



Programmazione Avanzata

Dispense e Appunti delle Lezioni

Autore: *Dott. Gabriele Vianello*

Anno Accademico: 2024-2025

Università Politecnica delle Marche

Indice

Queste dispense sono basate sui contenuti delle lezioni del corso Programmazione Avanzata 2024-2025 e su appunti condivisi dai compagni di corso (e sofferenze). Ulteriore documentazione sui linguaggi si può trovare digitando “man func_name” sul browser, mentre altri appunti nelle repository in <https://www.github.com/Vinello28>.

Sommario

Introduzione	6
Sistemi Runtime e JVM	6
Component Based Programming	6
Design Patterns e Frameworks	6
Linguaggi di Programmazione	7
Implementare un linguaggio di programmazione	7
Intermediate Abstract Machines	8
Altri schemi di compilazione	8
Multicore Processing, Virtualizzazione e Containerizzazione	9
Introduzione	9
Multicore Processing	9
Virtualizzazione	11
Containerizzazione	12
Docker	13
Introduzione	13
Anatomia di un Docker Container	13
Networking	14
Docker Compose	14
Il λ-Calcolo	15
Programmazione dichiarativa	16
Basi di λ-calcolo	16
JavaScript & Python	18
Introduzione	18
Il JavaScript	18
Programmazione a oggetti in JavaScript	21
TypeScript	22
Caratteristiche principali	22
Le interfacce	23

Le classi	23
Composizione.....	24
Esercitazione sul TypeScript.....	24
Moduli.....	26
Seminario Loccioni	27
L'azienda.....	27
Programmazione funzionale.....	28
Pipe	28
Pipe/Reduce e utilizzi.....	28
Introduzione ai Design Pattern	30
Design pattern.....	30
Design Pattern Architetture	32
DAO Pattern.....	32
Repository Pattern	32
Model-View-Controller Pattern	33
Model-View-Presenter Pattern.....	33
Model-View-ViewModel Pattern.....	33
Design Pattern Creazionali.....	35
Singleton Pattern	35
Prototype Pattern.....	35
Builder Pattern	36
Abstract Factory & Factory Method Pattern.....	36
Design Pattern Strutturali.....	37
Adapter Pattern	37
Decorator Pattern	37
Proxy Pattern.....	37
Design Pattern Comportamentali	38
Chain of Responsibility Pattern	38
Observer Pattern	38
Mediator Pattern	40
Reactive Programming	41
Le Promise.....	41
Reactive Extensions for JavaScript	42
Programmazione Asincrona, Fetch e AJAX.....	43
Introduzione ad AJAX	43
Fetch API & Promises	43

JSON	44
REST APIs e HTTP	44
Creare e Gestire Promises Avanzate	44
Suggerimenti Per Progetto & Esame.....	46
Progetto	46
Relazioni.....	46
Errori comuni.....	46

Introduzione

Negli ultimi anni si è resa più evidente la necessità di componenti riutilizzabili, che nel paradigma dell'OOP risultano più complessi da implementare. A questo scopo si possono definire gli ingredienti chiave per realizzare un buon software (complesso).

- *Funzionalità avanzate*, che estendano il linguaggio di programmazione utilizzato.
- *Modelli di componenti*, per massimizzare e assicurarsi la riusabilità.
- *Frameworks*, per supportare uno sviluppo efficiente di applicazioni basate su componenti.
- *Ambienti d'esecuzione*, per fornire supporto a runtime per ogni sistema software dinamico.

Sistemi Runtime e JVM

Come anticipato i sistemi runtime forniscono un ambiente d'esecuzione virtuale, che interfaccia un programma in esecuzione con il sistema operativo. Questi sistemi supportano anche il *Memory/Thread management*, la *Gestione delle Eccezioni e Sicurezza*, la compilazione AOT/JIT, link/load dinamico, *Debugging e Reflection*, *Verification*.¹

Component Based Programming

Alcuni esempi di frameworks basati sui componenti sono il .NET e Spring/Springbeans, ma anche Hadoop Map/Reduce (analizzato nel corso Big Data Analytics e Machine Learning).

Framework: è un insieme di funzionalità “generiche” (relativamente ad un compito specifico) che possono essere sovrascritte o specializzate dall'utilizzatore (developer) per implementare specifiche funzionalità. In altre parole, sono astrazioni riutilizzabili in maniera analoga a quanto accade con le librerie.

Application Framework: framework utilizzato per implementare la struttura di base di un'applicazione per uno specifico ambiente di sviluppo.

Con l'inversione di controllo si seguirà il pattern definito dal framework, così da avere un comportamento di default del codice, mantenendo però la capacità di estendere le funzionalità del framework (e.g. override).

Design Patterns e Frameworks

I design patterns sono più astratti rispetto ai framework, essi vanno a definire gli obiettivi e le conseguenze di un certo design. Generalmente un framework conterrà diversi design patterns ma non è detto il contrario; inoltre, i patterns possono essere solitamente utilizzati in qualunque tipo di applicazione.

¹ Ne è un esempio la Java Virtual Machine.

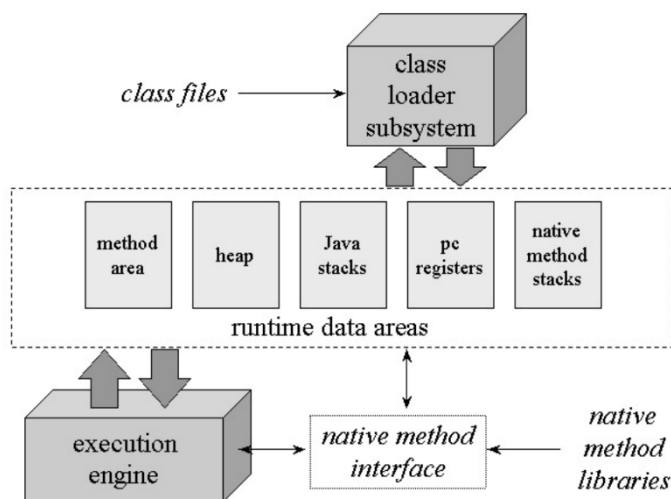
Linguaggi di Programmazione

Sono definiti dalla sintassi, dalla semantica e dalla pragmatica (destinazione d'uso). La sintassi è necessaria al compilatore per scannerizzare e tradurre (parsing) il linguaggio in uno comprensibile dalla macchina; la semantica invece va a definire il comportamento del linguaggio in corrispondenza di certe istruzioni, solitamente si usa il linguaggio naturale.

La pragmatica include le convenzioni da adottare, le linee guida per strutturare il codice e solitamente, anche la descrizione dei paradigmi di programmazione supportati.

Ciascun linguaggio definisce una certa Macchina Astratta che userà proprio questo linguaggio: implementare questa macchina su un'altra detta host (via compilazione/interpretazione) renderà il programma scritto nel linguaggio utilizzato eseguibile.

Quindi è possibile definire una macchina astratta come un insieme di strutture dati e algoritmi in grado di operare lo *storage* e l'esecuzione di un programma scritto nel linguaggio della macchina (astratta) stessa.



A sinistra lo schema rappresentativo della JVM, in evidenza tutti i suoi componenti e come dialogano tra loro. Si tenga presente che alla base dell'interprete JVM vi è uno switch che va ad eseguire un'azione in funzione dei vari opcode dell'istruzione.

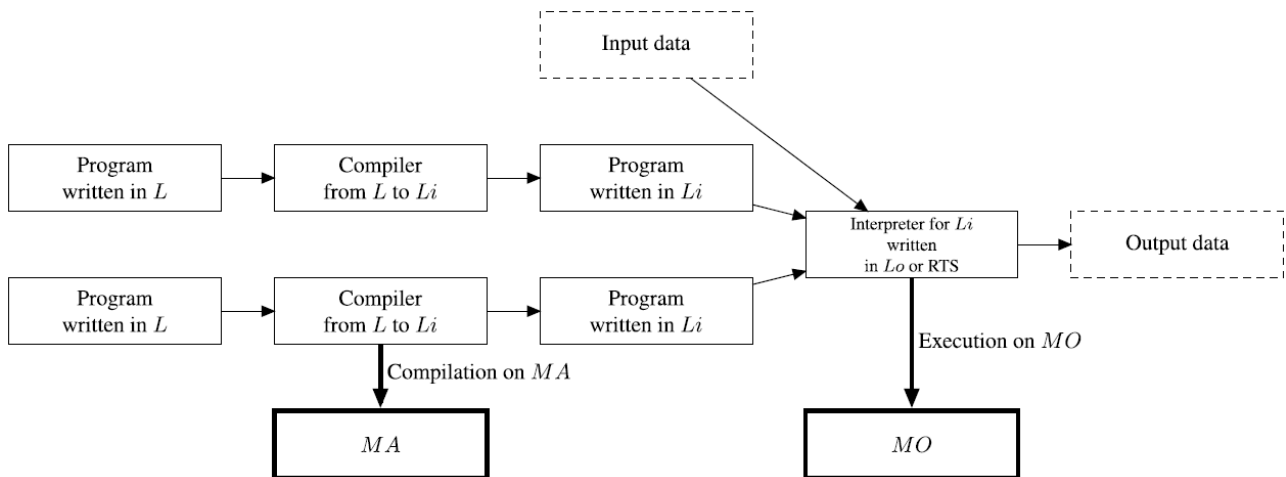
Di fatto si potrebbe implementare una macchina astratta nell'hardware o nel firmware, ma generalmente non conviene per il software ad alto livello. In tal caso si tende a costruire questa macchina come estensione di quella host, oppure interpretata da questa.

Implementare un linguaggio di programmazione

L'interpretazione pura risulta essere generalmente meno efficiente a causa delle operazioni aggiuntive di fetch e decode delle istruzioni. Mentre nella compilazione pura, i programmi scritti in un certo linguaggio sono tradotti direttamente nel linguaggio della macchina host e possono essere eseguiti direttamente da quest'ultima.

Ma i casi in cui si utilizza solo una di queste due modalità (pure, appunto) sono molto rari; in generale vale che la compilazione tende ad avere maggiori performance, con un'ottimizzazione del codice "aggressiva" e allocazioni delle variabili ottimali. Mentre nel caso dell'interpretazione si avrà un debugging e testing molto facilitato, dove si potranno invocare le procedure direttamente da riga di comando dall'utente e le variabili potranno essere ispezionate direttamente dall'utente.

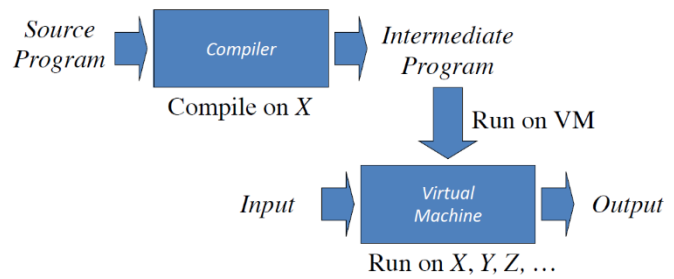
Di fatto si va' ad utilizzare una *Intermediate Abstract Machine* che sarà costruita come evidenziato dal diagramma seguente.



Intermediate Abstract Machines

I compilatori generano il programma “intermedio” che sarà poi interpretato dalla macchina virtuale.

Ne è un esempio concreto quanto accade con il C#, che viene compilato in CIL (common language infrastructure) che potrà essere eseguito in diversi ambienti, come ad esempio il .NET o .NET Core (dove sarà poi compilato per essere eseguito dalla macchina host).



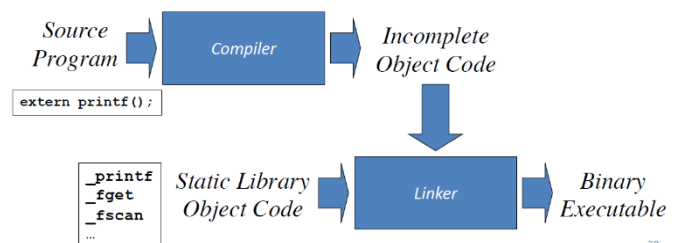
LLVM: è un’infrastruttura di compilazione progettata come insieme di librerie riutilizzabili con interfacce ben definite, ed è implementata in C++.

Alcuni vantaggi derivanti dall’utilizzo di queste macchine sono:

- *Portabilità*, e.g. si compila il source in Java, si distribuisce il bytecode e si esegue su ciascuna piattaforma che abbia una JVM installata.
- *Interoperabilità*, per ogni nuovo linguaggio sarà sufficiente aggiungere un apposito compilatore che traduca il codice nel bytecode JVM, in questo modo potrà utilizzare le librerie Java.

Altri schemi di compilazione

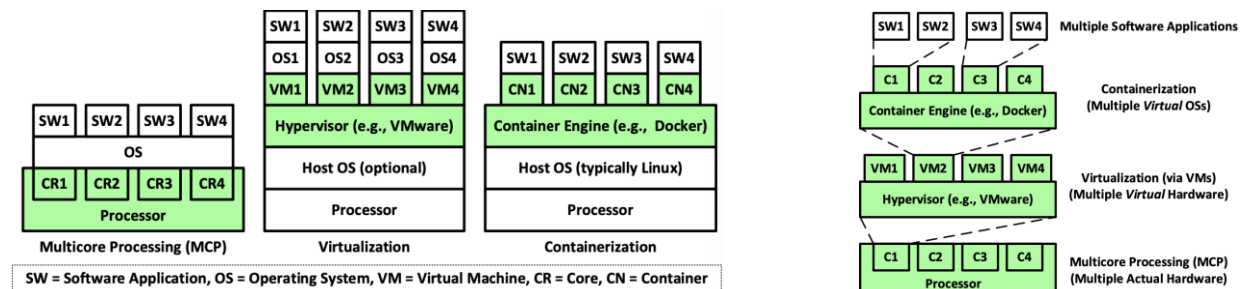
Compilazione pura e static linking: schema adottato tipicamente dai sistemi FORTRAN, dove le routine delle librerie sono linkate separatamente con l’object code del programma.



Multicore Processing, Virtualizzazione e Containerizzazione

Introduzione

I cyber-physical systems sono costruiti con una combinazione di multiprocessore, virtualizzazione e containerizzazione, tre tecniche che possono essere esemplificate dal seguente schema.



Questi sistemi sono caratterizzati dall'avere un'elevata robustezza, soprattutto al crescere della loro dimensione (o numero di sistemi associati). Si differenziano dai sistemi tradizionali in termini di complessità, interferenza e non determinismo; anche per questi motivi richiedono generalmente un'analisi più attenta, un testing più approfondito ed una migliore sicurezza nella politica di certificazione.

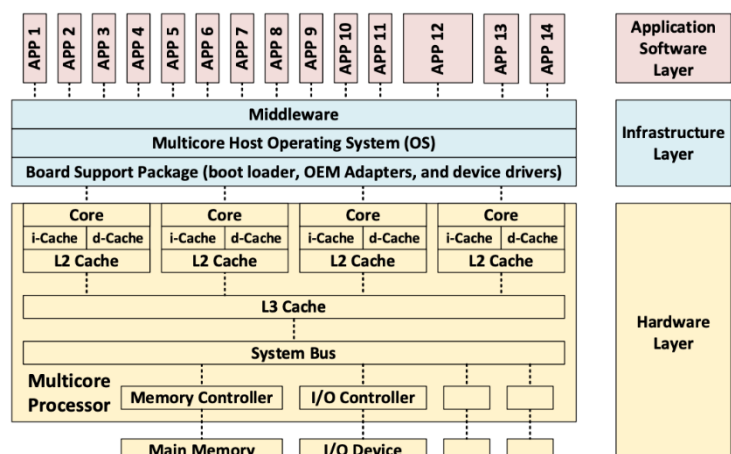
Attenzione! I thread hanno la propria località, pertanto sarà importante scegliere una memoria cache efficiente.

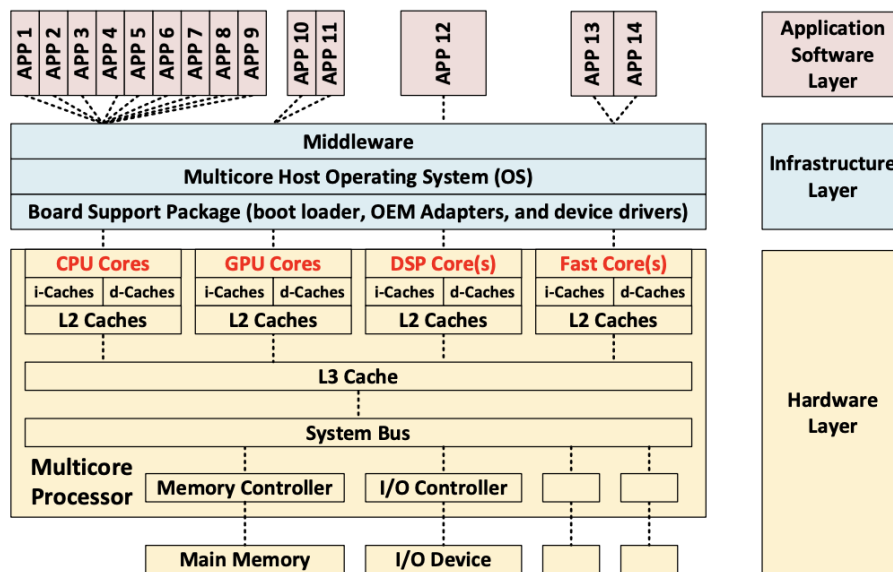
Multicore Processing

Un processore multicore è un singolo circuito integrato che contiene più cpu, possono essere distinti per numero di core, cpu omogenee/eterogenee, livelli di cache, interconnessione dei core. Il supporto per l'isolamento spaziale e temporale in-chip è minima.

Nel caso in cui si abbia un Multiprocessing Simmetrico si avrà una struttura composta come illustrato di seguito; è richiesto l'utilizzo un sistema operativo multicore.

Invece, nel Multiprocessing Asimmetrico si avranno core eterogenei ma stesso sistema operativo; un esempio è una macchina costituita da una processore single/multicore ed una GPU dedicata.





Osservazione: la tendenza attuale è quella di sostituire i sistemi a singolo processore, ma soprattutto di utilizzare sempre più il *multiprocessing asimmetrico*.

Vantaggi:

- Numero ridotto di macchine separate
- Come affermato dalla legge di Moore, minor bisogno di raffreddamento
- Consumi energetici ridotti
- Riduzione di SWAP-C (Size, Weight, Power e Cooling/Cost)
- Concorrenza reale
- Prestazioni elevate (dipende da processore utilizzato)
- Buon isolamento, importante nell'esecuzione indipendente di applicazioni a criticalità mista

Svantaggi:

- Condivisione delle risorse, causa interferenza e single point of failures
- Isolamento spaziale e temporale limitato dalla presenza dell'interferenza²
- All'aumentare del numero di core diminuisce il grado di isolamento del sistema ed aumentano di conseguenza le interferenze.
- Ulteriori problematiche sono causate dalla concorrenza, tra queste si ritrovano le seguenti
 - *Deadlock, livelock, starvation, suspension, data race, priority inversion, order violation, order vulnerability, atomicity violation.*
- Il non determinismo rende l'analisi complessa.

² Si ha interferenza quando il software in esecuzione su un core va' ad impattare il comportamento del software negli altri core ma all'interno dello stesso processore.

Virtualizzazione

Macchina virtuale (VM): anche detta macchina ospite, è la simulazione software di una piattaforma hardware che fornisce un ambiente operativo virtuale per gli OS che ospita.

Piattaforma VM: anche detto sistema VM o *full-virtualization* VM è una macchina virtuale che viene eseguita da un hypervisor³ e simula completamente una piattaforma hardware.

Applicazione VM: anche detto processo VM, è una macchina virtuale che viene eseguita come applicazione software in un certo linguaggio (e.g. JavaVM) dal sistema operativo host.

Anche questo tipo di soluzione è arrivata alla saturazione, almeno a livello server, nelle applicazioni IT/Data Center/Cloud computing. La virtualizzazione è sempre più utilizzata per la memorizzazione di massa, per virtualizzare la rete e per dispositivi mobili; solo di recente è stata introdotta nei sistemi real-time/safety-and-security-critical come l'automotive, IoT o il software ad uso militare.

Vantaggi

- Isolamento notevolmente aumentato (ma non garantito) dall'ambiente virtuale
- Possibilità di utilizzare software scritto per sistemi obsoleti
- Aumenta la portabilità
- Abbatte i costi necessari per l'hardware
- Ottimizzato per il general purpose computing
- Supporta e migliora il failover/recovering e la gestione dinamica delle risorse
- Migliora soprattutto la sicurezza del sistema, limitando l'impatto di eventuali attacchi/malware su singole VMs

Svantaggi:

- Paradossalmente, aumentano i costi dell'hardware → servono risorse per gestire la virtualizzazione (dipende da architettura software)
- Vengono condivise risorse → singoli *point of failure*
- Come nel precedente caso, saranno presenti tutte le problematiche dovute alla concorrenza e al non determinismo/difficoltà di analisi
- Dato che si tratta di una tecnologia "giovane" sono presenti più bug rispetto alle soluzioni precedenti

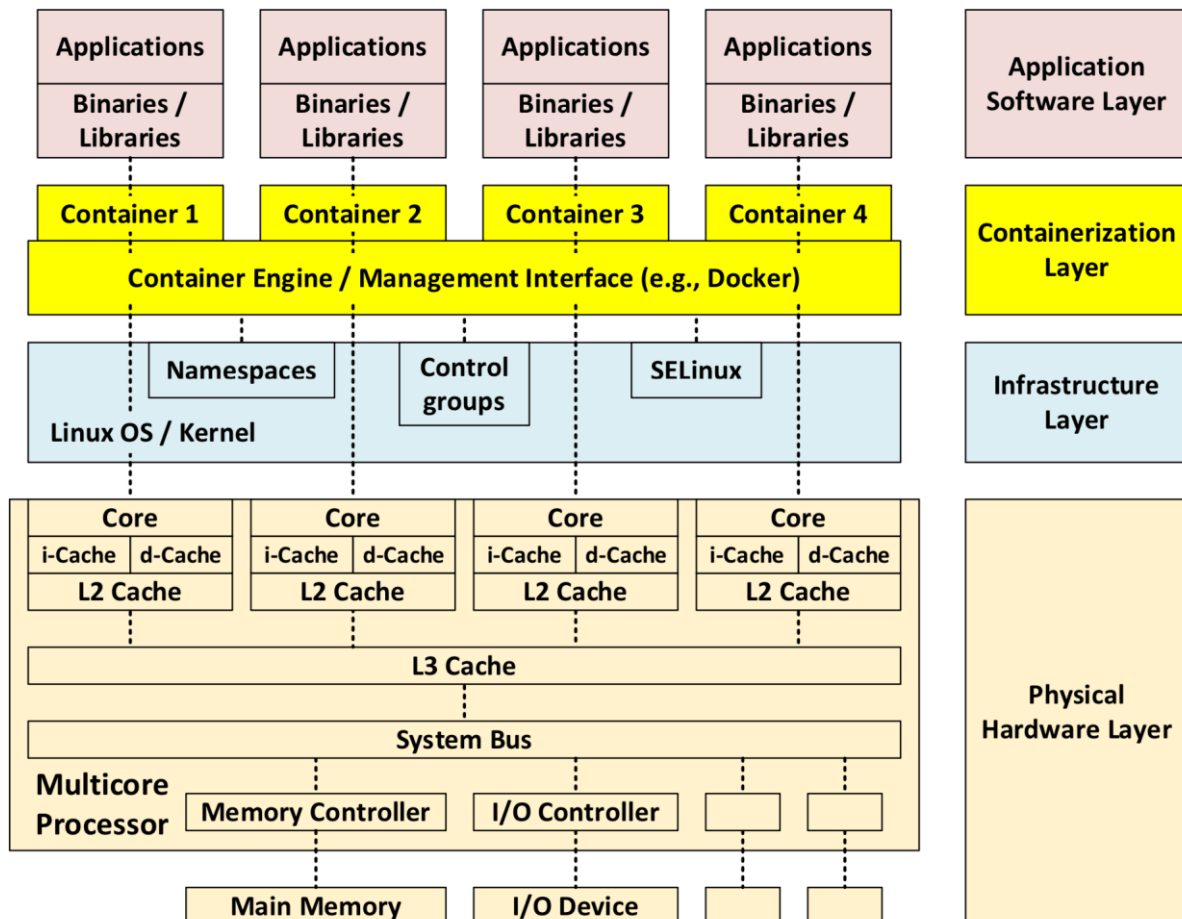
³ Per *Hypervisor*, anche detto VM Monitor, si intende il componente software che viene eseguito sull'hardware host ed opera la supervisione dei sistemi operativi ospiti.

Containerizzazione

Container: è un ambiente d'esecuzione virtuale eseguito da un OS kernel senza emulazione hardware. Talvolta indicata come virtualizzazione attraverso i container.

Pod: è un'insieme di container che condividono risorse.

Containerizzazione: processo di ingegnerizzazione di un'architettura software utilizzando container multipli, ossia sistemi operativi *virtuali* multipli. La gestione dei container è detta *orchestrazione*.



Vantaggi:

- Supporta isolamenti spaziale/temporale leggeri, ogni container ha le sue risorse ed i suoi namespace
- Minor overhead rispetto la virtualizzazione
- Istanziamenti multiple di container individuali relativamente semplici
- Timing consistente a differenza della virtualizzazione
- Supporta DevOps e l'integrazione/sviluppo continui (CI/CD)
- Le applicazioni distribuite sui container possono condividere binari e librerie, ciò porta a ridurre il codice (pro) ma aumenta il rischio di interferenza (contro)

Svantaggi:

- Condivisione delle risorse, come OS kernel e container engine (singoli points of failure)
- All'aumentare del numero di container, aumenta la difficoltà di analisi del sistema
- Per costruzione i container non sono sicuri

Docker

Introduzione

Il sistema Docker impiega i container per eseguire servizi/software in modo isolato e che condividano solamente il kernel del sistema operativo; funziona sulla maggior parte delle distribuzioni Linux, anche se sono nativi di Windows Server 2016.



Docker Image

Example: Ubuntu with Node.js and Application Code



Docker Container

Created by using an image. Runs your application.

Inoltre, si tratta di una piattaforma sicura che semplifica lo sviluppo e distribuzione di applicativi.

Ma qual è il vero vantaggio nell'utilizzo di questa piattaforma?

Il vantaggio di Docker è quello di poter comporre differenti servizi per farli lavorare assieme.

Docker Image: base del container, è un'app completa (e.g. *jar*).

Docker Engine: crea, gestisce ed esegue i Container.

Docker Container: l'unità standard dove risiede ed esegue l'Image.

Registry Service: storage/distribution service per le Images.

Anatomia di un Docker Container

A partire dalla struttura di un *Dockerfile*, si deve tener presente che ciascuno dei comandi utilizzati crea uno strato. Quando si effettua il pull di un immagine si andrà ad effettuare il pull di questi strati.

Docker Volumes: il volume monta una directory sull'host, dentro il container, in una specifica posizione. È utilizzato per montare il codice sorgente locale all'interno di un container in esecuzione.

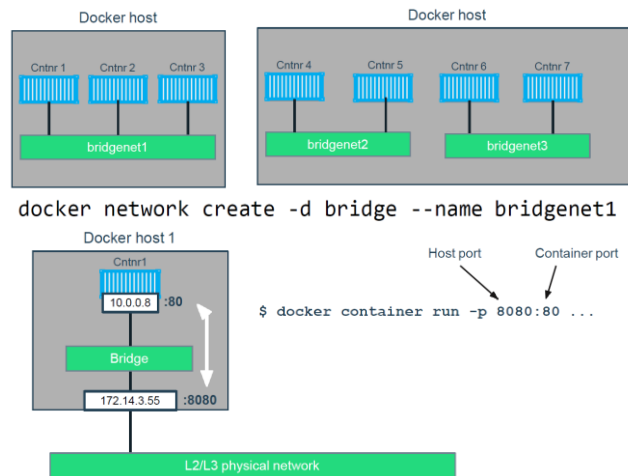
Permette di implementare la persistenza e dei dati nei container: di fatto si fa risiedere il database/tabella/file di database al di fuori del container.

Qualora il container terminasse, i dati non andrebbero persi, ma sarebbero nuovamente disponibili una volta riavviato il container. Inoltre, consentono la condivisione dei dati fra container; per essere eliminati dovranno essere cancellati esplicitamente.

```
1  #---Esempio Dockerfile---
2
3  # Crea un'immagine basata sull'immagine ufficiale Node 6 da dockerhub
4  FROM node:6
5
6  # Crea una directory dove verrà posizionata la nostra app
7  RUN mkdir -p /usr/src/app
8
9  # Cambia directory per eseguire i comandi nella nuova directory
10 WORKDIR /usr/src/app
11
12 # Copia le definizioni delle dipendenze
13 COPY package.json /usr/src/app
14
15 # Installa le dipendenze
16 RUN npm install
17
18 # Ottieni tutto il codice necessario per eseguire l'app
19 COPY . /usr/src/app
20
21 # Espone la porta su cui viene eseguita l'app
22 EXPOSE 4200
23
24 # Avvia l'app
25 CMD ["npm", "start"]
```

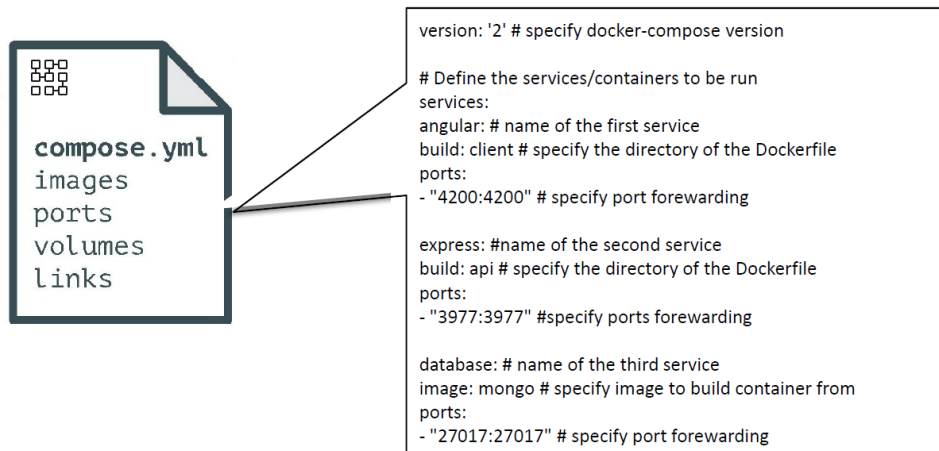
Networking

Il *Docker Bridge Networking* realizza l'interconnessione tra diversi container e rende effettivamente possibile l'interfacciamento con la rete esterna, attraverso il port mapping.



Docker Compose

Consente di definire applicazioni multi-container nel file *compose.yaml*, con un singolo comando si potrà distribuire tutta l'applicazione. Le dipendenze tra container saranno gestite automaticamente, funziona con Docker Swarm, Networking Volumes e Universal Control Pane.



Osservazioni: openmp è un supporto nativo del compilatore che permette di suddividere la computazione utilizzando i thread. Il MAC address è generalmente legato (oltre alla macchina a cui è assegnato) ad eventuali licenze, sia che si tratti di un MAC reale o virtuale. Per avere un accesso molto veloce, non persistente e di piccole dimensioni, si può utilizzare il RAM-FS: un file system che opera nella ram.

Docker ignore: previsto nella realizzazione di un buon progetto, analogo al gitignore è necessario per far in modo che alcuni file/directory/dipendenze non vengano caricati nel container.

Redis: è un sistema di caching che può memorizzare una struttura dati di tipo chiave valore (hash map); può anche essere implementato un TTL o Time-To-Live per cui una volta scaduto si resetta la struttura dati.

Docker-Nodemon: è un dockerfile che, invece di npm/start, lancia nodemon/app.js; in questo modo si implementa l'*hot-reload*, analogo a quello presente nelle applicazioni aspnet e Blazor con visual studio.

Il λ -Calcolo

Il λ -calcolo è un sistema formale in logica matematica che esprime il calcolo basato sull'astrazione e applicazione di funzioni utilizzando l'associazione e la sostituzione di variabili. Prima di procedere, è necessario osservare alcuni richiami teorici, elencati di seguito.

- Programmazione imperativa \rightarrow *procedurale* e a *oggetti*.
- Nel paradigma imperativo, i programmi sono sequenze di istruzioni che prescrivono dettagliatamente quali operazioni effettuare in un certo ordine.
- L'istruzione fondamentale è quella di *assegnamento*, che stabilisce come modificare il contenuto di una locazione di memoria individuata da un identificatore di variabile. Le variabili sono allocate staticamente o dinamicamente.
- Per controllare questo flusso sono utilizzate apposite istruzioni di selezione/ripetizione, è supportata la ricorsione.
- Le istruzioni sono raggruppate in blocchi detti *procedure*, ciascuna delle quali è definita una sola volta nel programma e può essere invocata da più istruzioni del programma. Una procedura può essere definita su dati di valore ignoto, detti *parametri formali*, che saranno inizializzati al momento dell'invocazione (i parametri possono essere passati per valore o per riferimento).
- La procedura invocata può restituire un risultato da utilizzare nell'istruzione contenente l'invocazione.
- Nel paradigma a oggetti si innalza il livello di astrazione ottenuto separando i dati dalle procedure, pur collocandoli vicino a quest'ultime. Le istruzioni vengono raggruppate assieme ai dati, il loro insieme è detto *Classe*.
- *Visibilità* \rightarrow la visibilità degli elementi di una classe (membri) può essere *pubblica*, *protetta* o *privata*. Ciascuna classe espone un'interfaccia costituita dalla dichiarazione dei soli membri pubblici e incapsula la dichiarazione di tutti gli attributi e la definizione di tutti i metodi.
- *Ereditarietà* \rightarrow nella OOP è possibile definire gerarchie di classi; si promuove il riutilizzo del codice attraverso l'estensione di classi già esistenti. Abilita il polimorfismo, cioè la possibilità di una classe derivata da una astratta di completare la definizione dei suoi metodi.
- Quindi, un programma a oggetti è una collezione di oggetti che interagiscono tra loro invocando reciprocamente i relativi metodi per consultarne o modificare i rispettivi dati; inoltre, sono dotati (generalmente) di meccanismi per la gestione delle eccezioni lanciate al verificarsi di determinate situazioni.
- L'oggetto è un'istanza di una classe, assimilabile all'inizializzazione e allocazione dinamica di una variabile avente come tipo la classe che istanzia. Gli oggetti di una classe hanno gli stessi metodi ma possono differire per valore degli attributi.

Programmazione dichiarativa

Si distingue in *funzionale* e *logica*, vede i programmi come collezioni di espressioni matematico-logiche che descrivono cosa deva essere calcolato senza specificarne il nome.

Le esecuzioni di queste istruzioni, sono assimilabili a deduzioni in un sistema formale; in netta contrapposizione a quanto visto al paragrafo precedente, dove si avevano sequenze di passi che modificano il contenuto della memoria.

Side effect: (provocato dall'attivazione di una funzione) per effetto collaterale si intende la modifica di una qualunque tra le variabili esterne. Si possono avere con parametri puntatore o con assegnamento a variabili esterne; se presenti, le funzioni non saranno più funzioni in senso matematico. Per evitare che ciò accada si possono adottare le seguenti soluzioni.

- Non avere parametri passati per riferimento (nelle intestazioni delle funzioni).
- Non introdurre assegnamenti a variabili esterne (nel corpo delle funzioni).

Trasparenza referenziale: indica un'espressione che può essere sostituita con il suo valore senza impattare sul comportamento del programma, cioè che si comporta come prima della modifica. Quindi, ogni volta che quest'espressione sarà valutata sugli stessi argomenti, avrà lo stesso risultato.

- Il risultato di una procedura che usa variabili globali, o passa parametri per indirizzo, può dipendere dal momento in cui è invocata.

Nota: la ricorsione è l'unico strumento linguistico per rappresentare l'iterazione, dove l'unico tipo di dato strutturato disponibile è la lista; la gestione dinamica della memoria per le liste è completamente automatizzata.

Un programma funzionale è una collezione di definizioni di funzioni e relative applicazioni e composizioni, che sfrutta la naturale trasparenza referenziale delle funzioni matematiche.

Basi di λ -calcolo

λ -Calcolo: descrive le funzioni nella loro piena generalità, incluse le ricorsive e quelle di ordine superiore, cioè che hanno funzioni come argomenti. Il tutto attraverso una sintassi semplice che distingue tra *definizione di funzione*⁴ e *applicazione di funzione*.

High Order Functions (HOF): una funzione di ordine superiore è una funzione che accetta altre funzioni come argomenti o restituisce una funzione stessa; oltre ad essere un concetto chiave nella programmazione funzionale, sono utilizzate per implementare determinati algoritmi/schemi di programmazione.

⁴ Anche detta λ -astrazione.

Funzione λ : funzione nella cui definizione non viene specificato il nome, solitamente indicata anche come *funzione anonima/letterale*. In essa sono descritti solamente gli argomenti ed i risultati.

```
function processAnonymous(operation) {  
  switch (operation) {  
    case "ADD":  
      return (a, b) => a + b;  
    case "SUB":  
      return (a, b) => a - b;  
  }  
}
```

Il lambda-calcolo è particolarmente utilizzato per il filtering delle collections, data la sua particolare caratteristica di riuscire ad “impilare” diverse funzioni (filtri) in cascata.

α -equivalenza: proprietà per la quale si considerano identici i termini che differiscono solamente per le variabili legate, in formule $\lambda x \rightarrow \lambda x. x \equiv_{\alpha} \lambda x \rightarrow \lambda y. y$.

Sostituzione: data $M[L/X]$ indica che ogni occorrenza libera di x in M è sostituita da L .

β -riduzione: un termine P si β -riduce in un passo al termine Q se Q può essere ottenuto da P sostituendo un sotto-termini di P , si scrive $P \rightarrow_{\beta} Q$.

Normalizzazione: se un termine non possiede β -riduzioni \Rightarrow è in *forma normale*; se un termine può essere ridotto in forma normale \Rightarrow possiede *una forma normale*.

β -conversione: il termine M è β -convertibile in N sse M può essere trasformato in N (o viceversa) eseguendo zero o più β -riduzioni/espansioni, tale proprietà sarà indicata con $M =_{\beta} N$.

Funzioni di Prima Classe: si possono definire di *prima classe* le funzioni che:

- Si possono usare come argomenti di altre funzioni
- Si possono usare come valore ritornato all'interno di un'altra funzione
- Si possono assegnare a variabili
- Si possono salvare in strutture dati

Funzione identità: la funzione identità riceve in ingresso un parametro/funzione e restituisce in output il parametro (o funzione) stesso; è una funzione del tipo `let f = x => x`.

Nota: nel lambda calcolo, la scrittura $\lambda x. \lambda y. (xy)$ si traduce in codice come `let f = x => y => x(y)`; cioè, funzione x con input y .

JavaScript & Python

Introduzione

Negli attuali paradigmi di programmazione si cerca di non ricorrere all’ereditarietà multipla, quanto piuttosto di utilizzare (implementare) le interfacce. L’architettura da implementare ha una struttura caratterizzata da servizi che eseguono operazioni a seconda delle richieste, in maniera analoga a quanto accade con l’iniezione di servizi presente su altri framework quali *Blazor* (.NET7+).

Strumenti: il professore consiglia di utilizzare le interfacce di *Sentry*, utilizzate anche da *JetBrains*. Sono API che consentono di utilizzare – anche – ambienti cloud ad esempio per il logging.

Compile.sh: questo file contiene tutti i comandi da far eseguire alla shell (automaticamente) per compilare il progetto/soluzione. Per la compilazione di soluzioni java⁵ è preferibile utilizzare *openjdk*.

Attenzione! Se si utilizza java, i file distribuibili sono quelli con estensione “.class”, dato che sono scritti in bytecode.

Curiosità 🔍: le content delivery network (*cdn*) sono utilizzate per caricare o comunque per comunicare al browser/server web di gestire gli script e altri media nelle pagine web. In questo modo si aumenterà la velocità dell’applicazione o sito.

Per evitare di rendere visibile il codice javascript all’interno delle pagine web (attraverso gli strumenti di sviluppo del browser) si utilizza la *minification*: processo che rende il codice meno leggibile rimuovendo tutti i dati e script non necessari. Dopodichè, sarà eseguita l’operazione di *obfuscation*⁶, per rendere ancora più difficile il *reverse engineering* del codice. Sarà comunque necessario adottare accorgimenti fondamentali come non pubblicare mai il software in versione di debug o release, altrimenti sarà facile ricostruire il codice a partire dall’eseguibile.

Il JavaScript



JavaScript: è l’unico linguaggio di programmazione (*interpretato*) che i browser sono in grado di eseguire nativamente; rappresenta l’unica opzione per consentire alle pagine web di eseguire azioni.

Come il *Python*, questo linguaggio non ha la classica tipizzazione, ma è definita in tempo reale; ha una sintassi flessibile, dove non mettere il ‘;’ è facoltativo (e.g.).

Variabili JS

- **var** → *function scope variable* (concettualmente assimilabile a classica variabile, non consigliato)
- **let** → *block scope variable* (più protettivo rispetto al precedente, consigliato)
- **const** → *block scope constant that cannot be reassigned* (massima protezione, costante non riassegnabile)

⁵ I file “.jar” sono essenzialmente delle raccolte/zip di file “.class”.

⁶ Gli *obfuscator* sono strumenti molto importanti che proteggono la proprietà intellettuale.

Un esempio di utilizzo di *let* è all'interno dei cicli *for*, se utilizzassi *var* per la variabile contatore questa avrebbe visibilità globale: **non va bene!**

Sebbene le variabili non abbiano tipi, i valori sì; in javascript esistono sei tipi primitivi e sono *boolean*, *number*, *string*, *symbol*, *null*, *undefined*. *Symbol* sarà un tipo approfondito più avanti nel corso, mentre se si incontra un valore numerico lungo con una lettera 'n' alla fine ⇒ si tratta di un *BigInt* e quella lettera finale è la notazione js necessaria per specificarlo. Inoltre, non si utilizza il '+' per concatenare stringhe.

- Le operazioni bitwise sono supportate nel *JavaScript*, come ad esempio i left/right shift.
- L'operatore ternario è supportato anche nella sua versione multipla.

```
let accessAllowed = (age > 18) ? true : false;
```

- È da sottolineare che *null* viene utilizzato per rappresentare l'assenza di un valore, in maniera simile a quanto visto in Java e C.

Un'altra particolare caratteristica del JavaScript è l'*hoisting*, cioè la possibilità di utilizzare le variabili prima che vengano dichiarate; se da un lato può essere comodo, d'altro canto può disordinare il codice ed essere sorgente di errori: sarà necessario fare attenzione ai nomi delle variabili utilizzati. È da tener presente che comunque i valori saranno assegnati in maniera sequenziale rispetto la posizione del codice. Nel javascript è possibile implementare funzioni innestate/catene di funzioni.

Attenzione! Cercare di scrivere codice dichiarando le variabili più come *let* o *const*, meglio non usare *var*.

Per creare funzioni in javascript si può procedere nel modo *classico*, quindi dichiarando la funzione e implementandone il codice. Oppure si possono definire funzioni anonime (anche innestate) scrivendo solo l'ingresso e il corpo della funzione *inline*; analogamente a quanto visto col C#.

Default parameters: come in altri OOP, anche il javascript supporta i parametri di default sulle funzioni, dove se il parametro non viene passato (di fatto è opzionale) sarà assegnato un valore – appunto – di default. Qualora il parametro non fosse specificato si avrebbe come contenuto "*undefined*"; nel caso si avessero molti parametri non si potrà indicare a chi assegnare il valore indicando il nome del parametro (*named parameter*).

- I parametri mandatori andranno messi tassativamente a sinistra, mentre gli opzionali a destra.

Rest parameters: sintassi dei parametri che permette di dare in ingresso ad una funzione un indefinito numero di valori (parametri). Donano ulteriore elasticità al linguaggio javascript.

Switch: nella nuova versione di javascript è stato aggiunto il *match*, costruito analogo allo switch negli altri linguaggi, con la possibilità di gestire anche case multipli con un solo return.

Objects: sono dei prototipi di classe, vicini ai concetti di dizionari e Map in altri linguaggi, sono coppie di *property name* e *value*; dove il nome di proprietà/chave è una stringa e il valore può essere qualunque cosa.

- Ai valori si può accedere utilizzando la *dot notation*, che può essere adoperata anche per assegnare un nuovo valore. Per eliminare una proprietà, andrà usata la keyword *delete* seguita da *object_name.PropertyName*.

- Allo stesso modo si può usare la notazione con parentesi quadre, dove inserendo il nome della proprietà tra parentesi quadre si avrà accesso al rispettivo valore.
- A differenza dei tipi “primitivi” gli oggetti sono copiati per riferimento.

Shallow copy e deep copy: per implementarle in javascript si dovranno seguire dei pattern. Per effettuare una deep copy si potrebbe serializzare un oggetto in JSON e poi riconvertirlo assegnandolo all’oggetto di destinazione della copia (*dirty method*). La *shallow copy* corrisponde alla copie per riferimento, mentre la *deep copy* alla copia per valore.

Array: consentono di salvare collezioni ordinate, sono dinamici (a dimensione variabile) e come in altri linguaggi si può ottenere il valore inserendo la posizione all’interno delle parentesi quadre. Inoltre, si differenziano da vettori in altri linguaggi dato che non devono obbligatoriamente contenere dati omogenei. Per l’array si può utilizzare sia il *for each* che il *for of*.

Method	Description	Method	Description
<code>arr.push(element)</code>	Add <i>element</i> to back	<code>arr.pop()</code>	Remove from back
<code>arr.unshift(element)</code>	Add <i>element</i> to front	<code>arr.shift()</code>	Remove from front

Method	Description
<code>arr.indexOf(element)</code>	Returns numeric index for <i>element</i> or -1 if none found

Reduce: simile alla funzione affrontata nell’analisi dei *big data*, ha bisogno di una *callback*⁷ e di un *accumulatore* che se non viene passato utilizza come valore di default un array vuoto (0 posizioni). Come il Map, è un operatore che può essere concatenato per eseguire operazioni di filtering e aggregazione (anche calcolo).

Filter: operatore specifico per filtrare strutture dati come le liste/array, accetta una funzione di *callback* che restituisce un booleano e restituisce una lista di oggetti filtrati con solamente quegli elementi che rispettano la condizione passata come funzione.⁸

Attenzione! Per utilizzare in js gli operatori precedentemente indicati, od in generale per la programmazione funzionale, sarà necessario importare la libreria *functools*.

⁷ Parametro puntatore a funzione, può anche essere specificata una *arrow function* nel caso in cui sia una funzione semplice. Ancora una volta si tratta di un concetto presente in diversi linguaggi già affrontati, né sono un esempio il Python ed il C#.

⁸ Assieme a *Map*, *Filter* e *Reduce*, sono funzioni utilizzate nelle *pipe*, ad esempio per la validazione di una collection di dati (e.g. andandola a filtrare).

Programmazione a oggetti in JavaScript

È possibile definire classi in javascript utilizzando una sintassi simile a quella del C++, una funzionalità introdotta nel 2015. Di seguito alcune caratteristiche.

- Il costruttore è opzionale.
- I parametri del costruttore e metodi sono definiti allo stesso modo delle funzioni globali.
- È presente il comando *this* che specifica di fare riferimento all'istanza attuale di un oggetto, come ad esempio un metodo di una stessa classe.
- Nel caso in cui si tratti di una classe ereditata, la prima istruzione del costruttore deve essere *super(...)*.
- I metodi privati sono presenti, ma non da utilizzare (piuttosto si adopera typescript), si dovrà seguire la sintassi seguente *#privateMethodName(...)*. I metodi pubblici invece sono quelli privi di ogni specifica all'infuori del nome del metodo seguito da parentesi.
- Anche per gli oggetti js sono presenti getters e setters, dove per i secondi sarà importante applicare la logica di business del programma; in altre parole, controllare che il dato inserito sia semanticamente corretto (e.g. *valore troppo basso*). Inoltre, si potrebbe far ritornare al *set* il valore appena inserito per garantire la coerenza dei dati agli utilizzatori della classe.
- È consigliato non utilizzare più di 5 parametri in ingresso al metodo costruttore, oltre questo valore è preferibile incapsulare i dati in altri oggetti. Si tratta di una questione puramente logica, non seguire questo “consiglio” non porterebbe ad errori o malfunzionamenti del programma.



Nota: in python, per `__main__` non si intende la *main function* ma il *main script*, cioè lo script principale di un applicativo python.

Nelle condizioni, per controllare l'uguaglianza tra oggetti sarà necessario costruire degli appositi metodi; si dovrà controllare l'area di memoria e poi passare al valore degli attributi.

TypeScript

È un linguaggio open-source⁹ che semplifica il codice JavaScript, lo rende più leggibile e soprattutto ne facilita il debug. Di seguito alcune delle caratteristiche principali del *TypeScript*.



- *Cross platform*, può essere eseguito su qualunque piattaforma che supporti js.
- *OOP language*, supporta classi, interfacce e moduli (sia lato server che client).
- *Static-type checking*, usa i tipi statici e permette il type checking a compile time; inoltre si possono trovare errori mentre si scrive il codice (senza compilarlo).
- *Optional static typing*, volendo si può comunque utilizzare la tipizzazione dinamica del js.
- *DOM manipulation*, può essere usato per manipolazioni sul DOM.
- *Supporta e integra ES6*, include la maggior parte delle funzione e caratteristiche dello standard ECMAScript.

I tipi di dati supportati sono

- | | |
|-----------|---------|
| • number | • enum |
| • string | • union |
| • boolean | • any |
| • array | • void |
| • tuple | • never |

Mentre le variabili possono essere dichiarate utilizzando *let*, *var* o *const*. Si noti che, nonostante sia un linguaggio molto versatile e flessibile, nel TypeScript molti meccanismi/funzioni andranno implementati.

Caratteristiche principali

Script: i file *TypeScript* sono generalmente chiamati “distribuibili”, dato che possono essere spostati e utilizzati con facilità e vengono tenuti separati dal resto del codice (apposita directory). Inoltre, sia le variabili che le funzioni possono avere tipi multipli (utilizzando i generici).

Tuple: nuovo tipo di dato introdotto dal *TypeScript*, è una coppia di valori che possono avere anche tipi diversi; ad esempio il primo valore può essere un numero ed il secondo una stringa. È un tipo che protegge da operazioni di assegnamento.

Enum: le enumerazioni numeriche sono utilizzate per associare ad un valore numerico un’etichetta, uniformando l'utilizzo di quel valore in tutta l'applicazione (e anche nel Database). Un esempio di enumerazione sono i codici di stato dell’http.

⁹ Tutta la documentazione è presente su www.typescriptlang.org.

Non utilizzare le enumerazioni, laddove possibile, è un errore.

Any: tipo di dato che permette di utilizzare la tipizzazione dinamica del javascript.

Void: come in altri linguaggi è principalmente utilizzato come tipo ritornato nelle procedure; può avere come valore *undefined* o *null*.

Never: nuovo tipo introdotto dal linguaggio, viene utilizzato come valore di ritorno delle funzioni che non ritornano mai al loro end point oppure che lanciano sempre un'eccezione.¹⁰ A differenza del precedente non può avere valori.

Cast: per effettuare il cast delle variabili si può usare, ad esempio, `as number` oppure la *bracket notation*.

Le interfacce

Interface: l'interfaccia di un'applicazione *TypeScript* definisce la sintassi e la struttura che le classi dovranno seguire, di fatto dichiara "l'intenzione"; possono essere gestiti come tipi, con un costrutto simile alle struct in C. Possono anche essere usate come tipo di funzione o come tipo di array. Per poter implementare un'interfaccia si dovranno implementare, obbligatoriamente, tutti i suoi metodi e gli attributi non opzionali.

- Un'interfaccia può *estendere* una o più interfacce.
- Come in altri linguaggi, è bene che l'estensione (come ereditarietà tra classi) sia effettuata al più una volta per classe.

Proprietà opzionali: dato che potrebbero non essere necessari tutti gli attributi dell'interfaccia, allora si dichiarano opzionali alcuni di essi inserendo un `?` subito dopo il tipo dell'attributo.

Proprietà in sola lettura: per proteggere quando necessario alcuni attributi delle interfacce, si può utilizzare la tipologia di variabile `readonly`, che rende modificabile una sola volta l'attributo contrassegnato (assegnamento operato solitamente dall'interfaccia).

Le classi

Si definiscono come nel *JavaScript*, dove possono includere costruttori, proprietà e metodi.

Ereditarietà: si usa l'ereditarietà singola, e si possono usare i comandi `extends` e `super` per rispettivamente, estendere la classe padre e passargli i parametri. Si osservi che una classe, come in altri linguaggi, può implementare una o più interfacce.

Problema del diamante: l'ereditarietà multipla è la capacità di una classe di poter ereditare più classi; non sono molti i linguaggi in grado di supportarlo, a causa del *problema del diamante* \Rightarrow data una classe padre che viene ereditata da due classi figlie, ne esiste una quarta che estende le due classi figlie, allora si avrà il problema in esame: porterà ad avere ambiguità nei metodi implementati dalla classe padre, dove il compilatore non

¹⁰ Perlopiù utilizzato nelle funzioni utilizzate per la gestione delle eccezioni.

riuscirebbe a scegliere quale metodo eseguire. Si tratta di un conflitto sui dati, perché i metodi avranno tutti la stessa firma.

Come in altri linguaggi di programmazione a oggetti, le classi e gli attributi *TypeScript* supportano gli identificatori di visibilità `private`, `protected` e `public` (ordinati dal più al meno protettivo). Generalmente si contrassegnano con `public` i metodi e `private` gli attributi, così da garantire il più possibile il rispetto della logica di business.

Quando un metodo è dichiarato `static` non è più un metodo dell'oggetto, ma una funzione della classe.

Generici: saranno utilizzati per soddisfare un'esigenza specifica, cioè l'accesso strutturato e regolamentato ad una base di dati (utilizzando il DAO pattern). Nel Typescript sono simili a quelli del C#, assicurano che il programma sia flessibile e scalabile nel lungo termine.

Composizione

La composizione evita le ambiguità tra metodi (problema del diamante) perché usa oggetti come proprietà o delega comportamenti; inietta questi oggetti utilizzabili nelle pagine per fare in modo che siano utilizzabili, il tutto senza ereditarietà.

Vantaggi:

- *Separazione delle responsabilità*
- *Maggiore riutilizzo del codice*
- *Testabilità migliorata*
- *Nessun vincolo gerarchico rigido*
- *Facilità di estensione o sostituzione dei comportamenti*

Esercitazione sul TypeScript

Specifica: realizzazione di una struttura dati generica in TypeScript, che contenga i seguenti elementi:

1. Implementare una struttura dati generica `Set<T>` usando solo vettori
2. Definire l'interfaccia `MySet<T>` con i seguenti metodi
 - a. `Add(value:T):void`
 - b. `Remove(value:T):void`
 - c. `Has(value:T):boolean`
 - d. `Intersect(other:MySet<T>):MySet<T>`
3. Creare una classe `ArraySet<T>` che implementi `MySet`
4. `T` deve estendere l'interfaccia `Equatable<T>` con il seguente metodo
 - a. `Equals(other:T):boolean`
5. Niente `Set` nativi, solamente array e confronto personalizzato
6. Creare infine un codice di esempio attraverso la classe `Persona` che implementi `Equatable`

A pagina seguente il risultato dell'esercizio, è possibile scaricare tutti gli esercizi svolti da github.

Risultato:

Vinello28, yesterday | 1 author (Vinello28)

```
interface MySet<T>{  
    add(value: T): void;  
    remove(value: T): void;  
    has(value: T): boolean;  
    intersect(otherSet: MySet<T>): MySet<T>;  
}
```

Vinello28, yesterday | 1 author (Vinello28)

```
class ArraySet<T> implements MySet<T> {  
    private Set: T[] = [];  
  
    add(value: T): void {  
        if (!this.has(value)) {  
            this.Set.push(value);  
        }  
    }  
  
    remove(value: T): void {  
        const index = this.Set.indexOf(value);  
        if (index !== -1) {  
            this.Set.splice(index, 1);  
        }  
    }  
  
    has(value: T): boolean {  
        return this.Set.indexOf(value) !== -1;  
    }  
  
    intersect(otherSet: MySet<T>): MySet<T> {  
        const intersection = new ArraySet<T>();  
        for (const value of this.Set) {  
            if (otherSet.has(value)) {  
                intersection.add(value);  
            }  
        }  
        return intersection;  
    }  
}
```

Vinello28, yesterday | 1 author (Vinello28)

```
interface Equatable<T>{  
    equals(other: T): boolean;  
}
```

Moduli

Nel Typescript tutto il codice è scritto nel *global scope* di default, quindi il codice in certo file sarà accessibile da altri file. Per far ciò sarà necessario usare il comando `export` sul file che si intende importare, ed `import` del file esportato in corrispondenza del file dove si intende utilizzarlo.

In modo simile a quanto visto nel Python, si può importare un modulo come variabile e, di conseguenza, gestirlo come tale; per compilare un modulo¹¹ si dovrà scrivere sul terminale di nodeJS il seguente comando:

```
tsc --module <target> <file path>
```

In breve, un modulo è un file che contiene codice js o ts, permette di suddividere il codice in parti riutilizzabile, generalmente ogni file è un modulo.

Namespace: utilizzato per il raggruppamento logico di funzionalità, può includere interfacce, classi, funzioni e variabili. Di default, i componenti dei namespace non sono visibili in altri namespace, quindi dovranno essere importati.

¹¹ Qualora si volessero compilare tutti i file con un solo comando, sarebbe sufficiente includerli o comunque specificarlo nell'apposito file *tsconfig.json* che contiene tutte le informazioni per la compilazione del progetto. In questo caso, il comando da usare sarebbe `tsc`.

Seminario Loccioni

L'azienda

Si occupa di *misure e miglioramento delle qualità*, nei settori dell'elettrificazione, della decarbonizzazione e digitalizzazione.

DevOps: lo sviluppo del software gestionale interno sfrutta docker e la suite di sviluppo azure, quindi C#, MAUI, .NET (in generale).

Inoltre, viene sottolineata l'importanza dell'implementazione dei pattern; tra i più utilizzati in Loccioni vi sono lo *strategy pattern* ed il *controller service repository pattern*.

Database Vettoriali: curiosità.

Possibilità di svolgere il tirocinio e tesi con l'azienda, a livello informatico si opererà principalmente sul loro gestionale, altrimenti sui banchi di test qualora ci sia interesse più verso l'automazione.

The logo for Luccioni, featuring the word "LUCCIONI" in a bold, red, sans-serif font. The letters are closely spaced and have a slightly blocky, modern appearance.

Programmazione funzionale


Pipe

Pipe: forma di reindirizzamento, cioè il trasferimento dello standard output verso altre destinazioni, particolarmente usato nei sistemi UNIX/Linux.¹² Questa tecnica viene utilizzata per combinare due o più comandi, dove l'output del primo viene dato in input al secondo e così via; a questo punto risulta evidente la caratteristica unidirezionalità della *pipe*, infatti i dati “scorrono” idealmente da sinistra verso destra.

In altre parole, la pipe è utilizzata come inter-cross-communication all'interno dei sistemi operativi per lavorare con una sequenza di processi. Particolarmente utilizzato con Angular; permette di lavorare direttamente sui valori.


I programmi richiamati con questa modalità, cioè dando in input i risultati precedenti, sono detti filtri; le *pipe*, infatti, sono generalmente utilizzate per realizzare le *catene di validazione*.

Esempi: di seguito alcuni esempi più o meno semplici di utilizzo delle *pipe*.



```
#Ex. 1
pstree -pci | grep python

#Ex. 2
cat s1.py | grep in
```



```
/*
sumByTwo e multiplyByThress → non si passa mai il valore di un fattore direttamente,
ma si danno una serie di valori alternativi a quelli di default per gestire gli errori
*/

let sumByTwo = x ⇒ x+2;
let multiplyByThress = x ⇒ x*2;
let pipe = (funA, funB) ⇒ arg ⇒ funB(funA(arg));

let p1 = pipe(sumByTwo, multiplyByThress);
let o = p1(3);
```

Pipe/Reduce e utilizzi

Con *Reduce* il numero di funzioni invocabili nella *pipe* diventano *N*, il che porta ad una maggiore flessibilità e rende superfluo lo specificare la struttura di chiamata delle funzioni.¹³

Utilizzo operativo: le *pipe* sono utilizzate per implementare il design pattern *Chain Of Responsibility*, cioè il pattern che definisce la struttura della *catena di validazione*; viene impiegato soprattutto nei middleware. Un'ulteriore utilizzo delle *pipe* è per impostare le *local-settings* in base al paese di riferimento.

¹² Per far ciò, nella shell dei comandi si utilizza il carattere | (pipe).

¹³ Data la semplicità introdotta dall'utilizzo del *reduce*, è preferibile usare questa tecnica invece della precedente.

Esempi: di seguito alcuni esempi di *pipe* che impiegano il meccanismo di *reduce* e la realizzazione della catena di validazione.

```
/*E' una funzione pass-true → viene sempre passato il valore numerico (no risultato  
altre funzioni. Appena un valore è "False" la catena viene fermata  
(cfr. ChainOfResponsability))  
*/  
pipe{ isNumber, isGreaterThan20, isaMultiplyOf3, formatCurrencyValue}(12)  
let checkMultiple3 = arg ⇒ {  
  if (arg % 3 === 0)  
    return arg;  
  else  
    throw new Error("Invalid number ...");  
}
```

JS

Introduzione ai Design Pattern

Pattern software: soluzione provata e applicabile ad un particolare problema di progettazione, descritta in forma standard cosicchè possa essere facilmente condivisa e riutilizzata. Tra i design patterns si possono distinguere tre differenti tipologie, *pattern architetturali*, *design pattern* e *idiomi*.

In altre parole, un pattern architetturale è un pacchetto di decisioni progettuali che è stato applicato ripetutamente, ha delle proprietà note che ne consentono il riuso e descrive una classe di architetture.

Analisi: nello studio di un pattern architetturale è necessario comprenderne le proprietà, mentre può risultare utile la comprensione delle singole decisioni progettuali implicate dal pattern.

Design pattern

Design pattern: descrizione di oggetti e classi che comunicano, personalizzati per risolvere un problema generale di progettazione in un contesto particolare. Si suddividono in pattern *creazionali*, *strutturali* e *comportamentali*.¹⁴

Anch'essi hanno un ruolo fondamentale nella definizione dell'architettura di una soluzione software, in particolare per raffinare gli elementi del sistema o la loro relazione, oppure per descrivere le connessioni tra diversi elementi architetturali. Inoltre, favoriscono la comunicazione tra l'architetto della soluzione e gli sviluppatori.

Pattern creazionali: consentono di fornire un livello di astrazione relativamente alla creazione di oggetti (istanziamento); i principali sono *Factory*, *Builder*, *Lazy initialization*, *Prototype*, *Singleton* e *Double-Checked Locking*.

Pattern strutturali: descrivono come comporre oggetti o classi per creare strutture complesse; i principali sono *Adapter*, *Bridge*, *Composite*, *Container*, *Decorator*, *Extensibility*, *Facade*, *Flyweight*, *Proxy* e *Private class data*.

Pattern comportamentali: descrivono come classi e oggetti interagiscono e distribuiscono fra loro delle responsabilità. I tipi principali sono *Chain-of-Responsibility*, *Command*, *Event-listener*, *Hierarchical-Visitor*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *Single-serving-visitor*, *State*, *Strategy*, *Template-method*, *Visitor* e *Null-object*.

¹⁴ Ulteriore materiale e approfondimenti su *Elements of Reusable Object-Oriented Software* ricercabile su Google Scholar, contiene 23 pattern utilizzati per risolvere problematiche ricorrenti nella programmazione a oggetti.

Object pattern o class pattern?

- *Pattern creazionali di classe* → risolvono il problema della creazione di oggetti attraverso la delega a sottoclassi.
- *Pattern di oggetti creazionali* → risolvono il problema attraverso la definizione di specifiche interazioni tra oggetti.
- *Pattern comportamentali di classe* → usano l'ereditarietà per definire algoritmi e flussi di controllo.
- *Pattern di oggetti comportamentali* → definiscono interazioni tra oggetti che permettano di svolgere una determinata attività.

Idioma: pattern di basso livello specifico di un linguaggio di programmazione, descrive come implementare aspetti particolari di elementi o relazioni tra essi, utilizzando una caratteristica di un certo linguaggio.

Componente: parte di codice relativamente indipendente la cui principale caratteristica è quella di essere riutilizzabile; un esempio di componente sono gli elementi grafici utilizzati all'interno di una *single page app*, come degli oggetti "card" utilizzati in una pagina xaml.

Design Pattern Architeturali

Pattern architetturale: pattern per definire l'architettura del software, affronta un problema architetaturalmente significativo proponendo una soluzione anch'essa strutturale. Esprime uno schema per l'organizzazione strutturale delle soluzioni software; è un'insieme di elementi architetaturali predefiniti, ciascuno con le proprie responsabilità, regole e linee guida per l'organizzazione della relazioni tra loro.

- Alcuni esempi di *pattern architetaturali* sono il *Layer*, *MVC*, *MVVM*, *DAO*, *DTO*, *Client-Server*, *Pipe-And-Filter*, *Broker*, *Repository*, *MicroKernel* ed il *Microservices* pattern.
- Ogni sistema è basato su un proprio pattern architetattuale *principale*, che potrà essere integrato con ulteriori pattern architetaturali *secondari*.
- Lo “stile” (o natura) di un sistema è definito dal pattern architetattuale principale.

DAO Pattern

Definizione: il *data access object* è un pattern strutturale che offre un astrazione della persistenza dei dati, considerato generalmente vicino al sistema di storage, solitamente *table-centric*.

Consente di isolare il livello di persistenza da quello di business o applicativo, utilizzando un'API astratta.

- L'API astratta nasconde la complessità delle operazioni CRUD, permettendo ai differenti livelli di evolvere in modo autonomo.

Implementazione: ogni classe avrà il suo sistema di persistenza, o meglio, una sua classe DAO; generalmente si ha un'API DAO per ciascun dominio, come ad esempio uno per gli utenti, uno per le merci e così via. Per far ciò sarà necessario implementare tale API utilizzando i generici.

Repository Pattern

Definizione: il *repository* pattern è un meccanismo per l'incapsulamento di storage, ricerca e risposta che emula una collezione di oggetti. Come nel DAO, lavora con i dati e nasconde la complessità delle query, ma si opera ad un livello più alto, più vicino alla logica di business dell'applicazione.

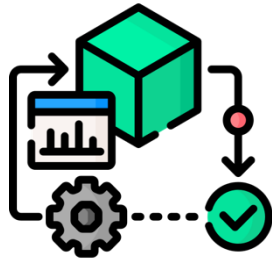
Un repository necessita del DAO per accedere ai dati e popolare gli oggetti del dominio, così come nel verso opposto, dove si occupa di preparare i dati degli oggetti per mandarli al sistema di storage attraverso il DAO.

Confronto con DAO

- DAO è un'astrazione della persistenza dei dati, opera a basso livello (interagisce col DB) nascondendo la complessità dovuta alle query.
- Repository è opera ad alto livello interagendo con gli oggetti del (suo) dominio, nasconde la complessità di preparazione e raccolta dei dati.
- Il DAO non può essere implementato utilizzando un repository, mentre un repository può essere implementato sfruttando DAO per l'accesso ai dati.

Model-View-Controller Pattern

MVC: rappresenta l'intera logica di business dell'applicazione, caratterizzato dai suoi tre elementi principali, descritti brevemente di seguito.¹⁵



- *View* → responsabile del rendering degli elementi UI, rappresenta i dati “fetchati” dal modello e nasconde la complessità di presentazione (difficile produrre unit test per questo modulo).
- *Controller* → responsabile dell'esecuzione di azioni dell'UI, aggiorna il modello e gestisce la logica del flusso di controllo.
- *Model* → gestisce la logica di business e tiene traccia dello stato.

Nella maggior parte delle applicazioni, questi tre elementi dialogano direttamente tra loro, mentre in altri approcci il controller determina anche quale vista mostrare.

Osservazione: *view* e *model* sono strettamente accoppiati, caratteristica che potrebbe portare ad avere una logica di business inquinata da features richieste dal livello di visualizzazione.

Use case: un esempio di utilizzo del pattern MVC si può ritrovare nella repository su GitHub “Library”, dove è contenuto il codice di applicativo java per gestire una biblioteca.

Model-View-Presenter Pattern

MVP: il *model view presenter* è un pattern di presentazione basato sui concetti dell'MVC, senza specificare come strutturare il sistema, ma solamente la *view*. Si progetterà il layer di presentazione come un insieme di *view* (rendering UI) e *presenter* (agisce anche come *controller*) dove quest'ultimo interagirà con *view* e *model* per gestire il comportamento di business e lo stato.

Osservazione: uno dei principali vantaggi dell'utilizzo di questo pattern è rappresentato dalla possibilità di testare parte del codice della *view*, cioè quello del *presenter*.

Model-View-ViewModel Pattern

MVVM: il *model view viewmodel* è un pattern architetturale basato sui concetti dell'MVC e dell'MVP, il cui obiettivo è quello di separare ulteriormente lo sviluppo delle interfacce utente da quello della logica di business e comportamento in un'applicazione.

Permette agli sviluppatori dell'UI di implementare bindings alla *ViewModel* nei documenti scritti in linguaggi di markup¹⁶, mentre gli sviluppatori backend si occuperanno del fornire i dati alla *ViewModel* stessa.

¹⁵ Il pattern MVC non specifica come *view* e *model* devono essere strutturati internamente, anche se generalmente si preferisce implementare il livello di *view* come una singola classe.

¹⁶ Tra questi HTML e XAML come nel caso di MAUI, framework basato su MVVM pattern.

Differenze con precedenti pattern:

- *Model* → in questo caso non si occupa della gestione del comportamento dell'applicazione, ma solamente di contenere informazioni.
- *ViewModel* → si occupa di gestire il comportamento dell'applicazione (business logic), è essenzialmente un *controller* specializzato che si occupa di convertire dati e gestire gli eventi scatenati dagli utenti nella *view*.
- *View* → si occuperà solamente della presentazione e formattazione dei dati, gestisce gli eventi passandoli al *ViewModel*.

Osservazione: è preferibile utilizzare questo pattern per applicativi UI-intensivi, dato che per interfacce semplici risulterebbe eccessivo. Mentre in applicativi di grandi dimensioni potrebbe risultare difficile riuscire a generalizzare il *ViewModel*.

*Quando si scrivono le rotte si deve mantenere la convenzione del naming, senza descrivere nella rotta stessa anche l'azione da performare su quel dominio. In altre parole, **i nomi delle risorse devono essere semantiche!***

Nota: si sconsiglia di utilizzare *beautify* nelle API, dato che aggiungerebbe degli spazi per migliorare la leggibilità delle risposte json porterebbe ad uno spreco di spazio, fondamentale in ambienti cloud; un ulteriore motivo, è rappresentato dal fatto che negli ambienti cloud solitamente si paga anche la banda occupata dalle risposte, per cui questi spazi risparmiati si tradurrebbero in crediti di utilizzo risparmiati.

Design Pattern Creazionali

Pattern creazionale: fornisce dei meccanismi standardizzati per la creazione di oggetti, massimizzano la flessibilità ed il grado di riutilizzo dei componenti implementati. I principali sono il *factory method*, l'*abstract factory*, *builder*, *prototype* e *singleton* (tra i più utilizzati).

Singleton Pattern

Singleton: pattern creazionale, garantisce che una classe abbia una singola istanza, fornendo un sistema globale per accedere a tale oggetto. Il suo principale svantaggio è la violazione del *Single Responsibility Principle*¹⁷, insieme ai problemi di sicurezza dovuti alla possibilità di accedere globalmente a questo oggetto.

Funzionamento: sarà necessario avere un costruttore (di default) privato, per prevenire che altri oggetti possano richiamarlo e creare l'istanza di quella classe. Quindi sarà implementato un metodo di creazione statico che funzionerà da costruttore per la classe *singleton*, il cui scopo sarà richiamare il costruttore privato e salvando l'oggetto in un campo statico.

Utilizzo: è consigliato utilizzarlo quando si opera con i database, dove è preferibile avere un singolo oggetto di interfaccia col DB che viene poi condiviso tra diversi client/parti del programma.

```
/** * The Singleton class defines the `getInstance` method that lets clients access * the unique singleton instance. */
class Singleton
{
    private static instance: Singleton;
    /** * The Singleton's constructor should always be private to prevent direct * construction calls with the `new` operator. */
    private constructor() {} /** * The static method that controls the access to the singleton instance. * This implementation let you subclass the Singleton class while keeping * just one instance of each subclass around. */
    public static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        } return Singleton.instance; }
    /** * Finally, any singleton should define some business logic, which can be * executed on its instance. */
    public someBusinessLogic() { // ... }
}
```

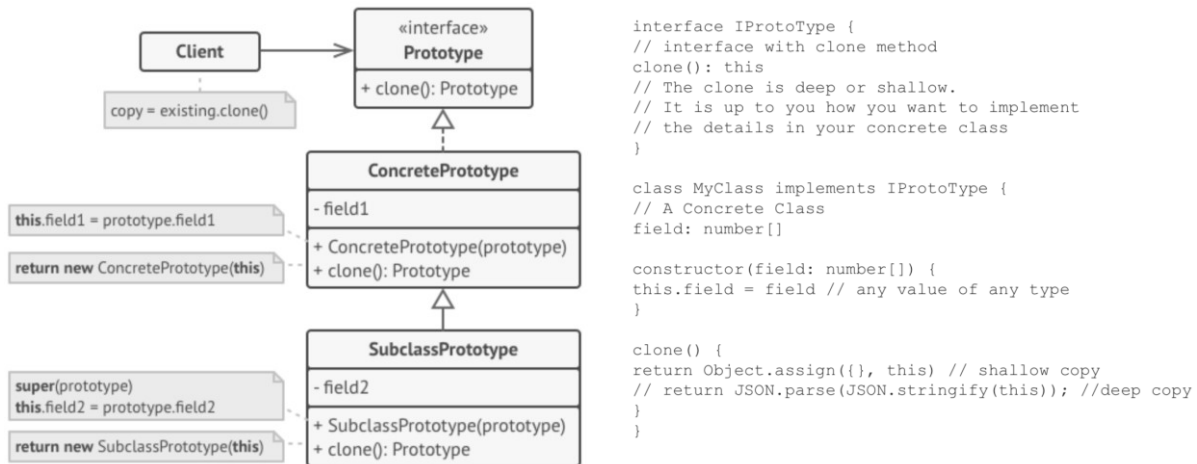
Prototype Pattern

Prototype: ha l'obiettivo di copiare oggetti esistenti senza rendere dipendente il codice dalle classi, delega il processo di clonazione agli oggetti che devono essere clonati. Un oggetto che supporta la clonazione è detto *prototype*.

Funzionamento: viene dichiarata un'interfaccia comune agli oggetti clonabili, le classi che la implementeranno dovranno a loro volta implementare il metodo `clone()`; questo metodo creerà un oggetto della classe attuale copiando tutti gli attuali valori degli attributi nel nuovo oggetto.

¹⁷ Principio fondamentale della programmazione a oggetti, dove un singolo modulo/classe/funzione ha piena responsabilità su una certa parte di programma.

Utilizzo: utilizzato quando si vuole rendere la propria implementazione indipendente dalle classi di oggetti che si vogliono copiare/clonare. Sarà necessario utilizzare un apposito costruttore che accetti un oggetto di quella classe come argomento.



Vantaggi: disaccoppiare il codice di clonazione dalle singole classi, dando la possibilità di produrre oggetti complessi in modo semplificato

Limiti: la clonazione di oggetti complessi, ad esempio con riferimenti circolari, risulterebbe particolarmente difficoltoso.

Builder Pattern

Builder: consente di costruire oggetti complessi *step-by-step* e producendo differenti tipi e rappresentazioni di un oggetto utilizzando lo stesso codice di “costruzione”.



Funzionamento: incapsula la creazione e l’assemblaggio delle parti di un oggetto complesso in un oggetto *Builder* separato; in altre parole, la classe delega la costruzione dei suoi oggetti ad un altro oggetto detto *Builder*, invece che crearli direttamente. Sarà implementata una classe di livello superiore, detta *Director*, che si occuperà di dirigere la costruzione dell’oggetto richiamando gli appositi metodi.

Abstract Factory & Factory Method Pattern

Abstract Factory: ha lo scopo di incapsulare il codice per la creazione di una famiglia di oggetti su un oggetto *Factory* separato.

Funzionamento: tutte le famiglie di oggetti supportati avranno un’apposita interfaccia comune, quindi si creerà una classe *concrete Factory* che implementerà tale interfaccia.

Factory Method: fornisce un’interfaccia per creare oggetti in una superclasse, consentendo alle sottoclassi di modificare il tipo di oggetti che saranno creati.

Funzionamento: si dovrà implementare un’interfaccia o estendere una classe astratta appositamente creata per istanziare oggetti differenti; contiene *Creator*, *ConcreteCreator*, *Product* e *Concrete Product*.

Utilizzo: un esempio è per creare differenti errori o lanciare eccezioni diverse in maniera automatizzata.

Design Pattern Strutturali

Gli *Structural Design Patterns* definiscono come assemblare oggetti e classi in strutture complesse, pur mantenendole flessibili ed efficienti.

Adapter Pattern

Adapter: aiuta ad adattare un'API¹⁸ in un'altra, è essenzialmente un *wrapper* di una certa classe/oggetto che fornisce un'API differente ma utilizzando l'oggetto originale.

Funzionamento: coinvolge una singola classe, con la responsabilità di unire le funzionalità di due interfacce indipendenti o incompatibili. Saranno implementate le classi *Target* (interfaccia presente lato client), *Adaptee* (libreria di oggetti), *Adaptor* (converte e adatta le chiamate del client alla libreria).

Decorator Pattern

Decorator: permette di aggiungere nuovi comportamenti agli oggetti, semplicemente inserendoli all'interno di apposite classi wrapper. Per il precedente motivo, viene anche indicato come *wrapper pattern*.

Adapter vs Decorator: la principale differenza risiede nel fatto che l'*adapter* va a cambiare l'interfaccia di un oggetto esistente, mentre il *decorator* integra l'oggetto senza modificarne l'interfaccia. Inoltre, il pattern *decorator* supporta la composizione ricorsiva, non possibile con l'*adapter*.

Funzionamento in breve: il decoratore è una funzione che aggiunge informazioni o azioni ad una classe, ai suoi membri o agli argomenti dei suoi metodi; essenzialmente, il decoratore non è altro che una funzione, la quale permette l'accesso all'oggetto della "decorazione".

Annotation: le annotazioni non sono altro che metadati sulle classi che riflettono la libreria di metadati. Un esempio di annotazione è `@Retryable`.

Proxy Pattern

Proxy¹⁹: è un "oggetto" che ne controlla un altro detto *subject*, entrambi condivideranno una stessa interfaccia che permetterà loro di "scambiarsi" in modo trasparente.

Funzionamento: "wrappa" le istanze del soggetto, mantenendone lo stato; di fatto protegge l'oggetto agendo da intermediario ed è particolarmente utile per la validazione. Come il decoratore, supporta la composizione.

¹⁸ Per API si intende l'insieme dei metodi di un particolare oggetto.

¹⁹ Anche conosciuto con il nome di *Surrogate Pattern*.

Design Pattern Comportamentali

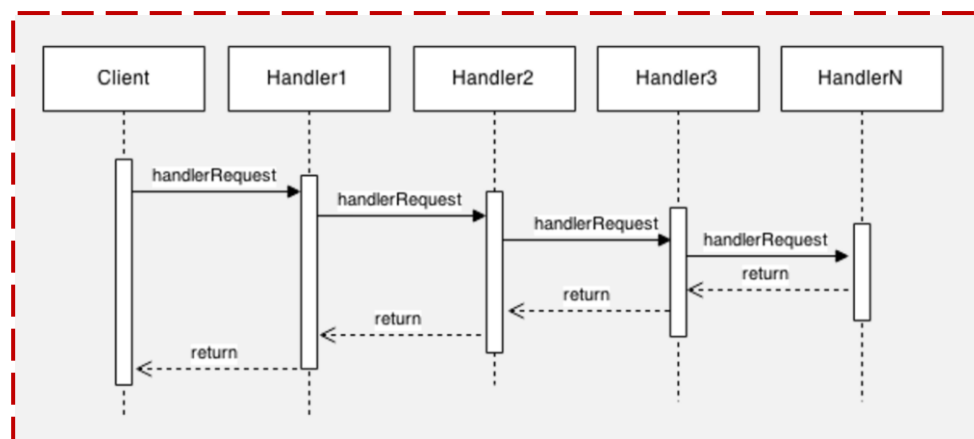
I *behavioural design patterns* realizzano pattern di comunicazione ricorrenti tra oggetti, dove gli algoritmi proposti consentono la separazione delle responsabilità tra gli oggetti stessi. I più utilizzati sono *Chain of Responsibility*, *Command*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Visitor*.

Chain of Responsibility Pattern

CoR: definisce come processare richieste, elaborandole lungo una catena di operazioni sequenziale performate dagli *handler*. Ogni *handler* può decidere in maniera “autonoma” se interrompere il flusso o passare al successivo l’informazione, sia originale che elaborata.

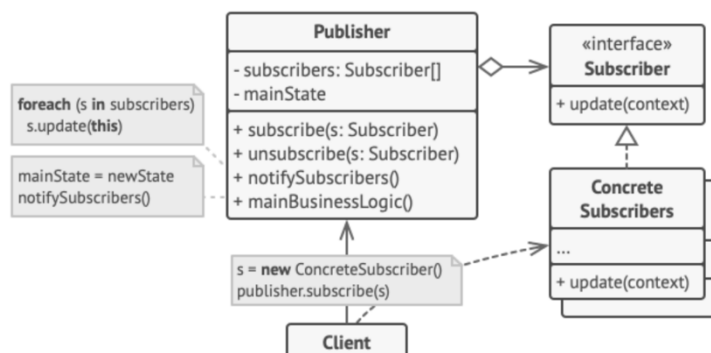
Utilizzo: un esempio di applicazione sono le catene di validazione, come nel login/autorizzazione.

Suggerimenti: per evitare di creare codice troppo complesso e disordinato, ogni controllo *handler* avrà un campo per memorizzare il riferimento al prossimo *handler* nella catena. Questo pattern va’ usato quando è necessario eseguire differenti *handler* in un certo ordine.



Observer Pattern

Observer: definisce un meccanismo di notifica dove più oggetti effettuano una sottoscrizione all’oggetto osservato, per questo motivo viene talvolta indicato come *Observer/Observable Pattern*.



L’oggetto che si vuole osservare è detto *subject*, mentre se è anche responsabile della notifica dei cambiamenti agli altri oggetti ci si riferirà ad esso come *publisher*; tutti quegli oggetti che tengono traccia dello stato del *publisher* sono chiamati *subscribers*.

Suggerimenti: è fondamentale che tutti i *subscribers* implementino la stessa interfaccia e che il *publisher* comunichi con loro solamente attraverso questa interfaccia.

Utilizzo: è necessario quando si vuole che al cambiamento dello stato di un oggetto, altri debbano essere modificati o cambiare stato a loro volta; oppure quando alcuni oggetti devono osservarne altri ma per un tempo limitato, o per casi specifici.

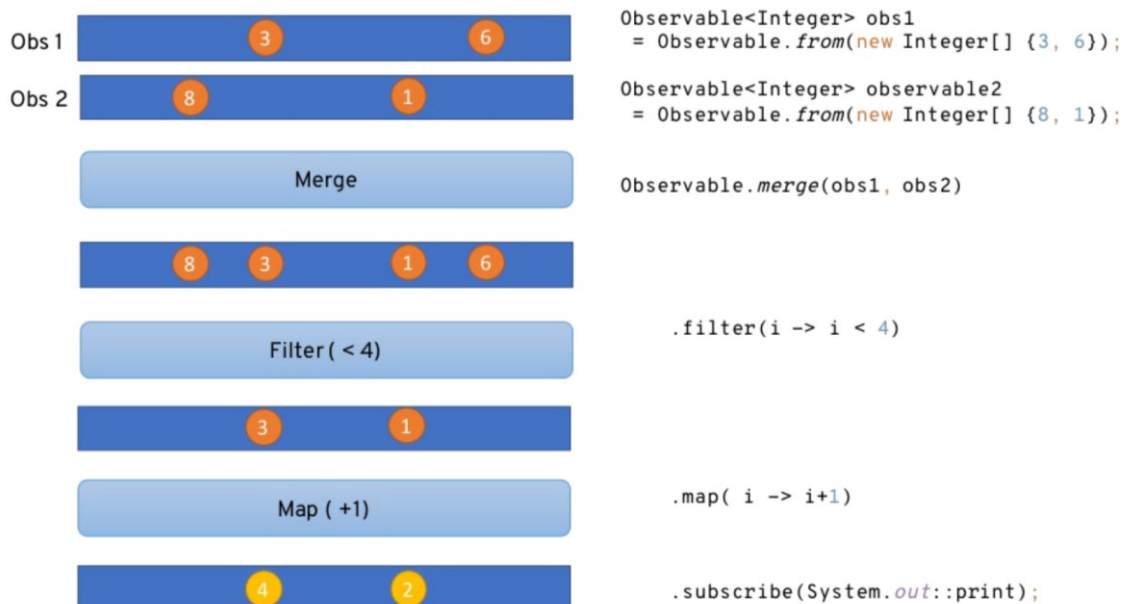
Vantaggi: si possono stabilire relazione tra oggetti a tempo di esecuzione, aggiungendo *subscribers* senza particolari problematiche per il *publisher*.

Svantaggi: i *subscribers* sono notificati in maniera randomica.

Programmazione reattiva: programmazione con stream di dati asincroni, il cui principale vantaggio è dato dalla possibilità di trasformare qualunque informazione in stream. Sarà necessario implementare una serie di funzioni per creare, combinare e filtrare gli stream; ciascuno dei quali potrà essere usato come input di un altro.

Lo stream è una sequenza di eventi in svolgimento sequenziale nel tempo, possono restituire un valore, un errore o un segnale di “fine elaborazione”.

Esempio di programmazione reattiva:



Mediator Pattern

Mediator: semplifica le dipendenze tra oggetti²⁰, costringendoli ad un'interazione per mezzo di un'apposito oggetto *mediatore*. Il *mediator* (incapsula le interconnessioni) controlla e coordina la comunicazione, aumentando notevolmente la flessibilità del sistema all'introduzione di modifiche.

Vantaggi: separazione delle responsabilità, si possono introdurre nuovi mediatori senza cambiare gli attuali componenti, maggiori possibilità di riutilizzo dei componenti.

Svantaggi: fare attenzione a non creare classi o oggetti "God" con troppa conoscenza/responsabilità, tener sempre presente che è meglio fare poche cose fatte bene.

²⁰ Risolve il problema di avere oggetti fortemente legati tra loro, situazione che andrebbe sempre evitata. In un certo senso, lavora come un *buffer*.

Reactive Programming

La programmazione reattiva viene utilizzata per creare e manipolare stream di dati/eventi in modo prevedibile, utilizzando funzioni come *map*, *filter* e *reduce* per il processamento dei flussi. Le modifiche introdotte saranno quindi propagate automaticamente in tutto il sistema.

Caratteristiche: un sistema *reattivo* sarà caratterizzato da *Responsiveness*, *Resilienza*, *Elasticità*, *Message-Driven*.

Approccio OOP: si trasforma un oggetto in una fonte di eventi (in modo simile ad observable) e si registra un *listener*; al verificarsi dell'evento il *listener* eseguirà il codice restituendo un output.

Approccio reattivo: si specificano le dipendenze dei dati in modo dichiarativo, dove il grosso del lavoro viene delegato a metodi funzionali, evitando di riscrivere la stessa logica.

Esempio. Un pulsante è uno **stream** di click.

Osservazione: in RxJS si creano stream osservabili (*Observable*) e li si processa con operatori funzionali.

```
// Unire più feed social e filtrare per hashtag
var messages = merge(twitter, fb, gplus, github);

messages
  .map(normalize)           // Uniforma il formato dei messaggi
  .filter(selectHashtag)    // Seleziona solo quelli con l'hashtag richiesto
  .pipe(process.stdout);    // Scrive lo stream in output su stdout
```

Le Promise

Promise: oggetto che incapsula un valore che potrebbe essere disponibile in futuro; in altre parole, “wrappa” un valore che potenzialmente sarà risolto più avanti.

Funzionamento: essenzialmente è uno stream di che emette un solo valore o errore. Possono essere sostituite dagli oggetti *observable*.

Reactive Extensions for JavaScript

RxJS: libreria per comporre programmi asincroni e basati su eventi usando *observable* e operatori di query. Gli sviluppatori rappresentano i flussi asincroni con *observable*, li interrogano con operatori funzionali e controllano la concorrenza tramite gli *scheduler*.

$$RxJS = Observables + Operators + Schedulers$$

- *Observable* → flussi invocabili di valori futuri o eventi.
- *Observer* → collezione di callback per ricevere valori.
- *Scheduler* → per gestire quando e come avviene l'esecuzione.
- *Subjects* → analoghi ad EventEmitter/Observable, per il multicasting.
- *Operators* → simili ai metodi degli array, come *map*, *filter*, *reduce*, ecc. possono essere *pipeable*²¹ o *creation operators*.

Perché usarla? unifica promise, callback e flussi di eventi, consentendo una ricca composizione di flussi asincroni e basati su eventi.

Programmazione Asincrona, Fetch e AJAX

Introduzione ad AJAX

Asynchronous JavaScript and XML (AJAX): insieme di tecniche di sviluppo web lato client per la creazione di applicazioni web asincrone.

- Le applicazioni dovranno mandare e ricevere dati dal server in modo asincrono (in *background*) senza interferire con il livello UI della pagina.
- AJAX consente alle pagine web di cambiare dinamicamente il contenuto senza ricaricare l'intera pagina.
- Viene solitamente usato il JSON per la comunicazione (meno frequente è l'XML).

Fetch API & Promises

Fetch: API utilizzata per caricare risorse esterne nel browser, caratterizzata dall'avere solamente la seguente funzione `fetch('images.txt');`. In output si avrà come oggetto di ritorno una *Promise*.

Promise: oggetto utilizzato nella gestione delle risposte asincrone, in particolare consentono la creazione di catene di risultati asincroni. Hanno un metodo `then(...)` che riceve in input le funzioni da eseguire in caso di successo e di errore.

Ora si preferisce utilizzare le promises rispetto alle callbacks (deprecate), dato che per eseguire azioni asincrone a catena vi era la necessità di annidare molteplici funzioni di callback²².

Q: How does this syntax work?

```
fetch('images.txt').then(onSuccess, onFail);
```

The syntax above is the same as:

```
const promise = fetch('images.txt');  
promise.then(onSuccess, onFail);
```

²² La situazione che si originerebbe da questo comportamento viene detta *callback's hell*.

Una *promise* può avere tre stati:

- *Pending* → stato iniziale, né completata né rifiutata.
- *Fulfilled* → operazione completata con successo.
- *Rejected* → operazione fallita.

Osservazione: all'interno di un browser, JavaScript non ha accesso ai file su disco, a differenza di *Node.js* che invece consente una gestione completa del filesystem.

Chaining promises: ritornare un valore da una callback in `.then()` crea una nuova promessa, che potrà essere utilizzata per incatenarne un'altra aggiungendo un ulteriore `.then()`.

JSON

JavaScript Object Notation (JSON): JSON è un formato standard per "serializzare" oggetti JavaScript, ovvero convertirli in una stringa per la trasmissione o l'archiviazione.

- `JSON.stringify(object)`: Converte un oggetto JavaScript in una stringa JSON.
- `JSON.parse(string)`: Converte una stringa JSON in un oggetto JavaScript.

REST APIs e HTTP

Web Services e API Endpoint: non tutte le URL puntano a un file statico. Spesso, un URL rappresenta un "API endpoint", cioè un punto di accesso a un servizio web. Invece di restituire un file, il server genera una risposta dinamica basata sulla richiesta.

API RESTful: è un'architettura per API basata su URL che utilizza i metodi standard del protocollo HTTP:

- **GET:** Per richiedere dati.
- **POST:** Per inviare/sottomettere dati.
- **PUT:** Per caricare un file.
- **PATCH:** Per aggiornare dati esistenti.
- **DELETE:** Per cancellare dati.

CORS (Cross-Origin Resource Sharing): per motivi di sicurezza, i browser applicano una policy che limita le richieste `fetch()` a domini diversi da quello della pagina di origine (richieste "cross-origin"). Un server può essere configurato per permettere queste richieste, come fanno tutte le API di terze parti (es. Google, Spotify, etc.).

Creare e Gestire Promises Avanzate

Creare una Promise: usando il costruttore `new Promise(executor)`. L'executor è una funzione che contiene il codice asincrono e accetta due argomenti:

- `resolve(value)`: Funzione da chiamare quando l'operazione ha successo.

- `reject(error)`: Funzione da chiamare in caso di errore.

Sintassi `async/await`: `async/await` è una sintassi speciale che permette di lavorare con le Promises in un modo che appare sincrono, rendendo il codice più pulito e leggibile.

- **`async`:** Una funzione dichiarata con `async` restituisce sempre una Promise. Se si restituisce un valore non-Promise, questo viene automaticamente "avvolto" in una Promise risolta.
- **`await`:** L'operatore `await` può essere usato solo all'interno di una funzione `async`. Mette in pausa l'esecuzione della funzione fino a quando la Promise non è risolta, e il suo valore di ritorno è il risultato della Promise stessa. Durante l'attesa, il browser non è bloccato e può continuare a eseguire altro codice.

Suggerimenti Per Progetto & Esame

Progetto

ToDo: alcune operazioni da fare prima, durante e dopo l'esecuzione del progetto.

- 1) **Creare un template:** iniziare a definire una struttura in linea di massima del progetto, concentrandosi sulle directory e meccanismi/comandi per avvio/debug/compilazione. Fare tutto in due settimane altrimenti è impossibile.
- 2) **Crittografia:** non usare chiavi pubbliche (crittografia *simmetrica*) ma chiavi sia pubbliche che private, cioè crittografia *asimmetrica*.
- 3) **Copia file remoti:** non chiaro il motivo, ma se si deve copiare un file su macchina remota usare winscp che almeno ha l'interfaccia grafica.

Payload: il payload del JWT non è il *payload* della richiesta, su quello del token dovranno essere messi in chiaro i dati necessari per capire chi effettua la richiesta, mentre sul contenuto della richiesta ci potrà essere qualunque cosa ma criptata.

Audience: attributo inserito nel *payload* del token, che consente di specificare l'applicazione sulla quale può essere utilizzato. Sistema attualmente utilizzato nei sistemi che utilizzano i token, in particolare nei JWT.

BullMQ: gestore di code che potrebbe essere utilizzato per la realizzazione del progetto.

Per la parte di database si utilizzerà *sequelize*.

Relazioni

Per la stesura della tesi si consiglia di utilizzare google scholar, particolarmente utile per ricercare articoli scientifici, citazioni e altri documenti (e di conseguenza fondamentale per la bibliografia).

Tips: nella progettazione dell'architettura software si deve sempre utilizzare un'approccio *top-down*, per evitare di definire API più complesse del necessario. Un modo utile per visualizzare i flussi di dati di un applicazione o farne il refactoring, può essere utilizzato il BPMN (notazione simile all'*activity diagram*).

Errori comuni

Prima di consegnare il codice del progetto controllare la formattazione con il *linter*!!!!

NON RICREARE GLI STATUS CODES NEL PROGETTO, MA UTILIZZARE UNA LIBRERIA, AD ESEMPIO HTTP-STATUS-CODES.

Attenzione (API e Token): non lasciare mai nel codice, o addirittura caricare su github, le API Key → il prof lo considererà come errore grave. Per la realizzazione del progetto si utilizzeranno i token JWT.

