PARADIGMI DI PROGRAMMAZIONE

A.A. 2021/2022

Laurea triennale in Informatica

23: Java 19

Le nuove funzionalità previste per Java 19 sono di particolare interesse per i contenuti del corso.

Vediamo che cosa è (alla data di questa lezione) previsto per la release di Settembre 2022.

JSR VS JEP

Lo sviluppo della specifica della JVM e del linguaggio Java è governato tramite il JCP, Java Community Process.

Si articola in JSR, Java Specification Requests, che seguono un preciso percorso per essere approvate.

Parallelamente, nel 2011 nasce il JDK Enhancement Proposal, un processo riservato a chi contribuisce direttamente al JDK, per selezionare più agilmente le feature da sviluppare nelle varie versioni di Java.

Dall'inizio della cadenza semestrale delle release, è diventato uso usare le JEP per selezionare e pubblicizzare le nuove feature contenute in ogni release.

JDK 19

Lo sviluppo di OpenJDK è pubblicizzato su http://jdk.java.net.

I progetti attivi, le discussioni sulle JEP e la pubblicazione dei binari ufficiali avvengono su questo sito.

Data	Fase
2022/06/09	Rampdown Phase One
2022/07/21	Rampdown Phase Two
2022/08/11	Initial Release Candidate
2022/08/25	Final Release Candidate
2022/09/20	General Availability

Ad oggi le seguenti JEP sono individuate per l'inclusione nella nuova versione:

JEP	Titolo
424	Foreign Function & Memory API (Preview)
426	Vector API (Fourth Incubator)
422	Linux/RISC-V Port

JEP	Titolo
405	Record Patterns (Preview)
427	Pattern Matching for switch (Third Preview)
425	Virtual Threads (Preview)
428	Structured Concurrency (Proposed)

Le JEP 405 e 427 riguardano le evoluzioni sintattiche legate all'uso dei Record.

Le JEP 425 e 428 sono invece più strettamente legate all'argomento del corso.

RECORD PATTERNS

La JEP 405 Record Patterns propone la prima Preview di una sintassi per de-costruire i Record, ovvero utilizzare direttamente le loro proprietà.

Questa sintassi è uno dei passi per arrivare ad una implementazione completa del Pattern Matching.

Prima della JEP 394:

```
record Point(int x, int y);

public void print(Object o) {
  if (o instanceof Point) {
    var ix = ((Point)o).x()
    var iy = ((Point)o).y()
    System.out.println("x,y: " + ix + "," + iy);
  }
}
```

Dopo la JEP 394:

```
record Point(int x, int y);

public void print(Object o) {
  if (o instanceof Point p) {
    System.out.println("x,y: " + p.x() + "," + p.y());
  }
}
```

Con la JEP 405:

```
record Point(int x, int y);

public void print(Object o) {
  if (o instanceof Point(int ix, int iy)) {
    System.out.println("x,y: " + ix + "," + iy);
  }
}
```

Ulteriore esempio:

```
enum Color {RED, GREEN, BLUE}
record ColoredPoint(Point p, Color color) {}
record Point(int x, int y) {}
record Square(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

Sintassi possibile:

```
public void printUpperLeftColoredPoint(Square s) {
  if (s instanceof Square(
    ColoredPoint(Point(var x, var y), var color), var leftRight))
    ){
    // x, y, color, leftRight sono definiti
  }
}
```

Il lavoro su questa JEP è ancora in corso, ma il chiaro obiettivo è collegarsi ai lavori fatti con le JEP 406 Pattern Matching for switch (First Preview) e JEP 420 Pattern Matching for switch (Second Preview) per arrivare ad una sintassi equivalente ad altri linguaggi.

PATTERN MATCHING

Strettamente legato alla precedente è il lavoro della JEP 427 Pattern Matching for switch (Third Preview).

In seguito al feedback ottenuto dalle preview precedenti, viene proposta una nuova sintassi e maggiore precisione nella gestione dei tipi all'interno dell'espressione.

Sintassi JEP 420 (Java 18)

```
static void testTriangle(Shape s) {
  switch (s) {
      case Triangle t && t.calculateArea() > 100 ->
          System.out.println("Large triangle");
      case Triangle t ->
          System.out.println("Small triangle");
      default ->
          System.out.println("Non-triangle");
  }
}
```

Sintassi JEP 427 (Java 19)

```
static void testTriangle(Shape s) {
  switch (s) {
    case Triangle t when t.calculateArea() > 100 ->
      System.out.println("Large triangle");
    case Triangle t ->
      System.out.println("Small triangle");
    case null ->
      System.out.println("No shape");
    default ->
      System.out.println("Non-triangle");
  }
}
```

L'introduzione della parola chiave `when` rende più chiaro il ruolo dell'espressione guardia, e più versatile la costruzione dell'etichetta del ramo dello switch.

Viene inoltre meglio definita la semantica dell'uso di `null` e del calcolo di esaustività da parte del compilatore.

VIRTUAL THREADS

La JEP 425 Virtual Threads introduce in preview, con il nome di *virtual thread*, uno degli obiettivi finali di Project Loom: le *fiber*.

In Java, come abbiamo visto, l'oggetto `Thread` modella una linea di esecuzione concorrente all'interno dello stesso processo della JVM.

La sua implementazione è normalmente un sottile strato attorno ai servizi del sistema operativo.

Questo fa sì che Java sia soggetto agli stessi limiti di tutti gli altri processi: nella pratica, un numero di `Thread` superiore a 10000 può mettere a dura prova la gestione delle risorse dell'OS e portare ad un esaurimento di alcune risorse.

Per algoritmi con un *Blocking Factor* vicino a 1, legati prevalentemente al IO, questa situazione non è ottimale. Si vuole cercare una soluzione che permetta di scrivere codice che genera molti più `Thread` sapendo che questi passeranno una parte importante del loro tempo in attesa.

Un approccio è quello di riscrivere il codice in stile asincrono, in modo che le librerie dedicate possano gestire il passaggio delle operazioni fra i `Thread` che li eseguono. Sebbene questo porti a diversi vantaggi, è comunque un costo non trascurabile e richiede una formazione specifica.

Per questo motivo si cerca una *unità di concorrenza* più piccola dell'attuale `Thread`.

I *virtual thread* sono sintatticamente simili a `Thread` normali, ma al loro interno le primitive di comunicazione si comportano in modo diverso.

La JVM gestisce la messa in attesa del *virtual thread* (*unmounting*) durante l'esecuzione di una operazione di IO, per poi rimetterlo in esecuzione quando i dati sono disponibili (*mounting*).

In questo modo il paradigma di programmazione rimane in gran parte invariato, ma i *virtual thread* appaiono come economici e abbondanti. Si possono lanciare contemporaneamente anche centinaia di migliaia di operazioni distinte senza preoccuparsi di raggiungere qualche limite di sistema.

```java
try (var executor =
    Executors.newVirtualThreadPerTaskExecutor()) {
  IntStream.range(0, 10_000).forEach(i -> {
      executor.submit(() -> {
          Thread.sleep(Duration.ofSeconds(1));
          return i;
      });
  });
}
```

I *virtual thread* dal canto loro sono ottimizzati per essere usati una volta sola (non è necessario raccoglierli in *pool* per il riuso), e la maggior parte delle operazioni della libreria standard che hanno un comportamento bloccante ne provocano lo smontaggio in modo trasparente e rapido.

Per esempio, in questa istruzione:

```
response.send(future1.get() + future2.get());
```

il *virtual thread* può essere *smontato* tre volte.

Alcune operazioni non possono però smontare un *virtual thread*:

- Codice `syncronized`
- Codice `native` o interfacciato con una *foreign function*

La JEP si può appoggiare sul lavoro già fatto durante l'aggiornamento del sottosistema di IO (JEP 353 e JEP 373) per estenderlo al resto della libreria standard ed introdurre le nuove classi necessarie per accedere alla nuova modalità di lavoro.

Nel capitolo "Alternatives" della JEP, viene argomentato perché si è scelto per questo approccio, invece di introdurre il costrutto delle *coroutines*, che porterebbe alla stessa funzionalità:

Adding syntactic stackless coroutines (i.e., async/await) [...] would be new, however, and separate from threads, similar to them in many respects yet different in some nuanced ways. It would split the world between APIs designed for threads and APIs designed for coroutines.

Most languages that have adopted syntactic coroutines have done so due to an inability to implement user-mode threads (e.g., Kotlin), legacy semantic guarantees (e.g., the inherently single-threaded JavaScript), or language-specific technical constraints (e.g., C++). These limitations do not apply to Java.

L'obiettivo quindi della JEP è ottenere gli stessi vantaggi della concorrenza collaborativa, senza però esporrne all'utente l'esplicita gestione; solo il codice di libreria può maneggiare le fiber e definire i contorni dell'esecuzione concorrente.

Uno svantaggio di questo approccio è che, in caso di compiti legati alla CPU, si può ottenere una *unfairness* nello scheduling. Questo potrebbe spingere ad introdurre una esplicita sintassi di *yield*; oppure a dichiarare che questa funzionalità è esplicitamente indirizzata ai soli compiti di IO.

STRUCTURED CONCURRENCY

La JEP 428 Structured Concurrency (Incubator) è ancora *Proposed to Target*, ed è marcata come *incubating API*.

Il suo scopo è introdurre una libreria per *strutturare la concorrenza* e permettere di indicare più compiti eseguiti da thread differenti come un'unica unità di lavoro.

Esempio:

```
Response handle() throws ExecutionException, InterruptedException {
  Future< String >  user  = es.submit(() -> findUser());
  Future< Integer > order = es.submit(() -> fetchOrder());
  String theUser  = user.get();   // Join findUser
  int    theOrder = order.get();  // Join fetchOrder
  return new Response(theUser, theOrder);
}
```

Gli scenari di fallimento sono:

- se `findUser()` fallisce, `fetchOrder()` viene "perso"
- se `handle()` viene interrotto, entrambi i `Future` vengono persi
- se `findUser()` è lento e `fetchOrder()` fallisce, si aspetta (inutilmente) a lungo su `user.get()`

Esempio:

```
Response handle() throws ExecutionException, InterruptedException {
  Future< String >  user  = es.submit(() -> findUser());
  Future< Integer > order = es.submit(() -> fetchOrder());
  String theUser  = user.get();   // Join findUser
  int    theOrder = order.get();  // Join fetchOrder
  return new Response(theUser, theOrder);
}
```

Prendendo ispirazione da idee di altri linguaggi (per es. la supervisione degli attori) si vuole cercare una struttura sintattica che descriva la relazione esistente fra i task, in modo da poter prendere decisioni in base al loro reciproco comportamento.

```
Response handle() throws ExecutionException, InterruptedException {
  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future< String >  user  = scope.fork(() -> findUser());
    Future< Integer > order = scope.fork(() -> fetchOrder());

    scope.join();           // Join both forks
    scope.throwIfFailed(); // ... and propagate errors

    // Here, both forks have succeeded, so compose their results
    return new Response(user.resultNow(), order.resultNow());
  }
}
```

La sintassi proposta fa leva sulla semantica di `try-with-resources` ed introduce una insieme di funzionalità per ottenere automaticamente i comportamenti desiderati:

- *short-circuiting* degli errori
- propagazione della cancellazione
- chiarezza della sintassi
- osservabilità del risultato

L'implementazione attuale risiede in un *modulo* separato dal JDK, che va esplicitamente importato per poterlo usare.

E' indicata come dipendenza la JEP 425 Virtual Threads, in quanto le problematiche affrontate sono comunque legate a compiti di IO.

ALTRE JEP

Oltre a quelle di maggiore interesse per gli argomenti del corso, altre novità sono previste per Java 19:

JEP 426: VECTOR API (FOURTH INCUBATOR)

Istruzioni specifiche per la programmazione di architetture CPU SIMD.

Questa JEP studia l'introduzione di sintassi volta a sfruttare quelle architetture che possono elaborare sequenze di istruzioni su insiemi (vettori) di dati contemporaneamente.

Si tratta di argomenti di interesse per HPC, calcolo scientifico e ML.

JEP 424: FOREIGN FUNCTION & MEMORY API (PREVIEW)

Nuove modalità di interazione con risorse non-JVM.

Questa JEP intende affiancare all'attuale sistema della JNI una nuova API per interagire con codice e memoria al di fuori del processo della JVM.

Lo scopo è permettere di sfruttare la memoria *off-heap*, e raggiungere codice sviluppato con linguaggi diversi da C/C++.

Il confronto con i risultati ottenuti in altri linguaggi (per es. Python `ctypes`, Rust) ha reso lo stato di JNI non più sostenibile e ha richiesto l'avvio di una nuova iniziativa per puntare alla stessa sicurezza e facilità d'uso.

JEP 422: LINUX/RISC-V PORT

Supporto al set di istruzioni RISC-V.