28/02/2022: Introduzione

Si parla di *paradigma di programmazione* o paradigmi più in generale come una tecnica o un insieme per affrontare una serie 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; contemporaneamente si parla di parallelismo con più linee di esecuzione su parti di problema differenti. Altro paradigma è l’esecuzione in rete, con un calcolo distribuito su nodi che condividono risorse e mezzi di comunicazione, asincrona e parallela comunque.

Analogamente, la distribuzione prevede nodi indipendenti e che condividono la stessa visione sullo stato compelssivo del sistema.

Concludendo, si parla anche di reattività, come base, comunicazione asincrona e flessibile, con bassa latenza alla risposta.

SI usa Java, compilato attraverso il bytecode, linguaggio macchina intermedio della JVM. Esso è interpretato e volendo già compilato durante l’esecuzione (aka JIT) oppure prima della stessa esecuzione. La filosofia è WORA (Write Once Run Anywhere). Il linking della compilazione è dinamico, dato che guarda i nomi a tempo di compilazione. Il codice stesso viene caricato non solo da filesystem locale ma anche da altre fonte, specififcate nel percorso. Esso viene gestito da una gerarchia di ClassLoader, dividendo il caricamento delle classi di libreria da quelle di estensione.

Il comando *javac* invoca il compilatore mentre il comando *jar* getisce le distribuzioni di codice.

Varie distribuzioni ed IDE esistono; un esempio è Maven, di Apache ed adattabile ad una serie di situazioni. Successivamente viene sviluppata Gradle, con una struttura non più fissa e non più in XML ma con Groovy.

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 bloccho 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 line aDNS 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 categoria di variabili, tra cui di istanza, quindi ogni oggetto ha la propria copia e fa parte dello stato e statiche, di cui ce ne sta una copia sola. Le variabilki statiche vengono allocate ed inizializzate dal ClassLoader e preparata per l’uso.

Oltre alle varaibili pubbliche, come sappiamo, possono esserci le variabili *protected*, lette e scritte da classi che estendono la classe, *private*, solo dal codice della classe.

Altri modificatori indicati sulle variabili sono *final*, variabile che non può più essere modificata dopo la costruzione, *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 maiuscola 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 titpo 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. 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 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 anonimi. Non si sa quando esattamente ciò avvenga, quindi quando la JVM decide di attivarlo e non è sotto il controllo della stessa esecuzione. Possono essere *static*, eseguiti quindi lessicalmente al caricamento della classe stessa. L’uso dei blocchi statici non è comune, comunque non inconsueto. Attenzione al tempo di 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.

Essi valgono, ancora di più, per gli inizializzatori statici.

In generale, attenzione all’ordine di inizializzazione delle classi o delle istanze, dato che si possono ottenere errori inattesi nei momenti meno opportuni; scriverne meno possibile, ma meglio non scrivere proprio inizializzatori, ma solo costruttori, dice il prof.

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 keyowrd *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 *hasCìhCode*, riconoscendo metodi diversi, *equals*, riconoscendolo l’uguaglianza tra tipi di oggetti, *toString*, dando l’emissione di un oggetto interpreatato come stringa quando viene implementato a console.