# Tìm hiểu Play framework

## Giới thiệu

### Play framework là gì?

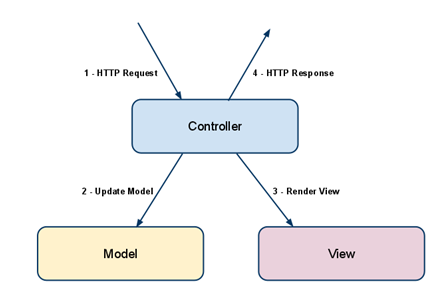
### Lịch sử phát triển

### Khác nhau giữa Play framework version 1 và version 2

## Đặc điểm Play framework

### Mô hình MVC

* The Play Framework tuân theo mô hình kiến trúc MVC (Model-View-Controller) để xây dựng các ứng dụng web. Mục đích của mô hình là để tách các phần khác nhau hoặc các lớp của một hệ thống để cho phép phát triển độc lập, kiểm thử và bảo trì.
* Một nhận thức sai lầm phổ biến của mô hình này là nó là CHỈ biểu diễn dữ liệu. Martin Fowler mô tả cách tiếp cận này là một mẫu đối lập (một phương pháp phổ biến đó là sự phản tác dụng hoặc không hiệu quả). Mô hình này là các dữ liệu đại diện và miền logic để các ứng dụng hoạt động dựa trên đó.
* Hầu hết các ứng dụng web sẽ có một cơ sở dữ liệu mà đặt phía sau mô hình dữ liệu để tăng thêm độ bền cho ứng dụng. Play làm cho sự lưu trữ dễ sử dụng JPA.
* View có trách nhiệm biểu diễn các mô hình trong một hình thức thích hợp. Trong các ứng dụng Web, thường là HTML, nhưng vẫn có thể là XML, JSON hoặc hình ảnh. Nó cũng có thể (và thường thuận lợi) có nhiều views trên mô hình dữ liệu giống nhau.
* Controller trả lời các sự kiện (thường sử dụng các hành động từ view) và xử lý chúng. Vòng đời điển hình của một controller là lắng nghe sự kiện (bằng cách lắng nghe các yêu cầu HTTP được nhận được từ trình duyệt), sau đó đọc dữ liệu được gửi như là một phần của yêu cầu (tham số được gửi cùng với một POST hoặc GET trong yêu cầu HTTP) và sau đó áp dụng các thay đổi đối với mô hình dữ liệu.



**Play MVC Lifecycle**

1-Một yêu cầu HTTP được nhận từ các trình duyệt

2-Bộ điều khiển nhận được yêu cầu và cập nhật mô hình phù hợp

3-Bộ điều khiển yêu cầu giao diện hiển thị

4-Đầu ra hiển thị trên giao diện như là 1 phản hồi HTTP cho trình duyệt

### HTTP Routing

HTTP định tuyến được cấu hình sử dụng một tập tin có tên đường dẫn, có thể được tìm thấy trong thư mục conf trong ứng dụng của bạn.

#### Mục đích

* Mục đích của file HTTP định tuyến là truyền lại các yêu cầu HTTP được nhận (từ trình duyệt hoặc không) vào các sự kiện mà sau đó được thực hiện bởi bộ điều khiển. Dựa vào loại yêu cầu được nhận (phương thức HTTP và URI), router sẽ gọi bộ điều khiển được chỉ định để thực hiện các phương pháp mong muốn.

#### Cú pháp

* Các tập tin đường dẫn chứa 3 yếu tố chính. Mục đầu tiên là phương thức HTTP (như GET hoặc POST), yếu tố thứ hai là URI (như / đăng ký) và yếu tố cuối cùng là bộ điều khiển và tên phương thức (như application.index).
* Một yếu tố cuối cùng cũng có thể được tùy chọn bao gồm để chỉ ra đối số tĩnh và / hoặc quy định cụ thể các loại nội dung.
* Nó cũng có thể để thêm các chỉ dẫn ​​vào một tập tin đường dẫn bằng cách sử dụng biểu tượng thăng (#).

#### Route ưu tiên

* Khi các tập tin cấu hình đường dẫn có thể chứa nhiều mục khác nhau, và nó có thể chứa nhiều mục trong tập tin đường dẫn có thể phù hợp với một yêu cầu HTTP, cần phải có một thứ tự ưu tiên được đặt trên từng đường dẫn trong tập tin. Để giữ cho quản lý tập tin đơn giản, thứ tự ưu tiên từ trên xuống dưới. Khi nhận được yêu cầu, nó sẽ đọc các tập tin đường dẫn một dòng tại một thời gian kiểm tra sự phù hợp. Nếu sự phù hợp được tìm thấy, nó sẽ sử dụng cái đầu tiên nó đi qua và gọi bộ điều khiển liên quan.

#### Nội dung tĩnh

* Như với hầu hết các trang web phục vụ công nghệ, Play cho phép nội dung tĩnh được phục vụ theo cách khác như nội dung động. Điều này làm cho xử lý yêu cầu nhanh hơn nhiều, và nó cũng làm cho bảo trì các ứng dụng đơn giản hơn nhiều nếu tất cả các nội dung tĩnh được giữ trong cùng một vị trí.
* Nội dung tĩnh được gọi mã mà không được tạo ra (ví dụ như các file Java và HTML) và khi có yêu cầu từ trình duyệt, chỉ đơn giản là trả lại mà không xử lý dữ liệu. Ví dụ về các dữ liệu tĩnh sẽ là javascript, css và hình ảnh.
* Nội dung tĩnh mặc định nằm trong thư mục public của ứng dụng của bạn. Để thuận tiện và thực hành tốt, các thư mục public tiếp tục chia thành các hình ảnh, javascript và stylsheets.

GET /public staticDir:public

* Đường dẫn này quy định rằng tất cả các yêu cầu được nhận bắt đầu với /public nên được giả định là nội dung tĩnh (vì nhận dạng staticDir đặc biệt) và vị trí của các nội dung trong ứng dụng là thư mục public (quy định bởi các định danh sau dấu hai chấm) .
* Nếu bạn muốn, bạn có thể chỉ định đường dẫn nội dung tĩnh hơn bằng cách sử dụng từ khóa staticDir. Tuy nhiên ưu tiên đường dẫn thông thường sẽ áp dụng như đã thảo luận trong phần trước.

### Controller

#### Mục đích

* Mục đích của controller là để tham gia cùng các phạm vi model và các sự kiện được yêu cầu từ các yêu cầu HTTP. Như đã thảo luận trong phần mô hình MVC, một controller không thực hiện business logic. Điều đó được thực hiện bởi model. Thay vào đó, controller nhận được sự kiện này, đọc các thông số được gửi đi với sự kiện và thực hiện các hành động cần thiết trên model. Sau khi hoàn thành, controller sẽ trả lại view đúng để hiển thị chính xác các kết quả của ứng dụng gọi điện thoại (thường là trình duyệt).
* Trong điều kiện của các khuôn khổ web khác, một controller có thể được coi là tương tự như các tiêu chuẩn HTTP Servlet, hoặc trong Struts nó sẽ là tương đương với các đối tượng hành động. Tuy nhiên Play đi nhiều bước hơn nữa để làm cho các ứng dụng xây dựng trang web nhanh hơn và dễ dàng hơn và controller là một trong những nơi đầu tiên bạn sẽ thấy lý do tại sao.

#### Tạo một controller

* Controller của ứng dụng là một lớp Java mà mở rộng play.mvc.Controller, chứa một số các phương thức đại diện cho hành động. Mỗi hành động là một cách cho controller đáp ứng các sự kiện từ view và thực hiện các hành động cần thiết.

LƯU Ý: Một hành động trong controller phải được công khai và tĩnh.

* Khi bạn tạo ra một ứng dụng mới trong Play, một controller sẽ tự động được thiết lập gọi là Application.java. Nó sẽ chứa một phương thức duy nhất (một hành động), mà chỉ đơn giản là làm chỉ mục cho các trang của ứng dụng.

#### Parameters

* Đối với một ứng dụng web để hoạt động như một ứng dụng, nó đòi hỏi dữ liệu được truyền qua lại giữa giao diện người sử dụng và controller của ứng dụng. Đối với mỗi sự kiện hay hành động, dữ liệu liên quan được yêu cầu phải thực hiện yêu cầu. Ví dụ, trong ví dụ code trước đây chúng tôi đã cho thấy một hành động tìm kiếm một bài báo từ một id duy nhất và sau đó biểu diễn bài viết đó. Các tham số ID là nền tảng cho sự thành công của lần hành động cụ thể.
* Trong HTTP thông số có thể được thông qua vào một ứng dụng trong một số cách khác nhau. Đó là:
* Là một phần của chuỗi truy vấn, chẳng hạn như /article?id=123
* Là một phần của nhosm yêu cầu (ví dụ, nếu một mẫu được gửi sử dụng POST)
* Như 1 phần của URI, như /article/ 123
* Sử dụng một phương pháp tiếp cận Servlet truyền thống, bạn sẽ có được các thông số từ các đối tượng HttpRequest, điều đó dễ dàng đủ cho hai ví dụ đầu tiên; tuy nhiên đối với các ví dụ thứ ba, bạn sẽ đặc biệt có cần cấu trúc URI để trích xuất các dữ liệu tham số.
* May mắn thay, Play làm tất cả điều đó cho bạn! Bởi vì các tập tin đường dẫn cho phép chúng tôi để xác định các thông số trong URI, Play xây dựng tất cả các dữ liệu từ các URI, chuỗi truy vấn và nhóm yêu cầu và thêm nó vào một đối tượng đồ gọi là params.
* Có sau đó hai cách bạn có thể làm việc với các dữ liệu từ biến params Map.

#### HTTP to Java Binding

* Như chúng ta đã thấy, Play dễ dàng có khả năng thiết lập thông số HTTP để đơn giản các kiểu dữ liệu của Java. Thật vậy, nó quản lý tất cả các kiểu dữ liệu Java nguyên thủy và tương đương đối tượng của họ (như Long, Boolean, Float vv) một cách dễ dàng. Tuy nhiên, nó cũng quản lý một số đối tượng phức tạp hơn, bao gồm cả:
* Dates
* Files
* Arrays
* Custom objects (POJO)
* JPA

#### Response

* HTTP được xây dựng trên một mô hình yêu cầu/phản hồi, có nghĩa là cho mỗi yêu cầu được nhận (mỗi sự kiện), một phản hồi phải được trả lại. Điều đó nói rằng, phản hồi có thể trống rỗng, nhưng sự kiện này phải được phản hồi trong một số cách để xác nhận yêu cầu thành công.
* Trong Play, một phản hồi sẽ được gửi bằng một trong những phương thức Render hoặc bằng cách chuyển hướng đến một URL được chỉ định.
* Như: renderText(), renderXML(), renderBinary(), renderJSON, render()

#### Controller Annotations

* The Play Controller đi kèm với một số chú thích cho phép các sự kiện hành động của controller bị chặn lại, tương tự như khái niệm về bộ lọc trong API Servlet. Đó là:
* @Before
* @After
* @Finally
* Một đường dẫn cần phải được tĩnh, nhưng không được công khai.

### View

* View là một trong ba thành phần của kiến ​​trúc MVC. Chúng làm việc thông qua các khái niệm của controller, do đó, bước tiếp theo là phải hiểu view hoạt động như thế nào.
* Tất cả các view có thể được tìm thấy trong thư mục app/views trong ứng dụng của bạn. Bạn sẽ thấy các thư mục phụ trong mỗi controller, vì vậy khi bạn lần đầu tiên bắt đầu ứng dụng của bạn giao diện thì các view đầu tiên được tạo ra sẽ được tìm thấy trong app/views/Application/index.html.

#### Mục đích

* Mục đích của lớp View của kiến ​​trúc MVC là để hiển thị cho người dùng trạng thái hiện tại của ứng dụng, bao gồm cả các nội dung của các lớp model, khi controller đã cho phép model để thực hiện các cập nhật cần thiết. Chúng sẽ xem xét các model chi tiết hơn trong phần tiếp theo, nhưng trước tiên chúng ta cần phải hiểu làm thế nào để tạo ra view.
* View có thể được trả lại trong bất kỳ định dạng văn bản chúng ta muốn. Chúng có thể xuất ra HTML, JSON, XML, RSS vv, và tất cả chúng đều được quản lý chính xác cùng một cách trong Play! Hơn nữa Play đi kèm với Groovy như một ngôn ngữ biểu diễn (nghĩ về nó tương tự như là JSP) để hỗ trợ sự phát triển của các quan điểm, trong đó bao gồm một hệ thống khuôn mẫu thông minh và một số tính năng rất gọn gàng.

#### Cú pháp

* Một view là một hỗn hợp của nội dung tĩnh và động. Nội dung tĩnh là những thứ như các thẻ HTML hay XML. Các nội dung động là nội dung mà được tính tại thời gian chạy bằng kỹ thuật Groovy và có thể có những điều như truy cập vào các thông số thông qua phương thức phản hồi ở controller, hoặc Tags, hoặc Comments, hoặc Actions, hoặc Messages, hoặc Scripts và nhiều hơn nữa.

### Model

* Model là phần cuối cùng của kiến ​​trúc MVC, hỗ trợ trong play framework.
* Tất cả các lớp model trong Play nằm trong thư mục app/models trong ứng dụng của bạn.

#### Mục đích

* Là một phần của MVC Model, là cốt lõi của ứng dụng. Nó chứa các dữ liệu tiêu biểu trong ứng dụng của bạn và business logic để thực hiện các chức năng của ứng dụng. Như đã thảo luận, mục đích của view là để biểu diễn các model, và mục đích của controller là đóng vai trò điều khiển, thực hiện các hành động trên mô hình dữ liệu từ người dùng.
* Trong khi framework không dứt khoát ngăn chặn bạn đặt business logic trong các controller và sử dụng các model như là một đối tượng truy cập dữ liệu đơn giản, framework nặng nề khuyên rằng việc model được sử dụng cho cả data và business logic.

1. Tạo một mô hình dữ liệu

* Tạo một mô hình dữ liệu đặc biệt dễ dàng. Một mô hình dữ liệu trong Play chỉ đơn giản là một tập hợp các lớp Java được đặt trong thư mục app/models. Tuy nhiên, có một vài sự khác biệt đáng chú ý là bạn cần phải nhận thức được. Đó là:
* Thuộc tính của lớp mẫu phải được công khai (cũng như không phải là tĩnh và không chính thức)
* Bạn không cần phải tạo ra setters và getters.

#### DB Persistence sử dụng JPA

* Lưu một cơ sở dữ liệu trong Play là một công việc cực kỳ đơn giản. Play giúp việc sử dụng của JPA (Java Persistence API) để lưu dữ liệu trong mô hình dữ liệu của bạn vào cơ sở dữ liệu chỉ với một vài dòng mã. Trong thực tế, như bạn đã mong đợi từ framework, nó thậm chí còn dễ dàng hơn so với sử dụng độc lập JPA.
* Không cần phải lo lắng về việc quản lý thực thể để lưu, xóa, cập nhật hoặc truy vấn cơ sở dữ liệu. Play tóm tắt tất cả các cách cho bạn và giúp bạn có một vài phương pháp đơn giản có sẵn cho các lớp model của bạn và chạy ngay lập tức.
* Các bước để lưu cơ sở dữ liệu sử dụng JPA:
* Thiết lập một cơ sở dữ liệu
* Sử dụng JPA
* Truy cập trực tiếp vào cơ sở dữ liệu
* Play Cache
* Setters and Getters

### Validation

#### Mục đích

* Trong hầu hết các ứng dụng web mà bạn viết, bạn sẽ phải kiểm tra một số mức độ hợp lệ trên các thông số mà controller đề nghị như là một phần của sự kiện. Play có một số cách để thực hiện xác nhận đầu vào người sử dụng và đầu ra để xác nhận lại cho trình duyệt.

#### Cách sử dụng cơ bản

* Cách đơn giản nhất để bắt đầu sử dụng xác nhận hợp lệ trong Play là sử dụng lớp Validation helper. Lớp này là lớp có sẵn thông qua xác nhận biến từ bên trong controller của bạn. Lớp Validation helper chứa một số phương thức giúp thực hiện xác nhận hợp lệ nhanh chóng trên đầu vào của bạn.
* Ví dụ:

public static void register***(*String** user**, String** email**, String** password***) {* // check the user has been supplied validation.required(user);  
  
 // check email address is valid validation.email(email);  
  
 // check user password is between 6 & 10 characters validation.minSize(password, 6);** validation**.**maxSize***(***password**, 10*)*;  
  
 ...** render***()*;  
*}***

#### Sử dụng chú thích

* Thay vì sử dụng các lớp Validation helper, Play cung cấp một cách khác để xác nhận các nội dung của dữ liệu được truyền vào controller, sử dụng các chú thích.
* Đối với mỗi phương thức Validation helper có sẵn, đó cũng là một chú thích tương đương.
* Chú thích đối tượng:
* Các chú thích ở trên là cực kỳ hữu ích cho các hình thức xác nhận các thông số cá nhân, nhưng chúng ta cũng có thể vượt qua các đối tượng là các thông số sử dụng Play. Có thể thêm các chú thích xác nhận đến các lớp mẫu của chúng tôi, để khi đi qua đối tượng như các tham số, chúng ta có thể tiếp tục sử dụng các phương pháp chú thích.
* Để làm điều này, chúng ta chỉ cần sử dụng các chú thích @Valid cho các đối tượng được thông qua như là một tham số, để chỉ ra rằng chúng ta muốn xác nhận các đối tượng, và sau đó sử dụng các chú thích xác nhận rằng chúng tôi đã nhìn thấy để xác nhận các thuộc tính cá nhân của các đối tượng mô hình .
* Controller có thể trông tương tự như sau.

public static void register(@Valid User user){

….

}

* Model

public **class User** ***{*** @Required  
 public String user**;** @Email  
 public String email**;** @MinSize***(*6*)*** @MaxSize***(*10*)*** public String password**;  
 ...**

***}***

### Testing

* Kiểm thử ứng dụng của bạn là một trong những phần quan trọng nhất của việc xây dựng một ứng dụng tuyệt vời.
* Play hỗ trợ dụng cụ để giúp đỡ với việc thử nghiệm các ứng dụng của bạn bằng cách sử dụng hai sản phẩm mã nguồn mở kiểm tra tự động được sử dụng rộng rãi tuyệt vời, JUnit và Selen.

#### Writing Tests:

* Unit Tests, Functional Tests, Acceptance Tests

#### Running Tests:

* Trong khi đi qua ba bài kiểm tra trên bạn nhìn thấy ảnh chụp màn hình của đầu ra của chạy thử nghiệm. Vì vậy, làm thế nào để bạn thực sự chạy thử nghiệm? Vâng, nó thực sự là khá đơn giản.
* Dừng ứng dụng của bạn nếu nó đang chạy, và chạy nó lên một lần nữa bằng cách sử dụng các lệnh kiểm tra Play, chứ không phải lệnh chạy Play

VD: play test helloworld.

* Chạy ứng dụng của bạn như bình thường, nhưng cũng sẽ bao gồm các module Test Runner, và tìm kiếm thư mục /test/ để tải các tests của bạn. Tất cả bạn phải làm bây giờ để chạy thử nghiệm của bạn là để đi đến URL cần thiết. <http://localhost:9000/@tests>

#### Test Database:

* Để kiểm tra hiệu quả ứng dụng của bạn, bạn cần phải kiểm soát các dữ liệu được tổ chức tại các cơ sở dữ liệu để chắc chắn về tình trạng của ứng dụng.
* VD: mở tệp conf/application.conf và điều hướng đến phần dưới cùng của tập tin, bạn sẽ thấy rất gần với đáy một vài dòng tương tự như sau:

%test.application.mode=dev

%test.db=mem

%test.jpa.ddl=create-drop

%test.mail.smtp=mock

#### Automation Test

* Play đi kèm với một lựa chọn khác cho chạy thử nghiệm của bạn. Bằng cách sử dụng các lệnh tự động kiểm tra, chứ không phải là lệnh kiểm tra từ dòng lệnh, Play sẽ tự động khởi động trình duyệt, chạy thử nghiệm và xuất kết quả vào dòng lệnh và cũng để các thư mục kết quả thử nghiệm.
* Các chức năng kiểm tra tự động cũng xuất ra một tập tin đánh dấu, mà là tên là result.passed, hoặc result.failed tùy thuộc vào việc thiết lập các bài kiểm tra thành công hay không.

## Đánh giá – Nhận xét

### So sánh với Spring framework

### Ưu điểm – Nhược điểm

## Hướng dẫn cài đặt - Ứng dụng Helloworld

# Ngôn ngữ Scala

## Giới thiệu

* Scala (Scalable Language - Ngôn ngữ có khả năng mở rộng) là ngôn ngữ lập trình đa mẫu hình, được thiết kế tích hợp các tính năng của lập trình hướng đối tượng và lập trình hàm.
* Scala chạy trên nền máy ảo Java và nó tương thích hoàn toàn với Java. Ngoài ra nó cũng có thể chạy trên .NET, tuy nhiên chưa ổn định.
* Scala có kiểu biên dịch giống Java nên nó có thể đọc được các thư viện của Java.
* Ở Scala, một số cấu trúc phức tạp của môi trường Java hay .NET đã được loại bỏ và thêm vào một số đặc tính ưu việt hơn, ví dụ như (1) a uniform object model, (2) pattern matching and higher-order functions, (3) novel ways to abstract and compose programs.

## Cơ bản về Scala

### Basic Syntax

* Nếu bạn có sự hiểu biết tốt về Java, sẽ rất dễ dàng cho bạn để tìm hiểu Scala. Sự khác biệt lớn nhất giữa cú pháp Scala và Java là: ký tự cuối cùng của dòng là không bắt buộc. Khi chúng ta xem xét một chương trình Scala nó có thể được định nghĩa là một tập hợp các đối tượng giao tiếp qua cách gọi mỗi phương thức khác nhau. Bây giờ chúng ta xem xét bên trong các lớp, đối tượng, phương thức và các biến có ý nghĩa gì.
* Object - Đối tượng có trạng thái và hành vi. Ví dụ: Một con chó có trạng thái - màu sắc, tên, giống cũng như các hành vi - vẫy đuôi, sủa, ăn uống. Một đối tượng là một thể hiện của một lớp.
* Class - Một lớp có thể được định nghĩa như là một mẫu/kế hoạch chi tiết mô tả các hành vi/trạng thái đối tượng mà nó hỗ trợ.
* Phương thức - Một phương thức cơ bản là một hành vi. Một lớp có thể chứa nhiều phương thức. Đó là trong phương thức mà các logic được viết, dữ liệu được thao tác và tất cả các hành động được thực thi.
* Fields - Mỗi đối tượng có một bộ các biến duy nhất, được gọi là các trường. Trạng thái của một đối tượng được tạo ra bởi các giá trị được gán cho các trường này.
* Cú pháp cơ bản:

Về chương trình Scala, nó là rất quan trọng để giữ trong tâm trí các điểm sau đây.

* Scala là trường hợp nhạy cảm, có nghĩa là định danh Hello và hello sẽ có ý nghĩa khác nhau trong Scala.
* Tên lớp - Đối với tất cả các tên lớp, các chữ cái đầu tiên phải viết hoa. Nếu một vài từ được sử dụng để tạo thành một tên của lớp, chữ cái đầu tiên mỗi từ bên trong nên viết hoa.

Ví dụ: MyFirstScala

* Tên phương thức - Tất cả các tên phương thức nên bắt đầu với kí tự thường. Nếu một vài từ được sử dụng để tạo thành tên của phương thức, sau đó chữ cái đầu tiên mỗi từ bên trong nên viết hoa.

Ví dụ def methodName ()

* Tên tập tin chương trình - Tên của tập tin chương trình sẽ so sánh chính xác tên đối tượng. Khi lưu tập tin bạn nên lưu nó bằng cách sử dụng tên đối tượng và thêm '.scala' vào cuối của tên. (Nếu tên tập tin và tên đối tượng không phù hợp với chương trình của bạn sẽ không biên dịch).

Ví dụ: Giả sử 'HelloWorld' là tên đối tượng. Sau đó, các tập tin sẽ được lưu lại dưới dạng 'HelloWorld.scala'

* def main(args: Array [String]) - Scala xử lý chương trình bắt đầu từ phương thức main () là một phần bắt buộc của mỗi chương trình Scala.
* Định danh Scala
* Chữ và số: Nhận diện chữ bắt đầu bằng chữ cái hoặc dấu gạch dưới, mà có thể được theo sau bởi chữ, chữ số, hoặc dấu gạch dưới. Các ký hiệu '$' là một từ khóa dành riêng trong Scala và không nên được sử dụng trong định danh. VD: age, salary, \_value, \_\_1\_value
* Toán tử: Một định danh toán tử bao gồm một hoặc nhiều ký tự toán tử. Ký hiệu toán tử là các ký tự ASCII in như +,:,, ~ hoặc #. VD: + ++ ::: <?>:>
* Hỗn hợp: Một dạng hỗn hợp bao gồm một nhận dạng chữ và số, được theo sau bởi một gạch dưới và một định danh toán tử. VD: unary\_+, myvar\_=
* Scala Keywords:



* Comment trong Scala: Scala hỗ trợ dòng đơn và nhiều dòng comment rất giống với Java. Nhiều dòng comment có thể được lồng vào nhau, nhưng yêu cầu phải được lồng nhau đúng. Tất cả các ký tự có sẵn trong bất kỳ comment sẽ được bỏ qua bởi trình biên dịch Scala.

VD:

**object HelloWorld *{*** */\* This is my first java program.   
 \* This will print 'Hello World' as the output  
 \* This is an example of multi-line comments.  
 \*/* **def main*(***args: **Array*[***String***]) {* // Prints Hello World  
 // This is also an example of single line comment.  
 println*(*"Hello, world!"*)  
 }  
}***

* Các dòng trống và khoảng trắng: Một dòng chỉ chứa khoảng trắng, có thể có một comment, được biết như một dòng trống, và Scala hoàn toàn bỏ qua nó. Dấu hiệu có thể tách bằng ký tự khoảng trắng and /or comment.
* Ký tự newline: Scala là một ngôn ngữ hướng đối tượng, câu lệnh có thể được chấm dứt bằng dấu chấm phẩy (;) hoặc newline. Một dấu chấm phẩy là cần thiết nếu bạn viết nhiều lệnh trên một dòng duy nhất:

**val *s* = "hello"; println *(s)***

* Package:
* Một package là tên một module có mã. Ví dụ, package tiện ích Lift là net.liftweb.util. Khai báo package là dòng đầu tiên không comment trong tập tin nguồn như sau:

**package** com**.**liftcode**.**stuff

* Package Scala có thể được nhập để họ có thể được tham chiếu trong phạm vi biên soạn hiện hành. Các khai báo sau nhập từ các nội dung của package scala.xml:

**import** scala**.**xml**.**\_

* Bạn có thể nhập một lớp đơn và đối tượng, ví dụ, HashMap từ package scala.collection.mutable:

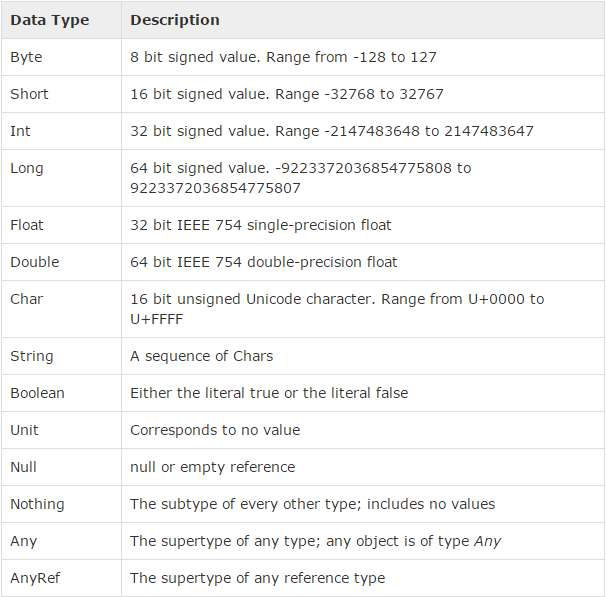
**import** scala**.**collection**.**mutable**.**HashMap

* Bạn có thể nhập nhiều hơn một lớp hoặc đối tượng từ một package đơn, ví dụ, TreeMap và TreeSet từ package scala.collection.immutable:

**import** scala**.**collection**.**immutable**.*{***TreeMap**,** TreeSet***}***

### Data Type

* Scala có tất cả các loại dữ liệu giống như Java, với bộ nhớ tương tự và chính xác. Sau đây là bảng đưa ra chi tiết về tất cả các loại dữ liệu có sẵn trong Scala:



* Tất cả các kiểu dữ liệu liệt kê ở trên là đối tượng. Không có kiểu dữ liệu gốc giống như trong Java. Điều này có nghĩa rằng bạn có thể gọi các phương thức trên một Int, Long, …

### Variable

* Khai báo biến
* Scala có cú pháp khác nhau cho việc khai báo của các biến và chúng có thể được định nghĩa là giá trị, hằng hoặc một biến. Sau đây là cú pháp để định nghĩa một biến sử dụng từ khóa var:

**var *myVar***: String **= "Foo"**

* Ở đây, myVar được khai báo dùng từ khóa var. Điều này có nghĩa rằng nó là một biến có thể thay đổi giá trị và điều này được gọi là biến có thể thay đổi. Sau đây là cú pháp để định nghĩa một biến sử dụng từ khóa val:

**val *Val***: String **= "Foo"**

* Ở đây, myVal được khai báo dùng val từ khóa. Điều này có nghĩa rằng nó là một biến mà không thể thay đổi và điều này được gọi là biến không thay đổi.
* Biến kiểu dữ liệu:
* Loại biến được xác định sau tên biến, và trước dấu bằng. Bạn có thể xác định bất kỳ loại biến Scala bằng cách nhắc đến kiểu dữ liệu của nó như sau:

**val** or **val *VariableName*** : DataType ***[*=** Initial Value***]***

* Nếu bạn không chỉ định bất kỳ giá trị ban đầu cho một biến, thì nó có giá trị như sau:

**var** myVar :**Int;  
val** myVal :String**;**

* Loaị biến suy luận:
* Khi bạn chỉ định một giá trị ban đầu cho một biến, trình biên dịch Scala có thể hình dung ra kiểu của biến dựa trên các giá trị được gán cho nó. Điều này được gọi là kiểu biến suy luận. Vì vậy, bạn có thể viết các khai báo biến như thế này:

**var *myVar* = 10;  
val *myVal* = "Hello, Scala!";**

* Ở đây theo mặc định myVar sẽ là loại biến Int và myVal sẽ trở thành loại biến String.
* Gán biến:
* Scala hỗ trợ gán biến. Nếu một khối mã hoặc phương thức trả về một tuple, các tuple có thể được gán cho một biến val.

**val *(myVar1***: **Int, *myVar2***: String***)* = Pair*(*40, "Foo"*)*val *(myVar1*, *myVar2)* = Pair*(*40, "Foo"*)***

### Access Modifier

* Thành phần private:
* Một thành phần private là có thể nhìn thấy chỉ bên trong lớp hoặc đối tượng có chứa các định nghĩa thành viên. Sau đây là ví dụ:

**class Outer *{* class Inner *{* private def f*() {* println*(*"f"*) }* class InnerMost *{* f*()* // OK  
 *}  
 }  
 (*new Inner*)*.**f***()* // Error: f is not accessible  
*}***

* Trong Scala, truy cập (new Inner).f() là bất hợp pháp vì f được khai báo là private ở Inner và các truy cập không phải là từ bên trong lớp Inner. Ngược lại, việc truy cập f ở lớp InnerMost là OK, vì truy cập được chứa trong thân của lớp Inner. Java sẽ cho phép truy cập cả hai bởi vì nó cho phép lớp bên ngoài truy cập thành viên private của lớp bên trong nó.
* Thành phần protected:
* Một thành phần protected chỉ truy cập từ các lớp con của lớp mà thành phần đó đã được xác định. Sau đây là ví dụ:
* Việc truy cập vào f trong lớp Sub là OK, vì f là protected trong Super và Sub là một lớp con của Super. Ngược lại các truy cập vào f trong Other không được phép, vì Other không kế thừa từ Super. Trong Java, các truy cập đó sẽ được vẫn được phép bởi vì Other nằm trong gói giống như Sub.

**package** p ***{* class Super *{* protected def f*() {* println*(*"f"*) }  
 }* class Sub extends Super *{* f*()  
 }* class Other *{  
 (*new Super*)*.**f***()* // Error: f is not accessible  
 *}  
}***

* Thành phần public:
* Mỗi thành phần không có nhãn private hoặc protected là public. Không có giới hạn rõ ràng cho các thành phần public. Các thành phần này có thể được truy cập từ bất cứ nơi nào. Sau đây là ví dụ:

**class Outer *{* class Inner *{* def f*() {* println*(*"f"*) }* class InnerMost *{* f*()* // OK  
 *}  
 }  
 (*new Inner*)*.f*()* // OK because now f() is public  
*}***

### Operators

* Một toán tử là một biểu tượng cho trình biên dịch để thực hiện các thao tác toán học hợp lý cụ thể. Scala thật là tốt để xây dựng toán tử và cung cấp các loại toán tử sau đây:
* Toán tử toán học:

(Giả sử biến A là 10 và biến B là 20)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Mô tả** | **Ví dụ** |
| **+** | Thêm hai toán hạng | A + B sẽ là 30 |
| **-** | Trừ toán hạng thứ hai từ đầu tiên | A - B sẽ là -10 |
| **\*** | Tích của hai toán hạng | A \* B sẽ là 200 |
| **/** | Thương của 2 toán hạng | B / A sẽ là 2 |

* Toán tử quan hệ:

(Giả sử biến A là 10 và biến B là 20)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Mô tả** | **Ví dụ** |
| == | Kiểm tra nếu các giá trị của hai toán hạng có tương đương hay không, nếu có thì điều kiện trở nên đúng. | (A == B) là đúng. |
| != | Kiểm tra nếu các giá trị của hai toán hạng có tương đương hay không, nếu giá trị không bằng nhau sau đó điều kiện trở nên đúng. | (A != B) là đúng. |
| > | Kiểm tra nếu giá trị của toán hạng bên trái lớn hơn giá trị của toán hạng bên phải, nếu có thì điều kiện trở nên đúng. | (A > B) là không đúng. |
| < | Kiểm tra nếu giá trị của toán hạng bên trái là ít hơn giá trị của toán hạng bên phải, nếu có thì điều kiện trở nên đúng. | (A < B) là đúng. |
| >= | Kiểm tra nếu giá trị của toán hạng bên trái lớn hơn hoặc bằng giá trị của toán hạng bên phải, nếu có thì điều kiện trở nên đúng. | (A >= B) là không đúng. |
| <= | Kiểm tra nếu giá trị của toán hạng bên trái nhỏ hơn hoặc bằng giá trị của toán hạng bên phải, nếu có thì điều kiện trở nên đúng. | (A <= B) là đúng. |

* Toán tử logic:

(Giả sử biến A là 10 và biến B là 20)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Mô tả** | **Ví dụ** |
| && | Được gọi là toán tử logic AND. Nếu cả hai toán hạng không bằng không thì điều kiện trở nên đúng. | (A && B) là sai. |
| || | Được gọi là toán tử logic OR. Nếu bất kỳ hai toán hạng là không bằng không thì điều kiện trở nên đúng. | (A || B) là đúng. |
| ! | Được gọi là toán tử logic NOT. Sử dụng để đảo ngược trạng thái logic của toán hạng của nó. Nếu một điều kiện là đúng thì toán tử logic NOT sẽ làm nó sai. | !(A && B) là đúng. |

* Toán tử bit:

(Giả sử biến A là 60 và biến B là 13)

A = 0011 1100

B = 0000 1101

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Mô tả** | **Ví dụ** |
| & | Toán tử thao tác bit **AND**, khi nào 2 bit đều là 1 thì kết quả trả về mới là 1, các trường hợp còn lại đều là 0. | (A & B) sẽ là 12, kết quả là 0000 1100 |
| | | Toán tử thao tác bit NOT, chỉ cần 1 trong 2 bit là 1 thì kết quả trả về sẽ là 1. | (A | B) sẽ là 61, kết quả là 0011 1101 |
| ^ | Toán tử thao tác bit XOR, nếu 2 bit khác nhau sẽ cho ra kết quả 1 và ngược lại, 2 bit giống nhau sẽ cho ra kết quả 0. | (A ^ B) sẽ là 49, kết quả là 0011 0001 |
| ~ | Phép toán not thay đổi bit 0 thành bit 1 và ngược lại, bit 1 thành bit 0. | (~A ) sẽ là -61, kết quả là 1100 0011. |
| << | Phép dịch bit sang trái, khi dịch dãy bit A sang phải n đơn vị tức là chỉ số của mỗi bit trong A sẽ bị trừ đi n đơn vị. Sau khi dịch bit, các bit có chỉ số âm sẽ bị bỏ đi | A << 2 sẽ là 240, kết quả là 1111 0000 |
| >> | Phép dịch bit sang phải, khi dịch sang trái n đơn vị tức là chỉ số của mỗi bit sẽ được cộng thêm n đơn vị. Sau khi dịch bit, các bit có chỉ số âm sẽ bị bỏ đi | A >> 2 sẽ là 15, kết quả là 1111 |
| >>> | Phép dịch chuyển bit sang phải không dấu. Các giá trị toán hạng bên trái được di chuyển bên phải theo số bit được xác định bởi các toán hạng bên phải và chuyển giá trị này được lấp đầy với số không. | A >>>2 sẽ là 15 kết quả là 0000 1111 |

* Toán tử gán

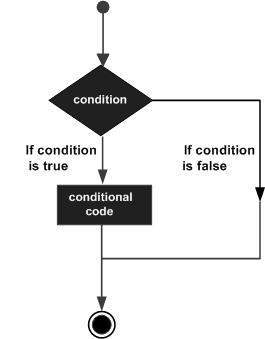
|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Mô tả** | **Ví dụ** |
| = | Toán tử gán đơn giản. Gán giá trị toán hạng bên phải cho toán hạng trái. | C = A + B sẽ gán giá trị của A + B vào cho C |
| += | Thêm giá trị toán hạng phải tới toán hạng trái và gán giá trị đó cho toán hạng trái. | C += A là tương đương với C = C + A |
| -= | Trừ đi giá trị toán hạng phải từ toán hạng trái và gán giá trị này cho toán hạng trái. | C -= A là tương đương với C = C - A |
| \*= | Nhân giá trị toán hạng phải với toán hạng trái và gán giá trị này cho toán hạng trái. | C \*= A là tương đương với C = C \* A |
| /= | Chia toán hạng trái cho toán hạng phải và gán giá trị này cho toán hạng trái. | C /= A là tương đương với C = C / A |
| %= | Lấy phần dư của phép chia toán hạng trái cho toán hạng phải và gán cho toán hạng trái. | C %= A là tương đương với C = C % A |
| <<= | Dịch trái toán hạng trái sang số vị trí là giá trị toán hạng phải. | C <<= 2 là giống như C = C << 2 |
| >>= | Dịch phải toán hạng trái sang số vị trí là giá trị toán hạng phải. | C >>= 2 là giống như C = C >> 2 |
| &= | Phép AND bit | C &= 2 là giống như C = C & 2 |
| ^= | Phép OR loại trừ bit | C ^= 2 là giống như C = C ^ 2 |

* Toán tử mở rộng trong Scala:

|  |  |  |
| --- | --- | --- |
| **Tên loại toán tử** | **Toán tử** | **Tính kết hợp** |
| Postfix | () [] | Từ trái sang phải |
| Unary | ! ~ | Phải sang trái |
| Multiplicative | \* / % | Từ trái sang phải |
| Additive | + - | Từ trái sang phải |
| Dịch bit | >> >>> << | Từ trái sang phải |
| Quan hệ | > >= < <= | Từ trái sang phải |
| Bằng | == != | Từ trái sang phải |
| Thao tác bit AND | & | Từ trái sang phải |
| Thao tác bit XOR | ^ | Từ trái sang phải |
| Thao tác bit OR | | | Từ trái sang phải |
| Logic AND | && | Từ trái sang phải |
| Logic OR | || | Từ trái sang phải |
| Gán | = += -= \*= /= %= >>= <<= &= ^= |= | Phải sang trái |
| Phẩy | , | Từ trái sang phải |

### If-else

* Sau đây là hình thức chung điển hình của cấu trúc IF ... ELSE được tìm thấy trong hầu hết các ngôn ngữ lập trình:



* Cấu trúc mệnh đề if:
* Cú pháp:

**if*(***Boolean\_expression***)  
{* // Thực thi khi biểu thức Boolean đúng  
*}***

Nếu biểu thức so sánh đúng thì khối mã bên trong câu lệnh if sẽ được thực thi. Nếu không, tập lệnh đầu tiên sau khi kết thúc của câu lệnh if (sau khi ngoặc kép đóng) sẽ được thực thi.

* VD:

Kết quả: This is else statement

* Cấu trúc mệnh đề IF…ELSE:

**object Test *{* def main*(***args: **Array*[***String***]) {* var *x* = 10;  
  
 if*( x*** < **20 *){* println*(*"This is if statement"*)*;  
 *}  
 }  
}***

* Cú pháp:

**if *(***Boolean\_expression***) {* // Thực thi khi biểu thức Boolean đúng  
*}*** Else ***{* // Thực thi khi biểu thức Boolean sai  
*}***

* VD:

**object Test *{* def main*(***args: **Array*[***String***]) {* var *x* = 30;  
  
 if*( x*** < **20 *){* println*(*"This is if statement"*)*;  
 *}*else*{* println*(*"This is else statement"*)*;  
 *}  
 }  
}***

Kết quả: This is else statement

* Cấu trúc IF…ELSE IF…ELSE:
* Cú pháp:

**if *(***Boolean\_expression **1*) {* // Thực thi khi biểu thức Boolean 1 đúng  
*}*** Else **if *(*Boolean** Biểu **2*) {* // Thực thi khi biểu thức Boolean 2 đúng  
*}*** Else **if *(***Boolean\_expression **3*) {* // Thực thi khi biểu thức Boolean 3 đúng  
*}*** Else ***{* // Thực thi khi không ai trong số các điều kiện nêu trên đúng sự thật.  
*}***

* VD:

**object Test *{* def main*(***args: **Array*[***String***]) {* var *x* = 30;  
  
 if*( x*** == **10 *){* println*(*"Value of X is 10"*)*;  
 *}*else if*( x*** == **20 *){* println*(*"Value of X is 20"*)*;  
 *}*else if*( x*** == **30 *){* println*(*"Value of X is 30"*)*;  
 *}*else*{* println*(*"This is else statement"*)*;  
 *}  
 }  
}***

Kết quả: Value of X is 30

* Cấu trúc chồng IF…ELSE:
* Cú pháp:

**if *(***Boolean\_expression **1*) {* // Thực thi khi biểu thức Boolean 1 đúng  
 if *(***Boolean\_expression **2*) {* // Thực thi khi biểu thức Boolean 2 đúng  
 *}  
}***

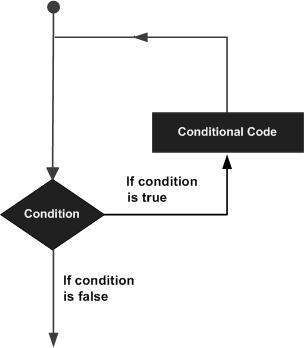
* VD:

**object Test *{* def main*(***args: **Array*[***String***]) {* var *x* = 30;  
 var *y* = 10;  
  
 if*( x*** == **30 *){* if*( y*** == **10 *){* println*(*"X = 30 and Y = 10"*)*;  
 *}  
 }  
 }  
}***

Kết quả: X = 30 and Y = 10

### Loop Type

* Một cấu trúc vòng lặp cho phép chúng thực hiện một lệnh hoặc một nhóm lệnh nhiều lần và sau đây là tổng quát một cấu trúc vòng lặp trong hầu hết các ngôn ngữ lập trình:



* Ngôn ngữ lập trình Scala cung cấp các loại sau đây của các vòng lặp để xử lý yêu cầu lặp.

|  |  |
| --- | --- |
| **Loại vòng lặp** | **Mô tả** |
| [**while loop**](http://www.tutorialspoint.com/scala/scala_while_loop.htm) | Lặp lại một câu lệnh hoặc khối lệnh khi một điều kiện đúng. Nó kiểm tra điều kiện trước khi thực hiện thân vòng lặp. |
| [**do...while loop**](http://www.tutorialspoint.com/scala/scala_do_while_loop.htm) | Giống như một vòng lặp thời gian, ngoại trừ việc nó kiểm tra điều kiện ở cuối thân vòng lặp |
| [**for loop**](http://www.tutorialspoint.com/scala/scala_for_loop.htm) | Thực thi một chuỗi các câu lệnh nhiều lần và rút gọn code quản lý các biến lặp. |

* Kiểm soát vòng lặp: break statement: Kết thúc vòng lặp và chuyển giao thực hiện để báo cáo kết quả sau vòng lặp.
* Vòng lặp không giới hạn: Một vòng lặp trở thành vòng lặp vô hạn nếu một điều kiện không bao giờ sai. Nếu bạn đang sử dụng Scala, vòng lặp while là cách tốt nhất để thực hiện vòng lặp vô hạn như sau:

**object Test *{* def main*(***args: **Array*[***String***]) {* var *a* = 10;  
 // An infinite loop.  
 while*(* true *){* println*(* "Value of a: "** + ***a )*;  
 *}  
 }  
}***

### Function

* Function được hiểu là một khối code thực hiện một logic nào đó và có thể được sử dụng lại.
* Function trong Scala được mô tả bằng hai tính chất:
* Được đặt tên
* Là một hay nhiều biểu thức có thể tái sử dụng
* Function trong Scala mang nhiều đặc điểm của Fuction programming.

#### Cách khai báo Function trong Scala

**Input-less Function (không có input)**

**Syntax: def<**identifier> **= *<expression>***

### Closures

### Strings

### Arrays

### Collections

### Class – Object

### Trait

### Pattern Matching

### Regular Expression

### Exception Handing

### Extractor

### File I/O

## Hướng dẫn cài đặt

### Cài đặt môi trường

### Cài đặt với Eclipse IDE

### Chương trình Helloworld