**Mục lục**

[**A.** **Элементы функционального программирования в синтаксисе Java.** 1](#_Toc155455393)

[**I.** **Функциональные интерфейсы** 1](#_Toc155455394)

[**Function<T, R>** 2](#_Toc155455395)

[**Consumer<T>** 2](#_Toc155455396)

[**Predicate<T>** 2](#_Toc155455397)

[**Supplier<T>** 3](#_Toc155455398)

[**II.** **Лямбда-выражения** 3](#_Toc155455399)

[**III.** **Ссылки на методы.** 4](#_Toc155455400)

[**1.** **Static method reference** 4](#_Toc155455401)

[**2.** **Instance method reference để nhận 1 tham số là object** 5](#_Toc155455402)

[**3.** **Instance method reference để 1 object nhận tham số** 5](#_Toc155455403)

[**4.** **Constructor method reference** 6](#_Toc155455404)

[**B.** **Custom exception classes (checked and unchecked)** 7](#_Toc155455405)

[**C.** **Inner/nested class (**Вложенные классы) 9](#_Toc155455406)

[**D.** **Local class trong java** 10](#_Toc155455407)

[**E.** **Java Anonymous Class** 10](#_Toc155455408)

[**F.** **Механизм рефлексии (reflection) в Java. Java reflection. Класс Class. Annotation.** 11](#_Toc155455409)

[**I.** **Classes** 11](#_Toc155455410)

[**II.** **Constructors** 13](#_Toc155455411)

[**III.** **Fields** 15](#_Toc155455412)

[**IV.** **Methods** 16](#_Toc155455413)

[**V.** **Annotations** 16](#_Toc155455414)

[**1.** **Định nghĩa** 17](#_Toc155455415)

[**2.** **Cách tạo Annotation** 17](#_Toc155455416)

[**3.** **Truy cập annotation với reflection.** 17](#_Toc155455417)

1. **Элементы функционального программирования в синтаксисе Java.**
2. **Функциональные интерфейсы**

* Функции представляются объектами
* Один из методов объекта вычисляет значение функции
* Интерфейсы для наиболее распространенных функций собраны в пакете **java.util.function**
* Такие интерфейсы с единственным абстрактным методом называются функциональными и помечаются аннотацией **@FunctionalInterface**
* Phương thức **default** and **static** không phá vỡ quy tắc của Functional interface. (Do phương thức abstract là public nên interface đó có thể có các phương thức khác là default hoặc static)
* Một Functional Interface có thể kế thừa từ một Functional Interface khác chỉ khi nó không có bất kỳ phương thức trừu tượng nào (Nếu nó thêm một phương thức trừu tượng mới, nó sẽ không còn là một Functional Interface nữa vì nó sẽ chứa nhiều hơn một abstract method.)

**Bốn functional interface cơ bản**

### **Function<T, R>**

import java.util.function;

@FunctionalInterface

public interface Function<T, R> {

R apply (T t);

}

Nhận một tham số T, trả về một giá trị tương ứng kiểu R. Dùng để biến đổi một giá trị thành một giá trị khác (map 1 với 1).

Function<String, Integer> getLength = str -> str.length();

getLength.apply("John Doe"); // 3

### **Consumer<T>**

import java.util.function;

@FunctionalInterface

public interface Consumer<T> {

void accept (T t);

}

Nhận vào một param kiểu T, không trả về gì cả. Nói cách dân dã là chỉ thích nhận lại nhưng không thèm cho đi.

Consumer<Integer> print = num -> System.out.println(num);

print.accept(3); // 3

### **Predicate<T>**

import java.util.function;

@FunctionalInterface

public interface Predicate<T> {

boolean test (T t);

}

Nhận một tham số T và trả về boolean. Dùng để check xem phần tử có thỏa mãn điều kiện hay không

Predicate<Integer> checkAge = age -> age > 18;

checkAge.test(10); // false

### **Supplier<T>**

import java.util.function;

@FunctionalInterface

public interface Supplier<T> {

T get ();

}

Functional interface này không nhận vào param nào, nhưng lại trả ra một giá trị nào đó. Thường dùng để generate stream vô hạn.

Supplier<Double> generate = () -> Math.random(100);

generate.get(); // Số random nào đó

4 functional interface trên khá cơ bản, được dùng trong các **stream operation** của Stream API.

* find(Predicate<T> predicate)/ filter(Predicate<T> predicate) 🡪 để lọc ra các phần tử phù hợp
* sort(Comparator<T> order)
* forEach(Consumer<T> action) chỉ dùng in ra thôi, không return gì nữa
* map(Function<T, R> func) để biến đổi phần tử 1-1

ĐỌC THÊM: [4 functional interface có sẵn và ví dụ với Stream API (tonghoangvu.id.vn)](https://blog.tonghoangvu.id.vn/4-functional-interface-co-san-va-vi-du-voi-stream-api)

1. **Лямбда-выражения**

[Nắm rõ JAVA LAMBDA EXPRESSION cho người mới bắt đầu (viblo.asia)](https://viblo.asia/p/nam-ro-java-lambda-expression-cho-nguoi-moi-bat-dau-RQqKLNwbl7z)

(argument-list) -> {body}

Biểu thức Lambda trong java gồm có 3 thành phần sau:

* Argument-list: danh sách tham số, có thể không có, có một hoặc nhiều tham số.
* Arrow-operator: toán tử mũi tiên được sử dụng để liên kết danh sách tham số và body của biểu thức.
* Body: nội dung thực thi, là 1 khối lệnh hoặc 1 biểu thức.

**(a, b, n) -> { doSomthing(); }**

Về khai báo danh sách tham số:

* Các tham số cách nhau bởi dấu phẩy. Danh sách tham số phải được đặt trong ngoặc tròn. Trường hợp có 1 tham số thì không bắt buộc và không cần thiết phải có dấu ngoặc tròn
* Không bắc buộc phải khai báo kiểu dữ liệu của tham số. Giả sử trường hợp dùng biểu thức Lambda để *implement* **function interface** thì kiểu dữ liệu của tham số sẽ được ngầm hiểu là kiểu dữ liệu đã khai báo ở abtract method

Về trình bày nội dung thực thi:

* Nội dung thực thi phải được đặt trong dấu ngoặc nhọn. Trường hợp có một dòng lệnh thì không bắt buộc
* Biểu thức Lambda nên chỉ có từ 2-5 dòng lệnh để tránh sự phức tạp về trình bày cũng như mục đích sử dụng biểu thức Lambda.

1. **Ссылки на методы.**

[Java 8 Method Reference (Phần 1) (viblo.asia)](https://viblo.asia/p/java-8-method-reference-phan-1-V3m5WObE5O7)

[Java 8 Method Reference (Phần 2) (viblo.asia)](https://viblo.asia/p/java-8-method-reference-phan-2-Az45bbPg5xY)

Một method reference là *cú pháp ngắn* của **lambda expression** giúp thực thi một method. Dưới đây là cú pháp chung cho một method reference:

Object :: methodName

Có 4 loại method preference

* **Method reference một static method**
* **Method reference một instance method của một object có kiểu đặc biệt.**
* **Method reference một instance method của một object đã tồn tại.**
* **Method reference một constructor**

1. **Static method reference**

*FunctionalInterface* test = Class::staticMethod

Consumer<String> getText = **System.out::println**;

getText.accept("Hello");

1. **Instance method reference để nhận 1 tham số là object**

(obj, args) -> obj.instanceMethod(args).

Method reference: *FunctionalInterface* test = **ObjectType::instanceMethod**

Ví dụ:

public class Human {  
 private String name;  
 private int age;  
  
 public Human(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
 public void move (){  
 System.out.println(name + " with " + age + " years old is running.");  
 }  
}

import java.util.function.Consumer;  
  
public class test {  
 public static void main(String[] args) {  
 Consumer<Human> test= **Human::move**;  
  
 Human human = new Human("Nghia", 20);  
 test.accept( human );  
 }  
}

1. **Instance method reference để 1 object nhận tham số**

(args) -> obj.instanceMethod(args)

Method reference: *FunctionalInterface* test = **obj::instanceMethod**

Ví dụ:

class Human có thêm 1 method như dưới

public class Human{

public void think (String what){

System.out.println(name + " with " + age + " years old is thinking about " + what);

}}

import java.util.function.Consumer;  
  
public class test {  
 public static void main(String[] args) {  
 Human human = new Human("Nghia", 20);

Consumer<String> move = **human::think**;  
 move.accept ("how to win the battle");  
  
 }  
}

1. **Constructor method reference**

(args) -> new ClassName(args)

Method reference : *FunctionalInterface* test = **ClassName::new**

Lamda expression tạo một object mới và ta chỉ cần reference constructor của class đó với từ khóa “**new**”. Ví dụ:

class Human {  
 private String name;  
 private int age;  
  
 public Human(String name) {  
 this.name = name;  
 }  
}

import java.util.function.Consumer;  
  
public class test {  
 public static void main(String[] args) {  
 Consumer<String> test= Human::new;  
 test.accept("Nghia");  
 }  
}

1. **Custom exception classes (checked and unchecked)**

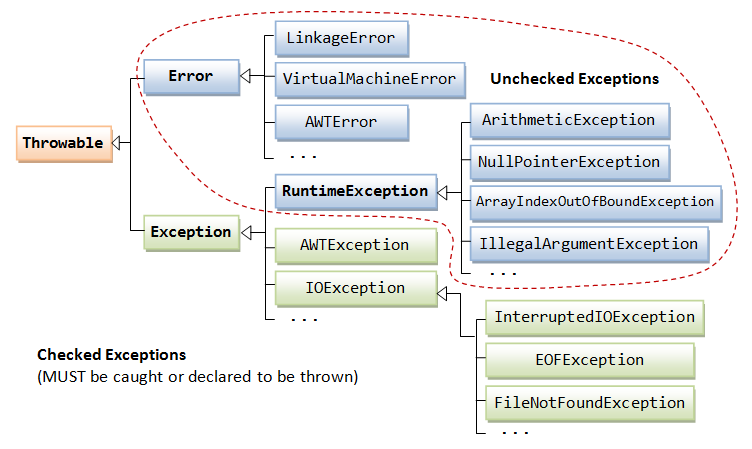
[Ngoại lệ và cách xử lý ngoại lệ trong Java (techmaster.vn)](https://techmaster.vn/posts/36954/ngoai-le-va-cach-xu-ly-ngoai-le-trong-java#toc-custom-exception)

[Nghệ Thuật Throw và Throws Trong Java - Bạn đã biết chưa? | Code Thu (youtube.com)](https://www.youtube.com/watch?v=q7FioKD7n1Q)

[Xử lý ngoại lệ trong Java (Exception Handling) - GP Coder (Lập trình Java)](https://gpcoder.com/2430-xu-ly-ngoai-le-trong-java-exception-handling/)

[Mình Đã Tạo Exception Java, Và Bạn Cũng Có Thể Làm Được! | Code Thu (youtube.com)](https://www.youtube.com/watch?v=e65dzoVGuvI)

Lớp **java.lang.Throwable** là lớp gốc của hệ thống phân cấp ngoại lệ. Nó được kế thừa bởi 2 lớp con là **Exception** và **Error**. Xem hình dưới đây:



  
**Các loại Exception trong Java**

Chủ yếu có 2 loại ngoại lệ: checked và unchecked. Mỗi error được coi là unchecked exception. Tuy nhiên, theo Oracle, có ba loại exception:

* **Checked Exception**: Ngoại lệ được kiểm tra bởi trình biên dịch; Trình biên dịch sẽ bắt bạn phải xử lý chúng ngay tại thời điểm biên dịch
* **Unchecked Exception**: Các loại ngoại lệ mà trình biên dịch không kiểm tra và không yêu cầu bạn phải xử lý tại thời điểm biên dịch. (Có thể hiểu là ngoại lệ mà TBD không check). **Unchecked exception** thường phản ánh các tình huống không thể dự đoán được tại thời điểm biên dịch và thường xuất hiện trong quá trình chạy chương trình.
* **Error**: **Error** biểu diễn những vấn đề nghiêm trọng mà chương trình không thể giải quyết hoặc xử lý được. Những lỗi này thường liên quan đến vấn đề về môi trường chạy của JVM, như hết bộ nhớ heap hoặc StackOverflow.

**Một số exception phổ biến:** [Ngoại lệ và cách xử lý ngoại lệ trong Java (techmaster.vn)](https://techmaster.vn/posts/36954/ngoai-le-va-cach-xu-ly-ngoai-le-trong-java#toc-custom-exception)

**Xử lý ngoại lệ (Exception Handling)**

Java cung cấp 5 từ khóa sử dụng để xử lý ngoại lệ đó là:

| **Từ khóa** | **Mô tả** |
| --- | --- |
| **try** | Sử dụng để chứa đoạn lệnh có thể gây ra ngoại lệ, nó phải được theo sau bởi khối **catch** hoặc **finally** |
| **catch** | Sử dụng để **xử lý** ngoại lệ, các khối code để xử lý ngoại lệ sẽ được đặt trong đây |
| **finally** | Sử dụng để thực thi đoạn mã cần thiết của chương trình. Khối này sẽ **luôn** được thực thi  [Khối lệnh finally trong java - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/exception-handling/khoi-lenh-finally-trong-java) |
| **throw** | Sử dụng để ném ra ngoại lệ cụ thể. Thông thường nó được sử dụng với *custom* *exception* (do người dùng tự định nghĩa)  [Từ khóa throw trong java - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/exception-handling/tu-khoa-throw-trong-java) |
| **throws** | Sử dụng để khai báo ngoại lệ. Nó được sử dụng chủ yếu với ***checked exception***  [Từ khóa throws trong java - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/exception-handling/tu-khoa-throws-trong-java) |

**Đọc thêm**:

Đa khối lệnh catch: [Đa khối lệnh catch trong java - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/exception-handling/da-khoi-lenh-catch-trong-java)

Khối lệnh try lồng nhau: [Khối lệnh try lồng nhau trong java - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/exception-handling/khoi-lenh-try-long-nhau-trong-java)

Sự khác nhau giữa throw và throws: [Sự khác nhau giữa throw và throws trong java - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/exception-handling/su-khac-nhau-giua-throw-va-throws-trong-java)

**Custom Exception**

Custom exception là ngoại lệ do người dùng tự định nghĩa, được sử dụng để tùy biến ngoại lệ theo yêu cầu của người dùng.

Để tạo custom exception thuộc loại *checked* chúng ta kế thừa từ lớp **Exception**, còn với *unchecked* *exception* thì kế thừa từ lớp **RuntimeException**

Cách tạo (custom) 1 exception: [Exception là gì? Cách xử lý lỗi và ngoại lệ trong Java (200lab.io)](https://200lab.io/blog/exception-la-gi-cach-xu-ly-exception-java/#5-custom-m%E1%BB%99t-exception-ho%E1%BA%B7c-error)

1. **Inner/nested class (**Вложенные классы)

[(31) Inner Class: từ ZERO đến HERO trong lập trình Java | Code Thu - YouTube](https://www.youtube.com/watch?v=JTs9bds61Vk)

[Java Inner Class - Viblo](https://viblo.asia/p/java-inner-class-PAoJeZoNL1j)

**Các kiểu của lớp lồng nhau trong java**

|  |  |
| --- | --- |
| **Kiểu** | **Mô tả** |
| [Member Inner Class](https://viettuts.vn/java-inner-class/member-inner-class-trong-java) | Một lớp được tạo ra bên trong một lớp và bên ngoài phương thức. |
| [Anonymous Inner Class](https://viettuts.vn/java-inner-class/anonymous-inner-class-trong-java) | Một lớp được tạo ra để implements interface hoặc extends class. Tên của nó được quyết định bởi trình biên dịch java.. |
| [Local Inner Class](https://viettuts.vn/java-inner-class/local-inner-class-trong-java) | Một lớp được tạo ra bên trong một phương thức. |
| [Static Nested Class](https://viettuts.vn/java-inner-class/static-nested-class-trong-java) | Một lớp static được tạo ra bên trong một lớp. |
| [Nested Interface](https://viettuts.vn/java-inner-class/nested-interface-trong-java) | Một interface được tạo ra bên trong một lớp hoặc một interface. |

1. **Local class trong java**

Local class là các [class](https://shareprogramming.net/object-va-class-trong-java/) được định nghĩa bên trong một *block*. *block* là một nhóm các câu lệnh nằm giữa cặp dấu “{}”. Thông thường *local class* được bên trong một [hàm](https://shareprogramming.net/ham-trong-java/).

Local class là một dạng đặc biệt của [Inner class](https://shareprogramming.net/inner-class-trong-java-voi-vi-du-cu-the/), vì vậy nó cũng có đầy đủ tính chất của một *Inner class*.

[1 Khai báo Local class](https://shareprogramming.net/local-class-trong-java/#Khai_bao_Local_class)

[2 Đặc điểm của local class](https://shareprogramming.net/local-class-trong-java/#Dac_diem_cua_local_class)

1. **Java Anonymous Class**

[Java Anonymous Class (viblo.asia)](https://viblo.asia/p/java-anonymous-class-n157G5J3vAje)

[Lớp Vô Danh (Anonymous) nhưng liệu có Vô Nghĩa? | Code Thu (youtube.com)](https://www.youtube.com/watch?v=bs3fv8W6GTo)

[Lớp Vô Danh để Triển Khai Interface Bạn Đã Biết? | Code Thu (youtube.com)](https://www.youtube.com/watch?v=rGVIyfDuf74)

Anonymous class là class được khai báo mà không có tên, được sử dụng để triển khai 1 abstract class hoặc 1 interface trực tiếp trong câu lệnh; Ví dụ:

abstract class B{

abstract void do(); }

public class A {

B temp = new B () { void do() { System.out.println(“Hello world”); };

temp.do();

}

Anonymos Class được sử dụng dưới 2 hình thức:

* Abstract class
* Interface

Thay vì sử dụng extends class, Anonymous Class có ưu điểm riêng. Trong các trường hợp trên, lớp Anonymous được viết ngay trong method của class khác, sử dụng ngay lúc định nghĩa xong. **Do vậy nên sử dụng Anonymous Class khi đối tượng chỉ được dùng một lần**. Thêm nữa Anonymous phù hợp trong trường hợp cần tạo đối tượng con, trong đó **method viết lại từ lớp cha là ít, và đơn giản**. Lúc đó code sẽ đơn giản hơn.

1. **Механизм рефлексии (reflection) в Java. Java reflection. Класс Class. Annotation.**

[Java reflection and anotation (viblo.asia)](https://viblo.asia/p/java-reflection-and-anotation-DbmvmWRMeAg)

[Reflection trong Java - Tìm hiểu nhanh về Java Reflection trong 5 phút (t3h.com.vn)](https://t3h.com.vn/tin-tuc/reflection-trong-java)

Trong Java, chúng ta có thể sử dụng Reflection để lấy các thông tin về:

* Classes
* Constructors
* Fields
* Methods

1. **Classes**

Khi sử dụng java reflection để duyệt qua một class thì việc đầu tiên thường phải làm đó là có được một đối tượng kiểu Class. Có ba cách lấy đối tượng của lớp Class như sau:

1. ***forName****(String className*)- phương thức **static** của lớp Class

class Demo {}

public class test {

public static void main(String args[]) {

try {

Class c = Class.forName("Demo");

System.out.println(c.getName());

}

catch(Exception e) {

System.out.println(e);

}

}

}

Output: *Demo*

forName() nhận tham số là **String** và trả lại dữ liệu kiểu **Class**. Không thể sử dụng cho kiểu dữ liệu nguyên thủy.

1. ***getClass*()-** phương thức của lớp **Object**

class Demo {}

class test {

void printName(Object obj) {

Class c = obj.getClass();

System.out.println(c.getName());

}

public static void main(String args[]) {

// Demo obj = new Demo();

String obj = new String ("xxx");

test t = new test();

t.printName(obj);

}

}

Output1: Demo

Output2: java.lang.String

getClass() không nhận tham số và trả lại dữ liệu kiểu **Class**

1. Cú pháp .***class***

Đôi khi, có tình huống khi một kiểu có sẵn nhưng không có thể hiện của lớp. Trong những trường hợp như thế, chúng ta có thể lấy lớp bằng cách thêm cú pháp .class vào tên của kiểu. Chúng ta cũng có thể sử dụng các cú pháp này với **các kiểu dữ liệu nguyên thủy**.

class Demo {  
 public static void main(String args[]) {  
 Class c1 = boolean.class;  
 System.*out*.println(c1.getName());  
  
 Class c2 = Demo.class;  
 System.*out*.println(c2.getName());  
 }  
}

Output:

*boolean*

*Demo*

1. **Constructors**

Các đối tượng lớp **Constructor** là những phuơng thức khởi tạo của một lớp. **Reflection** cho phép lấy ra những Contructor từ đối tượng kiểu **Class**.

* Sử dụng **getConstructors()** 🡪 Phương thức thuộc lớp **Class** này trả về một mảng **Constructor[]** chứa tất cả các constructor công khai (public) của lớp.
* Sử dụng **getConstructor(Class parameter)** 🡪 Phương thức thuộc lớp **Class** này trả về một đối tượng **Constructor** mô tả constructor công khai (public) của lớp, với kiểu dữ liệu tham số được chỉ định bởi mảng.

import **java.lang.reflect.Constructor**;  
class Demo{  
 private int a;  
 private String b;  
 public Demo(int a, String b) {  
 this.a = a;  
 this.b = b;  
 }  
}  
public class test {  
 public static void main(String[] args) {  
 // Lấy đối tượng Class cho lớp Demo  
 Class demoClass = Demo.class;  
 // Lấy danh sách các constructors của lớp Demo  
 Constructor[] constructors = demoClass.**getConstructors();** // In thông tin về mỗi constructor  
 for (Constructor constructor : constructors) {  
 System.*out*.println("Constructor: " + constructor);  
 }  
 }  
}

Output: *Constructor: public Demo(int,java.lang.String)*

Thậm chí có thể truy cập được đến tham số của các Constructors. (Các tham số này đều được đưa về kiểu Class). Ví dụ: Ta bổ sung thêm vào phần // In thông tin về mỗi constructor

for (Constructor constructor : constructors) {  
 System.*out*.println("Constructor: " + constructor);  
 **Class[] parameterTypes = constructor.getParameterTypes();**  
 for(Class parameterType : parameterTypes){  
 System.*out*.println(parameterType.getName());  
 }  
}

Output:

*Constructor: public Demo(int,java.lang.String)*

*int*

*java.lang.String*

Tạo một đối tượng của một Lớp từ đối tượng **Contructor** với phương thức **newInstance()** của lớp **Constructor**. Phương thức **newInstance**() trả lại kiểu dữ liệu **Object**

import **java.lang.reflect.Constructor**;  
class Demo {  
 private int a;  
 private String b;  
 public Demo(int a, String b) {  
 this.a = a;  
 this.b = b;  
 }  
 public int getA() {  
 return a;  
 }  
 public String getB() {  
 return b;  
 }  
}  
public class test {  
 public static void main(String[] args) {  
 try {  
 // Lấy đối tượng Class cho lớp Demo  
 Class demoClass = Demo.class;  
 // Lấy constructor với kiểu dữ liệu tham số int và String  
 Constructor constructor = demoClass.getConstructor(int.class, String.class);  
 // Tạo đối tượng Demo từ constructor; Ép kiểu dữ liệu từ Object -> Demo  
 Demo demoObject = (Demo) constructor.newInstance(2003, "Nghia");  
 // Sử dụng printf với định dạng %s và %s  
 System.*out*.printf("%s %s", demoObject.getA(), demoObject.getB());  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

Output: *2003 Nghia*

1. **Fields**

Field là lớp các đối tượng đại diện cho một thuộc tính của một lớp. Bạn có thể duyệt và lấy ra các đối tượng này từ class object. Sử dụng các phương thức:

* **getField(***String fieldName***) 🡪** trả lại kiểu dữ liệu **Field** có modify là **public**; thuộc lớp **Class**
* **getDeclaredField(***String fieldName***) 🡪** trả lại kiểu dữ liệu **Field** với mọi modifies; thuộc lớp **Class**
* **setAccessible**(*boolean value*) 🡪 không trả lại kiểu dữ liệu; thuộc lớp **Field**
* **get(Object obj)** 🡪 trả lại kiểu dữ liệu **Object**; thuộc lớp **Field**
* **set(Object obj, Object value)** 🡪 không trả lại kiểu dữ liệu; thuộc lớp **Field**

import **java.lang.reflect.Field**;  
class Demo {  
 private String a;  
 public Demo(String a) {  
 this.a = a;  
 }  
 public String getA() {  
 return a;  
 }  
}  
public class test {  
 public static void main(String[] args) {  
 try {  
 // Lấy đối tượng Class cho lớp Demo  
 Class demoClass = Demo.class;  
 // Tạo đối tượng Demo với giá trị cho trường a  
 Demo objectDemo = new Demo("Hello");  
 // Lấy trường private a của lớp Demo  
 **Field** field = demoClass.**getDeclaredField**("a");  
 // Cho phép truy cập vào trường private a  
 field.**setAccessible**(true);  
 // Lấy giá trị của trường a từ đối tượng Demo  
 Object value = field.**get**(objectDemo);  
 System.*out*.println("Before setting: " + value);  
 // Thiết lập giá trị mới cho trường a  
 field.**set**(objectDemo, "World");  
 // In giá trị của trường a sau khi thiết lập  
 System.*out*.println("After setting: " + objectDemo.getA());  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

Output:

*Before setting: Hello*

*After setting: World*

1. **Methods**

**Method getMethod (** String*nameMethod, parameterTypes***) –** phương thức thuộc lớp Class

**Method getDeclaredMethod (** String*nameMethod***,** *parameterTypes***) –** phương thức thuộc lớp Class

**void invoke (***Object, parameter***) –** phương thức thuộc lớp Method

**import java.lang.reflect.Method;**

class A {

public void AMethod(String message) {

System.out.println("MyMethod called with message: " + message);

}

}

public class test {

public static void main(String[] args) {

try {

// Lấy đối tượng Class cho lớp A

Class AClass = A.class;

// Lấy đối tượng Method của phương thức AMethod với kiểu dữ liệu của tham số là String

**Method** method = AClass.**getMethod**("AMethod", String.class);

// Tạo một đối tượng A

A myObject = new A();

// Gọi phương thức myMethod trên đối tượng myObject với đối số là "Hello, Reflection!"

method.**invoke**(myObject, "Hello, Reflection!");

} catch (Exception e) {

e.printStackTrace(); } } }

1. **Annotations**

[Annotation, sử dụng annotation với reflection trong java (viblo.asia)](https://viblo.asia/p/annotation-su-dung-annotation-voi-reflection-trong-java-PDOkqXxyejx)

1. **Định nghĩa**

**Annotations** là một dạng chú thích hoặc meta data được chèn vào code java để đặc tả dữ liệu cho một đối tượng, giá trị nào đó. Nó có thể được xử lý tại thời điểm compile hoặc tại thời điểm runtime. Ví dụ:

**@MyAnnotation**(*value = "Hello World")*

public class TheClass {

}

**Annotations** có thể chú thích cho class, method, parameter hoặc field.

1. **Cách tạo Annotation**

Định nghĩa Annotation giống như định nghĩa 1 interface. Ví dụ:

**@Retention(RetentionPolicy.RUNTIME)**

**@Target(ElementType.TYPE)**

public **@**interface MyAnnotation {

public String value() default "No description";

}

* **"@"** được khai báo trước interface để đánh dấu đó là một annotation.
* [**@Retention**](https://viblo.asia/u/Retention)**(RetentionPolicy.RUNTIME)** và [**@Target**](https://viblo.asia/u/Target)**(ElementType.TYPE)** xác định cách thức và thời điểm annotation được sử dụng
  + [**@Retention**](https://viblo.asia/u/Retention)**(RetentionPolicy.RUNTIME)**: cho biết annotation có thể được truy cập qua reflection tại thời điểm runtime, nếu bạn không khai báo nó khi định nghĩa một annotation bạn sẽ không thể truy cập nó tại thời điểm runtime.
  + [**@Target**](https://viblo.asia/u/Target)**(ElementType.TYPE):** cho biết việc bạn chỉ có thể sử dụng annotation với class interface và enum.

1. **Truy cập annotation với reflection.**
   1. **Class Annotations**

[**@Target**](https://viblo.asia/u/Target)**(ElementType.TYPE)**

**import java.lang.annotation.Annotation;**

@MyAnnotation(value = "This is A class")

class A{ }

public class test{

public static void main (String args []){

Class myClass = A.class;

**Annotation[]** annotations = myClass.**getAnnotations**();

for (**Annotation** annotation : annotations){

**MyAnnotation** myAnnotation = (**MyAnnotation)** annotation;

System.out.println("value = " + myAnnotation.**value**());

*//* ***Annotation*** *annotation = myClass.****getAnnotation****(MyAnnotation.class);  
//* ***MyAnnotation*** *myAnnotation = (****MyAnnotation****) annotation;  
// System.out.println("value = " + myAnnotation.****value****());*

}

}

}

* 1. **Method Annotations**

**@Target(ElementType.METHOD)**

import java.lang.annotation.Annotation;

import java.lang.reflect.Method;

class A{

**@MyAnnotation**(value = "This is a method")

public void doSomething(){}

}

public class test{

public static void main (String args []) throws NoSuchMethodException {

Class myClass = A.class;

Method myMethod = myClass.getMethod("doSomething");

**Annotation[]** annotations = myMethod.**getAnnotations();**

for( Annotation annotation : annotations){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("value = " + myAnnotation.value());

// ***Annotation*** *annotation = myMethod.****getAnnotation****(MyAnnotation.class);  
//* ***MyAnnotation*** *myAnnotation = (****MyAnnotation****) annotation;  
// System.out.println("value = " + myAnnotation.****value****());*

}

}}

* 1. **Parameter Annotations**

**@Target(ElementType.PARAMETER)**

import java.lang.annotation.Annotation;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

class A{

public void doSothing(**@MyAnnotation(value = "This is a field")** String parameter){}

}

public class test{

public static void main (String args []) throws NoSuchFieldException, NoSuchMethodException {

Class myClass = A.class;

Method method = myClass.getMethod("doSothing", String.class);

**Annotation[][]** parameterAnnotations = method.**getParameterAnnotations()**;

*//Trong 1 method có nhiều tham số, với mỗi tham số có thể có nhiều annotations nên dùng mảng 2 chiều*

for (**Annotation []** parameterAnnotation : parameterAnnotations){

for (**Annotation** annotation : parameterAnnotation){

**MyAnnotation** myAnnotation = (**MyAnnotation**) annotation;

System.out.println("value = " + myAnnotation.**value**());

}

}

}

}

* 1. **Field Annotations**

**@Target(ElementType.*FIELD*)**

import java.lang.annotation.Annotation;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

class A{

**@MyAnnotation**(value = "This is a field")

public int a;

}

public class test{

public static void main (String args []) throws NoSuchFieldException {

Class myClass = A.class;

Field field = myClass.getField("a");

**Annotation** annotation = field.**getAnnotation**(MyAnnotation.class);

**MyAnnotation** myAnnotation = (**MyAnnotation**) annotation;

System.out.println("value = " + myAnnotation.**value**());

}

}

Đối với annotations của method và parameters, ta phải “get” thông qua method object; còn lại thông qua class object.