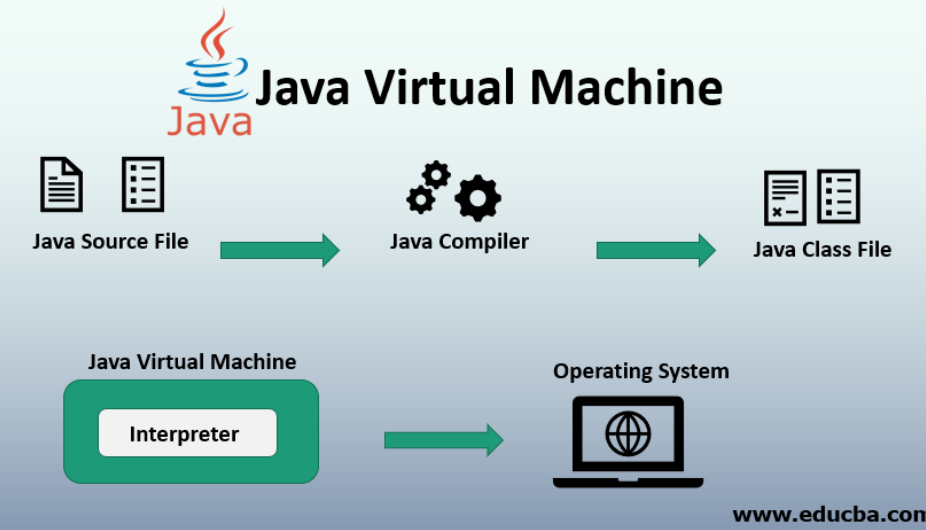
**28/02/2022: Lezione 1: Introduzione ai paradigmi, Java, JVM e concetti base**

Si parla di *paradigma di programmazione* o paradigmi più in generale come una tecnica o un insieme di tecniche per affrontare una classe di problemi (ad es. Von Neumann, con la CPU).

In particolare vedremo il paradigma della *concorrenza*, con più linee di esecuzione, asincrone, che condividono l’uso delle risorse in modo coordinato; contemporaneamente si parla di *parallelismo* con più linee di esecuzione che eseguono equivalentemente lo stesso calcolo su una partizione dei dati di ingresso. Altro paradigma è l’esecuzione *in rete*, con un calcolo distribuito su nodi che condividono risorse e mezzi di comunicazione per mezzo di un’interfaccia di rete.

Analogamente, la *distribuzione* prevede nodi indipendenti e, attraverso una rete non affidabile, coordinano l’esecuzione dello stesso lavoro in maniera consensuale su un certo sistema.

In ultimo, si considera la *reattività*, costruito sulle basi della comunicazione asincrona tramite messaggi da cui ottiene caratteristiche di flessibilità, resilienza e scalabilità, rimanendo quindi affidabile ed adattandosi nel corso del tempo.



Per cercare di studiare tutto ciò si usa Java, compilato attraverso il *bytecode*, linguaggio macchina intermedio interpretato dalla *Java Virtual Machine/JVM*. Esso è interpretato e volendo già compilato durante l’esecuzione (aka *JIT, Just In Time*) oppure prima della stessa esecuzione. La filosofia generale è *WORA* (Write Once Run Anywhere). Java si presta a tutti questi principi in linea di massima e di design, dato che la JVM stessa è sintomo di una idea di compilazione universale e semplice alle sue radici. Il linking della compilazione è dinamico, dato che guarda i nomi delle classi e dei metodi a tempo di compilazione/runtime. Il caricamento del codice viene gestito da una gerarchia di ClassLoader, dividendo il caricamento delle classi di libreria da quelle di estensione, separando la visibilità di classi particolare ed eventualmente permettendo il caricamento del codice da sorgenti differenti dal filesystem, per esempio da un URL. In particolare alla JVM è utile il *CLASSPATH*, quindi l’elenco delle locazioni in cui cercare una specifica classe.

Il comando *javac* invoca il compilatore e trasforma un file sorgente (di tipo *.java*) in un file bytecode (di tipo *.class*). I file sorgente devono chiamarsi come la classe in essi contenuta, il percorso delle directory deve corrispondere al loro package e, nei sorgenti o nel CLASSPATH, devono esserci tutti i tipi nominati dai sorgenti. L’ordine di compilazione, essendo una logica runtime, non è poi molto importante, dato che organizza tutto il compilatore stesso.

Il comando *java* avvia la JVM eseguendo il bytecode contenuto nel CLASSPATH; esso viene di norma contenuto dal file *.class*; similmente viene gestita anche l’archiviazione dei file e di codice Java con il comando *jar* che è stato per lungo tempo un formato di archiviazione utile, assieme a similari come *war* oppure *ear.* Utile notare che possono essere firmati per garantirne l’autenticità ed integrità.

Seguono alcune note storiche e commerciali: ricordiamo che Java è posseduta da Oracle e il software di tipo Java, per quanto in decadenza ormai, venga maggiormente utilizzato da fondazioni come Apache oppure Eclipse. Strumenti utili per costruzione e/o build usati anche in ambiti commerciali sono tuttora Apache Maven, che deve avere una chiara idea di come un progetto sia organizzato ed ogni libreria o componente può essere aggiunto ad un progetto indicandolo secondo delle coordinate: *gruppo:artefatto:versione*, volendo accedendo ad un repository remoto gestendo tutte le configurazioni degli strumenti della JVM per renderlo disponibile durante la programmazione.

Alternativamente a questo si usa Gradle, usando non più XML ma un linguaggio apposito di programmazione noto come Groovy, con un approccio generalmente più dinamico.

Entrambi i software sono tuttora esistenti e de facto sono i più utilizzati spartendosi il mercato.

**01/03/2022: Classi e tipi (1) - variabili, tipi, classi annidate, inizializzatori, tipi vari**

In Java l’unità principale di esecuzione sono le *classi*, che possono essere dei metodi con nomi o eventualmente blocchi anonimi. Una classe appartiene ad un *package*, che organizza le classi in maniera gerarchica. Un package normalmente è la prima linea non commento di un pezzo di codice e corrisponde ad una linea DNS scritta in ordine inverso. Dipende se appartiene allo stesso package (starà nella stessa directory) oppure sta in un altro package (quindi sta da un’altra parte). La JVM può essere configurata per impedire l’accesso a determinati insiemi di classi, magari anche per motivi di sicurezza.

Dichiarando una classe senza modificatori di visibilità, la classe è visibile solo nello stesso package su cui si sta lavorando e non a classi esterne allo stesso. Come al solito, si può usare la keyword *public* per poter essere ovunque visibile.

Si passa alle *variabili*, come al solito ciasscuna con il suo tipo, il nome e definiscono la struttura di un oggetto di una classe. Vi sono quindi due categorie di variabili:

1. di istanza, quindi ogni oggetto ha la propria copia e fa parte del suo stato
2. statiche, di cui ce ne sta una copia sola. Le variabili statiche vengono allocate ed inizializzate dal ClassLoader e preparate per l’uso.

Un esempio può essere:

Immagine che contiene testo

Descrizione generata automaticamente

Distinguiamo le variabili di tipo *public* visibili da qualsiasi altra classe caricata in cui, usando la direttiva *import*, si aggiunge al file sorgente il nome della classe importata.

Oltre alle variabili pubbliche, come sappiamo, possono esserci le variabili *protected*, lette e scritte da classi che estendono la classe, le variabili *private*, visibile solo dal codice della classe stessa, default, se visibili solamente da classi dal package stesso.

Altri modificatori indicati sulle variabili sono *final*, variabile che non può più essere modificata dopo la costruzione (e quindi richiedono obbligatoriamente un valore in fase di esecuzione), *transient*, variabile ignorata in sede di serializzazione, *volatile*, regolando l’accesso concorrente alla variabile.

Nota: le variabili hanno per convenzione nomi in Camel Case (con lettera minuscola iniziale), tranne le variabili *static final*, il cui nome si scrive solitamente in maiuscolo, magari con parole separate da underscore.

La classe organizza il proprio codice in metodi, con alcuni possibili modificatori, un tipo di ritorno, un nome, un elenco di parametri ed opzionalmente un blocco di eccezioni oppure anche un blocco di codice da eseguire. Con lo stesso nome della classe, solitamente ci sono i costruttori, chiamati quando si richiede la creazione di un oggetto specifico. La tupla formata da nome metodo, parametri di tipo ed elenco dei tipi degli argomenti è detta *firma/signature* del metodo. Se non si ritorna nessun valore, si avrà un ritorno di tipo *void*. Il compilatore controlla tutti i percorsi di esecuzione di un metodo. Il metodo dichiarato come statico è legato alla stessa classe; non può essere richiamato su un oggetto e non ha accesso alle variabili di istanza. Essi sono chiamati con la notazione *valore.nomemetodo(parametri)*. I costruttori stessi sono chiamati come al solito con la keyword *new.* Come al solito, i costruttori, se nessuno ne viene dichiarato, viene aggiunto quello di default; altrimenti, non viene proprio generato e quindi va creato manualmente assieme ad eventuali altri costruttori definiti dall’utente.

La gestione degli errori avviene con un sistema di *Tipi di Eccezione*, in cui le Eccezioni sono oggetti, ma vengono create ed usate in particolari situazioni. Non vengono più molto utilizzate e tuttora il loro utilizzo è fonte di discussione. Tutte le eccezioni derivano dalla classe *Throwable*, in cui vi sono due discendenti principali, in particolare *Exception*, errori nonostante i quali il programma prosegueed *Error*, errori per cui il programma non può proseguire. Sottoclasse particolare è la *RuntimeException*, quindi ogni errore recuperabile ed è lanciata direttamente dalla JVM. Errore tipico è la nullptr exception, variabile che non punta a nessuna istanza di oggetto. Solitamente non necessita di essere dichiarata e viene chiamata direttamente.

Eccezioni derivate da *RuntimeException* ed *Error* sono dette *unchecked exceptions* e non necessitano dichiarazione nella definizione di un metodo; tutte le altre, discendenti da *Exception* o *Throwable* sono *checked exceptions* e devono essere dichiarate nella definizione di un metodo.



Si parla poi di classi interne, dette *nested classes*, le quali posseggono le classiche 4 visibilità e sono una categoria di classi che si comporta in modo analogo agli altri casi. In queste abbiamo le *static nested classes*, classe dichiarata come statica all’interno di un’altra classe.

Altre classi interessanti di questo tipo sono le *inner classes*, parte dello stato di un oggetto del tipo ospite e, quindi, hanno lo stesso ciclo di vita ed ha un riferimento privilegiato all’oggetto ospitante. Detta in maniera più semplice, sono semplicemente classi non statiche dentro altre classi e non possono dichiarare membri statici ma solo membri di istanza. Le classi *static nested* sono spesso legate a qualche design pattern e solitamente vengono messe al di fuori della classe ospitante, per delineare meglio il legame esistente. Le inner classes vengono invece utilizzate come meccanismo di sicurezza e sono da usare con cautela, in quanto può essere complesso il loro utilizzo.

Esempi di static nested e inner classes:

Immagine che contiene testo

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

Come blocchi di codice anonimi abbiamo gli *inizializzatori*, eseguiti in sede di inizializzazione di classe o di oggetto e sono blocchi di codice anonimi. Possono essere *static*, eseguiti quindi lessicalmente al caricamento della classe stessa. L’uso dei blocchi statici non è comune in quanto potrebbe essere pesante per il caricamento del programma. I blocchi di inizializzazione privi di indicazioni sono eseguiti lessicalmente durante la creazione di ciascuna istanza di oggetto della classe. In particolare sono eseguiti dopo il supercostruttore (costruttore della superclasse) ma prima di qualsiasi altro costruttore.

In generale dato che sono anonimi meglio evitarne o quantomeno ridurne l’utilizzo; possono dar luogo a situazioni strane ed errori difficili da scovare.

Da quanto sembra, i blocchi anonimi possono anche essere usati, all’interno di una classe, per creare un tipo atto solo ad un contesto (quindi il metodo anonimo funge da tipo per un singolo oggetto). Da qui deriva l’effettiva ambiguità nell’utilizzo dei blocchi anonimi.

Un esempio di inizializzatori di istanza:

Immagine che contiene testo

Descrizione generata automaticamente

Java può ereditare esclusivamente da una sola superclasse, ereditandone codice e parte dello stato. Accede ai membri pubblici, a quelli protetti e ai package, non ai membri privati. Ciò evita il problema *dell’ereditarietà a diamante*, la selezione di un metodo che arriva da molteplici percorsi di ereditarietà. Una sottoclasse è anche un sottotipo della classe che estende, quindi usata quando richiesto dalla superclasse. Tutti i metodi di Java sono *virtual*, quindi tutti i metodi possibili all’esecuzione si scoprono a runtime. Ricordiamo che le sottoclassi si dichiarano con la keyword *extends*. Le classi invece dichiarate come *final* non possono essere usate come superclassi, quindi non ne deriviamo sottoclassi.

Contrariamente a questo, esiste la keyword *abstract* che dichiara che una certa classe debba essere usata come superclasse, quindi non istanziabile direttamente; ciò darebbe errore di compilazione. Abbiamo anche le classi dichiarate *sealed*, elencando i possibili sottotipi permessi (con la apposita keyword *permits*). Ciò venne fatto per introdurre il concetto di pattern matching, simile e possibile dalla programmazione funzionale, invocando specifiche funzioni solo in base a determinati contesti.

Il codice di dominio dovrebbe avere alberi di ereditarietà bassi, se possibile, dando maggior enfasi alla composizione in tutti i casi possibili. Implicitamente tutte le classi derivano da *java.lang.Object*, ereditandone alcuni metodi fondamentali, come *hashCode*, riconoscendo oggetti diversi tra di loro tramite una computazione hash, rilevata sulla memoria effettiva, *equals*, riconoscendo l’uguaglianza tra tipi di oggetti, *toString*, dando l’emissione di un oggetto visto come stringa quando viene implementato a console.

***07/03/2022: Classi e tipi (2) - interfacce, annotazioni, array, tipi primitivi, record (lezione 3)***

***Inizio delle istruzioni/espressioni in Java (lezione 4)***

Il primo costrutto che andiamo ad analizzare è l’*interfaccia*, che dichiara le caratteristiche di un tipo ma senza fornire una precisa implementazione; le classi che la dichiarano la devono implementare per contratto. Un’interfaccia può essere estesa solo da un’altra interfaccia; ciascuna di queste può avere visibilità pubblica o di package e i membri di ciascuna interfaccia sono pubblici.

Quando all’interno di una classe, è possibile modificarne il tipo di accesso; come al solito, è possibile contenga costanti, metodi astratti e statici. Implicitamente i metodi di un’interfaccia sono di tipo *abstract final*. Dopo Java 8, un’interfaccia può contenere anche metodi di default (essi reintroducono il Diamond Problem) oppure metodi privati. Segue un esempio:

Immagine che contiene testo

Descrizione generata automaticamente

Da un lato, la libreria standard necessitava di un aggiornamento stilistico; d’altro canto, però, non era possibile introdurre modifiche senza “rompere” tutto il codice esistente, soprattutto in caso di libreria standard. Naturalmente, nel caso di un’interfaccia, modificandone una occorre modificarne tutte le implementazioni. La soluzione è diventata parte della *JSR335*, introducendo il concetto di *default method*, metodi implementati che si comportano come metodi di superclassi. Per mezzo di questi un’interfaccia può essere modificata con nuovi metodi senza che le implementazioni debbano esserne toccate.

L’introduzione dell’interfaccia permette proprio di *evitare l’utilizzo di metodi*. Il Diamond Problem viene rilevato al momento della compilazione; se la gerarchia di ereditarietà comporta un’ambiguità nella selezione di un metodo, il compilatore segnala un errore. Nel qual caso si risolve solo *andando a modificare il codice*.

Vediamo un esempio di ereditarietà ed interfacce:

Immagine che contiene testo

Descrizione generata automaticamente

Altra cosa interessante, nel caso delle interfacce, il cui nome comincia con una chiocciola (@), dette *annotazioni*, applicate sintatticamente a classi, metodi e variabili e sono disponibili al momento dell’esecuzione. Arricchiscono le classi di *metadati,* consentendo la rilevazione durante l’uso e la compilazione rimanendo disponibili a runtime; grazie a questa cosa, a runtime è possibile avere una selezione delle classi per nome o per caratteristiche.

Annotazioni standard sono la *@Deprecated*, segnalando un metodo che verrà rimosso in futuro, oppure *@Override*, che segnala un membro che sostituisce o implementa un membro di superclasse o interfaccia. Se non implementata in quest’ultimo caso, darà errore.

Immagine che contiene testo

Descrizione generata automaticamenteAnalogamente abbiamo *@SuppressWarnings*, abbastanza autoesplicativo e usato quando effettivamente serve logicamente oppure *@FunctionalInterface*, usata se possibile nelle lambda espressioni qualora detenga un singolo metodo di utilizzo.

Esempio concreto qui a destra:

Le annotazioni possono avere parametri e anche metodi; il compilatore può eseguire degli *Annotation Processors*, producendo nuovi file a partire da singole annotazioni. Il loro uso è analogo alle macro.

Prima di Java 8, l’altra rivoluzione in Java fu la versione 5 del 2004, introducendo i cosiddetti *Generics/1-kind parametric types*.

Una classe o interfaccia/metodo viene definita *generica* quando dichiara uno o più parametri di tipo che possono essere specificati in seguito.

Un esempio segue:

Immagine che contiene testo, dispositivo, calibro

Descrizione generata automaticamente

Il parametro *T* può essere usato come un tipo, specie se non si sa ancora che tipo sia. Qualora venga creata un’istanza della classe, implementazione dell’interfaccia o chiamata del metodo, è necessario specificare un tipo concreto al posto del parametro. Si possono esprimere alcuni vincoli sui parametri come nel caso seguente (dove l’interfaccia viene implementata solo specificando un tipo che implementi l’interfaccia Comparable):

Immagine che contiene testo

Descrizione generata automaticamente

È anche possibile non esprimere vincoli sui parametri, ma solo sul tipo parametrizzato. L’uso più comune dei generici permette di evitare la duplicazione del codice e di semplificare l’uso delle classi contenitore; uso comune di questo costrutto è all’interno delle *collezioni*, quindi parliamo di liste, insiemi e code.

Tutte le informazioni sui tipi generici vengono cancellate a runtime.

Non tutto in Java è oggetto, essendo il focus iniziale verso le piattaforme embedded; i tipi di dato fondamentali infatti non sono oggetti ma vengono detti tipi *primitivi*, quindi tutti i tipi *char, long, int*, etc.

Un valore primitivo non è un oggetto e non può essere indicato come parametro di tipo; può essere usato per trasportare lo stesso valore nelle situazioni in cui si renda necessario.

In Java 5 è stato introdotto il concetto di *autoboxing*, dato che il compilatore riconosce il contesto d’uso del valore primitivo, applicando la trasformazione necessaria; tuttavia può rivelarsi pesante da un punto di vista elaborativo, ma può anche dare problemi di tipo semantico.

Gli *array*, invece, sono oggetti. La classe di un array discende direttamente da Object e viene creato dinamicamente come al solito nello heap. L’accesso ad un array è controllato a runtime, essendo la memoria di Java cosiddetta gestita. Un’altra categoria di classi utile è la *enum,* definito come tipo stringa che rappresenta numericamente un certo numero di elementi definiti alla dichiarazione. Essa non viene istanziata, ma ne si possono usare i valori.

In Java, viene introdotta la *named tuple*, chiamata *Record* in Java, implementando dalla programmazione funzionale il concetto di tupla, introdotta a tutti gli effetti da Java 16 (2021). La dichiarazione di un record è

*record Name(String firstName, String lastName){}*

Il record è *immutabile* e tutte le sue variabili sono considerabili *final* e non sono degli *Object*. Se dichiarati all’interno della classe, come sempre, saranno *static*. In automatico essi possiedono i membri privati con metodi di accesso pubblici, costruttore con tutti gli elementi del record e possiede metodi come *equals*, *hashCode, toString*.

I record non seguono lo standard OOP *JavaBean* e ad essi mancano metodi di modificazione di stato e di concorrrenza. In questo modo, il Record diventa una scorciatoria per definire tutte le classi che normalmente modellano valori di dominio trasportati da un punto ad un altro del sistema o scorciatoie di creazione delle classi, avvicinandosi alla forma mentis della programmazione funzionale, per esempio implementando il *Pattern Matching*.

Riassumiamo quindi i *modificatori di classe*¸sempre prima alla parola chiave *class* e sono naturalmente keywords comprese nel linguaggio.

Abbiamo:

* *abstract*, estesa da un’implementazione;
* *final*, non estesa da un’implementazione;
* *sealed*, medio tra *abstract* e *final*, permettendo solo a metodi specificati l’implementazione;
* *strictfp*, il codice della classe con operazione FP (floating point) restrittiva, implementando solo operazioni float/double.

Riassumiamo quindi i *modificatori di metodo*¸applicati appunto af un metodo specifico.

Abbiamo:

* *abstract*, estesa da un’implementazione di classe di estensione;
* *static*, legato alla classe e non ad un’istanza;
* *final*, non può essere reimplementato da una classe di estensione;
* *strictfp*, il codice della classe con operazione FP (floating point) restrittiva, implementando solo operazioni float/double;
* *native*, implementato da una libreria nativa;
* *synchronized,* il metodo può essere usato da un solo thread per volta.

Il codice in Java viene contenuto all’interno di blocchi delimitati da parentesi graffe, al cui interno stanno *statements* separati dal carattere “;”. Non ci sta quindi garanzia che l’ordine di esecuzione delle istruzioni sia lo stesso del codice. Un primo tipo di istruzione è la *dichiarazione*, dando un nome e un tipo, sia nel caso di variabili, classi, ecc. In quel caso sono dette locali.

Solitamente hanno un tipo, nome ed eventuale valore di inizializzazione; similmente ad *auto* in C++, il compilatore cerca di capire e dedurre il tipo da solo grazie alla keyword *var*. Può esserci anche una lista vuota di parametri <>, anche questa sarà dedotta dal compilatore.

Un’istruzione può produrre valori e quindi è un’*espressione*, che produce un valore anche nel caso questo non venga usato (ad esempio quando chiamano un metodo senza preoccuparci del suo valore di ritorno). Nella grande maggioranza dei casi, le espressioni hanno un comportamento atteso.

Altri esempi di valori sono i *valori letterali*, ad esempio i caratteri, le stringhe, i booleani, ecc.

Nei casi di *byte* e *short,* non è prevista la dichiarazione di una costante; semplicemente ne si forza il cast verso il tipo stesso.

Le costanti di tipo String possono essere su più righe (*text block*), sono oggetti senza bisogno dell’operatore *new* e sono immutabili. La parola chiave *null* indica il valore nullo e il riferimento che non punta a nessun oggetto. Viene sempre convertito in ogni altro oggetto e non si può esprimere in termini di valore.

Consegue l’assegnamento “=”, operatore che rappresenta un’istruzione di assegnazione dell’espressione ad una variabile specifica.

*La chiamata di un metodo* è un’espressione come le altre, ritornante un’espressione.

*La creazione di un oggetto* è per alcuni versi la chiamata di un metodo che ritorna il nuovo oggetto. Anche questa è un’espressione.

Una interfaccia può essere instanziata direttamente, fornendo l’implementazione al momento della creazione. Quindi (si noti che l’interfaccia è Comparator, parte dello standard):

Immagine che contiene testo

Descrizione generata automaticamente

Un’interfaccia di questo tipo può essere considerato tipo anonimo; il loro utilizzo è stato rimpiazzato dalle lambda espressioni. Operatori vari usati sono “+”, per concatenare stringhe, “[]”, subscripting ed accesso all’elemento i-esimo dell’array e l’operatore ternario, un *if* in linea diciamo.

Le parole chiave *this* e *super* permettono di usare nel primo caso l’istanza corrente, utile nel caso di ambiguità di denominazione o per capire usare il significato di una espressione, oppure l’utilizzo di un oggetto padre della superclasse in ordine di gerarchia nel secondo caso.

Caso d’uso del *this*:

Immagine che contiene testo

Descrizione generata automaticamente

Caso d’uso del *super*:

Immagine che contiene testo

Descrizione generata automaticamente

***08/03/2022: Lezione 4 – Istruzioni: condizioni, try/catch – Lezione 5 – Libreria standard: moduli***

Lynn Conway 🡪 OOOE (Out of Order Execution) all’interno della CPU.

Una delle più grandi novità di Java 8 furono le *lambda expressions*, rendendo più semplici e compatte espressioni verbose. Vengono usate sia nelle pratiche funzionali che nelle SAM (interfacce con un metodo solo).

Immagine che contiene testo

Descrizione generata automaticamente

Il compilatore individua il tipo atteso e, se il tipo atteso corrisponde ad una SAM, implementa l’interfaccia come istruzione del metodo singolo. Java non diventa per questo un linguaggio funzionale.

Da solo le seguenti espressioni non hanno tipo:

Immagine che contiene testo, dispositivo, metro, calibro

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

Questo rappresenta un modo compatto per scrivere l’implementazione di una singola interfaccia:

Immagine che contiene testo

Descrizione generata automaticamente

La combinazione delle *lambda expression*, l’inferenza (quindi deduzione dal contesto delle espressioni), *var* e *diamond operator* (cioè < >) scrivono un codice coinciso e comprensibile.

I costrutti condizionali in Java non sono espressioni, bensì istruzioni; ciò significa che eseguono blocchi di codice separati a seconda del valore della condizione. A tale scopo abbiamo i vari *If Then Else* o gli *switch case*.

Da Java 14 la sintassi di switch è un’espressione, in grado quindi di ritornare un valore. Pur essendo simile esteriormente, vi sono delle differenze. Vi sono 2 sintassi disponibili. Un’espressione dove viene assegnato il valore (distinguendo anche al posto di *break*, la keyword *yield*):

Immagine che contiene testo

Descrizione generata automaticamente

Ogni case per ritornare un valore deve contenere *yield* (non ci sta *fallthrough,* cioè dover forzosamente saltare ad una successiva istruzione *yield*) e l’elenco dei casi deve essere *esaustivo* (quindi deve esserci almeno un caso di default o comunque la lista di tutti i casi).

Anche le istruzioni di iterazione sono ispirate dal C; naturalmente parliamo del caro vecchio *while,* o anche la sintassi *do while*.

Un appunto (inclusione del bit del segno con lo shift, lavorando in bitmask):

Si ha inoltre il *for*, usata anche per un ciclo su un oggetto che implementa l’interfaccia *Iterable* oppure un array. Cioè (considerando che l’ordine di iterazione viene dato dal contenitore):



In questo modo noi stiamo dicendo al compilatore di scorrere tutti gli elementi di una struttura; ciò potrebbe permettere al compilatore di effettuare certe ottimizzazioni.

Seguono keywords come *break*/*continue*, interrompendo il ciclo di iterazione oppure proseguirla. Ogni istruzione può essere preceduta da un’etichetta, per saltare ad un punto preciso (*goto*).

Similmente un’altra istruzione è *return*, concludendo la chiamata al metodo attuale e ritornando il controllo al chiamante.

I metodi possono lanciare eccezioni e devono essere gestite tramite un blocco *try/catch*.

Un esempio molto semplice:

Immagine che contiene testo

Descrizione generata automaticamente

Nel blocco *finally* vi sono le istruzioni sempre eseguite a chiusura del codice, prima di ritornare il controllo al chiamante oppure dopo (per gestire normalmente risorse dell’esecuzione), a differenza del *catch* per esempio, che viene eventualmente eseguito; essi seguono un ordine lessicale, a seconda dell’importanza di ciascuna (es. IOException rispetto a FileNotFoundException, meglio mettere la prima in precedenza alla seconda, in quanto la seconda discende dalla prima).

Un’altra forma dell’istruzione *try* è detta *try-with-resources*, dichiarando una serie di risorse (quindi di oggetti) che devono essere chiusi alla fine dell’esecuzione del programma. Devono implementare l’interfaccia *AutoClosable*, dato che verranno certamente chiuse. In questa forma *catch/finally* sono opzionali. Esempio:

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamentePer lanciare esplicitamente eccezioni, abbiamo l’istruzione *throw*, richiedente un oggetto discendente da *Exception* e lanciato come fosse errore. Esempio:

Esiste anche il blocco vuoto oppure anche istruzioni vuote; alcuni IDE lo segnalano come errore, altri come warning, tuttavia sintatticamente è considerato corretto.

Vi è anche la keyword *assert*, che verifica determinate condizioni al momento dell’esecuzione del programma. Se l’espressione da verificare ritorna *false*, viene lanciato un errore. Queste possono anche essere totalmente ignorate, se richiesto. Esempio:



Il prof consiglia: andarsi a vedere le JavaDocs qualora servano spiegazioni precise di metodi o altro.

I JavaDoc sono appunto organizzati per “moduli”, introduzione di Java 9.

Un *modulo* controlla cosa è accessibile all’esterno dello stesso e in generale viene definito come *insieme di package e di tipi*. Il loro uso (da cui ispira il nome) era la modularizzazione, quindi la separazione del JDK in parti più piccole creando apposite distribuzione con i moduli necessari. Il progetto è riuscito solo in parte, in quanto i moduli servono solo in situazioni apposite e complessivamente si sono diffuse altre soluzioni (container, ecc.).

Parliamo subito della gestione I/O contenuto nel package *java.io,* le cui principali astrazioni sono *File, InputStream* ed *OutputStream*. Usi comuni sono attraverso *Reader* oppure *Writer*, fornendo metodi semplici per lettura/scrittura di file testuali. Esempio:

Immagine che contiene testo

Descrizione generata automaticamente

La libreria standard è organizzata per gerarchia di capacità, promuovendo l’uso della composizione per creare le necessarie catene elaborative. Questa versatilità produce API prolisse/ingombranti, dando spazio a tutti i punti di accesso per i vari casi d’uso.

L’oggetto System (parte di Java.Lang.System) possiede gli stream classici (*System.in, System.out, System.err).* Similmente, Java possiede *java.nio*, aggiungendo nuove astrazioni, gestione asincrona di I/O e miglioramenti di performance.

Altra parte vasta ed importante è la *Collections API,* quindi la API delle collezioni. L’interfaccia *Collection* è la radice della libreria. Essa contiene i metodi più generali (dimensioni, test di contenitore, aggiunta/rimozione), interfaccia specializzata da altre volendo, dato che non ci sono implementazioni dirette. Molti metodi dell’interfaccia sono marcati come “opzionali”, quindi le singole implementazioni ci sono se effettivamente esistenti, altrimenti ritorneranno una *UnsupportedOperationException* se non supportano una certa operazione. Caso tipico: viste non modificabili di altre collezioni, che non permettono la modifica del loro contenuto.

Moltissime collezioni si aspettano un uso coerente del metodo *java.lang.Object#equals()* in quanto non esiste un vero uso del confronto fra oggetti con l’operatore == (si può pensarlo come confronto tra puntatori, ritorna vero solo se puntano alla stessa cosa).

Esempio utile:

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo, dispositivo, metro, pannello di controllo

Descrizione generata automaticamente

Altro esempio con *equals* e anche *hashCode*:

Immagine che contiene testo

Descrizione generata automaticamente

Il seguente caso darà *true* invocando *twoTimes*:

Immagine che contiene testo

Descrizione generata automaticamente

*Immagine che contiene tavolo

Descrizione generata automaticamente*Immagine che contiene tavolo

Descrizione generata automaticamente*Collections* possiede vari metodi di utilità, quindi:

Altre interfacce importanti sono *Iterator*, elencando una collezione un elemento alla volta, oppure *Iterable*, fornendo un *Iterator* per essere attraversata oppure *Spliterator*, che attraversa in sequenza ogni singolo elemento di una collezione oppure di flussi di risorse in parallelo.

Gli elementi di Iterator più importanti sono:

Immagine che contiene testo

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

Alcune note: *remove* è opzionale, *forEachRemaining* accelera il consumo della collezione.

L’interfaccia *List* è un elenco ordinato di elementi, indirizzabili per posizione; sono permessi elementi duplicati. Fornisce un iteratore apposito (*ListIterator*) capace di movimento bidirezionale.

Implementazioni utili:

Immagine che contiene tavolo

Descrizione generata automaticamente

List fornisce anche un metodo *of* per creare rapidamente una lista a partire da un elenco di oggetti.



Un’altra interfaccia che definisce un insieme è *Set*, contenitore di oggetti senza ripetizioni non ordinato. Pessima idea mettere un elemento in un set e cambiarne il significato di *equals*, in quanto si potrebbe non trovarlo più sulla base di una specifica implementazione.

Alcuni tipi di set:

Immagine che contiene tavolo

Descrizione generata automaticamente

Ulteriori sono *SorkedSet*, insieme con un ordine totale e *NavigableSet*, insieme ordinato muovendosi e sfruttando l’ordine, cercando direttamente l’elemento minore/maggiore di un elemento dato ed *EnumSet*, con delle bitmask che sono estremamente efficienti.

***14/03/2022: Lezione 5 - Libreria standard***

Andiamo avanti con le interfacce, in particolare parliamo della *Dequeue*, usata sia come stack che come FIFO (essa è una Double Ended Queue) togliendo sia dall’inizio che dalla fine.

Esistono due implementazioni: *ArrayDequeue,* implementata come array oppure *LinkedList*, che può implementare sia la lista linkata singolarmente che la struttura Dequeue.

Essa possiede due set di metodi differenti, quindi:

* *(add, remove, get)* che può lanciare eccezione
* *(offer, poll, peek)* che può ritornare un valore speciale, scegliendo se usare try/catch oppure usare il valore che viene ritornato in quel momento. *Peek* prende il primo oggetto di una queue senza rimuoverlo, *poll* al contrario lo rimuove sempre dalla cima della queue, *offer* aggiunge un elemento nella queue se possibile farlo senza violare la capacity della stessa

Altra interfaccia utile è la *Map*, che rappresenta un’interfaccia chiave/valore. Gli oggetti usati come chiavi devono essere corretti per *equals/hashCode* correttamente definita. Valgono le stesse cautele di mutamento di stato di una chiave.

L’interfaccia mette a disposizione tre diverse viste sui contenuti:

* un elenco di *Entry*
* l’insieme delle chiavi
* l’elenco dei valori, insieme iterabile perché potrebbero esserci valori ripetuti

Non sono permesse chiavi non uniche. Le implementazioni variano riguardo al permettere valori come *null*. Alcuni esempi di implementazione:

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene tavolo

Descrizione generata automaticamente

Nel caso della *WeakHashMap* usa riferimenti deboli quindi, se un certo oggetto non viene riferito, il garbage collector lo dealloca autonomamente. Uso pratico di questa particolare struttura: la cache, in particolare si pensi agli hit/miss.

Similmente a prima, esistono le *SortedMap* e *NavigableMap* partendo da un ordine totale sulle chiavi.

Similmente ai casi di List/Set, ecc. posso costruire rapidamente mappe con il metodo *of*, lavorando con generici/inferenza di tipo per capire le variabili usate:



Una parte importante dell’aggiornamento di Java 8 è stata l’introduzione del concetto di *Stream*, aggiunto sotto forma di metodo per realizzare le varie implementazioni (*stream())*

Viene considerato sequenza di elementi, decidendo se iterare o meno sugli elementi, con una sequenza di elementi potenzialmente infinita. L’obiettivo è la descrizione dell’elaborazione e la sua ottimizzazione, duale all’obiettivo di una collezione.

Le operazioni vengono composte sequenzialmente, in una pipeline, finché non arrivo ad un’operazione terminale, che produce il risultato. Tutte le istruzioni di stream sono *lazy*, proprio perché vengono eseguite solo se necessarie per un’operazione terminale.

Il codice che implementa la pipeline può quindi prendere decisioni su quali operazioni effettivamente realizzare e quali saltare. Le operazioni intermedie non devono modificare gli elementi dello stream e, nella maggior parte dei casi non hanno uno stato interno.

Essi possono essere costruiti sia da collezioni di partenza sia altri tipi di astrazioni (file, canali di comunicazione, generatori casuali).

Le operazioni intermedie sono divise tra *stateful* e *stateless*, variando a seconda dell’efficienza della pipeline. Descriviamo di seguito le operazioni *stateful*, seguendo poi le *stateless*:

Immagine che contiene tavolo

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene tavolo

Descrizione generata automaticamenteSeguono istruzioni terminali/generatori:

Immagine che contiene tavolo

Descrizione generata automaticamenteImmagine che contiene tavolo

Descrizione generata automaticamente

Nota: *Supplier* è un interfaccia che solitamente implementa un metodo solo: potenzialmente può essere usato per stream o generatore di istanze all’infinito.

In generale quindi sono un’astrazione utile e consente di descrivere il significato dell’elaborazione, invece del metodo, una volta descritto il calcolo fatto; il compito di organizzare tutto il calcolo appartiene proprio allo stream.

L’uso del *null* porta una serie di conseguenze, perché si deve capire come utilizzarlo. Questa ambiguità porta a numerosi idiomi di *programmazione difensiva*, in cui si cerca di mantenere l’esecuzione nonostante possibili problemi (esempio spesso presente, la gestione delle NPE/Nullptr Exception e dei valori nulli, oppure anche asserzioni/precondizioni). Sicuramente è buona programmazione quest’ultima sotto alcune circostanze, ma non in tutte; volendo si usa il concetto di *programmazione offensiva*, che conosce l’errore e cerca di agire controbilanciandolo e prevenendolo.

Diciamo che noi introduciamo un concetto che è una via di mezzo, cioè la classe *Optional*, definita in altri linguaggi come *Option* o *MayBe*, indicando che la classe rende esplicita la rappresentazione di un valore che potrebbe esserci oppure no. In questi casi meglio inizializzare direttamente il valore risolvendo le ambiguità; se ben implementato può evitare le NPE.

Altro problema interessante è la *gestione del tempo*, che comporta varie irregolarità in merito alla sua gestione (es. alcuni hanno usato 62 secondi al posto di 60, oppure la data del 30 giugno in merito all’aggiunta dei secondi bisestili, ecc.).

La prima API temporale di Java ruota attorno a *java.util.Date*, assorbita poi nel package *java.time*.

Seguono inoltre *Instant*, singolo ed astratto istante nel tempo, nonché *LocalDate, LocalTime* e *LocalDateTime*, che rappresentano una data/ora/istante. Segue anche *ZonedDateTime* con l’informazione del fuso orario.

Ora finalmente facciamo degli esempi e parliamo del *TDD (Test Driven Development)*.

Le fasi sono:

* pensare al problema e a come scrivere il test
* scrivere il test
* osservarlo fallire
* si scrive il codice più semplice possibile per farlo passare

Abbiamo un primo esempio con Fibonacci.

Altra idea che viene dai kata del karate, descrive il fatto di cercare ripetutamente di replicare l’approccio di risoluzione del problema piuttosto che concentrarsi sul risultato del problema stesso.

Un altro esempio è il calcolo del punteggio del Bowling, considerando tiri vuoti, spare o strike.

***15/03/2022: Lezione 6 – Esempi svolti: Fibonacci/Bowling***

In questo esempio si ha una classe BowlingGame dove noi implementiamo il codice e dobbiamo produrre un metodo *score()* che calcoli il punteggio della partita.

Abbiamo una serie una serie di classi di test. In questo caso abbiamo diverse variabili da considerare:

* il numero di tiri, 2 per tutta la partita (9 frame), 3 nell’ultimo frame
* se in un frame il giocatore abbatte meno di 10 birilli, ottiene come punteggio il numero di birilli abbattuti (viene indicato con segno “-“ e semplicemente indica che in un frame, si ha un solo tiro, l’altro non si conta)
* lo spare, quindi abbattendo 10 birilli in due tiri si conta come bonus il numero di birilli abbattutti con il tiro seguente
* lo strike, abbattendo 10 birilli con un solo tiro, si conta come bonus il numero di birilli abbattuti nei due tiri seguenti
* nel decimo frame, se si ottiene uno strike/spare nei primi due tiri, si ha un terzo tiro con cui calcolare il bonus per il punto segnato

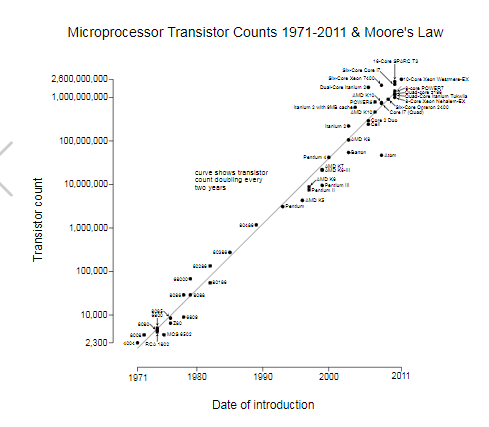
***(Il TicTacToe, se non sbaglio, ha detto che lo vedremo meglio in seguito, abbiamo comunque fatto un accenno a quello. Per completezza, non viene approfondito, ma è presente la lezione).***

***21/03/2022: Lezione 8 - Programmazione concorrente***

Si intende con *programmazione concorrente* quindi la teoria e l’insieme di tecniche utili per la gestione di più processi contemporaneamente sulla stessa macchina.

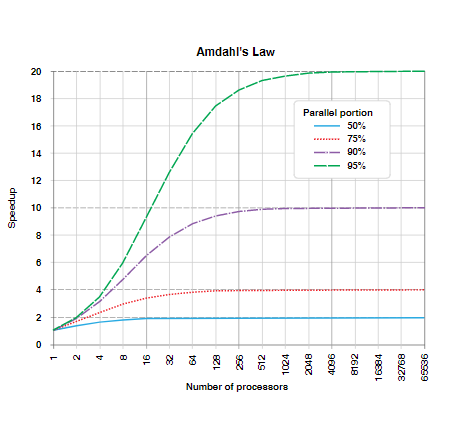
Nelle macchine con una sola CPU essa è diventata collo di bottiglia/bottleneck, partendo infatti dalla macchina di Von Neumann, che però eseguiva una sola istruzione alla volta. Per questo diventa rapidamente molto poco efficiente.

Presto vengono introdotti dei channels nei mainframe IBM, rendendo “intelligenti” le periferiche leggendo in nastri e stampando risultati. Il modello economico affronta il problema del coordinamento facendo in modo esso diventi più efficiente.



La concorrenza diventa quindi problema del sistema operativo, che si trova a gestire molteplici attività contemporaneamente o in rapida successione. Dunque le risorse disponibili vanno distribuite tra le attività.

La CPU è quindi efficace per un tempo maggiore, piuttosto che attendere l’esecuzione delle singole istruzioni. La legge di Moore (1965) fornisce risorse sempre crescenti, raddoppiando il numero di transistor a disposizione. Di fatto in dimensioni così ridotte, gli effetti termici sono molto rilevanti e si cerca di risolvere queste problematiche moltiplicando i core, predicendo dove i salti dei processori possano avvenire (caso Spectre/Meltdown).



Tutta questa gestione diventa complicata ed insicura. La legge di Amhdal (1965) individua i limiti matematici all’efficienza della parallelizzazione.

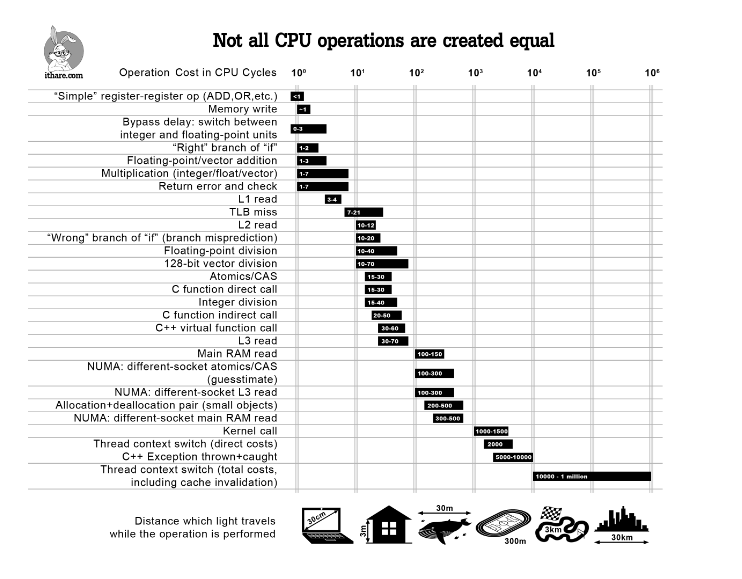
Il parallelismo quindi usa più risorse per ottenere i risultati pià velocemente; nella realtà teniamo solo quelli più utili.

Parallelizzare le attività non è sempre possibile e il lavoro deve essere valutato in base alla parallelizzazione ottenibile e l’accelerazione ricavabile.

Concettualmente in UNIX descriviamo un *processo* come un programma in esecuzione e tutte le sue risorse assegnate (memoria, canali I/O, interrupt/segnali, stato CPU).

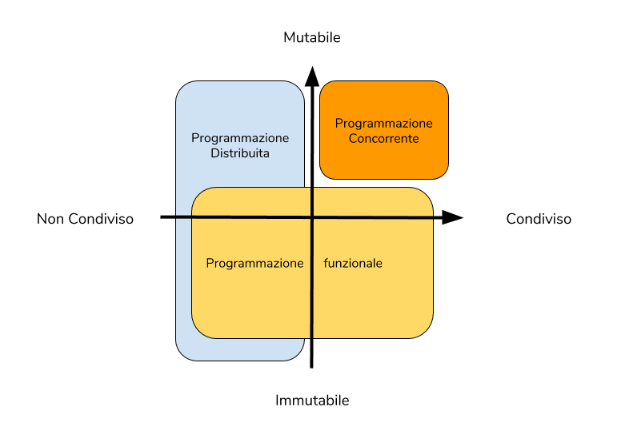
I processi non condividono risorse di norma, interagendo come entità separate, garantendo un uso efficace e paritario delle risorse. Per la simmetria/velocità della CPU nel resto del computer è quindi uno spreco. Il contesto della CPU deve essere salvato e messo da parte per evitare sprechi.

Segue un confronto logaritmico tra le operazioni:



Per gestire quindi più linee di esecuzione all’interno dello stesso processo nasce il concetto di *thread,* che sono linee di esecuzioni concorrenti in uno stesso processo, risparmiando le risorse in caso di cambio di contesto. Di fatto però in carico all’applicazione il problema della gestione ed accesso contemporaneo sarà in mano al sistema operativo.

In questo modo è responsabilità esplicita dell’applicazione questa gestione, come si vede qui, con nodi che non condividono risorse fisicamente separati:



Usiamo quindi una serie di tecniche e strumenti per navigare in questa situazione. La *programmazione distribuita* implica la comunicazione fra entità che non possono avere stato condiviso, mentre la *programmazione funzionale* tratta principalmente dati immutabili, con stato che può essere condiviso.

La *programmazione concorrente*, quindi, unisce le due cose.

La *programmazione ad oggetti* riguarda tutti e quattro gli stati sopra disegnati.

Parliamo poi del *non determinismo*, sapendo che un’esecuzione concorrente è inerentemente non deterministica e potrebbe morire di fame (*starvation*), competizione delle stesse risorse (*race conditions*), nessuno dei due può proseguire perché uno dei thread attende la risorsa dell’altro.

Le linee di esecuzione scambiano continuativamente le risorse dando “l’impressione” di stare avanzando.

Nel caso di deadlock tuttavia, si hanno le *condizioni di Coffman*:

* mutua esclusione
* attesa della risorsa e trattenimento della stessa
* no preemption
* attesa circolare

Queste sono necessarie perché avvenga un deadlock. Rimuovere la mutua esclusione può quindi non essere fattibile per certe risorse, richiedendo algoritmi specific detti *lock-free* o *wait-free.*

Rimuovere l’attesa, quindi, può portare a situazioni di starvation o attese indefinite.

Risolvere il problema richiede un sistema transazionale per ottenere più risorse.

Introdurre la pre-emption può essere costoso ma anche impossibile; con l’uso degli algoritmi detti può essere introdotto un controllo cosiddetto *optimistic concurrency control*.

Rimuovere la *circolarità* richiede di imporre un ordinamento alle risorse; ciò crea difficoltà, perché la linea di esecuzione è cosciente dell’esistenza delle altre e non sempre è facile da individuare/creare.

Seguono le tipologie di concorrenza:

Immagine che contiene tavolo

Descrizione generata automaticamente

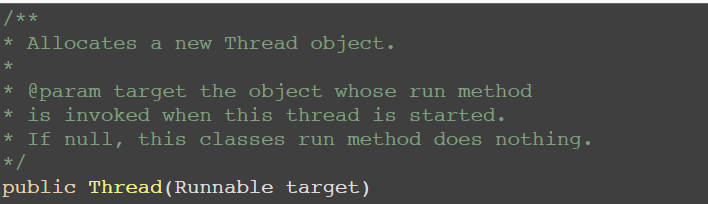
Nella concorrenza *collaborativa*, i programmi cedono il controllo ad intervalli regolari. Questo modello risulta essere rilevante in vari ambiti (embedded, very high performance, ecc.)

In quella di tipo *pre-emptive*, il sistema operativo è in grado di interrompere l’esecuzione di un programma e sottrargli il controllo delle risorse per affidarle al programma seguente, avendo dei programmi privilegiati.

Nella concorrenza real-time il sistema operativo deve garantire performance precise e numericamente specificate, per esempio gli interrupt entro un certo numero di ms.

Ulteriore concorrenza è la *event driven/async*, dove i programmi dichiarano le operazioni che vanno eseguite e lascia la decisione all’ambiente di esecuzione, decidendo quando e come assegnare le risorse. Quest’ultima sta diventando popolare nell’organizzazione delle applicazioni.

Parliamo quindi di *thread* in Java, usando un’interfaccia Runnable:

**

Usando il metodo *start()* la linea di esecuzione procede immediatamente avviando l’attività del thread, avviando un nuovo percorso di esecuzione nella JVM, condividendo lo stesso heap.

Immagine che contiene testo

Descrizione generata automaticamente

Altro metodo che spesso useremo sarà *sleep()*, che mette in pausa il thread corrente per un certo periodo di tempo, specificando i millisecondi:

Immagine che contiene testo

Descrizione generata automaticamente

Abbiamp poi l’interfaccia *Runnable*, che modella un compito da cui non ci si aspetta un risultato.

Immagine che contiene testo

Descrizione generata automaticamente

Un esempio possibile è l’interfaccia *ThreadSupplier*, fornitore di thread generando il loro tempo d’attesa:

Immagine che contiene testo

Descrizione generata automaticamente

Seguono altri esempi con:

* uso di un singolo thread

Immagine che contiene testo

Descrizione generata automaticamente

* uso di molteplici thread

Immagine che contiene testo

Descrizione generata automaticamente

***22/03/2022: Lezione 9 - Threads***

Cominciamo ad esaminare l’uso dei thread esaminandone avvio e stato; diamo la forma completa di thread specificando anche gli argomenti di solito opzionali:

Immagine che contiene testo

Descrizione generata automaticamente

Similmente, come visto ieri, *start()* lancia la linea di esecuzione parallela, o metodi come *getName()*, prendendo il nome e rendendo più facile la gestione dei log.

Il metodo che qualifica se è vivo/ancora attivo il thread è il booleano *isAlive()*. Poi il metodo *run()* esegue l’oggetto Runnable nella stessa istanza di esecuzione in quel momento.

Altro metodo statico del thread in cui ci troviamo in quel momento si ha *currentThread()*. Il thread principale è il *main()*. Metodo di temporizzazione approssimato, dato dall’orologio dell’hardware, è lo *sleep()*.

Immagine che contiene testo

Descrizione generata automaticamente

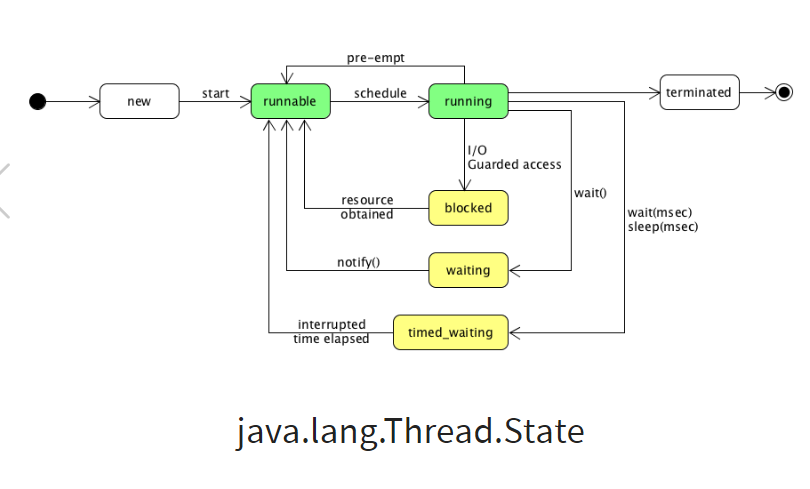
Costruiamo un thread che osserva lo stato di un altro thread:

Riceviamo come target un thread che dura 800 ms, avviando poi separatamente l’osservatore ed il bersaglio (sintatticamente, l’ordine di lancio conta). Come detto, si ha il non determinismo nell’esecuzione:

Immagine che contiene testo

Descrizione generata automaticamente

Oltre ai classici stati (vivo/terminato) i thread hanno una serie di stati (come visibile dal seguente scherma che segue il tipo enum):



La serie di stati quindi è:

* *new*, creazione del thread;
* *runnable*, a seguito del richiamo dello *start()*, mandando poi in esecuzione (volendo) il metodo *run();*
* *running*, una volta che ha ricevuto la risorsa e sta eseguendo istruzioni. Può uscire da questo stato se la CPU gli viene sottratta, salvando il suo stato di contesto;
* *blocked,* se ha richiesto accesso ad una risorsa monitorata e aspetta la disponibilità dei dati;
* *waiting,* il thread aspetta la disponibilità di alcuni dati, prendendo la risorsa di un *lock()*. Il metodo *notify()* seleziona successivi thread in attesa;
* *timed\_waiting,* attesa con un massimale di tempo dato dallo sleep, scaduta la quale torna Runnable
* *terminated*, mettendo il thread a disposizione della garbage collection

Quando tutti i thread sono stati terminati, la JVM ritorna il controllo alla shell richiamata.

Immagine che contiene testo

Descrizione generata automaticamenteParliamo quindi di *interruzioni* (fatte con il metodo *interrupt()*), terminando sia esplicitamente sia se l’eccezione non venisse gestita.

Vediamo quindi l’altro esempio, controllando se il thread è vivo e lo interrompe:

Passiamo quindi il target di esecuzione nel main:

Immagine che contiene testo

Descrizione generata automaticamente

Su un thread possiamo chiamare un gestore delle eccezioni:

Immagine che contiene testo

Descrizione generata automaticamente

Si possono anche rilanciare le eccezioni, aggirando l’obbligo sintattico del blocco precedente:

Immagine che contiene testo

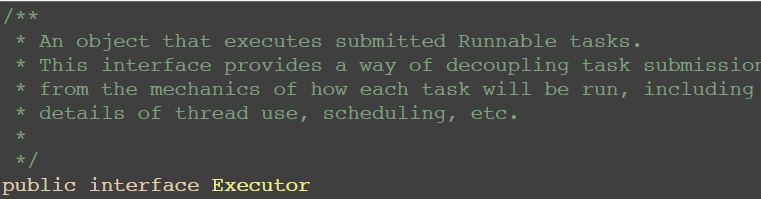
Descrizione generata automaticamente

Successivamente si imposta con una lambda la gestione dei thread che non hanno gestito le eccezioni:

Immagine che contiene testo

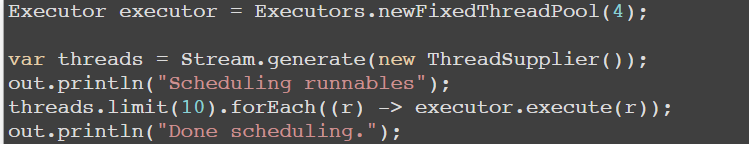
Descrizione generata automaticamente

Già maneggiare i singoli thread richiede un po’ di sintassi e l’amministrazione si complica al crescere del numero degli oggetti. Diamo quindi il controllo all’interfaccia *Executor,* prendendosi il carico su nostra indicazione come gestire lo scheduling:



Il metodo più usato è proprio *execute*(), con argomento un Runnable.

Un esempio di implementazione può essere (fornendo un pool prestabilito di thread):

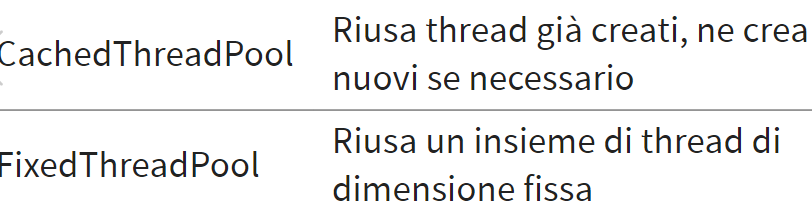


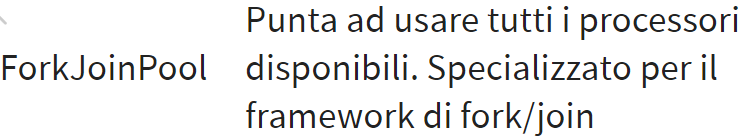
Similmente posso fornire l’esecuzione di un singolo thread, con l’esecutore di un unico thread:

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamenteDiamo quindi alcuni esempi di esecutori:



Quest’ultimo esecutore (fork) è ottimizzato per gestire attività che ne guidano e/o richiamano altre (es. prog. dinamica/ricorsiva) accelerando se possibile l’hardware.

Finora abbiamo lavorato con attività che non danno effetti collaterali (oggetti *Runnable*); ora parliamo dell’interfaccia *Callable*, che definiscono il risultato con un solo metodo, volendo lanciando un’eccezione:

Immagine che contiene testo

Descrizione generata automaticamente

Un Executor prende Runnable e non Callable; usiamo quindi *ExecutorService*, parametrizzato in base al tipo di Callable:

Immagine che contiene testo

Descrizione generata automaticamente

Il calcolo che prima o poi ritornerà un valore dà un oggetto F*uture,* che rappresenta un’interfaccia e anche qui parametrizzato in base al tipo di ritorno*:*

*Immagine che contiene testo

Descrizione generata automaticamente*

Il metodo *get()* ritorna il risultato calcolabile, altrimenti blocca il thread su cui si lavora, andando in stato di waiting. Quando il lavoro è completato, si usa il metodo *isDone()*.

Diamo un esempio di esecutore, implementando una serie di attività e verificando quante sono state completate, stampando la situazione corrente:

Immagine che contiene testo

Descrizione generata automaticamente

Ispezioniamo poi la lista di *future*, dando o meno il risultato. Si usa *factorialBuilder*, calcolando una serie di attività sulla base di Fibonacci in base al numero di calcolare. Di fatto esegue il fattoriale questa e poi vediamo i task completati:

Immagine che contiene testo

Descrizione generata automaticamente