Java 5.0 的新特性

自动装箱和自动拆箱:

自动封箱和自动拆箱，它实现了简单类型和封装类型的相互转化时，实现了自动转化。
自动封箱解箱只在必要的时候才进行。还有其它选择就用其它的
byte b -128~127
Byte b 多一个 null

简单类型和封装类型之间的差别
封装类可以等于 null ，避免数字得 0 时的二义性。
Integer i=null;
int ii=i; 会抛出 NullException 异常。
因为上面的赋值相当于 int ii=i.intValue();
Integer i=1；相当于 Integer i=new Integer(1);

在基本数据类型和封装类之间的自动转换
5.0 之前
Integer i=new Integer（4）;
int ii= i.intValue();

5.0 之后
Integer i=4;
Long l=4.3;

静态引入

静态成员的使用，使用 import static 引入静态成员，也就是可以用静态引入是导入包中的某个类的静态成员，在使用时不用再写类名。
很简单的东西，看一个例子：

没有静态导入，必需使用 类名.方法名 来操作， 如下：
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));

有了静态导入，则可以直接使用 方法名 来操作，如下：
import static java.lang.Math.*;
sqrt(pow(x, 2) + pow(y, 2));

其中 import static java.lang.Math.*;就是静态导入的语法，它的意思是导入 Math 类中的所有 static 方法和属性。这样我们在使用这些方法和属性时就不必写类名。

需要注意的是默认包无法用静态导入，另外如果导入的类中有重复的方法和属性则需要写出类名，否则编译时无法通过。

增强的 for 循环

for-each 循环实现了对数组和集合的便利的统一，解决遍历数组和遍历集合的不统一。
例：

```
import java.util.*;
import java.util.Collection;

public class Foreach
{
	private Collection<String> c = null;
	private String[] belle = new String[4];
	public Foreach()   //构造方法
	{
		belle[0] = "西施";
		belle[1] = "王昭君";
		belle[2] = "貂禅";
		belle[3] = "杨贵妃";
		c = Arrays.asList(belle);
	}
	public void testCollection(){
		for (String b : c){
			System.out.println("曾经的风化绝代:" + b);
		}
	}
	public void testArray(){
		for (String b : belle){
			System.out.println("曾经的青史留名:" + b);
		}
	}
	public static void main(String[] args){
		Foreach each = new Foreach();
		each.testCollection();
		each.testArray();
	}
}
```

对于集合类型和数组类型的，我们都可以通过 foreach 语法来访问它。上面的例子中，以前我们要依次访问数组，挺麻烦：

```
for (int i = 0; i < belle.length; i++){
		String b = belle[i];
		System.out.println("曾经的风化绝代:" + b);
}
```

PDF-XChange Click to buy NOW! www.docu-track.com

现在只需下面简单的语句即可：

```
for (String b : belle){
        System.out.println("曾经的青史留名:" + b);
 }
```

对集合的访问效果更明显。以前我们访问集合的代码：

```
for (Iterator it = c.iterator(); it.hasNext();){
        String name = (String) it.next();
        System.out.println("曾经的风化绝代:" + name);
}
```

现在我们只需下面的语句：

```
for (String b : c){
        System.out.println("曾经的风化绝代:" + b);
}
```

注：foreach 也不是万能的，它也有以下的缺点：
在以前的代码中,我们可以通过 Iterator 执行 remove 操作。

```
for (Iterator it = c.iterator(); it.hasNext();){
        it.remove();
}
```

但是，在现在的 for-each 版中，我们无法删除集合包含的对象。你也不能替换对象。
同时，你也不能并行的 for-each 多个集合。所以，在我们编写代码时，还得看情况而使用它。

可变长的参数

使用条件：只在必要的时候进行。同时有参数为数组，就不能使用变长参数，有变长参数，就不能使用数组，不能共存。一个方法最多只能有一个变长参数，而且是最后一个参数。

5.0 之前

```
public static void main(String[] args){
    // 主方法..
}
```

JVM 收到数据封装在数组里，然后传入方法，在 5.0 之后

```
public static void m(String... s){
    System.out.println("m(String)" +s);
    for(String s2:s) {
        System.out.println(s2);
    }
}
```

调用 m(String… s) 方法如下：

```
m(“jack”,”solo”,”lilei”);   //接收 3 个字符串参数
m(“hello”);   //接收 1 个字符串参数
```

以上调用都是 可行的。

PDF-XChange
Click to buy NOW!
www.docu-track.com

格式化输出

格式化 I/O(Formatted I/O)
java.util.Scanner 类可以进行格式化的输入，可以使用控制台输入，结合了 BufferedReader 和 StringTokener 的功能。

增加了类似 C 的格式化输入输出，简单的例子：

```
public class TestFormat{

   public static void main(String[] args){
      int a = 150000, b = 10;
      float c = 5.0101f, d = 3.14f;
      System.out.printf("%4d %4d%n", a, b);
      System.out.printf("%x %x%n", a, b);
      System.out.printf("%3.2f %1.1f%n", c, d);
      System.out.printf("%1.3e %1.3e%n", c, d*100);
   }
}
```

输出结果为：

```
150000   10
249f0 a
5.01 3.1
5.010e+00 3.140e+02
```

类型安全的枚举

枚举也是一个类型，枚举中的对象只能定义一次并在定义时给其初始化，定义之后不能再改变其值，只能从枚举中选择其一。

```
enum  枚举名{
      枚举值 1(..)，枚举值 2(..)，.....;
}
```

在 5.0 之前使用模式做出枚举

```
final class Season{
      public static final Season SPRING=new Season();
      public static final Season WINTER=new Season();
      public static final Season SUMMER=new Season();
      public static final Season AUTUMN=new Season();
      private Season(){}    //私有构造方法
}
```

PDF-XChange Click to buy NOW! www.docu-track.com

完全等价于

```
enum Season2{
    SPRING(..),//枚举值
    SUMMER(..),
    AUTUMN(..),
    WINTER(..);
.....
}
```

枚举是一个反射关联的典型，反射关联，即在类的定义中有自身类型的属性。
枚举本质上也是一个类，Enum 是枚举的父类。
枚举中的 values()方法会返回枚举中的所有枚举值
枚举中可以定义方法和属性，最后的一个枚举值要以分号和类定义分开，枚举中可以定义的构造方法。
枚举可以实现接口，枚举不能有子类也就是 final 的，枚举的构造方法是 private（私有的），枚举中可以定义抽象方法，可以在枚举值的值中实现抽象方法，枚举值就是枚举的对象，枚举默认是 final，枚举值可以隐含的匿名内部类来实现枚举中定义抽象方法。

枚举类(Enumeration Classes)和类一样，具有类所有特性。Season2 的父类是 java.lang.Enum;
隐含方法：　Season2[] ss=Season2.values();　每个枚举类型都有的方法。enum 可以 switch 中使用（不加类名）。

```
switch( s ){
    case SPRING:
            …….
    case SUMMER:
            …….
    ……..
}
```

枚举的有参构造

```
enum Season2{
    SPRING(“春”),  ----------逗号
    SUMMER(“夏”), ----------逗号
    AUTUMN(“秋”), ----------逗号
    WINTER(“冬”);  ----------分号
    private String name;
    Season2(String name){
        this.name=name;
    }
    String getName(){
        return name;
    }
}
```

Season2.SPRING.getName()　　----------春

枚举的高级用法：

```
enum Operation{
    ADD{
        public double calculate(double s1,double s2){
            return s1+s2;
        }
    },
    SUBSTRACT{
        public double calculate(double s1,double s2){
            return s1-s2;
        }
    },
    MULTIPLY{
        public double calculate(double s1,double s2){
            return s1*s2;
        }
    },
    DIVIDE{
        public double calculate(double s1,double s2){
            return s1/s2;
        }
    };
    public abstract double calculate(double s1 ,double s2);
}
```

有抽象方法枚举元素必须实现该方法。

java5.0 中的泛型

说明

增强了 java 的类型安全，可以在编译期间对容器内的对象进行类型检查，在运行期不必进行类型的转换。而在 java se5.0 之前必须在运行期动态进行容器内对象的检查及转换，泛型是编译时概念，运行时没有泛型

减少含糊的容器，可以定义什么类型的数据放入容器

```
List<Integer> aList = new ArrayList<Integer>();
aList.add(new Integer(1));
// ...
Integer myInteger = aList.get(0);
```

PDF-XChange Click to buy NOW! www.docu-track.com

支持泛型的集合，只能存放制定的类型，或者是指定类型的子类型。

我们可以看到，在这个简单的例子中，我们在定义 aList 的时候指明了它是一个只接受 Integer 类型的 ArrayList，当我们调用 aList.get(0)时，我们已经不再需要先显式的将结果转换成 Integer，然后再赋值给 myInteger 了。而这一步在早先的 Java 版本中是必须的。也许你在想，在使用 Collection 时节约一些类型转换就是 Java 泛型的全部吗？远不止。单就这个例子而言，泛型至少还有一个更大的好处，那就是使用了泛型的容器类变得更加健壮：早先，Collection 接口的 get()和 Iterator 接口的 next()方法都只能返回 Object 类型的结果，我们可以把这个结果强制转换成任何 Object 的子类，而不会有任何编译期的错误，但这显然很可能带来严重的运行期错误，因为在代码中确定从某个 Collection 中取出的是什么类型的对象完全是调用者自己说了算，而调用者也许并不清楚放进 Collection 的对象具体是什么类的；就算知道放进去的对象“应该”是什么类，也不能保证放到 Collection 的对象就一定是那个类的实例。现在有了泛型，只要我们定义的时候指明该 Collection 接受哪种类型的对象，编译器可以帮我们避免类似的问题溜到产品中。我们在实际工作中其实已经看到了太多的 ClassCastException。

用法

声明及实例化泛型类：
HashMap<String,Float> hm = new HashMap<String,Float>();
编译类型的泛型和运行时类型的泛型一定要一致。没有多态。
不能使用原始类型

GenList<int> nList = new GenList<int>(); //编译错误

Java SE 5.0 目前不支持原始类型作为类型参数(type parameter)

定义泛型接口：

```
public interface GenInterface<T> {
    void func(T t);
}
```

定义泛型类：

```
public class ArrayList<ItemType> { ... }
public class GenMap<T, V> { ... }
```

例 1：

```
public class MyList<Element> extends LinkedList<Element>
{
    public void swap(int i, int j){
        Element temp = this.get(i);
        this.set(i, this.get(j));
        this.set(j, temp);
    }
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
    public static void main(String[] args){
        MyList<String> list = new MyList<String>();
        list.add("hi");
        list.add("andy");
        System.out.println(list.get(0) + " " + list.get(1));
        list.swap(0,1);
        System.out.println(list.get(0) + " " + list.get(1));
    }
}
```

泛型的通配符"?"

? 是可以用任意类型替代。
<?>泛型通配符表示任意类型
<? extends 类型>表示这个类型是某个类型的子类型。
<? super 类型>表示这个类型是某个类型的父类型。

```
import java.util.*;
import static java.lang.System.*;   //静态导入
public class TestTemplate {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<String> l1=new ArrayList<String>();
        l1.add("abc");
        l1.add("def");
        List<Number> l2=new ArrayList<Number>();
        l2.add(1.3);
        l2.add(11);
        List<Integer> l3=new ArrayList<Integer>();
        l3.add(123);
        l3.add(456);

    //  print(l1);
        print(l2);
        print(l3);
    }
    static void print(List<? extends Number> l){   //所有 Number 的子类
        for(Object o:l){
            out.println(o);
        }
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
    }

static void print(List<? super Number> l){      //所有 Number 的父类
        for(Object o:l){
            out.println(o);
        }
    }

}
```

"?"可以用来代替任何类型，例如使用通配符来实现 print 方法。

```
public static void print(GenList<?> list) {})
```

泛型方法的定义

把数组拷贝到集合时，数组的类型一定要和集合的泛型相同。
<...>定义泛型，其中的"..."一般用大写字母来代替，也就是泛型的命名，其实，在运行时会根据实际类型替换掉那个泛型。

```
<E> void copyArrayToList(E[] os,List<E> lst){
        for(E o:os){
            lst.add(o);
        }
    }

static <E extends Number> void copyArrayToList(E[] os,List<E> lst){
        for(E o:os){
            lst.add(o);
        }
    }

static<E extends Number & Comparable> void copyArrayToList(E[] os,List<E> lst){
        for(E o:os){
            lst.add(o);
        }
    }
```

受限泛型是指类型参数的取值范围是受到限制的. extends 关键字不仅仅可以用来声明类的继承关系，也可以用来声明类型参数(type parameter)的受限关系.例如，我们只需要一个存放数字的列表，包括整数(Long, Integer, Short)，实数(Double, Float)，不能用来存放其他类型，例如字符串(String)，也就是说，要把类型参数 T 的取值泛型限制在 Number 极其子类中.在这种情况下，我们就可以使用 extends 关键字把类型参数(type parameter)限制为数字

PDF-XChange
Click to buy NOW!
www.docu-track.com

只能使用 extends 不能使用 super，只能向下，不能向上。
调用时用<?>定义时用 <E>

泛型类的定义

类的静态方法不能使用泛型，因为泛型类是在创建对象的时候产生的。

```
class MyClass<E>{
	public void show(E a){
		System.out.println(a);
	}
	public E get(){
		return null;
	}

}
```

受限泛型

```
class MyClass <E extends Number>{
	public void show(E a){

	}
}
```

泛型与异常

类型参数在 catch 块中不允许出现，但是能用在方法的 throws 之后。例：

```
import java.io.*;

interface Executor<E extends Exception> {
	void execute() throws E;
}

public class GenericExceptionTest {
	public static void main(String args[]) {
		try {
			Executor<IOException> e = new Executor<IOException>() {
				public void execute() throws IOException{
					// code here that may throw an
					// IOException or a subtype of
					// IOException
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
                    }
                };
            e.execute();
        } catch(IOException ioe) {
            System.out.println("IOException: " + ioe);
            ioe.printStackTrace();
        }
    }
}
```

泛型的一些局限型

catch 不能使用泛型，在泛型集合中，不能用泛型创建对象，不允许使用泛型的对象。

不能实例化泛型

```
T t = new T(); //error
```

不能实例化泛型类型的数组

```
T[] ts= new T[10];     //编译错误
```

不能实例化泛型参数数

```
Pair<String>[] table = new Pair<String>(10); // ERROR
```

类的静态变量不能声明为类型参数类型

```
public class GenClass<T> {
   private static T t;     //编译错误
}
```

静态方法可以是泛型方法，但是不可以使用类的泛型。

泛型类不能继承自 Throwable 以及其子类

```
public GenExpection<T>    extends Exception{}     //编译错误
```

不能用于基础类型 int 等

```
Pair<double> //error
Pair<Double> //right
```

作者：叶加飞　(steven ye)

mailto: leton.ye@gmail.com