



java 总结

第二季：Java 面向对象篇

每天一点点

隋隋隋丙跃

2012/4/14

这个技术是什么？

这个技术什么特点？

这个技术怎么用*？

这个技术什么时候用？

这个技术怎么讲？

——BXD

目录

1. 类与对象	3
1.1. 定义类	3
1.2. 定义成员	3
1.3. 定义方法	3
1.4. 对象的创建及其内存分配	4
1.5. this 的概念	5
1.6. 对象与二维数组的内存分配图	5
2. 方法	6
2.1. 方法的参数传递	6
2.2. 递归方法	7
2.3. 方法重载	8
2.4. 变量：成员变量 VS 局部变量	8
2.5. Main 函数详解	9
3. 三大特性一：封装（encapsulation）	9
3.1. 封装的特点及原则	9
3.2. 访问控制修饰符	9
3.3. 构造函数	10
3.4. 静态特点及注意事项	11
3.5. 静态 VS 非静态	11
3.6. 静态什么时候用？	11
3.7. 静态代码块的特点和作用？	11
3.8. 单例设计模式	12
4. 三大特性二：继承（inheritance）	12
4.1. 什么是继承	13
4.2. 为什么要使用继承	13
4.3. 继承的特点	13
4.4. 强制类型转换	13
4.5. 子类覆盖（Override）父类方法	13
4.6. 继承的应用细节	14
4.7. 子类对象实例化	14
4.8. 子类当做父类使用时需要注意	14
4.9. 抽象类概述	15
4.10. 抽象的抽象：接口	15
4.11. 一个对象的实例化过程：	17
5. 三大特性三：多态（Polymorphism）	18
5.1. 多态概述	18
5.2. 多态成员特点	18
5.3. 向上转型	18
5.4. Instanceof	19
5.5. final 修饰符	20
5.6. 内部类概述	20
5.7. 内部类使用方法及注意	22

5.8.	静态内部类&局部内部类	24
5.9.	匿名内部类	25
6.	错误收集	25
7.	异常	25
7.1.	异常概述	26
7.2.	异常小例一：	27
7.3.	异常的分类	28
7.4.	throw VS throws	29
7.5.	异常处理机制	31
7.6.	什么时候 try 什么时候 throw	31
7.7.	访问异常信息	32
7.8.	异常处理原则	33
7.9.	try catch final 组合特点	33
7.10.	异常处理中的 Finally	34
8.	线程	34
8.1.	进程与线程	34
8.2.	多线程的利与弊	35
8.3.	JVM 启动时的多线程	35
8.4.	创建线程两种方式 and 区别	36
8.5.	start 方法和 run 方法区别	37
9.	后续	39

Java 是面向对象的程序设计语言，提供了定义类、成员等基本功能，类可以理解为一种自定义的数据类型，可以使用类来定义变量，所有使用类定义的变量都是引用变量，他们会引用到类的对象。

类是用于描述客观世界里某一类对象的共同特征，对象则是类的具体存在，java 使用构造函数来创建该类的对象

面向对象的特征：

封装(encapsulation)

继承(inheritance)

多态(polymorphism)

1. 类与对象

- 类是对象的描述，对象是类的实例。执行者——《》》》指挥者

1.1. 定义类

```
[修饰符] class 类名
{
    成员;
    构造函数;
    方法;
}
```

1.2. 定义成员

[修饰符] 成员类型 成员名 [=默认值];

private int age = 18;

修饰符：java 提供了 3 个 private、protected、public、还有一个默认省略的修饰符，修饰成员，方法等，且只能出现其一，可以 static final 组合使用。

	Private	Default	Protected	Public
同一个类中	√	√	√	√
同一个包中		√	√	√
子类中			√	√
全局范围内				√

1.3. 定义方法

[修饰符] 方法返回值类型 方法名（形参列表）

```
{
    方法体（可无）;
}
```

方法的修饰符：可以省略，可以使 public、protected、private、static、final、abstract。前三者只能出现一个，后两者只能出现一个。

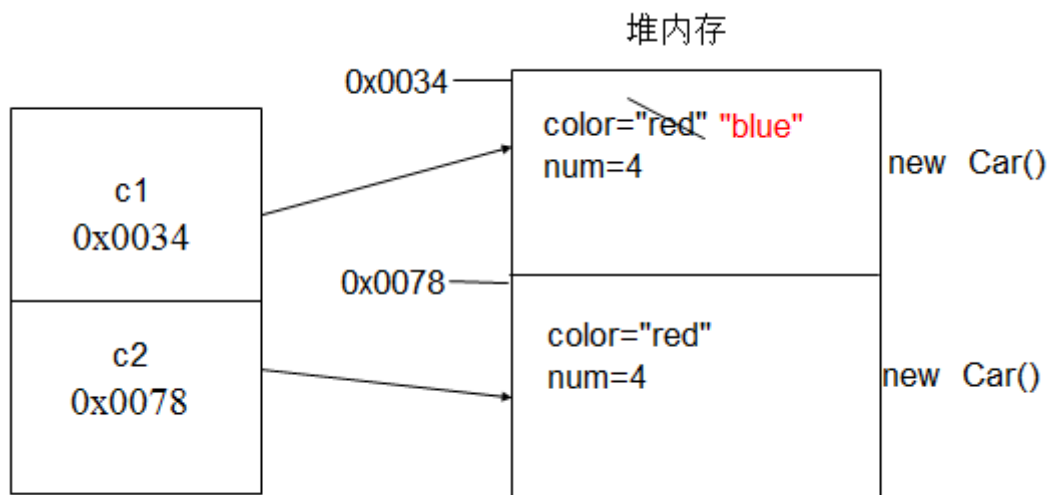
方法返回值类型：可以使基本数据类型和引用类型；如果声明了方法的返回值，则方法体内必须有一个有效的 return 语句，返回一个与声明相同类型的变量或者表达式。Void：表示无返回值。

形参列表：0 到多个，中间用逗号隔开形参类型与形参名空格隔开。

方法体执行顺序：排在方法体前面的先执行，后面的后执行。

1.4. 对象的创建及其内存分配

```
class Car//对 Car 这类事物进行描述
{
    String color = "red";
    int num = 4;
    void show()
    {
        System.out.println("color="+color+"..num="+num);
    }
}
class CarDemo
{
    public static void main(String[] args)
    {
        Car c1 = new Car();//声明并创建对象
        c1.color="blue";//修改对象的属性
        Car c2 = new Car();//声明并创建对象 c2
    }
}
```



1.5. this 的概念

特点：this 代表其所在函数所属对象的引用。

换言之：this 代本类对象的引用。

什么时候使用 this 关键字呢？

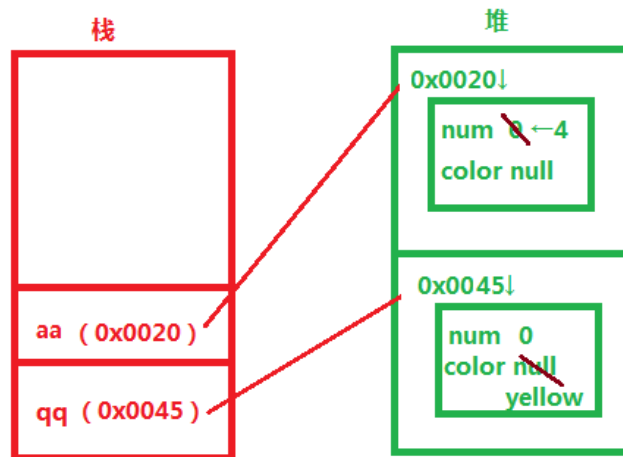
方法中局部变量和成员变量重名，想调用成员变量时就可以使用 this. 变量名形式访问成员变量。
在方法中要将调用该方法的对象作为参数传递给另一个方法时，可以将 this 作为实参传给该方法。
在内部类中访问外部类的成员时，需要使用外部类名.this. 成员名形式访问。

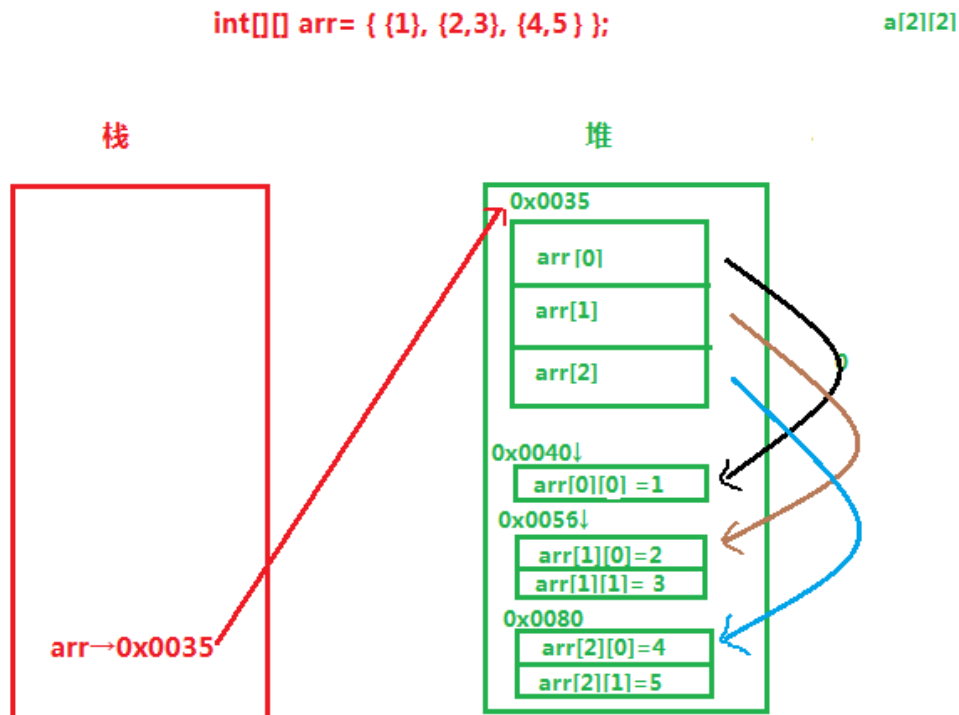
例：

```
class Person{
    int age;
    Person(int age)
    {
        this.age = age;
    }
}
```

1.6. 对象与二维数组的内存分配图

```
Car aa = new Car();
Car qq = new Car();
aa.num = 4;
qq.color = "yellow";
qq.run();
```





2. 方法

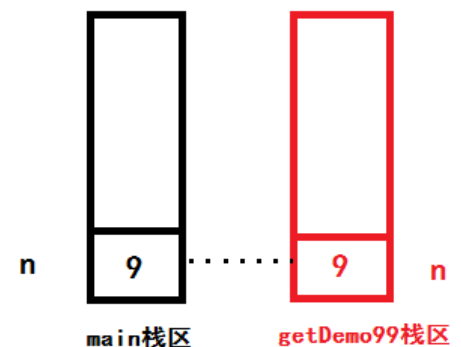
- 方法只能定义在类里，不能独立定义。
- 方法要么属于类，要么属于该类的某个对象。
- 永远不能独立执行方法，执行方法必须使用类或者对象作为调用者
- 函数的特点：
 - 可将功能代码进行封装
 - 便于对该功能进行复用
- 2 个明确
 - 返回类型的确定；这个功能实现过程中是否需要未知参数参与预算

2.1. 方法的参数传递

Java 按值传递。

例如：

```
class Demo{
public static void main(String[] args)//主函数入口
{
    int n = 9;
```



```

        getDemo99(n); //调用九九乘法表函数
    }
    public static void getDemo99(int n) //将 9 传递给 int 型的 n
    {    //forfor 循环，根据传递的参数，打印乘法表
        for (int i = 1; i <= n; i++) //外层循环控制行数
        {
            for (int j = 1; j <= i ; j++) //内层循环控制列数
            {
                System.out.print(j+"*"+i+"=" + (i*j)+"\t");
            }
            System.out.println();
        }
    }
}

```

```

C:\windows\system32\cmd.exe

D:\>java Demo
1*1= 1
1*2= 2 2*2= 4
1*3= 3 2*3= 6 3*3= 9
1*4= 4 2*4= 8 3*4= 12 4*4= 16
1*5= 5 2*5= 10 3*5= 15 4*5= 20 5*5= 25
1*6= 6 2*6= 12 3*6= 18 4*6= 24 5*6= 30 6*6= 36
1*7= 7 2*7= 14 3*7= 21 4*7= 28 5*7= 35 6*7= 42 7*7= 49
1*8= 8 2*8= 16 3*8= 24 4*8= 32 5*8= 40 6*8= 48 7*8= 56 8*8= 64
1*9= 9 2*9= 18 3*9= 27 4*9= 36 5*9= 45 6*9= 54 7*9= 63 8*9= 72 9*9= 81

D:\>

```

2.2. 递归方法

一个方法本身调用自己，这个方法就被成为递归方法。原则：向已知方向递归。

/*

已知： $f(0)=0, f(1)=4, f(n+2)=2*f(n+1)+f(n)$ ，其中 n 是大于 0 的整数，求 $f(10)$

分析： f_0, f_1 已知，所以要想小的方向递归，设 $x=n+2$ ，即 $n=x-2$ ，带入公式得到：

$f(x)=2*f(x-1)+f(x-2)$;

*/

```

public class Recursive
{
    public static int fn(int n)
    {
        if (n == 0)
            return 1;
        else if (n == 1)
            return 4;
        else

```



```
        //方法中调用它自身，就是方法递归
        return 2 * fn(n - 1) + fn(n - 2);
    }

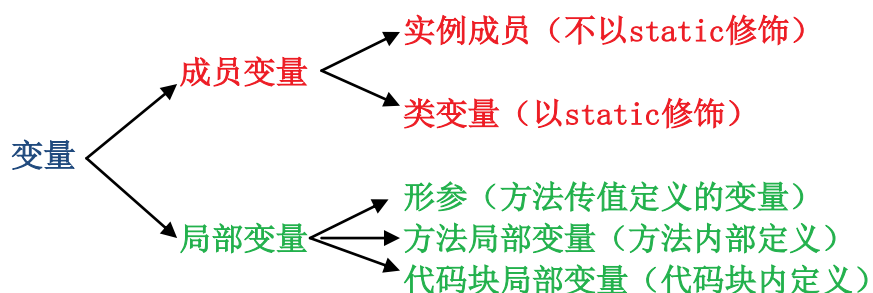
    public static void main(String[] args)
    {
        //输出 fn(10) 的结果
        System.out.println(fn(10));
    }
}
```

2.3. 方法重载

同一个类中允许多个同名函数存在，函数的重载和返回值类型修饰符无关。即同名不同参。（包括参数个数，参数顺序）

```
class Chongzai
{
    void show()
    {
        System.out.println("show");
    }
    void show(int n)
    {
        System.out.println("show"+n);
    }
}
```

2.4. 变量：成员变量 VS 局部变量



区别：

- 1: 成员变量定义在类中，整个类中都可以访问
局部变量定义在函数，语句，局部代码块中，只在所属的区域有效。
- 2: 成员变量存在于堆内存的对象中
局部变量存在于栈内存的方法中

- 3: 成员变量随着对象的创建而存在, 随着对象的消失而消失
局部变量随着区域的执行而存在, 随着所属区域的结束而释放
- 4: 成员变量都有默认初始化
局部变量必须初始化。

2.5. Main 函数详解

```
public static void main(String[] args)
```

主函数特殊之处:

- 1, 格式是固定的。
- 2, 被 jvm 所识别和调用。

public: 因为权限必须是最大的。

static: 不需要对象的, 直接用主函数所属类名调用即可。

void: 主函数没有具体的返回值。

main: 函数名, 不是关键字, 只是一个 jvm 识别的固定的名字。

String[] args: 这是主函数的参数列表, 是一个数组类型的参数, 而且元素都是字符串类型。

3. 三大特性一: 封装 (encapsulation)

- 隐藏对象的信息, 不允许外部程序直接访问, 而是通过该类所提供的方法来实现对内部信息的操作

3.1. 封装的特点及原则

好处:

- 将变化隔离。
- 便于使用。
- 提高重用性。
- 提高安全性。

封装原则:

- 将不需要对外提供的内容都隐藏起来。
- 把属性都隐藏, 提供公共方法对其访问。

3.2. 访问控制修饰符

Java 提供了 3 个访问控制符: private、protected、public, 还有一个默认的一共四个访问控制级别控制访问级别表:

	Private(当前访问权限)	Default (包访问权限)	Protected(子类访问权限)	Public (公共访问权限)
同一个类中	√	√	√	√
同一个包中		√	√	√

子类中			√	√
全局范围内				√

Private（当前访问权限）：将成员变量私有化，对外提供对应的 set，get 方法对其进行访问。提高对数据访问的安全性

Protected：子类可访问，通常是希望子类重写方法。

Public：通常提供对外方法，来访问被封装的数据。

```
public class Person
{
    //将Field使用private修饰，将这些Field隐藏起来
    private String name;
    private int age;//提供方法来操作age Field
    public void setAge(int age)
    {
        if (age > 100 || age < 0)//执行合理性校验，要求用户年龄必须在0~100之间
        {
            System.out.println("您设置的年龄不合法");
            return;
        }
        else
            this.age = age;
    }
    public int getAge()
    {
        return this.age;
    }
}
```

3.3. 构造函数

构造函数（Constructor）是一个特殊的函数。用于创建实例时执行初始化，是创建对象的重要途径，因此java类必须包含一个或一个以上的构造函数。

- 构造函数 VS 一般函数

构造函数：创建对象时，就会调用与之相应的构造函数，来给对象初始化

一般函数：创建对象后，需要函数功能时才调用

- 构造函数重载

同一个类里有多个构造函数，他们的形参列表不同，即被称为构造函数的重载。

- 默认构造函数

每一个类都有构造函数，即使我们没有显式定义构造函数，也会生成一个默认无参的构造函数，其中没有任何内容。

注意：如果我们定义了一个有参的构造函数，那么就没有默认构造函数了

3.4. 静态特点及注意事项

特点:

是一个修饰符，用于修饰成员

静态修饰的成员被所有对象所共享，存储的是共享数据，而非静态存储的是对象特有数据

静态优先于对象存在，因为 static 的成员随着类的加载就已经存在

静态修饰的成员可以用类名.静态变量直接调用

注意事项:

静态方法只能访问静态成员（非静态方法既可以访问静态也可以访问非静态）

静态方法不可以使用 this 和 super 关键词

主函数是静态的

3.5. 静态 VS 非静态

static 关键字用来修饰类的成员，被这个关键字修饰的成员都和类加载有关。

JVM 运行时不会将所有类加载到内存，因为无法确定程序中要使用哪些。类在第一次使用时加载，只加载一次。

静态不能调用非静态（静态在的时候，非静态还不在内存中存在呢）

3.6. 静态什么时候用？

1，静态变量。

当分析对象中所具备的成员变量的值都是相同的。

这时这个成员就可以被静态修饰。

只要数据在对象中都是不同的，就是对象的特有数据，必须存储在对象中，是非静态的。

如果是相同的数据，对象不需要做修改，只需要使用即可，不需要存储在对象中，定义成静态的。

2，静态函数。

函数是否用静态修饰，就参考一点，就是该函数功能是否有访问到对象中的特有数据。

简单点说，从源代码看，该功能是否需要访问非静态的成员变量，如果需要，该功能就是非静态的。

如果不需要，就可以将该功能定义成静态的。当然，也可以定义成非静态，

但是非静态需要被对象调用，而仅创建对象调用非静态的

没有访问特有数据的方法，该对象的创建是没有意义。

3.7. 静态代码块的特点和作用？

特点:随着类的加载而执行。而且只执行一次。

作用: 用于给类初始化。

扩展: 局部代码块: 对局部变量的生命周期进行控制。

构造代码块：对所有对象进行初始化。

3.8. 单例设计模式

解决的问题，以及设计思想。代码体现，全部都要会！

解决的问题：

解决的问题：就是可以保证一个类在内存中的对象的唯一性。

必须对于多个程序使用同一个配置信息时，就需要保证该对象的唯一性。

解决方法：

- 1: 不允许其他程序用 new 创建该类对象；
- 2: 在该类创建一个本类实例。
- 3: 对外提供一个方法让其他程序可以获取该对象

步骤：

- 1: 私有化该类的构造函数
- 2: 通过 new 关键字在本类中创建一个本类对象。
- 3: 定义一个 public 的方法，将创建的对象返回。

恶汉式：先加载

```
class Single
{
    private static Single s = new Single();
    private Single() {}
    public static Single getInstance()
    {
        return s;
    }
}
```

懒汉式：延迟加载

```
public class Single2 {
    private static Single2 sing;
    private Single2() {}
    public static Singleton getInstance() {
        if (sing == null) {
            sing = new Singleton();
        }
        return sing;
    }
}
```

4. 三大特性二：继承（inheritance）

- 现有父后有子，单继承多实现，关键字 extends

- 继承弊端：打破了封装性

4.1. 什么是继承

在程序中，可以使用 `extends` 关键字让一个类继承另外一个类。

继承的类为子类(派生类)，被继承的类为父类(超类，基类)。

子类会自动继承父类所有的方法和属性。

4.2. 为什么要使用继承

当我们发现一个类的功能不行，方法不够用时，就可以派生子类，增加方法。

当我们需要定义一个能实现某项特殊功能的类时，就可以使用继承。

最终还是为了一个目的，提高代码的复用性。

当我们定义一个类时，发现另一个类的功能这个类都需要，而这个类又要增加一些新功能时，就可以使用 `extends` 关键字继承那个类，这样那个被继承类的功能就都有了，不必重写编写代码。这时只要在新的类中编写新的功能即可，原有代码就被复用了。

4.3. 继承的特点

Java 只支持单继承，不支持多继承，但是可以多重继承

因为如果一个类继承多个类，多个类中有相同的方法，子类调用该方法时就不知道该调用哪一个类中的方法了。

注意接口可以多继承

4.4. 强制类型转换

把一个子类当做父类来用的时候，不能调用子类特有方法。

因为编译时编译器会做语法检查，看到变量是父类类型那么就会到父类中查找是否有该方法，没有则报错。

这种情况下，就需要强制类型转换，将父类类型强转成子类类型。

以(子类名)变量名形式进行强制类型转换

强制类型转换时，无论类型是否匹配编译都不会报错，但如果类型不匹配运行会报错，我们可以使用 `instanceof` 进行判断，编译时预知错误。

在子类当做父类来用时，不能调用特有方法，如果一定要调用，就需要强制类型转换回子类。在做转换时最好 `instanceof` 判断一下类型是否匹配。

4.5. 子类覆盖 (Override) 父类方法

覆盖方法必须和被覆盖方法具有相同的方法名称、参数列表和返回值类型。

子类的方法返回值类型可以是父类方法返回值类型的子类。

如果在子类中想调用父类中的那个被覆盖的方法，我们可以用“super. 方法名”的格式。

如果直接调用方法，是在当前子类中先查找，如果子类有会调用子类的。使用 super 形式只在父类中查找，子类有没有都不调用。

覆盖方法时，不能使用比父类中被覆盖的方法更严格的访问权限。

因为有可能将子类对象当做父类对象来使用，那么能获取到的父类对象中的方法在子类中必须都能获取到。

静态只能覆盖静态或者被静态覆盖

覆盖方法时，不能比父类抛出更多的异常。

子类只能比父类强，不能比父类弱。

重载（Overload）和重写（Override）的区别：

重载是方法名相同，参数列表不同，和返回值类型无关。

重写是方法名、参数列表、返回值类型全相同。

Override 注解，可以检查覆盖是否成功

4.6. 继承的应用细节

子类不继承父类私有成员

父类中私有成员对外不可见，子类对象中无法访问这些成员。

构造函数不被继承

构造函数通常用来初始化类的成员变量，父类和子类的成员变量不同，初始化方式也不同，构造函数的名字也不同。

4.7. 子类对象实例化

子类构造函数中可以使用 super 关键字调用父类构造函数。

在子类创建对象时一定会调用父类构造函数。即使没有显式调用，也会默认调用父类无参构造函数。

在子类中第一行用 this 关键字去调其他的构造方法，这时系统将不再自动调父类的。但其他构造函数中会调用父类构造函数。

在构造方法中 this 和 super 关键字只能出现一次，而且必须是第一个语句。

以后在设计类的时候，最好定义一个无参的构造方法，不然子类实例化的时候就容易出错。

4.8. 子类当做父类使用时需要注意

当我们在调用某个类的一个方法时，此方法声明需要一个父类对象，这时我们可以将一个子类对象作为实参传递过去，注意此时方法定义的形参为父类，在方法中使用父类变量调用方法时，其实是调用子类的方法。

思考：上述情形下，在方法中用父类变量访问属性，访问的是子类还是父类的属性？

在把子类当做父类来用时，使用父类变量访问方法，访问的是子类的方法，因为虚拟机找到变量引用的地址，根据这个地址来访问方法，这叫动态分配。

这种机制没有被使用到类的成员变量上，如果用父类变量访问属性，那么会直接找到父类的属性，不会看地址是哪个对象。

4.9. 抽象类概述

笼统，模糊，看不懂！不具体。这就是抽象

- **有得有失：**
得：抽象类多了一个能力：可以包含抽象方法；失：不能创建对象。
- **特点及细节：**
 - 1：方法只有声明，没有实现，该类就是一个抽象类。需被 `abstract` 修饰，抽象方法必须定义在抽象类中。
 - 2：抽象类不可以被实例化，因为调用抽象方法没有意义。
 - 3：抽象类必须由其子类覆盖所有抽象方法后，该子类才可以实例化。否则，这个子类还是一个抽象类。
- **抽象类成员**
成员变量、方法、构造函数、初始化块、内部类、枚举类(没讲呢)6种。
- **抽象类的作用**
从多个具有相同特征的类中抽象出一个类，一这个抽象类为其子类的模板，从而避免了子类设计的随意性。
- **使用注意**
定义抽象方法后面没有 `{}`
`Final` 修饰的方法不能重写，所以不能与 `abstrac` 同时使用。
`Static` 修饰的方法表示该方法属于类，通过类就可以调用，所以不能与 `abstract` 同时使用。
`Abstract` 修饰的方法不能定义为 `private` 权限，因为方法需要被子类重写。

4.10. 抽象的抽象：接口

✧ 抽象再抽象就是接口了

- **接口概念**
接口是从多个相似的类中抽象出来的规范，接口不提供任何实现。接口体现的是规范和实现分离的设计。例如主机板上提供的 PCI 插槽，只要一块显卡遵守 PCI 规范，接口就可以插入 PCI 插槽，与该主板正常通信。至于这块显卡是那个厂商制造的无需关心。
因此，接口定义的只是多个类或者抽象类共同的行为规范，这些行为是与外部交流的通道，也就意味着接口定义的是一组公用方法。
接口不可以创建对象，
- **接口的特点**
接口中当方法都是抽象方法，成员变量都是全局变量
接口中默认省略 `public static final` 关键字
接口中的成员都是公共的特点。
- **接口的定义及使用注意**
[修饰符] `interface` 接口名 `extends` 父接口 1, 父接口 2
{
 常量; (且只能是常量, 省略 `public static final`)
 抽象方法; (且只能是抽象方法, 省略 `abstract`)
}


```
}
```

- 1: 修饰符 public 或者省略，省略默认采用包访问权限
- 2: 接口命名规范，接口名首字母大写，多个单词无需分隔符。
- 3: 接口只能继承接口，不能继承类
- 4: 接口可以由多个直接父接口，但是接口只能继承接口，不能继承类。
- 5: 如果接口为 public 则该文件名应该与接口名相同

例：

```
public interface Output
{
    Int MAX_SIZE = 50; //省略 public static final
    Void out (); //省略 public abstract
}
```

● 实现接口

[修饰符] class 类名 extends 父类名 implements 接口 1, 接口 2...

例子：

```
interface InterfaceA //定义接口 A
{
    public static final int INA_NUM = 10; //声明变量 A_NUM
    public abstract void showA(); //声明 showA 抽象方法，由子类实现
}

interface InterfaceB
{
    public static final int INB_NUM = 20;
    public abstract void showB();
}

interface InterfaceC extends InterfaceA, InterfaceB //接口 C 继承 A、B 两个接口
{
    public static final int INC_NUM = 30;
    public abstract void showC();
}

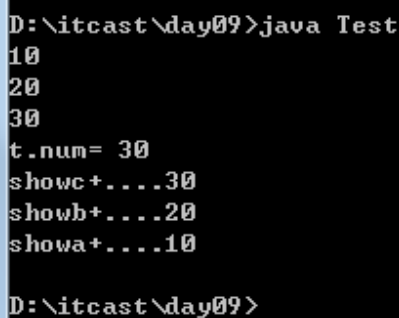
class TC implements InterfaceC //定义 TC 类，实现 C 接口，必须实现 ABC 三个接口的所有抽象方法。
{
    int num = INC_NUM;
    public void showC()
    {
        System.out.println("showc+..." + INC_NUM);
    }
    public void showB()
    {
        System.out.println("showb+..." + INB_NUM);
    }
}
```

```

        public void showA()
        {
            System.out.println("showa+...."+INA_NUM);
        }

class Test
{
    public static void main(String[] args)
    {
        System.out.println(InterfaceA.INA_NUM);
        System.out.println(InterfaceB.INB_NUM);
        System.out.println(InterfaceC.INC_NUM);
        TC t = new TC();
        System.out.println("t.num= "+t.num);
        t.showC();
    }
}

```



```

D:\itcast\day09>java Test
10
20
30
t.num= 30
showc+....30
showb+....20
showa+....10
D:\itcast\day09>

```

```

        t.showB();
        t.showA();
    }
}

```

4.11. 一个对象的实例化过程:

例如: `Person p = new person()`

- 1: jvm 首先回去读取指定路径下的 `Person.class` 文件, 并加载进内存
并会先加载 `Person` 的父类, (如果有直接父类的情况下)
- 2: 在对内存中开辟空间, 分配地址
- 3: 并在对象空间中, 对对象中的属性进行默认初始化。
- 4: 调用对应的构造函数进行初始化
- 5: 在构造函数中, 第一行会先到父类中构造函数进行初始化。
- 6: 父类初始化完毕后, 在对子类的属性或者方法进行显示初始化
- 7: 在进行子类构造函数的特定初始化。

初始化完毕, 将地址值付给引用变量

5. 三大特性三：多态（Polymorphism）

- 一个对象两种形态

5.1. 多态概述

多态在代码中的体现：

父类或者接口的引用指向其子类的对象。

多态的好处：

提高了代码的扩展性，前期定义的代码可以使用后期的内容。

多态的弊端：

前期定义的内容不能使用(调用)后期子类的特有内容。

多态的前提：

- 1，必须有关系，继承，实现。
- 2，要有覆盖。

5.2. 多态成员特点

1：成员变量：

编译时：参考引用型变量所属的类中是否有调用的成员变量，有，编译通过，没有，编译失败

运行时：参考引用型变量所属的类中是否有调用的成员变量，总是运行该所属类中的成员变量

简单说：编译和运行都是都参考等号的左边。

2：成员函数：

编译时：参考引用型变量所属的类中是否有调用的函数，有，编译通过，没有，编译失败

运行时：参考的是对象所属的类中是否有调用的函数。

简单说：编译看左边，运行看右边。

3：静态函数：

编译时：参考引用型变量所属的类中是否有调用静态方法

运行时：参考引用型变量所属的类中是否有调用静态方法

简单说：编译和运行都看左边。

其实对于静态方法，是不需要对象的。直接用类名调用即可。

可以说类的属性不存在多态，只有方法存在。

5.3. 向上转型

把一个子类当做父类来用是可以的，因为父类有的子类都有

把一个父类当做子类来用就不可以了，因为子类有的父类不一定有

可以定义一个父类类型的变量来记住子类对象，这在程序中称之为向上转型

多态示例：

```

abstract class Animal{
    abstract void eat();
}

class Cat extends Animal
{
    void eat()
    {    System.out.println("吃鱼"); }
    void catchMouse()
    {    System.out.println("抓老鼠");}
}

class DuoTaiDemo
{
    public static void main(String[] args)
    {
        Cat c = new Cat();    //创建一个猫类对象
        c.eat();              //调用猫的吃方法
        c.catchMouse();       //调用猫德抓老鼠方法
        Animal a = new Cat(); //自动类型提升，猫对象提升了动物类型。但特有功能无法访问。
                                //作用就是限制对特有功能的访问。
                                //专业讲：向上转型。将子类型隐藏。就不用使用子类特有方法。

        //    a.eat(); //调用被覆盖的 eat 方法。
        //    //如果还想用具体动物猫的特有功能。
        //    //你可以将该对象进行向下转型。

        //    Cat c = (Cat)a;//向下转型的目的是为了使用子类中的特有方法。
        //    c.eat();
        //    c.catchMouse();
        //    注意：对于转型，自始至终都是子类对象在做着类型的变化。
        //    Animal a1 = new Dog();
        //    Cat c1 = (Cat)a1;//ClassCastException
    }
}

```

5.4. Instanceof

返回 true 或者 false。

作用：在进行强制转换前，通常用于代码健壮性的判断，判断前一个对象是否可是后一个对象的实例是否可以转换成功。

注意：使用时，运算符前面操作数的编译时类型要么与后面的类相同，要么与后面具有子父关系，否则编译失败。

例：

```

public static void method(Animal a)//Animal a = new Dog();
{
    a.eat();
    if(a instanceof Cat)//instanceof：用于判断对象的具体类型。只能用于引用数据类型判断

```

//通常在向下转型前用于健壮性的判断。

```
{
    Cat c = (Cat)a;
    c.catchMouse();
}
else if(a instanceof Dog)
{
    Dog d = (Dog)a;
    d.lookHome();
}
else
{
}
```

5.5. final 修饰符

- 1, final 是一个修饰符，可以修饰类，方法，变量。
- 2, final 修饰的类不可以被继承。
- 3, final 修饰的方法不可以被覆盖。
- 4, final 修饰的变量是一个常量，只能赋值一次。

- i. 为什么要用 final 修饰变量。其实在程序如果一个数据是固定的，那么直接使用这个数据就可以了，但是这样阅读性差，所以它该数据起个名称。而且这个变量名称的值不能变化，所以加上 final 固定。
写法规范：常量所有字母都大写，多个单词，中间用_连接。

5.6. 内部类概述

● 什么是内部类

类是一个独立的程序单元，当我们吧一个类定义在另一个类的内部的时候，这个类就是内部类也叫嵌套类。

● 内部类有什么特点

- ✧ 提供了更好的封装，可以把内部类隐藏在外部类中，不允许同一个包中的类访问。
- ✧ 内部类可以直接访问外部类私有数据，但是外部类不能随便访问内部类的信息。要访问就要创建内部类对象
- ✧ 匿名内部类适合用于创建那些仅需要使用一次的类。

● 什么时候用内部类

分析事物时，发现该事物描述中还有事物，而且这个事物还在访问被描述事物的内容。这时就把还有的事物定义成内部类来描述。

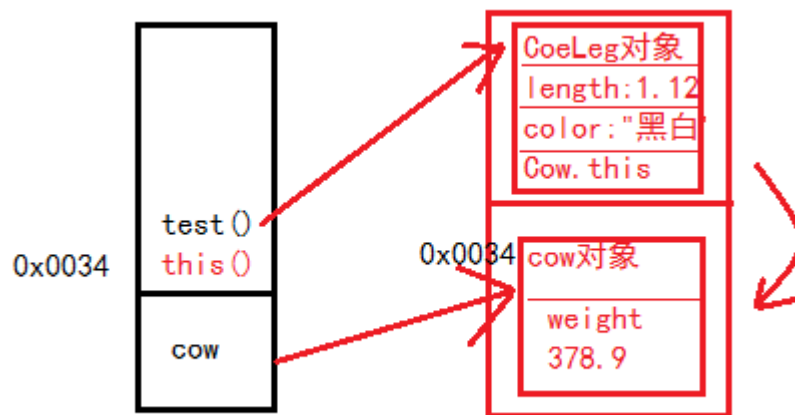
● 内部类外部类的关系

内部类对象必须寄存在外部类对象里，而外部类对象则不一定有内部类对象寄存在其中，简单说，如果存在一个内部类对象，则一定存在一个被他寄存的外部类对象，但外部类对象存在时，不一定寄存了内部类对象。因此外部类对象访问内部类对象时，可能内部类对象还不存在。

● 非静态内部类示例

运行后得到 Cow.class 文件和 Cow\$CowLeg.class，内部类总是以这种形式存在 OuterClass\$InnerClass.class

```
1) public class Cow{
2)     private double weight;
3)     public Cow() {} //外部类的两个重载的构造函数
4)     public Cow(double weight){
5)         this.weight = weight;
6)     }
7)     private class CowLeg //定义一个非静态内部类
8)     {
9)         private double length; //非静态内部类的两个Field
10)        private String color;
11)        public CowLeg() {} //非静态内部类的两个重载的构造函数
12)        public CowLeg(double length, String color) {
13)            this.length = length;
14)            this.color = color;
15)        }
16)        //省略 set get 方法
17)        public void info()//非静态内部类的实例方法
18)        {
19)            System.out.println("当前牛腿颜色是：" + color + "，高：" + length);
20)            System.out.println("本牛腿所在奶牛重：" + weight); //直接访问外部类的 private 修
    饰的Field
21)        }
22)    }
23)    public void test()//外部类方法
24)    {
25)        CowLeg cl = new CowLeg(1.12, "黑白相间");//创建内部类实例
26)        cl.info();//调用内部类实例方法
27)    }
28)    public static void main(String[] args){
29)        Cow cow = new Cow(378.9);
30)        cow.test();
31)    }
32) }
```



5.7. 内部类使用方法及注意

- 非静态内部类不可以由静态声明，否则会引起编译错误

```
class Outer {
    class Inner {
        static {
            System.out.println("inner_____");
        }
        private static int age;
    }
    public static void main(String[] args) {
        new Outer();
    }
}
```

```
D:\itcast>javac Outer.java
Outer.java:5: 内部类不能有静态声明
        static
        ^
Outer.java:9: 内部类不能有静态声明
        private static int age;
                ^
2 错误
D:\itcast>
```

- 如果内部类有和外部类同名变量，则内部类调用外部类成员用 OuterClass.this.fieldName 来访问外部类实例的 field，通过 this.fieldName 来访问内部类实例变量。

```
public class DiscernVariable
{
    private String prop = "外部类的实例变量";
    private class InClass
    {
```

```

private String prop = "内部类的实例变量";
public void info()
{
    String prop = "局部变量";
    //通过 外部类类名.this.varName 访问外部类实例 Field
    System.out.println("外部类的 Field 值: " + DiscernVariable.this.prop);
    //通过 this.varName 访问内部类实例的 Field
    System.out.println("内部类的 Field 值: " + this.prop); //直接访问局部变量
    System.out.println("局部变量的值: " + prop);
}
}
public void test()
{
    InClass in = new InClass();
    in.info();
}
public static void main(String[] args)
{
    new DiscernVariable().test();
}
}

```

- 如果在外部类以外的地方定义内部类（包括静态和非静态）：

OuterClass.InnerClass varName;

在外部类以外的地方创建内部类 OuterInstance.new InnerConstruntor ();

例：

```

class Out
{
    class In
    {
        public In(String msg)
        {
            System.out.println(msg);
        }
    }
}
public class CreateInnerInstance
{
    public static void main(String[] args)
    {
        Out.In in = new Out().new In("测试信息");
        /*
        上面代码可改为如下三行代码：
        使用 OuterClass.InnerClass 的形式定义内部类变量
        Out.In in;

```


创建外部类实例，非静态内部类实例将寄存在该实例中

```
Out out = new Out();
```

通过外部类实例和 new 来调用内部类构造函数创建非静态内部类实例

```
in = out.new In("测试信息");
```

```
*/
```

```
}
```

```
}
```

- 创建非静态内部类的子类时，必须保证让子类构造函数可以调用非静态内部类的构造函数，调用时必须存在一个外部类对象

```
public class SubClass extends Out.In
```

```
{
```

```
    public SubClass(Out out) //SubClass 的构造函数
```

```
    {
```

```
        out.super("hello");//通过传入的 Out 对象显式调用 In 的构造函数
```

```
    }
```

```
}
```

5.8. 静态内部类&局部内部类

- 静态内部类

用 static 修饰一个内部类，则这个类就属于外部类本身，而不属于外部类的某个对象。用 static 修饰的成员属于类而不属于对象。外部类的上一级单元是包，所以不能使用 static 修饰。但是内部类可以

```
class Outer
```

```
{
```

```
    private static int num = 31;
```

```
    static class Inner// 内部类。
```

```
    {
```

```
        void show()
```

```
        {
```

```
            System.out.println("show run..." + num);
```

```
        }
```

```
    }
```

```
    public void method()
```

```
    {
```

```
        Inner in = new Inner();
```

```
        in.show();
```

```
    }
```

```
}
```

```
class InnerClassDemo
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        //如果内部类是静态的。 相当于一个外部类
```

```
        Outer.Inner in = new Outer.Inner();
```

```
        in.show();
```

```
        //如果内部类是静态的，成员是静态的。
```

```
        Outer.Inner.function();
```

```
}
}
```

- 局部内部类

把一个类定义在方法里面这个类就是局部内部类。局部变量作用域只在方法区内有效，其他程序访问不到，所以不能用 `static` 修饰局部变量，包括局部类。

```
class LocalInnerClass
{
    public static void main(String[] args)
    {
        class Inner//定义局部内部类
        {
            int a;
        }
        class SubInner extends //Inner 定义局部内部类继承 Inner
        {
            int b;
        }

        SubInner s = new SubInner();
        s.a = 5;
        s.b = 6;
        System.out.println(s.a+" "+s.b);
    }
}
```

编译后出现一下三个文件：同一个类中可能有 2 个以上同名的内部类（处于不同方法中），所有内部类有编号

`LocalInnerClass.class`

`LocalInnerClass$1SubInner.class`

`LocalInnerClass$1Inner.class`

5.9. 匿名内部类

6. 错误收集

1: 编译或者运行是类名或者文件名写错误提示，文件不存在

2: 非法字符（可能输入中文字符）

2: 内部类不能有静态声明

省略慢慢收集吧

7. 异常

- 就是不正常

7.1. 异常概述

Java 中，我们用类的形式对不正常的情况进行了描述和封装，这个类就是异常类。异常问题很多，也就意味着描述的类比较多，将其共享向上抽取，形成了异常体系：Throwable。

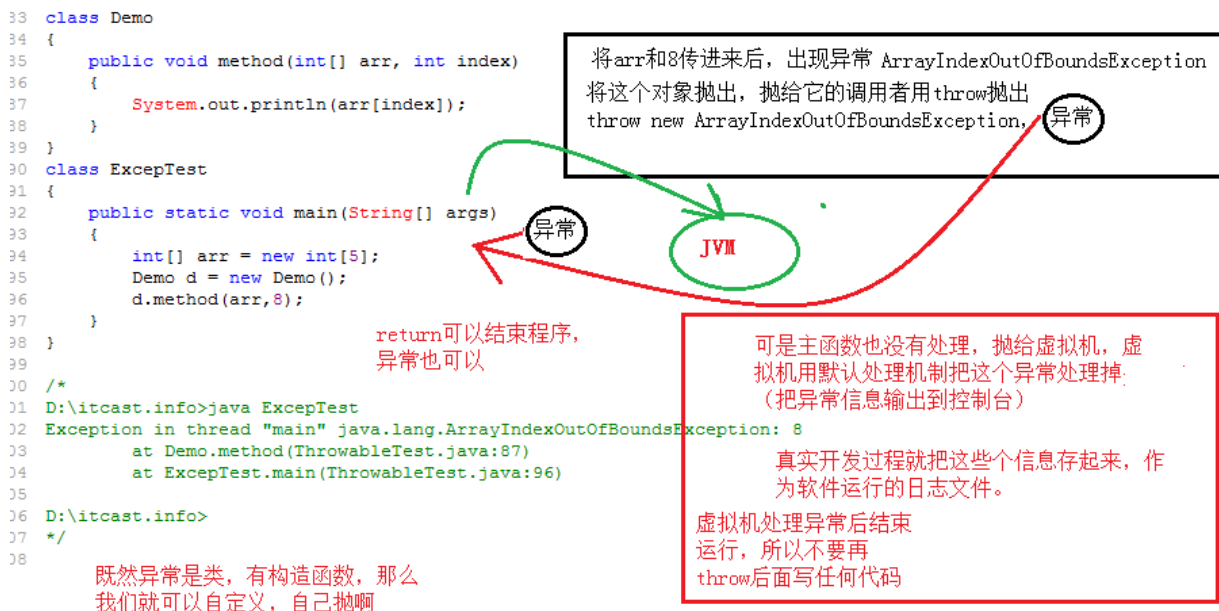
- 一般不可处理的 Error
 - 特点：由 jvm 抛出的严重性问题。这种问题发生一般不针对性处理。直接修改程序。
- 可以处理的 Exception
- 无论是 error，还是 exception，问题发生了就应该可以抛出，让调用者知道并处理。该体系的特点就在于 Throwable 及其所有的子类都具有可抛性。Throw & throws

class Demo

```
{
    public void method(int[] arr, int index) {
        System.out.println(arr[index]);
    }
}
```

class ExcepTest

```
{
    public static void main(String[] args)
    {
        int[] arr = new int[5];
        Demo d = new Demo();
        d.method(arr,8);
    }
}
```





7.2. 异常小例一:

```
class Demo
{
    public int arrOut(int[] arr, int index)
    {
        if (arr == null)
        {
            throw new NullPointerException("数组引用为空啦!! ");
        }
        if (index >= arr.length)
        {
            throw new ArrayIndexOutOfBoundsException("数组角标越界啦");
        }
        if (index < 0)
        {
            throw new ArrayIndexOutOfBoundsException("数组角标不能为负数啊");
        }
        return arr[index];
    }

    public static void main(String[] args)
    {
        int[] arr = new int[5];
        Demo d = new Demo();
        //int num = d.arrOut(null, 3); // throw new NullPointerException("数组引用为空啦!! ");
        //int num = d.arrOut(arr, 6); // throw new ArrayIndexOutOfBoundsException("数组角标越界啦");
        int num = d.arrOut(arr, 4); // 正常输出, 顺序执行
        System.out.println("num: " + num);
        System.out.println("Over");
    }
}
```

```

C:\windows\system32\cmd.exe

D:\itcast.info>java Demo
Exception in thread "main" java.lang.NullPointerException: 数组引用为空啦!!
    at Demo.arrOut(ThrowableTest.java:7)
    at Demo.main(ThrowableTest.java:25)

D:\itcast.info>javac ThrowableTest.java

D:\itcast.info>java Demo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 数组角标越界啦
    at Demo.arrOut(ThrowableTest.java:11)
    at Demo.main(ThrowableTest.java:25)

D:\itcast.info>javac ThrowableTest.java

D:\itcast.info>java Demo
num: 0
Over

D:\itcast.info>_

```

试想下:在上面的例子中 javaAPI 中并没有定义数组角标为负的异常,那么我们就需要按照 java 的异常思想,面向对象,将负数角标进行自定义描述,并封装成对象,这种自定义的问题描述,就是自定义异常。

7.3. 异常的分类

1: 编译时被检测的异常: 只要是 Exception 和其子类都是的,除了特殊子类 RuntimeException 体系。这种问题一旦出现,希望就在编译时就进行检测,让这种问题有对应的处理方式。这样的问题都可以针对性的处理。

2: 运行时异常: 就是 Exception 中的 RuntimeException 和其子类。这种问题的发生,无法让功能继续执行了,更多是因为调用者的原因导致而存活引发了内部状态的改变导致的,那么这种问题一般不予处理,直接编译通过,在运行的时候,让调用者调用时的程序强制停止,让调用者对代码进行修正。class FushuIndexException extends Exception//如果异常继承 Exception 类,则必须在函数上声明否则编译失败,

//而继承就不用声明,编译可以直接通过

RuntimeException

```

{
    FushuIndexException()
    {}
    FushuIndexException(String msg)
    {
        super(msg);//调用父类构造函数
    }
}

```

```

D:\itcast.info>javac ThrowableTest.java
ThrowableTest.java:62: 未报告的异常 FushuIndexException; 必须对其进行捕捉或声明
以便抛出
        throw new FushuIndexException<"数组角标不能为负数啊">;
        ^
1 错误
                                继承RuntimeException后编
D:\itcast.info>javac ThrowableTest.java 译通过
D:\itcast.info>_

```

- 注意：如果让一个类成为异常类，必须要继承异常体系，因为还有成为异常体系的子类才具备可抛性，才可以使用 2 个关键字 `throw` 和 `throws`（死肉，死肉丝）

7.4. throw VS throws

- 1: `throws` 使用在函数上，而 `throw` 使用在函数内
- 2: `throws` 抛出的是异常类，可以抛出多个，用逗号隔开。`Throw` 抛出的是异常对象。

修改 7.2 的代码;

- 自定义的异常不能被上一级识别，所以用 `throws` 标示在函数上
- 调用可能发生异常的函数，需要处理，或者 `throws` 或者 `catch`
- `Throws` 声明的目的是让高速调用者需要处理。

```
class FushuIndexException extends Exception
```

```

{
    FushuIndexException()
    {}
    FushuIndexException(String msg)
    {
        super(msg); //调用父类构造函数
    }
}

```

```
class Demo
```

```

{
    public int arrOut(int[] arr, int index) throws FushuIndexException //抛出的异常不能被识别，需要声明或捕捉
    {
        if (arr == null)
        {
            throw new NullPointerException("数组引用为空啦!! ");
        }
        if (index >= arr.length)
        {
            throw new ArrayIndexOutOfBoundsException("数组角标越界啦");
        }
    }
}

```

```

        if (index < 0)
        {
            throw new FushuIndexException("数组角标不能为负数啊");
        }
        return arr[index];
    }

    public static void main(String[] args) throws FushuIndexException
    {
        int[] arr = new int[5];
        Demo d = new Demo();
        //int num = d.arrOut(null, 3);
        //int num = d.arrOut(arr, 6);
        //int num = d.arrOut(arr, 4);
        int num = d.arrOut(arr, -4);
        System.out.println("num: " + num);
        System.out.println("Over");
    }
}

```

```

D:\itcast.info>javac ThrowableTest.java
ThrowableTest.java:54: 需要 ';'
        throw new NullPointerException("数组引用为空啦!!")
                ^
1 错误

D:\itcast.info>javac ThrowableTest.java
ThrowableTest.java:62: 未报告的异常 FushuIndexException; 必须对其进行捕捉或声明
以便抛出
        throw new FushuIndexException("数组角标不能为负数啊");
                ^
1 错误

D:\itcast.info>javac ThrowableTest.java
ThrowableTest.java:74: 未报告的异常 FushuIndexException; 必须对其进行捕捉或声明
以便抛出
        int num = d.arrOut(arr, -4);
                ^
1 错误

D:\itcast.info>javac ThrowableTest.java
D:\itcast.info>_

```

7.5. 异常处理机制

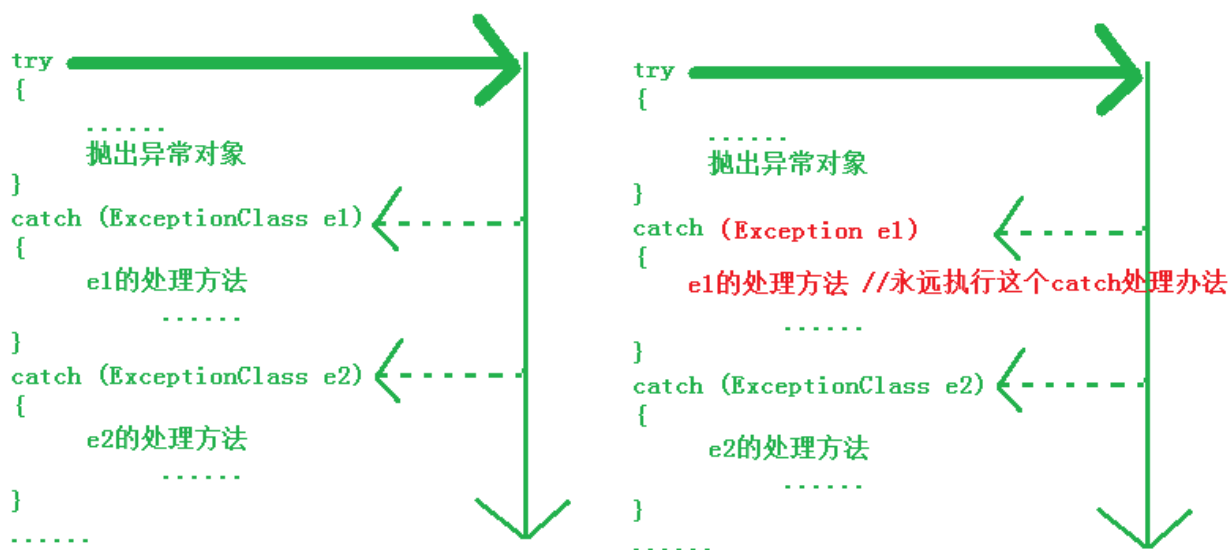
Java 的异常处理机制可以让程序具有良好的容错性，让程序更加健壮。让程序出现异常时，系统会自动生成一个 `Exception` 对象来告知程序，从而实现将业务功能代码和错误处理代码分离，提供良好的可读性。

Java 用 `try ..catch` 来处理异常，格式：

```
try
{ 业务代码
}
catch (异常类 变量)//用于接收发生的异常对象
{ 异常处理代码
}
Finally//通常用于关闭资源
{一定会处理的代码
}
```

执行 `try` 里面的代码出现异常生成异常对象，这个对象被抛出 `throw`，当 java 运行环境收到这个对象后，把这个对象交给 `catch` 处理，如果找不到 `catch` 块，则运行环境终止，java 程序也终止。

- 如果 `try` 被执行一次则后面只能有一个 `catch` 被执行
- 多 `catch` 中，如果存在父类，则父类一定放在后面（先处理小异常）
- 抛异常了才 `try`，没 `throws` 异常 `try` 是没用的



7.6. 什么时候 try 什么时候 throw

```
public int arrOut(int[] arr, int index)//输出函数//声明
{
    try
    {
        if (index < 0)
            throw new FushuIndexException("数组角标不能为负数啊");
    }
}
```



```

    }
    catch{}
    return arr[index];
}

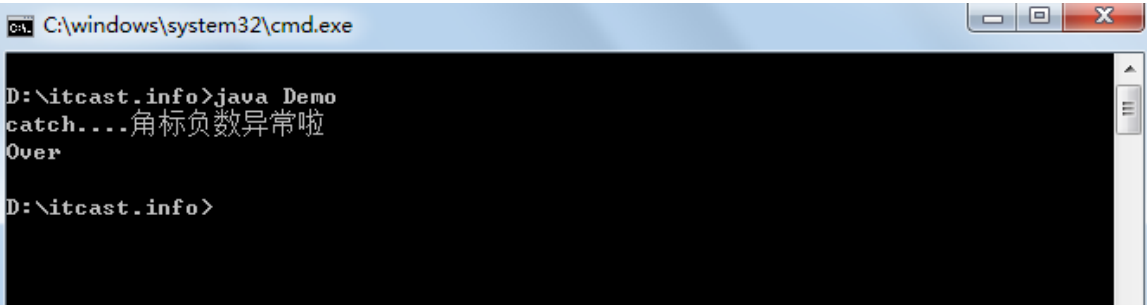
```

如果调用者传递一个负数角标，发生异常，我们需要 try 吗？我们 try 的话，arr 还是没有值，在内部没有解决这个问题。这个问题是调用者的问题，他传值错误，那是他的问题，所以我们声明出去

```

public static void main(String[] args)  {
    int[] arr = new int[5];
    Demo d = new Demo();
    try  {
        int num = d.arrOut(arr, -4);
        System.out.println("num: " + num);
    }
    catch(FuShuIndexException e)//处理要有针对性
    {
        System.out.println("数组角标越界")
    }
    System.out.println("Over");
}

```



在调用的时候又出现问题，我们能处理吗？我们的原则就是能处理就 try 处理不了就抛。
上面的 d.arrOut 是调用者，我们需要告诉调用者，

7.7. 访问异常信息

String	getMessage() 返回此 throwable 的详细消息字符串。
void	printStackTrace() 将此 throwable 及其追踪输出至标准错误流。

```

public static void main(String[] args)throws FushuIndexException
{
    int[] arr = new int[5];
    Demo d = new Demo();
    try
    {

```

```

        int num = d.arrOut(arr, -4);
        System.out.println("num: " + num);
    }
    catch (FushuIndexException e)
    {
        System.out.println("message: " + e.getMessage());
        System.out.println("message: " + e);//将对象变字符串打印，此处不是 e 的哈希值，异常有自己独特的方法
        e.printStackTrace();//JVM 默认的异常处理机制就是这个
        System.out.println("catch....角标负数异常啦");
    }

    System.out.println("Over");
}

```

```

C:\windows\system32\cmd.exe

D:\itcast.info>javac ThrowableTest.java

D:\itcast.info>java Demo
message: 自定义的数组角标不能为负数啊
message: FushuIndexException: 自定义的数组角标不能为负数啊
FushuIndexException: 自定义的数组角标不能为负数啊
    at Demo.arrOut<ThrowableTest.java:54>
    at Demo.main<ThrowableTest.java:68>
catch....角标负数异常啦
Over

D:\itcast.info>

```

7.8. 异常处理原则

- 函数内容如果抛出需要检测的异常，那么函数上必须要声明，否则必须在函数内部 trycatch 捕捉否则编译时失败
- 如果调用了声明异常的函数，那么要么 trycatch 要么 throws，否则编译失败
- 什么时候 catch，什么时候 throws?：功能内部可以解决用 catch，解决不了用 throws 告诉调用者，有调用者处理
- 一个功能如果抛出了多个异常，那么调用时，就必须对应多个 catch 进行捕捉，内部有几个需要检测的异常，就抛几个异常，抛出几个异常就 catch 几个
-

7.9. try catch final 组合特点

- try catch finally
- try catch（多个）当没有必要资源需要释放时，可以不用定义 finally
- try finally

void show()//没有声明就得 try

```
{
    Try{
        throw new Exception();}
        catch(Exception e){}
    }
}
```

7.10. 异常处理中的 Finally

```
void show()throws Exception
```

```
{
    Try{
        开启资源;
        throw new Exception();}
        finally{
            关闭资源; }
    }
}
```

主要用于回收资源，（如数据库连接，网络连接，磁盘文件等。）这些物力资源都必须显示回收。
除非在 trycatch 中调用了退出虚拟机的方法，否则，不管在 try、catch 中执行怎么样的代码，异常处理的 finally 都会被执行。

8. 线程

- 一个程序执行时，内部可能包含了多个顺序执行流，每个顺序执行流就是一个线程

8.1. 进程与线程

进程：（直译：正在进行中的程序。）操作系统可以同时运行多个任务，一个任务通常就是一个程序，每个运行中的程序就是一个进程。

线程：就是进程中一个负责程序执行的控制单元（执行路径）。一个进程中可以多执行路径，成为多线程。

扩展：一个进程中至少有一个线程；

进程特点：

独立性：进程是系统中独立存在的实体，每一个进程都有自己的地址空间。

动态性：进程有自己的生命周期和不同的状态。

并发性：多个进程可以再当个处理器上并发执行，多个进程之间不影响。

8.2. 多线程的利与弊

每一个线程都有自己运行的内容。这个内容可以成为线程要执行的任务。开启多个线程是**为了同时运行多部分代码**。

多线程好处：解决了多部分代码同时运行的问题

多线程弊端：线程太多导致效率的降低。

Run 方法：封装任务代码。

8.3. JVM 启动时的多线程

其实啊，JVM 启动的时候就启动了多个线程，至少有 2 个线程可以分析出来。

任何线程启动都有自己的任务，

1：执行 `main` 函数的线程

该线程的任务代码都定义在 `main` 函数中。

创建线程的目的是**为了新开启一条执行路径**，来执行指定的代码和其他代码实现同时运行。而运行的指定代码就是这个执行路径的任务。

jvm 创建的主线程的任务都定义在了主函数 `main` 中。

线程是需要任务的，`Thread` 类用于描述线程。所以 `Thread` 类也是对任务的描述。而这个任务就通过 `Thread`

类中的 **run 方法来体现**。也就是说，`run` 方法就是封装自定义线程运行任务的函数。

`run` 方法中定义就是线程要运行的任务代码。

开启线程是为了运行指定代码，所以创建线程必须继承 `Thread` 类，并复写 `run` 方法。

将运行的代码定义在 `run` 方法中即可。

2：负责垃圾回收的线程。

该线程的任务代码都在垃圾回收器中呢，在底层呢

`class Demo extends Object` //继承 `object` 类

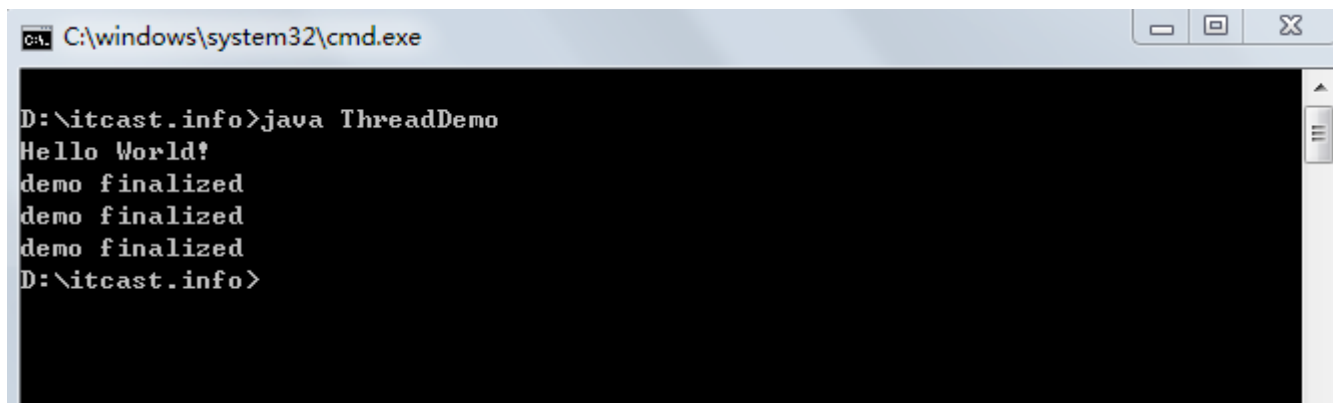
```
{
    public void finalize(){ //复写 finalize 方法
        System.out.println("demo finalized");
    }
}
```

`class ThreadDemo`

```
{
    public static void main(String[] args) {
        new Demo();
        new Demo();
        new Demo();
        System.gc();//上面 new 的对象，不定时回收，所以为了演示效果调用 System 的 gc（）函数，运行垃圾回收期
        System.out.println("Hello World!");
    }
}
```

```
}
```

下面只是输出结果其一，也有可能 `hello World` 在后面，也有可能 `dome finalized` 在前面，也与可能 `demo finalized` 只输出一次，因为回收线程可能还没有执行完毕，虚拟机就运行完了，之后会强制回收的。



```
C:\windows\system32\cmd.exe
D:\itcast.info>java ThreadDemo
Hello World!
demo finalized
demo finalized
demo finalized
D:\itcast.info>
```

8.4. 创建线程两种方式 and 区别

创建线程方式一：继承 `Thread` 类。

步骤： 定义一个类，继承 `Thread` 类。

覆盖 `Thread` 类中的 `run` 方法。

直接创建 `Thread` 的子类对象，创建线程。

调用 `start` 方法开启线程并调用线程的任务 `run` 方法执行。

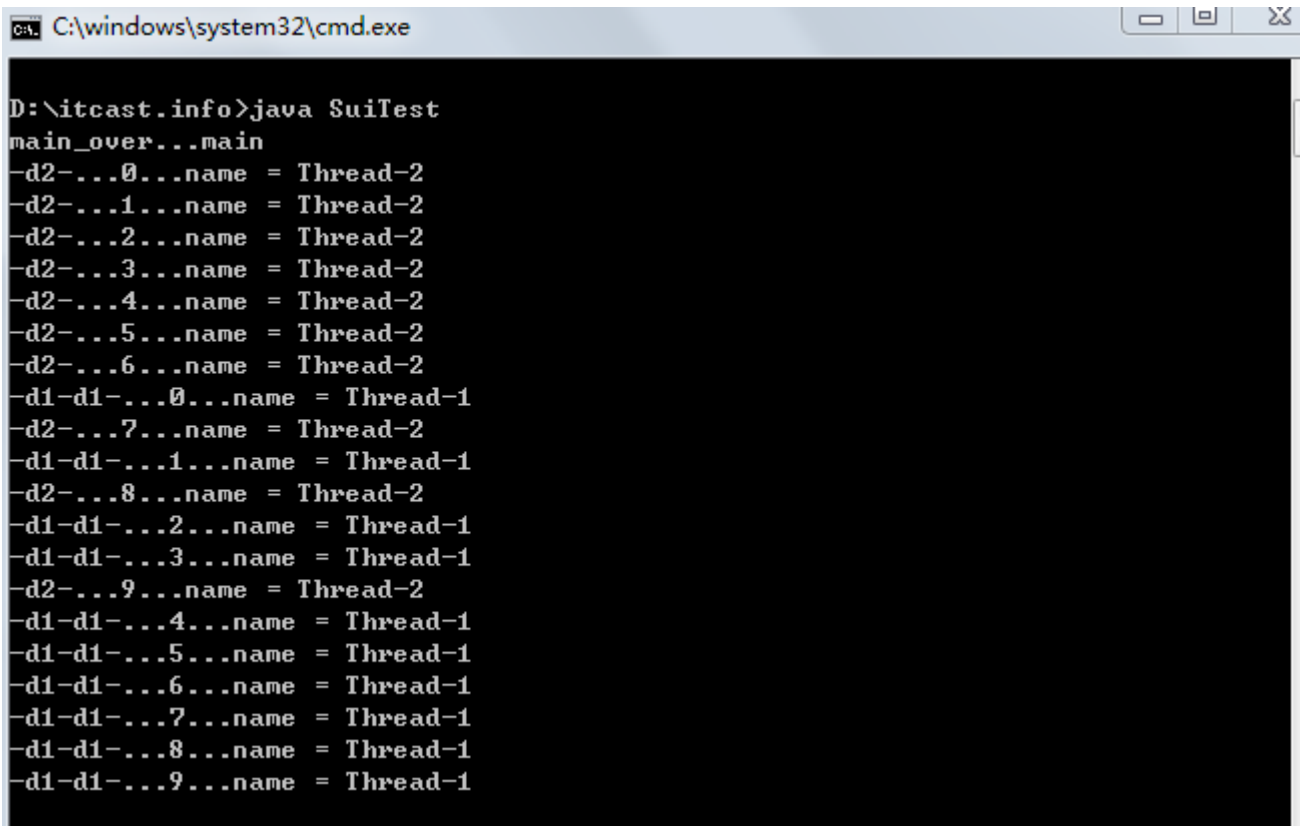
`class Demo extends Thread` // 继承 `Thread` 类

```
{
    private String name;
    public Demo(String name)
    {
        this.name = name;
    }
    public void run() // 复写 Thread 的 run 方法，开发过程中 run 方法中定义就是线程要运行的任务代码。
    {
        for (int i = 0; i < 10; i++)
        {
            for (int j = 10; j < 50000; j++) // 为了看看效果，多加一个空体的循环，不至于进程太快执行完毕
            {}
            System.out.println(name + "..." + i + "...name = " + Thread.currentThread().getName());
        }
    }
}

class SuiTest
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(); // 创建线程
    }
}
```

```
        Demo d1 = new Demo("-d1-d1-");
        Demo d2 = new Demo("-d2-");
        d1.start();//开启线程
        d2.start();
        System.out.println("main_over..." + Thread.currentThread().getName());
    }
}
/*
可以通过 Thread 的 getName 获取线程的名称 Thread-编号(从 0 开始)
Main 就是主线程。
*/
```

运行结果:



```
C:\windows\system32\cmd.exe

D:\itcast.info>java SuiTest
main_over...main
-d2-...0...name = Thread-2
-d2-...1...name = Thread-2
-d2-...2...name = Thread-2
-d2-...3...name = Thread-2
-d2-...4...name = Thread-2
-d2-...5...name = Thread-2
-d2-...6...name = Thread-2
-d1-d1-...0...name = Thread-1
-d2-...7...name = Thread-2
-d1-d1-...1...name = Thread-1
-d2-...8...name = Thread-2
-d1-d1-...2...name = Thread-1
-d1-d1-...3...name = Thread-1
-d2-...9...name = Thread-2
-d1-d1-...4...name = Thread-1
-d1-d1-...5...name = Thread-1
-d1-d1-...6...name = Thread-1
-d1-d1-...7...name = Thread-1
-d1-d1-...8...name = Thread-1
-d1-d1-...9...name = Thread-1
```

8.5. start 方法和 run 方法区别

9. 后续

下次整理的会陆续发上来，大家敬请期待