

第9章 泛型和枚举



#知识点一：Java泛型 Generics

#1、引入泛型

之前咱们的超级数组中只能存数字，不能存其他类型的数据，是不是还是显得有些鸡肋。那我们能不能改进一下，让它可以存任意类型。

第一种解决方案：

将内部的int数据，换成Object类型，使用引用数据类型替代基础数据类型。因为Object是所有类的超类，所以任何子类都可以传递进去，我们的代码可以简化如下：

```
1  package com.ydlclass;
2  public class SuperArray {
3      private Object[] array;
4      //根据下标查询数字
5
6      //当前最后一个数字的下边，要为-1，以为数组的第一个下标为0
7      private int currentIndex = -1;
8
9      //构造是初始化
10     public SuperArray(){
11         array = new Object[8];
12     }
13
14     //添加数据的方法
15     public void add(Object data){
16         System.out.println("我是数组的实现! ---add");
17         currentIndex++;
18         //自动扩容
19         if(currentIndex > array.length-1){
20             array = dilatation(array);
21         }
22         array[currentIndex] = data;
23     }
24
25     public Object get(int index){
26         System.out.println("我是数组的实现---get");
27         return array[index];
28     }
29
30     //数组扩容的方法
31     private Object[] dilatation(Object[] oldArray){
32         Object[] newArray = new Object[oldArray.length * 2];
33         for (int i = 0; i < oldArray.length; i++) {
```

```

34         newArray[i] = oldArray[i];
35     }
36     return newArray;
37 }
38
39 //验证下标是否合法
40 private boolean validateIndex(int index) {
41     //只要有一个不满足就返回false
42     return index <= currentIndex && index >= 0;
43 }
44 }

```

```

1 public static void main(String[] args) {
2     SuperArray superArray = new SuperArray();
3     superArray.add("abc");
4     String item = (String)superArray.get(0);
5 }

```

思考这样会有什么问题吗？

- 1、我们规定传入的对象只要是Object子类就行，那就意味着，所有的对象都可以往篮子里扔，可以扔水果，也可以扔炸弹，数据类型不能很好的统一。
- 2、从超级数组中获取数据后必须强转才能使用，这是不是意味着，极有可能发生ClassCastException。

```

1 SuperArray superArray = new SuperArray();
2 superArray.add(new Date());
3 superArray.add(new Dog());
4
5 (Dog)superArray.get(0);

```

那怎么解决这个问题呢？

我们的目标是：1、能够规定传入的数据类型必须是我们要求的。

- 2、从数组中获取的数据必须是确定的类型。

【泛型】就能够很好的解决这个问题。

#2、泛型的定义

什么是泛型？

看表面的意思，泛型就是指广泛的、普通的类型。泛型能够帮助我们把【类型明确】的工作推迟到创建对象或调用方法的时候。

意思就是：我定义类的时候不用管到底是什么类型，new这个对象或者调用这个方法时才确定具体的类型。

这听起来很不可思议，那到底是什么意思呢？咱们用以下的例子来说明情况：

(1) 泛型类

泛型类也就是把泛型定义在类上，这样用户在使用类的时候才把类型给确定下来。

具体的方法就是使用<>加一个未知数，通常用T K V等大写字母表示，事实上只要是个单词就可以。

```

1 package com.ydlclass;
2 // 加上<T>之后，表示以后的SuperArray只能存某种类型，但是这个类型暂时不确定，使用T来代替
3 public class SuperArray<T> {

```

```

4     private Object[] array;
5     //根据下标查询数字
6
7     //当前最后一个数字的下边，要为-1，以为数组的第一个下标为0
8     private int currentIndex = -1;
9
10    //构造是初始化
11    public SuperArray(){
12        array = new Object[8];
13    }
14
15    //添加数据的方法
16    public void add(T data){
17        System.out.println("我是数组的实现! ---add");
18        currentIndex++;
19        //自动扩容
20        if(currentIndex > array.length-1){
21            array = dilatation(array);
22        }
23        array[currentIndex] = data;
24    }
25
26    public T get(int index){
27        System.out.println("我是数组的实现---get");
28        return (T)array[index];
29    }
30
31    //数组扩容的方法
32    private Object[] dilatation(Object[] oldArray){
33        Object[] newArray = new Object[oldArray.length * 2];
34        for (int i = 0; i < oldArray.length; i++) {
35            newArray[i] = oldArray[i];
36        }
37        return newArray;
38    }
39
40    //验证下标是否合法
41    private boolean validateIndex(int index) {
42        //只要有一个不满足就返回false
43        return index <= currentIndex && index >= 0;
44    }
45 }

```

有了上边的代码，我们需要怎么去定义一个超级数组呢？

```

1     public static void main(String[] args) {
2         // jdk1.7以前
3         SuperArray<String> superArray = new SuperArray<String>();
4         // jdk1.7以后提出了钻石语法，可以进行类型的自动推断，后边的尖括号就不用写了
5         SuperArray<String> superArray = new SuperArray<>();
6         superArray.add("abc");
7         String item = superArray.get(0);
8     }

```

我们申明一个类的时候，无需关心将来我的超级数组存的是什么类型，但是new SuperArray的时候明确指出了这个超级数组只能存String类型的，其他类型就不能存。

可以看到上面这个程序，在使用时如果定义了类型，那么在使用时就可以不用进行强制类型转换，直接就可以得到一个T类型的对象。

(2) 泛型方法

有时候只关心某个方法，那么使用泛型时可以不定义泛型类，而是只定义一个泛型方法，如下：

```
1 public class Test2 {
2     public <T> T show(T t) {
3         System.out.println(t);
4         return t;
5     }
6
7     public static <T> T show2(T one) { //这是正确的
8         return null;
9     }
10
11    public static void main(String[] args) {
12        Test2 test2 = new Test2();
13        String show = test2.show("123");
14        Integer show1 = test2.show(123);
15    }
16 }
```

需要注意一下定义的格式，泛型必须得先定义才能够使用。

说明一下，定义泛型方法时，必须在返回值前边加一个，来声明这是一个泛型方法，持有一个泛型T，然后才可以用泛型T作为方法的返回值。

泛型方法最好要结合具体的返回值，否则和Object作为参数差别不大。

我们在学习反射的时候会学习类似的例子。

(3) 继承关系

泛型类在继承时，可以明确父类（泛型类）的参数类型，也可以不明确。还记得我们学习策略设计模式时的Comparator接口吗？之前的设计必须是User类型，而现在有了泛型我们可以灵活使用了：

```
1 // 泛型类
2 public interface Comparator<T>{
3     int compare(T o1, T o2);
4 }
```

```
1 public class StudentComparator implements Comparator {
2     @Override
3     public Integer compare(Object o1, Object o2) {
4         if(o1 instanceof Student && o2 instanceof Student){
5             return ((Student) o1).getAge() - ((Student) o2).getAge();
6         }
7         return null;
8     }
9 }
```

(1) 明确类型，子类存在的目的就是比较User对象

```
1 package com.ydlclass;
2
3 public class User {
4     private String name;
5     private int age;
6     private int height;
7
8     public User() {
9     }
10
11     public User(String name, int age, int height) {
12         this.name = name;
13         this.age = age;
14         this.height = height;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setName(String name) {
22         this.name = name;
23     }
24
25     public int getAge() {
26         return age;
27     }
28
29     public void setAge(int age) {
30         this.age = age;
31     }
32
33     public int getHeight() {
34         return height;
35     }
36
37     public void setHeight(int height) {
38         this.height = height;
39     }
40 }
```

```
1 //在实现泛型类时明确父类的类型
2 import java.util.Comparator;
3
4 public class UserAgeComparator implements Comparator<User> {
5     @Override
6     public int compare(User o1, User o2) {
7         return o1.getAge() - o2.getAge();
8     }
9 }
10
11 public static void main(String[] args) {
12     UserAgeComparator userAgeComparator = new UserAgeComparator();
13     int compare = userAgeComparator.compare(new User(), new User());
14     System.out.println(compare);
15 }
```

```
15 }
```

(2) 不明确类型

子类不去明确类型，明确类型的工作留在创建对象的时候：

```
1 public class UserAgeComparator<T> implements Comparator<T> {
2
3     @Override
4     public int compare(T o1, T o2) {
5         return o1.equals(o2) ? 0 : 1;
6     }
7
8     public static void main(String[] args) {
9         UserAgeComparator<User> userAgeComparator = new UserAgeComparator<>();
10        int compare = userAgeComparator.compare(new User(), new User());
11        System.out.println(compare);
12    }
13 }
```

#3、项目实战

用泛型改进超级数组和超级链表，这里只有部分代码：

```
1 package com.ydlclass;
2
3 public interface Super<T> {
4
5     /**
6      * 标记所有的子类实现必须有add方法，添加数据
7      * @param data
8      */
9     void add(T data);
10
11     /**
12      * 标记所有的子类实现必须有get方法，获取数据
13      * @param index
14      * @return
15      */
16     T get(int index);
17
18     /**
19      * 标记所有的子类实现必须有size方法，数据大小
20      * @return
21      */
22     int size();
23 }
```

```
1 package com.ydlclass;
2
3 public class SuperArray<T> implements Super<T> {
4
5     //维护一个数组,要想什么都存,就要使用顶级父类
6     private Object[] array;
7     //当前最后一个数字的下边,要为-1,以为数组的第一个下标为0
8     private int currentIndex = -1;
9 }
```

```

10      //构造是初始化
11      public SuperArray(){
12          array = new Object[8];
13      }
14
15      //添加数据的方法
16      public void add(T data){
17          System.out.println("我是数组的实现! ---add");
18          currentIndex++;
19          //自动扩容
20          if(currentIndex > array.length-1){
21              array = dilatation(array);
22          }
23          array[currentIndex] = data;
24      }
25
26
27      //根据下标查询数字
28      public T get(int index){
29          System.out.println("我是数组的实现---get");
30          return (T)array[index];
31      }
32
33      //查看当前有多少个数字
34      public int size(){
35          return currentIndex + 1;
36      }
37
38      //数组扩容的方法
39      private Object[] dilatation(Object[] oldArray){
40          Object[] newArray = new Object[oldArray.length * 2];
41          for (int i = 0; i < oldArray.length; i++) {
42              newArray[i] = oldArray[i];
43          }
44          return newArray;
45      }
46
47      //验证下标是否合法
48      private boolean validateIndex(int index) {
49          //只要有一个不满足就返回false
50          return index <= currentIndex && index >= 0;
51      }
52  }

```

```

1  package com.ydlclass;
2
3  /**
4   * @author itnanls
5   * @date 2021/7/16
6   */
7  public class SuperLinked<T> implements Super<T> {
8
9      private Node head = null;
10     private Node tail = null;
11
12     private int length = 0;
13

```

```

14      //添加元素
15      public void add(T data){
16          System.out.println("我是链表的实现-----add");
17          Node<T> node = new Node<>();
18          node.setNum(data);
19          if (length == 0) {
20              //如果第一次添加一共就一个节点
21              head = node;
22          }else{
23              //和尾巴拉手
24              tail.setNextNode(node);
25          }
26          //把新添加进来的当成尾巴
27          tail = node;
28          length ++;
29      }
30
31      //根据下标查询数字,非常有意思的写法
32      public T get(int index){
33          System.out.println("我是链表的实现-----get");
34          if(index > length){
35              return null;
36          }
37          //小技巧
38          Node targetNode = head;
39          for (int i = 0; i < index; i++) {
40              targetNode = targetNode.getNextNode();
41          }
42          return (T)(targetNode.getNum());
43      }
44
45      //查看当前有多少个数字
46      public int size(){
47          return length;
48      }
49
50      class Node<T> {
51
52          //存储的真实数据
53          private T num;
54
55          //写一个节点
56          private Node nextNode = null;
57
58          public T getNum() {
59              return num;
60          }
61
62          public void setNum(T num) {
63              this.num = num;
64          }
65
66          public Node getNextNode() {
67              return nextNode;
68          }
69
70          public void setNextNode(Node nextNode) {
71              this.nextNode = nextNode;

```



```
72     }
73 }
74 }
```

#4、类型通配符

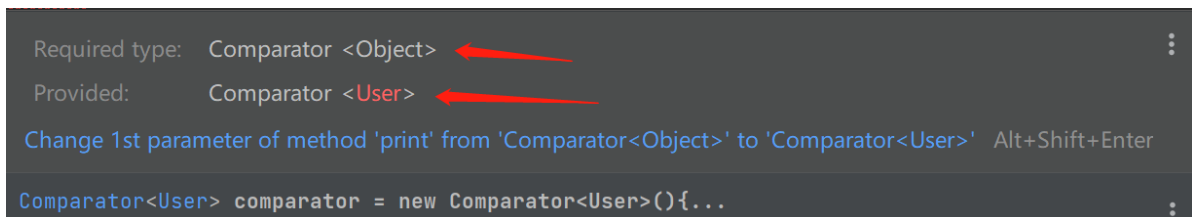
新建三个类：

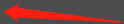
```
1 public class Animal {
2 }
3 public class Dog extends Animal {
4 }
5 public class Teddy extends Dog {
6 }
```


当我们的一个方法的参数需要传入一个带有参数的类型的时候，可以使用通配符来确定具体传入的对象范围。

```
1 public static void print(Comparator<Object> comparator){
2
3 }
4
5 public static void main(String[] args) {
6     Comparator<User> comparator = new Comparator<User>(){
7         @Override
8         public int compare(User o1, User o2) {
9             return o1.getAge() - o2.getAge();
10        }
11    };
12    print(comparator);
13 }
```

会有以下的报错信息：



Required type: Comparator <Object> 

Provided: Comparator <User> 

Change 1st parameter of method 'print' from 'Comparator<Object>' to 'Comparator<User>' Alt+Shift+Enter

```
Comparator<User> comparator = new Comparator<User>(){...
```

意思就是我需要一个泛型是Object的Comparator但你提供的泛型是User，使用通配符就能很好的解决这些问题：

(1) 无界

类型通配符我感觉上和泛型方法差不多，只是不用在使用前进行定义，例子如下：

```

1 public static void main(String[] args) {
2     SuperArray<Dog> superArray = new SuperArray<>();
3     superArray.add(new Dog());
4     superArray.add(new Teddy());
5     printSuperArray(superArray);
6 }
7
8 public static void printSuperArray(SuperArray<?> superArray){
9     for (int i = 0;i<superArray.size();i++){
10         System.out.println(superArray.get(i));
11     }
12 }

```

"?"可以接收任何类型，有些聪明的小伙伴可能发现不加?，连泛型也不要行不行，悄悄的告诉你，可以，但是程序会报一个警告：

```

1 public static void print(Comparator comparator){ }

```

```

1 Raw use of parameterized class 'xxxx' 警告
2 没有类型参数的泛型

```

使用原始类型（没有类型参数的泛型）是合法的，但是你永远不应该这样做。如果使用原始类型，就会失去泛型的安全性和表现力。既然你不应该使用它们，那么为什么语言设计者一开始就允许原始类型呢？答案是：为了兼容性。Java 即将进入第二个十年，泛型被添加进来时，还存在大量不使用泛型的代码。保持所有这些代码合法并与使用泛型的新代码兼容被认为是关键的。将参数化类型的实例传递给设计用于原始类型的方法必须是合法的，反之亦然。

(2) 上界

我们可以使用 (`SuperArray<? extends Dog> superArray`) 的形式来约定传入参数的上界，意思就是泛型只能是Dog的或者Dog的子类。

```

1 public static void main(String[] args) {
2     SuperArray<Animal> superArray = new SuperArray<>();
3     superArray.add(new Dog());
4     superArray.add(new Teddy());
5     superArray.add(new Animal());
6     printSuperArray(superArray);
7 }
8
9 public static void printSuperArray(SuperArray<? extends Dog> superArray){
10     for (int i = 0;i<superArray.size();i++){
11         System.out.println(superArray.get(i));
12     }
13 }

```

这种情况下能够接收A类或者A类的子类。

```

9      SuperArray<Animal> superArray = new SuperArray<>();
10     superArray.add(new Dog());
11     superArray.add(new Teddy());
12     superArray.add(new Animal());
13     printSuperArray(superArray);
14 }
15
16
17
18 @   public static void printSuperArray(SuperArray<? extends Dog> superArray){
19     for (int i = 0;i<superArray.size();i++){

```

Required type: SuperArray <? extends Dog>
 Provided: SuperArray <Animal>
 Cast parameter to 'com.ydclass.SuperArray<? extends com.ydclass.Dog>' Alt+Shift+Enter More actions
 SuperArray<Animal> superArray = new SuperArray<Animal>()

注:当我们使用extends时, 我们可以读元素, 因为元素都是A类或子类, 可以放心的用A类拿出。

(3) 下界

我们可以使用 (`SuperArray<? super Dog> superArray`) 的形式来约定传入参数的下界, 意思就是泛型只能是Dog的或者Dog的超类。

```

1  public static void main(String[] args) {
2      SuperArray<Teddy> superArray = new SuperArray<>();
3      superArray.add(new Teddy());
4      printSuperArray(superArray);
5  }
6  public static void printSuperArray(SuperArray<? super Dog> superArray){
7      for (int i = 0;i<superArray.size();i++){
8          System.out.println(superArray.get(i));
9      }
10 }

```

```

7  public class Test {
8      public static void main(String[] args) {
9          SuperArray<Teddy> superArray = new SuperArray<>();
10         superArray.add(new Teddy());
11         printSuperArray(superArray);
12     }
13
14
15
16
17 @   public static void printSuperArray(SuperArray<? super Dog> superArray){
18     for (int i = 0;i<superArray.size();i++){
19         System.out.println(superArray.get(i));
20     }
21 }

```

Required type: SuperArray <? super Dog>
 Provided: SuperArray <Teddy>
 Change 1st parameter of method 'printSuperArray' from 'SuperArray<? super Dog>' to 'SuperArray<Teddy>'
 SuperArray<Teddy> superArray = new SuperArray<Teddy>()

当使用super时, 可以添加元素, 因为都是A类或父类, 那么就可以安全的插入A类。

#5、类型擦除

我们刚刚讲过, 为了兼容性, 使用原始类型 (没有类型参数的泛型) 是合法的, 泛型被添加进来时, 还存在大量不使用泛型的代码。保持所有这些代码合法并与使用泛型的新代码兼容被认为是关键的。将参数化类型的实例传递给设计用于原始类型的方法必须是合法的, 反之亦然。

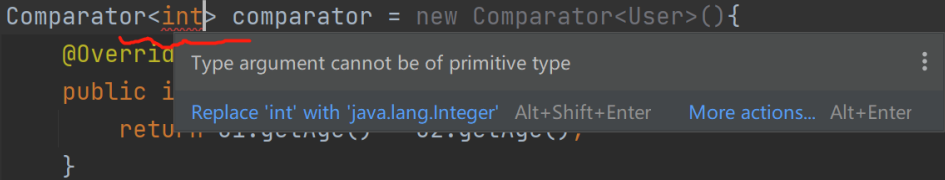
为了保持这种兼容性, Java的泛型其实是一种伪泛型, 这是因为Java在编译期间, 所有的泛型信息都会被擦掉, 正确理解泛型概念的首要前提是理解类型擦除。Java的泛型基本上都是在编译器这个层次上实现的, 在生成的字节码中是不包含泛型中的类型信息的, 使用泛型的时候加上类型参数, 在编译器编译的时候会去掉, 这个过程成为**类型擦除**。

如在代码中定义 `SuperArray<Object>` 和 `SuperArray<String>` 等类型，在编译后都会变成 `SuperArray`，JVM看到的只是 `SuperArray`，而由泛型附加的类型信息对JVM是看不到的。Java编译器会在编译时尽可能的发现可能出错的地方，但是仍然无法在运行时刻出现完全避免类型转换异常的情况。

(1) 泛型不能是基本数据类型

不能用类型参数替换基本类型。就比如，没有 `SuperArray<double>`，只有 `SuperArray<Double>`。因为当类型擦除后，`SuperArray` 的原始类型变为 `Object`，但是 `Object` 类型不能存储 `double` 值，只能引用 `Double` 的值。

```
public static void main(String[] args) {
    Comparator<int> comparator = new Comparator<User>(){
        @Override
        public int compare(Object o1, Object o2){
            return o1.getName().compareTo(o2.getName());
        }
    }
}
```



这一点尤其重要：必须要记住。

(2) 重载方法

如果泛型类型因为具体的泛型不同而导致方法签名不同，那么以下两个方法就是两种重载方法：


```
1 public static void print(Comparator<Object> comparator){
2
3 }
4
5 public static void print(Comparator<User> comparator){
6
7 }
```

然而事实上：

```
public static void print(Comparator<Object> comparator){
}

public static void print(Comparator<User> comparator){
}

}
```



因为泛型被擦除后，其实这两个方法是一致的，并不能构成泛型。

(3) 类型擦除和多态的冲突

现在有这样一个泛型类：

```
1 class Pair<T> {
2     private T value;
3     public T getValue() {
4         return value;
5     }
6     public void setValue(T value) {
7         this.value = value;
8     }
9 }
```

然后我们想要一个子类继承它。

```
1 package com.ydlclass;
2
3 import java.util.Date;
4
5 public class DatePair extends Pair<Date>{
6     @Override
7     public Date getValue() {
8         return super.getValue();
9     }
10
11     @Override
12     public void setValue(Date value) {
13         super.setValue(value);
14     }
15 }
```

在这个子类中，我们设定父类的泛型类型为 `Pair<Date>`，在子类中，我们重写了父类的两个方法，我们的原意是这样的：将父类的泛型类型限定为 `Date`，那么父类里面的两个方法的参数都为 `Date` 类型。

所以，我们在子类中重写这两个方法一点问题也没有，实际上，从他们的 `@Override` 标签中也可以看到，一点问题也没有，实际上是这样的吗？

分析：实际上，类型擦除后，父类的泛型类型全部变为了原始类型 `Object`，所以父类编译之后会变成下面的样子：

```
1 class Pair {
2     private Object value;
3     public Object getValue() {
4         return value;
5     }
6     public void setValue(Object value) {
7         this.value = value;
8     }
9 }
```

```
1 {
2     public com.ydlclass.Pair();
3     descriptor: ()V
4     flags: ACC_PUBLIC
5     Code:
6         stack=1, locals=1, args_size=1
7         0: aload_0
8         1: invokespecial #1           // Method java/lang/Object."
<init>":()V
9         4: return
10    LineNumberTable:
11        line 3: 0
12    LocalVariableTable:
13        Start Length Slot Name Signature
14        0      5     0 this Lcom/ydlclass/Pair;
15    LocalVariableTypeTable:
16        Start Length Slot Name Signature
17        0      5     0 this Lcom/ydlclass/Pair<TT>;>;
18
19    public T getValue();
```

```

20 // 这里我们知道这个方法的返回值是Object
21 descriptor: ()Ljava/lang/Object;
22 flags: ACC_PUBLIC
23 Code:
24     stack=1, locals=1, args_size=1
25         0: aload_0
26         1: getfield      #2                // Field value:Ljava/lang/Object;
27         4: areturn
28     LineNumberTable:
29         line 6: 0
30     LocalVariableTable:
31         Start Length Slot Name Signature
32         0      5      0 this Lcom/ydlclass/Pair;
33     LocalVariableTypeTable:
34         Start Length Slot Name Signature
35         0      5      0 this Lcom/ydlclass/Pair<TT>;
36     Signature: #20                // ()TT;
37
38 public void setValue(T);
39 // 这里我们知道这个方法的参数是引用数据类型, Object
40 descriptor: (Ljava/lang/Object;)V
41 flags: ACC_PUBLIC
42 Code:
43     stack=2, locals=2, args_size=2
44         0: aload_0
45         1: aload_1
46         2: putfield      #2                // Field value:Ljava/lang/Object;
47         5: return
48     LineNumberTable:
49         line 9: 0
50         line 10: 5
51     LocalVariableTable:
52         Start Length Slot Name Signature
53         0      6      0 this Lcom/ydlclass/Pair;
54         0      6      1 value Ljava/lang/Object;
55     LocalVariableTypeTable:
56         Start Length Slot Name Signature
57         0      6      0 this Lcom/ydlclass/Pair<TT>;
58         0      6      1 value TT;
59     Signature: #23                // (TT;)V
60 }
61 Signature: #24                //
62 <T:Ljava/lang/Object;>Ljava/lang/Object;
63 SourceFile: "Pair.java"

```

再看子类的两个重写的方法的类型:

```

1  @Override
2  public void setValue(Date value) {
3      super.setValue(value);
4  }
5  @Override
6  public Date getValue() {
7      return super.getValue();
8  }

```

```

1  {
2      public java.util.Date getValue();

```

```

3      descriptor: ()Ljava/util/Date;
4      flags: ACC_PUBLIC
5      Code:
6          stack=1, locals=1, args_size=1
7              0: aload_0
8              1: invokespecial #2                  // Method
com/ydlclass/Pair.getValue:()Ljava/lang/Object;
9              4: checkcast      #3                  // class java/util/Date
10             7: areturn
11     LineNumberTable:
12         line 8: 0
13     LocalVariableTable:
14         Start  Length  Slot  Name   Signature
15             0       8      0  this   Lcom/ydlclass/DatePair;
16
17     public void setValue(java.util.Date);
18     descriptor: (Ljava/util/Date;)V
19     flags: ACC_PUBLIC
20     Code:
21         stack=2, locals=2, args_size=2
22             0: aload_0
23             1: aload_1
24             2: invokespecial #4                  // Method
com/ydlclass/Pair.setValue:(Ljava/lang/Object;)V
25             5: return
26     LineNumberTable:
27         line 13: 0
28         line 14: 5
29     LocalVariableTable:
30         Start  Length  Slot  Name   Signature
31             0       6      0  this   Lcom/ydlclass/DatePair;
32             0       6      1 value   Ljava/util/Date;
33
34
35     // 桥接方法，一会分析
36     public void setValue(java.lang.Object);
37     descriptor: (Ljava/lang/Object;)V
38     flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
39     Code:
40         stack=2, locals=2, args_size=2
41             0: aload_0
42             1: aload_1
43             2: checkcast      #3                  // class java/util/Date
44             5: invokevirtual #5                  // Method setValue:
(Ljava/util/Date;)V
45             8: return
46     LineNumberTable:
47         line 5: 0
48     LocalVariableTable:
49         Start  Length  Slot  Name   Signature
50             0       9      0  this   Lcom/ydlclass/DatePair;
51
52     public java.lang.Object getValue();
53     descriptor: ()Ljava/lang/Object;
54     flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
55     Code:
56         stack=1, locals=1, args_size=1
57             0: aload_0

```

```

58      1: invokevirtual #6                      // Method getValue:
      ()Ljava/util/Date;
59      4: areturn
60      LineNumberTable:
61        line 5: 0
62      LocalVariableTable:
63        Start  Length  Slot  Name   Signature
64         0       5      0  this   Lcom/ydlclass/DatePair;
65    }
66    Signature: #29                      // Lcom/ydlclass/Pair<Ljava/util/Date;>;
67    SourceFile: "DatePair.java"

```

先来分析 `setValue` 方法，父类的类型是 `Object`，而子类的类型是 `Date`，参数类型不一样，这如果是在普通的继承关系中，根本就不会是重写，而是重载。

我们在一个main方法测试一下：

```

1  public static void main(String[] args) throws ClassNotFoundException {
2      DatePair DatePair = new DatePair();
3      DatePair.setValue(new Date());
4      DatePair.setValue(new Object()); //编译错误
5  }

```

如果是重载，那么子类中两个 `setValue` 方法，一个是参数 `Object` 类型，一个是 `Date` 类型，可是我们发现，根本就没有这样的一个子类继承自父类的Object类型参数的方法。所以说，确实是重写了，而不是重载了。

关键字：ACC_BRIDGE, ACC_SYNTHETIC

从编译的结果来看，我们本意重写 `setValue` 和 `getValue` 方法的子类，竟然有4个方法，其实不用惊奇，最后的两个方法，就是编译器自己生成的【桥方法】，我们从字节码中看到两个标志【ACC_BRIDGE, ACC_SYNTHETIC】。可以看到桥方法的参数类型都是Object，也就是说，子类中真正覆盖父类两个方法的就是这两个我们看不到的桥方法。而在我们自己定义的 `setValue` 和 `getValue` 方法上面的 `@Override` 只不过是假象。而桥方法的内部实现，就只是去调用我们自己重写的那两个方法。

所以，虚拟机巧妙的使用了桥方法，来解决类型擦除和多态的冲突。

并且，还有一点也许会有疑问，子类中的桥方法 `Object getValue()` 和 `Date getValue()` 是同时存在的，可是如果是常规的两个方法，他们的方法签名是一样的，也就是说虚拟机根本不能分辨这两个方法。如果是我们自己编写Java代码，这样的代码是无法通过编译器的检查的，但是虚拟机却是允许这样做的，编译器为了实现泛型的多态允许自己做这个看起来“不合法”的事情，然后交给虚拟机去区别。

#6、静态方法和静态类中的问题

泛型类中的静态方法和静态变量不可以使用泛型类所声明的泛型类型参数

举例说明：

```

1  public class Test2<T> {
2      public static T one;    //编译错误
3      public static T show(T one){ //编译错误
4          return null;
5      }
6  }

```

因为泛型类中的泛型参数的实例化是在定义对象的时候指定的，而静态变量和静态方法不需要使用对象来调用。对象都没有创建，如何确定这个泛型参数是何种类型，所以当然是错误的。

但是要注意区分下面的一种情况：

```
1 public class Test2<T> {
2
3     public static <T> T show(T one){ //这是正确的
4         return null;
5     }
6 }
```

因为这是一个泛型方法，在泛型方法中使用的T是自己在方法中定义的T，而不是泛型类中的T。

#知识点二：枚举 enum

在某些情况下，一个类的对象的实例有限且固定的，如季节类，它只有春夏秋冬4个对象，再比如星期，在这种场景下我们可以使用枚举。当然我们也可以有自己的方法来实现。

方案一：静态常量

```
1 public class SeasonConstant {
2     public static final int SPRING = 1;
3     public static final int SUMMER = 2;
4     public static final int AUTUMN = 3;
5     public static final int WINTER = 4;
6 }
```

这种方式，我们可以简单的表示春夏秋冬四个季节，但是扩展性很差，我们想给春夏秋冬附加更多信息的时候就无能为力的，静态常量能保证内存独此一份，更够很好的表示春夏秋冬四个季节，同时不允许别人修改。

方案二：利用类似单例模式的方案

既然使用基础数据类型无法表示丰富的内容，我们不妨把基础类型改为引用数据类型。

```
1 public class Season {
2     private int value;
3     private String name;
4
5     // 定义四个静态常量让每个季节在内存中独此一份
6     public static final Season SPRING = new Season(1, "春天");
7     public static final Season SUMMER = new Season(2, "夏天");
8     public static final Season AUTUMN = new Season(3, "秋天");
9     public static final Season WINTER = new Season(4, "冬天");
10
11     private Season(){}
12
13     private Season(int value, String name) {
14         this.value = value;
15         this.name = name;
16     }
17
18
19     public int getValue() {
20         return value;
21     }
22
23     public void setValue(int value) {
24         this.value = value;
```

```

25     }
26
27     public String getName() {
28         return name;
29     }
30
31     public void setName(String name) {
32         this.name = name;
33     }
34 }

```

我们这样做不仅仅能够保证内存中只有四个对象，我们不妨验证一下：

```

1  public static void main(String[] args) {
2      System.out.println(Season.SPRING == Season.SPRING);
3  }

```

使用==比较两个对象比较的是内存地址，内存地址一样不正好说明是同一个对象嘛。

只是这种写法略显复杂，我们可以使用简单的方式表达：

首先我们先简化一下代码，去掉其他属性：

```

1  public class Season {
2      // 定义四个静态常量让每个季节在内存中独此一份
3      public static final Season SPRING = new Season();
4      public static final Season SUMMER = new Season();
5      public static final Season AUTUMN = new Season();
6      public static final Season WINTER = new Season();
7  }

```

我们发现这个重复的太多了，直接干掉

```

1  public class Season {
2      // 定义四个静态常量让每个季节在内存中独此一份
3      SPRING, SUMMER, AUTUMN, WINTER;
4  }

```

如果能写成这个样子，是不是就好了，这仅仅是将所有的重复代码删掉了而已。

Java1.5 引入了 enum 来定义枚举类，就可以使用这样的书写方式了，但是要将class换成enum，如下：

```

1  public enum SeasonEnum {
2      SPRING, SUMMER, AUTUMN, WINTER;
3  }

```

当然，以上的例子都是为了更好的解释枚举类，事实上枚举还有

#1、基本Enum特性

- 枚举类的定义

```

1  public enum SeasonEnum {
2      SPRING, SUMMER, AUTUMN, WINTER;
3  }

```

不妨看看字节码文件：

这个是静态常量的：

```

1 C:\Users\zn\IdeaProjects\untitled\out\production\untitled>javap -v Season.class
2 Classfile
   /C:/Users/zn/IdeaProjects/untitled/out/production/com/ydlclass/Season.class
3   Last modified 2021-8-28; size 1103 bytes
4   MD5 checksum e7dac070287209c9e80194b38fa4b27b
5   Compiled from "Season.java"
6 public class Season
7   minor version: 0
8   major version: 52
9   flags: ACC_PUBLIC, ACC_SUPER
10  Constant pool:
11     #1 = Methodref          #14.#42          // java/lang/Object."<init>":()V
12     #2 = Fieldref           #4.#43            // Season.value:I
13     #3 = Fieldref           #4.#44            // Season.name:Ljava/lang/String;
14     #4 = Class               #45              // Season
15     #5 = String              #46              // 春天
16     #6 = Methodref          #4.#47            // Season."<init>":
      (ILjava/lang/String;)V
17     #7 = Fieldref           #4.#48            // Season.SPRING:LSeason;
18     #8 = String              #49              // 夏天
19     #9 = Fieldref           #4.#50            // Season.SUMMER:LSeason;
20     #10 = String             #51              // 秋天
21     #11 = Fieldref          #4.#52            // Season.AUTUMN:LSeason;
22     #12 = String             #53              // 冬天
23     #13 = Fieldref          #4.#54            // Season.WINTER:LSeason;
24     #14 = Class              #55              // java/lang/Object
25     #15 = Utf8               value
26     #16 = Utf8               I
27     #17 = Utf8               name
28     #18 = Utf8               Ljava/lang/String;
29     #19 = Utf8               SPRING
30     #20 = Utf8               LSeason;
31     #21 = Utf8               SUMMER
32     #22 = Utf8               AUTUMN
33     #23 = Utf8               WINTER
34     #24 = Utf8               <init>
35     #25 = Utf8               ()V
36     #26 = Utf8               Code
37     #27 = Utf8               LineNumberTable
38     #28 = Utf8               LocalVariableTable
39     #29 = Utf8               this
40     #30 = Utf8               (ILjava/lang/String;)V
41     #31 = Utf8               getValue
42     #32 = Utf8               ()I
43     #33 = Utf8               setValue
44     #34 = Utf8               (I)V
45     #35 = Utf8               getName
46     #36 = Utf8               ()Ljava/lang/String;
47     #37 = Utf8               setName
48     #38 = Utf8               (Ljava/lang/String;)V
49     #39 = Utf8               <clinit>
50     #40 = Utf8               SourceFile
51     #41 = Utf8               Season.java
52     #42 = NameAndType        #24:#25          // "<init>":()V
53     #43 = NameAndType        #15:#16          // value:I
54     #44 = NameAndType        #17:#18          // name:Ljava/lang/String;
55     #45 = Utf8               Season
56     #46 = Utf8               春天

```

```

57  #47 = NameAndType      #24:#30      // "<init>":(Ljava/lang/String;)V
58  #48 = NameAndType      #19:#20      // SPRING:LSeason;
59  #49 = Utf8              夏天
60  #50 = NameAndType      #21:#20      // SUMMER:LSeason;
61  #51 = Utf8              秋天
62  #52 = NameAndType      #22:#20      // AUTUMN:LSeason;
63  #53 = Utf8              冬天
64  #54 = NameAndType      #23:#20      // WINTER:LSeason;
65  #55 = Utf8              java/lang/Object
66  {
67      // 这一部分我们感觉像是四个静态常量
68      public static final Season SPRING;
69          descriptor: LSeason;
70          flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
71
72      public static final Season SUMMER;
73          descriptor: LSeason;
74          flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
75
76      public static final Season AUTUMN;
77          descriptor: LSeason;
78          flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
79
80      public static final Season WINTER;
81          descriptor: LSeason;
82          flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
83
84      public int getValue();
85          descriptor: ()I
86          flags: ACC_PUBLIC
87      Code:
88          stack=1, locals=1, args_size=1
89              0: aload_0
90              1: getfield      #2              // Field value:I
91              4: ireturn
92      LineNumberTable:
93          line 20: 0
94      LocalVariableTable:
95          Start Length Slot Name Signature
96              0      5     0  this  LSeason;
97
98      public void setValue(int);
99          descriptor: (I)V
100         flags: ACC_PUBLIC
101     Code:
102         stack=2, locals=2, args_size=2
103             0: aload_0
104             1: iload_1
105             2: putfield      #2              // Field value:I
106             5: return
107     LineNumberTable:
108         line 24: 0
109         line 25: 5
110     LocalVariableTable:
111         Start Length Slot Name Signature
112             0      6     0  this  LSeason;
113             0      6     1 value  I
114

```

```

115     public java.lang.String getName();
116     descriptor: ()Ljava/lang/String;
117     flags: ACC_PUBLIC
118     Code:
119         stack=1, locals=1, args_size=1
120             0: aload_0
121             1: getfield      #3                // Field name:Ljava/lang/String;
122             4: areturn
123     LineNumberTable:
124         line 28: 0
125     LocalVariableTable:
126         Start Length Slot Name Signature
127             0      5     0  this  LSeason;
128
129     public void setName(java.lang.String);
130     descriptor: (Ljava/lang/String;)V
131     flags: ACC_PUBLIC
132     Code:
133         stack=2, locals=2, args_size=2
134             0: aload_0
135             1: aload_1
136             2: putfield      #3                // Field name:Ljava/lang/String;
137             5: return
138     LineNumberTable:
139         line 32: 0
140         line 33: 5
141     LocalVariableTable:
142         Start Length Slot Name Signature
143             0      6     0  this  LSeason;
144             0      6     1  name  Ljava/lang/String;
145
146     static {};
147     descriptor: ()V
148     flags: ACC_STATIC
149     Code:
150         stack=4, locals=0, args_size=0
151             0: new            #4                // class Season
152             3: dup
153             4: iconst_1
154             // 以下就是给每个静态常量构造一个对象
155             // 但是我们发现人家调用的构造器是有一个参数的, String
156             5: ldc            #5                // String 春天
157             7: invokespecial #6                // Method "<init>":
158                 (Ljava/lang/String;)V
159             10: putstatic      #7                // Field SPRING:LSeason;
160             13: new            #4                // class Season
161             16: dup
162             17: iconst_2
163             18: ldc            #8                // String 夏天
164             20: invokespecial #6                // Method "<init>":
165                 (Ljava/lang/String;)V
166             23: putstatic      #9                // Field SUMMER:LSeason;
167             26: new            #4                // class Season
168             29: dup
169             30: iconst_3
170             31: ldc            #10               // String 秋天
171             33: invokespecial #6                // Method "<init>":
172                 (Ljava/lang/String;)V

```

```

170      36: putstatic      #11          // Field AUTUMN:LSeason;
171      39: new              #4           // class Season
172      42: dup
173      43: iconst_4
174      44: ldc              #12          // String 冬天
175      46: invokespecial #6           // Method "<init>":
      (Ljava/lang/String;)V
176      49: putstatic      #13          // Field WINTER:LSeason;
177      52: return
178      LineNumberTable:
179        line 6: 0
180        line 7: 13
181        line 8: 26
182        line 9: 39
183    }
184    SourceFile: "Season.java"

```

这个是枚举的:

```

1  C:\Users\zn\IdeaProjects\untitled\out\production\untitled>javap -v
SeasonEnum.class
2  Classfile
   /C:/Users/zn/IdeaProjects/untitled/out/production/untitled/SeasonEnum.class
3   Last modified 2021-8-28; size 974 bytes
4   MD5 checksum dc612af3d340c0984bbf18b7cfff2e6
5   Compiled from "SeasonEnum.java"
6   public final class SeasonEnum extends java.lang.Enum<SeasonEnum>
7     minor version: 0
8     major version: 52
9     flags: ACC_PUBLIC, ACC_FINAL, ACC_SUPER, ACC_ENUM
10  Constant pool:
11     #1 = Fieldref      #4.#42      // SeasonEnum.$VALUES:[LSeasonEnum;
12     #2 = Methodref     #43.#44     // "[LSeasonEnum;".clone:
      (Ljava/lang/Object;
13     #3 = Class         #23         // "[LSeasonEnum;"
14     #4 = Class         #45         // SeasonEnum
15     #5 = Methodref     #16.#46     // java/lang/Enum.valueOf:
      (Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;
16     #6 = Methodref     #16.#47     // java/lang/Enum."<init>":
      (Ljava/lang/String;I)V
17     #7 = String        #17         // SPRING
18     #8 = Methodref     #4.#47     // SeasonEnum."<init>":
      (Ljava/lang/String;I)V
19     #9 = Fieldref      #4.#48     // SeasonEnum.SPRING:LSeasonEnum;
20     #10 = String       #19         // SUMMER
21     #11 = Fieldref     #4.#49     // SeasonEnum.SUMMER:LSeasonEnum;
22     #12 = String       #20         // AUTUMN
23     #13 = Fieldref     #4.#50     // SeasonEnum.AUTUMN:LSeasonEnum;
24     #14 = String       #21         // WINTER
25     #15 = Fieldref     #4.#51     // SeasonEnum.WINTER:LSeasonEnum;
26     #16 = Class        #52         // java/lang/Enum
27     #17 = Utf8         SPRING
28     #18 = Utf8         LSeasonEnum;
29     #19 = Utf8         SUMMER
30     #20 = Utf8         AUTUMN
31     #21 = Utf8         WINTER
32     #22 = Utf8         $VALUES
33     #23 = Utf8         [LSeasonEnum;

```

```

34     #24 = Utf8          values
35     #25 = Utf8          ()[LSeasonEnum;
36     #26 = Utf8          Code
37     #27 = Utf8          LineNumberTable
38     #28 = Utf8          valueOf
39     #29 = Utf8          (Ljava/lang/String;)LSeasonEnum;
40     #30 = Utf8          LocalVariableTable
41     #31 = Utf8          name
42     #32 = Utf8          Ljava/lang/String;
43     #33 = Utf8          <init>
44     #34 = Utf8          (Ljava/lang/String;I)V
45     #35 = Utf8          this
46     #36 = Utf8          Signature
47     #37 = Utf8          ()V
48     #38 = Utf8          <clinit>
49     #39 = Utf8          Ljava/lang/Enum<LSeasonEnum;>;
50     #40 = Utf8          SourceFile
51     #41 = Utf8          SeasonEnum.java
52     #42 = NameAndType    #22:#23          // $VALUES:[LSeasonEnum;
53     #43 = Class          #23              // "[LSeasonEnum;"
54     #44 = NameAndType    #53:#54          // clone:()Ljava/lang/Object;
55     #45 = Utf8          SeasonEnum
56     #46 = NameAndType    #28:#55          // valueOf:
(Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;
57     #47 = NameAndType    #33:#34          // "<init>":(Ljava/lang/String;I)V
58     #48 = NameAndType    #17:#18          // SPRING:LSeasonEnum;
59     #49 = NameAndType    #19:#18          // SUMMER:LSeasonEnum;
60     #50 = NameAndType    #20:#18          // AUTUMN:LSeasonEnum;
61     #51 = NameAndType    #21:#18          // WINTER:LSeasonEnum;
62     #52 = Utf8          java/lang/Enum
63     #53 = Utf8          clone
64     #54 = Utf8          ()Ljava/lang/Object;
65     #55 = Utf8          (Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;
66 {
67 // 静态常量
68     public static final SeasonEnum SPRING;
69         descriptor: LSeasonEnum;
70         flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL, ACC_ENUM
71
72     public static final SeasonEnum SUMMER;
73         descriptor: LSeasonEnum;
74         flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL, ACC_ENUM
75
76     public static final SeasonEnum AUTUMN;
77         descriptor: LSeasonEnum;
78         flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL, ACC_ENUM
79
80     public static final SeasonEnum WINTER;
81         descriptor: LSeasonEnum;
82         flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL, ACC_ENUM
83
84     public static SeasonEnum[] values();
85         descriptor: ()[LSeasonEnum;
86         flags: ACC_PUBLIC, ACC_STATIC
87         Code:
88             stack=1, locals=0, args_size=0
89             0: getstatic      #1              // Field $VALUES:[LSeasonEnum;

```

```

90      3: invokevirtual #2                // Method "[LSeasonEnum;".clone:
    ()Ljava/lang/Object;
91      6: checkcast      #3                // class "[LSeasonEnum;"
92      9: areturn
93      LineNumberTable:
94      line 1: 0
95
96      public static SeasonEnum valueOf(java.lang.String);
97      descriptor: (Ljava/lang/String;)LSeasonEnum;
98      flags: ACC_PUBLIC, ACC_STATIC
99      Code:
100     stack=2, locals=1, args_size=1
101     0: ldc              #4                // class SeasonEnum
102     2: aload_0
103     3: invokestatic      #5                // Method java/lang/Enum.valueOf:
    (Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;
104     6: checkcast      #4                // class SeasonEnum
105     9: areturn
106     LineNumberTable:
107     line 1: 0
108     LocalVariableTable:
109     Start  Length  Slot  Name   Signature
110     0      10     0    name   Ljava/lang/String;
111
112     static {};
113     descriptor: ()V
114     flags: ACC_STATIC
115     Code:
116     stack=4, locals=0, args_size=0
117     0: new                #4                // class SeasonEnum
118     3: dup
119     4: ldc              #7                // String SPRING
120     6: iconst_0
121     7: invokespecial    #8                // Method "<init>":
    (Ljava/lang/String;I)V
122    10: putstatic        #9                // Field SPRING:LSeasonEnum;
123    13: new                #4                // class SeasonEnum
124    16: dup
125    17: ldc              #10               // String SUMMER
126    19: iconst_1
127    20: invokespecial    #8                // Method "<init>":
    (Ljava/lang/String;I)V
128    23: putstatic        #11               // Field SUMMER:LSeasonEnum;
129    26: new                #4                // class SeasonEnum
130    29: dup
131    30: ldc              #12               // String AUTUMN
132    32: iconst_2
133    33: invokespecial    #8                // Method "<init>":
    (Ljava/lang/String;I)V
134    36: putstatic        #13               // Field AUTUMN:LSeasonEnum;
135    39: new                #4                // class SeasonEnum
136    42: dup
137    43: ldc              #14               // String WINTER
138    45: iconst_3
139    46: invokespecial    #8                // Method "<init>":
    (Ljava/lang/String;I)V
140    49: putstatic        #15               // Field WINTER:LSeasonEnum;
141    52: iconst_4

```



```

142      53: anewarray    #4                  // class SeasonEnum
143      56: dup
144      57: iconst_0
145      58: getstatic    #9                  // Field SPRING:LSeasonEnum;
146      61: astore
147      62: dup
148      63: iconst_1
149      64: getstatic    #11                 // Field SUMMER:LSeasonEnum;
150      67: astore
151      68: dup
152      69: iconst_2
153      70: getstatic    #13                 // Field AUTUMN:LSeasonEnum;
154      73: astore
155      74: dup
156      75: iconst_3
157      76: getstatic    #15                 // Field WINTER:LSeasonEnum;
158      79: astore
159      80: putstatic    #1                  // Field $VALUES:[LSeasonEnum;
160      83: return
161      LineNumberTable:
162          line 2: 0
163          line 1: 52
164  }
165  Signature: #39                  // Ljava/lang/Enum<LSeasonEnum;>;
166  SourceFile: "SeasonEnum.java"

```

```

1  protected Enum(String name, int ordinal) {
2      this.name = name;
3      this.ordinal = ordinal;
4  }

```

常用方法

方法	说明
values() 静态的自动生成的	可以遍历enum实例，其返回enum实例的数组
ordinal() 父类的实例方法	返回每个实例在声明时的次序
name() 父类的实例方法	返回enum实例声明时的名称
getDeclaringClass()	返回其所属的enum类
valueOf() 静态的自动生成的	根据给定的名称返回相应的enum实例

```

1  public static void main(String[] args) {
2      SeasonEnum[] items = SeasonEnum.values();
3      for (int i = 0; i < items.length; i++) {
4          System.out.println(items[i].ordinal());
5          System.out.println(items[i].name());
6          System.out.println(items[i].getDeclaringClass());
7          System.out.println(SeasonEnum.valueOf(SeasonEnum.class,
items[i].name()));
8          System.out.println("-----");
9      }
10 }

```

结果：

```

1  0
2  SPRING
3  class SeasonEnum
4  SPRING
5  -----
6  1
7  SUMMER
8  class SeasonEnum
9  SUMMER
10 -----
11 2
12 AUTUMN
13 class SeasonEnum
14 AUTUMN
15 -----
16 3
17 WINTER
18 class SeasonEnum
19 WINTER
20 -----

```

#2、Enum中添加新方法

- Enum 可以看做是一个常规类（除了不能继承自一个enum），enum 中可以添加方法和 main 方法。

```

1  public enum SeasonEnum {
2      SPRING("春天", "春暖花开的季节"),
3      SUMMER("夏天", "热的要命，但是小姐姐都穿短裤"),
4      AUTUMN("秋天", "果实成熟的季节"),
5      WINTER("冬天", "冷啊，可以吃火锅");
6
7      private String name;
8      private String detail;
9
10     SeasonEnum() {
11     }
12
13     SeasonEnum(String name, String detail) {
14         this.name = name;
15         this.detail = detail;
16     }
17
18     public String getName() {
19         return name;
20     }
21
22     public void setName(String name) {
23         this.name = name;
24     }
25
26     public String getDetail() {
27         return detail;
28     }
29
30     public void setDetail(String detail) {
31         this.detail = detail;
32     }

```

#3、Switch语句中的Enum

- 正确用法

```

1  public static void main(String[] args) {
2      SeasonEnum season = SeasonEnum.SPRING;
3      switch (season){
4          case SPRING:
5              System.out.println("春天来了，又到了万物交配的季节！");
6          case SUMMER:
7              System.out.println("夏天来了，又可以穿大裤衩了！");
8          case AUTUMN:
9              System.out.println("秋天来了，又到了收获的季节！");
10         case WINTER:
11             System.out.println("冬天来了，又到了吃火锅的季节了！");
12         default:
13             System.out.println("也没有别的季节了。");
14     }
15 }

```

- 常规情况下必须使用 enum 类型来修饰 enum 实例，但在 case 语句中不必如此，
- 意思就是 `case SPRING:` 不需要写成 `case SeasonEnum.SPRING:`。

#4、Enum的静态导入

- static import 可以将 enum 实例的标识符带入当前类，无需再用enum类型来修饰 enum 实例

```

1  import static com.ydlclass.SeasonEnum.*;
2
3  public class Test {
4
5      public static void main(String[] args) {
6          System.out.println(SPRING.name());
7          System.out.println(SUMMER.name());
8      }
9  }

```

#5、枚举实现单例设计模式

目前我们的单例设计模式已经实现了三种了：

《Effective Java》

这种方法在功能上与公有域方法相近，但是它更加简洁，无偿提供了序列化机制，绝对防止多次实例化，即使是在面对复杂序列化或者反射攻击的时候。虽然这种方法还没有广泛采用，但是单元素的枚举类型已经成为实现 Singleton的最佳方法。——《Effective Java 中文版 第二版》

```

1  package com.ydlclass;
2
3  public class Singleton {
4
5      private Singleton(){}
6
7      public static Singleton getInstant(){

```

```

8         return SingletonHolder.INSTANT.instant;
9     }
10
11     private enum SingletonHolder{
12         INSTANT;
13         private final Singleton instant;
14
15         SingletonHolder(){
16             instant = new Singleton();
17         }
18     }
19
20
21     public static void main(String[] args) {
22         System.out.println(Singleton.getInstant() == Singleton.getInstant());
23     }
24
25 }

```

#6、枚举的优势

阿里《Java开发手册》对枚举的相关规定如下，我们在使用时需要稍微注意一下。

【强制】所有的枚举类型字段必须要有注释，说明每个数据项的用途。

【参考】枚举类名带上 Enum 后缀，枚举成员名称需要全大写，单词间用下划线隔开。说明：枚举其实就是特殊的常量类，且构造方法被默认强制是私有。正例：枚举名字为 ProcessStatusEnum 的成员名称：SUCCESS / UNKNOWN_REASON。

第一，`int` 类型本身并不具备安全性，假如某个程序员在定义 `int` 时少些了一个 `final` 关键字，那么就会存在被其他人修改的风险，而反观枚举类，它“天然”就是一个常量类，不存在被修改的风险（原因详见下半部分）；

第二，使用 `int` 类型的语义不够明确，比如我们在控制台打印时如果只输出 1...2...3 这样的数字，我们肯定不知道它代表的是什么含义。

那有人就说了，那就使用常量字符呗，这总不会还不知道语义吧？实现示例代码如下：

```

1 public static final String COLOR_RED = "RED";
2 public static final String COLOR_BLUE = "BLUE";
3 public static final String COLOR_GREEN = "GREEN";

```

但是这样同样存在一个问题，有些初级程序员会不按套路出牌，他们可能会直接使用字符串的值进行比较，而不是直接使用枚举的字段，实现示例代码如下：

```

1 public class EnumTest {
2
3     public static final String COLOR_RED = "RED";
4     public static final String COLOR_BLUE = "BLUE";
5     public static final String COLOR_GREEN = "GREEN";
6
7     public static void main(String[] args) {
8         String color = "BLUE";
9         if ("BLUE".equals(color)) {
10             System.out.println("蓝色");
11         }
12     }

```

```
13     }  
14 }
```

这样当我们修改了枚举中的值，那程序就凉凉了。

枚举比较推荐使用 ==