eBook Gratuit

APPRENEZ

Scala Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#scala

Table des matières

À propos 1

Chapitre 1: Démarrer avec le langage Scala 2

Remarques 2

Versions 2

Examples 3

Bonjour tout le monde en définissant une méthode 'principale' 3

Bonjour tout le monde en étendant l'application 4

Initialisation différée 4

Initialisation différée 4

Bonjour le monde en script 5

Utiliser la Scala REPL 5

Scala Quicksheet 6

Chapitre 2: Annotations 8

Syntaxe 8

Paramètres 8

Remarques 8

Examples 8

Utiliser une annotation 8

Annoter le constructeur principal 8

Création de vos propres annotations 9

Chapitre 3: Bibliothèque de continuation 11

Introduction 11

Syntaxe 11

Remarques 11

Examples 11

Les rappels sont des continuations 11

Créer des fonctions qui prennent des continuités 12

Chapitre 4: Classe d'option 14

Syntaxe 14

Examples 14

Options en tant que collections 14
Utilisation de l'option au lieu de null 14
Les bases 15
Exemple avec carte 16
Options pour la compréhension 16
Chapitre 5: Classes de cas 18
Syntaxe 18
Examples 18
Égalité de classe de cas 18
Artefacts de code générés 18
Les bases de la classe de cas 20
Classes de cas et immunité 20
Créer une copie d'un objet avec certaines modifications 21
Classes de cas à élément unique pour la sécurité de type 21
Chapitre 6: Classes de types 23
Remarques 23
Examples 23
Classe de type simple 23
Extension d'une classe de type 24
Ajouter des fonctions de classe de type aux types 25
Chapitre 7: Classes et Objets 27
Syntaxe 27
Examples 27
Instancier des instances de classe 27
Classe d'instanciation sans paramètre: {} vs () 28
Singleton & Companion Objects 29
Objets singleton 29
Objets compagnons 29
Objets 30
Vérification du type d'instance 30
Constructeurs 32
Constructeur principal 32

Constructeurs Auxiliaires 33
Chapitre 8: Collections 34
Examples 34
Trier une liste 34
Créer une liste contenant n copies de x 35
Cheatsheet Liste et Vecteur 35
Carte cheatsheet 36
Carte et filtre sur une collection 37
Carte 37
Multiplication des nombres entiers par deux 37
Filtre 37
Vérification des numéros de paire 38
Plus d'exemples de carte et de filtre 38
Introduction aux collections de la Scala 38
Types transversables 39
Plier 40
Pour chaque 41
Réduire 41
Chapitre 9: Collections parallèles 43
Remarques 43
Examples 43
Créer et utiliser des collections parallèles 43
Pièges 43
Chapitre 10: Combinateurs d'analyseur 45
Remarques 45
Examples 45
Exemple de base 45
Chapitre 11: Correspondance de motif 46
Syntaxe 46
Paramètres 46
Examples 46

Match simple 46
Correspondance de modèle avec un identifiant stable 47
Correspondance de motif sur une Seq 48
Gardes (si expressions) 49
Correspondance de motifs avec les classes de cas 49
Correspondance sur une option 50
Traitements scellés assortis 50
Correspondance de motifs avec Regex 51
Reliure de motif (@) 51
Types de correspondance 52
Correspondance de modèle compilée en tant que commutateur de table ou commutateur de reche 53
Faire correspondre plusieurs modèles à la fois 53
Correspondance des motifs sur les tuples 54
Chapitre 12: Currying 56
Syntaxe 56
Examples 56
Un multiplicateur configurable en tant que fonction curry 56
Plusieurs groupes de paramètres de différents types, paramètres de curry des positions arb 56
Currying une fonction avec un seul groupe de paramètres 56
Currying 57
Currying 57
Quand utiliser le curry 58
Une utilisation réelle du curry 59
Chapitre 13: Énumérations 61
Remarques 61
Examples 61
Jours de la semaine en utilisant l'énumération Scala 61
Utiliser des objets et des traits scellés 62
Utilisation d'objets trait et de cas scellés et allValues-macro 63
Chapitre 14: Expressions régulières 65
Syntaxe 65
Examples 65

Déclaration d'expressions régulières ... 65
Répétition de la correspondance d'un motif dans une chaîne ... 66
Chapitre 15: Extracteurs ... 67
Syntaxe ... 67
Examples ... 67
Extracteurs de tuple ... 67
Extracteurs de classe de cas ... 68
Unapply - Extracteurs personnalisés ... 68
Notation Infix Extractor ... 69
Extracteurs de regex ... 70
Extracteurs transformateurs ... 70
Chapitre 16: Fonction d'ordre supérieur ... 72
Remarques ... 72
Examples ... 72
Utiliser des méthodes comme valeurs de fonction ... 72
Fonctions d'ordre élevé (fonction comme paramètre) ... 73
Arguments évaluation paresseuse ... 73
Chapitre 17: Fonctions définies par l'utilisateur pour la ruche ... 75
Examples ... 75
Un simple UDF Hive au sein d'Apache Spark ... 75
Chapitre 18: Fonctions partielles ... 76
Examples ... 76
Composition ... 76
Utilisation avec `collect` ... 76
Syntaxe de base ... 77
Utilisation comme fonction totale ... 78
Utilisation pour extraire des tuples dans une fonction de carte ... 78
Chapitre 19: Futures ... 80
Examples ... 80
Créer un avenir ... 80
Consommer un avenir prospère ... 80
Consommer un avenir défaillant ... 80

Assembler l'avenir ... 81
Séquençage et traversée des Futures ... 81
Combiner plusieurs contrats à terme - Pour la compréhension ... 82
Chapitre 20: Implicits ... 84
Syntaxe ... 84
Remarques ... 84
Examples ... 84
Conversion implicite ... 84
Paramètres implicites ... 85
Classes implicites ... 86
Résolution des paramètres implicites Utilisation de 'implicitement' ... 87
Implique dans le REPL ... 87
Chapitre 21: Injection de dépendance ... 89
Examples ... 89
Modèle de gâteau avec la classe d'implémentation interne ... 89
Chapitre 22: Interopérabilité Java ... 90
Examples ... 90
Conversion de collections Scala en collections Java et vice versa ... 90
Tableaux ... 90
Conversions de type Scala et Java ... 91
Interfaces fonctionnelles pour les fonctions Scala - scala-java8-compat ... 92
Chapitre 23: Interpolation de chaîne ... 94
Remarques ... 94
Examples ... 94
Bonjour l'interpolation de chaîne ... 94
Interpolation de chaînes formatées à l'aide de l'interpolateur f ... 94
Utiliser l'expression dans les littéraux de chaîne ... 94
Interpolateurs de chaînes personnalisés ... 95
Interpolateurs de chaînes comme extracteurs ... 96
Interpolation de chaîne brute ... 96
Chapitre 24: Invocation dynamique ... 98
Introduction ... 98

Syntaxe 98
Remarques 98
Examples 98
Accès aux champs 98
Appels de méthode 99
Interaction entre l'accès aux champs et la méthode de mise à jour 99
Chapitre 25: JSON 101
Examples 101
JSON avec spray-json 101
Rendez la bibliothèque disponible avec SBT 101
Importer la bibliothèque 101
Lire JSON 101
Ecrire JSON 101
DSL 101
Lecture-écriture aux classes de cas 102
Format personnalisé 102
JSON avec Circé 103
JSON avec play-json 103
JSON avec json4s 106
Chapitre 26: La gestion des erreurs 109
Examples 109
Essayer 109
Non plus 109
Option 110
Correspondance de motif 110
Utiliser la carte et getOrElse 110
En utilisant fold 110
Conversion en Java 110
Gestion des erreurs provenant des futures 111
Utilisation de clauses try-catch 111
Convertir des exceptions en deux types d'option 112

Chapitre 27: Les fonctions 113
Remarques 113
Différence entre les fonctions et les méthodes: 113
Examples 113
Fonctions anonymes 113
Raccourci sténographie 114
Fonctions anonymes sans paramètres 114
Composition 114
Relation avec les fonctions partielles 115
Chapitre 28: Les meilleures pratiques 116
Remarques 116
Examples 116
Rester simple 116
Ne pas emballer trop dans une expression. 116
Préférer un style fonctionnel, raisonnablement 117
Chapitre 29: Macros 118
Introduction 118
Syntaxe 118
Remarques 118
Examples 118
Annotation Macro 118
Méthode Macros 119
Erreurs dans les macros 120
Chapitre 30: Manipulation XML 122
Examples 122
Beautify ou Pretty-Print XML 122
Chapitre 31: Mise en place de Scala 123
Examples 123
Sur Linux via dpkg 123
Installation d'Ubuntu via le téléchargement manuel et la configuration 123
Mac OSX via Macports 124

Chapitre 32: Monades 125
Examples 125
Monad Définition 125
Chapitre 33: Opérateurs à Scala 127
Examples 127
Opérateurs intégrés 127
Surcharge de l'opérateur 127
Priorité de l'opérateur 128
Chapitre 34: Paquets 130
Introduction 130
Examples 130
Structure du colis 130
Paquets et fichiers 130
Convection de nomenclature 131
Chapitre 35: Paramétrage de type (génériques) 132
Examples 132
Le type d'option 132
Méthodes paramétrées 132
Collection générique 133
Définir la liste des Ints 133
Définition de la liste générique 133
Chapitre 36: Pendant que les boucles 134
Syntaxe 134
Paramètres 134
Remarques 134
Examples 134
Pendant que les boucles 134
Boucles Do-While 134
Chapitre 37: Portée 136
Introduction 136
Syntaxe 136

Examples ... 136
Portée publique (par défaut) ... 136
Un champ privé ... 136
Une portée spécifique au package privé ... 137
Portée privée de l'objet ... 137
Portée protégée ... 137
Portée protégée du package ... 137
Chapitre 38: Pour les expressions ... 139
Syntaxe ... 139
Paramètres ... 139
Examples ... 139
Basic For Loop ... 139
Basic For Comprehension ... 139
Imbriqué pour boucle ... 140
Monadic pour les compréhensions ... 140
Itérer à travers des collections en utilisant une boucle For ... 141
Desugaring For Comprehensions ... 141
Chapitre 39: Programmation au niveau du type ... 143
Examples ... 143
Introduction à la programmation au niveau du type ... 143
Chapitre 40: Quasiquotes ... 145
Examples ... 145
Créer un arbre de syntaxe avec des quasi-notes ... 145
Chapitre 41: Récursivité ... 146
Examples ... 146
Récursion de la queue ... 146
Récursion régulière ... 146
Récursion de la queue ... 146
Récursivité sans empilement avec trampoline (scala.util.control.TailCalls) ... 147
Chapitre 42: Réflexion ... 149
Examples ... 149
Charger une classe en utilisant la réflexion ... 149

Chapitre 43: Ruisseaux 150
Remarques 150
Examples 150
Utiliser un flux pour générer une séquence aléatoire 150
Flux infinis via la récursivité 150
Flux infini d'auto-référence 151
Chapitre 44: Scala.js 152
Introduction 152
Examples 152
console.log dans Scala.js 152
Fonctions de flèche grasse 152
Classe simple 152
Collections 152
Manipuler DOM 152
Utiliser avec SBT 153
Sbt dépendance 153
Fonctionnement 153
En cours d'exécution avec une compilation continue: 153
Compiler en un seul fichier JavaScript: 153
Chapitre 45: Scaladoc 154
Syntaxe 154
Paramètres 154
Examples 155
Scaladoc simple à la méthode 155
Chapitre 46: scalaz 156
Introduction 156
Examples 156
ApplyUsage 156
FunctorUsage 156
ArrowUsage 157
Chapitre 47: Si des expressions 158
Examples 158

Expressions de base si ... 158
Chapitre 48: Surcharge de l'opérateur ... 159
Examples ... 159
Définition d'opérateurs d'infix personnalisés ... 159
Définition d'opérateurs unitaires personnalisés ... 159
Chapitre 49: Symboles littéraux ... 161
Remarques ... 161
Examples ... 161
Remplacement des chaînes dans les clauses cas ... 161
Chapitre 50: synchronisé ... 163
Syntaxe ... 163
Examples ... 163
synchroniser sur un objet ... 163
synchroniser implicitement sur ce ... 163
Chapitre 51: Test avec ScalaTest ... 164
Examples ... 164
Bonjour test de spécification mondiale ... 164
Spec Cheatsheet de test ... 164
Inclure la bibliothèque ScalaTest avec SBT ... 165
Chapitre 52: Tests avec ScalaCheck ... 166
Introduction ... 166
Examples ... 166
Scalacheck avec scalatest et messages d'erreur ... 166
Chapitre 53: Traits ... 169
Syntaxe ... 169
Examples ... 169
Modification empilable avec des traits ... 169
Les bases du trait ... 170
Résoudre le problème du diamant ... 170
Linéarisation ... 172
Chapitre 54: Travailler avec des données dans un style immuable ... 174

Remarques ... 174
Les noms de valeur et de variable doivent être en cas de chameau inférieur ... 174
Examples ... 174
Ce n'est pas juste val vs var ... 174
val et var ... 174
Collections immuables et Mutable ... 175
Mais je ne peux pas utiliser l'immuabilité dans ce cas! ... 175
"Pourquoi devons-nous muter?" ... 176
Créer et remplir la carte de result ... 176
Implémentation Mutable ... 176
Se plier à la rescousse ... 176
Résultat intermédiaire ... 177
Facilité de raisonnement ... 177
Chapitre 55: Travailler avec Gradle ... 178
Examples ... 178
Configuration de base ... 178
Créez votre propre plugin Gradle Scala ... 178
Ecrire le plugin ... 179
Utiliser le plugin ... 182
Chapitre 56: Tuples ... 184
Remarques ... 184
Examples ... 184
Créer un nouveau tuple ... 184
Tuples dans les collections ... 184
Chapitre 57: Type Inférence ... 186
Examples ... 186
Inférence de type local ... 186
Type Inférence Et Génériques ... 186
Limites à l'inférence ... 186
Prévenir l'inférence Rien ... 187
Chapitre 58: Type Variance ... 189
Examples ... 189

Covariance ... 189
Invariance ... 189
Contravariance ... 190
Covariance d'une collection ... 191
Covariance sur un trait invariant ... 191
Chapitre 59: Types de méthode abstraits uniques (types SAM) ... 193
Remarques ... 193
Examples ... 193
Syntaxe Lambda ... 193
Chapitre 60: Types de soi ... 194
Syntaxe ... 194
Remarques ... 194
Examples ... 194
Exemple simple de type auto ... 194
Chapitre 61: Unités de manutention (mesures) ... 195
Syntaxe ... 195
Remarques ... 195
Examples ... 195
Alias de type ... 195
Classes de valeur ... 195
Chapitre 62: Var, Val et Def ... 197
Remarques ... 197
Examples ... 197
Var, Val et Def ... 197
var ... 197
val ... 198
déf ... 198
Les fonctions ... 199
Lazy val ... 199
Quand utiliser 'paresseux' ... 200
Surcharge surcharge ... 201

Paramètres nommés....201
Crédits....203

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: scala-language

It is an unofficial and free Scala Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Scala Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec le langage Scala

Remarques

Scala est un langage de programmation multi-paradigme moderne conçu pour exprimer des modèles de programmation communs de manière concise, élégante et sûre. Il intègre en douceur les fonctionnalités des langages orientés objet et fonctionnels .

La plupart des exemples donnés nécessitent une installation Scala fonctionnelle. Ceci est la page d'installation de Scala , et c'est l'exemple «Comment configurer Scala» . scalafiddle.net est une bonne ressource pour l'exécution de petits exemples de code sur le Web.

Versions

Version	Date de sortie
2.10.1	2013-03-13
2.10.2	2013-06-06
2.10.3	2013-10-01
2.10.4	2014-03-24
2.10.5	2015-03-05
2.10.6	2015-09-18
2.11.0	2014-04-21
2,11.1	2014-05-21
2.11.2	2014-07-24
2.11.4	2014-10-30
2.11.5	2014-01-14
2.11.6	2015-03-05
2.11.7	2015-06-23
2.11.8	2016-03-08
2.11.11	2017-04-19
2.12.0	2016-11-03

Version	Date de sortie
2.12.1	2016-12-06
2.12.2	2017-04-19

Examples

Bonjour tout le monde en définissant une méthode 'principale'

Placez ce code dans un fichier nommé `HelloWorld.scala` :

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
  }
}
```

Démo en direct

Pour le compiler en bytecode qui est exécutable par la JVM:

```
$ scalac HelloWorld.scala
```

Pour l'exécuter:

```
$ scala Hello
```

Lorsque le runtime Scala charge le programme, il recherche un objet nommé `Hello` avec une méthode `main` . La méthode `main` est le point d'entrée du programme et est exécutée.

Notez que, contrairement à Java, Scala n'a pas besoin de nommer des objets ou des classes après le fichier dans lequel ils se trouvent. Au lieu de cela, le paramètre `Hello` transmis dans la commande `scala Hello` fait référence à l'objet à rechercher contenant la méthode `main` à exécuter. Il est parfaitement possible d'avoir plusieurs objets avec des méthodes principales dans le même fichier `.scala` .

Le tableau `args` contiendra les arguments de ligne de commande donnés au programme, le cas échéant. Par exemple, nous pouvons modifier le programme comme ceci:

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
    for {
      arg <- args
    } println(s"Arg=$arg")
  }
}
```

Compilez-le:

```
$ scalac HelloWorld.scala
```

Et ensuite l'exécuter:

```
$ scala HelloWorld 1 2 3
Hello World!
Arg=1
Arg=2
Arg=3
```

Bonjour tout le monde en étendant l'application

```
object HelloWorld extends App {
  println("Hello, world!")
}
```

Démo en direct

En étendant le trait `App` , vous pouvez éviter de définir une méthode `main` explicite. Le corps entier de l'objet `HelloWorld` est traité comme "la méthode principale".

2.11.0

> ## Initialisation différée
>
> Selon la documentation officielle , `App` utilise une fonctionnalité appelée *Initialisation différée* . Cela signifie que les champs d'objet sont initialisés *après* l'appel de la méthode principale.

2.11.0

> ## Initialisation différée
>
> Selon la documentation officielle , `App` utilise une fonctionnalité appelée *Initialisation différée* . Cela signifie que les champs d'objet sont initialisés *après* l'appel de la méthode principale.
>
> `DelayedInit` est maintenant **obsolète** pour un usage général, mais est *toujours pris en charge* pour `App` tant que cas particulier. Le support continuera jusqu'à ce qu'une fonctionnalité de remplacement soit décidée et implémentée.

Pour accéder aux arguments de ligne de commande lors de l'extension de l' `App` , utilisez `this.args` :

```
object HelloWorld extends App {
  println("Hello World!")
  for {
    arg <- this.args
  } println(s"Arg=$arg")
```

```
}
```

Lorsque vous utilisez `App` , le corps de l'objet sera exécuté en tant que méthode `main` , il n'est pas nécessaire de remplacer le `main` .

Bonjour le monde en script

Scala peut être utilisé comme langage de script. Pour illustrer, créez `HelloWorld.scala` avec le contenu suivant:

```
println("Hello")
```

Exécutez-le avec l'interpréteur de ligne de commande (le `$` est l'invite de la ligne de commande):

```
$ scala HelloWorld.scala
Hello
```

Si vous omettez `.scala` (comme si vous aviez simplement tapé `scala HelloWorld`), le coureur recherchera un fichier `.class` compilé avec bytecode au lieu de compiler, puis d'exécuter le script.

> **Remarque:** Si scala est utilisé comme langage de script, aucun package ne peut être défini.

Dans les systèmes d'exploitation utilisant des terminaux shell `bash` ou similaires, les scripts Scala peuvent être exécutés à l'aide d'un «préambule de shell». Créez un fichier nommé `HelloWorld.sh` et placez-le comme contenu:

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("Hello")
```

Les parties entre `#!` et `!#` est le préambule du shell et est interprété comme un script bash. Le reste est Scala.

Une fois que vous avez enregistré le fichier ci-dessus, vous devez lui accorder des autorisations «exécutables». Dans le shell, vous pouvez faire ceci:

```
$ chmod a+x HelloWorld.sh
```

(Notez que cela donne la permission à tout le monde: lisez à propos de chmod pour savoir comment le définir pour des ensembles d'utilisateurs plus spécifiques.)

Maintenant, vous pouvez exécuter le script comme ceci:

```
$ ./HelloWorld.sh
```

Utiliser la Scala REPL

Lorsque vous exécutez `scala` dans un terminal sans paramètres supplémentaires, il ouvre un interpréteur REPL (Read-Eval-Print Loop):

```
nford:~ $ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala>
```

Le REPL vous permet d'exécuter Scala selon une feuille de calcul: le contexte d'exécution est préservé et vous pouvez essayer manuellement les commandes sans avoir à créer un programme complet. Par exemple, en tapant `val poem = "As halcyons we shall be"` cela ressemblerait à ceci:

```
scala> val poem = "As halcyons we shall be"
poem: String = As halcyons we shall be
```

Maintenant, nous pouvons imprimer notre `val` :

```
scala> print(poem)
As halcyons we shall be
```

Notez que `val` est immuable et ne peut être écrasé:

```
scala> poem = "Brooding on the open sea"
<console>:12: error: reassignment to val
       poem = "Brooding on the open sea"
```

Mais dans le REPL, vous *pouvez* redéfinir un `val` (ce qui provoquerait une erreur dans un programme Scala normal, si cela était fait dans le même domaine):

```
scala> val poem = "Brooding on the open sea"
poem: String = Brooding on the open sea
```

Pour le reste de votre session REPL, cette variable nouvellement définie suivra la variable précédemment définie. Les REPL sont utiles pour voir rapidement le fonctionnement des objets ou du code. Toutes les fonctionnalités de Scala sont disponibles: vous pouvez définir des fonctions, des classes, des méthodes, etc.

Scala Quicksheet

La description	Code
Attribuer une valeur int immuable	`val x = 3`
Attribuer une valeur intable mutable	`var x = 3`
Attribuer une valeur immuable avec un type explicite	`val x: Int = 27`
Attribuer une valeur évaluée paresseuse	`lazy val y = print("Sleeping in.")`

La description	Code
Lier une fonction à un nom	`val f = (x: Int) => x * x`
Lier une fonction à un nom de type explicite	`val f: Int => Int = (x: Int) => x * x`
Définir une méthode	`def f(x: Int) = x * x`
Définir une méthode avec typage explicite	`def f(x: Int): Int = x * x`
Définir une classe	`class Hopper(someParam: Int) { ... }`
Définir un objet	`object Hopper(someParam: Int) { ... }`
Définir un trait	`trait Grace { ... }`
Obtenir le premier élément de la séquence	`Seq(1,2,3).head`
Si interrupteur	`val result = if(x > 0) "Positive!"`
Obtenez tous les éléments de la séquence sauf le premier	`Seq(1,2,3).tail`
Boucle à travers une liste	`for { x <- Seq(1,2,3) } print(x)`

Boucle imbriquée	`for {`
	`x <- Seq(1,2,3)`
	`y <- Seq(4,5,6)`
	`} print(x + ":" + y)`

Pour chaque fonction d'exécution de l'élément de liste	`List(1,2,3).foreach { println }`
Imprimer en standard	`print("Ada Lovelace")`
Trier une liste de manière alphanumérique	`List('b','c','a').sorted`

Lire Démarrer avec le langage Scala en ligne: https://riptutorial.com/fr/scala/topic/216/demarrer-avec-le-langage-scala

Chapitre 2: Annotations

Syntaxe

- @AnAnnotation def someMethod = {...}
- @AnAnnotation class someClass {...}
- @AnnotatioWithArgs (annotation_args) def someMethod = {...}

Paramètres

Paramètre	Détails
@	Indique que le jeton suivant est une annotation.
SomeAnnotation	Le nom de l'annotation
constructor_args	(facultatif) Les arguments transmis à l'annotation. Si aucun, les parenthèses ne sont pas nécessaires.

Remarques

Scala-lang fournit une liste d'annotations standard et leurs équivalents Java .

Examples

Utiliser une annotation

Cet exemple d'annotation indique que la méthode suivante est `deprecated` .

```
@deprecated
def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Cela peut aussi être écrit de manière équivalente:

```
@deprecated def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Annoter le constructeur principal

```
/**
 * @param num Numerator
```

```
 * @param denom Denominator
 * @throws ArithmeticException in case `denom` is `0`
 */
class Division @throws[ArithmeticException](/*no annotation parameters*/) protected (num: Int,
denom: Int) {
    private[this] val wrongValue = num / denom

    /** Integer number
     *  @param num Value */
    protected[Division] def this(num: Int) {
      this(num, 1)
    }
}
object Division {
  def apply(num: Int) = new Division(num)
  def apply(num: Int, denom: Int) = new Division(num, denom)
}
```

Le modificateur de visibilité (dans ce cas `protected`) devrait figurer après les annotations sur la même ligne. `@throws` l'annotation accepte des paramètres facultatifs (comme dans ce cas, `@throws` accepte une cause facultative), vous devez spécifier une liste de paramètres vide pour l'annotation: `()` avant les paramètres du constructeur.

Remarque: Plusieurs annotations peuvent être spécifiées, même du même type (annotations répétées).

De même avec une classe de cas sans méthode d'usine auxiliaire (et la cause spécifiée pour l'annotation):

```
case class Division @throws[ArithmeticException]("denom is 0") (num: Int, denom: Int) {
    private[this] val wrongValue = num / denom
}
```

Création de vos propres annotations

Vous pouvez créer vos propres annotations Scala en créant des classes dérivées de scala.annotation.StaticAnnotation ou scala.annotation.ClassfileAnnotation.

```
package animals
// Create Annotation `Mammal`
class Mammal(indigenous:String) extends scala.annotation.StaticAnnotation

// Annotate class Platypus as a `Mammal`
@Mammal(indigenous = "North America")
class Platypus{}
```

Les annotations peuvent ensuite être interrogées à l'aide de l'API de réflexion.

```
scala>import scala.reflect.runtime.{universe ⇒ u}

scala>val platypusType = u.typeOf[Platypus]
platypusType: reflect.runtime.universe.Type = animals.reflection.Platypus

scala>val platypusSymbol = platypusType.typeSymbol.asClass
```

```
platypusSymbol: reflect.runtime.universe.ClassSymbol = class Platypus

scala>platypusSymbol.annotations
List[reflect.runtime.universe.Annotation] = List(animals.reflection.Mammal("North America"))
```

Lire Annotations en ligne: https://riptutorial.com/fr/scala/topic/3783/annotations

Chapitre 3: Bibliothèque de continuation

Introduction

Le style de passage de continuation est une forme de flux de contrôle qui consiste à passer à des fonctions le reste du calcul en tant qu'argument de "continuation". La fonction en question invoque ultérieurement cette continuation pour continuer l'exécution du programme. Une façon de penser à une continuation est une fermeture. La bibliothèque de suites Scala apporte des continuations délimitées sous la forme des primitives `shift` / `reset` à la langue.

bibliothèque de continuations: https://github.com/scala/scala-continuations

Syntaxe

- reset {...} // Les prolongements se prolongent jusqu'à la fin du bloc de réinitialisation
- shift {...} // Crée une suite indiquant après l'appel, en la passant à la fermeture
- Un @cpsParam [B, C] // Un calcul qui nécessite une fonction A => B pour créer une valeur de C
- @ cps [A] // Alias pour @cpsParam [A, A]
- @suspendable // Alias pour @cpsParam [Unit, Unit]

Remarques

`shift` et `reset` sont des structures de flux de contrôle primitives, comme `Int.+` est une opération primitive et `Long` est un type primitif. Ils sont plus primitifs que l'un ou l'autre dans le sens où les continuations délimitées peuvent réellement être utilisées pour construire presque toutes les structures de flux de contrôle. Ils ne sont pas très utiles "prêts à l'emploi", mais ils brillent vraiment lorsqu'ils sont utilisés dans des bibliothèques pour créer des API riches.

Les continuations et les monades sont également étroitement liées. Des continuations peuvent être faites dans la monade de continuation , et les monades sont des continuations car leur opération `flatMap` prend une suite en paramètre.

Examples

Les rappels sont des continuations

```
// Takes a callback and executes it with the read value
def readFile(path: String)(callback: Try[String] => Unit): Unit = ???

readFile(path) { _.flatMap { file1 =>
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}
```

L'argument function de `readFile` est une continuation, en ce `readFile` que `readFile` invoque pour continuer l'exécution du programme après avoir fait son travail.

Afin de limiter ce qui peut facilement devenir un rappel, nous utilisons la bibliothèque de continuations.

```
reset { // Reset is a delimiter for continuations.
  for { // Since the callback hell is relegated to continuation library machinery.
        // a for-comprehension can be used
    file1 <- shift(readFile(path1)) // shift has type (((A => B) => C) => A)
    // We use it as (((Try[String] => Unit) => Unit) => Try[String])
    // It takes all the code that occurs after it is called, up to the end of reset, and
    // makes it into a closure of type (A => B).
    // The reason this works is that shift is actually faking its return type.
    // It only pretends to return A.
    // It actually passes that closure into its function parameter (readFile(path1) here),
    // And that function calls what it thinks is a normal callback with an A.
    // And through compiler magic shift "injects" that A into its own callsite.
    // So if readFile calls its callback with parameter Success("OK"),
    // the shift is replaced with that value and the code is executed until the end of reset,
    // and the return value of that is what the callback in readFile returns.
    // If readFile called its callback twice, then the shift would run this code twice too.
    // Since readFile returns Unit though, the type of the entire reset expression is Unit
    //
    // Think of shift as shifting all the code after it into a closure,
    // and reset as resetting all those shifts and ending the closures.
    file2 <- shift(readFile(path2))
  } processFiles(file1, file2)
}

// After compilation, shift and reset are transformed back into closures
// The for comprehension first desugars to:
reset {
  shift(readFile(path1)).flatMap { file1 => shift(readFile(path2)).foreach { file2 =>
processFiles(file1, file2) } }
}
// And then the callbacks are restored via CPS transformation
readFile(path1) { _.flatMap { file1 => // We see how shift moves the code after it into a
closure
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}  // And we see how reset closes all those closures
// And it looks just like the old version!
```

Créer des fonctions qui prennent des continuités

Si `shift` est appelé en dehors d'un bloc de `reset` délimitation, il peut être utilisé pour créer des fonctions qui créent elles-mêmes des continuations dans un bloc de `reset` . Il est important de noter que le type de `shift` n'est pas juste `(((A => B) => C) => A)` , il est en fait `(((A => B) => C) => (A @cpsParam[B, C]))` . Les annotations indiquent où les transformations CPS sont nécessaires. Les fonctions appelant `shift` sans `reset` ont leur type de retour "infecté" avec cette annotation.

À l'intérieur d'un bloc de `reset` , la valeur `A @cpsParam[B, C]` semble avoir la valeur `A` , bien que ce ne soit que de la simulation. La suite nécessaire pour terminer le calcul a le type `A => B` , donc le code suivant une méthode qui renvoie ce type doit renvoyer `B C` est le type de retour "réel", et

après la transformation CPS, l'appel de fonction a le type `C`

Maintenant, l'exemple tiré du Scaladoc de la bibliothèque

```
val sessions = new HashMap[UUID, Int=>Unit]
def ask(prompt: String): Int @suspendable = // alias for @cpsParam[Unit, Unit]. @cps[Unit] is
also an alias. (@cps[A] = @cpsParam[A,A])
  shift {
    k: (Int => Unit) => {
      println(prompt)
      val id = uuidGen
      sessions += id -> k
    }
  }

def go(): Unit = reset {
  println("Welcome!")
  val first = ask("Please give me a number") // Uses CPS just like shift
  val second = ask("Please enter another number")
  printf("The sum of your numbers is: %d\n", first + second)
}
```

Ici, `ask` stockera la suite dans une map, et plus tard, un autre code pourra récupérer cette "session" et transmettre le résultat de la requête à l'utilisateur. De cette façon, `go` peut réellement utiliser une bibliothèque asynchrone alors que son code ressemble à un code impératif normal.

Lire Bibliothèque de continuation en ligne: https://riptutorial.com/fr/scala/topic/8312/bibliotheque-de-continuation

Chapitre 4: Classe d'option

Syntaxe

- class Some [+ T] (valeur: T) étend Option [T]
- objet Aucun étend Option [Nothing]
- Option [T] (valeur: T)

  Constructeur pour créer soit `Some(value)` ou `None` selon la valeur fournie.

Examples

Options en tant que collections

`Option` ont des fonctions utiles de niveau supérieur qui peuvent être facilement comprises en affichant les options sous forme de *collections avec zéro ou un élément* - où `None` se comporte comme la collection vide et `Some(x)` se comporte comme une collection avec un seul élément, `x` .

```
val option: Option[String] = ???

option.map(_.trim) // None if option is None, Some(s.trim) if Some(s)
option.foreach(println) // prints the string if it exists, does nothing otherwise
option.forall(_.length > 4) // true if None or if Some(s) and s.length > 4
option.exists(_.length > 4) // true if Some(s) and s.length > 4
option.toList // returns an actual list
```

Utilisation de l'option au lieu de null

En Java (et dans d'autres langages), l'utilisation de `null` est un moyen courant d'indiquer qu'il n'y a pas de valeur attachée à une variable de référence. Dans Scala, utiliser `Option` est préférable à l'utilisation de `null` . `Option` encapsule les valeurs *pouvant* être `null` .

`None` est une sous-classe de `Option` enveloppe une référence null. `Some` est une sous-classe d' `Option` enveloppant une référence non NULL.

Envelopper une référence est facile:

```
val nothing = Option(null) // None
val something = Option("Aren't options cool?") // Some("Aren't options cool?")
```

Il s'agit d'un code typique lors de l'appel d'une bibliothèque Java susceptible de renvoyer une référence null:

```
val resource = Option(JavaLib.getResource())
// if null, then resource = None
```

```
// else resource = Some(resource)
```

Si `getResource()` renvoie une valeur `null` , la `resource` sera un objet `None` . Sinon, ce sera un objet `Some(resource)` . La manière préférée de traiter une `Option` consiste à utiliser les fonctions d'ordre supérieur disponibles dans le type d' `Option` . Par exemple, si vous voulez vérifier si votre valeur n'est pas `None` (similaire à vérifier si la `value == null`), vous utiliseriez la fonction `isDefined` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isDefined) {  // resource is `Some(_)` type
  val r: Resource = resource.get
  r.connect()
}
```

De même, pour rechercher une référence `null` vous pouvez le faire:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isEmpty) { // resource is `None` type.
  System.out.println("Resource is empty! Cannot connect.")
}
```

Il est préférable de traiter l'exécution conditionnelle sur la valeur `Option.get` d'une `Option` (sans utiliser la méthode `Option.get` «exceptionnelle») en traitant l' `Option` comme une monade et en utilisant `foreach` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
resource foreach (r => r.connect())
// if r is defined, then r.connect() is run
// if r is empty, then it does nothing
```

Si une instance `Resource` est requise (par rapport à une instance `Option[Resource]`), vous pouvez toujours utiliser `Option` pour vous protéger contre les valeurs NULL. Ici, la méthode `getOrElse` fournit une valeur par défaut:

```
lazy val defaultResource = new Resource()
val resource: Resource = Option(JavaLib.getResource()).getOrElse(defaultResource)
```

Le code Java ne gèrera pas facilement l’ `Option` de Scala, donc lorsqu’il transmet des valeurs au code Java, il est préférable de déballer une `Option` , en transmettant `null` ou une valeur par défaut sensible, le cas échéant:

```
val resource: Option[Resource] = ???
JavaLib.sendResource(resource.orNull)
JavaLib.sendResource(resource.getOrElse(defaultResource)) //
```

Les bases

Une `Option` est une structure de données contenant une valeur unique ou aucune valeur. Une `Option` peut être considérée comme une collection de zéro ou d'un élément.

Option est une classe abstraite avec deux enfants: `Some` et `None` .

`Some` contiennent une seule valeur et `None` ne contient aucune valeur.

`Option` est utile dans les expressions qui utiliseraient autrement `null` pour représenter l'absence d'une valeur concrète. Cela protège contre une `NullPointerException` et permet la composition de nombreuses expressions qui pourraient ne pas renvoyer de valeur en utilisant des combinateurs tels que `Map` , `FlatMap` , etc.

Exemple avec carte

```
val countries = Map(
  "USA" -> "Washington",
  "UK" -> "London",
  "Germany" -> "Berlin",
  "Netherlands" -> "Amsterdam",
  "Japan" -> "Tokyo"
)

println(countries.get("USA")) // Some(Washington)
println(countries.get("France")) // None
println(countries.get("USA").get) // Washington
println(countries.get("France").get) // Error: NoSuchElementException
println(countries.get("USA").getOrElse("Nope")) // Washington
println(countries.get("France").getOrElse("Nope")) // Nope
```

`Option[A]` est **scellée** et ne peut donc pas être étendue. Par conséquent, sa sémantique est stable et peut être invoquée.

Options pour la compréhension

`Option` ont une méthode `flatMap` . Cela signifie qu'ils peuvent être utilisés pour la compréhension. De cette manière, nous pouvons lever des fonctions régulières pour travailler sur `Option` s sans avoir à les redéfinir.

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = Option(2)

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = Some(3)
```

Lorsque l' une des valeurs est un `None` résultat de fin du calcul sera `None` aussi bien.

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = None

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
```

```
// myResult: Option[Int] = None
```

Remarque: ce modèle s'étend plus généralement aux concepts appelés `Monad` s. (Plus d'informations devraient être disponibles sur les pages relatives à la compréhension et `Monad` s)

En général, il n'est pas possible de mélanger différentes monades dans une compréhension. Mais comme `Option` peut être facilement convertie en une `Iterable` , nous pouvons facilement mélanger `Option` s et `Iterable` s en appelant la méthode `.toIterable` .

```
val option: Option[Int] = Option(1)
val iterable: Iterable[Int] = Iterable(2, 3, 4, 5)

// does NOT compile since we cannot mix Monads in a for comprehension
// val myResult = for {
//   optionValue <- option
//   iterableValue <- iterable
//} yield optionValue + iterableValue

// It does compile when adding a .toIterable on the option
val myResult = for {
  optionValue <- option.toIterable
  iterableValue <- iterable
} yield optionValue + iterableValue
// myResult: Iterable[Int] = List(2, 3, 4, 5)
```

Une petite note: si nous avions défini le notre pour la compréhension, la compréhension serait compilée car notre option serait convertie implicitement. Pour cette raison, il est utile de toujours ajouter cette `.toIterable` (ou fonction correspondante selon la collection que vous utilisez) pour la cohérence.

Lire Classe d'option en ligne: https://riptutorial.com/fr/scala/topic/2293/classe-d-option

Chapitre 5: Classes de cas

Syntaxe

- case class Foo () // Les classes de cas sans paramètres doivent avoir une liste vide
- case class Foo (a1: A1, ..., aN: AN) // Crée une classe de cas avec les champs a1 ... aN
- objet case Bar // Créer une classe de cas singleton

Examples

Égalité de classe de cas

Une fonctionnalité fournie gratuitement par les classes de cas est une méthode `equals` la valeur générée automatiquement qui vérifie l'égalité de valeur de tous les champs de membre individuels au lieu de vérifier uniquement l'égalité de référence des objets.

Avec des classes ordinaires:

```
class Foo(val i: Int)
val a = new Foo(3)
val b = new Foo(3)
println(a == b)// "false" because they are different objects
```

Avec des classes de cas:

```
case class Foo(i: Int)
val a = Foo(3)
val b = Foo(3)
println(a == b)// "true" because their members have the same value
```

Artefacts de code générés

Le modificateur de `case` permet au compilateur Scala de générer automatiquement un code standard commun pour la classe. L'implémentation manuelle de ce code est fastidieuse et source d'erreurs. La définition de classe de cas suivante:

```
case class Person(name: String, age: Int)
```

... le code suivant sera automatiquement généré:

```
class Person(val name: String, val age: Int)
  extends Product with Serializable
{
  def copy(name: String = this.name, age: Int = this.age): Person =
    new Person(name, age)

  def productArity: Int = 2
```

```
  def productElement(i: Int): Any = i match {
    case 0 => name
    case 1 => age
    case _ => throw new IndexOutOfBoundsException(i.toString)
  }

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productPrefix: String = "Person"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[Person]

  override def hashCode(): Int = scala.runtime.ScalaRunTime._hashCode(this)

  override def equals(obj: Any): Boolean = this.eq(obj) || obj match {
    case that: Person => this.name == that.name && this.age == that.age
    case _ => false
  }

  override def toString: String =
    scala.runtime.ScalaRunTime._toString(this)
}
```

Le modificateur de `case` génère également un objet compagnon:

```
object Person extends AbstractFunction2[String, Int, Person] with Serializable {
  def apply(name: String, age: Int): Person = new Person(name, age)

  def unapply(p: Person): Option[(String, Int)] =
    if(p == null) None else Some((p.name, p.age))
}
```

Appliqué à un `object` , le modificateur de `case` a des effets similaires (quoique moins dramatiques). Ici, les gains principaux sont une implémentation de `toString` et une valeur `hashCode` cohérente entre les processus. Notez que les objets case (correctement) utilisent une égalité de référence:

```
object Foo extends Product with Serializable {
  def productArity: Int = 0

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productElement(i: Int): Any =
    throw new IndexOutOfBoundsException(i.toString)

  def productPrefix: String = "Foo"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[this.type]

  override def hashCode(): Int = 70822 // "Foo".hashCode()

  override def toString: String = "Foo"
}
```

Il est toujours possible d'implémenter manuellement des méthodes qui seraient autrement

fournies par le modificateur de `case` dans la classe elle-même et dans son objet compagnon.

Les bases de la classe de cas

Par rapport aux classes ordinaires, la notation des classes de cas présente plusieurs avantages:

- Tous les arguments de constructeur sont `public` et peuvent être accédés sur des objets initialisés (normalement ce n'est pas le cas, comme démontré ici):

```
case class Dog1(age: Int)
val x = Dog1(18)
println(x.age) // 18 (success!)

class Dog2(age: Int)
val x = new Dog2(18)
println(x.age) // Error: "value age is not a member of Dog2"
```

- Il fournit une implémentation pour les méthodes suivantes: `toString` , `equals` , `hashCode` (basé sur les propriétés), `copy` , `apply` et ne pas `unapply` :

```
case class Dog(age: Int)
val d1 = Dog(10)
val d2 = d1.copy(age = 15)
```

- Il fournit un mécanisme pratique pour la correspondance de modèle:

```
sealed trait Animal // `sealed` modifier allows inheritance within current build-unit
only
case class Dog(age: Int) extends Animal
case class Cat(owner: String) extends Animal
val x: Animal = Dog(18)
x match {
    case Dog(x) => println(s"It's a $x years old dog.")
    case Cat(x) => println(s"This cat belongs to $x.")
}
```

Classes de cas et immunité

Le compilateur Scala préfixe chaque argument de la liste de paramètres par défaut avec `val` . Cela signifie que, par défaut, les classes de cas sont immuables. Chaque paramètre reçoit une méthode d'accesseur, mais il n'y a pas de méthode de mutation. Par exemple:

```
case class Foo(i: Int)

val fooInstance = Foo(1)
val j = fooInstance.i       // get
fooInstance.i = 2           // compile-time exception (mutation: reassignment to val)
```

Déclarer un paramètre dans une classe de cas comme `var` remplace le comportement par défaut et rend la classe de casable mutable:

```
case class Bar(var i: Int)

val barInstance = Bar(1)
val j = barInstance.i       // get
barInstance.i = 2           // set
```

Un autre cas où une classe de cas est «mutable» est lorsque la valeur de la classe de casse est modifiable:

```
import scala.collection._

case class Bar(m: mutable.Map[Int, Int])

val barInstance = Bar(mutable.Map(1 -> 2))
barInstance.m.update(1, 3)                  // mutate m
barInstance                                 // Bar(Map(1 -> 3)
```

Notez que la «mutation» qui se produit ici se trouve dans la carte sur laquelle `m` pointe, pas sur `m` lui-même. Ainsi, si un autre objet avait `m` tant que membre, il verrait également le changement. Notez que dans l'exemple suivant, change `instanceA` modifie également `instanceB` :

```
import scala.collection.mutable

case class Bar(m: mutable.Map[Int, Int])

val m = mutable.Map(1 ->2)
val barInstanceA = Bar(m)
val barInstanceB = Bar(m)
barInstanceA.m.update(1,3)
barInstanceA  // Bar = Bar(Map(1 -> 3))
barInstanceB  // Bar = Bar(Map(1 -> 3))
m  // scala.collection.mutable.Map[Int,Int] = Map(1 -> 3)
```

Créer une copie d'un objet avec certaines modifications

Les classes de cas fournissent une méthode de `copy` qui crée un nouvel objet qui partage les mêmes champs que l'ancien, avec certaines modifications.

Nous pouvons utiliser cette fonctionnalité pour créer un nouvel objet à partir d'un précédent qui présente certaines des mêmes caractéristiques. Cette classe de cas simple pour démontrer cette fonctionnalité:

```
case class Person(firstName: String, lastName: String, grade: String, subject: String)
val putu = Person("Putu", "Kevin", "A1", "Math")
val mark = putu.copy(firstName = "Ketut", lastName = "Mark")
// mark: People = People(Ketut,Mark,A1,Math)
```

Dans cet exemple, nous pouvons voir que les deux objets partagent des caractéristiques similaires (`grade = A1` , `subject = Math`), sauf s'ils ont été spécifiés dans la copie (`firstName` et `lastName`).

Classes de cas à élément unique pour la sécurité de type

Pour atteindre la sécurité de type, nous voulons parfois éviter l'utilisation de types primitifs sur notre domaine. Par exemple, imaginez une `Person` avec un `name` . En règle générale, nous coderons le `name` tant que `String` . Cependant, il ne serait pas difficile de mélanger une `String` représentant une `Person` de » `name` d'une `String` représentant un message d'erreur:

```
def logError(message: ErrorMessage): Unit = ???
case class Person(name: String)
val maybeName: Either[String, String] = ??? // Left is error, Right is name
maybeName.foreach(logError) // But that won't stop me from logging the name as an error!
```

Pour éviter de tels écueils, vous pouvez encoder les données comme ceci:

```
case class PersonName(value: String)
case class ErrorMessage(value: String)
case class Person(name: PersonName)
```

et maintenant notre code ne compilera pas si nous `PersonName` avec `ErrorMessage` , ou même une `String` ordinaire.

```
val maybeName: Either[ErrorMessage, PersonName] = ???
maybeName.foreach(reportError) // ERROR: tried to pass PersonName; ErrorMessage expected
maybeName.swap.foreach(reportError) // OK
```

Mais cela entraîne un petit temps d’exécution, car nous devons maintenant cocher / décocher les `String` vers / depuis leurs conteneurs `PersonName` . Pour éviter cela, on peut créer des classes de valeur `PersonName` et `ErrorMessage` :

```
case class PersonName(val value: String) extends AnyVal
case class ErrorMessage(val value: String) extends AnyVal
```

Lire Classes de cas en ligne: https://riptutorial.com/fr/scala/topic/1022/classes-de-cas

Chapitre 6: Classes de types

Remarques

Pour éviter les problèmes de sérialisation, en particulier dans les environnements distribués (par exemple, Apache Spark), il est recommandé d'implémenter le trait `Serializable` pour les instances de classe de type.

Examples

Classe de type simple

Une classe de type est simplement un `trait` avec un ou plusieurs paramètres de type:

```
trait Show[A] {
  def show(a: A): String
}
```

Au lieu d'étendre une classe de type, une instance implicite de la classe de type est fournie pour chaque type pris en charge. Placer ces implémentations dans l'objet compagnon de la classe de type permet une résolution implicite pour fonctionner sans importations spéciales:

```
object Show {
  implicit val intShow: Show[Int] = new Show {
    def show(x: Int): String = x.toString
  }

  implicit val dateShow: Show[java.util.Date] = new Show {
    def show(x: java.util.Date): String = x.getTime.toString
  }

  // ..etc
}
```

Si vous voulez garantir qu'un paramètre générique transmis à une fonction a une instance d'une classe de type, utilisez des paramètres implicites:

```
def log[A](a: A)(implicit showInstance: Show[A]): Unit = {
  println(showInstance.show(a))
}
```

Vous pouvez également utiliser un contexte lié :

```
def log[A: Show](a: A): Unit = {
  println(implicitly[Show[A]].show(a))
}
```

Appelez la méthode de `log` ci-dessus comme toute autre méthode. Il ne parviendra pas à compiler

si un implicite `Show[A]` la mise en œuvre ne peut être trouvée pour l' `A` vous passez à vous `log`

```
log(10) // prints: "10"
log(new java.util.Date(1469491668401L) // prints: "1469491668401"
log(List(1,2,3)) // fails to compile with
                 // could not find implicit value for evidence parameter of type
Show[List[Int]]
```

Cet exemple implémente la classe de type `Show` . C'est une classe de type commune utilisée pour convertir des instances arbitraires de types arbitraires en `String` s. Même si chaque objet a une méthode `toString` , il n'est pas toujours clair si `toString` est défini de manière utile. Avec l'utilisation de la classe de type `Show` , vous pouvez garantir que tout élément transmis à `log` a une conversion bien définie en `String` .

Extension d'une classe de type

Cet exemple traite de l'extension de la classe de type ci-dessous.

```
trait Show[A] {
  def show: String
}
```

Pour rendre une classe que *vous* contrôlez (et est écrite en Scala), étendez la classe de type, ajoutez un implicite à son objet compagnon. Montrons comment nous pouvons obtenir la classe `Person` de cet exemple pour étendre `Show` :

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}
```

Nous pouvons faire en sorte que cette classe étende `Show` en ajoutant un objet implicite à l'objet compagnon de `Person` :

```
object Person {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}
```

Un objet compagnon *doit être dans le même fichier* que la classe, vous avez donc besoin à la fois `class Person` et d' `object Person` dans le même fichier.

Pour créer une classe que vous ne contrôlez pas ou qui n'est pas écrite en Scala, étendez la classe de type, ajoutez un implicite à l'objet compagnon de la classe de type, comme indiqué dans l'exemple de classe de type simple .

Si vous ne contrôlez ni la classe ni la classe de type, créez un fichier implicite comme ci-dessus et `import` le. En utilisant la méthode `log` sur l' exemple de type Simple type :

```
object MyShow {
```

```
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}

def logPeople(persons: Person*): Unit = {
  import MyShow.personShow
  persons foreach { p => log(p) }
}
```

Ajouter des fonctions de classe de type aux types

L'implémentation Scala des classes de type est plutôt verbeuse. Une façon de réduire la verbosité consiste à introduire des "classes d'opération". Ces classes encapsulent automatiquement une variable / valeur lorsqu'elles sont importées pour étendre les fonctionnalités.

Pour illustrer cela, commençons par créer une classe de type simple:

```
// The mathematical definition of "Addable" is "Semigroup"
trait Addable[A] {
  def add(x: A, y: A): A
}
```

Ensuite, nous allons implémenter le trait (instancier la classe de type):

```
object Instances {

  // Instance for Int
  // Also called evidence object, meaning that this object saw that Int learned how to be
added
  implicit object addableInt extends Addable[Int] {
    def add(x: Int, y: Int): Int = x + y
  }

  // Instance for String
  implicit object addableString extends Addable[String] {
    def add(x: String, y: String): String = x + y
  }

}
```

Pour le moment, il serait très compliqué d'utiliser nos instances Addable:

```
import Instances._
val three = addableInt.add(1,2)
```

Nous préférerions simplement écrire write `1.add(2)` . Par conséquent, nous allons créer une "Classe d'opération" (également appelée "Classe Ops") qui `Addable` toujours un type qui implémente `Addable` .

```
object Ops {
  implicit class AddableOps[A](self: A)(implicit A: Addable[A]) {
    def add(other: A): A = A.add(self, other)
```

```
  }
}
```

Maintenant, nous pouvons utiliser notre nouvelle fonction `add` comme si elle faisait partie de `Int` et `String` :

```
object Main {

  import Instances._ // import evidence objects into this scope
  import Ops._       // import the wrappers

  def main(args: Array[String]): Unit = {

    println(1.add(5))
    println("mag".add("net"))
    // println(1.add(3.141)) // Fails because we didn't create an instance for Double

  }
}
```

Les classes "Ops" peuvent être créées automatiquement par les macros de la bibliothèque simulacrum :

```
import simulacrum._

@typeclass trait Addable[A] {
  @op("|+|") def add(x: A, y: A): A
}
```

Lire Classes de types en ligne: https://riptutorial.com/fr/scala/topic/3835/classes-de-types

Chapitre 7: Classes et Objets

Syntaxe

- `class MyClass{} // curly braces are optional here as class body is empty`
- `class MyClassWithMethod {def method: MyClass = ???}`
- `new MyClass() //Instantiate`
- `object MyObject // Singleton object`
- `class MyClassWithGenericParameters[V1, V2](vl: V1, i: Int, v2: V2)`
- `class MyClassWithImplicitFieldCreation[V1](val v1: V1, val i: Int)`
- `new MyClassWithGenericParameters(2.3, 4, 5)` ou avec un type différent: `new MyClassWithGenericParameters[Double, Any](2.3, 4, 5)`
- `class MyClassWithProtectedConstructor protected[my.pack.age](s: String)`

Examples

Instancier des instances de classe

Une classe dans Scala est un «modèle» d'une instance de classe. Une instance contient l'état et le comportement définis par cette classe. Pour déclarer une classe:

```
class MyClass{}  // curly braces are optional here as class body is empty
```

Une instance peut être instanciée à l'aide du `new` mot clé:

```
var instance = new MyClass()
```

ou:

```
var instance = new MyClass
```

Les parenthèses sont facultatives dans Scala pour créer des objets à partir d'une classe qui possède un constructeur sans argument. Si un constructeur de classe prend des arguments:

```
class MyClass(arg : Int)        // Class definition
var instance = new MyClass(2)   // Instance instantiation
instance.arg                    // not allowed
```

Ici, `MyClass` nécessite un argument `Int` , qui ne peut être utilisé qu'en interne pour la classe. `arg` ne peut pas être accessible en dehors `MyClass` sauf si elle est déclarée comme un champ:

```
class MyClass(arg : Int){
    val prop = arg  // Class field declaration
}

var obj = new MyClass(2)
obj.prop     // legal statement
```

Sinon, il peut être déclaré public dans le constructeur:

```
class MyClass(val arg : Int)   // Class definition with arg declared public
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   //arg is now visible to clients
```

Classe d'instanciation sans paramètre: {} vs ()

Disons que nous avons une classe MyClass sans argument de constructeur:

```
class MyClass
```

En Scala, nous pouvons l'instancier en utilisant la syntaxe ci-dessous:

```
val obj = new MyClass()
```

Ou nous pouvons simplement écrire:

```
val obj = new MyClass
```

Mais, sans attention, dans certains cas, la parenthèse facultative peut produire un comportement inattendu. Supposons que nous voulions créer une tâche qui devrait s'exécuter dans un thread séparé. Voici le code exemple:

```
val newThread = new Thread { new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    }

newThread.start   // prints no output
```

Nous pouvons penser que cet exemple de code, s'il est exécuté, imprime une `Performing task.` , mais à notre grande surprise, il n'imprimera rien. Voyons ce qui se passe ici. Si vous regardez de plus près, nous avons utilisé des accolades `{}` , juste après le `new Thread` . Il a créé une classe annonymous qui étend `Thread` :

```
val newThread = new Thread {
  //creating anonymous class extending Thread
}
```

Et puis, dans le corps de cette classe anonymes, nous avons défini notre tâche (créant à nouveau une classe annoncymée implémentant une interface `Runnable`). On aurait donc pu penser que nous `public Thread(Runnable target)` constructeur `public Thread(Runnable target)` mais en fait (en ignorant optional `()`) nous avons utilisé le constructeur `public Thread()` sans rien défini dans la méthode body of `run()` . Pour corriger le problème, nous devons utiliser des parenthèses au lieu d'accolades.

```
val newThread = new Thread ( new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    )
```

En d'autres termes, ici `{}` et `()` ne sont pas *interchangeables* .

Singleton & Companion Objects

Objets singleton

Scala prend en charge les membres statiques, mais pas de la même manière que Java. Scala fournit une alternative à cela appelé *Singleton Objects* . Les objets singleton sont similaires à une classe normale, sauf qu'ils ne peuvent pas être instanciés à l'aide du `new` mot clé. Voici un exemple de classe singleton:

```
object Factorial {
    private val cache = Map[Int, Int]()
    def getCache = cache
}
```

Notez que nous avons utilisé le mot-clé `object` pour définir un objet singleton (au lieu de «class» ou «trait»). Comme les objets singleton ne peuvent pas être instanciés, ils ne peuvent pas avoir de paramètres. L'accès à un objet singleton ressemble à ceci:

```
Factorial.getCache() //returns the cache
```

Notez que cela ressemble exactement à l'accès à une méthode statique dans une classe Java.

Objets compagnons

Dans Scala, les objets singleton peuvent partager le nom d'une classe correspondante. Dans un tel scénario, l'objet singleton est appelé *objet compagnon* . Par exemple, sous la classe `Factorial` est défini et un objet compagnon (également appelé `Factorial`) est défini en dessous. Par convention, les objets compagnons sont définis dans le même fichier que leur classe compagnon.

```
class Factorial(num : Int) {

  def fact(num : Int) : Int = if (num <= 1) 1 else (num * fact(num - 1))

  def calculate() : Int = {
    if (!Factorial.cache.contains(num)) {    // num does not exists in cache
      val output = fact(num) // calculate factorial
      Factorial.cache += (num -> output)     // add new value in cache
    }

    Factorial.cache(num)
```

```
  }
}

object Factorial {
  private val cache = scala.collection.mutable.Map[Int, Int]()
}

val factfive = new Factorial(5)
factfive.calculate  // Calculates the factorial of 5 and stores it
factfive.calculate  // uses cache this time
val factfiveagain = new Factorial(5)
factfiveagain.calculate  // Also uses cache
```

Dans cet exemple, nous utilisons un `cache` privé pour stocker la factorielle d'un nombre afin de gagner du temps de calcul pour des nombres répétés.

Ici, l' `object Factorial` est un objet compagnon et la `class Factorial` est la classe compagnon correspondante. Les objets compagnons et les classes peuvent accéder `private` membres `private` chacun. Dans l'exemple ci-dessus, la classe `Factorial` accède au membre de `cache` privé de son objet compagnon.

Notez qu'une nouvelle instanciation de la classe utilisera toujours le même objet compagnon, donc toute modification des variables membres de cet objet sera conservée.

Objets

Alors que les classes ressemblent plus à des plans, les objets sont statiques (c'est-à-dire déjà instanciés):

```
object Dog {
    def bark: String = "Raf"
}

Dog.bark() // yields "Raf"
```

Ils sont souvent utilisés comme compagnons de cours, ils vous permettent d'écrire:

```
class Dog(val name: String) {

}

object Dog {
    def apply(name: String): Dog = new Dog(name)
}

val dog = Dog("Barky") // Object
val dog = new Dog("Barky") // Class
```

Vérification du type d'instance

Type de vérification : `variable.isInstanceOf[Type]`

Avec correspondance de motif (pas si utile dans cette forme):

```
variable match {
  case _: Type => true
  case _ => false
}
```

`isInstanceOf` et la correspondance de modèle `isInstanceOf` uniquement le type de l'objet, pas son paramètre générique (pas de réification de type), sauf pour les tableaux:

```
val list: List[Any] = List(1, 2, 3)             //> list  : List[Any] = List(1, 2, 3)

val upcasting = list.isInstanceOf[Seq[Int]]     //> upcasting  : Boolean = true

val shouldBeFalse = list.isInstanceOf[List[String]]
                                                //> shouldBeFalse  : Boolean = true
```

Mais

```
val chSeqArray: Array[CharSequence] = Array("a") //> chSeqArray  : Array[CharSequence] =
Array(a)
val correctlyReified = chSeqArray.isInstanceOf[Array[String]]
                                               //> correctlyReified  : Boolean = false


val stringIsACharSequence: CharSequence = ""    //> stringIsACharSequence  : CharSequence = ""

val sArray = Array("a")                         //> sArray  : Array[String] = Array(a)
val correctlyReified = sArray.isInstanceOf[Array[String]]
                                                //> correctlyReified  : Boolean = true

//val arraysAreInvariantInScala: Array[CharSequence] = sArray
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
//Note: String <: CharSequence, but class Array is invariant in type T.
//You may wish to investigate a wildcard type such as `_ <: CharSequence`. (SLS 3.2.10)
//Workaround:
val arraysAreInvariantInScala: Array[_ <: CharSequence] = sArray
                                                //> arraysAreInvariantInScala  : Array[_ <:
CharSequence] = Array(a)


val arraysAreCovariantOnJVM = sArray.isInstanceOf[Array[CharSequence]]
                                                //> arraysAreCovariantOnJVM  : Boolean = true
```

Type casting : `variable.asInstanceOf[Type]`

Avec correspondance de motif :

```
variable match {
  case _: Type => true
}
```

Exemples:

```
  val x = 3                                       //> x  : Int = 3
  x match {
    case _: Int => true//better: do something
```

```
    case _ => false
  }                                           //> res0: Boolean = true

  x match {
    case _: java.lang.Integer => true//better: do something
    case _ => false
  }                                           //> res1: Boolean = true

  x.isInstanceOf[Int]                         //> res2: Boolean = true

  //x.isInstanceOf[java.lang.Integer]//fruitless type test: a value of type Int cannot also be
a Integer

  trait Valuable { def value: Int}
  case class V(val value: Int) extends Valuable

  val y: Valuable = V(3)                      //> y  : Valuable = V(3)
  y.isInstanceOf[V]                           //> res3: Boolean = true
  y.asInstanceOf[V]                           //> res4: V = V(3)
```

Remarque: il ne s'agit que du comportement sur la JVM, sur d'autres plates-formes (JS, native), le type casting / verification peut se comporter différemment.

Constructeurs

Constructeur principal

En Scala, le constructeur principal est le corps de la classe. Le nom de la classe est suivi d'une liste de paramètres, qui sont les arguments du constructeur. (Comme pour toute fonction, une liste de paramètres vide peut être omise.)

```
class Foo(x: Int, y: String) {
    val xy: String = y * x
    /* now xy is a public member of the class */
}

class Bar {
    ...
}
```

Les paramètres de construction d'une instance ne sont pas accessibles en dehors de son corps de constructeur, à moins qu'ils ne soient marqués comme membre d'instance par le mot clé `val` :

```
class Baz(val z: String)
// Baz has no other members or methods, so the body may be omitted

val foo = new Foo(4, "ab")
val baz = new Baz("I am a baz")
foo.x // will not compile: x is not a member of Foo
foo.xy // returns "abababab": xy is a member of Foo
baz.z // returns "I am a baz": z is a member of Baz
val bar0 = new Bar
val bar1 = new Bar() // Constructor parentheses are optional here
```

Toutes les opérations à effectuer lorsqu'une instance d'un objet est instanciée sont écrites directement dans le corps de la classe:

```
class DatabaseConnection
    (host: String, port: Int, username: String, password: String) {
    /* first connect to the DB, or throw an exception */
    private val driver = new AwesomeDB.Driver()
    driver.connect(host, port, username, password)
    def isConnected: Boolean = driver.isConnected
    ...
}
```

Notez qu'il est conseillé de placer le moins d'effets secondaires possible dans le constructeur. au lieu du code ci-dessus, on devrait envisager d'avoir des méthodes de `connect` et de `disconnect` pour que le code consommateur soit responsable de la planification des entrées-sorties.

Constructeurs Auxiliaires

Une classe peut avoir des constructeurs supplémentaires appelés «constructeurs auxiliaires». Celles-ci sont définies par des définitions de constructeur sous la forme `def this(...) = e` , où `e` doit invoquer un autre constructeur:

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

// usage:
new Person("Grace Hopper").fullName // returns Grace Hopper
new Person("Grace", "Hopper").fullName // returns Grace Hopper
```

Cela implique que chaque constructeur peut avoir un modificateur différent: seuls certains peuvent être disponibles publiquement:

```
class Person private(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

new Person("Ada Lovelace") // won't compile
new Person("Ada", "Lovelace") // compiles
```

De cette manière, vous pouvez contrôler la manière dont le code consommateur peut instancier la classe.

Lire Classes et Objets en ligne: https://riptutorial.com/fr/scala/topic/2047/classes-et-objets

Chapitre 8: Collections

Examples

Trier une liste

En supposant la liste suivante, nous pouvons trier une variété de manières.

```
val names = List("Kathryn", "Allie", "Beth", "Serin", "Alana")
```

Le comportement par défaut de `math.Ordering` `sorted()` consiste à utiliser `math.Ordering` , qui pour les chaînes génère un tri lexographique :

```
names.sorted
// results in: List(Alana, Allie, Beth, Kathryn, Serin)
```

`sortWith` vous permet de fournir votre propre commande en utilisant une fonction de comparaison:

```
names.sortWith(_.length < _.length)
// results in: List(Beth, Allie, Serin, Alana, Kathryn)
```

`sortBy` vous permet de fournir une fonction de transformation:

```
//A set of vowels to use
val vowels = Set('a', 'e', 'i', 'o', 'u')

//A function that counts the vowels in a name
def countVowels(name: String) = name.count(l => vowels.contains(l.toLower))

//Sorts by the number of vowels
names.sortBy(countVowels)
//result is: List(Kathryn, Beth, Serin, Allie, Alana)
```

Vous pouvez toujours inverser une liste ou une liste triée en utilisant `reverse:

```
names.sorted.reverse
//results in: List(Serin, Kathryn, Beth, Allie, Alana)
```

Les listes peuvent également être triées à l'aide de la méthode Java `java.util.Arrays.sort` et de son wrapper Scala `scala.util.Sorting.quickSort`

```
java.util.Arrays.sort(data)
scala.util.Sorting.quickSort(data)
```

Ces méthodes peuvent améliorer les performances lors du tri de collections plus volumineuses si les conversions de collections et le déballage / la boxe peuvent être évités. Pour une discussion plus détaillée sur les différences de performances, consultez l'article Scala Collection trié, trié et

triBy Performance .

Créer une liste contenant n copies de x

Pour créer une collection de `n` copies d'un objet `x` , utilisez la méthode de remplissage . Cet exemple crée une `List` , mais cela peut fonctionner avec d'autres collections pour lesquelles le `fill` sens:

```
// List.fill(n)(x)
scala > List.fill(3)("Hello World")
res0: List[String] = List(Hello World, Hello World, Hello World)
```

Cheatsheet Liste et Vecteur

> Il est désormais recommandé d'utiliser le `Vector` plutôt que la `List` car les implémentations ont de meilleures performances. Les caractéristiques de performances peuvent être trouvées ici . `Vector` peut être utilisé partout où la `List` est utilisée.

Création de liste

```
List[Int]()          // Declares an empty list of type Int
List.empty[Int]      // Uses `empty` method to declare empty list of type Int
Nil                  // A list of type Nothing that explicitly has nothing in it

List(1, 2, 3)        // Declare a list with some elements
1 :: 2 :: 3 :: Nil   // Chaining element prepending to an empty list, in a LISP-style
```

Prendre élément

```
List(1, 2, 3).headOption // Some(1)
List(1, 2, 3).head       // 1

List(1, 2, 3).lastOption // Some(3)
List(1, 2, 3).last       // 3, complexity is O(n)

List(1, 2, 3)(1)         // 2, complexity is O(n)
List(1, 2, 3)(3)         // java.lang.IndexOutOfBoundsException: 4
```

Éléments complémentaires

```
0 :: List(1, 2, 3)       // List(0, 1, 2, 3)
```

Ajouter des éléments

```
List(1, 2, 3) :+ 4       // List(1, 2, 3, 4), complexity is O(n)
```

Rejoindre (concaténer) les listes

```
List(1, 2) ::: List(3, 4) // List(1, 2, 3, 4)
```

```
List.concat(List(1,2), List(3, 4)) // List(1, 2, 3, 4)
List(1, 2) ++ List(3, 4)  // List(1, 2, 3, 4)
```

Opérations communes

```
List(1, 2, 3).find(_ == 3)                    // Some(3)
List(1, 2, 3).map(_ * 2)                      // List(2, 4, 6)
List(1, 2, 3).filter(_ % 2 == 1)              // List(1, 3)
List(1, 2, 3).fold(0)((acc, i) => acc + i * i) // 1 * 1 + 2 * 2 + 3 * 3 = 14
List(1, 2, 3).foldLeft("Foo")(_ + _.toString)  // "Foo123"
List(1, 2, 3).foldRight("Foo")(_ + _.toString) // "123Foo"
```

Carte cheatsheet

> Notez que cela concerne la création d'une collection de type `Map` , distincte de la méthode de `map` .

Création de carte

```
Map[String, Int]()
val m1: Map[String, Int] = Map()
val m2: String Map Int = Map()
```

Une carte peut être considérée comme une collection de `tuples` pour la plupart des opérations, où le premier élément est la clé et le second la valeur.

```
val l = List(("a", 1), ("b", 2), ("c", 3))
val m = l.toMap                               // Map(a -> 1, b -> 2, c -> 3)
```

Get element

```
val m = Map("a" -> 1, "b" -> 2, "c" -> 3)

m.get("a")  // Some(1)
m.get("d")  // None
m("a")      // 1
m("d")      // java.util.NoSuchElementException: key not found: d

m.keys      // Set(a, b, c)
m.values    // MapLike(1, 2, 3)
```

Ajouter des éléments

```
Map("a" -> 1, "b" -> 2) + ("c" -> 3)               // Map(a -> 1, b -> 2, c -> 3)
Map("a" -> 1, "b" -> 2) + ("a" -> 3)               // Map(a -> 3, b -> 2)
Map("a" -> 1, "b" -> 2) ++ Map("b" -> 3, "c" -> 4) // Map(a -> 1, b -> 3, c -> 4)
```

Opérations communes

Dans les opérations où une itération sur une carte se produit (`map` , `find` , `forEach` , etc.), les éléments de la collection sont des `tuples` . Le paramètre de fonction peut soit utiliser les

accesseurs de tuple (`_1` , `_2`), soit une fonction partielle avec un bloc de casse:

```
m.find(_._1 == "a")  // Some((a,1))
m.map {
  case (key, value) => (value, key)
}                    // Map(1 -> a, 2 -> b, 3 -> c)
m.filter(_._2 == 2)  // Map(b -> 2)
m.foldLeft(0){
  case (acc, (key, value: Int)) => acc + value
}                    // 6
```

Carte et filtre sur une collection

Carte

«Mappage» dans une collection utilise la fonction de `map` pour transformer chaque élément de cette collection de la même manière. La syntaxe générale est la suivante:

```
val someFunction: (A) => (B) = ???
collection.map(someFunction)
```

Vous pouvez fournir une fonction anonyme:

```
collection.map((x: T) => /*Do something with x*/)
```

Multiplication des nombres entiers par deux

```
// Initialize
val list = List(1,2,3)
// list: List[Int] = List(1, 2, 3)

// Apply map
list.map((item: Int) => item*2)
// res0: List[Int] = List(2, 4, 6)

// Or in a more concise way
list.map(_*2)
// res1: List[Int] = List(2, 4, 6)
```

Filtre

`filter` est utilisé lorsque vous souhaitez exclure ou filtrer certains éléments d'une collection. Comme avec `map` , la syntaxe générale prend une fonction, mais cette fonction doit renvoyer une valeur `Boolean` :

```
val someFunction: (a) => Boolean = ???
collection.filter(someFunction)
```

Vous pouvez fournir une fonction anonyme directement:

```
collection.filter((x: T) => /*Do something that returns a Boolean*/)
```

Vérification des numéros de paire

```
val list = 1 to 10 toList
// list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Filter out all elements that aren't evenly divisible by 2
list.filter((item: Int) => item % 2==0)
// res0: List[Int] = List(2, 4, 6, 8, 10)
```

Plus d'exemples de carte et de filtre

```
case class Person(firstName: String,
                  lastName: String,
                  title: String)

// Make a sequence of people
val people = Seq(
  Person("Millie", "Fletcher", "Mrs"),
  Person("Jim", "White", "Mr"),
  Person("Jenny", "Ball", "Miss") )


// Make labels using map
val labels = people.map( person =>
  s"${person.title}. ${person.lastName}"
)

// Filter the elements beginning with J
val beginningWithJ = people.filter(_.firstName.startsWith("J"))

// Extract first names and concatenate to a string
val firstNames = people.map(_.firstName).reduce( (a, b) => a + "," + b )
```

Introduction aux collections de la Scala

Selon ses auteurs , le cadre Scala Collections est conçu pour être facile à utiliser, concis, sûr, rapide et universel.

Le cadre est constitué de traits Scala conçus pour constituer des blocs de création de collections. Pour plus d'informations sur ces blocs de construction, lisez l'aperçu des collections Scala .

Ces collections intégrées sont séparées dans les packages immuables et mutables. Par défaut, les versions immuables sont utilisées. Construire une `List()` sans rien importer construira une liste *immuable* .

L'une des fonctionnalités les plus puissantes de l'infrastructure est l'interface cohérente et conviviale à travers des collections aux vues similaires. Par exemple, la somme de tous les

éléments d'une collection est la même pour les listes, les ensembles, les vecteurs, les seqs et les tableaux:

```
val numList = List[Int](1, 2, 3, 4, 5)
numList.reduce((n1, n2) => n1 + n2)  // 15

val numSet = Set[Int](1, 2, 3, 4, 5)
numSet.reduce((n1, n2) => n1 + n2)   // 15

val numArray = Array[Int](1, 2, 3, 4, 5)
numArray.reduce((n1, n2) => n1 + n2) // 15
```

Ces types partageant les mêmes idées héritent du trait `Traversable` .

> Il est désormais recommandé d'utiliser le `Vector` plutôt que la `List` car les implémentations ont de meilleures performances. Les caractéristiques de performances peuvent être trouvées ici . `Vector` peut être utilisé partout où la `List` est utilisée.

Types transversables

Classes de collection qui ont le `Traversable` trait mettre en œuvre `foreach` et héritera de nombreuses méthodes pour effectuer des opérations communes aux collections, toutes fonctionnent de manière identique. Les opérations les plus courantes sont répertoriées ici:

- Map - `map` , `flatMap` et `collect` produisent de nouvelles collections en appliquant une fonction à chaque élément de la collection originale.

```
List(1, 2, 3).map(num => num * 2) // double every number = List(2, 4, 6)

// split list of letters into individual strings and put them into the same list
List("a b c", "d e").flatMap(letters => letters.split(" ")) // = List("a", "b", "c", "d", "e")
```

- Conversions - `toList` , `toArray` et de nombreuses autres opérations de conversion transforment la collection actuelle en un type de collection plus spécifique. Ce sont généralement des méthodes précédées de «to» et du type plus spécifique (c.-à-d. «ToList» converti en une `List`).

```
val array: Array[Int] = List[Int](1, 2, 3).toArray // convert list of ints to array of ints
```

- Informations sur la taille - `isEmpty` , `nonEmpty` , `size` et `hasDefiniteSize` sont toutes des métadonnées sur l'ensemble. Cela permet aux opérations conditionnelles sur la collection ou au code de déterminer la taille de la collection, notamment si elle est infinie ou discrète.

```
List().isEmpty // true
List(1).nonEmpty // true
```

- Récupération d'éléments - `head` , `last` , `find` et leurs variantes `Option` permettent de récupérer le premier ou le dernier élément ou de rechercher un élément spécifique dans la collection.

```
val list = List(1, 2, 3)
list.head // = 1
list.last // = 3
```

- Les `filter` `tail` `slice` `drop` opérations de récupération de sous-collection - `filter` , `tail` , `slice` , `drop` , et d' autres opérations permettent de choisir les parties de la collection pour fonctionner sur plus loin.

```
List(-2, -1, 0, 1, 2).filter(num => num > 0) // = List(1, 2)
```

- Opérations de subdivision - `partition` , `splitAt` , `span` et `groupBy` divisent la collection actuelle en différentes parties.

```
// split numbers into < 0 and >= 0
List(-2, -1, 0, 1, 2).partition(num => num < 0) // = (List(-2, -1), List(0, 1, 2))
```

- Les tests d'éléments - `exists` , `forall` et `count` sont des opérations utilisées pour vérifier cette collection pour voir si elle satisfait un prédicat.

```
List(1, 2, 3, 4).forall(num => num > 0) // = true, all numbers are positive
List(-3, -2, -1, 1).forall(num => num < 0) // = false, not all numbers are negative
```

- Folds - `foldLeft` (`/:` `foldRight` , `foldRight` (`:\`), `reduceLeft` et `reduceRight` sont utilisés pour appliquer des fonctions binaires aux éléments successifs de la collection. Allez ici pour des exemples de pliage et allez ici pour réduire les exemples .

Plier

La méthode `fold` itération sur une collection, en utilisant une valeur d'accumulateur initiale et en appliquant une fonction qui utilise chaque élément pour mettre à jour l'accumulateur avec succès:

```
val nums = List(1,2,3,4,5)
var initialValue:Int = 0;
var sum = nums.fold(initialValue){
  (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 15 because 0+1+2+3+4+5 = 15
```

Dans l'exemple ci-dessus, une fonction anonyme a été fournie pour `fold()` . Vous pouvez également utiliser une fonction nommée qui prend deux arguments. En prenant ceci dans mon, l'exemple ci-dessus peut être ré-écrit ainsi:

```
def sum(x: Int, y: Int) = x+ y
val nums = List(1, 2, 3, 4, 5)
var initialValue: Int = 0
val sum = nums.fold(initialValue)(sum)
println(sum) // prints 15 because 0 + 1 + 2 + 3 + 4 + 5 = 15
```

Changer la valeur initiale affectera le résultat:

```
initialValue = 2;
sum = nums.fold(initialValue){
 (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 17 because 2+1+2+3+4+5 = 17
```

La méthode `fold` a deux variantes - `foldLeft` et `foldRight` .

`foldLeft()` itère de gauche à droite (du premier élément de la collection au dernier dans cet ordre). `foldRight()` itère de droite à gauche (du dernier élément au premier élément). `fold()` itère de gauche à droite comme `foldLeft()` . En fait, `fold()` appelle en réalité `foldLeft()` interne.

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
```

`fold()` , `foldLeft()` et `foldRight()` retourneront une valeur qui a le même type que la valeur initiale. Cependant, contrairement à `foldLeft()` et `foldRight()` , la valeur initiale donnée à `fold()` ne peut être que du même type ou d'un sur-type du type de la collection.

Dans cet exemple, l'ordre n'est pas pertinent, vous pouvez donc changer `fold()` en `foldLeft()` ou `foldRight()` et le résultat restera le même. L'utilisation d'une fonction sensible à l'ordre modifie les résultats.

En cas de doute, préférez `foldLeft()` sur `foldRight()` . `foldRight()` est moins performant.

Pour chaque

`foreach` est inhabituel parmi les itérateurs de collections en ce sens qu'il ne renvoie pas de résultat. Au lieu de cela, il applique une fonction à chaque élément qui n'a que des effets secondaires. Par exemple:

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x.foreach { println }
1
2
3
```

La fonction fournie à `foreach` peut avoir n'importe quel type de retour, mais le résultat sera ignoré . Typiquement, `foreach` est utilisé lorsque les effets secondaires sont souhaitables. Si vous souhaitez transformer des données, utilisez une `map` , un `filter` , une `for comprehension` ou une autre option.

Exemple de rejet des résultats

```
def myFunc(a: Int) : Int = a * 2
List(1,2,3).foreach(myFunc) // Returns nothing
```

Réduire

Les méthodes `reduceRight reduce()` , `reduceLeft()` et `reduceRight` sont similaires aux plis. La fonction passée à réduire prend deux valeurs et en génère une troisième. Lorsque vous utilisez une liste, les deux premières valeurs sont les deux premières valeurs de la liste. Le résultat de la fonction et la valeur suivante dans la liste sont ensuite réappliqués à la fonction, produisant un nouveau résultat. Ce nouveau résultat est appliqué avec la valeur suivante de la liste et ainsi de suite jusqu'à ce qu'il n'y ait plus d'éléments. Le résultat final est renvoyé.

```
val nums = List(1,2,3,4,5)
sum = nums.reduce({ (a, b) => a + b })
println(sum) //prints 15

val names = List("John","Koby", "Josh", "Matilda", "Zac", "Mary Poppins")

def findLongest(nameA:String, nameB:String):String = {
  if (nameA.length > nameB.length) nameA else nameB
}

def findLastAlphabetically(nameA:String, nameB:String):String = {
  if (nameA > nameB) nameA else nameB
}

val longestName:String = names.reduce(findLongest(_,_))
println(longestName) //prints Mary Poppins

//You can also omit the arguments if you want
val lastAlphabetically:String = names.reduce(findLastAlphabetically)
println(lastAlphabetically) //prints Zac
```

Il existe des différences dans le fonctionnement des fonctions de réduction par rapport aux fonctions de pliage. Elles sont:

1. Les fonctions de réduction n'ont pas de valeur d'accumulateur initiale.
2. Les fonctions de réduction ne peuvent pas être appelées sur des listes vides.
3. Les fonctions Réduire ne peuvent que renvoyer le type ou le sur-type de la liste.

Lire Collections en ligne: https://riptutorial.com/fr/scala/topic/686/collections

Chapitre 9: Collections parallèles

Remarques

Les collections parallèles facilitent la programmation parallèle en masquant les détails de la parallélisation de bas niveau. Cela facilite l'utilisation des architectures multicœurs. `ParArray` , `ParVector` , `mutable.ParHashMap` , `immutable.ParHashMap` et `ParRange` exemples de collections parallèles. Une liste complète peut être trouvée dans la documentation .

Examples

Créer et utiliser des collections parallèles

Pour créer une collection parallèle à partir d'une collection séquentielle, appelez la méthode `par` . Pour créer une collection séquentielle à partir d'une collection parallèle, appelez la méthode `seq` . Cet exemple montre comment transformer un `Vector` normal en un `ParVector` , puis à nouveau:

```
scala> val vect = (1 to 5).toVector
vect: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> val parVect = vect.par
parVect: scala.collection.parallel.immutable.ParVector[Int] = ParVector(1, 2, 3, 4, 5)

scala> parVect.seq
res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
```

La méthode `par` peut être chaînée, ce qui vous permet de convertir une collection séquentielle en une collection parallèle et d'y effectuer immédiatement une action:

```
scala> vect.map(_ * 2)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 4, 6, 8, 10)

scala> vect.par.map(_ * 2)
res2: scala.collection.parallel.immutable.ParVector[Int] = ParVector(2, 4, 6, 8, 10)
```

Dans ces exemples, le travail est en fait distribué à plusieurs unités de traitement, puis joint à nouveau une fois le travail terminé, sans intervention du développeur.

Pièges

N'utilisez pas de collections parallèles lorsque les éléments de la collection doivent être reçus dans un ordre spécifique.

Les collections parallèles effectuent des opérations simultanément. Cela signifie que tout le travail est divisé en plusieurs parties et distribué à différents processeurs. Chaque processeur n'est pas au courant du travail effectué par d'autres. Si l' *ordre de la collecte est* important, alors le travail traité en parallèle n'est pas déterministe. (Exécuter deux fois le même code peut donner des

résultats différents.)

Opérations non associatives

Si une opération est non associative (si l'ordre d'exécution est important), le résultat sur une collection parallélisée sera non déterministe.

```
scala> val list = (1 to 1000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10...

scala> list.reduce(_ - _)
res0: Int = -500498

scala> list.reduce(_ - _)
res1: Int = -500498

scala> list.reduce(_ - _)
res2: Int = -500498

scala> val listPar = list.par
listPar: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8,
9, 10...

scala> listPar.reduce(_ - _)
res3: Int = -408314

scala> listPar.reduce(_ - _)
res4: Int = -422884

scala> listPar.reduce(_ - _)
res5: Int = -301748
```

Effets secondaires

Les opérations qui ont des effets secondaires, tels que `foreach` , peuvent ne pas s'exécuter comme souhaité sur des collections parallélisées en raison de conditions de concurrence. Évitez cela en utilisant des fonctions sans effets secondaires, telles que `reduce` ou `map` .

```
scala> val wittyOneLiner = Array("Artificial", "Intelligence", "is", "no", "match", "for",
"natural", "stupidity")

scala> wittyOneLiner.foreach(word => print(word + " "))
Artificial Intelligence is no match for natural stupidity

scala> wittyOneLiner.par.foreach(word => print(word + " "))
match natural is for Artificial no stupidity Intelligence

scala> print(wittyOneLiner.par.reduce{_ + " " + _})
Artificial Intelligence is no match for natural stupidity

scala> val list = (1 to 100).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...
```

Lire Collections parallèles en ligne: https://riptutorial.com/fr/scala/topic/3882/collections-paralleles

Chapitre 10: Combinateurs d'analyseur

Remarques

Cas ParseResult

Un `ParseResult` décline en trois `ParseResult` :

- Succès, avec un marqueur pour le début du match et le prochain personnage à rechercher.
- Échec, avec un marqueur indiquant le début de la tentative de match. Dans ce cas, l'analyseur revient à cette position, où il se trouvera lorsque l'analyse se poursuivra.
- Erreur qui arrête l'analyse. Aucun retour en arrière ou analyse supplémentaire ne se produit.

Examples

Exemple de base

```
import scala.util.parsing.combinator._

class SimpleParser extends RegexParsers {
  // Define a grammar rule, turn it into a regex, and apply it the input.
  def word: Parser[String] = """[A-Z][a-z]+""".r ^^ { _.toString }
}

object SimpleParser extends SimpleParser {
  val parseAlice = parse(word, "Alice went to Alamo Square.")
  val parseBarb = parse(word, "barb went Upside Down.")
}

//Successfully finds a match
println(SimpleParser.parseAlice)
//Fails to find a match
println(SimpleParser.parseBarb)
```

La sortie sera la suivante:

```
[1.6] parsed: Alice
res0: Unit = ()

[1.1] failure: string matching regex `[A-Z][a-z]+' expected but `b' found

barb went Upside Down.
^
```

`[1.6]` dans l'exemple d' `Alice` indique que le début du match est à la position `1` et que le premier caractère restant à faire correspondre commence à la position `6` .

Lire Combinateurs d'analyseur en ligne: https://riptutorial.com/fr/scala/topic/3730/combinateurs-d-analyseur

Chapitre 11: Correspondance de motif

Syntaxe

- sélecteur match partielleFonction
- match de sélection {liste des alternatives à la casse) // C'est la forme la plus courante de ce qui précède

Paramètres

Paramètre	Détails
sélecteur	L'expression dont la valeur correspond à un motif.
des alternatives	une liste d'alternatives délimitées par des `case` .

Examples

Match simple

Cet exemple montre comment faire correspondre une entrée à plusieurs valeurs:

```
def f(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Unknown!"
}

f(2)  // "Two"
f(3)  // "Unknown!"
```

Démo en direct

Note: _ est la *chute* ou *par* cas par *défaut,* mais il est pas nécessaire.

```
def g(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
}

g(1)  // "One"
g(3)  // throws a MatchError
```

Pour éviter de lancer une exception, il est préférable de gérer le cas par défaut (`case _ => <do something>`). Notez que la correspondance sur *une classe de cas* peut aider le compilateur à générer un avertissement si une requête est manquante. Il en va de même pour les types définis par l'utilisateur qui étendent un trait scellé. Si la correspondance est totale, un cas par défaut peut

ne pas être nécessaire

Il est également possible de faire correspondre les valeurs qui ne sont pas définies en ligne. Celles-ci doivent être *des identifiants stables* , obtenus soit en utilisant un nom en majuscule, soit en entourant les backticks.

Avec `One` et `two` définis ailleurs, ou passés en tant que paramètres de fonction:

```
val One: Int = 1
val two: Int = 2
```

Ils peuvent être comparés de la manière suivante:

```
def g(x: Int): String = x match {
  case One => "One"
  case `two` => "Two"
}
```

Contrairement à d'autres langages de programmation comme Java par exemple, il n'y a pas de chute. Si un bloc de requête correspond à une entrée, il est exécuté et la correspondance est terminée. Par conséquent, le cas le moins spécifique devrait être le dernier cas.

```
def f(x: Int): String = x match {
  case _ => "Default"
  case 1 => "One"
}

f(5) // "Default"
f(1) // "Default"
```

Correspondance de modèle avec un identifiant stable

Dans la correspondance de modèle standard, l'identificateur utilisé masquera tout identifiant dans la portée englobante. Parfois, il est nécessaire de faire correspondre la variable de l'étendue englobante.

L'exemple de fonction suivant prend un caractère et une liste de tuples et renvoie une nouvelle liste de tuples. Si le caractère existait en tant que premier élément dans l'un des n-uplets, le deuxième élément est incrémenté. S'il n'existe pas encore dans la liste, un nouveau tuple est créé.

```
def tabulate(char: Char, tab: List[(Char, Int)]): List[(Char, Int)] = tab match {
  case Nil => List((char, 1))
  case (`char`, count) :: tail => (char, count + 1) :: tail
  case head :: tail => head :: tabulate(char, tail)
}
```

L'illustration ci-dessus montre une correspondance de modèle où l'entrée de la méthode, `char` , est conservée 'stable' dans la correspondance de modèle: si vous appelez `tabulate('x', ...)` , la première instruction sera interprétée comme:

```
case('x', count) => ...
```

Scala interprétera toute variable délimitée par une coche comme identifiant stable: elle interprétera également toute variable commençant par une majuscule de la même manière.

Correspondance de motif sur une Seq

Pour rechercher un nombre précis d'éléments dans la collection

```
def f(ints: Seq[Int]): String = ints match {
  case Seq() =>
      "The Seq is empty !"
  case Seq(first) =>
      s"The seq has exactly one element : $first"
  case Seq(first, second) =>
      s"The seq has exactly two elements : $first, $second"
  case  s @ Seq(_, _, _) =>
      s"s is a Seq of length three and looks like ${s}"  // Note individual elements are not
bound to their own names.
  case s: Seq[Int] if s.length == 4 =>
      s"s is a Seq of Ints of exactly length 4"  // Again, individual elements are not bound
to their own names.
  case _ =>
      "No match was found!"
}
```

Démo en direct

Pour extraire le ou les premier (s) élément (s) et conserver le reste en tant que collection:

```
def f(ints: Seq[Int]): String = ints match {
  case Seq(first, second, tail @ _*) =>
      s"The seq has at least two elements : $first, $second. The rest of the Seq is $tail"
  case Seq(first, tail @ _*) =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  // alternative syntax
  // here of course this one will never match since it checks
  // for the same thing as the one above
  case first +: tail =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  case _ =>
      "The seq didn't match any of the above, so it must be empty"
}
```

En général, toute forme pouvant être utilisée pour construire une séquence peut être utilisée pour établir une correspondance avec une séquence existante.

Notez que lorsqu'on utilise `Nil` et `::` fonctionnera lorsque le motif correspond à une séquence, il le convertit en une `List` et peut avoir des résultats inattendus. Limitez-vous à `Seq( ...)` et à `+:` pour éviter cela.

Notez que lorsque vous utilisez `::` ne fonctionnera pas pour `WrappedArray` , `Vector` etc, consultez:

```
scala> def f(ints:Seq[Int]) = ints match {
```

```
     | case h :: t => h
     | case _ => "No match"
     | }
f: (ints: Seq[Int])Any

scala> f(Array(1,2))
res0: Any = No match
```

Et avec `+:` :

```
scala> def g(ints:Seq[Int]) = ints match {
     | case h+:t => h
     | case _ => "No match"
     | }
g: (ints: Seq[Int])Any

scala> g(Array(1,2).toSeq)
res4: Any = 1
```

Gardes (si expressions)

Les instructions de cas peuvent être combinées avec des expressions if pour fournir une logique supplémentaire lors de la correspondance de modèle.

```
def checkSign(x: Int): String = {
    x match {
      case a if a < 0 => s"$a is a negative number"
      case b if b > 0 => s"$b is a positive number"
      case c => s"$c neither positive nor negative"
    }
}
```

Il est important de vous assurer que vos gardes ne créent pas une correspondance non exhaustive (le compilateur ne détectera souvent pas cela):

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case None    => doSomethingIfNone
}
```

Cela jette un `MatchError` sur les nombres impairs. Vous devez soit prendre en compte tous les cas, soit utiliser une casse générique:

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case _ => doSomethingIfNoneOrOdd
}
```

Correspondance de motifs avec les classes de cas

Chaque classe de cas définit un extracteur qui peut être utilisé pour capturer les membres de la classe de cas lorsque la correspondance de modèle:

```
case class Student(name: String, email: String)

def matchStudent1(student: Student): String = student match {
    case Student(name, email) => s"$name has the following email: $email" // extract name and
email
}
```

Toutes les règles normales de correspondance de modèle s'appliquent - vous pouvez utiliser des gardes et des expressions constantes pour contrôler la correspondance:

```
def matchStudent2(student: Student): String = student match {
    case Student("Paul", _) => "Matched Paul" // Only match students named Paul, ignore email
    case Student(name, _) if name == "Paul" => "Matched Paul" // Use a guard to match students
named Paul, ignore email
    case s if s.name == "Paul" => "Matched Paul" // Don't use extractor; use a guard to match
students named Paul, ignore email
    case Student("Joe", email) => s"Joe has email $email" // Match students named Joe, capture
their email
    case Student(name, email) if name == "Joe" => s"Joe has email $email" // use a guard to
match students named Joe, capture their email
    case Student(name, email) => s"$name has email $email." // Match all students, capture
name and email
}
```

Correspondance sur une option

Si vous correspondez à un type d' option :

```
def f(x: Option[Int]) = x match {
    case Some(i) => doSomething(i)
    case None    => doSomethingIfNone
}
```

C'est fonctionnellement équivalent à utiliser `fold` , ou `map` / `getOrElse` :

```
def g(x: Option[Int]) = x.fold(doSomethingIfNone)(doSomething)
def h(x: Option[Int]) = x.map(doSomething).getOrElse(doSomethingIfNone)
```

Traitements scellés assortis

Lorsque le motif correspondant à un objet dont le type est un trait scellé, Scala vérifiera au moment de la compilation que tous les cas sont «appariés de manière exhaustive»:

```
sealed trait Shape
case class Square(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape


def matchShape(shape: Shape): String = shape match {
    case Square(height, width) => "It's a square"
    case Circle(radius)        => "It's a circle"
    //no case for Point because it would cause a compiler warning.
```

```
}
```

Si une nouvelle `case class` pour `Shape` est ajoutée ultérieurement, toutes les instructions de `match` sur `Shape` commenceront à émettre un avertissement de compilation. Cela facilite le refactoring complet: le compilateur alertera le développeur de tout le code qui doit être mis à jour.

Correspondance de motifs avec Regex

```
val emailRegex: Regex = "(.+)@(.+)\\.(.+)".r

"name@example.com" match {
  case emailRegex(userName, domain, topDomain) => println(s"Hi $userName from $domain")
  case _ => println(s"This is not a valid email.")
}
```

Dans cet exemple, l'expression régulière tente de correspondre à l'adresse électronique fournie. Si c'est le cas, le nom d' `userName` et le `domain` sont extraits et imprimés. `topDomain` est également extrait, mais rien n'est fait dans cet exemple. L'appel de `.r` sur un String `str` est équivalent à `new Regex(str)` . La fonction `r` est disponible via une conversion implicite .

Reliure de motif (@)

Le signe `@` lie une variable à un nom lors d'une correspondance de modèle. La variable liée peut être soit l'objet correspondant entier, soit une partie de l'objet correspondant:

```
sealed trait Shape
case class Rectangle(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape

(Circle(5): Shape) match {
  case Rectangle(h, w) => s"rectangle, $h x $w."
  case Circle(r) if r > 9 => s"large circle"
  case c @ Circle(_) => s"small circle: ${c.radius}"  // Whole matched object is bound to c
  case Point => "point"
}
```

```
> res0: String = small circle: 5
```

L'identifiant lié peut être utilisé dans des filtres conditionnels. Ainsi:

```
case Circle(r) if r > 9 => s"large circle"
```

peut être écrit comme:

```
case c @ Circle(_) if c.radius > 9 => s"large circle"
```

Le nom ne peut être lié qu'à une partie du motif correspondant:

```
Seq(Some(1), Some(2), None) match {
  // Only the first element of the matched sequence is bound to the name 'c'
```

```
  case Seq(c @ Some(1), _*) => head
  case _ => None
}
```

```
> res0: Option[Int] = Some(1)
```

Types de correspondance

La correspondance de modèle peut également être utilisée pour vérifier le type d'une instance, plutôt que d'utiliser `isInstanceOf[B]` :

```
val anyRef: AnyRef = ""

anyRef match {
  case _: Number       => "It is a number"
  case _: String       => "It is a string"
  case _: CharSequence => "It is a char sequence"
}
//> res0: String = It is a string
```

L'ordre des cas est important:

```
anyRef match {
  case _: Number       => "It is a number"
  case _: CharSequence => "It is a char sequence"
  case _: String       => "It is a string"
}
//> res1: String = It is a char sequence
```

De cette manière, il est similaire à une instruction "switch" classique, sans la fonctionnalité fall-through. Cependant, vous pouvez également créer des correspondances et extraire des valeurs du type en question. Par exemple:

```
case class Foo(s: String)
case class Bar(s: String)
case class Woo(s: String, i: Int)

def matcher(g: Any):String = {
  g match {
    case Bar(s) => s + " is classy!"
    case Foo(_) => "Someone is wicked smart!"
    case Woo(s, _) => s + " is adventerous!"
    case _ => "What are we talking about?"
  }
}

print(matcher(Foo("Diana")))  // prints 'Diana is classy!'
print(matcher(Bar("Hadas")))  // prints 'Someone is wicked smart!'
print(matcher(Woo("Beth", 27)))   // prints 'Beth is adventerous!'
print(matcher(Option("Katie")))  // prints 'What are we talking about?'
```

Notez que dans les cas `Foo` et `Woo` , nous utilisons le trait de soulignement (`_`) pour "correspondre à une variable non liée". C'est-à-dire que la valeur (dans ce cas, `Hadas` et `27` , respectivement) n'est pas liée à un nom et n'est donc pas disponible dans le gestionnaire pour ce cas. Ceci est un

raccourci utile pour faire correspondre «n'importe quelle valeur» sans se soucier de sa valeur.

Correspondance de modèle compilée en tant que commutateur de table ou commutateur de recherche

L'annotation `@switch` indique au compilateur que l'instruction de `match` peut être remplacée par une seule instruction de `tableswitch` au niveau du bytecode. Il s'agit d'une optimisation mineure qui peut supprimer les comparaisons inutiles et les charges variables pendant l'exécution.

L'annotation `@switch` ne fonctionne que pour les correspondances avec les constantes littérales et les identificateurs de `final val` . Si la correspondance de modèle ne peut pas être compilée en tant que `tableswitch` / `lookupswitch` , le compilateur `lookupswitch` un avertissement.

```
import annotation.switch

def suffix(i: Int) = (i: @switch) match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

Les résultats sont identiques à ceux d'un modèle normal:

```
scala> suffix(2)
res1: String = "2nd"

scala> suffix(4)
res2: String = "4th"
```

De la documentation Scala (2.8+) - `@switch` :

> Une annotation à appliquer à une expression de correspondance. S'il est présent, le compilateur vérifiera que la correspondance a été compilée avec un commutateur de tables ou un commutateur de recherche et génère une erreur si elle compile au lieu de cela en une série d'expressions conditionnelles.

De la spécification Java:

- switch : "Accéder à la table de saut par index et saut"
- lookupswitch : "Accéder à la table de saut par match de la clé et sauter"

Faire correspondre plusieurs modèles à la fois

Le `|` peut être utilisé pour avoir une seule déclaration de cas avec plusieurs entrées pour obtenir le même résultat:

```
def f(str: String): String = str match {
  case "foo" | "bar" => "Matched!"
  case _ => "No match."
```

```
}

f("foo")  // res0: String = Matched!
f("bar")  // res1: String = Matched!
f("fubar")  // res2: String = No match.
```

Notez que, bien que les **valeurs** correspondent de cette manière, la correspondance des **types suivants** posera des problèmes:

```
sealed class FooBar
case class Foo(s: String) extends FooBar
case class Bar(s: String) extends FooBar

val d = Foo("Diana")
val h = Bar("Hadas")

// This matcher WILL NOT work.
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) | Bar(s) => print(s)  // Won't work: s cannot be resolved
    case Foo(_) | Bar(_) => _         // Won't work: _ is an unbound placeholder
    case _ => "Could not match"
  }
}
```

Si dans le dernier cas (avec _) vous n'avez pas besoin de la valeur de la variable non liée et que vous voulez juste faire autre chose, vous allez bien:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(_) | Bar(_) => "Is either Foo or Bar."  // Works fine
    case _ => "Could not match"
  }
}
```

Sinon, il vous reste à diviser vos affaires:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) => s
    case Bar(s) => s
    case _ => "Could not match"
  }
}
```

Correspondance des motifs sur les tuples

Étant donné la `List` suivante de tuples:

```
val pastries = List(("Chocolate Cupcake", 2.50),
                    ("Vanilla Cupcake", 2.25),
                    ("Plain Muffin", 3.25))
```

La correspondance de modèle peut être utilisée pour gérer chaque élément différemment:

```
pastries foreach { pastry =>
  pastry match {
    case ("Plain Muffin", price) => println(s"Buying muffin for $price")
    case p if p._1 contains "Cupcake" => println(s"Buying cupcake for ${p._2}")
    case _ => println("We don't sell that pastry")
  }
}
```

Le premier cas montre comment faire correspondre une chaîne spécifique et obtenir le prix correspondant. Le deuxième cas montre une utilisation de l' extraction if et tuple pour correspondre aux éléments du tuple.

Lire Correspondance de motif en ligne: https://riptutorial.com/fr/scala/topic/661/correspondance-de-motif

Chapitre 12: Currying

Syntaxe

- aFunction (10) _ // Utilisation de '_' Indique au compilateur que tous les paramètres des autres groupes de paramètres seront curry.
- nArityFunction.curried // Convertit une fonction n-arity en une version curry équivalente
- anotherFunction (x) (_: String) (z) // currying un paramètre arbitraire. Son type doit être explicitement indiqué.

Examples

Un multiplicateur configurable en tant que fonction curry

```
def multiply(factor: Int)(numberToBeMultiplied: Int): Int = factor * numberToBeMultiplied

val multiplyBy3 = multiply(3)_     // resulting function signature Int => Int
val multiplyBy10 = multiply(10)_ // resulting function signature Int => Int

val sixFromCurriedCall = multiplyBy3(2) //6
val sixFromFullCall = multiply(3)(2)    //6

val fortyFromCurriedCall = multiplyBy10(4) //40
val fortyFromFullCall = multiply(10)(4)    //40
```

Plusieurs groupes de paramètres de différents types, paramètres de curry des positions arbitraires

```
def numberOrCharacterSwitch(toggleNumber: Boolean)(number: Int)(character: Char): String =
  if (toggleNumber) number.toString else character.toString

// need to explicitly specify the type of the parameter to be curried
// resulting function signature Boolean => String
val switchBetween3AndE = numberOrCharacterSwitch(_: Boolean)(3)('E')

switchBetween3AndE(true) // "3"
switchBetween3AndE(false) // "E"
```

Currying une fonction avec un seul groupe de paramètres

```
def minus(left: Int, right: Int) = left - right

val numberMinus5 = minus(_: Int, 5)
val fiveMinusNumber = minus(5, _: Int)

numberMinus5(7)    //  2
fiveMinusNumber(7) // -2
```

Currying

Définissons une fonction de 2 arguments:

```
def add: (Int, Int) => Int = (x,y) => x + y
val three = add(1,2)
```

Le curry `add` transforme en une fonction qui prend **un** `Int` et retourne une **fonction** (d' **un** `Int` à un `Int`)

```
val addCurried: (Int) => (Int => Int) = add2.curried
//                ^~~ take *one* Int
//                          ^~~~ return a *function* from Int to Int

val add1: Int => Int = addCurried(1)
val three: Int = add1(2)
val allInOneGo: Int = addCurried(1)(2)
```

Vous pouvez appliquer ce concept à toute fonction qui prend plusieurs arguments. Le curry d'une fonction qui prend plusieurs arguments, le transforme en une série d'applications de fonctions qui prennent **un** argument:

```
def add3: (Int, Int, Int) => Int = (a,b,c) => a + b + c + d
def add3Curr: Int => (Int => (Int => Int)) = add3.curried

val x = add3Curr(1)(2)(42)
```

Currying

Currying, selon Wikipedia ,

> est la technique de traduction de l'évaluation d'une fonction qui prend plusieurs arguments pour évaluer une séquence de fonctions.

Concrètement, en termes de types de scala, dans le contexte d'une fonction qui prend deux arguments, (a arité 2), c'est la conversion de

```
val f: (A, B) => C // a function that takes two arguments of type `A` and `B` respectively
                   // and returns a value of type `C`
```

à

```
val curriedF: A => B => C // a function that take an argument of type `A`
                          // and returns *a function*
                          // that takes an argument of type `B` and returns a `C`
```

Donc, pour les fonctions arity-2, nous pouvons écrire la fonction curry comme:

```
def curry[A, B, C](f: (A, B) => C): A => B => C = {
  (a: A) => (b: B) => f(a, b)
```

```
}
```

Usage:

```
val f: (String, Int) => Double = {(_, _) => 1.0}
val curriedF: String => Int => Double = curry(f)
f("a", 1)        // => 1.0
curriedF("a")(1) // => 1.0
```

Scala nous offre quelques fonctionnalités linguistiques qui facilitent ceci:

1. Vous pouvez écrire des fonctions au curry en tant que méthodes. Donc `curriedF` peut être écrit comme `curriedF` :

```
def curriedFAsAMethod(str: String)(int: Int): Double = 1.0
val curriedF = curriedFAsAMethod _
```

2. Vous pouvez annuler le curry (c'est-à-dire passer de `A => B => C` à `(A, B) => C`) en utilisant une méthode de bibliothèque standard: `Function.uncurried`

```
val f: (String, Int) => Double = Function.uncurried(curriedF)
f("a", 1) // => 1.0
```

Quand utiliser le curry

Le curry *est la technique de traduction de l'évaluation d'une fonction qui prend plusieurs arguments pour évaluer une séquence de fonctions, chacune avec un seul argument* .

Ceci est normalement utile lorsque, par exemple:

1. différents arguments d'une fonction sont calculés **à des moments différents** . *(Exemple 1)*
2. différents arguments d'une fonction sont calculés **par différents niveaux de l'application** . *(Exemple 2)*

Exemple 1

Supposons que le revenu annuel total est une fonction composée du revenu et d'une prime:

```
val totalYearlyIncome:(Int,Int) => Int =  (income, bonus) => income + bonus
```

La version curry de la fonction 2-arity ci-dessus est:

```
val totalYearlyIncomeCurried: Int => Int => Int = totalYearlyIncome.curried
```

Notez dans la définition ci-dessus que le type peut également être vu / écrit comme suit:

```
Int => (Int => Int)
```

Supposons que la partie du revenu annuel soit connue à l'avance:

```
val partialTotalYearlyIncome: Int => Int = totalYearlyIncomeCurried(10000)
```

Et à un moment donné, le bonus est connu:

```
partialTotalYearlyIncome(100)
```

Exemple 2

Supposons que la fabrication de voitures implique l'application de roues de voiture et de carrosserie:

```
val carManufacturing:(String,String) => String = (wheels, body) => wheels + body
```

Ces pièces sont appliquées par différentes usines:

```
class CarWheelsFactory {
  def applyCarWheels(carManufacturing:(String,String) => String): String => String =
          carManufacturing.curried("applied wheels..")
}

class CarBodyFactory {
  def applyCarBody(partialCarWithWheels: String => String): String =
partialCarWithWheels("applied car body..")
}
```

Remarquez que la `CarWheelsFactory` ci-dessus `CarWheelsFactory` la fonction de fabrication de la voiture et applique uniquement les roues.

Le processus de fabrication de la voiture prendra alors la forme ci-dessous:

```
val carWheelsFactory = new CarWheelsFactory()
val carBodyFactory   = new CarBodyFactory()

val carManufacturing:(String,String) => String = (wheels, body) => wheels + body

val partialCarWheelsApplied: String => String  =
carWheelsFactory.applyCarWheels(carManufacturing)
val carCompleted = carBodyFactory.applyCarBody(partialCarWheelsApplied)
```

Une utilisation réelle du curry.

Nous avons une liste de cartes de crédit et nous aimerions calculer les primes pour toutes les cartes que la compagnie de carte de crédit doit payer. Les primes elles-mêmes dépendent du nombre total de cartes de crédit, de sorte que l'entreprise les ajuste en conséquence.

Nous avons déjà une fonction qui calcule la prime pour une seule carte de crédit et prend en compte le nombre total de cartes émises par l'entreprise:

```
case class CreditCard(creditInfo: CreditCardInfo, issuer: Person, account: Account)

object CreditCard {
  def getPremium(totalCards: Int, creditCard: CreditCard): Double = { ... }
}
```

Maintenant, une approche raisonnable de ce problème consisterait à mapper chaque carte de crédit sur une prime et à la réduire à une somme. Quelque chose comme ça:

```
val creditCards: List[CreditCard] = getCreditCards()
val allPremiums = creditCards.map(CreditCard.getPremium).sum //type mismatch; found : (Int,
CreditCard) ⇒ Double required: CreditCard ⇒ ?
```

Cependant, le compilateur ne va pas aimer cela, car `CreditCard.getPremium` nécessite deux paramètres. Application partielle à la rescousse! Nous pouvons appliquer partiellement le nombre total de cartes de crédit et utiliser cette fonction pour associer les cartes de crédit à leurs primes. Tout ce que nous devons faire est de curry la fonction `getPremium` en la changeant pour utiliser plusieurs listes de paramètres et nous sommes `getPremium` à aller.

Le résultat devrait ressembler à ceci:

```
object CreditCard {
  def getPremium(totalCards: Int)(creditCard: CreditCard): Double = { ... }
}

val creditCards: List[CreditCard] = getCreditCards()

val getPremiumWithTotal = CreditCard.getPremium(creditCards.length)_

val allPremiums = creditCards.map(getPremiumWithTotal).sum
```

Lire Currying en ligne: https://riptutorial.com/fr/scala/topic/1636/currying

Chapitre 13: Énumérations

Remarques

Une approche avec un `sealed trait` et des `case objects` est préférable car l'énumération Scala pose quelques problèmes:

1. Les énumérations ont le même type après l'effacement.
2. Le compilateur ne se plaint pas du fait que "Match n'est pas exhaustif", si un cas est manqué, il échouera lors de l'exécution `scala.MatchError` :

```
def isWeekendWithBug(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sun | WeekDays.Sat => true
}

isWeekendWithBug(WeekDays.Fri)
scala.MatchError: Fri (of class scala.Enumeration$Val)
```

Comparer avec:

```
def isWeekendWithBug(day: WeekDay): Boolean = day match {
  case WeekDay.Sun | WeekDay.Sat => true
}

Warning: match may not be exhaustive.
It would fail on the following inputs: Fri, Mon, Thu, Tue, Wed
def isWeekendWithBug(day: WeekDay): Boolean = day match {
                                              ^
```

Une explication plus détaillée est présentée dans cet article sur l'énumération Scala .

Examples

Jours de la semaine en utilisant l'énumération Scala

Les énumérations de type Java peuvent être créées en étendant l' énumération .

```
object WeekDays extends Enumeration {
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

def isWeekend(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sat | WeekDays.Sun => true
  case _ => false
}

isWeekend(WeekDays.Sun)
res0: Boolean = true
```

Il est également possible d'ajouter un nom lisible par l'homme pour les valeurs dans une

énumération:

```
object WeekDays extends Enumeration {
      val Mon = Value("Monday")
      val Tue = Value("Tuesday")
      val Wed = Value("Wednesday")
      val Thu = Value("Thursday")
      val Fri = Value("Friday")
      val Sat = Value("Saturday")
      val Sun = Value("Sunday")
}

println(WeekDays.Mon)
>> Monday

WeekDays.withName("Monday") == WeekDays.Mon
>> res0: Boolean = true
```

Méfiez-vous du comportement peu fiable, dans lequel différentes énumérations peuvent être évaluées comme le même type d'instance:

```
object Parity extends Enumeration {
   val Even, Odd = Value
}

WeekDays.Mon.isInstanceOf[Parity.Value]
>> res1: Boolean = true
```

Utiliser des objets et des traits scellés

Une alternative à l'extension de l' `Enumeration` consiste à utiliser `sealed` objets de boîtier `sealed` :

```
sealed trait WeekDay

object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
}
```

Le mot-clé `sealed` garantit que le trait `WeekDay` ne peut pas être étendu dans un autre fichier. Cela permet au compilateur de faire certaines hypothèses, y compris que toutes les valeurs possibles de `WeekDay` sont déjà énumérées.

Un inconvénient est que cette méthode ne vous permet pas d'obtenir une liste de toutes les valeurs possibles. Pour obtenir une telle liste, elle doit être fournie explicitement:

```
val allWeekDays = Seq(Mon, Tue, Wed, Thu, Fri, Sun, Sat)
```

Les classes de cas peuvent également étendre un trait `sealed` . Ainsi, les objets et les classes de

cas peuvent être mélangés pour créer des hiérarchies complexes:

```
sealed trait CelestialBody

object CelestialBody {
  case object Earth extends CelestialBody
  case object Sun extends CelestialBody
  case object Moon extends CelestialBody
  case class Asteroid(name: String) extends CelestialBody
}
```

Un autre inconvénient est qu'il n'y a aucun moyen d'accéder au nom de la variable d'une énumération d'objet `sealed` , ou de la rechercher. Si vous avez besoin d'une sorte de nom associé à chaque valeur, celle-ci doit être définie manuellement:

```
  sealed trait WeekDay { val name: String }

  object WeekDay {
      case object Mon extends WeekDay { val name = "Monday" }
      case object Tue extends WeekDay { val name = "Tuesday" }
      (...)
  }
```

Ou juste:

```
  sealed case class WeekDay(name: String)

  object WeekDay {
      object Mon extends WeekDay("Monday")
      object Tue extends WeekDay("Tuesday")
      (...)
  }
```

Utilisation d'objets trait et de cas scellés et allValues-macro

Ceci est juste une extension sur la variante de trait scellée où une macro génère un ensemble avec toutes les instances au moment de la compilation. Cela omet bien l'inconvénient qu'un développeur peut ajouter une valeur à l'énumération mais oublier de l'ajouter à l'ensemble allElements.

Cette variante devient particulièrement pratique pour les grandes énumérations.

```
import EnumerationMacros._

sealed trait WeekDay
object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
  val allWeekDays: Set[WeekDay] = sealedInstancesOf[WeekDay]
```

```
}
```

Pour que cela fonctionne, vous avez besoin de cette macro:

```
import scala.collection.immutable.TreeSet
import scala.language.experimental.macros
import scala.reflect.macros.blackbox

/**
A macro to produce a TreeSet of all instances of a sealed trait.
Based on Travis Brown's work:
http://stackoverflow.com/questions/13671734/iteration-over-a-sealed-trait-in-scala
CAREFUL: !!! MUST be used at END OF code block containing the instances !!!
*/
object EnumerationMacros {
  def sealedInstancesOf[A]: TreeSet[A] = macro sealedInstancesOf_impl[A]

  def sealedInstancesOf_impl[A: c.WeakTypeTag](c: blackbox.Context) = {
    import c.universe._

    val symbol = weakTypeOf[A].typeSymbol.asClass

    if  (!symbol.isClass || !symbol.isSealed)
      c.abort(c.enclosingPosition, "Can only enumerate values of a sealed trait or class.")
    else {

      val children = symbol.knownDirectSubclasses.toList

      if (!children.forall(_.isModuleClass)) c.abort(c.enclosingPosition, "All children must
be objects.")
      else c.Expr[TreeSet[A]] {

        def sourceModuleRef(sym: Symbol) =
Ident(sym.asInstanceOf[scala.reflect.internal.Symbols#Symbol
          ].sourceModule.asInstanceOf[Symbol]
        )

        Apply(
          Select(
            reify(TreeSet).tree,
            TermName("apply")
          ),
          children.map(sourceModuleRef(_))
        )
      }
    }
  }
}
```

Lire Énumérations en ligne: https://riptutorial.com/fr/scala/topic/1499/enumerations

Chapitre 14: Expressions régulières

Syntaxe

- re.findAllIn (s: CharSequence): MatchIterator
- re.findAllMatchIn (s: CharSequence): Iterator [Match]
- re.findFirstIn (s: CharSequence): Option [Chaîne]
- re.findFirstMatchIn (s: CharSequence): Option [Match]
- re.findPrefixMatchIn (s: CharSequence): Option [Match]
- re.findPrefixOf (s: CharSequence): Option [Chaîne]
- re.replaceAllIn (s: CharSequence, remplaçant: Match => String): String
- re.replaceAllIn (s: CharSequence, remplacement: String): chaîne
- re.replaceFirstIn (s: CharSequence, replacement: String): String
- re.replaceSomeIn (s: CharSequence, remplacement: Match => Option [String]): String
- re.split (s: CharSequence): Array [String]

Examples

Déclaration d'expressions régulières

La méthode `r` fournie implicitement via scala.collection.immutable.StringOps génère une instance de scala.util.matching.Regex à partir de la chaîne objet. La syntaxe de chaîne à trois guillemets de Scala est utile ici, car vous n'avez pas à échapper à des barres obliques inverses comme vous le feriez en Java:

```
val r0: Regex = """(\d{4})-(\d${2})-(\d{2})""".r     // :)
val r1: Regex = "(\\d{4})-(\\d{2})-(\\d{2})".r // :(
```

scala.util.matching.Regex implémente une API d'expression régulière idiomatique pour Scala en tant que wrapper sur java.util.regex.Pattern , et la syntaxe prise en charge est la même. Cela étant dit, la prise en charge par Scala des littéraux de chaîne multi-lignes rend le fanion `x` sensiblement plus utile, permettant des commentaires et ignorant les espaces blancs:

```
val dateRegex = """(?x:
   (\d{4}) # year
  -(\d{2}) # month
  -(\d{2}) # day
)""".r
```

Il existe une version surchargée de `r` , `def r(names: String*): Regex` qui vous permet d'attribuer des noms de groupe à vos captures de motif. Ceci est quelque peu fragile car les noms sont dissociés des captures et ne devraient être utilisés que si l'expression régulière sera utilisée à plusieurs endroits:

```
"""(\d{4})-(\d{2})-(\d{2})""".r("y", "m", "d").findFirstMatchIn(str) match {
```

```
  case Some(matched) =>
    val y = matched.group("y").toInt
    val m = matched.group("m").toInt
    val d = matched.group("d").toInt
    java.time.LocalDate.of(y, m, d)
  case None => ???
}
```

Répétition de la correspondance d'un motif dans une chaîne

```
val re = """\((.*?)\)""".r

val str =
"(The)(example)(of)(repeating)(pattern)(in)(a)(single)(string)(I)(had)(some)(trouble)(with)(once)"


re.findAllMatchIn(str).map(_.group(1)).toList
res2: List[String] = List(The, example, of, repeating, pattern, in, a, single, string, I, had,
some, trouble, with, once)
```

Lire Expressions régulières en ligne: https://riptutorial.com/fr/scala/topic/2891/expressions-regulieres

Chapitre 15: Extracteurs

Syntaxe

- extracteur de val (extractValue1, _ / * seconde valeur extraite ignorée * /) = valeurToBeExtracter
- match valueToBeExtracted {extracteur de cas (extractValue1, _) => ???}
- val (tuple1, tuple2, tuple3) = tupleWith3Elements
- objet Foo {def unapply (foo: Foo): Option [String] = Some (foo.x); }

Examples

Extracteurs de tuple

`x` et `y` sont extraits du tuple:

```
val (x, y) = (1337, 42)
// x: Int = 1337
// y: Int = 42
```

Pour ignorer une valeur, utilisez _ :

```
val (_, y: Int) = (1337, 42)
// y: Int = 42
```

Déballer un extracteur:

```
val myTuple = (1337, 42)
myTuple._1  // res0: Int = 1337
myTuple._2  // res1: Int = 42
```

Notez que tuples ont une longueur maximale de 22 ans , et donc `._1` par `._22` fonctionnera (en supposant que le tuple est au moins de cette taille).

Les extracteurs de tuples peuvent être utilisés pour fournir des arguments symboliques aux fonctions littérales:

```
val persons = List("A." -> "Lovelace", "G." -> "Hopper")
val names = List("Lovelace, A.", "Hopper, G.")

assert {
  names ==
    (persons map { name =>
      s"${name._2}, ${name._1}"
    })
}

assert {
```

```
  names ==
    (persons map { case (given, surname) =>
      s"$surname, $given"
    })
}
```

Extracteurs de classe de cas

Une classe de cas est une classe avec beaucoup de code standard standard inclus automatiquement. Un des avantages de ceci est que Scala facilite l'utilisation des extracteurs avec des classes de cas.

```
case class Person(name: String, age: Int)  // Define the case class
val p = Person("Paola", 42)  // Instantiate a value with the case class type

val Person(n, a) = p  // Extract values n and a
// n: String = Paola
// a: Int = 42
```

A ce stade, `n` et `a` sont tous deux `val` s dans le programme et sont accessibles en tant que tels: on dit qu'ils ont été «extraits» de p. Continue:

```
val p2 = Person("Angela", 1337)

val List(Person(n1, a1), Person(_, a2)) = List(p, p2)
// n1: String = Paola
// a1: Int = 42
// a2: Int = 1337
```

Nous voyons ici deux choses importantes:

- L'extraction peut avoir lieu à des niveaux «profonds»: les propriétés des objets imbriqués peuvent être extraites.
- Tous les éléments ne *doivent* pas être extraits. Le caractère générique _ indique que cette valeur particulière peut être n'importe quoi et est ignorée. Aucun `val` est créé.

En particulier, cela peut faciliter l'adaptation des collections:

```
val ls = List(p1, p2, p3)  // List of Person objects
ls.map(person => person match {
  case Person(n, a) => println("%s is %d years old".format(n, a))
})
```

Ici, nous avons du code qui utilise l'extracteur pour vérifier explicitement que cette `person` est un objet `Person` et extraire immédiatement les variables qui nous intéressent: `n` et `a` .

Unapply - Extracteurs personnalisés

Une extraction personnalisée peut être écrite en implémentant la méthode `unapply` et en renvoyant une valeur de type `Option` :

```
class Foo(val x: String)

object Foo {
  def unapply(foo: Foo): Option[String] = Some(foo.x)
}

new Foo("42") match {
  case Foo(x) => x
}
// "42"
```

Le type de retour d' `unapply` peut être différent de `Option` , à condition que le type renvoyé fournisse des méthodes `get` et `isEmpty` . Dans cet exemple, `Bar` est défini avec ces méthodes, et `unapply` renvoie une instance de `Bar` :

```
class Bar(val x: String) {
  def get = x
  def isEmpty = false
}

object Bar {
  def unapply(bar: Bar): Bar = bar
}

new Bar("1337") match {
  case Bar(x) => x
}
// "1337"
```

Le type de retour de `unapply` peut aussi être un `Boolean` , ce qui est un cas particulier qui ne répond pas aux exigences de `get` et `isEmpty` ci-dessus. Cependant, notez dans cet exemple que `DivisibleByTwo` est un objet, pas une classe, et ne prend pas de paramètre (et donc ce paramètre ne peut pas être lié):

```
object DivisibleByTwo {
  def unapply(num: Int): Boolean = num % 2 == 0
}

4 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// yes

3 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// no
```

Rappelez-vous que `unapply` va pas dans l'objet compagnon d'une classe, pas dans la classe. L'exemple ci-dessus sera clair si vous comprenez cette distinction.

Notation Infix Extractor

Si une classe de cas a exactement deux valeurs, son extracteur peut être utilisé en notation infixe.

```
case class Pair(a: String, b: String)
val p: Pair = Pair("hello", "world")
val x Pair y = p
//x: String = hello
//y: String = world
```

Tout extracteur qui retourne un 2-tuple peut fonctionner de cette façon.

```
object Foo {
    def unapply(s: String): Option[(Int, Int)] = Some((s.length, 5))
}
val a Foo b = "hello world!"
//a: Int = 12
//b: Int = 5
```

Extracteurs de regex

Une expression régulière avec des parties groupées peut être utilisée comme extracteur:

```
scala> val address = """(.+):(\d+)""".r
address: scala.util.matching.Regex = (.+):(\d+)

scala> val address(host, port) = "some.domain.org:8080"
host: String = some.domain.org
port: String = 8080
```

Notez que si elle ne correspond pas, une `MatchError` sera lancée à l'exécution:

```
scala> val address(host, port) = "something not a host and port"
scala.MatchError: something not a host and port (of class java.lang.String)
```

Extracteurs transformateurs

Le comportement des extracteurs peut être utilisé pour obtenir des valeurs arbitraires à partir de leurs entrées. Cela peut être utile dans les scénarios où vous souhaitez pouvoir agir sur les résultats d'une transformation en cas de réussite de la transformation.

Prenons l'exemple des différents formats de noms d'utilisateur utilisables dans un environnement Windows :

```
object UserPrincipalName {
  def unapply(str: String): Option[(String, String)] = str.split('@') match {
    case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
    case _ => None
  }
}

object DownLevelLogonName {
  def unapply(str: String): Option[(String, String)] = str.split('\\') match {
    case Array(d, u) if u.length > 0 && d.length > 0 => Some((d, u))
```

```
    case _ => None
  }
}

def getDomain(str: String): Option[String] = str match {
  case UserPrincipalName(_, domain) => Some(domain)
  case DownLevelLogonName(domain, _) => Some(domain)
  case _ => None
}
```

En fait, il est possible de créer un extracteur présentant les deux comportements en élargissant les types qu'il peut correspondre:

```
object UserPrincipalName {
  def unapply(obj: Any): Option[(String, String)] = obj match {
    case upn: UserPrincipalName => Some((upn.username, upn.domain))
    case str: String => str.split('@') match {
      case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
      case _ => None
    }
    case _ => None
  }
}
```

En général, les extracteurs sont simplement une reformulation pratique du modèle `Option` , appliqué aux méthodes avec des noms tels que `tryParse` :

```
UserPrincipalName.unapply("user@domain") match {
  case Some((u, d)) => ???
  case None => ???
}
```

Lire Extracteurs en ligne: https://riptutorial.com/fr/scala/topic/930/extracteurs

Chapitre 16: Fonction d'ordre supérieur

Remarques

Scala s'efforce de traiter les méthodes et les fonctions de manière identique sur le plan syntaxique. Mais sous le capot, ce sont des concepts distincts.

Une méthode est un code exécutable et n'a aucune représentation de valeur.

Une fonction est une instance d'objet réelle de type `Function1` (ou un type similaire d'une autre arité). Son code est contenu dans sa méthode d' `apply` . Effectivement, il agit simplement comme une valeur qui peut être transmise.

Incidemment, la capacité à traiter les fonctions comme des valeurs *est* exactement ce que l'on entend par un langage prenant en charge les fonctions d'ordre supérieur. Les instances de fonction sont l'approche de Scala pour implémenter cette fonctionnalité.

Une fonction réelle d'ordre supérieur est une fonction qui prend une valeur de fonction en tant qu'argument ou retourne une valeur de fonction. Mais dans Scala, comme toutes les opérations sont des méthodes, il est plus général de penser à des méthodes qui reçoivent ou renvoient des paramètres de fonction. Donc, la `map` , telle que définie sur `Seq` pourrait être considérée comme une "fonction d'ordre supérieur" en raison de son paramètre étant une fonction, mais ce n'est pas littéralement une fonction; c'est une méthode.

Examples

Utiliser des méthodes comme valeurs de fonction

Le compilateur Scala convertira automatiquement les méthodes en valeurs de fonction afin de les transmettre à des fonctions de niveau supérieur.

```
object MyObject {
  def mapMethod(input: Int): String = {
    int.toString
  }
}

Seq(1, 2, 3).map(MyObject.mapMethod) // Seq("1", "2", "3")
```

Dans l'exemple ci-dessus, `MyObject.mapMethod` n'est pas un appel de fonction, mais est transmis à `map` tant que valeur. En effet, la `map` *nécessite* une valeur de fonction qui lui est transmise, comme on peut le voir dans sa signature. La signature de la `map` d'une `List[A]` (une liste d'objets de type `A`) est:

```
def map[B](f: (A) ⇒ B): List[B]
```

La partie `f: (A) => B` indique que le paramètre de cet appel de méthode est une fonction qui prend

un objet de type `A` et retourne un objet de type `B` `A` et `B` sont définis arbitrairement. En revenant au premier exemple, nous pouvons voir que `mapMethod` prend un `Int` (qui correspond à `A`) et renvoie une `String` (qui correspond à `B`). Ainsi, `mapMethod` est une valeur de fonction valide à transmettre à la `map` . Nous pourrions réécrire le même code comme ceci:

```
Seq(1, 2, 3).map(x:Int => int.toString)
```

Cela inscrit la valeur de la fonction, ce qui peut ajouter de la clarté aux fonctions simples.

Fonctions d'ordre élevé (fonction comme paramètre)

Une fonction d'ordre supérieur, par opposition à une fonction du premier ordre, peut prendre l'une des trois formes suivantes:

- Un ou plusieurs de ses paramètres est une fonction et renvoie une valeur.
- Il retourne une fonction, mais aucun de ses paramètres n'est une fonction.
- Les deux ci-dessus: Un ou plusieurs de ses paramètres est une fonction, et il retourne une fonction.

```
object HOF {
    def main(args: Array[String]) {
    val list =
List(("Srini","E"),("Subash","R"),("Ranjith","RK"),("Vicky","s"),("Sudhar","s"))
    //HOF
     val fullNameList= list.map(n => getFullName(n._1, n._2))

     }

    def getFullName(firstName: String, lastName: String): String = firstName + "." +
lastName
    }
```

Ici, la fonction map prend comme paramètre une fonction `getFullName(n._1,n._2)` . Cela s'appelle **HOF (Fonction d'ordre supérieur).**

Arguments évaluation paresseuse

Scala prend en charge l'évaluation paresseuse des arguments de fonction en utilisant la notation: `def func(arg: => String)` . Un tel argument de fonction peut prendre un objet `String` régulier ou une fonction d'ordre supérieur avec le type de retour `String` . Dans le second cas, l'argument de fonction serait évalué sur l'accès aux valeurs.

S'il vous plaît voir l'exemple:

```
def calculateData: String = {
  print("Calculating expensive data! ")
  "some expensive data"
}
```

```
def dumbMediator(preconditions: Boolean, data: String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

def smartMediator(preconditions: Boolean, data: => String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

smartMediator(preconditions = false, calculateData)

dumbMediator(preconditions = false, calculateData)
```

`smartMediator` appel de `smartMediator` renvoie la valeur None et imprime le message `"Applying mediator"` .

`dumbMediator` appel `dumbMediator` renvoie la valeur None et imprime le message `"Calculating expensive data! Applying mediator"` .

L'évaluation différée peut être extrêmement utile lorsque vous souhaitez optimiser un calcul d'arguments coûteux.

Lire Fonction d'ordre supérieur en ligne: https://riptutorial.com/fr/scala/topic/1642/fonction-d-ordre-superieur

Chapitre 17: Fonctions définies par l'utilisateur pour la ruche

Examples

Un simple UDF Hive au sein d'Apache Spark

```
import org.apache.spark.sql.functions._

// Create a function that uses the content of the column inside the dataframe
val code = (param: String) => if (param == "myCode") 1 else 0
// With that function, create the udf function
val myUDF = udf(code)
// Apply the udf to a column inside the existing dataframe, creating a dataframe with the
additional new column
val newDataframe = aDataframe.withColumn("new_column_name", myUDF(col(inputColumn)))
```

Lire Fonctions définies par l'utilisateur pour la ruche en ligne:
https://riptutorial.com/fr/scala/topic/8241/fonctions-definies-par-l-utilisateur-pour-la-ruche

Chapitre 18: Fonctions partielles

Examples

Composition

Les fonctions partielles sont souvent utilisées pour définir une fonction totale dans les parties:

```
sealed trait SuperType
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val pfA: PartialFunction[SuperType, Int] = {
  case A => 5
}

val pfB: PartialFunction[SuperType, Int] = {
  case B => 10
}

val input: Seq[SuperType] = Seq(A, B, C)

input.map(pfA orElse pfB orElse {
  case _ => 15
}) // Seq(5, 10, 15)
```

Dans cette utilisation, les fonctions partielles sont tentées par ordre de concaténation avec la méthode `orElse` . En règle générale, une fonction partielle finale est fournie qui correspond à tous les cas restants. Collectivement, la combinaison de ces fonctions agit comme une fonction totale.

Ce modèle est généralement utilisé pour séparer les problèmes pour lesquels une fonction peut agir efficacement comme répartiteur pour des chemins de code disparates. Ceci est courant, par exemple, dans la méthode de réception d'un acteur Akka .

Utilisation avec `collect`

Alors que les fonctions partielles sont souvent utilisées comme syntaxe pratique pour les fonctions totales, en incluant une dernière correspondance générique (`case _`), dans certaines méthodes, leur partialité est la clé. Un exemple très courant dans Scala idiomatique est la méthode de `collect` , définie dans la bibliothèque de collections Scala. Ici, les fonctions partielles permettent aux fonctions courantes d'examiner les éléments d'une collection pour les mapper et / ou les filtrer pour qu'ils se produisent dans une syntaxe compacte.

Exemple 1

En supposant que nous avons une fonction racine carrée définie comme fonction partielle:

```
val sqRoot:PartialFunction[Double,Double] = { case n if n > 0 => math.sqrt(n) }
```

On peut l'invoquer avec le combinateur de `collect` :

```
List(-1.1,2.2,3.3,0).collect(sqRoot)
```

effectuer efficacement la même opération que:

```
List(-1.1,2.2,3.3,0).filter(sqRoot.isDefinedAt).map(sqRoot)
```

Exemple 2

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case class A(value: Int) extends SuperType
case class B(text: String) extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A(5), B("hello"), C, A(25), B(""))

input.collect {
  case A(value) if value < 10   => value.toString
  case B(text) if text.nonEmpty => text
} // Seq("5", "hello")
```

Il y a plusieurs choses à noter dans l'exemple ci-dessus:

- Le côté gauche de chaque correspondance de modèle sélectionne efficacement les éléments à traiter et à inclure dans la sortie. Toute valeur qui ne dispose pas d' une correspondance le `case` est simplement omis.
- Le côté droit définit le traitement spécifique à la casse à appliquer.
- La correspondance de modèle lie la variable à utiliser dans les instructions de garde (les clauses `if`) et à droite.

Syntaxe de base

Scala a un type spécial de fonction appelé fonction partielle , qui étend les fonctions normales - ce qui signifie qu'une instance de `PartialFunction` peut être utilisée partout où la `Function1` est attendue. Les fonctions partielles peuvent être définies de manière anonyme à l'aide de la syntaxe de `case` utilisée également dans la correspondance de modèle :

```
val pf: PartialFunction[Boolean, Int] = {
  case true => 7
}

pf.isDefinedAt(true) // returns true
pf(true) // returns 7

pf.isDefinedAt(false) // returns false
pf(false) // throws scala.MatchError: false (of class java.lang.Boolean)
```

Comme on le voit dans l'exemple, une fonction partielle n'a pas besoin d'être définie sur tout le domaine de son premier paramètre. Une instance standard de `Function1` est supposée être *totale* , ce qui signifie qu'elle est définie pour chaque argument possible.

Utilisation comme fonction totale

Les fonctions partielles sont très courantes dans les Scala idiomatiques. Ils sont souvent utilisés pour leur syntaxe pratique à base de `case` afin de définir les fonctions totales sur les traits :

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A, B, C)

input.map {
  case A => 5
  case _ => 10
} // Seq(5, 10, 10)
```

Cela enregistre la syntaxe supplémentaire d'une déclaration de `match` dans une fonction anonyme régulière. Comparer:

```
input.map { item =>
  item match {
    case A => 5
    case _ => 10
  }
} // Seq(5, 10, 10)
```

Il est également fréquemment utilisé pour effectuer une décomposition de paramètres en utilisant la correspondance de modèle, lorsqu'un tuple ou une classe de cas est transmis à une fonction:

```
val input = Seq("A" -> 1, "B" -> 2, "C" -> 3)

input.map { case (a, i) =>
   a + i.toString
} // Seq("A1", "B2", "C3")
```

Utilisation pour extraire des tuples dans une fonction de carte

Ces trois fonctions de carte sont équivalentes, utilisez donc la variante la plus lisible par votre équipe.

```
val numberNames = Map(1 -> "One", 2 -> "Two", 3 -> "Three")

// 1. No extraction
numberNames.map(it => s"${it._1} is written ${it._2}" )

// 2. Extraction within a normal function
numberNames.map(it => {
    val (number, name) = it
    s"$number is written $name"
})

// 3. Extraction via a partial function (note the brackets in the parentheses)
numberNames.map({ case (number, name) => s"$number is written $name" })
```

La fonction partielle **doit correspondre à toutes les entrées** : toute observation qui ne correspond pas déclenche une exception à l'exécution.

Lire Fonctions partielles en ligne: https://riptutorial.com/fr/scala/topic/1638/fonctions-partielles

Chapitre 19: Futures

Examples

Créer un avenir

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutureDivider {
    def divide(a: Int, b: Int): Future[Int] = Future {
        // Note that this is integer division.
        a / b
    }
}
```

Tout simplement, la méthode de la `divide` crée un futur qui se résoudra avec le quotient d' `a` over `b` .

Consommer un avenir prospère

La manière la plus simple de consommer un futur *réussi* - ou plutôt d'obtenir la valeur dans le futur - consiste à utiliser la méthode de la `map` . Supposons qu'un code appelle la méthode de `divide` de l'objet `FutureDivider` de l'exemple "Creating a Future". À quoi le code devrait-il ressembler pour obtenir le quotient d' `a` over `b` ?

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(quotient)
        }
    }
}
```

Consommer un avenir défaillant

Parfois, le calcul dans un avenir peut créer une exception, ce qui entraînera l'échec du futur. Dans l'exemple "Creating a Future", que se passe-t-il si le code d'appel passe `55` et `0` à la méthode de `divide` ? Il lancerait une `ArithmeticException` après avoir essayé de diviser par zéro, bien sûr. Comment cela serait-il géré en consommant du code? Il y a en fait une poignée de manières de traiter les échecs.

Gérer l'exception avec la `recover` et la correspondance de modèle.

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)
```

```
        eventualQuotient recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Traitez l'exception avec la projection `failed` , où l'exception devient la valeur du futur:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        // Note the use of the dot operator to get the failed projection and map it.
        eventualQuotient.failed.map {
            ex => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Assembler l'avenir

Les exemples précédents ont mis en évidence les caractéristiques individuelles d'un futur, la gestion des cas de réussite et d'échec. Cependant, les deux fonctionnalités sont généralement beaucoup plus complexes. Voici l'exemple, écrit de manière plus nette et plus réaliste:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(s"Quotient: $quotient")
        } recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Séquençage et traversée des Futures

Dans certains cas, il est nécessaire de calculer un montant variable de valeurs sur des contrats à terme séparés. Supposons avoir une `List[Future[Int]]` , mais au lieu de cela, une `List[Int]` doit être traitée. Ensuite, la question est de savoir comment transformer cette instance de `List[Future[Int]]` en `Future[List[Int]]` . Pour cela, il existe la méthode de `sequence` sur l'objet compagnon `Future` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.sequence(listOfFuture)
```

En général, la `sequence` est un opérateur communément connu dans le monde de la programmation fonctionnelle qui transforme `F[G[T]]` en `G[F[T]]` avec des restrictions sur `F` et `G`

Il existe un autre opérateur appelé `traverse` , qui fonctionne de manière similaire, mais prend une fonction comme argument supplémentaire. Avec la fonction d'identité `x => x` le paramètre se comporte comme l'opérateur de `sequence` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.traverse(listOfFuture)(x => x)
```

Cependant, l'argument supplémentaire permet de modifier chaque future instance dans la liste donnée `listOfFuture` . De plus, le premier argument n'a pas besoin d'être une liste de `Future` . Il est donc possible de transformer l'exemple comme suit:

```
def futureOfList: Future[List[Int]] = Future.traverse(List(1,2,3))(Future(_))
```

Dans ce cas, la `List(1,2,3)` est transmise directement en tant que premier argument et la fonction d'identité `x => x` est remplacée par la fonction `Future(_)` pour envelopper de la même manière chaque valeur `Int` dans un `Future` . Un avantage de ceci est que la `List[Future[Int]]` intermédiaire `List[Future[Int]]` peut être omise pour améliorer les performances.

Combiner plusieurs contrats à terme - Pour la compréhension

Le *pour la compréhension* est un moyen compact d'exécuter un bloc de code qui dépend du résultat réussi de plusieurs contrats à terme.

Avec `f1, f2, f3` trois `Future[String]` qui contiendront les chaînes `one, two, three` respectivement,

```
val fCombined =
    for {
        s1 <- f1
        s2 <- f2
        s3 <- f3
    } yield (s"$s1 - $s2 - $s3")
```

`fCombined` sera un `Future[String]` contenant la chaîne `one - two - three` une fois que tous les futurs auront été exécutés avec succès.

Notez qu'un ExectionContext implicite est supposé ici.

De plus, gardez à l'esprit que la compréhension n'est qu'un sucre syntaxique pour une méthode flatMap, de sorte que la construction d'objets futurs pour body éliminerait l'exécution simultanée de blocs de code entourés de futurs et aboutirait à un code séquentiel. Vous le voyez sur l'exemple:

```
val result1 = for {
  first <- Future {
    Thread.sleep(2000)
    System.currentTimeMillis()
  }
  second <- Future {
    Thread.sleep(1000)
    System.currentTimeMillis()
```

```
  }
} yield first - second

val fut1 = Future {
  Thread.sleep(2000)
  System.currentTimeMillis()
}
val fut2 = Future {
  Thread.sleep(1000)
  System.currentTimeMillis()
}
val result2 = for {
  first <- fut1
  second <- fut2
} yield first - second
```

La valeur entourée par l'objet `result1` serait toujours négative alors que `result2` serait positif.

Pour plus de détails sur la *compréhension* et le `yield` en général, voir http://docs.scala-lang.org/tutorials/FAQ/yield.html

Lire Futures en ligne: https://riptutorial.com/fr/scala/topic/3245/futures

Chapitre 20: Implicits

Syntaxe

- val implicite x: T = ???

Remarques

Les classes implicites permettent d'ajouter des méthodes personnalisées aux types existants, sans avoir à modifier leur code, ce qui enrichit les types sans avoir à contrôler le code.

L'utilisation de types implicites pour enrichir une classe existante est souvent appelée modèle «enrichir ma bibliothèque».

Restrictions sur les classes implicites

1. Les classes implicites ne peuvent exister que dans une autre classe, objet ou trait.
2. Les classes implicites ne peuvent avoir qu'un seul paramètre constructeur non implicite.
3. Il se peut qu'il n'y ait pas d'autre définition d'objet, de classe, de trait ou de membre de classe dans la même portée qui porte le même nom que la classe implicite.

Examples

Conversion implicite

Une conversion implicite permet au compilateur de convertir automatiquement un objet d'un type en un autre. Cela permet au code de traiter un objet comme un objet d'un autre type.

```
case class Foo(i: Int)

// without the implicit
Foo(40) + 2    // compilation-error (type mismatch)

// defines how to turn a Foo into an Int
implicit def fooToInt(foo: Foo): Int = foo.i

// now the Foo is converted to Int automatically when needed
Foo(40) + 2    // 42
```

La conversion est à sens unique: dans ce cas, vous ne pouvez pas convertir `42` en `Foo(42)` . Pour ce faire, une deuxième conversion implicite doit être définie:

```
implicit def intToFoo(i: Int): Foo = Foo(i)
```

Notez que c'est le mécanisme par lequel une valeur flottante peut être ajoutée à une valeur entière, par exemple.

> Les conversions implicites doivent être utilisées avec parcimonie car elles masquent ce qui se passe. Il est recommandé d'utiliser une conversion explicite via un appel de méthode, sauf s'il existe un gain de lisibilité tangible lié à l'utilisation d'une conversion implicite.

Il n'y a pas d'impact significatif sur les performances des conversions implicites.

Scala importe automatiquement diverses conversions implicites dans `scala.Predef` , y compris toutes les conversions de Java vers Scala et `scala.Predef` . Ceux-ci sont inclus par défaut dans toute compilation de fichiers.

Paramètres implicites

Les paramètres implicites peuvent être utiles si un paramètre d'un type doit être défini une fois dans la portée, puis appliqué à toutes les fonctions qui utilisent une valeur de ce type.

Un appel de fonction normal ressemble à ceci:

```
// import the duration methods
import scala.concurrent.duration._

// a normal method:
def doLongRunningTask(timeout: FiniteDuration): Long = timeout.toMillis

val timeout = 1.second
// timeout: scala.concurrent.duration.FiniteDuration = 1 second

// to call it
doLongRunningTask(timeout) // 1000
```

Maintenant, disons que nous avons des méthodes qui ont toutes une durée de temporisation, et nous voulons appeler toutes ces méthodes en utilisant le même délai. Nous pouvons définir le délai d'attente comme une variable implicite.

```
// import the duration methods
import scala.concurrent.duration._

// dummy methods that use the implicit parameter
def doLongRunningTaskA()(implicit timeout: FiniteDuration): Long = timeout.toMillis
def doLongRunningTaskB()(implicit timeout: FiniteDuration): Long = timeout.toMillis

// we define the value timeout as implicit
implicit val timeout: FiniteDuration = 1.second

// we can now call the functions without passing the timeout parameter
doLongRunningTaskA() // 1000
doLongRunningTaskB() // 1000
```

La façon dont cela fonctionne est que le compilateur scalac recherche une valeur dans l'étendue qui est **marquée comme implicite** et **dont le type correspond à** celui du paramètre implicite. S'il en trouve un, il l'appliquera comme paramètre implicite.

> Notez que cela ne fonctionnera pas si vous définissez deux implicits, voire plus, du

même type dans la portée.

Pour personnaliser le message d'erreur, utilisez l'annotation `implicitNotFound` sur le type:

```
@annotation.implicitNotFound(msg = "Select the proper implicit value for type M[${A}]!")
case class M[A](v: A) {}

def usage[O](implicit x: M[O]): O = x.v

//Does not work because no implicit value is present for type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
implicit val first: M[Int] = M(1)
usage[Int]     //Works when `second` is not in scope
implicit val second: M[Int] = M(2)
//Does not work because more than one implicit values are present for the type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
```

Un timeout est un cas d'utilisation habituel pour cela, ou par exemple dans Akka, ActorSystem est (la plupart du temps) toujours le même, donc il est généralement passé implicitement. Une autre utilisation serait la librairie, le plus souvent avec les bibliothèques FP qui reposent sur des classes de types (comme scalaz , chats ou ravissement).

> Il est généralement considéré comme une mauvaise pratique d'utiliser des paramètres implicites avec des types de base comme *Int* , *Long* , *String,* etc., car cela créerait de la confusion et rendrait le code moins lisible.

Classes implicites

Les classes implicites permettent d'ajouter de nouvelles méthodes à des classes précédemment définies.

La classe `String` n'a pas de méthode `withoutVowels` . Cela peut être ajouté comme ça:

```
object StringUtil {
  implicit class StringEnhancer(str: String) {
    def withoutVowels: String = str.replaceAll("[aeiou]", "")
  }
}
```

La classe implicite a un seul paramètre constructeur (`str`) avec le type que vous souhaitez étendre (`String`) et contient la méthode que vous souhaitez "ajouter" au type (`withoutVowels`). Les méthodes nouvellement définies peuvent désormais être utilisées directement sur le type amélioré (lorsque le type amélioré est implicite):

```
import StringUtil.StringEnhancer // Brings StringEnhancer into implicit scope

println("Hello world".withoutVowels) // Hll wrld
```

Sous le capot, les classes implicites définissent une conversion implicite du type amélioré à la classe implicite, comme ceci:

```
implicit def toStringEnhancer(str: String): StringEnhancer = new StringEnhancer(str)
```

Les classes implicites sont souvent définies en tant que classes Value pour éviter de créer des objets d'exécution et supprimer ainsi la surcharge d'exécution:

```
implicit class StringEnhancer(val str: String) extends AnyVal {
    /* conversions code here */
}
```

Avec la définition améliorée ci-dessus, une nouvelle instance de `StringEnhancer` n'a pas besoin d'être créée chaque fois que la méthode `withoutVowels` est appelée.

Résolution des paramètres implicites Utilisation de 'implicitement'

En supposant une liste de paramètres implicite avec plusieurs paramètres implicites:

```
case class Example(p1:String, p2:String)(implicit ctx1:SomeCtx1, ctx2:SomeCtx2)
```

Maintenant, en supposant que l'une des instances implicites n'est pas disponible (`SomeCtx1`) alors que toutes les autres instances implicites nécessaires sont dans la portée, pour créer une instance de la classe, une instance de `SomeCtx1` doit être fournie.

Cela peut être fait en préservant chaque autre instance implicite dans la portée en utilisant le mot-clé `implicitly` :

```
Example("something","somethingElse")(new SomeCtx1(), implicitly[SomeCtx2])
```

Implique dans le REPL

Pour afficher tous les `implicits` dans la portée d'une session REPL:

```
scala> :implicits
```

Inclure également les conversions implicites définies dans `Predef.scala` :

```
scala> :implicits -v
```

Si on a une expression et que l'on souhaite voir l'effet de toutes les règles de réécriture qui s'y appliquent (y compris les implicits):

```
scala> reflect.runtime.universe.reify(expr) // No quotes. reify is a macro operating directly
on code.
```

(Exemple:

```
scala> import reflect.runtime.universe._
scala> reify(Array("Alice", "Bob", "Eve").mkString(", "))
resX: Expr[String] = Expr[String](Predef.refArrayOps(Array.apply("Alice", "Bob",
```

```
"Eve")(Predef.implicitly)).mkString(", "))
```

)

Lire Implicits en ligne: https://riptutorial.com/fr/scala/topic/1732/implicits

Chapitre 21: Injection de dépendance

Examples

Modèle de gâteau avec la classe d'implémentation interne.

```
//create a component that will be injected
trait TimeUtil {
    lazy val timeUtil = new TimeUtilImpl()

    class TimeUtilImpl{
        def now() = new DateTime()
    }
}

//main controller is depended on time util
trait MainController {
    _ : TimeUtil => //inject time util into main controller

    lazy val mainController = new MainControllerImpl()

    class MainControllerImpl {
        def printCurrentTime() = println(timeUtil.now()) //timeUtil is injected from TimeUtil
trait
    }
}

object MainApp extends App {
    object app extends MainController
       with TimeUtil //wire all components

    app.mainController.printCurrentTime()
}
```

Dans l'exemple ci-dessus, j'ai montré comment injecter TimeUtil dans MainController.

La syntaxe la plus importante est l'auto-annotation (`_: TimeUtil =>`) qui consiste à injecter `TimeUtil` dans `MainController` . En d'autres termes, `MainController` dépend de `TimeUtil` .

J'utilise la classe interne (par exemple `TimeUtilImpl`) dans chaque composant car, à mon avis, il est plus facile de tester car nous pouvons nous moquer de la classe interne. Et il est également plus facile de suivre où la méthode est appelée lorsque le projet devient plus complexe.

Enfin, je connecte tous les composants ensemble. Si vous connaissez Guice, cela équivaut à `Binding`

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/scala/topic/5909/injection-de-dependance

Chapitre 22: Interopérabilité Java

Examples

Conversion de collections Scala en collections Java et vice versa

Lorsque vous devez transmettre une collection dans une méthode Java:

```
import scala.collection.JavaConverters._

val scalaList = List(1, 2, 3)
JavaLibrary.process(scalaList.asJava)
```

Si le code Java renvoie une collection Java, vous pouvez la transformer en une collection Scala de la même manière:

```
import scala.collection.JavaConverters._

val javaCollection = JavaLibrary.getList
val scalaCollection = javaCollection.asScala
```

Notez que ce sont des décorateurs, donc ils encapsulent simplement les collections sous-jacentes dans une interface de collection Scala ou Java. Par conséquent, les appels `.asJava` et `.asScala` ne copient pas les collections.

Tableaux

Les tableaux sont des tableaux JVM classiques avec une torsion qu'ils sont traités comme des invariants et ont des constructeurs spéciaux et des conversions implicites. Construisez-les sans le `new` mot-clé.

```
val a = Array("element")
```

Maintenant, `a` type a `Array[String]` .

```
val acs: Array[CharSequence] = a
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
```

Bien que `String` soit convertible en `CharSequence` , `Array[String]` n'est pas convertible en `Array[CharSequence]` .

Vous pouvez utiliser un `Array` comme les autres collections, grâce à une conversion implicite en `TraversableLike` `ArrayOps` :

```
val b: Array[Int] = a.map(_.length)
```

La plupart des collections Scala (`TraversableOnce`) ont une méthode `toArray` prenant un `ClassTag` implicite pour construire le tableau de résultats:

```
List(0).toArray
//> res1: Array[Int] = Array(0)
```

Cela facilite l'utilisation de `TraversableOnce` dans votre code Scala, puis le transmet au code Java qui attend un tableau.

Conversions de type Scala et Java

Scala propose des conversions implicites entre tous les principaux types de collection de l'objet JavaConverters.

Les conversions de type suivantes sont bidirectionnelles.

Type de Scala	Type Java
Itérateur	java.util.Iterator
Itérateur	java.util.Enumeration
Itérateur	java.util.Iterable
Itérateur	java.util.Collection
mutable.Buffer	java.util.List
mutable.Set	java.util.Set
mutable.Carte	java.util.map
mutable.ConcurrentMap	java.util.concurrent.ConcurrentMap

Certaines autres collections Scala peuvent également être converties en Java, mais ne sont pas reconverties au type Scala d'origine:

Scala Type	Type Java
Seq	java.util.List
mutable.Seq	java.util.List
Ensemble	java.util.Set
Carte	java.util.map

Référence :

Conversions entre les collections Java et Scala

Interfaces fonctionnelles pour les fonctions Scala - scala-java8-compat

Un kit de compatibilité Java 8 pour Scala.

La plupart des exemples sont copiés à partir de Readme

Convertisseurs entre scala.FunctionN et java.util.function

```
import java.util.function._
import scala.compat.java8.FunctionConverters._

val foo: Int => Boolean = i => i > 7
def testBig(ip: IntPredicate) = ip.test(9)
println(testBig(foo.asJava))  // Prints true

val bar = new UnaryOperator[String]{ def apply(s: String) = s.reverse }
List("cod", "herring").map(bar.asScala)    // List("doc", "gnirrih")

def testA[A](p: Predicate[A])(a: A) = p.test(a)
println(testA(asJavaPredicate(foo))(4))  // Prints false
```

Convertisseurs entre les classes scala.Option et java.util Facultatif, OptionalDouble, OptionalInt et OptionalLong.

```
import scala.compat.java8.OptionConverters._

    class Test {
      val o = Option(2.7)
      val oj = o.asJava        // Optional[Double]
      val ojd = o.asPrimitive  // OptionalDouble
      val ojds = ojd.asScala   // Option(2.7) again
    }
```

Convertisseurs de collections Scala en Java 8 Streams

```
import java.util.stream.IntStream

import scala.compat.java8.StreamConverters._
import scala.compat.java8.collectionImpl.{Accumulator, LongAccumulator}


  val m = collection.immutable.HashMap("fish" -> 2, "bird" -> 4)
  val parStream: IntStream = m.parValueStream
  val s: Int = parStream.sum
  // 6, potientially computed in parallel
  val t: List[String] = m.seqKeyStream.toScala[List]
  // List("fish", "bird")
  val a: Accumulator[(String, Int)] = m.accumulate // Accumulator[(String, Int)]

  val n = a.stepper.fold(0)(_ + _._1.length) +
    a.parStream.count // 8 + 2 = 10

  val b: LongAccumulator = java.util.Arrays.stream(Array(2L, 3L, 4L)).accumulate
  // LongAccumulator
```

```
val l: List[Long] = b.to[List] // List(2L, 3L, 4L)
```

Lire Interopérabilité Java en ligne: https://riptutorial.com/fr/scala/topic/2441/interoperabilite-java

Chapitre 23: Interpolation de chaîne

Remarques

Cette fonctionnalité existe dans Scala 2.10.0 et supérieur.

Examples

Bonjour l'interpolation de chaîne

L'interpolateur **s** permet l'utilisation de variables dans une chaîne.

```
val name = "Brian"
println(s"Hello $name")
```

Imprime "Hello Brian" sur la console quand il est exécuté.

Interpolation de chaînes formatées à l'aide de l'interpolateur f

```
val num = 42d
```

Imprimer deux décimales pour `num` utilisant *f*

```
println(f"$num%2.2f")
42.00
```

Imprimer `num` utilisant la notation scientifique en utilisant *e*

```
println(f"$num%e")
4.200000e+01
```

Imprimer `num` en hexadécimal avec *un*

```
println(f"$num%a")
0x1.5p5
```

D'autres chaînes de format sont disponibles à l' adresse https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

Utiliser l'expression dans les littéraux de chaîne

Vous pouvez utiliser des accolades pour interpoler des expressions dans des littéraux de chaîne:

```
def f(x: String) = x + x
val a = "A"
```

```
s"${a}"    // "A"
s"${f(a)}" // "AA"
```

Sans les accolades, scala n'interpolerait que l' *identifiant* après le `$` (dans ce cas, `f`). Comme il n'y a pas de conversion implicite de `f` en `String` il s'agit d'une exception dans cet exemple:

```
s"$f(a)"  // compile-time error (missing argument list for method f)
```

Interpolateurs de chaînes personnalisés

Il est possible de définir des interpolateurs de chaînes personnalisés en plus des interpolateurs intégrés.

```
my"foo${bar}baz"
```

Est développé par le compilateur pour:

```
new scala.StringContext("foo", "baz").my(bar)
```

`scala.StringContext` n'a pas `my` méthode et peut donc être fournie par une conversion implicite. Interpolateur personnalisé avec le même comportement que la fonction interne `s` l'interpolateur serait alors mis en œuvre comme suit:

```
implicit class MyInterpolator(sc: StringContext) {
  def my(subs: Any*): String = {
    val pit = sc.parts.iterator
    val sit = subs.iterator
    // Note parts.length == subs.length + 1
    val sb = new java.lang.StringBuilder(pit.next())
    while(sit.hasNext) {
      sb.append(sit.next().toString)
      sb.append(pit.next())
    }
    sb.toString
  }
}
```

Et l'interpolation de `my"foo${bar}baz"` serait destinée à:

```
new MyInterpolation(new StringContext("foo", "baz")).my(bar)
```

Notez qu'il n'y a pas de restriction sur les arguments ou le type de retour de la fonction d'interpolation. Cela nous conduit dans une voie sombre où la syntaxe d'interpolation peut être utilisée de manière créative pour construire des objets arbitraires, comme illustré dans l'exemple suivant:

```
case class Let(name: Char, value: Int)
```

```
implicit class LetInterpolator(sc: StringContext) {
  def let(value: Int): Let = Let(sc.parts(0).charAt(0), value)
}

let"a=${4}" // Let(a, 4)
let"b=${"foo"}" // error: type mismatch
let"c=" // error: not enough arguments for method let: (value: Int)Let
```

Interpolateurs de chaînes comme extracteurs

Il est également possible d'utiliser la fonction d'interpolation de chaîne de Scala pour créer des extracteurs élaborés («Pattern Matcher»), comme peut-être le plus utilisé dans l' API de quasiquotes des macros Scala.

Etant donné que `n"p0${i0}p1"` desugars à `new StringContext("p0", "p1").n(i0)` , il n'est peut-être pas surprenant que la fonctionnalité d'extracteur soit activée en fournissant une conversion implicite de `StringContext` en classe avec propriété `n` d'un type définissant une méthode `unapply` ou `unapplySeq` .

À titre d'exemple, considérons l'extracteur suivant qui extrait des segments de chemin en construisant une expression régulière à partir des parties `StringContext` . Nous pouvons alors déléguer la plupart des `unapplySeq` lourdes à la méthode `unapplySeq` fournie par scala.util.matching.Regex :

```
implicit class PathExtractor(sc: StringContext) {
  object path {
    def unapplySeq(str: String): Option[Seq[String]] =
      sc.parts.map(Regex.quote).mkString("^", "([^/]+)", "$").r.unapplySeq(str)
  }
}

"/documentation/scala/1629/string-interpolation" match {
  case path"/documentation/${topic}/${id}/${_}" => println(s"$topic, $id")
  case _ => ???
}
```

Notez que l'objet `path` pourrait également définir une méthode `apply` afin de se comporter comme un interpolateur régulier.

Interpolation de chaîne brute

Vous pouvez utiliser l'interpolateur **brut** si vous souhaitez qu'une chaîne soit imprimée telle quelle et sans aucune fuite de littéraux.

```
println(raw"Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Avec l'utilisation de l'interpolateur **brut** , vous devriez voir les éléments suivants imprimés dans la console:

```
Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde
```

Sans l'interpolateur **brut** , `\n` et `\t` auraient été échappés.

```
println("Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Impressions:

```
Hello World In English And French
English:       Hello World
French:        Bonjour Le Monde
```

Lire Interpolation de chaîne en ligne: https://riptutorial.com/fr/scala/topic/1629/interpolation-de-chaine

Chapitre 24: Invocation dynamique

Introduction

Scala vous permet d'utiliser l'appel dynamique lors de l'appel de méthodes ou de l'accès aux champs d'un objet. Au lieu de l' `scala.Dynamic` au langage, cela est possible grâce à la réécriture de règles similaires à celles des conversions implicites, activées par le trait de marqueur [`scala.Dynamic`] [Dynamic scaladoc]. Cela vous permet d'émuler la possibilité d'ajouter dynamiquement des propriétés aux objets présents dans les langages dynamiques, et plus encore. [Scaladoc dynamique]: http://www.scala-lang.org/api/2.12.x/scala/Dynamic.html

Syntaxe

- classe Foo étend Dynamic
- foo.field
- foo.field = valeur
- foo.method (args)
- foo.method (namedArg = x, y)

Remarques

Afin de déclarer les sous-types de `Dynamic` , la `dynamics scala.language.dynamics` linguistiques doit être activée, soit en important `scala.language.dynamics` soit en `-language:dynamics` option de compilation `-language:dynamics` . Les utilisateurs de cette `Dynamic` qui ne définissent pas leurs propres sous-types n'ont pas besoin de l'activer.

Examples

Accès aux champs

Ce:

```
class Foo extends Dynamic {
  // Expressions are only rewritten to use Dynamic if they are not already valid
  // Therefore foo.realField will not use select/updateDynamic
  var realField: Int = 5
  // Called for expressions of the type foo.field
  def selectDynamic(fieldName: String) = ???
  def updateDynamic(fieldName: String)(value: Int) = ???
}
```

permet un accès simple aux champs:

```
val foo: Foo = ???
foo.realField // Does NOT use Dynamic; accesses the actual field
foo.realField = 10 // Actual field access here too
```

```
foo.unrealField // Becomes foo.selectDynamic(unrealField)
foo.field = 10  // Becomes foo.updateDynamic("field")(10)
foo.field = "10" // Does not compile; "10" is not an Int.
foo.x() // Does not compile; Foo does not define applyDynamic, which is used for methods.
foo.x.apply() // DOES compile, as Nothing is a subtype of () => Any
// Remember, the compiler is still doing static type checks, it just has one more way to
// "recover" and rewrite otherwise invalid code now.
```

Appels de méthode

Ce:

```
class Villain(val minions: Map[String, Minion]) extends Dynamic {
  def applyDynamic(name: String)(jobs: Task*) = jobs.foreach(minions(name).do)
  def applyDynamicNamed(name: String)(jobs: (String, Task)*) = jobs.foreach {
    // If a parameter does not have a name, and is simply given, the name passed as ""
    case ("", task) => minions(name).do(task)
    case (subsys, task) => minions(name).subsystems(subsys).do(task)
  }
}
```

permet d'appeler des méthodes, avec et sans paramètres nommés:

```
val gru: Villain = ???
gru.blu() // Becomes gru.applyDynamic("blu")()
// Becomes gru.applyDynamicNamed("stu")(("fooer", ???), ("boomer", ???), ("", ???),
//          ("computer breaker", ???), ("fooer", ???))
// Note how the `???` without a name is given the name ""
// Note how both occurrences of `fooer` are passed to the method
gru.stu(fooer = ???, boomer = ???, ???, `computer breaker` = ???, fooer = ???)
gru.ERR("a") // Somehow, scalac thinks "a" is not a Task, though it clearly is (it isn't)
```

Interaction entre l'accès aux champs et la méthode de mise à jour

Légèrement contre-intuitif (mais aussi la seule façon saine de le faire fonctionner), ceci:

```
val dyn: Dynamic = ???
dyn.x(y) = z
```

est équivalent à:

```
dyn.selectDynamic("x").update(y, z)
```

tandis que

```
dyn.x(y)
```

est encore

```
dyn.applyDynamic("x")(y)
```

Il est important d'être conscient de cela, sinon il peut passer inaperçu et causer des erreurs étranges.

Lire Invocation dynamique en ligne: https://riptutorial.com/fr/scala/topic/8296/invocation-dynamique

Chapitre 25: JSON

Examples

JSON avec spray-json

spray-json fournit un moyen facile de travailler avec JSON. En utilisant des formats implicites, tout se passe "dans les coulisses":

Rendez la bibliothèque disponible avec SBT

Pour gérer `spray-json` avec des dépendances de bibliothèques gérées par SBT :

```
libraryDependencies += "io.spray" %% "spray-json" % "1.3.2"
```

Notez que le dernier paramètre, le numéro de version (`1.3.2`), peut être différent selon les projets.

La bibliothèque `spray-json` est hébergée sur repo.spray.io .

Importer la bibliothèque

```
import spray.json._
import DefaultJsonProtocol._
```

Le protocole JSON par défaut `DefaultJsonProtocol` contient des formats pour tous les types de base. Pour fournir une fonctionnalité JSON pour les types personnalisés, utilisez des générateurs de convenance pour les formats ou écrivez des formats explicitement.

Lire JSON

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = """{ "foo": "bar" }""".parseJson // JsValue = {"foo":"bar"}

res.convertTo[Map[String, String]] // Map(foo -> bar)
```

Ecrire JSON

```
val values = List("a", "b", "c")
values.toJson.prettyPrint // ["a", "b", "c"]
```

DSL

DSL n'est pas pris en charge.

Lecture-écriture aux classes de cas

L'exemple suivant montre comment sérialiser un objet de classe de cas au format JSON.

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = jsonFormat2(Address)
implicit val personFormat = jsonFormat2(Person)

// serialize a Person
Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = fred.toJson.prettyPrint
```

Cela se traduit par le JSON suivant:

```
{
  "name": "Fred",
  "address": {
    "street": "Awesome Street 9",
    "city": "SuperCity"
  }
}
```

Ce JSON peut à son tour être désérialisé dans un objet:

```
val personRead = fredJsonString.parseJson.convertTo[Person]
//Person(Fred,Address(Awesome Street 9,SuperCity))
```

Format personnalisé

Ecrivez un `JsonFormat` personnalisé si une sérialisation spéciale d'un type est requise. Par exemple, si les noms de champs sont différents dans Scala que dans JSON. Ou, si différents types de béton sont instanciés en fonction de l'entrée.

```
implicit object BetterPersonFormat extends JsonFormat[Person] {
    // deserialization code
    override def read(json: JsValue): Person = {
        val fields = json.asJsObject("Person object expected").fields
        Person(
            name = fields("name").convertTo[String],
            address = fields("home").convertTo[Address]
        )
    }
```

```
    // serialization code
    override def write(person: Person): JsValue = JsObject(
        "name" -> person.name.toJson,
        "home" -> person.address.toJson
    )
}
```

JSON avec Circé

Circé fournit des codecs dérivés à la compilation pour en / decode json dans les classes de cas. Un exemple simple ressemble à ceci:

```
import io.circe._
import io.circe.generic.auto._
import io.circe.parser._
import io.circe.syntax._

case class User(id: Long, name: String)

val user = User(1, "John Doe")

// {"id":1,"name":"John Doe"}
val json = user.asJson.noSpaces

// Right(User(1L, "John Doe"))
val res: Either[Error, User] = decode[User](json)
```

JSON avec play-json

play-json utilise des formats implicites comme les autres frameworks json

Dépendance de SBT: `libraryDependencies += ""com.typesafe.play" %% "play-json" % "2.4.8"`

```
import play.api.libs.json._
import play.api.libs.functional.syntax._ // if you need DSL
```

`DefaultFormat` contient des formats defaul pour lire / écrire tous les types de base. Pour fournir une fonctionnalité JSON pour vos propres types, vous pouvez utiliser des générateurs de convenance pour les formats ou écrire des formats de manière explicite.

Lire json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = Json.parse("""{ "foo": "bar" }""") // JsValue = {"foo":"bar"}

res.as[Map[String, String]]                  // Map(foo -> bar)
res.validate[Map[String, String]]            //JsSuccess(Map(foo -> bar),)
```

Écris json

```
val values = List("a", "b", "c")
```

```
Json.stringify(Json.toJson(values))          // ["a", "b", "c"]
```

DSL

```
val json = parse("""{ "foo": [{"foo": "bar"}]}""")
(json \ "foo").get                    //Simple path: [{"foo":"bar"}]
(json \\ "foo")                       //Recursive path:List([{"foo":"bar"}], "bar")
(json \ "foo")(0).get                  //Index lookup (for JsArrays): {"foo":"bar"}
```

Comme toujours, préférez la correspondance de modèle avec `JsSuccess` / `JsError` et essayez d'éviter les `.get` au `.get` , `array(i)` .

Lire et écrire dans une classe de cas

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = Json.format[Address]
implicit val personFormat = Json.format[Person]

// serialize a Person
val fred = Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = Json.stringify(Json.toJson(Json.toJson(fred)))

val personRead = Json.parse(fredJsonString).as[Person] //Person(Fred,Address(Awesome Street
9,SuperCity))
```

Format propre

Vous pouvez écrire votre propre JsonFormat si vous avez besoin d'une sérialisation spéciale de votre type (par exemple, nommer les champs différemment dans scala et Json ou instancier différents types de béton en fonction de l'entrée)

```
case class Address(street: String, city: String)

// create the formats and provide them implicitly
implicit object AddressFormatCustom extends Format[Address] {
  def reads(json: JsValue): JsResult[Address] = for {
    street <- (json \ "Street").validate[String]
    city <- (json \ "City").validate[String]
  } yield Address(street, city)

  def writes(x: Address): JsValue = Json.obj(
    "Street" -> x.street,
    "City" -> x.city
  )
}
// serialize an address
val address = Address("Awesome Street 9", "SuperCity")
val addressJsonString = Json.stringify(Json.toJson(Json.toJson(address)))
//{"Street":"Awesome Street 9","City":"SuperCity"}

val addressRead = Json.parse(addressJsonString).as[Address]
//Address(Awesome Street 9,SuperCity)
```

Alternative

Si json ne correspond pas exactement à vos champs de classe de cas (`isAlive` dans la classe de cas vs `is_alive` dans json):

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Boolean)

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").read[Boolean]
    ) (User.apply _)
}
```

Json avec des champs optionnels

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Option[Boolean])

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").readNullable[Boolean]
    ) (User.apply _)
}
```

Lecture des horodatages de json

Imaginez que vous ayez un objet Json, avec un champ d'horodatage Unix:

```
{
  "field": "example field",
  "date": 1459014762000
}
```

Solution:

```
case class JsonExampleV1(field: String, date: DateTime)
object JsonExampleV1{
  implicit val r: Reads[JsonExampleV1] = (
    (__ \ "field").read[String] and
      (__ \ "date").read[DateTime](Reads.DefaultJodaDateReads)
    )(JsonExampleV1.apply _)
}
```

Lecture de classes de cas personnalisées

Maintenant, si vous encapsulez vos identifiants d'objet pour la sécurité de type, vous apprécierez cela. Voir l'objet json suivant:

```
{
  "id": 91,
  "data": "Some data"
}
```

et les classes de cas correspondantes:

```
case class MyIdentifier(id: Long)

case class JsonExampleV2(id: MyIdentifier, data: String)
```

Il ne vous reste plus qu'à lire le type primitif (Long) et à mapper sur votre idenfier:

```
object JsonExampleV2 {
  implicit val r: Reads[JsonExampleV2] = (
      (__ \ "id").read[Long].map(MyIdentifier) and
    (__ \ "data").read[String]
    )(JsonExampleV2.apply _)
}
```

code à https://github.com/pedrorijo91/scala-play-json-examples

JSON avec json4s

json4s utilise des formats implicites comme les autres frameworks json.

Dépendance à SBT:

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.4.0"
//or
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.4.0"
```

Importations

```
import org.json4s.JsonDSL._
import org.json4s._
import org.json4s.native.JsonMethods._

implicit val formats = DefaultFormats
```

`DefaultFormats` contient des formats par défaut pour lire / écrire tous les types de base.

Lire json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = parse("""{ "foo": "bar" }""")           // JValue = {"foo":"bar"}
```

```
res.extract[Map[String, String]]                  // Map(foo -> bar)
```

Écris json

```
val values = List("a", "b", "c")
compact(render(values))         // ["a", "b", "c"]
```

DSL

```
json \ "foo"       //Simple path: JArray(List(JObject(List((foo,JString(bar))))))
json \\ "foo"      //Recursive path: ~List([{"foo":"bar"}], "bar")
(json \ "foo")(0)  //Index lookup (for JsArrays): JObject(List((foo,JString(bar))))
("foo" -> "bar") ~ ("field" -> "value")    // {"foo":"bar","field":"value"}
```

Lire et écrire dans une classe de cas

```
import org.json4s.native.Serialization.{read, write}

case class Address(street: String, city: String)
val addressString = write(Address("Awesome stree", "Super city"))
// {"street":"Awesome stree","city":"Super city"}

read[Address](addressString) // Address(Awesome stree,Super city)
//or
parse(addressString).extract[Address]
```

Lire et écrire des listes hétérogènes

Pour sérialiser et désérialiser une liste hétérogène (ou polymorphe), des indications de type spécifiques doivent être fournies.

```
trait Location
case class Street(name: String) extends Location
case class City(name: String, zipcode: String) extends Location
case class Address(street: Street, city: City) extends Location
case class Locations (locations : List[Location])

implicit val formats = Serialization.formats(ShortTypeHints(List(classOf[Street],
classOf[City], classOf[Address])))

val locationsString = write(Locations(Street("Lavelle Street"):: City("Super city","74658")))

read[Locations](locationsString)
```

Format propre

```
class AddressSerializer extends CustomSerializer[Address](format => (
  {
    case JObject(JField("Street", JString(s)) :: JField("City", JString(c)) :: Nil) =>
      new Address(s, c)
  },
  {
    case x: Address => ("Street" -> x.street) ~ ("City" -> x.city)
  }
```

```
  ))

implicit val formats = DefaultFormats + new AddressSerializer
val str = write[Address](Address("Awesome Stree", "Super City"))
// {"Street":"Awesome Stree","City":"Super City"}
read[Address](str)
// Address(Awesome Stree,Super City)
```

Lire JSON en ligne: https://riptutorial.com/fr/scala/topic/2348/json

Chapitre 26: La gestion des erreurs

Examples

Essayer

Utilisation de Try avec `map` , `getOrElse` et `flatMap` :

```
import scala.util.Try

val i = Try("123".toInt)      // Success(123)
i.map(_ + 1).getOrElse(321)   // 124

val j = Try("abc".toInt)      // Failure(java.lang.NumberFormatException)
j.map(_ + 1).getOrElse(321)   // 321

Try("123".toInt) flatMap { i =>
  Try("234".toInt)
    .map(_ + i)
}                             // Success(357)
```

En utilisant `Try` avec la correspondance de motif:

```
Try(parsePerson("John Doe")) match {
    case Success(person) => println(person.surname)
    case Failure(ex) => // Handle error ...
}
```

Non plus

Différents types de données pour erreur / succès

```
def getPersonFromWebService(url: String): Either[String, Person] = {

    val response = webServiceClient.get(url)

    response.webService.status match {
        case 200 => {
            val person = parsePerson(response)
            if(!isValid(person)) Left("Validation failed")
            else Right(person)
        }

        case _ => Left(s"Request failed with error code $response.status")
    }
}
```

Correspondance de motif sur l'une ou l'autre valeur

```
getPersonFromWebService("http://some-webservice.com/person") match {
    case Left(errorMessage) => println(errorMessage)
```

```
    case Right(person) => println(person.surname)
}
```

Convertir l'une ou l'autre valeur en option

```
val maybePerson: Option[Person] = getPersonFromWebService("http://some-
webservice.com/person").right.toOption
```

Option

L'utilisation de valeurs `null` est fortement déconseillée, sauf si elle interagit avec du code Java hérité qui attend `null` . Au lieu de cela, `Option` doit être utilisée lorsque le résultat d'une fonction peut être quelque chose (`Some`) ou rien (`None`).

Un bloc try-catch est plus approprié pour la gestion des erreurs, mais si la fonction peut légalement renvoyer rien, `Option` est appropriée à utiliser et simple.

Une `Option[T]` peut être soit `Some(value)` (contient une valeur de type `T`), soit `None` :

```
def findPerson(name: String): Option[Person]
```

Si personne n'est trouvé, `None` ne peut être retourné. Sinon, un objet de type `Some` contenant un objet `Person` est renvoyé. Ce qui suit sont des manières de gérer un objet de type `Option` .

Correspondance de motif

```
findPerson(personName) match {
    case Some(person) => println(person.surname)
    case None => println(s"No person found with name $personName")
}
```

Utiliser la carte et getOrElse

```
val name = findPerson(personName).map(_.firstName).getOrElse("Unknown")
println(name) // Prints either the name of the found person or "Unknown"
```

En utilisant fold

```
val name = findPerson(personName).fold("Unknown")(_.firstName)
// equivalent to the map getOrElse example above.
```

Conversion en Java

Si vous avez besoin de convertir un type d' `Option` en un type Java nul pour l'interopérabilité:

```
val s: Option[String] = Option("hello")
s.orNull           // "hello": String
s.getOrElse(null)  // "hello": String

val n: Option[Int] = Option(42)
n.orNull           // compilation failure (Cannot prove that Null <:< Int.)
n.getOrElse(null)  // 42
```

Gestion des erreurs provenant des futures

Lorsqu'une `exception` est lancée depuis un `Future` , vous pouvez (devrait) utiliser `recover` pour le gérer.

Par exemple,

```
def runFuture: Future = Future { throw new FairlyStupidException }

val itWillBeAwesome: Future = runFuture
```

... lancera une `Exception` partir de l' `Future` . Mais comme nous pouvons prédire qu'une `Exception` de type `FairlyStupidException` avec une probabilité élevée, nous pouvons traiter ce cas de manière élégante:

```
val itWillBeAwesomeOrIllRecover = runFuture recover {
    case stupid: FairlyStupidException =>
         BadRequest("Another stupid exception!")
}
```

Comme vous pouvez le voir, la méthode donnée pour `recover` est une fonction `PartialFunction` sur le domaine de `Throwable` , vous pouvez donc gérer certains types seulement et laisser le reste aller dans la gestion des exceptions à des niveaux supérieurs de la pile `Future` .

Notez que ceci est similaire à l'exécution du code suivant dans un contexte non `Future` :

```
def runNotFuture: Unit = throw new FairlyStupidException

try {
    runNotFuture
} catch {
    case e: FairlyStupidException => BadRequest("Another stupid exception!")
}
```

Il est très important de gérer les exceptions générées dans `Future` s car elles sont souvent insidieuses. Ils ne sont généralement pas présents dans votre visage, car ils s'exécutent dans un contexte d'exécution et un thread différents, et ne vous invitent donc pas à les corriger lorsqu'ils surviennent, surtout si vous ne remarquez rien dans les journaux ou le comportement du serveur. application.

Utilisation de clauses try-catch

En plus des constructions fonctionnelles telles que `Try` , `Option` et `Either` pour la gestion des

erreurs, Scala prend également en charge une syntaxe similaire à celle de Java, en utilisant une clause try-catch (avec également un bloc potential). La clause catch est une correspondance de modèle:

```
try {
  // ... might throw exception
} catch {
  case ioe: IOException => ... // more specific cases first
  case e: Exception => ...
  // uncaught types will be thrown
} finally {
  // ...
}
```

Convertir des exceptions en deux types d'option

Pour convertir des exceptions en types `Either` ou `Option` , vous pouvez utiliser des méthodes fournies dans `scala.util.control.Exception`

```
import scala.util.control.Exception._

val plain = "71a"
val optionInt: Option[Int] = catching(classOf[java.lang.NumberFormatException]) opt {
plain.toInt }
val eitherInt = Either[Throwable, Int] = catching(classOf[java.lang.NumberFormatException])
either { plain.toInt }
```

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/scala/topic/910/la-gestion-des-erreurs

Chapitre 27: Les fonctions

Remarques

Scala a des fonctions de première classe.

Différence entre les fonctions et les méthodes:

Une fonction n'est pas une méthode dans Scala: les fonctions sont une valeur et peuvent être affectées en tant que telles. Les méthodes (créées avec `def`), en revanche, doivent appartenir à une classe, à un trait ou à un objet.

- Les fonctions sont compilées dans une classe étendant un trait (tel que `Function1`) au moment de la compilation et sont instanciées à une valeur lors de l'exécution. Les méthodes, quant à elles, sont membres de leur classe, trait ou objet et n'existent pas en dehors de cela.
- Une méthode peut être convertie en une fonction, mais une fonction ne peut pas être convertie en une méthode.
- Les méthodes peuvent avoir un paramétrage de type, contrairement aux fonctions.
- Les méthodes peuvent avoir des valeurs par défaut, alors que les fonctions ne le peuvent pas.

Examples

Fonctions anonymes

Les fonctions anonymes sont des fonctions définies mais sans nom.

Voici une fonction anonyme qui prend deux entiers et renvoie la somme.

```
(x: Int, y: Int) => x + y
```

L'expression résultante peut être affectée à un `val` :

```
val sum = (x: Int, y: Int) => x + y
```

Les fonctions anonymes sont principalement utilisées comme arguments pour d'autres fonctions. Par exemple, la fonction `map` sur une collection attend une autre fonction comme argument:

```
// Returns Seq("FOO", "BAR", "QUX")
Seq("Foo", "Bar", "Qux").map((x: String) => x.toUpperCase)
```

Les types des arguments de la fonction anonyme peuvent être omis: les types sont inférés automatiquement :

```
Seq("Foo", "Bar", "Qux").map((x) => x.toUpperCase)
```

S'il n'y a qu'un seul argument, les parenthèses autour de cet argument peuvent être omises:

```
Seq("Foo", "Bar", "Qux").map(x => x.toUpperCase)
```

Raccourci sténographie

Il existe une syntaxe encore plus courte qui ne nécessite pas de noms pour les arguments. L'extrait ci-dessus peut être écrit:

```
Seq("Foo", "Bar", "Qux").map(_.toUpperCase)
```

_ représente les arguments de la fonction anonyme en position. Avec une fonction anonyme dotée de plusieurs paramètres, chaque occurrence de _ fera référence à un argument différent. Par exemple, les deux expressions suivantes sont équivalentes:

```
// Returns "FooBarQux" in both cases
Seq("Foo", "Bar", "Qux").reduce((s1, s2) => s1 + s2)
Seq("Foo", "Bar", "Qux").reduce(_ + _)
```

Lors de l'utilisation de cet raccourci, tout argument représenté par la position _ ne peut être référencé qu'une seule fois et dans le même ordre.

Fonctions anonymes sans paramètres

Pour créer une valeur pour une fonction anonyme ne prenant pas de paramètres, laissez la liste de paramètres vide:

```
val sayHello = () => println("hello")
```

Composition

La composition des fonctions permet à deux fonctions de fonctionner et d'être considérées comme une fonction unique. Exprimée en termes mathématiques, étant donné une fonction `f(x)` et une fonction `g(x)` , la fonction `h(x) = f(g(x))` .

Lorsqu'une fonction est compilée, elle est compilée dans un type lié à `Function1` . Scala propose deux méthodes dans la `Function1` mise en œuvre liées à la composition: `andThen` et `compose` . La méthode de `compose` correspond à la définition mathématique ci-dessus comme suit:

```
val f: B => C = ...
val g: A => B = ...

val h: A => C = f compose g
```

The `andThen` (pensez que `h(x) = g(f(x))`) a un sentiment plus "DSL-like":

```
val f: A => B = ...
val g: B => C = ...

val h: A => C = f andThen g
```

Une nouvelle fonction anonyme est allouée avec celle qui est fermée sur `f` et `g` . Cette fonction est liée à la nouvelle fonction `h` dans les deux cas.

```
def andThen(g: B => C): A => C = new (A => C){
  def apply(x: A) = g(self(x))
}
```

Si `f` ou `g` fonctionne via un effet secondaire, alors l'appel de `h` provoquera tous les effets secondaires de `f` et `g` dans l'ordre. La même chose est vraie de tout changement d'état mutable.

Relation avec les fonctions partielles

```
trait PartialFunction[-A, +B] extends (A => B)
```

Chaque `PartialFunction` un seul argument est également une `Function1` . Ceci est contre-intuitif dans un sens mathématique formel, mais correspond mieux au design orienté objet. Pour cette raison, il n'est pas nécessaire que `Function1` fournisse une méthode `isDefinedAt` `true` `isDefinedAt` .

Pour définir une fonction partielle (qui est aussi une fonction), utilisez la syntaxe suivante:

```
{ case i: Int => i + 1 } // or equivalently { case i: Int ⇒ i + 1 }
```

Pour plus de détails, consultez les fonctions partielles .

Lire Les fonctions en ligne: https://riptutorial.com/fr/scala/topic/477/les-fonctions

Chapitre 28: Les meilleures pratiques

Remarques

> Préférez les vals, les objets immuables et les méthodes sans effets secondaires. Atteignez-les en premier. Utilisez des vars, des objets mutables et des méthodes avec des effets secondaires lorsque vous en avez un besoin spécifique et que vous en avez la justification.

- *Programmation en Scala* , par Odersky, Spoon et Venners

Il y a plus d'exemples et de lignes directrices dans cette présentation par Odersky.

Examples

Rester simple

Ne pas trop compliquer les tâches simples. La plupart du temps, vous aurez seulement besoin de:

- types de données algébriques
- récursivité structurelle
- api semblable à une monade (`map` , `map flatMap` , `fold`)

Il y a beaucoup de choses compliquées à Scala, telles que:

- `Cake pattern` ou `Reader Monad` pour injection de dépendance.
- Passer des valeurs arbitraires comme arguments `implicit` .

Ces choses ne sont pas claires pour les nouveaux arrivants: évitez de les utiliser avant de les comprendre. L'utilisation de concepts avancés sans réel besoin brouille le code, le rendant *moins* facile à entretenir.

Ne pas emballer trop dans une expression.

- Trouver des noms significatifs pour les unités de calcul.
- Utiliser `for` compréhensions ou des `map` pour combiner des calculs ensemble.

Disons que vous avez quelque chose comme ça:

```
if (userAuthorized.nonEmtpy) {
  makeRequest().map {
    case Success(respone) =>
      someProcessing(..)
      if (resendToUser) {
        sendToUser(...)
      }
    ...
  }
```

```
}
```

Si toutes vos fonctions renvoient `Either` ou d'une autre `Validation` -comme type, vous pouvez écrire:

```
for {
  user     <- authorizeUser
  response <- requestToThirdParty(user)
  _        <- someProcessing(...)
} {
  sendToUser
}
```

Préférer un style fonctionnel, raisonnablement

Par défaut:

- Utilisez `val` , pas `var` , dans la mesure du possible. Cela vous permet de tirer parti de plusieurs utilitaires fonctionnels, notamment la distribution du travail.
- Utilisez la `recursion` et les `comprehensions` s, pas les boucles.
- Utilisez des collections immuables. Ceci est une corrolaire à l'utilisation de `val` chaque fois que possible.
- Concentrez-vous sur les transformations de données, la logique de style CQRS et non CRUD.

Il y a de bonnes raisons de choisir un style non fonctionnel:

- `var` peut être utilisé pour l'état local (par exemple, à l'intérieur d'un acteur).
- `mutable` donne de meilleures performances dans certaines situations.

Lire Les meilleures pratiques en ligne: https://riptutorial.com/fr/scala/topic/4376/les-meilleures-pratiques

Chapitre 29: Macros

Introduction

Les macros sont une forme de métaprogrammation de la compilation. Certains éléments du code Scala, tels que les annotations et les méthodes, peuvent être utilisés pour transformer d'autres codes lorsqu'ils sont compilés. Les macros sont des codes Scala ordinaires qui fonctionnent sur des types de données représentant un autre code. Le plugin [Macro Paradise] [] étend les capacités des macros au-delà du langage de base. [Macro Paradise]: http://docs.scala-lang.org/overviews/macros/paradise.html

Syntaxe

- def x () = macro x_impl // x est une macro, où x_impl est utilisé pour transformer le code
- def macroTransform (annottees: Any *): Any = macro impl // Utilisation dans les annotations pour les rendre macros

Remarques

Les macros sont une fonctionnalité de langage qui doit être activée, soit en important `scala.language.macros` soit avec l'option de compilation `-language:macros` . Seules les définitions de macro l'exigent. le code qui utilise des macros ne doit pas le faire.

Examples

Annotation Macro

Cette simple annotation de macro génère l'élément annoté tel quel.

```
import scala.annotation.{compileTimeOnly, StaticAnnotation}
import scala.reflect.macros.whitebox.Context

@compileTimeOnly("enable macro paradise to expand macro annotations")
class noop extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any = macro linkMacro.impl
}

object linkMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._

    c.Expr[Any](q"{..$annottees}")
  }
}
```

L'annotation `@compileTimeOnly` génère une erreur avec un message indiquant que le plugin `paradise` compiler doit être inclus pour utiliser cette macro. Les instructions pour inclure ceci via SBT sont

ici .

Vous pouvez utiliser la macro définie ci-dessus comme ceci:

```
@noop
case class Foo(a: String, b: Int)

@noop
object Bar {
  def f(): String = "hello"
}

@noop
def g(): Int = 10
```

Méthode Macros

Lorsqu'une méthode est définie pour être une macro, le compilateur prend le code qui est passé en tant qu'argument et le transforme en AST. Il appelle ensuite l'implémentation de la macro avec cet AST, et renvoie un nouvel AST qui est ensuite renvoyé sur son site d'appel.

```
import reflect.macros.blackbox.Context

object Macros {
  // This macro simply sees if the argument is the result of an addition expression.
  // E.g. isAddition(1+1) and isAddition("a"+1).
  // but !isAddition(1+1-1), as the addition is underneath a subtraction, and also
  // !isAddition(x.+), and !isAddition(x.+(a,b)) as there must be exactly one argument.
  def isAddition(x: Any): Boolean = macro isAddition_impl

  // The signature of the macro implementation is the same as the macro definition,
  // but with a new Context parameter, and everything else is wrapped in an Expr.
  def isAddition_impl(c: Context)(expr: c.Expr[Any]): c.Expr[Boolean] = {
    import c.universe._ // The universe contains all the useful methods and types
    val plusName = TermName("+").encodedName // Take the name + and encode it as $plus
    expr.tree match { // Turn expr into an AST representing the code in isAddition(...)
      case Apply(Select(_, `plusName`), List(_)) => reify(true)
      // Pattern match the AST to see whether we have an addition
      // Above we match this AST
      //             Apply (function application)
      //            /     \
      //         Select  List(_) (exactly one argument)
      // (selection ^ of entity, basically the . in x.y)
      //      /          \
      //    _              \
      //               `plusName` (method named +)
      case _                                      => reify(false)
      // reify is a macro you use when writing macros
      // It takes the code given as its argument and creates an Expr out of it
    }
  }
}
```

Il est également possible d'avoir des macros qui prennent `Tree` s comme arguments. Comme comment `reify` est utilisé pour créer `Expr` s, l'interpolateur de chaînes `q` (pour quasiquote) nous permet de créer et de déconstruire `Tree` s. Notez que nous aurions pu utiliser `q` ci-dessus (

`expr.tree` is, surprise, un `Tree` lui-même) également, mais pas à des fins démonstratives.

```
// No Exprs, just Trees
def isAddition_impl(c: Context)(tree: c.Tree): c.Tree = {
  import c.universe._
  tree match {
    // q is a macro too, so it must be used with string literals.
    // It can destructure and create Trees.
    // Note how there was no need to encode + this time, as q is smart enough to do it itself.
    case q"${_} + ${_}" => q"true"
    case _              => q"false"
  }
}
```

Erreurs dans les macros

Les macros peuvent déclencher des avertissements et des erreurs de compilation à travers l'utilisation de leur `Context` .

Disons que nous sommes particulièrement zélés en ce qui concerne le code erroné et que nous voulons marquer chaque instance de dette technique avec un message d'information de compilation (ne pensons pas à la gravité de cette idée). Nous pouvons utiliser une macro qui ne fait rien sauf émettre un tel message.

```
import reflect.macros.blackbox.Context

def debtMark(message: String): Unit = macro debtMark_impl
def debtMarkImpl(c: Context)(message: c.Tree): c.Tree = {
  message match {
    case Literal(Constant(msg: String)) => c.info(c.enclosingPosition, msg, false)
    // false above means "do not force this message to be shown unless -verbose"
    case _                              => c.abort(c.enclosingPosition, "Message must be a
string literal.")
    // Abort causes the compilation to completely fail. It's not even a compile error, where
    // multiple can stack up; this just kills everything.
  }
  q"()" // At runtime this method does nothing, so we return ()
}
```

De plus, au lieu d'utiliser `???` pour marquer du code non implémenté, nous pouvons créer deux macros, `!!!` et `?!?` , qui servent le même objectif, mais émettent des avertissements du compilateur. `?!?` provoquera un avertissement, et `!!!` causera une erreur pure et simple.

```
import reflect.macros.blackbox.Context

def ?!? : Nothing = macro impl_?!?
def !!! : Nothing = macro impl_!!!

def impl_?!?(c: Context): c.Tree = {
  import c.universe._
  c.warning(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
  // If someone were to shadow the scala package, scala.Predef.??? would not work, as it
  // would end up referring to the scala that shadows and not the actual scala.
  // ROOTPKG is the very root of the tree, and acts like it is imported anew in every
```

```
  // expression. It is actually named _root_, but if someone were to shadow it, every
  // reference to it would be an error. It allows us to safely access ??? and know that
  // it is the one we want.
}

def impl_!!!(c: Context): c.Tree = {
  import c.universe._
  c.error(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
}
```

Lire Macros en ligne: https://riptutorial.com/fr/scala/topic/3808/macros

Chapitre 30: Manipulation XML

Examples

Beautify ou Pretty-Print XML

L'utilitaire `PrettyPrinter` imprimera les documents XML. L'extrait de code suivant permet d'imprimer des fichiers XML non formatés:

```
import scala.xml.{PrettyPrinter, XML}
val xml = XML.loadString("<a>Alana<b><c>Beth</c><d>Catie</d></b></a>")
val formatted = new PrettyPrinter(150, 4).format(xml)
print(formatted)
```

Cela affichera le contenu en utilisant une largeur de page de `150` et une constante d'indentation de `4` caractères d'espace blanc:

```
<a>
    Alana
    <b>
        <c>Beth</c>
        <d>Catie</d>
    </b>
</a>
```

Vous pouvez utiliser `XML.loadFile("nameoffile.xml")` pour charger xml à partir d'un fichier au lieu d'une chaîne.

Lire Manipulation XML en ligne: https://riptutorial.com/fr/scala/topic/1453/manipulation-xml

Chapitre 31: Mise en place de Scala

Examples

Sur Linux via dpkg

Sur les distributions basées sur Debian, y compris Ubuntu, la méthode la plus simple consiste à utiliser le fichier d'installation `.deb` . Allez sur le site de Scala . Choisissez la version que vous souhaitez installer, puis faites défiler et recherchez `scala-xxxdeb` .

Vous pouvez installer le scala deb à partir de la ligne de commande:

```
sudo dpkg -i scala-x.x.x.deb
```

Pour vérifier qu'il est correctement installé, dans l'invite de commande du terminal:

```
which scala
```

La réponse renvoyée doit être équivalente à celle que vous avez placée dans votre variable PATH. Pour vérifier que Scala fonctionne:

```
scala
```

Cela devrait démarrer la REPL Scala et signaler la version (qui, à son tour, devrait correspondre à la version que vous avez téléchargée).

Installation d'Ubuntu via le téléchargement manuel et la configuration

Téléchargez votre version préférée de Lightbend avec `curl` :

```
curl -O http://downloads.lightbend.com/scala/2.xx.x/scala-2.xx.x.tgz
```

Décompressez le fichier `tar` dans `/usr/local/share` ou `/opt/bin` :

```
unzip scala-2.xx.x.tgz
mv scala-2.xx.x /usr/local/share/scala
```

Ajoutez le `PATH` à `~/.profile` ou `~/.bash_profile` ou `~/.bashrc` en incluant ce texte dans l'un de ces fichiers:

```
$SCALA_HOME=/usr/local/share/scala
export PATH=$SCALA_HOME/bin:$PATH
```

Pour vérifier qu'il est correctement installé, dans l'invite de commande du terminal:

```
which scala
```

La réponse renvoyée doit être équivalente à celle que vous avez placée dans votre variable `PATH` . Pour vérifier que `scala` fonctionne:

```
scala
```

Cela devrait démarrer la REPL Scala et signaler la version (qui, à son tour, devrait correspondre à la version que vous avez téléchargée).

Mac OSX via Macports

Sur les ordinateurs Mac OSX avec MacPorts installés, ouvrez une fenêtre de terminal et tapez:

```
port list | grep scala
```

Cela listera tous les paquets disponibles pour Scala. Pour en installer un (dans cet exemple la version 2.11 de Scala):

```
sudo port install scala2.11
```

(Le `2.11` peut changer si vous souhaitez installer une version différente.)

Toutes les dépendances seront automatiquement installées et votre paramètre `$PATH` mis à jour. Pour vérifier que tout a fonctionné:

```
which scala
```

Cela vous montrera le chemin vers l'installation Scala.

```
scala
```

Cela ouvrira la Scala REPL et signalera le numéro de version installé.

Lire Mise en place de Scala en ligne: https://riptutorial.com/fr/scala/topic/2921/mise-en-place-de-scala

Chapitre 32: Monades

Examples

Monad Définition

De manière informelle, une monade est un conteneur d'éléments, notés `F[_]` , contenant 2 fonctions: `flatMap` (pour transformer ce conteneur) et `unit` (pour créer ce conteneur).

Les exemples courants de bibliothèque incluent `List[T]` , `Set[T]` et `Option[T]` .

Définition formelle

Monad `M` est un type paramétrique `M[T]` avec deux opérations `flatMap` et `unit` , telles que:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

Ces fonctions doivent satisfaire à trois lois:

1. *Associativité* : `(m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)`
   En d'autres termes, si la séquence reste inchangée, vous pouvez appliquer les termes dans n'importe quel ordre. Ainsi, appliquer `m` à `f` , puis appliquer le résultat à `g` donnera le même résultat que si l'on appliquait `f` à `g` , puis appliquer `m` à ce résultat.
2. *Unité de gauche* : `unit(x) flatMap f == f(x)`
   C'est-à-dire que la monade d'unité de `x` cartographiée à travers `f` équivaut à appliquer `f` à `x` .
3. *Unité droite* : `m flatMap unit == m`
   C'est une «identité»: toute monade à plat contre une unité retournera une monade équivalente à elle-même.

Exemple :

```
val m = List(1, 2, 3)
def unit(x: Int): List[Int] = List(x)
def f(x: Int): List[Int] = List(x * x)
def g(x: Int): List[Int] = List(x * x * x)
val x = 1
```

1. *Associativité* :

```
(m flatMap f).flatMap(g) == m.flatMap(x => f(x) flatMap g) //Boolean = true
//Left side:
List(1, 4, 9).flatMap(g) // List(1, 64, 729)
//Right side:
 m.flatMap(x => (x * x) * (x * x) * (x * x)) //List(1, 64, 729)
```

2. *Unité de gauche*

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. *Unité droite*

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```

Les collections standard sont des monades

La plupart des collections standard sont des monades (`List[T]` , `Option[T]`) ou des monades (E `Either[T]` , `Future[T]`). Ces collections peuvent être facilement combinées entre elles `for` compréhensions (qui sont une manière équivalente d'écrire des transformations `flatMap`):

```
val a = List(1, 2, 3)
val b = List(3, 4, 5)
for {
  i <- a
  j <- b
} yield(i * j)
```

Ce qui précède est équivalent à:

```
a flatMap {
  i => b map {
    j => i * j
  }
}
```

Étant donné qu'une monade préserve la *structure de données* et n'agit que sur les éléments de cette structure, nous pouvons sans fin mettre en place *des structures de données* monadiques, comme illustré ici pour une compréhension.

Lire Monades en ligne: https://riptutorial.com/fr/scala/topic/4112/monades

Chapitre 33: Opérateurs à Scala

Examples

Opérateurs intégrés

Scala a les opérateurs intégrés suivants (méthodes / éléments de langage avec des règles de priorité prédéfinies):

Type	symbole	Exemple
Opérateurs arithmétiques	`+ - * / %`	`a + b`
Opérateurs relationnels	`== != > < >= <=`	`a > b`
Opérateurs logiques	`&& & || | !`	`a && b`
Opérateurs sur les bits	`& | ^ ~ << >> >>>`	`a & b` , `~a` , `a >>> b`
Opérateurs d'affectation	`= += -= *= /= %= <<= >>= &= ^= |=`	`a += b`

Les opérateurs Scala ont la même signification qu'en Java

Remarque : les méthodes qui se terminent par `:` lier à droite (et à droite associative), l'appel avec `value :: list` `list.::(value)` peut être écrit en tant que `value :: list` avec la syntaxe de l'opérateur. (`1 :: 2 :: 3 :: Nil` est identique à `1 :: (2 :: (3 :: Nil))`)

Surcharge de l'opérateur

En Scala, vous pouvez définir vos propres opérateurs:

```
class Team {
   def +(member: Person) = ...
}
```

Avec les définitions ci-dessus, vous pouvez l'utiliser comme:

```
ITTeam + Jack
```

ou

```
ITTeam.+(Jack)
```

Pour définir des opérateurs unaires, vous pouvez le préfixer avec `unary_` . Par exemple `unary_!`

```
class MyBigInt {
```

```
   def unary_! = ...
}

var a: MyBigInt = new MyBigInt
var b = !a
```

Priorité de l'opérateur

Catégorie	Opérateur	Associativité
Postfix	`() []`	De gauche à droite
Unary	`! ~`	De droite à gauche
Multiplicatif	`* / %`	De gauche à droite
Additif	`+ -`	De gauche à droite
Décalage	`>> >>> <<`	De gauche à droite
Relationnel	`> >= < <=`	De gauche à droite
Égalité	`== !=`	De gauche à droite
Bitwise et	`&`	De gauche à droite
Xor binaire	`^`	De gauche à droite
Bitwise ou	`|`	De gauche à droite
Logique et	`&&`	De gauche à droite
Logique ou	`||`	De gauche à droite
Affectation	`= += -= *= /= %= >>= <<= &= ^= |=`	De droite à gauche
Virgule	`,`	De gauche à droite

La programmation en Scala donne le contour suivant basé sur le 1er caractère de l'opérateur. Par exemple `>` est le 1er caractère de l'opérateur `>>>` :

Opérateur
(tous les autres caractères spéciaux)
`* / %`
`+ -`
`:`

Opérateur
= !
< >
&
^
|
(toutes les lettres)
(tous les opérateurs d'affectation)

La seule exception à cette règle concerne les *opérateurs d'affectation* , par exemple `+=` , `*=` , etc. Si un opérateur se termine par un caractère égal (=) et n'est pas l'un des opérateurs de comparaison `<=` , `>=` , `==` ou `!=` , alors la priorité de l'opérateur est la même que la simple affectation. En d'autres termes, inférieure à celle de tout autre opérateur.

Lire Opérateurs à Scala en ligne: https://riptutorial.com/fr/scala/topic/6604/operateurs-a-scala

Chapitre 34: Paquets

Introduction

Les packages de Scala gèrent les espaces de noms dans les grands programmes. Par exemple, la `connection` nom peut se produire dans les packages `com.sql` et `org.http` . Vous pouvez utiliser les fonctions `com.sql.connection` et `org.http.connection` , respectivement, pour accéder à chacun de ces packages.

Examples

Structure du colis

```
package com {
    package utility {
        package serialization {
            class Serializer
            ...
        }
    }
}
```

Paquets et fichiers

La clause package n'est pas directement liée au fichier où elle se trouve. Il est possible de trouver des éléments communs de la clause package dans différents fichiers. Par exemple, les clauses du package ci-dessous peuvent être trouvées dans le fichier math1.scala et dans le fichier math2.scala.

Fichier math1.scala

```
package org {
    package math {
        package statistics {
            class Interval
        }
    }
}
```

Fichier math2.scala

```
package org {
    package math{
        package probability {
            class Density
        }
    }
}
```

Fichier study.scala

```
import org.math.probability.Density
import org.math.statistics.Interval

object Study {

    def main(args: Array[String]): Unit = {
        var a = new Interval()
        var b = new Density()
    }
}
```

Convection de nomenclature

Les packages Scala doivent suivre les conventions de nommage du package Java.
Les noms de package sont écrits en minuscules pour éviter les conflits avec les noms de classes ou d'interfaces. Les entreprises utilisent leur nom de domaine Internet inversé pour commencer leur nom de package - par exemple,

```
io.super.math
```

Lire Paquets en ligne: https://riptutorial.com/fr/scala/topic/8231/paquets

Chapitre 35: Paramétrage de type (génériques)

Examples

Le type d'option

Un bon exemple de type paramétré est le type Option . Il s'agit essentiellement de la définition suivante (avec plusieurs autres méthodes définies sur le type):

```
sealed abstract class Option[+A] {
  def isEmpty: Boolean
  def get: A

  final def fold[B](ifEmpty: => B)(f: A => B): B =
    if (isEmpty) ifEmpty else f(this.get)

  // lots of methods...
}

case class Some[A](value: A) extends Option[A] {
  def isEmpty = false
  def get = value
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

Nous pouvons également voir que cela a une méthode paramétrée, `fold` , qui retourne quelque chose de type `B`

Méthodes paramétrées

Le type de retour d'une méthode peut dépendre du *type* du paramètre. Dans cet exemple, `x` est le paramètre, `A` est le *type* de `x` , appelé *paramètre de type* .

```
def f[A](x: A): A = x

f(1)          // 1
f("two")      // "two"
f[Float](3)   // 3.0F
```

Scala utilisera l' inférence de type pour déterminer le type de retour, ce qui contraint les méthodes pouvant être appelées sur le paramètre. Ainsi, il faut faire attention: ce qui suit est une erreur de compilation car `*` n'est pas défini pour chaque type `A` :

```
def g[A](x: A): A = 2 * x  // Won't compile
```

Définir la liste des Ints

```
trait IntList { ... }

class Cons(val head: Int, val tail: IntList) extends IntList { ... }

class Nil extends IntList { ... }
```

mais que se passe-t-il si nous devons définir la liste des booléens, doubles, etc.?

Définition de la liste générique

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: [T], val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

class Nil[T] extends List[T] {
  def isEmpty: Boolean = true

  def head: Nothing = throw NoSuchElementException("Nil.head")

  def tail: Nothing = throw NoSuchElementException("Nil.tail")
}
```

Lire Paramétrage de type (génériques) en ligne:
https://riptutorial.com/fr/scala/topic/782/parametrage-de-type--generiques-

Chapitre 36: Pendant que les boucles

Syntaxe

- while (expression_booléenne) {expression_bloc}
- faire {expression_bloc} while (expression_booléenne)

Paramètres

Paramètre	Détails
`boolean_expression`	Toute expression qui sera évaluée comme `true` ou `false` .
`block_expression`	Toute expression ou ensemble d'expressions à évaluer si l' `boolean_expression` évaluée à `true` .

Remarques

La principale différence entre les boucles `while` et `do-while` est de savoir si elles exécutent l' `block_expression` avant de vérifier si elles doivent être bouclées.

Comme les boucles `while` et `do-while` reposent sur une expression à évaluer à `false` pour se terminer, elles nécessitent souvent un état mutable pour être déclarées en dehors de la boucle, puis modifiées dans la boucle.

Examples

Pendant que les boucles

```
var line = 0
var maximum_lines = 5

while (line < maximum_lines) {
  line = line + 1
  println("Line number: " + line)
}
```

Boucles Do-While

```
var line = 0
var maximum_lines = 5

do {
  line = line + 1
```

```
  println("Line number: " + line)
} while (line < maximum_lines)
```

Le `do` / `while` boucle est rarement utilisée dans la programmation fonctionnelle, mais peut être utilisé pour contourner le manque de soutien pour la `break` / `continue` construction, comme on le voit dans d' autres langues:

```
if(initial_condition) do if(filter) {
  ...
} while(continuation_condition)
```

Lire Pendant que les boucles en ligne: https://riptutorial.com/fr/scala/topic/650/pendant-que-les-boucles

Chapitre 37: Portée

Introduction

Scope on Scala définit où accéder à une valeur (`def` , `val` , `var` ou `class`).

Syntaxe

- déclaration
- déclaration privée
- privé [cette] déclaration
- déclaration privée [fromWhere]
- déclaration protégée
- déclaration protégée [fromWhere]

Examples

Portée publique (par défaut)

Par défaut, la portée est `public` , la valeur est accessible depuis n'importe où.

```
package com.example {
  class FooClass {
    val x = "foo"
  }
}

package an.other.package {
  class BarClass {
    val foo = new com.example.FooClass
    foo.x // <- Accessing a public value from another package
  }
}
```

Un champ privé

Lorsque l'étendue est privée, vous ne pouvez y accéder qu'à partir de la classe en cours ou d'autres instances de la classe en cours.

```
package com.example {
  class FooClass {
    private val x = "foo"
    def aFoo(otherFoo: FooClass) {
      otherFoo.x // <- Accessing from another instance of the same class
    }
  }
  class BarClass {
    val f = new FooClass
```

```
    f.x // <- This will not compile
  }
}
```

Une portée spécifique au package privé

Vous pouvez spécifier un package dans lequel la valeur privée est accessible.

```
package com.example {
  class FooClass {
    private val x = "foo"
    private[example] val y = "bar"
  }
  class BarClass {
    val f = new FooClass
    f.x // <- Will not compile
    f.y // <- Will compile
  }
}
```

Portée privée de l'objet

La portée la plus restrictive est la portée *"object-private"* , qui autorise uniquement l'accès à cette valeur à partir de la même instance de l'objet.

```
class FooClass {
  private[this] val x = "foo"
  def aFoo(otherFoo: FooClass) = {
    otherFoo.x // <- This will not compile, accessing x outside the object instance
  }
}
```

Portée protégée

La portée protégée permet d'accéder à la valeur depuis n'importe quelle sous-classe de la classe en cours.

```
class FooClass {
  protected val x = "foo"
}
class BarClass extends FooClass {
  val y = x // It is a subclass instance, will compile
}
class ClassB {
  val f = new FooClass
  f.x // <- This will not compile
}
```

Portée protégée du package

La portée protégée par le package permet d'accéder à la valeur uniquement à partir de n'importe quelle sous-classe d'un package spécifique.

```
package com.example {
  class FooClass {
    protected[example] val x = "foo"
  }
  class ClassB extends FooClass {
    val y = x // It's in the protected scope, will compile
  }
}
package com {
  class BarClass extends com.example.FooClass {
    val y = x // <- Outside the protected scope, will not compile
  }
}
```

Lire Portée en ligne: https://riptutorial.com/fr/scala/topic/9705/portee

Chapitre 38: Pour les expressions

Syntaxe

- for {clauses} body
- pour {clauses} Rendement
- for (clauses) body
- for (clauses) corps entier

Paramètres

Paramètre	Détails
pour	Mot clé requis pour utiliser une boucle / compréhension
clauses	L'itération et les filtres sur lesquels fonctionne pour.
rendement	Utilisez cette option si vous souhaitez créer ou «céder» une collection. Utiliser `yield` entraînera le type de retour du `for` à être une collection au lieu de `Unit` .
corps	Le corps de l'expression for, exécutée à chaque itération.

Examples

Basic For Loop

```
for (x <- 1 to 10)
  println("Iteration number " + x)
```

Cela démontre l'itération d'une variable, `x` , de `1` à `10` et de faire quelque chose avec cette valeur. Le type de retour de ceci `for` compréhension est `Unit` .

Basic For Comprehension

Cela démontre un filtre sur une boucle for, et l'utilisation de `yield` pour créer une «compréhension de séquence»:

```
for ( x <- 1 to 10 if x % 2 == 0)
  yield x
```

La sortie pour ceci est:

```
scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

Un pour la compréhension est utile lorsque vous devez créer une nouvelle collection basée sur l'itération et ses filtres.

Imbriqué pour boucle

Cela montre comment vous pouvez parcourir plusieurs variables:

```
for {
  x <- 1 to 2
  y <- 'a' to 'd'
} println("(" + x + "," + y + ")")
```

(Notez que `to` ici est une méthode d'opérateur infixe qui renvoie une plage inclusive . Voir la définition ici .)

Cela crée la sortie:

```
(1,a)
(1,b)
(1,c)
(1,d)
(2,a)
(2,b)
(2,c)
(2,d)
```

Notez qu'il s'agit d'une expression équivalente, utilisant des parenthèses au lieu de crochets:

```
for (
  x <- 1 to 2
  y <- 'a' to 'd'
) println("(" + x + "," + y + ")")
```

Afin d'obtenir toutes les combinaisons dans un seul vecteur, nous pouvons `yield` le résultat et le définir sur un `val` :

```
val a = for {
  x <- 1 to 2
  y <- 'a' to 'd'
} yield "(%s,%s)".format(x, y)
// a: scala.collection.immutable.IndexedSeq[String] = Vector((1,a), (1,b), (1,c), (1,d),
(2,a), (2,b), (2,c), (2,d))
```

Monadic pour les compréhensions

Si vous avez plusieurs objets de types monadiques , nous pouvons obtenir des combinaisons de valeurs en utilisant un «pour la compréhension»:

```
for {
   x <- Option(1)
   y <- Option("b")
   z <- List(3, 4)
```

```
} {
    // Now we can use the x, y, z variables
    println(x, y, z)
    x  // the last expression is *not* the output of the block in this case!
}

// This prints
// (1, "b", 3)
// (1, "b", 4)
```

Le type de retour de ce bloc est `Unit` .

Si les objets sont du *même* type monadique `M` (par exemple, `Option`), l'utilisation de `yield` retournera un objet de type `M` au lieu de `Unit` .

```
val a = for {
   x <- Option(1)
   y <- Option("b")
} yield {
    // Now we can use the x, y variables
    println(x, y)
    // whatever is at the end of the block is the output
    (7 * x, y)
}

// This prints:
// (1, "b")
// The val `a` is set:
// a: Option[(Int, String)] = Some((7,b))
```

Notez que le mot-clé de `yield` *ne peut pas* être utilisé dans l'exemple d'origine, où il existe un mélange de types monadiques (`Option` et `List`). Si vous essayez de le faire, vous obtenez une erreur d'incompatibilité de type à la compilation.

Itérer à travers des collections en utilisant une boucle For

Cela montre comment imprimer chaque élément d'une carte

```
val map = Map(1 -> "a", 2 -> "b")
for (number <- map) println(number) // prints (1,a), (2,b)
for ((key, value) <- map) println(value) // prints a, b
```

Cela montre comment imprimer chaque élément d'une liste

```
val list = List(1,2,3)
for(number <- list) println(number) // prints 1, 2, 3
```

Desugaring For Comprehensions

`for` compréhension à Scala, il n'y a que du sucre syntaxique . Ces compréhensions sont implémentées à l'aide des `withFilter` , `foreach` , `flatMap` et `map` de leurs types de sujet. Pour cette raison, seuls les types pour lesquels ces méthodes sont définies peuvent être utilisés `for`

compréhension.

A `for` compréhension de la forme suivante, avec des motifs `pN` , des générateurs `gN` et des conditions `cN` :

```
for(p0 <- x0 if g0; p1 <- g1 if c1) { ??? }
```

... dé-sucre aux appels imbriqués en utilisant `withFilter` et `foreach` :

```
g0.withFilter({ case p0 => c0  case _ => false }).foreach({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).foreach({
    case p1 => ???
  })
})
```

Considérant une expression `for` / `yield` de la forme suivante:

```
for(p0 <- g0 if c0; p1 <- g1 if c1) yield ???
```

... `withFilter` les appels imbriqués en utilisant `withFilter` et soit `flatMap` ou `map` :

```
g0.withFilter({ case p0 => c0  case _ => false }).flatMap({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).map({
    case p1 => ???
  })
})
```

(Notez que la `map` est utilisée dans la compréhension la plus profonde, et que `flatMap` est utilisé dans toute compréhension externe.)

Un `for` compréhension peut être appliqué à tout type mettant en œuvre les méthodes requises par la représentation sans sucre. Il n'y a pas de restrictions sur les types de retour de ces méthodes, tant qu'elles sont composables.

Lire Pour les expressions en ligne: https://riptutorial.com/fr/scala/topic/669/pour-les-expressions

Chapitre 39: Programmation au niveau du type

Examples

Introduction à la programmation au niveau du type

Si nous considérons une liste hétérogène, dans laquelle les éléments de la liste ont des types variés mais connus, il peut être souhaitable de pouvoir effectuer des opérations sur les éléments de la liste collectivement sans ignorer les informations de type des éléments. L'exemple suivant implémente une opération de mappage sur une liste hétérogène simple.

Comme le type d'élément varie, la classe d'opérations que nous pouvons effectuer est restreinte à une forme de projection de type. Nous définissons donc un trait `Projection` `type Apply[A]` abstrait `type Apply[A]` calculant le *type* de résultat de la projection et `def apply[A](a: A): Apply[A]` calculant la *valeur* de résultat de la projection.

```
trait Projection {
  type Apply[A] // <: Any
  def apply[A](a: A): Apply[A]
}
```

En implémentant le `type Apply[A]` nous programmons au niveau du type (par opposition au niveau de la valeur).

Notre type de liste hétérogène définit une opération de `map` paramétrée par la projection souhaitée ainsi que par le type de projection. Le résultat de l'opération de carte est abstrait, variera selon la classe et la projection, et doit toujours être une liste `HList` :

```
sealed trait HList {
  type Map[P <: Projection] <: HList
  def map[P <: Projection](p: P): Map[P]
}
```

Dans le cas de `HNil` , la liste hétérogène vide, le résultat de toute projection sera toujours lui-même. Ici, nous déclarons le `trait HNil` comme une commodité afin que nous puissions écrire `HNil` comme un type au lieu de `HNil.type` :

```
sealed trait HNil extends HList
case object HNil extends HNil {
  type Map[P <: Projection] = HNil
  def map[P <: Projection](p: P): Map[P] = HNil
}
```

`HCons` sont la liste hétérogène non vide. Ici, nous affirmons que lorsqu'on applique une opération de carte, le type de tête résultant est celui qui résulte de l'application de la projection à la valeur

de tête (`P#Apply[H]`) et que le type de queue résultant projection sur la queue (`T#Map[P]`), connue pour être une `HList` :

```
case class HCons[H, T <: HList](head: H, tail: T) extends HList {
  type Map[P <: Projection] = HCons[P#Apply[H], T#Map[P]]
  def map[P <: Projection](p: P): Map[P] = HCons(p.apply(head), tail.map(p))
}
```

La projection la plus évidente consiste à exécuter une opération d'emballage - l'exemple suivant fournit une instance de `HCons[Option[String], HCons[Option[Int], HNil]]` :

```
HCons("1", HCons(2, HNil)).map(new Projection {
  type Apply[A] = Option[A]
  def apply[A](a: A): Apply[A] = Some(a)
})
```

Lire Programmation au niveau du type en ligne:
https://riptutorial.com/fr/scala/topic/3738/programmation-au-niveau-du-type

Chapitre 40: Quasiquotes

Examples

Créer un arbre de syntaxe avec des quasi-notes

Utilisez des quasi-notes pour créer un `Tree` dans une macro.

```
object macro {
  def addCreationDate(): java.util.Date = macro impl.addCreationDate
}

object impl {
  def addCreationDate(c: Context)(): c.Expr[java.util.Date] = {
    import c.universe._

    val date = q"new java.util.Date()" // this is the quasiquote
    c.Expr[java.util.Date](date)
  }
}
```

Il peut être arbitrairement complexe mais il sera validé pour une syntaxe Scala correcte.

Lire Quasiquotes en ligne: https://riptutorial.com/fr/scala/topic/4032/quasiquotes

Chapitre 41: Récursivité

Examples

Récursion de la queue

En utilisant la récursivité régulière, chaque appel récursif pousse une autre entrée sur la pile d'appels. Lorsque la récursivité est terminée, l'application doit désactiver chaque entrée. S'il y a beaucoup d'appels de fonctions récursives, cela peut aboutir à une énorme pile.

Scala supprime automatiquement la récursivité au cas où elle trouve l'appel récursif en position de queue. L'annotation (`@tailrec`) peut être ajoutée aux fonctions récursives pour garantir que l'optimisation de l'appel de la queue est effectuée. Le compilateur affiche alors un message d'erreur s'il ne peut pas optimiser votre récursivité.

Récursion régulière

Cet exemple n'est pas récursif en queue car lorsque l'appel récursif est effectué, la fonction doit garder une trace de la multiplication nécessaire avec le résultat après le retour de l'appel.

```
def fact(i : Int) : Int = {
   if(i <= 1) i
   else i * fact(i-1)
}

println(fact(5))
```

L'appel de fonction avec le paramètre entraînera une pile qui ressemble à ceci:

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 * 1)))))
(* 5 (* 4 (* 3 (* 2))))
(* 5 (* 4 (* 6)))
(* 5 (* 24))
120
```

Si nous essayons d’annoter cet exemple avec `@tailrec` nous aurons le message d’erreur suivant:
`could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position`

Récursion de la queue

En récursivité, vous effectuez d'abord vos calculs, puis vous exécutez l'appel récursif, en passant

les résultats de votre étape actuelle à l'étape récursive suivante.

```
def fact_with_tailrec(i : Int) : Long = {
   @tailrec
   def fact_inside(i : Int, sum: Long) : Long = {
      if(i <= 1) sum
      else fact_inside(i-1,sum*i)
   }
   fact_inside(i,1)
}

println(fact_with_tailrec(5))
```

En revanche, la trace de la pile pour la factorielle récursive de la queue ressemble à ceci:

```
(fact_with_tailrec 5)
(fact_inside 5 1)
(fact_inside 4 5)
(fact_inside 3 20)
(fact_inside 2 60)
(fact_inside 1 120)
```

Il suffit de garder la trace de la même quantité de données pour chaque appel à `fact_inside` car la fonction renvoie simplement la valeur qui lui a été `fact_inside` . Cela signifie que même si `fact_with_tail 1000000` est appelé, il ne nécessite que la même quantité d'espace que `fact_with_tail 3` . Ce n'est pas le cas avec l'appel non récursif, et de telles valeurs peuvent provoquer un débordement de pile.

Récursivité sans empilement avec trampoline (scala.util.control.TailCalls)

Il est très courant d'obtenir une erreur `StackOverflowError` lors de l'appel d'une fonction récursive. La bibliothèque standard Scala offre TailCall pour éviter le débordement de la pile en utilisant des objets de pile et des continuations pour stocker l'état local de la récursivité.

Deux exemples du scaladoc de TailCalls

```
import scala.util.control.TailCalls._

def isEven(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(true) else tailcall(isOdd(xs.tail))

def isOdd(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(false) else tailcall(isEven(xs.tail))

// Does this List contain an even number of elements?
isEven((1 to 100000).toList).result

def fib(n: Int): TailRec[Int] =
  if (n < 2) done(n) else for {
    x <- tailcall(fib(n - 1))
    y <- tailcall(fib(n - 2))
  } yield (x + y)

// What is the 40th entry of the Fibonacci series?
```

```
fib(40).result
```

Lire Récursivité en ligne: https://riptutorial.com/fr/scala/topic/3889/recursivite

Chapitre 42: Réflexion

Examples

Charger une classe en utilisant la réflexion

```
import scala.reflect.runtime.universe._
val mirror = runtimeMirror(getClass.getClassLoader)
val module = mirror.staticModule("org.data.TempClass")
```

Lire Réflexion en ligne: https://riptutorial.com/fr/scala/topic/5824/reflexion

Chapitre 43: Ruisseaux

Remarques

Les flux sont évalués avec précaution, ce qui signifie qu'ils peuvent être utilisés pour implémenter des générateurs, qui fourniront ou «généreront» un nouvel élément du type spécifié à la demande, plutôt qu'avant le fait. Cela garantit que seuls les calculs nécessaires sont effectués.

Examples

Utiliser un flux pour générer une séquence aléatoire

`genRandom` crée un flux de nombres aléatoires qui a une chance sur quatre de se terminer à chaque appel.

```
def genRandom: Stream[String] = {
  val random = scala.util.Random.nextFloat()
  println(s"Random value is: $random")
  if (random < 0.25) {
    Stream.empty[String]
  } else {
    ("%.3f : A random number" format random) #:: genRandom
  }
}

lazy val randos = genRandom  // getRandom is lazily evaluated as randos is iterated through

for {
  x <- randos
} println(x) // The number of times this prints is effectively randomized.
```

Notez la construction `#::` qui *récursivement paresseuse* : comme elle ajoute le nombre aléatoire actuel à un flux, elle n'évalue pas le reste du flux jusqu'à ce qu'il soit itéré.

Flux infinis via la récursivité

On peut construire des flux qui se réfèrent eux-mêmes et deviennent ainsi infiniment récursifs.

```
// factorial
val fact: Stream[BigInt] = 1 #:: fact.zipWithIndex.map{case (p,x)=>p*(x+1)}
fact.take(10)  // (1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
fact(24)       // 620448401733239439360000

// the Fibonacci series
val fib: Stream[BigInt] = 0 #:: fib.scan(1:BigInt)(_+_)
fib.take(10)  // (0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
fib(124)      // 36726740705505779255899443

// random Ints between 10 and 99 (inclusive)
def rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(10)  // (20, 95, 14, 44, 42, 78, 85, 24, 99, 85)
```

Dans ce contexte, la différence entre Var, Val et Def est intéressante. En tant que `def` chaque élément est recalculé chaque fois qu'il est référencé. En tant que `val` chaque élément est conservé et réutilisé après avoir été calculé. Cela peut être démontré en créant un effet secondaire à chaque calcul.

```
// def with extra output per calculation
def fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!!!!!!!!!!! 120
fact(4)  // !!!!!!!!!! 24
fact(7)  // !!!!!!!!!!!!!!!!!!!!!!!!!!!! 5040

// now as val
val fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!! 120
fact(4)  // 24
fact(7)  // !! 5040
```

Cela explique également pourquoi le nombre aléatoire `Stream` ne fonctionne pas comme un `val` .

```
val rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(5)  // (79, 79, 79, 79, 79)
```

Flux infini d'auto-référence

```
// Generate stream that references itself in its evaluation
lazy val primes: Stream[Int] =
  2 #:: Stream.from(3, 2)
    .filter { i => primes.takeWhile(p => p * p <= i).forall(i % _ != 0) }
    .takeWhile(_ > 0) // prevent overflowing

// Get list of 10 primes
assert(primes.take(10).toList == List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29))

// Previously calculated values were memoized, as shown by toString
assert(primes.toString == "Stream(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ?)")
```

Lire Ruisseaux en ligne: https://riptutorial.com/fr/scala/topic/3702/ruisseaux

Chapitre 44: Scala.js

Introduction

`Scala.js` est un port de `Scala` qui compile en `JavaScript` , qui à la fin sera exécuté en dehors de la `JVM` . Il présente des avantages comme le typage fort, l'optimisation du code au moment de la compilation, l'interopérabilité totale avec les bibliothèques JavaScript.

Examples

console.log dans Scala.js

```
println("Hello Scala.js") // In ES6: console.log("Hello Scala.js");
```

Fonctions de flèche grasse

```
val lastNames = people.map(p => p.lastName)
// Or shorter:
val lastNames = people.map(_.lastName)
```

Classe simple

```
class Person(val firstName: String, val lastName: String) {
  def fullName(): String =
    s"$firstName $lastName"
}
```

Collections

```
val personMap = Map(
  10 -> new Person("Roger", "Moore"),
  20 -> new Person("James", "Bond")
)
val names = for {
  (key, person) <- personMap
  if key > 15
} yield s"$key = ${person.firstName}"
```

Manipuler DOM

```
import org.scalajs.dom
import dom.document

def appendP(target: dom.Node, text: String) = {
  val pNode = document.createElement("p")
  val textNode = document.createTextNode(text)
  pNode.appendChild(textNode)
```

```
  target.appendChild(pNode)
}
```

Utiliser avec SBT

Sbt dépendance

```
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1" // (Triple %%%)
```

Fonctionnement

```
sbt run
```

En cours d'exécution avec une compilation continue:

```
sbt ~run
```

Compiler en un seul fichier JavaScript:

```
sbt fastOptJS
```

Lire Scala.js en ligne: https://riptutorial.com/fr/scala/topic/9426/scala-js

Chapitre 45: Scaladoc

Syntaxe

- Va au-dessus des méthodes, des champs, des classes ou des packages.
- Commence par `/**`
- Chaque ligne a un début `*` procédant avec les commentaires
- Se termine par `*/`

Paramètres

Paramètre	Détails
Classe spécifique	_
`@constructor detail`	Explique le constructeur principal de la classe
Méthode spécifique	_
`@return detail`	Détails sur ce qui est retourné sur la méthode.
Balises Method, Constructor et / ou Class	_
`@param x detail`	Détails sur le paramètre de valeur `x` sur une méthode ou un constructeur.
`@tparam x detail`	Détails sur le paramètre de type `x` sur une méthode ou un constructeur.
`@throws detail`	Quelles exceptions peuvent être levées.
Usage	_
`@see detail`	Fait référence à d'autres sources d'informations.
`@note detail`	Ajoute une note pour les conditions pré ou post, ou toute autre restriction ou attente notable.
`@example detail`	Fournit un exemple de code ou un exemple de documentation associée.
`@usecase detail`	Fournit une définition de méthode simplifiée lorsque la définition complète de la méthode est trop complexe ou trop bruyante.
Autre	_

Paramètre	Détails
`@author detail`	Fournit des informations sur l'auteur des éléments suivants.
`@version detail`	Fournit la version à laquelle appartient cette partie.
`@deprecated detail`	Marque l'entité suivante comme étant obsolète.

Examples

Scaladoc simple à la méthode

```
/**
  * Explain briefly what method does here
  * @param x Explain briefly what should be x and how this affects the method.
  * @param y Explain briefly what should be y and how this affects the method.
  * @return Explain what is returned from execution.
  */
def method(x: Int, y: String): Option[Double] = {
  // Method content
}
```

Lire Scaladoc en ligne: https://riptutorial.com/fr/scala/topic/4518/scaladoc

Chapitre 46: scalaz

Introduction

Scalaz est une bibliothèque Scala pour la programmation fonctionnelle.

Il fournit des structures de données purement fonctionnelles pour compléter celles de la bibliothèque standard Scala. Il définit un ensemble de classes de type fondamentales (par exemple, `Functor` , `Monad`) et les instances correspondantes pour un grand nombre de structures de données.

Examples

ApplyUsage

```
import scalaz._
import Scalaz._

scala> Apply[Option].apply2(some(1), some(2))((a, b) => a + b)
res0: Option[Int] = Some(3)

scala> val intToString: Int => String = _.toString

scala> Apply[Option].ap(1.some)(some(intToString))
res1: Option[String] = Some(1)

scala> Apply[Option].ap(none)(some(intToString))
res2: Option[String] = None

scala> val double: Int => Int = _ * 2

scala> Apply[List].ap(List(1, 2, 3))(List(double))
res3: List[Int] = List(2, 4, 6)

scala> :kind Apply
scalaz.Apply's kind is X[F[A]]
```

FunctorUsage

```
import scalaz._
import Scalaz._
scala> val len: String => Int = _.length
len: String => Int = $$Lambda$1164/96982033@7e758f40

scala> Functor[Option].map(Some("foo"))(len)
res0: Option[Int] = Some(3)

scala> Functor[Option].map(None)(len)
res1: Option[Int] = None

scala> Functor[List].map(List("qwer", "adsfg"))(len)
res2: List[Int] = List(4, 5)
```

```
scala> :kind Functor
scalaz.Functor's kind is X[F[A]]
```

ArrowUsage

```
import scalaz._
import Scalaz._
scala> val plus1 = (_: Int) + 1
plus1: Int => Int = $$Lambda$1167/111311649@6a6bfd97

scala> val plus2 = (_: Int) + 2
plus2: Int => Int = $$Lambda$1168/9243295480@6bbe050f

scala> val rev = (_: String).reverse
rev: String => String = $$Lambda$1227/1278001332@72685b74

scala> plus1.first apply (1, "abc")
res0: (Int, String) = (2,abc)

scala> plus1.second apply ("abc", 2)
res1: (String, Int) = (abc,3)

scala> rev.second apply (1, "abc")
res2: (Int, String) = (1,cba)

scala> plus1 *** rev apply(7, "abc")
res3: (Int, String) = (8,cba)

scala> plus1 &&& plus2 apply 7
res4: (Int, Int) = (8,9)

scala> plus1.product apply (1, 2)
res5: (Int, Int) = (2,3)

scala> :kind Arrow
scalaz.Arrow's kind is X[F[A1,A2]]
```

Lire scalaz en ligne: https://riptutorial.com/fr/scala/topic/9893/scalaz

Chapitre 47: Si des expressions

Examples

Expressions de base si

Dans Scala (contrairement à Java et à la plupart des autres langages), `if` est une **expression** au lieu d'une *déclaration* . Indépendamment, la syntaxe est identique:

```
if(x < 1984) {
   println("Good times")
} else if(x == 1984) {
   println("The Orwellian Future begins")
} else {
   println("Poor guy...")
}
```

L'implication de `if` être une expression est que vous pouvez affecter le résultat de l'évaluation de l'expression à une variable:

```
val result = if(x > 0) "Greater than 0" else "Less than or equals 0"
\\ result: String = Greater than 0
```

Nous voyons ci-dessus que l'expression `if` est évaluée et que le `result` est défini sur cette valeur.

Le type de retour d'une `if` expression est le **super** - **type** de toutes les branches logiques. Cela signifie que pour cet exemple, le type de retour est une `String` . Étant donné que tous `if` les expressions renvoient une valeur (par exemple , comme `if` la déclaration qui n'a pas d' `else` logique de branche), il est possible que le type de retour est `Any` :

```
val result = if(x > 0) "Greater than 0"
// result: Any = Greater than 0
```

Si aucune valeur ne peut être retournée (comme si seuls les effets secondaires comme `println` sont utilisés dans les branches logiques), le type de retour sera `Unit` :

```
val result = if(x > 0) println("Greater than 0")
// result: Unit = ()
```

`if` expressions dans Scala sont similaires à celles de l' opérateur ternaire dans les fonctions Java . En raison de cette similitude, il n'ya pas d'opérateur de ce type à Scala: il serait redondant.

> Les accolades peuvent être omis dans une `if` expression si le contenu est une seule ligne.

Lire Si des expressions en ligne: https://riptutorial.com/fr/scala/topic/4171/si-des-expressions

Chapitre 48: Surcharge de l'opérateur

Examples

Définition d'opérateurs d'infix personnalisés

Dans Scala, les opérateurs (tels que `+` , `-` , `*` , `++` , etc.) ne sont que des méthodes. Par exemple, `1 + 2` peut être écrit comme suit: `1.+(2)` . Ces sortes de méthodes sont appelées *«opérateurs infixes»* .

Cela signifie que des méthodes personnalisées peuvent être définies sur vos propres types, en réutilisant ces opérateurs:

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
}
```

Ces opérateurs définis comme méthodes peuvent être utilisés comme ceci:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val b = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))

// could also be written a.+(b)
val sum = a + b
```

Notez que les opérateurs infixes ne peuvent avoir qu'un seul argument. l'objet avant l'opérateur appellera son propre opérateur sur l'objet après l'opérateur. Toute méthode Scala avec un seul argument peut être utilisée comme opérateur infixe.

> Cela devrait être utilisé avec parcimonie. Il est généralement considéré comme une bonne pratique que si votre propre méthode fait exactement ce que l'on attend d'un opérateur. En cas de doute, utilisez un nom plus conservateur, comme `add` au lieu de `+` .

Définition d'opérateurs unitaires personnalisés

Les opérateurs unaires peuvent être définis en ajoutant l'opérateur `unary_` . Les opérateurs `unary_+` sont limités à `unary_+` , `unary_-` , `unary_!` et `unary_~` :

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)
```

```
    new Matrix(rows, cols, newData)
  }

  def unary_- = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) * -1

    new Matrix(rows, cols, newData)
  }
}
```

L'opérateur unaire peut être utilisé comme suit:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val negA = -a
```

> Cela devrait être utilisé avec parcimonie. Surcharger un opérateur unaire avec une définition qui n'est pas ce à quoi on pourrait s'attendre peut entraîner une confusion du code.

Lire Surcharge de l'opérateur en ligne: https://riptutorial.com/fr/scala/topic/2271/surcharge-de-l-operateur

Chapitre 49: Symboles littéraux

Remarques

Scala est livré avec un concept de **symboles** - des chaînes qui sont *internées* , c'est-à-dire que deux symboles du même nom (la même séquence de caractères), contrairement aux chaînes, feront référence au même objet pendant l'exécution.

Les symboles sont une caractéristique de nombreuses langues: Lisp, Ruby et Erlang et plus encore, mais dans Scala, ils sont relativement peu utilisés. Bonne fonctionnalité à avoir néanmoins.

Utilisation:

Tout littéral commençant par une seule citation ' , suivi d'un ou plusieurs chiffres, lettres ou sous-scores _ est un littéral de symbole. Le premier caractère est une exception car il ne peut pas s'agir d'un chiffre.

Bonnes définitions:

```
'ATM
'IPv4
'IPv6
'map_to_operations
'data_format_2006

// Using the `Symbol.apply` method

Symbol("hakuna matata")
Symbol("To be or not to be that is a question")
```

Mauvaises définitions:

```
'8'th_division
'94_pattern
'bad-format
```

Examples

Remplacement des chaînes dans les clauses cas

Disons que nous avons plusieurs sources de données qui incluent la *base de données, le fichier, prompt* et *argumentList* . En fonction de la source choisie, nous modifions notre approche:

```
def loadData(dataSource: Symbol): Try[String] = dataSource match {
  case 'database => loadDatabase() // Loading data from database
  case 'file =>  loadFile() // Loading data from file
  case 'prompt => askUser() // Asking user for data
```

```
  case 'argumentList => argumentListExtract() // Accessing argument list for data
  case _ => Failure(new Exception("Unsupported data source"))
}
```

Nous aurions pu très bien utiliser `String` à la place de `Symbol` . Nous ne l'avons pas fait, car aucune des fonctionnalités des chaînes n'est utile dans ce contexte.

Cela rend le code plus simple et moins sujet aux erreurs.

Lire Symboles littéraux en ligne: https://riptutorial.com/fr/scala/topic/6419/symboles-litteraux

Chapitre 50: synchronisé

Syntaxe

- `objectToSynchronizeOn.synchronized { /* code to run */}`
- `synchronized {/* code to run, can be suspended with wait */}`

Examples

synchroniser sur un objet

`synchronized` est une construction de concurrence de bas niveau qui peut empêcher plusieurs threads d'accéder aux mêmes ressources. Introduction pour la JVM utilisant le langage Java .

```
anInstance.synchronized {
  // code to run when the intristic lock on `anInstance` is acquired
  // other thread cannot enter concurrently unless `wait` is called on `anInstance` to suspend
  // other threads can continue of the execution of this thread if they `notify` or
`notifyAll` `anInstance`'s lock
}
```

En cas d' `object` il peut se synchroniser sur la classe de l'objet, pas sur l'instance singleton.

synchroniser implicitement sur ce

```
/* within a class, def, trait or object, but not a constructor */
synchronized {
  /* code to run when an intrisctic lock on `this` is acquired */
  /* no other thread can get the this lock unless execution is suspended with
   * `wait` on `this`
   */
}
```

Lire synchronisé en ligne: https://riptutorial.com/fr/scala/topic/3371/synchronise

Chapitre 51: Test avec ScalaTest

Examples

Bonjour test de spécification mondiale

Créez une classe testing dans le répertoire `src/test/scala` , dans un fichier nommé `HelloWorldSpec.scala` . Mettez ceci dans le fichier:

```
import org.scalatest.{FlatSpec, Matchers}

class HelloWorldSpec extends FlatSpec with Matchers {

  "Hello World" should "not be an empty String" in {
      val helloWorld = "Hello World"
      helloWorld should not be ("")
  }
}
```

- Cet exemple utilise `FlatSpec` et `Matchers` , qui font partie de la bibliothèque ScalaTest.
- `FlatSpec` permet aux tests d'être écrits dans le style **BDD (Behavior Driven Development)** . Dans ce style, une phrase est utilisée pour décrire le comportement attendu d'une unité de code donnée. Le test confirme que le code adhère à ce comportement. Voir la documentation pour plus d'informations .

Spec Cheatsheet de test

Installer

Les tests ci-dessous utilisent ces valeurs pour les exemples.

```
val helloWorld = "Hello World"
val helloWorldCount = 1
val helloWorldList = List("Hello World", "Bonjour Le Monde")
def sayHello = throw new IllegalStateException("Hello World Exception")
```

Vérification de type

Pour vérifier le type de donnée `val` :

```
helloWorld shouldBe a [String]
```

Notez que les parenthèses ici sont utilisées pour obtenir le type `String` .

Contrôle égal

Pour tester l'égalité:

```
helloWorld shouldEqual "Hello World"
helloWorld should === ("Hello World")
helloWorldCount shouldEqual 1
helloWorldCount shouldBe 1
helloWorldList shouldEqual List("Hello World", "Bonjour Le Monde")
helloWorldList === List("Hello World", "Bonjour Le Monde")
```

Pas de contrôle égal

Pour tester l'inégalité:

```
helloWorld should not equal "Hello"
helloWorld !== "Hello"
helloWorldCount should not be 5
helloWorldList should not equal List("Hello World")
helloWorldList !== List("Hello World")
helloWorldList should not be empty
```

Vérification de la longueur

Pour vérifier la longueur et / ou la taille:

```
helloWorld should have length 11
helloWorldList should have size 2
```

Vérification des exceptions

Pour vérifier le type et le message d'une exception:

```
val exception = the [java.lang.IllegalStateException] thrownBy {
  sayHello
}
exception.getClass shouldEqual classOf[java.lang.IllegalStateException]
exception.getMessage should include ("Hello World")
```

Inclure la bibliothèque ScalaTest avec SBT

En utilisant SBT pour gérer la dépendance de la bibliothèque , ajoutez ceci à `build.sbt` :

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.0"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.0" % "test"
```

Plus de détails peuvent être trouvés sur le site ScalaTest .

Lire Test avec ScalaTest en ligne: https://riptutorial.com/fr/scala/topic/5506/test-avec-scalatest

Chapitre 52: Tests avec ScalaCheck

Introduction

ScalaCheck est une bibliothèque écrite en Scala et utilisée pour des tests automatisés basés sur les propriétés de programmes Scala ou Java. ScalaCheck a été inspiré à l'origine par la bibliothèque QuickCheck de la bibliothèque Haskell, mais s'est également aventuré dans la sienne.

ScalaCheck n'a pas d'autres dépendances externes que l'exécution de Scala et fonctionne parfaitement avec l'outil de génération Scala. Il est également totalement intégré dans les frameworks de test ScalaTest et specs2.

Examples

Scalacheck avec scalatest et messages d'erreur

Exemple d'utilisation de scalacheck avec scalatest. Ci-dessous, nous avons quatre tests:

- "show pass example" - ça passe
- "affichez un exemple simple sans message d'erreur personnalisé" - le message a simplement échoué sans les détails, l'opérateur `&&` boolean est utilisé
- "show example with error messages on argument" - message d'erreur sur l'argument (`"argument" |:` :) La méthode Props.all est utilisée à la place de `&&`
- "show example with error messages on command" - message d'erreur sur la commande (`"command" |:` :) La méthode Props.all est utilisée à la place de `&&`

```
import org.scalatest.prop.Checkers
import org.scalatest.{Matchers, WordSpecLike}

import org.scalacheck.Gen._
import org.scalacheck.Prop._
import org.scalacheck.Prop

object Splitter {
  def splitLineByColon(message: String): (String, String) = {
    val (command, argument) = message.indexOf(":") match {
      case -1 =>
        (message, "")
      case x: Int =>
        (message.substring(0, x), message.substring(x + 1))
    }
    (command.trim, argument.trim)
  }

  def splitLineByColonWithBugOnCommand(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim + 2, argument.trim)
  }
```

```
  def splitLineByColonWithBugOnArgument(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim, argument.trim + 2)
  }
}

class ScalaCheckSpec extends WordSpecLike with Matchers with Checkers {

  private val COMMAND_LENGTH = 4

  "ScalaCheckSpec " should {
```

```
    "show pass example" in {
      check {
        Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
          (chars, expArgument) =>
            val expCommand = new String(chars.toArray)
            val line = s"$expCommand:$expArgument"
            val (c, p) = Splitter.splitLineByColon(line)
            Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
        }

      }
    }
```

```
"show simple example without custom error message " in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        c === expCommand && expArgument === p
    }

  }
}
```

```
"show example with error messages on argument" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

```
"show example with error messages on command" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnCommand(line)
```

```
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

La sortie (fragments):

```
[info] - should show example // passed
[info] - should show simple example without custom error message  *** FAILED ***
[info]    (ScalaCheckSpec.scala:73)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:73)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info] - should show example with error messages on argument *** FAILED ***
[info]    (ScalaCheckSpec.scala:86)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:86)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "" but got "2"
[info]       argument
[info] - should show example with error messages on command *** FAILED ***
[info]    (ScalaCheckSpec.scala:99)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:99)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "2" but got ""
[info]       command
```

Lire Tests avec ScalaCheck en ligne: https://riptutorial.com/fr/scala/topic/9430/tests-avec-scalacheck

Chapitre 53: Traits

Syntaxe

- trait ATrait {...}
- classe AClass (...) étend ATrait {...}
- classe AClass étend BClass avec ATrait
- classe AClass étend ATrait avec BTrait
- classe AClass étend ATrait avec BTrait avec CTrait
- classe ATrait étend BTrait

Examples

Modification empilable avec des traits

Vous pouvez utiliser des traits pour modifier les méthodes d'une classe, en utilisant des traits empilables.

L'exemple suivant montre comment les traits peuvent être empilés. L'ordre des traits est important. En utilisant un ordre différent de traits, un comportement différent est obtenu.

```
class Ball {
  def roll(ball : String) = println("Rolling : " + ball)
}

trait Red extends Ball {
  override def roll(ball : String) = super.roll("Red-" + ball)
}

trait Green extends Ball {
  override def roll(ball : String) = super.roll("Green-" + ball)
}

trait Shiny extends Ball {
  override def roll(ball : String) = super.roll("Shiny-" + ball)
}

object Balls {
  def main(args: Array[String]) {
    val ball1 = new Ball with Shiny with Red
    ball1.roll("Ball-1") // Rolling : Shiny-Red-Ball-1

    val ball2 = new Ball with Green with Shiny
    ball2.roll("Ball-2") // Rolling : Green-Shiny-Ball-2
  }
}
```

Notez que `super` est utilisé pour invoquer la méthode `roll()` dans les deux traits. Ce n'est qu'ainsi que nous pourrons réaliser des modifications empilables. En cas de modification empilable, l'ordre d'invocation de la méthode est déterminé par la *règle de linéarisation* .

Les bases du trait

C'est la version la plus basique d'un trait de Scala.

```
trait Identifiable {
  def getIdentifier: String
  def printIndentification(): Unit = println(getIdentifier)
}

case class Puppy(id: String, name: String) extends Identifiable {
  def getIdentifier: String = s"$name has id $id"
}
```

Comme aucune super classe n'est déclarée pour trait `Identifiable` , elle s'étend par défaut de la classe `AnyRef` . Comme aucune définition de `getIdentifier` n'est fournie dans `Identifiable` , la classe `Puppy` doit l'implémenter. Cependant, `Puppy` hérite de l'implémentation de `printIdentification` d' `Identifiable` .

Dans le REPL:

```
val p = new Puppy("K9", "Rex")
p.getIdentifier  // res0: String = Rex has id K9
p.printIndentification()  // Rex has id K9
```

Résoudre le problème du diamant

Le problème des diamants , ou héritage multiple, est géré par Scala à l'aide de Traits, qui sont similaires aux interfaces Java. Les caractéristiques sont plus flexibles que les interfaces et peuvent inclure des méthodes implémentées. Cela rend les traits similaires aux mixins dans d'autres langues.

Scala ne prend pas en charge l'héritage de plusieurs classes, mais un utilisateur peut étendre plusieurs caractéristiques dans une seule classe:

```
trait traitA {
  def name = println("This is the 'grandparent' trait.")
}

trait traitB extends traitA {
  override def name = {
    println("B is a child of A.")
    super.name
  }

}

trait traitC extends traitA {
  override def name = {
    println("C is a child of A.")
    super.name
  }
}
```

```
object grandChild extends traitB with traitC

grandChild.name
```

Ici, `grandChild` hérite à la fois de `traitB` et de `traitC` , qui héritent à leur tour de `traitA` . La sortie (ci-dessous) montre également l'ordre de priorité lors de la résolution des implémentations de méthode appelées en premier:

```
C is a child of A.
B is a child of A.
This is the 'grandparent' trait.
```

Notez que lorsque `super` est utilisé pour invoquer des méthodes en `class` ou en `trait` , la règle de linéarisation intervient pour décider de la hiérarchie des appels. L'ordre de linéarisation pour `grandChild` sera:

> grandChild -> traitC -> traitB -> traitA -> AnyRef -> Tous

Voici un autre exemple:

```
trait Printer {
  def print(msg : String) = println (msg)
}

trait DelimitWithHyphen extends Printer {
  override def print(msg : String) {
    println("-------------")
    super.print(msg)
  }
}

trait DelimitWithStar extends Printer  {
  override def print(msg : String) {
    println("*************")
    super.print(msg)
  }
}

class CustomPrinter extends Printer with DelimitWithHyphen with DelimitWithStar

object TestPrinter{
  def main(args: Array[String]) {
    new CustomPrinter().print("Hello World!")
  }
}
```

Ce programme imprime:

```
*************
-------------
Hello World!
```

La linéarisation de `CustomPrinter` sera:

CustomPrinter -> DelimitWithStar -> DelimitWithHyphen -> Imprimante -> AnyRef -> N'importe quel

Linéarisation

En cas de modification empilable , Scala organise les classes et les traits dans un ordre linéaire pour déterminer la hiérarchie des appels de méthode, appelée *linéarisation* . La règle de linéarisation est utilisée *uniquement* pour les méthodes impliquant un appel de méthode via `super()` . Considérons ceci par un exemple:

```
class Shape {
  def paint (shape: String): Unit = {
    println(shape)
  }
}

trait Color extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Color ")
  }
}

trait Blue extends Color {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Blue ")
  }
}

trait Border extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Border ")
  }
}

trait Dotted extends Border {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Dotted ")
  }
}

class MyShape extends Shape with Dotted with Blue {
  override def paint (shape : String) {
    super.paint(shape)
  }
}
```

La linéarisation se produit de l' *arrière vers l'avant* . Dans ce cas,

1. La première `Shape` sera linéarisée, ce qui ressemble à:

   ```
   Shape -> AnyRef -> Any
   ```

2. Puis `Dotted` est linéarisé:

   ```
   Dotted -> Border -> Shape -> AnyRef -> Any
   ```

3. Le suivant est le `Blue` . Normalement, la linéarisation de `Blue` sera:

```
Blue -> Color -> Shape -> AnyRef -> Any
```

car, dans la linéarisation de `MyShape` jusqu'à maintenant (*étape 2*), `Shape -> AnyRef -> Any` est déjà apparu. Par conséquent, il est ignoré. Ainsi, la linéarisation `Blue` sera:

```
Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any
```

4. Enfin, `Circle` sera ajouté et l'ordre de linéarisation final sera:

   Cercle -> Bleu -> Couleur -> Pointillé -> Bordure -> Forme -> AnyRef -> N'importe quel

Cet ordre de linéarisation détermine l'ordre d'invocation des méthodes lorsque `super` est utilisé dans une classe ou un trait. La première implémentation de la méthode à droite est appelée dans l'ordre de linéarisation. Si `new MyShape().paint("Circle ")` est exécuté, il imprimera:

```
Circle with Blue Color with Dotted Border
```

Vous trouverez plus d'informations sur la linéarisation ici .

Lire Traits en ligne: https://riptutorial.com/fr/scala/topic/1056/traits

Chapitre 54: Travailler avec des données dans un style immuable

Remarques

Les noms de valeur et de variable doivent être en cas de chameau inférieur

> Les noms constants doivent être en cas de chameau supérieur. Autrement dit, si le membre est final, immuable et qu'il appartient à un objet ou à un objet, il peut être considéré comme une constante
>
> La méthode, la valeur et les noms de variables doivent être dans le cas camel inférieur

Source: http://docs.scala-lang.org/style/naming-conventions.html

Cette compilation:

```
val (a,b) = (1,2)
// a: Int = 1
// b: Int = 2
```

mais ce n'est pas le cas:

```
val (A,B) = (1,2)
// error: not found: value A
// error: not found: value B
```

Examples

Ce n'est pas juste val vs var

`val` et `var`

```
scala> val a = 123
a: Int = 123

scala> a = 456
<console>:8: error: reassignment to val
       a = 456

scala> var b = 123
b: Int = 123

scala> b = 321
b: Int = 321
```

- `val` références `val` sont inchangeables: comme une variable `final` en `Java` , une fois initialisée, vous ne pouvez plus la changer
- `var` références `var` peuvent être réaffectées en tant que déclaration de variable simple en Java

Collections immuables et Mutable

```
  val mut = scala.collection.mutable.Map.empty[String, Int]
  mut += ("123" -> 123)
  mut += ("456" -> 456)
  mut += ("789" -> 789)

  val imm = scala.collection.immutable.Map.empty[String, Int]
  imm + ("123" -> 123)
  imm + ("456" -> 456)
  imm + ("789" -> 789)

  scala> mut
    Map(123 -> 123, 456 -> 456, 789 -> 789)

  scala> imm
    Map()

scala> imm + ("123" -> 123) + ("456" -> 456) + ("789" -> 789)
    Map(123 -> 123, 456 -> 456, 789 -> 789)
```

La bibliothèque standard Scala offre à la fois des structures de données immuables et mutables, et non la référence à celle-ci. Chaque fois qu'une structure de données immuable est "modifiée", une nouvelle instance est produite au lieu de modifier la collection d'origine sur place. Chaque instance de la collection peut partager une structure significative avec une autre instance.

Collection Mutable et Immuable (Documentation Scala Officielle)

Mais je ne peux pas utiliser l'immuabilité dans ce cas!

Prenons comme exemple une fonction qui prend 2 `Map` et renvoie une `Map` contenant tous les éléments de `ma` et `mb` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int]
```

Une première tentative pourrait être une itération à travers les éléments d'une des cartes en utilisant `for ((k, v) <- map)` et renvoyer d'une manière ou d'une autre la carte fusionnée.

```
def merge2Maps(ma: ..., mb: ...): Map[String, Int] = {

  for ((k, v) <- mb) {
    ???
  }

}
```

Ce tout premier pas ajoute immédiatement une contrainte: **une mutation en dehors de celle `for`**

est maintenant *nécessaire* . Ceci est plus clair lors de la suppression du sucre `for` :

```
// this:
for ((k, v) <- map) { ??? }

// is equivalent to:
map.foreach { case (k, v) => ??? }
```

"Pourquoi devons-nous muter?"

`foreach` repose sur les effets secondaires. Chaque fois que nous voulons que quelque chose se produise dans un `foreach` nous devons "modifier quelque chose", dans ce cas nous pourrions muter un `var result` variable ou utiliser une structure de données mutable.

Créer et remplir la carte de `result`

Supposons que `ma` et `mb` sont `scala.collection.immutable.Map` , nous pourrions créer le `result` Map de `ma` :

```
val result = mutable.Map() ++ ma
```

Ensuite, parcourez `mb` ajoutant ses éléments et si la `key` de l'élément actuel sur `ma` existe déjà, passons à celle de l'élément `mb` .

```
mb.foreach { case (k, v) => result += (k -> v) }
```

Implémentation Mutable

Jusqu'ici tout va bien, nous avons "dû utiliser des collections mutables" et une implémentation correcte pourrait être:

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  val result = scala.collection.mutable.Map() ++ ma
  mb.foreach { case (k, v) => result += (k -> v) }
  result.toMap // to get back an immutable Map
}
```

Comme prévu:

```
scala> merge2Maps(Map("a" -> 11, "b" -> 12), Map("b" -> 22, "c" -> 23))
  Map(a -> 11, b -> 22, c -> 23)
```

Se plier à la rescousse

Comment pouvons-nous nous débarrasser de `foreach` dans ce scénario? Si tout ce que nous faisons est essentiellement itératif sur les éléments de la collection et appliquer une fonction tout en accumulant le résultat, l'option pourrait utiliser `.foldLeft` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  mb.foldLeft(ma) { case (result, (k, v)) => result + (k -> v) }
  // or more concisely mb.foldLeft(ma) { _ + _ }
}
```

Dans ce cas, notre "résultat" est la valeur accumulée à partir de `ma` , le `zero` du `.foldLeft` .

Résultat intermédiaire

Evidemment, cette solution immuable produit et détruit de nombreuses instances de la `Map` lors du pliage, mais il convient de mentionner que ces instances ne sont pas un clone complet de la `Map` accumulée mais partagent une structure (données) importante avec l'instance existante.

Facilité de raisonnement

Il est plus facile de raisonner sur la sémantique si elle est plus déclarative que l’approche `.foldLeft` . L'utilisation de structures de données immuables pourrait contribuer à simplifier notre implémentation.

Lire Travailler avec des données dans un style immuable en ligne:
https://riptutorial.com/fr/scala/topic/6298/travailler-avec-des-donnees-dans-un-style-immuable

Chapitre 55: Travailler avec Gradle

Examples

Configuration de base

1. Créez un fichier nommé `SCALA_PROJECT/build.gradle` avec ces contenus:

```
group 'scala_gradle'
version '1.0-SNAPSHOT'

apply plugin: 'scala'

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "https://repo.typesafe.com/typesafe/maven-releases"
    }
}

dependencies {
    compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.6'
}


task  "create-dirs" << {
    sourceSets*.scala.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each { it.mkdirs() }
}
```

2. Exécuter des `gradle tasks` pour voir les tâches disponibles.

3. Exécutez `gradle create-dirs` pour créer un répertoire `src/scala, src/resources` .

4. Exécutez `gradle build` pour générer le projet et télécharger les dépendances.

Créez votre propre plugin Gradle Scala

Après avoir **suivi l'** exemple de **configuration de base** , vous pouvez vous retrouver à répéter la plupart de ces tâches dans chaque projet Scala Gradle. Ça sent le code passe-partout ...

Et si, au lieu d'appliquer le plug-in Scala proposé par Gradle, vous pouviez appliquer votre propre plug-in Scala, qui serait chargé de gérer toute votre logique de construction commune, tout en prolongeant le plug-in existant.

Cet exemple va transformer la logique de construction précédente en un plug-in Gradle réutilisable.

Heureusement, dans Gradle, vous pouvez facilement écrire des plug-ins personnalisés à l'aide de

l'API Gradle, comme indiqué dans la documentation . En tant que langage d'implémentation, vous pouvez utiliser Scala lui-même ou même Java. Cependant, la plupart des exemples que vous pouvez trouver dans les documents sont écrits dans Groovy. Si vous avez besoin de plus d'échantillons de code ou si vous voulez comprendre ce qui se cache derrière le plugin Scala, par exemple, vous pouvez vérifier le repo github gradle.

Ecrire le plugin

Exigences

Le plug-in personnalisé ajoutera les fonctionnalités suivantes lorsqu'il sera appliqué à un projet:

- un objet de propriété `scalaVersion` , qui aura deux propriétés par défaut remplaçables
  - major = "2.12"
  - minor = "0"
- une fonction `withScalaVersion` , qui s'applique à un nom de dépendance, ajoute la version principale de scala pour assurer la compatibilité binaire (l'opérateur sbt `%%` peut sonner, sinon, rendez-vous ici)
- une tâche `createDirs` pour créer l'arborescence de répertoires nécessaire, exactement comme dans l'exemple précédent

Directive de mise en œuvre

1. créer un nouveau projet de gradation et ajouter ce qui suit à `build.gradle`

```
apply plugin: 'scala'
apply plugin: 'maven'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile gradleApi()
    compile "org.scala-lang:scala-library:2.12.0"
}
```

Notes :

- l'implémentation du plugin est écrite en Scala, nous avons donc besoin du plugin Gradle Scala
- pour utiliser le plugin d'autres projets, le plugin Gradle Maven est utilisé; Cela ajoute la tâche d' `install` utilisée pour enregistrer le fichier de projet dans le référentiel local Maven.
- `compile gradleApi()` ajoute le `gradle-api-<gradle_version>.jar` au classpath

2. créer une nouvelle classe Scala pour l'implémentation du plugin

```
package com.btesila.gradle.plugins
```

```
import org.gradle.api.{Plugin, Project}

class ScalaCustomPlugin extends Plugin[Project] {
    override def apply(project: Project): Unit = {
        project.getPlugins.apply("scala")
    }
}
```

Notes :

- pour implémenter un plugin, il suffit d'étendre le trait de type `Plugin` in de type `Project` et de remplacer la méthode d' `apply`
- dans la méthode apply, vous avez accès à l'instance de `Project` laquelle le plug-in est appliqué et vous pouvez l'utiliser pour y ajouter une logique de construction
- ce plugin ne fait qu'appliquer le plug-in Gradle Scala déjà existant

3. ajouter la propriété d'objet `scalaVersion`

Tout d'abord, nous créons une classe `ScalaVersion` , qui contiendra les deux propriétés de la version

```
class ScalaVersion {
  var major: String = "2.12"
  var minor: String = "0"
}
```

Une chose intéressante à propos des plug-ins Gradle est le fait que vous pouvez toujours ajouter ou remplacer des propriétés spécifiques. Un plug-in reçoit ce type d'entrée d'utilisateur via le `ExtensionContainer` attaché à une instance de `Project` gradle. Pour plus de détails, consultez ceci . En ajoutant ce qui suit à la méthode `apply` , nous faisons essentiellement ceci:

- s'il n'y a pas de propriété `scalaVersion` définie dans le projet, on en ajoute une avec les valeurs par défaut
- sinon, nous obtenons l'existant comme instance de `ScalaVersion` , pour l'utiliser davantage

```
var scalaVersion = new ScalaVersion
if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
   project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
else
   scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]
```

Cela équivaut à écrire ce qui suit dans le fichier de génération du projet qui applique le plug-in:

```
ext {
    scalaVersion.major = "2.12"
    scalaVersion.minor = "0"

}
```

4. ajouter la bibliothèque `scala-lang` aux dépendances du projet, en utilisant la version `scalaVersion`

```
project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")
```

Cela équivaut à écrire ce qui suit dans le fichier de génération du projet qui applique le plug-in:

```
compile "org.scala-lang:scala-library:2.12.0"
```

5. ajouter la fonction `withScalaVersion`

```
val withScalaVersion = (lib: String) => {
    val libComp = lib.split(":")
    libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
    libComp.mkString(":")
}
project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)
```

6. Enfin, créez la tâche `createDirs` et ajoutez-la au projet
   Implémentez une tâche Gradle en étendant `DefaultTask` :

```
class CreateDirs extends DefaultTask {
  @TaskAction
  def createDirs(): Unit = {
    val sourceSetContainer =
this.getProject.getConvention.getPlugin(classOf[JavaPluginConvention]).getSourceSets

    sourceSetContainer forEach { sourceSet =>
      sourceSet.getAllSource.getSrcDirs.forEach(file => if (!file.getName.contains("java"))
file.mkdirs())
    }
  }
}
```

Remarque : le `SourceSetContainer` contient des informations sur tous les répertoires sources présents dans le projet. Ce que fait le plug- in Gradle Scala, c'est d'ajouter les jeux de sources supplémentaires aux jeux Java, comme vous pouvez le voir dans les docs du plug - in .

Ajoutez la tâche `createDir` au projet en ajoutant ceci à la méthode `apply` :

```
project.getTasks.create("createDirs", classOf[CreateDirs])
```

Au final, votre classe `ScalaCustomPlugin` devrait ressembler à ceci:

```
class ScalaCustomPlugin extends Plugin[Project] {
  override def apply(project: Project): Unit = {
    project.getPlugins.apply("scala")

    var scalaVersion = new ScalaVersion
    if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
```

```
      project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
    else
      scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]

    project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")

    val withScalaVersion = (lib: String) => {
      val libComp = lib.split(":")
      libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
      libComp.mkString(":")
    }
    project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)

    project.getTasks.create("createDirs", classOf[CreateDirs])
  }
}
```

Installation du projet de plug-in sur le référentiel Maven local

Cela se fait très facilement en exécutant `gradle install`
Vous pouvez vérifier l'installation en allant dans le répertoire du référentiel local, généralement situé dans le `~/.m2/repository`

Comment Gradle trouve-t-il notre nouveau plugin?

Chaque plugin Gradle a un `id` qui est utilisé dans l'instruction `apply` . Par exemple, en écrivant ce qui suit dans le fichier de compilation, cela se traduit par un déclencheur pour trouver et appliquer le plug-in avec id `scala` .

```
apply plugin: 'scala'
```

De même, nous aimerions appliquer notre nouveau plugin de la manière suivante,

```
apply plugin: "com.btesila.scala.plugin"
```

ce qui signifie que notre plugin aura l'identifiant `com.btesila.scala.plugin` .

Pour définir cet identifiant, ajoutez le fichier suivant:
src / main / resources / META-INF / gradle-plugin / com.btesil.scala.plugin.properties

```
implementation-class=com.btesila.gradle.plugins.ScalaCustomPlugin
```

Ensuite, `gradle install` .

Utiliser le plugin

1. créer un nouveau projet Gradle vide et ajouter les éléments suivants au fichier de

construction

```
buildscript {
    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
        //modify this path to match the installed plugin project in your local repository
        classpath 'com.btesila:working-with-gradle:1.0-SNAPSHOT'
    }
}

repositories {
    mavenLocal()
    mavenCentral()
}


apply plugin: "com.btesila.scala.plugin"
```

2. run `gradle createDirs` - vous devriez maintenant avoir tous les répertoires source générés
3. remplacer la version de la scala en l'ajoutant au fichier de construction:

```
ext {
    scalaVersion.major = "2.11"
    scalaVersion.minor = "8"

}
println(project.ext.scalaVersion.major)
println(project.ext.scalaVersion.minor)
```

4. ajouter une bibliothèque de dépendances compatible binaire avec la version Scala

```
dependencies {
    compile withScalaVersion("com.typesafe.scala-logging:scala-logging:3.5.0")
}
```

C'est tout! Vous pouvez maintenant utiliser ce plug-in pour tous vos projets sans répéter le même ancien modèle.

Lire Travailler avec Gradle en ligne: https://riptutorial.com/fr/scala/topic/3304/travailler-avec-gradle

Chapitre 56: Tuples

Remarques

Pourquoi les tuples sont-ils limités à la longueur 23?

Les tuples sont réécrits en tant qu'objets par le compilateur. Le compilateur a accès à `Tuple1` via `Tuple22` . Cette limite arbitraire a été décidée par les concepteurs de langues.

Pourquoi les longueurs de tuple comptent-elles à partir de 0?

Un `Tuple0` est équivalent à une `Unit` .

Examples

Créer un nouveau tuple

Un tuple est une collection hétérogène de deux à vingt-deux valeurs. Un tuple peut être défini à l'aide de parenthèses. Pour les tuples de taille `2` (également appelée paire), il existe une syntaxe de flèche.

```
scala> val x = (1, "hello")
x: (Int, String) = (1,hello)
scala> val y = 2 -> "world"
y: (Int, String) = (2,world)
scala> val z = 3 → "foo"     //example of using U+2192 RIGHTWARD ARROW
z: (Int, String) = (3,foo)
```

`x` est un tuple de taille deux. Pour accéder aux éléments d'un tuple, utilisez `._1` , via `._22` . Par exemple, nous pouvons utiliser `x._1` pour accéder au premier élément du tuple `x` . `x._2` accède au second élément. Plus élégamment, vous pouvez utiliser des extracteurs de tuple .

La syntaxe de flèche pour créer des tuples de taille deux est principalement utilisée dans Maps, qui sont des collections de paires `(key -> value)` :

```
scala> val m = Map[Int, String](2 -> "world")
m: scala.collection.immutable.Map[Int,String] = Map(2 -> world)

scala> m + x
res0: scala.collection.immutable.Map[Int,String] = Map(2 -> world, 1 -> hello)

scala> (m + x).toList
res1: List[(Int, String)] = List((2,world), (1,hello))
```

La syntaxe de la paire dans la carte est la syntaxe de la flèche, indiquant clairement que 1 est la clé et a la valeur associée à cette clé.

Tuples dans les collections

Les tuples sont souvent utilisés dans les collections, mais ils doivent être manipulés de manière spécifique. Par exemple, compte tenu de la liste suivante de tuples:

```
scala> val l = List(1 -> 2, 2 -> 3, 3 -> 4)
l: List[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Il peut sembler naturel d'ajouter les éléments en utilisant le tuple-unpacking implicite:

```
scala> l.map((e1: Int, e2: Int) => e1 + e2)
```

Cependant, cela entraîne l'erreur suivante:

```
<console>:9: error: type mismatch;
 found   : (Int, Int) => Int
 required: ((Int, Int)) => ?
              l.map((e1: Int, e2: Int) => e1 + e2)
```

Scala ne peut pas implicitement déballer les tuples de cette manière. Nous avons deux options pour corriger cette carte. La première consiste à utiliser les accesseurs positionnels `_1` et `_2` :

```
scala> l.map(e => e._1 + e._2)
res1: List[Int] = List(3, 5, 7)
```

L'autre option consiste à utiliser une instruction `case` pour décompresser les tuples à l'aide de la correspondance de modèle:

```
scala> l.map{ case (e1: Int, e2: Int) => e1 + e2}
res2: List[Int] = List(3, 5, 7)
```

Ces restrictions s'appliquent à toute fonction d'ordre supérieur appliquée à une collection de tuples.

Lire Tuples en ligne: https://riptutorial.com/fr/scala/topic/4971/tuples

Chapitre 57: Type Inférence

Examples

Inférence de type local

Scala dispose d'un puissant mécanisme d'inférence de type intégré au langage. Ce mécanisme est appelé «Inférence de type local»:

```
val i = 1 + 2                 // the type of i is Int
val s = "I am a String"       // the type of s is String
def squared(x : Int) = x*x    // the return type of squared is Int
```

Le compilateur peut déduire le type de variables de l'expression d'initialisation. De même, le type de retour des méthodes peut être omis, car elles sont équivalentes au type renvoyé par le corps de la méthode. Les exemples ci-dessus sont équivalents aux déclarations de type explicites ci-dessous:

```
val i: Int = 1 + 2
val s: String = "I am a String"
def squared(x : Int): Int = x*x
```

Type Inférence Et Génériques

Le compilateur Scala peut également déduire des paramètres de type lorsque des méthodes polymorphes sont appelées ou lorsque des classes génériques sont instanciées:

```
case class InferedPair[A, B](a: A, b: B)

val pairFirstInst = InferedPair("Husband", "Wife")  //type is InferedPair[String, String]

// Equivalent, with type explicitly defined
val pairSecondInst: InferedPair[String, String]
                      = InferedPair[String, String]("Husband", "Wife")
```

La forme d'inférence de type ci-dessus est similaire à l' opérateur Diamond , introduit dans Java 7.

Limites à l'inférence

Il existe des scénarios dans lesquels l'inférence de type Scala ne fonctionne pas. Par exemple, le compilateur ne peut pas déduire le type de paramètres de méthode:

```
def add(a, b) = a + b  // Does not compile
def add(a: Int, b: Int) = a + b // Compiles
def add(a: Int, b: Int): Int = a + b // Equivalent expression, compiles
```

Le compilateur ne peut pas déduire le type de retour des méthodes récursives:

```
// Does not compile
def factorial(n: Int) = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
// Compiles
def factorial(n: Int): Int = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
```

Prévenir l'inférence Rien

Basé sur cet article de blog .

Supposons que vous ayez une méthode comme celle-ci:

```
  def get[T]: Option[T] = ???
```

Lorsque vous essayez de l'appeler sans spécifier le paramètre générique, `Nothing` est déduit, ce qui n'est pas très utile pour une implémentation réelle (et son résultat n'est pas utile). Avec la solution suivante, le contexte `NotNothing` lié peut empêcher d'utiliser la méthode sans spécifier le type attendu (dans cet exemple, `RuntimeClass` est également exclu pour `ClassTags` not `Nothing` , mais `RuntimeClass` est déduit):

```
@implicitNotFound("Nothing was inferred")
sealed trait NotNothing[-T]

object NotNothing {
  implicit object notNothing extends NotNothing[Any]
  //We do not want Nothing to be inferred, so make an ambigous implicit
  implicit object `\n The error is because the type parameter was resolved to Nothing` extends
NotNothing[Nothing]
  //For classtags, RuntimeClass can also be inferred, so making that ambigous too
  implicit object `\n The error is because the type parameter was resolved to RuntimeClass`
extends NotNothing[RuntimeClass]
}

object ObjectStore {
  //Using context bounds
  def get[T: NotNothing]: Option[T] = {
    ???
  }

  def newArray[T](length: Int = 10)(implicit ct: ClassTag[T], evNotNothing: NotNothing[T]):
Option[Array[T]] = ???
}
```

Exemple d'utilisation:

```
object X {
  //Fails to compile
  //val nothingInferred = ObjectStore.get

  val anOption = ObjectStore.get[String]
  val optionalArray = ObjectStore.newArray[AnyRef]()

  //Fails to compile
  //val runtimeClassInferred = ObjectStore.newArray()
}
```

Lire Type Inférence en ligne: https://riptutorial.com/fr/scala/topic/4918/type-inference

Chapitre 58: Type Variance

Examples

Covariance

Le symbole `+` marque un paramètre de type comme *covariant* - nous disons ici que "le `Producer` est covariant sur `A` ":

```
trait Producer[+A] {
  def produce: A
}
```

Un paramètre de type covariant peut être considéré comme un type de "sortie". Marquer `A` comme covariant affirme que `Producer[X] <: Producer[Y]` condition que `X <: Y` Par exemple, un `Producer[Cat]` est un `Producer[Animal]` valide, car tous les chats produits sont également des animaux valides.

Un paramètre de type covariant ne peut pas apparaître en position contravariante (entrée). L'exemple suivant ne sera pas compilé car nous affirmons que `Co[Cat] <: Co[Animal]` , mais `Co[Cat]` a un `def handle(a: Cat): Unit` ne pouvant gérer aucun `Animal` comme requis par `Co[Animal]` !

```
trait Co[+A] {
  def produce: A
  def handle(a: A): Unit
}
```

Une approche pour traiter cette restriction consiste à utiliser des paramètres de type limités par le paramètre de type covariant. Dans l'exemple suivant, nous savons que `B` est un supertype de `A` Par conséquent, étant donné l' `Option[X] <: Option[Y]` pour `X <: Y` , nous savons que l' `Option[X] def getOrElse[B >: X](b: => B): B Option[X] def getOrElse[B >: X](b: => B): B` peut accepter n'importe quel supertype de `X` - qui inclut les supertypes de `Y` comme requis par l' `Option[Y]` :

```
trait Option[+A] {
  def getOrElse[B >: A](b: => B): B
}
```

Invariance

Par défaut, tous les paramètres de type sont invariants - étant donné le `trait A[B]` , nous disons que " `A` est invariant sur `B` ". Cela signifie que, compte tenu de deux paramétrisations `A[Cat]` et `A[Animal]` , nous n'affirmons pas de relation sub / superclass entre ces deux types - cela ne signifie pas que `A[Cat] <: A[Animal]` ni que `A[Cat] >: A[Animal]` quel que soit le lien entre le `Cat` et l' `Animal` .

Les annotations de variance nous permettent de déclarer une telle relation et imposent des règles sur l'utilisation des paramètres de type pour que la relation reste valide.

Contravariance

Le symbole `-` marque un paramètre de type comme *contravariant* - ici nous disons que " `Handler` est contravariant sur `A` ":

```
trait Handler[-A] {
  def handle(a: A): Unit
}
```

Un paramètre de type contravariant peut être considéré comme un type "d'entrée". Marquer `A` comme contravariant affirme que `Handler[X] <: Handler[Y]` fourni que `X >: Y` Par exemple, un `Handler[Animal]` est un `Handler[Cat]` valide `Handler[Cat]` , car un `Handler[Animal]` doit également manipuler des chats.

Un paramètre de type contravariant ne peut pas apparaître en position covariante (sortie). L'exemple suivant ne sera pas compilé car nous affirmons qu'un `Contra[Animal] <: Contra[Cat]` , mais un `Contra[Animal]` a un `def produce: Animal` qui n'est pas garanti pour produire des chats comme requis par `Contra[Cat]` !

```
trait Contra[-A] {
   def handle(a: A): Unit
   def produce: A
}
```

Attention toutefois: à des fins de surcharge de résolution, la contravariance inverse également la spécificité d'un type sur le paramètre de type contravariant - `Handler[Animal]` est considéré comme "plus spécifique" que `Handler[Cat]` .

Comme il n'est pas possible de surcharger les méthodes sur les paramètres de type, ce comportement ne devient généralement problématique que lors de la résolution des arguments implicites. Dans l'exemple suivant `ofCat` ne sera jamais utilisé, car le type de retour `ofAnimal` est plus spécifique:

```
implicit def ofAnimal: Handler[Animal] = ???
implicit def ofCat: Handler[Cat] = ???

implicitly[Handler[Cat]].handle(new Cat)
```

Ce comportement est actuellement prévu pour changer dans dotty , et c'est pourquoi (par exemple) `scala.math.Ordering` est invariant sur son paramètre de type `T` Une solution de contournement consiste à rendre l'invariant de votre classe de polices et à paramétrer la définition implicite dans le cas où vous souhaitez l'appliquer à des sous-classes d'un type donné:

```
trait Person
object Person {
  implicit def ordering[A <: Person]: Ordering[A] = ???
```

```
}
```

Covariance d'une collection

Comme les collections sont généralement covariantes dans leur type d'élément *, une collection d'un sous-type peut être transmise lorsqu'un super type est attendu:

```
trait Animal { def name: String }
case class Dog(name: String) extends Animal

object Animal {
  def printAnimalNames(animals: Seq[Animal]) = {
    animals.foreach(animal => println(animal.name))
  }
}

val myDogs: Seq[Dog] = Seq(Dog("Curly"), Dog("Larry"), Dog("Moe"))

Animal.printAnimalNames(myDogs)
// Curly
// Larry
// Moe
```

Cela peut ne pas sembler magique, mais le fait qu'un `Seq[Dog]` soit accepté par une méthode qui attend un `Seq[Animal]` est le concept entier d'un type de type supérieur (ici: `Seq`) étant covariant dans son paramètre type.

* Un contre-exemple étant le Set de la bibliothèque standard

Covariance sur un trait invariant

Il y a aussi un moyen d'avoir une seule méthode pour accepter un argument covariant, au lieu d'avoir le trait entier covariant. Cela peut être nécessaire car vous souhaitez utiliser `T` dans une position contravariante, tout en restant covariant.

```
trait LocalVariance[T]{
  /// ??? throws a NotImplementedError
  def produce: T = ???
  // the implicit evidence provided by the compiler confirms that S is a
  // subtype of T.
  def handle[S](s: S)(implicit evidence: S <:< T) = {
    // and we can use the evidence to convert s into t.
    val t: T = evidence(s)
    ???
  }
}

trait A {}
trait B extends A {}

object Test {
  val lv = new LocalVariance[A] {}

  // now we can pass a B instead of an A.
```

```
  lv.handle(new B {})
}
```

Lire Type Variance en ligne: https://riptutorial.com/fr/scala/topic/1651/type-variance

Chapitre 59: Types de méthode abstraits uniques (types SAM)

Remarques

Les méthodes abstraites uniques sont des types, introduits dans Java 8 , qui ont exactement un membre abstrait.

Examples

Syntaxe Lambda

NOTE: Ceci est uniquement disponible dans Scala 2.12+ (et dans les versions 2.11.x récentes avec les drapeaux du compilateur `-Xexperimental -Xfuture`)

Un type SAM peut être implémenté en utilisant un lambda:

2.11.8

```
trait Runnable {
  def run(): Unit
}

val t: Runnable = () => println("foo")
```

Le type peut éventuellement avoir d'autres membres non abstraits:

2.11.8

```
trait Runnable {
  def run(): Unit
  def concrete: Int = 42
}

val t: Runnable = () => println("foo")
```

Lire Types de méthode abstraits uniques (types SAM) en ligne:
https://riptutorial.com/fr/scala/topic/3664/types-de-methode-abstraits-uniques--types-sam-

Chapitre 60: Types de soi

Syntaxe

- trait type {selfId => / d' *autres membres peuvent se référer à* `selfId` *au cas où* `this` *signifie quelque chose* /}
- trait type {selfId: OtherType => / * les autres membres peuvent utiliser `selfId` et il sera de type `OtherType` * /
- type de trait {selfId: OtherType1 avec OtherType2 => / * `selfId` est du type `OtherType1` et `OtherType2` * /

Remarques

Souvent utilisé avec le motif de gâteau.

Examples

Exemple simple de type auto

Les types auto peuvent être utilisés dans les traits et les classes pour définir des contraintes sur les classes concrètes auxquelles ils sont mélangés. Il est également possible d'utiliser un identifiant différent pour `this` utilisant cette syntaxe (utile lorsque l'objet externe doit être référencé à partir d'un objet interne).

Supposons que vous souhaitiez stocker des objets. Pour cela, vous créez des interfaces pour le stockage et pour ajouter des valeurs à un conteneur:

```
trait Container[+T] {
  def add(o: T): Unit
}

trait PermanentStorage[T] {
  /* Constraint on self type: it should be Container
   * we can refer to that type as `identifier`, usually `this` or `self`
   * or the type's name is used. */
  identifier: Container[T] =>

  def save(o: T): Unit = {
    identifier.add(o)
    //Do something to persist too.
  }
}
```

De cette manière, ceux-ci ne sont pas dans la même hiérarchie d'objet, mais `PermanentStorage` ne peut pas être implémenté sans implémenter également `Container` .

Lire Types de soi en ligne: https://riptutorial.com/fr/scala/topic/4639/types-de-soi

Chapitre 61: Unités de manutention (mesures)

Syntaxe

- classe Meter (val metres: Double) étend AnyVal
- type Meter = Double

Remarques

Il est recommandé d'utiliser des classes de valeur pour les unités ou une bibliothèque dédiée pour celles-ci.

Examples

Alias de type

```
type Meter = Double
```

Cette approche simple présente de sérieux inconvénients pour la gestion des unités, car tout autre type de `Double` sera compatible avec celui-ci:

```
type Second = Double
var length: Meter = 3
val duration: Second = 1
length = duration
length = 0d
```

Toutes les données ci-dessus sont compilées. Dans ce cas, les unités ne peuvent être utilisées que pour marquer les types d'entrée / sortie pour les lecteurs du code (uniquement l'intention).

Classes de valeur

```
case class Meter(meters: Double) extends AnyVal
case class Gram(grams: Double) extends AnyVal
```

Les classes de valeur fournissent un moyen sûr d'encoder des unités, même si elles nécessitent un peu plus de caractères pour les utiliser:

```
var length = Meter(3)
var weight = Gram(4)
//length = weight //type mismatch; found : Gram required: Meter
```

En étendant `AnyVal` s, il n'y a pas de pénalité d'exécution pour les utiliser, au niveau de la JVM, ce

sont des types primitifs réguliers (`Double` s dans ce cas).

Si vous souhaitez générer automatiquement d'autres unités (comme `Velocity` aka `MeterPerSecond`), cette approche n'est pas la meilleure, bien qu'il existe des bibliothèques pouvant être utilisées dans ces cas également:

- Squants
- unités
- ScalaQuantité

Lire Unités de manutention (mesures) en ligne: https://riptutorial.com/fr/scala/topic/5966/unites-de-manutention--mesures-

Chapitre 62: Var, Val et Def

Remarques

Comme `val` sont sémantiquement statiques, ils sont initialisés "sur place" partout où ils apparaissent dans le code. Cela peut produire un comportement surprenant et indésirable lorsqu'il est utilisé dans des classes et des traits abstraits.

Par exemple, disons que nous aimerions créer un trait appelé `PlusOne` qui définit une opération d'incrémentation sur un `Int PlusOne` . Puisque les s `Int` sont immuables, la valeur plus un est connue à l'initialisation et ne sera jamais modifiée par la suite, donc sémantiquement, c'est un `val` . Cependant, sa définition de cette manière produira un résultat inattendu.

```
trait PlusOne {
    val i:Int

    val incr = i + 1
}

class IntWrapper(val i: Int) extends PlusOne
```

Peu importe quelle valeur `i` vous construis `IntWrapper` avec, en appelant `.incr` sur l'objet retourné retournera toujours 1. En effet , le val `incr` est initialisé *dans le trait,* avant la classe d' extension, et à ce moment - là , `i` ne dispose que la valeur par défaut `0` . (Dans d'autres conditions, il peut être rempli avec `Nil` , `null` ou une valeur par défaut similaire.)

La règle générale est donc d'éviter d'utiliser `val` sur toute valeur qui dépend d'un champ abstrait. Au lieu de cela, utilisez `lazy val` , qui n'évalue pas jusqu'à ce qu'il soit nécessaire, ou `def` , qui évalue chaque fois qu'il est appelé. Notez cependant que si le `lazy val` est forcé à évaluer par un `val` avant la fin de l'initialisation, la même erreur se produira.

Un violon (écrit en Scala-Js, mais le même comportement s’applique) peut être trouvé ici.

Examples

Var, Val et Def

var

Un `var` est une variable de référence, similaire aux variables dans des langages comme Java. Différents objets peuvent être librement assignés à une `var` , tant que l'objet donné a le même type que celui avec lequel la `var` été déclarée:

```
scala> var x = 1
x: Int = 1
```

```
scala> x = 2
x: Int = 2

scala> x = "foo bar"
<console>:12: error: type mismatch;
 found   : String("foo bar")
 required: Int
       x = "foo bar"
       ^
```

Remarquez dans l'exemple ci-dessus que le type du `var` été déduit par le compilateur en fonction de la première affectation de valeur.

val

Un `val` est une référence constante. Ainsi, un nouvel objet ne peut pas être attribué à un `val` qui a déjà été attribué.

```
scala> val y = 1
y: Int = 1

scala> y = 2
<console>:12: error: reassignment to val
       y = 2
         ^
```

Cependant, l'objet d' un `val` pointe est *pas* constante. Cet objet peut être modifié:

```
scala> val arr = new Array[Int](2)
arr: Array[Int] = Array(0, 0)

scala> arr(0) = 1

scala> arr
res1: Array[Int] = Array(1, 0)
```

déf

Un `def` définit une méthode. Une méthode ne peut pas être réaffectée à.

```
scala> def z = 1
z: Int

scala> z = 2
<console>:12: error: value z_= is not a member of object $iw
       z = 2
       ^
```

Dans les exemples ci-dessus, `val y` et `def z` renvoient la même valeur. Cependant, un `def` est évalué *quand il est appelé* , alors qu'un `val` ou `var` est évalué *quand il est assigné* . Cela peut

entraîner des comportements différents lorsque la définition a des effets secondaires:

```
scala> val a = {println("Hi"); 1}
Hi
a: Int = 1

scala> def b = {println("Hi"); 1}
b: Int

scala> a + 1
res2: Int = 2

scala> b + 1
Hi
res3: Int = 2
```

Les fonctions

Les fonctions étant des valeurs, elles peuvent être affectées à `val` / `var` / `def` s. Tout le reste fonctionne de la même manière que ci-dessus:

```
scala> val x = (x: Int) => s"value=$x"
x: Int => String = <function1>

scala> var y = (x: Int) => s"value=$x"
y: Int => String = <function1>

scala> def z = (x: Int) => s"value=$x"
z: Int => String

scala> x(1)
res0: String = value=1

scala> y(2)
res1: String = value=2

scala> z(3)
res2: String = value=3
```

Lazy val

`lazy val` est une fonctionnalité de langage dans laquelle l'initialisation d'un `val` est retardée jusqu'à ce qu'il soit accédé pour la première fois. Après cela, il agit comme un `val` régulier.

Pour l'utiliser, ajoutez le mot-clé `lazy` avant `val` . Par exemple, en utilisant le REPL:

```
scala> lazy val foo = {
     |   println("Initializing")
     |   "my foo value"
     | }
foo: String = <lazy>

scala> val bar = {
     |   println("Initializing bar")
```

```
     |   "my bar value"
     | }
Initializing bar
bar: String = my bar value

scala> foo
Initializing
res3: String = my foo value

scala> bar
res4: String = my bar value

scala> foo
res5: String = my foo value
```

Cet exemple illustre l'ordre d'exécution. Lorsque le `lazy val` est déclaré, tout ce qui est enregistré dans la valeur `foo` est un appel de fonction paresseux qui n'a pas encore été évalué. Lorsque le régulier `val` est réglé, nous voyons le `println` appel et exécuter la valeur est affectée à la `bar` . Lorsque nous évaluons `foo` la première fois que `println` s'exécute - mais pas quand il est évalué la deuxième fois. De même, lorsque `bar` est évalué, nous ne voyons pas `println` execute - seulement quand il est déclaré.

Quand utiliser 'paresseux'

1. **L'initialisation est coûteuse en calculs et l'utilisation de `val` est rare.**

   ```
   lazy val tiresomeValue = {(1 to 1000000).filter(x => x % 113 == 0).sum}
   if (scala.util.Random.nextInt > 1000) {
     println(tiresomeValue)
   }
   ```

   `tiresomeValue` prend beaucoup de temps à calculer, et il n'est pas toujours utilisé. Faire un `lazy val` enregistre des calculs inutiles.

2. **Résoudre les dépendances cycliques**

   Regardons un exemple avec deux objets qui doivent être déclarés en même temps lors de l'instanciation:

   ```
   object comicBook {
     def main(args:Array[String]): Unit = {
       gotham.hero.talk()
       gotham.villain.talk()
     }
   }

   class Superhero(val name: String) {
     lazy val toLockUp = gotham.villain
     def talk(): Unit = {
       println(s"I won't let you win ${toLockUp.name}!")
     }
   }
   ```

```
class Supervillain(val name: String) {
  lazy val toKill = gotham.hero
  def talk(): Unit = {
    println(s"Let me loosen up Gotham a little bit ${toKill.name}!")
  }
}

object gotham {
  val hero: Superhero = new Superhero("Batman")
  val villain: Supervillain = new Supervillain("Joker")
}
```

Sans le mot-clé `lazy` , les objets respectifs ne peuvent pas être membres d'un objet. L'exécution d'un tel programme entraînerait une `java.lang.NullPointerException` . En utilisant `lazy` , la référence peut être assignée avant son initialisation, sans crainte d'avoir une valeur non initialisée.

Surcharge surcharge

Vous pouvez surcharger un `def` si la signature est différente:

```
def printValue(x: Int) {
  println(s"My value is an integer equal to $x")
}

def printValue(x: String) {
  println(s"My value is a string equal to '$x'")
}

printValue(1)  // prints "My value is an integer equal to 1"
printValue("1") // prints "My value is a string equal to '1'"
```

Cela fonctionne de la même façon que ce soit à l'intérieur des classes, des traits, des objets ou non.

Paramètres nommés

Lors de l'appel d'un `def` , les paramètres peuvent être attribués explicitement par leur nom. Cela signifie qu'ils n'ont pas besoin d'être correctement commandés. Par exemple, définissez `printUs()` comme:

```
// print out the three arguments in order.
def printUs(one: String, two: String, three: String) =
   println(s"$one, $two, $three")
```

Maintenant, il peut être appelé de cette façon (entre autres):

```
printUs("one", "two", "three")
printUs(one="one", two="two", three="three")
printUs("one", two="two", three="three")
printUs(three="three", one="one", two="two")
```

Cela se traduit par `one, two, three` étant imprimés dans tous les cas.

Si tous les arguments ne sont pas nommés, les premiers arguments sont comparés par ordre. Aucun argument positionnel (non nommé) ne peut suivre un nommé:

```
printUs("one", two="two", three="three") // prints 'one, two, three'
printUs(two="two", three="three", "one") // fails to compile: 'positional after named
argument'
```

Lire Var, Val et Def en ligne: https://riptutorial.com/fr/scala/topic/3155/var--val-et-def

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage Scala	4444, Andy Hayden, Ani Menon, Community, David G., David Portabella, dk14, Donald.McLean, Gabriele Petronella, Grzegorz Oledzki, implicitdef, isaias-b, J Atkin, Jean, Jonathan, mammothbane, marcospereira, Marek Skiba, mdarwin, Nathaniel Ford, NeoWelkin, Nicofisi, Priya, rolve, Shoe, sschaef, Thomas Andrews, Tyler James Harden, Ven, Vogon Jeltz
2	Annotations	Gábor Bakos, Nathaniel Ford, Thomas Matecki
3	Bibliothèque de continuation	dmitry, HTNW
4	Classe d'option	Bruce Lowe, CPS, earldouglas, evan.oman, Governa, John Starich, Matthew Scharley, Nathaniel Ford, R Pieters, ScientiaEtVeritas, suj1th, Tzach Zohar, Vasiliy Levykin
5	Classes de cas	Andy Hayden, Dan Simon, dk14, Gábor Bakos, HTNW, J Cracknell, keegan, made raka teja, Marc Grue, Nathaniel Ford, pedrorijo91, Rumoku, ScientiaEtVeritas, suj1th, Suma
6	Classes de types	Arseniy Zhizhelev, Daniel C. Sobral, Gábor Bakos, gregghz, Nathaniel Ford, TomTom, Yawar
7	Classes et Objets	Aamir, Gábor Bakos, mdarwin, mirosval, MSmedberg, Nathaniel Ford, ScientiaEtVeritas, steve, Sudhir Singh, Tzach Zohar, vivek
8	Collections	Anton, Camilo Sampedro, deepkimo, Donald.McLean, doub1ejack, EdgeCaseBerg, Filippo Vitale, George, implicitdef, ipoteka, Jason, John Starich, Mr D, Nathaniel Ford, raam86, Shastick, Suma, Tundebabzy, Vasiliy Levykin
9	Collections parallèles	Nathaniel Ford, Shuklaswag
10	Combinateurs d'analyseur	Nathaniel Ford
11	Correspondance de motif	Ali Dehghani, Andrzej Jozwik, Andy Hayden, CPS, Dan Simon, Daniel Werner, Filippo Vitale, Gábor Bakos, implicitdef, insan-e, jilen, jozic, JRomero, Justin Bailey, Louis F., mammothbane, Matt, Nadim Bahadoor, Nathaniel Ford, Peter Neyens, Sergio, Shastick, Shoe, Simon, Suma, T.Grottker, user6062072, vdebergue, vsminkov, Yagüe

12	Currying	Adamos Loizou, alphaloop, Amr Gawish, dimitrisli, Luka Jacobowitz, Nathaniel Ford, rjsvaljean, Suma, vise890
13	Énumérations	Andy Hayden, Cortwave, Daniel Schröter, Gábor Bakos, implicitdef, ipoteka, Nathaniel Ford, phantomastray, Red Mercury
14	Expressions régulières	dmitry, J Cracknell, Nathaniel Ford
15	Extracteurs	Andy Hayden, Dan Hulme, Dan Simon, Gábor Bakos, gilad hoch, Idloj, J Cracknell, jwvh, knutwalker, Łukasz, Martin Seeler, Michael Ahlers, Nathaniel Ford, Suma, W.P. McNeill
16	Fonction d'ordre supérieur	acjay, ches, Nathaniel Ford, nukie, Rajat Jain, Srini
17	Fonctions définies par l'utilisateur pour la ruche	Camilo Sampedro
18	Fonctions partielles	acjay, Akash Sethi, David Leppik, dimitrisli, jwvh, Suma, Tzach Zohar
19	Futures	isaias-b, kevin628, Nathaniel Ford, nukie, Shastick
20	Implicits	Andy Hayden, dimitrisli, Gábor Bakos, HTNW, implicitdef, ipoteka, Jose Antonio Jimenez Saez, Michael Zajac, Nathaniel Ford, nattyddubbs, Simon, spiffman, Suma, Timo, vsminkov
21	Injection de dépendance	Hoang Ong
22	Interopérabilité Java	Andrzej Jozwik, Dan Hulme, Gábor Bakos, mvn, the21st, thekingofkings
23	Interpolation de chaîne	Andy Hayden, Ayberk, Brian, implicitdef, J Cracknell, Nadim Bahadoor
24	Invocation dynamique	HTNW
25	JSON	ipoteka, John, Muki, Nathaniel Ford, pedrorijo91, suj1th, void, Wogan, zoitol
26	La gestion des erreurs	Andy Hayden, Graham, John Starich, made raka teja, mnoronha, Nathaniel Ford, Simon, Suma, tacos_tacos_tacos, Tzach Zohar
27	Les fonctions	Aravindh S, Archeg, Camilo Sampedro, ches, corvus_192,

		Dawny33, Gábor Bakos, Gabriele Petronella, implicitdef, ipoteka , Jean, jwvh, michael_s, Nathaniel Ford, raam86, rjsvaljean, ScientiaEtVeritas, Shastick, stefanobaghino, Sven Koschnicke, vise890, wheaties
28	Les meilleures pratiques	corvus_192, ipoteka, Nathaniel Ford, RamenChef, Sarvesh Kumar Singh, Shuklaswag
29	Macros	gregghz, HTNW, Nathaniel Ford
30	Manipulation XML	Nathaniel Ford, Rockie Yang, vsnyc
31	Mise en place de Scala	Hristo Iliev, Matas Vaitkevicius, Nathaniel Ford, Rjk
32	Monades	ipoteka, Nathaniel Ford
33	Opérateurs à Scala	Gábor Bakos, Shaido, Suminda Sirinath S. Dharmasena
34	Paquets	Alex Javarotti, Nathaniel Ford, NetanelRabinowitz
35	Paramétrage de type (génériques)	akauppi, Andy Hayden, Eero Helenius, Nathaniel Ford, vivek
36	Pendant que les boucles	J Cracknell, Nathaniel Ford
37	Portée	Camilo Sampedro
38	Pour les expressions	Andy Hayden, J Cracknell, jwvh, LivingRobot, Nathaniel Ford, ScientiaEtVeritas
39	Programmation au niveau du type	J Cracknell
40	Quasiquotes	gregghz
41	Récursivité	Dmitry Bystritsky, Gábor Bakos, jilen, jwvh, michael_s, ScientiaEtVeritas, teldosas
42	Réflexion	Sachin Janani
43	Ruisseaux	jwvh, Nathaniel Ford, Oleg Pyzhcov
44	Scala.js	Camilo Sampedro
45	Scaladoc	Camilo Sampedro, Gábor Bakos, Nathaniel Ford
46	scalaz	chengpohi
47	Si des expressions	corvus_192, Nathaniel Ford, ScientiaEtVeritas

48	Surcharge de l'opérateur	corvus_192, implicitdef, inzi, mnoronha, Nathaniel Ford, Simon
49	Symboles littéraux	ZbyszekKr
50	synchronisé	Gábor Bakos
51	Test avec ScalaTest	Nadim Bahadoor, Nathaniel Ford
52	Tests avec ScalaCheck	Andrzej Jozwik
53	Traits	André Laszlo, Andy Hayden, Donald.McLean, Louis F., Nathaniel Ford, Rumoku, Sudhir Singh, Vogon Jeltz
54	Travailler avec des données dans un style immuable	Filippo Vitale
55	Travailler avec Gradle	Bianca Tesila, Nathaniel Ford, Rjk
56	Tuples	corvus_192, evan.oman, Lawsy, Nathaniel Ford
57	Type Inférence	Gábor Bakos, Nathaniel Ford, suj1th
58	Type Variance	acjay, J Cracknell, Reactormonk
59	Types de méthode abstraits uniques (types SAM)	Gábor Bakos, Gabriele Petronella, Nathaniel Ford
60	Types de soi	Gábor Bakos, irundaia
61	Unités de manutention (mesures)	Gábor Bakos
62	Var, Val et Def	Aamir, John Starich, jwvh, linkhyrule5, Nathaniel Ford, Shastick, Shuklaswag, stefanobaghino, ZbyszekKr