第5章 内部类和泛型

内部类是定义在另一个类内部的类。内部类可以访问其外部类的所有成员，但外部类不能直接访问内部类的成员。如果外部类要访问内部类的成员，那么必须创建内部类的实例。外部类和内部类的嵌套结构可以增强封装性，使相关的功能被组织在一起。

泛型是 Java 的一种特性，允许在类、接口和方法定义中使用类型参数，从而实现类型的参数化。程序开发者使用泛型，可以保证程序编译时的类型安全，也可以提高代码复用性。

5.1 内 部 类

根据内部类在外部类中出现位置的不同，内部类可以分为以下 4 种。

(1) 成员内部类：定义在外部类的内部，属于外部类的实例。

(2) 静态内部类：使用 static 修饰符定义，属于外部类，而且不依赖实例。

(3) 局部内部类：定义在方法中的类，只在该方法的作用域内可用。

(4) 匿名类(匿名内部类)：没有类名的内部类，一般用于创建一次性对象。

5.1.1 成员内部类

成员内部类就像是外部类的一个成员，可以由 private、public 等访问权限修饰。

【例 5.1】 声明内部类 InnerClass。代码清单 MemberInnerClassDemo.java 如下。

```
class OuterClass {
    private String outerField = "外部类的成员";
    public class InnerClass {
        public void display() {
            System.out.println("访问: " + outerField);
        }
    }
}
public class MemberInnerClassDemo {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();
    }
}
```

在上述代码中，OuterClass 要访问 InnerClass 的成员方法，必须先创建内部类的实例。内部类和外部类虽然存在于同一个 Java 源文件中，但是经过编译之后，内部类会形成单独的字节码文件。

程序运行结果如下。

```
访问：外部类的成员
```


如果需要把某个类的实现细节与外部类紧密绑定，那么可以使用成员内部类。例如，在构建复杂的 GUI 应用程序时，使用内部类表示特定的功能，如按钮的事件监听器。

5.1.2 静态内部类

静态内部类是使用 static 修饰的内部类，与外部类的实例无关。静态内部类可以拥有自己的静态成员，如静态字段、静态方法和静态初始化块。这些静态成员遵循普通类中声明和使用静态成员的规则。

由于被嵌套在外部类中，静态内部类可以直接访问外部类的静态成员，但不能直接访问外部类的非静态成员。如果确实需要访问，那么需要先创建外部类的实例。

【例 5.2】 静态内部类访问外部类成员的示例。代码清单 StaticInnerClassDemo.java 如下。

```
class OuterClass {
    private String instanceField = "Instance Field";      // 非静态成员
    private static String staticField = "Static Field";   // 静态成员
    public static class StaticInnerClass {
        public void display() {
            System.out.println("访问静态成员：" + staticField);
            // 需要先创建外部类的实例才能访问非静态成员
            OuterClass outer = new OuterClass();
            System.out.println("访问非静态成员：" + outer.instanceField);
        }
    }
}
public class StaticInnerClassDemo {
    public static void main(String[] args) {
        OuterClass.StaticInnerClass inner = new OuterClass.StaticInnerClass();
        inner.display();
    }
}
```

程序运行结果如下。

```
访问静态成员：Static Field
访问非静态成员：Instance Field
```

由于静态内部类无法直接访问非静态成员，代码中需要先创建外部类的实例，再访问实例成员。除了这种方法，我们还可以采用在内部类中持有外部类实例引用的方法。这是因为在复杂的数据结构（如链表、树等）或数据库操作中，内部类可能需要频繁地与外部类进行交互，持有外部类实例的引用可以更方便地访问其状态和行为。此外，如果需要访问外部类的方法，那么该方法可能会根据外部类的状态做出不同的反应。这时也需要让内部类持有外部类的实例引用，使内部类能够灵活地调用外部类的方法。

【例 5.3】 执行数据库操作前，需要先使用用户名和密码登录数据库，然后执行对数据库的操作。一种常见的做法是使用静态内部类专门处理用户名和密码，建立连接，执行查询。代码清单 DatabaseConnectionDemo.java 如下。

```
class DatabaseConnection {
    private String username;                                    // 外部类的非静态成员
    private String password;                                    // 外部类的非静态成员
    public DatabaseConnection(String username, String password) {
        this.username = username;
        this.password = password;
    }
    // 静态内部类
    public static class DatabaseQuery {
        private DatabaseConnection dbConnection;                // 持有外部类的实例引用
        // 构造器,接收外部类实例作为参数
        public DatabaseQuery(DatabaseConnection dbConnection) {
            this.dbConnection = dbConnection;                   // 保存外部类实例的引用
        }
        public void executeQuery(String query) {
            // 通过持有外部类实例引用来访问非静态成员
            System.out.println("执行查询: " + query);
            System.out.println("连接数据库的用户: " + dbConnection.username);
            // 对密码进行简单操作,可以根据需要进行加密或处理
            System.out.println("连接数据库的密码: " + dbConnection.password);
        }
    }
}
public class DatabaseConnectionDemo {
    public static void main(String[] args) {
        DatabaseConnection dbConnection = new DatabaseConnection("admin", "123456");
         DatabaseConnection.DatabaseQuery dbQuery = new DatabaseConnection.DatabaseQuery
(dbConnection);
        dbQuery.executeQuery("SELECT * FROM users");    // 执行查询
    }
}
```

例 5.3 中的代码不包含具体数据库操作,仅说明静态内部类的用法。上述代码在静态内部类 DatabaseQuery 中声明外部类的实例 dbConnection,之后使用实例引用成员。

程序运行结果如下。

```
执行查询: SELECT * FROM users
连接数据库的用户: admin
连接数据库的密码: 123456
```

使用静态内部类可以在不依赖外部类实例的情况下组织类的结构。例如,设计工具类或者集合类时需要将一些逻辑或实用功能放在同一类下,但又不希望它们直接与外部类的状态关联。类似这样的应用场景可以考虑设计静态内部类。

5.1.3 局部内部类

局部内部类是在一个方法内定义的类,只能在该方法内部使用。它可以访问方法的局部变量(必须是 final 或者事实上是 final 的变量),也可以访问外部类的成员。

【例 5.4】 使用牛顿迭代法求平方根。代码清单 CalculatorDemo.java 如下。

```
class Calculator {
    private double epsilon;                                    // 表示计算的精度
    public Calculator(double epsilon) {
        this.epsilon = epsilon;                                // 设置精度
    }
    // 方法内定义局部内部类
    public void calculateSquareRoot(double number) {
        // 局部内部类
        class SquareRootCalculator {
            public double compute() {
                // 检查输入
                if (number < 0) {
                    throw new IllegalArgumentException("负数没有平方根.");
                }
                double guess = number / 2.0;                   // 初始猜测值
                double squareRoot;
                // 使用牛顿法(Newton's Method)迭代求解平方根
                do {
                    squareRoot = guess;
                    guess = (squareRoot + number / squareRoot) / 2.0; // 更新猜测值
                } while (Math.abs(squareRoot - guess) > epsilon); // 直到收敛到精度内
                return guess;                                  // 返回最接近的平方根
            }
        }
        // 创建局部内部类实例并计算平方根
        SquareRootCalculator calculator = new SquareRootCalculator();
        double result = calculator.compute();
        System.out.println("数字 " + number + " 的平方根: " + result);
    }
}
public class CalculatorDemo {
    public static void main(String[] args) {
        Calculator calculator = new Calculator(0.000001);      // 创建计算器实例
        calculator.calculateSquareRoot(25);                    // 计算平方根
        calculator.calculateSquareRoot(9);                     // 计算平方根
    }
}
```

程序运行结果如下。

```
数字 25.0 的平方根: 5.0
数字 9.0 的平方根: 3.0
```

局部内部类可用于在方法内部封装特定逻辑,简化外部类的结构,同时实现相关功能的模块化。尤其是仅在特定方法中使用简短逻辑时应该使用局部内部类,如处理特定计算或转换,处理复杂数据结构的操作,隐藏某些实现细节以保持外部类接口的简洁。使用局部内部类,可以保持代码的整洁和良好的可读性。

5.2 匿 名 类

匿名类是没有名称的类,可以直接用来创建对象。一般而言,匿名类在创建对象时立即继承父类或者实现接口,其所创建的对象就是匿名对象。

使用匿名对象通常是作为方法参数传递或者作为返回值。当不需要在后续代码中引用这个对象时,使用匿名对象是合理的。相应地,匿名类通常用于简化代码,在实现接口或继承父类时,直接在实例化时定义匿名类。

5.2.1 匿名类的声明

声明匿名类的基本语法如下。

```
new ParentClassOrInterface() {
    // 类体,包含方法的具体实现
};
```

在上述语法格式中,ParentClassOrInterface 可以是一个类或者一个接口,通常是一个抽象类或普通类。花括号{}中需要实现方法,或者声明其他的变量和方法。

【例 5.5】 继承抽象类的匿名类。代码清单 InheritedAICAbstract.java 如下。

```
abstract class Animal {
    abstract void sound();
}
public class InheritedAICAbstract{
    public static void main(String[] args) {
        Animal dog = new Animal() { // 匿名内部类
            @Override
            void sound() {
                System.out.println("汪汪队立大功!");
            }
        };
        dog.sound();
    }
}
```

在上述代码中,匿名类创建 Animal 的一个实例对象 dog,并实现 sound()方法。

程序运行结果如下。

```
汪汪队立大功!
```

【例 5.6】 实现接口的匿名类。代码清单 InterfaceAIC.java 如下。

```
interface Greeting {
    void greet(String name);
}
public class InterfaceAIC {
    public static void main(String[] args) {
        Greeting greeting = new Greeting() { // 匿名内部类
            @Override
            public void greet(String name) {
                System.out.println("你好," + name + "!");
            }
        };
        greeting.greet("韩丁那");
    }
}
```

上述代码用匿名类实现接口的 greet()方法。

程序运行结果如下。

```
你好，韩丁那！
```

从例 5.5 和例 5.6 可以看到，通常使用匿名类的方法如下。


```
父类/接口 对象 = new 父类/接口( ) {  重写父类/接口中的方法  };
```

这样做相当于把子类继承父类、重写父类中的方法和创建子类对象这 3 个过程合并，从而一次完成；或者相当于把 3 个过程（类实现接口、重写接口中的方法和创建实现类的对象）合并完成。

如果子类和实现类只使用一次，即只定义一个对象，那么建议使用匿名类。这样就不需要单独编写子类和实现类的代码，从而实现简化代码的效果。匿名类除了适用于定义简短的实现类，还在回调、事件监听和需要临时实现接口的场景中有广泛的用途。

5.2.2 方法回调

回调（Callback）通常指的是将一个方法的引用传递到另一个方法中，在某个事件或操作完成时调用这个方法。实现回调的形式非常灵活，常见的有 4 种：使用反射、直接调用、接口调用和 Lambda 表达式。

1. 使用反射实现回调

在 Java 中，通过回调可以给其他对象提供一段信息，这为对象在之后的某个时间点回到调用者中提供可能。Java 的反射机制可以获得当前的信息，因此使用反射实现回调是一种容易想到的做法，但是由于这种方式需要传递的参数比较复杂，而且反射机制在运行时性能开销较大，一般不推荐使用。

2. 直接调用实现回调

直接通过对象来调用方法的具体做法有如下两个步骤。

（1）创建一个类 A 的实例对象 a。

（2）对象 a 作为其他类 B 的方法的参数，在类 B 中直接通过对象 a 来调用类 A 中的方法。这就实现了回调。

3. 使用接口和匿名类实现回调

回调可以是一个接口，而方法的实现一般通过匿名类来完成，这样可以减少代码的冗余，并且提高代码的可读性。通过匿名类实现回调，从代码上看是在不定义新类的情况下提供一次性分离的实现，这是实现回调常见且简洁的方法，特别适用于事件处理和某些异步操作。

【例 5.7】 模拟文件下载的过程，使用回调。代码清单 DownloaderDemo.java 如下。

```
interface DownloadCallback {
    void onDownloadComplete(String message);
}
class Downloader {
    public void download(String fileName, DownloadCallback callback) throws
InterruptedException {
        // 模拟下载过程
        System.out.println("正在下载 " + fileName + "...");
        // 假设下载过程 2s 完成后，调用回调方法
        Thread.sleep(2000);                    // 模拟耗时下载
        // 下载完成，使用回调，方法的具体实现不在此处声明
```

```
            callback.onDownloadComplete(fileName + " 下载完成.");
        }
    }
    public class DownloaderDemo {
        public static void main(String[] args) throws InterruptedException {
            Downloader downloader = new Downloader();
            // 在此处使用匿名类实现 DownloadCallback 接口的具体逻辑
            downloader.download("Java 编程.txt", new DownloadCallback() {
                @Override
                public void onDownloadComplete(String message) {
                    System.out.println(message);   // 输出下载完成的信息
                }
            });
        }
    }
```

例 5.7 中的代码定义了 DownloadCallback 接口，包含一个 onDownloadComplete()方法，用来处理下载完成后要执行的操作。操作的具体实现代码需要在其他实现该接口的类(很多情况下是匿名类)中编写。Downloader 类中的 download()方法模拟下载操作需要耗时 2s，下载完成后调用传入的回调实例 callback 的方法。

程序运行结果如下。

```
正在下载 Java 编程.txt...
Java 编程.txt 下载完成.
```

DownloaderDemo 类中加粗部分的代码通过匿名类实现了 DownloadCallback 接口，并提供了方法的具体实现。例 5.7 先声明对象 downloader，再声明匿名类。在相对复杂的场景中，如果需要多次使用对象，那么建议采用这种分开声明的方式，可以提高代码的可读性。如果在相对简单的场景中，从保持代码的简洁性出发，那么可以将 DownloaderDemo 类的代码进行如下简化。

```
public class DownloaderDemo {
    public static void main(String[] args) throws InterruptedException {
        new Downloader().download("Java 编程.txt", new DownloadCallback() {
            @Override
            public void onDownloadComplete(String message) {
                System.out.println(message); // 输出下载完成的信息
            }
        });
    }
}
```

当下载完成时，匿名类的 onDownloadComplete()方法被调用，通过执行回调来输出相关信息。例 5.7 用于说明匿名类是实现回调的方法之一，代码中的 Thread 是 Java 中用于实现多线程的类，在调用 Thread.sleep()方法时要检查是否有下载执行过程中被中断的 InterruptedException 异常。关于线程和异常的相关知识在后续内容中详细介绍。

5.2.3 Lambda 表达式

Lambda 表达式也称为闭包。在 Java 中，使用 Lambda 表达式可以把方法作为另一个方法的参数传递进其内部。

Java 中，Lambda 表达式的语法格式如下。

```
(parameters) -> expression 或 (parameters) ->{ statements; }
```

其中，parameters 是参数列表；expression 或{statements;}是 Lambda 表达式的主体。

有些情况下，如果 Lambda 表达式中只有一个参数，那么可以省略括号；如果没有参数，那么也需要空括号，如() -> System.out.println("Hello, World!");。

Lambda 表达式可以实现代码简洁且类型安全的回调方式。这种方法不需要额外定义类或接口，也不需要构造实例化对象。

将例 5.7 中 DownloaderDemo 类的代码进行简化，使用 Lambda 表达式实现回调，改写后 DownloaderDemo 的代码清单如下。

```
public class DownloaderDemo {
    public static void main(String[] args) throws InterruptedException {
        // 使用 Lambda 表达式实现 DownloadCallback 接口
        new Downloader().download("Java 编程.txt",
                message -> System.out.println(message) // 输出下载完成的信息
        );
    }
}
```

上述代码中加粗部分使用 Lambda 表达式的分隔符“->”，这表示输入参数与方法体之间的关系。接口和 Lambda 表达式相结合，可以实现回调。从对例 5.7 代码的改写可以看出，Lambda 表达式可以使代码变得更加简洁、紧凑。

使用接口或 Lambda 表达式可以显著减少代码的复杂性。同时，这两种方法提供对类型安全的支持，易于理解和使用。在大多数情况下推荐使用这两种方法实现回调，这样代码会更简洁、更高效。

Lambda 表达式支持函数式编程，可以提高代码的可读性。函数式编程提供了一种新的思维方式，即将计算视为数学函数的求值，强调使用函数和不可变数据结构来构建程序。Lambda 表达式是实现函数式编程的具体手段，可以用简洁的方式定义匿名函数。

【例 5.8】 使用 Lambda 表达式作为方法的参数，运用接口实现加法运算。代码清单 LambdaDemo.java 如下。

```
@FunctionalInterface
interface MyFunction {
    int apply(int a, int b);                              // 单一抽象方法
}
public class LambdaDemo {
    public static void main(String[] args) {
        MyFunction abc = (a, b) -> a + b;                 // 使用 Lambda 表达式实现接口
        int result = abc.apply(5, 3);                     // 调用 apply()方法
        System.out.println("结果: " + result);            // 输出
    }
}
```

程序运行结果如下。

```
结果：8
```

在例 5.8 的代码中，abc 是一个实现 MyFunction 接口的实例，指向 Lambda 表达式的引用。Lambda 表达式“(a,b)-> a+b”实现接口中 apply()方法的运算逻辑，向方法传递了两个参数，并相加。

Lambda 表达式不仅使代码更简洁，而且可以直接引用现有类或对象的方法。很多场景下使用 Lambda 表达式有助于提高可读性与可维护性，如集合操作、事件处理、线程执行等。需要注意的是，在 Java 中，Lambda 表达式只能引用被标记为 final 或有效 final 的外层局部变量。这是因为变量被关键字 final 修饰后，在它们的作用域中就不会被重新赋值，这样就可以确保线程安全和避免潜在的并发修改问题。

在函数式编程中，函数是一个独立的代码块，它能够接收输入参数，执行某种操作，并返回输出结果。函数可以在程序的任何地方被调用，可以是全局的或局部的，可以像其他数据类型(如整数、字符串等)一样被传递和操作。高阶函数就是指将函数作为参数传递。高阶函数使函数的组合和复用变得更加灵活。Lambda 表达式支持高阶函数，也就是说我们可以将一个函数或 Lambda 表达式作为参数，传递给另一个函数，从而实现更灵活的代码结构。

5.2.4 函数式接口

并非所有的接口都可以改写成 Lambda 表达式的形式。当且仅当一个接口中只有一个需要实现的方法，才可以使用 Lambda 表达式简化接口的实现类。这样的接口称为函数式接口。函数式接口可以被隐式地转换为 Lambda 表达式或方法引用，从而使代码更加简洁和易于理解。

在掌握函数式接口和 Lambda 表达式的基础上，我们可以通过 Lambda 表达式和匿名类相结合的方式实现回调。

【例 5.9】 匿名类和 Lambda 表达式在 GUI 编程中处理事件窗口的单击按钮事件。代码清单 JFrameDemo.java 如下。

```
import javax.swing.JButton;
import javax.swing.JFrame;
public class JFrameDemo {
    public static void main(String[] args) {
        // 匿名对象:直接创建并使用
        new JFrame("单击按钮") {{
            setSize(300, 200);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            add(new JButton("按钮") {{
                addActionListener(e -> System.out.println("按钮被单击!"));
            }});
            setVisible(true);
        }}.setVisible(true);
    }
}
```

在例 5.9 的代码中，使用匿名类(加灰色底纹)创建 JButton 的匿名实例对象(加虚线下画线)。事件监听器是 Java 中处理事件的机制，它可以使对象在某个事件发生时监听或通知其他对象。例 5.9 使用 addActionListener()方法为按钮添加了一个事件监听器。当按钮单击事件发生时，需要编写代码处理事件，具体来说这里的处理方式是在控制台输出文字。

由于 ActionListener 是一个函数式接口，它只有一个抽象方法 actionPerformed(ActionEvent e)。当使用 Lambda 表达式时，Java 编译器会自动将其转换为对这个方法的实现。具体而言，例 5.9 中的代码使用 Lambda 表达式 e -> System.out.println("按钮被单击!")来处理事件。代码中的 e 是一个事件对象，它是 ActionEvent 类的实例。这个对象包含关于事件的详细信息，如事件的源(触发事件的组件)和事件发生的时间等。Lambda 表达式实现 ActionListener 接口的 actionPerformed()方法，并将事件对象 e 作为参数传递。当按钮被单击时，方法的参数 e 捕获到事件，触发 Lambda 表达式，在控制台输出“按钮被单击!”。每次单击时，代码都将捕获事件并处理。

程序运行结果显示如图 5.1 所示的窗口。单击一次，控制台就会输出如下内容。

```
按钮被单击!
```

图 5.1　单击按钮窗口

5.2.5　函数式方法引用

函数式方法引用是指对现有方法的直接引用，省略参数和方法体的定义。其语法格式如下。

```
函数式接口名　变量名 = 类名或对象名 :: 方法名;
```

在上述语法格式中，::是函数式方法引用。例如，下面的代码使用 Lambda 表达式，表示对象 objects 的 forEach()方法接收一个参数 name，并调用 println()方法。

```
objects.forEach(name -> System.out.println(name));
```

上述代码可以改写成方法引用的形式，具体如下。

```
objects.forEach(System.out::println);
```

上述代码表示直接引用 System.out.println()方法，省略了参数的定义和方法体。

函数式方法引用可以理解为 Lambda 表达式的一种简化形式，适用于可以直接引用的方法。如果编程过程需要调用现有方法而不需要额外的逻辑时，那么使用方法引用可以使代码更具可读性，传达更明确的意图。

5.3 泛　　型

5.3.1 泛型的概念

在 Java 中，泛型(Generic Type 或者 Generics)是一种强类型的编程语言特性，允许在类、接口和方法中使用类型参数。泛型的本质是将类型参数化，也就是说所操作的数据类型被指定为一个参数。

泛型类型参数，也叫泛型标记符，表示泛型可以操作的类型。类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。常用的类型参数如下。

(1) E(Element)：在集合中使用，表示集合中存放的是元素。

(2) T(Type)：表示 Java 类。

(3) K(Key)：表示键。

(4) V(Value)：表示值。

(5) N(Number)：表示数值类型。

(6) ?：表示不确定的 Java 类型。

泛型可以使代码更加灵活和可重用，还可以在编译时检测类型安全，避免运行时错误。需要注意的是，实现重载方法时，不允许方法之间的区别仅为使用不同的泛型。Java 编译器会擦除掉关于泛型类型的信息，这是因为在字节码层面是没有泛型概念的。这种擦除机制会使 Java 代码兼容之前没有使用泛型的类库和代码。

5.3.2 泛型类

泛型类(Generic Class)是指封装不限定于特定数据类型的操作的类，可用于创建集合类。声明泛型类的语法格式如下。

```
修饰符 class 类名<类型参数>{
    // 类的成员
}
```

其中，类型参数可以写为任意标识，常见的如 T、E、K、V 等。

泛型类不能扩展 java.lang.Throwable 类。泛型的参数类型可以使用 extends 语句，如 < T extends superclass >。extends 关键字用于限制类型参数的范围，指定允许传递给类型参数的类型必须继承自某个类或其子类，又或某个接口的实现类。例如，一个操作数字的方法可能只希望接收 Number 或者 Number 子类的实例。

类型参数是有界的，用 extends 关键字设置上界，用 super 关键字设置下界。具体语法格式如下。

(1) ? extends T：表示参数类型上界必须是 T 或其子类型。例如，Box < T extends Number >表示 Box 类只能接收 Number 及其子类。

(2) ? super T：表示参数类型下界必须是 T 或其父类型。例如，List <? super Integer >表示接收一个 Integer 类或其超类的列表。

【例 5.10】 声明一个泛型类 Box。代码清单 Box.java 如下。

```
public class Box<T> {
    private T item;
    public void setItem(T item) {
        this.item = item;
    }
    public T getItem() {
        return item;
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.setItem(123);
        Integer item = integerBox.getItem();
        System.out.println(item.toString()); // 输出 123
    }
}
```

例 5.10 中的代码定义了泛型类 Box 并创建了对象的集合，使用< Integer >表明集合中的元素是 Integer 类型的对象。

程序运行结果如下。

```
123
```

泛型类型参数是与类的实例相关的，不能在静态成员和静态方法中引用泛型类型参数。这是因为使用 static 关键字声明的静态成员和静态方法都属于类本身，而不是类的实例。这意味着它们在类加载时会被创建，并且在所有实例之间共享。泛型类型参数是与类的实例相关的，其具体类型在实例化时确定，因此在静态上下文中无法引用它们。

5.3.3 泛型方法

泛型方法体的声明和其他方法一样。需要注意的是，类型参数只能是引用数据类型，不能是原始的基本数据类型，如 int、double、char 等。定义泛型方法的语法如下。

```
修饰符<类型> 返回值类型方法名(类型 变量名){
    // 方法体
}
```

假如一个方法能够独立于所属类的类型参数，那么就应当考虑将其定义泛型方法。

【例 5.11】 定义泛型方法。代码清单 GenericMethod.java 如下。

```
public class GenericMethod {
    public <T> void display(T item) {
        System.out.println(item);
    }
    GenericMethod gm = new GenericMethod();
    gm.display("Hello!");
    gm.display(123);
}
```

程序运行结果如下。

```
Hello!
123
```

5.3.4 静态泛型方法

静态泛型方法是指在类中定义的静态方法，该方法使用泛型类型参数。静态泛型方法

可以在不依赖类的实例的情况下，处理不同类型的数据。

虽然不能在静态成员和静态方法中引用泛型类型参数，但是如果确实需要在静态上下文中使用泛型，那么可以通过将类型参数作为方法参数来实现。静态上下文中使用泛型的常见应用场景是声明通用的工具类，用于处理不同类型的数据。

【例 5.12】 静态方法中使用泛型类型参数的示例。将类型参数作为方法参数，创建静态方法来比较两个对象并返回较大的一个。代码清单 GenericsInStaticMethodsDemo.java 如下。

```
class GenericUtils {
    // 静态方法
    public static <T extends Comparable<T>> T max(T a, T b) {
        return a.compareTo(b) > 0 ? a : b;                    // 返回较大的对象
    }
}
// 自定义类实现 Comparable 接口
class Person implements Comparable<Person> {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);          // 按年龄比较
    }
}
public class GenericsInStaticMethodsDemo {
    public static void main(String[] args) {
        // 比较整数
        Integer int1 = 10;
        Integer int2 = 20;
        Integer maxInt = GenericUtils.max(int1, int2);
        System.out.println("较大的整数是: " + maxInt);   // 输出: 20
        // 比较字符串
        String str1 = "apple";
        String str2 = "banana";
        String maxStr = GenericUtils.max(str1, str2);
        System.out.println("较大的字符串是: " + maxStr); // 输出: banana
        // 比较自定义对象
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);
        Person olderPerson = GenericUtils.max(person1, person2);
        System.out.println("年龄较大的人员是: " + olderPerson.getName()); // 输出: Alice
    }
}
```

在例 5.12 的代码中，< T extends Comparable < T >>声明了一个泛型类型参数 T，要求 T 必须实现 Comparable 接口。这确保了传递给该方法的对象可以进行比较。T max(T a, T b)方法接收两个类型为 T 的参数，并返回一个类型为 T 的值。

程序运行结果如下。

```
较大的整数是: 20
较大的字符串是: banana
年龄较大的人员是: Alice
```

通过例 5.12 可以发现，使用静态泛型方法将类型参数作为方法的参数传递，使该方法能够处理多种类型的数据，从而增强代码的复用性和灵活性。

5.3.5 泛型接口

泛型接口是 Java 泛型编程的基础，在定义接口时指定类型参数。通过创建不同的实现类，每个实现类使用不同的具体类型，从而实现同一接口的多态性和重用性。

使用泛型类来实现泛型接口的作用是让用户来定义接口所使用的变量类型，而不是像方法那样，把变量类型在类中固定。使用泛型接口便于创建可重用的组件和类，能够提供更强的类型安全和灵活性，并增强代码的复用性。

【例 5.13】 创建有序对，使用泛型接口的示例。代码清单 GenericInteface.java 如下。

```
interface Pair < K, V > {
    public K getKey();
    public V getValue();
}
class OrderedPair < K, V > implements Pair < K, V > {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
}
public class GenericInteface {
    public static void main(String[] args) {
        OrderedPair < Integer, String > pair = new OrderedPair <>(1, "Apple");
        Integer key = pair.getKey();
        String value = pair.getValue();
        System.out.println("键: " + key);
        System.out.println("值: " + value);
    }
}
```

例 5.13 的代码中声明 Pair 是一个泛型接口，接收两个类型参数 K 和 V，它们分别表示键和值的类型。接口定义了 getKey()和 getValue()两个方法：getKey()方法返回类型为 K 的键，getValue()方法返回类型为 V 的值。OrderedPair 类实现了 Pair 接口，通过相同的

类型参数 K 和 V 来实现接口的两个方法，从而返回存储的键和值。GenericInteface 类中的 main()方法创建了 OrderedPair 类的实例，指定 K 为 Integer，V 为 String，并传入键值对(1, "Apple")。OrderedPair 类使用泛型接口来创建一个简单的有序对，能够灵活地处理不同类型的键和值。

程序运行结果如下。

```
键: 1
值: Apple
```

泛型接口可以接收一个或多个类型参数，也可以包含多个方法。这些类型参数可以在接口的方法中使用，从而使接口能够处理不同类型的数据。

5.3.6 泛型函数式接口

泛型函数式接口是一个特殊类型的接口，它只包含一个抽象方法，并且可以接收一个或多个类型参数。由于只包含一个抽象方法，泛型函数式接口可以作为 Lambda 表达式的目标类型。

在 Java 中，java.util.function 包的 BiFunction 接口是一个泛型函数式接口。它表示一个接收两个参数并返回一个结果的函数，其中最后一个参数是返回结果。

【例 5.14】 使用 BiFunction 接口和 Lambda 表达式的示例。代码清单 HigherOrderFunctionDemo.java 如下。

```
import java.util.function.BiFunction;
public class HigherOrderFunctionDemo {
    // 高阶函数，接收一个函数和两个整数
    public static int operate(BiFunction<Integer, Integer, Integer> func, int a, int b) {
        return func.apply(a, b);
    }
    public static void main(String[] args) {
        int sum = operate((x, y) -> x + y, 5, 3);
        int product = operate((x, y) -> x * y, 5, 3);
        System.out.println("和: " + sum);
        System.out.println("积: " + product);
    }
}
```

程序运行结果如下。

```
和: 8
积: 15
```

此外，BiFunction 接口可以与其他函数式接口(如 Function、Consumer 等)结合使用，以实现更复杂的操作。常见泛型函数式接口的介绍见表 5.1。综合使用 Lambda 表达式和高阶函数，可以编写出更简洁、可重用的代码。

表 5.1 常用泛型函数式接口的介绍

泛型函数式接口	说　明	方　法	用　途
BiFunction<T,U,R>	接收两个参数并返回一个结果的函数式接口	R apply(T t,U u)：接收两个参数 t 和 u，返回一个结果	适用于需要处理两个输入并生成输出的场景，如数学运算、字符串操作等

续表

泛型函数式接口	说　明	方　法	用　途
Function＜T,R＞	接收一个参数并返回一个结果的函数式接口	R apply(T t)：接收一个参数 t，返回一个结果	适用于需要将输入转换为输出的场景，如数据转换、映射等
Consumer＜T＞	接收一个参数并不返回结果的函数式接口	void accept(T t)：接收一个参数 t，执行操作但不返回结果。 default Consumer＜T＞ and Then (Consumer ＜? super T＞ after)：返回一个组合的 Consumer，先执行当前的 Consumer，然后执行 after	适用于需要对输入执行某些操作但不需要返回结果的场景，如打印、修改状态等

5.4　常用集合类

5.4.1　List 集合

List 是 Java Collections Framework 中的一个接口，表示一个有序的集合，集合中允许包含重复的元素。List 接口的常用实现类包括 ArrayList、LinkedList 和 Vector 等，对这些类的常用方法和用途的说明见表 5.2。

表 5.2　List 接口的常用实现类

类的说明	常 用 方 法	用　途
ArrayList	add(E e)：添加元素。 get(int index)：获取指定索引的元素。 remove(int index)：删除指定索引的元素。 size()：返回列表的大小。 contains(Object o)：检查列表是否包含指定元素	适合频繁读取元素的场景，但插入和删除操作相对较慢
LinkedList	add(E e)：添加元素。 get(int index)：获取指定索引的元素。 remove(int index)：删除指定索引的元素。 addFirst(E e)：在列表开头添加元素。 addLast(E e)：在列表末尾添加元素。 peekFirst()：获取第一个元素但不删除。 peekLast()：获取最后一个元素但不删除	适合频繁插入和删除元素的场景，但随机访问性能较差
Vector	add(E e)：添加元素。 get(int index)：获取指定索引的元素。 remove(int index)：删除指定索引的元素。 size()：返回列表的大小。 contains(Object o)：检查列表是否包含指定元素。 addElement(E obj)：添加元素(与 add 方法相同)。 removeElement(Object obj)：删除指定元素	适合多线程环境

List 接口提供一种灵活的方式来存储和管理有序元素的集合，适用于多种场景，如存储对象、实现待办事项列表、排序和搜索等。实际应用中需要根据具体需求选择合适的实现类

来优化程序代码性能。

【例 5.15】 使用 List 接口，处理待办事项。代码清单 TodoListDemo.java 如下。

```
import java.util.LinkedList;
import java.util.List;
public class TodoListDemo {
    public static void main(String[] args) {
        List<String> todoList = new LinkedList<>();
        todoList.add("晨读打卡");
        todoList.add("跑步 40 分钟");
        todoList.add("认真吃早餐");
        System.out.println("待办事项列表:");
        for (String task : todoList) {
            System.out.println("- " + task);
        }
        todoList.remove("跑步 40 分钟");
        System.out.println("更新后的待办事项列表:");
        for (String task : todoList) {
            System.out.println("- " + task);
        }
    }
}
```

例 5.15 中的代码使用了泛型集合类 LinkedList 的对象 todoList。程序运行结果如下。

```
待办事项列表:
- 晨读打卡
- 跑步 40 分钟
- 认真吃早餐
更新后的待办事项列表:
- 晨读打卡
- 认真吃早餐
```

在编程中，使用接口类型而不是具体实现类的做法有以下优点。

(1) 增加代码灵活性。使用 List 接口定义变量，有利于将来更容易地更改实现类。例如，最初使用 ArrayList，但后来发现 LinkedList 更适合需求，这时只需更改实例化的部分，而不需要修改其他使用该变量的代码。

(2) 提高代码的可读性和可理解性。程序开发者看到 List 时，会立即知道这是一个列表，而不必关心具体的实现细节。

(3) 遵循面向接口编程的原则。面向接口编程是一个良好的编程实践，使用接口而不是具体实现类的做法使代码更具可扩展性和可维护性。

(4) 便于测试。在单元测试中，使用接口可以更容易地替换实现类。例如，使用模拟对象来替代实际的 ArrayList，而不是依赖具体的实现类。

(5) 减少耦合。依赖接口而不是具体实现类，可以减少代码之间的耦合。这样一来，更改一个类的实现不会影响到依赖该接口的其他类，更容易进行代码修改和重构。

【例 5.16】 使用 ArrayList 存储图书信息。代码清单 SimpleBookManager.java 如下。

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
```

```
class Book {
    private String title;
    private String author;
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
    @Override
    public String toString() {
        return "书名: " + title + ", 作者: " + author;
    }
}
class BookManager {
    private final List<Book> books;
    public BookManager() {
        this.books = new ArrayList<>();
    }
    public void addBook(String title, String author) {
        books.add(new Book(title, author));
        System.out.println("已添加图书: " + title);
    }
    public void printBooks() {
        System.out.println("当前库中图书:");
        for (Book book : books) {
            System.out.println(book);
        }
    }
}
public class SimpleBookManager {
    public static void main(String[] args) {
        BookManager bookManager = new BookManager();
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("输入图书名称 (输入 'exit' 结束): ");
            String title = scanner.nextLine();
            if (title.equalsIgnoreCase("exit")) {
                break;
            }
            System.out.println("输入图书作者: ");
            String author = scanner.nextLine();
            bookManager.addBook(title, author);
        }
        bookManager.printBooks();
        scanner.close();
    }
}
```

在例 5.16 的代码中,BookManager 是图书管理类,使用 ArrayList 来存储图书。它有 addBook()和 printBook()两个方法: addBook()方法用于添加图书; printBooks()方法用于打印当前的图书列表。

程序运行时,用户输入图书信息及打印库中已有图书的情况如图 5.2 所示。

```
输入图书名称（输入 'exit' 结束）：
生命册
输入图书作者：
李佩甫
已添加图书：生命册
输入图书名称（输入 'exit' 结束）：
宝水
输入图书作者：
乔叶
已添加图书：宝水
输入图书名称（输入 'exit' 结束）：
exit
当前库中图书：
书名：生命册，作者：李佩甫
书名：宝水，作者：乔叶
```

图 5.2　代码 SimpleBookManager 的运行情况

5.4.2　Set 集合

Set 是一个不允许有重复元素的集合。它有多个实现类，每个实现类都有其特定的性质和用途。因为 Set 是一个抽象的接口，所以编程中不能直接实例化。

任何试图添加到 Set 集合中的重复元素都会被自动忽略，因此 Set 接口的实现类广泛应用于去除重复元素和数据存储。Set 接口的常见实现类有 HashSet、LinkedHashSet、TreeSet、EnumSet、CopyOnWriteArraySet 等，这些类的常用方法和用途见表 5.3。

表 5.3　Set 接口的常用实现类

类的说明	常 用 方 法	用　　途
HashSet	add(E e)：添加元素。 remove(Object o)：移除指定元素。 contains(Object o)：检查集合是否包含指定元素。 size()：返回集合中元素的数量。 clear()：清空集合中所有元素。 iterator()：返回集合的迭代器	基于哈希表实现，提供常数时间的性能用于基本操作（如添加、删除和包含）。 当不需要保持元素的顺序时，使用 HashSet 是一个很好的选择
LinkedHashSet	与 HashSet 相同，增加了保持插入顺序的特性	继承自 HashSet，同时维护一个双向链表，以保持元素的插入顺序；提供常数时间的性能用于基本操作。 当需要保持元素的插入顺序时，使用 LinkedHashSet
TreeSet	add(E e)：添加元素。 remove(Object o)：移除指定元素。 contains(Object o)：检查集合是否包含指定元素。 first()：返回集合中的第一个元素（最小元素）。 last()：返回集合中的最后一个元素（最大元素）。 ceiling(E e)：返回大于或等于指定元素的最小元素。 floor(E e)：返回小于或等于指定元素的最大元素。 size()，clear()，iterator()：与 HashSet 相同	基于红黑树实现，提供有序的集合。元素按自然顺序排序，或者根据构造时提供的比较器进行排序。 当需要有序集合并且需要进行范围查询时，使用 TreeSet

续表

类的说明	常 用 方 法	用　　途
EnumSet	add(E e)：添加元素。 remove(Object o)：移除指定元素。 contains(Object o)：检查集合是否包含指定元素。 size()，clear()，iterator()：与 HashSet 相同	专门用于存储枚举类型的集合，具有高效的性能，内部使用位图实现。当需要存储枚举类型的集合时，使用 EnumSet
CopyOnWrite ArraySet	add(E e)：添加元素。 remove(Object o)：移除指定元素。 contains(Object o)：检查集合是否包含指定元素。 size()，clear()，iterator()：与 HashSet 相同	每次修改时都会复制底层数组，因此读取操作不会被阻塞。线程安全的 Set 实现。 在多线程编程中，读操作远多于写操作时使用

HashSet 作为 Set 接口的主要实现类，可以存储 null 值。我们根据 HashSet 中元素的哈希值，通过算法可以计算出该元素在底层实现数组中的存放位置。对于频繁的元素遍历操作，LinkedHashSet 的效率高于 HashSet 的效率。TreeSet 要求添加的元素是同一数据类型，支持按照添加元素的指定属性进行排序。程序设计中根据具体需求选择合适的 Set 实现类，可以有效提高程序的性能和可维护性。

例如，为了确保每个用户的电子邮件地址是唯一的，我们可以使用 HashSet 来存储已注册的电子邮件地址，以便于快速检查新注册的电子邮件是否已经存在。

【例 5.17】 模拟电子邮件注册检查的示例。代码清单 EmailRegistration.java 如下。

```
import java.util.HashSet;
import java.util.Scanner;
public class EmailRegistration {
    private HashSet<String> registeredEmails;
    public EmailRegistration() {
        registeredEmails = new HashSet<>();
    }
    public boolean registerEmail(String email) {
        return registeredEmails.add(email);   // 添加电子邮件地址，若存在则返回 false
    }
    public static void main(String[] args) {
        EmailRegistration emailRegistration = new EmailRegistration();
        Scanner scanner = new Scanner(System.in);
        String email;
        while (true) {
            System.out.print("请输入要注册的电子邮件地址(输入 exit 退出):");
            email = scanner.nextLine();
            if (email.equalsIgnoreCase("exit")) {
                break;
            }
            if (emailRegistration.registerEmail(email)) {
                System.out.println("电子邮件地址注册成功: " + email);
            } else {
                System.out.println( email + "电子邮件地址已被注册: " );
            }
        }
        scanner.close();
    }
}
```

在例 5.17 的代码中，registeredEmails 是一个 HashSet，用于存储已注册的电子邮件地址。使用 add()方法尝试添加新电子邮件地址，若电子邮件已存在则返回 false，否则返回 true。

程序运行时，输入电子邮件地址及已有电子邮件地址的情况如图 5.3 所示。

图 5.3　代码 EmailRegistration 的运行情况

从例 5.17 可以看出，HashSet 的 API 简单易用，适合快速开发和实现。HashSet 可以确保每个元素在集合中是唯一的，还可以提供常数时间复杂度的添加和查找操作，这使集合中元素的管理非常高效。

5.4.3　Map 接口

Map 接口用于存储键值对，并提供一种基于键进行查找和操作的数据结构。键和值之间保持一一对应的关系，也称为映射。

Map 接口有多个实现类，每个实现类都有其特定的性质和适用场景。Map 接口的常见实现类有 HashMap、LinkedHashMap、TreeMap、EnumMap、WeakHashMap 等，这些类常用的方法和用途见表 5.4。

表 5.4　Map 接口的常用实现类

类的说明	常用方法	用　途
HashMap	put(K key, V value)：将指定的值与指定的键关联。 get(Object key)：返回指定键所映射的值。 remove(Object key)：移除指定键的映射关系。 containsKey(Object key)：检查是否包含指定的键。 containsValue(Object value)：检查是否包含指定的值。 size()：返回映射中键值对的数量。 clear()：清空映射中的所有键值对	适合大多数需要键值对存储的场景，能够快速查找，但不保证顺序
LinkedHashMap	与 HashMap 相同，增加了保持插入顺序的特性	适合需要顺序遍历的场景
TreeMap	put(K key, V value)：将指定的值与指定的键关联。 get(Object key)：返回指定键所映射的值。 firstKey()：返回映射中的第一个键(最小键)。 lastKey()：返回映射中的最后一个键(最大键)。 ceilingKey(K key)：返回大于或等于指定键的最小键。 floorKey(K key)：返回小于或等于指定键的最大键	有序的键值对，适合需要排序的场景
EnumMap	与 HashMap 相同	高效存储枚举类型的键
WeakHashMap	使用弱引用存储键	允许键被垃圾回收，适合实现缓存和避免内存泄漏

在实际开发中，HashMap 是最常用的 Map 实现类。它提供了快速的查找、添加和删除操作，适合大多数需要键值对存储的场景。

【例 5.18】 模拟学生成绩管理系统。为了能够快速查找每个学生的成绩，我们可以使用 HashMap 来存储学生的姓名和对应的成绩。代码清单 StudentGrades.java 如下。

```
import java.util.HashMap;
import java.util.Scanner;
public class StudentGrades {
    private HashMap<String, Integer> grades;
    public StudentGrades() {
        grades = new HashMap<>();
    }
    public void addGrade(String studentName, int grade) {
        grades.put(studentName, grade);
    }
    public Integer getGrade(String studentName) {
        return grades.get(studentName);
    }
    public boolean isStudentExists(String studentName) {
        return grades.containsKey(studentName);
    }
    public static void main(String[] args) {
        StudentGrades studentGrades = new StudentGrades();
        Scanner scanner = new Scanner(System.in);
        String studentName;
        int grade;
        while (true) {
            System.out.print("请输入学生姓名(输入 exit 退出):");
            studentName = scanner.nextLine();
            if (studentName.equalsIgnoreCase("exit")) {
                break;
            }
            if (studentGrades.isStudentExists(studentName)) {
                System.out.println("该学生已存在，请输入不同的姓名。");
                continue;          // 继续循环，要求输入新的学生姓名
            }
            System.out.print("请输入学生成绩:");
            grade = scanner.nextInt();
            scanner.nextLine();    // 清除换行符
            studentGrades.addGrade(studentName, grade);
            System.out.println("已添加 " + studentName + " 的成绩: " + grade);
        }
        System.out.print("请输入要查询的学生姓名:");
        studentName = scanner.nextLine();
        Integer queriedGrade = studentGrades.getGrade(studentName);
        if (queriedGrade != null) {
            System.out.println(studentName + " 的成绩是: " + queriedGrade);
        } else {
            System.out.println("未找到该学生的成绩。");
        }
        scanner.close();
    }
}
```

在例 5.18 的代码中，grades 是一个 HashMap，用于存储学生姓名和对应的成绩。使用 put() 方法将学生姓名与成绩关联。isStudentExists()方法用于检查学生姓名是否已存在于 grades 中。在添加成绩之前，调用 isStudentExists()方法能够检查学生姓名是否已存在。如果该学生姓名已存在，那么提示用户并继续循环，要求用户输入新的学生姓名。程序运行时，输入学生姓名和成绩的情况如图 5.4 所示。

```
请输入学生姓名（输入exit退出）：张三
请输入学生成绩：98
已添加 张三 的成绩：98
请输入学生姓名（输入exit退出）：李四
请输入学生成绩：95
已添加 李四 的成绩：95
请输入学生姓名（输入exit退出）：李四
该学生已存在，请使用不同的姓名。
请输入学生姓名（输入exit退出）：exit
请输入要查询的学生姓名：张三
张三 的成绩是：98

Process finished with exit code 0
```

图 5.4　代码 StudentGrades 的运行情况

编程中根据具体的需求选择合适的 Map 实现类，可以提高程序的性能和可维护性。例如，如果需要元素保持顺序，那么可以选择 LinkedHashMap 或 TreeMap；如果保持程序运行性能是关键，那么通常首选 HashMap 或 ConcurrentHashMap；如果在多线程环境中编写代码，那么应该使用 ConcurrentHashMap；如果需要代码自动清理过期条目，那么可以使用 WeakHashMap。

5.4.4　泛型集合

泛型集合是 Java 中用于处理对象集合的一种机制，它允许在定义集合时指定集合中元素的类型，从而提供类型安全性和更好的代码可读性。泛型集合主要用于集合框架，如 List、Set、Map 等。泛型集合提供了一种高效处理对象集合的方式，既可以增强类型安全性，又可以提高代码可读性。使用泛型集合的优点如下。

（1）类型安全：泛型集合能够在编译时检查类型，避免了运行时类型转换异常。

（2）代码可读性：通过指定类型，代码更易于理解和维护。

（3）消除强制类型转换：使用泛型集合后，不需要在取出元素时进行强制类型转换。

需要注意的是，Java 的泛型在编译时会进行类型擦除，运行时不保留泛型信息。泛型集合不可以使用基本数据类型，如 int、char 等，但可以使用其包装类，如 Integer、Character 等，也可以使用对象类型。

【例 5.19】 使用 TreeMap 对学生成绩和汉字姓名进行排序。代码清单 GenericCollectionDemo.java 如下。

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
public class GenericCollectionDemo {
    public static void main(String[] args) {
        Map<String, Integer> grades = new TreeMap<>();
        grades.put("李四", 95);
        grades.put("王五", 85);
        grades.put("张三", 98);
        grades.put("赵六", 85);
        // 将 Map 转换为 List 以便排序
        List<Map.Entry<String, Integer>> gradeList = new ArrayList<>(grades.entrySet());
        // 按成绩排序，若成绩相同则按姓名的 Unicode 值排序
```

```
            gradeList.sort((entry1, entry2) -> {
                int compare = entry2.getValue().compareTo(entry1.getValue()); // 按成绩降序
                if (compare == 0) {                    // 按姓名的 Unicode 值升序
                    return entry1.getKey().compareTo(entry2.getKey());
                }
                return compare;
            });
            System.out.println("按成绩排序的学生成绩:");
            for (Map.Entry<String, Integer> entry : gradeList) {
                System.out.println(entry.getKey() + " 的成绩是: " + entry.getValue());
            }
        }
    }
```

例 5.19 的代码使用 TreeMap 来存储学生姓名和成绩，并自定义比较器排序。List<Map.Entry<String, Integer>>中的每个元素都是一个 Map.Entry 对象。Map.Entry 是 Map 接口的一个内部接口，表示一个映射中的键值对(Key-Value Pair)。由于 Java 中 String 类的 compareTo()方法使用 Unicode 值进行比较，当成绩相同时，“赵”的 Unicode 值是 U+8D75，“王”的 Unicode 值是 U+738B。由于按照姓名的 Unicode 值进行升序排列，“王五”会排在“赵六”的前面。

程序运行时，输出按序排列后的情况如图 5.5 所示。

```
按成绩排序的学生成绩:
张三 的成绩是: 98
李四 的成绩是: 95
王五 的成绩是: 85
赵六 的成绩是: 85

Process finished with exit code 0
```

图 5.5　代码 GenericCollectionDemo 的运行情况

Map 是一个键值对集合，它的主要目的是存储和快速查找数据，而不是维护元素的顺序。Map 接口本身并不提供直接的排序功能，因此直接对 Map 进行排序是不可能的。如果需要根据值而不是键对 Map 中的条目进行排序，那么必须将 Map 的条目转换为一个可以排序的集合，如 List。

5.5　反 射 机 制

反射机制允许程序在运行时检查和操作类的属性、方法和其他组成部分。JDK 提供了一组 API，允许程序在运行时访问和操作类的结构。Java 的反射机制主要是通过 java.lang.reflect 包中的类来实现。在处理泛型时，利用反射可以获取泛型的类型参数的信息，但由于类型擦除，某些信息在编译时可能不可用。

5.5.1　获取泛型信息

反射机制有利于获取类或方法的泛型参数信息。利用反射机制在运行时动态处理泛型类型，这在一些情况下非常有用。反射机制特别适用于数据库框架、依赖注入、序列化、对象

关系映射(Object Relational Mapping,ORM)等开发环境。反射机制允许程序开发人员使用 getGenericSuperclass()方法或 getGenericInterfaces()方法来获取类的泛型父类或接口。

【例 5.20】 获取泛型的相关信息。代码清单 GenericReflectionDemo.java 如下。

```
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
class GenericClass<T> {
    private T value;
    public GenericClass(T value) {
        this.value = value;
    }
    public T getValue() {
        return value;
    }
}
class StringGenericClass extends GenericClass<String> {
    public StringGenericClass(String value) {
        super(value);
    }
}
public class GenericReflectionDemo {
    public static void main(String[] args) {
         StringGenericClass stringGeneric = new StringGenericClass("泛型类的实例对象
stringGeneric");
        Type superclass = stringGeneric.getClass().getGenericSuperclass();
        if (superclass instanceof ParameterizedType) {
            ParameterizedType parameterizedType = (ParameterizedType) superclass;
            Type[] actualTypeArguments = parameterizedType.getActualTypeArguments();
            System.out.println("泛型类型参数: " + actualTypeArguments[0]);
        }
    }
}
```

在例 5.20 的代码中,GenericClass<T>是一个泛型类,接收一个类型参数 T。子类 StringGenericClass 继承自 GenericClass<String>,指定具体的类型参数。使用反射机制调用 getGenericSuperclass()方法获取父类的类型信息。ParameterizedType 是 Java 反射 API 中的一个接口,表示带有类型参数,用于获取实际的类型参数,如泛型类或接口。

程序运行结果如下。

```
泛型类型参数: class java.lang.String
```

ParameterizedType 接口主要有以下方法。

(1) getActualTypeArguments(): 返回一个 Type 数组,是参数化类型的实际类型参数。

(2) getRawType(): 返回一个 Type 对象,表示原始类型,即未参数化的类型。

(3) getOwnerType(): 返回一个 Type 对象,表示拥有该参数化类型的类型,若没有拥有者类型,表明当前类型是顶级类型,则返回 null。

5.5.2 类型擦除

由于 Java 中的泛型在编译时会进行类型擦除,这意味着在运行时,泛型类型的信息会

被移除。编译时将泛型类型的信息移除，在很多情况下是为了确保代码的向后兼容性。

例如，定义一个泛型类或接口时，编译器会在编译时将所有的泛型类型参数替换为它们的边界类型（如果存在），或者替换为 Object 类型（如果没有指定边界）。这意味着在运行时，泛型类型的信息将不再可用。

此外，泛型类或泛型方法中使用类型参数时，可能发生类型擦除的情况。

【例 5.21】 类型擦除示例。代码清单 TypeErasureDemo.java 如下。

```
class GenericClass<T> {
    private T value;
    public GenericClass(T value) {
        this.value = value;
    }
    public T getValue() {
        return value;
    }
}
public class TypeErasureDemo {
    public static void main(String[] args) {
        GenericClass<String> stringInstance = new GenericClass<>("我是字符串");
        GenericClass<Integer> integerInstance = new GenericClass<>(123);
        System.out.println("String 实例对象: " + stringInstance.getValue());
        System.out.println("Integer 实例对象: " + integerInstance.getValue());
        checkTypeErasure(stringInstance);
        checkTypeErasure(integerInstance);
    }
    public static void checkTypeErasure(GenericClass<?> instance) {
        Class<?> clazz = instance.getClass();
        System.out.println("泛型类: " + clazz.getName());
        if (clazz.getTypeParameters().length == 0) {
            System.out.println("类型参数的长度: 0 (擦除)");
        } else {
            System.out.println("类型参数: " + clazz.getTypeParameters().length);
        }
    }
}
```

在例 5.21 的代码中，clazz.getTypeParameters()方法通常用于反射获取与类或接口相关联的类型参数。这个方法返回一个 TypeVariable<?>数组，表示该类或接口的所有类型参数。如果 getTypeParameters()方法获取类的类型参数时，输出的长度为 0，那么表示在运行时类型被擦除，无法获取泛型类型参数的信息。代码运行时，GenericClass<String>和 GenericClass<Integer>会被擦除为 GenericClass<Object>。

程序运行结果如下。

```
String 实例对象: 我是字符串
Integer 实例对象: 123
泛型类: GenericClass
类型参数: 1
泛型类: GenericClass
类型参数: 1
```

使用反射时，可能无法直接获取泛型参数的具体类型。这是反射和泛型结合使用时需

要特别注意的。为了保证运行安全，建议程序开发者在代码中通过检查类型参数来获取有关泛型类的更多信息，如类型参数的名称、边界等。当泛型参数的类型不明确时，程序开发者需要结合其他辅助手段（如文档、注释或设计约定等）来提供正确的信息。

5.5.3 创建泛型实例

使用反射可以动态创建泛型类型的实例。尽管由于类型擦除，我们无法直接创建带有具体类型参数的泛型实例，但是通过反射获取类对象和构造函数，可以创建原始类型的实例，并在运行时进行类型转换。此外，我们可以通过方法参数接收类型信息，来动态创建泛型集合等。例如，我们可以使用 Class.getConstructor()方法或 Class.getDeclaredConstructor()方法来获取构造器，再通过构造器的 newInstance()方法来创建泛型类的实例，还可以使用 getConstructor(Object.class)获取构造函数。

使用反射动态创建泛型类型的实例时，可能会抛出多种类型的异常，常见的异常如下。

(1) InstantiationException：当尝试创建一个抽象类或接口的实例时，程序会抛出此异常。

(2) IllegalAccessException：当尝试访问一个不可访问的构造函数或字段时，程序会抛出此异常。

(3) InvocationTargetException：当调用的构造函数抛出异常时，程序会抛出此异常。

(4) NoSuchMethodException：当请求的构造函数不存在时，程序会抛出此异常。

(5) SecurityException：当安全管理器阻止访问时，程序会抛出此异常。

上述异常都是在程序运行时可能发生的，因此在使用反射时，建议通过 try-catch 块来处理这些异常，以确保程序的健壮性和稳定性。此外，是否使用反射应酌情考虑，评估如果使用反射是否会引入安全风险，是否会导致性能降低。大多数情况下应尽量使用静态类型的解决方案，以保持代码的清晰性和稳定性。

5.6 综合运用

综合使用本章知识点，设计程序实现对用户信息管理的模拟。设计相关类，实现用户的增加、删除、修改、排序、统计等功能。

5.6.1 项目功能和类的设计

项目需要设计用户类 User 和用户管理类 UserManagement。类的设计思路是将用户信息的表示与用户管理的逻辑分开，使代码结构清晰，易于维护。上述两个类的封装情况如图 5.6 所示。

User 类用于封装用户的基本信息（包括姓名、年龄、电子邮件和唯一标识符），通过 getter 和 setter 访问器，来确保可以安全地访问和修改这些属性。使用 JSON 注解使该类能够方便地与 JSON 数据进行交互。UserManagement 类用于管理用户的生命周期（包括添加、查询、更新和删除用户），通过使用集合来存储用户，从而模拟简单的用户信息数据库。该类还提供统计和排序功能。

UserManagement 类和 User 类之间存在一对多的关系。这样的设计可以方便地对用

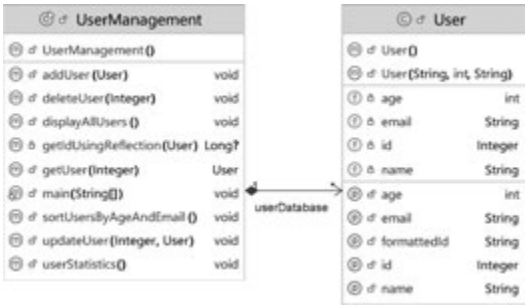

图 5.6　User 类和 UserManagement 类的封装情况

户信息进行操作，同时保持良好的代码组织结构。上述两个类的具体说明见表 5.5。

表 5.5　User 类和 UserManagement 类的具体说明

类	属　性	方　法
User	name（String）：用户的姓名，使用@JsonProperty 注解以便在 JSON 序列化和反序列化时使用自定义名称。 age（int）：用户的年龄，使用@JsonIgnore 注解以防止在 JSON 输出中显示该属性。 email（String）：用户的电子邮件地址，新增属性，用于存储用户的联系信息。 id（Integer）：用户的唯一标识符，新增属性，用于在用户管理系统中唯一标识每个用户	默认构造方法：用于创建 User 对象时不传递任何参数。 带参数的构造方法：允许在创建 User 对象时直接设置姓名、年龄和电子邮件。 getName()和 setName(String name)：获取和设置用户的姓名。 getAge()和 setAge(int age)：获取和设置用户的年龄。 getEmail()和 setEmail(String email)：获取和设置用户的电子邮件。 getId()和 setId(Integer id)：获取和设置用户的身份(Identity，ID)。 getFormattedId()：返回格式化后的 ID 字符串，以确保 ID 至少为 4 位数，不足时前面补零
UserManagement	userDatabase（List < User >）：一个列表，用于存储所有用户的集合，模拟用户数据库。 objectMapper（ObjectMapper）：用于处理 JSON 数据的对象，支持将 JSON 字符串转换为 User 对象	main(String[] args)：程序的入口，负责执行用户管理的主要逻辑。 displayAllUsers()：遍历 userDatabase 列表，打印所有用户的详细信息。 sortUsersByAgeAndEmail()：对用户列表按年龄和电子邮件进行排序，并打印。 userStatistics()：统计用户的总数、平均年龄、最大年龄和最小年龄，并打印。 addUser（User user）：将新用户添加到 userDatabase 列表中，并为其分配唯一的 ID。 getUser(Integer id)：根据用户 ID 查找并返回对应的 User 对象，若未找到则返回 null。 updateUser(Integer id, User updatedUser)：根据用户 ID 更新用户的信息。 deleteUser(Integer id)：根据用户 ID 删除对应的用户，并打印删除结果

5.6.2 在项目中使用 Jackson 库

Jackson 是一个流行的 JSON 数据处理库，它可以将 Java 对象转换为 JSON 数据，或者将 JSON 数据转换为 Java 对象。Jackson 是一个模块化的库，其核心有 3 个模块。这 3 个模块所对应的 JAR 文件具体如下。

(1) jackson-databind.jar：处理对象与 JSON 之间的映射。

(2) jackson-core.jar：提供核心功能，如 JSON 解析和生成。

(3) jackson-annotation s.jar：提供用于序列化和反序列化的注解。

因此，项目需要同时具备这 3 个模块所对应的 JAR 文件才能确保 Jackson 库正常工作。

在 Jackson 官网(https://www.json.org/)中，我们根据指引可以找到不同版本的 JAR 文件进行下载。我们在项目中添加 lib 文件夹，把所需的 JAR 文件复制到其中即可。在 IntelliJ IDEA 中，单击菜单栏中的 File→Project Structure 按钮(或使用快捷键 Ctrl+Alt+Shift+S)可以打开项目结构。在项目结构的左侧，我们先选择“Libraries”选项，单击右侧的 + 按钮，然后选择 Java 导航到项目的 lib 文件夹，接着选择 3 个 Jackson JAR 文件，最后单击 OK 按钮确认添加。单击 Apply 和 OK 按钮可保存对项目结构的更改。以 Jackson 2.17.2 版本为例，添加成功的项目结构如图 5.7 所示。添加后就可以在自己的项目中使用 Jackson 库。

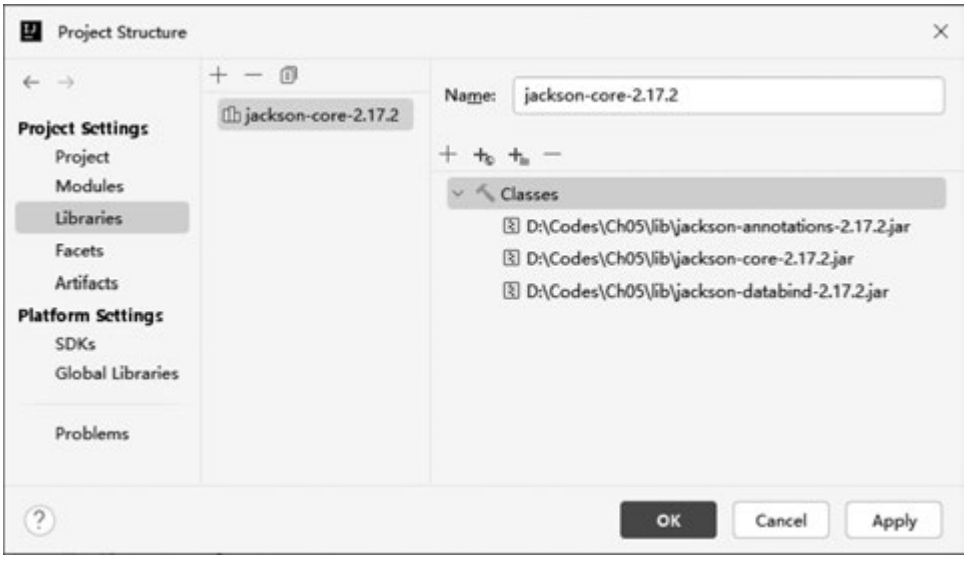

图 5.7 项目结构中添加 lib 的 Jackson 库

5.6.3 类的代码清单

在 User 类中，用户的姓名使用@JsonProperty 注解以便在 JSON 串中使用自定义名称。JSON 转换中只关注 name 和 age，所以设计两个注解。这样，UserManagement 类能够解析 JSON 串为对象，并按正常逻辑处理用户数据。代码清单 User.java 如下。

```
import com.fasterxml.jackson.annotation.JsonProperty;
public class User {
    @JsonProperty("user_name")
    private String name;
    @JsonProperty("user_age")
    private int age;
    private String email;
    private Integer id;
    public User() {}
    public User(String name, int age, String email) {
        this.name = name;
        this.age = age;
        this.email = email;
    }
    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;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getFormattedId() {
        return String.format("%04d", id); // 格式化后的 ID 字符串,不足 4 位前面补零
    }
}
```

使用 List 集合可以动态地添加、删除和访问用户对象,因此考虑把 User 对象存储在一个 List 泛型中。

为了简化集合操作,特别是在处理用户数据的集合时,使用 Stream 是一种很好的办法。Stream 可以对元素序列进行聚合操作。Stream 不存储数据,而是通过管道(Pipeline,它是指一系列的计算步骤,每个步骤都可以对数据进行转换或处理)处理数据。Stream 的操作主要如下。

(1) 中间操作:如 filter()、map()、sorted()等,这些操作返回一个新的 Stream。

(2) 终止操作:如 collect()、forEach()、count()等,这些操作会触发 Stream 的处理并

返回结果。

Stream 链式操作使数据处理更加简洁和高效。中间操作和终止操作相结合，可以实现复杂的数据处理逻辑。Stream 是一种一次性的数据处理工具，一旦执行了终止操作，如 collect()、forEach() 等，该 Stream 就会被消耗，无法再次使用。因此，每次需要处理数据时，都需要重新创建一个 Stream。

代码清单 UserManagement.java 如下。

```
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.IOException;
import java.lang.reflect.Field;
import java.util.*;
import java.util.stream.Collectors;
public class UserManagement {
    private static List<User> userDatabase = new ArrayList<>(); // 模拟数据存储
    private static ObjectMapper objectMapper = new ObjectMapper();
    public static void main(String[] args) {
        UserManagement management = new UserManagement();
        String jsonInput = "{\"user_name\":\"张三\",\"user_age\":30,\"email\":\"elfgirl
@sei.zua\"}";
        try {
            User user = objectMapper.readValue(jsonInput, User.class);
            management.addUser(user);
        } catch (IOException e) {
            e.printStackTrace();
        }
        management.addUser(new User("李四", 25, "loyalfriends@sei.zua"));
        management.addUser(new User("王五", 30, "modelkid@sei.zua"));
        management.addUser(new User("赵六", 30, "gohgenius@sei.zua"));
        management.addUser(new User("孙七", 25, "buddhistyouth@sei.zua"));
        management.displayAllUsers();
        User retrievedUser = management.getUser(2);
        if (retrievedUser != null) {
            System.out.println("查询到的用户信息:");
            System.out.println("姓名: " + retrievedUser.getName());
            System.out.println("年龄: " + retrievedUser.getAge());
            System.out.println("邮箱: " + retrievedUser.getEmail());
            System.out.println("邮箱: " + retrievedUser.getFormattedId());
        } else {
            System.out.println("用户未找到。");
        }
        User updatedUser = new User("张三", 31, "loyalfriends@sei.zua");
        management.updateUser(2, updatedUser);
        management.sortUsersByAgeAndEmail();
        management.userStatistics();
        management.deleteUser(2);
        management.displayAllUsers();
    }
    public void displayAllUsers() {
        System.out.println("当前用户列表:");
        for (User user : userDatabase) {
            System.out.println("用户 ID:" + user.getFormattedId() + ",姓名: " + user.
getName() + ", 年龄: " + user.getAge() + ", 邮箱: " + user.getEmail());
```

```
            }
        }
        public void sortUsersByAgeAndEmail() {
            List<User> sortedUsers = userDatabase.stream()
                    .sorted(Comparator.comparingInt(User::getAge)
                            .thenComparing(User::getEmail)) // 按年龄和邮箱排序
                    .collect(Collectors.toList());
            System.out.println("按年龄和邮箱升序排列后的用户列表:");
            for (User user : sortedUsers) {
                System.out.println("姓名: " + user.getName() + ", 年龄: " + user.getAge()
    + ", 邮箱: " + user.getEmail());
            }
        }
        public void userStatistics() {
            long totalCount = userDatabase.size();
            OptionalDouble averageAge = userDatabase.stream().mapToInt(User::getAge).average();
            OptionalInt maxAge = userDatabase.stream().mapToInt(User::getAge).max();
            OptionalInt minAge = userDatabase.stream().mapToInt(User::getAge).min();
            System.out.println("用户统计信息:");
            System.out.println("用户总数: " + totalCount);
            System.out.println("平均年龄: " + (averageAge.isPresent() ? averageAge.getAsDouble() : 0));
            System.out.println("最大年龄: " + (maxAge.isPresent() ? maxAge.getAsInt() : 0));
            System.out.println("最小年龄: " + (minAge.isPresent() ? minAge.getAsInt() : 0));
        }
        public void addUser(User user) {
            user.setId((Integer) (userDatabase.size() + 1)); // 设置 ID
            userDatabase.add(user);
        }
        public User getUser(Integer id) {
            return userDatabase.stream()
                    .filter(user -> user.getId().equals(id))
                    .findFirst()
                    .orElse(null);
        }
        public void updateUser(Integer id, User updatedUser) {
            User user = getUser(id);
            if (user != null) {
                user.setName(updatedUser.getName());
                user.setAge(updatedUser.getAge());
                user.setEmail(updatedUser.getEmail());
            } else {
                System.out.println("用户未找到,无法更新。");
            }
        }
        public void deleteUser(Integer id) {
            User userToDelete = getUser(id);
            if (userToDelete != null) {
                boolean removed = userDatabase.removeIf(user -> user.getId().equals(id));
                if (removed) {
                    System.out.printf("ID 为 %s 的用户已被删除。%n", userToDelete.getFormattedId());
```

```
            }
        } else {
            System.out.println("用户未找到,无法删除。");
        }
    }
    private Long getIdUsingReflection(User user) {
        try {
            Field idField = User.class.getDeclaredField("id");
            idField.setAccessible(true);
            return (Long) idField.get(user);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

在上述代码中,UserManagement 类的静态成员变量 userDatabase,是一个 List < User >集合,用于存储所有的用户对象。每次调用 userDatabase.stream()都可以直接进行函数链式操作,使代码更加简洁和易于理解,还可以根据需要灵活地应用不同的操作,而不需要担心之前的操作影响当前的处理逻辑。

在上述代码中,用户张三是通过 JSON 字符串创建的 User 对象。在处理 JSON 字符串时,有固定的读写方式和异常处理结构,这些在后续内容中再展开介绍。其他几个用户是通过直接创建 User 类的对象添加到集合中的。模拟用户管理的程序运行情况如图 5.8 所示。

```
当前用户列表:
用户ID:0001,姓名: 张三, 年龄: 30, 邮箱: elfgirl@sei.zua
用户ID:0002,姓名: 李四, 年龄: 25, 邮箱: loyalfriends@sei.zua
用户ID:0003,姓名: 王五, 年龄: 30, 邮箱: modelkid@sei.zua
用户ID:0004,姓名: 赵六, 年龄: 30, 邮箱: gohgenius@sei.zua
用户ID:0005,姓名: 孙七, 年龄: 25, 邮箱: buddhistyouth@sei.zua
查询到的用户信息:
姓名: 李四
年龄: 25
邮箱: loyalfriends@sei.zua
邮箱: 0002
按年龄和邮箱升序排列后的用户列表:
姓名: 孙七, 年龄: 25, 邮箱: buddhistyouth@sei.zua
姓名: 张三, 年龄: 30, 邮箱: elfgirl@sei.zua
姓名: 赵六, 年龄: 30, 邮箱: gohgenius@sei.zua
姓名: 王五, 年龄: 30, 邮箱: modelkid@sei.zua
姓名: 张三, 年龄: 31, 邮箱: loyalfriends@sei.zua
用户统计信息:
用户总数: 5
平均年龄: 29.2
最大年龄: 31
最小年龄: 25
ID为0002的用户已被删除。
当前用户列表:
用户ID:0001,姓名: 张三, 年龄: 30, 邮箱: elfgirl@sei.zua
用户ID:0003,姓名: 王五, 年龄: 30, 邮箱: modelkid@sei.zua
用户ID:0004,姓名: 赵六, 年龄: 30, 邮箱: gohgenius@sei.zua
用户ID:0005,姓名: 孙七, 年龄: 25, 邮箱: buddhistyouth@sei.zua

Process finished with exit code 0
```

图 5.8　模拟用户管理的程序运行情况

使用数据流可以简化集合的高级操作。Java 中的 Stream 是静态类型的，可以操作任何类型的元素。而且 Stream 按需计算，只在需要时才处理元素，这与传统的 for 循环一次处理所有元素不同。此外，Stream 可以很容易地转换为不同的数据视图，进行复杂的数据处理，如映射、过滤和排序等。

习　题　5

一、填空题

1. 静态内部类无法直接访问非静态成员，可以通过两种间接方法来访问。一种是通过在代码中先创建_____，再访问实例成员；另一种是在内部类中持有_____。

2. 使用_____可以在不依赖外部类实例的情况下组织类的结构。

3. 局部内部类是在一个方法内定义的类，只能在该方法_____使用。它可以访问方法的局部变量(必须是_____或者事实上是_____的变量)，也可以访问_____。

4. 回调通常指的是将一个方法的引用传递到另一个方法中。实现回调非常灵活，常见的 4 种方式有_____、_____、_____和_____。

5. 当且仅当一个接口中只有一个需要实现的方法，才可以使用 Lambda 表达式简化接口的实现类。这样的接口称为_____。

6. 泛型接口是 Java 泛型编程的基础，在定义接口时指定_____。通过创建不同的实现类，每个实现类使用不同的具体类型，从而实现同一接口的_____和_____。

7. ArrayList 类是基于_____实现的。ArrayList 类适合频繁_____的场景，插入和删除操作相对较慢。

8. Java 中的泛型在编译时会进行类型擦除，在很多情况下是为了确保代码的_____。

二、思考题

1. 什么是内部类？请举例说明内部类的定义和使用场景。

2. 内部类和外部类有什么区别和联系？

3. 匿名类的主要用途是什么？请给出使用匿名类实现接口的具体场景。

4. 什么是泛型？请解释泛型的好处，并给出一个使用泛型的简单示例。

5. 请列出 Java 中常用的集合类，并简要说明它们的特点和适用场景。

6. 什么是泛型的边界？请解释上界和下界。

7. 如何理解 Java 反射的概念？举例说明如何使用反射获取类信息(如类名、方法名等)？

8. 如何对一个 List 集合进行排序？如何对自定义对象进行排序？

9. 解释 Java 中的类型擦除机制，并讨论它对泛型的影响。

10. 什么情况下使用反射是合适的？什么情况下应避免使用反射？

11. 什么是函数式接口？Lambda 表达式如何与函数式接口结合使用？

12. 如何使用 Lambda 表达式替代匿名类？

三、程序设计

1. 设计一个简单的计算器类 Calculator，其中包含一个内部类 Operation，用于执行加法、减法、乘法和除法。用户可以通过调用 Calculator 类的方法传入参数，并返回计算结果。

2. 设计一个泛型栈类 Stack < T >,实现基本的入栈、出栈和查看栈顶元素的方法,并编写测试代码验证其功能。

3. 创建一个产品类 Product,包含名称和价格。使用 List < Product >存储多个产品,并使用 Lambda 表达式实现按价格排序和过滤价格低于某个值的产品。

4. 创建一个员工类 Employee,包含姓名和薪水。使用 List < Employee >存储多个员工,并使用 Lambda 表达式实现按薪水排序和过滤薪水高于某个值的员工。

5. 创建一个联系人类 Contact,包含姓名和电话号码。使用 HashMap 存储联系人信息,并提供添加、查找和删除联系人的功能。