Java语言入门

Java语言入门

◈ 在Java，选用正确的基本数据类型是非常重要的，而主要依据就是数据类型的取值范围。

◈ 基本数据类型是内置的数据类型。更大、更复杂的数据类型可以采用简单数据类型的组合来定义。

数据类型	所占位数	值的范围
boolean	1	true/false
char	16	0 ～65535
byte	8	-2^7～2^7-1
short	16	-2^{15}～$2^{15}-1$
int	32	-2^{31}～$2^{31}-1$
long	64	-2^{63}～$2^{63}-1$
float	32	3.4e－38 ～3.4e＋38
double	64	1.7e－308 ～1.7e＋308

计算机中的数据存储

原码：

byte：-127~+127

补码：

数据范围：-128~127

正数：补码 = 原码

负数：符号位不变，其余取反，最后+1

$2^8 - |x|$

正数+负数 <=> 正数+负数的补码

Java语言的元素——数据类型

8种基本类型

- `char`(只能单引号)(16)
  - **字符串**数据取值范围为`0-65535`，或者说\u0000 - \uFFFF
  - `String`类型表示一串字符，**不是8种基本类型**，使用**双引号**括起来，最后字符不是`\0`
- `numeric`：`byte`(8)，`int`(32)，`short`(16)，`long`(64)
  eg. 2L(long 类型)，077L(8进制)，0xBABEL(16进制)
  将一个int型变量赋值为short型或byte型变量，必须**显式使用类型转换**

```java
short b1 = 1, b2 = 2;
// short b3 = b1 + b2; // 会报错，因为b1+b2会变成int类型，需要进行类型转换
short b3 =(short)(b1 + b2);

byte b1 = 1, b2 = 0;
byte b3 = (byte)(b1 + b2); // 同理

long l1 = 65536 * 63356; // 乘法运算越界，l3为0
long l4 = 65535L - 63356; // l4 = 4294967296L
```

以最长的数字类型决定类型

`char`类型可以和`short`类型互相转化

```java
short s = 100;
char c = 'A';

// short 到 char 的隐式转换
c = s;

// char 到 short 的隐式转换
s = c;
```

- **float**: float(32)(不能超过$3.4*10^{31}$, double(64)
  - 浮点数包含小数点、指数，或者以字母F或D结尾，缺省为double.
  - 浮点数（实数）常量的两种表示法
    - 十机制小数形式： 0.23 .18 -234.
    - 科学计数法形式： 0.23e6 1.23E-4
    - 32位浮点数形式： 0.23f 1.23E-4f .18F
  - 示例
    - double d1 = 127.0; \\ 赋初值为127
    - double d2 = 127; \\ 赋初值为127
    - float f1 = 127.0f; \\ 必须在数字后加f或F
    - float f2 = 4.0e38f; \\ 错误！32位浮点数不能超过 3.4028234663852886e38
    - float f3 = (float)d1; \\ 必须强制类型转换
- *boolean*: true / false, false为缺省值
  - eg. boolean b = (b1 && b2) != false
- 浮点数*缺省为double型*，整数*缺省为int型*

Java中的基本类型转换分为两类：自动类型转换（隐式转换）和强制类型转换（显式转换）。

自动类型转换（隐式转换）

自动类型转换发生在当赋值给目标类型时，源类型的数据范围小于或等于目标类型时。Java编译器会自动完成这种转换。常见的自动类型转换包括：

- **从低精度到高精度：**
  - `byte` 到 `short`、`int`、`long`、`float` 或 `double`
  - `short` 到 `int`、`long`、`float` 或 `double`
  - `char` 到 `int`、`long`、`float` 或 `double`
  - `int` 到 `long`、`float` 或 `double`
  - `long` 到 `float` 或 `double`
  - `float` 到 `double`

强制类型转换（显式转换）

强制类型转换是在赋值时，源类型的数据范围大于目标类型，或者是两种不兼容类型之间的转换。在这种情况下，需要显式地进行类型转换。强制类型转换可能导致数据丢失或精度降低。语法是在要转换的值前面加上目标类型，括在圆括号内。
例如：

```java
double d = 9.7;
int i = (int) d; // 强制将double类型转换为int类型
```

这里，`double` 类型的 `d` 被强制转换为 `int` 类型的 `i`。由于 `double` 类型的精度高于 `int`，因此可能会丢失小数部分。

注意事项

- 当进行强制类型转换时，应当意识到可能的数据丢失或精度降低，特别是从高精度到低精度的转换。
- 对于布尔类型，没有隐式或显式的转换到或从其他类型。
- 在使用强制类型转换时，应当小心，确保转换是符合预期的。

其余均为对象

`class`,`array`, 引用

Java运算符

此处单独把位运算符摘出，以作为提醒

Java的运算符——位运算符

运算符	功能
~	位补运算符——翻转一个值的各位
&	位与运算符——使两个值的各位相与
|	位或运算符——使两个值的各位相或
^	位异或运算符——使两个值的各位相异或
<<	位左移运算符——把一个值的各位向左移动指定的
>>	位右移运算符——把一个值的各位向右移动指定的位数*
~	位取反运算符——对数据的每个二进制位取反
>>>	无符号右移运算符——把一个值的各位向右移动指定的位数*

Java语言中的变量与常量

Java语言的变量与常量

- 与C、C++不同，Java中不能通过#define命令把一个标识符定义为常量，而是用关键字final来实现，如
  - final double PI=3.14159
- Java中的常量值是用数值表示的，它区分为不同的类型
  - 如整型常量123，实型常量1.23，字符常量‘a’，布尔常量true、false，以及字符串常量"constantstring."
- 变量是Java程序中的基本存储单元，它的定义包括变量名、变量类型和作用域几个部分
  - 例如：int a,b,c;
  - doubled1,d2=0.0;

变量和作用区域：

- 变量在类中的定义位置也分为两类：
- 局部（local）
  - 定义在方法或构造器内，包括参数，又称自动、临时或堆栈变量
  - 方法或构造器参数，当方法或构造器调用时生成，伴随方法的存在
  - 当进入方法时，局部变量生成，退出方法时销毁，只可在定义它的方法内使用
  - 可以在多个不同方法中使用同名的变量
- 全局（global）
  - 全局变量定义在方法或构造器外，当调用new ClassName()构造对象时生成
  - 类全局变量，使用static定义，当类装载时产生，一直伴随类的存在而存在；
  - 实例全局变量，又称为成员变量，对象创建时生成，一直伴随对象的存在。

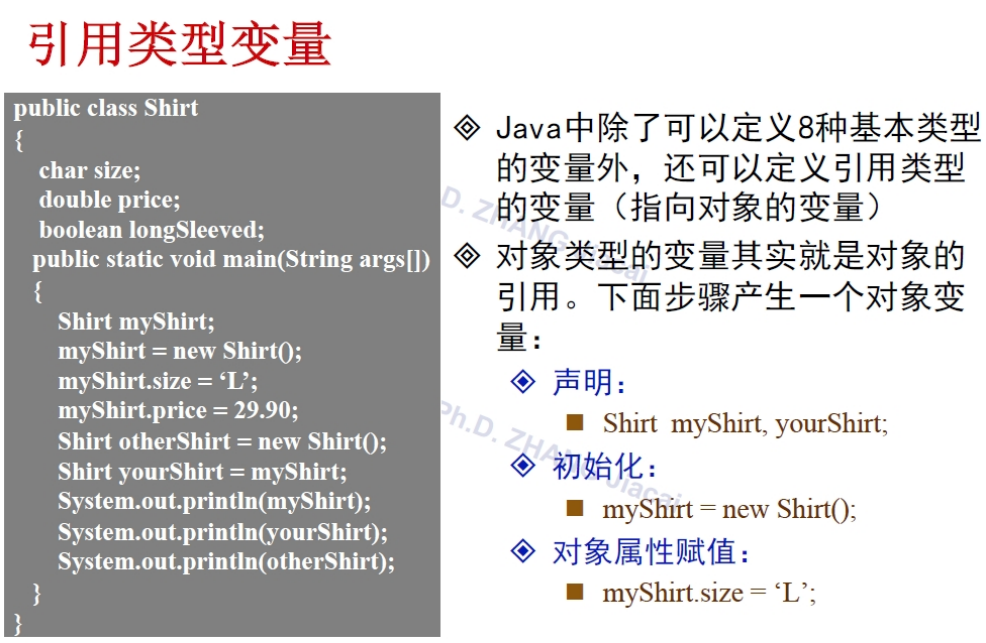
JAVA

```java
public class TestScope {
        public static void main(String[] args) {
                ScopeExample scope = new ScopeExample();
                scope.firstMethod();
                System.out.println(scope.getValue()); // 打印15
        }
}

class ScopeExample{
        private int i = 1;
        public void firstMethod(){
                int i = 4, j = 5; this.i = i + j;
                secondMethod(7);
        }
        public void secondMethod(int i){
                int j = 8; this.i = i + j;
        }
        public int getValue() {
                return this.i;
        }
}
```

在这个例子中，`scope`的`i` 一开始先被初始化为1，即`this.i = 1`，而后再调用`firstMethod`方法，使得`i`变成9，再调用`secondMethod`，使`i`变成15，因为这里始终修改的是`this.i`，是成员变量，所以可以在`getValue()`方法中显示

引用类型变量



引用类型变量（续）

```
public class Shirt
{
   char size;
   double price;
   boolean longSleeved;
  public static void main(String args[])
   {
      Shirt myShirt;
      myShirt = new Shirt();
      myShirt.size = 'L';
      myShirt.price = 29.90;
      Shirt otherShirt = new Shirt();
      Shirt yourShirt = myShirt;
      System.out.println(myShirt);
      System.out.println(yourShirt);
      System.out.println(otherShirt);
   }
}
```

◈ 对象类型的变量代表一个对象，本身的存储区域没有存放对象。对象存储在其它位置，变量代表的是对象的地址（就象C++中的指针）

原始变量	内存中的值
a	2
myShirt	0x34a012
yourShirt	0x34a012
otherShirt	0x99f311

此处必须要注意：对象类型的变量里面存储的只是对象的地址!!

方法重载与静态成员 > 判断相等

成员变量作为引用类型会自动初始化为`null`，但局部变量需要自行初始化（**数组除外**）

Java语句

由于大部分与C语言一致，故不赘述，此处只对特殊的语句形式进行叙述

Java语言的控制——转移语句break

◈ break语句总是和switch、for、while、do-while语句一起使用。break的作用是直接中断当前正执行的语句，跳出switch或循环语句。

◈ 在switch语句中，break用来终止switch的执行，使程序从switch语句后的第一个语句开始执行。

◈ 在循环中，break语句用来终止当前循环体语句的执行，使程序转移到循环体的下一个语句。

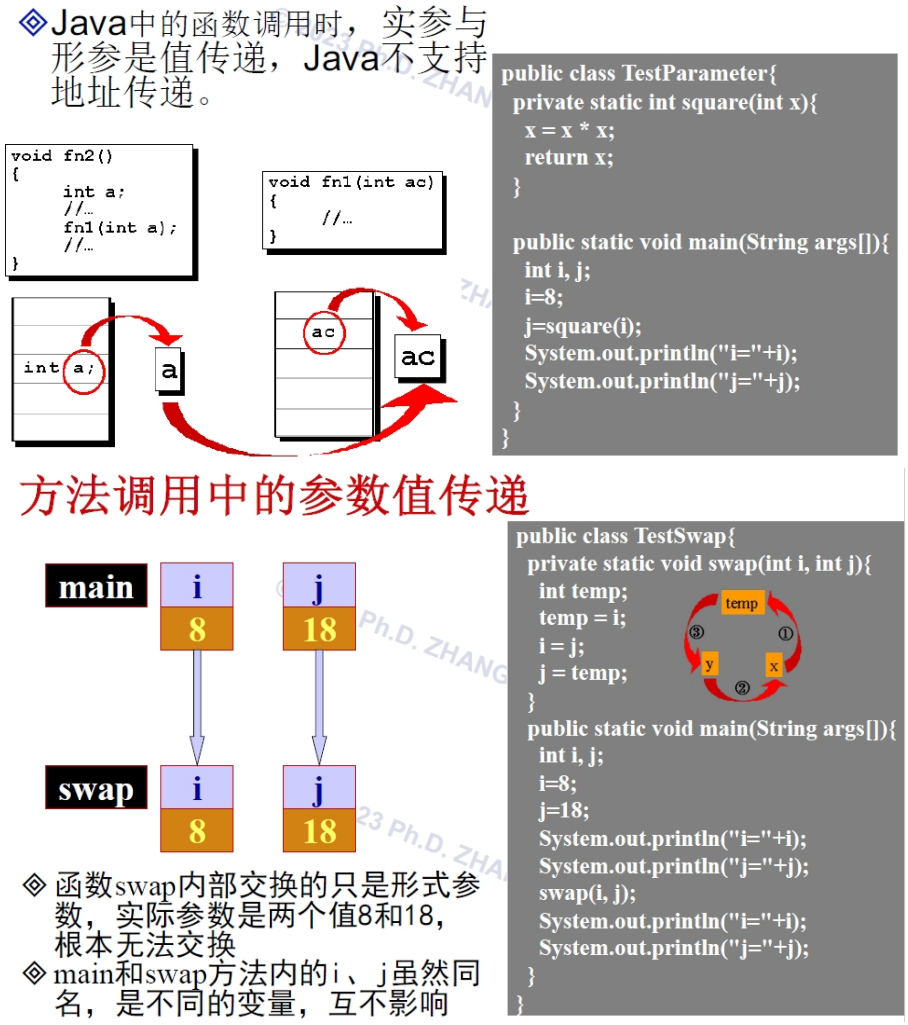
◈ 在多重循环中，可以使用label: statement; 对循环体语句指定标号

- ◈ label放在for, while和do/while语句前
- ◈ 当break后没有标号时，跳出其所在的内层循环
- ◈ 对于带标号的break语句，其格式为：break BlockLabel，跳出label所在的循环

Java语言的控制——转移语句 break

```
0  loop:
1  while(true){
2     for(int i=0; i<100; i++){
3        switch(c = in.read()){
4        case -1:
5           break;
6        case 'n':
7           break loop;
8        }
9     }
10 }
```

方法调用中的参数值传递



在 Java 中，确实不支持传递变量的地址（引用）进行直接的地址传递，因此不能像一些其他语言（如 C++）那样通过操作地址来实现 swap。在 Java 中，传递的是值的拷贝，而不是变量的地址。

然而，你可以通过其他方式实现 swap，其中一种方式是通过使用一个包装类或数组来传递参数。下面是两种可能的实现方式：

使用一个包装类

JAVA

```
class IntWrapper {
    int value;

    IntWrapper(int value) {
        this.value = value;
    }
}

public class SwapExample {
    public static void swap(IntWrapper a, IntWrapper b) {
        int temp = a.value;
        a.value = b.value;
        b.value = temp;
    }

    public static void main(String[] args) {
        IntWrapper x = new IntWrapper(5);
        IntWrapper y = new IntWrapper(10);

        swap(x, y);

        System.out.println("After swap: x = " + x.value + ", y = " + y.value);
    }
}
```

使用数组

JAVA

```
public class SwapExample {
    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] array = {5, 10};

        swap(array, 0, 1);

        System.out.println("After swap: array[0] = " + array[0] + ", array[1] = " +
array[1]);
    }
}
```

这两种方法都是通过传递一个包装类对象或数组，从而达到在方法内交换值的目的。这是因为对象和数组在 Java 中是引用类型，而引用是按值传递的，因此你可以修改对象或数组内的值。

Java数组

数组是一个对象！！

数组定义仅仅分配一个用于访问数组的**引用**，真正的数组元素空间在使用`new`语句或初始化数组时创建

创建数组

创建数组

- Java中数组是引用类型，声明或定义数组并没有创建出存放多个数组元素的空间来。
  - 数组的长度在定义中都没有指定。
- 生成数组使用new语句，但使用[]而不是()，生成对象的构造器用()。
  - Type var[];
  - var = new Type[Length];
- 基本类型的数组，使用new语句首先生成一个数组，并对数组赋以相应的缺省值。
- 比如生成一个包含26个字符值的数组，并将每个数组元素的初始值设为‘\u0000’。

```
//s如何定义;
s = new char[26];
```

- 数组下标往往从0开始，数组下标大于等于0，小于数组长度。

```
int []fieldValue = new int[10];

int []fieldValue;
//如何创建长度为20的fieldValue数组
```

Java数组定义

- 数组定义可以使用两种方式：
  - Type var[];
  - Type []var; 或Type[] var;
- 说明：
  - [ ]在变量右边，只有当前变量为数组；
  - [ ]在变量左边类型右边，对后面所有变量都有影响。
- 数组定义并没有指定数组的实际大小。

```
char    a[], b;
Point   c[];
char    [] s, t; s,t都是数组
Point   [] p;
```

```
int             a[5];
double [5]b;
```

数组元素

数组元素也要重新`new`一个

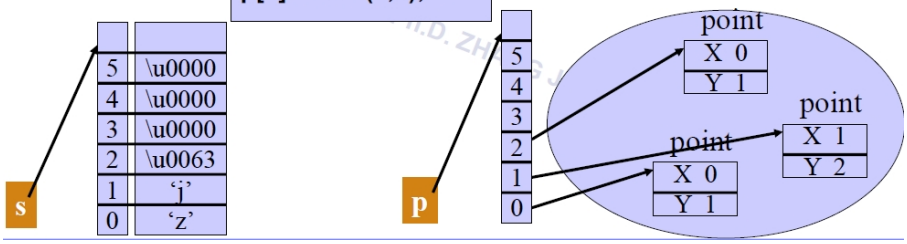
数组元素

◈ 对于基本类型的数组，每个元素是一个基本类型的值。

```
char s[] = new char[26];
s[0] = 'z'; s[1] = 'j'; s[2] = '\u0063';
```

◈ 对于引用类型的数组，每个元素是对象的一个引用，而不是对象本身。

```
p = new Point[10];
p[0]=new Point(0,1);
p[1]=Point(1,2); …
```



`int` 数组会自动初始化为0，对其他的数组则是自动初始化为`null`

初始化数组

◈ 生成数组时，数组中每个元素都被初始化，字符数组中元素初始化为 '\u0000'，对象数组初始化为null，也就是说并没有指向可用的对象。

◈ 所有变量，包括数组中元素的初始化对与系统的安全性有重要意义，避免了变量值的随机性。

◈ Java语言生成带初始值的数组。

```
String names[];
names=new String[3];
names[0]="Tom";
names[1]="Jen";
names[2]="Mike";

String [] names={"Tom", "Jen", "Mike"}
```

```
MyDate dates[];
dates=new MyDate[3];
dates[0]=new MyDate(24,7,2001);
dates[1]= new MyDate(17,2, 1975);
dates[2]= new MyDate(3,9,1994);

MyDate [] dates={
  new MyDate(24,7,2001),
  new MyDate(17,2, 1975),
  new MyDate(3,9,1994)
}
```

多维数组

◈ Java可以生成任何类型的数组，包括数组类型的数组（多维数组）。

```
int twoDim[][] = new int[4][];
twoDim[0] = new int[5];
twoDim[1] = new int[8];
```

◈ 上面第一行生成包含四个元素的数组，每个元素又是一个整型数组的空指针引用，必须分别进行初始化。

◈ 变量定义时，[]可以在变量左右，但其它使用数组的场合不可以随意，如：new int[4][]不可以写为new int [][4]。

◈ 通过上面的方法，可以生成非矩形的数组，这很特别的。

在Java中，可以创建不规则的二维数组，也就是每行长度不同的数组。要遍历这样的数组，你需要使用嵌套循环：外层循环遍历行，内层循环遍历每行的列。由于每行的长度可能不同，因此内层循环应该依赖于当前行的长度。

下面是遍历不规则二维数组的一个例子：

JAVA

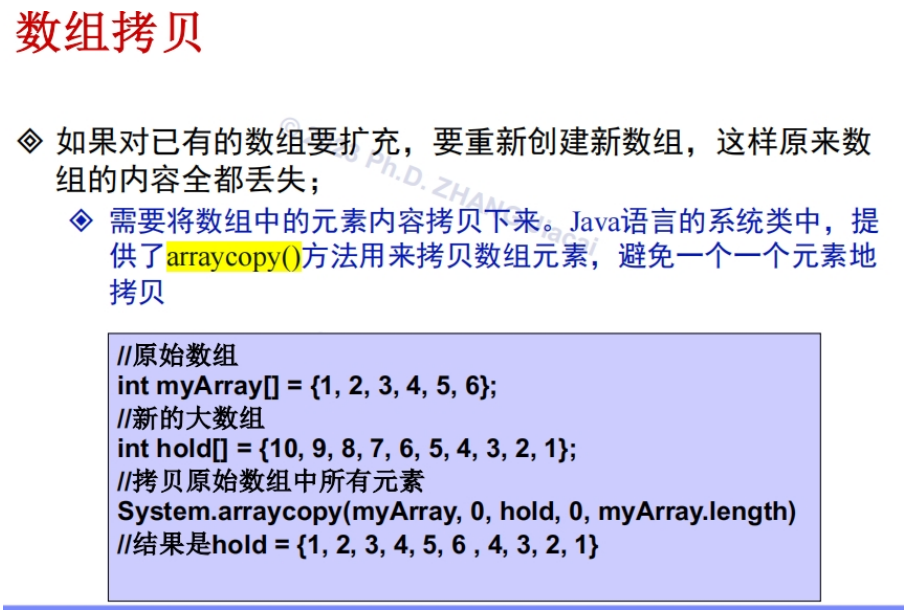
```java
public class IrregularArrayExample {
    public static void main(String[] args) {
        // 创建一个不规则的二维数组
        int[][] irregularArray = {
            {1, 2, 3},
            {4, 5},
            {6, 7, 8, 9}
        };

        // 遍历不规则数组
        for (int i = 0; i < irregularArray.length; i++) {
            for (int j = 0; j < irregularArray[i].length; j++) {
                System.out.print(irregularArray[i][j] + " ");
            }
            System.out.println(); // 换行，以分隔不同的行
        }
    }
}
```

在这个例子中，`irregularArray` 是一个不规则的二维数组。外层循环（`for (int i = 0; i < irregularArray.length; i++)`）遍历数组的每一行，而内层循环（`for (int j = 0; j < irregularArray[i].length; j++)`）遍历当前行中的每一个元素。使用 `irregularArray[i].length` 确保我们正确地获取每一行的长度。

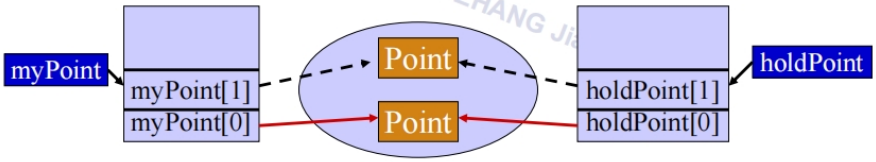
数组拷贝

注意拷贝基本类型和引用类型数组的区别



◈ 如果arraycopy方法拷贝的是引用类型的数组，那么它只拷贝引用，并不拷贝数组元素所引用的对象本身。

```
System.arraycopy(myPoint, 0, holdPoint, 0, myPoint.length)
```



补充：

课堂小问题(7)

What is the result of compiling and running the following code:

```
public class Test {
  static int total = 10;
  public static void main (String args []) { new Test(); }
  public Test () {
    System.out.println("In test");
    System.out.println(this);  // lesson9.Test@49e4cb85，会打印地址
    int temp = this.total;
    if (temp > 5) System.out.println(temp);
  }
}
```

答案：D

A. The class will not compile
B. The compiler reports and error at line 2
C. The compiler reports an error at line 9
D. The value 10 is one of the elements printed to the standard output
E. The class compiles but generates a runtime error

课堂小问题(12)

What appears in the standard output when the method named testing is invoked?

```
void testing() {
one:
two:
  for (int i = 0; i < 3; i++) {
three:
    for (int j = 10; j < 30; j+=10) {
      System.out.println(i + j);
      if (i > 2)
        continue one;
    }
  }
}
```

a) 10 and 20
b) 11 and 21
c) 12 and 22
d) 13 and 23
e) 30, 31, 32, 33

答案：a b c

课堂小问题(14)

Given the following code for the Demo class:

```
public class Demo{
  private int[] count;
  public Demo(){ count=new int[10];}
  public void setCount(int ct, int n){ count[n]=ct;}
  public void showCount(int n){System.out.println("Count is "+count[n]);}
  public int getCount(int n){ return count[n];}
}
```

what would be the result of calling the showCount method with a parameter of 9 immediately after creating an instance of Demo?

a) A NullPointerException would be thrown, halting the program.

b) Standard output would show "Count is 0". // 没有经过setCount设置数组内容，默认初始化为0

c) An ArrayIndexOutOfBoundsException would be thrown, halting the program.

d) Standard output would show "Count is null".

答案：b

课堂小问题(4)

编译或运行以下程序(命令为：java Sub)，会出现什么情况？

```
class Base{
  int var=0;
  public void method(){var = 1;}
}

public class Sub extends Base{
  public void method(){var = -2; // 改变的是继承的var属性}
  public static void main(String args[]){
    Base b=new Sub();
    b.method();// 执行的是Sub的method方法，改变了成员变量
    System.out.println(b.var);
  }
}
```

答案：2

在Java中，不同数据类型占用的内存大小通常如下（但具体大小可能依赖于JVM的实现和运行的平台）：

- `int` 类型通常占用 4 个字节。
- `char` 类型占用 2 个字节。

因此，对于 `Sample` 类的一个实例：

- `int a` 将占用 4 个字节。
- `char c` 将占用 2 个字节。
- 共6个字节

课堂小问题(19)

以下哪些选项的代码是合法的？

a) float a=12.3;
b) String b='Hello';
c) char c=33;
d) double d=14.23;
e) boolean b=TRUE;
f) int b=0x22;
g) long c=345L;
h) byte d=345;

答案：cdfg

在Java中，`char` 类型是一个16位的Unicode字符。虽然 `char` 类型通常用于表示字符，但它也可以用来存储整数值，因为它是一个整数数据类型。`char` 类型的范围是 `0` 到 `65535`，可以存储 Unicode 字符的编码值。

在你的例子中，`char c = 33;` 是合法的，因为整数 `33` 在 `char` 的范围内。在ASCII表中，`33` 对应着感叹号字符 `'!'` 的编码。

这是因为在Java中，可以将整数直接赋给 `char` 类型，而无需显式强制类型转换。当整数值在 `char` 类型的范围内时，它将被隐式转换为对应的字符。

要注意的是，如果整数值超出 `char` 类型的范围，你可能需要进行显式的类型转换。例如，`char c = (char) 1000;`，这里使用了强制类型转换，因为 `1000` 超出了 `char` 类型的范围。