CODEFORGE - BUỔI 16: INTERFACES

25 BÀI TẬP (18 CORE + 7 ADVANCED)

B01 - Interface Cơ Bản

Độ khó: ★★ Medium | **Loại:** Competitive

Đề bài

Bạn cần xây dựng một hệ thống vẽ hình học với **interface Drawable**. Interface này định nghĩa contract cho mọi hình có khả năng vẽ được.

Kiến thức áp dụng:

```
interface Drawable {
    void draw();  // Abstract method - bắt buộc implement
    double getArea();  // Tính diện tích
}

class Circle implements Drawable {
    private double radius;

    @Override
    public void draw() {
        System.out.println("Vẽ hình tròn bán kính " + radius);
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}

// ❌ Drawable d = new Drawable();  // COMPILE ERROR - không thể tạo instance
// ☑ Drawable d = new Circle(5);   // OK - reference qua interface
```

Các hình cần implement:

1. **Circle (Hình tròn):** có bán kính r
2. **Rectangle (Hình chữ nhật):** có chiều dài và rộng
3. **Triangle (Tam giác):** có 3 cạnh a, b, c

Yêu cầu cụ thể:

1. Tạo N hình từ input

2. Lưu tất cả vào `List<Drawable>` để demo polymorphism
3. Duyệt qua list và gọi `draw()` cho từng hình
4. Hiển thị thông tin chi tiết khi vẽ: loại hình, tham số, diện tích
5. Cuối cùng tính tổng diện tích tất cả các hình

Input

```
N
<loại_hình> <id> <các_tham_số>
```

Chi tiết từng loại:

- `CIRCLE <id> <bán_kính>` - Ví dụ: `CIRCLE c1 5.0`
- `RECTANGLE <id> <chiều_dài> <chiều_rộng>` - Ví dụ: `RECTANGLE r1 4.0 6.0`
- `TRIANGLE <id> <cạnh_a> <cạnh_b> <cạnh_c>` - Ví dụ: `TRIANGLE t1 3.0 4.0 5.0`

Output

```
=== HỆ THỐNG VẼ HÌNH (Interface Drawable) ===

Hình <thứ_tự>: <loại> <id>
- Tham số: <thông_tin_chi_tiết>
- draw(): <mô_tả_vẽ>
- getArea(): <diện_tích>

...

=== THỐNG KÊ ===
Tổng số hình: <N>
Tổng diện tích: <tổng>
```

Ví dụ

Input:

```
3
CIRCLE c1 5
RECTANGLE r1 4 6
TRIANGLE t1 3 4 5
```

Output:

```
=== VẼ 3 HÌNH ===
Hình 0: Circle c1
- draw(): Vẽ hình tròn bán kính 5.0
```

```
Hình 1: Rectangle r1
- draw(): Vẽ chữ nhật 4.0 × 6.0

Hình 2: Triangle t1
- draw(): Vẽ tam giác cạnh (3.0, 4.0, 5.0)

Tổng: 3 hình
```

B02 - Đa Kế Thừa Interface

Độ khó: ★★★ Hard | **Loại:** Competitive

Đề bài

Bạn đang xây dựng một **hệ thống UI components** cho ứng dụng. Mỗi component có thể có nhiều khả năng (capabilities) khác nhau:

- **Drawable:** Component có thể được vẽ trên màn hình
- **Clickable:** Component có thể nhận sự kiện click
- **Draggable:** Component có thể kéo thả
- **Resizable:** Component có thể thay đổi kích thước

Đặc điểm quan trọng: Một class có thể **implements NHIỀU interface** cùng lúc!

```
interface Drawable {
    void draw();
}

interface Clickable {
    void onClick();
}

interface Draggable {
    void onDrag(int x, int y);
}

// Button có 2 khả năng
class Button implements Drawable, Clickable {
    @Override
    public void draw() { ... }

    @Override
    public void onClick() { ... }
}

// Image có 3 khả năng
class Image implements Drawable, Draggable, Resizable {
    @Override
    public void draw() { ... }

    @Override
    public void onDrag(int x, int y) { ... }

    @Override
    public void resize(int w, int h) { ... }
}
```

Các loại component:

- **Button:** Drawable + Clickable (vẽ được và click được)
- **Image:** Drawable + Draggable + Resizable (vẽ được, kéo thả được, resize được)
- **Label:** Drawable only (chỉ vẽ được, không tương tác)
- **Canvas:** ALL 4 interfaces (có đầy đủ mọi khả năng)

Yêu cầu:

1. Tạo N components với các khả năng khác nhau
2. Thực hiện M thao tác trên các components
3. **Kiểm tra** xem component có hỗ trợ thao tác đó không (dùng `instanceof`)
4. Nếu không hỗ trợ → báo lỗi rõ ràng
5. Nếu hỗ trợ → thực hiện và mô tả chi tiết

Input

```
N
<loại_component> <id> <tham_số>

M
<thao_tác> <id> <tham_số_nếu_có>
```

Chi tiết component types:

- `BUTTON <id> <text>` - Ví dụ: `BUTTON btn1 Submit`
- `IMAGE <id> <src>` - Ví dụ: `IMAGE img1 photo.jpg`
- `LABEL <id> <text>` - Ví dụ: `LABEL lbl1 Title`
- `CANVAS <id>` - Ví dụ: `CANVAS cvs1`

Chi tiết thao tác:

- `DRAW <id>` - Vẽ component
- `CLICK <id>` - Click vào component
- `DRAG <id> <newX> <newY>` - Kéo component đến vị trí mới
- `RESIZE <id> <newWidth> <newHeight>` - Đổi kích thước

Output

```
=== THAO TÁC <thứ_tự>: <tên_thao_tác> ===
Component: <id> (<loại>)

Kiểm tra interface:
- implements Drawable: <Có/Không>
- implements Clickable: <Có/Không>
- implements Draggable: <Có/Không>
- implements Resizable: <Có/Không>
```

```
<kết_quả_thực_hiện_hoặc_thông_báo_lỗi>

---
```

Ví dụ

Input:

```
3
BUTTON btn1 Submit
IMAGE img1 photo.jpg 100 100
LABEL lbl1 Title
4
DRAW btn1
CLICK btn1
DRAG img1 150 150
CLICK lbl1
```

Output:

```
Thao tác 1: DRAW trên btn1
- Implements Drawable: Có ✓
→ Vẽ nút "Submit"

Thao tác 2: CLICK trên btn1
- Implements Clickable: Có ✓
→ Nút "Submit" được nhấn

Thao tác 3: DRAG trên img1
- Implements Draggable: Có ✓
→ Kéo ảnh từ (100, 100) → (150, 150)

Thao tác 4: CLICK trên lbl1
- Implements Clickable: Không ✗
→ LỖI: Label không hỗ trợ click

Tổng: 4 thao tác (3 thành công, 1 lỗi)
```

B03 - Kế Thừa Interface

Độ khó: ★ ★ ★ Hard | **Loại:** Competitive

Đề bài

Interface có thể kế thừa interface khác bằng `extends`:

```
interface Shape {
    double getArea();
}

interface ColoredShape extends Shape {
    String getColor();
    // Kế thừa getArea() từ Shape
}

class Circle implements ColoredShape {
    // PHẢI implement CẢ 2: getArea() + getColor()
}
```

Yêu cầu:

- Tạo N hình màu
- Hiển thị diện tích và màu

Input

```
N
<loại> <id> <tham_số> <màu>
```

Output

```
Hình <i>: <loại> <màu>
- getArea(): <diện_tích>
- getColor(): <màu>
```

Ví dụ

Input:

```
3
CIRCLE c1 5 Red
```

```
RECTANGLE r1 4 6 Blue
CIRCLE c2 3 Green
```

Output:

```
Hình 0: Circle Red
- getArea(): 78.54
- getColor(): Red

Hình 1: Rectangle Blue
- getArea(): 24.00
- getColor(): Blue

Hình 2: Circle Green
- getArea(): 28.27
- getColor(): Green

Tổng diện tích: 130.81
```

B04 - Default Method (Java 8+)

Độ khó: ★ ★ ★ Hard | **Loại:** Competitive

Đề bài

Java 8+ cho phép interface có **default method** (có implementation):

```
interface Vehicle {
    void start();  // Abstract - bắt buộc implement

    default void honk() {
        System.out.println("Bíp bíp!");
    }
}

class Car implements Vehicle {
    void start() { ... }
    // Có thể dùng default honk() hoặc override
}
```

Yêu cầu:

- Tạo N phương tiện
- Mỗi loại có thể dùng default hoặc override method `honk()`

Input

```
N
<loại> <id> <tên> <override_honk:Y/N>
```

Output

```
Xe <i>: <tên> (<loại>)
- start(): <khởi_động>
- honk(): <âm_thanh> [Default/Override]
```

Ví dụ

Input:

```
3
CAR c1 Toyota N
```

```
TRUCK t1 Ford Y
MOTORCYCLE m1 Honda N
```

Output:

```
Xe 0: Toyota (Car)
- start(): Khởi động xe con
- honk(): Bíp bíp! [Default]

Xe 1: Ford (Truck)
- start(): Khởi động xe tải
- honk(): ÚP ÚP! [Override - xe tải to hơn]

Xe 2: Honda (Motorcycle)
- start(): Khởi động xe máy
- honk(): Bíp bíp! [Default]
```

B05 - Static Method trong Interface

Độ khó: ★ ★ ★ Hard | **Loại:** Competitive

Đề bài

Java 8+ cho phép **static method** trong interface:

```
interface MathUtils {
    static int add(int a, int b) {
        return a + b;
    }
}

// Gọi qua tên interface
int sum = MathUtils.add(5, 3);
```

Yêu cầu: Thực hiện N phép tính toán học thông qua static methods trong interface.

Input

```
N
<phép_tính> <a> <b>
```

- Phép tính: ADD, MULTIPLY, MAX, MIN

Output

```
Phép <i>: MathUtils.<method>(<a>, <b>) = <kết_quả>
```

Ví dụ

Input:

```
4
ADD 5 3
MULTIPLY 4 2
MAX 10 7
MIN 3 8
```

Output:

```
Phép 1: MathUtils.add(5, 3) = 8
Phép 2: MathUtils.multiply(4, 2) = 8
Phép 3: MathUtils.max(10, 7) = 10
Phép 4: MathUtils.min(3, 8) = 3
```

B06 - Interface vs Abstract Class

Độ khó: ★★★ Hard | **Loại:** Competitive

Đề bài

Bạn là kiến trúc sư phần mềm, được yêu cầu **phân tích và đưa ra quyết định thiết kế** cho nhiều tình huống khác nhau: nên dùng **Interface** hay **Abstract Class**?

Bảng so sánh quan trọng:

Tiêu chí	Interface	Abstract Class
State (biến instance)	❌ Không có	☑ Có
Constructor	❌ Không có	☑ Có
Methods với implementation	⚠ Default methods (Java 8+)	☑ Concrete methods
Kế thừa	☑ Nhiều (implements nhiều)	❌ Đơn (extends 1)
Mục đích	Contract thuần túy	Share code + Contract
Khi nào dùng	Định nghĩa khả năng	Có code chung cần share

Ví dụ minh họa:

```
// ❌ Không thể làm với Interface (trước Java 8)
interface Employee {
    // ❌ Không có state
    String name;  // COMPILE ERROR

    // ❌ Không có constructor
    Employee(String name) { }  // COMPILE ERROR
}

// ☑ Abstract Class có thể
abstract class Employee {
    String name;  // ☑ OK - có state

    Employee(String name) {  // ☑ OK - có constructor
        this.name = name;
    }

    void displayInfo() {  // ☑ OK - concrete method
        System.out.println(name);
    }

    abstract double getSalary();  // Contract
}
```

```
// ☑ Interface cho multiple capabilities
interface Flyable { void fly(); }
interface Swimmable { void swim(); }

class Duck implements Flyable, Swimmable {
    // ☑ OK - có cả 2 khả năng
}

// ❌ Abstract Class không thể multiple inheritance
class Duck extends Bird, Fish {  // COMPILE ERROR
}
```

Yêu cầu:

Cho N tình huống thiết kế, mỗi tình huống có các đặc điểm:

- Có cần lưu trữ **state** (dữ liệu) không?
- Có cần **constructor** để khởi tạo không?
- Có cần **shared code** (code dùng chung) không?
- Có cần **multiple inheritance** (đa kế thừa) không?

Bạn cần:

1. Phân tích từng tình huống
2. Đưa ra khuyến nghị: INTERFACE hay ABSTRACT_CLASS
3. Giải thích lý do rõ ràng
4. Đưa ra ví dụ code mẫu minh họa

Input

```
N
SCENARIO <tên_tình_huống>
NEEDS_STATE <Y/N>
NEEDS_CONSTRUCTOR <Y/N>
NEEDS_SHARED_CODE <Y/N>
MULTIPLE_INHERITANCE <Y/N>
```

Ý nghĩa các trường:

- `NEEDS_STATE`: Cần lưu biến instance không? (ví dụ: id, name, salary)
- `NEEDS_CONSTRUCTOR`: Cần khởi tạo giá trị ban đầu không?
- `NEEDS_SHARED_CODE`: Có methods dùng chung cho tất cả subclass không?
- `MULTIPLE_INHERITANCE`: Cần kế thừa từ nhiều nguồn không?

Output

```
=== TÌNH HUỐNG <số>: <tên> ===

Phân tích yêu cầu:
- Cần state (biến instance): <Y/N>
  → <giải_thích_cụ_thể>
- Cần constructor: <Y/N>
  → <giải_thích_cụ_thể>
- Cần shared code: <Y/N>
  → <giải_thích_cụ_thể>
- Cần đa kế thừa: <Y/N>
  → <giải_thích_cụ_thể>

💡 KHUYẾN NGHỊ: <INTERFACE / ABSTRACT_CLASS / CẢ HAI>

Lý do chính:
- <lý_do_1>
- <lý_do_2>
- <lý_do_3>

Ví dụ thiết kế:
<code_example_minh_họa>

---
```

Ví dụ

Input:

```
2
SCENARIO Payment_System
NEEDS_STATE Y
NEEDS_CONSTRUCTOR Y
MULTIPLE_TYPES N
SCENARIO Plugin_System
NEEDS_STATE N
NEEDS_CONSTRUCTOR N
MULTIPLE_TYPES Y
```

Output:

```
Tình huống 1: Payment_System
- Cần state: Có
- Cần constructor: Có
- Đa kế thừa: Không

→ Khuyến nghị: ABSTRACT CLASS
Lý do: Cần lưu trữ dữ liệu chung (state) và khởi tạo (constructor)
```

```
Tình huống 2: Plugin_System
- Cần state: Không
- Cần constructor: Không
- Đa kế thừa: Có

→ Khuyến nghị: INTERFACE
Lý do: Contract thuần, cho phép plugin có nhiều khả năng
```

B07 - Marker Interface

Độ khó: ★★★ Hard | **Loại:** Competitive

Đề bài

Marker Interface = Interface rỗng dùng để đánh dấu:

```
interface Serializable {
    // RỖNG - chỉ để đánh dấu
}

class User implements Serializable {
    // Giờ có thể serialize
}
```

Yêu cầu:

- Tạo N objects với các marker khác nhau (Serializable, Cacheable, Loggable)
- Thực hiện M thao tác, kiểm tra marker trước khi thực hiện

Input

```
N
<class> <id> <Serializable:Y/N> <Cacheable:Y/N> <Loggable:Y/N>

M
<thao_tác> <id>
```

Output

```
Thao_tác <i>: <thao_tác> trên <id>
- instanceof <Marker>: <Có/Không>
<kết_quả>
```

Ví dụ

Input:

```
3
User u1 Y Y N
Product p1 Y N Y
Session s1 N Y N
```

```
4
SERIALIZE u1
CACHE p1
LOG p1
SERIALIZE s1
```

Output:

```
Thao tác 1: SERIALIZE trên u1
- instanceof Serializable: Có ✓
→ Serialize thành công

Thao tác 2: CACHE trên p1
- instanceof Cacheable: Không ✗
→ LỖI: Product không hỗ trợ cache

Thao tác 3: LOG trên p1
- instanceof Loggable: Có ✓
→ Ghi log thành công

Thao tác 4: SERIALIZE trên s1
- instanceof Serializable: Không ✗
→ LỖI: Session không thể serialize
```

B08 - Functional Interface

Độ khó: ★★★★ Very Hard | **Loại:** Competitive

Đề bài

Functional Interface = Interface có ĐÚNG 1 abstract method:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);  // DUY NHẤT abstract method
}

// Có thể dùng lambda
Calculator add = (a, b) -> a + b;
```

Yêu cầu:

- Đăng ký N functional interfaces
- Thực hiện M phép tính

Input

```
N
<tên_interface> <operation>

M
<interface> <a> <b>
```

Output

```
Phép <i>: <interface>(<a>, <b>)
Lambda: (a, b) -> <expression>
Kết quả: <value>
```

Ví dụ

Input:

```
3
Adder ADD
Multiplier MULTIPLY
MaxFinder MAX
```

```
4
Adder 5 3
Multiplier 4 2
MaxFinder 10 7
Adder 100 50
```

Output:

```
Phép 1: Adder(5, 3)
Lambda: (a, b) -> a + b
Kết quả: 8

Phép 2: Multiplier(4, 2)
Lambda: (a, b) -> a * b
Kết quả: 8

Phép 3: MaxFinder(10, 7)
Lambda: (a, b) -> (a > b) ? a : b
Kết quả: 10

Phép 4: Adder(100, 50)
Lambda: (a, b) -> a + b
Kết quả: 150
```

B09 - Hằng Số trong Interface

Độ khó: ★★ Medium | **Loại:** Competitive

Đề bài

Interface có thể chứa hằng số (tự động `public static final`):

```
interface HttpStatus {
    int OK = 200;
    int NOT_FOUND = 404;
    int SERVER_ERROR = 500;
}

int code = HttpStatus.OK;  // 200
```

Yêu cầu: Nhập N status code, tìm tên hằng số tương ứng.

Input

```
N
<status_code>
```

Output

```
Code <i>: <code>
→ HttpStatus.<TÊN_HẰNG>: <mô_tả>
```

Ví dụ

Input:

```
4
200
404
500
403
```

Output:

```
Code 1: 200
→ HttpStatus.OK: Yêu cầu thành công

Code 2: 404
→ HttpStatus.NOT_FOUND: Không tìm thấy

Code 3: 500
→ HttpStatus.SERVER_ERROR: Lỗi server

Code 4: 403
→ Không có hằng số (Forbidden)
```

B10 - Nguyên Tắc Phân Tách Interface (ISP)

Độ khó: ★★★★ Very Hard | **Loại:** Competitive

Đề bài

Bạn cần áp dụng **Interface Segregation Principle (ISP)** - một trong 5 nguyên tắc SOLID.

ISP nói gì? "*Clients should not be forced to depend on interfaces they don't use*"
→ Nghĩa là: Đừng ép một class phải implement các methods mà nó không dùng đến!

Ví dụ vi phạm ISP (thiết kế tệ):

```
// ❌ Fat Interface - Gộp quá nhiều methods
interface Worker {
    void work();
    void eat();
    void sleep();
}

class Human implements Worker {
    public void work() { System.out.println("Làm việc"); }
    public void eat() { System.out.println("Ăn uống"); }
    public void sleep() { System.out.println("Ngủ"); }
}

class Robot implements Worker {
    public void work() { System.out.println("Làm việc"); }
    public void eat() { /* ❌ Rỗng - lãng phí! */ }
    public void sleep() { /* ❌ Rỗng - lãng phí! */ }
}

// ❌ Vấn đề: Robot bị ép implement eat() và sleep()
// nhưng nó không cần!
```

Thiết kế tốt (tuân thủ ISP):

```
// ☑ Tách thành nhiều interface nhỏ
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

interface Sleepable {
    void sleep();
```

```
}

// Human cần cả 3
class Human implements Workable, Eatable, Sleepable {
    public void work() { System.out.println("Làm việc"); }
    public void eat() { System.out.println("Ăn uống"); }
    public void sleep() { System.out.println("Ngủ"); }
}

// Robot chỉ cần 1
class Robot implements Workable {
    public void work() { System.out.println("Làm việc"); }
    // ☑ Sạch sẽ - không có dummy methods!
}
```

Yêu cầu:

1. Tạo N workers với các khả năng khác nhau:
    - Human: work + eat + sleep
    - Robot: work only
    - Animal: eat + sleep only
2. Thực hiện M hành động trên workers
3. **Kiểm tra ISP:**
    - Trước khi thực hiện hành động, kiểm tra xem worker có implements interface tương ứng không
    - Nếu KHÔNG → báo lỗi rõ ràng (đây là điểm hay của ISP!)
    - Nếu CÓ → thực hiện hành động
4. Cuối cùng đưa ra nhận xét về lợi ích của ISP

Input

```
N
<loại_worker> <id> <work:Y/N> <eat:Y/N> <sleep:Y/N>

M
<hành_động> <id>
```

Chi tiết loại worker:

- `HUMAN <id> <work> <eat> <sleep>` - Ví dụ: `HUMAN h1 Y Y Y`
- `ROBOT <id> <work> <eat> <sleep>` - Ví dụ: `ROBOT r1 Y N N`
- `ANIMAL <id> <work> <eat> <sleep>` - Ví dụ: `ANIMAL a1 N Y Y`

Chi tiết hành động:

- `WORK <id>` - Yêu cầu làm việc

- `EAT <id>` - Yêu cầu ăn
- `SLEEP <id>` - Yêu cầu ngủ

Output

```
=== WORKER <số>: <id> (<loại>) ===
Interfaces implemented:
- Workable: <Có/Không>
- Eatable: <Có/Không>
- Sleepable: <Có/Không>

=== HÀNH ĐỘNG <số>: <action> ===
Worker: <id>

Kiểm tra interface:
- Cần interface: <Interface>
- Worker có implement: <Có/Không>

<kết_quả_hoặc_lỗi>

---

=== PHÂN TÍCH ISP ===
<nhận_xét_về_lợi_ích_của_ISP>
```

Ví dụ

Input:

```
2
HUMAN h1 Y Y Y
ROBOT r1 Y N N
3
WORK h1
EAT r1
SLEEP h1
```

Output:

```
Hành động 1: WORK bởi h1
- Implements Workable: Có ✓
→ Human làm việc

Hành động 2: EAT bởi r1
- Implements Eatable: Không ✗
→ LỖI: Robot không ăn được

Hành động 3: SLEEP bởi h1
```

```
- Implements Sleepable: Có ✓
→ Human ngủ

Lợi ích ISP: Robot không bị ép implement eat() và sleep()
```

B11 - Interface Comparable

Độ khó: ★ ★ ★ Hard | **Loại:** Competitive

Đề bài

Java cung cấp interface `Comparable<T>` để so sánh và sắp xếp:

```
interface Comparable<T> {
    int compareTo(T other);
    // < 0: this < other
    // = 0: this == other
    // > 0: this > other
}

class Student implements Comparable<Student> {
    public int compareTo(Student other) {
        return Double.compare(this.gpa, other.gpa);
    }
}
```

Yêu cầu:

- Nhập N sinh viên
- Sắp xếp theo tiêu chí (GPA/NAME/ID)

Input

```
N
<id> <tên> <gpa>
<tiêu_chí_sắp_xếp>
```

Output

```
Trước sắp xếp:
<danh_sách>

Sắp xếp theo <tiêu_chí>...

Sau sắp xếp:
<danh_sách_đã_sắp>
```

Ví dụ

Input:

```
4
103 Charlie 3.2
101 Alice 3.8
102 Bob 3.9
104 Diana 3.1
GPA
```

Output:

```
Trước sắp xếp:
1. Charlie (GPA: 3.2)
2. Alice (GPA: 3.8)
3. Bob (GPA: 3.9)
4. Diana (GPA: 3.1)

Sắp xếp theo GPA (tăng dần)...

Sau sắp xếp:
1. Diana (GPA: 3.1)
2. Charlie (GPA: 3.2)
3. Alice (GPA: 3.8)
4. Bob (GPA: 3.9)
```

B12 - Kết Hợp Nhiều Interface

Độ khó: ★★★★ Very Hard | **Loại:** Competitive

Đề bài

Xây dựng hệ thống UI phức tạp với nhiều interface:

- `Drawable`, `Clickable`, `Draggable`, `Resizable`

Component khác nhau có khả năng khác nhau:

- Button: Drawable + Clickable
- Image: Drawable + Draggable + Resizable
- Canvas: CẢ 4 interface

Yêu cầu:

- Tạo N components
- Thực hiện M thao tác, kiểm tra khả năng

Input

```
N
<loại> <id> <tham_số>

M
<thao_tác> <id> <tham_số>
```

Output

```
Thao_tác <i>: <action> trên <id>
<kiểm_tra_interface>
<kết_quả>
```

Ví dụ

Input:

```
3
BUTTON btn1 Submit
IMAGE img1 photo.jpg
CANVAS cvs1
4
DRAW btn1
```

```
CLICK btn1
RESIZE img1 300 200
DRAG cvs1 50 50
```

Output:

```
Thao tác 1: DRAW trên btn1
→ Vẽ nút Submit

Thao tác 2: CLICK trên btn1
→ Nhấn nút Submit

Thao tác 3: RESIZE trên img1
→ Đổi kích thước ảnh: 300×200

Thao tác 4: DRAG trên cvs1
→ Kéo canvas đến (50, 50)

Tổng: 4 thao tác thành công
```

B13 - Xung Đột Default Method

Độ khó: ★★★★ Very Hard | **Loại:** Competitive

Đề bài

Khi class implements 2 interface có cùng default method → **xung đột**:

```
interface A {
    default void print() { System.out.println("A"); }
}

interface B {
    default void print() { System.out.println("B"); }
}

class C implements A, B {
    // LỖI BIÊN DỊCH - phải giải quyết xung đột

    @Override
    public void print() {
        A.super.print();  // Chọn A
        // HOẶC B.super.print();  // Chọn B
        // HOẶC custom implementation
    }
}
```

Yêu cầu:

- N class với xung đột
- Mỗi class chọn cách giải quyết (USE_A, USE_B, CUSTOM)

Input

```
N
<className> <interface1> <interface2> <giải_pháp>
```

Output

```
Class <i>: <name>
Xung đột: <if1>.print() vs <if2>.print()
Giải pháp: <solution>
Kết quả: <output>
```

Ví dụ

Input:

```
3
MyClass Printable Loggable USE_A
YourClass Drawable Renderable USE_B
OurClass Display Showable CUSTOM
```

Output:

```
Class 1: MyClass
Xung đột: Printable.print() vs Loggable.print()
Giải pháp: USE_A
Kết quả: Gọi Printable.super.print()

Class 2: YourClass
Xung đột: Drawable.print() vs Renderable.print()
Giải pháp: USE_B
Kết quả: Gọi Renderable.super.print()

Class 3: OurClass
Xung đột: Display.print() vs Showable.print()
Giải pháp: CUSTOM
Kết quả: Implementation riêng
```

B14 - Interface trong Collections

Độ khó: ★ ★ ★ Hard | **Loại:** Competitive

Đề bài

Sử dụng interface để lưu trữ đa hình trong Collection:

```
List<Drawable> shapes = new ArrayList<>();
shapes.add(new Circle());
shapes.add(new Rectangle());

for (Drawable s : shapes) {
    s.draw();  // Polymorphism
}
```

Yêu cầu:

- Quản lý danh sách Drawable
- Thao tác: ADD, REMOVE, DRAW_ALL, FILTER, SIZE

Input

```
N
<loại> <id> <tham_số>

M
<thao_tác> <tham_số>
```

Output

```
Thao_tác <i>: <action>
<kết_quả>
```

Ví dụ

Input:

```
3
CIRCLE c1 5
RECTANGLE r1 4 6
TRIANGLE t1 3 4 5
5
```

```
ADD c1
ADD r1
DRAW_ALL
FILTER CIRCLE
SIZE
```

Output:

```
Thao tác 1: ADD c1
→ Đã thêm Circle c1

Thao tác 2: ADD r1
→ Đã thêm Rectangle r1

Thao tác 3: DRAW_ALL
→ Vẽ Circle c1
→ Vẽ Rectangle r1

Thao tác 4: FILTER CIRCLE
→ Tìm thấy: Circle c1

Thao tác 5: SIZE
→ Kích thước: 2
```

B15 - Interface với Generics

Độ khó: ★★★★ Very Hard | **Loại:** Competitive

Đề bài

Interface có thể có type parameter (generic):

```
interface Converter<F, T> {
    T convert(F from);
}

class StringToInt implements Converter<String, Integer> {
    public Integer convert(String from) {
        return Integer.parseInt(from);
    }
}
```

Yêu cầu:

- Đăng ký N converters
- Thực hiện M conversions

Input

```
N
<converter_type> <id>

M
<id> <input>
```

Output

```
Chuyển_đổi <i>: <id>
<F> → <T>
Kết quả: <output>
```

Ví dụ

Input:

```
3
STRING_TO_INT conv1
```

```
INT_TO_STRING conv2
STRING_TO_DOUBLE conv3
4
conv1 123
conv2 456
conv3 3.14
conv1 abc
```

Output:

```
Chuyển đổi 1: conv1
String → Integer
Kết quả: 123

Chuyển đổi 2: conv2
Integer → String
Kết quả: "456"

Chuyển đổi 3: conv3
String → Double
Kết quả: 3.14

Chuyển đổi 4: conv1
String → Integer
Kết quả: LỖI (định dạng không hợp lệ)
```

B16 - Repository Pattern

Độ khó: ★★★★ Very Hard | **Loại:** Competitive

Đề bài

Repository pattern với interface để trừu tượng hóa data access:

```
interface Repository<T, ID> {
    T findById(ID id);
    List<T> findAll();
    void save(T entity);
    void delete(ID id);
}
```

Yêu cầu: Implement UserRepository với CRUD operations.

Input

```
N
SAVE <id> <tên> <email>
FIND <id>
FIND_ALL
DELETE <id>
UPDATE <id> <field> <value>
```

Output

```
Thao_tác <i>: <action>
<kết_quả>
```

Ví dụ

Input:

```
5
SAVE U001 Alice alice@email.com
SAVE U002 Bob bob@email.com
FIND U001
DELETE U002
FIND_ALL
```

Output:

```
Thao tác 1: SAVE U001
→ Đã lưu User: Alice

Thao tác 2: SAVE U002
→ Đã lưu User: Bob

Thao tác 3: FIND U001
→ Tìm thấy: Alice (alice@email.com)

Thao tác 4: DELETE U002
→ Đã xóa User: Bob

Thao tác 5: FIND_ALL
→ 1 user: Alice
```

B17 - Strategy Pattern

Độ khó: ★★★★ Very Hard | **Loại:** Competitive

Đề bài

Bạn đang xây dựng **hệ thống thanh toán** cho một trang e-commerce. Khách hàng có thể thanh toán bằng nhiều phương thức khác nhau, mỗi phương thức có cách tính phí và xử lý riêng.

Strategy Pattern giúp bạn:

- Định nghĩa **family of algorithms** (họ các thuật toán)
- Đóng gói mỗi thuật toán trong một class riêng
- Cho phép **thay đổi thuật toán tại runtime** (không cần sửa code)

Cấu trúc Strategy Pattern:

```
// Strategy Interface - định nghĩa contract
interface PaymentStrategy {
    void pay(double amount);
}

// Concrete Strategies - các thuật toán cụ thể
class CreditCardStrategy implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Thanh toán " + amount + " bằng thẻ");
        // Phí 2%, xác thực thẻ, etc.
    }
}

class PayPalStrategy implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Thanh toán " + amount + " qua PayPal");
        // Phí 3%, OAuth, etc.
    }
}

// Context - sử dụng Strategy
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    // Thay đổi strategy tại runtime
    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(double amount) {
        paymentStrategy.pay(amount);  // Gọi strategy hiện tại
```

```
    }
}

// Client code
ShoppingCart cart = new ShoppingCart();
cart.setPaymentStrategy(new CreditCardStrategy());
cart.checkout(1000000);  // Dùng thẻ

cart.setPaymentStrategy(new PayPalStrategy());
cart.checkout(500000);   // Đổi sang PayPal
```

Các phương thức thanh toán:

1. **CREDIT_CARD** (Thẻ tín dụng)
    - Phí: 2% tổng tiền
    - Xác thực: Kiểm tra số thẻ
    - Thời gian xử lý: Ngay lập tức
2. **PAYPAL**
    - Phí: 3% tổng tiền
    - Xác thực: OAuth với tài khoản PayPal
    - Thời gian xử lý: Ngay lập tức
3. **CASH** (Tiền mặt)
    - Phí: 0đ
    - Xác thực: Không cần
    - Thời gian xử lý: Khi nhận hàng
4. **CRYPTO** (Cryptocurrency)
    - Phí: Network fee (~50,000đ)
    - Xác thực: Blockchain
    - Thời gian xử lý: 10-30 phút (chờ confirm)

Yêu cầu:

1. Nhập N giao dịch thanh toán
2. Mỗi giao dịch chọn 1 strategy
3. Xử lý thanh toán:
    - Tính phí theo từng phương thức
    - Hiển thị chi tiết xử lý
    - Tính tổng tiền thực tế khách phải trả
4. Cuối cùng thống kê:
    - Tổng số giao dịch
    - Tổng tiền gốc
    - Tổng phí
    - Tổng tiền thực thu

Input

```
N
<strategy> <số_tiền> <thông_tin_thêm>
```

Chi tiết từng strategy:

- `CREDIT_CARD <số_tiền> <số_thẻ>` - Ví dụ: `CREDIT_CARD 1000000 1234-5678-9012-3456`
- `PAYPAL <số_tiền> <email>` - Ví dụ: `PAYPAL 500000 user@paypal.com`
- `CASH <số_tiền> -` - Ví dụ: `CASH 200000 -`
- `CRYPTO <số_tiền> <wallet>` - Ví dụ: `CRYPTO 2000000 0x1234abcd`

Output

```
=== GIAO DỊCH <số> ===
Strategy: <tên_strategy>
Số tiền gốc: <amount>đ

Chi tiết xử lý:
- <bước_1>
- <bước_2>
- <bước_3>

Phí giao dịch: <fee>đ (<tỷ_lệ>)
Tổng thanh toán: <total>đ
Trạng thái: <SUCCESS/PENDING>

---

=== THỐNG KÊ TỔNG ===
Tổng giao dịch: <N>
Tổng tiền gốc: <sum>đ
Tổng phí: <total_fee>đ
Tổng thực thu: <total_collected>đ

Phân bố theo strategy:
- CREDIT_CARD: <count> giao dịch
- PAYPAL: <count> giao dịch
- CASH: <count> giao dịch
- CRYPTO: <count> giao dịch
```

Ví dụ

Input:

```
4
CREDIT_CARD 1000000
```

```
PAYPAL 500000
CASH 200000
CRYPTO 2000000
```

Output:

```
Thanh toán 1: CREDIT_CARD
Số tiền: 1,000,000đ
Phí: 20,000đ (2%)
Tổng: 1,020,000đ

Thanh toán 2: PAYPAL
Số tiền: 500,000đ
Phí: 15,000đ (3%)
Tổng: 515,000đ

Thanh toán 3: CASH
Số tiền: 200,000đ
Phí: 0đ
Tổng: 200,000đ

Thanh toán 4: CRYPTO
Số tiền: 2,000,000đ
Phí: 50,000đ
Tổng: 2,050,000đ
```

B18 - Observer Pattern

Độ khó: ★★★★★ Expert | **Loại:** Competitive

Đề bài

Observer pattern với interface để tạo hệ thống thông báo:

```
interface Observer {
    void update(String event);
}

interface Subject {
    void attach(Observer obs);
    void notify(String event);
}
```

Yêu cầu: Xây dựng hệ thống thông báo tin tức.

Input

```
N
<observer_id> <loại>

M
ATTACH <id>
DETACH <id>
NOTIFY <tin_tức>
```

Output

```
Thao_tác <i>: <action>
<kết_quả>
```

Ví dụ

Input:

```
3
obs1 EMAIL
obs2 SMS
obs3 PUSH
5
```

```
ATTACH obs1
ATTACH obs2
NOTIFY Breaking_News
DETACH obs1
NOTIFY Update
```

Output:

```
Thao tác 1: ATTACH obs1
→ Đăng ký EMAIL observer

Thao tác 2: ATTACH obs2
→ Đăng ký SMS observer

Thao tác 3: NOTIFY Breaking_News
→ EMAIL nhận: Breaking_News
→ SMS nhận: Breaking_News

Thao tác 4: DETACH obs1
→ Hủy EMAIL observer

Thao tác 5: NOTIFY Update
→ SMS nhận: Update
```

B19A - Plugin System

Độ khó: ★★★★★ Expert | **Loại:** Advanced

Đề bài

Xây dựng plugin system với interface và dependency injection:

```
interface Plugin {
    void initialize();
    void execute();
    void cleanup();
}
```

Yêu cầu:

- Đăng ký N plugins
- Quản lý lifecycle và dependencies

Input

```
N
<plugin_type> <id> <depends_on>

M
LOAD <id>
EXECUTE <id>
UNLOAD <id>
```

Output

```
Plugin <i>: <id>
<lifecycle_events>
```

Ví dụ

Input:

```
3
DATA_SOURCE ds1 -
TRANSFORM tf1 ds1
OUTPUT out1 tf1
3
```

```
LOAD out1
EXECUTE out1
UNLOAD out1
```

Output:

```
Plugin 1: LOAD out1
→ Phụ thuộc: tf1, ds1
→ Load ds1 → tf1 → out1

Plugin 2: EXECUTE out1
→ ds1: Load data
→ tf1: Transform
→ out1: Output result

Plugin 3: UNLOAD out1
→ Cleanup: out1 → tf1 → ds1
```

B20A - Service Layer

Độ khó: ★★★★★ Expert | **Loại:** Advanced

Đề bài

Service layer architecture với interface:

```
interface UserService {
    User createUser(UserDTO dto);
    User getUser(String id);
    void updateUser(String id, UserDTO dto);
    void deleteUser(String id);
}
```

Yêu cầu: Implement business logic layer với validation và transaction.

Input

```
N
CREATE <id> <data>
GET <id>
UPDATE <id> <field> <value>
DELETE <id>
```

Output

```
Service <i>: <operation>
<validation_và_kết_quả>
```

Ví dụ

Input:

```
5
CREATE U001 Alice 20
GET U001
UPDATE U001 age 21
DELETE U001
GET U001
```

Output:

```
Service 1: CREATE U001
→ Validate: OK
→ Business logic: OK
→ Tạo user: Alice (20 tuổi)

Service 2: GET U001
→ Tìm thấy: Alice

Service 3: UPDATE U001
→ Cập nhật age: 20 → 21

Service 4: DELETE U001
→ Đã xóa user

Service 5: GET U001
→ Không tìm thấy
```

B21A - Data Access Layer (DAO)

Độ khó: ★★★★★ Expert | **Loại:** Advanced

Đề bài

DAO pattern với interface để trừu tượng database access:

```
interface UserDAO {
    User findById(String id);
    List<User> findAll();
    void insert(User user);
    void update(User user);
    void delete(String id);
}
```

Yêu cầu: Implement DAO với các database khác nhau (MySQL, MongoDB).

Input

```
<db_type>
N
<sql_operation>
```

Output

```
DAO <i>: <operation>
<query_và_kết_quả>
```

Ví dụ

Input:

```
MYSQL
4
INSERT U001 Alice
SELECT U001
UPDATE U001 name Bob
DELETE U001
```

Output:

```
DAO 1: INSERT U001
→ SQL: INSERT INTO users VALUES(...)
→ Đã thêm: Alice

DAO 2: SELECT U001
→ SQL: SELECT * FROM users WHERE id='U001'
→ Kết quả: Alice

DAO 3: UPDATE U001
→ SQL: UPDATE users SET name='Bob' WHERE id='U001'
→ Đã cập nhật

DAO 4: DELETE U001
→ SQL: DELETE FROM users WHERE id='U001'
→ Đã xóa
```

B22A - API Gateway

Độ khó: ★★★★★ Expert | **Loại:** Advanced

Đề bài

API Gateway với interface routing:

```
interface APIGateway {
    Response route(Request req);
}

interface ServiceEndpoint {
    Response handle(Request req);
}
```

Yêu cầu: Route requests đến các services tương ứng.

Input

```
N
<method> <path> <service>

M
<method> <path>
```

Output

```
Request <i>: <method> <path>
→ Route to: <service>
→ Response: <result>
```

Ví dụ

Input:

```
3
GET /users UserService
POST /orders OrderService
GET /products ProductService
4
GET /users
POST /orders
```

```
GET /products
GET /invalid
```

Output:

```
Request 1: GET /users
→ Route to: UserService
→ Response: 200 OK

Request 2: POST /orders
→ Route to: OrderService
→ Response: 201 Created

Request 3: GET /products
→ Route to: ProductService
→ Response: 200 OK

Request 4: GET /invalid
→ Route to: None
→ Response: 404 Not Found
```

B23A - Microservices Communication

Độ khó: ★★★★★ Expert | **Loại:** Advanced

Đề bài

Microservices với interface contracts:

```
interface ServiceClient {
    Response call(String endpoint, Request req);
}
```

Yêu cầu: Giao tiếp giữa các microservices với circuit breaker.

Input

```
N
<service> <endpoint>

M
CALL <service> <endpoint> <data>
```

Output

```
Call <i>: <service>.<endpoint>
<trạng_thái_circuit>
<kết_quả>
```

Ví dụ

Input:

```
3
UserService getUser
OrderService createOrder
PaymentService processPayment
5
CALL UserService getUser U001
CALL OrderService createOrder O001
CALL PaymentService processPayment P001
CALL PaymentService processPayment P002
CALL PaymentService processPayment P003
```

Output:

```
Call 1: UserService.getUser
→ Circuit: CLOSED
→ Response: User data

Call 2: OrderService.createOrder
→ Circuit: CLOSED
→ Response: Order created

Call 3: PaymentService.processPayment
→ Circuit: CLOSED
→ Response: FAILED (1/3)

Call 4: PaymentService.processPayment
→ Circuit: CLOSED
→ Response: FAILED (2/3)

Call 5: PaymentService.processPayment
→ Circuit: OPEN (quá nhiều lỗi)
→ Response: Service unavailable
```

B24A - Event-Driven Architecture

Độ khó: ★★★★★ Expert | **Loại:** Advanced

Đề bài

Event-driven system với interface:

```
interface Event {
    String getType();
    Object getData();
}

interface EventHandler {
    void handle(Event event);
}

interface EventBus {
    void publish(Event event);
    void subscribe(String type, EventHandler handler);
}
```

Yêu cầu: Xây dựng event bus với pub/sub pattern.

Input

```
N
SUBSCRIBE <handler_id> <event_type>

M
PUBLISH <event_type> <data>
```

Output

```
Event <i>: <type>
<handlers_nhận_event>
```

Ví dụ

Input:

```
3
SUBSCRIBE h1 ORDER_CREATED
```

```
SUBSCRIBE h2 ORDER_CREATED
SUBSCRIBE h3 PAYMENT_PROCESSED
4
PUBLISH ORDER_CREATED O001
PUBLISH PAYMENT_PROCESSED P001
PUBLISH USER_REGISTERED U001
PUBLISH ORDER_CREATED O002
```

Output:

```
Event 1: ORDER_CREATED
→ h1 nhận: O001
→ h2 nhận: O001

Event 2: PAYMENT_PROCESSED
→ h3 nhận: P001

Event 3: USER_REGISTERED
→ Không có handler

Event 4: ORDER_CREATED
→ h1 nhận: O002
→ h2 nhận: O002
```

B25A - E-Commerce Platform (CAPSTONE)

Độ khó: ★★★★★ Expert | **Loại:** CAPSTONE

Đề bài

CAPSTONE PROJECT - Hệ thống e-commerce hoàn chỉnh với interface-based architecture:

Interfaces:

- `ProductRepository`: Quản lý sản phẩm
- `CartService`: Giỏ hàng
- `OrderService`: Đơn hàng
- `PaymentGateway`: Thanh toán
- `ShippingProvider`: Vận chuyển
- `NotificationService`: Thông báo

Yêu cầu: Xây dựng luồng mua hàng hoàn chỉnh từ browse → checkout → delivery.

Input

```
N
<product_id> <tên> <giá> <kho>

M
BROWSE <category>
ADD_TO_CART <customer_id> <product_id> <số_lượng>
CHECKOUT <customer_id> <payment_method>
TRACK <order_id>
```

Output

```
Thao_tác <i>: <action>
<chi_tiết_xử_lý>
```

Ví dụ

Input:

```
3
P001 Laptop 15000000 10
P002 Mouse 200000 50
P003 Keyboard 500000 30
5
```

```
BROWSE Electronics
ADD_TO_CART C001 P001 1
ADD_TO_CART C001 P002 2
CHECKOUT C001 CREDIT_CARD
TRACK ORD001
```

Output:

```
Thao tác 1: BROWSE Electronics
→ ProductRepository.findByCategory()
→ Tìm thấy: 3 sản phẩm

Thao tác 2: ADD_TO_CART C001 P001
→ CartService.addItem()
→ Đã thêm: Laptop × 1

Thao tác 3: ADD_TO_CART C001 P002
→ CartService.addItem()
→ Đã thêm: Mouse × 2

Thao tác 4: CHECKOUT C001 CREDIT_CARD
→ OrderService.createOrder()
→ PaymentGateway.process(): 15,400,000đ
→ ShippingProvider.schedule()
→ NotificationService.send()
→ Đơn hàng: ORD001

Thao tác 5: TRACK ORD001
→ OrderService.getStatus()
→ Trạng thái: Đang xử lý
→ Dự kiến giao: 3-5 ngày
```
