**Dependency Injection (DI)**

Trong tài liệu có nói thế này:

Dependency Injection is a design pattern, ...

Thế thì bạn có thể hiểu **nôm na** nó là một phương pháp lập trình, là một thiết kế để bạn có được hiệu quả cao hơn khi code. Trước khi phương pháp này ra đời, bạn vẫn code bình thường, nhưng bây giờ có rồi, đi theo nó sẽ giúp ích nhiều hơn cho việc lập trình của bạn.

Vậy cuối cùng Dependency Injection nó bảo chúng ta làm gì? 🙃 (nôm na nhiều mà quên mịa vấn đề chính)

Mình sẽ giải thích cho các bạn qua một ví dụ như lày:

public class Girl{

private Bikini outfit; // mỗi cô gái sẽ có một bộ bikini khi ra ngoài

public Girl(){

outfit = new Bikini(); // Khi bạn tạo ra 1 cô gái, bạn cho cô ta mặc Bikini chẳng hạn

}

}

Trước hết, qua đoạn code này, bạn sẽ thấy là khi bạn tạo ra một Girl, bạn sẽ tạo ra thêm 1 bộ Bikini đi kèm với cô gái đó. Lúc này, Bikini tồn tại mang ý nghĩa là **dependency** (phụ thuộc) của Girl.

Khi khởi tạo thuộc tính như này, bạn vô tình tạo ra một điểm thắt nút trong chương trình của mình, giả sử, Girl muốn mặc một bộ *Váy + Áo thun hở rốn* hay *không mặc gì* thì sao? Bạn sẽ phải thay class Bikini thành SkirtWithTshirt(Váy với áo T-shirt) hay Naked (Trần như nhộng) ư?

Hay nguy hiểm hơn, bộ đồ Bikini bị hỏng? (code lớp Bikini không hoạt động?) nó sẽ ảnh hưởng trực tiếp tới Girl.

Vấn đề là ở đó, nguyên tắc là:

Các Class không nên phụ thuộc vào các kế thừa cấp thấp, mà nên phụ thuộc vào Abstraction (lớp trừu tượng).

Nghe hơi khó hiểu. Bây giờ mình thay đoạn code như này:

// Một interface cho việc ăn mặc

public interface Outfit {

public void wear();

}

// Một object cấp thấp, implement của Outfits

public class Bikini implements Outfit {

public void wear() {

System.out.println("Đã mặc Bikini");

}

}

// Bây giờ Girl chỉ phụ thuộc vào Outfit. nếu muốn thay đổi đồ của cô gái, chúng ta chỉ cần cho Outfit một thể hiện mới.

public class Girl{

private Outfit outfit;

public Girl(){

outfit = new Bikini();

}

}

Tới đây, chúng ta mới chỉ Abtract hóa thuộc tính của Girl mà thôi, còn thực tế, Girl vẫn đang bị gắn với một bộ Bikini duy nhất. Vậy muốn thay đồ cho cô gái, bạn phải làm như nào.

Phải sửa code thêm chút nữa:

public class Girl{

private Outfit outfit;

public Girl(Outfit anything){

this.outfit = anything // Tạo ra một cô gái, với một món đồ tùy biến

// Không bị phụ thuộc quá nhiều vào thời điểm khởi tạo, hay code.

}

}

public class Main {

public static void main(String[] args) {

Outfit bikini = new Bikini(); // Tạo ra đối tượng Bikini ở ngoài đối tượng

Girl ngocTrinh = new Girl(bikini); // Mặc nó vào cho cô gái khi tạo ra cô ấy.

}

}

Với đoạn code ở trên, chúng ta đã *gần như* tách được Bikini ra hoàn toàn khỏi Girl. điều này làm giảm sự phụ thuộc giữa Girl và Bikini. Mà tăng tính tùy biến, linh hoạt cho code.

Bây giờ Girl sẽ hoạt động với Outfit mà thôi. Và Outfit ở đâu ra? Chúng ta **tạo ra** và **đưa nó vào** (Inject) cô gái Girl.



Khái niệm Dependency Injection từ đây mà ra~

Dependency Injection là việc các Object nên phụ thuộc vào các Abstract Class và thể hiện chi tiết của nó sẽ được Inject vào đối tượng lúc runtime.

Bây giờ muốn Girl mặc gì khác, bạn chỉ cần tạo một Class kế thừa Outfit và *Inject* (dịch là *Tiêm vào* cũng được) nó vào Girl là xong!

Các cách để *Inject dependency* vào một đối tượng có thể kể đến như sau:

* **Constructor Injection**: Cái này chính là ví dụ của mình, tiêm dependency ngay vào Contructor cho tiện.
* **Setter Injection**: Ồ, sao không chứ 😗 chúng ta học về Setter từ những bài học vỡ lòng rồi, quá hợp lý. Xài girl.setOutfit(new Naked()) 😈
* **Interface Injection**: Mỗi Class muốn inject cái gì, thì phải implement một Interface có chứa một hàm inject(xx) (Gần như thay thế cho setter ý bạn). Rồi bạn muốn inject gì đó thì gọi cái hàm inject(xx) ra. Cách này hơi dài và khó cho người mới.

**Inversion of Control**

Dependency Injection giúp chúng ta dễ dàng mở rộng code và giảm sự phụ thuộc giữa các dependency với nhau. Tuy nhiên, lúc này, khi code bạn sẽ phải kiêm thêm nhiệm vụ Inject dependency (tiêm sự phụ thuộc). Thử tưởng tượng một Class có hàng chục dependency thì bạn sẽ phải tự tay inject từng ý cái. Việc này lại dẫn tới khó khăn trong việc code, quản lý code và dependency

public static void main(String[] args) {

Outfit bikini = new Bikini();

Accessories gucci = new GucciAccessories();

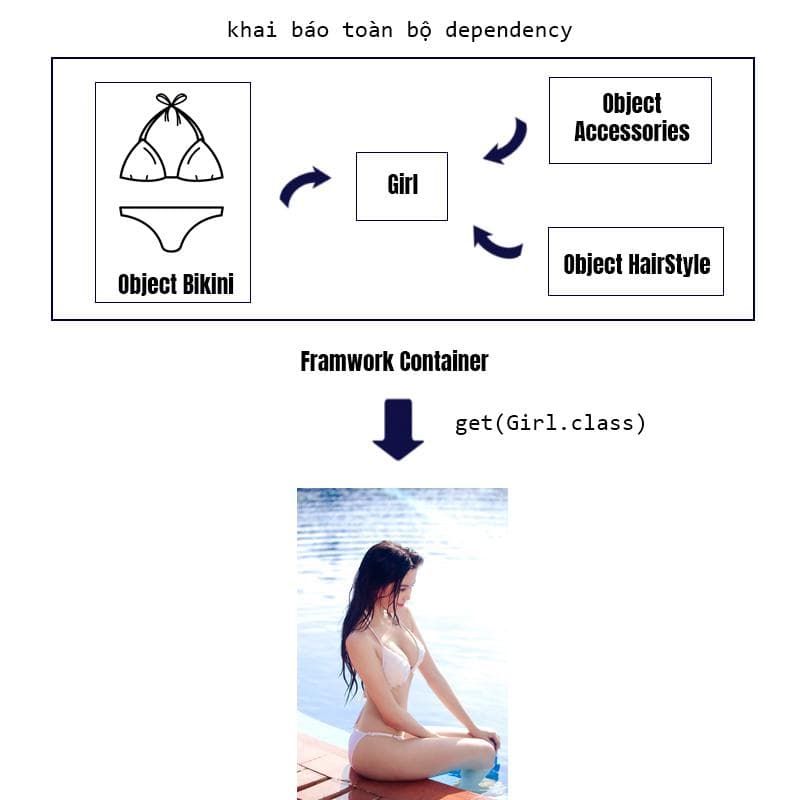
HairStyle hair = new KoreanHairStyle();

Girl ngocTrinh = new Girl(bikini, gucci, hair);

}

Giá như lúc này có thằng làm hộ được chúng ta việc này thì tốt biết mấy.

Bây giờ giả sử, chúng ta định nghĩa trước toàn bộ các dependency có trong Project, mô tả nó và tống nó vào 1 cái kho và giao cho một thằng tên là framework quản lý. Bất kỳ các Class nào khi khởi tạo, nó cần dependency gì, thì cái framework này sẽ tự tìm trong kho rồi *inject* vào đối tượng thay chúng ta. sẽ tiện hơn phải không?



That it, chính nó, đó cũng chính là nguyên lý chính của Inversion of Control (IOC) - Đảo chiều sự điều khiển

Nguyên văn Wiki:

Inversion of Control is a programming principle. flow of control within the application is not controlled by the application itself, but rather by the underlying framework.

Khi đó, code chúng ta sẽ chỉ cần như này, để lấy ra 1 đối tượng:

@Override

public void run(String... args) throws Exception {

Girl girl = context.getBean(Girl.class);

}

Đối với Java thì có một số Framework hỗ trợ chúng ta Inversion of Control (IOC), trong đó nổi bật có:

* **Spring framework**
* **Google Guice**

**Spring framework** là một framework từ những ngày đầu, ra đời để hiện thực ý tưởng *Inversion of Control (IOC)*, tuy nhiên, theo thời gian, Spring lớn mạnh và trở thành một hệ sinh thái rộng lớn phục vụ rất nhiều chức năng trên nền tảng IoC này.