1. 泛型
   1. 参数化类型。将类型作为参数，在调用过程中传入那种类型参数就是那种类型。
   2. 泛型类

class Cname<T>{ T：参数类型(形参)

private T t;

}

* 1. 泛型好处:
     1. 在编译期确定类型，保证类型的安全性
  2. 泛型使用只能使用类类型

1. ?（参数类型） 和 类型参数化(T)
   1. ?设置上界和下界，T只能设置上界
   2. ?只能设置一个上界，T可以设置多个上界
   3. 如果有多个上界使用&符号进行连接，该参数必须同时满足多个上界的需求。

|  |
| --- |
| //无界 通配符 ?    //数字 设置上界 extends(类型只能是该类及其该类的子类 Integer Float Double)  public void show(Test4<? extends Number> test){  System.out.println(test.getT());  }    //设置下界 super(类型只能是该类及其该类父类)  public void show2(Test4<? super Number> test){  System.out.println(test.getT());  }  //泛型(参数化类型)  //可以设置上界  //可以设置多个上界  //T:type E:element K:key v:value  **class** Test4<T **extends** Base & IA>{    **private** T t;  **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }    } |

1. 泛型接口，泛型构造器，泛型方法

|  |
| --- |
| //泛型构造器 泛型接口 泛型方法  **class** PersonN{  //构造方法  **public** <T> PersonN(T t) {  System.***out***.println(t);  }    //方法使用泛型  **public** <T> T show(T t){  **return** t;  }    }  //泛型接口使用  **interface** IA1<T>{  **void** show(T t);  }  //实现类仍旧不确定类型  **class** IAImpl<T> **implements** IA1<T>{  @Override  **public** **void** show(T t) {    }  }  //实现类确定类型  **class** IAImpl2 **implements** IA1<String>{  @Override  **public** **void** show(String t) {    }  } |

1. 类型擦除
   1. 针对于类型参数
      1. 无界<?> object
      2. 有上界的 使用上界
      3. 有多个上界 第一个上界类型
2. 自然排序
   1. 类必须实现compareable接口，同时重写compareTo方法。
   2. 外置比较器

|  |
| --- |
| **class** Student **implements** Comparable<Student>{    **private** **int** no;  **private** **int** age;    **public** Student(**int** no, **int** age) {  **super**();  **this**.no = no;  **this**.age = age;  }  **public** **int** getNo() {  **return** no;  }  **public** **void** setNo(**int** no) {  **this**.no = no;  }    **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }    @Override  **public** String toString() {  **return** "Student [no=" + no + ", age=" + age + "]";  }  //升序  @Override  **public** **int** compareTo(Student o) {  /\*if(this.no > o.no){  return 1;//正数 升序  }else if(this.no < o.no){  return -1;//负数  }else{  return 0;  }\*/  **return** **this**.no - o.no;  }  //比较  /\*@Override  public int compareTo(Object o) {  // this.no o.no  // **TODO** Auto-generated method stub  return 0;  }\*/    }  外置比较器  /\*Arrays.sort(stus, new Comparator<Student>() {    @Override  public int compare(Student o1, Student o2) {  return o1.getAge() - o2.getAge();  }  });\*/  Arrays.*sort*(stus,(s1,s2)->{**return** s1.getAge()-s2.getAge();}) |

1. 枚举
   1. 穷举法。
      1. 数据量不能过大
      2. 数据必须是确定
   2. 枚举类声明
      1. enum关键字声明枚举类
      2. 枚举类中可以声明属性，方法，构造方法等
      3. 如果重写了构造方法，声明对象时需要使用该构造器进行赋值。
   3. 使用场景
      1. valueof():赋值
      2. switch语句中

|  |
| --- |
| Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入值:");  Gender1 gender = Gender1.*valueOf*(sc.next());  **switch** (gender) {  **case** ***MALE***:  System.***out***.println("男的");  **break**;  **case** ***FEMALE***:  System.***out***.println("女的");  **break**;  } |