

第六天总结：

1.Java 中的参数传递问题：

(1) 基本数据类型：形式参数的改变不会影响实际参数。

(2) 引用数据类型：形式参数的改变会影响实际参数，是因为数据传递的是地址值，而不是值，只要地址值发生改变，那么指向就不是同一个对象。

2.面向对象以及面向对象的特点：

(1) 面向过程是以函数为基础，完成操作，强调的是过程，C 语言就是一种面向过程的语言。

(2)面向对象是以对象为基础，完成操作，强调是对象和结果。

(3)面向对象是基于面向过程的。

面向对象的特点：

(1)更加符合人们的思考习惯，思维模式。

(2)将复杂问题简单化了。

(3)从以前的执行者变成了调用者。

事物的体现方式两种：行为和属性。

4.类与对象的关系

简单就是一句话，类是事物行为和属性的集合，它是一个抽象的过程；对象是事物的具体体现。

5.创建对象

格式： 类名 变量 = new 类名();

使用成员变量和成员方法的格式：

A:成员变量 对象名.成员变量

B:成员方法 对象名.成员方法

6.成员变量和局部变量的区别：

A:定义的位置不同

成员变量定义在类中方法外，局部变量定义在方法中，或者在形式参数上。

B:初始化值不同

成员变量有默认的初始化值，而局部变量没有默认的初始化值，如果在定义时不初始化，在编译时可能会报错。

C:在内存中的存储位置不同

成员变量存储在堆内存中，随着对象的存在而存在；而局部变量存储在栈内存中，随着方法的存在而存在。

7.匿名对象

简单说就是没有名字的对象，就是 new 后面的。

应用场景：

(1) 当对象对成员变量和方法只进行一次调用时，可以使用匿名对象。

(2) 匿名对象可以作为参数进行传递。

匿名对象作为参数传递示例：

/*

A:将匿名对象作为参数传递的示例

B:当对象对成员变量和方法只进行一次调用时，可以使用匿名对象。

```

*/

class Demo
{
    int num = 34;
    public void show()
    {
        System.out.println("嘿嘿.....");
    }
}

class NiMingDemo
{
    public static void main(String[] args)
    {
        //Demo1 d = new Demo1();
        //d.show(new Demo());//将匿名对象作为参数进行传递
        new Demo().show();//匿名对象调用成员方法 show()
        int result = new Demo().num;//匿名对象调用成员变量
        System.out.println(result);
    }
}

```

第七天总结：

1. 封装
隐藏属性和实现细节，向外提供公共的访问方式，举例：类、方法其实都是封装的一种体现方式。
2. **private** 关键字
(1) 用于修饰成员变量和成员方法。
(2) 只要被该关键字修饰的内容只能在本类中访问，其他类中无法访问。
3. **private** 的应用：
类中的变量用 **private** 修饰，向外提供 **set** 和 **get** 方法。

```

class StudentDemo
{
    private String name;
    private int age;
    public StudentDemo(){}
    public StudentDemo(String name,int age)
    {
        this.name = name;
        this.age = age;
    }
    public void setName(String name)

```

```

{
    this.name = name;
}
public String getName()
{
    return name;
}
public void setAge(int age)
{
    this.age = age;
}
public int getAge()
{
    return age;
}
}

```

4. 构造方法

格式： 访问权限修饰符 类名(参数.....){}

权限修饰符 public private 等

特点：

A:方法名和类名一样

B:没有返回值类型

C:没有具体的返回值

注意：

A:在使用的过程中如果没有给出构造方法，那么 jvm 会自动给一个无参的构造方法。

B:如果已经手动给出构造方法，那么不会再去使用系统默认的构造方法。

C:建议手动设置构造方法。

5. this 关键字

this 是一个关键字，它代表的是本类对象的引用，也就是说在方法中那个对象调用，this 就代表那个对象，除了静态方法外，所有的方法中有隐含的有一个 this 引用(举例说明)。

/*

除了静态方法外，所有的方法中有隐含有一个 this 引用(举例说明)。

*/

class Demo

{

int num = 100;

public void show()

{

System.out.println(this.num);// 所有的方法中有隐含有一个 this 引用

}

}

class ThisDemo

{

```

public static void main(String[] args)
{
    Demo d = new Demo();
    d.show();
}
}

```

this 关键字的运用场景：

解决成员变量和局部变量相同的问题。

6. static 关键字

static 关键字用于修饰成员方法和成员变量。

静态的特点：

- A: 随着类的加载而加载。
- B: 优先于对象而存在
- C: 被静态修饰的成员被所有的类共享。
- D: 可以被类名直接调用。

在使用 static 时应该注意的两个问题：

A: 在静态方法中没有 this 关键字。

因为 this 是和对象相关，而静态是和类相关，这是两个不同的概念。

也可以理解成为静态优先于对象存在，而 this 只能是在创建对象后才能使用。

B: 静态方法只能访问静态成员变量，不能访问非静态成员变量。

示例说明：

```

class Demo
{
    static int num = 100;
    public static void show()
    {
        //int num = 90;
        //在静态方法中没有 this 关键字。
        //System.out.println(this.num); //无法从静态上下文中引用非静态 变量 this;
        System.out.println(num); // 静态方法只能访问静态成员变量
    }
}

class ThisDemo
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.show();
    }
}

```

静态内容的访问方式：

- A: 通过对象调用
- B: 通过类名调用

什么时候使用 `static` 呢？

当一些内容被几个对象共享是，可以用 `static` 进行修饰。

举例： 学校的饮水机
 厕所等

7. Math 类的随机数

在 `java.lang` 包的类不用导入，系统会默认导入的。

`Math` 类中的方法全是静态方法，用类名直接调用即可。

产生 1-100 之间的一个随机数：

```
int numble = (int)(Math.nextInt()*100+1)
```

第八天总结：

1. 继承

将多个类中的所公共的内容(属性和方法)进行抽取，封装到一个类中，在建立新类时去直接使用那个类中的方法和属性。

好处：

A:提高代码的复用性。

B:使类与类之间产生联系，即是多态的前提。

什么时候使用继承？

当一个类所属另一个类时，也就是说他们之间存在 `is a` 的关系时，就可以使用继承，切记不要为了使用某个功能而去继承。

继承的特点

A: `java` 中只支持单继承，不支持多继承。

B: `java` 中可以有多重继承，即可以有自己的继承体系。

`super` 和 `this` 的区别？

`super` 关键字的使用和 `this` 差不多，

A:在成员方法中的使用

`this.方法名()`----- 本类中的方法

`super.方法名()`-----父类中的方法

B: 在构造方法中的使用

`this.构造方法`-----本类的构造方法

`super.构造方法` -----父类的构造方法

C: 在成员变量中使用

`this.变量`-----本类中的变量

`super.变量`-----父类中的变量

子父类中成员变量的用法：

A:子类和父类中的变量名不同，使用即可。

B:子类和父类中的变量名相同时。(就近原则) 子类对象的在使用的時候：

先找子类中的局部，再找子类中的成员，最后找父类的成员范围。

子父类中成员方法的用法：

A:子类和父类中的方法不同名，使用即可。

B:子类和父类中的方法名相同时。(就近原则) 子类对象的在使用的時候：

先找子类中的方法，再找父类中的方法，

C:方法的重写

在子类中的方法声明(访问权限修饰符、方法名、参数、返回值类型)与父类中相同时,就会存在方法的重写。

方法的重写时应该注意的问题:

a:父类中的私有方法不能被子类重写

b:子类重写后的访问权限一定要大于等于父类的方法访问权限。

c:静态方法只能静态重写。

子父类中构造方法的用法:

子类的初始化过程中,首先会去执行父类的初始化动作

在子类的方法中默认一个 `super()`

在子类方法中如果没有指定使用父类中的构造方法,那么系统会默认使用父类的无参构造,原因是因为子类想要使用父类中的成员,必须在子类初始化之前完成,因此需要先执行父类的初始化过程。

注意:

当父类没有无参构造方法时,那么将会采用以下两种方法初始化

A: 使用 `super(参数.....)`去调用父类中的带参构造方法

B: 使用 `this(参数.....)`调用本类中的其他构造

代码块的执行顺序说明:

静态代码块 > 构造代码块 > 构造方法

注意的是: 静态代码块只执行一次。

示例:

/*

执行顺序:

静态代码块 --> 构造代码块 --> 构造方法。

注意:

静态代码块只执行一次。

*/

class Zi

{

//静态代码块

static

{

System.out.println("zi 静态代码块");

}

//构造代码块

{

System.out.println("zi 构造代码块");

}

public Zi()

{

System.out.println("zi 构造方法");

```

    }
}

class BlockCodeDemo
{
    public static void main(String[] args)
    {
        Zi z = new Zi();
        Zi z2 = new Zi();
    }
}

```

final

final 是一个关键字，用于修饰成员变量、成员方法和类。

特点：

- A: 被 final 修饰的成员变量是一个常量，值不可以改变。
- B: 被 final 修饰的成员方法不可被重写，最终方法
- C: 被 final 修饰的类不能被继承，最终类

第九天总结：

1.多态

定义：某类事物的在不同时刻的不同形态。

(1)多态的前提：

- A:必须要有继承关系(或者实现)。
- B:有方法的重写。
- C:父类(或者接口)引用指向子类对象。

(2)多态的成员的特点：方法有重写，变量没有重写。

A:成员变量

编译看左边，运行看左边。

B: 成员方法

编译看左边，运行看右边

多态的成员的特点示例：

```

class Person
{
    int num = 10000;
    public void show()
    {
        System.out.println("person show="+num);
        System.out.println("person show");
    }
}

```

```
class Student extends Person
```

```
{  
    int num = 100;  
    public void show()  
    {  
        System.out.println("student show="+num);  
        System.out.println("student show");  
    }  
}
```

```
class Teacher extends Person
```

```
{  
    public void show()  
    {  
        System.out.println("teacher show");  
    }  
    public void teach()  
    {  
        System.out.println("teach....");  
    }  
}
```

```
class DuoTaiTest
```

```
{  
    public static void main(String[] args)  
    {  
        Person p1 = new Student();  
        p1.show();  
        Person p2 = new Teacher();  
        p2.show();  
        //p2.teach();//编译报错，因为父类中没有 teach()方法  
    }  
}
```

(3)多态的弊端和解决方法

弊端：父类引用不能使用子类中的特有功能。

解决方法：向下转型

```
class Person
```

```
{  
    int num = 10000;  
    public void show()  
    {  
        System.out.println("person show="+num);  
        System.out.println("person show");  
    }  
}
```



```

    }

}

class Student extends Person
{
    int num = 100;
    public void show()
    {
        System.out.println("student show="+num);
        System.out.println("student show");
    }
}

class Teacher extends Person
{
    public void show()
    {
        System.out.println("teacher show");
    }
    public void teach()
    {
        System.out.println("teach....");
    }
}

class DuoTaiTest
{
    public static void main(String[] args)
    {
        Person p1 = new Student();
        p1.show();
        Person p2 = new Teacher();
        p2.show();
        //p2.teach();//编译报错，因为父类中没有 teach()方法
        Teacher t = (Teacher)p2;// 向下转型
        t.teach();
    }
}

```

4.类型转换中的问题

java.lang.ClassCastException 类型转换异常

5.多态的好处

提高代码的扩展性和可维护性

示例：

```
class Animal
{
    public void show()
    {
        System.out.println("Animal show");
    }
    public void eat()
    {
        System.out.println("Animal eat");
    }
}
class Dog extends Animal
{
    public void show()
    {
        System.out.println("dog show");
    }
    public void eat()
    {
        System.out.println("dog eat");
    }
}
class Pig extends Animal
{
    public void show()
    {
        System.out.println("pig show");
    }
    public void eat()
    {
        System.out.println("pig eat");
    }
}
class Cat extends Animal
{
    public void show()
    {
        System.out.println("cat show");
    }
    public void eat()
    {
        System.out.println("cat eat");
    }
}
```

```

class AnimalTool
{
    public static void printAnimal(Animal a)
    {
        a.show();
        a.eat();
    }
}

```

```

class DuoTaiDemo2
{
    public static void main(String[] args)
    {
        AnimalTool al = new AnimalTool();
        Animal a = new Dog();
        al.printAnimal(a);
    }
}

```

6.抽象类和抽象方法

(1)定义

抽取相同的方法声明，定义在一个类中，而这个没有方法体的方法的类就称为抽象类。

简单说就是没有方法体的方法就称为抽象方法。注意的是：有抽象方法的类一定是抽象类(或者接口)。

(2)特点

A: 用关键字 **abstract** 进行修饰。

B: 有抽象方法的类一定是抽象类(或者接口)。

C: 不能被实例化

说明原因：因为抽象类是抽象的，它不具体指那个对象，既然是不能被实例化的，那么怎么实现其中的功能呢？它是通过子类对象进行实例化的。

D: 抽象类中不一定有抽象方法

有抽象方法是为了强制让子类去实现父类的功能，而非抽象方法是为了让子类继承，提高代码的复用性。

E: 若一个类继承抽象类，要么这个类本身就是抽象类，要么该类重写父类中的所有方法，若该类本身就是抽象类，那么也可以不用重写父类中的方法，只是这样做没有任何意义。

F: 它的作用是：强制要求子类必须完成父类中的某些功能。

(3)抽象类中的成员特点

A: 成员变量

可以有成员变量和常量。

B: 构造方法

可以有构造方法，不能被实例化；它是通过子类对象进行实例化的。原因是子类在访问父类中的数据时，首先要进行父类初始化。

C: 成员方法

可以有抽象方法(强制执行的功能),也可以由非抽象方法(子类继承,提高代码复用性)。

(4) 抽象关键字 **abstract** 不可以和哪些关键字共存?

****private**

***私有的,外部直接无法访问。

****static**

***那么这个时候抽象方法就可以可以通过类名调用,但是这样是没有意义的。

****final**

*****final** 修饰的方法不能被重写。所以它和 **abstract** 冲突。

7.接口

(1)定义

接口是一种特殊的抽象类,用 **interface** 关键字定义,类和接口的关系是实现,用 **implements** 关键字实现,在接口中的方法都是抽象方法。

(2)接口特点

A: 接口不能被实例化

B: 接口中的方法要么被子类重写,要么子类也是抽象类。

(3)接口中的成员特点

A: 成员变量

接口中只有常量,因为变量都有默认的修饰符 **public static abstract** 修饰。

B: 构造方法

没有构造方法,因为成员变量都是静态的,父类对象不需要初始化。

C: 成员方法

接口中的方法都是抽象的,因为方法都有默认的修饰符 **public abstract** 进行修饰。

8.类与接口的关系(实现)

注意:所有的类都直接或者间接继承自 **object** 类, **object** 类提供了一个无参的构造,子类只能调用这个无参的构造方法,如果调用有参的构造方法就会编译报错,接口不能继承自 **object** 类。

A: 类与类之间的关系

继承关系,在类中 **java** 只支持单继承,支持多层继承。

B: 类与接口之间的关系

实现关系,可以单实现,可以多实现;也可以在继承的同时实现多个接口。

C: 接口与接口之间的关系

继承关系,可以单继承,可以多继承。

9.接口的思想特点

A: 功能扩展 -----接口多态

B: 接口多实现

C: 对外暴露规则

10.面试题

根据提示,用所学知识描述下面事物那些是类,抽象类,接口。

乒乓球运动员和教练

篮球运动员和教练

为了出国交流，乒乓球运动员和教练需要说英语。

```
//说英语的接口
interface SpeakEnglish
{
    public abstract void speak();
}
//定义人的抽象类
abstract class Person
{    //定义姓名
    private String name;
    //定义年龄
    private int age;

    public Person(){}

    public Person(String name,int age)
    {
        this.name = name;
        this.age = age;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return name;
    }
    public void setAge(int age)
    {
        this.age = age;
    }
    public int getAge()
    {
        return age;
    }
    //定义睡觉的非抽象方法
    public void sleep()
    {
        System.out.println("运动员和教练员休息");
    }
    //定义抽象吃饭的方法
    abstract public void eat();
}
```

```

}
//定义教练员抽象类
abstract class Coach extends Person
{
    public Coach(){}
    //定义教练员指导的抽象方法
    abstract public void teach();
}
//定义运动员抽象类
abstract class Player extends Person
{
    public Player(){}
    //定义运动员练习的抽象方法
    abstract public void exercise();
}
//定义乒乓球运动员类
class PingPangPlayer extends Player implements SpeakEnglish
{
    public PingPangPlayer(){}
    //定义乒乓球运动员重写父类中说的方法
    public void speak()
    {
        System.out.println("乒乓球运动员学习英语");
    }
    //定义乒乓球运动员重写父类中的 eat()方法
    public void eat()
    {
        System.out.println("乒乓球运动员吃蛋炒饭.....");
    }
    //定义乒乓球运动员重写父类中的 exercise()方法
    public void exercise()
    {
        System.out.println("乒乓球运动员练习接发球.....");
    }
}
//定义篮球运动员类
class BasketPlayer extends Player
{
    public BasketPlayer(){}
    //定义篮球运动员重写父类中的 eat()方法
    public void eat()
    {
        System.out.println("篮球运动员吃牛肉粉.....");
    }
}

```

```

    }
    //定义篮球运动员重写父类中的 exercise()方法
    public void exercise()
    {
        System.out.println("篮球运动员练习，运球，过人，投篮.....");
    }
}
//定义乒乓球教练类
class PingPangCoach extends Coach implements SpeakEnglish
{
    public PingPangCoach(){}

    public void speak()
    {
        System.out.println("乒乓球运动员学习英语");
    }
    public void eat()
    {
        System.out.println("乒乓球教练员吃盒饭.....");
    }
    public void teach()
    {
        System.out.println("乒乓球教练员指导运动员接发球.....");
    }
}
//定义篮球教练类
class BasketCoach extends Coach
{
    public BasketCoach(){}

    public void eat()
    {
        System.out.println("篮球教练员吃牛肉.....");
    }
    public void teach()
    {
        System.out.println("篮球教练员指导运动员，运球，过人，投篮.....");
    }
}

//定义测试类
class TestDemo
{
    public static void main(String[] args)

```

```

{
    PingPangPlayer ppp = new PingPangPlayer();
    ppp.setName("王浩");
    ppp.setAge(30);
    System.out.println(ppp.getName()+"是一个乒乓球运动员，他的年龄是"+ppp.getAge()+"
岁");
    ppp.eat();
    ppp.exercise();
    ppp.sleep();
    System.out.println("*****");
    BasketPlayer bp = new BasketPlayer();
    bp.setName("姚明");
    bp.setAge(40);
    System.out.println(bp.getName()+"是一个篮球运动员，他的年龄是"+bp.getAge()+"岁");
    bp.eat();
    bp.exercise();
    bp.sleep();
    System.out.println("*****");
    PingPangCoach ppc = new PingPangCoach();
    ppc.setName("刘国梁");
    ppc.setAge(45);
    System.out.println(ppc.getName()+"是一个乒乓球教练员，他的年龄是"+ppc.getAge()+"
岁");
    ppc.eat();
    ppc.teach();
    ppc.sleep();
    System.out.println("*****");
    BasketCoach bc = new BasketCoach();
    bc.setName("宫鲁鸣");
    bc.setAge(54);
    System.out.println(bc.getName()+"是一个篮球教练员，他的年龄是"+bc.getAge()+"岁");
    bc.eat();
    bc.teach();
    bc.sleep();
    System.out.println("*****");
}
}

```

第十天总结：

1.包

包相当于文件夹，用于区分相同的类名。

格式：

package 类名;

可以是单级包，也可以是多家包。

2.代码顺序

package > import > class

3.带包的类的编译和运行

方式 1: 手动建包

A: **javac** 生成 **class** 文件

B: 手动建包

C: 把 **class** 文件拷贝到文件夹中

D: 带包运行

命令: **java 包名.源文件名.java**

例如: **java com.itcast.PackageDemo**

方式 2: 自动建包

在编译的时候自动建包

命令: **java -d.Demo.java**

4.导包

用一个类就需要这个类的全路径，**java** 中提供一种导包机制

格式:

import 包名 1.包名 2.类名; 要那个类导那个类。

5.访问权限修饰符

	本类	同包	不同包(继承中的子类)	不同包(无关类)
private	Y			
默认	Y	Y		
protected	Y	Y	Y	
public	Y	Y	Y	Y

6. 不同修饰符修饰的内容(和内部类无关)

	类	成员变量	成员方法	构造方法
private	Y	Y	Y	N
默认	Y	Y	Y	Y
protected	N	Y	Y	Y
private	Y	Y	Y	N
static	N	Y	Y	Y
abstract	Y	N	Y	N
final	Y	Y	Y	N

7.内部类

将一个类定义在一个类里面，这个类就称为内部类，分为成员内部类和局部内部类。

(1).访问特点

A: 内部类可以直接访问外部类中的成员变量，包括私有的。

B: 外部类访问内部类需要在外部类中某个方法中创建内部类对象。

(2)外部类直接使用内部类中的成员

格式:

外部类.内部类.变量名 = **new** 外部类名().**new** 内部类名();

示例:

/*

内部类和外部类之间的调用方式：

内部类怎么调用外部类成员呢？

当成自己的成员来用

外部类调用内部类：

在自己某个方法中，创建一个内部类对象

内部类名 对象引用 = new 内部类名();

在测试类中直接调用内部类，格式：

外部类名.内部类名 对象引用 = new 外部类名().new 内部类名();

```
*/
class Outer
{
    private int num = 1000;
    class Inner
    {
        public void show()
        {
            System.out.println(num);
        }
    }

    public void show2()
    {
        Inner i = new Inner();
        i.show();
    }
}
class InnerDemo
{
    public static void main(String[] args)
    {
        //Outer.Inner i = new Outer().new Inner();
        Outer o = new Outer();
        o.show2();
    }
}
```

(3)成员内部类的修饰符

A: private

B: static

示例：

```
/*
```

内部类的修饰符:

A:private 为了安全考虑。常见用法。

B:static 为了方便。常见用法。

```
*/
class Outer
{
    private class Inner//定义一个用 private 修饰符修饰的内部类
    {
        public void show()
        {
            System.out.println("private inner show");
        }
        public void show2()
        {
            System.out.println("private inner show2");
        }
    }
    public void mothed()
    {
        Inner i = new Inner();
        i.show();
        i.show2();
    }
    static class Inner2//定义一个用 static 修饰符修饰的内部类
    {
        public void show3()
        {
            System.out.println("static inner2 show3");
        }
        public static void show4()//静态方法
        {
            System.out.println("static inner2 show4");
        }
    }
}
//定义测试类
class InnerDemo3
{
    public static void main(String[] args)
    {
        Outer o = new Outer();
        o.mothed();
        //访问静态内部类的格式:
        //外部类.内部类 变量 = new 外部类.内部类();
    }
}
```

```

        Outer.Inner2 oi = new Outer.Inner2();
        oi.show3();
        oi.show4();
    }
}

```

(4)局部内部类(方法中定义的类)

结论：局部内部类使用局部变量，局部变量必须用 **final** 修饰。
为什么?(面试)

局部变量会在方法调用完毕后，立马消失。

而局部内部类中如果有地方使用着局部变量，当方法消失后，这个方法区中的内容还没有消失，也就是说这个变量还必须存在。

所以，为了延长局部变量的生命周期，就加 **final** 修饰了。

示例：

```

class Outer
{
    //private int num = 10;

    public void method()
    {
        final int num = 10;
        //定义类
        class Inner
        {
            //int num = 10;
            public void show()
            {
                //int num = 20;
                System.out.println(num);
            }
        }

        //创建对象
        Inner i = new Inner();
        i.show();
    }

    public void function()
    {
        //错误
        //Inner i = new Inner();
        //i.show();
    }
}

```

```

class InnerTest3
{
    public static void main(String[] args)
    {
        Outer o = new Outer();
        o.method();
    }
}

```

8. 匿名内部类(局部内部类)

(1) 前提

必须存在一个类、接口或者抽象类。

(2) 格式:

```

new 类名(抽象类)或者接口(){
    重写接口中的方法或者抽象类中的方法,
    自己定义新方法。
}

```

可理解为:

是一个继承了类或者实现了接口的匿名子类对象。

(3) 什么时候使用匿名内部类

当接口或者抽象类中的方法在三个以下的时候, 可以使用匿名内部类。

最常用:

当一个方法的形式参数是接口或者抽象类, 用匿名内部类实现。

思想:

所有父类或者接口出现的地方都可以用子类替代。