第5章 Java基础类库

Java 语言经过二十多年的发展，已经积累了大量编写好的、可实现各种不同功能的类。Java 语言将这些类打包起来，以类库的形式提供给广大程序员使用。这些由 Java 语言自己所提供的类库统称为 **Java API**(Application Programming Interface)，参见图 5-1。

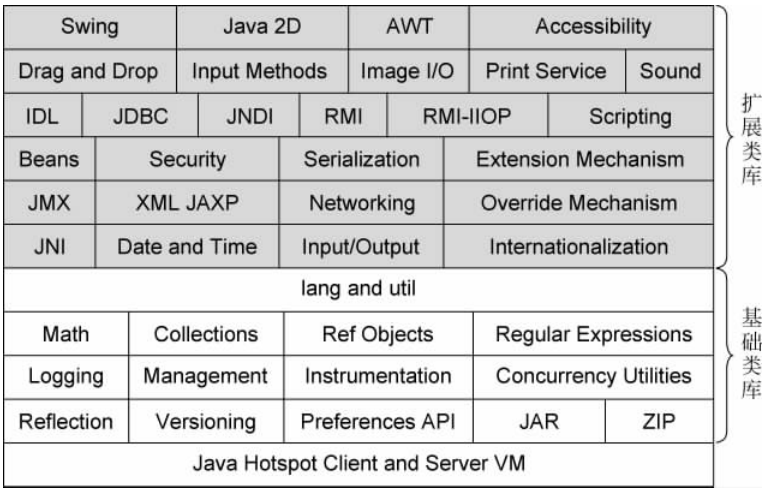


图 5-1　Java API 整体架构(摘自：Java Platform Standard Edition 8 Documentation)

类库相当于已经编写好的**程序零件**。重用类库中的类，相当于用现成的零件来组装程序，这样就能使程序员快速开发出各种功能强大的软件。用已有的程序零件来组装自己的软件产品，这就是程序的**应用开发**。

要想使用 Java API 类库中的类，首先要了解 Java API 中有哪些常用的包，每个包里又有哪些类，这些类的功能是什么，然后再深入学习每个类，了解类里有哪些成员，各成员的功能和用法是什么。通过不断学习和探索，逐步梳理清楚 Java API 类库的整体脉络。

了解 Java API 的最主要途径是学习 Java 官网上发布的 Java API 说明文档。它是 Java API 最权威的学习资料，为程序员提供了详细的帮助信息。学会阅读 Java API 说明文档，这也是 Java 语言程序设计学习的一部分。

本书前 4 章学习了 Java 基础语法和面向对象程序设计方法。后续章节主要学习如何利用 Java API 来进行程序的应用开发，内容包括：

- 学习阅读 Java API 说明文档的基本方法。
- 了解常见的编程应用场景，掌握 Java API 中相关类的使用方法。

• 探索 Java API 类库,循序渐进,从整体上把握 Java API 类库的架构。

在后续章节中,读者将接触到大量具体的程序应用场景和案例。后续章节的学习过程既是 Java 知识积累的过程,同时也是自学能力培养的过程。日积月累,化蛹成蝶,相信读者最终都能够独立开启自己的 Java 探索之旅。

5.1 数学类 Math

Java API 是一个类的海洋。探索 Java API 类库,从数学类 Math 开始(见图 5-2)。

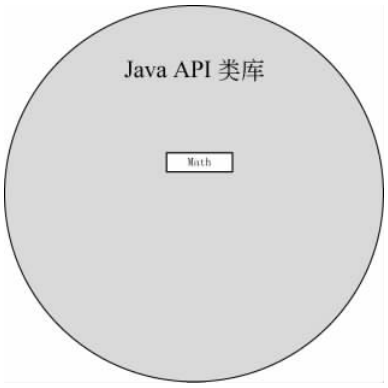


图 5-2 探索 Java API 类库从数学类 Math 开始

了解数学类 Math,从 Java API 说明文档开始。Java API 说明文档对数学类 Math 有如下描述(注:Java API 说明文档是英文的,没有中文翻译)。

The class **Math** contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

这段英文的中文翻译是:类 Math 包含一组数值运算方法,例如指数、对数、平方根和三角函数等初等函数。

Java API 说明文档对每个类都有详细的说明,其中包括类的简要介绍、类包含哪些成员,以及各成员的功能和用法。本书将其中的关键信息摘录下来,以方便读者阅读。首先请读者简单浏览一下数学类 Math 的说明文档。

java.lang.**Math** 类说明文档(摘自:Java Platform Standard Edition 8 Documentation,以下同)

public final class **Math**
extends **Object**

	修 饰 符	类成员(节选)	功 能 说 明
1	static	double **E**	字段,数学常数 e
2	static	double **PI**	字段,数学常数 π
3	static	int **abs**(int a)	求 int 型整数的绝对值
4	static	double **abs**(double a)	求 double 型实数的绝对值
5	static	double **sin**(double a)	求正弦值
6	static	double **cos**(double a)	求余弦值
7	static	double **log**(double a)	求自然对数
8	static	double **log**10(double a)	求以 10 为底的对数
9	static	double **pow**(double a, double b)	求 a 的 b 次方
10	static	double **random**()	返回一个 0～1 的随机数
11	static	double **sqrt**(double a)	求平方根
12	static	double **toDegrees**(double angrad)	将弧度转换为角度
13	static	double **toRadians**(double angdeg)	将角度转换为弧度
		…	

5.1.1 阅读 Java API 类的说明文档

在大致了解了类的功能之后，读者需要仔细阅读类的说明文档。类有 3 个学习要点，分别是类的**全称**、**定义**及其中的各个**成员**。

1. 类的全称

从数学类 Math 的全称 java.lang.Math 中可以读出哪些信息？

- **java**：包名。
- **lang**：java 包下的子包名。
- **Math**：java 包下 lang 子包里的类名。

Java API 有一套自己的命名风格，**包名**、**子包名**和**方法名**都以小写英文字母开头，而**类名**则是以大写英文字母开头的。建议读者在编写自己的 Java 程序时参考使用这种命名风格，这样可以在今后阅读程序时能更容易地区分出包名、类名和方法名。

java.lang 子包是 Java API 中最基础的**语言包**，每个 Java 程序都需要用到这个包。为了方便程序员，Java 编译器会为每个 Java 程序都自动导入这个语言包，这相当于是为程序添加了如下 import 语句：

```
import java.lang.*;                    //导入 Java 语言包里的所有类
```

因此，程序员在使用 Java 语言包里的数学类 Math 时可以直接用类名 Math，不需要写全称 java.lang.Math。

2. 类的定义

Java API 说明文档给出的数学类 Math 定义如下：

```
public final class Math
extends Object
```

从这个数学类 Math 的定义中可以读出哪些信息？

- **public**：Math 是一个公有类，任何 Java 程序都可以使用这个类。
- **final**：Math 是一个最终类，不能被继承、扩展。
- **extends**：Math 继承了一个超类，这个超类的名字叫 **Object**。

3. 类的成员

数学类 Math 包含很多成员，其中有字段 E、PI 等数学常数，还有方法 abs()、sin()等数学函数。阅读说明文档可以清楚地了解各类成员的功能和用法。

1）类成员的功能和用法

举例：int abs(int a)

这是数学类 Math 中的一个方法成员。阅读方法成员说明文档时要关注其功能是什么，方法名、形参、返回值类型分别是什么。至于这个方法是怎么编的，方法体是什么，读者没必要关心。这个方法的功能是什么，怎么用？这才是读者应该关心的。

```
double x = Math.abs( -8 );                    //方法 abs(int a)的功能是求 int 型整数的绝对值
```

2) static 成员

类成员前面的修饰符 static 表示静态成员。数学类 Math 中的成员全部都是静态成员。访问静态成员不需要定义对象,可以直接通过类名访问。例如 Math.**PI**、Math.**abs**(-8)。

3) 类成员的访问权限

类成员的访问权限有 4 种,分别是公有权限(public)、保护权限(protected)、私有权限(private)和默认权限(未指定访问权限)。

在 Java API 中,只有类的公有成员和保护成员是提供给 Java 程序员使用的,而私有成员和默认权限成员则是 Java API 内部使用的。因此,Java API 说明文档不会对私有成员或默认权限成员进行说明,因为这两种成员本来就不是给 Java 程序员使用的,它们是隐藏的。

5.1.2 编写测试程序来学习 Java API 类

在阅读了 Java API 类的说明文档之后,程序员需要编写测试程序来练习类的具体用法。例 5-1 给出一个数学类 Math 的测试程序示例代码。

例 5-1 一个数学类 Math 的测试程序示例代码(MathTest.java)

```
1  public class MathTest {                                          //测试类
2      public static void main(String[] args) {                     //主方法
3          System.out.println( Math.abs( -8 ) );                    //求绝对值
4          System.out.println( Math.sqrt( 16 ) );                   //求平方根
5          System.out.println( Math.sin( Math.PI/2 ) );             //求正弦值
6          System.out.println( Math.toDegrees( Math.PI ) );         //将弧度换算成角度
7          System.out.println( Math.random() );                     //取一个随机数
8          System.out.println( Math.random() );                     //再取一个随机数
9      }
10 }
```

在 Eclipse 集成开发环境中输入并运行例 5-1 的程序,运行结果如图 5-3 所示。将**运行结果**与**程序源代码**进行比对,可以帮助程序员准确理解类中各成员的功能和用法。**强调一下**:编写测试程序,这是学习 Java API 最有效的方法。

图 5-3 例 5-1 程序的运行结果

本节习题

1. 数学类 Math 的全称 java.lang.Math 中不包含(　　)信息。

A. 包名　　B. 子包名

C. 类名　　D. 超类名

2. Java API 说明文档给出的数学类 Math 定义如下：

```
public final class Math
extends Object
```

这个类定义中不包含(　　)信息。

A. 包名　　B. 类名

C. 超类名　　D. 类的访问权限

3. Java API 说明文档给出数学类 Math 的方法成员 sin()定义如下：

```
static double sin(double a)
```

这个方法成员定义中不包含(　　)信息。

A. 方法名　　B. 形参列表

C. 方法体　　D. 返回值类型

4. 如果 Java API 说明文档没有给出类成员的访问权限，则该类成员的权限是(　　)。

A. public　　B. private

C. protected　　D. 默认权限

5. 数学类 Math 中返回随机数的方法是(　　)。

A. sin()　　B. sqrt()

C. random()　　D. Random()

5.2 字符串类

Java API 将字符数组和字符串处理方法封装成字符串类。字符串类的功能更强，使用更方便。常用的字符串类有如下 3 个。

(1) 字符串类 **String**。用于存储和处理字符串常量。

(2) 可变字符串类 **StringBuilder**。适用于存储和处理字符串变量，主要用于文字处理和编辑。

(3) 多线程可变字符串类 **StringBuffer**。与 StringBuilder 功能相同，但可用于多线程程序。其字符串处理算法要复杂一些，运行速度相对也要慢一些。

注：多线程程序将在第 8 章讲解。

5.2.1 字符串类 String

请读者阅读下面的字符串类 String 说明文档。

java.lang.**String** 类说明文档

public final class **String**
extends **Object**
implements **Serializable**, **Comparable**<String>, **CharSequence**

	修饰符	类成员(节选)	功 能 说 明
1		**String**()	无参构造方法
2		**String**(char[] value)	有参构造方法
3		**String**(String original)	拷贝构造方法
4		int **length**()	返回字符串长度
5		char **charAt**(int index)	返回指定下标的字符
6		int **indexOf**(int ch)	返回指定字符 ch 在字符串中第一次出现的下标
7		int **indexOf**(String str)	返回子字符串 str 在字符串中第一次出现的下标
8		int **compareTo**(String anotherString)	与另一个字符串比较大小(按字母顺序)
9		int **compareToIgnoreCase**(String str)	与另一个字符串比较大小(不区分大小写)
10		String **substring**(int beginIndex, int endIndex)	取出指定下标区间里的子字符串
11		String **replace**(CharSequence target, CharSequence replacement)	替换子字符串
12		String[] **split**(String regex)	分割字符串
13		boolean **matches**(String regex)	检查是否与正则表达式匹配
14		String **concat**(String str)	将字符串 str 连接到当前字符串末尾,生成一个新字符串
15		String **toLowerCase**()	将字符串中的大写英文字母转换成小写
16		String **toUpperCase**()	将字符串中的小写英文字母转换成大写
17		String **trim**()	去除字符串两端的空格
18		byte[] **getBytes**()	按字符编码转换成字节数组
19		byte[] **getBytes**(Charset charset)	按指定编码转换成字节数组
20	static	String **format**(String format, Object…args)	生成格式化字符串
21	static	String **valueOf**(int i)	将整数转换成字符串
22	static	String **valueOf**(long l)	将长整数转换成字符串
23	static	String **valueOf**(float f)	将单精度浮点数转换成字符串
24	static	String **valueOf**(double d)	将双精度浮点数转换成字符串
25	static	String **valueOf**(char[] data)	将字符数组转换成字符串
		…	

1. 类的全称

从字符串类 String 的全称 java.lang.String 中可以读出哪些信息?

- **java**:包名。
- **lang**:java 包下的子包名。

- **String**：java 包下 lang 子包里的类名。

对比字符串类 String 和 5.1 节的数学类 Math，可以发现这两个类是在同一个包里的，即 java.lang 语言包。这个包存放了 Java API 中最基础的类和接口。

2. 类的定义

Java API 说明文档给出的字符串类 String 定义如下：

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

从这个类定义中可以读出哪些信息？

- **public**：String 是一个公有类，任何 Java 程序都可以使用这个类。
- **final**：String 是一个最终类，不能被继承、扩展。
- **extends**：String 继承了一个超类，这个超类的名字叫 **Object**。
- **implements**：String 类还实现了 3 个接口，分别是 **Serializable**（可序列化的）、**Comparable**（可比较的）、**CharSequence**（字符序列）。

对比字符串类 String 和 5.1 节的数学类 Math，可以发现这两个类继承的是同一个超类，即 Object。但字符串类 String 比数学类 Math 还多实现了 3 个接口。

3. 字符串类 String 的用法

下面简单介绍一些字符串类 String 的基本用法。

1）定义字符串对象

字符串类 String 包含多个构造方法，因此使用该类定义对象时可以有多种不同的初始化形式。例如：

```
String s1 = new String();                  //未初始化
String s2 = new String( "China 中国" );    //用字符串常量进行初始化
String s3 = new String( s2 );              //用已有的字符串对象进行初始化
```

2）操作字符串对象

字符串类 String 有很多字符串处理方法。例如：

```
System.out.println( s2.length() );         //求 s2 的字符串长度：s2 的长度为 7
System.out.println( s2.substring(1, 3) );  //取 s2 的子串：返回 1～3 的子串"hi"(不含 3)
```

3）字符串可以相加

可以使用加号“+”连接两个字符串。例如：

```
String s = s2 + ",你好";                   //连接两个字符串，并赋值给 s
s += ",世界";                              //在 s 后面再连接一个字符串",世界"
```

4）修改字符串

String 类的字符串对象在内容改变时总是会生成一个新的字符串对象。例如：

```
String s = new String ("abc" );            //引用变量 s 指向字符串对象"abc"
```

```
s += "ef";                                    //内存中将会有两个独立的字符串对象"abc"和"abcef"
```

String 类的字符串对象相加,不是在原字符串后面添加内容,而是相加后生成一个的新字符串对象"abcef"。引用变量 s 会改变指向,引用这个新生成的字符串。原字符串"abc"保持不变。字符串类 String 用于存储字符串常量,因此也称为不可变字符串类。

5) 比较两个字符串的大小

可以用方法 compareTo()来比较两个字符串的大小。例如:

```
String s = new String("cd");
System.out.println( s.compareTo("ab") );    //方法 compareTo()返回一个正数,因为"cd"大于"ab"
System.out.println( s.compareTo("cd") );    //方法 compareTo()返回 0,因为"cd"和"cd"相等
System.out.println( s.compareTo("ef") );    //方法 compareTo()返回一个负数,因为"cd"小于"ef"
```

6) 静态方法 format()

字符串类 String 提供了一个生成格式化字符串的静态方法 format(),其用法非常像 C 语言里的 printf()或 sprintf()。例如:

```
int x = 5;  double y = 16.8;
String s = String.format("x = %d, y = %5.2f", x, y);  //生成格式化字符串
System.out.println( s );                    //显示字符串 s,显示结果: x = 5, y = 16.80
```

4. 字符串类 String 的测试程序

例 5-2 给出一个字符串类 String 的测试程序示例代码。

例 5-2 一个字符串类 String 的测试程序示例代码(StringTest.java)

```
1  public class StringTest {                                    //测试类
2      public static void main(String[] args) {                 //主方法
3          int x = 5;  double y = 16.8;
4          String s = String.format("x = %d, y = %5.2f", x, y); //格式化字符串
5          System.out.println( s );                             //显示字符串 s
6          //下面演示字符串对象的定义与处理
7          String s1 = new String();                            //先定义 3 个字符串对象
8          String s2 = new String("Abcd");
9          String s3 = new String("Abcd cde");
10         System.out.println( s1.length() );                   //空字符串的长度为 0
11         System.out.println( s2.length() );                   //s2 的长度为 4
12         System.out.println( s2.toUpperCase() );              //返回一个大写的字符串
13         System.out.println( s3.indexOf("cd") );              //返回 cd 第一次出现的位置
14         System.out.println( s3.substring(1, 3) );            //返回 1～3 的子字符串
15         System.out.println( s3.concat("fg") );               //连接: s3 + "fg"
16         System.out.println( s3 + "fg" );                     //连接: s3 + "fg"
17 } }
```

在 Eclipse 集成开发环境中运行例 5-2 的程序,运行结果如图 5-4 所示。

5. 接口 Comparable

Comparable 是 Java API 定义的一个**可比较大小的**接口。

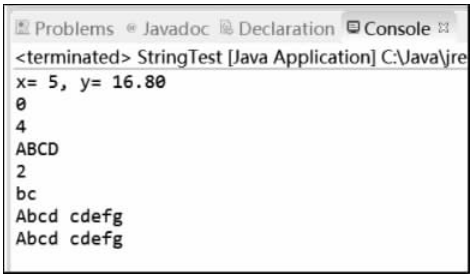


图 5-4 例 5-2 程序的运行结果

```
public interface Comparable<T> {
    int compareTo(T o);                    //比较两个对象大小的抽象方法
}
```

字符串类 String 实现了这个接口，具体实现了比较两个字符串大小的算法，因此可以用方法 compareTo()来比较字符串的大小。方法 compareTo()返回一个整数值，用负数、零、正数分别表示小于、等于和大于。**注**：<T>是泛型编程，将在 5.7 节讲解。

任何一个类都可以实现 Comparable 接口，然后编写 compareTo()方法，具体实现比较两个对象大小的算法。

5.2.2 可变字符串类 StringBuilder

字符串类 String 用于存储字符串常量。可变字符串类 StringBuilder 可存储字符串变量。StringBuilder 类中的字符串可以追加(append)、插入(insert)、替换(replace)和删除(delete)，这些修改属于“原地修改”。

StringBuilder 类有**长度**(length)和**容量**(capacity)两个概念。长度是指实际存储的字符个数，而容量则是指最多能存储的字符个数。当要保存的字符串长度大于容量时，StringBuilder 类将自动增加容量(即分配更多的内存)。StringBuilder 类相当于可变长的字符数组。请读者阅读下面的可变字符串类 StringBuilder 说明文档。

java.lang.**StringBuilder** 类说明文档

public final class **StringBuilder**
extends **Object**
implements **Serializable**, **CharSequence**

	修饰符	类成员(节选)	功 能 说 明
1		**StringBuilder**()	无参构造方法
2		**StringBuilder**(String str)	有参构造方法
3		**StringBuilder**(int capacity)	有参构造方法
4		StringBuilder **append**(CharSequence s)	追加一个字符串
5		StringBuilder **append**(char[] str)	追加一个字符数组
6		StringBuilder **append**(int i)	追加一个整数(转换成字符串)
7		StringBuilder **append**(double d)	追加一个实数(转换成字符串)
8		StringBuilder **insert**(int dstOffset,CharSequence s)	插入一个字符串

续表

	修饰符	类成员(节选)	功 能 说 明
9		StringBuilder **insert**(int offset, int i)	插入一个整数(转换成字符串)
10		StringBuilder **insert**(int offset, double d)	插入一个实数(转换成字符串)
11		StringBuilder **replace**(int start, int end,String str)	替换子字符串
12		StringBuilder **delete**(int start, int end)	删除子字符串
13		int **indexOf**(String str)	查找子字符串
14		String **substring**(int start, int end)	返回子字符串
15		int **length**()	返回字符串长度
16		int **capacity**()	返回字符容量
		…	

例 5-3 给出一个可变字符串类 StringBuilder 的测试程序示例代码。

例 5-3 一个可变字符串类 StringBuilder 的测试程序示例代码(SBuilderTest.java)

```
1  public class SBuilderTest {                          //测试类
2      public static void main(String[] args) {         //主方法
3          String Builder s = new StringBuilder( 100 );//字符容量为 100 个字符(单线程)
4          //String Buffer s = new StringBuffer( 100 ); //类 StringBuffer 可支持多线程
5          s.append("helloChina");                       //追加字符
6          System.out.println( s.toString() );           //显示所返回字符串内容
7          System.out.println( s.length() );             //显示字符串长度
8          System.out.println( s.capacity() );           //显示字符容量
9          s.setCharAt(0, 'H');                          //设定指定位置的字符
10         System.out.println( s );                      //可以省略".toString()"
11         s.replace(5, 10, "中国");                     //将 5～10 的字符替换成"中国"
12         System.out.println( s );
13         System.out.println( s.length() );             //显示新字符串长度
14         s.insert(5, ',');  System.out.println( s );  //插入一个字符",",然后显示内容
15         s.delete(5, 8);  System.out.println( s );    //删除 5～8 的字符,然后显示内容
16 }  }
```

在 Eclipse 集成开发环境中运行例 5-3 的程序,运行结果如图 5-5 所示。

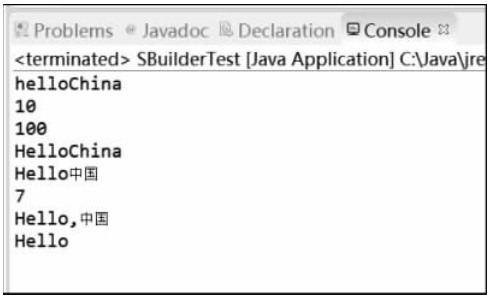


图 5-5 例 5-3 程序的运行结果

本节习题

1. 下面的类( )不是 Java API 中的字符串类。
   A. String B. StringBuilder C. StringBuffer D. Character
2. 字符串类 String 中取出指定位置字符的方法是( )。
   A. charAt() B. getBytes() C. substring() D. valueOf()
3. 字符串类 String 中取出某个位置区间内子字符串的方法是( )。
   A. charAt() B. getBytes() C. substring() D. valueOf()
4. 字符串类 String 中生成格式化字符串的静态方法是( )。
   A. print() B. println() C. compareTo() D. format()
5. 可变字符串类 StringBuilder 中替换某个位置区间内子字符串的方法是( )。
   A. append() B. insert() C. replace() D. delete()

5.3 基本数据类型的包装类

Java 语言预定义了 8 种基本数据类型。Java API 又将这 8 种基本数据类型以类的语法形式分别包装起来,定义了 8 个类。这 8 个类称为基本数据类型的**包装类**(wrapper class),见表 5-1。

表 5-1 基本数据类型及其包装类

基本数据类型	byte	short	int	long	float	double	char	boolean
包装类	Byte	Short	Integer	Long	Float	Double	Character	Boolean

包装类具有与基本数据类型相关的常量和处理方法。常量主要包括基本数据类型的最大值和最小值、占用字节数等。处理方法主要包括比较大小、类型转换等。表 5-1 中 8 个包装类的用法基本相同,本节以整数类 Integer 为例进行讲解。

1. 整数类 Integer

请读者阅读下面的整数类 Integer 说明文档。

java.lang.**Integer** 类说明文档

public final class **Integer**
extends **Number**
implements **Comparable**<Integer>

	修饰符	类成员(节选)	功能说明
1	static	int **BYTES**	字段,int 型占用字节数
2	static	int **MAX_VALUE**	字段,int 型的最大值
3	static	int **MIN_VALUE**	字段,int 型的最小值
4		**Integer**(int value)	构造方法

续表

	修饰符	类成员(节选)	功 能 说 明
5		**Integer**(String s)	构造方法
6		int **compareTo**(Integer anotherInteger)	与另一个 Integer 对象比较大小
7		int **intValue**()	将 Interger 转换成 int
8		String **toString**()	将 Interger 转换成 String
9	static	int **parseInt**(String s)	将字符串转换成 int
10	static	Integer **valueOf**(int i)	将 int 转换成 Integer
11	static	String **toString**(int i)	将 int 转换成字符串
		…	

例 5-4 给出一个整数类 Integer 的测试程序示例代码。

例 5-4 一个整数类 Integer 的测试程序示例代码(WrapperTest.java)

```
1  public class WrapperTest {                                          //测试类
2      public static void main(String[] args) {                        //主方法
1          System.out.println(Integer.BYTES);                          //int 型的字节数
4          System.out.println(Integer.MIN_VALUE);                      //int 型的最小值
5          System.out.println(Integer.MAX_VALUE);                      //int 型的最大值
6          //下面演示比较两个 Integer 对象的大小
7          Integer iObj1 = new Integer(20);                            //初始化为 20
8          Integer iObj2 = new Integer("30");                          //初始化为 30
9          System.out.println( iObj1 > iObj2 );                        //比较大小
10         System.out.println( iObj1.compareTo(iObj2) );               //比较大小
11         //下面演示 Integer 与其他类型之间的转换
12         int i = iObj2.intValue();  System.out.println( i );         //将 Interger 转换成 int
13         Integer iObj3 = Integer.valueOf( i + 10 );                  //将 int 转换成 Integer
14         System.out.println( iObj3.toString() );                     //将 Integer 转换成 String
15         i = Integer.parseInt("50");  System.out.println( i );       //将字符串转换成 int
16         System.out.println( Integer.toString(i) );                  //将 int 转换成字符串
17 } }
```

在 Eclipse 集成开发环境中运行例 5-4 的程序,运行结果如图 5-6 所示。

图 5-6 例 5-4 程序的运行结果

2. 数值类 Number

从整数类 Integer 的说明文档可以看出，整数类 Integer 继承了一个超类，即数值类 Number。实际上，表 5-1 的 8 个包装类中除了 Character 和 Boolean，剩下的 6 个都是数值类 Number 的子类。阅读数值类 Number 的说明文档发现，这个类也是从超类 Object 继承而来的。

java.lang.**Number** 类说明文档

public abstract class **Number**
extends **Object**
implements **Serializable**

	修饰符	类成员(节选)	功 能 说 明
1		**Number**()	构造方法
2		byte **byteValue**()	转换成 byte 类型
3		short **shortValue**()	转换成 short 类型
4		int **intValue**()	转换成 int 类型
5		long **longValue**()	转换成 long 类型
6		float **floatValue**()	转换成 float 类型
7		double **doubleValue**()	转换成 double 类型
…			

本节习题

1. 下面的类(　　)不是 Java API 中的基本数据类型包装类。
   A. Byte　　B. Int　　C. Float　　D. Double
2. 字符类型 char 的包装类是(　　)。
   A. Char　　B. String　　C. Character　　D. character
3. 整数类 Integer 中将字符串转成 int 的方法是(　　)。
   A. intValue()　　B. toString()　　C. parseInt()　　D. valueOf()
4. 整数类 Integer 中将 int 转成字符串的方法是(　　)。
   A. toString()　　B. toString(int i)　　C. parseInt()　　D. valueOf()
5. 下面的类(　　)不是数值类 Number 的子类。
   A. Byte　　B. Boolean　　C. Float　　D. Double

5.4 Java 语言的根类 Object

在学习数学类 Math、字符串类 String、可变字符串类 StringBuilder、基本数据类型包装类和数值类 Number 的过程中，可发现这些类都直接或间接继承了同一个超类，这就是 Object 类。Object 到底是个什么类？

Object 类被称为**对象类**，它是 Java 语言的根类(root class)。

5.4.1 对象类 Object

所有 Java 语言中的类都直接或间接继承了对象类 Object。

- 如果定义类时没有继承超类,则默认继承 Object,这属于直接继承了 Object 类。
- 如果定义类时继承了某个超类,则属于间接继承 Object,因为所继承的超类也一定直接或间接继承了 Object 类。

Object 类可以使用的成员(public 或 protected 权限)都是方法,总共有 12 个。所有 Java 类都继承了这 12 个方法,定义类时可以重写这些方法。请读者仔细阅读下面的对象类 Object 说明文档。

java.lang.**Object** 类说明文档

public class **Object**

	修饰符	类成员(可以使用的全部成员)	功 能 说 明
1		**Object**()	构造方法
2		String **toString**()	将对象转换成字符串
3		Class<?> **getClass**()	取得当前运行类的对象
4		boolean **equals**(Object obj)	与另一个对象比较其内容是否相同
5		int **hashCode**()	将对象映射成一个哈希码(int 型整数)
6	protected	void **finalize**()	JVM 回收对象前自动调用该方法,其作用相当于 C++里的析构方法
7	protected	Object **clone**()	克隆一个当前对象并返回其引用
8		void **wait**()	当前线程进入阻塞状态,用于多线程编程
9		void **wait**(long timeout)	
10		void **wait**(long timeout, int nanos)	
11		void **notify**()	唤醒阻塞状态的线程,用于多线程编程
12		void **notifyAll**()	

例 5-5 给出一个对象类 Object 的测试程序示例代码。

例 5-5 一个对象类 Object 的测试程序示例代码(ObjectTest.java)

```
1  public class ObjectTest {                          //测试类
2      public static void main(String[] args) {       //主方法
3          Object obj = new Object();
4          System.out.println( obj );                 //显示引用变量: 类名@地址
5          System.out.println( obj.toString() );      //转换成字符串
6          Class<?> c = obj.getClass();               //取得对象 obj 的运行类对象 c
7          System.out.println( c.getName() );         //取得运行类对象的类名
8          System.out.println( obj.getClass().getName() );    //直接取得对象 obj 的类名
9          System.out.println();
10         //下面演示 4.1.1 节中的 Clock 类,该类直接继承了 Object 类
11         Clock cObj = new Clock(8, 30, 15);
12         System.out.println( cObj );                //显示引用变量
13         System.out.println( cObj.toString() );     //转成字符串,可以重写 toString()方法
14         System.out.println( cObj.getClass().getName() );  //取得对象 cObj 的类名
15 } }
```

在 Eclipse 集成开发环境中运行例 5-5 的程序，运行结果如图 5-7 所示。

图 5-7 例 5-5 程序的运行结果

5.4.2 重写对象类 Object 的方法

所有 Java 语言(包括 Java API)中的类都直接或间接继承了对象类 Object。定义类时可以重写从 Object 继承来的方法。**请注意**，重写时，方法签名要与 Object 类里的方法签名完全相同。

(1) 重写方法 **toString**()，将对象转成可以理解的字符串。当显示对象或对象与字符串相加时，会自动调用 toString()方法将对象转成字符串。

(2) 重写方法 **hashCode**()，将对象映射成一个 int 型整数，今后可用于快速比较两个对象的内容是否相等。

(3) 重写方法 **equals**()，比较两个对象的内容是否相等。**注**：关系运算符“==”只能比较两个引用是否相等，即是否引用了同一个对象。

(4) 重写方法 **clone**()，创建一个和当前对象内容一样的新对象(即克隆)，并返回其引用。**注**：重写方法 clone()的类需实现可克隆的接口 Cloneable，否则 clone()方法没有激活，不能使用。

(5) 重写方法 **finalize**()，完成对象回收之前的善后工作，例如将对象数据保存到硬盘文件。Java 虚拟机在回收对象前会自动调用其所属类的 finalize()方法。**注**：finalize()方法的语法作用相当于 C++语言里的析构方法，Java 语言没有析构方法。

例 5-6 给出一个重写 Object 方法的新钟表类 Clock 及其测试类的示例代码。**注**：原钟表类 Clock 请参见 4.1.1 节中的例 4-1。

例 5-6 一个重写 Object 方法的新钟表类 Clock 及其测试类的示例代码

```
1  class Clock implements Cloneable {  //自动继承 Object 类,实现接口 Cloneable(Clock.java)
2      //此处省略例 4-1 中类 Clock 已有的代码,下面演示重写从 Object 类继承来的方法
3      public String toString()        //重写方法 toString()
4      {  return String.format("Clock@%d:%d:%d", hour, minute, second);  }
5      public int hashCode()           //重写方法 hashCode()
6      {  return second;  }            //生成哈希码:简单地将秒数作为钟表对象的哈希码
7      public boolean equals(Object obj) {//重写方法 equals()
8          if ((obj instanceof Clock) == false)  return false;     //类型不同,则直接返回
9          Clock c = (Clock)obj;       //将 Object 类型转换成 Clock 类型
```

```
        return c.hour == hour && c.minute == minute && c.second == second;
        //比较两个钟表对象的时间,如果时、分、秒都相同则返回 true,否则返回 false
    }
    public Object clone() throws CloneNotSupportedException      //重写方法 clone ()
    {  Clock c = (Clock)super.clone();  return c;  }           //克隆一个钟表对象
    //注: clone ()方法头后面的"throws …"是 Java 语言的异常处理,将在后面讲解
}
```

```
public class ObjectTest {                    //测试类(ObjectTest.java)
    public static void main(String[] args) {//主方法
        Clock cObj = new Clock(8, 30, 15);
        System.out.println( cObj );//显示引用变量
        System.out.println( cObj.toString() );//转换成字符串,使用重写的 toString()
        System.out.println( cObj.getClass().getName() );    //取得对象 cObj 的类名
        //下面演示如何比较两个钟表对象是否相等
        Clock cObj1 = new Clock(8, 30, 15);//新建对象,设置与 cObj 相同的时间
        Clock cObj2 = cObj;                //cObj2 与 cObj 引用同一钟表对象
        System.out.println( cObj1 == cObj );  //检查引用是否相同,即是否引用了同一对象
        System.out.println( cObj2 == cObj );  //检查两个引用是否相同
        System.out.println( cObj1.equals(cObj) );  //检查两个对象的内容(时间)是否相同
        System.out.println( cObj.hashCode() );    //显示 cObj 对象的哈希码
        System.out.println( cObj1.hashCode() );   //显示 cObj1 对象的哈希码
        //下面演示如何克隆一个钟表对象
        try {  //try - catch 是 Java 语言的异常处理,将在后面讲解
            Clock cObj3 = (Clock)cObj.clone();//克隆一个和 cObj 一样的对象
            System.out.println( cObj3.toString() );      //检查克隆对象的内容是否相同
        } catch(CloneNotSupportedException e) { };
}  }
```

在 Eclipse 集成开发环境中运行例 5-6 的测试程序 ObjectTest.java,运行结果如图 5-8 所示。**注**:如果将例 5-6 与例 5-5 放在同一个目录下(即同一个包中),则两个测试类 ObjectTest 会出现重名的问题,可以将其中一个更名为 ObjectTest1。

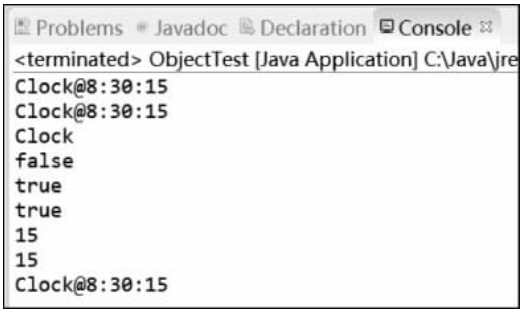


图 5-8 例 5-6 中测试程序 ObjectTest.java 的运行结果

需要特别说明的是,如果重写类的 equals()方法,则应当同时重写 hashCode()方法和 toString()方法。因为如果两个对象内容相等,即 equals()返回 true,则 hashCode()方法所返回的哈希码应当一样,toString()方法所转换出的字符串也应当一样。这就需要程序员在

重写 equals()方法后，必须同步重写 hashCode()方法和 toString()方法。

例 5-6 在定义钟表类 Clock 时还实现了一个可克隆的接口 Cloneable。接口 Cloneable 未定义任何成员，是一个空接口。其定义代码如下：

```
public interface Cloneable;          //接口 Cloneable 的定义代码
```

这种未定义任何成员的空接口称作**标记接口**(marker interface)。定义类时实现某个标记接口，其语法作用是为类激活(或称启用)某种功能。例如，钟表类 Clock 实现接口 Cloneable 的目的是为了激活克隆功能。类只有在激活克隆功能后，调用其中的 clone()方法才能真正实现克隆的功能。

标记接口未定义任何抽象方法，因此实现时也不需要编写任何具体的算法代码。

5.4.3 已探索的 Java API 类库

截至目前，已学习了数学类 Math、字符串类 String、可变字符串类 StringBuilder、基本数据类型包装类、数值类 Number，还有 Java 语言的根类 Object。图 5-9 给出了这些类的继承关系和接口实现示意图。

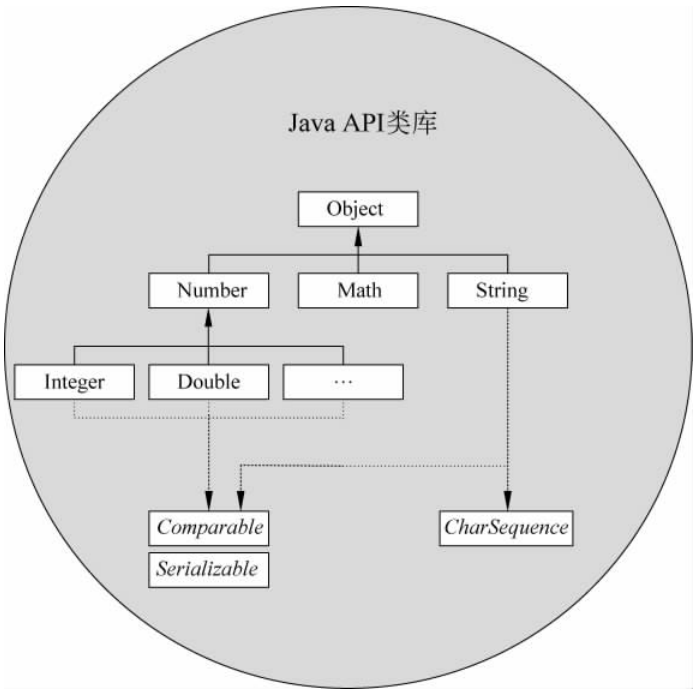


图 5-9 已探索的 Java API 类库

学习 Java API，要了解其中每个类的继承关系和所实现的接口，了解有哪些类族和接口族。这一点很重要，因为只有同一类族或同一接口族中的类才能共用算法代码。换句话说，利用对象的替换与多态机制，处理超类或接口的算法可以用于处理其所有子类的对象。

Java API 大量运用类的继承、接口的实现，以及对象的替换与多态机制，最大程度上实

现了代码的重用或共用。

本节习题

1. 对象类 Object 中将对象转成字符串的方法是(　　)。
   A. toString()　　B. equals()　　C. hashCode()　　D. finalize()
2. 对象类 Object 中比较两个对象内容相等的方法是(　　)。
   A. toString()　　B. equals()　　C. hashCode()　　D. finalize()
3. Java 虚拟机在回收对象之前会自动调用对象的方法成员(　　)。
   A. toString()　　B. equals()　　C. hashCode()　　D. finalize()
4. Java 语言中所有类都包含的成员是(　　)。
   A. toString()　　B. compareTo()　　C. length()　　D. valueOf()
5. 处理 Object 类对象的算法代码不能用于处理(　　)类型的数据。
   A. String　　B. StringBuilder　　C. Integer　　D. int

5.5 系统类 System

请读者阅读下面的系统类 System 说明文档,其中主要包含一些静态字段和静态方法。

java.lang.**System** 类说明文档

public final class **System**
extends **Object**

	修饰符	类成员(节选)	功 能 说 明
1	static final	InputStream **in**	字段,标准输入流对象
2	static final	PrintStream **out**	字段,标准输出流对象
3	static final	PrintStream **err**	字段,标准错误输出流对象
4	static	String **getProperty**(String key)	读取计算机系统属性
5	static	String **setProperty**(String key, String value)	设置计算机系统属性
6	static	void **arraycopy**(Object src, int srcPos, Object dest, int destPos, int length)	复制数组
7	static	long **currentTimeMillis**()	读取系统时间
8	static	void **gc**()	请求回收垃圾
9	static	void **exit**(int status)	退出当前程序的运行
10	static	void **loadLibrary**(String libname)	加载本地库文件
…			

下面简要介绍一下系统类 System 的主要功能。

1. 输入和输出

系统类 System 定义了 3 个静态字段,分别是标准输入流对象 in、标准输出流对象 out

和标准错误输出流对象 err。例如：

```
System.out.println( "Hello, World" );          //在显示器上显示"Hello, World"
```

其中：

- **System**：这是系统类的类名。
- **out**：这是系统类 System 包含的字段成员名。out 是输出流类 PrintStream 的对象。
- **println**：这是对象 out 包含的下级方法成员名，其功能是在显示器上显示信息。

注：System.in、System.out、System.err 相当于 C++语言里的 cin、cout 和 cerr。

2. 读取或设置计算机系统属性

表 5-2 给出了一台计算机系统所具有的主要属性(property)。Java API 分别用字符串形式的键(key)来指代不同的属性。

表 5-2 计算机系统的主要属性

属性的键	说 明
"java.class.path"	Java 类库的搜索路径
"java.home"	Java 运行环境(JRE)的安装目录
"java.version"	JRE 版本号
"os.arch"	操作系统架构
"os.name"	操作系统名称
"os.version"	操作系统版本号
"user.dir"	用户工作目录(当前目录)
"user.home"	用户根目录
"user.name"	用户账号

使用系统类 System 中的静态方法 getProperty()、setProperty()就可以读取或设置当前计算机系统的相关属性。例如：

```
System.out.println( System.getProperty("os.name") );     //读取并显示当前操作系统的名称
```

3. 复制数组

复制数组时，程序员通常需要使用循环语句遍历数组，逐个复制数组里的所有元素。使用系统类 System 中的静态方法 arraycopy()可以很方便地复制数组。例如：

```
int x[ ] = { 1, 2, 3, 4, 5 };
int y[ ] = new int[3];
// y = x;                                          //错误:不能实现复制数组的功能
System.arraycopy(x, 1, y, 0, 3);                   //从 x[1]开始将元素复制给 y[0],共复制 3 个元素
System.out.println(y[0] +"," +y[1] +"," +y[2]); //显示复制后的结果:2,3,4
```

4. 读取系统时间

调用系统类 System 中的静态方法 currentTimeMillis()可以读取当前计算机系统的时

间。这个时间是自 1970 年 1 月 1 日零时起,至系统当前时刻的毫秒数。例如:

```
System.out.println( System.currentTimeMillis() );    //读取并显示当前计算机系统的时间
```

5. 请求回收垃圾

调用系统类 System 中的静态方法 gc(),可以请求 Java 虚拟机的垃圾回收器(garbage collector)回收系统中当前未被引用的对象的内存单元。未被引用的对象,应该是程序已经不再使用的对象,其内存单元可以被收回。例如:

```
System.gc();                                    //请求回收垃圾
```

6. 退出当前程序

调用系统类 System 中的静态方法 exit()可以退出当前程序,同时也停止当前 Java 虚拟机的运行。

```
System.exit( 0 );                               //退出当前程序的运行
```

本节习题

1. 系统类 System 定义了几个输入输出流对象字段,其中不包括(　　)。
   A. in　　B. out　　C. err　　D. log
2. 系统类 System 中读取计算机系统属性的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()
3. 系统类 System 中复制数组的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()
4. 系统类 System 中读取系统时间的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()
5. 系统类 System 中请求 Java 虚拟机回收垃圾的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()

5.6 异常处理

程序中的错误可分为 3 种,分别是**语法**(syntax)错误、**语义**(semantics)错误(或称为**逻辑**错误),以及**运行时**(runtime)错误。针对不同错误,Java 语言具有不同的解决办法,最终保证所开发的程序能够正确、稳定地运行。

Java 语言针对程序运行时错误设计了专门的**异常处理**机制,即 try-catch 机制。Java

API 为异常处理机制提供描述不同异常情况的**异常类**。

5.6.1 3 种不同的程序错误

本节通过具体的程序实例，分别讲解什么是程序中的语法错误、语义错误和运行时错误，以及它们对应的解决办法。

1. 语法错误

例 5-7 给出一个简单的 Java 除法运算程序，其中存在语法错误。

例 5-7 一个简单的 Java 除法运算程序(存在**语法错误**)(SyntaxError.java)

```
 1  import java.util.Scanner;
 2  public class SyntaxError {                          //一个存在语法错误的类
 3      int Div(int n) {                                //方法功能:求 100÷n
 4          int result;
 5          result = 100 / n;                           //求 100÷n
 6          return result;
 7      }
 8
 9      public static void main(String[] args) {        //主方法是一个静态方法
10          int N;
11          Scanner sc = new Scanner( System.in );//创建键盘扫描器对象
12          N = sc.nextInt();                           //键盘输入 N 的值
13          int retValue = Div( N );                    //语法错误:调用非静态方法 Div 计算 100÷N
14          System.out.println( "100÷" + N + "= " + retValue );
15  } }
```

例 5-7 中，代码第 13 行有一个语法错误：静态的 main()方法不能调用非静态的 Div()方法。在 Eclipse 集成开发环境中运行该程序，编译器会提示如下错误信息：

```
Cannot make a static reference to the non - static method Div(int) from the type ErrorSyntax.
```

程序员应按照提示信息，查找错误原因并修改程序。按如下形式将例 5-7 中的方法 Div()定义为静态方法(代码第 3 行)：

```
static  int Div(int n) {              //方法功能:求 100÷n
```

这样就完成了语法错误的修改。再次运行修改后的程序，没有任何语法错误，编译通过。

如果程序员未能严格按照语法规则编写程序，这就属于**语法**错误。编译时，Java 编译器负责检查源程序中的语法错误，如无语法错误则将其编译成字节码程序，否则将提示错误信息。Java 编译器能够帮助程序员检查出所有的语法错误，因此语法错误易于检查，易于修改。

2. 语义错误

例 5-8 给出另一个 Java 除法运算程序，其中存在**语义错误**。代码第 5 行本应是除法运

算,但被错误写成了乘法运算,语法正确但语义错误。

例 5-8 另一个简单的 Java 除法运算程序(存在**语义错误**)(SemanticsError.java)

```
1  import java.util.Scanner;
2  public class SemanticsError {     //一个存在语义错误的类
3      static int Div(int n) {       //方法功能:求 100÷n
4          int result;
5          result = 100 * n;         //语义错误:将除法错误写成了乘法,语法正确但语义错误
6          return result;
7      }
8
9      public static void main(String[] args) {              //主方法
10         int N;
11         Scanner sc = new Scanner( System.in );            //创建键盘扫描器对象
12         N = sc.nextInt();                                 //键盘输入 N 的值
13         int retValue = Div( N );                          //调用方法 Div 计算:100÷N
14         System.out.println( "100÷" + N + "= " + retValue );
15 } }
```

在 Eclipse 集成开发环境中运行例 5-8 的程序,无任何语法错误,可以正常运行。例如,输入 2:

```
2 <回车键>
```

程序将显示如下结果:

```
100÷2= 200
```

这个结果显然是错误的,正确结果应为:

```
100÷2= 50
```

运行测试的结果表明,程序中存在**语义**错误,即程序的算法逻辑有错误。程序员需检查源程序,找出错误原因。本例中,将代码第 5 行的"100 * n"改成"100 / n",这样就完成了对语义错误的修改。

Java 编译器不能帮助程序员发现语义错误。程序员必须通过运行测试,比对程序结果才能发现语义错误。

3. 运行时错误

同样的除法运算,例 5-9 给出一份既没有语法错误,也没有语义错误的 Java 程序代码。

例 5-9 一个无任何语法或语义错误的 Java 除法运算程序(NoError.java)

```
1  import java.util.Scanner;
2  public class NoError {                    //一个无任何语法或语义错误的类
3      static int Div(int n) {               //方法功能:求 100÷n
4          int result;
5          result= 100 / n;                  //求 100÷n
```

```
 6          return result;
 7      }
 8
 9      public static void main(String[ ] args) {       //主方法
10          int N;
11          Scanner sc = new Scanner( System.in );      //创建键盘扫描器对象
12          N = sc.nextInt();                           //键盘输入 N 的值
13          int retValue = Div( N );                    //调用方法 Div 计算:100 ÷ N
14          System.out.println( "100 ÷ " + N + " = " + retValue );
15  }  }
```

在 Eclipse 集成开发环境中运行例 5-9 的程序，没有语法错误，可以正常运行。例如，输入 2：

```
2 <回车键>
```

程序将显示如下结果：

```
100 ÷ 2 = 50
```

运行结果也正确。但再次运行该程序，输入 0：

```
0 <回车键>
```

这时 Eclipse 将提示该程序运行出现了一个算术运算异常，如图 5-10 所示。因为任何数都不能被零除，计算机无法执行“100/0”这样的运算，因此将停止程序的运行。

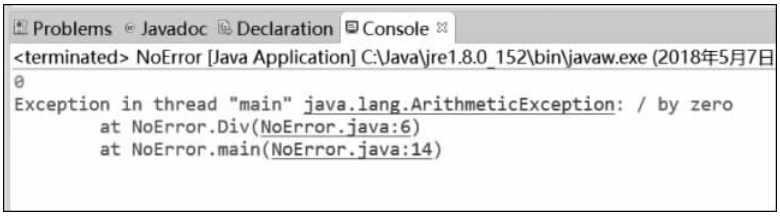


图 5-10 运行例 5-9 程序时出现的“被零除”异常

程序运行时，因运行环境差异或用户操作不当所造成的程序错误统称为**运行时**错误。运行环境存在差异，或用户出现操作不当，这些都称为程序运行时的**异常**(exception)。程序员应当对程序运行时可能出现的异常情况进行处理。

5.6.2 Java 语言的异常处理机制

程序运行过程中常见的异常情况如下。

- 用户操作不当。例如，运行例 5-9 的除法运算程序时输入了 0。
- 输入文件不存在。从文件输入数据，但文件不存在，这会导致文件输入异常。
- 网络连接中断。程序可以通过网络发送、接收数据，网络连接中断将导致网络通信异常。
- 非法访问内存单元。数组越界或空引用会导致程序非法访问内存单元异常。

程序员在编写程序时,应该能够预见到程序运行时可能会发生哪些异常,并在程序中添加异常处理机制,避免因异常情况而导致程序死机或意外中断等严重错误。

1. Java 语言的异常处理机制

Java 语言中,一个**异常处理机制**由如下 3 部分组成。

(1) **发现异常**。程序员应在可能出现异常的程序位置增加检查异常的代码,其目的是及时发现异常。Java 语言使用 **if** 语句来检查异常。Java API 为描述不同的异常情况专门提供了一组**异常类**。

(2) **报告异常**。发现异常后,程序应向 Java 虚拟机报告异常。Java 语言使用 **throw** 语句来报告异常。

(3) **处理异常**。Java 虚拟机在接收到异常报告后,将立即改变程序原来的执行流程,跳转去执行异常处理代码。所谓异常处理,就是在程序算法中增加异常处理流程。没有异常时,程序执行正常算法流程;发现异常时,程序执行异常处理流程。Java 语言使用 **try-catch** 语句来编写捕获和处理异常的代码。

为例 5-9 的除法运算程序添加 Java 异常处理机制,具体代码如例 5-10 所示。假设程序要求键盘输入的数必须为正整数,输入 0 或负数为异常情况。

例 5-10 一个添加了异常处理机制的 Java 除法运算程序(ErrorTryCatch.java)

```
1  import java.util.Scanner;
2  public class ErrorTryCatch {                    //一个添加了异常处理机制的类
3      static int Div(int n) {                     //方法功能:求 100÷n
4          int result;
5          if (n<= 0)                              //检查异常:如果 n<=0,则属于异常情况
6              throw ( new RuntimeException("输入的数值必须为正整数") );  //报告异常
7          result = 100 / n;
8          return result;
9      }
10
11     public static void main(String[] args) {    //主方法
12         int N;
13         Scanner sc = new Scanner( System.in );//创建键盘扫描器对象
14         N = sc.nextInt();                       //键盘输入 N 的值
15         try {                                   //启用 Java 异常处理机制
16             int retValue = Div( N );            //调用方法 Div 计算:100÷N
17             System.out.println( "100÷" + N + "= " + retValue );
18         }
19         catch( RuntimeException e)              //捕获并处理异常
20         {  System.out.println( e.getMessage() );  }
21 }  }
```

在 Eclipse 集成开发环境中运行例 5-10 的程序,输入 0:

```
0 <回车键>
```

这时 Eclipse 不会像例 5-9 那样意外中断程序执行，而是按照程序所设计的异常处理流程向用户显示一条提示信息，如图 5-11 所示。

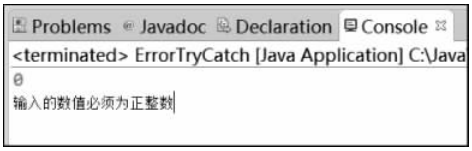


图 5-11 例 5-10 程序能够捕获并处理“被零除”的异常

2. Java API 提供的异常类族

Java API 总结了各种可能的异常情况，然后将它们定义成异常类，其中包括描述异常的相关信息。异常类是一个类族（见图 5-12），其根类是 **Throwable**（可抛出的类）。表 5-3 列出了这个类族中比较常用的异常类，并给出简要的功能说明。

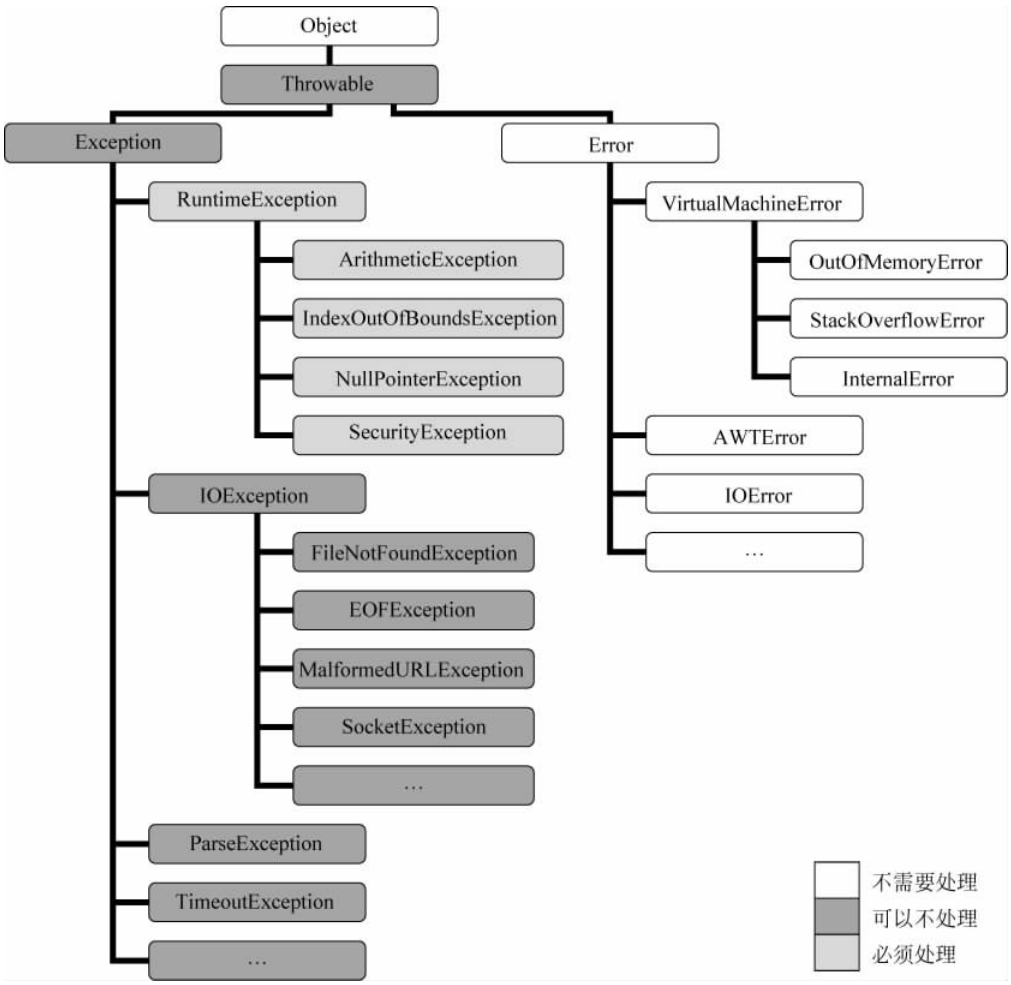


图 5-12 Java API 提供的异常类族

表 5-3 常用的异常类

异 常 类	功 能 说 明
Error	错误类
Exception	异常类
RuntimeException	运行时异常类
ArithmeticException	算术异常类
IndexOutOfBoundsException	下标越界异常类
NullPointerException	空指针(空引用)异常类
SecurityException	安全性异常类
IOException	I/O 读写异常类
FileNotFoundException	未找到文件异常类
EOFException	文件结束异常类
MalformedURLException	URL 格式异常类
SocketException	Socket 连接异常类
ParseException	字符串解析异常类
TimeoutException	超时异常类

请读者阅读下面的异常根类 Throwable 说明文档。

java.lang.**Throwable** 类说明文档

public class **Throwable**
extends **Object**
implements **Serializable**

	修 饰 符	类成员(节选)	功 能 说 明
1		**Throwable**()	构造方法
2		**Throwable**(String message)	构造方法
3		**Throwable**(String message, Throwable cause)	构造方法
4		String **getMessage**()	获取错误信息
5		Throwable **getCause**()	获取错误原因
6		void **printStackTrace**()	打印错误的轨迹
…			

3. throw 语句

Java 语言使用 throw 语句向 Java 虚拟机抛出一个异常对象,其目的是向 Java 虚拟机报告异常。异常对象必须是属于 Throwable 类族中异常类的对象,不能是任何其他类的对象。异常对象中包含了描述异常的相关信息。throw 语句有 3 种常用句型。

1) 基本句型

```
异常类名 eRef = new 异常类名( "异常信息" );     //先创建异常对象
throw  eRef;                                    //然后抛出异常对象
```

2) 简写句型

```
throw  new 异常类名( "异常信息" );              //创建异常对象并立即抛出
```

3）链接句型

```
throw  eRefLast;                    //接力抛出已被捕获的异常对象 eRefLast,形成异常处理链条
```

或

```
throw  new 异常类名( "附加异常信息", eRefLast); //抛出新异常对象,形成异常处理链条
```

计算机执行 throw 语句，会在抛出异常对象后立即改变执行流程，跳转去执行异常处理代码。程序员使用 try-catch 语句来编写捕获和处理异常的程序代码。

4. try-catch 语句

Java 语法：**try-catch 语句**

```
try {
  受保护代码(其中可能会直接或间接抛出异常)
}
catch ( 异常类型 1 引用变量 )
{  异常类型 1 的处理代码 }
catch ( 异常类型 2 引用变量 )
{  异常类型 2 的处理代码 }
…
finally
{  最终的善后处理代码 }
```

语法说明：

- **try-catch-finally** 语句是一个整体，其中包含 try 子句、catch 子句和 finally 子句。try 子句后面至少跟一条 catch 子句，或 finally 子句。finally 子句是可选项。
- **try** 子句：如果预计某个程序代码段在执行时可能发生异常，则程序员可使用 try 子句将该代码段保护起来。Java 虚拟机在执行受保护代码段时将启用异常处理机制，监控代码执行过程中的任何异常报告，包括代码所调用下级方法的异常报告。
- **catch** 子句：catch 子句负责捕获并处理异常，每个 catch 子句只负责一种类型的异常。
  - 若受保护代码段在执行过程中发生异常，抛出了某个异常对象，则 Java 虚拟机会根据异常类型依次匹配 catch 子句。如果异常对象的类型与某个 catch 子句中的异常类型匹配，称异常对象被**捕获**。此时 catch 子句中的引用变量将引用被捕获的异常对象。需要注意的是，超类可以匹配子类，即捕获超类的 catch 子句将会同时捕获到其所有子类的异常，因此通常将捕获超类的 catch 子句放在捕获子类 catch 子句的后面。
  - 如果异常对象被某个 catch 子句捕获，则执行该 catch 子句的处理代码。处理代码负责对异常情况进行处理，例如向用户显示提示信息；也可以使用 throw 语句的链接句型接力抛出异常，形成一个异常处理链条。可以调用异常类的方法 printStackTrace()显示异常处理链条的轨迹。
  - 每个异常最多只会被一个 catch 子句捕获，因此只会有一个 catch 子句的处理代码被执行，其他 catch 子句都不会被执行。

- 如果异常未被任何 catch 子句捕获,Java 虚拟机会自动逐级交由上级方法捕获、处理,直到被上级方法中的某个 catch 子句捕获。如果连最上级的主方法 main()也未能捕获异常,则中止当前程序的执行,并显示相关的异常信息。
- 若受保护代码段在执行过程中未抛出任何异常对象,即没有发生异常,则所有的 catch 子句都不会被执行。

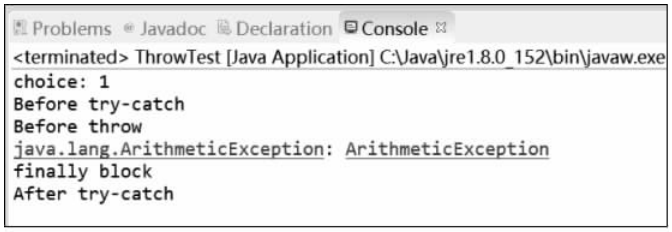
■ **finally** 子句:finally 子句为正常处理流程和异常处理流程提供统一的善后处理。简单地说,不管是否发生了异常,finally 子句中的代码都会被执行。finally 子句通常用于清理程序所占用的资源,例如关闭已打开的文件。

例 5-11 给出了一个 Java 异常处理机制的演示程序。

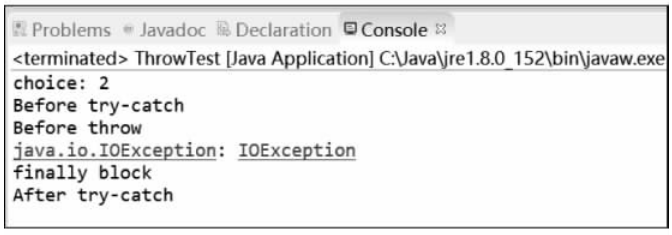
例 5-11 一个 Java 异常处理机制的演示程序(ExceptionTest.java)

```
1  import java.io.IOException;
2  public class ExceptionTest {                          //测试类
3      static void fun(int choice) {     //根据参数 choice 模拟不同的异常,然后进行异常处理
4          System.out.println( "choice: " + choice ); //显示提示信息,用于观察执行流程
5          System.out.println( "Before try - catch" );
6
7           try {
8              System.out.println( "Before throw" );
9              if (choice == 1)                        //1:模拟算术运算异常
10                 throw new ArithmeticException("ArithmeticException");
11             else if (choice == 2)                   //2:模拟输入输出异常
12                 throw new IOException("IOException");
13             System.out.println( "After throw" );
14         }
15          catch( ArithmeticException e)              //捕获并处理算术运算异常
16         {  System.out.println( e.toString() );  }
17          catch( IOException e)                      //捕获并处理输入输出异常
18         {  System.out.println( e.toString() );  }
19          finally                                    //善后处理
20         {  System.out.println( "finally block" );  }
21
22         System.out.println( "After try - catch" );
23     }
24
25     public static void main(String[] args)         //主方法
26     {  fun( 1 );  }          //通过不同实参来模拟不同的异常,例如 fun(1)、fun(2)、fun(0)
27 }
```

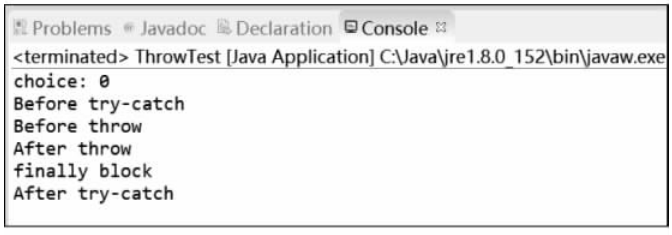
例 5-11 中,主方法 main()在调用子方法 fun()时通过不同实参来模拟不同的异常。在 Eclipse 集成开发环境中运行这个程序,对比屏幕提示信息(见图 5-13)和源代码,观察程序的执行流程,这样可以深入理解 Java 异常处理机制的工作原理。



(a) 主方法调用fun(1)模拟算术运算异常



(b) 主方法调用fun(2)模拟输入输出异常



(c) 主方法调用fun(0)模拟不发生异常的情况

图 5-13 例 5-11 程序的运行结果

5.6.3 Java 异常处理的代码框架

Java 程序通常都存在多级嵌套调用关系。例如，程序的主方法 main()调用子方法 fun1()，fun1()再调用子方法 fun2()，……，最终可能会调用 Java API 中的方法(见图 5-14)。其中，主方法 main()位于最顶层，Java API 则处于最底层。

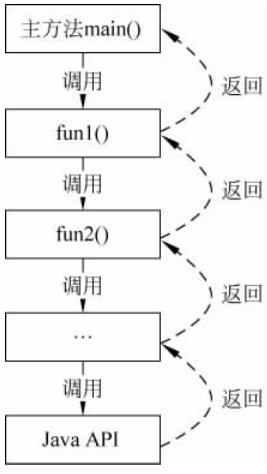


图 5-14 方法的多级嵌套调用

假设图 5-14 中的方法 fun2()在执行过程中可能会发生异常，该如何处理这个异常呢？程序员可以按如下 3 种不同的思路来设计异常处理流程。

1. 由方法 fun2()自己处理

如果由方法 fun2()自己处理所报告的异常，则程序员应当按如下代码框架来定义方法 fun2()。

```
… fun2() {        //方法 fun2()处理自己异常时的代码框架
    …
    try {         //启用异常处理机制
```

```
        …
        if (发现异常)  throw 异常对象;     //报告异常
        …
    }
    catch ( … ) { 异常处理代码 }          //捕获并处理异常
    …
}
```

2. 将异常交由上级方法 fun1()处理

方法 fun2()也可以只报告异常,将异常处理的工作交由其上级方法 fun1()去完成。这时,程序员应当按如下代码框架来定义方法 fun2()和 fun1()。

```
… fun2() {                          //方法 fun2()只报告但不处理异常时的代码框架
    …
    if (发现异常)  throw 异常对象;     //报告异常
    …
}

… fun1() {                          //方法 fun1()处理下级方法 fun2()所报告异常时的代码框架
    …
    try {                           //启用异常处理机制
        …
        fun2();                     //调用方法 fun2(),调用过程中可能会报告异常
        …
    }
    catch ( … ) { 异常处理代码 }     //捕获并处理异常
    …
}
```

下级方法在发现异常时只报告(即抛出异常对象)但不处理,由上级方法统一捕捉、处理所有的异常,这就是 Java 程序中的**多级异常处理**。

多级异常处理可以将分散在不同方法中的异常处理代码剥离出来,集中交由某个上级方法统一进行处理。多级异常处理的好处主要体现在以下两个方面。

- 将原来分散在不同方法中的异常处理代码集中到一起,这样可以减少异常处理中重复的代码。
- 将方法中的异常处理代码剥离出来,让方法专注于正常算法流程,这样可以简化算法设计,优化代码结构。

3. 多级异常处理链条

方法 fun1()在处理完 fun2()所报告的异常之后,可以向更上级的方法继续报告异常。在图 5-14 中,这个更上级的方法就是主方法 main()。这时,程序员应当按如下代码框架来修改 fun1(),并在主方法 main()中添加异常处理代码。

```
… fun1() {     //方法 fun1()在处理下级方法 fun2()报告的异常后,继续向更上级的方法报告异常
    …
    try {                   //启用异常处理机制
```

```
        …
        fun2();              //调用方法 fun2(),调用过程中可能会报告异常
        …
    }
    catch ( … e ) {          //第 1 次捕获异常对象 e
        异常处理代码 1;      //第 1 次处理异常对象 e
        throw  e;            //接力抛出已被捕获的异常对象 e,形成异常处理链条
        //或: throw  new 异常类名( "附加异常信息", e);  //接力抛出新的异常对象
    }
    …
}

… main( … ) {                //主方法 main()处理下级方法 fun1()所报告异常时的代码框架
    …
    try {                    //启用异常处理机制
        …
        fun1();              //调用方法 fun1()
        …
    }
    catch ( … e ) {          //第 2 次捕获异常对象 e
        异常处理代码 2;      //第 2 次处理异常对象 e
    }
    …
}
```

捕获异常对象,处理后再接力抛出,交由更上级的方法继续捕捉、处理,这就形成了一个多级异常处理链条。这么做的原因是,某些异常需要经过不同层级代码多次处理才能完成,每个层级只是整个异常处理流程中的一个环节。

5.6.4 不同性质的异常

综合分析 Java 程序运行过程中可能发生的各种异常,可以按发生原因将它们划分成 3 种不同的性质,分别是系统导致的异常、程序员导致的异常,还有用户导致的异常。针对这 3 种不同性质的异常,Java 语言要求程序员区别对待,分别按不同的方式去处理它们。

1. 3 种不同性质的异常

1) 系统异常

由于 Java 虚拟机或 Java API 自身原因导致的异常称为系统异常。Java API 将描述系统异常的类都定义成 **Error** 类的子类,可统称为系统异常类。例如,描述 Java 虚拟机内存不足的异常类 OutOfMemoryError、描述 Java 虚拟机内部错误的异常类 InternalError 等,参见图 5-12。

程序员无法预见系统异常,也处理不了系统异常。因此 Java 语言在语法上不强制要求程序员必须使用异常处理机制去捕获或处理系统异常。针对系统异常,程序员可以处理,也可以不做任何处理。如果程序运行过程中发生了系统异常,唯一能做的只能是中断程序的执行。

2) 编程异常

因程序员在编程时考虑不周而导致的异常称为编程异常。Java API 将描述编程异常的类都定义成 **RuntimeException** 类的子类,可统称为编程异常类。例如,描述被零除等算术运算错误的异常类 ArithmeticException、描述空引用访问错误的异常类 NullPointerException、描述数组越界错误的异常类 IndexOutOfBoundsException 等,参见图 5-12。

如果程序运行过程中因编程异常导致程序意外中断,这对用户来说是不可理解的,也是不可接受的。Java 语言认为,程序员应当通过周密的设计和完善的测试,完全杜绝程序中的编程异常,否则就是失职。因此对于编程异常,Java 语言在语法上也不强制要求程序员必须使用异常处理机制进行处理,即程序员可以处理,也可以不做任何处理。

3) 用户异常

因操作不当或计算机系统配置不当等用户因素而导致的异常称为用户异常。Java API 将描述用户异常的类都定义成 **Exception** 类下除 RuntimeException 之外的其他子类,它们可统称为用户异常类。例如,描述输入输出错误的异常类 IOException、描述未找到输入文件错误的异常类 FileNotFoundException、描述网路连接错误的异常类 SocketException 等,参见图 5-12。

程序运行过程中,如果发现操作不当等用户异常,Java 程序不应该中断执行,而应该立即捕捉异常并向用户显示错误提示,同时还应保持程序正常运行,让用户能够继续操作程序。为了保证这一点,Java 语言在语法上强制要求程序员必须添加异常处理机制对用户异常进行处理。如果程序员没有对程序中可能发生的用户异常进行捕捉、处理,则程序编译不能通过。

Java 语言将必须被捕捉、处理的用户异常称为**勾选**(checked)异常或受检异常;而将系统异常、编程异常这两种没有被强制要求处理的异常称为**非勾选**(unchecked)异常或非受检异常。

2. 勾选异常的处理

如果一个方法在执行过程中可能抛出**勾选异常**(即因用户因素导致的用户异常),包括其调用下级方法过程中抛出的勾选异常,则该方法必须对勾选异常进行**捕捉或声明**(catch or specify)。

1) 捕捉

捕捉就是在方法体中编写 **try-catch** 语句,捕获并处理本方法或其调用的下级方法所抛出的勾选异常。

假设图 5-14 中的方法 fun1()可能抛出一个 A 类的勾选异常对象 eA,其调用的下级方法 fun2()还可能抛出一个 B 类的勾选异常对象 eB。如果由方法 fun1()负责捕获并处理勾选异常,则程序员应当按如下代码框架来定义方法 fun1()。

```
… fun1() {                              //方法 fun1()负责捕获并处理勾选异常时的代码框架
    …
    try {                               //启用异常处理机制
        …
        if (发现勾选异常 A)   throw eA;//报告异常:抛出一个 A 类的勾选异常对象 eA
        fun2();        //调用方法 fun2(),调用过程中还可能会抛出一个 B 类的勾选异常对象 eB
```

```
        …
    }
    catch ( A  e) {  异常处理代码 1 }      //捕获并处理 A 类的勾选异常
    catch ( B  e) {  异常处理代码 2 }      //捕获并处理 B 类的勾选异常
    …
}
```

2) 声明

方法可以自己捕获并处理所抛出的勾选异常,或者将勾选异常交由上级方法处理。如果方法自己不处理勾选异常,而是将它们提交给上级方法处理,则必须向上级方法**声明**这些勾选异常。这就是 Java 语言对勾选异常所制定的捕捉或声明原则。

声明勾选异常,就是在方法头的后面使用关键字 **throws** 给出**勾选异常列表**,向上级方法列出自己可能会抛出哪些勾选异常。如果采用这种声明方式,则程序员应当按如下代码框架来修改 1)中 fun1()的定义代码。

```
… fun1() throws  A, B {  //方法 fun1()通过声明将勾选异常交由上级方法处理时的代码框架
    …
    if (发现勾选异常 A) throw eA;                  //报告异常:抛出一个 A 类的勾选异常对象 eA
    fun2();              // 调用方法 fun2(),调用过程中还可能会抛出一个 B 类的勾选异常对象 eB
    …
}
```

方法 fun1()所声明的勾选异常 A、B 仍需要被调用它的上级方法(例如图 5-14 中的主方法 main())捕获并处理,否则上级方法的编译就不能通过。

在图 5-14 所示的方法多级嵌套调用关系中,如果其中的某个方法可能抛出勾选异常,则该方法的每个上级方法都需遵循捕捉或声明原则,直至该勾选异常被捕获并处理。Java 虚拟机会按照方法调用的返回顺序逐级向上,查找能够捕获勾选异常的 catch 子句。如果一直到主方法 main()也没找到能够捕获勾选异常的 catch 子句,则程序的编译不能通过。

简单地说,程序员在调用某个声明了勾选异常的方法时,必须对其所声明的勾选异常进行捕捉或继续声明,否则属于语法错误。Java API 中的某些方法就会抛出勾选异常,程序员在阅读 Java API 说明文档时需要关注方法的异常声明。如果方法声明了勾选异常,则调用时必须进行捕捉或继续声明。

5.6.5 自定义异常类

程序员可以定义自己的异常类,这样就能描述自己程序中可能发生的特定异常情况。例如,身份证号由 18 数字组成(最后一位可能是字母),假设用户输入了错误的身份证号,程序员可以定义一个 ID 异常类来描述这种异常。例 5-12 给出了一个描述身份证号异常的 ID 异常类示例代码。

例 5-12 一个描述身份证号异常的 ID 异常类示例代码(MyIDException.java)

```
1  class MyIDException extends Exception {  //ID 异常类:继承 Java API 中的异常类 Exception
2      private String ID = null;            //添加字段:存储错误的身份证号
3      public MyIDException(String msg, String id) {  //构造方法
4          super( msg );                    //调用超类 Exception 的构造方法
5          ID = id;
6      }
```

```
7       public String toString() {          //重写 toString()方法,增加 ID 信息
8           return( ID + ":" + super.toString() );
9   }   }
```

通常,程序员定义异常类时会选择从 Java API 中的异常类 Exception 或运行时异常类 RuntimeException 继承,然后在此基础上扩展。请注意这两者的区别。

(1) 从异常类 **Exception** 继承。所定义出的子类属于勾选异常,必须遵循捕捉或声明原则进行处理。

(2) 从运行时异常类 **RuntimeException** 继承。所定义出的子类是非勾选异常。针对非勾选异常,Java 语言不做强制要求,程序员可以处理,也可以不做任何处理。

本节习题

1. Java 程序中的语法错误主要通过(　　)来进行排查。
   A. Java 编译器　　B. 运行测试
   C. Java 虚拟机　　D. Java 异常处理机制
2. Java 程序中的语义(逻辑)错误主要通过(　　)来进行排查。
   A. Java 编译器　　B. 运行测试
   C. Java 虚拟机　　D. Java 异常处理机制
3. Java 程序中的运行时错误主要通过(　　)来进行排查。
   A. Java 编译器　　B. 运行测试
   C. Java 虚拟机　　D. Java 异常处理机制
4. 下列选项中,(　　)不属于 Java 异常处理机制的范畴。
   A. 发现异常　　B. 报告异常
   C. 处理异常　　D. 异常对象的垃圾回收
5. 下面的异常类中,(　　)属于必须被捕捉或声明的勾选异常。
   A. Error 类及其子类　　B. RuntimeException 类及其子类
   C. IOException 类及其子类　　D. NullPointerException 类
6. 下列抛出异常对象的语句中,错误的是(　　)。
   A. Exception e = new Exception(); throw e;
   B. throw new Exception();
   C. throw new IOException ();
   D. throw new String();
7. 在 try-catch 语句中,不能被省略的子句是(　　)。
   A. try 子句　　B. catch 子句
   C. finally 子句　　D. 以上 3 个子句都能省略
8. 在 try-catch 语句中,有可能不执行的子句是(　　)。
   A. try 子句　　B. catch 子句
   C. finally 子句　　D. 以上 3 个子句都有可能

5.7 泛型与数据集合类

泛型(generics)是从 JDK 1.5 开始引入的一种新特性,其目的是通过**类型参数化**来提高程序代码的重用性。类型参数化就是将类、接口或方法所处理数据的类型抽象成参数,这样可以定义出**泛型类**、**泛型接口**或**泛型方法**。同一个泛型类、泛型接口或泛型方法可以处理多种不同类型的数据,称其**代码**可以被不同数据类型**重用**。

5.7.1 类型参数化

本节通过一个具体的程序实例讲解什么是类型参数化,以及如何利用类型参数化来提高程序代码的重用性。例 5-13 给出了两个分别存放 Integer 型数据和 Double 型数据的集合类示例代码。

例 5-13 两个分别存放 Integer 型数据和 Double 型数据的集合类示例代码

```
1  class IntegerSet {                          //Integer 型集合类
2        public Integer set[];                 //用于存放 Integer 型数据的数组
3        public IntegerSet( Integer p[])       //构造方法
4        {  set = p;  }
5        public void show() {                  //显示数据集合中的元素
6            for (int n = 0; n < set.length; n++)
7                  System.out.print( set[n] +" " );
8            System.out.println();
9  }  }
```

```
1  class DoubleSet {                           //Double 型集合类
2        public Double set[];                  //用于存放 Double 型数据的数组
3        public DoubleSet( Double p[])         //构造方法
4        {  set = p;  }
5        public void show() {                  //显示数据集合中的元素
6            for (int n = 0; n < set.length; n++)
7                  System.out.print( set[n] +" " );
8            System.out.println();
9  }  }
```

可以使用例 5-13 中的两个类分别定义出 Integer 型集合对象和 Double 型集合对象。例如:

```
//定义一个 Integer 型集合对象
Integer ia[] = { 10, 20, 30 };
IntegerSet is = new IntegerSet( ia );
is.show();              //显示集合对象 is 中的元素,显示结果:10 20 30
//定义一个 Double 型集合对象
Double da[] = { 10.5, 20.5, 30.5 };
DoubleSet ds = new DoubleSet( da );
ds.show();              //显示集合对象 ds 中的元素,显示结果:10.5 20.5 30.5
```

仔细分析例 5-13 中的 Integer 型集合类和 Double 型集合类,可以看出这两个类的功能

完全相同,唯一不同的是集合元素的数据类型不一样,一个是 Integer 型,另一个是 Double 型。将这两个类合并成一个类,就可以有效降低程序的编码工作量。

Java 语言可以将 Integer、Double 等具体数据类型抽象成一个类型参数(称为**类型形参**),这就是**类型参数化**。假设将类型形参命名为 T,利用类型形参 T 可以将 Integer 型集合类和 Double 型集合类合并成一个 T 类型的集合类,这就是一个**泛型类**。这里的类型形参 T 可以指代任意一种具体的数据类型,或者说类型形参 T 是一种通用数据类型(称为**泛型**)。例 5-14 给出一个 T 类型的泛型集合类示例代码。

例 5-14 一个 T 类型的泛型集合类示例代码

```
1  class GenericSet < T > {          //泛型集合类:类型形参 T 可以指代任意一种具体的数据类型
2        public T set[];             //用于存放 T 类型数据的数组
3        public GenericSet( T p[]) //构造方法
4        {  set = p;  }
5        public void show() {        //显示数据集合中的元素
6            for (int n = 0; n < set.length; n++)
7                  System.out.print( set[n] +" " );
8            System.out.println();
9  }  }
```

使用例 5-14 中的泛型集合类 GenericSet < **T** >时,需要明确给出类型形参 T 所指代的具体数据类型(称为**类型实参**)。不同类型实参表示不同类型的集合类。例如:

- GenericSet < **Integer** >表示 Integer 类型的集合类,类型实参为 Integer。
- GenericSet < **Double** >表示 Double 类型的集合类,类型实参为 Double。

可以使用这两个类分别定义出 Integer 型集合对象或 Double 型集合对象。例如:

```
//定义一个 Integer 型集合对象
Integer ia[] = { 10, 20, 30 };
GenericSet < Integer > is = newGenericSet < Integer >( ia );  //类型实参为 Integer
//或:GenericSet < Integer > is = new GenericSet <>( ia ); //可省略第 2 个类型实参 Integer
is.show();                                                  //显示集合对象 is 中的元素:10 20 30
//定义一个 Double 型集合对象
Double da[] = { 10.5, 20.5, 30.5 };
GenericSet < Double > ds = new GenericSet < Double >( da );      //类型实参为 Double
ds.show();                                                 //显示集合对象 ds 中的元素:10.5 20.5 30.5
```

还可以使用例 5-14 中的泛型类 GenericSet < **T** >定义出其他类型的集合对象。例如:

```
//定义一个 Short 型集合对象
Short sa[] = { 10, 20, 30 };
GenericSet < Short > ss = new GenericSet < Short >( sa );       //类型实参为 Short
ss.show();                                                      //显示集合对象 ss 中的元素:10 20 30
//定义一个 Float 型集合对象
Float fa[] = { 10.5f, 20.5f, 30.5f };
GenericSet < Float > fs = new GenericSet < Float >( fa );       //类型实参为 Float 型
fs.show();                                                 //显示集合对象 fs 中的元素:10.5 20.5 30.5
```

使用泛型类 GenericSet < **T** >可以定义出各种不同数据类型的集合类。换句话说,泛型

类 GenericSet < **T** >是一种能够被重用的代码，它可以被不同的数据类型重用。利用类型参数化，可以有效提高程序代码的重用性。

5.7.2 泛型编程

通过 5.7.1 节的集合类程序例子，读者已经直观地了解了什么是类型参数化，什么是泛型类以及泛型类的使用方法。使用泛型类可以定义出各种不同数据类型的具体类。

Java 语言中，带类型参数的类称为**泛型类**。同理，带类型参数的接口称为**泛型接口**，带类型参数的方法称为**泛型方法**。在了解了泛型的基本概念之后，读者就可以使用 Java API 中的泛型类、泛型接口或泛型方法来编写程序了。例如，读者可以使用 Java API 提供的数据集合类来编写复杂的数据集合处理程序。

5.7.1 节讲解的是如何**使用**别人编写的泛型类。本节所要讲解的是如何**编写**自己的泛型类，即**泛型编程**。**注**：泛型编程是一种高级 Java 编程技术，内容比较难。初学者可跳过本节，这不会影响后续内容的学习。

1. 泛型类或泛型接口

这里给出定义泛型类或泛型接口的 Java 语法，读者可对照 5.7.1 节所讨论的泛型集合类 GenericSet < T >来理解泛型语法的应用语境。

Java 语法：定义**泛型类**或**泛型接口**

```
class 泛型类名<类型形参列表> { 类成员 }
interface 泛型接口名<类型形参列表> { 接口成员 }
```

语法说明：

- 定义泛型类、泛型接口时，需在类名或接口名后面用一对尖括号“< >”给出**类型形参列表**。
- **类型形参**是一种表示数据类型的参数。多个类型形参之间用逗号“,”隔开，例如 < T >、< T1, T2 >、< K, V >等。类型参数名需符合标识符的命名规则，习惯上用 T、E、K、N、V 等表示。
- 泛型类(或泛型接口)定义代码的其余部分与普通类(或普通接口)一样。所不同的是，类型形参就像是一种新的数据类型，可以用来定义字段成员或方法成员中的形参、局部变量或返回值类型。
- 使用泛型类(或泛型接口)时，需明确给出类型形参所指代的**类型实参**(即某一种具体的数据类型)。指定了类型实参的泛型类(或泛型接口)称为**具体类**(或**具体接口**)。**请注意**，类型实参只能是**引用数据类型**(例如类、接口、数组等)，不能是基本数据类型(例如 int、double 等)。
- 如果对类型形参不做限定(例如< **T** >)，则类型形参可以指代任意一种引用数据类型，即使用泛型类(或泛型接口)时的类型实参可以是任意一种引用数据类型。如果希望将类型实参限定在某个**类族**或**接口族**范围内，则需要按如下格式来定义类型形参。

```
T extends 超类名      //类型形参 T 可以指代某个超类及其所有子类
T extends 接口名      //类型形参 T 可以指代任意实现了该接口的类,或从其扩展出的子接口
```

- 使用泛型类(或泛型接口)可以定义出不同数据类型的具体类(或具体接口)。换句话说,泛型类(或泛型接口)是一种能被重用的代码,它可以被不同的数据类型重用。

使用例 5-14 给出的泛型集合类 GenericSet < T >可以定义出不同的具体类,例如 GenericSet < **Integer** >表示 Integer 型的集合类,GenericSet < Double >表示 Double 型的集合类,GenericSet < Object >表示 Object 型的集合类,等等。

泛型集合类 GenericSet < T >没有对类型形参 T 做任何限制,因此使用该类的类型实参可以是任意一种引用数据类型。例如,下列定义集合类对象的语句都是正确的。

```
GenericSet < Integer > is = new GenericSet < Integer >( … ); //定义一个 Integer 型集合对象
GenericSet < Double > ds = new GenericSet < Double >( … );   //定义一个 Double 型集合对象
GenericSet < Character > cs = new GenericSet < Character >( … );  //定义一个 Character 型集合
                                                                   //对象
GenericSet < String > ss = new GenericSet < String >( … );   //定义一个 String 型集合对象
GenericSet < Object > os = new GenericSet < Object >( … );   //定义一个 Object 型集合对象
```

注:上述语句小括号中的"…"表示被省略的集合初始值。

如果希望将类型实参限定在某个**类族**或**接口族**范围内,例如限定在数值类 Number 及其子类范围内,则例 5-14 中的泛型集合类需要按如下格式来定义类型形参 T。

```
class GenericSet < T extends Number > {       //类型形参 T 只能指代数值类 Number 或其子类
      …                                       //其余代码不变,省略
}
```

使用这个泛型集合类 GenericSet < T extends Number >只能定义数值型的集合对象。例如,下列定义集合对象语句中的类型实参必须是数值类 Number 或其子类。

```
GenericSet < Number > ns = new GenericSet < Number >( … );  //正确:定义一个 Number 型集合对象
GenericSet < Integer > is = new GenericSet < Integer >( … );  //正确:Integer 是 Number 的子类
GenericSet < Double > ds = new GenericSet < Double >( … );  //正确:Double 是 Number 的子类
GenericSet < Character > cs = new GenericSet < Character >( … );
                                                             //错误:Character 不是 Number 的子类
GenericSet < String > ss = new GenericSet < String >( … );   //错误:String 不是 Number 的子类
GenericSet < Object > os = new GenericSet < Object >( … );   //错误:Object 不是 Number 的子类
```

2. 泛型族

这里以 Java API 中的数值类 Number 为例,具体讲解什么是泛型族。数值类 Number 有 6 个子类,分别是 Byte、Short、Integer、Long、Float 和 Double,它们构成一个以类 Number 为根类的数值类族。

1) 泛型族介绍

基于**同一泛型类**为某个**类族**或**接口族**中的每个类分别定义出一个具体的类,这些具体类合在一起称为一个**泛型族**。例 5-15 给出一个简单的泛型类 A < T >,下面讨论如何基于这个泛型类 A < T >定义出一个泛型族。

例 5-15 一个简单的泛型类 A < T >示例代码

```
1  class A<T> {                          //一个简单的泛型类 A
2        public  T a;                    //字段:T 类型
3        public A(T x) {  a = x;  }      //构造方法
4  }
```

可以基于泛型类 A < T >为数值类族中的根类 Number 及其 6 个子类分别定义一个具体类,即 A < **Number** >、A < **Byte** >、A < **Short** >、A < **Integer** >、A < **Long** >、A < **Float** >、A < **Double** >,这 7 个具体类就组成了一个基于泛型类 A < T >的数值类泛型族。

2) 通配符类型的引用变量

可以使用泛型族中的具体类定义引用变量或创建对象。例如:

```
A< Integer> ia ;            //定义 A< Integer >类的引用变量 ia
ia = new A< Integer>(10); //创建一个 A< Integer >类的对象,将其引用赋值给引用变量 ia
```

这里,A < Integer >类的引用变量 ia 引用的是一个同类型对象,即 A < Integer >类的对象。

Java 语言中,超类或接口的引用变量可以引用其子类的对象,其目的是为了让类族或接口族中的类共用算法代码(对象替换与多态机制)。Java 语言也专门为泛型族设计了 3 种用问号"?"表示的**通配符类型**(wildcard type)引用变量,其目的是为了让**泛型族**共用算法代码。

(1) 泛型名<?>:可引用基于泛型类或泛型接口所定义出的任何具体类的对象。例如:

```
A<?>  ref ;
```

该语句定义了一个 A <?>通配符类型的引用变量 ref,它可以引用 A < Integer >、A < Double >、A < String >、A < Object >等任何具体类的对象。

(2) 泛型名<? extends 类名>:只能引用基于泛型类或泛型接口由某个类及其子类所定义出的具体类的对象,即只能引用某个泛型族中类的对象。例如:

```
A< ? extends Number>  ref ;
```

该语句定义了一个 A <? extends **Number** >通配符类型的引用变量 ref,它只能引用 A < Number >、A < Integer >、A < Double >等由类 Number 及其子类所定义出的数值类泛型族中的具体类对象。

(3) 泛型名<? super 类名>:只能引用基于泛型类或泛型接口由**某个**类及其超类所定义出的具体类的对象。例如:

```
A< ? super Integer>  ref ;
```

该语句定义了一个 A <? super **Integer** >通配符类型的引用变量 ref,它只能引用 A < Integer >、A < Number >、A < Object >等由类 Integer 及其超类所定义出的具体类的对象。

通配符类型的语法细则:通配符类型可用于定义方法的**形参**或**返回值类型**,或定义方法中的**局部引用变量**,也可用于定义类中的**字段成员**,但不能用于创建对象。

3) 泛型族共用算法代码

通过对象替换与多态机制,同一类族或接口族中的类可以共用算法代码。Java 语言还通过对象替换与多态机制,再结合通配符类型,可以继续让同一泛型族中的类共用算法代码。

方法是描述某种数据处理算法的代码,方法中形参的数据类型决定了算法能够处理哪种类型的数据。例 5-15 曾给出一个简单的泛型类 A<T>,假设有如下一个显示方法 show():

```
void show( A<Integer> aRef )              //形参 aRef 为 A<Integer>类的对象引用
{  System.out.println( aRef.a );  }       //显示 A<Integer>类对象的字段成员 a
```

显示方法 show()只能处理 A<Integer>类的对象,例如:

```
show( new A<Integer>(5) );                //处理 A<Integer>类的对象,显示结果:5
```

如果将显示方法 show()的形参类型改为通配符类型,则可以让这个方法被某个泛型族共用。例如,按如下形式修改方法 show()中形参 aRef 的数据类型:

```
void show( A<? extends Number>  aRef ) //aRef 可引用基于 A<T>的 Number 泛型族中的所有对象
{  System.out.println( aRef.a );  }
```

修改后的方法 show()可以处理基于泛型类 A<T>的数值类 Number 泛型族中的所有对象,即数值类泛型族可以共用方法 show()的算法代码。例如:

```
show( new A<Integer>(5) );                //处理 A<Integer>类的对象,显示结果:5
show( new A<Double>(5.5) );               //处理 A<Double>类的对象,显示结果:5.5
show( new A<Float>(5.5f) );               //处理 A<Float>类的对象,显示结果:5.5
```

3. 泛型类的继承与扩展

泛型类可以被继承、扩展,扩展时可以继续增加类型形参。例 5-16 定义了两个泛型类 B1<T>和 B2<T1, T2>。这两个类继承并扩展了例 5-15 中的泛型类 A<T>,它们是泛型类 A<T>的**泛型子类**。反过来,泛型类 A<T>被称为是泛型类 B1<T>、B2<T1, T2>的**泛型超类**。

例 5-16 继承泛型类 A<T>所扩展出的两个泛型子类 B1<T>、B2<T1, T2>示例代码

```
1  class B1<T> extends A<T> {          //定义泛型类 B1<T>时继承泛型类 A<T>
2        public  T b;                  //新增成员
3        public B1(T x, T y) {         //构造方法
4             super(x);                //调用超类的构造方法
5             b = y;
6  }  }
```

```
1  class B2<T1, T2> extends A<T1> {    //定义泛型类 B2<T1, T2>时继承泛型类 A<T>
2        public  T2 b;                 //新增成员
3        public B2(T1 x, T2 y) {       //构造方法
4             super(x);                //调用超类的构造方法
5             b = y;
6  }  }
```

1）泛型类 B1 < T >

例 5-16 中，泛型类 B1 < T >是泛型类 A < T >的子类。基于泛型类 A < T >可以定义出不同类型的具体类，例如 A < Number >、A < Integer >、A < Double >等，它们构成了一个基于泛型超类 A < T >的泛型族（参见图 5-15）。

同样，基于泛型类 B1 < T >也可以定义出不同类型的具体类，例如 B1 < Number >、B1 < Integer >、B1 < Double >等，它们构成了一个基于泛型子类 B1 < T >的泛型族。这两个泛型族中的具体类之间存在什么样的继承关系呢？

- 同一泛型族中的具体类之间不存在继承关系。例如，虽然 Integer 是 Number 的子类，但 A < Integer >不是 A < Number >的子类。同理，B1 < Integer >也不是 B1 < Number >的子类。
- 泛型超类与泛型子类的同类型具体类之间存在继承关系。例如 B1 < Number >是 A < Number >的子类、B1 < Integer >是 A < Integer >的子类等。

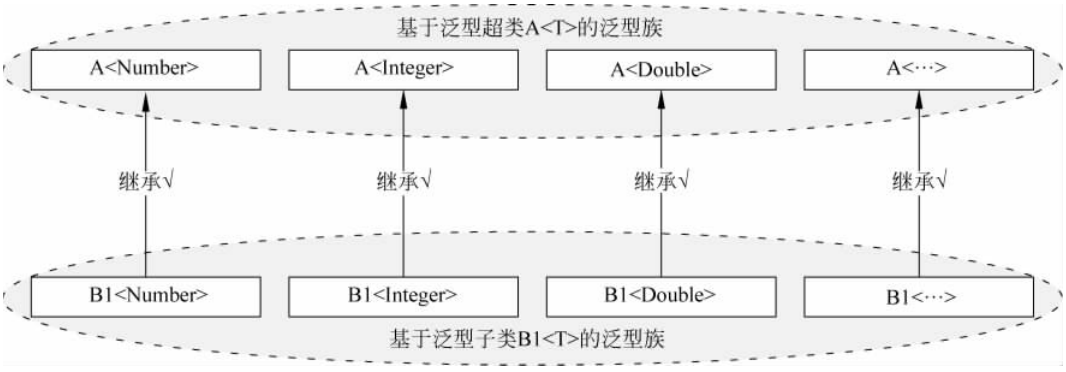


图 5-15 基于泛型超类 A < T >的泛型族与基于泛型子类 B1 < T >的泛型族

2）泛型类 B2 < T1，T2 >

例 5-16 中，泛型类 B2 < T1，T2 >也是泛型类 A < T >的子类。但与 B1 < T >不同的是，泛型类 B2 < T1，T2 >有两个类型参数。对基于泛型超类 A < T >的泛型族和基于泛型子类 B2 < T1，T2 >的泛型族，图 5-16 给出了这两个泛型族中具体类之间的继承关系图。

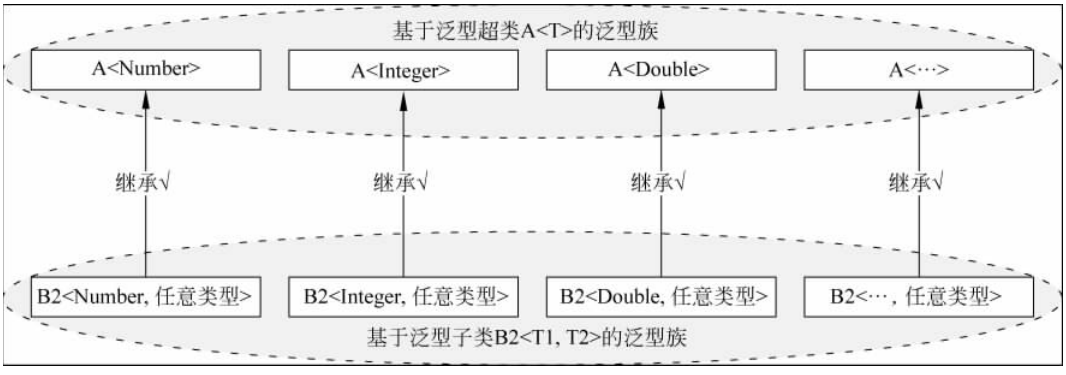


图 5-16 基于泛型超类 A < T >的泛型族与基于泛型子类 B2 < T1，T2 >的泛型族

4. 泛型方法

可以将类中的某个方法成员单独定义成一个**泛型方法**。例如：

```
class A {                                          //将类 A 中的方法成员 show()单独定义成一个泛型方法
      …                                            //其他代码省略
      public static < T > void show(T obj)         //泛型方法:显示 T 类型的对象
      {   System.out.println( obj.toString() );   }
}
```

其中,类型形参 T 可指代任意一种具体的引用数据类型。方法 show()可理解为是一个能够处理任意引用类型数据的泛型方法。定义泛型方法时,需在紧邻返回值的前面指定类型形参。可以为泛型方法指定多个类型形参,多个类型形参之间用逗号“,”隔开。

调用泛型方法时,Java 编译器会自动根据所处理对象的类型推断出类型形参所指代的具体数据类型(即类型实参),程序员不需要显式指定。例如：

```
A.show( new Integer(5) );      //编译器可推断出类型形参 T 指代的是 Integer 类型,显示结果:5
A.show( new Double(5.5) );     //编译器可推断出类型形参 T 指代的是 Double 类型,显示结果:5.5
```

5.7.3 数据集合

数据集合就是一组数据的集合。例如,表 5-4 所示的学生成绩单就是一组关于学生姓名和成绩的数据集合。

表 5-4 学生成绩单

姓　　名	成　　绩
张三	92
李四	86
王五	95
…	…

关于数据集合,存在如下 3 个层次。

(1) **数据项**(data item)。数据项是数据集合中的最小单位。例如表 5-4 中第 2 行的姓名“张三”、成绩“92”都分别是一个的数据项。数据项也称作**字段**(field)。

(2) **数据元素**(data element)。数据元素是由多个具有内在关联关系的数据项组成。例如,表 5-4 中第 2 行“张三,92”表示张三的成绩是 92,“张三”和“92”这两个数据项具有内在关联关系,它们就构成了一个数据元素。一个数据元素也称作一条**记录**(record)。

(3) **数据集合**(data set)。数据集合由多个并列的数据元素所组成。例如,表 5-4 就是一个由多个数据元素(每一行就是一个数据元素)组成的数据集合。一个数据集合也称作一张**表**(table)。

如果数据集合中各数据元素之间的逻辑关系是有序的,则称该数据集合为有序数据集合,否则称为无序数据集合。对数据集合的处理通常包括增加、查找、修改或删除数据元素,这被统称为**增查改删**(Create、Read、Update、Delete,CRUD)。为了提高查找速度,可以将数据集合中的数据元素按照某种规则事先进行排序。

编写一个处理数据集合的计算机程序，要涉及两个方面的内容：一是如何组织和存储数据集合，即数据结构；二是如何基于数据结构进行数据处理，即算法。同样的数据集合，存储的数据结构不同，将导致处理的算法也会有所不同。不同数据结构适用于不同的算法。计算机学科中的**数据结构**(data structure)就是专门研究数据结构与算法关系的课程。数据结构的研究对象就是数据集合，其研究内容是如何对数据集合进行组织、存储和处理的一般方法。

例 5-17 给出一个存储和处理表 5-4 中学生成绩单的 Java 示例代码。其中的学生类 Student 实现一个 Comparable 接口，通过 compareTo()方法定义了比较两个学生对象大小的规则。学生类 Student 还重写了 hashCode()方法和 equals()方法，用于比较两个学生对象的内容是否相等。

例 5-17 一个存储和处理学生成绩单的 Java 示例代码(StudentScoreTest.java)

```
1  public class StudentScoreTest {                    //主类
2      public static void main(String[] args) {       //主方法
3          Student sa[] = new Student[3];              //创建一个保存 3 名学生成绩的对象数组
4          sa[0] = new Student( "张三", 92 );          //添加数据元素
5          sa[1] = new Student( "李四", 86 );
6          sa[2] = new Student( "王五", 95 );
7           for (int n = 0; n < sa.length; n++)        //遍历数组,显示学生成绩单
8                  System.out.println( sa[n].toString() );
9           //比较两个对象的大小: 调用对象的 compareTo()方法
10          System.out.println( sa[0].compareTo(sa[1]) );  //比较 sa[0]和 sa[1]的大小,显示 6
11          System.out.println( sa[0].compareTo(sa[2]) );  //比较 sa[0]和 sa[2]的大小,显示 -3
12          //比较对象的内容是否相等: hashCode()可用于快速比较,equals()则是全面比较
13          Student s = new Student("赵六", 92);  //再创建一个与 s[0]成绩相同的对象 s
14          System.out.println( sa[0].hashCode() == s.hashCode() );  //显示结果为 true
15          System.out.println( sa[0].equals(s) );     // 同时比较成绩和姓名,显示结果为 false
16 }  }
17
18 class Student implements Comparable<Student> {  //学生成绩类
19       private String name;                        //姓名
20       private int score;                          //成绩
21       public Student(String p1, int p2)           //构造方法
22       {  name = p1;  score = p2;  }
23       public String toString()                    //重写 toString()方法
24       {  return String.format("%s: %d", name, score);  }
25       public int compareTo(Student s) {       //实现 Comparable 接口所规定的比较大小方法
26             //假设比较两个学生大小的规则是先比较成绩,成绩相同时再比较姓名
27             int n = score - s.score;              //先比较成绩
28             if (n != 0)  return n;                //成绩不同时,直接返回成绩比较的结果
29             else  return name.compareTo(s.name); //成绩相同时,再比较姓名
30       }
31       public int hashCode() {                     //重写 hashCode()方法
32       {  return score;  }                    //生成哈希码: 简单地将成绩作为学生的哈希码
33       public boolean equals(Object obj) {         //重写 equals()方法
```

```
34          if ((obj instanceof Student) == false)  return false;  //类型不同,则直接返回 false
35          Student s = (Student)obj;
36          if(score != s.score)  return false;           //先判断成绩,成绩不同则学生不同
37          if(name.equals(s.name) != true)  return false;  //再判断姓名是否相同
38          return true;                //姓名和成绩都相同时,两个对象的内容就相同,返回 true
39    }  }
```

例 5-17 使用对象数组来存储学生成绩的数据集合。实际应用中,对学生成绩这个数据集合还会有很多后续处理,例如增查改删,或排序等,程序员还需要为这些处理编写大量的算法代码。

为方便程序员,Java API 提供了很多具有不同功能的数据集合类。使用这些类,程序员能够快速编写出存储和处理数据集合的 Java 程序。下一节将具体介绍 Java API 中的这些数据集合类。

5.7.4 Java API 中的数据集合类

使用 Java API 所提供的数据集合类可以实现**动态数组**(或称为可变长数组)、**队列**或**堆栈**、**集合**、**映射**(或称为字典)等数据存储功能。另外 Java API 还配套提供了相关的**增查改删**和**排序**算法。

Java API 对数据集合类进行了再次抽象,提炼出 **Collection**(集合)和 **Map**(映射)两个接口,然后按功能要求由数据集合类具体实现这两个接口。Java API 这么做的目的是让多个不同的数据集合类实现相同的接口,这样就构成了一个接口族。同一接口族中的类是可以共用算法代码的。

Java API 在定义集合接口 Collection 和映射接口 Map 时还运用了泛型编程技术,这样可以基于同一泛型接口定义出不同数据类型的具体集合类,例如 Integer 型集合、Double 型集合、String 型集合、Student 型集合等,参见 5.7.1 节的例 5-14。这些不同数据类型的具体集合类构成一个泛型族,同一泛型族中的类也是可以共用算法代码的。

请读者阅读下面的集合接口 Collection<E>和映射接口 Map<K, V>说明文档,了解其中为数据集合操作所定义的一些常用方法接口。

java.util.**Collection**<**E**>接口说明文档

public interface **Collection**<**E**>
extends **Iterable**<**E**>

	修 饰 符	接口成员(节选)	功 能 说 明
1		boolean **add**(E e)	添加一个元素
2		boolean **contains**(Object o)	是否包含某个指定的元素
3		boolean **hasNext**()	是否还有下一个元素
4		E **next**()	返回下一个元素并后移一个元素
5		void **remove**()	删除迭代器当前指向的元素
6		int **size**()	返回元素的个数

续表

	修 饰 符	接口成员(节选)	功 能 说 明
7		boolean **isEmpty**()	集合是否为空
8		void **clear**()	删除所有的元素
9		Object[] **toArray**()	返回一个数组形式的集合
10	default	Stream<E> **stream**()	返回一个流形式的集合
11		Iterator<E> **iterator**()	返回一个迭代器
…			

java.util.**Map**<**K**,**V**>接口说明文档

public interface **Map**<**K**,**V**>

	修 饰 符	接口成员(节选)	功 能 说 明
1	static	interface Map.**Entry**<K,V>	内部接口,表示一个键值对
2		V **put**(K key, V value)	添加一个键值对
3		V **get**(Object key)	读取某个键的值
4	default	V **replace**(K key, V value)	修改某个键的值
5	default	V **remove**(Object key)	删除某个键值对
6		boolean **containsKey**(Object key)	是否包含指定的键
7		boolean **containsValue**(Object value)	是否包含指定的值
8		int **size**()	返回键值对的个数
9		boolean **isEmpty**()	映射是否为空
10		void **clear**()	删除所有的键值对
…			

下面具体讲解 Java API 中常用的数据集合类。

1. 数组列表类 ArrayList<E>

Java API 中的数组列表类 **ArrayList**<E>实现了集合接口 Collection<E>,可实现动态数组的功能。

1) 创建数组列表

例 5-18 给出一个使用类 ArrayList<E>创建数组列表的 Java 示例代码。

例 5-18 一个使用类 ArrayList<E>创建数组列表的 Java 示例代码(ArrayListTest.java)

```
1  import java.util.ArrayList;                                    //导入数组列表类 ArrayList
2  public class ArrayListTest {                                   //测试类
3      public static void main(String[] args) {                   //主方法
4          ArrayList<Integer> v1 = new ArrayList<Integer>();      //Integer 型数组列表
5          v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );     //添加元素
6          for (int n = 0; n<v1.size(); n++)    //使用序号(下标)遍历显示各元素
7                System.out.print( v1.get(n) +", " );
8          System.out.println();
9
10         ArrayList<Student> v2 = new ArrayList<>();             //Student 型数组列表
```

```
11          v2.add( new Student("张三", 92) );          //添加元素
12          v2.add( new Student("李四", 86) );
13          v2.add( new Student("王五", 95) );
14          for (int n = 0; n < v2.size(); n++)         //使用序号(下标)遍历显示各元素
15            System.out.println( v2.get(n) );
16  }  }
```

在 Eclipse 集成开发环境中运行例 5-18 的程序,运行结果如图 5-17 所示。

图 5-17　例 5-18 程序的运行结果

2) 使用增强 for 语句遍历数组列表

除了常规 for 语句之外,程序员还可以使用增强 for 语句遍历数组列表中的元素。例如:

```
ArrayList < Integer > v1 = new ArrayList < Integer >();     //Integer 型数组列表
v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );      //添加元素
for ( Integer e : v1)                                       //使用增强 for 语句遍历显示各元素
      System.out.print( e + ", " );
```

使用增强 for 语句遍历数组列表 v1 的显示结果为:

```
3, 9, 7, 5,
```

3) 使用迭代器遍历数组列表

遍历数组列表的第 3 种方法是**迭代器**(iterator)。迭代器是 Java API 提供的一种遍历数据集合的模式,可调用数据集合类的 **iterator**()方法获得数据集合的迭代器对象。迭代器对象描述了遍历数据集合时的元素位置信息,如图 5-18 所示。

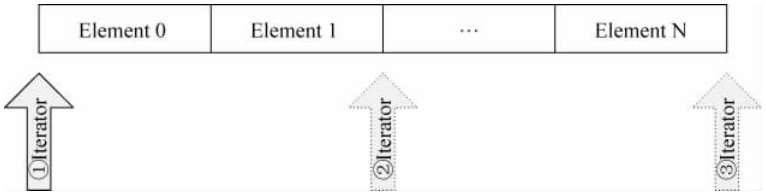


图 5-18　数据集合的迭代器对象

假设有一个如下的数组列表 v1:

```
ArrayList < Integer > v1 = new ArrayList < Integer >();      //Integer 型数组列表
v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );       //添加元素
```

使用迭代器遍历数据集合 v1 的代码框架如下:

```
Iterator<Integer> iv1 = v1.iterator();  //获取 v1 的迭代器对象,此时迭代器处于图 5-18
                                        //中①的位置
while ( iv1.hasNext() ) {           //如果有下一个元素则为 true,例如图 5-18 中①、②的位置
    Integer e = iv1.next();         //取出下一个元素,然后迭代器自动后移一个元素
    System.out.print( e +", " );
}  //当迭代器处于图 5-18 中③的位置时,hasNext()返回 false,循环结束
```

Java API 中所有实现 Collection<E>接口的数据集合类都可以使用迭代器进行遍历操作。使用迭代器,不同数据集合类的遍历代码框架都是一样的。换句话说,不同的数据集合类可以通过迭代器共用遍历算法代码。

Java API 中的迭代器 **Iterator** 是一个泛型接口。请读者阅读下面的迭代器接口 Iterator<E>说明文档。

java.util.**Iterator**<E>接口说明文档

public interface **Iterator**<**E**>

	修 饰 符	接口成员(全部)	功 能 说 明
1		boolean **hasNext**()	是否还有下一个元素
2		E **next**()	返回下一个元素并后移一个元素位置
3	default	void **remove**()	删除迭代器当前指向的元素
4	default	void **forEachRemaining**(Consumer<? super E> action)	遍历并处理集合中剩余的元素

4) 使用 Collections 类中的静态方法处理数组列表

为了方便程序员,Java API 提供了一个专门的数据集合算法类 **Collections**,其中包括各种数据集合处理算法。这些处理算法被定义成静态成员,可直接通过类名 Collections 进行调用。例 5-19 给出一个使用 Collections 类中静态方法处理数组列表的 Java 示例代码。

例 5-19 一个使用 Collections 类中静态方法处理数组列表的 Java 示例代码(ArrayListTest.java)

```
1  import java.util.ArrayList;                              //导入数组列表类 ArrayList
2  import java.util.Collections;                            //导入集合算法类
3  public class ArrayListTest {                             //测试类
4      public static void main(String[] args) {             //主方法
5          ArrayList<Integer> v1 = new ArrayList<>();       //Integer 型数组列表
6          v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );  //添加元素
7          System.out.println( v1 );                        //直接显示整个数组列表
8          Collections.sort( v1 );                          //对元素进行排序
9          System.out.println( v1 );
10         Collections.reverse( v1 );                       //逆序排列所有元素
11         System.out.println( v1 );
12         System.out.println( Collections.max( v1 ) );     //求数据集合中元素的最大值
13 } }
```

在 Eclipse 集成开发环境中运行例 5-19 的程序,运行结果如图 5-19 所示。

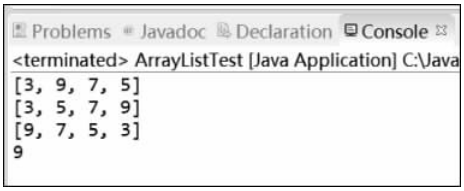


图 5-19　例 5-19 程序的运行结果

2. 双端队列类 LinkedList < E >

Java API 中的双端队列类 **LinkedList** < E >实现了集合接口 Collection < E >。使用这个双端队列类可以实现**队列**的功能,也可以实现**堆栈**的功能。队列和堆栈是两种存储数据集合时常用的数据结构。

如果将数据集合看作是一组数据元素的有序队列,则**队列**(queue)就是在添加元素时总是被加在**队列尾**,取出元素时总是取出排在队列最前面(**队列头**)的那个元素,这种排序规则称为**先进先出**(First-In-First-Out,**FIFO**)。双端队列类 LinkedList < E >中实现队列功能的两个主要方法成员如下。

- **offer**():在队列尾添加一个元素。
- **poll**():取出并删除队列头的元素。

如果将数据集合看作是一组数据元素的有序堆叠,则**堆栈**(stack)就是在添加元素时总是被放在堆栈的顶部(**栈顶**),取出元素时也总是取出栈顶(即最后被放入堆栈)的那个元素,这种排序规则称为**后进先出**(Last-In-First-Out,**LIFO**)。双端队列类 LinkedList < E >中实现堆栈功能的两个主要方法成员如下。

- **push**():向堆栈中压入一个元素。
- **pop**():从堆栈中弹出一个元素。

例 5-20 给出一个使用类 linkedList < E >实现队列和堆栈功能的 Java 示例代码。

例 5-20　一个使用类 linkedList < E >实现队列和堆栈功能的 Java 示例代码(LinkedListTest.java)

```
1  import java.util.LinkedList;                                    //导入双端队列类 LinkedList
2  public class LinkedListTest {                                   //测试类
3      public static void main(String[] args) {                    //主方法
4          int n;
5          //下面演示使用 Integer 型双端队列实现一个队列
6          LinkedList< Integer> q = new LinkedList<>();            //Integer 型双端队列
7          for (n = 1; n <= 3; n++)
8                q.offer( n );                                     //实现一个队列:在队尾添加一个元素
9          System.out.println( q );                                //显示队列
10         for (n = 1; n <= 3; n++)
11               System.out.print( q.poll() + "," );               //取出并删除队列头的元素
12         System.out.println( "\n" + q + "\n" );                  //再次显示队列,此时队列为空
13         //下面演示使用 Integer 型双端队列实现一个堆栈
```

```
14          LinkedList<Integer> s = new LinkedList<>();              //Integer 型双端队列
15          for (n = 1; n<= 3; n++)
16                s.push( n );                              //实现一个堆栈:向栈中压入一个元素
17          System.out.println( s );                        //显示堆栈
18          for (n = 1; n<= 3; n++)
19                System.out.print( s.pop() + "," );  //从栈中弹出一个元素
20          System.out.println( "\n" + s );             //再次显示堆栈,此时堆栈为空
21    }  }
```

在 Eclipse 集成开发环境中运行例 5-20 的程序,运行结果如图 5-20 所示。

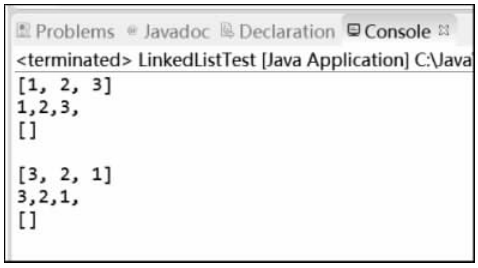


图 5-20 例 5-20 程序的运行结果

3. 集合类 HashSet<E>

Java API 中的集合类 **HashSet**<E>实现了集合接口 Collection<E>。集合类 HashSet<E>具有如下特点。

- 集合中的数据元素是无序的。
- 集合中的数据元素不能重复。**注**:集合类判别数据元素是否重复的方法是,先调用数据元素的 **hashCode**()方法进行快速判重;如果两个元素的哈希码相同,则再调用数据元素的 **equals**()方法进行精确判重(判重就是判断两个元素的内容是否完全一样)。
- 遍历集合中的数据元素,可以使用增强 for 语句和迭代器。**注**:集合中的元素没有序号(下标),不能使用普通 for 语句。

例 5-21 给出一个使用类 HashSet<E>实现集合功能的 Java 示例代码。

例 5-21 一个使用类 HashSet<E>实现集合功能的 Java 示例代码(HashSetTest.java)

```
1  import java.util.HashSet;                                  //导入集合类 HashSet
2  public class HashSetTest {                                 //测试类
3      public static void main(String[] args) {               //主方法
4          HashSet<String> s = new HashSet<>();               //String 型集合
5          s.add("1st");  s.add("2nd");  s.add("3rd");  //添加元素
6          s.add("4th");  s.add("5th");
7          System.out.println( s );                  //显示集合,各元素按其哈希码的顺序存放
8          s.remove("4th");                                   //删除元素
9          System.out.println( s );                           //再次显示集合
10         System.out.println( s.size() );                    //显示集合中的元素个数
11 }  }
```

在 Eclipse 集成开发环境中运行例 5-21 的程序,运行结果如图 5-21 所示。

图 5-21 例 5-21 程序的运行结果

4. 映射类 HashMap < K, V >

Java API 中的映射类 **HashMap** < K, V >实现了映射接口 Map < K, V >。使用这个映射类可以存储类似于字典形式的**键值对**(key-value pair)数据。

一个键值对包括两个数据项:一个称为**键**,另一个称为**值**。其含义是将键映射到某个值。例如,学生成绩单中的"张三-92"就是一种"姓名-分数"键值对,英汉字典中的"China-中国"就是一种"英文-中文"键值对。**注**:在程序设计中,映射也可以被称为字典。

使用映射类 HashMap < K, V >所存储的键值对数据集合具有以下特点。

- 映射类中的数据元素是无序的。
- 映射类中数据元素的键不能重复。**注**:映射类判别数据元素的键是否重复的方法是,先调用键所属类的 **hashCode**()方法进行快速判重;如果两个键的哈希码相同,则再调用键的 **equals**()方法进行精确判重。
- 映射类没有迭代器,其中的元素也没有序号(下标)。如需遍历映射,则可将映射或其键转换成集合,再按集合的方式进行遍历。

例 5-22 给出一个使用映射类 HashMap < K, V >存储表 5-4 中学生成绩单的 Java 示例代码。

例 5-22 使用映射类 HashMap < K, V >存储表 5-4 中学生成绩单的 Java 示例代码(HashMapTest.java)

```
1  import java.util.HashMap;                                    //导入映射类 HashMap
2  import java.util.Set;                                        //导入集合类 HashSet
3  public class HashMapTest {                                   //测试类
4      public static void main(String[] args) {                 //主方法
5          HashMap<String, Integer> h = new HashMap<>();        //String - Integer 型映射
6          h.put( "张三", 92 );  h.put( "李四", 86 );  h.put( "王五", 95 );    //添加元素
7          //遍历映射:取出键的集合,然后遍历该集合
8          Set<String> kSet = h.keySet();                       //取出映射对象 h 中键的集合
9          for (String k: kSet)  //使用增强 for 语句遍历键的集合 kSet
10         {  System.out.println( k +" - " +h.get(k) );  }//取出映射 h 中键 k 所对应的值
11 } }
```

在 Eclipse 集成开发环境中运行例 5-22 的程序,运行结果如图 5-22 所示。

图 5-22 例 5-22 程序的运行结果

5. Java API 数据集合类的继承与实现关系

图 5-23 给出了 Java API 数据集合类的继承与实现关系。

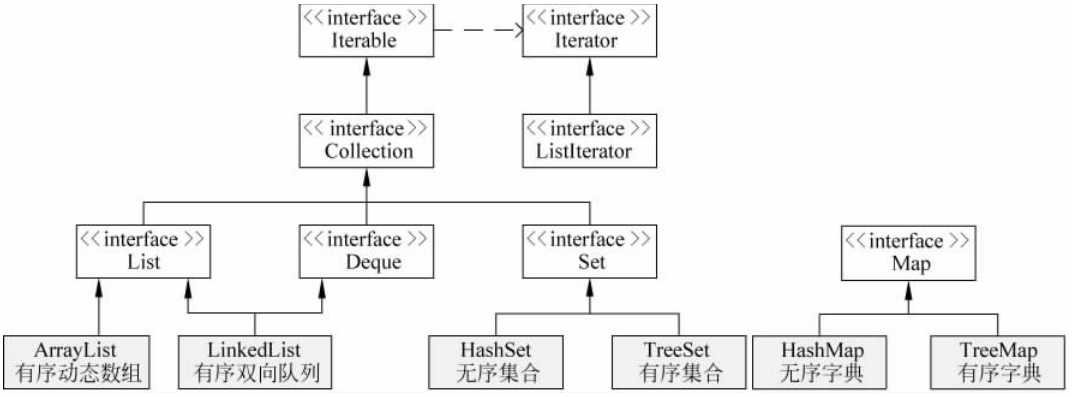


图 5-23 Java API 数据集合类的继承与实现关系

注 1：图 5-23 中，集合类 HashSet 基于 Hash 表存储数据（无序），另外还有一种集合类 TreeSet 则基于红黑树存储数据（自动排序）。这两个类的使用方法是一样的，所不同的是，HashSet 类的内部会通过元素的 hashCode()和 equals()方法来判断集合中是否有重复元素，而 TreeSet 类则是通过元素的 compareTo()方法来进行元素的判重和排序。

注 2：图 5-23 中，映射类 HashMap 基于 Hash 表存储数据（无序），另外还有一种映射类 TreeMap 则基于红黑树存储数据（自动排序）。这两个类的使用方法是一样的，所不同的是，HashMap 类的内部会通过键所属类的 hashCode()和 equals()方法来判断映射中是否有重复元素，而 TreeMap 类则是通过键所属类的 compareTo()方法来进行元素的判重和排序。

这里对本节所讲解的 Java API 数据集合类做一个总结。

（1）动态数组类 **ArrayList** 可实现动态数组的功能；双端队列类 **LinkedList** 可实现队列或堆栈的功能；集合类 **HashSet** 用于保存无序的数据集合。这 3 个数据集合类都实现了 **Collection**<E>接口，都可以使用增强 for 语句和迭代器遍历集合元素。动态数组类 ArrayList 和双端队列类 LinkedList 是有序集合，可使用普通 for 语句按照元素序号（下标）遍历集合。

（2）映射类 **HashMap** 用于保存键值对形式的数据集合。映射类实现了 **Map**<K，V>接口。映射类 HashMap 没有迭代器，也没有元素序号（下标）。

（3）算法类 **Collections** 以静态方法的形式定义了一组专门处理上述数据集合类的算法。

本节习题

1. 下列关于泛型类的描述中,错误的是(　　)。
   A. 带类型参数的类称为泛型类　　B. 类型形参可指代某种具体的数据类型
   C. 使用泛型类时可省略类型实参　　D. 使用泛型类可定义出不同类型的具体类
2. 下列关于泛型的描述中,错误的是(　　)。
   A. 带类型参数的类称为泛型类　　B. 带类型参数的接口称为泛型接口
   C. 带类型参数的方法称为泛型方法　　D. 带类型参数的字段称为泛型字段
3. 下面的类(　　)不是泛型类 G < T >定义出的具体类。
   A. G < Integer >　B. G < String >　C. G < Object >　D. G < double >
4. 下面的类中(　　)没有实现集合接口 Collection < E >。
   A. ArrayList < E >　　B. LinkedList < E >
   C. HashSet < E >　　D. HashMap < K, V >
5. 动态数组类 ArrayList < E >可以实现(　　)的功能。
   A. 动态数组　　B. 堆栈　　C. 无序集合　　D. 字典
6. 双端队列类 LinkedList < E >可以实现(　　)的功能。
   A. 动态数组　　B. 堆栈　　C. 无序集合　　D. 字典
7. 映射类 HashMap < K, V >可以实现(　　)的功能。
   A. 动态数组　　B. 堆栈　　C. 无序集合　　D. 字典
8. 下面的类中(　　)是 Java API 中专门用于处理数据集合的算法类。
   A. ArrayList < E >　　B. LinkedList < E >
   C. Collection < E >　　D. Collections

5.8 枚举类型

和 Java 语言中其他基本数据类型相比,布尔(boolean)类型的主要特点是其值域只有两个取值,即真和假,分别用关键字 true 和 false 表示。实际程序设计任务中也经常会碰到与布尔类型相似的数据,它们的值域是有限的(称为是可枚举的)。例如一个星期只有星期一、星期二、……星期日等 7 天,其值域是可枚举的。Java 语言可以将值域可枚举的数据定义成**枚举类型**。枚举类型值域中的每个取值称为一个**枚举常量**。

注:C 语言中有枚举类型、联合体和结构体等自定义数据类型。Java 语言保留了枚举类型,但没有联合体和结构体。在 Java 语言中,结构体可以用类实现,即结构体实际上是一个只包含公有字段的类。

1. 定义枚举类型

定义枚举类型就是列出该类型所有可能的取值,即枚举常量。枚举常量由程序员命名,习惯上使用全大写字母。定义枚举类型时需使用关键字 **enum**。例如:

```
public enum WeekDay {                                    //定义一个表示星期几的枚举类型 WeekDay
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY        //枚举常量,共 7 个
}
```

2. 使用枚举类型定义变量

可以定义枚举类型的变量,称为**枚举变量**。枚举变量可以保存枚举类型的数据。例如:

```
WeekDay d = WeekDay.MONDAY;          //定义枚举变量 d,初始化为枚举常量 MONDAY(星期一)
```

可以看出,枚举常量便于程序员记忆,所编写的源代码也更容易理解。

3. 枚举类型是一种特殊的类

Java 语言中的枚举类型实际上是一个类,并且都自动继承枚举类 **Enum**<E>。定义枚举类型,实际上是先继承枚举类,然后在此基础上进行扩展,例如添加字段或方法。请读者阅读下面的枚举类 Enum<E>说明文档。

java.lang.**Enum**<E extends Enum<E>>类说明文档

public abstract class **Enum**<E extends Enum<E>>
extends **Object**
implements **Comparable**<E>, **Serializable**

	修 饰 符	类成员(节选)	功 能 说 明
1		String **name**()	返回枚举常量的名字
2		int **ordinal**()	返回枚举常量的序号
3		String **toString**()	将枚举类型转换成字符串
		…	

Java 编译器在编译枚举类型的类定义代码时会自动添加一个返回枚举常量数组的静态方法 **values**()。例 5-23 给出一个完整的枚举类型 WeekDay 定义及使用示例代码。

例 5-23 一个完整的枚举类型 WeekDay 定义及使用示例代码(EnumTest.java)

```
1  public class EnumTest {                                  //测试类
2      public static void main(String[] args) {             //主方法
3          for ( WeekDay e: WeekDay.values() )              //列出 WeekDay 中的所有枚举常量
4              System.out.print( e.name() +", " );          //显示枚举常量的名称
5          System.out.println();
6          WeekDay d = WeekDay.MONDAY;                      //定义枚举变量并初始化为 MONDAY
7          System.out.println( d.ordinal() );               //显示 MONDAY 的内部整数编号
8          System.out.println( d.name() );                  //显示 MONDAY 的名称
9          d.isWeekend();                                   //检查 MONDAY 是否周末
10 } }
11
12 enum WeekDay {  //定义一个表示星期几的枚举类型 WeekDay
13     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,THURSDAY, FRIDAY, SATURDAY;  //枚举常量
14     public  void isWeekend() {                           //添加方法成员:检查是否周末
15         if (this == SATURDAY || this == SUNDAY)          //枚举类型可以做关系运算
16             System.out.println( "The day is Weekend." );
17         else
18             System.out.println( "The day is not Weekend." );
19 } }
```

在 Eclipse 集成开发环境中运行例 5-23 的程序,运行结果如图 5-24 所示。

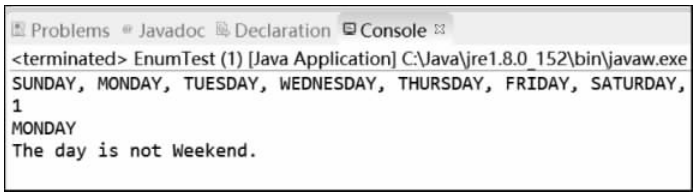


图 5-24 例 5-23 程序的运行结果

本节习题

1. C/C++语言中有数组、枚举类型、联合体和结构体等自定义数据类型,Java 语言无法描述的数据类型是(　　)。

A. 数组　　B. 枚举类型

C. 联合体　　D. 结构体

2. 返回枚举类型中枚举常量数组的方法是(　　)。

A. name()　　B. ordinal()

C. values()　　D. toString()

3. 返回枚举类型中枚举常量名称的方法是(　　)。

A. name()　　B. ordinal()

C. values()　　D. toString()

4. 返回枚举类型中枚举常量序号的方法是(　　)。

A. name()　　B. ordinal()

C. values()　　D. toString()

5. 将枚举类型转换成字符串的方法是(　　)。

A. name()　　B. ordinal()

C. values()　　D. toString()

5.9 Java 源程序中的注释和注解

程序员可以对源程序中的代码进行**注释**(comment),其作用是便于今后自己或其他程序员阅读、理解源代码。Java 语言为程序员提供了 3 种不同的注释形式。

(1) 单行注释。以“//”开头,到所在行的行尾结束。例如:

```
//单行注释内容
```

(2) 多行注释。以“/*”开头,以“*/”结束。例如:

```
/*
    注释内容,可以有多行
*/
```

(3) 文档注释。Java 语言还提供了第 3 种注释形式,这就是**文档注释**(documentation comment)。文档注释以"/**"开头,以"*/"结束。例如:

```
/**
    文档注释的内容,可以有多行
*/
```

5.9.1 文档注释

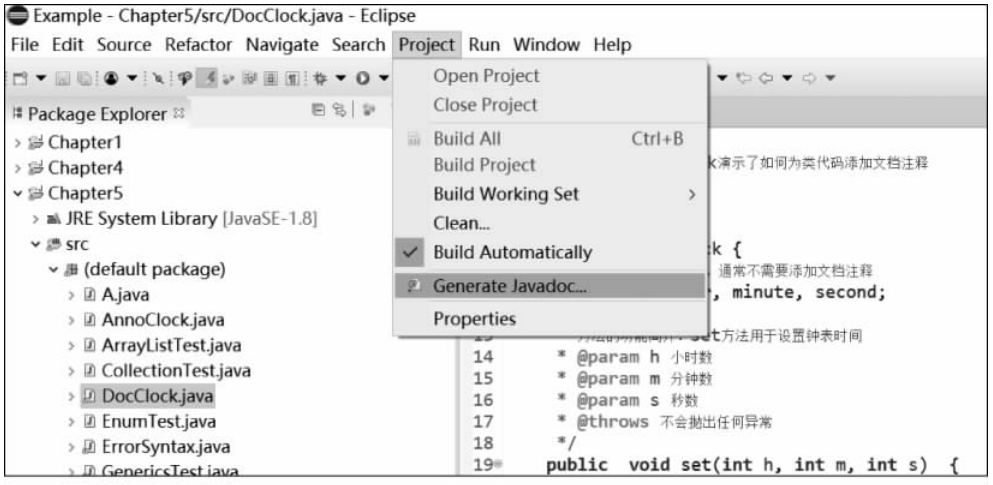
Java 源程序文件(*.java)会被编译成字节码程序文件(*.class),程序员可以将编译后的字节码程序提供给其他程序员使用。为了让其他程序员了解字节码程序中类的功能和使用方法,程序员需要另外编写一份说明文档。例如,Java API 就提供了全套的说明文档,程序员通过该文档可以详细了解 Java API 的功能和使用方法。

程序员在编写 Java 源程序时,可以为类以及其中的字段、方法添加**文档注释**,然后使用文档生成工具软件(\JDK 安装目录\bin**javadoc.** exe)自动生成说明文档,这样就能大大减轻程序员编写说明文档的工作量。文档注释以"/**"开头,以"*/"结束。例 5-24 给出一个添加了文档注释的钟表类 DocClock 示例代码。

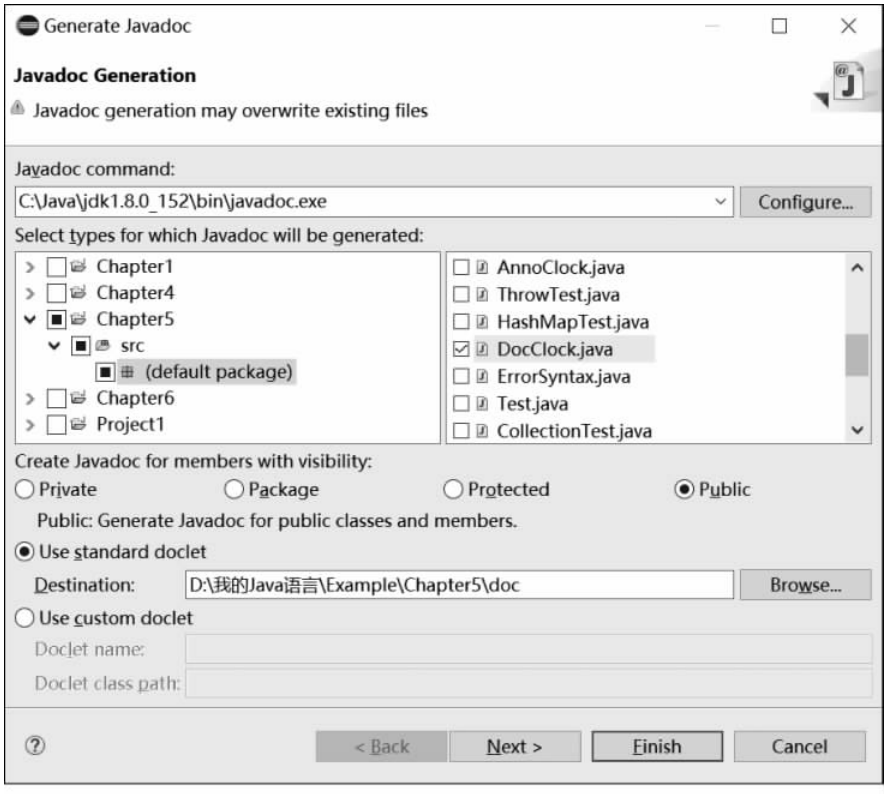
例 5-24 一个添加了文档注释的钟表类 DocClock 示例代码(DocClock.java)

```
 1  /**
 2   * 类的功能简介:DocClock 是一个钟表类,其中添加了文档注释。
 3   */
 4  public class DocClock {                    //添加了文档注释的钟表类
 5      private int hour, minute, second;      //私有成员不对外开放,通常不需要添加文档注释
 6       /**
 7        * 方法 set():设置钟表时间,h-小时数,m-分钟数,s-秒数。
 8        */
 9      public void set(int h, int m, int s)   //添加了文档注释的方法
10      {  hour = h;  minute = m;  second = s;  }
11      public  void show()                    //未添加文档注释的方法
12      {  System.out.println( hour + ":" + minute + ":" + second );  }
13       /**
14        * 方法 toString():将时分秒数据转成字符串格式(重写从 Object 继承来的方法)。
15        */
16      public String toString()               //添加了文档注释的方法
17      {  return String.format("Clock@ %d:%d:%d", hour, minute, second);  }
18       /**
19        * 构造方法:设置钟表时间,h-小时数,m-分钟数,s-秒数。
20        */
21      public DocClock(int h, int m, int s)   //添加了文档注释的方法
22      {  hour = h;  minute = m;  second = s;}
23  }
```

在 Eclipse 集成开发环境中,程序员可以很方便地调用 JDK 的文档生成工具软件 **javadoc.** exe。选择 Project→Generate Javadoc,进入生成 Java 文档对话框,如图 5-25 所示。



(a) 在Eclipse集成开发环境中选择Generate Javadoc菜单



(b) 进入生成Java文档对话框

图 5-25　在 Eclipse 集成开发环境中调用 JDK 的文档生成工具软件

图 5-25 演示了在 Eclipse 集成开发环境中为例 5-24 钟表类 DocClock 自动生成说明文档的操作过程。在图 5-25(b)所示的生成 Java 文档对话框中,程序员需要设置的选项主要有 4 项。

- Javadoc command。在此选项中设置 JDK 文档生成工具软件 javadoc.exe 的安装路径。

- Select types for which Javadoc will be generated。在此选项中选择为哪些 Java 源程序文件生成说明文档。
- Create Javadoc for members with visibility。在此选项中选择为哪种访问权限的类以及类成员生成说明文档。通常应选择 Public，即为公有类以及公有类成员生成说明文档。
- Destination。在此选项中设置所生成说明文档文件的保存路径。

单击图 5-25(b)中的 Finish 按钮，将在 Java 项目根目录下的 doc 子目录中生成一组 HTML 格式的文档说明文件，其中的 index.html 就是文档说明的主页。使用浏览器打开主页文件 index.html，查看所生成的钟表类 DocClock 说明文档，如图 5-26 所示。

(a) 类及构造器（即构造方法）概要部分

(b) 方法成员的概要部分

图 5-26 查看为例 5-24 钟表类 DocClock 自动生成的说明文档

JDK 文档生成工具软件会从 Java 源程序文件中提取出如下说明文档内容。

(1) **类的信息**。将具有指定访问权限的类名、类的继承和实现关系、字段成员、方法成员签名等信息提取出来,放入说明文档。

(2) **文档注释**。将所有以"/**"开头、以"*/"结束的文档注释内容提取出来,放入说明文档。

5.9.2 注解

程序员可以定义**注解**(annotation),用于规范文档注释中的某些关键信息,例如程序作者、方法的参数或返回值等说明信息。注解是 Java 语言中一种特殊的接口类型。

1. 注解的定义和使用

Java 语法:**注解的定义和使用**

```
@Documented            //该语句需放在注解定义之前,其作用是指示 Javadoc 识别并提取注解信息
[public] @interface 注解名 {                                    //定义注解的语法
    数据类型 注解项 1()  [default 默认值];
    数据类型 注解项 2()  [default 默认值];
    … ;
}

@注解名 (注解项 1 = 注解内容, 注解项 2 = 注解内容, …)    //使用注解的语法
```

语法说明:

- 注解是一种特殊的接口类型,定义时在关键字 **interface** 之前加注解标记符"@"。
- 注解项是以方法成员的形式定义的,定义时可提供默认值。
- 使用注解时需在注解名前加注解标记符"@",并在小括号中给出各注解项的内容。有默认值的注解项可以缺省,缺省时将使用其默认值。
- 如果注解中只有一个注解项,使用时可省略"注解项=",直接给出注解内容。
- 为了指示文档生成工具 Javadoc 识别并提取注解中的信息,要求在注解定义前加"@Documented"进行说明。

例 5-25 给出一个定义和使用注解的钟表类 AnnoClock 示例代码。

例 5-25 一个定义和使用注解的钟表类 AnnoClock 示例代码(AnnoClock.java)

```
1  import java.lang.annotation.*;  //导入定义注解所需的类
2
3  @Documented                //@Document 表示下面的注解 Author 可以被 Javadoc 识别并提取
4  @interface Author {        //定义一个注解 Author,用于生成关于作者信息的说明文档
5      String value();
6  }
7
```

```
 8  @ Documented                          //@Document 表示下面的注解 Info 可以被 Javadoc 识别
 9  @interface Info {                     //定义一个注解 Info,用于生成关于版本和日期的说明文档
10      int version()  default 2;
11      String date()  default "2018/01/01";
12  }
13
14  /**
15   * 类的功能简介:AnnoClock 是一个钟表类,其中同时添加了文档注释和注解.
16   */
17  @ Author( "Kan" )                                  //使用注解 Author 来说明类的作者信息
18  @ Info (version = 1, date = "2018/08/20")          //使用注解 Info 来说明类的版本和日期
19  public class AnnoClock {                           //同时添加了文档注释和注解的钟表类
20      …                                              //省略部分代码,参见例 5-24
21      /**
22       * 方法 set():设置钟表时间,h-小时数,m-分钟数,s-秒数.
23       */
24      @ Author( "Tom" )
25      public  void set(int h, int m, int s)   //同时添加了文档注释和注解的方法
26      {  hour = h;  minute = m;  second = s;  }
27      public  void show()                            //未添加文档注释或注解的方法
28      {  System.out.println( hour + ":" + minute + ":" + second );  }
29      …                                              //省略部分代码,参见例 5-24
30  }
```

在 Eclipse 集成开发环境中为例 5-25 中的钟表类 DocClock 生成说明文档,然后使用浏览器查看其中的主页文件 index.html,查看结果如图 5-27 所示。

(a) 类AnnoClock的说明文档中增加了注解Author和Info信息

图 5-27 查看为例 5-25 钟表类 AnnoClock 自动生成的说明文档

(b) 方法成员set()的详细资料部分增加了注解Author信息

图 5-27 （续）

2. 注解的应用

除了用于生成说明文档之外，注解更主要的用途是在类定义代码中插入附加信息。这些附加信息可以被编译器用于检查语法错误，还可以在运行时向其他程序提供更多关于类的信息。

注解可以有不同的特性，例如注解是否需要被文档生成工具 Javadoc 识别并提取、注解可应用于什么程序元素、注解被保留到什么时候（源代码、字节码或运行时）等。Java 语言使用**元注解**（meta-annotation）来描述这些特性。元注解本身也是一种注解，被定义在 java.lang.annotation 包中。表 5-5 列出了 Java 语言中 5 个常用的元注解。

表 5-5 Java 语言常用的元注解（java.lang.annotation 包）

元 注 解	语 法	说 明
@**Documented**	@Documented	指定注解需要被文档生成工具 Javadoc 识别并提取
@**Repeatable**	@Repeatable(注解容器) 举例：定义一个可重复的注解 Schedule @Repeatable(**Schedules**.class) public @interface **Schedule** { … } public @interface **Schedules** { Schedule[] value(); }	指定注解可以对同一程序元素重复使用。编译重复注解时，Java 编译器会将重复的注解保存到一个注解容器中。程序员需要为重复注解另外定义一个注解容器，其中必须包含一个数组类型的注解项 value()
@**Target**	@Target(元素类型常量) 举例：注解仅用于类里的方法成员 @Target(ElementType.METHOD)	指定注解可应用于什么程序元素。元素类型常量可选择枚举类型 ElementType 中定义的枚举常量：ANNOTATION_TYPE、CONSTRUCTOR、FIELD、LOCAL_VARIABLE、METHOD、PACKAGE、PARAMETER、TYPE 或 TYPE_PARAMETER

续表

元 注 解	语 法	说 明
@**Retention**	@Retention(注解保留常量) 举例：注解仅保留在源代码中 @Retention(RetentionPolicy. SOURCE)	指定注解被保留到什么时候。注解保留常量可选择枚举类型 RetentionPolicy 中定义的枚举常量 SOURCE、CLASS 或 RUNTIME
@**Inherited**	@Inherited	指定超类中的注解可以被子类继承

3. Java API 预定义的注解

Java API 还在 java. lang 包中预定义了若干种常用的注解，主要用于向编译器提供附加信息。表 5-6 列出了其中 3 个常用的预定义注解。

表 5-6 Java 语言常用的预定义注解(java. lang 包)

注 解	语 法	说 明
@**Override**	@Override	表示重新定义超类继承来的方法，即覆盖超类方法。编译器在编译时将检查相关的语法错误，例如方法签名不一致等
@**Deprecated**	@Deprecated	表示类或类成员是早期(过时)版本，已被弃用，不建议继续使用。如果程序使用了过时版本，编译器在编译时将显示错误提示（warning）
@**SuppressWarnings**	@SuppressWarnings(错误列表) 举例：不提示“过时版本”错误 @SuppressWarnings("deprecation")	告知编译器不要显示错误列表中指定的错误提示信息

在例 5-24 中，钟表类 DocClock 重新定义了从超类 Object 继承来的方法 toString()。可以在方法 toString()的定义代码前加上预定义注解“@Override”，明确告知编译器该方法是一个重写的方法。例如：

```
/**
 * 方法 toString():将时、分、秒数据转成字符串格式(重写从 Object 继承来的方法)。
 */
@ Override
public String toString()          //同时添加了文档注释和注解的方法
{   return String.format("Clock@ %d: %d: %d", hour, minute, second);   }
```

本节习题

1. Java 语言没有形如(　　)的注释形式。

   A. //……　　B. /* …… */　　C. /** …… */　　D. //* …… */

2. 下列注释形式中，(　　)可以被 Java 文档生成工具 Javadoc 自动识别并提取。

   A. //……　　B. /* …… */　　C. /** …… */　　D. //* …… */

3. Java 语言中的注解是一种特殊的(　　)。

A. 类　　B. 接口　　C. 方法成员　　D. 字段成员

4. 使用注解时,注解名前需要添加的字符是(　　)。

A. @　　B. #　　C. *　　D. %

5. 下面的注解中,(　　)表示重写超类继承来的方法。

A. @Override　　B. @Documented

C. @Deprecated　　D. @SuppressWarnings

本章学习要点

- 熟练掌握 Java API 说明文档的阅读方法。
- 学习 Java API 的使用,例如数学类 Math、字符串类 String、基本数据类型的包装类、根类 Object 和系统类 System 等。
- 理解并掌握 Java 语言的 try-catch 异常处理机制。
- 理解泛型编程,并能通过 Java API 中的数据集合类实现动态数组、队列、堆栈、集合和映射等功能。
- 掌握 Java 语言文档注释和注解的基本用法。

本章习题

1. 编写程序。模仿 5.2.1 节的例 5-2,编写一个字符串类 String 的 Java 测试程序。

2. 编写程序。模仿 5.6.2 节例 5-10 的代码结构编写一个求平方根的 Java 程序。要求使用 try-catch 语句处理用户输入负数时的异常。

3. 编写程序。模仿 5.7.4 节的例 5-18,编写一个求学生平均成绩的 Java 程序。要求使用 Java API 中的数组列表类 ArrayList<E>来存储学生成绩。

4. 编写程序。继承 5.9.1 节例 5-24 中的钟表类 DocClock,定义一个手表类 DocWatch,并为其添加文档注释。在 Eclipse 集成开发环境中生成手表类 DocWatch 的说明文档,查看生成的文档说明结果。