MIT 6.092

> Tip
>
> **Java基础：MIT 6.092: Introduction To Programming In Java**
>
> 学习时间：2025.01.24 - 2025.02.01

结合智谱清言、DeepSeek、通义千问、GPT-4o mini进行整体内容的总结复盘。

- [x] Types, variables, operators
- [x] More types, methods, conditionals
- [x] Loops and arrays
- [x] Objects and classes
- [x] Access control, class scope, packages, Java API
- [x] Design, debugging, interfaces
- [x] Inheritance, exceptions, file I/O

Lecture 1

Java

- “最受欢迎”的语言。
- 运行在“虚拟机” (JVM) 上。
- 比某些语言更复杂（例如 Python）。
- 比其他语言更简单（例如 C++）。

编译 Java

- 源代码 (.java) -> 字节码 (.class) -> Java

第一个程序

```
class Hello {
    public static void main(String[] arguments) {
    // 程序执行从这里开始
        System.out.println("Hello world.");
    }
}
```

程序结构

```
class CLASSNAME {
    public static void main(String[] arguments) {
        STATEMENTS
    }
}
```

输出

- `System.out.println(some String)` 将输出到控制台。

- 示例：`System.out.println(“output”);`

第二个程序

```
class Hello2 {
    public static void main(String[] arguments) {
        System.out.println("Hello world."); // 打印一次
        System.out.println("Line number 2"); // 再次!
    }
}
```

类型

- 可以存储和操作值的种类。
- `boolean`：布尔值 (true 或 false)。
- `int`：整数 (0, 1, -47)。
- `double`：实数 (3.14, 1.0, -2.1)。
- `String`：文本 (“hello”, “example”)。

变量

- 存储特定类型值的命名位置。
- 格式：`TYPE NAME;`
- 示例：`String foo;`

赋值

- 使用 `=` 给变量赋值。
- 示例：

```
String foo;
foo = “IAP 6.092”;
```

- **赋值**可以与变量声明结合使用。
- 示例：

```
double badPi = 3.14;
boolean isJanuary = true;
```

示例程序

```
class Hello3 {
    public static void main(String[] arguments) {
        String foo = "IAP 6.092";
        System.out.println(foo);
        foo = "Something else";
        System.out.println(foo);
    }
}
```

运算符

- 执行简单计算符号。
- 赋值：`=`
- 加法：`+`
- 减法：`-`
- 乘法：`*`
- 除法：`/`

运算顺序

- 遵循标准的数学规则：
    1. 括号
    2. 乘法和除法
    3. 加法和减法

示例程序

```
class DoMath {
    public static void main(String[] arguments) {
        double score = 1.0 + 2.0 * 3.0;
        System.out.println(score);
        score = score / 2.0;
        System.out.println(score);
    }
}
```

示例程序

```
class DoMath2 {
    public static void main(String[] arguments) {
        double score = 1.0 + 2.0 * 3.0;
        System.out.println(score);
        double copy = score;
        copy = copy / 2.0;
        System.out.println(copy);
        System.out.println(score);
    }
}
```

字符串连接 (+)

- 示例：

```
String text = "hello" + " world";
text = text + " number " + 5;
// text = "hello world number 5"
```

作业：重力计算器

- 计算下落物体的位置：
    - $$x(t) = 0.5 \times at^2 + v_i t + x_i$$

```java
class GravityCalculator {
    public static void main(String[] arguments) {
        double gravity = -9.81;  // Earth's gravity in m/s^2
        double initialVelocity = 0.0;
        double fallingTime = 10.0;
        double initialPosition = 0.0;
        // double finalPosition = 0.0;
        // The object's position after 10.0 seconds is 0.0 m.
        double finalPosition = initialVelocity * fallingTime + 0.5 * gravity *
Math.pow(fallingTime, 2) + initialPosition;
        System.out.println("The object's position after " + fallingTime +
                " seconds is " + finalPosition + " m.");
        // The object's position after 10.0 seconds is -490.5 m.
    }
}
```

Lecture 2

More Types

- **布尔类型 (`boolean`)**: 存储真值（`true`）或假值（`false`）。
- **整型 (`int`)**: 存储整数，例如 0, 1, -47 等。
- **浮点型 (`double`)**: 存储带小数的数字，例如 3.14, 1.0, -2.1 等。
- **字符串类型 (`String`)**: 存储文本，例如 "hello", "example" 等。
- **变量**: 存储值的命名位置，例如 `int x = 5;`。
- **运算符**: 执行简单计算的符号，例如：
  - **赋值运算符 (`=`)**: 将值赋给变量。
  - **算术运算符**: 进行数学运算，例如 `+`（加法）、`-`（减法）、`*`（乘法）、`/`（除法）。
- **运算符优先级**: 解释运算符的优先级规则，例如先进行乘除运算，再进行加减运算。
- **类型不匹配**: 不能将不同类型的值赋值给变量，例如将整型值赋值给字符串变量。
- **类型转换**:
  - **隐式转换**: 自动将一种类型的值转换为另一种类型，例如将整型值转换为浮点型值。
  - **显式转换 (强制转换)**: 使用 `(类型) 值` 的语法将一种类型的值转换为另一种类型，例如将浮点型值转换为整型值。`int a = (int)18.7;`

Methods

- **方法**: 一组语句的集合，用于执行特定的任务。
- **定义方法**: 使用 `public static void 方法名()` 的语法定义方法。
- **方法参数**: 方法接受的输入值，例如 `public static void printSquare(int x) {...}`。
- **方法调用**: 使用 `方法名(参数)` 的语法调用方法，例如 `printSquare(5)`。
- **方法返回值**: 方法执行完成后返回的值，例如 `public static double square(double x) {... return x*x; ...}`。
- **void 方法**: 不返回任何值的方法。
- **变量作用域**: 变量定义的位置决定了它的作用域，例如在方法中定义的变量只能在方法内部使用。
- **参数作用域**: 方法参数在方法内部有效，在方法外部无效。

- **方法抽象性**: 使用方法可以将复杂的任务分解为更小的、更易于管理的部分。
- **Java 库中的数学函数**: 例如 `Math.sin()`（计算正弦值）、`Math.cos()`（计算余弦值）、`Math.pow()`（计算幂）、`Math.log()`（计算对数）。

Conditionals

- **条件语句**: 根据条件的真假执行不同的代码。
- `if` **语句**: 如果条件为真，则执行代码块。
- **比较运算符**: 用于比较值，例如 `==`（等于）、`>`（大于）、`<`（小于）等。
- **布尔运算符**: 用于组合多个条件，例如 `&&`（逻辑与）和 `||`（逻辑或）。
- `else` **语句**: 如果 `if` 语句的条件为假，则执行 `else` 代码块。
- `else if` **语句**: 用于处理多个条件分支，例如 `if (条件1) {...} else if (条件2) {...} else {...}`。

Assignment: FooCorporation

- **作业任务**: 编写代码计算员工的工资，考虑加班情况。
- **作业要求**:
  - 使用方法计算基本工资和加班工资。
  - 使用条件语句判断是否需要支付加班工资。
  - 输出员工的最终工资。

```
public class FooCorporation {
    /**
     * desc: prints the total pay or an error message
     * args[0]: base pay, type: double
     * args[1]: hours worked, type: int
     * return: void
     */
    public static void calculate(double basePay, int hoursWorked) {
        double totalPay = 0;
        if (basePay < 8.0) {
            System.out.println("Error: Base pay cannot be less than $8 per
hour.");
            return;
        }
        if (hoursWorked > 60) {
            System.out.println("Error: Hours worked cannot exceed 60 hours per
week.");
            return;
        }
        if (hoursWorked > 0 && hoursWorked <= 40) {
            totalPay = basePay * hoursWorked;
        } else if (hoursWorked > 40 && hoursWorked <= 60) {
            totalPay = basePay * 40 + (hoursWorked - 40) * basePay * 1.5;
        }
        System.out.println("Total pay: $" + totalPay);
    }

    public static void main(String[] args) {
        // Test cases
```

```
        calculate(7.50, 35);
        calculate(8.20, 47);
        calculate(10.00, 73);
        // Error: Base pay cannot be less than $8 per hour.
        // Total pay: $414.1
        // Error: Hours worked cannot exceed 60 hours per week.
    }

}
```

Lecture 3

1. 良好的编程风格

- **使用有意义的变量和方法名**：
  - 变量名应清晰地描述其存储的数据类型和用途。
  - 方法名应清晰地描述其功能。
  - 例如，使用 `firstName` 和 `lastName` 而不是 `a1` 和 `a2`。
- **使用缩进**：
  - 使用缩进来清晰地表示代码块的层次结构。
  - 例如，使用缩进来表示 `if` 语句的条件块和循环语句的循环体。
- **使用空格**：
  - 在复杂的表达式和语句中使用空格，以提高可读性。
  - 在代码块之间添加空行，以提高可读性。
- **不要重复测试**：
  - 在条件语句中，避免对同一条件进行多次测试。
  - 例如，在 `if-else if` 语句中，确保每个 `else if` 语句的条件与前面的条件不同。

2. 循环

- `while` **循环**：
  - 语法：`while (condition) { statements }`
  - 功能：当条件为 `true` 时，重复执行代码块。
  - 注意：确保循环条件最终会变为 `false`，否则会导致无限循环。
- `for` **循环**：
  - 语法：`for (initialization; condition; update) { statements }`
  - 功能：重复执行代码块，直到条件为 `false`。
  - 初始化：在循环开始前执行一次。
  - 条件：在每次循环开始前进行测试。
  - 更新：在每次循环结束后执行。
- `break` **语句**：
  - 功能：立即**终止当前循环**，并跳转到**循环后面的代码**。
- `continue` **语句**：
  - 功能：跳过**当前循环的剩余部分**，并继续执行**下一次循环**。

- **嵌套循环**：
  - 在一个循环内部使用另一个循环。
  - 例如，使用嵌套循环遍历二维数组。

3. 数组

- **数组的定义和使用**：
  - 语法：`TYPE[] arrayName = new TYPE[size];`
  - 功能：创建一个指定类型和大小的新数组。
  - 例如，`int[] numbers = new int[10];` 创建一个包含 10 个整数的数组。
- **数组元素的访问**：
  - 语法：`arrayName[index]`
  - 功能：访问数组中指定索引处的元素。
  - 索引从 0 开始，到 `length - 1` 结束。
- **数组的长度**：
  - 语法：`arrayName.length`
  - 功能：获取数组的长度。
- **数组的初始化**：
  - 语法：`TYPE[] arrayName = { value1, value2, ... };`
  - 功能：在声明数组时为其元素赋值。
- **字符串数组**：
  - 字符串数组是存储字符串的数组。
  - 例如，`String[] names = { "Alice", "Bob", "Charlie" };` 创建一个包含三个字符串的数组。
- **循环遍历数组**：
  - 使用 `for` 循环或 `while` 循环遍历数组中的每个元素。
  - 例如，使用 `for` 循环遍历数组并打印每个元素：

```
for (int i = 0; i < arrayName.length; i++) {
    System.out.println(arrayName[i]);
}
```

作业：

- 编写程序找出波士顿马拉松比赛中表现最好和第二好的选手。

```
class Marathon {

    public static int findFastest(int[] times) {
        int index = 0;
        for (int i = 1; i < times.length; i++) {
            if (times[i] < times[index]) {
                index = i;
            }
        }
```

```
        return index;
    }

    public static int findSecondFastest(int[] times) {
        // find the fastest runner
        int fastest = findFastest(times);
        // find the second fastest runner
        // make sure we don't pick the fastest runner again
        int secondFastest = (fastest == 0) ? 1 : 0;
        for (int i = 0; i < times.length; i++) {
            if (i != fastest && times[i] < times[secondFastest]) {
                secondFastest = i;
            }
        }
        return secondFastest;
    }

    public static void main (String[] arguments) {
        String[] names = {
            "Elena", "Thomas", "Hamilton", "Suzie", "Phil", "Matt", "Alex",
            "Emma", "John", "James", "Jane", "Emily", "Daniel", "Neda",
            "Aaron", "Kate"
        };

        int[] times = {
            341, 273, 278, 329, 445, 402, 388, 275, 243, 334, 412, 393, 299,
            343, 317, 265
        };

        for (int i = 0; i < names.length; i++) {
            System.out.println(names[i] + ": " + times[i]);
        }
        int fastest = findFastest(times);
        int secondFastest = findSecondFastest(times);
        System.out.println("Fastest: " + names[fastest] + " and cost: " +
times[fastest]);
        System.out.println("Second fastest: " + names[secondFastest] + " and
cost: " + times[secondFastest]);
    }
}
```

Lecture 4

- **常见问题解析**:
  - **数组索引 vs 数组值**： 区分数组索引和数组中存储的值。
  - **条件语句和循环语句的花括号**： 花括号在条件语句和循环语句中的作用，以及缺少花括号可能导致的错误。

```java
public class HelloWorld {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++)
            System.out.println("Hi");
            System.out.println("Bye");
    }
}
```

```
Hi
Hi
Hi
Hi
Hi
Bye
```

- **变量初始化**：确保变量在使用前进行初始化（注意选择**合适的值**进行初始化），避免出现意外结果。
- **方法内部定义方法**：减少在方法内部定义方法。
- **调试技巧**：使用 `System.out.println` 打印语句来跟踪代码的执行过程，以及使用格式化工具提高代码可读性。

- **面向对象编程**：
  - **介绍**：理解面向对象编程的概念，以及它如何帮助我们模拟现实世界。
  - **使用类**：
    - **定义类**：使用 `public class` 语句定义类，每个类对应一个文件。
      - 类名大写开头
      - 有 `main` 方法意味着该类可以 `run`
    - **字段**：定义类的属性，例如 `String name`、`int numPoops` 等。
    - **方法**：定义类的行为，例如 `void sayHi()`、`void eat(double foodWeight)` 等。
    - **构造函数**：用于初始化对象的特殊方法，例如 `Baby(String myname, boolean maleBaby)`。
      - 注意构造函数名就等于类名。
      - 注意构造函数永远不要返回任何东西。
      - 至少需要一个构造函数，没有自定义构造函数会默认有一个。
    - **实例化对象**：使用 `new` 关键字创建类的实例，例如 `Baby shiloh = new Baby("Shiloh Jolie-Pitt", true)`。
    - **访问字段**：使用 `对象.字段名` 的方式访问对象的字段，例如 `shiloh.name`。
    - **调用方法**：使用 `对象.方法名(参数们)` 的方式调用对象的方法，例如 `shiloh.sayHi()`。
  - **引用与值**：
    - **基本类型 vs 引用类型**：区分基本类型（如 `int`、`double`）和引用类型（数组和对象，如 `String`、`int[]`、`Baby`）。
    - **对象的存储方式**：
      - 对象过于庞大以至于不能放在变量之中。
      - 对象存储在堆中，变量存储的是对象的引用（内存地址）。

        - **引用的赋值和比较**： 使用 `=` 赋值时，实际上是在**改变引用的指向**；使用 `==` 比较时，是在比较引用是否指向同一个对象。

          ```
          Baby shiloh1 = new Baby("shiloh");
          Baby shiloh2 = new Baby("shiloh");
          // but shiloh1 != shiloh2
          ```

        - **方法和引用**： 方法可以接收引用类型的参数，并修改对象的状态。
    - **静态类型和方法**:
        - **静态字段**： 属于类本身，而不是属于某个特定的实例。
        - **静态方法**： 属于类本身，而不是属于某个特定的实例，可以使用 `类名.方法名()` 的方式调用。
        - **静态方法的调用**： 静态方法不能直接访问非静态字段，但可以访问静态字段。
        - `main`方法被声明为`static`
- **作业**：建模书籍和图书馆
    - 实现`Book`类。

```
public class Book {

    String title;
    boolean borrowed;

    // Creates a new Book
    public Book(String bookTitle) {
        // Implement this method
        title = bookTitle;
        borrowed = false;
    }

    // Marks the book as rented
    public void borrowed() {
        // Implement this method
        borrowed = true;
    }

    // Marks the book as not rented
    public void returned() {
        // Implement this method
        borrowed = false;
    }

    // Returns true if the book is rented, false otherwise
    public boolean isBorrowed() {
        // Implement this method
        return borrowed;
    }

    // Returns the title of the book
    public String getTitle() {
        // Implement this method
        return title;
    }
```

```
    public static void main(String[] arguments) {
        // Small test of the Book class
        Book example = new Book("The Da Vinci Code");
        System.out.println("Title (should be The Da Vinci Code): " +
example.getTitle());
        System.out.println("Borrowed? (should be false): " +
example.isBorrowed());
        example.borrowed();
        System.out.println("Borrowed? (should be true): " +
example.isBorrowed());
        example.returned();
        System.out.println("Borrowed? (should be false): " +
example.isBorrowed());
    }
}
```

- 实现 `Library` 类。

```
import java.util.ArrayList;

public class Library {
    // Add the missing implementation to this class

    // address of the library

    String address;
    ArrayList<Book> books;

    // constructor
    public Library(String address) {
        this.address = address;
        this.books = new ArrayList<Book>();
    }

    public static void printOpeningHours() {
        System.out.println("Libraries are open daily from 9am to 5pm.");
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public void printAddress() {
        System.out.println(address);
    }

    public void borrowBook(String title) {
        for (Book book : books) {
            if (book.getTitle().equals(title)) {
                if (!book.isBorrowed()) {
                    book.borrowed();
                    System.out.println("You successfully borrowed " +
title);
                } else {
```

```
                    System.out.println("Sorry, this book is already
borrowed.");
                }
                return;
            }
        }
        System.out.println("Sorry, this book is not in our catalog.");
    }

    public void printAvailableBooks() {
        boolean flag = false;
        for (Book book : books) {
            if (!book.isBorrowed()) {
                flag = true;
                System.out.println(book.getTitle());
            }
        }
        if (flag == false) {
            System.out.println("No book in catalog.");
        }
    }

    public void returnBook(String title) {
        for (Book book : books) {
            if (book.getTitle().equals(title)) {
                if (book.isBorrowed()) {
                    book.returned();
                    System.out.println("You successfully returned " +
title);
                } else {
                    System.out.println("Sorry, this book is not
borrowed.");
                }
                return;
            }
        }
        System.out.println("Sorry, this book is not in our catalog.");
    }

    public static void main(String[] args) {
        // Create two libraries
        Library firstLibrary = new Library("10 Main St.");
        Library secondLibrary = new Library("228 Liberty St.");

        // Add four books to the first library
        firstLibrary.addBook(new Book("The Da Vinci Code"));
        firstLibrary.addBook(new Book("Le Petit Prince"));
        firstLibrary.addBook(new Book("A Tale of Two Cities"));
        firstLibrary.addBook(new Book("The Lord of the Rings"));

        // Print opening hours and the addresses
        System.out.println("Library hours:");
        printOpeningHours();
        System.out.println();

        System.out.println("Library addresses:");
```

```
        firstLibrary.printAddress();
        secondLibrary.printAddress();
        System.out.println();

        // Try to borrow The Lords of the Rings from both libraries
        System.out.println("Borrowing The Lord of the Rings:");
        firstLibrary.borrowBook("The Lord of the Rings");
        firstLibrary.borrowBook("The Lord of the Rings");
        secondLibrary.borrowBook("The Lord of the Rings");
        System.out.println();

        // Print the titles of all available books from both libraries
        System.out.println("Books available in the first library:");
        firstLibrary.printAvailableBooks();
        System.out.println();
        System.out.println("Books available in the second library:");
        secondLibrary.printAvailableBooks();
        System.out.println();

        // Return The Lords of the Rings to the first library
        System.out.println("Returning The Lord of the Rings:");
        firstLibrary.returnBook("The Lord of the Rings");
        System.out.println();

        // Print the titles of available from the first library
        System.out.println("Books available in the first library:");
        firstLibrary.printAvailableBooks();
    }
}
```

Lecture 5

概述

- **复习**：回顾之前的内容。
- **访问控制**：管理类成员的可见性。
- **类作用域**：理解类中变量的作用域。
- **包**：将类组织到包中。
- **Java API**：使用Java内置库。

复习

- **Counter类示例**：演示实例变量和静态变量。

```
public class Counter {
    int myCount = 0;
    static int ourCount = 0;
    void increment() {
        myCount++;
        ourCount++;
    }
}
```

访问控制

- **Public vs. Private**：
  - **Public**：任何类都可以访问。
  - **Private**：只能在类内部访问。
- **示例**：

```
public class CreditCard {
    private String cardNumber;
    private double expenses;
    public void charge(double amount) {
        expenses += amount;
    }
    public String getCardNumber(String password) {
        if (password.equals("SECRET!3*!")) {
            return cardNumber;
        }
        return "jerkface";
    }
}
```

- **为什么需要访问控制**：
  - 保护私有信息。
  - 明确类的使用方式。
  - 将实现与接口分离。

类作用域

- **作用域回顾**：变量在`{}`内可访问。
  - **方法级作用域**：在方法内部声明的变量。
  - **类级作用域**：作为类成员声明的变量。
- **示例**：

```
public class ScopeReview {
    private int var3;
    void scopeMethod(int var1) {
        var3 = var1;
        String var2;
        if (var1 > 0) {
            var2 = "大于0";
        } else {
            var2 = "小于或等于0";
        }
        System.out.println(var2);
    }
}
```

- `this`**关键字**：引用当前对象。

```
public class Baby {
    int servings;
    void feed(int servings) {
        this.servings += servings;
    }
}
```

包

- **定义**：将类组织到**目录**中，可将包理解为一种目录。
- **使用**：
  - **定义包**：

    ```
    package path.to.package.foo;
    class Foo { }
    ```

  - **使用包**：

    ```
    import path.to.package.foo.Foo;
    import path.to.package.foo.*;
    ```

- **示例**：

```
package parenttools;
public class Baby { }
package adult;
import parenttools.Baby;
public class Parent {
    public static void main(String[] args) {
        Baby baby = new Baby();
    }
}
```

- **为什么使用包**：
  - 组合相似的功能。
  - 区分相似名称的类。

Java API

- **概述**：Java包含许多内置的包和类。
- **重用**：利用现有类避免重复工作。
- **示例**：`ArrayList`、`TreeSet`、`HashMap`。

```
import java.util.ArrayList;
public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> strings = new ArrayList<>();
        strings.add("Evan");
        strings.add("Eugene");
        strings.add("Adam");
        System.out.println(strings.size());
```

```
        System.out.println(strings.get(0));
        strings.set(0, "Goodbye");
        strings.remove(1);
        for (String s : strings) {
            System.out.println(s);
        }
    }
}
```

数组

- **数组的创建和使用**:
  - 使用方括号声明数组，例如 `int[] numbers = new int[10];`
  - 使用索引访问数组元素，例如 `numbers[0] = 5;`
  - 数组大小固定，无法动态扩展。

集合框架

- **ArrayList**:
  - 动态数组，可以动态添加和删除元素。
  - 使用 add() 方法添加元素，使用 remove() 方法删除元素。
  - 示例:

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("Hello");
strings.remove(0);
```

- **Set**:
  - 存储不重复的元素。
  - TreeSet: 排序集合，按照元素的自然顺序或比较器排序。
  - HashSet: 无序集合，使用哈希表实现。
  - 示例:

```
TreeSet<String> strings = new TreeSet<String>();
strings.add("Hello");
strings.remove("World");
```

- **Map**:
  - 存储键值对。
  - TreeMap: 排序映射，按照键的自然顺序或比较器排序。
  - HashMap: 无序映射，使用哈希表实现。
  - 示例:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("Hello", 5);
map.get("Hello");
```

其他知识点

- **可比较接口 (Comparable)**:
  - 实现 Comparable 接口的类可以进行比较排序。
  - 需要实现 compareTo() 方法。
- **equals 和 hashCode 方法**:
  - 用于 HashSet 和 HashMap 的元素比较和哈希码生成。
  - 需要保证 equals() 和 hashCode() 方法的一致性。

总结

- **复习**：关键概念的回顾。
- **访问控制**：Public与Private的区别。
- **类作用域**：方法级和类级作用域。
- **包**：组织和使用包。
- **Java API**：使用内置库。

作业：图形

- **资源**:
  - Graphics API
  - ArrayList API
- **修改部分:**

```
import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class DrawGraphics {
    ArrayList<BouncingBox> boxes;

    /** Initializes this class for drawing. */
    public DrawGraphics() {
        boxes = new ArrayList<BouncingBox>();
        BouncingBox box = new BouncingBox(200, 50, Color.RED);
        box.setMovementVector(1, 0);
        boxes.add(box);
        box = new BouncingBox(200, 50, Color.BLUE);
        box.setMovementVector(0, 1);
        boxes.add(box);
        box = new BouncingBox(200, 50, Color.GREEN);
        box.setMovementVector(1, 1);
        boxes.add(box);
    }

    /** Draw the contents of the window on surface. Called 20 times per second.
*/

    public void draw(Graphics surface) {
        surface.drawLine(50, 50, 250, 250);
        surface.drawRect(20, 20, 10, 20);
        surface.drawOval(90, 60, 20, 20);
        surface.drawString("hello, world!", 90, 50);
```

```
        for (BouncingBox bouncingBox : boxes) {
            bouncingBox.draw(surface);
        }
    }
}
```

- 由于此前章节中我们知道可以使用等号更新引用，故而可以多次使用`box`，每次为其指向不同实例即可。

Lecture 6

作业5：

- **main**() 方法
  - 程序从 `main()` 方法开始，但多个类可以有 `main()` 方法。
  - 示例代码：

```
public class SimpleDraw {
    public static void main(String args[]) {
        SimpleDraw content = new SimpleDraw(new DrawGraphics());
    }
}
```

- **DrawGraphics** 类
  - `DrawGraphics` 类包含一个 `BouncingBox` 对象，并在构造函数中初始化。
  - `draw` 方法用于绘制图形。
  - 示例代码：

```
public class DrawGraphics {
    BouncingBox box;
    public DrawGraphics() {
        box = new BouncingBox(200, 50, Color.RED);
    }
    public void draw(Graphics surface) {
        surface.drawLine(50, 50, 250, 250);
        box.draw(surface);
    }
}
```

- **扩展 DrawGraphics 类**
  - 使用 `ArrayList` 存储多个 `BouncingBox` 对象，并为每个对象设置移动向量。
  - 示例代码：

```
public class DrawGraphics {
    ArrayList<BouncingBox> boxes = new ArrayList<BouncingBox>();
    public DrawGraphics() {
        boxes.add(new BouncingBox(200, 50, Color.RED));
        boxes.add(new BouncingBox(10, 10, Color.BLUE));
        boxes.add(new BouncingBox(100, 100, Color.GREEN));
        // 此处采用get区别于我的实现。
        boxes.get(0).setMovementVector(1, 0);
```

```
            boxes.get(1).setMovementVector(-3, -2);
            boxes.get(2).setMovementVector(1, 1);
        }
        public void draw(Graphics surface) {
            for (BouncingBox box : boxes) {
                box.draw(surface);
            }
        }
    }
```

好的程序设计

- **正确性**：程序应无错误。
- **易理解性**：代码应易于理解。
- **易修改性**：代码应易于修改和扩展。
- **性能**：程序应有良好的性能。

一致性

- 代码风格一致性有助于编写和理解代码。
- Java 有广泛接受的代码风格指南。

命名规范

- **变量**：名词，首字母小写，单词间用大写字母分隔，如 `x`，`shape`，`highScore`。
- **方法**：动词，首字母小写，如 `getSize()`，`draw()`。
- **类**：名词，首字母大写，如 `Shape`，`WebPage`。

好的类设计

- 好的类应易于理解和使用。
- 默认将字段和方法设为私有，**仅在需要时设为公有。**
- 如果**需要访问字段，应创建方法**，如 `public int getBar() { return bar; }`。

调试

- 调试是查找和纠正程序错误的过程，是编程中的基本技能。

调试步骤

1. **避免错误**：尽量不引入错误。
2. **尽早发现错误**：越早发现错误，越容易修复。
3. **重现错误**：找出如何重复错误，创建**最小**测试用例。
4. **生成假设**：假设错误的原因。
5. **收集信息**：验证假设，使用 `System.out.println()` 或调试器。
6. **检查数据**：检查数据，确认假设，修复错误或生成新假设。

伪代码设计

- 伪代码是程序的高级描述，不关注细节，关注结构。
- 示例：判断数字是否在区间 `[x, y)` 内：

```
if number < x return false
if number >= y return false
return true
```

工具与测试

- **警告**：警告可能意味着错误，建议修复所有警告。
- **断言**：验证代码行为，如果断言为假，程序崩溃。
- **单元测试**：使用 JUnit 进行测试。

方法与对象

- **方法**：避免代码重复，**封装**功能。使用方法而不关心实现。
- **对象**：将相关变量和方法组合在一起，提供简单接口。

接口

- 接口是一组**共享方法**的类集合。绝大多数情况下接口中**只有方法。**
- 示例：`Drawable` 接口：

```
interface Drawable {
    void draw(Graphics surface);
    void setColor(Color color);
}
```

- 类可以实现多个接口，接口**只提供方法定义**，不提供代码。

使用接口

- 只能访问接口中的方法。
- 如果需要访问**特定类型的方法**，可以使用**类型转换**：

```
Drawable d = new BouncingBox(...);
BouncingBox box = (BouncingBox) d;
box.setMovementVector(1, 1);
```

作业：更多图形

- 实现 `Sprite` 接口的新类：这里实现一个圆

```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class Oval implements Sprite {

    private int width;
    private int height;
    private Color color;

    public Oval(int width, int height, Color color) {
        this.width = width;
        this.height = height;
```

```
        this.color = color;
    }

    public void draw(Graphics surface, int x, int y) {
        // Draw the object
        surface.setColor(color);
        surface.fillOval(x, y, width, height);
        surface.setColor(Color.BLACK);
        ((Graphics2D) surface).setStroke(new BasicStroke(3.0f));
        surface.drawOval(x, y, width, height);
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

}
```

- 添加`StraightMovers`列表：略
- 创建`Mover`接口减少重复

```
import java.awt.Graphics;

public interface Mover {
    void setMovementVector(int xIncrement, int yIncrement);

    void draw(Graphics graphics);
}
```

- 更改后的`DrawGraphics`代码：

```
import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class DrawGraphics {
    // ArrayList<Bouncer> movingSprites = new ArrayList<>();
    // ArrayList<StraightMover> straightMovers = new ArrayList<>();
    ArrayList<Mover> movers = new ArrayList<>();

    /** Initializes this class for drawing. */
    public DrawGraphics() {
        // the first sprite
        movers.add(new Bouncer(100, 170, new Rectangle(15, 20, Color.RED)));
        movers.get(0).setMovementVector(3, 1);
        // the second sprite
        movers.add(new Bouncer(50, 170, new Oval(15, 20, Color.BLUE)));
        movers.get(1).setMovementVector(-3, 1);
        // the third sprite
        movers.add(new StraightMover(100, 170, new Rectangle(15, 20,
Color.GREEN)));
```

```
        movers.get(2).setMovementVector(2, 0);
        // the fourth sprite
        movers.add(new StraightMover(50, 170, new Oval(15, 20,
Color.YELLOW)));
        movers.get(3).setMovementVector(-1, 0);
    }

    /** Draw the contents of the window on surface. */
    public void draw(Graphics surface) {
        for (Mover mover : movers) {
            mover.draw(surface);
        }
    }
}
```

Lecture 7

接口（Interfaces）

- **接口是契约**：实现接口的类必须实现接口中定义的所有方法。
- 接口中的字段是 `final` 的，不能被修改。
- 示例：

```
public interface ICar {
    boolean isCar = true;
    int getNumWheels();
}
```

接口实现

- 示例：`BigRig` 类实现 `ICar` 接口：

```
class BigRig implements ICar {
    int getNumWheels() {
        return 18;
    }
}
```

作业：Bouncer 与 Sprite

- `Bouncer` 绘制一个 `Sprite`，`Sprite` 是一个接口，可以绘制任何图形。
- `Mover` 负责更新 `Sprite` 的坐标。

继承（Inheritance）

- **继承的基本概念**：子类继承父类的字段和方法。**You don't have to Copy & Paste!**
- 示例：`Wizard` 继承 `Dude`：
  - Wizard can use everything the Dude has!
  - Wizard can do everything Dude can do!
  - **You can use a Wizard like a Dude too!**
  - 注意私有字段和方法除外。

```
public class Wizard extends Dude {
    ArrayList<Spell> spells;
    public void cast(String spell) {
        mp -= 10;
    }
}
```

继承的机制

- Java 在调用方法时，会**从子类开始查找**，如果找不到则**向上查找父类。**
- 示例：`grandWizard1.punchFace(dude1)` 的调用过程：
    1. 查找 `GrandWizard` 类中的 `punchFace` 方法。
    2. 如果找不到，查找父类 `Wizard`。
    3. 如果还找不到，继续查找父类 `Dude`。
    4. 找到后调用该方法。
- `((Dude)grandWizard1).sayName()` 方法如何执行?
    - 转换到 `Dude` 类型
    - 在 `Dude` 类中查找 `sayName()` 方法

继承的限制

- Java 不支持多重继承，一个类**只能继承一个父类。**
- 如果多个父类有相同的方法，Java 无法确定调用哪个方法。

异常（Exceptions）

- **异常**是程序运行时发生的意外事件，如 `NullPointerException`、`ArrayIndexOutOfBoundsException` 等。
- 异常用于通知调用者发生了错误。
- 可通过继承自 `Exception` 类来定义自己的异常类型。
- 示例：抛出 `ArrayIndexOutOfBoundsException`：

```
public Object get(int index) throws ArrayIndexOutOfBoundsException {
    if (index < 0 || index >= size())
        throw new ArrayIndexOutOfBoundsException("" + index);
}
```

捕获异常

- 使用 `try-catch` 块捕获异常：

```
try {
    get(-1);
} catch (ArrayIndexOutOfBoundsException err) {
    System.out.println("oh dear!");
}
```

重新抛出异常

- 如果不想处理异常，可以重新抛出：

```
void doBad() throws ArrayIndexOutOfBoundsException {
    get(-1);
}
```

- 注意重新抛出异常会不断向上，直到异常被处理，最后如果传到`main`，异常不可被再抛出。

输入输出（I/O）

- **InputStream**：字节流，逐个读取**字节**。
- **InputStreamReader**：将字节流转换为**字符**流。
- **BufferedReader**：缓冲字符流，支持**逐行读取**。
- 示例：从控制台读取输入：

```
InputStreamReader ir = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(ir);
br.readLine();
```

文件读取

- 使用`FileReader`和`BufferedReader`读取文件：

```
FileReader fr = new FileReader("readme.txt");
BufferedReader br = new BufferedReader(fr);
String line = null;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
br.close();
```

作业：魔方阵

- 读取两个文件，检查所有行和列以及对角元素的和是否为定值。

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;

public class MagicSquares {
    public static boolean testMagic(String pathName) throws IOException {
        // Open the file
        BufferedReader reader = new BufferedReader(new FileReader(pathName));

        boolean isMagic = true;
        int lastSum = -1;

        // For each line in the file ...
        String line;
        String[][] nums = null;
        int row = 0;
        while ((line = reader.readLine()) != null) {
            // ... sum each row of numbers
            String[] parts = line.split("\t");
            int sum = 0;
```

```
            for (String part : parts) {
                sum += Integer.valueOf(part);
            }

            if (lastSum == -1) {
                // If this is the first row, remember the sum and the length of
the row
                nums = new String[parts.length][parts.length];
                lastSum = sum;
            } else if (lastSum != sum) {
                // if the sums don't match, it isn't magic, so stop reading
                isMagic = false;
                reader.close();
                return isMagic;
            }
            assert (nums != null);
            for (int i = 0; i < parts.length; i++) {
                nums[row][i] = parts[i];
            }
            // Read the empty line between the rows
            line = reader.readLine();
            // the number of rows inc
            row++;
        }
        reader.close();

        // Check the columns
        for (int i = 0; i < nums.length; i++) {
            int sum = 0;
            for (String[] strings : nums) {
                sum += Integer.parseInt(strings[i]);
            }
            if (lastSum != sum) {
                isMagic = false;
                return isMagic;
            }
        }
        // Check the diagonals
        int sum = 0;
        for (int i = 0; i < nums.length; i++) {
            sum += Integer.parseInt(nums[i][i]);
        }
        if (lastSum != sum) {
            isMagic = false;
            return isMagic;
        }
        sum = 0;
        for (int i = 0; i < nums.length; i++) {
            sum += Integer.parseInt(nums[i][nums.length - 1 - i]);
        }
        if (lastSum != sum) {
            isMagic = false;
        }
        return isMagic;
    }
```

```
    public static void main(String[] args) {
        String[] fileNames = { "Mercury.txt", "Luna.txt", "Test.txt" };
        for (String fileName : fileNames) {
            try {
                System.out.println(fileName + " is magic? " +
testMagic(fileName));
            } catch (IOException e) {
                System.out.println("Error reading " + fileName);
            }
        }
    }
}
```