Q1:集合

`Collection`

`Collection`作为集合类型的框架接口，定义了集合的基本操作，比如`add()`、`remove()`、`size()`、`iterator()`。

其子接口有`List`和`Set`。

`List`

作为有序，可重复的数据结构，保持了元素的插入顺序，可以通过索引访问。

常见实现有`ArrayList`,`LinkedList`

`ArrayList`

功能：添加，访问，删除，修改元素；计算大小；迭代数组列表

- **特点**： 动态数组，可变大小。
- **优点**： 高效的随机访问和快速尾部插入。
- **缺点**： 中间插入和删除相对较慢。

`LinkedList`

功能：添加，访问，删除，修改元素（快速获取列表开头结尾的元素）；计算大小；迭代数组列表

- **特点**： 双向链表，元素之间通过指针连接。
- **优点**： 插入和删除元素高效，迭代器性能好。
- **缺点**： 随机访问相对较慢。

`Set`

作为无序，不重复的数据结构，不能存放重复的元素

常见实现有`HashSet`， `TreeSet`

`HashSet`

功能：添加，判断，删除元素；计算大小；迭代容器中元素

- **特点**：HashSet 基于 HashMap 来实现的，是一个不允许有重复元素的集合。
- **优点**：高效的查找和插入操作,允许插入null值。
- **缺点**： 不是线程安全的， 如果多个线程尝试同时修改 HashSet，则最终结果是不确定的。 您必须在多线程访问时显式同步对 HashSet 的并发访问。该结构是无序的。

`TreeSet`

功能：添加，判断，删除元素；计算大小；迭代容器中元素；自动排序与查询

- **特点**：TreeSet 是有序集合，底层基于红黑树实现，不允许重复元素。

- **优点：** 提供自动排序功能，适用于需要按顺序存储元素的场景。
- **缺点：** 性能相对较差，不允许插入 null 元素。

`Map`

`Maps`用于存储键值对，常见的实现有 HashMap 和 TreeMap。

`HashMap`

功能：添加，访问，删除元素；计算大小；迭代容器中元素（keys, values, 键值对）；替换和缺省

- **特点：** 基于哈希表实现的键值对存储结构。
- **优点：** 高效的查找、插入和删除操作。
- **缺点：** 无序，不保证顺序。

`TreeMap`

功能：添加，访问，删除元素；计算大小；迭代容器中元素（keys, values, 键值对）；替换和缺省；排序；子Map

- **特点：** 基于红黑树实现的有序键值对存储结构。
- **优点：** 有序，支持按照键的顺序遍历。
- **缺点：** 插入和删除相对较慢。

数组与集合的区别

数组

1. 存储基本类型或对象
2. 长度由初始化时指定，无法进行更改
3. 能够以索引进行逐个访问
4. 功能较单一，只支持存取与遍历

集合

1. 只能存储对象，故存储基本类型时需要用包装类
2. 长度可变，能够动态增删
3. 通过迭代器和key访问
4. 功能多样，能够实现丰富操作（详见上述各种实现类）

Q2:增强for循环遍历list中的元素

代码实现

```
package uestc.glimmerjava7;

import java.util.List;
import java.util.ArrayList;

public final class ForEach {
    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 num : list) {
            System.out.println(num);
        }
    }
}
```

运行结果截图

Q3:匿名内部类和函数式接口

匿名内部类

- 匿名内部类是**没有名字的内部类**：即一个类中包含另外一个类，且不需要提供任何的类名直接实例化。
- 匿名类的实现通常是为了**继承一个父类**或**实现一个接口**。
- 它是一种临时写出来只用一次的类，通常出现在：

想要**快速创建一个接口或抽象类的实例**，但又**不想单独定义一个新类**的时候。

主要用途：

创建一个对象来执行特定的任务，可以使代码更加简洁。

函数式接口

函数式接口是**仅含有一个方法声明**的接口，用于专门表示**一个函数**

函数式接口一般通过匿名内部类或Lambda表达式进行直接实现而不用专门写一个独立的类

思考：为什么会存在匿名内部类和函数式接口?

匿名内部类

1. OOP：java作为一个面向对象程序语言，所有的行为都必须通过类和对象来实现，但是有些类只用一次，为此新建一个文件太麻烦
2. 应对事件驱动编程：直接在调用处定义**“事情发生时应如何应对”**，更加直观

函数式接口

匿名内部类也存在一些问题：

1. 语法冗长：每次都需要写 `new 接口名() { ... }`，还要写 `@Override` 和方法具体实现，使得可读性差
2. 性能低下：每次都会生成一个`.class`文件

故出现了函数式接口(JDK8)

1. Lambda表达式的出现使语法冗长得到了解决
2. 函数的形式取代了每次生成类文件的冗长

代码实现

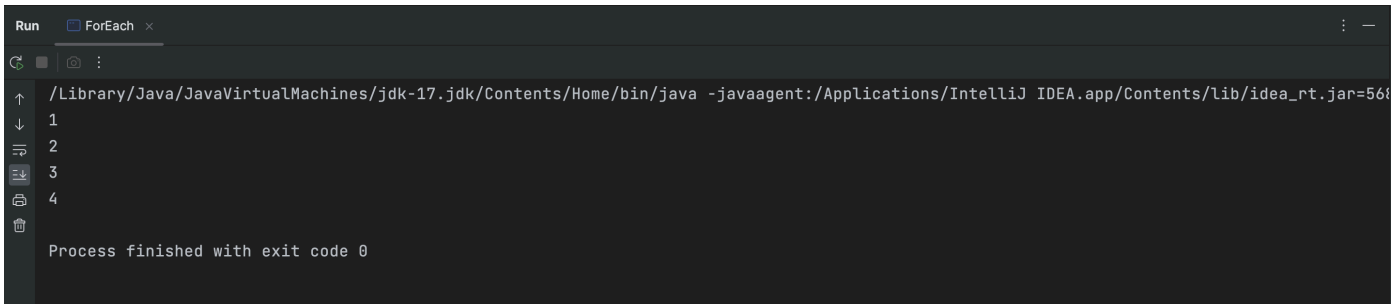
```
package uestc.glimmerjava7;

import java.util.List;
import java.util.ArrayList;

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

        list.forEach(num -> System.out.println(num));
    }
}
```

运行截图



Lambda表达式的用法

语法

```
(参数列表) -> { 代码块 }
```

- `(参数列表)`：传入的参数，可以省略参数类型，交由编译器自动判断
- `->`：Lambda操作符
- `{ 代码块 }`：函数体，可以包含表达式或语句

用法

主要用于四大函数接口的实现：

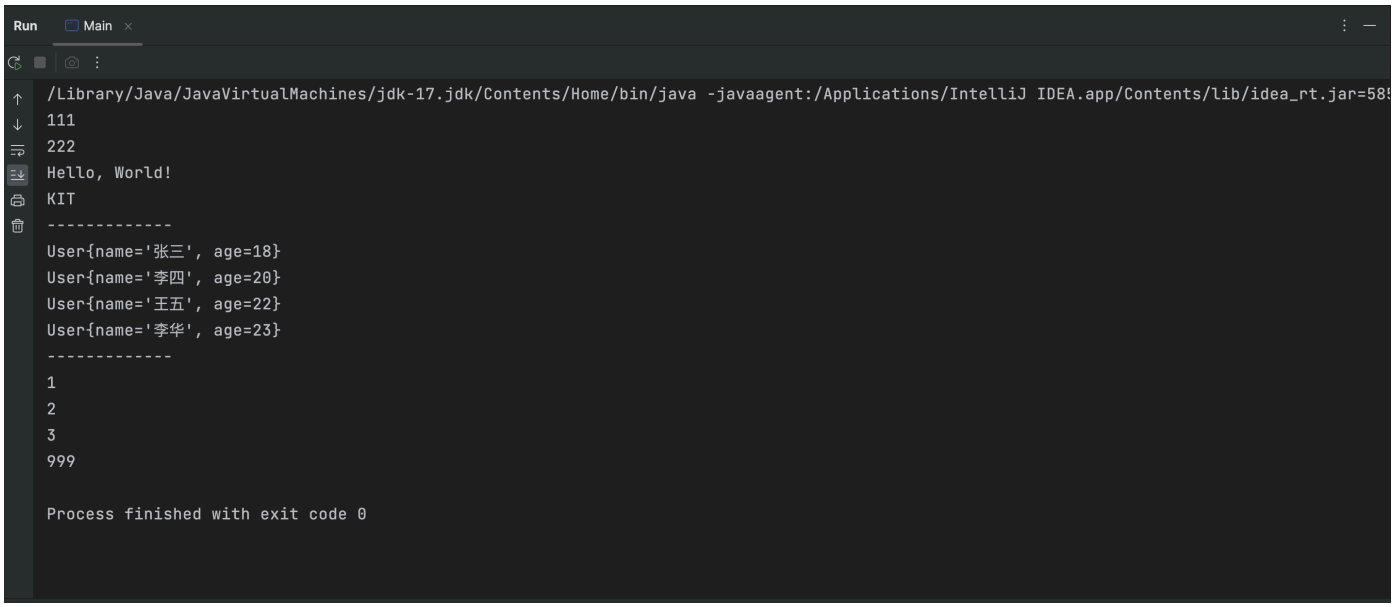
接口名	抽象方法	参数	返回值	作用
Consumer	`void accept(T t)`	有	无	消费一个数据（处理但不返回）
Supplier	`T get()`	无	有	供给数据（生产者）
Function<T,R>	`R apply(T t)`	有	有	将输入映射成输出（函数转换）
Predicate	`boolean test(T t)`	有	`boolean`	判断条件（返回真假）

Q4:模拟项目

代码详见

`package uestc.glimmerjava7.task3;`

输出结果



Q5:模拟案例

```java
package uestc.glimmerjava7.task4;

import java.util.ArrayList;
import java.util.List;

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");

        songs.sort((a, b) -> {
            if (a.length() != b.length()) {
                return Integer.compare(a.length(), b.length());
            }

            for (int i = 0; i < a.length(); i++) {
                char c1 = a.charAt(i);
                char c2 = b.charAt(i);
                int i1 = getCharType(c1);
                int i2 = getCharType(c2);

                if (i1 != i2) {
                    return Integer.compare(i1, i2);
                }

                return Character.compare(c1, c2);
```

```
            }

            return 0;
        });
        return songs;
    }

    public static void main(String[] args) {
        List<String> songs = getSongStrings();
        for (String song: songs) {
            System.out.println(song);
        }
    }

    //用于判断字母类型（属于字母，数字或都不是）
    private static int getCharType(char c) {
        if (Character.isLetter(c)) {
            return 0;
        }
        if (Character.isDigit(c)) {
            return 1;
        }
        return 2;
    }
}
```

输出结果