2008

JGENTLE FRAMEWORK REFERENCE 1.0

From Technologies to Solutions

A step-by-step guide to Java development with the developer-friendly JGentle framework.

Le Quoc Chung

2/9/2008

JGENTLE FRAMEWORK PROJECT REFERENCE

1 - Chỉ mục

2 - Giới thiệu JGentle

Hệ thống *JGentle Framework* là một *AOE (Aspect Oriented Environment)*, một *lightweight container* kết hợp với một tập hợp các API giúp đơn giản hóa các vấn đề về *Dependency Injection (DI) , Inversion Of Control (IOC), Aspect Oriented Programing(AOP),* ... Nhưng không giống như một số *framework*, hay các *AOP framework* khác tập trung vào việc xử lý AOP hay cung cấp một IoC *container* đơn thuần dựa trên việc cấu hình với XML, JGentle cung cấp các chức năng *DI và AOP* hướng đến tính mở rộng và tương thích với các hệ AOE khác (như Spring, ...), kết hợp với *annotation (*được chuyển đổi thành *Definition)* để trợ giúp cho hệ thống cấu hình, và đồng thời đoạn tuyệt hoàn toàn với việc sử dụng XML làm dữ liệu định nghĩa thông tin cấu hình.

Dựa trên nền *DI, ADI (Annotation Dependency Injection), AOH (Annotation Object Handling), dDI (Deep Dependency Injection), Definition*, ... JGentle cung cấp các *services* quản lý như *Event Services, Queued Component, Data Locator,* ... và cung cấp một cơ chế cấu hình ứng dụng sử dụng *annotation, Definition* một cách tùy biến. JGentle hoàn toàn là POJOs, xây dựng dựa trên các đối tượng *pure java* (thuần java) nên có thể tích hợp, tương thích với bất kì hệ thống nào phát triển bằng Java. Mặt khác JGentle AOP tuân thủ chặt chẽ đặc tả kĩ thuật của *AOPAlliance* nên các thành phần *AOP* trong JGentle hoàn toàn có thể tương thích với các *thành phần AOP* được xây dựng trên bất kì một *AOE* nào, miễn là *AOE* đó tuân thủ theo *AOPAlliance*.

Kết hợp điểm mạnh của các hệ thống khác, đơn giản hóa các *API* thao tác, JGentle đã tạo ra một *environment* mạnh mẽ để phát triển các *module* tách rời với các thông tin cấu hình hệ thống, giúp cho hệ thống ổn định, chịu đựng được các phát sinh mở rộng và linh hoạt khi phát triển cũng như bảo trì.

2.1 - Tại sao sử dụng JGentle

Một vài lý do cho mục đích sử dụng JGentle thay vì các *framework* khác:

- Mục tiêu của JGentle là nhắm đến một khía cạnh quan trọng mà các *framework* hiện tại không thực hiện hoặc chưa thực hiện. Đó là cung cấp một cơ chế quản lý các *business objects, một cơ chế DI* thông qua *annotation, definition,* loại bỏ hoàn toàn sự phụ thuộc với cách truyền thống trong việc sử dụng các *XML files* hoặc *property files* làm dữ liệu cấu hình, đồng thời thay thế cách sử dụng *metadata* với *annotation* trực tiếp bằng thông tin dữ liệu *definition* gián tiếp, tách rời *logic code* với thông tin *annotation*.

- JGentle gia tăng khả năng kết nối và tích hợp với các *framework* khác (như *Hibernate, Lucene,* ...) cũng như các kĩ thuật chuẩn khác trên nền JAVA như *JDBC, RMI, JMX,* ... cung cấp nhiều xử lý tự động khác nhau với các mã lệnh phải quản lý ở mức *low-level*. JGentle được thiết kế như là một giải pháp tổng thể dựa trên nhiều kĩ thuật khác nhau, tạo lập một nền tảng trung tâm kết nối với các nền tảng kĩ thuật tách rời khác, giảm thiểu một số lượng lớn công việc cho nhà phát triển nếu như muốn tích hợp nhiều kĩ thuật khác nhau trên cùng một ứng dụng.

- Thành phần cốt lõi của JGentle là một **Inversion Of Control** *container* (IoC) hay còn có thể được biết đến với một khái niệm tương tự đó là **Dependency Injection** (DI) - một thuật ngữ được đưa ra bởi Martin Fowler và Rod Johnson (cha đẻ *Spring framework*) cùng nhóm phát triển *PicoContainer* vào khoảng cuối năm 2003. Nhưng thay vì như một số *framework* hỗ trợ DI khác, chỉ cho phép chỉ định *dependencies* tại thời điểm *creation-time* (vd *Spring*), JGentle IoC còn cho phép chỉ định các *dependencies* tại thời điểm *invocation time* (thời điểm lúc *bean* được triệu gọi thực thi). DI trong JGentle đưa ra nhiều cải tiến, cho phép nhà phát triển *filter* được dữ liệu *dependencies*, chỉ định *inject* các *dependencies* khác nhau trong những điều kiện khác nhau, đồng thời còn cho phép *outject* (khái niệm ngược lại với *inject*) các dữ liệu *dependencies* ngược trở lại lại *container* theo một *scope* chỉ định. Và điều quan trọng là thông tin cấu hình DI trong JGentle hoàn toàn là *annotation*, các cách thức *wiring beans* cũng như việc chỉ định *dependencies* trong JGentle không hề phụ thuộc vào bất kì một file *XML* cấu hình nào nhưng vẫn đảm bảo dữ liệu thông tin cấu hình độc lập với *logic code*.

- JGentle cung cấp một cơ chế *framework* toàn diện cho Data Access, bao gồm việc tương tác với JDBC hay một ORM *framework* bất kì như Hibernate, …

- JGentle tự bên trong có cung cấp một hệ thống các *services* khác nhau, cung cấp các chức năng khác nhau trợ giúp cho việc phát triển và xây dựng hệ thống. Không những thế, JGentle còn cho phép các nhà phát triển cấp thấp có thể tự xây dựng các hệ thống *services* khác dựa trên nền JGentle, cũng như tận dụng được những điểm mạnh trong JGentle như *DI, ADI, AOH* ... trong khi xây dựng *services*. *Services* sau khi được xây dựng có thể được sử dụng như là một thành phần trong JGentle *container*.

- JGentle cung cấp một cơ chế quản lý *metadata* dựa trên *annotation*, thay thế cách thức sử dụng *annotation* một cách trực tiếp, thay vào đó sử dụng gián tiếp thông qua *definition* (một loại *object* mà thông tin dữ liệu được chuyển đổi từ *annotation*), cho phép tách rời nội dung *metadata* và *logic code*.

- Cung cấp một cơ chế *Annotation Dependency Injection* (ADI), cho phép *inject* các *annotations* khác nhau vào *logic code* tại thời điểm *run-time* mà không cần thao tác vào *source code*. Dữ liệu *annotation* giờ đây trong JGentle có thể xem như là những *dependency instances*, có thể *inject* vào bất kì vị trí nào chỉ định.

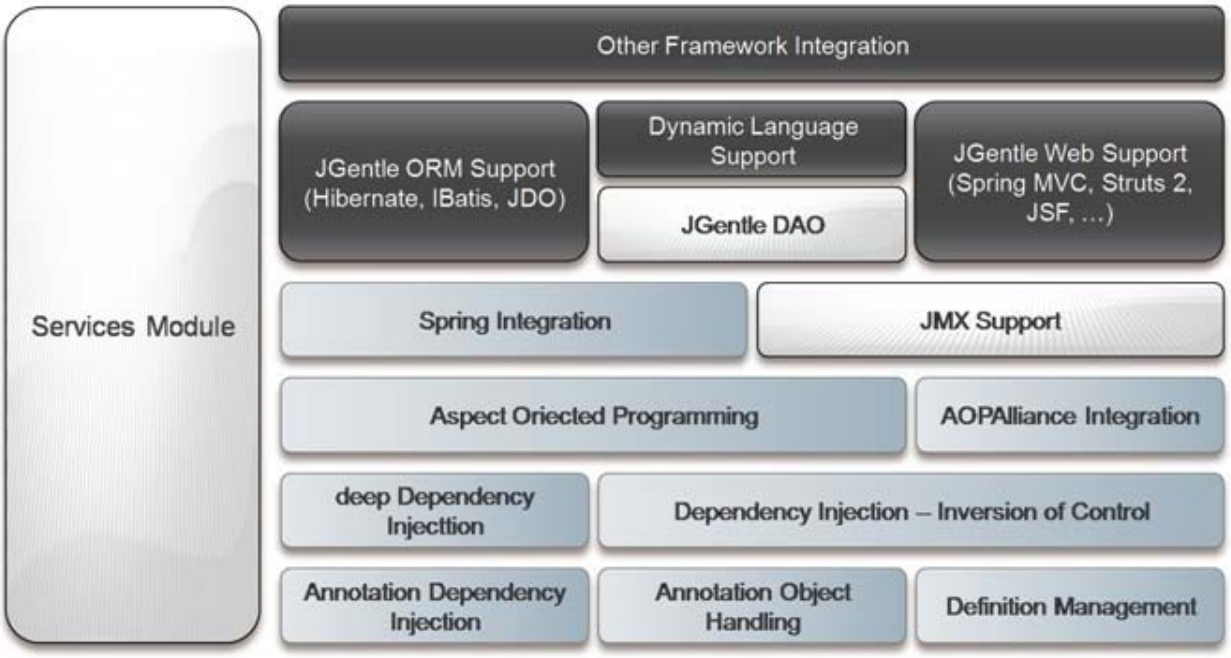
- JGentle hỗ trợ bên trong một hệ thống *framework* con, *Aspect Oriented Programming framework* (AOP), được chỉ định cấu hình hoàn toàn với *annotation*, tuân thủ *AOPAlliance*, do đó đảm bảo có thể tích hợp được với *AOP framework* khác miễn là *AOE (Aspect Oriented Environment)* trên các hệ *framework* này cũng tuân thủ AOPAlliance.

- Được thiết kế nhằm đến việc mở rộng và tương thích với các hệ *lightweight container* và các *framework* hiện hành. Cho phép các *beans* được quản lý bởi JGentle có thể tương tác với các *beans* được quản lý trên các hệ *container* khác vd như Spring, Hivemind, ... Điều này giúp tăng tính khả chuyển, cho phép các ứng dụng đã được xây dựng trên các hệ *lightweight container* khác có thể tích hợp và cùng làm việc với JGentle một cách dễ dàng.

3 - Cấu trúc JGentle

- Cấu trúc các thành phần trong JGentle

3.1 - Cấu trúc các thành phần trong JGentle



3.1.1 - Definition Management – DM

Là thành phần cơ bản và là lõi của toàn bộ của hệ thống *metadata configuration* trong JGentle container. DM chịu trách nhiệm quản lý các thông tin *Definition*, khởi tạo, *convert*, gỡ bỏ … thông tin *Definition*. Ngoài việc quản lý các thông tin *Definition*, DM còn chịu trách nhiệm thực thi các tác vụ *validate* các *annotation* tương ứng khi thực thi *convert* thông tin *annotation* thành *Definition*, đồng thời chịu trách nhiệm đăng kí hoặc gỡ bỏ các thông tin đăng kí *catch* các *exception* nếu như có ngoại lệ được ném ra trong quá trình *validate* thông tin *annotation*.

3.1.2 - Annotation Object Handling – AOH

Là thành phần chịu trách nhiệm quản lý và khởi tạo các *customize services*. Là thành phần lõi của toàn bộ *services* trong JGentle, về mặt bản chất *InjectCreator container* cũng chính là một *services* được đăng kí bên trong một AOH, cũng như tất cả các thành phần

services khác trong JGentle như *AOP, DDI, ADI, EventServices, DataLocator,* ... tất cả là những services được xây dựng bên trong lõi của một AOH thống nhất, các services theo một số tiêu chí cụ thể sẽ được tổng hợp và kết hợp lại thành một thể thống nhất (vd: *InjectCreator container* chính là một tập hợp các services như *AOP, dDI, DI, ADI,* ....). Nhà phát triển có thể sử dụng hệ thống API và cơ chế *config* có sẵn trong AOH để tự tạo cho riêng mình các *services* khác kết hợp với hệ thống *services* có sẵn cung cấp bởi JGentle hoặc tạo cho riêng mình một *container* độc lập.

3.1.3 - Annotation Dependency Injection – ADI

Là thành phần chịu trách nhiệm quản lý chức năng *inject* các *Annotation Dependencies.* Trong hệ thống JGentle, *Annotation* cũng có thể xem như là một trong những thành phần d*ependencies*. ADI chịu trách nhiệm quản lý, cũng như *inject* các thông tin *dependencies* này vào các thực thể *beans* chỉ định. Nhờ ADI quản lý các *Annotation Dependencies* nên giờ đây các *annotation* độc lập hoàn toàn với *logic code*, các *annotation* giờ đây có thể chỉ định tại một nơi và sau đó có thể *inject* vào bất kì *beans* nào chỉ định thông qua cơ chế cấu hình với *Definition* của JGentle.

3.1.4 - Dependency Injection – DI

Dependency Injection - DI chịu trách nhiệm quản lý các tác vụ thực thi việc *inject* các *dependencies.*

*Nguyên lý cơ bản của **Dependency Injection** là các objects sẽ **"định nghĩa"** các **dependencies** của chúng (những **objects** khác) chỉ cần thông qua tham số truyền của constructor hoặc properties. Sau đó là công việc của container mới thật sự inject những **dependencies** này vào object khi khởi tạo bean.*

*Điều này được gọi là sự đảo ngược (**Inversion**), tên đầy đủ là **Inversion Of Control** (IoC) – "đảo ngược sự điều khiển". "Đảo ngược điều khiển" ở đây có nghĩa là thay vì object trước đây sẽ phải tự khởi tạo, quyết định **dependencies** nào, ở đâu sẽ làm việc với nó, thì giờ đây công việc này được chuyển cho container quản lý, objects chỉ cần quan tâm đến **business logic code** của chính mình.*

Trong JGentle, DI sẽ quản lý các thông tin *dependencies* thông qua cơ chế cấu hình bằng *Definition* trong JGentle. Thay vì sử dụng XML để cấu hình như các *container* khác, JGentle lựa chọn *Annotation* (được chuyển đổi thành *Definition*) để cấu hình các thao tác *Dependency Injection*. Khả năng quản lý *dependencies* của DI trong JGentle dựa trên *Definition* khiến cho việc cấu hình và quản lý *beans* của bạn giờ đây trở nên đơn giản, xúc tích, ngắn gọn thay vì quản lý một tập hợp hỗn độn các file XML như trong các hệ *container* trước đây.

3.1.5 - Deep Dependency Injection – dDI

Là một chức năng cải tiến của DI trong JGentle, giờ đây với dDI, các *dependencies* không những có thể được *inject* vào trong một bean được cấu hình mà còn có thể *outject* ra khỏi bean đi ngược trở lại container trở thành một thành phần có thể *injectable*, cho phép sử dụng như là những *dependency instances* cho các *Beans* khác. Ngoài ra thông tin *dependencies* giờ đây còn có thể được *filter* trước khi được *inject* vào *Bean*, hoặc có thể chỉ định *builder* cho *dependencies* trong trường hợp *dependencies* là *null*. *Deep DI* nâng cấp khả năng *inject dependencies* thông thường của *DI*, hỗ trợ việc *inject* tại thời điểm *invocation* thay vì *creation-time* (các *dependencies được inject* tại thời điểm khởi tạo *Bean*, và giữ nguyên các *dependencies* này đến khi hết *Lifecycle* của *Bean*) như các hệ *container* hỗ trợ DI khác (vd Spring).

3.1.6 - Aspect Oriented Programming – AOP

Thành phần AOP chịu trách nhiệm quản lý các chức năng *AOP* trong JGentle. Các thành phần của AOP như *advice, pointcut, joinpoint, aspect, weaving*, ... giờ đây có thể cấu hình đơn giản nhất có thể thông qua *annotation*. JGentle cung cấp một cơ chế giải quyết các vấn đề của AOP, và một tập hợp các *API* cho phép quản lý và cấu hình các chức năng *AOP* cho các *beans* trong *JGentle container*. Đồng thời JGentle AOP tuân thủ chặt chẽ *AOPAlliance*, do đó các thành phần *AOP* được xây dựng trên các hệ *AOP framework* khác có thể hoạt động và tái sử dụng trong *JGentle* mà không gặp bất kì trở ngại nào, miễn là các AOE *(Aspect Oriented Environments)* đó tuân thủ *AOPAlliance* (vd như Spring).

3.1.7 - Spring Integration

Spring là một *lightweight container*, cung cấp một tập hợp các chức năng quản lý, khởi tạo và điều phối vòng đời của *beans*, đồng thời cung cấp mạnh mẽ các chức năng như *Dependency Injection*, *AOP, Transaction Management, RMI, JMX support,* ... thông qua cơ chế cấu hình với *XML* hoặc *annotation (Spring java-config)*. Để tận dụng hệ thống đồ sộ chức năng mà Spring cung cấp, JGentle cung cấp một cơ chế hỗ trợ việc tích hợp các *beans* được quản lý bởi *Spring container,* để có thể làm việc, hoạt động và sử dụng trong *JGentle container*. Điều này cho phép các *beans* được quản lý bởi *JGentle container* có thể dễ dàng tương tác hoặc kết hợp với các các *beans* khác được quản lý bởi *Spring*, do đó các *beans* trong JGentle vẫn có thể tận dụng được cơ chế cấu hình linh hoạt của JGentle, và các thành phần chức năng không có trong Spring nhưng đồng thời vẫn nhận được các chức năng mạnh mẽ từ Spring.

3.1.8 - JMX Support

JGentle hỗ trợ *DI, dDI, ADI*, … là những cách cấu hình mạnh mẽ cho các *properties* của *bean*, cũng như cấu hình *annotation* trong ứng dụng. Nhưng một khi ứng dụng đã được triển khai và hoạt động, khả năng DI, hay ADI không phải là những công cụ hữu dụng để có thể thay đổi, điều khiển, theo dõi cũng như cấu hình các thông tin *configuration*. Trong khi đó, Java Management Extensions (JMX) - một đặc tả chuẩn của JAVA, lại cho phép công cụ hóa ứng dụng, cho phép quản lý, giám sát (*monitoring*) cũng như cấu hình (*configuration*), và điều quan trọng hơn nữa rằng JMX cho phép can thiệp vào hoạt động của hệ thống ngay tại thời điểm *run-time*. JMX cho phép nhà phát triển có thể giám sát được tài nguyên hệ thống, cho phép ứng dụng có thể "mở hóa" các thông tin về thuộc tính, hành vi, thông tin cấu hình, cung cấp một hệ thống đặc tả chuẩn và nhiều công cụ khác nhau giúp quản lý, giám sát, và bảo trì ứng dụng. Do đó, việc hỗ trợ JMX trở thành công việc quan trọng và không thể thiếu trong các hệ thống *bean container*, và dĩ nhiên JGentle cũng không phải là trường hợp ngoại lệ.

Dựa vào hệ thống JMX *support* trong JGentle, các *beans* được quản lý bởi JGentle *container* hoàn toàn có thể tận dụng được các thế mạnh của JMX. JGentle JMX *module* cho phép nhà phát triển có thể xuất bản các JGentle *beans* như là những Model MBeans, để có thể truy xuất vào bên trong hệ thống, tối ưu dữ liệu thông tin cấu hình – thậm chí ngay cả khi ứng dụng đang được vận hành. Việc triển khai các MBeans trong JGentle được xử lý dựa

trên khả năng DI, ADI do đó được đơn giản hóa đến tối đa, vừa có thể hiện như là các MBeans được quản lý trong một MBean *server*, nhưng đồng thời vẫn được quản lý trong JGentle *container* như là những JGentle *beans* một cách bình thường.

3.1.9 - JGentle Data Access

Gần như trong hầu hết các hệ thống, việc truy vấn đến cơ sở dữ liệu (*database*) lúc nào cũng là một vấn đề quan trọng và là quá trình không thể thiếu trong khi phát triển hệ thống. Hệ thống *Data Access framework* trong JGentle giúp đơn giản hóa các thao tác truy xuất cơ sở dữ liệu, cho phép tích hợp với nhiều kĩ thuật truy vấn dữ liệu *"data access"* khác nhau. Bất cứ khi nào bạn cần truy vấn thông tin dữ liệu thông qua *JDBC, iBatis* hay thông qua một *ORM framework (Object Ralational Mapping)* ví dụ như *Hibernate*, JGentle đều hỗ trợ, đồng thời giúp giải phóng các đoạn *code* truy vấn phức tạp, thay vào đó là phát triển trên một hệ thống cấu hình các *persistence code,* đơn giản thông qua *DI, ADI,* ... trợ giúp quản lý *transactions*, *handle* các ngoại lệ *exceptions*, khởi tạo cũng như quản lý *connections, connection pool* ... Do đó giúp nhà phát triển có thể tập trung nhiều hơn vào *business logic code* của ứng dụng, tránh khỏi việc phải thực hiện các công việc truy vấn dữ liệu ở mức *low-level* như trước. Giờ đây, nhà phát triển chỉ cần viết các *logic code* của dữ liệu, cấu hình *Data Access* thông qua *JGentle DI, ADI*, và không còn cần bận tâm bất kì điều gì nữa, ... tất cả phần còn lại đã có JGentle xử lý.

3.1.10 - Framework Integration

Bản thân hệ thống JGentle mặc dù cung cấp rất nhiều chức năng khác nhau giúp giải quyết các vấn đề khác nhau trong khi phát triển một ứng dụng *enterprise*, nhưng vẫn còn nhiều giải pháp khác ngoài JGentle, đáp ứng được những nhu cầu, khía cạnh riêng biệt mà JGentle chưa hỗ trợ. Đồng thời, có nhiều giải pháp đã được phát triển từ nhiều năm nay, được áp dụng trên nhiều hệ thống vận hành hiện tại, và dĩ nhiên đã trải qua một khoảng thời gian đủ … để có thể nói là đã trưởng thành, ổn định và đáp ứng được sự phát triển cũng như mở rộng. Do đó, thay vì ôm đồm giải quyết tất cả mọi vấn đề, JGentle lựa chọn giải pháp kết hợp với các nền tảng *framework* khác ví dụ như *Spring, Hibernate, iBatis, JMX, Struts, Ajax framework, Lucene* ... và nhiều *framework* khác nhằm tạo ra một hệ thống phát triển toàn diện, bù đắp những chỗ còn thiếu sót trên các nền tảng này (ví dụ *Definition, ADI,* ...), cung cấp các thao tác triển khai khác trên DI. Đồng thời JGentle tạo ra một tập

hợp các *class* trợ giúp, kết hợp với các *framework*, giúp đơn giản hóa các cách thức triển khai, cũng như xây dựng một hệ thống tích hợp nhiều nền tảng *framework* khác nhau mà vẫn thừa hưởng triệt để được điểm mạnh của từng hệ thống.

3.1.11 - Services Module

Là một tập hợp các *services* được xây dựng sẵn trong JGentle, cung cấp các chức năng như là một hệ thống tổng thể giải quyết các vấn đề khác nhau trong quá trình phát triển ứng dụng. Như đã mô tả ở trên, bản chất *InjectCreator container* cũng chính là một *services* trong JGentle cung cấp các chức năng quản lý *DI, dDI, AOP,* ... Đó là các thành phần *services* cơ bản trong *InjectCreator container.* Ngoài các *services* trên, JGentle còn cung cấp các *Services* xử lý các tác vụ khác như: *EventServices, DataLocator, RMI support, ObjectPooling, Queued Component* ... được tổng hợp và quản lý trong *ServicesContext container*. Đây là các thành phần *services* xử lý bổ sung ở mức *enterprise* kèm theo cung cấp bởi JGentle ngoài các chức năng hệ thống lõi cơ bản.

Ngoài ra, kết hợp với sự hỗ trợ của thành phần *AOH (Annotation Object Handling),* JGentle còn cho phép tạo các chức năng *services* riêng và có thể thêm vào hệ thống *container* có sẵn của JGentle, giúp bạn có thể xây dựng các hệ thống *services* riêng giải quyết các bài toán cụ thể mà JGentle chưa giải quyết mà vẫn tận dụng được cơ chế cấu hình bằng *Definition* linh hoạt trong JGentle.

4 - Core Technologies

- Giới thiệu
- Container, Configuration và Beans
- Definition
- Dependency Injection
- Deep Dependency Injection
- Annotation Dependency Injection
- Aspect Oriented Programming – AOP
- Annotation Object Handling

4.1 - Giới thiệu

Về bản chất JGentle là một *lightweight container*, và lõi của nó là cài đặt của các thành phần cơ bản như *Inversion Of Control (IoC), Dependency Injection (DI), Aspect Oriented Programming (AOP)* .... Nhưng trong cài đặt thực tế, *JGentle framework* cung cấp một vài hệ thống các *container* khác nhau đảm đương các trách nhiệm cũng khác nhau như: *InjectCreator container, ServicesContext container, WebContext container,* ...

Trong khi *InjectCreator container* chỉ cung cấp những thành phần cơ bản để khởi tạo và quản lý bean như *(DI, dDI, ADI, AOH, ...)* thì *ServicesContext container* lại cung cấp thêm các thành phần bổ sung *services* đóng vai trò nhưng nhà cung ứng các dịch vụ *services* lõi trong hệ thống *container* như: *EventService, RMI support, DataLocator, ...* đồng thời cung cấp các cơ chế *integrate* với các nền tảng hệ thống *framework* phổ biến khác. Các *container* khác nhau đảm đương các công việc khác nhau, cũng như một phạm vi ứng dụng khác nhau, do đó tùy vào hoàn cảnh, trường hợp cụ thể bạn có thể khởi tạo các *container* chỉ định khác nhau.

- Các container:

InjectCreator : InjectCreator interface cung cấp các thành phần cơ bản để quản lý *Dependency Injection (DI), Deep Dependency Injection (dDI), Annotation Dependency Injection (ADI), Annotation Object Handling (AOH), Aspect Oriented Programming (AOP),* ... *InjectCreator* cũng chịu trách nhiệm về quản lý vòng đời (Lifecycle) của *beans*, *scope* hoặc các thông tin khởi tạo *bean* khác, đồng thời *InjectCreator* cũng chịu trách nhiệm việc đọc thông tin cấu hình *config* trong các *objects* hoặc *object classes* (extends từ *AbstractConfig class,* hoặc *implements* từ *Configurable interface*) để cấu hình các thành phần trong hệ thống.

ServicesContext : cài đặt của *ServicesContext interface* được xây dựng hoàn toàn dựa trên cài đặt của *InjectCreator* và thêm vào các thành phần chức năng *services* khác như *Event Service, ObjectPooling, DataLocator, RMI Integration, WebServices,* ... Về bản chất *InjectCreator* chỉ cung cấp các thành phần chức năng cấu hình cơ bản, còn *ServicesContext* thì lại bổ sung các chức năng *Enterprise* dựa trên nền *InjectCreator*, do đó nó có toàn bộ các thành phần chức năng của *InjectCreator*.

WebContext : là một cài đặt khác tương tự *ServicesContext* nhưng được phát triển nhằm cho mục đích vận dụng *JGentle Containe*r trên nền hệ thống web (*JSP/Servlet*), là *container* được khởi tạo trong thành phần trung tâm của *JGentle Web Framework*.

4.2 - Container, Configuration và Beans

4.2.1 - Beans

Trong JGentle, các *objects* tạo nên các thành phần cơ bản trong ứng dụng của bạn và được quản lý bởi *JGentle IoC container* được ngầm hiểu như là các *beans*. Các *beans* đơn giản chỉ là các *objects* được khởi tạo và quản lý bởi *JGentle IoC container.* Những *beans* này có thể có hoặc không các *dependencies* giữa chúng, hoặc được cấu hình các thông tin chức năng khác nhau thông qua *Annotation* hoặc *Definition*, và được quản lý thông qua cơ chế cấu hình của *JGentle container*. Các *beans* được quản lý bởi JGentle *container* bao gồm việc khởi tạo, quản lý *dependencies*, *scope*, *interceptors*, v….vv. Hình thức khởi tạo *beans*, và các chức năng của *beans* sẽ được viết trong "dữ liệu cấu hình" và thông tin này sẽ được gửi cho *container* lúc *container* được khởi tạo. Nhờ đó ứng với mỗi *container* khác nhau các *beans* sẽ có những hình thái hoạt động cũng như chức năng năng hoàn toàn khác nhau.

4.2.2 - Khởi tạo một container

Các *container* trong JGentle được khởi tạo thông qua các hàm *static methods* của một *class* đặc biệt chỉ định đó là ***JGentle*** *class* nằm trong gói *package org.exxlabs.jgentle.context***.** Nội tại **JGentle** *class* cung cấp một số *static methods* cho việc khởi tạo các *container*, đồng thời cung cấp các *static methods* khác trợ giúp cho việc quản lý cấu hình các hệ thống *container* (phần này sẽ trình bày chi tiết trong Annotation Object Handling – AOH).

4.2.2.1 - InjectCreator

Bạn có thể khởi tạo một *InjectCreator container* đơn giản thông qua static method *buildInjectCreator* của *JGentle class* theo cách sau :

```
InjectCreator injector = JGentle.buildInjectCreator();
```

Việc triệu gọi method *buildInjectCreator* sẽ trả về một object *InjectCreator*

tương ứng với container. Các *overloading method* của method này cung cấp các tham số khởi tạo khác cho *InjectCreator container* , nếu như việc khởi tạo *InjectCreator* có đính kèm thông tin cấu hình.

4.2.2.2 - ServicesContext

Việc khởi tạo một *ServicesContext container* cũng tương tự như cách khởi tạo *injectCreator container.*

```
InjectCreator injector = JGentle.buildServicesContext();
```

Cũng tương tự như *buildInjectCreator method*, *buildServicesContext method* cũng có nhiều bản *overloading* khác nhau cung cấp các chức năng khởi tạo khác nhau.

4.2.3 - Quản lý cấu hình Configuration

Thông thường các *container* không được khởi tạo riêng lẻ một mình, mà luôn đính kèm các thông tin cấu hình chỉ định để container có thể quản lý các thông tin *dependencies, AOP, configurations*, … hoặc chỉ định cấu hình các *services* cho *ServicesContext*. Do đó các *container* cần được cung cấp các *"thông tin cấu hình"* cần thiết trước khi được khởi tạo. *"Thông tin cấu hình"* trong JGentle thực chất là các dữ liệu cấu hình được thiết lập thông qua việc triệu gọi các *"abstract method"* được chỉ định sẵn cung cấp bởi JGentle, do đó có thể hiểu "cách thức triển khai cấu hình" như là một tiến trình *"startup"* trước khi khởi tạo *container*.

Trong *JGentle*, *"thông tin dữ liệu cấu hình"* được cất trữ trong các *objects* được khởi tạo từ các *Class* chỉ định extends từ *AbstractConfig class hoặc implements từ Configurable interface.* Thay vì sử dụng XML, JGentle chủ yếu sử dụng *objects* của JAVA và *annotation* chuyển đổi thành *Definition*, vận hành như các phương tiện quản lý cũng như cất trữ thông tin *metadata*. Do đó khả năng cấu hình của *JGentle* rất uyển chuyển, đơn giản đến mức tối đa và tận dụng triệt để cú pháp của *language* (ở đây là JAVA).

4.2.3.1 - AbstractConfig Configuration (ACC)

JGentle cung cấp nhiều cách cấu hình khác nhau, nhưng hệ thống cấu hình chủ yếu và quan trọng nhất đó chính là *AbstractConfig Configuration (ACC)*. Thông qua ACC, nhà phát triển cần phải viết các thông tin cấu hình trong một *class* đặc biệt, gọi là *Configurable Class*. Một *configurable class* là một *abstract class*, là môi trường để soạn thảo các "công thức khởi tạo *bean*", khởi tạo *service*, … cho phép cất trữ các thông tin cấu hình thậm chí quản lý các dữ liệu thông tin cấu hình. Nếu như các dữ liệu thông tin cấu hình trước đây được cất trữ trong XML, và chỉ có thể truy vấn đến sau khi thông tin của file cấu hình (file XML) được nạp và diễn dịch vào trong *container*, thì giờ đây những thông tin này chỉ đơn thuần được cất trữ trong *Java object*, và sẵn sàng có thể truy vấn ngay cả khi đang thực hiện cấu hình lúc *run-time*, đồng thời có thể tận dụng triệt để được cú pháp của ngôn ngữ.

4.2.3.1.1 - Kế thừa từ AbstractConfig class

Một trong những cách đơn giản nhất để có thể hiện thực một *configurable class* là kế thừa lại từ *AbstractConfig class*, khi này một phần thông tin cấu hình trong JGentle sẽ được cấu hình trong phương thức *configure() method* được *override* lại từ *AbstractConfig*.

```
abstract class Config extends AbstractConfig {

	@Override
	public void configure() {
		// triệu gọi các phương thức cấu hình
	}
}
```

Ví dụ trên minh họa cách tạo một *configurable class* đơn giản, một *configurable class* đơn thuần chỉ là một *class* kế thừa từ *AbstractConfig class, nằm trong gói package org.exxlabs.jgentle.configure.jgentle.* Đây là một *abstract class*, chứa đựng các *method* thực thi các thao tác cấu hình cơ bản của JGentle. *Class* kế thừa từ *AbstractConfig class* bắt buộc phải *override* lại *configure method*, đây là phương thức xử lý chính cho các thao tác *config* trong JGentle. Khi một *configurable class* được add vào trong *container*, *container* sẽ tự

động *invoke method* này để khởi tạo các thông tin cấu hình.

Phần hiện thực của *configure method*, nhà phát triển có thể tận dụng các *method* được cung cấp sẵn trong *AbstractConfig class* để cấu hình các thông tin cần thiết (vd như *Dependency Injection*) hoặc cũng có thể triệu gọi các *method* tự tạo để thực thi các xử lý logic riêng kết hợp với các phương thức cấu hình.

Sử dụng thông tin *configurable class như sau*:

```
InjectCreator injector = Jgentle.buildInjectCreator(Config.class);

... hoặc ...

ServicesContext context = JGentle.buildServicesContext(Config.class);
```

Việc sử dụng thông tin được cấu hình chỉ đơn giản là đưa *configurable class* vào như là một đối số cho *buildInjectCreator* method dưới dạng *Object Class.*

Lưu ý rằng configurable class phải luôn là abstract class, vì lý do rằng trong nội tại của AbstractConfig class cung cấp rất nhiều methods cấu hình, một số đã được hiện thực hóa, một số vẫn còn ở mức abstract. Sau khi configurable class được add vào trong container, các methods này mới được container phát sinh các hiện thực cụ thể, nhờ vậy cho phép container trong những trường hợp khác nhau có thể chỉ định các hiện thực khác nhau mặc dù được chỉ định thực thi cùng một method cấu hình. Vì thế, trong trường hợp nếu như configurable class cố tình hiện thực cài đặt lại các abstract methods này, thì khi thực thi các xử lý cấu hình (invoke configure method), container sẽ nghiễm nhiên xem các hiện thực này là các methods hợp lệ và sẽ chỉ thực thi các methods này thay vì các methods cấu hình thực sự được phát sinh bởi container.

4.2.3.1.2 - Implements Configurable interface

Việc kế thừa lại từ *AbstractConfig class* để hiện thực một *configurable class* đôi khi gây nhiều bất tiện cho nhà phát triển, vì khi này *configurable class* sẽ không thể *extends* lại

được một *java class* nào khác. Để khắc phục vấn đề này, JGentle có cung cấp một *interface* tên là *Configurable* (nằm trong gói *package org.exxlabs.jgentle.configure.jgentle*), việc hiện thực *configurable class* thay vì *extends* từ *AbstractConfig class,* giờ đây chỉ cần đơn giản là thực thi *implements* interface này và cũng *override* lại *configure method* tương tự như khi ta làm việc với *AbstractConfig*.

```
abstract class Config implements Configurable {

	@Override
	public void configure() {
		// thực thi xử lý cấu hình
	}
}
```

Lưu ý rằng việc *implements Configurable interface* cũng tương tự như khi ta *extends* lại từ *AbstractConfig class*, cả 2 đều mang tính hình thức để giúp cho *configurable class* có thể có các *methods* xử lý cấu hình đúng khuông mẫu và đồng thời giúp cho *container* nhận dạng được *java class* hiện hành là một *configurable class*. Do đó, việc hiện thực hóa các *methods* trong *Configurable interface* lúc này cũng là không cần thiết, vì thế *configurable class* cũng nên là một *abstract class*.

4.2.3.1.3 - Chỉ định khối Config Block

Hệ thống cấu hình trong JGentle được thiết kế rất linh hoạt, và hướng đến khả năng mở rộng, nhờ đó cho phép nhà phát triển có thể mở rộng thêm các thao tác cấu hình cho *configurable class* của mình bằng cách chỉ cần đơn giản *implements* các *interface* cấu hình chỉ định. Thực chất, *Configurable interface* cũng chỉ là một trong những *interfaces* cấu hình đặc biệt trong JGentle, cung cấp các phương thức cấu hình như (quản lý DI, quản lý AOH,…) bên cạnh các *interfaces* cấu hình khác (vd như *EventServicesConfig interface* cung cấp các *method* cấu hình *EventServices*). Ngoài ra, để có thể mở rộng hệ thống cấu hình, thông qua AOH, nhà phát triển còn có thể tự xây dựng các *container* tự tạo, đồng thời tự định nghĩa các *interfaces* cấu hình, sau đó đưa vào trong JGentle *container*, hoạt động như là những *interfaces* cấu hình cung cấp để hiện thực các *configurable class*.

Chính vì thế nên một *configurable class* có thể *implements* một hoặc nhiều *interfaces* cấu hình khác nhau, của nhiều nhà phát triển khác nhau cùng kết hợp với các *interfaces* cấu hình được cung cấp sẵn trong hệ thống JGentle. Do các *interface* cấu hình đơn giản chỉ là các mô tả của các *methods* cấu hình, do đó không có bất cứ ràng buộc nào về việc định danh tên *(name)* và các *parameter* đi cùng của các *methods*. Điều này dẫn đến sẽ có thể có trường hợp nhiều *interfaces* cấu hình, trực thuộc những hệ thống *services* khác nhau nhưng lại có cùng định nghĩa chung một *method* (cùng *method name*, cùng *parameter* hay kiểu trả về). Nếu như *configurable class implements* cả 2 *interfaces* này cùng lúc, và khi thực thi cùng một *method* cấu hình, *container* sẽ không thể nào nhận biết được nên thực thi *method* nào ứng với *service object* nào. Trong trường hợp này, *container* sẽ lựa chọn "hiện thực của *method"* tương ứng với *interface "đầu tiên"* mà nó tìm thấy để thực thi xử lý. Hay nói cách khác, cài đặt còn lại của *method* kia (nếu có) sẽ không bao giờ có thể hoạt động. Điều này dẫn đến file cấu hình *configurable class* không thể thực hiện cấu hình đúng như nhà phát triển yêu cầu.

Để giải quyết vấn đề này, JGentle có cung cấp một *annotation* đặc biệt gọi là @Block *annotation* (nằm trong gói package *org.exxlabs.jgentle.configure.jgentle.annotation*) giúp nhận dạng một *method* được định nghĩa trong *configurable class* như là một khối *block* tương ứng với một nhóm các *methods* cấu hình liên quan dựa trên kiểu *type* của *interface* cấu hình cung cấp.

Việc sử dụng Block *annotation* như sau:

```
abstract class Config implements Configurable, EventServicesConfig, IA {

    @Override
    public void configure() {
        eventServiceConfig();
    }

    @Block(EventServicesConfig.class)
    void eventServiceConfig() {
        // Xử lý cấu hình
```

```
        }
}
```

@Block *annotation* cho phép chỉ định một danh sách các *Object Class*, tương ứng với các *interface* cấu hình mà *method* hiện hành muốn sử dụng thực thi cấu hình. Như ví dụ trên đây phương thức *eventServiceConfig* được chỉ định @Block *annotation* với thuộc tính là EventServicesConfig.**class,** điều này có nghĩa rằng khi các *method* cấu hình được triệu gọi trong thân xử lý của *eventServiceConfig method, container* sẽ ưu tiên cho các *methods* được mô tả bởi *EventServicesConfig interface* trước tiên thay vì là *Configurable interface hay IA interface*. Nếu trong trường hợp *EventServicesConfig interface và IA interface* mô tả cùng một *method* cấu hình giống nhau, *container* sẽ lựa chọn cài đặt hiện thực của *method* trên *EventServicesConfig* trước, và nếu như *EventServicesConfig* không có chỉ định *method* cấu hình cần yêu cầu xử lý, thì sau đó container mới tìm kiếm các *methods* phù hợp khác trên các *Configurable interface và IA interface.*

Trong trường hợp khi *eventServiceConfig method* đang xử lý, và có triệu gọi một *method* khác của *configurable class* để thực thi một tập hợp các thao tác cấu hình nhưng *method* này lại không chỉ định @Block *annotation* tường minh, thì *container* cũng sẽ áp dụng quyền ưu tiên thực thi trên *method* này tương tự như *eventServiceConfig method* khi được chỉ định @Block *annotation.*

```
abstract class config implements Configurable, EventServicesConfig, IA {

        @Override
        public void configure() {
                eventServiceConfig();
        }

        @Block(EventServicesConfig.class)
        void eventServiceConfig() {
                // Xử lý cấu hình
          ...
```

```
            otherConfig();
        }

        void otherConfig() {
                // Xử lý cấu hình
        }
}
```

Như ví dụ trên *otherConfig method* cũng sẽ bị ảnh hưởng bởi @Block *annotation* tương tự như *eventServiceConfig method.*

Bạn cũng có thể chỉ định quyền ưu tiên thực thi với nhiều *interface* cấu hình khác nhau trên cùng một *method* như sau:

```
abstract class config implements Configurable, EventServicesConfig, IA {

        @Override
        public void configure() {
                eventServiceConfig();
        }

        @Block({EventServicesConfig.class, IA.class})
        void eventServiceConfig() {
                // Xử lý cấu hình
        }
}
```

Trong trường hợp này thứ tự chỉ định *object class* của *interface* cấu hình chính là thứ tự ưu tiên thực thi phương thức cấu hình, và theo như ví dụ trên *container* sẽ ưu tiên thực thi *method* cấu hình trên *EventServicesConfig* trước rồi mới đến *IA interface,* nếu như cả 2 *interfaces* trên đều không có chỉ định *method* yêu cầu, *container* mới tìm kiếm đến các *interface* cấu hình còn lại (*Configurable Interface*).

4.2.3.2 - Annotation Type Configuration (ATC)

Ngoài cách cấu hình sử dụng class kế thừa từ *AbstractConfig class* theo *ACC hoặc implements Configurable interface*, JGentle còn cung cấp một dạng cấu hình khác dựa trên *Annotation Type*. Khi sử dụng *Annotation Type Configuration (ATC)*, tất cả thông tin bạn chỉ định cho việc cấu hình giờ đây chỉ hoàn toàn là *annotation*, và các định dạng cấu trúc của *annotation*.

Thông tin cấu hình giờ đây ngắn gọn và tuân thủ chặt chẽ cú pháp *annotation*, nhưng mặt bất lợi là không tận dụng được các điểm mạnh trong cú pháp của ngôn ngữ (*JAVA*) cũng như tận dụng các lợi thế của *OOP* (như kế thừa) trong khi cấu hình.

Để có thể sử dụng *ATC*, bạn chỉ cần khởi tạo một *Annotation* theo đúng cú pháp với tên tùy ý, và chỉ định cho *annotation class* của bạn với *@Configurations* annotation (nằm trong gói *package org.exxlabs.jgentle.configure.injecting.annoconfig*) :

```
@Retention(RetentionPolicy.RUNTIME)
@Configurations
@interface Configuration {
        // Chỉ định thông tin cấu hình
}
```

Điều cần lưu ý là *annotation* của bạn bắt buộc phải được chỉ định @Retention(RetentionPolicy.*RUNTIME*) *annotation* để thông tin *annotation* có hiệu lực lúc run-time.

Để có thể sử dụng được thông tin cấu hình với ATC, bạn có thể chỉ định ATC lên *configure() method* của *ACC class* hoặc sử dụng *usingConfig() method* của *AbstractConfig class* (nếu như *Annotation config* của bạn được chỉ định tại một nơi khác) theo cách như sau:

```
Abstract class Config extends AbstractConfig {
```

```
        @Configuration1 // sử dụng Annotation Type Configuration 1
        @Override
        public void configure() {
                // sử dụng Annotation Type Configuration 2
                 usingConfig(OtherClass.class.getAnnotation(Configuration2.class));
        }
}
```

Khi *Config class* được nạp vào *JGentle container*, *container* sẽ tự động tìm kiếm các *annotation* được chỉ định trên *configure() method* xem có phải là ATC hay không (được đính kèm @Configurations), nếu có sẽ thêm *annotation instance* đó vào một danh sách *Annotation Config List* và thực thi thông tin cấu hình.

Điểm lưu ý là mặt dù là một hình thái triển khai cấu hình khác với ACC nhưng ATC không hề thay thế ACC. ATC chỉ mang tính bổ sung cho *ACC* khi cần, cho phép chỉ định tách rời *annotation* với *logic code*, thông tin cấu hình thuần *annotation*, nhưng khi vận hành vẫn phải phụ thuộc vào dẫn xuất của **một** *ACC* chỉ định.

4.2.4 - Bean Scopes

Trong JGentle, để có thể khởi tạo một Bean, bạn cần phải viết các thông tin cấu hình định nghĩa *Bean*, điều này được thực hiện thông qua *configurable class* kết hợp với cách cấu hình *annotation* trực tiếp, hoặc gián tiếp thông qua *Definition*, hoặc thông qua cơ chế *ADI (Annotation Dependency Injection)*. Việc viết các thông tin cấu hình chính là viết ra các *"công thức"* để chế tạo *Bean*. Ý tưởng về *"công thức chế tạo Bean"* chính là điểm mấu chốt trong hầu hết các *"hệ thống quản lý Beans",* vì với cùng một *Class* nhưng với các *"công thức khởi tạo Bean"* khác nhau bạn sẽ có được các *Beans* ứng với các chức năng hoàn toàn khác nhau. *"Công thức chế tạo Bean"* hay *"thông tin cấu hình"* chính là cốt lõi của sự khác nhau giữa các *Beans*.

Trong JGentle, ngoài việc bạn có thể định nghĩa các thông tin *"công thức"* để quản lý các *dependencies* khác nhau của *Bean*, chỉ định các thành phần *AOP* trên Bean, thay thế

hoặc bổ sung các chức năng cho *Bean* thì bạn còn có thể định nghĩa *Scope* (phạm vi hoạt động) của *Bean*. Việc ràng buộc các *beans* được tạo ra vào một trong các *scope* chỉ định sẽ giúp cho việc quản lý *beans* trở nên dễ dàng và đơn giản, linh hoạt trong việc quản lý được *bean lifecycle*, cũng như mềm dẻo trong việc kiểm soát các đặc tính *stateless* hoặc *stateful* của *bean.* Ngoài ra, JGentle còn cho phép có thể tự định nghĩa riêng các *Scope*, và sử dụng chúng như là các *Scope* hệ thống có thể chỉ định cho bất cứ *bean* nào được quản lý bởi *JGentle container*.

Các *Scope* được hỗ trợ bởi JGentle bao gồm:

Scope	Mô tả
Singleton	Là *scope* của một định nghĩa Bean tương ứng chỉ một *object instance* ứng với một *JGentle IoC container.*
Prototype	Là *scope* của một định nghĩa Bean tương ứng với nhiều *object instances.*
Request	Là *scope* của một định nghĩa Bean tương ứng với một vòng đời lifecycle của một *HTTP request*. Có nghĩa là mỗi *HTTP request* sẽ có một object *instance* tương ứng được khởi tạo từ định nghĩa Bean. Giá trị chỉ định của scope này chỉ có hiệu lực trong trường hợp *JGentle container* đang sử dụng là *WebContext*.
Session	Là *scope* của một định nghĩa Bean tương ứng với một vòng đời *lifecycle* của một *HTTP session.* Giá trị chỉ định của scope này chỉ có hiệu lực trong trường hợp *JGentle container* đang sử dụng là *WebContext*.
Application	Là *scope* của một định nghĩa *Bean* tương ứng với vòng đời của một *HTTP ServletContext.* Giá trị chỉ định của *scope* này chỉ có hiệu lực trong trường hợp *JGentle container* đang sử dụng là *WebContext*.

4.2.4.1 - Singleton Scope

Khi bean được chỉ định là một *singleton bean*, thì chỉ có duy nhất 1 và chỉ 1 *object instance* sẽ được *JGentle container* quản lý, bất cứ yêu cầu nào cần truy vấn đến Bean, thì *JGentle container* cũng chỉ trả về duy nhất một *object instance* chỉ định như trên. Hay nói cách khác, khi bạn chỉ định thông tin cấu hình của một định nghĩa *bean* ứng với *scope* là *singleton*, thì *JGentle container* sẽ chỉ khởi tạo chỉ duy nhất 1 *object instance* tương ứng với định nghĩa *bean* đó, sau đó *object instance* trên sẽ được cất trữ tại *cache* chứa các *singleton beans* của container, và khi bất cứ yêu cầu trả về bean nào được gửi đến, *container* sẽ tìm kiếm trong *cache* và trả về *object instance* chỉ định.

Chú ý rằng khái niệm *singleton* của bean trong JGentle không giống hoàn toàn như khái niệm *singleton pattern.* Trong khi *singleton pattern* khởi tạo duy nhất một và chỉ một *object instance* tương ứng với mỗi *Class* được nạp lên bởi 1 *classloader* chỉ định, thì *singleton bean* trong JGentle có nghĩa là mỗi 1 container sẽ chỉ có 1 và chỉ 1 *object instance* được khởi tạo ứng với 1 định nghĩa bean. Hay nói cách khác, khái niệm *singleton scope* trong JGentle được hiểu như là "*1 container 1 bean*".

Trong JGentle, mặc định nếu như bạn không chỉ định bất cứ thông tin gì về *scope*, *scope* của định nghĩa *bean* chỉ định sẽ là *singleton*. Để có thể định nghĩa *singleton scope* trong JGentle bạn có thể viết cấu hình như sau:

```
abstract class Config extends AbstractConfig {
        @Override
        public void configure() {

                attach(IAClass.class).to(AClass.class).scope(Scope.SINGLETON);

                ... hoặc ...

                bindBean().to().inClass(AClass.class,
                                "ID_AClass").scope(Scope.SINGLETON);
        }
}
```

4.2.4.2 - Prototype Scope

Khi "định nghĩa bean" chỉ định là *prototype scope*, điều này có nghĩa là mỗi khi có yêu cầu trả về bean được gửi đến, *container* sẽ khởi tạo một *bean* mới tương ứng. Hay nói cách khác, *container* sẽ luôn tạo mới bean mỗi khi có yêu cầu cần đến *bean*, yêu cầu có thể là truy vấn trực tiếp thông qua *getBean() method* của *container* hoặc yêu cầu cũng có thể là truy vấn để *inject một prototype-scoped bean* chỉ định này vào một bean khác. Tùy theo ứng dụng, bạn có thể dùng *prototype scope* cho tất cả các beans là *stateful*, trong khi với *singleton scope* bạn có thể sử dụng cho các *stateless beans*.

> ***Enum Scope*** *(có thể tìm thấy tại package* **org.exxlabs.jgentle.configure.injecting.enums***) chứa đựng thông tin về các loại scope được hỗ trợ bởi* ***JGentle Container.***

Khi triển khai một *bean* trong *prototype scope*, một điều cần lưu ý đó là *JGentle container* sẽ không quản lý toàn bộ vòng đời *(lifecycle)* của một *prototype bean*. *Container* sẽ khởi tạo, cấu hình và thực thi các chức năng DI, vân vân ... trả *bean* về cho *client*, và sau đó không còn cất giữ bất kì thông tin, hay *reference* gì về *prototype instance*. Điều này có nghĩa rằng client sẽ phải tự chịu trách nhiệm về việc xóa bỏ các *objects, beans* chỉ định là *prototype scope* sau khi sử dụng, cũng như tự chịu trách nhiệm về việc giải phóng các tài nguyên khác mà *prototype bean* nắm giữ.

Để có thể định nghĩa *scope* trong JGentle bạn có thể viết cấu hình như sau:

```
abstract class Config extends AbstractConfig {
        @Override
        public void configure() {

                attach(IAClass.class).to(AClass.class).scope(Scope.PROTOTYPE);

                ... hoặc ...

                bindBean().to().inClass(AClass.class,
```

```
                        "ID_AClass").scope(Scope.PROTOTYPE);
    }
}
```

4.2.4.2.1 - Singleton-scoped bean có dependencies đến Prototype-scoped bean:

Không như các *IoC container* khác (vd Spring), các *dependencies* được thực hiện chỉ định một lần duy nhất tại thời điểm *creation-time* (thời điểm khởi tạo bean), *JGentle IoC Container* cho phép bạn chỉ định việc *inject* các *dependencies* tại cả 2 thời điểm khác nhau: *Creation-Time* và *Invocation-Time* (thời điểm *bean* được triệu gọi – *invoke để* xử lý một thực thi nào đó). Đối với các hệ thống *IoC container* chỉ xử lý *dependencies* tại thời *điểm creation-time*, khi sử dụng các *singleton-scoped beans* có *dependencies* đến các *bean* chỉ định *scope* là *prototype (prototype-scoped bean)*, cần phải lưu ý rằng các *dependencies* đó chỉ được chỉ định ngay tại lúc khởi tạo. Điều này có nghĩa là nếu như có một hành vi *inject* một *dependency* là một *prototype-scoped bean* vào trong một *singleton-scoped bean*, thì khi này một *prototype bean* **mới** sẽ được *container* tạo mới tương ứng, và sau đó được *inject* vào *singleton bean* chỉ khi *singleton bean* này được khởi tạo … và tất cả chỉ có như thế. Lúc này *prototype-scoped bean* chỉ định sẽ mất hoàn toàn ý nghĩa là một *prototype*, vì bị ràng buộc chặt chẽ *"wiring" (về mặt khía cạnh scope)* với một *singleton-scoped bean,* như thế *prototype-scoped bean* khi này hoạt động như thể là một *singleton-scoped bean.*

Do đó nếu như bạn muốn rằng *singleton-scoped* bean của bạn luôn luôn thu được một *dependency* là *instance* mới nhất của một *prototype-scoped bean* chỉ định tại thời điểm *run-time*, thì bạn cần chỉ định thuộc tính *invocation* của hành vi *inject* là **true**. (vui lòng xem chi tiết tại phần mô tả thành phần *Deep Dependency Injection*). Khi thuộc tính này được chỉ định là **true**, điều này có nghĩa là mỗi khi bean của bạn được *invoke* (lúc khởi tạo hoặc một lời triệu gọi *method* …), *container* sẽ tự nhận biết được các *dependencies* đang được chỉ định trong bean hiện hành, nếu *dependencies* có ràng buộc yêu cầu thiết lập *dependencies* tại thời điểm *invocation*, *container* sẽ thực thi "lại" việc *inject dependency* (đảm bảo *dependency* được sử dụng tại thời điểm *run-time* luôn được tạo mới mỗi khi triệu gọi).

4.2.4.3 - Request Scope

Các bean được chỉ định là *request scope* chỉ có thể được sử dụng trong trường hợp *container* hiện hành là *WebContext*. Một bean khi được chỉ định là *request scope*, điều này có nghĩa rằng *JGentle container* sẽ khởi tạo một *instance* mới và sử dụng cho mỗi *HTTP request*, hay nói cách khác rằng *bean* sẽ chỉ có tầm vực ảnh hưởng trong một *HTTP request*. Bạn có thể thay đổi trạng thái của bean (nếu muốn) một cách an toàn mà không cần quan tâm đến các *request* khác được gửi đến, do mỗi *request* đều được chỉ định khởi tạo một *bean* tương ứng theo định nghĩa *bean* chỉ định, và hoàn toàn độc lập với *request-scoped bean* mà bạn đang làm việc. Khi một *request* kết thúc xử lý, thì những *beans* được chỉ định trong *scope* của *request* sẽ được tự động gỡ bỏ.

4.2.4.4 - Session Scope

Các bean được chỉ định là *session scope* chỉ có thể được sử dụng trong trường hợp *container* chỉ định là *WebContext*. Một *bean* khi được chỉ định là *session scope*, điều này có nghĩa là *JGentle container* sẽ khởi tạo một *instance* mới và sử dụng trong tầm vực của mỗi *HTTP session*. Cũng tương tự như *request-scoped beans*, bạn có thể sử dụng, thay đổi trạng thái của bean một cách an toàn mà không ảnh hưởng đến các tầm vực *HTTP session* khác, do mỗi *session* đều được chỉ định khởi tạo một bean tương ứng theo định nghĩa bean chỉ định, và cũng hoàn toàn độc lập với *session-scoped bean* mà bạn hiện đang làm việc. Khi một *HTTP session* hết hạn *Time-Out*, hoặc *session* bị chủ động gỡ bỏ, những *beans* mà được chỉ định trong *scope* của *session* cũng sẽ được tự động gỡ bỏ.

4.2.4.5 - Application Scope

Tương tự như *request scope* và *session scope*, các *bean* được chỉ định là *session scope* chỉ có thể được sử dụng trong trường hợp *container* là *WebContext*. Khi một bean được chỉ định là *Application scope*, tương tự như trên, nhưng thay vì *bean* sẽ được *JGentle container* khởi tạo ở mức *request* hoặc *session*, giờ đây *bean* được chỉ định khởi tạo mức *Application* của *ServletContext* của ứng dụng *web* hiện hành. Cách làm việc cũng như hoạt động của *application-scoped bean* giờ đây được chỉ định ở mức của ứng dụng, có thể được sử dụng và truy xuất tại bất kì đâu trong *web application*.

4.2.4.6 - Custom Scope

Ngoài các "*scope* hệ thống" mặc định được chỉ định trong JGentle, nhà phát triển có thể tự xây dựng các *scope instances* tùy biến, để có thể tự quy định cũng như tự định nghĩa phạm vi hoạt động của *bean* khởi tạo tương ứng. Thông tin về các *custom scope* được chỉ định trong JGentle được xây dựng như thể là một JGentle *bean* đơn thuần và có thể sử dụng tương tự như các *scope* hệ thống. Do đó các *custom scope* có thể tái sử dụng trong các *container* khác nhau của JGentle mà không hề gặp bất kì trở ngại nào.

Chức năng tùy biến *scope* trên các *custom scope* cung cấp cho nhà phát triển khả năng tự *"phân vùng hoạt động"* của các *beans* sẽ khởi tạo, khả năng tự điều phối *life-cycle bean* tùy biến, không cần tuân theo một phạm vi cố định của hệ thống đặt ra, đồng thời cho phép tái sử dụng các định nghĩa *scope* khác nhau trong những trường hợp chỉ định của các *container* khác nhau, tránh khỏi việc bị ràng buộc chặt chẽ vào các quy định của một hệ thống *scope* trong một *container* tương ứng.

4.2.4.6.1 - Khởi tạo và định nghĩa custom Scope

Như đã nói ở trên, thông tin của một *custom scope* chỉ đơn giản được chỉ định tương tự như là một *object bean* được thêm vào trong *container*. Để có thể khởi tạo một *object scope* chỉ cần đơn giản là *implements ScopeImplementation interface* (nằm trong gói *package org.exxlabs.jgentle.context.injecting.scope*). Bất kì một *Scope Instance* nào được chỉ định trong JGentle *container*, muốn có thể sử dụng trong lòng hệ thống JGentle đều nhất thiết cần phải *implements interface* này. *ScopeImplementation* một mặt chỉ định cho hệ thống JGentle biết rằng *bean* khởi tạo sẽ là một *scope*, đồng thời một mặt chỉ định các *method* cài đặt cần thiết để một *scope* thực thi.

ScopeImplementation bao gồm một số *method* cài đặt như sau:

```
Object getBean(Class<?> classInstance, String ID, String nameScope,
                    ObjectBeanFactory objFactory);

Object remove(String nameScope, ObjectBeanFactory objFactory)
                    throws InvalidRemovingOperationException;
```

```
Object putBean(String nameScope, Object bean, ObjectBeanFactory objFactory)
                    throws InvalidAddingOperationException;
```

Phương thức getBean(): được chỉ định sẽ chịu trách nhiệm khởi tạo và trả về *object bean*, nếu *object bean* hiện tại chưa được khởi tạo trong *scope* hiện hành. Ngược lại nếu *scope* hiện hành đã tồn tại *object bean* tương ứng với *name scope* cung cấp, cài đặt của *method* này cần phải trả về *object bean* từ *scope* được hiện thực.

Phương thức remove(): được chỉ định sẽ chịu trách nhiệm gỡ bỏ một *object bean* ứng với một tên định danh *name scope* cung cấp ra khỏi *scope* hiện hành. Trong trường hợp không tìm thấy *object bean* tương ứng, kết quả trả về sẽ là một giá trị ***null***. Lưu ý rằng hiện thực cài đặt trên *callback method* này sẽ được *container* tự động triệu gọi tại thời điểm ***run-time*** khi cần thiết hoặc cũng có thể được chủ động triệu gọi từ phía *caller* bất kì.

Phương thức putBean(): cài đặt của *method* này được chỉ định việc thực thi *add* một *object bean* vào trong *scope* hiện hành. *Method* này cũng sẽ được triệu gọi tự động bởi *container* khi cần thiết (ví dụ: khi có một *outject request* một *object bean* vào *scope*), tuy nhiên *method* này vẫn có thể được chủ động triệu gọi từ phía *caller* bất kì.

Vì rằng *scope* là một thể hiện đối tượng ở mức toàn cục trong *container* cho nên khi được triệu gọi thực thi (*invoke*) nhất thiết phải là *Singleton* và được *thread-safe*. Điều này nên cần được tuân thủ chặt chẽ, để đảm bảo các *object beans* được trả về hoặc được chỉ định khởi tạo trên nhiều *thread* khác nhau của cùng một hiện thực *scope* được thực thi đúng đắn. Hay nói cách khác, việc chủ động triệu gọi các *caller methods* trên hiện thực của *ScopeImplementation* cần được đảm bảo an toàn tuyến đoạn và đối tượng *scope instance* nên là *Singleton*. Công việc này được JGentle giao lại hoàn toàn cho người thực thi triển khai cài đặt *scope* trên *ScopeImplementation interface* đảm nhận. Và theo như mặc định, bất cứ khi nào có yêu cầu triệu gọi một *scope* từ phía *container* (JGentle), *container* đều nắm lấy ***lock*** của chính đối tượng *ScopeImplementation*.

4.2.4.6.2 - Sử dụng custom Scope

Việc sử dụng một *custom scope* khá đơn giản

4.2.5 - Quản lý Life-cycle Bean

4.2.5.1 - Init và Destroy

4.2.5.1.1 - Initializing Bean

4.2.5.1.2 - Disposable Bean

4.2.6 - Tương tác container

4.2.6.1 - InjectCreatorAware

4.2.6.2 - Automatic Detector

4.2.6.3 - Before Configure

4.2.6.4 - Before Init Context

4.2.6.5 - Component Service Context

4.3 - Definition

4.3.1 - Giới thiệu

Toàn bộ các thông tin cấu hình, *metadata* được sử dụng trong hệ thống JGentle đều dựa trên *Definition*, JGentle không lựa chọn cách sử dụng *annotation* trực tiếp tại thời điểm *run-time*, mà truy vấn thông tin *annotation* gián tiếp thông qua *Definition* để có thể tránh khỏi những hạn chế khi vận dụng *annotation* (ví dụ như thông tin cấu hình *annotation* bắt buộc phải dính liền với *logic code*). Việc tách rời được thông tin *metadata* và *logic code* sẽ đảm bảo được tính tái sử dụng của thông tin *metadata*, mềm dẻo hơn trong xử lý cũng như mở rộng hệ thống về sau.

Đối với một số hệ thống *lightweight container* như *Spring, Hibernate* ... điều này đã được giải quyết rất tốt, do các hệ thống này sử dụng XML như là công cụ chính để quản lý thông tin cấu hình thay vì *annotation*. Trong trường hợp này XML đóng vai trò như một thành phần tách rời (một file *text* thô tuân thủ một cú pháp XML nhất định do các *container* đề ra), việc quản lý cũng như triệu gọi, xử lý thông tin cấu hình từ file XML sẽ do *container* đảm nhiệm. Điều này mang lại nhiều lợi ích trong việc quản lý thông tin cấu hình do thông tin *metadata* và *logic code* đã được tách rời chuyên biệt nhưng cũng kèm theo những hạn chế về mặt *security* do tính không an toàn của file XML, cũng như thông tin cấu hình quá dài dòng, dẫn đến việc hệ thống tràn ngập các file XML (khó quản lý).

JGentle đã tránh được các vướng mắc của cả *annotation* và *XML* khi đề ra *Definition*, đóng vai trò như là một thành phần *object* có trong hệ thống, chuyên biệt, chịu trách nhiệm quản lý, cất trữ thông tin cấu hình. Việc vận dụng cũng như xử lý thông tin cấu hình giờ đây dễ dàng hơn bao giờ hết, thông tin cấu hình uyển chuyển có thể thay đổi ngay cả khi *run-time*, tách rời với *logic code* đồng thời vẫn tuân thủ các cú pháp truy xuất cơ bản chuẩn của *annotation*. Do đó việc sử dụng, truy vấn *Definition* hoàn toàn tương tự như *annotation*.

4.3.2 - Annotation Configuration vs XML Configuration

Hiện nay, một vài hệ thống *container* đã đưa *annotation* vào sử dụng, nhằm thay thế 1 phần cách thức cấu hình với *XML* truyền thống, một vài hệ thống khác lại lựa chọn cách thức kết hợp cả 2 hệ thống cấu hình (vừa *XML*, vừa *annotation*) để trợ giúp cách thức triển

khai cấu hình (như Spring với Spring Java-Config). Tuy nhiên, bản chất của *annotation* là s*ource-code metadata*, do đó thông tin cấu hình bắt buộc phải dính liền với *source code,* và xét trên phương diện tính mềm dẻo và tùy biến, XML hoàn toàn có lợi thế so với *annotation* do thông tin hoàn toàn được chỉ định tách rời với *logic code*. Ngoài ra, mặc dù hệ thống triển khai với *annotation* đơn giản trong cấu hình, dễ dàng *debug*, *test* cũng như *validate* nhưng thực sự vẫn chưa thể thay thế hoàn toàn XML do thông tin dữ liệu cấu hình vẫn bị ràng buộc chặt chẽ bởi một số giới hạn của một số quy tắc đặt ra trong *annotation* (vd trong một thực thể không được phép chỉ định "nhiều hơn 1" *annotation* giống nhau).

Chính vì nhằm giải quyết các giới hạn cố hữu tồn tại bởi cả hai cách thức cấu hình, nên JGentle đưa ra Definition (vừa là một cơ chế, vừa là một loại đối tượng cất trữ thông tin dữ liệu cấu hình), xây dựng dựa trên cơ sở *annotation,* khắc phục những hạn chế hiện có của *annotation* đồng thời trợ giúp việc quản lý thông tin cấu hình tại thời điểm *run-time*.

4.3.2.1 - So sánh giữa annotation configuration và XML configuration

Annotation Configuration	XML Configuration
Annotation Objects, Java Objects	String Identifiers
Config ngắn gọn, nhưng không rõ ràng	Config dài dòng, nhưng rõ ràng
Thông tin *configuration* được *compiled*	Thông tin *configuration* phải được *interpretered*
Xử lý nhanh với dữ liệu config	Xử lý chậm do cần phải convert sang java objects
Tùy biến với các config dạng loop hoặc condition	Chỉ định bắt buộc tường minh (không loop hay có condition)
Dính liền với logic code	Tách rời với logic code, do được chỉ định trên file XML.

4.3.2.2 - 5 lý do không sử dụng Annotation trực tiếp

- *Annotation* không cung cấp cơ chế *validate* lúc *run-time*.
- Thông tin *annotation* không thể thay đổi lúc *run-time.*
- *Annotation* không có cơ chế quản lý thông tin tổng thể (thông tin *annotation* được chỉ định trên từng cá thể riêng biệt).

- Thông tin của *annotation* phải dính liền với *logic code*.
- Trên một thực thể chỉ định không thể định nghĩa "nhiều hơn 1" *annotation* giống nhau.

4.3.2.3 - 5 lý do sử dụng Definition thay thế XML

- *XML* không có cơ chế *validate* dữ liệu thông tin cấu hình (cả lúc *compiled* lẫn *run-time*).
- Thông tin cấu hình hoàn toàn là *String Identifier.*
- Thông tin *config* rõ ràng nhưng quá rườm rà, dài dòng, khó kiểm soát cũng như *debug*. *(XML Hell).*
- Không hỗ trợ *condition config* hoặc *loop config.*
- Thông tin cấu hình hoàn toàn là *text* thô, không đảm bảo *security* nếu như không có hỗ trợ từ *container*.

4.3.2.4 - Definition

- Xây dựng dựa trên một tập hợp các thông tin *AnnoMeta* (*Annotation Metadata*) được chuyển đổi từ thông tin của *annotation*. Các thông tin *annotation* sẽ được JGentle *container* chuyển đổi thành *Definition* khi chúng được nạp vào *container*.
- Hỗ trợ *validate* lúc *run-time* (kết hợp với JGentle *container*, *Annotation Object Handling - AOH*), do đó đảm bảo thông tin sau khi chuyển đổi là thông tin đã sẵn sàng sử dụng, đồng thời cung cấp một cơ chế linh hoạt để *validate* dữ liệu *annotation*.
- Giữ nguyên cách cấu hình theo cách của *annotation* truyền thống, chỉ thay đổi thông tin dữ liệu cấu hình chuyển đổi thành *Definition*. Do đó đảm bảo tương thích với các hệ thống sử dụng *annotation*, việc chuyển đổi thành *Definition* được đơn giản hóa đến mức tối đa.
- Xây dựng dựa trên nền *annotation* do đó tuân thủ những quy tắc cấu hình chuẩn của *annotation* từ *JDK5.0* trở đi.
- Cho phép thay đổi thông tin cấu hình lúc *run-time* (điều bị hạn chế trong *annotation*).
- Tách biệt thông tin *annotation code* và *logic code.*
- Không như *annotation* khi sử dụng phải được chỉ định trên 1 thực thể chỉ định.

Definition có thể là tập hợp các thông tin định nghĩa trên 1 *class* và tất cả những *member* (*Fields* hoặc *Methods*) của class đó. Cung cấp một cơ chế cấu hình, quản lý thông tin cấu hình tổng thể ở mức *Class*.

- *Definition* là pure java.
- Cung cấp một hệ thống các *annotation* có sẵn trong JGentle container, chịu trách nhiệm *validate* thông tin các *AnnoMeta* trong *Definition*.

4.3.3 - Sử dụng Definition

Như đã mô tả ở trên, Definition vừa là một loại đối tượng cất trữ thông tin dữ liệu cấu hình, vừa là một cơ chế "cục bộ" trong cơ chế "tổng thể" của JGentle (bao gồm cả AOH) được sử dụng để quản lý thông tin cấu hình tại "thời điểm *run-time*". Do đó, thông tin được quản lý bởi Definition cũng là một loại dữ liệu "*run-time*", không giống như *annotation* truyền thống (nhờ vào việc tích hợp trực tiếp trong *source code*), thông tin trong *annotation* có thể truy xuất tại thời điểm *compile* hoặc *deploy.* Chính vì đặc điểm này, Definition không thay thế hoàn toàn *annotation* mà vẫn giữ nguyên các cách thức, cũng như nguyên tắc đặt ra trong *annotation*, và được vận dụng như thể là một cơ chế trợ giúp đi kèm với *annotation* trong việc thực thi cấu hình tại lúc *run-time*.

4.3.3.1 - Chuyển đổi thông tin annotation thành Definition

Trong trường hợp nếu sử dụng *DI, dDI, AOP*, … hay bất kì *services* quản lý *bean* nào được cung cấp bởi *JGentle container*, mặc định tất cả các *object instances*, *beans* được quản lý bởi *container* đều đã được chuyển đổi thông tin *annotation* (được chỉ định trong *Object Class* khởi tạo của beans nếu có) thành *Definition*. Vui lòng xem phần mô tả chi tiết của các thành phần *core services* của JGentle để có thể truy xuất và chỉnh sửa các thông tin *Definition* này.

Còn trong trường hợp chỉ sử dụng *JGentle container* để quản lý *Definition* hoặc tự xây dựng *services container* trên nền *AOH container*, bạn có thể sử dụng *Definition* theo cách như sau:

```
// Khởi tạo InjectCreator container
```

```
InjectCreator injector = JGentle.buildInjectCreator();

// Lấy ra đối tượng Definition Manager
DefinitionManager defManager = injector.getDefManager();

// Chuyển đổi thông tin annotation có trong MyClass class
defManager.loadDefinition(MyClass.class);

// Lấy ra thông tin Definition
Definition def = defManager.getDefinition(MyClass.class);
```

Trong JGentle, mỗi *container* đều có 1 *object* quản lý *Definition* gọi là *Definition Manager (DM)*. Đối tượng *DefinitionManager* quản lý tất cả thông tin *Definition* trong *container* hiện hành, bao gồm các thực thi khởi nạp thông tin *annotation*, chuyển đổi thành *Definition*, *validate* thông tin *annotation* trước khi chuyển đổi, đăng kí hoặc gỡ bỏ các *annotation validator* ..v..vv... *DM* chịu trách nhiệm như là thành phần trung tâm trong mỗi *container*, nhận và thực thi tất cả mọi yêu cầu xử lý liên quan đến *Definition*. Để có thể lấy ra được đối tượng *DM* của một *container* chỉ cần đơn giản triệu gọi phương thức *getDefManager*() method trên *container* đó.

*Lưu ý rằng các methods hoặc fields là **static** nếu có chỉ định annotation cũng sẽ được diễn dịch như các method, field thường khác, ... hay nói cách khác là không có ngoại lệ cho các **static instance***

Để có thể chuyển đổi thông tin *annotation* có trong 1 *object class* chỉ định chỉ cần triệu gọi *loadDefinition() method* với object class chỉ định là tham số truyền. Sau khi thông tin *annotation* trong *object class* chỉ định đã được chuyển đổi thành *Definition*, việc sử dụng *Definition* chỉ cần đơn giản triệu gọi phương thức *getDefinition()* của DM.

Với cách khởi nạp *Definition* sử dụng *object class* (hoặc *object method*, *object field*) trực tiếp, ta chỉ có thể có 1 *Definition* ứng với 1 thực thể chỉ định. Để có thể khởi nạp nhiều *Definition* khác nhau của cùng một thực thể, *DM* cung cấp nhiều *overloading method* của *loadDefinition()* cho phép khởi nạp *Definition* ứng với một tên định danh ID chỉ định. Và đồng thời việc lấy ra *Definition* giờ đây cũng sẽ thực thi thông qua ID, cú pháp như sau:

```
void loadDefinition(Class<?> clazz, String ID)
```

Với cách khởi nạp như trên, giờ đây *Definition* của *object class* chỉ định sẽ tương ứng với một tên định danh ID, việc truy xuất thông tin *Definition* cũng chỉ có thể truy xuất được thông qua ID của *Definition*. Tên định danh ID của *Definition* là duy nhất ứng với một *container*, do đó với một *object class* bạn có thể thực thi nhiều khởi nạp *Definition* khác nhau tại nhiều thời điểm khác nhau ứng với các ID riêng biệt. Bạn có thể thay đổi, chỉnh sửa, tùy biến … thông tin *Definition* ứng với một ID chỉ định mà không hề ảnh hưởng đến các định nghĩa *Definition* tương ứng khác.

Với định danh ID, JGentle đã giải quyết trọn vẹn bài toán 1 thực thể chỉ định có thể định nghĩa nhiều thông tin *annotation* khác nhau (tại thời điểm run-time), không như *annotation* truyền thống chỉ có thể định nghĩa 1 thông tin duy nhất trên 1 thực thể. Hơn thế nữa, với *ADI (Annotation Dependency Injection)*, JGentle container còn cho phép các thông tin *annotation* chỉ định có thể định nghĩa rời rạc rồi sau đó được *inject* vào *bean* tại thời điểm *configuration* (xem thêm tại đây). Ngoài ra để có thể tùy biến *Definition*, JGentle *container* còn cung cấp các phương thức khác cho phép bổ sung thông tin cho *annotation* ứng với một ID của một *Definition* cụ thể được chỉ định như sau:

Class:

```
void loadDefinition(String ID, Class<?> clazz, Annotation annotation)
```

Method:

```
Definition loadDefinition(String ID, Method method, Class<?> clazz,
                          Annotation annotation)
```

Field:

```
Definition loadDefinition(String ID, Field field, Class<?> clazz, Annotation annotation)
```

Khi dữ liệu *annotation* chỉ định ở mức class, ta chỉ có 3 tham số, tham số thứ 1 chỉ tên định danh ID của *Definition*, tham số thứ 2 chỉ định *object class* của *class source*, và tham số thứ 3 là *annotation* chỉ định sẽ chuyển đổi thành *Definition*. Với dữ liệu *annotation* chỉ định ở mức *method* và *field*, ta cần chỉ định thêm tham số *method* và *field* tương ứng với

method và *field* nào của *object class* chỉ định của *annotation*.

Thông thường thì việc can thiệp trực tiếp thế này ít khi được sử dụng đến do các *services* trong *JGentle container* đã cung cấp sẵn các cơ chế cấu hình giúp chúng ta chỉ định các thông tin trên ở mức *configuration*. Nhưng trong một số trường hợp, bạn có thể muốn sử dụng *Definition* một cách mềm dẻo cho các *services* tự tạo hoặc với một lý do nào đó thì với các *overloading methods* trên, ta hoàn toàn có thể chỉ định bằng tay *annotation* nào ứng với *class, method, field* nào và tên định ID là gì sẽ được chuyển đổi thành *Definition* tương ứng.

Thêm vào đó, ngoài việc cung cấp việc sử dụng *Definition* như là thành phần cất trữ *metadata* ở mức tổng thể (mức *class*), JGentle vẫn cho phép chỉ định *Definition* ở từng thành phần *method*, hoặc *field* thông qua 2 *method* sau:

```
void loadDefinition(Field field)

và

void loadDefinition(Method method)
```

Tuy nhiên khi chỉ định *Definition* ở mức này (*method, field*) bạn sẽ không thể chỉ định được các *Definition* ứng với các ID khác nhau. Hay nói cách khác *Definition ID* trong *JGentle* hiểu như là ID của một *Class* (định nghĩa ở mức *class*), *JGentle* không phân biệt *Definition* của *method, field* như là các thành phần định nghĩa tách rời trừ khi nó được chỉ định trong một *object class* cụ thể. Nếu so sánh *Definition ID* của JGentle với các *light-weight container* khác như Spring, … có thể xem ở mức độ nào đó ID của *Definition* trong JGentle tương đương như ID của định nghĩa "*Bean*" trong Spring.

4.3.3.2 - Truy vấn thông tin Definition sau khi đã được khởi nạp

Việc lấy ra thông tin *Definition* sau khi đã được khởi nạp chỉ cần đơn giản triệu gọi phương thức *getDefinition() method* của Definition Manager. Trong trường hợp *Definition* được khởi nạp ứng với một ID chỉ định, tham số cần truyền cho *getDefinition() method* là

chuỗi tên định danh của ID, còn nếu như *Definition* không được chỉ định ID, tham số truyền phải là thực thể chứa *Definition* được khởi nạp (thực thể chỉ định chỉ có thể là *Object Class, Method instance, hoặc Field instance*). ID hay *Object Class (Method, Field)* đều là những định danh duy nhất cho *Definition* do đó khi khởi nạp một *Definition* bạn cần phải chỉ định rõ định danh duy nhất này của *Definition* cho container.

Đối tượng trả về của *getDefinition() method* là một Definition, một *object* mô tả tương ứng với *Definition* trả về. Đối tượng Definition cho phép thao tác vào bên trong *Definition*, đọc, truy vấn thông tin *Definition*, lấy ra dữ liệu thông tin *annotation* gốc … thậm chí thay thế, chỉnh sửa dữ liệu *metadata* trong *Definition*, … Với khả năng cất trữ thông tin *metadata* tương tự *annotation,* bạn có thể chỉ định cho *Definition* quản lý bất cứ thông tin gì với cách làm việc hoàn toàn tương đồng như khi làm việc với *annotation*.

```
// Lấy ra thông tin Definition
Definition def = defManager.getDefinition(MyClass.class);
```

Như đã nói ở trên, ngoài việc quản lý thông tin *metadata* ở mức tổng thể (mức *class*), *Definition* còn cho phép quản lý thông tin *metadata* ở mức cục bộ (mức *method, field*). Do đó một đối tượng *Definition* có thể là *object-class Definition* nhưng cũng có thể là *method-instance Definition* hoặc *field-instance Definition*. Để có thể kiểm tra được *Definition* chỉ định thuộc loại nào bạn có thể sử dụng các *method* sau của Definition:

- ***Kiểm tra thông tin defintition có phải được diễn dịch từ một Class hay không:***
  boolean *isInterpretativeClass*()
- ***Kiểm tra thông tin Definition có phải được diễn dịch từ một Method hay không:***
  public boolean *isInterpretativeMethod*()
- ***Kiểm tra thông tin Definition có phải được diễn dịch từ một Field hay không:***
  boolean *isInterpretativeField*()

Nếu là *object-class Definition* bạn có thể truy vấn được thông tin *metadata* của từng thành phần thành viên của nó. Nếu *Definition* chỉ định là *method-instance Definition*, bạn có thể truy vấn thông tin *metadata* (*annotation* tương ứng) của từng *parameter* nếu có. Cũng tương tự như *annotation*, *Definition* không cho phép truy vấn các thông tin *metadata* được chỉ định trong nội bộ *local* của một *method* trực tiếp.

4.3.3.3 - Truy vấn, thay đổi thông tin Definition

Do *Definition* chính là một tổng quát hóa của những *annotations*, nên về mặt bản chất *Definition* chỉ đơn giản là một *wrapping* của một tập hợp các *annotations*, và được thể hiện dưới dạng như là một thùng chứa (*container*) các *annotation proxy*, do đó cú pháp truy vấn thông tin *metadata* trong *Definition* được mô tả gần như tương tự *annotation* để đơn giản hóa và tạo cảm giác gần gũi như khi thao tác với *annotation* trong việc truy vấn dữ liệu *metadata*. Thêm vào đó, *Definition* cung cấp thêm một số thao tác đặc trưng để quản lý thông tin *Definition*, cũng như cho phép truy xuất vào trong thông tin *annotation* gốc, … cùng với một số thao tác đặc thù khác của từng loại *defintion*.

4.3.3.3.1 - Kiểm tra thông tin một annotation có được chỉ định trong Definition hiện hành hay không ?

Tương tự như khi thao tác với *annotation* trong việc kiểm tra một *annotation* chỉ định có tồn tại trong thực thể hiện hành, *Definition* cũng cung cấp một phương thức *(isAnnotationPresent method)* cho phép kiểm tra một *annotation* cụ thể có được chỉ định hay không với cú pháp sau:

```
boolean isAnnotationPresent(Class<? extends Annotation> annotation)
```

Không có gì khó khăn ngoại trừ một điều lưu ý ở đây rằng thông tin kiểm tra trên được thực thi trên danh sách *annotation* gốc của *Definition* chứ không phải được kiểm tra trên thông tin *Definition* sau khi được diễn dịch. Điều này có nghĩa rằng không đảm bảo thông tin kiểm tra trên là đồng bộ với dữ liệu thông tin đang hiện được cất trữ trên *Definition* tại thời điểm *run-time*.

Nếu trong trường hợp *Definition* hiện hành được diễn dịch từ một *object class*, bạn có thể kiểm tra thông tin *Definition* hiện hành có hay không cất trữ một *method* hoặc *field* có được định nghĩa *annotation*.

```
boolean isFieldAnnotationPresent()

... hoặc ...

boolean isMethodAnnotationPresent()
```

Cũng tương tự như *isAnnotationPresent()* method nhưng điểm khác biệt ở đây lại là 2 *method* trên kiểm tra thông tin hiện hữu của *annotation* không phải dựa trên thông tin *annotation* gốc mà từ dữ liệu sau khi được diễn dịch của *Definition*. Lý do của sự khác biệt với *isAnnotationPresent method* chính là nằm ở chỗ cấu trúc cất trữ thông tin của *Definition*. Để hiểu rõ hơn về cấu trúc *MetaData* trong *Definition*, vui lòng xem tại đây.

4.3.3.3.2 - Truy vấn thông tin Definition

Việc đọc thông tin nội dung *Definition* tương tự như sử dụng *reflection* trong java để truy vấn thông tin *annotation*. Đối tượng *Definition* cũng cung cấp 2 *method* là *getAnnotation method và getAnnotations method* cho phép lấy và truy vấn đến các *annotation* hiện đã được diễn dịch từ *annotation* gốc.

```
<T extends Annotation> T getAnnotation(Class<T> annotation)
Và …
Annotation[] getAnnotations()
```

Lưu ý rằng thông tin annotation được trả về này là thông tin đã được diễn dịch trong *Definition*, chứ không phải thông tin trên *annotation* gốc. Để có thể lấy được danh sách các thông tin trên *annotation* gốc bạn cần thực thi phương thức *getRawAnnotations method*.

```
<T extends Annotation> T getRawAnnotation(Class<T> annotation)
```

Thông tin được trả về từ *getRawAnnotation method* là thông tin của *annotation* gốc, điều này có nghĩa rằng khi dữ liệu *Definition* thay đổi thì thông tin truy vấn từ *getRawAnnotation method* cũng không hề thay đổi. Tuy rằng ít khi bạn cần sử dụng trực tiếp đến *annotation* gốc, do thông tin *metadata* được diễn dịch của *Definition* đã cung cấp đủ tất cả thông tin cần thiết, nhưng đôi lúc khi cần, … khả năng cho phép bạn có thể truy vấn vào thông tin *annotation* gốc có thể giải quyết được nhiều bài toán về *validate* dữ liệu, cũng như so khớp dữ liệu giữa *Definition* và *annotation* gốc … v..vv..

4.3.4 - Object-class Definition

Đối với *object-class Definition*, ngoài việc có thể truy vấn thông tin *annotation* từ chính thực thể chỉ định, bạn có thể truy vấn thông tin *annotation* trên các thực thể thành viên của chính nó (*methods, fields, constructor*, …).

Về mặt bản chất các thông tin *annotation* của từng thành phần con của 1 *object-class Definition* cũng chính là những *Definition*, do đó việc truy vấn các thông tin *metadata* được cất trữ trong các thành phần con của *object-class Definition* cũng sẽ trả về *Definition*. Trong trường hợp này, "*Definition* cha" sẽ chứa các mảng danh sách chứa các thông tin *Definition* của từng thành phần con ***"nếu có".*** Lưu ý rằng ***"nếu có"*** ở đây có nghĩa rằng không phải bất kì "*thành phần con*" nào của *object-class* cũng sẽ được chuyển đổi thành *Definition*, … mà chỉ những thành phần có chỉ định tường minh thông tin *annotation* lúc *compiled* hoặc lúc *configuration* (thông qua *JGentle container*) thì mới được chuyển đổi thành *Definition*.

Để có thể truy vấn các thông tin trên sử dụng cú pháp sau:

- ***Trả về danh sách chứa Definition của từng Field trong class:***
  HashMap<***Field***, Definition> *getFieldDefList()*

- ***Trả về danh sách chứa Definition của từng Method trong class:***

***HashMap*<*Method*, Definition> *getMethodDefList*()**

- ***Trả về danh sách chứa Definition của parameters của từng Method:***
  ***HashMap*<*Method*, Definition[]> *getParameterDefList*()**

Các *method* trên đều trả về một *hashmap* chứa đựng các thông tin, việc truy vấn vào các danh sách *hashmap* này đều cần phải được cung cấp các *key* tương ứng với các thực thể *method* hoặc *field* chỉ định (là thành phần con của *object-class*). Nếu trong trường hợp *Definition* chỉ định lại không phải là một *object-class Definition*, khi triệu gọi các *method* trên kết quả trả về sẽ là các danh sách *hashmap* rỗng.

Lưu ý rằng các từ khóa *key* của danh sách *hashmap* bao gồm *field* và *method* của *source class* chỉ định của *Definition* và đồng thời có cả những *field* hoặc *method* mà *source class* kế thừa từ các *super class* của nó nếu có. Do đó thông qua *reflection*, để có thể chỉ định đúng *field* hay *method* ứng với một *key* cụ thể bạn cần phải biết rõ *field* hay *method* đó thuộc cụ thể ở *derived class (source class)* hay là của *super class* nào của *source class* chỉ định.

Để đơn giản hóa việc truy vấn thông tin *Definition* từ các thành phần thành viên của một *object-class Definition*, đối tượng Definition cung cấp một số các phương thức truy vấn thông tin *Definition* trực tiếp:

Trường hợp definition là object-class definition và object class chỉ định có extends từ một hay nhiều base class khác:

- Trong trường hợp này, khi thông tin *definition* được khởi nạp, JGentle *container* sẽ ưu tiên các thông tin *annotation* được chỉ định trên *derived class* trước kế đến mới là thực thi việc *convert* thông tin trên *base class*. Hay nói cách khác, nếu như *derived class* có *override* lại thông tin nào đó từ *base class* nếu có *(methods, field)* thì *container* sẽ ưu tiên chuyển đổi thông tin từ *derived class* thành *definition*, kế đến **"nếu như"** *derived class* không chỉ định bất kì thông tin *annotation* nào thì dữ liệu *annotation* trên *base class* mới được thực thi diễn dịch.
- Ngoài ra, đối với các thành phần static *(method hoặc field), JGentle container* chỉ diễn dịch trên các thành phần của *class* chỉ định chứ không diễn dịch trên các *super class* của nó.

- **Trả về danh sách Definition tương ứng với các parameters của một method chỉ định nếu có:**

Definition[] *getArgsDefinitionChild*(***Method*** method)

- **Trả về Definition tương ứng với một field chỉ định (dựa trên tên field) nếu có:**
  Definition *getDefinitionChild*(***String*** fieldName)

- **Trả về Definition tương ứng với một field chỉ định nếu có:**
  Definition *getDefinitionChild*(***Field*** field)

- **Trả về Definition tương ứng với một method chỉ định (dựa trên tên name của method cùng với danh sách kiểu của tham số truyền) nếu có:**
  Definition *getDefinitionChild*(***String*** methodName, ***Class***<?>[] args)

- **Trả về Definition tương ứng với một method chỉ định nếu có:**
  Definition *getDefinitionChild*(***Method*** method)

Sau khi nắm được *Definition* của các thành phần thành viên, bạn có thể hoàn toàn truy xuất tương tự như một *Definition* thông thường để có thể lấy ra thông tin *annotation* tương ứng đã được diễn dịch.

4.3.5 - Thay đổi thông tin dữ liệu Definition

Mặt hạn chế của *annotation* (tính tại thời điểm hiện tại JDK 6.0) đó chính là thông tin trong *annotation* không thể thay đổi tại thời điểm *run-time*. Dữ liệu được cất trữ trong *annotation* mặc dù chỉ đơn thuần là những *java objects* nhưng lại chỉ có thể được chỉ định tại thời điểm *compiled*. Điều này có nghĩa rằng nếu bạn muốn chỉ định thông tin *annotation* có thể tùy biến tại thời điểm *run-time*, bạn cần phải tự *build* lấy một hệ thống quản lý *metadata* riêng hoặc chuyển đổi thông tin *annotation* thành một thành phần dữ liệu khác có thể thay đổi được, và sử dụng nó thay cho thông tin lấy từ *annotation*.

Như mô tả ở trên, việc chuyển đổi này đã được JGentle giải quyết bằng cách sử dụng thông tin *Definition* (được *convert* từ *annotation*) thay vì sử dụng *annotation* trực tiếp. Và dĩ nhiên thông tin trong *Definition* là hoàn toàn có thể tùy biến tại thời điểm *run-time*. Hay nói cách khác JGentle đã xử lý *problem* này cho các nhà phát triển, giờ đây nếu ứng dụng có xử

lý đến các dữ liệu *metadata*, nhà phát triển chỉ cần quan tâm đến vấn đề *logic* của hệ thống, việc quản lý các thông tin *metadata* sẽ do JGentle *container* đảm nhiệm. Thông tin *metadata* giờ đây với sự trợ giúp của *Definition* và cơ chế quản lý *services* với *AOH (Annotation Object Handling)* có thể dễ dàng tuỳ biến lúc cấu hình, *compiled-time*, hay tại thời điểm *run-time*.

```
Object setValueOfAnnotation(Class<? extends Annotation> annotation,
                            String valueName, Object value)
```

Để có thể thay đổi thông tin dữ liệu *metadata* của một *annotation* chỉ định ứng với một *Definition*, bạn chỉ cần đơn giản triệu gọi hàm *setValueOfAnnotation method* và cung cấp các tham số cần thiết. Thông tin trả về của *method* này chính là giá trị trước khi thay đổi của thuộc tính của *annotation* chỉ định. Và dĩ nhiên rằng thông tin của *annotation* gốc vẫn giữ nguyên, điều này có nghĩa là thông tin sẽ được thay đổi trong *metadata* của *annotation proxy* (*AnnoMeta* của *annotation* gốc) sau khi đã được diễn dịch.

Việc thay đổi thông tin dữ liệu *metadata* như cách trên sẽ thay đổi ứng với một *annotation proxy* chỉ định của một *annotation gốc*. Để có thể thay đổi toàn bộ thông tin *metadata* của toàn bộ *Definition*, JGentle cung cấp phương thức *setRawAnnotationList method* cho phép thay đổi toàn bộ thông tin *annotation gốc* bằng một danh sách list các *annotation* mới, và đồng thời thực thi diễn dịch lại toàn bộ thông tin *Definition.*

```
void setRawAnnotationList(Annotation[] rawAnnotationList)
```

Thông tin của *Definition* sau khi triệu gọi sẽ được diễn dịch lại toàn bộ, nhưng lưu ý rằng với một *object-class Definition* thì thông tin *Definition* sẽ chỉ diễn dịch lại ở mức *object-class*. Hay nói cách khác, các thông tin *Definition* tương ứng của các thành phần *methods, fields* là con của *object class* chỉ định sẽ không được thực thi diễn dịch. Để có thể diễn dịch lại thông tin *Definition* của các thành phần này, hay toàn bộ tận gốc dữ liệu của một *object-class Definition*, bạn cần truy vấn đến từng *Definition* cụ thể và thực thi *setRawAnnotationList method* của *Definition* đó để có thể có được thông tin dữ liệu mới.

Như vậy, việc thay đổi thông tin danh sách *annotation* gốc sẽ khiến cho toàn bộ

thông tin *Definition* sẽ được diễn dịch lại. Điều này được thực hiện do phương thức *setRawAnnotationList()* sau khi thực hiện việc thay đổi dữ liệu *annotation* xong sẽ tự động triệu gọi phương thức *buildAnnoMeta()* để tái cấu trúc và diễn dịch lại dữ liệu *Definition*.

```
void buildAnnoMeta()
```

Với *buildAnnoMeta* method, bạn có thể tự triệu gọi phương thức *buildAnnoMeta()* bằng tay tại thời đểm *run-time* thay vì thông qua phương thức *setRawAnnotationList()* để có thể diễn dịch lại thông tin *Definition* khi cần thiết.

4.3.6 - Cấu trúc MetaData trong Definition

Tuân theo cấu trúc thông tin dữ liệu trong *annotation*, *Definition* chuyển đổi thông tin trong *annotation* thành một dạng dữ liệu gián tiếp, và sử dụng để ánh xạ đến thông tin *annotation* gốc, trong JGentle, thông tin này được gọi là *AnnoMeta (Annotation Metadata).* Bản chất thông tin *AnnoMeta* đơn giản chỉ là một *object* ánh xạ đến *"một hoặc nhiều"* đối tượng *annotations*, diễn dịch lại thông tin nội bộ của *annotations* dưới dạng *MetaData (một loại dữ liệu được chuyển đổi từ từng giá trị thuộc tính cụ thể của annotation chỉ định) .* Ngoài ra, *AnnoMeta* cũng có thể chứa cả các *AnnoMeta* khác là con của nó và mỗi một *Definition* đều có chứa đựng bên trong một *AnnoMeta*. *AnnoMeta* của *Definition* chính là đối tượng chứa đựng thông tin của *annotations* của *source object* sau khi đã được diễn dịch.

Thông tin của *Definition* nói riêng, hay toàn bộ hệ thống *JGentle container* nói chung đều được quy đổi thành *AnnoMeta* và *MetaData* và được *container* sử dụng như là những *"siêu dữ liệu"* thông tin cấu hình vận hành *container* hoạt động.

Definition, AnnoMeta, hay *MetaData* tất cả đều là những đối tượng mang khoác lấy những nội dung thông tin *metadata* hoàn toàn khác nhau, tuy nhiên đều có một cặp giá trị *Key, Value* chứa đựng thông tin nội dung mà chúng hiện đang quản lý. Cặp giá trị *Key-Value* mô tả các thành phần thông tin dữ liệu chủ yếu của một thực thể *metadata*. Để có thể lấy ra được *Key, Value* trên từng thực thể chỉ định bạn chỉ cần đơn giản triệu gọi phương thức *getKey() và getValue()*, giá trị trả về của 2 *method* trên thuộc kiểu *Object* mô tả dữ liệu nội dung của thực thể đó. Và điều dĩ nhiên rằng đối với từng loại thực thể chứa *metadata* khác

nhau *(Definition, AnnoMeta, MetaData)*, cặp *Key* và *Value* đều biểu thị cho những loại dữ liệu khác nhau.

- ***Lấy ra Key:***

Object getKey();

- ***Lấy ra Value:***

Object getValue();

4.3.6.1 - Key và Value của Definition

Đối với *Definition, key* của *Definition* chính là đối tượng chứa thông tin được cấu hình (*Annotation*), còn *value* chính là đối tượng *AnnoMeta* của nó. Hay nói cách khác, việc triệu gọi phương thức getKey() trên *Definition* sẽ trả về đối tượng *object* là *"đối tượng chủ thể"* chứa đựng thông tin *annotation* được diễn dịch thành *Definition*, còn việc triệu gọi phương thức getValue() sẽ trả về chính đối tượng *AnnoMeta* của *Definition* hiện hành chứa đựng thông tin *Definition* sau khi đã được diễn dịch.

Điều này có nghĩa rằng mệnh đề sau:
Definition.*getAnnoMeta*() == Definition.*getValue*() sẽ luôn trả về kết quả là ***true***.

"Đối tượng chủ thể" ở đây có thể là bất kì đối tượng nào có thể có đính kèm thông tin *annotation (Class, Method, Field, …)* lúc được *compiled* mà JGentle *container* cho phép diễn dịch thành *Definition*, nhưng cũng có thể là chính các đối tượng này nhưng không đính kèm thông tin *annotation* lúc *compiled* (mà được chỉ định thực thi *@inject annotation* tại thời điểm *configuration* thông qua **ADI** hoặc lúc *run-time*).

Việc lấy ra được đối tượng chủ thể được chỉ định *annotation* (trực tiếp hoặc gián tiếp) cho phép nhà phát triển biết được *Definition* hiện hành được diễn dịch từ đối tượng nào, thông tin *metadata* thực chất của chủ thể lúc *compiled* hoặc có thể so khớp thông tin diễn dịch từ *Definition* với thông tin *metadata* trên chủ thể, … v..vv..

Như đã nói ở trên, đối tượng trả về khi triệu gọi phương thức *getValue*() của

Definition sẽ chính là *AnnoMeta* của *Definition* đó. Tuy rằng chỉ đơn thuần là một *AnnoMeta*, nhưng *AnnoMeta* đại diện của một *Definition* có những đặc tính về *key* và *value* hơi khác so với các *AnnoMeta* thông thường khác. Thông tin chi tiết mô tả về *AnnoMeta* vui lòng xem phần mô tả *"Key và Value của AnnoMeta".*

4.3.6.2 - Key và Value của AnnoMeta

AnnoMeta bản chất chính là đối tượng cất trữ thông tin của một *annotation* sau khi được diễn dịch và chuyển đổi thông tin. Do đó đối với *AnnoMeta*, *key* của *AnnoMeta* chính là đối tượng *object class* của *annotation* gốc của *AnnoMeta*, còn *value* của *AnnoMeta* chính là đối tượng *annotation* gốc mà *AnnoMeta* hiện hành được diễn dịch từ đó. Hay nói cách khác nếu như bạn triệu gọi phương thức getKey() trên AnnoMeta, đối tượng trả về sẽ là *object-class* (hay *annotation type*) của *annotation* được sử dụng để diễn dịch lên thông tin trong *AnnoMeta*. Còn khi triệu gọi phương thức *getValue*() trên *AnnoMeta* kết quả trả về sẽ là chính *annotation* gốc đó.

Điều này có nghĩa rằng mệnh đề sau:

annoMeta.*getKey*()== ((Annotation)annoMeta.*getValue*()).*annotationType*() sẽ trả về kết quả là ***true***.

Tuy nhiên đó chỉ là trong trường hợp *AnnoMeta* thể hiện như là một đối tượng diễn dịch thông tin từ *annotation*. Còn trong trường hợp *AnnoMeta* là thể hiện đối tượng chứa đựng thông tin của 1 *Definition*, thì *key* và *value* của *AnnoMeta* lại không mang ý nghĩa như trên. Do đó, nếu như trong trường hợp bạn lấy ra *AnnoMeta* của *Definition* bằng cách triệu gọi phương thức Definition.*getAnnoMeta*() hoặc Definition.*getValue*() thì *key* của *AnnoMeta* được lấy ra sẽ chính là "*key* của *Definition*" chứa đựng *AnnoMeta* hiện hành còn *value* của *AnnoMeta* sẽ là chính *Definition* đó. Hay nói cách khác, trong trường hợp này key của *AnnoMeta* sẽ chính là đối tượng chủ thể chứa đựng thông tin *annotation* được diễn dịch thành *Definition* và 2 mệnh đề sau sẽ luôn luôn trả về ***true:***

Definition.*getAnnoMeta*().*getKey*() == Definition.*getKey*()

...và

Definition.*getValue*() == Definition

Lưu ý rằng *AnnoMeta* khi lấy ra bởi việc triệu gọi phương thức Definition.*getAnnoMeta*() có thể sẽ là ***null*** hoặc là một "*AnnoMeta* rỗng" (*AnnoMeta* không hề chứa đựng bất kì thông tin gì bên trong) nếu như *Definition* hiện hành được diễn dịch từ 1 *Class* nhưng *Class* đó không được chỉ định thông tin *annotation* (trực tiếp hoặc gián tiếp thông qua *inject*) mà chỉ có *Methods* hoặc *Fields* của nó mới có chỉ định thông tin *annotation*. Trong trường hợp *AnnoMeta* của *Definition* là một *AnnoMeta* rỗng, có nghĩa rằng *AnnoMeta* trên không chứa đựng bất kì thông tin gì bên trong có thể truy vấn được "ngoại trừ" *key* và *value* của *AnnoMeta*.

4.3.6.3 - Key và Value của MetaData

Nếu như dữ liệu bên trong *Definition* là một *AnnoMeta* đại diện cho nội dung cần chứa đựng thì bên trong *AnnoMeta* là một danh sách gồm một hoặc nhiều *MetaData* biểu thị cho nội dung bên trong của *AnnoMeta*. Giao diện *interface AnnoMeta* chính là một *interface* dẫn xuất, kế thừa từ *interface Metadata* (một *interface class* của *AOPAlliance*), do đó đôi lúc có thể xem một *AnnoMeta* như là một *MetaData*.

Thông tin *MetaData* chứa đựng có thể là những dữ liệu cơ bản đã được diễn dịch từ các thuộc tính của *annotation* hoặc cũng có thể là chính *AnnoMeta*. Nhưng thông thường theo cách hiểu về mặt ngữ nghĩa, một *MetaData* chính là một thông tin thuộc tính của một *annotation* chỉ định đã được diễn dịch. Và trong trường hợp này, *key* của *MetaData* chính là tên của thuộc tính còn *value* của *MetaData* chính là dữ liệu trả về của nó trong *annotation* chỉ định hiện hành.

MetaData là thành phần dữ liệu cơ bản nhất tạo thành được sử dụng trong hệ thống JGentle container, về mặt bản chất cả *Definition* hay *AnnoMeta* cũng có thể xem như là những *MetaData* vì cả hai đều được *implements* từ *interface Metadata* (nằm trong gói *package* org.aopalliance.reflect.Metadata của *AOPAlliance*). Việc thay đổi nội dung dữ liệu trong *Definition* nói chung hay *AnnoMeta* nói riêng đều chính là thay đổi thông tin từng *MetaData* cụ thể chỉ định. Do đó để có thể thay đổi thông tin của một *Definition*, ngoài việc tương tác với các *proxy object* bên ngoài *Definition*, sử dụng các API thích hợp mà JGentle cung cấp bạn cũng có thể thay đổi thông tin của một *Definition* bằng cách truy xuất trực tiếp vào thông tin dữ liệu *MetaData* của nó.

4.3.6.4 - Mối quan hệ giữa Definition, AnnoMeta và MetaData

Trong JGentle, *Definition*, *AnnoMeta* và *MetaData* có mối quan hệ rất mật thiết. Nội dung bên trong *Definition* bản chất chính là một thể hiện của *AnnoMeta* đã được diễn dịch. Còn *AnnoMeta*, một mặt có thể hiện như là một *metadata* của *Definition*, mặt khác thể hiện như là một dữ liệu diễn dịch của một *annotation* chỉ định. Và cuối cùng là *MetaData*, được xem như là thành phần cơ bản nhất tạo nên thông tin *metadata* trong JGentle, thể hiện như là một loại dữ liệu diễn dịch từ một thuộc tính của một *annotation* chỉ định nào đó. *MetaData* chính là thành phần tạo nên thông tin dữ liệu trong một *AnnoMeta*.

Xét về mặt tổ chức logic, có vẻ như *Definition*, *AnnoMeta* và *MetaData* là những thành phần dữ liệu tách rời, nhưng thực chất tất cả đều có thể ép kiểu về 1 kiểu chung duy nhất đó là *Metadata interface* do tất cả các đối tượng trên đều *implements* cùng 1 *interface Metadata* và đều cùng kế thừa từ *MetadataImpl class*. Hay nói cách khác, *Definition, AnnoMeta, và MetaData* đều là những *MetaData* với những đặc tính chuyên biệt, chịu trách nhiệm quản lý và chứa đựng các nội dung *dữ liệu cấu hình* đi trong *container*.

Như đã nói ở trên, một *Definition* có thể có hoặc không các *Definition child*, là các *Definition* con được chỉ định cho các thành phần con *(Methods, Fields)* nếu *Definition* hiện hành diễn dịch từ 1 *object-class*. Tương tự như thế, một *AnnoMeta* cũng có thể có hoặc không *AnnoMeta parents*, là *AnnoMeta* cha của nó trong trường hợp *AnnoMeta* hiện hành được chứa đựng bên trong 1 *AnnoMeta* khác (vd như *definition_AnnoMeta* chứa bên trong các *AnnoMeta* của các *annotation*). Để có thể lấy ra được *AnnoMeta parent* của *AnnoMeta* hiện hành bạn chỉ cần đơn giản triệu gọi phương thức *getParents*(), kết quả trả về sẽ là một *AnnoMeta*. Nếu trong trường hợp *AnnoMeta* hiện hành là *definition_AnnoMeta*, kết quả trả về sẽ là ***null***.

Ngoài ra, *AnnoMeta* còn chứa đựng các thông tin thuộc tính phụ khác như tên định danh *name* và *class-type* của *AnnoMeta*. Để có thể truy xuất tên định danh name và *class-type* của *AnnoMeta* bạn chỉ cần đơn giản triệu gọi 2 phương thức sau của *AnnoMeta*:

```
String getName()

Và ...
```

```
Class<?> getType()
```

Mặc định, tên định danh *name* của *AnnoMeta* sẽ là tên *Class* của *object-class* của *annotation* gốc mà *AnnoMeta* diễn dịch từ đó. Trong trường hợp *AnnoMeta* là một *definition_AnnoMeta*, sẽ có 2 trường hợp :

- **Definition diễn dịch từ Class:** lúc này tên định danh *name* của *AnnoMeta* sẽ là tên *Class* chỉ định.
- **Definition diễn dịch từ Method, Field:** lúc này tên định danh *name* của *AnnoMeta* sẽ là tên *Class* mà *Method* hoặc *Field* chỉ định được khai báo bên trong.

Tương tự như tên định danh *name*, *class-type* của *AnnoMeta* mặc định sẽ là *object class* của *annotation* gốc mà *AnnoMeta* diễn dịch từ đó. Còn trong trường hợp *AnnoMeta* là một *definition_AnnoMeta* thì nếu *Definition* diễn dịch từ *Class* thì *class-type* sẽ là *object class* của *class* chỉ định, còn nếu *Definition* diễn dịch từ *Method hoặc Field* thì *class-type* sẽ là *object-class* của *class* mà mà *Method* hoặc *Field* chỉ định được khai báo bên trong.

Class-type của *AnnoMeta* **không thể** thay đổi tại thời điểm *run-time* nhưng tên định danh *name* thì có thể. Bạn có thể sử dụng phương thức *setName*(***String*** name) để thay đổi tên *name* tương ứng làm định danh cho một *AnnoMeta* cụ thể tại *lúc run-time*.

Cuối cùng là *MetaData*, đơn vị cất trữ dữ liệu nội dung cơ bản nhất, không chứa đựng bất kì thông tin gì đặc biệt ngoại trừ cặp giá trị *Key* và *Value*. Nếu như bên trên *Definition* và *AnnoMeta* sử dụng *Key và Value* để mô tả các thông tin khái quát của chính mình thì đối với *MetaData*, *Key và Value* lại chính là dữ liệu thật sự mà nó hiện đang cất giữ. Dữ liệu trong *MetaData* được mô tả bởi cặp giá trị *Key và Value* với kiểu là *Object* (đối tượng tổng quát nhất có thể). Và trong *AnnoMeta* có cất trữ một *list* các *MetaData* của nó trong một danh sách *Hashmap* với *key* chính là *Key* của *MetaData* tương ứng. Hay nói cách khác *AnnoMeta* quản lý các *MetaData* của nó dựa trên chính *Key* của *MetaData*.

Do đó nếu biết được chính xác *Key* của *MetaData,* bạn có thể dễ dàng truy vấn ra *MetaData* đó trong một *AnnoMeta* cụ thể. Và tại phiên bản hiện tại của JGentle, *MetaData* được sử dụng như là một đơn vị dữ liệu đã được diễn dịch của một thuộc tính trong một

annotation, với *Key* của *MetaData* trong trường hợp này chính là tên của thuộc tính (kiểu *String*), còn *Value* chính là dữ liệu trả về của thuộc tính đó. Dĩ nhiên trong khi vận dụng thực tế bạn hoàn toàn có thể sử dụng *MetaData* theo một cách logic riêng để quản lý các thông tin *metadata* phù hợp với xử lý của từng trường hợp đặc thù.

Với cấu trúc quản lý dữ liệu như trên bạn hoàn toàn có thể truy vấn trực tiếp vào nội dung diễn dịch của một *Definition*, hay thậm chí là *AnnoMeta* hay *MetaData* để chỉnh sửa , thay đổi, truy vấn một cách linh hoạt mà không cần phải thông qua *Annotation Proxy* của *Definition*. Chi tiết thế nào xin xem tiếp phần dưới.

4.3.6.5 - Annotation Proxy

Quay lại với việc thao tác truy vấn và chỉnh sửa thông tin Definition. Khi chúng ta triệu gọi phương thức *getAnnotation() hoặc getAnnotations()* trên *Definition*, *Definition* sẽ trả về một *annotation* hoặc một danh sách các *annotations* tương ứng, cho phép chúng ta thao tác vào bên trong nội dung *metadata* đã được diễn dịch của *Definition* tương tự như cách chúng ta thao tác vào một *annotation* thông thường thông qua *Reflection*. Tuy nhiên đó không phải thật sự là một *annotation*, đó chỉ là một *annotation proxy instance*, mang *"hình hài"* của *annotation* gốc. Và khi bạn truy vấn vào dữ liệu bên trong của các *annotation* này, thông tin truy vấn sẽ được *container* chuyển hướng đến các dữ liệu *Metadata* hiện đang được cất trữ trong *Definition* hiện hành, chuyển đổi kiểu, rồi trả về cho *client* theo một hình thái y hệt như một *annotation* thật sự. Với giải pháp này, JGentle cho phép bạn thao tác vào bên trong thông tin của *annotation*, truy vấn thậm chí chỉnh sửa dữ liệu nội dung tại thời điểm *run-time* mà không hề ảnh hưởng gì đến dữ liệu *annotation* gốc. Ngoài ra, cùng với việc tách biệt dữ liệu thông tin cấu hình, chứa đựng tại một nơi khác, tách rời thông tin *annotation* với *logic code* cho phép nhà phát triển dễ dàng tái sử dụng thông tin cấu hình (trong *annotation*), mở rộng cũng như bảo trì *logic code và configuration code*.

Tất cả mọi phương thức có trên *annotation proxy* đều tương tự hoặc gần như tương tự so với *annotation* gốc, ngoại trừ 1 điểm lưu ý rằng: nếu như bạn triệu gọi các phương thức sau trên *annnotation proxy* ví dụ như *equals(), toString(), hoặc hashCode(), annotation proxy* sẽ tự động chuyển hướng *redirect* sang cho đối tượng *annotation* gốc. Hay nói cách khác việc triệu gọi các phương thức trên sẽ cho ra kết quả y hệt như khi bạn triệu gọi chúng trên *annotation* gốc.

4.3.7 - Truy vấn thông tin Definition trực tiếp thông qua AnnoMeta

Ngoài việc có thể truy vấn thông tin *Definition* thông qua *annotation proxy*, bạn còn có thể truy vấn được thông tin của một *Definition* bằng cách truy xuất trực tiếp vào *AnnoMeta* của *Definition*. Cách này có vẻ rườm rà và không thuận tiện như khi bạn làm việc với *annotation proxy* nhưng đôi khi sẽ rất hữu ích nếu như bạn muốn có các xử lý linh hoạt khi muốn can thiệp sâu vào dữ liệu *metadata*.

Như trên đã có trình bày để có thể truy vấn vào *AnnoMeta* của một *Definition* bạn triệu gọi phương thức sau:

```
AnnoMeta getAnnoMeta()
```

Sau khi đã có được nội dung *AnnoMeta* của *Definition* bạn có thể truy vấn trực tiếp vào các dữ liệu con bên trong *AnnoMeta*, ví dụ bạn có 1 *annotation* tên là *MyAnnotation* và được chỉ định lên *Client class* như sau :

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
	String name() default "JGentle";
}

@MyAnnotation
class Client {
}
```

Để có thể truy cập vào nội dung *metadata* của *AnnoMeta*, bạn làm như sau:

```
public class AnnoMetaTest {
	public static void main(String[] args) {
```

```
            // Khởi tạo InjectCreator container
             InjectCreator injector = JGentle.buildInjectCreator();

            // Lấy ra đối tượng Definition Manager
             DefinitionManager defManager = injector.getDefManager();
             defManager.loadDefinition(Client.class); // Nạp thông tin Definition

             Definition def = defManager.getDefinition(Client.class);

            // Lấy ra đối tượng AnnoMeta của Definition
             AnnoMeta root = def.getAnnoMeta();

            // Lấy ra đối tượng AnnoMeta tương ứng với @MyAnnotation
             AnnoMeta annoMeta = (AnnoMeta) root.getMetadata(MyAnnotation.class);
             Metadata metadata = annoMeta.getMetadata("name");

            // Hiển thị thông tin metadata
             System.out.println(metadata.getValue());
        }
}
```

Như đã nói ở trên, một *AnnoMeta* sẽ quản lý các dữ liệu *metadata* (*MetaData* hoặc *AnnoMeta* khác) bên trong nó dựa trên key của chính thực thể *metadata* đó. Điều này cũng không ngoại lệ đối với *definition_AnnoMeta*. Và cũng như đã nói ở trên, *key* của một *AnnoMeta* khi được diễn dịch từ một *annotation* sẽ là *object-class* của *annotation* gốc của nó. Do đó, sau khi đã lấy ra được *root AnnoMeta (definition_AnnoMeta)* của *definition* hiện hành, bạn có thể truy vấn lấy ra các *AnnoMeta* con của nó bằng cách cung cấp dữ liệu *object-class* của *annotation* cần truy vấn như là tham số cho phương thức *getMetadata*() của *root AnnoMeta*.

Sau khi đã lấy ra được *AnnoMeta* tương ứng với *annotation* chỉ định, để có thể truy vấn ra các thuộc tính, bạn chỉ cần đơn giản cung cấp tên của thuộc tính cho phương thức *getMetadata*() của *AnnoMeta* hiện hành (ở đây là chuỗi *"name"*). Kết quả trả về sẽ là một đối tượng *Metadata* ứng với thuộc tính chỉ định. Và cuối cùng là chỉ cần triệu gọi phương

thức *getValue*() trên *MetaData* để lấy ra được giá trị cuối cùng cần truy vấn.

Việc truy vấn ra thông tin của *Definition* thông qua *annotation proxy* thực chất cũng chỉ đơn giản là tự động hóa lại quy trình tương tự như trên dưới hình thức là một *proxy instance* của một *annotation type*. Do đó, nếu với *annotation proxy* bạn có thể làm được những gì thì với cách thức truy cập bằng *AnnoMeta,* bạn cũng có thể làm được y như thế. Điểm khác biệt duy nhất giữa 2 phương pháp trên là đối với *annotation proxy*, bạn sẽ không được phép can thiệp thô bạo vào trong cấu trúc của *annotation* sau khi được diễn dịch (ví dụ: *annotation* gốc có 5 thuộc tính thì *annotation proxy* cũng sẽ có 5 thuộc tính). Việc thêm vào 1 thuộc tính (một *MetaData*) tại thời điểm *run-time* sẽ không được *annotation proxy* cho phép. Nhưng với cách truy cập thông tin *metadata* trực tiếp, bạn có thể hoàn toàn làm chủ được nội dung của *AnnoMeta hay Definition*, việc thêm bớt, xóa, sửa dữ liệu *metadata* hoàn toàn không có bất kì ràng buộc gì, do đó bạn có thể tự do tùy biến nội dung *AnnoMeta* tùy theo mục đích *logic* của ứng dụng.

> *Lưu ý rằng, trong hầu hết các trường hợp thông thường, khi sử dụng Definition để cất trữ thông tin của dữ liệu annotation, JGentle khuyến cáo bạn nên tuân theo những quy tắc chung do Definition, AnnoMeta hay annotation proxy đặt ra, để tránh các trường hợp sai lệch thông tin dữ liệu metadata khi sử dụng trong các tầng services khác nhau của JGentle (như DI, ADI, AOH, …). Hay nói cách khác, nếu không có trường hợp nào đặc biệt, bạn nên sử dụng cách truy vấn thông tin Definition bằng cách thông thường thông qua annotation proxy.*

4.3.8 - Khởi tạo bằng tay các dữ liệu metadata

Đôi lúc khi cần, việc khởi tạo các đối tượng *metadata* không cần thiết phải ủy nhiệm cho JGentle *container* mà bạn hoàn toàn có thể tự khởi tạo các đối tượng này (*Definition, AnnoMeta, MetaData*, …) thông qua hệ thống API được cung cấp sẵn trong JGentle. JGentle có cung cấp một *class* đặc biệt tên là ***MetaDataFactory*** *(chỉ định trong gói package org.exxlabs.jgentle.core.reflection.aohreflect.metadata*) là một *abstract* class cung cấp một tập hợp các *static methods* khác nhau trợ giúp cho việc khởi tạo các đối tượng trên.

Để khởi tạo *Definition*:

```
static Definition createDefinition (Object key, IAnnotationVisitor visitor)

static Definition createDefinition (Object key,
                    IAnnotationVisitor visitor, Annotation[] annoList)

static Definition createDefinition (Object key, Annotation[] annoList)
```

Khởi tạo *AnnoMeta*:

```
static AnnoMeta createAnnoMeta (Object key, Object value)

static AnnoMeta createAnnoMeta (Object key, Object value,
                    AnnoMeta annoParents)

static AnnoMeta createAnnoMeta (Object key, Object value,
                    AnnoMeta annoParents, String name)
```

Khởi tạo *MetaData:*

```
static Metadata createMetaData(Object key)

static Metadata createMetaData(Object key, Object value)
```

Việc khởi tạo các đối tượng cất trữ thông tin *metadata* giờ đây rất đơn giản, bạn có thể hoàn toàn chủ động trong việc chỉ định nội dung cho một *metadata* cũng như tạo mới chúng thông qua các *static methods* trên. Nhờ vậy, sử dụng kết hợp giữa việc tùy biến thông tin dữ liệu *AnnoMeta*, *MetaData* cùng với ***MetaDataFactory*** cho phép bạn có thể tự tạo các cấu trúc *metadata* riêng, ứng với các *logic* xử lý riêng một cách dễ dàng.

4.3.9 - Quản lý các điểm Extension-Points trong khi diễn dịch Definition

Hệ thống JGentle được thiết kế để có thể chủ động mở rộng cũng như tùy biến, và *extension-points (EP)* chính là một trong những hình thức cho phép mở rộng chức năng của JGentle. Trong khi diễn dịch các thông tin *Definition*, hệ thống JGentle có cung cấp một số các *extension-points* chỉ định, cho phép ứng dụng nắm lấy, cấu hình và vận hành theo một *logic* nào đó, tùy biến hệ thống tại thời điểm *run-time*. Các *EP* trong JGentle cho phép bạn can thiệp vào các tiến trình xử lý, chỉ định cách thức xử lý cũng như điều khiển hệ thống hoạt động theo một chu trình tự tạo. Nhờ đó mà nhà phát triển có thể tự điều khiển, dẫn dắt hệ thống hoạt động một cách linh hoạt, không phụ thuộc vào hình thức mặc định trong JGentle, và không bị bó buộc vào một tiến trình cứng nhắc do hệ thống đặt ra.

Để có thể mở rộng và xử lý các *EP* trong khi *diễn dịch Definition*, JGentle IOC *container* có đề ra một loạt các *interface* đặc biệt ứng với các chức năng cụ thể. Các *interface* này đóng vai trò như là một hệ thống khai báo các *callback methods*, tương ứng với từng *method* chỉ định bạn sẽ có được những hình thái chức năng riêng biệt. Và công việc còn lại của nhà phát triển cần làm chỉ là *implements* các *interface* này và nhúng chúng vào JGentle *container* như là các *bean* chỉ định cần hoạt động.

4.3.9.1 - DefinitionPostProcessor - Tùy biến tiến trình diễn dịch Definition

Trong quá trình diễn dịch thông tin từ một thực thể chỉ định thành *Definition*, JGentle có đặt ra một vài EP cho phép bạn can thiệp vào tiến trình xử lý, và một trong số đó được gọi là DefinitionPostProcessor tương ứng với một *interface* cùng tên DefinitionPostProcessor (được chỉ định trong gói *package org.exxlabs.jgentle.core.reflection.aohreflect*). Thông qua ***EP*** này bạn có thể can thiệp vào tiến trình diễn dịch *Definition trước* hoặc *sau* khi việc diễn dịch *Definition* được tiến hành. *Interface* DefinitionPostProcessor bao gồm 3 *callback methods* sau:

```
void beforePost (Annotation[] annoArray, AnnoMeta annoMeta) throws
                                                    DefinitionPostException

void afterPost (Annotation[] annoArray, AnnoMeta annoMeta) throws
                                                    DefinitionPostException
```

```
boolean catchException (Exception ex, Annotation[] annoArray,
                        AnnoMeta annoMeta, boolean bool)
```

Phương thức *beforePost* tương ứng với các thực thi sẽ được chỉ định trước khi một *Definition* được xử lý diễn dịch, còn phương thức *afterPost* tương ứng với các thực thi sẽ được chỉ định sau khi quá trình diễn dịch *Definition* đã hoàn tất. Cả 2 *methods* trên đều có cùng kiểu đối số truyền: annoArray chính là danh sách các *annotation* được chỉ định sẽ diễn dịch thành *Definition*, còn annoMeta chính là đối tượng *root AnnoMeta* của *Definition* trước và sau khi diễn dịch.

Trong quá trình thực thi, nếu như có bất kì ngoại lệ nào được ném ra (ngoại lệ mặc định ở đây là *DefinitionPostException*), thì quá trình thực thi sẽ được chuyển hướng xuống cho *catchException method* xử lý. Phương thức *catchException* hoạt động như là một phương thức chận hứng tất cả các ngoại lệ phát sinh từ 2 *method beforePost và afterPost* nếu có, nhận và xử lý các ngoại lệ khác nhau, và cho phép hay không tiến trình diễn dịch thực thi sau khi nhận bắt được *exception*. Phương thức *catchException* có kiểu trả về là *boolean*, nếu cài đặt của *method* này trả về true, quá trình diễn dịch sẽ tiếp tục (*tiếp tục ở đây có nghĩa rằng quá trình sẽ tiếp tục được thực thi, có thể là bắt đầu diễn dịch Definition (thời điểm before) hoặc kết thúc quá trình diễn dịch (thời điểm after) hoặc chuyển hướng sang cho DefinitionPostProcessor*-DPP *kế tiếp nếu container có cài đặt nhiều* DPP *khác nhau*). Còn nếu cài đặt của *catchException method* trả về *false*, quá trình diễn dịch sẽ lập tức được dừng lại. Trường hợp ngoại lệ ném ra tại thời điểm *beforeMethod* thực thi và *catchException* lại trả về *false* thì đồng nghĩa sẽ không có bất kì *Definition* nào được diễn dịch.

> *Lưu ý rằng implementation của* ***EP*** *DefinitionPostProcessor sẽ được tự động thực thi* ***khi*** *diễn dịch một Definition, kể cả các Definition child của Definition hiện hành cũng sẽ có tác động hiệu quả tương tự. Hay nói cách khác nếu như Definition hiện hành là một Object-Class Definition, và có các thành phần con (Method hoặc Field) có chỉ định annotation được diễn dịch thành Definition thì DefinitionPostProcessor sẽ có thể thực thi nhiều lần dù cho phương thức loadDefinition chỉ được triệu gọi 1 lần duy nhất.*

Một *container* sẽ có thể có nhiều *DefinitionPostProcessor* khác nhau cùng lúc, thứ tự

thực thi sẽ chính là thứ tự khi các DPP được *add* vào trong *container*, DPP nào được chỉ định trước sẽ được triệu gọi thực thi trước, còn DPP nào được chỉ định sau sẽ được triệu gọi thực thi sau. Việc triệu gọi các DPP sẽ diễn ra tuần tự, do đó nếu như một trong số các DPP có thi hành những xử lý treo (ví dụ một vòng lặp vô tận) thì điều này có nghĩa rằng toàn bộ tiến trình diễn dịch sẽ bị treo và các DPP kế tiếp sẽ không thể thực thi xử lý cho đến khi xử lý treo hết hạn.

Để có thể thêm một cài đặt của *DefinitionPostProcessor* vào trong *container*, bạn có thể chỉ định trực tiếp bằng *code* thông qua *Definition Manager* của container hoặc gián tiếp thông qua lúc *configuration.*

Giả sử bạn có *Class ABC* là một cài đặt của *DefinitionPostProcessor* như sau:

```
class ABC implements DefinitionPostProcessor {

	@Override
	public void afterPost(Annotation[] annoArray, AnnoMeta annoMeta)
			throws DefinitionPostException {
		System.out.println("====== after post ======");
	}

	@Override
	public void beforePost(Annotation[] annoArray, AnnoMeta annoMeta)
			throws DefinitionPostException {
		System.out.println("====== before post ======");
	}

	@Override
	public boolean catchException(Exception ex, Annotation[] annoArray,
			AnnoMeta annoMeta, boolean bool) {
		return true;
	}
}
```

Thêm DPP trực tiếp thông qua *Definition Manager*:

```
public class Test {

	@SuppressWarnings("unchecked")
	public static void main(String[] args) {

		InjectCreator injector = JGentle.buildInjectCreator(Config.class);
		DefinitionManager defManager = injector.getDefManager();

	 // Thêm DPP vào container
		defManager.addDefinitionPostProcessor(new ABC());
	  ... hoặc ...
		defManager.addDefinitionPostProcessor(ABC.class);
	}
}
```

Thêm DPP lúc *configuration*:

```
abstract class Config extends AbstractConfig {

	@Override
	 public void configure() {

		addDefinitionPostProcessor(ABC.class);
	  ...hoặc...
		addDefinitionPostProcessor(new ABC());
	}
}
```

Cả hai cách thức *add* các *DefinitionPostException* đều có hành xử và kết quả tương tự như nhau, chỉ khác một điều rằng nếu như bạn chỉ định tại thời điểm *configuration* thì tất cả các *Definition* được chỉ định diễn dịch tại thời điểm *configuration* đều được áp dụng DPP, còn

nếu bạn chỉ định tại thời điểm *run-time* thông qua *DefinitionManager* thì nếu trong lúc thực thi *configuration* có thực thi diễn dịch *Definition* nào đó thì các Definition này đều không bị ảnh hưởng gì bởi DPP vì lý do tại thời điểm này, DPP chưa hề tồn tại. *Khuyến cáo rằng nếu như không có lý do gì đặc biệt, bạn nên chỉ định các* DPP *tại thời điểm configuration.*

Bạn có thể add nhiều DPP khác nhau bằng cách triệu gọi nhiều lần phương thức *addDefinitionPostProcessor* hoặc thông qua tham số *varara* của phương thức *addDefinitionPostProcessor* như sau:

```
abstract class Config extends AbstractConfig {

	@Override
	 public void configure() {
		addDefinitionPostProcessor(ABC.class, DEF.class, XYZ.class);
	 }
}
```

… và dĩ nhiên rằng thức tự add DPP chính là thứ tự thực thi của chúng.

4.3.9.2 - AnnotationBeanProcessor - Tùy biến tiến trình diễn Annotation

Nếu như *DefinitionPostProcessor* là một *EP* cho phép bạn can thiệp vào các vị trí trước, sau hoặc thời điểm *throws exception* trong khi diễn dịch *Definition* thì AnnotationBeanProcessor lại là một *EP* cho phép bạn can thiệp vào các vị trí diễn dịch khi tiến trình xử lý diễn dịch một *annotation* hoạt động. Hay nói cách khác, AnnotationBeanProcessor-ABP cho phép bạn can thiệp vào tiến trình diễn dịch ở mức *annotation* thay vì ở mức *Definition* như *DPP*.

Khác với *DefinitionPostProcessor* chỉ tương ứng với một *interface* chỉ định của một chức năng, AnnotationBeanProcessor bao gồm 2 loại khác nhau ứng với 2 *interface* chỉ định khác nhau, và thực hiện 2 chức năng khác nhau của *EP* khi diễn dịch *annotation*, đó là AnnotationPostProcessor và AnnotationHandler.

4.3.9.2.1 - AnnotationPostProcessor

EP-AnnotationPostProcessor (APP) cũng tương tự như *DefinitionPostProcessor,* tương ứng với một *interface* cùng tên AnnotationPostProcessor (nằm trong gói package *org.exxlabs.jgentle.core.reflection.aohreflect.annohandler*). AnnotationPostProcessor *interface* bản chất chính là một AnnotationBeanProcessor do *extends* từ AnnotationBeanProcessor *interface*. Thông qua ***EP*** này bạn có thể can thiệp vào tiến trình diễn dịch *annotation trước* hoặc *sau* khi việc diễn dịch *Definition* được tiến hành, *interface* AnnotationPostProcessor bao gồm 3 *callback methods* sau:

```
void before(T anno, AnnoMeta parents, Annotation[] listAnno,
                    Object objConfig) throws AnnotationBeanException;

void after(T anno, AnnoMeta parents, AnnoMeta annoMeta,
                    Annotation[] listAnno, Object objConfig)
                    throws AnnotationBeanException;

void catchException(Exception ex, T anno, AnnoMeta parents,
                    AnnoMeta annoMeta, Annotation[] listAnno, Object objConfig)
                    throws AnnotationBeanException;
```

Như *DefinitionPostProcessor,* APP cũng cung cấp các *callback methods* có chức năng gần như tương tự DPP với *before method* thực thi các xử lý trước khi *annotation* chỉ định được diễn dịch, còn *after method* sẽ thực thi các xử lý ngay sau khi *annotation* được diễn dịch xong, và *catchException method* sẽ chận hứng các *exception* được ném ra bởi 2 *methods* trên nếu có (ngoại lệ mặc định ở đây là AnnotationBeanException).

Khác với DPP có thể cho phép nhiều cài đặt khác nhau cùng chỉ định trong 1 *container*, APP chỉ cho phép 1 và duy nhất chỉ 1 cài đặt của APP ứng với một kiểu *annotation* chỉ định trong 1 *container.* Hay nói cách khác ứng với mỗi *annotation* khi được diễn dịch sẽ chỉ có 1 APP cho phép được cài đặt trong 1 *container*. Và dĩ nhiên rằng khi *implement* AnnotationPostProcessor *interface,* bạn nên chỉ định kiểu *general type* do *interface* AnnotationPostProcessor đã quy định ứng với kiểu *annotation type* mà cài đặt chỉ định tương ứng. Giả sử bạn có một *annotation* như sau:

```
@Retention(RetentionPolicy.RUNTIME)
@interface TestAnno {
        String name() default "JGentle";
}
```

Bạn có thể *implement* *AnnotationPostProcessor* *interface* tương ứng với *annotation* TestAnno như sau:

```
class MyAPP implements AnnotationPostProcessor<TestAnno> {

        @Override
        public void after(TestAnno anno, AnnoMeta parents, AnnoMeta annoMeta,
                        Annotation[] listAnno, Object objConfig)
                        throws AnnotationBeanException {
                // xử lý logic
        }

        @Override
        public void before(TestAnno anno, AnnoMeta parents, Annotation[] listAnno,
                        Object objConfig) throws AnnotationBeanException {
                // xử lý logic
        }

        @Override
        public void catchException(Exception ex, TestAnno anno, AnnoMeta parents,
                        AnnoMeta annoMeta, Annotation[] listAnno, Object objConfig)
                        throws AnnotationBeanException {
                // xử lý logic
        }
}
```

Thực thi việc *implements* như trên đã chỉ định cho *container* hiểu rằng *MyAPP class* là

một *AnnotationPostProcessor* và *annotation* tương ứng của nó là *TestAnno annotation*.

Khác với *catchException method* của DPP, có kiểu dữ liệu trả về là *boolean* cho phép *container* biết tiến trình diễn dịch sẽ tiếp tục hay không, *catchException method* của APP có kiểu trả về là *void*, không trả về một dữ liệu nào cả, thay vào đó *method* này lại cho phép ném ra ngoại lệ *AnnotationBeanException*. *Container* sẽ nhận biết được việc ném ra ngoại lệ này của *catchException*, và nếu *container* nhận được ngoại lệ này trong lúc thực thi sẽ lập tức cho dừng tiến trình diễn dịch. Hay nói cách khác, việc ném ra một ngoại lệ *AnnotationBeanException* trong xử lý của *catchException* sẽ là điểm dấu hiệu cho *container* biết rằng cần phải dừng việc xử lý diễn dịch *annotation*.

> *Lưu ý rằng chỉ khi ngoại lệ AnnotationBeanException được ném ra trong khi catchException thực thi mới được container nhận biết như là một yêu cầu nên ngưng tiến trình diễn dịch của annotation hiện hành, còn tất cả những run-time exceptions khác nếu có được ném ra trong khi catchException xử lý đều không được container quan tâm. Ngoài ra, khi nhận được ngoại lệ AnnotationBeanException từ catchException method, container sẽ dừng việc diễn dịch annotation tương ứng (nghĩa là bỏ qua việc diễn dịch này) chứ không phải là dừng toàn bộ quá trình diễn dịch Definition. Hay nói cách khác, sau khi nhận được ngoại lệ AnnotationBeanException, container sẽ bỏ qua việc diễn dịch annotation đang thi hành, và tiếp tục diễn dịch annotation kế tiếp nếu có.*

Sau khi đã thực thi các cài đặt cần thiết của một APP (*implements* AnnotationPostProcessor *interface*), bạn có thể dễ dàng add APP của mình vào trong *container* thông qua *configuration* hoặc thông qua các *methods* được cung cấp trong *DefinitionManager* như sau:

Add APP thông qua cấu hình:

```
abstract class Config implements Configurable {

    @Override
    public void configure() {
        addAnnotationBeanProcessor (TestAnno.class, MyAPP.class);
```

```
        }
}
```

Hoặc :

```
abstract class Config implements Configurable {

        @Override
        public void configure() {
                MyAPP myApp = new MyAPP();
                addAnnotationBeanProcessor (TestAnno.class, myApp);
        }
}
```

Add APP thông qua *Definition Manager*:

```
public class PostProcessorTest {

        public static void main(String[] args) {

                InjectCreator injector = JGentle.buildInjectCreator();
                DefinitionManager dm = injector.getDefManager();

                dm.addAnnotationBeanProcessor(TestAnno.class, new MyAPP());
        }
}
```

Ngoài ra khi sử dụng DefinitionManager để điều khiển APP bạn còn có thể xóa bỏ hoặc thay thế một APP chỉ định thông qua *removeAnnotationBeanProcessor* method và *replaceAnnotationBeanProcessor* method:

```
public <T extends Annotation> void replaceAnnotationBeanProcessor(
                        AnnotationBeanProcessor<T> oldHandler,
                        AnnotationBeanProcessor<T> newHandler)

public <T extends Annotation> AnnotationBeanProcessor<?>
                         removeAnnotationBeanProcessor (Class<T> key)
```

Bạn để ý rằng tất cả các *method* xử lý cho *AnnotationPostProcessor* cả thời điểm cấu hình lẫn cả trong *DefinitionManager* lúc *run-time* đều có cùng kiểu đối số truyền là *AnnotationBeanProcessor* chứ không phải là *AnnotationPostProcessor*. Đó là do *AnnotationPostProcessor* thực chất chính là một *extension* của *AnnotationBeanProcessor*. Cũng tương tự như *AnnotationHandler* (một *extension* khác của *AnnotationBeanProcessor*), cũng được JGentle container quản lý dựa trên một giao diện chung *AnnotationBeanProcessor*. Bạn chỉ cần *implements* thể hiện phù hợp, JGentle sẽ tự động nhận biết được chức năng và xử lý các tác vụ theo đúng chức năng mà cài đặt của bạn chỉ định.

4.3.9.2.2 - AnnotationHandler

Như đã nói phía trên, *AnnotationHander* (AHR) chính là một *extension* khác của *AnnotationBeanProcessor*, tương ứng là một *interface* cùng tên (*AnnotationHandler interface*) nằm trong gói *package org.exxlabs.jgentle.core.reflection.aohreflect.annohandler*.

Không như *AnnotationPostProcessor* chỉ cho phép bạn can thiệp vào các điểm khác nhau kế cận "thao tác diễn dịch" (trước, sau hoặc lúc ném ra ngoại lệ), thì AHR lại cho phép bạn thực sự chỉ định việc diễn dịch một *annotation* sẽ thực thi như thế nào. "Chỉ định việc diễn dịch" ở đây có nghĩa rằng bạn hoàn toàn có thể kiểm soát được quá trình diễn dịch này, chỉ định việc diễn dịch một *annotation* chỉ định theo một cấu trúc quản lý dữ liệu *metadata* riêng (dĩ nhiên vẫn dựa trên *AnnoMeta* của JGentle), và có thể tùy biến nội dung diễn dịch từ *annotation* thành *AnnoMeta*.

Thông thường, tiến trình chuyển đổi nội dung từ *annotation* thành *AnnoMeta* này đã được JGentle thực thi tự động (nếu như không có bất kì AHR nào chỉ định). Tuy nhiên, mặc

dù là hiếm, nhưng đôi lúc có khi bạn cũng sẽ cần đến khả năng này để có thể quản lý nội dung *AnnoMeta* theo chiều sâu, hoặc đối với một vài *annotation* đặc biệt bạn muốn tự thực hiện tiến trình diễn dịch này và trong các trường hợp tương tự thế ..., thì với AHR bạn hoàn toàn có thể làm chủ được nội dung diễn dịch của một *annotation* tương ứng.

Nội dung cần cài đặt của *AnnotationHandler interface* chỉ có 1 *callback method* duy nhất như sau:

```
public interface AnnotationHandler<T extends Annotation> extends
            AnnotationBeanProcessor<T> {

    public AnnoMeta handleVisit(T annotation, AnnoMeta annoMeta,
                DefinitionManager defManager);
}
```

Khi cài đặt của bạn thể hiện hóa *handleVisit method*, chính là đã hiện thực thao tác diễn dịch một *annotation* chỉ định thành một AnnoMeta, với đối tượng trả về chính là một *AnnoMeta* sau khi diễn dịch. Tham số truyền *annotation* chính là thực thể *annotation* với kiểu T tương ứng, còn *annoMeta* chính là *AnnoMeta* cha (là *root_AnnoMeta* của *Definition* hiện hành) chứa *AnnoMeta* sắp được diễn dịch. Việc sử dụng *AnnotationHandler* cũng đơn giản tương tự như APP, bạn chỉ cần *implements AnnotationHander interface*, thực hiện các thao tác diễn dịch trong *handleVisit method* và cuối cùng là add AHR vào trong *container*.

```
class MyAHR implements AnnotationHandler<TestAnno> {

    @Override
    public AnnoMeta handleVisit(TestAnno annotation, AnnoMeta annoMeta,
                DefinitionManager defManager) {
        // có thể sử dụng MetaDataFactory để khởi tạo dữ liệu metadata
    }
}
```

Việc chỉ định cài đặt như trên gần như tương tự với APP ngoại trừ *method* cần hiện

thực. Và dĩ nhiên, AHR cũng tương tự như APP, chỉ cho phép một và chỉ một AHR duy nhất chỉ định trên một *annotation* tương ứng, do đó kiểu *general type* hiện thực trên *AnnotationHandler* chính là kiểu type của *annotation* chỉ định của AHR.

Sau khi cài đặt xong *AnnotationHander*, bạn có thể *add AHR* vào trong *container* theo cách hoàn toàn tương tự như APP phía trên, trong trường hợp này JGentle *container* đủ thông minh để nhận dạng được *AnnotationBeanProcessor* bạn vừa add vào là APP hay là một AHR hoặc thậm chí là một tổng hợp của cả 2 loại trên dựa vào kiểu *interface* mà *object class* chỉ định đã *implements*.

4.3.9.2.2.a - Triệu gọi thực thi diễn dịch annotation

Tuy rằng thông qua AHR bạn hoàn toàn có thể làm chủ được quá trình diễn dịch của một *annotation* nhưng JGentle vẫn cung cấp thêm cơ chế cho bạn có thể tự triệu gọi việc diễn dịch *annotation* của JGentle thông qua API của nó lúc *run-time*. Việc có thể tùy ý triệu gọi tiến trình diễn dịch của JGentle cho phép bạn hoàn toàn có thể làm chủ được nội dung diễn dịch khi cần, và linh hoạt hơn trong việc quản lý dữ liệu *metadata* của *annotation*. Khuyến cáo rằng, trong hầu hết các trường hợp, nếu không có yêu cầu gì đặc biệt, "nhà phát triển" nên triệu gọi tiến trình diễn dịch nội tại của JGentle, sau đó tùy biến dựa trên nội dung dữ liệu này để đảm bảo dữ liệu *metadata* tuân thủ các quy tắc chuẩn của JGentle thay vì tự khởi tạo dữ liệu *metadata* từ đầu.

Giả sử TestAnno *annotation* có nội dung chỉ định như sau:

```
@Retention(RetentionPolicy.RUNTIME)
@interface TestAnno {
	boolean value() default true;
}
```

Thực thi cài đặt AHR:

```
class MyAHR implements AnnotationHandler<TestAnno> {
```

```
        @Override
        public AnnoMeta handleVisit(TestAnno annotation, AnnoMeta annoMeta,
                        DefinitionManager defManager) {

                if (annotation.value() == false) {

                        return ReflectToolKit.buildAnnoMeta(annotation, annoMeta,
                                        defManager);
                }
                else {
                        // Thực thi diễn dịch
                        return null;
                }
        }
}
```

Phương thức *buildAnnoMeta* là một *static method* nằm trong ReflectToolKit *class*, là một *class* tiện ích đi kèm hệ thống JGentle API, cho phép thực thi một số các thao tác *reflection* cụ thể khi vận hành hệ thống JGentle container. Và trong đó *buildAnnoMeta* là một *method* cho phép bạn tự tay triệu gọi việc diễn dịch thông tin một *annotation* thành *AnnoMeta* theo các quy tắc mặc định của JGentle. Đối tượng trả về của *method* này chính là một *AnnoMeta* sau khi đã được diễn dịch.

4.3.9.2.3 - Kết hợp AnnotationPostProcessor và AnnotationHandler

Lưu ý rằng khi bạn add một APP hay một AHR vào trong *container*, *container* đều tự động nhận biết thực thể bạn vừa chỉ định là một *AnnotationBeanProcessor*. Do đó bạn không thể vừa *implements* APP trên một cài đặt và vừa *implements* AHR trên một cài đặt khác của cùng một *annotation* và add chúng vào *container*. Vì lý do rằng *container* chỉ chấp nhận 1 và chỉ 1 thể hiện *AnnotationBeanProcessor* duy nhất ứng với 1 *annotation*. Như vậy, trong trường hợp này, cài đặt nào được *add* vào *container* sau sẽ đè lên cài đặt trước đó nếu có.

Để có thể có cả 2 thể hiện APP và AHR cho cùng một *annotation* bạn có thể kết hợp cả 2 loại APP và AHR vào trong một cài đặt duy nhất. Do cả *AnnotationPostProcessor* và *AnnotationHander* đều là những mở rộng của *AnnotationBeanProcessor*, nhờ vậy bạn có thể *implements* cùng lúc cả 2 *interface* là *AnnotationPostProcessor* và *AnnotationHandler* để chỉ định một cài đặt vừa là một APP, đồng thời có thể vừa là một AHR.

```
class AHRandAPP implements AnnotationPostProcessor<TestAnno>,
AnnotationHandler<TestAnno> {

	@Override
	public void after(TestAnno anno, AnnoMeta parents, AnnoMeta annoMeta,
			Annotation[] listAnno, Object objConfig) throws AnnotationBeanException {
	}

	@Override
	public void before(TestAnno anno, AnnoMeta parents, Annotation[] listAnno,
				Object objConfig) throws AnnotationBeanException {
	}

	@Override
	public void catchException(Exception ex, TestAnno anno, AnnoMeta parents,
				AnnoMeta annoMeta, Annotation[] listAnno, Object objConfig)
				throws AnnotationBeanException {
	}

	@Override
	public AnnoMeta handleVisit(TestAnno annotation, AnnoMeta annoMeta,
				DefinitionManager defManager) {
			return null;
	}
}
```

Với cách *implements* này, cài đặt ***AHRandAPP*** *class* thâu tóm toàn bộ tiến trình diễn

dịch một *annotation* chỉ định, và về mặt bản chất việc thực thi diễn dịch một *annotation* thông qua xử lý tự động của JGentle cũng chỉ đơn giản là tự động hóa các thao tác trên APP và AHR tương tự như ***AHRandAPP*** *class*.

4.3.9.3 - Đăng kí Extension-Points thông qua IoC

4.3.9.4 - Kết hợp InjectCreatorAware và PointStatus trong Extension-Points

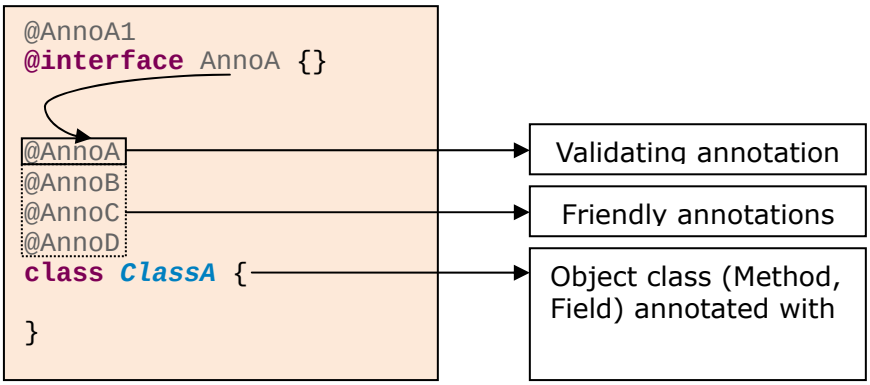
4.3.10 - Validate dữ liệu annotation trước khi diễn dịch

Như đã giới thiệu ở phần đầu, JGentle không những cung cấp cơ chế quản lý dữ liệu *annotation*, chuyển đổi thông tin các *annotations* thành *Definition* mà còn cho phép các thông tin *annotation* chỉ định diễn dịch có thể được *validate* tại thời điểm *run-time*. Hệ thống "*Annotation Validation*" trong JGentle không những cho phép nhà phát triển *validate* nội dung của từng thực thể *annotation* trước khi diễn dịch mà còn cho phép *validate* theo một hệ cây phân cấp (nếu như *annotation* được *validate* có được chỉ định một *annotation* khác) và đồng thời còn cho phép nhà phát triển tự xây dựng các *annotation validator* của riêng mình, cho phép sử dụng các *annotation* tự tạo như là một công cụ để *validate* thông tin nội dung của các *annotation* tương ứng khác.

Thông qua hệ thống "*Annotation Validation*", không những bạn có thể can thiệp vào tiến trình *validation* mà còn có thể chứng thực nội dung của *annotation*, danh sách các *friendly annotations* đi cùng của nó, kể cả "*thực thể*" có chỉ định *annotation* và thậm chí có thể tương tác với *DefinitionManager* để thực thi các tác vụ *logic* ngay khi *validate* thông tin *annotation*. Thông tin dữ liệu *annotation* giờ đây không những được kiểm tra dựa trên thông tin nội dung nội tại của nó mà còn có thể được kiểm tra và xác định vị trí của nó (*annotation*) được chỉ định lên thực thể nào, nó có đính kèm thông tin *annotation* khác hay không, và điều kiện kiểm tra của các *annotation* khác mà nó có đính kèm sẽ ảnh hưởng thế nào đến tiến trình *validate* của chính nó. Tiến trình thực thi *validate* giờ đây tạo thành một mắc xích có hệ thống, không chỉ *validate* trên thông tin *annotation* chỉ định mà cả các *annotation* liên quan và vị trí chỉ định của chúng. Nhờ đó mà bạn có thể xác định và kiểm tra được dữ liệu thông tin của các *annotation* có hợp lệ hay không trong nhiều trường hợp và

vị trí chỉ định khác nhau trên các thực thể khác nhau. Ngoài ra, JGentle còn cho phép đăng kí các *validator* tự tạo chận hứng các ngoại lệ ném ra bởi các *validator* chính của một *annotation* chỉ định trong một hệ cây phân cấp.

Mô hình cấu trúc các thành phần chỉ định sẽ được đưa vào tiến trình validate như sau:



Theo hình vẽ, các *annotations* như: AnnoB, AnnoC, AnnoD là các *friendly annotations* đi kèm trong cùng một danh sách *annotation list* với AnnoA *annotation* (*annotation* được chỉ định đang được *validate*). AnnoA *annotation* có thể có hoặc không được chỉ định các *annotation* khác, trong trường hợp này AnnoA *annotation* có được chỉ định đính kèm một *annotation* khác (đó là AnnoA1 *annotation*). Và cuối cùng là ClassA, chính là đối tượng chủ thể chỉ định được đính kèm *annotation* đang được *validate*. Trong các trường hợp khác đối tượng chủ thể có thể là *Method, Field,* hoặc thậm chí là một *Annotation Class*.

Để có thể thực thi một cài đặt *validator* cho một *annotation* chỉ định bạn chỉ cần đơn giản *implements AnnotationValidator interface* (nằm trong gói package *org.exxlabs.jgentle.core.metadatahandling.aohhandling.pvdhandler*) và hiện thực 2 *callback methods* là *validate method* và *catchException method*:

```
public interface AnnotationValidator<T extends Annotation> {

    void validate(T annotation, Annotation[] annoList, Object object,
                  Class<?> clazz, DefinitionManager definitionManager);

    <V extends RuntimeException, U extends Annotation> boolean catchException(
                  V exception, U annotation);
```

```
}
```

Phương thức *validate* chính là phương thức sẽ xử lý chính cho tiến trình *validate annotation*, với các đối số truyền: annotation chính là đối tượng *annotation* hiện hành đang được *validate* với kiểu T chỉ định, annoList chính là danh sách các *friendly annotations* đi cùng annotation, object chính là đối tượng gốc (chủ thể) được chỉ định annotation và definitionManager chính là đối tượng *Definition Manager* của *container* hiện hành. Và cuối cùng là clazz, điều đặc biệt ở đối số này là không mang một giá trị nhất định, mà clazz sẽ mang những hình thái giá trị khác nhau trong những trường hợp khi object mang những giá trị khác nhau. Trong trường hợp object:

- ***Nếu là Method hoặc Field***: chủ thể mà annotation được chỉ định đính kèm là một *Method* hoặc *Field*, clazz khi này sẽ chính là *object class* của *Class* mà *method* hoặc *field* đó được khai báo bên trong.
    - *Lưu ý rằng trong trường hợp* object *là Method có thể* annotation *chỉ định đang được validate không phải là* annotation *được đính kèm lên method đó mà có thể là* annotation *được đính kèm lên parameter của method. Và trong cả 2 trường hợp này* object *đều là method chỉ định.*
- ***Nếu là Class***: chủ thể mà annotation được chỉ định đính kèm là một *Class (Class ở đây có thể là Class, Interface hay thậm chí Enum … hay bất kì chủ thể nào có thể đính kèm annotation được biểu diễn dưới dạng object class)*, trong trường hợp này clazz sẽ chính là object. Hay nói cách khác mệnh đề sau sẽ trả về *true*:
    - object.equals(clazz)
- ***Nếu là Annotation***: chủ thể mà annotation đang được *validate* lại là một *annotation* khác (*annotation* ở đây có nghĩa rằng đối tượng *annotation object*, không phải *annotation type - object class*). Trong trường hợp này, clazz sẽ chính là *annotation type* của *annotation* chủ thể (object).

Trong khi *validate method* đang thực thi xử lý, bất cứ khi nào có một *run-time*

exception được ném ra, *container* đều xem như là những dấu hiệu cho biết tiến trình *validate* có thể có những điều kiện không phù hợp, và quá trình thực thi xử lý ngay lập tức sẽ được chuyển hướng xuống cho *catchException method* xử lý. Khi này *catchException* sẽ thực thi kiểm tra các *annotation* lần cuối cùng và trả về kết quả là một giá trị *boolean*. Nếu kết quả trả về là ***true***, *container* sẽ xem như dữ liệu hợp lệ, … điều này có nghĩa rằng dữ liệu *annotation* hoặc những thông tin của các thực thể liên quan đã được điều chỉnh cho hợp lệ với xử lý *validate* và phương thức *validate* sẽ được triệu gọi lại để kiểm tra thông tin cần *validate* mới. Hay nói cách khác, nếu như *catchException method* trả về ***true***, *container* sẽ tự động triệu gọi ngược lại phương thức *validate* 1 lần nữa để kiểm tra dữ liệu thông tin mới của *annotation*. Do đó, tiến trình *validate* chỉ thực sự kết thúc khi không còn bất kì một *run-time exceptions* nào được ném ra bởi *validate method*. Còn nếu kết quả trả về của *catchException method* là ***false***, container sẽ nhận biết như là dấu hiệu *annotation* hiện hành là không hợp lệ và sẽ *throw* ngược lại *run-time exception* nhận được từ *catchException method* ra ngoài.

Đến khi nào *annotation* được kiểm tra *validate* và không còn bất kì một ngoại lệ nào được ném ra thì *annotation* đó được xem như là hợp lệ và có thể được sử dụng để diễn dịch thành *Definition* hay sử dụng cho các mục đích *logic* khác … Và theo mặc định của tiến trình diễn dịch *Definition* của JGentle, trước khi diễn dịch một *annotation* bất kì thành *AnnoMeta*, JGentle đều kiểm tra *validate* thông tin *annotation* (nếu *annotation* chỉ định có đăng kí một *validator* tương ứng).

4.3.10.1 - Đăng kí annotation và annotation validator

Ngoại trừ khi bạn chỉ định tường minh thực thi *validate* một *annotation* thông qua mã lệnh lúc *run-time*, trong hầu hết các các trường hợp còn lại ví dụ như khi diễn dịch một *annotation*, tiến trình *annotation validation* của JGentle không phải kiểm tra trên tất cả các *annotations* của thực thể gốc, mà chỉ thực thi *validate* trên các *annotation* đã được *register* vào trong *container*, để tránh trường hợp *validate* trên các *annotations* không cần thiết (vd như @Target). Cũng tương tự như thế đối với các *annotation validator,* các *validator* này chỉ có thể *add* vào *container* khi và chỉ khi *annotation* tương ứng của nó đã được đăng kí trước đó.

Do đó, để có thể giúp cho *container* có thể nhận biết được đâu là các *annotations* hợp lệ cần *validate*, và chỉ định cho *container* các *validators* tương ứng bạn cần phải đăng kí

annotations, và các *validators* vào trong JGentle *container* thông qua đối tượng *Annotation Register*.

Cũng tương tự như *Definition Manager* quản lý các *Definitions*, *Annotation Register* cũng là một đối tượng của JGentle *container*, chịu trách nhiệm quản lý, đăng kí, cũng như điều khiển các *annotation* trong *container*, đồng thời cũng quản lý các *validator* tương ứng của từng *annotation* có trong hệ thống. Để có thể lấy ra được đối tượng *Annotation Register* bạn chỉ cần đơn giản triệu gọi phương thức *getAnnotationRegister* thông qua *DefinitionManager* như sau:

```
public class Test {
	public static void main(String[] args) {
		InjectCreator injector = JGentle.buildInjectCreator();
		DefinitionManager defManager = injector.getDefManager();
		...
		AnnotationRegister annoReg = defManager.getAnnotationRegister();
		...
	}
}
```

Sau đó để có thể đăng kí *annotation* chỉ định vào trong *container*, thông qua *Annotation Register* bạn có thể thực hiện thông qua 2 *methods* sau:

```
public void registerAnnotation(Class<? extends Annotation> anno)

và ...

public <T extends Annotation> void registerAnnotation(Class<T> anno,
		AnnotationValidator<T> validator)
```

Một phương thức *registerAnnotation* sẽ đăng kí một annotation chỉ định thông qua đối số là *object class (annotation type)* của *annotation* cần đăng kí, còn phương thức còn lại vừa cho phép đăng kí *annotation* vừa cho phép chỉ định *add validator* tương ứng của nó. Do

đó, đồng thời bạn vừa có thể *register annotation* chỉ định, vừa *add validator* tương ứng vào trong *container*.

Nếu trong trường hợp annotation chỉ định tương ứng của validator đã được đăng kí, bạn có thể sử dụng phương thức addValidator mà AnnotationRegister cung cấp để đăng kí validator như sau:

```
public <T extends Annotation> boolean addValidator(Class<T> anno,
                        AnnotationValidator<T> validator)
```

Lưu ý rằng đối số truyền của *addValidator method và registerAnnotation method* có vẻ như tương tự nhau nhưng thật sự lại hoạt động hơi khác biệt. Trong khi *registerAnnotation method* khi được thực thi sẽ tự động đăng kí *annotation*(anno) truyền vào, sau đó sẽ đăng kí *validator* tương ứng còn *addValidator method* thì ngược lại, luôn xem anno như là mặc định đã được đăng kí, *addValidator method* chỉ làm một nhiệm vụ duy nhất, đó là đăng kí validator tương ứng với anno chỉ định. Trong trường hợp anno chưa thực sự được đăng kí vào trong *container*, một *run-time exception* sẽ được ném ra khi *addValidator method* được thực thi. Do đó khi sử dụng *addValidator method,* bạn luôn luôn cần phải đăng kí *annotation* trước đó để đảm bảo *addValidator method* có thể thực thi đúng đắn.

4.3.10.1.1 - Đăng kí annotation thông qua Annotation Object Handler

Ngoài việc đăng kí *annotation* thông qua *Annotation Register* bạn còn có thể thực thi bằng cách thông qua đối tượng *Annotation Object Handler (AOHler)*, là đối tượng chính trong *Annotation Object Handling service*. Để có thể biết thêm chi tiết về *AOH service*, vui lòng xem phần mô tả tại Annotation Object Handling.

Đối tượng *AOHler* tương tự như *Annotation Register*, cũng cung cấp 2 phương thức *registerAnnotation* y hệt như *Annotation Register, ...* kể cả *addValidator method*. Về mặt bản chất, *AOHler* cũng chỉ đơn giản chuyển yêu cầu *invoke* từ các *methods* trên xuống đối tượng *AnnotationRegister* mà nó tham chiếu đến. Tuy nhiên, thay vì như *AnnotationRegister* chỉ có thể đăng kí từng *annotation* chỉ định bằng cách triệu gọi *registerAnnotation method,*

AOHler còn cho phép đăng kí một tập hợp nhiều *Annotations* và nhiều *validator* tương ứng khác nhau thông qua một đối tượng cấu hình ... bằng cách cung cấp thêm một *overloading* khác của *registerAnnotation method như sau:*

```
public void registerAnnotations(Class<? extends Enum<?>> enumAnnoClass)
```

Như đã mô tả ở phần quản lý cấu hình, chủ yếu JGentle *container* chỉ cung cấp 2 cách thức cấu hình, đó là thông qua ACC hoặc ATC. Nhưng thông qua các hệ thống *services* bên trong JGentle, hoặc hệ thống các *services* tự tạo từ các nhà phát triển khác, các cách cấu hình có thể thay đổi hoặc các đối tượng quản lý thông tin cấu hình có thể thay đổi tùy theo từng *services*. Ở góc độ nào đó, bản thân đối tượng AOHler cũng có thể tự xem như là một *service* ở mức cơ bản và *object class* của *enum* cung cấp cho *registerAnnotations* của *AOHler* cũng có thể xem như là một loại dữ liệu cấu hình (dữ liệu *startup*), một loại dữ liệu đầu vào giúp cho "AOHler *services"* có thể hoạt động theo ý định của nhà phát triển. Do đó mỗi một *service* đều có thể tự định nghĩa cho mình dữ liệu đầu vào riêng, tuân theo một cấu trúc nhất định hoặc kết hợp với *annotation* để tự tạo ra các dữ liệu đối tượng cấu hình riêng biệt.

Tuân theo cấu trúc cấu hình của *registerAnnotations*(*Class*<? **extends** *Enum*<?>> enumAnnoClass) *method*, đối số đầu vào cần phải là một *object class* được *extends* từ *Enum interface*, hay nói cách khác thông tin mô tả tập hợp các *annotation* phải được chứa đựng trong một *enum*. Và dĩ nhiên *enum* này cần phải tuân theo một cấu trúc nhất định, cấu trúc này được quy định bởi AOHler. Giả sử bạn có một *2 annotation @AnnoA và @AnnoB* nằm trong *package example* như sau:

```
package example;

@Retention(RetentionPolicy.RUNTIME)
public @interface AnnoA {
}

@Retention(RetentionPolicy.RUNTIME)
```

```
@interface AnnoB {
}
```

Việc viết một *enum* cất trữ thông tin *annotation @AnnoA* chỉ cần đơn giản là viết khai báo *Enumeration Constant (EC)* như là đường dẫn đầy đủ của *annotation class* cần đăng kí:

```
enum AnnoRegister {
        example_ AnnoA, example_ AnnoB;
}
```

Enum AnnoRegister trong ví dụ trên chỉ định 2 EC, đó chính là tên và đường dẫn đầy đủ của *annotation AnnoA và AnnoB*. Lưu ý rằng kí tự phân cách giữa các *package và class* từ kí tự dấu chấm "**.**" được chuyển thành kí tự gạch dưới "**_**" . Vd: nếu như *AnnoA* nằm trong *package org.exxlabs.jgentle.example.AnnoA* sẽ được viết lại với hình thức là tên của EC như sau *org_exxlabs_jgentle_example_AnnoA*. Sau đó chỉ cần đưa *object class của AnnoRegister* vào như là tham số truyền cho *registerAnnotation method* của AOHler:

```
...
InjectCreator injector = JGentle.buildInjectCreator();
AnnotationObjectHandler aohler = injector.getAnnoObjectHandler();
aohler.registerAnnotations(AnnoRegister.class);
...
```

Việc sử dụng kí tự "_" thay cho kí tự phân cách dấu "." giữa *class* và *package* trong khai báo EC chỉ đơn thuần là một *convention* mặc định trong JGentle. Vậy trong những trường hợp còn lại, trong tên của *package* hoặc *annotation class* nếu cũng có chứa kí tự "_" ngay trong tên định danh, … thì *convention* này sẽ không còn hoạt động đúng đắn. Do đó, ngoài cách sử dụng *convention* mặc định, JGentle vẫn cho phép định nghĩa *annotation class* như là một thuộc tính thành phần của EC trong *enum* như sau:

```
enum AnnoRegister implements GetAnnotationClass {

	AnnoA(AnnoA.class), AnnoB(AnnoB.class);
	Class<? extends Annotation> annoClazz;

	AnnoRegister (Class<? extends Annotation> clazz) {
		this.annoClazz = clazz;
	}

	@Override
	public Class<? extends Annotation> getAnnotationClass() {
		return annoClazz;
	}
}
```

Ví dụ trên là một *enum* chỉ định một tập hợp các *annotations* cần đăng kí vào trong *container*. Các hằng liệt kê (*enumeration constant* - **EC**) của *enum* buộc cần phải chỉ định kèm theo ít nhất một thuộc tính thành phần có kiểu là ***Class***<? **extends** Annotation>, đây chính là *object class (hay Annotation Type)* của *annotation* tương ứng của EC chỉ định. Lúc này tên khai báo của EC không còn cần thiết phải chỉ định tường minh như là đường dẫn của *annotation class* nữa, mà chỉ cần khai báo như là một đại diện định danh bất kì của một EC.

Do đó, ngoài việc khai báo EC, trong *enum* còn cần phải chỉ định thêm một *constructor* với tham số đầu vào có kiểu ***Class***<? **extends** Annotation>, đồng thời *enum AnnoRegister* cần phải *implements GetAnnotationClass interface* và thực thi cài đặt cho phương thức *getAnnotationClass().* Phương thức *getAnnotationClass* chính là phương thức sẽ trả về thuộc tính *annotation class* của EC. Mặc định nếu như enum có *implements GetAnnotationClass interface, khi* thực thi *registerAnnotation method, JGentle container* sẽ tự động triệu gọi *getAnnotationClass method* trên từng *enum instance* để lấy ra được thuộc tính *annotation class*.

Trong trường hợp bạn không muốn *implements GetAnnotationClass interface*, bạn cũng có thể sử dụng *@AnnotationClass annotation* (nằm trong gói *package org.exxlabs.jgentle.configure.injecting.annotation*) chỉ định trên bất kì *method* nào bạn

muốn trong *enum* để có thể quy định cho *container* biết rằng đó chính là *method* sẽ trả về *annotation class* tương ứng của EC (dĩ nhiên rằng *method* này cần phải có kiểu trả về là *Class hoặc Class<? extends Annotation>,* nếu không một ngoại lệ sẽ được ném ra lúc *run-time*).

```
enum AnnoRegister { // không cần implements GetAnnotationClass interface
	AnnoA(AnnoA.class), AnnoB(AnnoB.class);
	Class<? extends Annotation> annoClazz;

	AnnoRegister (Class<? extends Annotation> clazz) {
		this.annoClazz = clazz;
	}

	@AnnotationClass // chỉ định method này sẽ trả về annotation class
	public Class<? extends Annotation> getAnnoClazz() {
		return annoClazz;
	}
}
```

Lưu ý rằng nếu trong trường hợp bạn vừa chỉ định *implements GetAnnotationClass interface*, vừa chỉ định *@AnnotationClass annotation*, vừa chỉ định tên khai báo EC "hợp lệ theo đường dẫn" thì *container* sẽ ưu tiên theo thứ tự sau:

Tên khai báo EC -> @AnnotationClass -> GetAnnotationClass interface

Nếu trong trường hợp *method* trả về *AnnotationClass* (được quy định bởi *@AnnotationClass* hoặc *implements* từ *GetAnnotationClass interface*) trả về kết quả là ***null***, một ngoại lệ sẽ được ném ra lúc *run-time*.

4.3.10.1.2 - Đăng kí annotation validator thông qua Annotation Object Handler

Tương tự như khi chỉ định *annotation* với *enum*, đối với *validator* tương ứng của *annotation*, *enum* chỉ định cũng cần phải *implements GetAnnotationValidator interface*, và

thực thi cài đặt cho *getAnnotationValidator method*. Giả sử tương ứng với *@AnnoA và @AnnoB annotation* bạn có các *validator là AnnoAValidator và AnnoBValidator* như sau:

Annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@interface AnnoA {
}

@Retention(RetentionPolicy.RUNTIME)
@interface AnnoB {
}

@Retention(RetentionPolicy.RUNTIME)
@interface AnnoC {
}
```

Validator:

```
class AnnoAValidator implements AnnotationValidator<AnnoA> {

	@Override
	public <V extends RuntimeException, U extends Annotation> boolean catchException(V exception, U annotation) {

		return false;
	}

	@Override
	public void validate(AnnoA annotation, Annotation[] annoList,
		Object object, Class<?> clazz, DefinitionManager definitionManager) {
```

```
            System.out.println(annotation + " is validated.");
        }
}

class AnnoBValidator implements AnnotationValidator<AnnoB> {
        @Override
        public <V extends RuntimeException, U extends Annotation> boolean
catchException(V exception, U annotation) {

            return false;
        }

        @Override
        public void validate(AnnoB annotation, Annotation[] annoList,
            Object object, Class<?> clazz, DefinitionManager definitionManager) {

            System.out.println(annotation + " is validated.");
        }
}
```

Enum:

```
enum AnnoRegister implements GetAnnotationClass, GetAnnotationValidator {

        // Chỉ định enumeration constant
        AnnoA(AnnoA.class, new AnnoAValidator()),
        AnnoB(AnnoB.class, new AnnoBValidator()),
        AnnoC(AnnoC.class, null);

        Class<? extends Annotation> annoClazz = null;
        AnnotationValidator<? extends Annotation> validator = null;
```

```
        <T extends Annotation> AnnoRegister(Class<T> clazz,
                        AnnotationValidator<T> validator) {
                this.annoClazz = clazz;
                this.validator = validator;
        }

        @Override
        public Class<? extends Annotation> getAnnotationClass() {
                return annoClazz;
        }

        @Override
        public AnnotationValidator<? extends Annotation> getAnnotationValidator() {
                return validator;
        }
}
```

Lúc này, ngoài thuộc tính thành phần chỉ định *annotation class* (annoClazz), *enum AnnoRegister* còn chỉ định thêm thuộc tính validator có kiểu là AnnotationValidator<? **extends** Annotation> quy định đối tượng *validator* tương ứng của *annotation class*. Ngoài ra, cùng với việc *implements GetAnnotationClass interface, enum AnnoRegister* còn *implements GetAnnotationValidator interface*, và thực thi cài đặt cho phương thức *getAnnotationValidator*, trả về thuộc tính validator của e*num instance*.

Cách khai báo, cũng như sử dụng *GetAnnotationValidator interface* hoàn toàn tương tự như khi vận dụng *GetAnnotationClass interface*. Nếu như muốn *enum* chỉ định không thực thi *implements*, bạn hoàn toàn có thể sử dụng *@AnnotationValidators annotation* (nằm trong gói *package org.exxlabs.jgentle.configure.injecting.annotation*) thay thế cho *GetAnnotationValidator interface* như sau:

```
enum AnnoRegister { // không implements
        // Chỉ định enumeration constant
```

```
        AnnoA(AnnoA.class, new AnnoAValidator()),
        AnnoB(AnnoB.class, new AnnoBValidator()),
        AnnoC(AnnoC.class,  null);

        Class<? extends Annotation> annoClazz = null;
        AnnotationValidator<? extends Annotation> validator = null;

        <T extends Annotation> AnnoRegister(Class<T> clazz,
                        AnnotationValidator<T> validator) {
                this.annoClazz = clazz;
                this.validator = validator;
        }

        @AnnotationClass
        public Class<? extends Annotation> getAnnoClazz() {
                return annoClazz;
        }

        @AnnotationValidators // Chỉ định method này sẽ trả về validator tương ứng
        public AnnotationValidator<? extends Annotation> getAnnoValidator() {
                return validator;
        }
}
```

Lưu ý nếu như *method* trả về đối tượng *validator* tương ứng của *annotation* trả về kết quả có giá trị là ***null***, thì khi này *container* chỉ đăng kí *annotation* tương ứng chỉ định.

4.3.10.2 - Đăng kí nhận Exception từ Annotation Parents

4.3.10.3 - Thực thi bằng tay tiến trình validate annotation lúc run-time

4.3.10.4 - Kiểm tra một object

Khi nhận được object truyền vào thông qua *validate method* của cài đặt *annotation validator*, đôi khi bạn cần muốn kiểm tra object được chỉ định truyền vào thuộc loại nào, *object class, method, field* hay *annotation*. Để có thể kiểm tra được đối tượng object nhận được có phải là một *Method, Field, Class* hay là một *Annotation*, JGentle có cung cấp một số các *static methods* nằm trong lớp tiện ích *ReflectToolKit class* cho phép nhận biết được loại đối tượng object.

```
public static boolean isField(Object obj)

public static boolean isMethod(Object obj)

public static boolean isClass(Object obj)

public static boolean isAnnotation(Object obj)
```

Hoặc có thể kiểm tra bằng phương thức *getObjectType,* là một *static method* nằm trong lớp *ObjectChecker class (là một lớp tiện ích khác trong JGentle tương tự như ReflectToolKit class)* như sau*:*

```
@Override
public void validate(A annotation, Annotation[] annoList, Object object,
                     Class<?> clazz, DefinitionManager defManager) {

    if (ObjectChecker.getObjectType(object).equals(ObjectChecker.CLASS)) {

        // do something
    }
    else if (ObjectChecker.getObjectType(object).equals(ObjectChecker.DONTKNOW))
    {
        throw new RuntimeException();
```

```
        }
}
```

Phương thức *getObjectType* sẽ trả về một giá trị *Integer* mô tả loại kiểu của object, đồng thời *ObjectChecker class* cũng cung cấp một loạt các *static fields* tương ứng với các chỉ số *Integer* chỉ định kiểu *type* của object.

4.3.10.5 - Validatable Annotation

4.4 - Dependency Injection

4.4.1 - Giới thiệu

4.5 - Deep Dependency Injection

4.5.1 - Giới thiệu

4.6 - Annotation Dependency Injection

4.6.1 - Giới thiệu

4.7 - Aspect Oriented Programming – AOP trong JGentle

4.7.1 - Giới thiệu

4.7.2 - Advice

4.7.3 - Aspect

4.7.4 - Weaving

4.7.5 - Pointcut

4.7.6 - JoinPoint

4.7.7 - Introduction

4.8 - Annotation Object Handling

4.8.1 - Giới thiệu

5 - Integration

- JGentle Remoting – Web Services
- JGentle JDBC
- JGentle Data Access framework
- JMX Support
- JCA Support
- Spring Integration

5.1 - Remoting – Web services

5.1.1 - RMI Interation

5.1.2 - Hessian – Burlap Integration

5.1.3 - Web Services

5.1.4 - JMS

5.2 - JGentle JDBC

5.2.1 - Giới thiệu JGentle JDBC

5.3 - Data Access

5.3.1 - Giới thiệu

5.3.2 - Hibernate

5.3.3 - iBatis

5.3.4 - JDO

5.4 - JMX Support

5.4.1 - Giới thiệu JGentle JMX

5.5 - JCA Support

5.5.1 - Giới thiệu JGentle JCA

5.6 - Spring integration

5.6.1 - Giới thiệu

6 - JGentle Web

6.1 - Giới thiệu JGentle Web

7 - JGentle Security

7.1 - Giới thiệu JGentle Security

8 - JGentle Services

- JGentle GUI
- Event Service
- Object Pooling
- Thread Pooling
- Queued Component
- Data Locator – Registry Object

8.1 - JGentle GUI

8.2 - Event Service

8.2.1 - Giới thiệu chung

8.2.2 - Loosely Coupled vs Tightly Coupled

8.2.3 - Event Class

8.2.3.1 - Event of Event Class

8.2.3.2 - Register Event

8.2.4 - Publisher

8.2.4.1 - Persistent Publisher

8.2.4.2 - Transient Publisher

8.2.5 - Subscriber

8.2.5.1 - Multiple Subscriber

8.2.5.2 - Subscriber priority

9 - Phụ lục A - Annotation

10 - Phụ lục B – RMI

11 - Phụ lục C - Spring framework