

Java语言程序设计

配套教材由清华大学出版社出版发行

第5章 Java基础类库



中國農業大學

阚道宏

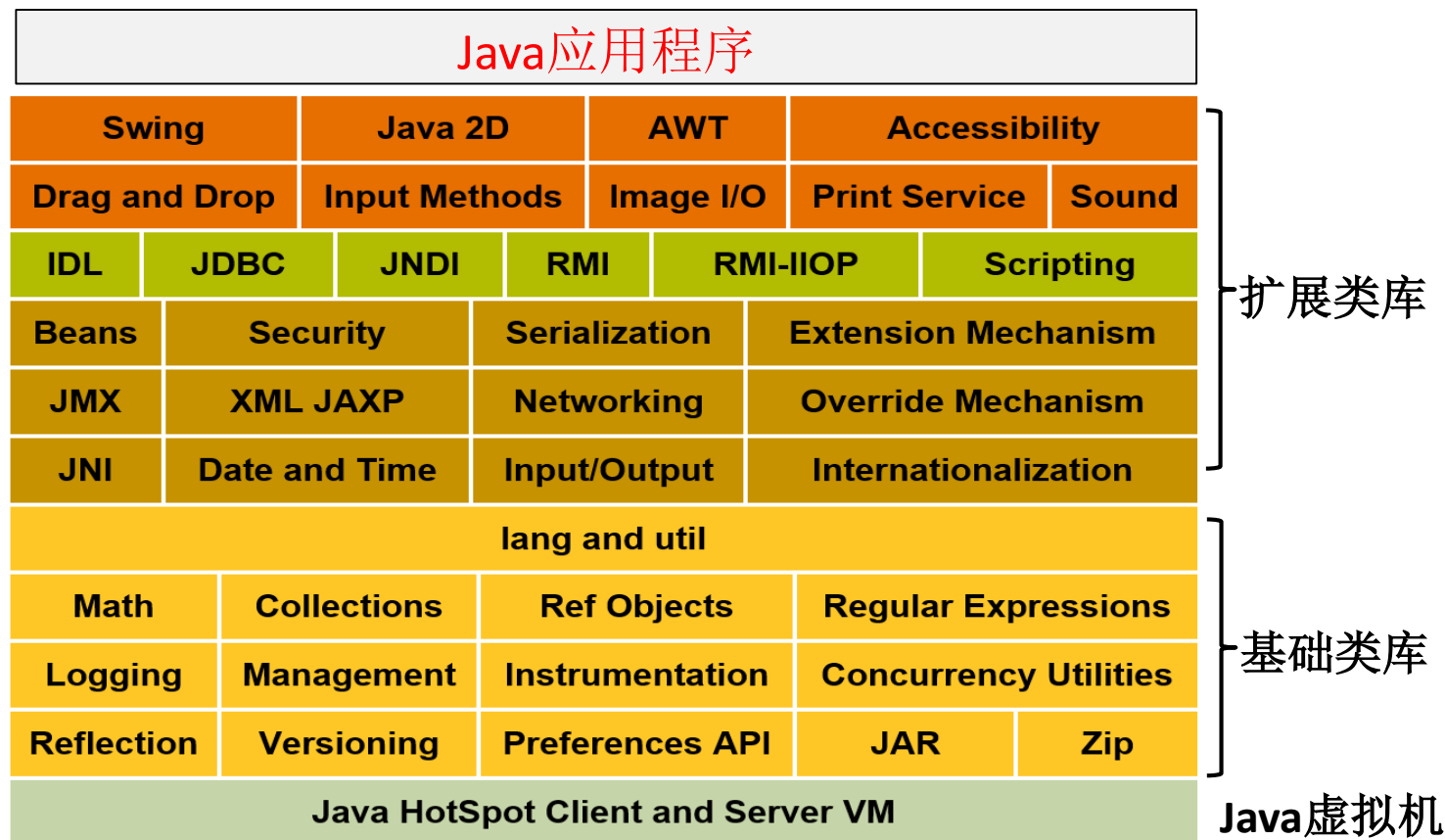
第5章 Java基础类库

- 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被统称为 **Java API**（Application Programming Interface）



第5章 Java基础类库

- Java API一览



中國農業大學

閻道宏

第5章 Java基础类库

- 类库：程序零件
- 应用开发：用已有的程序零件来**组装**软件
- 使用Java API类库中的类
 - 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？
 - 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？
- 逐步梳理清楚Java API类库的**整体脉络**
- Java API**说明文档**



第5章 Java基础类库

- 前4章学习了Java**基础语法**和**面向对象**程序设计方法
- 后续章节学习如何利用Java API进行应用开发
 - 学习阅读Java API**说明文档**的基本方法
 - 了解常见编程**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，把握Java API类库的**整体架构**
- 后续章节的学习要点
 - 继续积累Java知识
 - 培养**自学**能力
 - 最终能够**独立**开启自己的Java探索之旅



第5章 Java基础类库

- 本章内容
 - [5.1 数学类Math](#)
 - [5.2 字符串类](#)
 - [5.3 基本数据类型的包装类](#)
 - [5.4 Java语言的根类Object](#)
 - [5.5 系统类System](#)
 - [5.6 异常处理](#)
 - [5.7 泛型与数据集合类](#)
 - [5.8 枚举类型](#)
 - [5.9 Java源程序中的注释和注解](#)

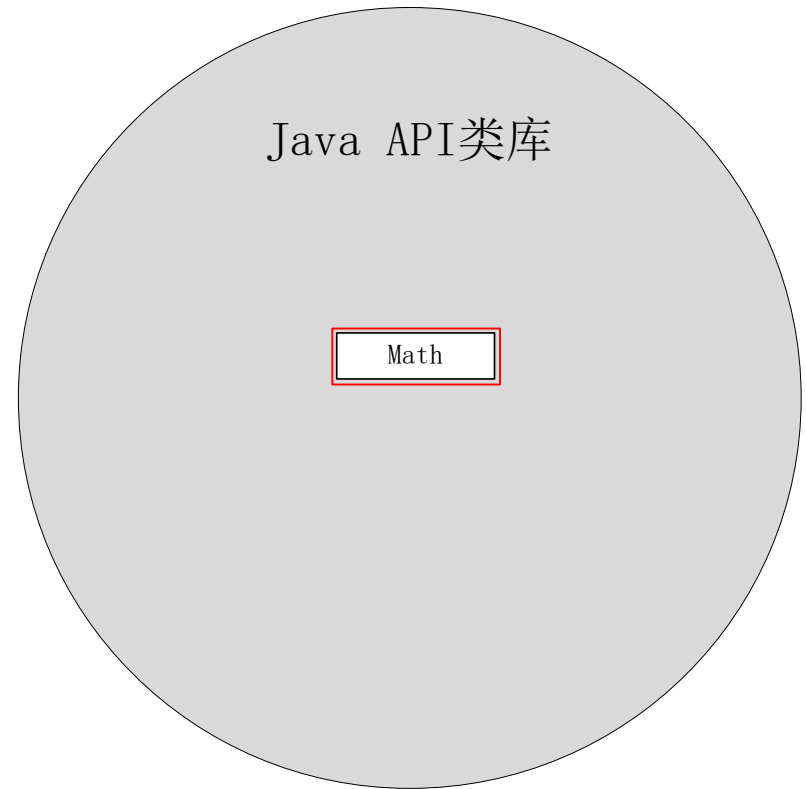


5.1 数学类Math

- 探索Java API类库
 - 从数学类**Math**开始
 - 从类的说明文档开始

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

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。



5.1 数学类Math

• 阅读类的说明文档

| | | | |
|---|----------|---------------------------------|----------------|
| java.lang.Math类说明文档 ← | | | |
| 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 |
| | static | int abs(int a) ← | 求int型整数的绝对值 |
| 3 | static | double abs(double a) | 求double型实数的绝对值 |
| 4 | static | double sin(double a) | 求正弦值 |
| 5 | static | double cos(double a) | 求余弦值 |
| 6 | static | double log(double a) | 求自然对数 |
| 7 | static | double log10(double a) | 求以10为底的对数 |
| 8 | static | double pow(double a, double b) | 求a的b次方 |
| 9 | static | double random() | 返回一个0到1之间的随机数 |
| 10 | static | double sqrt(double a) | 求平方根 |
| 11 | static | double toDegrees(double angrad) | 将弧度转为角度 |
| 12 | static | double toRadians(double angdeg) | 将角度转为弧度 |
| | | | |



5.1 数学类Math

- 类三个要点：全称、定义、成员

java.lang.Math

- 类名叫什么？
- 在哪个包里？
- 需要import导入“java.lang”语言包吗？
import java.lang.* ; // 导入Java语言包里的所有类

public final class Math

extends Object

- 这个类的访问权限是什么？
- 这个类可以被继承吗？
- 这个类继承了哪个类？



中國農業大學

阚道宏

5.1 数学类Math

- 从数学类的说明文档中可以读出什么信息？
 - 类的各个成员

int abs(int a)

- 这个方法成员的功能是什么，方法名、形参、返回值类型分别是什么？
- 这个方法是怎么编的，方法体是什么？没必要关心
- 这个方法功能是什么，怎么用？这才是应该关心的

double x = Math.**abs**(-8);

修饰符：**static**

- static是什么意思？static成员怎么用？
- 静态成员不需创建对象，直接用：Math.**PI**，Math.**abs**(-8)



中國農業大學

閻道宏

5.1 数学类Math

- 说明文档中的类成员都是什么访问权限？
 - **public**(未标注)
 - **protected**
- 说明文档没有包含**private**或默认权限的成员
 - 它们不是给Java程序员用的
 - 它们是被Java API隐藏的成员



5.1 数学类Math

- 测试程序

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 }
```

Problems @ Javadoc Declaration Console

<terminated> MathTest [Java Application] C:\Java\jre

8
4.0
1.0
180.0
0.2998906009136072
0.10413587602432794




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**String**: 存储和处理字符串**常量**
 - 可变字符串类**StringBuilder**: 适用于存储和处理字符串**变量**
 - 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



• 阅读类的说明文档

| | | | |
|---|--------|--|----------------------------|
| 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str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 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 |
| 12 | | String toLowerCase() | 将字符串中的大写英文字母改成小写 |
| 13 | | String toUpperCase() | 将字符串中的小写英文字母改成大写 |
| 14 | | String trim() | 去除字符串两头的空格 |
| 15 | static | String format(String format, Object... args) | 生成格式化字符串 |
| 16 | static | String valueOf(int i) | 将整数转成字符串 |
| 17 | static | String valueOf(long l) | 将长整数转成字符串 |
| 18 | static | String valueOf(float f) | 将单精度浮点数转成字符串 |
| 19 | static | String valueOf(double d) | 将双精度浮点数转成字符串 |
| 20 | static | String valueOf(char[] data) | 将字符数组转成字符串 |
| | | | |



5.2 字符串类

- 从字符串类的说明文档中可以读出什么信息？

java.lang.String

- 类名叫什么？
- 在哪个包里？
- 和数学类Math在同一个包？

public final class String

extends Object

implements Serializable, Comparable<String>, CharSequence

- 这个类的访问权限是什么？
- 这个类可以被继承吗？
- 这个类继承了哪个类？
- 这个类实现了哪些接口？



中國農業大學

閻道宏

5.2 字符串类

- 字符串类String的用法
 - 定义字符串对象

```
String s1 = new String( );  
String s2 = new String( "China中国" ); // 初始化  
String s3 = new String( s2 );           // 初始化
```

- 操作字符串对象

```
System.out.println( s2.length() );           // s2的长度为7  
System.out.println( s2.substring(1, 3) );    // 返回1~3之间的子串“hi”（不含3）
```

- 字符串可以相加

```
String s = s2 + “， 你好”;  
s += “， 你好”;
```



5.2 字符串类

- 字符串类String的使用
 - 修改字符串。String类字符串对象在改变内容时，不是改变对象自己，而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"; // 相加后，内存中会有2个字符串对象“abc”和 “abcef”  
           // 此时s改变指向，引用新对象“abcef”
```

- 比较两个字符串的大小

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



5.2 字符串类

- 字符串类String的使用
 - 静态方法**format**。这是String类提供的一个生成格式化字符串的方法，其用法非常象C语言里的printf()、sprintf()。例如，

```
int x = 5; double y = 16.8;
```

```
String s = String.format("x= %d, y= %5.2f", x, y);
```

```
System.out.println( s ); // 显示字符串s
```



5.2 字符串类

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() );
13        System.out.println( s3.indexOf("cd") );
14        System.out.println( s3.substring(1, 3) );
15        System.out.println( s3.concat("fg") );
16        System.out.println( s3 + "fg" );
17    } }
```

Problems @ Javadoc Declaration Console

<terminated> StringTest [Java Application] C:\Java\jre

```
x= 5, y= 16.80
0
4
ABCD
2
bc
Abcd cdefg
Abcd cdefg
```



5.2 字符串类

- 接口Comparable

- “可比较大小的” 接口

```
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

- <**T**>是泛型编程，将在后面讲解。

- 任何一个类都可以实现Comparable接口



5.2 字符串类

- 可变字符串类StringBuilder
 - 字符串类String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**）
 - StringBuilder类有**长度**（length，实际存储的字符个数）和**容量**（capacity，最多能存储的字符个数）两个概念
 - 当要保存的字符串长度大于容量时，StringBuilder类将自动增加容量（即自动分配更多的内存），StringBuilder类相当于是**可变长**的字符数组



5.2 字符串类

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（转成字符串） |
| 7 | | StringBuilder append (double d) | 追加一个实数（转成字符串） |
| 8 | | StringBuilder insert (int dstOffset, CharSequence s) | 插入一个字符串 |
| 9 | | StringBuilder insert (int offset, int i) | 插入一个整数（转成字符串） |
| 10 | | StringBuilder insert (int offset, double d) | 插入一个实数（转成字符串） |
| 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 () | 返回字符串容量 |



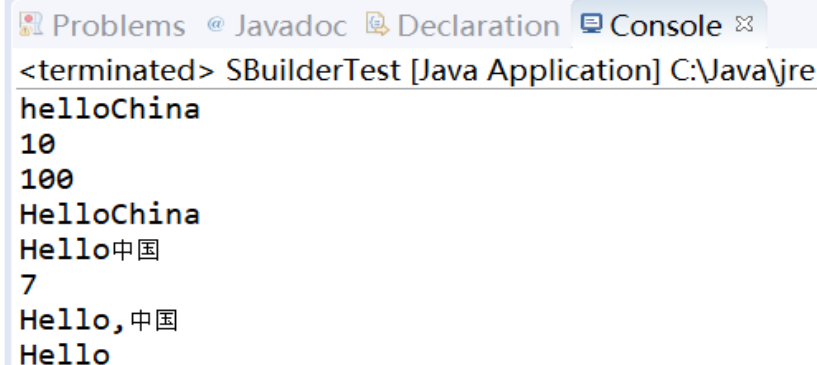
中國農業大學

閻道宏

5.2 字符串类

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         StringBuilder s = new StringBuilder( 100 ); // 字符容量为100个字符（单线程）
4         // StringBuffer 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之
12        System.out.println( s );
13        System.out.println( s.length() ); // 显
14        s.insert(5, ','); System.out.println( s );
15        s.delete(5, 8); System.out.println( s );
16    } }
```



The screenshot shows a Java IDE window with a tab labeled "Console". The output of the program is as follows:

```
<terminated> SBuilderTest [Java Application] C:\Java\jre
helloChina
10
100
HelloChina
Hello中国
7
Hello, 中国
Hello
```



中國農業大學

閻道宏

5.3 基本数据类型的包装类

- Java语言提供了8种基本数据类型
- Java语言还以类的形式将基本数据类型包装起来，这样的类被称为**包装类**（wrapper classes）

| | | | | | | | | |
|--------|------|-------|---------|------|-------|--------|-----------|---------|
| 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 |

- 包装类具有与数据类型相关的常量和处理方法
 - 常量主要包括数据类型的最大/最小值、占用字节数等
 - 方法主要包括比较大小、类型转换等
 - 这些包装类在使用方法上基本一样，以**Integer类**为例进行讲解



5.3 基本数据类型的包装类

| 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 |
| 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Integer转成int |
| 8 | | String toString() | 将Integer转成String |
| 9 | static | int parseInt(String s) | 将字符串转成int |
| 10 | static | Integer valueOf(int i) | 将int转成Integer |
| 11 | static | String toString(int i) | 将int转成字符串 |
| | | | |

● 阅读类的说明文档



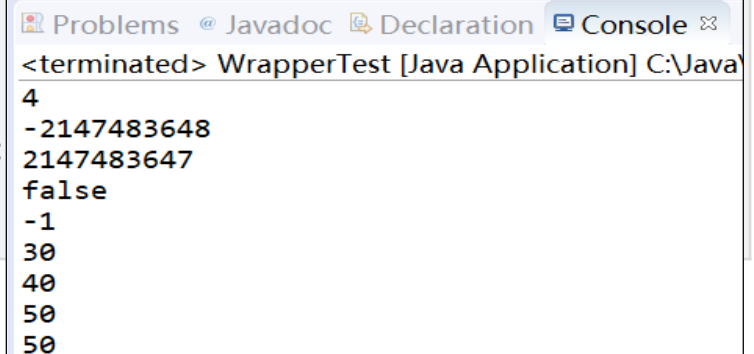
中國農業大學

閻道宏

5.3 基本数据类型的包装类

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
3         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 );
13        Integer iObj3 = Integer.valueOf( i+10 );
14        System.out.println( iObj3.toString() );
15        i = Integer.parseInt("50"); System.out.println( i );
16        System.out.println( Integer.toString(i) );
17    } }
```



The screenshot shows the console output of the WrapperTest program. The output is as follows:

```
<terminated> WrapperTest [Java Application] C:\Java\
4
-2147483648
2147483647
false
-1
30
40
50
50
```



5.3 基本数据类型的包装类

- 数值类Number

Byte、Short、Integer、Long、Float、Double

| | | | |
|-------------------------------------|-----|-----------------------------|------------|
| 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() | 转成byte类型 |
| 3 | | short shortValue() | 转成short类型 |
| 4 | | int intValue() | 转成int类型 |
| 5 | | long longValue() | 转成长类型 |
| 6 | | float floatValue() | 转成float类型 |
| 7 | | double doubleValue() | 转成double类型 |
| | | | |



5.4 Java语言的根类Object

- 探索Java API类库
 - 数学类Math
 - 字符串类String、StringBuilder
 - 8个基本数据类型的包装类
 - 数值类Number
- 超类： **Object**
- Object类被称为“**对象类**”，它是Java语言的**根类**（root class）



5.4 Java语言的根类Object

- 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



5.4 Java语言的根类Object

●
阅读类的说明文档

| 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回收对象前字段调用该方法，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() | 让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() | |



5.4 Java语言的根类Object

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：第4章4.1.1小节中的Clock类继承了Object类中的成员
11        Clock cObj = new Clock(8, 30, 15);
12        System.out.println( cObj );           // 显示引用
13        System.out.println( cObj.toString() ); // 转成字符串
14        System.out.println( cObj.getClass().getName() );
15    } }
```

Problems @ Javadoc Declaration Console

<terminated> ObjectTest [Java Application] C:\Java\jre

java.lang.Object@70dea4e
java.lang.Object@70dea4e
java.lang.Object
java.lang.Object

Clock@5c647e05
Clock@5c647e05
Clock



5.4 Java语言的根类Object

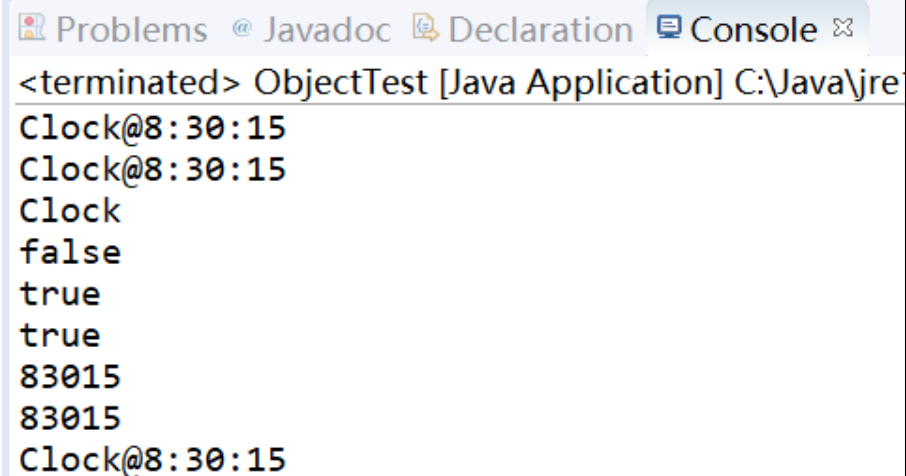
- 重写对象类Object的方法
 - 重写方法**toString()**，将对象转成可以理解的字符串
 - 重写方法**hashCode()**，将对象映射成一个int型整数，今后可用于快速比较两个对象的内容是否相对
 - 重写方法**equals()**，比较两个对象的**内容**是否相等。**注**：关系运算符“**==**”只能比较两个**引用**是否相等，即是否引用了同一个对象
 - 重写方法**clone()**，创建一个和当前对象内容一样的新对象（即**克隆**），并返回其引用。**注**：重写方法**clone()**的类需实现“可克隆的”接口**Cloneable**，否则**clone()**方法没有激活，不能使用
 - 重写方法**finalize()**，完成某些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



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 hour*10000 + minute*100 + second; } // 生成哈希码
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false
9         Clock c = (Clock)obj; // 将Object类型转成Clock类型
10        return c.hour== hour && c.minute==minute && c.second==second; // 比较时分秒
11        // return c.hashCode() == hashCode(); // 本例中也可以比较哈希码
12    }
13    public Object clone() throws CloneNotSupportedException // 重写方法clone ()
14    { Clock c = (Clock)super.clone(); return c; } // 克隆一个钟表对象
15    // 注：clone ()方法头后面的“throws ...”是Java语言的异常处理，将在后面讲解
16 }
```

```
1 public class ObjectTest { // 测试类 (ObjectTest.java)
2     public static void main(String[] args) { // 主方法
3         Clock cObj = new Clock(8, 30, 15);
4         System.out.println( cObj ); // 显示引用变量
5         System.out.println( cObj.toString() ); // 转成字符串，使用重写的toString
6         System.out.println( cObj.getClass().getName() );
7         // 下面演示如何比较两个钟表对象是否相等
8         Clock cObj1 = new Clock(8, 30, 15); // 新建对象
9         Clock cObj2 = cObj; // cObj2与cObj引用同一对象
10        System.out.println( cObj1 == cObj ); // 检查引用是否相等
11        System.out.println( cObj2 == cObj ); // 检查两对象是否指向同一内存地址
12        System.out.println( cObj1.equals(cObj) ); // 检查两对象是否相等
13        System.out.println( cObj.hashCode() ); // 显示cObj的哈希码
14        System.out.println( cObj1.hashCode() ); // 显示cObj1的哈希码
15        // 下面演示如何克隆一个钟表对象
16        try { // try-catch是Java语言的异常处理，将在后面讲解
17            Clock cObj3 = (Clock)cObj.clone(); // 克隆一个cObj对象
18            System.out.println( cObj3.toString() ); // 检查克隆后的对象
19        } catch(CloneNotSupportedException e) { };
20    }
```



Problems @ Javadoc Declaration Console

<terminated> ObjectTest [Java Application] C:\Java\jre

```
Clock@8:30:15
Clock@8:30:15
Clock
false
true
true
83015
83015
Clock@8:30:15
```

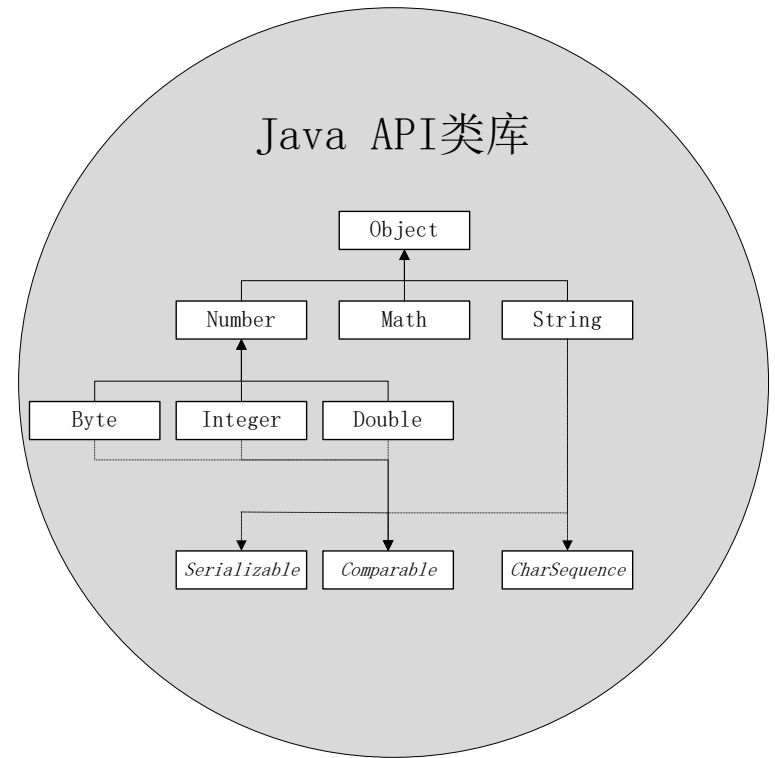
5.4 Java语言的根类Object

- “可克隆的” 接口Cloneable
 - 接口Cloneable未定义任何成员，是一个空接口
`public interface Cloneable; // 接口Cloneable的定义代码`
 - 这种未定义任何成员的空接口被称作**标记接口**（marker interface）
 - 类实现某个标记接口，其语法作用是为类**激活**（或称**启用**）某种功能
 - 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 Java语言的根类Object

- 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**
- 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**



5.5 系统类System

| 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 |
| | | | |



5.5 系统类System

- 系统类System的主要功能
 - 输入和输出
 - 系统类System定义了3个静态字段
 - 标准输入流对象in
 - 标准输出流对象out
 - 标准错误输出流对象err

`System.out.println("Hello, World");`

- **System**: 系统类的类名
- **out**: 系统类System的字段成员名（输出流类PrintStream的对象）
- **println**: 这是对象out包含的下级方法成员名
- System.in、System.out、System.err相当于C++语言里的cin、cout和cerr



5.5 系统类System

- 系统类System的主要功能
 - 读取或设置计算机系统属性
 - 相关属性的名称（称为key）

| Key | 说明 |
|-------------------|-------------------------------------|
| "java.class.path" | Java类库的搜索路径 |
| "java.home" | Java Runtime Environment (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 |

- **getProperty()**、**setProperty()**
System.out.println(System.**getProperty**("os.name"));



5.5 系统类System

- 系统类System的主要功能

- 复制数组方法**arraycopy()**

```
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);
```

```
System.out.println(y[0] + "," + y[1] + "," + y[2]);
```

```
显示结果： 2,3,4
```

- 读取系统时间方法**currentTimeMillis()**

```
System.out.println( System.currentTimeMillis() );
```

- **gc()**方法，请求JVM启动垃圾回收器（garbage collector），回收未被引用的对象的内存单元

- **exit()**方法，退出当前程序，并停止JVM的运行



中國農業大學

閻道宏

5.6 异常处理

- 程序中的错误可分为三种
 - 语法（syntax）错误
 - 语义（semantics）错误（或称为逻辑错误）
 - 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了描述不同异常情况的异常类



5.6 异常处理

- 三种不同的程序错误
 - 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) { 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         Cannot make a static reference to the non-static method Div(int) from the type ErrorSyntax.
15     } }
```



5.6 异常处理

- 三种不同的程序错误
 - 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     } }
```

2 <回车键>

100 ÷ 2 = 200

2 <回车键>

100 ÷ 2 = 50



中國農業大學

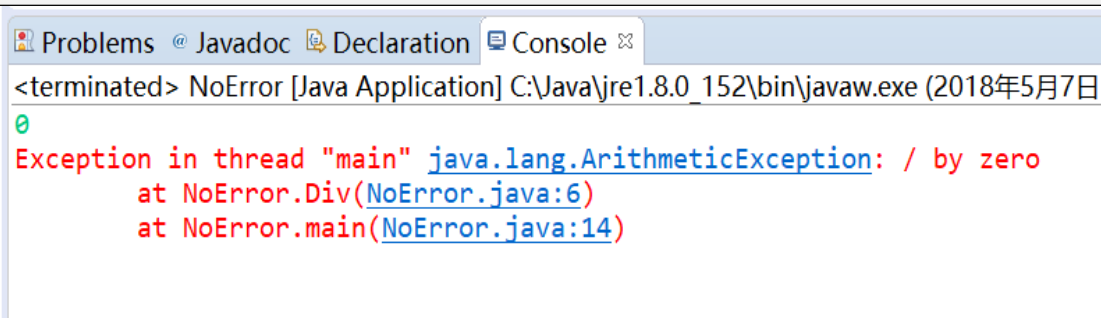
閻道宏

5.6 异常处理

- 三种不同的程序错误
 - 运行时错误

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
10         int N;
11         Scanner sc = new Scanner
12         N = sc.nextInt( );
13         int retValue = Div( N );
14         System.out.println( "100
15     } }
```



中國農業大學

閻道宏

5.6 异常处理

- Java语言的异常处理机制
 - 程序运行过程中常见的**异常**情况
 - **用户操作不当**
 - **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



5.6 异常处理

- Java语言的异常处理机制
 - 一个异常处理机制由3部分组成
 - **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。另外，Java API还为描述不同的异常情况专门提供了一组**异常类**
 - **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
 - **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

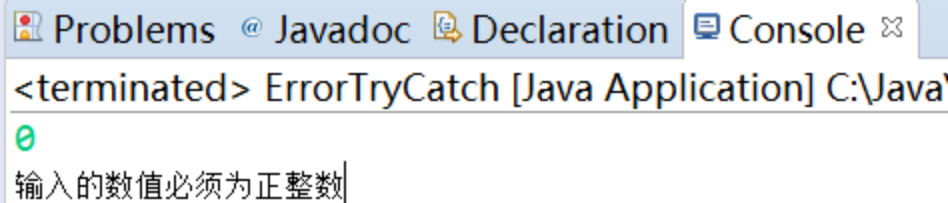


5.6 异常处理

- Java语言的异常处理机制

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
15         try { // 启用Java
16             int retValue = Div( N ); // 调用方法
17             System.out.println( "100÷" + N + "=" + retValue );
18         }
19         catch(RuntimeException e) // 捕获
20         { System.out.println( e.getMessage() ); }
21     } }
```



Problems @ Javadoc Declaration Console

<terminated> ErrorTryCatch [Java Application] C:\Java

输入的数值必须为正整数

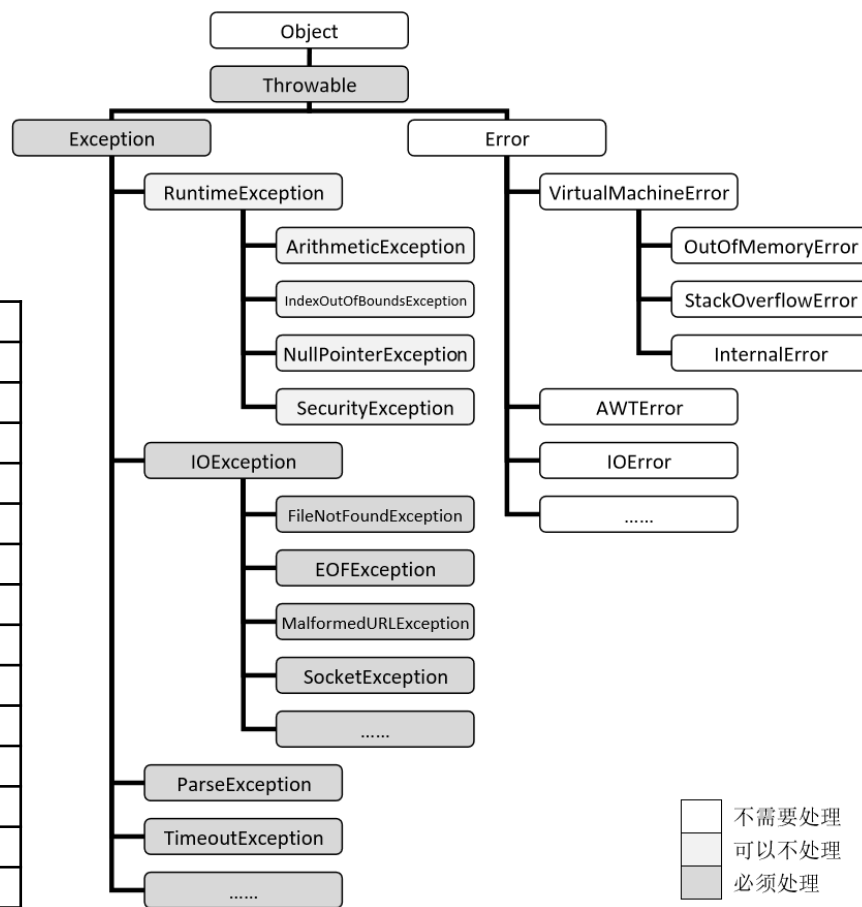


5.6 异常处理

- Java语言的异常处理机制
 - 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 | IO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 |



5.6 异常处理

- Java语言的异常处理机制

| | | | |
|--|-----|--|---------|
| 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 |
| | | | |



5.6 异常处理

- Java语言的异常处理机制

- throw语句

- 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
```

- 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
```

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

- 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



# 5.6 异常处理

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，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，我们称异常对象被**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 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
27 }
```

Problems @ Javadoc Declaration Console

<terminated> ThrowTest [Java Application] C:\Java\jre1.8.0\_152\bin\javaw.exe

choice: 1  
Before try-catch  
Before throw  
[java.lang.ArithmeticException: ArithmeticException](#)  
finally block  
After try-catch

Problems @ Javadoc Declaration Console

<terminated> ThrowTest [Java Application] C:\Java\jre1.8.0\_152\bin\javaw.exe

choice: 2  
Before try-catch  
Before throw  
[java.io.IOException: IOException](#)  
finally block  
After try-catch

Problems @ Javadoc Declaration Console

<terminated> ThrowTest [Java Application] C:\Java\jre1.8.0\_152\bin\javaw.exe

choice: 0  
Before try-catch  
Before throw  
After throw  
finally block  
After try-catch



中國農業大學

网络中心

## 5.6 异常处理

- Java异常处理的代码框架
  - Java程序的多级嵌套调用关系
  - 三种不同的异常处理流程
    - 由方法fun2()自己处理
    - 将异常交由上级方法fun1()处理
    - 多级异常处理链条

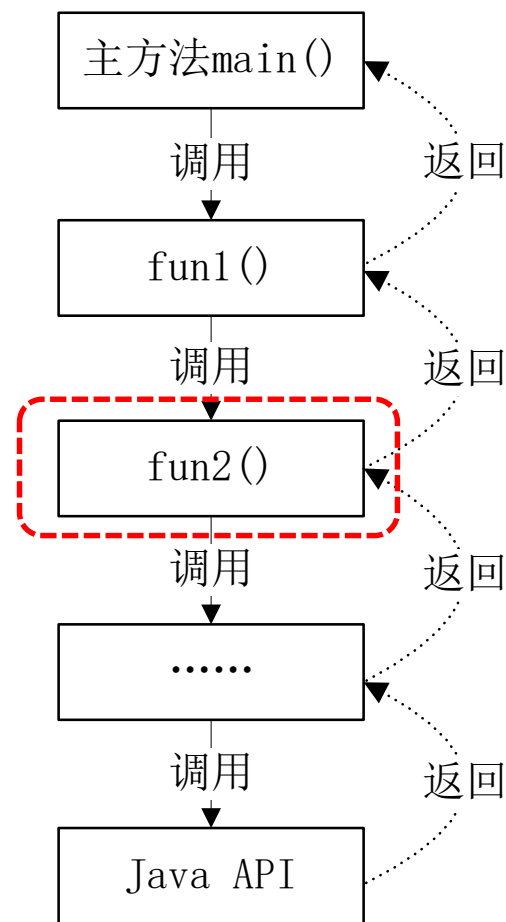
```
..... fun2() { // 方法 fun2()处理自己异常时的代码框架

 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

}
```



# 5.6 异常处理

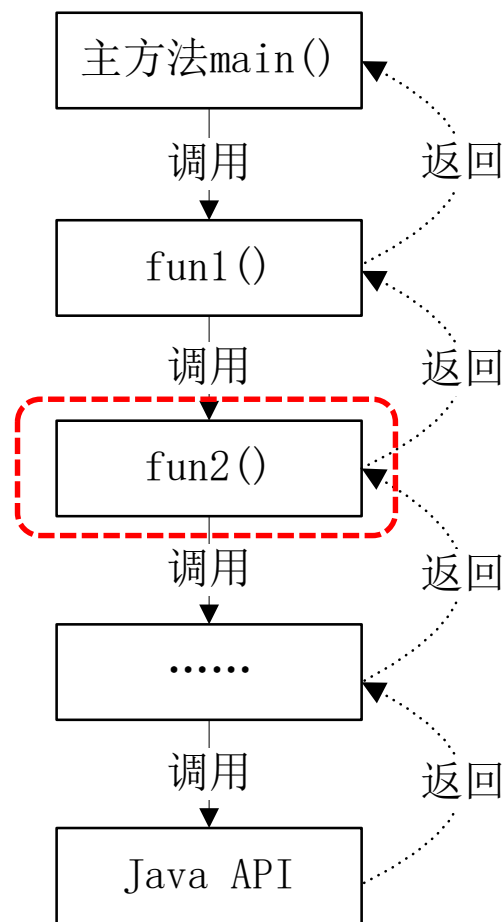
- Java异常处理的代码框架
  - Java程序的多级嵌套调用关系
  - 三种不同的异常处理流程
    - 由方法fun2()自己处理
    - 将异常交由上级方法fun1()处理
    - 多级异常处理链条

```
..... 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
.....
}
```



## 5.6 异常处理

- Java异常处理的代码框架
  - 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处理
    - 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



# 5.6 异常处理

- Java异常处理的代码框架
  - Java程序的多级嵌套调用关系
  - 三种不同的异常处理流程
    - 由方法fun2()自己处理
    - 将异常交由上级方法fun1()处理
    - 多级异常处理链条

```
..... fun1() { // 方法 fun1()在处理下级方法 fun2()所报告的异常后，继续向更上级的方法报告异常
.....
 try {

 fun2();

 }
 catch (... e) {
 异常处理代码
 throw e;
 // 或: throw
 }

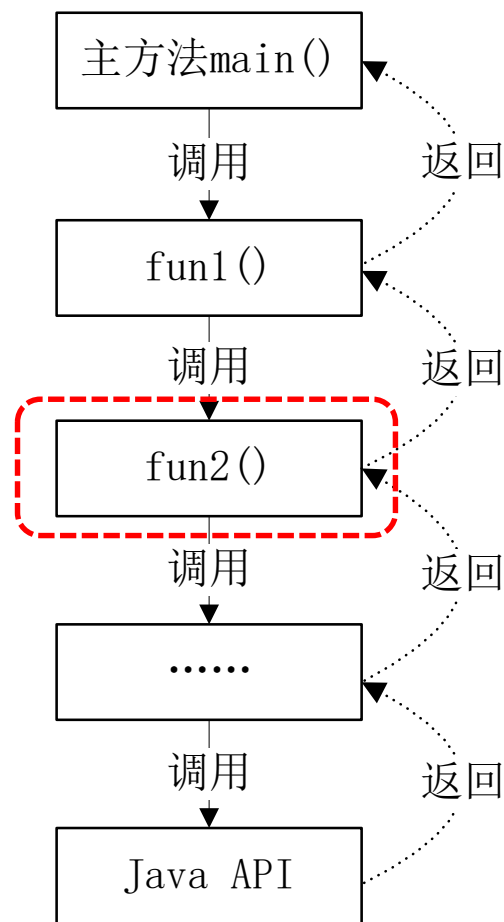
}

..... 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
 }

}
```



## 5.6 异常处理

- 三种不同性质的异常
  - 系统导致的异常
  - 程序员导致的异常
  - 用户导致的异常
- 系统异常
  - 系统异常类：Error类的子类
    - OutOfMemoryError、InternalError等
  - 程序员无法预见，另外也处理不了系统异常
  - 可以处理，也可以不处理





# 5.6 异常处理

- 三种不同性质的异常
  - 编程异常
    - 编程异常类：**RuntimeException**类的子类  
ArithmeticException、NullPointerException等
    - 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
    - 可以处理，也可以不处理



# 5.6 异常处理

- 三种不同性质的异常

- 用户异常

- 用户异常类：**Exception**类下除**RuntimeException**之外的其他子类  
IOException、FileNotFoundException、SocketException等
    - 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
    - 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，否则程序编译不能通过

- Java语法

- **勾选**（checked）异常：用户异常，**强制**处理
    - **非勾选**（unchecked）异常：系统异常、编程异常，**自愿**处理



# 5.6 异常处理

- 三种不同性质的异常
  - 勾选异常的处理
    - 勾选异常必须 “**捕捉或声明**” (Catch or Specify)

```
..... 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

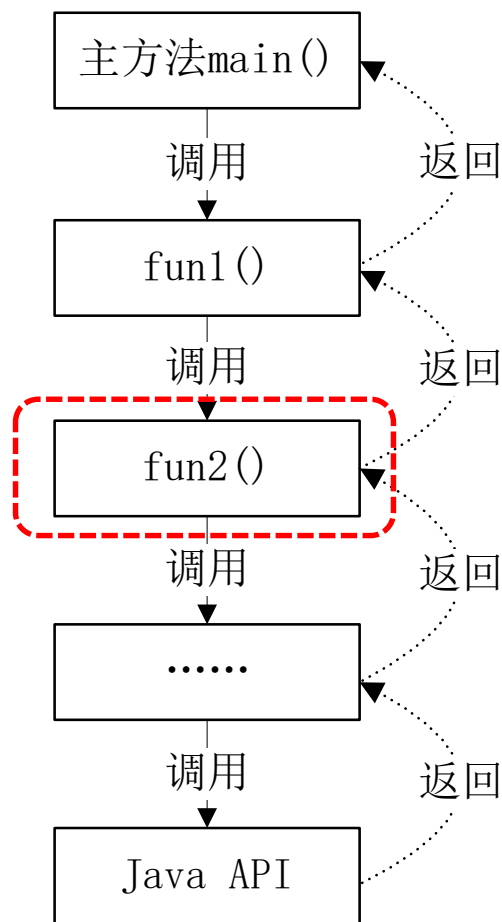
 if (发现勾选异常 A) throw eA; // 报告异常：抛出一个 A 类的勾选异常对象 eA
 fun2(); // 调用方法 fun2(), 调用过程中还可能会抛出一个 B 类的勾选异常对象 eB

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

..... 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

}
```



# 5.6 异常处理

- 自定义异常类

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 } }
```

- 选择从异常类Exception或运行时异常类RuntimeException继承
  - 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
  - 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



## 5.7 泛型与数据集合类

- 泛型（generics）
  - 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，我们称其代码可以被不同数据类型**重用**



# 5.7 泛型与数据集合类

- 类型参数化

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
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 } }
```

```
// 定义一个Integer型集合对象
Integer ia[] = { 10, 20, 30 };
IntegerSet is = new IntegerSet(ia);
is.show();
```

```
1 class DoubleSet { // Double型集合类
2 public Double set[]; // 用于存放Double型数据
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 } }
```

```
// 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();
```



# 5.7 泛型与数据集合类

- 类型参数化
  - Java语言可以将Integer、Double等具体数据类型抽象成一个**类型参数**（被称为类型形参），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**（被称为**泛型**）



# 5.7 泛型与数据集合类

- 类型参数化

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
2 public T set[]; // 用于存放数据
3 public GenericSet(T p[]) // 构造函数
4 { set = p; }
5 public void show() { // 显示数据
6 for (int n = 0; n < set.length; n++)
7 System.out.print(set[n] + " ");
8 System.out.println();
9 } }
```

```
Integer ia[] = { 10, 20, 30 };
GenericSet<Integer> is = new GenericSet<Integer>(ia);
// 或: GenericSet<Integer> is = new GenericSet<>(ia);
```

```
Double da[] = { 10.5, 20.5, 30.5 };
GenericSet<Double> ds = new GenericSet<Double>(da);
```

```
Short sa[] = { 10, 20, 30 };
GenericSet<Short> ss = new GenericSet<Short>(sa);
```

```
Float fa[] = { 10.5f, 20.5f, 30.5f };
GenericSet<Float> fs = new GenericSet<Float>(fa);
```

- **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





# 5.7 泛型与数据集合类

- 泛型编程
  - Java语言中
    - 带类型参数的类被称为**泛型类**
    - 带类型参数的接口被称为**泛型接口**
    - 带类型参数的方法称为**泛型方法**
  - 使用Java API中的泛型类、泛型接口或泛型方法编写程序。例如，**Java API数据集合类**
  - 如何编写自己的泛型类，即**泛型编程\*\*\***



# 5.7 泛型与数据集合类

## • 泛型编程

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）被称为具体类（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。



# 5.7 泛型与数据集合类

- 泛型编程

- 泛型集合类**GenericSet<T>**

```
GenericSet<Integer> is = new GenericSet<Integer>(...);
GenericSet<Double> ds = new GenericSet<Double>(...);
GenericSet<Character> cs = new GenericSet<Character>(...);
GenericSet<String> ss = new GenericSet<String>(...);
GenericSet<Object> os = new GenericSet<Object>(...);
```

- GenericSet<T extends Number>**

```
GenericSet<Number> ns = new GenericSet<Number>(...);
GenericSet<Integer> is = new GenericSet<Integer>(...);
GenericSet<Double> ds = new GenericSet<Double>(...);
```

```
GenericSet<Character> cs = new GenericSet<Character>(...); // 错误
GenericSet<String> ss = new GenericSet<String>(...); // 错误
GenericSet<Object> os = new GenericSet<Object>(...); // 错误
```



# 5.7 泛型与数据集合类

- 泛型编程

- 泛型族

- 类族

- 数值类Number的子类：Byte、Short、Integer、Long、Float和Double
      - 这些类构成一个以类Number为根类的**数值类族**

- 泛型族

- 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**合在一起就被称为是一个**泛型族**



# 5.7 泛型与数据集合类

- 泛型编程
  - 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 }
```

- 基于泛型类A<T>为数值类族中的类分别定义一个具体类  
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
- Java语言又引入通配符类型，这样可以让泛型族共用算法代码



## 5.7 泛型与数据集合类

- 泛型编程

- 泛型族：**通配符类型**的引用变量

- 使用泛型族中的具体类定义引用变量，或创建对象

- ```
A<Integer> ia;    // 定义A<Integer>类的引用变量ia
```

- ```
ia = new A<Integer>(10);
```

- 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语言也为**泛型族**专门设计了三种用问号“**?**”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**



# 5.7 泛型与数据集合类

- 泛型编程
  - 泛型族：三种通配符类型
    - 泛型名`< ? >`:
      - `A< ? > ref ;`
        - 引用变量`ref`可引用`A<Integer>`、`A<Double>`、`A<String>`、`A<Object>`等任何具体类的对象
    - 泛型名`< ? extends 类名 >`
      - `A< ? extends Number > ref ;`
        - 引用变量`ref`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
    - 泛型名`< ? super 类名 >`
      - `A< ? super Integer > ref ;`
        - 引用变量`ref`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



# 5.7 泛型与数据集合类

- 泛型编程

- 泛型族：通配符类型的**语法规则**

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。

- 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





# 5.7 泛型与数据集合类

- 泛型编程

- 泛型族共用算法代码

```
void show(A<Integer> aRef) // 具体类的形参
{ System.out.println(aRef.a); }
```

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

- 改用通配符类型的形参

```
void show(A< ? extends Number > aRef) // 通配符类型的形参
{ System.out.println(aRef.a); }
```

```
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
```



# 5.7 泛型与数据集合类

- 泛型编程

- 泛型类的继承与扩展

- 继承泛型类**A**<T>定义泛型类**B1**<T>、**B2**<T1, T2>

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 } }
```

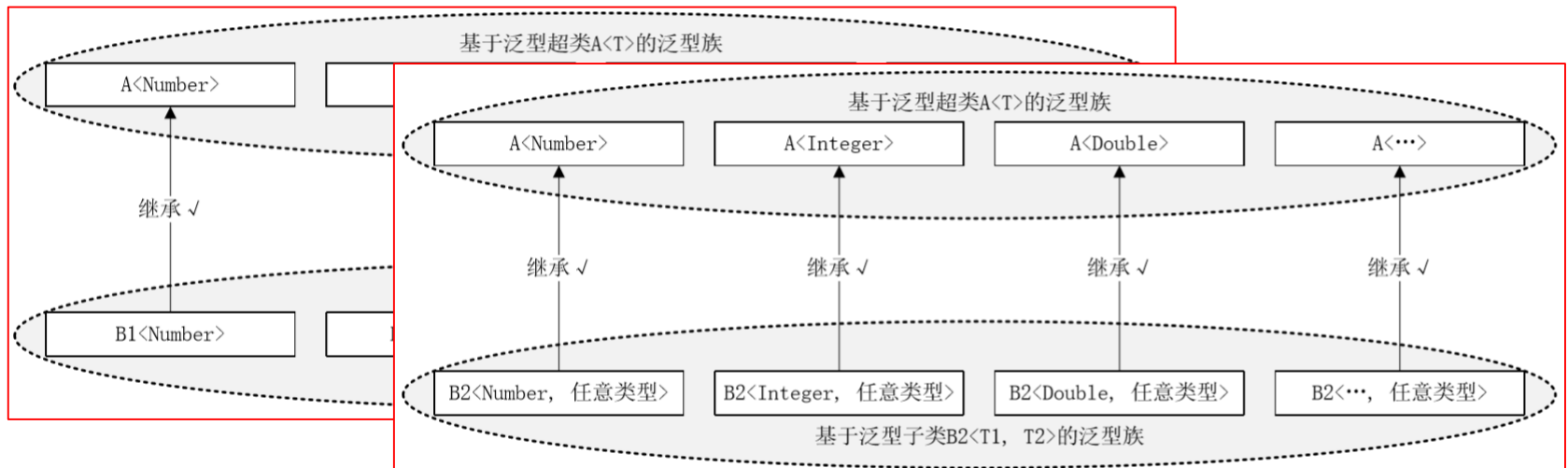


# 5.7 泛型与数据集合类

- 泛型编程

- 泛型类的继承与扩展

- 继承泛型类**A**<T>定义泛型类**B1**<T>、**B2**<T1, T2>
      - 类**B1**<T>泛型类族 与 类**A**<T>泛型类族
      - 类**B2**<T1, T2>泛型类族 与 类**A**<T>泛型类族



# 5.7 泛型与数据集合类

- 泛型编程

- 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

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

- 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

```
A.show(new Integer(5)); // 类型形参T指代的是Integer类型
```

```
A.show(new Double(5.5)); // 类型形参T指代的是Double类型
```



# 5.7 泛型与数据集合类

- 数据集合
  - **数据项**（Data Item）。数据项是数据集合中的最小单位。数据项也被称作字段（field）
  - **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。一个数据元素也被称作是一条记录（record）
  - **数据集合**（Data Set）。数据集合由多个并列的数据元素所组成。一个数据集合也被称作是一张表（table）
  - 有序、无序
  - 增查改删、排序
  - 数据结构

表5-4 学生成绩单

| 姓名    | 成绩    |
|-------|-------|
| 张三    | 92    |
| 李四    | 86    |
| 王五    | 95    |
| ..... | ..... |



# 5.7 泛型与数据集合类

- 数据集合

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", 80); // 添加数据元素
5 sa[1] = new Student("李四", 90);
6 sa[2] = new Student("王五", 70);
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 } }
10
11 class Student { // 学生成绩类
12 private String name; // 姓名
13 private int score; // 成绩
14 public Student(String p1, int p2) // 构造方法
15 { name = p1; score = p2; }
16 public String toString() // 重写toString()方法
17 { return String.format("%s: %d", name, score); }
18 public boolean equals(Object obj) { // 重写equals()方法
19 if ((obj instanceof Student) == false) return false; // 类型不同，则直接返回false
20 Student s = (Student)obj;
21 return name.equals(s.name); // 比较姓名是否相同
22 } }
```



# 5.7 泛型与数据集合类

- 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）
    - Java API还配套提供了相关的增查改删和排序算法
  - Java API的Collection（集合）和Map（映射）接口：接口族。同一接口族中的类可以共用算法代码
  - 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：泛型族。同一泛型族中的类可以共用算法代码



# 5.7 泛型与数据集合类

- Java API中的数据集合类

|                                                                       |         |                                    |              |
|-----------------------------------------------------------------------|---------|------------------------------------|--------------|
| java.util. <b>Collection</b> <E>接口说明文档                                |         |                                    |              |
| public interface <b>Collection</b> <E><br>extends <b>Iterable</b> <E> |         |                                    |              |
|                                                                       | 修饰符     | 接口成员（节选）                           | 功能说明         |
| 1                                                                     |         | boolean <b>add</b> (E e)           | 添加一个元素       |
| 2                                                                     |         | boolean <b>contains</b> (Object o) | 是否包含某个指定的元素  |
| 3                                                                     |         | boolean <b>hasNext</b> ()          | 是否还有下一个元素    |
| 4                                                                     |         | E <b>next</b> ()                   | 返回下一个元素      |
| 5                                                                     |         | void <b>remove</b> ()              | 删除迭代器当前指向的元素 |
| 6                                                                     |         | int <b>size</b> ()                 | 返回元素的个数      |
| 7                                                                     |         | boolean <b>isEmpty</b> ()          | 集合是否为空       |
| 8                                                                     |         | void <b>clear</b> ()               | 删除所有的元素      |
| 9                                                                     |         | Object[] <b>toArray</b> ()         | 返回一个数组形式的集合  |
| 10                                                                    | default | Stream<E> <b>stream</b> ()         | 返回一个流形式的集合   |
| 11                                                                    |         | Iterator<E> <b>iterator</b> ()     | 返回一个迭代器      |
| .....                                                                 |         |                                    |              |





# 5.7 泛型与数据集合类

- Java API中的数据集合类

| java.util. <b>Map</b> <K, V>接口说明文档 |         |                                             |              |
|------------------------------------|---------|---------------------------------------------|--------------|
| public interface <b>Map</b> <K, V> |         |                                             |              |
|                                    | 修饰符     | 接口成员（节选）                                    | 功能说明         |
| 1                                  | static  | interface <b>Map.Entry</b> <K, V>           | 内部接口，表示一个键值对 |
| 2                                  |         | V <b>put</b> (K key, V value)               | 添加一个键值对      |
| 3                                  |         | V <b>get</b> (Object key)                   | 读取某个键的值      |
| 4                                  | default | V <b>replace</b> (K key, V value)           | 修改某个键的值      |
| 5                                  | default | V <b>remove</b> (Object key)                | 删除某个键值对      |
| 6                                  |         | boolean <b>containsKey</b> (Object key)     | 是否包含指定的键     |
| 7                                  |         | boolean <b>containsValue</b> (Object value) | 是否包含指定的值     |
| 8                                  |         | int <b>size</b> ()                          | 返回键值对的个数     |
| 9                                  |         | boolean <b>isEmpty</b> ()                   | 映射是否为空       |
| 10                                 |         | void <b>clear</b> ()                        | 删除所有的键值对     |
| .....                              |         |                                             |              |

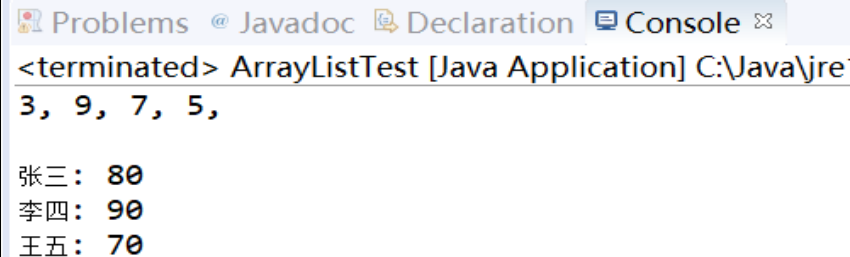


# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 数组列表类ArrayList<E>

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.print("\n\n");
9
10 ArrayList<Student> v2 = new ArrayList<>(
11 v2.add(new Student("张三", 80)); //
12 v2.add(new Student("李四", 90));
13 v2.add(new Student("王五", 70));
14 for (int n = 0; n < v2.size(); n++) //
15 System.out.println(v2.get(n));
16 } }
```



<terminated> ArrayListTest [Java Application] C:\Java\jre  
3, 9, 7, 5,  
张三: 80  
李四: 90  
王五: 70



## 5.7 泛型与数据集合类

- Java API中的数据集合类

- 数组列表类ArrayList<E>

- 使用**增强for语句**遍历数组列表

```
ArrayList<Integer> v1 = new ArrayList<Integer>();
v1.add(3); v1.add(9); v1.add(7); v1.add(5);
```

```
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



# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 数组列表类ArrayList<E>
    - 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提供了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**

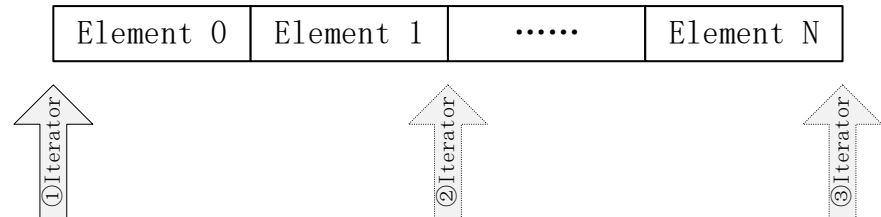


# 5.7 泛型与数据集合类

- Java API中的数据集合类

- 数组列表类ArrayList<E>

- 使用**迭代器**遍历数组列表



- 假设有一个如下的数组列表v1:

```
ArrayList<Integer> v1 = new ArrayList<Integer>();
v1.add(3); v1.add(9); v1.add(7); v1.add(5);
```

- 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
```

```
while (iv1.hasNext()) { // 如果有下一个元素则为true
 Integer e = iv1.next(); // 取出下一个元素
 System.out.print(e + ", ");
} // 当hasNext()返回false, 循环结束
```



# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 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

| java.util. <b>Iterator</b> <E>接口说明文档 |         |                                                           |               |
|--------------------------------------|---------|-----------------------------------------------------------|---------------|
| public interface <b>Iterator</b> <E> |         |                                                           |               |
|                                      | 修饰符     | 接口成员（全部）                                                  | 功能说明          |
| 1                                    |         | boolean <b>hasNext</b> ()                                 | 是否还有下一个元素     |
| 2                                    |         | E <b>next</b> ()                                          | 返回下一个元素       |
| 3                                    | default | void <b>remove</b> ()                                     | 删除迭代器当前指向的元素  |
| 4                                    | default | void <b>forEachRemaining</b> (Consumer<? super E> action) | 遍历并处理集合中剩余的元素 |

- Java API中的迭代器**Iterator**是一个泛型接口



# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 数组列表类ArrayList<E>

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v1进行排序
9 System.out.println(v1);
10 Collections.reverse(v1); // 对v1进行反转
11 System.out.println(v1);
12 System.out.println(Collections.max(v1));
13 } }
```

Problems @ Javadoc Declaration Console

<terminated> ArrayListTest [Java Application] C:\Java

```
[3, 9, 7, 5]
[3, 5, 7, 9]
[9, 7, 5, 3]
9
```



# 5.7 泛型与数据集合类

- Java API中的数据集合类

- 双端队列类LinkedList<E>

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和堆栈的功能。

- 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，这种排序规则被称为“先进先出”（First-In-First-Out，简称FIFO）

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，这种排序规则被称为“后进先出”（Last-In-First-Out，简称LIFO）

push(): 向堆栈中压入一个元素

pop(): 从堆栈中弹出一个元素



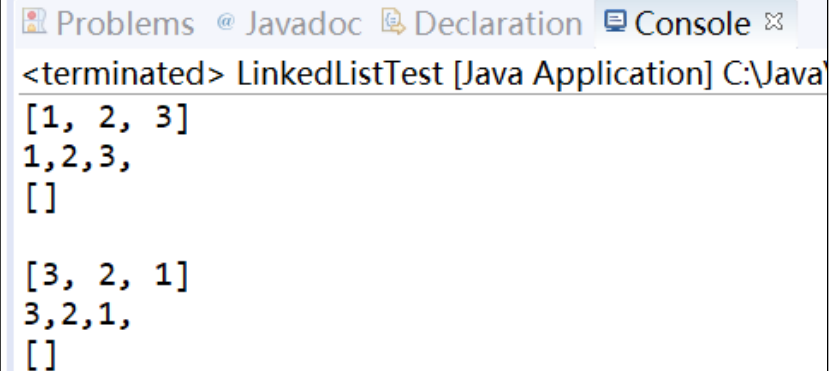


# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 双端队列类LinkedList<E>

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() + ","); // 取出并删除队首元素
12 System.out.println("\n" + q + "\n"); // 再次显示队列
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 } }
```



Problems @ Javadoc Declaration Console

<terminated> LinkedListTest [Java Application] C:\Java\

[1, 2, 3]  
1,2,3,  
[]

[3, 2, 1]  
3,2,1,  
[]

# 5.7 泛型与数据集合类

- Java API中的数据集合类

- 集合类HashSet<E>

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：**集合类通过数据元素所属类的方法成员equals()来判断两个元素是否重复

- 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



# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 集合类HashSet<E>

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
10 System.out.println(s.size()); // 显示元素个数
11 } }
```

Problems @ Javadoc Declaration Console

<terminated> HashSetTest [Java Application] C:\Java\

[1st, 3rd, 2nd, 4th, 5th]  
[1st, 3rd, 2nd, 5th]  
4



# 5.7 泛型与数据集合类

- Java API中的数据集合类

- 映射类HashMap<K, V>

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。

- “张三-92”就是一种“姓名-分数”键值对
    - “China-中国”就是一种“英文-中文”键值对

- 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**：映射类通过键所属类的方法成员equals()来判断两个元素的键是否重复
    - 映射没有迭代器，其中的元素也没有序号（下标）。如需遍历映射，可将映射或其键转成集合，再按集合的方式进行遍历



# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 映射类HashMap<K, V>

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
10 { System.out.println(k + " - " + h.get(k)); }
11 }
```

Problems @ Javadoc Declaration Console

<terminated> HashMapTest [Java Application] C:\Java\

李四 - 86  
张三 - 92  
王五 - 95



中國農業大學

閻道宏

# 5.7 泛型与数据集合类

- Java API中的数据集合类

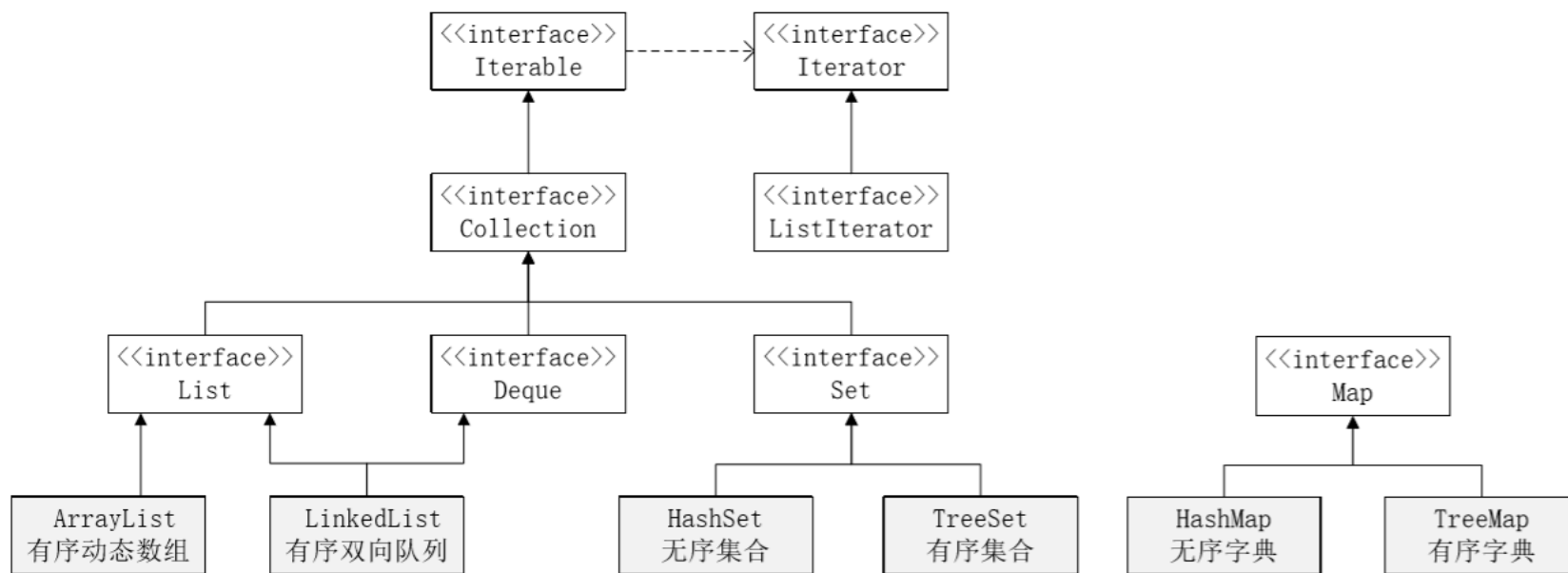


图 5-23 Java API 数据集合类的继承与实现关系



# 5.7 泛型与数据集合类

- Java API中的数据集合类
  - 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
  - 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）
  - 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



## 5.8 枚举类型

- 布尔（boolean）类型的值域：true和false
- 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被称为是一个**枚举常量**





# 5.8 枚举类型

- 定义枚举类型

```
public enum WeekDay {
 SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

- 使用枚举类型定义变量

```
WeekDay d = WeekDay.MONDAY;
```

- 枚举类型是一种特殊的类，自动继承枚举类Enum<E>

|                                                                                                           |     |                   |            |
|-----------------------------------------------------------------------------------------------------------|-----|-------------------|------------|
| java.lang.Enum<E extends Enum<E>>类说明文档                                                                    |     |                   |            |
| public abstract class Enum<E extends Enum<E>><br>extends Object<br>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() | 将枚举类型转成字符串 |
| .....                                                                                                     |     |                   |            |



## 5.8 枚举类型

- 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 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 } }
```

Problems Javadoc Declaration Console

<terminated> EnumTest (1) [Java Application] C:\Java\jre1.8.0\_152\bin\javaw.exe

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,

1 MONDAY

The day is not Weekend.

# 5.9 Java源程序中的注释和注解

- Java语言3种不同的注释形式
  - **单行注释**。以“//”开头，到所在行的行尾结束  
// 单行注释内容
  - **多行注释**。以“/\*”开头，以“\*/”结束  
/\*  
  注释内容，可以有多行  
\*/
  - **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  
/\*\*  
  文档注释的内容，可以有多行  
\*/



# 5.9 Java源程序中的注释和注解

- 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**
  - **Java API**就提供了全套的说明文档
  - 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。



# 5.9 Java源程序中的注释和注解

- 文档注释

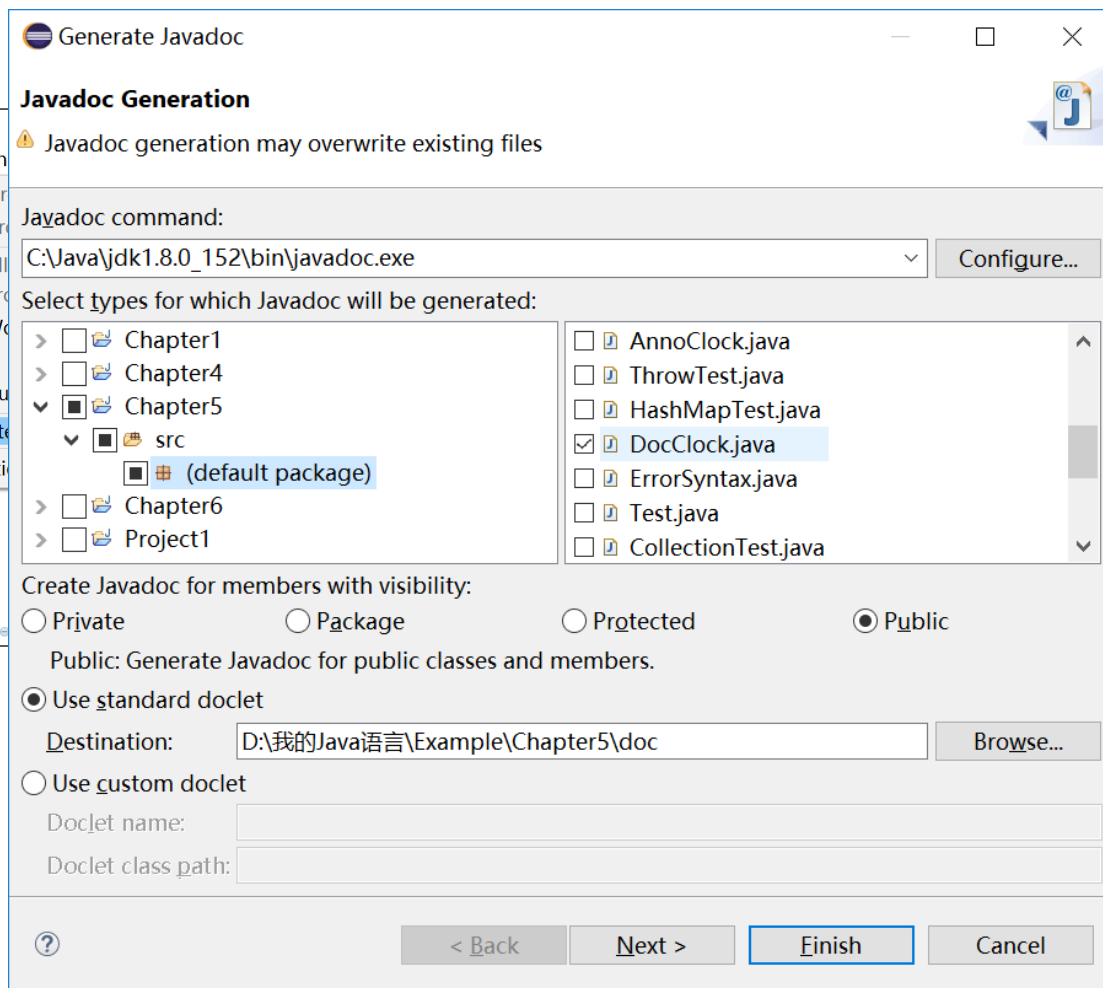
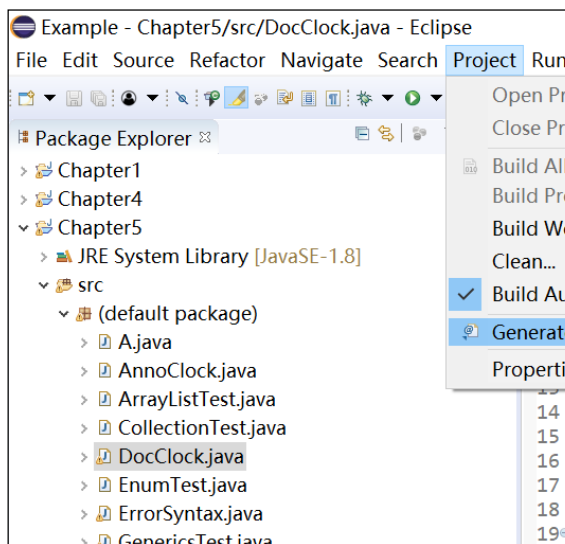
例5-24 一个添加了文档注释的钟表类DocClock示例代码（DocClock.java）

```
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 }
```



# 5.9 Java源程序中的注释和注解

- 文档注释



# 5.9 Java源程序中的注释和注解

- 文档注释

程序包 类 使用 树 已过时 索引 帮助

上一个类 下一个类 框架 无框架 所有类

概要: 嵌套 | 字段 | 构造器 | 方法 详细资料: 字段 | 构造器 | 方法

## 类 DocClock

java.lang.Object  
DocClock

```
public class DocClock
extends java.lang.Object
```

类的功能简介: DocClock是一个钟表类, 其中添加了文档注

### 构造器概要

#### 构造器

#### 构造器和说明

**DocClock**(int h, int m, int s)  
构造方法: 设置钟表时间, h-小时数, m-分钟数, s-秒数

JDK文档生成工具软件会从Java源程序文件中提取**类的信息**和**文档注释**。

### 方法概要

| 所有方法             | 实例方法                     | 具体方法                                        |
|------------------|--------------------------|---------------------------------------------|
| 限定符和类型           | 方法和说明                    |                                             |
| void             | set(int h, int m, int s) | 方法set: 设置钟表时间, h-小时数, m-分钟数, s-秒数。          |
| void             | show()                   |                                             |
| java.lang.String | toString()               | 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)。 |

### 从类继承的方法 java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait



中國農業大學

閻道宏

# 5.9 Java源程序中的注释和注解

- 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。

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。



# 5.9 Java源程序中的注释和注解

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 }
```



# 5.9 Java源程序中的注释和注解

- 注解

程序包 类 使用 树 已过时 索引 帮助

上一个类 下一个类 框架 无框架 所有类

概要: 嵌套 | 字段 | 构造器 | 方法 详细资料: 字段 | 构造器 | 方法

类 AnnoClock

java.lang.Object  
AnnoClock

```
@Author(value="Kan")
@Info(version=1,
 date="2018/08/20")
public class AnnoClock
extends java.lang.Object
```

类的功能简介: DocClock是一个钟表类, 其

## 方法详细资料

### set

```
@Author(value="Tom")
public void set(int h,
 int m,
 int s)
```

方法set: 设置钟表时间, h-小时数, m-分钟数, s-秒数。

### show

```
public void show()
```



中國農業大學

阚道宏

# 5.9 Java源程序中的注释和注解

- 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，另外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



# 5.9 Java源程序中的注释和注解

- 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                                                                                                                                            |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <b>@Documented</b> | @Documented                                                                                                                                                                              | 指定注解需要被文档生成工具javadoc识别并提取                                                                                                                     |
| <b>@Repeatable</b> | @Repeatable(注解容器)<br>举例：定义一个可重复的注解Schedule<br>@Repeatable(Schedules.class)<br>public @interface <b>Schedule</b> { ... }<br>public @interface <b>Schedules</b><br>{ 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()                                               |
| <b>@Target</b>     | @Target(元素类型常量)<br>举例：注解仅用于类里的方法成员<br>@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 |
| <b>@Retention</b>  | @Retention(注解保留常量)<br>举例：注解仅保留在源代码中<br>@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                                                                         |
| <b>@Inherited</b>  | @Inherited                                                                                                                                                                               | 指定超类中的注解可以被子类继承                                                                                                                               |



# 5.9 Java源程序中的注释和注解

- Java API预定义的注解

表5-6 Java语言常用的预定义注解（java.lang包）

| 注解                       | 语法                                                                            | 说明                                                       |
|--------------------------|-------------------------------------------------------------------------------|----------------------------------------------------------|
| <b>@Override</b>         | @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      |
| <b>@Deprecated</b>       | @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 |
| <b>@SuppressWarnings</b> | @SuppressWarnings(错误列表)<br>举例：不提示“过时版本”错误<br>@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                                  |

- 钟表类DocClock重新定义了从超类Object继承来的方法toString()

```
/**
```

```
 * 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); }
```



中國農業大學

閻道宏

# 第5章 Java基础类库

- 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语言中文档**注释**和**注解**的基本用法

