

Tarena Teaching System

# JAVA核心API（上）

## 学员用书



Java企业应用及互联网  
高级工程师培训课程

# 目 录

<b>Unit01</b> .....	<b>1</b>
1. API 文档 .....	3
1.1. JDK API.....	3
1.1.1. 什么是 JDK API .....	3
1.1.2. JDK 包结构 .....	3
1.2. 文档注释规范 .....	3
1.2.1. 文档注释.....	3
1.2.2. 文档注释规范 .....	4
1.2.3. javadoc 命令生成文档 .....	4
2. 字符串基本操作 .....	5
2.1. String 及其常用 API .....	5
2.1.1. String 是不可变对象.....	5
2.1.2. String 常量池 .....	5
2.1.3. 内存编码及长度 .....	6
2.1.4. 使用 indexOf 实现检索 .....	6
2.1.5. 使用 substring 获取子串 .....	7
2.1.6. trim ( 查阅 API ) .....	7
2.1.7. charAt ( 查阅 API ) .....	8
2.1.8. startsWith 和 endsWith ( 查阅 API ) .....	8
2.1.9. 大小写变换 ( 查阅 API ) .....	8
2.1.10. valueOf ( 查阅 API ) .....	9
2.2. StringBuilder 及其 API .....	9
2.2.1. StringBuilder 封装可变字符串 .....	9
2.2.2. StringBuilder 常用方法 .....	9
2.2.3. StringBuilder .....	10
2.2.4. append 方法 ( 查阅 API ) .....	10
2.2.5. insert ( 查阅 API ) .....	10
2.2.6. delete ( 查阅 API ) .....	11
2.2.7. StringBuilder 总结 .....	11
经典案例 .....	12
1. 规范注释及文档生成 ( Eclipse ) .....	12
2. 测试 String 常量池 .....	15
3. 获取 String 对象的长度 .....	22
4. 在一个字符串中检索另外一个字符串 .....	24
5. 在一个字符串中截取指定字符串 .....	28

---

6. 去掉一个字符串的前导和后继空字符 .....	30
7. 遍历一个字符串中的字符序列 .....	32
8. 检测一个字符串是否以指定字符串开头或结尾.....	34
9. 转换字符串中英文字母的大小写形式 .....	37
10. 将其他类型转换为字符串类型 .....	39
11. 测试 StringBuilder 的 append 方法 .....	41
12. 测试 StringBuilder 的 insert 方法 .....	44
13. 测试 StringBuilder 的 delete 方法 .....	47
课后作业 .....	49
<b>Unit02</b> .....	51
1. 正则表达式 .....	53
1.1. 基本正则表达式 .....	53
1.1.1. 正则表达式简介 .....	53
1.1.2. 分组 “()” .....	54
1.1.3. “^” 和 “\$” .....	55
1.2. String 正则 API .....	55
1.2.1. matches 方法 .....	55
1.2.2. split 方法 .....	56
1.2.3. replaceAll 方法 .....	56
2. Object .....	57
2.1. Object.....	57
2.1.1. Object .....	57
2.2. toString 方法 .....	57
2.2.1. 如何重写 toString 方法 .....	57
2.2.2. String 重写 toString 方法 .....	58
2.3. equals 方法 .....	58
2.3.1. equals 方法 .....	58
2.3.2. 如何重写 equals 方法 .....	59
2.3.3. String 重写 equals 方法 .....	59
2.3.4. equals 和==的区别 .....	59
3. 包装类 .....	60
3.1. 包装类概述 .....	60
3.1.1. 包装类概述 .....	60
3.2. 8 种基本类型包装类 .....	60
3.2.1. Number 及其主要方法 .....	60
3.2.2. Integer 常用功能 .....	61
3.2.3. Double 常用功能 .....	62
3.2.4. 自动装箱和拆箱操作 .....	62
经典案例 .....	64

1. 编写验证 Email 的正则表达式 .....	64
2. 使用 split 方法拆分字符串 .....	66
3. 使用 replaceAll 实现字符串替换 .....	68
4. 重写 Cell 类的 toString 方法 .....	70
5. 重写 Cell 类的 equals 方法 .....	73
6. 测试 Number 的 intValue 和 doubleValue 方法 .....	76
7. 测试 Integer 的 parseInt 方法 .....	80
8. 测试 Double 的 parseDouble 方法 .....	83
课后作业 .....	86
<b>Unit03 .....</b>	<b>88</b>
1. 日期操作 .....	90
1.1. Date 及其常用 API .....	90
1.1.1. Java 中的时间 .....	90
1.1.2. Date 类简介 .....	90
1.1.3. setTime 和 getTime 方法 .....	90
1.1.4. Date 重写 toString 方法 .....	91
1.2. SimpleDateFormat .....	91
1.2.1. SimpleDateFormat 简介 .....	91
1.2.2. 日期模式匹配字符 .....	92
1.2.3. 将 Date 格式化为 String .....	92
1.2.4. 将 String 解析为 Date .....	92
1.3. Calendar .....	93
1.3.1. Calendar 简介 .....	93
1.3.2. getInstance 方法 .....	93
1.3.3. 设置日期及时间分量 .....	94
1.3.4. 获取日期及时间分量 .....	94
1.3.5. getActualMaximum 方法 .....	95
1.3.6. add 方法 .....	95
1.3.7. setTime 与 getTime 方法 .....	95
2. 集合框架 .....	96
2.1. Collection .....	96
2.1.1. List 和 Set .....	96
2.1.2. 集合持有对象的引用 .....	96
2.1.3. add 方法 .....	97
2.1.4. contains 方法 .....	97
2.1.5. size、clear、isEmpty .....	98
经典案例 .....	99
1. 使用 setTime 和 getTime 方法操作时间毫秒 .....	99
2. 使用 format 方法格式化日期 .....	102

---

3. 使用 parse 方法将字符串解析为日期 .....	104
4. 测试 getInstance 方法 .....	106
5. 调用 set 方法设置日期分量 .....	111
6. 调用 get 方法获取日期分量 .....	113
7. 输出某一年的各个月份的天数 .....	116
8. 输出一年后再减去 3 个月的日期 .....	118
9. 使 Date 表示的日期与 Calendar 表示的日期进行互换 .....	120
10. 测试集合持有对象 .....	122
11. 使用 add 方法向集合中添加元素 .....	126
12. 使用 contains 方法判断集合中是否包含某元素 .....	129
13. 测试方法 size、clear、isEmpty 的用法 .....	133
课后作业 .....	138
<b>Unit04</b> .....	140
1. 集合框架 .....	142
1.1. Collection .....	142
1.1.1. addAll、containsAll .....	142
1.2. Iterator .....	142
1.2.1. hasNext、next 方法 .....	142
1.2.2. remove 方法 .....	143
1.2.3. 增强型 for 循环 .....	144
1.3. 泛型机制 .....	144
1.3.1. 泛型在集合中的应用 .....	144
2. 集合操作——线性表 .....	145
2.1. List .....	145
2.1.1. ArrayList 和 LinkedList .....	145
2.1.2. get 和 set .....	146
2.1.3. 插入和删除 .....	146
2.1.4. subList .....	147
2.1.5. List 转换为数组 .....	148
2.1.6. 数组转换为 List .....	148
2.2. List 排序 .....	149
2.2.1. Collections.sort 方法实现排序 .....	149
2.2.2. Comparable .....	150
2.2.3. Comparator .....	150
2.3. 队列和栈 .....	151
2.3.1. Queue .....	151
2.3.2. Deque .....	152
经典案例 .....	153
1. 测试方法 addAll、containsAll 的用法 .....	153
2. 使用 Iterator 的 hasNext 方法、next 方法遍历集合 .....	157
3. 使用 Iterator 的 remove 方法移除元素 .....	160

---

4. 使用增强型 for 循环遍历集合 .....	163
5. 测试 List 的 get 方法和 set 方法 .....	165
6. 测试向 List 中插入和删除元素 .....	170
7. 测试 List 的 subList 方法 .....	174
8. 将 List 转换为数组 .....	179
9. 将数组转换为 List .....	181
10. 使用 Collections.sort 方法实现排序 .....	185
11. 使用 Comparator 接口实现排序 .....	188
12. 测试 Queue 的用法 .....	191
13. 测试 Deque 的用法 .....	195
课后作业 .....	199
<b>Unit05 .....</b>	<b>202</b>
1. 查询表 .....	204
1.1. Map 接口 .....	204
1.1.1. Map 接口 .....	204
1.1.2. put()方法 .....	204
1.1.3. get()方法 .....	205
1.1.4. containsKey()方法 .....	205
1.2. HashMap .....	206
1.2.1. Hash 表原理 .....	206
1.2.2. hashCode 方法 .....	206
1.2.3. 重写 hashCode 方法 .....	207
1.2.4. 装载因子及 HashMap 优化 .....	207
1.3. Map 的遍历 .....	207
1.3.1. 使用 keyset()方法 .....	207
1.3.2. 使用 entryset()方法 .....	208
1.4. 有序的 Map .....	208
1.4.1. LinkedHashMap 实现有序 Map .....	208
2. 文件操作—File .....	208
2.1. 创建 File 对象 .....	208
2.1.1. File(String pathname) .....	208
2.1.2. File(File parent, String child) .....	209
2.1.3. isFile()方法 .....	210
2.2. File 表示文件信息 .....	210
2.2.1. length()方法 .....	210
2.2.2. exists()方法 .....	211
2.2.3. createNewFile()方法 .....	211
2.2.4. delete()方法 .....	212
2.2.5. isDirectory()方法 .....	212

---

2.3. File 表示目录信息 .....	213
2.3.1. mkdir()方法 .....	213
2.3.2. mkdirs()方法 .....	214
2.3.3. delete()方法 .....	214
经典案例 .....	215
1. PM2.5 监控程序——统计各点 PM2.5 最大值.....	215
2. hashCode 方法的重写 .....	218
3. PM2.5 监控程序——遍历各点 PM2.5 最大值.....	225
4. PM2.5 监控程序——统计并遍历各点 PM2.5 最大值（要求顺序） .....	230
5. 查看一个文件的大小.....	232
6. 创建一个空文件 .....	233
7. 删除一个文件 .....	235
8. 创建一个目录 .....	236
9. 创建一个多级目录 .....	238
10. 删除一个目录 .....	239
课后作业 .....	241

# Java 核心 API (上)

## Unit01

知识体系.....**Page 3**

API 文档	JDK API	什么是 JDK API
		JDK 包结构
	文档注释规范	文档注释
		文档注释规范
		javadoc 命令生成文档
字符串基本操作	String 及其常用 API	String 是不可变对象
		String 常量池
		内存编码及长度
		使用 indexOf 实现检索
		使用 substring 获取子串
		trim ( 查阅 API )
		charAt ( 查阅 API )
		startsWith 和 endsWith ( 查阅 API )
		大小写变换 ( 查阅 API )
		valueOf ( 查阅 API )
	StringBuilder 及其 API	StringBuilder 封装可变字符串
		StringBuilder 常用方法
		StringBuilder
		append 方法 ( 查阅 API )
		insert ( 查阅 API )
		delete ( 查阅 API )
		StringBuilder 总结

经典案例.....**Page 12**

规范注释及文档生成 ( Eclipse )	什么是 JDK API
	JDK 包结构
	文档注释
	文档注释规范
	javadoc 命令生成文档

测试 String 常量池	String 常量池
获取 String 对象的长度	内存编码及长度
在一个字符串中检索另外一个字符串	使用 indexOf 实现检索
在一个字符串中截取指定字符串	使用 substring 获取子串
去掉一个字符串的前导和后继空字符	trim ( 查阅 API )
遍历一个字符串中的字符序列	charAt ( 查阅 API )
检测一个字符串是否以指定字符串开头或结尾	startsWith 和 endsWith ( 查阅 API )
转换字符串中英文字母的大小写形式	大小写变换 ( 查阅 API )
将其他类型转换为字符串类型	valueOf ( 查阅 API )
测试 StringBuilder 的 append 方法	append 方法 ( 查阅 API )
测试 StringBuilder 的 insert 方法	insert ( 查阅 API )
测试 StringBuilder 的 delete 方法	delete ( 查阅 API )

课后作业.....Page 49

## 1. API 文档

### 1.1. JDK API

#### 1.1.1. 【JDK API】什么是 JDK API

**什么是JDK API**

**Tarena**  
达内科技

知识讲解

- JDK中包含大量的API类库，所谓API ( Application Programming Interface , 应用程序编程接口 ) 就是一些已写好、可供直接调用的功能 ( 在Java语言中，这些功能以类的形式封装 )。
- JDK API 包含的类库功能强大，经常使用的有：字符串操作、集合操作、文件操作、输入输出操作、网络操作、多线程等等。

+

#### 1.1.2. 【JDK API】JDK 包结构

**JDK包结构**

**Tarena**  
达内科技

知识讲解

- 为了便于使用和维护，JDK类库按照包结构划分，不同功能的类划分在不同的包中；
- 经常使用的包如下表所示：

包	功能
java.lang	Java程序的基础类，如字符串、多线程等，该包中的类使用的频率非常高，不需要import，可以直接使用
java.util	常用工具类，如集合、随机数生成器、日历、时钟等
java.io	文件操作、输入/输出操作
java.net	网络操作
java.math	数学运算相关操作
java.security	安全相关操作
java.sql	数据库访问
java.text	处理文字、日期、数字、信息的格式

+

### 1.2. 文档注释规范

#### 1.2.1. 【文档注释规范】文档注释

**文档注释**

**Tarena**  
达内科技

知识讲解

- 以 `/**` 开始，以 `*/` 结束；
- 加在类和方法的开头，用于说明作者，时间，版本，要实现功能的详细描述等信息；
- 通过javadoc工具，可以轻松的将此注释转换为HTML文档说明；学习者和程序员主要通过文档了解API的功能；
- 文档注释不同于普通的注释 (`//...` 或 `/*...*/`)，普通注释写在程序之中，用于程序员进行代码维护和交流，无法通过工具生成文档；而文档注释 (`/**...*/`) 写在类和方法的开头，专门用于生成供API使用者进行参考的文档资料。

+

### 1.2.2. 【文档注释规范】文档注释规范

知识讲解

### 文档注释规范

```
/*
 * The <code>String</code> class represents character strings...
 * ...
 * @author Lee Boynton
 * @version 1.204, 06/09/06
 * @see java.lang.StringBuffer
 * @since JDK1.0
 */
public final class String
    implements java.io.Serializable, Comparable<String>,
    CharSequence {
    ...
}
```

**+**

知识讲解

### 文档注释规范 ( 续1 )

```
/*
 * ...
 * ...
 * @param charsetName
 *   The name of a supported
 * @return The resultant byte array
 * @throws UnsupportedEncodingException
 *   If the named charset is not supported
 *
 */
public byte[] getBytes(String charsetName) {
    ...
}
```

**+**

### 1.2.3. 【文档注释规范】javadoc 命令生成文档

知识讲解

### javadoc 命令生成文档

- 目标HTML文档存储到docDir下，执行以下步骤：
  - 切换到包含想要生成文档的源文件目录。如果有嵌套的包，则必须切换到包含子目录的目录
  - 如果是一个包，运行命令
  - javadoc -d docDir nameOfPackage
  - 多个包生成文档，运行
  - javadoc -d docDir nameOfPackage1,nameOfPackage2...
  - 如果文件在默认包中，运行
  - javadoc -d docDir \*.java

**+**

## 2. 字符串基本操作

### 2.1. String 及其常用 API

#### 2.1.1. 【String 及其常用 API】String 是不可变对象

知识讲解

#### String是不可变对象

• java.lang.String使用了final修饰，不能被继承；  
• 字符串底层封装了字符数组及针对字符数组的操作算法；  
• 字符串一旦创建，对象永远无法改变，但字符串引用可以重新赋值；  
• Java字符串在内存中采用Unicode编码方式，任何一个字符对应两个字节的定长编码。

+

#### 2.1.2. 【String 及其常用 API】String 常量池

知识讲解

#### String常量池

• Java为了提高性能，静态字符串（字面量/常量/常量连接的结果）在常量池中创建，并尽量使用同一个对象，重用静态字符串；  
• 对于重复出现的字符串直接量，JVM会首先在常量池中查找，如果存在即返回该对象。

+

知识讲解

#### String常量池（续1）

```
/** 测试String常量池 */
@Test
public void testConstantPool() {
    String str1 = "Hello";
    // 不会创建新的String对象，而是使用常量池中已有的"Hello".
    String str2 = "Hello";
    System.out.println(str1 == str2); // 输出？
    // 使用new关键字会创建新的String对象。
    String str3 = new String("Hello");
    System.out.println(str1 == str3); // 输出？
}
```

+

### 2.1.3. 【String 及其常用 API】内存编码及长度

知识讲解

#### 内存编码及长度

• String在内存中采用Unicode编码，每个字符占用两个字节；任何一个字符（无论中文还是英文）都算1个字符串长度，占用两个字节。

```
/** 获取String对象的长度 */
@Test
public void testLength(){
    String str1 = "Hello";
    System.out.println(str1.length());
    String str2 = "你好，String";
    System.out.println(str2.length());
}
```

+

### 2.1.4. 【String 及其常用 API】使用 indexOf 实现检索

知识讲解

#### 使用 indexOf 实现检索

• indexOf方法用于实现在字符串中检索另外一个字符串  
• String提供几个重载的indexOf方法

int indexOf (String str )	在字符串中检索str，返回其第一次出现的位置，如果找不到则返回-1
int indexOf ( String str, int fromIndex)	从字符串的 fromIndex 位置开始检索

• String还定义有 lastIndexOf 方法：

int lastIndexOf ( String str, int from )	str在字符串中多次出现时，将返回最后一个出现的位置
--	----------------------------

+

知识讲解

#### 使用 indexOf 实现检索（续1）

```
/** 在一个字符串中检索另外一个字符串 */
@Test
public void testIndexOf() {
    String str = "I can because i think i can";
    int index = str.indexOf("can");
    System.out.println(index);           // 2
    index = str.lastIndexOf("can");
    System.out.println(index);           // 24
    index = str.indexOf("can", 6);
    System.out.println(index);           // 24
    index = str.indexOf("Can");
    System.out.println(index);           // -1
}
```

+

### 2.1.5. 【String 及其常用 API】使用 substring 获取子串

**知识讲解**

#### 使用substring获取子串

• substring方法用于返回一个字符串的子字符串。  
 • substring常用重载方法定义如下：

String substring( int beginIndex, int endIndex)	返回字符串中从下标beginIndex（包括）开始到endIndex（不包括）结束的子字符串
String substring( int beginIndex)	返回字符串中从下标beginIndex（包括）开始到字符串结尾的子字符串

+

**知识讲解**

#### 使用substring获取子串（续1）

```
/** 在一个字符串中截取指定的字符串 */
@Test
public void testSubstring() {
    String str = "http://www.oracle.com";
    String subStr = str.substring(11, 17);
    System.out.println(subStr); // oracle

    subStr = str.substring(7);
    System.out.println(subStr); // www.oracle.com
}
```

+

### 2.1.6. 【String 及其常用 API】trim（查阅 API）

**知识讲解**

#### trim（查阅API）

```
/** 去掉一个字符串的前导和后继空字符 */
@Test
public void testTrim() {
    String userName = " good man ";
    userName = userName.trim();
    System.out.println(userName.length()); // 8
    System.out.println(userName); // good man
}
```

+

### 2.1.7. 【String 及其常用 API】charAt (查阅 API)

知识讲解

#### charAt ( 查阅API )

• String中定义有charAt ( ) 方法 :

char charAt (int index)	方法charAt ( ) 用于返回字符串指定位置的字符。参数index表示指定的位置
----------------------------	--

```
/** 遍历一个字符串中的字符序列 */
@Test
public void testCharAt() {
    String name = "Whatisjava?";
    for (int i = 0; i < name.length(); i++){
        char c = name.charAt(i);
        System.out.print(c + " ");
    }
} // W h a t i s j a v a ?
```

+

### 2.1.8. 【String 及其常用 API】startsWith 和 endsWith (查阅 API)

知识讲解

#### startsWith和endsWith ( 查阅API )

```
/** 检测一个字符串是否以指定字符串开头或结尾 */
@Test
public void testStartWithAndEndWith() {
    String str = "Thinking in Java";
    System.out.println(str.endsWith("Java")); // true
    System.out.println(str.startsWith("T")); // true
    System.out.println(str.startsWith("thinking")); // false
}
```

+

### 2.1.9. 【String 及其常用 API】大小写变换 (查阅 API)

知识讲解

#### 大小写变换 ( 查阅API )

```
/** 转换字符串中英文字母的大小写形式 */
@Test
public void testToUpperCaseAndLowerCase() {
    String str = "我喜欢Java";
    str = str.toUpperCase();
    System.out.println(str); // 我喜欢JAVA
    str = str.toLowerCase();
    System.out.println(str); // 我喜欢java
}
```

+

### 2.1.10. 【String 及其常用 API】valueOf (查阅 API)

知识讲解

---

---

---

---

---

**valueOf ( 查阅API )**

```
/** 将其他类型转换为字符串类型 */
public void testValueOf() {
    double pi = 3.1415926;    int value = 123;
    boolean flag = true;
    char[] charArr = { 'a', 'b', 'c', 'd', 'e', 'f', 'g' };
    String str = String.valueOf(pi);
    System.out.println(str);
    str = String.valueOf(value);
    System.out.println(str);
    str = String.valueOf(flag);
    System.out.println(str);
    str = String.valueOf(charArr);
    System.out.println(str);
}
```

+

## 2.2. StringBuilder 及其 API

### 2.2.1. 【StringBuilder 及其 API】StringBuilder 封装可变字符串

知识讲解

---

---

---

---

---

**StringBuilder封装可变字符串**

- StringBuilder封装可变的字符串，对象创建后可以通过调用方法改变其封装的字符序列。
- StringBuilder有如下常用构造方法：

```
public StringBuilder()
public StringBuilder( String str )
```

+

### 2.2.2. 【StringBuilder 及其 API】StringBuilder 常用方法

知识讲解

---

---

---

---

---

**StringBuilder常用方法**

StringBuilder类的常用方法	功能描述
StringBuilder append(String str)	追加字符串
StringBuilder insert (int dstOffset , String s)	插入字符串
StringBuilder delete(int start , int end)	删除字符串
StringBuilder replace(int start , int end , String str)	替换字符串
StringBuilder reverse()	字符串反转

+

### 2.2.3. 【StringBuilder 及其 API】StringBuilder

知识讲解

#### StringBuilder

- StringBuilder的很多方法的返回值均为StringBuilder类型。这些方法的返回语句均为：return this。
- 由于改变封装的字符序列后又返回了该对象的引用。可以按照如下简洁的方式书写代码：

```
buf.append("ibm").append("java")
      .insert(3, "oracle")
      .replace(9, 13, "JAVA");
System.out.println(buf.toString());
```

+

### 2.2.4. 【StringBuilder 及其 API】append 方法（查阅 API）

知识讲解

#### append方法（查阅API）

```
/** 测试StringBuilder的append方法 */
@Test
public void testAppend() {
    StringBuilder buf =
        new StringBuilder("Programming Language:");
    buf.append("java").append("cpp")
      .append("php")
      .append("c#")
      .append("objective-c");
    System.out.println(buf.toString());
}
```

+

### 2.2.5. 【StringBuilder 及其 API】insert（查阅 API）

知识讲解

#### insert（查阅API）

```
/** 测试StringBuilder的insert方法 */
@Test
public void testInsert() {
    StringBuilder buf = new StringBuilder
        ("javacppc#objective-c");
    buf.insert(9, "php");
    System.out.println(buf);
}
```

+

### 2.2.6. 【StringBuilder 及其 API】 delete (查阅 API)

#### delete ( 查阅API )



```
/** 测试StringBuilder的delete方法 */
@Test
public void testDelete() {
    StringBuilder buf = new StringBuilder
        ("javaoraclecppc#php");
    buf.delete(4, 4 + 6);
    System.out.println ( buf );
}
```

知识讲解



### 2.2.7. 【StringBuilder 及其 API】 StringBuilder 总结

#### StringBuilder 总结



- StringBuilder是可变字符串。字符串的内容计算，建议采用StringBuilder实现，这样性能会好一些；
- java的字符串连接的过程是利用StringBuilder实现的  

```
String s = "AB"; String s1 = s + "DE"+1;
String s1 = new StringBuilder(s).append("DE")
    .append(1).toString();
```
- StringBuffer 和StringBuilder
  - StringBuffer是线程安全的，同步处理的，性能稍慢
  - StringBuilder是非线程安全的，并发处理的，性能稍快

知识讲解



## 经典案例

### 1. 规范注释及文档生成 ( Eclipse )

- 问题

请看下列代码，其作用如下：

- 1) 类 JavadocTest 的作用是用于测试生成 Javadoc 文档；作者为 jessica；版本为 1.0。
- 2) 方法 hello 的作用为获取打招呼字符串 该方法的参数 name 表示指定向谁打招呼；该方法返回打招呼的字符串。

```
package day01;
public class JavadocTest {
    public String hello(String name) {
        return name + ",你好!";
    }
}
```

使用文档注释，为上述代码添加注释，并生成 Javadoc 文档。

- 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：为代码添加文档注释

根据问题的描述，使用/\*\* \*/为代码添加注释。代码如下所示：

```
package day01;

/**
 * <strong>JavadocTest</strong>类用于测试生成 Javadoc 文档
 *
 * @author jessica
 * @version 1.0
 */
public class JavadocTest {
    /**
     * 获取打招呼字符串
     *
     * @param name
     *          该参数指定向谁打招呼
     * @return 返回打招呼的字符串
     */
    public String hello(String name) {
        return name + ",你好!";
    }
}
```

## 步骤二：使用 Eclipse 生成文档注释

首先，点击 Project-->Generate Javadoc，如图-1 所示。

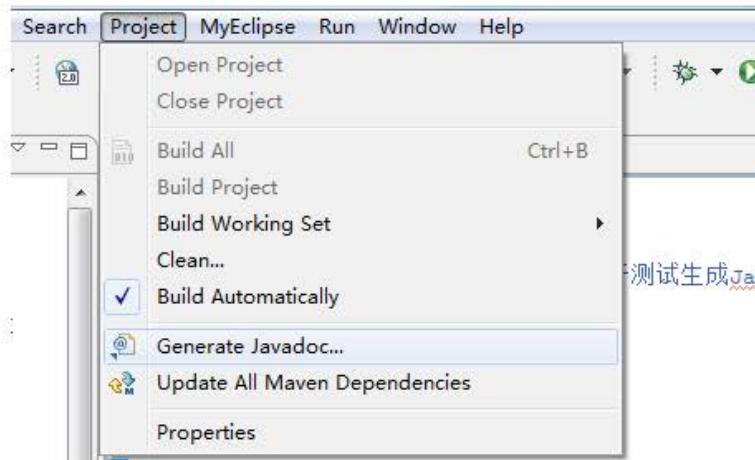


图- 1

点击 Generate Javadoc 以后，进入下一界面，选择你要生成 Javadoc 的包，或包下的类以及生成的 Javadoc 所存在的路径，默认生成到当前工程目录下，如图-2 所示。

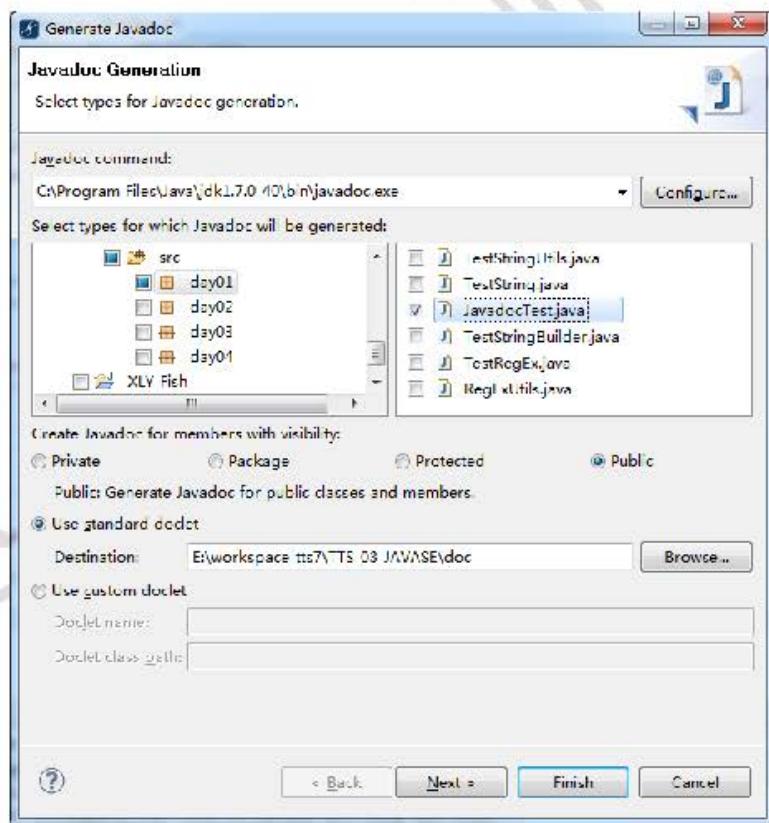


图- 2

点击 Finish，即可生成 Javadoc。生成的 Javadoc 文件如图-3 所示：

名称	修改日期	类型	大小
day01	2014/2/21 15:41	文件夹	
index-files	2014/2/21 15:41	文件夹	
resources	2014/2/21 15:41	文件夹	
allclasses-frame.html	2014/2/21 15:41	360 se HTML Do...	1 KB
allclasses-noframe.html	2014/2/21 15:41	360 se HTML Do...	1 KB
constant-values.html	2014/2/21 15:41	360 se HTML Do...	4 KB
deprecated-list.html	2014/2/21 15:41	360 se HTML Do...	4 KB
help-doc.html	2014/2/21 15:41	360 se HTML Do...	7 KB
index.html	2014/2/21 15:41	360 se HTML Do...	3 KB
overview-tree.html	2014/2/21 15:41	360 se HTML Do...	4 KB
package-list	2014/2/21 15:41	文件	1 KB
stylesheet.css	2014/2/21 15:41	层叠样式表文档	12 KB

图- 3

点击 index.html，查看具体内容，如图-4 所示。



图- 4

### • 完整代码

本案例中，添加注释后的完整代码如下所示：

```

package day01;

/**
 * <strong>JavadocTest</strong>类用于测试生成 Javadoc 文档
 *
 * @author jessica
 * @version 1.0
 */

public class JavadocTest {
    /**
     * 获取打招呼字符串
     *
     * @param name
     */

```

```
*      该参数指定向谁打招呼
* @return 返回打招呼的字符串
*/
public String hello(String name) {
    return name + ",你好!";
}
```

## 2. 测试 String 常量池

- 问题

在 Java 中，出于性能的考虑，JVM 会将字符串直接量对象缓存在常量池中；对于重复出现的字符串直接量，JVM 会首先在缓存池中查找，如果存在即返回该对象。

本案例要求使用 JUnit 构建测试方法，测试 String 常量池的特点。即，首先，使用直接量“字符序列”的方式创建两个字符串对象，字符串的内容都为“Hello”；然后，使用“==”比较这两个字符串对象是否相等并输出比较结果；最后，使用 new 的方式构建第三个字符串对象，字符串的内容也为“Hello”，接着，使用“==”比较第一个字符串对象和第三个字符串对象是否相等并输出比较结果，根据输出结果验证 String 常量池的特点。

- 方案

首先，构建工程，包和类。

其次，在工程中添加 JUnit 的支持。

第三，在类中新建测试方法。

第四，首先，使用直接量“字符序列”的方式创建两个字符串对象，字符串的内容为“Hello”，然后，使用“==”比较两个字符串对象是否相等并输出比较结果。两个对象使用“==”进行比较，如果返回值为 true，说明两个对象的引用所指向的内存区域相同，即，指向了同一个对象。在此，可以说明，第一个字符串和第二个字符串是同一字符串且来自 String 常量池。因此，对于重复出现的字符串直接量，JVM 会首先在缓存池中查找，如果存在即返回该对象。

第五，使用 new 的方式构建第三个字符串对象，字符串的内容为“Hello”，使用“==”比较第一个字符串对象和第三个字符串对象是否相等并输出比较结果。两个对象使用“==”进行比较，如果返回值为 false，说明两个对象的引用所指向的内存区域不同，同时也说明，使用 new 方式构建的第三个字符串不会存储在 String 常量池中。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建工程，包和类

首先，新建名为 JavaSE 的工程，然后，在工程的 src 下新建名为 day01 的包，在包下新建名为 TestString 的类，工程结构图如图-5 所示。TestString 代码如下所示：

```
package day01;

public class TestString {
}
```

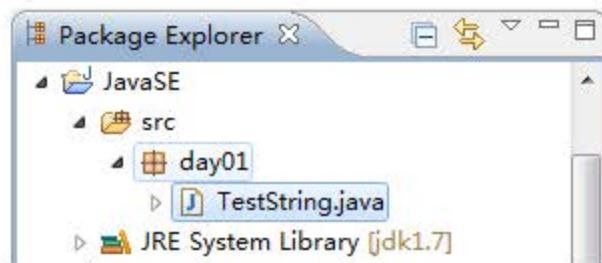


图 - 5

## 步骤二：添加 JUnit 的支持

给 JavaSE 工程添加 JUnit 支持。首先，选中工程，右键-->Build Path-->Add Libraries...，如图-6 所示。

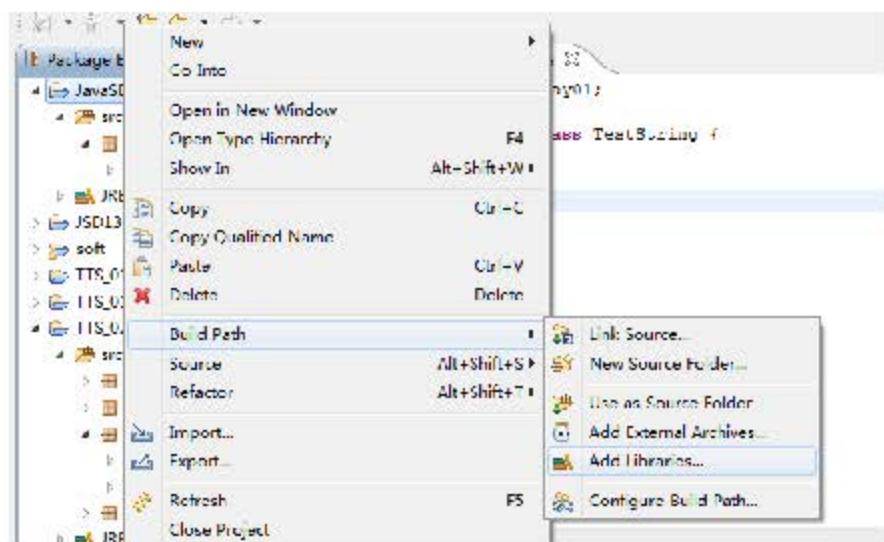


图 - 6

点击 “Add Libraries...” 后进入界面如图-7 所示。

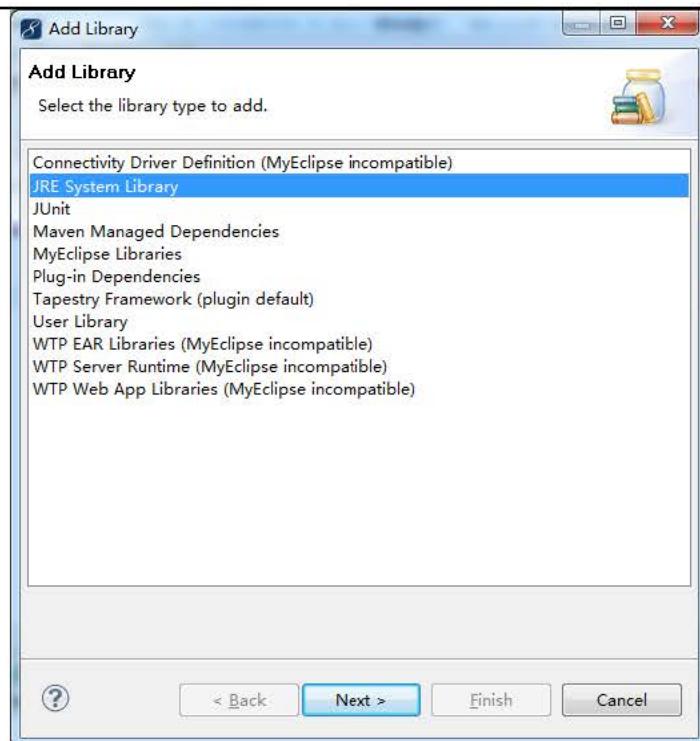


图- 7

接着，选择 JUnit，如图-8 所示，点击 Next，进入如图-9 所示界面。

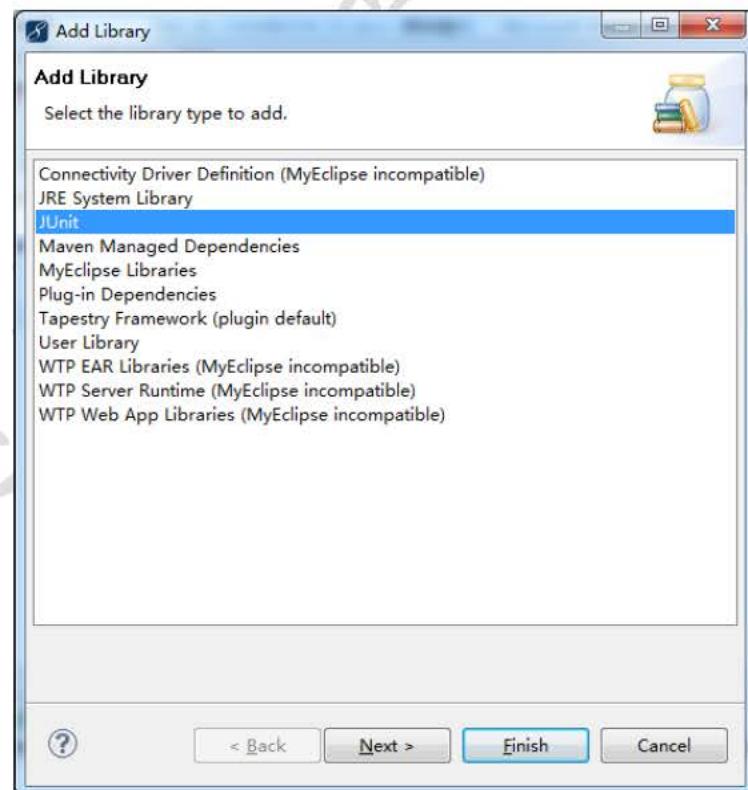


图- 8

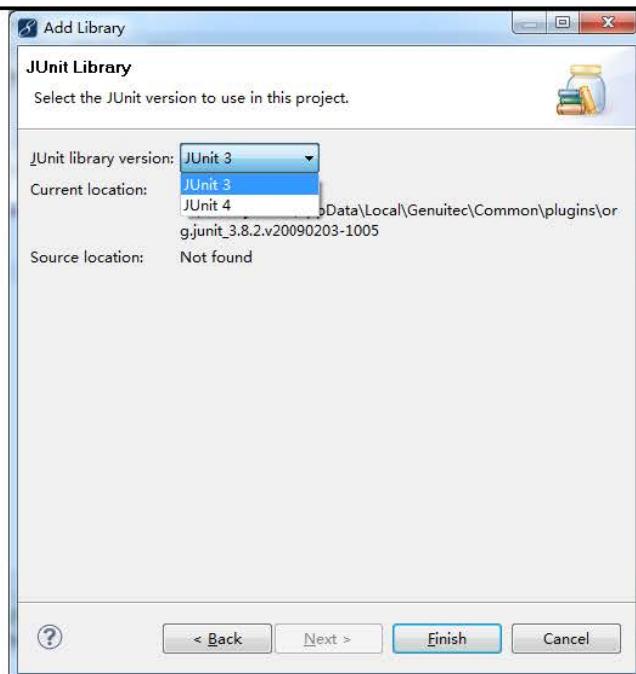


图- 9

最后，选择 JUnit4，点击 “Finish” 后，工程结构图如图-10 所示。

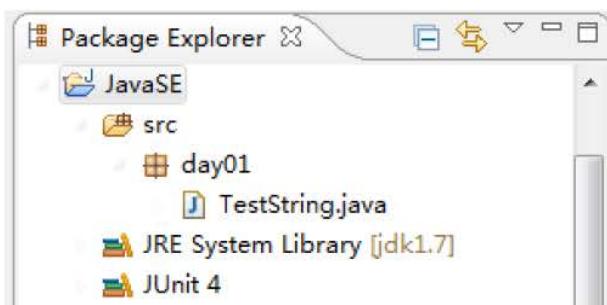


图- 10

从图-10 中可以看到 JUnit4 的支持已经添加成功。

### 步骤三：新建测试方法

在类 TestString 中，新建方法 testConstantPool 并在该方法前使用注解 @Test，表明该方法为 JUnit4 要进行测试的单元测试方法。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 测试 String 常量池
     */

    @Test
}
```

```
public void testConstantPool() {
}

}
```

注意导入包：org.junit.Test。

#### 步骤四：比较直接量“字符序列”的方式创建两个字符串对象

首先，使用直接量“字符序列”的方式创建两个字符串对象 str1 和 str2，两个字符串的内容都为“Hello”，然后，使用“==”比较 str1 和 str2 是否相等并输出比较结果。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 测试 String 常量池
     */
    @Test
    public void testConstantPool() {

        String str1 = "Hello";
        // 不会创建新的 String 对象，而是使用常量池中已有的"Hello".
        String str2 = "Hello";
        System.out.println(str1 == str2);

    }
}
```

#### 步骤五：运行

首先，点开 TestString 类的目录结构，如图-11 所示。

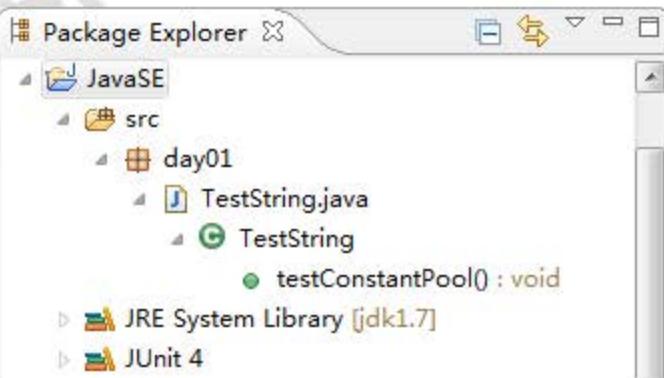


图- 11

在图-11 中，可以看到我们刚刚加入的方法 testConstantPool 方法，选中该方法，右键-->Run As -->JUnit Test，就可以运行该方法，如图-12 所示。

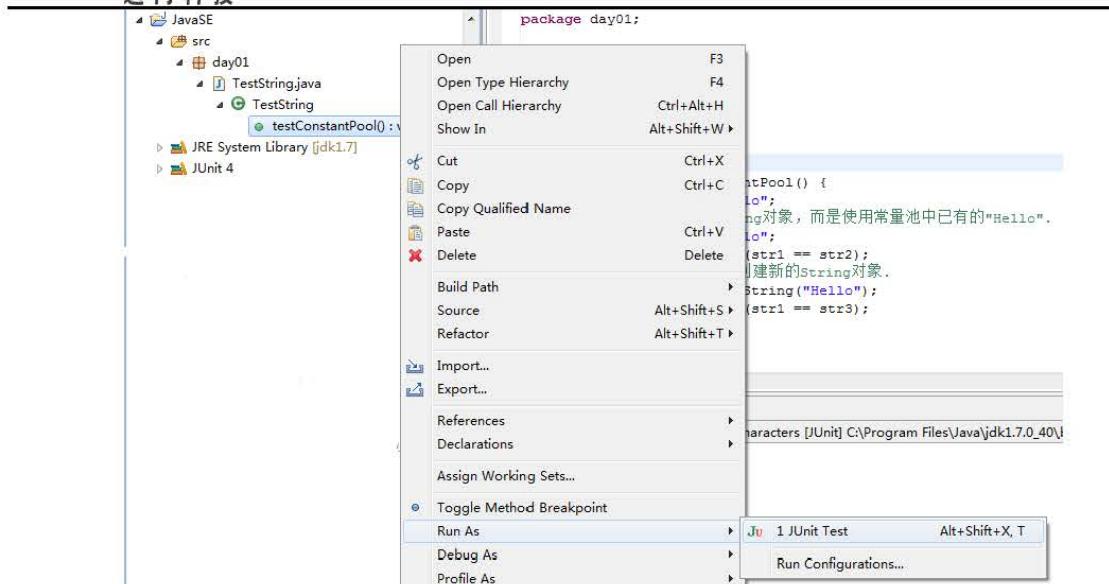


图 - 12

上述运行方式，只运行当前选中方法。也可以选中方法所在的类，右键-->Run As -->JUnit Test 运行，但是，如果该类中有很多测试（@Test 注解）方法，那么，所有方法都将运行。

运行 testConstantPool 方法后，会打开 JUnit 视图，如图-13 所示，在该视图中，可以看到绿色的进度条，说明方法运行正确，否则，进度条为红色。

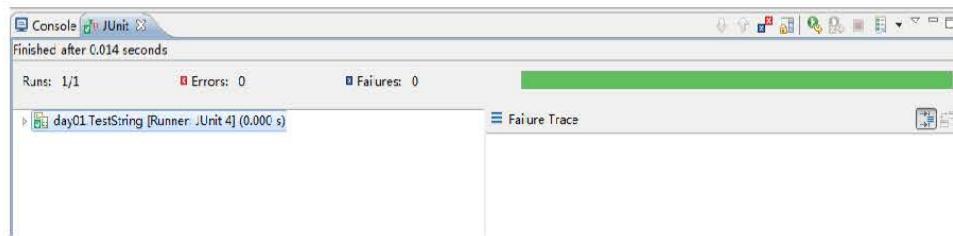


图 - 13

点击 Console 打开控制台视图，可以看到控制台输出结果为：

```
true
```

从输出结果可以看出，str1 和 str2 使用 “==” 比较的结果为 true。两个对象使用 “==” 进行比较，如果返回值为 true，说明两个对象的引用所指向的内存区域相同，即，指向了同一个对象。在此，可以说明，str1 和 str2 指向同一个对象，该对象来自 String 常量池，即，字符串直接量对象缓存在常量池中；对于重复出现的字符串直接量，JVM 会首先在缓存池中查找，如果存在即返回该对象。

#### 步骤六：使用 new 的方式构建 str3，比较 str1 和 str3 是否相等

使用 new 的方式构建字符串 str3，字符串的内容也为 “Hello”，使用 “==” 比较 str1 和 str3 是否相等并输出比较结果。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 测试 String 常量池
     */
    @Test
    public void testConstantPool() {
        String str1 = "Hello";
        // 不会创建新的 String 对象，而是使用常量池中已有的"Hello".
        String str2 = "Hello";
        System.out.println(str1 == str2);

        // 使用 new 关键字会创建新的 String 对象.
        String str3 = new String("Hello");
        System.out.println(str1 == str3);

    }
}
```

## 步骤七：运行

再次运行 testConstantPool 方法，控制台输出结果如下：

```
true  
  
false
```

可以看出 str1 和 str3 使用 “==” 比较的结果为 false。两个对象使用 “==” 进行比较，如果返回值为 false，说明两个对象的引用所指向的内存区域不同，在此说明，str3 不是来自常量池，即，使用 new 方式创建的字符串对象，不会缓存在 String 常量池中。

### • 完整代码

本案例的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 测试 String 常量池
     */
    @Test
    public void testConstantPool() {
        String str1 = "Hello";
        // 不会创建新的 String 对象，而是使用常量池中已有的"Hello".
        String str2 = "Hello";
        System.out.println(str1 == str2);

        // 使用 new 关键字会创建新的 String 对象.
        String str3 = new String("Hello");
        System.out.println(str1 == str3);

    }
}
```

### 3. 获取 String 对象的长度

- 问题

在上一案例的基础上，使用 String 类的 length 方法获取字符串 “Hello” 以及字符串 “你好，String” 的长度，并总结 length 方法的特点。

- 方案

首先，在类 TestString 中添加单元测试方法 testLength，然后，定义字符串对象 str1 和 str2，分别初始化为 “Hello” 和 “你好，String”，最后，调用 String 类的 length 方法分别计算对象 str1 和 str2 的长度。length 方法的声明如下：

```
int length()
```

以上方法返回字符串字符序列的长度。

- 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：创建 testLength 方法

在类 TestString 中添加单元测试方法 testLength。代码如下所示：

```
package day01;  
  
import org.junit.Test;  
  
public class TestString {  
  
    /**  
     * 获取 String 对象的长度  
     */  
    @Test  
    public void testLength() {  
    }  
  
}
```

#### 步骤二：使用 length 方法，获取字符串对象的长度

首先，定义字符串对象 str1 和 str2，分别初始化为 “Hello” 和 “你好，String”，最后，调用 String 类的 length 方法分别计算对象 str1 和 str2 的长度，代码如下所示：

```
package day01;  
  
import org.junit.Test;  
  
public class TestString {  
    /**
```

```
* 获取 String 对象的长度
*/
@Test
public void testLength() {

    String str1 = "Hello";
    System.out.println(str1.length());

    // 在内存中采用 Unicode 编码，每个字符占用两个字节。
    // 任何一个字符(无论中文还是英文)都算 1 个字符长度。
    String str2 = "你好，String";
    System.out.println(str2.length());

}
}
```

### 步骤三：测试

运行 testLength 方法，控制台输出结果如下所示：

5  
9

观察输出结果，5 为 str1 字符串对象的长度，9 为 str2 字符串对象的长度。对象 str1 的内容为 “Hello”，由 5 个字母组成，其计算出来长度为 5，即，一个英文字母是 1 个长度。接着，查看对象 str2 的长度情况，对象 str2 的内容为 “你好，String”，去除 “你好”，str2 剩余的长度为 7，如果一个中文按照 2 个长度计算，那么对象 str2 的长度应为 11，但是，输出结果却为 9，说明一个中文没有按照 2 个长度计算，而是按照 1 个长度。

另外，在 Java 中，字符在内存里采用的是 Unicode 编码，每个字符占用两个字节，请注意区别。

#### • 完整代码

本案例中，类 TestString 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    // ... (之前案例的代码，略)

    /**
     * 获取 String 对象的长度
     */
    @Test
    public void testLength() {
        String str1 = "Hello";
        System.out.println(str1.length());

        // 在内存中采用 Unicode 编码，每个字符占用两个字节。
        // 任何一个字符(无论中文还是英文)都算 1 个字符长度。
        String str2 = "你好，String";
    }
}
```

```
System.out.println(str2.length());  
}  
}
```

## 4. 在一个字符串中检索另外一个字符串

- 问题

在上一案例的基础上，检索一个字符串在另一个字符串中的索引位置。即，现有字符串“*I can because i think i can*”，检索字符串“can”在此字符串中出现的索引位置。详细要求如下：

- 1 )，检索字符串“can”在此字符串中第一次出现的索引位置。
- 2 )，检索字符串“can”在此字符串中最后一次出现的索引位置。
- 3 )，从索引位置为 6 开始，检索字符串“can”在此字符串第一次出现的索引位置。
- 4 )，检索字符串“Can”在此字符串中第一次出现的索引位置。注意字符串“Can”中的字母 C 为大写。

- 方案

1 ) 定义字符串对象 str，初始化为“*I can because i think i can*”，接着，调用 indexOf 方法，检索字符串“can”在此字符串中第一次出现的索引位置。该方法声明如下：

```
int indexOf(String str)
```

以上 indexOf 方法表示在字符串中检索 str，返回其第一出现的索引位置，如果找不到则返回-1。

2 ) 调用 lastIndexOf 的方法，检索字符串“can”在此字符串中最后一次出现的索引位置。该方法声明如下：

```
int lastIndexOf (String str)
```

lastIndexOf 方法和 indexOf 方法类似，只是当 str 在字符串中多次出现时，将返回最后一个出现的索引位置，如果找不到则返回-1。

3 ) 调用重载的 indexOf 方法，从索引位置为 6 开始，检索字符串“can”在此字符串中第一次出现的索引位置。该方法声明如下：

```
int indexOf(String str, int fromIndex)
```

以上方法类似 indexOf(String)，但是是从字符串的 fromIndex 位置开始检索。

4 ) 调用 indexOf ( String str ) 检索字符串“Can”在此字符串中第一次出现的索引位置，如果找不到则返回-1。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：使用 indexOf ( String str )

首先，在类 TestString 中，添加测试方法 testIndexOf。在该方法中，首先定义字符串对象 str，初始化为 “I can because i think i can”，接着，调用 indexOf 方法，检索字符串 “can” 在字符串 str 中第一次出现的索引位置。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 在一个字符串中检索另外一个字符串
     */
    @Test
    public void testIndexOf() {

        String str = "I can because i think i can";
        int index = str.indexOf("can");
        System.out.println(index); // 2

    }
}
```

运行 testIndexOf 方法，控制台输出结果如下：

2

从输出结果可以看出，字符串 “can” 在字符串中第一次出现的索引位置为 2。

### 步骤二：使用 lastIndexOf 方法

调用 lastIndexOf 的方法，检索字符串 “can” 在字符串 str 中最后一次出现的索引位置。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 在一个字符串中检索另外一个字符串
     */
    @Test
    public void testIndexOf() {
        String str = "I can because i think i can";
        int index = str.indexOf("can");
        System.out.println(index); // 2

        index = str.lastIndexOf("can");
        System.out.println(index); // 24
    }
}
```

```
}
```

运行 `testIndexOf` 方法，控制台输出结果如下：

```
2
```

```
24
```

从输出结果可以看出，字符串 “can” 在 str 中最后一次出现的索引位置为 24。

### 步骤三：使用 `indexOf(String str, int fromIndex)`

调用重载的 `indexOf` 方法，从索引位置为 6 开始，检索字符串 “can” 在字符串 str 中第一次出现的索引位置。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 在一个字符串中检索另外一个字符串
     */
    @Test
    public void testIndexOf() {
        String str = "I can because i think i can";
        int index = str.indexOf("can");
        System.out.println(index); // 2

        index = str.lastIndexOf("can");
        System.out.println(index); // 24

        index = str.indexOf("can", 6);
        System.out.println(index); // 24
    }
}
```

运行 `testIndexOf` 方法，控制台输出结果如下：

```
2
24
```

```
24
```

从输出结果可以看出，从索引位置 6 开始，字符串 “can” 在 str 中第一次出现的索引位置为 24。

### 步骤四：使用 `indexOf(String str)`，要检索的字符串不存在的情况

调用 `indexOf (String str)` 检索字符串 “Can” 在此字符串中第一次出现的索引位置，如果找不到则返回-1。代码如下所示：

```
package day01;
```

```
import org.junit.Test;

public class TestString {
    /**
     * 在一个字符串中检索另外一个字符串
     */
    @Test
    public void testIndexOf() {
        String str = "I can because i think i can";
        int index = str.indexOf("can");
        System.out.println(index); // 2

        index = str.lastIndexOf("can");
        System.out.println(index); // 24

        index = str.indexOf("can", 6);
        System.out.println(index); // 24

        index = str.indexOf("Can");
        System.out.println(index); // -1
    }
}
```

运行 `testIndexOf` 方法，控制台输出结果如下：

```
2
24
24
```

```
-1
```

从输出结果可以看出，控制台的输出结果为-1，说明字符串 “Can” 在 str 中不存在。

#### • 完整代码

本案例中，类 `TestString` 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    // ... (之前案例的代码, 略)

    /**
     * 在一个字符串中检索另外一个字符串
     */
    @Test
    public void testIndexOf() {
        String str = "I can because i think i can";
        int index = str.indexOf("can");
        System.out.println(index); // 2

        index = str.lastIndexOf("can");
        System.out.println(index); // 24

        index = str.indexOf("can", 6);
        System.out.println(index); // 24

        index = str.indexOf("Can");
    }
}
```

```
        System.out.println(index); // -1
    }
}
```

## 5. 在一个字符串中截取指定字符串

### • 问题

在一个字符串中截取指定的字符串，即，现有字符串“`http://www.oracle.com`”，截取此字符串中的部分字符串形成新字符串。详细要求如下：

- 1 ) 截取此字符串中的字符串 “oracle”。
- 2 ) 从索引位置 7 ( 包括 ) 开始，截取到此字符串的结尾。

### • 方案

1 ) 首先，定义字符串对象 `str`，初始化为 “`http://www.oracle.com`”。接着，使用重载的 `substring` 方法，截取字符串 `str` 中的字符串 “oracle”，该 `substring` 方法的声明如下：

```
String substring(int beginIndex, int endIndex)
```

以上 `substring` 方法，返回字符串中从下标 `beginIndex`( 包括 )开始到 `endIndex`( 不包括 )结束的子字符串。

2 ) 用重载的 `substring` 方法，从索引位置 7( 包括 )开始，截取到此字符串的结尾，该 `substring` 方法声明如下：

```
String substring(int beginIndex)
```

以上 `substring` 方法，返回字符串中从下标 `beginIndex` ( 包括 ) 开始到字符串结尾的子字符串。

### • 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：使用 `substring(int beginIndex, int endIndex)` 方法

首先，在 `TestString` 类中添加测试方法 `testSubstring`；然后，在该方法中，定义字符串对象 `str`，初始化为“`http://www.oracle.com`”；接着，使用重载的 `substring(int beginIndex, int endIndex)` 方法，截取字符串 `str` 中的字符串 “oracle”，代码如下所示：

```
package day01;
import org.junit.Test;
```

```
public class TestString {  
    /**  
     * 在一个字符串中截取指定的字符串  
     */  
  
    @Test  
    public void testSubstring() {  
        String str = "http://www.oracle.com";  
        String subStr = str.substring(11, 17);  
        System.out.println(subStr); // oracle  
    }  
  
}
```

运行 `testSubstring` 方法，控制台输出结果如下：

```
oracle
```

从输出结果可以看出，取到了子字符串 “oracle”。

观察输出结果，并对比代码，不难发现，在使用 `substring(int beginIndex, int endIndex)` 方法时，其参数索引位置的特点为“前包括后不包括”，这样设计的目的是使得后一个参数减去前一个参数的值正好是截取子字符串的长度。

### 步骤二：`substring(int beginIndex)`方法

用重载的 `substring(int beginIndex)` 方法，从索引位置 7（包括）开始，截取到此字符串的结尾，代码如下所示：

```
package day01;  
  
import org.junit.Test;  
  
public class TestString {  
    /**  
     * 在一个字符串中截取指定的字符串  
     */  
    @Test  
    public void testSubstring() {  
        String str = "http://www.oracle.com";  
        String subStr = str.substring(11, 17);  
        System.out.println(subStr); // oracle  
  
        subStr = str.substring(7);  
        System.out.println(subStr); // www.oracle.com  
    }  
}
```

运行 `testSubstring` 方法，运行结果如下所示：

```
oracle
```

```
www.oracle.com
```

从输出结果可以看出，从索引位置 7 开始后的字符串为 “www.oracle.com”。

- **完整代码**

本案例的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    //... (之前案例的代码, 略)

    /**
     * 在一个字符串中截取指定的字符串
     */
    @Test
    public void testSubstring() {
        String str = "http://www.oracle.com";
        String subStr = str.substring(11, 17);
        System.out.println(subStr); // oracle

        subStr = str.substring(7);
        System.out.println(subStr); // www.oracle.com
    }
}
```

## 6. 去掉一个字符串的前导和后继空字符

- **问题**

在上一案例的基础上，去掉一个字符串的前导和后继空白，即，现有字符串“ good man ”，该字符串的 good 前面有两个空格，man 后面有两个空格，本案例要求去掉该字符串前后的空格，得到去除空格后的字符串，在此过程中对比去除空格前后字符串的长度。

- **方案**

首先，定义字符串对象 `userName`，并初始化为“ good man ”，然后，使用 `length` 方法获取字符串 `userName` 的长度并输出；接着，调用 `trim` 方法，去除字符串 `userName` 的前导和后继空白；最后，再次使用 `length` 方法获取字符串 `userName` 的长度并输出长度和该字符串。`trim` 方法的声明如下：

```
String trim()
```

以上 `trim` 方法返回字符串为原始字符串去掉前导和后继的空白。

- **步骤**

实现此案例需要按照如下步骤进行。

**步骤一：使用 `length` 方法**

首先，在 `TestString` 类中添加测试方法 `testTrim`，然后，定义字符串对象 `userName`，并初始化为“ good man ”，最后，使用 `length` 方法获取字符串 `userName` 的长度并输出。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 去掉一个字符串的前导和后继空字符
     */
    @Test
    public void testTrim() {

        String userName = " good man ";
        System.out.println(userName.length()); // 12

    }
}
```

运行 `testTrim` 方法，控制台输出结果如下：

12

从输出结果可以看出，此时字符串 `userName` 的长度为 12。

## 步骤二：使用 `trim` 方法

首先，调用 `trim` 方法，去除字符串 `userName` 的前导和后继空白；最后，再次使用 `length` 方法获取字符串 `userName` 的长度并输出该长度和该字符串，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 去掉一个字符串的前导和后继空字符
     */
    @Test
    public void testTrim() {
        String userName = " good man ";
        System.out.println(userName.length()); // 12

        userName = userName.trim();
        System.out.println(userName.length()); // 8
        System.out.println(userName); // good man

    }
}
```

运行 `testTrim` 方法，控制台输出结果如下：

12  
8  
good man

查看输出结果，可以看出去掉前导和后继空白后的字符串长度为 8。

- **完整代码**

本案例中，类 `TestString` 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    //... (之前案例的代码, 略)

    /**
     * 去掉一个字符串的前导和后继空字符
     */
    @Test
    public void testTrim() {
        String userName = " good man ";
        System.out.println(userName.length()); // 12
        userName = userName.trim();
        System.out.println(userName.length()); // 8
        System.out.println(userName); // good man
    }
}
```

## 7. 遍历一个字符串中的字符序列

- **问题**

在上一案例的基础上，遍历一个字符串中的字符序列，即，现有字符串“Whatisjava?”，遍历该字符串中每一个字符并输出。

- **方案**

首先，定义字符串对象 `name`，并初始化为“Whatisjava?”；然后，使用固定次数的循环，循环的条件为小于字符串 `name` 的长度，即，`name.length()`；最后，在循环中使用 `charAt` 方法，获取字符串 `name` 各个位置的字符，即，`char c = name.charAt(i);`，并输出字符 `c`。其中，`charAt` 方法的声明如下所示：

```
char charAt(int index)
```

以上 `charAt` 方法用于返回字符串指定位置的字符。参数 `index` 表示指定的位置。

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：创建测试方法

在 TestString 类中添加测试方法 testCharAt，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 遍历一个字符串中的字符序列
     */

    @Test
    public void testCharAt() {
    }

}
```

### 步骤二：遍历字符串中的字符序列

首先，定义字符串对象 name，并初始化为 "What is java?"；然后，使用固定次数的循环，循环的条件为小于字符串 name 的长度，即，name.length()；最后，在循环中使用 charAt 方法，获取字符串 name 各个位置的字符，即，char c = name.charAt(i)；并输出字符 c。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 遍历一个字符串中的字符序列
     */
    @Test
    public void testCharAt() {

        String name = "What is java?";
        for (int i = 0; i < name.length(); i++) {
            char c = name.charAt(i);
            System.out.print(c + " ");
        }
        // W h a t i s j a v a ?
    }
}
```

### 步骤三：运行

运行 testCharAt 方法，控制台输出结果如下所示：

```
W h a t i s j a v a ?
```

从输出结果，再结合代码，可以看出已经遍历到字符串 name 中的每一个字符。

- 完整代码

本案例中，类 `TestString` 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    // ... (之前案例的代码, 略)

    /**
     * 遍历一个字符串中的字符序列
     */
    @Test
    public void testCharAt() {
        String name = "Whatisjava?";
        for (int i = 0; i < name.length(); i++) {
            char c = name.charAt(i);
            System.out.print(c + " ");
        }
        // W h a t i s j a v a ?
    }
}
```

## 8. 检测一个字符串是否以指定字符串开头或结尾

- 问题

在上一案例的基础上，检测一个字符串是否以指定字符串开头或结尾，即，现有字符串“Thinking in Java”，检索该字符串是否以字符串“Java”作为结尾；是否以字符串“T”作为开头；是否以“thinking”作为开头。

- 方案

首先，定义字符串对象 `str`，并初始化为“Thinking in Java”；

然后，调用 `endsWith` 方法，判断字符串 `str` 是否以字符串“Java”作为结尾。`endsWith` 方法的声明如下所示：

```
boolean endsWith(String suffix)
```

以上 `endsWith` 方法用于判断字符串是否以参数字符串结尾，如果是则返回 `true`，否则返回 `false`。

第三，调用 `startsWith` 方法，判断字符串 `str` 是以已字符串“T”或者“thinking”作为开头。`startsWith` 方法的声明如下所示：

```
boolean startsWith(String prefix)
```

以上 `startsWith` 方法用于判断字符串是否以参数字符串开头，如果是则返回 `true`，否则则返回 `false`。

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：添加测试方法

首先，在 `TestString` 类中添加测试方法 `testStartsWithAndEndsWith`，然后，在该方法中，定义字符串对象 `str`，并初始化为“Thinking in Java”，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {

    /**
     * 检测一个字符串是否以指定字符串开头或结尾
     */
    @Test
    public void testStartsWithAndEndsWith() {
        String str = "Thinking in Java";
    }

}
```

### 步骤二：使用 `endsWith` 方法

调用 `endsWith` 方法，判断字符串 `str` 是否以字符串 “Java” 作为结尾。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 检测一个字符串是否以指定字符串开头或结尾
     */
    @Test
    public void testStartsWithAndEndsWith() {
        String str = "Thinking in Java";

        System.out.println(str.endsWith("Java")); // true
    }
}
```

运行 `testStartsWithAndEndsWith` 方法，控制台的输出结果为：

```
true
```

从输出结果可以看出，字符串 `str` 确实以 “Java” 作为结尾。

### 步骤三：使用 `startsWith` 方法

调用 `startsWith` 方法，判断字符串 str 是否以字符串 “T” 或者字符串 “thinking” 作为开头，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 检测一个字符串是否以指定字符串开头或结尾
     */
    @Test
    public void testStartsWithAndEndsWith() {
        String str = "Thinking in Java";
        System.out.println(str.endsWith("Java")); // true

        System.out.println(str.startsWith("T")); // true
        System.out.println(str.startsWith("thinking")); // false

    }
}
```

运行 `testStartsWithAndEndsWith` 方法，控制台输出结果为：

```
true

true
false
```

从输出结果可以看出，字符串 str 以 “T” 作为开头，没有以 “thinking” 作为开头。

- **完整代码**

本案例中，类 `TestString` 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    //... (之前案例的代码, 略)

    /**
     * 检测一个字符串是否以指定字符串开头或结尾
     */
    @Test
    public void testStartsWithAndEndsWith() {
        String str = "Thinking in Java";
        System.out.println(str.endsWith("Java")); // true
        System.out.println(str.startsWith("T")); // true
        System.out.println(str.startsWith("thinking")); // false
    }
}
```

## 9. 转换字符串中英文字母的大小写形式

- 问题

在上一案例的基础上，将字符串中的英文字符都转换成小写或者都转换成大写形式，即，现有字符串“我喜欢 Java”，将该字符串中的英文 Java 都转成小写形式或者转成大写形式。

- 方案

首先，使用 String 类的 `toUpperCase` 方法将字符串中的所有英文字符转换成大写，该方法声明如下所示：

```
String toUpperCase()
```

以上方法用于返回字符串的大写形式。

其次，使用 String 类的 `toLowerCase` 方法将字符串中的所有英文字符转换成小写，该方法声明如下所示：

```
String toLowerCase()
```

以上方法用于返回字符串的小写形式。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：使用 `toUpperCase` 方法

首先，在类 `TestString` 中，添加测试方法，然后，在该方法中，首先，定义字符串 `str`，然后初始化为“我喜欢 Java”，接着，使用 String 类的 `toUpperCase` 方法将字符串中的所有英文字符转换成大写形式。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {

    /**
     * 转换字符串中英文字母的大小写形式
     */
    @Test
    public void testToUpperAndToLower() {
        String str = "我喜欢 Java";

        str = str.toUpperCase();
        System.out.println(str); // 我喜欢 JAVA
    }
}
```

运行 `testToUpperCaseAndtoLowerCase` 方法，控制台输出结果如下：

我喜欢 JAVA

从输出结果可以看出，已经将小写 “Java”，转成了大写形式的 “JAVA”。

### 步骤二：使用 `toLowerCase` 方法

使用 `String` 类的 `toLowerCase` 方法将字符串中的所有英文字符转换成小写。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 转换字符串中英文字母的大小写形式
     */
    @Test
    public void testToUpperCaseAndtoLowerCase() {
        String str = "我喜欢 Java";

        str = str.toUpperCase();
        System.out.println(str); // 我喜欢 JAVA

        str = str.toLowerCase();
        System.out.println(str); // 我喜欢 java
    }
}
```

运行 `testToUpperCaseAndtoLowerCase` 方法，控制台输出结果如下所示：

我喜欢 JAVA

我喜欢 java

从输出结果可以看出，又将大写的 “JAVA” 转成了小写 “java”的形式。

#### • 完整代码

本案例中，类 `TestString` 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
```

```
//... (之前案例的代码, 略)

/**
 * 转换字符串中英文字母的大小写形式
 */
@Test
public void testToUpperCaseAndLowerCase() {
    String str = "我喜欢 Java";

    str = str.toUpperCase();
    System.out.println(str); // 我喜欢 JAVA

    str = str.toLowerCase();
    System.out.println(str); // 我喜欢 java
}
```

## 10. 将其他类型转换为字符串类型

- 问题

在上一案例的基础上, 将其他类型转换为字符串类型, 即, 将 double 类型, int 类型, boolean 类型以及 char 数组类型的变量转换为 String 类型变量。

- 方案

使用 String 类的 valueOf 重载的方法, 可以将 double 类型, int 类型, boolean 类型以及 char 数组类型等变量转换为 String 类变量。在图-14 展示了 Java API 提供的 valueOf 重载的方法。

static String	<a href="#">valueOf</a> (boolean b)	返回 boolean 参数的字符串表示形式。
static String	<a href="#">valueOf</a> (char c)	返回 char 参数的字符串表示形式。
static String	<a href="#">valueOf</a> (char[] data)	返回 char 数组参数的字符串表示形式。
static String	<a href="#">valueOf</a> (char[] data, int offset, int count)	返回 char 数组参数的特定子数组的字符串表示形式。
static String	<a href="#">valueOf</a> (double d)	返回 double 参数的字符串表示形式。
static String	<a href="#">valueOf</a> (float f)	返回 float 参数的字符串表示形式。
static String	<a href="#">valueOf</a> (int i)	返回 int 参数的字符串表示形式。
static String	<a href="#">valueOf</a> (long l)	返回 long 参数的字符串表示形式。
static String	<a href="#">valueOf</a> (Object obj)	返回 Object 参数的字符串表示形式。

图- 14

- 步骤

### 步骤一：构建测试方法

首先，在 `TestString` 类中，添加 `testValueOf` 方法，然后，分别定义 `double` 类型，`int` 类型，`boolean` 类型以及 `char` 数组类型的变量，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 将其他类型转换为字符串类型
     */

    @Test
    public void testValueOf() {
        double pi = 3.1415926;
        int value = 123;
        boolean flag = true;
        char[] charArr = { 'a', 'b', 'c', 'd', 'e', 'f', 'g' };
    }

}
```

### 步骤二：使用 `valueOf` 方法

使用 `valueOf` 重载的方法，将 `double` 类型，`int` 类型，`boolean` 类型以及 `char` 数组类型等变量转换为 `String` 类变量，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    /**
     * 将其他类型转换为字符串类型
     */
    @Test
    public void testValueOf() {
        double pi = 3.1415926;
        int value = 123;
        boolean flag = true;
        char[] charArr = { 'a', 'b', 'c', 'd', 'e', 'f', 'g' };

        String str = String.valueOf(pi);

        System.out.println(str);

        str = String.valueOf(value);
        System.out.println(str);

        str = String.valueOf(flag);
        System.out.println(str);

        str = String.valueOf(charArr);
        System.out.println(str);
    }
}
```

```
}
```

### 步骤三：运行

运行 `testValueOf` 方法，控制台输出结果如下所示：

```
3.1415926
123
true
abcdefg
```

从输出结果可以看出，已经将 `double` 类型数据、`int` 类型数据、`boolean` 类型数据以及字符数组类型数据转成类 `String` 类型数据。

#### • 完整代码

本案例的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestString {
    //... (之前案例的代码, 略)

    /**
     * 将其他类型转换为字符串类型
     */
    @Test
    public void testValueOf() {
        double pi = 3.1415926;
        int value = 123;
        boolean flag = true;
        char[] charArr = { 'a', 'b', 'c', 'd', 'e', 'f', 'g' };

        String str = String.valueOf(pi);
        System.out.println(str);

        str = String.valueOf(value);
        System.out.println(str);

        str = String.valueOf(flag);
        System.out.println(str);

        str = String.valueOf(charArr);
        System.out.println(str);
    }
}
```

## 11. 测试 `StringBuilder` 的 `append` 方法

#### • 问题

`StringBuilder` 类提供将各种数据类型变量的字符串形式追加到当前序列中的 `append` 方法，在 Java API 中提供的 `append` 重载方法如图-15 所示：

<code>StringBuilder</code>	<code>append(boolean b)</code> 将 boolean 参数的字符串表示形式追加到序列。
<code>StringBuilder</code>	<code>append(char c)</code> 将 char 参数的字符串表示形式追加到此序列。
<code>StringBuilder</code>	<code>append(char[] str)</code> 将 char 数组参数的字符串表示形式追加到此序列。
<code>StringBuilder</code>	<code>append(char[] str, int offset, int len)</code> 将 char 数组参数的子数组的字符串表示形式追加到此序列。
<code>StringBuilder</code>	<code>append(CharSequence s)</code> 向此 Appendable 追加到指定的字符序列。
<code>StringBuilder</code>	<code>append(CharSequence s, int start, int end)</code> 将指定 CharSequence 的子序列追加到此序列。
<code>StringBuilder</code>	<code>append(double d)</code> 将 double 参数的字符串表示形式追加到此序列。
<code>StringBuilder</code>	<code>append(float f)</code> 将 float 参数的字符串表示形式追加到此序列。
<code>StringBuilder</code>	<code>append(int i)</code> 将 int 参数的字符串表示形式追加到此序列。
<code>StringBuilder</code>	<code>append(long lng)</code> 将 long 参数的字符串表示形式追加到此序列。
<code>StringBuilder</code>	<code>append(Object obj)</code> 追加 Object 参数的字符串表示形式。
<code>StringBuilder</code>	<code>append(String str)</code> 将指定的字符串追加到此字符序列。

图- 15

本案例要求将字符串 “java”、“cpp”、“php”、“c#” 以及 “objective-c” 追加到字符串序列“Programming Language:”的后面。

### • 方案

首先，实例化 `StringBuilder` 类的对象，并且初始化该对象内容为“Programming Language:”字符串；然后调用 `append` 方法，再为该对象追加字符串 “java”、“cpp”、“php”、“c#” 以及 “objective-c”；最后，调用 `StringBuilder` 类的 `toString` 方法，将该对象转换为 `String` 类型变量并输出。

另外，`StringBuilder` 的很多方法的返回值均为 `StringBuilder` 类型。这些方法的返回语句均为：`return this`。可见，这些方法在对 `StringBuilder` 所封装的字符序列进行改变后又返回了该对象的引用。基于这样的设计的目的在于可以按照如下简洁的方式书写代码：

```
sb.append("java").append("cpp").append("php").append("c#")
    .append("objective-c");
```

### • 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：创建类 `TestStringBuilder`，并在其中添加测试方法

在工程 JavaSE 的 day01 包下，新建名为 `TestStringBuilder` 类，并在该类中添加测试方法 `testAppend`，工程结构如图-16 所示，代码如下所示：

```

package day01;

import org.junit.Test;

public class TestStringBuilder {

    /**
     * 测试 StringBuilder 的 append 方法
     */
    @Test
    public void testAppend() {
    }
}

```

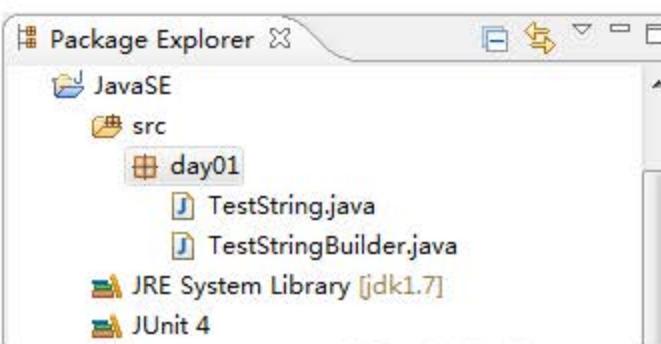


图- 16

### 步骤二：使用 append 方法

首先，实例化 `StringBuilder` 类的对象，并且初始化该对象内容为“Programming Language:”字符串；然后调用 `append` 方法，再为该对象追加字符串 “java”、“cpp”、“php”、“c#” 以及 “objective-c”。代码如下所示：

```

package day01;

import org.junit.Test;

public class TestStringBuilder {

    /**
     * 测试 StringBuilder 的 append 方法
     */
    @Test
    public void testAppend() {

        StringBuilder sb = new StringBuilder("Programming Language:");
        sb.append("java").append("cpp").append("php").append("c#")
            .append("objective-c");

    }
}

```

### 步骤三：使用 `toString` 方法

调用 `StringBuilder` 类的 `toString` 方法，将该对象转换为 `String` 类型变量并输出，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {

    /**
     * 测试 StringBuilder 的 append 方法
     */
    @Test
    public void testAppend() {
        StringBuilder sb = new StringBuilder("Programming Language:");
        sb.append("java").append("cpp").append("php").append("c#")
            .append("objective-c");

        System.out.println(sb.toString());
    }
}
```

#### 步骤四：运行

运行 TestStringBuilder 类中的 testAppend 方法，控制台输出结果如下所示：

```
Programming Language:javacpphpcc#objective-c
```

从运行结果可以看出，将字符串 “java”、“cpp”、“php”、“c#” 以及 “objective-c” 追加在 “Programming Language:” 的后边。

- **完整代码**

本案例中，类 TestStringBuilder 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {

    /**
     * 测试 StringBuilder 的 append 方法
     */
    @Test
    public void testAppend() {
        StringBuilder sb = new StringBuilder("Programming Language:");
        sb.append("java").append("cpp").append("php").append("c#")
            .append("objective-c");
        System.out.println(sb.toString());
    }
}
```

## 12. 测试 StringBuilder 的 insert 方法

- **问题**

StringBuilder 类提供将各种数据类型变量的字符串形式插入到当前序列中的

`insert` 方法，在 Java API 中提供的 `insert` 重载方法如图-17 所示：

<code>StringBuilder</code>	<code>insert(int offset, boolean b)</code> 将 boolean 参数的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int offset, char c)</code> 将 char 参数的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int offset, char[] str)</code> 将 char 数组参数的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int index, char[] str, int offset, int len)</code> 将数组参数 str 子数组的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int dstOffset, CharSequence s)</code> 将指定 CharSequence 插入此序列中。
<code>StringBuilder</code>	<code>insert(int dstOffset, CharSequence s, int start, int end)</code> 将指定 CharSequence 的子序列插入此序列中。
<code>StringBuilder</code>	<code>insert(int offset, double d)</code> 将 double 参数的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int offset, float f)</code> 将 float 参数的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int offset, int i)</code> 将 int 参数的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int offset, long l)</code> 将 long 参数的字符串表示形式插入此序列中。
<code>StringBuilder</code>	<code>insert(int offset, Object obj)</code> 将 Object 参数的字符串表示形式插入此字符序列中。
<code>StringBuilder</code>	<code>insert(int offset, String str)</code> 将字符串插入此字符序列中。

图- 17

本案例要求在字符序列"javacppc#objective-c"中的 "#" 后插入字符串 "php"。

### • 方案

使用 `StringBuilder` 类的 `insert` 方法将字符串 "php" 插入到字符序列 "javacppc#objective-c" 的 "#" 后面，即，插入到索引位置为 9 的位置，原本该索引位置及其后面的字符向后顺延。

### • 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：添加测试方法

在 `TestStringBuilder` 类中添加测试方法 `testInsert`。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {

    /**
     * 测试 StringBuilder 的 insert 方法
     */
    @Test
    public void testInsert() {
```

}

}

## 步骤二：使用 insert 方法

使用 `StringBuilder` 类的 `insert` 方法将字符串 “php” 插入到字符序列 “javacppc#objective-c”的 “#” 后面，即，插入到索引位置为 9 的位置，原本该索引位置及其后面的字符向后顺延。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {
    /**
     * 测试 StringBuilder 的 insert 方法
     */
    @Test
    public void testInsert() {

        StringBuilder sb = new StringBuilder("javacppc#objective-c");
        sb.insert(9, "php");
        System.out.println(sb);

    }
}
```

以上代码输出字符序列信息时，没有调用 `toString` 方法，但输出 `sb` 和 `sb.toString` 效果是一样的。

## 步骤三：测试

运行 `testInsert` 方法，控制台输出结果如下：

```
javacppc#phoprojective-c
```

观察以上输出结果可以看到字符串 “php” 已经插入到 “#” 后面。

### • 完整代码

本案例中，类 `TestStringBuilder` 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {
    //... (之前案例的代码，略)

    /**
     * 测试 StringBuilder 的 insert 方法
     */
    @Test
    public void testInsert() {
```

```
        StringBuilder sb = new StringBuilder("javacppc#objective-c");
        sb.insert(9, "php");
        System.out.println(sb);
    }
```

## 13. 测试 StringBuilder 的 delete 方法

- 问题

测试 StringBuilder 的 delete 方法，即，现有字符序列"javaoraclecppc#php"，删除该字符序列中的 "oracle"。

- 方案

使用 StringBuilder 的 delete 方法，可以实现将字符序列"javaoraclecppc#php"中的 "oracle" 删除，代码如下所示：

```
sb.delete(4, 4 + 6);
```

以上代码表示从索引 4 到索引 10 之间的字符，删除时，包含索引位置 4 的字符，但是不包含索引位置为 10 的字符，即，“前包括后不包括”。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：添加测试方法

在 TestStringBuilder 类中添加测试方法 testDelete。代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {

    /**
     * 测试 StringBuilder 的 delete 方法
     */
    @Test
    public void testDelete() {
    }

}
```

### 步骤二：使用 delete 方法

使用 StringBuilder 的 delete 方法，可以实现将字符序列"javaoraclecppc#php"中的 "oracle" 删除，代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {
    /**
     * 测试 StringBuilder 的 delete 方法
     */
    @Test
    public void testDelete() {

        StringBuilder sb = new StringBuilder("javaoraclecpc#php");
        sb.delete(4, 4 + 6);
        System.out.println(sb);

    }
}
```

### 步骤三：测试

运行 `testDelete` 方法，控制台输出结果如下：

```
javacppc#php
```

观察以上输出结果可以看到字符串 “oracle” 被删除。

#### • 完整代码

本案例中，类 `TestStringBuilder` 的完整代码如下所示：

```
package day01;

import org.junit.Test;

public class TestStringBuilder {
    //... (之前案例的代码，略)

    /**
     * 测试 StringBuilder 的 delete 方法
     */
    @Test
    public void testDelete() {
        StringBuilder sb = new StringBuilder("javaoraclecpc#php");
        sb.delete(4, 4 + 6);
        System.out.println(sb);
    }
}
```

## 课后作业

### 1. 下列说法正确的是：

- A . 在 Java 语言中，使用 new 关键字创建的字符串缓存在常量池中。
- B . 在 Java 语言中，可以使用直接量“字符序列”的方式创建字符串。
- C . 在 Java 语言中，对于使用 new 关键字创建的字符串序列，如果重复出现，JVM 会首先在常量池中查找，如果存在即返回该对象。
- D . 下列代码的输出结果为 false

```
String str1 = "WhatisJava";
String str2 = "WhatisJava";
System.out.println(str1 == str2);
```

### 2. 下面关于字符串长度说法正确的是：

- A . 使用 Java 中的 String 类的 length 方法计算字符串的长度，如果是英文算 1 个长度，如果是中文算 2 个长度。
- B . 使用 Java 中的 String 类的 length 方法计算字符串的长度，由于 Java 采用 Unicode 编码任何一个字符在内存中算 2 个长度，因此，length 方法中也是一个字符 2 个长度，不区分中文还是英文。
- C . 使用 Java 中的 String 类的 length 方法计算字符串的长度，无论中文还是英文都算 1 个长度。
- D . 使用 Java 中的 String 类的 length 方法计算字符串"你好 String"的长度为 10。

### 3. 获取一个字符串中最后一个"/"后的字符序列

获取一个字符串中最后一个"/"后的字符序列，即，现有字符串 "someapp/manager/emplist.action"，截取该字符串最后一个"/"后的所有字符，截取后的结果为字符串 "emplist.action"。

在课上案例“将其他类型转换为字符串类型”的基础上完成当前案例。

### 4. 说出 trim 方法一般用在何处

### 5. 检测一个字符串是否为回文

回文字符串是指正着读和反着读该字符串都是相同拼写，比如 "radar"、"level"。

本案例要求编写程序判断字符串 "able was i ere i saw elba" 是否为回文。

在课后案例“获取一个字符串中最后一个"/"后的字符序列”的基础上完成当前案例。

---

## 6. 生成一个包含所有汉字的字符串

生成一个包含所有汉字的字符串，即，编写程序输出所有汉字，每生成 50 个汉字进行换行输出。

在课上案例“测试 `StringBuilder` 的 `delete` 方法”的基础上完成当前案例。

## 7. 阅读 Apache Commons-lang.jar 中 StringUtils 文档，掌握 `leftPad`、 `rightPad`、`repeat`、`abbreviate`、`join` 等方法(提高题，选做)

# Java 核心 API(上)

## Unit02

知识体系.....Page 53

正则表达式	基本正则表达式	正则表达式简介
		分组 “0”
		“^” 和 “\$”
	String 正则 API	matches 方法
		split 方法
		replaceAll 方法
Object	Object	Object
	toString 方法	如何重写 toString 方法
		String 重写 toString 方法
	equals 方法	equals 方法
		如何重写 equals 方法
		String 重写 equals 方法
		equals 和==的区别
包装类	包装类概述	包装类概述
	8 种基本类型包装类	Number 及其主要方法
		Integer 常用功能
		Double 常用功能
		自动装箱和拆箱操作

经典案例.....Page 64

编写验证 Email 的正则表达式	正则表达式简介
	分组 “0”
	“^” 和 “\$”
	matches 方法
使用 split 方法拆分字符串	split 方法
使用 replaceAll 实现字符串替换	replaceAll 方法
重写 Cell 类的 toString 方法	如何重写 toString 方法
	String 重写 toString 方法
重写 Cell 类的 equals 方法	equals 方法

	如何重写 equals 方法
测试 Number 的 intValue 和 doubleValue 方法	Number 及其主要方法
测试 Integer 的 parseInt 方法	Integer 常用功能
测试 Double 的 parseDouble 方法	Double 常用功能

**课后作业.....Page 86**

## 1. 正则表达式

### 1.1. 基本正则表达式

#### 1.1.1. 【基本正则表达式】正则表达式简介

**正则表达式简介**

**Tarena**  
达内科技

知识讲解

- 实际开发中，经常需要对字符串数据进行一些复杂的匹配、查找、替换等操作。通过“正则表达式”，可以方便的实现字符串的复杂操作。
- 正则表达式是一串特定字符，组成一个“规则字符串”，这个“规则字符串”是描述文本规则的工具。正则表达式就是记录文本规则的代码。
- 例如：
  - 正则表达式：“[a-z]” 表示a到z的任意一个字符
  - 正则表达式：“[a-z]+” 表示由1个或多个a-z字符组成的字符串。

+

**正则表达式简介（续1）**

**Tarena**  
达内科技

知识讲解

正则表达式	说明
[abc]	a、b、c中任意一个字符
[^abc]	除了a、b、c的任意字符
[a-z]	a、b、c、.....、z中的任意一个字符
[a-zA-Z0-9]	a~z、A~Z、0~9中任意一个字符
[a-zA-Z0-9]&&[^bc]	a~z中除了b和c以外的任意一个字符，其中&&表示“与”的关系

+

**正则表达式简介（续2）**

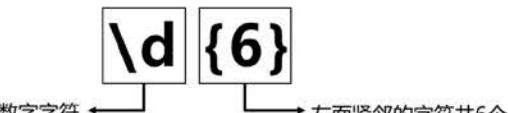
**Tarena**  
达内科技

知识讲解

正则表达式	说明
.	任意一个字符
\d	任意一个数字字符，相当于[0-9]
\w	单词字符,相当于 [ a-zA-Z0-9_ ]
\s	空白字符，相当于[ \t\n\x0B\f\r ]
\D	非数字字符
\W	非单词字符
\S	非空白字符

+

知识讲解	正则表达式简介 ( 续3 )														
	<b>Tarena</b> 达内科技														
	<b>数量词</b>														
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr><th style="width: 30%;">正则表达式</th><th style="width: 70%;">说明</th></tr> </thead> <tbody> <tr><td>X?</td><td>表示0个或1个X</td></tr> <tr><td>X*</td><td>表示0个或任意多个X</td></tr> <tr><td>X<sup>+</sup></td><td>表示1个到任意多个X ( 大于等于1个X )</td></tr> <tr><td>X{n}</td><td>表示n个X</td></tr> <tr><td>X{ n , }</td><td>表示n个到任意多个X ( 大于等于n个X )</td></tr> <tr><td>X{ n , m }</td><td>表示n个到m个X</td></tr> </tbody> </table>	正则表达式	说明	X?	表示0个或1个X	X*	表示0个或任意多个X	X <sup>+</sup>	表示1个到任意多个X ( 大于等于1个X )	X{n}	表示n个X	X{ n , }	表示n个到任意多个X ( 大于等于n个X )	X{ n , m }	表示n个到m个X
正则表达式	说明														
X?	表示0个或1个X														
X*	表示0个或任意多个X														
X <sup>+</sup>	表示1个到任意多个X ( 大于等于1个X )														
X{n}	表示n个X														
X{ n , }	表示n个到任意多个X ( 大于等于n个X )														
X{ n , m }	表示n个到m个X														
+															

知识讲解	正则表达式简介 ( 续4 )
	<b>Tarena</b> 达内科技
	<ul style="list-style-type: none"> <li>• 检索邮政编码 :</li> <li>– 规则为6位数字</li> <li>– 第一种匹配规则 [0-9][0-9][0-9][0-9][0-9][0-9]</li> <li>– 简化第一种规则 \d\d\d\d\d\d</li> <li>– 简化第二种规则 \d{ 6 }</li> </ul>
+	

### 1.1.2. 【基本正则表达式】分组 “()”

知识讲解	分组 “()”
	<b>Tarena</b> 达内科技
	<ul style="list-style-type: none"> <li>• 分组 : () 圆括号表示分组，可以将一系列正则表达式看做一个整体，分组时可以使用 “ ” 表示“或”关系，例如：匹配手机号码前面的区号：</li> </ul>
+	$(\+86 0086)?\$?\d{11}$ 上述例子中，圆括号表示这里需要出现“ +86” 或者 “0086”

知识讲解

### 分组 “()” (续1)

• 检索手机号码：+86 13838389438

- +86 可有可无
- +86与后面的号码之间空格可以没有或者有多个
- 电话号码为11位数字

字符转义，匹配 + 号 前面的字符数量  $\geq 0$

+  
+

#### 1.1.3. 【基本正则表达式】“^”和“\$”

知识讲解

### “^”和“\$”

- 边界匹配
  - ^ 代表字符串开始
  - \$ 代表字符串结束
- 例如：匹配用户名规则 – 从头到尾连续8~10个单词字符
 

```
\w{8,10}  
^\w{8,10}$
```

  - 如果使用第一种写法，则 “abcd1234\_abcd” 是可以验证通过的；
  - 使用第二种写法由于有从头到尾整体的限定，则验证不能通过。

+  
+

### 1.2. String 正则 API

#### 1.2.1. 【String 正则 API】matches 方法

知识讲解

### matches方法

- matches ( 正则表达式 ) 方法：将一个字符串与正则表达式进行匹配
- 如果匹配成功就返回true，否则返回false

```
/** 测试email是否合法 */
public void email() {
    String emailRegEx = "^[a-zA-Z0-9_-]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,4}$";
    String email = "bjliyi@tarena.com.cn";
    System.out.println(email.matches(emailRegEx));
}
```

+

## 1.2.2. 【String 正则 API】split 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<h3>split方法</h3> <p>String[ ] split( String regex ) 参数regex为正则表达式 以regex所表示的字符串为分隔符，将字符串拆分成字符串数组。</p> <p>数据：“3, tarena, 3, tarena@tarena.com, 33” 拆分为：[ “3”, “ tarena ”, “ 3 ”, “ tarena@tarena.com ”, “ 33 ” ] 拆分方式：str . split ( “\s*” ); 解释：以逗号开头，&gt;=0个的空格组合为分割</p>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<h3>split方法（续1）</h3> <pre>/** 使用split方法拆分字符串 */ public void testSplit() {     // 按空格拆分     String str = "java cpp php c# objective-c";     String[] strArr = str.split("\s");     System.out.println(Arrays.toString(strArr));      // 按+、-、=符号拆分     String line = "100+200-150=150";     strArr = line.split("[\+\-\=]");     System.out.println(Arrays.toString(strArr)); }</pre>
---	--

## 1.2.3. 【String 正则 API】replaceAll 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<h3>replaceAll方法</h3> <p>String replaceAll (String regex, String replacement)</p> <p>将字符串中匹配正则表达式 regex的字符串替换成 replacement。</p>
---	---

### replaceAll方法 ( 续1 )

```
/** 使用replaceAll方法实现字符串替换 */
@Test
public void testReplaceAll() {
    // 将str中的所有数字替换为“数字”二字
    String str = "abc123bcd45ef6g7890";
    str = str.replaceAll("\d+", "数字");
    System.out.println(str);
}
```

知识讲解



+

## 2. Object

### 2.1. Object

#### 2.1.1. 【Object】Object

### Object



- 在Java类继承结构中，java.lang.Object类位于顶端；
- 如果定义一个Java类时没有使用extends关键字声明其父类，则其父类默认为java.lang.Object类；
- Object类型的引用变量可以指向任何类型对象。

知识讲解

+

### 2.2. toString方法

#### 2.2.1. 【toString方法】如何重写toString方法

### 如何重写toString方法



- Object类中的重要方法，用于返回对象值的字符串表示；
- 原则上建议重写，格式大多数遵循“类的名字[域值]”，例如：

```
public String toString() {
    return getClass().getName()
        +" [name= "+name
        +",salary= "+salary
        +",gender= "+gender
        +"]";
}
```

知识讲解

+

## 如何重写toString方法（续1）

- Java语言中很多地方会默认调用对象的toString方法：
  - 字符串+对象，自动调用对象的toString方法
  - System.out.print(任意对象)，直接调用toString方法
- 如果不重写toString方法，将使用Object的toString方法，其逻辑为：
  - 类名@散列码
- toString方法是非常有用的调试工具；
- JDK中的标准类库中，许多类都定义了toString方法，方便用户获得有关对象状态的必要信息；
- 强烈建议为自定义的每一个类增加toString方法。

知识讲解



### 2.2.2. 【toString方法】String 重写 toString 方法

## String重写toString方法

- String的toString()方法就是将自身返回了。

```
public String toString(){  
    return this;  
}
```

知识讲解



### 2.3. equals 方法

#### 2.3.1. 【equals 方法】equals 方法

## equals方法

- Object中的方法，作用在于检测一个对象是否等于另外一个对象；
- 在Object类中，这个方法判断两个对象是否具有相同的引用，即是否为相同的对象；
- 在实际应用中，一般需要重写该方法，通过比较对象的成员属性，使该方法更有意义，例如：对于Cell类，如果不重写equals方法，下面代码在cell1和cell2指向同一个对象时才为true，可以将其重写为：当x和y坐标相等时两个对象即相等，这样更有意义一些。  
`cell1.equals(cell2);`

知识讲解



### 2.3.2. 【equals 方法】如何重写 equals 方法

#### 如何重写equals方法

```
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (this == obj) {
        return true;
    }
    if (obj instanceof Cell) {
        Cell cell = (Cell) obj;
        return cell.row == row && cell.col == col;
    } else {
        return false;
    }
}
```

知识讲解



### 2.3.3. 【equals 方法】String 重写 equals 方法

#### String重写equals方法

- String的equals方法用于比较两个字符串对象的字符序列是否相等。

```
/** 测试字符串的比较*/
public class Demo {
    public static void main(String[] args){
        String s1 = new String("abc");
        String s2 = new String("abc");
        String s3 = new String("A");

        System.out.println(s1.equals(s2)); //true
        System.out.println(s1.equals(s3)); //false
    }
}
```

知识讲解



### 2.3.4. 【equals 方法】equals 和==的区别

#### equals和==的区别

- == 用于比较变量的值，可以应用于任何类型，如果用于引用类型，比较的是两个引用变量中存储的值（地址信息），判断两个变量是否指向相同的对象；
- equals是Object的方法，重写以后，可以用于比较两个对象的内容是否“相等”；
- 需要注意的是，Object默认的equals方法的比较规则同 ==

知识讲解



### 3. 包装类

#### 3.1. 包装类概述

##### 3.1.1. 【包装类概述】 包装类概述

**包装类概述**

• 在进行类型转换的范畴内，有一种特殊的转换，需要将int这样的基本数据类型转换为对象；  
 • 所有基本类型都有一个与之对应的类，即包装类（wrapper）。



知识讲解
+

**包装类概述（续1）**

• 包装类是不可变类，在构造了包装类对象后，不允许更改包装在其中的值；  
 • 包装类是final的，不能定义他们的子类。

基本类型	基本类型	父类
int	java.lang.Integer	java.lang.Number
long	java.lang.Long	java.lang.Number
double	java.lang.Double	java.lang.Number
short	java.lang.Short	java.lang.Number
float	java.lang.Float	java.lang.Number
byte	java.lang.Byte	java.lang.Number
char	java.lang.Character	java.lang.Object
boolean	java.lang.Boolean	java.lang.Object

知识讲解
+

#### 3.2. 8 种基本类型包装类

##### 3.2.1. 【8 种基本类型包装类】 Number 及其主要方法

**Number及其主要方法**

- 抽象类 Number 是 Byte、Double、Float、Integer、Long 和 Short 类的父类；
- Number的子类必须提供将表示的数值转换为byte , double , float , int , long和short的方法：
  - doubleValue ( ) 以double形式返回指定的数值
  - intValue ( ) 以int形式返回指定的数值
  - floatValue ( ) 以float形式返回指定的数值

知识讲解
+

60

## Number及其主要方法 (续1)

```
/**测试Number的intValue方法和doubleValue方法*/
public void testIntValueAndDoubleValue() {
    Number d = 123.45;    Number n = 123;
    // 输出d和n对象所属的类型
    System.out.println(d.getClass().getName()); // java.lang.Double
    System.out.println(n.getClass().getName()); // java.lang.Integer

    int intValue = d.intValue();
    double doubleValue = d.doubleValue();
    System.out.println(intValue + "," + doubleValue); // 123,123.45

    intValue = n.intValue();
    doubleValue = n.doubleValue();
    System.out.println(intValue + "," + doubleValue); // 123,123.0
}
```

知识讲解



### 3.2.2. 【8种基本类型包装类】Integer 常用功能

#### Integer常用功能

- 该类提供了多个方法，能在int类型和String类型之间互相转换，还提供一些常量例如：
  - static int MAX\_VALUE 值为  $2^{31} - 1$  的常量，表示int类型能表示的最大值
  - static int MIN\_VALUE 值为  $-2^{31}$  的常量，表示int类型能表示的最小值

知识讲解



#### Integer常用功能 (续1)

- Integer的静态方法parseInt用于将字符串转换为int。

```
/** String 转换为 int */
public void testParseInt() {
    String str = "123";
    int value = Integer.parseInt(str);
    System.out.println(value); // 123
    str = "壹佰贰拾叁";
    // 会抛出NumberFormatException
    // value = Integer.parseInt(str);
}
```

知识讲解



### 3.2.3. 【8种基本类型包装类】Double 常用功能

**Double常用功能**

**Tarena**  
达内科技

- 在Double类的对象中包装一个基本类型double的值
- 构造方法
  - Double ( double value )
  - Double ( String s )
- 方法
  - double doubleValue ( ) 返回此Double对象的double值
  - static double parseDouble ( String s ) 返回一个新的double值，该值被初始化为用指定的String表示的值

+

**Double常用功能 ( 续1 )**

**Tarena**  
达内科技

```
/** String 转换为 double */
public void testParseDouble() {
    String str = "12345.00";
    double value = Double.parseDouble(str);
    // 12345.0
    System.out.println(value);

    str = "¥12345.00";

    // 会抛出NumberFormatException
    // value = Double.parseDouble(str);
}
```

+

### 3.2.4. 【8种基本类型包装类】自动装箱和拆箱操作

**自动装箱和拆箱操作**

**Tarena**  
达内科技

- 从Java 5.0版本以后加入了 autoboxing功能
- 自动“拆箱”和“装箱”是依靠JDK5的编译器在编译期的“预处理”工作
- 下列代码写法均为正确形式：

```
Integer a = 100;      // 装箱
Integer b = 200;      // 装箱
Integer c = a + b;    // 拆箱再装箱
double d = c;         // 拆箱
```

+

知识讲解

自动装箱和拆箱操作（续1）

• 装箱和拆箱是“编译器”认可的，而不是虚拟机。编译器在生成类的字节码时插入必要的方法调用

- Integer a = 100 => Integer a = Integer.valueOf(100)
- Integer b = 200 => Integer b = Integer.valueOf(200)
- Integer c = a + b =>

```
Integer c =  
    Integer.valueOf(a.intValue() + b.intValue())  
- int d = c => int d = c.intValue()
```

+

知识讲解

自动装箱和拆箱操作（续2）

• 方法的参数

```
void takeNumber ( Integer i ) { }
```

• 返回值

```
int giveNumer ( ) {  
    return x;  
}
```

3 Integer 3 int  
3 Integer 3 int

+

## 经典案例

### 1. 编写验证 Email 的正则表达式

- 问题

写出与 email 匹配的正则表达式，并测试该正则表达式的正确性。

- 方案

首先，分析 email 的正则表达式，email 的正则表达式如下：

```
[a-zA-Z0-9\.-]+@[a-zA-Z0-9-]+\.\{2,4}
```

以上正则表达式可以分成三部分，各个部分如下：

1 ) [a-zA-Z0-9\.-]+@

表示 a~z、A~Z、0~9、下划线、点以及减号可以出现至少 1 次也可以多次，然后出现@符号。

2 ) ([a-zA-Z0-9-]+\.\)+

这段中的 “( )” 表示分组，这段正则表示这一组至少出现一次，也可以出现多次。括号中的内容表示，a~z、A~Z、0~9 及减号可以至少出现 1 次也可以出现多次，然后出现 “.” 字符。

3 ) [a-zA-Z0-9]{2,4}

表示 a~z、A~Z、0~9 可以出现大于等于 2 次，小于等于 4 次。

其次，使用 String 类的 matches 方法，可以测试一个字符串是否和一个正则表达式匹配，matches 方法的声明如下：

```
public boolean matches(String regex)
```

以上方法表示当字符串和正则表达式 regex 匹配时返回 true，否则，返回 false。

- 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：添加测试方法

首先，在名为 JavaSE 的工程下的 src 下新建名为 day02 的包，然后，在该包下新建名为 TestRegEx 的类，然后在该类中添加测试方法 email，代码如下所示：

```
package day02;  
import org.junit.Test;
```

```
public class TestRegEx {  
  
    /**  
     * 测试 email 是否合法  
     */  
    @Test  
    public void email() {  
  
    }  
  
}
```

## 步骤二：测试 email 正则表达式的正确性

首先，定义字符变量 emailRegEx，该变量赋值为正则表达式：

```
"^ [a-zA-Z0-9_\\.\\-] + @ ([a-zA-Z0-9-] + \\.) + [a-zA-Z0-9] {2,4} $"
```

其中，^表示匹配输入的开始位置，\$表示匹配输入的结束位置。此处需要注意的是，在正则表达式中“\.” 表示的是“.” 字符，但是，在 Java 代码中需要对“\”字符做转意，即“\\.” 表示一个“\.”。因此在定义 Java 的正则表达式变量 emailRegEx 时，使用“\\.” 来表示一个“.” 字符。

然后，测试 email ( bjliyi@tarena.com.cn ) 是否和正则表达式 emailRegEx 匹配，代码如下所示：

```
package day02;  
  
import org.junit.Test;  
  
public class TestRegEx {  
    /**  
     * 测试 email 是否合法  
     */  
    @Test  
    public void email() {  
  
        String emailRegEx =  
            "^ [a-zA-Z0-9_\\.\\-] + @ ([a-zA-Z0-9-] + \\.) + [a-zA-Z0-9] {2,4} $";  
  
        String email = "bjliyi@tarena.com.cn";  
  
        System.out.println(email.matches(emailRegEx));  
  
    }  
}
```

## 步骤三：运行

运行方法 email，控制台输出结果如下：

```
true
```

从输出结果可以看出 email ( bjliyi@tarena.com.cn ) 和正则表达式 emailRegEx 是匹配的。

- 完整代码

```
package day02;

import org.junit.Test;

public class TestRegEx {
    /**
     *
     * 测试 email 是否合法
     */
    @Test
    public void email() {
        String emailRegEx =
            "^[a-zA-Z0-9_\\.-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9]{2,4}$";
        String email = "bjliyi@tarena.com.cn";
        System.out.println(email.matches(emailRegEx));
    }
}
```

## 2. 使用 split 方法拆分字符串

- 问题

使用 split 方法进行字符串的拆分，具体要求如下所示：

- 1) 使用空格对字符串"java cpp php c# objective-c"进行拆分。
- 2) 使用+、-以及=对字符串"100+200-150=150"进行拆分。

- 方案

在 Java API 中的 String 类提供了 split 方法，实现使用正则表达式对字符串进行拆分的方法，split 方法的声明如下所示：

```
String[] split(String regex)
```

以上方法表示以 regex 所表示的字符串为分隔符，将字符串拆分成字符串数组，其中，参数 regex 为正则表达式。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：添加测试方法

在 TestRegEx 中添加测试方法 testSplit，代码如下所示：

```
package day02;

import org.junit.Test;

import java.util.Arrays;

public class TestRegEx {

    /**
     * 使用 split 方法拆分字符串
     *
     */
    @Test
    public void testSplit() {
    }

}
```

## 步骤二：使用 split 方法

首先，使用空格对字符串"java cpp php c# objective-c" 进行分割，在正则表达式中，使用 "\s" 来表示一个空白字符。

然后，使用+、-以及=对字符串"100+200-150=150"进行拆分，在正则表达式中，使用 "[\+\-\=]" 来表示+、-以及=。代码如下所示：

```
package day02;

import org.junit.Test;
import java.util.Arrays;

public class TestRegEx {
    /**
     * 使用 split 方法拆分字符串
     *
     */
    @Test
    public void testSplit() {

        // 按空格拆分
        String str = "java cpp php c# objective-c";
        String[] strArr = str.split("\s");
        System.out.println(Arrays.toString(strArr));

        // 按+、-、=符号拆分
        String line = "100+200-150=150";
        strArr = line.split("[\+\-\=]");
        System.out.println(Arrays.toString(strArr));

    }
}
```

## 步骤三：运行

运行方法 testSplit，控制台输出结果如下：

```
[java, cpp, php, c#, objective-c]  
[100, 200, 150, 150]
```

从输出结果可以看出已经将字符串"java cpp php c# objective-c"和字符串"100+200-150=150"按照要求的方式拆分为数组中元素。

- **完整代码**

本案例中，类 TestRegEx 的完整代码如下所示：

```
package day02;  
  
import org.junit.Test;  
import java.util.Arrays;  
  
public class TestRegEx {  
    //... (之前案例的代码, 略)  
  
    /**  
     * 使用 split 方法拆分字符串  
     */  
    @Test  
    public void testSplit() {  
        // 按空格拆分  
        String str = "java cpp php c# objective-c";  
        String[] strArr = str.split("\\s");  
        System.out.println(Arrays.toString(strArr));  
  
        // 按+、-、=符号拆分  
        String line = "100+200-150=150";  
        strArr = line.split("[\\+\\-\\=]");  
        System.out.println(Arrays.toString(strArr));  
    }  
}
```

### 3. 使用 replaceAll 实现字符串替换

- **问题**

使用 replaceAll 实现字符串替换，具体要求为将字符串"abc123bcd45ef6g7890"中的数字替换为汉字“数字”，如果是连续的数字，那么替换为一个汉字“数字”。

- **方案**

在 Java API 中的 String 类提供了 replaceAll 方法，实现将字符串中匹配正则表达式的字符串替换成其它字符串，replaceAll 方法的声明如下所示：

```
String replaceAll(String regex, String replacement)
```

以上方法实现将字符串中匹配正则表达式 regex 的字符串替换成 replacement。

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：添加测试方法

在 TestRegEx 中添加测试方法 testReplaceAll，代码如下所示：

```
package day02;

import org.junit.Test;
import java.util.Arrays;

public class TestRegEx {

    /**
     * 使用 replaceAll 方法实现字符串替换
     *
     */
    @Test
    public void testReplaceAll() {

    }

}
```

### 步骤二：使用 replaceAll 方法

将字符串"abc123bcd45ef6g7890"中的数字替换为汉字“数字”，如果是连续的数字，那么替换为一个汉字“数字”，在正则表达式中，使用"\d+"来表示可以出现一个或多个数字，代码如下所示：

```
package day02;

import org.junit.Test;
import java.util.Arrays;

public class TestRegEx {
    /**
     * 使用 replaceAll 方法实现字符串替换
     *
     */
    @Test
    public void testReplaceAll() {

        // 将 str 中的所有数字替换为“数字”二字
        String str = "abc123bcd45ef6g7890";
        str = str.replaceAll("\d+", "数字");
        System.out.println(str);

    }
}
```

### 步骤三：运行

运行方法 testReplaceAll，控制台输出结果如下：

从输出结果可以看出已经将字符串"abc123bcd45ef6g7890"中的数字替换为汉字 "数字"。

- **完整代码**

本案例中，类 TestRegEx 的完整代码如下所示：

```
package day02;

import org.junit.Test;
import java.util.Arrays;

public class TestRegEx {
    //... (之前案例的代码, 略)

    /**
     * 使用 replaceAll 方法实现字符串替换
     */
    @Test
    public void testReplaceAll() {
        // 将 str 中的所有数字替换为“数字”二字
        String str = "abc123bcd45ef6g7890";
        str = str.replaceAll("\\d+", "数字");
        System.out.println(str);
    }
}
```

## 4. 重写 Cell 类的 `toString` 方法

- **问题**

Object 是 Java 的继承 root 类，Java 类继承了 Object 的所有方法 如: `toString()`, `hashCode()`, `equals()`，其中 `toString()`方法的特点如下：

- 1 ), `toString()` 方法 返回对象的文本描述。
- 2 ), 经常被系统默认调用， 默认返回：全限定名@HashCode， 默认调用是指输出对象时会默认调用 `toString` 方法。
- 3 ), 建议 Java 类覆盖 `toString()`， 返回合理的文本。

本案例要求在 Cell 类中覆盖 `toString` 方法，返回行列的值，例如：6 , 3。

- **方案**

首先，新建 Cell 类，然后在其中覆盖 `toString` 方法，返回 `row` 和 `col` 的值；

然后，测试 `toString` 方法是否覆盖生效。首先，在 JavaSE 工程下的 day02 包下新建 `TestCell`，然后，在该类中添加测试方法 `testToString`，最后，实例化一个 Cell 类的对象，其 `row`、`col` 为 6、3，并输出该对象。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：新建 Cell 类

新建 Cell 类，然后在其中覆盖 `toString` 方法，返回 `row` 和 `col` 的值，代码如下所示：

```
package day02;

public class Cell {
    int row;
    int col;

    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public Cell() {
        this(0, 0);
    }

    public Cell(Cell cell) {
        this(cell.row, cell.col);
    }

    public void drop() {
        row++;
    }

    public void moveRight() {
        col++;
    }

    public void moveLeft() {
        col--;
    }

    @Override
    public String toString() {
        return row + "," + col;
    }
}
```

### 步骤二：写测试方法

测试 `toString` 方法是否覆盖生效。首先，在 JavaSE 工程下的 `day02` 包下新建 `TestCell`，然后，在该类中添加测试方法 `testToString`，最后，实例化一个 `Cell` 类的对象，其 `row`、`col` 为 6、3，并输出该对象，代码如下所示：

```
package day02;

import org.junit.Test;

public class TestCell {
    /**
```

```
* 测试 toString 方法
*/
@Test
public void testToString() {
    Cell cell = new Cell(6, 3);
    System.out.println(cell); // 6,3
}
```

### 步骤三：运行

运行 `testToString` 方法，控制台输出结果如下所示：

```
6,3
```

从运行结果可以看出，已经成功覆盖了 `toString` 方法，返回实际 `row`、`col` 的值 6、3。

### • 完整代码

本案例中，类 `Cell` 的完整代码如下所示：

```
package day02;

public class Cell {
    int row;
    int col;

    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public Cell() {
        this(0, 0);
    }

    public Cell(Cell cell) {
        this(cell.row, cell.col);
    }

    public void drop() {
        row++;
    }

    public void moveRight() {
        col++;
    }

    public void moveLeft() {
        col--;
    }

    @Override
    public String toString() {
        return row + "," + col;
    }
}
```

`TestCell` 类完整代码如下：

```
package day02;

import org.junit.Test;

public class TestCell {
    /**
     * 测试 toString 方法
     */
    @Test
    public void testToString() {
        Cell cell = new Cell(6, 3);
        System.out.println(cell); // 6,3
    }
}
```

## 5. 重写 Cell 类的 equals 方法

- 问题

Java 程序中测试两个变量是否相等有两种方式：一种是利用`==`运算符，另一种是利用`equals`。

当使用`==`比较两个引用变量时，它们必须指向同一个对象，`==`判断才会返回`true`。

`equals`方法是`Object`类提供的一个实例方法，因此所有引用变量都可以调用该方法来判断是否与其它引用变量相等。但使用这个方法判断两个对象相等的标准与使用`==`运算符没有区别，同样要求两个引用变量指向同一个对象才会返回`true`。因此这个`Object`类提供的`equals`方法没有太大的实际意义，如果希望采用自定义的相等标准，则可以采用重写`equals`方法。

本案例要求在`Cell`类中重写`equals`方法，两个`Cell`引用变量相等的条件是行行相等，列列相等。

- 方案

首先，在上一案例的基础上，在`Cell`类中覆盖`equals`方法，两个`Cell`引用变量相等的条件是行行相等，列列相等。

然后，测试`equals`方法是否覆盖生效。首先，在类`TestCell`中添加测试方法`testEquals`，然后，创建`cell1`对象和`cell2`对象，代码如下：

```
Cell cell1 = new Cell(6, 3);
Cell cell2 = new Cell(6, 3);
```

最后分别使用`==`和`equals`比较`cell1`对象和`cell2`对象是否相等。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：在`Cell`类中覆盖`equals`方法

在上一案例的基础上，在`Cell`类中覆盖`equals`方法，两个`Cell`引用变量调用`equals`

方法返回 true 的条件是行行相等，列列相等，代码如下所示：

```
package day02;

public class Cell {
    int row;
    int col;

    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public Cell() {
        this(0, 0);
    }

    public Cell(Cell cell) {
        this(cell.row, cell.col);
    }

    public void drop() {
        row++;
    }

    public void moveRight() {
        col++;
    }

    public void moveLeft() {
        col--;
    }

    @Override
    public String toString() {
        return row + "," + col;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (this == obj) {
            return true;
        }
        if (obj instanceof Cell) {
            Cell cell = (Cell) obj;
            return cell.row == row && cell.col == col;
        } else {
            return false;
        }
    }
}
```

## 步骤二：测试 equals 方法是否生效

测试 equals 方法是否覆盖生效。首先，在类 TestCell 中添加测试方法 testEquals，然后，创建 cell1 对象和 cell2 对象；最后分别使用 == 和 equals 比较 cell1 对象和 cell2 对象是否相等，代码如下所示：

```
package day02;

import org.junit.Test;

public class TestCell {

    /**
     * 测试 equals 方法
     */
    @Test
    public void testEquals() {
        Cell cell1 = new Cell(6, 3);
        Cell cell2 = new Cell(6, 3);
        System.out.println(cell1 == cell2); // false
        System.out.println(cell1.equals(cell2)); // true
    }

}
```

### 步骤三：运行

运行 testEquals 方法，控制台输出结果如下所示：

```
false
true
```

从运行结果可以看出，使用`==`比较的结果为`false`、使用`equals`比较的结果为`true`，说明`Cell`类成功覆盖了`equals`方法，实现了行行相等并且列列相等两个`Cell`的引用变量则相等。

- **完整代码**

本案例中，类`Cell`的完整代码如下所示：

```
package day02;

public class Cell {
    int row;
    int col;

    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public Cell() {
        this(0, 0);
    }

    public Cell(Cell cell) {
        this(cell.row, cell.col);
    }

    public void drop() {
        row++;
    }

    public void moveRight() {
```

```

        col++;
    }

    public void moveLeft() {
        col--;
    }

    @Override
    public String toString() {
        return row + "," + col;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (this == obj) {
            return true;
        }
        if (obj instanceof Cell) {
            Cell cell = (Cell) obj;
            return cell.row == row && cell.col == col;
        } else {
            return false;
        }
    }
}

```

TestCell 类完整代码如下：

```

package day02;

import org.junit.Test;

public class TestCell {
    //... (之前案例的代码, 略)

    /**
     * 测试 equals 方法
     */
    @Test
    public void testEquals() {
        Cell cell1 = new Cell(6, 3);
        Cell cell2 = new Cell(6, 3);
        System.out.println(cell1 == cell2); // false
        System.out.println(cell1.equals(cell2)); // true
    }
}

```

## 6. 测试 Number 的 intValue 和 doubleValue 方法

- 问题

在 Java 中，抽象类 Number 是 BigDecimal、BigInteger、Byte、Double、Float、Integer、Long 和 Short 类的超类。该类提供了六个方法，如图-1 所示。

byte	<a href="#">byteValue()</a>	以 byte 形式返回指定的数值。
abstract double	<a href="#">doubleValue()</a>	以 double 形式返回指定的数值。
abstract float	<a href="#">floatValue()</a>	以 float 形式返回指定的数值。
abstract int	<a href="#">intValue()</a>	以 int 形式返回指定的数值。
abstract long	<a href="#">longValue()</a>	以 long 形式返回指定的数值。
short	<a href="#">shortValue()</a>	以 short 形式返回指定的数值。

图- 1

以上六个方法中 ,[floatValue](#) 方法和 [doubleValue](#) 方法在使用时 , 可能涉及到舍入 ; 其它四个方法可能涉及到舍入或者取整。

本案例要求测试 Number 类的 [intValue](#) 方法和 [doubleValue](#) 方法 , 即 , 首先 , 定义 Number 类型的两个变量 :

```
Number d = 123.45;
Number n = 123;
```

要求获取 d 对象和 n 对象所属的数据类型。

然后 , 将 d 对象分别转换为 int 类型变量和 double 类型变量 , 并输出转换后的结果。

最后 , 将 n 对象分别转换为 int 类型变量和 double 类型变量 , 并输出转换后的结果。

### • 方案

首先 , 构建包和类 , 并在类中新建测试方法。

其次 , 使用代码 `d.getClass().getName()` 和代码 `n.getClass().getName()` 获取 d 对象和 n 对象所属的数据类型。此处只需了解这样可以获取对象所属数据类型即可。

第三 , 使用 Number 类的 [intValue](#) 方法和 [doubleValue](#) 方法 , 将 d 对象分别转换为 int 类型变量和 double 类型变量 , 并输出转换后的结果。

第四 , 使用 Number 类的 [intValue](#) 方法和 [doubleValue](#) 方法 , 将 n 对象分别转换为 int 类型变量和 double 类型变量 , 并输出转换后的结果。

### • 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：构建包、类以及测试方法

首先在 day02 包下新建名为 TestNumber 的类 , 然后在该类中新建测试方法 `testIntValueAndDoubleValue` , 代码如下所示 :

```
package day02;
import org.junit.Test;
```

```
public class TestNumber {  
    /**  
     * 测试 Number 的 intValue 方法和 doubleValue 方法  
     */  
    @Test  
    public void testIntValueAndDoubleValue() {  
    }  
}
```

## 步骤二：获取 Number 类的对象所属类型

使用代码 d.getClass().getName() 和代码 n.getClass().getName() 获取 d 对象和 n 对象所属的数据类型。此处只需了解这样可以获取对象所属数据类型即可。代码如下所示：

```
package day02;  
  
import org.junit.Test;  
  
public class TestNumber {  
    /**  
     * 测试 Number 的 intValue 方法和 doubleValue 方法  
     */  
    @Test  
    public void testIntValueAndDoubleValue() {  
  
        Number d = 123.45;  
        Number n = 123;  
  
        // 输出 d 和 n 对象所属的类型  
        System.out.println(d.getClass().getName()); // java.lang.Double  
        System.out.println(n.getClass().getName()); // java.lang.Integer  
  
    }  
}
```

## 步骤三：运行

运行步骤二中的 testIntValueAndDoubleValue 方法，控制台输出结果如下所示：

```
java.lang.Double  
java.lang.Integer
```

从输出结果可以看出，d 的所属类型为 Double，n 的所属类型为 Integer，这是因为，123.45 该直接量的类型为 double，经过自动装箱以后转换为 Double 类型，Double 类型是 Number 类型的子类，子类型转换为父类类型属于自动类型转换，因此，d 的所属类型为 Double，同理，n 的所属类型为 Integer。

## 步骤四：将 d 对象分别转换为 int 类型变量和 double 类型变量

使用 Number 类的 intValue 方法和 doubleValue 方法，将 d 对象分别转换为 int 类型变量和 double 类型变量，并输出转换后的结果，代码如下所示：

```
package day02;
```

```
import org.junit.Test;

public class TestNumber {
    /**
     * 测试 Number 的 intValue 方法和 doubleValue 方法
     */
    @Test
    public void testIntValueAndDoubleValue() {
        Number d = 123.45;
        Number n = 123;

        // 输出 d 和 n 对象所属的类型
        System.out.println(d.getClass().getName()); // java.lang.Double
        System.out.println(n.getClass().getName()); // java.lang.Integer

        int intValue = d.intValue();
        double doubleValue = d.doubleValue();
        System.out.println(intValue + "," + doubleValue); // 123,123.45
    }
}
```

## 步骤五：运行

运行方法 `testIntValueAndDoubleValue`，最后一条输出语句的输出内容为：

123,123.45

从输出结果上可以看出，将原本 Double 类型的对象 d，转换为 int 类型后，取了 d 对象的整数部分。但转换为 double 类型后没有变化。

## 步骤六：将 n 对象分别转换为 int 类型变量和 double 类型变量

使用 Number 类的 `intValue` 方法和 `doubleValue` 方法，将 n 对象分别转换为 int 类型变量和 double 类型变量，并输出转换后的结果。

```
package day02;

import org.junit.Test;

public class TestNumber {
    /**
     * 测试 Number 的 intValue 方法和 doubleValue 方法
     */
    @Test
    public void testIntValueAndDoubleValue() {
        Number d = 123.45;
        Number n = 123;

        // 输出 d 和 n 对象所属的类型
        System.out.println(d.getClass().getName()); // java.lang.Double
        System.out.println(n.getClass().getName()); // java.lang.Integer

        int intValue = d.intValue();
        double doubleValue = d.doubleValue();
        System.out.println(intValue + "," + doubleValue); // 123,123.45
    }
}
```

```
    intValue = n.intValue();
    doubleValue = n.doubleValue();
    System.out.println(intValue + "," + doubleValue); // 123,123.0

}
```

### 步骤七：运行

再次运行 testIntValueAndDoubleValue 方法，最后一条输出语句的输出内容为：

```
123,123.0
```

从输出结果上可以看出，将原本 Integer 类型的对象 n，转换为 double 类型后，小数点后保留一位小数。但转换为 int 类型后没有变化。

### • 完整代码

本案例的完整代码如下所示：

```
package day02;

import org.junit.Test;

public class TestNumber {
    /**
     * 测试 Number 的 intValue 方法和 doubleValue 方法
     */
    @Test
    public void testIntValueAndDoubleValue() {
        Number d = 123.45;
        Number n = 123;

        // 输出 d 和 n 对象所属的类型
        System.out.println(d.getClass().getName()); // java.lang.Double
        System.out.println(n.getClass().getName()); // java.lang.Integer

        int intValue = d.intValue();
        double doubleValue = d.doubleValue();
        System.out.println(intValue + "," + doubleValue); // 123,123.45

        intValue = n.intValue();
        doubleValue = n.doubleValue();
        System.out.println(intValue + "," + doubleValue); // 123,123.0
    }
}
```

## 7. 测试 Integer 的 parseInt 方法

### • 问题

测试 Integer 的 parseInt 方法，即，首先将字符串 “123” 转换为 int 类型并输出结果，然后，将字符串“壹佰贰拾叁”转换为 int 类型，并查看运行效果。

## • 方案

首先，使用 Integer 的 parseInt 方法，将字符串 “123” 转换为 int 类型，正常运行并可以输出 int 类型的值 123；然后，再次使用 Integer 的 parseInt 方法，将字符串“壹佰贰拾叁”转换为 int 类型，运行后，会出现异常。

## • 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

在 day02 包下新建类 TestInteger，在该类中添加单元测试方法 testParseInt，代码如下所示：

```
package day02;

import org.junit.Test;

public class TestInteger {
    /**
     * 测试 Integer 的 parseInt 方法
     */
    @Test
    public void testParseInt() {
    }
}
```

### 步骤二：将字符串 “123” 转成整数类型

使用 Integer 的 parseInt 方法，将字符串 “123” 转换为 int 类型，正常运行并可以输出 int 类型的值 123，代码如下所示：

```
package day02;

import org.junit.Test;

public class TestInteger {
    /**
     * 测试 Integer 的 parseInt 方法
     */
    @Test
    public void testParseInt() {

        String str = "123";
        int value = Integer.parseInt(str);
        System.out.println(value); // 123
    }
}
```

运行上述代码，控制台输出结果为：

说明已经成功的将字符串 "123" 转换成 int 类型的 123。

### 步骤三：字符串"壹佰贰拾叁"转换为 int 类型

再次使用 Integer 的 parseInt 方法，将字符串"壹佰贰拾叁"转换为 int 类型，代码如下所示：

```
package day02;

import org.junit.Test;

public class TestInteger {
    /**
     * 测试 Integer 的 parseInt 方法
     */
    @Test
    public void testParseInt() {
        String str = "123";
        int value = Integer.parseInt(str);
        System.out.println(value); // 123

        str = "壹佰贰拾叁";
        // 会抛出 NumberFormatException
        value = Integer.parseInt(str);

    }
}
```

运行上述代码后，会出现异常：

```
java.lang.NumberFormatException: For input string: "壹佰贰拾叁"
```

从异常情况可以看出，Java 程序无法将字符串"壹佰贰拾叁"转换为 int 类型，因此，抛出了 java.lang.NumberFormatException 异常。所以在使用 Integer 的 parseInt 方法时，要注意传入的方法的参数为数字。

#### • 完整代码

本案例中，类 TestInteger 的完整代码如下所示：

```
package day02;

import org.junit.Test;

public class TestInteger {
    /**
     * 测试 Integer 的 parseInt 方法
     */
    @Test
    public void testParseInt() {
        String str = "123";
```

```
int value = Integer.parseInt(str);
System.out.println(value); // 123

str = "壹佰贰拾叁";
// 会抛出 NumberFormatException
// value = Integer.parseInt(str);
}
```

## 8. 测试 Double 的 parseDouble 方法

- 问题

测试 Double 的 parseDouble 方法，即，首先将字符串“12345.00”转换为 double 类型并输出结果，然后，将字符串“¥12345.00”转换为 double 类型，并查看运行效果。

- 方案

首先，使用 Double 的 parseDouble 方法，将字符串“12345.00”转换为 double 类型，正常运行并可以输出 double 类型的值 12345.00；然后，再次使用 Double 的 parseDouble 方法，将字符串“¥12345.00”转换为 double 类型，运行后，会出现异常。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

在 TestInteger 类中添加单元测试方法 testParseDouble，代码如下所示：

```
package day02;

import org.junit.Test;

public class TestInteger {

    /**
     * 测试 Double 的 parseDouble 方法
     */
    @Test
    public void testParseDouble() {
    }
}
```

### 步骤二：将字符串“12345.00”转成 double 类型

使用 Double 的 parseDouble 方法，将字符串“12345.00”转换为 double 类型，正常运行并可以输出 double 类型的值 12345.0，代码如下所示：

```
package day02;
```

```
import org.junit.Test;

public class TestInteger {
    /**
     * 测试 Double 的 parseDouble 方法
     */
    @Test
    public void testParseDouble() {

        String str = "12345.00";
        double value = Double.parseDouble(str); // 12345.0
        System.out.println(value);

    }
}
```

运行上述代码，控制台输出结果为：

```
12345.0
```

说明已经成功的将字符串 “12345.00” 转换成 double 类型的 12345.0。

### 步骤三：字符串“¥12345.00”转换为 double 类型

再次使用 Double 的 parseDouble 方法 将字符串“¥12345.00”转换为 double 类型，  
代码如下所示：

```
package day02;

import org.junit.Test;

public class TestInteger {
    /**
     * 测试 Double 的 parseDouble 方法
     */
    @Test
    public void testParseDouble() {
        String str = "12345.00";
        double value = Double.parseDouble(str); // 12345.0
        System.out.println(value);

        str = "¥12345.00";
        // 会抛出 NumberFormatException
        // value = Double.parseDouble(str);

    }
}
```

运行上述代码后，会出现异常：

```
java.lang.NumberFormatException: For input string: "¥12345.00"
```

从异常情况可以看出，Java 程序无法将字符串“¥12345.00”转换为 double 类型，因此，抛出了 `java.lang.NumberFormatException` 异常。所以在使用 `Double` 的 `parseDouble` 方法时，要注意传入的方法的参数为数字。

- **完整代码**

本案例中，类 `TestInteger` 的完整代码如下所示：

```
package day02;

import org.junit.Test;

public class TestInteger {
    //... (之前案例的代码, 略)

    /**
     * 测试 Double 的 parseDouble 方法
     */
    @Test
    public void testParseDouble() {
        String str = "12345.00";
        double value = Double.parseDouble(str); // 12345.0
        System.out.println(value);

        str = "¥12345.00";

        // 会抛出 NumberFormatException
        // value = Double.parseDouble(str);
    }
}
```

## 课后作业

- 编写 RegExUtils 类，提供静态方法实现对身份证号码、邮政编码、手机号码的合法性校验
- 重写员工类（Emp）的 equals 和 toString 方法

在面向对象的课程中，我们曾使用过 Emp 类，本案例要求重写 Emp 类的 toString 方法和 equals 方法，详细要求如下：

- 重写 Emp 类的 toString 方法，返回 Emp 对象的名字、年龄、性别和工资信息，例如：张三，23，男，5000。
- 重写 Emp 类的 equals 方法，两个 Emp 引用变量相等的条件是名字相等。

### 3. 输入数字字符串，转换为整数或浮点数

用户从控制台接收一个字符串，通过程序判断该字符串是整数，还是小数。如果既不是整数也不是小数，程序输出“数字格式不正确”。程序交互过程如下所示：

用户输入整数字符串的情况，如图-1 所示：



图- 2

用户输入为小数的情况，如图-2 所示：

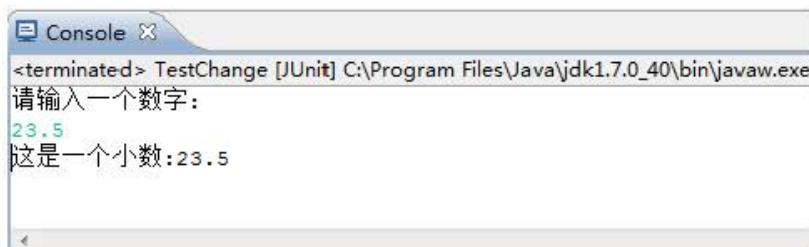


图- 3

用户输入的既不是整数也不是小数的情况，如图-3 所示：

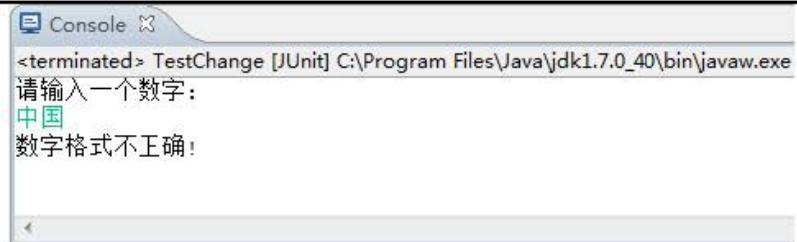


图- 4

#### 4. 简述自动装箱和拆箱的原理

# Java 核心 API(上)

## Unit03

知识体系.....**Page 90**

日期操作	Date 及其常用 API	Java 中的时间
		Date 类简介
		setTime 和 getTime 方法
		Date 重写 toString 方法
	SimpleDateFormat	SimpleDateFormat 简介
		日期模式匹配字符
		将 Date 格式化为 String
		将 String 解析为 Date
	Calendar	Calendar 简介
		getInstance 方法
		设置日期及时间分量
		获取日期及时间分量
		getActualMaximum 方法
		add 方法
		setTime 与 getTime 方法
	集合框架	List 和 Set
		集合持有对象的引用
		add 方法
		contains 方法
		size、clear、isEmpty

经典案例.....**Page 99**

使用 setTime 和 getTime 方法操作时间毫秒	setTime 和 getTime 方法
使用 format 方法格式化日期	将 Date 格式化为 String
使用 parse 方法将字符串解析为日期	将 String 解析为 Date
测试 getInstance 方法	getInstance 方法
调用 set 方法设置日期分量	设置日期及时间分量
调用 get 方法获取日期分量	获取日期及时间分量
输出某一年的各个月份的天数	getActualMaximum 方法

输出一年后再减去 3 个月的日期	add 方法
使 Date 表示的日期与 Calendar 表示的日期进行互换	setTime 和 getTime 方法
测试集合持有对象	集合持有对象的引用
使用 add 方法向集合中添加元素	add 方法
使用 contains 方法判断集合中是否包含某元素	contains 方法
测试方法 size、clear、isEmpty 的用法	size、clear、isEmpty

课后作业.....Page 138

## 1. 日期操作

### 1.1. Date 及其常用 API

#### 1.1.1. 【Date 及其常用 API】Java 中的时间

知识讲解

#### Java中的时间

- Java中的时间使用标准类库的Date类表示，是用距离一个固定时间点的毫秒数（可正可负，long类型）表达一个特定的时间点；
- 固定的时间点叫纪元（epoch），是UTC时间1970年1月1日 00:00:00；
- UTC（Universal Time Coordinated世界协调时间）与GMT（Greenwich Mean Time格林威治时间）一样，是一种具有实际目的的科学标准时间。

+

#### 1.1.2. 【Date 及其常用 API】Date 类简介

知识讲解

#### Date类简介

- java.util.Date 类封装日期及时间信息。
- Date类的大多数用于进行时间分量计算的方法已经被Calendar取代。

```
Date date = new Date();
// 系统当前的日期及时间信息
System.out.println(date);
// Sun Jan 06 11:52:55 CST 2013
long time = date.getTime();
//1970年1月1日至今的毫秒数
```

+

#### 1.1.3. 【Date 及其常用 API】setTime 和 getTime 方法

知识讲解

#### setTime和getTime方法

```
/** 使用setTime和getTime设置及获取时间 */
public void testSetTime() {
    Date date = new Date();
    // 输出当天此时此刻的日期和时间
    System.out.println(date);
    long time = date.getTime();
    // 增加一天所经历的毫秒数
    time += 60 * 60 * 24 * 1000;
    date.setTime(time);
    // 输出明天此时此刻的日期和时间
    System.out.println(date);
}
```

+

知识讲解

**setTime和getTime方法 ( 续1 )**

```
/** 获取当前系统时间 */
public void testGetTime() {
    Date date = new Date();
    System.out.println(date);

    // 1970年1月1日零时距此刻的毫秒数
    long time = date.getTime();
    System.out.println(time);
}
```

+

#### 1.1.4. 【Date 及其常用 API】Date 重写 toString 方法

知识讲解

**Date重写toString方法**

- Date重写了toString()方法，用一个字符串来描述当前 Date 对象所表示的时间。格式如下:
- Mon Feb 17 15:36:55 CST 2014

+

### 1.2. SimpleDateFormat

#### 1.2.1. 【SimpleDateFormat】SimpleDateFormat 简介

知识讲解

**SimpleDateFormat简介**

- java.text.SimpleDateFormat是一个以与语言环境有关的方式来格式化和解析日期的具体类。它允许进行格式化 (日期 -> 文本)、解析 (文本 -> 日期) 和规范化
- 构造方法
  - SimpleDateFormat ( )
  - SimpleDateFormat ( String pattern ) 用给定的模式和默认语言环境的日期格式符号构造SimpleDateFormat
- 方法
  - final String format ( Date date ) **Date => String**
  - Date parse(String source) throws ParseException **String => Date**

+

### 1.2.2. 【SimpleDateFormat】日期模式匹配字符

日期模式匹配字符		
字符	含义	示例
y	年	yyyy年—2013年 ; yy—13年
M	月	MM月—01月 ; M月—1月
d	日	dd日—06日 ; d日—6日
E	星期	E—星期日 ( Sun )
a	AM或PM标识	a—下午 ( PM )
H	小时 ( 24小时制 )	
h	小时 ( 12小时制 )	a h时—下午 12时
m	分钟	HH:mm:ss—12:46:33
s	秒	hh(a):mm:ss—12(下午):47:48

### 1.2.3. 【SimpleDateFormat】将 Date 格式化为 String

将Date格式化为String		
知识讲解	<pre>/** 日期格式化 */ public void testFormat() {     SimpleDateFormat sdf = new         SimpleDateFormat("yyyy-MM-dd HH:mm:ss");     Date date = new Date();     String dateStr = sdf.format(date);     System.out.println(dateStr); }</pre>	

### 1.2.4. 【SimpleDateFormat】将 String 解析为 Date

将String解析为Date		
知识讲解	<pre>/**和format方法相反，parse方法用于按照特定格式将表示时间 的字符串转换为Date对象*/ public void testParse() throws Exception {     String str = "2013-12-25";     SimpleDateFormat sdf =         new SimpleDateFormat("yyyy-MM-dd");     Date date = sdf.parse(str);     System.out.println(date); }</pre>	

### 1.3. Calendar

#### 1.3.1. 【Calendar】Calendar 简介

知识讲解

#### Calendar简介

• java.util.Calendar 类用于封装日历信息，其主要作用在于其方法可以对时间分量进行运算；  
• Calendar是抽象类，其具体子类针对不同国家的日历系统，其中应用最广泛的是GregorianCalendar（格里高里历，即通用的阳历），对应世界上绝大多数国家/地区使用的标准日历系统。

+

#### 1.3.2. 【Calendar】getInstance 方法

知识讲解

#### getInstance方法

• Calendar提供了一个类方法getInstance，以获得此类型的一个通用的对象  
• Calendar的getInstance方法返回一个Calendar对象，其日历字段已由当前日期和时间初始化  
`Calendar c = Calendar.getInstance();`

+

知识讲解

#### getInstance方法（续1）

```
/** 使用Calendar及子类获取时间 */
public void testGetInstance() {
    Calendar c = Calendar.getInstance();
    // 输出Calendar对象所属的实际类型
    System.out.println(c.getClass().getName());
    // getTime方法返回对应的Date对象
    System.out.println(c.getTime());
    // 创建GregorianCalendar对象
    GregorianCalendar c1 =
        new GregorianCalendar(2013, Calendar.DECEMBER, 25);
    System.out.println(c1.getTime());
}
```

+

### 1.3.3. 【Calendar】设置日期及时间分量

知识讲解

#### 设置日期及时间分量

```
/** 设置日期及分量 */
public void testSet() {
    Calendar c = Calendar.getInstance();
    c.set(Calendar.YEAR, 2014);
    c.set(Calendar.MONTH, Calendar.DECEMBER);
    c.set(Calendar.DATE, 25);
    System.out.println(c.getTime());
    //Thu Dec 25 16:02:08 CST 2014
    c.set(Calendar.DATE, 32);
    System.out.println(c.getTime());
    //Thu Jan 01 16:02:08 CST 2015
}
```

+

### 1.3.4. 【Calendar】获取日期及时间分量

知识讲解

#### 获取日期及时间分量

- 使用Calendar提供的get方法及一些常量可以获取日期及时间分量
- static int YEAR 指示年份的字段数字
- static int MONTH 指示月份的字段数字
- static int DATE 指示一个月份中的第几天
- static int DAY\_OF\_WEEK  
指示一个星期中的某天，1为星期日

+

知识讲解

#### 获取日期及时间分量（续1）

```
/** 获取时间及分量 */
public void testGet() {
    Calendar c = Calendar.getInstance();
    c.set(Calendar.YEAR, 2014);
    c.set(Calendar.MONTH, Calendar.DECEMBER);
    c.set(Calendar.DATE, 25);

    int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);
    System.out.println(dayOfWeek);
    // 输出为5，表示周四，周日为每周的第一天.
}
```

+

### 1.3.5. 【Calendar】getActualMaximum 方法

#### getActualMaximum方法

- int getActualMaximum ( int field ) 给定此Calendar 的时间值，返回指定日历字段可能拥有的最大值，例如：

```
int year = 2014;
Calendar c = Calendar.getInstance();
c.set(Calendar.YEAR, year);
c.set(Calendar.DATE, 1);
for (int month = Calendar.JANUARY;
     month <= Calendar.DECEMBER; month++) {
    c.set(Calendar.MONTH, month);
    System.out.println(year + "年"
                       + (month + 1) + "月： "
                       + c.getActualMaximum (Calendar.DATE)
                       + "天");
}
```

知识讲解



### 1.3.6. 【Calendar】add 方法

#### add方法

- void add ( int field,int mount ) 为给定的时间分量的值加上给定的值，若给定的值为负数则是减去给定的值

/\*输出一年后再减去3个月的日期\*/

```
public void testAdd(){
    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.YEAR, 1); //加一年
    calendar.add(Calendar.MONTH, -3); //减3个月
    System.out.println("year:" + calendar.get(Calendar.YEAR));
    System.out.println("month:" +
                       (calendar.get(Calendar.MONTH)+1));
    System.out.println("day:" +
                       calendar.get(Calendar.DAY_OF_MONTH));
}
```

知识讲解



### 1.3.7. 【Calendar】setTime 与 getTime 方法

#### setTime与getTime方法

- Date getTime()

使用Date描述Calendar表示的日期并返回

- void setTime(Date d)

使Calendar表示Date所描述的日期

/\*通过Date对象设置日期，在获取日期\*/

```
public void testsetTimeAndGetTime(){
    Calendar calendar = Calendar.getInstance();
    Date date = new Date();
    calendar.setTime(date);
    date = calendar.getTime();
}
```

知识讲解



## 2. 集合框架

### 2.1. Collection

#### 2.1.1. 【Collection】List 和 Set

**List和Set**

知识讲解

- 在实际开发中，需要将使用的对象存储于特定数据结构的容器中。JDK提供了这样的容器——集合(Collection)。Collection是一个接口，定义了集合相关操作方法，其有两个子接口：List与Set
- List:可重复集      元素是否重复，取决于元素的equals()比较的结果
- Set:不可重复集

Tarena  
Technology

```

classDiagram
    class Collection {
        <<interface>>
    }
    class List {
        <<interface>>
    }
    class Set {
        <<interface>>
    }
    Collection <|-- List
    Collection <|-- Set
  
```

#### 2.1.2. 【Collection】集合持有对象的引用

**集合持有对象的引用**

知识讲解

- 集合中存储的都是引用类型元素，并且集合只保存每个元素对象的引用，而并非将元素对象本身存入集合。

**集合持有对象的引用（续1）**

知识讲解

```

public void testRef() {
    Collection<Cell> cells = new ArrayList<Cell>();
    cells.add(new Cell(1, 2));
    Cell cell = new Cell(2, 3);
    cells.add(cell);
    System.out.println(cell); // (2,3)
    System.out.println(cells); // [(1,2), (2,3)]

    cell.drop();
    System.out.println(cell); // (3,3)
    System.out.println(cells); // [(1,2), (3,3)]
}
  
```

Tarena  
Technology

### 2.1.3. 【Collection】add 方法

+ 知识讲解

add方法

Tarena  
达内科技

- Collection定义了一个add方法用于向集合中添加新元素。
  - boolean add(E e)
- 该方法会将给定的元素添加进集合，若添加成功则返回true,否则返回false

+

+ 知识讲解

add方法 ( 续1 )

Tarena  
达内科技

```
public void testAdd() {
    Collection<String> c = new ArrayList<String>();
    System.out.println(c); // []
    c.add("a");
    c.add("b");
    c.add("c");
    System.out.println(c); // [a, b, c]
}
```

+

### 2.1.4. 【Collection】contains 方法

+ 知识讲解

contains方法

Tarena  
达内科技

- boolean contains(Object o)
- 该方法会用于判断给定的元素是否被包含在集合中。若包含则返回true,否则返回false。
- 这里需要注意的是，集合在判断元素是否被包含在集合中是根据每个元素的equals()方法进行比较后的结果。
- 通常有必要重写equals()保证contains()方法的合理结果

+

### contains方法 (续1)

```
public void testContains() {
    Collection<Cell> cells = new ArrayList<Cell>();
    cells.add(new Cell(1, 2));
    cells.add(new Cell(1, 3));
    cells.add(new Cell(2, 2));
    cells.add(new Cell(2, 3));
    Cell cell = new Cell(1, 3);
    // List集合contains方法和对象的equals方法相关
    boolean flag = cells.contains(cell);
    // 如果Cell不重写equals方法将为false
    System.out.println(flag); // true
}
```

知识讲解



### 2.1.5. 【Collection】size、clear、isEmpty

### size、clear、isEmpty

- int size()

该方法用于返回当前集合中的元素总数。

知识讲解

- void clear()

该方法用于清空当前集合。

- boolean isEmpty()

该方法用于判断当前集合中是否不包含任何元素



### size、clear、isEmpty (续1)

```
public void testSizeAndClearAndIsEmpty() {
    Collection<String> c = new HashSet<String>();
    System.out.println(c.isEmpty()); // true
    c.add("java"); c.add("cpp"); c.add("php");
    c.add("c#"); c.add("objective-c");
    System.out.println(
        "isEmpty:" + c.isEmpty() + ",size: " + c.size()
    ); // isEmpty:false, size: 5
    c.clear();
    System.out.println(
        "isEmpty:" + c.isEmpty() + ",size: " + c.size()
    ); // isEmpty:true, size: 0
}
```

知识讲解



## 经典案例

### 1. 使用 setTime 和 getTime 方法操作时间毫秒

- 问题

使用 setTime 和 getTime 方法操作毫秒表示的日期-时间，详细要求如下：

- 1) 获取 1970 年 1 月 1 日零时距当前时刻的毫秒数。
- 2) 获取明天此时此刻的日期-时间。

- 方案

- 1) 首先，实例化 Date 类的对象获取当前日期-时间，代码如下所示：

```
Date date = new Date();
```

然后，调用 Date 类的 getTime 方法，获取 1970 年 1 月 1 日零时距当前时刻的毫秒数，代码如下：

```
long time = date.getTime();
```

2) 首先，实例化 Date 类的对象获取当前日期-时间 date 对象，然后，调用 Date 类的 getTime 方法，获取 1970 年 1 月 1 日零时距当前时刻的毫秒数，并计算当前日期-时间增加一天后的毫秒数，最后，调用 Date 类的 setTime 方法将当前日期-时间 date 设置为增加一天后的日期-时间。

- 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：构建测试方法

首先，在名为 JavaSE 的工程下的 src 下新建名为 day03 的包，然后，在该包下新建类 TestDate，在该类中添加测试方法 testGetTime，代码如下所示：

```
package day03;

import java.util.Date;
import org.junit.Test;

public class TestDate {
    /**
     * 测试 getTime 方法
     */
    @Test
    public void testGetTime() {
    }
}
```

## 步骤二：获取 1970 年 1 月 1 日零时距当前时刻的毫秒数

首先，实例化 Date 类的对象获取当前时间；然后，调用 Date 类的 `getTime` 方法，获取 1970 年 1 月 1 日零时距当前时刻的毫秒数，代码如下所示：

```
package day03;

import java.util.Date;
import org.junit.Test;

public class TestDate {
    /**
     * 测试 testGetTime 方法
     */
    @Test
    public void testGetTime() {

        Date date = new Date();
        System.out.println(date);

        // 1970 年 1 月 1 日零时距此刻的毫秒数
        long time = date.getTime();

        System.out.println(time);

    }
}
```

## 步骤三：运行

运行 `testGetTime` 方法，运行结果如下所示：

```
Mon Jan 13 17:56:52 CST 2014
1389607012302
```

观察输出结果，运行 `testGetTime` 方法时的日期-时间为 2014 年 1 月 13 日 17:56:52，1970 年 1 月 1 日零时距该时刻的毫秒数为 1389607012302。

## 步骤四：实现当前日期-时间基础上增加一天

首先，获取当前日期-时间 `date`，然后，调用 Date 类的 `getTime` 方法，获取 1970 年 1 月 1 日零时距当前时刻的毫秒数，并计算当前日期-时间增加一天后的毫秒数，最后，调用 Date 类的 `setTime` 方法将当前日期-时间 `date` 设置为增加一天后的日期和时间，代码如下所示：

```
package day03;

import java.util.Date;
import org.junit.Test;

public class TestDate {
    /**
     * 测试 testGetTime 方法
     */
}
```

```
/*
@Test
public void testGetTime() {
    Date date = new Date();
    System.out.println(date);

    // 1970 年 1 月 1 日零时距此刻的毫秒数
    long time = date.getTime();

    System.out.println(time);
}

/**
 * 测试 testSetTime 方法
 */
@Test
public void testSetTime() {
    Date date = new Date();

    // 输出当天此时此刻的日期和时间
    System.out.println(date);

    long time = date.getTime();

    // 增加一天所经历的毫秒数
    time += 60 * 60 * 24 * 1000;

    date.setTime(time);

    // 输出明天此时此刻的日期和时间
    System.out.println(date);
}

}
```

## 步骤五：运行

运行 testSetTime 方法，运行结果如下所示：

```
Mon Jan 13 18:02:38 CST 2014
Tue Jan 14 18:02:38 CST 2014
```

观察输出结果，运行 testSetTime 方法时的时间日期为 2014 年 1 月 13 日 18:02:38，  
在此时间基础上增加一天后的时间为 2014 年 1 月 14 日 18:02:38。

### • 完整代码

本案例的完整代码如下所示：

```
package day03;

import java.util.Date;
import org.junit.Test;

public class TestDate {
    /**
```

```
* 测试 testGetTime 方法
*/
@Test
public void testGetTime() {
    Date date = new Date();
    System.out.println(date);

    // 1970 年 1 月 1 日零时距此刻的毫秒数
    long time = date.getTime();

    System.out.println(time);
}

/**
 * 测试 testSetTime 方法
 */
@Test
public void testSetTime() {
    Date date = new Date();

    // 输出当天此时此刻的日期和时间
    System.out.println(date);

    long time = date.getTime();

    // 增加一天所经历的毫秒数
    time += 60 * 60 * 24 * 1000;

    date.setTime(time);

    // 输出明天此时此刻的日期和时间
    System.out.println(date);
}
```

## 2. 使用 format 方法格式化日期

- 问题

从上一案例中可以看出，输出日期-时间时显示的默认格式为“Tue Jan 14 18:02:38 CST 2014”，这种输出格式不符合我们日常看时间的形式，我们日常看的时间格式为“2014年1月13日 18:02:38”或者为“2014-01-13 18:02:38”。

本案例要求将当前时间日期以“2014-01-13 18:02:38”这样的格式输出到控制台。

- 方案

1) 实例化 `SimpleDateFormat` 类的对象，该类可以使用户选择自定义的日期-时间格式的模式，代码如下所示：

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```

以上代码中，设置的日期-时间格式为“yyyy-MM-dd HH:mm:ss”。

2) 首先，实例化 `Date` 类的对象，获取当前日期-时间；然后，调用 `SimpleDateFormat` 类的 `format` 方法，将 `Date` 类的对象格式化为日期-时间字符串，代码如下所示：

```
Date date = new Date();
String dateStr = sdf.format(date);
```

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：新建类及测试方法

首先新建名为 `TestSimpleDateFormat` 的类，并在该类中新建测试方法 `testFormat`，代码如下所示：

```
package day03;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.junit.Test;

public class TestSimpleDateFormat {
    /**
     * 测试 format 方法
     */
    @Test
    public void testFormat() {
    }
}
```

### 步骤二：设置自定义的日期-时间格式

实例化 `SimpleDateFormat` 类的对象，该类可以使用户选择自定义的日期-时间格式的模式，代码如下所示：

```
package day03;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.junit.Test;

public class TestSimpleDateFormat {
    /**
     * 测试 format 方法
     */
    @Test
    public void testFormat() {

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    }
}
```

### 步骤三：使用 `format` 方法将 `Date` 类的对象格式化为日期-时间字符串

首先，实例化 `Date` 类的对象，获取当前日期-时间；然后，调用 `SimpleDateFormat` 类的 `format` 方法，将 `Date` 类的对象格式化为日期-时间字符串，代码如下所示：

```
package day03;
```

```
import java.text.SimpleDateFormat;
import java.util.Date;
import org.junit.Test;

public class TestSimpleDateFormat {
    /**
     * 测试 format 方法
     */
    @Test
    public void testFormat() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        Date date = new Date();
        String dateStr = sdf.format(date);
        System.out.println(dateStr);

    }
}
```

#### 步骤四：运行

运行 testFormat 方法，运行结果如下所示：

```
2014-01-15 16:03:41
```

观察输出结果，可以看出已经将当前日期时间转换为“yyyy-MM-dd HH:mm:ss”格式。

#### • 完整代码

本案例的完整代码如下所示：

```
package day03;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.junit.Test;

public class TestSimpleDateFormat {
    /**
     * 测试 format 方法
     */
    @Test
    public void testFormat() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date date = new Date();
        String dateStr = sdf.format(date);
        System.out.println(dateStr);
    }
}
```

### 3. 使用 parse 方法将字符串解析为日期

#### • 问题

在上一案例的基础上，将字符串表示的日期“2013-12-25”转换为 Date 类型表示的日

期。

- 方案

1 ) 实例化 `SimpleDateFormat` 类的对象 , 该类可以使用户选择自定义的日期 - 时间格式的模式 , 代码如下所示 :

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
```

以上代码中 , 设置的日期 - 时间格式为 "yyyy-MM-dd" , 这是因为 , 我们要转换的时间格式 "2013-12-25" 要和 `SimpleDateFormat` 所构造的格式匹配 , 才可以进行转换。

2 ) 调用 `SimpleDateFormat` 类的 `parse` 方法 , 将日期字符串转换为 `Date` 类的对象 , 代码如下所示 :

```
Date date = sdf.parse(str);
```

- 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一 : 构建测试方法及 `SimpleDateFormat` 类的对象

首先 , 在 `TestSimpleDateFormat` 类中新建测试方法 `testParse` ; 然后 , 实例化 `SimpleDateFormat` 类的对象 , 该类可以使用户选择自定义的日期 - 时间格式的模式 , 代码如下所示 :

```
package day03;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.junit.Test;

public class TestSimpleDateFormat {

    /**
     * 测试 parse 方法
     */
    @Test
    public void testParse() throws Exception {
        String str = "2013-12-25";
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    }

}
```

请注意 , 在此处 `SimpleDateFormat` 构造的日期格式要和字符串 `str` 日期格式相匹配。

#### 步骤二 : 将日期字符串转换为 `Date` 类的对象

调用 `SimpleDateFormat` 类的 `parse` 方法 , 将日期字符串转换为 `Date` 类的对象 , 代码如下所示 :

```
package day03;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.junit.Test;

public class TestSimpleDateFormat {
    /**
     * 测试 parse 方法
     */
    @Test
    public void testParse() throws Exception {
        String str = "2013-12-25";
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

        Date date = sdf.parse(str);
        System.out.println(date);

    }
}
```

### 步骤三：运行

运行 `testParse` 方法，控制台输出结果如下：

```
Wed Dec 25 00:00:00 CST 2013
```

查看输出结果，可以是按照默认时间格式输出的日期-时间。

- **完整代码**

本案例中，类 `TestSimpleDateFormat` 的完整代码如下所示：

```
package day03;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.junit.Test;

public class TestSimpleDateFormat {
    //... (之前案例的代码，略)

    /**
     * 测试 parse 方法
     */
    @Test
    public void testParse() throws Exception {
        String str = "2013-12-25";
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        Date date = sdf.parse(str);
        System.out.println(date);
    }
}
```

## 4. 测试 `getInstance` 方法

- **问题**

Calendar 类是一个抽象类，它为特定瞬间与一组诸如 YEAR、MONTH、DAY\_OF\_MONTH、HOUR 等日历字段之间的转换提供了一些方法，并为操作日历字段提供了一些方法。

本案例要求获取 Calendar 类的实例，获取实例后做如下操作：

- 1 ) 获取 Calendar 实例所属的实际类型，并输出。
- 2 ) 将 Calendar 对象转换为 Date 对象，并输出该 Date 对象。
- 3 ) 使用 GregorianCalendar 构建对象，该对象对应的日期为 2013 年 12 月 25 日，将 GregorianCalendar 对象转换为 Date 对象，并输出该 Date 对象。

### • 方案

1 ) 首先，使用 Calendar 类的 getInstance 方法获取 Calendar 类的对象，然后使用下列代码获取该对象所属的实际类型，并输出，代码如下所示：

```
System.out.println(c.getTime());
```

2 ) 使用 Calendar 类的 getTime 方法，可以将 Calendar 对象转换为 Date 对象，然后，输出该 Date 类的对象。

3 ) 使用 GregorianCalendar 构建日期为 2013 年 12 月 25 日的对象，然后，GregorianCalendar 类的 getTime 方法将 GregorianCalendar 对象转换为 Date 对象，并输出该 Date 对象。

### • 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：创建测试方法

首先新建类 TestCalendar；然后，在该类下新建测试方法 testGetInstance，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 getInstance 方法
     */
    @Test
    public void testGetInstance() {
    }
}
```

#### 步骤二：创建 Calendar 类的实例

首先，使用 Calendar 类的 getInstance 方法获取 Calendar 类的对象，然后使用下

列代码获取该对象所属的实际类型，并输出，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 getInstance 方法
     */
    @Test
    public void testGetInstance() {

        Calendar c = Calendar.getInstance();
        // 输出 Calendar 对象所属的实际类型
        System.out.println(c.getClass().getName());
    }
}
```

### 步骤三：运行

运行 testGetInstance 方法，控制台输出结果如下所示：

```
java.util.GregorianCalendar
```

从控制台的输出结果可以看出，对象 c 所属的实际类型为 GregorianCalendar，这是因为，历史上有着许多种纪年方法，它们的差异实在太大了，比如说一个人的生日是“七月七日”，那么一种可能是阳（公）历的七月七日，它也可以是阴（农）历的七月七日。为了统一计时，全世界通常选择最普及、最通用的日历：Gregorian Calendar（格里高历），也就是我们所说的公历。

Calendar 类本身是一个抽象类，它是所有日历类的模板，并提供了一些所有日历通用的方法；但它本身不能直接实例化对象，程序只能创建 Calendar 子类的实例，Java 本身提供一个 GregorianCalendar 类，一个代表格里高历的子类，它代表了我们通常所说的公历。Calendar 类提供了几个静态 getInstance 方法来获取 Calendar 对象，这些方法根据 TimeZone、Locale 类来获取特定的 Calendar，如果不指定 TimeZone、Locale，则使用默认的 TimeZone、Locale 来创建 Calendar 对象。

本案例中使用的 getInstance 方法刚好使用了默认的 TimeZone、Locale 来创建 Calendar 对象，通过输出结果可以看出默认的 TimeZone、Locale 创建的 Calendar 对象所属的实际类型为 GregorianCalendar。

### 步骤四：将 Calendar 对象转换为 Date 对象

使用 Calendar 类的 getTime 方法，可以将 Calendar 对象转换为 Date 对象，然后，输出该 Date 类的对象，代码如下所示：

```
package day03;
```

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 getInstance 方法
     */
    @Test
    public void testGetInstance() {
        Calendar c = Calendar.getInstance();

        // 输出 Calendar 对象所属的实际类型
        System.out.println(c.getClass().getName());

        // getTime 方法返回对应的 Date 对象
        System.out.println(c.getTime());
    }
}
```

## 步骤五：使用 GregorianCalendar 构建日期对象

使用 GregorianCalendar 构建日期为 2013 年 12 月 25 日的对象，然后，GregorianCalendar 类的 getTime 方法将 GregorianCalendar 对象转换为 Date 对象，并输出该 Date 对象，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 getInstance 方法
     */
    @Test
    public void testGetInstance() {
        Calendar c = Calendar.getInstance();

        // 输出 Calendar 对象所属的实际类型
        System.out.println(c.getClass().getName());

        // getTime 方法返回对应的 Date 对象
        System.out.println(c.getTime());

        // 创建 GregorianCalendar 对象
        GregorianCalendar c1 = new GregorianCalendar(2013, Calendar.DECEMBER, 25);
        System.out.println(c1.getTime());
    }
}
```

## 步骤六：运行

运行 `testGetInstance` 方法，控制台输出结果如下所示：

```
java.util.GregorianCalendar
Mon Jan 20 10:40:39 CST 2014
Wed Dec 25 00:00:00 CST 2013
```

观察上述代码可以看出已经将 `Calendar` 对象和 `GregorianCalendar` 对象转换为 `Date` 对象。

总结前几个案例可以知道时间的表示方式可以是 `Date`、`long` 或者 `Calendar`，它们之间的相互转换如图-1 所示。

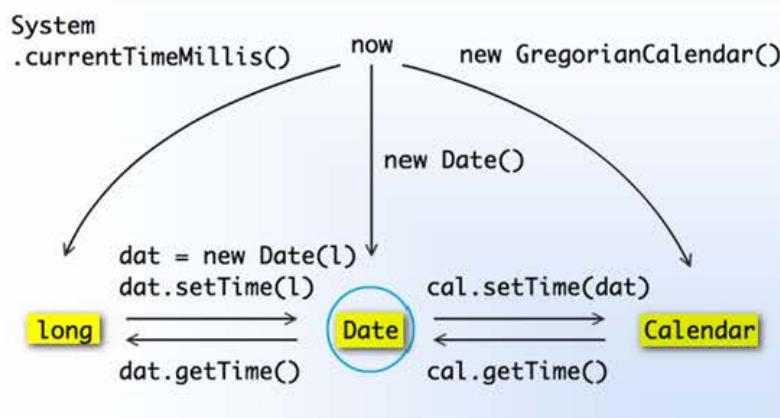


图- 1

### • 完整代码

本案例中，类 `TestCalendar` 的完整代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 getInstance 方法
     */
    @Test
    public void testGetInstance() {
        Calendar c = Calendar.getInstance();

        // 输出 Calendar 对象所属的实际类型
        System.out.println(c.getClass().getName());

        // getTime 方法返回对应的 Date 对象
        System.out.println(c.getTime());

        // 创建 GregorianCalendar 对象
        GregorianCalendar c1 = new GregorianCalendar(2013, Calendar.DECEMBER,
                25);
    }
}
```

```
        System.out.println(c1.getTime());  
    }  
}
```

## 5. 调用 set 方法设置日期分量

- 问题

在上一案例的基础上，使用 `Calendar` 类的 `set` 方法设置日期-时间的各个分量，详细要求如下：

- 1 ) 使用 `Calendar` 类表示时间，设置日期为 2013 年 12 月 25 日，时间为此时此刻的时间，并将该日期-时间日期转换为 `Date` 类输出。
- 2 ) 修改 `Calendar` 类的对象，将日期中的日设置为 32，其余不变，然后将该日期-时间日期转换为 `Date` 类输出。

- 方案

使用 `Calendar` 类的 `set` 方法设置日期时间的各个分量，该方法声明如下：

```
void set ( int field , int value )
```

该方法表示将给定的日历字段设置为给定值。该方法需要一个 `int` 类型的 `field` 参数，`field` 是 `Calendars` 类的静态字段，如 `Calendar.YEAR`、`Calendar.MONTH` 等分别代表了年、月、日、小时、分钟、秒等时间字段。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：添加测试方法

在 `TestCalendar` 类中添加测试方法 `testSet`，代码如下所示：

```
package day03;  
  
import java.util.Calendar;  
import java.util.GregorianCalendar;  
import org.junit.Test;  
  
public class TestCalendar {  
  
    /**  
     * 测试 set 方法  
     */  
    @Test  
    public void testSet() {  
    }  
}
```

## 步骤二：使用 Calendar 类的 set 方法设置日期分量

首先，通过 Calendar 类的 getInstance 方法获取当前日期-时间；然后，使用 set 方法将日期改变为 2013 年 12 月 25 日，并将 Calendar 表示的时间转换为 Date 表示的时间输出到控制台，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 set 方法
     */
    @Test
    public void testSet() {

        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, 2013);
        c.set(Calendar.MONTH, Calendar.DECEMBER);
        c.set(Calendar.DATE, 25);
        System.out.println(c.getTime()); // Wed Dec 25 16:45:01 CST 2013
    }
}
```

## 步骤三：运行

运行 testSet 方法，控制台输出结果为：

```
Wed Dec 25 14:53:53 CST 2013
```

从输出结果可以看出已经将日期设置为 2013 年 12 月 25 日，时间为当前时间。

## 步骤四：将步骤二中的对象 c 的日设置 31

使用 set 方法将步骤二中的对象 c 的“日”设置为 32，并将 Calendar 表示的日期-时间转换为 Date 表示的日期-时间输出到控制台，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 set 方法
     */
    @Test
    public void testSet() {
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, 2013);
        c.set(Calendar.MONTH, Calendar.DECEMBER);
        c.set(Calendar.DATE, 25);
```

```
System.out.println(c.getTime()); // Wed Dec 25 16:45:01 CST 2013
```

```
// 实际日期值为 1 月 1 日
c.set(Calendar.DATE, 32);
System.out.println(c.getTime()); // Wed Jan 01 16:45:01 CST 2014

}
```

## 步骤五：运行

运行 testSet 方法，控制台输出结果为：

```
Wed Dec 25 14:53:53 CST 2013
Wed Jan 01 14:53:53 CST 2014
```

可以看出将日设置为 32 后，日期变化 2014 年 1 月 1 日，这是因为，当被修改的字段超出它允许的范围时，会发生进位，即上一级字段会变大。在本案例中，将日设置为 32 后，但是，2013 年 12 月只有 31 天，超出了日允许的范围，因此发生进位，变为 1 月 1 日。

### • 完整代码

本案例中，类 TestCalendar 的完整代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    //... (之前案例的代码，略)

    /**
     * 测试 set 方法
     */
    @Test
    public void testSet() {
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, 2013);
        c.set(Calendar.MONTH, Calendar.DECEMBER);
        c.set(Calendar.DATE, 25);
        System.out.println(c.getTime()); // Wed Dec 25 16:45:01 CST 2013

        // 实际日期值为 1 月 1 日
        c.set(Calendar.DATE, 32);
        System.out.println(c.getTime()); // Wed Jan 01 16:45:01 CST 2014
    }
}
```

## 6. 调用 get 方法获取日期分量

### • 问题

在上一案例的基础上，使用 `Calendar` 类的 `get` 方法获取 `Calendar` 类表示的日期-时间的各个分量，详细要求为：首先，使用 `Calendar` 类表示时间，设置日期为 2013 年 12 月 25 日，时间为此时此刻的时间，然后，获取该日期-时间是周几。

- 方案

使用 `Calendar` 类的 `get` 方法获取 `Calendar` 类表示的时间-日期各个分量，该方法的声明如下所示：

```
int get( int field )
```

该方法表示返回指定日历字段的值。

- 步骤

实现此案例需要按照如下步骤进行。

**步骤一：**

首先，为 `TestCalendar` 类添加测试方法 `testGet`，然后，在该方法中，首先，通过 `Calendar` 类的 `getInstance` 方法获取当前日期-时间；然后，使用 `set` 方法将日期改变为 2013 年 12 月 25 日，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {

    /**
     * 测试 get 方法
     */
    @Test
    public void testGet() {
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, 2013);
        c.set(Calendar.MONTH, Calendar.DECEMBER);
        c.set(Calendar.DATE, 25);
    }
}
```

**步骤二：使用 `get` 方法**

使用 `Calendar` 类的 `get` 方法获取 2013 年 12 月 25 日对应的星期几，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
```

```
import org.junit.Test;

public class TestCalendar {
    /**
     * 测试 get 方法
     */
    @Test
    public void testGet() {
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, 2013);
        c.set(Calendar.MONTH, Calendar.DECEMBER);
        c.set(Calendar.DATE, 25);

        int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);

        System.out.println(dayOfWeek); // 输出为 4，表示周三，周日为每周的第 1 天.

    }
}
```

### 步骤三：运行

运行 testGet 方法，控制台输出结果如下所示：

4

可以看出控制台的输出结果为 4，表示周三，周日为每周的第 1 天。

#### • 完整代码

本案例中，类 TestCalendar 的完整代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    //... (之前案例的代码，略)

    /**
     * 测试 get 方法
     */
    @Test
    public void testGet() {
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, 2013);
        c.set(Calendar.MONTH, Calendar.DECEMBER);
        c.set(Calendar.DATE, 25);

        int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);

        System.out.println(dayOfWeek); // 输出为 4，表示周三，周日为每周的第 1 天.

    }
}
```

## 7. 输出某一年的各个月份的天数

- **问题**

在上一案例的基础上，获取 2013 年各个月份的天数。

- **方案**

首先，使用 Calendar 类的 getInstance 方法获取当前日期-时间；然后，修改该日期-时间为 2013 年，日为 1 日；最后，使用循环，循环从 1 月循环到 12 月，在循环中修改当前日期-时间对象的月份，可以使用 getActualMaximum 方法获取各个月份的天数，该方法的声明如下：

```
getActualMaximum(int field)
```

该方法表示给定此 Calendar 的时间值，返回指定日历字段可能拥有的最大值。可以指定日期字段为 Calendar.DATE。

- **步骤**

### 步骤一：构建测试方法

首先，使用 Calendar 类的 getInstance 方法获取当前日期-时间；然后，修改该日期-时间为 2013 年，日为 1 日，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {

    /**
     * 输出某一年的各个月份的天数
     */
    @Test
    public void testGetActualMaximum() {
        int year = 2013;
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, year);
        c.set(Calendar.DATE, 1);
    }
}
```

### 步骤二：使用 getActualMaximum 方法

使用循环，循环从 1 月循环到 12 月，在循环中修改当前日期-时间对象的月份，可以使用 getActualMaximum 方法获取各个月份的天数，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 输出某一年的各个月份的天数
     */
    @Test
    public void testGetActualMaximum() {
        int year = 2013;
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, year);
        c.set(Calendar.DATE, 1);

        for (int month = Calendar.JANUARY; month <= Calendar.DECEMBER; month++) {
            c.set(Calendar.MONTH, month);
            System.out.println(year + "年" + (month + 1) + "月：" +
                + c.getActualMaximum(Calendar.DATE) + "天");
        }
    }
}
```

此处需要注意的是，Calenar.MONTH 字段表示月份，月份的起始值不是 1，而是 0，所示需要在取到的月份值的基础上加 1 才能与实际的月份对应。

### 步骤三：运行

运行 testGetActualMaximum 方法，控制台输出结果如下所示：

```
2013 年 1 月 : 31 天
2013 年 2 月 : 28 天
2013 年 3 月 : 31 天
2013 年 4 月 : 30 天
2013 年 5 月 : 31 天
2013 年 6 月 : 30 天
2013 年 7 月 : 31 天
2013 年 8 月 : 31 天
2013 年 9 月 : 30 天
2013 年 10 月 : 31 天
2013 年 11 月 : 30 天
2013 年 12 月 : 31 天
```

从输出结果可以看出，已经将 2013 年的各个月对应的天数获取到了。

### • 完整代码

本案例的完整代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    //... (之前案例的代码, 略)

    /**
     * 输出某一年的各个月份的天数
     */
    @Test
    public void testGetActualMaximum() {
        int year = 2013;
        Calendar c = Calendar.getInstance();
        c.set(Calendar.YEAR, year);
        c.set(Calendar.DATE, 1);

        for (int month = Calendar.JANUARY; month <= Calendar.DECEMBER; month++) {
            c.set(Calendar.MONTH, month);
            System.out.println(year + "年" + (month + 1) + "月: "
                + c.getActualMaximum(Calendar.DATE) + "天");
        }
    }
}
```

## 8. 输出一年后再减去 3 个月的日期

- 问题

在上一案例的基础上，输出当前日期基础上一年后再减去 3 个月的日期的年、月、日。

- 方案

首先 使用 `Calendar` 类的 `getInstance` 方法获取当前日期-时间 然后 使用 `Calendar` 类的 `add` 方法，加上一年；最后，再次，使用 `add` 方法，加上-3 月，即减去 3 个月并输出计算后的年月日。

- 步骤

### 步骤一：构建测试方法

首先，在 `TestCalendar` 类中添加测试方法 `testAdd`，然后，使用 `Calendar` 类的 `getInstance` 方法获取当前日期-时间；最后，使用 `Calendar` 类的 `add` 方法，加上一年，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
```

```
* 输出一年后再减去 3 个月的日期
*/
@Test
public void testAdd() {
    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.YEAR, 1); // 加一年
}

}
```

## 步骤二：使用 add 方法，实现减去 3 个月

再次，使用 add 方法，加上 -3 月，即减去 3 个月并输出计算后的年月日，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 输出一年后再减去 3 个月的日期
     */
    @Test
    public void testAdd() {
        Calendar calendar = Calendar.getInstance();
        calendar.add(Calendar.YEAR, 1); // 加一年

        calendar.add(Calendar.MONTH, -3); // 减 3 个月
        System.out.println("year:" + calendar.get(Calendar.YEAR));
        System.out.println("month:" + (calendar.get(Calendar.MONTH) + 1));
        System.out.println("day:" + calendar.get(Calendar.DAY_OF_MONTH));

    }
}
```

此处需要注意的是，Calenar.MONTH 字段表示月份，月份的起始值不是 1，而是 0，所示需要在取到的月份值的基础上加 1 才能与实际的月份对应。

## 步骤三：运行

运行 testAdd 方法，控制台输出结果如下所示：(注：我的运行时间为 2014 年 2 月 24 日)

```
year:2014
month:11
day:24
```

从输出结果可以看出，实现了输出当前日期基础上一年后再减去 3 个月的日期的年、月、日。

- **完整代码**

本案例的完整代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    //... (之前案例的代码, 略)

    /**
     * 输出一年后再减去 3 个月的日期
     */
    @Test
    public void testAdd() {
        Calendar calendar = Calendar.getInstance();
        calendar.add(Calendar.YEAR, 1); // 加一年
        calendar.add(Calendar.MONTH, -3); // 减 3 个月
        System.out.println("year:" + calendar.get(Calendar.YEAR));
        System.out.println("month:" + (calendar.get(Calendar.MONTH) + 1));
        System.out.println("day:" + calendar.get(Calendar.DAY_OF_MONTH));
    }
}
```

## 9. 使 Date 表示的日期与 Calendar 表示的日期进行互换

- **问题**

在上一案例的基础上，使 Date 表示的日期与 Calendar 表示的日期进行互换。

- **方案**

首先，使用 Calendar 类的 getInstance 方法获取当前日期-时间 calendar；然后，实例化 Date 类，获取表示的当前日期-时间对象 date；最后，首先，使用 Calendar 的 setTime 方法将 Date 对象转换为 Calendar 对象，然后，使用 Calendar 对象的 getTime 方法再将 Calendar 表示对象转换为 Date 对象。

- **步骤**

### 步骤一：构建测试方法

首先，在 TestCalendar 类中添加测试方法 testSetTimeAndgetTime；然后，使用 Calendar 类的 getInstance 方法获取当前日期-时间 calendar；最后，实例化 Date 类，获取表示的当前日期-时间对象 date，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
```

```
import org.junit.Test;

public class TestCalendar {

    /**
     * 使 Date 表示的日期与 Calendar 表示的日期进行互换
     */
    @Test
    public void testSetTimeAndGetTime() {
        Calendar calendar = Calendar.getInstance();
        Date date = new Date();
    }

}
```

## 步骤二：使用 setTime 方法

首先，使用 Calendar 的 setTime 方法将 Date 对象转换为 Calendar 对象，然后，使用 Calendar 对象的 getTime 方法再将 Calendar 表示对象转换为 Date 对象，代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    /**
     * 使 Date 表示的日期与 Calendar 表示的日期进行互换
     */
    @Test
    public void testSetTimeAndGetTime() {
        Calendar calendar = Calendar.getInstance();
        Date date = new Date();

        calendar.setTime(date); // 将 Date 转换为 Calendar
        date = calendar.getTime(); // 将 Calendar 转换为 Date

    }
}
```

### • 完整代码

本案例的完整代码如下所示：

```
package day03;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import org.junit.Test;

public class TestCalendar {
    // ... (之前案例的代码，略)
```

```
/*
 * 使 Date 表示的日期与 Calendar 表示的日期进行互换
 */
@Test
public void testSetTimeAndGetTime() {
    Calendar calendar = Calendar.getInstance();
    Date date = new Date();
    calendar.setTime(date); // 将 Date 转换为 Calendar
    date = calendar.getTime(); // 将 Calendar 转换为 Date
}
```

## 10. 测试集合持有对象

- 问题

测试集合持有对象，详细要求如下：

- 1 ) 使用 ArrayList 构造集合对象 cells。
- 2 ) 构造行和列为 ( 1 , 2 ) 的 Cell 对象，将其放入集合 cells 中。
- 3 ) 构造行和列为 ( 2 , 3 ) 的 Cell 类的对象 cell1，将其放入集合 cells 中。
- 4 ) 输出 cell1 对象和 cells 对象。
- 5 ) 将 cell1 对象下落一个格子。
- 6 ) 再次输出 cell1 对象和 cells 对象，比较两次输出的结果。

- 方案

首先，构建包和类，并在类中新建测试方法。

其次，使用 ArrayList 构造集合对象 cells，代码如下：

```
Collection<Cell> cells = new ArrayList<Cell>();
```

以上代码表明集合存放的数据为 Cell 类型。

第三，构造行和列为 ( 1 , 2 ) 的 Cell 对象，使用集合的 add 方法将该对象放入集合 cells 中。

第四，构造行和列为 ( 2 , 3 ) 的 Cell 类的对象 cell1，使用集合的 add 方法将该对象放入集合 cells 中，并输出 cell1 对象和 cells 对象。

第五，将 cell1 对象下落一个格子，然后再次输出 cell1 对象和 cells 对象。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建包、类以及测试方法

首先新建名为 TestCollection 的类，最后，在该类中新建测试方法 testRef，代码如下所示：

```

package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 集合持有对象的引用
     */
    @Test
    public void testRef() {
    }
}

```

### 步骤二：拷贝 Cell 类

拷贝 day02 包下的 Cell 类到 day03 包下，工程结构如图-2 所示。

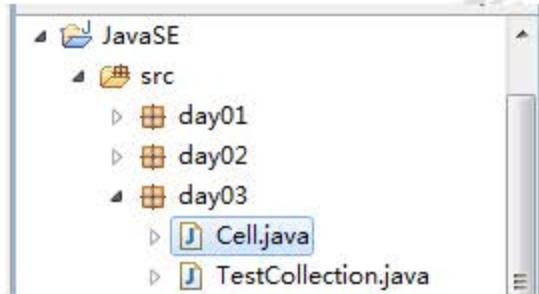


图-2

### 步骤三：构建集合

使用 ArrayList 构造集合对象 cells，代码如下所示：

```

package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 集合持有对象的引用
     */
    @Test
    public void testRef() {

        Collection<Cell> cells = new ArrayList<Cell>();

    }
}

```

### 步骤四：构造两个 Cell 类的对象，将其放入集合

首先，构造行和列为 (1, 2) 的 Cell 对象，使用集合的 add 方法将该对象放入集合 cells 中。其次，构造行和列为 (2, 3) 的 Cell 类的对象 cell，使用集合的 add 方法将

该对象放入集合 `cells` 中，并输出 `cell` 对象和 `cells` 对象，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 集合持有对象的引用
     */
    @Test
    public void testRef() {
        Collection<Cell> cells = new ArrayList<Cell>();

        cells.add(new Cell(1, 2));
        Cell cell = new Cell(2, 3);
        cells.add(cell);
        System.out.println(cell); // (2,3)
        System.out.println(cells); // [(1,2), (2,3)]
    }
}
```

## 步骤五：运行

运行方法 `testRef`，控制台输出内容为：

```
(2,3)
[(1,2), (2,3)]
```

从输出结果上可以看出，构造的 `cell` 对象的行和列为( 2 , 3 )，在集合中显示也为( 2 , 3 )。

## 步骤六：将 `cell` 对象下落一个格子

将 `cell` 对象下落一个格子，然后再次输出 `cell` 对象和 `cells` 对象。

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 集合持有对象的引用
     */
    @Test
    public void testRef() {
        Collection<Cell> cells = new ArrayList<Cell>();
        cells.add(new Cell(1, 2));
        Cell cell = new Cell(2, 3);
        cells.add(cell);
        System.out.println(cell); // (2,3)
        System.out.println(cells); // [(1,2), (2,3)]
    }
}
```

```
    cell.drop();
    System.out.println(cell); // (3,3)
    System.out.println(cells); // [(1,2), (3,3)]  

}  
}
```

## 步骤七：运行

再次运行 testRef 方法，控制台的输出内容为：

```
(2,3)
[(1,2), (2,3)]  
  
(3,3)
[(1,2), (3,3)]
```

从输出结果上可以看出，将 cell 对象下落一个格子后，其行和列变为 (3,3)，在集合中该对象的行和列也为 (3,3)，因此可以说明，集合 cells 持有对象 cell 的引用，cell 对象的属性发生变化，cells 集合中该对象的属性也发生变化。

### • 完整代码

本案例中，TestCollection 类的完整代码如下所示：

```
package day03;  
  
import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;  
  
public class TestCollection {
    /**
     * 集合持有对象的引用
     */
    @Test
    public void testRef() {
        Collection<Cell> cells = new ArrayList<Cell>();
        cells.add(new Cell(1, 2));
        Cell cell = new Cell(2, 3);
        cells.add(cell);
        System.out.println(cell); // (3,3)
        System.out.println(cells); // [(1,2), (3,3)]  
  
        cell.drop();
        System.out.println(cell); // (3,3)
        System.out.println(cells); // [(1,2), (3,3)]
    }
}
```

Cell 类的完整代码如下所示：

```
package day03;  
  
public class Cell{
    int row;
    int col;
```

```

public Cell(int row, int col) {
    this.row = row;
    this.col = col;
}

public Cell() {
    this(0, 0);
}

public Cell(Cell cell) {
    this(cell.row, cell.col);
}

public void drop() {
    row++;
}

public void moveRight() {
    col++;
}

public void moveLeft() {
    col--;
}

@Override
public String toString() {
    return "(" + row + "," + col + ")";
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (this == obj) {
        return true;
    }
    if (obj instanceof Cell) {
        Cell cell = (Cell) obj;
        return cell.row == row && cell.col == col;
    } else {
        return false;
    }
}
}

```

## 11. 使用 add 方法向集合中添加元素

- 问题

使用 add 方法向集合中添加元素，详细要求如下：

- 1 ) 使用 ArrayList 构造集合对象 c，并输出该对象。
- 2 ) 将字符串 “a”、“b”、“c” 放入集合 c 中，再次输出集合对象。

- 方案

首先，使用 ArrayList 构造集合对象 c，代码如下所示：

```
Collection<String> c = new ArrayList<String>();
```

接着输出对象 c。

然后，使用 Collection 的 add 方法将字符串 “a”、“b”、“c” 分别放入集合中，最后，再次输出集合 c。

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

在上一案例的基础上，在类 TestCollection 中构建测试方法 testAdd，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {

    /**
     * 测试 add 方法
     */
    @Test
    public void testAdd() {
    }

}
```

### 步骤二：构造集合对象

首先，使用 ArrayList 构造集合对象 c，然后输出对象 c，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 add 方法
     */
    @Test
    public void testAdd() {

        Collection<String> c = new ArrayList<String>();
        System.out.println(c); // []

    }
}
```

运行上述代码，控制台输出结果为：

观察上述输出结果，可以看出集合 c 中当前没有任何元素。

### 步骤三：使用 add 方法向集合中添加元素

首先，使用 Collection 的 add 方法将字符串 “a”、“b”、“c” 分别放入集合中，然后，再次输出集合 c，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 add 方法
     */
    @Test
    public void testAdd() {
        Collection<String> c = new ArrayList<String>();
        System.out.println(c); // []

        c.add("a");
        c.add("b");
        c.add("c");
        System.out.println(c); // [a, b, c]
    }
}
```

运行上述代码后，控制台的输出结果为：

[a, b, c]

从输出结果可以看出，已经将字符串 “a”、“b”、“c” 放入到集合 c 中。

#### • 完整代码

本案例中，类 TestCollection 的完整代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    //... (之前案例的代码，略)

    /**
     * 测试 add 方法
     */
    @Test
    public void testAdd() {
        Collection<String> c = new ArrayList<String>();
```

```
System.out.println(c); // []
c.add("a");
c.add("b");
c.add("c");
System.out.println(c); // [a, b, c]
}
}
```

## 12. 使用 contains 方法判断集合中是否包含某元素

- 问题

判断某集合中是否包含某元素，详细要求如下：

- 1) 使用 ArrayList 构建集合对象 cells，使用行和列为 (1, 2) (1, 3) (2, 2) (2, 3) 构建四个 Cell 类的对象，并将这四个对象放入集合 cells 中。
- 2) 使用行和列为 (1, 3) 构建 Cell 类的对象 cell。
- 3) 比较 cell 对象在集合 cells 中是否存在。

- 方案

1) 首先，使用 ArrayList 构建集合对象 cells；然后，使用行和列为 (1, 2) (1, 3) (2, 2) (2, 3) 构建四个 Cell 类的对象，并将这四个对象放入集合 cells 中，代码如下所示：

```
Collection<Cell> cells = new ArrayList<Cell>();

cells.add(new Cell(1, 2));
cells.add(new Cell(1, 3));
cells.add(new Cell(2, 2));
cells.add(new Cell(2, 3));
```

2) 使用行和列为 (1, 3) 构建 Cell 类的对象 cell，代码如下所示：

```
Cell cell = new Cell(1, 3);
```

3) 使用 Collection 接口中提供的 contains 方法 判断 cell 对象是否存在于 cells 集合中，代码如下所示：

```
boolean flag = cells.contains(cell);
```

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

在 TestCollection 类中添加单元测试方法 testContains，代码如下所示：

```
package day03;
```

```
import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {

    /**
     * 测试 contains 方法
     */
    @Test
    public void testContains() {

    }

}
```

## 步骤二：构建集合对象及其元素

首先，使用 `ArrayList` 构建集合对象 `cells`；然后，使用行和列为 (1, 2) (1, 3) (2, 2) (2, 3) 构建四个 `Cell` 类的对象，并将这四个对象放入集合 `cells` 中，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 contains 方法
     */
    @Test
    public void testContains() {

        Collection<Cell> cells = new ArrayList<Cell>();

        cells.add(new Cell(1, 2));
        cells.add(new Cell(1, 3));
        cells.add(new Cell(2, 2));
        cells.add(new Cell(2, 3));

    }
}
```

## 步骤三：构建 `Cell` 类的对象 `cell`

使用行和列为 (1, 3) 构建 `Cell` 类的对象 `cell`，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 contains 方法
     */
}
```

```
/*
@Test
public void testContains() {
    Collection<Cell> cells = new ArrayList<Cell>();

    cells.add(new Cell(1, 2));
    cells.add(new Cell(1, 3));
    cells.add(new Cell(2, 2));
    cells.add(new Cell(2, 3));

    Cell cell = new Cell(1, 3);

}
```

#### 步骤四：使用 contains 方法判断集合中是否存在某元素

使用 Collection 接口中提供的 contains 方法，判断 cell 对象是否存在于 cells 集合中，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 contains 方法
     */
    @Test
    public void testContains() {
        Collection<Cell> cells = new ArrayList<Cell>();

        cells.add(new Cell(1, 2));
        cells.add(new Cell(1, 3));
        cells.add(new Cell(2, 2));
        cells.add(new Cell(2, 3));

        Cell cell = new Cell(1, 3);

        // List 集合 contains 方法和对象的 equals 方法相关
        boolean flag = cells.contains(cell);
        // 如果 Cell 不重写 equals 方法将为 false
        System.out.println(flag); // true

    }
}
```

#### 步骤五：运行

运行上述代码后，控制台输出结果如下：

```
true
```

从输出结果可以看出，返回结果为 true。判断集合中是否包含某个对象，是根据对象

的 `equals` 方法进行判断的，当集合中的某个对象和当前对象进行比较，如果这两个对象使用 `equals` 方法进行比较返回 `true`，那么说明集合中存在当前对象；反之，则返回 `false`，表示不存在当前对象。

本案例中的 `Cell` 类，在 `day02` 中已经重写了 `equals` 方法，实现了行行相等，同时列列相等，那么两个 `Cell` 对象相等。因此，在集合 `cells` 中存在 `cell` 对象，返回结果为 `true`。

### • 完整代码

本案例中，类 `TestCollection` 的完整代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.Test;

public class TestCollection {
    //... (之前案例的代码, 略)

    /**
     * 测试 contains 方法
     */
    @Test
    public void testContains() {
        Collection<Cell> cells = new ArrayList<Cell>();

        cells.add(new Cell(1, 2));
        cells.add(new Cell(1, 3));
        cells.add(new Cell(2, 2));
        cells.add(new Cell(2, 3));

        Cell cell = new Cell(1, 3);

        // List 集合 contains 方法和对象的 equals 方法相关
        boolean flag = cells.contains(cell);
        // 如果 Cell 不重写 equals 方法将为 false
        System.out.println(flag); // true
    }
}
```

`Cell` 类的完整代码如下所示：

```
package day03;

public class Cell{
    int row;
    int col;

    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public Cell() {
        this(0, 0);
    }
}
```

```
public Cell(Cell cell) {
    this(cell.row, cell.col);
}

public void drop() {
    row++;
}

public void moveRight() {
    col++;
}

public void moveLeft() {
    col--;
}

@Override
public String toString() {
    return "(" + row + "," + col + ")";
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (this == obj) {
        return true;
    }
    if (obj instanceof Cell) {
        Cell cell = (Cell) obj;
        return cell.row == row && cell.col == col;
    } else {
        return false;
    }
}
```

### 13. 测试方法 size、clear、isEmpty 的用法

- 问题

测试 Collection 接口中 size 方法、clear 方法、isEmpty 方法的用法，详细要求如下：

- 1) 使用 HashSet 构建集合 c，并判断当前集合是否为空。
- 2) 将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 c 中，并判断集合 c 是否为空以及集合 c 中元素的个数。
- 3) 清空集合 c，并判断集合 c 是否为空以及集合 c 中元素的个数。

- 方案

1) 首先，使用 HashSet 构建集合 c；然后，使用 Collection 接口提供的 isEmpty 方法判断集合 c 是否为空，代码如下所示：

```
Collection<String> c = new HashSet<String>();
System.out.println(c.isEmpty()); // true
```

2) 首先，使用 Collection 接口提供的 add 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 c 中；然后，使用 Collection 接口提供的 isEmpty 方法判断集合 c 是否为空；最后，使用 Collection 接口提供的 size 方法获取集合 c 中元素的个数，代码如下所示：

```
c.add("java");
c.add("cpp");
c.add("php");
c.add("c#");
c.add("objective-c");
System.out.println("isEmpty:" + c.isEmpty() + ",size: " + c.size());
```

3) 首先，使用 Collection 接口提供的 clear 方法清空集合 c，然后，使用 isEmpty 方法和 size 方法，判断集合 c 是否为空以及获取集合中元素的个数。

```
c.clear();
System.out.println("isEmpty:" + c.isEmpty() + ", size: " + c.size());
```

- **步骤**

实现此案例需要按照如下步骤进行。

#### 步骤一：构建测试方法

在类 TestCollection 中添加测试方法 testSizeAndClearAndIsEmpty，代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {

    /**
     * 测试 size 方法、clear 方法、isEmpty 方法
     */
    @Test
    public void testSizeAndClearAndIsEmpty() {
    }

}
```

#### 步骤二：使用 isEmpty 方法判断集合 c 是否为空

首先，使用 HashSet 构建集合 c；然后，使用 Collection 接口提供的 isEmpty 方法判断集合 c 是否为空，代码如下所示：

```
package day03;
```

```

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 size 方法、clear 方法、isEmpty 方法
     */
    @Test
    public void testSizeAndClearAndIsEmpty() {

        Collection<String> c = new HashSet<String>();
        System.out.println(c.isEmpty()); // true

    }
}

```

运行 `testSizeAndClearAndIsEmpty` 方法，控制台输出结果如下：

```
true
```

观察输出结果，可以看出当前集合 `c` 为空。

### 步骤三：使用 size 方法获取集合中元素的个数

首先，使用 `Collection` 接口提供的 `add` 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 `c` 中；然后，使用 `Collection` 接口提供的 `isEmpty` 方法判断集合 `c` 是否为空；最后，使用 `Collection` 接口提供的 `size` 方法获取集合 `c` 中元素的个数，代码如下所示：

```

package day03;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 size 方法、clear 方法、isEmpty 方法
     */
    @Test
    public void testSizeAndClearAndIsEmpty() {
        Collection<String> c = new HashSet<String>();
        System.out.println(c.isEmpty()); // true

        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");

        System.out.println("isEmpty:" + c.isEmpty() + ",size: " + c.size());
        // isEmpty:false, size: 5
    }
}

```

#### 步骤四：运行

运行 testSizeAndClearAndIsEmpty 方法，运行结果如下所示：

```
true  
  
isEmpty:false,size: 5
```

观察输出结果，可以看出集合 c 不再是空，而且其中有 5 个元素，所以其 size 为 5。

#### 步骤五： clear 方法的使用

首先，使用 Collection 接口提供的 clear 方法清空集合 c；然后，使用 isEmpty 方法和 size 方法，判断集合 c 是否为空以及获取集合 c 中元素的个数，代码如下所示：

```
package day03;  
  
import java.util.ArrayList;  
import java.util.Collection;  
import java.util.HashSet;  
import org.junit.Test;  
  
public class TestCollection {  
    /**  
     * 测试 size 方法、clear 方法、isEmpty 方法  
     */  
    @Test  
    public void testSizeAndClearAndIsEmpty() {  
        Collection<String> c = new HashSet<String>();  
        System.out.println(c.isEmpty()); // true  
  
        c.add("java");  
        c.add("cpp");  
        c.add("php");  
        c.add("c#");  
        c.add("objective-c");  
  
        System.out.println("isEmpty:" + c.isEmpty() + ",size: " + c.size());  
        // isEmpty:false, size: 5  
  
        c.clear();  
  
        System.out.println("isEmpty:" + c.isEmpty() + ", size: " + c.size());  
        // isEmpty:true, size: 0  
    }  
}
```

#### 步骤六：运行

运行 testSizeAndClearAndIsEmpty 方法，运行结果如下所示：

```
true  
isEmpty:false,size: 5
```

**isEmpty:true, size: 0**

观察输出结果，集合 c 调用 clear 方法以后，将集合进行了清空，集合中的元素个数变为 0。

- **完整代码**

本案例的完整代码如下所示：

```
package day03;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {
    //... (之前案例的代码, 略)

    /**
     * 测试 size 方法、clear 方法、isEmpty 方法
     */
    @Test
    public void testSizeAndClearAndIsEmpty() {
        Collection<String> c = new HashSet<String>();
        System.out.println(c.isEmpty()); // true

        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");

        System.out.println("isEmpty:" + c.isEmpty() + ",size: " + c.size());
        // isEmpty:false, size: 5

        c.clear();

        System.out.println("isEmpty:" + c.isEmpty() + ", size: " + c.size());
        // isEmpty:true, size: 0
    }
}
```



## 课后作业

### 1. 商品促销日期计算程序

用户输入商品生产日期和保质期，通过程序计算促销日期。计算规则为：到保质期前 14 天所在周的周三为促销日。控制台交互情况如图-1 所示。

```
Console <terminated> TestCalendar.testDiscountDate [JUnit] C:\Program Files\Java\jdk1.7.0_40\bin\javaw.exe
请输入生产日期(yyyy-MM-dd) :
2014-03-01
请输入保质期(天数) :
30
促销日为:2014-03-19
```

图- 1

### 2. 下面代码输出结果是？

Cell 类的代码如下所示：

```
public class Cell{
    int row;
    int col;
    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }
    public void drop() {
        row++;
    }
    @Override
    public String toString() {
        return "(" + row + "," + col + ")";
    }
}
```

运行 testRef 方法，程序的输出结果是：( )。

```
public void testRef() {
    Collection<Cell> cells = new ArrayList<Cell>();
    Cell c0 = new Cell(5, 6);
    Cell c1 = new Cell(7, 9);
    cells.add(c0);
    cells.add(c1);
    System.out.println(c0 + "," + c1);
    System.out.println(cells);
    c0.drop();
    c1.drop();
    System.out.println(c0 + "," + c1);
    System.out.println(cells);
}
```

- 
- A . (5,6),(7,9)  
[(5,6), (7,9)]  
(6,6),(8,9)  
[(6,6), (8,9)]
  - B . (5,6),(7,9)  
[(5,6), (7,9)]  
(5,6),(7,9)  
[(5,6), (7,9)]
  - C . (5,6),(7,9)  
[(5,6), (7,9)]  
(6,6),(8,9)  
[(5,6), (7,9)]
  - D . (6,6),(8,9)  
[(6,6), (8,9)]  
(6,6),(8,9)  
[(6,6), (8,9)]

### 3. 简述 contains 方法和 equals 方法的关系

### 4. 下面代码输出的结果是 ?

请看如下代码 :

```
public void testSizeAndClearAndIsEmpty() {  
    Collection<String> c = new ArrayList<String>();  
    System.out.println("isEmpty:" + c.isEmpty());  
    c.add("terry");  
    c.add("allen");  
    c.add("jerry");  
    c.add("smith");  
    System.out.println("isEmpty:" + c.isEmpty() + ",size: " + c.size());  
    c.clear();  
    System.out.println("isEmpty:" + c.isEmpty() + ", size: " + c.size());  
}
```

运行 testSizeAndClearAndIsEmpty 方法 , 程序的输出结果是 : ( )。

- A . isEmpty:false  
isEmpty:true,size: 4  
isEmpty:false, size: 0
- B . isEmpty:true  
isEmpty:false,size: 4  
isEmpty:true, size: 4
- C . isEmpty:false  
isEmpty:true,size: 4  
isEmpty:false, size: 4
- D . isEmpty:true  
isEmpty:false,size: 4  
isEmpty:true, size: 0

# Java 核心 API(上)

## Unit04

知识体系.....Page 142

集合框架	Collection	addAll、containsAll
	Iterator	hasNext、next 方法
		remove 方法
		增强型 for 循环
	泛型机制	泛型在集合中的应用
集合操作——线性表	List	ArrayList 和 LinkedList
		get 和 set
		插入和删除
		subList
		List 转换为数组
		数组转换为 List
	List 排序	Collections.sort 方法实现排序
		Comparable
		Comparator
	队列和栈	Queue
		Deque

经典案例.....Page 153

测试方法 addAll、containsAll 的用法	addAll、containsAll
使用 Iterator 的 hasNext 方法、next 方法遍历集合	hasNext、next 方法
使用 Iterator 的 remove 方法移除元素	remove 方法
使用增强型 for 循环遍历集合	增强型 for 循环
测试 List 的 get 方法和 set 方法	get 和 set
测试向 List 中插入和删除元素	插入和删除
测试 List 的 subList 方法	subList
将 List 转换为数组	List 转换为数组
将数组转换为 List	数组转换为 List
使用 Collections.sort 方法实现排序	Collections.sort 方法实现排序
使用 Comparator 接口实现排序	Comparator

测试 Queue 的用法	Queue
测试 Deque 的用法	Deque

课后作业.....Page 199

达内IT培训集团

## 1. 集合框架

### 1.1. Collection

#### 1.1.1. 【Collection】 addAll、containsAll

知识讲解

**addAll、containsAll**

- boolean addAll(Collection<? extends E> c)

该方法需要我们传入一个集合，并将该集合中的所有元素添加到当前集合中。

如果此 collection 由于调用而发生更改，则返回 true

- boolean containsAll(Collection<?> c)

该方法用于判断当前集合是否包含给定集合中的所有元素，若包含则返回true。

知识讲解

**addAll、containsAll ( 续1 )**

```
public void testAddAllAndContainsAll() {
    Collection<String> c1 = new ArrayList<String>();
    c1.add("java");           c1.add("cpp");           c1.add("php");
    c1.add("c#");            c1.add("objective-c");
    System.out.println(c1); // [java, cpp, php, c#, objective-c]
    Collection<String> c2 = new HashSet<String>();
    c2.addAll(c1);
    System.out.println(c2); // [cpp, php, c#, java, objective-c]
    Collection<String> c3 = new ArrayList<String>();
    c3.add("java");           c3.add("cpp");
    System.out.println(c1.containsAll(c3)); // true
}
```

### 1.2. Iterator

#### 1.2.1. 【Iterator】 hasNext、next 方法

知识讲解

**hasNext、next方法**

- 迭代器用于遍历集合元素。获取迭代器可以使用 Collection 定义的方法：
  - Iterator iterator()
- 迭代器 Iterator 是一个接口，集合在重写 Collection 的 iterator() 方法时利用 内部类 提供了迭代器的实现。
- Iterator 提供了统一的遍历集合元素的方式，其提供了用于遍历集合的两个方法：
  - boolean hasNext(); 判断集合是否还有元素可以遍历。
  - E next(); 返回迭代的下一个元素

### hasNext、next方法 (续1)

```

public void testHasNextAndNext() {
    Collection<String> c = new HashSet<String>();
    c.add("java");           c.add("cpp");
    c.add("php");
    c.add("c#");           c.add("objective-c");
    Iterator<String> it = c.iterator();
    while (it.hasNext()) {
        String str = it.next();
        System.out.println(str);
    }
}

```

知识讲解



### 1.2.2. 【Iterator】remove 方法

#### remove方法

- 在使用迭代器遍历集合时，不能通过集合的remove方法删除集合元素，否则会抛出并发更改异常。我们可以通过迭代器自身提供的remove()方法来删除通过next()迭代出的元素。
  - void remove()
- 迭代器的删除方法是在原集合中删除元素。
- 这里需要注意的是，在调用remove方法前必须通过迭代器的next()方法迭代过元素，那么删除的就是这个元素。并且不能再次调用remove方法，除非再次调用next()后方可再次调用。



#### remove方法 (续1)

```

public void testRemove() {
    Collection<String> c = new HashSet<String>();
    c.add("java");           c.add("cpp");
    c.add("php");
    c.add("c#");           c.add("objective-c");
    System.out.println(c); // [cpp, php, c#, java, objective-c]
    Iterator<String> it = c.iterator();
    while (it.hasNext()) {
        String str = it.next();
        if (str.indexOf('c') != -1) {
            it.remove(); // 删除包含字母c的元素
        }
    }
    System.out.println(c); // [php, java]
}

```

知识讲解



### 1.2.3. 【Iterator】增强型 for 循环

增强型for循环

知识讲解

- Java5.0之后推出了一个新的特性，增强for循环，也称为新循环。该循环不适用于传统循环的工作，其只用于遍历集合或数组。
- 语法：

```
for(元素类型 e : 集合或数组){  
    循环体  
}
```
- 新循环并非新的语法，而是在编译过程中，编译器会将新循环转换为迭代器模式。所以新循环本质上是迭代器。

+

增强型for循环（续1）

知识讲解

```
public void testForeach() {  
    Collection<String> c = new HashSet<String>();  
    c.add("java");  
    c.add("cpp");  
    c.add("php");  
    c.add("c#");  
    c.add("objective-c");  
    for (String str : c) {  
        System.out.print(str.toUpperCase() + " ");  
    }  
    // CPP PHP C# JAVA OBJECTIVE-C  
}
```

+

## 1.3. 泛型机制

### 1.3.1. 【泛型机制】泛型在集合中的应用

泛型在集合中的应用

知识讲解

- 泛型是Java SE 5.0引入的特性，泛型的本质是参数化类型。在类、接口和方法的定义过程中，所操作的数据类型被传入的参数指定。
- Java泛型机制广泛的应用在集合框架中。所有的集合类型都带有泛型参数，这样在创建集合时可以指定放入集合中元素的类型。Java编译器可以据此进行类型检查，这样可以减少代码在运行时出现错误的可能性。

+

## 泛型在集合中的应用（续1）

- ArrayList类的定义中，<E>中的E为泛型参数，在创建对象时可以将类型作为参数传递，此时，类定义所有的E将被替换成传入的参数；
- ```
public class ArrayList<E> {
    ...
    public boolean add(E e) {...};
    public E get(int index) {...};
}
ArrayList<String> list = new ArrayList<String>();
list.add("One");
list.add(100);
```
- Java 编译器类型检查错误，此时add方法应传入的参数类型是String。

知识讲解



## 2. 集合操作——线性表

### 2.1. List

#### 2.1.1. 【List】ArrayList 和 LinkedList

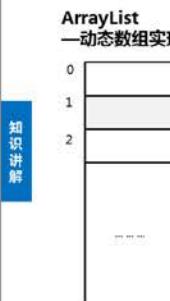
### ArrayList和LinkedList

- List接口是Collection的子接口，用于定义线性表数据结构。可以将List理解为存放对象的数组，只不过其元素个数可以动态的增加或减少。
- List接口的两个常见实现类为ArrayList和LinkedList，分别用动态数组和链表的方式实现了List接口。
- 可以认为ArrayList和LinkedList的方法在逻辑上完全一样，只是在性能上有一定的差别。ArrayList更适合于随机访问而LinkedList更适合于插入和删除。在性能要求不是特别苛刻的情形下可以忽略这个差别。

知识讲解



### ArrayList和LinkedList（续1）



知识讲解



## 2.1.2. 【List】get 和 set

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

知识讲解

Tarena  
Technology  
达内科技

### get和set

- List除了继承Collection定义的方法外，还根据其线性表的数据结构定义了一系列方法，其中最常用的就是基于下标的get和set方法：
  - E get(int index)  
获取集合中指定下标对应的元素，下标从0开始。
  - E set(int index, E element)  
将给定的元素存入给定位置，并将原位置的元素返回。

+

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

知识讲解

Tarena  
Technology  
达内科技

### get和set（续1）

```
public void testGetAndSet() {  
    List<String> list = new ArrayList<String>();  
    list.add("java");    list.add("cpp");    list.add("php");  
    list.add("c#");      list.add("objective-c");  
    // get方法遍历List  
    for (int i = 0; i < list.size(); i++) {  
        System.out.println(list.get(i).toUpperCase());  
    }  
    String value = list.set(1, "c++");  
    System.out.println(value); // cpp  
    System.out.println(list); // [java, c++, php, c#, objective-c]  
    list.set(1, list.set(3, list.get(1))); // 交换位置1和3上的元素  
    System.out.println(list); // [java, c#, php, c++, objective-c]  
}
```

+

## 2.1.3. 【List】插入和删除

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

知识讲解

Tarena  
Technology  
达内科技

### 插入和删除

- List根据下标的操作还支持插入与删除操作。
  - void add(int index, E element):  
将给定的元素插入到指定位置，原位置及后续元素都顺序向后移动。
  - E remove(int index):  
删除给定位置的元素，并将被删除的元素返回。

+

### 插入和删除 (续1)

```
public void testInsertAndRemove() {
    List<String> list = new ArrayList<String>();
    list.add("java");
    list.add("c#");
    System.out.println(list); // [java, c#]

    list.add(1, "cpp");
    System.out.println(list); // [java, cpp, c#]

    list.remove(2);
    System.out.println(list); // [java, cpp]
}
```

知识讲解

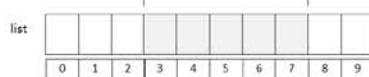


#### 2.1.4. 【List】 subList

### subList

- List的subList方法用于获取子List。
- 需要注意的是，subList获取的List与原List占有相同的存储空间，对子List的操作会影响原List。
- List<E> subList(int fromIndex, int toIndex);
- fromIndex和toIndex是截取子List的首尾下标（前包括，后不包括）

subList = list.subList(3, 8)



### subList (续1)

```
public void testSubList() {
    List<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < 10; i++) {
        list.add(i);
    }
    System.out.println(list); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    List<Integer> subList = list.subList(3, 8);
    System.out.println(subList); // [3, 4, 5, 6, 7]
    // subList获得的List和源List占有相同的数据空间
    for (int i = 0; i < subList.size(); i++) {
        subList.set(i, subList.get(i) * 10);
    }
    System.out.println(subList); // [30, 40, 50, 60, 70]
    System.out.println(list); // [0, 1, 2, 30, 40, 50, 60, 70, 8, 9]
    list.subList(3, 8).clear(); // 可以用于删除连续元素
    System.out.println(list);
}
```

知识讲解



## 2.1.5. 【List】List 转换为数组

知识讲解

### List转换为数组

- List的toArray方法用于将集合转换为数组。但实际上该方法是在Collection中定义的，所以所有的集合都具备这个功能。
- 其有两个方法：  
`Object[] toArray()`  
`<T>T[] toArray(T[] a)`

其中第二个方法是比较常用的，我们可以传入一个指定类型的数组，该数组的元素类型应与集合的元素类型一致。返回值则是转换后的数组，该数组会保存集合中所有的元素。

+

知识讲解

### List转换为数组（续1）

```
public void testListToArray() {  
    List<String> list = new ArrayList<String>();  
    list.add("a");  
    list.add("b");  
    list.add("c");  
    String[] strArr = list.toArray(new String[] {});  
  
    // [a, b, c]  
    System.out.println(Arrays.toString(strArr));  
}
```

+

## 2.1.6. 【List】数组转换为 List

知识讲解

### 数组转换为List

- Arrays类中提供了一个静态方法asList，使用该方法我们可以将一个数组转换为对应的List集合。
- 其方法定义为：  
`static <T>List<T> asList<T... a>`
- 返回的List的集合元素类型由传入的数组的元素类型决定。
- 并且要注意的是，返回的集合我们不能对其增删元素，否则会抛出异常。并且对集合的元素进行修改会影响数组对应的元素。

+

### 数组转换为List (续1)

```

public void testArrayToList() {
    String[] strArr = { "a", "b", "c" };
    List<String> list = Arrays.asList(strArr);
    System.out.println(list); // [a, b, c]
    // list.add("d"); // 会抛出UnsupportedOperationException

    // java.util.Arrays$ArrayList
    System.out.println(list.getClass().getName());

    List<String> list1 = new ArrayList<String>();
    list1.addAll(Arrays.asList(strArr));
}

```

知识讲解



## 2.2. List 排序

### 2.2.1. 【List 排序】Collections.sort 方法实现排序

#### Collections.sort方法实现排序

- Collections是集合的工具类，它提供了很多便于我们操作集合的方法，其中就有用于集合排序的sort方法。
- 该方法定义为：
  - void sort(List<T> list)
 

该方法的作用是对给定的集合元素进行自然排序。

知识讲解



#### Collections.sort方法实现排序 (续1)

```

public void testSort() {
    List<Integer> list = new ArrayList<Integer>();
    Random r = new Random(1);
    for (int i = 0; i < 10; i++) {
        list.add(r.nextInt(100));
    }
    // [85, 88, 47, 13, 54, 4, 34, 6, 78, 48]
    System.out.println(list);

    Collections.sort(list);
    // [4, 6, 13, 34, 47, 48, 54, 78, 85, 88]
    System.out.println(list);
}

```

知识讲解



## 2.2.2. 【List 排序】 Comparable

知识讲解

+ +

### Comparable

**Comparable**

- Collections的sort方法是对集合元素进行自然排序，那么两个元素对象之间就一定要有大小之分。这个大小之分是如何界定的？实际上，在使用Collections的sort排序的集合元素都必须是Comparable接口的实现类，该接口表示其子类是可比较的，因为实现该接口必须重写抽象方法：

- int compareTo(T t);

该方法用于使当前对象与给定对象进行比较。

- 若当前对象大于给定对象，那么返回值应为 $>0$ 的整数。
- 若小于给定对象，那么返回值应为 $<0$ 的整数。
- 若两个对象相等，则应返回 $0$ 。

+ +

知识讲解

+ +

### Comparable (续1)

**Comparable** (续1)

```
public void testComparable() {
    /* Cell实现了Comparable接口， CompareTo方法逻辑为按照row值的大小排序 */
    // public int compareTo(Cell o) {return this.row - o.row; }
    List<Cell> cells = new ArrayList<Cell>();
    cells.add(new Cell(2, 3));
    cells.add(new Cell(5, 1));
    cells.add(new Cell(3, 2));
    Collections.sort(cells);
    System.out.println(cells); // [(2,3), (3,2), (5,1)]
}
```

+ +

## 2.2.3. 【List 排序】 Comparator

知识讲解

+ +

### Comparator

**Comparator**

- 一旦Java类实现了Comparable接口，其比较逻辑就已经确定；如果希望在排序的操作中临时指定比较规则，可以采用Comparator接口回调的方式。
- Comparator接口要求实现类必须重写其定义的方法：

- int compare(T o1, T o2)

- 该方法的返回值要求：

- 若 $o1 > o2$ 则返回值应 $>0$
- 若 $o1 < o2$ 则返回值应 $<0$
- 若 $o1 == o2$ 则返回值应为 $0$

+ +

### Comparator (续1)

```
public void testComparator() {
    List<Cell> cells = new ArrayList<Cell>();
    cells.add(new Cell(2, 3));
    cells.add(new Cell(5, 1));
    cells.add(new Cell(3, 2));
    // 按照col值的大小排序
    Collections.sort(cells, new Comparator<Cell>() {
        public int compare(Cell o1, Cell o2) {
            return o1.col - o2.col;
        }
    });
    System.out.println(cells); // [(5,1), (3,2), (2,3)]
}
```

知识讲解



## 2.3. 队列和栈

### 2.3.1. 【队列和栈】Queue

#### Queue



- 队列 ( Queue ) 是常用的数据结构，可以将队列看成特殊的线性表，队列限制了对线性表的访问方式：只能从线性表的一端添加 ( offer ) 元素，从另一端取出 ( poll ) 元素。
- 队列遵循先进先出 ( FIFO First Input First Output ) 的原则。
- JDK中提供了Queue接口，同时使得LinkedList实现了该接口（选择LinkedList实现Queue的原因在于Queue经常要进行添加和删除的操作，而LinkedList在这方面效率较高）。

知识讲解



#### Queue (续1)



- Queue接口中主要方法如下：

|                     |                           |
|---------------------|---------------------------|
| boolean offer(E e); | 将一个对象添加至队尾，如果添加成功则返回true。 |
| E poll();           | 从队首删除并返回一个元素。             |
| E peek();           | 返回队首的元素（但并不删除）。           |



---



---



---



---



---



---



---



---



---

## Queue ( 续2 )

```
public void testQueue() {
    Queue<String> queue = new LinkedList<String>();
    queue.offer("a");
    queue.offer("b");
    queue.offer("c");
    System.out.println(queue); // [a, b, c]
    String str = queue.peek();
    System.out.println(str); // a
    while (queue.size() > 0) {
        str = queue.poll();
        System.out.print(str + " "); // a b c
    }
}
```

知识讲解



### 2.3.2. 【队列和栈】 Deque

---



---



---



---



---



---



---



---



---

## Deque

- Deque是Queue的子接口，定义了所谓“双端队列”即从队列的两端分别可以入队（offer）和出队（poll），LinkedList实现了该接口。
- 如果将Deque限制为只能从一端入队和出队，则可实现“栈”（Stack）的数据结构，对于栈而言，入栈称之为push，出栈称之为pop。
- 栈遵循先进后出（FILO First Input Last Output）的原则。




---



---



---



---



---



---



---



---



---

## Deque ( 续1 )

```
public void testStack() {
    Deque<String> stack = new LinkedList<String>();
    stack.push("a");
    stack.push("b");
    stack.push("c");
    System.out.println(stack); // [c, b, a]
    String str = stack.peek();
    System.out.println(str); // c
    while (stack.size() > 0) {
        str = stack.pop();
        System.out.print(str + " "); // c b a
    }
}
```

知识讲解



## 经典案例

### 1. 测试方法 `addAll`、`containsAll` 的用法

- 问题

测试 `Collection` 接口中 `addAll` 方法、`containsAll` 方法的用法，详细要求如下：

- 1 ) 使用 `ArrayList` 构建集合 `c1`，将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 `c1` 中，并输出集合 `c1`。
- 2 ) 使用 `HashSet` 构建集合 `c2`，将集合 `c1` 中的元素添加到集合 `c2`，并输出集合 `c2`。
- 3 ) 使用 `ArrayList` 构建集合 `c3`，将字符串 “java”、“cpp” 放入集合 `c3` 中，并判断集合 `c1` 中是否包含集合 `c3`。

- 方案

- 1 ) 首先，使用 `ArrayList` 构建集合 `c1`；然后，使用 `Collection` 接口提供的 `add` 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 `c1` 中；最后，输出集合 `c1`，代码如下所示：

```
Collection<String> c1 = new ArrayList<String>();
c1.add("java");
c1.add("cpp");
c1.add("php");
c1.add("c#");
c1.add("objective-c");

System.out.println(c1); // [java, cpp, php, c#, objective-c]
```

- 2 ) 首先，使用 `HashSet` 构建集合 `c2`；然后，使用 `Collection` 接口提供的 `addAll` 方法将集合 `c1` 中的元素添加到集合 `c2` 中，代码如下所示：

```
Collection<String> c2 = new HashSet<String>();

c2.addAll(c1);
System.out.println(c2); // [cpp, php, c#, java, objective-c]
```

- 3 ) 首先，使用 `ArrayList` 构建集合 `c3`；然后，使用 `Collection` 接口提供的 `add` 方法将字符串 “java”、“cpp” 放入集合 `c3` 中；最后，使用 `Collection` 接口提供的 `containsAll` 方法判断集合 `c1` 中是否包含集合 `c3` 中的元素，代码如下所示：

```
Collection<String> c3 = new ArrayList<String>();
c3.add("java");
c3.add("cpp");
System.out.println(c1.containsAll(c3)); // true
```

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：新建类及测试方法

首先，在名为 JavaSE 的工程下的 src 下新建名为 day04 的包，然后，在该包下新建名为 TestCollection 的类，并在类中新建测试方法 testAddAllAndContainsAll，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {

    /**
     * 测试 addAll 方法和 containsAll 方法
     */
    @Test
    public void testAddAllAndContainsAll() {
    }

}
```

### 步骤二：构建集合 c1

首先，使用 ArrayList 构建集合 c1；然后，使用 Collection 接口提供的 add 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 c1 中；最后，输出集合 c1，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 addAll 方法和 containsAll 方法
     */
    @Test
    public void testAddAllAndContainsAll() {

        Collection<String> c1 = new ArrayList<String>();
        c1.add("java");
        c1.add("cpp");
        c1.add("php");
        c1.add("c#");
        c1.add("objective-c");

        System.out.println(c1); // [java, cpp, php, c#, objective-c]
    }
}
```

运行 `testAddAllAndContainsAll` 方法，控制台输出结果如下所示：

```
[java, cpp, php, c#, objective-c]
```

从输出结果可以看出，已经将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放到集合 `c1` 中。

### 步骤三：构建结合 c2

首先，使用 `HashSet` 构建集合 `c2`；然后，使用 `Collection` 接口提供的 `addAll` 方法将集合 `c1` 中的元素添加到集合 `c2` 中，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 addAll 方法和 containsAll 方法
     */
    @Test
    public void testAddAllAndContainsAll() {
        Collection<String> c1 = new ArrayList<String>();
        c1.add("java");
        c1.add("cpp");
        c1.add("php");
        c1.add("c#");
        c1.add("objective-c");

        System.out.println(c1); // [java, cpp, php, c#, objective-c]

        Collection<String> c2 = new HashSet<String>();

        c2.addAll(c1);
        System.out.println(c2); // [cpp, php, c#, java, objective-c]
    }
}
```

### 步骤四：运行

运行 `testAddAllAndContainsAll` 方法，运行结果如下所示：

```
[java, cpp, php, c#, objective-c]
```

```
[cpp, php, c#, java, objective-c]
```

观察输出结果，已经将集合 `c1` 中的元素添加到 `c2` 中。

### 步骤五：构建集合 c3，使用 `containsAll` 判断集合 c1 是否包含集合 c3

首先，使用 `ArrayList` 构建集合 `c3`；然后，使用 `Collection` 接口提供的 `add` 方法将字符串 “java”、“cpp” 放入集合 `c3` 中；最后，使用 `Collection` 接口提供的 `containsAll`

方法判断集合 c1 中是否包含集合 c3 中的元素，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;

public class TestCollection {
    /**
     * 测试 addAll 方法和 containsAll 方法
     */
    @Test
    public void testAddAllAndContainsAll() {
        Collection<String> c1 = new ArrayList<String>();
        c1.add("java");
        c1.add("cpp");
        c1.add("php");
        c1.add("c#");
        c1.add("objective-c");

        System.out.println(c1); // [java, cpp, php, c#, objective-c]

        Collection<String> c2 = new HashSet<String>();

        c2.addAll(c1);
        System.out.println(c2); // [cpp, php, c#, java, objective-c]

        Collection<String> c3 = new ArrayList<String>();
        c3.add("java");
        c3.add("cpp");
        System.out.println(c1.containsAll(c3)); // true

    }
}
```

## 步骤六：运行

运行 testAddAllAndContainsAll 方法，运行结果如下所示：

```
[java, cpp, php, c#, objective-c]
[cpp, php, c#, java, objective-c]

true
```

观察输出结果，可以看出集合 c1 是包含集合 c3 的。

### • 完整代码

本案例的完整代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import org.junit.Test;
```

```

public class TestCollection {
    /**
     * 测试 addAll 方法和 containsAll 方法
     */
    @Test
    public void testAddAllAndContainsAll() {
        Collection<String> c1 = new ArrayList<String>();
        c1.add("java");
        c1.add("cpp");
        c1.add("php");
        c1.add("c#");
        c1.add("objective-c");

        System.out.println(c1); // [java, cpp, php, c#, objective-c]

        Collection<String> c2 = new HashSet<String>();

        c2.addAll(c1);
        System.out.println(c2); // [cpp, php, c#, java, objective-c]

        Collection<String> c3 = new ArrayList<String>();
        c3.add("java");
        c3.add("cpp");
        System.out.println(c1.containsAll(c3)); // true
    }
}

```

## 2. 使用 Iterator 的 hasNext 方法、next 方法遍历集合

- 问题

使用 Iterator 的 hasNext 方法、next 方法遍历集合，详细要求如下：

- 1) 使用 HashSet 构建集合 c 将字符串“java”、“cpp”、“php”、“c#”、“objective-c”放入集合 c 中。
- 2) 遍历集合中的每一个元素，并输出。

- 方案

- 1) 首先，使用 HashSet 构建集合 c；然后，使用 Collection 接口提供的 add 方法将字符串“java”、“cpp”、“php”、“c#”、“objective-c”放入集合 c 中，代码如下所示：

```

Collection<String> c = new ArrayList<String>();
c.add("java");
c.add("cpp");
c.add("php");
c.add("c#");
c.add("objective-c");

```

- 2) 使用 Collection 接口的 iterator 方法，获取迭代器接口 Iterator，代码如下所示：

```
Iterator<String> it = c.iterator();
```

3 ) 首先 , 使用 `Iterator` 接口提供的 `hasNext` 方法 , 循环判断集合中是否还有未迭代的元素 ; 然后 , 在循环中 , 使用 `Iterator` 接口提供的 `next` 方法返回当前指针位置的元素 , 并将指针后移 , 代码如下所示 :

```
Iterator<String> it = c.iterator();
while (it.hasNext()) {
    String str = it.next();
    System.out.println(str);
}
```

### • 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：构建测试方法及构建集合对象 c

首先 , 新建类 `TestIterator` , 在该类中新建测试方法 `testHasNextAndNext` ; 然后 , 使用 `HashSet` 构建集合 `c` ; 最后 , 使用 `Collection` 接口提供的 `add` 方法将字符串 “java” 、 “cpp” 、 “php” 、 “c#” 、 “objective-c” 放入集合 `c` 中 , 代码如下所示 :

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {
    /**
     * 测试 Iterator 的 hasNext 方法和 next 方法
     */
    @Test
    public void testHasNextAndNext() {
        Collection<String> c = new HashSet<String>();
        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");
    }
}
```

#### 步骤二：循环遍历集合 c 中的元素

首先 , 使用 `Iterator` 接口提供的 `hasNext` 方法 , 循环判断集合中是否还有未迭代的元素 ; 然后 , 在循环中 , 使用 `Iterator` 接口提供的 `next` 方法返回当前指针位置的元素 , 并将指针后移 , 代码如下所示 :

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {
    /**

```

```
* 测试 Iterator 的 hasNext 方法和 next 方法
*/
@Test
public void testHasNextAndNext() {
    Collection<String> c = new HashSet<String>();
    c.add("java");
    c.add("cpp");
    c.add("php");
    c.add("c#");
    c.add("objective-c");

    Iterator<String> it = c.iterator();
    while (it.hasNext()) {
        String str = it.next();
        System.out.println(str);
    }

}
```

### 步骤三：运行

运行 testHasNextAndNext 方法，控制台输出结果如下：

```
cpp
php
c#
java
objective-c
```

查看输出结果，可以看出输出了集合中的每一个元素。

#### • 完整代码

本案例中，类 TestIterator 的完整代码如下所示：

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {
    /**
     * 测试 Iterator 的 hasNext 方法和 next 方法
     */
    @Test
    public void testHasNextAndNext() {
        Collection<String> c = new HashSet<String>();
        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");

        Iterator<String> it = c.iterator();
        while (it.hasNext()) {
            String str = it.next();
```

```
        System.out.println(str);
    }
}
```

### 3. 使用 Iterator 的 remove 方法移除元素

- 问题

使用 Iterator 的 remove 方法移除集合中的元素，详细要求如下：

- 1 ) 使用 HashSet 构建集合 c ,将字符串“java”、“cpp”、“php”、“c#”、“objective-c”放入集合 c 中。
- 2 ) 删 除 集 合 c 中 包 含 字 符 “c” 的 字 符 串 元 素。

- 方案

- 1 ) 首先，使用 HashSet 构建集合 c ；然后，使用 Collection 接口提供的 add 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 c 中，代码如下所示：

```
Collection<String> c = new HashSet<String>();
c.add("java");
c.add("cpp");
c.add("php");
c.add("c#");
c.add("objective-c");
```

- 2 ) 使用 Collection 接口的 iterator 方法，获取迭代器接口 Iterator ，代码如下所示：

```
Iterator<String> it = c.iterator();
```

- 3 ) 首先，使用 Iterator 接口提供的 hasNext 方法和 next 方法遍历到集合 c 中的每一个元素；然后，在循环中，使用 String 类的 indexOf 方法判断遍历到的字符串元素是否包含字符 “c” ，当 indexOf 方法返回非 -1 的其它值时，说明该字符串中包含字符 “c” ；最后，使用 Iterator 接口提供的 remove 方法将该字符串元素移除，代码如下所示：

```
Iterator<String> it = c.iterator();
while (it.hasNext()) {
    String str = it.next();
    if (str.indexOf('c') != -1) {
        it.remove();
    }
}
```

- 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：创建测试方法

首先，在 TestIterator 类中新建测试方法 testRemove；然后，使用 HashSet 构建集合 c；最后，使用 Collection 接口提供 add 方法将字符串“java”、“cpp”、“php”、“c#”、“objective-c”放入集合 c 中，代码如下所示：

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {

    /**
     * 测试 Iterator 的 remove 方法
     */
    @Test
    public void testRemove() {
        Collection<String> c = new HashSet<String>();
        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");

        System.out.println(c); // [cpp, php, c#, java, objective-c]
    }

}
```

## 步骤二：运行

运行 testRemove 方法，此时输出集合 c 中的元素如下：

```
[cpp, php, c#, java, objective-c]
```

## 步骤三：使用 Iterator 的 remove 方法移除集合中的元素

首先，使用 Iterator 接口提供的 hasNext 方法和 next 方法遍历到集合 c 中的每一个元素；然后，在循环中，使用 String 类的 indexOf 方法判断遍历到的字符串元素是否包含字符“c”，当 indexOf 方法返回非-1 的其它值时，说明该字符串中包含字符“c”；最后，使用 Iterator 接口提供的 remove 方法将该字符串元素移除，代码如下所示：

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {

    /**
     * 测试 Iterator 的 remove 方法
     */
    @Test
    public void testRemove() {
```

```

Collection<String> c = new HashSet<String>();
c.add("java");
c.add("cpp");
c.add("php");
c.add("c#");
c.add("objective-c");

System.out.println(c); // [cpp, php, c#, java, objective-c]

Iterator<String> it = c.iterator();
while (it.hasNext()) {
    String str = it.next();
    if (str.indexOf('c') != -1) {
        it.remove();
    }
}
System.out.println(c); // [php, java]

}

```

上述代码中使用了 `Iterator` 的 `remove` 方法，该方法使用时，每次 `next` 方法调用完之后只能调用一次该方法，该方法删除刚刚 `next` 方法返回的元素。在迭代的过程中不能调用集合的 `remove` 等方法删除元素，否则会抛出异常 `ConcurrentModificationException`。

#### 步骤四：运行

运行 `testRemove` 方法，控制台输出结果如下所示：

```
[cpp, php, c#, java, objective-c]
[php, java]
```

从输出结果可以看出，第二次输出集合 `c` 时，集合 `c` 中包含字符 “c” 的元素已经都删除掉。

- **完整代码**

本案例中，类 `TestIterator` 的完整代码如下所示：

```

package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {
//... (之前案例的代码, 略)

/**
 * 测试 Iterator 的 remove 方法
 */
@Test
public void testRemove() {
    Collection<String> c = new HashSet<String>();
    c.add("java");
    c.add("cpp");
}

```

```

        c.add("php");
        c.add("c#");
        c.add("objective-c");

        System.out.println(c); // [cpp, php, c#, java, objective-c]

        Iterator<String> it = c.iterator();
        while (it.hasNext()) {
            String str = it.next();
            if (str.indexOf('c') != -1) {
                it.remove();
            }
        }
        System.out.println(c); // [php, java]
    }
}

```

## 4. 使用增强型 for 循环遍历集合

- 问题

使用 `foreach` 循环遍历集合中的元素，详细要求如下：

- 1 ) 使用 `HashSet` 构建集合 `c`，将字符串“java”、“cpp”、“php”、“c#”、“objective-c”放入集合 `c` 中。
- 2 ) 使用 `foreach` 循环遍历集合中的每一个元素，并将每一个字符串元素转化为大写形式。

- 方案

- 1 ) 首先，使用 `HashSet` 构建集合 `c`；然后，使用 `Collection` 接口提供 `add` 方法将字符串“java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 `c` 中，代码如下所示：

```

Collection<String> c = new HashSet<String>();
c.add("java");
c.add("cpp");
c.add("php");
c.add("c#");
c.add("objective-c");

```

- 2 ) 然后，使用 `foreach` 循环遍历集合中的每一个元素，并调用 `String` 类的 `toUpperCase` 方法将遍历到的字符串元素转化为大写形式，代码如下所示：

```

for (String str : c) {
    System.out.print(str.toUpperCase() + " ");
}

```

以上代码可以理解为每次从集合 `c` 中取出一个 `String` 对象，然后赋值给循环变量 `str`。事实上，Java 编译器在编译前会将其转换为迭代器的形式（因此不能在循环体中对集合进行删除操作）。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：添加测试方法

首先，在类 TestIterator 中新建测试方法 testForeach；然后，使用 HashSet 构建集合 c；最后，使用 Collection 接口提供的 add 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 c 中，代码如下所示：

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {

    /**
     * 测试 foreach 循环
     */
    @Test
    public void testForeach() {
        Collection<String> c = new HashSet<String>();
        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");
    }

}
```

### 步骤二：使用 foreach 循环遍历集合中的元素

使用 foreach 循环遍历集合中的每一个元素，并调用 String 类的 toUpperCase 方法将遍历到的字符串元素转化为大写后输出，代码如下所示：

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {
    /**
     * 测试 foreach 循环
     */
    @Test
    public void testForeach() {
        Collection<String> c = new HashSet<String>();
        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");

        for (String str : c) {
            System.out.print(str.toUpperCase() + " ");
        }
    }
}
```

```
}
```

```
}
```

### 步骤三：运行

运行 `testForEach` 方法，控制台输出结果为：

```
CPP PHP C# JAVA OBJECTIVE-C
```

从输出结果可以看出已经遍历到了集合中的每一个元素，并将每一个元素转换为大写形式。

- **完整代码**

本案例中，类 `TestIterator` 的完整代码如下所示：

```
package day04;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.junit.Test;

public class TestIterator {
    //... (之前案例的代码，略)

    /**
     * 测试 foreach 循环
     */
    @Test
    public void testForEach() {
        Collection<String> c = new HashSet<String>();
        c.add("java");
        c.add("cpp");
        c.add("php");
        c.add("c#");
        c.add("objective-c");
        for (String str : c) {
            System.out.print(str.toUpperCase() + " ");
        }
        // CPP PHP C# JAVA OBJECTIVE-C
    }
}
```

## 5. 测试 List 的 get 方法和 set 方法

- **问题**

使用 `List` 的 `set` 方法和 `get` 方法设置和获取集合中的元素，详细要求如下：

- 1 ) 创建 `List` 接口的引用 `list`，使该引用指向 `ArrayList` 的实例。
- 2 ) 将字符串 “`java`”、“`cpp`”、“`php`”、“`c#`”、“`objective-c`” 放入集合 `list` 中。

3 ) 使用 List 接口提供的 get 方法和 size 方法遍历集合 list , 并将每一个字符串元素转换成大写。

4 ) 将索引位置为 1 的元素替换为 c++ , 并输出被替换掉的元素以及集合 list 。

5 ) 交换集合 list 中索引位置为 1 和索引位置为 3 的元素。

### • 方案

1 ) 首先 , 创建 List 接口的引用 list , 使该引用指向 ArrayList 的实例 , 并使用 add 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 list 中 , 代码如下所示 :

```
List<String> list = new ArrayList<String>();
list.add("java");
list.add("cpp");
list.add("php");
list.add("c#");
list.add("objective-c");
```

2 ) 然后 , 使用 List 接口提供的 get 方法、 size 方法配合循环来遍历集合 list , 代码如下所示 :

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i).toUpperCase());
}
```

上述代码中使用了 get 方法 , 该方法将根据参数索引位置返回对应的对象元素。

3 ) 使用 List 接口提供的 set 方法将索引位置为 1 的元素替换为 c++ , 并输出被替换掉的元素以及集合 list , 代码如下所示 :

```
String value = list.set(1, "c++");
System.out.println(value); // cpp
System.out.println(list); // [java, c++, php, c#, objective-c]
```

上述代码中使用到 set 方法 , 该方法将指定索引位置设置为指定的元素 , 其返回值为未设置前此位置的元素。

4 ) 使用 List 接口提供的 set 方法交换集合 list 中索引位置为 1 和索引位置为 3 的元素 , 代码如下所示 :

```
list.set(1, list.set(3, list.get(1)));
System.out.println(list); // [java, c#, php, c++, objective-c]
```

set 方法将指定索引位置设置为指定的元素 , 其返回值为未设置前此位置的元素。鉴于 set 方法这样的设计 , 可以使用此语句实现将 list 中第 i 个和第 j 个元素交换的功能 :

```
list.set(i, list.set(j, list.get(i)));
```

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：创建测试方法并构建集合 list

首先，新建类 TestList，在该类中新建测试方法 testGetAndSet；然后，创建 List 接口的引用 list，使该引用指向 ArrayList 的实例，并使用 add 方法将字符串 “java”、“cpp”、“php”、“c#”、“objective-c” 放入集合 list 中，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 set 方法和 get 方法
     */
    @Test
    public void testGetAndSet() {
        List<String> list = new ArrayList<String>();
        list.add("java");
        list.add("cpp");
        list.add("php");
        list.add("c#");
        list.add("objective-c");
    }
}
```

### 步骤二：使用 get 方法遍历集合 list

使用 List 接口提供的 get 方法、size 方法配合循环来遍历集合 list，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 set 方法和 get 方法
     */
    @Test
    public void testGetAndSet() {
        List<String> list = new ArrayList<String>();
        list.add("java");
        list.add("cpp");
        list.add("php");
        list.add("c#");
        list.add("objective-c");

        // get 方法遍历 List
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i).toUpperCase());
        }
    }
}
```

```
    }  
  
}  
}
```

### 步骤三：运行

运行 testGetAndSet 方法，控制台输出结果如下所示：

JAVA  
CPP  
PHP  
C#  
OBJECTIVE-C

从输出结果可以看出，遍历到了每一个元素，并转换为大写。

### 步骤四：使用 set 方法设置集合元素

使用 List 接口提供的 set 方法将索引位置为 1 的元素替换为 c++，并输出被替换掉的元素以及集合 list，代码如下所示：

```
package day04;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import org.junit.Test;  
  
public class TestList {  
    /**  
     * 测试 set 方法和 get 方法  
     */  
    @Test  
    public void testGetAndSet() {  
        List<String> list = new ArrayList<String>();  
        list.add("java");  
        list.add("cpp");  
        list.add("php");  
        list.add("c#");  
        list.add("objective-c");  
  
        // get 方法遍历 List  
        for (int i = 0; i < list.size(); i++) {  
            System.out.println(list.get(i).toUpperCase());  
        }  
  
        String value = list.set(1, "c++");  
        System.out.println(value); // cpp  
        System.out.println(list); // [java, c++, php, c#, objective-c]  
    }  
}
```

### 步骤五：运行

运行 testGetAndSet 方法，控制台输出结果如下所示：

JAVA  
CPP  
PHP  
C#  
OBJECTIVE-C

cpp  
[java, c++, php, c#, objective-c]

从输出结果可以看出，将索引位置为 1 的元素替换为 c++，并返回了设置前索引位置为 1 的元素 cpp。

### 步骤六：使用 set 方法交换 list 结合中的两个元素

使用 List 接口提供的 set 方法交换集合 list 中索引位置为 1 和索引位置为 3 的元素，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 set 方法和 get 方法
     */
    @Test
    public void testGetAndSet() {
        List<String> list = new ArrayList<String>();
        list.add("java");
        list.add("cpp");
        list.add("php");
        list.add("c#");
        list.add("objective-c");

        // get 方法遍历 List
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i).toUpperCase());
        }

        String value = list.set(1, "c++");
        System.out.println(value); // cpp
        System.out.println(list); // [java, c#, php, c++, objective-c]

        // 交换位置 1 和 3 上的元素
        list.set(1, list.set(3, list.get(1)));
        System.out.println(list); // [java, c#, php, c++, objective-c]
    }
}
```

### 步骤七：运行

运行 testGetAndSet 方法，控制台输出结果如下所示：

JAVA

CPP  
PHP  
C#  
OBJECTIVE-C  
cpp  
[java, c++, php, c#, objective-c]

[java, c#, php, c++, objective-c]

对比两次集合中的输出结果，可以看出已经交换了集合 list 中索引位置为 1 和索引位置为 3 的元素。

- **完整代码**

本案例中，类 TestList 的完整代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 set 方法和 get 方法
     */
    @Test
    public void testGetAndSet() {
        List<String> list = new ArrayList<String>();
        list.add("java");
        list.add("cpp");
        list.add("php");
        list.add("c#");
        list.add("objective-c");

        // get 方法遍历 List
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i).toUpperCase());
        }

        String value = list.set(1, "c++");
        System.out.println(value); // cpp
        System.out.println(list); // [java, c++, php, c#, objective-c]

        // 交换位置 1 和 3 上的元素
        list.set(1, list.set(3, list.get(1)));
        System.out.println(list); // [java, c#, php, c++, objective-c]
    }
}
```

## 6. 测试向 List 中插入和删除元素

- **问题**

向 List 集合的指定索引位置插入元素以及移除指定索引位置的元素，详细要求如下：

- 1 ) 创建 List 接口的引用 list，使该引用指向 ArrayList 的实例。将字符串“java”、“c#”放入集合 list 中。

2) 在索引位置为 1 处，插入字符串元素 “cpp”。

3) 移除索引位置为 2 的元素。

### • 方案

首先，创建 List 接口的引用 list，使该引用指向 ArrayList 的实例，并使用 add 方法将字符串 “java”、“c#” 放入集合 list 中，代码如下所示：

```
List<String> list = new ArrayList<String>();
list.add("java");
list.add("c#");
System.out.println(list); // [java, c#]
```

然后，使用 List 接口的带有索引位置参数的 add 方法，在索引位置为 1 处，插入字符串元素 “cpp”，代码如下所示：

```
list.add(1, "cpp");
System.out.println(list); // [java, cpp, c#]
```

List 重载了带有索引位置参数的 add 方法，该方法将对象插入集合的指定索引位置( 相当于前插，可以理解为对象插入集合后其索引为指定的参数 )。

最后，使用 List 重载的带有索引位置参数的 remove 方法，移除 list 集合中索引位置为 2 的元素，代码如下所示：

```
list.remove(2);
System.out.println(list); // [java, cpp]
```

List 也重载了索引位置为参数的 remove 方法，该方法可以删掉指定索引位置的对象元素，其返回值为刚刚删掉的对象引用。

### • 步骤

#### 步骤一：构建测试方法

首先，在 TestList 中新建测试方法 testInsertAndRemove；然后，创建 List 接口的引用 list，使该引用指向 ArrayList 的实例，并使用 add 方法将字符串 “java”、“c#” 放入集合 list 中，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {

    /**
     * 测试插入和移除元素
}
```

```
/*
@Test
public void testInsertAndRemove() {
    List<String> list = new ArrayList<String>();
    list.add("java");
    list.add("c#");
    System.out.println(list); // [java, c#]
}

}
```

## 步骤二：使用 List 重载的 add 方法向集合中插入元素

使用 List 重载的带有索引位置参数的 add 方法，在索引位置为 1 处，插入字符串元素“cpp”，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试插入和移除元素
     */
    @Test
    public void testInsertAndRemove() {
        List<String> list = new ArrayList<String>();
        list.add("java");
        list.add("c#");
        System.out.println(list); // [java, c#]

        list.add(1, "cpp");
        System.out.println(list); // [java, cpp, c#]

    }
}
```

## 步骤三：运行

运行 testInsertAndRemove 方法，控制台输出结果如下所示：

```
[java, c#]
[java, cpp, c#]
```

从输出结果可以看出，已经将“cpp”插入到集合 list 中索引位置为 1 处。

## 步骤四：使用 List 重载的 remove 方法从集合中删除元素

使用 List 重载的带有索引位置参数的 remove 方法，移除 list 集合中索引位置为 2 的元素，代码如下所示：

```
package day04;
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试插入和移除元素
     */
    @Test
    public void testInsertAndRemove() {
        List<String> list = new ArrayList<String>();
        list.add("java");
        list.add("c#");
        System.out.println(list); // [java, c#]

        list.add(1, "cpp");
        System.out.println(list); // [java, cpp, c#]

        list.remove(2);
        System.out.println(list); // [java, cpp]
    }
}
```

## 步骤五：运行

运行 `testInsertAndRemove` 方法，控制台输出结果如下所示：

```
[java, c#]
[java, cpp, c#]

[java, cpp]
```

从输出结果可以看出，已经将索引位置为 2 的元素 “c#” 移除掉。

### • 完整代码

本案例的完整代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    //... (之前案例的代码，略)

    /**
     * 测试插入和移除元素
     */
    @Test
    public void testInsertAndRemove() {
        List<String> list = new ArrayList<String>();
        list.add("java");
        list.add("c#");
        System.out.println(list); // [java, c#]
```

```
list.add(1, "cpp");
System.out.println(list); // [java, cpp, c#]

list.remove(2);
System.out.println(list); // [java, cpp]
}
```

## 7. 测试 List 的 subList 方法

- 问题

使用 List 接口提供的 subList 方法获取子 List，详细要求如下：

- 1) 创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；将 0~9 十个数字作为十个元素放入到集合 list 中，并输出集合 list。
- 2) 获取 list 集合的子集合 subList，subList 子集合的元素为[3, 4, 5, 6, 7]并输出子集合 subList。
- 3) 将 subList 集合中的每一个元素扩大 10 倍，输出 list 集合和 subList 集合，验证 subList 获得的 List 集合和源 List 集合占用相同的数据空间。
- 4) 清除 list 集合中索引位置为 3~7 ( 包含 3 和 7 ) 的元素，并输出 list。

- 方案

- 1) 创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；将 0~9 十个数字作为十个元素放入到集合 list 中，并输出集合 list，代码如下所示：

```
List<Integer> list = new ArrayList<Integer>();
for (int i = 0; i < 10; i++) {
    list.add(i);
}
```

- 2) 使用 List 接口的 subList 方法，获取 list 集合的子集合 subList，subList 集合的元素为[3, 4, 5, 6, 7]并输出集合 subList，代码如下所示：

```
List<Integer> subList = list.subList(3, 8);
System.out.println(subList); // [3, 4, 5, 6, 7]
```

subList 方法的声明如下：

```
List<E> subList(int fromIndex, int toIndex);
```

该方法用于获取子 List；该方法中 fromIndex 和 toIndex 是截取子 List 的首尾下标（前包括，后不包括）。

- 3) 将 subList 集合中的每一个元素扩大 10 倍，输出 list 集合和 subList 结合，验证 subList 获得的 List 和源 List 占有相同的数据空间，代码如下所示：

```
// subList 获得的 List 和源 List 占有相同的数据空间
for (int i = 0; i < subList.size(); i++) {
    subList.set(i, subList.get(i) * 10);
}
System.out.println(subList); // [30, 40, 50, 60, 70]
System.out.println(list); // [0, 1, 2, 30, 40, 50, 60, 70, 8, 9]
```

List 接口提供的 subList 方法用于获取子 List。需要注意的是，subList 获取的子 List 与源 List 占有相同的存储空间，对子 List 的操作会影响到源 List。

4 ) 使用 List 接口提供的 clear 方法清除 list 集合中索引位置为 3 ~ 7 ( 包含 3 和 7 ) 的元素，并输出 list，代码如下所示：

```
// 可以用于删除连续元素
list.subList(3, 8).clear();
System.out.println(list);
```

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先 在 TestList 中新建测试方法 testSubList 然后 创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；将 0~9 十个数字作为十个元素放入到集合 list 中，并输出集合 list，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {

    /**
     * 测试 subList 方法
     */
    @Test
    public void testSubList() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add(i);
        }
        System.out.println(list); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    }
}
```

### 步骤二：使用 List 的 subList 方法获取子 List

使用 List 接口的 subList 方法，获取 list 集合的子集合 subList，subList 子集合的元素为 [3, 4, 5, 6, 7]，并输出集合 subList，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 subList 方法
     */
    @Test
    public void testSubList() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add(i);
        }
        System.out.println(list); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        List<Integer> subList = list.subList(3, 8);
        System.out.println(subList); // [3, 4, 5, 6, 7]
    }
}
```

### 步骤三：运行

运行 testSubList 方法，控制台输出结果如下：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7]
```

可以看出获取到了子 List，其元素为[3, 4, 5, 6, 7]。

### 步骤四：验证 subList 获得的子 List 和源 List 占有相同的数据空间

将 subList 集合中的每一个元素扩大 10 倍，输出 list 集合和 subList 结合，验证 subList 获得的 List 和源 List 占有相同的数据空间，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 subList 方法
     */
    @Test
    public void testSubList() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add(i);
        }
        System.out.println(list); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        List<Integer> subList = list.subList(3, 8);
        for (int i = 0; i < subList.size(); i++) {
            subList.get(i) *= 10;
        }
        System.out.println(subList);
    }
}
```

```
System.out.println(subList); // [3, 4, 5, 6, 7]
```

```
// subList 获得的 List 和源 List 占有相同的数据空间
for (int i = 0; i < subList.size(); i++) {
    subList.set(i, subList.get(i) * 10);
}
System.out.println(subList); // [30, 40, 50, 60, 70]
System.out.println(list); // [0, 1, 2, 30, 40, 50, 60, 70, 8, 9]

}
```

## 步骤五：运行

运行 testSubList 方法，控制台输出结果如下所示：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7]
```

```
[30, 40, 50, 60, 70]
[0, 1, 2, 30, 40, 50, 60, 70, 8, 9]
```

从运行结果可以看出，subList 中每一个元素扩大了 10 倍，list 中的索引位置 3 ~ 7 处的元素也扩大了 10 倍，因此，subList 获取的子 List 与源 List 占有相同的存储空间，对子 List 的操作会影响到源 List。

## 步骤六：使用 List 的 clear 方法清除 List 中的某一段元素

使用 List 接口提供的 clear 方法清除 list 集合中索引位置为 3 ~ 7（包含 3 和 7）的元素，并输出 list，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 subList 方法
     */
    @Test
    public void testSubList() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add(i);
        }
        System.out.println(list); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        List<Integer> subList = list.subList(3, 8);
        System.out.println(subList); // [3, 4, 5, 6, 7]

        // subList 获得的 List 和源 List 占有相同的数据空间
        for (int i = 0; i < subList.size(); i++) {
            subList.set(i, subList.get(i) * 10);
        }
        System.out.println(subList); // [30, 40, 50, 60, 70]
```

```
System.out.println(list); // [0, 1, 2, 30, 40, 50, 60, 70, 8, 9]
```

```
// 可以用于删除连续元素
list.subList(3, 8).clear();
System.out.println(list);

}
```

## 步骤七：运行

运行 testSubList 方法，控制台输出结果如下所示：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7]
[30, 40, 50, 60, 70]
[0, 1, 2, 30, 40, 50, 60, 70, 8, 9]

[0, 1, 2, 8, 9]
```

从运行结果可以看出，已经将 list 集合中索引位置为 3 ~ 7 的元素清除掉。

### • 完整代码

本案例中，类 TestList 的完整代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    //... (之前案例的代码, 略)

    /**
     * 测试 subList 方法
     */
    @Test
    public void testSubList() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add(i);
        }
        System.out.println(list); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        List<Integer> subList = list.subList(3, 8);
        System.out.println(subList); // [3, 4, 5, 6, 7]

        // subList 获得的 List 和源 List 占有相同的数据空间
        for (int i = 0; i < subList.size(); i++) {
            subList.set(i, subList.get(i) * 10);
        }
        System.out.println(subList); // [30, 40, 50, 60, 70]
        System.out.println(list); // [0, 1, 2, 30, 40, 50, 60, 70, 8, 9]

        // 可以用于删除连续元素
        list.subList(3, 8).clear();
    }
}
```

```
        System.out.println(list);
    }
}
```

## 8. 将 List 转换为数组

- 问题

将集合 List 转换为数组，详细要求如下所示：

- 1) 创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；将字符串 “a”、“b”、“c” 放入集合 list 中。
- 2) 将集合 list 转换为数组，并输出数组中的元素。

- 方案

首先，创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；将字符串 “a”、“b”、“c” 放入集合 list 中，代码如下所示：

```
List<String> list = new ArrayList<String>();
list.add("a");
list.add("b");
list.add("c");
```

然后，使用 List 接口提供的 toArray 方法将集合 list 转换为数组，代码如下所示：

```
String[] strArr = list.toArray(new String[] {});
System.out.println(Arrays.toString(strArr)); // [a, b, c]
```

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先，在 TestList 中新建测试方法 testListToArray；然后，创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；将字符串 “a”、“b”、“c” 放入集合 list 中，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {

    /**
     * 测试 toArray 方法
     */
    @Test
```

```
public void testListToArray() {  
    List<String> list = new ArrayList<String>();  
    list.add("a");  
    list.add("b");  
    list.add("c");  
}  
}
```

## 步骤二：使用 toArray 方法将 list 集合转换为数组

使用 List 接口提供的 toArray 方法将集合 list 转换为数组，代码如下所示：

```
package day04;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import org.junit.Test;  
  
public class TestList {  
    /**  
     * 测试 toArray 方法  
     */  
    @Test  
    public void testListToArray() {  
        List<String> list = new ArrayList<String>();  
        list.add("a");  
        list.add("b");  
        list.add("c");  
  
        String[] strArr = list.toArray(new String[] {});  
        System.out.println(Arrays.toString(strArr)); // [a, b, c]  
    }  
}
```

## 步骤三：运行

运行 testListToArray 方法，控制台输出结果如下所示：

```
[a, b, c]
```

从运行结果看，已经将 list 转换为数组了，输出的结果为数组中的三个元素。

### • 完整代码

本案例中，类 TestList 的完整代码如下所示：

```
package day04;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import org.junit.Test;  
  
public class TestList {
```

```
//... (之前案例的代码, 略)

/**
 * 测试 toArray 方法
 */
@Test
public void testListToArray() {
    List<String> list = new ArrayList<String>();
    list.add("a");
    list.add("b");
    list.add("c");

    String[] strArr = list.toArray(new String[] {});
    System.out.println(Arrays.toString(strArr)); // [a, b, c]
}
```

## 9. 将数组转换为 List

### • 问题

将数组转换为 List 集合，详细要求如下所示：

- 1 ) 构建字符串数组 strArr，数组元素为 “a”、“b”、“c”。
- 2 ) 将数组 strArr 转换为 List 集合变量 list，并输出集合 list；然后向 list 集合中添加元素 “d”，运行后查看输出结果。
- 3 ) 获取变量 list 所属的类型的名称。
- 4 ) 构建集合 list1 使用 List 的 addAll 方法将数组转换来的集合添加到 list1 中。

### • 方案

- 1 ) 首先，构建字符串数组 strArr，数组元素为 “a”、“b”、“c”；然后，使用工具类 Arrays 提供的方法 asList 将数组 strArr 转换为 List 集合变量 list，并输出集合 list，代码如下所示：

```
String[] strArr = { "a", "b", "c" };
List<String> list = Arrays.asList(strArr);
System.out.println(list); // [a, b, c]
```

- 2 ) 向 list 集合中添加元素 “d”；运行后，将该代码注释，然后获取变量 list 所属的类型的名称并输出，代码如下所示：

```
// list.add("d"); // 会抛出 UnsupportedOperationException
System.out.println(list.getClass().getName()); //  
java.util.Arrays$ArrayList
```

- 3 ) 构建集合 list1，使用 List 的 addAll 方法将数组转换来的集合添加到 list1 中，代码如下所示：

```
List<String> list1 = new ArrayList<String>();
list1.addAll(Arrays.asList(strArr));
```

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先，在类 `TestList` 中新建测试方法 `testArrayToList`；然后构建字符串数组 `strArr`，数组元素为 “a”、“b”、“c”，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {

    /**
     * 测试 asList 方法
     */
    @Test
    public void testArrayToList() {
        String[] strArr = { "a", "b", "c" };
    }

}
```

### 步骤二：使用 `Arrays` 的 `asList` 方法将数组转换为 `List`

使用 `Arrays` 提供的方法 `asList` 将数组 `strArr` 转换为 `List` 集合变量 `list`，并输出集合 `list`，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 asList 方法
     */
    @Test
    public void testArrayToList() {
        String[] strArr = { "a", "b", "c" };

        List<String> list = Arrays.asList(strArr);
        System.out.println(list); // [a, b, c]
    }
}
```

### 步骤三：运行

运行 `testArrayList` 方法，控制台输出的结果为：

```
[a, b, c]
```

从运行输出的结果可以看出，已经将数组 `strArr` 转换为 `list` 集合，集合的元素和数组的元素相同，都为 `[a, b, c]`。

#### 步骤四：向集合中添加元素 “d”

向 `list` 集合中添加元素 “d”，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 asList 方法
     */
    @Test
    public void testArrayList() {
        String[] strArr = { "a", "b", "c" };

        List<String> list = Arrays.asList(strArr);
        System.out.println(list); // [a, b, c]

        list.add("d"); // 会抛出 UnsupportedOperationException
    }
}
```

#### 步骤五：运行

运行 `testArrayList` 方法，会提示发生异常 `UnsupportedOperationException`，如图 - 1 所示。

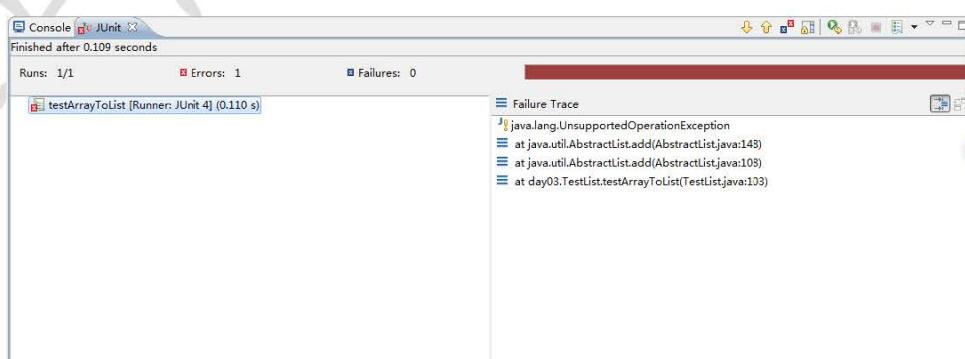


图- 1

从运行效果上看，说明不支持添加操作。如果 `list` 的实际类型是 `ArrayList`，那么应该允许添加操作，因此，要查看一下当前集合 `list` 的实际类型。

## 步骤六：输出 list 集合的实际类型名称

首先，注释掉向集合 list 中插入元素 “d” 的代码；然后，获取集合 list 的实际类型名称，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 asList 方法
     */
    @Test
    public void testArrayList() {
        String[] strArr = { "a", "b", "c" };

        List<String> list = Arrays.asList(strArr);
        System.out.println(list); // [a, b, c]

        // list.add("d"); // 会抛出 UnsupportedOperationException
        System.out.println(list.getClass().getName()); // java.util.Arrays$ArrayList
    }
}
```

## 步骤七：运行

运行 testArrayList 方法，控制台输出结果如下：

```
java.util.Arrays$ArrayList
```

从输出结果可以看出，当前集合 list 的实际类型为 `java.util.Arrays$ArrayList`，即为 `Arrays` 类的内部类 `ArrayList`，而不是我们之前提到的 `List` 接口的实现类 `java.util.ArrayList`。那么，如何正确使用数组转换而来的集合呢。

## 步骤八：数组转换而来的集合的正确使用方式

数组转换而来的集合的正确使用方式，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    /**
     * 测试 asList 方法
     */
    @Test
    public void testArrayList() {
        String[] strArr = { "a", "b", "c" };
    }
}
```

```
List<String> list = Arrays.asList(strArr);
System.out.println(list); // [a, b, c]
// list.add("d"); // 会抛出 UnsupportedOperationException
System.out.println(list.getClass().getName()); // java.util.Arrays$ArrayList

List<String> list1 = new ArrayList<String>();
list1.addAll(Arrays.asList(strArr));

}
```

按照上述的方式，当你向 list1 中添加元素时，将不再会发生异常。

- **完整代码**

本案例中，类 TestList 的完整代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.junit.Test;

public class TestList {
    //... (之前案例的代码，略)

    /**
     * 测试 asList 方法
     */
    @Test
    public void testArrayList() {
        String[] strArr = { "a", "b", "c" };

        List<String> list = Arrays.asList(strArr);
        System.out.println(list); // [a, b, c]
        // list.add("d"); // 会抛出 UnsupportedOperationException
        //java.util.Arrays$ArrayList
        System.out.println(list.getClass().getName());

        List<String> list1 = new ArrayList<String>();
        list1.addAll(Arrays.asList(strArr));
    }
}
```

## 10. 使用 Collections.sort 方法实现排序

- **问题**

使用工具类 Collections 提供的 sort 方法实现排序，详细要求如下：

1 ) 创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；使用 1 作为随机数生成器的种子，生成 10 个 0~100 之间的随机数（包含 0 不包含 100）；将这 10 个随机数放入 list 集合中并输出集合元素。

2 ) 按照自然顺序对 list 集合中的元素进行升序排列并输出排序后的 list 集合。

- 方案

1 ) 首先，创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；然后，使用 java.util 包下的 Random 类来生成随机数，在生成随机数时，使用 1 作为随机数生成器的种子（保证每次生成的随机数相同），生成 10 个 0~100 之间的随机数（包含 0 不包含 100），并将这 10 个随机数放入 list 集合中输出，代码如下所示：

```
List<Integer> list = new ArrayList<Integer>();
Random r = new Random(1);
for (int i = 0; i < 10; i++) {
    list.add(r.nextInt(100));
}
System.out.println(list); // [85, 88, 47, 13, 54, 4, 34, 6, 78, 48]
```

2 ) 使用 Collections 的 sort 方法，按照自然顺序对 list 集合中的元素进行升序排列并输出排序后的 list 集合，代码如下所示：

```
Collections.sort(list);
System.out.println(list); // [4, 6, 13, 34, 47, 48, 54, 78, 85, 88]
```

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先新建类 TestSort，在该类中新建测试方法 testSort，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import org.junit.Test;

public class TestSort {
    /**
     * 测试 sort 方法
     */
    @Test
    public void testSort() {
    }
}
```

### 步骤二：生成 10 个随机数并加入到集合中

首先，创建 List 接口的引用 list，使该引用指向 ArrayList 的实例；然后，使用 java.util 包下的 Random 类来生成随机数，在生成随机数时，使用 1 作为随机数生成器的种子（保证每次生成的随机数相同），生成 10 个 0~100 之间的随机数（包含 0 不包含 100），

并将这 10 个随机数放入 list 集合中输出，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import org.junit.Test;

public class TestSort {
    /**
     * 测试 sort 方法
     */
    @Test
    public void testSort() {

        List<Integer> list = new ArrayList<Integer>();
        Random r = new Random(1);
        for (int i = 0; i < 10; i++) {
            list.add(r.nextInt(100));
        }
        System.out.println(list); // [85, 88, 47, 13, 54, 4, 34, 6, 78, 48]

    }
}
```

### 步骤三：使用 Collections 的 sort 方法对 list 集合排序

使用 Collections 的 sort 方法，按照自然顺序对 list 集合中的元素进行升序排列并输出排序后的 list 集合，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import org.junit.Test;

public class TestSort {
    /**
     * 测试 sort 方法
     */
    @Test
    public void testSort() {
        List<Integer> list = new ArrayList<Integer>();
        Random r = new Random(1);
        for (int i = 0; i < 10; i++) {
            list.add(r.nextInt(100));
        }
        System.out.println(list); // [85, 88, 47, 13, 54, 4, 34, 6, 78, 48]

        Collections.sort(list);
        System.out.println(list); // [4, 6, 13, 34, 47, 48, 54, 78, 85, 88]

    }
}
```

#### 步骤四：运行

运行 `testSort` 方法，控制台输出结果如下所示：

```
[85, 88, 47, 13, 54, 4, 34, 6, 78, 48]  
[4, 6, 13, 34, 47, 48, 54, 78, 85, 88]
```

从运行结果可以看出，实现了对 `list` 集合中的元素从小到大的排序。

#### • 完整代码

本案例中，类 `TestSort` 的完整代码如下所示：

```
package day04;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
import java.util.Random;  
import org.junit.Test;  
  
public class TestSort {  
    /**  
     * 测试 sort 方法  
     */  
    @Test  
    public void testSort() {  
        List<Integer> list = new ArrayList<Integer>();  
        Random r = new Random(1);  
        for (int i = 0; i < 10; i++) {  
            list.add(r.nextInt(100));  
        }  
        System.out.println(list); // [85, 88, 47, 13, 54, 4, 34, 6, 78, 48]  
  
        Collections.sort(list);  
        System.out.println(list); // [4, 6, 13, 34, 47, 48, 54, 78, 85, 88]  
    }  
}
```

## 11. 使用 Comparator 接口实现排序

#### • 问题

使用 `Comparator` 接口实现对集合中的元素排序，详细要求如下：

- 1 ) 使用 `ArrayList` 构建集合对象 `cells`，该集合中要求存储 `Cell` 类型的数据。
- 2 ) 分别构造行和列为 ( 2 , 3 ) ( 5 , 1 ) ( 3 , 2 ) 的 `Cell` 类对象，并将这三个对象加入到集合 `cells` 中。
- 3 ) 使用 `Collections` 类提供的 `sort` 方法，按照 `Cell` 对象的 `col` 值的大小升序排列 `cells` 集合中的对象，`sort` 方法的声明如下：

```
public static <T> void sort(List<T> list,  
                           Comparator<? super T> c)
```

- 方案

首先，使用 `ArrayList` 构建集合对象 `cells`，该集合中要求存储 `Cell` 类型的数据。

然后，分别构造行和列为 (2, 3)、(5, 1)、(3, 2) 的 `Cell` 类对象，并将这三个对象加入到集合 `cells` 中，代码如下：

```
List<Cell> cells = new ArrayList<Cell>();  
cells.add(new Cell(2, 3));  
cells.add(new Cell(5, 1));  
cells.add(new Cell(3, 2));
```

最后，使用 `Collections` 类提供的 `sort` 方法，按照 `Cell` 对象的 `col` 值的大小升序排列 `cells` 集合中的对象。在使用 `sort` 方法时，该方法的第二个参数为 `Comparator` 接口类型，`Comparator` 接口是比较器接口，实现该接口中的 `compare` 方法，给出对象的比较逻辑。下列代码中，给出了 `Cell` 对象的比较逻辑：

```
Collections.sort(cells, new Comparator<Cell>() {  
    @Override  
    public int compare(Cell o1, Cell o2) {  
        return o1.col - o2.col;  
    }  
});
```

以上代码表示的比较逻辑为按照 `Cell` 对象的 `col` 值大小比较集合 `cells` 中的元素的大小。

- 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：构建测试方法

首先，在 `TestSort` 类中新建测试方法 `testComparator`；然后，使用 `ArrayList` 构建集合对象 `cells`，代码如下所示：

```
package day04;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
import java.util.Random;  
import org.junit.Test;  
  
public class TestSort {  
  
    /**  
     * 测试 Comparator 接口  
     */  
    @Test  
    public void testComparator() {  
        List<Cell> cells = new ArrayList<Cell>();  
    }  
}
```

## 步骤二：向集合 cells 中添加对象

分别构造行和列为 ( 2 , 3 ) ( 5 , 1 ) ( 3 , 2 ) 的 Cell 类对象，并将这三个对象加入到集合 cells 中，代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;
import org.junit.Test;

public class TestSort {
    /**
     * 测试 Comparator 接口
     */
    @Test
    public void testComparator() {
        List<Cell> cells = new ArrayList<Cell>();

        cells.add(new Cell(2, 3));
        cells.add(new Cell(5, 1));
        cells.add(new Cell(3, 2));

    }
}
```

## 步骤三：使用 Comparator 接口实现排序

使用 Collections 类提供的 sort 方法 按照 Cell 对象的 col 值的大小升序排列 cells 集合中的对象。代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;
import org.junit.Test;

public class TestSort {
    /**
     * 测试 Comparator 接口
     */
    @Test
    public void testComparator() {
        List<Cell> cells = new ArrayList<Cell>();
        cells.add(new Cell(2, 3));
        cells.add(new Cell(5, 1));
        cells.add(new Cell(3, 2));

        // 按照 col 值的大小排序
        Collections.sort(cells, new Comparator<Cell>() {
```

```
    @Override
    public int compare(Cell o1, Cell o2) {
        return o1.col - o2.col;
    }
});
System.out.println(cells); // [(5,1), (3,2), (2,3)]
```

#### 步骤四：运行

运行 testComparator 方法，控制台输出结果如下所示：

```
[ (5,1), (3,2), (2,3) ]
```

从运行结果可以看出，已经对集合中的 Cell 对象按照 col 升序排列。

- 完整代码

本案例中，类 TestSort 的完整代码如下所示：

```
package day04;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;
import org.junit.Test;

public class TestSort {
    //... (之前案例的代码, 略)

    /**
     * 测试 Comparator 接口
     */
    @Test
    public void testComparator() {
        List<Cell> cells = new ArrayList<Cell>();
        cells.add(new Cell(2, 3));
        cells.add(new Cell(5, 1));
        cells.add(new Cell(3, 2));

        // 按照 col 值的大小排序
        Collections.sort(cells, new Comparator<Cell>() {
            @Override
            public int compare(Cell o1, Cell o2) {
                return o1.col - o2.col;
            }
        });
        System.out.println(cells); // [(5,1), (3,2), (2,3)]
    }
}
```

## 12. 测试 Queue 的用法

- 问题

队列 ( Queue ) 是常用的数据结构，可以将队列看成特殊的线性表，队列限制了对线性表的访问方式：只能从线性表的一端添加 ( offer ) 元素，从另一端取出 ( poll ) 元素。队列遵循先进先出 ( FIFO First Input First Output ) 的原则。

JDK 中提供了 Queue 接口，同时使得 LinkedList 实现了该接口（选择 LinkedList 实现 Queue 的原因在于 Queue 经常要进行插入和删除的操作，而 LinkedList 在这方面效率较高）。

本案例要求测试 Queue 的用法，详细要求如下：

- 1 ) 使用 LinkedList 构建队列，将字符串 “a”、“b”、“c” 放入队列中。
- 2 ) 获取队列中队首元素。
- 3 ) 从队首开始删除元素，直到队列中没有元素为止，并在删除的同时输出删除的队首元素。

#### • 步骤

实现此案例需要按照如下步骤进行。

#### 步骤一：构建队列

首先新建类 TestQueueAndDeque；然后，在该类中新建测试方法 testQueue；最后，使用 LinkedList 构建队列 queue，代码如下所示：

```
package day04;

import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Queue 的用法
     */
    @Test
    public void testQueue() {
        Queue<String> queue = new LinkedList<String>();
    }
}
```

#### 步骤二：向队列中添加元素

使用 Queue 接口提供的 offer 方法，将字符串 “a”、“b”、“c” 添加到队列中并输出队列的元素，代码如下所示：

```
package day04;

import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Queue 的用法
     */
}
```

```
@Test
public void testQueue() {
    Queue<String> queue = new LinkedList<String>();

    queue.offer("a");
    queue.offer("b");
    queue.offer("c");
    System.out.println(queue); // [a, b, c]

}
```

运行 `testQueue` 方法，控制台输出结果如下：

```
[a, b, c]
```

从输出结果可以看出，已经将字符串 “a”、“b”、“c” 添加到队列中。

### 步骤三：查看队首元素

使用 `Queue` 接口提供的 `peek` 方法，查看队列 `queue` 的队首元素并输出队首元素，代码如下所示：

```
package day04;

import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Queue 的用法
     */
    @Test
    public void testQueue() {
        Queue<String> queue = new LinkedList<String>();
        queue.offer("a");
        queue.offer("b");
        queue.offer("c");
        System.out.println(queue); // [a, b, c]

        String str = queue.peek();
        System.out.println(str); // a
    }
}
```

运行 `testQueue` 方法，控制台输出结果为：

```
a
```

从输出结果可以看出，队首元素为 `a`。

### 步骤四：循环删除队首元素

使用循环。在循环中，从队首开始删除元素，直到队列中没有元素为止，并在删除的同时输出删除的队首元素，代码如下所示：

```
package day04;

import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Queue 的用法
     */
    @Test
    public void testQueue() {
        Queue<String> queue = new LinkedList<String>();
        queue.offer("a");
        queue.offer("b");
        queue.offer("c");
        System.out.println(queue); // [a, b, c]

        String str = queue.peek();
        System.out.println(str); // a

        while (queue.size() > 0) {
            str = queue.poll();
            System.out.print(str + " "); // a b c
        }
    }
}
```

另外，从上述代码中可以看到使用了 Queue 接口提供的 poll 方法删除队首元素。

运行 testQueue 方法，控制台输出结果为：

```
a b c
```

上述输出结果为被删除的队首元素。

#### • 完整代码

本案例中，类 TestQueueAndDeque 的完整代码如下所示：

```
package day04;

import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Queue 的用法
     */
    @Test
    public void testQueue() {
        Queue<String> queue = new LinkedList<String>();
        queue.offer("a");
        queue.offer("b");
        queue.offer("c");
    }
}
```

```
System.out.println(queue); // [a, b, c]

String str = queue.peek();
System.out.println(str); // a

while (queue.size() > 0) {
    str = queue.poll();
    System.out.print(str + " "); // a b c
}
}
```

## 13. 测试 Deque 的用法

- 问题

Deque 是 Queue 的子接口，定义了所谓“双端队列”即从队列的两端分别可以入队（offer）和出队（poll），LinkedList 实现了该接口。如果将 Deque 限制为只能从一端入队和出队，则可实现“栈”（Stack）的数据结构，对于栈而言，入栈称之为 push，出栈称之为 pop。栈遵循先进后出（FILO First Input Last Output）的原则。

本案例要求测试 Deque 的用法，详细要求如下：

- 1 ) 使用 LinkedList 构建栈，将字符串 “a”、“b”、“c” 放入栈中。
- 2 ) 获取栈中栈顶元素。
- 3 ) 从栈顶开始删除元素，直到栈中没有元素为止，并在删除的同时输出删除的栈顶元素。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建队列

首先，在类 TestQueueAndDeque 中新建测试方法 testStack；然后，使用 LinkedList 构建栈 stack，代码如下所示：

```
package day04;

import java.util.Deque;
import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Stack 的用法
     */

    @Test
    public void testStack() {
        Deque<String> stack = new LinkedList<String>();
    }
}
```

## 步骤二：向栈中添加元素

使用 Deque 接口提供的 push 方法，将字符串 “a”、“b”、“c” 添加到栈中并输出栈的元素，代码如下所示：

```
package day04;

import java.util.Deque;
import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Stack 的用法
     */
    @Test
    public void testStack() {
        Deque<String> stack = new LinkedList<String>();

        stack.push("a");
        stack.push("b");
        stack.push("c");
        System.out.println(stack); // [c, b, a]

    }
}
```

运行 testStack 方法，控制台输出结果如下：

```
[c, b, a]
```

从输出结果可以看出，已经将字符串 “a”、“b”、“c” 添加到栈中。

## 步骤三：查看栈顶元素

使用 Deque 接口提供的 peek 方法，查看栈 stack 的栈顶元素并输出栈顶元素，代码如下所示：

```
package day04;

import java.util.Deque;
import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Stack 的用法
     */
    @Test
    public void testStack() {
        Deque<String> stack = new LinkedList<String>();
        stack.push("a");
        stack.push("b");
        stack.push("c");
        System.out.println(stack); // [c, b, a]
    }
}
```

```
String str = stack.peek();
System.out.println(str); // c

}
```

运行 testStack 方法，控制台输出结果为：

```
c
```

从输出结果可以看出，栈顶元素为 c。

#### 步骤四：循环删除栈顶元素

使用循环。在循环中，从栈顶开始删除元素，直到栈中没有元素为止，并在删除的同时输出删除的栈顶元素，代码如下所示：

```
package day04;

import java.util.Deque;
import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    /**
     * 测试 Stack 的用法
     */
    @Test
    public void testStack() {
        Deque<String> stack = new LinkedList<String>();
        stack.push("a");
        stack.push("b");
        stack.push("c");
        System.out.println(stack); // [c, b, a]

        String str = stack.peek();
        System.out.println(str); // c

        while (stack.size() > 0) {
            str = stack.pop();
            System.out.print(str + " "); // c b a
        }
    }
}
```

另外，从上述代码中可以看到使用了 Deque 接口提供的 pop 方法使栈顶元素出栈。

运行 testStack 方法，控制台输出结果为：

```
c b a
```

上述输出结果，可以看出栈的特点为先进后出。

- **完整代码**

本案例中，类 TestQueueAndDeque 的完整代码如下所示：

```
package day04;

import java.util.Deque;
import java.util.LinkedList;
import java.util.Queue;
import org.junit.Test;

public class TestQueueAndDeque {
    //... (之前案例的代码, 略)

    /**
     * 测试 Stack 的用法
     */
    @Test
    public void testStack() {
        Deque<String> stack = new LinkedList<String>();
        stack.push("a");
        stack.push("b");
        stack.push("c");
        System.out.println(stack); // [c, b, a]

        String str = stack.peek();
        System.out.println(str); // c

        while (stack.size() > 0) {
            str = stack.pop();
            System.out.print(str + " "); // c b a
        }
    }
}
```

## 课后作业

### 1. 下面代码输出的结果是？

请看下列代码：

```
public void testAddAllAndContainsAll() {  
    Collection<String> c1 = new ArrayList<String>();  
    c1.add("terry");  
    c1.add("allen");  
    System.out.println(c1);  
    Collection<String> c2 = new HashSet<String>();  
    c2.addAll(c1);  
    System.out.println(c2);  
    Collection<String> c3 = new ArrayList<String>();  
    c3.add("terry");  
    System.out.println(c1.containsAll(c3));  
}
```

运行 testAddAllAndContainsAll 方法，程序的输出结果是：( )。

- A . [terry, allen]  
[allen, terry]  
false
- B . [terry, allen]  
[allen, terry]  
true
- C . [terry, allen]  
[]  
True
- D . [terry, allen]  
[]  
false

### 2. 关于 Iterator，说法正确的是？

- A . Iterator 用于遍历集合元素。获取 Iterator 可以使用 Collection 定义的 iterator 方法。
- B . Iterator 提供了统一的遍历集合元素的方式，其提供了用于遍历集合的两个方法，`hasNext` 用于返回迭代的下一个元素，`next` 方法用于判断集合是否还有元素可以遍历。
- C . 在使用 Iterator 遍历集合时，不能通过集合的 `remove` 方法删除集合元素，否则会抛出异常。我们可以通过迭代器自身提供的 `remove()` 方法来删除通过 `next()` 迭代出的元素。
- D . Java5.0 之后推出了一个新的特性，增强 `for` 循环，也称为新循环。该循环只用于遍历集合或数组。在遍历集合时，该循环是与 Iterator 完全不同的迭代方式。

### 3. 简述 ArrayList 和 LinkedList 的不同

#### 4. 下面代码输出的结果是？

请看下列代码：

```
public void testInsertAndRemove() {  
    List<String> list = new ArrayList<String>();  
    list.add("terry");  
    list.add("allen");  
    list.add("smith");  
    list.add(2, "marry");  
    System.out.println(list);  
    list.remove(1);  
    System.out.println(list);  
}
```

运行 testInsertAndRemove 方法，程序的输出结果是：( )。

- A . [terry, marry, allen, smith]  
[terry, marry, smith]
- B . [terry, marry, allen, smith]  
[allen, marry, smith]
- C . [terry, allen, marry, smith]  
[allen, marry, smith]
- D . [terry, allen, marry, smith]  
[terry, marry, smith]

#### 5. 下面代码输出的结果是？

请看下列代码：

```
public void testSubList() {  
    List<Integer> list = new ArrayList<Integer>();  
    for (int i = 0; i < 10; i++) {  
        list.add(i);  
    }  
    List<Integer> subList = list.subList(2, 5);  
    for (int i = 0; i < subList.size(); i++) {  
        subList.set(i, subList.get(i) * 10);  
    }  
    System.out.println(subList);  
    System.out.println(list);  
}
```

运行 testSubList 方法，程序的输出结果是：( )。

- A . [20, 30, 40, 50]  
[0, 1, 20, 30, 40, 50, 6, 7, 8, 9]
- B . [20, 30, 40, 50]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
- C . [20, 30, 40]

- 
- [0, 1, 20, 30, 40, 5, 6, 7, 8, 9]  
D . [20, 30, 40]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## 6. 简述 List 和数组直接相互转化的方法

## 7. 使用 Comparator 接口实现排序

使用 Comparator 接口实现对集合中的元素排序，详细要求如下：

- 1 ) 使用 ArrayList 构建集合对象 emps，要求该集合中存储 Emp 类型的数据。
- 2 ) 分别构造 name、age、gender 以及 salary 为( "Terry", 25, 'm', 6000 )( "Allen", 21, 'f', 4000)、( "Smith", 23, 'm', 3000 )的三个 Emp 类的对象，并将这三个对象加入到集合 emps 中。
- 3 ) 使用 Collections 类提供的 sort 方法，按照 Emp 对象的 salary 属性的值升序排列集合 emps，sort 方法的声明如下：

```
public static <T> void sort(List<T> list,  
                           Comparator<? super T> c)
```

## 8. 简述队列和栈的不同，以及在 Java 语言中如何实现这两个数据结构

# Java 核心 API(上)

## Unit05

知识体系.....**Page 204**

|           |                  |                                 |
|-----------|------------------|---------------------------------|
| 查询表       | Map 接口           | Map 接口                          |
|           |                  | put()方法                         |
|           |                  | get()方法                         |
|           |                  | containsKey()方法                 |
|           | HashMap          | Hash 表原理                        |
|           | hashcode 方法      |                                 |
|           | 重写 hashCode 方法   |                                 |
|           | 装载因子及 HashMap 优化 |                                 |
|           | Map 的遍历          | 使用 keyset()方法                   |
|           |                  | 使用 entryset()方法                 |
|           | 有序的 Map          | LinkedHashMap 实现有序 Map          |
| 文件操作—File | 创建 File 对象       | File(String pathname)           |
|           |                  | File(File parent, String child) |
|           |                  | isFile()方法                      |
|           | File 表示文件信息      | length()方法                      |
|           |                  | exists()方法                      |
|           |                  | createNewFile()方法               |
|           |                  | delete()方法                      |
|           |                  | isDirectory()方法                 |
|           | File 表示目录信息      | mkdir()方法                       |
|           |                  | mkdirs()方法                      |
|           |                  | delete()方法                      |

经典案例.....**Page 215**

|                            |                |
|----------------------------|----------------|
| PM2.5 监控程序——统计各点 PM2.5 最大值 | put 方法         |
|                            | get 方法         |
|                            | containsKey 方法 |
| hashCode 方法的重写             | hashCode 方法    |
| PM2.5 监控程序——遍历各点 PM2.5 最大值 | 使用 keyset()方法  |

|                                       |                        |
|---------------------------------------|------------------------|
|                                       | 使用 entryset()方法        |
| PM2.5 监控程序—统计并遍历各点 PM2.5 最大值 ( 要求顺序 ) | LinkedHashMap 实现有序 Map |
| 查看一个文件的大小                             | length()方法             |
| 创建一个空文件                               | exists()方法             |
|                                       | createNewFile()方法      |
| 删除一个文件                                | delete()方法             |
| 创建一个目录                                | mkdir()方法              |
| 创建一个多级目录                              | mkdirs()方法             |
| 删除一个目录                                | delete()方法             |

课后作业.....Page 241

## 1. 查询表

### 1.1. Map 接口

#### 1.1.1. 【Map 接口】 Map 接口

Map接口

• Map接口定义的集合又称查找表，用于存储所谓“Key-Value”映射对。Key可以看成是Value的索引，作为Key的对象在集合中不可以重复。

• 根据内部数据结构的不同，Map接口有多种实现类，其中常用的有内部为hash表实现的HashMap和内部为排序二叉树实现的TreeMap。

#### 1.1.2. 【Map 接口】 put()方法

put()方法

• Map接口中定义了向Map中存放元素的put方法：  
– V put(K key, V value)

• 将Key-Value对存入Map，如果在集合中已经包含该Key，则操作将替换该Key所对应的Value，返回值为该Key原来所对应的Value（如果没有则返回null）。

put()方法(续1)

```
public void testPut(){  
    //向map中添加元素  
    employees.put(  
        "张三", new Emp("张三",25,"男",5000)  
    );  
    employees.put(  
        "李四", new Emp("李四",21,"女",6000)  
    );  
}
```

### 1.1.3. 【Map 接口】get()方法

知识讲解

#### get()方法

- Map接口中定义了从Map中获取元素的get方法:
  - V get(Object key)
- 返回参数key所对应的Value对象，如果不存在则返回null。

+

知识讲解

#### get()方法(续1)

```
public void testGet(){  
    //从Map中使用key获取value  
    Emp emp = employees.get("张三");  
    System.out.println(emp);  
}
```

+

### 1.1.4. 【Map 接口】containsKey()方法

知识讲解

#### containsKey()方法

- Map接口中定义了判断某个key是否在Map中存在:
  - boolean containsKey(Object key);  
若Map中包含给定的key则返回true，否则返回false。

+

### containsKey()方法(续1)

```
public void testContainsKey(){
    boolean has = employees.containsKey("李四");
    System.out.println("是否有员工李四:" + has);
}
```

知识讲解



## 1.2. HashMap

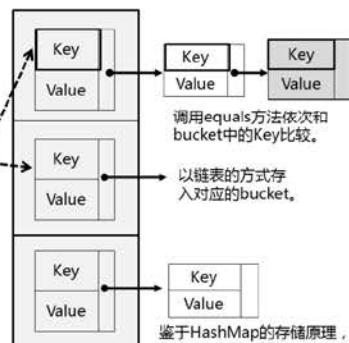
### 1.2.1. 【HashMap】 Hash 表原理

#### Hash表原理

获取Key的hashCode值，通过hash算法确定将要存储的空间 ( bucket )。



获取Key的hashCode值，通过hash算法确定要检索的空间 ( bucket )。



鉴于HashMap的存储原理，存入HashMap的Key必须妥善的重写hashCode方法。



### 1.2.2. 【HashMap】 hashCode 方法

#### hashCode方法

- 从HashMap的原理中我们可以看到，key的hashCode()方法的返回值对HashMap存储元素时会起着很重要的作用。而hashCode()方法实际上是在Object中定义的。那么应当妥善重写该方法：
- 对于重写了equals方法的对象，一般要妥善的重写继承自Object类的hashCode方法（Object提供的hashCode方法将返回该对象所在内存地址的整数形式）。
- 重写hashCode方法是需注意两点：其一、与equals方法的一致性，即equals比较返回true的两个对象其hashCode方法返回值应该相同；其二、hashCode返回的数值应符合hash算法的要求，试想如果有很多对象的hashCode方法返回值都相同，则会大大降低hash表的效率，一般情况下可以使用IDE（如Eclipse）提供的工具自动生成hashCode方法。



### 1.2.3. 【HashMap】重写 hashCode 方法

**重写hashCode方法**

public int hashCode() {  
 final int prime = 31;  
 int result = 1;  
 result = prime \* result + age;  
 result = prime \* result + ((gender == null) ? 0 :  
 gender.hashCode());  
 result = prime \* result + ((name == null) ? 0 :  
 name.hashCode());  
 long temp;  
 Temp = Double.doubleToLongBits(salary);  
 result = prime \* result + (int) (temp ^ (temp  
 >>> 32));  
 return result;  
}

Tarena  
Technology  
达内科技

### 1.2.4. 【HashMap】装载因子及 HashMap 优化

**装载因子及HashMap优化**

- Capacity : 容量, hash表里bucket(桶)的数量, 也就是散列数组大小。
- Initial capacity : 初始容量, 创建hash表时,初始bucket的数量, 默认构建容量是16. 也可以使用特定容量。
- Size : 大小, 当前散列表中存储数据的数量。
- Load factor : 加载因子, 默认值0.75(就是75%), 当向散列表增加数据时如果 size/capacity 的值大于Load factor则发生扩容并且重新散列(rehash)。
- 性能优化: 加载因子较小时,散列查找性能会提高, 同时也浪费了散列桶空间容量。 0.75是性能和空间相对平衡结果。在创建散列表时指定合理容量, 减少rehash提高性能。

Tarena  
Technology  
达内科技

## 1.3. Map 的遍历

### 1.3.1. 【Map 的遍历】使用 keyset()方法

**使用keyset()方法**

- Map提供了三种遍历方式:
  - 遍历所有的Key
  - 遍历所有的Key-Value对
  - 遍历所有的Value(不常用)
- 遍历所有Key的方法:
  - Set<K> keySet()
  - 该方法会将当前Map中所有的key存入一个Set集合后返回。

Tarena  
Technology  
达内科技

### 1.3.2. 【Map 的遍历】使用 entryset()方法

知识讲解

+

**使用entryset()方法**

Tarena  
Technology  
达内科技

- 遍历所有的键值对的方法：
  - Set<Entry<K,V>> entrySet()
  - 该方法会将当前Map中每一组key-value对封装为一个Entry对象并存入一个Set集合后返回。

## 1.4. 有序的 Map

### 1.4.1. 【有序的 Map】 LinkedHashMap 实现有序 Map

知识讲解

+

**LinkedHashMap实现有序Map**

Tarena  
Technology  
达内科技

- 使用Map接口的哈希表和链表实现，具有可预知的迭代顺序。此实现与 HashMap 的不同之处在于：
  - LinkedHashMap维护着一个双向循环链表。此链表定义了迭代顺序，该迭代顺序通常就是存放元素的顺序。
- 需要注意的是，如果在Map中重新存入已有的key，那么key的位置不会发生改变，只是将value值替换。

## 2. 文件操作—File

### 2.1. 创建 File 对象

#### 2.1.1. 【创建 File 对象】 File(String pathname)

知识讲解

+

**File(String pathname)**

Tarena  
Technology  
达内科技

- java.io.File用于表示文件（目录），也就是说程序员可以通过File类在程序中操作硬盘上的文件和目录。
- File类只用于表示文件（目录）的信息（名称、大小等），不能对文件的内容进行访问。
- 构造方法：
  - File(String pathname)
  - 通过将给定路径名字符串转换成抽象路径名来创建一个新 File 实例
  - 抽象路径应尽量使用相对路径，并且目录的层级分隔符不要直接写“/”或“\”，应使用File.separator这个常量表示，以避免不同系统带来的差异。

### File(String pathname)(续1)

```
public void testFile(){  
    File file = new File(  
        "demo"+File.separator  
        +"HelloWorld.txt"  
    );  
    System.out.println(file);  
}
```

知识讲解



### 2.1.2. 【创建 File 对象】 File(File parent, String child)

### File(File parent, String child)

- File还提供另一个构造方法:
  - File(File parent, String child)
  - 根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例

知识讲解



### File(File parent, String child)(续1)

```
public void testFile2(){  
    File parent = new File("demo");  
    File file = new File(parent,"HelloWorld.txt");  
    System.out.println(file);  
}
```

知识讲解



### 2.1.3. 【创建 File 对象】`isFile()`方法

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

+

**isFile()方法**

**Tarena**  
Technology  
达内科技

• File的`isFile`方法用于判断当前File对象所表示的是否为一个文件

- boolean `isFile()`
- 返回值:当前File对象所表示是一个文件时返回true

+

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

+

**isFile()方法(续1)**

**Tarena**  
Technology  
达内科技

```
public void testIsFile(){  
    File file = new File(  
        "demo"+File.separator  
        +"HelloWorld.txt"  
    );  
    System.out.println(  
        file+"是否是一个文件"+file.isFile()  
    );  
}
```

+

## 2.2. File 表示文件信息

### 2.2.1. 【File 表示文件信息】`length()`方法

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

+

**length()方法**

**Tarena**  
Technology  
达内科技

• File的`length`方法用于返回由此抽象路径名表示的文件的长度(占用的字节数量)

- long `length()`
- 返回值:当前File对象所表示的文件所占用的字节数量

+

length()方法(续1)

```
public void testLength(){  
    File file = new File(  
        "demo"+File.separator  
        +"HelloWorld.txt "  
    );  
    System.out.println(  
        file + "占用字节量:" + file.length()  
    );  
}
```

知识讲解

+

### 2.2.2. 【File 表示文件信息】exists()方法

exists()方法

- File的exists方法用于测试此抽象路径名表示的文件或目录是否存在
  - boolean exists()
  - 返回值:若该File表示的文件或目录存在则返回true,否则返回false

知识讲解

+

### 2.2.3. 【File 表示文件信息】createNewFile()方法

createNewFile()方法

- File的createNewFile方法用于当且仅当不存在具有此抽象路径名指定的名称的文件时，原子地创建由此抽象路径名指定的一个新的空文件。
  - boolean createNewFile()
  - 返回值:如果指定的文件不存在并成功地创建，则返回 true；如果指定的文件已经存在，则返回 false

知识讲解

+

#### 2.2.4. 【File 表示文件信息】 delete()方法

知识讲解

### delete()方法

• File的delete方法用于删除此抽象路径名表示的文件或目录。  
– boolean delete()  
– 返回值:当且仅当成功删除文件或目录时，返回 true；否则返回 false  
– 需要注意的是，若此File对象所表示的是一个目录时，在删除时需要保证此为空目录才可以成功删除(目录中不能含有任何子项)。

+

知识讲解

### delete()方法(续1)

```
public void testDeleteFile(){  
    File file = new File  
        ("demo"+File.separator  
        +"Hello.txt ");  
    file.delete();  
}
```

+

#### 2.2.5. 【File 表示文件信息】 isDirectory()方法

知识讲解

### isDirectory()方法

• File的isDirectory方法用于判断当前File表示的是否为一个目录。  
– boolean isDirectory()  
– 返回值：当File对象表示的是一个目录时返回 true；否则返回 false

+

知识讲解

isDirectory()方法(续1)

```
public void testIsDirectory(){  
    File file = new File("demo");  
    System.out.println(  
        file+"是否是一个目录"+file.isDirectory()  
    );  
}
```

+

## 2.3. File 表示目录信息

### 2.3.1. 【File 表示目录信息】mkdir()方法

知识讲解

mkdir()方法

- File的mkdir方法用于创建此抽象路径名指定的目录。
  - boolean mkdir()
  - 返回值：当且仅当已创建目录时，返回 true；否则返回 false

+

知识讲解

mkdir()方法(续1)

```
public void testMkdir(){  
    File dir = new File("myDir");  
    dir.mkdir();  
}
```

+

### 2.3.2. 【File 表示目录信息】mkdirs()方法

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

+

知识讲解

#### mkdirs()方法

**Tarena**  
达内科技

- File的mkdirs方法用于创建此抽象路径名指定的目录，包括所有必需但不存在的父目录。注意，此操作失败时也可能已经成功地创建了一部分必需的父目录。

– `boolean mkdirs()`

– 返回值：当且仅当已创建目录以及所有必需的父目录时，返回 `true`；否则返回 `false`

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

+

知识讲解

#### mkdirs()方法(续1)

**Tarena**  
达内科技

```
public void testMkDirs(){  
    File dir = new File(  
        "a" + File.separator +  
        "b" + File.separator +  
        "c");  
    dir.mkdirs();  
}
```

### 2.3.3. 【File 表示目录信息】delete()方法

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

+

知识讲解

#### delete()方法

**Tarena**  
达内科技

- File的delete方法用于删除此抽象路径名表示的文件或目录。

– `boolean delete()`

– 返回值：当且仅当成功删除文件或目录时，返回 `true`；否则返回 `false`

– 需要注意的是，若此File对象所表示的是一个目录时，在删除时需要保证此为空目录才可以成功删除(目录中不能含有任何子项)。

## 经典案例

### 1. PM2.5 监控程序——统计各点 PM2.5 最大值

#### • 问题

北京某日空气质量 PM2.5 监测站点全程检测的数据如下表所示，一个监测站点有可能在一天内多次采集，因此同一个站点可能有多个数据，如表-1 所示；

表-1 PM2.5 监测数据

| 检测站点  | PM2.5 浓度 |
|-------|----------|
| 东四    | 423      |
| 丰台花园  | 378      |
| 天坛    | 406      |
| 海淀区万柳 | 322      |
| 官园    | 398      |
| 通州    | 406      |
| 昌平镇   | 366      |
| 怀柔镇   | 248      |
| 定陵    | 306      |
| 前门    | 231      |
| 永乐店   | 422      |
| 古城    | 368      |
| 昌平镇   | 268      |
| 怀柔镇   | 423      |
| 定陵    | 267      |
| 前门    | 377      |
| 永乐店   | 299      |
| 秀水街   | 285      |

请统计各空气质量监测站点 PM2.5 的最高值。

#### • 方案

分析问题中提出的要求为统计各空气质量监测站点 PM2.5 的最高值，再结合提供的 PM2.5 浓度数据，可以发现，站点名可以作为 Map 的 key，对应的 PM2.5 浓度的数据可以作为 Map 的 value，利用 Map 接口提供的 put 方法，该方法可以根据 key 去更新 value 来实现统计各个空气监测站点 PM2.5 的最高值。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：新建 TestHashMap 类，添加测试方法 test1 方法

首先，在工程 JavaSE 下新建包 day05，在该包下新建类 TestHashMap；然后，在该类中添加方法 test1，代码如下所示：

```
package day05;

import org.junit.Test;

public class TestHashMap{
    /**
     *统计各空气质量监测站点 PM2.5 的最高值
     */
    @Test
    public void test1() {

    }
}
```

### 步骤二：将数据拼接成字符串

将问题中的表格数据拼接成字符串，形式如：“农展馆=423,东四=378”；然后，使用 String 类的 split 方法，将拼接后字符串使用正则[ , = ]进行拆分，代码如下所示：

```
package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    /**
     * 统计各空气质量监测站点 PM2.5 的最高值
     */
    @Test
    public void test1() {

        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
                + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
                + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
                + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛=277,"
                + "海淀区万柳=270,官园=268,通州=315";
        String[] arr = pm25.split("[,=]");
    }
}
```

上述代码中，拆分后的数组数据如下所示：

```
[农展馆, 423, 东四, 378, 丰台花园, 406, 天坛, 322, 海淀区万柳, 398, 官园, 406, 通]
```

州, 366, 昌平镇, 248, 怀柔镇, 306, 定陵, 231, 前门, 422, 永乐店, 368, 古城, 268, 昌平镇, 423, 怀柔镇, 267, 定陵, 377, 前门, 299, 永乐店, 285, 秀水街, 277, 农展馆, 348, 东四, 356, 丰台花园, 179, 天坛, 277, 海淀区万柳, 270, 官园, 268, 通州, 315]

观察上述数组会发现，如果农展馆的索引为 0，其对应的数据的索引为 1，并且每隔一个就是下一个站点的名字，如果循环取各个站点名，那么代码如下：

```
for (int i = 0; i < arr.length; i += 2) {}
```

其中 arr[i] 为站点名，arr[i+1] 为对应的 PM2.5 数据。

### 步骤三：利用 Map 的特点，获取各个站点的 PM2.5 最大值

首先，构建 map 对象，该 map 对象的 key 存储站点名，value 存储对应的浓度。

然后，构建循环，循环的次数为 arr 数组的长度；在循环中，获取各个站点的 PM2.5 最大值。

最后，输出 map 对象，可以看到字符串中各个字符的个数，代码如下所示：

```
package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    /**
     * 统计各空气质量监测站点 PM2.5 的最高值
     */

    @Test
    public void test1() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
                + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
                + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
                + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛"
                + "=277,"
                + "海淀区万柳=270,官园=268,通州=315";
        String[] arr = pm25.split(",=");

        Map<String, Integer> map = new HashMap<String, Integer>();
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        System.out.println(map);
    }
}
```

- 完整代码

本案例的完整代码如下所示：

```
package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    /**
     * 统计各空气质量监测站点 PM2.5 的最高值
     */

    @Test
    public void test1() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398," +
                      "+官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422," +
                      "+永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299," +
                      "+永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛" +
                      "=277," +
                      "+海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new HashMap<String, Integer>();
        String[] arr = pm25.split(",=]");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        System.out.println(map);
    }
}
```

## 2. hashCode 方法的重写

- 问题

测试 hashCode 的作用，详细要求如下：

- 1 ) 将没有重写 hashCode 方法的 Emp 类存入 Map 进行中，进行测试。
- 2 ) 将重写了 hashCode 方法的 Emp 类存入 Map 进行中，再次进行测试，并说明两次不同结果的原因。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：新建 Emp 类

在 JavaSE 工程的 day05 包下，新建 Emp 类，该类的代码如下所示：

```
package day05;

public class Emp {
```

```

private String name;
private int age;
private String gender;
private double salary;

public Emp(String name, int age, String gender, double salary) {
    this.name = name;
    this.age = age;
    this.gender = gender;
    this.salary = salary;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public double getSalary() {
    return salary;
}

public void setSalary(double salary) {
    this.salary = salary;
}

@Override
public String toString() {
    return "Emp [name=" + name + ", age=" + age + ", gender=" + gender
           + ", salary=" + salary + "]";
}
}

```

## 步骤二：在 Emp 类中重写 equals 方法

在 Emp 类中重写 equals 方法，使得两个 Emp 引用变量的 name 值相等，则两个 Emp 对象的 equals 方法返回 true，代码如下所示：

```

package day05;

public class Emp {
    private String name;
    private int age;
    private String gender;
    private double salary;

    public Emp(String name, int age, String gender, double salary) {

```

```

        this.name = name;
        this.age = age;
        this.gender = gender;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Emp [name=" + name + ", age=" + age + ", gender=" + gender
               + ", salary=" + salary + "]";
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Emp other = (Emp) obj;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}

```

### 步骤三： 使用 Map 存储 Emp 类的对象

在 JavaSE 工程的 day05 包下新建类 TestMap，在该类中，首先，新建 testPut 方法，在方法中将 Emp 对象作为 key、名字作为 value 存储 Map 对象 employees 中；然后，新建 testGet 方法，该方法用于从 Map 中根据 key 获取 value；最后，新建 testContainsKey 方法，该方法用于判断 Map 对象 employees 中是否包含某个 Emp 对象，代码如下所示：

```
package day05;

import java.util.HashMap;
import java.util.Map;
import org.junit.Before;
import org.junit.Test;

public class TestMap {
    private Map<Emp, String> employees = new HashMap<Emp, String>();
    @Before
    public void testPut(){
        //向 map 中添加元素
        employees.put(new Emp("张三", 25, "男", 5000), "张三");
        employees.put(new Emp("李四", 21, "女", 6000), "李四");
    }
    @Test
    public void testGet(){
        Emp zhangsan=new Emp("张三", 25, "男", 5000);
        String name = employees.get(zhangsan);
        System.out.println(name);
    }
    @Test
    public void testContainsKey(){
        Emp zhangsan=new Emp("张三", 25, "男", 5000);
        boolean has = employees.containsKey(zhangsan);
        System.out.println("是否有员工李四：" + has);
    }
}
```

在上述代码中的 testPut 方法之前，使用了@Before 注解，在 JUnit 中该注解表示每个测试方法（@Test 修饰的方法）执行时都会执行一次有其修饰方法，用于数据的初始化。

运行 TestMap 类，控制台输出结果如下：

```
null
是否有员工李四: false
```

从输出结果可以看出，并没有从 map 对象 employees 中，获取到名字为“张三”的 Emp 对象。这是因为，比较两个对象的时候，首先根据他们的 hashCode 去 hash 表中找它的对象，当两个对象的 hashCode 相同，也就是说这两个对象放在 Hash 表中的同一个 key 上，则他们一定在这个 key 上的链表上。那么，此时就只能根据 Object 的 equals 方法来比较这个对象是否相等。当两个对象的 hashCode 不同时，则这两个对象一定不能相等。

#### 步骤四：在 Emp 类中重写 hashCode 方法

在 Emp 类中重写 hashCode 方法。重写 hashCode 方法是需注意两点：其一、与 equals 方法的一致性，即 equals 比较返回 true 的两个对象其 hashCode 方法返回值应该相同；

其二、`hashCode` 返回的数值应符合 `hash` 算法的要求，试想如果有很多对象的 `hashCode` 方法返回值都相同，则会大大降低 `hash` 表的效率，一般情况下可以使用 IDE( 如 Eclipse ) 提供的工具自动生成 `hashCode` 方法。代码如下所示：

```
package day05;

public class Emp {
    private String name;
    private int age;
    private String gender;
    private double salary;

    public Emp(String name, int age, String gender, double salary) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Emp [name=" + name + ", age=" + age + ", gender=" + gender
               + ", salary=" + salary + "]";
    }
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}
```

```
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Emp other = (Emp) obj;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    return true;
}
```

## 步骤五：运行

运行 TestMap 类，控制台输出结果如下：

```
是否有员工李四: true
张三
```

从输出结果可以看出，Emp 重写 hashCode 方法后，可以在 Map 对象 employees 中查找到名称“张三”的对象了。

### • 完整代码

本案例中，类 Emp 的完整代码如下所示：

```
package day05;

public class Emp {
    private String name;
    private int age;
    private String gender;
    private double salary;

    public Emp(String name, int age, String gender, double salary) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
```

```

        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Emp [name=" + name + ", age=" + age + ", gender=" + gender
               + ", salary=" + salary + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
}

```

类 TestMap 的完整代码如下所示：

```

package day05;

import java.util.HashMap;
import java.util.Map;
import org.junit.Before;
import org.junit.Test;

public class TestMap {
    private Map<Emp, String> employees = new HashMap<Emp, String>();
    @Before
    public void testPut(){
        //向 map 中添加元素
        employees.put( new Emp("张三",25,"男",5000),"张三");
        employees.put(new Emp("李四",21,"女",6000),"李四");
    }
    @Test
    public void testGet(){
        Emp zhangsan=new Emp("张三",25,"男",5000);
        String name = employees.get(zhangsan);
        System.out.println(name);
    }
    @Test
    public void testContainsKey(){

```

```

        Emp zhangsan=new Emp("张三",25,"男",5000);
        boolean has = employees.containsKey(zhangsan);
        System.out.println("是否有员工李四：" + has);
    }
}

```

### 3. PM2.5 监控程序——遍历各点 PM2.5 最大值

- 问题

在“PM2.5 监控程序——统计各点 PM2.5 最大值”案例的基础上，分别使用迭代 Key 的方式和迭代 Entry 的方式遍历集合 map。

- 步骤

实现此案例需要按照如下步骤进行。

**步骤一：构建测试方法**

在 TestHashMap 类中添加测试方法 test2，使 test2 中的代码和 test1 中的代码保持一致，代码如下所示：

```

package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {

    /**
     * 使用迭代 key 的方式遍历 map 集合
     */
    @Test
    public void test2() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
            + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
            + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
            + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛=277,"
            + "海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new HashMap<String, Integer>();
        String[] arr = pm25.split(",=");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        System.out.println(map);
    }
}

```

## 步骤二：使用迭代 key 的方式遍历集合 map

使用迭代 key 的方式遍历集合 map，代码如下所示：

```
package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    /**
     * 使用迭代 key 的方式遍历 map 集合
     */
    @Test
    public void test2() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398," +
                      "+ 官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422," +
                      "+ 永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299," +
                      "+ 永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛" +
                      "=277," +
                      "+ 海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new HashMap<String, Integer>();
        String[] arr = pm25.split(",=");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        System.out.println(map);

        Set<String> keys = map.keySet();
        for (String key : keys) {
            System.out.println(key + ":" + map.get(key));
        }
    }
}
```

## 步骤三：构建测试方法

在 TestHashMap 类中添加测试方法 test3，使 test3 中的代码和 test1 中的代码保持一致，代码如下所示：

```
package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    /**
     * 使用迭代 key 的方式遍历 map 集合
     */
    @Test
    public void test3() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398," +
                      "+ 官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422," +
                      "+ 永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299," +
                      "+ 永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛" +
                      "=277," +
                      "+ 海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new HashMap<String, Integer>();
        String[] arr = pm25.split(",=");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        System.out.println(map);

        Set<String> keys = map.keySet();
        for (String key : keys) {
            System.out.println(key + ":" + map.get(key));
        }
    }
}
```

```

public void test2() {
    String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
        + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
        + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
        + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
        + "海淀区万柳=270,官园=268,通州=315";
    Map<String, Integer> map = new HashMap<String, Integer>();
    String[] arr = pm25.split(",=");
    for (int i = 0; i < arr.length; i += 2) {
        if (!map.containsKey(arr[i]))
            || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
            map.put(arr[i], Integer.parseInt(arr[i + 1]));
        }
    }
    Set<String> keys = map.keySet();
    for (String key : keys) {
        System.out.println(key + ":" + map.get(key));
    }
}

/**
 * 使用迭代 Entry 的方式遍历 map 集合
 */
@Test
public void test3() {
    String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
        + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
        + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
        + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛=277,"
        + "海淀区万柳=270,官园=268,通州=315";
    Map<String, Integer> map = new HashMap<String, Integer>();
    String[] arr = pm25.split(",=");
    for (int i = 0; i < arr.length; i += 2) {
        if (!map.containsKey(arr[i]))
            || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
            map.put(arr[i], Integer.parseInt(arr[i + 1]));
        }
    }
    System.out.println(map);
}
}

```

#### 步骤四：使用迭代 Entry 的方式迭代集合 map

使用迭代 Entry 的方式迭代集合 map，代码如下所示：

```

package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    /**

```

```

* 使用迭代 key 的方式遍历 map 集合
*/
@Test
public void test2() {
    String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
        + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
        + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
        + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
        + "海淀区万柳=270,官园=268,通州=315";
    Map<String, Integer> map = new HashMap<String, Integer>();
    String[] arr = pm25.split(",=");
    for (int i = 0; i < arr.length; i += 2) {
        if (!map.containsKey(arr[i]))
            || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
            map.put(arr[i], Integer.parseInt(arr[i + 1]));
        }
    }
    Set<String> keys = map.keySet();
    for (String key : keys) {
        System.out.println(key + ":" + map.get(key));
    }
}

/**
 * 使用迭代 Entry 的方式遍历 map 集合
*/
@Test
public void test3() {
    String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
        + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
        + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
        + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
        + "海淀区万柳=270,官园=268,通州=315";
    Map<String, Integer> map = new HashMap<String, Integer>();
    String[] arr = pm25.split(",=");
    for (int i = 0; i < arr.length; i += 2) {
        if (!map.containsKey(arr[i]))
            || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
            map.put(arr[i], Integer.parseInt(arr[i + 1]));
        }
    }
    System.out.println(map);

    Set<Map.Entry<String, Integer>> entrys = map.entrySet();
    for (Map.Entry<String, Integer> entry : entrys) {
        System.out.println(entry.getKey() + ":" + entry.getValue());
    }
}
}

```

### • 完整代码

本案例的完整代码如下所示：

```

package day05;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    //... (之前案例的代码, 略)

    /**
     * 使用迭代 key 的方式遍历 map 集合
     */
    @Test
    public void test2() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
            + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
            + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
            + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
            + "海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new HashMap<String, Integer>();
        String[] arr = pm25.split(",=");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        Set<String> keys = map.keySet();
        for (String key : keys) {
            System.out.println(key + ":" + map.get(key));
        }
    }

    /**
     * 使用迭代 Entry 的方式遍历 map 集合
     */
    @Test
    public void test3() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
            + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
            + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
            + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
            + "海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new HashMap<String, Integer>();
        String[] arr = pm25.split(",=");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        Set<Map.Entry<String, Integer>> entrys = map.entrySet();
        for (Map.Entry<String, Integer> entry : entrys) {
            System.out.println(entry.getKey() + ":" + entry.getValue());
        }
    }
}

```

## 4. PM2.5 监控程序——统计并遍历各点 PM2.5 最大值（要求顺序）

- 问题

在“PM2.5 监控程序——遍历各点 PM2.5 最大值”练习的基础上，按照放入 map 中的站点名的顺序遍历出来。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：将 HashMap 改成 LinkedHashMap

首先，构建测试方法 test4、test5；然后，复制 test2 的代码到 test4 中，复制 test3 的代码到 test5 中；最后，将 test4、test5 中构建 map 对象的实现类改成 LinkedHashMap，代码如下所示：

```
package day05;

import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
    /**
     * 按照 key 放入的顺序遍历集合
     */
    @Test
    public void test4() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398," +
                      "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422," +
                      "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299," +
                      "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛" +
                      "=277," +
                      "海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new LinkedHashMap<String, Integer>();
        String[] arr = pm25.split(",=");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        Set<String> keys = map.keySet();
        for (String key : keys) {
            System.out.println(key + ":" + map.get(key));
        }
    }

    /**
     * 按照 key 放入的顺序遍历集合
     */
    @Test
    public void test5() {
```

```

String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
+ "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
+ "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
+ "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
+ "海淀区万柳=270,官园=268,通州=315";
Map<String, Integer> map = new LinkedHashMap<String, Integer>();
String[] arr = pm25.split(",=");
for (int i = 0; i < arr.length; i += 2) {
    if (!map.containsKey(arr[i])
        || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
        map.put(arr[i], Integer.parseInt(arr[i + 1]));
    }
}
Set<Map.Entry<String, Integer>> entrys = map.entrySet();
for (Map.Entry<String, Integer> entry : entrys) {
    System.out.println(entry.getKey() + ":" + entry.getValue());
}
}
}

```

上述代码中的 `LinkedHashMap` 是 `Map` 接口的哈希表和链表实现，具有可预知的迭代顺序。此实现与 `HashMap` 的不同之处在于，`LinkedHashMap` 维护着一个双向循环链表。此链表定义了迭代顺序，该迭代顺序通常就是存放元素的顺序。

### • 完整代码

本案例的完整代码如下所示：

```

package day05;

import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;
import org.junit.Test;

public class TestHashMap {
//... (之前案例的代码, 略)

/**
 * 按照 key 放入的顺序遍历集合
 */
@Test
public void test4() {
    String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
+ "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
+ "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
+ "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
+ "海淀区万柳=270,官园=268,通州=315";
    Map<String, Integer> map = new LinkedHashMap<String, Integer>();
    String[] arr = pm25.split(",=");
    for (int i = 0; i < arr.length; i += 2) {
        if (!map.containsKey(arr[i])
            || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
            map.put(arr[i], Integer.parseInt(arr[i + 1]));
        }
    }
}

```

```

        Set<String> keys = map.keySet();
        for (String key : keys) {
            System.out.println(key + ":" + map.get(key));
        }
    }

    /**
     * 按照 key 放入的顺序遍历集合
     */
    @Test
    public void test5() {
        String pm25 = "农展馆=423,东四=378,丰台花园=406,天坛=322,海淀区万柳=398,"
            + "官园=406,通州=366,昌平镇=248,怀柔镇=306,定陵=231,前门=422,"
            + "永乐店=368,古城=268,昌平镇=423,怀柔镇=267,定陵=377,前门=299,"
            + "永乐店=285,秀水街=277,农展馆=348,东四=356,丰台花园=179,天坛
=277,"
            + "海淀区万柳=270,官园=268,通州=315";
        Map<String, Integer> map = new LinkedHashMap<String, Integer>();
        String[] arr = pm25.split(",=");
        for (int i = 0; i < arr.length; i += 2) {
            if (!map.containsKey(arr[i]))
                || Integer.parseInt(arr[i + 1]) > map.get(arr[i])) {
                map.put(arr[i], Integer.parseInt(arr[i + 1]));
            }
        }
        Set<Map.Entry<String, Integer>> entrys = map.entrySet();
        for (Map.Entry<String, Integer> entry : entrys) {
            System.out.println(entry.getKey() + ":" + entry.getValue());
        }
    }
}

```

## 5. 查看一个文件的大小

- 问题

查看当前工程下 demo 文件夹下 HelloWorld.txt 文件的大小。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：新建类及测试方法

首先，新建类 TestFile，并在该类中新建测试方法 testLength，代码如下所示：

```

package day05;

import java.io.File;
import org.junit.Test;

public class TestFile {
    @Test
    public void testLength() {
    }
}

```

### 步骤二：获取 HelloWorld.txt 文件大小

首先，使用 `File` 类构建表示当前工程下的 `demo` 文件夹下的 `HelloWorld.txt` 文件的对象 `file`；然后，使用 `File` 类的 `length` 方法获取该文件的大小并输出，代码如下所示：

```
package day05;

import java.io.File;
import org.junit.Test;

public class TestFile {
    @Test
    public void testLength() {

        File file = new File("demo" + File.separator + "HelloWorld.txt");
        System.out.println(file + "占用字节量:" + file.length());

    }
}
```

### 步骤三：运行

运行 `testLength` 方法，控制台输出结果如下所示：

```
demo\HelloWorld.txt 占用字节量:0
```

如果当前工程下并不存在 `demo` 文件夹，那就更没有 `HelloWorld.txt` 文件的存在了，因此占用字节长度为 0。现在，在当前工程下创建 `demo` 文件夹，然后，在该文件夹下创建文件 `HelloWorld.txt` 并将该文件内容改为“hello”，再次运行 `testLength` 方法，控制台输出结果如下所示：

```
demo\HelloWorld.txt 占用字节量:5
```

- **完整代码**

本案例的完整代码如下所示：

```
package day05;

import java.io.File;
import org.junit.Test;

public class TestFile {
    @Test
    public void testLength() {
        File file = new File("demo" + File.separator + "HelloWorld.txt");
        System.out.println(file + "占用字节量:" + file.length());
    }
}
```

## 6. 创建一个空文件

- **问题**

在上一案例的基础上，在 demo 文件夹下创建文件 Hello.txt。

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先，在 TestFile 类中新建测试方法 testCreateNewFile，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {

    @Test
    public void testCreateNewFile() throws IOException {
    }

}
```

### 步骤二：创建文件

首先，使用 File 类构建表示当前工程下的 demo 文件夹下的 Hello.txt 文件的对象 file；然后，使用 File 类的 exists 方法判断文件是否存在，如果不存在，使用 File 类的 createNewFile 方法创建该文件，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    @Test
    public void testCreateNewFile() throws IOException {

        File file = new File("demo" + File.separator + "Hello.txt");
        // 若不存在，就创建该文件
        if (!file.exists()) {
            file.createNewFile();
        }

    }
}
```

### 步骤三：运行

运行 testCreateNewFile 方法，会发现在当前工程的 demo 文件夹下多了一个文件 Hello.txt。

- **完整代码**

本案例中，类 TestFile 的完整代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    //... (之前案例的代码, 略)

    @Test
    public void testCreateNewFile() throws IOException {
        File file = new File("demo" + File.separator + "Hello.txt");
        // 若不存在, 就创建该文件
        if (!file.exists()) {
            file.createNewFile();
        }
    }
}
```

## 7. 删除一个文件

- **问题**

将上一案例中创建的 Hello.txt 文件删除。

- **步骤**

实现此案例需要按照如下步骤进行。

**步骤一：构建测试方法**

首先，在 TestFile 类中新建测试方法 testDeleteFile，代码如下所示：

```
package day05;

import java.io.File;
import org.junit.Test;
import java.io.IOException;

public class TestFile {

    @Test
    public void testDeleteFile() {
    }
}
```

**步骤二：删除文件**

首先，使用 File 类构建表示当前工程下的 demo 文件夹下的 Hello.txt 文件的对象 file；然后，使用 File 类的 delete 方法删除该文件，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    @Test
    public void testDeleteFile() {

        File file = new File("demo" + File.separator + "Hello.txt");
        file.delete();

    }
}
```

### 步骤三：运行

运行 `testDeleteFile` 方法，会发现在当前工程的 `demo` 文件夹 `Hello.txt` 文件已经不存在了。

- **完整代码**

本案例中，类 `TestFile` 的完整代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    //... (之前案例的代码，略)

    @Test
    public void testDeleteFile() {
        File file = new File("demo" + File.separator + "Hello.txt");
        file.delete();
    }
}
```

## 8. 创建一个目录

- **问题**

在当前工程下创建 `myDir` 目录。

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先，在 `TestFile` 类中新建测试方法 `testMkdir`，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {

    @Test
    public void testMkdir() {

    }

}
```

## 步骤二：创建目录

首先，使用 `File` 类构建表示当前工程下的 `myDir` 目录的对象 `file`；然后，使用 `File` 类的 `mkdir` 方法创建目录，代码如下所示：

```
package day05;
import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    @Test
    public void testMkdir() {

        File dir = new File("myDir");
        dir.mkdir();

    }
}
```

## 步骤三：运行

运行 `testMkdir` 方法，会发现工程下多了一个 `myDir` 文件夹。

### • 完整代码

本案例中，类 `TestFile` 的完整代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    //... (之前案例的代码，略)

    @Test
    public void testMkdir() {
        File dir = new File("myDir");
        dir.mkdir();
    }
}
```

## 9. 创建一个两级目录

- 问题

在当前工程下，创建 a 目录，然后在 a 目录下创建 b 目录，最后在 b 目录下创建 c 目录。

- 步骤

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先，在 TestFile 类中新建测试方法 testMkdirs，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {

    @Test
    public void testMkdirs() {
    }
}
```

### 步骤二：创建两级目录

首先，使用 File 类构建表示当前工程下的两级目录 a，b，c；然后，使用 File 类的 mkdirs 方法创建两级目录，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    @Test
    public void testMkdirs() {
        File dir = new File("a" + File.separator + "b" + File.separator + "c");
        dir.mkdirs();
    }
}
```

### 步骤三：运行

运行 testMkdirs 方法，会发现工程下多了一个 a 文件夹，并且 a 文件夹下有 b 文件

夹，b 文件夹下有 c 文件夹。

- **完整代码**

本案例中，类 TestFile 的完整代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    //... (之前案例的代码, 略)

    @Test
    public void testMkDirs() {
        File dir = new File("a" + File.separator + "b" + File.separator + "c");
        dir.mkdirs();
    }
}
```

## 10. 删除一个目录

- **问题**

删除一个空目录。

- **步骤**

实现此案例需要按照如下步骤进行。

### 步骤一：构建测试方法

首先，在 TestFile 类中新建测试方法 testDeleteDir，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {

    @Test
    public void testDeleteDir() {
    }
}
```

### 步骤二：删除目录

首先，使用 File 类构建表示当前工程下的 demo 目录的对象 file；然后，使用 File 类的 delete 方法删除该目录，代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    @Test
    public void testDeleteDir() {

        File file = new File("demo");
        file.delete();

    }
}
```

### 步骤三：运行

运行 `testDeleteDir` 方法，会发现 `demo` 文件夹不存在了。此处注意：`demo` 文件夹要为一个空目录。

- **完整代码**

本案例的完整代码如下所示：

```
package day05;

import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    //... (之前案例的代码, 略)

    @Test
    public void testDeleteDir() {
        File file = new File("demo");
        file.delete();
    }
}
```

## 课后作业

- 1. 简述 Map 集合的用处**
- 2. 统计一句话中各个字符的个数**
- 3. 简述 hashCode 方法的意义**
- 4. 按照需求，下面的代码如何优化**

现有 1000 万个对象，需用使用 HashMap 进行存储。使用如下代码构建 HashMap 对象：

```
Map map=new HashMap();
```

上述代码构建的 map，存在效率问题，应如何进行优化。

- 5. 简述 Map 对象的两种遍历方式**
- 6. 统计一句话中重复单词的个数——遍历结果**
- 7. 统计一句话中重复单词的个数——遍历结果（顺序）**
- 8. 使用 commons-io API 查看一个文件的大小**

使用查看当前工程下 demo 文件夹下 HelloWorld.txt 文件的大小。

---

## 9. 使用 commons-io API 查看一个目录的大小

使用 commons-io API , 查看当前工程下 src 目录的大小。

## 10. 使用 commons-io API 创建多级目录

使用 commons-io API , 在当前工程下 , 创建 a 目录 , 然后在 a 目录下创建 b 目录 , 最后在 b 目录下创建 c 目录。

## 11. 使用 Commons-io API 实现删除目录及其内容

删除上一案例中创建的 a 目录及其子目录。