在JDK1.7之前运行时常量池逻辑包含字符串常量池存放在方法区，此时hotspot虚拟机对方法区的实现为永久代

在JDK1.7 字符串常量池被从方法区拿到了堆中，这里没有提到运行时常量池,也就是说字符串常量池被单独拿到堆,运行时常量池剩下的东西还在方法区，也就是hotspot中的永久代

在JDK1.8 hotspot移除了永久代用元空间(Metaspace)取而代之，这时候字符串常量池在堆中，运行时常量池还在方法区，只不过方法区的实现从永久代变成了元空间(Metaspace)
元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，**元空间的大小仅受本地内存限制**

对于直接做+运算的两个字符串（字面量）常量，并不会放入字符串常量池中，而是直接把运算后的结果放入字符串常量池中
(String s = "abc"+ "def", 会直接生成“abcdef"字符串常量 而不把 "abc" "def"放进常量池)

对于先声明的字符串字面量常量，会放入字符串常量池，但是**若使用字面量的引用进行运算就不会把运算后的结果放入字符串常量池中了**
(String s = new String("abc") + new String("def"),在构造过程中不会生成“abcdef"字符串常量)

总结一下就是**JVM会对字符串常量的运算进行优化，未声明的，只放结果；已经声明的，只放声明**

它的主要使用方法（或者说如何保证变量指向的是字符串常量池中的数据）有两种：
1.直接使用双引号声明出来的String对象会直接存储在字符串常量池中。
String s = "abc";

2.如果不是用双引号声明的String对象，可以使用String提供的intern方法，这个下面会解释，先记住以下结论。
字符串常量池存的东西有两种情况：
1.字符串对象，比如上面的“abc”
2.堆对象的引用地址。(jdk8)

new String("ab")会创建几个对象？

两个对象，一个是在堆空间中，一个在字符串常量池中（字节码指令ldc）。有兴趣的小伙伴可以去看编译后的字节码文件。

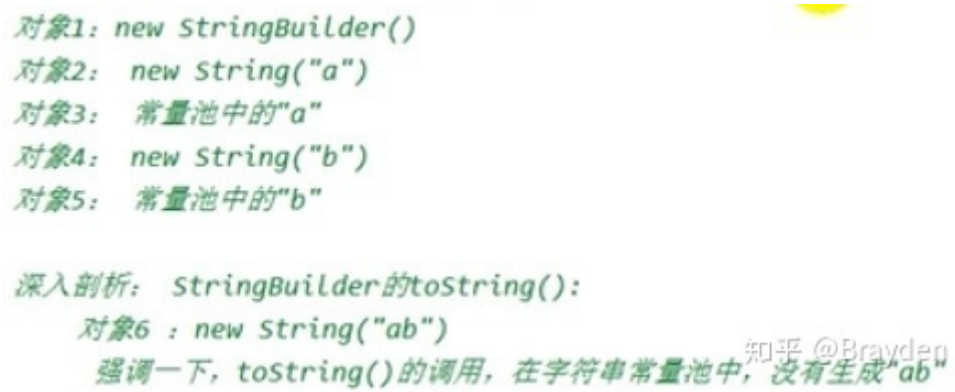
例如： String s = new String("s").intern();

解读该行代码：new String("s")代表创建了两个对象，一个是在堆空间中，一个在字符串常量池中。new String("s").intern()则代表**返回字符串常量池中，new String("s")；实例对象在堆中的引用地址(jdk8)**

字符串拼接的时候只要其中有一个是变量(非final修饰)，拼接出来的对象就在堆中，相当于在堆空间中new String("XXX")（不是在字符串常量池中）。变量拼接的原理是StringBuilder调用append方法然后再调用toString方法。

new String("a")+new String("b")会创建几个对象呢？

六个对象。有兴趣的小伙伴可以去看编译后的字节码文件。 通过看字节码文件可知：



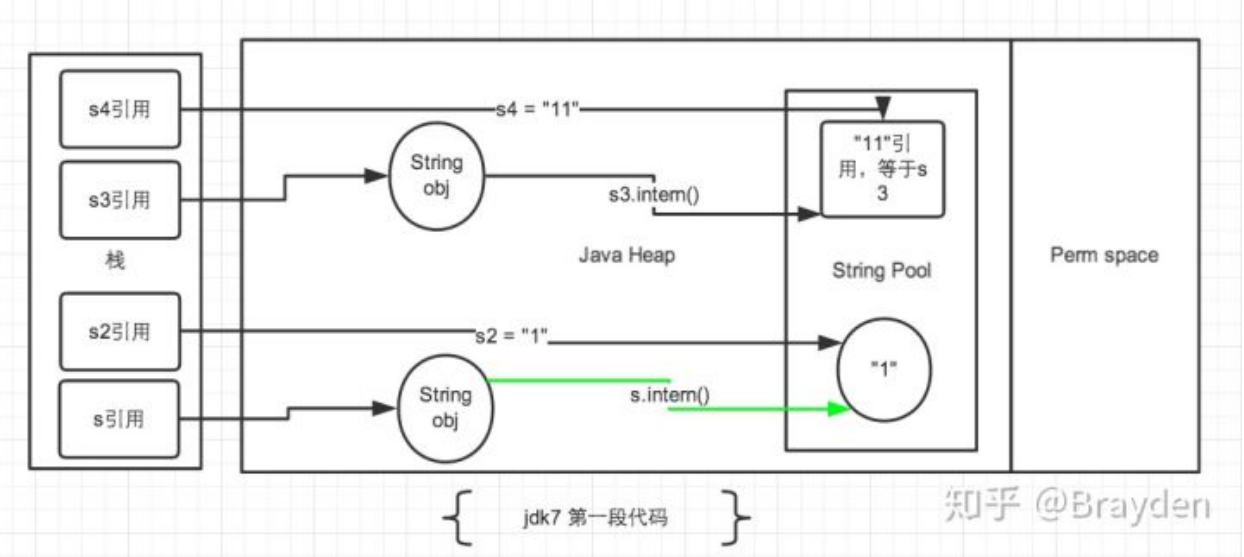

```
package intern;

// 在jdk8中，intern方法的作用是如果字符串常量池已经包含一个等于(通过equals方法比较)此String对象的字符串，则返回字符串常量池中这个字符串的引用,
// 否则将当前String对象的引用地址（堆中对象的引用地址）添加（或者叫复制）到字符串常量池中并返回，这么做是为了节约堆空间，毕竟都在堆中。jdk1.7之前会直接将对象复制到字符串常量池中，并返回在串池中的引用
public class InternTest {
    public static void main(String[] args) {
        String s = new String("l");
        s.intern();// 调用此方法之前，字符串常量池中已经存在了“l”，所以并不会把s对象的引用地址加入字符串常量池中
        String intern = s.intern();// 调用此方法之前，字符串常量池中已经存在了“l”，所以并不会把s对象的引用地址加入字符串常量池中
        String s2 = "l";
```

```
        System.out.println(s == s2);  // false  s 表示堆中 s 对象的引用地址
        System.out.println(intern == s2);  // false  s 表示堆中 s 对象的引用地址

        String s3 = new String("l") + new String("r");  // s3变量记录的地址为：new String("lr")，堆中。
        // 执行完上一行代码以后，字符串常量池中，不存在“lr”
        s3.intern();  //jdk7/jdk8中，此时字符串常量中并没有创建“lr”，而是添加一个指向堆空间中new String("11")的引用地址


        String s4 = "lr";
        System.out.println(s3 == s4);  // true
    }
}
```



总结String的intern()的使用：

- jdk1.6中，将这个字符串对象尝试放入串池。
  - 如果串池中有，则并不会放入。返回已有的串池中的对象的地址
  - 如果没有，会把此对象复制一份，放入串池，并返回串池中的对象地址
- Jdk1.7起，将这个字符串对象尝试放入串池。
  - 如果串池中有，则并不会放入。返回已有的串池中的对象的地址
  - 如果没有，则会把对象的引用地址复制一份，放入串池，并返回串池中的引用地址

Q：下列程序的输出结果：

String s1 = “abc”;

```
final String s2 = “a”;
final String s3 = “bc”;
String s4 = s2 + s3;
System.out.println(s1 == s4);
```

A：true，因为final变量在编译后会直接替换成对应的值，所以实际上等于s4=”a”+”bc”，而这种情况下，编译器会直接合并为s4=”abc”，所以最终s1==s4。

输出结果？创建了几个对象？

```
String s3 = new String("xyz");
String s4 = new String("xyz");
System.out.println(s3==s4);
```

结果输出：

```
false
```

创建了3个对象。采用new关键字新建一个字符串对象时，JVM首先在字符串池中查找有没有"xyz"这个字符串对象，如果有，则不在池中再去创建"xyz"这个对象了，直接在堆中创建一个"xyz"字符串对象，然后将堆中的这个"xyz"对象的地址返回赋给引用s3，这样，s3就指向了堆中创建的这个"xyz"字符串对象；如果没有，则首先在字符串池中创建一个"xyz"字符串对象，然后再在堆中创建一个"xyz"字符串对象，然后将堆中这个"xyz"字符串对象的地址返回赋给s3引用，这样，s3指向了堆中创建的这个"xyz"字符串对象。s4则指向了堆中创建的另一个"xyz"字符串对象。s3 、s4是两个指向不同对象的引用，结果当然是false。

当intern()方法被调用的时候，如果字符串常量池中已经存在这个字符串对象了，就返回常量池中该字符串对象的地址；如果字符串常量池中不存在，就在常量池中创建一个指向该对象堆中实例的引用，并返回这个引用地址（jdk1.7之前会直接将对象赋值到常量池中）。

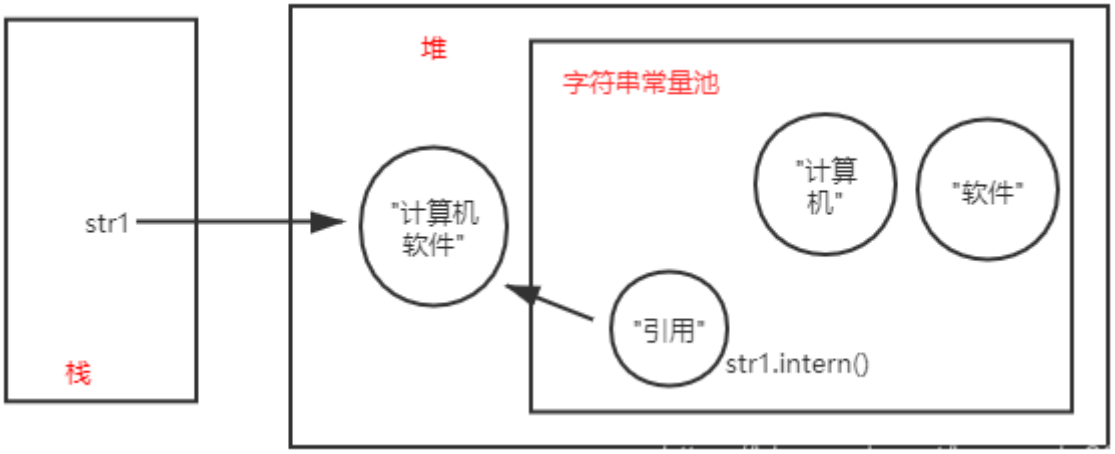
```
String str1 = new StringBuilder("计算机").append("软件").toString();
System.out.println(str1.intern()==str1); // true
```

第一行代码

- 执行到括号中的“计算机”时会在字符串常量池中添加“计算机”对象；
- 执行到括号中的“软件”时会在字符串常量池中添加“软件”对象；
- 执行完毕后会在堆中创建“计算机软件”对象，并将栈中的str1变量指向该对象；

第二行代码

执行到str1.intern()的时候发现字符串常量池中没有“计算机软件”这个字符串，于是在字符串常量池中创建一个引用，指向堆中的对象（即指向和str1相同），然后返回该引用；
执行完毕后打印true；
内存中的图示如下，由于str1和str1.intern()都指向堆中的对象，因此结果为true。

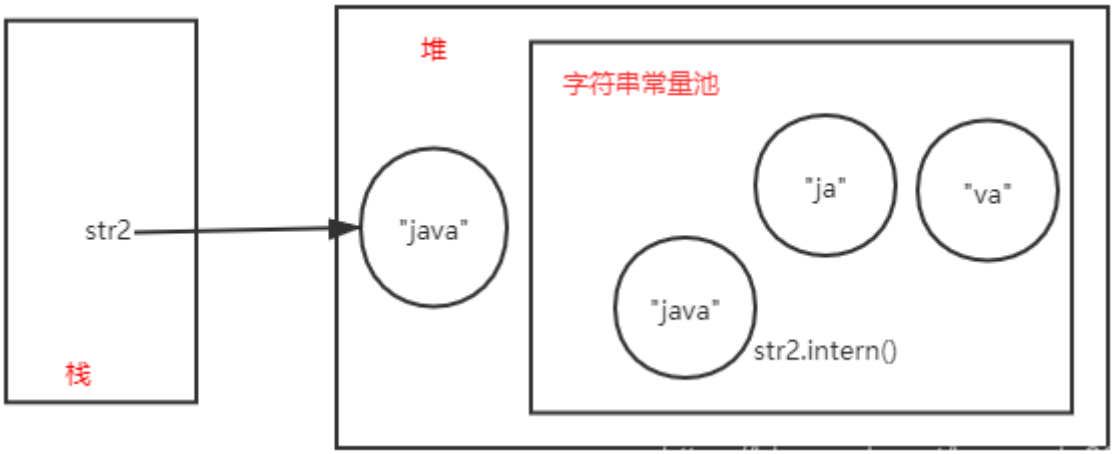



```
String str2 = new StringBuilder("ja").append("va").toString();
System.out.println(str2.intern() == str2); // false
```

第一行代码

- 执行到“ja”的时候在字符串常量池中创建“ja”对象；
- 执行到“va”的时候在字符串常量池中创建“va”对象；
- 执行完毕后会在堆中创建“java对象”，并将栈中的变量str2指向该对象；

第二行代码
执行到str2.intern()的时候发现字符串常量池中已经存在“java”对象，因此返回该对象的引用（这个java对象并不是我们创建的，因为“java”是java语言中的关键字，它是字符串常量池中默认就存在的）；
执行完毕后打印false；
内存中的图示如下，因为str2.intern()指向字符串常量池中的对象，而str2指向堆中的对象，因此结果为false。



```
String s = new String("1");
s.intern();
String s2 = "1";
System.out.println(s == s2); // false
```

第一行代码

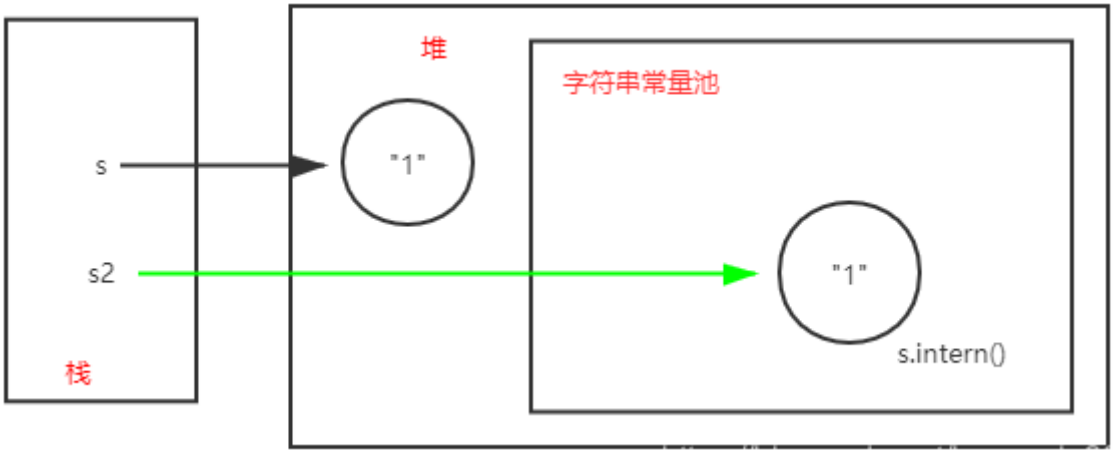
- 执行到括号中的“1”的时候会在字符串常量池中创建“1”对象；
- 执行完毕后会在堆中创建“1”对象，并将栈中的s变量指向该对象；

第二行代码执行完毕后会因为“1”已经在字符串常量池中了，而不做其他操作，只会把这个“1”的地址返回，但是我们并没有接收；

第三行代码执行完毕时会因为字符串常量池中已经有“1”这个对象了，而不执行创建操作，直接将栈中的变量s2指向这个对象；

第四行代码执行完毕后打印false；

内存中的图示如下，因为s指向堆中的对象，而s2指向字符串常量池中的对象，因此结果为false。




```
String s3 = new String("1") + new String("1");
s3.intern();
String s4 = "11";
System.out.println(s3 == s4); // true
```

第一行代码

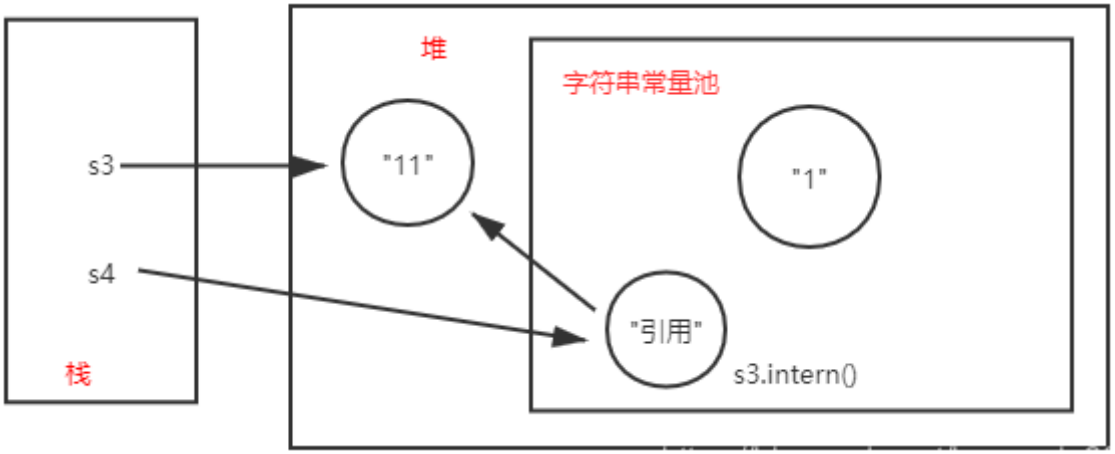
- 执行到括号中的“1”时会在字符串常量池中创建一个对象“1”，执行到第二个括号中的“1”时因为池中已经有“1”这个对象了，因此不重复创建；
- 执行完毕后在堆中创建对象“11”，并将栈中的变量s3指向该对象；

第二行代码执行完毕后会因为字符串常量池中没有“11”这个对象，而在字符串常量池中创建一个引用，指向堆中的“11”这个对象，也就是和s3的指向相同；

第三行代码执行完毕后发现字符串常量值中已经有“11”了，只不过它是一个指向堆中的引用（这里可以理解为堆中已经有“11”这个对象了），那么不重复创建，而是直接将s4指向该引用。

第四行代码执行完毕后打印true

内存中的图示如下，因为s3和s4同时指向堆中的对象，因此结果为true




```
String str1 = new String("SEU") + new String("Calvin");
System.out.println(str1.intern() == str1); // true
System.out.println(str1 == "SEUCalvin"); // true
```

第一行代码

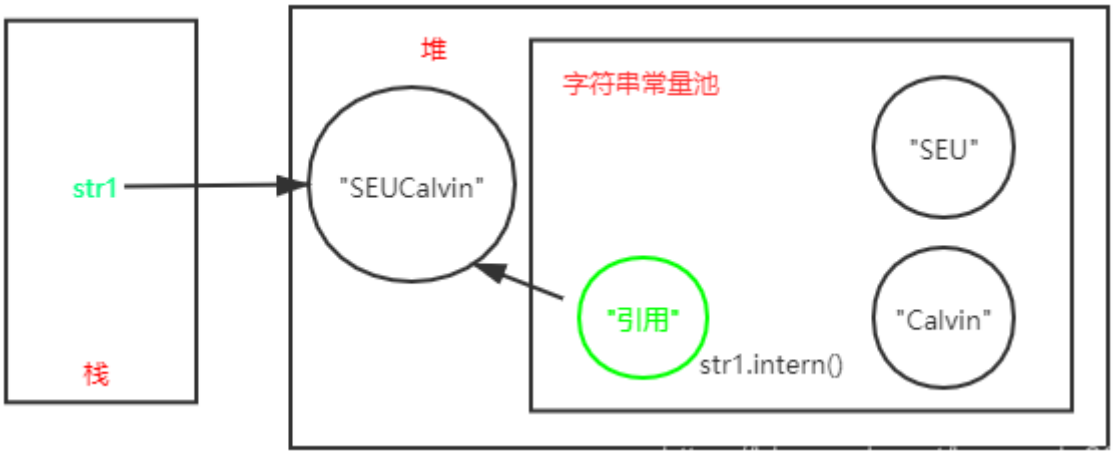
- 执行到第一个括号中的“SEU”时在字符串常量池中创建“SEU”对象，执行到第二个括号中的“Calvin”时在字符串常量池中创建“Calvin”对象；
- 执行完毕后在堆中创建“SEUCalvin”对象，并将栈中的变量str1指向该对象；

第二行代码

- 执行到str1.intern()时因为字符串常量池中没有“SEUCalvin”对象，而在常量池中创建一个引用，该引用指向堆中的“SEUCalvin”对象，也就是该引用和str1的指向相同；
- 执行完毕后打印true；

第三行代码执行完毕后打印true；

内存中的图示如下，因为str1.intern()和str1同时指向堆中的同一个对象，因此第二行结果为true；因为字符串常量池中的“SEUCalvin”对象是一个指向堆中“SEUCalvin”对象的引用，而str1也指向堆中的这个对象，因此第三行结果为true。



其实第二行和第三行代码比较的都是两个绿色部分是否相等。

```
String str2 = "SEUCalvin"; // 新加的一行代码，其余不变
String str1 = new String("SEU") + new String("Calvin");
System.out.println(str1.intern() == str1); // false
System.out.println(str1 == "SEUCalvin"); // false
```

这几行代码只是在最前面加了一行，其他三行完全是上面的代码，但结果却截然不同。

第一行代码会在字符串常量池中创建“SEUCalvin”对象，并将栈中的变量str2指向该对象；

第二行代码

- 执行到括号中的“SEU”时会在字符串常量池中创建“SEU”对象，执行到括号中的“Calvin”时会在字符串常量池中创建“Calvin”对象；
- 执行完毕后会在堆中创建“SEUCalvin”对象，并将栈中的str1变量指向该对象；

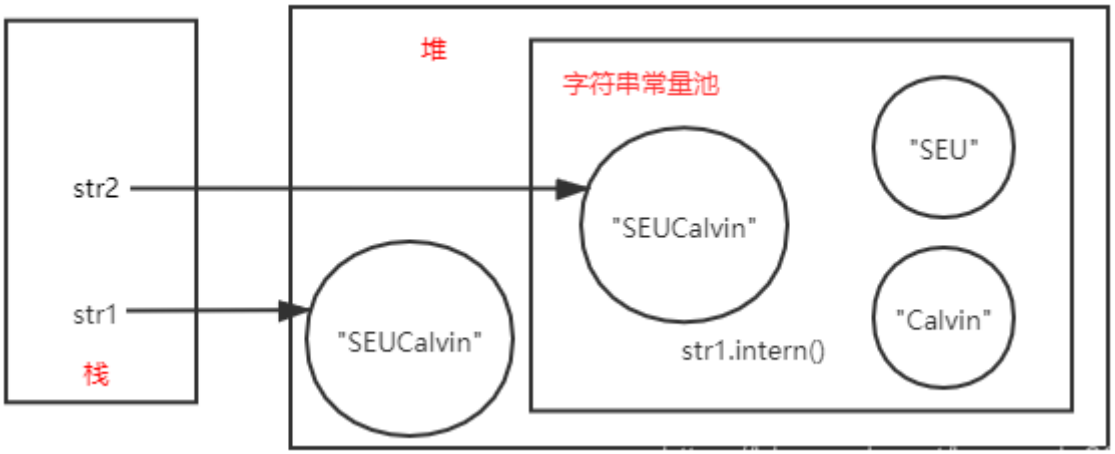
第三行代码

- **执行到str1.intern()时发现字符串常量池中已经存在“SEUCalvin”对象，因此直接返回池中该对象地址；**
- 执行完毕后打印false；

第四行代码

- 执行到括号中的“SEUCalvin”会因字符串常量池中已经存在该对象而不做任何操作；
- 执行完毕打印false；

内存中的图示如下，因为str1.intern()指向字符串常量池中的对象，而str1指向堆中的对象，因此结果为false。



String str1 = new String("SEU") + new String("Calvin");

此行代码会创建6个对象。分别是：

- 常量池中的“SEU”对象
- 常量池中的“Calvin”对象
- 堆中的“SEU”对象，直接变成垃圾
- 堆中的“Calvin”对象，直接变成垃圾
- 堆中 new StringBuilder().append("SEU").append("Calvin").toString();
- 堆中的“SEUClvin”对象，栈中的str1变量指向该对象

由于堆中的“SEU”对象和“Calvin”对象直接变成垃圾了，因此上文中并没有分析这两个对象。

使用指令 javap -v xxx.class分析字节码文件可得：

```
{
  static final int low;
    descriptor: I
    flags: ACC_STATIC, ACC_FINAL
    ConstantValue: int -128

  public Test();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 6: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=2, args_size=1
1       0: new           #2                  // class java/lang/StringBuilder
         3: dup
         4: invokespecial #3                  // Method java/lang/StringBuilder."<init>":()V
2        7: new           #4                  // class java/lang/String
        10: dup
3       11: ldc           #5                  // String a
        13: invokespecial #6                  // Method java/lang/String."<init>":(Ljava/lang/String;)V
        16: invokevirtual #7                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
4       19: new           #4                  // class java/lang/String
        22: dup
5       23: ldc           #8                  // String b
        25: invokespecial #6                  // Method java/lang/String."<init>":(Ljava/lang/String;)V
        28: invokevirtual #7                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
6       31: invokevirtual #9                  // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
        34: astore_1
        35: return
      LineNumberTable:
        line 10: 0
        line 12: 35
}
SourceFile: "Test.java"
```