

第11章 泛型

学习目标

- ☐ 能够使用泛型定义类、接口、方法
- ☐ 能够理解泛型上限
- ☐ 能够阐述泛型通配符的作用
- ☐ 能够识别通配符的上下限

第十一章 泛型

11.1 泛型的概念

11.1.1 泛型的引入

Generics /dʒəˈnɛrɪks/

例如：生产瓶子的厂家，一开始并不知道我们将来会用瓶子装什么，我们什么都可以装，但是有的时候，我们在使用时，想要限定某个瓶子只能用来装什么，这样我们不会装错，而用的时候也可以放心的使用，无需再三思量。我们生活中是**在使用这个瓶子时在瓶子上“贴标签”**，这样就轻松解决了问题。



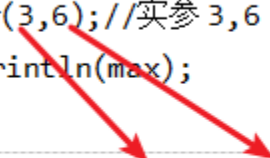
还有，在Java中我们在声明方法时，当在完成方法功能时如果有未知的数据需要参与，这些未知的数据需要在调用方法时才能确定，那么我们把这样的数据通过形参表示。那么在方法体中，用这个形参名来代表那个未知的数据，而调用者在调用时，对应的传入值就可以了。

```

public static void main(String[] args) {
    int max = max(3,6); //实参 3,6
    System.out.println(max);
}

public static int max(int a, int b) { //形参 a,b
    return a > b ? a : b;
}

```



受以上两点启发，JDK1.5设计了泛型的概念。泛型即为“类型参数”，这个类型参数在声明它的类、接口或方法中，代表未知的通用的类型。例如：

java.lang.Comparable接口和java.util.Comparator接口，是用于对象比较大小的规范接口，这两个接口只是限定了当一个对象大于另一个对象时返回正整数，小于返回负整数，等于返回0。但是并不确定是什么类型的对象比较大，之前的时候只能用Object类型表示，使用时既麻烦又不安全，因此JDK1.5就给他们增加了泛型。

```

public interface Comparable<T>{
    int compareTo(T o) ;
}

```

```

public interface Comparator<T>{
    int compare(T o1, T o2) ;
}

```

其中就是类型参数，即泛型。

11.1.2 泛型的好处

示例代码：

JavaBean：圆类型

```

class Circle{
    private double radius;

    public Circle(double radius) {
        super();
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public String toString() {
        return "Circle [radius=" + radius + "]";
    }
}

```

```
}
```

比较器

```
import java.util.Comparator;

public class CircleComparator implements Comparator{

    @Override
    public int compare(Object o1, Object o2) {
        //强制类型转换
        Circle c1 = (Circle) o1;
        Circle c2 = (Circle) o2;
        return Double.compare(c1.getRadius(), c2.getRadius());
    }

}
```

测试类

```
public class TestGeneric {
    public static void main(String[] args) {
        CircleComparator com = new CircleComparator();
        System.out.println(com.compare(new Circle(1), new Circle(2)));

        System.out.println(com.compare("圆1", "圆2")); //运行时异常:
        ClassNotFoundException
    }
}
```

那么我们在使用如上面这样的接口时，如果没有泛型或不指定泛型，很麻烦，而且有安全隐患。

因为在设计（编译）Comparator接口时，不知道它会用于哪种类型的对象比较，因此只能将compare方法的形参设计为Object类型，而实际在compare方法中需要向下转型为Circle，才能调用Circle类的getRadius()获取半径值进行比较。

使用泛型：

比较器：

```
class CircleComparator implements Comparator<Circle>{

    @Override
    public int compare(Circle o1, Circle o2) {
        //不再需要强制类型转换，代码更简洁
        return Double.compare(o1.getRadius(), o2.getRadius());
    }

}
```

测试类

```
import java.util.Comparator;

public class TestGeneric {
    public static void main(String[] args) {
        CircleComparator com = new CircleComparator();
        System.out.println(com.compare(new Circle(1), new Circle(2)));

        //      System.out.println(com.compare("圆1", "圆2")); //编译错误，因为"圆1", "圆2"不是Circle类型，是String类型，编译器提前报错，而不是冒着风险在运行时再报错
    }
}
```

如果有了泛型并使用泛型，那么既能保证安全，又能简化代码。

因为把不安全的因素在编译期间就排除了；既然通过了编译，那么类型一定是符合要求的，就避免了类型转换。

11.1.3 泛型的相关名词

<类型>这种语法形式就叫泛型。

- GenericArrayType: 泛化的数组类型，即T[]
- ParameterizedType: 参数化类型，例如：Comparator, Comparator
- TypeVariable: 类型变量，例如：Comparator中的T, Map<K,V>中的K,V
- WildcardType: 通配符类型，例如：Comparator<?>等

11.1.4 在哪里可以声明类型变量<T>

- 声明类或接口时，在类名或接口名后面声明类型变量，我们把这样的类或接口称为泛型类或泛型接口

```
【修饰符】 class 类名<类型变量列表> 【extends 父类】 【implements 父接口们】 {
}
【修饰符】 interface 接口名<类型变量列表> 【implements 父接口们】 {
}

例如：
public class ArrayList<E>
public interface Map<K,V>{
    ....
}
```

- 声明方法时，在【修饰符】与返回值类型之间声明类型变量，我们把声明（是**声明**不是单纯的使用）了类型变量的方法称为泛型方法

```
【修饰符】 <类型变量列表> 返回值类型 方法名(【形参列表】)【throws 异常列表】 {
    //...
}

例如：java.util.Arrays类中的
public static <T> List<T> asList(T... a){
    ....
}
```

11.2 自定义泛型结构

11.2.1 自定义泛型类和泛型接口

当我们在声明类或接口时，类或接口中定义某个成员时，该成员有些类型是不确定的，而这个类型需要在使用这个类或接口时才可以确定，那么我们可以使用泛型。

1. 声明泛型类与泛型接口

语法格式：

```
【修饰符】 class 类名<类型变量列表> 【extends 父类】 【implements 父接口们】 {  
  
}  
【修饰符】 interface 接口名<类型变量列表> 【implements 父接口们】 {  
  
}
```

注意：

- <类型变量列表>：可以是一个或多个类型变量，一般都是使用单个的大写字母表示。例如：、<K,V> 等。
- 当类或接口上声明了<类型变量列表>时，其中的类型变量不能用于静态成员上。

示例：

```
//泛型接口  
public interface GerInterface<T> {  
    void show(T t);  
}
```

```
//泛型类  
public class GerInterfaceImpl<T> implements GerInterface<T> {  
    private T t;  
  
    public T getT() {  
        return t;  
    }  
  
    public void setT(T t) {  
        this.t = t;  
    }  
  
    @Override  
    public void show(T t) {  
        System.out.println(t);  
    }  
  
    // public static void test(T t){ }           //此时类型变量T不能用在静态成员上  
}
```

2. 使用自定义泛型类和接口

在使用这种参数化的类与接口时，我们需要指定泛型变量的实际类型参数：

- 实际类型参数必须是引用数据类型，不能是基本数据类型

- 在创建类的对象时指定类型变量对应的实际类型参数
- 在继承泛型类或实现泛型接口时，可以指定类型变量对应的实际类型参数
- 当使用参数化类型的类或接口时，如果没有指定泛型，即为泛型擦除，相当于Object类型。

```
public static void main(String[] args) {
    GerInterface<String> gi = new GerInterfaceImpl<String>();
    //gi.show(123); //泛型确定了String, 这里编译失败
    gi.show("hello");
}
```

指定泛型实参时，必须左右两边一致。JDK1.7支持简写形式：

```
GerInterface<String> gi = new GerInterfaceImpl<>(); //省略右边泛型类型
```

练习：

我们要声明一个学生类，该学生包含姓名、成绩，而此时学生的成绩类型不确定，为什么呢，因为，语文老师希望成绩是“优秀”、“良好”、“及格”、“不及格”，数学老师希望成绩是89.5, 65.0，英语老师希望成绩是'A','B','C','D','E'。那么我们在设计这个学生类时，就可以使用泛型。

定义泛型类：

```
public class Student<T>{
    private String name;
    private T score;

    public Student() {
        super();
    }
    public Student(String name, T score) {
        super();
        this.name = name;
        this.score = score;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public T getScore() {
        return score;
    }
    public void setScore(T score) {
        this.score = score;
    }
    @Override
    public String toString() {
        return "姓名: " + name + ", 成绩: " + score;
    }
}
```

使用泛型类：

```
public class TestGeneric{
```

```

public static void main(String[] args) {
    //语文老师使用时:
    Student<String> stu1 = new Student<String>("张三", "良好");

    //数学老师使用时:
    //Student<double> stu2 = new Student<double>("张三", 90.5); //错误, 必须
是引用数据类型
    Student<Double> stu2 = new Student<Double>("张三", 90.5);

    //英语老师使用时:
    Student<Character> stu3 = new Student<Character>("张三", 'c');

    //错误的指定
    //Student<Object> stu = new Student<String>(); //错误的
}
}

```

继承泛型类并指定类型变量:

```

class ChineseStudent extends Student<String>{//继承时确定了泛型类型

    public ChineseStudent() {
        super();
    }

    public ChineseStudent(String name, String score) {
        super(name, score);
    }

}

```

```

public class TestGeneric{
    public static void main(String[] args) {
        //语文老师使用时:
        ChineseStudent stu = new ChineseStudent("张三", "良好");
    }
}

```

```

class Circle implements Comparable<Circle>{//实现类确定了泛型类型
    private double radius;

    public Circle(double radius) {
        super();
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override

```

```

    public String toString() {
        return "Circle [radius=" + radius + "]";
    }

    @Override
    public int compareTo(Circle c){
        return Double.compare(radius,c.radius);
    }

}

```

类型擦除:

```

public class CircleComparator implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        //未指定泛型类型，默认为Object，使用时还要强制类型转换
        Circle c1 = (Circle) o1;
        Circle c2 = (Circle) o2;
        return Double.compare(c1.getRadius(), c2.getRadius());
    }
}

```

}

11.2.2 自定义泛型方法

前面介绍了在定义类、接口时可以声明<类型变量>，在该类的方法和属性定义、接口的方法定义中，这些<类型变量>可被当成普通类型来用。**单独的方法也可以泛型化，称为泛型方法。**

语法格式:

```

【修饰符】 <类型变量列表> 返回值类型 方法名(【形参列表】)【throws 异常列表】{
    //...
}

```

- 静态方法也可以单独泛型化。区别泛型类或接口中的方法（静态方法不能使用泛型类或接口定义的泛型变量）。
- <类型变量列表>：可以是一个或多个类型变量，一般都是使用单个的大写字母表示。例如：、<K,V>等。

示例:

```

public class GernericMethod {
    //泛型方法
    public static <T> T getMsg(T t){
        return t;
    }
}

```

```

public static void main(String[] args) {
    GernericMethod.getMsg("hello");
}

```

练习:

我们编写一个数组工具类，包含可以给任意对象数组进行从小到大排序，要求数组元素类型必须实现 Comparable接口

```
public class MyArrays{
    public static <T extends Comparable<T>> void sort(T[] arr){
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length-i; j++) {
                if(arr[j].compareTo(arr[j+1])>0){
                    T temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}
```

测试类

```
public class TestGeneric{
    public static void main(String[] args) {
        int[] arr = {3,2,5,1,4};
        // MyArrays.sort(arr); //错误的，因为int[]不是对象数组

        String[] strings = {"hello","java","chai"};
        MyArrays.sort(strings);
        System.out.println(Arrays.toString(strings));

        Circle[] circles = {new Circle(2.0),new Circle(1.2),new Circle(3.0)};
        MyArrays.sort(circles);
        System.out.println(Arrays.toString(circles));
    }
}
```

练习

1. 练习1

1、声明一个坐标类Coordinate，它有两个属性：x,y，都为T类型 2、在测试类中，创建两个不同的坐标类对象，分别指定T类型为String和Double，并为x,y赋值，打印对象

```
public class TestExer1 {
    public static void main(String[] args) {
        Coordinate<String> c1 = new Coordinate<>("北纬38.6", "东经36.8");
        System.out.println(c1);

        // Coordinate<Double> c2 = new Coordinate<>(38.6, 38); //自动装箱与拆箱只能
        // 与对应的类型 38是int，自动装为Integer
        Coordinate<Double> c2 = new Coordinate<>(38.6, 36.8);
        System.out.println(c2);
    }
}

class Coordinate<T>{
    private T x;
    private T y;
}
```

```

    public Coordinate(T x, T y) {
        super();
        this.x = x;
        this.y = y;
    }
    public Coordinate() {
        super();
    }
    public T getX() {
        return x;
    }
    public void setX(T x) {
        this.x = x;
    }
    public T getY() {
        return y;
    }
    public void setY(T y) {
        this.y = y;
    }
    @Override
    public String toString() {
        return "Coordinate [x=" + x + ", y=" + y + "]";
    }
}

```

2. 练习2

1、声明一个Person类，包含姓名和伴侣属性，其中姓名是String类型，而伴侣的类型不确定，因为伴侣可以是Person，可以是Animal（例如：金刚），可以是Ghost鬼（例如：倩女幽魂），可以是Demon妖（例如：白娘子），可以是Robot机器人（例如：剪刀手爱德华）。。。

2、在测试类中，创建Person对象，并为它指定伴侣，打印显示信息

```

public class TestExer3 {
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static void main(String[] args) {
        Person<Demon> xu = new Person<Demon>("许仙", new Demon("白娘子"));
        System.out.println(xu);

        Person<Person> xie = new Person<Person>("谢学建", new Person("徐余
        龙"));
        Person fere = xie.getFere();
        fere.setFere(xie);
        System.out.println(xie);
        System.out.println(fere);
    }
}
class Demon{
    private String name;

    public Demon(String name) {
        super();
        this.name = name;
    }
}

```

```

@Override
public String toString() {
    return "Demon [name=" + name + "]";
}
}
class Person<T>{
    private String name;
    private T fere;
    public Person(String name, T fere) {
        super();
        this.name = name;
        this.fere = fere;
    }
    public Person(String name) {
        super();
        this.name = name;
    }

    public Person() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public T getFere() {
        return fere;
    }
    public void setFere(T fere) {
        this.fere = fere;
    }
    @SuppressWarnings("rawtypes")
    @Override
    public String toString() {
        if(fere instanceof Person){
            Person p = (Person) fere;
            return "Person [name=" + name + ", fere=" + p.getName() + "]";
        }
        return "Person [name=" + name + ", fere=" + fere + "]";
    }
}

```

3. 练习3

- 1、声明员工类型Employee，包含姓名（String），薪资（double），年龄（int）
- 2、员工类Employee实现java.lang.Comparable接口，指定T为Employee类型，重写抽象方法，按照薪资比较大小，薪资相同的按照姓名的自然顺序比较大小。
- 3、在测试类中创建Employee数组，然后调用Arrays.sort(Object[] arr)方法进行排序，遍历显示员工信息
- 4、再次调用Arrays.sort(Object[] arr,Comparator c)方法进行按照年龄排序，年龄相同的安装姓名自然顺序比较大小，遍历显示员工信息

```

public class TestExer3 {

```

```

@Test
public void test01() {
    Employee[] arr = new Employee[3];
    arr[0] = new Employee("Irene", 18000, 18);
    arr[1] = new Employee("Jack", 14000, 28);
    arr[2] = new Employee("Alice", 14000, 24);

    Arrays.sort(arr);

    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}

@Test
public void test02() {
    Employee[] arr = new Employee[3];
    arr[0] = new Employee("Irene", 18000, 18);
    arr[1] = new Employee("Jack", 14000, 28);
    arr[2] = new Employee("Alice", 14000, 24);

    //Arrays.sort(T[] arr, Comparator<T> c)
    Arrays.sort(arr, new Comparator<Employee>() {

        //按照年龄排序, 年龄相同的安装姓名自然顺序比较大小
        @Override
        public int compare(Employee o1, Employee o2) {
            if(o1.getAge() != o2.getAge()) {
                return o1.getAge() - o2.getAge();
            }
            return o1.getName().compareTo(o2.getName());
        }
    });

    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}

class Employee implements Comparable<Employee>{
    private String name;
    private double salary;
    private int age;
    public Employee(String name, double salary, int age) {
        super();
        this.name = name;
        this.salary = salary;
        this.age = age;
    }
    public Employee() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

```

    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", salary=" + salary + ", age=" +
age + "]";
    }

    //重写抽象方法，按照薪资比较大小，薪资相同的按照姓名的自然顺序比较大小。
    @Override
    public int compareTo(Employee o) {
        if(this.salary != o.salary) {
            return Double.compare(this.salary, o.salary);
        }
        return this.name.compareTo(o.name); //name是String类型，有compareTo方法
    }
}

```

11.3 类型通配符

当我们声明一个变量/形参时，这个变量/形参的类型是一个泛型类或泛型接口，例如：Comparator类型，但是我们仍然无法确定这个泛型类或泛型接口的类型变量的具体类型，此时我们考虑使用类型通配符。

例如：

这个学生类是一个参数化的泛型类，代码如下（详细请看§11.2.1中的示例说明）：

```

public class Student<T>{
    private String name;
    private T score;

    public Student() {
        super();
    }
    public Student(String name, T score) {
        super();
        this.name = name;
        this.score = score;
    }
    public String getName() {
        return name;
    }
}

```

```

public void setName(String name) {
    this.name = name;
}
public T getScore() {
    return score;
}
public void setScore(T score) {
    this.score = score;
}
@Override
public String toString() {
    return "姓名: " + name + ", 成绩: " + score;
}
}

```

11.4.1 <?>任意类型

例如：我们要声明一个学生管理类，这个管理类要包含一个方法，可以遍历学生数组。

学生管理类：

```

class StudentService {
    public static void print(Student<?>[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}

```

测试类

```

public class TestGeneric {
    public static void main(String[] args) {
        // 语文老师使用时：
        Student<String> stu1 = new Student<String>("张三", "良好");

        // 数学老师使用时：
        // Student<double> stu2 = new Student<double>("张三", 90.5); // 错误，必须是
引用数据类型
        Student<Double> stu2 = new Student<Double>("张三", 90.5);

        // 英语老师使用时：
        Student<Character> stu3 = new Student<Character>("张三", 'C');

        Student<?>[] arr = new Student[3];
        arr[0] = stu1;
        arr[1] = stu2;
        arr[2] = stu3;

        StudentService.print(arr);
    }
}

```

11.4.2 <? extends 上限>

例如：我们要声明一个学生管理类，这个管理类要包含一个方法，找出学生数组中成绩最高的学生对象。

要求学生的成绩的类型必须可比较大小，实现Comparable接口。

学生管理类：

```
class StudentService {
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Student<? extends Comparable> max(Student<? extends Comparable>[] arr){
        Student<? extends Comparable> max = arr[0];
        for (int i = 0; i < arr.length; i++) {
            if(arr[i].getScore().compareTo(max.getScore())>0){
                max = arr[i];
            }
        }
        return max;
    }
}
```

测试类

```
public class TestGeneric {
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static void main(String[] args) {
        Student<? extends Double>[] arr = new Student[3];
        arr[0] = new Student<Double>("张三", 90.5);
        arr[1] = new Student<Double>("李四", 80.5);
        arr[2] = new Student<Double>("王五", 94.5);

        Student<? extends Comparable> max = StudentService.max(arr);
        System.out.println(max);
    }
}
```

11.4.3 <? super 下限>

现在要声明一个数组工具类，包含可以给任意对象数组进行从小到大排序，只要你指定定制比较器对象，而且这个定制比较器对象可以是当前数组元素类型自己或其父类的定制比较器对象

数组工具类：

```
class MyArrays{
    public static <T> void sort(T[] arr, Comparator<? super T> c){
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length-i; j++) {
                if(c.compare(arr[j], arr[j+1])>0){
                    T temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}
```

例如：有如下JavaBean

```
class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public Person() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "name=" + name + ", age=" + age;
    }
}

class Student extends Person{
    private int score;

    public Student(String name, int age, int score) {
        super(name, age);
        this.score = score;
    }

    public Student() {
        super();
    }

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return super.toString() + ",score=" + score;
    }
}
```



```

public class TestGeneric {
    public static void main(String[] args) {
        Student[] all = new Student[3];
        all[0] = new Student("张三", 23, 89);
        all[1] = new Student("李四", 22, 99);
        all[2] = new Student("王五", 25, 67);

        MyArrays.sort(all, new Comparator<Person>() {

            @Override
            public int compare(Person o1, Person o2) {
                return o1.getAge() - o2.getAge();
            }
        });

        System.out.println(Arrays.toString(all));

        MyArrays.sort(all, new Comparator<Student>() {

            @Override
            public int compare(Student o1, Student o2) {
                return o1.getScore() - o2.getScore();
            }
        });
        System.out.println(Arrays.toString(all));
    }
}

```

11.4.4 使用类型通配符来指定类型参数的问题

：不可变，因为类型不确定，编译时，任意类型都是错

<? extends 上限>：因为<? extends 上限>的?可能是上限或上限的子类，即类型不确定，编译按任意类型处理都是错。

<? super 下限>：可以将值修改为下限或下限子类的对象，因为<? super 下限>代表是下限或下限的父类，那么设置为下限或下限子类的对象是安全的。

```

public class TestGeneric {
    public static void main(String[] args) {
        Student<?> stu1 = new Student<>();
        stu1.setScore(null); //除了null，无法设置为其他值

        Student<? extends Number> stu2 = new Student<>();
        stu2.setScore(null); //除了null，无法设置为其他值

        Student<? super Number> stu3 = new Student<>();
        stu3.setScore(56); //可以设置Number或其子类的对象
    }
}

class Student<T>{
    private String name;
    private T score;
}

```

```

public Student() {
    super();
}
public Student(String name, T score) {
    super();
    this.name = name;
    this.score = score;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public T getScore() {
    return score;
}
public void setScore(T score) {
    this.score = score;
}
@Override
public String toString() {
    return "姓名: " + name + ", 成绩: " + score;
}
}

```

练习

在数组工具类中声明如下泛型方法：

- (1) 可以在任意类型的对象数组中，查找某个元素的下标，按照顺序查找，如果有重复的，就返回第一个找到的，如果没有返回-1
- (2) 可以在任意类型的对象数组中，查找最大值，要求元素必须实现Comparable接口
- (3) 可以在任意类型的对象数组中，查找最大值，按照指定定制比较器来比较元素大小
- (4) 可以给任意对象数组进行从小到大排序，要求数组元素类型必须实现Comparable接口
- (5) 可以给任意对象数组进行从小到大排序，只要你指定定制比较器对象，不要求数组元素实现Comparable接口
- (6) 可以将任意对象数组的元素拼接为一个字符串返回

```

public class MyArrays {
    //可以在任意类型的对象数组中，查找某个元素的下标，按照顺序查找，如果有重复的，就返回第一个找到的，如果没有返回-1
    public static <T> int find(T[] arr, T value) {
        for (int i = 0; i < arr.length; i++) {
            if(arr[i].equals(value)) { //使用==比较太严格，使用equals方法，因为任意对象都有equals方法
                return i;
            }
        }
        return -1;
    }

    //可以在任意类型的对象数组中，查找最大值，要求元素必须实现Comparable接口
}

```

```

public static <T extends Comparable<? super T>> T max(T[] arr) {
    T max = arr[0];
    for (int i = 0; i < arr.length; i++) {
        if(max.compareTo(arr[i])<0) { //if(max < arr[i]) {
            max = arr[i];
        }
    }
    return max;
}

```

//可以在任意类型的对象数组中，查找最大值，按照指定定制比较器来比较元素大小

```

public static <T> T max(T[] arr, Comparator<? super T> c) {
    T max = arr[0];
    for (int i = 0; i < arr.length; i++) {
        if(c.compare(max, arr[i])<0) { //if(max < arr[i]) {
            max = arr[i];
        }
    }
    return max;
}

```

//可以给任意对象数组进行从小到大排序，要求数组元素类型必须实现Comparable接口

```

public static <T extends Comparable<? super T>> void sort(T[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
        int minIndex = i;
        for (int j = i+1; j < arr.length; j++) {
            if(arr[minIndex].compareTo(arr[j])>0) {
                minIndex = j;
            }
        }
        if(minIndex!=i) {
            T temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
}

```

//可以给任意对象数组进行从小到大排序，只要你指定定制比较器对象，不要求数组元素实现Comparable接口

```

public static <T> void sort(T[] arr, Comparator<? super T> c) {
    for (int i = 0; i < arr.length-1; i++) {
        int minIndex = i;
        for (int j = i+1; j < arr.length; j++) {
            if(c.compare(arr[minIndex],arr[j])>0) {
                minIndex = j;
            }
        }
        if(minIndex!=i) {
            T temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
}

```

//可以将任意对象数组的元素拼接为一个字符串返回

```

public static <T> String toString(T[] arr) {

```

```
String str = "[";
for (int i = 0; i < arr.length; i++) {
    if(i==0) {
        str += arr[i];
    }else {
        str += "," + arr[i];
    }
}
str += "]";
return str;
}
}
```