Corso di Programmazione Concorrente

Programmazione Asincrona

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Introduzione
  - Task CPU vs I/O *intensive*
- *Computazioni Asincrone ed Operazioni non-blocking*
  - ***java.util.concurrent.Future***
- Trasformare una API sincrona in asincrona
  - **java.util.concurrent.CompletableFuture**
- Esempio: *best-price-finder* application
  - *Parallel Stream* vs *CompletableFuture*
- Dimensionamento di **ExecutorService** per task I/O intensive
- Implementazione di Diagrammi delle Precedenze tramite **CompletableFuture**
- Composizione di ***Future***

Introduzione

- Due evidenti tendenze degli ultimi anni costringono a ripensare il modo in cui si scrive software:

  1) Diffusione di architetture multi-core
     - ha reso evidente a tutti l'importanza della PC
     - ...anche se molti percepiscono immediatamente *solo* l'utilità della decomposizione parallela
  2) Diffusione di servizi internet accessibili tramite API pubbliche
     - Ad esempio:
       - *Google*: servizi di localizzazione, traduzione
       - *Facebook*: social information
       - *Twitter*: news
       - …
     - Le applicazioni moderne usano servizi remoti (anche *esterni* all'applicazione stessa) e contenuti aggregati da diverse sorgenti

Task CPU vs I/O *Intensive*

- Pertanto, più frequente l'uso di I/O e distribuzione
- Situazioni diverse che richiedono soluzioni diverse:
  - decomporre parallelamente *task CPU-intensive* su una singola (locale ed affidabile) macchina
    - si può utilizzare il Fork/Join Framework
    - l'obiettivo è mantenere tutte le CPU il più possibile impegnate
  - eseguire *task I/O-intensive* per accedere a molteplici servizi (remoti ed inaffidabili)
    - l'obiettivo è non dissipare il tempo di CPU nell'attesa di risultati restituiti da remoto
    - l'obiettivo è gestire i fallimenti (che non sono più visti come *eccezionali*) senza conseguenze irrimediabili per gli utenti finali
    - si possono utilizzare i nuovi

      `java.util.concurrent.CompletableFuture` >>

      disponibili a partire da Java 8

java.util.concurrent.*Future*

- Interfaccia introdotta in Java 5
  - nel contesto del *Lightweight Executable Framework*
- Modella un risultato disponibile solo successivamente alla corrispondente richiesta di esecuzione, in un certo momento futuro: *computazione asincrona*
  - comune nelle gestione di dispositivi periferici
  - riscoperta nel nuovo contesto dei servizi internet che espongono API pubbliche
- Gestisce l'*hand-off* del risultato di una computazione tra
  - un *worker-thread* (sottostante un *ExecutorService*)
  - il thread (client) che vuole consumare il risultato
    - spesso quello che aveva sottomesso il corrispondente *task* all'*ExecutorService*

`java.util.concurrent.`*`Future`*

- Interfaccia minimale in Java 5
- Forse troppo?
  - Non era nemmeno prevista la possibilità di scrivere esplicitamente il risultato (perché non immediatamente utile nel contesto del *LEF* versione Java 5)

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`boolean`	`cancel(boolean mayInterruptIfRunning)` Attempts to cancel execution of this task.
`V`	`get()` Waits if necessary for the computation to complete, and then retrieves its result.
`V`	`get(long timeout, TimeUnit unit)` Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
`boolean`	`isCancelled()` Returns `true` if this task was cancelled before it completed normally.
`boolean`	`isDone()` Returns `true` if this task completed.

Future: Esempio di utilizzo

- Modalità di utilizzo pre-Java 8

```
ExecutorService executor = Executors.newFixedThreadPool();
// submit(): chiamata non-bloccante
Future<Double> future = executor.submit(new Callable<Double>() {
        public Double call() {
           return eseguiUnPesanteElungoCalcolo();
        }
    });
maRestaLiberoDiFareAltro();
// quando serve il risultato, punto di sincronizzazione
try {  // get(): chiamata bloccante
       Double result = future.get(1, TimeUnit.SECONDS);
} catch (ExecutionException ee) {
    // exception durante il calcolo
} catch (InterruptedException ie) {
    // worker thread "istigato al suicidio"
} catch (TimeoutException te) {
    // timeout
}
```

Checked Exception

java.util.concurrent.*Future*

- In Java 8 è stata introdotta una nuova implementazione:

  `java.util.concurrent.CompletableFuture`

- E' una significativa estensione di ***`Future`***:
  - consente il completamento esplicito
  - recepisce l'introduzione del lambda calcolo avvenuta in Java 8
  - composizione di funzioni come base per la specifica di computazioni asincrone (cfr. *Stream*).

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage<T>, Future<T>

- La nuova interfaccia ***`CompletionStage`*** (ed i suoi 38 metodi) evidenziano che **`CompletableFuture`** è una significativa estensione più che una semplice implementazione di ***`Future`***

Sincrono → Asincrono (1)

- Esempio: metodo **`Shop.getPrice()`**
- L'invocazione di **`getPrice()`** è *bloccante*, ovvero sincrona

```
// un metodo sincrono
public double getPrice(String product) {
    return calculatePrice(product);
}

// Simulazione del calcolo di un prezzo
private double calculatePrice(String product) {
   delay();
   return random.nextDouble() * product.charAt(0) + product.charAt(1);
}

// Simulazione di una reale operazione I/O di lunga durata
public static void delay() {
    try {
        Thread.sleep(1000L);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Sincrono → Asincrono (2)

- L'invocazione di **`getPriceAsync()`** è *non-bloccante*, ovvero asincrona

```
// metodo asincrono
public Future<Double> getPriceAsync(String product) {
        CompletableFuture<Double> futurePrice =
                new CompletableFuture<>();
        new Thread( () -> {
                double price = getPrice(product);
                futurePrice.complete(price);
        }).start();
        return futurePrice;
}
```

Functional Interface: **`Runnable`**

```
// invocazione di metodo asincrono
Shop shop = new Shop("BestShop");
Future<Double> futurePrice =
                shop.getPriceAsync("my favorite product");
threadInvocanteLiberoDiPassareAfareAltro();
…
// ora serve il risultato: classico punto di sincronizzazione
double price = futurePrice.get(); // chiamata bloccante
…
```

Sincrono → Asincrono (3)

- Il nuovo metodo **`complete()`** consente di finalizzare esplicitamente il valore che deve restituire il ***`Future`***
- Il metodo **`completeExceptionally()`** è complementare:
  - consente la terminazione anomala della computazione

```
// metodo asincrono
public Future<Double> getPriceAsync(String product) {
    CompletableFuture<Double> futurePrice =
        new  CompletableFuture<>();
    new Thread( () -> {
    try {
        double price = getPrice(product);
        futurePrice.complete(price);
    } catch (Exception ex) {
        futurePrice.completeExceptionally(ex);
    }}).start();
    return futurePrice;
}
```

java.util.concurrent.CompletableFuture

- Alcuni factory method statici permettono di semplicare:

```
// metodo asincrono
public Future<Double> getPriceAsync(String product) {
    return CompletableFuture.supplyAsync(
        () -> calculatePrice(product)
    );
}
```

Functional Interface: **Supplier<U>**

Method Detail

supplyAsync

```
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
```

Returns a new CompletableFuture that is asynchronously completed by a task running in the `ForkJoinPool.commonPool()` with the value obtained by calling the given Supplier.

Type Parameters:

`U - the function's return type`

Parameters:

`supplier - a function returning the value to be used to complete the returned CompletableFuture`

Returns:

`the new CompletableFuture`

Esempio: applicazione *best-price-finder*

- Supponiamo di disporre solo della versione *sincrona* del metodo **`getPrice()`**

```
List<Shop> shops = Arrays.asList(new Shop("BestPrice"),
                                 new Shop("LetsSaveBig"),
                                 new Shop("MyFavoriteShop"),
                                 new Shop("BuyItAll"));
```

- Versione *seriale* sfruttando gli *Stream*:

```
public List<String> findPrices(String product) {
  return shops.stream()
    .map(shop -> String.format("%s price is %.2f",
                               shop.getName(),
                               shop.getPrice(product)))
    .collect(toList());
}
```

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
MyFavoriteShop price is 214.13, BuyItAll price is 184.74]
Done in 4032 msecs
```

Parallel Stream vs *CompletableFuture* (1)

- Sfruttando i *Parallel Stream*:

```
public List<String> findPrices(String product) {
  return shops.parallelStream()
     .map(shop -> String.format("%s price is %.2f",
                                shop.getName(),
                                shop.getPrice(product)))
     .collect(toList());
}
```

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
MyFavoriteShop price is 214.13, BuyItAll price is 184.74]

Done in 1180 msecs
```

- I *CompletableFuture* permettono di strutturare più liberamente le computazioni asincrone

Parallel Stream vs *CompletableFuture* (2)

- Sfruttando i *CompletableFuture*:

```
List<CompletableFuture<String>> priceFutures =
    shops.stream()
      .map(shop -> CompletableFuture.supplyAsync(
           () -> String.format("%s price is %.2f",
           shop.getName(), shop.getPrice(product))))
      .collect(toList());
```

- Serve un secondo *Stream*!
  - Altrimenti una **`map()`** addizionale sul primo *stream* per raccogliere i risultati delle **`join()`** porterebbe alla serializzazione
    - **`join()`**: versione di **`get()`** senza *checked exception*

```
// Un secondo stream per "raccogliere" i risultati
return priceFutures.stream()
         .map(CompletableFuture::join)
         .collect(toList());
```

Parallel Stream vs *CompletableFuture* (3)

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
MyFavoriteShop price is 214.13, BuyItAll price is 184.74]
```

`Done in 2005 msecs`

- La soluzione basata su *parallel stream* sembra
  - Più semplice!
  - Più efficiente: *Parallel Stream* **`1180 msecs`** `<` *CompletableFuture* **`2005 msecs`**
- Ma basta aggiugere un quinto **`Shop`** per osservare un recupero
  - N.B. Supponiamo **`Runtime.availableProcessors()==4`**
- *Parallel Stream*: **`2177 msecs`**
- *CompletableFuture*: **`2006 msecs`**
  - Con nove **`Shop`**, similarmente: **`3143`** vs **`3009 msecs`** (*Parallel Stream* vs *CompletableFuture*)

`ForkJoinPool.commonPool()`

- I comportamenti delle due soluzioni finiscono per essere quasi identici in quanto entrambe sottomettono task equivalenti allo stesso ***`ExecutorService`***
  - **`ForkJoinPool.commonPool()`**
  - è l'***`Executor`*** finale in entrambi i casi
- Un ***`ExecutorService`*** *centralizzato* a livello di *JVM*
- Esecuzioni dei task a "gruppi" dimensionati come **`Runtime.availableProcessors()`**

Method Detail

commonPool

```
public static ForkJoinPool commonPool()
```

Returns the common pool instance. This pool is statically constructed; its run state is unaffected by attempts to `shutdown()` or `shutdownNow()`. However this pool and any ongoing processing are automatically terminated upon program `System.exit(int)`. Any program that relies on asynchronous task processing to complete before program termination should invoke `commonPool().awaitQuiescence`, before exit.

Returns:

the common pool instance

Dimensionamento di un ***ExecutorService***

- Le API consentono di modificare programmaticamente l'***ExecutorService*** usato dai **CompletableFuture** ma non quello usato dai *parallel stream*.
- Per il *common-pool* si possono usare alcune *system properties*, tra le quali:
  - **java.util.concurrent.ForkJoinPool.common.parallelism**
  - Si tratta comunque di una istanza condivisa a livello di *JVM*
    - non è raccomandibile cambiare il livello di parallelismo per un singolo utilizzo
  - Il valore di default è pensato per task CPU *intensive*
- Per configurare un singolo ***ExecutorService***:

```
private final Executor customExecutorService =
  Executors.newFixedThreadPool(parallelism,
    new ThreadFactory() {
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r);
            t.setDaemon(true);
            return t;
        } });
```

Impegnare le CPU con Task I/O *intensive*

- I task in questione sono (simulazioni di task) I/O *intensive*
  - le CPU restano largamente inutilizzate in attesa di operazioni di I/O
- Supponiamo che:
  - si voglia scalare sul numero di task I/O intensive (es. N_{shops}>50)
  - il singolo task utilizzi la CPU per solamente l'1% circa del tempo, mentre il restante tempo è dovuto ad attesa per I/O
- W/C ≈ 100 : *wait-to-compute-time ratio*
- Se volessimo impegnare tutte le N_{CPU} CPU disponibili per il 100% del tempo dovremmo creare almeno W/C *worker thread* per processare ciascuno un singolo task I/O intensive
  - in realtà i 100 thread difficilmente si spartiranno *"perfettamente"* il rispettivo 1/100 di utilizzo della CPU
    - senza tutto questo lavoro certamente non impegneremo tutte le CPU
    - non è vero, purtroppo, che così le impegneremo certamente tutte
- Quindi con:
  $N_{threads} = N_{CPU} * (1 + W/C)$
  esiste abbastanza lavoro per *sperare* di impegnare tutte le CPU

ExecutorService Specializzato

- Per l'esempio di prima $N_{threads}$=400 !
- Iniettando un esecutore specializzato sul problema:

```
private final Executor ioExec =
  Executors.newFixedThreadPool(Math.min(shops.size(), 100),
    new ThreadFactory() {
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r);
            t.setDaemon(true);
            return t;
        } });
```

- Da usarsi come ***ExecutorService*** dei `CompletableFuture`:

```
CompletableFuture.supplyAsync(() -> shop.getName() +" price is "+
                    shop.getPrice(product), ioExec);
```

- E' possibile stimare i tempi di esec. per $5 \leq N_{shop} \leq 400$
  - *Parallel Stream*: $\approx \lceil N_{shop} / N_{CPU} \rceil$ sec.
  - `CompletableFuture`: $\approx$ 1 sec.
- Ad es. per N_{shop}=9: **3143 msecs** vs **1022 msecs**

✔ N.B. Non è possibile iniettare l'esecutore dei *parallel stream*

Composizione di Task Asincroni

- Sinora i task creati dovevano essere eseguiti parallelamente senza particolari inter-dipendenze
- Molti *diagrammi delle precedenze* reali sono più articolati
- Supponiamo che un addizionale servizio remoto fornisca il *tax-rate*, da applicare per ogni vendita per ottenere il prezzo finale
- La necessità di un servizio dedicato e centralizzato è legato alla logica non banale che ne fissa la dinamica
  - aggiornamenti legislativi del *tax-rate*
  - *back-to-school* shopping days

...meglio centralizzare tale logica per tutti i negozi per non doverla progettare e mantenere per ciascuno

Piccolo Refactoring

```
// Nuova classe Price immutabile
public class Price {
    private String shop;
    private double price;
    public Price(String shop, Price price) {
        this.shop  = shop;
        this.price = price;
    }
    public Price applyRate(double rate) {
        return new Price(this.shop, this.price * (1 + rate));
    }
    public String toString() {
        return String.format("%s price is %.2f", shop, price);
    }
}

// Il metodo sincrono Shop.getPrice() viene così riscritto:
public Price getPrice(String product) {
   return new Price(this.getName(), calculatePrice(product));
}
```

Servizio *Tax-Rate*

```
public class TaxRate {
    public static Price applyTaxRate(Price priceTaxFree) {
        double taxRate = getTaxRate();
        return priceTaxFree.applyRate(taxRate);
    }
    private static double getTaxRate() {
        delay(); // simulazione di un servizio remoto
        return new Random().nextDouble();
    }
}
```

```
// Versione seriale
public List<String> findPrices(String product) {
   return shops.stream()
      .map(shop -> shop.getPrice(product)) // Shop → Price
      .map(TaxRate::applyTaxRate)          // Price → Price
      .map(String::toString())             // Price → String
      .collect(toList());
}
```

```
[BestPrice price is 133.12, LetsSaveBig price is 180.58,
MyFavoriteShop price is 225.72, BuyItAll price is 199.74, ShopEasy
price is 190.28]
Done in 10028 msecs
```

Esecuzione *Sincrona* vs *Asincrona*

- Diversi metodi che vedremo sono forniti in due varianti
  - **`thenCompose()`** vs **`thenComposeAsync()`**
  - **`thenApply()`** vs **`thenApplyAsync()`**
- E' possibile specificare dei task da far eseguire direttamente al *worker thread* "del *completamento"* del future ed altri da eseguire con un nuovo e separato task (esec. *asincrona)*
  - ✔ Attenzione: *task != worker-thread*

- Actions supplied for dependent completions of *non-async* methods may be performed by the thread that completes the current CompletableFuture, or by any other caller of a completion method.
- All *async* methods without an explicit Executor argument are performed using the `ForkJoinPool.commonPool()` (unless it does not support a parallelism level of at least two, in which case, a new Thread is used). To simplify monitoring, debugging, and tracking, all generated asynchronous tasks are instances of the marker interface `CompletableFuture.AsynchronousCompletionTask`.

Composizione di Computazioni Sincrone ed Asincrone

- Due interrogazioni asincrone dei servizi remoti
- Due operazioni sincrone sui loro risultati

```
public List<String> findPrices(String product) {
  List<CompletableFuture<String>> priceFutures =
      shops.stream()
        .map(shop   -> CompletableFuture.supplyAsync(
                         () -> shop.getPrice(product), ioExec))
        .map(future -> future.thenCompose( price ->
                           CompletableFuture.supplyAsync(
                     () -> TaxRate.applyTaxRate(price), ioExec)))
        .map(future -> future.thenApply(String::toString))
        .collect(toList());

   return priceFutures.stream()
        .map(CompletableFuture::join)
        .collect(toList());
}
```

CompletableFuture<T>.thenApply()

- Applicazione *sincrona* di una funzione
  - applica una funzione che trasforma il risultato di un **CompletableFuture** non appena disponibile. In effetti:
    - **CompletableFuture<T>** → **CompletableFuture<U>**
    - Mediante una funzione: **T** → **U**
  - es. per invocare **toString()** inutile generare un task separato

thenApply

```
<U> CompletionStage<U> thenApply(Function<? super T,? extends U> fn)
```

Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied function. See the `CompletionStage` documentation for rules covering exceptional completion.

Type Parameters:

`U - the function's return type`

Parameters:

`fn - the function to use to compute the value of the returned CompletionStage`

Class CompletableFuture<T>

`CompletableFuture<T>.thenCompose()`

- Composizione *sincrona*
  - applica una funzione che trasforma il risultato di un **`CompletableFuture`** non appena disponibile. In effetti:
    - **`CompletableFuture<T>`** → **`CompletableFuture<U>`**
    - Mediante una funzione: **`T`** → **`CompletableFuture<U>`**
  - Se ne considerassimo l'uso nell'esempio di prima (**`thenComposeAsync()`** al posto di **`thenCompose()`**)
    - **`TaxRate.applyTaxRate(price)`** si basa su un servizio remoto
    - la dipendenza del nuovo *future* dal risultato del precedente rende la questione quasi irrilevante (sono comunque eseguiti serialmente)
    - ✓ e le segnature sono tali che il codice non si semplificherebbe

thenCompose

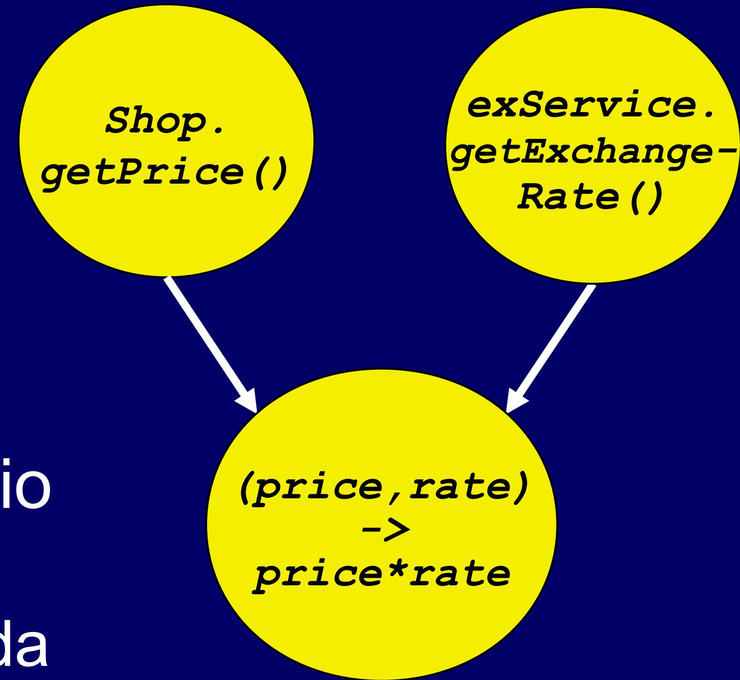
```
public <U> CompletableFuture<U> thenCompose(Function<? super T,? extends CompletionStage<U>> fn)
```

Description copied from interface: `CompletionStage`

Returns a new CompletionStage that, when this stage completes normally, is executed with this stage as the argument to the supplied function. See the `CompletionStage` documentation for rules covering exceptional completion.

Combinazione di Task

- Supponiamo di volere fornire i prezzi in EUR anche per **`Shop`** remoti in USD
- Un servizio di *exchange-rate* remoto fornisce il tasso di cambio corrente USD-EUR
  - come prima: logica non banale da creare e mantenere




```
Future<Double> futurePriceInUSD =
   CompletableFuture.supplyAsync(() -> shop.getPrice(product))
     .thenCombine(
          CompletableFuture.supplyAsync(
           () -> exService.getExchangeRate(Money.EUR, Money.USD)
          ) ,
        (price, rate) -> price * rate
));
```

CompletionStage

- La nuova interfaccia ***CompletionStage*** chiarisce la vera natura dei **CompletableFuture**
  - API ricca per specificare task di processamento (da eseguire *sincronicamente* od *asincronicamente)* tramite composizione di operazioni più semplici
    - fanno leva sull'introduzione del lambda calcolo
    - similarmente agli **java.util.stream.*Stream***

Interface CompletionStage<T>

All Known Implementing Classes:
CompletableFuture

---

public interface **CompletionStage<T>**

A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes. A stage completes upon termination of its computation, but this may in turn trigger other dependent stages.

Gestire la Latenza dei Servizi Remoti

- Una simulazione più credibile dei servizi remoti considera tempi di risposta variabili:

```
// Nuova versione di delay()
private static final Random random = new Random();
public static void randomDelay() {
  int delay = 500 + random.nextInt(2000);
  try {
    Thread.sleep(delay);
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}
```

- ✔ Meglio notificare all'utente un prezzo non appena disponibile, senza aspettare che già tutti lo siano

Gestire il Completamento di un `CompletableFuture`

```
public Stream<CompletableFuture<String>>
             findPricesStream(String product) {
   shops.stream()
      .map(shop    -> CompletableFuture.supplyAsync(
                 () -> shop.getPrice(product), ioExec))
      .map(future -> future.thenCompose( price ->
                        CompletableFuture.supplyAsync(
                 () -> TaxRate.applyTaxRate(price), ioExec)))
      .map(future -> future.thenApply(String::toString))
}
```

Per stampare un risultato appena disponibile:

```
long start = System.nanoTime();
CompletableFuture[] futures = findPricesStream("myPhone27S")
    .map(f -> f.thenAccept(
                s -> System.out.println(s + " (done in " +
               ((System.nanoTime()-start)/1_000_000) + " msecs)")))
    .toArray(size -> new CompletableFuture[size]);
CompletableFuture.allOf(futures).join();
System.out.println("All shops have now responded in " +
                    ((System.nanoTime()-start)/1_000_000) + " msecs");
```

`CompletableFuture<T>.thenAccept()`

- Composizione *sincrona*
  - applica una funzione che "consuma" il risultato di un **`CompletableFuture`** non appena disponibile. In effetti:
    - **`CompletableFuture<T>`** → **`CompletableFuture<Void>`**
    - Mediante una funzione: **`T`** → **`void`**
  - È solo una specializzazione della **`thenApply()`**
- Es. **`map()`** di future per stampare gli oggetti dello stream

```
findPricesStream("myPhone")
        .map(f -> f.thenAccept(System.out::println));
```

thenAccept

```
public CompletableFuture<Void> thenAccept(Consumer<? super T> action)
```

Description copied from interface: `CompletionStage`

Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied action. See the `CompletionStage` documentation for rules covering exceptional completion.

Composizione di *Future*

- **`CompletableFuture.allOf(futures)`**
  - Factory method che crea un nuovo **`CompletableFuture<Void>`**: si completa quando tutti i future componenti si completano

allOf

```
public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
```

Returns a new CompletableFuture that is completed when all of the given CompletableFutures complete. If any of the given CompletableFutures complete exceptionally, then the returned CompletableFuture also does so, with a CompletionException holding this exception as its cause. Otherwise, the results, if any, of the given CompletableFutures are not reflected in the returned CompletableFuture, but may be obtained by inspecting them individually. If no CompletableFutures are provided, returns a CompletableFuture completed with the value `null`.

Among the applications of this method is to await completion of a set of independent CompletableFutures before continuing a program, as in: `CompletableFuture.allOf(c1, c2, c3).join();`.

anyOf

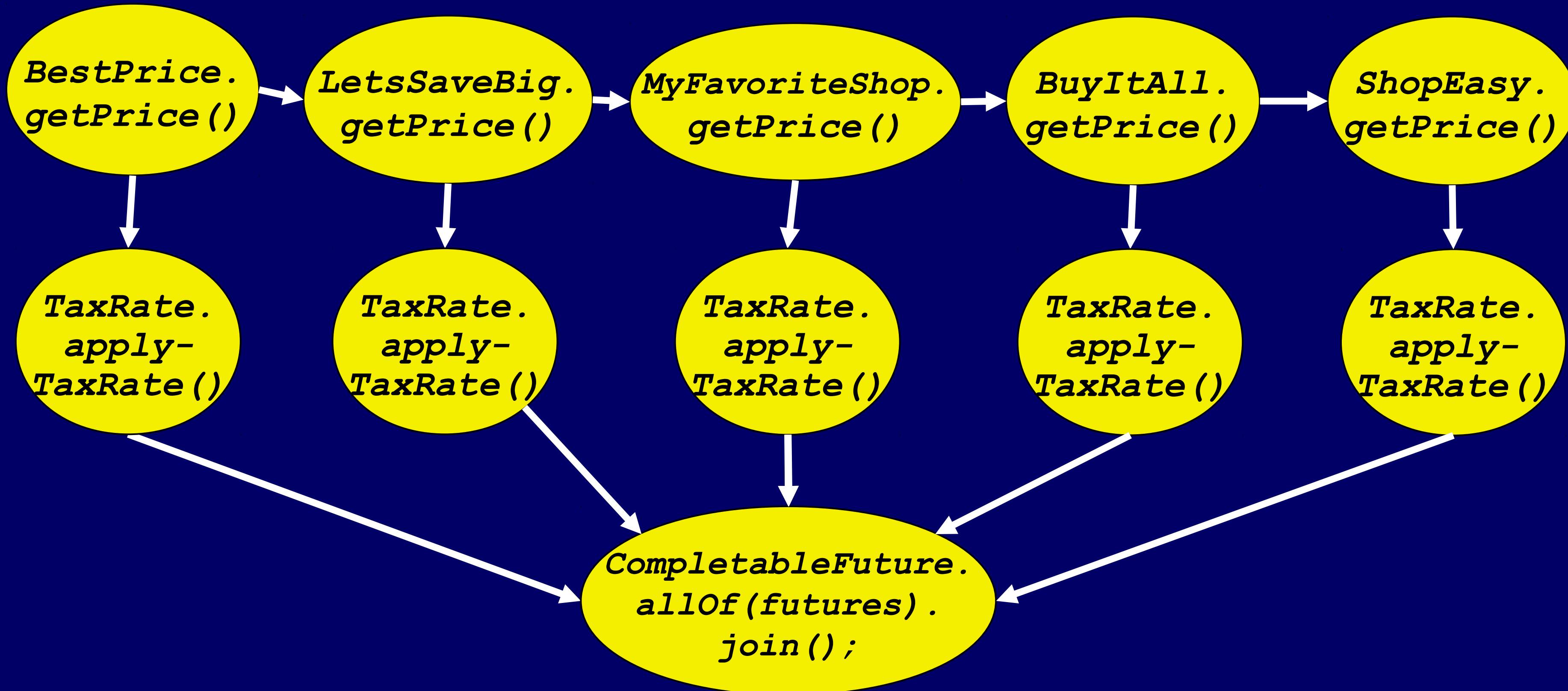
```
public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
```

Returns a new CompletableFuture that is completed when any of the given CompletableFutures complete, with the same result. Otherwise, if it completed exceptionally, the returned CompletableFuture also does so, with a CompletionException holding this exception as its cause. If no CompletableFutures are provided, returns an incomplete CompletableFuture.

Programmazione Asincrona e Diagramma delle Precedenze (1)

- I "nuovissimi" ***`CompletableFuture`*** disponibili da Java 8 possono essere visti come uno strumento programmatico per specificare
  - Un diagramma delle precedenze
    - componendo ("monadicamente") funzioni sui future
  - Quali e quanti task eseguire
    - cfr. metodi *synch* vs *asynch*
  - Sfruttando un (possibilmente implicito come `ForkJoinPool.commonPool()`) **`ExecutorService`** sottostante

Programmazione Asincrona e Diagramma delle Precedenze (2)




```
BuyItAll price is 184.74 (done in 2005 msecs)
MyFavoriteShop price is 192.72 (done in 2157 msecs)
LetsSaveBig price is 135.58 (done in 3301 msecs)
ShopEasy price is 167.28 (done in 3869 msecs)
BestPrice price is 110.93 (done in 4188 msecs)
All shops have now responded in 4188 msecs
```

Riferimenti

- Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and functional-style programming* – Manning – Capitolo 11