

# 1.1 Strategy Design

策略设计模式

# 面向对象编程

## Objective Oriented Programming

- 对象 (Object)
- 封装 (Encapsulation)
- 继承 (Inheritance)
- 抽象 (Abstraction)
- 多态 (Polymorphism)

# 封装 Encapsulation

# 封装

## Encapsulation

- 利用class, private, public, static, protect等访问控制来对数据和代码进行管控

- ```
public class Person {  
    private String name; // 私有变量封装  
    private int age;  
  
    public Person(String name, int age) { // 结构  
        this.name = name;  
        this.age = age;  
    }  
  
    // 提供对外访问方式 (getter和setter)  
    public String getName() {  
        return name;  
    }  
  
    public void setAge(int age) {  
        if (age >= 0) this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

# 封装 - 访问修饰

## Encapsulation - Access Modifiers

- private - 当前类
- {empty} - 当前类、同包类
- protected - 当前类、同包类、子类（不同包）
- public - 当前类、同包类、子类（不同包）、外部类

# 封装 - 访问修饰

## Encapsulation - Access Modifiers

- private - 当前类
- {empty} - 当前类、同包类
- protected - 当前类、同包类、子类（不同包）
- public - 当前类、同包类、子类（不同包）、外部类
- final - “不可修改”，常用于常量，类似c++/c 里的const
- static - “静态”，不因创建多个对象/实例方法而独立

# 静态方法

## Static Method

静态方法在类加载时就被创建，因此不依赖对象实例化

```
public class MyClass {  
    public static void myStaticMethod() {  
        System.out.println("静态方法");  
    }  
  
    public void myInstanceMethod() {  
        System.out.println("实例方法");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        // 直接通过类名调用  
        MyClass.myStaticMethod();  
  
        // 不能直接访问非静态方法  
        // MyClass.myInstanceMethod(); // ERROR  
  
        // 必须创建对象后才能调用非静态方法  
        MyClass obj = new MyClass();  
        obj.myInstanceMethod();  
    }  
}
```

类似静态方法，Java 中的静态变量（static 变量）是类级别的

在内存中只有一份拷贝

所有对象共享使用同一份数据

继承 Inheritance

抽象 Abstraction

多态 Polymorphism



# 继承

## Inheritance

一个类通过extends来继承另一个类的变量、属性和方法  
以此实现代码复用

Dog 和 Cat 是 Animal 的子类 (is-a 关系) ;  
不同种类动物可以有各自扩展。

```
public class Animal {  
    public void eat() {  
        System.out.println("动物在吃东西");  
    }  
}  
  
public class Dog extends Animal {  
    public void bark() {  
        System.out.println("狗: 汪汪");  
    }  
}  
  
public class Cat extends Animal {  
    public void meow() {  
        System.out.println("猫: 喵喵");  
    }  
}
```

# 抽象

## Abstraction

抽取出共有特征，不关心细节，只关心接口/行为

可以用抽象类（abstract class）或接口（interface）实现

统一接口/规范（每种动物必须实现 makeSound()）；

抽象类不能被直接实例化，只能被继承

```
public abstract class Animal {  
    public abstract void makeSound(); // 抽象方法  
  
    public void eat() {  
        System.out.println("动物在吃东西");  
    }  
}  
  
public class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("狗：汪汪！");  
    }  
}  
  
public class Cat extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("猫：喵喵！");  
    }  
}
```

# 多态

## Polymorphism

同一个方法在不同对象上有不同表现。分为编译时多态（重载）和运行时多态（重写）。

变量类型是 Animal（父类）

实际对象是 Dog / Cat

编译时看变量类型 → Animal

运行时看实际对象 → Dog / Cat。

同一个接口，调用同一个方法，但运行时表现不同。

```
public abstract class Animal {  
    public abstract void makeSound();  
    public void eat() {  
        System.out.println("动物在吃东西");  
    }  
}  
  
public class Dog extends Animal {  
    @Override //重载  
    public void makeSound() {  
        System.out.println("狗: 汪汪! ");  
    }  
}  
  
public class Cat extends Animal {  
    @Override //重载  
    public void makeSound() {  
        System.out.println("猫: 喵喵! ");  
    }  
}
```

关系 Relationship

# UML Relationship

## UML中的关系

[Animal] <|-- [Dog]           // 继承

[Flyable] <|.. [Bird]       // 接口实现

[Car] \*-- [Engine]         // 组合

[Car] o-- [Passenger]       // 聚合

[Teacher] --> [Student]     // 关联（成员变量）

[OrderService] ..> [Logger] // 依赖（临时使用）

# Life Cycle

## 生命周期

生命周期 (Life Cycle) 是类 (class) 之间的绑定关系

### 生命周期绑定

A 包含 B, A 死了 B也会死 (A 控制 B)

### 非生命周期绑定

A 引用 B, 但 B 独立存在, A 死了 B 可能还活着

# UML Relationship

## UML中的关系

|                             |             |       |
|-----------------------------|-------------|-------|
| [Animal] < -- [Dog]         | // 继承       | 【绑定】  |
| [Flyable] < .. [Bird]       | // 接口实现     | 【不绑定】 |
| [Car] *-- [Engine]          | // 组合       | 【强绑定】 |
| [Car] o-- [Passenger]       | // 聚合       | 【不绑定】 |
| [Teacher] --> [Student]     | // 关联（成员变量） | 【不绑定】 |
| [OrderService] ..> [Logger] | // 依赖（临时使用） | 【不绑定】 |

# Composition & Inheritance

## 组合与继承

“多用组合，少用继承”

既然 组合 (Composition) 和继承 (Inheritance) 都是“生命周期绑定”，那为什么Java 编程常说 “多用组合，少用继承”呢？



# Composition & Inheritance

## 组合与继承

“多用组合，少用继承”

继承：“is-a”，

树状继承，子类继承并暴露父类的所有public & protected方法

组合：“has-a”

嵌套应用，组合只暴露需要的方法，可以随时替换组合对象

# Composition & Inheritance - Example 1

## 组合与继承 - 示例 1

```
public class Animal {  
    public void makeSound() {  
        System.out.println("动物叫了一声");  
    }  
}  
  
public class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("狗: 汪汪! ");  
    }  
}
```

**Dog 继承了 Animal 的全部方法**

**强耦合：**Animal 一旦修改，Dog 必须跟着改；

**Dog 不能“替换声音模块”，只有一种叫声行为。**

# Composition - Delegation

## 组合中的委托

委托 (Delegation) 是一种通过组合 (has-a) + 方法转发实现行为复用的设计模式

Caller (调用者)



委托方法



Delegate (被委托者)

```
class Logger {
    public void log(String msg) {
        System.out.println("[日志] " + msg);
    }
}

class OrderService {
    private Logger logger = new Logger(); // 委托使用

    public void placeOrder(String product) {
        logger.log("下单: " + product); // 不自己记录日志, 交给 Logger
        System.out.println("订单完成");
    }
}
```

# Composition & Inheritance - Example 2

## 组合与继承 - 示例 2

```
public interface SoundBehavior {  
    void makeSound();  
}  
  
public class DogSound implements SoundBehavior {  
    public void makeSound() {  
        System.out.println("狗: 汪汪! ");  
    }  
}  
  
public class CatSound implements SoundBehavior {  
    public void makeSound() {  
        System.out.println("猫: 喵喵! ");  
    }  
}
```

```
public class Dog {  
    private SoundBehavior soundBehavior;  
  
    public Dog(SoundBehavior soundBehavior) {  
        this.soundBehavior = soundBehavior;  
    }  
  
    public void performSound() {  
        soundBehavior.makeSound();  
    }  
  
    // 支持动态更换行为  
    public void setSoundBehavior(SoundBehavior soundBehavior) {  
        this.soundBehavior = soundBehavior;  
    }  
}
```

# Composition & Inheritance - Example 2

## 组合与继承 - 示例 2

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Dog dog = new Dog(new DogSound());  
  
        dog.performSound(); // 狗：汪汪!  
  
        // 动态切换成猫叫（低耦合）  
  
        dog.setSoundBehavior(new CatSound());  
  
        dog.performSound(); // 猫：喵喵!  
  
    }  
  
}
```

# Composition & Inheritance

## 组合与继承

“多用组合，少用继承”

既然 组合 (Composition) 和继承 (Inheritance) 都是“生命周期绑定”，  
那为什么Java 编程常说 “多用组合，少用继承”呢？

继承获得的是结构和身份 (is-a)

组合获得的是能力和行为 (has-a + can-do)

如果只想复用行为，就使用组合或接口

# UML Relationship

## UML中的关系

[Animal] <|-- [Dog]            // 继承

[Flyable] <|.. [Bird]        // 接口实现

[Car] \*-- [Engine]            // 组合

[Car] o-- [Passenger]        // 聚合

[Teacher] --> [Student]      // 关联（成员变量）

[OrderService] ..> [Logger] // 依赖（临时使用）

# Composition & Implements

## 组合与接口实现

似乎都是解耦合，那么这两者又有什么区别？

组合的关键：通过包含另一个类/对象来执行行为委托，强绑定生命周期

接口的关键：包含interface和implements关键字，interface类内无实现，只提供范例接口，实现代码在implements类内



# Composition & Implements - Example 1

## 组合与接口实现 - 示例 1

```
class Wing {  
    public void flap() {  
        System.out.println("扑腾翅膀！");  
    }  
}  
  
class Bird {  
    private Wing wing = new Wing(); // Bird 包含 Wing  
  
    public void fly() {  
        wing.flap(); // 委托给 wing 来执行  
    }  
}
```

这里组合关系并没有直接实现 Flyable 接口，  
而是把“飞”的能力组合委托给了一个对象Wing

# Composition & Implements - Example 2

## 组合与接口实现 - 示例 2

```
interface Flyable {  
    void fly(); // 只有方法签名，无实现  
}  
  
class Bird implements Flyable {  
    public void fly() {  
        System.out.println("鸟在飞");  
    }  
}
```

interface: 接口说明书 (规范)

implements: 类实现接口 (实现能力)

“Bird 通过interface Flyable 承诺 我能飞，并实现接口定义的所有方法。”

# Composition & Implments

## 组合与接口实现

等等，接口实现 (Interface & Implements) 也有继承机制！

这里的继承机制并非前面提到的继承关系——接口实现本身 ≠ 类继承

```
interface Flyable {  
    void fly();  
}
```

```
class Bird implements Flyable {  
    public void fly() {  
        System.out.println("鸟在飞");  
    }  
}
```

} //这里是“类实现接口”，表示“Bird 能飞”，但这不是继承，而是“实现规范”

# Composition & Implments

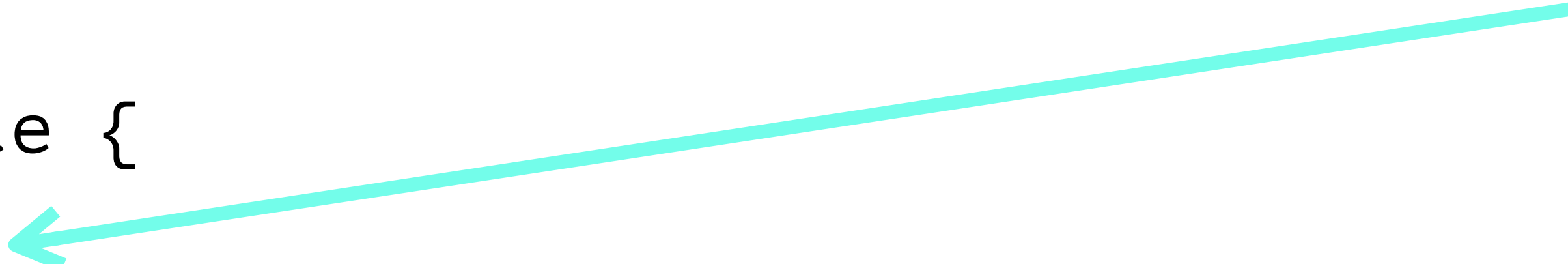
## 组合与接口实现

继承机制如下：

接口之间是可以通过 extends 继承的，并且是多继承，只继承方法签名

```
interface Moveable {  
    void move();  
}
```

```
interface Flyable extends Moveable {  
    void fly();  
} // 接口之间可以继承（这是真正的继承）
```



# Practice 1

## 小练习 1

```
// 接口声明我们的可驾驶签名方法
interface Drivable {
    void drive();
}

// 继承接口的新接口
interface SmartCar extends Drivable {
    void navigate();
}

// 普通的类
class Engine {
    public void start() {
        System.out.println("引擎启动
中...");
    }
}
```

```
// 针对drivable的实现方案/代码
class Car implements Drivable {
    protected Engine engine = new Engine(); // 新创
    建引擎对象，可用于组合

    @Override
    public void drive() {
        engine.start(); // 委托 delegation
        System.out.println("汽车行驶中...");
    }
}

// 针对smartcar实现，也继承了car
class ElectricCar extends Car implements SmartCar {

    @Override // 多态重载
    public void drive() {
        System.out.println("电动车安静地启动...");
    }

    @Override // 多态重载
    public void navigate() {
        System.out.println("正在自动导航到目的地...");
    }
}
```

# Practice 1

## 小练习 1

```
// 接口定义：可驾驶
interface Drivable {
    void drive();
}

// 接口继承：智能车 = 可驾驶 + 导航能力
interface SmartCar extends Drivable {
    void navigate();
}

// 组合类：引擎
class Engine {
    public void start() {
        System.out.println("引擎启动
中...");
    }
}
```

```
// 父类：Car (组合了 Engine, 实现了 Drivable)
class Car implements Drivable {
    protected Engine engine = new Engine(); // 组合

    @Override
    public void drive() {
        engine.start(); // 使用组合对象
        System.out.println("汽车行驶中...");
    }
}

// 子类：ElectricCar 继承 Car, 实现 SmartCar
class ElectricCar extends Car implements SmartCar {

    @Override
    public void drive() {
        System.out.println("电动车安静地启动...");
    }

    @Override
    public void navigate() {
        System.out.println("正在自动导航到目的地...");
    }
}
```

# UML Relationship

## UML中的关系

[Animal] <|-- [Dog] // 继承

[Flyable] <|.. [Bird] // 接口实现

[Car] \*-- [Engine] // 组合

[Car] o-- [Passenger] // 聚合

[Teacher] --> [Student] // 关联（成员变量）

[OrderService] ..> [Logger] // 依赖（临时使用）

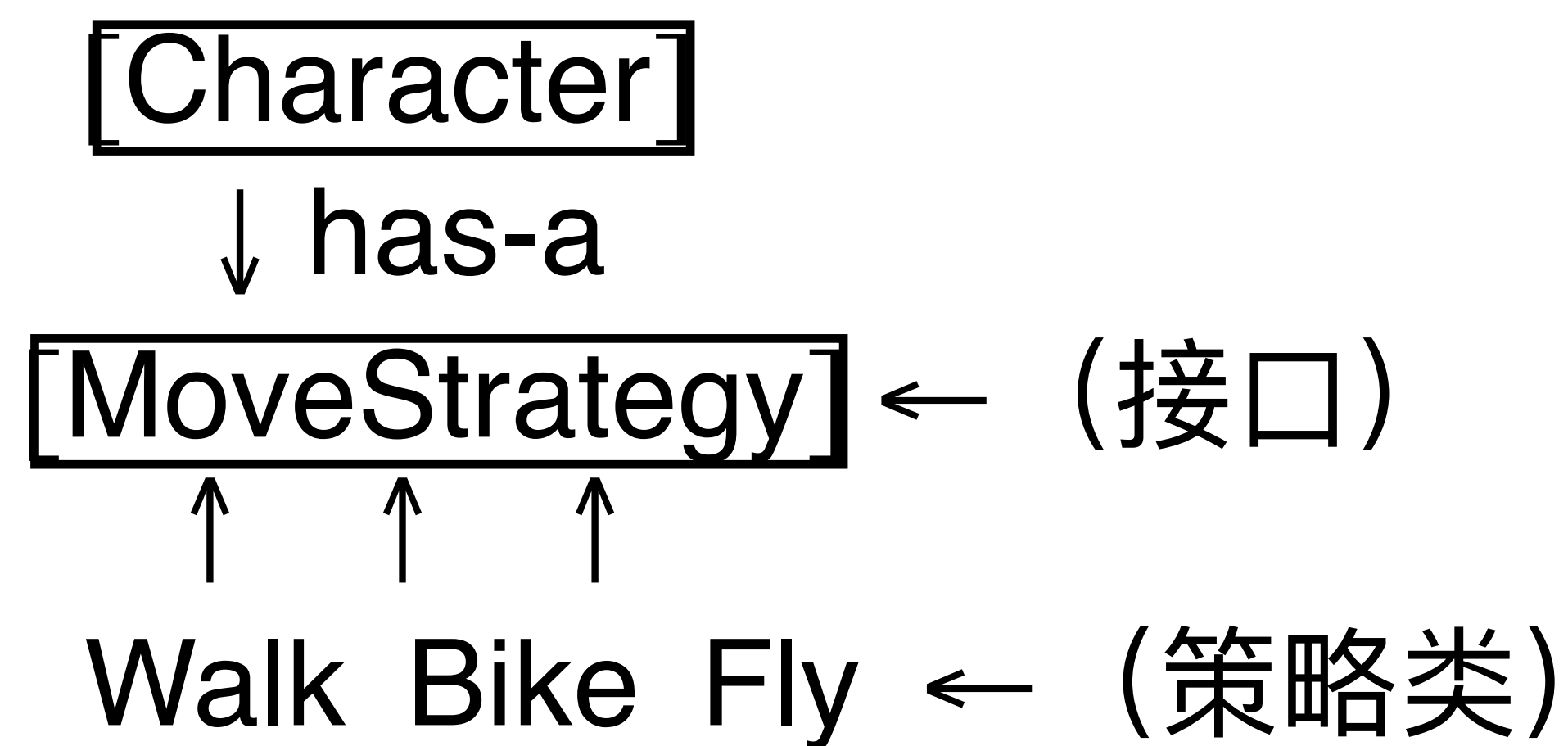
# 策略模式 Strategy Design



# Strategy Design

## 策略模式

策略模式是一种“可互换的行为封装”方式，能在运行时切换行为逻辑



角色拥有一个行为接口

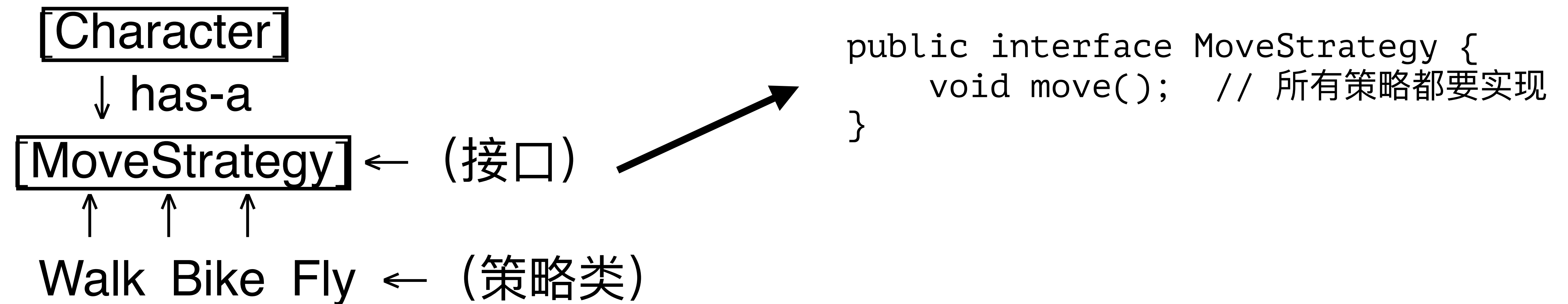
每个策略类实现这个接口

角色不关心怎么移动，只管“我现在用哪个策略”

# Strategy Design

## 策略模式

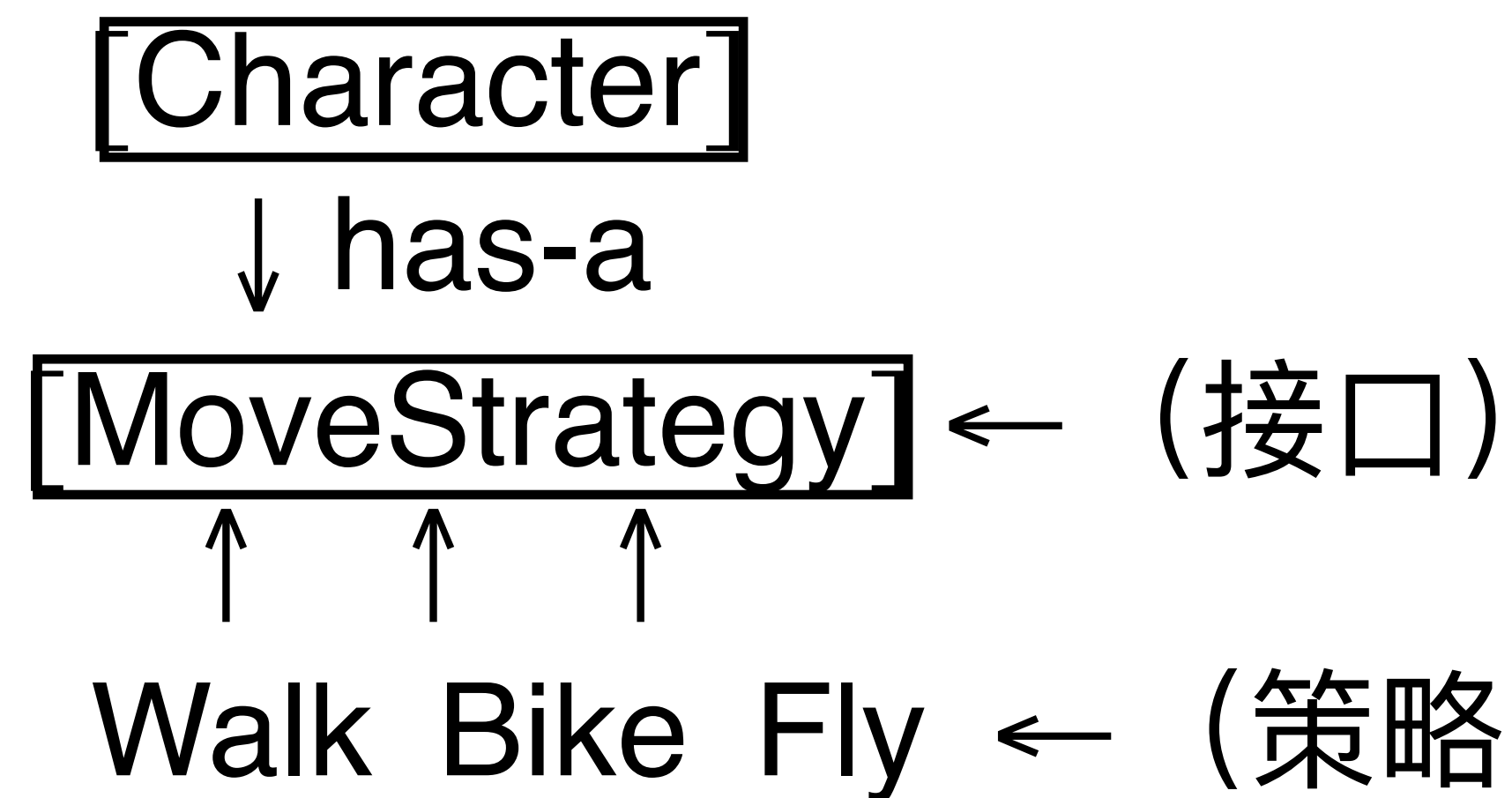
策略模式是一种“可互换的行为封装”方式，能在运行时切换行为逻辑



# Strategy Design

## 策略模式

策略模式是一种“可互换的行为封装”方式，能在运行时切换行为逻辑



```
public interface MoveStrategy {  
    void move(); // 所有策略都要实现  
}
```

```
public class WalkStrategy implements MoveStrategy {  
    public void move() {  
        System.out.println("角色正在用脚走路");  
    }  
}
```

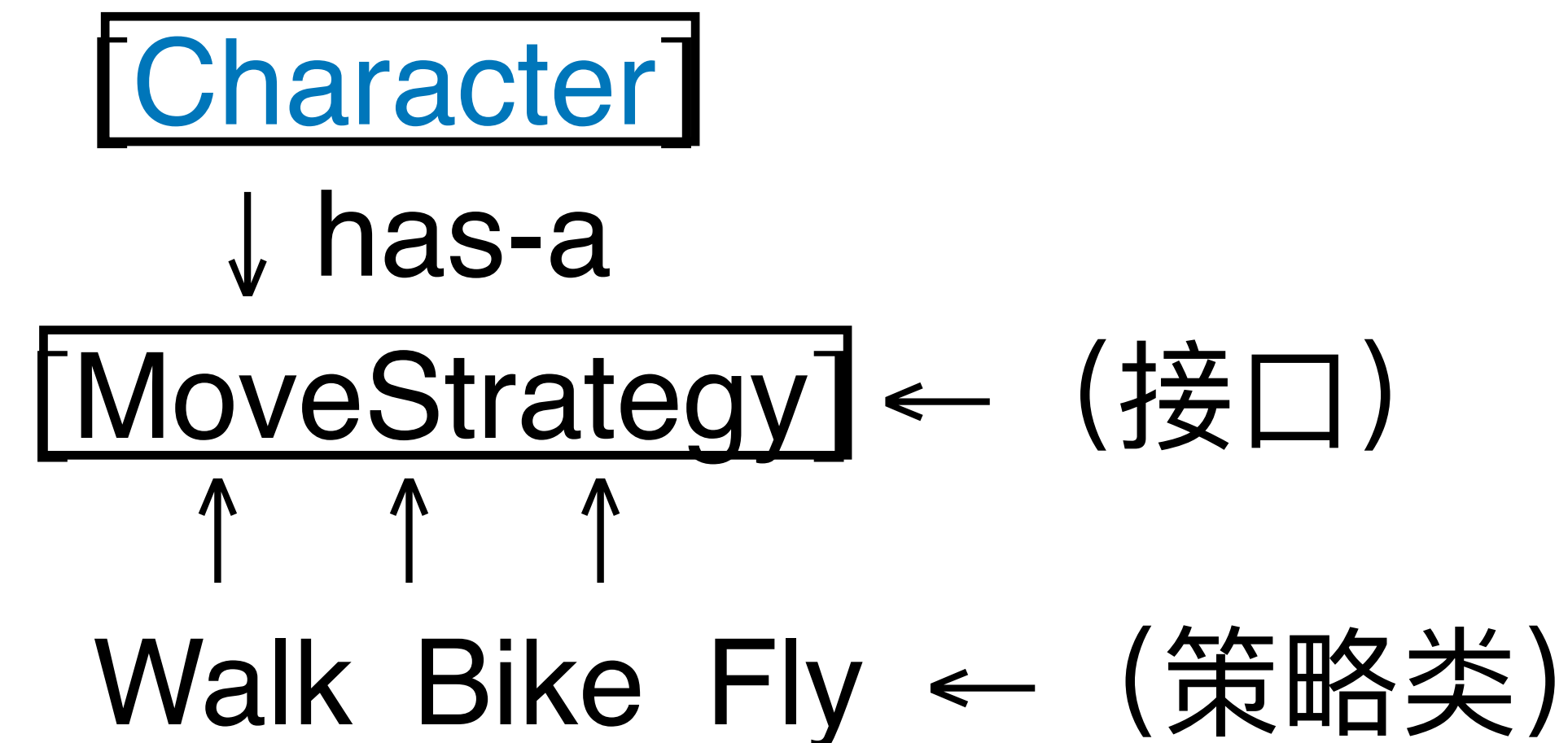
```
public class BikeStrategy implements MoveStrategy {  
    public void move() {  
        System.out.println("角色骑着自行车前进");  
    }  
}
```

```
public class FlyStrategy implements MoveStrategy {  
    public void move() {  
        System.out.println("角色飞在空中");  
    }  
}
```

# Strategy Design - Context Class

## 策略模式 - 上下文类

在策略模式中，上下文类负责使用策略对象的类



```
public class GameCharacter {
    private MoveStrategy strategy; // 上下文类持有策略接口

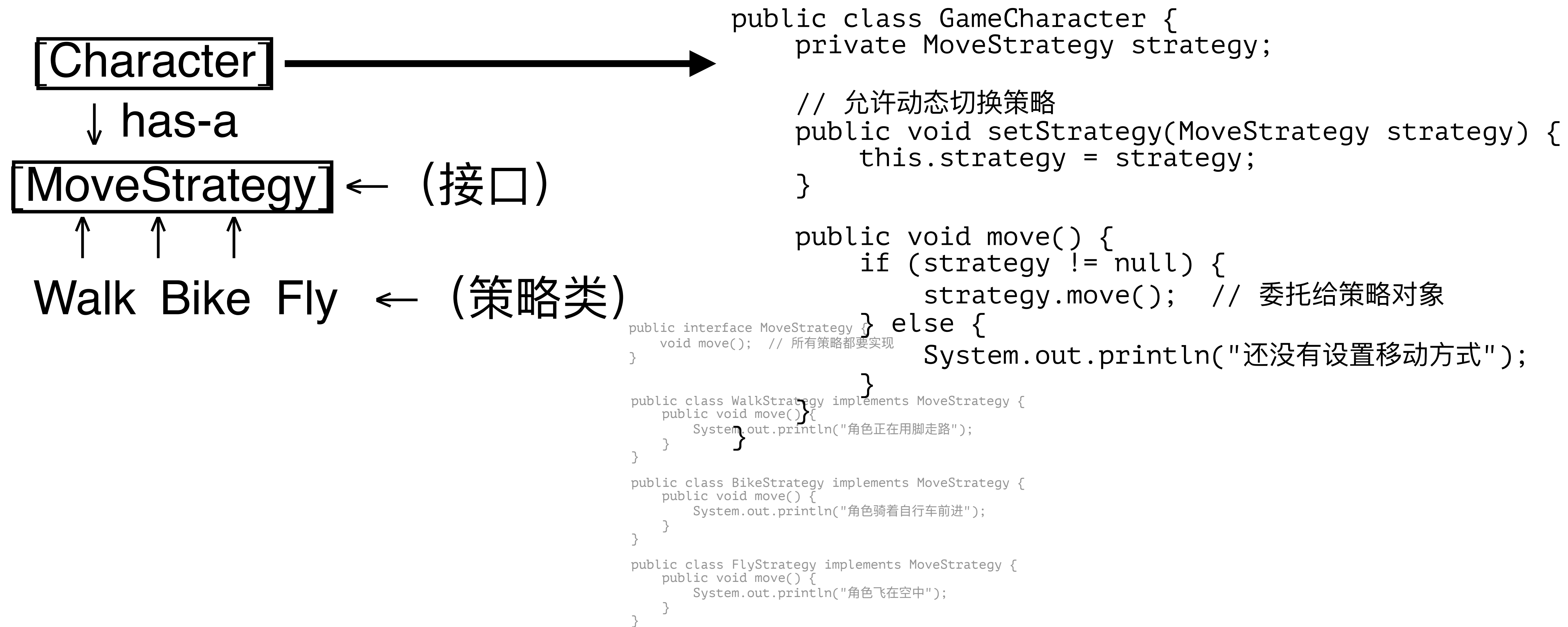
    public void setStrategy(MoveStrategy strategy) {
        this.strategy = strategy; // 允许动态设置
    }

    public void move() {
        if (strategy != null) {
            strategy.move(); // 委托执行
        } else {
            System.out.println("角色不知道怎么移动! ");
        }
    }
}
```

# Strategy Design

## 策略模式

策略模式是一种“可互换的行为封装”方式，能在运行时切换行为逻辑



# Strategy Design

## 策略模式

具体调用：

- **解耦算法和使用者**

主类不关心怎么做，把行为交给策略类

- **易于扩展**

增加一个策略，只需新建类即可，主类代码**0**改动

- **支持运行时切换行为**

可以在执行中自由切换策略，让程序“行为多变”

```
public class Main {  
    public static void main(String[] args) {  
        GameCharacter player = new GameCharacter()  
  
        // 切换策略：走路  
        player.setStrategy(new WalkStrategy());  
        player.move(); // 走路  
  
        // 切换策略：飞行  
        player.setStrategy(new FlyStrategy());  
        player.move(); // 飞行  
  
        // 切换策略：自行车  
        player.setStrategy(new BikeStrategy());  
        player.move(); // 骑车  
    }  
}
```

# Strategy Design - Practice

## 策略模式 - 练习

实现一个通用支付系统，可以支持以下三类支付方式（具体执行可以直接使用print表示）：

1. 刷卡支付 (Credit Card)
2. 现金支付 (Cash)
3. 电子支付 (E-Payment)
  - 微信支付 (WeChat)
  - 支付宝 (Alipay)

# Quiz

## 练习题

Q1. 下列关于 Java 接口的说法，正确的是？

- A. 接口可以有成员变量并赋初值
- B. 接口中的方法默认是 private
- C. 接口可以多继承
- D. 接口不能被任何类实现



# Quiz

## 练习题

Q2. 使用 implements 的类通常表示什么关系？

A. is-a (extends)

B. has-a (composition / delegation)

C. can-do (interface / implements)

D. depends-on

# Quiz

## 练习题

**Q3. Java 中，以下哪种关键字用于继承类？**

- A. interface
- B. extends
- C. implements
- D. super

# Quiz

## 练习题

**Q4. 策略模式的主要目的是：**

- A. 动态改变对象的类结构
- B. 把算法封装起来，使它们可以互换
- C. 自动调用所有子类方法
- D. 强化继承树的深度

# Quiz

## 练习题

**Q5.** 下列哪个是“组合”关系的特征?

- A. 子类继承父类的方法
- B. 对象内部 new 另一个类的实例
- C. 类实现接口
- D. 调用类的静态方法

# Quiz

## 练习题

**Q6.** “上下文类”在策略模式中主要负责：

- A. 定义所有策略的接口
- B. 实现所有策略逻辑
- C. 保存策略对象并委托执行行为
- D. 管理多个子类

# Quiz

## 练习题

**Q7. 下列哪个属于“委托”的实现形式?**

- A. 使用 `super` 调用父类方法
- B. 一个方法中调用成员对象的方法完成工作
- C. 用 `static` 方法封装逻辑
- D. 重写接口方法

# Quiz

## 练习题

**Q8. 接口不能包含以下哪种内容?**

- A. 方法签名
- B. default 方法
- C. 构造函数
- D. 被多个类实现

# Strategy Design - Practice

## 策略模式 - 练习

### 接口和策略类

- PayStrategy: 通用支付接口
- EPaymentStrategy extends PayStrategy: 电子支付子接口
- 实现类:
  - CreditCardPay implements PayStrategy
  - CashPay implements PayStrategy
  - WeChatPay implements EPaymentStrategy
  - AlipayPay implements EPaymentStrategy