；
4. 类名严格区分大小写；
5. 可以但不要叫中文。

注意区分.java文件与类的区别：

在一个.java文件中可以包含多个类，但是只有一个类能被public（公共的）修饰，且被public修饰的类名必须与.class文件名一模一样。

```
public class Test01_类
{

}
```

类定义完成之后，肯定无法直接使用。如果要使用，必须依靠对象，那么由于类属于引用数据类型，所以对象的产生格式（两种格式）如下：（1）格式一：声明并实例化对象

```
类名称 对象名称 = new 类名称 ();
```

（2）格式二：先声明对象，然后实例化对象：

```
类名称 对象名称 = null ;
对象名称 = new 类名称 ();
```

引用数据类型与基本数据类型最大的不同在于：引用数据类型需要内存的分配和使用。所以，关键字new的主要功能就是分配内存空间，也就是说，只要使用引用数据类型，就要使用关键字new来分配内存空间。

当一个实例化对象产生之后，可以按照如下的方式进行类的操作：

- 对象.属性：表示调用类之中的属性；
- 对象.方法()：表示调用类之中的方法。

两种内存空间的概念：

1. 堆内存：保存对象的属性内容。堆内存需要用new关键字来分配空间；
2. 栈内存：保存的是堆内存的地址（在这里为了分析方便，可以简单理解为栈内存保存的是对象的名字）。

1.4 方法的定义和使用

方法（Method）的定义

```
[修饰符] 返回值类型 方法名([参数类型 参数1 ,参数类型 参数2]) [throws 异常]
{
    // 方法体；
    [return 返回值类型；]

}
```

说明:

- 修饰符：访问控制符public|protected|private、静态修饰符static、抽象修饰符abstract、最终修饰符final等。
- 返回值类型:void(无返回值的方法) 或 数据类型（有返回值的方法）。
- 参数列表：多个参数时逗号分割也可以没有参数。

1.5 构造函数

构造器（Constructor）：是一个用来创建对象的特殊方法，用来初始化对象的属性。

- 构造器的名字与类名相同构造器。
- 没有返回值。
- 构造器所包含的语句用来对所创建的对象进行初始化。
- 没有参数的构造器称为"无参构造器"每个Java类都至少有一个构造器，如果该类没有显式地声明任何构造器，系统会默认地为该类提供一个不包含任何语句的无参构造器。

```
public class Point
{
    public int x = 0;
    public int y = 0;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public Point()
    {

    }
}
```

2 封装

封装（Encapsulation）是面向对象方法的重要原则，就是把对象的属性和操作（或服务）结合为一个独立的整体，并尽可能隐藏对象的内部实现细节。

- 将类的某些信息隐藏在类的内部，不允许外部程序进行直接的访问调用。
- 通过该类提供的方法来实现对隐藏信息的操作和访问。
- 隐藏对象的信息。
- 留出访问的对外接口。

2.1 封装的特点

- 对成员变量实行更准确的控制。
- 封装可以隐藏内部程序实现的细节。
- 良好的封装能够减少代码之间的耦合度。
- 外部成员无法修改已封装好的程序代码。

- 方便数据检查，有利于保护对象信息的完整性，同时也提高程序的安全性。
- 便于修改，体高代码的可维护性。

2.2 封装的使用

- 使用private修饰符，表示最小的访问权限。
- 对成员变量的访问，统一提供setXXX，getXXX方法。

3 继承

继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。当然，如果在父类中拥有私有属性(private修饰)，则子类是不能被继承的。

3.1 继承的特点

1、注意事项

- 只支持单继承，即一个子类只允许有一个父类，但是可以实现多级继承，及子类拥有唯一的父类，而父类还可以再继承。
- 子类可以拥有父类的属性和方法。
- 子类可以拥有自己的属性和方法。
- 子类可以重写覆盖父类的方法。

2、特点

- 提高代码复用性。
- 父类的属性方法可以用于子类。
- 可以轻松的定义子类。
- 使设计应用程序变得简单。

3.2 继承的使用

1、在父子类关系继承中，如果成员变量重名，则创建子类对象时，访问有两种方式。

a、直接通过子类对象访问成员变量

等号左边是谁，就优先使用谁，如果没有就向上找。

b、间接通过成员方法访问成员变量

该方法属于谁，谁就优先使用，如果没有就向上找。

2、成员方法

成员方法也是一样的，创建的对象是谁，就优先使用谁，如果没有则直接向上找。

注意事项： 无论是成员变量还是成员方法，如果没有都是向上父类中查找，绝对不会向下查找子类的。

3、在继承关系中，关于成员变量的使用：

- 局部成员变量：直接使用
- 本类成员变量：this.成员变量
- 父类成员变量：super.父类成员变量

3.3 重写、重载

重写（Override）是子类对父类的允许访问的方法的实现过程进行重新编写，返回值和形参都不能改变。

重写的规则：

1. 参数列表必须与被重写方法相同。
2. 访问权限不能比父类中被重写的方法的访问权限更低（public>protected>(default)>private）。
3. 父类成员的方法只能被它的子类重写。
4. 被final修饰的方法不能被重写。
5. 构造方法不能重写。

重载（Overload）是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。 最常用的地方就是构造器的重载。

重载的规则：

1. 被重载的方法必须改变参数列表（参数个数或者类型不一样）。
2. 被重载的方法可以改变返回类型。
3. 被重载的方法可以改变访问修饰符。

3.4 this、super关键字

super()关键字的用法 ：

1. 子类的成员方法中，访问父类的成员变量。
2. 子类的成员方法中，访问父类的成员方法。
3. 子类的构造方法中，访问父类的构造方法。

this关键字用法：

1. 本类成员方法中，访问本类的成员变量。
2. 本类成员方法中，访问本类的另一个成员方法。
3. 本类的构造方法中，访问本类的另一个构造方法。

注意：

- this关键字同super一样，必须在构造方法的第一个语句，且是唯一的。
- this()与super()不能同时存在。

3.5 继承中的构造器

继承关系中，父子类构造方法的访问特点：

1. 在子类构造方法中有一个默认隐含的 super(); 调用，因此一定是先调用父类构造方法，再调用子类构造方法。
2. 子类构造可以通过 super(); 调用父类的重载构造。(重载)
3. super(); 的父类调用构造方法，必须在子类构造中的第一行，就是第一个;号结束的元素，并且只能调用一次。

3.6 继承的注意事项

1. Java语言是单继承的，一个子类只能有唯一一个父类 。

2. Java语言可以是多级继承，一个子类有一个父类，一个父类还可以有一个父类。
3. 一个子类只有一个父类，但是一个父类可以有多个子类。
4. 一个子类不可以有多个父类。

3.7 Object类

Java Object 类是所有类的父类，也就是说 Java 的所有类都继承了 Object，子类可以使用 Object 的所有方法。

Object 类位于 java.lang 包中，编译时会自动导入，我们创建一个类时，如果没有明确继承一个父类，那么它就会自动继承 Object，成为 Object 的子类。

Object 类可以显示继承，也可以隐式继承，以下两种方式时一样的：

显示继承：

```
public class Student extends Object
{

}
```

隐式继承：

```
public class Student
{

}
```

4 多态

多态是同一个行为具有多个不同表现形式或形态的能力。

4.1 多态的特点

1. 消除类型之间的耦合关系，实现低耦合。
2. 灵活性。
3. 可扩充性。
4. 可替换性。

4.2 多态的体现形式

- 继承
- 父类引用指向子类的对象
- 重写

注意：在多态中，编译看左边，运行看右边

4.3 向上转型

格式：父类名称 对象名 = new 子类名称();

```
person p1 = new chinese();
```

含义：右侧创建一个子类对象，把它当作父类来使用。

注意：向上转型一定是安全的。

缺点：一旦向上转型，子类中原本特有的方法就不能再被调用了。

补充：可以使用`instanceof`来判断对象是否是某一个类或类的子类。

```
ying y = new kafe();
if (y instanceof kafe) {
    //如果变量y是kafe类或其子类，向下转型
    kafe k = (kafe) y;
}
```

4.4 向下转型

格式：子类名称 对象名 = (子类名称) 父类对象;

```
animal p1 = new cat();
cat c1 = (cat) p1;
```

含义：将父类对象，还原成为本来的子类对象。

注意：必须保证对象本来创建的时候，就是猫，才能向下转型成为猫。

如果对象本来创建的时候本来就不是猫，现在非要向下转型成为猫，就会报错。（`ClassCastException`）

4.5 包

包是一种松散的类的集合，通常把需要在一起工作的类(互相访问)放入一个包。

在Java语言程序设计中，通常需要定义许多类；就像利用 “文件夹”把许多文件组织在一起，使硬盘管理的文件更清晰、更有条理一样；Java利用“包”把 一些需要在一起操作的类组织在一起，以便程序员更好地管理操作这些类。

引入的原因： 容易找到和使用类避免名称冲突

控制访问定义：包是一个相关的类和接口的集合，它可以提供访问保护和名称空间管理。

例如：基本的类在 java.lang中；用于输入和输出的类在java.io中。

声明包在源文件的开始加上：

package 包名(要求全部小写)(反域名制) package com.sun ;

使用包中的类如使用不同包的类，必须加入 import 关键字。

```
import java.util.Scanner;
```

4.6 访问权限修饰符

作用域	类	同一package	子孙	其他package
public	√	√	√	√
protected	√	√	√	×
friendly	√	√	×	×
private	√	×	×	×

不写时默认为friendly。

5 Static关键字

5.1 修饰成员变量

在我们平时的使用当中，static最常用的功能就是修饰类的属性和方法，让他们成为类的成员属性和方法，我们通常将用static修饰的成员称为类成员或者静态成员。

```
class Person
{
    String name;
    static int age; //属性属于类，而不属于对象
}
```

推荐使用"类名.属性"调用，而不是使用"对象.属性"。

```
//不推荐
System.out.println(p.age);   //对象.属性
//推荐
System.out.println(Person.age);  //类名.属性  表达了属性属于类
```

5.2 修饰成员方法

static的另一个作用，就是修饰成员方法。相比于修饰成员属性，修饰成员方法对于数据的存储上面并没有多大的变化，因为我们从上面可以看出，方法本来就是存放在类的定义当中的。

static修饰成员方法最大的作用，就是可以使用"类名.方法名"的方式操作方法，避免了先要new出对象的繁琐和资源消耗，我们可能会经常在帮助类中看到它的使用：

```
public class PrintHelper
{
    //静态方法中不能直接使用非静态属性
    public static void print(Object o)
    {
        System.out.println(o);
    }
}
```

static使得修饰的方法成为类的方法，使用时通过“类名.方法名”的方式就可以方便的使用了，相当于定义了一个全局的函数（只要导入该类所在的包即可）。

不过它也有使用的局限，一个static修饰的类中，不能使用非static修饰的成员变量和方法。

5.3 静态块

```
static Book b1;
static Book b2;
static
{
    System.out.println("静态代码块1执行了");
    b1 =  new Book("static book1 创建了");
    b2 =  new Book("static book2 创建了");
}
```

5.4 静态导包

```
/* PrintHelper.java文件 */
package com.soft863.test;

public class PrintHelper
{
    public static void print(Object o)
    {
        System.out.println(o);
    }
}
```

```
/* Main.java文件 */
import static com.soft863.test.PrintHelper.*;  //静态导包

public class Main
{
    public static void main( String[] args )
    {
        print("Hello World!");
    }

}
```

在App.java文件中，我们首先将`PrintHelper`类导入，这里在导入时，我们使用了static关键字，而且在引入类的最后还加上 了“.*”，它的作用就是将`PrintHelper`类中的所有类方法直接导入。

不同于非static导入，采用static导入包后，在不与当前类的方法名冲突的情况下，无需使用“类名.方法名”的方法去调用类方法了，直接可以采用"方法名"去调用类 方法，就好像是该类自己的方法一样使用即可。

5.5 总结

static是java中非常重要的一个关键字，而且它的用法也很丰富，主要有四种用法：

1. 用来修饰成员变量，将其变为类的成员，从而实现所有对象对于该成员的共享；
2. 用来修饰成员方法，将其变为类方法，可以直接使用“类名.方法名”的方式调用，常用于工具类；
3. 静态块用法，将多个类成员放在一起初始化，使得程序更加规整，其中理解对象的初始化过程非常关键；
4. 静态导包用法，将类的方法直接导入到当前类中，从而直接使用“方法名”即可调用类方法，更加方便。

6 Final关键字

6.1 修饰数据

在编写程序时，我们经常需要说明一个数据是不可变的，我们称为常量。在java中，用final关键字修饰的变量，只能进行一次赋值操作，并且在生存期内不可以改变它的值。更重要的是，final会告诉编译器，这个数据是不会修改的，那么编译器就可能会在编译时期就对该数据进行替换甚至执行计算，这样可以对我们的程序起到一点优化。不过在针对基本类型和引用类型时，final关键字的效果存在细微差别。

```
class Teacher
{
    //final在声明时必须赋值
    final String name = "hello";
}
```

```
public class final修饰引用数据类型 {
    public static void main(String[] args) {
        Teacher t = new Teacher();
        t.s.name = "李四";
        //引用数据类型不能在改变了，但是这个对象中的属性还可以修改
        // t.s = new Student();
    }
}

class Teacher {
    final Student2 s = new Student2();
}

class Student {
    String name="张三";
}
```

final修饰的值是不可改变的，我们知道引用变量的值实际上是它所引用的对象的地址，也就是说该地址的值是不可改变的，从而说明了为什么引用变量不可以改变引用对象。而实际引用的对象实际上是不受final关键字的影响的，所以它的值是可以改变的。

注意：同时使用static和final修饰的成员在内存中只占据一段不能改变的存储空间。

6.2 修饰方法参数

我们编写方法时，可以在参数前面添加final关键字，它表示在整个方法中，我们不会（实际上是不能）改变参数的值。

```
class Teacher
{
    public void hello(final String name) {
    //  name 参数不能改变
        System.out.println(name);
    }
}
```

6.3 修饰方法

用final关键字修饰方法，它表示该方法不能被覆盖。这种使用方式主要是从设计的角度考虑，即明 确告诉其他可能会继承该类的程序员，不希望他们去覆盖这个方法。这种方式我们很容易理解，然而，关于private 和final关键字还有一点联系，这就是类中所有的private方法都隐式地指定为是final的，由于无法在类外使用 private方法，所以也就无法覆盖它。

```
class Teacher {
    //被final修饰的方法不能被重写
    public final void say() {
        System.out.println("say hello");
    }
}
```

6.4 修饰类

使用final关键字修饰类的作用，就是用final修饰的类是无法被继承的。

```
//final 修饰类表示，表示该类不能被继承
final class Teacher {

}
```

6.5 总结

final关键字是我们经常使用的关键字之一，它的用法有很多，但是并不是每一种用法都值得我们去广泛使用。它的主要用法有以下四种：

1. 用来修饰数据，包括成员变量和局部变量，该变量只能被赋值一次且它的值无法被改变。对于成员变量来讲，我们必须在声明时或者构造方法中对它赋值；
2. 用来修饰方法参数，表示在变量的生存期中它的值不能被改变；
3. 修饰方法，表示该方法无法被重写；
4. 修饰类，表示该类无法被继承。

7 抽象类与接口

7.1 抽象类与抽象方法

抽象方法是一种特殊的方法：它只有声明，而没有具体的实现。抽象方法的声明格式为：

```
public abstract void open();
```

抽象方法必须使用abstract关键字进行修饰。如果一个类含有抽象方法，则称这个类为抽象类，抽象类必须在类前用abstract关键字修饰。因为抽象类中无具体实现的方法，所以不能用抽象类创建对象。

```
public abstract class Door
{
    public abstract void open();
    public abstract void close();
}
```

包含抽象方法的类称为抽象类，但并不意味着抽象类中只能有抽象方法，它和普通类一样，同样可以拥有成员变量和普通的成员方法。注意，抽象类和普通类的主要有三点区别：

1）抽象方法必须为public或者protected（因为如果为private，则不能被子类继承，子类便无法实现该方法），缺省情况下默认为public。

2）抽象类不能用来创建对象；

3）如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。如果子类没有实现父类的抽象方法，则必须将子类也定义为为abstract类。

其他方面，抽象类和普通的类并没有区别。

7.2 接口

接口（interface），在软件工程中，接口泛指供别人调用的方法或者函数。从这里，我们可以体会到Java语言设计者的初衷，它是对行为的抽象。在Java中，定一个接口的形式如下：

```
public interface Door
{
    public void open();
    public void close();
}
```

接口中可以含有变量和方法。但是要注意，接口中的变量会被隐式地指定为`public static final`变量（并且只能是`public static final`变量，用private修饰会报编译错误），而方法会被隐式地指定为`public abstract`方法且只能是`public abstract`方法（用其他关键字，比如private、protected、static、 final等修饰会报编译错误），并且接口中所有的方法不能有具体的实现，也就是说，接口中的方法必须都是抽象方法。

从这里可以隐约看出接口和抽象类的区别，接口是一种极度抽象的类型，它比抽象类更加“抽象”，并且一般情况下不在接口中定义变量。

要让一个类遵循某组特地的接口需要使用`implements`关键字，具体格式如下：

```
//              实现
class MuDoor implements Door {

    @Override
    public void open() {
        System.out.println("开门...");
    }

    @Override
    public void close() {
        System.out.println("关门...");
    }
}
```

```
class teacher extends father implements teach, play {

    //优先继承，子类自动有了teach方法，就不在需要重写了
//    @Override
//    public void teach()
//    {
//        System.out.println("边教边玩");
//    }
}

interface teach {
    void teach();
}

interface play {
    void teach();
}

class father {
    public void teach() {
        System.out.println("父亲玩");
    }
}
```

允许一个类实现多个特定的接口。如果一个非抽象类实现了某个接口，就必须实现该接口中的所有方法。对于实现某个接口的抽象类，可以不实现该接口中的抽象方法。

在`JDK8`之后：

1. default修饰，接口里允许定义默认的方法，但默认方法也可以覆盖重写。
2. 接口里允许定义静态方法。

接口的注意事项：不能通过接口的实现类对象去调用接口中的静态方法。

正确语法：接口名称调用静态方法。

```
interface door {
    public static final String name = "夜";
    String sex = "男";  //隐式的 public static final

    //静态方法
    public static void info() {
        System.out.println("hello");
    }

    public abstract void open();

    void close();  //隐式的 public abstract
}
```

接口当中的常量的使用：

1. 接口当中定义的常量：可以省略`public static final`。
2. 接口当中定义的常量：必须进行赋值。
3. 接口当中定义的常量：常量的名称要全部大写，多个名称之间使用下划线进行分割。

使用接口的注意事项：

1. 接口是没有静态代码块或者构造方法。
2. 一个类的直接父类是唯一的，但是一个类可以同时实现多个接口。
3. 如果实现类没有覆盖重写接口中所有的抽象方法，那么实现类就必须是一个抽象类 。
4. 如果实现类中实现多个接 口，存在重复的抽象方法，那么只需要覆盖重写一次即可。
5. 在Java中，如果实现类的直接继承父类与实现接口 发生冲突时，父类优先级高于接口。

接口之间的关系：

1. 多个接口之间是继承关系。
2. 多个父接口当中默认方法如果重复，那么子接口必须进行默认方法的覆盖重写。

7.3 接口与抽象类的区别

语法层面上的区别：

1. 抽象类可以提供成员方法的实现细节，而接口中只能存在`public abstract`方法；
2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是`public static final`类型的；
3. 接口中不能含有静态代码块，而抽象类可以有静态代码块；
4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

设计层面上的区别：

1. 抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

2. 设计层面不同，抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。

8 枚举

8.1 枚举类型

枚举类型是Java 5中新增特性的一部分，它是一种特殊的数据类型，之所以特殊是因为它既是一种 类(class)类型却又比类类型多了些特殊的约束，但是这些约束的存在也造就了枚举类型的简洁性、安全性以及便捷性。

```
public class DayDemo
{
    public static final int MONDAY =1;
    public static final int TUESDAY=2;
    public static final int WEDNESDAY=3;
    public static final int THURSDAY=4;
    public static final int FRIDAY=5;
    public static final int SATURDAY=6;
    public static final int SUNDAY=7;
}
```

上述的常量定义常量的方式称为 `int` 枚举模式，这样的定义方式并没有什么错，但它存在许多不 足，如在类型安全和使用方便性上并没有多少好处，如果存在定义 `int` 值相同的变量，混淆的几率还是很大的，编译器也不会提出任何警告，因此这种方式在枚举出现后并不提倡。

现在我们利用枚举类型来重新定义上述的常量，同时也感受一把枚举定义的方式，如下定义周一到周日的常量 。

```
//枚举类型，使用关键字enum
enum Day
{
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

在定义枚举类型时我们使用的关键字是 `enum` ，与 class 关键字类似，只不过前者是定义枚举类型，后者是定义类类型。枚举类型Day中分别定义了从周一到周日的值，这里要注意，值一般是大写的字母，多个值之间以逗号分隔。

枚举类型可以像类(class)类型一样，定义为一个单独的文件，当然也可以定义在其他类内部。

```
public class EnumDemo
{
    public static void main(String[] args)
    {
        //直接引用
        Day day = Day.MONDAY;
    }
}
```

8.2 枚举实现的原理

实际上在使用关键字 `enum` 创建枚举类型并编译后，编译器会为我们生成一个相关的类，这个类继承了 `Java API` 中的 `java.lang.Enum` 类，也就是说通过关键字 `enum` 创建枚举类型在编译后事实上也是一 个类类型而且该类继承自 `java.lang.Enum` 类。

8.3 `Values` 方法与 `ValueOf` 方法

`values()` 方法和 `valueOf(String name)` 方法是编译器生成的 static 方法。

因此从前面的分析中，在 `Enum` 类中并没出现 `values()` 方法，但 `valueOf()` 方法还是有出现的，只不过编译器生成的 `valueOf()` 方法需传递一个 `name` 参数，而 `Enum` 自带的静态方法 `valueOf()` 则需要传递两个方法，从前面反编译后的代码可以看出，编译器生成的 `valueOf` 方法最终还是调用了 `Enum` 类的 `valueOf` 方法。

```
//获取所有枚举的值
day[] days = day.values();
System.out.println(Arrays.toString(days));

//通过值来获取枚举
day d1 = day.valueOf("MONDAY");
System.out.println(d1);
```

`values()` 方法的作用就是获取枚举类中的所有变量，并作为数组返回，而 `valueOf(String name)` 方法与 `Enum` 类中的 `valueOf` 方法的作用类似根据名称获取枚举变量，只不过编译器生成的 `valueOf` 方法更 简洁些只需传递一个参数。

由于 `values()` 方法是由编译器插入到枚举类中的 static 方法，所以如果我们将枚举实例向上转型为 `Enum` ，那么 `values()` 方法将无法被调用，因为 `Enum` 类中并没有 `values()` 方法， `valueOf()` 方法也是同样的道理，注意是一个参数的。

```
//正常使用
Day[] ds = Day.values();
//向上转型Enum
Enum e = Day.MONDAY;
//无法调用,没有此方法
//e.values();
```

8.4 向 `enum` 类添加方法与自定义构造函数

```
public class 枚举
{
    public static final String MONDAY = "星期一";
    public static void main(String[] args) {
        //通过枚举获取值
        System.out.println(day.MONDAY);
        System.out.println(day.SUNDAY);


        //获取所有枚举的值
```

```
        day[] days = day.values();
        System.out.println(Arrays.toString(days));

        //通过值来获取枚举
        day d1 = day.valueOf("MONDAY");
        System.out.println(d1);
    }
}

enum day
{
    //day MONDAY = new day();
    MONDAY("星期一"),   //public static final String MONDAY = "星期一";
    SUNDAY("星期日"),
    TUESDAY;  //public static final String TUESDAY = "TUESDAY";

    day()  //防止 TUESDAY 报错，有有参构造器
    {

    }

    private String name;  //属性

    day(String name)
    {
        this.name = name;
    }

    @Override
    public String toString()
    {
        return this.name;
    }
}
```

在 `enum` 类中确实可以像定义常规类一样声明变量或者成员方法。但是我们必须注意到，如果打 算在 `enum` 类中定义方法，务必在声明完枚举实例后使用分号分开，倘若在枚举实例前定义任何方法，编译器都将会报错，无法编译通过，同时即使自定义了构造函数且 `enum` 的定义结束，我们也永远无法手动调用构造函数创建枚举实例，毕竟这事只能由编译器执行。

8.5 枚举与switch

使用switch进行条件判断时，条件参数一般只能是 `byte short int char String enum`。

```
public class 枚举与switch
{
    public static void main(String[] args)
    {
        switch (color.RED)
        {
            case RED:
                System.out.println("红色");
```

```
                break;
            case BLUE:
                System.out.println("蓝色");
                break;
            case GREEN:
                System.out.println("绿色");
                break;
        }
    }
}

enum color
{
    RED,GREEN,BLUE;
}
```

9 内部类

9.1 内部类

```
public class 内部类
{
    public static void main(String[] args)
    {
       //先去创建外部类
       Father f = new Father();
       //再去创建内部类
       Father.Money m = f.new Money();
       m.cost();
    }
}

class Father
{
    String name;
    public void playGame()
    {
        System.out.println("父亲的兴趣爱好");
    }

    //内部类
    class Money
    {
        double money = 10000;
        //花费
        public void cost()
        {
            //内部类中可以使用外部类的属性
            System.out.println(name+"的小金库");

            playGame();
```

```
            System.out.println("花费了：￥"+100);
            System.out.println("还剩：￥"+this.money);
        }
    }
}
```

9.2 静态内部类

```
public class 静态内部类
{
    public static void main(String[] args)
    {
        Fater.Money m = new Fater.Money();
        m.cost();

        Fater.Money.hello();  //静态内部类的静态方法
    }
}
class Fater
{
    String name;

    public static  void playGame()
    {
        System.out.println("兴趣爱好");
    }

    //静态内部类
    static class Money
    {
        public void cost()
        {
            //静态的只能调用静态的，不能调用非静态
            //System.out.println(name);
            playGame();
        }

        public static void hello()
        {
            System.out.println("hello");
        }
    }
}
```

9.3 局部内部类

```
class Father
{
    String name;
    public void playGame()
```

```
    {
        //局部内部类
        class Money
        {
            public void cost()
            {
                System.out.println("花钱");
            }
        }

        //只能在该方法中使用，且在类的后面
        Money m = new Money();
        m.cost();
    }
}
```

9.4 匿名内部类

```
public class 匿名内部类
{
    public static void main(String[] args)
    {
        game g = new game()  //匿名内部类，只在生存周期有效
        {
            @Override
            public void hello()
            {
                System.out.println("匿名内部对大家说你好");
            }
        };

        g.hello();
    }
}

interface game
{
    void hello();
}
```