

# 常用API

## Important

API: Application Programming Interface, 应用程序编程接口。

## 1. API文档

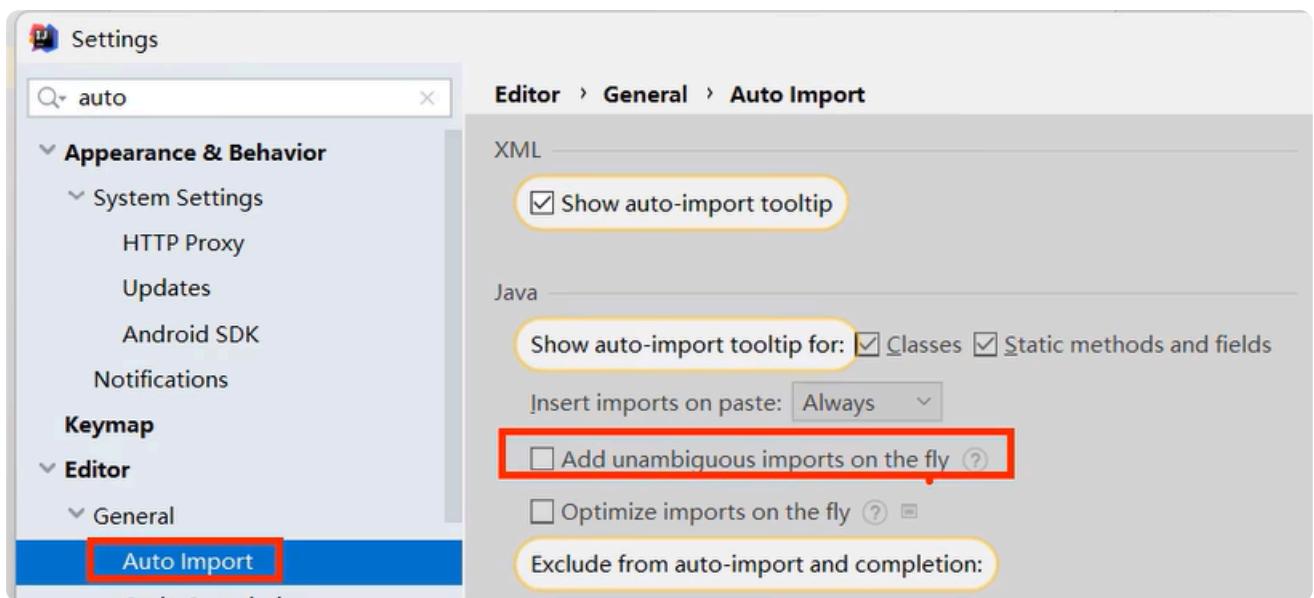
- [JDK21-Docs](#)

## 2. 包

### 2.1 概述

- 包是用来分门别类地管理各种不同程序的，类似于文件夹。
- 建包有利于程序的管理和维护。

### 2.2 IDEA中设置自动导包

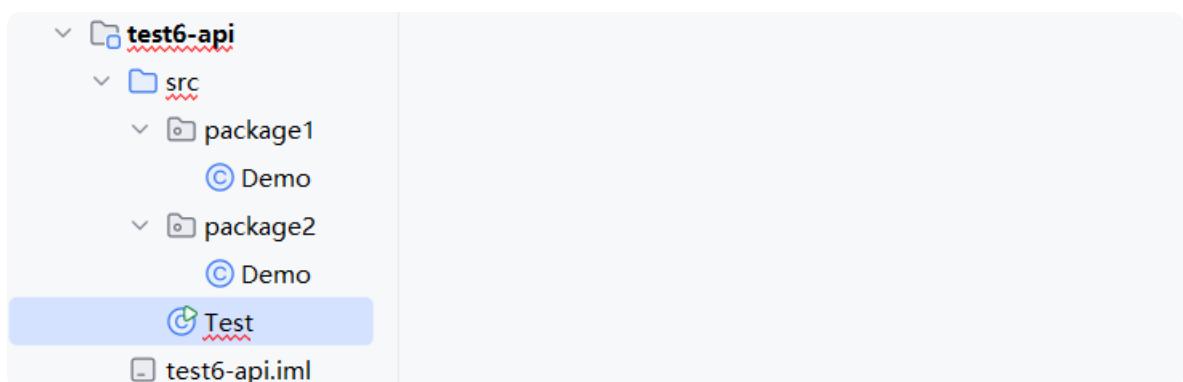


## 2.3 调用其他包下程序的注意事项

- 同一个包下的类可以互相直接调用。
- 若当前程序中需要调用其他包下的程序，则必须在当前程序中导包。

- 导包格式：`import 包名.类名；`
- `java.lang` 包下的程序无需导包即可调用。
  - 若当前程序中需要调用多个不同包下的程序，而这些程序名恰好一样，则此时默认只能导入一个程序，另一个程序必须带包访问。示例如下：

- 先创建两个包 `package1` 和 `package2`，在两个包里面创建一个名字相同的类 `Demo`。



- `package1` 中的 `Demo` 类：

```
1 package package1;
2
3 public class Demo {
4     public void print() {
5         System.out.println("this class locates in package1");
6     }
7 }
```

- `package2` 中的 `Demo` 类：

```
1 package package2;
2
3 public class Demo {
4     public void print() {
5         System.out.println("this class locates in package2");
6     }
7 }
```

- 此时若尝试在 `Test` 类中同时导入这两个名字相同的类，则一定会报错。因为系统无法识别 `main` 方法中 `new` 出来的 `Demo` 类具体是哪一个。

```
1 import package1.Demo;
2 import package2.Demo;
3 
4 public class Test {
5     public static void main(String[] args) {
6         Demo d1=new Demo();
7         d1.print();
8     }
9 }
10 
```

- 正确做法为带包访问，如下：

```
1 import package1.Demo;
2 
3 public class Test {
4     public static void main(String[] args) {
5         Demo d1 = new Demo();
6         d1.print();
7         package2.Demo d2 = new package2.Demo(); // 带包访问
8         d2.print();
9     }
10 }
```

- 控制台输出结果：

```
1 this class locates in package1
2 this class locates in package2
```

### 3. `Scanner`

- 作用：**接收用户键盘输入的数据。
- 代码示例：**输出用户输入的年龄和名字。

```
1 import java.util.Scanner;
```

```
2
3 public class Test {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6
7         System.out.println("请输入您的年龄: ");
8         int age = sc.nextInt();
9         System.out.println("您的年龄是: " + age);
10
11         System.out.println("请输入您的名字: ");
12         String name = sc.next();
13         System.out.println("您的名字是: " + name);
14     }
15 }
```

- 控制台输出结果：

请输入您的年龄:

20

您的年龄是: 20

请输入您的名字:

zsh

您的名字是: zsh

Process finished with exit code 0

## 4. Random

- **作用：**生成随机数。
- **代码示例：**生成 $[0, bound)$ 内的一个随机数。

```
1 import java.util.Random;
2 import java.util.Scanner;
3
4 public class Test {
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7         Random r = new Random();
8
9         System.out.println("请输入bound: ");
10        int bound = sc.nextInt();
11        int number = r.nextInt(bound);
```

```
12     System.out.println("生成了[0," + bound + ")内的一个随机数: " +
13         number);
14 }
```

- 控制台输出结果：

```
请输入bound:
50
生成了[0,50)内的一个随机数: 30

Process finished with exit code 0
```

## 5. String

### 5.1 String 创建对象封装字符串数据的方式

方式一：Java程序中的所有字符串（例如“abc”）都为 String 类的对象。

```
1 String name="black";
2 String schoolName="HIT";
```

方式二：调用 String 类的构造器来初始化字符串对象。

序号	构造器	说明
01	public String()	创建一个空字符串对象，不含任何内容。
02	public String(String original)	根据传入的字符串内容来创建字符串对象。
03	public String(char[] chars)	根据字符数组的内容来创建字符串对象。
04	public String(byte[] bytes)	根据字节数组的内容来创建字符串对象。

## 5.2 String 的常用方法

序号	方法	说明
01	<code>int length()</code>	获取字符串的长度（即字符个数）并返回。
02	<code>char charAt(int index)</code>	获取索引位置处的字符并返回。
03	<code>char[] toCharArray()</code>	将当前字符串转换成字符数组并返回。
04	<code>boolean equals(Object anObject)</code>	判断当前字符串与另一字符串的内容是否一样，若一样则返回 <code>true</code> 。
05	<code>boolean equalsIgnoreCase(String anotherString)</code>	同上，但忽略内容字母的大小写。
06	<code>String subString(int beginIndex,int endIndex)</code>	根据起止索引截取字符串（包前不包后），返回截取后的字符串。
07	<code>String subString(int beginIndex)</code>	只根据起始索引截取字符串，一直截取到字符串末尾，返回截取后的字符串。
08	<code>String Replace(CharSequence target,CharSequence replacement)</code>	使用新的字符片段替换旧的字符片段，返回新的字符串。
09	<code>boolean contains(CharSequence s)</code>	判断当前字符串中是否包含了某个字符片段，若是则返回 <code>true</code> 。
10	<code>boolean startsWith(String prefix)</code>	判断当前字符串是否以某个字符串为开头，若是则返回 <code>true</code> 。
11	<code>String[] split(String regex)</code>	将字符串按照指定的字符串内容进行分割，并返回分割后的字符串数组。

## 5.3 遍历字符串

### 5.3.1 结合02方法

序号	方法	说明
02	<code>char charAt(int index)</code>	获取索引位置处的字符并返回。

- 代码示例：

```
1 public class StringDemo1 {  
2     public static void main(String[] args) {  
3         String s = "ArthurMorgan";  
4         for (int i = 0; i < s.length(); i++) {  
5             char ch = s.charAt(i);  
6             System.out.print(ch + " ");  
7         }  
8     }  
9 }
```

- 控制台输出结果：

```
1 | ArthurMorgan
```

### 5.3.2 结合03方法

序号	方法	说明
03	<code>char[] toCharArray()</code>	将当前字符串转换成字符数组并返回。

- 代码示例：

```
1 public class StringDemo1 {  
2     public static void main(String[] args) {  
3         String s = "ArthurMorgan";  
4         char[] chars = s.toCharArray();  
5         for (int i = 0; i < chars.length; i++) {  
6             System.out.print(chars[i] + " ");  
7         }  
8     }  
9 }
```

- 控制台输出结果：

```
1 | ArthurMorgan
```

## 5.4 比较字符串

- 错误示例：直接用双等号比较字符串是最常见的错误。

```
1 String s1=new String("zsh");  
2 String s2=new String("zsh");  
3 System.out.println(s1 == s2);
```

- 控制台输出结果：

```
1 | false
```

- 正确方法：使用04方法。

序号	方法	说明
04	boolean equals(Object anObject)	判断当前字符串与另一字符串的内容是否一样，若一样则返回 true。

- 示例：

```
1 | String s1=new String("zsh");
2 | String s2=new String("zsh");
3 | System.out.println(s1.equals(s2));
```

- 控制台输出结果：

```
1 | true
```

## 5.5 08方法演示

序号	方法	说明
08	<code>String Replace(CharSequence target,CharSequence replacement)</code>	使用新的字符片段替换旧的字符片段，返回新的字符串。

- 代码示例：替换敏感词。

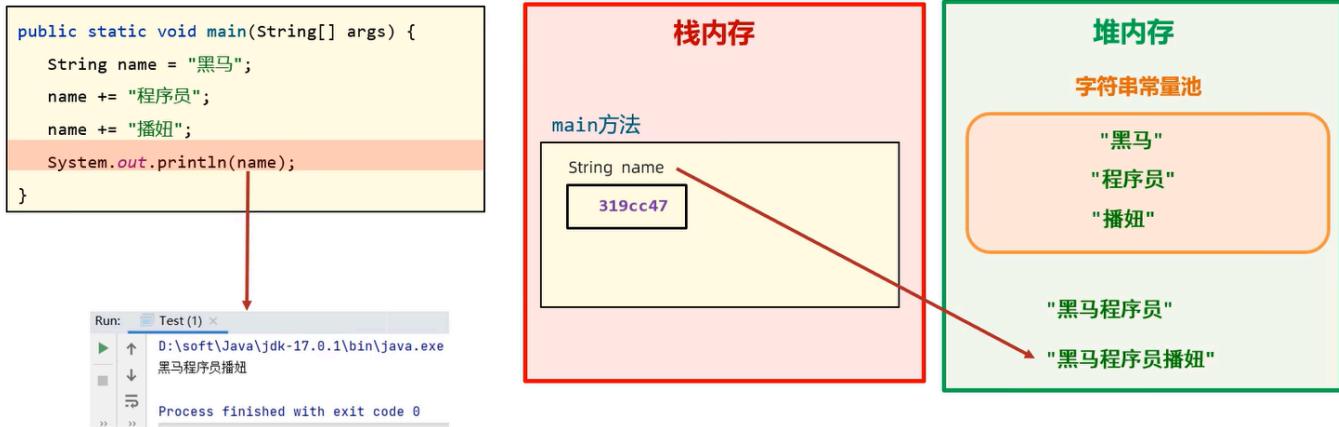
```
1 | String info = "这个电影简直是个垃圾，垃圾电影！！";
2 | String rs = info.replace("垃圾","**");
3 | System.out.println(rs);
```

- 控制台输出结果：

```
1 | 这个电影简直是个**, **电影！！
```

## 5.6 注意事项

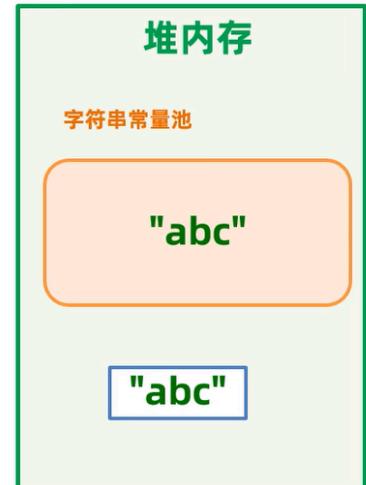
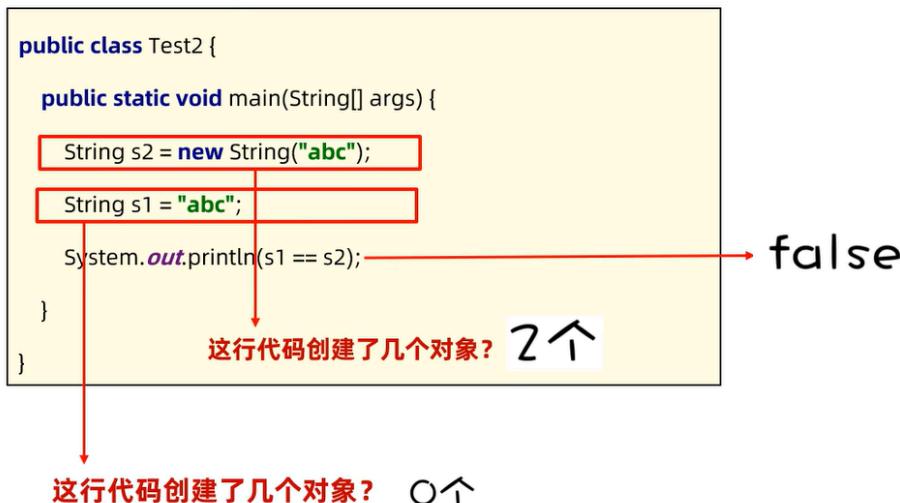
1. `String` 对象的内容不可改变，被称为不可变字符串对象。



每次试图改变 `String` 对象时，实际上产生了新的 `String` 对象，旧的 `String` 对象的内容并没有改变，编译器只是将变量指向的 `String` 对象由旧对象修改为新对象。

2.只要是以 `"..."` (包括 `new String("...")`) 方式写出的字符串对象，都会存放到堆内存中的字符串常量池，且相同内容的字符串只存储一份（节约内存），即它们的地址是一样的。

若通过 `String str = new String("...")` 来创建字符串对象，则每 `new` 一次都会产生一个新的对象存放于堆内存中。



3.编译优化机制：

```
public class Test4 {  
    public static void main(String[] args) {  
        String s1 = "abc";  
        String s2 = "a" + "b" + "c";  
        System.out.println(s1 == s2);  
    }  
}
```

→ true

Java存在编译优化机制，程序在编译时：“a” + “b” + “c” 会直接转成 “abc”，以提高程序的执行性能

## 6. ArrayList

### 6.1 构造器

序号	构造器	说明
01	<code>public ArrayList()</code>	构造一个初始容量为10的空集合，后续会随需求自动扩容。
02	<code>public ArrayList(int initialCapacity)</code>	构造一个具有指定初始容量的空集合，后续会随需求自动扩容。

## 6.2 常用方法

序号	方法	说明
01	<code>boolean add(E e)</code>	将指定元素添加到集合末尾。
02	<code>void add(int index,E element)</code>	在集合的指定索引处插入指定元素。
03	<code>E get(int index)</code>	获取集合指定索引处的元素并返回。
04	<code>int size()</code>	获取集合的元素个数并返回。
05	<code>E remove(int index)</code>	删除集合指定索引处的元素，返回被删除的元素。
06	<code>boolean remove(Object o)</code>	删除集合的指定元素（默认删除第一次出现的元素），若删除成功则返回 <code>true</code> 。
07	<code>E set(int index,E element)</code>	修改集合指定索引处的元素，返回被修改前的元素。

## 6.3 典型易错案例

### 6.3.1 概述

**需求：**假如购物车中存储了如下商品：Java入门、宁夏枸杞、黑枸杞、人字拖、特级枸杞、枸杞子。现在用户不想买枸杞了，选择了批量删除含有“枸杞”二字的商品，请完成该需求。

**分析：**

1. 使用 `ArrayList` 集合表示购物车，并存储以上商品。
2. 遍历集合中的所有元素，若某元素包含“枸杞”二字则删除它。
3. 打印集合到控制台以判断需求是否完成。

### 6.3.2 原代码分析

```
1 import java.util.ArrayList;
2
3 public class Test {
4     public static void main(String[] args) {
5         ArrayList<String> list = new ArrayList<>();
6         list.add("Java入门");
7         list.add("宁夏枸杞");
8         list.add("黑枸杞");
9         list.add("人字拖");
10        list.add("特级枸杞");
11        list.add("枸杞子");
12        System.out.println("原购物车: " + list);
13
14        for (int i = 0; i < list.size(); i++) {
15            String good = list.get(i);
16            if (good.contains("枸杞")) {
17                list.remove(good);
18            }
19        }
20        System.out.println("删除枸杞后的购物车: " + list);
21    }
22 }
```

控制台输出结果：

```
1 原购物车: [Java入门, 宁夏枸杞, 黑枸杞, 人字拖, 特级枸杞, 枸杞子]
2 删 除枸杞后的购物车: [Java入门, 黑枸杞, 人字拖, 枸杞子]
```

观察发现，我们的代码出现了bug，“黑枸杞”和“枸杞子”这两个商品并没有删除掉，具体分析如下：

```
1 i=0时, list[i]为"Java入门", 不含"枸杞"
2 购物车仍为: [Java入门, 宁夏枸杞, 黑枸杞, 人字拖, 特级枸杞, 枸杞子]
3 -----
4 i=1时, list[i]为"宁夏枸杞", 含有"枸杞", 故从集合中删除掉
5 购物车变为: [Java入门, 黑枸杞, 人字拖, 特级枸杞, 枸杞子]
6 注意: 此时"Java入门"后面的元素往前移动了一格, 索引也随之减少了1, 这就是bug的产生原因。
7 -----
8 i=2时, list[i]为"人字拖", 不含"枸杞"
```

```
9 购物车仍为: [Java入门, 黑枸杞, 人字拖, 特级枸杞, 枸杞子]  
10 注意: "黑枸杞"这个元素由于索引变化而被我们的程序忽略了, 由此产生了bug。  
11 -----  
12 i=3时, list[i]为"特级枸杞", 含有"枸杞", 故从集合中删除掉  
13 购物车变为: [Java入门, 黑枸杞, 人字拖, 枸杞子]  
14 -----  
15 i=4时, list.size()已经缩减为4, 二者相等, 循环结束。
```

### 6.3.3 debug

#### 方式一：每删除一个元素则索引减1

```
1 for (int i = 0; i < list.size(); i++) {  
2     String good = list.get(i);  
3     if (good.contains("枸杞")) {  
4         list.remove(good);  
5         i--;  
6     }  
7 }
```

#### 方式二：倒序遍历并删除

```
1 for (int i = list.size()-1; i >= 0; i--) {  
2     String good = list.get(i);  
3     if (good.contains("枸杞")) {  
4         list.remove(good);  
5     }  
6 }
```

## 7. Object

### 7.1 概述

`Object` 类是Java中所有类的祖宗类，因此Java中所有类的对象都可以直接使用 `Object` 类中提供的一些方法。

### 7.2 常用方法

序号	方法	说明
01	<code>public String toString()</code>	返回对象的字符串表示形式，实际开发中主要交给子类重写，以便打印出对象的具体内容而非地址。
02	<code>public boolean equals(Object o)</code>	比较两个对象是否相等（默认比较地址），实际开发中主要交给子类重写，以便子类自定义比较规则。
03	<code>protected Object clone()</code>	克隆对象。由于 <code>protected</code> ，该方法只能在 <code>Object</code> 类所处包下的其他类中使用，子类若想使用则必须重写该方法。

注意：以上3个方法重写都可以通过IDEA快捷生成。

### 7.3 03方法讲解

#### 7.3.1 注意事项

**Tip1：**若想使用该方法，则这个类**必须重写该方法并且实现 `Cloneable` 接口**，否则报错。

`Cloneable` 接口源码：

© Student.java

© Cloneable.java ×

© Test.java

```
1 > /.../  
25  
26 package java.lang;  
27
```

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

By convention, classes that implement this interface should override `Object.clone` (which is protected) with a public method. See `Object.clone()` for details on overriding this method.

Note that this interface does *not* contain the `clone` method. Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.

Since: 1.0

See Also: `CloneNotSupportedException`,  
`Object.clone()`



```
52 © public interface Cloneable {  
53 }
```

可以发现，该接口中什么内容都没有，这种接口称为**标记接口**，只有这样Java虚拟机才能识别并赋予这个类克隆对象的能力。

**Tip2：**此外还必须在 `main` 方法开头抛出 `CloneNotSupportedException` 异常，否则依旧报错。

```
1 public class Test {  
2     public static void main(String[] args) throws  
3         CloneNotSupportedException {  
4             Student s1 = new Student("zsh", 20);  
5             Student s2 = (Student) s1.clone(); //记得进行类型转换  
6 }
```

### 7.3.2 浅拷贝与深拷贝

#### 💡 Tip

也叫浅克隆与深克隆。

- 浅拷贝：拷贝出的对象与原对象中的数据一模一样（引用类型数据拷贝的只是地址）。

## 堆内存



- 深拷贝：

- 对象中的基本类型数据直接拷贝。
- 对象中的字符串数据拷贝的还是地址。
- 对象中的其他引用类型数据不会拷贝地址，而会创建新对象。

## 堆内存



## 8. Objects

### 8.1 概述

`Objects` 类是一个工具类，提供了很多操作对象的静态方法给我们使用，可以直接用 `Objects.类名` 的方式调用。

### 8.2 常用方法

序号	方法	说明
01	<code>static boolean equals(Object a, Object b)</code>	先做非空判断，再比较两个对象是否相等。
02	<code>static boolean isNull(Object obj)</code>	判断对象是否为空，是返回 <code>true</code> 。
03	<code>static boolean nonNull(Object obj)</code>	判断对象是否为非空，是返回 <code>true</code> 。

### 8.3 01方法讲解

思考：`String` 已经提供了 `equals` 方法用于比较两个对象是否相等，为什么官方更推荐使用 `Objects.equals` 方法呢？

来看下面的例子：

```
1 import java.util.Objects;
2
3 public class Test {
4     public static void main(String[] args) {
5         String s1 = null;
6         String s2 = "itheima";
7
8         System.out.println(s1.equals(s2));
9     }
10 }
```

控制台输出结果：

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "String.equals(Object)" because "s1" is null
at api_objects.Test.main(Test.java:10)

Process finished with exit code 1
```

可以发现，代码产生了 `NullPointerException`，即空指针异常。

现在，我们将代码第10行修改为 `System.out.println(Objects.equals(s1, s2));`，再来看控制台输出结果：

```
false

Process finished with exit code 0
```

可以发现，控制台能够正常输出，说明代码已经没有bug了。

debug原因参见源码：

```
public static boolean equals( @Nullable Object a,  @Nullable Object b) {
    return (a == b) || (a != null && a.equals(b));
}
```

**总结：**使用 `Objects.equals` 方法能够避免入参中有 `null` 而产生空指针异常的问题，程序健壮性更好。

# 9. 包装类

## 9.1 概述

包装类用于把基本数据类型包装成对象，从而实现Java“万物皆对象”的理念，同时也能更好地支持泛型。

## 9.2 基本数据类型对应的包装类

基本数据类型	对应的包装类（引用数据类型）
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean

## 9.3 Integer

### 9.3.1 包装方法

方法	说明
<code>static Integer valueOf(int i)</code>	将 int 类型的数据转换成 Integer 对象。

示例：`Integer a = Integer.valueOf(12);`

### 9.3.2 自动装箱与自动拆箱机制

- 自动装箱：可以自动把基本数据类型转换成对应的包装类对象。
  - 示例：`Integer a1 = 12;`
- 自动拆箱：可以自动把包装类对象转换成对应的基本数据类型。
  - 示例：

```
1 | Integer a2 = 12;  
2 | int a3 = a2; // 自动拆箱
```

### 9.4 包装类的常见操作（以 `Integer` 为例）

- 可以把基本数据类型转换成字符串类型：
  - `public static String toString(double d)`
  - `public String toString()`
- 可以把字符串类型的数值转换成数值本身对应的数据类型：
  - `public static int parseInt(String s)`
  - `public static Integer valueOf(String s)`

---

## 10. `StringBuilder` 与 `StringBuffer`

### 10.1 概述

- `StringBuilder` 代表可变字符串对象，相当于一个容器，里面装的字符串是可以改变的。
- `StringBuilder` 就是用来操作字符串的。
- 优点：`StringBuilder` 比 `String` 更适合做字符串的修改操作，效率更高，代码更简洁。

## 10.2 构造器

序号	构造器	说明
01	<code>public StringBuilder()</code>	创建一个初始容量为16个字符的可变字符串对象，不含任何字符。
02	<code>public StringBuilder(String str)</code>	创建一个初始含有指定字符串内容的可变字符串对象。

## 10.3 常用方法

序号	方法	说明
01	<code>StringBuilder append(任意类型)</code>	添加数据并返回 <code>StringBuilder</code> 对象本身，支持链式编程。
02	<code>StringBuilder reverse()</code>	反转对象内容。
03	<code>int length()</code>	返回对象内容长度（字符数）。
04	<code>String toString()</code>	将 <code>StringBuilder</code> 转换成 <code>String</code> 。

## 10.4 使用 `StringBuilder` 操作字符串的优势

### P1 原代码

```
1 public class Test2 {  
2     public static void main(String[] args) {  
3         String rs = "";  
4         for (int i = 0; i < 1000000; i++) {  
5             rs += "abc";  
6         }  
7         System.out.println(rs);  
}
```

8 | }  
9 | }

运行此代码后，控制台迟迟未输出结果，只能手动终止，证明以上代码运行效率极低。

```
D:\CS-Softwares\JDK\JDK21\bin\java.exe "-javaagent:D:\CS-  
Process finished with exit code 130
```

## P2 使用 `StringBuilder` 优化后的代码

```
1 public class Test2 {  
2     public static void main(String[] args) {  
3         StringBuilder stringBuilder=new StringBuilder();  
4         for (int i = 0; i < 1000000; i++) {  
5             stringBuilder.append("abc");  
6         }  
7         System.out.println(stringBuilder);  
8     }  
9 }
```

运行此代码后，控制台只用了1~2秒便输出结果，效率大大提高。

P3 分析及启示

- 分析：参见 [5.6 注意事项](#)。
  - `String` 是不可变字符串对象，也就是说它一旦被赋值就不可改变。每次试图改变 `String` 对象时，实际上产生了新的 `String` 对象，旧的 `String` 对象的内容并没有改变，编译器只是将变量指向的 `String` 对象由旧对象修改为新对象，因此原代码在 `for` 循环中每运行一次就会创建一个新的 `String` 对象，这大大增加了内存开销，导致运行效率极低。
  - 而 `StringBuilder` 是可变字符串对象，这意味着在 `for` 循环中无论运行多少次，变量指向的都是最初的 `StringBuilder` 对象，内存开销减少，效率自然就上来了。

- 启示：

- 如果需要频繁拼接或修改字符串，建议使用 `StringBuilder`，效率会更高！
- 如果对字符串操作较少，或者不需要操作字符串，只是为了定义字符串变量，则建议使用 `String`。

## 10.5 `StringBuffer`

- `StringBuffer` 和 `StringBuilder` 的用法完全一致。
- 区别： `StringBuffer` 是线程安全的， `StringBuilder` 是线程不安全的。

## 10.6 综合案例

- 需求：设计一个方法，将一个整型数组的内容输出为指定格式的字符串，形如 `[11, 22, 33]`。

- 代码：

```
1 public class Test3 {  
2     public static void main(String[] args) {  
3         System.out.println(transferArrayToFormattedString(new int[]  
4             {11, 22, 33}));  
5     }  
6     public static String transferArrayToFormattedString(int[]  
7         array) {  
8         // 1. 非空校验  
9         if (array == null) {  
10             return null;  
11         }  
12         // 2. 若array非空，则使用StringBuilder将其内容输出为指定格式的字符  
13         StringBuilder stringBuilder = new StringBuilder();  
14         stringBuilder.append("[");  
15         // 遍历到倒数第二个元素即可，否则最后一个元素末尾也会拼接"，"  
16         for (int i = 0; i < array.length - 1; i++) {  
17             stringBuilder.append(array[i]).append(", ");  
18         }  
19     }  
20 }
```

```

19         stringBuilder.append(array[array.length-1]).append("]");
20
21     // 3.将StringBuilder对象转换为String对象再返回
22     return stringBuilder.toString();
23 }
24 }
```

- **控制台：**

```
D:\CS-Softwares\JDK\JDK21\bin\java.exe "-javaagent:D:\CS-Soft
[11, 22, 33]

Process finished with exit code 0
```

## 11. StringJoiner

### 11.1 概述

- **JDK8** 开始才推出的新 **API**，和 **StringBuilder** 一样也是用来操作字符串的，也可以看作是一个容器，创建之后内容可变。
- **优势：**不仅能提高字符串的操作效率，而且在某些场景下使用它操作字符串，**代码会更简洁**，参见**11.4**

### 11.2 构造器

序号	构造器	说明
01	<code>public StringJoiner(间隔符)</code>	创建一个 <b>StringJoiner</b> 对象，指定拼接时的间隔符。
02	<code>public StringJoiner(间隔符, 开始符, 结束符)</code>	创建一个 <b>StringJoiner</b> 对象，指定拼接时的间隔符、开始符和结束符。

## 11.3 常用方法

序号	常用方法	说明
01	<code>StringJoiner add(CharSequence newElement)</code>	拼接字符序列，并返回对象本身。
02	<code>int length()</code>	返回对象内容长度（字符数）。
03	<code>String toString()</code>	返回拼接之后的字符串。

## 11.4 使用 `StringJoiner` 重新实现综合案例 10.6

- 代码：

```
1 import java.util.StringJoiner;
2
3 public class Test3 {
4     public static void main(String[] args) {
5         System.out.println(transferArrayToFormattedString(new int[]
6             {11, 22, 33}));
7     }
8
9     public static String transferArrayToFormattedString(int[]
10    array){
11         // 1.非空校验
12         if (array == null) {
13             return null;
14         }
15         // 2.若array非空，则使用StringJoiner将其内容输出为指定格式的字符
16         // 串
17         StringJoiner stringJoiner=new StringJoiner(", ", "[", "]");
18         for (int i = 0; i < array.length; i++) {
19             stringJoiner.add(String.valueOf(array[i]));
20         }
21         // 3.将StringJoiner对象转换为String对象再返回
22         return stringJoiner.toString();
23     }
24 }
```

```
22      }
23 }
```

- 控制台：

```
D:\CS-Softwares\JDK\JDK21\bin\java.exe "-javaagent:D:\CS-Softwares\IntelliJ IDEA 2021.3.1\lib\idea_rt.jar=55003,D:\CS-Softwares\IntelliJ IDEA 2021.3.1\bin" -Dfile.encoding=UTF-8 -classpath "D:\CS-Softwares\Java\MyFirstJava\MyFirstJava\out\production\MyFirstJava" com.itheima.Test
[11, 22, 33]
```

```
Process finished with exit code 0
```

## 12. **Math**

### 12.1 概述

- **Math** 即数学类，是一个工具类，里面提供的都是对数据进行操作的一些静态方法。

### 12.2 常用方法

序号	方法	说明
01	<code>static int abs(int a)</code>	返回a的绝对值
02	<code>static double ceil(double a)</code>	对a向上取整
03	<code>static double floor(double a)</code>	对a向下取整
04	<code>static int round(float a)</code>	对a四舍五入
05	<code>static int max(int a, int b)</code>	返回a和b中的较大值
06	<code>static double pow(double a, double b)</code>	返回 $a^b$
07	<code>static double random()</code>	返回一个 <code>double</code> 类型的 <b>伪随机数</b> ，范围是[0.0, 1.0)

## 13. System

### 13.1 概述

- System 指代程序所在的系统，也是一个工具类。

### 13.2 常用方法

序号	方法	说明
01	<code>static void exit(int status)</code>	退出当前运行的JVM，status 取0时表示正常退出，取其它值表示异常退出。
02	<code>static long currentTimeMills()</code>	以毫秒形式返回当前系统时间戳。

## 14. Runtime

### 14.1 概述

- Runtime 指代程序所在的运行环境，分析源码可知它是一个单例类。

```
public class Runtime {  
    private static final Runtime currentRuntime = new Runtime();
```

```
    private static Version version;
```

Returns the runtime object associated with the current Java application. Most of the methods of class `Runtime` are instance methods and must be invoked with respect to the current runtime object.

Returns: the `Runtime` object associated with the current Java application.

```
    public static Runtime getRuntime() { return currentRuntime; }
```

Don't let anyone else instantiate this class

```
    private Runtime() {}
```

## 14.2 常用方法

序号	方法	说明
01	<code>static Runtime getRuntime()</code>	返回与当前Java应用程序关联的运行时对象。
02	<code>void exit(int status)</code>	退出当前运行的JVM。
03	<code>int availableProcessors()</code>	返回JVM可用的处理器数量。
04	<code>long totalMemory()</code>	返回JVM中的内存总量（单位：字节）。
05	<code>long freeMemory()</code>	返回JVM中的可用内存量（单位：字节）。
06	<code>Process exec(String command)</code>	执行某个进程，并返回代表该进程的对象。

## 15. **BigDecimal**

### 15.1 引入背景

**BigDecimal** 是为了解决浮点数运算时结果失真的问题。

首先编写以下测试代码：

```
1 public class Test1 {  
2     public static void main(String[] args) {  
3         double a = 0.1;  
4         double b = 0.2;  
5         System.out.println(a + "+" + b + "=" + (a + b));  
6     }  
7 }
```

代码运行后控制台输出结果与我们的常识不符：

```
D:\CS-Softwares\JDK\JDK21\bin\java.exe "-javaagent:  
0.1+0.2=0.30000000000000004  
  
Process finished with exit code 0
```

这是由于浮点数精度是有限的，因为0.1和0.2是二进制下的无限循环小数，所以它们转换成IEEE754双精度浮点数后势必丢失精度，因此结果转换回十进制会不精确。

### 15.2 底层原理：为什么 **BigDecimal** 是精确的？

**BigDecimal** 根本不把数字存储为二进制小数。它直接存储十进制数字的每一位（作为整数），并记录小数点位置。所有的计算都是基于整数运算完成的，因此可以精确表示像 0.1 和 0.2 这样的数。

## 15.3 构造器

序号	构造器	说明
01	<code>public BigDecimal(double val)</code>	将 <code>double</code> 类型变量转换为 <code>BigDecimal</code> 类型变量, 不推荐使用。
02	<code>public BigDecimal(String val)</code>	将 <code>String</code> 类型变量转换为 <code>BigDecimal</code> 类型变量。

## 15.4 常用方法

序号	方法	说明
01	<code>static BigDecimal valueOf(double val)</code>	将 <code>double</code> 类型变量转换为 <code>BigDecimal</code> 类型变量, 推荐使用。
02	<code>BigDecimal add(BigDecimal b)</code>	加
03	<code>BigDecimal subtract(BigDecimal b)</code>	减
04	<code>BigDecimal multiply(BigDecimal b)</code>	乘
05	<code>BigDecimal divide(BigDecimal b)</code>	除
06	<code>BigDecimal divide(BigDecimal b, 精确几位, 舍入模式)</code>	除, 可以控制精确到小数点后几位和采用何种舍入模式。
07	<code>double doubleValue()</code>	将 <code>BigDecimal</code> 类型变量转换为 <code>double</code> 类型变量。

## 15.5 使用 `BigDecimal` 重构测试代码

- 代码：

```
1 public class Test1 {  
2     public static void main(String[] args) {  
3         double a = 0.1;  
4         double b = 0.2;  
5         BigDecimal a1 = BigDecimal.valueOf(a);  
6         BigDecimal b1 = BigDecimal.valueOf(b);  
7         System.out.println(a + "+" + b + "=" + a1.add(b1));  
8     }  
9 }
```

- 控制台：输出了正确结果，没有发生精度丢失。

```
D:\CS-Softwares\JDK\JDK21\bin\java.exe "-javaagent:D:\  
0.1+0.2=0.3  
  
Process finished with exit code 0
```

---

## 16.日期与时间

### 16.1 `JDK8` 之前传统的日期与时间

### 16.2 `JDK8` 开始新增的日期与时间