

Java程序设计



第5章 深入理解JAVA语言



第5章 深入理解JAVA语言

- 本章介绍Java语言中的一些机制及细节。
- 5.1 变量及其传递
- 5.2 多态和虚方法调用
- 5.3 对象构造与初始化
- 5.4 对象清除与垃圾回收
- 5.5 内部类与匿名类
- 5.6 Lambda表达式
- 5.7 装箱、枚举、注解
- 5.8 没有指针的Java语言



变量及其传递

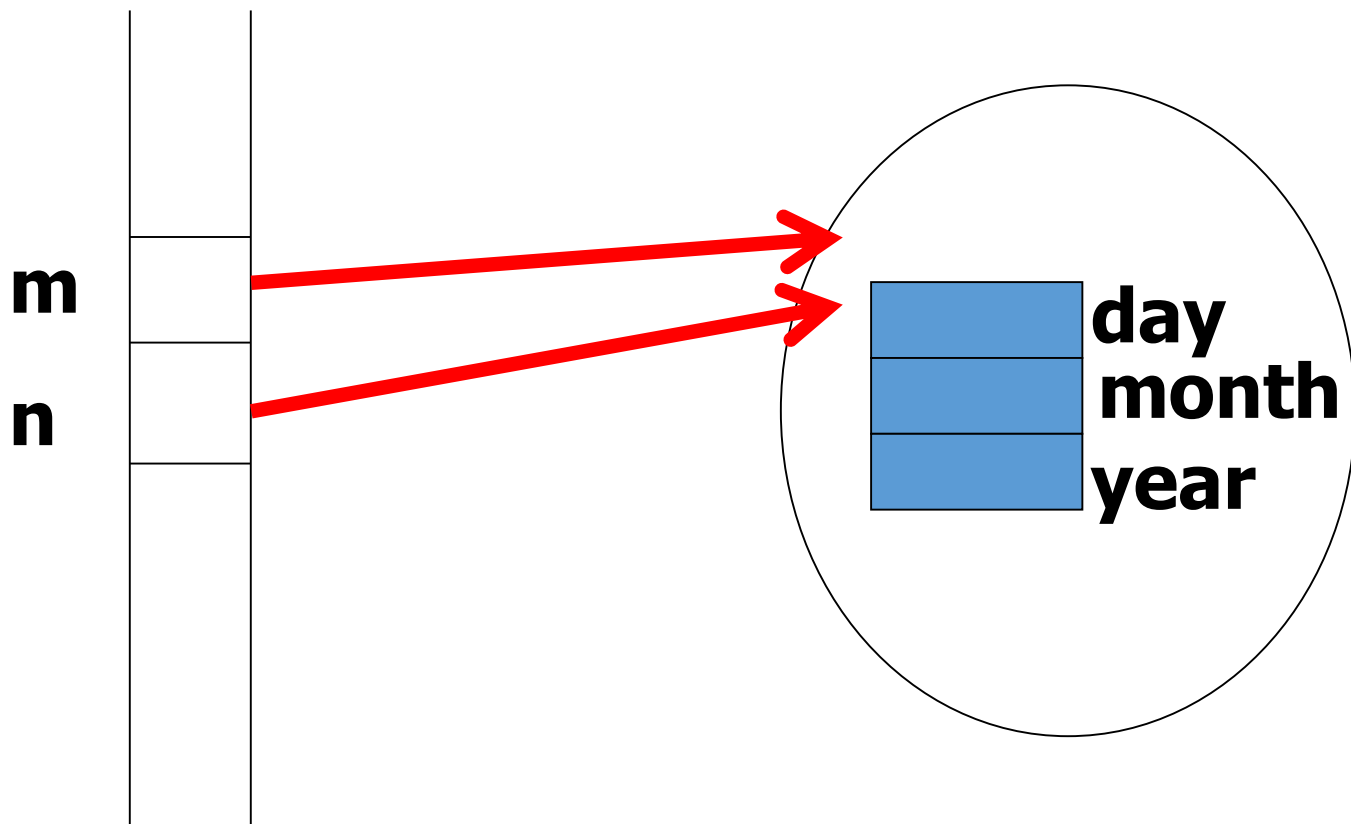




基本类型变量与引用型变量

- 基本类型(primitive type)：其值直接存于变量中。 “**在这里**”
- 引用型(reference type) 的变量除占据一定的内存空间外，它所引用的对象实体（由new 创建）也要占据一定空间。 “**在那里**”

引用型变量与对象实体的关系





- 示例：MyDate.java
- MyDate m,n;
- m=new MyDate();
- n=m;
- n.addYear();



字段变量与局部变量

- 字段变量 (field)与局部变量(Local variable)
 - 前者是在类中，后者是方法中定义的变量或方法的参变量
- 从内存角度看
 - 存储位置**，字段变量为对象的一部分、存在于**堆**中的，局部变量是存在于**栈**中。
 - 生命周期**不同
 - 初始值**：字段变量可以**自动赋初值**，局部变量则须显式赋值

```
class Test()  
{  
    int a;  
    void m(){  
        int b;  
        System.out.println(b);//编译不能通过需要//初始化。  
    }  
}
```




两种变量的区别

- 从语法角度看
 - 字段变量属于类，可以用public,private,static,final 修饰。
 - 局部变量不能够被访问控制符及static修饰
 - 都可以被final修饰



- 调用对象方法时，要传递参数。在传递参数时，
- Java 是值传递，即，是将表达式的值复制给形式参数。
- 对于引用型变量，传递的值是引用值，而不是复制对象实体
 - 可以改变对象的属性
- [TransByValue.java](#)



变量的返回

- 方法的返回：
 - 返回基本类型。
 - 返回引用类型。它就可以存取对象实体。
- `Object getNewObject()`
- `{`
- `Object obj=new Object();`
- `return obj;`
- `}`
- 调用时：`Object p= GetNewObject();`



补充：不定长参数

- 不定长参数 (Variable length arguments) , 从JDK1.5开始
- 用省略号表示, 并且是最后一个参数
- 实际上Java当成一个数组
- ```
int sum(int ... nums){
 □ int s=0;
 □ for(int n : nums) s+=n;
 □ return s;
}
```
- 调用 : `sum(1,2,3,4);`
- 又例如 : `public static void main( String...argv)`



# 多态和虚方法调用





- 多态(Polymorphism)是指一个程序中相同的名字表示不同的含义的情况。
- 多态有两种情形
  - 编译时多态：
    - 重载(overload) ( 多个同名的不同方法 )。
    - 如 `p.sayHello();` `p.sayHello( "Wang" );`
  - 运行时多态：
    - 覆盖(override) ( 子类对父类方法进行覆盖 )
    - 动态绑定 ( dynamic binding ) ----虚方法调用(virtual method invoking)
    - 在调用方法时，程序会正确地调用子类对象的方法。
- 多态的特点大大提高了程序的抽象程度和简洁性



- 上溯造型 ( upcasting )
  - ▣是把派生类型当作基本类型处理
- `Person p = new Student();`
- `void fun(Person p ){...}`      `fun(new Person());`





- [TestVirtualInvoke.java](#)
- 用虚方法调用，可以实现运行时的多态！
  - 子类重载了父类方法时，运行时
  - 运行时系统根据调用该方法的**实例的类型**来决定选择哪个方法调用
  - **所有的**非final方法都会自动地进行**动态绑定**！





# 虚方法调用示例

- 示例 TestVirtualInvoke.java
- `void doStuff(Shape s){`
- `s.draw();`
- `}`
- `Circle c = new Circle();`
- `Triangle t = new Triangle();`
- `Line l = new Line();`
- `doStuff(c);`
- `doStuff(t);`
- `doStuff(l);`

```
Draw Circle
Draw Three Lines
Draw Line
```



# 动态类型确定

- 变量 `instanceof` 类型
- 结果是boolean 值
- 示例 [InstanceOf.java](#)



# 什么情况不是虚方法调用

- Java中，普通的方法是虚方法
- 但static,private方法不是虚方法调用
- static,private与虚方法编译后用的指令是不同的

- 演示：JavaP3methods

```
8: aload_1
9: invokevirtual #4 // Method f:()V
12: aload_1
13: invokespecial #5 // Method p:()V
16: aload_1
17: pop
18: invokestatic #6 // Method s:()V
```



# 三种非虚的方法

- static的方法，以声明的类型为准，与实例类型无关
- private方法子类看不见，也不会被虚化
- final方法子类不能覆盖，不存在虚化问题
- 对比 [TestStaticInvoke.java](#)



# 对象构造与初始化





# 构造方法

- 构造方法 ( constructor)
  - 对象都有构造方法
  - 如果没有，编译器加一个 **default** 构造方法
- 思考：
  - 抽象类 ( abstract ) 有没有构造方法？





# 调用本类或父类的构造方法

- 调用本类或父类的构造方法
  - this调用本类的其他构造方法。
  - super调用直接父类的构造方法
  - this或super要放在第一条语句,且只能够有一条
- 如果没有this及super, 则编译器自动加上super(), 即调用直接父类不带参数的构造方法
- 因为必须令所有父类的构造方法都得到调用, 否则整个对象的构建就可能不正确。





- 在构造函数中使用this和super.
- [ConstructCallThisAndSuper.java](#)

# 一个问题



- `class A`
- `{`
- `A(int a){}`
- `}`
- `class B extends A`
- `{`
- `B(String s){} //编译不能够通过.`
- `}`
- 编译器会自动调用`B(String s){ super();}` 出错.
- 解决方法:
  - ▣ 在B的构造方法中,加入`super(3);`
  - ▣ 在A中加入一个不带参数的构造方法,`A(){}`
  - ▣ 去掉A中全部的构造方法,则编译器会自动加入一个不带参数的构造方法,称为默认的构造方法.



# 创建对象时初始化

- `p = new Person(){ { age=18; name="李明"; } };`
- 这样可以针对没有相应构造函数，但又要赋值
- 注意双括号



# 实例初始化与静态初始化

- 实例初始化 ( Instance Initializers )
- 在类中直接写
  - { 语句.... }
  - 实例初始化，先于构造方法{}中的语句执行
- 静态初始化 ( Static Initializers )
  - static { 语句.... }
  - 静态初始化，在第一次使用这个类时要执行，
  - 但其执行的具体时机是不确定的
    - 但是可以肯定的是：**总是先于实例的初始化**
  - 例：[InitialTest.java](#)



# 构造方法的执行过程

- 构造方法的执行过程遵照以下步骤：
  - 调用本类或父类的构造方法，直至最高一层（Object）
  - 按照声明顺序执行字段的初始化赋值
  - 执行构造函数中的各语句
- 简单地说：
  - 先父类构造，再本类成员赋值，最后执行构造方法中的语句。
- 演示：JavaPConstructor



## ConstructSequence.java

```
开始构造Person(),此时this.name=未命名,this.age=-1
Person()构造完成,此时this.name=黎明,this.age=18
开始构造Student(),此时this.name=黎明,this.age=18,this.school=未定学校
Student()构造完成,此时this.name=黎明,this.age=18,this.school=北大
```



# 一个问题



- 构造方法内部调用别的方法
- 如果这个方法是虚方法，结果如何？
  - 从语法上来说这是合法的，但有时会造成事实上的不合理
- [ConstructInvokeVirtual.java](#)
  - A Student, name:Li Ming, age: 18, school: null





# 示例中的问题

- 在本例中，在构造方法中调用了一个动态绑定的方法sayHello(),这时，会使用那个方法被覆盖的定义，而这时对象尚未完全构建好，所以School还没有赋值。
- 在构造方法中尽量避免调用任何方法，尽可能简单地使对象进入就绪状态
- 惟一能够安全调用的是final的方法。



# 对象清除与垃圾回收

A horizontal line of white dots spanning the width of the slide, positioned below the title.



- 我们知道：new创建对象
- 那么如何销毁对象？
- Java中是自动清除
  - 不需要使用delete



# 对象的自动清除

- 垃圾回收(garbage collection )
- 对象回收是由 Java虚拟机的垃圾回收线程来完成的。
- 为什么系统知道对象是否为垃圾
  - 任何对象都有一个引用计数器，当其值为0时，说明该对象可以回收。



# 引用计数示意

- String method(){
- String a,b;
- a=new String("hello world");
- b=new String("game over");
- System.out.println(a+b+"Ok");
- a=null;
- a=b;
- return a;
- }

# System.gc()方法



- System.gc()方法
- 它是System类的static方法
- 它可以要求系统进行垃圾回收
- 但它仅仅只是”建议(suggest)”



# finalize()方法



- Java中没有 “析构方法(destructor)”
- 但Object的finalize() 有类似功能
  - 系统在回收时会自动调用对象的finalize() 方法。
  - protected void finalize() throws Throwable{}
- 子类的finalize()方法
  - 可以在子类的finalize()方法释放系统资源
  - 一般来说，子类的finalize()方法中应该调用父类的finalize()方法，以保证父类的清理工作能够正常进行。
-





# try-with-resources

- 由于finalize()方法的调用时机并不确定，所以一般不用finalize()
- 关闭打开的文件、清除一些非内存资源等工作需要进行处理
- 可以使用try-with-resources语句 ( JDK1.7以上 )
- 对于实现了java.lang.AutoCloseable的对象
- `try( Scanner scanner= new Scanner( ... ) ){`
  - ▣ . . . . .
- `}`
- 会自动调用其close()方法,相当于
- `finally{`
  - ▣ `Scanner.close();`
- `}`



# 内部类与匿名类

A horizontal line composed of white dots, spanning the width of the slide, positioned below the title.



# 内部类与匿名类

- 内部类( inner class )是在其他类中的类
- 匿名类( anonymous class)是一种特殊的内部类，它没有类名。



# 内部类 ( Inner class)

- 内部类的定义

- 将类的定义 `class xxxx{...}` 置入一个类的内部即可
- 编译器生成 `xxxx$xxxx` 这样的class文件
- 内部类不能够与外部类同名

- 内部类的使用

- 在封装它的类的内部使用内部类，与普通类的使用方式相同
- 在其他地方使用
  - 类名前要冠以外部类的名字。
  - 在用new创建内部类实例时，也要在 new 前面冠以对象变量。
    - 外部对象名.new 内部类名(参数)

- 示例 TestInnerClass.java



# 在内部类中使用外部类的成员

- 内部类中可以直接访问外部类的字段及方法
  - ▣ 即使private也可以
- 如果内部类中有与外部类同名的字段或方法，则可以用
  - ▣ 外部类名.this.字段及方法
- [TestInnerThis.java](#)



# 内部类的修饰符

- 内部类与类中的字段、方法一样是外部类的成员，它的前面也可以有访问控制符和其他修饰符。
  - 访问控制符：public, protected, 默认及 private。
    - 注：外部类只能使用 public 修饰或者默认
  - final, abstract





- 用static修饰内部类 表明该内部类**实际是一种外部类**
  - 因为它与外部类的实例无关
  - 有人认为static的类是嵌套类（ nested class ），不是内部类inner class
- static类在使用时：
  - 1、实例化static类时，在 new前面**不需要用对象实例变量**；
  - 2、static类中**不能**访问其外部类的**非static**的字段及方法，既只能够访问static成员。
  - 3、static方法中不能访问非static的域及方法，也不能够不带前缀地new 一个非static的内部类。
- 示例 TestInnerStatic.java



- 在一个方法中也可以定义类，这种类称为“方法中的内部类”
- 或者叫**局部类**（ local class ）
- 示例 [TestInnerMethod.java](#)



# 使用局部类

- 1、同局部变量一样，方法中的内部类
  - 不能够用 `public, private, protected, static` 修饰，
  - 但可以被 `final` 或者 `abstract` 修饰。
- 2、可以访问其外部类的成员
- 3、不能够访问该方法的局部变量，除非是 `final` 局部变量。



- 匿名类( anonymous class)是一种特殊的内部类
  - 它没有类名，在定义类的同时就生成该对象的一个实例
  - “一次性使用” 的类
- 示例 TestInnerAnonymous.java



# 匿名类的使用

- 1、**不取名字**，直接用其**父类或接口的名字**。
  - 也就是说，该类是父类的子类，或者实现了一个接口
  - 编译器生成 `xxxxxx$1`之类的名字
- 2、类的定义的同时就创建实例，即类的定义**前面有一个new**
  - `new 类名或接口名 ( ) {.....}`
  - 不使用关键词`class`，也不使用`extends`及`implements`。
- 3、在构造对象时使用父类构造方法
  - 不能够定义构造方法，因为它没有名字
  - 如果`new`对象时，要带参数，则使用父类的构造方法



- 用到界面的事件处理

- 注册一个事件侦听器

- 示例 AutoScore.java 中

- `//SymAction lSymAction = new SymAction();`
    - `//btnNew.addActionListener(lSymAction);`
    - `btnNew.addActionListener(new ActionListener(){`
    - `public void actionPerformed(ActionEvent event)`
    - `{`
    - `btnNew_ActionPerformed(event);`
    - `}`
    - `});`





- 作为方法的参数

- ▣ 排序，给一个比较大小的接口

- ▣ 如 SortTest.java

- Arrays.<Book>sort( books, new Comparator<Book>(){
    - public int compare(Book b1, Book b2){
    - return b1.getPrice()-b2.getPrice();
    - }
    - });



# lambda表达式





- Lambda表达式是从Java8增加的新语法
- Lambda表达式 (  $\lambda$  expression)的基本写法
  - (参数) -> 结果
  - 如 (String s) -> s.length()
  - 如 x -> x\*x
  - 如 () -> { System.out.println("aaa"); }
- 大体上相当于其他语言的 “匿名函数” 或 “函数指针”
- 在Java中它实际上是 “匿名类的一个实例”

# 起因 (参见 LambdaRunnable.java)



```
Runnable dolt = new Runnable(){
 public void run(){
 System.out.println("aaa");
 }
};
new Thread(dolt).start();
```



```
Runnable doIt = () ->System.out.println("aaa");
new Thread(doIt).start();
```



```
new Thread(() ->System.out.println("aaa")).start();
```



# 可以看出

- Lambda表达式是**接口**或者说是**接口函数**的简写
- 其基本写法是 参数->结果
- 这里，参数是()或1个参数或(多个参数)
- 结果是指 表达式 或 语句 或 {语句}



# 示例：积分 LambdaIntegral.java

```
interface Fun { double fun(double x);}
```

```
double d = Integral(new Fun(){
 public double fun(double x){
 return Math.sin(x);
 }
}, 0, Math.PI, 1e-5);
```



```
double d = Integral(x->Math.sin(x) ,
 0, Math.PI, 1e-5);
```





- Lambda大大地简化了书写
- 在线程的例子中
  - ▣ `new Thread( ()->{ ... } ).start();`
- 在积分的例子中
  - ▣ `d = Integral( x->Math.sin(x), 0, 1, EPS );`
  - ▣ `d = Integral( x->x*x, 0, 1, EPS );`
  - ▣ `d = Integral( x->1, 0, 1, EPS );`
- 在按钮事件处理中
  - ▣ `btn.addActionListener( e->{ ... } ) );`

# 能写成Lambda的接口的条件



- 由于Lambda只能表示一个函数，所以
- 能写成Lambda的**接口**要求包含且最多只能有一个**抽象函数**
- 这样的接口可以（但不强求）用注记
  - `@FunctionalInterface` 来表示。称为**函数式接口**
- 如
  - `@FunctionalInterface`
  - `interface Fun { double fun( double x );}`



# 再举一例：排序

```
Comparator<Person> compareAge =
 (p1, p2) -> p1.age-p2.age;
Arrays.sort(people, compareAge);
```

```
Arrays.sort(people,
 (p1, p2) -> p1.age-p2.age);
Arrays.sort(people,
 (p1, p2) -> (int)(p1.score-p2.score));
Arrays.sort(people,
 (p1, p2) -> p1.name.compareTo(p2.name));
Arrays.sort(people,
 (p1, p2) -> -p1.name.compareTo(p2.name));
```




# 由此可见

- Lambda表达式，不仅仅是简写了代码，
- **更重要**的是：
- 它将**代码也当成数据**来处理



# 装箱、枚举、注解

A horizontal line composed of white dots, spanning the width of the slide, positioned below the title.

# 新的语法



- 从JDK1.5起，增加了一些新的语法
- 大部分是编译器自动翻译的，称为Compiler sugar





- 基本类型的包装类
  - 它将基本类型 ( primitive type) 包装成Object(引用类型 )
  - 如int → Integer
  - 共8类：
    - Boolean, Byte, Short, Character, Integer, Long, Float, Double
- Integer I = new Integer(10);

# 装箱与拆箱



- 装箱 ( Boxing )      `Integer I = 10;`
- 拆箱 ( Unboxing )    `int i = I;`
- 实际译为
  - ▣ `Integer I= Integer.valueOf(10);`
  - ▣ `int i = I.intValue();`
- 主要方便用于集合中 , 如 :
- `Object [] ary = { 1, "aaa"};`



- 枚举(enum)是一种特殊的class类型
- 在简单的情况下，用法与其他语言的enum相似
  - `enum Light { Red, Yellow, Green };`
  - `Light light = Light.Red;`
- 但实际上，它生成了 `class Light extends java.lang.Enum`



- 可以在enum定义体中，添加字段、方法、构造方法

- enum Direction

- {
  - EAST("东",1), SOUTH("南",2),
  - WEST("西",3), NORTH("北",4);
  - private Direction(String desc, int num){
  - this.desc=desc; this.num=num;
  - }
  - private String desc;
  - private int num;
  - public String getDesc(){ return desc; }
  - public int getNum(){ return num; }
- }



- 注解 ( annotation )
  - 又称为注记、标记、标注、注释 ( 不同于comments)
  - 是在各种语法要素上加上附加信息, 以供编译器或其他程序使用
- 所有的注解都是 `java.lang.annotation. Annotation` 的子类



# 常用的注解

- 常用的注解，如
  - `@Override` 表示覆盖父类的方法
  - `@Deprecated` 表示过时的方法
  - `@SuppressWarnings` 表示让编译器不产生警告
- 自定义注解，比较复杂
  - `public @interface Author {`
  - `String name();`
  - `}`
- 请参见教材





# 没有指针的Java语言





# 引用与指针

- 引用 ( reference ) 实质就是指针 ( pointer )
- 但是它是受控的、安全的
- 比如
  - 会检查空指引
  - 没有指针运算  $*(p+5)$
  - 不能访问没有引用到的内存
  - 自动回收垃圾



# C语言指针在Java中的体现

- (1)传地址 → 对象
  - 引用类型，引用本身就相当于指针
    - 可以用来修改对象的属性、调用对象的方法
  - 基本类型：没用对应的
    - 如交换两个整数
    - `void swap(int x, int y){ int t=x; x=y; y=t; }`
    - `int a=8, b=9; swap(a,b);`
    - 一种变通的办法，传出一个有两个分量x,y的对象



# C语言指针在Java中的体现

- (2)指针运算 → 数组
  - $*(p+5)$  则可以用 `args[5]`
- (3)函数指针 → 接口、Lambda表达式
  - 例：求积分，线程、回调函数、事件处理
  - `Integral.java`



# C语言指针在Java中的体现

- (4)指向结点的指针 → 对象的引用

- class Node {

- Object data;
    - Node **next**;

- }

- 例List.java实现链表

- (5)使用JNI

- Java Native Interface(JNI)

- 它允许Java代码和其他语言写的代码进行交互



# 相等还是不等

- ==
- 简单地说，基本类型是值相等，引用类型是引用相等
- 但有不少的具体情况具体分析：





# 基本类型的相等

- 基本类型
  - 数值类型：转换后比较
  - 浮点数，最好不直接用==
  - Double.NAN==Double.NAN 结果为false
    - 请参见JDK的API文档
  - boolean型无法与int相比较



- Integer i = new Integer(10);
- Integer j = new Integer(10);
- System.out.println(i==j); //false , 因为对象是两个
  
- Integer m = 10;
- Integer n = 10;
- System.out.println(m==n); //true , 因为对象有缓存
  
- Integer p = 200;
- Integer q = 200;
- System.out.println(p==q); //false , 因为对象是两个

# 装箱对象是否相等



## □注意缓存

- If the value `p` being boxed is `true`, `false`, a byte, or a char in the range `\u0000` to `\u007f`, or an int or short number between -128 and 127 (inclusive), then let `r1` and `r2` be the results of any two boxing conversions of `p`. It is always the case that `r1 == r2`.



# 枚举、引用对象是否相等

- 枚举类型

- 内部进行了惟一实例化，所以可以直接判断

- 引用对象

- 是直接看两个引用是否一样

- 如果要判断内容是否一样，则要重写equals方法

- 如果重写equals方法，则最好重写 hashCode()方法

# String对象的特殊性



- String对象

- 判断相等，一定不要用==，要用equals
- 但是字符串常量（String literal）及字符串常量会进行内部化（interned），相同的字符串常量是==的



# 例 TestStringEquals.java

- `String hello = "Hello", lo = "lo";`
- `System.out.println( hello == "Hello"); //true`
- `System.out.println( Other.hello == hello ); //true`
- `System.out.println( hello == ("Hel"+"lo") ); //true`
- `System.out.println( hello == ("Hel"+lo) ); //false`
- `System.out.println( hello == new String("Hello")); //false`
- `System.out.println( hello == ("Hel"+lo).intern()); //true`