Java 程序设计精编教程（第 3 版）笔记

李弦哲

第 1 章　Java 入门

1.1　导读

- 主要内容：
  - Java 的平台无关性
  - Java 的地位
  - 安装 JDK
  - 一个简单的 Java 应用程序

1.2　Java 的平台无关性

- **Java 虚拟机（Java Virtual Machine）**：
  - Java 可以在操作系统之上提供一个 Java 运行环境，由 Java 虚拟机、类库及核心文件组成。
  - Java 编译器不针对特定操作系统和 CPU 芯片进行编译，而是将 Java 源程序编译为字节码。
  - **字节码**：一种中间代码，由 Java 虚拟机识别和执行。
  - Java 虚拟机将字节码翻译成当前平台的机器码并运行。
- **平台无关性**：
  - Java 生成的字节码文件不依赖平台，可以在不同平台（如 Windows、UNIX）上运行，只需对应的 Java 运行环境（JRE）。

1.3　Java 之父 - James Gosling

- **历史背景**：
  - 1990 年，Sun 公司成立了由 James Gosling 领导的开发小组，致力于开发可移植的跨平台语言。
  - 1995 年 5 月，Sun 公司推出 Java Development Kit(JDK) 1.0a2 版本，标志着 Java 的诞生。
  - 这个过程被形容为“有心栽花花不开，无心插柳柳成荫”。

1.4　Java 的地位

- **网络地位**：
  - Java 的平台无关性使其成为编写网络应用程序的佼佼者，提供了许多网络应用核心技术。
- **语言地位**：
  - Java 是面向对象编程语言，涉及网络、多线程等基础知识，是软件设计开发者应掌握的基础语言。
- **需求地位**：
  - IT 行业对 Java 人才的需求不断增长，许多新技术领域都涉及 Java 语言。

1.5 安装 JDK

- **Java 平台版本**：
  - Java SE（Java 标准版或标准平台）
  - Java EE（Java 企业版或企业平台）
- **虚拟机功能**：
  - 虚拟机将字节码文件加载到内存并采用解释方式执行，根据平台指令翻译并执行字节码文件。
- **JDK 版本更新**：
  - Oracle 收购 Sun 公司后，Java 更新频率从若干年一次变为每半年一次，大部分版本只有半年的支持期。
  - 截至 2023 年上半年，最新版本为 JDK20，实际生产环境中主要使用长期支持版本 JDK8、11、17。
- **JDK 安装步骤**：
  - 下载、安装 JDK1.8，建议修改默认安装路径为：E:\jdk1.8。
  - 设置环境变量 Path，添加 JDK 的 bin 目录路径。

1.6 Java 程序的开发步骤

1. **编写源文件**：
   - 扩展名为.java 的源文件。
2. **编译源程序**：
   - 使用 Java 编译器（javac.exe）编译源文件，生成字节码文件。
3. **运行程序**：
   - 使用 Java 解释器（java.exe）解释执行字节码文件。

1.7 一个简单的 Java 应用程序

- **例子 1：Hello 程序**：
  1. **编写源文件**：

```
public class Hello {
    public static void main (String args[]) {
        System.out.println("这是一个简单的Java应用程序");
    }
}
```

  - 文件命名为 Hello.java，保存至 C:\ch1。
  - 注意良好的编码习惯和命名规则。

2. **编译源程序**:

```
C:\ch1>javac Hello.java
```

3. **运行程序**:

```
C:\ch1>java Hello
```

第 2 章 Java 应用程序的基本结构

2.1 导读

- 主要内容:
  - 问题的提出
  - 简单的 Circle 类
  - 使用 Circle 类创建对象
  - 在 Java 应用程序中使用对象
  - Java 应用程序的基本结构
  - 编程风格

2.2 问题的提出

- **计算圆面积的 Java 应用程序**:
  - 如果多个应用程序都需要计算圆的面积，每个应用程序都需编写相同的代码。
  - **目标**：将与圆有关的数据及计算代码封装，使得其他应用程序无需重复编写计算代码。

2.3 简单的 Circle 类

- **面向对象的抽象**:
  - 抽象关键：数据和数据上的操作。
  - **Circle 类的定义**:

```
class Circle {
    double radius;
    double getArea() {
        double area = 3.14 * radius * radius;
        return area;
    }
}
```

  - 类声明和类体：Circle.java。
  - 类体内容包括：
    * **域变量（成员变量）**：描述圆的属性，如半径。
    * **方法**：描述行为，如计算圆面积的方法。

2.4 使用 Circle 类创建对象

- **类是 Java 中的数据类型**：
  - 创建对象需经过两个步骤：声明对象和为对象分配变量。

2.4.1 用类声明对象

- **类声明变量**：
  - 用类声明的变量即对象，例如：

```
Circle circleOne;
```

  - 声明后，circleOne 是一个空对象，需为其分配变量。

2.4.2 为对象分配变量

- **分配变量**：
  - 使用 new 运算符和构造方法，例如：

```
circleOne = new Circle();
```

  - 可以在声明时同时分配变量：

```
Circle circleOne = new Circle();
```

2.4.3 使用对象

- **操作对象变量**：
  - 使用 . 运算符操作对象的变量和调用方法，例如：

```
circleOne.radius = 100;
double area = circleOne.getArea();
```

2.5 在应用程序中使用对象

- **例子：Circle 类和 Example2_1 类**：
  - **Circle 类**：

```
class Circle {
    double radius;
    double getArea() {
        double area = 3.14 * radius * radius;
        return area;
    }
}
```

  - **Example2_1 类**：

```
public class Example2_1 {
    public static void main(String args[]) {
        Circle circleOne, circleTwo;
        circleOne = new Circle();
        circleTwo = new Circle();
        circleOne.radius = 123.86;
        circleTwo.radius = 69;
        double area = circleOne.getArea();
        System.out.println("circleOne的面积:" + area);
        area = circleTwo.getArea();
        System.out.println("circleTwo的面积:" + area);
    }
}
```

2.6 Java 应用程序的基本结构

- **Java 应用程序的组成**：
  - 由若干个类构成，但必须有一个包含 `main` 方法的主类。
  - 各类可存放在不同的源文件中或同一个源文件中。
- **例子**：
  - `Example2_2.java`、`Rect.java`、`Lader.java` 保存在 `C:\ch2` 中，`Example2_2.java` 为主类源文件。

2.7 在一个源文件中编写多个类

- **编写多个类**：
  - 一个源文件中可编写多个类，但只能有一个类使用 `public` 修饰。
  - **例子 3**：

    * 命名保存源文件为 `Rectangle.java`。
    * 编译后会生成两个字节码文件。
    * 运行时使用主类的名字。

2.8 编程风格

- **Allman 风格**:
  - 也称“独行”风格，左、右大括号各自独占一行。
- **Kernighan 风格**:
  - 也称“行尾”风格，左大括号在上一行的行尾，右大括号独占一行。
- **注释**:
  - 单行注释使用`//`。
  - 多行注释使用`/* ... */`。

第 3 章 标识符与简单数据类型

3.1 导读

- 主要内容:
  - 标识符与关键字
  - 简单数据类型
  - 简单数据类型的级别与类型转换
  - 从命令行窗口输入、输出数据

3.2 标识符与关键字

- **标识符**:
  - 用于标识类名、变量名、方法名、类型名、数组名、文件名的有效字符序列。
  - 简单来说，标识符就是一个名字。
- **关键字**:
  - 关键字是 Java 语言中已经被赋予特定意义的单词。
  - 关键字不能用作标识符。

3.3 简单数据类型

- **数据类型分类**:
  - 基本数据类型：整数、小数、字符、布尔型
  - 对象型数据：数组、由类生成的对象
- **基本数据类型的解释**:
  - 涉及字节数、编码格式、能参与的运算类型等。

3.3.1 逻辑类型

- **常量**：`true`, `false`
- **变量**：

```
boolean xok = true;
boolean 关闭 = false;
```

3.3.2 整数类型

1. **int 型**：
   - 常量：123, 6000（十进制），077（八进制），0x3ABC（十六进制）
   - 变量：

```
int x = 12;
int 平均 = 9898;
```

   - 占用 4 个字节（32 位）
2. **byte 型**：
   - 常量：无 byte 型常量表示法，可以用 int 型常量赋值给 byte 型变量
   - 变量：

```
byte x = -12;
byte tom = 28;
```

   - 占用 1 个字节（8 位）
3. **short 型**：
   - 常量：无 short 型常量表示法，可以用 int 型常量赋值给 short 型变量
   - 变量：

```
short x = 12;
short y = 1234;
```

   - 占用 2 个字节（16 位）
4. **long 型**：

- 常量：用后缀 L 表示

```
long width = 12L;
long height = 2005L;
```

- 占用 8 个字节（64 位）

3.3.3 字符类型

- **常量**：
  - 用单引号扩起的 Unicode 表中的一个字符，例如：'A', 'b', ' 好', '\t'
- **变量**：

```
char ch = 'A';
char home = '家';
```

- **转义字符**：
  - 例如：\n（换行），\b（退格），\t（水平制表），\'（单引号），\"（双引号），\\（反斜线）

3.3.4 浮点类型

1. **float 型**：
   - 常量：453.5439f, 2e40f
   - 变量：

```
float x = 22.76f;
float tom = 1234.987f;
```

   - 占用 4 个字节（32 位），精度保留 8 位有效数字
2. **double 型**：
   - 常量：2389.539d, 1e-90
   - 变量：

```
double height = 23.345;
double width = 34.56D;
```

   - 占用 8 个字节（64 位），精度保留 16 位有效数字

3.4 简单数据类型的级别与类型转换运算

- **数据类型精度排列**（不包括逻辑类型）：
  - byte < short < char < int < long < float < double
- **自动类型转换**：
  - 低级别变量赋值给高级别变量时，系统自动转换，例如：

```
float x = 100;
```

- **强制类型转换**：
  - 高级别变量赋值给低级别变量时，必须使用强制类型转换，例如：

```
int x = (int) 34.89;
```

- **范围超出处理**：
  - int 型常量超出 byte 和 short 型变量范围时，需进行类型转换，例如：

```
byte a = (byte) 128;  // 超出范围会导致精度损失
```

3.5 从命令行输入、输出数据

3.5.1 输入基本型数据

- **使用 Scanner 类**：

```
Scanner reader = new Scanner(System.in);
```

- **读取用户输入**：
  - `nextBoolean()`, `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`

3.5.2 输出基本型数据

- **使用 System.out**：
  - `System.out.println()`: 输出后换行
  - `System.out.print()`: 输出后不换行
- **并置符号**：

  - 使用 + 将变量、表达式或常数值与字符串并置输出，例如：

```
System.out.println(m + "个数的和为" + sum);
```

- **格式化输出**：
  - JDK1.5 新增 printf 方法，类似 C 语言中的 printf，例如：

```
System.out.printf("%d %f %s", 123, 456.789, "Hello");
```

第 4 章　运算符、表达式与语句

4.1　导读

- 主要内容：
  - 运算符与表达式
  - 语句概述
  - if 条件分支语句
  - switch 开关语句
  - 循环语句
  - break 和 continue 语句
  - 数组

4.2　运算符与表达式

- **Java 提供了丰富的运算符**：
  - 算术运算符、关系运算符、逻辑运算符、位运算符等。

4.2.1　算术运算符与算术表达式

1. **加减运算符**：+，-
   - 二目运算符，结合方向从左到右，操作元为整型或浮点型数据，优先级为 4 级。
2. **乘、除和求余运算符**：*，/，%
   - 二目运算符，结合方向从左到右，操作元为整型或浮点型数据，优先级为 3 级。
3. **算术表达式**：
   - 用算术符号和括号连接起来的符合 Java 语法规则的式子。

4.2.2 自增，自减运算符

- **自增、自减运算符**：++，--
  - 单目运算符，可以放在操作元之前或之后，操作元必须是整型或浮点型变量。
  - ++x 表示在使用 x 之前，先使 x 的值增 1。
  - x++ 表示在使用 x 之后，使 x 的值增 1。

4.2.3 算术混合运算的精度

- **精度从低到高排列顺序**：byte, short, char, int, long, float, double
- **计算规则**：
  1. 有 double 型数据时，按 double 精度运算。
  2. 有 float 型数据时，按 float 精度运算。
  3. 有 long 型数据时，按 long 精度运算。
  4. 精度低于 int 型整数时，按 int 精度运算。

4.2.4 关系运算符与关系表达式

- **关系运算符**：
  - 二目运算符，用来比较两个值的关系，结果为 boolean 型，成立时结果为 true，否则为 false。

4.2.5 逻辑运算符与逻辑表达式

- **逻辑运算符**：&&，||，!
  - && 和 || 为二目运算符，实现逻辑与、逻辑或。
  - ! 为单目运算符，实现逻辑非。
  - 操作元必须是 boolean 型数据，逻辑运算符可连接关系表达式。

4.2.6 赋值运算符与赋值表达式

- **赋值运算符**：=
  - 二目运算符，左操作元必须是变量，优先级较低（14 级），结合方向从右到左。

4.2.7 位运算符

- **位运算符**：
  1. “按位与”运算：&
  2. “按位或”运算：|
  3. “按位非”运算：~
  4. “按位异或”运算：^
- 参与运算的为整型数据，结果也是整型数据。

4.2.8 instanceof 运算符

- **instanceof 运算符**:
  - 二目运算符，左操作元是对象，右操作元是类，当对象是该类或其子类创建时，结果为 true，否则为 false。

4.2.9 运算符综述

- **表达式**:
  - 用运算符连接的符合 Java 规则的式子。
  - 运算符的优先级和结合性决定表达式中运算的先后顺序。
  - 推荐使用括号来明确运算次序。

4.3 语句概述

- **Java 语句的分类**:
  1. **方法调用语句**：如 `System.out.println("Hello");`
  2. **表达式语句**：如 `x=23;`
  3. **复合语句**：用 `{}` 括起来的语句块
  4. **空语句**：单个分号
  5. **控制语句**：条件分支、开关语句、循环语句
  6. **package 语句和 import 语句**：将在第 5 章讲解

4.4 if 条件分支语句

- **if 语句**:
  - 单条件分支语句，根据一个条件控制程序执行流程。

4.4.1 if 语句

- **语法格式**:

```
if (表达式) {
    若干语句
}
```

4.4.2 if-else 语句

- **语法格式**:

```
if (表达式) {
    若干语句
} else {
    若干语句
}
```

4.4.3 if-else if-else 语句

- **语法格式**:

```
if (表达式) {
    若干语句
} else if (表达式) {
    若干语句
} else {
    若干语句
}
```

4.5 switch 开关语句

- **switch 语句**:
  - 单条件多分支语句，语法格式如下:

```
switch (表达式) {
    case 常量值1:
        若干个语句
        break;
    case 常量值2:
        若干个语句
        break;
    // ...
    default:
        若干语句
}
```

4.6 循环语句

- **循环语句**:
  - 包括 for 循环、while 循环和 do-while 循环。

4.6.1 for 循环语句

- **语法格式**:

```
for (表达式1; 表达式2; 表达式3) {
    若干语句
}
```

- **执行规则**:

1. 计算表达式 1，完成初始化。
2. 判断表达式 2，若为 true，则执行循环体，再计算表达式 3。
3. 若表达式 2 为 false，则结束 for 语句。

4.6.2 while 循环

- **语法格式**:

```
while (表达式) {
    若干语句
}
```

- **执行规则**:

1. 计算表达式的值，若为 true，则执行循环体，再计算表达式。
2. 若为 false，则结束 while 语句。

4.6.3 do-while 循环

- **语法格式**:

```
do {
    若干语句
} while (表达式);
```

- **执行规则**:

1. 执行循环体，再计算表达式的值。
2. 若为 true，则继续执行循环体，否则结束 do-while 语句。

4.7 break 和 continue 语句

- **break 语句**:
  - 用于结束整个循环语句。

- **continue 语句**：
  - 用于结束本次循环，进入下一次循环。

4.8 数组

- **数组**：
  - 相同类型变量按顺序组成的复合数据类型，称为数组的元素或单元。
  - 数组通过数组名加索引使用元素，创建数组需声明和分配变量两个步骤。

4.8.1 声明数组

- **一维数组声明格式**：

```
元素类型 数组名[];
元素类型[] 数组名;
```

- **二维数组声明格式**：

```
元素类型 数组名[][];
元素类型[][] 数组名;
```

4.8.2 为数组分配元素

- **分配内存空间格式**：

```
数组名 = new 元素类型[元素个数];
```

4.8.3 数组元素的使用

- **一维数组**：
  - 通过索引符访问元素，如 `boy[0]`。
- **二维数组**：
  - 通过索引符访问元素，如 `a[0][1]`。

4.8.4 length 的使用

- **数组长度**：
  - 一维数组：数组名`.length`
  - 二维数组：数组名`.length` 为包含的一维数组个数。

4.8.5 数组的初始化

- **声明时初始化**:

```
float boy[] = {21.3f, 23.89f, 2.0f, 23f, 778.98f};
```

- **二维数组初始化**:

```
int a[][] = {{1}, {11}, {121}, {1331}, {14641}};
```

4.8.6 数组的引用

- **数组属于引用型变量**:
  - 两个相同类型的数组具有相同引用时，它们有完全相同的元素。

```
int a[] = {1, 2, 3};
int b[];
b = a;
```

4.8.7 遍历数组

1. **基于循环语句的遍历**:

```
for (声明循环变量 : 数组名) {
    // 语句
}
```

2. **使用 toString() 方法遍历数组**:

```
int[] a = {1, 2, 3, 4, 5, 6};
System.out.println(Arrays.toString(a));
```

第 5 章 类与对象

5.1 导读

- 主要内容:
  - 面向对象的特性

- 类
- 构造方法与对象的创建
- 参数传值
- 对象的组合
- 实例成员与类成员
- 方法重载与多态
- this 关键字
- 包
- import 语句，访问权限
- 可变参数

5.2 面向对象的特性

- **面向对象编程的四个特性**:
  - 抽象
  - 封装
  - 继承
  - 多态

5.3 类

- **类的定义**:
  - 类是 Java 程序的基本要素，封装对象的状态和方法，是定义对象的模板。
  - 类的实现包括类声明和类体，格式如下:

```
class 类名 {
    类体的内容
}
```

5.3.1 类声明

- **类声明格式**:
  - `class 类名`
  - 类名首字母大写，容易识别，见名知意。

5.3.2 类体

- **类体内容**:
  - 由变量定义（刻画属性）和方法定义（刻画功能）构成。

5.3.3 成员变量和局部变量

- **成员变量**:
  - 在类体中声明，整个类内有效。
- **局部变量**:
  - 在方法体中声明，仅在方法内有效。
- **隐藏规则**:
  - 局部变量名与成员变量名相同时，成员变量在方法内暂时失效。

5.3.4 方法

- **方法定义**:
  - 包括方法声明和方法体。
  - 方法声明部分包含方法名和返回类型，格式如下:

```
返回类型 方法名() {
    方法体的内容
}
```

5.3.5 注意事项

- **成员变量操作**:
  - 只能在方法中操作成员变量，不可在类体中单独赋值。

5.3.6 类的 UML 图

- **UML 图**:
  - 用于描述系统静态结构，包括类图、接口图、泛化关系、关联关系、依赖关系、实现关系。
  - 类图包括名字层、变量层（属性层）、方法层（操作层）。

5.4 构造方法与对象的创建

- **对象创建**:
  - 类声明的变量称为对象，需通过构造方法创建对象，格式如下:

```
类名 对象名 = new 类名();
```

5.4.1 构造方法

- **构造方法**:
  - 特殊方法，名字与类名相同，无返回类型。
  - 一个类可有多个构造方法，但参数必须不同。

5.4.2 创建对象

- **步骤**:
  1. 声明对象：`Lader lader;`
  2. 分配内存：`lader = new Lader();`

5.4.3 使用对象

- **操作对象**:
  - 使用 . 操作变量和调用方法。

5.4.4 对象的引用和实体

- **对象的引用**:
  - 对象存放着引用，引用相同的对象具有相同的实体。

5.5 参数传值

- **参数传值**:
  - 方法的参数为局部变量，调用方法时，参数变量必须有具体值。

5.5.1 传值机制

- **传值**:
  - 所有参数为传值，传递的是值的拷贝。

5.5.2 基本数据类型参数的传值

- **基本数据类型传值**:
  - 传递的值级别不高于参数级别。

5.5.3 引用类型参数的传值

- **引用类型传值**:
  - 传递的是变量中的引用，而不是实体。

5.6 对象的组合

- **对象组合**:
  - 一个类可将对象作为成员变量，创建的对象包含其他对象。

5.7 实例成员与类成员

- **实例变量和类变量**:
  - 实例变量不加 `static` 修饰，类变量加 `static` 修饰。

5.7.1 实例变量和类变量的声明

- **声明格式**:
  - 实例变量不加 `static`，类变量加 `static`。

5.7.2 实例变量和类变量的区别

1. 不同对象的实例变量互不相同。
2. 所有对象共享类变量。
3. 类变量可通过类名直接访问。

5.7.3 实例方法和类方法的定义

- **方法定义**:
  - 实例方法不加 `static`，类方法加 `static`。

5.7.4 实例方法和类方法的区别

1. **对象调用实例方法**:
   - 实例方法操作实例变量。
2. **类名调用类方法**:
   - 类方法不操作实例变量。

5.8 方法重载与多态

- **方法重载**:
  - 一个类中多个方法具有相同名字，但参数不同。

5.9 this 关键字

- **this 关键字**:
  - 表示某个对象，可在实例方法和构造方法中使用。

5.9.1 在构造方法中使用 this

- **使用方式**:
  - 表示使用该构造方法创建的对象。

5.9.2 在实例方法中使用 this

- **使用方式**:
  - 表示调用该方法的当前对象。

5.10 包

- **包机制**:
  - 用于管理类，区分名字相同的类。

5.10.1 包语句

- **声明格式**:

```
package 包名;
```

5.10.2 有包名的类的存储目录

- **目录结构**:
  - 存储文件的目录结构需包含包名结构。

5.10.3 运行有包名的主类

- **运行格式**:

```
java 包名.主类名
```

5.11 import 语句

- **引入类**:
  - 使用 import 语句引入不同包中的类。

5.11.1 引入类库中的类

- **引入格式**:

```
import java.util.Date;
```

5.11.2 引入自定义包中的类

- **引入格式**:

```
import 包名.*;
```

5.12 访问权限

- **访问权限**:
  - 对象是否可以操作自己的变量或使用类中的方法。

5.12.1 何谓访问权限

- **访问修饰符**:
  - private、protected、public 用于修饰成员变量或方法。

5.12.2 私有变量和私有方法

- **私有修饰符**:
  - private 修饰的成员变量和方法，仅在本类中访问。

5.12.3 共有变量和共有方法

- **公共修饰符**:
  - public 修饰的成员变量和方法，可在任何类中访问。

5.12.4 友好变量和友好方法

- **友好访问**:
  - 在同一包中的类中访问。

5.12.5 受保护的成员变量和方法

- **保护修饰符**:
  - protected 修饰的成员变量和方法，可在同一包和子类中访问。

5.12.6 public 类与友好类

- **类修饰符**:
  - public 类在任何类中使用，不加 public 为友好类，仅在同一包中使用。

5.13 基本类型的类包装

- **类包装**:
  - Java 提供类包装基本数据类型，如 Byte、Integer、Short、Long、Float、Double 和 Character。

5.13.1 Double 和 Float 类

- **类包装**:
  - Double 和 Float 类包装 double 和 float 类型，提供相应的方法获取基本数据类型值。

5.13.2 Byte、Short、Integer、Long 类

- **类包装**:
  - Byte、Short、Integer、Long 类包装对应基本数据类型，提供相应方法获取值。

5.13.3 Character 类

- **类包装**:
  - Character 类包装 char 类型，提供方法判断字符属性和转换字符大小写。

5.14 可变参数

- **可变参数**:
  - 方法参数列表中从某项至最后一项参数数量不固定，格式为类型... 参数名。

第 6 章 子类与继承

6.1 导读

- 主要内容:
  - 子类与超类
  - 使用 super 关键字
  - final 修饰符
  - 向上转型
  - 继承层次
  - 继承的优点和缺点

6.2 子类与超类

- **继承**:
  - 通过扩展已有的类，创建新类。新类称为子类或派生类，已有的类称为超类或基类。
  - 继承允许子类继承超类的属性和方法。

6.2.1 继承的实现

- **定义子类**:
  - 使用 `extends` 关键字定义子类，格式如下:

```
class 子类 extends 超类 {
    // 子类的特有成员变量
    // 子类的特有方法
}
```

6.2.2 继承的特点

- 子类拥有超类的所有成员变量和方法，但不包括超类的构造方法。
- 子类可以添加新的成员变量和方法，也可以重写超类的方法。

6.3 使用 super 关键字

- **super 关键字**:
  - 用于调用超类的构造方法和超类的成员变量和方法。

6.3.1 调用超类构造方法

- **格式**:

```
super(参数列表);
```

- **注意**:
  - 必须在子类构造方法的第一行调用超类的构造方法。

6.3.2 调用超类成员变量和方法

- **格式**:

```
super.成员变量;
super.方法名(参数列表);
```

6.4 final 修饰符

- **final 类**:
  - 不能被继承的类，用 `final` 关键字修饰。
- **final 方法**:
  - 不能被子类重写的方法，用 `final` 关键字修饰。
- **final 变量**:
  - 值不能改变的变量，用 `final` 关键字修饰。

6.5 向上转型

- **向上转型**:
  - 将子类对象的引用赋给超类变量。
  - 格式:

```
超类 对象名 = new 子类();
```

- **特点**:
  - 只能访问超类中的成员变量和方法，不能访问子类的特有成员变量和方法。

6.6 继承层次

- **继承层次**:
  - 多层继承形成继承层次结构，最顶层的类称为根类。
  - Java 中所有类最终继承自 `Object` 类。

6.7 继承的优点和缺点

- **优点**:
  - 提高代码的复用性。
  - 提高代码的可维护性。
  - 提供多态性。
- **缺点**:
  - 增加了类之间的耦合性。
  - 继承层次过深，会使系统复杂化。

第 7 章 接口与实现

7.1 导读

- 主要内容:
  - 接口
  - 实现接口
  - 理解接口
  - 接口回调
  - 接口与多态
  - 接口变量做参数
  - 面向接口编程

7.2 接口

- **接口概念**:
  - 接口克服了 Java 单继承的限制，一个类可以实现多个接口。
  - 使用 `interface` 关键字定义接口，格式如下:

```
interface 接口名 {
    final int MAX = 100;
    void add();
    float sum(float x, float y);
}
```

  - 接口包含常量定义和方法定义。

7.3 实现接口

- **实现接口**:

- 一个类通过 `implements` 关键字声明实现一个或多个接口，例如：

```
class A implements Printable, Addable {
    ...
}
```

- 类实现接口时，必须重写接口的所有方法。
- Java 提供的接口在相应包中，通过 `import` 语句引入包中的类和接口，例如：

```
import java.io.*;
```

7.4 理解接口

- **接口功能**：
  - 接口增加许多类需要具有的功能，不同类可以实现相同接口，同一类也可以实现多个接口。
  - 接口只关心操作，不关心具体实现。

7.5 接口的 UML 图

- **接口 UML 图**：
  - 类似类的 UML 图，使用一个长方形描述接口的主要构成，分为名字层、常量层、方法层。

7.6 接口回调

- **接口回调**：
  - 可以将实现接口的类创建的对象引用赋给接口变量，接口变量可以调用被类重写的方法。
  - 当接口变量调用重写的方法时，通知相应对象调用该方法。

7.7 接口与多态

- **接口与多态**：
  - 接口中声明若干个抽象方法，具体实现由实现接口的类完成。
  - 接口回调的核心思想是接口变量存放实现接口类的对象引用，从而回调类实现的方法。

7.8 接口变量做参数

- **接口参数**：
  - 方法的参数是接口类型时，可以传递实现该接口的类实例的引用，接口参数可以回调类实现的方法。

7.9 abstract 类与接口的比较

- **比较**:
  1. 抽象类和接口都可以有抽象方法。
  2. 接口中只可以有常量，不能有变量；抽象类中即可以有常量也可以有变量。
  3. 抽象类中可以有非抽象方法，接口不可以。

7.10 面向接口编程

- **面向接口编程**:
  - 在接口中声明若干个抽象方法，方法实现由实现接口的类完成。
  - 核心思想是使用接口回调，接口变量存放实现接口类的对象引用，从而回调类实现的方法。

第 8 章 内部类与异常类

8.1 导读

- 主要内容:
  - 内部类
  - 匿名类
  - 异常类
  - 断言

8.2 内部类

- **内部类定义**:
  - Java 支持在一个类中声明另一个类，这样的类称为内部类，包含内部类的类称为外嵌类。
  - 内部类的类体中不可以声明类变量和类方法。
  - 外嵌类的类体中可以用内部类声明对象作为外嵌类的成员。
  - 内部类仅供它的外嵌类使用，其他类不能用某个类的内部类声明对象。

8.3 匿名类

8.3.1 和子类有关的匿名类

- **匿名类定义**:
  - Java 允许直接使用一个类的子类的类体创建一个子类对象。
  - 创建子类对象时，使用父类的构造方法外加类体，称为匿名类。
  - 例如:

```
new Bank() {
    // 匿名类的类体
};
```

8.3.2 和接口有关的匿名类

- **接口匿名类**:
  - Java 允许直接用接口名和类体创建一个匿名对象，称为匿名类。
  - 例如:

```
new Computable() {
    // 实现接口的匿名类的类体
};
```

8.4 异常类

- **异常定义**:
  - 程序运行时可能出现错误，异常处理改变程序控制流程，让程序对错误进行处理。
  - 异常对象方法:
    1. `getMessage()`
    2. `printStackTrace()`
    3. `toString()`

8.4.1 try-catch 语句

- **异常处理语句**:
  - Java 使用 `try-catch` 语句处理异常，格式如下:

```
try {
    // 可能发生异常的语句
} catch(ExceptionSubClass1 e) {
    // 异常处理
} catch(ExceptionSubClass2 e) {
    // 异常处理
}
```

8.4.2 自定义异常类

- **自定义异常**:
  - 扩展 `Exception` 类定义自己的异常类。
  - 方法在声明时使用 `throws` 关键字声明要产生的异常，在方法体中用 `throw` 抛出异常对象。

8.4.3 finally 子语句

- **finally 语句**:

- try-catch 语句可以带 finally 子语句，格式如下：

```
try {
    // 可能发生异常的语句
} catch(ExceptionSubClass e) {
    // 异常处理
} finally {
    // 无论是否发生异常都执行
}
```

- 执行机制：无论 try 部分是否发生异常，finally 子语句都会执行。
- 特殊情况：try-catch 中执行 return 语句，finally 仍然执行；执行 System.exit(0) 不执行 finally。

8.5 断言

- **断言语句**：
  - 用于调试代码阶段，发现致命错误。
  - 使用 assert 关键字声明断言语句，有两种格式：

```
assert booleanExpression;
assert booleanExpression: messageException;
```

第 9 章 常用实用类

9.1 导读

- 主要内容：
  - String 类
  - StringBuffer 类
  - StringTokenizer 类
  - Date 类
  - Calendar 类
  - Math 与 BigInteger 类
  - DecimalFormat 类
  - Pattern 与 Matcher 类
  - Scanner 类

9.2 String 类

- **String 类概述**：
  - 位于 java.lang 包中，用于创建字符串变量，字符串变量是对象。

9.2.1 构造字符串对象

1. **常量对象**：
   - 用双引号括起的字符序列，如：” 你好”、”12.97”、”boy” 等。
2. **字符串对象**：
   - 声明：`String s;`
   - 构造方法：

```
String(s)
String(char a[])
String(char a[], int startIndex, int count)
```

3. **引用字符串常量对象**：

```
String s1 = "How are you";
String s2 = "How are you";
```

9.2.2 String 类的常用方法

1. **获取长度**：`public int length()`
2. **比较字符串**：`public boolean equals(String s)`
3. **前缀、后缀判断**：
   - `public boolean startsWith(String s)`
   - `public boolean endsWith(String s)`
4. **按字典序比较**：
   - `public int compareTo(String s)`
   - `public int compareToIgnoreCase(String s)`
5. **判断是否包含子字符串**：`public boolean contains(String s)`
6. **检索子字符串位置**：
   - `public int indexOf(String s)`
   - `public int indexOf(String s, int startpoint)`
   - `public int lastIndexOf(String s)`
7. **获取子字符串**：
   - `public String substring(int startpoint)`
   - `public String substring(int start, int end)`
8. **去除前后空格**：`public String trim()`

9.2.3 字符串与基本数据的相互转化

- **字符串转化为基本数据类型**:
  - `Integer.parseInt(String s)` 将字符串转化为 int 型数据
  - 类似方法存在于 `Byte`、`Short`、`Long`、`Float`、`Double` 类中。
- **基本数据类型转化为字符串**:
  - `String.valueOf(byte n)` 等方法。
  - `Long.toBinaryString(long i)` 等方法得到整数的各种进制字符串表示。

9.2.4 对象的字符串表示

- **toString() 方法**:
  - `Object` 类中的 `public String toString()` 方法返回对象的字符串表示。

9.2.5 字符串与字符、字节数组

1. **字符串与字符数组**:
   - 构造方法：`String(char[])` 和 `String(char[], int offset, int length)`
   - 将字符串存放到字符数组中:`public void getChars(int start, int end, char c[], int offset)`
   - 将字符串中的全部字符存放在一个字符数组中：`public char[] toCharArray()`
2. **字符串与字节数组**:
   - 构造方法：`String(byte[])` 和 `String(byte[], int offset, int length)`
   - 将字符串转化为字节数组：`public byte[] getBytes()`
   - 使用指定字符编码转化为字节数组：`public byte[] getBytes(String charsetName)`

9.2.6 正则表达式及字符串的替换与分解

1. **正则表达式**:
   - 使用 `public boolean matches(String regex)` 方法判断当前字符串是否与指定正则表达式匹配。
2. **字符串的替换**:
   - 使用 `public String replaceAll(String regex, String replacement)` 方法将匹配正则表达式的子字符串替换。
3. **字符串的分解**:
   - 使用 `public String[] split(String regex)` 方法分解字符串。

9.3 StringBuffer 类

9.3.1 StringBuffer 对象的创建

- **构造方法**:
  - `StringBuffer()`
  - `StringBuffer(int size)`
  - `StringBuffer(String s)`

9.3.2 StringBuffer 类的常用方法

1. **追加字符串**：`StringBuffer append(String s)`
2. **获取单个字符**：`public char charAt(int n)`
3. **替换字符**：`public void setCharAt(int n, char ch)`
4. **插入字符串**：`StringBuffer insert(int index, String str)`
5. **翻转字符串**：`public StringBuffer reverse()`
6. **删除子字符串**：`StringBuffer delete(int startIndex, int endIndex)`
7. **替换子字符串**：`StringBuffer replace(int startIndex, int endIndex, String str)`

9.4 StringTokenizer 类

- **构造方法**：
  - `StringTokenizer(String s)`
  - `StringTokenizer(String s, String delim)`
- **方法**：
  1. `nextToken()`
  2. `hasMoreTokens()`
  3. `countTokens()`

9.5 Date 类

9.5.1 构造 Date 对象

1. **无参数构造方法**：

```
Date nowTime = new Date();
```

2. **有参数构造方法**：

```
Date date1 = new Date(1000);
Date date2 = new Date(-1000);
```

9.5.2 日期格式化

- **使用 SimpleDateFormat**：

```
SimpleDateFormat format = new SimpleDateFormat("pattern");
format(Date date)
```

9.6 Calendar 类

1. **初始化日历对象**:

```
Calendar calendar = Calendar.getInstance();
```

2. **设置时间**:

```
public final void set(int year, int month, int date)
public final void set(int year, int month, int date, int hour, int
    minute)
public final void set(int year, int month, int date, int hour, int
    minute, int second)
```

3. **获取时间信息**:

```
public int get(int field)
```

4. **获取毫秒表示**:

```
public long getTimeInMillis()
```

9.7 Math 和 BigInteger 类

9.7.1 Math 类

- **常用类方法**:

1. `public static long abs(double a)`
2. `public static double max(double a, double b)`
3. `public static double min(double a, double b)`
4. `public static double random()`
5. `public static double pow(double a, double b)`
6. `public static double sqrt(double a)`
7. `public static double log(double a)`
8. `public static double sin(double a)`
9. `public static double asin(double a)`

9.7.2 BigInteger 类

- **构造方法**：`public BigInteger(String val)`
- **常用类方法**：
  1. `public BigInteger add(BigInteger val)`
  2. `public BigInteger subtract(BigInteger val)`
  3. `public BigInteger multiply(BigInteger val)`
  4. `public BigInteger divide(BigInteger val)`
  5. `public BigInteger remainder(BigInteger val)`
  6. `public int compareTo(BigInteger val)`
  7. `public BigInteger abs()`
  8. `public BigInteger pow(int a)`
  9. `public String toString()`
  10. `public String toString(int p)`

9.8 DecimalFormat 类

9.8.1 格式化数字

- **格式化整数位和小数位**：`DecimalFormat format = new DecimalFormat("pattern");`
- **格式化为百分数或千分数**：在模式尾加% 或\u2030
- **格式化为科学计数**：在模式尾加 E0
- **格式化为货币值**：在模式尾加货币符号（如 $、\ $)

9.8.2 将格式化字符串转化为数字

- **将格式化字符串转化为数字**：`DecimalFormat df = new DecimalFormat("pattern");`
- **解析字符串**：`Number num = df.parse("string");`
- **获取数值**：`double d = num.doubleValue();`

9.9 Pattern 与 Matcher 类

9.9.1 模式对象

- **创建模式对象**：`Pattern p = Pattern.compile("regex");`
- **使用 flags 创建模式对象**：`Pattern.compile(String regex, int flags)`

9.9.2 匹配对象

- **创建匹配对象**：`Matcher m = p.matcher(CharSequence input);`
- **常用方法**：
  1. `public boolean find()`
  2. `public boolean matches()`

3. `public boolean lookingAt()`
4. `public boolean find(int start)`
5. `public String replaceAll(String replacement)`
6. `public String replaceFirst(String replacement)`

9.10 Scanner 类

1. **默认分隔标记解析字符串**:

```
Scanner scanner = new Scanner("string");
```

2. **使用正则表达式作为分隔标记解析字符串**:

```
Scanner scanner = new Scanner("string").useDelimiter("regex");
```

第 10 章 输入流与输出流

10.1 导读

- 主要内容:
  - 字节流与字符流
  - 缓冲流
  - 随机流
  - 数组流
  - 数据流
  - 对象流
  - 序列化与对象克隆
  - 文件锁
  - 使用 Scanner 解析文件

10.2 概述

- **输入、输出流**:
  - 提供一条通道用于读取源数据或将数据传送到目的地。
  - 输入流指向源，程序从输入流中读取数据。
  - 输出流指向目的地，程序通过输出流写入数据。

10.3 File 类

10.3.1 File 对象

- **用于获取文件信息，不涉及读写操作**。
- 构造方法：

```
File(String filename);
File(String directoryPath, String filename);
File(File f, String filename);
```

10.3.2 文件的属性

- **获取文件信息的方法**：
  1. `getName()`: 获取文件名
  2. `canRead()`: 判断文件是否可读
  3. `canWrite()`: 判断文件是否可写
  4. `exists()`: 判断文件是否存在
  5. `length()`: 获取文件长度（字节）
  6. `getAbsolutePath()`: 获取绝对路径
  7. `getParent()`: 获取父目录
  8. `isFile()`: 判断是否为文件
  9. `isDirectory()`: 判断是否为目录
  10. `isHidden()`: 判断是否为隐藏文件
  11. `lastModified()`: 获取最后修改时间

10.3.3 目录

- **创建目录**：

```
mkdir(): 创建目录
```

- **列出目录中的文件**：

```
list(): 以字符串形式返回目录下所有文件
listFiles(): 以File对象形式返回目录下所有文件
list(FilenameFilter obj): 返回目录下指定类型文件
listFiles(FilenameFilter obj): 返回目录下指定类型文件
```

10.3.4 文件的创建与删除

- **创建文件**:

```
createNewFile(): 创建新文件
```

- **删除文件**:

```
delete(): 删除文件
```

10.3.5 运行可执行文件

- **使用 Runtime 类**:

```
Runtime.getRuntime().exec(String command): 打开可执行文件或执行操作
```

10.4 字节流与字符流

- **字节流**:
  - InputStream 类: 字节输入流的父类
  - OutputStream 类: 字节输出流的父类

10.4.1 InputStream 类与 OutputStream 类

- **InputStream 类常用方法**:

```
read(): 读取字节
read(byte b[]): 读取字节数组
read(byte b[], int off, int len): 读取指定长度字节数组
close(): 关闭流
skip(long numBytes): 跳过字节
```

- **OutputStream 类常用方法**:

```
write(int n): 写入字节
write(byte b[]): 写入字节数组
write(byte b[], int off, int len): 写入指定长度字节数组
close(): 关闭流
```

10.4.2 Reader 类与 Writer 类

- **Reader 类常用方法**:

```
read(): 读取字符
read(char b[]): 读取字符数组
read(char b[], int off, int len): 读取指定长度字符数组
close(): 关闭流
skip(long numBytes): 跳过字符
```

- **Writer 类常用方法**:

```
write(int n): 写入字符
write(char b[]): 写入字符数组
write(char b[], int off, int len): 写入指定长度字符数组
close(): 关闭流
```

10.4.3 关闭流

- **关闭流**:
  - 调用 `close()` 方法显式关闭流，保证缓冲区内容写到目的地。

10.5 文件字节流

10.5.1 文件字节输入流

- **创建文件字节输入流**:

```
FileInputStream(String name)
FileInputStream(File file)
```

- **读取方法**:

```
int read()
int read(byte b[])
int read(byte b[], int off, int len)
```

10.5.2 文件字节输出流

- **创建文件字节输出流**:

```
FileOutputStream(String name)
FileOutputStream(File file)
```

- **写入方法**:

```
void write(byte b[])
void write(byte b[], int off, int len)
```

10.6 文件字符流

- **文件字符输入流和输出流**:

```
FileReader(String filename)
FileReader(File filename)
FileWriter(String filename)
FileWriter(File filename)
```

10.7 缓冲流

- **BufferedReader 和 BufferedWriter**:
  - 构造方法:

```
BufferedReader(Reader in)
BufferedWriter(Writer out)
```

  - 方法:

```
readLine()
write(String s, int off, int len)
newLine()
```

10.8 随机流

- **RandomAccessFile 类**:
  - 构造方法:

```
RandomAccessFile(String name, String mode)
RandomAccessFile(File file, String mode)
```

- 方法:

```
seek(long a)
getFilePointer()
```

10.9 数组流

10.9.1 字节数组流

- **ByteArrayInputStream 和 ByteArrayOutputStream**:
  - 构造方法:

```
ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf, int offset, int length)
ByteArrayOutputStream()
ByteArrayOutputStream(int size)
```

  - 方法:

```
read()
read(byte[] b, int off, int len)
write(int b)
write(byte[] b, int off, int len)
toByteArray()
```

10.9.2 字符数组流

- **CharArrayReader 和 CharArrayWriter**

10.10 数据流

- **DataInputStream 和 DataOutputStream**:
  - 构造方法:

```
DataInputStream(InputStream in)
DataOutputStream(OutputStream out)
```

10.11 对象流

- **ObjectInputStream 和 ObjectOutputStream**:
  - 构造方法:

```
ObjectInputStream(InputStream in)
ObjectOutputStream(OutputStream out)
```

  - 方法:

```
writeObject(Object obj)
readObject()
```

10.12 序列化与对象克隆

- **序列化**:
  - 实现 `Serializable` 接口
- **对象克隆**:
  - 调用 `clone()` 方法

10.13 文件锁

- **文件锁功能**:
  - 使用 FileLock、FileChannel 类
  - 步骤:
    1. 使用 RandomAccessFile 流建立指向文件的流对象
    2. 调用 `getChannel()` 方法获得 FileChannel 对象
    3. 调用 `tryLock()` 或 `lock()` 方法获得 FileLock 对象

10.14 使用 Scanner 解析文件

1. **默认分隔标记解析文件**:

```
Scanner sc = new Scanner(new File("filename"));
```

2. **使用正则表达式作为分隔标记解析文件**:

```
Scanner sc = new Scanner(new File("filename")).useDelimiter("regex");
```

10.15 总结

- 本章详细介绍了 Java 中的输入流与输出流，包括文件操作、字节流与字符流、缓冲流、随机流、数组流、数据流、对象流、序列化与对象克隆、文件锁以及使用 Scanner 解析文件。这些内容是 Java I/O 编程的基础，掌握这些知识可以有效地进行文件和数据的读写操作。

第 11 章 组件及事件处理

11.1 导读

- 主要内容:
  - Java Swing 概述
  - 窗口
  - 常用组件与布局
  - 处理事件
  - 使用 MVC 结构
  - 对话框
  - 发布 GUI 程序

11.2 Java Swing 概述

- **Java 的 java.awt 包**:
  - 提供了许多用来设计 GUI 的组件类。

11.3 窗口

11.3.1 JFrame 类

- **JFrame 类的实例是一个底层容器，其他组件必须被添加到底层容器中，以便借助这个底层容器和操作系统进行信息交互**。
- JFrame 类是 Container 类的间接子类。

11.3.2 JFrame 常用方法

- **常用方法**:

```
JFrame(): 创建一个无标题的窗口。
JFrame(String s): 创建标题为s的窗口。
setBounds(int a, int b, int width, int height): 设置窗口的初始位置和
    大小。
setSize(int width, int height): 设置窗口的大小。
setLocation(int x, int y): 设置窗口的位置。
setVisible(boolean b): 设置窗口是否可见。
setResizable(boolean b): 设置窗口是否可调整大小。
dispose(): 撤消当前窗口, 并释放当前窗口所使用的资源。
setExtendedState(int state): 设置窗口的扩展状态。
setDefaultCloseOperation(int operation): 设置窗口关闭操作。
```

11.3.3 菜单条、菜单、菜单项

- **菜单条**:

```
JMenuBar: 创建菜单条。
setJMenuBar(JMenuBar bar): 将菜单条添加到窗口中。
```

- **菜单**:

```
JMenu: 创建菜单。
add(JMenuItem item): 向菜单增加菜单项。
```

- **菜单项**:

```
JMenuItem: 创建菜单项。
setEnabled(boolean b): 设置菜单项是否可被选择。
setAccelerator(KeyStroke keyStroke): 为菜单项设置快捷键。
```

- **嵌入子菜单**:

```
JMenu是JMenuItem的子类, 可以嵌入子菜单。
```

- **菜单上的图标**:

```
使用Icon和ImageIcon类创建图标。
```

11.4 常用组件与布局

11.4.1 常用组件

- **常用组件**:

```
JTextField: 创建文本框。
JTextArea: 创建文本区。
JButton: 创建按钮。
JLabel: 创建标签。
JCheckBox: 创建选择框。
JRadioButton: 创建单选按钮。
JComboBox: 创建下拉列表。
JPasswordField: 创建密码框。
```

11.4.2 常用容器

- **常用容器**:

```
JPanel: 创建面板, 默认布局是FlowLayout。
JScrollPane: 创建滚动窗格。
JSplitPane: 创建拆分窗格。
JLayeredPane: 创建分层窗格。
```

11.4.3 常用布局

- **布局管理器**:

```
FlowLayout: 顺序布局。
BorderLayout: 边界布局。
CardLayout: 卡片布局。
GridLayout: 网格布局。
null布局: 空布局, 精确定位组件的位置和大小。
```

11.4.4 选项卡窗格

- **JTabbedPane**:

```
创建选项卡窗格, 默认布局是CardLayout。
```

11.5 处理事件

11.5.1 事件处理模式

- **事件源**:

```
能够产生事件的对象。
```

- **监视器**:

```
事件源通过方法将对象注册为监视器。
```

- **处理事件的接口**:

```
监视器对象自动调用方法处理事件。
```

11.5.2 ActionEvent 事件

- **事件源**:

```
文本框、按钮、菜单项、密码框和单选按钮。
```

- **注册监视器**:

```
addActionListener(ActionListener listener): 注册监视器。
```

- **ActionListener 接口**:

```
actionPerformed(ActionEvent e): 处理事件。
```

- **ActionEvent 类方法**:

```
getSource(): 获取事件源对象。
getActionCommand(): 获取命令字符串。
```

11.5.3 ItemEvent 事件

- **事件源**:

```
选择框、下拉列表。
```

- **注册监视器**:

```
addItemListener(ItemListener listener): 注册监视器。
```

- **ItemListener 接口**:

```
itemStateChanged(ItemEvent e): 处理事件。
```

- **ItemEvent 类方法**:

```
getItemSelectable(): 返回事件源。
```

11.5.4 DocumentEvent 事件

- **事件源**:

```
文本区维护的文档。
```

- **注册监视器**:

```
addDocumentListener(DocumentListener listener): 注册监视器。
```

- **DocumentListener 接口**:

```
changedUpdate(DocumentEvent e)
removeUpdate(DocumentEvent e)
insertUpdate(DocumentEvent e)
```

11.5.5 MouseEvent 事件

- **事件源**:

```
组件上的鼠标事件。
```

- **注册监视器**:

```
addMouseListener(MouseListener listener)
addMouseMotionListener(MouseMotionListener listener)
```

- **MouseListener 接口**:

```
mousePressed(MouseEvent)
mouseReleased(MouseEvent)
mouseEntered(MouseEvent)
mouseExited(MouseEvent)
mouseClicked(MouseEvent)
```

- **MouseMotionListener 接口**:

```
mouseDragged(MouseEvent)
mouseMoved(MouseEvent)
```

11.5.6 焦点事件

- **事件源**:

```
组件可以触发焦点事件。
```

- **注册监视器**:

```
addFocusListener(FocusListener listener)
```

- **FocusListener 接口**:

```
1 focusGained(FocusEvent e)
2 focusLost(FocusEvent e)
```

11.5.7 键盘事件

- **事件源**:

```
1 组件处于激活状态时触发键盘事件。
```

- **注册监视器**:

```
1 addKeyListener(KeyListener listener)
```

- **KeyListener 接口**:

```
1 keyPressed(KeyEvent e)
2 keyTyped(KeyEvent e)
3 keyReleased(KeyEvent e)
```

11.5.8 匿名类实例或窗口做监视器

- **匿名类**:

```
1 匿名类的外嵌类的成员变量在匿名类中有效。
```

- **窗口做监视器**:

```
1 事件源所在类的实例作为监视器。
```

11.5.9 事件总结

- **授权模式**:

```
1 基于授权模式的事件处理。
```

- **接口回调**:

```
事件源发生事件时，接口回调方法。
```

- **方法绑定**:

```
事件源触发事件后，调用相应的方法。
```

- **保持松耦合**:

```
事件处理保持系统的松耦合性。
```

11.6 使用 MVC 结构

- **MVC 结构**:

```
模型（Model）：存储数据的对象。
视图（View）：显示数据的对象。
控制器（Controller）：处理用户交互的对象。
```

11.7 对话框

11.7.1 对话框类型

- **对话框类型**:

```
无模式对话框：能再激活其它窗口，不堵塞线程执行。
有模式对话框：只响应对话框内部的事件，堵塞其它线程执行。
```

11.7.2 消息对话框

- **JOptionPane 类**:

```
showMessageDialog(Component parentComponent, String message, String
    title, int messageType): 创建消息对话框。
```

11.7.3 输入对话框

- **JOptionPane 类**:

```
showInputDialog(Component parentComponent, Object message, String
    title, int messageType): 创建输入对话框。
```

11.7.4 确认对话框

- **JOptionPane 类**:

```
showConfirmDialog(Component parentComponent, Object message, String
    title, int optionType): 创建确认对话框。
```

11.7.5 颜色对话框

- **JColorChooser 类**:

```
showDialog(Component component, String title, Color initialColor): 创
    建颜色对话框。
```

11.7.6 文件对话框

- **JFileChooser 类**:

```
showSaveDialog(Component a): 显示保存文件对话框。
showOpenDialog(Component a): 显示打开文件对话框。
```

11.7.7 自定义对话框

- **JDialog 类**:

```
创建对话框, 通过建立JDialog的子类来实现。
```

11.8 发布 GUI 程序

- **发布步骤**:

1. 编写清单文件。
2. 生成 JAR 文件。
3. 在安装了 Java 运行环境的计算机上运行 JAR 文件。

11.9 总结

- 本章详细介绍了 Java 中的组件及事件处理，包括 Java Swing 的基本概念、窗口、常用组件与布局、事件处理模式、使用 MVC 结构、对话框和发布 GUI 程序。这些内容是 Java GUI 编程的重要组成部分，掌握这些知识可以有效地开发和发布 Java 图形用户界面程序。

补充内容：泛型与集合框架

11.10 导读

- 主要内容：
  - 泛型
  - 集合框架

11.11 泛型

11.11.1 泛型的定义

- 泛型实现了参数化类型的能力，使代码可以应用于多种类型。
- 目的：使类、方法和接口具备广泛的表达能力。
- Java 允许定义泛型类、泛型接口和泛型方法。

11.11.2 泛型类

- 定义泛型类的格式：

```
public class 类名<类型参数表> {
    // 成员变量
    // 成员方法
}
```

- 例子：定义一个 Pair 类，其中有两个私有成员变量 `first` 和 `second`，类型可以为任意类型。

11.11.3 泛型方法

- 定义泛型方法的格式：

```
public <T> 返回类型 方法名(参数列表) {
    // 方法体
}
```

11.11.4 泛型接口

- 定义泛型接口的格式：

```
interface 接口名称<类型参数表> {
    // 方法原型
}
```

- 例子：定义 Info 接口，声明的 `print()` 方法可以输出任意类型对象的提示信息。

11.12 集合框架

11.12.1 集合框架概述

- Java 集合框架提供一系列能有效组织和操作数据的数据结构。
- 主要包含以下三部分：
  1. 对外的接口
  2. 接口的实现
  3. 对集合进行运算的算法

11.12.2 Iterator 接口

- **Iterator 接口**：用于遍历集合中的元素。
- 隐藏各种集合类的底层实现细节，提供统一的编程接口。

11.12.3 常用集合类

1. **ArrayList 类**：
   - 动态数组，实现了 List 接口。
   - 例子：创建 ArrayList 对象，添加元素并使用迭代器遍历集合。
2. **LinkedList 类**：
   - 双向链表，实现了 List 接口。
   - 例子：创建两个链表，合并链表并删除元素。
3. **Stack 类**：
   - 栈，后进先出（LIFO）容器。
   - 例子：把英文字符串按空格分隔后依次入栈，再依次出栈。
4. **HashMap 类**：
   - 映射表，存放键值对，通过键查找对应的值。
   - 例子：使用 HashMap 类存放和查找键值对。

第 12 章 Java 多线程机制

12.1 导读

- 主要内容:
  - 进程与线程
  - Java 中的线程
  - 线程的创建
  - 线程的生命周期
  - 线程同步
  - 线程联合

12.2 进程与线程

- **进程**: 程序的一次执行过程，可以产生多个线程。
- **线程**: 更小的执行单位，可以共享进程的内存。

12.3 Java 中的线程

- Java 支持多线程，JVM 管理线程调度。
- 线程有四种状态: 新建、运行、中断和死亡。

12.4 线程的创建

- 通过继承 `Thread` 类或实现 `Runnable` 接口创建线程。
- `Thread` 类的方法:
  - `start()`: 启动线程。
  - `run()`: 线程的执行代码。
  - `sleep(int millisecond)`: 使线程休眠。

12.5 线程的生命周期

- 线程在其生命周期中经历新建、就绪、运行、阻塞和死亡等状态。

12.6 线程同步

- 使用 `synchronized` 关键字进行方法同步。
- 使用 `wait()`, `notify()`, `notifyAll()` 进行线程间通信。

12.7 线程联合

- 使用 `join()` 方法使一个线程等待另一个线程执行完毕。

第 13 章 Java 网络编程

13.1 导读

- 主要内容:
  - URL 类
  - InetAddress 类
  - 套接字
  - UDP 数据报
  - 广播数据报
  - Java 远程调用 (RMI)

13.2 URL 类

- **URL 类**: 封装一个统一资源定位符 (Uniform Resource Locator)。
- 通过 `URL(String spec)` 和 `URL(String protocol, String host, String file)` 构造 URL 对象。
- 使用 `openStream()` 方法获取输入流读取资源。

13.3 InetAddress 类

- 表示 Internet 上的主机地址，包含域名和 IP 地址。
- 使用 `getByName(String host)` 和 `getLocalHost()` 获取 InetAddress 对象。

13.4 套接字

- 客户端使用 `Socket` 类与服务器建立连接。
- 服务器使用 `ServerSocket` 类监听客户端连接，使用 `accept()` 方法接收连接。

13.5 UDP 数据报

- 使用 `DatagramPacket` 和 `DatagramSocket` 类进行数据包的发送和接收。
- 通过 `send(DatagramPacket p)` 发送数据包，`receive(DatagramPacket p)` 接收数据包。

13.6 广播数据报

- 使用 D 类地址（224.0.0.0 224.255.255.255）进行数据报的广播。

13.7 Java 远程调用 (RMI)

- 允许一个 JVM 调用另一个 JVM 中的对象方法。
- 远程对象实现 `Remote` 接口，使用 `Naming` 类绑定远程对象。

第 14 章 JDBC 数据库操作

14.1 导读

- 主要内容：
  - Derby 数据库
  - 连接内置 Derby 数据库
  - 操作表
  - JDBC 操作
  - 使用预处理语句

14.2 Derby 数据库

- JDK 1.6 及以上版本提供的嵌入式数据库。
- 准备工作：将 `derby.jar`、`derbynet.jar` 和 `derbyclient.jar` 复制到 JDK 的 `lib` 目录中。

14.3 连接内置 Derby 数据库

- 使用 `connect 'jdbc:derby: 数据库;create=true|false';` 命令连接数据库。

14.4 操作表

- **创建表**：

```
create table 表名(字段1 属性, 字段2 属性, ...);
```

- **插入记录**：

```
insert into 表名 values (字段1值, 字段2值, ...);
```

- **查询记录**：

```
select * from 表名;
```

- **更新记录**：

```
update 表名 set 字段名=新值 where 条件;
```

- **删除记录**:

```
delete from 表名 where 条件;
```

14.5 JDBC 操作

- **加载驱动**:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

- **建立连接**:

```
Connection conn = DriverManager.getConnection("jdbc:derby:数据库名");
```

- **创建语句**:

```
Statement stmt = conn.createStatement();
```

- **执行查询**:

```
ResultSet rs = stmt.executeQuery("SQL语句");
```

- **更新操作**:

```
stmt.executeUpdate("SQL语句");
```

- **关闭连接**:

```
conn.close();
```

14.6 使用预处理语句

- **使用 `PreparedStatement` 提高执行效率和安全性。**
- **事务处理**:

- 使用 `conn.setAutoCommit(false)` 开启事务，`conn.commit()` 提交事务，`conn.rollback()` 回滚事务。