PARADIGMI DI PROGRAMMAZIONE

A.A. 2023/2024

Laurea triennale in Informatica

24: Riepilogo

RIEPILOGO

Riassumiamo gli argomenti trattati durante il corso per evidenziare i più rilevanti.

JAVA

Il linguaggio Java è un linguaggio orientato ad oggetti, eseguito su di una virtual machine in modalità mista: interpretato, e gradualmente compilato durante l'esecuzione (JIT).

La Java Virtual Machine permette di eseguire lo stesso *bytecode* ottenuto da un programma in Java (o in un altro linguaggio) su ogni sistema operativo e piattaforma su cui è implementata.

La compilazione di un insieme di sorgenti Java è un processo deterministico, insensibile all'ordine con cui le classi si presentano al compilatore.

Ogni classe deve apparire unicamente in una determinata posizione nel filesystem: quindi è semplice per il compilatore cercare una classe e decidere se l'ha trovata o meno.

E' disponibile anche una modalità di compilazione statica che produce direttamente un eseguibile nativo, al fine di ridurre il tempo di avvio e la dimensione dell'eseguibile.

A regime, le prestazioni potrebbero essere leggermente inferiori, ma questo tipo di tecnica è pensata per esecuzioni molto brevi.

Il collegamento fra nomi e codice avviene dinamicamente al runtime: è considerato normale caricare nuovo codice durante l'esecuzione; in determinate condizioni è possibile sostituire, senza fermare il programma, codice esistente per caricarne una nuova definizione sotto lo stesso nome.

La sintassi di Java richiama i linguaggi della famiglia del C. Il codice è organizzato in **classi**, all'interno di **metodi** che sono identificati dal nome e dal numero e tipo degli argomenti.

I componenti di una classe Java possono avere una delle seguenti classi di visibilità:

- **private**: solo altri componenti della stessa classe
- **protected**: solo discendenti della classe
- **default**: altre classi dello stesso package
- **public**: tutte le altre classi

Gli errori applicativi sono gestiti tramite **eccezioni**, che sono oggetti non molto dissimili dagli altri.

E' obbligatorio dichiarare la possibilità per un metodo di lanciare alcuni tipi di eccezioni. La coerenza di queste dichiarazioni è controllata dal compilatore.

Il modello di ereditarietà è assimmetrico: singola nei confronti delle classi, multipla nei confronti delle interfacce.

Il compilatore considera un diamond problem un errore semantico bloccante.

Le classi definiscono un tipo di oggetto. Possono contenere variabili, metodi, inizializzatori e altre definizioni di classi.

Una classe può avere componenti propri, definiti come *static*.

Una **interfaccia** costituisce la dichiarazione di un contratto cui una classe può aderire. In questo senso, definisce un tipo.

Una interfaccia può essere implementata da una classe anonima, specificando immediatamente il corpo dei metodi richiesti.

Una interfaccia può contenere metodi completi di implementazione, per i quali le classi aderenti non hanno l'obbligo di fornire una implementazione.

Una interfaccia con un solo metodo è detta *functional interface*. Il compilatore permette di abbreviare la sintassi necessaria per istanziarla.

Una **annotazione** è una interfaccia speciale che può essere usata per aggiungere metadati a strutture sintattiche nel codice.

Tali metadati possono essere usati durante la compilazione o al runtime, per cambiare il comportamento degli utilizzatori del codice stesso.

Un **record** è uno speciale tipo di oggetto immutabile, per il quale il compilatore completa un insieme di metodi di default.

Ha a disposizione una sintassi di *pattern matching* che ne permette la decostruzione per semplificare alcune categorie di confronti.

Le classi possono accettare parametri di tipo detti **generici**. In questo modo è possibile scrivere codice largamente indipendente dallo specifico tipo usato durante l'esecuzione, facendo il minimo delle ipotesi possibile durante la scrittura e la compilazione.

Non tutti i valori in Java sono oggetti: esistono alcuni tipi detti **primitivi** la cui rappresentazione è più semplice, e non costituisce un oggetto.

Hanno di norma una versione cosiddetta *boxed* che invece ha tutte le caratteristiche di un oggetto vero e proprio, ma è meno efficiente da utilizzare.

L'assenza di un valore oggetto è rappresentata con il valore **null**. Un valore primitivo non può essere assente, quindi nessuna variabile di tipo primitivo può assumere il valore *null*.

Un tipo dotato di un numero limitato e definito a priori di valori è detto una **enumerazione**.

Le enumerazioni permettono di modellare tipi di dati specifici e dotati di pochi valori costanti.

Le espressioni in Java producono un valore di un certo tipo, primitivo od oggetto. Il compilatore può in molti casi dedurre il tipo del risultato di una espressione e quindi richiedere meno precisione nella sua indicazione.

Le istruzioni in Java non producono valore. La loro esecuzione è garantita semanticamente, ma non sequenzialmente: il compilatore può permettersi grandi libertà nel riorganizzare il codice a scopo di efficienza, a meno di non richiedere diversamente con precise sintassi.

Una **lambda expression** è una sintassi che può sostituire, in certe situazioni, una interfaccia con un singolo metodo per ottenere una scrittura più compatta e leggibile. Il compilatore si occupa di individuare il tipo richiesto e completare la dichiarazione sintetizzandola.

Le istruzioni condizionali comprendono le classiche istruzioni **if** e **switch**, oltre ad una sintassi di **switch** che è considerata una espressione, e quindi produce un valore.

Caratteristiche delle forme di *switch*:

Istruzione	**Espressione**
fall-through	stesso tipo
`break`	`yield`
non esaustiva	esaustiva

Le istruzioni di ciclo comprendono i classici **do**/**while**, **while**, **for**, oltre ad una sintassi di **for** in grado di riconoscere una collezione di oggetti ed attraversarla.

L'istruzione **try** è richiesta per circondare codice che lancia eccezioni specifiche e per dichiaraer come gestirle.

Una apposita sintassi permette di dichiarare istanze di oggetti che implementano **Closable**, per i quali verranno sintetizzate le corrette istruzioni di chiusura al termine del blocco indicato.

PARADIGMI

Il corso ha analizzato i seguenti paradigmi di programmazione:

Risorse/Scopo	**Diverso**	**Comune**
Comuni	*Concorrenza*	*Parallelismo*
Isolate	*Rete*	*Distribuzione*

Olte alle precedenti, sono stati presentati i paradigmi:

- Reattivo
- Attori

CONCORRENZA

Gestione di risorse condivise fra linee di esecuzione in competizione fra loro.

PARALLELISMO

Efficiente assegnazione di risorse condivise fra linee di esecuzione che si suddividono il carico di parti di un compito.

IN RETE

Interazione fra nodi con ruoli differenti separati da una rete di comunicazione.

DISTRIBUZIONE

Interazione fra nodi appartenenti ad uno stesso sistema che collaborano per il raggiungimento di uno scopo.

REATTIVO

Semantica più ricca per l'elaborazione asincrona di flussi di dati non limitati.

ATTORI

Semantica di alto livello per modellare sistemi distribuiti, aggressivamente concorrenti, orientati alla robustezza.

CONCORRENZA

Il paradigma concorrente nasce per motivazioni economiche allo scopo di sfruttare in modo più completo tutte le parti dell'hardware di una stessa macchina, mitigando le diverse velocità relative di CPU, memoria e canali di comunicazione.

I sistemi concorrenti devono mitigare quattro problematiche principali:

- Non determinismo
- Starvation
- Race conditions
- Deadlock

Le *condizioni di Coffman* sono necessarie perché si produca un deadlock:

- Mutual exclusion
- Resource holding
- No preemption
- Circular wait

Gli approcci alla concorrenza si possono distinguere in:

- Collaborativa
- Pre-Emptive
- Tempo reale
- Orientata agli eventi

Come in molti altri sistemi, una linea di esecuzione è rappresentata in Java da un oggetto **Thread**.

Un nuovo Thread può essere avviato per ottenere una linea di esecuzione indipendente e parallela al chiamante. La JVM termina correttamente solo quando tutti i thread creati dall'utente terminano.

Un Thread attraversa un insieme di stati:

- New
- Runnable / Running
- Blocked / Waiting / Timed Waiting
- Terminated

L'interfaccia **Runnable** definisce un compito che può essere eseguito da una linea di esecuzione.

L'interfaccia **Callable** definisce un compito che produce un risultato.

Un **Executor** rappresenta una strategia di gestione di Threads a cui possono essere consegnati Runnables o Callables perché siano eseguiti.

Richiedendo l'esecuzione di un Callable si ottiene un **Future**, cioè una rappresentazione del calcolo in corso, che può essere interrogato sul completamento e per ottenerne il risultato.

Per manipolare dati tramite più linee di esecuzione concorrenti è necessario usare tipi di dati specifici:

- Variabili *Atomic*
- Strutture dati concorrenti
- Variabili *Thread Local*

La libreria standard propone una struttura dati di **Stream** che può distribuire il consumo e l'elaborazione degli elementi in modo parallelo, se necessario.

La struttura può esaminare le caratteristiche di una catena di elaborazione e prendere decisioni su come eseguirla efficacemente.

L'interfaccia **Spliterator** può essere implementata per alimentare uno stream parallelizzabile.

L'interfaccia **Collector** invece è dedicata alle operazioni di riduzione di uno stream che, per efficienza, richiedono un accumulatore mutabile.

La valutazione del grado di parallelismo ottimale per una determinata pipeline di esecuzione deve tenere conto del **Blocking Factor** del compito da eseguire:

- uso costante della CPU: `BF = 0`
- blocco costante in attesa di I/O: `BF = 1`
- `#threads <= (#cores)/(1-BF)`

Di recente sono stati introdotti i *Virtual Threads* come alternativa più "leggera" e performante per alcune applicazioni.

Le API bloccanti della libreria standard possono, richiamate da un *Virtual Thread*, cambiare contesto più rapidamente.

La sincronizzazione di più linee di esecuzione richiede l'uso di primitive apposite; in ordine di complessità ed espressività:

- `syncronized`
- `wait()`/`notify()`
- **Locks**, **Conditions**
- **Semaphores**

DISTRIBUZIONE

Il paradigma distribuito nasce per ricercare:

- scalabilità superiore a quella raggiungibile su di un singolo nodo
- resistenza/ridondanza di fronte al guasto
- localizzazione più vicina agli utenti

Le problematiche principali del paradigma sono:

- concorrenza fra i nodi componenti
- asincronia degli eventi
- imperscrutabilità dei fallimenti

La comunicazione fra diversi nodi può essere modellata come uno scambio di messaggi.

Vari metodi cercano, con alterni successi, di avvicinare questo modello alla chiamata di un metodo di un altro oggetto.

SOCKET

L'oggetto **Socket** modella il corrispondente concetto di connessione TCP/IP: un flusso di dati non limitato, bidirezionale, asincrono.

I due lati della comunicazione devono concordare sul protocollo di scambio dati per gestire i turni di invio e lettura delle informazioni.

La comunicazione trasporta esclusivamente bytes: quindi, è necessario aggiungere informazioni, o codificarle nel protocollo, per ricavare da questi oggetti usabili.

Per es: encoding delle stringhe, layout delle informazioni, eccetera.

In attesa di I/O, il thread è bloccato; la sua prosecuzione dipende da un evento esterno.

Va gestita la corretta attenzione ad altri eventi che potrebbero essere di interesse per l'applicazione (interazione con l'utente, segnali di interruzione, eccetera).

DATAGRAM

L'oggetto **Datagram** modella il paccketto UDP corrispondente ad un messaggio singolo inviato ad uno o più destinatari.

Al contrario del Socket, non c'è connessione: l'invio è asincrono e non blocca il mittente.

Il ricevente si deve mettere in ascolto, attendendo la ricezione di un messaggio.

Per le caratteristiche intrinseche del protocollo, i messaggi di questo tipo hanno una affidabilità minore.

CHANNELS

Per offrire un'alternativa alla gestione manuale delle attese su Socket e Datagram, è stata introdotta l'astrazione del **Channel**.

In questa astrazione, una volta scelto il canale di comunicazione, si può predisporre l'azione da eseguire in risposta ad un evento di I/O, in modo da delegare al componente la gestione delle risorse in attesa.

Le chiamate sono completamente asincrone, e richiedono di propagare manualmente un oggetto di contesto per identificare e legare fra loro eventi che riguardano la medesima conversazione.

FALLACIES

Le difficoltà insite nel paradigma distribuito sono a volte ingannevoli, e portano a fare ipotesi che sono in realtà pesantemente false. La pratica ha suggerito 8 di queste ipotesi su cui è molto comune cadere in errore.

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn’t change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

FRAMEWORKS

Un framework è un insieme di componenti, tecnologie e scelte metodologiche che propone un insieme di soluzioni ad un determinato campo di problemi.

Nel caso dello sviluppo di applicazioni distribuite, un framework proporrà un modello di comunicazione più comodo delle primitive di base, ed un insieme di regole ed interfacce per interagirci.

Usare un framework è vantaggioso perché:

- si può usare una infrastruttura testata e realizzata con competenza
- si possono usare soluzioni già fatte a problemi comuni
- possiamo concentrarci sul codice che risolve il nostro problema

Usare un framework è svantaggioso perché:

- gli sviluppatori potrebbero non avere le nostre stesse priorità
- il nostro caso d'uso non è ben supportato
- possiamo incontrare errori e guasti per noi imprevedibili
- l'evoluzione del framework può richiederci costi di sviluppo non controllabili

STATO DISTRIBUITO

Le stesse motivazioni che hanno portato alla ricerca della distribuzione per l'esecuzione di un compito, sono valide anche per la conservazione di dati.

- disponibilità anche in caso di guasto parziale
- mole maggiore delle capacità di un solo nodo
- accessibilità da più posizioni geografiche

Nel caso in cui un sistema conservi il suo stato interno suddiviso fra i nodi che lo compongono, nasce il problema di mantenere questa rappresentazione coerente nel suo complesso. Tale problema è detto **problema del Consenso**.

Il problema richiede l'uso di particolari algoritmi che garantiscano l'affidabilità del consenso del sistema anche in presenza di alcune classi di errori o di guasti.

In letteratura, i più diffusi sono **Paxos** e **Raft**.

CAP THEOREM

Il Teorema CAP fornisce un limite alle funzionalità che un sistema distribuito può realizzare in caso di un certo tipo di guasti.

- **C**: Consistency
- **A**: Availability
- **P**: (Network) partition

Il teorema afferma che in caso di suddivisione della rete in due parti, il sistema deve scegliere se:

- mantenere la consistenza, sacrificando la disponibilità
- continuare a rispondere, accettando i conflitti

Una esensione considera anche il funzionamento nominale:

- **P**: in case of partitioning
- **A**: (choose between) availability
- **C**: and consistency
- **E**: else, when normal, choose between
- **L**: latency
- **C**: and consistency

CRDT

Per gestire il caso di modelli dati in cui più nodi devono poter contemporaneamente scrivere, è possibile utilizzare delle strutture dati CRDT.

Una struttura dati *Conflict-Free Replicated Data-Type* garantisce di avere sempre un modo per riconciliare scritture avvenute contemporaneamente su versioni diverse delle informazioni, in modo da poter sempre convergere su di un valore che le include tutte.

Le strutture dati CRDT sono abbastanza facili da usare, ma molto limitate nelle possibilità semantiche e a volte molto costose in termini di spazio e complessità di calcolo.

REATTIVITÀ

La reattività è il paradigma di programmazione che ha come obiettivo la realizzazione di sistemi fortemente asincroni ma comunque semplici da programmare, con una particolare enfasi nel mantenere molto bassa la latenza alla risposta a stimoli esterni.

REACTIVE EXTENSIONS

Per sfruttare la semplicità del modello di elaborazione dello Stream con maggiore potenza espressiva, le Reactive Extensions definiscono un modello semantico più preciso.

Vengono definiti quattro componenti astratti, ed un ampio insieme di operatori che permettono di usarli:

- Observable
- Scheduler
- Subscriber
- Subject

Ogni componente è attentamente definito sia nel funzionamento sia nel comportamento asincrono e concorrente. Questa modalità di utilizzo è di primaria importanza nella costruzione della semantica del sistema.

Il risultato è un metodo che si adatta a vari linguaggi e porta in essi la stessa potenza espressiva e facilità d'uso.

REACTIVE STREAMS

Costruendo sull'esperienza delle Reactive Extensions, i Reactive Streams ampliano la gamma di garanzie disponibili.

La pipeline di elaborazione gestisce esplicitamente la **back-pressure**, ovvero la possibilità di opporre resistenza alla velocità dei dati in ingresso.

La semantica che viene aggiunta prevede che un componente possa comunicare al precedente quanti dati è in grado di elaborare.

Questo consente di stabilizzare la capacità della pipeline di elaborazione e garantire che nessun componente venga soverchiato dal flusso di dati in ingresso, ottenendo maggiore affidabilità complessiva.

Inoltre, la possibilità che i componenti di elaborazione si trovino su nodi differenti è esplicitamente gestita, allo scopo di ricercare scalabilità ed efficienza ai guasti.

ATTORI

Nella ricerca di una affidabilità estremamente elevata, viene ideato un paradigma completamente nuovo, basato su di una architettura estremamente concorrente, trasparentemente distribuita e tollerante ai guasti.

Il paradigma si basa sulla modellazione della singola esecuzione come un **Attore**, con una semantica molto ben definita delle operazioni che un tale oggetto può fare, e delle garanzie che ha a disposizione durante l'esecuzione.

Un attore comunica con l'esterno esclusivamente tramite messaggi indirizzati ad altri attori.

Al suo interno, è completamente thread-safe: ogni risposta ad un messaggio verrà elaborata da un unico thread.

Nel rispondere ad un messaggio un attore può unicamente:

- cambiare il proprio stato interno
- creare nuovi attori
- inviare messaggi
- cambiare il proprio comportamento per il prossimo messaggio

Un attore può fallire in qualsiasi momento. L'attore che l'ha creato può ricevere un messaggio e decidere che strategia usare per garantire il proseguo del funzionamento del sistema.

Con questo tipo di primitiva è possibile realizzare sistemi estremamente performanti e resilienti, dotati di caratteristiche di scalabilità quasi lineare.

Programmare un attore è tuttavia molto complesso, e richiede una formazione ed una mentalità ad hoc, data la distanza dal normale modo di scrivere codice.

CONCLUSIONI

Motivazioni economiche e tecnologiche ci portano a dover eseguire il nostro codice in modo concorrente o distribuito. Per ottenere delle soluzioni corrette è necessario approcciare correttamente queste situazioni.

Gli strumenti di base sono molto legati allo specifico problema, ma spesso difficili da usare correttamente.

E' spesso utile cercare una astrazione di livello superiore, verificando quali garanzie sono mantenute.

L'ecosistema attorno a Java e alla JVM è in grado di coprire una vastissima gamma di problemi, in differenti modalità di esecuzione, con semplicità d'uso e grande compatibilità con ogni piattaforma di esecuzione.

La piattaforma JVM è molto adatta, e molto usata, per lo studio e la realizzazione di strumenti innovativi e di grande efficacia in ogni paradigma di programmazione.