Một vài chủ đề nâng cao

Bộ môn Công nghệ phần mềm
Khoa Công nghệ thông tin
Trường Đại học Công nghệ - ĐHQGHN

Nội dung

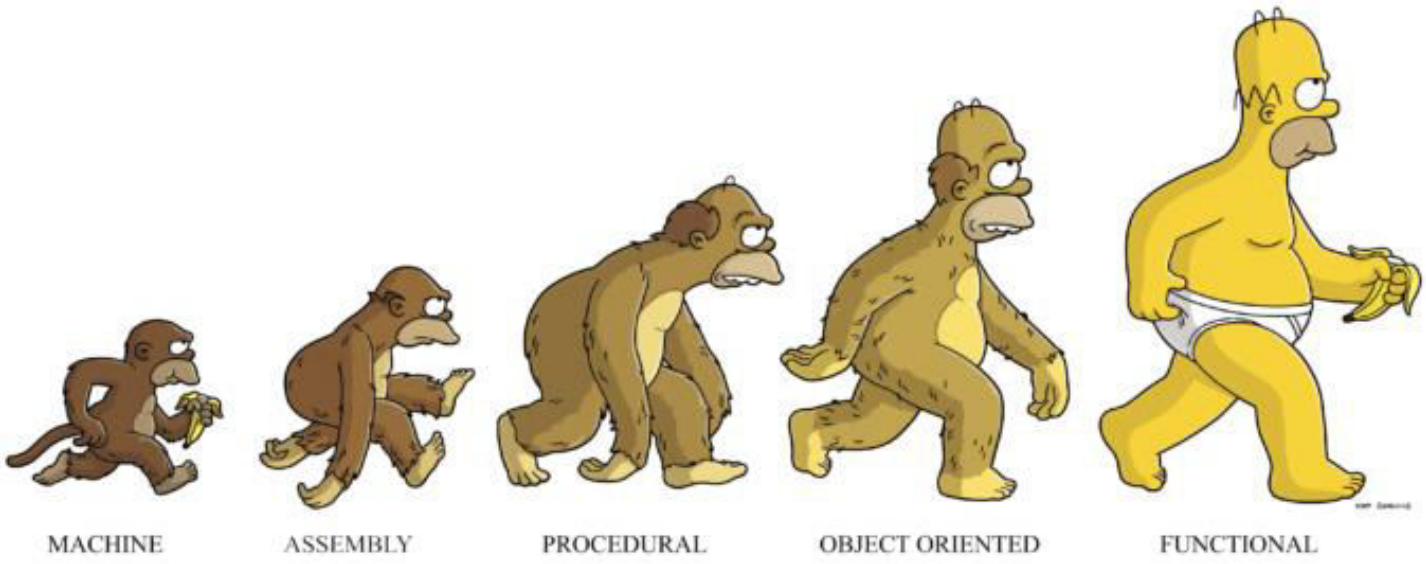
- Lập trình hàm trong Java
- Java Annotation
- Lớp lồng nhau (Nested Class)

Mục tiêu

- Nắm được kiến thức lập trình hàm và biết vận dụng các tính năng trong Java 8 để phục vụ lập trình hàm: Lambda Expression, Functional Interface, Method Reference
- Hiểu nhiệm vụ của Annotation và biết cách tạo một Annotation
- Hiểu về khái niệm lớp lồng nhau và biết phân biệt Nested Class và Inner Class

Lập trình hàm trong Java

Lập trình hàm



Lập trình hàm (Functional Programming - FP) là phương pháp lập trình chú trọng sử dụng các "hàm" để làm đơn vị xây dựng chương trình

FP vs OOP

```
List<Student> students = …
for (Student student : students) {
    System.out.println(student.getInfo());
}
```

OOP (Java)

```
val students: ArrayBuffer[Student] = …
students.foreach { s => println(s.getInfo) }
```

FP (Scala)

+ Mức trừu tượng cao -> súc tích, dễ hiểu
+ Sử dụng hàm -> dễ kiểm thử và gỡ lỗi
- Khó viết hơn OOP

FP vs OOP

FP	OOP
Tối đa hóa sử dụng hàm. Đơn vị: hàm, biến.	Dựa trên khái niệm mô phỏng các đối tượng. Đơn vị: đối tượng, hành vi.
Dữ liệu bất biến	Dữ liệu có thể thay đổi
Kết quả không bị ảnh hưởng bởi thứ tự thực thi các câu lệnh	PHẢI thực thi các câu lệnh theo thứ tự định sẵn
Sử dụng đệ quy để duyệt dữ liệu lặp	Sử dụng các cấu trúc lặp (for, while) để duyệt dữ liệu lặp
Tuân theo mô hình lập trình khai báo	Tuân theo mô hình lập trình mệnh lệnh
Hỗ trợ lập trình song song	KHÔNG hỗ trợ lập trình song song

Hỗ trợ FP trước Java 8

```
Runnable myTask = new Runnable() {

    @Override
    public void run() {
        System.out.println("Hello World in another Thread!");
    }
}

new Thread(myTask).start();
```

Sử dụng **Anonymous class**

-> Tốn quá nhiều (6) dòng code!

Khi Java 8 xuất hiện

```
Runnable myTask = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello World in another Thread!");
    }
}
new Thread(myTask).start();
```

```
Runnable runnableLE = () -> System.out.println("Hello World from
another thread!");
new Thread(runnableLE).start();
```

Gọi là Lambda Expression

Lambda Expression là gì?

Hỗ trợ FP trong Java 8

Các tính năng mới của Java 8 để hỗ trợ lập trình hàm:

- Lambda Expression
- Functional Interface
- Method Reference
- ...

Java 8 Lambda Expression (LE)

- Lambda Expression -> **Anonymous Function**
- Cú pháp: **(params) -> { body }**

```
// 0 tham số, in thông điệp
() -> System.out.println("Hello World in another Thread!")

// 1 tham số, tính bình phương
x -> x * x

// 2 tham số, tính tổng hai số
(a, b) -> a + b

// tính tổng hai số, sử dụng biến cục bộ
(a, b) -> { int sum = a + b; return sum; }
```

Functional Interface (FI)

- Interface chỉ có **duy nhất** một khai báo phương thức và là non-default
- Được tạo ra cho Lambda Expression sử dụng

```
@FunctionalInterface // đảm bảo khai báo duy nhất một phương thức
interface Greeter {
    String greet(String name);
}

// Cài đặt phương thức greet()
Greeter greeter = o -> "Hi " + o;
System.out.println(greeter.greet("UET Student"));
```

Output:
_> Hi UET Student

Các loại Functional Interface

Tên	Ví dụ sử dụng thực tế
Consumer	forEach(Consumer), peek(Consumer)
Supplier	reduce(Supplier), collect(Supplier)
Predicate	filter(Predicate)
Function	map(Function)

Xem thêm trong package **java.util.function.***

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Ví dụ FI 1: **Consumer**

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
Consumer<String> printer = message -> System.out.println(message);
printer.accept("Hello world from lambda expression!");
```

```
_> Hello world from lambda expression!
```

- Nhận **một** tham số, **không** trả về dữ liệu

Ví dụ FI 2: **Supplier**

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

```
Supplier<LocalDateTime> s = () -> LocalDateTime.now();
LocalDateTime time = s.get();
System.out.println(time);
```

```
_> 2020-11-23T10:21:20.00
```

- **Không** có tham số, trả về dữ liệu

Ví dụ FI 3: **Predicate**

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

```
List<Integer> numList = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
Predicate<Integer> greaterThanFive = x -> x > 5;

List<Integer> filterNums = numList.stream()
        .filter(greaterThanFive)
        .collect(Collectors.toList());
System.out.println(filterNums);
```

```
_> [6, 7, 8, 9]
```

- Nhận **một** tham số, trả về dữ liệu **boolean**

Ví dụ FI 4: **Function**

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

```
Function<String, Integer> computeLength = x -> x.length();
System.out.println(computeLength.apply("UET"));
```

_> 3

```
Function<String, Integer> computeLength = x -> x.length();
Function<Integer, Integer> dbl = x -> x * 2;
System.out.println(computeLength.andThen(dbl).apply("UET"));
```

_> 6

- Nhận **một** tham số, trả về dữ liệu **tùy chọn**

Method Reference

- Lambda Expression gọi vào phương thức có sẵn -> rút gọn code
- **Danh sách tham số** phải **khớp** giữa PT được gọi và PT khai báo trong Interface

```
//Consumer<String> printer=message -> System.out.println(message);

Consumer<String> printer = message -> System.out::println;
printer.accept("Hello world!");
```

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

T = String

```
public void println(String x){
    synchronized (this){
        ... }
}
```

Các kiểu Method Reference

Kiểu	Cú pháp	Ví dụ
static method	ClassName::staticMethodName	String::valueOf
constructor	ClassName::new	Student::new
instance method	object::instanceMethodName	new Student()::getInfo
arbitrary object	ContainingType::methodName	Student::getInfo

Ví dụ Method Reference

- Chuyển dữ liệu từ **List** sang **Array**

```
// Sử dụng Lambda Expression

List<String> list = Arrays.asList("C", "C++", "Java");
String[] array = list.stream().toArray(n -> new String[n]);
```

```
// Sử dụng Method Reference, ngắn gọn và dễ hiểu hơn!

List<String> list = Arrays.asList("C", "C++", "Java");
String[] array = list.stream().toArray(Array[]::new);
```

Java Annotation

Annotation là gì?

- Là một dạng siêu dữ liệu (metadata), cung cấp thêm thông tin cho các thành phần mã nguồn, vd: lớp, phương thức, thuộc tính, tham số,...

- Phục vụ cho 3 mục đích chính:
  - Chỉ dẫn cho **trình biên dịch**
  - Chỉ dẫn tại **thời điểm build**
  - Chỉ dẫn tại **thời gian chạy**

- Ví dụ: **@Override**, **@Deprecated**, **@SuppressWarnings**,...

Ví dụ Annotation

```
class Person {
    public void greet() {System.out.println("Hello");}
}
class Student extends Person {
    @Override
    public void greet2() {System.out.println("What zup, bro!");}
}
```

Lỗi biên dịch: ***Method does not override method from its superclass***

=> Annotation **@Override** chỉ cho trình biên dịch kiểm tra **greet2** ghi đè hợp lệ phương thức ở lớp cha hay không

Tự tạo Annotation: **CodeInfo**

```
// Khai báo một annotation với từ khóa @interface
@interface CodeInfo {
    public String author();
    public String desc() default “”;
}
```

```
// Sử dụng
class Student {
    @CodeInfo(
        author = “Developer 1”,
        desc = “Allow student to register his/her desired course”
    )
    public void registerCourse(Course course) {...}
}
```

CodeInfo chứa thông tin tác giả và mô tả về thành phần mã nguồn

Tự tạo Annotation: @Retention

```
@Retention(RetentionPolicy.RUNTIME)
@interface CodeInfo {
    public String author();
    public String desc() default “”;
}
```

Cung cấp thông tin thời gian tồn tại của annotation:

- **RetentionPolicy.SOURCE**: Chỉ trên mã nguồn
- **RetentionPolicy.CLASS**: Trên mã nguồn và bytecode
- **RetentionPolicy.RUNTIME**: Trên cả mã nguồn, bytecode, và lúc chạy trên JVM

RetentionPolicy.CLASS là mức mặc định, trình biên dịch có thể nhận ra sự có mặt của annotation từ mức này

Tự tạo Annotation: @Target

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@interface CodeInfo {
    public String author();
    public String desc() default “”;
}
```

Quy định vị trí mã nguồn có thể sử dụng annotation:

PACKAGE, TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE

Lấy thông tin annotation

```
for (Method method : Student.class.getMethods()) {
    if (method.isAnnotationPresent(CodeInfo.class)) {
        CodeInfo info = method.getAnnotation(CodeInfo.class);
        System.out.println("Method name: " + method.getName());
        System.out.println("Author: " + info.author());
        System.out.println("Description: " + info.desc());
    }
}
```

```
_> Method name: registerCourse
Author: Developer 1
Description: Allow student to
register his/her desired course
```

Sử dụng Reflection API để truy xuất thông tin lưu trữ tại các vị trí mã nguồn sử dụng annotation **CodeInfo**

Lớp lồng nhau (Nested Class)

Nested Class là gì?

Là một lớp Java được **khai báo bên trong** một lớp khác

```
class OuterClass {
    private class NestedClass {}
    ...
    private NestedClass nested = new NestedClass();
}
```

Nested Class có thể truy cập mọi thuộc tính, phương thức của Outer Class, kể cả khi khai báo là ***private***

+ Gộp các lớp liên quan với nhau
+ Tăng tính đóng gói
+ Code dễ đọc, dễ bảo trì

Static Nested Class

- Là một Nested Class được khai báo **static**
- Chỉ truy cập được các thuộc tính, phương thức tĩnh của lớp bên ngoài
- Có thể gọi dựa trên tên lớp bên ngoài

```
// Code thực tế tạo Dialog trong Android
AlertDialog myAlertDialog = new
AlertDialog.Builder(MainActivity.this).create();
```

Inner Class

- Là một Nested Class KHÔNG được khai báo **static**
- Được gán với mỗi đối tượng của lớp bên ngoài
- Không thể có thuộc tính, phương thức static
- Có thể gọi dựa trên tên của đối tượng lớp bên ngoài

```
// Code khởi tạo Camera trong Android
Camera cam = Camera.open();
Size size = cam.new Size(100, 100);
```

Tham khảo

- https://docs.oracle.com/javase/tutorial
- http://tutorials.jenkov.com/java