Distribuzione E Parallelismo

- L'hardware moderno disponibile consiste di processori multicore, quindi per migliorare l'esecuzione occorre far in modo che l'applicazione sia organizzata in più task paralleli
- La disponibilità di servizi distribuiti su Internet accessibili tramite APIs fa sì che le applicazioni possano essere sviluppate per richiedere dati da tali sorgenti
  - Le sorgenti esterne potrebbero essere lente a rispondere o non rispondere
  - Durante la richiesta ad un servizio remoto è bene non aspettare in modo bloccante la risposta, altrimenti si spreca tempo (il thread richiedente rimane in attesa)

Future

- L'interfaccia `Future` rappresenta un risultato che sarà disponibile in un momento successivo (futuro). Rappresenta una esecuzione asincrona ed è un riferimento al risultato
- L'avvio di un'attività per mezzo di un `Future` permette al thread chiamante di continuare a fare del lavoro, anziché aspettare il risultato dell'operazione

```
// prima di Java 8
ExecutorService esecutore = Executors.newCachedThreadPool();
Future<Double> risultato = esecutore.submit(new Callable<Double>() {
   public Double call() {
      return calcola();
   }
});
calcolaAltro();
```

- `Executors` e `newCachedThreadPool()` permettono di creare un nuovo thread, ovvero l'istanza di `ExecutorService`. Tale istanza esegue l'operazione richiesta (questa operazione prende tanto tempo per eseguire)
- `submit()` avvia un'operazione che ritorna un `Future` che rappresenta il risultato dell'operazione. La chiamata ritorna subito

Lettura Risultato Da Future

- Il thread aggiuntivo è fornito da `ExecutorService`
- Quando non si può più procedere senza il risultato dell'operazione che prende tanto tempo e che sta eseguendo sul thread aggiuntivo, si può prendere il risultato chiamando `get()` sul `Future` (ovvero sulla variabile `risultato`)
- `get()` restituirà immediatamente il valore se l'operazione era già stata completata, altrimenti il thread che chiama `get()` si blocca in attesa che il risultato diventi disponibile
- Il metodo `get()` con un parametro (overloaded) accetta un timeout che definisce il tempo massimo di attesa per il thread che aspetta il risultato. Se si raggiunge il tempo massimo di attesa si ha l'eccezione `TimeoutException`

Considerazioni Su Future

- `Future` non permette di implementare codice conciso quando
  - Si vuol usare il risultato della prima esecuzione asincrona in un'altra esecuzione asincrona [vedi `thenApply()`]
  - Si vuol aspettare il completamento di tutti i compiti svolti da un certo numero di esecuzioni asincrone [vedi `allOf()`]
  - Si vuol aspettare il completamento solo del più veloce fra i compiti svolti in modo asincrono e si vuol prendere il risultato [vedi `anyOf()`]
  - Si vuol fornire il risultato di un `Future` separatamente dal compito che era stato avviato [vedi `complete()`]
  - Si vuol essere notificati del completamento di un `Future`, così da poter avviare altre attività col risultato, non appena disponibile, anziché aspettare il risultato. Ovvero, reagire al completamento di un `Future` [vedi callback]

CompletableFuture e avvio thread

- Da Java 8 è disponibile `CompletableFuture`, il cui valore può essere impostato, e permette di avviare esecuzioni parallele
- Sia `calcolaPrezzo()` un metodo che accede ad un servizio remoto che ha un tempo di esecuzione lungo, `Double calcolaPrezzo(String s)`
- Per avviare un thread separato che esegue `calcolaPrezzo()`

```
CompletableFuture<Double> prezzo =
    CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto"));
```

- `supplyAsync()` prende in ingresso il tipo `Supplier`, che è un'interfaccia funzionale che definisce il metodo `get()`
- `supplyAsync()` avvia un thread per eseguire il `Supplier` che passiamo, esso restituirà il valore da inserire nel `CompletableFuture`, e ritorna il `CompletableFuture`
- Il `Supplier` verrà eseguito da uno dei thread degli `Executors`

CompletableFuture e get

- `get()` permette di estrarre il valore dal `CompletableFuture`
- `get()` aspetta se necessario per il completamento dell'operazione
- `get()` può lanciare delle eccezioni, quando: il thread è stato interrotto, l'esecuzione del thread è stata cancellata, o l'esecuzione ha generato un'eccezione

```
CompletableFuture<Double> prezzo =
    CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto"));

try {
   prezzo.get();
} catch (InterruptedException e) {
   e.printStackTrace();
} catch (ExecutionException e) {
   e.printStackTrace();
}
```


CompletableFuture e complete

- `isDone()` ritorna true se l'esecuzione è stata completata in uno dei possibili modi (normale, con eccezioni, cancellata)
- `complete()` imposta il valore del `Future` al parametro passato in caso l'esecuzione non sia già stata completata
  - Il valore impostato è quello che restituirà `get()`
  - Ritorna `true` se il valore è stato impostato, quindi lo stato è passato a completato

```
CompletableFuture<Double> prezzo =
    CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto"));

prezzo.complete(20d);
```


CompletableFuture e timeout

- `get()` permette di estrarre il valore da un `CompletableFuture`
- La variante con timeout di `get()` permette di aspettare un numero massimo di `TimeUnit` il completamento dell'operazione
- `get(long timeout, TimeUnit unit)`

```
prezzo.get(50L, TimeUnit.MILLISECONDS);
```

- La variante con timeout di `complete()` aspetta un certo tempo prima di completare con il dato valore, ritorna un `CompletableFuture`
- `completeOnTimeout(T value, long timeout, TimeUnit unit)`

```
prezzo.completeOnTimeout(20d, 50L, TimeUnit.MILLISECONDS);
```

- Il metodo `orTimeout()` aspetta un certo tempo l'esecuzione del thread e quindi completa con un'eccezione se il thread non ha completato prima, ritorna un `CompletableFuture`

```
prezzo.orTimeout(50L, TimeUnit.MILLISECONDS);
```

CompletableFuture e seconda operazione

- Si vuol usare il risultato di una prima esecuzione asincrona in un'altra esecuzione asincrona. La chiamata avviene in modo sincrono, ovvero non appena il risultato della prima esecuzione è disponibile
- Sia abbia un metodo con signature `String trasforma(Double d)`

```
CompletableFuture<Double> prezzo =
    CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto"));

CompletableFuture<String> prSconto = prezzo.thenApply(p -> trasforma(p));
```

- `thenApply()` prende in ingresso una funzione che usa il contenuto del `CompletableFuture prezzo`
- Quando il risultato di `calcolaPrezzo()` sarà disponibile, `trasforma()` sarà chiamato (e userà il risultato). `thenApply()` non blocca il codice chiamante
- Inoltre, in modo asincrono, il risultato di due operazioni asincrone (`calcolaPrezzo()` e `trasforma()`) verrà inserito in un unico `CompletableFuture prSconto`

Array di CompletableFuture

- Si vuol aspettare il completamento di tutti i compiti svolti da un certo numero di esecuzioni asincrone. Primo passo: si avviano le esecuzioni asincrone con `supplyAsync()` e si mette il risultato su un elemento di un array di `CompletableFuture`
- Quindi, si usa il metodo statico `allOf()` che ritorna un `CompletableFuture<Void>` che è completato quando tutti gli elementi dell'array hanno completato
- Infine, si può usare `join()` per aspettare che tutte le esecuzioni abbiano completato, quest'ultimo ritorna un `Void`

```
CompletableFuture<Double>[] prezzi = new CompletableFuture[2];
prezzi[0] = CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto0"));
prezzi[1] = CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto1"));
CompletableFuture.allOf(prezzi).join();
try {
   System.out.println(prezzi[0].get());
   System.out.println(prezzi[1].get());
} catch (InterruptedException e) {
   e.printStackTrace();
} catch (ExecutionException e) {
   e.printStackTrace();
}
```


Array di CompletableFuture

- Si vuol aspettare il completamento solo del più veloce fra i compiti svolti in modo asincrono e si vuol prendere il risultato
- Si usa il metodo statico `anyOf()` per avere un `CompletableFuture` che è completato quando uno degli elementi dell'array ha completato. Il `CompletableFuture` contiene il risultato di chi ha completato

```
CompletableFuture<Double>[] prezzi = new CompletableFuture[2];
prezzi[0] = CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto0"));
prezzi[1] = CompletableFuture.supplyAsync(() -> calcolaPrezzo("prodotto1"));
CompletableFuture<Object> p = CompletableFuture.anyOf(prezzi);
```

- Il tipo `CompletableFuture` ritornato da `anyOf()` contiene un `Object`

Callback

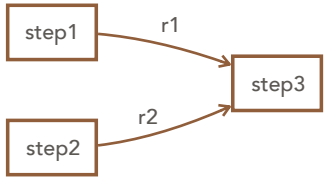
- Sono disponibili vari metodi di `CompletableFuture` per permettere di registrare una callback, ovvero una chiamata quando un evento si verifica
- Registrazione callback per mezzo di `thenApply()`
  - In un esempio precedente, il metodo `trasforma()` è stato registrato con `thenApply()` ed è chiamato quando l'esecuzione precedente è completa, e il valore di ritorno (`prezzo`) diventa disponibile
- Registrazione callback per mezzo di `thenAccept()`
  - Il metodo `thenAccept()` prende come parametro un `Consumer`, ovvero un metodo che consuma il valore del `CompletableFuture`

```
prSconto.thenAccept(System.out::println);
```

  - Il metodo `thenAccept()` restituisce un `CompletableFuture<Void>`

Alcuni Step Di Esecuzione Asincrona

- Spesso occorre eseguire più passi (step) di esecuzione per completare un lavoro (task o workflow)
- Supponiamo che due step indipendenti step1 e step2 siano avviati ciascuno in un thread dedicato, ciascuno con il proprio tempo d'esecuzione
- I risultati prodotti (r1 e r2) da ciascuno step sono utili ad uno step successivo step3 che quindi deve aspettare l'esecuzione degli step precedenti per poter eseguire



Esempio Di Esecuzione Di Step

- Siano `stepPrezzo()` e `stepSconto()` due operazioni che possono essere avviate in parallelo (ovvero step1 e step2) e che restituiscono ciascuno un valore double, e sia `stepCalcola()` un'operazione (step3) che prende un parametro `double` e restituisce un `CompletableFuture<Double>`
- Soluzione 1

```
CompletableFuture<Double> fsc = CompletableFuture.supplyAsync(() -> stepSconto());
CompletableFuture<Double> fpr = CompletableFuture.supplyAsync(() -> stepPrezzo());
// blocking wait
CompletableFuture<Double> d = stepCalcola(fpr.join() * fsc.join());
System.out.println("prezzo scontato: " + d.join());
```

- Soluzione 2

```
CompletableFuture<Double> fsc = CompletableFuture.supplyAsync(() -> stepSconto());
CompletableFuture<Double> fps = CompletableFuture.supplyAsync(() -> stepPrezzo())
            .thenCombine(fsc, (a, b) -> a * b)
            .thenCompose(p -> stepCalcola(p));
System.out.println("prezzo scontato: " + fps.join());
```


Metodi Utili

- `thenCombine()` prende due parametri, un `CompletableFuture` e una funzione con due argomenti, e restituisce un `CompletableFuture`. Quando il `CompletableFuture` su cui è chiamato `thenCombine()` e quello passato come primo parametro hanno completato, allora i due risultati sono passati come argomento alla funzione fornita

```
thenCombine(fsc, (a, b) -> a * b)
```

- `thenCompose()` prende una funzione con un parametro, e restituisce un `CompletableFuture`. Quando l'esecuzione sul `CompletableFuture` su cui è chiamato `thenCompose()` è completata, chiama la funzione passata e restituisce il `CompletableFuture` che è il risultato della funzione chiamata

```
thenCompose(p -> stepCalcola(p))
```