INFO1113

Xiyuan Pan

March 2025

目录

1 Introduction

INFO1113 是悉尼大学开设的面向对象编程(Object-Oriented Programming) 课程，旨在教授学生面向对象编程的核心概念和实践技能。使用 Java 语言进行面向对象编程实践，涵盖类的定义、方法调用、实例创建等。

2 Week1

本周主要介绍 Java 的基础语法，包括:

- Java 语法结构
- Java 源代码、编译与执行
- 数据类型
- 命令行参数
- 输入与 ‘Scanner‘
- 条件语句（‘if-else‘）

2.1 Lecture

2.1.1 Java 语法结构

Java 代码通常由 **类（class）** 组成，主方法（main method）是 Java 程序的入口。

2.1.2 Hello World 程序

Java 的第一个程序通常是 ‘Hello World‘, 它的作用是打印 ‘”Hello World!”‘ 到屏幕:

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello␣World!");
4     }
5 }
```

Listing 1: Hello World 程序

这个程序包含：

- **类定义**：‘public class HelloWorld‘
- **主方法**：‘public static void main(String[] args)‘
- **输出语句**：‘System.out.println(”Hello World!”);‘

运行此程序后，终端将输出：

```
Hello World!
```

2.1.3 解释型语言 vs. 编译型语言

Python（解释执行）：

```
> python myprogram.py
Hello Everyone! This is python! Remember me?
```

Python 代码直接运行，无需编译。

Java（编译执行）：

```
> javac MyProgram.java  % 先编译
> java MyProgram        % 运行编译后的字节码
Hello Everyone! This is a java program
```

解释 vs. 编译对比：

- **解释型语言**（如 Python）：逐行解释执行，无需编译。
- **编译型语言**（如 C++）：需要先编译，才能运行。
- **Java 介于二者之间**：Java 代码先编译为 **字节码**，然后由 JVM 解释执行。

2.1.4 Java 编译过程

Java 代码编译流程：

1. **Java 源代码**（`.java` 文件）：程序员编写的 Java 代码。
2. **Java 编译器**（`javac`）：将 Java 源代码编译为字节码。

3. **字节码文件**（.class 文件）：不可读的 Java 字节码，由 JVM 解释执行。

示例：

```
> javac MyProgram.java   % 编译
> java MyProgram         % 运行
Hello, Java!
```

JVM 的作用：

- 解释并运行字节码。
- 使 Java 具备**跨平台性**（"Write once, run anywhere"）。

2.1.5 Java 源文件结构解析

Java 代码示例：

```java
import java.util.Scanner;

public class Anatomy {
    public static void main(String[] args) {
        System.out.println("Hello! This will output to the
            screen");
        int integerVar = 1;
        double d1 = 1.5;
        Scanner keyboard = new Scanner(System.in);
        String s = "This is a string!";
        s = keyboard.nextLine();
        System.out.println(s);
    }
}
```

Listing 2: Java 源文件示例

代码结构解析：

- **导入 Scanner 类**：`import java.util.Scanner;`
- **类定义**：`public class Anatomy`

- **主方法**：`public static void main(String[] args)`
- **变量声明**：
  - `int integerVar = 1;`（整数变量）
  - `double d1 = 1.5;`（浮点数变量）
  - `String s = "This is a string!";`（字符串变量）
- **用户输入**：`Scanner keyboard = new Scanner(System.in);`
- **输出语句**：`System.out.println(s);`

Java 代码执行流程：

```
> javac Anatomy.java  % 编译
> java Anatomy        % 运行
Hello! This will output to the screen
```

2.1.6 Java 作用域（Scope）解析

Java 代码示例：

```java
public class Scoping {
    public static void main(String[] args) {
        int x = 0; // x 作用域开始
        {
            int y = 1; // y 作用域开始
            {
                int z = 2; // z 作用域开始
            } // z 作用域结束
        } // y 作用域结束
    } // x 作用域结束
}
```

Listing 3: 变量作用域示例

变量作用域范围：

- 变量 x 可在整个 `main()` 方法内使用。
- 变量 y 仅可在 y 所在的 {} 代码块内使用。

- 变量 `z` 仅可在 `z` 所在的 {} 代码块内使用，不能在 `y` 作用域外访问。

作用域访问规则:

- 外层变量可以被内层访问。
- 内层变量不能被外层访问。
- 同级作用域的变量互不影响。

示例测试：

```java
public class ScopingTest {
    public static void main(String[] args) {
        int x = 10;
        {
            int y = 20;
            {
                int z = 30;
                System.out.println(x); //  访问外层变量
                System.out.println(y); //  访问上一层变量
                System.out.println(z); //  访问当前层变量
            }
            System.out.println(y); //  y 仍然可访问
            // System.out.println(z); //  错误，z 超出作用域
        }
        // System.out.println(y); //  错误，y 超出作用域
    }
}
```

Listing 4: 变量作用域访问测试

2.1.7 Java 数据类型与变量

1. Java 数据类型概述

Java 主要分为：

- **基本数据类型（Primitive Types）**：存储数据本身。
- **引用数据类型（Reference Types）**：存储对象的地址。

2. 基本数据类型：

3. 引用数据类型：

- `String`：字符串类型，存储字符数组。
- `Arrays`：存储相同类型的元素集合。
- `Class`：用户自定义类也是引用类型。

4. 变量存储机制：

Name	Kind	Memory	Range
boolean	boolean	1 byte	true / false
byte	integer	1 byte	-128 to 127
short	integer	2 bytes	-32,768 to 32,767
int	integer	4 bytes	-2,147,483,648 to 2,147,483,647
long	integer	8 bytes	$\sim 9 \times 10^{18}$
float	floating-point	4 bytes	$\pm 3.4 \times 10^{38}$
double	floating-point	8 bytes	$\pm 1.8 \times 10^{308}$
char	character	2 bytes	Unicode 0 to 65535

- **基本数据类型**直接存储值，在**栈（Stack）**中存储。
- **引用数据类型**存储地址，实际对象存储在**堆（Heap）**中。

2.1.8 Java 类型转换（Type Conversion）

1. Java 支持两种类型转换：

- **自动类型转换（Widening Casting）**：低精度 → 高精度，自动完成。
- **强制类型转换（Narrowing Casting）**：高精度 → 低精度，需手动指定。

2. 代码示例：

```java
public class TypeCastingExample {
    public static void main(String[] args) {
        int numberOfItem = 1;
        double price = 1.0;

        System.out.println(numberOfItem / 2); // 0
        System.out.println(price / 2); // 0.5
        System.out.println(price / numberOfItem); // 1.0
    }
}
```

Listing 5: Java 类型转换

3. 结果解析：

- `numberOfItem / 2`:
  - `numberOfItem` 是 `int`，`2` 也是 `int`，整数除法截断小数部分。
  - 计算结果是 `0`（而不是 `0.5`）。
- `price / 2`:
  - `price` 是 `double`，`2` 是 `int`，Java 进行自动转换 `2 → 2.0`。
  - 计算结果是 `0.5`。
- `price / numberOfItem`:
  - `numberOfItem` 是 `int`，但 `price` 是 `double`，Java 进行自动转换 `1 → 1.0`。
  - 计算结果是 `1.0`。

4. Java 类型转换规则:

- **自动类型转换**: `byte → short → int → long → float → double`
- **强制类型转换**: `double → float → long → int → short → byte`

5. 重点总结:

规则	说明
`int / int`	结果是 `int`，小数部分截断
`double / int`	`int` 自动转换为 `double`，结果是 `double`
`double / double`	结果是 `double`
`(int) double`	可能丢失精度，截断小数部分

2.1.9 Java 命令行参数（Command Line Arguments）

1. 什么是命令行参数？

- Java 允许程序通过 `main(String[] args)` 方法接收命令行输入。
- 这些参数以 **字符串数组** `args` 的形式传递。
- 运行示例:

```
    javac CommandLineArgs.java
    java CommandLineArgs Hello
```

2. 代码示例

```java
public class CommandLineArgs {
    public static void main(String[] args) {
        String input = args[0];  // 获取第一个命令行参数
        System.out.println(input + "!");
    }
}
```

Listing 6: 命令行参数示例

3. 运行流程

- **编译代码**:

```
    javac CommandLineArgs.java
```

- **运行程序并传递参数**:

```
    java CommandLineArgs Hi
```

- **输出结果**:

```
    Hi!
```

4. 处理多个参数

```java
public class CommandLineExample {
    public static void main(String[] args) {
        System.out.println("第一个参数: " + args[0]);
        System.out.println("第二个参数: " + args[1]);
    }
}
```

Listing 7: 访问多个命令行参数

5. 防止数组越界异常

```java
public class SafeCommandLine {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("输入参数:␣" + args[0]);
        } else {
            System.out.println("未提供命令行参数");
        }
    }
}
```

Listing 8: 检查参数是否为空

6. 类型转换

```java
public class SumCalculator {
    public static void main(String[] args) {
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        int sum = num1 + num2;
        System.out.println("和:␣" + sum);
    }
}
```

Listing 9: 转换字符串参数为整数

7. 重点总结

规则	说明
`String[] args`	Java ‘main‘ 方法接收命令行参数的数组
`args[0]`	获取第一个命令行参数
`args.length`	获取命令行参数的个数，避免 `ArrayIndexOutOfBoundsException`
`Integer.parseInt(args[i])`	将命令行参数转换为 `int`
`Double.parseDouble(args[i])`	将命令行参数转换为 `double`

2.1.10 Java Scanner 类与用户输入

1. 什么是 Scanner？

- Scanner 是 Java 提供的 **输入类**，用于从 **键盘（System.in）**、**文件**、**字符串**读取数据。
- 需要导入：`import java.util.Scanner;`

2. 基本使用示例

```
import java.util.Scanner;

public class UsingScanner {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("请输入一行文字:");
        String s = keyboard.nextLine();

        System.out.println("你输入的是:␣" + s);
    }
}
```

Listing 10: Scanner 读取用户输入

3. 常见 Scanner 方法

方法	作用	示例输入	示例输出
`nextLine()`	读取一整行	Hello Java	”Hello Java”
`next()`	读取一个单词	Hello World	”Hello”
`nextInt()`	读取整数	42	42
`nextDouble()`	读取小数	3.14	3.14

4. 避免 nextInt() 和 nextLine() 连用

```
import java.util.Scanner;

public class ScannerIssue {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
```

```
6
7           System.out.print("请输入年龄:␣");
8           int age = input.nextInt();  // 读取整数，但不会消费换行
                符
9
10          input.nextLine();  // 清除换行符
11
12          System.out.print("请输入姓名:␣");
13          String name = input.nextLine();  // 读取字符串
14
15          System.out.println("姓名:␣" + name + ",␣年龄:␣" + age);
16      }
17  }
```

Listing 11: 清除换行符避免跳过输入

5. 关闭 Scanner

```
1  Scanner scanner = new Scanner(System.in);
2  scanner.close();  // 关闭 Scanner
```

Listing 12: 关闭 Scanner 避免资源泄漏

6. 重点总结

规则	说明
`import java.util.Scanner;`	需要手动导入 Scanner 类
`Scanner scanner = new Scanner(System.in);`	创建 Scanner 对象
`scanner.nextLine();`	读取整行字符串
`scanner.next();`	读取单个单词
`scanner.nextInt();`	读取整数
`scanner.nextDouble();`	读取小数
`scanner.nextLine();` 之后要清空换行符	避免 ‘nextInt()‘ 之后 ‘nextLine()‘ 被跳过
`scanner.close();`	用完 Scanner 需要关闭

2.1.11 Java 布尔类型（Boolean）与逻辑运算

1. Boolean 基础

- Java 的 `boolean` 变量只能取值 `true` 或 `false`。
- **不能使用 0 或 1** 代替 `true` 或 `false`。
- **默认值**：如果未初始化，`boolean` 变量默认为 `false`。

2. 逻辑运算符

运算符	名称	作用	示例	结果
`&&`	逻辑与（AND）	两个条件都为 true 才返回 true	`true && true`	true
`||`	逻辑或（OR）	只要有一个条件为 true 就返回 true	`true || false`	true
`!`	逻辑非（NOT）	取反 true 变 false，false 变 true	`!true`	false

3. 逻辑运算示例

```java
public class BooleanExample {
    public static void main(String[] args) {
        boolean t = true;
        boolean f = false;

        boolean exampleAnd = t && f;  // false
        boolean exampleOr = f || t;   // true
        boolean exampleNot = !t;      // false

        System.out.println("t␣&&␣f␣=␣" + exampleAnd);
        System.out.println("f␣||␣t␣=␣" + exampleOr);
        System.out.println("!t␣=␣" + exampleNot);
    }
}
```

Listing 13: Java 逻辑运算示例

4. 逻辑运算符的短路特性

```java
public class ShortCircuit {
    public static void main(String[] args) {
        int x = 0;
        boolean result = (x > 0) && (++x > 0);  // x > 0 为 false, ++x 不执行
        System.out.println("x = " + x);  // x 仍然是 0

        boolean result2 = (x < 1) || (++x > 0); // x < 1 为 true, ++x 不执行
        System.out.println("x = " + x);  // x 仍然是 0
    }
}
```

Listing 14: 短路计算示例

5. 逻辑运算与 if 语句

```java
public class IfExample {
    public static void main(String[] args) {
        boolean isRaining = false;
        boolean haveUmbrella = true;

        if (!isRaining || haveUmbrella) {
            System.out.println("可以出门");
        } else {
            System.out.println("不能出门");
        }
    }
}
```

Listing 15: Java 逻辑运算与 if 语句

2.1.12 Java 条件判断（If-else 语句）与三元运算符

1. If-else 语句

```java
public class IfElseProgram {
    public static void main(String[] args) {
        int marks = 60;
```

```
        String result;

        if (marks >= 50) {
            result = "Pass";
        } else {
            result = "Fail";
        }

        System.out.println(result);
    }
}
```

Listing 16: Java If-else 语句示例

2. If-else-if 语句

```
public class GradeCheck {
    public static void main(String[] args) {
        int marks = 85;
        String grade;

        if (marks >= 90) {
            grade = "A";
        } else if (marks >= 80) {
            grade = "B";
        } else if (marks >= 70) {
            grade = "C";
        } else if (marks >= 60) {
            grade = "D";
        } else {
            grade = "F";
        }

        System.out.println("Your grade: " + grade);
    }
}
```

Listing 17: If-else-if 语句示例

3. 三元运算符

```java
public class TernaryExample {
    public static void main(String[] args) {
        int marks = 60;
        String result = (marks >= 50) ? "Pass" : "Fail"; // 三元运算符
        System.out.println(result);
    }
}
```

Listing 18: 三元运算符示例

4. If-else 语句 vs 三元运算符

```java
// 方式 1: if-else 语句
if (marks >= 50) {
    result = "Pass";
} else {
    result = "Fail";
}

// 方式 2: 三元运算符
result = (marks >= 50) ? "Pass" : "Fail";
```

Listing 19: If-else vs 三元运算符

方式	代码量	可读性	适用场景
If-else 语句	代码较长	可读性较高	适用于复杂逻辑
三元运算符 ? :	代码简洁	可读性较低（嵌套时）	适用于简单的条件赋值

2.2 Tutorial

2.2.1 Java 环境准备——常用终端命令

在学习 Java 编程之前，我们需要熟悉终端（Terminal）和 UNIX 命令行环境。以下是一些常用命令：

2.2.2 目录符号（Directory Symbols）

这些符号可以帮助你在命令行中高效地遍历文件系统：

Command	Description
`pwd`	Print the current working directory（显示当前目录路径）
`cd`	Change current directory（切换当前目录）
`mkdir`	Create new directory（创建新目录）
`ls`	List the contents of the current directory（列出当前目录内容）
`rmdir`	Remove directory（删除目录）
`rm`	Remove a file（删除文件）
`mv <src> <dest>`	Move (and rename) <src> to <dest>（移动或重命名文件）
`cp`	Copy a file（复制文件）
`diff`	Display the differences in the files by comparing them line by line（逐行比较文件差异）

表 1: 常用 UNIX 命令

Symbol	Description	Usage
`..`	Parent directory（上一级目录）	`cd ..`
`-`	Previous directory（上一个目录）	`cd -`
`~`	Home directory（用户主目录）	`cd ~`
`/`	Root directory（根目录）	`cd /`

表 2: 常用目录符号

2.2.3 实践练习

在终端环境中尝试以下操作：

- 使用 `mkdir` 创建一个名为 `OOP` 的目录。
- 使用 `cd` 进入 `OOP` 目录。
- 使用 `touch` 创建一个文件，文件名为 `HelloWorld.java`。

```
mkdir OOP
cd OOP
touch HelloWorld.java
```

Listing 20: 实践命令示例

1. If 语句

```
if (条件) {
    // 当条件为 true 时执行
}
```

Listing 21: Java If 语句示例

2. Java 需要 boolean 类型

```
public class IfProgram {
    public static void main(String[] args) {
        int a1 = 1;
        if (a1) { //   错误, a1 不是 boolean
            System.out.println("This will be executed");
        }
    }
}
```

Listing 22: Java If 语句错误示例

3. 正确的写法

```
//   显式比较
if (a1 != 0) {
    System.out.println("This will be executed");
}
```

Listing 23: Java If 语句正确示例

4. If-else 语句

```
if (score >= 60) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

Listing 24: If-else 语句

5. If-else-if 语句

```
if (marks >= 90) {
    grade = "A";
} else if (marks >= 80) {
```

```
    grade = "B";
} else {
    grade = "C";
}
```

Listing 25: If-else-if 语句示例

6. 三元运算符

```
String result = (score >= 60) ? "Pass" : "Fail";
```

Listing 26: 三元运算符

7. Java If 语句总结

特性	描述
If 语句	仅在条件 true 时执行
If-else 语句	true 执行 if 块，false 执行 else 块
If-else-if 语句	处理多个条件的分支
Java 只接受 boolean 类型条件	不能像 C 语言使用 int 作为条件
三元运算符	简化 if-else 语句

2.2.4 编译 Java 程序（Compiling Java）

Java 是一种编译型语言，源代码需要编译为字节码后，由 JVM 执行。

示例代码：

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

编译与运行步骤：

1. 文件命名：必须为 类名.java，如 Hello.java。
2. 编译文件：javac Hello.java，生成 Hello.class。
3. 运行程序：java Hello（不加.class 后缀）。

Java 11 单文件执行：

- 直接运行源文件（无需编译）：

```
    java Hello.java
```

- 仅支持 Java 11 及以上版本。

2.2.5 Scanner 类的常用方法详解

- `hasNext()`：判断输入流中是否还有下一个输入项。**返回值**：`boolean` 类型，若有更多输入返回 `true`，否则返回 `false`。
- `next()`：读取下一个输入项（以空格或分隔符为界）。**返回值**：`String` 类型，仅返回分隔符前的数据，不包括空格和换行。
- `useDelimiter(String pattern)`：设置输入分隔符，改变默认的空白字符分隔行为。**参数**：正则表达式（例如 `","` 表示以逗号分隔）。

方法对比总结：

方法	作用	典型用法
`hasNext()`	判断是否还有输入项。	常用在 `while` 循环中。
`next()`	获取下一个输入（以默认分隔符或自定义分隔符为界）。	读取单词或下一个数据项。
`useDelimiter()`	设置输入的分隔符。	处理逗号分隔、分号分隔等输入格式。

表 3: Scanner 常用方法对比

示例代码：

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

```

```
        // 自定义分隔符为英文逗号
        scan.useDelimiter(",");

        System.out.println("请输入3个数据，使用英文逗号分隔：")
            ;

        // hasNext() 判断是否还有数据
        while (scan.hasNext()) {
            // next() 读取下一个输入
            String data = scan.next();
            System.out.println("读取到的数据是：" + data);
        }

        scan.close(); // 关闭 Scanner
    }
}
```

Listing 27: Scanner 输入示例（使用分隔符）

输入示例:

```
apple,banana,orange
```

输出示例:

```
请输入3个数据，使用英文逗号分隔：
读取到的数据是：apple
读取到的数据是：banana
读取到的数据是：orange
```

2.2.6 Java Arrays 类

1. 概述

- `java.util.Arrays` 是一个工具类，提供常用的数组操作方法。
- 主要作用包括：数组排序、搜索、比较、填充和转换等。

2. 常见方法说明

方法	说明
`Arrays.toString(array)`	将一维数组转换为字符串表示，便于输出查看。
`Arrays.sort(array)`	对数组元素进行排序（按升序）。
`Arrays.equals(array1, array2)`	比较两个数组是否相等（元素逐个比较）。
`Arrays.fill(array, value)`	使用指定值填充整个数组。
`Arrays.binarySearch(array, key)`	在已排序数组中查找指定元素，返回索引值。

表 4: java.util.Arrays 常用方法

3. 为什么使用 `Arrays.toString()`

- 直接打印数组名（如 `System.out.println(args)`）输出的是数组的内存地址，而不是元素内容。
- 使用 `Arrays.toString(array)` 能清晰显示数组元素列表。

4. 示例代码:

```
import java.util.Arrays;

public class ProgramClass {
    public static void main(String[] args) {
        // 输出 args 数组的字符串表示
        System.out.println(Arrays.toString(args));
    }
}
```

Listing 28: Arrays.toString() 示例

5. 命令行执行示例

```
$ javac ProgramClass.java
$ java ProgramClass one two three
```

输出:

```
[one, two, three]
```

6. 什么时候引入 `java.util.Arrays`

- 需要打印数组内容（调试、展示数据时）。
- 需要排序数组（排序算法调用）。
- 需要比较两个数组是否相等。
- 需要查找或填充数组。

2.2.7 布尔类型与条件语句 (Boolean Type, Conditional Statements)

1. 基本语法 Java 使用 `if-else` 语句实现条件分支。

```
if (expression) {
    // 代码块
}
```

2. 表达式类型要求

- **表达式 (expression) 必须返回 `boolean` 类型**，结果为 `true` 或 `false`。
- Java 与 C/C++ 不同，不允许将整数直接用作条件判断。

3. 正确示例

```
int x = 5;
boolean exp = x < 6;

if (exp) {
    System.out.println("x is less than 6");
}
```

或者可以直接写表达式：

```
int x = 5;

if (x < 6) {
    System.out.println("x is less than 6");
}
```

4. 错误示例（不要这样写）

```java
int x = 5;

if (x) { // 错误！不能将 int 类型用作条件表达式
    System.out.println("x is not zero");
}
```

5. 小结

- Java 的 `if` 语句只接受 `boolean` 类型。
- 条件表达式通常通过比较运算符（例如 `==`、`<`、`>` 等）产生布尔值。

2.2.8 Java 模块导入与包（Importing Modules and Packages）

1. 什么是包（Package）？

- Java 中，一组相关类（Classes）和接口（Interfaces）被组织到一起，称为 **包（Package）**。
- 示例包：`java.util`（包含 `Scanner`、`Arrays`、`Random` 等工具类）。

2. 为什么需要导入模块（Import）？

- 即使某些模块属于 Java 标准库，但默认不会自动导入，必须通过 `import` 显式引入。
- 未导入时，编译会报错（"class not found"）。

3. 导入模块的语法

```java
import java.util.Scanner;
import java.util.Arrays;
```

4. 通配符导入（Wildcard Import）

- 通过 `import java.util.*;` 可以一次性导入整个 `java.util` 包的所有类。

```java
import java.util.*;

public class Test {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);  // Scanner 可以直接使用
        System.out.println(Arrays.toString(args));  // Arrays 也可以使用
    }
}
```

Listing 29: Wildcard Import 示例

通配符导入的优缺点

- **优点**：简化代码，避免逐个导入类。
- **缺点**：降低代码可读性，容易产生命名冲突，且不利于明确依赖关系。

5. 默认导入（Default Package）

- `java.lang` 包自动导入，无需手动 `import`。
- 包含常用类如：`System`、`Math`、`String`、`Exception` 等。

6. 随机数程序示例（Die Roll Program）

- 使用 `Math.random()`（无需导入）。
- 使用 `java.util.Random`（需要导入）。

使用 Random 类生成 1 到 6 的随机数

```java
import java.util.Random;

public class Roll {
    public static void main(String[] args) {
        Random rand = new Random();
        int roll = rand.nextInt(6) + 1; // 生成 1 到 6 之间的随机数
```

```
        System.out.println("You rolled a: " + roll);
    }
}
```

Listing 30: Roll.java 示例

解释：

- `nextInt(6)` 返回 0-5，`+1` 使范围变为 1-6。

2.2.9 排序例题

题目要求：编写一个程序，从命令行参数接收 3 个整数，按从大到小排序输出。**限制条件**：只能使用 `if` 语句，不使用数组、循环或排序函数。

程序完整代码：

```
public class Sort3Integers {
    public static void main(String[] args) {

        // 从命令行读取三个整数参数
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int z = Integer.parseInt(args[2]);

        // 保证 x >= y
        if (x < y) {
            int temp = x;
            x = y;
            y = temp;
        }

        // 保证 x >= z
        if (x < z) {
            int temp = x;
            x = z;
            z = temp;
        }

        // 此时 x 已经是最大值
```

```
        // 接着保证 y >= z
        if (y < z) {
            int temp = y;
            y = z;
            z = temp;
        }

        // 最终 x 是最大值，y 次之，z 最小
        // 按顺序输出
        System.out.println(x);
        System.out.println(y);
        System.out.println(z);
    }
}
```

代码详细讲解：

1. **读取输入**：
   - `args[]` 数组保存命令行输入的三个字符串。
   - 使用 `Integer.parseInt()` 方法将字符串转为整数。
   - 例如输入：`1 2 3`，结果是 `x = 1, y = 2, z = 3`。
2. **第一轮 if 判断（x 和 y 的比较）**：
   - 判断 `x < y` 是否成立。
   - 若成立，则交换 `x` 和 `y`，保证 `x` 是当前最大值。
3. **第二轮 if 判断（x 和 z 的比较）**：
   - 再判断 `x < z`。
   - 若成立，说明 `z` 更大，将 `x` 和 `z` 交换。
   - 结束后，`x` 一定是三个数中最大的。
4. **第三轮 if 判断（y 和 z 的比较）**：
   - 判断 `y < z`。
   - 若成立，交换 `y` 和 `z`，保证 `y` 比 `z` 大。

5. **输出排序结果**：

   - 此时 `x >= y >= z`。
   - 直接依次输出 `x，y，z` 即为从大到小的顺序。

输入输出样例：

```
$ javac Sort3Integers.java
$ java Sort3Integers 1 2 3
3
2
1
```

交换逻辑（核心算法）：

- 每次 if 判断都尝试把较大值交换到前面。
- 使用一个临时变量 `temp` 完成两个变量的交换。
- 三次比较最多需要三次交换，但保证正确性和简洁性。

算法时间复杂度：

- 时间复杂度 $O(1)$，因为只比较和交换固定的 3 个数。
- 没有循环结构，代码非常高效。

代码优化建议：

- 当前实现结构清晰，易于理解，不需要优化。
- 若输入数据变多，应该使用数组 + 循环或排序算法（如冒泡、选择、快速排序等）。

3 Week2

本周主要介绍 Java 的控制流与数据结构，包括：

- 控制流（Control Flow）
- 循环结构（while/do-while/for）
- 增强 for 循环（for-each）
- 静态方法（Static Methods）
- 数组（Arrays）
- 多维数组（Multidimensional Arrays）
- 字符串（Strings）
- 可变字符串（StringBuilder）

3.1 Lecture

3.1.1 Java 流程控制（Control Flow）

1. Java 流程控制概述

Java 提供了多种循环结构，用于控制程序的执行流程：

- **while** 循环
- **do-while** 循环
- **for** 循环
- **for-each** 循环

3.1.2 Java while 循环

1. while 语法规则：

```
while (condition) {
    doWork();  // 只要 condition 为 true，则执行 doWork()
}
```

Listing 31: while 循环示例

2. 执行逻辑:

- 先判断 ‘condition‘ 是否为 ‘true‘。
- 如果 ‘true‘，则执行循环体中的代码 ‘doWork()‘。
- 执行完后再次判断 ‘condition‘，如果仍为 ‘true‘，则继续循环。
- 直到 ‘condition‘ 变为 ‘false‘，循环终止。

3.1.3 Java do-while 循环

1. do-while 语法规则:

```
do {
    doWork();  // 至少执行一次
} while (condition);
```

Listing 32: do-while 循环示例

2. 特点:

- ‘do-while‘ 循环 ** 至少执行一次 **，因为 ‘doWork()‘ 在 ‘condition‘ 判断之前就执行了。
- ‘while‘ 语句会在每次循环结束后检查 ‘condition‘，决定是否继续循环。

3.1.4 Java for 循环

1. for 语法规则:

```
for (int i = 0; i < 10; i++) {
    doWork();
}
```

Listing 33: for 循环示例

2. for 语句的结构:

- ‘初始化部分 ‘：‘int i = 0;‘ 仅在循环开始时执行一次。
- ‘条件判断部分 ‘：‘i < 10‘，如果为 ‘true‘，执行循环体，否则跳出循环。
- ‘更新部分 ‘：‘i++‘，每次循环结束后执行，用于控制循环次数。

3. for 循环的执行顺序：

1. 执行 ‘初始化部分 ‘。
2. 判断 ‘条件部分 ‘，如果为 ‘false‘，结束循环。
3. 如果 ‘true‘，执行 ‘循环体 ‘。
4. 执行 ‘更新部分 ‘，回到步骤 2 继续判断 ‘条件部分 ‘。

3.1.5 Java for-each 循环

1. for-each 语法规则：

```
for (binding : collection) {
    doWork(binding);
}
```

Listing 34: for-each 循环示例

2. 关键概念解析：

- **binding**：绑定的变量，它在每次循环时存储 ‘collection‘ 中的当前元素。
- **collection**：可迭代的数据集合，如 ‘List‘、‘Set‘、‘Array‘，它的元素会依次赋值给 ‘binding‘。

3. 代码示例：

```
String[] strings = {"Java", "Python", "C++"};
for (String str : strings) {
    System.out.println(str);
}
```

Listing 35: 遍历字符串数组

4. for-each 的特点：

- 语法简洁，不需要手动管理 ‘index‘。
- 适用于 ‘Collection‘（如 ‘ArrayList‘、‘Set‘）和 ‘数组 ‘（如 ‘String[]‘）。
- ** 不能 ** 直接访问索引（index），如果需要索引值，应该使用 ‘for‘ 循环。

5. for-each 适用场景：

- 适用于 ** 遍历所有元素 **，不需要索引的情况。
- 不适用于 ** 需要修改集合元素 ** 或 ** 按索引访问 ** 的情况。

6. for-each 限制：

- 不能用来修改 ‘collection‘ 的元素，例如 ‘List.set(index, newValue)‘。
- 无法获取当前元素的索引（‘index‘），如 ‘i‘ 的值。

7. 解决 for-each 不能获取索引的问题：

```
String[] strings = {"Java", "Python", "C++"};
for (int i = 0; i < strings.length; i++) {
    System.out.println("Index␣" + i + ":␣" + strings[i]);
}
```

Listing 36: 使用 for 循环获取索引

8. 适用数据结构示例：

- `String[]`（数组）
- `ArrayList<String>`
- `List<String>`
- `Set<String>`
- `Deque<String>`

3.1.6 Java 静态方法（Static Methods）

1. static 方法的定义：

```
public static int addThree(int a, int b, int c) {
    return a + b + c;
}
```

Listing 37: 静态方法示例

2. 特点：

- ‘static‘ 关键字表示该方法属于 ‘类 ‘，而不是 ‘对象 ‘。
- 不需要创建对象即可调用该方法：‘ClassName.addThree(1,2,3);‘

3.1.7 Java 调用栈（Call Stack）

1. 调用栈的作用：

- Java 是基于栈的语言，每个方法调用都会被压入 ‘call-stack‘。
- 栈顶方法是当前正在执行的方法。
- 当方法执行结束后，它会被从 ‘call-stack‘ 中弹出。

2. 示例：

```java
public class CallStackExample {
    public static void main(String[] args) {
        methodA();
    }

    static void methodA() {
        methodB();
    }

    static void methodB() {
        System.out.println("Inside methodB");
    }
}
```

Listing 38: 调用栈示例

3. 执行过程：

- ‘main()‘ 被压入 ‘call-stack‘，执行 ‘methodA()‘。
- ‘methodA()‘ 被压入 ‘call-stack‘，执行 ‘methodB()‘。
- ‘methodB()‘ 执行 ‘println‘，然后结束并从 ‘call-stack‘ 弹出。
- ‘methodA()‘ 结束并弹出，‘main()‘ 结束，程序终止。

3.1.8 Java 数组（Arrays）

1. 数组的概念

数组是 Java 中一种连续存储相同类型数据的结构，初始化后会分配一块连续的内存，并返回数组的地址。

2. 数组的初始化：

```java
int[] numbers = new int[5];  // 创建一个长度为5的整数数组
int[] values = {1, 2, 3, 4}; // 直接初始化
int[] data = new int[] {10, 20, 30, 40}; // 显式指定数组元素
```

Listing 39: 数组初始化示例

3. 数组的存储特点:

- 数组在 ** 内存堆（Heap）** 中分配，并返回指向数组首地址的引用。
- 访问数组元素时，通过 ‘index‘ 来读取。
- ** 数组的长度是固定的 **，无法动态扩展。

4. 访问数组元素:

```java
int[] numbers = {5, 10, 15, 20};
System.out.println(numbers[0]);  // 输出 5
System.out.println(numbers[1]);  // 输出 10
```

Listing 40: 访问数组元素示例

5. 遍历数组:

```java
int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Listing 41: 使用 for 循环遍历数组

3.1.9 Java 引用类型与基本类型数组 (Reference and Primitive Type Arrays)

1. 数组初始化的一般规则:

- ** 基本整数类型 **（`byte`, `short`, `int`, `long`）的默认值是 **0**。
- ** 布尔（boolean）数组 ** 的默认值是 **false**。
- ** 浮点数类型 **（`float`, `double`）的默认值分别是 **0.0f** 和 **0.0d**。
- ** 字符数组（char[]）** 的默认值是 **\u0000**（空字符）。

- ** 引用类型数组 **（如 `String[]`）的默认值是 **null**。

2. 数组初始化示例：

```
public class ArrayInitialization {
    public static void main(String[] args) {
        int[] intArray = new int[5];        // 默认值 0
        boolean[] boolArray = new boolean[5]; // 默认值 false
        float[] floatArray = new float[5]; // 默认值 0.0f
        char[] charArray = new char[5];     // 默认值 '\u0000'
        String[] strArray = new String[5]; // 默认值 null

        System.out.println(intArray[0]);  // 输出 0
        System.out.println(boolArray[0]); // 输出 false
        System.out.println(floatArray[0]); // 输出 0.0
        System.out.println(charArray[0]);  // 输出 空字符
        System.out.println(strArray[0]);  // 输出 null
    }
}
```

Listing 42: 默认数组初始化示例

3. 引用类型数组与基本类型数组的区别：

- ** 基本类型数组 **（如 `int[]`）：存储实际数值。
- ** 引用类型数组 **（如 `String[]`）：存储对象的内存地址，默认值是 **null**，因为数组本身不会初始化对象。

4. 访问未初始化的引用数组元素：

```
String[] names = new String[3]; // 所有元素均为 null
System.out.println(names[0]);   // 输出 null
```

Listing 43: 引用类型数组默认值示例

5. 总结：

- ** 基本类型数组 ** 会自动初始化默认值，如 ‘0‘、‘false‘、‘\u0000‘。
- ** 引用类型数组 ** 默认初始化为 ‘null‘，必须手动赋值后才能使用。
- ** 可以使用 ‘Arrays.fill()‘ 方法手动填充数组 **。

3.1.10 Java 多维数组（Multidimensional Arrays）

1. 多维数组的定义：

- ** 多维数组 ** 是数组的数组，常见的是 ** 二维数组 **（矩阵）。
- ** 定义格式 **：‘数据类型 [][] 数组名 = new 数据类型 [行数][列数];‘

```
int[][] matrix = new int[2][3];  // 2 行 3 列
```

Listing 44: 二维数组示例

2. 初始化多维数组：

- ** 动态初始化 **：先声明数组，再赋值。
- ** 静态初始化 **：声明时直接赋值，省略 ‘new‘ 关键字。

```
// 动态初始化
int[][] array = new int[2][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;

// 静态初始化
int[][] array2 = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Listing 45: 多维数组初始化示例

3. 遍历多维数组：

- 使用 **嵌套 ‘for‘ 循环**遍历行列元素。
- 外层 ‘for‘ 遍历 ** 行（Row）**，‘array.length‘ 获取行数。
- 内层 ‘for‘ 遍历 ** 列（Column）**，‘array[i].length‘ 获取当前行的列数。

```
int[][] array = {
    {1, 2, 3},
```

```
    {4, 5, 6}
};

for (int i = 0; i < array.length; i++) { // 遍历行
    for (int j = 0; j < array[i].length; j++) { // 遍历列
        System.out.print(array[i][j] + "␣");
    }
    System.out.println(); // 换行
}
```

Listing 46: 遍历多维数组示例

4. 复杂二维数组遍历:

```
int[][] array = new int[5][5];

// 赋值 0~24
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        array[i][j] = i * 5 + j;
    }
}

// 遍历并打印
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j] + "␣");
    }
    System.out.println();
}
```

Listing 47: 遍历 5×5 二维数组

5. 代码输出示例（假设 ‘array‘ 被赋值 0-24）:

```
> java ArrayOutput
0  1  2  3  4
5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

```
<program end>
```

Listing 48: 二维数组输出示例

6. 重要概念:

- ** 第一维代表行 **，‘array.length‘ 获取行数。
- ** 第二维代表列 **，‘array[i].length‘ 获取该行的列数。
- ** 不规则数组（Jagged Array）**：不同行的列数不同。

7. 不规则数组示例（Jagged Array）:

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[2]; // 第一行有 2 列
jaggedArray[1] = new int[4]; // 第二行有 4 列
jaggedArray[2] = new int[3]; // 第三行有 3 列

for(int i = 0; i < jaggedArray.length; i++) {
    for(int j = 0; j < jaggedArray[i].length; j++) {
        System.out.print(jaggedArray[i][j] + "␣");
    }
    System.out.println();
}
```

Listing 49: 不规则数组示例

3.1.11 Java 字符串（Strings）

1. 字符串的定义:

```
String greeting = "Hello,␣World!";
```

Listing 50: 字符串示例

2. 字符串的不可变性:

- 在 Java 中，‘String‘ 是 ** 不可变（immutable）** 的，一旦创建后无法修改。
- 字符串拼接 ‘+‘ 操作会生成新的 ‘String‘ 对象，而不是修改原字符串。

3. 字符串拼接:

```
String message = "Hello";
message += ",␣World!";
System.out.println(message);  // 输出 "Hello, World!"
```

Listing 51: 字符串拼接示例

4. String Pool 机制:

- Java 维护一个 ** 字符串池（String Pool）**，所有字符串字面量都会存入其中。
- ** 相同的字符串字面量 ** 在内存中只存储一份，多个变量共享相同的引用。

5. 字符串内存管理示例:

```
String cat1 = "Meow";
String cat2 = "Meow";
System.out.println(cat1 == cat2);  // true
```

Listing 52: String Pool 示例

解释:

- ‘”Meow”‘ 作为 ** 字符串字面量 ** 存储在 ‘String Pool‘ 中。
- ‘cat1‘ 和 ‘cat2‘ 都指向 ‘String Pool‘ 中的同一个 ‘”Meow”‘ 对象。
- 因此，‘cat1 == cat2‘ 为 ‘true‘，因为它们的引用地址相同。

6. ‘new‘ 关键字的影响:

```
String cat1 = "Meow";
String cat2 = new String("Meow");
System.out.println(cat1 == cat2);  // false
System.out.println(cat1.equals(cat2));  // true
```

Listing 53: 使用 new 关键字创建字符串

解释:

- ‘cat1‘ 指向 ‘String Pool‘ 中的 ‘”Meow”‘。

- ‘cat2‘ 通过 ‘new String(”Meow”)‘ ** 创建了一个新对象 **，存储在堆（Heap）中。
- ‘cat1 == cat2‘ 为 ‘false‘，因为它们的引用地址不同。
- ‘cat1.equals(cat2)‘ 为 ‘true‘，因为 ‘equals()‘ 方法比较的是字符串的 ** 内容 **。

7. 推荐的字符串比较方式：

- ** 使用 ‘==‘ 比较地址 **（仅适用于判断是否是 ‘String Pool‘ 中的同一对象）。
- ** 使用 ‘.equals()‘ 比较内容 **（最安全的字符串比较方式）。

```
String str1 = "Hello";
String str2 = new String("Hello");

System.out.println(str1 == str2);       // false（不同内存地址）
System.out.println(str1.equals(str2));  // true（相同内容）
```

Listing 54: 推荐的字符串比较方式

8. ‘intern()‘ 方法：

- ‘intern()‘ 方法用于手动将字符串加入 ‘String Pool‘。

```
String str1 = new String("Java");
String str2 = str1.intern(); // str2 现在指向 String Pool
String str3 = "Java";

System.out.println(str2 == str3); // true（都指向 String Pool）
```

Listing 55: 使用 intern() 方法

3.1.12 Java StringBuilder

1. 为什么需要 StringBuilder ?：

- 在 Java 中，‘String‘ 是不可变的（Immutable）。

- 每次 ‘String‘ 拼接 ‘+‘ 都会创建新的 ‘String‘ 对象，导致内存开销大。
- ‘StringBuilder‘ 通过 ** 可变数组（char array）** 优化字符串拼接，提高性能。

2. StringBuilder 的定义：

```
StringBuilder sb = new StringBuilder();
```

Listing 56: StringBuilder 初始化

- ‘StringBuilder‘ 是可变的字符串序列，适用于大量字符串拼接操作。
- ‘StringBuilder‘ 维护一个 ** 内部 ‘char[]‘ 数组 **，动态扩展空间以容纳新的字符。

3. StringBuilder 常用方法：

```
StringBuilder sb = new StringBuilder("Hello");
sb.append("␣World");  // 追加字符串
sb.insert(5, ",");    // 插入字符
sb.replace(6, 11, "Java"); // 替换子字符串
sb.delete(5, 6);      // 删除字符
System.out.println(sb.toString());  // 输出 "HelloJava"
```

Listing 57: StringBuilder 常用操作

4. String vs StringBuilder：

特性	String	StringBuilder
是否可变	不可变	可变
内存占用	高（每次修改都创建新对象）	低（使用 ‘char array‘ 动态扩展）
线程安全	线程安全（有 ‘String Pool‘）	线程不安全（适用于单线程）
适用场景	小规模字符串操作	频繁修改字符串的情况

5. StringBuilder 内部机制：

- ‘StringBuilder‘ 维护一个 **char 数组 **，当字符串长度超过容量时，数组会 ** 动态扩展 **。
- ‘append()‘ 方法不会创建新的对象，而是直接修改 ‘char array‘。

6. StringBuilder 扩展机制:

```
StringBuilder sb = new StringBuilder(10); // 初始容量为10
sb.append("HelloWorld"); // 不触发扩容
sb.append("Java");  // 触发扩容，容量增大
```

Listing 58: StringBuilder 预分配机制

7. ‘StringBuffer‘ vs ‘StringBuilder‘:

- ‘StringBuffer‘ 是 ‘StringBuilder‘ 的 ** 线程安全 ** 版本，适用于多线程环境。
- ‘StringBuilder‘ 更快，但不适用于并发环境。

3.1.13 Java 数组 vs 字符串

1. 数组和字符串的主要区别:

- 数组长度是 ** 固定的 **，而字符串可以变化（但 ‘String‘ 是不可变的）。
- ‘char[]‘ 可以手动修改元素，而 ‘String‘ 不能。

2. 将字符串转换为字符数组:

```
String text = "Java";
char[] chars = text.toCharArray();
```

Listing 59: 字符串转字符数组

3. 结论:

- 需要可变字符串时，推荐使用 ‘StringBuilder‘。
- 处理单个字符时，可以使用 ‘char[]‘。

3.2 Tutorial

3.2.1 Java 中的 while 与 do-while 循环

1. 基本区别

- `while`：先判断条件，后执行循环体。
- `do-while`：先执行循环体，后判断条件。
- **do-while 至少执行一次**，而 while 可能一次也不执行。

2. do-while 示例

```
boolean b = false;
do {
    System.out.println("This is false!");
} while (b);
```

执行结果：输出 `"This is false!"`，因为 do 块先执行，再判断 `b`。

3. 错误的 while 循环示例

```
int norm = 10;
while(norm) {  // 错误！条件必须是 boolean
    System.out.println("yes");
    norm = norm - 1;
}
```

为什么错误？ Java 要求 while 条件是 `boolean` 类型，不能直接使用 `int`。

4. 正确写法

```
int norm = 10;
while(norm != 0) {
    System.out.println("yes");
    norm--;
}
```

说明：

- `norm != 0` 是 boolean 表达式。
- 只有在 norm 不等于 0 时继续执行循环体。

3.2.2 统计游戏

程序功能：编写一个方法，统计字符串中所有的小写元音字母（a, e, i, o, u）个数。

完整代码：

```
public class CountWords {

    // 定义静态方法，统计元音个数
    static int countVowels(String s) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == 'a' || s.charAt(i) == 'e' ||
                s.charAt(i) == 'i' || s.charAt(i) == 'o' ||
                s.charAt(i) == 'u') {
                count++;
            }
        }
        return count;
    }

    // 主函数，执行程序
    public static void main(String[] args) {
        String s = "astronaut";
        int count = countVowels(s);
        System.out.println("Number of vowels: " + count); // 4
    }
}
```

代码讲解：

1. `countVowels(String s)` 方法遍历字符串，查找每个字符是否为元音。
2. 使用 `s.charAt(i)` 获取第 i 个字符。
3. 满足条件时，计数器 `count++`。
4. 最终返回元音字母的总个数。

输入样例:

```
astronaut
```

元音分析:

- 'a' -> 是元音
- 'o' -> 是元音
- 'a' -> 是元音
- 'u' -> 是元音

输出结果:

```
Number of vowels: 4
```

3.2.3 数组的 length 属性和 clone() 方法

属性 vs 方法区别:

- `arr.length` 是数组的 **属性**，表示数组的长度
- `arr.clone()` 是数组的 **方法**，返回数组的浅拷贝

示例代码:

```
import java.util.Arrays;

public class D {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4};

        // 获取数组长度
        int len = arr.length;

        // 浅拷贝数组
        int[] arr_copy = arr.clone();

```

```
        // 修改原数组
        arr[3] = 5;

        System.out.println(len); // 输出 4
        System.out.println(Arrays.toString(arr));       // 输出
            [1, 2, 3, 5]
        System.out.println(Arrays.toString(arr_copy)); // 输出
            [1, 2, 3, 4]
    }
}
```

注意事项:

- length 没有括号，访问属性。
- clone() 必须带括号，是方法，返回数组副本。
- 浅拷贝：如果数组元素是对象，拷贝的只是引用；如果是基本类型，直接复制值。

3.2.4 数组元素查找与计数

代码示例

```
class Operations {

    static boolean contains(int[] a, int element) {
        for (int num : a) {
            if (num == element) {
                return true;
            }
        }
        return false;
    }

    static int count(int[] a, int element) {
        int i = 0;
        for (int num : a) {
            if (num == element) {
                i++;
```

```
            }
        }
        return i;
    }

    public static void main(String[] args) {
        int[] array = {1, 1, 5, 6, 5, 3, 8, 1, 9, 2, 8};

        System.out.println("contains(array, 5): " + contains(
            array, 5));
        System.out.println("contains(array, 10): " + contains(
            array, 10));

        System.out.println("count(array, 1): " + count(array,
            1));
        System.out.println("count(array, 5): " + count(array,
            5));
    }
}
```

运行流程

1. **调用 contains(array, 5)** 遍历到第 3 个元素，找到 5，返回 true。
2. **调用 contains(array, 10)** 遍历完整个数组，未找到 10，返回 false。
3. **调用 count(array, 1)** 遍历数组，找到 1 共 3 次，返回 3。
4. **调用 count(array, 5)** 遍历数组，找到 5 共 2 次，返回 2。

控制台输出

```
contains(array, 5): true
contains(array, 10): false
count(array, 1): 3
count(array, 5): 2
```

3.2.5 计数重复元素

题目要求统计数组中出现重复（至少出现 2 次）元素的不同元素个数。

示例输入

```
int[] dups = {1, 1, 5, 6, 5, 3, 8, 1, 9, 2, 8};
```

分析元素 1 出现 3 次，5 出现 2 次，8 出现 2 次。所以输出 3。

完整代码

```
class Operations2 {
    static int countDuplicates(int[] a) {
        int count = 0;
        int[] num = new int[20];

        for (int i = 0; i < a.length; i++) {
            num[a[i]]++;
        }

        for (int i = 0; i < num.length; i++) {
            if (num[i] >= 2) {
                count++;
            }
        }

        return count;
    }

    public static void main(String[] args) {
        int[] dups = {1, 1, 5, 6, 5, 3, 8, 1, 9, 2, 8};
        System.out.println(countDuplicates(dups)); // 输出：3
    }
}
```

运行过程

1. 遍历输入数组，统计次数:
   - num[1] = 3
   - num[5] = 2
   - num[8] = 2
2. 统计 num 数组中次数 ≥ 2 的元素个数，共 3 个。

3. 最终输出: 3

输出结果

```
3
```

3.2.6 Java Map 与 HashMap 基础

在 Java 中，**Map** 是一个存储 **键值对（key-value）** 的接口，最常用的实现类是 **HashMap**。Map 可以通过键快速查找值，键不能重复，而值可以重复。

- `Map<K, V>` 是接口，`HashMap<K, V>` 是其常用实现类。
- 泛型 `K` 表示键类型，`V` 表示值类型。
- **注意：** Java 泛型不支持基本数据类型（`int`、`char`），需使用包装类（`Integer`、`Character`）。

1. Map 的创建和初始化

创建一个空的 HashMap，键和值都为字符串类型：

```
Map<String, String> store = new HashMap<>();
```

Listing 60: 创建 Map 和 HashMap 实例

2. 基本操作

1. **put(K key, V value)**：向 Map 添加键值对。
2. **get(K key)**：通过键获取对应的值。
3. **keySet()**：返回所有键的集合，可以用于遍历。

示例代码：

```
Map<String, String> store = new HashMap<>();
store.put("key1", "value1");
store.put("key2", "value2");

```

```
System.out.println(store.get("key1")); // 输出: value1
System.out.println(store.get("key3")); // 输出: null (不存在)

for (String key : store.keySet()) {
    System.out.println("键: " + key + " -> 值: " + store.get(
        key));
}
```

Listing 61: Map 基本操作演示

3. Map 中 get 返回 null 的情况

- 当调用 get() 方法，若 key 不存在于 Map 中，会返回 null。
- 可以通过判断 if (value == null) 进行判空处理。

4. 字符统计案例：使用 Map 统计字符串中每个字符出现的次数

```
import java.util.*;

public class CharCounter {
    public static void main(String[] args) {
        String uut = "pcasdouh5nlwak[ as[ q3\\n =qw3c *]";
        Map<Character, Integer> store = new HashMap<>();

        for (int i = 0; i < uut.length(); i++) {
            char c = uut.charAt(i);
            Integer count = store.get(c);

            if (count == null) {
                count = 0;
            }

            store.put(c, count + 1);
        }

        System.out.println("  字符统计: " + store);
    }
}
```

Listing 62: 统计字符串中字母出现次数

运行结果展示

```
字符统计: { =1, *=1, a=4, c=2, d=1, 3=2, h=1, ...}
```

5. 重要概念总结

- **HashMap** 实现了 `Map` 接口，并重写了 `toString()` 方法，因此可以直接通过 `System.out.println()` 打印输出。
- `Integer` 和 `Character` 是包装类，代替 `int` 和 `char`。
- `Map` 是接口，可以通过不同实现（如 `HashMap`、`TreeMap`）来存储和操作键值对。

6. 扩展讨论

- 为什么 `HashMap` 打印无需 `Arrays.toString()`?

  `HashMap 重写了 toString() 方法，自动格式化输出。`

- 如果访问一个不存在的 key,会返回 `null`,要注意判空,避免 `NullPointerException`。

4 Week3

4.1 Lecture

- **Part A**
  - Classes（类）
  - Object Attributes（对象属性）
  - Instance Methods（实例方法）
  - UML Class Diagram（UML 类图）
- **Part B**

- `this` 关键字和非静态上下文
- 静态（static）和非静态（non-static）上下文混合使用
- 文本输入与输出（Text I/O）

4.1.1 Classes: Where Reference Types Come From

Primitive vs Reference Types

- Java 数据类型分为两大类：**Primitive Type（基本类型）**和 **Reference Type（引用类型）**。
- **Primitive Type**: 包括 `int`、`double`、`char` 等。
- **Reference Type**: 来源于 **Class**。任何类的对象都是 Reference Type。

Reference Types are Classes

- 参考类型（Reference Types）来自类的实例化。
- 类是数据结构化和行为封装的基本单元。

作用

- 我们在之前的程序中已经频繁使用了内置类（如 `Scanner`、`String` 等），但从未定义过自己的类。
- 大多数编程语言都有一种机制用于代码重用，而在 Java 中，**Class** 是结构化数据与方法复用的主要手段。

4.1.2 What's a Class

定义

> "*A class defines a type or kind of object. It is a blueprint for defining the objects. All objects of the same class have the same kinds of data and the same behaviours. When the program is run, each object can act alone or interact with other objects to accomplish the program' s purpose.*"

简化理解

- 类是**蓝图（Blueprint）**、**模板（Template）**或 **概念（Concept）**。

已有经验

- 所有 Java 程序中都使用了类，但我们此前未自行实例化（instantiate）自定义类的对象。
- 已使用的内置类包括：
  - `Scanner`
  - `String`
  - `StringBuilder`

4.1.3 Objects

对象是类的实例

- 对象（Object）是特定类的一个实例（Instance）。
- 通过 `new` 关键字分配内存并调用构造器（Constructor）实例化对象。

对象创建示例

```
Car toyota = new Car("fuel", "white");
Car mazda  = new Car("Hybrid", "blue");
Car tesla  = new Car("electric", "white");
```

Listing 63: 创建 Car 对象

4.1.4 Class Definition

基本类结构

```
public class Cupcake {
    public boolean delicious;
    public String name;
}
```

Listing 64: Cupcake 类定义

实例化对象

```java
Cupcake c = new Cupcake();
```

Listing 65: 实例化 Cupcake 对象

- `Cupcake c`：声明了一个引用类型变量。
- `new Cupcake()`：分配内存空间并调用构造方法。

4.1.5 Constructors（构造方法）

默认构造器

- Java 每个类都有构造方法，即使没有显式定义，编译器也会提供一个默认构造方法。

自定义构造方法

```java
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake() {
        // NO OP
    }
}
```

Listing 66: 空操作（NO OP）构造方法

构造方法没有返回类型

- 构造方法类似方法，但没有返回值。
- 其作用是初始化对象。

4.1.6 Providing Default Values in Constructors

无参构造器

```java
public Cupcake() {
    delicious = true;
    name = "Chocolate␣Cupcake";
```

```
}
```

Listing 67: 无参构造器初始化属性

有参构造器

```
public Cupcake(boolean isTasty) {
    delicious = isTasty;
    name = "Chocolate␣Cupcake";
}
```

Listing 68: 带参数的构造器

扩展说明

- 可以通过传参给构造器，创建不同状态的对象。
- 例如在实例化时传递不同的布尔值控制 `delicious` 属性。

4.1.7 小结

- 类是对象的蓝图。
- 构造方法负责对象的初始化。
- `new` 关键字用于实例化对象，分配内存和调用构造方法。

4.1.8 Cupcake 类定义与构造器

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate␣Cupcake";
    }
}
```

Listing 69: Cupcake 类与带参构造方法

说明:

- 该类有两个公共属性 delicious 和 name。
- 构造方法通过参数 isTasty 来初始化 delicious，并将 name 设置为”Chocolate Cupcake”。

4.1.9 对象的实例化

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);

System.out.println(mine.delicious);     // 输出 true
System.out.println(toShare.delicious);  // 输出 false
```

Listing 70: 对象实例化与属性访问

说明:

- mine 是一个 Cupcake 对象,构造时传递 true,表示它是美味的(delicious = true)。
- toShare 是另一个 Cupcake 对象，传递 false，表示它不是美味的(delicious = false)。

4.1.10 对象属性访问

通过对象名和点（.）操作符可以访问对象属性，例如:

```
System.out.println(mine.delicious);
```

知识点回顾:

- 和 Scanner、String 类似，我们可以通过 object.attribute 来访问属性。
- 也可以通过方法（如果存在）来访问对象行为。
- 可以为 Cupcake 类添加更多构造器和方法（如 eat()、describe()等)。
- 通过创建多个对象，理解面向对象编程中“对象是类的实例”的概念。

4.1.11 Instance Methods（实例方法）

定义

- 实例方法（Instance Methods）是面向对象编程（OOP）中定义在类（Class）内部的方法。
- 它们操作实例对象的属性（Attributes），通过对象实例来调用。
- 与 `static` 方法不同，实例方法必须依赖对象才能使用。

基本语法

```
[final] return_type methodName([parameters]) {
    // 方法体
}
```

关键点

- 不再使用 `static` 修饰符。
- 实例方法只能在创建对象后通过对象引用访问。

实例说明

扩展 `Cupcake` 类，增加实例方法：

```
1  public class Cupcake {
2      public boolean delicious;
3      private String name;
4
5      public Cupcake(boolean isTasty, String cupcakeName) {
6          delicious = isTasty;
7          name = cupcakeName;
8      }
9
10     // Setter 方法：修改 name 属性
11     public void setName(String n) {
12         name = n;
13     }
14
15     // Getter 方法：获取 name 属性
16     public String getName() {
```

```
        return name;
    }
}
```

Listing 71: Cupcake 类，添加实例方法

访问控制

- `public` 关键字允许属性/方法在类外部访问。
- `private` 关键字限制属性/方法只能在类内部访问。

4.1.12 对象方法调用示例

创建对象并调用实例方法：

```
Cupcake mine = new Cupcake(true, "My Cupcake!");
Cupcake toShare = new Cupcake(false, "Everyone's Cupcake");

mine.setName("My Cupcake, Don't touch!");
System.out.println(mine.getName());
```

Listing 72: 调用 setName 和 getName 方法

4.1.13 扩展功能：eat 方法

我们添加一个 `eat` 方法，让对象状态改变：

```
public class Cupcake {
    public boolean delicious;
    private String name;
    private boolean eaten; // 新增属性：是否被吃掉

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
        eaten = false;
    }

    public void setName(String n) { name = n; }

```

```
14      public String getName() { return name; }
15
16      // 新增实例方法：吃掉蛋糕
17      public void eat() {
18          eaten = true;
19      }
20  }
```

Listing 73: 添加 eat 方法

4.1.14 eat 方法扩展：状态反馈

继续扩展 eat 方法，加入用户反馈机制：

```
1  public void eat() {
2      if (!eaten) {
3          System.out.println("That was nice!");
4          eaten = true;
5      } else {
6          System.out.println("There is nothing left to eat!");
7      }
8  }
```

Listing 74: eat 方法增加状态检查和输出

执行示例

```
1  Cupcake cupcake = new Cupcake(true, "Tasty Cupcake");
2
3  cupcake.eat(); // 输出 That was nice!
4  cupcake.eat(); // 输出 There is nothing left to eat!
```

Listing 75: eat 方法使用示例

小结

- 实例方法通过修改实例的状态实现对象行为。
- 访问控制修饰符（private、public）确保封装性。
- getter 和 setter 方法用于访问和修改私有属性。

4.1.15 UML（统一建模语言）

概念

- **UML**，全称 **Unified Modelling Language**，是一种用于设计应用程序和系统的可视化语言。
- 通过图形化方式展示系统的结构、行为以及组件间的关系。
- 本课程重点关注 **UML Class Diagrams（类图）**。

UML Class Diagram（类图）

- 类图帮助程序员在实现类之前设计系统结构。
- 通过建模，能够在不编写实际代码的前提下，了解系统结构及其组成部分。

组成要素

1. 类名（Class Name）
2. 属性（Attributes）：类的状态信息。
3. 方法（Methods）：类的行为功能。

可见性符号

- + 表示 `public`，公共可见性，成员对所有类可访问。
- - 表示 `private`，私有可见性，成员仅对类自身可访问。
- （可选）# 表示 `protected`，受保护，仅对子类和同包类可访问。

4.1.16 Lamp 类 UML 示例

Java 源代码实现

```
1 public class Lamp {
2     private int lumens;
3     private boolean on;
4     private float height;
5
6     public void switchOn() {
```

```
        // 实现细节
    }

    public boolean isOn() {
        // 实现细节
        return on;
    }

    public void changeBulb(int lumens) {
        this.lumens = lumens;
    }

    public int lumens() {
        return lumens;
    }

    public float getHeight() {
        return height;
    }
}
```

Listing 76: Lamp 类源码

UML 类图展示

```
+--------------------------------+
|            Lamp                |  <-- Class Name
+--------------------------------+
| - lumens : int                 |  <-- Attribute (private)
| - on : boolean                 |  <-- Attribute (private)
| - height : float               |  <-- Attribute (private)
+--------------------------------+
| + switchOn() : void            |  <-- Method (public)
| + isOn() : boolean             |
| + changeBulb(lumens : int) : void |
| + lumens() : int               |
| + getHeight() : float          |
+--------------------------------+
```

解析

- **Lamp** 是类名。
- **- lumens : int** 表示私有属性 `lumens`，类型为 `int`。
- **+ switchOn() : void** 表示公共方法 `switchOn`，返回值为 `void`。
- 类图中属性和方法的显示顺序为：属性在上，方法在下。

设计 UML 类图时注意

- 确保每个类只关注一种职责（单一职责原则 SRP）。
- 明确可见性（+ / -）以确保封装性（Encapsulation）。
- 提前在类图中定义方法签名和参数类型，为实现做准备。

UML 类图与 Java 代码的映射关系

- 类图是 Java 代码的设计蓝图。
- UML 使团队在没有实际编码的情况下也能讨论系统结构。
- 类图设计好后，可直接映射为 Java 类的实现。

4.1.17 this 关键字

this 的基本作用

- **this 关键字**用于引用当前对象的实例。
- 在 **实例方法**中，可以使用 this 来消除参数名与属性名的歧义。
- this 不能在 static 方法中使用，只能在实例方法中使用。
- 还可以用来调用当前类的其他构造方法（将在第 5 周详细讲解）。

this 的实际用途

- 解决参数与成员变量命名冲突。
- 在对象内部传递对当前实例的引用。

示例说明：参数名冲突时的歧义

```java
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String sender, String receiver, String
        address, String contents) {
        sender = sender; //   歧义，编译器无法确定左边和右边是
            哪个
    }
}
```

Listing 77: 存在歧义的问题

解决歧义的常规方法

```java
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String s, String r, String a, String c) {
        sender = s;
        receiver = r;
        address = a;
        contents = c;
    }
}
```

Listing 78: 重命名参数解决歧义

问题：可读性降低

- 上述做法虽然消除了歧义，但牺牲了代码的可读性。
- 参数名应保持语义清晰，而不是使用 cryptic letters（晦涩字母）。

推荐使用 this 关键字解决歧义

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String sender, String receiver, String
        address, String contents) {
        this.sender = sender;
        this.receiver = receiver;
        this.address = address;
        this.contents = contents;
    }
}
```

Listing 79: 使用 this 消除歧义

总结

- this 关键字提高了代码的清晰度和可读性。
- 建议在构造方法和实例方法中养成使用 this 的良好习惯。

4.1.18 Keyword

概念

- `this` 是 Java 中的引用，指向当前对象实例。
- 主要作用：
  - 消除构造器或方法参数与实例变量之间的命名歧义。
  - 在实例方法中引用当前对象。
  - **不能**在 `static` 方法中使用 `this`，因为 `static` 不属于实例。

使用场景

1. 参数与成员变量同名，避免歧义：

```
    public class Postcard {
        String sender;
        String receiver;
        String address;
        String contents;

        public Postcard(String sender, String receiver,
            String address, String contents) {
            this.sender = sender;
            this.receiver = receiver;
            this.address = address;
            this.contents = contents;
        }
    }
```

2. 类似 Python 的 `self`，指向当前实例。

3. 构造方法调用（将在第 5 周详细讲解）。

4.1.19 Instance Methods

概念

- 实例方法依赖于对象实例，不能通过类名直接调用。
- 实例方法可以访问类的实例变量和实例方法。

语法

```
[final] return_type methodName ([parameters]) {
    // 方法体
}
```

示例代码

```
public class Cupcake {
    public boolean delicious;
    private String name;
    private boolean eaten;

```

```
    public Cupcake(boolean isTasty, String cupcakeName) {
        this.delicious = isTasty;
        this.name = cupcakeName;
        this.eaten = false;
    }

    public void setName(String n) { name = n; }
    public String getName() { return name; }

    public void eat() {
        if (!eaten) {
            System.out.println("That was nice!");
            eaten = true;
        }
    }
}
```

4.1.20 Static Methods vs Instance Methods

静态方法

- 不属于任何实例对象，通过类名调用。
- 不能访问实例变量或实例方法（没有对象实例）。

错误示例：静态方法访问实例变量

```
public class Postcard {
    String sender;
    String receiver;
    boolean received;

    public static boolean inTransit() {
        return !received; // 错误！ received 是实例变量
    }
}
```

正确用法：传入对象实例

```
public class Postcard {
    String sender;
```

```
    String receiver;
    boolean received;

    public boolean inTransit() {
        return !received;
    }

    public static boolean hasArrived(Postcard p) {
        return !p.inTransit();
    }
}
```

重点对比

内容	实例方法 (Instance Method)	静态方法 (Static Method)
是否需要对象?	是	否
是否可以访问实例变量?	是	否
典型调用方式	object.method()	Class.method()
使用场景	操作对象状态	无需依赖对象、工具类方法

课堂代码讲解总结

- `this` 消除歧义，类似 Python 中的 `self`。
- 实例方法操作具体对象，`this` 隐式传递。
- 静态方法属于类，不能访问实例变量和方法，除非显式传入对象。

4.1.21 实例方法与静态方法结合使用

代码示例

```
public class Postcard {
    String sender;
    String receiver;
    boolean received;
    // ...snip...

    public boolean alreadyArrived() {
        return hasArrived(this);
```

```
    }

    public static boolean hasArrived(Postcard p) {
        if (!p.inTransit()) {
            return true;
        } else {
            return false;
        }
    }
}
```

Listing 80: Postcard 类中实例方法与静态方法结合

代码解析

- **alreadyArrived()** 是一个 **实例方法**。
  - 该方法调用了静态方法 `hasArrived()`。
  - 使用 `this` 关键字将当前实例对象传递给静态方法。
- **hasArrived(Postcard p)** 是一个 **静态方法**。
  - 该方法接收一个 `Postcard` 类型的对象。
  - 通过 `p.inTransit()` 判断邮件是否还在传递途中。
  - 如果邮件不在传递途中（`!p.inTransit()`），返回 `true`，表示已经到达。

关键点说明

- **this 关键字**：
  - `this` 用于引用当前对象实例。
  - 只能在 **实例方法**和 **构造方法**中使用，不能在 **静态方法**中使用。
- **实例方法调用静态方法**：
  - `alreadyArrived()` 是实例方法，但可以调用 `hasArrived()` 静态方法。
  - 通过传递 `this`，静态方法获得当前实例的引用。

结论

- 这是合法且常见的 Java 设计模式。
- 静态方法 `hasArrived()` 不依赖于对象本身，但可以通过参数引用特定实例。
- 在实例方法中传递 `this` 给静态方法是完全可行的。

补充说明

- 这与直接调用 `Postcard.hasArrived(this)` 等价。
- 类内部可以省略类名直接调用 `hasArrived(this)`。

4.1.22 Using Scanner

We can now use **Scanner** to read files. As the name implies it **Scan**'s for input and provides functionality to read it.

```
import java.io.File;
import java.util.Scanner;

public class FileHandle {
    public static void main(String[] args) {
        File f = new File("README.txt");
        Scanner scan = new Scanner(f);
    }
}
```

- We have **File** object that will abstract represent the file stored at a **Path**.
- **Scanner** accepts a file as an argument and is able to read contents there.

Compilation Error

```
> javac FileHandle.java
FileHandle.java:7: error: unreported exception
    FileNotFoundException;
```

```
Must be caught or declared to be thrown
    Scanner scan = new Scanner(f);
1 error
```

- As with most **IO operations** we will be required to perform some exception handling.

Handle FileNotFoundException

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

public class FileHandle {
    public static void main(String[] args) {
        File f = new File("README.txt");
        try {
            Scanner scan = new Scanner(f);
        } catch (FileNotFoundException e) {
            System.out.println("File not found!");
        }
    }
}
```

- If the file does not exist we are unable to read from it. This allows the programmer to have a branch for both: a state where we can read data and one without reading data.
- Java forces us to provide some checks to ensure we are handling certain exception cases correctly.

4.1.23 How is Reading Performed?

Reading any kind of file is analogous to working with **contiguous memory**.

- Let's say we have the following file called `README.txt` which contains the following contents:

`Today is great!`

This can be represented with the following array:

T	o	d	a	y		i	s		g	r	e	a	t!

Code Example

```
File f = new File("README.txt");
Scanner scan = new Scanner(f);

scan.next(); // Today
scan.next(); // is
scan.next(); // great!
```

- **Scanner** itself doesn't support reading character by character. Reasoning behind this is because the idea of a character depends on how it is encoded.
- Executing `next()` will move the cursor to the next space (or whatever token we want to separate words by).

4.1.24 Cursor Movement During Reading

- Each time `next()` is called, the cursor moves forward to the next token.
- This simulates moving through a contiguous block of memory, where each word is processed sequentially.

Example Continued

```
File f = new File("README.txt");
Scanner scan = new Scanner(f);

scan.next(); // Today
scan.next(); // is
scan.next(); // great!
```

- The cursor has moved once `next()` has been called.
- Each subsequent call moves the cursor to the next token.

4.1.25 Writing Text Data

As discussed prior, **Scanner** only performs reading an object. So how about writing?

PrintWriter

- **PrintWriter** allows for printing formatted representations of objects to a text-output stream.

```
1  import java.io.File;
2  import java.io.PrintWriter;
3  import java.io.FileNotFoundException;
4
5  public class FileHandle {
6      public static void main(String[] args) {
7          File f = new File("README.txt");
8          try {
9              PrintWriter writer = new PrintWriter(f);
10         } catch (FileNotFoundException e) {
11             e.printStackTrace();
12         }
13     }
14 }
```

- We have a class that allows writing formatted data.

PrintWriter Example With Output

```
1  import java.io.File;
2  import java.io.PrintWriter;
3  import java.io.FileNotFoundException;
4
5  public class FileHandle {
6      public static void main(String[] args) {
7          File f = new File("README.txt");
8          try {
9              PrintWriter writer = new PrintWriter(f);
10             writer.println(1.0);
11             writer.println(120);
12             writer.println("My␣String!");
```

```
            writer.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

- **PrintWriter** has methods very similar to `System.out`. That is no coincidence!
- This will write output `1.0`, `120` and `"My String!"` to the file **README.txt**.

5 Week 4

第四周我们继续深入学习 Java 编程的相关知识，主要内容包括：

- **Binary Input/Output（二进制输入输出）**
- **Garbage Collector（垃圾回收机制）**
- **ArrayList（动态数组）**

5.1 Lecture

5.1.1 Binary Input/Output（二进制输入输出）

什么是二进制输入输出？在上一周的学习中，我们了解了如何通过 **Scanner（扫描器）**读取文本文件。本周我们将学习如何读写二进制文件，即通过非文本方式存储和读取数据。

应用场景

- 与设备交互（例如手柄、MIDI 设备）
- 修改可执行文件
- 图片编码
- 自定义文件格式
- 汽车软件开发

- 视频音频流
- 网络通信

Java 中的二进制流类在 Java 中，常用的二进制读写类有：

- **DataInputStream（数据输入流）**
- **DataOutputStream（数据输出流）**

这些类常配合 **FileInputStream** 和 **FileOutputStream** 使用，利用 `readInt()`、`writeInt()` 等方法进行操作。

二进制文件写入示例

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryWriter {
    public static void main(String[] args) {
        try {
            FileOutputStream f = new FileOutputStream("newfile.
                bin");
            DataOutputStream output = new DataOutputStream(f);

            output.writeInt(50); // 写入一个整数
            output.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Listing 81: BinaryWriter 示例

二进制文件读取示例

```java
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryReader {
    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream("newfile.
                bin");
            DataInputStream input = new DataInputStream(f);

            int n = input.readInt(); // 读取一个整数
            System.out.println(n);

            input.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Listing 82: BinaryReader 示例

5.1.2 Garbage Collector（垃圾回收机制）

垃圾回收简介在 Java 中，**Garbage Collector（垃圾回收器）**会自动释放没有引用的对象占用的内存空间。

堆内存（Heap Memory）使用 `new` 关键字创建的对象会存储在堆内存中。垃圾回收器会在对象无引用后，自动清理内存。

工作机制当某个对象不再被引用（所有引用超出作用域），垃圾回收器会标记它为垃圾，然后在空闲时间执行清理。

示例

- `String x;`

- `String y, z;`
- `int[] array;`
- `int[][] matrix;`
- `matrix[0];`

当这些对象超出作用域或没有引用时，系统将回收它们。

Java 中 [] 与 <> 的区别

基本概念

- `[]` 表示数组（Array），用于存储固定长度、类型相同的数据集合。
- `<>` 表示泛型（Generics），提供类型参数化机制，常用于集合类。

核心区别

区别点	数组 `[]`	泛型 `<>`
作用	存储固定数量、类型一致的数据元素。	定义类型参数，增强类型安全。
数据结构	固定长度线性结构。	灵活的集合类（List, Map, Set）。
类型检查	编译时已知，元素类型固定。	编译时类型检查，避免类型转换错误。
灵活性	长度固定，无法动态增加或删除。	灵活，集合可以动态增删元素。
效率	高效，占用内存连续，读写性能高。	相对较低，依赖封装机制。
常用语法	`int[] arr = new int[5];`	`List<Integer> list = new ArrayList<>();`
使用场景	已知元素数量，性能要求高的场合。	数据规模变化大，需类型抽象的场合。

代码示例

1. 数组 `[]` 使用示例

```java
int[] numbers = new int[5];
numbers[0] = 10;
numbers[1] = 20;

System.out.println(numbers[0]);  // 输出 10
```

2. 泛型 <> 使用示例

```java
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");

System.out.println(names.get(0));  // 输出 Alice
```

综合举例：二维数组与泛型集合的对比

```java
// 数组中的二维数组
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6}
};

// 泛型中的嵌套列表
List<List<Integer>> list = new ArrayList<>();
list.add(Arrays.asList(1, 2, 3));
list.add(Arrays.asList(4, 5, 6));
```

总结

- `[]`（数组）：适合存储已知数量、类型相同的元素，结构简单、效率高。
- `<>`（泛型）：适合存储数据类型灵活、需要动态操作的集合，增强类型安全，扩展性强。

5.1.3 ArrayList（动态数组）

简介 **ArrayList** 是 Java 集合框架中的动态数组实现，可以自动调整大小。

基本语法

```java
import java.util.ArrayList;

ArrayList<String> list = new ArrayList<String>();
```

Listing 83: ArrayList 声明

基本操作

```java
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();

        list.add("First␣String!");
        list.add("Second␣String!");
        list.add("Woof!!");

        list.remove(1); // 删除索引 1 处元素
        list.set(1, "Meow"); // 修改索引 1 处元素

        System.out.println(list.get(0)); // 输出 "First String
            !"
        System.out.println(list.get(1)); // 输出 "Meow"
    }
}
```

Listing 84: ArrayList 基本操作

ArrayList 如何扩容

1. 初始容量数组。
2. 调用 `add()` 方法时，依次填充。
3. 数组填满后，创建新的更大数组，并复制旧数据。
4. 继续在新数组中添加元素。

简化说明

- 初始数组：`nil nil nil nil`

- 添加元素后：`2 8 3 4`
- 数组已满，扩容为双倍长度：`2 8 3 4 nil nil nil nil`
- 再次添加元素后：`2 8 3 4 42 nil nil nil`

5.1.4 DynamicArray 动态扩容机制

背景引入

在 Java 中，`ArrayList` 是一个常用的动态数组，它能够在存储空间不足时自动扩容。我们实现了简化版的 **DynamicArray**，用于理解其内部扩容机制。

DynamicArray 工作流程

1. 初始时，`DynamicArray` 内部维护一个长度为 8 的数组，所有元素初始化为 `null`。

   `[ nil, nil, nil, nil, nil, nil, nil, nil ]`

2. 每次调用 `add()` 方法，会在数组末尾添加元素。

3. 随着元素逐步插入，数组状态如下：

   - 添加 2：
     `[ 2, nil, nil, nil, nil, nil, nil, nil ]`
   - 添加 8：
     `[ 2, 8, nil, nil, nil, nil, nil, nil ]`
   - 添加 3、4、90、12、45、32：
     `[ 2, 8, 3, 4, 90, 12, 45, 32 ]`
   - 第八个位置填满，此时容量为 8，继续添加元素 42，触发扩容。

4. 扩容过程如下：

   (a) 新建一个长度为 16 的数组。
   `[ nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil ]`

(b) 将原数组所有元素复制到新数组前 8 个位置。

```
[ 2, 8, 3, 4, 90, 12, 45, 32, nil, nil, nil, nil, nil,
nil, nil, nil ]
```

(c) 添加元素 42 到第九个位置。

```
[ 2, 8, 3, 4, 90, 12, 45, 32, 42, nil, nil, nil, nil, nil,
nil, nil ]
```

5. 继续添加新元素时，将沿用扩容后的数组，直到再次填满，进入下一轮扩容（翻倍增长）。

DynamicArray 扩容机制总结

- 初始容量固定（通常为 8）。
- 每次容量满时，**扩容为原容量的两倍**，即 $8 \to 16 \to 32 \to 64 \ldots$。
- 扩容时进行：
  1. 新建更大数组
  2. 复制原数据
  3. 添加新元素
  4. 老数组失效，等待垃圾回收
- 扩容存在性能开销（复制操作），但不频繁，均摊后效率较高。

核心实现代码

```
// 添加元素
public void add(int element) {
    if (size == array.length) {
        resize();
    }
    array[size++] = element;
}

// 扩容操作
private void resize() {
```

```
    int newCapacity = array.length * 2;
    int[] newArray = new int[newCapacity];

    // 复制旧元素
    for (int i = 0; i < size; i++) {
        newArray[i] = array[i];
    }

    array = newArray;
}
```

扩容性能说明

- 每次扩容成本：$O(n)$，因为需要复制现有所有元素。
- 每次插入均摊成本：$O(1)$，大部分时间不扩容，扩容发生的次数少。

5.1.5 LinkedList（链表）

概念 LinkedList（链表）是 Java 提供的集合类之一，它不同于 **ArrayList**（动态数组）。链表由一系列节点（Node）组成，每个节点包含数据部分和指向下一个节点的引用。

- 适用于频繁插入和删除元素的场景。
- 插入、删除的时间复杂度为 $O(1)$。
- 随机访问的时间复杂度为 $O(n)$。

基本用法

```
import java.util.LinkedList;

public class Example {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<String>();

        list.add("First␣Node");
        list.add("Second␣Node");
        list.addFirst("Head␣Node"); // 添加到头部
```

```
10          list.addLast("Tail Node");  // 添加到尾部
11
12          System.out.println(list.getFirst()); // 访问第一个节点
13          System.out.println(list.getLast());  // 访问最后一个节
                点
14
15          list.removeFirst(); // 删除第一个节点
16          list.removeLast();  // 删除最后一个节点
17      }
18  }
```

Listing 85: LinkedList 示例

总结

- `addFirst()`：在链表头添加元素。
- `addLast()`：在链表尾添加元素。
- `removeFirst()`：删除头元素。
- `removeLast()`：删除尾元素。

5.1.6 ArrayList 和 LinkedList 的对比及 LinkedList 的实现机制

在 Java 中，**ArrayList** 和 **LinkedList** 是两种常用的 **List（列表）**实现类。二者虽然都实现了 **List 接口**，但底层数据结构和使用场景却有明显差异。

ArrayList 底层是基于数组（**Array**）实现的。它内部维护了一个动态增长的对象数组，初始容量默认为 10，当元素数量超过容量时，数组会自动扩容（通常增长为原来的 1.5 倍）。ArrayList 允许通过下标（**index**）快速随机访问元素，时间复杂度为 $\mathcal{O}(1)$。但是，在数组中插入或删除元素时，由于需要移动元素，会导致操作复杂度变高，平均时间复杂度为 $\mathcal{O}(n)$。

LinkedList 则是基于双向链表（**Doubly Linked List**）实现的。每个节点（**Node**）包含三部分内容：元素（**item**）、指向前一个节点的引用（**prev**）、指向下一个节点的引用（**next**）。因此，LinkedList 插入和删除节点时，只需要修改前后节点的引用，时间复杂度为 $\mathcal{O}(1)$，不需要像 ArrayList

一样移动元素。但 LinkedList 无法通过下标直接访问元素，只能从头或尾进行线性遍历，时间复杂度为 $\mathcal{O}(n)$。

ArrayList 和 LinkedList 的性能对比如下表所示：

操作	ArrayList	LinkedList
随机访问（get/set）	$\mathcal{O}(1)$	$\mathcal{O}(n)$
在尾部插入元素（add）	$\mathcal{O}(1)$（均摊）	$\mathcal{O}(1)$
在中间插入/删除元素	$\mathcal{O}(n)$	$\mathcal{O}(1)$（已知节点）
空间消耗	较低（仅存储元素）	较高（存储元素 + 两个引用）

LinkedList 的底层实现机制采用了典型的双向链表结构。内部使用 **Node** 类表示链表节点，核心结构如下：

```
private static class Node<E> {
    E item;         // 当前节点的元素值
    Node<E> next;   // 指向下一个节点
    Node<E> prev;   // 指向前一个节点

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

添加元素时，LinkedList 会通过 `linkLast` 方法向尾部插入新节点，具体过程如下：

1. 创建新节点，`prev` 指向当前 `last` 节点，`next` 为空；
2. 当前 `last` 节点的 `next` 指向新节点；
3. 更新 `last` 引用为新节点；
4. 如果链表为空，则 `first` 和 `last` 同时指向新节点。

```
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
```

```
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

删除元素时，通过断开节点之间的引用完成，删除头节点的代码如下：

```
E unlinkFirst(Node<E> f) {
    final E element = f.item;
    final Node<E> next = f.next;
    f.item = null;
    f.next = null; // help GC
    first = next;
    if (next == null)
        last = null;
    else
        next.prev = null;
    size--;
    modCount++;
    return element;
}
```

在遍历元素时，LinkedList 无法像 ArrayList 通过下标快速访问元素。对于 `get(index)` 方法，LinkedList 会根据索引大小判断是从 `first` 节点正向查找还是从 `last` 节点反向查找，以提高查找效率（但时间复杂度依然为 $\mathcal{O}(n)$）。

总结：如果应用场景中频繁进行插入和删除操作，推荐使用 LinkedList；如果更多是随机访问元素，推荐使用 ArrayList。

5.1.7 Maps（映射）和 Sets（集合）

概念

- **Map（映射）**存储键值对（key-value pairs），通过 key 获取 value。
- **Set（集合）**存储唯一元素，不能重复。

HashMap（哈希映射）

```
import java.util.HashMap;

public class Example {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<String,
            Integer>();

        map.put("Apple", 1);
        map.put("Banana", 2);
        map.put("Cherry", 3);

        System.out.println(map.get("Banana")); // 输出 2

        map.remove("Cherry"); // 删除键 "Cherry"

        System.out.println(map.containsKey("Apple")); // 判断是
            否包含 "Apple"
        System.out.println(map.containsValue(2));     // 判断是
            否包含值 2
    }
}
```

Listing 86: HashMap 示例

HashSet（哈希集合）

```
import java.util.HashSet;

public class Example {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<String>();

        set.add("Red");
        set.add("Green");
        set.add("Blue");

        System.out.println(set.contains("Green")); // 输出 true

        set.remove("Red"); // 删除 "Red"
```

```
14
15          for (String color : set) {
16              System.out.println(color); // 遍历输出剩余元素
17          }
18      }
19  }
```

Listing 87: HashSet 示例

总结

- HashMap 基于键值对存储，快速查找、插入和删除。
- HashSet 只存储唯一元素，无顺序。

5.1.8 Map 和 Set 的区别示例

Map（映射）和 **Set（集合）**都是 Java 中常用的数据结构，但它们在数据的存储方式和处理规则上有明显区别。

1. Map 处理重复 key 的规则

Map 是以 **键（key）-值（value）**的形式存储数据。相同的 key 只能出现一次，但其对应的 value 可以被更新。当插入新的相同 key 时，会覆盖原有 key 对应的 value。

```
1   import java.util.HashMap;
2
3   public class MapDemo {
4       public static void main(String[] args) {
5           HashMap<String, String> map = new HashMap<>();
6
7           // 第一次插入 key 为 "apple", value 为 "red"
8           map.put("apple", "red");
9
10          // 第二次插入相同 key, 为 "apple", value 改为 "green"
11          map.put("apple", "green");
12
13          // 输出 key 为 "apple" 的值, 只会显示最新的 value
14          System.out.println(map.get("apple"));  // 输出: green
15
```

```
        // 打印整个 Map
        System.out.println(map);  // 输出：{apple=green}
    }
}
```

Listing 88: Map 中 key 不重复，value 可覆盖

2. Set 处理重复元素的规则

Set 集合不允许存储重复元素。如果尝试添加相同的元素，Set 会忽略新的插入，而不会产生错误或覆盖。

```
import java.util.HashSet;

public class SetDemo {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();

        // 插入元素 "apple"
        set.add("apple");

        // 再次插入相同的元素 "apple"
        set.add("apple");

        // 插入不同的元素 "banana"
        set.add("banana");

        // 输出 Set 集合
        System.out.println(set);  // 输出：[apple, banana]
    }
}
```

Listing 89: Set 不允许重复元素

3. Map 和 Set 的结构对比

- Map 以 key 作为唯一标识，每个 key 对应一个 value，key 不允许重复，value 可以重复。
- Set 只存储单个元素，不含 key，元素不允许重复，内部使用 HashMap 存储 key（值），value 固定为常量对象。

4. 其他操作对比示例

```java
import java.util.HashMap;
import java.util.HashSet;

public class CompareDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("apple", 1);
        map.put("banana", 2);

        // 是否包含某 key
        System.out.println(map.containsKey("apple")); // true

        // 是否包含某 value
        System.out.println(map.containsValue(2)); // true

        // 移除元素
        map.remove("apple");
        System.out.println(map); // {banana=2}

        // 遍历 Map
        for (String key : map.keySet()) {
            System.out.println(key + "->" + map.get(key));
        }

        // ---------

        HashSet<String> set = new HashSet<>();
        set.add("apple");
        set.add("banana");

        // 是否包含某元素
        System.out.println(set.contains("apple")); // true

        // 删除元素
        set.remove("apple");
        System.out.println(set); // [banana]

        // 遍历 Set
```

```
        for (String item : set) {
            System.out.println(item);
        }
    }
}
```

Listing 90: Map 和 Set 常用方法对比

5. 总结

- **Map** 以 key 唯一索引，支持 value 更新和覆盖。
- **Set** 保证元素唯一，不允许重复插入。
- Map 常用于 **键值对存储**，如字典、索引等。
- Set 常用于 **元素去重**、数学集合操作等。

5.1.9 Checked 和 Unchecked Operations（受检与非受检操作）

Checked Operations（受检操作）

- 编译时检查异常（Checked Exception）。
- 必须通过 try-catch 或 throws 显式处理。

常见 Checked Exception

- `IOException`
- `FileNotFoundException`
- `SQLException`

示例：Checked Exception

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Example {
    public static void main(String[] args) {
        try {
```

```
            File file = new File("data.txt");
            Scanner scanner = new Scanner(file);

            while(scanner.hasNextLine()) {
                System.out.println(scanner.nextLine());
            }

            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found!");
        }
    }
}
```

Listing 91: 受检异常示例

Unchecked Operations（非受检操作）

- 运行时异常（Runtime Exception）。
- 不强制捕获或声明处理。

常见 Unchecked Exception

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`

示例：Unchecked Exception

```
public class Example {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};

        // 会抛出 ArrayIndexOutOfBoundsException
        System.out.println(numbers[5]);
    }
}
```

Listing 92: 非受检异常示例

总结

- 受检异常在编译阶段强制处理，增强程序健壮性。
- 非受检异常在运行时抛出，通常由于程序逻辑错误。

6 week5

6.1 Lecture

6.1.1 继承的基本概念

面向对象编程中的 ** 继承 ** (Inheritance) 是一种允许新的类 (子类) 重用已有类 (父类) 属性和方法的机制。这不仅提高了代码 ** 重用性 **, 还使得类型之间形成层次结构。在继承关系中，上层的类称为 ** 父类 ** (superclass)，下层的类称为 ** 子类 ** (subclass)。通过继承，子类自动拥有父类的功能，并可在此基础上扩展新的行为。

继承带来的主要好处包括:

- ** 属性和方法的重用 **: 子类无需重复定义父类中已有的属性和方法，直接继承提高开发效率。
- ** 定义子类的新方法 **: 子类可以在父类基础上增加自身特有的方法，实现功能扩展。
- ** 覆盖父类方法 **: 子类可以提供与父类同名但实现不同的方法 (override)，以定制或改进继承来的功能。
- ** 类型信息的延伸 **: 子类对象可被视为父类类型使用 (遵循里氏替换原则)，形成多态性的基础。

例如，`Animal` 类可以作为一般动物的父类，而 `Cat` (猫) 和 `Dog` (狗) 可以继承自 `Animal` 作为其子类。这样我们说 Cat *is-a* Animal，Dog *is-a* Animal，即猫和狗都是动物的一种。需要注意，Java 中每个类最多只能继承一个直接父类 (Java 不支持类的多重继承)。

父类：Bottle

```java
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    // 默认构造方法 (No-argument constructor)
    public Bottle() {
    }

    public double volume() {
        return height * width * depth;
    }
}
```

Listing 93: 父类 Bottle

说明：

- Bottle 类是父类 (Parent Class)，包含多个 protected 属性，允许子类 (Subclass) 访问。
- 构造方法 public Bottle() 是一个无参构造方法 (No-argument Constructor)，不接受任何参数。
- 方法 volume() 返回使用对象属性计算出的体积。

子类：GlassBottle

```java
public class GlassBottle extends Bottle {

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
```

```
        return shattered;
    }
}
```

Listing 94: 子类 GlassBottle

说明:

- GlassBottle 类继承了父类 Bottle 中的所有 protected 和 public 属性与方法。
- 属性 shattered 是 private，因此父类 Bottle 无法访问它。
- shatter() 和 isBroken() 是子类独有的方法。

构造方法继承示例

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();   // 创建子类对象
    System.out.println(b.isBroken());    // 输出: false
    System.out.println(b.name);          // 输出: null (因为
        name 没有被初始化)
}
```

Listing 95: 主类示例

输出:

```
false
null
```

关键点总结

- 如果父类提供了默认构造方法 (Default Constructor)，子类会自动调用它。
- protected 修饰的属性与方法会被子类继承，而 private 则不会。
- 构造方法不会被继承，但在创建对象时会被调用。
- 如果父类只有参数化构造方法 (Parameterized Constructor)，子类必须显式调用父类构造方法，使用 super()。

section 参数化构造方法 (Parameterized Constructor)

父类：Bottle

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    // 参数化构造方法
    public Bottle(String name, double width, double height,
        double depth) {
        this.name = name;
        this.width = width;
        this.height = height;
        this.depth = depth;
        this.litresFilled = 0.0;
    }

    public double volume() {
        return height * width * depth;
    }
}
```

Listing 96: 父类 Bottle 的参数化构造方法

子类构造方法的调用

错误示例：未调用父类构造方法

```
public class GlassBottle extends Bottle {
    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

```

```
12      public boolean isBroken() {
13          return shattered;
14      }
15  }
```

Listing 97: 子类 GlassBottle 未调用父类构造方法

问题: 父类 Bottle 定义了一个参数化构造方法，因此默认构造方法不会自动存在。尝试使用 new GlassBottle() 时会引发编译错误。

6.1.2 使用 super 调用父类构造方法

```
1   public class GlassBottle extends Bottle {
2       public GlassBottle() {
3           super("", 0.0, 0.0, 0.0);
4           this.name = "Glass␣Bottle";
5       }
6
7       private boolean shattered = false;
8
9       public void shatter() {
10          shattered = true;
11      }
12
13      public boolean isBroken() {
14          return shattered;
15      }
16  }
```

Listing 98: 子类 GlassBottle 使用 super

解释: 使用 super() 明确调用父类的构造方法，并提供所有参数，否则会引发编译错误。

匹配父类构造方法

```
1   public class GlassBottle extends Bottle {
2       public GlassBottle(String name, double width, double height
            , double depth) {
3           super(name, width, height, depth);
4       }
```

```
5
6       private boolean shattered = false;
7
8       public void shatter() {
9           shattered = true;
10      }
11
12      public boolean isBroken() {
13          return shattered;
14      }
15  }
```

Listing 99: 子类 GlassBottle 提供匹配的参数化构造方法

总结:

- 如果父类定义了参数化构造方法而没有提供默认构造方法，子类必须显式调用父类构造方法。
- 使用 super() 是调用父类构造方法的唯一方式，且必须是子类构造方法的第一行。
- 如果父类的构造方法需要参数，子类必须提供相应的参数进行调用。

6.1.3 练习题与答案解析

第一题：父类与子类的构造方法匹配

题目:

给定如下两个类:

```
1   public class Fruit {
2       protected String name;
3
4       public Fruit(String name) {
5           this.name = name;
6       }
7   }
```

Listing 100: 父类 Fruit

```java
public class Apple extends Fruit {
    public Apple() {
        super("Apple");
        System.out.println("Apple␣is␣created.");
    }
}
```

Listing 101: 子类 Apple

```java
public static void main(String[] args) {
    Apple myApple = new Apple();
    System.out.println(myApple.name);
}
```

Listing 102: 测试代码

答案：

输出：

```
Apple is created.
Apple
```

第二题：多个构造方法的调用

题目：

```java
public class Building {
    protected int floors;

    public Building() {
        this.floors = 1;
    }

    public Building(int floors) {
        this.floors = floors;
    }
}

public class House extends Building {
    public House() {
        super(2);
```

```
    }

    public House(int floors) {
        super(floors);
    }
}
```

Listing 103: 父类 Building 和子类 House

```
public static void main(String[] args) {
    House myHouse1 = new House();
    House myHouse2 = new House(3);

    System.out.println(myHouse1.floors);  // 输出：2
    System.out.println(myHouse2.floors);  // 输出：3
}
```

Listing 104: 测试代码

答案：

2

3

第三题：使用 super() 调用父类构造方法

题目：

```
public class Vehicle {
    protected String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }
}

public class Car extends Vehicle {
    protected int speed;

    public Car(String brand, int speed) {
        super(brand);
        this.speed = speed;
```

```
    }

    public Car() {
        super("Unknown");
        this.speed = 0;
    }
}
```

Listing 105: 父类 Vehicle 和子类 Car

```
public static void main(String[] args) {
    Car myCar1 = new Car("Toyota", 120);
    Car myCar2 = new Car();

    System.out.println(myCar1.brand + "␣-␣" + myCar1.speed);
        // 输出: Toyota - 120
    System.out.println(myCar2.brand + "␣-␣" + myCar2.speed);
        // 输出: Unknown - 0
}
```

Listing 106: 测试代码

答案:

```
Toyota - 120
Unknown - 0
```

总结

- 子类必须显式调用父类的构造方法，如果父类只提供参数化构造方法。
- super() 必须是子类构造方法中的第一行代码。
- 如果父类的构造方法需要参数，子类必须提供相应的参数进行调用。

继承关系：Is-a 与 Has-a
Is-a 关系 (Extension)

- 指的是“某个类是另一个类的一种”，通常是通过继承实现的。
- 例如：

  - Cat 是一个 Animal (Cat extends Animal)。
  - Dog 是一个 Animal (Dog extends Animal)。
- 在 Java 中，使用 extends 关键字来建立 Is-a 关系。
- 子类从父类继承属性和方法，可以重写或扩展它们。

Has-a 关系 (Composition)

- 指的是 “某个类拥有另一个类”，通常是通过类的属性来实现的。
- 例如：
  - Car 拥有一个 Engine (Car has-a Engine)。
  - Library 拥有多个 Book (Library has-a Book)。
- 在 Java 中，通过组合 (composition) 的方式实现。
- Has-a 关系不涉及继承，只是通过组合实现功能。

6.1.4 UML 通用化 (Generalization)

- 在 UML 图中：
  - 父类 (Superclass) 通常位于上方。
  - 子类 (Subclass) 通过一个空心箭头指向父类。
- 例子：Bottle, PlasticBottle, 和 GlassBottle
- 通用化的意义在于：不同的子类可以继承同一个父类的属性和方法。

父类与子类的规则总结

- 父类并不知道自己的子类存在。
- 子类不能通过父类的构造方法进行实例化。
- Private 修饰的属性和方法不能被子类访问或继承。
- 继承应该符合逻辑的 Is-a 关系。
- Java 中，一个类只能继承自一个父类 (单继承 Single Inheritance)。

- 在 UML 图中，继承关系称为 **Generalization** (通用化)。

总结:

- `Public` 可以在任何地方访问。
- `Protected` 可以在子类中访问，即使它们在不同的包中。
- `Private` 只能在声明它的类内部访问，无法被继承或访问。
- `Default` (包私有) 只能在同一个包中访问。

6.1.5 构造方法重载 (Constructor Overloading)

在 Java 中，我们可以定义多个构造方法，只要它们的参数列表不同。这就是构造方法重载。这样做的目的是为了提供不同的初始化方式，增加类的灵活性和兼容性。

示例代码 Person 类)

```java
public class Person {
    private static int DEFAULT_AGE = 21;
    private String name;
    private int age;

    // 构造方法 1: 无参构造方法
    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    // 构造方法 2: 接受一个参数的构造方法
    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

    // 构造方法 3: 接受两个参数的构造方法
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
}
```

测试代码

```
public static void main(String[] args) {
    Person p1 = new Person();                // 使用无参构造方法
    Person p2 = new Person("Janice");        // 使用单参构造方法
    Person p3 = new Person("Dave", 32);      // 使用双参构造方法
}
```

使用 this() 关键字优化构造方法 (Constructor Chaining)

在重载构造方法时，重复的代码可能会显得冗余。我们可以通过 `this()` 调用来优化。

优化前的代码

```
public class Person {
    private static int DEFAULT_AGE = 21;
    private String name;
    private int age;

    public Person() {
        this.name = "Jeff";
        this.age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

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

优化后的代码

```
public class Person {
    private static int DEFAULT_AGE = 21;
    private String name;
```

```
    private int age;

    // 构造方法 1: 无参构造方法
    public Person() {
        this("Jeff", DEFAULT_AGE); // 调用构造方法 3
    }

    // 构造方法 2: 接受一个参数的构造方法
    public Person(String name) {
        this(name, DEFAULT_AGE);  // 调用构造方法 3
    }

    // 构造方法 3: 接受两个参数的构造方法
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

优化后的好处

- **减少代码重复：**多个构造方法共享相同的初始化逻辑时，可以通过 `this()` 调用来重用已有构造方法的逻辑。
- **代码更加简洁、易于维护：**所有构造方法最终调用了最全面的构造方法 (`Person(String name, int age)`)。
- **构造方法链 (Constructor Chaining)：**Java 允许在一个构造方法中调用另一个构造方法，但必须是第一条语句。

6.2 文字总结

6.2.1 封装的访问控制

** 封装 **（Encapsulation）强调将数据和操作隐藏在类的内部，仅通过限定的接口与外界交互。Java 提供了多种访问控制修饰符来实现封装，包括 `public`（公共）、`protected`（受保护）、默认（包级私有）和 `private`（私有）。通过合理使用这些修饰符，可以控制类成员对外的可见性，从而实现信息隐藏。

这里重点介绍 protected 修饰符。它的行为类似于 private，即限制类外访问，但允许 ** 子类 ** 访问。具体来说：

- ** 本类内部 **：protected 成员可以访问（与 private 相同，仅对自身可见）。
- ** 子类 **：protected 成员对子类可见（即使子类在不同包中，也可以访问）。
- ** 同一包内的其他类 **：protected 成员可见（在包内部，protected 与缺省包级权限效果相同）。
- ** 非子类的外部包 **：protected 成员不可见（只有继承关系或同包关系才能访问）。

另外，如果不指定修饰符，则采用包级访问权限（default），即仅允许在同一包内访问，包外和子类（若在不同包）均不能访问。

通过受保护的访问控制，父类可以将自己的部分内部数据开放给子类使用，但对无关的外部类仍保持隐藏。例如，在后续的继承代码示例中，Bottle 类将属性声明为 protected，这使得其子类 GlassBottle 能够直接访问这些属性。由此可见，封装和继承相结合，在保证内部实现细节不被泄露的同时，又允许子类对父类的内部状态进行合理的利用。

6.2.2 继承的编程实现

在 Java 中使用关键字 extends 来声明继承关系。其基本语法形式为：class 子类名 extends 父类名。通过 extends，子类将继承父类的所有 **public** 和 **protected** 成员。子类可以直接使用这些继承来的属性和方法，也可以增加新的属性/方法，或重写父类的方法。

下面通过一个示例来演示继承的实际编程效果。代码定义了一个父类 Bottle 和一个子类 GlassBottle。GlassBottle 继承了 Bottle，并添加了新的功能：

```
1 public class Bottle {
2     protected String name;
3     protected double width;
4     protected double height;
5     protected double depth;
```

```
6       protected double litresFilled;
7
8       public double volume() {
9           return height * width * depth;
10      }
11  }
12
13  public class GlassBottle extends Bottle {
14      protected String name;
15      protected double width;
16      protected double height;
17      protected double depth;
18      protected double litresFilled;
19      private boolean shattered = false;
20
21      public void shatter() {
22          System.out.println("We␣lost␣" + litresFilled + "␣Litres
                ");
23          litresFilled = 0;
24          shattered = true;
25      }
26
27      public boolean isBroken() {
28          return shattered;
29      }
30  }
```

Listing 107: Bottle 类与 GlassBottle 类的继承示例

上述代码中，`GlassBottle` 通过 `extends Bottle` 继承了父类 `Bottle`。因为 Bottle 将属性定义为 `protected`，GlassBottle 子类可以直接访问这些属性。例如，GlassBottle 的 `shatter()` 方法中使用了继承自 Bottle 的 `litresFilled` 属性。** 注意：** 在 GlassBottle 中重复声明了 `name`、`width` 等属性，这并非继承所必需——实际上这样做会隐藏父类的同名属性，一般不推荐。这么处理只是为了强调 GlassBottle 拥有这些继承的属性。除此之外，GlassBottle 还新增了一个私有属性 `shattered` 以及 `shatter()` 和 `isBroken()` 两个方法。GlassBottle 同时自动继承了 Bottle 的 `volume()` 方法（虽然未在 GlassBottle 内显式定义），因此 GlassBottle 实例也具备计算

体积的功能。

6.2.3 is-a 关系与 UML 建模

当我们说“B 类继承自 A 类”时，也可表述为“B 是 A 的一种”，即 B 与 A 存在“is-a”关系。继承关系本质上就是一种 **is-a 关系 **：子类是父类的特殊化。比如，前面的例子中 Cat 类和 Dog 类都是 Animal 类的一种，它们继承了 Animal 的属性和行为。

在 UML 类图中，我们用图形来表达继承的 is-a 关系。通常的表示法是：用一条带空心三角箭头的直线从子类指向父类，箭头指向父类一侧，表示“继承自”。例如，我们可以用文字描述 UML 图中的继承关系：`GlassBottle` 是 `Bottle` 的子类，继承了 `Bottle` 所有受保护和公共的属性与方法。同样地，`Cat` 是 `Animal` 的子类，`Dog` 也是 `Animal` 的子类。通过这样的描述，我们明确了类之间的 is-a 关系，即子类从父类继承特性。

6.2.4 方法重载的规则

** 方法重载 **（Method Overloading）指在同一个类中，可以定义多个同名的方法，但参数列表（参数的类型、数量或顺序）不同。重载允许我们针对不同的参数情况使用相同的方法名，提高代码的可读性和灵活性。

例如，我们可以定义两个名称相同的 `add` 方法：一个用于相加两个整数，另一个用于相加三个整数：

```
public int add(int a, int b) {
    return a + b;
}

public int add(int a, int b, int c) {
    return a + b + c;
}
```

Listing 108: `add` 方法的重载示例

上面的代码在同一个类中声明了两个 `add` 方法：第一个接受两个整数参数，第二个接受三个整数参数。这就是方法重载——调用 `add(1, 2)` 时会匹配执行有两个参数的版本，而调用 `add(1, 2, 3)` 则会执行有三个参数的版本。Java 编译器会根据传入参数的数量和类型 ** 在编译期 ** 自动决定调用哪个重载方法。

需要注意，Java 判断两个方法是否构成重载，只考虑方法的 ** 签名 **（方法名和参数列表），返回类型不属于方法签名的一部分。因此，不能仅通过改变返回类型来声明同名方法。例如，下面的两个方法由于参数列表完全相同，并不能构成重载：

```
// 错误示例：以下两个方法仅返回类型不同，签名重复
float[] crossProduct(float[] a, float[] b) { ... }
int[] crossProduct(float[] a, float[] b) { ... }
```

Listing 109: 无效的重载示例（仅返回类型不同）

上例中试图定义 `crossProduct` 方法的两个版本，但它们的参数列表都是两个 `float[]`，只有返回类型分别是 `float[]` 和 `int[]`。由于返回类型不计入方法签名，这样的重复定义将导致编译错误（方法重复定义）。

另外，当多个重载方法的参数类型存在继承或兼容关系时，可能出现调用歧义的情况。例如，假设有两个重载方法 `crossProduct(int[] a, int[] b)` 和 `crossProduct(float[] a, float[] b)`。如果我们直接调用 `crossProduct(null, null)`，由于 `null` 可以转换为任意引用类型，编译器将无法确定调用哪一个方法，因而报错。下面的代码演示了这一情况以及解决办法：

```
public int[] crossProduct(int[] a, int[] b) { ... }
public int[] crossProduct(float[] a, float[] b) { ... }

public static void main(String[] args) {
    // 调用重载方法：
    int[] result = crossProduct(null, null); // 编译错误：调用不明确

    // 通过强制类型转换可以明确调用的方法版本：
    int[] r1 = crossProduct((int[]) null, (int[]) null);      // 调用 crossProduct(int[], int[])
    int[] r2 = crossProduct((float[]) null, (float[]) null); // 调用 crossProduct(float[], float[])
}
```

Listing 110: 重载方法调用的歧义示例

在上面的代码清单中，对 `crossProduct(null, null)` 的调用无法通过编译，因为编译器不知道该匹配 `crossProduct(int[], int[])` 还是

`crossProduct(float[], float[])` ——两个重载版本都符合 `null` 参数。编译器会报类似 “reference to `crossProduct` is ambiguous”（引用不明确）的错误。通过如上所示对 `null` 显式地进行类型转换，我们分别调用了特定的重载版本，成功消除了歧义。

值得一提的是，方法重载的分派是在编译阶段完成的（静态绑定）：编译器根据传入参数的静态类型选择合适的方法实现。这一点与 “方法覆盖”（override，动态绑定）的机制不同。

6.2.5 构造函数重载

除了普通方法，** 构造函数 ** 也可以重载。在一个类中，我们可以提供多个构造函数，只要它们的参数列表不同。这样可以方便地根据不同初始化需求创建对象。

下面的代码定义了一个 `Person` 类，它包含三个重载的构造函数：

```
public class Person {
    private static int DEFAULT_AGE = 21;
    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

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

Listing 111: Person 类的构造函数重载示例

如上所示，Person 类提供了三种构造方法：无参构造函数 `Person()` 将 `name` 初始化为"Jeff",`age` 初始化为默认值 21；单参数构造函数 `Person(String name)` 将传入的姓名赋给 `name`,`age` 仍使用默认值；双参数构造函数 `Person(String name, int age)` 用传入的姓名和年龄赋值相应字段。这样，使用者可以根据需要调用不同的构造方法来创建 Person 对象，例如：`new Person()`、`new Person("Alice")` 或 `new Person("Bob", 25)`。

为了避免在多个构造函数中出现重复的初始化代码，我们可以使用 `this(...)` 在一个构造方法内部调用本类的另一个构造方法。例如，我们可以将上例中的构造函数优化如下：将无参构造函数改写为调用双参数构造函数：`public Person() { this("Jeff", DEFAULT_AGE); }`，而单参数构造函数调用双参数构造函数：`public Person(String name) { this(name, DEFAULT_AGE); }`。通过这种方式，`Person()` 和 `Person(String)` 都复用了 `Person(String, int)` 构造函数的初始化逻辑，减少了代码重复。

在继承情况下，Java 还允许在子类构造函数中通过 `super(...)` 调用父类的构造函数。必须注意，构造函数中调用 `this(...)` 或 `super(...)` 必须是方法体的第一条语句。默认情况下，若子类构造函数没有显式调用父类构造函数，编译器会在其开头隐式插入 `super()` 调用父类的无参构造函数（要求父类必须有无参构造函数）。如果父类没有无参构造函数，则子类构造函数需要显式调用父类的其它构造函数，否则编译无法通过。

我们在日常编程中已经使用过构造函数重载的例子，例如 Java 标准库中的 `java.util.Scanner` 类就提供了多种构造函数：允许从输入流、文件、字符串等不同的数据源创建 Scanner，这些构造函数具有不同的参数列表。通过构造函数重载，类可以为创建对象提供多样化的初始化方式，方便程序使用。

6.2.6 使用包组织应用程序

当应用程序逐渐庞大时，合理地组织代码变得很重要。Java 提供了 ** 包 **（Package）机制来对类进行分类和组织。使用包可以将相关的类归组，避免类名冲突，并提高代码的可维护性。我们可以在每个 Java 源文件的开头使用 `package` 语句来声明该文件所属的包。

包名通常使用小写字母，并以层次结构表示。语法形式为：

```
package 包名 [. 子包名 [. 子包名...]];
```

例如，package telephone; 声明该类处于 telephone 包中，而 package telephone.state; 则声明类处于 telephone 包下的 state 子包中。包名与文件系统的目录结构直接对应——Java 要求源文件所在的文件夹路径必须与包名一致。

下面展示一个按包组织的项目目录结构示例。假设我们有一个程序涉及电话状态管理，包含主程序类和多个表示电话状态的类。我们可以将它们组织在名为 telephone 的包和其子包 telephone.state、telephone.input 中，如下所示：

```
src/
`-- telephone/
    |-- Telephone.java
    |-- input/
    |   |-- Keyboard.java
    |   `-- Pen.java
    `-- state/
        |-- TelephoneState.java
        |-- LineBusy.java
        `-- LineWaiting.java
```

Listing 112: 按包组织的项目目录结构示例

如上，telephone 主包和其子包 input、state 作为目录划分，每个 .java 源文件存放在对应的文件夹中。例如，Telephone.java 位于 src/telephone 目录，声明 package telephone;；而 LineBusy.java 位于 src/telephone/state 目录，源码文件开头声明 package telephone.state;。

接下来，我们来看各文件中内容的示例。首先是 telephone 包下的主程序类 Telephone，它依赖于 telephone.state 包下的状态类：

```
package telephone;
import telephone.state.TelephoneState;
import telephone.state.LineWaiting;

public class Telephone {
    private TelephoneState state;

    public Telephone() {
        state = new LineWaiting();
    }
```

```
11
12      public void dial(String phonenumber) {
13          state = state.dial(phonenumber);
14      }
15
16      public void hangup() {
17          state = state.hangup();
18      }
19
20      public static void main(String[] args) {
21          Telephone phone = new Telephone();
22          phone.dial("12341234");
23          phone.hangup();
24      }
25  }
```

Listing 113: telephone 包中的主类 Telephone

在上面的 Telephone 类中，我们首先使用 package telephone; 指明该类属于 telephone 包。由于 Telephone 类需要使用不同包下的类，我们通过 import telephone.state.* 的方式引入了 TelephoneState 和 LineWaiting 类（也可以使用通配符 import telephone.state.* 一次性导入整个 state 包下所有公有类）。随后定义了一个成员变量 state，其类型为抽象类 TelephoneState。在构造函数 Telephone() 中，state 被初始化为 LineWaiting 的实例，表示电话开始时处于“空闲等待”状态。然后，dial 方法和 hangup 方法分别调用当前状态对象的 dial 和 hangup 方法，使电话状态发生转换。主方法 main 展示了创建 Telephone 对象并依次调用 dial 和 hangup 的过程。

接下来是 telephone.state 子包内的几个类。这个子包包含一个抽象父类 TelephoneState，以及两个具体状态类 LineBusy（通话中）和 LineWaiting（空闲）供 Telephone 使用：

```
1  // 文件: TelephoneState.java
2  package telephone.state;
3  public abstract class TelephoneState {
4      protected String numberDialed;
5      public abstract TelephoneState dial(String phonenumber);
6      public abstract TelephoneState hangup();
```

```
}

// 文件: LineBusy.java
package telephone.state;
public class LineBusy extends TelephoneState {
    public LineBusy(String number) {
        super();
        numberDialed = number;
    }
    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }
    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }
}

// 文件: LineWaiting.java
package telephone.state;
public class LineWaiting extends TelephoneState {
    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }
    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }
}
```

Listing 114: telephone.state 包中的状态类

上述代码片段展示了 telephone.state 包中的三个类。抽象类 TelephoneState 定义了抽象方法 dial 和 hangup，并包含一个受保护的属性 numberDialed（记录拨打的号码）。LineBusy 和 LineWaiting 类分别继承 TelephoneState，实现了其抽象方法：例如，在 LineBusy（“占线”状态）下，调用 dial 会抛出异常阻止再次拨号，而调用 hangup 会结束当前通话并返回新的状态 LineWaiting；相反，在 LineWaiting（“空闲”状态）下，调用 dial 将输出拨号信息并切换状态为 LineBusy，而调用 hangup 则因为没有进行中的

通话而抛出异常。这里的 `InvalidPhoneState` 是一个自定义的运行时异常类（代码未列出），用于表示非法的电话状态操作。

通过以上设计，我们实现了一个简单的状态机：`Telephone` 类根据自身持有的 `TelephoneState` 子类实例来决定 `dial` 和 `hangup` 操作的行为变化。这个例子不仅演示了如何使用包来组织相关类，也暗示了更高级的设计模式（状态模式）的运用。

在 Java 中，包为我们提供了清晰的代码组织方式。在大型应用中，合理划分包结构能够使模块职责分明，类命名不冲突，代码维护更加方便。使用包时，记得将源码文件放在对应的目录，并通过 `import` 导入不同包的类。综上，本讲示例展示了继承（包括封装的访问控制）、方法重载以及使用包组织代码等面向对象编程的重要概念和技巧。

7 week6

7.1 Lecture

本周的内容大体可以分为两个部分，为抽象类和接口类，所以和前几周的结构略有不同

Part A：抽象类 (Abstract Classes)

7.1.1 定义 (Definition)

抽象类 (Abstract Class) 是一种不能被实例化的类。它可以定义方法和属性，允许子类继承并提供具体实现。抽象类也可以继承自其他类。

7.1.2 用途 (Usage)

当我们希望定义一个一般化的类，而不希望创建其实例时，可以使用抽象类。例如：

- **Shape** 是 **Triangle**、**Square**、**Circle** 的父类，但不应有 Shape 的实例。
- **Furniture** 是 **Chair**、**Sofa**、**Table**、**Desk** 的父类，但不应有 Furniture 的实例。

以下是一个演示 **抽象类与抽象方法**的代码示例。

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

public abstract class Furniture {
    private String name;
    private List<Part> parts;

    public Furniture(String name) {
        this.name = name;
        this.parts = new ArrayList<Part>();
    }

    public void addPart(Part p) {
        parts.add(p);
    }

    public abstract void stack(Furniture f);
}
```

Listing 115: 声明抽象类 Furniture

```java
public class Chair extends Furniture {
    public Chair() {
        super("Chair");
    }
}

// 编译错误：没有实现抽象方法 stack
// Chair.java:1: error: Chair is not abstract and does not
    override abstract method stack(Furniture) in Furniture
// public class Chair extends Furniture {
//                   ^
// 1 error
```

Listing 116: 声明子类 Chair 但未实现 stack 方法

```java
public class Chair extends Furniture {
    public Chair() {
        super("Chair");
```

```
    }

    public void stack(Furniture f) {
        System.out.println("Don't␣put␣furniture␣on␣chairs!");
    }
}
```

Listing 117: 实现 stack 方法

```
public class FurnitureStore {
    public static void main(String[] args) {
        Chair ch = new Chair();
        ch.stack(new Chair());  // 输出: Don't put furniture on
            chairs!
    }
}
```

Listing 118: 测试类 FurnitureStore

总结:

- 抽象类不能被实例化。
- 抽象类可以包含已实现的方法和抽象方法（声明但没有实现）。
- 抽象方法必须在非抽象的子类中实现，否则编译器会报错。
- 子类使用 `@Override` 注解来明确表示覆盖了父类的方法。
- 子类必须实现所有从抽象类继承的抽象方法，才能被实例化。

7.1.3 Java 中的声明方式 (Declaration in Java)

抽象类通过 `abstract` 关键字声明:

```
public abstract class Furniture {
    private String name;

    public Furniture(String name) {
        this.name = name;
    }

```

```
    public String getName() {
        return name;
    }

    public abstract void stack(Furniture f);
}
```

Listing 119: 抽象类的声明

7.1.4 子类的实现 (Subclass Implementation)

以下代码展示了一个子类 Chair 对抽象类 Furniture 的继承与实现：

```
public class Chair extends Furniture {
    public Chair() {
        super("Chair");
    }

    @Override
    public void stack(Furniture f) {
        System.out.println("不能将家具堆放在椅子上！");
    }
}

public class FurnitureStore {
    public static void main(String[] args) {
        Chair chair = new Chair();
        chair.stack(new Chair());  // 输出：不能将家具堆放在椅
            子上！
    }
}
```

Listing 120: 子类实现抽象类的例子

7.1.5 UML 表示方法 (UML Representation)

在 UML 类图中，抽象类用 *斜体*表示，抽象方法同样用斜体表示。比如：

- **Furniture** 是抽象类，因此在类图中以斜体字体表示。

- **stack()** 是抽象方法，因此也以斜体字体表示。

Part B：接口 (Interfaces)

7.1.6 接口的定义与特点 (Definition and Characteristics)

接口 (Interface) 是一种定义方法签名但不提供具体实现的结构。类可以实现多个接口，弥补了 Java 单继承的限制。

接口的特点：

- 不能包含实例属性。
- 可以包含常量 (`static final`)。
- 只能定义方法的签名，而不包含具体实现（除非是 Java 8 引入的默认方法）。
- 一个类可以实现多个接口。

接口（Interfaces）与抽象类类似，但有如下区别：

- **声明：** 使用关键字 `interface` 而非 `class`。
- **实现：** 使用关键字 `implements` 实现接口。
- **特性：**
  - 不允许定义实例属性。
  - 只定义方法签名，通常不提供方法实现（除非是 Java 8 引入的 `default` 方法）。
  - 不能实例化接口。
  - 一个类可以实现多个接口（多重实现）。

接口的使用与抽象类类似，但接口允许我们对程序设计进行更灵活的定义。例如，一个类可以实现多个接口，而一个类只能继承一个父类。

示例：声明接口 Swim

```
public interface Swim {
    void swim();
}
```

Listing 121: 声明接口 Swim

使用接口

接口的实现通过使用 `implements` 关键字：

```
public class Fish implements Swim {
    @Override
    public void swim() {
        System.out.println("Fish is swimming.");
    }
}
```

Listing 122: 实现接口的类 Fish

多个接口的实现

Java 支持一个类实现多个接口。

```
public interface Jump {
    void jump();
}

public class Dolphin implements Swim, Jump {
    @Override
    public void swim() {
        System.out.println("Dolphin is swimming.");
    }

    @Override
    public void jump() {
        System.out.println("Dolphin is jumping.");
    }
}
```

Listing 123: 实现多个接口的类 Dolphin

总结：

- 接口定义了一组规范，类通过实现接口来保证提供这些规范中声明的方法。
- 接口与抽象类不同的是，它允许 **多重实现**，即一个类可以实现多个接口。
- 接口的实现必须提供接口中声明的所有方法。

本例子展示了如何通过接口 Move 来实现不同类的行为。

接口是 Java 提供的一种机制，用于定义多个类可以实现的通用行为。通过接口，我们可以实现 Java 中的 **多重继承**。一个类可以实现多个接口，从而拥有多个接口中定义的方法。

在这个例子中，我们定义了一个接口 Move，并让两个类 Dog 和 Dolphin 实现了它。它们都提供了各自对于 move() 方法的实现。**这一机制使得不同类可以共享相同的接口，并提供各自的实现细节**。

接口定义

接口使用关键字 interface 来定义。

```
public interface Move {
    public void move(double hours);
}
```

Listing 124: 接口定义 Move

这个接口定义了一个方法 move()，接受一个双精度参数 hours。注意到接口中没有实现方法体。实现这个接口的类必须提供具体的方法实现。

实现类 Dog

```
public class Dog implements Move {
    private String region;  // Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }

    @Override
    public void move(double hours) {
        if (region.equals("water")) {
            kmTravelled += (waterSpeed_kmh * hours);
        } else if (region.equals("land")) {
            kmTravelled += (landSpeed_kmh * hours);
        }
    }

```

```
20      public double getKMTravelled() {
21          return kmTravelled;
22      }
23  }
```

Listing 125: 实现接口的类 Dog

类 Dog 实现了 Move 接口，并提供了对 move() 方法的具体实现。

实现类 Dolphin

```
1   public class Dolphin implements Move {
2       private String region;  // Water or Land
3       private double landSpeed_kmh = 1.0;
4       private double waterSpeed_kmh = 60.0;
5       private double kmTravelled = 0.0;
6
7       public Dolphin(String region) {
8           this.region = region;
9       }
10
11      @Override
12      public void move(double hours) {
13          if (region.equals("water")) {
14              kmTravelled += (waterSpeed_kmh * hours);
15          } else if (region.equals("land")) {
16              kmTravelled += (landSpeed_kmh * hours);
17          }
18      }
19
20      public double getKMTravelled() {
21          return kmTravelled;
22      }
23  }
```

Listing 126: 实现接口的类 Dolphin

类 Dolphin 同样实现了 Move 接口，并根据其特性提供了 move() 方法的不同实现。

测试类 MovingAnimals

我们可以创建一个 Move[] 数组来存储所有实现了 Move 接口的对象，从而实现 **接口多态性**。

```
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");

        Move[] movingAnimals = {dog, dolphin};

        for (Move m : movingAnimals) {
            m.move(1.0);  // 移动 1 小时
        }

        System.out.println(dog.getKMTravelled());       // 输出：50.0
        System.out.println(dolphin.getKMTravelled());  // 输出：1.0
    }
}
```

Listing 127: 测试类 MovingAnimals

在这个例子中：

- 类 Dog 和 Dolphin 都实现了 Move 接口。
- 虽然它们的实现方法不同，但它们都可以被视为 Move 类型使用。
- 通过创建一个 Move[] 数组，我们可以遍历不同类的对象，并统一调用 move() 方法。

总结

- **接口类型的数组：**所有实现了接口 Move 的类都可以存储在一个 Move[] 数组中。
- **多态性：** 接口允许我们使用通用类型来处理不同类的对象。
- **灵活性：** 如果我们想要扩展这个系统，可以添加新的类并实现 Move 接口，而不需要修改现有的代码。

7.1.7 接口的声明与实现 (Declaration and Implementation)

接口使用 `interface` 关键字定义。例子:

```
public interface Move {
    void move(double hours);
}

public class Dog implements Move {
    public void move(double hours) {
        System.out.println("Dog␣moves␣for␣" + hours + "␣hours."
            );
    }
}

public class Dolphin implements Move {
    public void move(double hours) {
        System.out.println("Dolphin␣swims␣for␣" + hours + "␣
            hours.");
    }
}
```

Listing 128: 接口的定义与实现

7.1.8 UML 表示方法 (UML Representation)

在 UML 中，接口的表示与抽象类有所不同。为了展示接口，我们使用特殊的标识:

- 使用标签 `<<interface>>` 来表示接口。
- 接口名称使用正常字体表示。
- 抽象方法使用斜体来表示，表示它们是多态方法。

PropulsionEngine 接口

在下图中，我们定义了一个接口 `PropulsionEngine`，它包含两个方法:

- `refuel(amount: int): void`
- `fly(duration: double): void`

UML 表示：

```
<<interface>>
PropulsionEngine
--------------------------
+refuel(amount: int): void
+fly(duration: double): void
```

说明：

- **接口名称：**使用标签 `<<interface>>` 来区分。
- **方法：**使用斜体来表示接口中的方法是多态的。

7.1.9 实现 PropulsionEngine 接口的类

下图展示了两个实现了 `PropulsionEngine` 接口的类：

- `PlasmaEngine`
- `JetEngine`

两者都提供了对接口中定义的方法的具体实现。

UML 类图表示：

```
<<interface>>                PlasmaEngine          JetEngine
PropulsionEngine             -----------------     -----------------
---------------------        +refuel(): void       +refuel(): void
+refuel(amount: int): void    +fly(): void         +fly(): void
+fly(duration: double): void
```

接口实现的表示

在 UML 图中：

- 实现关系使用 **虚线 + 空心三角箭头**表示，指向接口。
- 这种关系不同于类的继承关系（使用实线）。
- 用于表明 **实现类提供了接口定义的所有方法实现**。

注意：接口的实现是一种特殊的多态性，因为实现类可以提供自己的方法实现方式，同时符合接口的定义

7.1.10 Default Methods

在 Java 8 之前，接口中只能包含抽象方法，所有实现接口的类必须提供方法的具体实现。然而，自 Java 8 起，允许在接口中定义 **默认方法 (Default Methods)**。

默认方法是一种具有方法体的接口方法，使用关键字 `default` 声明。

语法 (Syntax)

- **声明格式：**

```
[modifier] default <returntype> MethodName([parameters]) {
    // 方法体
}
```

- **示例：**

```java
interface Talk {
    public void talk();

    public default void talking() {
        System.out.println("I␣am␣talking.");
    }
}
```

Listing 129: 默认方法的定义

实现类与默认方法

类可以实现接口中的默认方法，而不必提供其实现；但也可以选择重写它。

```java
public class Alien implements Talk {
    @Override
    public void talk() {
        System.out.println("zzzfer342aa");
    }
}

public class Cat implements Talk {
```

```
    @Override
    public void talk() {
        System.out.println("meow");
    }

    @Override
    public void talking() {
        System.out.println("Overridden in Cat");
    }
}
```

Listing 130: 实现类 Alien 和 Cat

说明

- 类 `Alien` 只实现了接口中的抽象方法 `talk()`。它可以直接调用接口中定义的默认方法 `talking()`。
- 类 `Cat` 实现了接口中的 `talk()`，并且重写了默认方法 `talking()`。

测试类

```
public class Main {
    public static void main(String[] args) {
        Alien alien = new Alien();
        Cat cat = new Cat();

        alien.talk();             // 输出：zzzfer342aa
        alien.talking();          // 输出：I am talking.

        cat.talk();               // 输出：meow
        cat.talking();            // 输出：Overridden in Cat
    }
}
```

Listing 131: 测试类 Main

总结

- **接口中的默认方法**提供了一种机制，使得接口可以演化而不会破坏已有的实现类。

- 如果不重写默认方法，类将继承接口中提供的实现。
- 如果希望修改默认方法的行为，可以在实现类中进行 **方法重写 (Override)**。

8 week7

8.1 Lecture

8.1.1 Generics 的动机与案例分析

在没有泛型（Generics）的世界里，需要为每种数据类型重复编写类，如：

- IntArrayList, DoubleArrayList, FloatArrayList, StringArrayList 等。

或者，把所有数据都存入 `Object` 类型的数组中，但这样在使用时就必须进行显式类型转换，容易引发运行时错误。例如，当调用者忽略了强制转换时，编译器无法在编译期发现潜在错误，而在运行时可能抛出 `ClassCastException`。下面展示了不使用泛型时可能出现的问题：

```
public class Container {
    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        Object oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }
}
```

Listing 132: Container 类（无泛型）

调用时还必须进行类型转换：

```
Container c = new Container("Hello Box!");
String s1 = (String)c.get(); // 强制类型转换
c.set("New string here!");
String s2 = (String)c.get();
```

Listing 133: 调用示例

扩展说明：在课件中还讨论了如果不使用泛型，类中的方法调用不但显得冗长，而且在编译阶段无法确保类型安全，导致运行时检查困难，增加了维护成本。

8.1.2 Generics 的优势

使用泛型带来的主要优势包括：

- **编译期更强的类型检查**：在编译期间就能检测到类型不匹配的错误，从而降低运行时错误的风险。
- **消除不必要的类型转换**：使用泛型后无需显式地进行强制转换，使代码更简洁，更容易阅读和维护。
- **使程序员能够实现通用算法**：泛型方法和泛型类使得算法可用于多种数据类型，避免了代码重复。

扩展说明：课件强调了泛型不仅使代码更加模块化，还使得 API 的意图更加明确，调用者可以更直观地理解应传入何种类型数据，从而提高整体开发效率。

8.1.3 泛型类的语法与定义

泛型类允许在类定义中指定类型参数，这样就可以在整个类中使用该参数来描述变量、返回类型和方法参数。其基本语法如下：

```
public class Container<T> {
    private T element;

    public Container(T element) {
        this.element = element;
    }
```

```
7
8       public T set(T element) {
9           T oldElement = this.element;
10          this.element = element;
11          return oldElement;
12      }
13
14      public T get() {
15          return element;
16      }
17  }
```

Listing 134: 定义泛型类

扩展说明：

- 类型参数 T 可以更换为任意标识符，通常用大写字母表示（例如 E, K, V 等）。
- 泛型类允许在实例化时指定具体的数据类型，使得编译器可以在编译期进行类型检查。
- 课件中还提到，对于不需要泛型能力的场景，避免过度设计非常重要，只有在充分利用类型安全检查时才应使用泛型。

8.1.4 泛型容器示例

例如，上述 Container<T> 类使得调用代码可以这样写：

```
1  public static void main(String[] args) {
2      // 泛型实例化时指定具体的类型，这里使用 String 类型
3      Container<String> c = new Container<String>("Hello␣Box!");
4      String s1 = c.get();
5      c.set("New␣string␣here!");
6      String s2 = c.get();
7
8      System.out.println(s1);
9      System.out.println(s2);
10 }
```

Listing 135: 使用泛型 Container

扩展说明：

- Java 7 以后可以使用菱形语法简化实例化，如 `new Container<>("Hello Box!");`。
- 此示例展示了如何利用泛型确保存储和取出的数据类型一致，避免了强制类型转换的繁琐和错误风险。

8.1.5 泛型在数据结构中的应用

泛型的使用不仅限于单个容器，也常应用于数据结构中，例如链表。下面是一个使用泛型实现的链表（Linked List）的部分代码：

```
public class LinkedList<T> {
    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    public int size() {
        return size;
    }
}

```

```
public class Node<T> {
    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
        next = null;
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T v) {
        value = v;
    }
}
```

Listing 136: 泛型链表及节点的定义

扩展说明:

- 泛型链表允许存储任意类型的数据，使用时只需指定相应的类型参数，如 `LinkedList<String>` 或 `LinkedList<Integer>`。
- 课件中强调，泛型的数据结构在遍历、添加和删除操作上逻辑一致，无需为不同的数据类型编写重复代码。

8.1.6 泛型静态方法

静态方法也可定义为泛型方法，其语法与普通方法不同，需要在方法前声明类型参数:

```java
public static <T> T find(T needle, T[] haystack) {
    for(T element : haystack) {
        if(element.equals(needle)) {
            return element;
        }
    }
    return null;
}
```

Listing 137: 泛型静态方法示例

使用时可以显式传入类型参数，如：

```java
Points.<AbsolutePoint>find(point, points);
```

Listing 138: 调用泛型静态方法

扩展说明：

- 泛型静态方法的类型参数与类的类型参数无关，它们仅对该方法生效。
- 在使用这些方法时，编译器通常能够进行类型推断，从而使得调用更加简洁。
- 这种方法特别适用于实现通用工具类，例如排序、搜索等操作。

8.1.7 UML 模板类与泛型

在 UML 中，用于描述带泛型类（模板类）的方式通常会在类名旁边注明类型参数。例如：

Container<T>

这表明类 `Container` 使用了一个类型参数 `T`，并在内部使用该类型来标记属性和方法。扩展说明：

- UML 中还会标注类型参数的约束，如 `T extends Item`，从而表示类型参数必须继承自某个特定类。
- 这种标注有助于设计者与实现者更好地理解类之间的关系以及泛型约束。

8.1.8 有界类型参数

有界类型参数允许对泛型参数设置约束，例如要求所有使用的类型都必须继承自某个超类型。这不仅增强了类型安全性，还可以调用超类型定义的方法。示例如下：

```
public class Barrel<T extends Liquid> {
    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayList<T>();
    }

    public void add(T liquid) {
        liquids.add(liquid);
    }

    public void outputVolume() {
        double total = 0.0;
        for(T e : liquids) {
            total += e.getLitres();
            System.out.println(e + ":␣" + e.getLitres() + "L");
        }
        System.out.println("Total:␣" + total + "L\n");
    }
}

public class Liquid {
    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }
    public double getLitres() {
        return litres;
    }
}

public class Water extends Liquid {
    public Water(double litres) {
```

```
35          super(litres);
36      }
37      public String toString() { return "Water"; }
38  }
39
40  public class Oil extends Liquid {
41      public Oil(double litres) {
42          super(litres);
43      }
44      public String toString() { return "Oil"; }
45  }
```

Listing 139: 有界类型参数示例 — Barrel 类

扩展说明：

- 由于 T 限制为 Liquid 或其子类，因此在 outputVolume() 方法中可以安全调用 getLitres() 方法。
- 这种约束使得容器仅适用于存储具有某些共有特性的对象（例如液体的体积），从而提高了程序的健壮性。

8.1.9 泛型数组的问题与解决方法

在 Java 中不能直接创建泛型数组，例如下面代码会报错：

```
1  public class DynamicArray<T> {
2      private T[] array;
3
4      public DynamicArray() {
5          array = new T[4];  // 编译错误：generic array creation
6      }
7  }
```

Listing 140: 错误示例 — 泛型数组创建

解决方法是利用强制类型转换：

```
1  public class DynamicArray<T> {
2      private T[] array;
3
4      @SuppressWarnings("unchecked")
```

```
    public DynamicArray() {
        array = (T[]) new Object[4];
    }
}
```

Listing 141: 正确示例 — 泛型数组创建

扩展说明:

- Java 的泛型采用类型擦除机制，运行时无法获知真正的类型信息，因此直接创建泛型数组是不允许的。
- 使用 `@SuppressWarnings("unchecked")` 可以抑制编译器的不安全操作警告，但需要程序员自行保证类型安全。

8.1.10 Iterator 与 Iterable 接口

迭代器（Iterator）用于遍历集合，它包含 `hasNext()` 和 `next()` 方法。集合类若实现 `Iterable` 接口，就可以支持 foreach 循环，示例如下:

```
LinkedList<String> list = new LinkedList<String>();
for(String s : list) {
    System.out.println(s);
}

Iterator<String> iterator = list.iterator();
while(iterator.hasNext()){
    String s = iterator.next();
    System.out.println(s);
}
```

Listing 142: Iterator 使用示例

扩展说明:

- Iterator 接口通常还定义了 `remove()` 方法，用于删除集合中的元素，但如果不支持删除操作，则应抛出 `UnsupportedOperationException`。
- 课件中提到，通过 Iterator 遍历集合使得代码与底层实现解耦，增强了集合操作的一致性和可靠性。

8.1.11 将 LinkedList 改造为 Iterable 集合

为了使自定义的链表能在 foreach 循环中使用，需要让其实现 `Iterable<T>` 接口，并提供自定义的迭代器类：

```
public class LinkedList<T> implements Iterable<T> {
    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    public int size() {
        return size;
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator<T>(head);
    }
}

class LinkedListIterator<T> implements Iterator<T> {
    private Node<T> cursor;

    public LinkedListIterator(Node<T> head) {
```

```
36          cursor = head;
37      }
38
39      public boolean hasNext() {
40          return cursor != null;
41      }
42
43      public T next() {
44          T element = cursor.getValue();
45          cursor = cursor.getNext();
46          return element;
47      }
48  }
```

Listing 143: 改造后的 LinkedList 和自定义迭代器

扩展说明:

- 实现 `Iterable<T>` 接口后，该链表可以直接用于增强 for 循环，从而简化遍历过程。
- 迭代器内部通过一个 `cursor` 指针来跟踪当前位置，每次调用 `next()` 后将其指向下一个节点。
- 课件还强调，若在遍历过程中修改链表结构，可能会导致迭代器行为异常，因此实际应用中还需要考虑并发修改的问题。

8.1.12 For-Each 循环与传统 for 循环的比较

使用 foreach 循环的优点:

- 代码简洁，无需管理迭代器或数组下标。
- 提高代码可读性和可维护性。
- 内部调用了 Iterator，保证了遍历的一致性与类型安全。

扩展说明:

- 传统 for 循环虽灵活，但容易引入边界错误；而 foreach 循环抽象掉了迭代器的细节，使得代码更不易出错。

- 课件中指出，foreach 循环只适用于实现了 `Iterable<T>` 接口的集合，因此自定义数据结构应尽量支持该接口。

9 week8

9.1 Lecture

本周我们主要学习了 Java 编程中的三大模块：异常（**Exception**）、枚举类型（**Enum**）以及单元测试框架（**JUnit**）中的断言（**Assert**）机制。

9.1.1 异常（Exception）

定义：异常（Exception）是程序在运行过程中遇到的不正常状态或错误。Java 使用异常处理机制来优雅地管理这些不可预测的情况。

使用情景：

- 用户输入不合法（如除以零）
- 访问空对象（NullPointer）
- 打开不存在的文件
- 服务器响应超时

Java 异常分为三类：

- **Checked Exception**（受检异常）：必须显式处理，适用于 I/O、数据库、网络等可能失败的外部交互；
- **Unchecked Exception / RuntimeException**（运行时异常）：逻辑或程序错误，例如数组越界、空指针；
- **Error**（错误）：JVM 层面的错误，如栈溢出、内存溢出，不建议捕获。

关键语法：

```
try {
    // 可能出错的代码块
} catch (IOException e) {
    e.printStackTrace();
}
```

Listing 144: try-catch 语法结构

自定义异常：

```
public class InvalidRefreshRateException extends Exception {
    public InvalidRefreshRateException() {
        super("Unsupported refresh rate value");
    }
}
```

Listing 145: 定义并抛出受检异常

在方法中抛出：

```
public double setRefreshRate(double hz) throws
    InvalidRefreshRateException {
    if (hz < 0 || hz > MAX_REFRESH_RATE) {
        throw new InvalidRefreshRateException();
    }
    return hz;
}
```

与 if 语句对比：异常用于表示严重的、非预期的错误，而非程序控制流程，避免用异常替代正常的判断语句。

9.1.2 枚举类型（Enum）

定义：Enum（枚举）是一种特殊的类，表示一组固定的常量，通常用于状态、命令、选项等固定集合。

使用场景：

- 表示方向（Direction）
- 表示星期（月、季节）
- 表示系统状态（如网络连接状态）

基本语法：

```
public enum Suit {
    HEARTS, DIAMONDS, SPADES, CLUBS;
```

```
}
```

Listing 146: 定义基本枚举类型

遍历与序号：

```
for (Suit s : Suit.values()) {
    System.out.println(s.name() + ": " + s.ordinal());
}
```

带字段与构造方法：

```
public enum Suit {
    HEARTS(2, "Red"),
    SPADES(3, "Black");

    private int number;
    private String colour;

    Suit(int n, String colour) {
        this.number = n;
        this.colour = colour;
    }

    public String getColour() {
        return this.colour;
    }
}
```

注意事项：

- 枚举构造器默认为 private；
- 枚举不能被继承；
- 枚举值在类加载时初始化，不可动态创建。

9.1.3 断言（Assert）

定义： Java 中的 **Assert（断言）** 是一种用于调试阶段的机制，用来验证某个条件在运行时是否成立。如果条件不满足，程序会抛出 `AssertionError` 并中止执行。

典型使用场景：

- 检查前置条件（如参数不为 null）；
- 验证中间状态（如对象处于有效状态）；
- 后置条件校验（例如数组排序后是否有序）；
- 不用于用户输入判断或程序核心控制流程。

语法结构：

```
assert condition;
assert condition : "Message if failed";
```

运行方式：断言默认是禁用的。需使用参数 `-ea`（enable assertions）启用：

```
java -ea MyProgram
```

示例：

```
public void install(File f, License key) {
    assert f.exists();                  // 文件必须存在
    assert key != null;                 // 密钥不能为空
    assert key.isValid();               // 密钥必须有效
}
```

注意事项：

- assert 适用于开发与调试阶段；
- 不建议替代正常的 if-else 判断；
- 若断言失败将抛出 `AssertionError`，程序终止。

—

9.1.4 JUnit 单元测试框架（JUnit Testing Framework）

定义：JUnit 是 Java 最广泛使用的单元测试框架，支持开发者对方法进行系统性验证，确保功能实现正确性。

测试类型对比：

- **Black-box testing（黑盒测试）**：测试者不关心程序内部逻辑；
- **White-box testing（白盒测试）**：基于源代码结构测试；
- **Unit testing（单元测试）**：对类或方法进行独立验证，是白盒测试的一种形式。

常用注解（Annotations）：

- `@Test`：标识一个测试方法；
- `@Before` / `@After`：每个测试方法前/后执行；
- `@BeforeClass` / `@AfterClass`：所有测试运行前/后只执行一次（需声明为 static 方法）。

常见断言方法：

- `assertTrue(condition)`：断言条件为真；
- `assertFalse(condition)`：断言条件为假；
- `assertEquals(expected, actual)`：断言两个值相等；
- `assertNull(obj)`：断言对象为 null；
- `assertSame(obj1, obj2)`：断言引用一致。

完整示例：

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3
4  public class ContainerTest {
5      @Test
6      public void checkNull() {
7          Container a = new Container(null);
8          assertNull(a.get());
9      }
10
11     @Test
12     public void checkValue() {
13         Container b = new Container("INFO1113");
```

```
        assertEquals("INFO1113", b.get());
    }
}
```

编译与运行测试：

```
# 编译测试类（需 .jar 文件支持）
javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar ContainerTest.
    java

# 运行测试类
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.
    runner.JUnitCore ContainerTest
```

JUnit 优势：

- 能快速验证代码逻辑正确性；
- 提高重构安全性；
- 支持持续集成；
- 可覆盖多种测试路径与边界情况。

10 week9

10.1 Lecture

10.1.1 递归（Recursion）

定义与要点

- 递归：方法内部调用自身。
- 必须有基准情况(Base case)以终止递归,否则会发生 `StackOverflowError`。
- 每层调用会消耗栈空间，滥用会导致性能和内存问题。

示例：阶乘

```
// 计算 n!
public static int factorial(int n){
    if(n <= 1)            // 基准情况
```

```
        return 1;
    else                    // 递归情况
        return n * factorial(n-1);
}
```

调用过程（以 factorial(3) 为例）

1. factorial(3) → 3 × factorial(2)
2. factorial(2) → 2 × factorial(1)
3. factorial(1) 基准，返回 1
4. 回溯：factorial(2)=2×1=2； factorial(3)=3×2=6

Quiz 示例

1. **问：**为什么 factorial(3) 输出 6 而不是 4?

   答：

   - 首次：3×factorial(2)
   - 次次：2×factorial(1)
   - factorial(1) =1
   - 合并：$3 \times (2 \times 1) = 6$

10.1.2 面向对象下的递归（Recursion with OOP）

类结构

```
class FamilyMember {
    private String name;
    List<FamilyMember> children;

    public FamilyMember(String name){
        this.name = name;
        children = new ArrayList<>();
    }
    public void addChild(FamilyMember c){
        children.add(c);
    }
```

```
    // 递归方法：收集自己及所有后代
    public List<FamilyMember> getAllParents(){
        List<FamilyMember> parents = new ArrayList<>();
        if(children.size()>0){
            parents.add(this);
            for(FamilyMember c: children)
                parents.addAll(c.getAllParents());
        }
        return parents;
    }
}
```

示例树结构

Liz
/ \
Sue Joe
| / \
Max Ali Fin

调用 `Liz.getAllParents()` 返回

[Liz, Sue, Max, Joe, Ali, Fin]

Quiz 示例

1. **问：**若只调用 `Sue.getAllParents()`，输出是什么？**答：**

   [Sue, Max]

   因为 Sue 有一个孩子 Max，再往下没有孩子，返回两人列表。

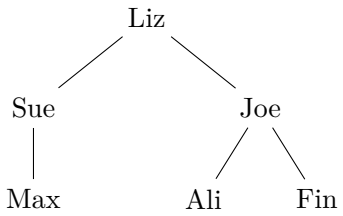
10.1.3 面向对象下的递归（Family Tree）

```
// 每个成员可以有多个子成员
class FamilyMember {
    private String name;
    List<FamilyMember> children;

    public FamilyMember(String name) {
        this.name = name;
```

```
        children = new ArrayList<>();
    }
    public void addChild(FamilyMember c) {
        children.add(c);
    }
    /**
     * 收集自己和所有后代
     */
    public List<FamilyMember> getAllParents() {
        List<FamilyMember> parents = new ArrayList<>();
        if (children.size() > 0) {
            parents.add(this);
            for (FamilyMember c : children) {
                parents.addAll(c.getAllParents());
            }
        }
        return parents;
    }
}
```

示例家族树



调用 Liz.getAllParents() 的详细展开

1. 调用 `Liz.getAllParents()`：

$$\text{parents}_{\text{Liz}} = []$$

- 因为 `children.size()>0`，先 `parents.add(Liz)`

$$\text{parents}_{\text{Liz}} = [\text{Liz}]$$

2. 遍历 `Liz.children`，第一个孩子是 `Sue`：

$$\text{parents}_{\text{Liz}}.\text{addAll}(\mathit{Sue.getAllParents}())$$

(a) 调用 `Sue.getAllParents()`：

$$\text{parents}_{\text{Sue}} = []$$

- `parents.add(Sue)`

$$\text{parents}_{\text{Sue}} = [Sue]$$

(b) 遍历 `Sue.children`，只有 `Max`：

$$\text{parents}_{\text{Sue}}.\text{addAll}(Max.getAllParents())$$

i. 调用 `Max.getAllParents()`：

$$children.size() == 0 \;\Rightarrow\; \text{return } []$$

$$\text{parents}_{\text{Sue}} = [Sue]$$

(c) `Sue.getAllParents()` 返回 $[Sue]$。

$$\text{parents}_{\text{Liz}} = [Liz] \cup [Sue] = [Liz,\, Sue]$$

3. 遍历 `Liz.children`，第二个孩子是 `Joe`：

$$\text{parents}_{\text{Liz}}.\text{addAll}(Joe.getAllParents())$$

(a) 调用 `Joe.getAllParents()`：

$$\text{parents}_{\text{Joe}} = []$$

- `parents.add(Joe)`

$$\text{parents}_{\text{Joe}} = [Joe]$$

(b) 遍历 `Joe.children`：先 `Ali`，后 `Fin`：

- `Ali.getAllParents()` 返回 []（无孩子）。
- `Fin.getAllParents()` 返回 []（无孩子）。

$$\text{parents}_{\text{Joe}} = [Joe]$$

(c) `Joe.getAllParents()` 返回 $[Joe]$。

$$\text{parents}_{\text{Liz}} = [Liz,\, Sue] \cup [Joe] = [Liz,\, Sue,\, Joe]$$

4. `Liz.getAllParents()` 完成，最终结果：

$$[Liz,\ Sue,\ Joe]$$

但注意：`getAllParents` 仅收集自己和所有“有孩子”的节点。若想列出整个家族，包括叶节点，可改为无条件先 `add(this)`。

改进：收集所有节点（包括叶节点）将方法改为：

```
public List<FamilyMember> getAllDescendants(){
    List<FamilyMember> list = new ArrayList<>();
    list.add(this);
    for(FamilyMember c: children){
        list.addAll(c.getAllDescendants());
    }
    return list;
}
```

调用 `Liz.getAllDescendants()` 得到：

$$[Liz,\ Sue,\ Max,\ Joe,\ Ali,\ Fin]$$

10.1.4 记忆化（Memoization）

定义与要点

- 在递归中，重复子问题大量浪费性能。
- 用 `Map<Key,Value>` 保存已计算结果，再次命中则直接返回。

示例：带缓存的 Fibonacci

```
public class FibonacciCache {
    private Map<Integer,Integer> cache = new HashMap<>();

    public int fib(int n){
        if(cache.containsKey(n))       // 命中缓存
            return cache.get(n);
        int result;
        if(n<=1) result = n;            // 基准情况
        else result = fib(n-1)+fib(n-2);
        cache.put(n,result);            // 存入缓存
```

```
        return result;
    }
}
```

运行逻辑

1. 首次计算某个 n，结果存入 cache。
2. 再次请求相同 n，直接 cache.get(n)，不再递归。
3. 时间复杂度由 $O(2^n)$ 降至 $O(n)$。

Quiz

1. **问：**在上例中，“不重复计算”是在哪行实现的？**答：**

```
if(cache.containsKey(n))
    return cache.get(n);
```

10.1.5 文档生成（Javadoc）

定义与要点

- Javadoc 是 Java 自带的文档生成工具。
- 使用 /** ... */ 注释，并配合标签：@param、@return、@throws 等。
- 生成命令：

```
javadoc -d docs *.java
```

示例

```
/**
 * 判断一个整数是否为偶数。
 * @param number 要检查的整数
 * @return 如果是偶数返回 true，否则返回 false
 */
public boolean isEven(int number){
    return number%2==0;
}
```

Quiz

1. **问：**如何为方法抛出的异常生成文档？**答：**

```java
/**
 * 打开文件并读取第一行。
 * @param path 文件路径
 * @return 第一行内容
 * @throws IOException 如果读取失败
 */
public String readLine(String path) throws IOException {
    ... }
```

本讲稿主要围绕两大核心知识点：**匿名类（Anonymous Class）**与**Lambda 表达式（Lambda Expression）**，并最后比较二者异同。

11 week10

11.1 Lecture

本次课件涉及以下知识点，每个知识点均以\subsubsection独立呈现，并附带课件中所有代码片段。

11.1.1 匿名类（Anonymous Class）的定义与概念

匿名类（Anonymous Class）是一种无需显示命名的局部类。**当我们只需要在某处使用一次某个类型的实现时**，可以通过匿名类同时声明并实例化该类型，从而简化代码结构。

核心概念

- **即声明即实例化**：不需要单独定义类名，直接在需要的地方用`new 接口/类() { … }`完成。
- **用途场景**：常用于事件处理（如 GUI 按钮监听）、一次性回调或策略模式的简易实现。

11.1.2 匿名类的语法与使用示例

匿名类的基本语法形式如下：

```
// 基本语法
new Type() {
    // 字段（可选）
    // 方法实现
};
```

课件示例：定义接口并通过匿名类实现一次性调用：

```
interface SayHello {
    void hello();
}

public class Demo {
    public static void main(String[] args) {
        SayHello hi = new SayHello() {
            @Override
            public void hello() {
                System.out.println("Hello!");
            }
        };
        hi.hello(); // 输出 Hello!
    }
}
```

Listing 147: SayHello 接口匿名类示例

11.1.3 匿名类的特性（Properties）

匿名类具有以下显著特性：

- **唯一实例**：每个匿名类声明都会生成一个新的类型并实例化，无法复用。
- **局部声明**：通常在方法内部或代码块内声明，作用域仅限于声明处。
- **继承/实现**：匿名类可以继承自一个类或实现一个接口，但由于不可命名，不能有显式构造函数。

11.1.4 匿名类示例详解：IntegerBinaryOperation 接口与计算器应用

本例展示如何使用匿名类为简单的二元整型运算定义策略：

```
interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class Calculator {
    public static void main(String[] args) {
        IntegerBinaryOperation add = new IntegerBinaryOperation
            () {
            @Override
            public int apply(int x, int y) {
                return x + y;
            }
        };
        IntegerBinaryOperation subtract = new
            IntegerBinaryOperation() {
            @Override
            public int apply(int x, int y) {
                return x - y;
            }
        };
        IntegerBinaryOperation multiply = new
            IntegerBinaryOperation() {
            @Override
            public int apply(int x, int y) {
                return x * y;
            }
        };

        System.out.println(add.apply(1, 1));              // 2
        System.out.println(subtract.apply(3, 5));         // -2
        System.out.println(add.apply(3,
            subtract.apply(3, multiply.apply(2, 6))));  // -6
    }
}
```

Listing 148: Calculator.java——使用匿名类实现加减乘运算

该示例中：

- `IntegerBinaryOperation` 为功能性接口（Functional Interface），仅含一个抽象方法 `apply(int,int)`。
- 对于加、减、乘三个运算，分别通过三个匿名类实例完成策略定义。
- 调用链中，多个运算可嵌套组合，实现简单的表达式计算。

11.1.5 匿名类在集合（HashMap）中的应用

将匿名类与集合结合，可动态按键获取对应行为：

```
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class CalculatorWithMap {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations =
            new HashMap<>();
        operations.put("ADD", new IntegerBinaryOperation() {
            @Override public int apply(int x, int y) { return x
                + y; }
        });
        operations.put("SUB", new IntegerBinaryOperation() {
            @Override public int apply(int x, int y) { return x
                - y; }
        });
        operations.put("MUL", new IntegerBinaryOperation() {
            @Override public int apply(int x, int y) { return x
                * y; }
        });

        System.out.println(operations.get("ADD").apply(1, 1));
            // 2
        System.out.println(operations.get("SUB").apply(3, 5));
            // -2
        System.out.println(operations.get("ADD").apply(3,
```

```
            operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6))));
                                // -6
    }
}
```

Listing 149: CalculatorWithMap.java——匿名类与 HashMap

此模式灵活：键名（如"ADD"）即策略标识，可方便扩展新运算而无需修改调用逻辑。

11.1.6 匿名类的优势与应用场景

- **减少类文件**：无需显式声明一个完整的类并单独编译。
- **封装局部逻辑**：例如 GUI 事件监听器常用匿名类实现一次性回调。
- **符合策略模式**：将不同实现封装于不同匿名类实例，通过统一接口调用。

11.1.7 Lambda 表达式（Lambda Expression）的定义与概念

Lambda 表达式（Lambda Expression）是 Java 8 引入的匿名函数实现，适用于仅含单一抽象方法的「功能性接口（Functional Interface）」。其本质是一个无名方法，使代码更加简洁。

核心概念

- **无需创建类或匿名内部类**，直接编写函数体。
- **目标类型推断**：参数类型可由编译器根据上下文自动推断。
- **函数式编程风格**：支持传递行为、组合函数等。

11.1.8 Lambda 表达式的语法与示例

基本语法：

```
// 单行表达式
(arg1, arg2, ..., argN) -> expression

```

```
// 多行代码块
(arg1, arg2) -> {
    // 多行语句
    return result;
}
```

课件示例：

```
SayHello hi = () -> System.out.println("Hello!");
IntegerBinaryOperation add = (x, y) -> x + y;
IntegerBinaryOperation addExplicit = (int x, int y) -> x + y;
```

Listing 150: Lambda 简单示例

11.1.9 Lambda 表达式示例详解：CalculatorLambdas

与匿名类示例对应，使用 Lambda 表达式重构：

```
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class CalculatorLambdas {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations =
            new HashMap<>();
        operations.put("ADD", (x, y) -> x + y);
        operations.put("SUB", (int x, int y) -> x - y);
        operations.put("MUL", (x, y) -> x * y);

        System.out.println(operations.get("ADD").apply(1, 1));
            // 2
        System.out.println(operations.get("SUB").apply(3, 5));
            // -2
        System.out.println(operations.get("ADD").apply(3,
            operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6))));
                                // -6
```

```
    }
}
```

Listing 151: CalculatorLambdas.java——使用 Lambda

11.1.10 多行 Lambda 表达式

当函数体不止一行时，使用大括号包裹：

```
SayHello hi = () -> {
    System.out.println("Hello!");
    System.out.println("Yo!");
};
```

Listing 152: 多行 Lambda 示例

11.1.11 接口中的默认方法（Default Methods）与 Lambda

Java 8 引入**默认方法（Default Method）**，允许在接口中定义带有方法体的方法：

- 默认方法不会影响 Lambda 表达式的语法或使用，但 Lambda 只能实现接口的抽象方法，**无法直接覆盖默认方法**。
- 在默认方法内部，可调用 Lambda 表达式或将其作为参数传递，进一步体现函数式编程的灵活性。

11.1.12 匿名类 vs Lambda 表达式的区别比较

	匿名类（Anonymous Class）	Lambda 表达式（Lambda Expression）
本质	无名内部类	无名方法（函数）
语法	new 接口() {…}	(参数)->表达式或代码块
适用接口	任意接口或抽象类	仅限功能性接口（一个抽象方法）
编译产物	每个匿名类生成单独.class 文件	转化为外部类的私有静态方法，无额外.class 文件
内存	运行时动态生成类型实例	常驻 JVM 方法区
代码简洁度	相对冗长	更加简洁、易读

12 Week11

12.1 Lecture

本周我们学习了 Java 泛型（Generics）中的通配符（Wildcards）。以下先展示课件中的完整示例代码，然后分别讲解三种通配符的定义与应用。专业词第一次出现时用中文 (英文) 表示。

12.1.1 完整示例代码（课件示例）

```
// Person 基类
class Person {
    String name;
    public Person(String name) { this.name = name; }
    @Override
    public String toString() { return name; }
}

// Employee 子类
class Employee extends Person {
    int salary;
    public Employee(String name, int salary) {
        super(name);
        this.salary = salary;
    }
    @Override
```

```
17      public String toString() { return "[" + name + "," + salary
            + "]"; }
18  }
19
20  // Customer 子类
21  class Customer extends Person {
22      int customerID;
23      public Customer(String name, int customerID) {
24          super(name);
25          this.customerID = customerID;
26      }
27      @Override
28      public String toString() { return "[" + name + "," +
            customerID + "]"; }
29  }
30
31  // 演示泛型不变性和通配符
32  public class GenericsWildcard {
33      // 泛型不变性示例
34      public static void readPeopleInvariant(List<Person> people)
            {
35          for (Person p : people) System.out.println(p);
36      }
37      // 上界通配符示例（只读）
38      public static void readPeople(List<? extends Person> people
            ) {
39          for (Person p : people) System.out.println(p);
40      }
41      // 下界通配符示例（只写）
42      public static void addEmployees(List<? super Employee> list
            , Employee e) {
43          list.add(e);
44      }
45
46      public static void main(String[] args) {
47          List<Employee> employees = new ArrayList<>();
48          employees.add(new Employee("Jeff", 76559));
49          employees.add(new Employee("Alice", 27584));
50          employees.add(new Employee("Kelly", 4332));
```

```
        // 不变性示例：编译错误
        // readPeopleInvariant(employees);
        // 通配符示例
        readPeople(employees); // ? extends Person; 只读
        List<Person> persons = new ArrayList<>();
        addEmployees(persons, new Employee("Bob", 50000)); // ? super Employee; 只写
    }
}
```

12.1.2 无界通配符（Unbounded Wildcard）

“无界”表示对元素类型不做任何限定。只能安全地读取为 Object，写入只能添加 null。

```
List<?> list = new ArrayList<String>();
Object obj = list.get(0);     % 读取当作 Object
// list.add("abc");        % 编译错误，无法添加具体元素
list.add(null);              % 允许添加 null
```

12.1.3 上界通配符（Upper-Bounded Wildcard）

使用? extends T 限定元素为 T 或其子类，适用于生产者（只读）场景，可安全地当作 T 读取，但写入除 null 外均不允许。

```
// 只读方法示例
public static void readPeople(List<? extends Person> people) {
    for (Person p : people) {
        System.out.println(p);
    }
}

List<Employee> emps = List.of(
    new Employee("Jeff", 76559),
    new Employee("Alice", 27584),
    new Employee("Kelly", 4332)
);
readPeople(emps);            % 允许传入 List<Employee>
// people.add(new Person("Tom")); % 编译错误：无法写入
```

12.1.4 下界通配符（Lower-Bounded Wildcard）

使用? super T 限定元素为 T 或其父类，适用于消费者（只写）场景，可安全地 add(T) 及其子类实例，但读取时只能当作 Object。

```
// 只写方法示例
public static void addBook(List<? super Book> list, Book b) {
    list.add(b);
}

List<Media> media = new ArrayList<>();
List<Book> books = new ArrayList<>();
addBook(media, new Book("Pride and Prejudice", 432));  % OK
addBook(books, new Book("War and Peace", 1225));       % OK
Object o = media.get(0);    % 读取当作 Object
```

12.1.5 Wildcard 特性与应用补充

- 泛型类型参数本身**不具备协变性（invariance）**：List<Employee> 绝非 List<Person> 的子类型。
- **PECS 原则**：
  - *Producer Extends*：生产者（只读）用 ? extends T；
  - *Consumer Super*：消费者（只写）用 ? super T。
- 在方法签名上用通配符可让 API 既安全又灵活：
  - ”只读”方法：List<? extends T> 接收任意 T 或子类列表；
  - ”只写”方法：List<? super T> 接收任意 T 或父类列表。
- 使用通配符列表时：
  - ? extends 列表只能读取；
  - ? super 列表只能写入；
  - 无界通配符 <?> 列表既不能写入具体对象，也只能读取为 Object。

12.1.6 运行时调试（Runtime Debugging）

本部分介绍 Java 自带的调试器 JDB（Java Debugger），用于在程序运行时定位逻辑错误。

编译准备

```
javac -g MyProgram.java   % -g 保留调试符号：变量名、行号等
```

启动调试会话

```
jdb MyProgram             % 启动 JDB，会话开启
```

常用 JDB 命令

- `stop at <Class>:<line>`：在指定类的行号设置断点。
- `run`：运行程序，命中断点时暂停。
- `locals`：显示当前方法所有局部变量及其值。
- `dump <var>`：打印对象内部状态（类似 toString）。
- `print <expr>`：计算并打印任意表达式。
- `step` / `next`：单步进入/跳过方法调用。
- `set <var>=<val>`：临时修改变量值，用于测试不同分支。
- `cont`：继续执行至下一个断点或程序结束。
- `quit`：退出调试会话。

调试流程示例

1. 编译：`javac -g MyProgram.java`。
2. 启动：`jdb MyProgram`。
3. 设置断点：`stop at MyProgram:32`。
4. 运行：`run`，遇断点暂停。

5. 检查：`locals`、`print x`、`dump obj`。

6. 控制：使用 `step`/`next` 观察执行流。

7. 修改：`set flag=false`，重新验证逻辑。

8. 继续或退出：`cont` / `quit`。

重点提醒 始终用 `-g` 编译；在可疑逻辑处打断点；结合变量观察和单步，快速锁定逻辑错误。