Java

第一章 Java语言基础

1.计算机程序语言经历了**机器语言， 汇编语言，高级程序语言**的发展历史。

2.Java是一种面向对象的程序设计语言。

3.Java支持**继承**，通过抽象类与接口支持**多态**，通过接口支持**多重继承**

4.Java不支持**指针。**

5.Java JDK , JRE, JVM

- JDK: java development kit, java开发工具包，用来开发Java程序的
- JRE: java runtime environment, java运行时环境
- JVM: java virtual machine，java虚拟机 用来解释执行字节码文件(class文件)的。

编写一次，到处运行

myProgram.java--->compiler-->**myProgram.class**--->JVM-->**my program**

6.java支柱:**异常处理机制，自动垃圾回收机制，“沙箱“运行模式，JVM**

```
public class Main {
    public static void main(String[] args) {
        System.out.println("this is a simple program!");
    }
}
```

7.java程序代码常用结构

package;

import;

public class defination;

class defination;

interface defination;

文件中含有主类，文件基本名必须以主类命名

```java
public class SimpleApp1 {
    public static void main(String[] args){
        SimpleApp1 simpleApp = new SimpleApp1();
        simpleApp.sayASentence();
        simpleApp.sayAnotherSentence();
    }
    public void sayASentence(){
        System.out.println("this is a simple program!");
    }
    public void sayAnotherSentence(){
        System.out.println("it is very easy to learn");
    }
}
```

注意：主方法带有static是静态方法

第二章 结构化程序设计

2.1标识符，关键字， 数据类型

1.java语言采用**unicode字符集**

2.标识符：以字母，下划线，或**$符号**开头的后面含有字母，下划线，或$符号和数字的字符串。大小写相关

3.标识符使用惯例：

- 类和接口：每个单词首字母大写 SimpleApp
- 方法/变量：首单词小写，后面单词首字母大写 processResult
- 常量：全部大写字母, PI

4.Java语言规定了48个关键字，都是小写

5.Java保留但已经不再使用的两个关键字： **const, goto**

6.数据类型：

数据

- 基本数据
    - 数值型:整数类型（byte,short,int,long） 浮点类型(float,double)
    - 字符类型

      单个字符被定义为char类型，必须用**单引号**括起来

      支持转义字符

把字符串定义为一个类----String类，必须用**双引号**括起来

- 逻辑类型

  两个取值：true false

  java语言中逻辑类型与整数类型不能直接转换

- 引用数据
  - 类
  - 接口
  - 数组

7.常量

- 文字常量： 3.14159
- 符号常量：PI　static final PI = 3.14159（用final修饰的)

8.变量

声明，创建，初始化

方法体以外的变量，系统可自动赋初值

方法体内声明的变量，需要语句赋初值

2.2操作符

1.赋值：基本类型赋的值在**栈**里，引用类型的值在**堆**里。

2.关系：equals与==　https://www.cnblogs.com/dolphin0520/p/3592500.html

3.逻辑：^ 异或：不相同为true,相同为false

&& 和 || 如果仅靠左边的运算数即可判定运算结果，右计算数将不被计算（条件与 条件或）

4.类型转换操作符：

自动转换：低到高：byte short int long float double

char ---> int

https://www.cnblogs.com/lk13227/p/13184034.html

强制转换：高到低

（type）expression

2.3表达式与语句

1.只有一个分号的语句为空语句，这种情况在if,for中

2.语句块：{}

2.4控制流程

1.顺序结构： 语句块1 --> 语句块2

2.选择结构:

(1)

if else

(2)

switch(expression){

case const1: statements; break;

.....

default: statements;

}

expression只能是整数类型或字符型

3.循环结构：

(1)while(boolean)

statement_or_block;

(2)do

statement_or_block

while(boolean)

(3) for(init_expr;boolean;alter_expr)

statement_or-block

break continue

- 不带label
- 带label：break/continue指定label的块

return

- return variable
- return ； 不返回任何数值，只向调用方法返回控制。

第三章 类与对象

对象是变量和相关方法的集合。

类是若干对象所具有的共性。

消息是**对象**之间的交互方式和交互内容。

面向对象的三大特性：**封装性，继承性，多态性**

3.1java类

1.类声明形式：

```
[public] [abstract|final] class classname
{
   classbody
}
```

2.变量成员与方法成员

变量成员声明格式：

```
[public|private|protected] [static] [final][transient][volatile] type
variableName;
```

变量类型可以是基本数据类型也可以是引用类型。

其名字在类成员中是唯一的，不能与其他变量同名。**但可以与类内其他某个方法使用同一个名字。**

变量成员的作用域为整个类。

方法成员声明格式：

```
[public|private|protected] [static] returnType methodName([paramList])
{
    methodbody
}
```

3.static属性

用static声明的变量成员为**类变量**，否则为**实例变量。**

类变量：多个对象共享一个内存区，共享同一个值。既可以通过对象实例名访问，也可以通过类名直接访问。

实例变量：不同对象各有各的内存区。只能通过对象实例名来访问。

用static声明的方法成员为**类方法**，否则为**实例方法。**

类方法：不能使用this或super，**不能访问实例变量，只能访问类变量。**

4.声明的作用域

Java语言的作用域分为类级、方法级、语句块级、语句级。

在类体中声明的变量成员和方法成员的作用域是整个类。
在方法中声明的参数和在方法中所有语句之外声明的变量的作用域是整个方法体。
在语句块中声明的局部变量的作用域是该语句块。
在语句中声明的变量的作用域是该语句。

5.访问控制符

public: 可以被任何其他的类访问。
protected: protected控制符修饰的成员能在它自己的类中和继承它的子类中被访问，也可以被同一包中的类访问。
defalut：类、方法和变量是可以被同一个包中任何一个类访问。
private:成员只能在它自己的类中被访问。

	同一个类中	**同一个包中**	**不同包中的子类**	**不同包中的非子类**
public	√	√	√	√
protected	√	√	√	
默认	√	√		
private	√			

6.构造方法

构造方法用来**初始化类对象**

方法名与类名相同，无返回值。

如果定义类没有定义构造方法，Java系统自动提供无参数的构造方法，任何类都有构造方法。

构造方法可以重载。
构造方法只能用**new关键字**调用。
若构造方法体定义为空，则自动调动其父类的构造方法

7.终结处理方法finalize()

负责回收无用对象占据的特殊的内存资源

3.2java对象

1.对象的创建

```
type objectName = new type([paramlist]);
```

type objectName声明了一个对象实例的引用。

String str=new String(“abc”);
栈 堆

2.对象的初始化

3.成员初始化顺序

- 先执行内部**静态对象**的构造函数，如果有多个按定义的先后顺序执行；
- 再执行**父类**的构造函数
- 再执行内部**普通对象**的构造函数
- 最后执行该类本身的构造函数

4.垃圾回收机制

垃圾收集的一个潜在的缺点是它的开销影响程序性能

3.3包：库单元

1.package语句

package语句放在Java源程序文件的第一行，**指明该文件中定义的类存放到哪个包中。**

2.import 语句

import语句的作用是引入所需的类。

import语句在程序中放在**package语句之后，类定义之前。**

3.4Java标准库类

第四章 类的多态与复用

java基于面向对象的三个主要特性：封装，继承，多态

4.1类的复用

1.组合

在一个对象里面使用了一些已有的对象，**使之成为新对象的一部分。**

新的对象通过控制这些已有对象达到复用这些对象的目的。

2.继承

被继承的类叫**超类，父类，基类**

继承超类的类叫**子类，派生类，扩展类**

```
class subclass-name extends superclass-name
{
   body of class
}
```

- 派生出来的子类进行继承时**不限于使用超类中原有的方法**，也可以在自己内部**定义属于自己的方法成员，定义属于自己的成员变量。**
- 子类包括父类的所有成员， **除了超类中被声明为private的成员。**
- Superclass和subclass类都是**完全独立的类。**

Java支持单重继承， 而**通过接口机制实现多重继承。**

3.重写与重载

重写：发生在继承类中

- **方法重写**：子类修改超类已有的方法叫方法重写。
-
  - 超类方法与被子类重写的方法的**参数列表与返回类型必须相同。**
  - 被子类重写的方法**不能拥有比超类更加严格的访问权限。**

-
- **变量重写：**
- 在子类中**定义**与父类已有的成员变量**使用相同的名字的变量成员**，称为变量成员的**覆盖或重写。**

- 如果新旧变量的类型相同，则新的变量被使用。
- 如果新旧变量的类型不同，则**根据类型，对应调用相应的成员。**

重载：发生在同一个类中

方法重载是指在一个类中，多个方法的方法名相同，**但是参数列表不同。**可以有不同的访问修饰符，不同的返回类型，不同的异常抛出。

- **在使用重载时只能通过不同的参数列表辨别。**

4.abstract与final

abstract修饰符：表示修饰的成分没有完全实现，还不能实例化。

- 如果在类的方法声明中使用abstract修饰符，标明这个方法是一个抽象方法，它**需要在子类实现。**
- 如果一个**类**的**包含抽象方法**，那么这个类是**抽象类**，**必须使用abstract修饰符**，并且**不能被实例化。**

final修饰符: final类不能被继承，因此**final类的方法成员**没有机会被覆盖，**默认都是fina**l的。

- final 变量： 用final修饰的成员变量表示常量，值一旦给定就无法改变。
- final 变量可以**先声明，再给初值。**

5.this和super

this(有参数/无参数)：用于调用本类对应相应的**构造方法。**

this.变量/方法：使用本类的变量成员或方法成员。

super(有参数/无参数)：用于调用父类的构造方法

super.变量/方法：(权限为protected/public):使用超类的变量/方法。

4.2多态

1.向上转型

子类转型成超类。通过**定义一个超类类型的引用指向一个子类的对象**，**既可以使用子类的强大功能，又可以抽取超类的共性。**

- 超类中的一个**方法**只有在**超类中定义**而在**子类中没有重写**的情况下，才可以被超类类型的引用调用。

- 对于超类中的方法，如果子类中重写了该方法，那么超类类型的引用将会将会调用子类中的这个方法，这就是**动态绑定。**

2.多态性：一个程序中不同方法共存的情况下，发送消息给某个对象，让对象自行决定响应何种行为方法。

当超类类型的引用指向一个子类的对象时，**由被引用的对象类型决定调用谁的成员方法。**但是这个方法必须是在超类中定义过的。

每个实例对象都自带一个**虚拟函数表**，这个表中**存储的是指向虚函数的指针**，实例对象通过这个表来调用虚函数，以实现多态。

3.运行时绑定

将一个方法调用和一个方法主体连接到一起称为绑定，**运行时绑定也叫动态绑定。**

- 在程序运行之前进行绑定，（由编译器和链接程序完成），静态绑定。
- 在程序运行期间进行绑定，运行时绑定。

在处理Java的成员变量时，采用的是静态绑定。

动态绑定针对的只是对象的方法。

4.多态的实现

继承实现的多态：

- 方法的重写
- 方法的重载

抽象类实现的多态

接口实现的多态

4.3接口

1.接口的定义：

接口是**常量**和**抽象方法**的集合。

```
[public] interface 接口名
{
   [public][static][final] 常量;
   [public][abstract] 方法;
}
```

接口可以**继承**，并且**支持多重继承**，子接口的成员集合是各个父接口成员集合的并集

```
[public] interface 接口名 extends 父接口名列表
{
   [public][static][final] 常量;
   [public][abstract] 方法;
}
```

接口可以在类中实现，类实现接口的过程就是**给出接口的方法**，所需要实现的方法是接口列表中**所有**接口中的方法。

```
[访问修饰符] class 类名 implements 接口名列表
{
  变量成员定义;
  方法成员定义和方法实现;
}
```

2.接口的特征：

- Java接口中的成员变量默认都是public,static,final,(可忽略),但是必须被**显示的初始化**，即接口中的成员变量为常量。
- java接口中的方法默认都是public abstract类型的，**没有方法体**。
- 接口中无构造方法，**不能被**直接**实例化**。
- 一个接口不能实现另一个接口，但是可以**继承多个其他多个接口**。
- 一个类只能继承一个直接的超类，但是可以实现多个接口，**间接的实现了多重继承**。
- java接口必须**通过类实现**。
- 当类实现了某个Java接口时，它必须实现接口中的所有抽象方法，**否则这个类必须声明为抽象**。
- 不允许创建接口的实例，但是允许定义接口类型的引用变量，**该引用变量引用实现了这个接口的类的实例**。

3.接口与抽象类对比

抽象类是部分抽象的，接口是纯抽象的。

4.4内部类

1.**内部类**的定义

一个类的定义放在另一个类的内部

- 嵌套类：静态的
- 成员内部类： 非静态的

内部类**可以修饰为public,default,protected,private**,

也可以是**静态static**的，即**嵌套类**。

创建内部类的对象：

```
//内部类
outerClass outerObject = new outerClass(parameters)
outerClass.innerClass innerObject = outerObject.new innerclass(parameters) //+命名空间

//嵌套类

outerClass.innerClass innerObject = outClass.new innerClass(parameters)
```

局部内部类：

定义在一个方法中或者一个代码块中。

- 访问局部内部类必须要有外部类对象
- 局部内部类只能访问外部类的`final类型`的局部变量

匿名内部类：

需要一个内部类不需要它的名字

```
new interfacename(){};

new superclass(){};
```

2.内部类与外部类

内部类访问外部类：

成员内部类：

可以在不需要任何特殊条件的情况下访问外部类的所有成员。

第五章 异常处理

5.1异常概述

异常，即异常事件，只程序在运行过程中出现的影响正常程序流程的事件。

java的异常处理机制体现了java语言的**鲁棒性**

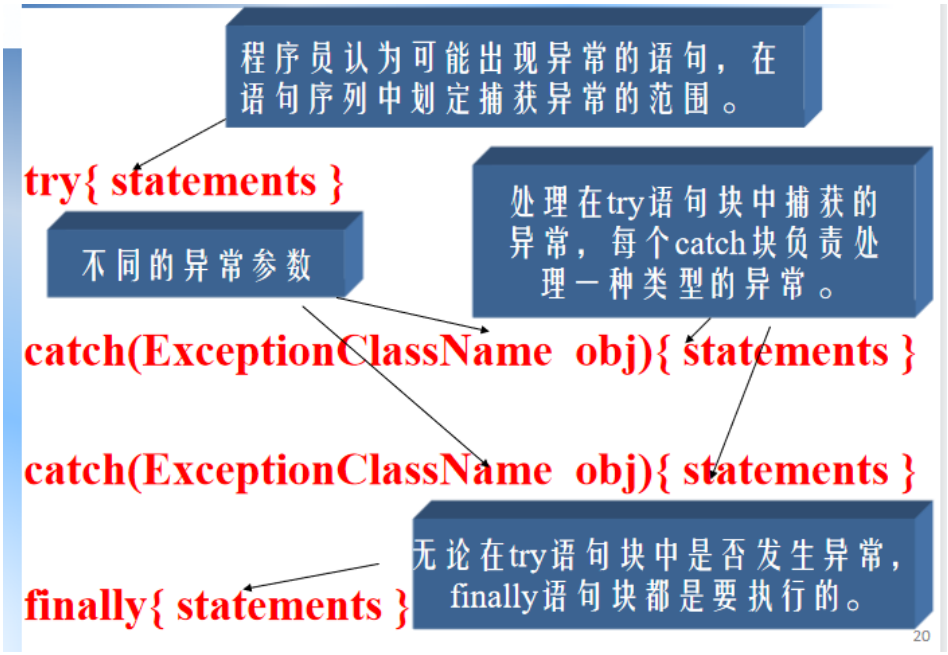
异常和异常处理是java语言**特有的**

异常事件分为两种：

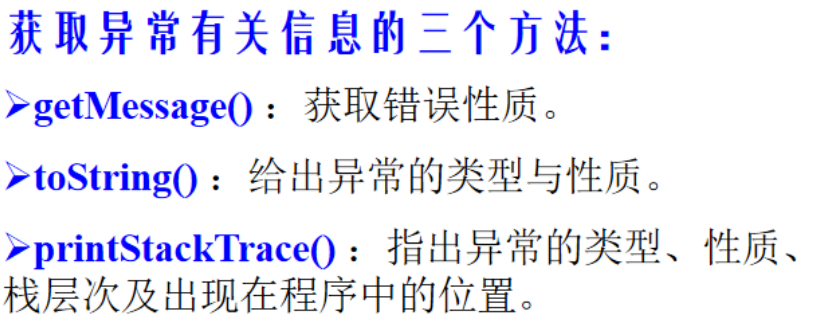
- Exception(异常):非致命的，经过处理可以不中断程序的执行。如输入输出异常，运行时异常
- Error（错误）：致命的，将使程序中断执行而退出系统。如虚拟机错误，内存溢出错误

5.2异常处理方法

捕获和处理异常



多个catch块时候，**只会匹配其中一个异常类**并执行catch块代码，而不会再执行别的catch块，并且匹配catch语句的顺序是**由上到下。**



抛出异常

Java语言也允许指明出现的异常不在当前方法内处理，而是**将其抛出**，送交到**调用它的方法**来处理，在调用序列中逐级**向上传递**，乃至**传递到Java运行时系统**，直至**找到一个运行层次可以处理它**为止。

关键字：

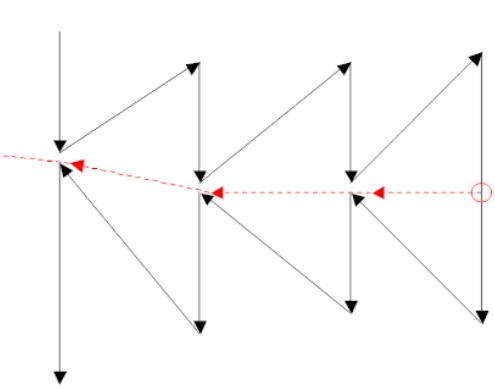
throw:出现在**方法体的内部**，是一个**具体的执行动作**，作用是**抛出一个具体异常对象。**

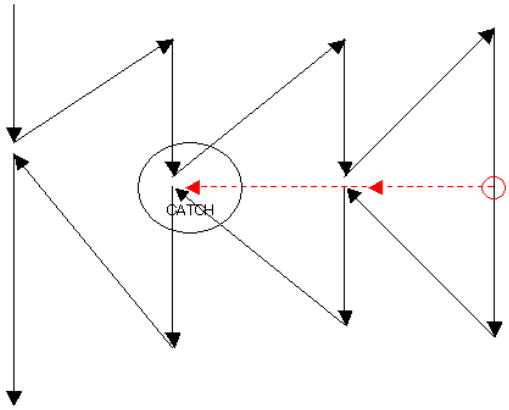
throws:在方法声明时放在**方法头**中的，作用是**声明一个方法可能抛出的所有异常。**

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

```
//运行结果
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

3.异常传递链





如果在方法中产生了异常，方法会在抛出异常的地方退出；

如果不想终止方法，那就需要在特定的区域用try来捕获异常。

5.3自定义异常

可以在Java类库中**现有异常类的基础上**由用户**创建新的异常类**，新的异常类必须用extends子句声明**是Exception类的子类**

```
public class MyException extends Exception{
    public MyException(String ErrorMessage){
          super(ErrorMessage);
    }
}
```

定义异常类的代码写在程序中，与其他类定义并列成为程序的一部分，在使用时与已有的异常类基本相同，只是在throw子句中使用下面的语法:

```
throw new MyException( );
```

第六章 对象的容纳

6.1数组

数组元素的访问： 数组名[索引值]

数组中除了数组元素之外，还存在唯一一个可被访问元素length。下标从0开始编号到a.length-1

数组的声明

两种形式：

- 类型 [] 数组名；
- 类型 数组名 []；

类型可以是**基本类型**也可以是**引用类型。**

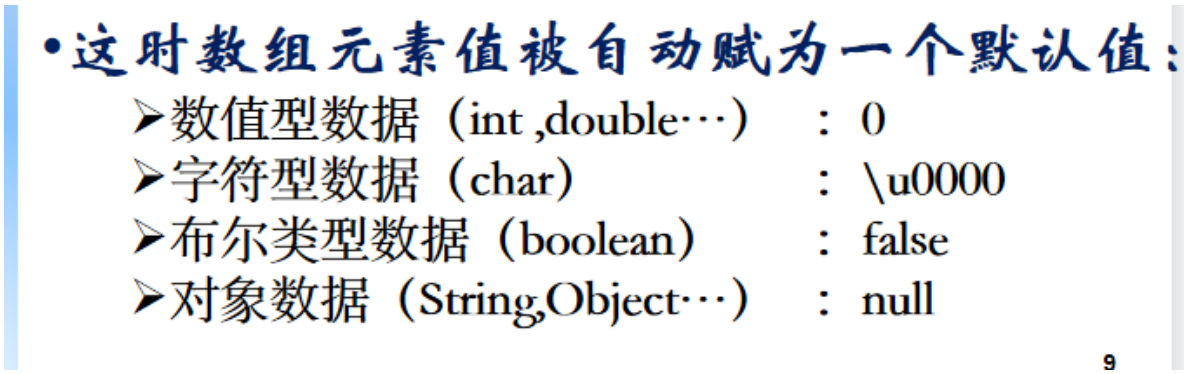
数组初始化

静态初始化

```
int a[] = {21, 34, 7, 8};
int [] a = new int [] {1,2,3,4};
```

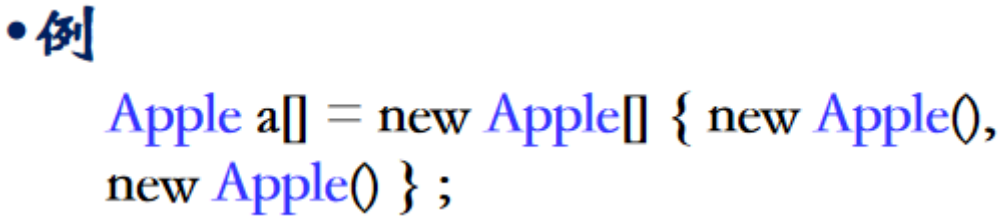
动态初始化

```
int [] a = new int [10]; //长度为10
```



对象数组初始化

类型[] 数组名=new 类型 [] {
new构造方法()，…， new构造方法()}



多维数组

声明方式：

int [] [] a;

int a [] [];

静态初始化：

int [] [] a = {{1,2,3},{1,2,3}};

int [] [] a = new int [] [] {{1,2,3},{1,2,3}};

动态初始化：

int a [] [] = new int [2] [3];

java语言的不规则数组：

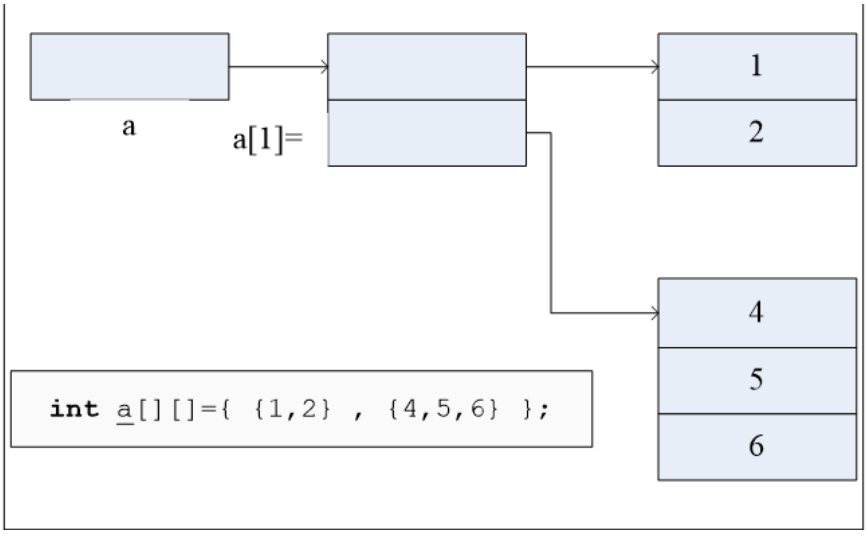
```
int[][] a = new int[2][];
a[0]=new int[2];    a[1]=new int[3];
```

多维数组的length:

a.length指**第一维数组**的长度
a[0].length数组中第1个元素**引用的数组的长度**

数组与数组的引用

Java数组是特殊的对象，数组变量存放一个数组对象的引用



数组工具类

◆Arrays类有一套static方法，提供了操作数组的实用功能

方　法	描　述
copyOf()	将一个数组中的值拷贝到新的数组中
sort()	将一个数组中的值进行排序，默认是升序排列
binarySearch()	在已排好序的数组中查找特定值
equals()	判断两个数组是否相等
asList()	将数组重构为列表

对象比较接口

自己实现函数

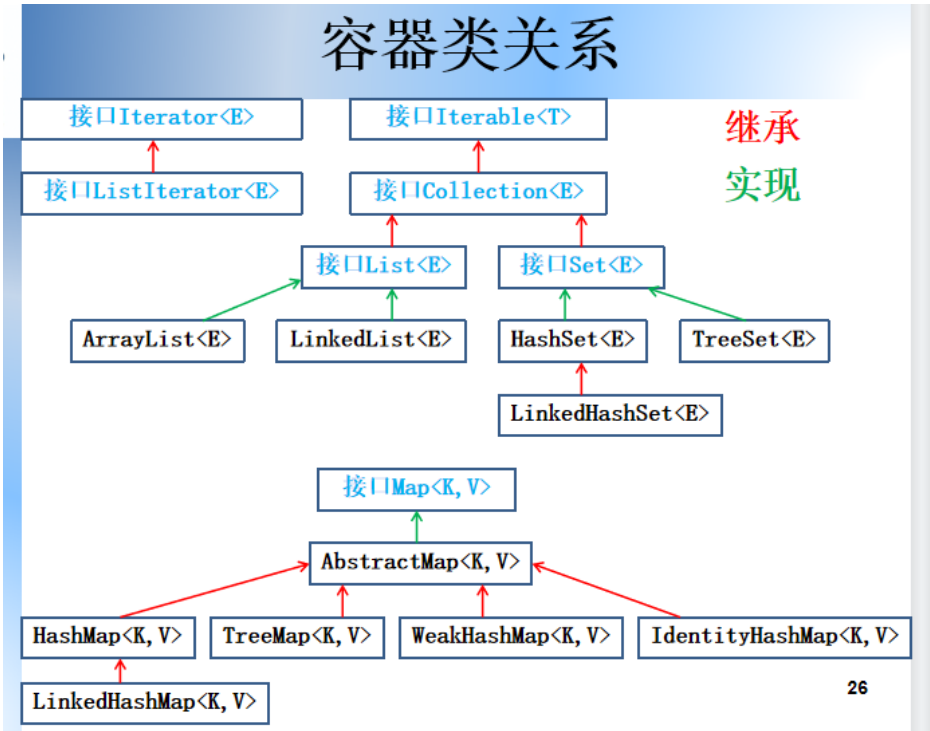
sort（）函数比较时会自动调用相关函数

6.2枚举

创建枚举类型使用enum关键字

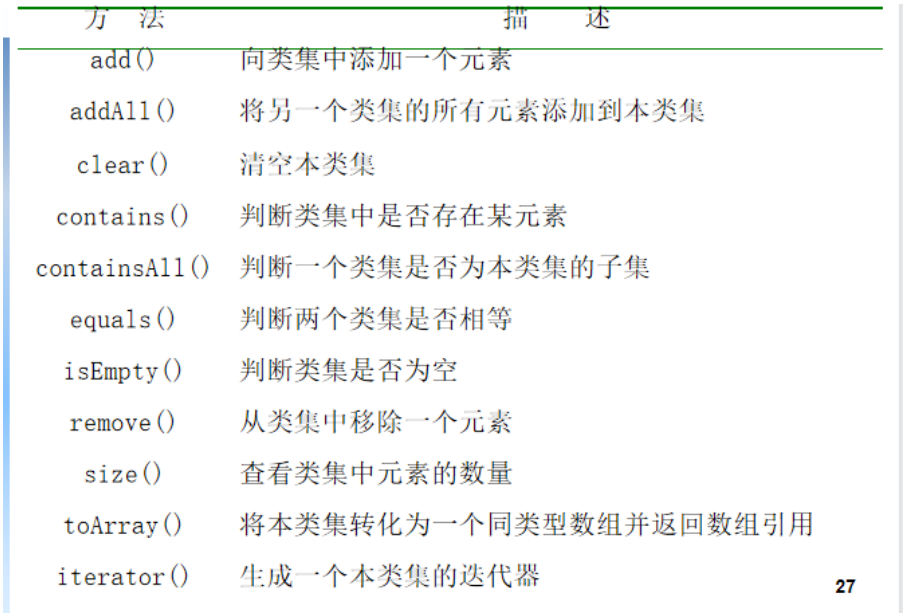
```
enum Season { SPRING, SUMMER, FALL, WINTER};
Season s = Season.SPRING;
System.out.println(s);

//输出
SPRING
```

6.3容器



collection接口

collection:一组单独元素集合



方　法	描　述
add()	向类集中添加一个元素
addAll()	将另一个类集的所有元素添加到本类集
clear()	清空本类集
contains()	判断类集中是否存在某元素
containsAll()	判断一个类集是否为本类集的子集
equals()	判断两个类集是否相等
isEmpty()	判断类集是否为空
remove()	从类集中移除一个元素
size()	查看类集中元素的数量
toArray()	将本类集转化为一个同类型数组并返回数组引用
iterator()	生成一个本类集的迭代器

27

List(列表)

List是Collection的子接口，是有序的Collection

方　法	描　述
add()	重载的add()方法，可在指定位置处插入元素
remove()	重载的remove()方法，可在指定位置处删除元素
get(int)	获取指定位置的元素
set()	重设指定位置的元素的值
indexOf()	获取指定元素的位置
lastIndexOf()	获取指定元素的最后一次出现的位置

28

```
List a = new ArrayList();
//List a = new LinkedList();
a.add( "cat" );
a.add( "dog" );
a.add( "cat" );
a.remove(0);
for(int i=0; i<a.size(); i++) { // a.size()=2
    System.out.print(a.get(i)+",");
}
```

ArrayList:

实现了线性表数据结构

内部维护了一个Object类型数组

容量可以动态变化，也被称为动态数组

LinkedList:

实现了“链表”数据结构

内部维护了一个带头结点的双向链表

◆特有方法

addFirst()	向链表头插入元素
addLast ()	向链表尾插入元素
getFirst ()	获取链表头元素
getLast ()	获取链表尾元素
removeFirst ()	移除链表头元素
removeLast ()	移除链表尾元素

◆使用LinkedList可以很简单地构造类似“栈”（stack）、“队列”（queue）这样的数据结构

35

泛型

泛型语法

在**声明泛型类**的变量时，使用尖括号“<>”来指定形式类型参数

```
class  类名<类型参数列表> {类体}
// 例：Class A <X1,X2> {X1 a;  X2 b;}
```

在**应用泛型类**时，必须用具体类型填入类型参数，即泛型类的具体化

```
类名<具体类型列表> 变量名=new 类名<具体类型列表>（构造函数的参数列表）；
// A<String,Integer> m = new A<String,Integer>();
```

泛型的使用

在**类，接口，方法**中均可使用泛型

容器的泛型

示例：定义泛型的类

```
class StackL<T> {   //构造“栈”
    private LinkedList<T> list = new LinkedList();

    public void push(T o) {            //进栈
        list.addFirst(o);
    }
    public T top() {            //查看栈顶元素
        return list.getFirst();
    }
    public T pop() {            //出栈
        return list.removeFirst();
    }
}
```

集合

不允许保存重复元素

常用的实现类**HashSet 和 TreeSet 以及 LinkedHashSet**

```
Set<String> set = new HashSet<String>();
set.add("cat");
set.add("dog");
set.add("cat");
set.add("cat");//无法插入
set.remove("cat");
System.out.println(set);
```

迭代器iterator

是一个对象，可以遍历并选择序列中的对象

- 调用方法**iterator()** 返回一个迭代器。第一次调用Iterator的next()方法时，它返回序列的第一个元素。
- 使用**next()**获得序列中的下一个元素，每成功调用一次迭代器向后移动一个元素。
- 使用**hasNext()**检查序列中是否还有元素。
- 使用**remove()**将迭代器新返回的元素删除。

for each遍历

for(变量类型 变量名:集合){...}

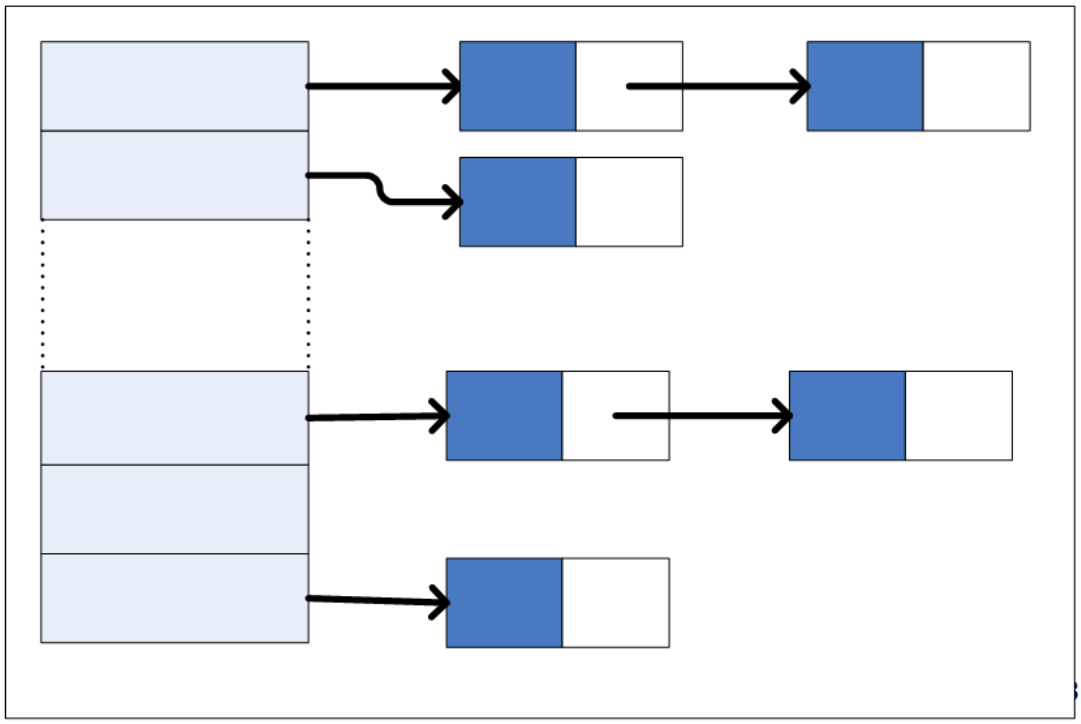
for-each只能遍历两种类型的对象:
- 数组
- 实现了java.lang.Iterable接口的类的实例

```
Set<String> set = new HashSet<String>();
…
for(String s : set) {
    System.out.println(s);
}
```

HashSet

内部维护一个**链表数组**
拥有**常数**查找时间
要求对象重写**hashCode()方法与equals()**方法

◆链表数组



HashSet()	**构造空的HashSet，容量16，载入因子0.75**
HashSet(Collection)	**从其它类集构造HashSet**
HashSet(int)	**指定初始容量，载入因子0.75的HashSet**
HashSet(int,float)	**指定初始容量与载入因子的HashSet**

容量为链表数组的大小，载入因子为数组重构的条件。（扩展哈希集的条件）

- 系统默认hashCode()返回对象在内存中的地址
- 系统默认equals()方法比较两个对象内存中的地址

其他实现：

- TreeSet： 元素间有序的Set
- LinkedHashSet：HashSet+链表

映射 MAP

- 维护**“键（key） - 值(value)”**关系对结构的无序容器
- 键与值都为对象，键值对也为对象
- 一个Map中不能包含相同的key，每个key只能映射一个value。
- 常用的实现类：**HashMap 和 TreeMap以及LinkedHashMap。**

遍历：

```
//方法一（遍历值）
public void byValue(Map<String,Student>  map) {
    Collection<Student> c = map.values();
    Iterator it;
    for (it=c.iterator(); it.hasNext();){
        Student s = (Student)it.next();
    }
}

//方法二（通过键遍历值）
public void byKey (Map<String, Student> map) {
    Set<String> key = map.keySet();
    Iterator it;
    for (it = key.iterator(); it.hasNext();) {
        String s = (String) it.next();
        Student value = map.get(s);
    }
}


//方法三:Map.Entry:Map内部定义的一个接口，专门用来保存键值对的内容
public static void byEntry(Map<String, Student> map) {
    Set<Map.Entry<String, Student>> set = map.entrySet();
    Iterator<Map.Entry<String, Student>> it;
    for (it= set.iterator(); it.hasNext();) {
        Map.Entry<String, Student> entry = it.next();
        System.out.println(entry.getKey() + "--->" + entry.getValue());
    }
}
```