28/02/2022: 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

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. Essa 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 mptivi di sicurezza.

Dichiarando una classe senza modificatori di visibilità, la classe è visibile solo allo stesso 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. ù

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 creati ed usati 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. Essa si comporta in modo analogo agli altri casi. Esistono anche le *static nested classes*, normale classe di package con un nome prefissato da quello della classe ospite.

Altra classe interessante 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, sonp 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 ma sono segnale di modello dati complesso; da usare con cautela e queste sono comunque statiche.

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 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.

Java può ereditare esclusivamente da una sola superclasse, ereditandone codice e parte dello stato. Accede ai membri pubblici, protetti e ai package, non ai membri privati naturalmente. 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’esecuzzione 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 metodi diversi tra di loro, *equals*, riconoscendo l’uguaglianza tra tipi di oggetti, *toString*, dando l’emissione di un oggetto visto come stringa quando viene implementato a console.