HEIG VD

Répondre à tous les quiz sur https://etc.ch/2XRK

Programmation Appliquée en Scala

Chapitre 9: Programmation asynchrone avec les Futurs

Prof. Nastaran Fatemi

Christopher Meier

Printemps 2025

Dans ce cours

- Calculs asynchrones utilisant des objets `Future`
- Gestion des résultats de calculs asynchrones par **attente active**
- Installation de **rappel** pour gérer les résultats des calculs asynchrones
- Sémantique d'exception des objets `Futures` : utilisation du type `Try`
- Composition fonctionnelle des objets `Futures`
- *For comprehention* pour les objets `Futures`
- Gestion des échecs des objets `Futures`

Programmation asynchrone

La programmation asynchrone fait référence au style de programmation dans lequel les exécutions se produisent indépendamment du flux du programme principal.

La programmation asynchrone **aide à éliminer le blocage**

- A la place de suspendre le *thread* chaque fois qu'une ressource n'est pas disponible ; **un calcul séparé est planifié** pour se poursuivre une fois que la ressource devient disponible.

Dans cette section, nous étudions une abstraction en Scala, appelée **future** qui est spécialement conçue pour faciliter la composition des calculs asynchrones.

Threads et blocage

Des exécutions parallèles dans un programme concurent se poursuivent sur des entités appelé **threads**.

À tout moment, l'exécution d'un *thread* peut être temporairement suspendue jusqu'à ce qu'une condition spécifique soit remplie. Quand cela arrive, nous disons que le *thread* est **bloqué**.

Pourquoi les *threads* sont bloqués?

1. Nous avons une quantité limitée de ressources; plusieurs calculs qui partagent ces ressources doivent parfois attendre.
2. Parfois, un calcul a besoin de données spécifiques pour continuer, et que les données ne sont pas encore disponibles (*thread* lent ou données externes).

Un exemple

Supposons que nous avons une méthode `getWebpage`, qui prenant l'URL de l'emplacement de la page Web, renvoie le contenu de cette page Web:

```
def getWebpage (url: String): String
```

Cette méthode doit renvoyer une chaîne de caractères avec le contenu de la page Web. Cependant, sur l'envoi d'une demande HTTP, **le contenu de la page Web n'est pas disponible immédiatement**.

- La seule façon pour la méthode de renvoyer le contenu de la page Web comme une valeur de chaîne de caractères est **d'attendre que la réponse HTTP arrive**.
- Comme le thread qui a appelé la méthode `getWebpage` ne peut pas continuer sans le contenu de la page Web, **il doit suspendre son exécution** => **bloquer**

Comment faire mieux?

On sait déjà que le blocage peut avoir des effets secondaires négatifs, donc **peut-on changer la valeur de retour de la méthode `getWebpage` en une valeur spéciale qui peut être retournée immédiatement?**

– Oui! : en Scala, cette valeur spéciale est appelée un **future**.

Un **future est un espace réservé**, c'est-à-dire un emplacement de mémoire pour la valeur. Cet espace réservé n'a pas besoin de contenir de valeur lorsque le `future` est créé; la valeur peut être finalement (eventually) placée dans le future par la méthode `getWebpage`.

```
def getWebpage (url: String): Future[String]
```

Ici, le type `Future[String]` signifie que l'objet future peut finalement contenir une valeur `String`.

Valeur future vs calcul future

Quand on utilise le terme **future**, généralement on veut dire une **valeur future**:

- Une valeur de type «T» dans le programme qui pourrait ne pas être disponible actuellement, mais qui pourrait devenir disponible plus tard.

Dans le paquetage `scala.concurrent`, les futures sont représentés par le trait:

```
trait Future[T]
```

Un **calcul future** est un calcul asynchrone qui produit une valeur future.

- Il peut être démarré en appelant la méthode `apply` sur l'objet compagnon `Future`. Il a la signature suivante dans le paquetage `scala.concurrent`:

```
def apply [T] (b: => T) (implicit e: ExecutionContext): Future[T]
```

Valeur future vs calcul future

```
def apply[T](b: =>T)(implicit e: ExecutionContext): Future[T]
```

La méthode `apply` ci-dessus prend deux paramètres:

- Un premier paramètre par nom (by-name) de type «T». Ceci est le corps (body) du calcul asynchrone qui génère une valeur de type «T».
- Un deuxième paramètre implicite **`ExecutionContext`**, qui résume où et quand le thread est exécuté. Vous pouvez le considérer comme le **pool de threads**.

Les paramètres implicites de Scala peuvent être spécifiés lors de l'appel d'une méthode, de la même manière que les paramètres normaux, ou, comme ici ils peuvent être laissés de côté, et dans ce cas le compilateur Scala recherche une valeur de type `ExecutionContext` dans la portée environnante.

Démarrer des calculs futures

Voyons comment démarrer un calcul future dans un exemple:

```
import scala.concurrent.*
import ExecutionContext.Implicits.global
@main def FuturesCreate =
  def log(msg: String): Unit =
    println(s"${Thread.currentThread.getName}: $msg")

  Future {log ("the future is here")}
  log ("the future is coming")
```

L'ordre dans lequel les appels de la méthode `println` s'exécutent (à l'intérieur du calcul future et dans le thread principal) est **non déterministe**.

L'objet singleton `Future` suivi d'un bloc est du sucre syntaxique pour appel de la méthode `Future.apply`.

Attente active: `isCompleted`

Dans l'exemple suivant, nous pouvons utiliser l'objet `scala.io.Source` pour lire le contenu de notre fichier `build.sbt` dans un calcul future.

On voit les tentatives d'interrogation de la valeur du future jusqu'à ce qu'elle soit terminée.

```
import scala.io.Source
@main def futureChapter =
  val buildFile = Future {
      val f = Source.fromFile("build.sbt")
      try f.getLines.mkString("\n") finally f.close()
  }
  log(s"started reading the build file asynchronously")
  log(s"status: ${buildFile.isCompleted}")
  Thread.sleep(250)
  log(s"status: ${buildFile.isCompleted}")
  log(s"value: ${buildFile.value}")
```

HEIG VD

*Répond à la **question 1** sur https://etc.ch/2XRK*

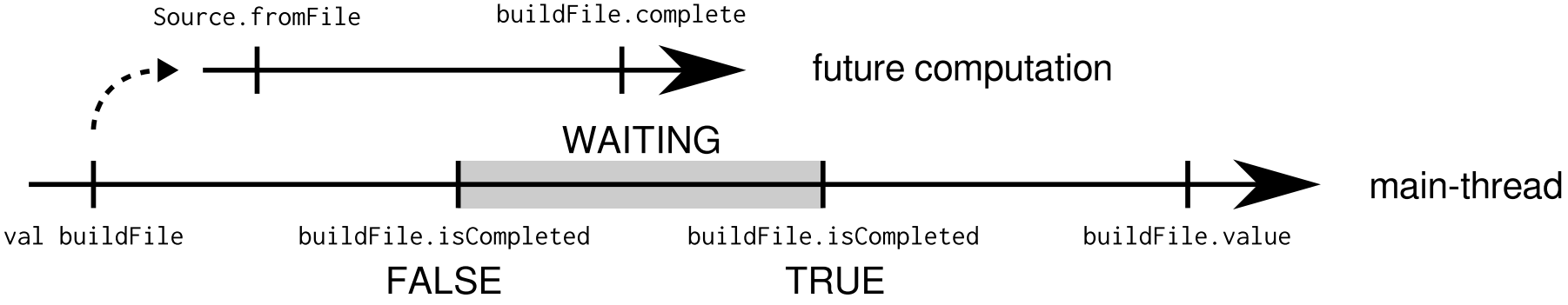
Quiz

Quel est le type de la variable `buildFile` dans l'exemple du slide précédent?

A String
B Array[String]
C Future[String]
D Unit
E Un autre type

Attente active: `isCompleted`

- Le thread principal appelle la méthode `isCompleted` sur la valeur future `buildFile` renvoyée par le calcul future.
- Il y a de fortes chances que le fichier build n'ait pas été lu si rapidement, donc `isCompleted` renvoie faux.
- Après 250 millisecondes, le thread principal appelle la méthode `isCompleted` à nouveau et cette fois, `isCompleted` renvoie true.
- Enfin, le thread principal appelle la méthode `value`, qui retourne le contenu du fichier build.



Timeline: `buildFile`

Attente active

L'attente active c'est comme appeler votre employeur potentiel toutes les cinq minutes pour demander si vous êtes embauché!

Ce que vous voulez vraiment faire, c'est :

- déposer une candidature, puis **postuler pour d'autres emplois**, au lieu d'attendre pour la réponse de l'employeur.
- une fois que votre employeur décide de vous embaucher, il **vous donnera un appel sur le numéro de téléphone que vous leur avez laissé**.

Nous voulons que les futurs fassent de même; quand ils sont terminés, ils devraient appeler une fonction spécifique que nous leur avons laissée. C'est le sujet des **callback**.

Rappel (Callback)

Un **rappel** (callback) est une fonction qui est appelée une fois que ses arguments deviennent disponible.

Lorsqu'un future Scala prend un rappel, il **finit par appeler ce rappel. Mais, le future n'appelle jamais le rappel avant sa terminaison**.

Dans l'exemple suivant, on suppose qu'on doit rechercher les détails de la spécification d'URL du consortium W3

- Cette spécification est disponible en tant que document texte sur https://www.w3.org/.
- On est intéressés par toutes les occurrences du mot clé `telnet`.

Rappel : Exemple (1/5)

```
@main def FutureCallBacks =
  def getUrlSpec(): Future[List[String]] = Future {
    val url = "http://www.w3.org/Addressing/URL/url-spec.txt"
    val f = Source.fromURL(url)
    try f.getLines.toList finally f.close()
  }
  val urlSpec: Future[List[String]] = getUrlSpec()
```

La méthode `getUrlSpec` utilise des futures pour exécuter de manière asynchrone la requête HTTP.

- Il appelle d'abord la méthode `fromURL` pour obtenir un objet `Source` avec le document texte.
- Il appelle ensuite `getLines` pour obtenir une liste des lignes distinctes du document.

Rappel : Exemple (2/5)

```
def find(lines: List[String], keyword: String): String =
  lines.zipWithIndex collect {
      case (line, n) if line.contains(keyword) => (n, line)
  } mkString("\n")
```

Pour trouver les lignes contenant le mot-clé `telnet`, on utilise la méthode `find`.

- La méthode find prend un paramètre `List[String]`, mais `urlSpec` est de type `Future[List[String]]`.
- On ne peut passer le future `urlSpec` directement à la méthode `find`; et pour une bonne raison, la valeur peut ne pas être disponible au moment où nous appelons la méthode `find`!

Rappel : Exemple (3/5)

```
urlSpec onComplete {
  case Success(lines) => log(find(lines, "telnet"))
  case Failure (t) => println("An error has occurred: " +
  t.getMessage)
}
log("callback registered, continuing with other work")
Thread.sleep(2000)
```

On installe un **rappel** sur le future en utilisant la méthode **`onComplete`**. Cette méthode prend une fonction de type `Try [T] => U`.

- Le rappel est appliqué à la valeur de type `Success [T]` si le future se termine avec succès,
- ou sinon, à une valeur de type `Failure [T]`.

L'installation d'un rappel est une opération **non bloquante**. Le log dans le thread principal s'exécute immédiatement après l'enregistrement du rappel, mais l'instruction log dans le rappel peut être appelée beaucoup plus tard.

Le type `Try`

`Try[T]` est similaire à `Option[T]`, car il s'agit d'une unité potentiellement détenant une valeur d'un certain type.

Une `Option[T]` pourrait être:

- soit une valeur: `Some[T]`
- soit aucune valeur: `None`

`Try[T]`, a été spécialement conçu pour contenir:

- soit une valeur: `Success[T]`
- soit une exception: `Failure[T]`. Ainsi, `Failure[T]` contient plus d'informations qu'un simple `None` en disant pourquoi la valeur n'est pas là.

La signature de la méthode onComplete

```
abstract def onComplete[U](f: (Try[T]) => U)
                          (implicit executor: ExecutionContext): Unit
```

La méthode onComplete prend alors en paramètre :

- une fonction `f: (Try[T]) => U`
- le contexte d'éxecution : thread pool

Lorsque le future est terminé, que ce soit par le biais d'une exception ou d'une valeur, onComplete applique la fonction fournie.

Rappel : Exemple (4/5)

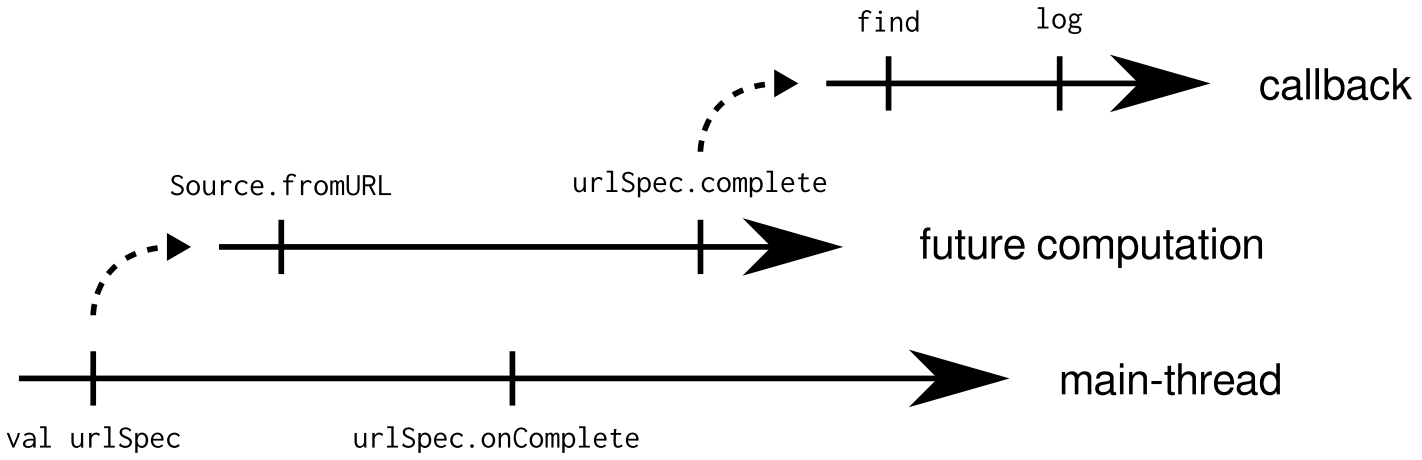
On n'est pas limité à installer un seul rappel sur le futur. Par exemple, si on veut également trouver toutes les occurrences du mot-clé `password`, on peut installer un autre rappel.

```
  urlSpec onComplete {
    case Success(lines) => log(find(lines, "password"))
    case Failure (t) => println("An error has occurred: " +
                                t.getMessage)
  }
  Thread.sleep(1000)
...
```

Remarque

Une fois le future terminé, le rappel est appelé finalement et indépendamment d'autres rappels sur le même future. Le contexte d'exécution spécifié décide quand et sur quel thread le rappel est exécuté.

Rappel : Exemple (5/5)

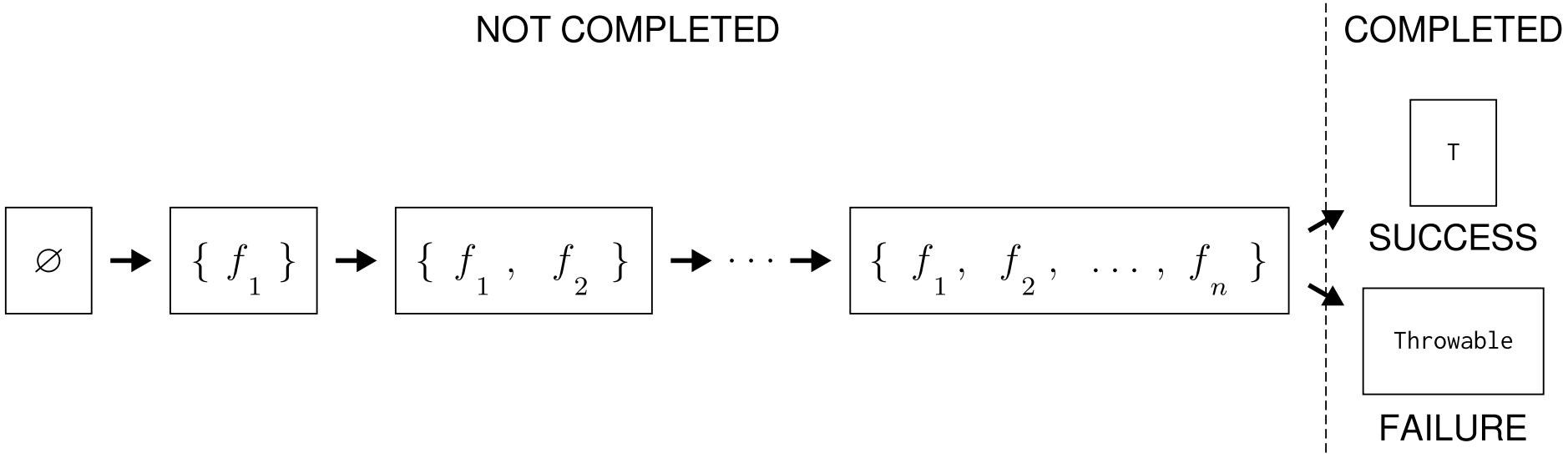


Timeline: `urlSpec`

– Notez que le rappel n’est pas nécessairement appelé immédiatement après que le future est terminé.

Diagramme d'état de future

La figure ci-dessous montre différents états d'un future en tant que diagramme d'état.



Future state diagram

Diagramme d'état de future

Explications sur la figure précédente:

- Un future est créé sans aucun rappel associé.
- Ensuite, n'importe quel nombre de rappels f_1, f_2, ..., f_n peut lui être affecté.
- Lorsque le future est **terminé**, il s'est terminé **avec succès** ou **a échoué**.
- Après cela, l'état du future ne change plus et l'enregistrement d'un rappel planifie immédiatement son exécution.

Autres fonctions de rappel

La méthode **`onComplete`** est générale dans le sens où elle permet au client de gérer le résultat de **à la fois des calculs futures échoués et réussis**.

Une autre fonction de rappel couramment utilisée est **`foreach`** qui ne gère que **les résultats réussis**.

Par exemple:

```
urlSpec foreach {
  case lines => log(find(lines, "telnet"))
}
```

Enchaîner les résultats futures

Les rappels sont utiles, mais ils peuvent rendre le raisonnement sur le flux de contrôle difficile lorsque les programmes deviennent plus complexes. Ils empêchent également certains modèles de programmation asynchrone.

Par exemple, nous pourrions avoir besoin d'initier de nouveaux futures basés sur un calcul future. Il nous faudrait alors :

- **Un mécanisme pour transformer un future à un autre future !**
- Est-ce que cela vous rappelle quelque chose?

Enchaîner les résultats futures : Exemple

Supposons que nous ayons une API pour interfacer avec un service d'échange de devise. Nous voulons acheter des dollars américains, mais uniquement lorsque c'est rentable.

Ce que nous pouvons faire jusqu'à présent:

```
val rateQuote = Future {
  connection.getCurrentValue(USD)
}
val purchase = ??? // Ici, nous avons besoin d'un mécanisme pour
                   // dire qu'une fois le future rateQuote terminé,
                   // nous voulons démarrer un autre future, purchase,
                   // qui décide d'acheter uniquement s'il est rentable,
                   // puis envoyer une demande.
```

Enchaîner les résultats futures : utilisant le rappel

D'abord, essayons le mécanisme de rappel

```
val rateQuote = Future {
  connection.getCurrentValue(USD)
}
rateQuote onComplete {
  case Success(value) => val purchase = Future {
    if isProfitable(quote) then connection.buy(amount, quote)
    else throw new Exception("not profitable")
  }
  purchase onComplete {
    case Success(value) => println("Purchased " + value + " USD")
    case Failure (e1) =>  println("An error occurred: " + e1.getMessage)
  }
  case Failure(e2) => println("An error occurred: " + e2.getMessage)
}
```

Pourquoi ce n’est pas une bonne solution

Cela fonctionne, mais n’est pas pratique pour deux raisons:

- Nous devons utiliser `onComplete` et y imbriquer le deuxième futur `purchase`. Imaginez qu’une fois l’achat terminé, nous voulons vendre une autre devise. Nous aurions à **répéter ce modèle dans le rappel `onComplete`**, ce qui rendrait le code trop imbriqué, volumineux et difficile à raisonner.
- Le futur `purchase` n’est pas dans la portée du reste du code. Il ne peut être exécuté qu’à partir du rappel `onComplete`. Cela signifie que **d’autres parties de l’application ne voient pas ce futur** et ne peuvent pas enregistrer un autre rappel, par exemple, pour vendre une autre devise.

Composition fonctionnelle

Nous pouvons utiliser le combinateur `map` de `Future` pour prendre le résultat du premier `Future` et *appliquer* une fonction pour décider si elle est rentable puis générer un nouveau `Future`:

```
val rateQuote = Future {
  connection.getCurrentValue(USD)
}
val purchase = rateQuote map { quote =>
  if isProfitable(quote) then connection.buy(amount, quote)
  else throw new Exception("not profitable")
}
purchase onComplete {
  case Success(value) => println("Purchased " + value + " USD")
  case Failure (t) =>  println("An error occurred: " + t.getMessage)
}
```

`map` sur `Future`

Les futures fournissent des combinateurs (`map`, `flatMap` et `filter`) qui permettre une composition simple.

La méthode `map` mappe la valeur dans un future à une valeur dans un autre future:

```
def map [S] (f: T => S) (e implicite: ExecutionContext): Future[S]
```

Cette méthode n'est pas bloquante: elle retourne l'objet `Future[S]` immédiatement. Une fois le future d'origine terminé avec une valeur `x`, l'objet `Future[S]` retourné est finalement complété par `f(x)`.

Vous pouvez raisonner sur le map des futurs de la même manière que vous raisonnez sur le map des collections.

Comment fonctionne le `map` sur `Future`?

- Si le `Future` d'origine a terminé avec succès, alors le `Future` retourné est complété avec une valeur mappée du `Future` d'origine.
- Si la fonction de mappage lève une exception, le `Future` est terminé avec cette exception.
- Si le `Future` d'origine échoue avec une exception, le `Future` retourné contient également la même exception. Cette sémantique de propagation d'exception est également présente dans le reste des combinateurs.

For comprehensions

L'un des objectifs de la conception des `Futures` était de permettre leur utilisation dans les *for comprehensions*. Pour cette raison, les `Futures` ont également le `flatMap` et les combinateurs `withFilter`.

La méthode `flatMap` prend une fonction qui mappe la valeur à un nouveau future `g`, puis retourne un future qui est terminé une fois que `g` est terminé.

Exemple: Supposons que nous voulons échanger des dollars américains contre des francs suisses (CHF).

- Nous devons obtenir des cotations pour les deux devises,
- puis décider sur l'achat basé sur les deux cotations.

For comprehensions

```
val usdQuote = Future { connection.getCurrentValue(USD) }
val chfQuote = Future { connection.getCurrentValue(CHF) }

val purchase = for {
  usd <- usdQuote
  chf <- chfQuote
  if isProfitable(usd, chf)
} yield connection.buy(amount, chf)

purchase foreach { _ =>
    println("Purchased " + amount + " CHF")
}
```

Attention: Le future `purchase` est completé seulement quand `usdQuote` et `chfQuote` sont tous les deux complétés – ici cela dépend des valeurs de ces deux Futures donc son calcul ne peut pas commencer plus tôt.

Traduction de *for comprehensions*

On a déjà vu, les *for comprehensions* sont traduites en utilisant les combinateurs `map`, `flatMap` et `withFilter` correspondants.

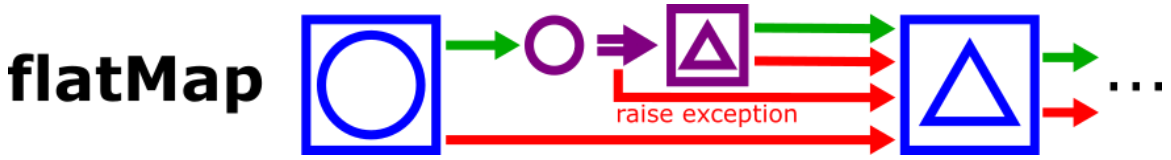
Voici la traduction de la *for comprehension* dans notre dernier exemple:

```
val purchase = usdQuote flatMap { usd =>
  chfQuote
      .withFilter(chf => isProfitable(usd, chf))
      .map(chf => connection.buy(amount, chf))
}
```

flatMap sur Future

```
def flatMap [S] (f: T => Future[S]) (implicit e: ExecutionContext): Future[S]
```

- Le combinateur `flatMap` utilise le future actuel avec le type `Future[T]` pour produire un autre future avec le type `Future[S]`.
- Le `Future[S]` résultant est complété en prenant la valeur `x` du type `T` du future actuel, et le mappage de cette valeur à un autre future `f(x)`.
- Alors que le `Future` résultant d'une méthode `map` se termine lorsque la fonction de mappage `f` est terminée, le `Future` résultant d'un `flatMap` se termine lorsqu'à la fois `f` et le `Future` retourné par `f` sont achevés.



HEIG VD

*Répond à la **question 2** sur https://etc.ch/2XRK*

Quiz

Dans l'exemple ci-dessous, Quels sont les types des variables a, b, et c?

```
def f(x: Int) = Future {
  println(s"creating future number $x")
  nextInt(1000)
}

val first = f(1)
val second = f(2)

val a = for {
  b <- first
  c <- second
} yield (b * c)
```

A Future[Unit], Int, Future[Int]

B Future[Int], Future[Int], Future[Int]

C Future[Int], Int, Int

D Int, Unit, Unit

Gérer les échecs

Comment gérer un Future complété par une failure ?

Les méthodes `map` et `flatMap` ont l'inconvénient qu'on ne peut pas gérer l'échec.

Regardons de nouveau les signatures de ces deux méthodes:

```
def map [S] (f: T => S) (implicit e: ExecutionContext): Future[S]
```

```
def flatMap [S] (f: T => Future[S]) (implicit e: ExecutionContext): Future[S]
```

Rappel : les méthode `map` et flatMap

La fonction `f` passée en paramètre est de type:

- **`T => S`** dans le cas de map, et
- **`T => Future[S]`** dans le cas de flatMap.

Dans les deux cas, `f` permet seulement de transformer un type d'entré `T`. `T` est le type de résultat réussi du premier future. C.à.d `f` ne permet pas d'agir en fonction de `Success` et `Failure`.

Si on veut avoir la possibilité de gérer aussi le cas d'échec, **on a besoin des méthodes qui prennent une foncion avec un type d'entré `Try[T]` et non `T`**.

Les méthodes **transform** et **transformWith** ont été mis en place avec ce but.

Les méthodes `transform` et `transformWith`

Les méthodes `transform` et `transformWith` sont des homologues de `map` et `flatMap`, avec la différence qu'elles permettent aussi de gérer les cas d'échec.

Voici leurs signatures:

```
def transform[S](f: Try[T] => Try[S])(implicit executor: ExecutionContext): Future[S]
```

```
def transformWith[S](f: Try[T] => Future[S])(implicit executor: ExecutionContext): Future[S]
```

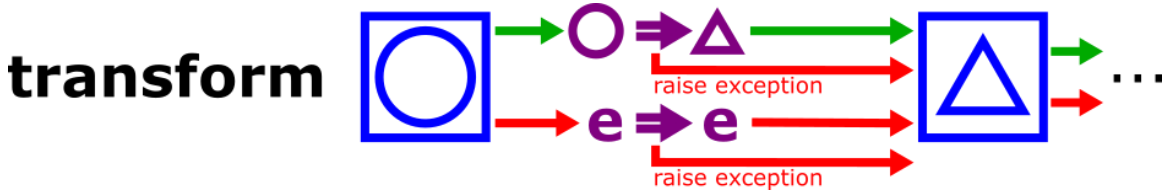
Comme attendu, le type de `f` est:

- Dans `transform`, **`Try[T] => Try[S]`**, au lieu de `T => S`
- Dans `transformWith`, **`Try[T] => Future[S]`**, au lieu de `T => Future[S]`

La méthode `transform`

La méthode `transform` permet de transformer un future de manière suivante:

- Avec une fonction qui accepte une valeur `Try` en entrée, on peut gérer à la fois un `Future` complété avec succès et un Future complété avec une exception.
- De plus, **la fonction passé en paramètre renvoie une autre valeur `Try`**, ce qui signifie que nous pouvons décider de:
  - retourner un succès, ou
  - retourner un échec



La méthode `transform` (suite)

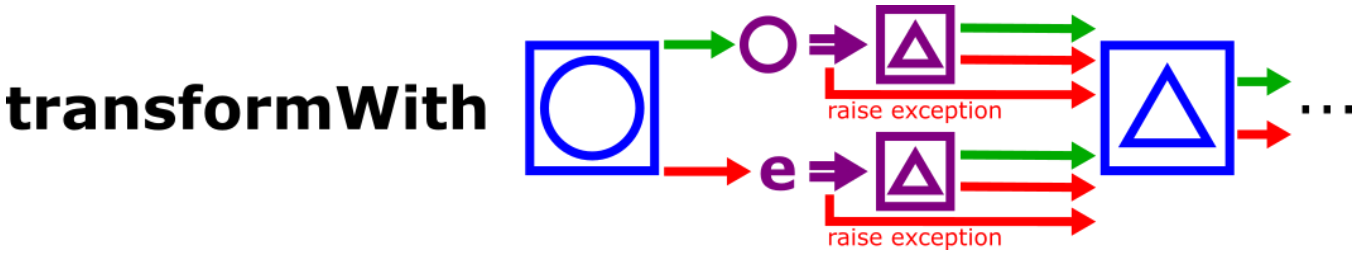
Voici un exemple de code avec `transform`:

```
future{...}
    .transform {
      case Success(result) =>
        ... // calculate something based on result
        Try(something)
      case Failure(error) =>
        // either reflect the failed result with Failure(error)
        Failure(error)
        // or do something else
        Try(somethingElse)
        // the result can be either a Success or a raised exception
    }
```

La méthode `transformWith`

```
def transformWith[S](f: Try[T] => Future[S])(implicit executor: ExecutionContext): Future[S]
```

La méthode `transformWith` est similaire à la fonction `transform`, sauf la fonction qui lui est passée en paramètre renvoie un nouveau `Future` au lieu d'un `Try`.



La méthode `transformWith` (suite)

Voici un exemple de code avec `transformWith`:

```
future{...}
    .transformWith {
      case Success(result) =>
        ... // calculate something based on result
        Future(something)
      case Failure(error) =>
        // either reflect the failed result with Future.failed(error)
        Future.failed(error)
        // or do something else asynchronously that can fail or succeed
        Future(somethingElse)
    }
```

Les méthodes `recover` et `recoverWith`

Il est aussi possible de **seulement s'intéresser au cas d'échec** d'un future et agir en fonction.

Les méthodes `recover` et `recoverWith` ont ce but.

Voici leurs signatures:

```
def recover[U >: T](pf: PartialFunction[Throwable, U])
   (implicit executor: ExecutionContext): Future[U]
```

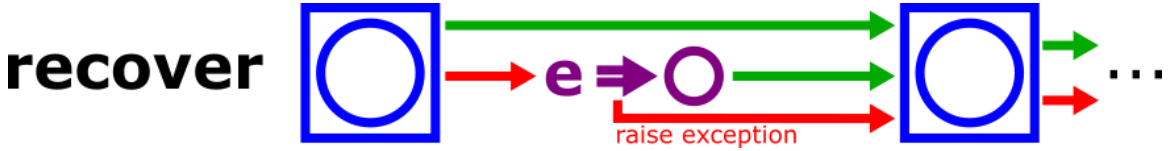
```
def recoverWith[U >: T](pf: PartialFunction[Throwable, Future[U]])
   (implicit executor: ExecutionContext): Future[U]
```

Note: La fonction qu'on fourni à `recover` et `recoverWith` est une fonction partielle.

La méthode recover

La méthode `recover` permet de transformer le future en un nouveau future. Celle-ci va:

- gérer les exceptions qui correspondent à la fonction partielle `pf`.
- s'il n'y a pas de match, ou si le future termine avec succès, le nouveau future va contenir les même résultats.



Exemple:

```
Future (6 / 0) recover { case e: ArithmeticException => 0 } // result: 0
```

```
Future (6 / 0) recover { case e: IOException   => 0 } // result: ArithmeticException
```

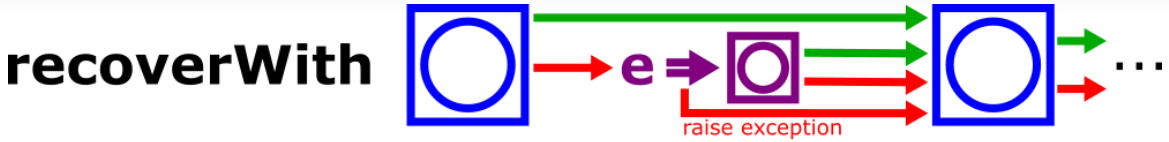
```
Future (6 / 2) recover { case e: ArithmeticException => 0 } // result: 3
```

La méthode `recoverWith`

La méthode `recoverWith` créer un nouveau future. Celui-ci va:

- gérer les exceptions qui correspondent à la fonction partielle `pf`, en lui attribuant une valeur d'une autre future.
- s'il n'y a pas de match ou si le future termine avec succès, le nouveau future va contenir les même résultats.

```
val f = Future { Int.MaxValue }
Future (6 / 0) recoverWith { case e: ArithmeticException => f }
// result: Future(Success(Int.MaxValue))
```



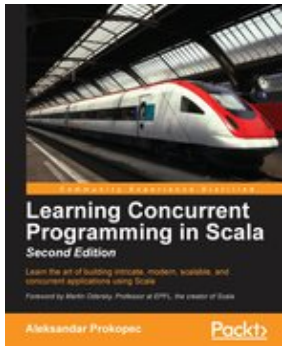
Autres combinateurs

Les `Futures` viennent avec d'autres combinateurs tels que `filter`, `fallbackTo`, ou `zip`, mais nous ne les couvrons pas ici.

La compréhension des combinateurs de base vous sera utile pour apprendre les autres méthodes.

Si vous souhaitez, vous pouvez étudier les combinateurs restants dans la documentation de l'API : https://dotty.epfl.ch/api/scala/concurrent/Future.html

Références



Learning Concurrent Programming in Scala Second Edition, Aleksandar Prokopec, Packt Publishing, February 22, 2017.

Futures, Scala documentation: https://docs.scala-lang.org/overviews/core/futures.html