ISPW – CONCETTI CHIAVE DELLA PARTE JAVA

# Lezione5 – Java: linguaggio e piattaforma

**Il modificatore final**:  il modificatore final può essere utilizzato, per definire una variabile che di fatto diverrà una **costante** per l'istanza della classe. Final può essere applicato anche ai metodi di una determinata classe e un metodo definito come final (finale non modificabile) implica che se eredito la classe che contiene il metodo su questo non potrà essere eseguito l'override. Anche per quanto riguarda una classe questa può essere definita come final e di conseguenza una classe definita in questo modo non potrà essere ereditata. Ciò è logicamente comprensibile in quanto se dichiaro una classe con il modificatore final intendo una classe finale ovvero una classe completa che non necessita di specializzazioni o estensioni e dunque è più che logico che non sia possibile ereditarla.

**Il modificatore static**: il modificatore static può essere applicato a metodi e a variabili di una classe. L'effetto di dichiarare un metodo static è quello di rendere il metodo di fatto comune a tutte le istanze della classe. Dunque se dichiariamo un metodo static potremo evitare di istanziare la classe che lo contiene e richiamare il metodo statico con la sintassi: **nomeClasse.nomeMetodo()**. Quando si usa il modificatore static per una variabile, di fatto si rende l'attributo della classe comune a tutte le istanze della classe. Dunque all'atto pratico se dichiariamo una variabile di istanza utilizzando il costrutto **public static** avremo come risultato il definire una **variabile globale** accessibile a tutte le istanze di una classe. Le variabili globali sono un arma a doppio taglio in quanto se il fatto di poter condividere una variabile con lo stesso valore per tutte le istanze della classe è sicuramente un vantaggio, si ha che nel caso in cui, per errore, un'istanza modifichi la variabile globale, tutte le altre istanze della classe troveranno il valore modificato compromettendo la robustezza dell'intera applicazione.

Nell’ereditarietà assumono importanza i modificatori Public, Private e Protected:

**Il modificatore di accesso public**: permette di dichiarare dati e metodi visibili e quindi utilizzabili dalla sottoclasse senza limitazioni.

**Il modificatore di accesso private**: nasconde completamente dati e metodi dichiarati tali.

**Il modificatore di accesso protected**: permette l’accesso a metodi e dati a tutte le sottoclassi, all’interno o meno dello stesso package, e alle sole classi dello stesso package se si tratta di una variabile reference\*.

\*Tutte le istanze di classi create in Java sono definite mediante l’utilizzo di *reference*. Un reference è, dunque, una variabile speciale che tiene traccia di istanze di tipi non primitivi. I reference possono tenere traccia soltanto di oggetti di tipo compatibile: ovvero un reference ad un oggetto di tipo MyClass non potrà tenere traccia di oggetti di diverso tipo.

\* class package subclass world  
\* public si si si si  
\* protected si si si no  
\* default si si no no  
\* private si no no no  
\*

# Lezione6 – I linguaggi di modellazione && UML

**UML** (Unified Modeling Language) è un linguaggio visuale per la specifica, la costruzione e la documentazione degli elaborati di un sistema software.

Si tratta di un vero e proprio **linguaggio di modellazione**, che riunisce costrutti e concetti di approcci già esistenti, come la programmazione object-oriented, le macchine a stati e la *objectory vision* (visione oggettiva). Inoltre: - Prevede meccanismi di estensibilità (è possibile aggiungervi nuove librerie). - Astrae dai linguaggi di programmazione (è un linguaggio più ad alto livello rispetto a qualsiasi linguaggio di programmazione). - Rappresenta i concetti (molto spesso tramite una notazione visuale).

UML mette a disposizioni tre tipi di diagrammi principali per descrivere:

struttura statica del sistema: **Structure diagrams** (e.g. class diagrams, object diagrams, component diagrams, package diagrams)

comportamento del sistema: **Behavior diagrams** (e.g. state diagrams, activity diagrams, use case diagrams)

interazione tra gli oggetti: **Interaction diagrams1** (e.g. sequence diagrams, communication diagrams, timing diagrams) [sono un sottotipo dei behavior diagrams]

Un sistema software espone molteplici aspetti che devono essere trattati appositamente:

- **Aspetti funzionali** (legati ai requisiti utente, ai requisiti implementativi e alle leggi)

- **Aspetti extra-funzionali** (legati agli aspetti di qualità del servizio offerto, che sono relativi alla soddisfazione del cliente, e ai vincoli sul tempo e sulle risorse a disposizione)

- **Aspetti organizzativi** (legati agli aspetti tecnologici e non)

UML consente di scomporre un processo di sviluppo software secondo **cinque prospetti** (o viste), ciascuno dei quali enfatizza la descrizione di aspetti specifici del sistema in diverse fasi dello sviluppo.

1) **Use case view**: si occupa dell’analisi / modellazione dei requisiti utente e di come il sistema viene percepito dall’utente (progettazione black-box); descrive quali funzionalità devono essere progettate (e non come). Nella pratica, consiste nell’individuare tutti gli attori, i casi d’uso e le loro relazioni. Target: clienti, progettisti, sviluppatori, tester.

2) **Logical view**: si occupa della progettazione della struttura del sistema (progettazione white-box); descrive come devono essere progettate e realizzate le funzionalità del sistema. Target: progettisti, sviluppatori.

3) **Implementation view**: si occupa di strutturare i moduli implementativi organizzando il codice del sistema in moduli. Target: progettisti, sviluppatori, tester.

4) **Process view**: comprende modelli che descrivono la dinamica del sistema, specificando i processi da eseguire e le entità che eseguono tali processi; si usa per un utilizzo efficace delle risorse, per stabilire l’esecuzione parallela degli oggetti e per gestire eventi asincroni (esterni al sistema). Target: sviluppatori, integratori di sistema.

5) **Deployment view**: si occupa di come e dove devono avvenire le installazioni dei sistemi software. Comprende modelli che descrivono la topologia e l’organizzazione delle macchine fisiche (e.g. computer, dispositivi mobili, connessioni fisiche tra i nodi) e modelli che descrivono come le parti del sistema software sono mappate sull’architettura fisica. Target: sviluppatori, integratori di sistema, tester.

# Lezione9 – Classi, oggetti e information hiding

**Oggetto**: colloquialmente il termine viene spesso utilizzato (in modo ambivalente) per indicare una classe o una sua istanza. Tuttavia, sarebbe più opportuno usarlo solo per queste ultime.

**Classe**: modella una famiglia di entità del dominio di applicazione o un elemento che non fa parte del nel dominio ma che è introdotto durante il processo di sviluppo. In entrambi i casi include la definizione delle proprietà(attributi) e del comportamento(operazioni). Una classe raggruppa un insieme coeso di entità (insieme coeso: tutti gli elementi modellano aspetti utili al raggiungimento dello stesso fine/funzionalità). Viceversa, la non-coesione è sinonimo di accoppiamento e consiste nel raccogliere all’interno della medesima classe elementi o funzionalità semanticamente molto distanti tra loro (da evitare nella progettazione / programmazione object-oriented). Un oggetto si relaziona ad una classe allo stesso modo di come un dato si relaziona ad un tipo (nei ling. di progr.). Tutti gli oggetti(istanze) afferenti una classe condividono lo stesso insieme di comportamenti, proprietà e relazioni; in generale differiscono nei valori delle proprietà.

**Istanza**: A una classe, che cattura in modo astratto un concetto, possono afferire delle **istanze** (=oggetti), che rappresentano gli elementi effettivamente appartenenti al mondo reale e, quindi, sono manifestazioni concrete di un’astrazione (ad esempio, la penna di Marco può essere vista come un’istanza della classe “penna”).

**Dinamica di un sistema**

- **Approccio procedurale** (tipico del C): esiste un *main* che coordina il comportamento del sistema in base allo scheletro del diagramma di flusso codificato nelle istruzioni. Il *main* gestisce direttamente i dati del sistema, occupandosi esplicitamente della loro transizione tra le varie funzioni (o procedure o subroutine, a seconda dello specifico linguaggio di programmazione).

- **Approccio object oriented** (tipico del Java): ogni oggetto ha i suoi “dati” (ovvero i valori per gli attributi di istanza). Gli attributi di istanza sono allocati e inizializzati alla creazione dell’oggetto e vengono mantenuti dall’istanza anche dopo l’esecuzione dei metodi (a differenza di quanto accade per le tradizionali invocazioni a funzione nell’approccio procedurale): tutti i dati sono mantenuti fino alla distruzione (esplicita o implicita) dell’oggetto. Qui il *main* non fa più da “direttore d’orchestra”: il comportamento del sistema è generato dall’interazione di più istanze tra loro. In particolare, l’interazione comporta lo scambio di **messaggi** tra oggetti diversi e può produrre transizioni di stato nelle entità coinvolte. A valle di tutto questo, con l’approccio object oriented è possibile costruire un programma con tante classi *main* e si può decidere come far funzionare il programma in base alla classe da cui deve partire l’esecuzione; è inoltre possibile avere classi senza alcun *main*.

**Operazione vs metodo vs messaggio**

- **Operazione**: è una funzionalità che può essere eseguita su oggetti di una determinata classe. Ha semplicemente una segnatura (=firma), che specifica il nome dell’operazione e il tipo degli eventuali parametri passati in input. Può specificare anche il tipo di ritorno. In altre parole, è la specifica di un “*segnatura*” (prototipo) che un oggetto mette a disposizione di altri oggetti; essa rappresenta il tipo di funzionalità che una classe può usare nei confronti di un’altra classe.

- **Metodo**: è l’implementazione vera e propria di un’operazione messa a disposizione da una classe e, quindi, è un sottoprogramma associato in modo esclusivo a tale classe. Oltre alla segnatura e al tipo di ritorno, è caratterizzato anche da un corpo, che contiene una o più sequenze (o blocchi) di istruzioni scritte per eseguire una determinata azione sulla base dei parametri passati in input. Inoltre, è in grado di restituire al chiamante un valore di ritorno (=output) dello stesso tipo di quello dichiarato inizialmente insieme al nome e ai parametri. Nei linguaggi che dispongono di un meccanismo di gestione delle eccezioni (come Java), il blocco del metodo può terminare sollevando un’eccezione nel caso si verifichi una situazione anomala che impedisce il corretto completamento delle istruzioni. Il concetto di metodo, così come quello di operazione, è “statico”, cioè è definito nel momento in cui si sta programmando (a design-time) ma non è detto che venga attivato a run-time.

- **Messaggio**: è la richiesta a run-time dell’invocazione di un metodo fornito da un’istanza B da parte di un’istanza A (in tal caso si dice che “A invia un messaggio a B”). In quanto tale, potrebbe fallire.

**Class diagram**

È uno dei tipi di diagrammi che possono comparire in un modello UML e ritrae la struttura statica del sistema (infatti comprende elementi definiti a design-time). Ha una rappresentazione logica a grafo, in cui i nodi raffigurano **classi** e **interfacce**, mentre gli archi raffigurano **relazioni**. Una classe all’interno del class diagram è caratterizzata da un nome, da degli attributi e dalle operazioni che possono essere eseguite sugli attributi. È in pratica lo stesso concetto che in O.O. (object orientation): semplificando, rappresenta un tipo di dato non primitivo. Talvolta, può essere usata per raggruppare elementi e contenere package o sottosistemi. D’altra parte, le relazioni corrispondono alla definizione di possibili interazioni tra le classi di un modello e, in particolare, di legami (link) che possono sussistere tra gli oggetti di tali classi. Possono essere corredate da un insieme di informazioni aggiuntive, come il ruolo svolto da ogni classe o la molteplicità (che indica il numero di oggetti delle due classi che possono essere coinvolti in un collegamento). In particolare, una relazione tra una classe A e una classe B implica che A è a conoscenza di B e, in qualche modo, può interagirvi, al fine di fornire, in modo cooperativo, un comportamento complesso. Perciò, l’assenza di relazioni in una determinata classe comporta l’isolamento di tale classe, la quale si ritrova impossibilitata a interagire con le altre classi. Esistono più tipi diversi di relazione, ciascuno dei quali definisce il modo di interazione tra le due classi coinvolte. Questi tipi di relazione sono:

- Aggregazione

- Associazione

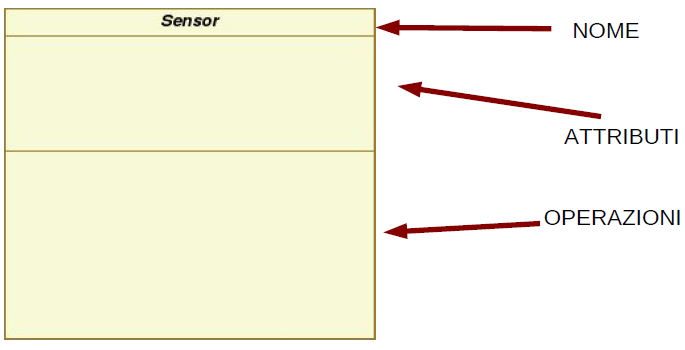
- Composizione

- Dipendenza

- Generalizzazione

- Realizzazione

In UML, una classe è strutturata da tre comparti: **nome**, **attributi** e **operazioni**.



**Comparto nome:** Definisce il nome di un’entità e consiste in una stringa di testo che, per convenzione, ha la lettera iniziale maiuscola. Definizione completa:

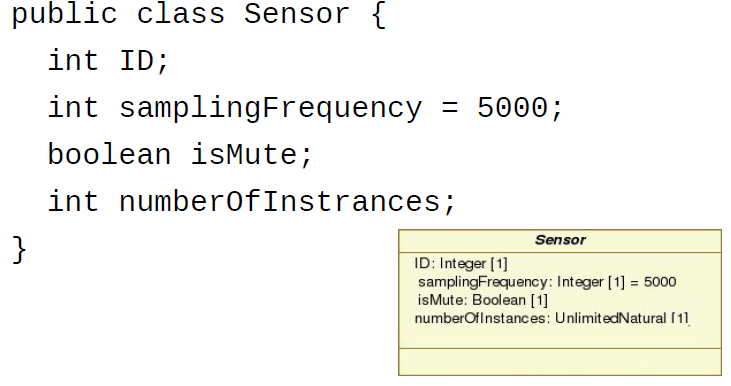
- Java: package + “.” + nome\_classe

- UML: prefisso + “::” + nome\_classe

(notare che se il nome della classe in UML è in *corsivo* allora in Java è richiesto il modificatore “abstract” e viceversa)

**Comparto attributi:** Modella le proprietà di una classe: ogni attributo descrive un insieme di valori che la proprietà può avere quando vengono istanziati oggetti di quella determinata classe. Infatti, le proprietà sono condivise tra tutti gli oggetti appartenenti a una particolare classe. In generale, sono i valori a non essere condivisi tra le istanze.

NB: Tra i tipi di dato degli attributi in Java e i tipi di dato degli attributi in UML non c’è sempre una corrispondenza univoca o esatta. Per esempio, in UML è possibile definire un attributo di tipo *UnlimitedNatural* che, tuttavia, non esiste in Java e, quindi, in fase di programmazione, può essere tradotto in più modi possibili, anche in base alle scelte dell’analista o del progettista: una possibilità è considerare l’*UnlimitedNatural* come un semplice *integer* e imporre al relativo attributo la condizione per cui deve obbligatoriamente assumere valori non negativi; un’altra possibilità è creare una nuova classe Java *UnlimitedNatural* in grado di rappresentare tutti e soli i numeri naturali. In ogni caso, è buona norma diminuire il più possibile il gap che si potrebbe creare tra la fase di progettazione e la fase di programmazione.



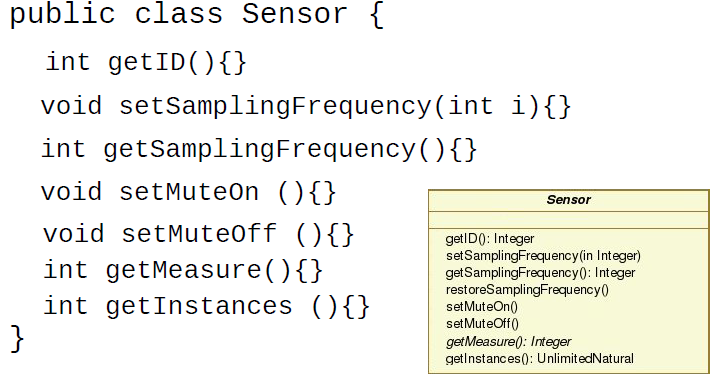
**Comparto operazioni:** specifica i servizi che la classe offre, ovvero che cosa essa può fare, non come. Infatti, in UML non vengono mai specificati i metodi delle varie operazioni. Notiamo che la totale assenza di relazioni per una determinata classe è possibile causa della mancanza di operazioni che agiscono sullo stato di tale classe. Le operazioni manipolano lo stato degli oggetti, ovvero il valore degli attributi di una particolare classe. Hanno una segnatura, che consiste di:

- Un **tipo** (che in realtà appartiene alla segnatura solo in UML e in alcuni linguaggi di programmazione, ma non in Java)

- Un **nome**

- Una **lista di parametri**

NB: Come già accennato, in Java il tipo di ritorno non appartiene alla segnatura di un’operazione o di un metodo: benché sia sempre da specificare, non è in grado da solo di distinguere un’operazione dalle altre.



**Costruttore**: è un’operazione speciale che serve a creare nuove istanze delle classi. La chiamata è effettuata automaticamente all’atto della creazione di un nuovo oggetto di una classe e, nella maggior parte dei linguaggi, non è possibile effettuare un’invocazione manualmente in un secondo tempo. Il costruttore ha un ambito di classe e non di istanza e, infatti, pre-esiste agli oggetti. È utile anche per inizializzare lo stato delle nuove istanze e definire un contesto di esecuzione.

Generalmente, una classe può avere più costruttori, i quali rappresentano più modi differenti di creare le istanze e devono avere tutti lo stesso nome, ovvero quello della classe in cui sono definiti; perciò, si distinguono l’uno dall’altro esclusivamente per il numero e l’ordine dei loro parametri. Per giunta, non vogliono che sia indicato esplicitamente un tipo di ritorno, sia perché è implicitamente dato dal nome dei costruttori stessi, sia perché il tipo di ritorno è di default l’oggetto stesso. Tra i possibili tipi di costruttore ricordiamo quello semplice (che ha uno o più parametri qualsiasi), il **default constructor** (che non prevede alcun parametro) e il **copy constructor** (che riceve come parametro un’altra istanza della medesima classe e ne copia lo stato sulla nuova istanza creata). In genere il costruttore rappresenta un comportamento puramente implementativo, per cui non viene mai esplicitato nei class diagram e viene considerato direttamente in fase di programmazione.

**Distruttore**: ha il compito di deallocare lo spazio occupato da una specifica istanza. In alcuni linguaggi di programmazione object-oriented (come Java) non è previsto che il distruttore venga invocato dal programmatore: questi linguaggi vengono detti *garbage collected* perché mettono a disposizione il cosiddetto **garbage collector**, che è un thread a bassa priorità della Java Virtual Machine (JVM) e lavora nell’heap per verificare se esistono istanze isolate; se sì, dealloca queste istanze tramite una chiamata implicita del distruttore della classe di interesse. In effetti, le istanze isolate sono inutili e sprecano solo memoria, dato che non sono in alcun modo raggiungibili o referenziabili poiché Java e, più in generale, i linguaggi *garbage collected* non prevedono l’algebra dei puntatori. Neanche il distruttore viene specificato all’interno dei class diagram e nelle fasi di analisi e progettazione in generale.

**L’istanza speciale *this***: indica il riferimento all’istanza corrente ed è implicitamente definita come attributo (*NomeClasse this*; ). È per lo più utilizzata nei seguenti ambiti:

- All’interno di un costruttore per invocarne un altro della medesima classe (e.g. *this(“ciao”)* invoca il costruttore di sé stesso che accetta una stringa come unico parametro).

- All’interno di metodi e/o costruttori per disambiguare i riferimenti agli attributi e ai metodi della specifica istanza.

**Attributi e operazioni di classe**:

- **Attributo di classe**: è valido indipendentemente dall’esistenza di istanze della classe. Il suo valore è condiviso tra tutte le eventuali istanze, le quali contengono un campo corrispondente a un puntatore all’unica area di memoria (nell’heap) in cui si trova l’attributo di classe.

- **Operazione di classe**: non richiede l’esistenza o l’impiego di un’istanza della classe per poter essere chiamata: l’invocazione può infatti avvenire direttamente mediante il nome della classe stessa (NomeClasse.nomeMetodo()).

Sia gli attributi che le operazioni di classe si indicano:

- Col modificatore ***static*** in Java

- Con la sottolineatura in UML

**Incapsulamento e information hiding**: nei linguaggi di programmazione object-oriented, il termine “incapsulamento” può essere usato per riferirsi a due concetti o alla combinazione dei due:

- Un meccanismo del linguaggio di programmazione atto a limitare l’accesso diretto agli elementi dell’oggetto

- Un costrutto del linguaggio di programmazione che favorisce l’integrazione dei metodi all’interno della classe

I termini “incapsulamento” e “information hiding” vengono spesso usati come sinonimi, anche se tra loro esiste una sottile differenza concettuale: l’information hiding è il principio teorico su cui si basa la tecnica dell’incapsulamento. Secondo il concetto di information hiding, i dettagli implementativi di una classe sono nascosti all’utente. Pertanto, una parte di programma può nascondere informazioni incapsulandole in un costrutto dotato di interfaccia, permettendo appunto l’information hiding. Tuttavia, l’incapsulamento non è garanzia di information hiding, poiché potrebbe, se mal utilizzato o per motivi particolari, non nascondere i dettagli implementativi. L’incapsulamento riduce il costo da pagare per correggere gli errori in fase di sviluppo di un programma. Questo risultato viene ottenuto strutturando l’intero progetto, e i moduli che lo compongono, in modo che un’errata decisione presa nell’implementazione di un singolo modulo non si ripercuota sull’intero progetto e possa essere corretta modificando soltanto quel modulo. Si potrà così evitare di dover modificare anche i moduli *client*, che interagiranno con quello incapsulato soltanto attraverso interfacce.

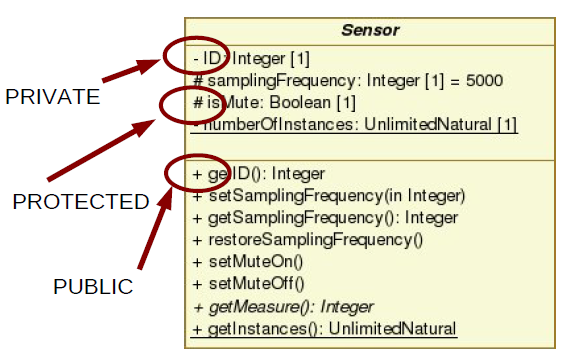
Un altro possibile motivo per ricorrere all’incapsulamento è la necessità di applicare dei controlli sull’accesso e/o sulla manipolazione delle proprietà delle istanze; ad esempio, potrebbe essere opportuno che un attributo di tipo intero di una data classe assuma in realà solo valori naturali. Per avere un controllo sui valori assunti da tale attributo, bisogna mantenere quest’ultimo “nascosto” ai *client*. Per fare ciò (o anche per nascondere le scelte che possono essere soggette a cambiamenti), occorre introdurre il concetto di **visibilità** degli attributi e delle operazioni, che indica quali classi hanno la possibilità di accedere a tali attributi / operazioni e può essere:

- **Pubblica**, se l’attributo / operazione può essere acceduto/a da qualunque classe raggiungibile. In Java si indica col modificatore *public* anteposto all’attributo / operazione. In UML si indica col simbolo “+” anteposto all’attributo / operazione.

- **Privata**, se l’attributo / operazione può essere acceduto/a esclusivamente dalla classe nella quale è definito/a. In Java si indica col modificatore *private* anteposto all’attributo / operazione. In UML si indica col simbolo “-“ anteposto all’attributo / operazione.

- **Protetta**, se l’attributo / operazione può essere acceduto/a esclusivamente dalla classe nella quale è definito/a e dalle eventuali classe figlie (questo aspetto sara maggiormente chiaro più avanti). In Java si indica col modificatore *protected* anteposto all’attributo / operazione. In UML si indica col simbolo “#” anteposto all’attributo / operazione.

La regola pratica che consente di applicare correttamente la tecnica dell’incapsulamento consiste nel porre gli attributi sempre privati o protetti, mentre le operazioni possono essere pubbliche; i *client* dovrebbero poter accedere agli attributi solo se strettamente necessario ed esclusivamente per mezzo di operazioni *get* / *set* (queste ultime in particolare consentono di introdurre qualunque controllo sul nuovo valore da assegnare al relativo attributo).



# Lezione10 – Ereditarietà