《复习一》

笔记本: JAVA复习、作业本
创建时间: 2018/8/5 9:21 **更新时间:** 2018/8/18 16:19
作者: Debao Xu

匿名对象的使用

```
public class Demo1
{
	public static void main(String[] args)
	{
		System.out.println(new Temp1());
	}
}
```

封装

```
//对变量age进行封装

class Temp1
{
	private int age;  // 关键字private不要忘
	String Name;
	public void setAge(int age)
	{
		if((age>120)||(age<0))  // 注意使用“或操作”
			System.out.println("!");
		else
			this.age = age;
	}
	public int getAge()
	{
		return age;
	}
	void study()
	{
		System.out.println(age);
	}
}
public class Demo1
{
	public static void main(String[] args)
	{
		Temp1 t = new Temp1();
		t.setAge(250);
		System.out.println(t.getAge());
	}
}
```

构造函数（作用：给对象进行初始化用的）

```
Temp1(int age,String Name)
{
     this.age = age;
     this.Name = Name;
}

// 注意构造函数的书写规则（函数名、无返回值）
```

1、构造函数可以调用一般函数；
2、一般函数不可以调用构造函数（因为构造函数先于一般函数存在）；
3、构造函数一旦被私有化，其他程序就无法创建该构造函数对应的对象；
4、构造函数与构造函数之间也是可以相互调用的，具体调用的方法是通过this关键字来解决，通过this带上参数列表的形式就可以调用其他构造函数，并且用于调用其他构造函数的this语句必须放在构造函数的第一行，因为初始化的动作要先执行。

```
// 构造函数之间相互调用
Temp1(int age)
{
	this.age = age;
}
Temp1(int age,String Name)
{
	this(28);  //this关键字调用构造函数，并且在第一行
	this.Name = Name;
}
```

当成员变量和局部变量同名时，可以通过this关键字来区分。

那么，某个函数什么时候需要用静态关键字static修饰呢?
答：如果该函数没有访问过对象中的属性时（如上面输出时只是打印了“huhu”），就需要用静态修饰。

注意事项：
1、静态方法只能访问静态成员，不能访问非静态成员。这就是静态方法的访问局限性，而非静态方法可以访问静态成员。
2、静态的方法中不能出现this或者 super关键字，原因是：静态的数据先在，对象后在，如果静态中有this的话，那么这个this就要去找对应的对象，而此时对象还没有被创建，这就矛盾了。
3、主函数是静态的。

静态代码块是给“类”初始化的（只执行一次）；构造代码块是给“所有的对象”初始化的；构造函数是给“对应的对象”初始化的；局部代码块可以控制局部变量的生命周期

```
class Temp1
{
	static //静态代码块
	{
		System.out.println("静态代码块给类进行初始化，并完成一些初始化的工作，且只执行一次");
	}
}
```

```
class Temp1
{
	private int age;
	String Name;

	//构造代码块，给所有的对象初始化
	{
		System.out.println("Cry!");
	}

	Temp1(int age,String Name)
	{
		this.age = age;
		this.Name = Name;
	}
	public int getAge()
	{
		return age;
	}
}
```

```
public class Demo1
{
	public static void main(String[] args)
	{
		Temp1 t1 = new Temp1(0,"Rex");
		Temp1 t2 = new Temp1(26,"Debao");
		System.out.println(t1.getAge());
		System.out.println(t2.getAge());
	}
}
```

运行结果：

```
Cry!  // 第一个对象初始化的结果
Cry!  // 第二个对象初始化的结果
0
26
```

创建一个对象的流程（例如下面创建的对象p，及对应的内存图）：

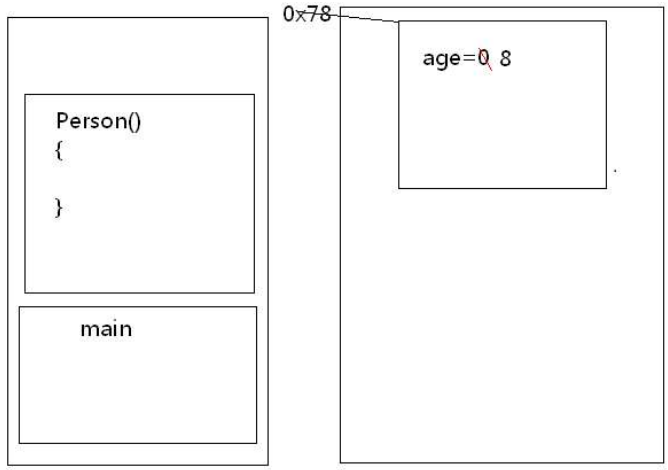
1、加载指定的字节码文件（.class文件）进内存，即创建对象要从编译后的文件开始
2、通过new在堆内存中开辟空间，分配首地址值（0x78）
3、对对象中的属性进行默认初始化（age = 0）
4、调用与之对应的构造函数，构造函数压栈（Person()压栈）
5、构造函数中执行隐式的语句super() 访问父类中的构造函数
6、对属性进行显示初始化（age = 8）。
7、调用类中的构造代码块（{ Sys.out.println("constructor code run。。。"+age); }）
8、执行构造函数中自定义的初始化代码（Person()）
9、初始化完毕，将地址赋给指定的引用（此处赋给引用 p）

```
创建一个对象p，  Person p = new Person();
class Person
{
	private int age = 8;//显示初始化。

	{//构造代码块。给所有对象进行初始化的。
	  System.out.println("constructor code run。。。"+age);

	}
	Person()
	{
	  1,super();//调用父类构造函数。没学。
	  2,显示初始化。
	  3,构造代码块初始化。
	  System.out.println("person run");
	}
	Person(int age)
	{
	  this.age = age;
	  System.out.println("Perosn(age) run");
	}

}
```



单例设计模式

主要是为了解决“对象唯一性”的问题。例如程序A要在程序B的基础之上进一步进行操作，就要求这两个文件对象是同一个。

```
class Person
{
	private int age;
	private String name;
	private Person(int age,String name)
	{
		this.age = age;
		this.name = name;
```

```
        }
        static Person p = new Person(26,"Rex"); //此处创建的对象p，必须是static的，因为下面调用p的是静态方法（即静态方法只能调用静态成员）
        static Person getInstance() // 此处为了主函数可以进行“类名调用”，必须是static的，返回对象使用关键字 getInstance()
        {
                return p;
        }
        public void setAge(int age)
        {
                this.age = age;
        }
        public int getAge()
        {
                return age;
        }
}
public class Demo2
{
        public static void main(String[] args)
        {
                Person s1 = Person.getInstance();
                System.out.println(s1.getAge());
                s1.setAge(45);
                System.out.println(s1.getAge());
        }
}
运行结果：//此处的运行结果就是同一个对象了
26
45
```

继承

在本类中要是出现了局部变量和成员变量同名，则通过this关键字来区分；在子父类中要是出现了变量同名，则通过super关键字来区分

子父类出现之后，代码上会发生一些变化，主要体现在 成员变量、成员方法、构造函数 这3个方面。
在成员变量上，如果子父类中有相同的变量名，则可以通过super关键字来访问父类中的成员变量；
在成员方法上，如果子父类中有相同的成员函数（返回值类型，函数名，参数列表都一样），则可以通过”覆盖“来在子类中重新定义；
在构造函数上，在执行子类中的构造函数时，会默认先执行父类中的构造函数（因为在子类构造函数中，默认第一行就是用super关键字来调用父类中的构造函数）

通过前面的学习，我们已经知道，在”本类“中相互调用构造函数时要用this关键字的语句，而在子类中调用父类中的构造函数时，需要使用super关键字语句

【注意】：当父类中没有空参数构造函数时，子类需要通过显示定义super语句指定要访问的父类中的构造函数
【注意】：用来调用父类构造函数的super语句在子类构造函数中必须定义在第一行，因为父类的初始化要先完成

```
class Fu
{
        int num = 4;
        private int age;
        private String name;
        Fu(int age,String name)
        {
                this.age = age;
                this.name = name;
        }
        public void setAge(int age)
        {
                this.age = age;
        }
```

```
        public int getAge()
        {
                return age;
        }
        public void setName(String name)
        {
                this.name = name;
        }
        public String getName()
        {
                return name;
        }
    void show(String name)
        {
                System.out.println(name+" Cry!");
        }
}
class Zi extends Fu
{
        int num = 6;

        Zi()
        {
                super(23,"Rex");//父类中没有空构造函数，那么在子类的构造函数中就要用关键字super显式定义，且必须放在第一行
        }
        void show()
        {
                super.show("Rex"); //通过关键字super也可以调用父类中的函数
                int num = 5;
                System.out.println("Debao");
                System.out.println(num);  //num为本类中的局部变量
                System.out.println(this.num);  //num为本类中的成员变量
                System.out.println(super.num);  //num为父类中的成员变量
        }
}
public class Demo3
{
        public static void main(String[] args)
        {
                Zi z = new Zi();
                z.show();
        }
}
```

父类中没有空构造函数时，由super关键字定义的构造函数必须**放在子类的构造函数中**

关键字final可以将数据常量化；final可以修饰类、方法、变量、常量；final修饰过的类不可以被继承，修饰过的方法不可以被覆盖，修饰过数据会变成常量且只能定义一次（即一个变量经过final修饰后，就会被”固定“）。

```
 public static final int AGE = 27; //AGE为全局常量，public表示对外可以被访问；static表示不需要创建对象，直接用类名就可以访问；final表示将其常量化

public static final int AGE = 30;  //这就不可以了，因为final修饰的变量只能被定义一次，上面已经定义过一次了。
```

抽象类

```
abstract class employee
{
```

```
        private int age;
        private String name;
        employee(int age,String name) // 抽象类中的构造函数用于给子类的对象实例化
        {
                this.age = age;
                this.name = name;
        }
        abstract void work(); //抽象类中的抽象方法要用abstract修饰，并且没有函数体
        abstract void eat();
        void walk()  // 抽象类中的非抽象方法，不需要用abstract修饰但是要写出函数体
        {

        }
}
class programmer extends employee
{
        programmer(int age,String name)
        {
                super(age,name); //调用父类的构造函数
        }
        void work() //必须将抽象类中的所有的抽象方法，都进行覆盖
        {
                System.out.println("Coding!");
        }
        void eat() //必须将抽象类中的所有的抽象方法，都进行覆盖
        {

        }
}
class manager extends employee
{
        private double bonus; // 注意，要先定义类型
        manager(int age,String name,double bonus)
        {
                super(age,name);
                this.bonus = bonus; // 属于该类的特有属性，并且是本类中的变量，所以用this关键字来修饰
        }
        void work()
        {
                System.out.println("Management!");
        }
        void eat()
        {

        }
}
public class Demo4
{
        public static void main(String[] args)
        {
                programmer p = new programmer(26,"Rex");
                manager m = new manager(23,"Debao",3000);
                p.work();
                p.walk(); //抽象类中的非抽象方法可以直接调用，不需要在子类中进行覆盖
                m.work();
        }
}
```

接口

接口中的成员和抽象类中成员的定义的不同之处在于;

1、接口中常见的成员有两种：全局常量 、抽象方法；抽象类中一般不含全局常量 2、而且在接口中定义的数据**都有**固定的修饰符public；抽象类中的方法**不一定要**public修饰

接口的特点：
1、接口不可以实例化，因为其中包含了抽象类方法；这一点类似于抽象类
2、需要覆盖了接口中的所有的抽象方法的子类，才可以实例化，否则，该子类还是一个抽象类；这点类似于抽象类
3、接口是用来被实现的；对比之前抽象类的操作，是用一个类继承抽象类并覆盖抽象类中的抽象方法，从而达到实例化的效果的

接口的好处（接口产生的原因）：通过接口，解决了Java中多继承时（此处是指Java不支持类的多继承）出现的调用不确定的问题。接口的实现扩展了功能，降低了耦合性。

通过继承和接口实现能够解决“单继承的局限性”

```
//继承得到通用方法，接口实现特有方法
interface Inter1
{
	public void examing(); //这是一些特有的方法，不属于父类，需要通过接口来实现
}
class Person1
{
	private int age;
	private String name;
	Person1(int age,String name)
	{
		this.age = age;
		this.name = name;
	}
	void eat() // 这是父类中的方法，是所有子类所共有的方法，子类通过继承可以得到
	{
		System.out.println("Eating.");
	}
}
class student extends Person1 implements Inter1
{
	student(int age,String name)
	{
		super(age,name);
	}
	public void examing()  //接口中的方法在子类中进行覆盖时需要用public修饰
	{
		System.out.println("examing!");
	}
}
public class Demo5
{
	public static void main(String[] args)
	{
		student s = new student(26,"Rex");
		s.eat(); // 父类方法
		s.examing(); // 特有的方法
	}
}
运行结果：
Eating.
examing!
```

关系

类与类是继承关系，类与接口是实现关系，接口与接口是继承关系，当一个子类所实现的接口继承了其他的接口时，若要该子类实例化，则需要把前面所有继承的接口都要进行覆盖。

```
interface Inter0 // 接口
{
	public void method0();
}
interface Inter1 // 接口
{
	public void method1();
}
interface Inter2 extends Inter1, Inter0 // 接口Inter2继承了接口Inter1, Inter0
{
	public void method2();
}
class Fu implements Inter2 // 这个类要实现接口Inter2,必须要覆盖三个方法，除了覆盖Inter2中的方法，
还要覆盖Inter2所继承的两个接口中的方法，即Inter0中的方法（method0()），与Inter1中的方法
（method()1）,这样才能实例化
{
	public void method0()
	{

	}
	public void method1()
	{

	}
	public void method2()
	{

	}
}
```

提取接口的部分功能

如果我们只需要接口中的部分功能，但是为了实例化，还是必须覆盖接口中的全部方法，这样代码的复用性就会很差。怎么办呢？解决的方法是，**定义一个抽象类，将接口中的方法全部进行“空实现”，然后在其他的类中继承该类。**

抽象类和接口的区别？

1、类与类之间是继承关系；类与接口之间是实现关系。
2、抽象类中可以定义**抽象和非抽象方法**，子类可以**直接使用**，**或者覆盖使用**；接口中定义的**都是抽象方法**，**必须**实现才能使用。

牢记

```
*********************************************************************************
* 类是用于描述事物的共性基本功能。例如，“犬”类的基本功能有，吃、叫、睡 *
* 接口用于定义的都是事物的额外功能。例如，“缉毒”                          *
*********************************************************************************
```

多态

这里说的多态指的是对象的多态性，例如一个“狗”对象既是狗类型，又是动物类型，即有多种形态。当我们**重复的面对各个子类中的一些“共性行为”**的时候，我们应该考虑直接面对父类，这样可以提高代码的复用性。

多态的弊端：不能使用子类的特有方法，因为我们面对的就是共性行为
多态出现的前提：
1、必须有关系、继承、实现，因为只有出现继承、实现，才能出现“子父类”、“接口和子类”，这样才能使用多态。
2、通常有覆盖。

向上转型和向下转型

什么时候使用“向上转型”呢？只使用父类的功能即可完成操作，就使用向上转型。

向上转型的弊端是：只能使用父类中的功能，不能使用子类的特有功能

向下转型的好处是：可以使用子类的特有功能

什么时候用向下转型？需要用到子类型中的特有的方法时，就要对对象进行向下转型，但是一定要进行“匹配判断”

向下转型的弊端是：容易发生ClassCastException（转换类型异常），解决的方法就是对“是否匹配”进行判断，使用一个关键字 instanceof，**使用格式**如下：

对象 instanceof 类型

```
abstract class animal
{
	abstract void eat();
}
class dog extends animal
{
	void eat()
	{
		System.out.println("鱼");
	}
	void catchMouse()
	{
		System.out.println("抓老鼠");
	}
}
class cat extends animal
{
	void eat()
	{
		System.out.println("鱼");
	}
	void catchMouse()
	{
		System.out.println("抓老鼠");
	}
}
public class DuoTai
{
	public static void main(String[] args)
	{
		animal d1 = new cat();  // 向上转型
		d1.eat();
		if(!(d1 instanceof cat))  // 向下转型匹配判断
		{
			System.out.println("类型不匹配！ ");
			return;
		}
		cat c = (cat) d1; // 向下转型（相当于强制类型转换）
		c.catchMouse(); //向下转型之后，就可以调用子类中的方法了
	}
}
```