Appunti programmazione avanzata

Programmazione Avanzata (Università degli Studi di Camerino)

Programmazione avanzata

Paradigmi di programmazione

Un paradigma è un modo per classificare i linguaggi di programmazione.
Definisce un approccio alla programmazione basato su una teoria matematica o un insieme di principi. Ogni paradigma si basa quindi su concetti che lo rendono adeguato per un certo problema.

Tipi di paradigmi

- **IMPERATIVO / PROCEDURALE**: descrive computazioni "chiuse" (es. sistemi operativi). Le istruzioni sembrano **ordini** impartiti alla macchina virtuale e spesso scritte con verbi all'imperativo. Seguono uno stile prescrittivo, ovvero:
  - eseguono istruzioni in maniera sequenziale;
  - utilizzano procedure per l'organizzazione del codice.
- **FUNZIONALE**: si basa su funzioni, i calcoli avvengono attraverso queste. Il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche. Il punto di forza principale di questo paradigma è la mancanza di effetti collaterali (*side-effect*) delle funzioni, il che comporta una più facile verifica della correttezza e della mancanza di bug del programma e la possibilità di una maggiore ottimizzazione dello stesso. La computazione avviene principalmente nello *stack* e non sono presenti variabili.
- **DICHIARATIVO / LOGICO**: descrive a cosa una certa entità assomiglia (proprietà dei dati), piuttosto che prescrivere come un'entità può essere creata. Ad esempio le pagine web HTML sono dichiarative, perché descrivono cosa la pagina dovrebbe contenere. Programmi e strutture dati non sono separati, ma coesistono nella medesima formalizzazione.
- **ORIENTATO AGLI OGGETTI**: permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi. Particolarmente adatta nei contesti in cui si possono definire delle relazioni di interdipendenza tra i concetti da modellare.

Definizione

OGGETTO: elemento della computazione che contiene dati memorizzati attraverso i campi dell'oggetto e codici attraverso dei metodi.
La computazione di un programma si realizza attraverso l'interazione tra vari oggetti.

Programmazione funzionale

Un programma consiste in una composizione di funzioni, che sono l'elemento base della computazione. La programmazione funzionale si basa sulla definizione di un insieme di funzioni.

Nella programmazione funzionale le funzioni sono valori di primo livello (una funzione restituisce una funzione) dette ***first-class***, ovvero un valore che può essere trattato come qualsiasi altro valore.
In pratica, ciò significa che il valore può essere:

- assegnato a una variabile o una struttura dati;
- passato come parametro ad una funzione;
- restituito come valore di ritorno da una funzione;
- creato dinamicamente durante l'esecuzione del programma.

In pratica, questo significa che le funzioni possono essere trattate esattamente come qualsiasi altro valore all'interno del linguaggio.

N.B.
Più il codice è semplice, più è "meglio" *(Simpler is better).* Massimo 10/15 righe di codice per funzione comprese le parentesi e la firma del metodo.

N.B.
La presenza di valori first-class è spesso considerata una caratteristica importante dei linguaggi di programmazione funzionale, poiché consente di scrivere codice più espressivo e flessibile.

Richiami di matematica:

DEF. **INSIEME**: concetto primitivo di collezione di elementi che possono essere indicati:
-definendoli ----> {1,2,3,4,5,6,…}
-usando insiemi noti ----> N , R , Q , Z
-descrivendo proprietà ----> {x | x mod 2 = 0} (numeri pari)

DEF. **RELAZIONE BINARIA**: sottoinsieme del prodotto cartesiano A x B
Proprietà:
-Riflessiva: ogni elemento è in relazione con se stesso
-Simmetrica: *(x,y)* ∈ R ==> *(y,x)* ∈ R
-Anti-simmetrica: *(x,y)* ∈ R ^ *(y,x)* ∈ R ==> *x = y*
-Transitiva: *(x,y)* ∈ R ^ *(y,z)* ∈ R ==> *(x,z)* ∈ R

DEF. **FUNZIONE**: particolare relazione che associa ad ogni elemento del dominio al più un elemento del codominio. (in programmazione funzionale indica un tipo)
-***iniettiva***: elementi distinti del dominio hanno immagini distinte
-***suriettiva***: ogni elemento del codominio è immagine di almeno un elemento del dominio
-***biettiva***: se è sia iniettiva che suriettiva

F#

Il linguaggio **F#** è un linguaggio **fortemente tipizzato**, ovvero ogni tipo di dato (come intero, carattere, esadecimale, decimale compresso e così via) è predefinito come parte del linguaggio di programmazione e tutte le costanti o variabili definite per un dato programma devono essere descritte con uno dei tipi di dati. In un linguaggio di programmazione fortemente tipizzato, il compilatore esegue un controllo rigoroso sul tipo di ogni variabile, costante, espressione e operazione, in modo da garantire che siano sempre utilizzati in modo coerente.

I tipi base del linguaggio **F#** sono *bool, int, float, char, string*.

In particolare **F#** è **tipizzato staticamente**, ovvero il tipo della variabile viene risolto a tempo di compilazione, e non a tempo di esecuzione. Ciò significa che il compilatore verifica se le operazioni eseguite sulle variabili sono compatibili con il tipo della variabile e segnala eventuali errori di tipo prima che il programma venga eseguito.

In un sistema di tipi statico, ***il tipo di una variabile viene specificato esplicitamente*** dal programmatore e verificato dal compilatore durante la compilazione del codice sorgente. Ciò significa che ogni variabile ha un tipo specifico e non può cambiare durante l'esecuzione del programma.

Algoritmo di typing

L'***algoritmo di typing*** è il processo attraverso il quale il compilatore determina il tipo di ogni espressione e dichiarazione in un programma, in modo esplicito e senza inferenza. In altre parole, con l'algoritmo di typing, il programmatore deve specificare in modo esplicito i tipi delle variabili e delle espressioni nel codice. Esso verifica che tutto in fase di compilazione non dia errore di tipo.

```
let x : int = 5        //ha tipo int
let y : float = 3.14   //ha tipo float
let z : float = x + y //operazione non consetita
                       //Errore di tipo!
```

In questo esempio, il programmatore specifica esplicitamente il tipo di ogni variabile utilizzando la sintassi "**: tipo**". In particolare, **x** è di tipo *int*, **y** è di tipo *float* e **z** è di tipo *float*. In questo caso, la somma tra un intero e un float non è un'operazione consentita in F#, quindi il compilatore segnalerà un errore di tipo.

L'algoritmo di typing in **F#** verifica che ogni espressione nel programma sia compatibile con il tipo dichiarato per le variabili.

L'algoritmo di typing è particolarmente utile quando si vuole garantire che il codice sia corretto a livello di tipo, perché obbliga il programmatore a specificare esplicitamente il tipo di ogni variabile. Ciò può rendere il codice più sicuro e meno incline a errori di tipo.

Algoritmo di type inference

L'***algoritmo di type inference*** in F# è un processo automatico in cui il compilatore deduce il tipo di una variabile o di un'espressione in base al contesto in cui viene utilizzata. In questo caso, il programmatore ***non deve specificare in modo esplicito i tipi delle variabili*** e delle espressioni nel codice. Il compilatore analizza il codice sorgente per creare un grafo dei tipi e cerca di trovare un tipo comune per tutte le espressioni presenti in una data espressione. Una volta che il tipo di tutte le espressioni è stato dedotto, il compilatore genera il codice macchina corrispondente.

L'algoritmo di type inference **funziona in modo ricorsivo**, partendo dalle espressioni più semplici e procedendo verso quelle più complesse. Inizialmente, ogni espressione viene considerata come avente un tipo generico, ad esempio *a*. L'algoritmo quindi cerca di dedurre il tipo di ogni espressione basandosi sul tipo delle espressioni circostanti.

Utilizza un **sistema di equazioni di tipi** per inferire i tipi delle espressioni. Ad esempio, se abbiamo un'operazione di addizione tra due espressioni di tipo *int*, l'algoritmo dedurrà che il tipo dell'espressione risultante sarà anch'esso *int*.

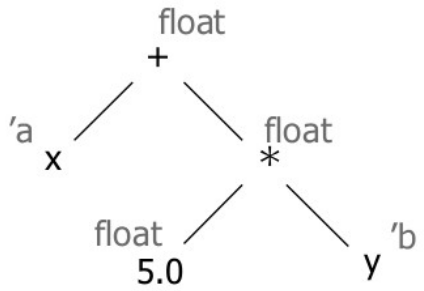
In generale, l'algoritmo cercherà di risolvere le equazioni di tipi per trovare una soluzione consistente e completa per tutti i tipi presenti nel programma.

Se l'algoritmo non riesce a inferire il tipo di un'espressione, ciò significa che il programma contiene un errore di tipo. In tal caso, il compilatore o l'interprete genererà un errore di tipo.

1. Data un'espressione;
2. Ed una tabella in cui viene associato ad ogni nome definito un tipo;
3. Consideriamo l'albero sintattico dell'espressione...
   - le foglie sono valori o nomi
   - i nodi interni sono operatori
4. Assegnamo un tipo ad ogni foglia
5. Assegnamo un tipo ai nodi a partire dai tipi dei figli
6. Il tipo della radice è il tipo dell'espressione

```
let aFunction (x, y) = x+(5.0*y)
```

$\Gamma = \{x: \text{float}, y: \text{float}\}$



Le funzioni

Le funzioni sono valori **first-class** e possono essere associate con i nomi come ogni altro tipo di dato. Sono solitamente definite come espressioni, cioè un pezzo di codice che restituisce un valore.

```
let f1 = fun x → x+1
let f2(x) = x+1

let f k = k*k
let f3(x, f) = f(x+2)+1
let y = f3(1, f)
```

N.B.

Le funzioni ***first-class*** sono funzioni che possono essere trattate come qualsiasi altro valore, come ad esempio una variabile o una costante. Ciò significa che le funzioni possono essere passate come argomenti ad altre funzioni, restituite come valori di funzione da altre funzioni e assegnate a variabili.

Le funzioni in un linguaggio funzionale possono anche essere definite come composizione di altre funzioni. Non hanno effetti collaterali, il che significa che non modificano lo stato del programma o dell'ambiente circostante.

Esempio

Considera la seguente funzione ***add*** che prende due argomenti ***a*** e ***b*** e restituisce la loro somma.

```
let add a b = a + b

val add : a: ^a → b: ^b →  ^c
    when ( ^a or  ^b) : (static member ( + ) :  ^a *  ^b →  ^c)
```

In questo caso, F# utilizza l'algoritmo di type inference per determinare il tipo della funzione *add*. Poiché la funzione utilizza l'operatore +, F# inferisce che *a* e *b* devono essere di tipo numerico, ad esempio *int*, *float* o *decimal*. Di conseguenza, F# determina che *^a*, *^b* e *^c* sono variabili di tipo che rappresentano i tipi di *a*, *b* e il valore di ritorno della funzione, rispettivamente. F# utilizza il simbolo □ per rappresentare il tipo di una funzione. L'algoritmo di type inference in **F#** è in grado di inferire i tipi delle funzioni anche in presenza di funzioni generiche e funzioni lambda.

```
//Nessuna annotazione ------------------------> inferito tipo int a f
let f(x,y) = x + y
//Il parametro x è annotato come float -------> inferito tipo float a f
let f(x: float, y) = x + y
//Il nome c è annotato come float ------------> inferito tipo float a f
let f(x, y) = (x: float) + y
//Il tipo di ritorno è annotato come float ---> inferito il tipo float a f
let f(x, y): float = x + y
```

Funzioni ricorsive

L'uso delle funzioni ricorsive è cruciale nella programmazione funzionale.

Una funzione ricorsiva è una funzione che si chiama da sola, generalmente con argomenti diversi, fino a raggiungere un caso base che interrompe la ricorsione.

N.B.
L'uso del *rec* ha un costo in quanto l'inferire il tipo di una funzione ricorsiva ha un costo esponenziale

Le funzioni ricorsive vengono definite utilizzando la parola chiave ***rec*** dopo la parola chiave ***let***.

```
let fib(x)=
    let rec _fib(x)=
        if x⩽2 then
            (1,1)
        else
            let(a,b)=_fib(x-1)
            in(a+b, a)
    in
        let(a,_)=_fib(x)
        in
            a
```

```
let rec _prime a b =
    if (a / b+1 = 1)
        then "FALSE"
    else "TRUE"

let prime a =
    if (a < 3)
        then "TRUE"
    else _prime a 0
```

Es.
Scrivere una funzione che ricevuto un intero ***n*** restituisce ***true*** se n è un numero primo, ***false*** altrimenti.

Valutazione parziale

La valutazione parziale, è una tecnica utilizzata in molti linguaggi di programmazione funzionale per creare nuove funzioni a partire da funzioni esistenti applicando solo alcuni dei loro argomenti.

In altre parole, la valutazione parziale consente di fissare uno o più argomenti di una funzione, creando così una nuova funzione che richiede solo gli argomenti rimanenti per essere completamente valutata.

Es. La funzione *add* prende due argomenti di tipo *int* e restituisce la loro somma.

```
let add x y = x + y

let addFive = add 5

// Utilizzo della funzione addFive
let result = addFive 3 // Restituisce 8
```

La funzione *addFive* è una nuova funzione creata con la valutazione parziale di *add* con il primo argomento fissato a 5. Quando la funzione *addFive* viene chiamata con un secondo argomento, viene applicata la funzione *add* completa con l'argomento 5 fissato e il secondo argomento passato alla chiamata.

In parole povere, la funzione *addFive* sarà [*add 5 3*]. Se avessimo usato 18 al posto di 3 sarebbe [*add 5 18*].

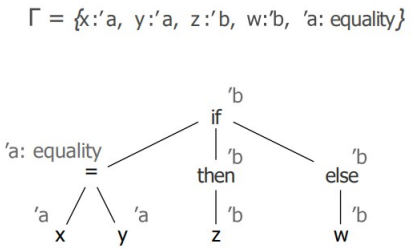
Parametri di tipo

Il tipo inferito per la funzione (che diventa una funzione polimorfa) è il seguente:

Consideriamo la seguente definizione:

```
let select x y z w = if (x=y) then z else w
```

N.B.
ogni operatore ha una propria regola per l'assegnazione del tipo



Nonostante il compilatore non abbia sufficienti informazioni per inferire il tipo dei parametri x, y, z e w, riesce comunque a determinare che:

- x e y hanno lo stesso tipo (che chiamiamo 'a);
- z e w hanno lo stesso tipo (che chiamiamo 'b);
- il tipo 'a supporta l'operatore di confronto (equality).

Quindi, ogni tipo che soddisfi queste proprietà può essere utilizzato al posto di *'a* e *'b* che vengono considerati **parametri di tipo**

Viene inferito il tipo *'a* ad *x*. Tramite l'operatore *equality (=)* viene inferito lo stesso tipo anche ad *y*.
Non avendo informazioni per inferire un tipo a *z* e *w* viene assegnato loro il tipo *'b*. Per cui l'algoritmo deduce che la funzione *select* restituirà il tipo *'b*.

```
val select: x:
    'a⟶y: 'a⟶z: 'b⟶w: 'b⟶'b
        when 'a: equality
```

Tipi aggregati e generici

I **tipi aggregati** rappresentano tipi di dati immutabili che possono contenere uno o più valori di dati e includono record, tuple e elenchi.
I **tipi generici** sono tipi di dati *parametrici* che possono essere istanziati con diversi tipi di dati. I tipi generici sono utili quando si scrivono funzioni o classi che lavorano con tipi di dati diversi, ma seguono lo stesso schema o comportamento.

Tuple

Le tuple sono un tipo di dato comune nei linguaggi funzionali e sono utilizzate per raggruppare insieme un numero arbitrario di valori di diversi tipi. In una tupla, ogni valore è identificato da una posizione o un indice, simile a un array.

```
let myTuple : int * string = (42, "hello")

let myTuple2: int * int * float * char = (4, 5, 2.3, 'A')
```

In F#, le tuple possono essere definite in modo implicito. Il tipo di una tupla viene definito utilizzando una sintassi simile a quella di una lista di tipi separati da un asterisco *.
Le tuple sono immutabili, il che significa che una volta create non possono essere modificate. Ciò le rende utili in situazioni in cui è necessario rappresentare un insieme di valori correlati in modo sicuro e *thread-safe*.

Liste

Le **liste** sono un tipo di dati immutabile e ordinati che possono contenere un numero variabile di valori dello stesso tipo. Le liste sono rappresentate utilizzando la sintassi **[ ]** e ogni elemento della lista è separato da un punto e virgola.

```
let intList = [1; 2; 3] //tipo int
let emptyList = [ ]     //tipo 'a list (vuota)'

let myMixedList = [1; "ciao"; true]
let myExplicitList : string list = ["ciao"; "mondo"]
```

Il tipo di una lista viene inferito in base ai tipi degli elementi contenuti nella lista stessa. Ad una lista contenente elementi di tipi diversi, il compilatore inferirà il tipo *list<'a>* dove *'a* rappresenta un tipo generico che può essere sostituito con il tipo specifico degli elementi contenuti nella lista.
È anche possibile specificare esplicitamente il tipo di una lista utilizzando la sintassi *<tipo> list* .

Gli **operatori** più comuni nelle liste sono:

- **::** > aggiunge un elemento all'inizio di una lista;
- **@** > concatenare due liste;
- **..** > permette di creare sequenza di numeri interi con un incremento implicito di 1;
- **...** > utilizzato per creare una sequenza con un incremento specifico.

```
let myList = 1 :: 2 :: 3 :: [4; 5; 6]   // myList = [1; 2; 3; 4; 5; 6]

let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
let myList = list1 @ list2              // myList = [1; 2; 3; 4; 5; 6]

let myRange = 1..5                      // myRange = [1; 2; 3; 4; 5]

let myRange2 = 0..2...10                // myRange2 = [0; 2; 4; 6; 8; 10]
```

Pattern matching

Il **pattern matching** è una funzionalità che permette di confrontare il valore di una variabile con uno o più pattern e di eseguire azioni diverse in base al pattern corrispondente. In altre parole, il pattern matching è un modo per gestire i flussi di controllo del programma in modo più espressivo e flessibile.

Può essere utilizzato con variabili di qualsiasi tipo, inclusi i tipi di base come *numeri*, *stringhe* e *booleani*, ma anche con i tipi complessi come le *liste*, le *tuple* e gli *union type*.

Il pattern matching è spesso utilizzato con le espressioni ***match***, che sono costrutti sintattici utilizzati per specificare i pattern e le azioni da eseguire in base al pattern corrispondente.

```
match value with
| pattern1 → action1
| pattern2 → action2
| pattern3 → action3
...
| _ → defaultAction
```

Il pattern matching può essere utilizzato per gestire una vasta gamma di situazioni, tra cui la manipolazione di liste e array, l'estrazione dei valori da tuple, il riconoscimento di schemi di dati complessi e la gestione degli errori.
L'uso di condizioni (espressioni booleane) può essere utilizzato per limitare o gestire la selezione.

Esempio: Valutazione di polinomi

Un polinomio può essere rappresentato come una lista di coefficienti.

Supponiamo di dover scrivere una funzione che valuti il risultato di un polinomio dati i suoi coefficienti. Sarà del tipo:

```
let rec eval cList (x: float) =          //cList è la lista dei coefficienti e
    match cList with                     //x è l'incognita.
    |[] → 0.0                            //Se la lista è vuota restituisce 0.0;
    |[c] → c                             //se la lista di coefficienti contiene un solo
                                         //elemento, restituisce il coefficiente.
    |c::tail → c+(x*(eval tail x))       //il valore della funzione è il primo coefficiente
                                         //sommato al prodotto di "x" per la valutazione del
                                         //resto della lista di coefficienti.

let coefficients = [2.0; 3.0; -4.0; 1.0]
let x = 2.0
let result = eval coefficients x

printfn "Il valore del polinomio per x=%f è %f" x result
```

> Il valore del polinomio per x=2.0 è 0.0

Tipo opzione

Il tipo **option** è utilizzato per gestire i valori che potrebbero essere nulli o che potrebbero non esistere. *Option* è un tipo generico che può essere utilizzato con qualsiasi tipo di valore, inclusi i tipi di base come numeri e stringhe, ma anche con i tipi complessi come le tuple e le liste.

Option rappresenta due possibili valori: ***Some*** e ***None***. Il valore *Some* rappresenta un valore esistente, mentre il valore *None* rappresenta l'assenza di un valore

```
let rec findFirstMatching pred x =
    match x with
    |[] → None
    |v::tail → if (pred v)
        then Some v
        else findFirstMatching pred tail
```

N.B.
Il parametro *pred* indica un predicato, ovvero una funzione che prende valori di un certo tipo e mi restituisce *true* o *false*.
La funzione avrà tipo:

```
pred:('a → bool) → l:'a list → 'a option
```

Unione discriminate

Le unioni discriminanti sono un costrutto che permette di definire un tipo di dato composto da uno o più elementi, dove ogni elemento ha un nome (o una "etichetta") associato che lo identifica univocamente. Utilizzano il pattern matching per gestire i diversi casi.

```
type Shape =
    | Rectangle of width : float * length : float
    | Circle of radius : float
    | Prism of width : float * float * height : float
```

Es.
Shape identifica un interfaccia; *Rectangle, Circle e Prism* identificano le proprietà della classe

Le unioni discriminanti sono molto utili per la modellizzazione di tipi di dati complessi, dove è possibile avere diversi casi con parametri associati. Grazie al pattern matching, è possibile scrivere codice più robusto e leggibile per gestire i diversi casi del tipo di dato.

Binary Search Tree (albero di ricerca binario)

I **BST** sono rappresentati come tipi di dati algebrici costituiti da un'opzione vuota o da un nodo (*Node*) che contiene un valore e due sotto-alberi sinistro e destro, a loro volta BST.

```
type 'a BST =
    | Empty
    | Node of value: 'a * left: 'a BST * right: 'a BST
```

Qui, *left : 'a BST e right: 'a BST* rappresentano rispettivamente il sotto-albero destro e sinistro dell'albero. Tutte le operazioni possibili sugli alberi (ricerca, rimozione, inserimento di un elemento e unione di due alberi) possono essere implementate.

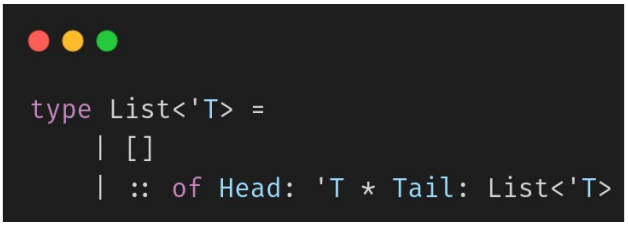
Moduli

Un **modulo** è un'entità che raggruppa funzioni, tipi, costanti e altre definizioni correlate, consentendo di organizzare e suddividere il codice in modo logico e modulare. I moduli in F# sono implementati come file sorgente separati o come parti di un unico file sorgente.

Un modulo viene dichiarato utilizzando la parola chiave *module* seguita dal nome del modulo.

Le definizioni all'interno di un modulo possono essere organizzate ulteriormente in sotto-moduli, utilizzando la sintassi *module* annidata.

List



Il modulo **List** fornisce funzioni e tipi per lavorare con le liste. Il tipo di lista in F# è una lista concatenata di elementi dello stesso tipo. Il tipo List è un tipo generico, dove il parametro *'T* rappresenta il tipo degli elementi della lista. La definizione utilizza la sintassi delle unioni discriminanti per rappresentare una lista come una serie di elementi concatenati tra loro.

Il modulo List mette a disposizioni diverse funzioni:

- **average**: calcola la lunghezza della lista come un valore in virgola mobile e restituisce la somma divisa per la lunghezza per ottenere la media aritmetica. Restituisce sempre un valore di tipo *float*, che rappresenta la media aritmetica dei valori nella lista, indipendentemente dal tipo di input;
- **averageBy**: prende una funzione di proiezione e una sequenza di valori, quindi applica la funzione di proiezione a ogni valore e restituisce la media aritmetica dei risultati. Una **funzione di proiezione** è una funzione che prende un valore di tipo *'T* e restituisce un valore di tipo *'U* e viene applicata a ciascun valore nella sequenza di input;
- **filter**: restituisce una nuova lista contenente solo gli elementi della lista che soddisfano il predicato dato. Un **predicato** è una funzione che prende un valore di tipo *'T* e restituisce un valore booleano che indica se il valore soddisfa il predicato;
- **map**: crea una lista i cui elementi sono ottenuti applicando la funzione data (***mapping***) ad ogni elemento della lista. Una funzione *mapping* è una funzione che prende un valore di tipo *'T* e restituisce un valore di tipo *'U*, dove *'T* e *'U* possono essere di tipi diversi;
- **reduce**: applica una funzione (*folder*) ad ogni elemento della lista per calcolare un unico valore. Una **funzione folder** è una funzione di aggregazione che prende due argomenti: uno stato accumulatore di tipo *State* e un valore di tipo *`T*, e restituisce un nuovo stato accumulatore di tipo *State*.

```fsharp
//sintassi ed esempio di average
val average : float list → float
let numbers = [1.0; 2.0; 3.0; 4.0; 5.0]
let avg = List.average numbers // restituisce 3.0

//sintassi ed esempio di averageBy
val averageBy : projection:('T → 'U) → source:seq<'T> → float
let strings = ["apple"; "banana"; "cherry"; "date"]
let averageLength = List.averageBy (fun s → float (String.length s)) strings // restituisce 5.0

//sintassi ed esempio di filter
val filter : predicate:('T → bool) → source:seq<'T> → seq<'T>
let numbers = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let evenNumbers = List.filter (fun x → x % 2 = 0) numbers // restituisce [2; 4; 6; 8; 10]

//sintassi ed esempio di map
val map : mapping:('T → 'U) → source:seq<'T> → seq<'U>
let numbers = [1; 2; 3; 4; 5]
let doubledNumbers = List.map (fun x → x * 2) numbers // restituisce [2; 4; 6; 8; 10]

//sintassi ed esempio di reduce
val reduce : folder:('State → 'T → 'State) → state:'State → source:seq<'T> → 'State
let numbers = [1; 2; 3; 4; 5]
let sum = List.reduce (fun acc x → acc + x) 0 numbers // restituisce 15
```

Filter-Map-Reduce

E' un pattern di programmazione per l'elaborazione di una grossa mole di dati. Utile quando si lavora con grandi quantità di dati o quando si vuole eseguire elaborazioni complesse su una sequenza di valori. Gli elementi del dataset non sono elaborati in isolamento ma come parti di un gruppo. Consiste di tre funzioni principali:

- **Filter**: utilizzata per selezionare solo gli elementi della sequenza che soddisfano una determinata condizione;
- **Map**: utilizzata per trasformare ogni elemento della sequenza filtrata in un nuovo valore;
- **Reduce**: utilizzata per combinare tutti i valori della sequenza mappata in un singolo risultato.

Programmazione ad oggetti

La **programmazione ad oggetti** è un paradigma di programmazione che si basa sulla creazione di oggetti, che sono entità software che racchiudono dati e comportamenti associati.

La programmazione ad oggetti si concentra sulla **modularità** e sulla **riusabilità** del codice, poiché gli oggetti possono essere utilizzati in diversi contesti e possono essere combinati per creare applicazioni complesse. Inoltre, offre anche il concetto di **incapsulamento**, che consente di proteggere i dati interni degli oggetti e di rendere più facile la **manutenzione** e la **modifica** del codice.

I concetti base sono quindi **CLASSI** e **OGGETTI**

Un oggetto è una istanza di una classe, che è un *modello astratto* che definisce le proprietà e i comportamenti di un insieme di oggetti correlati. Le proprietà di una classe sono rappresentate dalle variabili di istanza, mentre i comportamenti sono rappresentati dai metodi. Sono composte da:

- **attributi**: definiscono le caratteristiche o proprietà di una classe;
- **metodi**: definiscono le operazioni fornite dalla classe.

Principi della OOP (Object-Oriented Programming)

1. Astrazione

Il principio di astrazione afferma che i dettagli di implementazione di un oggetto dovrebbero essere nascosti all'esterno e che solo le informazioni essenziali dovrebbero essere esposte.

In altre parole, un oggetto dovrebbe presentarsi all'esterno come una singola entità coesa e facilmente utilizzabile, senza rivelare i suoi dettagli interni o il funzionamento interno.

L'astrazione è importante perché consente di gestire la complessità del codice e di isolare gli effetti delle modifiche ai dettagli di implementazione. In questo modo, si può mantenere il codice più semplice e più facile da leggere, scrivere e mantenere nel tempo.

In Java l'astrazione avviene tramite **interfacce** e **classi astratte**.

N.B.
L'interfaccia è la prima cosa che va scritta quando si crea un'applicazione

2. Ereditarietà

Il principio dell'ereditarietà afferma che una classe può ereditare le proprietà e i metodi da una **classe padre** (o classe **base**) e che la **classe figlia** (o classe **derivata**) può estendere o modificare il comportamento della classe padre.

Una classe figlia può utilizzare i metodi e le proprietà della classe padre senza doverli re-implementare, semplificando la scrittura del codice e aumentando la modularità e la riutilizzabile. La classe figlia può anche aggiungere nuove proprietà e metodi o sovrascrivere quelli ereditati per personalizzare il proprio comportamento.

Esistono due forme di ereditarietà: **singola** e **multipla**. In *Java* l'ereditarietà singola si usa principalmente per le classi, mentre l'ereditarietà multipla per le interfacce.

L'ereditarietà è importante perché consente di creare classi gerarchiche e di organizzare il codice in modo più efficiente.

3. Polimorfismo

Il principio di polimorfismo afferma che gli oggetti di classi diverse possono essere trattati in modo uniforme, come se fossero dello stesso tipo, purché implementino lo stesso insieme di metodi o di proprietà.

In pratica, questo significa che si può scrivere del codice che lavora con oggetti di una determinata classe generica, e che funzionerà correttamente anche con oggetti di classi derivate o di altre classi che implementano la stessa interfaccia. Questo consente di scrivere codice più generale e flessibile, che può essere utilizzato con oggetti di diverse classi senza dover scrivere codice specifico per ciascuna classe.
La classe derivata adatterà il proprio comportamento in base ai metodi che fornisce.

4. Incapsulamento

Il principio di incapsulamento afferma che gli oggetti dovrebbero nascondere i loro dettagli di implementazione e permettere l'accesso solo attraverso un'interfaccia pubblica ben definita.

N.B.
I campi di un oggetto devono essere sempre privati. Se non lo sono ci deve essere una buona ragione.

In pratica, questo significa che le proprietà e i metodi interni di un oggetto dovrebbero essere protetti da accessi esterni e che solo le informazioni essenziali dovrebbero essere esposte attraverso l'interfaccia pubblica. Ciò consente di garantire che gli oggetti vengano utilizzati in modo corretto e coerente, e di evitare modifiche indesiderate ai loro dati interni.

Per rispettare il principio di incapsulamento in programmazione ad oggetti, è necessario seguire alcune linee guida:

1. **definire l'interfaccia pubblica** dell'oggetto;
2. dichiarare le **proprietà** come **private o protette;**
3. **implementare i metodi pubblici in modo robusto;**
4. **evitare di esporre dettagli di implementazione.**

L'incapsulamento è importante perché consente di migliorare la sicurezza e la robustezza del codice, evitando accessi non autorizzati e minimizzando gli effetti delle modifiche ai dettagli di implementazione.

Concetti base

- **Metodi**: Quando viene dichiarato un metodo dobbiamo definire **nome**, **tipo e nome dei suoi parametri**, **tipo di ritorno**. Poi va definito il corpo e quindi le varie operazioni che il metodo andrà ad eseguire.
- **Keyword *this***: è una parola chiave riservata che rappresenta un riferimento all'oggetto corrente all'interno di un metodo o di una funzione di un oggetto. Quando un metodo o una funzione viene chiamato su un oggetto, il riferimento *this* viene automaticamente impostato con il valore dell'oggetto stesso. In questo modo, è possibile accedere alle proprietà e ai metodi dell'oggetto all'interno del metodo o della funzione.
- **Costruttore**: è simile ad un metodo ed ha **lo stesso nome della classe** e **non c'è tipo di ritorno**. Un costruttore è una funzione speciale utilizzata per creare e inizializzare un nuovo oggetto di una classe. Il costruttore ha il compito di assegnare i valori iniziali alle proprietà dell'oggetto e può essere personalizzato per accettare parametri di input per personalizzare l'oggetto creato. Se la classe non ha costruttori, gliene viene assegnato uno di default che non ha argomenti ed inizializza le variabili con valori di default.
  Possiamo avere più di una versione dei costruttori. In questo caso si dice che il costruttore è **sovraccaricato** (o ***overloaded***). Per evitare codice ripetuto è possibile chiamare un costruttore da un altro.
  - **Inizializzazione di default**: se un campo non viene inizializzato con un valore da un costruttore, gliene viene assegnato uno di default:
    - **0** per valori numerici;
    - per valori booleani;
    - per gli oggetti.

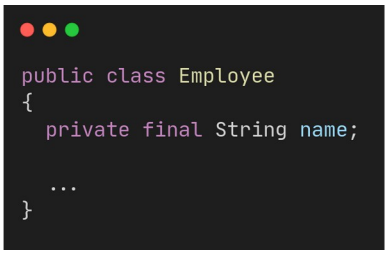
    Inoltre le variabili di istanza possono avere già un valore di default. Questa inizializzazione

```
public Employee (String name)
{
  this.name = name;
  //salary è automaticamente
  //settato a zero
}
```

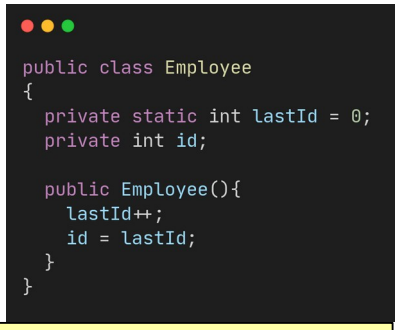
N.B.
Per evitare errori è conveniente assegnare sempre valori espliciti a tutti i campi di un oggetto.

viene eseguita dopo che l'oggetto è stato allocato in memoria e prima che venga eseguito il costruttore. Quindi un costruttore può tranquillamente sovrascrivere questo metodo.

- **Variabili di istanza *final***: le variabili di istanza **final** sono variabili di istanza il cui valore non può essere modificato una volta che è stato assegnato un valore iniziale.
  Le variabili di istanza *final* devono essere inizializzate durante la dichiarazione o nel costruttore della classe. Se non vengono inizializzate, il compilatore genererà un errore.

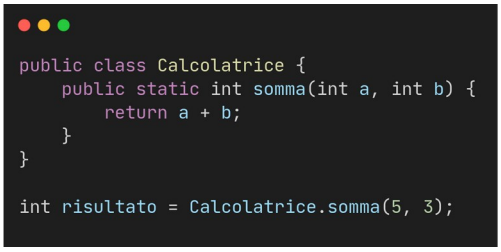


- **Variabili statiche**: sono variabili che appartengono alla classe piuttosto che a un'istanza specifica della classe. Ciò significa che tutte le istanze della classe condivideranno la stessa variabile statica e ogni modifica effettuata su di essa sarà visibile a tutte le istanze.
  Le variabili statiche vengono dichiarate utilizzando la parola chiave ***static*** dopo il modificatore di accesso e il tipo di dati della variabile.
  Le variabili statiche sono utilizzate spesso per tenere traccia delle informazioni che riguardano la classe stessa, come ad esempio il numero di istanze create.



N.B.
I campi statici tendenzialmente non vanno usati. Questo tipo di *modificatore* va usato con attenzione. Principalmente viene usato per le *costanti* (*final*).

- **Metodi statici** sono metodi che appartengono alla classe piuttosto che a un'istanza specifica della classe. Ciò significa che il metodo può essere chiamato utilizzando il nome della classe, invece che attraverso un'istanza specifica della classe.
  Poiché i metodi statici appartengono alla classe piuttosto che a un'istanza specifica della classe, non possono accedere alle variabili non statiche o ai metodi non statici dell'istanza. Invece, possono accedere solo ad altre variabili o metodi statici.



- **Pacchetti**: sono un meccanismo di organizzazione logica delle classi e delle interfacce in un progetto. Un pacchetto può contenere un numero arbitrario di classi e interfacce correlate tra loro, ed è utile per raggruppare insieme funzionalità correlate e per evitare possibili conflitti di nomi.

Le **regole** per la creazione di un pacchetto e l'uso di esso tramite il suo *path* sono:

- il nome del pacchetto dovrebbe essere descrittivo e riflettere il contenuto delle classi che contiene;
- il nome del pacchetto dovrebbe essere breve e non troppo complesso;
- utilizzare solo lettere minuscolo, numeri e il carattere punto (".");
- utilizzare lo schema di creazione di un URL in ordine inverso
  (es. *http://quasylab.unicam.it* ☐ *it.unicam.quasylab*);
- il nome del pacchetto deve identificare una struttura del *file system*.
  (es. *it.unicam.cs.pa* ☐ *it/unicam/cs/pa*);
- i pacchetti dovrebbero essere organizzati gerarchicamente, in modo da riflettere la relazione tra le classi contenute.

N.B.
I pacchetti annidati in Java sono pacchetti che sono contenuti all'interno di un altro pacchetto.
I pacchetti "*java.util*" e "*java.util.regex*" non sono pacchetti annidati, ma sono pacchetti separati all'interno del package *Java* e ognuno è un insieme di classi totalmente indipendenti

Ogni classe ha un ***fully qualified name,*** ovvero un nome completo di una classe che specifica il pacchetto in cui la classe è contenuta. Il fully qualified name è composto dal nome del pacchetto e dal nome della classe separati da un punto. Ad esempio, il fully qualified name per la classe *ArrayList* contenuta nel pacchetto *java.util* è *java.util.ArrayList*.
Il *fully qualified name* viene spesso utilizzato per specificare il nome completo di una classe in contesti in cui potrebbero esserci classi con lo stesso nome in pacchetti diversi. In questo modo, si può essere sicuri di identificare la classe corretta. Viene anche utilizzato per caricare dinamicamente le classi in Java.

N.B. per definizione ogni *package* è aperto e quindi è possibile aggiungere nuove classi ad esso. Per questo da Java 1.9 sono stati introdotti i **moduli**.

- **Modificatori di visibilità**: sono parole chiave che vengono utilizzate per definire l'accessibilità dei campi, dei metodi e delle classi all'interno di un programma. Ci sono quattro tipi di modificatori di visibilità:
  - **Public**: i membri pubblici sono accessibili da qualsiasi parte del programma, inclusi i pacchetti esterni. I metodi e i campi pubblici sono spesso utilizzati come interfacce pubbliche delle classi.
  - **Private**: i membri privati sono accessibili solo all'interno della classe in cui sono dichiarati. Questo garantisce l'incapsulamento dei dati e impedisce l'accesso non autorizzato ai membri della classe da parte di altre parti del programma.
  - **Protected**: i membri protetti sono accessibili all'interno della classe stessa, dei sottoclassi e del pacchetto in cui la classe è definita. Questo garantisce una certa flessibilità nell'accesso ai membri delle classi, consentendo l'estensione della classe e la condivisione dei membri all'interno del pacchetto.

    - **Default** (o package-private): i membri senza modificatore di visibilità (cioè senza la parola chiave public, private o protected) sono accessibili solo all'interno del pacchetto in cui la classe è definita. Questo permette di creare classi che sono visibili solo all'interno di un pacchetto e che non possono essere utilizzate da parti esterne.
- **Import delle classi**: l'importazione delle classi viene utilizzata per accedere alle classi definite in altri pacchetti o nel pacchetto corrente.
  L'importazione di una classe è necessaria se si vuole utilizzare una classe definita in un pacchetto diverso da quello corrente.
  Utilizzando l'istruzione *import static*, è possibile importare direttamente il metodo e il campo in modo che possano essere utilizzati direttamente nel codice senza specificare il nome della classe.
  L'istruzione *import static* può essere utilizzata solo per metodi e campi statici, non per metodi e campi di istanza. Inoltre, è buona pratica utilizzare questa istruzione solo per le classi che vengono utilizzate frequentemente e che contengono molte costanti o metodi statici.

```
//import singolo
import java.util.ArrayList;

//import multiplo
import java.util.*;

public class MyClass {
   // corpo della classe
}


import static MyClass.myMethod;
import static MyClass.myField;

...

myMethod();
int x = myField;
```

Compilazione delle classi Java

Ogni progetto Java dovrebbe avere *folder* differenti per:

- sorgenti
- file binari .*class* generati
- librerie utilizzate
- documentazione

La **compilazione** delle classi Java è il processo di conversione del codice sorgente Java in codice *bytecode*, che è il linguaggio di basso livello che viene eseguito sulla JVM.

Per compilare una classe Java, è sufficiente digitare il comando ***javac*** seguito dal nome del file sorgente Java con estensione ".java".

Classpath

Il **classpath** è una variabile d'ambiente che indica al compilatore e all'interprete Java dove trovare le classi e le librerie necessarie per eseguire un'applicazione Java. Il classpath è costituito da un insieme di percorsi di **directory** che indicano dove trovare i file **.class** e i file **.jar** necessari all'esecuzione dell'applicazione.

Se una classe o una libreria non viene trovata nel classpath, il compilatore genera un errore di compilazione.

Factory Methods

Nella programmazione ad oggetti, il **Factory Method** è uno dei design pattern fondamentali. Permette di definire un'interfaccia per la creazione di oggetti, ma lascia alle sottoclassi la decisione su quale classe concreta istanziare.

```
public interface FiguraGeometrica {
    void disegna();
}

public class Cerchio implements FiguraGeometrica {
    public void disegna() {
        System.out.println("Disegna cerchio.");
    }
}

public class Triangolo implements FiguraGeometrica {
    public void disegna() {
        System.out.println("Disegna triangolo.");
    }
}

public class FiguraGeometricaFactory {
    public static FiguraGeometrica creaFiguraGeometrica(String figura) {
        if (figura.equalsIgnoreCase("Cerchio")) {
            return new Cerchio();
        } else if (figura.equalsIgnoreCase("Triangolo")) {
            return new Triangolo();
        } else {
            return null;
        }
    }
}
```

Essi quindi permettono di creare nuovi oggetti con parametri differenti che l'utilizzatore della classe non conosce.

Il Factory Method può essere implementato mediante un metodo statico all'interno di una classe factory. Il metodo statico viene chiamato per creare un oggetto e restituisce un'istanza della classe desiderata. La classe factory è responsabile della creazione di nuove istanze dell'oggetto e dell'inizializzazione di eventuali parametri necessari.

Nell'esempio, la classe factory *FiguraGeometricaFactory* ha un metodo statico *creaFiguraGeometrica* che restituisce un'istanza della classe *FiguraGeometrica* in base al parametro passato ("Cerchio", "Triangolo" o "Rettangolo").

Interfacce

Le **interfacce** sono un meccanismo per definire un insieme di metodi che una classe deve implementare.
Un'interfaccia può essere pensata come un contratto che una classe deve rispettare se vuole implementarla.

```
public interface Movable {
    void moveForward(int distance);
    void moveBackward(int distance);
}

public class Car implements Movable {
    public void moveForward(int distance) {
        // implementazione del metodo moveForward per la classe Car
    }
    public void moveBackward(int distance) {
        // implementazione del metodo moveBackward per la classe Car
    }
}
```

Le interfacce sono la prima cosa che si scrive quando si programma, per definire i concetti che devono emergere e quali sono le responsabilità che bisogna rappresentare.

Per definire un'interfaccia in Java, si utilizza la parola chiave ***interface*** seguita dal nome dell'interfaccia e dalle definizioni dei metodi, senza alcuna implementazione.

Una classe può implementare un'interfaccia con il metodo ***implements*** e può implementare un qualsiasi numero di interfacce. Quella classe poi dovrà fornire le implementazioni di tutti i metodi che queste interfacce definiscono.

Le interfacce possono contenere **costanti**, ovvero valori che non possono essere modificati dopo la loro definizione. Le costanti definite in un'interfaccia sono *public*, *static* e *final*, il che significa che possono essere utilizzate da qualsiasi classe che implementa l'interfaccia.

Per utilizzare una costante definita in un'interfaccia, è sufficiente fare riferimento al nome della costante preceduto dal nome dell'interfaccia.

N.B.

Se una classe implementa due o più interfacce che contengono un metodo con lo stesso nome, la classe deve fornire l'implementazione di quel metodo. La soluzione dipende da come si vuole gestire la situazione in cui le interfacce contengono metodi con la stessa firma.

Se le implementazioni dei metodi nella classe sono uguali, allora non ci sono problemi. In caso contrario, è necessario fornire un'implementazione specifica del metodo per ciascuna interfaccia che contiene il metodo con lo stesso nome.

```java
public interface Colors {
    public static final String RED = "FF0000";
    public static final String GREEN = "00FF00";
    public static final String BLUE = "0000FF";
}


public class MyObject implements Colors {
    public void printColor() {
        System.out.println("My color is " + Colors.BLUE);
    }
}
```

Interfacce e tipi

Def.

Il **tipo statico** di una variabile è quello dichiarato quando viene definita la variabile. Viene utilizzato dal compilatore per verificare che le operazioni effettuate su quella variabile siano consentite dal tipo dichiarato.

Il **tipo dinamico** è il tipo effettivo dell'oggetto assegnato alla variabile in fase di esecuzione. Viene utilizzato a runtime per determinare quali metodi e campi sono disponibili per l'oggetto effettivamente assegnato alla variabile.

Un **sovratipo** (o **superclasse**) è una classe da cui altre classi ereditano attributi e metodi. Un sovratipo può essere una classe astratta o concreta.

Un **sottotipo** (o **sottoclasse**) è una classe che eredita attributi e metodi da un sovratipo. Un sottotipo può aggiungere nuovi attributi e metodi o ridefinire quelli ereditati dal sovratipo.

Il sovratipo è una classe più generica che fornisce il comportamento di base per uno o più sottotipi più specifici, che ereditano da essa. La classe sottotipo può quindi estendere il comportamento del sovratipo fornendo nuovi attributi e metodi o ridefinendo quelli ereditati.

"*S* è un *sovratipo* di T (il *sottotipo*) quando ogni valore di tipo T può essere assegnato ad una variabile di tipo S senza un *cast*."

In altre parole, il tipo T può essere utilizzato in ogni altro contesto in cui mi aspetto un tipo S. Se **necessario** è possibile utilizzare il comando *instanceof* per verificare la correttezza dell'operazione di cast inverso.

Esempio

Consideriamo la seguente classe *Vehicle* e la classe *Car* che eredita da *Vehicle.*

In questo caso, *Vehicle* è il sovratipo di *Car*, mentre *Car* è il sottotipo di *Vehicle*.

Ciò significa che un oggetto di tipo *Car* può essere utilizzato ovunque un oggetto di tipo *Vehicle* è richiesto, ma non viceversa.

In questo caso, abbiamo creato un oggetto di tipo *Car* e lo abbiamo assegnato a una variabile di tipo *Vehicle*. Grazie alla relazione di ereditarietà, possiamo chiamare i metodi ereditati da *Vehicle* sull'oggetto *myVehicle*, ad esempio:

Tuttavia, non possiamo chiamare il metodo *getModel()* sull'oggetto *myVehicle*, in quanto *getModel()* è un metodo definito solo nella classe *Car* e non è presente nella classe *Vehicle*.

```
public class Vehicle {
    private int speed;

    public void accelerate(int increment) {
        speed += increment;
    }

    public int getSpeed() {
        return speed;
    }
}

public class Car extends Vehicle {
    private String model;

    public Car(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}


Vehicle myVehicle = new Car("Toyota");
myVehicle.accelerate(10);
int speed = myVehicle.getSpeed();
```

La relazione tra sovratipo e sottotipo è un tipo di relazione "è un/a" (o "is-a" in inglese). Ad esempio, possiamo dire che *Car* è un sottotipo di *Vehicle*, poiché una macchina è un tipo di veicolo.

Estendere le interfacce

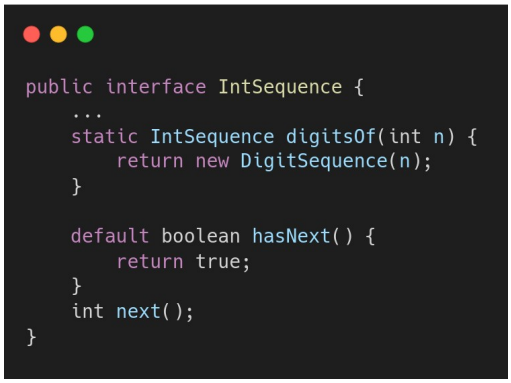
Un'interfaccia può essere estesa da un'altra interfaccia, richiedendo o fornendo metodi aggiuntivi o costanti a quelli forniti dall'interfaccia originale. Un'interfaccia può estendere più di una interfaccia madre.

```
public interface InterfacciaFiglia extends InterfacciaMadre {
    // qui puoi aggiungere nuovi metodi e costanti
}

public interface InterfacciaFiglia extends InterfacciaMadre1, InterfacciaMadre2, ... {
    // qui puoi aggiungere nuovi metodi e costanti
}
```

Metodi di un interfaccia

Le interfacce possono contenere diverse tipologie di metodi con un'implementazione *concreta*:

- **metodi astratti:** sono metodi che non hanno un'implementazione e devono essere implementati da tutte le classi che implementano l'interfaccia;
- **metodi statici**: sono metodi che sono definiti come statici e non richiedono un'istanza dell'interfaccia per essere chiamati. Vengono utilizzati per definire delle funzionalità generiche (*factory methods*) che costruiscono specifiche istanze dell'interfaccia nascondendo però l'esatta implementazione;



```
public interface IntSequence {
    ...
    static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }

    default boolean hasNext() {
        return true;
    }
    int next();
}
```

- **metodi di default**: sono metodi che forniscono un'implementazione di default per un metodo nell'interfaccia stessa. Le classi che implementano l'interfaccia possono scegliere di utilizzare l'implementazione di default o di fornire una loro implementazione personalizzata. A differenza dei metodi astratti, i quali devono essere implementati da tutte le classi che implementano l'interfaccia, i metodi di default non sono obbligatoriamente implementati dalle classi che implementano l'interfaccia, ma possono essere utilizzati come implementazione di default per i metodi dell'interfaccia;
- **metodi privati**: sono metodi che sono definiti come privati e possono essere utilizzati per fornire funzionalità ausiliarie agli altri metodi dell'interfaccia. Questi metodi non sono visibili al di fuori dell'interfaccia stessa. Possono essere *static* o meno e possono solo utilizzare altri metodi definiti nelle interfacce.

L'utilizzo di queste diverse tipologie di metodi nelle interfacce consente di definire comportamenti comuni che possono essere utilizzati dalle classi che implementano l'interfaccia, senza dover ripetere il codice per ogni classe. Inoltre, l'aggiunta di metodi di default e statici alle interfacce consente di estendere le funzionalità delle interfacce senza rompere la compatibilità con le implementazioni esistenti.

Ereditarietà

L'**ereditarietà** è un meccanismo che consente a una classe di ereditare le proprietà e i comportamenti di un'altra classe. La classe che eredita da un'altra classe viene chiamata **sottoclasse** o classe derivata, mentre la classe ereditata viene chiamata **superclasse** o classe di base.

N.B.
Una sottoclasse ha più funzionalità di una superclasse in quanto può implementare nuovi metodi.

Per definire una *sottoclasse* in Java, si utilizza la parola chiave ***extends*** seguita dal nome della superclasse.

Quando una sottoclasse eredita da una superclasse, eredita tutti i metodi e le variabili pubblici e protetti della superclasse. La sottoclasse può anche aggiungere i propri metodi e variabili. Tuttavia, non può accedere direttamente ai metodi e alle variabili privati della superclasse.

Per **sovrascrivere** (***@Override***) un metodo, la sottoclasse deve definire un metodo con lo stesso nome, gli stessi parametri e lo stesso tipo di ritorno del metodo della superclasse (firma). Il metodo della sottoclasse sostituirà il metodo della superclasse quando viene chiamato sulla sottoclasse.

N.B.
Il nuovo metodo potrebbe avere un tipo di ritorno diverso da quello che viene sovrascritto purché quest'ultimo sia un sottotipo del tipo di ritorno del metodo sovrascritto

L'ereditarietà è un concetto importante nella programmazione orientata agli oggetti perché consente di riutilizzare il codice e creare gerarchie di classi correlate. Definendo una funzionalità comune in una superclasse, è possibile ridurre la duplicazione del codice e rendere il codice più facile da mantenere.

Costruzione di una sottoclasse

Una sottoclasse deve invocare l'opportuno costruttore della sua superclasse per inizializzare i campi *private*.

Esso viene invocato con il metodo ***super***. Se non è presente viene invocato il costruttore di default della superclasse senza parametri.

```java
public class Employee {
  private final String name;
  private double salary;
  private static int lastId=0;
  private int id;

  public Employee (String name, double salary) {
    Employee.lastId++;
    this.id=lastId;
    this.name=name;
    this.salary=salary;
}


public class Manager extends Employee
{
  ...

  public Manager (String name, double salary)
  {
      super(name, salary);
      this.bonus = 0;
  }
  ...
}
```

Assegnamento – Dynamic Lookup

E’ legale assegnare un oggetto di una sottoclasse ad una variabile il cui tipo è una superclasse. Questo perché il tipo dinamico è sempre un sottotipo del tipo statico.

```java
public class Persona {
    public void saluta() {
        System.out.println("Ciao, sono una persona!");
    }
}

public class Studente extends Persona {
    @Override
    public void saluta() {
        System.out.println("Ciao, sono uno studente!");
    }
}

public class Main {
    public static void main(String[] args) {
        Persona p = new Persona();
        Studente s = new Studente();
        Persona ps = new Studente();

        p.saluta();  // Stampa "Ciao, sono una persona!"
        s.saluta();  // Stampa "Ciao, sono uno studente!"
        ps.saluta(); // Stampa "Ciao, sono uno studente!"
    }
}
```

Il ***dynamic method lookup*** è un meccanismo di Java (e di molti altri linguaggi di programmazione orientati agli oggetti) che permette di risolvere dinamicamente quale metodo deve essere eseguito in una classe, in base al tipo dell'oggetto a cui il metodo è applicato, durante l'esecuzione del programma.

Quando si chiama un metodo su un oggetto, il compilatore cerca prima il metodo nella classe dell'oggetto e, se non lo trova, risale la gerarchia delle classi fino a trovare un metodo con lo stesso nome e gli stessi parametri.

N.B.
Nel terzo caso, l'oggetto è di tipo Studente, ma la variabile a cui è assegnato è di tipo Persona, quindi viene eseguito il metodo della sottoclasse perché il *dynamic method lookup* si basa sul tipo effettivo dell'oggetto, non sulla variabile a cui è assegnato.

Tuttavia, se il metodo viene trovato in una classe superiore, ma viene sovrascritto in una sottoclasse, il metodo della sottoclasse verrà eseguito invece di quello della superclasse.

Cast

Il **cast** è un'operazione che permette di convertire un oggetto di una classe in un oggetto di un'altra classe. Questa operazione può essere utilizzata anche in combinazione con l'ereditarietà per convertire un oggetto di una classe padre in un oggetto di una classe figlia o viceversa.

```java
if(empl instanceof Manager) {
  Manager mgr = (Manager) empl;
  mgr.setBonus(10000);
}
```

L'operatore ***instanceof*** è utilizzato per verificare se un oggetto è di una determinata classe o sottoclasse prima di effettuare un cast esplicito. L'utilizzo dell'operatore *instanceof* prima del cast esplicito è una pratica comune per evitare errori di esecuzione del programma dovuti al cast di un oggetto che non è effettivamente un'istanza della classe specificata. In questo modo, possiamo gestire l'eccezione *ClassCastException* che verrebbe altrimenti sollevata.

Metodi *final*

Un metodo può essere dichiarato come *final* per indicare che non può essere sovrascritto dalle sottoclassi. La dichiarazione di un metodo come *final* viene utilizzata per proteggere la logica del metodo da eventuali modifiche indesiderate nelle sottoclassi. Ciò è particolarmente utile in situazioni in cui un metodo è fondamentale per il funzionamento della classe e deve essere mantenuto intatto per garantire che il comportamento della classe rimanga consistente.

```
public interface IntSequence {
    ...
    static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }

    default boolean hasNext() {
        return true;
    }
    int next();
}
```

Metodi e classi astratte

Le classi astratte e i metodi astratti sono utilizzati per fornire una struttura base per le sottoclassi e definire metodi che **devono** essere implementati nelle sottoclassi.

Una **classe astratta** è una classe che non può essere istanziata, ma può essere utilizzata come classe base per definire le sottoclassi. Una classe astratta può contenere metodi concreti (cioè implementati) e metodi astratti (cioè non implementati).

Un **metodo astratto** è un metodo che non ha un'implementazione definita nella classe in cui viene dichiarato, ma deve essere implementato nelle sottoclassi. Un metodo astratto è definito utilizzando la parola chiave *abstract* nella sua dichiarazione.

Sia le classi astratte che le interfacce sono uno strumento di **astrazione**.

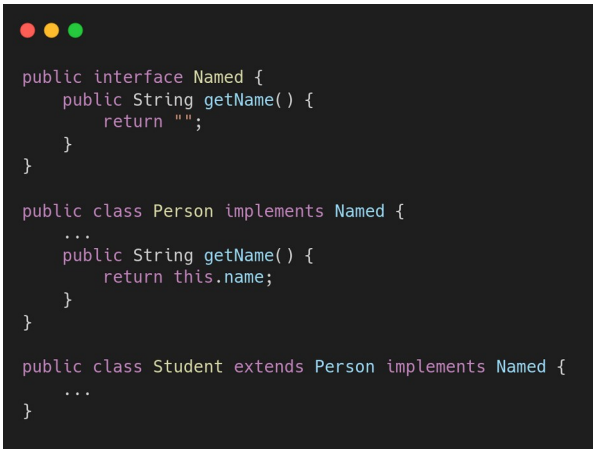
```
public abstract class Person {
    private final String name;

    public Person(String name) {
      this.name = name
    }

    public final String getName() {
        return name;
    }

    public abstract int getId();
}


public class Student extends Person {
    private final int id;

    public Student(int id, String name){
        super(name);
        this.id=id;
    }

    public final int getId(){
        return id;
    }
}
```

Le classi astratte sono usate, **in combinazione con le interfacce**, quando:

- vogliamo definire un comportamento comune con uno strato comune definito
- abbiamo un comportamento che dipende da alcuni metodi da personalizzare

Approccio *class-win* (ereditarietà multipla)

Quando abbiamo a che fare con una classe che estende un'altra classe e implementa allo stesso tempo un'interfaccia ed entrambe queste ultime hanno un metodo con lo stesso nome, Java adotta l'**approccio *class-win***.



```
public interface Named {
    public String getName() {
        return "";
    }
}

public class Person implements Named {
    ...
    public String getName() {
        return this.name;
    }
}

public class Student extends Person implements Named {
    ...
}
```

In questo approccio, quando una classe estende una superclasse e implementa un'interfaccia, ereditando il medesimo metodo da entrambe, vengono utilizzati i metodi definiti nella superclasse. Questo significa che l'implementazione fornita dalla classe prevale su quella fornita dall'interfaccia.

La superclasse *Object*

Object è una classe predefinita che è la superclasse di tutte le altre classi. Ogni classe in Java è direttamente o indirettamente derivata dalla classe *Object*. Ciò significa che ogni classe Java eredita metodi e campi dalla classe *Object*, che possono essere utilizzati o sovrascritti a seconda delle esigenze.

La classe Object contiene diversi metodi:

- ***String toString()***: questo metodo restituisce una rappresentazione stringa dell'oggetto. Di default, questo metodo restituisce il nome della classe seguito dall'hashcode dell'oggetto, ma può essere sovrascritto dalle sottoclassi per fornire una rappresentazione più significativa dell'oggetto;
- ***protected void finalize()***: questo metodo viene chiamato dal *garbage collector* prima di eliminare l'oggetto dalla memoria per eseguire attività di pulizia su quell'oggetto. Le sottoclassi possono sovrascrivere questo metodo per eseguire azioni specifiche prima che l'oggetto venga eliminato;
- ***wait(), notify(), notifyAll()***: vengono utilizzati per la sincronizzazione tra thread e sono ereditati da tutte le classi Java;
- ***Class<?> getClass()***: questo metodo restituisce l'oggetto *Class* che rappresenta la classe dell'oggetto;

- ***boolean equals(Object other)***: questo metodo viene utilizzato per confrontare due oggetti per uguaglianza. Di default, questo metodo confronta gli oggetti per riferimento (cioè se gli oggetti sono gli stessi), ma può essere sovrascritto dalle sottoclassi per definire un confronto più specifico. Il contratto del metodo ***equals()*** in Java specifica le regole che devono essere seguite per la definizione di questo metodo in una classe. Il contratto stabilisce che:
  - **Riflessività**: ogni oggetto deve essere uguale a se stesso. In altre parole, l'espressione ***x.equals(x)*** deve sempre restituire ***true***;
  - **Simmetria**: se due oggetti sono uguali, allora devono essere uguali in entrambe le direzioni. In altre parole, se ***x.equals(y)*** restituisce ***true***, allora anche ***y.equals(x)*** deve restituire ***true***;
  - **Transitività**: se tre oggetti sono uguali, allora anche il confronto tra i primi due oggetti e il confronto tra il secondo e il terzo oggetto devono restituire ***true***. In altre parole, se ***x.equals(y)*** e ***y.equals(z)*** restituiscono entrambi ***true***, allora anche ***x.equals(z)*** deve restituire ***true***;
  - **Consistenza**: il risultato del metodo *equals()* deve essere coerente con lo stato degli oggetti coinvolti nel confronto. In altre parole, se il metodo *equals()* viene chiamato su due oggetti, il risultato deve essere sempre lo stesso, fintanto che lo stato degli oggetti non cambia;
  - **Nullità**: il metodo *equals()* deve restituire ***false*** se viene confrontato con ***null***;

  Inoltre, il metodo *equals()* dovrebbe essere sovrascritto nelle sottoclassi solo se ha senso definire un confronto più specifico rispetto a quello di default basato sul riferimento degli oggetti.
- ***int hashCode()***: questo metodo restituisce un codice hash per l'oggetto. Di default, questo metodo restituisce un codice hash basato sull'indirizzo dell'oggetto in memoria, ma può essere sovrascritto dalle sottoclassi per fornire un codice hash più significativo.

```
int hash = 0;
for (int i = 0; i<length(); i++) {
    hash = 31 * hash + charAt(i);
}
```

  Gli hashcodes dovrebbero essere ottenuti in modo tale che se x e y sono due oggetti e non sono uguali allora ***x.hashCode()*** e ***y.hashCode()*** dovrebbero restituire valori differenti con alta probabilità. Il contratto del metodo ***hashCode()*** in Java specifica le regole che devono essere seguite per la definizione di questo metodo in una classe. Il contratto richiede che:
  - È importante che il valore di hash di un oggetto sia consistente con il suo stato. In altre parole, se un oggetto viene modificato in modo tale da renderlo uguale a un altro oggetto, allora il suo valore di hash deve essere modificato in modo tale da renderlo uguale al valore di hash dell'altro oggetto;

    - Se il metodo *equals()* di due oggetti restituisce true, allora il loro valore di hash deve essere uguale. In altre parole, se ***x.equals(y)*** restituisce ***true***, allora ***x.hashCode()*** deve essere uguale a ***y.hashCode()***.

- ***protected Object clone()***: utilizzato per creare una copia di un oggetto esistente. Questo metodo è dichiarato *protected* in modo tale che possa essere sovrascritto se necessario.
  Il metodo ***clone()*** di default esegue una copia a livello di bit dell'oggetto clonato, ma se la classe contiene riferimenti ad altri oggetti, questi riferimenti verranno copiati come valori di riferimento, quindi i riferimenti puntano agli stessi oggetti originali dell'oggetto originale (***shallow copy***). Questo può portare a inconsistenze e problemi di sicurezza, in quanto, se l'oggetto clonato modifica dei valori ad esempio di una variabile, la stessa modifica viene riportata nell'oggetto originale.
  La ***deep copy***, o copia profonda, crea una copia dell'oggetto originale e di tutti gli oggetti a cui fa riferimento, in modo da avere copie completamente separate e indipendenti. In questo modo, eventuali modifiche apportate a un oggetto copiato non influiranno sull'oggetto originale o su altri oggetti copiati. Per eseguire una deep copy in Java, è necessario fare l'override del metodo *clone()* e implementare la copia dei campi in modo personalizzato, in modo da garantire la copia di tutti gli oggetti a cui fa riferimento l'oggetto originale. La scelta tra shallow copy e deep copy dipende dalle esigenze specifiche del programma.

Enumerazioni

Una ***enum*** è un tipo di dato che rappresenta un insieme di costanti ben definite e limitate. Le enumerazioni sono definite come una classe speciale che ha un elenco di valori costanti che rappresentano i membri dell'enumerazione.

Poiché gli elementi di un'enumerazione sono costanti, è garantito che due elementi con lo stesso nome rappresentano la stessa istanza dell'enumerazione, quindi l'operatore di uguaglianza "==" può essere utilizzato per confrontare due elementi di un'enumerazione. Esiste inoltre un'implementazione predefinita del metodo *toString()* per le enumerazioni in Java. Quando si chiama il metodo *toString()* su un'istanza di un'enumerazione, viene restituito il nome dell'elemento dell'enumerazione in formato di stringa.

Le enumerazioni forniscono anche alcuni metodi utili come:

- ***values()***: restituisce un'array di tutte le costanti dell'enumerazione in ordine in cui sono state definite;
- ***valueOf(String name)***: restituisce l'istanza dell'enumerazione corrispondente al nome specificato;
- ***ordinal()***: restituisce l'indice dell'elemento dell'enumerazione nell'array delle costanti dell'enumerazione (ovvero la posizione in cui è stato definito l'elemento dell'enumerazione).

Le enumerazioni implementano l'interfaccia ***Comparable<E>***. Questo significa che è possibile confrontare le costanti dell'enumerazione utilizzando il metodo *compareTo()*.

Se necessario è possibile aggiungere costruttori, metodi o campi ai tipi *enum.*

Tipo Sealed

Un interfaccia o una classe di tipo ***sealed*** viene utilizzata per dichiarare una classe sigillata. Una classe sigillata è una classe che limita quali altre classi possono estenderla. Quando una classe è dichiarata come sigillata, è necessario specificare esplicitamente le classi che possono estenderla. Questo offre un maggiore controllo sulla gerarchia delle classi.

Una classe di tipo *sealed* ha la capacità di controllare l'ereditarietà di queste in maniera più dettagliata consentendo di definire i sottotipo consentiti.

Per dichiarare una classe sigillata, si utilizza la parola chiave *sealed* seguita dalla definizione della classe. Insieme a *sealed*, è necessario specificare quali classi sono consentite a estendere la classe sigillata utilizzando la clausola *permits*.

L'utilizzo di classi sigillate migliora la chiarezza e la robustezza del design delle classi, impedendo l'estensione non controllata e garantendo che la gerarchia delle classi sia più gestibile. Tuttavia, è importante pianificare attentamente l'utilizzo delle classi sigillate per evitare una progettazione troppo restrittiva che potrebbe complicare l'estensione futura del codice.

Programmazione generica

La programmazione generica consiste nella creazione di costrutti di programmazione che possano essere utilizzati con molti tipi di dati diversi, senza doverli definire esplicitamente prima dell'esecuzione del programma.

```
public class Entry<K, V> {
    private K key;
    private V value;
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return this.key;
    }
    public V getValue() {
        return this.value;
    }
}

Entry<String, Integer> entry =
    new Entry<String, Integer>("Harry", 17);
```

Una **classe generica** è una classe che è definita con **uno** o **più parametri di tipo generico**, invece di utilizzare un tipo specifico. I parametri di tipo generico sono rappresentati da nomi di tipo che vengono scelti dal programmatore e possono essere utilizzati all'interno della classe per specificare il tipo di un oggetto o di una collezione. Quando si utilizza una classe generica, il tipo specifico deve essere specificato quando si crea un'istanza della classe. Nel costruttore il parametro di tipo può essere omesso in quanto può venire assegnato anche dal contesto.

La coppia vuota "*<>*" è obbligatoria ed è chiamata ***diamond*** (operatore). Le classi generiche sono utili per progettare ***strutture dati*** (es. *ArrayList<T>*). I parametri di tipo **non possono essere istanziati con tipi primitivi** (es. *Entry<String, int>* non è valido).

```
public class Arrays {
    public static <T> T[] swap (int i, int j, T ... values) {
        T temp = values[i];
        values[i] = values[j];
        values[j] = temp;
        return values;
    }
}
```

E' possibile anche definire un **metodo generico** a livello di classe. E' un metodo statico perché è un metodo la cui esecuzione non dipende dal contesto, per questo non darà un errore. I parametri di tipo <T> sono posizionati dopo il modificatore e prima del tipo di ritorno. Esso serve per *introdurre* il tipo *T* all'interno del metodo utilizzato.

ES.

```
Double[] result = Arrays.swap(0, 1, 1.5, 2, 3);

Double[] result = Arrays.<Double>swap(0, 1, 1.5, 2, 3);
```

Il codice a fianco darà un errore di tipo in quanto, secondo il metodo "*swap*" scritto sopra, nei parametri che vengono passati al metodo esso si aspetta di trovare una sequenza di numeri dello stesso tipo (*T … values*). Ma all'invocazione del metodo trova anche dei valori di tipo *int* oltre a quelli *double*. A questo punto, quando il metodo andrà a fare il **boxing**, gli *int* diventeranno degli *Integer.* La conversione non può avvenire in quanto le classi *Integer* e *Double* non sono confrontabili (il cast può avvenire tra *int* e *double* ma non tra *Integer* e *Double*).

N.B.

Il **boxing** si riferisce al processo di incapsulamento di un tipo primitivo in un oggetto *wrapper* corrispondente. Questo processo è necessario quando si utilizzano tipi primitivi come argomenti per le classi generiche o i metodi generici, in quanto questi richiedono che i tipi di dati siano oggetti.

Per risolvere questo errore, dovremmo fare un cast esplicito a *double* per gli interi.

Type checking (Verifica del tipo)

Si riferisce al processo attraverso il quale il compilatore controlla se i tipi di dati utilizzati nel codice siano compatibili con quanto dichiarato nel programma. In Java, il type checking è una parte fondamentale del sistema di tipo statico del linguaggio.

Per eseguire questi controlli il verificatore deve tenere un record del tipo associato a ciascun nome e controllare questo tipo ogni volta che si fa riferimento al nome nel programma. Se c'è una discrepanza tra i tipi dichiarati e i tipi effettivi utilizzati nel codice, il compilatore segnalerà un errore di tipo.

Un altro controllo da fare quando si dichiara una variabile è che le venga assegnato un tipo esistente. Se il programmatore dichiara una variabile di tipo definito dall'utente, deve aver definito anche il tipo. Questo controllo rileverà anche semplici errori tipografici nei nomi dei tipi.

Contribuisce a prevenire errori di tipo a tempo di compilazione, garantendo che le operazioni sui dati rispettino le regole dei tipi dichiarati. Il type checking statico aiuta a prevenire molti errori comuni a tempo di esecuzione, contribuendo a rendere il codice più robusto e prevedibile.

Type erasure (Cancellazione del tipo)

Java utilizza un sistema di tipo generico basato su type erasure, il che significa che le informazioni sui tipi generici sono cancellate (o "cancellate") durante la compilazione.
Quando una classe generica è compilata, tutte le informazioni sul tipo generico sono perse e le classi sono trasformate in tipi ***raw***, un tipo che non ha informazioni. Ad esempio, la classe Entry, tutti i parametri di tipo vengono persi e tutto viene trattato come *Object*. In questo modo l'*algoritmo di typing* di Java sa che, dopo aver controllato in fase di compilazione tutti i tipi, non avrà più bisogno di controllarli a *runtime*. Così il sistema di typing è più efficiente perché:

- non deve fare controlli di tipi a *runtime*;
- potrà andare più veloce perché potrà ottimizzare il codice generato.

Fornisce compatibilità con le versioni precedenti del bytecode Java.

Type inference (Inferenza di tipo)

L'inferenza del tipo è la capacità del compilatore di dedurre il tipo di una variabile o di un'espressione senza richiedere una dichiarazione esplicita del tipo da parte del programmatore.

Rende il codice più conciso e leggibile. È spesso associata all'uso di generici per ridurre la necessità di specificare il tipo in determinate situazioni.

Wildcards

Nelle classi generiche come *ArrayList<Employee>* non posso passare un *ArrayList<Manager>*. Ma può essere conveniente utilizzare un meccanismo che permette di specificare argomenti il cui tipo estenda la classe che voglio passare.
Per risolvere il problema vengono utilizzare le **wildcards *<?>***. Sono utilizzate nei generics per rendere più flessibili le dichiarazioni di tipo, consentendo di lavorare con tipi sconosciuti o generici.

Variabili di tipo
- **E**: tipo di un elemento in una raccolta;
- **K**: tipo di una chiave in una mappa
- **V**: tipo di un valore in una mappa
- **T**: tipo generico
- **S, U**: ulteriori tipi generici

N.B.
Le wildcards possono essere utilizzate come argomento di tipo ma non come tipo.
Es. *? Temp = elements...*☐ non possibile

Type Bound

Il **type bound** è un meccanismo utilizzato nella programmazione generica per limitare i tipi di dati che possono essere utilizzati come argomenti per una classe generica o un metodo generico. Il type bound viene utilizzato per specificare un intervallo di tipi di dati ammissibili per il parametro di tipo generico.

I type bounds sono specificati tramite l'uso dell'operatore di estensione (*extends*) o dell'operatore di super (*super*). L'operatore ***extends*** viene utilizzato per specificare un **upper bound**, ovvero un limite superiore per il tipo di dati che può essere utilizzato come parametro di tipo generico, mentre l'operatore ***super*** viene utilizzato per specificare un **lower bound**, ovvero un limite inferiore.

I type bounds possono essere specificati anche con i wildcards nei generics. Ciò consente di imporre ulteriori vincoli sui tipi che possono essere utilizzati come argomenti di tipo nei generics.

Es. Upper bound

In questo esempio, il parametro di tipo *T* è limitato dall'**upper bound *extends Comparable<T>***. Ciò significa che il tipo *T* deve essere un sottotipo dell'interfaccia *Comparable<T>*. Ciò consente al metodo di confrontare gli oggetti del tipo *T* utilizzando il metodo *compareTo()*.
In questo caso, il tipo *T* viene dedotto come *Integer*, che è un sottotipo dell'interfaccia *Comparable<Integer>*.

```
public static <T extends Comparable<T>> T massimo(T[] array) {
    T max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (array[i].compareTo(max) > 0) {
            max = array[i];
        }
    }
    return max;
}

Integer[] arrayInt = {1, 2, 3, 4, 5};
Integer maxInt = massimo(arrayInt); // restituisce 5
```

Es. Lower bound

```java
public static <T> void aggiungiElemento(List<? super T> lista, T elemento) {
    lista.add(elemento);
}

List<Number> listaNumeri = new ArrayList<>();
aggiungiElemento(listaNumeri, 10); // aggiunge l'intero 10 alla lista

List<Object> listaOggetti = new ArrayList<>();
aggiungiElemento(listaOggetti, "ciao"); // Errore in compilazione, "ciao" non è un supertipo di Integer
```

In questo esempio, il parametro *lista* ha un **lower bound** definito come ***<? super T>***. Ciò significa che *lista* è una lista di un tipo sconosciuto *?*, ma che deve essere un *supertipo* di *T* (o *T* stesso). Il parametro *elemento* ha il tipo *T*, quindi il metodo accetta un oggetto del tipo *T* e lo aggiunge alla lista.
Il vantaggio di utilizzare un *lower bound* è che il metodo può accettare liste di tipo diverso, purché siano *supertipi* del tipo *T*.

Ricapitolando:

- l'***upper bound*** utilizza l'operatore *extends* e specifica che il tipo di dato che il metodo/classe *className<T extends A>* accetta, è un tipo generico a patto che il tipo di *T* sia sottotipo di *A* (es. *class B extends A {…)*;
- il ***lower bound*** utilizza l'operatore *super* e specifica che il tipo di dato che il metodo/classe *className<T super A>* accetta, è un tipo generico a patto che il tipo di *T* sia sovratipo di *A*, dunque la classe *A* deve estendere la classe *T* e, viceversa, la classe *T* deve essere estesa da *A*.

Subtyping e covarianza (Arrays)

Il **subtyping** si riferisce alla relazione in cui un tipo può essere considerato un sottotipo di un altro tipo. Ad esempio, se B è un sottotipo di A, allora un'istanza di B può essere utilizzata dove ci si aspetta un'istanza di A. L'idea di *subtyping* è quella di consentire ad un oggetto di una classe di essere utilizzato ovunque un oggetto di una sua *superclasse* sia richiesto. In altre parole, un oggetto di una classe più specifica (la sottoclasse) può essere utilizzato in luogo di un oggetto di una classe più generale (la superclasse).

La **covarianza** si riferisce alla relazione tra i tipi generici, indicando che un tipo generico parametrizzato con un tipo più specifico è considerato un sottotipo del tipo generico parametrizzato con il tipo più generale. È spesso utilizzata in contesti di subtyping per rendere più flessibile l'utilizzo di tipi generici. Quando si lavora con ereditarietà e tipi generici, è importante che la subtyping e la covarianza siano gestite correttamente per garantire che il codice sia sicuro e corretto.

Gli array sono ***covarianti***, il che significa che è possibile assegnare un array di un tipo più specifico a una variabile di un tipo più generico.

Es.

Consideriamo la classe ***Employee*** e la sua sottoclasse ***Manager***.
Creiamo un array di *Manager* e settiamo con il metodo ***set()*** un nuovo ***Employee*** alla posizione 0 dell'array ***mngrs***.

Quando selezioniamo l'oggetto dell'array ***mngrs*** nello stesso indice passato al metodo ***set()***, questo non sarà più di tipo ***Manager*** ma sarà diventato un ***Employee***. Ciò significa che non sarà più possibile richiamare i metodi della classe ***Manager***. Per cui nell'esempio, a *compile-time*, il compilatore restituirà un errore e viene lanciata un eccezione *ArrayStoreException*.

```
public class Employee {
    private String name;
    private int salary;

    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
}

public class Manager extends Employee
{
    ...
    public Manager(String name, int salary) {
        super(name, salary);
        this.bonus = 0;
    }
    public void callManagerSpecificMethod(){
        ...
    }
}

public void set(Employee[] staff, int i, Employee e)
{
    staff[i] = e;
}

Manager[] mngrs = ...;
set(mngrs, 0, new Employeee());
mngrs[0].callManagerSpecificMethod();
```

Per evitare questi problemi, in generale, è consigliabile utilizzare le collezioni generiche di Java (come *ArrayList*) anziché gli array, in quanto le collezioni offrono una maggiore sicurezza dei tipi. La covarianza degli array è uno dei motivi per cui le collezioni generiche sono preferite in molti scenari.

Restrizioni sui Generics

- Non è possibile usare i tipi primitivi come argomenti di tipo;
- A runtime, tutti i tipi sono ***raw*** quindi un segmento di codice come *if (a instanceof ArrayList<String>)* è un errore in quanto non ho più controllo sul tipo;
- Le variabili di tipo non possono essere inizializzate (es. *T x = new T()* //ERRORE);
- Gli array con tipo parametrico non possono essere allocati in memoria perché non saprei quanto spazio allocare in memoria in quanto non conosco il tipo;
- Le variabili di tipo non possono essere usate in un contesto statico.

I principi Solid

I principi **SOLID** sono linee guida per lo sviluppo di software **leggibile**, **estendibile** e **manutenibile**, in particolare nel contesto di pratiche di sviluppo agili e fondate sull'identificazione di *code smell* e sul *refactoring*.

Single Responsibility Principle (SRP)

Ogni classe dovrebbe avere una ed una sola responsabilità, interamente incapsulata al suo interno

Una responsabilità è una singola famiglia di funzionalità che servono ad un particolare **attore**. Nella prima fase della realizzazione del progetto è fondamentale individuare le responsabilità di ogni elemento o componente dell'applicazione. Quindi quando sviluppiamo una soluzione software dobbiamo:

- individuare gli **attori**;
- identificare le **responsabilità** di questi attori;
- raggruppare le funzionalità nelle differenti interfacce/classi in modo che ognuna abbia una singola responsabilità.

Secondo questo principio, è buona norma **creare classi, strutture o funzioni che svolgono una sola operazione**. Questo non implica che le classi debbano contenere un unico metodo, ma che i ***metodi servono a svolgere una singola e specifica funzionalità***. In questo modo, le classi saranno più semplici e chiare e potranno combinarsi per svolgere compiti più complessi, rimanendo indipendenti l'una dall'altra.

L'applicazione di questo principio, inoltre, ci aiuta a strutturare meglio le classi per l'incapsulamento, in quanto le classi sono molto ridotte e semplici da gestire.

Open/Closed Principle (OCP)

Un'entità software dovrebbe essere aperta alle estensioni, ma chiusa alle modifiche

Un'entità software dovrebbe essere aperta alle estensioni, ma chiusa alle modifiche. Ciò significa che è possibile cambiare il comportamento di una componente software senza dover modificare il codice sorgente che rimane nascosto. Un modulo è detto:

- **aperto** se questo permette le estensioni;
- **chiuso** se è disponibile per l'uso da parte di altre componenti senza la necessità di conoscerne l'esatta implementazione.

Per aggiungere nuove funzionalità ad una classe, questo principio consiglia di **utilizzare il polimorfismo**, cioè la possibilità di creare una nuova classe che estenda le funzionalità di una classe di base, lasciando inalterata la classe base. In questo modo la classe base è sia *aperta* a nuove funzionalità tramite estensione e sia *chiusa* alle modifiche.

Per cui, invece di modificare la classe base per aggiungere le nuove funzionalità, crei una nuova classe che derivi dalla classe base e sei sicuro di non intaccare il funzionamento sia della classe base sia delle altre classi derivate. Ovviamente ciò non esclude la possibilità di modificare una classe base nel caso in cui siano presenti bug.

Liskov Substitution Principle (LSP)

Gli oggetti dovrebbero poter essere sostituiti con dei loro sottotipi, senza alterare il comportamento del software che li utilizza

Gli oggetti dovrebbero poter essere sostituiti con dei loro sottotipi, senza alterare il comportamento del programma che li utilizza. Tale principio è una particolare definizione di *sottotipo*.

> Sia $q(x)$ una proprietà verificata da tutti gli oggetti x di tipo T. Allora $q(y)$ sarà verificata da tutti gli oggetti y di tipo S dove S è un sottotipo di T.

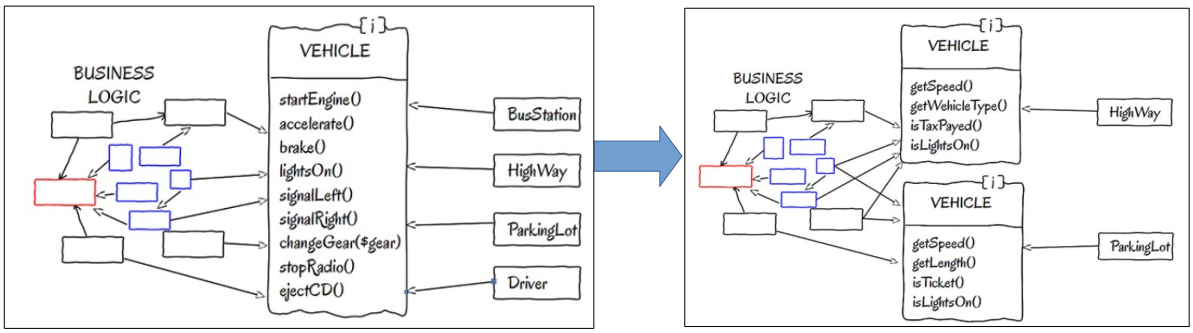
In base a questo principio, **una classe dovrebbe poter essere sostituita da qualsiasi classe derivata**, mantenendo il codice perfettamente funzionante. In altre parole, una classe derivata che mantiene tutte le caratteristiche della classe base (eccetto le sue funzioni specifiche) può essere utilizzata anche in sostituzione della classe base da cui deriva, senza ulteriori modifiche. Per cui, si possono utilizzare le nuove classi derivate al posto delle classi base senza dover modificare il codice esistente.

Interface Segregation Principle (ISP)

Sarebbero preferibili più interfacce specifiche, che una singola generica

È consigliabile, **suddividere le interfacce per scopi specifici** (come suggerito anche nel primo principio). In tal modo, le classi possono implementare solo le interfacce di cui hanno necessariamente bisogno, evitando la dichiarazione di metodi vuoti e inutili. Per cui, il codice viene ridotto notevolmente, riducendo, di conseguenza, i tempi di sviluppo e la probabilità di presenza di bug.

N.B.
Non devo mai avere interfacce grandissime dove alcuni metodi non vengono utilizzati da nessuna classe.



Dependency Inversion Principle

> *I moduli di alto livello non devono dipendere da quelli di basso livello, entrambi devono dipendere da astrazioni; Le astrazioni non devono dipendere dai dettagli, sono i dettagli che dipendono dalle astrazioni*

I moduli di alto livello sono le **classi che realizzano le proprie funzioni facendo uso dei moduli di basso livello**, attraverso le interfacce esposte da questi ultimi.

Il principio di inversione delle dipendenze cerca di **disaccoppiare quanto più possibile i due moduli, connettendoli entrambi tramite astrazioni**. In questo modo, i moduli di alto livello usano determinate astrazioni, mentre i moduli di basso livello le implementano.

Code Smells

Un **code smell** rappresenta una caratteristica del codice di un programma che indica la possibile presenza di un problema serio. Sono **cattive pratiche di programmazione** che rendono il codice inusabile o particolarmente pericoloso.

Determinare un code smell è soggettivo, e varia da linguaggio, sviluppatore e metodologia di sviluppo. Ma esistono dei criteri generali da seguire:

- **Metodi troppo lunghi**: la presenza di un metodo che è più lungo di **10 righe**. Questo indica una cattiva organizzazione del codice, una probabilità di contenere codice ripetuto e renderà il codice difficile da testare e manutenere. Inoltre, se per spiegare il codice ho bisogno di aggiungere commenti all'interno del metodo è segno che il codice sia scritto male.
  La cosa più semplice è dividere il metodo. Utilizzare dei ***metodi semplici, con poche righe di codice***, che abbiano un nome semplice e significativo. Inoltre è bene fattorizzare il codice creando metodi di utilità.
- **Classi troppo grandi**: una classe che ha molti campi, metodi e linee è quasi sicuramente scritta nel modo sbagliato in quanto potrebbe avere **troppe responsabilità** e ciò viola il primo principio SOLID.
  In questo caso è bene dividere la classe spostando alcuni comportamenti fuori dalla classe, estraendo sottoclassi o interfacce e utilizzando, quando possibile, la composizione tra oggetti.

- **Ossessione dei tipi primitivi**: l'uso di tipi primitivi, costanti o loro array per rappresentare informazioni strutturate. Per far emergere un concetto non è sempre corretto utilizzare un tipo primitivo.
- **Lunga lista di parametri**: metodi con più di tre/quattro parametri indicano che sto rappresentando male le informazioni. E' bene aggregarli in una singola classe che contiene tutte le informazioni che mi servono.
- **Data Clumps**: l'uso di un gruppo di variabili distinte che hanno una dipendenza tra di loro. E' bene combinarli in un unico oggetto.
- **Campi temporanei**: l'uso di campi in una classe che sono utilizzati solo in alcune circostanze altrimenti sono vuoti o ignorati. Questi campi vengono utilizzati solo nell'algoritmo e rimangono inutilizzati per il resto del tempo.
- **Eredità rifiutata**: se una sottoclasse utilizza solo alcuni dei metodi e delle proprietà ereditati dai suoi genitori, la gerarchia è fuori controllo. I metodi non necessari possono semplicemente non essere utilizzati o essere ridefiniti e generare eccezioni. Se l'ereditarietà non ha senso e la sottoclasse non ha davvero nulla in comune con la superclasse, elimina l'ereditarietà. Altrimenti, se l'ereditarietà è appropriata, eliminare i campi e i metodi non necessari nella sottoclasse.
- **Classi diverse, stessa interfaccia**: due classi svolgono funzioni identiche ma hanno nomi di metodi diversi. Il programmatore che ha creato una delle classi probabilmente non sapeva che esisteva già una classe funzionalmente equivalente. E' bene rinominare i metodi per renderli identici in tutte le classi, oppure rendere le firme dei metodi e le implementazioni identiche.
- **Modifiche a cascata**: modificare molti metodi non correlati quando apporti modifiche a una classe. Ad esempio, quando si aggiunge un nuovo tipo di prodotto è necessario modificare i metodi per trovare, visualizzare e ordinare i prodotti. Spesso queste modifiche divergenti sono dovute alla scarsa struttura del programma o alla "programmazione copia-incolla".
  Per migliorare il codice è bene dividere il comportamento delle classi, ad esempio estraendo e creando nuove classi. Se classi diverse hanno lo stesso comportamento, potresti voler combinare le classi tramite l'ereditarietà.
- **Gerarchie parallele**: ogni volta che crei una sottoclasse per una classe, ti ritrovi a dover creare una sottoclasse per un'altra classe. Con l'aggiunta di nuove classi, apportare modifiche diventa sempre più difficile. Spesso questo è dipeso da una violazione dell'ultimo principio SOLID.
  È possibile deduplicare le gerarchie di classi parallele in due passaggi. Innanzitutto, fare in modo che le istanze di una gerarchia si riferiscano a istanze di un'altra gerarchia. Quindi, rimuovere la gerarchia nella classe di riferimento.

- **Commenti eccessivi**: un metodo è pieno di commenti esplicativi. I commenti vengono solitamente creati con le migliori intenzioni, quando l'autore si rende conto che il suo codice non è intuitivo o ovvio. In questi casi, i commenti sono come un deodorante che maschera il codice che potrebbe essere migliorato. Il miglior commento è un buon nome per un metodo o una classe.
  Se ritieni che un frammento di codice non possa essere compreso senza commenti, prova a modificare la struttura del codice in modo da rendere superflui i commenti.
- **Codice duplicato**: dei frammenti di codice che sembrano quasi identici.
- **Codice morto**: una variabile, un parametro, un campo, un metodo o una classe non vengono più utilizzati.
- **Invidia**: un metodo utilizza maggiormente i dati di un altro oggetto che non i propri. Come regola di base, se le cose cambiano contemporaneamente, dovresti tenerle nello stesso posto. Di solito i dati e le funzioni che utilizzano questi dati vengono modificati insieme.
- **Intimità inappropriata**: una classe utilizza i campi interni e i metodi di un'altra classe. Tipicamente le classi dovrebbero "conoscersi" il meno possibile in modo da renderle più facili da mantenere e riutilizzare.

Lambda expression

Una ***lambda expression*** è un breve blocco di codice che accetta parametri e restituisce un valore. Le espressioni Lambda sono simili ai metodi, ma non necessitano di un nome e possono essere implementate direttamente nel corpo di un metodo.

Possono essere definite come sequenze di parametri seguiti da un espressione:

(String first, String second) --> first.length() - second.length();

o da blocchi:

```
(String first, String second) −> {
        int difference = first.length()−second.length();
        if(difference<0) return −1;
        elseif(difference>0) return 1;
        else return 0;
}
```

Le espressioni lambda forniscono un modo conciso di scrivere codice per implementare interfacce funzionali (interfacce con un solo metodo astratto).

Nell'esempio troviamo la versione "classica" di come viene scritta una classe utilizzando le classi annidate, in cui viene associato all'oggetto *FactoryRegistry* l'istanza della classe i cui metodi sono descritti all'interno di quest'ultima. Il vantaggio di questa operazione è la possibilità delle classi annidate di accedere anche ai campi privati della classe che la contiene. Nel secondo caso troviamo lo stesso esempio fatto con una *lambda expression*. In questo caso viene costruito un *FactoryRegistry* a cui viene passato un *id* e un *ledger* e la funzione mi restituisce un nuovo *LedgerTransaction.*
Nell'ultimo caso, costruisco un *FactoryRegistry* passandogli direttamente il costruttore della classe *LedgerTransaction*.

Functional interface

Le **interfacce funzionali** sono un concetto chiave per supportare le espressioni lambda e il paradigma di programmazione funzionale. Un'interfaccia funzionale è un'interfaccia Java che contiene un solo metodo astratto. Questo metodo astratto rappresenta l'unico comportamento richiesto dall'interfaccia. Le interfacce funzionali possono avere anche altri metodi non astratti (metodi di default o statici), ma devono comunque avere esattamente un metodo astratto. L'utilità principale delle interfacce funzionali è che possono essere utilizzate insieme alle espressioni lambda per scrivere codice più conciso e leggibile.

Runnable, *ActionListener*, *Comparable* sono alcuni degli esempi di interfacce funzionali.

```java
@FunctionalInterface
interface OperazioneMatematica {
    int eseguiOperazione(int a, int b);
}
public class Main {
    public static void main(String[] args) {
        // Utilizzo di una lambda expression per l'addizione
        OperazioneMatematica addizione = (a, b) -> a + b;
        System.out.println("Addizione: " + addizione.eseguiOperazione(5, 3));

        // Utilizzo di una lambda expression per la sottrazione
        OperazioneMatematica sottrazione = (a, b) -> a - b;
        System.out.println("Sottrazione: " + sottrazione.eseguiOperazione(5, 3));

        // Utilizzo di una lambda expression per la moltiplicazione
        OperazioneMatematica moltiplicazione = (a, b) -> a * b;
        System.out.println("Moltiplicazione: " + moltiplicazione.eseguiOperazione(5, 3));
    }
}
```

Per definire una interfaccia funzionale dobbiamo adottare la notazione *@FunctionalInterface*.

In Java, con un'espressione lambda è possibile svolgere un unico compito, ovvero inserirla in una variabile il cui tipo sia un'interfaccia funzionale, in modo che sia convertita in un'istanza di detta interfaccia.

Method references

Possiamo utilizzare i metodi come se fossero oggetti o valori primitivi e possiamo trattarli come una variabile.

Un riferimento al metodo è la sintassi abbreviata per un'espressione lambda che contiene solo una chiamata al metodo. I riferimenti al metodo possono essere usati solo per sostituire un singolo metodo dell'espressione lambda. In generale, non è necessario passare argomenti ai riferimenti dei metodi.

```java
List<Integer> list = new ArrayList<>();
// Add some element to list
// USING AN ANONYMOUS CLASS
transformAndAdd(list, new Function<Integer, Integer>(){
  public Integer apply(Integer i){
    return OpsUtil.doHalf(i);
  }
});

//USING LAMBDA EXPRESSION
transformAndAdd(list, i -> OpsUtil.doHalf(i));

// USING A METHOD REFERENCE
transformAndAdd(list, OpsUtil::doHalf);
```

Ci sono tre varianti di riferimenti al metodo: **Class::instanceMethod**, **Class::staticMethod**, **object::instanceMethod**, **Class::new**.

Streams

Gli **Streams** rappresentano una ***sequenza di elementi sui cui eseguire operazioni*** intermedie o terminali. Le terminali generano un risultato di un determinato tipo mentre le intermedie generano un nuovo Stream su cui invocare altre operazioni dando vita ad una catena di chiamate a metodi.

Gli Streams sono creati a partire da una sorgente che può essere rappresentata da una qualsiasi classe compatibile con l'interfaccia *java.util.Collection*. Le mappe non sono supportate. Un aspetto interessante riguarda la possibilità di eseguire operazioni su uno Stream in modo sequenziale o parallelo.

Uno stream assomiglia a una collezione, in quanto consente di trasformare e recuperare i dati, ma vi sono alcune differenze:

- gli stream non salvano nulla in memoria perché **creano una vista** dei dati e permettono di accedere in maniera efficiente alle informazioni senza dover alterare i dati;
- gli stream **non cambiano la loro sorgente**, ogni operazione su uno stream crea un altro stream;
- gli stream sono **lazy**, ovvero il valore viene calcolato solo quando ce n'è bisogno. Per esempio, se si richiedono solamente le prime cinque parole, invece di tutte quante, il metodo *filter* smetterà di filtrare parole dopo la quinta corrispondenza, il che porta con sé la conseguenza che si possono avere anche stream infiniti.

Quindi il workflow è del tipo:

1. crea uno *stream*;
2. specifica delle **operazioni immediate** per trasformare lo *stream* in un altro *stream* (come ad esempio *filter, map, …*), eventualmente in più passaggi;

```java
// Creazione di una lista di stringhe
List<String> frutta =
    Arrays.asList("mela", "banana", "arancia", "ananas", "uva");

    // Utilizzo degli Stream per filtrare e stampare le stringhe
    // che iniziano con la lettera 'a'
    frutta.stream()
          .filter(s -> s.startsWith("a"))
          .forEach(System.out::println);
```

3. applica infine un'**operazione terminale** per produrre un risultato. Dopo questa operazione lo *stream* non è più utilizzabile.

Stream creation

Uno stream non è mai creato direttamente ma saranno le strutture dati a crearlo.

L'interfaccia *Collection* mette a disposizione un metodo *stream()* che trasforma qualsiasi collezione in uno stream. Se si ha un array, si usa invece il metodo statico *Stream.of()*.

```
Stream<T> generate(Supplier<? extends T> s)

Stream<T> Stream.iterate(T seed,
  UnaryOperator<T> f)

Stream<T> Stream.iterate(T seed,
  Predicate<? super T> hasNext,
  UnaryOperator<T> next)
```

L'interfaccia *Stream* ha due metodi statici per la creazione di stream infiniti. Il metodo *generate()* prende una funzione priva di argomenti. Per produrre sequenze infinite, come 0 1 2 3 …, si usa *iterate()*, che prende un valore e una funzione e applica ripetutamente la funzione al risultato precedente.

Stream transformation

Come detto prima, uno stream non viene modificato, ma viene trasformato in un altro stream avente elementi derivati dal primo stream. Questa trasformazione è logica e non applicata realmente. Ad esempio:

- ***filter()***: viene utilizzata per selezionare solo gli elementi che soddisfano una determinata condizione. Questa operazione restituisce uno Stream che contiene solo gli elementi che superano il filtro.

N.B.
L'aggiunta di **"? super T, ? extends R"**, indica che voglio rappresentare una qualsiasi funzione da T ad R. Scrivendo invece solo **"T, R"** si fissano invece solo funzioni che prendono **T** come ingresso e restituiscono **R**.
Per utilizzare quindi una funzione che sia ***sottotipo*** di questa devo prendere una funzione il cui argomento è ***sovratipo*** di T (ecco perché **? super T**) e produrre qualcosa che sia sottotipo di R (ecco perché **? extends R**)

Stream<T> stream.filter(Predicate<? super T> predicate);

- ***map()***: viene utilizzata per applicare una funzione a ciascun elemento dello stream e ottenere uno stream risultante contenente i risultati delle applicazioni della funzione. Spesso si utilizzano espressioni lambda.

Stream<S> stream.map(Function<? super T, ? extends R> mapper);

- ***flatMap()***: viene utilizzata per gestire situazioni in cui abbiamo uno Stream di elementi e vogliamo mappare ciascun elemento in zero o più elementi, combinando tutti i risultati in un unico Stream. È spesso utilizzato per gestire situazioni in cui uno Stream contiene Stream annidati.

Stream<R> stream.flatMap(Function<? Super T, ? extends Steam<? extends R>> mapper);

Estrazione di substream e combinazione di streams

Dato uno *stream* è possibile estrarre dei *substream*:

- ***limit(n)***: può essere utilizzato per selezionare solo i primi *n* elementi dello *stream*

  Stream<T> limit(long n);

- ***skip(n)***: può essere utilizzato per ignorare i primi *n* elementi dello *stream*

  Stream<T> skip(long n);

- ***takeWhile()*** e ***dropWhile()***: seleziona o scarta tutti gli elementi dello stream fino a che un certo predicato non è soddisfatto

  Stream<T> takeWhile(Predicate<? Super T> predicate);

  Stream<T> dropWhile(Predicate<? Super T> predicate);

- ***concat()***: metodo statico che combina due *stream*. Ovviamente, il primo stream non deve essere infinito

  Stream<T> concat(Stream<? extends T> a, Stream<? Extends T> b);

Altre trasformazioni degli stream

- ***distinct()***: rimozione di duplicati da uno *stream* [***Stream<T> distinct();***]
- ***sorted()***: riordinamento di uno *stream*

  Stream<T> sorted(); //valido se T implements Comparable<T>

  Stream<T> sorted(Comparator<? Super T> comparator);

- ***peek()***: prende un *Consumer* che è una interfaccia funzionale che prende un elemento e "ci fa qualche cosa" (es. per stampare qualcosa utilizzo *System.out::println()* è un Consumer di stringhe perché è un metodo che prende una stringa e restituisce *void*).

  Stream<T> peek (Consumer<? super T> action);

Tipo *Optional*

Un **Optional<T>** è un contenitore per un valore i tipo T e il valore può esserci o meno. Il tipo *Optional* da la possibilità di produrre, dall'esecuzione di un metodo, due alternative: un valore, se presente, oppure nulla. Ci mette a disposizione degli *utility methods*:

- ***orElse(T other)***: restituisce il valore memorizzato nell'*Optional* se quel valore esiste, altrimenti mi restituisce il valore che gli ho passato come argomento;
- ***orElseGet(Supplier<? extends T> other)***: restituisce il valore opzionale se presente, altrimenti restituisce il risultato della valutazione fornita dalla funzione *Supplier*;
- ***orElseThrow(Supplier<? extends X> exceptionSupplier)***: restituisce il valore opzionale se presente, altrimenti lancia un'eccezione fornita dalla funzione *Supplier*;
- ***ifPresent(Consumer<? super T> consumer)***: esegue l'azione specificata solo se il valore opzionale è presente;
- ***ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)***: esegue l'azione specificata solo se il valore opzionale è presente, altrimenti eseguo un'altra azione.

```java
// Esempio orElse
Optional<String> optionalValue = Optional.empty();
String result = optionalValue.orElse("Default Value");
System.out.println("Result: " + result);

// Esempio orElseGet
Optional<String> optionalValue = Optional.empty();
String result = optionalValue.orElseGet(() -> "Default Value");
System.out.println("Result: " + result);

// Esempio orElseThrow
Optional<String> optionalValue = Optional.empty();
try {
    String result = optionalValue.orElseThrow(() -> new RuntimeException("Value is absent"));
    System.out.println("Result: " + result);
} catch (RuntimeException e) {
    System.out.println("Caught exception: " + e.getMessage());
}

// Esempio ifPresent
Optional<String> optionalValue = Optional.of("Hello");
optionalValue.ifPresent(value -> System.out.println("Value is present: " + value));
```

Reductions

Le riduzioni sono **operazioni terminali** che trasformano lo *stream* in un valore utilizzabile o in un primitivo. Un'operazione di riduzione può essere vista come un modo di combinare gli elementi di uno stream, ad esempio, per calcolare la somma di tutti gli elementi o trovare il massimo valore. L'API Java 8 Stream contiene una serie di operazioni di riduzione predefinite:

- ***max()***: restituisce il valore massimo dello *stream* secondo un dato *Comparator*

  Optional<T> max (Comparator <? super T> comparator);

- ***min()***: restituisce il valore minimo dello *stream* secondo un dato *Comparator*

  Optional<T> min (Comparator <? super T> comparator);

- ***findFirst()***: restituisce il primo elemento dello *stream*

  Optional<T> findFirst();

- ***findAny()***: restituisce la prima computazione a terminare

  Optional<T> findAny();

- ***reduce()***: combina gli elementi di uno stream in un unico risultato. Esso prende un accumulatore e una funzione di combinazione come argomenti e restituisce un valore risultante. I concetti chiave del metodo sono:
  - **identity**: un elemento che rappresenta il valore iniziale dell'operazione di riduzione e il risultato predefinito se il flusso è vuoto. In altre parole, "*da cosa cominciare*";
  - **accumulator**: una funzione che accetta due parametri: un risultato parziale dell'operazione di riduzione e l'elemento successivo del flusso.

  T reduce(T identity, BinaryOperator<T> accumulator);

  L'operazione di riduzione inizia con il valore iniziale (*identity*) dell'accumulatore, che viene utilizzato per combinare ogni elemento dello stream con il risultato corrente. La funzione *accumulator* viene applicata iterativamente a ciascun elemento dello stream e all'accumulatore corrente. Il risultato finale è il valore dell'accumulatore, che rappresenta la riduzione completa dello stream.

  Quando si lavora con stream paralleli in Java e si utilizza il metodo *reduce* con un *identity*, *accumulator* e *combiner*, si stanno gestendo aspetti di parallelismo. In situazioni parallele, è necessario considerare come combinare i risultati parziali ottenuti da diverse porzioni dello stream. Un ***combiner*** è una funzione utilizzata per combinare il risultato parziale dell'operazione di riduzione quando la riduzione è parallelizzata o quando c'è una mancata corrispondenza tra i tipi degli argomenti dell'accumulatore e i tipi dell'implementazione dell'accumulatore.

La firma del metodo reduce in questo contesto è la seguente:

U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);

Nota che reduce con *combiner* è utile quando si lavora con stream paralleli. La funzione *combiner* viene utilizzata per combinare i risultati parziali ottenuti da diverse porzioni dello stream parallelizzato. In uno stream sequenziale, il *combiner* non viene chiamato, e l'*identity* e l'*accumulator* sono sufficienti.

Raccolta dei risultati

Quando si termina di lavorare con uno stream, spesso è necessario vederne i risultati. A questo scopo è possibile:

- chiamare il metodo ***forEach()*** per applicare una funzione a ciascun elemento.

  void forEach(Consumer<? Super T> action);

  Su uno stream parallelo il metodo attraversa gli elementi in ordine arbitrario. Per iterarli nell'ordine in cui appaiono è possibile chiamare il metodo ***forEachOrdered()***;

- se i risultati devono essere raccolti in un array, è possibile chiamare il metodo ***toArray()*** e ottenere un array degli elementi dello stream. Il metodo restituisce un tipo *Object[]*. Per ottenere un array del tipo corretto è necessario passare il tipo al costruttore dell'array.

  <T> T[] toArray(IntFunction<T[]> generator);

- chiamate il metodo ***collect()*** che riceve un'istanza dell'interfaccia *Collection*. Questo metodo converte gli elementi dello stream in una diversa forma, spesso in una collezione o un'altra struttura dati. Il metodo *collect()* accetta un *Collector*, che definisce come gli elementi dello stream devono essere raccolti.

  <R, A> R collect(Collector<? super T, A, R> collector);

  Dove:

  - **T** è il tipo degli elementi dello stream;
  - **A** è il tipo del risultato intermedio accumulato durante il processo di raccolta (l'accumulatore);
  - **R** è il tipo del risultato finale ottenuto dal processo di raccolta.

  Il parametro *collector* è un oggetto della interfaccia *Collector* che definisce come gli elementi dello stream devono essere raccolti. L'interfaccia *Collector* ha vari metodi di fabbrica che restituiscono implementazioni specifiche di *Collector* per diverse operazioni di raccolta. Alcuni dei metodi sono:

  - ***Collector.joining()***: restituisce un *Collector* che concatenata gli elementi dello stream in una singola stringa. È spesso usato per concatenare stringhe con uno delimitatore;

- ***Collectors.toList()***: restituisce un *Collector* che accumula gli elementi dello stream in una *List*;
- ***Collectors.toSet()***: restituisce un *Collector* che accumula gli elementi dello stream in un *Set*.

Parallel Streams

Utilizzando **Stream paralleli**, possiamo dividere il codice in più flussi che vengono eseguiti in parallelo su core separati e il risultato finale è la combinazione dei singoli risultati.

L'ordine di esecuzione, tuttavia, non è sotto il nostro controllo. Pertanto, è consigliabile utilizzare flussi paralleli nei casi in cui, indipendentemente dall'ordine di esecuzione, il risultato non viene modificato e lo stato di un elemento non influisce sull'altro e anche la fonte dei dati rimane inalterata.

E' possibile creare un *parallel stream* utilizzando il metodo:

- ***parallelStream()*** su una raccolta (interfaccia *Collection*): restituisce un possibile flusso parallelo con la raccolta come origine;
- ***parallel()*** su uno *Stream* (interfaccia *BaseStream*): restituisce un flusso parallelo equivalente.

Per far si che i risultati ottenuti con *Stream* paralleli siano identici a quelli ottenuti in maniera sequenziale, tutte le operazioni devono essere **stateless**, ovvero non devono avere effetti sullo stato.

Es.

```
int [] shortWords = new int[12];
words.parallelStream().forEach(
  s -> {
    if(s.length()<12)
      shortWords[s.length()]++
    }
);
```

L'idea del codice è che ogni volta che, nel *parallelStream* di *words*, viene trovata una parola più lunga di dodici caratteri, incremento il contatore. Ma questa operazione crea confusione perché due attività che vengono svolte sulla stessa cella di memoria potrebbero creare problemi. Infatti i *thread* nei nostri computer lavorano principalmente con delle copie dell'area di memoria. Perciò si creano più copie di aree di memoria che, una volta finito di processare, vengono copiate nella memoria principale. Questa situazione crea una **race condition**, ovvero una condizione in cui più *thread* operano sulla stessa area di memoria.

```
Map<Integer, Long> shortWordCounts =
words.parallelStream()
  .filter(s -> s.length()<12)
  .collect(groupingBy(
    String::length, counting()
  ));
```

Nel secondo caso invece limito il numero di parole da controllare a quelle con una lunghezza minore di 12 (*filter()*), e colleziono gli elementi con un collettore particolare che è ***groupingBy*** che prende, per ogni elemento, la lunghezza delle stringhe e il metodo ***counting()***.
In questo modo, il codice crea una mappa che associa ad ogni intero, il numero degli elementi con quella data occorrenza.
Es.
<1,3> --> n°1 parola avente 3 elementi
<2,5> --> n°2 parole aventi 5 elementi

Concurrent Programming

La programmazione concorrente in Java si riferisce alla scrittura di programmi che possono eseguire **più attività in parallelo**. Questo è particolarmente utile per migliorare le prestazioni del software e sfruttare i sistemi multi-core, consentendo a più thread di eseguire operazioni simultaneamente.

Ogni programma che viene mandato in esecuzione sulla JVM dà origine ad un processo.

Per la gestione del **multithreading** la JVM non si affida al sistema operativo ed utilizza una politica di gestione dei thread di tipo ***fixed-priority scheduling***. Questa è basata essenzialmente sulla priorità, che si riferisce al livello di importanza o di urgenza associato a un processo. Un processo con una priorità più alta sarà eseguito prima di uno con una priorità più bassa, ed è preventiva poiché garantisce che, se in qualunque momento si rende eseguibile un thread con priorità maggiore di quella del thread attualmente in esecuzione, il thread a maggiore priorità prevale sull'altro, fatto salvo casi particolari in cui debbono essere gestite potenziali situazioni di stallo.

Definizioni

Thread: sono le più piccole unità di elaborazione all'interno di un processo. I thread condividono lo stesso spazio di indirizzi e risorse di un processo ma eseguono in modo indipendente all'interno di esso. Consentono l'esecuzione simultanea di più attività all'interno di un singolo processo. I thread all'interno dello stesso processo condividono lo stesso contesto, compreso lo spazio di indirizzi e i descrittori di file.

Binaries: sono programmi dormienti che risiedono su un supporto di memorizzazione, compilati in un formato accessibile da un dato sistema operativo, pronti per essere eseguiti.

Processo: è un'istanza di un programma in esecuzione in un sistema informatico. Sono l'astrazione del sistema operativo che rappresentano quei binari in azione (es. il binario caricato, la memoria virtualizzata, le risorse del kernel come i file aperti, un utente associato e così via). Un processo può contenere uno (***single thread***) o più *thread* (***multithread***).
Ogni processo ha il proprio spazio di indirizzi, memoria e risorse assegnate, rendendolo isolato da altri processi.

Il primo step per sviluppare seguendo la programmazione concorrente è quello di dividere le attività in ***task***.

Executor e ExecutorService

Executor è un'interfaccia base nel package **java.util.concurrent** di Java che dichiara un solo metodo:

void execute(Runnable command);

L'implementazione di questo metodo è responsabile di eseguire l'attività specificata. Fornisce una forma più semplice di gestione delle attività rispetto a *ExecutorService*. Non offre funzionalità avanzate come la gestione di risultati futuri o il controllo dello stato dell'esecuzione.

Un **ExecutorService** estende l'interfaccia *Executor* e offre metodi aggiuntivi per la gestione avanzata delle attività. Consente di eseguire operazioni asincrone e di gestire un pool di thread in modo più efficiente. L'esecuzione del *task* quindi partirà indipendentemente dal flusso corrente.

N.B.

Executor e **ExecutorService** sono due interfacce correlate in Java, entrambe appartenenti al package java.util.concurrent. Entrambe forniscono un'astrazione per l'esecuzione di attività in modo asincrono, ma ci sono alcune differenze chiave tra di loro.

Mette a disposizione dei ***factory methods*** per creare istanze di ExecutorService preconfigurate:

- ***static ExecutorService newCachedThreadPool()*** : restituisce un *ExecutorService* che crea nuovi thread se tutti i thread attuali sono occupati e li elimina se sono inattivi per un certo periodo di tempo. È utile quando si hanno molte attività brevi o quando la quantità di lavoro varia dinamicamente. Il numero massimo di thread può crescere o diminuire dinamicamente.
- ***static ExecutorService newFixedThreadPool(int nThreads)*** : restituisce un *ExecutorService* che mantiene un numero fisso di thread nel pool. È adatto quando si desidera controllare il numero massimo di thread in esecuzione simultanea e si ha una quantità nota e costante di lavoro. Può essere più efficiente rispetto a newCachedThreadPool in scenari in cui la creazione e la distruzione frequente di thread è costosa.

Inoltre, fornisce metodi aggiuntivi per la gestione e il controllo delle attività eseguite da un pool di thread:

- ***submit()***: invia un'attività *Callable* o *Runnable* a un *ExecutorService* e restituisce un risultato di tipo *Future*;
- ***invokeAny()***: assegna una raccolta di attività a un *ExecutorService* , provocandone l'esecuzione e restituisce il risultato di un'esecuzione riuscita di un'attività (se si è verificata un'esecuzione riuscita);
- ***invokeAll()***: assegna una raccolta di attività a un *ExecutorService* , provocandone l'esecuzione e restituisce il risultato di tutte le esecuzioni delle attività sotto forma di un elenco di oggetti di tipo *Future*;

In generale, *ExecutorService* non verrà distrutto automaticamente quando non ci sono attività da elaborare. Rimarrà in vita e aspetterà nuovo lavoro da svolgere.

Per chiudere correttamente un *ExecutorService* , abbiamo i metodi:

- ***shutdown()***: non causa la distruzione immediata di ExecutorService . Farà sì che ExecutorService smetta di accettare nuove attività e si arresti dopo che tutti i thread in esecuzione avranno terminato il loro lavoro corrente;
- ***shutdownNow()***: tenta di distruggere immediatamente ExecutorService , ma non garantisce che tutti i thread in esecuzione verranno interrotti contemporaneamente.

Un buon modo per chiudere *ExecutorService* è utilizzare entrambi questi metodi combinati con il metodo ***waitTermination()*** che smetterà di accettare nuove attività e quindi attenderà fino a un periodo di tempo specificato per il completamento di tutte le attività. Se tale tempo scade, l'esecuzione viene interrotta immediatamente.

Runnable

Java mette a disposizione l'interfaccia **Runnable**, un interfaccia funzionale che viene utilizzata per definire una singola unità di lavoro eseguibile, generalmente da un thread. L'implementazione di questa interfaccia consente di separare il codice eseguibile dalla logica del thread, fornendo un modo pulito e flessibile per definire attività concorrenti.

Il metodo ***run()*** rappresenta il punto di ingresso del codice eseguibile che verrà eseguito quando il thread viene avviato.

È utilizzata per situazioni in cui si vuole cercare un risultato dell'esecuzione del thread. Una task quindi può essere eseguita tramite uno specifico thread, o tramite un ***executor***.

Inoltre è possibile creare dei thread personalizzati attraverso la classe ***ThreadFactory***, un'interfaccia funzionale e viene utilizzata per creare nuovi oggetti thread. Consente di definire una politica per la creazione e la personalizzazione dei thread all'interno di un'applicazione.

Callable

L'interfaccia **Callable** è un'interfaccia funzionale che fa parte del package *java.util.concurrent* e viene utilizzata per rappresentare una computazione che restituisce un risultato e può sollevare un'eccezione. A differenza dell'interfaccia *Runnable*, che rappresenta un'attività senza restituzione di risultati o gestione di eccezioni, *Callable* consente di eseguire un'operazione che restituisce un valore e può generare eccezioni.

```
@Override
public String call() throws Exception {
    // Simuliamo un'attività che restituisce
    //una stringa dopo un certo tempo
    Thread.sleep(2000);
    return "Task completed successfully!";
}

public static void main(String[] args) {
    // Creazione di un'istanza di Callable
    Callable<String> myCallable = new MyCallable();

    // Chiamata al metodo call()
    String result = myCallable.call();
}
```

L'interfaccia *Callable* implementa un solo metodo ***call()*** che rappresenta il punto di ingresso per l'esecuzione della computazione e restituisce un risultato. Può anche dichiarare di sollevare un'eccezione di tipo Exception.

Tramite l'*ExecutorService* possiamo inviare un oggetto *Callable*. Il risultato dell'operazione è di tipo ***Future***, ovvero un oggetto che rappresenterà il risultato dell'esecuzione in un qualsiasi tempo futuro. E' lo strumento che viene utilizzato per permettere di far interagire chi chiama il servizio con il thread che si occupa di eseguire il servizio.

```java
// Creazione di un ExecutorService (nel nostro caso, un pool di thread fisso)
ExecutorService executorService = Executors.newFixedThreadPool(1);

// Creazione di un'istanza di Callable
Callable<String> myCallable = () -> {
    // Simuliamo un'attività che restituisce una stringa dopo un certo tempo
    Thread.sleep(2000);
    return "Task completed successfully!";
};

// Sottomissione di Callable al pool di thread e ottenere un Future per il risultato
Future<String> futureResult = executorService.submit(myCallable);

// Attendi il risultato e stampalo
String result = futureResult.get();
System.out.println(result);
```

N.B.
Il metodo *submit()* invia un *Callable* per essere eseguito

Future

L'interfaccia **Future** è parte del package *java.util.concurrent* e rappresenta il risultato di un'operazione asincrona. Essenzialmente, un oggetto Future è un contenitore per un valore che può non essere ancora disponibile. Fornisce metodi per controllare lo stato dell'operazione e ottenere il risultato quando è disponibile.

I metodi che mette a disposizione sono:

- ***V get() throws InterruptException, ExecutionException*** : restituisce il valore, quando disponibile, altrimenti viene lanciata un'eccezione. Inoltre, blocca il thread chiamante fino a quando il risultato non è pronto.;
- ***V get(long timeout, TimeUnit unit) throws InterruptException, ExecutionException, TimeoutException*** : restituisce il risultato dell'operazione quando è disponibile, aspettando al massimo il tempo specificato. Il thread chiamante viene bloccato fino a quando il risultato non è pronto o scade il timeout;
- ***boolean cancel(boolean mayInterruptIfRunning)*** : Cerca di annullare l'operazione associata all'oggetto *Future*. Se l'operazione non è ancora iniziata, può essere annullata. Se è già in corso, il comportamento dipende dal valore di *mayInterruptIfRunning*. Se *mayInterruptIfRunning* è *true*, il thread in esecuzione può essere interrotto. Restituisce *true* se l'operazione è stata annullata con successo;
- ***boolean isCancelled()*** : verifica se l'operazione sia stata cancellata o meno;

- ***boolean isDone()*** : verifica se l'operazione sia stata terminata o meno, indipendentemente che sia stata completata normalmente o annullata.

E' possibile inoltre eseguire più task insieme ed ottenere una lista di possibili *Future* con il metodo ***invokeAll()*** che prende come ingresso una *Collection* di *Callable.*

```
// Creazione di una lista di Callable
List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(() -> "Task 1");
callableTasks.add(() -> "Task 2");
callableTasks.add(() -> "Task 3");

List<Future<String>> futures =
  executorService.invokeAll(callableTasks);
```

Con il metodo ***invokeAny()*** è possibile eseguire più *task* insieme ed attendere che termini con successo almeno una di esse per riprendere l'esecuzione del thread. Le altre *task* non terminate verranno cancellate.

N.B.
Molto del lavoro è svolto dallo *ExecutorService* che è responsabile dell'esecuzione e della coordinazione delle *task*

```
// Creazione di una lista di Callable
List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(() -> {
  Thread.sleep(2000);
  return "Task 1 completed successfully";
});
callableTasks.add(() -> {
  Thread.sleep(1000);
  throw new RuntimeException("Task 2 failed");
});
callableTasks.add(() -> {
  Thread.sleep(3000);
  return "Task 3 completed successfully";
});

String result = executorService.invokeAny(callableTasks);
```

Asynchronous computations

La **computazione asincrona** si riferisce all'esecuzione di operazioni che possono essere eseguite indipendentemente senza bloccare il flusso principale del programma (**codice non bloccante**), eseguendo un'attività su un thread separato rispetto al thread principale dell'applicazione e notificando al thread principale il suo avanzamento, completamento o errore. In questo modo, il thread principale non si blocca/attende il completamento dell'attività e può eseguire altre attività in parallelo.

In Java, ci sono diverse API e concetti che consentono di gestire la computazione asincrona, inclusi thread, *ExecutorService*, *CompletableFuture*, e *Future.*

N.B.
Il calcolo asincrono è difficile da ragionare. Di solito, vogliamo pensare a qualsiasi calcolo come una serie di passaggi, ma nel caso del calcolo asincrono, le azioni rappresentate come *callback* tendono a essere sparse nel codice o profondamente annidate l'una nell'altra. Le cose peggiorano ulteriormente quando dobbiamo gestire gli errori che potrebbero verificarsi durante uno dei passaggi.

CompletableFuture

La classe ***CompletableFuture*** definisce il contratto per una fase di calcolo asincrona che possiamo combinare con altre fasi.

Implementa l'interfaccia *Future* in modo da poterla utilizzare come implementazione *Future* ma **con una logica di completamento aggiuntiva**. Ad esempio, possiamo creare un'istanza di questa classe con un costruttore no-arg per rappresentare un risultato futuro, distribuirlo ai consumatori e completarlo in un momento futuro. I consumatori possono utilizzare il metodo ***get()*** per bloccare il thread corrente finché non viene fornito questo risultato.

```java
public Future<String> calculateAsync() throws InterruptedException {
    CompletableFuture<String> completableFuture = new CompletableFuture<>();

    Executors.newCachedThreadPool().submit(() -> {
        Thread.sleep(500);
        completableFuture.complete("Hello");
        return null;
    });

    return completableFuture;
}

Future<String> completableFuture = calculateAsync();

String result = completableFuture.get();
assertEquals("Hello", result);
```

N.B.
Si noti che il metodo *calcolaAsync* restituisce un'istanza Future. Chiamiamo semplicemente il metodo, riceviamo l' istanza Future e chiamiamo il metodo get su di essa quando siamo pronti a bloccare il risultato.

Una classe *CompletableFuture* può completarsi in due modi:

- il risultato viene elaborato
- viene lanciata un'eccezione

I principali metodi forniti dalla classe *CompletableFuture* sono:

- **static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)**: Avvia un'attività asincrona restituendo un *CompletableFuture* che sarà completato con il risultato fornito dal *Supplier*. L' interfaccia **Supplier** è un'interfaccia funzionale generica con un singolo metodo (***get()***) che non ha argomenti e restituisce un valore di tipo parametrizzato. L'interfaccia *Supplier* viene spesso utilizzata per rappresentare un fornitore di valori, ovvero qualcosa che fornisce un risultato senza richiedere alcun input. Questo ci consente di fornire un'istanza del *Supplier* come espressione lambda che esegue il calcolo e restituisce il risultato;

- **complete(T value)** o **completeExceptionally(Throwable ex)**: vengono utilizzati per completare un CompletableFuture con un risultato o un'eccezione. Entrambi i metodi sono utilizzati per segnalare che l'operazione asincrona è stata completata. Il primo metodo completa il *CompletableFuture* con il valore specificato (Se l'operazione asincrona ha avuto successo, si può chiamare questo metodo per completare il CompletableFuture con il risultato ottenuto), mentre il secondo completa il *CompletableFuture* con l'eccezione specificata (se l'operazione asincrona fallisce, si può chiamare questo metodo per completare il CompletableFuture con l'eccezione generata);
- **whenComplete()**: permette di eseguire un'azione quando un CompletableFuture è completato, indipendentemente se l'operazione ha avuto successo o ha generato un'eccezione. Questo metodo prende in input un *BiConsumer*, che accetta il risultato e l'eventuale eccezione come argomenti.

Metodi utili per la manipolazione di risultati:

- **thenApply()**: applica una funzione al risultato quando è disponibile e restituisce un nuovo *CompletableFuture* che rappresenta il risultato trasformato. Concatena anche più fasi di calcolo insieme. Viene eseguito sullo stesso thread dell'attività *supplyAsync()*. In alternativa, viene eseguito nel thread principale se l'attività *supplyAsync()* viene completata immediatamente.
  Possiamo usare questo metodo per lavorare con il risultato della chiamata precedente. Tuttavia, un punto chiave da ricordare è che il tipo restituito sarà combinato per tutte le chiamate.
- Quindi questo metodo è utile quando vogliamo trasformare il risultato di una chiamata CompletableFuture ;
- **thenAccept()**: esegue un'azione sul risultato quando è disponibile, senza restituire un valore;
- **thenCompose()**: restituisce un nuovo *CompletableFuture* ottenuto applicando la funzione fornita al risultato quando è disponibile. È simile a thenApply() in quanto entrambi restituiscono un nuovo CompletionStage. Tuttavia, thenCompose() utilizza la fase precedente come argomento . Si appiattirà e restituirà direttamente un Future con il risultato, anziché un future nidificato come abbiamo osservato in thenApply():;
- **thenCombine()**: restituisce un nuovo *CompletableFuture* ottenuto combinando il risultato dell'attuale *CompletableFuture* con un altro *CompletionStage* usando la funzione fornita;
- **handle()**: restituisce un nuovo *CompletableFuture* gestendo sia il risultato che un'eventuale eccezione dell'operazione;
- **thenRun()**: esegue un'azione quando il risultato è disponibile, senza accedere al risultato stesso.

Approfondimento utilizzo metodi (1) - Approfondimento utilizzo metodi (2)

Regole di visibilità

I core non lavorano sulla memoria locale. Quando un thread lavora su un core, può mantenere una copia locale dei dati nella sua cache, che è più veloce rispetto alla memoria principale. Quindi i thread lavorano su memorie separate. Questo può portare a situazioni in cui i diversi thread operano su copie locali separate dei dati, e ciò può causare problemi di coerenza della memoria.

La sincronizzazione tra la memoria cache (accessibile al solo thread) e la memoria locale (accessibile a tutti), e viceversa, avviene solo in alcuni casi.

In particolare, per rendere visibile un aggiornamento sulla memoria locale anche alla memoria cache del core si utilizzano:

- **variabili final**: il suo valore non può essere modificato dopo l'inizializzazione. Questo comportamento implica che la variabile è sicura per essere letta da tutti i thread senza la necessità di ulteriori meccanismi di sincronizzazione. L'inizializzazione di una variabile final è visibile a tutti i thread;
- **variabili static**: sono associate alla classe e non a un'istanza specifica. Quando una variabile statica viene inizializzata, il suo stato è reso visibile a tutti i thread e le successive modifiche saranno anch'esse visibili;
- **variabili volatile**: le letture e le scritture della variabile devono avvenire in modo atomico e che le modifiche apportate da un thread sono visibili immediatamente a tutti gli altri thread. Le variabili *volatile* impediscono l'utilizzo della cache locale del core, garantendo che le letture e le scritture avvengano direttamente sulla memoria principale;

Quando due o più thread tentano di accedere e modificare la stessa risorsa condivisa contemporaneamente, il risultato può essere imprevedibile e non deterministico. Questo può portare ad una **race condition**, a **problemi di sincronizzazione** o addirittura a **errori critici**.

Per evitare le race condition in Java, è importante utilizzare la sincronizzazione dei thread:

- **Confinamento**: limitare la visibilità delle variabili solo a un thread specifico. Riducendo la quantità di dati condivisi, si riducono anche le opportunità per le race condition. Questo può essere ottenuto dichiarando variabili come locali al thread o utilizzando strutture dati thread-safe quando possibile;
- **Immutabilità**: gli oggetti immutabili, una volta creati, non possono essere modificati. Poiché non possono essere modificati, possono essere condivisi tra thread senza il rischio di race condition;
- **Sezione critica/Blocco**: Una sezione critica è una parte di codice in cui le operazioni su dati condivisi vengono eseguite. L'utilizzo di blocchi **synchronized** o oggetti **Lock** consente di garantire che solo un thread alla volta possa eseguire le operazioni all'interno della sezione critica. Questo previene le race condition assicurando l'accesso esclusivo alla risorsa condivisa.

Inoltre, l'utilizzo di blocchi **synchronized** o **oggetti Lock** è fondamentale per garantire la mutua esclusione in una sezione critica. Questi meccanismi consentono a un solo thread alla volta di acquisire un *lock* per una risorsa condivisa, evitando così race condition.

Synchronized e Lock

I blocchi **synchronized** sono un meccanismo di sincronizzazione che viene utilizzato per garantire la mutua esclusione tra thread quando accedono a risorse condivise o eseguono operazioni che devono essere atomiche. Un blocco synchronized viene definito attorno a una sezione critica del codice, assicurando che solo un thread alla volta possa eseguire tale sezione critica.

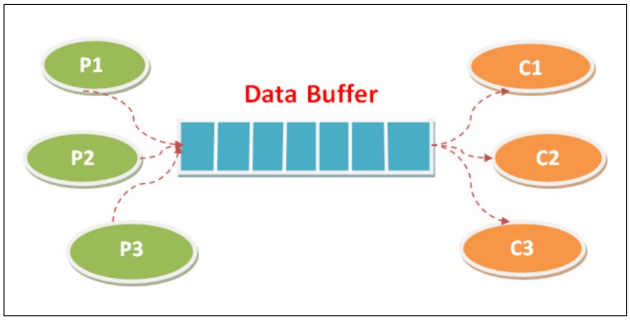
In questo esempio, la classe *SharedResource* ha un campo *counter* che rappresenta una risorsa condivisa. I metodi *increment* e *getCounter* sono sincronizzati utilizzando lo stesso oggetto lock. Questo significa che solo un thread alla volta può eseguire ciascun metodo, garantendo che le operazioni all'interno della sezione critica siano eseguite in modo atomico.

```java
public class SharedResource {
    private int counter = 0;
    // Oggetto utilizzato come lock
    private final Object lock = new Object();
    public void increment() {
        synchronized (lock) {
            // Sezione critica
            counter++;
        }
    }

    public int getCounter() {
        synchronized (lock) {
            // Sezione critica
            return counter;
        }
    }
}
```

Alcuni punti chiave sui blocchi synchronized:

- **Mutua Esclusione**: un blocco synchronized garantisce che solo un thread alla volta possa eseguire la sezione critica;
- **Rischi di Deadlock**: è importante evitare la possibilità di deadlock, in cui due o più thread attendono indefinitamente il rilascio di un lock;
- **Performance**: l'utilizzo eccessivo di blocchi synchronized può portare a problemi di prestazioni in quanto i thread potrebbero dover attendere l'accesso alla sezione critica;
- **Lock del Monitor**: ogni oggetto Java ha un monitor associato e può essere utilizzato come lock. I blocchi synchronized utilizzano il lock del monitor associato all'oggetto specificato;
- **Volatile e Synchronized**: mentre volatile offre garanzie sulla visibilità, i blocchi synchronized forniscono sia garanzie sulla visibilità che sulla mutua esclusione.

Producer/Consumer pattern



Il pattern **Producer-Consumer** è un modello di progettazione multithreading che coinvolge due tipi di thread: i produttori (*Producers*) e i consumatori (*Consumers*). L'obiettivo principale è consentire una comunicazione sicura e sincronizzata tra i thread, evitando race condition e garantendo che i consumatori ottengano dati solo quando sono disponibili.

In questo modello, uno o più produttori generano dati e li mettono in un buffer o una coda condivisa, mentre uno o più consumatori estraggono i dati dalla coda e li elaborano.

Il pattern produttore-consumatore è spesso utilizzato in situazioni in cui i produttori e i consumatori devono operare a velocità diverse o su sistemi distribuiti.

Il modello è composto dai seguenti elementi:

- **buffer** o una coda condivisa: funge da area di scambio tra produttori e consumatori. Questo buffer deve essere progettato in modo da essere thread-safe, garantendo l'accesso sicuro ai dati;
- **uno o più produttori**: sono responsabili di generare o produrre dati. Questi dati vengono poi inviati o inseriti in un buffer o una coda condivisa;
- **uno o più consumatori**: sono responsabili di prelevare o consumare i dati prodotti dai produttori. Consumano i dati dalla stessa coda o buffer condiviso.

Il pattern produttore-consumatore è implementato utilizzando la sincronizzazione, al fine di garantire la corretta comunicazione e coordinazione tra i produttori e i consumatori. In particolare, la sincronizzazione è utilizzata per garantire che il buffer o la coda condivisa sia accessibile in modo sicuro e che i produttori e i consumatori si sincronizzino correttamente quando accedono alla coda (che i produttori non sovrascrivano dati non consumati e che i consumatori non tentino di consumare dati non ancora disponibili).

Monitors

In programmazione concorrente, un **monitor** è un meccanismo di sincronizzazione utilizzato per gestire l'accesso concorrente a una risorsa condivisa da parte di più thread.

Un monitor è un'astrazione di alto livello che fornisce un'interfaccia per gestire l'accesso concorrente a una risorsa condivisa.

In pratica, il monitor fornisce metodi per bloccare l'accesso alla risorsa, in modo che solo un thread alla volta possa accedere alla risorsa. Inoltre, il monitor può fornire metodi per gestire l'attesa dei thread che cercano di accedere alla risorsa quando questa è già in uso.

In Java, un monitor è implementato utilizzando la parola chiave *synchronized*. Quando un metodo è marcato come *synchronized*, il thread che lo esegue acquisisce il blocco associato al monitor e blocca l'accesso ad altri thread fino a quando il blocco non viene rilasciato.

Qualsiasi oggetto in Java può svolgere il ruolo di monitor. Solo gli oggetti che implementano l'interfaccia *java.lang.Object* forniscono i metodi ***wait()***, ***notify()***, e ***notifyAll()***, che consentono di implementare la sincronizzazione tra i thread.

Il metodo ***wait()*** consente a un thread di sospendere la sua esecuzione e rilasciare il blocco sincronizzato su un oggetto monitor. Il thread rimane in uno stato di attesa finché non viene notificato da un altro thread attraverso il metodo *notify()* o *notifyAll()*, oppure finché non scade il tempo di attesa specificato [***wait(timer)***].

Il metodo ***notify()*** viene utilizzato per notificare un thread in attesa di un oggetto monitor che qualcosa è cambiato e che il thread può riprendere l'esecuzione.

Il metodo ***notifyAll()*** notifica tutti i thread in attesa dell'oggetto monitor.

Questi metodi vengono utilizzati comunemente per implementare il pattern "Producer-Consumer", in cui uno o più thread produttori generano dati che vengono poi elaborati da uno o più thread consumatori. I produttori scrivono i dati in una coda condivisa e i consumatori leggono i dati dalla coda. Quando la coda è vuota, i consumatori si mettono in attesa utilizzando il metodo *wait()*, mentre i produttori notificano i consumatori quando aggiungono dati alla coda utilizzando il metodo *notify()* o *notifyAll()*.

La differenza principale tra i due metodi è la modalità di notifica dei thread:

- ***notify()***: notifica un solo thread in attesa sull'oggetto monitor. Il thread notificato viene scelto casualmente dal sistema operativo. Tutti gli altri thread in attesa rimangono in attesa finché non vengono notificati nuovamente o finché non scade il timeout impostato;
- ***notifyAll()***: notifica tutti i thread in attesa sull'oggetto monitor. Tutti i thread notificati tentano di acquisire il lock dell'oggetto monitor e riprendere l'esecuzione.

In generale, è sempre consigliabile utilizzare *notifyAll()* invece di *notify()*, a meno che non si **sappia esattamente quale thread debba essere notificato** e si abbia una buona ragione per farlo. Utilizzando *notifyAll()*, si garantisce che tutti i thread in attesa sull'oggetto monitor verranno notificati e si evitano problemi di deadlock o di starvation.

In sintesi, utilizzare *notifyAll()* è sempre una scelta sicura, mentre l'utilizzo di *notify()* richiede una maggiore attenzione per evitare problemi di concorrenza.

High Level Concurrency Objects

Quelli elencati fin'ora sono oggetti di programmazione che consentono ai programmatori di scrivere codice concorrente in modo più semplice e sicuro. Sono rappresentati nelle classi del package *java.util.concurrent*.

Queste classi forniscono un set di oggetti di alto livello:

- **Lock**: rappresenta un oggetto che permette di acquisire e rilasciare in modo esclusivo un blocco di codice. A differenza dei blocchi synchronized, i lock possono essere interrotti e supportano la concorrenza tra lettori/scrittori. Gli oggetti lock sono utilizzati per evitare situazioni di race condition, in cui due o più thread cercano di accedere contemporaneamente ad una risorsa condivisa. Quando un thread acquisisce un lock, gli altri thread devono aspettare che il lock venga rilasciato prima di poter accedere alla risorsa condivisa. Ciò garantisce che solo un thread alla volta possa modificare la risorsa condivisa, evitando così problemi di race condition. I metodi che mette a disposizione sono:
    - **void lock()**: acquisisce il blocco. Se il lock è già stato acquisito da un altro thread, il thread corrente viene sospeso fino a quando il lock non è disponibile;
    - **void unlock()**: rilascia il lock precedentemente acquisito.
    - **void lockInterruptibly()**: permette al thread di essere interrotto mentre attende il lock. Se il lock è già stato acquisito da un altro thread, il metodo *lockInterruptibly()* sospende il thread corrente fino a quando il lock non viene rilasciato da un altro thread o fino a quando il thread corrente non viene interrotto. Se il thread viene interrotto mentre attende il lock, viene generata un'eccezione *InterruptedException;*
    - **boolean tryLock()**: tenta di acquisire il lock. Restituisce *true* se il lock è stato acquisito con successo, altrimenti restituisce *false*. Questo metodo non blocca il thread in attesa del lock, ma ritorna immediatamente;
    - **boolean tryLock(long time, TimeUnit unit)**: acquisisce il lock se è libero entro il tempo di attesa dato;

- **Condition newCondition()**: restituisce un oggetto *Condition* associato al lock corrente. L'oggetto **Condition** può essere utilizzato per gestire l'accesso concorrente a una risorsa specifica, consentendo ai thread di attendere la disponibilità di tale risorsa. Un oggetto *Condition* rappresenta una situazione in cui i thread devono aspettare per un evento specifico per poter procedere. Ad esempio, supponiamo che ci sia una risorsa condivisa tra due thread e che il primo thread debba attendere finché la risorsa non è disponibile per poterla utilizzare. In questo caso, il primo thread può utilizzare una *Condition* per sospendersi e attendere fino a quando la risorsa non è disponibile, mentre il secondo thread può segnalare la condizione quando la risorsa diventa disponibile.

```
private final Lock lock = new ReentrantLock();

public void performOperation() {
    lock.lock(); // Acquisisce il blocco
    try {
        // Sezione critica: operazioni sicure da eseguire sotto il blocco
        System.out.println("Operazione sicura sotto il blocco");
    } finally {
        lock.unlock(); // Rilascia il blocco anche in caso di eccezione
    }
}

public static void main(String[] args) {
    LockExample example = new LockExample();

    // Esempio di utilizzo del blocco in un contesto multithreading
    Thread thread1 = new Thread(() -> example.performOperation());
    Thread thread2 = new Thread(() -> example.performOperation());

    thread1.start();
    thread2.start();
}
```

I metodi utilizzabili sono:

- **await()**: mette il thread corrente in attesa fino a quando un altro thread non segnala che la condizione associata è cambiata. Durante l'attesa, il blocco del thread viene rilasciato;
- **await(long time, TimeUnit unit)**: attende il cambiamento della condizione per il periodo di tempo specificato. Se la condizione cambia prima che scada il timeout, il thread si risveglia e procede. Se il timeout scade senza che la condizione sia soddisfatta, il thread ritorna da **await()**;
- **awaitUninterruptibly()**: sospende il thread corrente e attende che la *Condition* venga soddisfatta, ignorando eventuali interruzioni;
- **awaitUninterruptibly(long time, TimeUnit unit)**: sospende il thread corrente e attende che la condition venga soddisfatta per al massimo il tempo specificato, ignorando eventuali interruzioni;
- **awaitUntil(Date deadline)**: sospende il thread corrente e attende che la condition venga soddisfatta fino alla data specificata;

    - **awaitNanos(long nanosTimeout)**: sospende il thread corrente e attende che la *Condition* venga soddisfatta per al massimo il numero di nanosecondi specificato;
    - **signal()**: risveglia uno dei thread in attesa su questa condizione. Il thread risvegliato avrà la possibilità di acquisire il blocco prima di procedere;**signalAll()**: risveglia tutti i thread in attesa su questa condizione. Questa opzione è utile quando più di un thread può procedere dopo che la condizione è stata modificata.
- **Executor**: rappresenta un oggetto che gestisce l'esecuzione di task in modo asincrono;
- **Concurrent Collections**: permettono l'accesso concorrente a una collezione, garantendo la consistenza dei dati e la sincronizzazione tra i thread.
- **Atomic Variables**: sono degli oggetti che consentono di eseguire operazioni su variabili condivise in modo atomico, cioè in modo indivisibile e non interferibile da altri thread. Ciò significa che una volta che un thread ha iniziato un'operazione su una variabile atomica, gli altri thread non possono intervenire finché l'operazione non è stata completata.
  Le **Atomic Variables** sono utilizzate principalmente in situazioni in cui è necessario garantire la consistenza dei dati in presenza di thread concorrenti. Ad esempio, in un'applicazione server che gestisce richieste concorrenti, potrebbe essere necessario utilizzare una *AtomicLong* per contare il numero di richieste gestite, senza incorrere in problemi di race condition.
  L'utilizzo delle *Atomic Variables* è più efficiente rispetto ai meccanismi di sincronizzazione come i lock, poiché non richiede l'acquisizione e il rilascio di lock, ma richiede comunque una buona conoscenza della programmazione concorrente e delle best practice per evitare situazioni di deadlock o di race condition.

UML (Unified Modelling Language)

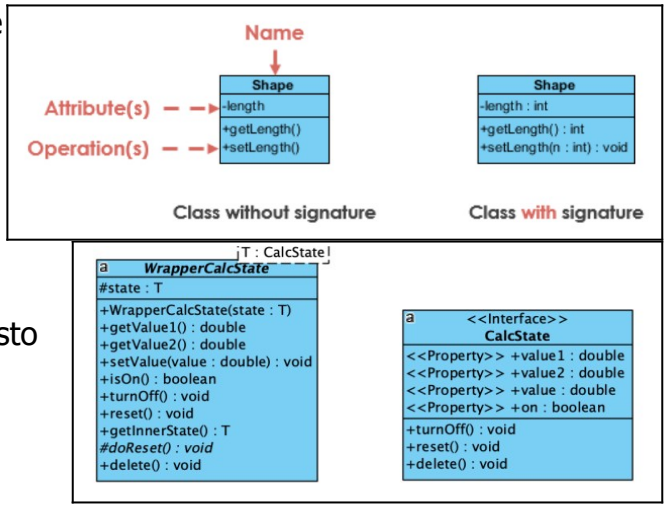
È un linguaggio standardizzato per la modellazione e la documentazione dei sistemi software. È utilizzato per visualizzare, specificare, costruire e documentare i componenti di un sistema software. UML è ampiamente utilizzato nell'ingegneria del software e fornisce un insieme di notazioni grafiche per rappresentare visivamente diversi aspetti di un sistema.

Un **class diagram** (diagramma delle classi) è uno dei tipi di diagrammi più comuni. Rappresenta la struttura statica di un sistema, evidenziando le classi che compongono il sistema e le relazioni tra di esse. I class diagram sono utilizzati per modellare la struttura a oggetti di un sistema software, mostrando le classi, gli attributi delle classi e le relazioni tra le classi. Gli elementi chiave di un class diagram sono:

- **Classe**: rappresenta un'astrazione di un concetto o di un oggetto nel sistema. Una classe è rappresentata graficamente da un rettangolo diviso in tre sezioni: il nome della classe nella parte superiore, gli attributi nella sezione intermedia e i metodi nella parte inferiore;
- **Attributo**: rappresenta una caratteristica o una proprietà della classe. Gli attributi sono elencati nella sezione centrale della classe e sono generalmente composti da un nome e un tipo di dato;
- **Metodo**: rappresenta un comportamento o un'azione associata alla classe. I metodi sono elencati nella parte inferiore della classe e includono il nome del metodo, i parametri e il tipo di ritorno;
- **Relazioni**: indicano le associazioni e le dipendenze tra le classi. Ci sono diversi tipi di relazioni, come l'associazione, l'aggregazione, la composizione e l'ereditarietà, che possono essere rappresentate attraverso linee e frecce che collegano le classi.

UML Class Notation

Un singolo diagramma è utilizzato per fornire tutte le informazioni di una classe. E' possibile inserire la *signature*, ovvero le informazioni sui tipi di parametri e tipi di ritorno. Essendo una notazione **informale** questa struttura può essere adattata alle nostre esigenze. Inoltre possiamo anche omettere alcuni metodi e attributi. Tutto questo perché UML è utilizzato solo come accompagnamento alla progettazione del software.



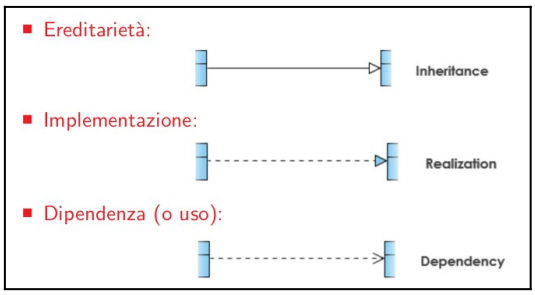
I simboli **+**, **-**, e **#** indicano la visibilità di quel metodo o attributo:

- **+** -> visibilità pubblica
- **-** -> visibilità privata
- **#** -> visibilità protected
- l'assenza del simbolo indica una visibilità di default

Le classi e le interfacce astratte sono rappresentati mediante simboli aggiuntivi (a) posizionati intorno al nome.

Relazioni

In un diagramma UML sono anche rappresentate delle relazioni tra le classe.



- Abbiamo una **relazione di ereditarietà** quando una classe ne estende un'altra.
- Abbiamo una **relazione di implementazione** quando una classe implementa una interfaccia.
- Abbiamo una **relazione di dipendenza** quando una classe utilizza nel proprio stato un oggetto che è istanza di un'altra classe.

Pattern architetturali

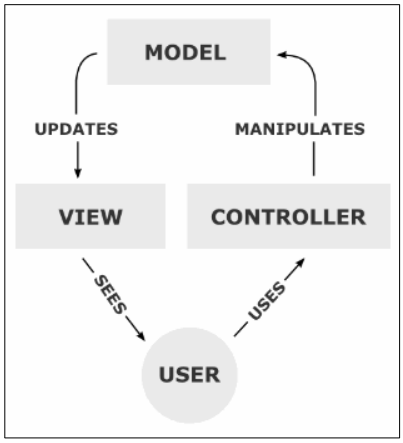
Un pattern architetturale è un modello generale che fornisce una soluzione organizzativa e strutturale a problemi ricorrenti nell'architettura del software. Questi pattern forniscono un approccio riconosciuto e consolidato per progettare, organizzare e strutturare il codice in modo da affrontare sfide specifiche legate all'architettura di un'applicazione.

I pattern architetturali forniscono linee guida e best practice che possono essere applicate a livello di alto livello per la progettazione del sistema, facilitando la comprensione, la manutenzione e l'evoluzione del software nel tempo. Molti di questi pattern sono applicabili a diverse tipologie di progetti e linguaggi di programmazione.

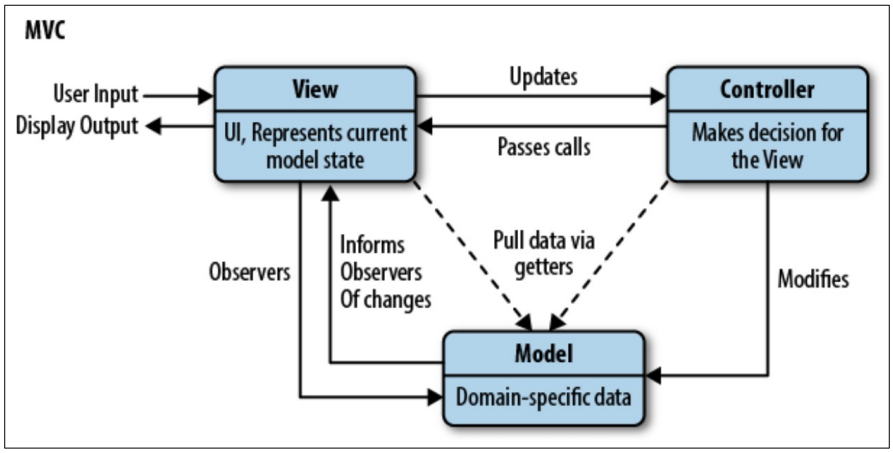
Model View Controller

È un pattern architetturale molto diffuso nello sviluppo di software che divide un'applicazione in tre componenti aventi ognuno un compito in base al ruolo che ricopre. Questo pattern promuove la separazione delle responsabilità, facilitando la gestione, la manutenzione e l'evoluzione del codice.

- **Model**: fornisce i metodi per accedere ai dati utili all'applicazione. È un componente principale dell'MVC, gestisce direttamente i dati, la logica, e le regole dell'applicazione.
- **View**: visualizza i dati contenuti nel modello e si occupa dell'interazione con gli utenti. Una vista può essere una qualsiasi rappresentazione in output di informazioni, come un grafico o un diagramma. Sono possibili viste multiple delle stesse informazioni.



- **Controller**: riceve i comandi dall'utente e li attua modificando lo stato degli altri due componenti. Accetta l'input e lo converte in comandi per il modello e/o vista.



Il flusso di interazione nell'architettura MVC è generalmente il seguente:

1. L'utente interagisce con la View.
2. Il Controller cattura l'input dell'utente.
3. Il Controller interagisce con il Model per ottenere o aggiornare i dati.
4. Il Model notifica la View dei cambiamenti nei dati.
5. La View ottiene i dati dal Model e si aggiorna di conseguenza.
6. L'interfaccia utente è ora aggiornata e pronta per ulteriori interazioni.

Pro

- **Separazione delle responsabilità**: permette un riutilizzo efficiente del codice e lo sviluppo parallelo da parte di più programmatori. La modifica in un componente non richiede necessariamente modifiche nei componenti correlati. Consente il raggruppamento logico di azioni correlate su un controller insieme.
- **Riutilizzo del codice**: i componenti possono essere riutilizzati in diverse parti dell'applicazione o in progetti diversi. I dettagli dell'interazione fra i componenti di questa struttura dipendono molto dalle tecnologie usate e dal tipo di applicazione.

- **Manutenibilità**: facilita la manutenzione del codice, in quanto la localizzazione degli errori o la modifica delle funzionalità è più agevole grazie alla separazione delle responsabilità.
- **Testabilità**: i singoli componenti possono essere testati separatamente, facilitando il testing unitario e migliorando la manutenibilità del codice.

Contro

- Navigabilità del codice complessa perché introduce nuovi livelli di astrazione e richiede agli utenti di adattarsi ai criteri di scomposizione di MVC;
- La scomposizione provoca la dispersione. Quindi la comprensione dei vari aspetti e componenti è talvolta complessa;
- Gli sviluppatori che utilizzano MVC devono essere esperti su più tecnologie.

Observer

E' un design pattern utilizzato come base architetturale di molti sistemi di gestione di eventi. E' utilizzato soprattutto in quei software che richiedono la gestione di eventi provenienti da diversi oggetti. Definisce una dipendenza uno a molti tra oggetti in modo che quando un oggetto cambia stato, tutti i suoi dipendenti siano notificati e aggiornati automaticamente

Il pattern si basa su uno o più oggetti chiamati osservatori (o ***observer***), che vengono registrati per **gestire un evento** che potrebbe essere generato dall'oggetto osservato.

Uno degli aspetti fondamentali è che tutto il funzionamento dell'*observer* si basa su meccanismi di ***callback***, implementabili in diversi modi e spesso a questa funzione vengono passati dei parametri in fase di generazione dell'evento.

Dunque un *Observer* è in grado di:

- gestire una **dipendenza uno-a-molti** tra gli oggetti senza rendere gli oggetti ben accoppiati;
- **notificare il cambiamento di stato** di un numero illimitato di oggetti;
- **definire oggetti *Observable* ed *Observer***;

Un oggetto *Observable* è un oggetto osservabile nel paradigma *MVC*. Dopo che un'istanza osservabile cambia, un metodo di *Observable* fa sì che tutti i suoi osservatori ricevano una notifica della modifica tramite una chiamata al loro metodo di aggiornamento.

N.B.
Un oggetto osservabile può avere uno o più osservatori. Un osservatore può essere qualsiasi oggetto che implementa l'interfaccia *Observer*.

Gradle Build Tool

E' un sistema avanzato per la gestione del processo di compilazione e gestione delle dipendenze nei progetti software. Funziona per tutti i linguaggi di programmazione basato su **Groovy** e **Kotlin**, due linguaggi di programmazione costruiti sopra la Virtual Machine di Java, che permettono di scrivere del codice utilizzando una sintassi più moderna.

Esso sopporta il download automatico e la configurazione di dipendenze di altre librerie. Semplicemente basterà "dire" a *Gradle* che si ha bisogno di una libreria e lui la scaricherà, la installerà nella repository dell'applicazione e sarà possibile utilizzarla per costruire del codice. Supporta anche il building di progetti multipli.

Gradle è basato su nozioni di:

- **progetto**: rappresenta l'unità di *build*. Può corrispondere a un'applicazione, a una libreria o a un modulo del progetto. Contiene informazioni come le dipendenze, le configurazioni di build, i task e altro ancora. Un progetto può anche avere dipendenze da altri progetti, il che è particolarmente utile in scenari di progetti multipli;
- **task**: rappresenta un'unità di lavoro atomica all'interno di un progetto. Può essere qualsiasi cosa, da una compilazione di codice a un'operazione di test o all'esecuzione di uno script personalizzato. I task possono essere eseguiti singolarmente o possono dipendere l'uno dall'altro, formando una catena di dipendenze.

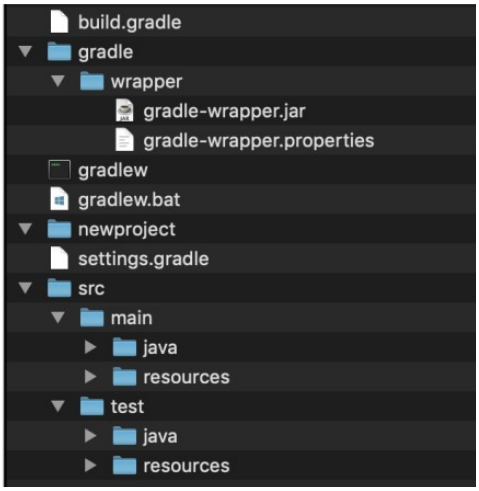
Quando si definisce uno script di build in Gradle, si dichiarano progetti e task, specificando le dipendenze tra di essi. Gradle quindi si occupa di eseguire i task in modo efficiente, tenendo traccia delle dipendenze e eseguendo solo i task che sono necessari in base alle modifiche apportate al codice sorgente o alle risorse.

Iniziare un progetto

Anzitutto si crea una directory dove il progetto dovrà essere salvato utilizzando il comando ***$ gradle init*** che genererà lo scheletro della nostra applicazione.

Verranno creati i seguenti file e directory:

- **src** (directory)
  - **main** (directory): che conterrà tutti i sorgenti della nostra applicazione;
  - **test** (directory): che conterrà i test per la nostra applicazione;
- **build.gradle** (file): contenente le informazioni riguardanti il progetto corrente;



- **gradle.wrapper** (directory)
  - **gradle-wrapper.jar** (file): contenente un file JAR eseguibile che non è altro che un'applicazione Java che è l'interprete di *Gradle*;
  - **gradle-wrapper.properties** (file): che conterrà la configurazione del *Gradle Wrapper*;
- **gradlew** (file): uno script per i sistemi Unix-based;
- **gradlew.bat** (file): uno script per i sistemi Windows;
- **settings.gradle** (file): un file contenente la configurazione di *Gradle* per il processo di build.

Build

Per fare la **build** del progetto basterà eseguire il comando gradle o utilizzando lo script wrapper messo a disposizione nella cartella (opzione migliore).

La prima volta che viene eseguita la build, Gradle verificherà di avere tutte le librerie necessarie a disposizione e, se necessario, provvederà a scaricarle nella directory ***.gradle***. Il *task build* compilerà tutte le classi Java, eseguirà i test generando gli opportuni report accessibili dalla directory ***build/reports/tests/test/index.html***.

La lista dei *task* disponibili è ottenibile utilizzando il comando **./gradlew tasks**

Run

Infine per **generare** una applicazione Java possiamo utilizzare il comando ***./gradlew run*** che eseguirà l'applicazione richiamando il metodo *main* associato.

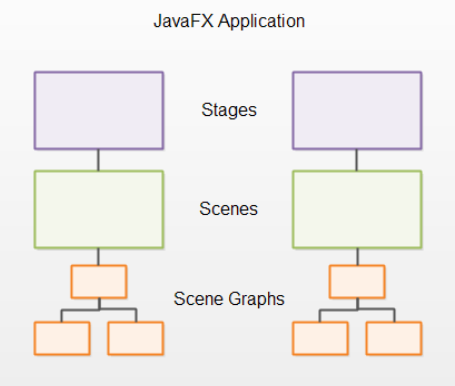
Distribuzione dell'applicazione

Può anche essere usato per generare i file *zip* e *tar* contenenti tutto ciò che è necessario per eseguire la nostra applicazione attraverso un plugin chiamato **Distribution**.

JavaFX

E' un insieme di strumenti per lo sviluppo di **GUI** (***Graphical User Interface***) per applicazioni desktop, mobile e giochi. Presenta già diversi componenti pre-costruiti ed altri nuovi possono essere aggiunti alla libreria.

E' possibile definire l'interfaccia grafica non solo tramite codice Java, ma utilizzando file FXML o fogli di stile (CSS).



Struttura

Un'applicazione JavaFX (che è in sé un oggetto) contiene:

- **Stage**: possono essere uno o più e corrispondono alla finestra dell'applicazione che comprende tutti gli oggetti. Ogni finestra aperta sul desktop ha un proprio *Stage*. Ogni *Stage* è rappresentata da un oggetto messo a disposizione da JavaFX;
- **Scene**: sono contenute negli stage e corrispondono ai contenuti fisici di un'applicazione. Ogni *Stage* può rappresentare una sola scena alla volta, che possono cambiare molteplici volte. Una *scena* è rappresentata da un oggetto *Scene* fornito da JavaFX;
- **Scene graphs e nodi**: sono contenuti a loro volta nelle scene e rappresentano gli oggetti delle scene in una struttura gerarchica. Ogni oggetto all'interno della scena deve essere collegato ad una scena, e quella *scena* deve essere collegata ad uno *stage* per essere visibile. Il grafico dei layout e dei controlli è chiamato ***grafico della scena*** (*scene graph,* non visibile nella scena). Tutti i componenti che possono essere inseriti nello *scene graph* sono detti ***nodi***. Si possono ottenere due tipi di nodi:
  - **Branch nodes**: possono contenere al loro interno altri nodi (*child nodes*);
  - **Leaf nodes**: non possono contenere altri nodi al loro interno;
- **JavaFX controls**: sono quegli elementi grafici ai quali possiamo fornire un comportamento (bottoni, tabelle, ecc…). Per essere visibili devono essere "attaccati" allo *scene graph*;
- **JavaFX layouts**: sono componenti critici di ogni applicativo grafico. Essi contengono gli altri componenti al loro interno e controllano le dimensioni. È associato alla scena ed è considerato come il componente *Parent* in quanto contiene gli altri oggetti al suo interno sono sottoclassi di JavaFX (*javafx.scene.Parent*). Un layout può contenere anche altri layout (un po' come i *div* in linguaggio HTML).

Lo spazio grafico di lavoro di JavaFX è un oggetto *Stage*, radice di ogni applicazione JavaFX, che può intercambiare oggetti *Scene*, dei contenitori (oggetti che estendono la classe Container) di generici componenti grafici (*Node*). A tutti gli effetti un oggetto Scene è un albero di nodi, in cui ogni nodo può essere sia un contenitore (che ad esempio specifica la disposizione bidimensionale dei suoi sotto componenti), sia un nodo grafico, cioè un *Lightweight Component* con una opportuna rappresentazione ed una serie di attributi che ne descrivono le proprietà (posizione, dimensioni, colorazione ecc.): il concetto ricorda molto quello di DOM di una pagina HTML, in cui possono essere disposti opportuni tag con alcuni attributi.

JavaFX Application

Per essere eseguita, un'applicazione con JavaFX necessita di una *primary launch class* che estenda la classe *javafx.application.Application*. Tutte le sottoclassi di *Application* devono contenere il metodo ***start()*** che viene invocato quando si avvia l'applicazione. Il metodo *start()* conterrà come argomento lo *stage* che voglio invocare per primo.

```
@Override
public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("My First JavaFX App");

    primaryStage.show();
}
```

```
public static void main(String[] args) {
    Application.launch(args);
}
```

Il metodo ***main(String[] args)*** conterrà solo il richiamo al metodo ***launch()*** di *Application* che passerà gli argomenti necessari all'avvio dell'applicazione.

JavaFX Event Handling

Un evento è un operazione associata ad un input dell'utente e rappresenta un'attività a cui l'applicazione deve reagire. Possono essere caratterizzati secondo due diversi approcci:

- **Foreground Events**: quegli eventi che sono dovuti ad una diretta interazione da parte dell'utente (click di un bottone, movimento del mouse, ecc..). Per la gestione di tali eventi JavaFX mette a disposizione la classe *Event*. Un evento ha tre diversi elementi:
  - **Taget**: i nodi dove l'evento è generato;
  - **Source**: la sorgente dell'evento;
  - **Type**: il tipo dell'evento;
- **Background Events**: quegli eventi che non richiedono un'interazione da parte dell'utente (errori hardware o software, timer scaduti, operazioni completate, ecc…);

La gestione di un evento segue diverse fasi:

- **Route construction**: quando l'evento viene generato, la route che viene scelta segue una catena di invio degli eventi (*Event Dispatch Chain*), parte dallo stage primario che contiene il bottone fino al nodo sorgente dell'evento;
- **Event capturing phase**: creata la catena ogni elemento invia, a partire dalla *root*, l'evento al suo sotto-componente corrispondente fino a ritornare al nodo finale. Ad ogni passaggio un nodo può *gestire l'evento* o *filtrarlo*. In questo modo posso personalizzare in modo completo la gestione dei vari eventi e far si che ogni componente possa gestirlo (purché il codice sia scritto in modo opportuno);
- **Event handlers and filters**: i filtri e i gestori contengono la logica di gestione degli eventi che decidono se gestire l'evento o passarlo al nodo sottostante;
- **Event bubbling phase**: l'evento passa dal nodo sorgente al nodo radice e quando incontra un handler che sia in grado di gestire quell'evento, esso viene eseguito. Se non vengono trovati gestori opportuni l'evento raggiunge il nodo radice e il processo si conclude.

JavaFXML

JavaFXML è una tecnologia che combina JavaFX con FXML (FXML è un linguaggio dichiarativo basato su XML) per la creazione di interfacce utente in applicazioni JavaFX. La combinazione di JavaFX e XML semplifica la progettazione e la gestione delle interfacce utente attraverso la separazione della struttura dell'interfaccia utente dal codice sorgente Java che gestisce il comportamento dell'interfaccia. In particolare è possibile caricare i tag che si vuole ed ogni applicazione può definire l'insieme dei tag inseribili. Successivamente basterà caricare il file FXML e lo eseguo nel codice specifico.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox>
    <children>
        <Label text="Hello world FXML"/>
    </children>
</VBox>
```

```
FXMLLoader loader = new FXMLLoader();
loader.setLocation(anUrl);
VBox vbox = loader.<VBox>load();

Scene scene = new Scene(vbox);
primaryStage.setScene(scene);
primaryStage.show();
```

Questo approccio favorisce la separazione delle preoccupazioni, dove la progettazione dell'interfaccia è separata dalla logica di gestione degli eventi.

Testing

Un test è un pezzo di codice che si occupa di eseguire altro codice per verificare il corretto funzionamento. Questo lo fa verificando:

- se lo stato del codice dopo l'esecuzione è quello atteso -> **state testing**
- se gli eventi o le istruzioni eseguite sono quelle attese -> **behavior testing**

I test sono utili allo sviluppatore per verificare che la logica di ogni pezzo di codice sia corretta.

Per poter eseguire i test del codice in modo efficace è necessaria una **esecuzione automatica dei test** così da poterli ripetere in modo continuativo e verificare continuamente che nessun test venga violato.

Aumentare la porzione di codice verificate da un test (***test coverage***) permette di tenere sotto controllo lo sviluppo delle funzionalità della nostra applicazione senza la necessità di svolgere test manuali.

Livelli di test

1. **Unit Testing**: individua l'insieme dei test che verificano le funzionalità specifiche di una porzione di codice, tipicamente a livello di classe o metodo. Si concentra su singole unità di codice (solitamente singoli metodi o classi);
2. **Integration Testing**: mira a verificare la corretta integrazione delle diverse componenti di una applicazione o sistema software. Coinvolge l'integrazione di più unità di codice o componenti;
3. **System Testing**: l'insieme di test che verificano il sistema nella sua interezza e che mira a controllare la corretta implementazione delle funzionalità richieste. Verifica che l'intero sistema soddisfi i requisiti funzionali e non funzionali.

Unit Testing

Uno **unit test** è un pezzo di codice scritto dallo sviluppatore al fine di eseguire specifiche funzionalità nel codice sotto test e che controlla stato o comportamento del codice. Un'unità di codice può essere la più piccola parte di un'applicazione che può essere testata separatamente. Questo può essere un metodo, una classe o anche una singola funzionalità all'interno di un metodo. Le percentuali di codice che viene testato dagli *unit test* è detta **test coverage**.

Esso si focalizza tipicamente su un metodo o una classe. Le dipendenze esterne dovrebbero essere rimosse nel test rimpiazzandole con opportune implementazioni o *mock* (dimostratori) creati mediante gli ambienti di test. Si controlla che la classe e i suoi metodi si comportino nel modo corretto.

I test vengono scritti ed eseguiti **in fase di sviluppo** e non sono presenti nel software finale. Ogni nuova funzionalità introdotta dovrebbe essere accompagnata da un nuovo insieme di test. Ogni test deve essere indipendente dagli altri e rappresenta un *contratto* sul comportamento di un singolo oggetto o funzione. Spesso sono scritti per verificare il comportamento di un oggetto in condizioni estreme.

JUnit

E' un framework per Unit Testing in Java, open source ospitato su GitHub, che ci permette di ripetere i test in modo automatico. Ogni test prende il nome di ***test case***; insiemi di *test case* compongono una ***test suite***.

Un test **JUnit** è un singolo metodo che viene inserito in una classe, detta **classe di test**, che ha il colo scopo di essere usata per il testing. Quando eseguo il test quindi, JUnit andrà ad eseguire tutti i metodi contenuti nella classe di test.

Per verificare che il risultato di un test sia effettivamente quello che mi aspettavo, vengono utilizzati i metodi ***assert methods***. Ad ogni *assert* viene associato un nome allo scopo di identificare l'eventuale errore nel report e semplificare le correzioni.

Normalmente il nome della classe di test viene creato aggiungendo "*Test*" alla fine del nome della classe da testare. Per nominare i metodi invece, è necessario nominarli a seconda di cosa esegue il metodo. In questo modo si può capire il ruolo del test senza dover leggere il codice (**nome esplicativo**).

JUnit 5 (ultima versione) consiste di diversi componenti:

- **JUnit Platform**: è il layer base che permette l'esecuzione e l'integrazione di differenti ambienti di test nella JVM;
- **JUnit Jupiter**: è l'ambiente di test fornito da JUnit 5 che è eseguito dalla JUnit Platform;
- **JUnit Vintage**: che permette l'esecuzione di vecchi test.

Annotazioni

- ***@Test*** ---> identifica un metodo di un test (è possibile creare anche altri metodi di supporto all'interno della classe di test che non necessariamente siano dei test);
- ***@RepeatedTest(n)*** ---> indica che il test deve essere ripetuto *n* volte;
- ***@BeforeEach*** ---> indica che il metodo deve essere eseguito *prima* di ogni metodo di test (utile per inizializzare le risorse);
- ***@AfterEach*** ---> indica che il metodo deve essere eseguito *dopo* di ogni metodo di test (utile per deallocare le risorse);
- ***@BeforeAll*** ---> identifica un metodo che viene eseguito una volta all'inizio del test (il metodo deve essere *static*);
- ***@AfterAll*** ---> identifica un metodo che viene eseguito alla fine di tutti i test della classe (il metodo deve essere *static*);
- ***@Tag("<TagName>")*** ---> viene utilizzato per associare ad un test un particolare *tag*, utile per filtrare i test;
- ***@Disabled*** ---> disabilita un particolare test; utile per rimuovere momentaneamente un test;
- ***@DisplayName("<Name>")*** ---> indica il nome (<*Name*>) che verrà mostrato dall'esecutore dei test. Il nome può contenere spazi.

Test Suites

Per eseguire più test contemporaneamente, questi possono essere raggruppati in una *suite*:

- ***@SelectPackages***: specifica il nome del pacchetto dei test da utilizzare;

```
@RunWith(JUnitPlatform.class)
@SelectPackages("it.unicam.cs.pa.battleship19.tests")
public class AllTests {}
```

- ***@SelectClasses***: specifica l'insieme dei test da eseguire;

```
@RunWith(JUnitPlatform.class)
@SelectClasses({AssertionTest.class, AssumptionTest.class
    , ExceptionTest.class})
public class AllTests {}
```

Asserzioni

Le asserzioni sono dichiarazioni o espressioni che esprimono una condizione o una verità che si presume sia vera in un determinato punto del codice. Negli ambienti di sviluppo e testing, le asserzioni vengono spesso utilizzate per verificare che le condizioni attese siano soddisfatte. Le asserzioni sono ampiamente utilizzate nei test unitari per verificare che il comportamento del software sia conforme alle aspettative.

- **assertThrow()**: verifica che una certa porzione di codice lancia una particolare eccezione;

```
@Test
void shouldThrowException() {
  int size = 10;
  BattleField field = new MatrixBattleField(size);
  IllegalLaunchException exception = assertThrows(
  IllegalLaunchException.class , () -> field.launch(new
  Location(size+1,size+1)));
  assertEquals("Illegal location",exception.getMessage());
}
```

- **assertTimeout()**: verifica che un certo test restituisca il risultato entro un certo periodo di tempo;

```
@Test
void timeoutNotExceeded() {
    assertTimeout(ofMinutes(1), () -> service.doBackup());
}
```

Logging

È il processo di registrazione degli eventi e delle attività che si verificano in un sistema o in un'applicazione. In altre parole, si tratta di una pratica che consente di catturare e registrare dati sul funzionamento di un sistema, come errori, avvisi, messaggi informativi, ecc.

Il logging è importante perché consente ai programmatori e agli operatori di sistema di monitorare il funzionamento dell'applicazione e di identificare eventuali problemi. In particolare, il logging può aiutare a:

- **Trovare e risolvere errori**: i log possono fornire informazioni su errori e bug nell'applicazione, aiutando i programmatori a individuare e risolvere i problemi;
- **Monitorare le prestazioni**: il logging può aiutare a monitorare le prestazioni dell'applicazione, identificando eventuali problemi di latenza o di carico elevato;
- **Verificare la sicurezza**: il logging può aiutare a individuare eventuali attività sospette o attacchi di sicurezza;
- **Eseguire l'audit**: il logging può aiutare a mantenere un registro delle attività dell'applicazione, consentendo di eseguire l'audit e verificare la conformità alle normative.

In Java, il logging è gestito tramite un framework di logging, utilizzando la classe *Logger* della libreria *java.util.logging*. Quando avviamo un'istanza della classe Logger possiamo definire un nome al quale verranno associati tutti i log di quella istanza.

Ogni log è caratterizzato da diversi livelli:

1. **SEVERE**: il livello di gravità più alto, utilizzato per indicare errori critici che impediscono il funzionamento del sistema;
2. **WARNING**: utilizzato per indicare situazioni di avviso che non sono necessariamente errori, ma che potrebbero indicare un problema potenziale;
3. **INFO**: utilizzato per registrare informazioni generali sul funzionamento del sistema, come ad esempio l'avvio del sistema o il completamento di un'operazione;
4. **CONFIG**: utilizzato per registrare informazioni sulla configurazione del sistema, come ad esempio la configurazione di un database o di una connessione di rete;
5. **FINE**: utilizzato per registrare messaggi di debug fine, che possono essere utili per il debug dell'applicazione;
6. **FINER**: utilizzato per registrare messaggi di debug più dettagliati di fine, che possono essere utili per il debug dell'applicazione in fasi di sviluppo;
7. **FINEST**: il livello di gravità più basso, utilizzato per registrare messaggi di debug estremamente dettagliati, che possono essere utili per il debug dell'applicazione in fasi di sviluppo avanzate.

Ciascun metodo di log ha come parametro una stringa che rappresenta il messaggio da registrare nel log. Alcuni dei metodi più comuni sono i seguenti:

- **severe(String message)**: registra un messaggio di errore critico nel log;
- **warning(String message)**: registra un messaggio di avviso nel log;
- **info(String message)**: registra un messaggio informativo nel log;
- **config(String message)**: registra un messaggio di configurazione nel log;
- **fine(String message)**: registra un messaggio di debug fine nel log;
- **finer(String message)**: registra un messaggio di debug più dettagliato di fine nel log;
- **finest(String message)**: registra un messaggio di debug estremamente dettagliato nel log;

Inoltre, è possibile utilizzare i seguenti metodi per registrare messaggi di log con informazioni aggiuntive:

- **log(Level level, String message)**: registra un messaggio di log con un determinato livello di gravità specificato dal parametro level;
- **log(Level level, Throwable exception, String message, Object[] params)**: registra un messaggio di log con un'eccezione e parametri aggiuntivi specificati dai parametri exception e params;
- **logp(Level level, String sourceClass, String sourceMethod, String msg)**: in questo metodo, il parametro *sourceClass* rappresenta il nome della classe che ha generato il messaggio di log, mentre *sourceMethod* rappresenta il nome del metodo all'interno della classe che ha generato il messaggio di log. Queste informazioni possono essere utili per il debug dell'applicazione, poiché consentono di identificare la posizione esatta in cui è stato generato un messaggio di log;
- **logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Object[] params)**: in questo metodo, oltre ai parametri di logp, viene specificato anche il nome del file di properties in cui sono definiti i messaggi di log (bundleName) e un array di oggetti (params) che possono essere utilizzati per sostituire i segnaposto nei messaggi di log.
  In generale, l'uso dei metodi **logp** e **logrb** è consigliato in situazioni in cui è importante identificare la posizione esatta in cui è stato generato un messaggio di log e quando si utilizzano file di properties per definire i messaggi di log localizzati in più lingue.

Log Record

È un oggetto che rappresenta un singolo evento di logging. Contiene informazioni sul messaggio di log, come il suo livello di gravità, il logger che lo ha generato, il momento in cui è stato generato e il messaggio stesso.

L'oggetto **LogRecord** è utilizzato internamente dalla libreria di logging di Java per rappresentare i messaggi di log e viene passato come parametro ai vari handler di logging, che possono essere utilizzati per registrare o visualizzare i messaggi di log.

Ogni LogRecord contiene almeno le seguenti informazioni:

- il **livello di gravità** del messaggio di log;
- il **nome del logger** che ha generato il messaggio di log;
- il **momento (tempo)** in cui il messaggio di log è stato generato;
- il **messaggio** di log vero e proprio.

Inoltre, un LogRecord può contenere anche altre informazioni, come ad esempio il nome della classe e del metodo che hanno generato il messaggio di log, il thread che ha generato il messaggio di log e qualsiasi parametro aggiuntivo che possa essere stato specificato dal programmatore.

L'oggetto LogRecord viene solitamente creato dalla libreria di logging di Java, ma può essere personalizzato o esteso per adattarsi alle esigenze specifiche dell'applicazione.

Log Handlers

È un oggetto che viene utilizzato per gestire i messaggi di log generati dal sistema. Il suo compito principale è quello di ricevere i messaggi di log inviati dal logger e determinare come gestirli, ad esempio scrivendoli su file, inviandoli via email o visualizzandoli sulla console.

La libreria di logging di Java fornisce diversi tipi di handler di logging predefiniti, come ad esempio:

- **ConsoleHandler**: scrive i messaggi di log sulla console utilizzando il *System.err* o il *System.out.stream*, a seconda del livello di gravità del messaggio di log;
- **FileHandler**: scrive i messaggi di log su un file di log specifico, che viene creato automaticamente se non esiste già. Il nome e la posizione del file di log possono essere specificati durante la creazione dell'istanza del *FileHandler*.
  Il FileHandler utilizza anche un meccanismo di rotazione dei file per gestire i file di log di grandi dimensioni. Quando il file di log raggiunge una certa dimensione massima, il FileHandler crea automaticamente un nuovo file di log e continua a scrivere i messaggi di log sul nuovo file. In questo modo, è possibile limitare le dimensioni del file di log e gestirlo in modo efficiente;

- **SocketHandler**: utilizza un socket di rete per inviare i messaggi di log, che possono essere gestiti da un server che si occupa dell'archiviazione e dell'analisi dei messaggi di log. Questo può essere utile per le applicazioni distribuite o per le applicazioni che eseguono su macchine remote.

Inoltre, è possibile estendere la classe Handler per creare handler di logging personalizzati. Un Handler di logging ha solitamente tre compiti principali:

- **Filtrare i messaggi di log** in base al loro livello di gravità: un Handler può essere configurato per gestire solo i messaggi di log con un determinato livello di gravità o per ignorare i messaggi di log con un livello di gravità inferiore a un certo valore;
- **Formattare i messaggi di log**: un Handler può essere configurato per formattare i messaggi di log in un determinato modo, ad esempio aggiungendo un timestamp o il nome del logger che ha generato il messaggio;
- **Gestire i messaggi di log**: un Handler può essere configurato per scrivere i messaggi di log su file, inviarli via email o visualizzarli sulla console, ad esempio.

Per utilizzare un Handler di logging, è necessario creare un'istanza dell'handler e associarla al logger corretto utilizzando il metodo addHandler del logger. In questo modo, ogni messaggio di log generato dal logger verrà inviato all'handler specificato per la gestione.

Formatters

Consentono di formattare i messaggi di log in un formato specifico. Il **Formatter** riceve in input un oggetto LogRecord, che contiene tutte le informazioni relative al messaggio di log, e restituisce una stringa che rappresenta il messaggio di log formattato.

Esistono diversi tipi di Formatter predefiniti in Java, tra cui:

- **SimpleFormatter**: formatta i messaggi di log in un formato semplice, che include il livello di gravità del messaggio, il nome del logger, il messaggio di log e la data di registrazione.
- **XMLFormatter**: formatta i messaggi di log in un formato XML, che facilita l'analisi e l'elaborazione dei messaggi di log da parte di altri programmi.
- **JSONFormatter**: formatta i messaggi di log in un formato JSON, che è utile per l'elaborazione dei messaggi di log da parte di altri programmi o servizi.

Inoltre, è possibile creare formatters personalizzati per soddisfare le esigenze specifiche dell'applicazione.

Log Manager

È una classe fornita dal framework di logging di Java che consente di gestire il comportamento di logging del sistema. Il **LogManager** si occupa di coordinare i Logger, gli Handler e i Formatter e di configurare il sistema di logging in base alle preferenze dell'applicazione.

La classe LogManager gestisce un singolo oggetto Logger radice, che rappresenta il logger di default per l'applicazione. Inoltre, il LogManager consente di definire logger aggiuntivi e di specificare gli Handler e i Formatter associati a ciascun logger.

Il comportamento di logging del LogManager può essere configurato utilizzando un file di configurazione di logging (*logging.properties*) o tramite il codice Java. Il file di configurazione di logging è un file di testo che specifica le preferenze di logging dell'applicazione, tra cui gli Handler, i Formatter e i livelli di gravità dei messaggi di log per ciascun logger.

```
// Otteniamo l'istanza del LogManager
LogManager logManager = LogManager.getLogManager();

// Carichiamo la configurazione di logging dal file di configurazione
try {
  InputStream config = new FileInputStream("logging.properties");
  logManager.readConfiguration(config);
} catch (IOException ex) {
  // Gestione dell'eccezione
}

// Otteniamo il logger per la nostra classe
Logger logger = Logger.getLogger(MyClass.class.getName());

// Impostiamo il livello di gravità del logger a "FINE"
logger.setLevel(Level.FINE);

// Aggiungiamo un FileHandler al logger
try {
  FileHandler fileHandler = new FileHandler("myapp.log");
  logger.addHandler(fileHandler);
} catch (IOException ex) {
  // Gestione dell'eccezione
}

// Configuriamo il formatter per il FileHandler
SimpleFormatter formatter = new SimpleFormatter();
fileHandler.setFormatter(formatter);
```

Es.

In questo esempio, il LogManager viene utilizzato per leggere la configurazione di logging dal file di configurazione *logging.properties*, per ottenere il logger per la classe *MyClass* e per aggiungere un FileHandler al logger. Infine, viene impostato un SimpleFormatter per il FileHandler.