day2

时间：2022年4月7日10:35:57

目录

一、类初始化和实例初始化

1.类初始化

a.步骤

- 一个类要创建实例需要先加载并初始化该类
    - main方法所在的类需要先加载和初始化
- 一个子类要初始化需要先初始化父类
- 一个类初始化就是执行()方法
    - () 方法由静态类变量显示赋值代码和静态代码块组成
    - 类变量显示赋值代码和静态代码块代码从上到下顺序执行
    - ()方法只执行一次

b.示例

```
package com.atguigu.test;

public class Father{
    private int i = test();
    private static int j = method();

    static{
        System.out.print("(1)");
    }
    Father(){
        System.out.print("(2)");
    }
    {
        System.out.print("(3)");
    }

    public int test(){
        System.out.print("(4)");
        return 1;
    }
    public static int method(){
        System.out.print("(5)");
        return 1;
    }
```

```
public class Son extends Father{
    private int i = test();
    private static int j = method();
    static{
        System.out.print("(6)");
    }
    Son(){
        System.out.print("(7)");
    }
    {
        System.out.print("(8)");
    }
    public int test(){
        System.out.print("(9)");
        return 1;
    }
    public static int method(){
        System.out.print("(10)");
        return 1;
    }
    public static void main(String[] args) {
        Son s1 = new Son();
        System.out.println();
        Son s2 = new Son();
    }
```

```
类构造
    父类初始化<clinit>
        1.静态变量显示赋值代码：j = method()
        2.静态代码块：(1)
        3.这两个从上到下顺序执行
    子类的初始化<clinit>
        1.先初始化父类
        2.静态变量显示赋值代码：j = method()
        3.静态代码块：(6)
        4.这两个从上到下顺序执行
    结果：(5)(1)(10)(6)
```

2.实例初始化

a.步骤

- ()方法可能重载有多个，有几个构造器就有几个方法
- () 方法由**非静态实例变量显示赋值代码**和**非静态代码块**、**对应构造器代码**组成
- **非静态实例变量显示赋值代码**和**非静态代码块代码**从上到下顺序执行，而对应构造器的代码最后执行
- 每次创建实例对象，调用对应构造器，执行的就是对应的方法
- 方法的首行是super ()或super(实参列表)，即对应父类的方法

```
注：super方法默认在首行，不写也有
```

b.示例

```
实例构造
    父类实例初始化
        1.i = test()
        2.父类的非静态代码块
        3.父类的构造器
    子类实例初始化
        1.super()
        2.i = test()
        3.子类的非静态代码块
        4.子类的构造器
结果：后序再说
```

c.this问题

- 非静态方法前面其实有一个默认的对象this
- this在构造器(或)它表示的是正在创建的对象，因为这里是在创建Son对象，所以
- test()执行的是子类重写的代码(面向对象多态)
- 所以在执行super的时候，**父类中的this是子类正在初始化的实例**，调用的方法也自然是**子类的test()方法**

```
结果
    父类
        1.i = test(): (9)
        2.非静态代码块: (3)
        3.构造器:      (2)
    子类
        1.super():    (9) (3) (2)
        2.i = test()  (9)
        2.非静态代码块: (8)
        4.构造器:      (7)
结果: (9) (3) (2) (9) (8) (7)

加上之前的类构造，就是最终结果
```

3.方法重写的Override

a.那些方法不可以被重写

- final方法
- 静态方法
- private等子类中不可见方法

b.对象的多态性

- 子类如果重写了父类的方法，通过子类对象调用的一定是子类重写过的代码
- 非静态方法默认的调用对象是this
- this对象在构造器或者说方法中就是正在创建的对象

4.注意点

- 代码块和成员的初始化是顺序执行的
- 父类中的this，指向的是子类正在创建的对象

二、值传递和引用传递

1.基本数据类型：值传递

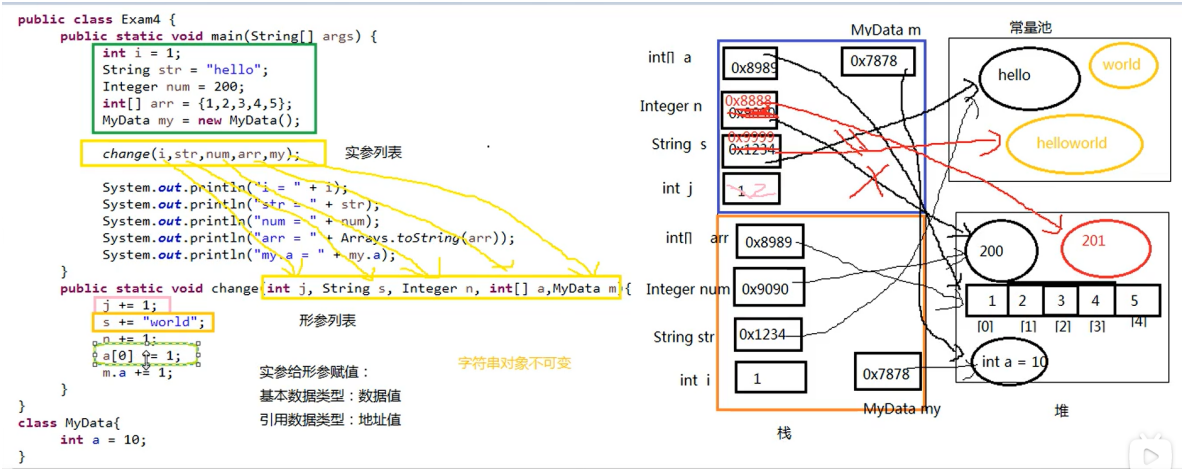
2.引用数据类型

- 地址传递
- String和包装类的不可变性

```
String类和包装类
    传递的任然是引用，但是在修方法中改的时候，会生成一个新的数据放入堆中，并将形参的值设为新的值的地址
    实参的地址以及堆中的值没有发生任何变化

Integer类
    凡是new的对象，即使值相等，引用也是不同的，但是可以使用equals()判断值
        Integer int1 = new Integer(127);
        Integer int2 = new Integer(127);
        int1 == int2; // false
        int1.equals(int2); //true
    jdk提供了-128~127的缓存，如果代码中使用了，就直接获取，所以Integer对象的引用是相同的(不是new出来的)
```

```
12          Integer int1 = 127;
13          Integer int2 = 127;
14          int1 == int2; // true
```

3.示例



重点在于理解堆栈结构以及常量池

三、编程题（台阶问题）

1.问题

有n阶台阶，每次可以走1或者2步，有几种走法（有点类似于斐波那契）

2.递归

a.思路

- 可以想象成最后一次走就到顶了
- 由于最后一次可以走1或者2，所以需要求前n-1阶的走法+前n-2阶走法

递归

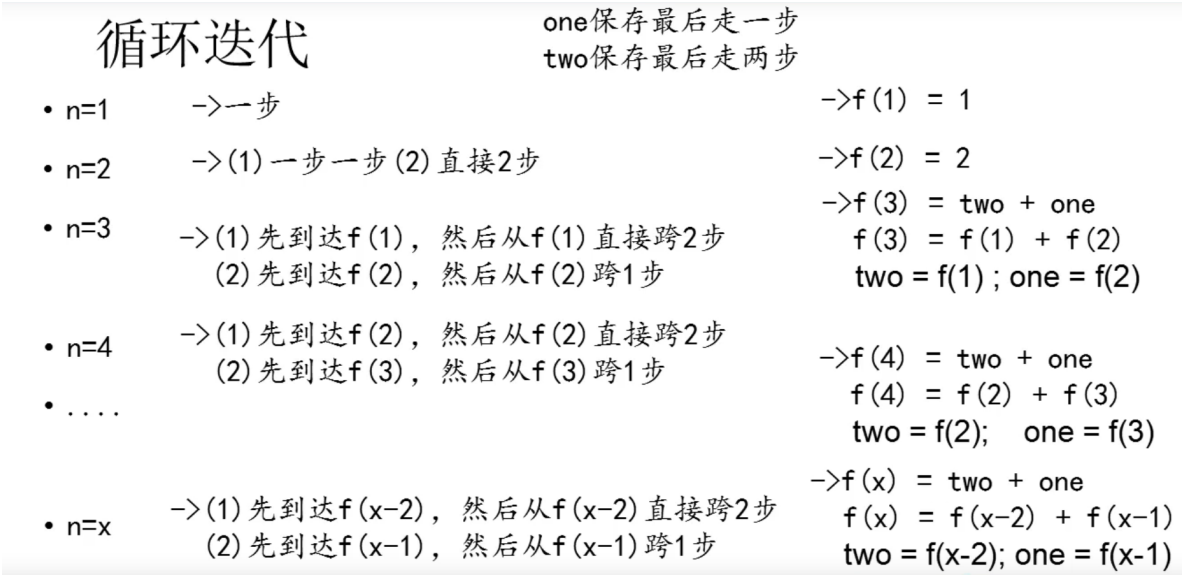
- n=1 ->一步 ->f(1) = 1
- n=2 ->(1)一步一步 (2)直接2步 ->f(2) = 2
- n=3 ->(1)先到达f(1)，然后从f(1)直接跨2步
(2)先到达f(2)，然后从f(2)跨1步 ->f(3) = f(1) + f(2)
- n=4 ->(1)先到达f(2)，然后从f(2)直接跨2步
(2)先到达f(3)，然后从f(3)跨1步 ->f(4) = f(2) + f(3)
- ....
- n=x ->(1)先到达f(x-2)，然后从f(x-2)直接跨2步
(2)先到达f(x-1)，然后从f(x-1)跨1步 ->f(x) = f(x-2) + f(x-1)

b.代码实现

```java
package com.atguigu.step;

import org.junit.Test;

public class TestStep{
    @Test
    public void test(){
        System.out.println(f(4));
    }

    //实现f(n)：求n步台阶，一共有几种走法
    public int f(int n){
        if(n<1){
            throw new IllegalArgumentException(n + "不能小于1");
        }
        if(n==1 || n==2){
            return n;
        }
        return f(n-2) + f(n-1);
    }
}
```

3.非递归

a.思路



b.代码实现

```java
public class TestStep2 {
    public int loop(int n){
        if(n<1){
            throw new IllegalArgumentException(n + "不能小于1");
        }
        if(n==1 || n==2){
            return n;
        }

        int one = 2;//初始化为走到第二级台阶的走法
        int two = 1;//初始化为走到第一级台阶的走法
        int sum = 0;

        for(int i=3; i<=n; i++){
            //最后跨2步 + 最后跨1步的走法
            sum = two + one;
            two = one;
            one = sum;
        }
        return sum;
    }
}
```

4.总结

- 递归可读性强，代码量少，但是效率可能慢
- 迭代可读性查，代码量多， 但是效率非常高