**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

**KHOA CÔNG NGHỆ PHẦN MỀM**

****

**BÁO CÁO**

**Môn: Đồ án chuyên ngành**

GVHD: Thầy Trần Anh Dũng Nhóm CD:

Nguyễn Văn Cảnh 12520034

Trần Ngọc Dân 12520050

**Mục lục**

[Chương 1. Tìm hiểu Play framework 1](#_Toc448564869)

[I. Giới thiệu 1](#_Toc448564870)

[1. Play framework là gì? 1](#_Toc448564871)

[2. Lịch sử phát triển 1](#_Toc448564872)

[3. Những điểm nổi bật Play Framework 2 so với Play Framework 1 3](#_Toc448564873)

[II. Đặc điểm Play framework 3](#_Toc448564874)

[1. Mô hình MVC 3](#_Toc448564875)

[2. HTTP Routing 4](#_Toc448564876)

[3. Controller 6](#_Toc448564877)

[4. View 9](#_Toc448564878)

[5. Model 10](#_Toc448564879)

[6. Validation 11](#_Toc448564880)

[7. Testing 13](#_Toc448564881)

[III. Đánh giá – Nhận xét 14](#_Toc448564882)

[1. So sánh với Spring framework 14](#_Toc448564883)

[2. Ưu điểm – Nhược điểm 14](#_Toc448564884)

[IV. Hướng dẫn cài đặt - Ứng dụng Helloworld 14](#_Toc448564885)

[1. Những công cụ cần thiết 14](#_Toc448564886)

[2. Cài đặt và kiểm tra (sử dụng gói Play standalone) 15](#_Toc448564887)

[3. Tạo mới chương trình sử dụng Play Framework 17](#_Toc448564888)

[Chương 2. Ngôn ngữ Scala 19](#_Toc448564889)

[I. Giới thiệu 19](#_Toc448564890)

[II. Cơ bản về Scala 19](#_Toc448564891)

[1. Basic Syntax 19](#_Toc448564892)

[2. Data Type 24](#_Toc448564893)

[3. Variable 25](#_Toc448564894)

[4. Access Modifier 26](#_Toc448564895)

[5. Operators 27](#_Toc448564896)

[6. If-else 35](#_Toc448564897)

[7. Loop Type 38](#_Toc448564898)

[8. Function 39](#_Toc448564899)

[9. Closures 42](#_Toc448564900)

[10. Strings 47](#_Toc448564901)

[11. Arrays 47](#_Toc448564902)

[12. Collections 47](#_Toc448564903)

[13. Class – Object 47](#_Toc448564904)

[14. Trait 53](#_Toc448564905)

[15. Pattern Matching 58](#_Toc448564906)

[16. Regular Expression 63](#_Toc448564907)

[17. Exception Handing 63](#_Toc448564908)

[18. Extractor 63](#_Toc448564909)

[19. File I/O 63](#_Toc448564910)

[III. Hướng dẫn cài đặt 63](#_Toc448564911)

[1. Công cụ hỗ trợ 63](#_Toc448564912)

[2. Chương trình Helloworld 63](#_Toc448564913)

# 

# Tìm hiểu Play framework

## Giới thiệu

### Play framework là gì?

* Play là một chương trình khung dùng để tạo các ứng dụng web, được viết bằng Scala và Java theo kiến trúc Model-View-Controller (MVC). Play được tạo ra với mục đích tăng hiệu suất lập trình bằng cách sử dụng quy ước bằng cấu hình, tải lại mã nóng và hiển thị lỗi trên trình duyệt.

### Lịch sử phát triển

#### Play Framework 1



Lược đồ kiến trúc Play Framework 1

* Play framework 1 được tích hợp rất nhiều gói thư viện hỗ trợ, mỗi gói thư việc hỗ trợ một số chức năng nhất định, sau đây là những gói thư viện chính:
* **Netty**: Netty là mộtframework hỗ trợ giao tiếp theo dạng NIO (Non-Blocking I/O) giữa máy chủ và máy trạm.Điều này giúp cho việc giao tiếp giữa máy chủ và máy trám trở nên nhanh chóng và dễ dàng hơn rất nhiều. Nó đơn giản và phù hợp với việc lập trìnhsocket các phương thức như TCP và UDP ở máy chủ. Mặc dù Play được thiết kế để chạy trên máy chủ web JBoss Netty, nhưng nó hoàn có thể đóng gói theo kiểu WAR và sử dụng trong những máy chủ khác bằng Java EE.
* **JPA and Hibernate**: Đây là 2 gói thư viện hỗ trợ quản lý thực thể và truy vấn cơ sở dữ liệu
* **Groovy**: Được sử dụng như mộtcông cụ để tạo ra giao diện web. Mã nguồn giao diện web sử dụng Groovy ở dạng text như html nhưng có thể chèn thêm những dòng lệnh để sinh ra nội dung đọng. Cú pháp của ngôn ngữ Groovy cũng tương tự như Java.

#### Play Framework 2



Lược đồ kiến trúc Play Framework 2

* Play 2 là đánh dấu một bước chuyển biến lớn của Play đối với ngôn ngữ Scala. Trong Play 2 thư viện cho ngôn ngữ đã được sử dụng. Những phiên bản trước đó của Play Framework chủ yếu viết cho ngôn ngữ Java (Scala chỉ được sử dụng như plug-in), nhưng trong Play 2.0 Scala đã trở thành 1 trong 2 ngôn ngữ chính (ngang tầm với Java).
* Trong phiên bản mới của Play Framework đã có nhiều thay đổi nhằm đơn giản và tối ưu hoá cho Play. Sau đây là những thay đổi chính:
* **Scala – library**: Gói thư viện cho ngôn ngữ Scala được chính thức thêm vào.
* **Groovy** đã bị lược đi: Play 2 không còn sử dụng Groovy nữa, thay vào đó là Scala Templates.
* **JPA and Hibernate** cũng bị lược bỏ: Trong phiên bản 2.0 trở đi Play không còn sử dụng hai gói thư viện này để hỗ trợ quản lý thực thể nữa, thay vào đó là một lớp truy xuất dữ liệu mới tên là Anorm. Anorm được sử dụng để truy xuất dữ liệu từ cơ sở dữ liệu sau đó chuyển vào các thực thể 1 cách rất đơn giản.
* **Akka** được sử dụng để làm tác nhân chính để đáp ứng xu hướng mới là phát triển một ứng dụng hiệu năng cao và đáng tin cậy.

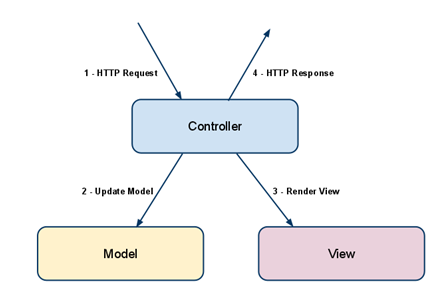
### Những điểm nổi bật Play Framework 2 so với Play Framework 1

* Play Framework 2 là một framework để viết web bằng ngôn ngữ java và scala với tiêu chí nhẹ, không trạng thái, thân thiện với nền tảng web các đặc điểm sau:
* Dựa trên nền tảng java JVM(Java Virtual Machine): Có thể sử dụng đa nền.
* Thân thiện với người phát triển: Bạn chỉ cần chỉnh sửa trong text-editor và sau đó nhấn refresh trên trình duyệt web, mọi chỉnh sửa sẽ có hiệu lực.
* Quy mô đơn giản:Bên dưới nền tảng Play sử dụng một mô hình hoàn toàn bất đồng bộ được xây dựng trên Akka. Kết hợp với đặc điểm không trạng thái của web, quy mô của Play trở nên đơn giản.
* Thích hợp cho cả Web và moblie:Play có thể hoạt động tốt trên cả nền tảng web và mobile.

## 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

### Những công cụ cần thiết

Để có thể sử dụng được Play Framework 2 ta cần có những công cụ sau:

* JDK 6 hoặc mới hơn. Có thể tải tại:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

* Mã nguồn Play Framework 2. Có thể dùng 1 trong 2 bản sau:
* **Typesafe Activator** bao gồm: Activator, Akka, Play, Scala.

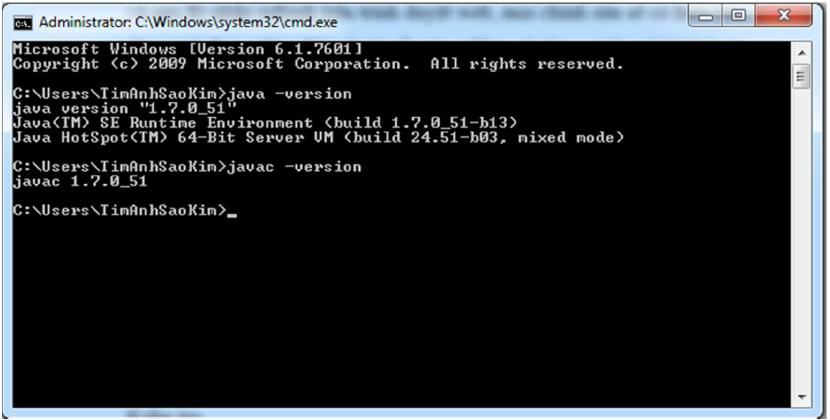
<http://www.lightbend.com/activator/download>

* **Play standalone distribution** chỉ bao gồm Play

<https://playframework.com/download>

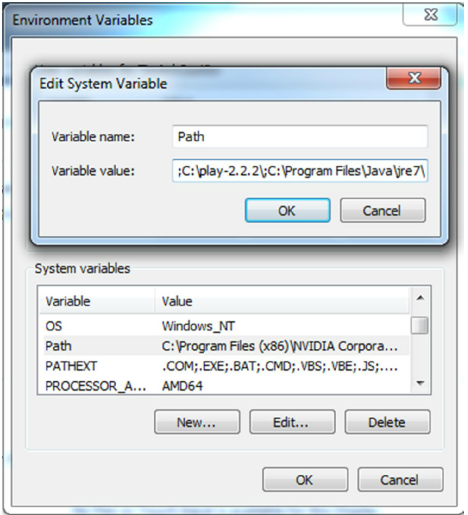
### Cài đặt và kiểm tra (sử dụng gói Play standalone)

* Bước 1: Cần đảm bảo đã cài đặt JDK. Có thể kiểm tra bằng Command Line (CMD) bằng cú pháp **java-version** và **javac-version**



Hình 2.1 Kiểm tra JDK

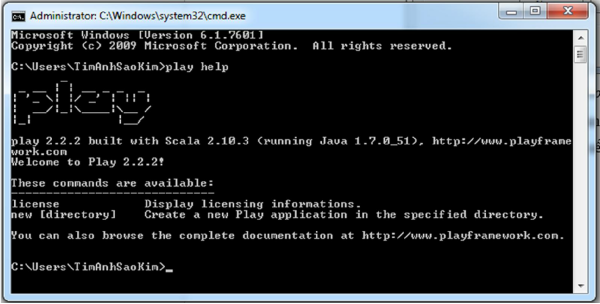
* Bước 2: Giải nén gói **Play stondalone distribution** ở trên và copy vào một nơi bất kỳ (ví dụ tên thư mục là **play-2.2.2** đặt ở đường dẫn C:\)
* Bước 3: Thêm đường dẫn thư mục chứa Play vào biến môi trường (Environment variables).



Hình 2.2 Thêm đường dẫn vào biến môi trường

* Bước 4: Kiểm tra sự tồn tại của Play Framework 2

Trong CMD sử dụng cú pháp **play help** nếu có chứa thông tin của Play Framework 2 xem như đã cài đặt thành công.



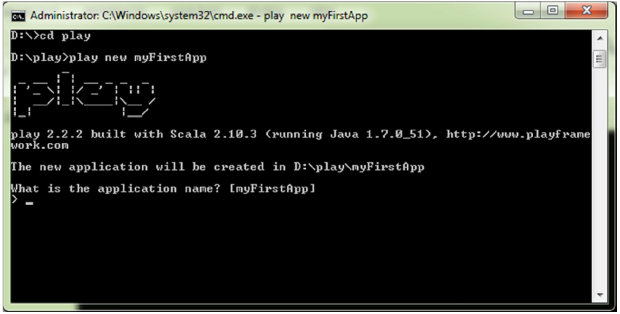
Hình 2.3 Kiểm tra Play Framework 2

### Tạo mới chương trình sử dụng Play Framework

* Để tạo mới chương trình bằng Play ta chọn vị trí cần tạo và dùng lệnh:

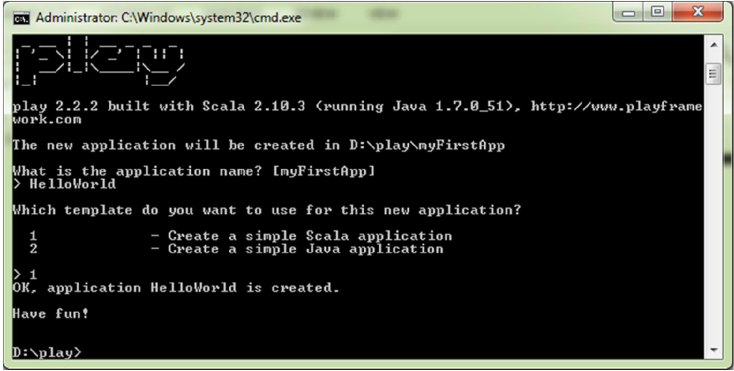
**$ play new myFirstApp**

* Tiếp theo ta nhập tên Project.



Hình 3.4 Tạo project mới trong Play Framework 2

* Sau đó ta chọn Java hoặc Scala tương ứng, chọn số 1 để chọn Scala Application.
* Nhấn Enter và tạo Project Scala trên Play.



Hình 3.5 Chọn ngôn ngữ cho project

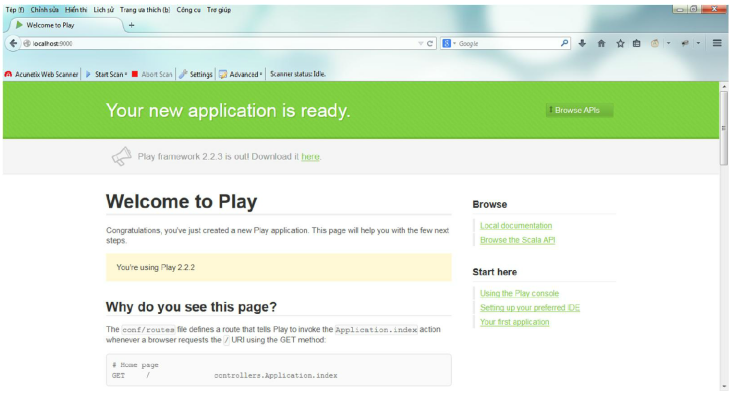
* Để tạo cho eclipse có thể import được ta dùng lệnh:

**$ play eclipse**

* Sau đó chạy project bằng lệnh:

**$ play run**

* Mở trình duyệt web với port 9000 ta sẽ thấy giao diện của Play Framework



Hình 3.6 Chạy project trên trình duyệt

* Import project vừa tạo vào eclipse và bắt đầu tạo web bằng ngôn ngữ Scala.

# 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:

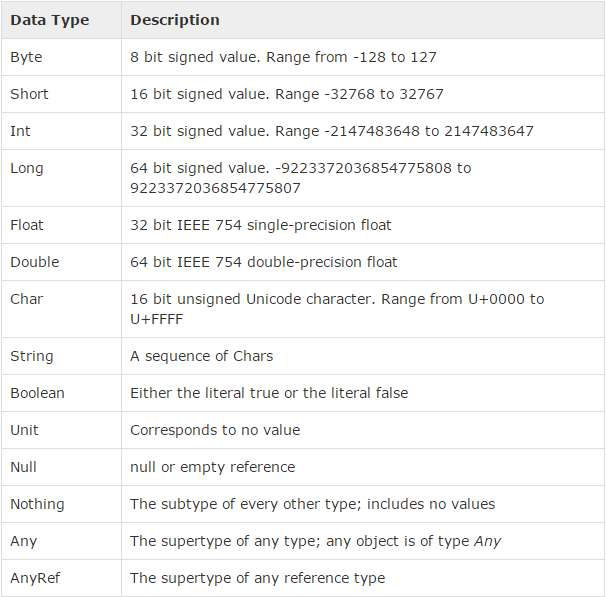
**import** scala**.**collection**.**mutable**.**HashMap

* 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

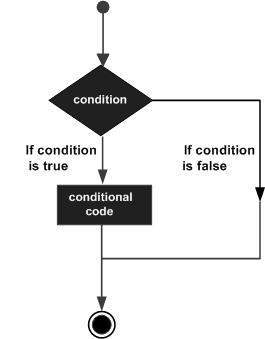
|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Mô tả** | **Ví dụ** |
| = | Toán tử gán đơn giản. Gán giá trị toán hạng bên phải cho toán hạng trái. | C = A + B sẽ gán giá trị của A + B vào cho C |
| += | Thêm giá trị toán hạng phải tới toán hạng trái và gán giá trị đó cho toán hạng trái. | C += A là 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  
 *}  
}***

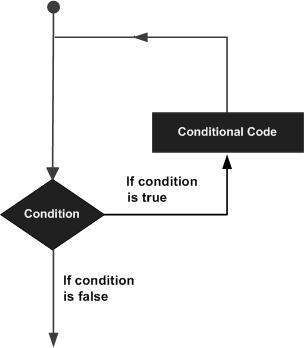
* VD:

**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>*

Vd:

scala> def hello = "Hello"  
hello: String  
scala> hello  
res0: String = Hello

Một cách khác để khai báo một function không có input đó là dùng một cặp dấu ngoặc đơn () – Empty parenthese

scala> def helloWorld() = "Hello World"  
helloWorld: ()String

Điểm khác biệt giữa hai cách khai báo này là:

scala> helloWorld ()  
res1: String = Hello World  
scala> helloWorld  
res2: String = Hello World  
scala> hello()  
<console>:12: error: not enough arguments for method apply: (index: Int)Char in class StringOps.  
Unspecified value parameter index.  
hello()

**Function with a return type (trả về kiểu cố định)**

Như bạn thấy ở trên, nếu bạn không set kiểu trả về cho function, Scala sẽ tự tính toán và set cho bạn, trong thực tế bạn nên tự set kiểu cho function mình viết, điều đấy khiến cho code của bạn rõ ràng hơn và tránh được nhiều sai sót ngoài ý muốn.

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

Vd:

scala> def kieuTraVeLaString : String = "đây là 1 String"  
kieuTraVeLaString: String  
scala> def kieuTraVeLaInt : Int = 100  
kieuTraVeLaInt: Int

**Định nghĩa đầy đủ 1 Function**

Syntax: *def <identifier>(<identifier>: <type>[, ... ]): <type> = <expression>*

Vd:

scala> def cong2So(soThuNhat: Int , soThuHai: Int) : Int = soThuNhat + soThuHai  
cong2So: (soThuNhat: Int, soThuHai: Int)Int  
scala> cong2So(2,3)  
res4: Int = 5

Scala cho phép bạn gọi function với các input với tên được định nghĩa trước đó,ta thử gọi lại function trên với thứ tự input đảo ngược:

scala> cong2So(soThuHai = 3,soThuNhat = 2)  
res5: Int = 5

**Vararg Parameters**

Đôi khi bạn định nghĩa 1 function với với input cùng kiểu và không biết rõ số lượng , Scala đồng ý bạn viết như sau :

scala> def tinhTongNhieuSo(cacSo: Int\*): Int = {  
        | var tong = 0  
        | for (i <- cacSo) tong += i  
        | tong  
        | }  
tinhTongNhieuSo: (cacSo: Int\*)Int  
scala> tinhTongNhieuSo(1,2,3)  
res6: Int = 6  
scala> tinhTongNhieuSo(5,10,15,20)  
res7: Int = 50

**Parameter Groups**

Ngoài ra ta có 1 cách khai báo biến khác trong scala, đó là tách thành nhiều group parameter.

scala> def soNaoLon(soThuNhat: Int)(soThuHai: Int) = if(soThuNhat > soThuHai) soThuNhat

else soThuHai  
soNaoLon: (soThuNhat: Int)(soThuHai: Int)Int  
scala> soNaoLon(3)(1)  
res8: Int = 3  
scala> soNaoLon(3)(5)  
res9: Int = 5

**Function Invocation with Expression Blocks**

Theo cách thông thường ,bạn sẽ chuẩn bị trc input và sau đó gọi function .  
Vd: viết hàm kiểm tra 1 số là chẵn hay lẻ :

scala> def chanHayLe(x : Int) {  
         |    if (x % 2 == 0) println("so chan")  
         |    else println("so le")  
         | }  
chanHayLe: (x: Int)Unit

Giờ tôi sẽ cộng 2 số bất kì và kiểm tra xem  số đó là chẵn hay lẻ:

scala> val tong2SoBatKy = 5 + 9  
tong2SoBatKy: Int = 14  
scala> chanHayLe(tong2SoBatKy)  
so chan

Với Scala mọi thứ đơn giản hơn 1 bước :

scala> chanHayLe { 5 + 9}  
so chan

**Function’s Type Parameters**

Đôi khi bạn cần viết 1 function mà kiểu trả về là động , ví dụ bạn viết 1 function mà kiểu trả về tưởng ứng với kiểu input.

Cách 1: Sử dụng Any vì Any là root của mọi data trong Scala.

scala> def anyThings(x : Any) : Any = x  
anyThings: (x: Any)Any

scala> val soInt: Int = anyThings(1)

<console>:11: error: type mismatch;

 found   : Any

required: Int

val soInt: Int = anyThings(1)

Ở đây anyThings của 1 không trả về 1 số Int vì qua hàm anyThings nó đã biến thành kiểu Any

Cách 2: Scala cung cấp cho bạn 1 giải pháp cho tình huống này:

scala> def anyThings[A](x : A) : A = x

anyThings: [A](x: A)A  
scala> val soInt: Int = anyThings(1)  
soInt: Int = 1

**Nested Functions**

Scala cho phép bạn khai báo function trong function.

scala> def timSoLonTrong3So(soI: Int, soII: Int, soIII: Int) = {  
            |    def timSoLonTrong2So(soI: Int, soII: Int) = if(soI > soII) soI else soII  
            |    timSoLonTrong2So(soI,timSoLonTrong2So(soII,soIII))  
            | }  
timSoLonTrong3So: (soI: Int, soII: Int, soIII: Int)Int  
scala> timSoLonTrong3So(5,15,10)  
res1: Int = 15

**Procedures**

Procedures là function mà không trả về giá trị gì cả hay đúng hơn nó trả về 1 Unit.

scala> def procedure(ten: String) = println("Hello "+ ten)  
procedure: (ten: String)Unit  
scala> procedure("UIT")  
Hello UIT

### Closures

* Một trong những điều cốt lõi của functional programming là function nên là fisrt-class, đơn giản bạn có thể hiểu nó không chỉ đơn giản được định nghĩa và được invoked mà nó được sử dụng như một phần của ngôn ngữ lập trình như là một data type.
* Vậy *first-class function* :
* Có thể được coi như các loại data type khác.
* Có thể được tạo từ một literal.
* Được lưu trữ như một value, một variable hay một data structure.
* Được sử dụng như một parameter cho một function khác – gọi là *higher-order functions.*
  1. ***Kiểu dữ liệu Function***

Kiểu dữ liệu của function được diễn tả bằng một cặp [] bao gồm một nhóm input type và output type được nối với nhau bằng một mũi tên “=>”

Syntax: ([<type>, . . .]) => <type>

* 1. ***Function Literals***

Function literals có đặc điểm:

* + - Syntax: ([<identifier>: <type>, ... ]) => <expression>
    - Không có định danh (Anonymous Function – là khi ta viết một hàm mà hàm đó chỉ sử dụng một lần duy nhất)
    - Vd:

scala> val maximize = (a: Int, b: Int) => if (a > b) a else b  
maximize: (Int, Int) => Int = <function2>

Ở đây (a: Int, b: Int) => if (a > b) a else b là một Literal Function

* 1. ***Higher-Order-Function***

Là một function mà có đối số là một function khác hoặc có kết quả trả về là một function.

Vd:

scala> def dollar2Vnd(number: Int): Int = number \* 21750  
dollar2Vnd: (number: Int)Int

scala> def soDu(soTienDo : Int,f : (Int) => Int){  
             | println(f(soTienDo)+" VND")  
             | }  
soDu: (soTienDo: Int, f: Int => Int)Unit

scala> soDu(5,dollar2Vnd)  
108750 VND

* 1. ***By-Name Parameter***

Là function mà ta khai báo với hệ thống là ta chỉ quan tâm đến output – giá trị trả về của function. Hàm được khai báo với input đầu vào là kiểu dữ liệu Unit hoặc (), kiểu dữ liệu output được khai báo để trả về giá trị của hàm.

Vd:

scala> def nhanDoi (x : => Int) : Int = x \* 2  
nhanDoi: (x: => Int)Int

scala> nhanDoi(4)  
res20: Int = 8

* 1. ***Partial Function***

- Một partial function là một trait mà khi thi hành có thể sử dụng để xây dựng chương trình để giải quyết vấn đề.

- Một trait PartialFunction yêu cầu phương thức “isDefineAt” và “apply” được thực thi.

Vd:

val doubleEvens: PartialFunction[Int, Int] = new PartialFunction[Int, Int] {

//partial function đảm nhận nhiệm vụ

def isDefinedAt(x: Int) = x % 2 == 0

//nếu nhiệm vụ phù hợp thì sẽ thực hiện chức năng này

def apply(v1: Int) = v1 \* 2

}

doubleEvens: PartialFunction[Int,Int] = <function1>

val tripleOdds: PartialFunction[Int, Int] = new PartialFunction[Int, Int] {

def isDefinedAt(x: Int) = x % 2 != 0

def apply(v1: Int) = v1 \* 3

}

tripleOdds: PartialFunction[Int,Int] = <function1>

val whatToDo = doubleEvens orElse tripleOdds

whatToDo: PartialFunction[Int,Int] = <function1>

whatToDo(3)

res3: Int = 9

whatToDo(4)

res4: Int = 8

* + Câu lệnh “case” là một cách nhanh chóng để tạo ra Partial Function. Khi tạo câu lệnh “case” thì “apply” và “isDefineAt” được tạo cho bạn.
  + Vd:

val doubleEvens: PartialFunction[Int, Int] = {

case x if (x % 2) == 0 => x \* 2 //Case statement

}

doubleEvens: PartialFunction[Int,Int] = <function1>

val tripleOdds: PartialFunction[Int, Int] = {

case x if (x % 2) != 0 => x \* 3//Case statement

}

tripleOdds: PartialFunction[Int,Int] = <function1>

val whatToDo = doubleEvens orElse tripleOdds

whatToDo: PartialFunction[Int,Int] = <function1>

* + Từ khóa “andThen” để kết nối các funtion.

val doubleEvens: PartialFunction[Int, Int] = {

case x if (x % 2) == 0 => x \* 2

}

val tripleOdds: PartialFunction[Int, Int] = {

case x if (x % 2) != 0 => x \* 3

}

val printEven: PartialFunction[Int, String] = {

case x if (x % 2) == 0 => "Even"

}

val printOdd: PartialFunction[Int, String] = {

case x if (x % 2) != 0 => "Odd"

}

val whatToDo = doubleEvens orElse tripleOdds andThen (printEven orElse printOdd)

* 1. ***Partially Applied Functions***
  + **Placeholder Syntax**

Placeholder (\_) có thể được dùng vào 2 mục đích sau :

* Thay thể cho những function type được định nghĩa rõ ràng
* Thay thế cho những parameter được sử dụng không quá 1 lần.
* Có lẽ vẫn hơi khó hiểu,ta cùng xem ví dụ :

scala> val nhanDoi: Int => Int = \_ \* 2  
nhanDoi: Int => Int = <function1>

scala> nhanDoi(5)  
res7: Int = 10

* + **Partially Applied Functions**

Một partially Applied function là một hàm không có một vài tham số hoặc không có tham số nào để tạo ra hàm khác.

Ví dụ một partially applied function không có đối số.

def sum(a: Int, b: Int, c: Int) = a + b + c  
//sum: sum[](val a: Int,val b: Int,val c: Int) => Int  
val sum3 = sum \_  
//sum3: (Int, Int, Int) => Int = <function3>  
sum3(1, 9, 7)  
//res8: Int = 17  
sum(4, 5, 6)  
//res8: Int = 17

Ví dụ partial applied function thay thế một số tham số:

def sum(a: Int, b: Int, c: Int) = a + b + c  
//sum: sum[](val a: Int,val b: Int,val c: Int) => Int  
val sumC = sum(1, 10, \_: Int)  
//sumC: Int => Int = <function1>  
sumC(4)  
//res9: Int = 15  
sum(4, 5, 6)  
//res10: Int = 15

* + **Currying Function**

Currying là một kỹ thuật để chuyển đổi chức năng với nhiều tham số vào nhiều chức năng có một tham số.

def multiply(x: Int, y: Int) = x \* y  
//multiply: multiply[](val x: Int,val y: Int) => Int  
(multiply \_).isInstanceOf[Function2[\_, \_, \_]]  
//res10: Boolean = true  
val multiplyCurried = (multiply \_).curried  
//multiplyCurried: Int => (Int => Int) = <function1>  
multiply(4, 5)  
//res11: Int = 20  
multiplyCurried(3)(2)  
//res12: Int = 6  
val multiplyCurriedFour=multiplyCurried(4)  
//multiplyCurriedFour: Int => Int = <function1>  
multiplyCurriedFour(2)  
//res13: Int = 8  
multiplyCurriedFour(4)  
//res13: Int = 16

Currying cho phép bạn tạo ra các phiên bản riêng của chức năng tổng quát:

def customFilter(f: Int => Boolean)(xs: List[Int]) = {  
 xs filter f  
}  
//customFilter: customFilter[](val f: Int => Boolean)(val xs: List[Int]) => List[Int]  
def onlyEven(x: Int) = x % 2 == 0  
onlyEven: onlyEven[](val x: Int) => Boolean  
  
val xs = **List**(12, 11, 5, 20, 3, 13, 2)  
//xs: List[Int] = List(12, 11, 5, 20, 3, 13, 2)  
  
customFilter(onlyEven)(xs)  
//res0: List[Int] = List(12, 20, 2)  
  
val onlyEvenFilter = customFilter(onlyEven) \_  
//onlyEvenFilter: List[Int] => List[Int] = <function1>  
onlyEvenFilter(xs)  
//res1: List[Int] = List(12, 20, 2)

* 1. ***Cloure***

Closure là 1 funtion mà giá trị trả về của nó thay đổi bởi 1 hoặc nhiều variables được định nghĩa ở bên ngoài function.

ví dụ:

đây đơn thuần là 1 function literal:

val multiplier = (i:Int) => i \* 10  
//multiplier: Int => Int = <function1>

Ta thay 10 bằng 1 biến được khai báo trước đó,bên ngoài function :

val multiplier = (i:Int) => i \* factor  
//multiplier: Int => Int = <function1>

giá trị trả về bây giờ phụ thuộc vào giá trị của factor nên nó là 1 Closure.

### Strings

### Arrays

### Collections

### Class – Object

* 1. ***Class***

Class trong Scala có ý nghĩa giống với class trong Java.

Class trong Scala là thành phần static có thể được khởi tạo vào nhiều đối tượng tại runtime.

Vd:

class Point(xc: Int, yc: Int) {  
 var **x**: Int = xc  
 var **y**: Int = yc  
 def move(dx: Int, dy: Int) {  
 **x** = **x** + dx  
 **y** = **y** + dy  
 }  
 override def toString(): String = "(" + **x** + ", " + **y** + ")";  
}

Class “Point” có 2 biến x và y, có 2 phương thức “move” và “toString”

Phương thức “move” có 2 tham số nhưng không trả về giá trị nào (ngầm trả về kiểu Unit). Phương thức “toString” không có tham số, trả về String kế thức phương thức “toString” với từ khóa “override”

Khi khai báo class trong Scala thì bạn có thể viết tất cả parameter của constructor vào câu khai báo và bên trong thì không cần viết lại constructor nữa. Với mỗi class về cơ bản chỉ được có một constructor duy nhất, Mặc dù về mặt cú pháp bạn vẫn có thể định nghĩa nhiều constructor nhưng thực tế sử dụng thì chỉ trong những trường hợp rất đặc biệt. Thêm nữa, constructor đầu tiên có một ý nghĩa đặc biệt hơn các constructor khác (nếu được định nghĩa) và được gọi là **primary constructor**.

Class trong Scala được khởi tạo bởi từ khóa “new”:

object Main {  
 def main(args: Array[String]) {  
 val pt = new Point(1, 2)  
 println(pt)  
 pt.move(10, 10)  
 println(pt)  
 }  
}

Chương trình định nghĩa class “Main” thực thi dưới dạng một đối tượng top-level singleton.

Parameter được định nghĩa với từ khóa val hoặc var như cách 2 là công khai với phạm vi ngoài class, và tất nhiên cả với các method bên trong class nữa.

Vd:

class Saiyan(val name: String, val age: Int) {  
 def +(p: Saiyan): Saiyan = {  
 new Saiyan(name + p.name, age + p.age)  
 }  
}  
val goku = new Saiyan("Goku", 100)  
val vegeta = new Saiyan("Vegeta",100)  
val vegeto = goku + vegeta  
vegeto.name  
// res0: String = GokuVegeta  
vegeto.age  
// res1: Int = 200

**Property**

Property là một phần không thể thiếu của class. Property có thể là private hoặc protected. Nếu không viết gì đằng trước thì property là public.  
Trước property trong class có thể dùng từ khóa val cho trường hợp không thể thay đổi giá trị, và var cho trường hợp có thể thay đổi giá trị.

**Method**

Method là những function thuộc về class ta đang xét. Mỗi loại dữ liệu định nghĩa như một class và có nhiều function đi kèm class đó.

Một số cách định nghĩa method:

**class Person**(**val** name: String, **val** age: Int) {  
 **def** +(p: **Person**): **Person** = **new** Person(name + p.name, age + p.age)  
}  
  
**class Person**(**val** name: String, **val** age: Int) {  
 **def** +(p: **Person**): **Person** = {  
 **new** Person(name + p.name, age + p.age)  
 }  
}

* + Trước mỗi method có thể được thêm private, protected, private[this], protected[package\_name]
* private: method chỉ có thể truy cập từ bên trong class.
* protected: method chỉ có thể truy cập từ bên trong class và các class con kế thừa nó.
* private[this]: method chỉ có thể truy cập từ bên trong object (cái này sẽ nói sau trong phần object).
* protected[package\_name]: method có thể truy cập từ bên trong package có tên là package\_name.
* Method không có gì ở đầu thì mặc định là public, tức là có thể truy cập ở bất cứ đâu.

**Curry method**

Method trong Scala có thể có nhiều dãy parameter, mỗi dãy được viết trong ngoặc đơn (). Method kiểu này gọi là **curry method**, là một kỹ thuật thông dụng trong các ngôn ngữ lập trình hàm và cần thiết cho phần **implicit parameter**. Method “union” dưới đây là một curry method.

**class Saiyan**(**val** *name*: *String*, **val** *age*: Int) {  
 **def** +(*p*: **Saiyan**): **Saiyan** = {  
 **new** Saiyan(*name* + *p*.*name*, *age* + *p*.*age*)  
 }  
}

//defined class Saiyan

**val** goku = **new** Saiyan(**"Goku"**, **100**)  
//goku: Saiyan = Saiyan@559507b5  
**val** vegeta = **new** Saiyan(**"Vegeta"**,**100**)  
//vegeta: Saiyan = Saiyan@3a75489c  
  
**class Fusion** {  
 **def** union(*left*: **Saiyan**)(*right*: **Saiyan**): **Saiyan** = *left* + *right*}  
//defined class Fusion  
  
**var** magic = **new** Fusion()  
// magic: Fusion = Fusion@447a020  
**val** vegeto = magic.union(goku)(vegeta)  
// magic: Fusion = Fusion@447a020  
magic.union(goku) \_  
// res3: Saiyan => Saiyan = <function1>

Ở dòng cuối cùng, khi chỉ gọi một dãy parameter thì curry method sẽ trả về một function.

**Kế thừa**

Tính chất quan trọng nhất của class là tính kế thừa. Kế thừa trong Java có 2 loại: kế thừa class cha (từ khóa extends) và kế thừa interface (từ khóa implements). Mỗi class chỉ có thể kế thừa duy nhất một class cha duy nhất.

Trong Scala tính kế thừa đã được phát triển thêm. Một class có thể kế thừa nhiều trait khác nhau. Trait sẽ được nói đến trong một phần khác. Ở đây chúng ta sẽ chỉ nói đến kế thừa class thông dụng. Vd:

**class Saiyan**(**val** *strength*: Int) {  
 **def** recover(): Unit = {  
 **println**(*strength*\***10**)  
 }  
}  
//defined class Saiyan  
  
**val** kakarot = **new** Saiyan(**10**)  
//kakarot: Saiyan = Saiyan@2706f37b  
  
kakarot.recover()  
// 100  
//res0: Unit = ()  
  
**class SuperSaiyan**(*strength*: Int) **extends** Saiyan(*strength*) {  
 **override def** recover(): Unit = {  
 **println**(*strength*\***100**)  
 }  
}  
//defined class SuperSaiyan  
  
**val** goku = **new** SuperSaiyan(**10**)  
//goku: SuperSaiyan = SuperSaiyan@5860c0fa  
  
goku.recover()  
// 1000  
//res1: Unit = ()

* 1. ***Object***

Trong Scala, tất cả mọi giá trị đều là một object, tất cả mọi method đều trực thuộc một object nào đó.   
Khi trong Java bạn thường thấy các property và method được gán nhãn static trong một class thì trong Scala việc đó trở nên không cần thiết. Trong Scala chúng ta có thể dùng từ khóa object để định nghĩa một khoảng không gian với một tên gọi nhất định giống như là một "singleton object". Tất cả những thứ nằm trong phần không gian nói trên cũng là một object, vì vậy hoàn toàn có thể định nghĩa method và property ở trong này.

Một Object được thể hiện bằng từ khóa “object” có đặc điểm sau:

* Có thể đặt các utility method hoặc global property trong này (giống khoảng bên trong static ở Java).
* Có thể đặt các factory method cho object.
* Sử dụng như là Singleton Pattern (Cái này là một design pattern cơ bản nhé).

Cách khai báo object:

**object obj\_name extends** class\_name **with** trait\_name1 **with** trait\_name2 {

}

Scala có một object tên là **Predef** được định nghĩa là import sẵn. Ví dụ hàm “println()” thực ra là một method của object **Predef**.

Sau đây mình sẽ đưa ra một ví dụ tạo object trùng tên với một class:

**class Saiyan**(**val** name: String, **val** age: Int)  
  
**object Saiyan** {  
 **def** apply(name: String, age: Int): **Saiyan** = **new** Saiyan(name, age)  
}

apply là một hàm đặc biệt. Sau khi định nghĩa như trên thì 2 dòng dưới đây là như nhau:

Saiyan.apply("goku", 100)  
Saiyan("goku",100)

So với cách gọi trực tiếp new Saiyan("goku", 100) thì viết thông qua object thế này có 2 cái lợi:

* + - * Ẩn được nội dung cụ thể của class Saiyan
      * Có thể trả về một instance của một subclass của Saiyan.

Cách viết như trên còn có thể viết gọn nữa như sau:

**case class Saiyan**(name: String, age: Int)  
*//defined class Saiyan*

**Case Class**

Case class là một đối tượng có hai cách dùng. Cách dùng thứ nhất mình sẽ trình bày sau đây và cách dùng thứ hai là trong **Pattern Matching** mà chúng ta sẽ gặp trong một phần khác.

***Case class*** mang bên trong mình một class cùng tên với các constructor parameter *đều là công khai (public)*. Ngoài ra mang thêm *những method cơ bản của một object*.

Thế nào là *những method cơ bản của một object* ?

* equals()
* hashCode()
* toString()

**case class Saiyan**(name: String, age: Int)  
*//defined class Saiyan  
Saiyan*("goku",100).equals(*Saiyan*("goku",100)) *// true  
//res0: Boolean = true*

**Companion Object**

Object trùng tên với class như bên trên được gọi là **Companion Object**. Companion Object khác biệt với Object bình thường ở chỗ: có thể truy cập những property hay method là *private.*

**class Saiyan**(**val** name: String, **private val** age: Int)  
*//defined class Saiyan***object Saiyan** {*//defined module Saiyan* **def** apply(name: String, age: Int): **Saiyan** = {  
 **val** s = **new** Saiyan(name, age)  
 *println*(s.age) *// OK* **return** s  
 }  
}

Nếu object mang một cái tên khác, nó sẽ không còn được coi là companion object và đoạn code trên sẽ ko chạy.

Đến đây có thể bạn sẽ nghĩ, nếu đến *private* của class mà Companion Object còn đọc được thì có gì mà Companion Object không đọc được nữa đâu ?

Câu trả lời là *private[this]*. Nếu gắn *private[this]* vào trước một *property/method* thì nó sẽ chỉ được nhìn thấy duy nhất bên trong class đó.

### Trait

Trait trong Scala có những điểm khác biệt chủ yếu với class như sau:

+ Một Trait hoặc một class có thể kế thừa nhiều trait khác nhau (tính năng này gọi là **Mixin** hoặc **Multiple inheritance**)

+ Không thể khởi tạo trực tiếp instance từ Trait.

+ Không thể nhận parameter (constructor parameter) như class.

**Multible Inheritance**

Vd:

**trait** Saiyan  
**trait** Namek  
**class Human  
class** God{}  
  
*// OK***class SuperHero extends** Human **with** Saiyan **with** Namek  
  
*// Compile error***class SemiGod extends** Human **with** God

một class không thể kế thừa 2 class khác cùng một lúc. Tuy nhiên với trait thì không có giới hạn.

**Không thể tạo instance trực tiếp**

**trait** Saiyan  
**object** {  
 **val** bardock = **new** Saiyan  
 *// Compile error  
 // trait Saiyan is abstract; cannot be instantiated*}

Khi gặp trường hợp thế này thì bạn có 2 lựa chọn:

* Tạo một class kế thừa trait nói trên và tạo instance của class đó
* Tạo instance thông qua việc thêm implement

**trait** Saiyan  
**class SaiyanSoldier extends** Saiyan  
  
**object SaiyanArmy** {  
 **val** *bardock* = **new** SaiyanSoldier *// OK* **val** *kingVegeta* = **new** Saiyan {} *// OK*}

**Không thể nhận constructor như một class**

Vì không thể tạo instance nên Trait cũng không có khả năng nhận constructor parameter như class.

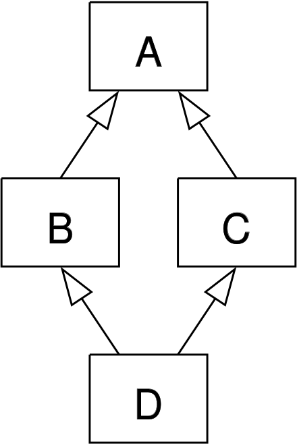
**trait** Saiyan(name: String) {}  
*// Compile error  
// traits or objects may not have parameters*

Để giải quyết tình huống này cũng có 2 cách tương tự như trên:

* Tạo class kế thừa và gọi new với class đó
* truyền thẳng paramter vào implement của trait

**trait** Saiyan {  
 **val** name: String  
}  
**class SaiyanSoldier**(**val** name: String) **extends** Saiyan  
  
**object SaiyanArmy** {  
 **val** *bardock* = **new** SaiyanSoldier("bardock")  
 **val** *kingVegeta* = **new** Saiyan { **val** *name* = "kingVegeta" }  
}

**Diamond Problem**

**Diamond Problem** là một danh từ khá nổi tiếng chỉ về một vấn đề trong cấu trúc kế thừa của lập trình hướng đối tượng. Với lý thuyết về Trait bên trên, bạn thử tưởng tượng nếu 2 trait cùng kế thừa một trait khởi điểm và override cùng một method, vậy khi class mới kế thừa 2 trait nói trên thì chuyện gì sẽ xảy ra ?

**trait** A {  
 **def** greet(): Unit  
}  
  
**trait** B **extends** A {  
 **def** greet(): Unit = *println*("Hello!")  
}  
  
**trait** C **extends** A {  
 **def** greet(): Unit = *println*("Hi!")  
}  
  
**class D extends** B **with** C  
*// Copile error  
/\*\*  
error: class D inherits conflicting members:  
 method greet in trait B of type ()Unit and  
 method greet in trait C of type ()Unit  
(Note: this can be resolved by declaring an override in class D.)  
 class D extends B with C  
 \*/*

Cách giải quyết y như thông báo lỗi đã gợi ý, chúng ta sẽ override method trong class D. Khi overide bạn có thể gọi lại method từ trait nào đã kế thừa tùy thích hoặc thậm chí gọi cả 2 luôn cũng được.

**class D extends** B **with** C {  
 **override def** greet(): Unit = {  
 **super**[B].greet()  
 **super**[C].greet()  
 }  
}

**Stackable Trait**

Giờ mình sẽ nhắc đến một trường hợp đặc biệt hơn. Nếu định nghĩa hàm greet trong cả trait B và C đều có từ khóa override thì **thứ tự kế thừa sau cùng** trong câu khai báo class sẽ quyết định hàm nào được sử dụng. Điều này gọi là **Linearization**. Các trait ở trường hợp này gọi là **Stackable Trait**.

**trait** A {  
 **def** greet(): Unit = *print*(" Borned! ")  
}  
  
**trait** B **extends** A {  
 **super**.greet()  
 **override def** greet(): Unit = *println*("Hello!")  
}  
  
**trait** C **extends** A {  
 **super**.greet()  
 **override def** greet(): Unit = *println*("Hi!")  
}  
  
**class D extends** B **with** C  
**class E extends** C **with** B  
(**new** D).greet()  
*// Borned!  
// Hello  
// Hi*(**new** E).greet()  
*// Borned!  
// Hi  
// Hello*

Ở trường hợp trên, nếu method greet ở trait A không được định nghĩ thì bạn không thể gọi super.greet(). Khi này định nghĩa cho greet sẽ cần thêm từ khóa abstract.

**trait** A {  
 **def** greet(): Unit  
}  
  
**trait** B **extends** A {  
 **abstract override def** greet(): Unit = {  
 **super**.greet()  
 **override def** greet(): Unit = *println*("Hello!")  
 }  
}

Và với kiểu khai báo trên thì bạn không thể tạo ra một class kế thừa trực tiếp trait B được. Lý do là, method greet chưa được định nghĩa? Muốn thoát khỏi tình huống này thì cần phải implement method greet thông qua một trait khác và kế thừa trait đó

**trait** C **extends** A {  
 **def** greet(): Unit = *println*("Hello!")  
}  
  
*// Compile error  
// class D needs to be a mixin, since method greet in trait B of type ()Unit is marked `abstract' and `override', but no concrete implementation could be found in a base***class D extends** B  
  
*// Ok***class ClassB extends** C **with** B

**Seft Types**

**Self types** cho phép định nghĩa trước một trait trừu tượng và thêm implement về sau. **Self types** thường được dùng khi muốn ứng dụng **Dependency Injection** hay gọi tắt là **DI**. Mỗi khi nào bạn nhìn thấy kiểu self types thế này thì có thể chuẩn bị tinh thần rằng "chuẩn bị có DI xảy ra đây".

**Vd:**

**trait** Saiyan {  
 **def** kamehameha(): Unit  
}  
*//defined trait Saiyan***trait** SuperSaiyan {  
 self: Saiyan =>  
  
 **def** damage(): Unit = kamehameha()  
}  
*//defined trait SuperSaiyan***trait** SaiyanSoldier **extends** Saiyan {  
 **def** kamehameha(): Unit = *println*("KaaaaMeeeeHaaaMeeeeHaaaa!")  
}  
*//defined trait SaiyanSoldier***val** vegeta = **new** SuperSaiyan **with** SaiyanSoldier  
*//vegeta: SuperSaiyan with SaiyanSoldier = $anon$1@6587414c*vegeta.damage()  
*//KaaaaMeeeeHaaaMeeeeHaaaa!  
//res0: Unit = ()*

Khi định nghĩa SuperSaiyan với từ khóa self: Saiyan như trên, trait SuperSaiyan có thể sử dụng method kamehameha trong khi bản thân không extends trực tiếp *Saiyan*. Khi tạo một trait SaiyanSoldier để implement thì có thể tạo được instance của SuperSaiyan.

Tạo sao lại phải tạo trait để implement thì lý do khá giống với phần **abstract override** bên trên, Scala không cho phép tạo instance với trait có method chưa được implement.

Vậy **self types** khác với extends trực tiếp thế nào? SuperSaiyanGod dưới đây có hành vi gì khác với SuperSaiyan?

**trait** Saiyan {  
 **def** kamehameha(): Unit  
}  
*//defined trait Saiyan***trait** SuperSaiyanGod **extends** Saiyan {  
 **def** damage(): Unit = kamehameha()  
}  
*//defined trait SuperSaiyanGod*

Câu trả lời là khi khởi tạo instance và chỉ rõ kiểu *SuperSaiyan* hay *SuperSaiyanGod* thì cách nhìn của instance đó với method gốc *kamehameha*() sẽ khác nhau.

**val** vegeta: SuperSaiyan = **new** SuperSaiyan **with** SaiyanSoldier  
vegeta.kamehameha()

*// error: value kamehameha is not a member of SuperSaiyan*

**val** goku: SuperSaiyanGod = **new** SuperSaiyanGod **with** SaiyanSoldier  
goku.kamehameha()

*// KaaaaMeeeeHaaaMeeeeHaaaa*

### Pattern Matching

Case class và matching là hai thành phần rất hữu dụng trong scala. Ở phần này chúng ta sẽ tìm hiểu về chúng.

**Case class**

Ví dụ khai báo các case class:

**abstract class Expr***//defined class Expr***case class Var**(name: String) **extends** Expr  
*//defined class Var***case class Number**(num: Double) **extends** Expr  
*//defined class Number***case class UnOp**(operator: String, arg: **Expr**) **extends** Expr  
*//defined class UnOp***case class BinOp**(operator: String, left: **Expr**, right: **Expr**) **extends** Expr  
*//defined class UnOp*

Việc khai báo case class rất đơn giản. Nó giống như khai báo class nhưng thêm tiền tố **case** vào đầu. Việc khởi tạo case class cũng không phải khởi tạo với **new** mà có thể dẽ dàng thực hiện như sau:

**val** v = Var("x")

*//v: Var = Var(x)***val** op = BinOp("+", Number(1), v)

*//op: BinOp = BinOp(+,Number(1.0),Var(x))*

Các parameter trong case class còn mặc định được gán là val.

**Pattern Matching**

Xét hàmsimplifyTop:

**def** simplifyTop(expr: **Expr**): **Expr** = expr **match** {  
 **case** *UnOp*("-", *UnOp*("-", e)) => e *// Double negation* **case** *BinOp*("+", e, *Number*(0)) => e *// Adding zero* **case** *BinOp*("\*", e, *Number*(1)) => e *// Multiplying by one* **case** \_ => expr  
}  
*//simplifyTop: simplifyTop[](val expr: Expr) => Expr*

simplifyTop(*UnOp*("-", *UnOp*("-", *Var*("x"))))  
*//res0: Expr = Var(x)*

Như vậy **match** trong scala giống với **switch** trong java. Việc sử dụng case \_ => expr giống với default trong java.

**Kinds Pattern**

Để sử dụng **match** một cách tốt nhất ta nên hiểu các pattern của nó. Dưới đây là các pattern quan trọng:

* **Wildcard patterns (mẫu ký tự đại diện)**

Trong scala wildcard pattern được sử dụng là (\_) nó đại diện cho tất cả các object. Nó được sử dụng như default trong java. Ta xem ví dụ sau:

expr **match** {  
 **case** *BinOp*(op, left, right) =>  
 *println*(expr +"is a binary operation")  
 **case** \_ =>  
}

Ở đây ta thấy rằng trong trường hợp expr không phải là kiểu BinOp thì sẽ không sử lý gì cả. Nếu trong trường hợp bạn không cần đến dữ liệu op, left, right trong BinOp(op, left, right) thì chúng ta cũng có thể viêt bằng cách khác như sau:

expr **match** {  
 **case** *BinOp*(\_, \_, \_) => *println*(expr +"is a binary operation")  
 **case** \_ => *println*("It's something else")  
}

* **Constant Pattern**

Sử dụng Constant patterns khi muốn chỉ rõ một giá trị nào đó ví dụ như 5, true, "hello"…

**def** describe(x: **Any**) = x **match** {  
 **case** 5 => "five"  
 **case true** => "truth"  
 **case** "hello" => "hi!"  
 **case** *Nil* => "the empty list"  
 **case** \_ => "something else"  
}

* **Variable Pattern**

Một Variable patterns sẽ nhận 1 giá trị bất kỳ giống như wildcard. Nhưng khác biệt ở đây là ta có thể sử dụng giá trị đó.

expr **match** {  
 **case** 0 => "zero"  
 **case** somethingElse => "not zero: "+ somethingElse  
}

Khi ta sử dụng đồng thời cả Variable patterns và Wildcard patterns thì việc gì sẽ sảy ra?

**import Math**.{*E*, *PI*}  
  
**val** pi = **Math**.*PI //pi: Double = 3.141592653589793  
  
E* **match** {  
 **case** pi => "strange math? E = "+ pi  
}  
*//res9: java.lang.String = strange math? Pi = 2.7182818...*

pi ở trong case pi là một Variable patterns vì vậy nó sẽ nhận giá trị = E = 2.7182818...

Nếu bạn thực hiện code như sau:

*E* **match** {  
 **case** pi => "strange math? Pi = "+ pi  
 **case** \_ => "OK"  
}  
<**console**>:10: error: unreachable code  
 case \_ => "OK"

Có lỗi sảy ra vì pi ở đây là 1 Variable patterns nó sẽ nhận tất cả các giá trị khi đó case \_ sẽ trở nên vô nghĩa.

vậy làm sao để để so sánh E với pi (Constant patterns) được khai báo ở val pi = Math.Pi? Rất đơn giản ta chỉ cần thêm ` vào trước và sau pi.

*E* **match** {  
 **case** `pi` => "strange math? Pi = "+ pi  
 **case** \_ => "OK"  
}  
*//res13: java.lang.String = OK*

* **Constructor pattern**

Constructor thực sự làm cho pattern matching chở nên mạnh mẽ. Một constructor pattern trông giống như là "BinOp("+", e, Number(0))". Nó bao gồm kiểu là "BinOp" và các patterns nhận giá trị là "+", e, Number(0)

expr **match** {  
 **case** BinOp("+", e, Number(0)) => *println*("a deep match")  
 **case** \_ =>  
}

* **Sequence pattern**

Chúng ta xem ví dụ với việc match 1 list có đội dài là 3 và có phần tử đầu tiên là 0

expr **match** {  
 **case** List(0, \_, \_) => *println*("found it")  
 **case** \_ =>  
}

Khi ban muốn match với 1 list không giới hạn độ dài ban có thể sử dụng (\_\*)

expr **match** {  
 **case** List(0, \_\*) => *println*("found it")  
 **case** \_ =>  
}

* **Tuple pattern**

Cũng như List hoặc Array việc match tuple khá đơn giản.

**def** tupleDemo(expr: **Any**) =  
 expr **match** {  
 **case** (a, b, c) => *println*("matched "+ a + b + c)  
 **case** \_ =>  
 }

* **Typed pattern**

Chúng ta cũng có thể match theo type.

**def** generalSize(x: **Any**) = x **match** {  
 **case** s: String => s.length  
 **case** m: Map[\_, \_] => m.size  
 **case** \_ => -1  
}  
generalSize("abc")  
*//res14: Int = 3*generalSize(*Map*(1 -> 'a', 2 -> 'b'))  
*//res15: Int = 2*generalSize(**Math**.Pi)  
*//res16: Int = -1*

**Variable Binding**

Khi muốn tạo rằng buộc khi match ta sủ dung thêm ký tự @ để tạo rằng buộc.

expr **match** {  
 **case** UnOp("abs", e @ UnOp("abs", \_)) => e  
 **case** \_ =>  
}

Ở đây bắt buộc e phải có kiểu là UnOp và parameter đầu tiên phải là "abs" Khi muốn tạo thêm rằng buộc ta có thể sử dụng toán tử if ngay trong câu lệnh case.

**def** simplifyAdd(e: Expr) = e **match** {  
 **case** BinOp("+", x, y) **if** x == y => BinOp("\*", x, Number(2))  
 **case** \_ => e  
}

Ở ví dụ trên ta thêm rằng buộc chỉ chấp nhận BinOp có x = y

* **Sealed Classes**

Khi bạn thực hiện match một đối tượng case class thuộc 1 lớp nào đó mà không thực hiện hết việc match bao phủ hết các trường hợp của nó thì scala sẽ đưa ra thông báo cho chúng ta.

**sealed abstract class Expr  
case class Var**(name: String) **extends** Expr  
**case class Number**(num: Double) **extends** Expr  
**case class UnOp**(operator: String, arg: **Expr**) **extends** Expr  
**case class BinOp**(operator: String,  
 left: **Expr**, right: **Expr**) **extends** Expr  
  
**def** describe(e: **Expr**): String = e **match** {  
 **case** *Number*(\_) => "a number"  
 **case** *Var*(\_) => "a variable"  
}

Khi compile đoạn code trên scala sẽ đưa ra cho ta thông báo

warning: **match** is not exhaustive!  
 missing combination UnOp  
missing combination BinOp

Để loại bỏ vấn đề này ta có 2 cách. Cách thứ 1 là bắt hết các case

**def** describe(e: Expr): String = e **match** {  
 **case** Number(\_) => "a number"  
 **case** Var(\_) => "a variable"  
 **case** \_ => **throw new** RuntimeException *// Should not happen*}

Cách thứ 2 là tắt chế độ check của scala

**def** describe(e: Expr): String = (e: *@unchecked*) **match** {  
 **case** Number(\_) => "a number"  
 **case** Var(\_) => "a variable"  
}

### Regular Expression

### Exception Handing

### Extractor

### File I/O

## Hướng dẫn cài đặt

### Công cụ hỗ trợ

* Để có thể viết và biên dịch được chương trình được viết bằng ngôn ngữ Scala cần có những công cụ sau:
* Java Developer Kit (JDK) phiên bản 5 trở lên. Có thể tải ở đường dẫn sau:

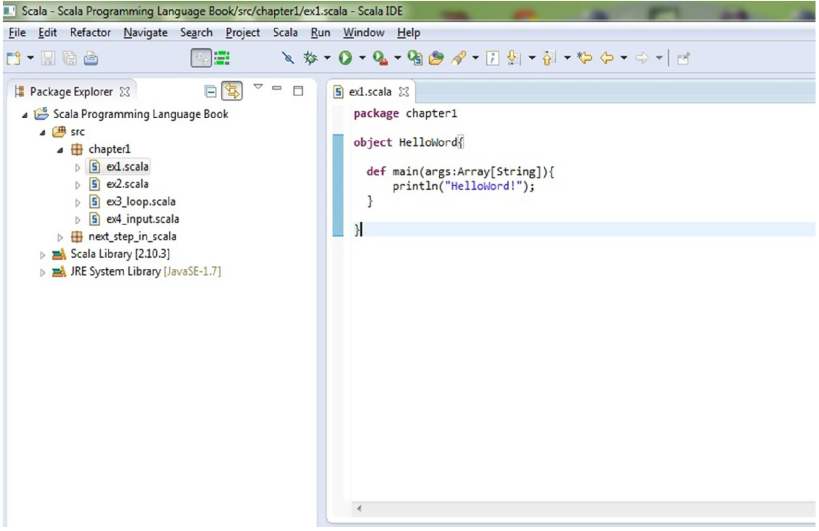
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

* Scala SDK và Bundle of the Scala IDE for Eclipse. Có thể tải ở đường dẫn sau:

<http://scala-ide.org/download/sdk.html>

### Chương trình Helloworld

* Sau khi khởi động **Bundle of the Scala IDE for Eclipse**, tương tự như Java, tạo project ta chọn: File -> New -> Scala Project.



Hình Tạo mới project bằng **Bundle of the Scala IDE for Eclipse**

* Tạo ứng dụng “Hello world”:

object HelloWord{

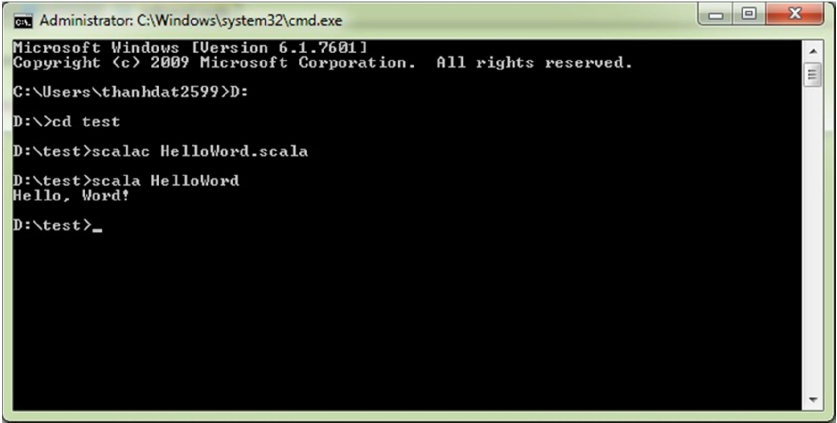
def main(args: Array[String]){

println(“Hello, World!”)

}

}

* Biên dịch và thực thi Scala source bằng CMD:
* Chương trình tên ta lưu thành file HelloWorld.scala, giả sử đường dẫn tại D:\Test
* Đầu tiên ta gọi lệnh : >D:
* Tiếp theo trỏ đến thư mục chứa file HelloWorld.scala bằng lệnh: >cd Test
* Để biên dịch ta gọi lệnh: >scalac HelloWorld.scala
* Nếu không có lỗi sẽ không có thông báo gì, tiếp theo để thực thi chương trình ta dùng lệnh: >scala HelloWorld
* Chương trình sẽ in ra kết quả: Hello, World!



Hình biên dịch và thực thi bằng CMD