Java07-集合与泛型

写在最前

此Task知识点多而杂，完整记录本人学习历程，超长篇幅预警，感谢审核人（的眼睛和时间）。

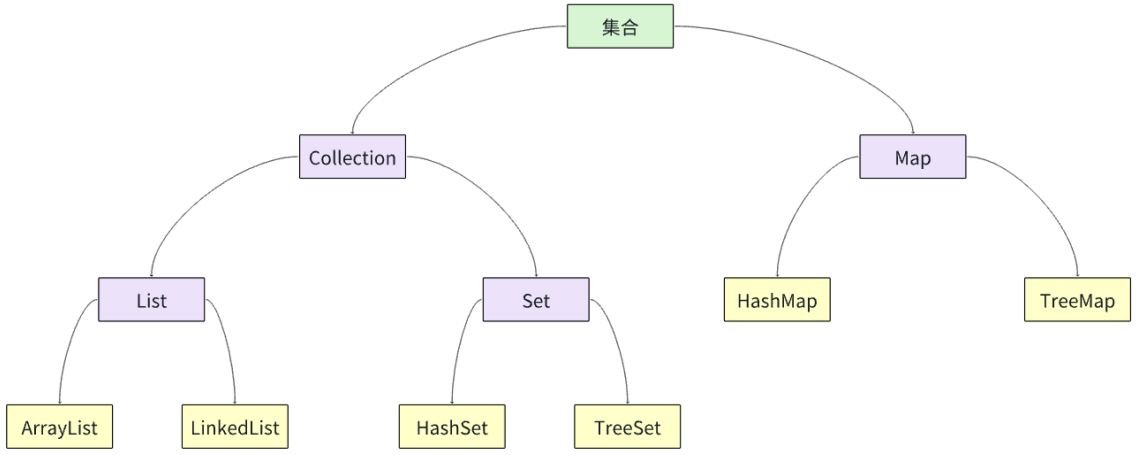
那我们开始吧。

Task1.集合

Task1以下内容都算对Q1的学习/回答：（超长预警）

何为集合

在 Java 中，**集合（Collection）是一个用来存储和管理一组对象的容器**，我们可以把它想象成一个更强大、更灵活的“**数组**”。



**紫色方框是接口 而黄色方框是其实现类*

集合 vs. 数组

特性	数组	集合
长度	固定，一旦创建不可改变	**动态可变**，可根据需要自动扩容
存储类型	可以存储基本数据类型（`int`，`char` 等）和对象	**只能存储对象引用**（但通过自动装箱/拆箱可以方便地处理基本类型，如 `ArrayList<Integer>`）
功能	功能简单，主要就是通过索引存取	功能丰富，提供了各种算法（排序、查找等）、数据结构（链表、栈、队列、哈希表等）
安全性	需要程序员自己控制边界，容易越界	有完善的内部机制，使用更安全

集合框架——核心接口

1. `Collection` 接口

——存储**单个对象**的序列。

`Collection` 是所有单列集合的根接口，它有三个主要的子接口：

- `List`（**列表/序列**）：
  - **特点**：**元素有序、可重复。**
  - **关注索引**：可以通过索引（类似数组的下标）来精确操作元素。
  - **常用实现类**：`ArrayList`，`LinkedList`，`Vector`。
- `Set`（**集**）：
  - **特点**：**元素无序、不可重复。**就像数学中的集合。
  - **用途**：用于存储唯一的元素，判断一个对象是否已存在。
  - **常用实现类**：`HashSet`，`LinkedHashSet`，`TreeSet`。
- `Queue`（**队列**）：
  - **特点**：**先进先出（FIFO）** 的特定存储规则。
  - **用途**：模拟排队场景，用于实现任务调度、消息传递等。
  - **常用实现类**：`LinkedList`（也实现了 `Queue`），`PriorityQueue`。

2. `Map` 接口

——存储**键值对（Key-Value）**映射。

`Map` 是双列集合的根接口，它存储的是“键值对”。

- **特点**：
  - **键（Key）唯一**：每个键最多只能映射到一个值。
  - **值（Value）可重复**：不同的键可以对应相同的值。
  - **通过键找值**：通过键（Key）来检索对应的值（Value），效率很高。
- **常用实现类**：`HashMap`，`LinkedHashMap`，`TreeMap`，`Hashtable`。

常用实现类

学习了上图中出现的常用核心类。

~~（我也不知道为啥 `Set` 和 `Map` 的标题没有在大纲上展示出来啊啊啊啊啊。）~~

`List` 的实现类

1. `ArrayList`（**最常用**）
   - **底层结构**：**动态数组**
   - **优点**：
     - 因为基于数组，所以**根据索引随机访问元素的速度极快**（时间复杂度 O(1)）。
   - **缺点**：
     - 在列表中间进行**插入和删除操作速度较慢**（需要移动后续元素，时间复杂度 O(n)）。

- **适用场景**：绝大多数需要“读多写少”的场景。
- **使用方法**：**（集合基本都是围绕增删改查）**

操作	方法	例子	返回值
创建	`ArrayList<类型> 名字 = new ArrayList<>();`	`ArrayList<String> list = new ArrayList<>();`	-
添加	`.add(元素)`	`list.add("Hello");`	是否成功(true)
获取	`.get(索引)`	`String s = list.get(0);`	该位置的元素
大小	`.size()`	`int len = list.size();`	元素个数
删除	`.remove(索引)`	`list.remove(0);`	被删除的元素
	`.remove(元素)`	`list.remove("Hello");`	是否成功(true)
检查	`.contains(元素)`	`boolean has = list.contains("Hello");`	true 或 false

2. `LinkedList`

链表就是一种像“寻宝游戏”一样，通过“指针”把零散的数据连接起来的数据结构。它的优势是能灵活、高效地增加或删除数据，但缺点是查找效率不高。

- **术语解析**：

1. **节点（Node）**：链表的基本单位。
2. **数据（Data）**：节点里存储的**数据**（比如一个数字、一段文字）。
3. **指针（Pointer）**：寻宝游戏里，纸条上写的“下一个线索藏在哪里”就是**指针**。它不存储实际数据，只负责指向下一个节点。

- **底层结构**：**双向链表。**
- **优点**：
  - 在链表头尾进行**插入和删除操作速度极快**（时间复杂度 O(1)）。
- **缺点**：
  - **随机访问速度慢**（需要从头部或尾部开始遍历，时间复杂度 O(n)）。
- **适用场景**：需要频繁在集合首尾进行添加/删除操作的场景，或者可以用来实现栈和队列。
- **使用方法**：

操作类别	方法名	功能描述	示例代码	返回值
创建链表	`LinkedList<Type>()`	创建空链表	`LinkedList<String> list = new LinkedList<>();`	-
添加元素	`add(E e)`	在**尾部**添加元素	`list.add("Apple");`	`boolean`
	`addFirst(E e)`	在**头部**添加元素	`list.addFirst("First");`	`void`
	`addLast(E e)`	在**尾部**添加元素	`list.addLast("Last");`	`void`
	`add(int index, E e)`	在**指定位置**插入元素	`list.add(1, "Middle");`	`void`
	`offer(E e)`	在尾部添加元素（队列操作）	`list.offer("New");`	`boolean`
删除元素	`remove()`	删除**头部**元素	`list.remove();`	被删除的元素
	`removeFirst()`	删除**头部**元素	`list.removeFirst();`	被删除的元素
	`removeLast()`	删除**尾部**元素	`list.removeLast();`	被删除的元素
	`remove(int index)`	删除**指定位置**元素	`list.remove(0);`	被删除的元素

操作类别	方法名	功能描述	示例代码	返回值
	`remove(Object o)`	删除**指定元素**（第一次出现）	`list.remove("Apple");`	`boolean`
	`clear()`	**清空**所有元素	`list.clear();`	`void`
访问元素	`get(int index)`	获取**指定位置**元素	`String item = list.get(0);`	指定位置的元素
	`getFirst()`	获取**头部**元素	`String first = list.getFirst();`	头部元素
	`getLast()`	获取**尾部**元素	`String last = list.getLast();`	尾部元素
	`peek()`	查看**头部**元素（不删除）	`String head = list.peek();`	头部元素
修改元素	`set(int index, E e)`	**修改**指定位置的元素	`list.set(0, "NewValue");`	被替换的旧元素
查找判断	`contains(Object o)`	判断是否包含指定元素	`list.contains("Apple");`	`boolean`
	`indexOf(Object o)`	查找元素**第一次出现**的位置	`list.indexOf("Apple");`	索引位置（-1表示未找到）

操作类别	方法名	功能描述	示例代码	返回值
	`lastIndexOf(Object o)`	查找元素**最后一次出现**的位置	`list.lastIndexOf("Apple");`	索引位置（-1表示未找到）
状态信息	`size()`	获取链表**元素个数**	`int count = list.size();`	元素数量
	`isEmpty()`	判断链表**是否为空**	`if (list.isEmpty()) { ...`	

`Set` 的实现类

1. `HashSet`**（最常用）**
   - **底层结构**：基于 `HashMap` 实现，实质上是用一个 `HashMap` 的 Key 来存储元素。
   - **特点**：**无序**、查询速度非常快。它是 Set 接口的**标准实现**。
   - **使用情景**：所以，当我们需要一个集合，并且最频繁的操作是"**检查某个元素是否存在**"时，HashSet 是最佳选择。
- **使用方法**：仍然是增删查改，在此不赘述，在此暂时不解释一些特殊用法。

2. `TreeSet`

- **底层结构**：基于 `TreeMap` 实现，使用**红黑树**结构。
- **特点**：元素**可以排序**（默认自然排序，如数字从小到大，字符串按字典序），或者通过自定义 `Comparator` 来排序。
- **使用方法**：仍然增删改查，不赘述，在此暂时不解释一些特殊用法。

`Map`的实现类

1. `HashMap`**（最常用）**

   HashMap 的本质就是一个"键-值对"集合：
   - **键（Key）**：就是"书名"（例如：`"哈利波特"`）。它是唯一的，用来查找。

- **值（Value）**：就是“书本身”（例如：`一本《哈利波特》的实体书`）。它是你要存储的数据。
- **核心优点**：**查找、插入、删除的速度极快！**因为它是直接计算位置，而不是一个个遍历。
- **底层结构**：**数组+链表/红黑树**（JDK 1.8 优化）。
- **特点**：**键值对允许为** `null`，**线程不安全**，但效率最高。是 Map 接口的**标准实现。**
- **使用方法**：仍然增删改查，不赘述，只展示如何遍历：

获取所有键的集合	`keySet()`	`for (String key : map.keySet()) { ... }` **or** `for(String name:keys)`	**返回一个包含所有键的** `Set` **视图，常用于遍历。**
获取所有值的集合	`values()`	`for (Integer value : map.values()) { ... }` **or** `for (String value: values)`	返回一个包含所有值的 `Collection` 视图。
获取所有键值对的集合	`entrySet()`	`for (Map.Entry<String, Integer> entry : map.entrySet()) { ... }`	返回一个 `Set<Map.Entry<K, V>>` 视图。**遍历时效率最高。**

2. `TreeMap`
   - **底层结构**：**红黑树。**
   - **特点**：键（Key）是**有序的**（默认自然排序或自定义排序）。
   - **使用方法**：仍然增删改查，不赘述，在此不展示特殊用法。

Task2.遍历

增强for循环

举一个例子，我们用传统的for循环遍历时稍显冗余，因为要用for循环固定的格式和模板，如下：

```
String[] fruits = {"Apple", "Banana", "Orange"};
for (int i = 0; i < fruits.length; i++) {
    System.out.println(fruits[i]);
}
```

我们运用增强for循环来简化：

```
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

在这里我们直接定义和使用了一个`fruit`变量来输出。

增强for循环格式如下:

```
for(元素类型 局部变量 : 数组或集合对象) {
    // 循环体，使用局部变量
}
```

问题解决：Q2

> **请你用增强for循环遍历list中的元素 依次打印出结果 给出你的代码和运行结果截图**

代码实现如下:

```
package List.demoarray;
import java.util.ArrayList;
import java.util.List;

public class demo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);

        for(Integer number : list){
            System.out.println(number);
        }
    }
}

```

输出结果如图:

```
1
2
3
4

进程已结束，退出代码为 0
```

---

forEach方法

匿名内部类

先来想一个问题：我们每次使用方法时都需要先定义一个**类**，而类的定义又需要一大坨固定的模板和语法。
有的方法在程序中可能只需要使用几次甚至一次，那么为了~~这点醋包饺子~~使用这个方法我们需要大费周章定义一个类，实在是太不划算了！！（深有体会）

所以，匿名内部类出现了。

literally，匿名内部类就是**没有显示名称的内部类，通常用于创建临时的、只需使用一次的内部类**。其好处就是**即用即丢**。可以实现接口、继承类或者扩展抽象类。

- 语法：
  - 创建匿名内部类

```
接口名 对象名 = new 接口名(){
    //匿名内部类的实现
};
```

```
父类名 对象名 = new 接父类名(){
    //匿名内部类的实现
};
```

  - 代码示例：（以接口类为例）

    接口Eat：

```
package Anonymous;

public interface Eat {
    void eat();
    void goschool();
}

```

    Test类：

```
package Anonymous;

public class Test {
    public static void main(String[] args) {
        Eat demo = new Eat(){

            @Override
            public void eat() {
                System.out.println("吃饭");
            }

            @Override
            public void goschool() {
                System.out.println("上学");
            }
```

```
        };
        //直接调用
        demo.eat();//输出“吃饭”
        demo.goschool();//输出“上学”
    }
}

```

---

lambda表达式

因为函数式接口的简化写法严重依赖于lambda表达式~~(好子其实是懒罢)~~，所以我认为有必要先简要了解什么是lambda表达式，为我们后面了解函数式接口，和一步步从匿名内部类的基础写法推导到lambda表达式的写法打下基础。

像匿名内部类的出现一样，首先我们还是来思考，为什么会有lambda表达式?

lambda是啥意思?
（下面这段话来源于ds）

> - **Lambda（λ）** 是希腊字母的第11个字母。
> - 在计算机科学中，它来源于 **λ演算**，这是一个由数学家阿隆佐·邱奇在20世纪30年代提出的一套用于研究函数定义、函数应用和递归的形式系统。在λ演算中，`λ` 符号被用来标识一个**匿名函数**。
> - 因此，编程语言借用了这个符号和概念，用来表示“匿名函数”。

欸原来lambda表达式就是**匿名函数（匿名方法）**的意思，那是不是和我们刚刚了解到的**匿名内部类**有点关系呢?

常规方法的调用一般是这样的:

```
private String doSth(参数列表){
    方法体;
}
```

lambda表达式相当于**简化**了这个过程:

`(参数列表) -> { 表达式体 }`

- `->`: 是 Lambda 操作符，读作 “goes to”（变为）。
- **左侧** `(参数列表)`: 是方法的参数。
- **右侧** `{ 表达式体 }`: 是方法的具体实现。

所以一句话定义: **Lambda 表达式是一个简洁的、可传递的匿名函数。**

特性	解释
目的	用极其简洁的语法来实例化**函数式接口**，实现代码的延迟执行和传递。
核心语法	`(参数) -> { 代码 }`

特性	解释
优点	1. **代码简洁**：大大减少模板代码。 2. **可读性强**：直接表达行为意图。 3. **便于函数式编程**：为 Stream API 等现代 Java 特性打下基础。
使用条件	**必须针对函数式接口**（只有一个抽象方法的接口）。

函数式接口

- **什么是函数式接口**：指的是**只有一个抽象方法**的接口，它是Lambda表达式和方法引用的目标类型。
- **核心特征**：
  - 只有一个抽象方法（可以有多个默认方法或静态方法）
  - 支持Lambda表达式实现
- **基本语法结构（不使用lambda表达式）**

  要求：
  - 必须用 `new 接口名<>() {}`
  - 必须重写 `@Override` 抽象方法
  - **方法名固定**：accept/apply/test/get（下面的四大函数式接口）
  - 代码结构统一，只是接口类型和方法名不同

```
接口类型<泛型> 变量名 = new 接口类型<泛型>() {
    @Override
    public 返回值类型 抽象方法名(参数) {
        // 方法实现
    }
};
```

下面我们来了解四大函数式接口。

1. `Consumer`：消费者（消费型接口）

- **核心方法**：`void accept(T t)`（接受一个参数，无返回值）
- **工作场景**：打印、修改对象内部状态、保存数据等。
- **代码例子**：

```
Consumer<String> consumer = new Consumer<String>() {
    @Override
    public void accept(String t) {
        System.out.println(t);
    }
};
```

2. `Function`：函数（功能型接口）

- **核心方法**：`R apply(T t)`（接受一个T类型参数，返回一个R类型结果）
- **工作场景**：类型转换、数学计算、提取对象信息等。
- **代码例子**：

```
Function<Integer, String> function = new Function<Integer, String>() {
    @Override
    public String apply(Integer t) {
        return String.valueOf(t);
    }
};


```

3. `Predicate`：断言（判断型接口）

- **核心方法**：`boolean test(T t)`（接受一个参数，返回boolean）
- **工作场景**：筛选、条件检查。
- **代码例子**：

```
Predicate<String> predicate = new Predicate<String>() {
    @Override
    public boolean test(String t) {
        return t.length() > 5;
    }
};
```

4. `Supplier`：供应者（供给型接口）

- **核心方法**：`T get()`（无参数，返回一个T类型结果）
- **工作场景**：创建新对象、生成随机数、获取固定配置等。
- **代码例子**：

```
Supplier<Double> supplier = new Supplier<Double>() {
    @Override
    public Double get() {
        return Math.random();
    }
};
```

很好，现在我们已经了解了四大函数式接口，现在我们要做的就是**一步步将其改写为lambda表达式。**

如何改写为lambda表达式

Step 1：完整的匿名内部类
这是最完整的写法，所有部分都在。

Step 2：去掉固定模板
去掉`new Xxx(){}`、方法签名等固定代码。

Step 3：简化参数类型
编译器能推断类型时，可以省略参数类型。

以`Consumer`来推导：

Step 1：完整的匿名内部类

```
Consumer<String> printer = new Consumer<String>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
};
```

Step 2：去掉固定模板

- 去掉 `new Consumer<String>() { @Override public void accept`
- 去掉 `}` 和结尾的 `}`
- 保留：**参数** `(String s)` 和 **方法体** `{ System.out.println(s); }`

变成：

```
Consumer<String> printer = (String s) -> {
    System.out.println(s);
};
```

Step 3：简化参数类型

- 左边已经声明了`Consumer<String>`，编译器知道`s`是String，所以可以去掉参数类型`String`

变成：

```
Consumer<String> printer = s -> System.out.println(s);//因为是单行代码，所以可以把{};也去掉
```

核心思想：Lambda就是只保留【最核心的逻辑】，其他都是模板代码！

---

forEach方法

在 Java 8 引入 Lambda 表达式之前，`forEach` 需要配合 **匿名内部类** 来使用，语法会更复杂一些。

基本语法

```
集合.forEach(new Consumer<元素类型>() {
    @Override
    public void accept(元素类型 变量名) {
        // 对每个元素执行的操作
    }
});
//其实就是在上面匿名内部类的写法上稍稍改造了一下，去掉了new左边的式子，加了一个集合.forEach();在最外面
```

lambda表达式的写法

先来举一个例子：

```
fruits.forEach(new Consumer<String>() {
    @Override
    public void accept(String fruit) {
        System.out.println(fruit);
    }
});
```

还是像刚刚一样层层递推：

1. **类型推断**：既然 `fruits` 是 `List<String>`，那么参数肯定是 `String` 类型，所以不需要写类型
2. **方法名推断**：`Consumer` 接口只有一个方法 `accept`，所以不需要写方法名
3. **new 关键字**：因为只有一个方法，所以不需要创建整个匿名类

可以得到改写后的lambda表达式如下：

```
fruits.forEach((String fruit) -> {
    System.out.println(fruit);
});
```

还可以继续简化：

简化1：去掉参数类型（编译器能推断）

```
fruits.forEach((fruit) -> {
    System.out.println(fruit);
});
```

简化2：只有一个参数时，去掉括号

```
fruits.forEach(fruit -> {
    System.out.println(fruit);
});
```

简化3：方法体只有一行时，去掉大括号和分号

```
fruits.forEach(fruit -> System.out.println(fruit));
```

记忆口诀："三去掉"

1. **去掉** `new Xxx(){}` - 框架
2. **去掉** `@Override public void accept` - 方法声明
3. **去掉参数类型** - 类型推断

问题解决：Q3

> **Q3. 什么是匿名内部类 什么是函数式接口?（感兴趣的同学可以去了解一下四大函数式接口 体会一下OOP和FP的区别）把上面的遍历代码用lambda表达式改写 给出你的代码和运行结果截图 并总结一下lambda表达式的用法**

```
list.forEach(new Consumer<Integer>() {
 @Override
 public void accept(Integer integer) {
     System.out.println(integer);
 }
});
```

理论性知识部分在上面已经学习和解释，下面为代码修改和结果展示。

根据上面从0开始的保姆教程，我们可以用lambda表达式进行如下化简：

```
list.forEach(integer -> System.out.println(integer));//不要搞混了：这个integer是参数名不是泛型
```

直接在**Q2**上的代码cv和注释一下，完整代码如下：

```
package List.demoarray;
import java.util.ArrayList;
import java.util.List;

public class demo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);

/*
        for(Integer number : list){
            System.out.println(number);
        }
*/

        list.forEach(integer -> System.out.println(integer));//不要搞混了：这个integer是参数名不是泛型
    }
}

```

运行结果如下：~~（绝对和Q2的图是两张图啊我重新截的）~~

```
1
2
3
4


进程已结束，退出代码为 0
```

Task3.泛型

在**Java03-数据类型与数据结构**我们很粗浅地了解了什么是**包装类**（包括装箱和拆箱的知识），那么现在来深入了解什么是泛型。

从字面上来理解**泛型(Generics)**：**通用的、泛指的—— 不针对某个特定的事物。**我认为这里从字面意思来入手相当简单且重要，为我们的熟练使用打下了基础。

泛型类

对于非泛型类，我们以前在定义一个变量时，用了`int`、`double`等来定义一个变量所属的类型。
但泛型类既然是通用的、泛指的，**我们在使用时才知道其类型**，所以在**定义时**就需要一个类似于**占位符**的东西来代替——**类型参数声明**就出现了。

常见的类型参数命名约定

虽然可以用任何名字，但通常使用有意义的单个大写字母，以区别于普通的类名：

- `T` - Type（类型）
- `E` - Element（元素），常用于集合中
- `K` - Key（键）
- `V` - Value（值）
- `N` - Number（数字）
- `S`, `U`, `V` - 当需要第二个、第三个类型参数时使用

我们可以把泛型类想象成一个什么都能装的**盒子**，但是一旦使用后就不能改变其装的东西的类型。（详见下面的示例）

如何声明

相较于非泛型类，不过是类名后面加了个`<T>`。

```
class 类名<T> {
    // 使用T作为类型
    private T 变量名;

    public 类名(T 参数) {
        this.变量名 = 参数;
    }

    public T get方法() {
        return 变量名;
    }
}
```

如何使用

相较于创建非泛型类对象，不过是类名后面加上`<具体类型>`和`<>`。

```
// 创建泛型类对象
类名<具体类型> 对象名 = new 类名<>(值);
```

示例如下：

```
public class Main {
    public static void main(String[] args) {
        // 创建装字符串的盒子
        Box<String> stringBox = new Box<>("Hello World");
        String text = stringBox.getContent();  // 直接得到String类型

        // 创建装整数的盒子
        Box<Integer> intBox = new Box<>(100);
        int number = intBox.getContent();  // 直接得到Integer类型

        // 创建装自定义对象的盒子
        Box<Student> studentBox = new Box<>(new Student("张三"));
        Student student = studentBox.getContent();  // 直接得到Student类型
    }
}

class Student {
    private String name;
    public Student(String name) { this.name = name; }
}
```

泛型方法

同上泛型类的概念很类似，泛型方法就是一个**可以适应不同数据类型的通用方法。**

如何定义

```
[访问修饰符] <类型参数列表> 返回类型 方法名(参数列表) {
    // 方法体
}
```

- 当没有返回值时，返回类型就写`void`。
- 当有返回值时，返回类型写`T`占位。

如何使用

定义+使用完整的示例如下：

```
public class GenericMethodDemo {

    // 简单的泛型方法 - 打印任意类型的值
```

```java
    public <T> void printValue(T value) {
        System.out.println("值: " + value);
        System.out.println("类型: " + value.getClass().getSimpleName());
    }

    // 带返回值的泛型方法
    public <T> T getFirstElement(T[] array) {
        if (array == null || array.length == 0) {
            return null;
        }
        return array[0];
    }

    // 测试
    public static void main(String[] args) {
        GenericMethodDemo demo = new GenericMethodDemo();

        // 使用不同类型调用同一个方法
        demo.printValue("Hello");        // String类型
        demo.printValue(100);            // Integer类型
        demo.printValue(3.14);           // Double类型

        // 获取数组第一个元素
        String[] names = {"Alice", "Bob", "Charlie"};
        String first = demo.getFirstElement(names);
        System.out.println("第一个名字: " + first);
    }
}
```

泛型接口

同上，泛型接口在对于不同的类实现接口时，可以**指定不同的数据类型。**

如何定义

```java
interface 接口名<T> {
    T 方法名(T 参数);
}
```

如何使用

有两种情况：

- 实现接口时**指定具体类型**

```java
// 定义泛型接口
interface MyInterface<T> {
    T process(T input);
}

// 实现接口时，指定 T 为 String
class StringProcessor implements MyInterface<String> {
    @Override
    public String process(String input) {
        return input.toUpperCase();
    }
}
```

- 实现接口时，**继续保留泛型**，使用时再决定类型

```java
// 定义泛型接口
interface MyInterface<T> {
    T process(T input);
}

// 实现接口时，保留泛型 T
class GenericProcessor<T> implements MyInterface<T> {
    @Override
    public T process(T input) {
        return input;
    }
}
```

问题解决：Q4

> **Q4. 请你完成下面的模拟项目**
>
> - 创建一个泛型接口 该接口是一个仓库 要求能够储存任何类型的数据 (使用泛型)

```java
public interface Repository
```

> - 这个接口内有两个方法 save( ) 方法用于向仓库中添加数据 getById( ) 方法用来根据id获取数据
> - 创建MyRepository类 实现上面的接口 完成上述两个方法的具体实现 (注意 id是整数类型 从0开始自增 是数据的唯一标识)
> - 提示: 可以使用hashMap

```java
public class MyRepository
```

> 给出下述User类

```java
public class User {
    private String name;
    private int age;

```

```
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{name='" + name + "', age=" + age + "}";
    }
}
```

- 要求分别在你的MyRepository储存String User Integer类型的三组数据 并调用你写的遍历方法 在main函数中打印出仓库中所有元素的内容 给出你的完整代码 以及输出截图

嗯……新手第一次做这样一个规模有一丢丢大的项目会有一点不知所措（不知道该从哪里下手的无力感），究其根源大概是这个Task的知识点比较多和杂，一时间有点消化不良罢（。）没关系让我一步一步慢慢复习分析。

创建泛型接口

诶这个应该是这道题最简单的部分了（）代码如下：

```
package Task03;

public interface Repository<T> {

    public abstract void save(T data);
    public abstract T getById(int id);
}

```

MyRepository类

- 这个接口内有两个方法 save( ) 方法用于向仓库中添加数据 getById( ) 方法用来根据id获取数据
- 创建MyRepository类 实现上面的接口 完成上述两个方法的具体实现 (注意 id是整数类型 从0开始自增 是数据的唯一标识)
- 提示: 可以使用hashMap

好了这个地方信息量就有点大了，我们一步步分析。

-> 首先要实现接口，因为我们在使用时才知道类型，所以在**实现接口时应该继续保留泛型。**
-> 既然需要添加数据和根据id获取数据，同时根据提示需要用到`HashMap`，那就需要new一个`HashMap`出来来实现数据的put和get——前面我们学习了**HashMap本质是一个键-值对集合**，所以在这道题的语境下**“id”是键，“数据data”是值。**
-> id是整数类型，且**从0开始自增**，这个应该就是用来储存具体的数据用的便于查找，要用上自增运算符。
-> 还需要遍历输出，那就使用**增强for循环**来遍历。

好的差不多分析完了，那我们来上手吧。

```
package Task03;
```

```java

import java.util.HashMap;
import java.util.Map;

public class MyRepository<T> implements Repository<T> {

    //我们只能确定id是int（在这里用泛型integer），要储存的数据类型只有在运用时才知道，所以用T占位
    //id从0开始自增计数
    private HashMap<Integer, T> store = new HashMap<>();
    private int id = 0;

    //save用来储存键值对
    @Override
    public void save(T data) {
        //注意此处id要自增！！！
        store.put(id++, data);
    }

    //获取id
    @Override
    public T getById(int id) {
        return store.get(id);
    }

    //用增强for循环遍历并输出
    public void print() {
        for (Map.Entry<Integer, T> entry : store.entrySet()) {
            System.out.println("id=" + entry.getKey() + ", data=" + entry.getValue());
        }
    }

}
```

Test类及输出结果

User已经给出，那我们现在开始写main类来调用这些方法和接口。

> - 要求分别在你的MyRepository储存String User Integer类型的三组数据 并调用你写的遍历方法 在main函数中打印出仓库中所有元素的内容 给出你的完整代码 以及输出截图

Test类代码如下：

```java
package Task03;

import java.io.StringReader;

public class Test {
    public static void main(String[] args) {
        MyRepository<String> stringrepo = new MyRepository<>();
        stringrepo.save("Zack");
        stringrepo.save("ISEKAI");
        stringrepo.print();

```

```
        MyRepository<Integer> integerrepo = new MyRepository<>();
        integerrepo.save(18);
        integerrepo.save(19);
        integerrepo.print();

        MyRepository<User> userrepo = new MyRepository<>();
        userrepo.save(new User("Glimmer",18));
        userrepo.save(new User("Zack",19));
        userrepo.print();
    }
}

```

现在来看一看输出结果：

```
id=0, data=Zack
id=1, data=ISEKAI
id=0, data=18
id=1, data=19
id=0, data=User{name='Glimmer', age=18}
id=1, data=User{name='Zack', age=19}
```

哎写到这儿对新手来说花了很多时间真的很不容易T^T，但是在跌跌撞撞中又强化了知识！！

问题解决：Q5

> 假设你现在是一名负责管理酒吧点唱机的程序员，你的任务是负责将收到的歌曲数据进行排序，其他人会负责将数据封装到一个List中。而你现在有一个模拟类来测试你的代码：

```
public class MockSongs {
 public static List<String> getSongStrings(){
     List<String> songs = new ArrayList<>();
     //模拟将要处理的列表
     songs.add("sunrise");
     songs.add("thanks");
     songs.add("$100");
     songs.add("havana");
     songs.add("114514");
     //TODO
     //在这里完成你的代码
     //END
     return songs;
 }
}
```

> 要求你根据**字符串的长度**进行排序 长度短的排在前面 如果长度相同 则字母排在数字前面 数字排在其他符号前面
>
> 你可以选择下面的任意一种方法调用

```
songs.sort();
Collections.sort();
```

> 重写他们的排序规则即可 使用匿名内部类或lambda表达式均可 这道题是对上面知识点的巩固

好的，阅读题目时就遇到了疑惑：`sort`是啥？sort是排序的意思，可是它到底是什么，有什么用，如何使用?

何为排序（Sort）

`sort`指的是**对数组或集合进行排序的方法**，初阶我们主要分为三种了解：

- **Arrays.sort() - 数组排序**
- **Collections.sort() - 集合排序**
- **自定义排序规则 - 使用`Comparator`比较器**

对于这道题，我希望用`Collection.sort()`解决，同时也需要了解`Comparator`的相关知识，那我们开始吧。

何为`Collection.sort()`

一言以蔽之，`Collection.sort()`是 Java 集合框架提供的**静态工具方法**，用于对实现了`List`接口的集合（如`ArrayList`、`LinkedList`）进行元素排序。（也就是专对`List`进行排序）

`Comparable`接口

在了解`Comparator`接口前，我认为有必要先了解`Comparable`接口，稍后我们将对它们进行一个比较。

`Comparable`是 Java 的排序接口，定义该类对象的**默认排序规则**，其中的`compareTo`是接口唯一的抽象方法，负责具体实现 “两个对象如何比较大小”。

`compareTo`方法

`compareTo`方法就是用来**定义两个对象如何比较大小**的。它只做一件事：**比较当前对象（`this`）和另一个指定对象（参数`o`）。**

它返回一个`int`类型的值，这个值代表比较的结果，规则非常简单，只有三种情况：

返回值	含义	通俗理解
负数	当前对象 **小于** 参数对象	`this` 应该排在 `o` **前面**
零	当前对象 **等于** 参数对象	`this` 和 `o` 一样大，顺序无所谓
正数	当前对象 **大于** 参数对象	`this` 应该排在 `o` **后面**

代码语法如下，有两种写法：

- 一种常见的写法：直接相减

```java
@Override
    public int compareTo(Student otherStudent) {
        // 规则：按身高排序
        // this.height 是当前学生的身高
        // otherStudent.height 是另一个学生的身高

        // 如果当前学生身高 < 另一个学生身高，返回负数（当前学生排前面）
        // 如果当前学生身高 > 另一个学生身高，返回正数（当前学生排后面）
        // 如果相等，返回0

        return this.height - otherStudent.height;

```

- 另一种等价的、更清晰的写法：

```java
       if (this.height < otherStudent.height) {
            return -1;
        } else if (this.height > otherStudent.height) {
            return 1;
        } else {
            return 0;
        }
```

关于升序和逆序

在学习时我遇到了这样的写法：

> 例如：现在要对学生的成绩进行排序，我们分别从升序和逆序两个顺序来排列。
>
> ```java
> // 升序排列：从小到大
> public int compareToAsc(Student other) {
>     return this.score - other.score;
> }
>
> // 降序排列：从大到小
> public int compareToDesc(Student other) {
>     return other.score - this.score;
> }
> }
> ```

从上面的知识可得知，排序是通过两数差的正负来进行的。但在上面的写法中，为什么改变顺序就可以实现升序和逆序的转变？
初见时我确实有这样的疑问，但是换个角度想就很好解释了：

改变减法顺序的本质是改变了比较的视角：

- `this - other`：关注**"我是否比你小"**
- `other - this`：关注**"我是否比你大"**

```
// 升序：想象成问"我比你小吗？"
if (this.score < other.score) return -1;  // 我应该排前面
if (this.score > other.score) return 1;   // 我应该排后面
return 0;

// 降序：想象成问"我比你大吗？"
if (this.score > other.score) return -1;  // 我应该排前面
if (this.score < other.score) return 1;   // 我应该排后面
return 0;
```

这样就很好理解了。理解了本质原理，我们需要用到升序降序时就不再需要写冗杂的if语句，直接使用return即可。

`Comparator`接口

`Comparator<T>` 的主要用途是**定义一个用于比较两个同类型对象的规则。**我们提供一个 `Comparator` 的实现，告诉排序方法（如 `Collections.sort` 或 `Arrays.sort`）或者有序集合（如 `TreeSet`）如何确定一个对象是“小于”、“等于”还是“大于”另一个对象。

`Compare`方法

匿名内部类写法

```
Comparator<类型> 变量名 = new Comparator<类型>() {
    @Override
    public int compare(类型 o1, 类型 o2) {
        // 比较逻辑
        return 结果;
    }
};
```

例如，我们现在需要对字符串进行升序排序：

```
List<String> words = Arrays.asList("apple", "cat", "elephant", "dog");

Comparator<String> byLength = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
};

Collections.sort(words, byLength);//对于words这个List,用我们定义的Bylength比较器进行排序
System.out.println(words); //输出 [cat, dog, apple, elephant]
```

直接内联使用

示例如下：

```java
List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 9);

// 直接在 sort 方法中使用匿名内部类
Collections.sort(numbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer a, Integer b) {
        return b - a;
    }
});

System.out.println(numbers);
```

lambda表达式

现在我们有一个完整的匿名内部类：

```java
Comparator<Integer> comparator = new Comparator<Integer>() {
    @Override
    public int compare(Integer a, Integer b) {
        return a - b;
    }
};
```

根据lambda表达式的核心，我们只需要保留最**核心的部分**即可，在此不作一步一步的推导。结果如下：

```java
Comparator<Integer> comparator = (a, b) -> {
    return a - b;
};
```

因为方法体只有一条语句，所以还可以把大括号和`return`去掉，得到最简形式：

```java
Comparator<Integer> comparator = (a, b) -> a - b;
```

（但就个人而言，大抵是不熟悉，目前阶段我会更习惯匿名内部类的完整写法）

两种接口的比较

所以`Comparable`和`Comparator`到底有什么区别呢？

从字面意思来理解：

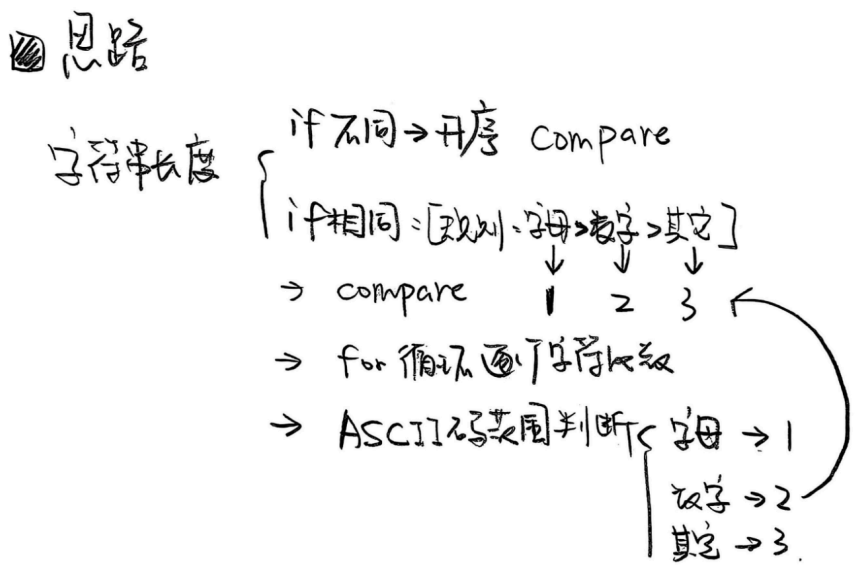
- `Comparable`（可比较的）：这种比较能力是**内在的、与生俱来**的。
- `Comparator`（比较器）：些工具是**外部的、可更换**的。

方面	Comparable	Comparator
方法名	`compareTo(T o)`	`compare(T o1, T o2)`
参数	1个（与当前对象比较）	2个（比较两个对象）

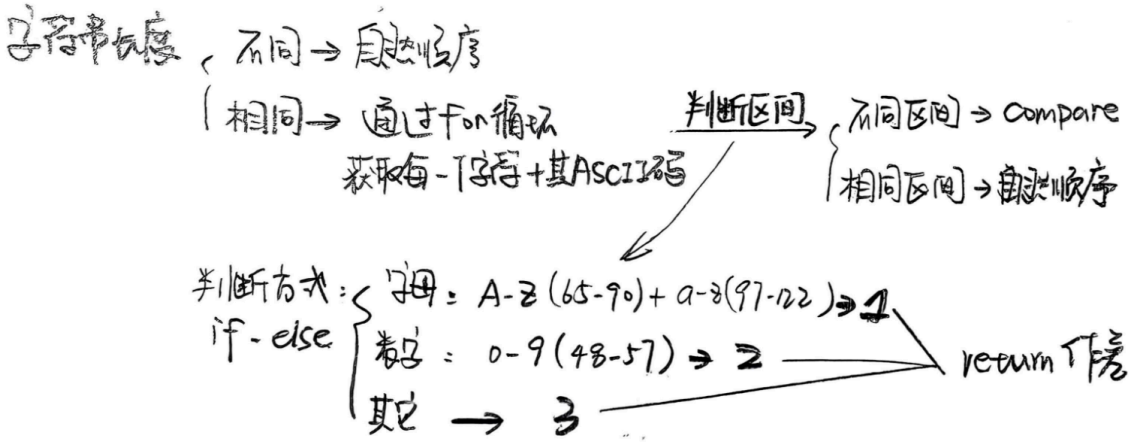
方面	Comparable	Comparator
含义	**自然顺序**、默认排序	**定制排序**、特殊排序
实现位置	在要排序的**类内部**实现	在**外部**单独实现
排序调用	`Collections.sort(list)`	`Collections.sort(list, comparator)`
灵活性	一种排序规则	多种排序规则
修改影响	修改类会影响所有排序	修改比较器只影响特定排序

真的解题过程！！！

好的，学习完这些知识我真的要开始解题了。通过再次阅读题目，我有了以下思路：



这是初步的思路，我们再详细地完善一下就更清晰了：



*手写勿喷

思路大致是清晰的，现在让我们根据mindmap一步步、分块来完成这个项目。

我决定直接**内联使用**，所以最后加一个`Collection.sort();`再稍微补充一下List的名字和new一下`Comparator`就可以了，我们先来写里面的部分。

判断字符串长度

```
            @Override
            public int compare(String s1, String s2) {
                //比较字符串长度
                int len;
                if (s1.length() < s2.length()) {
                    len = -1;
                } else if (s1.length() > s2.length()) {
                    len = 1;
                } else {
                    len = 0;
                }

                if (len != 0){
                    return len;//字符串长度不同时，直接return
                }
```

for循环判断区间

若字符串长度相同，此时我们需要逐个字符比较来判断其ASCII码区间。

```
                for (int i = 0; i < s1.length(); i++) {
                    //用String类里的charAt方法来获取字符
                    char c1 = s1.charAt(i);
                    char c2 = s2.charAt(i);

                    // 如果字符不同，就通过ASCII码区间判断优先级
                    if (c1 != c2) {
                        return comparePriority(c1, c2);
                    }
                }
                //如果字符串完全相同，就无需换序
                return 0;
```

在这里我定义了一个**函数** `comparePriority` 来比较优先级，定义字母 -> 1数字 -> 2，其他 -> 3，这样就可以作差得到顺序了。
但是在作差之前，我们又必须要先确定该字符的ASCII码区间，才能确定其优先级（是1，2还是3），所以**还需要一个函数来判断ASCII码区间。**

判断区间&判断优先级

根据上面的分析，那我们先来写判断ASCII码区间的函数，命名为

```
            // 通过ASCII码范围判断字符类型
            public int getPriority(char c) {
                // 通过强制类型转换获取字符的ASCII码
                int ascii = (int) c;

                // 判断字母：A-Z (65-90) 或 a-z (97-122)
                if ((ascii >= 65 && ascii <= 90) || (ascii >= 97 && ascii <= 122)) {
                    return 1;
                }
                // 判断数字：0-9 (48-57)
                else if (ascii >= 48 && ascii <= 57) {
                    return 2;
                }
                // 判断其他
                else {
                    return 3;
                }
            }
```

所以，现在我们在函数 `comparePriority` 直接嵌套调用 `getPriority` 就可以了，`comparePriority` 这个函数实现作差即可。

```
        public int comparePriority(char c1, char c2) {
                int priority1 = getPriority(c1);
                int priority2 = getPriority(c2);

                if (priority1 != priority2) {
                    return priority1 - priority2;
                }
                // 若是同一类型，按ASCII码自然顺序比较即可
                return c1 - c2;
            }
```

最终结果

好！！！到这里我们把每一个模块都写完了，现在把它们组合起来，再把整个一坨（？）作为一个静态方法方便在Test类中调用。
完整代码如下：
MockSongs类：

```java
package Task05;
import java.util.*;

public class MockSongs {
    public static List<String> getSongStrings() {
        List<String> songs = new ArrayList<>();
        songs.add("sunrise");
        songs.add("thanks");
        songs.add("$100");
        songs.add("havana");
        songs.add("114514");

        Collections.sort(songs, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                //比较字符串长度
                int len;
                if (s1.length() < s2.length()) {
                    len = -1;
                } else if (s1.length() > s2.length()) {
                    len = 1;
                } else {
                    len = 0;
                }

                if (len != 0){
                    return len;
                }

                //当长度相同，for循环取每一个字符进行比较
                for (int i = 0; i < s1.length(); i++) {
                    //用String类里的charAt方法来获取字符
                    char c1 = s1.charAt(i);
                    char c2 = s2.charAt(i);

                    // 如果字符不同，就通过ASCII码区间判断优先级
                    if (c1 != c2) {
                        return comparePriority(c1, c2);
                    }
                }
                //如果字符串完全相同，就无需换序
                return 0;
            }

            // 通过ASCII码区间比较字符优先级
            public int comparePriority(char c1, char c2) {
                int priority1 = getPriority(c1);
                int priority2 = getPriority(c2);

                if (priority1 != priority2) {
                    return priority1 - priority2;
                }
                // 若是同一类型，按ASCII码自然顺序比较即可
                return c1 - c2;
            }

```

```
            // 通过ASCII码范围判断字符类型
            public int getPriority(char c) {
                // 通过强制类型转换获取字符的ASCII码
                int ascii = (int) c;

                // 判断字母：A-Z (65-90) 或 a-z (97-122)
                if ((ascii >= 65 && ascii <= 90) || (ascii >= 97 && ascii <= 122)) {
                    return 1;
                }
                // 判断数字：0-9 (48-57)
                else if (ascii >= 48 && ascii <= 57) {
                    return 2;
                }
                // 判断其他
                else {
                    return 3;
                }
            }
        });

        return songs;
    }
```

Test类：

```
package Task05;
import java.util.*;

public class Test {
    public static void main(String[] args) {
        List<String> songs = MockSongs.getSongStrings();
        System.out.println(songs);
    }
}
```

运行结果如图：

```
[$100, havana, thanks, 114514, sunrise]

进程已结束，退出代码为 0
```

完结撒花！(｀ω´)✧

做完这个Task的一点心得（掏心窝子time）

9.27的晚上我又一次来到了微光的宣讲会，犹记得后端方向的负责人学姐说，在写md文档时想象自己即将把这篇文章发布在csdn或者某些技术平台上，目的不仅是记录自己的学习历程，还需要能让别人（初学者）看懂你的文章。

于是怀着这样的心情我用时不知道几天完成了Task07。知识点很多很杂，尽量做到了记录自己的学习路径和疑问，从中琢磨出自己的学习体会；也把自己想象成一名创作者，写出尽量简单而通俗易懂的“教程”。
比如初见**匿名内部类**会觉得有些不知所云（？），但从“匿名”这个词语入手就很简单了。
比如如何从匿名内部类一步步推导到**lambda表达式**，这样不仅掌握了lambda表达式的用法（虽然本人现在还没有熟练使用），同时知其然而知其所以然。
比如**泛型**的理解，我们从同样从字面意思“泛”入手就迎刃而解了。

在完成Java07后更深刻地认识和体会到了OOP的魅力。短时间内接收这么多知识点会有点消化不良甚至痛苦，但其实如果把这些知识点完全消化后，它们就是一个个供人使用的工具，即拿即用，非常方便。

排行榜发布后既有压力，更是动力。写下这些话以此自勉。最后感谢审核人看到这里，感谢。