**HỌC JAVA CƠ BẢN VÀ NÂNG CAO**

*Java là ngôn ngữ lập trình máy tính có tính chất hướng đối tượng, dựa trên các lớp, thường được sữ dụng cho các hệ thống có tính độc lập cao, hỗ trợ chạy trên các nền tảng hỗ trợ java mà không cần phải biên dịch lại*

*- Các ứng dụng sao khi được biên dịch thành ngôn ngữ bytecode có thể chạy trên bất kì máy ảo java nào*

*Bytecode được lưu ở dạng mã nhị phân, các lệnh, số, hằng, và tham chiếu(thường là địa chỉ bằng số)\**

## Java là gì?

Java là một ngôn ngữ lập trình và là một Platforrm

**Ngôn ngữ lập trình:** Java là một ngôn ngữ lập trình có tính bảo mật cao, hướng đối tượng, bậc cao và mạnh mẽ

**Platform:**Bất cứ môi trường phần cứng hoặc phần mền nào mà trong đó một chương trình chạy, thì được biết đến như là một Platform. Với môi trường runtime riêng cho mình là JRE và API, Java được gọi là Platform.

## Ví dụ về Java

Bạn theo dõi ví dụ đơn giản sau để in Hello World, phần giải thích chi tiết sẽ được trình bày trong chương tiếp theo.

class Simple{

public static void main(String args[]){

System.out.println("Hello World");

} }

Từ khóa **class** được sử dụng để khai báo một lớp trong Java.

Từ khóa **public** là một Access Modifier mà biểu diễn tính nhìn thấy, có nghĩa rằng nó là nhìn nhất với tất cả

Từ khóa **static** là một từ khóa mà nếu chúng ta khai báo bất cứ phương thức nào là **static** thì nó còn được gọi là phương thức tĩnh hoặc phương thức **static**. Lợi thế chủ yếu của phương thức static là không cần thiết tạo đối tượng để triệu hồi phương thức static. Phương thức main được thực thi bởi JVM, vì thế không cần thiết tạo một đối tượng để gọi phương thức main. Việc này giúp tiết kiệm bộ nhớ.

**void** là kiểu trả về của phương thức, nghĩa là phương thức không trả về bất cứ giá trị nào.

**main** tượng trưng cho khởi động chương trình.

**String[] args** được sử dụng như là lệnh in.

## Đơn giản

- Java được phát triển dựa trên ngôn ngữ C++

- Hạn chế những phương thức ít sử dụng như, tường minh, nạp chồng toán tử

## Hướng đối tượng

Có thể hiểu là chúng ta tổ chức phần mềm dưới dạng một sự kết hợp của nhiều loại đối tượng khác nhau mà kết hợp chặt chẽ cả về dữ liệu lẫn hành vi của chúng.

Lập trình hướng đối tượng (OOP) là một phương pháp giúp đơn giản hóa việc phát triển và duy trì phần mềm bằng việc cung cấp một số qui tắc.

**Lập trình Hướng đối tượng (OOP) gồm 4 tính chất:**

* Tính kế thừa
* Tính đa hình
* Tính trừu tượng
* Tính đóng gói

## Độc lập nần tảng

Một Platform là môi trường phần cứng hoặc phần mềm. Trong đó có 2 loại Platform: một loại dựa trên phần mềm ( software-based) và một loại dựa trên phần cứng (hardware-based)

Java cung cấp software-based platform, Java Platform khác với nhiều nền tảng khác ở chổ nó chạy được trên các nền tảng hardware-based khác, nó có 2 thành phần

* Runtime Enviroment
* API (Application Programming Interface)

Java code có thể chạy được trên nhiều nền tảng khác nhau như Windows, Linux, Mac/OS,.. được biên dịch bởi **bộ biên dịch** **Compiler và** chuyển đổi thành dạng Bytecode.

Bytecode là một code độc lập nền tảng bởi vì nó có thể chạy trên nhiều nền tảng khác nhau.

## Bảo mật

Java an toàn bởi vì:

* Không có con trỏ tường minh
* Chương trình chạy bên trong các hộp thiết bị ảo

**Classloader:** Thêm sự bảo vệ bằng việc phân biệt riêng lẽ các package cho các lớp của hệ thống local file mà từ đó import với các file từ nguồn mạng.

**Bytecode Vertifier:** Kiểm tra các đoạn code để tìm ra phần code không hợp lệ mà có thể truy cập trái phép tới các đối tượng

**Security Manager:** Quyết định xem nguồn resource nào là một lớp có thể truy cập, chẳng hạn như đọc và ghi tới local disk

\*Những tính năng này được cung cấp bởi ngôn ngữ Java

## Đa luồng (Multi-thread)

Một thread(chủ đề) giống một chương trình riêng lẽ, thực thi một cách đồng thời. Chúng ta có thể viết các chương trình Java mà xử lý nhiều tác vụ cùng lúc bằng việc định nghĩa nhiều Thread. Lợi thế chính của Multi-Thread là nó chía sẽ vùng bộ nhớ. Các Thread là trọng cho Multi-Media, Web App,..

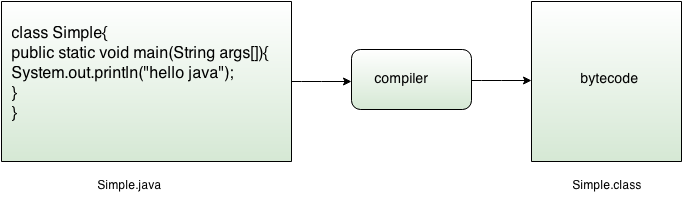
## Các Java Editor phổ biến:

Để viết các chương trình chạy bằng Java, bạn sẽ cần một Text Editor. Có rất nhiều loại, và bạn nên sử dụng một trong các loại sau:

* **Notepad:** Trên thiết bị Windown bạn có thể sử dụng trình soạn thảo text đơn giản như Notepad.
* **Netbeans:** Là một chương trình IDE mã nguồn mở miễn phí, ban có thể Download trên web
* **Eclipse:** Là một trình IDE được phát triển bởi cộng đồng Eclipse, có thể download trên web

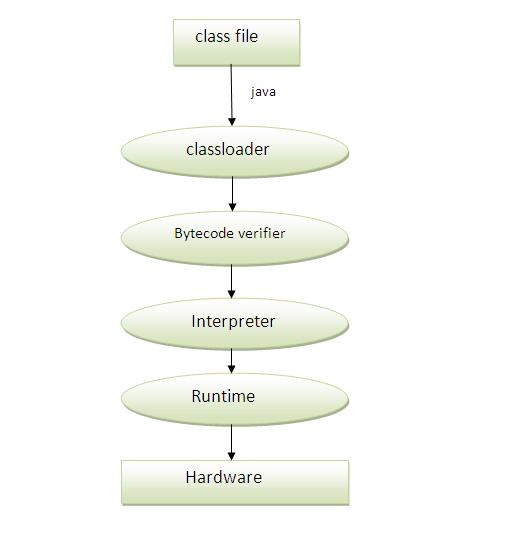
## Những gì sảy ra tại Compile time:

Tại compiler time (Thời gian biên dịch), java file được biên dịch bởi Compiler (Nó không tương tác với OS) và chuyển đổi java code thành dạng Bytecode.



## Điều gì sảy ra tại Runtime:

Tại Runtime (Thời gian biên dịch), gồm có 5 bước thực hiện như sau:



**Classloader:** là hệ thống con của JVM được sử dụng để tải class file.

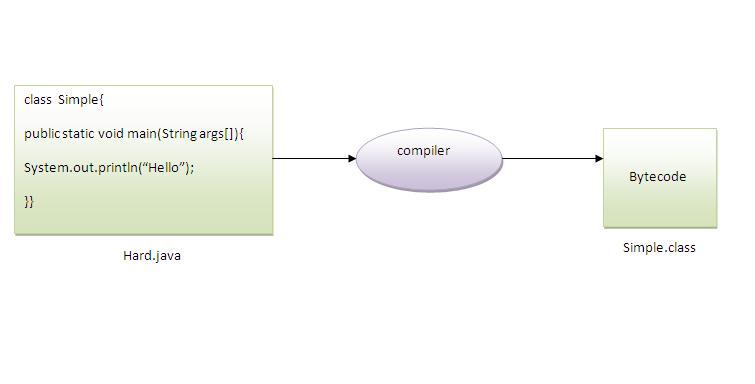
**Bytecode Vertifier:** Kiểm tra các đoạn code để xem có hay không các phần code không hợp lệ có truy cập không hợp lệ tới các đối tượng.

**Interpreter:** Đọc Bytecode Stream, sau đó thực thi các chỉ thị.

### **Câu hỏi:**

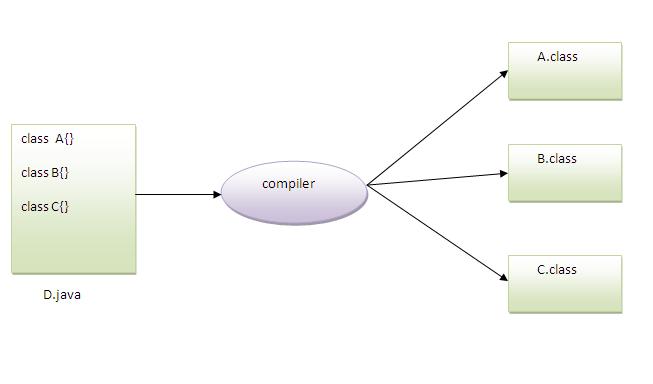
### Bạn có thể lưu một source file bởi một tên khác ngoài tên lớp không?

Có, nếu lớp không phải là public. Bạn có thể theo dõi ở sơ đồ dưới:



### Bạn có thể có nhiều lớp trong một source file không?

Có, sơ đồ sau minh họa điều này:



*Chúng ta có thể coi chương trình Java như một tập hợp các đối tượng mà có thể trao đổi lẫn nhau dùng các phương thức. Dưới đây chúng ta sẽ nêu ra một số định nghĩa cơ bản của lớp, đối tượng, phương thức cũng như biến trong Java:*

* **Đối tượng -** Đối tượng có các trạng thái và hành vi. Ví dụ: Một con chó có các trạng thái của màu da, tên tuổi, thức ăn cũng như các hành vi như sủa, ăn, vẫy đuôi.
* **Lớp -** Một lớp có thể được định nghĩa như một bản thiết kế, mẫu mà có thể mô tả các trạng thái, hành vi của một đối tượng mà nó hỗ trợ.
* **Phương thức -** Một phương thức đơn giản là một hành vi. Một lớp có thể bao gồm nhiều phương thức. Trong mỗi phương thức có những phép toán logic, dữ liệu được xử lý và tất cả các hành động được thực thi.
* **Biến -** Mỗi đối tượng có một tập các biến duy nhất. Mỗi trạng thái của đối tượng được khởi tạo bởi các giá trị và gán với những biến.

## Biến trong Java:

* Các loại biến trong java như sau:
* Biến Local
* Biến của class (Biến static)
* Biến đối tượng (Không phải biến static)

## Tính kế thừa trong Java:

Trong java, các Class có thể được suy ra từ các Class khác. Về cơ bản, nếu cần tạo một class mới và ở đây đã là một Class mà có một số đoạn code bạn cần, thì khi đó nó có thể suy ra từ một class mới từ code tồn tại.

Trong tình huống này, class đang tồi tại được gọi là supperclass và class được kế thừa được gọi là subclass.

## Interface trong Java:

Trong ngôn ngữ Java, một interface có thể được định nghĩ như là một contract giữa các đối tượng về cách giao tiếp với nhau. Các interface đóng vai trò thiết yếu khi nó đi với khái niệm về tính kế thừa.

Một interface định nghĩa các phương thức, các subclass nên sử dụng. Nhưng thực thi của các phương thức lại hoàn toàn do các subclass.

*“Hiểu rõ sự khác nhau giữa JDK, JRE và JVM là điều khá quan trọng trong Java. Ở đây, chúng tôi trình bày miêu tả ngắn gọn về JVM, để biết thêm chi tiết về nó, bạn truy cập vào chương tiếp theo. Đầu tiên chúng ta tìm hiểu sự khác nhau cơ bản giữa JDK, JRE và JVM.”*

## Sơ lược về JVM:

JVM (viết tắt của Java Virtual Machine) là một thiết bị trừu tượng(ảo) có thể giúp máy tính chạy các chương trình Java. Nó cung cấp môi trường Runtime mà trong đó Java Bytecode có thể thực thi.

JVM là có sẵn cho nhiều nền tảng (Windowns, Linux, Mac/OS,…). JVM, JRE và JDK phụ thuộc nền tảng, bởi vì cấu hình của mỗi OS(Hệ điều hành) là khác nhau. Nhưng Java là độc lập nền tảng.

## Giới thiệu JRE:

JRE (là viết tắt của Java Runtime Enviroment) được sử dụng để cung cấp môi trường runtime. Nó là trình triển khai của JVM. JRE bao gồm tập hợp các thư viện và các file khác mà JVM sử dụng tại runtime.

## Sơ lược về JDK:

JDK (là viết tắt của Java Developer Kit) bao gồm JRE và các Developer Tool.

## Biến và các ví dụ minh họa:

Biến là tên một khu nhớ. Có 3 kiểu biến: Biến cục bộ (Còn gọi là biến local), biến thuộc tính(biến của instance - đối tượng) và biến static.

Ví dụ về cách khai báo và khởi tạo biến hợp lệ trong Java:

int a, b, c; // Khai bao ba bien kieu int la a, b, và c.

int a = 5, b = 7; // Vi du ve khoi tao bien

byte A = 11; // Khoi tao mot bien kieu byte ten la A.

double pi = 3.14159; // Khai bao va gan mot gia tri cua PI.

char nam = 'b'; // Bien nam duoc khoi tao voi gia tri 'a'.

## Biến local trong Java:

Các biến được khai báo trong các phương thức, constructor, hoặc khối.

Các biến được tạo khi phương thức, constructor hoặc khối được nhập và biến bị hủy khi phương thức, constructor hoặc khối kết thúc.

Chỉ định truy cập (access modifier) có thể không được sử dụng cho các biến local.

Các biến local chỉ được nhìn thấy trong phương thức, constructor, hoặc khối được khai báo.

Các biết locla được thực thi nội bộ.

Không có giá trị mặc định nào cho biến local, vì thế các biến local nên được khai báo và một giá trị khởi tạo nên được gán trước khi sử dụng.

## Ví dụ:

Ở đây, *tuoi* là biến local. Nó được định nghĩa bên trong phương thức *tuoiCon()* và phạm vi của nó được giới hạn chỉ trong phương thức này.

public class Test{

public void tuoiCon(){

int tuoi = 0;

tuoi = tuoi + 10;

System.out.println("Tuoi con la : " + tuoi);

}

public static void main(String args[]){

Test test = new Test();

test.tuoiCon();

}}

## Ví dụ:

Ví dụ sau sử dụng *tuoi* mà không khởi tạo nó, vì thế nó sẽ tạo một lỗi tại thời gian biên dịch.

public class Test{

public void tuoiCon(){

int tuoi;

tuoi = tuoi + 10;

System.out.println("Tuoi con la : " + tuoi);

}

public static void main(String args[]){

Test test = new Test();

test.tuoiCon();

}}

Nó sẽ cho một lỗi sau trong khi biên dịch:

Test.java:4:variable number might not have been initialized

tuoi = tuoi + 10;

^1 error

### Đối tượng (Object)

Tất cả những thực thể có trạng thái và hành vi được biết đến như là một đối tượng. Ví dụ: bàn, ghế, bút chì, xe dạp, ô tô...

### Lớp (Class)

Tập hợp các đối tượng được gọi là lớp.

### Kế thừa (Inheritance *sub in he ri tờn*)

Khi một đối tượng con được truyền lại tất cả các thuộc tính và phương thức của đối tượng cha được gọi là kế thừa. Kế thừa giúp tái sử dụng lại mã nguồn. Nó được sử dụng cho đa hình lúc runtime.

Tìm hiểu thêm về [Tính kế thừa (Inheritance) trong java](https://viettuts.vn/java/tinh-ke-thua-trong-java)

### Đa hình (Polymorphism *sub pô ly mọt phim*)

Khi một nhiệm vụ được thực hiện bởi nhiều cách khác nhau, tính chất này được gọi là đa hình. Ví dụ có nhiều cách để thuyết phục các khách hàng khác nhau, để vẽ một cái gì đó như hình tròn, hình chữ nhật, ...

Trong java, để áp dụng tính đa hình chúng ta sử dụng phương thức orverloading hoặc overriding.

Tìm hiểu thêm về [Tính đa hình (Polymorphism) trong java](https://viettuts.vn/java/tinh-da-hinh-trong-java)

### Trừu tượng (Abstraction *sub áp trắc sờn*)

Trừu tượng là sự ẩn đi những chi tiết bên trong và hiển thị ra các chức năng, tính chất này gọi là trừu tượng. Ví dụ: khi gọi điện thoại chúng ta không biết xử lý nội bộ thế nào, khi đi xe máy cũng vậy, mà chúng ta chỉ biết đến các chức năng thông qua giao tiếp bên ngoài.

Trong java, chúng ta áp dụng tính chất trừu tượng bằng cách sử dụng abstract class và interface.

Tìm hiểu thêm về [lớp abstract trong java](https://viettuts.vn/java/abstract-class-trong-java) và [interface trong java](https://viettuts.vn/java/interface-trong-java).

### Đóng gói (Encapsulation *sub en cáp su lay sờn*)

Việc ràng buộc giữa code và data với nhau tạo thành một khối duy nhất được biết đến là đóng gói. Ví dụ: viên thuốc con nhộng được đóng gói với nhiều loại thuốc bên trong.

Một class trong java là một ví dụ về đóng gói. Java bean là một lớp được đóng gói hoàn toàn vì tất cả các dữ liệu thành viên là private.

Tìm hiểu thêm về [Tính đóng gói (Encapsulation) trong java.](https://viettuts.vn/java/tinh-dong-goi-trong-java)

**Spring boot**

*“Spring Boot là một bước tiếp theo của Spring, để làm cho Spring dễ dàng hơn trong việc thiết lập và phát triển ứng dụng. Với Spring Boot các cấu hình Spring được giảm thiểu tối đa. Spring Boot hỗ trợ các bộ chứa nhúng điều này cho phép các ứng dụng web có thể chạy độc lập mà không cần phải triển khai trên các Web Server”*

**Lợi ích của Spring Boot Framework:**

- Giúp cho việc phát triển các ứng dụng dựa trên Spring va Java được dễ đàng hơn

- Giúp cho việc tương tác các ứng dụng Spring Boot với các hệ sinh thái của Spring như Spring JDBC, Spring Data, Spring Security được dễ dàng hơn.

- Cung cấp các Server nhúng như Tomcat, Jetty,.. để phát triển và test các ứng dụng trên web được nhanh chóng và dễ dàng.

- Cung cấp nhiều plugin để làm việc với các cơ sở dữ liệu nhúng và các cơ sở dữ liệu lưu trữ trên bộ nhớ một cách dễ dàng

**Mô hình Spring MVC Framework:**

**Spring MVC Framework** là một project mã nguồn mở của Spring

**Spring MVC Framework** cung cấp kiến trúc MVC ( Model-View-Controller)

Trong mô hình MVC thì:

**Model:** là các file POJO, Service, DAO thực hiện truy cập database, xử lí business.

**View:** là các file JSP, html,…

**Control:** là Dispathcher Controller, Handler Mapping, Controller thực hiện điều hướng các request.

## ****Dưới đây là một số Annotation trong Spring thường dùng đi kèm với các ví dụ cụ thể****

### ****@Autowired****

****@Autowired****: Annotation này trong spring giúp tự động liên kết các ****Bean****lại với nhau, đây là một trong những đặc điểm nỗi bật của Spring để thể hiện tính chất [DI và IOC (Dependency Injection và Inversion of Control)](https://itphutran.com/spring-framework/dependency-injection-va-inversion-control-trong-spring-framework).

Ví dụ về annotation ****@Autowired:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class FooService {      private FooFormatter fooFormatter;      @Autowired      public FooService(FooFormatter fooFormatter) {          this.fooFormatter = fooFormatter;      }  } |

### ****@Scope****

****@Scope****: đây là môt Annotation mặc định trong Spring, hầu hết ****scope**** phổ biến cho các autodetected components là ****singleton**** (singleton: Với mỗi bean container chỉ tao duy nhất một đối tượng), để thay đổi phạm vi ta sử dụng @Scope.

Ví dụ về annotation ****@Scope:****

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Bean  @Scope("singleton")  public Person personSingleton() {      return new Person();  } |

### ****@Component****

****@Component****: dùng để scan các components.

Ví dụ về annotation ****@Component:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Component("fooFormatter")  public class FooFormatter {      public String format() {          return "foo";      }  }  @Component  public class FooService {        @Autowired      private FooFormatter fooFormatter;  } |

### ****@Repository****

****@Repository****:  dùng để scan các components để đánh dấu các lớp DAO/Repository của Spring.

Ví dụ về annotation ****@Repository:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public interface EmployeeDAO  {      public EmployeeDTO createNewEmployee();  }  @Repository ("employeeDao")  public class EmployeeDAOImpl implements EmployeeDAO  {      public EmployeeDTO createNewEmployee()      {          EmployeeDTO e = new EmployeeDTO();          e.setId(1);          e.setFirstName("Lokesh");          e.setLastName("Gupta");          return e;      }  } |

### ****@Service****

****@Service****: dùng để scan các components để đánh dấu các Service của lớp business.

Ví dụ về annotation ****@Service:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public interface EmployeeManager  {      public EmployeeDTO createNewEmployee();  }  @Service ("employeeManager")  public class EmployeeManagerImpl implements EmployeeManager  {      @Autowired      EmployeeDAO dao;        public EmployeeDTO createNewEmployee()      {          return dao.createNewEmployee();      }  } |

### ****@Transactional****

****@Transactional****: Annotation này trong spring giúp đánh dấu các class có sử dụng đến Transaction do Spring quản lý.

Ví dụ về annotation ****@Transactional:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class CustomerManagerImpl implements CustomerManager {    private CustomerDAO customerDAO;    public void setCustomerDAO(CustomerDAO customerDAO) {  this.customerDAO = customerDAO;  }    @Override  @Transactional  public void createCustomer(Customer cust) {  customerDAO.create(cust);  }    } |

### ****@Controller****

****@Controller****: đây là môt Annotation trong spring mvc dùng để đánh dấu đây là lớp có chức năng là Controller.

Ví dụ về annotation ****@Controller:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Controller ("employeeController")  public class EmployeeController  {          @Autowired      EmployeeManager manager;        public EmployeeDTO createNewEmployee()      {          return manager.createNewEmployee();      }  } |

### ****@RequestMapping****

****@RequestMapping****: đây là môt Annotation trong spring mvc dùng để cấu hình URL tương ứng với @WebServlet và mapping trong file xml với [JSP/servlet](https://itphutran.com/category/lap-trinh-java/jsp-servlet).

Ví dụ về annotation ****@RequestMapping:****

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value="/method0")  @ResponseBody  public String method0(){  return "method0";  } |

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value={"/method1","/method1/second"})  @ResponseBody  public String method1(){  return "method1";  } |

### ****@RequestParam****

****@RequestParam****: đây là môt Annotation trong spring mvc dùng để gán một request parameter vào một parameter của phương thức.Thường dùng để lấy dữ liệu từ form và các biến tham số trên URL.

Ví dụ về annotation ****@RequestParam:****

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value="/method9")  @ResponseBody  public String method9(@RequestParam("id") int id){  return "method9 with id= "+id;  } |

### ****@InitBinder****

****@InitBinder****: đây là môt Annotation trong spring mvc dùng để đánh dấu một phương thức có tùy biến các ràng buộc dữ liệu.

Ví dụ về annotation ****@InitBinder:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @InitBinder  public void dataBinding(WebDataBinder binder) {  binder.addValidators(userValidator, emailValidator);  SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");  dateFormat.setLenient(false);  binder.registerCustomEditor(Date.class, "dob", new CustomDateEditor(dateFormat, true));  } |

### ****@SessionAttribute****

****@SessionAttribute****: đây là môt Annotation trong spring mvc dùng để chỉ định một modelAttribute sẽ được lưu trong session.

Ví dụ về annotation ****@SessionAttribute:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @SessionAttribute("myModel")  @Controller  public class MyController {      @RequestMapping(...)      public String displayForm(@RequestParam("id") long id, ModelMap model) {          MyModel m = findById(id);          model.put("myModel", m); // Initialized          return ...;      }        @RequestMapping(...)      public String submitForm(@ModelAttribute("myModel") @Valid MyModel m,          BindingResult errors, SessionStatus status) {          if (errors.hasErrors()) {              // Will render a view with updated MyModel              return ...;          } else {              status.setComplete(); // MyModel is removed from the session              save(m);              return ...;          }        }  } |

### ****@Valid****

****@Valid****: đây là môt Annotation trong spring mvc dùng để ràng buộc đối tượng hay các Param bị ràng buộc để thực hiện kiểm tra tính hợp lệ cho form.

Ví dụ về annotation ****@Valid:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @Controller  @RequestMapping("/customer")  public class SignUpController {    @RequestMapping(value = "/signup", method = RequestMethod.POST)  public String addCustomer(@Valid Customer customer, BindingResult result) {    if (result.hasErrors()) {  return "SignUpForm";  } else {  return "Done";  }    }    @RequestMapping(method = RequestMethod.GET)  public String displayCustomerForm(ModelMap model) {    model.addAttribute("customer", new Customer());  return "SignUpForm";    }    } |

### ****@Configuration****

****@Configuration** là một **annotation** đánh dấu trên một class. Cho phép **Spring Boot** biết được đây là nơiđịnh nghĩa một lớp **@Bean**.**

**@Bean là một Annotation được đánh dấu trên các method cho phép Spring Boot biết được đây là Bean và sẽ thực hiện đưa Bean này vào Context**

### ****@PathVariable****

****@PathVariable:****đây là môt Annotation dùng để trích xuất cách phần mẫu trên URL, được đại diện bởi biến id.

Ví dụ về annotation ****@PathVariable****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @SessionAttribute("myModel")  @Controller  public class MyController {        @Autowired       ProductService proService;      }        @GetMapping(“/rest/product/{id}”)      public Product create(@PathVariable("id") Integer id) {              return proService.save(id);          }    @GetMapping(“/rest/product/{id}”)      public void delete(@PathVariable("id") Integer id) {              proService.deleteById(id);      }  } |
|  |  |

### ****@ModelAttribute****

****@ModelAttribute:****đây là môt Annotation trong spring mvc dùng để lấy các tham số và trả về một đối tượng (Object).

Ví dụ về annotation ****@ModelAttribute:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | @Controller  @ControllerAdvice  public class EmployeeController {      private Map<Long, Employee> employeeMap = new HashMap<>();      @RequestMapping(value = "/addEmployee", method = RequestMethod.POST)      public String submit(        @ModelAttribute("employee") Employee employee,        BindingResult result, ModelMap model) {          if (result.hasErrors()) {              return "error";          }          model.addAttribute("name", employee.getName());          model.addAttribute("id", employee.getId());          employeeMap.put(employee.getId(), employee);          return "employeeView";      }      @ModelAttribute      public void addAttributes(Model model) {          model.addAttribute("msg", "Welcome to the Netherlands!");      }  } |

## **@Entity**

@Entity được sử dụng để chú thích một class là một Entity.

Thuộc tính name của @Entity là không bắt buộc. Việc chỉ định rõ name của @Entity cho phép viết ngắn câu HSQL.

Entity khớp với một bảng lấy theo tên theo thứ tự ưu tiên:

1. name trong @Table.
2. name trong @Entity.
3. name của class.

## **@Table**

Một table trong database có thể có nhiều ràng buộc unique (duy nhất). Chúng ta có thể sử dụng @Table để mô tả các ràng buộc này.

@Table cho phép chú thích tên bảng thông qua thuộc tính name (thuộc tính này không bắt buộc).

Nếu không chỉ rõ tên bảng trong phần tử name, Hibernate sẽ dựa vào phần tử name của @Entity sau đó mới tới tên của class.

## **@Column**

@Column được sử dụng để chỉ định thông tin chi tiết của cột mà một field của entity sẽ được ánh xạ với một column trong database.

* Thuộc tính ****name****được sử dụng để chị định tên cột nào trong database map với tên field được chú thích. Nếu không chỉ định, Hibernate sẽ lấy tên field map với tên cột trong database.
* Thuộc tính ****length**** cho phép kích thước của cột. @Column không chỉ rõ phần tử length, mặc định nó là ****255****.
* Thuộc tính ****nullable**** cho phép cột được đánh dấu KHÔNG ****NULL**** khi schema được tạo ra. Giá trị ****nullable**** mặc định là ****true****.
* Thuộc tính ****unique****cho phép cột được đánh dấu chỉ chứa các giá trị duy nhất.

Chúng ta có thể không cần chỉ định @Column cho tất cả các field nếu tên field và tên column và tên trong database là giống nhau.

## **@Transient**

Trong một số trường hợp, trong entity class có chứa một field mà field này không tồn tồn tại trong database. Khi đó chúng ta sẽ gặp lỗi “java.sql.SQLSyntaxErrorException: Unknown column ‘additionalPropery’ in ‘field list'”.

Để tránh lỗi này, chúng ta có thể sử dụng @Transient để thông báo rằng thuộc tính/ phương thức này không liên quan gì tới một cột nào dưới database. Khi đó, Hibernate sẽ bỏ qua field này.

## **@Temporal**

@Temporal sử dụng để chú thích cho cột dữ liệu ngày tháng và thời gian (date time).

Có 3 giá trị cho TemporalType:

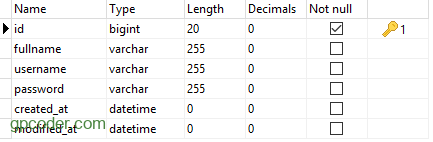
* TemporalType.DATE : chú thích cột sẽ lưu trữ ngày tháng năm (bỏ đi thời gian).
* TemporalType.TIME : chú thích cột sẽ lưu trữ thời gian (Giờ phút giây).
* TemporalType.TIMESTAMP : chú thích cột sẽ lưu trữ ngày tháng và cả thời gian.

## **@Id**

@Id được sử dụng để mô tả đây là Id (Identity) của Entity, nó tương đương với cột đó là khóa chính (Primary Key) của table trong database.

Khóa chính có thể là một trường duy nhất hoặc kết hợp nhiều trường tùy thuộc vào cấu trúc bảng trong database.

Ví dụ:

[](https://gpcoder.com/wp-content/uploads/2019/10/hiberante-example-user-table.png)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **CREATE** **TABLE** `user` (  `id` **bigint**(20) unsigned NOT NULL AUTO\_INCREMENT,  `fullname` **varchar**(255) **COLLATE** utf8mb4\_unicode\_ci **DEFAULT** NULL,  `username` **varchar**(255) **CHARACTER** **SET** utf8mb4 **DEFAULT** '',  `**password**` **varchar**(255) **COLLATE** utf8mb4\_unicode\_ci **DEFAULT** NULL,  `created\_at` datetime **DEFAULT** NULL **ON** **UPDATE** current\_timestamp(),  `modified\_at` datetime **DEFAULT** NULL **ON** **UPDATE** current\_timestamp(),  **PRIMARY** **KEY** (`id`),  **UNIQUE** **KEY** `unique\_user\_username` (`username`)  ) ENGINE=InnoDB AUTO\_INCREMENT=3 **DEFAULT** CHARSET=utf8mb4 **COLLATE**=utf8mb4\_unicode\_ci; |

Ta có thể tạo Entity tương ứng với database trên như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | **package** com.gpcoder.entities;    **import** java.util.Date;    **import** javax.persistence.Column;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  **import** javax.persistence.Table;  **import** javax.persistence.Temporal;  **import** javax.persistence.TemporalType;  **import** javax.persistence.Transient;    **import** lombok.Data;    @Data  @Entity(name = "User")  @Table(name = "user")  **public** **class** User {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        @Column  **private** String fullname;        @Column(nullable = **false**, length = 255, unique = **true**)  **private** String username;        @Column(nullable = **false**)  **private** String password;        @Column(name = "created\_at")      @Temporal(value = TemporalType.TIMESTAMP)  **private** Date createdAt;        @Column(name = "modified\_at")      @Temporal(value = TemporalType.TIMESTAMP)  **private** Date modifiedAt;        @Transient  **private** String additionalPropery;  } |

## **@GeneratedValue**

@GeneratedValue được sử dụng để Hibernate tự động tạo ra giá trị và gán vào cho một cột trong trường hợp insert mới một Entity vào database. Nó có thể gắn trên cột Id hoặc một cột nào đó.

Annotation này cũng được sử dụng cùng với @Generator.

Theo mặc định, chú thích @Id sẽ tự động xác định chiến lược tạo primary key, nhưng có thể ghi đè bằng cách áp dụng chú thích @GeneratedValue có hai tham số ****strategy****và ****generator****.

### **GenerationType.AUTO**

Column được đánh dấu bởi @GeneratedValue(strategy= AUTO) sẽ được gán giá trị tự động, giá trị đó có thể được sinh ra bởi SEQUENCE hoặc tự tăng (Nếu cột này có kiểu IDENTITY). Nó phụ thuộc vào loại database.

Với Oracle, PostgreSQL, Hibernate nó sẽ gọi một Sequence có tên Hibernate\_Sequence để tạo ra một giá trị tăng dần để gán giá trị cho cột này. Với các DB khác chẳng hạn như MySQL, DB2, SQL Server, Sysbase cột có thể là kiểu IDENTITY và giá trị của nó có thể tự tăng.

### **GenerationType.IDENTITY**

Cột có kiểu IDENTITY chỉ được hỗ trợ bởi một vài loại cơ sở dữ liệu, không phải là tất cả, ví dụ MySQL, DB2, SQL Server, Sybase và PostgreSQL. Oracle không hỗ trợ cột kiểu này.

### **GenerationType.SEQUENCE**

SEQUENCE là một đối tượng trong cơ sở dữ liệu, lưu trữ một giá trị tăng dần sau mỗi lần gọi nó để lấy giá trị tiếp theo, và được hỗ trợ bởi Oracle, DB2, và Postgres.

### **GenerationType.TABLE**

Khi sử dụng ****GenerationType.TABLE**** , chúng ta cần định nghĩa một table ****hibernate\_sequences**** như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **CREATE** **TABLE** `hibernate\_sequences` (  `sequence\_name` **char**(255) NOT NULL,  `next\_val` **decimal**(19,0) **DEFAULT** NULL,  **PRIMARY** **KEY** (`sequence\_name`)  ) ENGINE=InnoDB **DEFAULT** CHARSET=utf8mb4; |

Ta có bảng Category sử dụng GenerationType.TABLE như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Data  @Entity  @Table  **public** **class** Category {        @Id      @GeneratedValue(strategy = GenerationType.TABLE)  **private** Long id;        @Column  **private** String name;  } |

Hibernate cho phép chúng ta tạo Id table dựa trên ****org.hibernate.id.enhified.TableGenerator**** xác định một bảng có thể chứa nhiều giá trị được đặt tên cho bất kỳ Entity nào.

Chẳng hạn, chúng ta có table table\_identifier, bảng này sử dụng để quản lý các id tự tăng của các Entity. Cấu trúc table này như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **CREATE** **TABLE** `table\_identifier` (  `column\_name` **varchar**(255) NOT NULL,  `value\_column\_name` **decimal**(19,0) **DEFAULT** NULL,  **PRIMARY** **KEY** (`column\_name`)  ) ENGINE=InnoDB **DEFAULT** CHARSET=utf8mb4; |

Thêm 1 record cho bảng này: column\_name = category , value\_column\_name = 10.

Chúng ta sẽ sử dụng GenerationType.TABLE với custom bảng table\_identifier như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | **package** com.gpcoder.entities;    **import** javax.persistence.Column;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  **import** javax.persistence.Table;  **import** javax.persistence.TableGenerator;    **import** lombok.Data;    @Data  @Entity  @Table  **public** **class** Category {        @Id      //@GeneratedValue(strategy = GenerationType.TABLE)      @GeneratedValue(          strategy = GenerationType.TABLE,          generator = "table-generator"      )      @TableGenerator(          name =  "table-generator",          table = "table\_identifier",          pkColumnName = "column\_name",          valueColumnName = "value\_column\_name",          pkColumnValue = "category",          allocationSize = 5      )  **private** Long id;        @Column  **private** String name;  } |

Mỗi lần 1 Category được thêm vào database, giá trị value\_column\_name trong table table\_identifier ứng với column\_name=category sẽ được cộng thêm 5 (allocationSize) và giá trị này sẽ là id của record được thêm vào table category.

### **UUID**

UUID là một lớp của Java cho phép bạn tạo ra một chuỗi 36 ký tự ngẫu nhiên. Và với 36 ký tự xác suất trùng nhau là vô cùng bé. Chúng ta có thể sử dụng UUID với @GeneratedValue để Hibernate tạo ra một chuỗi ngẫu nhiên kiểu này gán cho giá trị của cột.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Table  **public** **class** UserVerification {        @Id      @GeneratedValue  **private** UUID id;    **private** String value;  } |

## **@Lob**

@Lob thường chú thích cùng với @Column để nói rằng cột đó có kiểu BLOB hoặc CLOB. Trong một số Database có phân biệt TINY, MEDIUM, LARGE BLOB/CLOB, còn một số database thì không.

Phần tử ****length**** trong @Column trong trường hợp này sẽ quyết định nó map vào BLOB/ CLOB nào. Trong trường hợp cho phép BLOB/CLOB tối đa hãy để length = Integer.MAX\_VALUE.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Entity  @Table  **public** **class** User {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        @Lob      @Column(name = "avatar", nullable = **true**, length = Integer.MAX\_VALUE)  **private** **byte**[] avatar;  } |

## **@ManyToOne**

@ManyToOne mô tả một quan hệ  ****N-1**** (Nhiều – Một), nó thường được sử dụng cùng với @JoinColumn.

Hibernate có các công cụ cho phép tạo ra các lớp Entity từ các bảng trong Database và Hibernate cũng cho phép tạo ra bảng từ các lớp Entity, bao gồm cả giàng buộc giữa các bảng (Foreign Key). Annotation @ForeignKey cho phép chỉ định rõ tên Foreign Key sẽ được tạo ra.

### **FetchType.LAZY**

LAZY nói với Hibernate rằng, hãy tải dữ liệu một cách lười biếng, nghĩa là chỉ tải khi được gọi (khi cần thiết).

Chẳng hạn bạn có một đối tượng Account, và gọi phương thức getOpenBranch() nó trả về một đối tượng Branch, đối tượng Branch chỉ trường (field) branchId được gán giá trị, các trường khác thì không.

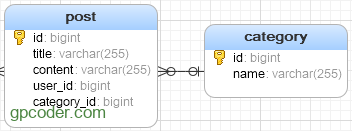
Thực tế hibernate chưa tải dữ liệu từ bản ghi tương ứng của bảng BRANCH vào đối tượng này. Nó chỉ thực hiện truy vấn dữ liệu khi bạn làm gì đó với đối tượng Branch vừa có được, chẳng hạn gọi phương thức branch.getName().

### **FetchType.EAGER**

EAGER nói với Hibernate rằng, hãy truy vấn toàn bộ các cột của bảng liên quan.

Chẳng hạn bạn có đối tượng Account, và gọi phương thức getOpenBranch() trả về đối tượng Branch đã có sẵn các giá trị cho các trường ( name, address, …). Thực tế dữ liệu của nó có được cùng trong 1 lần truy vấn với bảng Account.

Lưu ý: nên sử dụng LAZY thay vì EAGER vì lý do hiệu năng chương trình.

[](https://gpcoder.com/wp-content/uploads/2019/10/hibernate-many-to-one-e1573659727160.png)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Entity  @Table  **public** **class** Post {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        // ...        @ManyToOne(fetch = FetchType.EAGER)      @JoinColumn(name = "category\_id", nullable = **false**,          foreignKey = @ForeignKey(name = "fk\_post\_category"))  **private** Category category;  } |

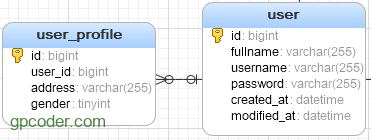
## **@OneToMany**

@OneToMany mô tả quan hệ ****1-N**** (Một – Nhiều). Nó là đảo ngược của @ManyToOne, và vì vậy nó dựa vào @ManyToOne để định nghĩa ra @OneToMany.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Entity  @Table  **public** **class** Category {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        @OneToMany(fetch = FetchType.LAZY, mappedBy = "category")  **private** Set<Post> posts;  } |

## **@OneToOne**

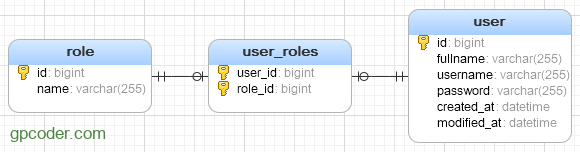
@OneToOne mô tả quan hệ ****1-1**** (Một – Một).

[](https://gpcoder.com/wp-content/uploads/2019/10/hibernate-one-to-one-1.png)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | @Entity  @Table(name = "user\_profile")  **public** **class** UserProfile {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        // ...        @OneToOne(fetch = FetchType.LAZY)      @PrimaryKeyJoinColumn(name = "user\_id", foreignKey = @ForeignKey(name = "fk\_user\_profile"))  **private** User user;  }      @Entity(name = "User")  @Table(name = "user")  **public** **class** User {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        // ...        @OneToOne(fetch = FetchType.LAZY, mappedBy = "user", cascade = CascadeType.ALL)  **private** UserProfile userProfile;  } |

## **@ManyToMany**

@ManyToMany mô tả quan hệ ****N-N**** (Nhiều – Nhiều).

[](https://gpcoder.com/wp-content/uploads/2019/10/hibernate-many-to-many.png)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | @Entity  @Table  **public** **class** Role {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        @ManyToMany(fetch = FetchType.LAZY, mappedBy = "roles")  **private** Set<User> users;  }    @Entity(name = "User")  @Table(name = "user")  **public** **class** User {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;        // ...        @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)      @JoinTable(name = "user\_roles",          joinColumns = { @JoinColumn(name = "user\_id", nullable = **false**, updatable = **false**) },          inverseJoinColumns = { @JoinColumn(name = "role\_id", nullable = **false**, updatable = **false**) })  **private** Set<Role> roles;  } |

## **@OrderBy**

@OrderBy được sử dụng để sắp xếp một danh sách, vì vậy nó có thể được sử dụng cùng với @OneToMany, @ManyToMany.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Table  **public** **class** Category {        // ...        @OneToMany(fetch = FetchType.LAZY, mappedBy = "category")      @OrderBy("title")  **private** Set<Post> posts;  } |

## **1, Design Patterns là gì?**

-****Design patterns**** là một kỹ thuật trong ****lập trình hướng đối tượng,**** được các nhà nghiên cứu đúc kết và tạo ra các mẫu thiết kế chuẩn. Và design pattern không phải là một ngôn ngữ lập trình cụ thể nào cả, mà nó có thể sự dụng được trong hầu hết các ngôn lập trình có hỗ trợ OOP hiện nay.

## **2, Tại sao nên sử dụng design patterns?**

-Design pattern giúp cho dự án của chúng ta dễ bảo trì, nâng cấp và mở rộng.

-Hơn nữa do design pattern đã được các nhà nghiên cứu đúc kết ra nên khi sử dụng design pattern thì chúng ta sẽ hạn chế được các lỗi tiềm ẩn.

-Và cuối cùng là khi sử dụng design pattern thì sẽ giúp code của chúng ta sẽ dễ đọc hơn. Điều này rất có lợi khi làm việc nhóm.

## **3, Cần gì để học design patterns.**

-Điều đầu tiên mình muốn nói là ****design pattern không dành cho nhưng bạn mới bắt đầu tìm hiểu về lập trình.****

-Muốn tìm hiểu và học được design pattern thì bạn phải nắm chắc được [kiến thức oop](https://toidicode.com/php-huong-doi-tuong-oop-32) đặc biệt là về abstract class,interface và static.

### ****Java Core là gì?****

Đây là một thuật ngữ chung và cơ bản nhất của Java, đặt nền tảng cho những phiên bản Java khác cùng với một tập hợp các công nghệ liên quan. Chẳng hạn như CORBA (kiến trúc môi giới các đối tượng), Java VM (máy ảo Java),…

Java Core đề cập đến một tập hợp các thư viện chứ không chỉ riêng một ngôn ngữ lập trình. Đây là dạng Java thuần túy nhất, chủ yếu được sử dụng để phát triển các ứng dụng desktop.

Đây là một ứng dụng Java độc lập bao gồm mọi thứ từ các khái niệm OOP đến những toán tử đặc biệt, từ kiểu dữ liệu đến các lớp wrapper, từ danh sách liên kết đến danh sách mảng, và xếp hàng đợi để xử lý ngoại lệ

**Điểm khác nhau giữa Map và Set**

Map(key, value) là một tập hợp các giá trị bằng key như Object. Điểm khác biệt giữa map và Object là key có thể thuộc bất kì kiểu dữ liệu nào.

Set(value) là tập hợp các giá trị bằng value và không có key truyền vào, mỗi value chỉ có một

Biến Final: là hằng số, trong suốt quá trình, giá trị ko thay đổi

**Sự khác nhau giữa lớp abstract và Interface**

|  |  |
| --- | --- |
| **Lớp Abstract** | **Interface** |
| Không hỗ trợ đa kế thừa  Sử dụng các biến static, non-static, final, non-final  Có phương thức trừu tượng hoặc không trừu tượng  Sử dụng từ khóa absstract để khai báo | Hỗ trợ đa kế thừa  Sử dụng biến static và final  Có các phương thức trừu tượng  Sử dụng từ khóa interface để khai báo |

### 1. Overloading (nạp chồng) phương thức là gì?

Nếu một lớp có nhiều phương thức có tên giống nhau nhưng các tham số khác nhau, được gọi là overloading phương thức (nạp chồng phương thức). Nó giúp code rõ ràng, dễ hiểu hơn.

### 2. Ghi đè (overriding) phương thức là gì?

Nếu lớp con có phương thức giống lớp cha được gọi là ghi đè (overriding) phương thức trong java.

Nói cách khác, nếu lớp con cung cấp sự cài đặt cụ thể cho phương thức đã được cung cấp bởi một lớp cha của nó được gọi là ghi đè (overriding) phương thức trong java.

### 9. Có thể sử dụng cả this() và super() trong một constructor?

Không, vì this() gọi đến một constructor khác trong lớp hiện tại, còn super() gọi constructor của lớp cha.

### 6. Sự khác nhau giữa nạp chồng và ghi đè là gì?

|  |  |  |
| --- | --- | --- |
| **No.** | **Nạp chồng phương thức (overloading)** | **Ghi đè phương thức (overriding)** |
| 1) | Nạp chồng phương thức được sử dụng để giúp code của chương trình dễ đọc hơn. | Ghi đè phương thức được sử dụng để cung cấp cài đặt cụ thể cho phương thức được khai báo ở lớp cha. |
| 2) | Nạp chồng được thực hiện bên trong một class. | Ghi đè phương thức xảy ra trong 2 class có quan hệ kế thừa. |
| 3) | Nạp chồng phương thức thì tham số phải khác nhau. | Ghi đè phương thức thì tham số phải giống nhau. |
| 4) | Nạp chồng phương thức là ví dụ về đa hình lúc biên dịch. | Ghi đè phương thức là ví dụ về đa hình lúc runtime. |
| 5) | Trong java, nạp chồng phương thức không thể được thực hiện khi chỉ thay đổi kiểu giá trị trả về của phương thức. Kiểu giá trị trả về có thể giống hoặc khác. Giá trị trả về có thể giống hoặc khác, nhưng tham số phải khác nhau. | Giá trị trả về phải giống nhau. |

Một interface có thể kế thừa một interface khác

Một interface không thể được kế thừa từ một lớp

Một lớp có thể implement nhiều interface

Tất cả các phương thức trong interface đều là absstract

Biến static có thể được sử dụng làm thuộc tính chung, để dùng chung dữ liệu cho tất cả objects (hoặc instances ) của lớp đó và điều đó giúp cho chương trình tiết kiệm bộ nhớ hơn.

Một phương thức static thuộc lớp chứ không phải đối tượng của lớp.

Một phương thức static có thể được gọi mà không cần tạo khởi tạo (instance) của một lớp.

Phương thức static có thể truy cập biến static và có thể thay đổi giá trị của nó.

Một phương thức static ****không**** thể được sử dụng từ khóa ****this**** và ****super****.

Người dùng không thể override (đè) phương thức static trong Java

các đối tượng được khởi tạo thông qua hàm khởi tạo (constructor). Mỗi lần bạn tạo mới một đối tượng sẽ có ít nhất một hàm khởi tạo được thực thi.

void Person() {} // đây là một method, không khải constructor

static Person() {} // không thể đi kèm với từ khóa static

final Person() {} // không thể đi kèm với từ khóa final

abstract Person() {} // không thể đi kèm với từ khóa abstract

****@Autowired****: Annotation này trong spring giúp tự động liên kết các ****Bean****lại với nhau, đây là một trong những đặc điểm nỗi bật của Spring để thể hiện tính chất [DI và IOC (Dependency Injection và Inversion of Control)](https://itphutran.com/spring-framework/dependency-injection-va-inversion-control-trong-spring-framework).

****@Service****: dùng để scan các components để đánh dấu các Service của lớp business.

****@Controller****: đây là môt Annotation trong spring mvc dùng để đánh dấu đây là lớp có chức năng là Controller.

****@RequestMapping****: đây là môt Annotation trong spring mvc dùng để cấu hình URL tương ứng với @WebServlet và mapping trong file xml với [JSP/servlet](https://itphutran.com/category/lap-trinh-java/jsp-servlet).

****@RequestParam****: đây là môt Annotation trong spring mvc dùng để gán một request parameter vào một parameter của phương thức.Thường dùng để lấy dữ liệu từ form và các biến tham số trên URL.

****@ModelAttribute:****đây là môt Annotation trong spring mvc dùng để lấy các tham số và trả về một đối tượng (Object).