IS216 - LẬP TRÌNH JAVA

Chương 2
LTHĐT TRONG JAVA
OOP IN JAVA

Giảng viên: Tạ Việt Phương
E-mail: phuongtv@uit.edu.vn

1

Nội dung

1. Giới thiệu
2. Lớp và đối tượng
3. Các tính chất: Đóng gói, Kế thừa, Đa hình, Trừu tượng
4. Gói, lớp trừu tượng và giao diện
5. Lamda Expressions và InnerClass
6. SOLID trong Java

2

1. GIỚI THIỆU

3

Lập trình hướng đối tượng

LÀ GÌ ?

4

Giới thiệu

- Object-Oriented Programming (OOP)
- **Lập trình hướng đối tượng** là mô hình (bao gồm cả thiết kế và phát triển) lập trình dựa trên kiến trúc **lớp** (class) và **đối tượng** (object). Nó giúp tổ chức chương trình bằng cách đóng gói dữ liệu và hành vi thành các thực thể có quan hệ với nhau.

5

Giới thiệu

- **Đặc điểm:**
  - Chương trình được chia thành các đối tượng.
  - Các cấu trúc dữ liệu được thiết kế sao cho đặc tả được đối tượng.
  - Các hàm thao tác trên các vùng dữ liệu của đối tượng được gắn với cấu trúc dữ liệu đó.
  - Dữ liệu được đóng gói lại, được che giấu và không cho phép các hàm ngoại lai truy nhập tự do.
  - Các đối tượng tác động và trao đổi thông tin với nhau qua các hàm
  - Có thể dễ dàng bổ sung dữ liệu và các hàm mới vào đối tượng nào đó khi cần thiết
  - Chương trình được thiết kế theo cách tiếp cận từ dưới lên (bottom-up)

6

Giới thiệu

- Phương pháp giải quyết 'top-down' (từ trên xuống) cũng còn được gọi là 'lập trình hướng cấu trúc' (structured programming). Nó xác định những chức năng chính của một chương trình và những chức năng này được phân thành những đơn vị nhỏ hơn cho đến mức độ thấp nhất.
- Phương pháp OOP giúp tổ chức chương trình theo mô hình đối tượng, trong đó dữ liệu và hành vi của đối tượng được đóng gói trong các lớp. Cho phép sử dụng lại code bằng cách kế thừa lớp cũ, tái sử dụng mã nguồn. Dễ dàng mở rộng hệ thống mà không ảnh hưởng đến các phần khác. Đóng gói dữ liệu giúp kiểm soát quyền truy cập và bảo vệ dữ liệu khỏi thay đổi không mong muốn.

7

7

Giới thiệu

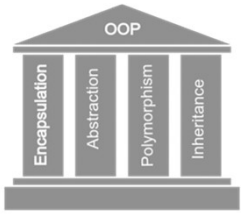
Phương pháp Top-Down	OOP
Chúng ta sẽ xây dựng một khách sạn.	Chúng ta sẽ xây dựng một tòa nhà 10 tầng với những dãy có các phòng trung bình, có các phòng sang trọng, và một phòng họp lớn.
Chúng ta sẽ thiết kế các tầng lầu, các phòng và phòng họp.	Chúng ta sẽ xây dựng một khách sạn với những thành phần trên.

8

8

Giới thiệu

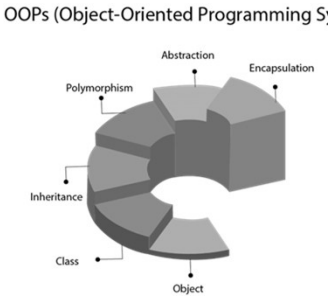
- 4 basic pillars (nguyên tắc trụ cột) of OOP
  - Abstraction: Trừu tượng hóa
  - Encapsulation: Đóng gói
  - Inheritance: Kế thừa
  - Polymorphism: Đa hình



9

9

Giới thiệu



10

10

Các khái niệm cơ bản

➢**Lớp (class):** Là khuôn mẫu (template), là bản mô tả những tính chất và hành vi chung của những sự vật, hiện tượng tồn tại trong thực tế.

Ví dụ: các lớp: Con người, Sinh viên, Lớp học, Môn học, Phòng học….

➢**Lớp** là cách phân loại *(classify)* các đối tượng dựa trên đặc điểm chung của các đối tượng đó.

11

11

Các khái niệm cơ bản

➢**Đối tượng (object):** trong thế giới thực khái niệm đối tượng có thể xem như một thực thể: người, vật, bảng dữ liệu,… Đối tượng trong thế giới thực: là một thực thể cụ thể mà ta có thể sờ, nhìn thấy hay cảm nhận được.

- Đối tượng giúp hiểu rõ thế giới thực
- Cơ sở cho việc cài đặt trên máy tính
- Mỗi đối tượng có định danh, thuộc tính, hành vi
- Ví dụ: đối tượng sinh viên

  MSSV: "17521091"; Tên sinh viên: "Đỗ Phú Quí"

12

12

Các khái niệm cơ bản

- **Đối tượng (object) của lớp:** một đối tượng cụ thể thuộc 1 lớp là 1 thể hiện cụ thể của 1 lớp đó. Một lớp có thể có nhiều thể hiện cụ thể (nhiều đối tượng)

Ví dụ: Sinh viên A, Sinh viên B là đối tượng của lớp Sinh viên.

- Quan hệ giữa Class và Object là quan hệ giữa mô tả (khuôn mẫu) và sự vật, hiện tượng cụ thể được sinh ra từ mô tả đó.

13

13

Các khái niệm cơ bản

- **Lớp (class):** là khuôn mẫu (*template*) để sinh ra đối tượng. Lớp là sự trừu tượng hóa của tập các đối tượng có các thuộc tính, hành vi tương tự nhau, và được gom chung lại thành 1 lớp.

Ví dụ: lớp các đối tượng **Sinhviên**

- Sinh viên "Nguyễn Văn A", mã số TH0701001 → 1 đối tượng thuộc lớp **Sinhviên**
- Sinh viên "Nguyễn Văn B", mã số TH0701002 → là 1 đối tượng thuộc lớp **Sinhviên**

14

14

Các khái niệm cơ bản

- **Hệ thống các đối tượng:** là 1 tập hợp các đối tượng
  - Mỗi đối tượng đảm trách 1 công việc
  - Các đối tượng có thể quan hệ với nhau
  - Các đối tượng có thể trao đổi thông tin với nhau
  - Các đối tượng có thể xử lý song song, hay phân tán

15

15

2. LỚP VÀ ĐỐI TƯỢNG

16

16

Lớp và đối tượng trong Java

- Lớp, Đối tượng
- Thuộc tính, Phương thức
- Constructor và Overloading Constructor

17

17

2.1. Lớp và đối tượng

- **Lớp** (Class): là một khuôn mẫu, là bản thiết kế (blueprint) để tạo ra các đối tượng. Nó định nghĩa các thuộc tính (dữ liệu) và phương thức (hành vi) mà đối tượng có thể có.
- Mỗi class có thể có các thành phần:
  - Thuộc tính.
  - Phương thức
  - Thành phần khác: Constructor (Hàm khởi tạo), Block khởi tạo, Nested class, Enum.
- **Đối tượng** (Object) là một thể hiện cụ thể của một lớp, với các giá trị thuộc tính và hành vi riêng biệt.

18

18

2.1. Lớp và đối tượng

- **Thuộc tính**
  - Một thuộc tính của một lớp là một trạng thái chung được đặt tên của tất cả các thể hiện của lớp đó có thể có.
  - Là các thông tin, trạng thái mà đối tượng của lớp đó có thể mang
  - Ví dụ: Lớp Ô tô có các thuộc tính
    - Màu sắc
    - Vận tốc
  - Bản chất của các thuộc tính là các thành phần dữ liệu của đối tượng, là các biến lưu trữ trạng thái của đối tượng.

19

19

2.1. Lớp và đối tượng

- **Thuộc tính**
  - Các thuộc tính của lớp cũng là các giá trị trừu tượng. Khi một đối tượng được tạo, đối tượng có bản sao các thuộc tính của riêng nó
  - **Ví dụ:** một chiếc Ô tô đang đi có thể có màu đen, vận tốc 60 km/h
  - Các thuộc tính phải được khai báo bên trong lớp
  - Được gọi là biến thành viên (Instance Variables) nếu không khai báo là static.

20

20

2.1. Lớp và đối tượng

- **Phương thức (Methods)**
  - Xác định các hoạt động chung mà tất cả các thể hiện của lớp có thể thực hiện được.
  - Xác định cách một đối tượng đáp ứng lại một thông điệp
  - Thông thường các phương thức sẽ hoạt động trên các thuộc tính và thường làm thay đổi các trạng thái của đối tượng.
  - Bất kỳ phương thức nào cũng phải thuộc về một lớp nào đó
  - **Ví dụ:** Lớp Ô tô có các phương thức
    - Tăng tốc
    - Giảm tốc

21

21

2.1. Lớp và đối tượng

- Class có các tính chất sau:
  - Đóng gói: chứa đựng dữ liệu và các phương thức liên quan
  - Che giấu dữ liệu: các thực thể phần mềm khác không can thiệp trực tiếp vào dữ liệu bên trong được mà phải thông qua các phương thức cho phép
  - Trừu tượng: Lớp chính là kết quả của quá trình trừu tượng hóa dữ liệu: Lớp định nghĩa một kiểu dữ liệu mới, trừu tượng hóa một tập các đối tượng

22

22

2.1. Lớp và đối tượng

➢**Khai báo lớp (class):**

```
class <ClassName>
{
    <danh sách thuộc tính>
    <các khởi tạo>
    <danh sách các phương thức>
}
```

23

23

2.1. Lớp và đối tượng

➢**Thuộc tính:** các đặc điểm mang giá trị của đối tượng, là vùng dữ liệu được khai báo bên trong lớp

```
class <ClassName> {
    <Phạm vi truy cập> <kiểu dữ liệu> <tên thuộc tính>;
}
```

<Phạm vi truy cập> (access modifier) Kiểm soát truy cập đối với thuộc tính

Còn gọi là **<tiền tố> hoặc <Các mức truy cập>**

Nếu thuộc tính không đi kèm với từ khóa static thì gọi là biến thành viên (Instance Variables)

24

24

2.1. Lớp và đối tượng

➢**Phương thức:** chức năng xử lý, hành vi của các đối tượng. Phương thức xác định các hoạt động của lớp

```
class <ClassName> {
    ...
    <Phạm vi truy cập> <kiểu trả về> <tên phương thức>(<các đối số>){
     ...
    }
}
```

25

25

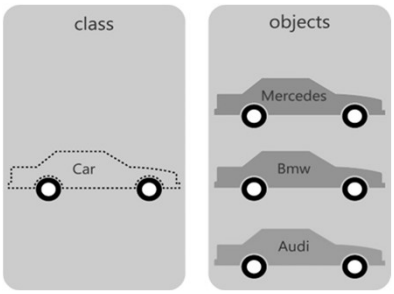
2.1. Lớp và đối tượng

- *<kiểu trả về>:* có thể là kiểu **void**, **kiểu cơ sở** hay **một lớp**.
- *<Tên phương thức>:* đặt theo quy ước giống tên biến.
- *<các đối số>:* có thể rỗng

26

26

2.1. Lớp và đối tượng



27

27

2.1. Lớp và đối tượng

Ví dụ 1

```
class Car {
        String name;
        int speed;

        void accelerate() {
                speed += 10;
        }
}
public class Main {
        public static void main(String[] args) {
                Car myCar = new Car();
                myCar.name = "Toyota";
                myCar.speed = 100;
                myCar.accelerate();
                System.out.println(myCar.speed); // Output: 110
                }
}
```

28

28

2.1. Lớp và đối tượng

Ví dụ 2

```
class Car {
    String brand;
    int speed;
    // Phương thức tăng tốc
    void accelerate() {
        speed += 10;
    }
    // Phương thức hiển thị thông tin xe
    void displayInfo() {
        System.out.println("Xe " + brand + " có tốc độ " + speed + " km/h.");
    }
}
```

29

29

2.1. Lớp và đối tượng

- Gọi phương thức từ một đối tượng:

Ví dụ 2

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.brand = “McLaren";
        myCar.speed = 332;

        myCar.accelerate();
        myCar.displayInfo();
    }
}
```

Xe McLaren có tốc độ 342 km/h.

30

30

2.1. Lớp và đối tượng

- Để tạo đối tượng, ta dùng từ khóa **new** kết hợp với một lớp
- Mỗi lần gọi new, một đối tượng mới được tạo trong bộ nhớ

```
ClassName objectName = new ClassName();
```

31

31

```
class Car {
    String brand;
    int speed;
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();  // Tạo đối tượng car1
        Car car2 = new Car();  // Tạo đối tượng car2

        car1.brand = "Toyota";
        car2.brand = "Ford";

        System.out.println(car1.brand);  // Output: Toyota
        System.out.println(car2.brand);  // Output: Ford
    }
}
```

32

32

2.2. Constructor

- **Hàm khởi tạo (hàm dựng hoặc constructor):** là một loại phương thức đặc biệt của lớp, dùng để khởi tạo giá trị ban đầu cho các thuộc tính của một đối tượng.
  - Dùng để khởi tạo giá trị cho các thuộc tính của đối tượng.
  - Cùng tên với lớp.
  - Không có giá trị trả về.
  - Tự động thi hành khi tạo ra đối tượng (new)
  - Có thể có tham số hoặc không.
- **Lưu ý:** Mỗi lớp sẽ có 1 constructor mặc định do Java tự động khai báo(nếu ta không khai báo constructor nào). Ngược lại nếu ta có khai báo 1 constructor khác thì constructor mặc định chỉ dùng được khi khai báo tường minh.

33

33

2.2. Constructor

Ví dụ 1

```
class Sinhvien
{
    …
    // Không có định nghĩa constructor nào
}
…
// Dùng constructor mặc định
Sinhvien sv = new Sinhvien();
```

34

34

2.2. Constructor

Ví dụ 2:

```
class Sinhvien{
   …
   // không có constructor mặc định
   // constructor có đối số
   Sinhvien(<các đối số>) {…}
}
…
Sinhvien sv = new Sinhvien();
// lỗi biên dịch
```

```
class Sinhvien{
   …
   // khai báo constructor mặc định
   Sinhvien(){}
   // constructor có đối số
   Sinhvien(<các đối số>) {…}
}
…
Sinhvien sv = new Sinhvien();
//Hoặc
Sinhvien sv = new Sinhvien (<các đối số>);
```

35

35

2.2. Constructor

```
package constructor;
class SinhVien {
private String Ten;
   public void In()
   {
     System.out.println("Ten:"+Ten);
   }
}
```

```
package constructor;
public class Constructor {
public static void main(String[] args) {
      SinhVien s= new SinhVien();
      s.In();
   }
}
```

```
Ten:null
```

36

36

2.2. Constructor

```
package constructor;
class SinhVien {
private String Ten;
public SinhVien(){
  Ten="Dang My Hang";
}
public void In()
{
  System.out.println("Ten:"+Ten);
}
}
```

```
package constructor;
public class Constructor {
public static void main(String[] args) {
      SinhVien s= new SinhVien();
      s.In();
   }
}
```

```
Ten:Dang My Hang
```

37

37

2.2. Constructor

```
package constructor;
class SinhVien {
   private String MSSV;
   private String Ten;
   public SinhVien(){
      Ten="Ly Trung Binh";
   }
    public SinhVien(String str){
      Ten=str;
   }
   public void In(){
      System.out.println("Ten:"+Ten);
   }
}
```

```
package constructor;
public class Constructor {
public static void main(String[] args) {
   SinhVien s= new SinhVien("Lieu Nhu Yen ");
   s.In();
   }
}
```

```
Ten: Lieu Nhu Yen
```

38

38

2.3. Nạp chồng Constructor

- Nạp chồng Constructor cho phép một lớp có nhiều constructor với danh sách tham số khác nhau.
- Java chọn constructor phù hợp dựa vào số lượng và kiểu tham số.
- Dùng Overloading Constructor giúp linh hoạt hơn khi khởi tạo đối tượng

39

39

2.3. Nạp chồng Constructor

```
class Car {
   String brand;
   int speed;

   // Constructor mặc định
   Car() {
      this.brand = "Unknown";
      this.speed = 0;
   }

   // Constructor với 1 tham số
   Car(String brand) {
      this.brand = brand;
      this.speed = 0;
   }
```

```
   // Constructor với 2 tham số
   Car(String brand, int speed) {
      this.brand = brand;
      this.speed = speed;
   }

   void displayInfo() {
      System.out.println("Xe " + brand + " có tốc
độ " + speed + " km/h.");
   }
}
```

40

40

2.3. Nạp chồng Constructor

```
public class Main {
   public static void main(String[] args) {
      Car car1 = new Car();  // Gọi constructor mặc định
      Car car2 = new Car("Toyota");  // Gọi constructor với 1 tham số
      Car car3 = new Car("Porsche", 240);  // Gọi constructor với 2 tham số

      car1.displayInfo();
      car2.displayInfo();
      car3.displayInfo();
   }
}
```

```
Xe Unknown có tốc độ 0 km/h.
Xe Toyota có tốc độ 0 km/h.
Xe Honda có tốc độ 120 km/h.
```

41

41

2.4. Các mức truy cập trong Java

➢Có 2 loại là Access Modifier và Non-access Modifier

- **Access Modifer** trong Java xác định phạm vi có thể truy cập của biến, phương thức, constructor hoặc lớp. Trong java, có 4 phạm vi truy cập của Access Modifier như sau:
  1. private
  2. default
  3. protected
  4. public
- Ngoài ra, còn có nhiều **Non-access Modifier** như static, abstract, synchronized, native, volatile, transient,... không ảnh hưởng đến phạm vi truy cập nhưng thay đổi **hành vi** của biến, phương thức, lớp.

42

42

2.4. Các mức truy cập trong Java

➢Để bảo vệ dữ liệu tránh bị truy nhập tự do từ bên ngoài, Java sử dụng các từ khoá quy định phạm vi truy nhập các thuộc tính và phương thức của lớp:

- **public:** Thành phần công khai, truy nhập tự do từ bên ngoài.
- **protected**: Thành phần được bảo vệ, được hạn chế truy nhập.
- **default** (không viết gì): Truy nhập trong nội bộ gói.
- **private**: Truy nhập trong nội bộ lớp.

43

43

2.4. Các mức truy cập trong Java

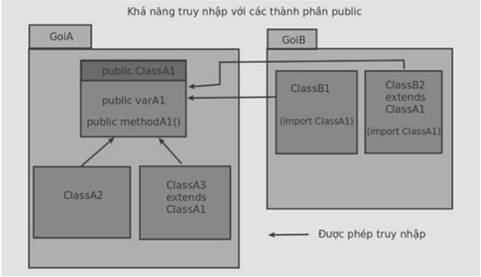
- Public: thường dùng cho các **API hoặc phương thức chung** có thể gọi từ bất kỳ đâu. **Có thể truy cập từ mọi nơi**, không có giới hạn về package.

```
public class Example {
    public int data = 10;

    public void display() {
        System.out.println("Giá trị: " + data);
    }
}
```

44

44

2.4. Các mức truy cập trong Java



45

45

2.4. Các mức truy cập trong Java

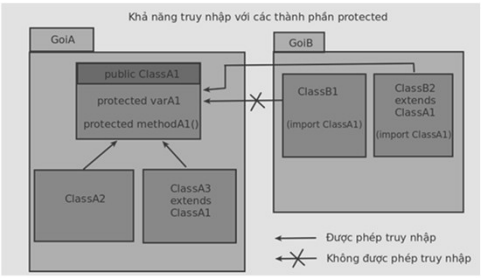
- Protected (Bảo vệ): Có thể truy cập từ **lớp con trong package khác** (thông qua kế thừa). Có thể truy cập **trong cùng package**. thường dùng khi muốn **cho phép kế thừa nhưng hạn chế truy cập từ ngoài package**.

```
class Parent {
    protected int data = 20;

    protected void display() {
        System.out.println("Giá trị: " + data);
    }
}
class Child extends Parent {
    void show() {
        display(); // Có thể truy cập protected từ lớp cha
    }
}
```

46

46

2.4. Các mức truy cập trong Java



47

47

2.4. Các mức truy cập trong Java

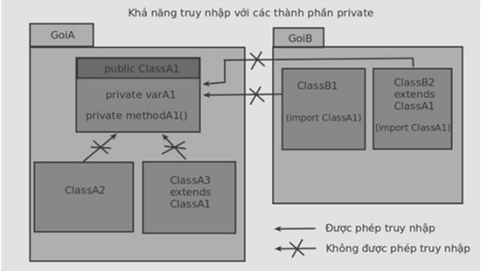
- Private (Riêng tư): Chỉ có thể truy cập **bên trong cùng một lớp.** Không thể truy cập từ lớp khác, kể cả lớp con.

```
class Example {
    private int data = 100;

    private void display() {
        System.out.println("Giá trị: " + data);
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example();
        // obj.data = 200; // Lỗi: Không thể truy cập private
        // obj.display(); // Lỗi: Không thể truy cập private
    }
}
```

48

48

2.4. Các mức truy cập trong Java



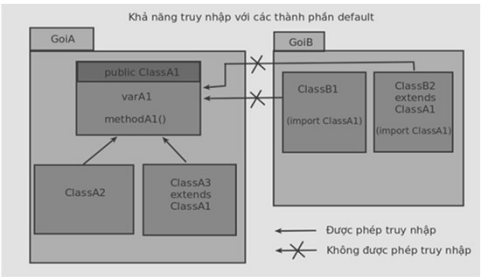
49

2.4. Các mức truy cập trong Java

- Default: Nếu không khai báo Access Modifier, phạm vi mặc định là **default**. Chỉ có thể truy cập **trong cùng package**, không thể truy cập từ bên ngoài package (các package khác).

50

2.4. Các mức truy cập trong Java



51

2.4. Các mức truy cập trong Java

- Đặc tính truy xuất của 4 loại modifier được thể hiện như sau:

	Private	Default	Protected	Public
Cùng class	**Yes**	**Yes**	**Yes**	**Yes**
Cùng package, khác class	No	**Yes**	**Yes**	**Yes**
Class con trong cùng package với class cha	No	Yes	**Yes**	**Yes**
Khác package, khác class	No	No	No	**Yes**
Class con khác package với class cha	No	No	**Yes**	**Yes**

- **Lưu ý**: Đối với class thì chỉ có 2 Access modifier đó là public và default.

52

2.4. Các mức truy cập trong Java

```
class A {
    private int privateVar = 1;
    int defaultVar = 2;
    protected int protectedVar = 3;
    public int publicVar = 4;
}

class B extends A {
    void show() {
                    // privateVar không thể truy cập
        System.out.println(defaultVar);   // Truy cập trong cùng package
        System.out.println(protectedVar); // Truy cập trong subclass
        System.out.println(publicVar);    // Truy cập mọi nơi
    }
}
```


53

2.5. Getter và Setter

- **Getter:** Phương thức lấy giá trị của biến thành viên. Còn gọi là **Accessor Methods**
- **Setter:** Phương thức thiết lập giá trị, có thể thêm điều kiện kiểm tra. Còn gọi là **Mutator Methods**
- Mục đích:
  - Kiểm soát truy cập dữ liệu: Hạn chế việc thay đổi giá trị không hợp lệ.
  - Đảm bảo nguyên tắc đóng gói: Không cho phép truy cập trực tiếp vào biến thành viên.
  - Dễ dàng mở rộng: Nếu cần thay đổi logic xử lý, chỉ cần sửa trong getter/setter.

54

2.5. Getter và Setter

- **Getter:** Thường bắt đầu với get + tên thuộc tính (ví dụ: getName(), getAge()). Giúp truy xuất dữ liệu **mà không làm thay đổi giá trị**.
- **Setter:** Thường bắt đầu với set + tên thuộc tính (ví dụ: setName(), setSpeed()). Có thể thêm kiểm tra điều kiện để đảm bảo dữ liệu hợp lệ. Giúp thay đổi giá trị, **có thể kiểm tra hợp lệ trước khi gán**

55

55

2.5. Getter và Setter

```
class Car {
    private String brand;
    private int speed;

    // Constructor
    public Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }

    // Accessor Method (Getter)
    public String getBrand() {
        return brand;
    }

    public int getSpeed() {
        return speed;
    }
```

```
    // Mutator Method (Setter)
    public void setBrand(String brand) {
        this.brand = brand;
    }

    public void setSpeed(int speed) {
        if (speed > 0) {
            this.speed = speed;
        } else {
            System.out.println("Tốc độ phải lớn hơn 0!");
        }
    }

    public void display() {
        System.out.println("Xe: " + brand + ", Tốc độ: " + speed + "
km/h");
    }
}
```

56

56

2.5. Getter và Setter

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

        // Sử dụng Accessor (Getter)
        System.out.println("Thương hiệu: " + myCar.getBrand());

        // Sử dụng Mutator (Setter)
        myCar.setSpeed(150);
        myCar.setBrand("Honda");

        myCar.display();
    }
}
```

```
Thương hiệu: Toyota
Xe: Honda, Tốc độ: 150 km/h
```

57

57

2.5. Getter và Setter

- Tối giản Getter/Setter với record (Java 14+): Java 14+ giới thiệu record, giúp tự động tạo getter mà không cần viết thủ công.
- Record là immutable (không thể thay đổi giá trị sau khi khởi tạo)

58

58

2.5. Getter và Setter

```
public record Car(String brand, int speed) {}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 120);

        // Java tự động tạo getter (không cần viết thủ công)
        System.out.println(myCar.brand()); // Toyota
        System.out.println(myCar.speed()); // 120
    }
}
```

- Tự động tạo getter (brand() và speed()).
- Không cần setter vì record là immutable.
- Không thể thay đổi giá trị sau khi khởi tạo (không có setBrand() hoặc setSpeed()).

59

59

2.6. Ví dụ

Ví dụ 1: class Sinhvien {

// Danh sách thuộc tính
String maSv, tenSv, dcLienlac;
int tuoi;
…
// Danh sách các khởi tạo
Sinhvien(){}
Sinhvien (…) { …}
…
// Danh sách các phương thức
public void capnhatSV (…) {…}
public void xemThongTinSV() {…}
…
}

60

60

2.6. Ví dụ

```
...
// Tạo đối tượng mới thuộc lớp Sinhvien
Sinhvien sv = new Sinhvien();
...
// Gán giá trị cho thuộc tính của đối tượng
sv.maSv = “23521113” ;
sv.tenSv = “Ngo Thua An”;
sv.tuoi = 20;
sv.dcLienlac = “KP6, Linh Trung, Thu Duc”;
...
// Gọi thực hiện phương thức
sv.xemThongTinSV();
```

61

61

2.6. Ví dụ

Ví dụ 2: class Sinhvien {

```
    // Danh sách thuộc tính
    private String maSv;  String tenSv, dcLienlac;  int
    tuoi;
    ...
}
…
Sinhvien sv = new Sinhvien();
```

sv.maSv = “23521114”; // Lỗi truy cập **thuộc tính private** từ bên ngoài lớp khai báo

```
Sv.tenSv = “Hong Hai Nhi”;
```

62

62

2.6. Ví dụ

```
class Sinhvien {
   private String maSv, tenSv, dcLienlac;
   int     tuoi;
...
   // Mutator Methods
   public void setmaSV(String maSv){
          this.maSv = maSv;
   }
  // Accessor Methods
  public String getmaSV(String (){
          return maSv;
  }
```

63

63

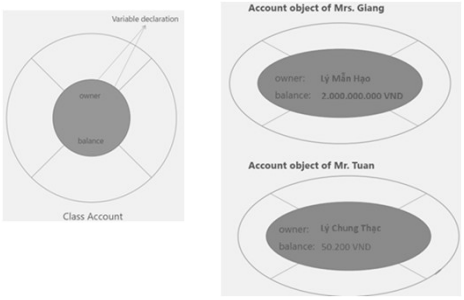
2.6. Ví dụ

```
public class Account {
// instance variable
String owner; // Account name
long balance; // Balance
//...
// value setting method
void setAccountInfo(String owner, long balance) {
this.owner = owner; this.balance = balance;
} //...
}
```

64

64

2.6. Ví dụ



65

65

2.6. Ví dụ

- Truy cập đến các thuộc tính

```
public class Account {
          String name; //Account name
           long balance; //Balance
          void display(){ System.out.println(…);
          }
          void deposit (long money){ balance += money;
          }
}
Account acc1 = new Account();
acc1.name = “Lý Mẫn Hạo”; acc1.balance = “2000000000”;
Account acc2 = new Account();
acc2.name = “Lý Chung Thạc”; acc2.balance = “50200”;
```

66

66

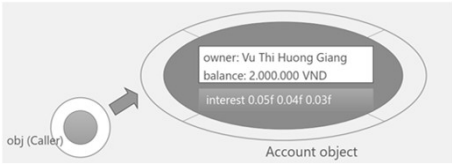
2.6. Ví dụ

- Truy cập đến các phương thức

```
public class Account {
        String name; //Account name
         long balance; //Balance
        void display(){ System.out.println(…);
}
        void deposit (long money){ balance += money;
} }
// Class that uses  methods of Account object
Account obj = new Account();
obj.display();
obj.deposit(1000000);
```


67

2.6. Ví dụ



68

Bài tập

Vào trang

https://introcs.cs.princeton.edu/java/30oop/

Chọn 1 bài. Đọc và giải thích ý nghĩa các dòng code cũng như chức năng của các lớp.

69

3. ĐÓNG GÓI, KẾ THỪA, ĐA HÌNH, TRỪU TƯỢNG

70

Đặc điểm hướng đối tượng trong java

1. Tính đóng gói (Encapsulation)
2. Tính kế thừa (Inheritance)
3. Tính đa hình (Polymorphism)
4. Tính trừu tượng (Abstract)

71

3.1. Tính đóng gói

- **Encapsulation**
- **Đóng gói:**
  - Nhóm những gì có liên quan với nhau vào thành một và có thể sử dụng một cái tên để gọi.
  - Dùng để che giấu một phần hoặc tất cả thông tin, chi tiết cài đặt bên trong với bên ngoài, bảo vệ dữ liệu bằng cách giới hạn quyền truy cập.
  - Dữ liệu chỉ có thể được truy cập thông qua các phương thức được cấp quyền.
  - Giúp thay đổi nội bộ mà không ảnh hưởng đến mã nguồn bên ngoài.

72

3.1. Tính đóng gói

```
class BankAccount {
   private double balance;

   public BankAccount(double balance) {
      this.balance = balance;
   }

   public void deposit(double amount) {
      balance += amount;
   }

   public double getBalance() {
      return balance;
   }
}
```

- Biến balance là private, ngăn không cho truy cập trực tiếp từ bên ngoài.
- Phương thức getBalance() là public, cho phép lấy thông tin số dư an toàn.

3.1. Tính đóng gói

- **Tính đóng gói thể hiện ở:**
  - Các phương thức đóng gói các câu lệnh.
  - Đối tượng đóng gói dữ liệu và các hành vi / Phương thức liên quan.
  - Các mức độ truy cập.
  - Biến thành viên (Instance Variables) thường được đặt là private để ngăn truy cập trực tiếp từ bên ngoài.
  - Getter-Setter
  - …

```
class Car {
   private String brand;  // Biến thành viên (thuộc tính)
   private int speed;     // Biến thành viên
}
```


3.1.1. Packages

- Những phần của một chương trình Java:
  - Lệnh khai báo gói(**package)**
  - Lệnh chỉ định gói được dùng (Lệnh i**mport)**
  - Khai báo lớp public (một file java chỉ chứa 1 lớp public class)
  - Các lớp khác (classes private to the package)
- Tập tin nguồn Java có thể chứa tất cả hoặc một vài trong số các phần trên.

3.1.1. Packages

- Gói (Packages) có thể xem như là thư mục lưu trữ những lớp, interface và các gói con khác. Đó là những thành phần của gói.
- Là một cơ chế nhóm các loại lớp, giao diện và các lớp con tương tự nhau dựa trên chức năng.
- Package trong Java là cách tổ chức mã nguồn theo thư mục để nhóm các lớp liên quan. Giúp quản lý mã nguồn hiệu quả, tránh xung đột tên lớp.

3.1.1. Packages

- Những ưu điểm khi dùng gói:
  - Cho phép tổ chức các lớp vào những đơn vị nhỏ hơn
  - Khả năng sử dụng lại: Các lớp có trong các gói package của chương trình khác có thể dễ dàng sử dụng lại
  - Giúp tránh được tình trạng trùng lặp khi đặt tên.
  - Cho phép bảo vệ các lớp đối tượng
  - Tên gói (Package) có thể được dùng để nhận dạng chức năng của các lớp.
  - Đóng gói dữ liệu : Chúng cung cấp một cách để ẩn các lớp, ngăn các chương trình khác truy cập các lớp chỉ dành cho sử dụng nội bộ

3.1.1. Packages

- Những lưu ý khi tạo gói:
  - Mã nguồn phải bắt đầu bằng lệnh ‘package’
  - Mã nguồn phải nằm trong cùng thư mục mang tên của gói
  - Tên gói nên bắt đầu bằng ký tự thường (lower case) để phân biệt giữa lớp đối tượng và gói
  - Những lệnh khác phải viết phía dưới dòng khai báo gói là mệnh đề **import**, kế đến là các mệnh đề định nghĩa lớp đối tượng
  - Những lớp đối tượng trong gói cần phải được biên dịch
  - Để chương trình Java có thể sử dụng những gói này, ta phải **import** gói vào trong mã nguồn

3.1.1. Packages

- Các gói package được chia thành hai loại:
  - Gói package được xây dựng sẵn (buit-in)
  - Gói package do người dùng xác định (defined)
- Import gói (Importing packages):
  - Xác định tập tin cần được import trong gói
  - Hoặc có thể import toàn bộ gói

79

79

3.1.1. Packages

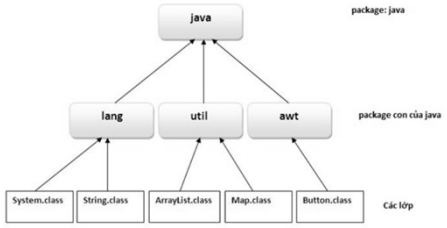
- **Gói package được xây dựng sẵn**
  - Được xác định trước là các gói package đi kèm như một phần của JDK để đơn giản hóa nhiệm vụ của lập trình viên Java.
  - Một số gói package tích hợp thường được sử dụng là **java.lang, java.io,java.util, java.applet**…

80

80

3.1.1. Packages

- **Gói package được xây dựng sẵn: Ví dụ**



81

81

3.1.1. Packages

```
import java.util.ArrayList;
public class BuiltInPackage {
        public static void main(String[] args){
                ArrayList<Integer> myList = new
        ArrayList<>(3);
                myList.add(3);
                myList.add(2);
                myList.add(1);
                System.out.println("Cac thanh phan cua
danh sach la: " + myList);
        }
}
```

82

82

3.1.1. Packages

- Một số gói (Package) tích hợp thường dùng trong Java:
  - java.lang (Mặc định - Không cần import): Chứa các lớp lõi của Java, như String, Math, System, Object.
  - java.util (Cấu trúc dữ liệu & tiện ích): Chứa các lớp làm việc với danh sách, tập hợp, ngày giờ, Scanner...
  - java.io (Nhập/Xuất dữ liệu): Chứa các lớp để làm việc với file, input/output stream: PrintWriter, BufferedReader, File.
  - java.sql (Làm việc với cơ sở dữ liệu)
  - javax.swing (Xây dựng giao diện đồ họa)

83

83

3.1.1. Packages

- Khai báo gói

  package MyPackage;
- Import những gói chuẩn (xây dựng sẵn) cần thiết
- Khai báo và định nghĩa các lớp đối tượng có trong gói
- Lưu các định nghĩa trên thành tập tin .**java**, và biên dịch những lớp đối tượng đã được định nghĩa trong gói.

```
package mypackage;
public class Simple {
        public static void main(String args[]) {
                System.out.println("Learn java package");
        }
}
```

84

84

3.1.2. Sử dụng những gói do người dùng định nghĩa (user-defined packages)

- Mã nguồn của những chương trình này phải ở cùng thư mục của gói do người dùng định nghĩa.
- Để những chương trình Java khác sử dụng những gói này, import gói vào trong mã nguồn

85

85

3.1.2. Sử dụng những gói do người dùng định nghĩa (user-defined packages)

```
package pack;
public class A {
   public void msg() {
      System.out.println("Hello");
   }
}
```

Sử dụng packagename.*

```
package mypack;
import pack.*;
class B {
   public static void main(String args[]) {
      A obj = new A();
      obj.msg();
   }
}
```

Hello

86

86

3.1.2. Sử dụng những gói do người dùng định nghĩa (user-defined packages)

```
package pack;
public class A {
   public void msg() {
      System.out.println("Hello mấy ní");
   }
}
```

Sử dụng packagename.classname

```
package mypack;
import pack.A;
class B {
   public static void main(String args[]) {
      A obj = new A();
      obj.msg();
   }
}
```

Hello mấy ní

87

87

3.1.2. Sử dụng những gói do người dùng định nghĩa (user-defined packages)

```
package pack;
public class A {
   public void msg() {
      System.out.println("Hello");
   }
}
```

Sử dụng tên đầy đủ

```
package mypack;
class B {
   public static void main(String args[]) {
      pack.A obj = new pack.A();
      obj.msg();
   }
}
```

Hello

88

88

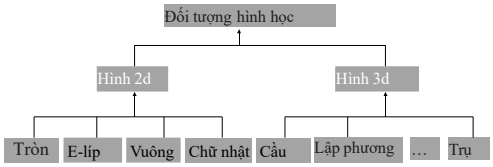
3.2. Tính kế thừa

- **Inheritance**
- Bản chất **kế thừa (inheritance)** là phát triển lớp mới dựa trên các lớp đã có. Xây dựng các lớp mới có sẵn các đặc tính của lớp cũ, đồng thời lớp mới có thể mở rộng các đặc tính của nó.
- **Lớp kế thừa** gọi là lớp con (child, subclass), lớp dẫn xuất (derived class). Lớp được kế thừa gọi là lớp cha (parent, superclass), lớp cơ sở (base class).
- **Mối quan hệ kế thừa:** Lớp con là một loại (is-a-kind-of) của lớp cha, kế thừa các thành phần dữ liệu và các hành vi của lớp cha. Có thể khai báo thêm thuộc tính, phương thức cho phù hợp với mục đích sử dụng mới.

89

89

3.2. Tính kế thừa



- Thừa hưởng các thuộc tính và phương thức đã có
- Bổ sung, chi tiết hóa cho phù hợp với mục đích sử dụng mới
  - ✓ **Thuộc tính**: thêm mới
  - ✓ **Phương thức**: thêm mới hay hiệu chỉnh

90

90

3.2. Tính kế thừa

- Ứng dụng của Kế thừa
  - Tái sử dụng mã nguồn: Giúp giảm lặp lại code, tiết kiệm thời gian phát triển.
  - Tổ chức mã nguồn tốt hơn: Xây dựng quan hệ cha - con giữa các lớp.
  - Mở rộng chức năng: Lớp con có thể bổ sung hoặc thay đổi phương thức của lớp cha.
  - Hỗ trợ đa hình (Polymorphism): Giúp dễ dàng thay đổi hành vi của đối tượng.

3.2. Tính kế thừa

- **Lớp dẫn xuất** hay **lớp con (SubClass)**
- **Lớp cơ sở** hay **lớp cha (SuperClass)**
- Lớp con có thể kế thừa tất cả hay một phần các thành phần dữ liệu (thuộc tính), phương thức của lớp cha (public, protected, default)
- Dùng từ khóa ***extends.***

Ví dụ:

```
class nguoi { …
}
class sinhvien extends nguoi { …
}
```


3.2.1. super và this

- **Super:**
  - Dùng để gọi constructor hoặc phương thức của lớp cha
  - Tránh bị ghi đè (Override), giúp gọi lại phương thức gốc của lớp cha
  - Nếu gọi tường minh thì phải là câu lệnh đầu tiên

3.2.1. super và this

```
class Nguoi {
   public Nguoi(String ten, int tuoi){
        …
   }
}
class SinhVien extends nguoi {
   public void show(){
          System.out.println("....");
          super('A',20);
    }
}
```

Lỗi do super phải là câu lệnh đầu tiên

3.2.1. super và this

- Tham chiếu **this**: là một biến ẩn tồn tại trong tất cả các lớp, this được sử dụng trong khi chạy và tham khảo đến bản thân lớp chứa nó.
- Cho phép truy cập vào đối tượng hiện tại của lớp
- Không dùng bên trong các khối lệnh static

3.2.1. super và this

```
// Lớp cha Car
class Car {
    String brand;

    // Constructor của lớp cha
    Car(String brand) {
        this.brand = brand;
    }

    // Phương thức startEngine
    void startEngine() {
        System.out.println(brand + " đang khởi động động cơ...");
    }
}
```

3.2.1. super và this

```
// Lớp con ElectricCar kế thừa từ Car
class ElectricCar extends Car {
    int batteryCapacity;
    // Constructor của lớp con
    ElectricCar(String brand, int batteryCapacity) {
        super(brand);  // Gọi constructor của lớp cha
        this.batteryCapacity = batteryCapacity;
    }

    // Ghi đè phương thức startEngine
    @Override
    void startEngine() {
        super.startEngine();  // Gọi phương thức gốc của lớp cha
        System.out.println(brand + " là xe điện, khởi động êm ái...");
    }
}
```

97

97

3.2.1. super và this

```
public class Main {
    public static void main(String[] args) {
        ElectricCar myCar = new ElectricCar("BYD Seal", 82);
        myCar.startEngine();
    }
}
```

```
BYD Seal đang khởi động động cơ...
BYD Seal là xe điện, khởi động êm ái...
```

98

98

3.2.2. Constructor Inheritance

- Sự thừa kế trong hàm khởi tạo
  - Khai báo về thừa kế trong hàm khởi tạo
  - Chuỗi các hàm khởi tạo (Constructor Chaining)
  - Các nguyên tắc của hàm khởi tạo (Rules)
  - Triệu hồi tường minh hàm khởi tạo của lớp cha

99

99

3.2.2. Constructor Inheritance

- Trong Java, hàm khởi tạo **không thể thừa kế** từ lớp cha như các loại phương thức khác.
- Khi tạo một thể hiện của lớp dẫn xuất, trước hết phải gọi đến hàm khởi tạo của lớp cha, tiếp đó mới là hàm khởi tạo của lớp con.
- Có thể triệu hồi hàm khởi tạo của lớp cha bằng cách sử dụng từ khóa **super** trong phần khai báo hàm khởi tạo của lớp con.

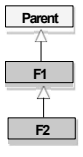
100

100

3.2.2. Constructor Inheritance

- **Constructor Chaining** - Chuỗi hàm khởi tạo

```
class Parent{
    public Parent(){
        System.out.println("This is constructor of Parent class");
    }
}
class F1 extends Parent{
    public F1() {
        System.out.println("This is constructor of F1 class");
    }
}
class F2 extends F1{
    public F2(){
        System.out.println("This is constructor of F2 class");
    }
}
```



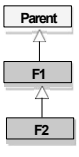
?

101

101

3.2.2. Constructor Inheritance

```
class Parent{
    public Parent(){
        System.out.println("This is constructor of Parent class");
    }
}
class F1 extends Parent{
    public F1() {
        super(); // Gọi constructor của Parent
        System.out.println("This is constructor of F1 class");
    }
}
class F2 extends F1{
    public F2(){
        super(); // Gọi constructor của F1
        System.out.println("This is constructor of F2 class");
    }
}
```



102

102

3.2.2. Constructor Inheritance

```
public class Test {
    public static void main(String[] args) {
        F2 obj = new F2(); // Gọi constructor của F2
    }
}
```

```
This is constructor of Parent class
This is constructor of F1 class
This is constructor of F2 class
```

1. Object
2. Parent() call super()
3. F1() call super ()
4. F2() call super()
5. Main() call new F2()

Khi tạo một thể hiện của lớp dẫn xuất, trước hết phải gọi đến hàm khởi tạo của lớp cha, tiếp đó là hàm khởi tạo của lớp con.

103

103

3.2.2. Constructor Inheritance

- Hàm khởi tạo mặc nhiên (default constructor) sẽ tự động sinh ra bởi trình biên dịch nếu lớp không khai báo hàm khởi tạo.
- Hàm khởi tạo mặc nhiên luôn luôn không có tham số (no-arg)
- Nếu trong lớp có định nghĩa hàm khởi tạo, hàm khởi tạo mặc nhiên sẽ không còn được sử dụng.
- Nếu không có lời gọi tường minh đến hàm khởi tạo của lớp cha tại lớp con, trình biên dịch sẽ tự động chèn lời gọi tới hàm khởi tạo mặc nhiên (implicity) hoặc hàm khởi tạo không tham số (explicity) của lớp cha trước khi thực thi đoạn code khác trong hàm khởi tạo lớp con.

104

104

3.2.2. Constructor Inheritance

- **Có 1 vấn đề?**

```
public class Parent
{
    private int a;

    public Parent(int value)
    {
        a = value;
        System.out.println("Invoke parent parameter constructor");
    }
}
public class F1 extends Parent
{
    public F1()
    {
        System.out.println("Invoke F1 default constructor");
    }
}
```

Lỗi: Không tìm hàm khởi tạo của lớp Parent không có tham số

105

105

3.2.2. Constructor Inheritance

- Sửa như thế nào?

Sửa: Thêm hàm khởi tạo lớp Parent

```
public class Parent
{
    private int a;

    public Parent()
    {
        a = 0;
        System.out.println("Invoke parent default constructor");
    }

    public Parent(int value)
    {
        a = value;
        System.out.println("Invoke parent parameter constructor");
    }
}
public class F1 extends Parent
{
    public F1()
    {
        System.out.println("Invoke F1 default constructor");
    }
}
```

106

106

3.2.2. Constructor Inheritance

- Các nguyên tắc của hàm khởi tạo: Triệu hồi tường minh hàm khởi tạo lớp cha

```
public class Parent
{
    private int a;

    public Parent(int value)
    {
        a = value;
        System.out.println("Invoke parent parameter constructor");
    }
}
public class F1 extends Parent
{
    public F1(int value)
    {
        super (value);
        System.out.println("Invoke F1 default constructor");
    }
}
```

107

107

3.2.2. Constructor Inheritance

- Không gọi tường minh hàm khởi tạo lớp cha ở lớp con

```
public class Parent
{
    private int a;

    public Parent()
    {
        a = 0;
        System.out.println("Invoke parent default constructor");
    }

    public Parent(int value)
    {
        a = value;
        System.out.println("Invoke parent parameter constructor");
    }
}
public class F1 extends Parent
{
    public F1()
    {
        System.out.println("Invoke F1 default constructor");
    }
}
```

108

108

3.2.2. Constructor Inheritance

```
public static void main(String[] args) { F1
    f1=new F1();
  }
```

```
Kết quả:
Invoke parent default constructor
Invoke F1default constructor
```

109

109

3.2.3. Ví dụ

```
class Person {
    private String CMND;
    private String Name;
    private int age;
    public Person(String cm, String na, int a){
            CMND=cm;
            Name=na;
            age=a;
     }
    public void Print(){
        System.out.println("Chung minh"+"\t"+"Tên"+"\t"+"Tuoi");
        System.out.print(CMND+"\t"+Name+"\t"+age);
    }
}
```

110

110

3.2.3. Ví dụ

```
class Employee extends Person {
    private double salary;
    public Employee(String cm, String na, int a, double sa){
        super(cm,na,a);
        salary=sa;
    }
    public void Print(){
        super.Print();
        System.out.print("Luong thang:"+salary);
    }
}
```

111

111

3.2.3. Ví dụ

```
class Maneger extends Employee {
    private double allowance;
    public Maneger(String cm,String na,int a,double sa,double allow){
        super(cm,na,a,sa);
        allowance=allow;
    }
    public void Print(){
        super.Print();
        System.out.print("Phu cap:"+allowance);
    }
}
```

112

112

3.2.3. Ví dụ

```
class Maneger extends Employee {
    private double allowance;
    public Maneger(String cm, String na, int a, double sa, double
        allow){ super(cm,na,a,sa);
        allowance=allow;
    }
    public void
        Print(){
        super.Print();
        System.out.print("Phu cap:"+allowance);
    }
}
```

113

113

3.2.3. Ví dụ

```
public class Nhanvien {
    public static void main(String[] args) {
        Person p=new Person("1234", "NGuyen Huu Dat",23);
        Employee e=new Employee("2345","Tran Ngoc Tuan", 24,10000000);
        Maneger mng= new Maneger("3456", "Lê Văn
        Toàn",25,10000000,2000000);
        System.out.println("Thong tin nguoi:");
        p.Print();
        System.out.println();
        System.out.println("Thong nhan vien:");
        e.Print();
        System.out.println();
        System.out.print("Thong tin quan ly");
        mng.Print();
    }
}
```

114

114

3.2.3. Ví dụ

```
Thong tin nguoi:
Chung minh  Tên Tuoi
1234      Nguyen Huu Dat 23 Thong nhan
vien:
Chung minh  Tên Tuoi
2345      Tran Ngoc Tuan   24Luong thang:1.0E7 Thong tin
quan lyChung minh           Tên Tuoi
3456      Lê Văn Toàn 25Luong thang:1.0E7Phu cap:2000000.0
```

115

115

3.2.4. Từ khóa Final

- Từ khóa final trong Java chủ yếu thể hiện tính đóng gói (Encapsulation) trong lập trình hướng đối tượng (OOP). Ngoài ra, nó cũng có thể hỗ trợ tính kế thừa (Inheritance) bằng cách ngăn chặn việc ghi đè (Overriding) hoặc kế thừa lớp.
  - Biến Final - Final Variables
  - Phương thức Final - Final Methods
  - Lớp Final - Final Classes

116

116

3.2.4. Từ khóa Final – biến Final

- Từ khóa "final" được sử dụng với biến để chỉ rằng giá trị của biến là hằng số.
- Hằng số là giá trị được gán cho biến vào thời điểm khai báo và sẽ không thay đổi về sau.

```
public final int MAX_COLS =100;
```

117

117

3.2.4. Từ khóa Final – biến Final

```
class Car {
    private final String brand; // Biến final phải được khởi tạo một lần duy nhất

    // Constructor
    Car(String brand) {
        this.brand = brand;
    }

    void displayBrand() {
        System.out.println("Hãng xe: " + brand);
    }
}
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota");
        myCar.displayBrand();

        // myCar.brand = "Honda"; //  Lỗi: Không thể thay đổi giá trị biến final
    }
}
```

118

118

3.2.4. Từ khóa Final - Phương thức final

- Phương thức hằng
- Được sử dụng để ngăn chặn việc ghi đè (override) hoặc che lấp (hidden) trong các lớp Java.
- Phương thức được khai báo là private hoặc là một thành phần của lớp final thì được xem là phương thức hằng.
- Phương thức hằng không thể khai báo là trừu tượng (abstract).

```
public final void find()
{
    //.....
}
```

119

119

3.2.4. Từ khóa Final - Phương thức final

```
class Car {
    final void startEngine() {
        System.out.println("Xe đang khởi động...");
    }
}

class ElectricCar extends Car {
    //  Lỗi: Không thể ghi đè phương thức final
    // @Override
    // void startEngine() {
    //     System.out.println("Xe điện khởi động...");
    // }
}
```

120

120

3.2.4. Từ khóa Final - Lớp Final

- Lớp hằng

➢ Là lớp không có lớp con (lớp vô sinh)

➢ Hay là lớp không có kế thừa

➢ Được sử dụng để hạn chế việc thừa kế và ngăn chặn việc sửa đổi một lớp.

```
public final class Student {
    // ...
}
```

121

121

3.2.4. Từ khóa Final - Lớp Final

```
final class Car {
    void startEngine() {
        System.out.println("Xe đang khởi động...");
    }
}

// Lỗi: Không thể kế thừa lớp final
// class ElectricCar extends Car { }
```

Giúp **bảo vệ cấu trúc lớp**, đảm bảo không có lớp con nào thay đổi hành vi.

122

122

3. Tính đa hình

➢ **Đa hình** (Polymorphism): Cùng một phương thức có thể có những cách thi hành (cách triển khai) khác nhau tại những thời điểm khác nhau. Trong Java tự động thể hiện tính đa hình.

➢ Cho phép thay đổi hành vi của phương thức tùy theo ngữ cảnh.

➢ Có 2 loại đa hình:

- ➢ Đa hình tại thời điểm biên dịch (Compile-time Polymorphism) → Method Overloading.
- ➢ Đa hình tại thời điểm chạy (Runtime Polymorphism) → Method Overriding.

123

123

3.3. Tính đa hình

```
package tronvuong;
class Hinh {
    public void Ve(){
        System.out.println("Ve hinh");
    }
}
```

```
package tronvuong;
class HinhTron extends Hinh {
    public void Ve(){
        System.out.println("Ve tron");
    }
}
```

```
package tronvuong;
class HinhVuong extends Hinh{
    public void Ve(){
        System.out.println("Ve vuong");
    }
}
```

124

124

3.3. Tính đa hình

```
package tronvuong;
public class TronVuong {
    public static void main(String[]
        args) { Hinh h=new Hinh();
        h.Ve();
        Hinh h1 = new
        HinhVuong(); h1.Ve();
        Hinh h2 = new
        HinhTron(); h2.Ve(); }
}
```

Kết quả xuất ra màn hình:

- Ve hinh
- Ve vuong
- Ve tron

125

125

3.3. Tính đa hình

```
interface Animal {
    void makeSound();
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println("Méo méo");
    }
}
```

```
class Dog implements Animal {
    public void makeSound() {
        System.out.println("Quẩu quẩu");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal pet = new Dog();
        pet.makeSound(); // Output: ?
    }
}
```

126

126

3.3. Tính đa hình

- **Đa hình tại thời điểm biên dịch** (Compile-time Polymorphism)
  - Xảy ra khi chương trình được biên dịch, phương thức nào được gọi được xác định ngay khi biên dịch.
  - Được thực hiện thông qua Method Overloading (Nạp chồng phương thức).
  - Trình biên dịch chọn phương thức phù hợp dựa vào danh sách tham số.

127

127

3.3.1. Nạp chồng phương thức

- **Overloading method:** Việc khai báo trong **một lớp** nhiều **phương thức có cùng tên nhưng khác tham số** (khác kiểu dữ liệu, khác số lượng tham số) gọi là **khai báo chồng phương thức (nạp chồng phương thức)**

128

128

3.3.1. Nạp chồng phương thức

- **Overloading method:**

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

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

129

129

3.3.1. Nạp chồng phương thức

```
class Car {
   void startEngine() {
      System.out.println("Xe đang khởi động...");
   }

   // Overloading: Thêm tham số
   void startEngine(String mode) {
      System.out.println("Xe đang khởi động ở chế độ: " + mode);
   }

   // Overloading: Thay đổi kiểu dữ liệu tham số
   void startEngine(int power) {
      System.out.println("Xe đang khởi động với công suất: " + power + " mã lực");
   }
}
```

130

130

3.3. Tính đa hình

- **Đa hình tại thời điểm chạy** (Runtime Polymorphism)
  - Xảy ra khi chương trình đang chạy.
  - Được thực hiện thông qua Method Overriding (Ghi đè phương thức).
  - Java quyết định phương thức nào sẽ được gọi dựa vào đối tượng thực sự tại runtime

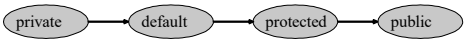
131

131

3.3.2. Ghi đè phương thức

➢ **Overriding Method**

- Khi lớp con cung cấp cách triển khai mới cho phương thức của lớp cha.
- Được định nghĩa trong lớp con
- Có tên, kiểu trả về & các đối số giống với phương thức của lớp cha
- Có kiểu, phạm vi truy cập không "nhỏ hơn" phương thức trong lớp cha
- Không thể ghi đè phương thức final, static hoặc private.



132

132

3.3.2. Ghi đè phương thức

```
class Hinhhoc { ...
        public float tinhdientich() {
                return 0;
        }
...
}
class HinhVuong extends Hinhhoc {
        private int canh;
        public float tinhdientich() {
                return canh*canh;
        }
...
}
```

Chỉ có thể **public** do phương thức tinhdientich() của lớp cha là public

133

3.3.2. Ghi đè phương thức

```
class HinhChuNhat extends HinhVuong {
 private int cd;
 private int cr;
    public float tinhdientich() { return cd*cr;
 }
 ...
}
```

Chỉ có thể public do phương thức tinhdientich() của lớp cha là public

134

3.3.2. Ghi đè phương thức

```
class Car {
    void startEngine() {
        System.out.println("Xe đang khởi động...");
    }
}
// Lớp con ghi đè phương thức của lớp cha
class ElectricCar extends Car {
    @Override
    void startEngine() {
        System.out.println("Xe điện khởi động êm ái...");
    }
}
                                public class Main {
                                    public static void main(String[] args) {
                                        Car myCar = new ElectricCar();
                                        myCar.startEngine(); // Gọi phương thức ghi đè
                                    }
                                }
```


135

3.3.3. Overloading và overriding

‘	Override	Overload
Hành vi	Thay đổi hành vi hiện tại của phương thức.	Thêm hoặc mở rộng cho hành vi của phương thức.
Đa hình	Thể hiện tính đa hình tại run time.	Thể hiện tính đa hình tại compile time.
Danh sách tham số	Danh sách tham số phải giống nhau.	Danh sách tham số có thể khác nhau.
Quyền truy cập	Phương thức ghi đè ở lớp con phải có quyền truy cập bằng hoặc lớn hơn phương thức được ghi đè ở lớp cha.	Các phương thức nạp chồng có thể có quyền truy cập khác nhau.
Giá trị trả về	Kiểu trả về bắt buộc phải giống nhau.	Kiểu trả về có thể khác nhau.
Phạm vi	Xảy ra giữa 2 class có quan hệ kế thừa	Xảy ra trong phạm vi cùng 1 class.

136

3.3.3. Overloading và overriding

Ví dụ:

```
class Sinhvien {
…
 public void xemThongTinSV() {
     …
 }
 public void xemThongTinSV(String psMaSv) {
     …
 }
}
```


137

3.4. Tính trừu tượng

- Tính trừu tượng (Abstraction) trong Java là tính chất không thể hiện cụ thể mà chỉ nêu tên vấn đề. Đó là một quá trình **ẩn đi các chi tiết triển khai bên trong** và chỉ **hiển thị những tính năng thiết yếu** của đối tượng tới người dung.
- Giúp **giảm sự phức tạp của hệ thống**, tập trung vào những gì quan trọng nhất.
- Có 2 cách (cơ bản) để đạt được sự trừu tượng hóa trong java
  - Sử dụng lớp abstract
  - Sử dụng interface

138

3.4. Tính trừu tượng

- Ví dụ: Lái xe hơi
- Khi lái một chiếc xe hơi, bạn chỉ cần biết cách sử dụng vô lăng, phanh, ga, mà không cần biết cách động cơ hoạt động bên trong.
- Abstraction trong lập trình giống như cách chúng ta sử dụng xe hơi mà không cần quan tâm đến cách động cơ hoạt động!

139

139

3.4. Tính trừu tượng

- Có 2 cách để thể hiện trừu tượng:
  - Lớp trừu tượng (abstract class)
  - Giao diện (interface)

140

140

3.4. Tính trừu tượng

```
abstract class Car {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }

    abstract void startEngine(); // Phương thức trừu tượng

    void displayInfo() { // Phương thức bình thường
        System.out.println("Hãng xe: " + brand);
    }
}
```

```
class ElectricCar extends Car {
    ElectricCar(String brand) {
        super(brand);
    }

    @Override
    void startEngine() {
        System.out.println(brand + " khởi động êm ái...");
    }
}
```

Lớp Car có phương thức trừu tượng startEngine() → Lớp con phải triển khai nó.
Ẩn đi chi tiết triển khai → Chỉ quan tâm đến hành vi chung của xe.

141

141

3.4.1. Lớp trừu tượng (Abstract Class)

- Là lớp đặc biệt, các phương thức chỉ được khai báo ở dạng khuôn mẫu (template) mà không được cài đặt chi tiết. Có thể chứa cả phương thức trừu tượng (không có thân) và phương thức thông thường (có thân).
- Dùng để định nghĩa các phương thức và thuộc tính chung cho các lớp con của nó.
- Dùng từ khóa **abstract** để khai báo một lớp trừu tượng
- Lớp **abstract** không thể tạo ra đối tượng, chỉ có thể được kế thừa

142

142

3.4.1. Lớp trừu tượng (Abstract Class)

- Có thể khai báo 1 hoặc nhiều phương thức trừu tượng bên trong lớp. Có thể khai báo các thuộc tính trong lớp trừu tượng. Lớp trừu tượng có cả phương thức bình thường (regular method) và phương thức trừu tượng (abstract method).
- Không thể tạo ra một đối tượng thuộc lớp trừu tượng, nhưng có thể khai báo biến thuộc lớp trừu tượng để tham chiếu đến các đối tượng thuộc lớp con của nó.
- Nếu lớp abstract có bất kỳ phương thức abstract nào, thì tất cả những lớp con kế thừa lớp abstract này đều phải định nghĩa lại những phương thức abstract của lớp đó.

143

143

3.4.1. Lớp trừu tượng (Abstract Class)

- **Khai báo:**

  [public] abstract class *Tênlớp*{
  <các thuộc tính>
  <các phương thức trừu tượng>
  }

- **Tính chất:** mặc định là public, bắt buộc phải có từ khoá abstract để xác định đây là một lớp trừu tượng.
- **Lưu ý:** Lớp trừu tượng cũng có thể kế thừa một lớp khác, nhưng lớp cha cũng phải là một lớp trừu tượng

144

144

3.4.1. Lớp trừu tượng (Abstract Class)

- **Khai báo:**

[public] abstract <kiểu dữ liệu trả về> <tên phương thức>([<các tham số>])

[throws <danh sách ngoại lệ>];

- Không khai báo tường minh mặc định là public.
- Tính chất của phương thức trừu tượng không được là **private** hay **static**
- Phương thức trừu tượng chỉ được khai báo dưới dạng khuôn mẫu nên không có phần dấu móc, ngoặc nhọn "{}" mà kết thúc bằng dấu chấm phẩy ";".
- Nếu một lớp bao gồm abstract method nhưng không được khai báo là lớp abstract thì sẽ gây ra lỗi

145

145

3.4.1. Lớp trừu tượng (Abstract Class)

```
package tronvalapphuong;
 astract class Hinh
 {
        static final double PI=3.1415;
        public abstract double DienTich();
        public abstract double TheTich();
  }
```

146

146

3.4.1. Lớp trừu tượng (Abstract Class)

```
package tronvalapphuong;
class HinhTron extends Hinh {
    private double R;
    public HinhTron(double r) {
        R=r;
    }
    @Override
    public double DienTich() {return PI*R*R; }
   @Override
    public double TheTich() {
        return 0;  }
}
```

147

147

3.4.1. Lớp trừu tượng (Abstract Class)

```
package tronvalapphuong;
class HinhLapPhuong extends Hinh {
    private double a;private double b;private double c;
    public HinhLapPhuong(double aa, double bb, double cc) {
        a=aa;b=bb;c=cc;
    }
    @Override
public double DienTich() {return(2*(a*b+b*c+a*c));   }
@Override
public double TheTich() {
    return a*b*c;
}
}
```

148

148

3.4.1. Lớp trừu tượng (Abstract Class)

```
package  tronvalapphuong;
public class TronVaLapPhuong {
    public static void main(String[] args) {
        Hinh  hr= new HinhTron(5.5);
        System.out.println("Hinh tron");
        System.out.println("Dien Tich: "+hr.DienTich());
        System.out.println("The Tich: "+hr.TheTich());
        Hinh hlp=new HinhLapPhuong(2,3,4);
        System.out.println("Hinh lap phuong: ");
        System.out.println("Dien Tich: "+hlp.DienTich());
        System.out.println("The Tich: "+hlp.TheTich());
    }
    }
```

149

149

3.4.1. Lớp trừu tượng (Abstract Class)

Hinh tron
DienTich: 95.03037499999999
TheTich: 0.0
Hinh lap phuong:
DienTich: 52.0
TheTich: 24.0

150

150

3.4.2. Giao tiếp (giao diện)

- Interfaces: giao tiếp của một lớp, là phần đặc tả (không có phần cài đặt cụ thể) của lớp, nó chứa các khai báo phương thức và thuộc tính để bên ngoài có thể truy xuất được.
  - Lớp sẽ cài đặt các phương thức trong interface.
  - Trong lập trình hiện đại các đối tượng không đưa ra cách truy cập cho một lớp, thay vào đó cung cấp các interface. Người lập trình dựa vào interface để gọi các dịch vụ mà lớp cung cấp.
  - Thuộc tính của interface là các hằng (final) và các phương thức là trừu tượng (mặc dù không có từ khóa abstract).
  - Một lớp implements nhiều interface (Multiple Inheritance)

151

151

3.4.2. Giao tiếp (giao diện)

- Không thể dẫn xuất từ lớp khác, nhưng có thể dẫn xuất từ những interface khác
- Nếu một lớp dẫn xuất từ một interface mà interface đó dẫn xuất từ các interface khác thì lớp đó phải định nghĩa tất cả các phương thức có trong các interface đó
- Khi định nghĩa một interface mới thì một kiểu dữ liệu tham chiếu cũng được tạo ra.
- Tất cả phương thức trong interface phải là public.
- Các phương thức phải được định nghĩa trong lớp dẫn xuất giao diện đó.

152

152

3.4.2. Giao tiếp (giao diện)

- **Khai báo:**

[public] interface <tên giao tiếp> [extends <danh sách giao tiếp>]

- **Tính chất:**
  - Luôn luôn là public, không khai báo tường minh thì mặc định là public.
  - *Danh sách các giao tiếp:* danh sách các giao tiếp cha đã được định nghĩa để kế thừa, các giao tiếp cha được phân cách nhau bởi dấu phẩy.
  - *Lưu ý:* Một giao tiếp chỉ có thể **kế thừa** từ các **giao tiếp khác** mà **không thể** được **kế thừa từ các lớp sẵn có**.

153

153

3.4.2. Giao tiếp (giao diện)

- **Phương thức của Interfaces**
- **Khai báo:**

[public] <kiểu giá trị trả về> <tên phương thức> ([<các tham số>]) [throws <danh sách ngoại lệ>];

154

154

3.4.2. Giao tiếp (giao diện)

- **Phương thức của Interfaces**
- **Tính chất:**
  - Thuộc tính hay phương thức luôn luôn là public, không khai báo tường minh thì mặc định là public.
  - Phương thức được khai báo dưới dạng mẫu, không có cài đặt chi tiết, không có phần dấu móc "{}" mà kết thúc bằng dấu chấm phẩy ";". Phần cài đặt chi tiết được thực hiện trong lớp sử dụng giao tiếp.
  - *Thuộc tính* luôn phải thêm từ khóa là hằng (final) và tĩnh (static)
  - *Thuộc tính* luôn có tính chất là hằng (final), tĩnh (static) và public. Do đó, cần gán giá trị khởi đầu ngay khi khai báo thuộc tính.

155

155

3.4.2. Giao tiếp (giao diện)

- **Sử dụng**
- **Khai báo:**

<tính chất> class <tên lớp> implements <các giao tiếp>{…}

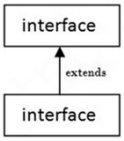
  - **Tính chất** và **tên lớp** được sử dụng như trong khai báo lớp thông thường.
  - **Các giao tiếp:** một lớp có thể cài đặt nhiều giao tiếp. Khi đó, lớp phải cài đặt cụ thể tất cả các phương thức của tất cả các giao tiếp mà nó sử dụng.
  - *Lưu ý:* Một phương thức được khai báo trong giao tiếp phải được cài đặt cụ thể trong lớp có cài đặt giao tiếp nhưng **không được** phép **khai báo chồng**.

156

156

3.4.2. Giao tiếp (giao diện)

- Quan hệ giữa các lớp interface

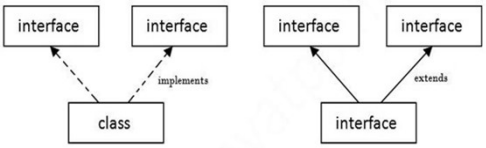


157

157

3.4.2. Giao tiếp (giao diện)

- Đa thừa kế trong Java



158

158

3.4.2. Giao tiếp (giao diện)

- **Ví dụ:**

```
// Định nghĩa một interface "Hinh" trong tập tin
package vidu1;
public interface Hinh {
        final double PI=3.1415;
        public double DienTich();
        public double ChuVi();
        public String LayTenHinh();
        public void Nhap();
}
```

159

159

3.4.2. Giao tiếp (giao diện)

- Định nghĩa lớp "HinhTron" implement từ lớp "Hinh"

```
package vidu1;
import java.util.Scanner;
class HinhTron implements Hinh{
   private double R;
   @Override
   public double DienTich() {
      return PI*R*R;
   }
   @Override
   public double ChuVi() {
      return 2*PI*R;
   }
```

```
@Override
   public String LayTenHinh() {
      return ("Hinh tròn");
   }
   @Override
   public void Nhap(){
      System.out.print("Nhap R=");
      Scanner scan = new
Scanner(System.in);
      R=scan.nextDouble();
   }
}
```

160

160

3.4.2. Giao tiếp (giao diện)

- Định nghĩa lớp "HinhVuong" implement từ lớp "Hinh"

```
package vidu1;
import java.util.Scanner;
class HinhVuong implements Hinh{
   private double canh;
   @Override
   public double DienTich() {
      return canh*canh;
   }
   @Override
   public double ChuVi() {
      return canh*4;
   }
```

```
@Override
   public String LayTenHinh() {
      return ("Hinh vuông");
   }
   @Override
   public void Nhap()
   {
      System.out.print("Nhap canh=");
      Scanner scan = new
Scanner(System.in);
      canh=scan.nextDouble();
   }
}
```

161

161

3.4.2. Giao tiếp (giao diện)

- Sử dụng các lớp

```
package vidu1;  public class Vidu1 {
  public static void main(String[] args) {
          Hinh h=new  HinhTron();  h.Nhap();
          System.out.println("Dien tich hình tron:"+h.DienTich());
          System.out.println("Chu vi hình tron:"+h.ChuVi());
          h= new HinhVuong();
          h.Nhap();
          System.out.println("Dien tich hình vuong:"+h.DienTich());
          System.out.println("Chu vi hình :"+h.ChuVi());
          }
}
```

162

162

3.4.2. Giao tiếp (giao diện)

- Kết quả

Nhap R=2

Dien tich hình tron:12.566 Chu vi hình tron:12.566

Nhap canh=4

Dien tich hình vuong:16.0 Chu vi hình :16.0

3.4.2. Giao tiếp (giao diện)

- Hiện tại, ngoài các phương thức trừu tượng, Java đã bổ sung thêm:
- Default methods: Cho phép cung cấp phương thức có thân trong interface.
- Static methods: Cho phép định nghĩa phương thức tĩnh trong interface

3.4.2. Giao tiếp (giao diện)

- **Khai báo Default methods :**

default <kiểu giá trị trả về> <tên phương thức> ([<các tham số>]) {.....}

- **Tính chất:**
  - Phương thức *default* giúp mở rộng interface mà không phải lo ngại phá vỡ các class được implements từ nó
  - Phương thức default giúp tháo gỡ các class cơ sở (base class), có thể tạo phương thức default và trong class được implement có thể chọn phương thức để *override*
  - Phương thức default cũng có thể được gọi là phương thức Defender (Defender Methods) hay là phương thức Virtual mở rộng (Virtual extension methods)

3.4.2. Giao tiếp (giao diện)

```
public   interface  Interface1 {
         void         method1(String str);
         default   void      log(String str){
                   System.out.println("I1 logging::"+str); print(str);
         }
}
```


3.4.2. Giao tiếp (giao diện)

- **Khai báo Static methods:**

static <kiểu giá trị trả về> <tên phương thức> ([<các tham số>]) {.....}

3.4.2. Giao tiếp (giao diện)

- **Tính chất Static methods :**
  - Cũng giống phương thức *default* ngoại trừ việc nó **không thể được override** trong class được implements.
  - Phương thức static chỉ hiển thị trong phương thức của interface, **nếu xóa** phương thức static trong **class được implements**, chúng ta sẽ không thể sử dụng nó cho đối tượng (object) của class được implements
  - Phương thức static rất hữu ích trong việc cung cấp các phương thức tiện ích, ví dụ như là kiểm tra null, sắp xếp tập hợp, v.v…
  - Phương thức static giúp chúng ta bảo mật, không cho phép class implements từ nó có thể override

3.4.2. Giao tiếp (giao diện)

```
// Interface Vehicle với Static Method
interface Vehicle {
    void startEngine();

    // Static method có thân
    static void showGuide() {
        System.out.println("Hướng dẫn sử dụng xe: Kiểm tra nhiên liệu trước khi khởi động.");
    }
}

// Lớp Car triển khai Vehicle
class Car implements Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Xe hơi đang khởi động...");
    }
}
```

169

169

3.4.2. Giao tiếp (giao diện)

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.startEngine();

        // Gọi Static Method từ interface mà không cần tạo đối tượng
        Vehicle.showGuide();
    }
}
```

```
Xe hơi đang khởi động...
Hướng dẫn sử dụng xe: Kiểm tra nhiên liệu trước khi khởi động.
```

170

170

3.4.2. Giao tiếp (giao diện)

- Dùng default methods khi:
  - Cần mở rộng interface mà không phá vỡ các lớp cũ.
  - Muốn cung cấp hành vi mặc định cho các lớp triển khai.
- Dùng static methods khi:
  - Cần cung cấp phương thức tiện ích chung (utility function): phương thức được sử dụng rộng rãi trong nhiều phần khác nhau của chương trình mà không cần tạo đối tượng
  - Không cần hoặc không muốn lớp triển khai ghi đè.

171

171

3.4.2. Giao tiếp (giao diện)

- Quan hệ giữa Class và Interfaces

	Class	Interface
Class	extends	implements
Interface		extends

172

172

3.4. Tính trừu tượng

Đặc điểm	**Lớp Trừu Tượng (Abstract Class)**	**Giao Diện (Interface)**
Tính trừu tượng	Có (có thể chứa phương thức có thân)	Có (chỉ chứa phương thức trừu tượng hoặc mặc định)
Tính đa hình	Có (đa hình qua kế thừa)	Có (đa hình qua implements)
Chứa biến thành viên?	Có (biến protected, private, public)	Không (chỉ có static final)
Kế thừa nhiều lớp?	Không hỗ trợ	Hỗ trợ đa kế thừa
Tốc độ thực thi	Nhanh hơn (do có thể có phương thức thực thi sẵn)	Chậm hơn (do phải ghi đè tất cả phương thức)

173

173

5. NHỮNG VẤN ĐỀ NÂNG CAO

174

174

Những vấn đề nâng cao

- Block khởi tạo (Initializer Block),
- Anonymous Class
- Lambda expression
- Nested Class (Lớp lồng nhau)
- Enum (Liệt kê)

175

175

5.1. Block khởi tạo

- **Block khởi tạo (Initializer Block):** là một khối lệnh được sử dụng để khởi tạo giá trị ban đầu cho các thuộc tính của lớp. Bao gồm:
  - Instance Initializer Block (Khởi tạo cho từng đối tượng).
  - Static Initializer Block (Chỉ chạy một lần cho cả lớp).

176

176

5.1. Block khởi tạo

- Instance Initializer Block: được gọi mỗi khi một đối tượng mới được tạo, trước cả constructor. Được sử dụng để thực hiện các tác vụ khởi tạo chung mà tất cả constructor đều cần.

177

177

5.1. Block khởi tạo

OLD

```
class Car {
    String brand;
    int speed;

    // Constructor 1
    Car() {
        speed = 50; // Lặp lại trong mỗi constructor
        System.out.println("Constructor không tham số được gọi!");
    }

    // Constructor 2
    Car(String brand) {
        this.brand = brand;
        speed = 50; // Lặp lại trong mỗi constructor
        System.out.println("Constructor có tham số được gọi!");
    }
}
```

178

178

5.1. Block khởi tạo

NEW

```
class Car {
    String brand;
    int speed;
    // Instance Initializer Block (chạy trước mọi constructor)
    {
        speed = 50;
        System.out.println("Instance Initializer Block được gọi!");
    }
    // Constructor 1
    Car() {
        System.out.println("Constructor không tham số được gọi!");
    }
    // Constructor 2
    Car(String brand) {
        this.brand = brand;
        System.out.println("Constructor có tham số được gọi!");
    }
}
```

179

179

5.1. Block khởi tạo

- Static Initializer Block (Khối khởi tạo tĩnh): Static Initializer Block (static {}) được dùng để khởi tạo dữ liệu tĩnh (static) trước khi lớp được sử dụng. Được sử dụng để khởi tạo các biến static.
- Thường dùng để khởi tạo tài nguyên như **kết nối CSDL, cấu hình hệ thống**.

180

180

5.2. Anonymous Class

- Anonymous Class (Lớp vô danh) là một lớp không có tên.
- Được sử dụng khi cần tạo một lớp con chỉ sử dụng một lần.
- Được sử dụng khi: Khi cần ghi đè (override) phương thức của lớp cha hoặc interface mà không muốn tạo một lớp con riêng biệt. Khi cần triển khai nhanh một interface hoặc một lớp trừu tượng chỉ một lần duy nhất. Khi cần một cách viết gọn gàng thay vì tạo một file/class riêng.

181

181

5.2. Anonymous Class

```
// Định nghĩa interface Vehicle
interface Vehicle {
    void startEngine();
}
public class Main {
    public static void main(String[] args) {
        // Anonymous Class triển khai interface Vehicle
        Vehicle myCar = new Vehicle() {
            @Override
            public void startEngine() {
                System.out.println("Xe đang khởi động...");
            }
        };

        myCar.startEngine();
    }
}
```

Tạo một lớp vô danh (Anonymous Class) triển khai trực tiếp Vehicle.Không cần tạo một lớp Car riêng để triển khai Vehicle

182

182

5.2. Anonymous Class

```
// Lớp trừu tượng Car
abstract class Car {
    abstract void startEngine();
}
public class Main {
    public static void main(String[] args) {
        // Anonymous Class kế thừa từ Car
        Car myCar = new Car() {
            @Override
            void startEngine() {
                System.out.println("Xe điện khởi động êm ái...");
            }
        };

        myCar.startEngine();
    }
}
```

Anonymous Class kế thừa trực tiếp từ lớp trừu tượng Car và ghi đè phương thức startEngine().Không cần tạo một lớp ElectricCar riêng.

183

183

5.2. Anonymous Class

- Trước đây, Anonymous Class là cách duy nhất để tạo lớp con tạm thời (temporary subclass) nhằm triển khai một interface hoặc lớp trừu tượng mà không cần tạo một file riêng. Tuy nhiên, Anonymous Class có cú pháp dài dòng, đặc biệt khi triển khai các interface có một phương thức duy nhất (Functional Interface).
- Vì vậy, hiện nay, Lambda Expression ra đời để giúp viết mã ngắn gọn hơn và rõ ràng hơn khi làm việc với Functional Interface.

184

184

5.3. Lambda Expressions

- Lambda giúp viết code ngắn gọn hơn, thay thế Anonymous Class
- Bắt buộc phải có Functional Interface
- Cú pháp Lambda Expressions:

  (parameters) -> expression
- Ví dụ:

```
(x, y) -> x + y
() -> System.out.println("Hello World");
```

185

185

5.3. Lambda Expressions

- Functional Interface: Interface có đúng **một phương thức trừu tượng**.

```
@FunctionalInterface
interface Vehicle {
    void startEngine();
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = () -> System.out.println("Xe đang khởi động...");
        car.startEngine();
    }
}
```

186

186

5.3. Lambda Expressions

- Lợi ích của Lambda Expressions
  - Code ngắn gọn, dễ đọc hơn.
  - Giảm số dòng code khi làm việc với Collections **& Stream API**.

```
List<Integer> list = Arrays.asList(1, 2, 3);
list.forEach(n -> System.out.println(n));
```

187

187

5.4. Nested Class (Lớp lồng nhau)

- Nested Class là một lớp được khai báo bên trong một lớp khác.
- **Các loại Nested Class**
  - Static Nested Class: Lớp lồng tĩnh
  - Non-Static Nested Class (Inner Class): Lớp lồng không tĩnh
    - Member Inner Class (Lớp bên trong thông thường)
    - Local Inner Class (Lớp lồng cục bộ)
    - Anonymous Inner Class (Lớp lồng vô danh)

188

188

5.4. Nested Class (Lớp lồng nhau)

- Static Nested Class
- Có từ khóa static, không cần tham chiếu đến lớp ngoài.

```
class Outer {
    static class Nested {
        void display() {
            System.out.println("Static Nested Class");
        }
    }
}
```

189

189

5.4. Nested Class (Lớp lồng nhau)

- Static Nested Class

```
class Car {
    String brand;

    // Static Nested Class
    static class Engine {
        void start() {
            System.out.println("Động cơ đang khởi động...");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Car.Engine engine = new Car.Engine();
        engine.start();
    }
}
```

Lớp Engine có thể được sử dụng mà không cần tạo đối tượng Car

190

190

5.4. Nested Class (Lớp lồng nhau)

- Non-Static Inner Class
- Có thể truy cập trực tiếp thành viên của lớp ngoài.

```
class Outer {
    class Inner {
        void display() {
        System.out.println("Non-Static Inner Class"); \
        }
    }
}
```

191

191

5.4. Nested Class (Lớp lồng nhau)

- Non-static Inner Class

```
class Car {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }

    // Non-static Inner Class
    class Engine {
        void start() {
            System.out.println(brand + " có động cơ đang
khởi động...");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car car = new Car("Toyota");
        Car.Engine engine = car.new Engine();
        engine.start();
    }
}
```

Lớp Engine cần đối tượng Car để khởi tạo

192

192

5.4. Nested Class (Lớp lồng nhau)

- Anonymous Inner Class
- Không có tên, thường dùng để override một phương thức.

```
Runnable r = new Runnable() {
    public void run() {
            System.out.println("Hello"); \
    }
};
new Thread(r).start();
```

193

193

5.4. Nested Class (Lớp lồng nhau)

- Local Inner Class
- Khai báo trong một phương thức, chỉ có hiệu lực trong phương thức đó

```
void myMethod() {
    class Local {
        void msg() { System.out.println("Local Inner Class");
}
    }
    Local obj = new Local();
    obj.msg();
}
```

194

194

5.5. Enum

- enum là kiểu dữ liệu đặc biệt chứa danh sách các hằng số.
- Dùng thay thế final static, giúp mã nguồn dễ đọc hơn.

```
enum CarType {
    ELECTRIC, GASOLINE, HYBRID
}

public class Main {
    public static void main(String[] args) {
        CarType myCar = CarType.ELECTRIC;
        System.out.println("Loại xe của tôi: " + myCar);
    }
}
```

195

195

5.5. Enum

```
enum CarType {
    ELECTRIC("Xe điện"), GASOLINE("Xe chạy xăng"), HYBRID("Xe lai");

    private String description;

    CarType(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }
}

public class Main {
    public static void main(String[] args) {
        CarType myCar = CarType.ELECTRIC;
        System.out.println("Loại xe của tôi: " + myCar.getDescription());
    }
}
```

Enum có thể chứa constructor, phương thức, và biến thành viên.

196

196

6. SOLID TRONG JAVA

197

197

SOLID

- SOLID là một tập hợp năm nguyên tắc thiết kế phần mềm hướng đối tượng, giúp tạo ra mã nguồn dễ bảo trì, mở rộng và tái sử dụng. SOLID là viết tắt của:
  - S - Single Responsibility Principle (SRP) - Nguyên tắc trách nhiệm đơn lẻ
  - O - Open/Closed Principle (OCP) - Nguyên tắc mở/đóng
  - L - Liskov Substitution Principle (LSP) - Nguyên tắc thay thế Liskov
  - I - Interface Segregation Principle (ISP) - Nguyên tắc phân tách giao diện
  - D - Dependency Inversion Principle (DIP) - Nguyên tắc đảo ngược sự phụ thuộc

198

198

SOLID - S

- S - Single Responsibility Principle (SRP) - Nguyên tắc trách nhiệm đơn lẻ
  - Một class chỉ nên giữ 1 trách nhiệm duy nhất (Chỉ có thể sửa đổi class với 1 lý do duy nhất)
  - Điều này giúp mã nguồn dễ bảo trì hơn, giảm thiểu sự phụ thuộc giữa các phần của hệ thống.

199

199

SOLID - S


```
class Invoice {
    private String customer;

    public void saveToDatabase() {
        // Code lưu hóa đơn vào DB
    }

    public void printInvoice() {
        // Code in hóa đơn
    }
}
```


```
class Invoice {
    private String customer;
    // Getter, Setter...
}

class InvoicePersistence {
    public void saveToDatabase(Invoice invoice) {
        // Code lưu hóa đơn vào DB
    }
}
```

200

200

SOLID - O

- Open/Closed Principle (OCP) - Nguyên tắc mở/đóng
- Class mở để mở rộng nhưng đóng để sửa đổi. Khi muốn thêm tính năng, ta mở rộng class thay vì sửa trực tiếp.

201

201

SOLID - L

- Liskov Substitution Principle (LSP) - Nguyên tắc thay thế Liskov
- Lớp con có thể thay thế hoàn toàn lớp cha mà không làm thay đổi tính đúng đắn của chương trình.

202

202

SOLID - I

- Interface Segregation Principle (ISP) - Nguyên tắc phân tách giao diện
- Một interface không nên ép các lớp triển khai phải sử dụng các phương thức mà chúng không cần.

203

203

SOLID - D

- Dependency Inversion Principle (DIP) - Nguyên tắc đảo ngược sự phụ thuộc
- Các module cấp cao không nên phụ thuộc vào module cấp thấp. Cả hai nên phụ thuộc vào abstraction.

204

204

Q & A

Giảng viên: Tạ Việt Phương
E-mail:phuongtv@uit.edu.vn

205

Bài tập

1. Viết chương trình sử dụng Inner Class để quản lý danh sách sinh viên
2. Viết chương trình sử dụng Lambda Expression để lọc danh sách số chẵn từ một danh sách số nguyên
3. Hãy thiết kế một hệ thống quản lý nhân viên bao gồm:

- Lớp Person có các thuộc tính: name (Tên), age (Tuổi), address (Địa chỉ)
- Lớp Employee kế thừa từ Person, bổ sung: employeeId (Mã nhân viên), salary (Lương)
- Lớp Manager kế thừa từ Employee, bổ sung: bonus (Tiền thưởng) và Phương thức calculateTotalSalary() để tính tổng lương (Lương + Thưởng).

206

Bài tập

Sử dụng kế thừa, constructor, ghi đè phương thức toString() để làm bài 3

207