

# Advanced Programming

## Lezione 2 – Languages and Abstract machines

Un computer è una **macchina fisica** che esegue algoritmi che sono opportunamente formalizzati in modo che la macchina possa eseguirli. Una **abstract machine** è un'astrazione del concetto di physical computer.

Gli algoritmi che vogliamo eseguire devono essere rappresentati utilizzando le istruzioni di un **linguaggio di programmazione L**. Un linguaggio di programmazione L è definito da sintassi, semantica e pragmatica.

La **Sintassi** indica come i costrutti del linguaggio sono disposti in modo da produrre un programma ben formato. Determina quali sequenze di caratteri costituiscono un programma valido.

Ogni linguaggio ha una sintassi **formalmente definita**.

La **Grammatica lessicale** definisce delle unità fondamentali (come parole chiave, operatori e identificatori) dette **token**. È descritta usando una **grammatica regolare**.

La **Grammatica sintattica** definisce le strutture del linguaggio, come le espressioni e le istruzioni, utilizzando una **grammatica libera dal contesto** (context-free grammar). Questa parte è usata dal compilatore per "scansionare" e "analizzare" il codice, in modo da verificare che il codice rispetti la sintassi.

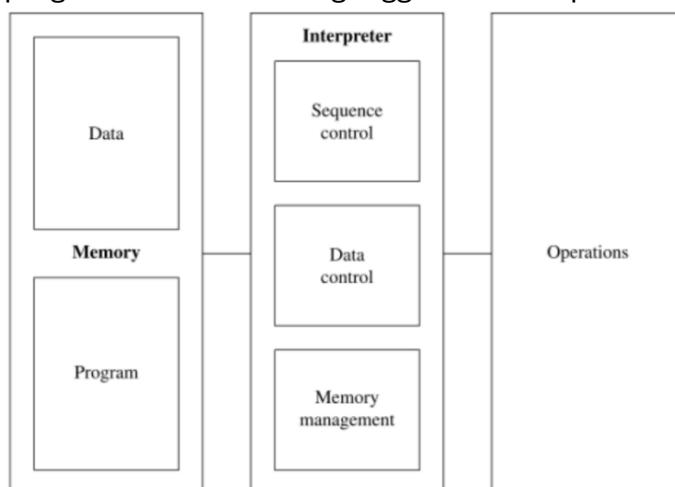
La **semantica** descrive il significato del programma, cioè cosa fa effettivamente il codice scritto in un linguaggio di programmazione. In genere è descritta in linguaggio naturale.

La **pragmatica** si riferisce all'uso pratico e alle convenzioni di stile che aiutano a scrivere codice in modo chiaro, leggibile ed elegante. Include anche la descrizione dei paradigmi di programmazione supportati.

**Paradigmi di programmazione:** Stili di programmazione che influenzano il modo in cui il codice è organizzato.

**Imperativo:** Uso di variabili, comandi e procedure. **Object-oriented:** Oggetti, classi, metodi. **Concorrenza:** Processi e comunicazione. **Funzionale:** Funzioni e valori, funzioni di ordine superiore. **Logico:** Dichiarazioni e relazioni logiche.

**Def. (Abstract Machine)** Sia L un linguaggio di programmazione. Una **abstract machine** ML per L è un qualsiasi insieme di **algoritmi e strutture** dati che può **eseguire** i programmi scritti nel linguaggio L. E' composta da una **memoria** e da un **interprete**.



La memoria serve a memorizzare i dati e i programmi, mentre l'interprete è il componente che esegue le istruzioni contenute nei programmi.

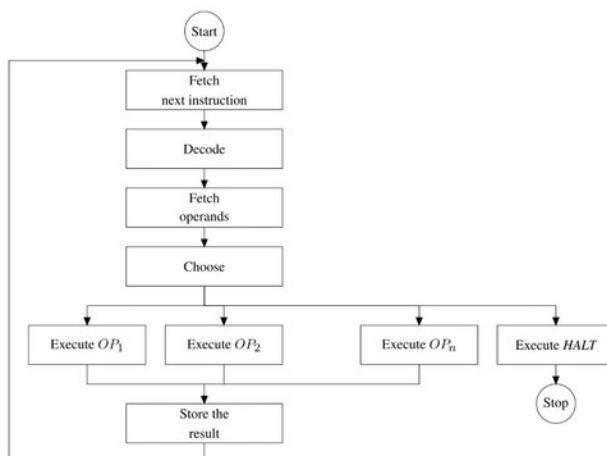
Ogni linguaggio L definisce implicitamente una Abstract Machine ML che ha L come linguaggio macchina.

Ogni programma scritto in un linguaggio L deve essere eseguibile. Implementare ML in una **macchina host esistente** M0 (via compilation/interpretation) rende il programma scritto in L eseguibile.

L'interprete di un linguaggio L esegue quattro tipi principali di operazioni: manipolazione di dati primitivi come numeri per eseguire algoritmi; controllo del flusso per gestire l'ordine delle istruzioni tramite salti e cicli; trasferimento di dati tra memoria e interprete per recuperare e utilizzare operandi; gestione dinamica della memoria, includendo allocazione e utilizzo di strutture come gli stack.

Il ciclo di esecuzione di un interprete è organizzato come una successione di step. Il primo step effettua il **fetch** dalla memoria della prossima istruzione da eseguire.

L'istruzione presa dalla memoria viene **decodificata** per determinare quale operazione deve essere eseguita, insieme ai suoi operandi. L'istruzione, che deve essere una primitiva della macchina, **viene eseguita**. Infine, i risultati vengono salvati in memoria e l'esecuzione passa alla prossima istruzione.



**Definition 1.2 (Machine language).** Data una abstract machine ML, il linguaggio L “compreso” dall'interprete di ML è chiamato **language machine** di ML.

Nel caso dell'assembly, l'interprete per le macchine hw è implementato come un insieme di dispositivi fisici che comprende la Control Unit e che supporta l'esecuzione del ciclo fetch-decode-execute.

Una abstract machine corrisponde univocamente ad un linguaggio, il suo machine language. Dato un linguaggio L, invece, possono esistere infinite abstract machines che adottano L come loro machine language. Queste machines differiscono da ogni altra dal modo in cui l'interprete è implementato e per le strutture dati che utilizzano. Implementare un programming language L significa implementare una abstract machine che ha L come suo machine language.

### Implementazione Abstract Machine

Qualsiasi implementazione di una abstract machine ML deve utilizzare un dispositivo fisico per eseguire le istruzioni di L. Le opzioni studiate per implementare una abstract machine sono tre:

**Implementation in hardware:** Le strutture dati e gli algoritmi che costituiscono la abstract machine sono implementate in hw. I circuiti fisici, quali memoria, circuiti

aritmetico-logici etc., sono utilizzati per implementare una macchina fisica il cui linguaggio macchina coincide con L.

**Simulation using software:** Le strutture dati e gli algoritmi richiesti da ML sono implementate utilizzando programmi scritti in un **altro linguaggio L'**, che sono già stati implementati. Usando la macchina **M' L'**, possiamo implementare la macchina ML usando appropriati programmi scritti in L', che interpretano i costrutti di L **simulando le funzionalità di ML**.

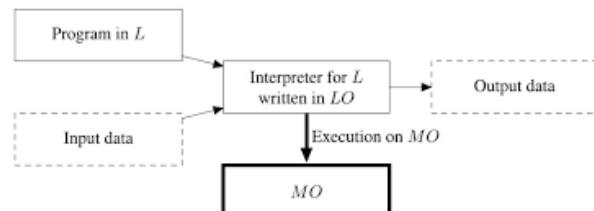
**Simulation (emulation) using firmware:** Consiste nella emulazione delle strutture dati e degli algoritmi for ML in microprogrammi, che utilizzano uno speciale linguaggio a basso livello che è memorizzato in una read-only memory e che può essere eseguito in una macchina fisica ad alta velocità.

### Caso ideale

Assumiamo che per la nostra implementazione di ML abbiamo a disposizione una abstract machine **M0L0** chiamata **host machine**, che è già implementata e consente di usare i costrutti del machina language L0 direttamente. Intuitivamente l'implementazione di L nella host machine M0L0 ha luogo utilizzando una **traduzione** da L a L0.

**Programma puramente INTERPRETATO:** In un'implementazione puramente interpretata, un **interprete I(L0->L)** per ML è un programma implementato in L0 che interpreta tutte le istruzioni di L.

Una volta implementato tale interprete, è sufficiente eseguire il programma I(L0->L) sulla macchina M0L0, con il programma da eseguire in L e i relativi input passati come suoi dati di input.



I programmi non sono tradotti esplicitamente, ma vi è solo una procedura di decoding, e il codice viene eseguito direttamente, senza che l'interprete fornisca un output intermedio formato dal codice tradotto in L0. L'interprete decodifica ed esegue il codice interpretato a runtime.

**Programma puramente COMPILATI:** L'implementazione di L viene effettuata con una **traduzione esplicita** dei programmi scritti in L in programmi scritti in L0, tramite un programma chiamato compilatore C(L->L0).

I programmi sono tradotti esplicitamente, fornendo un output intermedio formato dal codice tradotto in L0. Il codice oggetto tradotto viene eseguito in M0.

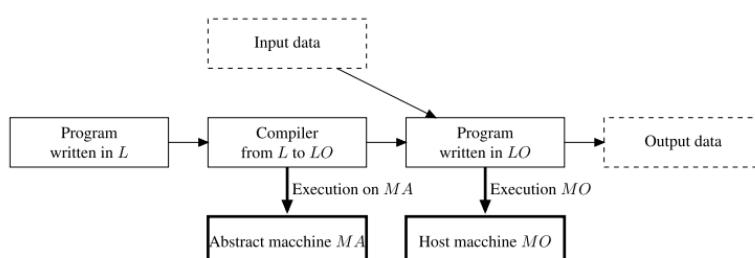


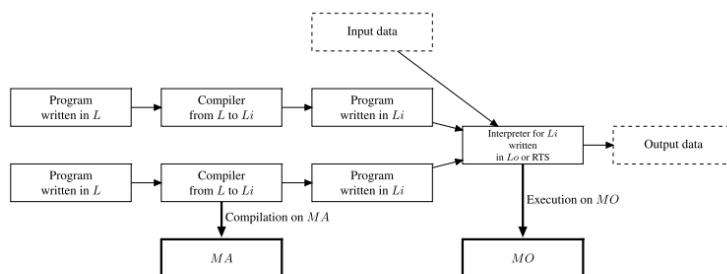
Fig. 1.5 Pure compiled implementation

Nell'interpretazione, un segmento di codice scritto in L iterato N volte viene decodificato N volte, implicando così un'inefficienza. Interpretare i costrutti del programma durante l'esecuzione permette un'interazione diretta con l'esecuzione runtime del programma. Il programma è memorizzato solo nella sua versione sorgente, senza produrre nuovo codice.

Nella compilazione, trascurando il tempo di compilazione, l'esecuzione di un programma compilato risulta più efficiente, poiché non vi è il sovraccarico della fase di decodifica delle istruzioni. La decodifica avviene una sola volta durante la compilazione e il codice prodotto viene eseguito più volte. A runtime viene persa la struttura del programma sorgente, rendendo più difficile l'interazione a runtime.

Nelle implementazioni interpretate **reali**, viene compilato L in un linguaggio intermedio L', che viene poi interpretato.

Nelle implementazioni compilate **reali**, molti costrutti complessi vengono simulati. Ad esempio, molte istruzioni di I/O possono essere tradotte in linguaggio macchina, ma è preferibile tradurle in chiamate di sistema, che simulano a runtime le istruzioni di alto livello.



**Fig. 1.6** Implementation: the real case with intermediate machine

Nelle implementazioni **reali** dei linguaggi ad alto livello L, abbiamo un compilatore C(L->Li) che traduce le istruzioni di L in un linguaggio intermedio Li e un interprete I(Li->L0) che decodifica ed esegue le istruzioni in L0 nella macchina M0L0.

Non è sempre necessario implementare un intero interprete I(Li->L0) nel caso il linguaggio Li e L0 non siano troppo distanti. Potrebbe essere sufficiente utilizzare l'interprete della host machine M0, esteso con appositi programmi chiamati "runtime support", in modo da simulare la macchina intermedia.

### Java Virtual Machine Example

Il linguaggio java utilizza una macchina intermedia chiamata Java Virtual Machine, che ha come machine language il Java Byte Code.

Il linguaggio Java viene prima compilato in Java Byte Code. La JVM agisce come un intermediario interpretando il bytecode per la macchina reale, permettendo al programma Java di funzionare su diverse piattaforme. In alcuni casi, la JVM può anche tradurre il bytecode direttamente in codice macchina, specifico per la piattaforma in uso, attraverso la compilazione *just-in-time* (JIT).

Per un nuovo linguaggio Lnew, basta fornire un compilatore che generi *bytecode* JVM, e il nuovo linguaggio potrà sfruttare le librerie Java esistenti.

### Gerarchia delle abstract machines

Possiamo immaginare una gerarchia di macchine M0L0, ..., MnLn, in cui la generica macchina MiLi è implementata usando il linguaggio della macchina immediatamente precedente Mi-1Li-1 e offre il suo linguaggio Li alla macchina Mi+1Li+1. MLi non può accedere direttamente alle risorse offerte dalle macchine sottostanti ma può solo fare

uso di qualsiasi cosa il linguaggio Li-1 offre. Ogni modifica ad un livello sottostante non deve alterare il funzionamento del livello soprastante.

Per esempio, al livello più basso troviamo l'hw implementato con dispositivi elettronici. Al di sopra, troviamo la microprogrammed machine. Al di sopra la abstract machine fornita dal SO che è implementata da programmi scritti in linguaggio macchina. Questa abstract machine fa da host machine a qualsiasi linguaggio di alto livello.

**La compilazione pura con linking statico** prevede che, durante il processo di compilazione, le routine delle librerie necessarie vengano collegate al codice oggetto del programma, creando un eseguibile che include tutte le dipendenze richieste.

**La compilazione, assemblaggio e collegamento statico** estende questo concetto, assicurando che tutto il codice, incluse le librerie, sia integrato nel programma già a tempo di compilazione, generando un eseguibile completamente autonomo.

**La compilazione, assemblaggio e collegamento dinamico** differisce perché le librerie necessarie, come le *Dynamic Link Libraries* (.dll, .so, .dylib), non vengono incorporate nell'eseguibile, ma collegate a runtime dal sistema operativo attraverso "stub" inclusi nel programma.

## Lezione 3: Runtime Systems and JVM

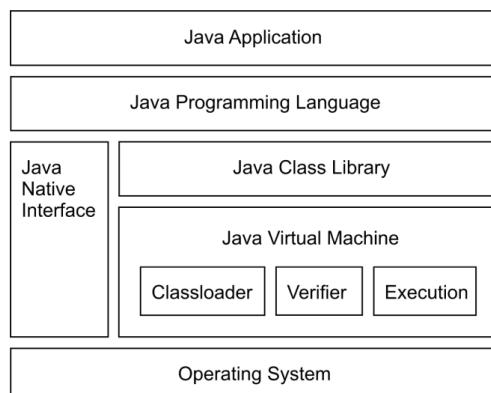
Il **modello di esecuzione** definisce come i programmi scritti in un determinato linguaggio vengono eseguiti.

Il **sistema runtime** è l'insieme di **strumenti e servizi** che il **programma utilizza** durante l'esecuzione, forniti dal compilatore e dal sistema operativo. Il compilatore non solo genera il codice eseguibile, ma crea anche un ambiente in cui i programmi possono essere eseguiti.

L'**ambiente runtime** si occupa **dell'allocazione** e della deallocazione della memoria (garbage collector) per gli oggetti e le variabili del programma. Assicura che le **chiamate a funzioni** o procedure siano correttamente collegate alle loro **definizioni**. Gestisce il meccanismo con cui i **parametri vengono passati** alle funzioni e alle procedure, sia per valore che per riferimento, secondo le regole del linguaggio. Fornisce un'interfaccia per le operazioni di **input/output (I/O)**, l'accesso ai file, la **gestione dei processi** e altre funzionalità che richiedono il coinvolgimento del sistema operativo. Gestisce il meccanismo delle **eccezioni**. Include il supporto per **thread** e processi paralleli, permettendo l'esecuzione di più operazioni contemporaneamente.

Il supporto runtime è essenziale sia per linguaggi interpretati che compilati, ma differisce in entità: i linguaggi interpretati generalmente richiedono un maggiore supporto runtime rispetto a quelli compilati.

### Java Runtime Environment



Il **Java Runtime Environment (JRE)** include tutto ciò che serve per eseguire programmi Java compilati. Comprende la **Java Virtual Machine** e la **JCL (Java Class Library)**, che è una raccolta di classi standard necessarie per il funzionamento dei programmi Java.

La **JVM** è una macchina astratta, che **non esiste** fisicamente. Infatti la sua specifica non fornisce dettagli implementativi come la gestione della memoria, ma tali dettagli possono variare a seconda della piattaforma di destinazione.

La specifica JVM definisce invece una struttura di **class file**, con forti vincoli sintattici, indipendente dalla macchina che tutte le implementazioni JVM devono supportare. Qualsiasi linguaggio di programmazione che può essere espresso in termini di un file di classe valido può teoricamente essere eseguito sulla JVM.

I file .class, che contengono la **rappresentazione esterna del codice** indipendente dalla piattaforma, vengono **caricati** nella JVM. Una volta caricati la rappresentazione interna dipende dall'**implementazione specifica** della JVM, che gestisce l'implementazione di oggetti, classi, metodi, array, stringhe e tipi primitivi.

Per i tipi primitivi, la JVM supporta byte, short, int, long, char, float, double, boolean (supportato solo per gli array) e returnAddress (per la gestione delle eccezioni).

Per i tipi di riferimento, supporta tipi di classi, tipi di array e tipi di interfacce.

A runtime non esiste informazione sui tipi delle variabili locali, i tipi degli operandi sono determinati dagli opcode delle operazioni (es: iadd per interi, fadd per float).

La rappresentazione degli oggetti è lasciata all'implementazione della JVM. Tuttavia, ci sono alcune caratteristiche generali:

La JVM utilizza un livello di indirezione per accedere ai dati di istanza e di classe, facilitando la garbage collection. Ogni oggetto deve includere: Un **mutex lock** (per la sincronizzazione dei thread) e flag di informazioni sullo stato del **garbage collection**.

## Java Thread

La JVM consente l'esecuzione di più thread contemporaneamente. Nella HotSpot JVM, ogni thread Java è direttamente mappato a un thread nativo del SO. Al momento dell'inizializzazione, dopo aver preparato tutto lo stato per un thread Java, come la memoria locale del thread, i buffer di allocazione, gli oggetti di sincronizzazione, gli stack e il program counter, la JVM crea il thread nativo.

Il SO è responsabile della schedulazione di tutti i thread e della loro distribuzione tra i core della CPU. Una volta che il thread nativo viene inizializzato, esso invoca il metodo run() del thread Java. Quando il metodo run() ritorna, le eccezioni non catturate vengono gestite, quindi il thread nativo conferma se la JVM deve essere terminata come risultato della terminazione del thread. Quando il thread termina, tutte le risorse del thread nativo e del Java thread vengono rilasciate.

Per la sua esecuzione Java utilizza numerosi thread in esecuzione in background, che vengono creati in aggiunta al thread Main.

I principali thread di sistema nella HotSpot JVM sono:

**VM Thread:** Esegue operazioni che richiedono un "safepoint", uno stato in cui l'heap è immutabile, necessario per attività safe come garbage collection e compilazione.

**Periodic Task Thread:** Pianifica ed esegue operazioni periodiche, come raccolta di statistiche o controllo dello stato della JVM.

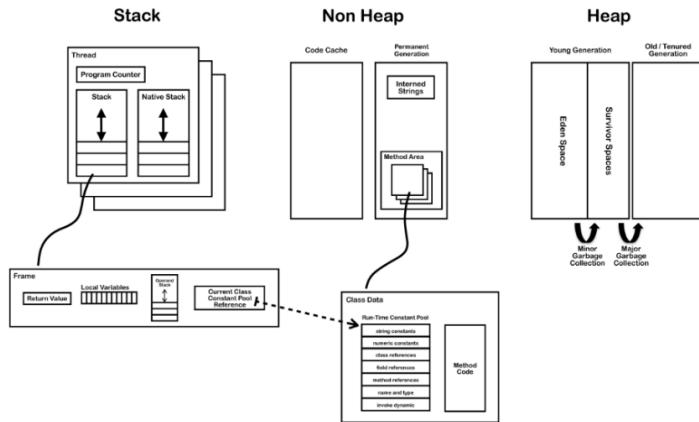
**GC Threads:** Responsabili della garbage collection, eseguono attività specifiche per liberare memoria non più utilizzata e ottimizzare l'heap.

**Compiler Threads:** Compilano il bytecode Java in codice nativo a runtime utilizzando tecniche come la compilazione Just-In-Time (JIT).

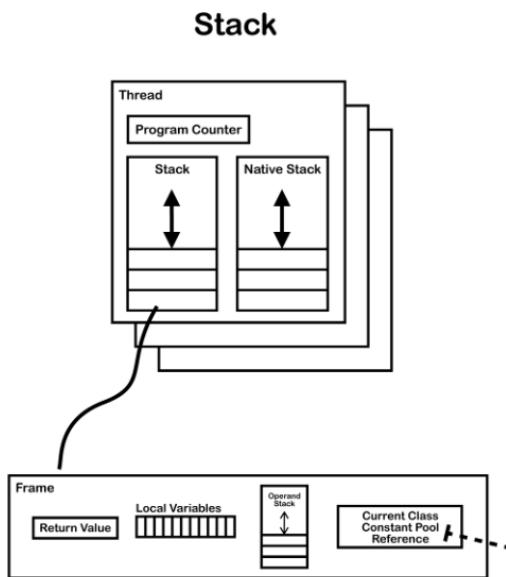
**Signal Dispatcher Thread:** Riceve segnali inviati al processo JVM e li gestisce chiamando i metodi appropriati della JVM.

**Metodo nativo:** Metodo implementato direttamente nel sistema operativo o in linguaggio nativo, come C.

## JVM Architecture



### Architecture per thread



Ogni thread eseguito nella JVM ha le seguenti componenti:

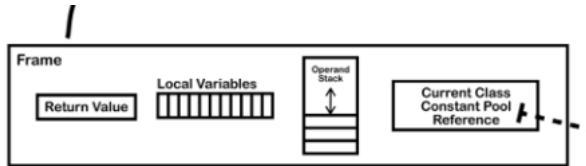
**Program Counter (PC):** È un registro che contiene l'indirizzo dell'istruzione corrente (o opcode) che il thread sta eseguendo. Dopo l'esecuzione di un'istruzione, il PC viene aggiornato per puntare all'istruzione successiva da eseguire. Il PC punta a un indirizzo di memoria nella **Method Area**, dove risiede il codice del metodo corrente.

Se il thread sta eseguendo un metodo nativo, il PC è considerato **indefinito**, poiché la JVM delega l'esecuzione al sistema sottostante.

**Stack:** Struttura dati **LIFO** (Last-In, First-Out) utilizzata per gestire le chiamate ai metodi. Per ogni metodo invocato, viene creato un **frame**, che contiene tutte le informazioni necessarie per eseguire il metodo, come le variabili locali e lo stato degli operandi. Il frame viene aggiunto in cima allo stack quando il metodo viene chiamato (**push**) e rimosso (**pop**) quando il metodo termina.

Lo stack può avere una dimensione dinamica o fissa. Se un thread richiede uno stack più grande di quello consentito, viene lanciato uno StackOverflowError. Se un thread richiede un nuovo frame e non c'è abbastanza memoria per allocarlo, viene lanciato un OutOfMemoryError.

**Native Stack:** Stack utilizzato per i metodi nativi. Un metodo nativo può invocare un metodo Java. Il controllo passa dallo stack nativo al normale stack Java, creando un frame Java sullo stack standard.

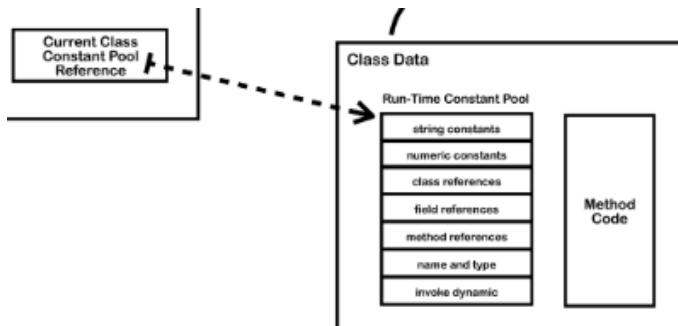


**Frame:** Per ogni invocazione di un metodo, viene creato un nuovo frame e aggiunto in cima allo stack. Il frame viene rimosso quando il metodo ritorna. Ogni frame contiene:

**Return Value:** Valore di ritorno del metodo

**Local variables array:** Contiene tutte le variabili utilizzate durante l'esecuzione del metodo. Per i metodi statici i parametri del metodo partono dallo slot zero, mentre per i metodi di istanza lo slot zero è riservato a this.

**Operand Stack:** Stack utilizzato per memorizzare i valori temporanei durante l'esecuzione di un metodo. Quando viene eseguita un'operazione (addizione), la JVM pusha gli operandi nella pila, estrae al momento dell'esecuzione gli operandi, e segue le operazioni e inserisce il risultato all'interno dell'operand stack.



**Reference to Constant Pool of the current class:** Struttura dati che contiene informazioni sul programma, come *Stringhe costanti*, *Nomi di metodi e campi*, *Valori numerici*, *Indirizzi di metodi* e *Riferimenti a classi e interfacce*, in modo da recuperarle rapidamente.

## Dynamic Linking

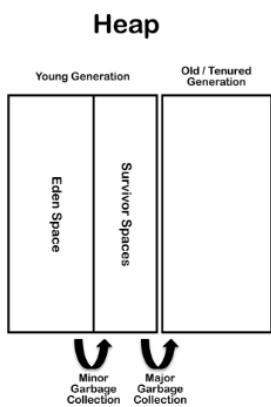
Il **dynamic linking** in Java si basa sull'uso del **runtime constant pool** per risolvere i riferimenti ai metodi e alle variabili **dinamicamente** durante l'esecuzione, anziché durante la compilazione. Ogni volta che un metodo viene eseguito, la JVM crea un **frame** per quel metodo. All'interno di questo frame, è presente un riferimento al **runtime constant pool** della classe che ha definito il metodo. Questo constant pool contiene riferimenti simbolici a variabili, metodi e classi utilizzate nel programma. Quando un programma Java viene compilato, i riferimenti a metodi e variabili non sono indirizzi di memoria fisici, ma **riferimenti simbolici**. La JVM può risolvere questi riferimenti simbolici in due momenti: Quando la classe viene caricata e verificata (**Eager/Statica**) o solo quando il riferimento simbolico viene utilizzato per la prima volta durante l'esecuzione (**Lazy/Tardiva**). In ogni caso, la JVM deve comportarsi come se

stesse risolvendo i riferimenti **la prima volta che vengono utilizzati**. Quando la JVM incontra un riferimento simbolico durante l'esecuzione lo **risolve** sostituendolo con un **riferimento diretto (binding)**.

Se un riferimento simbolico si riferisce a una classe che non è stata ancora caricata, la JVM provvede a caricare quella classe prima di procedere con la risoluzione. Dopo la risoluzione, la JVM memorizza il riferimento diretto come un **offset** rispetto alla struttura di memoria associata al metodo o alla variabile. Questo permette alla JVM di eseguire le future invocazioni in modo molto più efficiente, poiché non sarà necessario risolvere nuovamente il riferimento simbolico.

## Shared Between Threads

### Heap



L'**Heap** è utilizzato per allocare array e istanze di classi a runtime. Gli array e gli oggetti non vengono mai salvati nello stack perché un frame dello stack non è progettato per cambiare dimensione dopo la sua creazione. I frame salvano solo i **riferimenti** agli oggetti o array allocati nell'heap.

Gli oggetti non vengono rimossi automaticamente alla terminazione del metodo che li ha creati. La loro gestione è affidata al **garbage collector**, che li rimuove solo quando non sono più raggiungibili. Per ottimizzare il processo di garbage collection, l'heap è diviso in tre sezioni, **Eden Space**, dove vengono creati i nuovi oggetti e array, **Survivor Space**, dove vengono spostati gli oggetti sopravvissuti a un primo ciclo di (Minor) garbage collection, **Old Generation**: dove vengono spostati gli oggetti che hanno attraversato più cicli di (Major) garbage collection senza essere eliminati.

### Non-Heap Memory

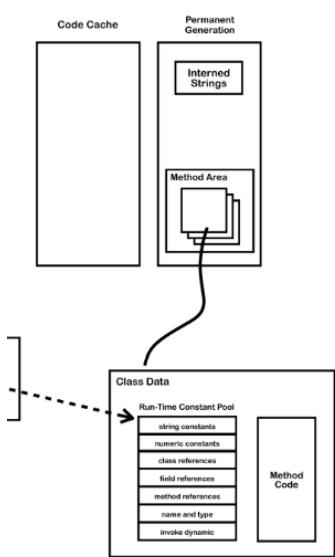
La **Non-Heap Memory** è l'area di memoria utilizzata per memorizzare oggetti che fanno parte delle meccaniche interne della JVM, piuttosto che per oggetti creati dinamicamente dalle applicazioni. Include:

**Method Area:** Contiene informazioni sulle classi caricate, come nomi di classi, metodi, campi, e bytecode del programma.

**Interned Strings:** Raccoglie e memorizza le stringhe immutabili, riducendo la duplicazione e ottimizzando l'uso della memoria per stringhe identiche.

**Code Cache:** Utilizzata dal **Just-In-Time (JIT) Compiler** per memorizzare metodi compilati in codice nativo. Consente l'accesso rapido ai metodi ottimizzati durante l'esecuzione, migliorando le prestazioni delle applicazioni.

## Non Heap



## Just In Time (JIT) Compilation

Il Java byte code viene interpretato, ma non è così veloce come l'esecuzione diretta del codice nativo sulla CPU host della JVM. Per migliorare le performance l'Oracle Hotspot VM cerca le "hot areas" del byte code che vengono eseguite frequentemente e le compila in codice nativo della macchina.

Il codice nativo viene poi salvato nella code cache nella non-heap memory. In questo modo l'Hotspot VM prova a scegliere il modo più appropriato per bilanciare il tempo extra di compilazione del codice con il tempo extra di esecuzione del codice interpretato.

## Method Area

La **Method Area** nella JVM è una parte della memoria condivisa tra tutti i thread, dove vengono memorizzate le informazioni relative alle classi che la JVM utilizza durante l'esecuzione di un programma. Contiene:

**Classloader Reference:** Un riferimento al **classloader** che ha caricato la classe in memoria.

**Runtime Constant Pool:** Contiene costanti come stringhe, numeri e riferimenti simbolici a metodi e campi.

**Numeric Constants:** Contiene valori costanti numerici come interi e floating-point utilizzati all'interno della classe.

**Field References:** Riferimenti ai campi (variabili) della classe. Vengono salvati il nome del campo, il tipo di dati del campo (es. int, String) e i modificatori (private, public, static, final).

FieldType descriptors

FieldType term	Type	Interpretation
B	byte	signed byte
C	char	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L ClassName ;	reference	an instance of class <i>ClassName</i>
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

**Method References:** Riferimenti ai metodi definiti nella classe. Vengono salvati il nome del metodo, il tipo di ritorno del metodo (es. void, int), i tipi dei parametri, elencati in ordine e i modificatori.

A *method descriptor* contains

- a sequence of zero or more *parameter descriptors* in brackets
- a *return descriptor* or V for void descriptor

Example: The descriptor of

`Object m(int i, double d, Thread t) {...}`

is:

`(IDLjava/lang/Thread;)Ljava/lang/Object;`

**Method code:** Contiene le istruzioni eseguibili della JVM corrispondenti al metodo, la dimensione massima della **operand stack** che il metodo utilizza, il numero di variabili locali che il metodo può contenere e la tabella che associa le variabili locali a nomi e tipi

Ogni metodo che gestisce le eccezioni ha una tabella che specifica il punto iniziale dell'area di codice da proteggere, il punto finale dell'area di codice, l'offset nel bytecode dove si trova il codice del gestore dell'eccezione e l'indice nel constant pool che identifica la classe dell'eccezione gestita.

Tutti i thread condividono la stessa method area, quindi gli accessi ai dati della method area e il processo di dynamic linking devono essere thread safe. Ad esempio, se due thread cercano di accedere a un metodo o campo di una classe che non è ancora stata caricata, la classe deve essere caricata solo **una volta**. Entrambi i thread devono attendere che la classe sia completamente caricata prima di continuare l'esecuzione.

## Struttura del Class File

# Class file structure

ClassFile {	
u4   magic;	0xCAFEBABE
u2   minor_version;	Java Language Version
u2   major_version;	
u2   constant_pool_count;	Constant Pool
cp_info   constant_pool[constant_pool_count-1];	
u2   access_flags;	access modifiers and other info
u2   this_class;	References to Class and Superclass
u2   super_class;	
u2   interfaces_count;	References to Direct Interfaces
u2   interfaces[interfaces_count];	
u2   fields_count;	Static and Instance Variables
field_info   fields[fields_count];	
u2   methods_count;	Methods
method_info   methods[methods_count];	
u2   attributes_count;	Other Info on the Class
attribute_info   attributes[attributes_count];	
}	

**magic, minor\_version, major\_version:** specificano le informazioni sulla versione della classe e la versione del JDK per cui è stata compilata.

**constant\_pool:** simile a una tabella dei simboli, ma contiene più dati.

**access\_flags:** fornisce l'elenco dei modificatori per questa classe (public...).

**this\_class:** indice nel constant\_pool che fornisce il nome completo della classe.

**super\_class:** indice nel constant\_pool che fornisce un riferimento simbolico alla superclasse.

**interfaces:** array di indici nel constant\_pool che fornisce riferimenti simbolici a tutte le interfacce implementate.

**fields:** array di indici nel constant\_pool che descrive ogni attributo.

**methods:** array di indici nel constant\_pool che descrive ogni firma di metodo. Se il metodo non è astratto o nativo, il bytecode è presente.

È possibile visualizzare il bytecode di una classe Java compilata utilizzando il comando javap.

```
package org.jvminternals;
public class SimpleClass {

    public void sayHello() {
        System.out.println("Hello");
    }
}
```

**javap -v -p -s -sysinfo -constants classes/org/jvminternals/SimpleClass.class**

# SimpleClass.class: constructor and method

```

Local variable 0 = "this"
{
    public org.jvminternals.SimpleClass();
        descriptor: ()V
        flags: ACC_PUBLIC
        Code:
            stack=1, locals=1, args_size=1
                0: aload_0
                1: invokespecial #1    // Method java/lang/Object."<init>": ()V
                4: return
        LineNumberTable:
            line 2: 0
}

public void sayHello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
        stack=2, locals=1, args_size=1
            0: getstatic    #2    // Field java/lang/System.out:Ljava/io/PrintStream;
            3: ldc         #3    // String Hello
            5: invokevirtual #4    // Method
        java/io/PrintStream.println:(Ljava/lang/String;)V
            8: return
    LineNumberTable:
        line 4: 0
        line 5: 8
}

```

*Method descriptors*

*Index into constant pool*

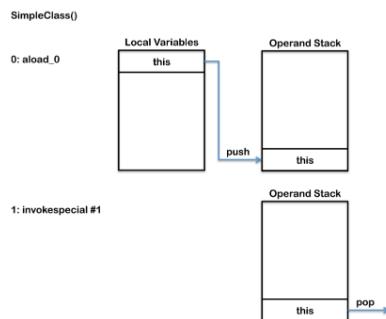
*String literal*

*Field descriptor*

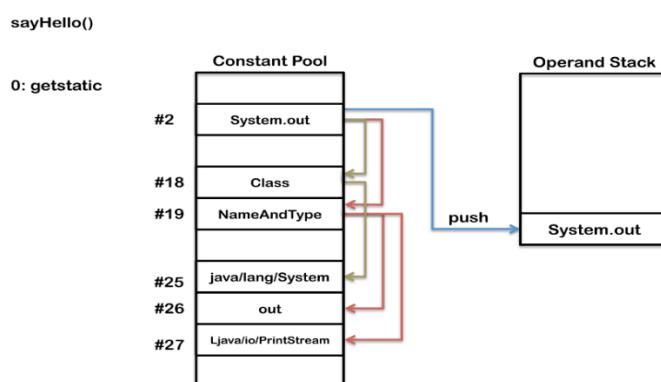
**Constant Pool:** simile a una tabella dei simboli e descritta in maggior dettaglio.

**Methods:** ogni metodo include quattro aree: firma e flag di accesso, bytecode, LineNumberTable – indica quale linea corrisponde a quale istruzione bytecode, LocalVariableTable – elenca tutte le variabili locali nel frame, in questo esempio l'unica variabile locale è this.

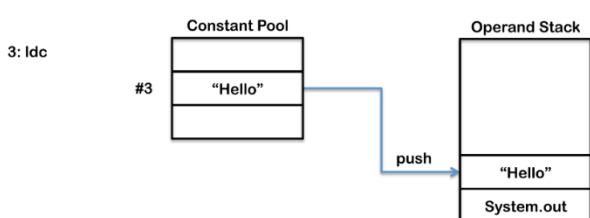
Come in qualsiasi bytecode tipico, la maggior parte degli operandi interagisce con le variabili locali, lo stack degli operandi e la **run time constant pool** nel seguente modo:  
Il costruttore ha due istruzioni: prima, il riferimento a this viene spinto nello stack degli operandi; successivamente, il costruttore della superclasse viene invocato, consumando il valore di this e quindi rimuovendolo dallo stack degli operandi.



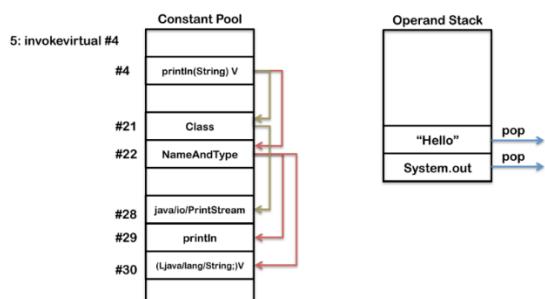
Il metodo `sayHello()` è più complesso, poiché deve risolvere riferimenti simbolici a riferimenti reali utilizzando la **run time constant pool**. Il primo operando, `getstatic`, viene utilizzato per spingere un riferimento al campo statico `out` della classe `System` nello stack degli operandi.



Il successivo operando, `ldc`, spinge la stringa "Hello" nello stack degli operandi.



L'ultimo operando, invokevirtual, invoca il metodo println di System.out, che rimuove "Hello" dallo stack degli operandi come argomento e crea un nuovo frame per il thread corrente.



## Classloader

La JVM si avvia caricando una classe iniziale utilizzando il **bootstrap classloader**. La classe viene poi collegata e inizializzata prima che venga invocato il metodo public static void main(String[]). L'esecuzione di questo metodo porterà al loading, al linking e all'inizializzazione di altre classi e interfacce, se necessario.

Il **loading** è il processo di **ricerca** del class file e la lettura di tale file in un byte array. Successivamente, i byte vengono analizzati per confermare che rappresentano un oggetto di tipo Class. Viene creato quindi un class or interface object dalla rappresentazione binaria.

Il **linking** è il processo di **verifica e preparazione** di una classe e della sua superclasse . Il collegamento consiste in tre fasi: verifica, preparazione e, facoltativamente, risoluzione.

**Verifica** è il processo di conferma che la **rappresentazione della classe sia strutturalmente corretta** e rispetti i requisiti semanticici del linguaggio di programmazione Java e della JVM. Ad esempio, vengono eseguiti i seguenti controlli: tabella dei simboli coerente e formattata correttamente, metodi/final class non sovrascritti, i metodi rispettano le parole chiave di controllo dell'accesso, i metodi hanno il numero corretto e il tipo corretto di parametri, il bytecode non manipola lo stack in modo errato, le variabili sono inizializzate prima di essere lette, le variabili hanno un valore del tipo corretto.

Durante il caricamento del file di classe si verifica che il file sia **formattato correttamente** e che tutti i suoi dati siano riconosciuti dalla JVM.

Durante il collegamento si effettuano tutti i controlli che **non coinvolgono le istruzioni** e si effettuano tutti i controlli che coinvolgono i metodi.

La prima volta che un metodo viene invocato si verifica che il **metodo o campo** referenziato esista nella classe specificata e il metodo correntemente in esecuzione abbia **accesso** al metodo o campo referenziato.

Eseguire questi controlli durante la fase di verifica significa che non devono essere ripetuti a runtime. La verifica durante il collegamento rallenta il caricamento delle classi, ma evita la necessità di eseguire questi controlli più volte durante l'esecuzione del bytecode.

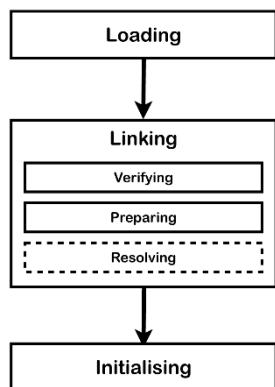
La **preparazione** comporta **l'allocazione di memoria** per lo storage statico e per **qualsiasi struttura dati** utilizzata dalla JVM, come le tabelle dei metodi. I campi statici vengono creati e inizializzati ai loro valori predefiniti.

La **Resolving** è una fase opzionale che prevede la verifica dei riferimenti simbolici, caricando le classi referenziate e controllando che i riferimenti siano corretti.

L'**inizializzazione** di una **classe** o interfaccia consiste **nell'esecuzione** del metodo di inizializzazione di classe **<clinit>**.

Il metodo **<clinit>** **inizializza le variabili di classe** (statiche) alla **prima invocazione** diretta di un metodo statico, **costruzione di un oggetto** o **accesso a un campo statico**. Gli inizializzatori statici vengono eseguiti e la superclasse deve essere inizializzata prima. L'inizializzazione è sincronizzata per garantire la sicurezza.

Il metodo **<init>** inizializza le **istanze degli oggetti** ed è invocato tramite l'istruzione **invokespecial**, solo su istanze non ancora inizializzate.



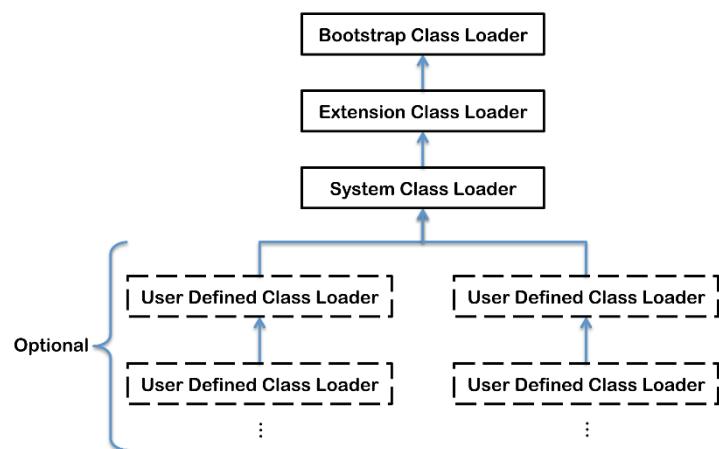
Nella JVM ci sono **classloader** multipli con ruoli diversi. Ogni classloader delega le sue operazioni al classloader genitore (che lo ha caricato), tranne il **bootstrap classloader**, che si trova al vertice della gerarchia.

Il **bootstrap classloader** è solitamente implementato come codice nativo perché viene istanziato molto presto, quando la JVM viene caricata. È responsabile del caricamento delle API Java di base, come ad esempio rt.jar. Carica solo le classi presenti nel **boot classpath**, che hanno un livello di fiducia più elevato; di conseguenza, evita molte delle validazioni che vengono eseguite per le classi normali.

L'**extension classloader** carica le classi dalle standard Java extension APIs, come ad esempio funzioni di estensione della sicurezza.

Il **system classloader** è il classloader predefinito delle applicazioni, che carica le classi delle applicazioni dal **classpath**.

I **classloader definiti dall'utente** possono essere utilizzati per caricare le classi delle applicazioni per scopi specifici, come il ricaricamento delle classi a runtime o la separazione tra diversi gruppi di classi caricate, solitamente necessaria nei server web come Tomcat.



## JVM Exit()

Una classe può essere unloaded dalla memoria quando non esistono più istanze della classe o l'oggetto di classe è irraggiungibile (non ha riferimenti attivi).

La JVM esce quando: Tutti i **thread non daemon** (thread normali) terminano o Viene chiamato Runtime.exit o System.exit, a patto che sia sicuro farlo.

## AP-05: The JVM instruction set

**Corra:** Cosa significa che nella JVM l'instruction set non è ortogonale? Significa che le istruzioni della JVM non sono generiche per tutti i tipi ma esistono istruzioni specifiche per ciascun tipo, come iadd per interi e fadd per numeri in virgola mobile.

Un'istruzione della Java Virtual Machine è formata da un **opcode** di un byte che specifica **l'operazione da eseguire**, seguito da zero o più **operandi** che forniscono gli **argomenti** utilizzati dall'operazione.

Molte istruzioni non hanno operandi e consistono solo in un opcode.

Il ciclo interno di un interprete della Java Virtual Machine è:

```
do {  
    atomically calculate pc and fetch opcode at pc;  
    if (operands) fetch operands;  
    execute the action for the opcode;  
} while (there is more to do);
```

Le istruzioni della JVM dipendono dalla dimensione degli operandi. Il numero e la dimensione degli operandi sono determinati dall'opcode.

Se un operando è più grande di un byte, allora è memorizzato in formato big-endian.

Le istruzioni in Java sono memorizzate in una sequenza di byte, e ogni istruzione è **allineata su un singolo byte**. Tuttavia per le istruzioni **lookupswitch** e **tableswitch** alcuni degli operandi devono essere allineati su confini di 4 byte, quindi vengono aggiunti byte di "riempimento" per rispettare questo allineamento.

La JVM usa **riferimenti simbolici** per fare riferimento a metodi, variabili e oggetti, che vengono risolti durante l'esecuzione. Quando la JVM effettua dei **salti**, questi sono sempre **relativi** alla posizione attuale nel bytecode, non a un indirizzo assoluto.

**L'operand stack** viene usato per passare argomenti ai metodi, restituire il risultato da un metodo e memorizzare risultati intermedi durante l'elaborazione di espressioni.

La JVM supporta tre modalità di indirizzamento per accedere ai dati:

**Immediate addressing mode:** Il valore costante è direttamente parte dell'istruzione. (Un numero può essere inserito direttamente nell'istruzione stessa).

**Indexed addressing mode:** Si accede alle variabili nell' **local variable array** utilizzando un indice.

**Stack addressing mode:** I valori vengono recuperati dall'**operand stack**.

Le **istruzioni dell'operand stack JVM prendono argomenti** dalla cima dell'operand stack (pop), eseguono delle operazioni e **posizionano il risultato** in cima all'operand stack (push).

La maggior parte delle istruzioni nel set di istruzioni della JVM codifica informazioni sul **tipo** all'interno delle operazioni che eseguono.

Ad esempio, l'istruzione **iload** carica il contenuto di una variabile locale, che deve essere un **int**, nell'operand stack. L'istruzione **fload** fa lo stesso con un valore **float**. Le due istruzioni possono avere implementazioni identiche, ma hanno opcode distinti.

Per la maggior parte delle istruzioni tipizzate, il tipo dell'istruzione è rappresentato esplicitamente nell'**mnemonico** dell'opcode da una lettera: **i** per operazioni su **int**, **l** per **long**, **s** per **short**, **b** per **byte**, **c** per **char**, **f** per **float**, **d** per **double**, e **a** per i riferimenti.

Alcune istruzioni, per cui il tipo è inequivocabile, non hanno una lettera di tipo nel mnemonico. Ad esempio, **arraylength** opera sempre su un oggetto che è un array. Alcune istruzioni, come **goto**, un salto incondizionato, non operano su operandi tipizzati.

i int	<b>iload</b>	integer load
l long	<b>lload</b>	long load
s short	<b>fload</b>	float load
b byte	<b>dload</b>	double load
c char	<b>aload</b>	reference-type load
f float		
d double		
a for reference		

Tuttavia, il set di istruzioni della Java Virtual Machine offre un livello ridotto di supporto per i tipi in certe operazioni, non comprendendo opcode per tutti i tipi. Alcune istruzioni possono essere utilizzate per convertire tra tipi di dati non supportati e tipi supportati.

opcode	byte	short	int	long	float	double	char	reference
<i>Tipush</i>	<i>bipush</i>	<i>sipush</i>						
<i>Tconst</i>			<i>iconst</i>	<i>lconst</i>	<i>fconst</i>	<i>dconst</i>		<i>acconst</i>
<i>Tload</i>			<i>iload</i>	<i>lload</i>	<i>fload</i>	<i>dload</i>		<i>aload</i>
<i>Tstore</i>			<i>istore</i>	<i>lstore</i>	<i>fstore</i>	<i>dstore</i>		<i>astore</i>
<i>Tinc</i>			<i>iinc</i>					
<i>Taload</i>	<i>baload</i>	<i>saload</i>	<i>iaload</i>	<i>laload</i>	<i>faload</i>	<i>daload</i>	<i>caload</i>	<i>aaload</i>
<i>Tastore</i>	<i>bastore</i>	<i>sastore</i>	<i>iastore</i>	<i>lastore</i>	<i>fastore</i>	<i>dstore</i>	<i>castore</i>	<i>aastore</i>
<i>Tadd</i>			<i>iadd</i>	<i>ladd</i>	<i>fadd</i>	<i>dadd</i>		
<i>Tsub</i>			<i>isub</i>	<i>lsub</i>	<i>fsub</i>	<i>dsub</i>		
<i>Tmul</i>			<i>imul</i>	<i>lmul</i>	<i>fmul</i>	<i>dmul</i>		
<i>Tdiv</i>			<i>idiv</i>	<i>ldiv</i>	<i>fdiv</i>	<i>ddiv</i>		
<i>Trem</i>			<i>irem</i>	<i>lrem</i>	<i>frem</i>	<i>drem</i>		
<i>Tneg</i>			<i>ineg</i>	<i>lneg</i>	<i>fneg</i>	<i>dneg</i>		
<i>Tshl</i>			<i>ishl</i>	<i>lshl</i>				
<i>Tshr</i>			<i>ishr</i>	<i>lshr</i>				
<i>Tushr</i>			<i>iushr</i>	<i>lushr</i>				
<i>Tand</i>			<i>iand</i>	<i>land</i>				
<i>Tor</i>			<i>ior</i>	<i>lor</i>				
<i>Txor</i>			<i>ixor</i>	<i>lxor</i>				
<i>i2T</i>	<i>i2b</i>	<i>i2s</i>		<i>i2l</i>	<i>i2f</i>	<i>i2d</i>		
<i>I2T</i>			<i>i2i</i>		<i>I2f</i>	<i>I2d</i>		
<i>f2T</i>			<i>f2i</i>	<i>f2l</i>		<i>f2d</i>		
<i>d2T</i>			<i>d2i</i>	<i>d2l</i>	<i>d2f</i>			
<i>Tcmp</i>				<i>lcmp</i>				
<i>Tcmpl</i>					<i>fcmpl</i>	<i>dcmpl</i>		
<i>Tcmpg</i>					<i>fcmpg</i>	<i>dcmpg</i>		
<i>if_TcmpOP</i>			<i>if_icmpOP</i>				<i>if_acmpOP</i>	
<i>Treturn</i>			<i>ireturn</i>	<i>lreturn</i>	<i>freturn</i>	<i>dreturn</i>		<i>areturn</i>

Un'istruzione specifica, con informazioni sul tipo, è costruita sostituendo la **T** nel modello di istruzione della colonna opcode con la lettera indicata nella colonna tipo. Se la colonna tipo per un certo modello di istruzione è vuota, allora non esiste un'istruzione che supporti quel tipo di operazione.

È importante notare che la maggior parte delle istruzioni nella tabella non supporta i tipi **byte**, **char** e **short**. Nessuna istruzione supporta il tipo **boolean**. Un compilatore codifica il caricamento di valori letterali dei tipi **byte** e **short** usando istruzioni che convertono quei valori in **int**, estendone il valore con degli zeri al momento della compilazione o dell'esecuzione.

Tpush	Inserisce un valore di tipo T nello stack.
Tconst	Inserisce una costante di tipo T nello stack.
Tload	Carica una variabile locale di tipo T nello stack.
Tstore	Salva il valore dello stack in una variabile locale di tipo T.
Tinc	Incrementa una variabile locale di tipo T.
Taload	Carica un elemento di tipo T da un array nello stack.
Tastore	Salva un elemento di tipo T dello stack in un array.
Tadd	Esegue l'addizione tra due valori di tipo T nello stack.
Tsub	Esegue la sottrazione tra due valori di tipo T nello stack.
Tmul	Esegue la moltiplicazione tra due valori di tipo T nello stack.
Tdiv	Esegue la divisione tra due valori di tipo T nello stack.
Trem	Calcola il resto della divisione tra due valori di tipo T nello stack.
Tneg	Negazione di un valore di tipo T nello stack.
Tshl	Esegue uno shift a sinistra su un valore di tipo T nello stack.
Tshr	Esegue uno shift a destra aritmetico su un valore di tipo T nello stack.
Tushr	Esegue uno shift a destra logico su un valore di tipo T nello stack.
Tand	Esegue l'operazione AND bit-a-bit tra due valori di tipo T nello stack.
Tor	Esegue l'operazione OR bit-a-bit tra due valori di tipo T nello stack.
Txor	Esegue l'operazione XOR bit-a-bit tra due valori di tipo T nello stack.
T2T	Converte un valore da un tipo T a un altro tipo T.
Tcmp	Confronta due valori di tipo T e produce un risultato (es: -1, 0, 1).
TcmpOP	Confronta due valori di tipo T usando un'operazione condizionale (es: <, ==).
Treturn	Ritorna un valore di tipo T dal metodo corrente.

## Istruzioni di Load e Store

Le istruzioni di **load** e **store** trasferiscono valori tra il **local variable array** (o una costante) e **l'operand stack** di un frame della JVM:

**Caricare la variabile locale** di indice n nell'Operand Stack: Tload n

**Salvare un valore** dall'operand stack nella variabile locale di indice n: Tstore n

**Caricare una costante** nello stack degli operandi: (si/)bipush valCostante, Tconst valCostante, ldc2\_w valCostante

**Accedere a più variabili locali** utilizzando un indice più ampio o un operando immediato più grande: wide.

Assembly code      Binary instruction code layout

iload_0	26	Pushes local variable 0 on operand stack		
iload_1	27			
iload_2	28			
iload_3	29			
iload n	21	n		
wide iload n	196	21	n	

### 2.11.3. Arithmetic Instructions

Le istruzioni aritmetiche prendono due valori dall'operand stack, calcolano il risultato e reinseriscono il risultato, nell'operand stack. Le principali istruzioni sono Tadd, Tsub, Tmul, Tneg, iand, ior, ixor.

Remainder/Modulo: irem, lrem, frem, drem. Shift: ishl, ishr, iushr, lshl, lshr, lushr. Local variable increment: iinc. Comparison: dcmpg, dcmpl, fcmpg, fcmpl, lcmp.

**Istruzione iadd:** viene eseguita prendendo i primi due valori, value1 e value2 presenti nell'operand stack di tipo int, calcolando value1 + value2 e inserendolo nell'operand stack.

Il risultato è costituito dai 32 bit meno significativi del vero risultato matematico in un formato sufficientemente ampio a complemento a due, rappresentato come un valore di tipo int. Se si verifica un overflow, il segno del risultato potrebbe non essere lo stesso del segno della somma matematica dei due valori. Nonostante l'overflow possa verificarsi, l'esecuzione di un'istruzione iadd non genera mai un'eccezione a tempo di esecuzione.

### Use di costanti variabili locali e costrutti di controllo

Il metodo `spin()` esegue a vuoto un ciclo for 100 volte:

Lo 0 viene inserito utilizzando un'istruzione `iconst_0`. L'istruzione `istore_1` preleva un int dall'operand stack e lo memorizza nella variabile locale 1. L'istruzione `iinc 1 1` incrementa variabile locale con slot 1 di 1 (`i++`). L'istruzione `iload_1` inserisce il valore della variabile locale 1 nello stack degli operandi.

L'istruzione `bipush` inserisce il valore 100 nell'operand stack come `int`, quindi l'istruzione `if_icmplt` rimuove quel valore dall'operand stack e lo confronta con `i`.

- Sample Code

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ; // Loop body is empty
    }
}
```

- Can compile to

0 <code>iconst_0</code> 1 <code>istore_1</code> 2 <code>goto 8</code> 5 <code>iinc 1 1</code> 8 <code>iload_1</code> 9 <code>bipush 100</code> 11 <code>if_icmplt 5</code> 14 <code>return</code>	// Push int constant 0 // Store into local variable 1 (i=0) // First time through don't increment // Increment local variable 1 by 1 (i++) // Push local variable 1 (i) // Push int constant 100 // Compare and loop if less than (i < 100) // Return void when done
--	---

Se il confronto ha esito positivo (la variabile `i` è minore di 100), il controllo viene trasferito all'indice 5 e inizia la successiva iterazione del ciclo `for`. Altrimenti, il controllo passa all'istruzione successiva a `if_icmplt`.

### int vs. double: lack of opcodes for double

#### requires longer bytecode

- Sample Code

```
void dspin() {
    double i;
    for (i = 0.0; i < 100.0; i++) {
        ; // Loop body is empty
    }
}
```

- Can compile to

0 <code>dconst_0</code> 1 <code>dstore_1</code> 2 <code>goto 9</code> 5 <code>dload_1</code> 6 <code>dconst_1</code> 7 <code>dadd</code> 8 <code>dstore_1</code> 9 <code>dload_1</code> 10 <code>ldc2_w #4</code> 13 <code>dcmpg</code> 14 <code>iflt 5</code> 17 <code>return</code>	// Push double constant 0.0 // Store into local variables 1 and 2 // First time through don't increment // Push local variables 1 and 2 // Push double constant 1.0 // Add; there is no dinc instruction // Store result in local variables 1 and 2 // Push local variables 1 and 2 // Push double constant 100.0 // There is no if_dcmplt instruction // Compare and loop if less than(i < 100.0) // Return void when done
--	--

16

Si ricorda che i valori di tipo `double` occupano due variabili locali, anche se vengono acceduti solo usando l'indice inferiore delle due variabili locali.

Non tutte le istruzioni sono implementate per i double. Ad esempio, il confronto di valori di tipo `int` nell'istruzione `for` dell'esempio `spin` può essere implementato utilizzando una singola istruzione `if_icmplt`; tuttavia, non esiste un'istruzione singola nel set di istruzioni che esegua un salto condizionale su valori di tipo `double`. Pertanto, `dspin` deve implementare il confronto di valori di tipo `double` utilizzando un'istruzione `dcmpg` seguita da un'istruzione `iflt`.

### 3.4. Accessing the Run-Time Constant Pool

- Sample Code

```
void useManyNumeric() {  
    int i = 100;  
    int j = 1000000;  
    long l1 = 1;  
    long l2 = 0xffffffff;  
    double d = 2.2;  
    ...do some calculations... }
```

- Can compile to

0 bipush 100 2 istore_1 3 ldc #1 5 istore_2 6 lconst_1 7 lstore_3 8 ldc2_w #6 11 lstore_5 13 ldc2_w #8 16 dstore_7 ...	// Push small int constant with bipush  // Push large int (1000000) with ldc  // A tiny long value uses fast lconst_1  // Push long 0xffffffff (that is, int -1) // Any long can be pushed with ldc2_w // Push double constant 2.200000  ...do those calculations...
--	--

Molte costanti numeriche, così come oggetti, campi e metodi, vengono acceduti tramite il **run-time constant pool** della classe corrente.

Le istruzioni **ldc** e **ldc\_w** vengono utilizzate per accedere ai valori nel **run-time constant pool** tramite il suo indice. L'istruzione **ldc\_w** viene utilizzata al posto di **ldc** solo quando ci sono molti elementi nel **run-time constant pool** e un indice più grande è necessario per accedere a un elemento. L'istruzione **ldc2\_w** viene utilizzata per accedere ai valori **double** e **long**.

### 3.6. Receiving Arguments

- Sample Code

```
int addTwo(int i, int j) {  
    return i + j;  
}
```

- Can compile to

0 iload_1 1 iload_2 2 iadd 3 ireturn	// Push value of local variable 1 (i) // Push value of local variable 2 (j) // Add; leave int result on operand stack // Return int result
---	---

- Local variable 0 used for **this** in instance methods

- Sample Code

```
static int addTwo(int i, int j) {  
    return i + j;  
}
```

- Can compile to

0 iload_0 1 iload_1 2 iadd 3 ireturn
---

18

### 3.7 Invoking Methods

La normale invocazione di un metodo di istanza si basa sul tipo dell'oggetto a runtime. Tale invocazione è implementata utilizzando l'istruzione **invokevirtual**, che prende come argomento un indice a una entry del **run-time constant pool** che fornisce la forma interna del nome binario del tipo di classe dell'oggetto, il nome del metodo da invocare e il method descriptor.

- Sample Code

```
int add12and13() {  
    return addTwo(12, 13);  
}
```

- Can compile to

0 aload_0 1 bipush 12 3 bipush 13 5 invokevirtual #4 8 ireturn	// Push local variable 0 (this) // Push int constant 12 // Push int constant 13 // Method Example.addtwo(II)I // Return int on top of operand stack; // it is the int result of addTwo()
--	---

Viene inserito il riferimento a **this** nell'operand stack e gli argomenti **int 12** e **13**. Quando viene creato il frame per il metodo **addTwo**, gli argomenti passati al metodo diventano i

valori iniziali delle variabili locali del nuovo frame. Infine, **addTwo** viene invocato. Quando restituisce, il suo valore di ritorno **int** viene pushato nell'operand stack del frame del metodo invocatore.

L'istruzione **ireturn** prende il valore **int** restituito da **addTwo**, presente nell'operand stack del frame corrente, e lo pusha nell'operand stack del frame dell'invocatore. Poi restituisce il controllo all'invocatore.

**invokestatic**: Viene utilizzato per chiamare metodi con modificatori "static". Non passa **this** e copia gli argomenti nelle variabili locali a partire dall'indice 0.

**invokespecial**: Utilizzato per chiamare costruttori, metodi privati o metodi della superclasse. In questo caso, **this** viene sempre passato.

**invokeinterface**: Simile a **invokevirtual**, ma viene utilizzato quando il metodo chiamato è dichiarato in un'interfaccia. Richiede un diverso tipo di ricerca del metodo.

**invokedynamic**: Introdotto in Java SE 7 per supportare il tipo dinamico. Verrà discusso in relazione alle espressioni lambda.

## Working with objects

- Sample Code
- Can compile to

```
Object create() {  
    return new Object();  
}
```

```
0 new #1           // Class java.lang.Object  
3 dup  
4 invokespecial #4 // Method java.lang.Object.<init>()V  
7 areturn
```

Le istanze di classe nella JVM vengono create utilizzando l'istruzione **new**. Il costruttore di una classe, a livello JVM, è rappresentato da un metodo speciale denominato **<init>**, generato automaticamente dal compilatore. Questo metodo funge da "metodo di inizializzazione dell'istanza". Una classe può avere più metodi **<init>**, ognuno corrispondente a un costruttore diverso definito nella classe. Dopo la creazione dell'istanza, tutte le variabili di istanza, comprese quelle ereditate dalle superclassi, vengono inizializzate ai loro valori predefiniti. Successivamente, uno dei metodi **<init>** viene invocato per completare l'inizializzazione. Le istanze di classe, essendo tipi di riferimento, vengono passate e restituite in modo simile ai valori numerici, ma utilizzano un insieme specifico di istruzioni per la gestione dei riferimenti.

## Accessing fields (instance variables)

- Sample Code
- Can compile to

```
void setIt(int value) {  
    i = value;  
}  
int getIt() {  
    return i;  
}
```

```
Method void setIt(int)  
0 aload_0  
1 iload_1  
2 putfield #4 // Field Example.i I  
5 return  
Method int getIt()  
0 aload_0  
1 getfield #4 // Field Example.i I  
4 ireturn
```

I campi di un oggetto vengono accessi utilizzando le istruzioni **getfield** e **putfield**, seguiti da un offset del runtime constant pool che indicano il campo a cui accedere.

## 3.9. Arrays

Gli array della JVM sono considerati oggetti. L'istruzione **newarray** viene utilizzata per creare un array di tipo numerico. L'istruzione **anewarray** viene utilizzata per creare un array monodimensionale di riferimenti a oggetti.

## Using Arrays

- Sample Code
- Can compile to

```
void createBuffer() {
    int buffer[];
    int bufsz = 100;
    int value = 12;
    buffer = new int[bufsz];
    buffer[10] = value;
    value = buffer[11];
}
```

0 bipush 100	// Push int constant 100 (bufsz)
2 istore_2	// Store bufsz in local variable 2
3 bipush 12	// Push int constant 12 (value)
5 istore_3	// Store value in local variable 3
6 iload_2	// Push bufsz and...
7 newarray int	// ... create new int array of that length
9 astore_1	// Store new array in buffer
10 aload_1	// Push buffer
11 bipush 10	// Push int constant 10
13 iload_3	// Push value
14 iastore	// Store value at buffer[10]
15 aload_1	// Push buffer
16 bipush 11	// Push int constant 11
18 iaload	// Push value at buffer[11]...
19 istore_3	// ...and store it in value
20 return	

23

## 3.10. Compiling Switches

Le istruzioni **switch** sono compilate dalla JVM utilizzando le istruzioni **tableswitch** e **lookupswitch**.

L'istruzione **tableswitch** viene utilizzata quando i case dello switch possono essere rappresentati come indici all'interno di un **range continuo**.

Il target default dello switch viene utilizzato se il valore dell'espressione dello switch ricade al di fuori del range di indici validi.

## Compiling switches (1)

- Sample Code
- Can compile to

```
int chooseNear(int i) {
    switch (i) {
        case 0: return 0;
        case 1: return 1;
        case 2: return 2;
        default: return -1;
    }
}
```

0 iload_1	// Push local variable 1 (argument i)
1 tableswitch 0 to 2:	// Valid indices are 0 through 2
0: 28	// If i is 0, continue at 28
1: 30	// If i is 1, continue at 30
2: 32	// If i is 2, continue at 32
default:34	// Otherwise, continue at 34
28 iconst_0	// i was 0; push int constant 0...
29 ireturn	// ...and return it
30 iconst_1	// i was 1; push int constant 1...
31 ireturn	// ...and return it
32 iconst_2	// i was 2; push int constant 2...
33 ireturn	// ...and return it
34 iconst_m1	// otherwise push int constant -1...
35 ireturn	// ...and return it

L'istruzione **lookupswitch** viene utilizzata quando i case dello switch sono sparsi. Associa chiavi di tipo **int** (i valori delle etichette dei casi) ad offset di destinazione in una tabella. Quando un'istruzione **lookupswitch** viene eseguita, il valore dell'espressione dello switch viene confrontato con le chiavi nella tabella. Se una delle chiavi corrisponde al valore dell'espressione, l'esecuzione continua all'offset di destinazione associato. Se nessuna chiave corrisponde, l'esecuzione continua al target predefinito. Le chiavi sono ordinate, quindi la ricerca è binaria. Gli switch funzionano solo col tipo int, quindi altri tipi richiedono conversione.

```

int chooseFar(int i) {
    switch (i) {
        case -100: return -1;
        case 0:    return 0;
        case 100:   return 1;
        default:   return -1;
    }
}

```

```

Method int chooseFar(int)
0  iload_1
1  lookupswitch 3:
     -100: 36
       0: 38
      100: 40
     default: 42
36  iconst_m1
37  ireturn
38  iconst_0
39  ireturn
40  iconst_1
41  ireturn
42  iconst_m1
43  ireturn

```

## Operations on the Operand Stack

- Sample Code
- Can compile to

```

public long nextIndex() {
    return index++;
}
private long index = 0;

```

0 aload_0	// Push this
1 dup	// Make a copy of it
2 getfield #4	// One of the copies of this is consumed // pushing long field index, // above the original this
5 dup2_x1	// The long on top of the operand stack is // copied into the operand stack below the // original this
6 lconst_1	// Push long constant 1
7 ladd	// The index value is incremented...
8 putfield #4	// ...and the result stored in the field
11 lreturn	// The original value of index is on top of // the operand stack, ready to be returned

La istruzione **dup** duplica il valore in cima allo stack degli operandi e lo spinge di nuovo nello stack.

L'istruzione **dup2\_x1** duplica i due valori in cima allo stack e li spinge tre posizioni più in basso nello stack. (Prima: [B, A] Dopo: [B, A, B, A])

## 3.12. Throwing and Handling Exceptions

- Sample Code
- Can compile to

```

void cantBeZero(int i) throws TestExc
{
    if (i == 0) {
        throw new TestExc();
    }
}

```

0 iload_1	// Push argument 1 (i)
1 ifne 12	// If i==0, allocate instance and throw
4 new #1	// Create instance of TestExc
7 dup	// One reference goes to its constructor
8 invokespecial #7	// Method TestExc.<init>()V
11 athrow	// Second reference is thrown
12 return	// Never get here if we threw TestExc

Quando viene eseguita l'istruzione **athrow** per lanciare un'eccezione, la JVM cerca un blocco di cattura (catch block) all'interno del metodo corrente. Questo viene fatto utilizzando una tabella delle eccezioni, che contiene informazioni su dove si trovano i blocchi di cattura per le eccezioni lanciate.

**Se trova un blocco catch**, l'operand stack viene **svuotato**. Il controllo viene trasferito al **primo comando** del blocco di cattura corrispondente, che gestisce l'eccezione.

**Se non trova un blocco catch**, il **frame corrente** viene **scartato**. L'eccezione viene **rilanciata** al chiamante. Se alla fine l'eccezione non viene catturata da nessun blocco di

cattura in nessun metodo della catena di metodi invocati, il **thread** in esecuzione viene **terminato**.

## try-catch

- Sample Code
- Can compile to

<pre>void catchOne() {     try {         tryItOut();     } catch (TestExc e) {         handleExc(e);     } }</pre>	<pre>0  aload_0 1  invokevirtual #6 4  return 5  astore_1 6  aload_0 7  aload_1 8  invokevirtual #5 11 return</pre>	<pre>// Beginning of try block // Method Example.tryItOut()V // End of try block; normal return // Store thrown value in local var 1 // Push this // Push thrown value // Invoke handler method: // Example.handleExc(LTestExc;)V // Return after handling TestExc</pre>
<b>Exception table:</b>		<ul style="list-style-type: none"><li>• Compilation of <b>finally</b> more tricky</li></ul>

Se durante l'esecuzione del blocco try non viene lanciata alcuna eccezione, si comporta come se il try non fosse presente: viene invocato tryItOut e restituisce. La tabella delle eccezioni per il metodo catchOne ha una voce corrispondente all'unico argomento (un oggetto TestExc) che la clausola catch di catchOne può gestire.

Se durante l'esecuzione viene lanciata un'eccezione istanza di TestExc, il controllo viene trasferito al codice all'indice 5, che implementa il blocco della clausola catch. Se il valore lanciato non è un'istanza di TestExc, la clausola catch non può gestirlo, e l'eccezione viene rilanciata all'invocatore del metodo.

## Altre Istruzioni

**monitoreenter:** Inizia la sincronizzazione su un oggetto.

**monitorexit:** Termina la sincronizzazione su un oggetto.

**instanceof:** Controlla se un oggetto è un'istanza di una determinata classe o interfaccia.

**checkcast:** Verifica che un oggetto possa essere convertito in un tipo specificato.

**nop:** Un'istruzione che non esegue alcuna azione.

## Limitazioni della JVM

**Numero massimo di entry nel costant pool:** 65.535

**Numero massimo di campi, metodi e superinterfacce dirette:** 65.535

**Numero massimo di variabili nel local variavle array di un frame:** 65.535, a causa dell'indicizzazione delle variabili locali a 16 bit dell'istruzione della JVM.

**Dimensione massima dell'operand stack:** 65.535.

**Numero massimo di parametri di un metodo:** 255

**Lunghezza massima dei nomi di campi e metodi:** 65.535 caratteri

**Numero massimo di dimensioni in un array:** 255

## JIT Compilation vs AOT Compilation and Interpretation

**AOT Compilation:** **Compila** il codice in modo completo **prima** dell'esecuzione, ottimizzando le prestazioni grazie all'allocazione efficiente delle variabili e ottimizzazioni hardware. Riduce il costo dell'elaborazione a run-time, ma sacrifica flessibilità a runtime.

**AOT Interpretation:** **Interpreta** il codice linea per linea, permettendo un debug interattivo a runtime. Consente di modificare valori delle variabili e invocare procedure a runtime, risultando utile in contesti di sviluppo e testing.

**JIT (Just-In-Time) Compilation:** Tecnica che combina compilazione e interpretazione AOT. In JIT, il codice sorgente viene compilato in **codice macchina al momento dell'esecuzione**, anziché in anticipo. Mantiene la flessibilità dell'interpretazione, rendendo possibile modificare il codice e vedere immediatamente i risultati senza dover ricompilare tutto.

Il JIT compiler ha accesso a informazioni di runtime, il che significa che può osservare come il programma viene eseguito mentre è in esecuzione. Questa capacità gli consente di effettuare ottimizzazioni più efficaci, come ad esempio sostituire le chiamate a una funzione con il corpo della funzione stessa.

## JIT vs AOT Compilation

### Processo di esecuzione

Il processo di esecuzione del JIT funziona all'interno dello stesso processo dell'applicazione e compete per le risorse (CPU, memoria) con l'applicazione stessa. AOT compila il codice prima dell'esecuzione, quindi non compete con l'applicazione per le risorse durante l'esecuzione.

JIT può sfruttare nuove opportunità di ottimizzazione basate sulle informazioni di esecuzione. Se un'ottimizzazione risulta inefficace durante l'esecuzione, il JIT può annullarla (deoptimizzare) e ripristinare un comportamento più semplice. Può fare speculazioni su come sarà eseguito il codice in futuro e ottimizzare di conseguenza, potendo correggere eventuali errori in seguito.

Il JIT compiler riceve bytecode come input e lo traduce in codice macchina eseguibile dalla CPU. Inoltre, la JVM verifica i file di classe al momento del caricamento, il che riduce la necessità di ulteriori operazioni di parsing o verifica al momento della compilazione.

### HotSpot's JIT execution model

Le assunzioni del HotSpot's JIT execution model sono le seguenti. La maggior parte del codice viene eseguita raramente, quindi compilarlo farebbe sprecare risorse di cui il JIT compiler ha bisogno. Solo un sottoinsieme di metodi viene eseguito frequentemente. L'interprete è subito pronto per eseguire qualsiasi codice.

Il codice compilato è molto più veloce, ma produrlo richiede molte risorse e è disponibile solo dopo che il processo di compilazione è terminato, il che richiede tempo.

Il modello di esecuzione di HotSpot funziona nel seguente modo. Il codice inizia a essere eseguito in modo interpretato senza ritardi. I metodi che vengono eseguiti comunemente (hot) vengono JIT compiled. Una volta che il compiled code è disponibile, l'esecuzione passa ad esso.

### Multi-tiered execution

Nella HotSpot, l'interprete mantiene un conteggio per metodo del numero di volte che un metodo viene chiamato. Poiché i metodi hot di solito contengono loop, raccoglie anche il numero di volte che viene preso un branch di ritorno all'inizio di un loop. All'ingresso del metodo, l'interprete somma i due numeri e, se il risultato supera una soglia, inserisce il metodo in coda per la compilazione.

Un thread del compilatore che viene eseguito in parallelo con i thread che eseguono il codice Java elabora la richiesta di compilazione. Mentre la compilazione è in corso, l'esecuzione interpretata continua, inclusi i metodi in fase di JIT compiling. Una volta che il codice compilato è disponibile, l'interprete passa ad esso.

Il compromesso è tra l'interprete che è veloce da avviare ma lento ad eseguire e il codice compilato che è lento da avviare ma veloce da eseguire. Quanto sia lento da avviare il codice compilato è sotto il controllo della virtual machine. Il compilatore può essere progettato per ottimizzare meno, in modo che il codice è disponibile prima ma non performa altrettanto bene, o di più, portando a codice più veloce che arriva in ritardo. Un design pratico che sfrutta questa osservazione è avere un sistema multi-tier.

## The three tiers of execution

HotSpot ha un sistema a tre livelli composto dall'interprete, dal quick compiler e dall'optimizing compiler.

Il codice Java inizia l'esecuzione nell'interprete. Poi, quando un metodo diventa warm, viene inserito in coda per la compilazione dal quick compiler. L'esecuzione passa al codice compilato appena pronto. Se un metodo in esecuzione nel secondo livello diventa hot, viene quindi inserito in coda per la compilazione dall'optimizing compiler. L'esecuzione continua nel compiled code di secondo livello fino a quando il codice più veloce non è disponibile. Appena disponibile, esegue nel terzo livello.

### Deoptimization and speculation

interpreter → Low tier compiler → Optimizing compiler

In realtà, esistono anche transizioni da codice più ottimizzato a codice meno ottimizzato.

interpreter → Low tier compiler → Optimizing compiler



Quando un thread deoptimizza, smette di eseguire un metodo compilato in un certo punto del metodo e riprende l'esecuzione nello stesso metodo Java esattamente nello stesso punto, ma nell'interprete.

A volte è conveniente non complicare eccessivamente il compilatore con il supporto per casi limite rari di alcune funzionalità. Piuttosto, quando viene incontrato quel particolare caso limite, il thread deoptimizza e passa all'interprete.

Il secondo motivo è che la deoptimization consente ai JIT compilers di speculare. Quando si specula, il compilatore fa assunzioni che dovrebbero rivelarsi corrette data l'attuale stato della virtual machine e che dovrebbero consentirgli di generare codice migliore. Tuttavia, il compilatore non può dimostrare che le sue assunzioni siano vere. Se un'assunzione viene invalidata, allora il thread che esegue un metodo che fa tale assunzione deoptimizza per non eseguire codice errato, basato su assunzioni sbagliate.

### Example 1: Null checks in the C2 tier

Un esempio di speculazione che C2 utilizza ampiamente è la gestione dei null checks. In Java, ogni accesso a un campo o a un array è protetto da un null check.

```
if (object == null) {
    throw new NullPointerException();
}
val = object.field;
```

È molto raro che una NullPointerException(NPE) non sia causata da un errore di programmazione, quindi C2 specula che le NPE non si verifichino mai. Ecco quella speculazione in pseudocodice:

```
if (object == null) {
    deoptimize();
}
val = object.field;
```

Se le NPE non si verificano mai, tutta la logica per la creazione, il lancio e la gestione delle eccezioni non è necessaria. Se un oggetto nullo viene visto durante l'accesso al campo nel pseudocodice, il thread deoptimizza, viene registrata la speculazione fallita e il codice del metodo compilato viene scartato.

## Example 2: Class hierarchy analysis

```
class C {  
    void virtualMethod() {}  
}  
  
void compiledMethod(C c) {  
    c.virtualMethod();  
}
```

La chiamata in `compiledMethod()` è una chiamata virtuale. Con solo la classe `C` caricata ma nessuna delle sue potenziali sottoclassi, quella chiamata può invocare solo `C.virtualMethod()`. Quando `compiledMethod()` viene JIT compiled, il compilatore potrebbe sfruttare questo fatto per devirtualizzare la chiamata. Se in un momento successivo viene caricata una sottoclasse della classe `C`, eseguire `compiledMethod()`, che è stato compilato assumendo che `C` non abbia sottoclassi, causa un'esecuzione errata.

La soluzione a quel problema è che il JIT compiler registri una dipendenza tra il metodo compilato di `compiledMethod()` e il fatto che `C` non ha sottoclassi. Il codice del metodo compilato stesso non incorpora alcun controllo extra a runtime ed è generato come se `C` non avesse sottoclassi. Quando viene caricata una classe, le dipendenze vengono verificate. Se viene trovato un metodo compilato con una dipendenza in conflitto, quel metodo viene deoptimizzato. Un nuovo metodo compilato può essere generato che terrà conto della gerarchia delle classi aggiornata.

### Safepoints and deoptimization

Durante la deoptimizzazione, la macchina virtuale deve essere in grado di ricostruire lo **stato di esecuzione** in modo che l'interprete possa **riprendere** il thread dal **punto** nel metodo in cui l'esecuzione compilata si è fermata. In un safepoint, esiste una **mappatura** tra gli **elementi dello stato dell'interprete** (variabili locali, monitor bloccati, ecc.) e la loro **posizione nel codice compilato**—come un registro, stack, ecc.

Nel caso di una deoptimizzazione sincrona (o trap non comune), un safepoint viene inserito nel punto della trap e cattura lo stato necessario per la deoptimizzazione. Nel caso di una deoptimizzazione asincrona, il thread nel codice compilato deve raggiungere uno dei safepoints che sono stati compilati nel codice per deoptimizzare.

Come parte del processo di ottimizzazione, i compilatori spesso riordinano le operazioni in modo che il codice risultante venga eseguito più rapidamente. Riordinare le operazioni attraverso un safepoint in quel modo causerebbe una differenza tra lo stato nel safepoint e lo stato atteso in quella posizione dall'interprete, e non può essere consentito.

Di conseguenza, non è fattibile avere un safepoint per ogni bytecode di un metodo. Fare ciò limiterebbe eccessivamente le ottimizzazioni. Un metodo compilato include solo pochi safepoints (al ritorno, alle chiamate e nei cicli) e c'è bisogno di un equilibrio tra safepoints abbastanza comuni, in modo che una deoptimizzazione non venga ritardata, e safepoints abbastanza rari, affinché il compilatore abbia la libertà di ottimizzare tra di essi.

Questo influenza anche sulla garbage collection e su altre operazioni della macchina virtuale che si basano sui safepoints. Infatti, le operazioni di garbage collection necessitano delle posizioni degli oggetti attivi sugli stack di un thread, che sono disponibili solo nei safepoints. In generale, nel codice compilato, i safepoints sono le uniche posizioni in cui è disponibile uno stato su cui la macchina virtuale può lavorare.

# AP-06: Software Components

## Definizione di componente sw

“Un **componente software** è un’unità di composizione con interfacce specificate contrattualmente e dipendenze di contesto esplicite. Un componente software può essere distribuito in modo indipendente ed è soggetto a **composizione da parte di terzi.**”

**Unità di Composizione:** I componenti sono progettati per essere modulari, indipendenti e facilmente integrabili con altri componenti, in modo da costruire componenti complessi.

**Interfacce Contrattuali:** Le modalità con cui un componente interagisce con altri sono chiaramente definite, specificando cosa offre e cosa richiede.

**Dipendenze Esplicite:** I componenti dichiarano chiaramente le loro dipendenze, come librerie o servizi necessari per funzionare.

**Distribuibili Indipendentemente:** Possono essere distribuiti e aggiornati senza influenzare altri componenti del sistema.

**Composizione da Parte di Terzi:** Altri sviluppatori o sistemi possono integrare questi componenti senza necessità di conoscere i dettagli interni.

Un componente è visto come una "scatola nera", il che significa che chi lo usa non deve preoccuparsi di come funziona internamente, ma solo di come interagire con esso attraverso le sue interfacce.

Il **glue code** è il codice che collega diversi componenti tra loro. Assicura che i componenti possano comunicare e lavorare insieme correttamente. Coordina le dipendenze tra i componenti, assicurando che siano soddisfatte le richieste di ciascun componente.

Un sistema software in questo contesto viene costruito come un insieme di componenti combinati per realizzare una funzionalità complessa.

I componenti sono spesso distribuiti come **unità binarie** (eseguibili o librerie compilate) che funzionano come "scatole nere", senza esporre il codice sorgente.

Non è possibile distribuire solo una **parte** del sistema; **tutte** le componenti necessarie devono essere **presenti** e correttamente integrate.

I componenti **non mantengono uno stato** che possa essere **osservato** o modificato dall'esterno, garantendo così l'incapsulamento.

Le **copie** dei componenti sono **identiche** e non possono essere differenziate l'una dall'altra.

**Un'Interfaccia** è la descrizione di come il componente può essere usato da altri software, che rappresenta ciò che il componente **"espone"** all'esterno.

**Contratto:** È un insieme di **regole** o vincoli che vengono associati **all'interfaccia**.

Queste regole vincolano sia chi utilizza il componente (client) che chi lo fornisce (provider), assicurando che entrambi rispettino le condizioni stabilite. Stabilisce un accordo su come il componente debba **comportarsi** e su come debba essere **utilizzato**.

Gli aspetti funzionali del contratto riguardano le operazioni che un componente mette a disposizione tramite la sua interfaccia, chiamate **API** (Application Programming Interface).

**Le Pre-condizioni** del contratto sono le condizioni che devono essere vere prima che una certa operazione possa essere eseguita.

**Le Post-condizioni** del contratto sono le condizioni che devono essere vere dopo che l'operazione è stata eseguita correttamente.

Gli **Aspetti Non Funzionali** riguardano vincoli aggiuntivi che non influenzano direttamente le operazioni, ma che sono comunque importanti per l'uso del componente. Ad esempio:

- **Requisiti di prestazioni:** Il componente potrebbe avere limiti di tempo, come la capacità di restituire un risultato entro 1 secondo.

→ **Requisiti di ambiente:** Il componente potrebbe richiedere un certo tipo di hardware o sistema operativo per funzionare correttamente.

## Contratto

Un **contratto** definisce come un componente può essere utilizzato e cosa esso offre. Per specificare le interfacce di un componente, serve un meccanismo formale, come l'**Interface Definition Language**, utilizzato per descrivere le interfacce in modo indipendente dall'implementazione, permettendo a diverse parti di un sistema di interagire senza conoscere i dettagli interni.

Un contratto fornisce informazioni su come il componente **deve essere distribuito** o installato. Definisce **come creare un'istanza** del componente. Descrive il **comportamento** previsto delle istanze quando interagiscono attraverso le interfacce pubblicizzate.

Il contratto va oltre la semplice definizione di **singole interfacce**. Non si limita a elencare cosa ogni interfaccia fa separatamente, ma stabilisce anche come **diverse interfacce collaborano** e come il componente deve funzionare nel suo insieme.

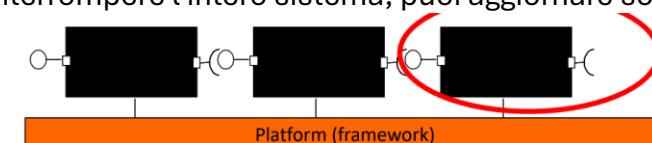
Per **dipendenza di contesto esplicita** si intende che un componente specifica **in modo chiaro** quali sono gli strumenti, le risorse, piattaforme o altri componenti di cui ha bisogno per funzionare correttamente.

**Le Interfacce Fornite (Provided Interface)** definiscono le funzionalità che il componente espone ad altri componenti o sistemi.

**Le Interfacce Richieste (Required Interface)** definiscono le risorse o i servizi di cui il componente ha bisogno per funzionare.



**Distribuito Indipendentemente** significa che un componente software può essere distribuito e installato separatamente dagli altri componenti del sistema. Non è necessario distribuire tutto il sistema per aggiornare o modificare un singolo componente. Se un componente ha bisogno di un aggiornamento, non è necessario interrompere l'intero sistema; puoi aggiornare solo quel componente.



- Late binding - dependencies are resolved at load or run-time.



Una **Piattaforma (framework)** è il contesto tecnologico (un insieme di strumenti, librerie e regole) su cui un componente si basa per essere eseguito (Spring/Java).

**Un Connettore** è il meccanismo che permette a un componente di comunicare o interagire con la piattaforma o altri componenti. Funziona come un "ponte" tra il componente e il resto del sistema. (Ad esempio un connettore può essere un protocollo o un'interfaccia che collega un componente a un database.)

Per **Late Binding** (collegamento tardivo) si intende il fatto che le dipendenze di un componente vengono **risolte al momento del caricamento o dell'esecuzione** del programma, piuttosto che durante la compilazione. Questo permette una maggiore flessibilità, poiché il sistema può decidere quale implementazione specifica utilizzare solo al momento dell'esecuzione.

**Il Meccanismo di composizione** indica come i vari componenti possono essere collegati tra loro per cooperare e svolgere un compito.

Un modello a componenti **non dipende** da un linguaggio di programmazione specifico. Componenti scritti in linguaggi diversi possono lavorare insieme su una piattaforma comune.

# **CBSE – Component-Based Software Engineering**

**Component-Based Software Engineering** è un approccio che fornisce metodi per creare sistemi complessi combinando componenti già pronti. Se un componente necessita di aggiornamenti o correzioni, è possibile sostituirlo senza influire sull'intero sistema.

L'architettura di un sistema CBSE è descritta in termini di componenti, ognuno con un ruolo specifico nel sistema e interazioni chiaramente definite con altri componenti.

## **Ciclo di vita di un component**

- **Component Specification:** La fase in cui viene definito cosa deve fare il componente, quali funzionalità offre e quali dipendenze ha.
- **Component Interface:** Fase in cui si definisce l'interfaccia, specificando quali metodi o funzioni possono essere chiamate dall'esterno.
- **Component Implementation:** È la fase in cui il componente viene effettivamente codificato secondo la specificazione.

Un **Installed Component** è una copia di un'**implementazione del componente** che è stata distribuita e registrata all'interno di un ambiente di esecuzione. La registrazione è un passaggio essenziale, poiché consente all'ambiente di riconoscere il **Componente Installato** al momento della creazione di un'istanza o durante l'esecuzione delle sue operazioni.

Un **Component Object** rappresenta l'**istanza in esecuzione** del componente nel sistema ed è un concetto di runtime. Ogni object ha i propri dati e una **identità unica**, consentendo di gestire stati e comportamenti in modo indipendente. Gli object sono le entità che eseguono le azioni definite dalle interfacce e possono interagire con altri oggetti all'interno del sistema.

Una **tecnologia di componenti** è una realizzazione concreta di un modello di componenti. Ad esempio, tecnologie come JavaBeans o .NET sono implementazioni pratiche di modelli di componenti.

## **Esempi di componenti**

I grandi componenti di successo comprendono architetture client-server, sistemi operativi avanzati e tecnologie modulari recenti. Nei sistemi client-server, i server di database, come quelli relazionali o orientati agli oggetti, si basano su API standard come SQL, mentre X-Windows, con la sua comunicazione basata su callback, offre grande adattabilità, pur essendo complesso da usare in contesti specifici. Tra i sistemi operativi, Unix e MS Windows rappresentano piattaforme avanzate: Unix si distingue per l'API POSIX e le sue interfacce a basso livello, mentre Windows presenta un'architettura distinta ma altrettanto complessa.

Le architetture a plugin, come Netscape Navigator e tecnologie server-side come ASP e JSP, dimostrano l'evoluzione verso componenti software più granulari. Tecnologie recenti come Visual Basic, Java Beans, Enterprise JavaBeans, Microsoft COM+ e app Android ampliano ulteriormente lo sviluppo modulare, fornendo strumenti scalabili e riutilizzabili. I server di applicazioni basati su J2EE e .NET/COM+ completano questo panorama.

Tutti questi esempi condividono caratteristiche chiave: un'infrastruttura di base solida, la possibilità di acquistare e integrare componenti da fornitori indipendenti, e servizi complessi che rendono inutile svilupparli da zero. I componenti sono progettati per coesistere senza conflitti e offrono un valore diretto agli utenti, grazie a un livello di astrazione chiaro e funzionalità facilmente comprensibili e personalizzabili.

## **AP-07: JavaBeans**

### **JavaBeans**

La **JavaBeans API** è stata creata con l'obiettivo di definire un **modello di componenti software per Java**. Questo permette ai fornitori di creare e distribuire componenti Java che possono essere combinati dagli utenti finali per creare applicazioni.

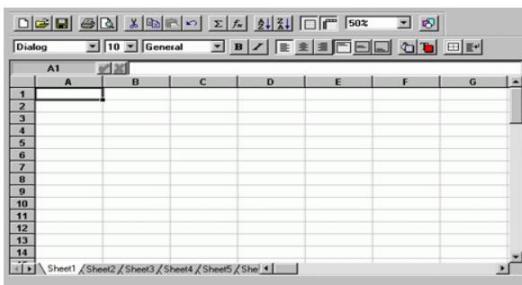
**Def.** I **Java Beans** sono componenti software riutilizzabili che possono essere manipolati **visivamente** tramite strumenti di sviluppo (detti "builder tools"). Questi

strumenti possono essere utilizzati per creare pagine web, applicazioni visive, layout di interfacce grafiche (GUI) o applicazioni server. Anche editor di documenti possono gestire Java Beans.

I JavaBeans possono rappresentare elementi dell'interfaccia grafica, come pulsanti, caselle di testo o altre interazioni utente.



An application constructed from Beans



I JavaBeans sono progettati per essere componenti **riutilizzabili** in diverse applicazioni. Possono essere manipolati visivamente, cioè possono essere configurati e collegati ad altri componenti usando strumenti di sviluppo **grafico**. Tipicamente, un Java Bean ha una rappresentazione grafica (es. pulsanti o slider), ma non sempre è necessario. Esistono anche **bean invisibili**, cioè componenti che non hanno un'interfaccia visiva, ma che svolgono funzioni logiche.

## Requisiti necessari di un Java Bean

Un **JavaBean** deve avere un **costruttore** pubblico che **non richiede argomenti**. Deve **implementare** l'interfaccia **java.io.Serializable**, il che consente di salvare e ripristinare lo stato dell'oggetto. Deve essere incluso in un file **JAR** (archivio Java) e contenere un file manifest che dichiari Java-Bean: True per indicare che è un Java Bean. Devono inoltre fornire **supporto per l'introspezione**, in modo che uno strumento di sviluppo possa analizzare come funziona un bean, **supporto per la personalizzazione**, in modo che un utente possa personalizzare l'aspetto e il comportamento di un bean, **supporto per gli eventi** come semplice meccanismo di comunicazione per collegare tra loro i bean, **supporto per le proprietà**, sia per la personalizzazione che per l'uso programmatico e **supporto per la persistenza**, in modo che un bean possa avere il suo stato personalizzato salvato e ricaricato in seguito.

I bean visibili devono ereditare da **java.awt.Component** in modo da poter essere aggiunti a contenitori visivi, ma i bean invisibili non devono ereditare da nessuna classe o interfaccia.

## Concetto di JavaBeans come componenti software modulari

I JavaBeans sono **componenti binari** (file .class), cioè moduli riutilizzabili che possono essere assemblati per costruire applicazioni o altri JavaBeans.

Durante lo sviluppo, i JavaBeans possono essere personalizzati e configurati. Dopo la personalizzazione, vengono distribuiti come componenti pronti all'uso.

I JavaBeans possono essere combinati insieme per creare **nuovi componenti** o **applicazioni complete**, scrivendo del codice "colla" per connettere i vari JavaBeans.

I **Beans lato client** gestiscono l'interfaccia utente. I **Beans lato server** sono usati per la logica di business (non hanno una rappresentazione visiva).

## Design time vs. Run-time:

Un **bean** deve essere in grado di funzionare all'interno di uno strumento di sviluppo chiamato "**ambiente di design**" visivo, in cui il bean deve fornire informazioni di design e permettere all'utente di personalizzarne l'aspetto e il comportamento.

Le informazioni di design, come un "wizard" di personalizzazione che guida l'utente, possono essere molto estese e a volte più grandi del codice di esecuzione del bean stesso.

Per questo motivo, è importante separare le funzionalità di **design time** (che riguardano la personalizzazione) da quelle di **run-time** (che riguardano l'esecuzione vera e propria del bean). In questo modo, quando un'applicazione viene eseguita, il bean non deve caricare tutto il suo codice di design, ma solo ciò che serve per il run-time, ottimizzando le risorse.

Questa separazione può essere gestita tramite una classe chiamata **<BeanName>BeanInfo.java**, che contiene tutte le informazioni necessarie per la personalizzazione del bean durante il design time.

Se la classe BeanInfo non è presente, le informazioni di personalizzazione vengono estratte direttamente dal codice del bean stesso.

## Proprietà semplici

Le **proprietà semplici** sono **attributi modificabili** dei JavaBeans che determinano **l'aspetto o il comportamento di un componente**, e possono essere gestite sia durante la progettazione che durante l'esecuzione dell'applicazione.

Questi attributi possono essere, ad esempio, il colore di sfondo, la dimensione.

Una proprietà viene identificata da metodi pubblici chiamati **getter** e **setter**.

Le proprietà possono essere cambiate in due momenti: **Design time**, ovvero durante la fase di progettazione, per personalizzare l'aspetto o il comportamento del bean oppure durante **l'esecuzione dell'applicazione**, in base alla logica del programma.

*Esempio.* Consideriamo un pulsante con la proprietà **background** (colore di sfondo), identificata dai suoi metodi getter e setter.

```
public java.awt.Color getBackground();  
public void setBackground(java.awt.Color color);
```



Grazie all'introspezione, un builder tool può mostrare informazioni sul bean agli sviluppatori, permettendo loro di personalizzare il comportamento o l'aspetto del componente.

L'introspezione può avvenire in modo implicito, utilizzando **reflection**. Se un bean ha metodi `getNome()` e `setNome()`, il tool capisce che "Nome" è una proprietà che può essere letta e modificata.

In alternativa, si può creare una classe chiamata **<BeanName>BeanInfo** che fornisce in modo esplicito tutte le informazioni sul bean, come proprietà, eventi e metodi, al tool di sviluppo.

Un Java Bean può specificare esplicitamente quali proprietà, eventi e metodi supporta fornendo una classe che implementa l'interfaccia **BeanInfo**. Questo permette agli strumenti di sviluppo di sapere esattamente quali funzionalità il bean offre. Lo sviluppatore può scegliere di rendere visibili solo alcune funzionalità del Bean, nascondendo quelle non necessarie. Può utilizzare BeanInfo per gestire alcune caratteristiche del Bean e affidarsi alla reflection per esporne altre in modo più diretto e dinamico. È possibile associare un'icona rappresentativa al Bean, utile ad esempio negli ambienti di sviluppo visivo per identificare il Bean graficamente. Può definire una classe personalizzata per configurare il Bean in modo interattivo, offrendo una migliore esperienza utente per la configurazione di proprietà complesse. Le funzionalità del

Bean possono essere classificate in categorie, come "normali" o "esperte", per rendere più intuitivo l'uso del Bean a seconda del livello di competenza dell'utente. Puo fornire un nome descrittivo più chiaro o aggiungere ulteriori informazioni alle caratteristiche del Bean, migliorandone la leggibilità e la comprensione negli strumenti di sviluppo.

### Proprietà, eventi e metodi

Un Java Bean possiede tre caratteristiche fondamentali: **proprietà, metodi e eventi**.

Le **proprietà** di un Java Bean sono **attributi nominati** che definiscono **lo stato del bean**. Questi attributi possono essere letti o modificati esternamente tramite metodi specifici **getter e setter**, tramite i quali viene derivato il nome e il tipo dell'attributo. Se è presente solo il metodo **getter**, allora la proprietà è **di sola lettura**. Se è presente solo il metodo **setter**, la proprietà è **di sola scrittura**.

I **metodi** di un Java Bean sono **funzioni** Java standard che possono essere chiamate da altri componenti esterni, e definiscono **il comportamento del bean** permettendo di eseguire operazioni specifiche.

Gli **eventi** permettono a un bean di **comunicare** con altri componenti **quando succede qualcosa di rilevante**.

Ad esempio, un bean potrebbe generare un evento quando cambia il valore di una sua proprietà, o quando si verifica un'azione particolare. Questi eventi sono gestiti attraverso listener, che sono oggetti specializzati che "ascoltano" gli eventi emessi dal bean. Quando un evento si verifica, il bean notifica tutti i listener registrati chiamando metodi specifici su di essi, permettendo così agli altri componenti di reagire all'evento.

### Proprietà Indicizzate

Se una proprietà è un **array**, i metodi **getter** e **setter** possono accettare un **indice** o **l'intero array**. Ad esempio per la proprietà **Color[] Spectrum**:

**public java.awt.Color getSpectrum(int index):** ottiene il colore alla posizione specificata dall'indice.

**public java.awt.Color[] getSpectrum():** ottiene l'intero array di colori.

**public void setSpectrum(int index, java.awt.Color color):** imposta un colore specifico all'indice dato.

**public void setSpectrum(java.awt.Color[] colors):** imposta l'intero array di colori.

Utilizzando questi metodi, lo strumento di sviluppo tramite **introspezione** deduce l'esistenza della proprietà **spectrum**, che è di tipo **java.awt.Color[]**.

### Proprietà Bound e Constrained:

Una **proprietà bound (legata)** genera un **evento** quando il suo valore viene **modificato**.

Gli oggetti che "ascoltano" (ovvero gli **osservatori**) possono ricevere una **notifica** ogni volta che il valore della proprietà cambia.

Esempio: Se un componente cambia il colore di sfondo, tutti gli ascoltatori registrati verranno avvisati di questa modifica.

Una **proprietà constrained (vincolata)** può cambiare il suo valore solo se **nessuno degli osservatori registrati** pone un **veto**.

Prima di accettare il cambiamento, gli osservatori hanno l'opportunità di **bloccare** il cambiamento, se non è consentito o appropriato.

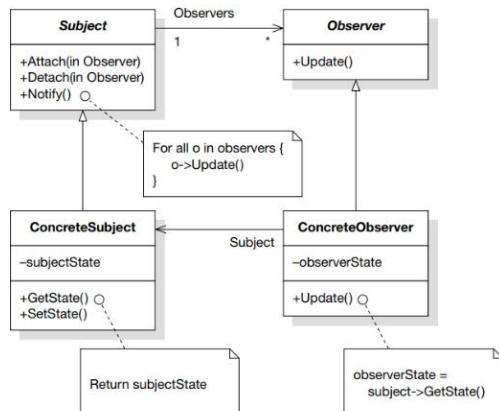
Esempio: Se si tenta di impostare un valore non valido per una proprietà, un osservatore potrebbe porre il voto e impedire che il valore venga modificato.

**La Connection-oriented programming** è un paradigma usato per collegare insieme i vari componenti tra loro all'interno di un'applicazione in un **tool di sviluppo** (builder tool), come interfacce grafiche (GUI). Per esempio, puoi collegare un pulsante a un campo di testo, in modo che quando premi il pulsante, il testo cambia.

**Loose coupling (accoppiamento debole):** I componenti sono **debolmente accoppiati**, il che significa che possono comunicare tra loro senza dipendere strettamente l'uno dall'altro.

## Design pattern observer

Il **pattern Observer** è un **design pattern** comportamentale che consente di definire una relazione uno-a-molti tra oggetti, in modo tale che quando un oggetto **subject** cambia stato, tutti gli oggetti dipendenti **observer** vengano notificati automaticamente e aggiornati.



**Un subject** è l'oggetto che cambia stato e che deve notificare agli altri oggetti.

Mantiene come attributo una **lista di osservatori** da notificare.

I metodi principali che la classe astratta espone sono:

**Attach(Observer)**: Aggiunge un osservatore alla lista.

**Detach(Observer)**: Rimuove un osservatore dalla lista.

**Notify()**: Notifica tutti gli osservatori registrati di un cambiamento, chiamando per ogni elemento della lista il loro metodo **Update()**.

**ConcreteSubject** è la classe che implementa il **Subject**. Mantiene come attributo lo stato concreto che interessa agli **Observer**. I metodi che espone sono:

**GetState()**: Restituisce lo stato del soggetto.

**SetState()**: Modifica lo stato del soggetto e chiama **Notify()** per informare gli osservatori del cambiamento.

Gli **observers** sono gli oggetti che vogliono essere informati quando lo stato del soggetto cambia.

Quando il **soggetto** cambia stato, invia una notifica a tutti gli **osservatori** registrati, che si aggiornano di conseguenza. Il metodo principale che la classe astratta espone è:

**Update()**: Viene chiamato dal **Subject** per aggiornare l'osservatore con le nuove informazioni.

**ConcreteObserver** è la classe che implementa l'**Observer**. Memorizza come attributo (il soggetto e) lo stato che viene sincronizzato con quello del soggetto. Il metodo che espone è:

**Update()**: Viene chiamato quando il soggetto cambia stato, e aggiorna lo stato dell'osservatore (con **observerState = subject->GetState()**).

## Flusso delle operazioni:

Il **ConcreteSubject** cambia stato tramite il metodo **SetState()**. Dopo aver cambiato lo stato, chiama il metodo **Notify()**, che consiste nel chiamare il metodo **Update()** su tutti gli osservatori presenti nella lista. Ogni **ConcreteObserver** riceve la notifica tramite il metodo **Update()** e aggiorna il proprio stato (**observerState**) con il nuovo stato del **Subject** (**subject->GetState()**).

## Pattern Observer in Java

In Java, il pattern Observer è utilizzato attraverso un sistema di **eventi** e **listener**.

Gli eventi rappresentano **cambiamenti o azioni** che si verificano in un oggetto e che altri oggetti possono ascoltare.

Concettualmente, gli eventi sono un meccanismo per **propagare notifiche** di cambiamento di stato tra un oggetto sorgente e uno o più oggetti destinatari ascoltatori.

Un **evento** è un oggetto creato da una **event source** (ad esempio, un pulsante che viene premuto). Questo evento viene poi **propagato** agli **ascoltatori di eventi** (event listeners) registrati, che sono interessati a sapere quando accade qualcosa in quella sorgente.

Le **event source** devono fornire metodi per **registrare e deregistrare** i loro event listener, permettendo così un flusso di eventi dalla sorgente agli ascoltatori (i listener). Questo viene fatto usando due metodi:

**public void add<ListenerType>(<ListenerType> listener):** Aggiunge l'ascoltatore alla lista di ascoltatori registrati per quel tipo di evento.

**public void remove<ListenerType>(<ListenerType> listener):** Rimuove l'ascoltatore dalla lista.

La presenza di questi metodi identifica la classe come una **sorgente di eventi multicast**, ovvero più ascoltatori possono essere registrati per lo stesso tipo di evento. Anche il nome dell'evento viene estratto dalla signature.

Example: from

```
public void addUserSleepsListener (UserSleepsListener l);  
public void removeUserSleepsListener (UserSleepsListener l);
```

infers that the class generates a **UserSleeps** event

Quando un evento viene attivato, la sorgente chiama ogni **ascoltatore registrato** che è idoneo a ricevere quell'evento.

In generale, **tutti** gli ascoltatori registrati vengono considerati idonei per la notifica, ma la sorgente può decidere di limitare a un **sottoinsieme** di ascoltatori, in base a criteri specifici (come lo stato del sistema o la natura dell'evento).

Un evento ha **semantica unicast** se la sua sorgente permette la registrazione di **un solo ascoltatore**.

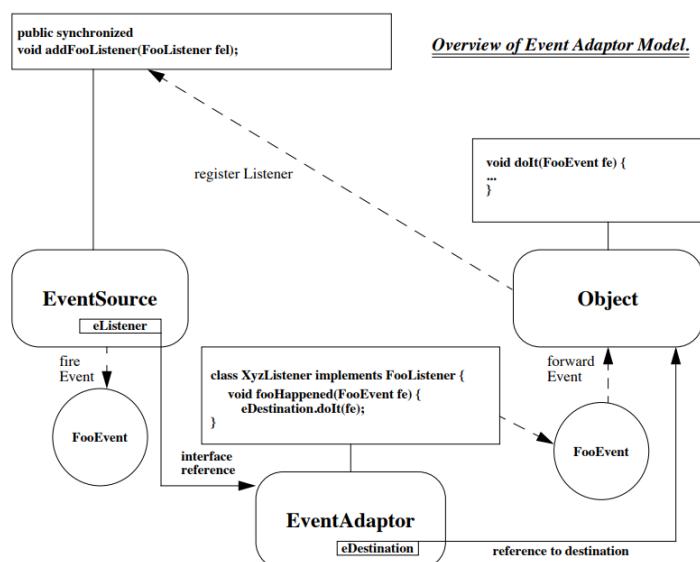
Questo viene indicato dal fatto che il metodo **add<ListenerType>** (per aggiungere un ascoltatore) può lanciare l'eccezione **TooManyListenersException**. Ciò avviene se si tenta di aggiungere più di un ascoltatore a una sorgente di eventi che supporta solo un listener.

- Example:

```
public void addJackListener(JackListener t)  
throws java.util.TooManyListenersException;  
  
public void removeJackListener(JackListener t);
```

## Event adaptor Architecture

Gli **Event Adaptors** sono oggetti che vengono posti tra una **sorgente di eventi** e un **ascoltatore** (listener), con lo scopo di facilitare la gestione degli eventi. Un event adaptor riceve gli eventi dalla sorgente, **li gestisce e poi li passa** agli ascoltatori finali o esegue altre operazioni su di essi.



**La EventSource** è la sorgente che genera l'evento. Quando si verifica un evento, viene creato un oggetto evento chiamato **FooEvent**, e la sorgente chiama il metodo appropriato per avvisare gli ascoltatori registrati. Espone il metodo addFooListener(FooListener fel) per registrare gli ascoltatori interessati a ricevere gli eventi.

**L'EventAdaptor** funge da intermediario tra la sorgente dell'evento e l'oggetto destinatario. La classe XyzListener implementa l'interfaccia FooListener. Quando un **FooEvent** viene generato, il metodo fooHappened(FooEvent fe) viene chiamato dall'EventSource, e l'EventAdaptor inoltra l'evento all'oggetto destinatario invocando il metodo doIt(FooEvent fe). L'EventAdaptor mantiene un riferimento a eDestination, ovvero l'oggetto destinatario.

**Object** rappresenta l'oggetto finale che deve elaborare l'evento. Quando l'EventAdaptor riceve l'evento dalla sorgente, lo inoltra all'oggetto chiamando il metodo doIt(FooEvent fe), che gestisce l'evento.

**Object** non è un listener, ma un destinatario che elabora l'evento dopo che il listener (EventAdaptor) lo ha inoltrato. Riceve l'evento indirettamente, tramite l'EventAdaptor, che lo inoltra invocando il metodo doIt(FooEvent fe).

La sorgente di eventi genera un **FooEvent** e lo invia all'EventAdaptor tramite il metodo **fooHappened(FooEvent fe)**. L'EventAdaptor riceve l'evento e lo inoltra all'oggetto destinazione chiamando il metodo **doIt(FooEvent fe)**. L'oggetto destinatario elabora l'evento ricevuto.

#### Esempi di usi comuni degli adaptors:

**Meccanismo di accodamento degli eventi:** Un adaptor può essere utilizzato per implementare una **coda di eventi** tra la sorgente e l'ascoltatore, gestendo gli eventi in modo sequenziale, anche se la sorgente li genera rapidamente.

**Filtro:** Un adaptor può agire come **filtro**, decidendo quali eventi far passare agli ascoltatori e quali ignorare in base a criteri specifici.

**Demultiplexing:** Un adaptor può gestire eventi da **sorgenti multiple** e reindirizzarli a un singolo ascoltatore. Questo è utile quando hai più sorgenti di eventi che devono essere gestite da un unico listener.

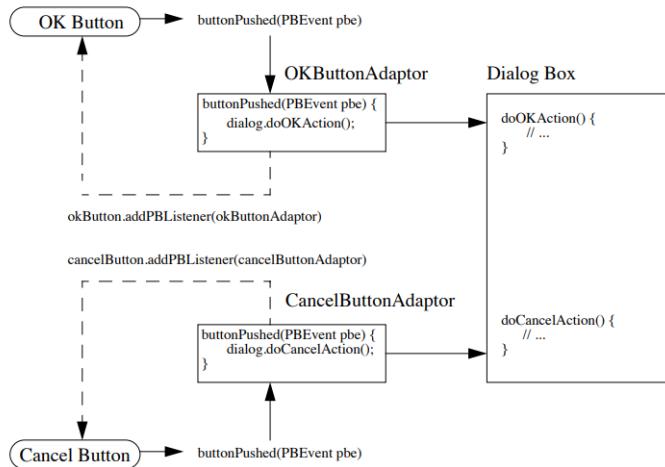
**Wiring manager:** Gli adaptors possono funzionare come un **gestore di collegamenti** tra le sorgenti di eventi e gli ascoltatori, semplificando la connessione e la gestione dei flussi di eventi.

#### Esempio del demultiplexing

Un dato oggetto event listener può implementare una data interfaccia EventListener **solo** una volta. Quindi, se un ascoltatore si registra con **molteplici sorgenti** di eventi per lo **stesso** evento, l'ascoltatore deve determinare da solo **quale** sorgente ha effettivamente **emesso** un particolare evento.

Per rendere la consegna degli eventi più comoda, ci piacerebbe permettere che gli eventi di diverse sorgenti di eventi vengano consegnati a diversi **metodi target** sull'oggetto ascoltatore finale. Quindi, per esempio, se ho due pulsanti "OKButton" e "CancelButton", mi piacerebbe essere in grado di organizzare affinché una pressione sul pulsante "OKButton" risulti in una chiamata su un metodo, mentre una pressione sul pulsante "CancelButton" risulti in una chiamata su un altro metodo.

# Event adaptors example: Demultiplexing multiple event sources



L'immagine mostra un esempio di **Event Adaptor** utilizzato per **demultiplexare** eventi provenienti da più sorgenti, in questo caso da due bottoni: un **OK Button** e un **Cancel Button**.

Quando viene premuto l'**OK Button**, questo genera un evento chiamato **buttonPushed(PBEvent pbe)**. Questo evento viene intercettato dall'**OKButtonAdaptor**, che ha il metodo **buttonPushed(PBEvent pbe)**. L'Event Adaptor inoltra l'evento alla finestra **Dialog Box** chiamando il suo metodo **doOKAction()** che esegue l'azione desiderata.

Allo stesso modo, quando viene premuto il **Cancel Button**, genera lo stesso tipo di evento **buttonPushed(PBEvent pbe)**. Questo evento viene catturato dal **CancelButtonAdaptor**, il quale inoltra l'evento alla **Dialog Box** chiamando il suo metodo **doCancelAction()**, che esegue l'azione desiderata.

Questo approccio permette di **demultiplexare** eventi provenienti da **più sorgenti** (OK Button e Cancel Button), gestendoli separatamente ma inviandoli alla stessa destinazione, ovvero la **Dialog Box**. A seconda di quale bottone è stato premuto, l'azione appropriata viene eseguita.

## Back to bound properties

Le **Bound Properties** (proprietà legate) sono proprietà in un JavaBean che, quando cambiano, generano un **evento** per notificare il cambiamento agli ascoltatori registrati. L'evento generato è di tipo **PropertyChangeEvent**. Questo evento contiene informazioni sul cambiamento della proprietà, come il vecchio e il nuovo valore. Gli oggetti che desiderano essere notificati di questo cambiamento devono prima **registrarsi** come ascoltatori. Una volta registrati, riceveranno una notifica ogni volta che la proprietà cambia.

Il **bean con la proprietà bound** è l'**event source**. È responsabile di generare l'evento quando la proprietà cambia.

Il **bean che implementa il listener** è l'**event target**. Riceve l'evento e può reagire al cambiamento della proprietà.

Nelle API Java esistono **classi di supporto** che aiutano a gestire l'invio e la ricezione delle notifiche in modo più efficiente.

## Passaggi per implementare una Bound Property:

Prima di tutto, è necessario importare il package **java.beans** per poter utilizzare le classi relative alla gestione degli eventi delle properties.

```
import java.beans.PropertyChangeSupport, java.beans.PropertyChangeListener;
```

Si istanzia un oggetto della classe **PropertyChangeSupport**, che aiuta a gestire gli ascoltatori e a inviare notifiche quando la proprietà cambia.

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

È necessario implementare i metodi per **aggiungere** e **rimuovere** gli ascoltatori che vogliono essere notificati quando la proprietà cambia.

Il metodo **addPropertyChangeListener()** aggiunge un ascoltatore alla lista:

```
public void addPropertyChangeListener(PropertyChangeListener l) {  
    changes.addPropertyChangeListener(l);  
}
```

Allo stesso modo, è necessario implementare il metodo

**removePropertyChangeListener()** per permettere di rimuovere un ascoltatore dalla lista:

```
public void removePropertyChangeListener(PropertyChangeListener l) {  
    changes.removePropertyChangeListener(l);  
}
```

Bisogna modificare il **setter** della proprietà in modo che, ogni volta che il valore cambia, venga generato un evento di cambiamento di proprietà. Per fare ciò, il metodo setter deve chiamare **firePropertyChange()** sulla variabile changes creata in precedenza.

```
public void setX(int newX) {  
    int oldX = this.x;  
    this.x = newX;  
    changes.firePropertyChange("x", oldX, newX);  
}
```

Questo metodo **firePropertyChange()** notifica tutti gli ascoltatori registrati del cambiamento della proprietà "x", passando il vecchio valore (oldX) e il nuovo valore (newX).

Per implementare un **Bound Property Listener** in Java, devi seguire alcuni passaggi chiave:

Il bean che agirà da ascoltatore deve implementare l'interfaccia

**PropertyChangeListener**.

Questa interfaccia richiede l'implementazione del metodo **propertyChange()**.

```
public class MyLstnr implements PropertyChangeListener, Serializable {  
    @Override  
    public void propertyChange(PropertyChangeEvent evt) {  
        // Logica per gestire il cambiamento della proprietà  
    }  
}
```

Il metodo **propertyChange(PropertyChangeEvent evt)** viene eseguito quando la proprietà legata cambia. Questo metodo riceve un oggetto di tipo **PropertyChangeEvent** che contiene informazioni sul cambiamento, come il nome della proprietà, il vecchio valore e il nuovo valore.

```
@Override  
public void propertyChange(PropertyChangeEvent evt) {  
    System.out.println("Proprietà " + evt.getPropertyName() + " cambiata.");  
    System.out.println("Vecchio valore: " + evt.getOldValue());  
    System.out.println("Nuovo valore: " + evt.getNewValue());  
}
```

Una volta implementato il listener, devi registrarlo col un **bean** che supporta le **Bound Properties**.

```
Button button = new OurButton(); // Bean che genera l'evento  
MyLstnr lis = new MyLstnr(); // Il listener  
button.addPropertyChangeListener(lis); // Registrazione del listener
```

## Constrained Property:

Una **Constrained Property** può cambiare il suo valore solo se **nessuno degli osservatori registrati** pone un **veto**.

Prima di accettare il cambiamento, gli osservatori hanno l'opportunità di **bloccare** il cambiamento, se non è consentito o appropriato.

A differenza delle **Bound Properties**, una **Constrained Property** genera un evento **prima** che il cambiamento avvenga, ogni volta che si cerca di modificare il valore di una proprietà.

L'evento generato è di tipo **PropertyChangeEvent**, che contiene informazioni come il vecchio valore e il nuovo valore proposto della proprietà. Quando si tenta di cambiare una proprietà, viene creato un oggetto **PropertyChangeEvent**.

Gli oggetti che si sono precedentemente registrati per ricevere notifiche (listener) vengono informati quando si tenta di cambiare il valore della proprietà.

Gli oggetti ascoltatori hanno la possibilità di **vietare** il cambiamento della proprietà. Questo significa che possono sollevare un'**eccezione** (spesso chiamata **PropertyVetoException**) che impedisce la modifica del valore.

Grazie a questo meccanismo, un bean può operare in modo diverso a seconda delle condizioni di runtime. Ad esempio, se in determinate condizioni un cambiamento di valore non è appropriato, gli oggetti ascoltatori possono intervenire per impedirlo.

Il **source bean** è l'oggetto che contiene una o più proprietà vincolate (**Constrained Properties**). Quando si tenta di modificare il valore di una proprietà vincolata, il bean genera un evento e lo invia agli ascoltatori registrati. Il bean deve gestire le richieste di cambio di proprietà e, in caso di veto da parte di un ascoltatore, annullare il cambiamento.

I **listener** sono gli osservatori che "ascoltano" le modifiche alle proprietà vincolate.

Devono implementare l'interfaccia **VetoableChangeListener**.

Quando un tentativo di modifica viene fatto, il metodo **vetoableChange()** viene chiamato. Se l'ascoltatore ritiene che il cambiamento sia inaccettabile, solleva una **PropertyVetoException**, bloccando il cambiamento.

```
public class MyVetoListener implements VetoableChangeListener {  
    @Override  
    public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {  
        // Logica per accettare o rifiutare la modifica  
        if ((int) evt.getNewValue() < 0) {  
            throw new PropertyVetoException("Il valore non può essere negativo", evt);  
        }  
    }  
}
```

L'oggetto **PropertyChangeEvent** contiene:

**Nome della proprietà:** Il nome della proprietà che si sta tentando di modificare.

**Valore vecchio:** Il valore attuale della proprietà.

**Valore nuovo:** Il valore proposto che si vuole assegnare alla proprietà.

Questo oggetto viene passato agli ascoltatori registrati, che decidono se accettare o rifiutare il cambiamento.

## Implementazione

Prima di tutto, il bean deve importare il pacchetto **java.beans** per utilizzare le classi che gestiscono gli eventi di proprietà vincolate.

```
import java.beans.VetoableChangeSupport;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyVetoException;
import java.beans.VetoableChangeListener;
```

Si istanzia un oggetto **VetoableChangeSupport**, che gestirà la registrazione, la rimozione e la notifica degli ascoltatori che vogliono "vietare" i cambiamenti.

```
private VetoableChangeSupport vetos = new VetoableChangeSupport(this);
```

Il bean deve fornire metodi per aggiungere e rimuovere ascoltatori che implementano l'interfaccia **VetoableChangeListener**.

Il metodo **addVetoableChangeListener()** viene usato per registrare un ascoltatore.

```
public void addVetoableChangeListener(VetoableChangeListener l) {
    vetos.addVetoableChangeListener(l);
}
```

Allo stesso modo, è necessario implementare il metodo

**removeVetoableChangeListener()** per rimuovere un ascoltatore dalla lista:

```
public void removeVetoableChangeListener(VetoableChangeListener l) {
    vetos.removeVetoableChangeListener(l);
}
```

Il **setter** della proprietà vincolata deve generare un evento **PropertyChangeEvent** e consentire agli ascoltatori di vietare il cambiamento.

Prima di cambiare il valore della proprietà, viene invocato il metodo

**fireVetoableChange()**, che avvisa tutti gli ascoltatori e dà loro l'opportunità di sollevare una **PropertyVetoException** per bloccare la modifica.

```
public void setX(int newX) {
    int oldX = X;
    try {
        vetos.fireVetoableChange("X", oldX, newX);
        // Se nessun veto è sollevato, aggiorna la proprietà
        X = newX;
        // Aggiungi codice qui per notificare altri cambiamenti, se necessario
    } catch (PropertyVetoException e) {
        // Codice da eseguire se il cambiamento è rifiutato
        System.out.println("Il cambiamento di X è stato rifiutato: " + e.getMessage());
    }
}
```

Se un ascoltatore solleva una **PropertyVetoException**, il cambiamento viene impedito.

Il metodo **fireVetoableChange()** dell'oggetto **VetoableChangeSupport** notifica tutti i **VetoableChangeListener**s registrati del cambiamento proposto. Viene passato il nome della proprietà, il valore vecchio e il nuovo valore proposto. Se uno degli ascoltatori rifiuta il cambiamento, solleverà una **PropertyVetoException**, impedendo l'aggiornamento della proprietà.

Se uno degli ascoltatori rifiuta il cambiamento (solleva una **PropertyVetoException**), il blocco **catch** cattura l'eccezione. Qui puoi gestire il rifiuto del cambiamento, ad esempio mostrando un messaggio di errore o registrando l'evento.

### Implementa un listener **VetoableChangeListener**

L'interfaccia **VetoableChangeListener** ha un metodo astratto chiamato **vetoableChange()**. Ogni ascoltatore che vuole ricevere notifiche sui cambiamenti di

proprietà vincolate deve implementare questa interfaccia. Il metodo astratto da implementare è:

```
public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException;
```

Il metodo **vetoableChange()** verrà chiamato ogni volta che il bean sorgente tenta di modificare una proprietà vincolata.

```
@Override
```

```
public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {
```

```
    String propertyName = evt.getPropertyName();
```

```
    int newValue = (int) evt.getNewValue();
```

```
    if (propertyName.equals("X") && newValue < 0) {
```

```
        throw new PropertyVetoException("Il valore non può essere negativo", evt);
```

```
}
```

```
}
```

Se l'ascoltatore decide che il cambiamento non deve avvenire, deve sollevare una **PropertyVetoException**. In questo modo, il cambiamento della proprietà sarà annullato. Se il cambiamento è accettabile, basta **restituire il controllo** senza fare nulla.

## Lab NetBeans

### Creazione di un Progetto JavaBeans con NetBeans

Per comprendere e utilizzare al meglio i JavaBeans, segui questa guida passo dopo passo utilizzando NetBeans:

1. **Configurazione di NetBeans:** Scarica e installa NetBeans 7.0 o una versione successiva. Questo strumento semplifica l'utilizzo dei JavaBeans grazie alla sua funzionalità di "bean builder".

2. **Creazione di un Nuovo Progetto:**

- Apri NetBeans e vai su File > New Project.
- Seleziona Java sotto Categories e Java Application sotto Projects. Clicca su Next.
- Assegna un nome al progetto (ad esempio, SnapApp) e deselectiona l'opzione Create Main Class. Clicca su Finish.

3. **Creazione di un JFrame:**

- Nella finestra Projects, fai clic con il tasto destro sul progetto SnapApp e scegli New > JFrame Form.
- Dai un nome alla classe (es. SnapFrame) e specifica il package (es. snapapp). Clicca su Finish.

4. **Progettazione Visiva:**

- NetBeans aprirà il designer visivo, dove puoi creare l'interfaccia grafica del progetto. La palette sulla destra contiene componenti che puoi trascinare sul JFrame.
- Per esempio, trascina un pulsante (Button) dalla palette sullo JFrame. Il pulsante stesso è un JavaBean.

5. **Proprietà del Pulsante:**

- Seleziona il pulsante aggiunto. Nella finestra Properties, puoi modificare aspetti come il colore del testo (Foreground), il font o il testo del pulsante (Text). Per alcune proprietà, come il colore, clicca sul pulsante ... accanto alla proprietà per scegliere un valore.
- Nota che queste modifiche vengono fatte senza scrivere codice.

6. **Gestione degli Eventi:**

- I JavaBeans possono generare eventi. Vai alla scheda Events nella finestra delle proprietà del pulsante. Qui troverai una lista di eventi, come actionPerformed.
- Ad esempio, puoi connettere un'etichetta (Label) al pulsante in modo che il testo cambi quando il pulsante viene cliccato. Per farlo:

- Abilita la modalità di connessione cliccando sull'icona Connection Mode nella toolbar del designer.
- Clicca sul pulsante, quindi sull'etichetta. Il *Connection Wizard* ti guiderà nella selezione dell'evento (es. actionPerformed) e dell'azione (es. cambiare il testo).

## 7. Esecuzione del Progetto:

- Clicca su Run Main Project o premi F6 per avviare il progetto. Quando si apre la finestra dell'applicazione, cliccando sul pulsante vedrai l'effetto delle configurazioni fatte.

## 8. Aggiunta di JavaBeans di Terze Parti:

- Scarica un componente JavaBean (es. BumperSticker) in formato JAR.
- Vai su Tools > Palette > Swing/AWT Components, clicca su Add from JAR..., seleziona il file JAR e aggiungi i componenti alla sezione desiderata della palette (es. Beans).
- Trascina il bean appena aggiunto sul JFrame e configura le sue proprietà o eventi come fatto in precedenza.

# Reflection

La **Reflection** è la capacità di un programma di ispezionare se stesso e, in alcuni casi, modificarsi dinamicamente.

La reflection può fornire semplici informazioni, ad esempio sui tipi di dati presenti, oppure può riflettere l'intera struttura del programma, inclusi i metodi e le variabili. Un'altra caratteristica importante è se la reflection consente solo di **leggere** lo stato del programma o anche di **modificarlo**.

**Introspezione:** la capacità di un programma di **osservare** il proprio stato, esaminando la propria struttura o comportamento.

**Intercessione:** la capacità di un programma di **modificare** il proprio stato di esecuzione o la sua interpretazione, il suo funzionamento.

**Reificazione:** processo di rappresentare lo stato di esecuzione come dati.

**Structural Reflection:** Si occupa della capacità di un linguaggio di programmazione di offrire una rappresentazione completa (**reificazione**) della struttura del programma in esecuzione. Include la rappresentazione del codice, della sua struttura e dei suoi tipi di dati astratti, come classi, oggetti e altri costrutti.

**Behavioral Reflection:** Si concentra sulla capacità del linguaggio di fornire una reificazione completa della sua semantica e del suo comportamento. Consente di analizzare e intervenire sul significato del linguaggio stesso, sulla sua interpretazione (il "processore" che lo esegue) e sui dettagli del sistema di run-time, come la gestione della memoria, i thread o altri elementi che governano l'esecuzione del programma.

La reflection è **fondamentale** per molti strumenti di sviluppo. I *browser di classi* utilizzano la reflection per elencare i membri delle classi, come variabili e metodi, consentendo di esplorare la struttura di un programma. Gli *ambienti di sviluppo visivi* sfruttano la reflection per accedere alle informazioni sui tipi di dati, fornendo suggerimenti e autocompletamenti che aiutano gli sviluppatori a scrivere codice corretto. I *debugger* la impiegano per esaminare e accedere ai membri privati delle classi, offrendo dettagli sullo stato interno di un programma durante l'esecuzione. Negli *strumenti di test*, la reflection consente di ispezionare dinamicamente tutte le parti di un programma per migliorare la copertura del codice, assicurando che ogni riga venga eseguita almeno una volta. Infine, le *funzionalità di estensibilità* utilizzano la reflection per caricare classi definite dall'utente e creare oggetti dinamicamente, permettendo l'aggiunta di nuove funzionalità senza modificare il codice principale. La reflection presenta diversi **svantaggi**. Innanzitutto, comporta un overhead di performance, poiché richiede la risoluzione dinamica dei tipi di dati durante l'esecuzione, impedendo al compilatore di ottimizzare il codice. L'uso della reflection

richiede permessi speciali e, in ambienti con gestori di sicurezza (ad esempio, Applet con restrizioni), potrebbe non essere possibile eseguire operazioni riflessive. Infine, la reflection espone elementi interni delle classi, come campi privati, rompendo il principio di astrazione.

## La reflection in Java

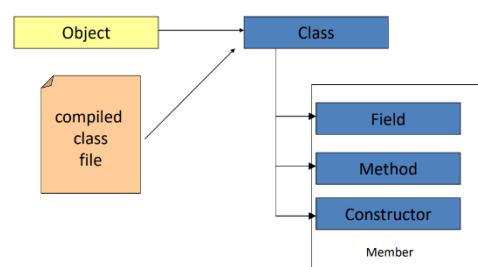
Java consente **l'introspezione**, cioè la capacità di esaminare la struttura del programma, come classi, metodi e campi, e **l'invocazione riflessiva**, che permette di chiamare metodi e accedere a campi a runtime. Tuttavia, non permette la modifica diretta del codice.

Per ogni tipo in Java, la JVM mantiene un oggetto della classe **java.lang.Class**, che rappresenta il tipo e "riflette" le sue caratteristiche, fondamentale per la reflection. Gli oggetti Class sono creati automaticamente dalla JVM quando le classi vengono caricate.

L'oggetto Class fornisce informazioni come il **nome della classe** e i **modificatori** (es. public, private), la **superclasse** e le **interfacce implementate**, i **metodi**, i **campi** (variabili) e i **costruttori**. Gli oggetti Class sono creati automaticamente dalla JVM quando le classi vengono caricate.

L'API per la reflection in Java si trova nel package **java.lang.reflect**, che fornisce gli strumenti per accedere a metodi, campi, costruttori e per eseguire operazioni dinamiche sulle classi.

The Reflection Logical Hierarchy in Java



Il metodo **Object.getClass()** restituisce l'oggetto Class associato all'istanza di un oggetto.

```
Class c = "foo".getClass(); // Restituisce la classe String
```

In alternativa, si può usare il campo **.class** di un tipo.

```
Class c = String.class; // Restituisce la classe String
```

Il metodo statico **Class.forName(String)** accetta come argomento il **nome completo** della classe e restituisce l'oggetto Class corrispondente.

```
Class c = Class.forName("java.util.List"); // Restituisce la classe List
```

### Altri metodi importanti

**getMethods()**: Restituisce un array contenente i metodi pubblici della classe (compresi quelli ereditati).

**getDeclaredMethods()**: Restituisce tutti i metodi dichiarati nella classe, inclusi quelli privati.

**getMethod(String name, Class<?>... parTypes)**: Restituisce un metodo specifico della classe, dato il nome e i tipi di parametri.

**getDeclaredMethod(String name, Class<?>... parameterTypes)**: Restituisce il metodo della classe con il nome specificato e i tipi di parametri indicati, anche se privato.

**getSuperclass()**: Restituisce l'oggetto Class della superclasse.

**isAssignableFrom(Class<?> cls)**: Verifica se la classe corrente è assegnabile dal tipo di classe specificato.

**newInstance()**: Crea una nuova istanza della classe rappresentata dall'oggetto Class.

**isInterface()**: Restituisce true se la classe rappresentata è un'interfaccia.

**getInterfaces()**: Questo restituisce un array di oggetti Class che rappresentano le interfacce implementate dalla classe.

**getSuperclass()**: Questo restituisce l'oggetto Class della superclasse della classe corrente. Se la classe non ha una superclasse (ad esempio, per Object), restituisce null.

**getModifiers()**: Restituisce un intero che rappresenta i modificatori (come public, abstract, final).

#### Metodi per localizzare gli attributi:

**getDeclaredField(String name)**: Restituisce un oggetto Field che rappresenta il campo con il nome specificato. Il campo deve appartenere alla **classe stessa**, quindi non può essere ereditato. Può accedere **anche** a campi **privati**.

```
Field field = myClass.getDeclaredField("age");
```

**getField(String name)**: Restituisce un oggetto Field che rappresenta il campo pubblico con il nome specificato. Il campo **deve** essere **pubblico** e può appartenere a una **superclasse o superinterfaccia**.

**getDeclaredFields()**: Restituisce un array di oggetti Field che riflettono **tutti i campi dichiarati** nella classe o interfaccia, **anche** a quelli privati. Non include campi **ereditati** da superclassi.

**getFields()**: Restituisce un array di oggetti Field che riflettono tutti i campi **pubblici accessibili** della classe o interfaccia, inclusi quelli **ereditati** da superclassi o superinterfacce.

#### Metodi per localizzare gli attributi:

**getDeclaredConstructor(Class<?>... parameterTypes)**: Restituisce un oggetto Constructor che riflette il costruttore specificato della classe, dato il tipo e l'ordine dei parametri. Questo metodo accede ai costruttori **dichiarati nella classe stessa**, inclusi quelli **privati o protetti**.

```
Constructor<?> constructor = myClass.getDeclaredConstructor(String.class, int.class);
```

**getConstructor(Class<?>... parameterTypes)**: Restituisce un oggetto Constructor che riflette solo il **costruttore pubblico** della classe, dato il tipo e l'ordine dei parametri.

Può accedere **solo** ai costruttori **pubblici** della classe.

**getDeclaredConstructors()**: Restituisce un array di oggetti Constructor che riflettono **tutti i costruttori** dichiarati nella classe, inclusi quelli **privati**.

**getConstructors()**: Restituisce un array contenente oggetti Constructor che riflettono tutti i costruttori **pubblici** accessibili della classe.

## Pacchetto java.lang.reflect.\*

L'interfaccia **Member** è l'interfaccia comune per tutti i membri di una classe, come campi, metodi e costruttori.

I campi hanno un **tipo** (es. int, String) e un **valore**. La classe java.lang.reflect.Field fornisce metodi per ottenere informazioni sul tipo di un campo e impostare e ottenere il valore di un campo su un dato oggetto.

```
Object value = field.get(myObject);
field.set(myObject, newValue)
```

I metodi hanno **valori di ritorno**, dei parametri e delle eccezioni che possono lanciare.

La classe **java.lang.reflect.Method** fornisce metodi per **accedere alle informazioni** sul tipo di ritorno e sui parametri del metodo e **invocare il metodo** su un determinato oggetto a runtime, anche se è privato.

```
Class<?> returnType = method.getReturnType();
Object result = method.invoke(myObject, arg1, arg2);
```

I costruttori sono simili ai metodi, ma non restituiscono nulla, il loro scopo è creare una nuova istanza della classe. Inoltre l'invocazione di un costruttore crea un nuovo oggetto di una classe.

La classe **java.lang.reflect.Constructor** offre funzioni per **accedere alle informazioni sui parametri** del costruttore e **invocare il costruttore** per creare una nuova istanza di un oggetto.

```
Class<?>[] paramTypes = constructor.getParameterTypes();
Object newInstance = constructor.newInstance(arg1, arg2);
```

```
public class Btest {
    public String aPublicString;
    private String aPrivateString;
    public Btest(String aString) {
        // ...
    }
    public Btest() {
        // ...
    }
    public Btest(String s1, String s2) {
        // ...
    }
    private void Op1(String s) {
        // ...
    }
    protected String Op2(int x) {
        // ...
    }
    public void Op3() {
        // ...
    }
}
```

```
public class Dtest extends Btest {
{
    public int aPublicInt;
    private int aPrivateInt;
    public Dtest(int x)
    {
        // ...
    }

    private void OpD1(String s) {
        // ...
    }

    public String OpD2(int x) {
        // ...
    }
}
```

## Example: retrieving **public** fields

```
// get all public fields
try{
    Class c = Class.forName("Dtest");
    Field[] publicFields = c.getFields();
    for (int i = 0; i < publicFields.length; ++i) {
        String fieldName = publicFields[i].getName();
        Class typeClass = publicFields[i].getType();
        System.out.println("Field: " + fieldName +
                           " of type " + typeClass.getName());
    }
} catch (ClassNotFoundException e){
    System.out.println("Class not found...");
}
```

```
Field: aPublicInt of type int
Field: aPublicString of type java.lang.String
```

## Example: retrieving **declared** fields

```
Class c = Class.forName("Dtest");

// get all declared fields
Field[] publicFields = c.getDeclaredFields();
for (int i = 0; i < publicFields.length; ++i) {
    String fieldName = publicFields[i].getName();
    Class typeClass = publicFields[i].getType();
    System.out.println("Field: " + fieldName + " of type " +
                       typeClass.getName());
}
```

```
Field: aPublicInt of type int
Field: aPrivateInt of type int
```

## Example: retrieving public constructors

```
// get all public constructors

Constructor[] ctors = c.getConstructors();
for (int i = 0; i < ctors.length; ++i) {
    System.out.print("Constructor (");
    Class[] params = ctors[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k){
        String paramType = params[k].getName();
        System.out.print(paramType + " ");
    }
    System.out.println(")");
}
```

Constructor (int)

## Example: retrieving public methods

```
//get all public methods

Method[] ms = c.getMethods();
for (int i = 0; i < ms.length; ++i) {
    String mname = ms[i].getName();
    Class retType = ms[i].getReturnType();
    System.out.print("Method : " + mname + " returns " + retType.getName() + " "
parameters : ( "));
    Class[] params = ms[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k)
    {
        String paramType = params[k].getName();
        System.out.print(paramType + " ");
    }
    System.out.println(" ");
}
```

Method : OpD2 returns java.lang.String parameters : ( int )  
Method : Op3 returns void parameters : ()  
Method : wait returns void parameters : ()  
Method : wait returns void parameters : ( long int )  
Method : wait returns void parameters : ( long )  
Method : hashCode returns int parameters : ()  
Method : getClass returns java.lang.Class parameters : ()  
Method : equals returns boolean parameters : ( java.lang.Object )  
Method : toString returns java.lang.String parameters : ()  
Method : notify returns void parameters : ()  
Method : notifyAll returns void parameters : ()

## Example: retrieving declared methods

```
//get all declared methods

Method[] ms = c.getDeclaredMethods();
for (int i = 0; i < ms.length; ++i) {
    String mname = ms[i].getName();
    Class retType = ms[i].getReturnType();
    System.out.print("Method : " + mname + " returns " + retType.getName()
+ " parameters : ( ");
    Class[] params = ms[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k)
    {
        String paramType = params[k].getName();
        System.out.print(paramType + " ");
    }
    System.out.println(" ");
}
```

Method : OpD1 returns void parameters : ( java.lang.String )  
Method : OpD2 returns java.lang.String parameters : ( int )

## Generic methods: effects of erasure

Il metodo **getMethod(String name, Class<?>... parameterTypes)** restituisce un oggetto *Method* corrispondente al metodo pubblico specificato, dato il nome e i tipi di parametri.

```
try{
    LinkedList<String> list = new LinkedList<String>();
    Class c = list.getClass();
    Method add = c.getMethod("add", String.class);
} catch (Exception e){
    System.out.println("Method not found");
}
```

In Java, a causa della **erasure dei tipi generici**, le informazioni sui tipi generici (come `<String>`) non sono disponibili a runtime. Questo significa che, quando si utilizza la reflection, il tipo generico viene "cancellato" e Java vede il `LinkedList<String>` come un semplice `LinkedList`. Non esiste alcuna informazione che indichi che la lista è specificatamente di tipo `String`.

**Soluzione:** La reflection può essere utilizzata non solo per **introspezione** ma anche per **manipolazione del programma**, consentendo operazioni dinamiche.

Tramite la reflection, puoi istanziare classi sconosciute a runtime, ad esempio con il metodo `newInstance()` e puoi accedere a campi e invocare metodi privati o sconosciuti come con `getDeclaredField()` o `invoke()`.

## Usare i costruttori di default

Il metodo `newInstance()` di `Class` viene usato per creare una nuova istanza di una classe utilizzando il **costruttore di default** (senza argomenti).

```
Rectangle r = new Rectangle(); // Modo tradizionale  
Class c = Class.forName("java.awt.Rectangle");  
Rectangle r = (Rectangle) c.newInstance(); // Usando reflection
```

Se il costruttore richiede degli argomenti, devi ottenere l'oggetto `Constructor` e poi usare `newInstance()` con i parametri.

```
Rectangle r = new Rectangle(12, 24); // Modo tradizionale  
Class c = Class.forName("java.awt.Rectangle");  
Class[] intArgsClass = new Class[]{int.class, int.class};  
Object[] intArgs = new Object[]{12, 24};  
Constructor ctor = c.getConstructor(intArgsClass);  
Rectangle r = (Rectangle) ctor.newInstance(intArgs); // Usando reflection
```

Usando reflection, puoi ottenere il **valore di un campo** (anche **privato**) con `get()`.

```
Rectangle r = new Rectangle(12, 24);  
Class c = r.getClass();  
Field f = c.getField("height"); // Ottenere il campo 'height'  
Integer h = (Integer) f.get(r); // Leggere il valore
```

Puoi anche **modificare** il valore di un campo con `set()`.

```
Rectangle r = new Rectangle(12, 24);  
Class c = r.getClass();  
Field f = c.getField("width"); // Ottenere il campo 'width'  
f.set(r, 30); // Impostare il nuovo valore
```

Puoi invocare metodi usando reflection con `invoke()`. Devi prima ottenere l'oggetto `Method` con `getMethod()`.

```
String s1 = "Hello";  
String s2 = "World";  
Class c = String.class;  
Class[] paramTypes = new Class[]{String.class};  
Object[] args = new Object[]{s2}; // Argomento 'World'  
Method concatMethod = c.getMethod("concat", paramTypes); // Ottenere il metodo 'concat'  
String result = (String) concatMethod.invoke(s1, args); // Invocare il metodo
```

## AccessibleObject

La classe **Accessible Objects** in Java permette di aggirare le regole di accesso definite dai modificatori di visibilità, come **private** e **final**, in circostanze particolari tramite la reflection.

In genere non è permesso modificare un campo **final**, leggere o scrivere un campo **privato** e invocare un metodo **privato** se non si è un membro della classe, anche se si usano i metodi di reflection.

Un programmatore può richiedere che oggetti **Field**, **Method** e **Constructor** siano "accessibili", anche se sono normalmente non accessibili a causa delle regole di visibilità. E' possibile se non è presente un gestore di sicurezza, o se il gestore di

sicurezza esistente permette di eseguire l'operazione. Se l'accesso è garantito, puoi leggere/scrivere un campo o invocare un metodo, anche se normalmente inaccessibile tramite le regole di visibilità.

La classe **AccessibleObject** è la superclasse di **Field**, **Method**, e **Constructor** e fornisce metodi per gestire l'accessibilità.

**isAccessible()**: Restituisce il valore booleano che indica se l'oggetto è accessibile o meno.

**setAccessible(boolean flag)**: Imposta il flag di accessibilità per l'oggetto. Se impostato a true, l'oggetto diventa accessibile anche se normalmente non lo sarebbe.

**setAccessible(AccessibleObject[] array, boolean flag)**: Imposta l'accessibilità per un array di oggetti (ad esempio più campi o metodi) con un solo controllo di sicurezza.

La classe **java.lang.reflect.Field** permette di accedere ai campi di una classe utilizzando la reflection. Il metodo **get(Object obj)** è utilizzato per ottenere il valore di un campo su un oggetto specificato.

Se il campo ha un tipo primitivo (es. *int*, *boolean*), il valore viene automaticamente convertito e "avvolto" in un oggetto corrispondente (es. *Integer*, *Boolean*).

In background, il valore del campo viene recuperato dal campo sottostante del **Field**.

Se questo oggetto **Field** applica il controllo di accesso del linguaggio Java e il campo sottostante è **inaccessibile**, il metodo lancia un'eccezione **IllegalAccessException**.

Se il campo è **statico**, la classe che ha dichiarato il campo viene **inizializzata**, se non lo è già stata.

**IllegalArgumentException**: Se l'oggetto passato come argomento non è compatibile con la classe o il tipo del campo.

**IllegalAccessException**: Se il campo è inaccessibile per le regole di visibilità e l'accessibilità non è stata forzata tramite `setAccessible(true)`.

## Accessing private fields (2)

The diagram shows a Java code snippet on the left and its execution flow on the right. The code defines a method `getString` that iterates through the declared fields of an object's class. It suppresses Java's access checking for each field and checks if it is static. If not static, it appends the field name and value to a state string. A call to `getString` on a `Cell` object is shown on the right, resulting in the output `[value=5.]`.

```
public static String getString( Object o ) {  
    if ( o == null ) return "null";  
    Class toExamine = o.getClass( );  
    String state = "[";  
    Field[ ] fields = toExamine.getDeclaredFields( );  
    for ( int fi = 0; fi < fields.length; fi++ )  
        try {  
            Field f = fields[ fi ];  
            f.setAccessible( true );  
            if ( !Modifier.isStatic( f.getModifiers( ) ) )  
                state += f.getName() + "=" + f.get( o ) + ", ";  
        } catch ( Exception e ) { return "Exception"; }  
    return state + "]";  
}  
  
class Cell {  
    private int value = 5;  
    ...  
}  
  
Cell c = new Cell( );  
String s = getString( c );  
System.out.println( s );  
  
[value=5.]
```

## Java Annotation

I **Modifiers** come static, final, public, ecc., sono keyword riservate in Java. Forniscono informazioni aggiuntive sugli elementi del programma (ad esempio, una variabile dichiarata final non può essere modificata). Poiché sono parte del linguaggio, i modificatori sono "hardcoded" e non possono essere modificati o estesi.

A volte è necessario aggiungere ulteriori informazioni agli elementi del programma senza modificare il linguaggio stesso. Gli sviluppatori potrebbero voler fornire più metadati personalizzati che i modificatori esistenti non possono gestire.

Le **annotations** si possono considerare come **modificatori "definibili dall'utente"** che forniscono informazioni aggiuntive o direttive per il compilatore e il runtime.

Possono essere utilizzate per specificare comportamenti a livello di compilazione o runtime (ad esempio, ignorare avvertimenti, generare codice, ecc.) e fornire

informazioni a strumenti esterni come framework di testing o di gestione delle dipendenze.

Un esempio comune di annotation è `@Override`, che indica al compilatore che un metodo sta sovrascrivendo un metodo della classe genitore.

Le **annotations** in Java seguono una struttura semplice, composta da un **nome** e, optionalmente, da una serie di **attributi** sotto forma di coppie "**nome = valore**".

**Annotation semplice:** La forma più semplice di un'annotation è solo il nome.

`@Override`

**Annotation con un valore costante** (shorthand): Se un'annotation ha un solo attributo di nome value, puoi usare una scorciatoia per specificare solo il valore.

`@SuppressWarnings("unchecked") == @SuppressWarnings(value = "unchecked")`

**Annotation con più attributi:** Se l'annotation ha più attributi, devi specificare ogni coppia "nome = valore".

`@MyAnnotation(name_1 = "value1", name_2 = 5).`

Gli attributi devono essere espressioni **costanti** valutabili a tempo di compilazione (numeri, stringhe). Gli attributi devono avere un **tipo**, e i valori forniti devono essere convertibili a quel tipo.

Puoi **applicare** annotations alle **dichiarazioni di package**, per esempio per specificare comportamenti o metadati relativi a tutto il pacchetto.

Le annotations possono essere applicate a **classi**, inclusi i tipi enumerativi. *Esempio:* `@Deprecated` su una classe per indicare che non dovrebbe più essere usata.

Anche le **interfacce** possono essere annotate. *Esempio:* `@FunctionalInterface` per indicare che l'interfaccia è funzionale.

Puoi annotare i campi di una classe o **variabili locali** all'interno dei metodi. *Esempio:* `@Inject` per indicare che un campo dovrebbe essere iniettato tramite un framework di dependency injection.

Le annotations si possono applicare anche a **metodi e costruttori**, per specificare comportamenti come il sovraccarico o l'uso in test unitari.

*Esempio:* `@Override` per indicare che un metodo sovrscrive un metodo della classe base.

Anche i **parametri dei metodi** o dei costruttori possono essere annotati, ad esempio per la convalida degli input. *Esempio:* `@NotNull` su un parametro per indicare che non deve essere nullo.

In versioni recenti di Java (dalla versione 8 in poi), le annotations possono essere applicate su qualsiasi tipo, anche in espressioni o dichiarazioni di tipo.

Le annotations possono comparire insieme ad altri **modificatori** (come public, static, final) senza limitazioni sul loro numero. Puoi aggiungere più annotations ad un singolo elemento se necessario.

```
@SuppressWarnings("unchecked")
@Deprecated
public void myMethod() {
    // Codice
}
```

In Java, il compilatore riconosce un insieme di **annotations predefinite** che hanno uno scopo ben preciso. Queste annotations sono direttamente comprese dal compilatore e servono per fornire informazioni specifiche sul codice. Le annotations definite dagli utenti, invece, vengono ignorate dal compilatore ma possono essere usate da strumenti esterni come framework o librerie.

## Alcune Annotations Predefinite:

**@Override**: Serve per dichiarare esplicitamente che un metodo sta sovrascrivendo un metodo della superclasse.

**@Deprecated**: Serve per indicare che l'elemento annotato (metodo, classe o interfaccia) è considerato obsoleto e potrebbe non essere incluso nelle versioni future dell'API Java.

**@SuppressWarnings**: Serve per dire al compilatore di non generare avvisi specifici per certe situazioni, come deprecation, unchecked, empty, ecc.

```
@SuppressWarnings({"deprecation", "empty"})
void antiqueMethod() {
    OldClass.deprecatedMethod();
    // Non c'è nulla qui, ma nessun avviso verrà emesso
}
```

**@FunctionalInterface**: Serve per indicare che un'interfaccia è una **functional interface**, ossia un'interfaccia che deve avere un solo metodo astratto, ideale per l'uso con lambda expressions.

## Perché Definire Annotations Personalizzate?

Le annotations possono essere utilizzate per documentare il codice, fornendo informazioni aggiuntive agli sviluppatori senza influire sulla logica del programma. Gli strumenti esterni possono processare i file .class generati dal compilatore e utilizzare le annotations per generare codice o creare analisi basate su di essi. Puoi usare riflessione (reflection) per ispezionare le annotations durante l'esecuzione del programma e comportarti in base ai metadati definiti.

## Sintassi per Definire un'Annotation Personalizzata

La dichiarazione di un'annotation in Java è simile a quella di un'interfaccia, ma inizia con **@interface**. Gli attributi dell'annotation sono definiti come campi di un'interfaccia, con i loro tipi di ritorno e valori predefiniti (se necessario). Questi attributi funzionano come le proprietà di un'annotation, e possono avere dei valori predefiniti.

*Esempio.*

```
public @interface InfoCode {
    String author();
    String date();
    int ver() default 1;
    int rev() default 0;
    String[] changes() default {};
}
```

Gli attributi di un'annotation possono avere solo **tipi primitivi, String, Class, Enum, Annotation o array** di questi tipi. Puoi specificare un **valore predefinito** per un attributo usando default, come nel caso di ver(), rev() e changes().

Ogni metodo dichiarato nell'annotation determina il nome e il tipo di un attributo. Ad esempio, il metodo String author() definisce un attributo chiamato author di tipo String. Una volta definita un'annotation, può essere utilizzata come qualsiasi altra annotation predefinita. Puoi applicarla a vari elementi del programma come classi, metodi, campi, ecc.

```
@InfoCode(author = "Cristian", date = "2024-10-07", changes = {"Initial version"})
public class MyClass {
    // Codice della classe
}
```

Le definizioni di **annotations** possono essere a loro volta annotate usando le **meta-annotations**, che descrivono i metadati dell'annotation stessa.

**@Target**: Limita gli elementi del programma su cui l'annotation può essere applicata, come METHOD, FIELD, CONSTRUCTOR, PACKAGE, ecc. Si usa l'enum ElementType[] per specificare questi elementi.

**@Retention**: Definisce lo scope in cui l'annotation sarà disponibile. Le opzioni sono:

- SOURCE**: L'annotation esiste solo nel codice sorgente.

- b. **CLASS**: L'annotation è inclusa nel bytecode ma non disponibile a runtime.
- c. **RUNTIME**: L'annotation è accessibile durante l'esecuzione del programma tramite riflessione.

**@Inherited**: È una **marker annotation** che indica che l'annotation è ereditata dalle sottoclassi quando viene applicata a una classe.

Le **annotations** definite nel codice possono essere recuperate ed elaborate a runtime utilizzando la **Reflection API** di Java.

## Recupero delle Annotations a Runtime

Gli strumenti possono analizzare le annotations presenti nei file .class. Il pacchetto **javax.annotation.processing** fornisce un'API per scrivere strumenti che processano queste annotations.

Le annotations possono essere recuperate durante l'esecuzione del programma utilizzando la **Reflection API**, che permette di ispezionare classi, metodi e campi.

Le classi del pacchetto **java.lang.reflect** e **java.lang.Class** offrono metodi per accedere alle annotations associate agli elementi del programma.

**getAnnotations():Class**: Restituisce un array di istanze di Annotation, che contiene tutte le annotations presenti su un elemento.

```
Annotation[] annotations = MyClass.class.getAnnotations();
```

**getAnnotation(Class<T> annotationClass):Method**: Restituisce l'annotation di un elemento specifico (come un metodo) per un determinato tipo di annotation. Se l'annotation non è presente, restituisce null.

```
Deprecated deprecated = MyClass.class.getMethod("myMethod")
    .getAnnotation(Deprecated.class);
```

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.PACKAGE})
interface InfoCode {
    String author();
    String date();
    int ver() default 1;
    int rev() default 0;
    String[] changes() default {};
}

@InfoCode(author="Gigi", date="8/12/2008")
public class TestAnno {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Class c = TestAnno.class;
        System.out.print("I am: " + c.toString());
        InfoCode ic = (InfoCode)c.getAnnotation(InfoCode.class);
        if (ic != null)
            System.out.print(" v" + ic.ver() + "." + ic.rev()
                + " by " + ic.author());
        System.out.println();
    }
}
```

**A comprehensive example**

## AP09-FRAMEWORK E IOC

Un **software framework** è una raccolta di codice che offre funzionalità generiche e predefinite, progettata per essere personalizzata o estesa con il codice dell'utente al fine di soddisfare requisiti specifici.

Fornisce un approccio standardizzato per organizzare il codice e include strumenti e librerie per compiti comuni, come la gestione di interfacce utente, database e input/output. I principali vantaggi sono riutilizzabilità del codice, minor tempo di sviluppo, struttura del codice ben organizzata e possibilità di estendere il codice.

### Principali framework

**Spring (Java):** Framework progettato per lo sviluppo di applicazioni enterprise e web, basato su principi come l'iniezione di dipendenze e l'inversione di controllo (IoC). Offre strumenti per gestione delle transazioni, programmazione orientata agli aspetti (AOP), e un modulo completo per lo sviluppo web (Spring MVC).

**.NET Framework (C# e altri linguaggi .NET):** Piattaforma sviluppata da Microsoft per creare ed eseguire applicazioni su Windows. Fornisce un runtime per la gestione dell'esecuzione del codice, una libreria di classi completa per lo sviluppo di interfacce grafiche, applicazioni web e accesso ai dati, e supporto per diversi linguaggi di programmazione come C#, VB.NET e F#.

**Django (Python):** Framework progettato per lo sviluppo rapido di applicazioni web scalabili, che adotta la filosofia "batterie incluse", offrendo funzionalità integrate come un ORM per l'interazione con i database, un pannello di amministrazione preconfigurato, sistemi di autenticazione e gestione delle sessioni. È strutturato per promuovere un design del codice pulito seguendo il pattern Model-View-Template (MVT).

**Ruby on Rails (Ruby):** Framework per lo sviluppo di applicazioni web, progettato seguendo l'architettura MVC. Promuove il principio "Convention over Configuration", riducendo la necessità di configurazioni manuali e accelerando il processo di sviluppo attraverso l'uso di convenzioni predefinite. Inoltre, adotta il principio "Don't Repeat Yourself" (DRY), incoraggiando la scrittura di codice riutilizzabile e mantenibile.

**React (JavaScript/TypeScript):** Libreria sviluppata da Facebook per la costruzione di interfacce utente interattive e riutilizzabili. Si basa su un'architettura a componenti, consentendo la suddivisione dell'interfaccia in parti modulari e gestibili. Utilizza un DOM virtuale per ottimizzare le prestazioni del rendering, aggiornando in modo efficiente solo le parti dell'interfaccia che cambiano. React supporta JSX, una sintassi che combina JavaScript e HTML, e può essere utilizzato con TypeScript per aggiungere tipizzazione statica, migliorando la robustezza e la manutenzione del codice.

**Laravel (PHP):** Framework per lo sviluppo di applicazioni web in PHP, basato sul pattern architettonico MVC, che fornisce strumenti integrati per il routing, l'ORM (Eloquent) per la gestione dei database, migrazioni, autenticazione, e testing. Laravel promuove best practice di sviluppo, facilitando la scrittura di codice pulito e manutenibile grazie al sistema di template Blade e a un ecosistema ricco di pacchetti estensibili.

**Qt (C++):** Framework per lo sviluppo di applicazioni multiplattforma, che fornisce librerie per la creazione di interfacce grafiche, gestione di eventi, networking, accesso ai database e altre funzionalità essenziali. Supporta anche QML, un linguaggio dichiarativo per lo sviluppo di interfacce utente moderne e reattive. Qt consente di sviluppare software compatibile con diversi SO, riducendo la necessità di modifiche al codice sorgente per ciascuna piattaforma.

**Apache Hadoop:** Framework open-source per l'elaborazione e la gestione di grandi volumi di dati distribuiti su cluster di computer. Basato sul paradigma di programmazione MapReduce, include il file system distribuito HDFS (Hadoop Distributed File System) per lo storage scalabile. Hadoop consente l'elaborazione parallela dei dati, garantendo tolleranza ai guasti e scalabilità orizzontale, facilitando l'analisi e la gestione di dataset di dimensioni elevate in ambienti distribuiti.

I framework e gli IDE sono concetti ortogonali: Un framework può essere supportato da diversi IDE (es: Spring supportato da Eclipse, IntelliJ IDEA, NetBeans). Un IDE può supportare più framework (es: NetBeans supporta JavaBeans, Spring, J2EE, Maven, Hibernate, ecc.).

## Component framework

Un **component framework** è una struttura progettata per supportare la creazione, composizione, rilascio ed esecuzione di **componenti sw** seguendo un modello definito (Component Model). Fornisce agli sviluppatori strumenti per progettare interfacce precise, assicurando compatibilità e coerenza tra i componenti. Facilita la composizione e la connessione dei componenti, permettendo di integrarli facilmente secondo i meccanismi definiti dal modello. Implementa il paradigma plug-and-play, consentendo l'aggiunta e la rimozione flessibile dei componenti per una gestione modulare e scalabile. Inoltre, mette a disposizione funzionalità predefinite, come strumenti per automatizzare operazioni comuni, e crea un ambiente di esecuzione controllato, garantendo che i componenti interagiscano in modo sicuro ed efficiente.

## Caratteristiche principali dei framework

Un framework offre un'architettura astratta con comportamenti predefiniti e parti già implementate e riutilizzabili, come librerie con API e programmi pronti all'uso ma estendibili, utili per risolvere problemi di design comuni.

Consente al programmatore di personalizzare il comportamento inserendo codice in punti specifici, spesso tramite ereditarietà o implementazione di interfacce. Grazie all'**inversion of control**, il framework gestisce il flusso del programma chiamando il codice personalizzato nei momenti opportuni, diversamente dalle librerie, in cui è il programmatore a invocare le funzioni. I framework offrono astrazioni riutilizzabili che semplificano la programmazione e strumenti per sviluppo, configurazione o gestione dei contenuti. Sebbene il codice del framework non sia modificabile direttamente, può essere esteso tramite sovrascritture selettive (overriding) per adattarlo alle esigenze specifiche. Inoltre, gestiscono il ciclo di vita dei componenti, incluse creazione, inizializzazione e distruzione, spesso tramite tecniche come Dependency Injection o gestori di eventi.

### Metodi di estensione di un framework

I framework sono progettati per essere estesi, permettendo agli sviluppatori di personalizzarne il comportamento in base alle necessità specifiche dell'applicazione.

**Sottoclassi e overriding:** Creare sottoclassi e sovrascrivere metodi esistenti per modificare o aggiungere comportamenti specifici.

**Implementazione di interfacce:** Implementare interfacce predefinite dal framework per definire comportamenti personalizzati.

**Registrazione di gestori di eventi:** Definire e registrare *event handlers* per reagire a eventi o cambiamenti specifici gestiti dal framework.

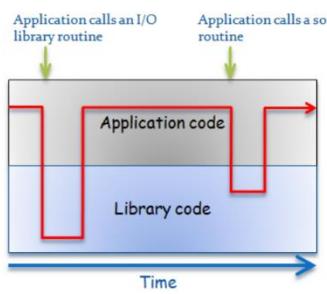
**Plug-in:** Alcuni framework supportano l'aggiunta di codice extra in formato modulare, come plug-in, per integrare nuove funzionalità senza alterare direttamente il codice del framework.

### Inversion of Control

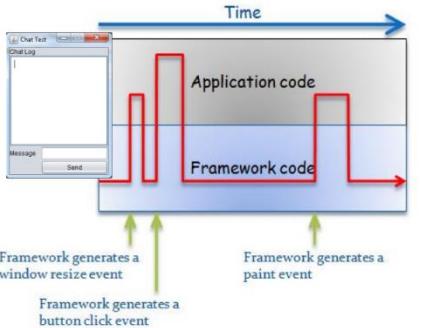
L'**Inversion of Control** (IoC) è un principio secondo cui il controllo del flusso dell'applicazione è delegato a un framework, che gestisce l'esecuzione e le interazioni tra i componenti. Invece di essere il codice dell'applicazione a invocare esplicitamente funzioni o metodi, è il framework che richiama i metodi personalizzati dell'utente al momento opportuno, in base a eventi o configurazioni predefinite. Questo consente di astrarre la gestione del flusso e di concentrarsi sulla logica specifica dell'applicazione.

Il framework impone una struttura applicativa, ma consente personalizzazioni tramite callback o specializzazioni di classi. Il framework funge da scheletro estensibile, orchestrando il ciclo di vita degli oggetti e l'interazione tra i componenti. Gli sviluppatori si concentrano sulla logica applicativa, mentre il framework gestisce operazioni di basso livello, come la creazione, gestione degli oggetti e gestione degli eventi.

## Traditional Program Execution



## Inversion of Control



A prima vista, i framework non sono molto diversi dalle **librerie**, che forniscono anche codice riutilizzabile. La **differenza principale** è l'uso dell'**Inversion of Control**. Con un framework, è il framework che chiama il codice del programmatore, mentre con una libreria è il programmatore a chiamare le funzioni della libreria. A volte il termine "framework" è usato a sproposito per indicare una **libreria ben progettata**.

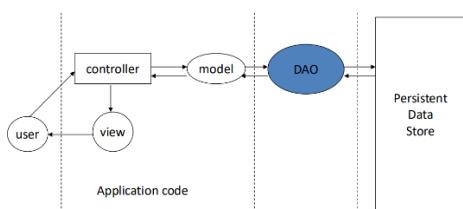
Il **Java Collection Framework** (Java) e la **C++ Standard Template Library (STL)** sono esempi di librerie ben progettate, ma non sono propriamente framework, poiché il controllo rimane al programmatore e non al framework.

## Dependency Injection

### Esempio concreto Trade Monitor

Un trader desidera che il sistema **rifiuti operazioni** quando l'esposizione raggiunge un certo limite. Per realizzare questo: La classe **TradeMonitor** fornisce un metodo chiamato **TryTrade** che verifica se l'operazione può essere accettata o meno. Il metodo **TryTrade** controlla l'esposizione attuale e il limite di esposizione, che sono memorizzati in un archivio persistente, utilizzando un altro componente chiamato **DAO (Data Access Object)**. Il DAO gestisce tutte le operazioni di lettura e scrittura dei dati da una fonte, come un db, mantenendo il codice dell'applicazione indipendente dai dettagli tecnici di accesso ai dati.

- A Java EE design pattern



## Trade Monitor – The first design

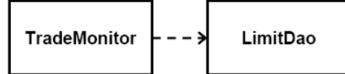
```
public class TradeMonitor
{
    private LimitDao limitDao;

    public TradeMonitor()
    {
        limitDao = new LimitDao();
    }

    public bool TryTrade(string symbol, int amount)
    {
        int limit = limitDao.GetLimit(symbol);
        int exposure = limitDao.GetExposure(symbol);
        return (exposure + amount > limit) ? false : true;
    }
}
```

```
public class LimitDao
{
    public int GetExposure(string symbol)
    {
        // Do something with the database
    }

    public int GetLimit(string symbol)
    {
        // Do something with the database
    }
}
```



In questo scenario, il **TradeMonitor** dipende direttamente dal **DAO** per ottenere i dati necessari. Questo crea un legame troppo forte tra i due componenti. Cambiare la fonte

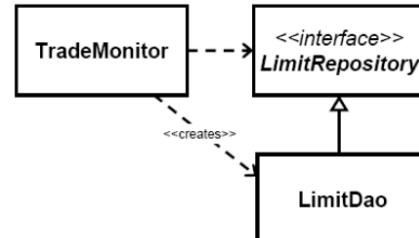
dei dati, ad esempio passando da un database a una cache distribuita, richiederebbe modifiche dirette a **TradeMonitor**. È difficile testare **TradeMonitor** in isolamento, poiché i dati reali del database devono essere usati, complicando i test.

## Trade Monitor – The Design Refactored (1)

```
public interface LimitRepository
{
    int GetExposure(string symbol);
    int GetLimit(string symbol);
}
public class LimitDao extends LimitRepository
{
    public int GetExposure(string symbol) {...}
    public int GetLimit(string symbol) {...}
}
public class TradeMonitor
{
    private LimitRepository limitRepository;

    public TradeMonitor()
    {
        limitRepository = new LimitDao();
    }

    public bool TryTrade(string symbol, int amount)
    {
        ...
    }
}
```



27

Viene creata un'interfaccia chiamata **LimitRepository**, che definisce i metodi `GetExposure` e `GetLimit`. Questa interfaccia è implementata da **LimitDao**. Ora, la logica di **TradeMonitor** non dipende direttamente dall'implementazione concreta **LimitDao**, ma dall'interfaccia **LimitRepository**, ma il costruttore di **TradeMonitor** continua a creare un'istanza di **LimitDao**, creando una dipendenza forte. Se si vuole usare un'altra implementazione di **LimitRepository**, si deve comunque modificare il codice del costruttore di **TradeMonitor**.

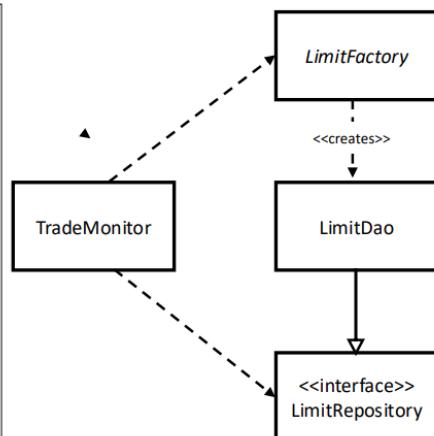
## Trade Monitor – The Design Refactored (2)

```
public class LimitFactory
{
    public static LimitRepository GetLimitRepository()
    {
        return new LimitDao();
    }
}

public class TradeMonitor
{
    private LimitRepository limitRepository;

    public TradeMonitor()
    {
        limitRepository = LimitFactory.GetLimitRepository();
    }

    public bool TryTrade(string symbol, int amount)
    {
        ...
    }
}
```



28

La **Factory** è una classe separata che ha il compito di creare le istanze necessarie. In questo caso, la classe **LimitFactory** è responsabile della creazione dell'istanza di **LimitRepository**. La classe **TradeMonitor** non crea direttamente un'istanza di **LimitDao**, ma si affida alla **LimitFactory** per ottenere l'implementazione di **LimitRepository**. **TradeMonitor** ora è **disaccoppiato da LimitDao**, poiché non fa direttamente riferimento alla sua implementazione. Tuttavia, il codice è ancora **fortemente accoppiato alla Factory**, che decide quale implementazione di **LimitRepository** utilizzare. Questo significa che, se volessimo cambiare l'implementazione di **LimitRepository** (ad esempio, passare da **LimitDao** a una cache distribuita), dovremmo modificare la **LimitFactory**, ma non più la classe **TradeMonitor**.

## Trade Monitor – The Design Refactored (3)

```

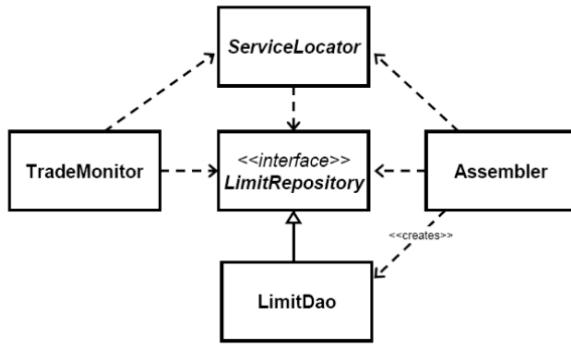
public class ServiceLocator{
    public static void RegisterService(Type t, object o)
    { . . . }
    public static object GetService(Type t)
    { . . . }
}

public class TradeMonitor{
    private LimitRepository limitRepository;

    public TradeMonitor(){
        object o =
            ServiceLocator.GetService(typeof(LimitRepository));
        limitRepository = (LimitRepository) o;
    }

    public bool TryTrade(string symbol, int amount){
        . .
    }
}

```



Il **Service Locator** è un oggetto che funge da registro centrale per la gestione delle dipendenze. Registra e restituisce istanze di servizi (ad esempio, LimitDao) tramite due metodi principali:

**RegisterService(Type t, object o):** Registra un'istanza associata a un tipo specifico.

**GetService(Type t):** Restituisce il servizio registrato per il tipo richiesto.

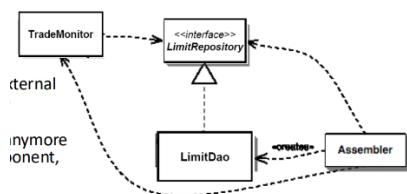
La classe **TradeMonitor** utilizza il Service Locator per ottenere un'istanza di LimitRepository (ad esempio, LimitDao) senza conoscere la sua implementazione specifica, garantendo maggiore flessibilità.

Un **Assembler esterno** è responsabile della configurazione iniziale del Service Locator, registrando quali implementazioni devono essere utilizzate per i vari tipi.

Il Service Locator **disaccoppia TradeMonitor da LimitDao**, poiché TradeMonitor non dipende direttamente dall'implementazione concreta di LimitRepository. Ciò riduce il legame tra i componenti, migliorando la modularità del sistema.

Tuttavia, ogni componente che necessita di una dipendenza deve avere un riferimento al Service Locator. Questo crea un punto centrale di dipendenza, aggiungendo complessità e richiedendo una configurazione accurata, che può diventare difficile da gestire in sistemi di grandi dimensioni.

L'utilizzo del **Service Locator** rappresenta una forma di **Inversion of Control**, in cui il controllo sulla gestione delle dipendenze è invertito. Senza IoC, **TradeMonitor** crea direttamente un'istanza di LimitDao, legandosi alla sua implementazione. Con IoC, un **Assembler esterno** configura LimitDao (o un'altra implementazione di LimitRepository) e lo registra nel Service Locator, che lo fornisce a **TradeMonitor** su richiesta. In questo modo, **TradeMonitor** non gestisce più direttamente LimitDao, rendendo il sistema più flessibile e facilmente testabile, poiché le dipendenze possono essere sostituite con implementazioni diverse o mock.



31

## Dependency injection

La **Dependency Injection (DI)** è un design pattern in cui le dipendenze di un oggetto (altri oggetti o servizi di cui ha bisogno per funzionare) vengono fornite dall'esterno anziché essere create al suo interno. Questo processo trasferisce al framework o a un componente esterno la responsabilità di istanziare e gestire le dipendenze.

La DI offre numerosi **vantaggi**, rendendo i sistemi più modulari, flessibili e manutenibili. Elimina l'accoppiamento forte, poiché le dipendenze non sono codificate direttamente nei componenti, consentendo di scegliere dinamicamente tra diverse implementazioni di un'interfaccia di dipendenza durante l'esecuzione. La DI migliora la testabilità, permettendo di sostituire le dipendenze reali con oggetti mock durante i test, facilitando la verifica del codice. Inoltre, favorisce la riutilizzabilità, poiché i componenti progettati con DI possono essere utilizzati in contesti diversi senza modifiche. Infine, consente l'estendibilità del sistema, permettendo di aggiungere o sostituire componenti senza modificare il codice principale.

## Dependency injection based on setter methods

```
public class TradeMonitor
{
    private LimitRepository limitRepository;

    // Metodo setter per iniettare la dipendenza
    public void SetLimitRepository(LimitRepository repository)
    {
        limitRepository = repository;
    }

    public bool TryTrade(string symbol, int amount)
    {
        // Logica per gestire la trade
        return true;
    }
}
```

**Setter Injection:** È una tecnica di iniezione delle dipendenze in cui una dipendenza viene fornita a un oggetto tramite un metodo setter. La creazione e la risoluzione della dipendenza sono delegate a un componente o framework esterno, mentre l'oggetto riceve la dipendenza dopo la sua istanziazione.

**Svantaggi:** Un oggetto potrebbe essere istanziato senza tutte le dipendenze necessarie, causando potenziali errori se non gestito correttamente. Le dipendenze possono essere modificate durante l'esecuzione, introducendo il rischio di comportamenti imprevisti rispetto a metodi più rigidi come la constructor injection.

## Dependency injection based on constructor

```
public class TradeMonitor
{
    private LimitRepository limitRepository;

    public TradeMonitor(LimitRepository
                        limitRepository)
    {
        this.limitRepository = limitRepository;
    }

    public bool TryTrade(string symbol, int amount){
        . .
    }
}
```

**Constructor Injection:** È una tecnica di iniezione delle dipendenze in cui le dipendenze vengono fornite all'oggetto tramite il costruttore al momento della sua creazione. Questo approccio garantisce che tutte le dipendenze necessarie siano disponibili prima che l'oggetto venga utilizzato. Tutte le dipendenze sono fornite al momento della creazione, evitando che l'oggetto sia incompleto o mal configurato.

**Svantaggi:** Se ci sono dipendenze reciproche tra oggetti (A dipende da B e B dipende da A), gestirle può diventare difficile con il costruttore. Con molte dipendenze, il costruttore può diventare ingombrante e difficile da gestire. Se ci sono molte dipendenze opzionali, si possono creare numerose versioni del costruttore. Se aggiungi nuove dipendenze nel tempo, tutti gli utenti della classe devono essere aggiornati per gestire il nuovo parametro del costruttore.

## Contenitori IoC

Un **container** di un framework è un ambiente di esecuzione che gestisce automaticamente la creazione, configurazione, interazione e ciclo di vita dei componenti di un'applicazione. Offre funzionalità essenziali come gestione delle risorse, iniezione di dipendenze, comunicazione tra componenti e automazione di operazioni comuni, consentendo agli sviluppatori di concentrarsi sulla logica di business senza preoccuparsi di dettagli infrastrutturali.

Con l'**Inversion of Control** il framework chiama automaticamente metodi specifici del componente o configura le sue dipendenze, ad esempio tramite setters o metodi predefiniti.

In un sistema ben progettato, **alta coesione e basso accoppiamento** lavorano insieme per creare una rete di oggetti che collaborano in modo efficace e indipendente.

**Alta coesione:** Proprietà di un oggetto che garantisce che svolga un insieme di compiti ben definiti e strettamente correlati. Ogni oggetto include tutte le funzionalità necessarie per completare il proprio compito, riducendo la dipendenza da altre parti del sistema.

**Basso accoppiamento:** Caratteristica di un sistema in cui gli oggetti interagiscono tra loro con il minimo legame possibile. Ogni oggetto ha conoscenza limitata degli altri e comunica tramite interfacce ben definite. In questo modo è più facile aggiungere nuove funzionalità senza modificare l'intero sistema. Ogni componente può essere testato separatamente, semplificando i test unitari. I componenti possono essere riutilizzati in altri contesti o progetti senza troppe modifiche.

La configurazione delle dipendenze nei container IoC è spesso esterna, tramite file (es. XML) o annotazioni. Il container crea gli oggetti necessari e soddisfa tutte le loro dipendenze in base alla configurazione, assicurandosi che gli oggetti siano pronti all'uso. Il contenitore gestisce anche il ciclo di vita degli oggetti, come l'inizializzazione e la distruzione, e può eseguire codice specifico dopo l'iniezione delle dipendenze. Tramite tecniche come la **reflection**, può determinare automaticamente le dipendenze esaminando i costruttori e creando le istanze richieste. La maggior parte dei contenitori IoC supporta il **auto-wiring**, collegando automaticamente oggetti (bean) a dipendenze dichiarate tramite costruttore o setter, basandosi su criteri come il nome o il tipo. Ad esempio, se una classe ha una proprietà chiamata "service" e il contenitore conosce un bean di tipo Service, questo viene automaticamente associato senza configurazione manuale.

## Framework Spring

Nel framework Spring, il **contenitore IoC (Inversion of Control)** è il nucleo del sistema. Si occupa di gestire gli oggetti, chiamati **bean**, che rappresentano i componenti dell'applicazione. I bean vengono creati, configurati e gestiti dal contenitore durante tutto il loro ciclo di vita, dalla creazione fino alla distruzione.

Il contenitore utilizza la **Dependency Injection (DI)** per collegare e configurare i bean in base alle loro dipendenze, definite tramite **metadata di configurazione**. Questi metadata specificano: Come creare un bean. Il ciclo di vita del bean. Le dipendenze del bean.

La configurazione può essere fornita in tre modi:

**File XML:** Metodo standard per definire i metadata di configurazione.

**Annotazioni:** Definizione diretta della configurazione nel codice sorgente.

**Configurazione basata su Java:** Utilizzo di classi Java per specificare i bean e le loro dipendenze.

Il contenitore IoC supporta funzionalità come l'**auto-wiring**, che permette il collegamento automatico delle dipendenze basandosi su nome o tipo, riducendo il codice necessario e migliorando la leggibilità.

```

public class HelloWorld {
    private String message;

    // Metodo setter per l'iniezione della dipendenza
    public void setMessage(String message) {
        this.message = message;
    }

    public void getMessage() {
        System.out.println("Your Message: " + message);
    }
}

```

Copia codice

#### File di Configurazione XML

```

xml

    <!-- Definizione del bean -->
    <bean id="helloWorld" class="com.example.HelloWorld">
        <!-- Iniezione della dipendenza tramite il setter -->
        <property name="message" value="Hello, Spring!" />
    </bean>
</beans>

```

Copia codice

## AP-10: On Designing Software Frameworks

La comprensione dei software frameworks richiede quattro livelli di apprendimento:

**Concetti di OOP:** È indispensabile conoscere inheritance, polymorphism, encapsulation e delegation, poiché i frameworks sono tipicamente implementati in linguaggi a oggetti come Java.

**Utilizzo dei frameworks:** Bisogna conoscere e saper applicare i concetti e le tecniche necessarie per creare applicazioni personalizzate usando i frameworks.

**Progettazione dei frameworks:** È importante essere in grado di progettare e implementare dettagliatamente frameworks per i quali gli aspetti comuni e variabili sono già noti.

**Analisi delle software family:** Imparare ad analizzare una potenziale famiglia di software, identificandone gli aspetti comuni e variabili e valutando le architetture alternative del framework.

Una **software family** è un insieme di programmi correlati che condividono un nucleo di caratteristiche comuni, ma includono elementi variabili progettati per soddisfare esigenze specifiche. Il principio fondamentale consiste nel massimizzare il riutilizzo del codice per le parti comuni e nel facilitare l'adattamento delle parti variabili.

### Terminologia nei framework

**Frozen Spot:** Aspetto comune e invariabile condiviso da tutti i membri della software family. Rappresenta le parti fisse che non cambiano tra le diverse applicazioni della famiglia.

**Hot Spot:** Aspetto variabile della software family. Rappresenta le parti che si differenziano tra i vari membri per adattarsi a esigenze specifiche.

**Template Method:** È un metodo concreto (non astratto) definito in una classe base (astratta). Questo metodo implementa un frozen spot, che è un comportamento che è comune a tutti i membri della famiglia. Quindi, ogni membro della famiglia utilizza questo comportamento di base.

**Hook Methods:** Metodi astratti che rappresentano gli hot spots. Sono implementati in modo diverso nei membri della famiglia, consentendo la personalizzazione del comportamento.

**Inversion of Control:** Il *template method* chiama un *hook method* per invocare una funzione specifica per uno dei membri della famiglia. Questo approccio viene chiamato "Inversion of Control" perché invece che lo scenario di un membro della famiglia che chiama direttamente le proprie funzioni, è il *template method* che decide quando e come chiamare i *hook methods*.

**Hot Spot Subsystem:** Un hot spot subsystem è realizzato come un sottosistema costituito da una classe base astratta e da sottoclassi concrete. Queste ultime

implementano i **hook methods** per definire il comportamento variabile specifico dei membri della famiglia.

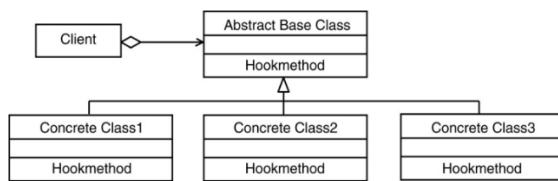


Fig. 1. Hot spot subsystem.

## Two Principles for Framework Construction

### Unification Principle (basato sul Template Method Design Pattern)

L'**Unification Principle** utilizza l'**ereditarietà** per implementare il sottosistema dei *hot spots*. I **template methods** (metodi concreti) e i **hook methods** (metodi astratti che rappresentano i *hot spots*) sono definiti nella **stessa classe base astratta**.

I **hook methods** vengono implementati nelle sottoclassi della classe base, consentendo la personalizzazione delle parti variabili del framework.

### Separation Principle (basato sul Strategy Design Pattern):

Il **Separation Principle** utilizza **la delega** per implementare il sottosistema dei *hot spots*. I **template methods** sono definiti in una classe concreta, mentre i **hook methods** sono definiti in una classe astratta separata e implementati nelle relative sottoclassi. I **template methods** delegano il lavoro a un'istanza della sottoclasse che implementa i **hook methods**, separando così il comportamento fisso da quello variabile.

## Template Method Design Pattern

Il **Template Method** è un pattern che permette di strutturare un algoritmo, lasciando che alcune parti specifiche siano personalizzate dalle sottoclassi.

L'intento è definire lo scheletro di un algoritmo in un metodo `templateMethod`, delegando alle sottoclassi l'implementazione di alcuni passi.

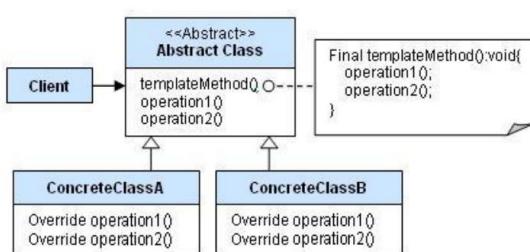
Il **template method** è definito in una classe astratta e descrive l'algoritmo in termini di operazioni astratte che le sottoclassi possono sovrascrivere per fornire comportamenti specifici.

Il template method può includere tre tipi di operazioni:

**Concrete Operations:** Operazioni già implementate nella classe base e comuni a tutti.

**Primitive Operations:** Operazioni astratte che devono essere implementate nelle sottoclassi per fornire funzionalità obbligatorie.

**Hook Operations:** Operazioni opzionali con un comportamento predefinito che le sottoclassi possono sovrascrivere se necessario. Se non sovrascritte, queste operazioni non hanno effetto.



L'implementazione del **Template Method** varia in base al linguaggio di programmazione, utilizzando i relativi modificatori di visibilità o controlli di accesso:

**In Java**

**Template method:** Questo metodo definisce la struttura dell'algoritmo e non deve essere modificato nelle sottoclassi. Per impedirne l'override, può essere dichiarato come **public final**.

**Concrete Operations:** Le operazioni comuni a tutti non devono essere accessibili direttamente dalle sottoclassi, ma solo dal *template method*, quindi devono essere dichiarate **private**.

**Primitive Operations:** I metodi astratti che le sottoclassi devono implementare per personalizzare l'algoritmo, devono essere dichiarate **protected abstract**, indicando che le sottoclassi sono obbligate a fornirne un'implementazione.

**Hook Operations:** I metodi astratti che le sottoclassi possono sovrascrivere se necessario devono essere dichiarate **protected**.

### Esempio Divide and Conquer framework

Il *Divide and Conquer framework* è un esempio di come **costruire e utilizzare un framework** basato sulla famiglia di algoritmi *divide and conquer*. Questa tecnica risolve un problema suddividendolo ricorsivamente in sotto-problemi dello stesso tipo, risolvendo ogni sotto-problema in modo indipendente, e poi combinando le soluzioni per ottenere la soluzione finale.

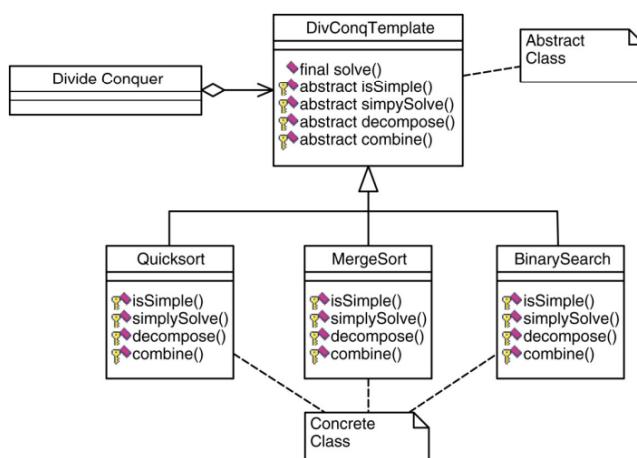


Fig. 3. Template method for divide and conquer.

```

abstract public class DivConqTemplate
{
    public final Solution solve(Problem p)
    {
        Problem[] pp;
        if (isSimple(p)) { return simplySolve(p); }
        else { pp = decompose(p); }
        Solution[] ss = new Solution[pp.length];
        for(int i=0; i < pp.length; i++)
        {
            ss[i] = solve(pp[i]);
        }
        return combine(p,ss);
    }
    abstract protected boolean isSimple (Problem p);
    abstract protected Solution simplySolve (Problem p);
    abstract protected Problem[] decompose (Problem p);
    abstract protected Solution combine(Problem p,Solution[] ss)

    function solve (Problem p) returns Solution // template method
    { if isSimple(p)                                // hot spots
        return simplySolve(p);
    else
        sp[] = decompose(p);
        for (i= 0; i < sp.length; i = i+1)
            sol[i] = solve(sp[i]);
    return combine(sol);
}
  
```

Il metodo **solve()** è il **template method** perché la sua implementazione è comune a tutti gli algoritmi della famiglia. Questo metodo gestisce la struttura generale della risoluzione del problema: verifica se il problema è semplice, lo risolve direttamente se lo è, altrimenti lo scomponete, risolve i sotto-problemi e combina le soluzioni.

I metodi **isSimple()**, **simplySolve()**, **decompose()**, e **combine()** sono i **hook methods** perché la loro implementazione varia tra i diversi membri della famiglia. Ogni algoritmo avrà una diversa versione di questi metodi.

Il codice seguente rappresenta l'implementazione di quicksort usando il framework DivideConquer tramite Unification Principle.

```
public class QuickSort extends DivConqTemplate {
    protected boolean isSimple(Problem p) {
        return (((QuickSortDesc) p).getFirst() >= ((QuickSortDesc) p).getLast());
    }

    protected Solution simplySolve(Problem p) {
        return (Solution) p;
    }

    protected Problem[] decompose(Problem p) {
        int first = ((QuickSortDesc) p).getFirst();
        int last = ((QuickSortDesc) p).getLast();
        int[] a = ((QuickSortDesc) p).getA();
        int x = a[first]; // pivot value
        int sp = first;
        for (int i = first + 1; i <= last; i++) {
            if (a[i] < x) {
                swap(a, ++sp, i);
            }
        }
        swap(a, first, sp);
        Problem[] ps = new QuickSortDesc[2];
        ps[0] = new QuickSortDesc(a, first, sp - 1);
        ps[1] = new QuickSortDesc(a, sp + 1, last);
        return ps;
    }

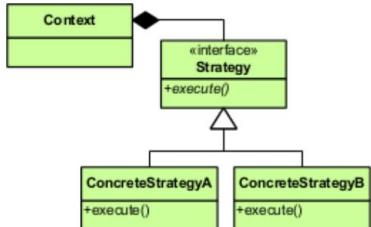
    protected Solution combine(Problem p, Solution[] ss) {
        return (Solution) p;
    }

    private void swap(int[] a, int first, int last) {
        int temp = a[first];
        a[first] = a[last];
        a[last] = temp;
    }
}
```

Algoritmi simili, come **MergeSort**, possono essere implementati usando lo stesso framework. In quel caso, il metodo `combine()` svolgerebbe gran parte del lavoro per unire i sottoarray ordinati.

## Strategy Design Pattern – Separation Principle

Il **Strategy Design Pattern** è un pattern che consente la selezione di un algoritmo (o parte di esso) a runtime, delegando la responsabilità dell'implementazione alle sottoclassi che implementano una stessa interfaccia.

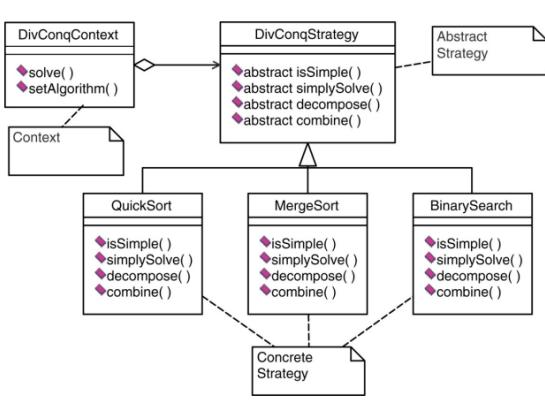


**Interfaccia Strategy:** Definisce un metodo generico (ad esempio, `execute()`) che rappresenta l'operazione da personalizzare.

**Concrete Strategies:** Sono classi che implementano l'interfaccia `Strategy`, fornendo diverse versioni o varianti dell'algoritmo.

**Context:** La classe che utilizza il pattern. Contiene un riferimento a un'istanza di `Strategy` e invoca il metodo definito nell'interfaccia (`execute()`), delegando il comportamento alle concrete strategies. Permette di scegliere quale strategy utilizzare.

Il **client** crea o fornisce al **context** un'istanza di una specifica strategia concreta. Il **context** utilizza l'interfaccia comune `Strategy` per invocare il metodo appropriato, senza preoccuparsi dei dettagli dell'implementazione.



```

function solve (Problem p) returns Solution // template method
{ if isSimple(p) // hot spots
    return simplySolve(p);
else
    sp[] = decompose(p);
    for (i= 0; i < sp.length; i = i+1)
        sol[i] = solve(sp[i]);
    return combine(sol);
}

```

**DivConqContext** è la classe concreta che funge da contesto. Contiene il metodo `solve()` (template method), che implementa la logica generale, delegando le operazioni specifiche ai metodi definiti dalla strategia. Offre un metodo `setAlgorithm()` per impostare dinamicamente l'algoritmo (strategia) da utilizzare.

Il contesto (**DivConqContext**) è indipendente dai dettagli della strategia specifica, consentendo la selezione dinamica dell'algoritmo.

**DivConqStrategy:** Interfaccia astratta che definisce i metodi (astratti) necessari per implementare la logica specifica di un problema:

Le sottoclassi concrete, come `QuickSort`, implementano i metodi definiti in `DivConqStrategy`. Queste strategie specificano come risolvere problemi particolari, mantenendo separata la logica generale.

```

public final class DivConqContext
{
    public DivConqContext (DivConqStrategy dc)
    {
        this.dc = dc;
    }
    public Solution solve (Problem p)
    {
        Problem[] pp;
        if (dc.isSimple(p)) { return dc.simplySolve(p); }
        else { pp = dc.decompose(p); }
        Solution[] ss = new Solution[pp.length];
        for (int i = 0; i < pp.length; i++)
        {
            ss[i] = solve(pp[i]);
        }
        return dc.combine(p, ss);
    }
    public void setAlgorithm (DivConqStrategy dc)
    {
        this.dc = dc;
    }
    private DivConqStrategy dc;
}

```

Fig. 8. Strategy context class implementation.

```

abstract public class DivConqStrategy
{
    abstract public boolean isSimple (Problem p);
    abstract public Solution simplySolve (Problem p);
    abstract public Problem[] decompose (Problem p);
    abstract public Solution combine(Problem p, Solution[] ss);
}

```

Fig. 9. Strategy object abstract class.

Usando il template method, il coupling (accoppiamento) tra il client e l'algoritmo scelto è **statico** e determinato in fase di compilazione tramite ereditarietà.

Usando lo strategy pattern, il contesto può cambiare dinamicamente la strategia utilizzata tramite **dependency injection** (es. metodo `setAlgorithm()`). Il coupling tra il client e l'algoritmo scelto è **dinamico** e può essere modificato a runtime.

## Framework development by generalization

## Livello 4: Analisi di una software family

L'analisi di una sw family consiste nell'**analizzare una potenziale famiglia di software**, identificando gli **aspetti comuni** (frozen spots), ovvero le parti invariabili condivise e gli **aspetti variabili** (hot spots), ovvero le parti modificabili per adattarsi a esigenze specifiche. Si valutano le possibili architetture alternative per il framework. Il design di un framework è un processo **incrementale**, che evolve gradualmente il progetto invece di essere scoperto in un unico passaggio.

Il processo di evoluzione del framework include **esaminare i design esistenti** per i membri della famiglia; **identificare frozen e hot spots** della famiglia, distinguendo tra le parti fisse e quelle variabili; **generalizzare la struttura del programma** per riutilizzare il codice delle parti comuni (frozen spots) e supportare implementazioni diverse per le parti variabili (hot spots).

## Binary Tree and Preorder Traversal

Si presenta un esempio in cui si parte da un algoritmo concreto per stampare un albero con un **traversal in preorder**, e si evolve verso una struttura più generalizzata che consenta la riusabilità e la personalizzazione.

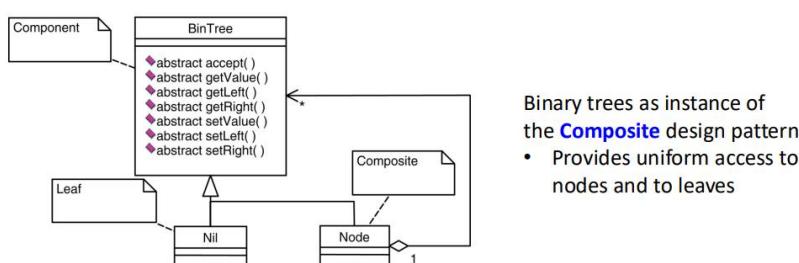


Fig. 10. Binary tree using Composite design pattern.

```
procedure preorder(t)
{   if t null, then return;
    perform visit action for root node of tree t;
    preorder(left subtree of t);
    preorder(right subtree of t);
}
```

Pseudo-code of generic **depth-first preorder left-to-right** traversal (**action** not specified)

**Struttura dell'albero binario:** La classe **BinTree** rappresenta l'albero, con sottoclassi per **Node** (nodo dell'albero) e **Nil** (foglia vuota). Questo è un esempio di utilizzo del *Composite design pattern*, dove si può trattare un singolo nodo o una composizione di nodi nello stesso modo.

**Visita dell'albero:** Il traversal è una tecnica per "visitare" tutti i nodi di un albero. Ad esempio, un traversal in **preorder** visita il nodo corrente prima di esplorare i suoi figli.

```
abstract public class BinTree
{
    public void setValue(Object v) { }           // mutators
    public void setLeft(BinTree l) { }             // default
    public void setRight(BinTree r) { }
    abstract public void preorder();               // traversal
    public Object getValue() { return null; }      // accessors
    public BinTree getLeft() { return null; }       // default
    public BinTree getRight() { return null; }
}

public class Node extends BinTree
{
    public Node(Object v, BinTree l, BinTree r)
    {
        value = v; left = l; right = r;
    }
    public void setValue(Object v) { value = v; } // mutators
    public void setLeft(BinTree l) { left = l; }
    public void setRight(BinTree r) { right = r; }
    public void preorder() {                     // traversal
        System.out.println("Visit node with value: " + value);
        left.preorder(); right.preorder();
    }
    public Object getValue() { return value; }     // accessors
    public BinTree getLeft() { return left; }
    public BinTree getRight() { return right; }
    private Object value;                         // instance data
    private BinTree left, right;
}

public class Nil extends BinTree
{
    private Nil() { } // private to require use of getNil()
    public void preorder() { };                   // traversal
    static public BinTree getNil() { return theNil; } // Singleton
    static public BinTree theNil = new Nil();
}
```

## Identifying Frozen and Hot Spots

Vediamo come identificare i **Frozen Spots** e i **Hot Spots** in un framework per algoritmi di traversamento degli alberi binari, generalizzando un programma concreto per creare una famiglia di algoritmi.

### Frozen Spots (Aspetti Fissi)

Sono caratteristiche invariabili condivise da tutti gli algoritmi della famiglia:

**Struttura dell'albero:** Definita dalla gerarchia BinTree, include nodi e foglie.

**Accesso agli elementi:** Un algoritmo di traversamento visita ogni elemento dell'albero almeno una volta, anche se può interrompersi prima di completare la visita.

**Esecuzione delle azioni di visita:** Durante il traversamento, viene eseguita una o più azioni su ciascun elemento visitato.

### Hot Spots (Aspetti Variabili)

Rappresentano le parti che cambiano tra gli algoritmi della famiglia:

**Azione di visita:** L'azione che viene eseguita quando si visita un nodo può variare. Ad esempio, può essere stampare il valore del nodo, oppure aggiornare uno stato accumulato.

**Ordine di visita:** L'ordine in cui i nodi vengono visitati può cambiare (ad esempio, *preorder*, *postorder* o *in-order*).

**Tecnica di navigazione dell'albero:** L'ordine di accesso ai nodi può variare. Ad esempio, può essere da sinistra a destra, oppure depth-first.

### Hot Spot #1: Generalizzazione dell'azione di visita

L'azione di visita è resa generica tramite un **metodo astratto visitPre**, definito nell'interfaccia PreorderStrategy. Questo approccio permette di applicare **diverse azioni di visita** sullo stesso albero binario, mantenendo il traversal separato dalla logica specifica dell'azione.

```
public interface PreorderStrategy
{
    abstract public Object visitPre(Object ts, BinTree t); }

abstract public class BinTree
{
    ...
    abstract public Object preorder(Object ts, PreorderStrategy v);
    ...
}

public class Node extends BinTree
{
    ...
    public Object preorder(Object ts, PreorderStrategy v) //traversal
    {
        ts = v.visitPre(ts, this);
        ts = left.preorder(ts, v);
        ts = right.preorder(ts, v);
        return ts;
    }
    ...
}

public class Nil extends BinTree
{
    ...
    public Object preorder(Object ts, PreorderStrategy v)
    {
        return ts;
    }
    ...
}
```

New BinTree hi

The **preorder** m  
the action from  
and handles ac

**Exercise:** define  
printing the val  
nodes, and for  
sum / max of al

**Interfaccia PreorderStrategy:** Definisce il metodo astratto visitPre(Object ts, BinTree t), che rappresenta l'azione da eseguire su un nodo dell'albero.

**Classe astratta BinTree:** Contiene il metodo astratto preorder(Object ts, PreorderStrategy v), che implementa il traversal in preorder utilizzando la strategia fornita.

**Classi concrete (Node e Nil):** Implementa il metodo preorder visitando il nodo corrente tramite visitPre, poi ricorsivamente attraversa i sottoalberi sinistro e destro.

### Hot Spot #2: Generalizing the visit order

L'**Hot Spot #2** riguarda la **generalizzazione dell'ordine di visita** in un traversamento di un albero binario.

### Generalizzazione dell'ordine di visita

Viene introdotta l'interfaccia **EulerStrategy**, che permette di visitare ogni nodo dell'albero **tre volte**:

**Left:** All'ingresso del nodo (preorder).

**Bottom:** Dopo aver visitato il sottoalbero sinistro (in-order).

**Right:** Dopo aver visitato il sottoalbero destro (postorder).

**visitNil:** Metodo specifico per gestire nodi nulli o foglie vuote.

Usando la EulerStrategy possiamo implementare qualsiasi combinazioni di pre-order, post-order or in-order traversal.

```
public interface EulerStrategy
{
    abstract public Object visitLeft(Object ts, BinTree t);
    abstract public Object visitBottom(Object ts, BinTree t);
    abstract public Object visitRight(Object ts, BinTree t);
    abstract public Object visitNil(Object ts, BinTree t);
}

abstract public class BinTree
{
    ...
    abstract public Object traverse(Object ts, EulerStrategy v);
    ...
}

public class Node extends BinTree
{
    ...
    public Object traverse(Object ts, EulerStrategy v) // traversal
    {
        ts = v.visitLeft(ts, this);           // upon arrival from above
        ts = left.traverse(ts, v);
        ts = v.visitBottom(ts, this);         // upon return from left
        ts = right.traverse(ts, v);
        ts = v.visitRight(ts, this);         // upon completion
        return ts;
    }
    ...
}

public class Nil extends BinTree
{
    ...
    public Object traverse(Object ts, EulerStrategy v)
    {
        return v.visitNil(ts, this);
    }
}
```

we gener  
subsystem  
• The Et  
three i  
bottom

**Interfaccia EulerStrategy:** Definisce i metodi astratti visitLeft, visitBottom, visitRight e visitNil.

**Classe astratta BinTree:** Contiene il metodo astratto traverse(Object ts, EulerStrategy v), che implementa il traversal utilizzando l'ordine definito dalla strategia.

### 1. Classi concrete (Node e Nil):

Implementa il metodo traverse seguendo il traversal Euleriano:

Esegue visitLeft all'ingresso del nodo. Visita ricorsivamente il sottoalbero sinistro.

Esegue visitBottom dopo il sottoalbero sinistro. Visita ricorsivamente il sottoalbero destro. Esegue visitRight al completamento del nodo. Aggiorna e restituisce l'accumulatore ad ogni passo.

**Risultati:** Il metodo preorder viene generalizzato in **traverse**, che ora supporta qualsiasi combinazione di traversal (preorder, postorder, in-order, o altri). La strategia EulerStrategy consente di definire e personalizzare il comportamento dell'ordine di visita senza modificare la struttura dell'albero.

## Hot Spot #3: Generalizing the tree navigation

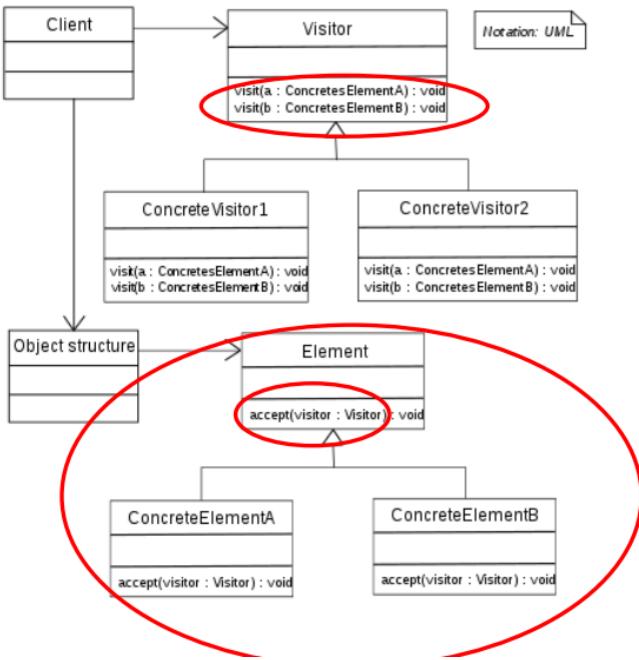
L'*hot spot* #3 richiede la personalizzazione della **navigazione dell'albero** per supportare diversi ordini di attraversamento dell'albero, come l'attraversamento in ampiezza e l'attraversamento in profondità.

**Supporto per diversi tipi di traversal:** Traversal in ampiezza (**breadth-first**). Traversal in profondità (**depth-first**). Traversal in ordine da sinistra a destra o da destra a sinistra. Traversal parziale o personalizzato.

**Frozen Spots** (aspetti invariabili): **Struttura dell'albero:** Definita dalla gerarchia BinTree, non modificabile. **Accesso agli elementi:** Ogni elemento dell'albero è visitato almeno una volta, con possibilità di interrompere la visita prima di completarla.

## Uso del Visitor Design Pattern

Invece di generalizzare ulteriormente il metodo traverse, si utilizza il **Visitor Design Pattern**, che consente di definire nuovi comportamenti per la navigazione dell'albero senza modificare la struttura dei dati (BinTree) e separa chiaramente l'algoritmo dalla struttura dell'albero.



**Il Visitor Design Pattern** permette di separare la logica di elaborazione (algoritmo) dalla struttura dei dati, rendendo più semplice l'aggiunta di nuovi comportamenti senza modificare le classi della struttura esistente.

**Element:** Classe base astratta o interfaccia per i componenti della struttura dati. Definisce il metodo **accept(Visitor visitor)**, che consente a un oggetto Visitor di "visitare" l'elemento.

**Concrete Elements:** Classi concrete che rappresentano i vari componenti della struttura dati. Implementano il metodo `accept`, chiamando il metodo appropriato del Visitor (`visit`).

**Visitor:** Interfaccia che definisce i metodi `visit` per ciascun tipo di elemento concreto (es. `visit(ConcreteElementA)`, `visit(ConcreteElementB)`). Contiene la logica di elaborazione per ogni tipo di elemento.

**Concrete Visitors:** Implementazioni del Visitor che forniscono comportamenti specifici (es. calcolo, stampa, validazione).

**Client:** Coordina l'uso del Visitor, passando gli elementi della struttura al Visitor per l'elaborazione tramite il metodo `accept`.

**Funzionamento** Ogni componente della struttura dati (elemento concreto) chiama il metodo `visit` corrispondente sul Visitor tramite il metodo `accept`. La logica di elaborazione è contenuta nel Visitor, mentre la struttura dati rimane separata.

**Esempio:** Una struttura dati composta da diversi tipi di nodi (es. `ConcreteElementA` e `ConcreteElementB`) può essere attraversata da Visitor come `ConcreteVisitor1` per calcolare una somma o `ConcreteVisitor2` per stampare i valori.

### Hot Spot #3: Binary Tree Visitor framework

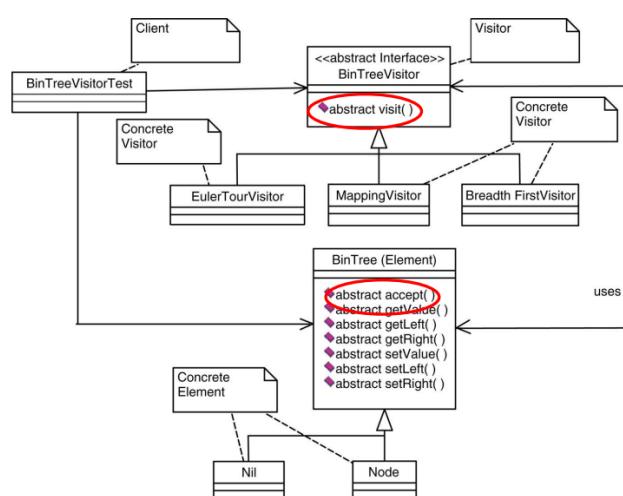


Fig. 14. Binary tree Visitor framework.

La gerarchia **BinTree** assume il ruolo di "Element" nella descrizione del Visitor pattern. Introduciamo un'interfaccia **BinTreeVisitor**, che svolge il ruolo di "Visitor" e implementa i metodi necessari per visitare i nodi dell'albero. Il metodo **traverse()** della gerarchia **BinTree** viene sostituito da un metodo **accept()**. Questo metodo riceve un oggetto **BinTreeVisitor** e delega il lavoro del traversal al metodo appropriato dell'oggetto visitor.

```

public interface BinTreeVisitor
{
    abstract void visit(Node t);
    abstract void visitNil(t);
}

abstract public class BinTree
{
    public void setValue(Object v) {}           // mutators
    public void setLeft(BinTree l) {}            // default
    public void setRight(BinTree r) {}          

    abstract public void accept(BinTreeVisitor v); // accept Visitor
    public Object getValue() { return null; } // accessors
    public BinTree getLeft() { return null; } // default
    public BinTree getRight() { return null; }
}

public class Node extends BinTree
{
    public Node(Object v, BinTree l, BinTree r)
    {
        value = v; left = l; right = r;
    }

    public void setValue(Object v) { value = v; } // mutators
    public void setLeft(BinTree l) { left = l; }
    public void setRight(BinTree r) { right = r; }

    // accept a Visitor object
    public void accept(BinTreeVisitor v) { v.visit(this); }

    public Object getValue() { return value; } // accessors
    public BinTree getLeft() { return left; }
    public BinTree getRight() { return right; }

    private Object value; // instance data
    private BinTree left, right;
}

public class Nil extends BinTree
{
    private Nil() {} // private to require use of getNil()
    // accept a Visitor object
    public void accept(BinTreeVisitor v) { v.visit(this); }

    static public BinTree getNil() { return theNil; } // Singleton
    static public BinTree theNil = new Nil();
}

```

The **BinTree** code is almost unchanged, only the **traverse** method is changed to

- **accept** an instance of **Visitor**
- invoke **visit(this)** on it

27

## Framework basato sul Visitor Design Pattern

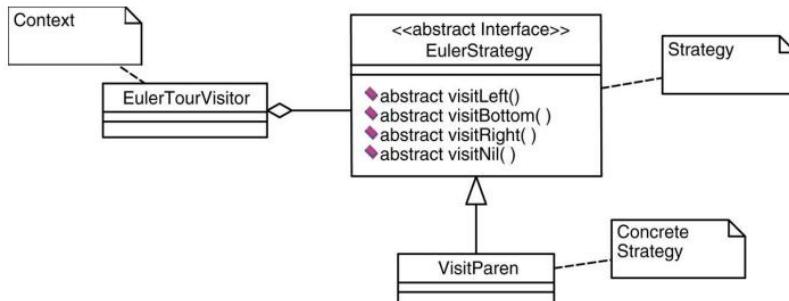


Fig. 16. Euler tour traversal Visitor framework

Si definisce una classe concreta **EulerTourVisitor**, che implementa l'interfaccia **BinTreeVisitor**.

**Classe concreta EulerTourVisitor:** Implementa l'interfaccia **BinTreeVisitor**. Le azioni specifiche di visita sono delegate a un oggetto di tipo **EulerStrategy**, che è iniettato tramite il costruttore. Questo approccio permette di cambiare dinamicamente la strategia utilizzata.

**Metodo visit(Node t):** È il metodo principale che implementa il traversal di un nodo: Chiama **visitLeft** al primo ingresso nel nodo. Passa ricorsivamente al sottoalbero sinistro tramite il metodo **accept** sul nodo sinistro. Chiama **visitBottom** dopo il completamento del sottoalbero sinistro. Passa ricorsivamente al sottoalbero destro tramite il metodo **accept** sul nodo destro. Chiama **visitRight** alla fine del nodo. Aggiorna lo stato del traversal tramite l'accumulatore **ts**.

**Interfaccia EulerStrategy:** Definisce i metodi astratti per le diverse fasi del traversal:  
**visitLeft:** Azione eseguita all'ingresso di un nodo. **visitBottom:** Azione eseguita dopo il ritorno dal sottoalbero sinistro. **visitRight:** Azione eseguita al completamento del nodo. **visitNil:** Azione eseguita quando si incontra un nodo vuoto o foglia.

La logica del traversal è completamente gestita da EulerTourVisitor, mentre le azioni specifiche sono delegate all'oggetto EulerStrategy. L'oggetto EulerStrategy può essere sostituito dinamicamente, consentendo di implementare diversi comportamenti di visita senza modificare la logica del traversal.

## Visitor for Euler Traversal using Strategy

```
public interface EulerStrategy
{
    abstract public Object visitLeft(Object ts, BinTree t);
    abstract public Object visitBottom(Object ts, BinTree t);
    abstract public Object visitRight(Object ts, BinTree t);
    abstract public Object visitNil(Object ts, BinTree t);
}
```

```
public class EulerTourVisitor implements BinTreeVisitor
{
    public EulerTourVisitor(EulerStrategy es, Object ts)
    {
        this.es = es; this.ts = ts;
    }
    public void setVisitStrategy(EulerStrategy es) // mutators
    {
        this.es = es;
    }
    public void setResult(Object r) { ts = r; }
    public void visit(Node t) // Visitor hook implementations
    {
        ts = es.visitLeft(ts,t); // upon first arrival from above
        t.getLeft().accept(this);
        ts = es.visitBottom(ts,t); // upon return from left
        t.getRight().accept(this);
        ts = es.visitRight(ts,t); // upon completion of this node
    }
    public void visitNil(Node t) { ts = es.visitNil(ts,t); }
    public Object getResult(){ return ts; } // accessor
    private EulerStrategy es; // encapsulates state changing ops
    private Object ts; // traversal state
}
```

- The navigation logic is in the `visit()` method
- It exploits `accept()` to pass to the next node
- The concrete actions are defined in an object implementing `EulerStrategy`
- The strategy is injected with the constructor and can be changed dynamically.

29

## Comparing tree traversal with and without visitor object

```
public class Node extends BinTree
{
    ...
    public Object traverse(Object ts, EulerStrategy v) // traversal
    {
        ts = v.visitLeft(ts,this); // upon arrival from above
        ts = left.traverse(ts,v);
        ts = v.visitBottom(ts,this); // upon return from left
        ts = right.traverse(ts,v);
        ts = v.visitRight(ts,this); // upon completion
        return ts;
    }
    ...
}

public class EulerTourVisitor implements BinTreeVisitor
{
    public EulerTourVisitor(EulerStrategy es, Object ts)
    {
        this.es = es; this.ts = ts;
    }
    public void setVisitStrategy(EulerStrategy es) // mutators
    {
        this.es = es;
    }
    public void setResult(Object r) { ts = r; }
    public void visit(Node t) // Visitor hook implementations
    {
        ts = es.visitLeft(ts,t); // upon first arrival from above
        t.getLeft().accept(this);
        ts = es.visitBottom(ts,t); // upon return from left
        t.getRight().accept(this);
        ts = es.visitRight(ts,t); // upon completion of this node
    }
    public void visitNil(Node t) { ts = es.visitNil(ts,t); }
    public Object getResult(){ return ts; } // accessor
    private EulerStrategy es; // encapsulates state changing ops
    private Object ts; // traversal state
}
```

Depth-first, left-to-right traversal starts with

`root.traverse(acc, es)`

Traversal starts with

`root.accept(eulerTVistor) -> eulerTourVisitor.visit(root)`

30

**Traversal implementato direttamente:** Nella classe `Node`, il metodo `traverse()` implementa un **traversal in profondità** (depth-first), da sinistra a destra, utilizzando direttamente una strategia definita dall'oggetto `EulerStrategy`. Questo approccio usa un traversal direttamente all'interno della classe `Node` e passa da un nodo all'altro chiamando ricorsivamente il metodo `traverse()` sui sottoalberi sinistro e destro.

**Traversal implementato con visitor:** Nella classe `EulerTourVisitor`, viene utilizzato il **Visitor pattern**, che separa la logica di navigazione dall'azione specifica da eseguire su ogni nodo. In questo approccio, la logica del traversal è separata dall'azione di visita, e le operazioni specifiche su ciascun nodo sono delegate all'oggetto `Visitor`. Si utilizza `accept()` per iniziare il traversal, e l'oggetto `Visitor` gestisce il traversal dei nodi.

# AP-11: Polymorphism

Il **polimorfismo** si riferisce alla capacità di una funzione di **assumere diverse forme**, ovvero di comportarsi in modi diversi a seconda del contesto.

Una **funzione** è considerata **polimorfica** se può essere chiamata con argomenti di diverso tipo e comportarsi in **modo appropriato** per ciascun tipo.

Anche i tipi di dati stessi possono essere polimorfici. I **tipi generici** (List<T> in Java) permettono di definire strutture di dati che funzionano con una varietà di tipi.

**Polimorfismo ad hoc (Overloading):** Lo stesso nome di funzione può rappresentare **algoritmi diversi**, e l'algoritmo specifico che verrà utilizzato è determinato dai **tipi reali** degli argomenti forniti.

```
// Metodo per sommare due interi
public int add(int a, int b) {
    return a + b;
}

// Metodo per concatenare due stringhe
public String add(String a, String b) {
    return a + b;
}
```

**Polimorfismo universale:** esiste **un solo algoritmo** che è in grado di operare su **oggetti di diversi tipi**. La stessa implementazione della funzione può essere applicata a più tipi di dati. Si divide in due sottotipi principali:

**Polimorfismo parametrico:** Permette di scrivere funzioni **generiche** che operano su parametri di tipo.

```
// Polimorfismo parametrico: metodo generico per invertire una lista di qualsiasi tipo
public static <T> List<T> reverseList(List<T> list) {
    Collections.reverse(list);
    return list;
}
```

**Polimorfismo di sottotipo:** È possibile usare un oggetto di una **sottoclasse** dove viene richiesto un oggetto della classe base.

```
// Metodo che accetta la classe base ma usa il metodo specifico delle sottoclassi
public static void animalSpeak(Animal animal) {
    System.out.println(animal.speak());
}
```

## Binding Time

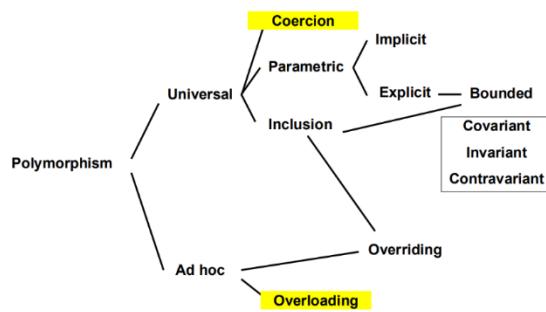
Il **Binding time** è il momento in cui viene stabilito il collegamento tra un nome di funzione e il codice effettivo che deve essere eseguito.

**Binding a Compile Time (early/static binding):** Il binding avviene **durante la compilazione**, durante la quale il compilatore decide esattamente quale codice eseguire quando si richiama una funzione.

**Binding a Linking Time** Il binding avviene **durante il linking** del programma, quando diversi moduli compilati separatamente vengono combinati in un unico eseguibile, associando i riferimenti delle funzioni con le loro implementazioni reali.

**Binding a Execution Time (late/dynamic binding):** Il binding avviene **durante l'esecuzione** del programma. Nei linguaggi orientati agli oggetti, quando si utilizza l'ereditarietà e il polimorfismo, il metodo specifico da invocare potrebbe essere determinato solo a runtime (tempo di esecuzione).

Se il binding si estende su più fasi, si considera il **tempo di binding come l'ultima fase** in cui il collegamento viene stabilito. **Prima è il binding, meglio è per il debugging**, perché molti errori vengono individuati in anticipo.



## Overloading (Polimorfismo ad hoc)

**Overloading:** Capacità di utilizzare lo stesso nome per più funzioni o operatori, ma con implementazioni diverse a seconda del tipo di argomenti forniti.

L'overloading è presente in quasi tutti i linguaggi di programmazione per gli operatori aritmetici integrati, come +, \*, -, ecc., che si comportano diversamente a seconda del tipo degli operandi. Ad esempio, + può essere usato per sommare numeri interi o concatenare stringhe, a seconda dei tipi coinvolti.

Alcuni linguaggi permettono di definire più versioni di una stessa funzione con lo stesso nome, ma con diversi tipi o numeri di parametri.

Altri linguaggi consentono anche di sovraccaricare operatori primitivi (come +, \*) attraverso funzioni definite dall'utente.

Nei linguaggi tipizzati staticamente, come C++ e Java, il binding avviene a compile time e il compilatore sa in anticipo quale versione della funzione sovraccaricata eseguire in base ai tipi degli argomenti.

Nei linguaggi tipizzati dinamicamente come **Python**, il tipo degli argomenti viene determinato a runtime, quindi la scelta della funzione appropriata avviene solo a quel momento.

- Function for squaring a number:

```
sqr(x) { return x * x; }
```

- Typed version (like in C):

```
int sqr(int x) { return x * x; }
```

- Multiple versions for different types (C):

```
int sqrInt(int x) { return x * x; }
double sqrDouble(double x) { return x * x; }
```

- Overloading (Java, C++):

```
int sqr(int x) { return x * x; }
double sqr(double x) { return x * x; }
```

In **Haskell**, l'overloading viene gestito in modo elegante attraverso il concetto di **type classes**. Un tipo che implementa un'interfaccia typeclass può sovraccaricare le sue funzioni. Qualsiasi tipo istanzia quella typeclass, userà la sua implementazione.

Esempio: Qualsiasi tipo che istanzia Num (Int, Float, Double) può utilizzare questi operatori, e Haskell sa quale versione di ogni funzione utilizzare a seconda del tipo.

## Coercion (Polimorfismo Universale)

**Coercion:** Conversione **automatica**, tramite la quale quando si passa un valore a una funzione che si aspetta un tipo diverso, il compilatore può convertire il valore automaticamente.

```
int a = 5;
double d = sqrt(a); // a è un int, ma viene convertito in double
```

**Casting:** Conversione di tipo **esplicita**, indicata dal programmatore.

```
int a = 5;
double d = sqrt((double) a); // Casting esplicito da int a double
```

La coercion è considerata una forma **degenerata di polimorfismo** perché non richiede la definizione esplicita di comportamenti polimorfici, ma è una semplice operazione di linguaggio.

Nei linguaggi ben progettati, la coercion è **limitata** e viene applicata solo quando **non c'è perdita di informazione**.

Java consente la coercion solo quando la conversione è sicura e non porta alla perdita di dati (ad esempio, convertire un int in un double). C++ consente la coercion più liberamente, ma questo può portare a situazioni problematiche.

## Inclusion polymorphism

Il polimorfismo di inclusione è garantito dal **Principio di Sostituzione di Liskov**: **Un oggetto di un sottotipo può essere usato ovunque sia richiesto un oggetto del supertipo**, senza che il comportamento atteso del programma venga alterato.

Se le sottoclassi non ridefiniscono un metodo della superclasse, viene utilizzata la versione ereditata. Se invece lo ridefiniscono, viene chiamata la nuova versione specifica della sottoclasse.

## Overriding

**Overriding** si riferisce alla capacità di una sottoclasse di **ridefinire** un metodo della sua superclasse.

La nuova implementazione del metodo nella sottoclasse sostituisce la versione ereditata dalla superclasse quando viene invocato nell'istanza della sottoclasse.

```
class A {  
    public void m() {  
        // stampa "A"  
    }  
}  
  
class B extends A {  
    public void m() {  
        // stampa "B"  
    }  
}
```

L'**Overriding** introduce il **polimorfismo ad hoc** nel **polimorfismo universale** dell'ereditarietà. Questo perché permette alla stessa chiamata (a.m()) di comportarsi in modo diverso a seconda dell'oggetto effettivo (A o B) su cui viene invocata.

In **Java**, il **binding dinamico** viene gestito dal meccanismo di invokevirtual della JVM. Quando si chiama un metodo su un oggetto, la JVM esegue una ricerca (lookup) per determinare quale versione del metodo chiamare in base all'oggetto reale, non solo al tipo della variabile.

In **C++**, l'overriding è gestito attraverso un meccanismo chiamato **dynamic method dispatch**, che utilizza una tabella virtuale **v-table**.

Ogni classe che contiene metodi virtuali ha una v-table, che è una struttura dati che contiene puntatori ai metodi virtuali della classe.

Quando si chiama un metodo virtuale su un oggetto, il programma utilizza la v-table per determinare quale versione del metodo eseguire, permettendo il **binding dinamico** a runtime.

Un problema può sorgere quando si combinano overloading e overriding. Se una sottoclasse ridefinisce un metodo della superclasse, e la superclasse ha metodi sovraccaricati con lo stesso nome, il compilatore può non vedere correttamente tutti i metodi.

## Overloading + Overriding: C++ vs Java

```
class A {  
public:  
    virtual void onFoo() {}  
    virtual void onFoo(int i) {}  
};  
  
class B : public A {  
public:  
    virtual void onFoo(int i) {}  
};  
  
class C : public B {  
};  
  
int main()  
{  
    C* c = new C();  
    c->onFoo();  
    //Compile error -  
    // doesn't exist  
}
```

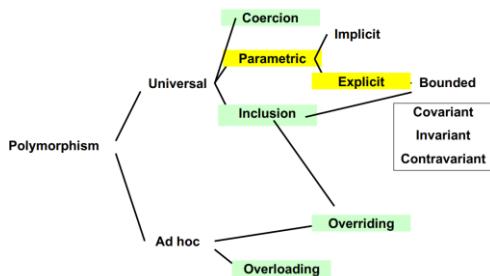
```
class A {  
public:  
    public void onFoo() {}  
    public void onFoo(int i) {}  
}  
  
class B extends A {  
public:  
    public void onFoo(int i) {}  
}  
  
class C extends B {  
};  
  
class D {  
public static void main(String[] s)  
{  
    C c = new C();  
    c.onFoo();  
} //Compiles !!  
}
```

16

Per risolvere questo problema, si può usare la direttiva **using** per rendere visibile la versione sovraccaricata della superclasse. `using A::onFoo;`

## AP-12: Parametric Polymorphisms: C++ Templates

### Classification of Polymorphism



## Polimorfismo Parametrico o Programmazione Generica

Il **polimorfismo parametrico** (o *programmazione generica*) permette di creare funzioni e classi che **funkzionano con diversi tipi di dati** senza **dover riscrivere il codice** per ciascun tipo specifico.

### C++: Templates (dal 1990)

I **Templates** in C++ consentono di scrivere funzioni e classi generiche che possono accettare qualsiasi tipo di dato, utilizzando **variabili di tipo** (T, U, ecc.).

**Monomorfizzazione:** Durante la compilazione, ogni volta che un template viene usato con un tipo specifico, il compilatore crea una **copia specializzata del codice** per quel tipo. Se si usa `vector<int>` e `vector<double>`, il compilatore genera due versioni separate del template `vector<T>`.

### Java: Generics

I Java Generics, in modo simile ai template C++, permettono di definire metodi e classi che operano con variabili di tipo (T, E, K, V, ecc.).

**Type Checking Forte:** Il compilatore controlla i tipi durante la compilazione, assicurandosi che i tipi siano utilizzati correttamente.

**Type Erasure:** S differenza di C++, le informazioni sui tipi generici vengono **rimosse** e sostituite col tipo **Object** dopo la compilazione.

### Function Templates in C++

I **templates** di funzione permettono di creare una singola definizione di funzione che può essere usata con vari tipi di dati (anche tipi primitivi).

`template <class T> // o <typename T>`

`T sgr(T x) { return x * x; }`

In molti casi, il compilatore è in grado di **dedurre automaticamente** il tipo (T) dai parametri passati alla funzione. Tuttavia, se c'è ambiguità, è possibile **specificare il tipo esplicitamente**.

```
auto result2 = sgr(3.2); // Dedotto come `double`  
auto result3 = sgr<int>(3); // Specificato esplicitamente come `int`
```

```
template<class T> // or <typename T>  
T sqr(T x) { return x * x; }  
  
int a = 3;  
double b = 3.14;  
int aa = sqr(a);  
double bb = sqr(b); // also sqr<double>(b)  
  
Generates  
int sqr(int x){return x*x;}  
  
Generates  
double sqr(double x){return x*x;}
```

```
class Complex {  
public:  
    double real; double imag;  
    Complex(double r, double im): real(r),imag(im){};  
    Complex operator*(Complex y) { //overloading of *  
        return Complex(  
            real * y.real - imag * y.imag,  
            real * y.imag + imag * y.real);  
    };  
  
{ ...  
    Complex c(2, 2);  
    Complex cc = sqr(c);  
    cout << cc.real << " " << cc.imag << endl;  
... }
```

```
template <class T>  
T GetMax (T a, T b) {  
    T result;  
    result = (a>b)? a : b;  
    return (result);  
}  
  
{ ...  
    int i = 5, j = 6, k;  
    long l = 10, m = 5, n, v;  
    k = GetMax<int>(i, j); //ok  
    n = GetMax(l, m); //ok: GetMax<long>  
// v = GetMax(i, m); //no: ambiguous  
    v = GetMax<int>(i,m); //ok  
... }
```

- Decoupling the two arguments:

```
template <class T, class U>  
T GetMax (T a, U b) {  
    return (a>b)? a : b;  
}
```

## Macro in C++

Una macro è semplicemente una **sostituzione di testo** effettuata dal processore che avviene prima della compilazione. Ad esempio:

```
#define SQR(x) ((x) * (x))  
int a = 2;  
int aa = SQR(a++); // int aa = ((a++) * (a++)); //=6 X
```

Le macro possono portare side effects indesiderati. Nell'esempio precedente, l'incremento `a++` viene duplicato, quindi `aa` diventa 12 invece di 9.

Il preprocessore non esegue controlli sui tipi, il che può portare a errori difficili da individuare. Non essendo gestite dal compilatore, le macro non possono essere ottimizzate né analizzate in modo statico, causando comportamenti non previsti. Le macro non possono essere utilizzate per creare funzioni ricorsive, poiché non supportano la valutazione dinamica durante il runtime.

```
#define FACT(n) (n == 0 ? 1: FACT(n - 1) * n) // Non funziona, causa un errore di compilazione
```

## Non-Type Template Arguments in C++

Oltre a parametri di tipo (class T), i template possono accettare anche **parametri di valore**. Questi devono essere **costanti** e di tipo specifico, come interi (int), puntatori o riferimenti. Quando si istanzia il template, bisogna specificare sia il tipo che il valore passato.

```

template <class T, int N>
T fixed_multiply(T val) {
    return val * N;
}
std::cout << fixed_multiply<int, 2>(10) << '\n'; // Output: 20
std::cout << fixed_multiply<int, 3>(10) << '\n'; // Output: 30

```

In questi casi, il compilatore genera versioni specializzate della funzione `fixed_multiply` per  $N = 2$  e  $N = 3$ . Il secondo parametro ( $N$ ) deve essere una **espressione costante**. Non si possono usare variabili che cambiano a runtime.

## Template Specialization in C++

La **specializzazione del template** consente di creare versioni specifiche di un template generico per gestire casi particolari.

Una **specializzazione completa** è una versione del template definita per uno specifico tipo. Viene creata una versione totalmente nuova del template, che sostituisce quella generica solo per i tipi specificati.

```

// Template generico
template <typename T>
class Box{
    // Implementazione generica
};

// Specializzazione completa per int
template <>
class Box<int>{
    // Implementazione specifica per int
};

```

Una **specializzazione parziale** consente di definire una versione del template che è **più specifica** del template generico, ma non è completamente specializzata. Si può specificare solo alcuni parametri, lasciando gli altri generici.

```

template <typename T, typename U>
class Container{
    // Implementazione generica
};

// Specializzazione parziale per T = int
template <typename U>
class Container<int, U>{
    // Implementazione specifica per Container<int, U>
};

```

### Need of template specialization, an example

<pre> // Full specialization of GetMax for char* template &lt;&gt; const char* GetMax(const char* a, const char* b) { return strcmp(a, b) &gt; 0 ? a : b; }  template &lt;class T&gt; T GetMax(T a, T b) { return a &gt; b ? a : b; }  int main() {     cout &lt;&lt; "max(10, 15) = " &lt;&lt; GetMax(10, 15) &lt;&lt; endl ;     cout &lt;&lt; "max('k', 's') = " &lt;&lt; GetMax('k', 's') &lt;&lt; endl ;     cout &lt;&lt; "max(10.1, 15.2) = " &lt;&lt; GetMax(10.1, 15.2) &lt;&lt; endl ;     cout &lt;&lt; "max(\"Al\", \"Bob\") = " &lt;&lt; GetMax("Al", "Bob") &lt;&lt; endl ;     return 0 ; } </pre>	<pre> Output: max(10, 15) = 15 max('k', 's') = s max(10.1, 15.2) = 15.2 max("Al", "Bob") = Al //not expected </pre>	<pre> Output of main with specialization: max(10, 15) = 15 max('k', 's') = s max(10.1, 15.2) = 15.2 max("Al", "Bob") = Bob </pre>
---	---	---

## Template Metaprogramming (TMP) in C++

La **Template Metaprogramming (TMP)** è una tecnica avanzata di programmazione in C++ che sfrutta i **template** per eseguire calcoli durante la **compilazione** del codice.

I **template** possono essere utilizzati per generare automaticamente **codice sorgente temporaneo** durante la compilazione. Questo codice viene poi integrato con il resto del sorgente e compilato.

Questo permette di eseguire calcoli complessi e generare strutture di dati senza overhead a runtime, poiché tutto è risolto prima dell'esecuzione del programma.

```
#include <iostream>

int triangular(int n) {
    return (n == 1)? 1 : triangular(n-1) + n;
}

int main () {
    int result = triangular(20);
    std::cout << result << '\n';
}
```

```
template <int t>
int triangular() {
// constexpr int triangular() {
    return triangular<t - 1>() + t;
}
template <>
int triangular<1>() {
// constexpr int triangular<1>() {
    return 1;
}
int main () {
    int result = triangular<20>();
    std::cout << result << '\n';
}
```

- C++ function to compute sum of first  $n$  integers
- C++ template with specialization computing the same
- **constexpr** “invites” the compiler to evaluate the expression

17

La TMP lavora solo con **espressioni costanti**. Tutti i calcoli e le valutazioni devono essere risolvibili a compile-time. Non si possono usare variabili mutabili che cambiano a runtime. Questo limita il tipo di operazioni possibili, ma garantisce che il risultato sia calcolato **prima dell'esecuzione**.

## Implementazione dei Template in C++

Il **codice di un template** non viene compilato finché il compilatore non vede un'**istanza** del template utilizzata nel programma. Quando viene trovata una **chiamata** a un template, il compilatore esegue un'**istanza del template** a compile-time, sostituendo i parametri generici con i tipi specifici forniti.

Il compilatore seleziona il template **migliore** basandosi sulla **corrispondenza** dei parametri di tipo.

Se ci sono più template che potrebbero corrispondere, il compilatore preferisce la specializzazione più specifica. Una volta scelto il template, il compilatore **crea un'istanza** del template. Dopo che i parametri del template sono stati sostituiti, il compilatore esegue la overloading resolution per determinare quale funzione o operatore chiamare. Se il tipo specifico fornito non ha un operatore definito che il template richiede, la compilazione fallirà. Tutto questo avviene a compile time.

In C/C++, è comune **dichiarare** funzioni e classi in un file di intestazione (.h) e **definire** le implementazioni in un file di implementazione (.cpp).

Tuttavia, per **istanziare** un template, il compilatore ha bisogno sia della dichiarazione che della **definizione completa** del template, poiché il compilatore deve **generare codice** per l'istanza di template durante la compilazione.

Un modo per aggirare alcune limitazioni è attraverso l'**esplicita istanziazione** dei template. Si può forzare il compilatore a generare codice per un'istanza specifica di un template, senza dover includere la definizione completa nel file di intestazione.

```
// In un file di intestazione (GetMax.h)
```

```
template <class T>
```

```
T GetMax(T a, T b);
```

```
// In un file di implementazione (GetMax.cpp)
```

```
template <class T>
```

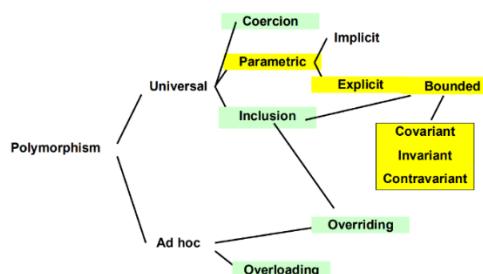
```
T GetMax(T a, T b){
```

```
    return a > b ? a : b;
```

```
}
```

```
// Forzare un'istanziazione esplicita:  
template int GetMax<int>(int a, int b);
```

## AP-13: Java Generics



I **Generics in Java** permettono di creare classi, interfacce e metodi che funzionano con parametri di tipo, consentendo il polimorfismo parametrico. Nella definizione di una classe o metodo generico, i parametri di tipo possono essere utilizzati come se fossero normali tipi di dato.

```
interface List<E> {  
    boolean add(E n);  
    E get(int index);  
}  
  
List<Integer>  
List<Number>  
List<String>  
List<List<String>>  
...
```

Quando si crea un'istanza di una classe generica, come `List<Integer>`, si specifica che il parametro di tipo `E` sarà sostituito dal tipo `Integer`. Questo permette di creare liste che contengono solo numeri interi, oppure liste di qualsiasi altro tipo specificato.

I **metodi generici** in Java sono metodi che possono utilizzare parametri di tipo. Se un metodo è definito all'interno di una classe generica, può utilizzare i parametri di tipo definiti nella classe stessa. Inoltre, i metodi generici possono anche introdurre propri parametri di tipo, indipendentemente dai parametri definiti nella classe.

Quando invii una chiamata a un metodo generico, tutti i parametri di tipo devono essere istanziati, cioè devi specificare il tipo effettivo da usare per quei parametri, esplicitamente o tramite all'inferenza dei tipi del compilatore, che si basa sui valori passati al metodo.

### Bounded Type Parameters

I **Bounded Type Parameters** permettono di vincolare i tipi che possono essere utilizzati come argomenti di tipo per una classe o un metodo generico.

```
class NumList<E extends Number> {  
    void m(E arg) {  
        arg.intValue(); // OK, Number and its  
                        // subtypes support intValue()  
    }  
}
```

La sintassi generale per definire un **upper bound** è: `<T extends SuperType>`.

T rappresenta il parametro di tipo e SuperType è il **tipo limite**. Solo SuperType o le sue sottoclassi possono essere utilizzate come argomento di tipo.

Se vuoi utilizzare più limiti, puoi farlo con: `<T extends ClassA & InterfaceB & InterfaceC>`

In questo caso, T deve essere una sottoclasse di ClassA e implementare le interfacce InterfaceB e InterfaceC. Java permette di avere solo una classe come limite superiore, ma puoi specificare più interfacce.

Questi *type bounds* servono per garantire che l'argomento di tipo supporti le operazioni che userai nel corpo del metodo. Se specifichi un metodo che utilizza il parametro di tipo T, puoi essere sicuro che quel tipo avrà accesso ai metodi definiti nella classe o interfacce specificate come limiti.

In Java il controllo dei tipi avviene in fase di compilazione, che garantisce che qualsiasi overloading sia risolto con successo durante il controllo dei tipi, evitando errori in fase di esecuzione legati all'uso improprio dei tipi generici.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
    for (T e : anArray)  
        if (e > elem) // compiler error  
            ++count;  
    return count;  
}
```

```
public interface Comparable<T> { // classes implementing  
    public int compareTo(T o); // Comparable provide a  
} // default way to compare their objects
```

```
public static <T extends Comparable<T>>  
int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
    for (T e : anArray)  
        if (e.compareTo(elem) > 0) // ok, it compiles  
            ++count;  
    return count;  
}
```

Un aspetto cruciale dell'introduzione dei generics è stato il mantenimento della compatibilità con il codice precedente alla sua introduzione. Questo è stato possibile grazie al **type erasure**, che consiste nel **cancellare** i tipi generici durante la compilazione e sostituirli con i loro **tipi limite** (o **Object**).

Sebbene in Java un tipo specifico possa essere un sottotipo di un altro, questo non si traduce automaticamente per le versioni generiche di quei tipi. Per fare esempio, Integer è una sottoclasse di Number, tuttavia questo non significa che List<Integer> sia un sottotipo di List<Number>.

**Regola sottotipizzazione invariante:** Date due classi concrete A e B, se A è un sottotipo di B, non significa che MyClass<A> sia un sottotipo di MyClass<B>.

Per risolvere questo problema di mancanza di sottotipizzazione diretta tra classi generiche, Java introduce il concetto di **wildcards** (caratteri jolly). Il tipo **MyClass<?>** rappresenta una classe generica con un parametro di tipo sconosciuto e agisce come genitore comune per tutte le varianti di MyClass.

D'altra parte, se hai delle classi generiche che estendono altre classi generiche, la sottotipizzazione si comporta come previsto. Ad esempio, se A estende B e hai un tipo C, allora A<C> è un sottotipo di B<C>. Un caso comune di questo è che ArrayList<Integer> è un sottotipo di List<Integer>, poiché ArrayList è una sottoclasse di List.

```
interface WoList<T> {  
    boolean add(T elt);  
}  
  
type WoList<Number>;  
    boolean add(Number elt);  
  
type WoList<Integer>;  
    boolean add(Integer elt);
```

The diagram illustrates the inheritance relationship between two generic types. On the left, there is a box containing the code for the **WoList<T>** interface, which defines a single method **add(T elt)**. To its right, another box contains the code for the **WoList<Number>** type, also defining the **add** method with the same signature. An arrow points from the **WoList<T>** box up to the **WoList<Number>** box, indicating that **WoList<Number>** is a subtype of **WoList<T>**. A green checkmark is placed at the junction of the arrow and the **WoList<Number>** box, signifying that this inheritance is valid according to Java's wildcards rules.

Java non supporta nativamente la covarianza nei generics, quindi il comportamento dell'esempio precedente non è consentito per impostazione predefinita senza usare wildcard (? extends).

La **contravarianza** funziona al contrario della covarianza. In un contesto contravariante, una WoList<Number> può essere considerata un sottotipo di WoList<Integer>, poiché possiamo aggiungere un oggetto di tipo Integer in una lista destinata a contenere elementi di tipo Number, ma non possiamo leggere correttamente da essa senza rischiare errori di tipo.

Tuttavia, Java non supporta nativamente la contravarianza nei generics, quindi questo comportamento non è possibile per impostazione predefinita senza meccanismi come <? Super> per gestire il tipo.

## Array e generics

In Java, gli *array* sono trattati come contenitori built-in, e hanno una proprietà importante: sono **covarianti**. Questo significa che, se Type1 è un sottotipo di Type2, allora Type1[] è un sottotipo di Type2[].

Questo permette, ad esempio, di trattare un array di Integer[] come un array di Number[].

```
Number[] numArray = new Integer[10];
numArray[0] = 3.14; // Errore a runtime: ArrayStoreException
```

Per evitare di rompere la sicurezza del tipo, ogni aggiornamento di un array in Java comporta un controllo a runtime. Se si tenta di inserire un oggetto di un tipo non compatibile con l'array sottostante, la JVM lancia un'eccezione chiamata ArrayStoreException.

## Recalling "Type erasure"

A runtime, a causa del type erasure, una volta che il codice generico è stato compilato, tutte le istanze dello stesso tipo generico sono trattate come se avessero lo stesso tipo, indipendentemente dal parametro di tipo che avevano in fase di compilazione. Questo porta al fatto che, anche se crei due liste generiche con tipi diversi, come List<String> e List<Integer>, entrambe saranno viste come semplici ArrayList<Object> a runtime.

```
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass() // true
```

Non è possibile creare array di tipi generici, perché potrebbe portare a errori di tipo non rilevabili a runtime, compromettendo la sicurezza del tipo in Java.

## Wildcards for covariance

In Java, i **wildcards** (caratteri jolly) rappresentano un tipo anonimo, indicato con il simbolo ?, che viene utilizzato quando il tipo preciso non è noto o non è importante specificarlo. I wildcards permettono di gestire la covarianza e la contravarianza nei generics, fornendo flessibilità quando si vuole lavorare con tipi generici senza doverli specificare esattamente.

A differenza della dichiarazione dei tipi generici che avviene nel sito di dichiarazione del metodo, i wildcards sono usati nel sito di utilizzo, dove si vuole indicare una variazione nei tipi accettabili senza specificarli nel dettaglio.

```

interface Set<E> {
    // Adds to this all elements of c
    // (not already in this)
    void addAll(??? c);
}

```

- void **addAll**(Set<E> c) // and List<E>?
- void **addAll**(Collection<E> c)
 // and collections of T <: E?
- void **addAll**(Collection<? extends E> c); // ok

**<? extends SuperType>**: Questo indica che il tipo deve essere un sottotipo di SuperType.

**<?>**: Scorciatoia per **<? extends Object>**, che denota qualsiasi tipo possibile.

**<? super SubType>**: Indica che è possibile utilizzare un supertipo di SubType.

Si utilizza **<? extends T>** quando vuoi estrarre valori da un contenitore, ma non hai bisogno di inserire nuovi valori, supportando la covarianza, perché permette di trattare contenitori di sottotipi come contenitori del tipo base per leggere valori in modo sicuro.

```

public static void printNumbers(List<? extends Number> numbers) {
    for (Number number : numbers) { // Puoi estrarre come Number
        System.out.println(number);
    }
    // Non puoi aggiungere valori, perché il tipo esatto è incerto.
    // numbers.add(1); // Errore!
}

public static void main(String[] args) {
    List<Integer> intList = List.of(1, 2, 3);
    List<Double> doubleList = List.of(1.1, 2.2, 3.3);

    printNumbers(intList); // Funziona con Integer (sottoclasse di Number)
    printNumbers(doubleList); // Funziona con Double (sottoclasse di Number)
}

```

Si usa **<? super T>** quando vuoi inserire valori in un contenitore, ma non hai bisogno di estrarlo come un tipo specifico, supportando la controvarianza, perché permette di trattare contenitori di supertipi come contenitori del tipo base per scrivere valori in modo sicuro.

```

public static void addNumbers(List<? super Integer> list) {
    list.add(1); // Puoi aggiungere valori di tipo Integer o suoi sottotipi.
    list.add(2);
    // Non puoi estrarre in modo sicuro come un tipo più specifico.
    // Integer number = list.get(0); // Errore!
}

public static void main(String[] args) {
    List<Number> numberList = new ArrayList<>();
    List<Object> objectList = new ArrayList<>();

    addNumbers(numberList); // Funziona con Number (superclasse di Integer)
    addNumbers(objectList); // Funziona con Object (superclasse di Integer)

    System.out.println(numberList); // [1, 2]
    System.out.println(objectList); // [1, 2]
}

```

Non si usa nessun tipo di wildcard ?, ma si utilizza il tipo preciso T quando vuoi sia estrarre che inserire valori.

- A wildcard type is anonymous/unknown, and almost nothing can be done:

```
List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples; // covariance
fruits.add(new Strawberry()); // compile-time error! OK
Fruits f = fruits.get(0); // OK
fruits.add(new Apple()); // compile-time error???
fruits.add(null); //ok, the only thing you can add ☺
```

```
List<Fruit> fruits = new ArrayList<Fruits>();
List<? super Apples> apples = fruits; // contravariance
apples.add(new Apple()); // OK
apples.add(new FujiApple()); // OK
apples.add(new Fruit()); // compile-time error, OK
Fruits f = apples.get(0); // compile-time error???
Object o = apples.get(0); //ok, the only way to get
```

## Limitations of Java Generics

I generics funzionano solo con **tipi di riferimento** e non con tipi primitivi (int, double, ecc.).

Non è possibile creare un'istanza direttamente del parametro di tipo generico, poiché il parametro di tipo viene cancellato durante la compilazione. Questo accade perché, a runtime, Java non sa più qual è il tipo effettivo di T.

```
T obj = new T(); // Non compila
```

I campi statici non possono essere parametrici perché i parametri di tipo sono associati alle istanze della classe e non alla classe stessa.

```
public class C<T> {
    public static T local; // Non compila
}
```

Non è possibile verificare con instanceof se un oggetto è di un tipo parametrico specifico, poiché le informazioni sui tipi vengono rimosse con il *Type Erasure*.

Java non consente di creare array di tipi generici, poiché anche qui il *Type Erasure* impedisce di sapere qual è il tipo effettivo a runtime. List<String>[] arrayOfLists = new List<String>[10]; // Non compila

Non è possibile creare eccezioni o oggetti parametrizzati con i generics, né catturarli o lanciarli: throw new T(); // Non compila

Non è possibile sovraccaricare metodi che differiscono solo per il tipo parametrico, poiché entrambi i metodi si cancellano allo stesso tipo grezzo dopo il *Type Erasure*:

```
public class Example {
    public void print(Set<String> strSet) {}
    public void print(Set<Integer> intSet) {} // Non compila
}
```

## AP-14: C++ Standard Template Library

La Standard Template Library (STL) di C++ è una libreria che ha come obiettivo rappresentare gli algoritmi nella forma più generale possibile senza sacrificare l'efficienza. Utilizza intensivamente i template e l'overloading di funzioni, e si basa esclusivamente sul binding statico.

La STL usa gli iteratori per separare gli algoritmi dai contenitori, trattandoli come astrazioni dei puntatori. Integra molte astrazioni generiche, come tipi e operazioni astratte polimorfiche.

Le principali entità della STL (Standard Template Library) sono:

**Container:** Strutture dati generiche come vector, list, set, map, ecc., progettate per memorizzare e organizzare dati in modo efficiente.

**Iterator:** Oggetti simili a puntatori che consentono di navigare e manipolare gli elementi all'interno dei contenitori.

**Algorithm:** Funzioni generiche per eseguire operazioni come ordinamento, ricerca, copia e trasformazione degli elementi nei contenitori.

**Adaptor:** Interfacce che trasformano contenitori, iteratori o funzioni in versioni con funzionalità modificate o limitate (es. stack come adattatore per deque).

**Function object:** Oggetti che agiscono come funzioni attraverso la definizione di operator(), spesso utilizzati per personalizzare algoritmi con comportamenti specifici.

**Allocator:** Strumenti per la gestione personalizzata della memoria, come allocazione dinamica e gestione efficiente dei buffer.

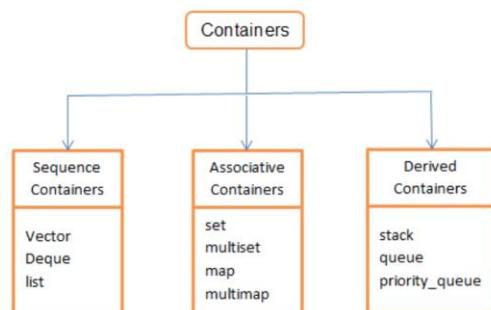
## Contenitori

I **contenitori** sono divisi in tre categorie principali:

**Sequence Containers:** Contengono dati in sequenza e includono strutture come vector, deque, e list. Vengono utilizzati quando l'ordine degli elementi è importante.

**Associative Containers:** Organizzano i dati in modo che possano essere cercati velocemente, come set, multiset, map, e multimap. Questi contenitori mantengono gli elementi ordinati automaticamente e sono ottimali per le ricerche rapide.

**Derived Containers:** Sono contenitori derivati da altri contenitori per fornire funzionalità specifiche. Esempi sono stack, queue, e priority\_queue, che sono implementati utilizzando altri contenitori come le liste o i deque, ma con comportamenti specifici (es. LIFO o FIFO).



Gli **algoritmi** della STL lavorano su questi contenitori e sono progettati per essere indipendenti dai tipi di dati e dai contenitori specifici. Questo è possibile grazie ai template, che permettono agli algoritmi di essere indipendenti dai tipi di dati, e grazie agli iteratori, che permettono agli algoritmi di essere applicati a contenitori diversi senza dipendere dalla loro implementazione interna.

## Iteratori in Java

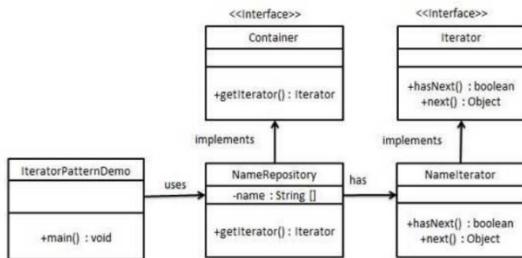
Gli **iteratori** risolvono il problema degli algoritmi di non potere essere utilizzati direttamente su tipi diversi di collezione, fornendo un accesso uniforme e lineare agli elementi delle diverse collezioni. In Java, un iteratore su una collezione offre i seguenti due metodi.

**boolean hasNext():** Verifica se ci sono altri elementi nella collezione.

**T next():** Restituisce il prossimo elemento della collezione.

Gli **iteratori** in Java sono supportati attraverso l'interfaccia Iterator<T>, che sfrutta i **generics** per lavorare con oggetti di qualsiasi tipo, come fanno le collezioni stesse. Gli iteratori vengono definiti in genere come **classi membro private non statiche**.

all'interno delle classi delle collezioni. Ogni istanza di un iteratore è associata a un'istanza della classe della collezione, permettendo l'accesso ai suoi elementi in maniera ordinata.



Le collezioni che sono equipaggiate con iteratori devono implementare l'interfaccia **Iterable<T>**, che fornisce il metodo `getIterator()`, consentendo di ottenere un iteratore per quella collezione.

```

for (Iterator<Integer> it = myBinTree.iterator(); it.hasNext(); ) {
    Integer i = it.next();
    System.out.println(i);
}
  
```

Il ciclo **foreach** sfrutta gli iteratori per semplificare la sintassi del ciclo. Per utilizzare il ciclo `for` migliorato, `myBinTree` deve essere un array di interi oppure deve implementare l'interfaccia `Iterable<Integer>`.

```

for (Integer i : myBinTree) {
    System.out.println(i);
}
  
```

Nel caso degli array l'iterazione si basa sulla lunghezza fissa dell'array. Per un iteratore arbitrario, il comportamento dipende da come l'iteratore è implementato.

## Iteratori in C++

Gli **iteratori** in Java e C++ sono concettualmente simili, ma sfruttano le caratteristiche dei rispettivi linguaggi in modo diverso.

In Java, vengono utilizzati metodi come `next()` e `hasNext()` per scorrere gli elementi, mentre in C++ si fa uso di operatori predefiniti, come l'operatore di **dereferenziazione (\*)** e l'operatore di **incremento (++)**, per muoversi attraverso gli elementi di un contenitore, in modo del tutto simile all'utilizzo dei puntatori.

```

#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec; // create a vector to store int
    // push 5 values into the vector
    for(int i = 0; i < 5; i++) {
        vec.push_back(i);
    }
    for(int i = 0; i < 5; i++) { // access to elements
        cout << "vec [" << i << "] = " << vec[i] << endl;
    }
    // use iterator to access the values
    vector<int>::iterator v = vec.begin();
    while( v != vec.end()) {
        cout << "value of v = " << *v << endl;
        v++;
    }
    return 0;
}
  
```

La STL sfrutta i **namespace** per organizzare i tipi e le funzioni in modo da evitare conflitti di nomi. Ogni contenitore nella STL definisce un tipo denominato **iterator** all'interno del proprio namespace, fornendo così un modo specifico per iterare sugli elementi di quel contenitore. Ad esempio, il tipo di iteratore per un std::vector di stringhe viene definito come std::vector<std::string>::iterator, dove std è il namespace standard e vector è il contenitore.

Ogni classe in C++ introduce implicitamente un nuovo namespace, il che significa che all'interno di quel contesto, il nome iterator assume il significato appropriato per quel tipo di contenitore. Il tipo di iterator cambia in base al contesto in cui viene utilizzato.

### Implementazione iteratori in C++

Gli **iteratori** vengono implementati direttamente dai contenitori nella STL e di solito sono realizzati come **struct**. Un iteratore mantiene informazioni interne sullo **stato della visita** del contenitore.

```
template <class T>
struct v_iterator {
    T *v;
    int sz;
    v_iterator(T* v, int sz) : v(v), sz(sz) {}
    // != implicitly defined
    bool operator==(v_iterator& p) { return v == p->v; }
    T operator*() { return *v; }
    v_iterator& operator++() { // Pre-increment
        if (sz) ++v, --sz; else v = NULL;
        return *this;
    }
    v_iterator operator++(int) { // Post-increment!
        v_iterator ret = *this;
        ++(*this); // call pre-increment
        return ret;
    }
};
```

Nel precedente esempio, per un array di tipo T, v\_iterator memorizza un **puntatore T \*v** all'**elemento corrente** dell'array e un intero **sz** che indica la **dimensione rimanente** dell'array. Le operazioni principali implementate sono:

**operator==**: Confronta se due iteratori puntano allo stesso elemento.

**operator\***: Dereferenzia il puntatore, restituendo il valore dell'elemento attualmente visitato.

**operator++ (pre-incremento)**: Se rimangono slot da visitare, avanza l'iteratore al prossimo elemento e decrementa sz.

**operator++(int) (post-incremento)**: Crea una copia temporanea dell'iteratore, avanza l'iteratore attuale e restituisce la copia precedente (per supportare il post-incremento).

### Complessità delle operazioni sui contenitori

Viene garantito che le operazioni di inserimento e cancellazione alla **fine di un vettore** richiedano un tempo costante ammortizzato. Ciò significa che, anche se alcune operazioni possono occasionalmente richiedere più tempo (ad esempio quando il vettore deve essere ridimensionato), in media il costo per operazione rimane costante.

D'altra parte, le operazioni di inserimento e cancellazione nel **mezzo di un vettore** richiedono **tempo lineare**, poiché è necessario spostare tutti gli elementi successivi per fare spazio o per riordinare la sequenza.

Container	insert/erase overhead at the beginning	in the middle	at the end
Vector	linear	linear	amortized constant
List	constant	constant	constant
Deque	amortized constant	linear	amortized constant

La **complessità nell'uso degli iteratori** dipende dal tipo di iteratore che un algoritmo richiede e dal tipo di contenitore su cui sta operando.

```
std::list<std::string> l;  
quick_sort(l.begin(), l.end());
```

Nell'esempio precedente, viene utilizzata una lista concatenata e si tenta di applicare l'algoritmo quick\_sort. Questo non è ragionevole perché quick\_sort assume che l'algoritmo abbia accesso **casuale** agli elementi del contenitore, cioè deve poter accedere rapidamente a qualsiasi elemento del contenitore in tempo costante o logaritmico.

Per controllare la complessità degli algoritmi e garantire che il codice si comporti come previsto, è essenziale abbinare l'**algoritmo** corretto con il **tipo di iteratore** giusto. Alcuni algoritmi richiedono iteratori con **accesso casuale** (come i vettori o i deque), mentre altri funzionano con **accesso sequenziale** (come le liste concatenate).

La **STL** classifica gli **iteratori** assumendo che tutte le operazioni sugli iteratori vengano eseguite in **tempo costante**. I vari contenitori possono supportare tipi diversi di iteratori a seconda della loro struttura, consentendo diversi livelli di accesso agli elementi.

**Forward iterators:** Consentono di **dereferenziare** gli elementi con operator\* e di **avanzare** tramite gli operatori di pre/post-incremento (operator++). Sono utilizzati per attraversare i contenitori in avanti in modo sequenziale.

**Input e Output iterators:** Simili ai forward iterators, ma specifici per **operazioni di input** e output. A causa della natura delle operazioni I/O, possono avere restrizioni o comportamenti meno prevedibili nella dereferenziazione.

**Bidirectional iterators:** Estendono le funzionalità dei forward iterators aggiungendo il supporto per gli operatori di **pre/post-decremento** (operator--), permettendo di attraversare i contenitori sia in avanti che indietro.

**Random access iterators:** Includono tutte le capacità dei bidirectional iterators e aggiungono supporto per operazioni aritmetiche con indici. È possibile **sommare o sottrarre interi** ( $p + n$ ,  $p - q$ ) per accedere direttamente a elementi specifici, come accade con gli array.

I **forward iterators** consentono di **dereferenziare** gli elementi con operator\* e di **avanzare** tramite gli operatori di pre/post-incremento (operator++). Sono utilizzati per attraversare i contenitori in avanti in modo sequenziale.

Gli **input iterators** e **output iterators** (cin/cout) sono specifici per operazioni di IO. Ereditano le caratteristiche dei forward iterators, ma impongono i seguenti vincoli: Un input o output iterator **non può essere salvato** e riutilizzato in seguito per riprendere l'iterazione dalla posizione precedente. Non è garantito che si possa **assegnare** un valore all'oggetto ottenuto dereferenziando (\*) un input iterator. Non è possibile **leggere** l'oggetto ottenuto dereferenziando un output iterator. Non è garantito che si possa **confrontare** due output iterators per uguaglianza o disuguaglianza, poiché gli operatori == e != potrebbero non essere definiti.

**Bidirectional iterators:** Estendono le funzionalità dei forward iterators aggiungendo il supporto per gli operatori di **pre/post-decremento** (operator--), permettendo di attraversare i contenitori sia in avanti che indietro.

**Random access iterators:** Includono tutte le capacità dei bidirectional iterators e aggiungono supporto per operazioni aritmetiche con indici. È possibile **sommare o sottrarre interi** ( $p + n$ ,  $p - q$ ) per accedere direttamente a elementi specifici, come accade con gli array.

Gli operatori + e - consentono di sommare o sottrarre un intero da un iteratore per ottenere un nuovo iteratore che punta a una posizione diversa nella sequenza.

Gli operatori di confronto <, >, <=, >= permettono di confrontare due iteratori per determinare il loro ordine relativo all'interno della sequenza.

Gli iteratori in C++ si basano sull'overloading **degli operatori**, consentendo di utilizzare una sintassi simile ai puntatori, per lavorare con diversi tipi di contenitori.

Ogni categoria include solo quelle funzioni che possono essere realizzate in **tempo costante**, un aspetto cruciale per l'efficienza nella STL. Tuttavia, ogni contenitore è abbinato alla propria categoria di iteratori.

<i>Container</i>	<i>Iterator Category</i>
vector	random access iterators
list	bidirectional iterators
deque	random access iterators

Quando un contenitore viene modificato, ad esempio con operazioni di inserimento o cancellazione, gli iteratori che puntano agli elementi del contenitore **possono** diventare **invalidi**. Dopo operazioni come queste, è necessario ottenere nuovi iteratori validi dal contenitore modificato.

<i>Container</i>	<i>operation</i>	<i>iterator validity</i>
vector	inserting	reallocation necessary - all iterators get invalid
	erasing	no reallocation - all iterators before insert point remain valid
list	inserting	all iterators after erasee point get invalid
	erasing	only iterators to erased elements get invalid
deque	inserting	all iterators get invalid
	erasing	all iterators get invalid

Gli **iteratori** in C++ consentono di scorrere gli elementi in una sequenza uno dopo l'altro. Questo modello è sufficiente per definire algoritmi che operano su contenitori **monodimensionali**. Se è necessario accedere a una struttura più complessa, come un **albero** o un **grafo**, dove l'ordine degli elementi non è lineare e può richiedere percorsi personalizzati, il modello standard degli iteratori non è sufficiente. In questi casi, è necessario **definire un nuovo tipo di iteratore** che gestisca la specifica organizzazione del contenitore.

## An algorithm of STL: inner product

```
#include <iostream>
#include <numeric>

int main() {
    int A1[] = {1, 2, 3};
    int A2[] = {4, 1, 2};
    const int N1 = sizeof(A1) / sizeof(A1[0]);

    std::cout << std::inner_product(A1, A1 + N1, A2, 0)
        << std::endl;
    return 0;
}
```

It will print 12:  

$$0 = 0 + 1 * 4 + 2 * 1 + 3 * 2$$

Initial value  
for the  
accumulator

Start of A1      End of A1      Start of A2

int std::inner\_product<int\*, int\*, int>(int\*, int\*, int\*, int)

Normalmente, l'algoritmo `inner_product` utilizza gli operatori `*` per moltiplicare e `+` per sommare gli elementi, ma nel caso delle stringhe, questi operatori non sarebbero utili.

L'idea è sfruttare l'algoritmo `inner_product` per **concatenare** le stringhe delle etichette e dei valori, separandole con una tabulazione (`\t`) al posto di `*` e aggiungendo una nuova riga (`\n`) al posto di `+` tra una coppia e l'altra. Tuttavia, sovraccaricare gli operatori `*` e `+` per gestire queste operazioni avrebbe poco senso, poiché potremmo interferire con i sovraccarichi già definiti per le stringhe.

Esiste una versione di `inner_product` che permette di specificare **funzioni oggetto** al posto degli operatori predefiniti `*` e `+`, che ci consente di definire le operazioni personalizzate di concatenazione con separatori (tabulazione e nuova riga) senza dover sovraccaricare gli operatori in modo inappropriate.

```

std::cout << inner_product(A1, A1 + N1, A2, 0)
      << std::endl;

std::cout <<
    inner_product(A1, A1 + N1, A2,
                  std::string("") , CatS(std::string("\n")),
                  CatS(std::string("\t")) ) << std::endl;

```

## Inheritance? No thanks!

La **STL** evita l'uso dell'**ereditarietà** per implementare la genericità. Al contrario, si affida ai **typedefs** combinati con i **namespace** per definire i tipi, come `container::iterator`, che permette al programmatore di conoscere il tipo di iteratore associato a un contenitore specifico.

Non esiste alcuna **relazione** tra gli iteratori di contenitori diversi, e questo è fatto intenzionalmente per motivi di **prestazioni**. Evitando l'ereditarietà, i tipi vengono risolti **a tempo di compilazione**, il che permette al compilatore di generare codice più efficiente.

**Sacrificare l'ereditarietà** migliora le prestazioni ma riduce l'**espressività** del codice e limita il **controllo dei tipi** a tempo di compilazione. Ad esempio, se si usa un iteratore sbagliato, il compilatore potrebbe generare errori difficili da interpretare, che sembrano provenire dalla libreria stessa.

## Inlining

L'**Inlining** è una tecnica in cui una chiamata di metodo viene **controllata a livello di tipo** dal compilatore e poi **sostituita** direttamente con il corpo del metodo stesso, eliminando così l'overhead di una chiamata di funzione.

I **metodi inline** devono essere definiti nei file header .h e possono essere etichettati come inline oppure definiti direttamente all'interno della definizione della classe. Tuttavia, l'inlining non viene applicato in ogni caso: il compilatore decide di solito di eseguire l'inlining solo per metodi con **corpi piccoli** e senza iterazioni, poiché l'inlining su metodi complessi o lunghi potrebbe peggiorare le prestazioni, anziché migliorarle.

Uno dei vantaggi principali del C++ è che il compilatore è in grado di **determinare i tipi** a tempo di compilazione, e di solito applica l'inlining automaticamente, specialmente quando si usano **function objects** (oggetti funzione). Questa capacità permette alla STL di ottenere efficienza, riducendo il costo delle chiamate di funzione nelle operazioni sugli algoritmi e sui contenitori.

La gestione della memoria nella STL è astratta grazie a un concetto chiamato **allocators**. Gli allocators encapsulano tutte le informazioni relative al **modello di memoria**, permettendo alla STL di gestire la memoria in modo flessibile e indipendente dal modello specifico utilizzato.

Ogni contenitore della STL è parametrizzato da un **allocator**, il che significa che il modo in cui viene gestita la memoria può essere modificato senza cambiare l'implementazione del contenitore stesso. La gestione della memoria può essere personalizzata in base alle esigenze dell'applicazione.

Un esempio tipico è la dichiarazione della classe `vector`, dove il secondo argomento template è un allocator predefinito:

```

template <class T,
template <class U> class Allocator = allocator>
class vector{
    // implementazione del vector
};

```

Se l'utente non specifica un allocator diverso, il contenitore utilizza il "**allocator**" predefinito della STL, che implementa le sue strategie di gestione della memoria. Tuttavia, è possibile specificare un allocator personalizzato se si desidera gestire la memoria in modo diverso.

Il principale problema con la STL riguarda il **controllo degli errori**. La maggior parte degli strumenti di controllo del compilatore fallisce nel gestire correttamente errori legati alla STL, producendo messaggi di errore molto lunghi e complessi che spesso finiscono con riferimenti a errori **all'interno della libreria stessa**.

Un altro problema è il cosiddetto **code bloat**, ovvero l'aumento delle dimensioni del codice generato. La STL, e più in generale l'approccio generativo del compilatore C++ (che crea codice specializzato per ciascun uso dei template), può portare alla generazione di grandi quantità di codice duplicato.

Il **code bloat** diventa problematico se il set di lavoro di un processo (ovvero la quantità di memoria necessaria per eseguire il programma) diventa troppo grande.

## AP-15: Functional Programming Introduction

### Learning Haskell

#### Haskell

- ❑ Quali sono le differenze tra le interfacce e le classi in Java, i traits in Rust e le typeclass in Haskell?
- ❑ L'IO Monad in Haskell permette di gestire correttamente gli effetti collaterali come input/output e il concetto di variabile. Perché? Come viene definita l'operazione bind? Quale ruolo gioca il "mondo" nel garantire la stretta sequenzialità?
- ❑ Qual è il meccanismo di parameter passing in Haskell? Continuare a ritardare l'esecuzione di una funzione può causare problemi di memoria?

L'idea centrale dei linguaggi funzionali è quella di costruire il programma **combinando funzioni** che prendono input e restituiscono output **senza alterare** lo stato del sistema. In questi linguaggi non esiste stato mutabile e non ci sono effetti collaterali: ogni funzione dipende solo dai suoi parametri e produce sempre lo stesso risultato per gli stessi input, garantendo un comportamento **prevedibile**.

Nei linguaggi funzionali le **funzioni** possono essere trattate come **valori**, il che significa che possono essere passate come argomenti ad altre funzioni, restituite come risultato di una funzione o memorizzate in variabili. Le funzioni non alterano variabili esterne e restituiscono semplicemente il risultato del calcolo.

Nella programmazione funzionale, la **ricorsione** sostituisce l'iterazione tradizionale, poiché non ci sono variabili di controllo. Le funzioni si richiamano da sole per iterare sui dati. Il polimorfismo consente alle funzioni e alle strutture di dati di operare su diversi tipi di dati. Sono ampiamente utilizzate strutture come tuple e record per organizzare i dati. Queste strutture non possono essere modificate dopo la creazione. Se i dati devono cambiare, è necessario creare una nuova struttura, mantenendo l'immutabilità dei dati.

In uno **scoping statico**, le strutture dati e le funzioni definite all'interno di una funzione possono sopravvivere anche dopo la terminazione di quest'ultima. Questo perché potrebbero essere ancora referenziate da altre parti del programma. Poiché lo stack è una memoria temporanea legata all'esecuzione della funzione, queste strutture non possono essere allocate lì; devono invece essere allocate nell'heap. La gestione della loro deallocazione richiede un garbage collector, che libera automaticamente la memoria quando non ci sono più riferimenti a queste strutture.

Haskell è un linguaggio di programmazione **puramente funzionale**.

Un **linguaggio puramente funzionale** è un linguaggio di programmazione in cui ogni operazione è espressa come una funzione matematica priva di **effetti collaterali**.

**Assenza di effetti collaterali:** Le funzioni non **alterano lo stato** del programma, non leggono né modificano **variabili globali**, e non interagiscono direttamente con **I/O**.

**Trasparenza referenziale:** Una funzione dato uno stesso input restituisce sempre lo stesso output.

**Immutabilità:** Non è possibile modificare **variabili esistenti**; ogni operazione **crea nuovi dati** (modificati).

**Composizione:** Le funzioni possono essere combinate facilmente per costruire operazioni più complesse.

## Laziness

Haskell è un linguaggio **lazy (pigro)**.

**Linguaggio lazy:** A meno che non sia richiesto esplicitamente, il linguaggio esegue le operazioni solo nel momento in cui è strettamente necessario. Chiamare una funzione non esegue immediatamente il calcolo ma rimanda l'operazione, registrandola come una trasformazione da applicare in futuro.

**Esempio.** doubleMe(doubleMe(doubleMe(doubleMe([1,2,3,4,5,6,7,8]))) in un linguaggio imperativo comporterebbe tre iterazioni complete attraverso la lista, con creazione di copie intermedie. In un linguaggio lazy richiede una sola iterazione; Quando il risultato viene richiesto, le trasformazioni vengono eseguite in cascata: La terza doubleMe calcola il primo elemento ( $1 * 2 = 2$ ). Il risultato (2) viene passato alla seconda doubleMe, che lo raddoppia a 4. Il risultato (4) viene passato alla prima doubleMe, che restituisce 8. Poi si passa al secondo elemento.

## Tipizzazione statica

Haskell è un linguaggio **tipizzato staticamente**. Durante la **compilazione**, il compilatore conosce i tipi di ogni parte del codice (numeri, stringhe, ecc.), rilevando gli errori di tipo **prima** dell'esecuzione.

Haskell utilizza un sistema di **inferenza di tipi**, grazie al quale il compilatore può dedurre il tipo di ogni elemento **senza dichiarazioni esplicite**. L'inferenza dei tipi rende il codice più generale: una funzione che somma due parametri funzionerà con qualsiasi tipo che si comporta come un numero, senza dover specificare il tipo.

Haskell supporta il polimorfismo parametrico隐式, permettendo alle funzioni di operare su diversi tipi.

## Tipi di base di Haskell

**Unit:** rappresenta un tipo che ha un solo valore vuoto, (), ed è simile a **void** in altri linguaggi.

**Booleans:** tipi booleani con i valori True e False.

**Integers:** numeri interi di dimensione arbitraria.

**Char:** Singoli caratteri, es. 'A'

**Reals:** numeri reali, per rappresentare valori decimali.

**Strings:** sequenze di caratteri, rappresentate come liste di Char.

**Tuples:** gruppi ordinati di dimensione fissa di valori eterogenei, es. (3, "hello").

**Lists:** collezioni omogenee di elementi di dimensione variabile, es. [1, 2, 3].

**Records:** strutture di dati con campi nominati, simili agli oggetti.

## Binding delle Variabili in Haskell

**Binding delle variabili:** Associazione di un **nome** (variabile) a **un'espressione**. Questo avviene in modo **lazy**, cioè il valore associato non viene calcolato finché non è necessario.

### Scope del Binding

Una volta definita una variabile, rimane valida per tutta la durata della sessione o dello scope in cui è stata dichiarata.

```
ghci> let a = 6  
-- a non viene valutato immediatamente  
ghci> let b = a + 2  
-- b non viene calcolato finché non richiesto  
ghci> b
```

8

Se proviamo a ridefinire a come  $a = a + 1$ , Haskell entra in un **loop infinito**, perché a si riferisce a se stesso senza una base per terminare, a causa della laziness.

## Aritmetica di base

```
haskell  
  
ghci> 2 + 15  
17  
ghci> 49 * 100  
4900  
ghci> 1892 - 1472  
420  
ghci> 5 / 2  
2.5
```

Copia codice

## Algebra booleana

Gli operatori booleani principali sono `&&` per *and* logico, `||` per *or* logico, `not` per negare un valore booleano.

## Operazioni di uguaglianza

Per confrontare i valori, usiamo `==` per verificare l'uguaglianza e `/=` per verificare la disuguaglianza.

Gli operatori come `+` funzionano solo con numeri, mentre `==` può confrontare qualunque cosa sia comparabile, purché i tipi siano identici.

## Funzioni in Haskell

In Haskell, quasi tutto è definito come una funzione. La chiamata delle funzioni segue una sintassi **prefissa**: Il nome della funzione è seguito dai parametri, separati da spazi.

```
haskell  
  
ghci> succ 8  
9  
ghci> min 9 10  
9  
ghci> max 100 101  
101
```

## Funzioni infisse

Se una funzione prende due parametri, può essere usata in forma **infissa** racchiudendola tra ````: *param1 `funcName` param2*

```
haskell  
  
ghci> 92 `div` 10  
9
```

## Definire funzione con un parametro: *nameFunc* *parametro* = espressione

```
haskell  
  
square x = x * x
```

## Definire funzione con più parametri: *nameFunc* *param1...paramN* = espressione

```
haskell  
  
add x y = x + y
```

## Funzione con condizionale (if):

**nameFunc** *param* = **if** *condizione*  
                 **then** *espressione1*  
                 **else** *espressione2*

Esempio:

```
haskell

max' x y = if x > y
             then x
             else y
```

**Nomi e definizioni immutabili:** Le funzioni senza parametri possono essere usate come definizioni o una **costante immutabile**. (conanO'Brien = "It's a-me, Conan O'Brien!")

## Le liste in Haskell

In Haskell, le **liste** sono una struttura dati **omogenea**, che contengono solo elementi dello stesso tipo.

```
haskell

ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

**Concatenare liste:** L'operatore `++` concatena due liste.

```
haskell

ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
```

**Append:** L'operatore `:` è utilizzato per appendere un elemento in cima ad una lista

```
haskell

ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

## Struttura delle liste

`[]` rappresenta una lista vuota. Una lista come `[1,2,3]` è solo una sintassi più semplice per `1:2:3:[ ]`. Se aggiungiamo elementi con `:`, otteniamo nuove liste.

## Accedere agli elementi per indice

Per accedere a un elemento di una lista, usiamo l'operatore `!!`. Gli indici partono da 0:

```
[-2,1,3,5]
ghci> v !! 0
1
ghci> |
```

## Liste di liste

Le liste possono contenere altre liste, che a loro volta possono contenere ulteriori liste:

```
ghci> let v = [1,3,5]
ghci> let b =[v, 3:v, v ++ [3]]
ghci> b
[[1,3,5],[3,1,3,5],[1,3,5,3]]
ghci> |
```

Tutte le liste all'interno di una lista devono contenere elementi dello stesso tipo.

## Comparazione di liste

Le liste possono essere confrontate se gli elementi possono essere confrontati. Il confronto avviene in ordine **lessicografico**.

```
ghci> [3,2,1] == [1,2,3]
False
ghci> |
```

## Funzioni di base per liste

**head:** restituisce il primo elemento della lista.

```

ghci> head [5,4,3,2,1]
5
ghci> ■
last: restituisce l'ultimo elemento.
ghci> last [5,4,3,2,1]
1
tail: restituisce tutti gli elementi tranne il primo.
ghci> tail [5,4,3,2,1]
[4,3,2,1]
init: restituisce tutti gli elementi tranne l'ultimo.
ghci> init [5,4,3,2,1]
[5,4,3,2]
length: restituisce la lunghezza di una lista.
ghci> length [5,4,3,2,1]
5
-
null: verifica se una lista è vuota.
ghci> null []
True
-
reverse: inverte la lista.
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
take: estraie i primi n elementi.
ghci> take 3 [5,4,3,2,1]
[5,4,3]
drop: elimina i primi n elementi.
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
maximum e minimum: restituiscono rispettivamente il massimo e il minimo.
ghci> maximum [1,9,2,3,4]
9
sum e product: calcolano rispettivamente la somma e il prodotto.
ghci> sum [5,2,1,6,3,2,5,7]
31
elem: verifica se un elemento è presente nella lista.
ghci> 4 `elem` [3,4,5,6]
True

```

## Creazione di liste con intervalli

I **range** sono un modo per creare liste che rappresentano sequenze aritmetiche di elementi enumerabili. La sintassi è **[elemStart..elemFinish]**

```

ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]

```

Possiamo specificare il numero di **passi** separando i primi due elementi con una virgola. I passi saranno dedotti dalla differenza di questi due numeri.

```

ghci> [3,6..21]
[3,6,9,12,15,18,21]

```

Non possiamo creare sequenze non aritmetiche e gli intervalli funzionano solo con un passo costante.

## Intervalli decrescenti

Per creare una lista decrescente, bisogna specificare i primi due valori:

```

ghci> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]

```

## Liste infinite

Non specificando un limite superiore, si possono creare liste infinite:

```

ghci> take 24 [13,26..]
[13,26,39,52,65,78,91,104,117,130,143,156,169,182,195,208,221,234,247,260,273,286,299,312]

```

Poiché Haskell è **lazy**, non calcola l'intera lista finché non è necessario, permettendo di estrarre solo i valori richiesti.

## Funzioni per liste infinite

**cycle**: ripete una lista all'infinito.

```

ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]

```

**repeat**: crea una lista infinita con un singolo elemento.

```
ghci> take 10 (repeat 5)
```

```
[5,5,5,5,5,5,5,5,5,5]
```

**replicate**: crea una lista con un elemento ripetuto un numero finito di volte.

```
ghci> replicate 3 10
```

```
[10,10,10]
```

## Compreensioni di liste

Le **list comprehensions** permettono di creare nuove liste applicando trasformazioni e filtri su liste esistenti.

```
ghci> [x*2 | x <- [1..10]]
```

```
[2,4,6,8,10,12,14,16,18,20]
```

Qui, x viene estratto da [1..10] e moltiplicato per 2.

### Aggiungere condizioni (predicati)

Possiamo filtrare i risultati aggiungendo predicati:

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
```

```
[12,14,16,18,20]
```

### Più predicati

Gli elementi devono soddisfare tutti i predicati per essere inclusi:

```
ghci> [x | x <- [10..20], x /= 13, x /= 15, x /= 19]
```

```
[10,11,12,14,16,17,18,20]
```

### Più liste

Quando si disegnano elementi da più liste, si generano tutte le combinazioni:

```
ghci> [x*y | x <- [2,5,10], y <- [8,10,11]]
```

```
[16,20,22,40,50,55,80,100,110]
```

Possiamo anche filtrare le combinazioni:

```
ghci> [x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
```

```
[55,80,100,110]
```

---

## Esempi pratici

### Generare combinazioni di aggettivi e sostantivi:

```
ghci> let nouns = ["hobo", "frog", "pope"]
```

```
ghci> let adjectives = ["lazy", "grouchy", "scheming"]
```

```
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
```

```
["lazy hobo", "lazy frog", "lazy pope", "grouchy hobo", "grouchy frog", "grouchy
```

```
pope", "scheming hobo", "scheming frog", "scheming pope"]
```

### Calcolare la lunghezza di una lista:

```
length' xs = sum [1 | _ <- xs]
```

### Filtrare lettere maiuscole da una stringa:

```
removeNonUppercase st = [c | c <- st, c `elem` ['A'..'Z']]
```

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
```

```
"HA"
```

### Compreensioni di liste nidificate:

```
ghci> let xxss = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
```

```
ghci> [[x | x <- xs, even x] | xs <- xxss]
```

```
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

---

## Tuple

Le **tuple** sono collezioni che hanno una **dimensione fissa**. Il loro tipo dipende dal numero e dai tipi dei componenti, che possono essere **eterogenei**.

I tuple sono denotati da **parentesi tonde** con i componenti separati da virgole:

(elem0...elemN-1)

Ad esempio, (1,2) o (1,"hello",True).

Le tuple sono utili quando sappiamo in anticipo quanti elementi una struttura deve contenere. Essendo più rigidi delle liste, garantiscono che i tipi e il numero di elementi siano coerenti.

## Operazioni sui tuple

**fst**: restituisce il primo elemento di una coppia.

```
ghci> fst (8,11)
```

8

**snd**: restituisce il secondo elemento di una coppia.

```
ghci> snd (8,11)
```

11

**Nota:** Queste funzioni funzionano solo con coppie e non con tuple di dimensioni maggiori.

**Zip:** La funzione zip combina due liste in una lista di coppie tuple:

```
ghci> zip [1,2,3] ["one", "two", "three"]
```

```
[(1,"one"),(2,"two"),(3,"three")]
```

Se le liste hanno lunghezze diverse, la lista più lunga viene troncata:

Possiamo combinare una lista finita con una infinita grazie alla laziness di Haskell.

## Types and Typeclasses

### Verificare i tipi con GHCI

Possiamo esaminare i tipi delle espressioni usando il comando `:t` in GHCI. Il simbolo `::` si legge come "**ha tipo**".

```
ghci> :t 'a'  
'a' :: Char  
ghci> :t True  
True :: Bool  
ghci> :t "Hello"  
"Hello" :: String  
ghci> :t ("Ciao",2)  
("Ciao",2) :: Num b => (String, b)  
ghci> █
```

I tipi esplicativi iniziano sempre con una lettera maiuscola. Il comando `:set +t` abilita il compilatore a stampare i tipi di dato ad ogni espressione.

### Tipi delle funzioni

Le funzioni hanno tipo. Quando scriviamo funzioni, possiamo scegliere di dichiarare esplicitamente il loro tipo.

```
removeNonUppercase :: [Char] -> [Char]  
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

removeNonUppercase ha tipo `[Char] -> [Char]`, il che significa che mappa una stringa in un'altra stringa.

```
addThree :: Int -> Int -> Int -> Int  
addThree x y z = x + y + z
```

Quando si passano più parametri, i parametri sono **separati da ->**, senza distinzione speciale tra parametri e tipo di ritorno. Il **tipo di ritorno** è l'ultimo elemento nella dichiarazione.

### Variabili di tipo

Esaminiamo il tipo della funzione head:

```
ghci> :t head
```

```
head :: [a] -> a
```

Il simbolo `a` è una **variabile di tipo**, cioè un tipo **generico**, che può rappresentare qualsiasi tipo. Le funzioni che utilizzano variabili di tipo sono chiamate **funzioni polimorfiche**.

### Typeclasses 101

Le typeclasses sono **interfacce** che specificano un **insieme di operazioni** comuni per diversi tipi. Un **tipo** è un'istanza di una typeclass se **implementa** le operazioni definite nell'interfaccia.

Questo consente il polimorfismo ad hoc, permettendo di scrivere funzioni generiche che operano su qualsiasi tipo che soddisfa una determinata typeclass.

## Esempio

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

`==` è una funzione come tutti gli altri operatori in Haskell. La parte `(Eq a) =>` è una **constraint** che specifica che il tipo degli argomenti della funzione deve essere un'istanza della typeclass `Eq`.

## Typeclass Eq

La typeclass **Eq** fornisce un'interfaccia per **verificare l'uguaglianza**. Qualsiasi tipo che consente di confrontare l'uguaglianza appartiene alla typeclass `Eq`. Tutti i tipi standard di Haskell, eccetto `IO` (per input/output) e le funzioni, appartengono a questa typeclass.

```
ghci> 5 == 5
```

```
True
```

## Typeclass Ord

**Ord** è la typeclass per i tipi che supportano un **ordinamento**. Include tutti i tipi standard tranne le funzioni.

```
ghci> :t (>)
(>) :: Ord a => a -> a -> Bool
```

```
ghci> 5 >= 2
```

```
True
```

La funzione `compare` restituisce un valore di tipo `Ordering`: `GT`, `LT`: less than (minore), `EQ`: equal (uguale).

```
ghci> 5 `compare` 2
```

```
GT
```

## Typeclass Show

I membri di `Show` sono tipi che possono essere **convertiti in stringhe**. La funzione più usata con questa typeclass è `show`, che prende un valore e lo restituisce come stringa.

```
ghci> show 3
```

```
"3
```

## Typeclass Read

`Read` è l'opposto di `Show`. I membri di `Read` sono stringhe che possono essere convertite in tipi. La funzione `read` prende una stringa e restituisce un valore di un tipo che appartiene a `Read`.

```
ghci> read "8.2" + 3.8
```

```
12.0
```

Se non specifichiamo come usare il valore restituito da `read`, Haskell non saprà che tipo restituire. Per risolvere, possiamo specificare il tipo esplicitamente:

```
ghci> read "5" :: Float
```

```
5.0
```

## Typeclass Enum

I membri di `Enum` sono tipi **ordinati sequenzialmente**, che possono essere usati in intervalli di liste e implementano funzioni come `succ` (successore) e `pred` (precedente).

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

```
deriving (Eq, Show)
```

```
-- Istanza della typeclass Enum per il tipo Day
instance Enum Day where
    succ Sunday = Monday -- Dopo Sunday viene Monday
    succ day    = toEnum ((fromEnum day + 1) `mod` 7)

    pred Monday = Sunday -- Prima di Monday viene Sunday
    pred day    = toEnum ((fromEnum day - 1) `mod` 7)

    toEnum n = case n `mod` 7 of
        0 -> Monday
        1 -> Tuesday
        2 -> Wednesday
        3 -> Thursday
        4 -> Friday
        5 -> Saturday
        6 -> Sunday
        _ -> error "Invalid day"
```

## Typeclass Bounded

I membri della typeclass `Bounded` in Haskell sono tipi che hanno un valore minimo, rappresentato da `minBound`, e un valore massimo, rappresentato da `maxBound`. Questi valori sono definiti per ogni tipo che è istanza di `Bounded` e rappresentano i limiti estremi del tipo.

Ad esempio, per un tipo di coppie (`Bool`, `Int`, `Char`), il massimo (`maxBound`) è (`True`, `2147483647`, `'1114111'`), dove ogni componente assume il proprio valore massimo.

### Typeclass Num

I membri di `Num` sono tipi numerici, di diversi tipi di precisione.

```
ghci> :t 20
20 :: (Num t) => t
ghci> 20 :: Float
20.0
```

### Typeclass Integral

`Integral` include solo numeri interi (`Int` e `Integer`).

### Typeclass Floating

`Floating` include solo numeri in virgola mobile (`Float` e `Double`).

### Funzione fromIntegral

`fromIntegral` è utile per convertire numeri interi in numeri più generali. Esempio:

```
fromIntegral :: (Num b, Integral a) => a -> b
```

Utile quando bisogna combinare tipi interi e in virgola mobile:

```
ghci> fromIntegral (length [1,2,3,4]) + 3.2
```

```
7.2
```

## Sintassi nelle Funzioni

### Pattern Matching

Il **pattern matching** è una tecnica di espressione del corpo di una funzione in cui i dati in input vengono confrontati ad un insieme pattern. In base al pattern assunto, viene eseguito il corrispondente corpo di funzione.

Quando si chiama una funzione che utilizza il pattern matching, i modelli vengono verificati dall'alto verso il basso.

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

### Funzione Fattoriale

La funzione fattoriale può essere definita ricorsivamente come in matematica.

Partiamo dicendo che il fattoriale di 0 è 1. Poi, affermiamo che il fattoriale di un intero positivo è quel numero moltiplicato per il fattoriale del suo predecessore.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

La **clausola base** deve essere scritta prima di quella ricorsiva, altrimenti il calcolo non si fermerebbe mai a causa della laziness.

### Pattern Matching su Tuples

È possibile utilizzare il pattern matching anche sulle tuple.

```
addVectors :: (Num a) => (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

### Pattern Matching su Liste

Per le liste si può utilizzare il pattern **x:xs**, dove **x** rappresenta la **testa** e **xs** il **resto** della lista.

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (x:xs) = 1 + length' xs
```

## Pattern Matching Avanzato

Gli **as patterns** in Haskell permettono di mantenere un **riferimento al valore di input originale** mentre si scomponete il valore in base a un pattern. La sintassi è **nameFunc aliasName@(firstPart:rest) = expr**

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

In questo caso all è un alias per l'intera stringa originale.

## Where

La parola chiave **where** in Haskell consente di **definire nomi** o funzioni locali **all'interno** di una funzione, in modo da migliorare la leggibilità in caso di pattern matching. Questi nomi hanno **scope** solo nella funzione in cui sono definiti.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
| weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
| weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pfffft, I bet you're ugly!"
| weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise                 = "You're a whale, congratulations!"

bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
| bmi <= skinny = "You're underweight, you emo, you!"
| bmi <= normal = "You're supposedly normal. Pfffft, I bet you're ugly!"
| bmi <= fat     = "You're fat! Lose some weight, fatty!"
| otherwise       = "You're a whale, congratulations!"
where bmi = weight / height ^ 2
      skinny = 18.5
      normal = 25.0
      fat = 30.0
```

## Usare il Pattern Matching nelle Basi where

Le associazioni where permettono anche di usare il pattern matching. Ad esempio, possiamo riscrivere la sezione where della funzione sopra così:

```
...
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

## Let binding

Il **let binding** è un'espressione che permette di **definire variabili locali o funzioni** all'interno di un'espressione più grande. A differenza di where, let è un'espressione che può essere **utilizzata ovunque**, ma i suoi nomi **hanno scope** limitato alla parte **in**. La sua forma è: **let <bindings> in <expression>**

**Bindings:** Sono definizioni locali di variabili o funzioni.

**Expression:** È l'espressione che utilizza le variabili definite nei bindings.

```
squarePlusOne :: Int -> Int
✓ squarePlusOne x =
| let square = x * x
| in square + 1
```

## Espressioni case

Un'**espressione case** consente di effettuare il pattern matching direttamente su un valore, determinando il risultato in output in base ai diversi pattern corrispondenti.

**case <espressione> of**

```
pattern1 -> risultato1
pattern2 -> risultato2
```

```
...
```

```
patternN -> risultatoN
```

```
describeNumber :: Int -> String
describeNumber x = case x of
| 0 -> "Zero"
| 1 -> "One"
| _ -> "Some other number"
```

```

describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of
    [] -> "empty."
    [x] -> "a singleton list."
    _ -> "a longer list."

```

Se nessun pattern corrisponde e non c'è un caso catch-all (ad esempio `_`), si verifica un errore a runtime.

## Ricorsione

La **ricorsione** è un modo per definire funzioni in cui una funzione viene richiamata all'interno della propria definizione. In Haskell, la ricorsione viene utilizzata al posto dei costrutti `while` e `for` per effettuare iterazioni.

La definizione dei casi base è essenziale per assicurare la terminazione della funzione ricorsiva. Senza di esse, una funzione ricorsiva continuerebbe indefiniteamente.

### Esempi di Funzioni Ricorsive

La funzione `maximum` in Haskell trova il massimo elemento di una lista. Con la ricorsione possiamo implementarla così:

```

maximum' :: (Ord a) -> [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = if x > maximum' xs then x else maximum' xs

```

Divide la lista in testa e coda (`x:xs`), confronta il primo elemento con il massimo della coda.

La funzione `replicate'` ripete un elemento un certo numero di volte

```

replicate' :: (Num i, Ord i) -> i -> a -> [a]
replicate' n x
| n <= 0      = []
| otherwise = x : replicate' (n-1) x

```

La funzione `lengthR` calcola la lunghezza di una lista.

```

lengthR :: (Num a) -> [a] -> a
lengthR [] = 0
lengthR (x:xs) = 1 + lengthR xs

```

La funzione `take'` prende i primi `n` elementi di una lista.

```

take' :: (Num i, Ord i) -> i -> [a] -> [a]
take' n _
| n <= 0      = []
take' _ []      = []
take' n (x:xs) = x : take' (n-1) xs

```

## Funzioni di Ordine Superiore in Haskell

Le **funzioni di ordine superiore** sono funzioni che accettano altre funzioni come parametri o restituiscono funzioni come risultato.

### Funzioni Currificate

In Haskell, ogni funzione accetta **un solo parametro alla volta**. Tuttavia, tramite la **currificazione**, una funzione che sembra accettare più parametri è in realtà una **sequenza di funzioni**, ognuna delle quali accetta un parametro e **restituisce una nuova funzione**.

`max :: (Ord a) => a -> a -> a`

Il tipo può essere riscritto come:

`max :: (Ord a) => a -> (a -> a)`

Questo significa che `max` accetta un valore di tipo `a` e restituisce una funzione che accetta un altro valore di tipo `a` e restituisce un risultato di tipo `a`. Ad esempio:

`ghci> (max 4) 5`

`5`

`max 4` restituisce una **funzione che confronta 4 con un altro valore**. `5` viene applicato a questa funzione, producendo il risultato finale.

## Applicazione Parziale

L'applicazione parziale consiste nel chiamare una funzione con meno parametri rispetto a quelli richiesti, ottenendo una nuova funzione che attende i parametri mancanti.

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

Chiamando multThree con meno parametri, otteniamo nuove funzioni:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
```

## Sezioni di Operatori Infix

Gli operatori infix possono essere parzialmente applicati utilizzando **sezioni**. Per creare una sezione, basta racchiudere l'operatore tra parentesi e fornire un argomento su uno dei due lati. Verrà applicato l'operatore, col secondo parametro, al primo parametro successivamente passato.

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
ghci> divideByTen 200
20.0
```

## Funzioni di Ordine Superiore

**applyTwice**: Una funzione che accetta un'altra funzione come parametro e la applica due volte a un valore:

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
ghci> applyTwice (+3) 10
16
```

**zipWith**: La funzione zipWith accetta una funzione a due parametri, due liste, e combina le liste applicando la funzione a ogni coppia di elementi corrispondenti alle due liste:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

---

Esempi:

```
ghci> zipWith' (+) [1,2,3] [4,5,6]
[5,7,9]
```

**Map**: La funzione map applica una funzione a ogni elemento di una lista, restituendo una nuova lista:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
ghci> map (+3) [1,2,3]
[4,5,6]
```

**Filter**: La funzione filter prende una lista e un predicato (una funzione che restituisce un booleano) e restituisce una lista con gli elementi che soddisfano il predicato:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
| p x      = x : filter p xs
| otherwise = filter p xs
ghci> filter' (>3) [1,2,3,4,5]
[4,5]
```

## Funzioni anonime

Le **lambda expressions**, sono funzioni definite senza nome, utili per scrivere codice conciso e spesso utilizzate quando le funzioni sono necessarie solo temporaneamente. La sintassi di una funzione anonima in Haskell è **\param1 paramN -> <espressione>**, dove:

\param1 paramN: Indica che la funzione accetta i parametri param1...paramN

->: Separa il parametro dal corpo della funzione.

<espressione>: Corpo della funzione

```
ghci> (\x -> x + 1) 5
```

```
6
```

Le funzioni anonime possono essere assegnate anche ad una variabile.

```
ghci> f = \x -> x + 1
```

```
ghci> f 7
```

```
8
```

Le funzioni anonime possono utilizzare **pattern matching** per scomporre argomenti complessi, come tuple o liste.

```
ghci> h = \(x, y) -> x + y
```

```
ghci> h (3, 4)
```

```
7
```

```
ghci> k = \(x:xs) -> length xs
```

```
ghci> k "hello"
```

```
4
```

```
-
```

## Folds

Le **folds** sono funzioni che consentono di ridurre una lista a un **singolo valore**.

Le folds prendono **tre parametri**: Una **funzione** che accetta **due parametri**, un valore iniziale **accumulatore** e Una lista.

La funzione binaria viene **applicata** tra l'accumulatore e il primo elemento della lista, aggiornando l'accumulatore. Questo processo viene iterato per ogni elemento della lista.

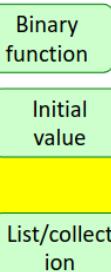
## the reduce combinator

**reduce (foldl, foldr)**: takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.

```
-- folds values from end to beginning of list
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

-- folds values from beginning to end of list
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

-- variants for non-empty lists
foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
```



### Fold Sinistra: foldl

La funzione foldl riduce una lista applicando una funzione binaria da sinistra a destra, utilizzando un valore iniziale. La sua firma è: foldl :: (b -> a -> b) -> b -> [a] -> b

**Funzione binaria (b -> a -> b)**: Prende il risultato parziale e un elemento della lista, restituendo un nuovo risultato parziale.

**Valore iniziale b**: Il valore di partenza utilizzato quando la lista è vuota.

**Lista [a]**: La collezione di elementi da ridurre.

**Comportamento:**

```
foldl f z [] = z
```

```
foldl f z (x:xs) = foldl f (f z x) xs
```

Se la lista è vuota, restituisce il valore iniziale. Se la lista non è vuota, applica la funzione binaria al valore iniziale e al primo elemento della lista, poi utilizza il risultato per continuare con il resto della lista.

La **foldl (fold left)** elabora la lista da sinistra verso destra.

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

### Esempio

Calcoliamo la somma di [3, 5, 2, 1]:

1. Partiamo con un accumulatore 0.
2. Applichiamo la funzione binaria a ogni elemento:
  - $0 + 3 = 3$
  - $3 + 5 = 8$
  - $8 + 2 = 10$
  - $10 + 1 = 11$
3. Risultato: 11.

## Fold Destra: foldr

La funzione foldr riduce una lista applicando una funzione binaria da destra a sinistra, utilizzando un valore iniziale. La sua firma è: `foldr :: (a -> b -> b) -> b -> [a] -> b`

### Componenti:

**Funzione binaria (a -> b -> b):** Prende un elemento della lista e il risultato parziale, restituendo un nuovo risultato parziale.

**Valore iniziale b:** Il valore di partenza utilizzato quando la lista è vuota.

**Lista [a]:** La collezione di elementi da ridurre.

### Comportamento:

`foldr f z [] = z`

`foldr f z (x:xs) = f x (foldr f z xs)`

Se la lista è vuota, restituisce il valore iniziale. Se la lista non è vuota, applica la funzione binaria al primo elemento e al risultato della riduzione del resto della lista. Utilizza la laziness di haskell in modo che tutte le operazioni siano in attesa finché non viene calcolato l'ultimo elemento. Poi vengono richiamate a cascata da dx verso sx.

```
-- Somma di una lista usando foldr
sumList :: [Int] -> Int
sumList xs = foldr (+) 0 xs
```

## Implementazione mapF

```
mapF :: (a -> b) -> [a] -> [b]
mapF f xs = foldr (\x acc -> f x : acc) [] xs

Ok, one module loaded.
ghci> mapF (+3) [1,2,3,4]
[4,5,6,7]
```

La funzione mapF applica una funzione f a ogni elemento di una lista, creando una nuova lista con i risultati, utilizzando foldR. La funzione ausiliaria applica la funzione f al primo elemento e lo inserisce in testa alla lista accumulatore. La foldr applica questa dall'ultimo elemento al primo.

Applichiamo la funzione (+3) alla lista [1, 2, 3]:

1. Partiamo con un accumulatore [] .
2. Elaboriamo gli elementi da destra a sinistra:
  - $3 : (+3) 3 = 6$ , aggiunto all'accumulatore  $\rightarrow [6]$ .
  - $2 : (+3) 2 = 5$ , aggiunto all'accumulatore  $\rightarrow [5, 6]$ .
  - $1 : (+3) 1 = 4$ , aggiunto all'accumulatore  $\rightarrow [4, 5, 6]$ .
3. Risultato: [4, 5, 6].

## Differenze tra foldl e foldr

**Ordine:** foldl inizia dall'accumulatore iniziale, mentre foldr inizia dall'ultimo elemento. Foldr può lavorare con **liste infinite** perché costruisce e valuta l'espressione pigramente, solo quando necessario.

Foldl **non può gestire liste infinite** perché necessita di attraversarle completamente prima di restituire un risultato, il che è impossibile per una lista infinita.

## Funzioni Implementabili con Folds

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
```

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []
```

## Scans

Le scans (scanl, scanr) sono simili alle folds, ma restituiscono una lista di tutti gli stati intermedi dell'accumulatore.

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
```

## Composizione di Funzioni

In matematica, la composizione di funzioni è definita come  $f(g(x))$ . In Haskell, la composizione si fa con l'operatore “.”.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

$f . g$  produce una nuova funzione che applica  $g$  al suo argomento e poi applica  $f$  al risultato.

“.” è **destra-associativo**, quindi  $f . g . h$  equivale a  $f . (g . h)$ .

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
```

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
```

```
[-5,-3,-6,-7,-3,-2,-19,-24]
```

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
```

Diventa:

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

## Operatore |>

L'operatore **|>** permette di applicare una funzione a un valore in modo sequenziale, migliorando la leggibilità del codice.

```
(|>) :: t1 -> (t1 -> t2) -> t2
x |> f = f x
```

Prende un valore di tipo  $t1$ . Prende una funzione che trasforma  $t1$  in  $t2$ . Restituisce un risultato di tipo  $t2$ . L'operatore applica la funzione al valore usando la sintassi  $a |> f$ , equivalente a  $f a$ .

```
length (tail (reverse [1,2,3]))
```

Passaggi:  $\text{reverse } [1,2,3] \rightarrow [3,2,1]$  ,  $\text{tail } [3,2,1] \rightarrow [2,1]$  ,  $\text{length } [2,1] \rightarrow 2$

```
[1,2,3] |> reverse |> tail |> length
```

**Interpretazione:**  $[1,2,3]$  viene passato a  $\text{reverse}$ . Il risultato di  $\text{reverse} ([3,2,1])$  viene passato a  $\text{tail}$ . Il risultato di  $\text{tail} ([2,1])$  viene passato a  $\text{length}$ .

## Funzioni Ricorsive di Coda (Tail-Recursive Functions)

Le **funzioni ricorsive di coda** sono funzioni in cui la chiamata ricorsiva è l'**ultima operazione** eseguita dalla funzione prima di restituire un valore, restituendo direttamente il risultato della chiamata ricorsiva senza ulteriori calcoli o modifiche.

```
int trfun() {
    ...
    return trfun(); // Nessuna operazione dopo la chiamata ricorsiva.
}
```

## Esempio: Funzione Non Ricorsiva di Coda

```
int rfun() {
    ...
    return 1 + rfun(); // Operazione (aggiunta di 1) dopo la chiamata ricorsiva.
}
```

In questo caso, la funzione deve eseguire un'operazione (aggiungere 1) dopo la chiamata ricorsiva, quindi non è ricorsiva di coda.

---

## Vantaggi delle Funzioni Ricorsive di Coda

Nelle chiamate ricorsive tradizionali, ogni chiamata aggiunge un nuovo frame allo stack di esecuzione, aumentando il consumo di memoria.

Nelle chiamate ricorsive di coda, lo stato corrente della funzione non è più necessario dopo la chiamata ricorsiva. Il compilatore può **riutilizzare il frame esistente**, riducendo l'uso dello stack e prevenendo possibili stack overflow.

### Ottimizzazione della Ricorsione di Coda (Tail-Recursion Optimization):

Il compilatore può trasformare le chiamate ricorsive di coda in semplici **salti (jumps)** all'inizio della funzione. Questo elimina la necessità di gestire ogni chiamata come una nuova funzione, migliorando l'efficienza del codice.

**foldl** è ricorsiva di coda (tail-recursive), il che significa che può essere ottimizzata per evitare il consumo di stack. Tuttavia, a causa della laziness di Haskell, questa ottimizzazione non viene applicata automaticamente.

**foldr**, d'altro canto, non è ricorsiva di coda, quindi utilizza più risorse di memoria in situazioni normali.

## Creare i propri tipi e classi di tipi

### Introduzione ai tipi di dati algebrici

Vediamo come viene definito il tipo Bool nella libreria standard:

```
data Bool = False | True
```

La keyword **data** indica che stiamo definendo un nuovo tipo di dato. La parte prima del simbolo = rappresenta il **nome del tipo** (Bool), mentre le parti dopo = sono i **costruttori di valore**, che rappresentano i diversi valori che il tipo può assumere.

Sia il nome del tipo che i costruttori di valore devono iniziare con una lettera maiuscola.  
Il tipo Int viene definito come:

```
data Int = -2147483648 | -2147483647 | ... | 2147483647
```

Un **tipo di dato algebrico** in Haskell è un costrutto che permette di definire un tipo composto da uno o più **costruttori di valore**, ognuno dei quali rappresenta un possibile modo di costruire un valore di quel tipo. Ogni costruttore può accettare zero o più argomenti, che rappresentano i dati contenuti nell'istanza del tipo.

Ora consideriamo come potremmo rappresentare una figura geometrica in Haskell.

```
data Point = Point Float Float deriving (Show)  
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

**Costruttori di valore:** Sono funzioni che creano istanze del tipo definito. Hanno una firma che indica i tipi degli **argomenti che accettano** e il tipo che **producono**.

**Costruttori di valore:** Circle :: Point -> Float -> Shape: accetta un Point e un Float (rappresentano il centro e il raggio) e restituisce un valore di tipo Shape. Rectangle :: Point -> Point -> Shape: accetta due Point (vertici opposti) e restituisce un valore di tipo Shape.

La keyword deriving (Show) serve affinchè haskell generi automaticamente una rappresentazione testuale dei valori Shape.

### Funzioni sui nuovi tipi

I costruttori di valore sono funzioni, quindi possiamo utilizzarli direttamente. Ad esempio, possiamo creare una funzione per calcolare l'area di una figura:

```
surface :: Shape -> Float  
surface (Circle(Point x y) r) = pi * r ^ 2  
surface (Rectangle (Point x1 y1) (Point x2 y2)) = abs (x2 - x1) * abs (y2 - y1)  
  
ghci> surface (Circle(Point 1 2)2)  
12.566371
```

Possiamo aggiungere altre funzioni utili, ad esempio per traslare una figura:

```
nudge :: Shape -> Float -> Float -> Shape  
nudge (Circle (Point x y) r) dx dy = Circle (Point (x+dx) (y+dy)) r  
nudge (Rectangle (Point x1 y1) (Point x2 y2)) dx dy =  
    Rectangle (Point (x1+dx) (y1+dy)) (Point (x2+dx) (y2+dy))
```

Questa funzione sposta la figura di un determinato offset in orizzontale (dx) e in verticale (dy).

### Moduli e visibilità

Infine, possiamo **esportare** il nostro tipo Shape e le relative funzioni in un **modulo**, controllando quali costruttori di valore e funzioni sono accessibili:  
module Shapes

```
( Point(..)
, Shape(..)
, surface
, nudge
) where
```

Questo **modulo rende visibili** tutti i costruttori di Point e Shape, ma consente di controllare selettivamente quali funzioni esportare.

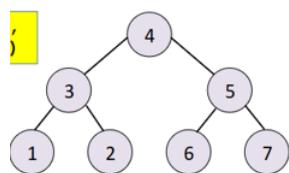
## Struttura Dati Ricorsiva e Pattern Matching

Definiamo ora ricorsivamente un albero:

```
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

**Tree** è un tipo di dato algebrico che rappresenta un albero. Il **Costruttore Leaf**: rappresenta una foglia dell'albero e contiene un valore Int. Il **Costruttore Node**: rappresenta un nodo interno che contiene un valore Int e due sottoalberi (Tree).

```
(Node (4, Node (3, Leaf 1, Leaf 2), Node (5, Leaf 6, Leaf 7)))
```



```
sumT :: Tree -> Int
sumT (Leaf n) = n
sumT (Node (n, t1, t2)) = n + sumT t1 + sumT t2
```

**Caso base:** Se il nodo è una foglia (Leaf n), restituisce il valore contenuto nella foglia (n).

**Caso ricorsivo:** Se il nodo è un nodo interno (Node), somma il valore del nodo (n) con le somme dei due sottoalberi (t1 e t2).

## Sintassi dei Record

Quando dobbiamo rappresentare un tipo di dato complesso, come una persona, possiamo utilizzare una definizione di tipo algebrico. Supponiamo di voler memorizzare informazioni quali il nome, il cognome, l'età, l'altezza, il numero di telefono e il gusto di gelato preferito.

```
data Person = Person String String Int Float String String deriving (Show)
```

Qui, ogni campo rappresenta un'informazione, ma non ha un nome esplicito, rendendo difficile accedere ai valori individuali.

Haskell offre una soluzione più elegante con la **sintassi dei record**. Ecco come possiamo riscrivere il tipo Person:

```
data Person = Person { firstName :: String
                      , lastName :: String
                      , age :: Int
                      , height :: Float
                      , phoneNumber :: String
                      , flavor :: String
                    } deriving (Show)
```

Haskell crea automaticamente funzioni per accedere ai campi.

```
ghci> let guy = Person {firstName="Buddy", lastName="Finklestein", age=43, height=184.2, phoneNumber="526-2928", flavor="Chocolate"}
ghci> guy
Person {firstName = "Buddy", lastName = "Finklestein", age = 43, height = 184.2, phoneNumber = "526-2928", flavor = "Chocolate"}
ghci> firstName guy
"Buddy"
```

## Type parameters

Un **costruttore di valore** può accettare dei parametri per creare un nuovo valore. Ad esempio, il costruttore Person accetta sei valori e produce un valore del tipo Person.

## Definizione di Costruttore di Tipo

Un **costruttore di tipo** in Haskell è un costrutto che accetta **parametri di tipo** per produrre nuovi tipi concreti. È simile ai **template** in C++ e consente di creare tipi generici che possono essere adattati a diversi tipi di dati.

Un costruttore di tipo **non è un tipo concreto**. Deve essere "riempito" con un parametro di tipo per produrre un tipo utilizzabile.

I **parametri di tipo** in Haskell permettono di creare tipi generici e flessibili che funzionano indipendentemente dal tipo specifico dei dati che contengono. La sintassi usa un segnaposto per il tipo generico, che viene specificato solo quando si usa il costruttore di tipo

La sintassi per i parametri di tipo è la seguente:

```
data typeName typeParameter = Costruttore typeParameter
```

**typeParameter** è il parametro di tipo.

**typeName** è un **costruttore di tipo**, che può generare tipi concreti come typeName Int, typeName String, ecc.

**Costruttore** è un **costruttore di valore**, che crea valori di tipo typeName typeParameter.

## Quando Usare i Type parameters

I parametri di tipo sono utili quando un tipo deve funzionare in modo indipendente dal tipo dei valori che contiene.

Versione senza Parametri di Tipo:

```
data Car = Car { company :: String, model :: String, year :: Int } deriving (Show)
```

Versione con Parametri di Tipo:

```
data Car typeA typeB typeC = Car { company :: typeA, model :: typeB, year :: typeC } deriving (Show)
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

## Maybe

```
data Maybe typeParameter = Nothing | Just typeParameter
```

Maybe è utile per rappresentare dati opzionali o risultati di calcoli che possono fallire, fornendo una gestione elegante per l'assenza di valori.

**Parametro di tipo** typeParameter: a rappresenta un tipo generico (può essere Int, String, Char, ecc.).

**Costruttori di valore:** **Nothing**: rappresenta l'assenza di valore. **Just** typeParameter: incapsula un valore di tipo a in un valore di tipo Maybe typeParameter.

Per esempio, Maybe Int: tipo che rappresenta un intero opzionale.

## Funzione Just

La funzione Just è uno dei **costruttori di valore** di Maybe. La sua firma è:

```
Just :: typeParameter -> Maybe typeParameter
```

Accetta un valore di tipo generico a e restituisce un valore di tipo Maybe typeParameter, **incapsulando** il valore originale.

## Valore Nothing

**Nothing** rappresenta l'**assenza** di valore. Ha un tipo generico Maybe typeParameter, dove a può essere riempito in base al contesto.

Poiché Nothing non contiene alcun valore, può agire come qualsiasi tipo Maybe typeParameter:

## Esempi Funzionali di Maybe

Supponiamo di avere una funzione che calcola il reciproco di un numero:

```
reciprocal :: Double -> Maybe Double
reciprocal 0 = Nothing
reciprocal x = Just (1 / x)

ghci> reciprocal 4
Just 0.25
```

## Type Synonyms (Sinonimi di Tipo)

I **sinonimi di tipo** in Haskell permettono di dare un nuovo nome alias a un tipo esistente.

Per definire un sinonimo di tipo, si utilizza la parola chiave type:

**type NuovoNome = TipoEsistente**

Il tipo String è definito come sinonimo per [Char] nella libreria standard: type String = [Char]

## Sinonimi di Tipo Parametrizzati

I sinonimi di tipo possono essere parametrizzati, rendendoli generici e flessibili:

type AssocList k v = [(k, v)]

### Esempio:

Una funzione che cerca un valore dato un chiave in una lista di associazioni:

findValue :: (Eq k) => k -> AssocList k v -> Maybe v

findValue key alist = lookup key alist

## Typeclasses 102

La typeclass Eq è usata per tipi che possono essere confrontati per uguaglianza.

Definisce le funzioni (==) e (/=):

*class Eq a where*

(==) :: a -> a -> Bool

(/=) :: a -> a -> Bool

x == y = not (x /= y)

x /= y = not (x == y)

a è una variabile di tipo. Le funzioni == e /= hanno dichiarazioni di tipo ma non necessariamente implementazioni predefinite. La definizione ricorsiva tra == e /= mostra come sia possibile definire una funzione in termini dell'altra.

### Creare un'Istanza di una Typeclass

Per rendere un tipo un'istanza di una typeclass, si usa la parola chiave **instance**.

#### Esempio: Tipo TrafficLight

data TrafficLight = Red | Yellow | Green

Facciamolo un'istanza di Eq implementando (==):

```
instance Eq TrafficLight where
    Red == Red = True
    Yellow == Yellow = True
    Green == Green = True
    _ == _ = False
ghci> Red == Red
True
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
ghci>
```

## Typeclass per Tipi Parametrizzati

Le typeclass possono essere usate con tipi parametrizzati. Ad esempio, Maybe è un costruttore di tipo:

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

(Eq m) => specifica che il contenuto di Maybe deve essere un'istanza di Eq.

Questo rende qualsiasi tipo della forma Maybe m un'istanza di Eq, a condizione che m sia in Eq.

---

## Creare una TypeClass Personalizzata

La sintassi per la definizione di un'interfaccia **typeclass** e di una sua istanza è:

```
class nameInterface nameType where
```

```
    functionToImplementDef
```

```
instance nameInterface nameTypeConcr where
```

```
    functionImplementation
```

Definiamo una typeclass YesNo che verifica se un valore può essere considerato "vero" o "falso":

```
class YesNo a where
|   yesno :: a -> Bool
```

Per Int, 0 è "falso", tutti gli altri sono "veri".

```
instance YesNo Int where
|   yesno 0 = False
|   yesno _ = True
```

Per liste [] è "falso", le liste non vuote sono "vere"):

```
instance YesNo [a] where
|   yesno [] = False
|   yesno _ = True
```

```
ghci> yesno []
False
ghci> yesno [1,11]
True
```

In Haskell, un dizionario è una struttura dati interna che mappa un tipo alle implementazioni delle funzioni di una typeclass per quel tipo. Quando viene utilizzato un simbolo sovraccaricato (ad esempio + o ==), il compilatore lo traduce in una funzione con un parametro aggiuntivo: il dizionario. Questo dizionario contiene le implementazioni della typeclass specifiche per il tipo, e il compilatore lo utilizza per determinare l'operazione corretta durante l'esecuzione.

### Processo di Compilazione delle Type Classes

In Haskell, una typeclass è dichiarata utilizzando la parola chiave class, specificando i metodi che devono essere implementati per i tipi che ne diventano istanze. Il compilatore trasforma ogni typeclass in un nuovo tipo di "dizionario", rappresentato internamente come una struttura dati che:  
Contiene le implementazioni dei metodi definiti nella typeclass AND Fornisce funzioni selettore per accedere a questi metodi.

```
haskell

class Num n where
  (+) :: n -> n -> n
  (*) :: n -> n -> n
  negate :: n -> n
```

viene tradotta internamente in qualcosa di simile a:

```
haskell

data Num n = MkNum {
  plus :: n -> n -> n,
  times :: n -> n -> n,
  negate :: n -> n
}
```

### Dichiarazione delle Istanze

Ogni dichiarazione di istanza viene trasformata in un dizionario che contiene le implementazioni dei metodi della typeclass per un tipo specifico. Questo dizionario viene poi passato implicitamente e utilizzato per accedere alle implementazioni concrete durante l'esecuzione.

```
instance Num Int where
  a + b = intPlus a b
  a * b = intTimes a b
  negate a = intNeg a
```

viene tradotta in:

```
haskell

dNumInt :: Num Int
dNumInt = MkNum intPlus intTimes intNeg
```

## Chiamate a Funzioni Sovraccaricate

Il compilatore riscrive le chiamate a funzioni sovraccaricate aggiungendo un parametro per il dizionario. Usa il tipo statico qualificato della funzione per selezionare il dizionario corretto.

square :: Num n => n -> n

square x = x \* x

Viene tradotto in:

square :: Num n -> n -> n

square d x = (\*) d x x

## Funzioni con Dizionari Multipli

Per funzioni che utilizzano più istanze di type class, il compilatore genera un dizionario per ciascun tipo.

### Esempio:

squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)

squares (x, y, z) = (square x, square y, square z)

Diventa:

squares :: (Num a, Num b, Num c) -> (a, b, c) -> (a, b, c)

squares (da, db, dc) (x, y, z) = (square da x, square db y, square dc z)

## Sostituzione delle definizioni di default

Le dichiarazioni di istanza per Eq possono scegliere di sovrascrivere le implementazioni di default con versioni più specifiche e ottimizzate. Ad esempio, per il tipo Int, potremmo implementare (==) direttamente senza fare affidamento sulla definizione di default.

## Sistema di inferenza dei tipi

L'**inferenza dei tipi** è un meccanismo con cui il compilatore **determina automaticamente i tipi** delle variabili e delle funzioni senza che questi siano esplicitamente specificati.

## Algoritmo di inferenza dei tipi

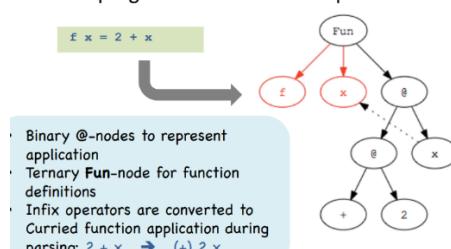
Si inizia con l'analisi del testo del programma per costruire un albero di parsing. Viene assegnata una variabile di tipo a ciascun nodo dell'albero di parsing. Queste variabili rappresentano tipi sconosciuti che saranno risolti in seguito. Si generano vincoli basati su costanti conosciute (es. il numero "2"), operatori predefiniti (es. +), e funzioni note (es. tail).

Si generano vincoli anche dalla forma dell'albero. Ad esempio, per i nodi di applicazione di funzione o astrazione, il tipo del nodo applicato deve corrispondere al tipo di funzione (dominio  $\rightarrow$  codominio).

I vincoli generati vengono risolti utilizzando l'unificazione, un processo che trova valori per le variabili di tipo che soddisfano tutti i vincoli, determinando tipi coerenti per tutte le espressioni. Infine, una volta risolti i vincoli, si può determinare il tipo delle dichiarazioni principali del programma, assegnando i tipi più generali compatibili.

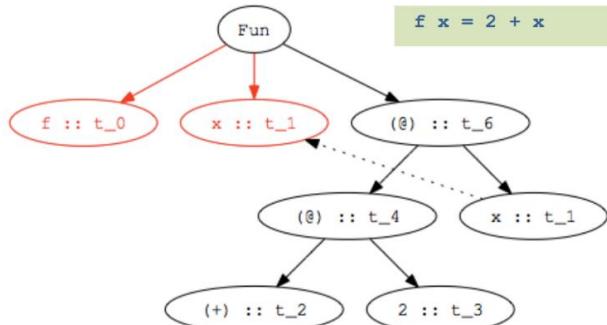
### Step 1: Parse Program

- Parse program text to construct parse tree.



**Parse Program:** Il compilatore analizza il testo del programma e costruisce un albero di parsing (parse tree). Ad esempio, l'espressione  $f x = 2 + x$  viene rappresentata come un albero, con nodi per la funzione, le applicazioni, e le operazioni aritmetiche. Operatori infissi (come  $+$ ) con più parametri vengono convertiti in funzioni **curried** durante il parsing, ovvero funzioni che prendono un argomento e restituiscono una funzione che prende i restanti argomenti come dominio e restituisce il codominio.

## Step 2: Assign type variables to nodes

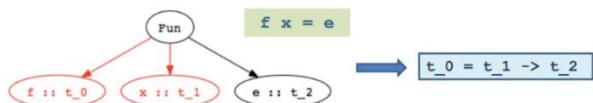


Variables are given same type as binding occurrence.

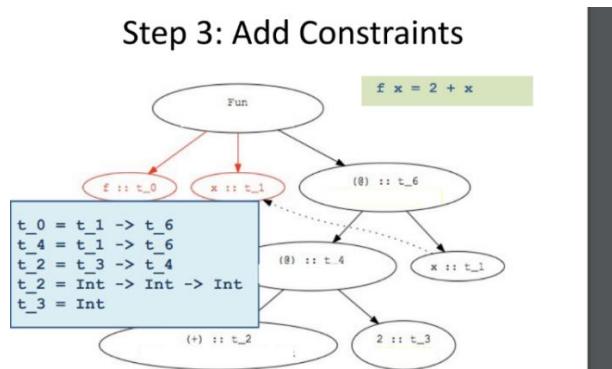
42

**Assign Type Variables:** A ogni nodo dell'albero viene assegnata una variabile di tipo. Per esempio, il simbolo  $f$  riceve il tipo  $t_0$ ,  $x$  riceve  $t_1$ , e così via. Variabili dello stesso nome condividono lo stesso tipo. In questa fase non ci sono vincoli, ma ogni elemento ha un tipo generico.

## Constraints from Abstractions



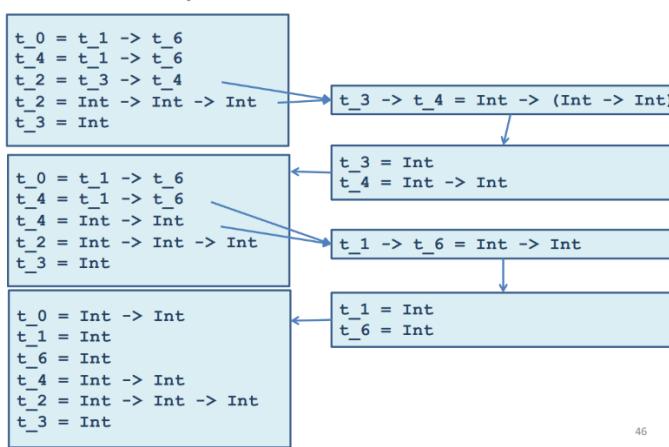
## Step 3: Add Constraints



**Add Constraints:** Il compilatore aggiunge vincoli basati sull'uso delle funzioni e operatori. I nodi di applicazione come  $(@)$  generano vincoli del tipo  $t_0 = t_1 \rightarrow t_2(6)$ , indicando che  $f$  è una funzione che mappa  $x$  a un risultato. Operazioni come  $+$  aggiungono il vincolo che sia una funzione su interi:  $t_2 = \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

**Solve Constraints:** I vincoli raccolti vengono risolti tramite un processo di unificazione (unification), che confronta i tipi e li rende coerenti. Ad esempio, se  $t_2 = t_3 \rightarrow t_4$ , con  $t_2 = \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$  e  $t_3 = \text{Int}$ , allora  $t_4$  deve essere  $\text{Int} \rightarrow \text{Int}$ . Allora  $t_1 = \text{Int}$  e  $t_6 = \text{Int}$ . Quindi  $T_0 = \text{Int} \rightarrow \text{Int}$ .

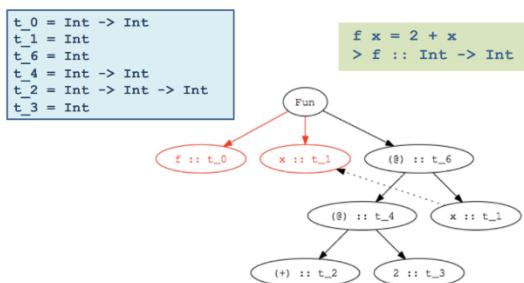
## Step 4: Solve Constraints



46

**Determine Type of Declaration:** Una volta risolti i vincoli, il compilatore determina il tipo della dichiarazione originale. Nel caso di  $f x = 2 + x$ , si determina che  $f$  è una funzione  $\text{Int} \rightarrow \text{Int}$ , cioè prende un intero come input e restituisce un intero come output.

### Step 5: Determine type of declaration



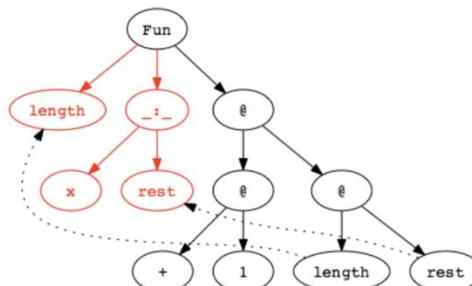
Nella gestione di tipi di dati polimorfi, una funzione può avere più clausole per gestire diverse situazioni, tramite il pattern matching. Ad esempio, la funzione length può essere definita come segue:

```
length [] = 0
length (x:rest) = 1 + (length rest)
```

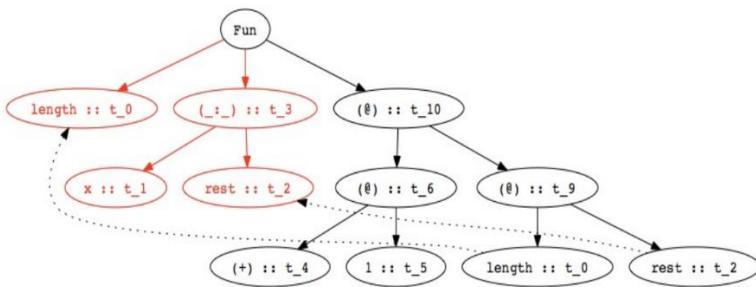
In caso di chiamate ricorsive, come in  $(\text{length rest})$ , si assume che la funzione chiamata abbia lo stesso tipo della sua definizione, rispettando la coerenza tipica durante la ricorsione.

## Type Inference with Datatypes

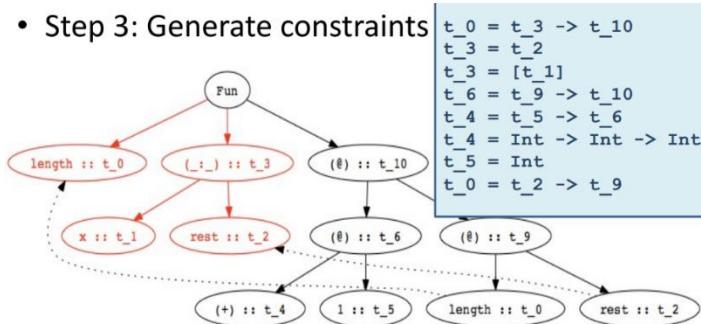
- Example:  $\text{length } (\text{x:rest}) = 1 + (\text{length rest})$
- Step 1: Build Parse Tree



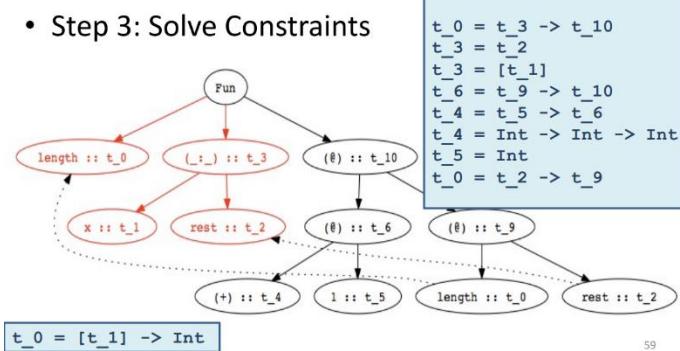
- Step 2: Assign type variables



- Step 3: Generate constraints



- Step 3: Solve Constraints



59

La **type inference** (inferenza dei tipi) in Haskell produce il tipo più generale per una funzione, chiamato anche tipo principale. Consideriamo l'esempio della funzione map, che applica una funzione a ogni elemento di una lista:

```
map (f, []) = []
map (f, x:xs) = f x : map (f, xs)
```

In questo caso, l'inferenza produce il tipo più generale:

```
map :: (t_1 -> t_2, [t_1]) -> [t_2]
```

Questo tipo principale consente alla funzione map di lavorare con vari tipi di input e output. Le funzioni possono avere anche versioni meno generali del tipo, applicabili a casi specifici.

Nell'inferenza di tipi con overloading, il compilatore infere un **tipo qualificato** nella forma  $Q \Rightarrow T$ .

**T** rappresenta il tipo della funzione.

**Q** è un insieme di predicati sulle classi di tipo, chiamato **vincolo** (constraint), che specifica le proprietà che i tipi devono avere.

Consideriamo la funzione di esempio:

```
example z xs = case xs of
  [] -> False
  (y:ys) -> y > z || (y == z && ys == [z])
```

Per questa funzione, il tipo **T** è  $a \rightarrow [a] \rightarrow \text{Bool}$ , poiché example prende un elemento  $a$  e una lista  $xs$  e restituisce un valore booleano. Il vincolo **Q** è  $\{\text{Ord } a, \text{Eq } a, \text{Eq } [a]\}$ , che significa che il tipo  $a$  deve supportare le operazioni di ordine (Ord) e uguaglianza (Eq), e anche  $[a]$  deve supportare Eq.

Per semplificare i vincoli sui tipi, noti come set di vincoli **Q**, si applicano alcune regole: Se in un set ci sono vincoli duplicati, possono essere eliminati. Ad esempio,  $(\text{Eq } a, \text{Eq } a)$  può essere ridotto a  $\text{Eq } a$ .

Se esiste una dichiarazione di istanza che implica un vincolo, possiamo semplificare. Ad esempio, se esiste l'istanza  $\text{Eq } a \Rightarrow \text{Eq } [a]$ , allora  $(\text{Eq } a, \text{Eq } [a])$  può essere ridotto a  $\text{Eq } a$ .

Se una dichiarazione di classe collega due vincoli, possiamo utilizzarla per semplificare. Ad esempio, se  $\text{Eq } a \Rightarrow \text{Ord } a$ , allora  $(\text{Ord } a, \text{Eq } a)$  può essere ridotto a  $\text{Ord } a$ .

Applicando queste regole, un set di vincoli come  $(\text{Ord } a, \text{Eq } a, \text{Eq } [a])$  può essere ridotto a  $\text{Ord } a$ . Queste semplificazioni rendono più gestibile il set di vincoli per il compilatore.

Il tipo della funzione example viene indicato come **T**, ed è  $a \rightarrow [a] \rightarrow \text{Bool}$ , il che significa che la funzione accetta un elemento  $a$ , una lista di elementi di tipo  $a$  e restituisce un  $\text{Bool}$ .

Applicando le semplificazioni sui vincoli, possiamo ridurre **Q** a  $\text{Ord } a$ , poiché  $\text{Ord}$  implica  $\text{Eq}$  in questo contesto.

Il tipo finale della funzione example, dopo aver considerato l'overloading, diventa:

```
example :: Ord a => a -> [a] -> Bool
```

Questo tipo qualifica example affinché accetti solo tipi  $a$  che appartengono alla classe  $\text{Ord}$ , permettendo così sia confronti di uguaglianza che di ordine.

## AP-19: Monads in Haskell

### La Typeclass Functor

In Haskell, **Functor** è una **typeclass** utilizzata per rappresentare strutture che possono essere **mappate**. Questo significa che puoi applicare una funzione a ogni elemento contenuto in tali strutture. Gli esempi più comuni di functor includono liste, il tipo **Maybe**, alberi e altro ancora.

La typeclass Functor è definita nello standard di Haskell nel seguente modo:

```
class Functor f where
```

```
    fmap :: (a -> b) -> f a -> f b
```

Il simbolo **f** è un costruttore di tipo che accetta un parametro di tipo. Un'istanza di Functor deve implementare la funzione **fmap**.

La funzione **fmap** prende:

1. Una funzione da un tipo  $a$  ad un tipo  $b$  ( $a \rightarrow b$ ).
2. Una struttura  $f a$  (per esempio, una lista di  $a$  o un **Maybe**  $a$ ).
3. Restituisce una struttura  $f b$  (per esempio, una lista di  $b$  o un **Maybe**  $b$ ).

Il tipo lista (`[]`) è un Functor. La definizione è:

```
instance Functor [] where
```

```
    fmap = map
```

Poiché fmap è equivalente a map per le liste, il suo comportamento è il seguente:

```
ghci> fmap (*2) [1, 2, 3]
```

```
[2, 4, 6]
```

```
ghci> fmap (*2) []
```

```
[]
```

## Leggi dei functor

I Functor devono rispettare due leggi:

**Legge dell'identità:** fmap id x == x

Mappare la funzione identità su un Functor non deve cambiarne il valore. (fmap id [1,

```
2, 3] == [1, 2, 3])
```

**Legge della composizione:**

```
fmap (f . g) x == fmap f (fmap g x)
```

Comporre due funzioni e poi mapparle è lo stesso che mappare una funzione e poi l'altra.

- Funzione `f`: (+1) (aggiunge 1 a un valore).
- Funzione `g`: (\*2) (moltiplica un valore per 2).
- Lista `x`: [1, 2, 3] (una lista di numeri interi).

```
fmap (f . g) [1, 2, 3]
== [ (f . g) 1, (f . g) 2, (f . g) 3 ]
== [ ((1 * 2) + 1), ((2 * 2) + 1), ((3 * 2) + 1) ]
== [ 3, 5, 7 ]

fmap g [1, 2, 3]
== [ g 1, g 2, g 3 ]
== [ (1 * 2), (2 * 2), (3 * 2) ]
== [ 2, 4, 6 ]

fmap f [2, 4, 6]
== [ f 2, f 4, f 6 ]
== [ (2 + 1), (4 + 1), (6 + 1) ]
== [ 3, 5, 7 ]
```

Queste proprietà garantiscono un comportamento coerente e prevedibile per tutti i Functor.

## Monad in Haskell

I **monads** sono costruttori di tipo istanze della classe Monad.

Un **Monad** estende il concetto di **Functor**, introducendo due operazioni aggiuntive, **return** e **bind**, che rendono possibile lavorare con funzioni che consentono di creare **catene di operazioni** e producono **valori boxed** come risultato, facilitando la gestione dei dati incapsulati e degli effetti collaterali (come input/output, stati, eccezioni, etc.) in modo controllato e funzionale.

## La Classe Monad

La classe Monad è definita in Haskell come:

```
class Monad m where
  return :: a -> m a
  (=>)  :: m a -> (a -> m b) -> m b -- "bind"
  (=>)  :: m a -> m b -> m b           -- "then"
```

1. **Return:** Questa operazione permette di prendere un valore e metterlo in un contenitore **detto box** (come Maybe, List, etc.).

Esempio: Se usiamo `return 5 :: [Int]` con una lista, il valore 5 verrà incapsulato in [5].

2. **Bind**: L'operatore `>>= (bind)` consente di concatenare computazioni monadiche. Se hai un **valore encapsulato** in un contesto monadico `x :: m a`; una **funzione f :: a -> m b** che prende un valore e restituisce un nuovo valore monadico.

Allora l'operazione `x >>= f` **estrae** il valore da `x`, lo passa a `f` e restituisce una nuovo valore monadico `m b`. Questo permette di collegare più operazioni in modo fluido.

```
f :: Int -> [Int]
f x = [x, x * 2]

g :: Int -> [Int]
g x = [x, x ^ 2]

result = [1, 2] >>= f >>= g
```

Applichiamo `f` a ogni elemento di `[1, 2]`:

```
haskell                               ⌂ Copia ⌁ Modifica

[1, 2] >>= f
== f 1 ++ f 2
== [1, 2] ++ [2, 4]
== [1, 2, 2, 4]
```

Prendiamo il risultato `[1, 2, 2, 4]` e applichiamo `g`:

```
haskell                               ⌂ Copia ⌁ Modifica

[1, 2, 2, 4] >>= g
== g 1 ++ g 2 ++ g 4
== [1, 1] ++ [2, 4] ++ [2, 4] ++ [4, 16]
== [1, 1, 2, 4, 2, 4, 4, 16]
```

3. **then (>>)**: Concatena due computazioni: esegue la prima, ignora il risultato e passa alla seconda. `x >> y = x >>= (\_ -> y)`. Qui, `\_ -> y` è una funzione anonima che ignora il proprio argomento e restituisce `y`.

Con `>>` (ignoro l'input)

```
haskell                               ⌂ Copia ⌁ Modifica

example3 :: IO ()
example3 = do
    putStrLn "Inserisci qualcosa (ma ignoro l'input):"
    getLine >> putStrLn "Ignorato e continuo con questa azione."
```

Esempio di Output:

```
css                               ⌂ Copia ⌁ Modifica

Inserisci qualcosa (ma ignoro l'input):
Hello
Ignorato e continuo con questa azione.
```

Con `>>=` (uso l'input)

```
haskell                               ⌂ Copia ⌁ Modifica

example4 :: IO ()
example4 = do
    putStrLn "Inserisci qualcosa (uso l'input):"
    getLine >>= (\input -> putStrLn $ "Hai scritto: " ++ input)
```

Esempio di Output:

```
css                               ⌂ Copia ⌁ Modifica

Inserisci qualcosa (uso l'input):
Hello
Hai scritto: Hello
```

## Maybe

La definizione data **Maybe a = Nothing | Just a** significa che un valore di tipo **Maybe a** può essere: **Nothing**, che indica l'assenza di un valore, o **Just a**, che rappresenta un valore presente di tipo `a`.

Una funzione con tipo `f :: a -> Maybe b` è detta **funzione parziale** perché non garantisce un output valido per ogni input; al contrario, potrebbe restituire `Nothing` se il calcolo non produce un valore.

```
max [] = Nothing
max (x:xs) = Just
    (foldr (\y z -> if y > z then y else z) x xs)
max :: Ord a => [a] -> Maybe a
```

Nell'esempio, la funzione max calcola il massimo di una lista. Se la lista è vuota, max [] restituisce Nothing. Se ci sono elementi, restituisce Just con il valore massimo.

L'operatore bind ( $>=$ ) consente di comporre funzioni parziali e gestire automaticamente i casi di indefinitezza.

### Maybe come istanza di Monad

```
class Monad m where
    return :: a -> m a
    (>=)   :: m a -> (a -> m b) -> m b -- "bind"
    ... -- + something more
```

- **m** is a type constructor
- **m a** is the type of **monadic values**

```
instance Monad Maybe where
    return :: a -> Maybe a
    return x = Just x
    (>=)   :: Maybe a -> (a -> Maybe b) -> Maybe b
    y >= g = case y of
        Nothing -> Nothing
        Just x -> g x
```

La funzione bind per Maybe su un valore Maybe y e una funzione g, funziona nel seguente modo: Verifica il tipo di valore contenuto in y. Se il corrispondente valore x è Nothing, il risultato dell'intera espressione è Nothing. Altrimenti applica la funzione g al valore estratto tramite Just, x.

### Sintassi do

La sintassi **do** in Haskell è uno zucchero sintattico per semplificare le espressioni con  $>=$ .

```
bothGrandfathers p =
    father p >=
        (\dad -> father dad >=
            (\gf1 -> mother p >=
                (\mom -> father mom >=
                    (\gf2 -> return (gf1,gf2) ))))
```

```
bothGrandfathers p = do {
    dad <- father p;
    gf1 <- father dad;
    mom <- mother p;
    gf2 <- father mom;
    return (gf1, gf2);
}
```

```
bothGrandfathers p = do
    dad <- father p
    gf1 <- father dad
    mom <- mother p
    gf2 <- father mom
    return (gf1, gf2)
```

### Some Haskell Monads

Monad	Imperative semantics
Maybe	Exception (Anonymous)
Error	Exception (with error description)
State	Global state
IO	Input/output
[] (lists)	Non-determinism
Reader	Environment
Writer	Logger

### Implementazione di bind ( $>=$ )

Illustriamo come implementare l'operazione di bind ( $>=$ ) per le liste usando operazioni di base come map, return e concat.

**map**: Applica una funzione a ciascun elemento di una lista e restituisce una lista dei risultati.  $\text{map} :: (\text{a} \rightarrow \text{b}) \rightarrow [\text{a}] \rightarrow [\text{b}]$

**return**: Incapsula un singolo valore in una lista.  $\text{return } x = [x]$  -- produce una lista con un solo elemento

**concat**: Unisce una lista di liste in una lista semplice, "appiattendo" la struttura.  $\text{concat } [[1,2],[3],[4]] = [1,2,3,4]$

L'operatore `>=` viene definito per una lista come:

```
(>=) :: [a] -> (a -> [b]) -> [b]  
xs >= f = concat (map f xs)
```

map f xs applica la funzione f a ciascun elemento x della lista xs, producendo una lista di liste. concat appiattisce questa lista di liste in una lista unica, concatenando i risultati.

Esempio: Sia f :: Int -> [Int], f x = [x, x \* 2]. map f [1, 2, 3] == [[1, 2], [2, 4], [3, 6]]. concat [[1, 2], [2, 4], [3, 6]] == [1, 2, 2, 4, 3, 6]

## Problemi della Programmazione Puramente Funzionale

La programmazione funzionale pura è caratterizzata dall'assenza di effetti collaterali. Tuttavia, per costruire programmi che siano realmente utili e possano interagire con il mondo esterno, è necessario introdurre **funzionalità impure**, come il supporto all'IO, alla modifica dello stato di una variabile, alla gestione delle eccezioni, all'interagire con altri linguaggi e a gestire la concorrenza.

Un programma deve poter interagire con l'ambiente esterno. L'**approccio diretto** consiste nell'**aggiungere costrutti imperativi** al linguaggio in modo tradizionale, rendendolo capace di gestire operazioni con effetti collaterali.

### I/O tramite funzioni con effetti collaterali

```
putchar 'x' + putchar 'y'
```

Questa operazione coinvolge l'uso di funzioni che, oltre a restituire un valore, hanno effetti collaterali come la stampa di caratteri su uno schermo.

### Operazioni imperative con celle di riferimento assegnabili:

```
z = ref 0; (* z è una cella di riferimento inizializzata a 0 *)  
z := !z + 1; (* Assegna a z il valore incrementato di 1 *)  
f(z); (* Chiama la funzione f con z *)  
w = !z; (* Legge il valore di z *)
```

L'uso delle celle di riferimento permette di mantenere e modificare lo stato. La domanda "Qual è il valore di w?" suggerisce che l'ordine di valutazione può influire sul risultato.

Questo approccio funziona solo se il linguaggio determina l'ordine di valutazione delle espressioni. Nei linguaggi funzionali *lazy* come Haskell, l'ordine di valutazione delle espressioni non è definito. Il programma valuta solo ciò che è strettamente necessario, il che può creare comportamenti inattesi quando si tratta di operazioni con effetti collaterali.

**res = putchar 'x' + putchar 'y':** L'output dipende dall'ordine di valutazione dell'operatore `+`. In un linguaggio *lazy*, non si può sapere a priori se `putchar 'x'` o `putchar 'y'` verrà valutato per primo, quindi l'ordine in cui i caratteri vengono stampati è imprevedibile.

**ls = [putchar 'x', putchar 'y']:** L'output dipende da come viene utilizzata la lista ls.

Se la lista viene usata solo nella funzione `length`, ad esempio `length ls`, nulla sarà stampato, perché `length` non valuta gli elementi della lista, ma semplicemente ne conta il numero.

# Effetti collaterali e I/O in Haskell con i Monad

In Haskell, è possibile aggiungere caratteristiche imperative senza alterare il significato delle espressioni pure grazie ai **Monad**. In particolare, il monad **IO** consente di definire valori monadici (azioni) e specifica come comporli sequenzialmente, gestendo in modo funzionale gli effetti collaterali come l'I/O.

## Modelli pre-Monad per l'I/O

Prima dell'introduzione dei Monad, i linguaggi funzionali utilizzavano modelli come **stream** e **continuazioni** per gestire l'I/O e gli effetti collaterali:

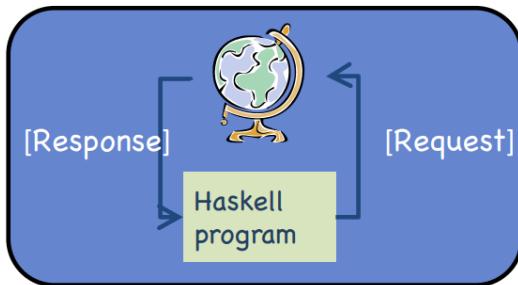
**Stream:** L'I/O veniva trattato come un flusso continuo di dati tra programma e sistema operativo. Il programma inviava richieste al sistema operativo e riceveva risposte.

**Continuazioni:** Gli utenti passavano funzioni (continuazioni) alle routine di I/O per elaborare i risultati, rendendo il flusso più flessibile.

Il **Rapporto Haskell 1.0** adottò il modello di **stream**, dimostrando però che stream e continuazioni erano interdefinibili, ovvero esprimibili l'uno con l'altro.

## Il Modello di Stream

Nel modello di stream, la funzione **main** aveva il tipo **main :: [Response] -> [Request]**, separando gli effetti collaterali dalla logica funzionale.

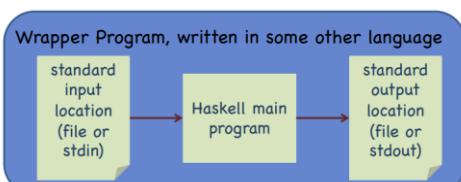


La funzione main prende in input una lista di Response perché rappresentano i risultati delle operazioni precedenti. Ogni Response fornisce il contesto o lo stato necessario per determinare quali Request successive devono essere generate.

La funzione main restituisce una lista di Request perché il suo compito è descrivere le **operazioni future** (e.g. read/write) basandosi sulle risposte precedenti.

```
data Request = ReadFile Filename  
             | WriteFile Filename String  
             | ...  
  
data Response = RequestFailed  
              | ReadOK String  
              | Writeok  
              | Success  
              | ...
```

Un **programma wrapper**, scritto in un **altro linguaggio**, interpretava le richieste, **eseguiva le operazioni** e forniva le risposte, mantenendo il programma Haskell puramente funzionale.



Questo wrapper **gestiva l'I/O** leggendo l'input (ad esempio, da stdin), passandolo al programma Haskell e scrivendo l'output (ad esempio, su stdout).

Tuttavia, il modello presentava diverse limitazioni: A causa della valutazione lazy, potevano essere emesse richieste prima di ricevere le risposte necessarie, causando comportamenti imprevisti. Non vi era un collegamento diretto tra una richiesta e la sua risposta, aumentando il rischio di disallineamenti e deadlock. Ogni nuova operazione di I/O richiedeva modifiche sia ai tipi Request e Response sia al programma wrapper, aumentando la complessità. Non era possibile combinare facilmente più funzioni main, limitando modularità e riusabilità.

## Superamento dei Limiti con i Monad

### Monadic I/O: Concetti Chiave

In Haskell, il tipo **IO** è un **costruttore di tipo**, istanza della classe **Monad**. Rappresenta "azioni" che possono eseguire operazioni di input/output prima di restituire un risultato di tipo t.

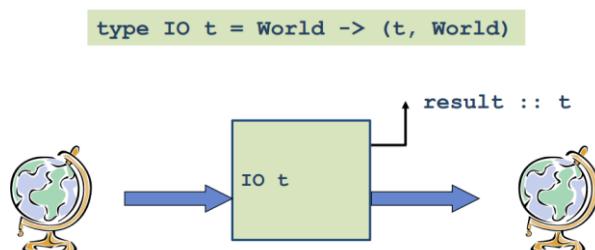
**Azioni di tipo IO t:** Sono valori di prima classe che descrivono **operazioni I/O**. La **valutazione** di un'azione non produce effetti collaterali; solo l'**esecuzione effettiva** li genera.

**Costrutti fondamentali: return:** Incapsula un valore in un contesto IO senza eseguire alcuna operazione di I/O.

**bind (>=):** Permette di comporre azioni sequenziali collegando una computazione a una funzione che restituisce un'altra azione.

**then (>>):** Comporta due azioni sequenziali, ignorando il risultato della prima.

Un'azione **può essere eseguita** solo quando viene chiamata da main :: IO () .



Il tipo **IO t** può essere interpretato come una funzione che modifica lo stato del mondo:  
**type IO t = World -> (t, World)**

**type IO t = World -> (t, World)** significa che **un'azione di tipo IO t** è una **funzione** che prende in **input** lo **stato del mondo** (l'ambiente esterno), **completa operazioni** e restituisce in **output** un **risultato di tipo t** insieme al **nuovo stato del mondo**.

Non si può "saltare" un'azione o riordinare le operazioni, perché ognuna **dipende** dalla **versione aggiornata del mondo** generata dall'**azione precedente**. Questo garantisce la sequenzialità.

Le azioni di tipo IO possono essere passate come argomenti, restituite da funzioni e combinate con altre azioni.

## Purezza di haskell mantenuta nonostante l'I/O

In Haskell, le funzioni **non hanno effetti collaterali** visibili all'esterno.

L'IO Monad trasforma invece **un'azione** potenzialmente **impura** in un **valore di tipo IO** a, che rappresenta **la descrizione** di quell'effetto, ma **non lo esegue** immediatamente.

**Mantenimento purezza:** L'esecuzione degli effetti avviene solo quando il programma "consegna" il controllo al runtime (tipicamente alla fine, in main), garantendo che tutto il resto del codice rimanga puro.

**Aggirare problema laziness:** Invece di lasciare che le chiamate a funzioni con effetti collaterali possano avvenire in qualsiasi ordine, Haskell impone la costruzione di una catena di azioni, tramite `>>= bind`, stabilita dal programmatore.

**Gestione variabili:** Per la gestione delle variabili, l'uso di IORef o di altre strutture mutabili rimane confinato dentro il tipo IO, evitando che la "mutabilità" contami il resto del programma, che resta puramente funzionale. Questo modello non può essere rotto se non con l'operazione unsafePerform.

**Isolamento grazie al concetto di World:** Attraverso l'uso dei monad, il codice che utilizza la monad IO dichiara come interagire col mondo esterno, ma la logica funzionale resta "pulita, perché gli effetti sono limitati al di fuori della parte puramente funzionale del linguaggio.

## Sequenzialità garantita da Word

Ogni azione di IO riceve in ingresso lo stato globale del **mondo esterno** e ne produce **uno nuovo** in uscita modificato in base all'effetto dell'azione, che viene passato implicitamente all'azione successiva.

Non si può "saltare" un'azione o riordinare le operazioni, perché ognuna dipende dalla **versione aggiornata del mondo** generata dall'**azione precedente**. Questo garantisce la sequenzialità.

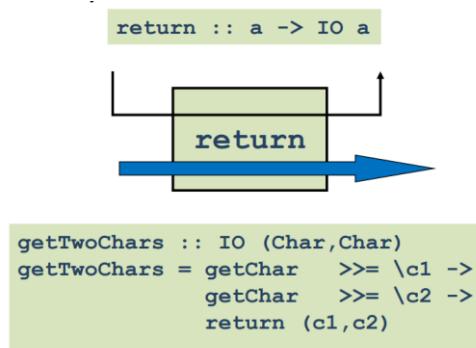
## Implementazione delle operazioni principali di Monad IO

### Il Combinatore return in Haskell

Il combinatore return **incapsula un valore puro in un contesto monadico** senza eseguire alcuna operazione. `return :: a -> IO a`

**a:** Un valore puro, privo di effetti collaterali. **IO a:** Il valore a encapsulato in un contesto IO.

Il combinatore return è spesso utilizzato come **ultimo passo** in una catena di azioni monadiche per restituire un risultato. Facilita la gestione di valori intermedi all'interno di un contesto monadico.



**La funzione getTwoChars** legge il primo carattere e lo assegna a c1, legge il secondo carattere e lo assegna a c2. Con **return (c1, c2)** combina i due caratteri in una coppia (c1, c2) e li restituisce nel contesto IO.

### Implementazione

```
return :: a -> IO a
return a = \w -> (a, w)
```

Prende un **valore a** e restituisce una **funzione** che **accetta il "mondo"** e restituisce a insieme al "mondo" invariato.

## Il Combinatore $\geq$ (Bind)

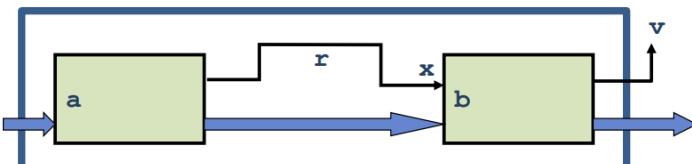
L'operatore  $\geq$  (bind) permette di **comporre azioni sequenziali**, collegando il risultato di una computazione al passo successivo.

$(\geq) :: IO a \rightarrow (a \rightarrow IO b) \rightarrow IO b$

**IO a:** Una computazione che restituisce un valore di tipo a nel contesto IO.

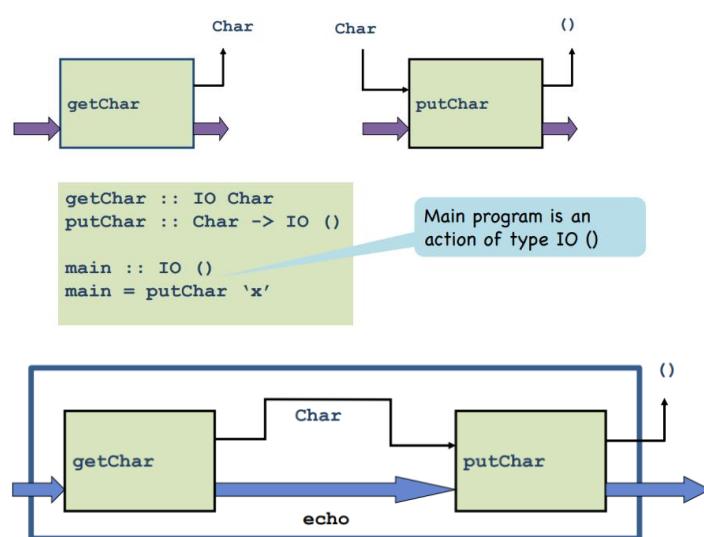
**(a  $\rightarrow$  IO b):** Una funzione che prende un valore di tipo a e restituisce una nuova computazione di tipo IO b.

**IO b:** Il risultato della composizione delle due computazioni.



Quando viene eseguita l'azione  $a \geq \lambda x \rightarrow b$ , il combinatore  $\geq$  esegue i seguenti passaggi: **L'azione a (getChar)** viene eseguita, restituendo un **valore r** (carattere letto). Il **risultato r** è passato alla **funzione anonima  $\lambda x \rightarrow b$  (putchar)**, che prende in input il risultato della precedente computazione e restituisce una nuova computazione. La nuova computazione **b, basata su r, viene eseguita** (putchar sull'output di getChar). Il valore finale della seconda computazione (v) viene restituito.

### Simple I/O actions



## Implementazione

```
(>=) :: IO a -> (a -> IO b) -> IO b  
>= action nextAction = \world -> case action world of  
  (result, newWorld) -> nextAction result newWorld
```

**Esecuzione della prima azione (action):** L'azione **action** viene **applicata** sul mondo corrente (**world**) tramite il comando (**case**) **action world** e produce un risultato e uno stato del mondo aggiornato. Queste vengono destrutturate in (**result**, **newWorld**). **Result** e **newWorld** vengono passati a **nextAction** **in modo esplicito** tramite l'operatore **->** che consente di prenderli come parametri lambda. L'azione **nextAction** viene eseguita su **result** e sul nuovo stato del mondo (**newWorld**).

```
main :: IO ()  
main = getLine >= (\name -> putStrLn ("Ciao, " ++ name ++ "!"))
```

Esempio: **action** è **getLine**, che legge una stringa (**name**) dall'input dell'utente. Produce **result** = la stringa presa da **getLine** (es. "Alice") e **newWorld** = stato del mondo dopo aver

letto l'input. L'azione nextAction è la funzione anonima `\name -> putStrLn ("Ciao, " ++ name ++ "!")`. Prende il risultato (name) dell'operazione precedente e restituisce una nuova computazione IO che stampa "Ciao, Alice!".

`>>= collega getLine e putStrLn, garantendo che il risultato della prima azione venga passato alla seconda.`

## Il Combinatore `>>(Then)`

Il **combinatore `>> (then)`** è usato per eseguire due azioni IO **in sequenza** quando **non** c'è bisogno di **passare un valore** dalla prima alla seconda azione. La sua firma è: `(>>) :: IO a -> IO b -> IO b`

L'operatore `>>` è definito in termini di `>>=: m >> n = m >>= (\_ -> n)`.

Qui, m viene eseguito per primo, il suo risultato è ignorato (`\_ ->`), e poi n viene eseguito.

### Esempio

**echoDup:** Un'azione IO che legge un carattere e lo stampa due volte.

```
echoDup :: IO ()  
echoDup = getChar >>= \c -> putStrLn c >> putStrLn c
```

getChar legge un carattere c. `putChar c >> putChar c` stampa c due volte in sequenza.

**echoTwice:** Un'azione IO che esegue due volte l'azione echo. Esegue l'azione echo, poi la esegue di nuovo, ignorando i risultati intermedi.

```
echoTwice :: IO ()  
echoTwice = echo >> echo
```

## La Notazione do in Haskell

La **notazione do** è uno zucchero sintattico che rende il codice monadico simile a uno stile imperativo. Consente di scrivere sequenze di operazioni monadiche in modo lineare: Ogni riga corrisponde a un'azione. L'assegnazione dei risultati avviene tramite l'operatore `<-`.

```
-- Plain Syntax  
getTwoChars :: IO (Char,Char)  
getTwoChars = getChar >>= \c1 ->  
              getChar >>= \c2 ->  
              return (c1,c2)
```

```
-- Do Notation  
getTwoCharsDo :: IO (Char,Char)  
getTwoCharsDo = do { c1 <- getChar ;  
                   c2 <- getChar ;  
                   return (c1,c2) }
```

```
do { x }      = x  
do { x; stmts } = x >> do { stmts }  
do { v<-x; stmts } = x >>= \v -> do { stmts }  
do {let ds; stmts } = let ds in do { stmts }
```

Le variabili assegnate con `<-` sono visibili solo nel resto del blocco do:

### Esempio Complesso: `getLine`

La funzione `getLine` legge una linea di input carattere per carattere:

```
getLine :: IO [Char]
getLine = do { c <- getChar ;
              if c == '\n' then
                return []
              else
                do { cs <- getLine;
                     return (c:cs) }}
```

La funzione `getChar` legge un carattere. Se il carattere è `\n`, termina restituendo una lista vuota. Altrimenti, richiama ricorsivamente `getLine` per leggere il resto della linea e combina il carattere letto con il resto.

### Strutture di Controllo con i Monad in Haskell

I Monad in Haskell permettono di definire strutture di controllo generiche, sfruttando i combinatori monadici (`>>`, `>>=`) per eseguire operazioni in modo sequenziale e controllato.

#### Esempio 1: Esecuzione Ripetuta con `repeatN`

La funzione `repeatN` esegue un'azione monadica un numero specificato di volte:

```
repeatN 0 x = return ()
repeatN n x = x >> repeatN (n-1) x
repeatN :: (Num a, Monad m, Eq a) => a -> m a1 -> m ()
```

```
Main> repeatN 5 (putChar 'h')
```

**Caso base (n = 0):** Quando il numero di ripetizioni è zero, la funzione restituisce `return ()`.

**Caso ricorsivo (n > 0):** L'azione `x` viene eseguita e concatenata con la chiamata ricorsiva `repeatN (n-1) x`, grazie al combinatore `>>`.

Questo esempio stampa la lettera 'h' cinque volte.

#### Esempio 2: Iterazione su Liste con `for`

La funzione `for` applica un'azione monadica a ogni elemento di una lista:

```
for []      fa = return ()
for (x:xs)  fa = fa x  >>  for xs fa
for :: Monad m => [t] -> (t -> m a) -> m ()
```

```
Main> for [1..10] (\x -> putStrLn (show x))
```

**Caso base (lista vuota):** Quando la lista è vuota, restituisce `return ()`.

**Caso ricorsivo:** Applica l'azione `fa` all'elemento corrente `x` e procede ricorsivamente sugli elementi rimanenti (`xs`).

In questo caso, la funzione stampa i numeri da 1 a 10.

### Sequenzializzazione di Azioni con `sequence`

La funzione `sequence` converte una **lista di azioni monadiche** in un'unica azione che **restituisce una lista di risultati**:

```
-- sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a:as) = do { r <- a;
                      rs <- sequence as;
                      return (r:rs) }
sequence :: Monad m => [m a] -> m [a]
```

ample use:

```
Main> sequence [getChar, getChar, getChar]
```

**Caso base (lista vuota):** Restituisce return [], un'azione che incapsula una lista vuota.

**Caso ricorsivo:** Esegue l'azione a, raccoglie il risultato r, e combina ricorsivamente i risultati delle altre azioni (as) in una lista.

```
Main> sequence [getChar, getChar, getChar]
```

Input abc ===> ['a','b','c']

Legge tre caratteri consecutivi dall'input e li restituisce come lista.

## Gestione dei File e Riferimenti con il Monad IO in Haskell

Il **Monad IO** di Haskell offre un'ampia collezione di funzioni per la gestione dei file e per lavorare con riferimenti mutabili.

### Accesso ai File

Il Monad IO fornisce analogie dirette con le funzioni standard della libreria C per i file.

**openFile:** Apre un file specificando il percorso (FilePath) e la modalità (IODevice), restituendo un Handle per accedere al file.

openFile :: FilePath -> IOMode -> IO Handle

**hPutStr:** Scrive una stringa in un file usando un Handle.

hPutStr :: Handle -> String -> IO ()

**hGetLine:** Legge una linea di testo da un file usando un Handle.

hGetLine :: Handle -> IO String

**hClose:** Chiude un file aperto.

hClose :: Handle -> IO ()

### Riferimenti Mutabili con IORef

Il tipo **IORef** consente di lavorare con **celle di memoria mutabili** in un contesto IO, simulando il comportamento di **variabili mutabili**.

**Tipologia di IORef:**

data IORef a -- Tipo astratto

**Funzioni principali:**

**newIORef**: Crea un nuovo riferimento inizializzato con un valore.

newIORef :: a -> IO (IORef a)

**readIORef**: Legge il valore contenuto in un riferimento.

readIORef :: IORef a -> IO a

**writeIORef**: Scrive un nuovo valore in un riferimento.

writeIORef :: IORef a -> a -> IO ()

### Esempio: Somma dei primi n numeri interi:

1. Inizializza un IORef a 0.
2. Incrementa iterativamente il valore contenuto nel riferimento fino a sommare tutti gli interi da 1 a n.

```
import Data.IORef -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
  { r <- newIORef 0;
    addToN r 1 }
  where
    addToN :: IORef Int -> Int -> IO Int
    addToN r i | i > n      = readIORef r
               | otherwise = do
                  { v <- readIORef r
                    ; writeIORef r (v + i)
                    ; addToN r (i+1)}
```

La funzione **principale count** accetta un intero n e restituisce una computazione IO che produce un Int. **newIORef 0**: Crea un riferimento mutabile inizializzato al valore 0. Questo riferimento (r) conterrà la somma parziale durante il calcolo. **addToN r 1**: Inizia il calcolo ricorsivo passando il riferimento r e il valore iniziale 1.

La funzione ausiliaria **addToN** addToN accetta un riferimento IORef Int e un intero i. Restituisce una computazione IO che produce un Int.

**Parametri:** r: Il riferimento che contiene la somma parziale. i: L'indice corrente, che rappresenta il numero da aggiungere alla somma.

**Caso base (i > n):** Quando i supera il valore massimo n, la funzione termina leggendo e restituendo il valore finale contenuto nel riferimento r. Questo corrisponde al risultato della somma dei primi n numeri interi.

**Caso ricorsivo (i <= n):** readIORef r: Legge il valore attuale della somma parziale.

writeIORef r (v + i): Aggiorna il riferimento aggiungendo i alla somma parziale. addToN r (i + 1): Prosegue ricorsivamente con il prossimo numero (i + 1).

### Vincoli del Monad IO

In Haskell puro, **non è possibile trasformare un valore di tipo IO a in un valore puro di tipo a**.

Questo vincolo è fondamentale per garantire la separazione tra codice puro e operazioni che coinvolgono effetti collaterali.

Se una funzione restituisce un IO String, non puoi direttamente "estrarre" la String per usarla in un contesto puro.

## Problema: Leggere un File di Configurazione

Supponiamo di voler leggere un file di configurazione e utilizzare i suoi contenuti come una lista di stringhe.

```
configFileContents :: [String]  
configFileContents = lines (readFile "config") -- ERRORE
```

La funzione `readFile` restituisce un valore di tipo `IO String`, che non può essere convertito direttamente in una `String` pura.

## Opzioni per affrontare il problema

**Eseguire l'intero programma nel contesto IO:** Scrivere tutto il programma in stile monadico, sacrificando la purezza delle funzioni che dipendono dai dati letti.

**Rompere il vincolo di purezza:** Utilizzare la funzione `unsafePerformIO` per "uscire" dal contesto `IO`, **convertendo IO a in a**. Questo è non permesso in Haskell puro perché comprometterebbe la garanzia di purezza e trasparenza referenziale.

## unsafePerformIO: Una Scorciatoia Pericolosa

`unsafePerformIO` permette di "uscire" dal contesto `IO`, convertendo **un valore IO a in un valore puro a**. Il suo utilizzo viola intenzionalmente la separazione tra puro e impuro, rompendo le garanzie del sistema di tipi.

### unsafePerformIO :: IO a -> a

Supponiamo di voler leggere un file di configurazione che non cambia. Qui, `unsafePerformIO` converte l'azione `IO` della lettura del file in una lista pura di stringhe.

```
unsafePerformIO :: IO a -> a  
configFileContents :: [String]  
configFileContents = lines (unsafePerformIO (readFile "config"))
```

L'utilizzo di `unsafePerformIO` rompe la "soundness" del sistema di tipi. Ad esempio, un programma può diventare imprevedibile se l'azione `IO` non è effettivamente immutabile.

Quando si usa `unsafePerformIO`, il programmatore promette implicitamente al compilatore che: L'operazione è puramente funzionale dal **punto di vista logico**. Il momento della sua esecuzione **non influisce** sul comportamento complessivo.

`unsafePerformIO` può introdurre effetti collaterali impliciti, difficili da rilevare e debug.

## Esempi di Problemi

```
r = unsafePerformIO (newIORRef (error "urk"))  
r :: IORRef a    -- Type of the stored value is generic  
  
cast x = unsafePerformIO (do {writeIORRef r x;  
                           readIORRef r      })  
> :t (\x -> cast x)  
(\x -> cast x) :: a1 -> a2  
> cast 65:: Char  
'A'
```

Questo codice usa `unsafePerformIO` per manipolare un riferimento mutabile in modo non sicuro, consentendo di "cambiare" il tipo di una variabile. Qui si usa `unsafePerformIO` per creare un riferimento `IO` (`IORRef`) in modo non sicuro. Il valore iniziale del riferimento è un errore (`error "urk"`), ma non è immediatamente valutato perché Haskell è lazy. Il tipo del riferimento `r` è generico (`IORRef a`), il che significa che il tipo effettivo non è specificato. La funzione `cast` usa `unsafePerformIO` per scrivere un valore `x` nel riferimento `r` e poi leggerlo. Questo approccio sfrutta la genericità del tipo di `r`, consentendo di "convertire" un valore di tipo `a1` in uno di tipo `a2`. L'espressione `\x -> cast x` indica che `cast` può essere usata per convertire qualsiasi tipo (`a1`) in un altro tipo

arbitrario (a2). Come test il numero 65 viene convertito nel carattere corrispondente ('A'), sfruttando la rappresentazione interna dei dati.

## Riepilogo sui Monadi in Haskell

Un programma Haskell completo è un'unica azione IO chiamata **main**. All'interno del contesto IO, il codice è **single-threaded**, cioè segue una sequenza deterministica di operazioni. Le azioni IO più grandi vengono costruite **combinando azioni più piccole**.

Ciò avviene utilizzando: **bind ( $>=$ )**: per concatenare le azioni in modo sequenziale e **return**: per convertire il codice puro in un'azione IO.

Le azioni IO sono **first-class citizens**, il che significa che possono: Essere passate come argomenti a funzioni. Essere restituite da funzioni. Essere archiviate in strutture dati. Questa proprietà consente di definire nuovi combinatori per creare sequenze personalizzate di azioni.

Il Monad IO permette a Haskell di mantenere la **purezza funzionale** gestendo in modo efficiente gli effetti collaterali. Fornisce un modo per modellare gli effetti collaterali (come input/output) senza compromettere la trasparenza referenziale.

Il sistema di tipi di Haskell garantisce che il codice puro e il codice con effetti collaterali siano rigorosamente separati. Questo approccio facilita il ragionamento sul programma e ne aumenta la prevedibilità e la sicurezza.

---

## Conclusione

Il Monad IO rappresenta un equilibrio tra la purezza funzionale di Haskell e la necessità di gestire effetti collaterali. La sua progettazione consente di combinare azioni in modo flessibile e sicuro, preservando il rigore del sistema di tipi del linguaggio.

In Haskell, il programmatore ha la libertà di scegliere quando operare nella monade IO e quando lavorare nel contesto del codice puro e funzionale. Quindi, non è Haskell che manca di funzionalità imperative, ma piuttosto gli altri linguaggi che mancano della capacità di distinguere in modo statico un sottoinsieme puro del codice.

## Appendix: Monad Laws

```
1) return x >= f = f x
2) m >= return = m
3) (x >= f) >= g = x >= (\v -> f v >= g)
```

- In do-notation:

```
1) do { w <- return v; f w }
      = do { f v }

2) do { v <- x; return v }
      = do { x }

3) do { x <- m1;
        y <- m2;
        m3 }           =   do { y <- do { x <- m1;
                                         m2 };
                                m3 }
```

## Derived Laws for ( $>>$ ) and done

```
(>>) :: IO a -> IO b -> IO b
m >> n = m >= (\_ -> n)
```

```
done :: IO ()
done = return ()
```

```
done >> m          = m
m >> done          = m
m1 >> (m2 >> m3) = (m1 >> m2) >> m3
```

# AP-18: Lambdas and Streams in Java 8

Java 8 introduce due principali estensioni linguistiche: le **lambda expressions** e lo **Stream API**.

**Lambda Expressions:** Funzionalità che consente di definire funzioni anonime in modo conciso. La sintassi è: **(parametri) -> { corpo }**.

Le lambda expressions sono utilizzate principalmente per implementare **interfacce funzionali** (interfacce con un unico metodo astratto), eliminando la necessità di creare classi anonime verbose.

Le funzioni anonime sono definite **senza un nome**, e vengono utilizzate spesso per passare **comportamenti** come **argomenti di metodo**.

**Method References:** Sono una funzionalità che consente di fare riferimento a metodi già esistenti in modo conciso, tramite la sintassi `ClassName::methodName`.

**Inferenza dei Tipi:** Consente al compilatore Java di dedurre i tipi dei parametri basandosi sul contesto in cui la lambda è usata.

```
// Lambda con inferenza dei tipi
List<String> filteredNames = names.stream()
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList());
```

Lo **Stream API** consente la manipolazione e l'elaborazione di dati in modo funzionale, con operazioni come filter, map e reduce.

Le lambda expressions permettono di trattare il comportamento come un dato, consentendo di passare **funzioni come argomenti a metodi**. Inoltre, favoriscono la lazy evaluation tramite l'integrazione con lo **Stream API**, che ritarda l'esecuzione delle operazioni fino al momento in cui i dati sono effettivamente richiesti.

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> System.out.println(x));
```

**x -> System.out.println(x):** Funzione con parametro x (il singolo elemento della lista), che esegue l'istruzione `System.out.println(x)`.

**Parametri:** I parametri della lambda sono definiti prima del simbolo `->`. Se ci sono più parametri, devono essere racchiusi in parentesi: `(x, y) -> ....`

**Corpo:** Il corpo della lambda è definito dopo `->` e può essere un'istruzione singola o un blocco di codice racchiuso tra parentesi graffe `{}` se ci sono più istruzioni.

La riga `intSeq.forEach(x -> f(x));` scorre tutta la lista e applica ad ogni elemento la funzione `f()`.

```
// equivalent syntax
intSeq.forEach(Integer x) -> System.out.println(x);

intSeq.forEach(x -> {System.out.println(x);});

intSeq.forEach(System.out::println); //method reference
```

```

List<Integer> intSeq = Arrays.asList(1,2,3);
// multiline: curly brackets necessary
intSeq.forEach(x -> {
    x += 2;
    System.out.println(x);
});

// local variable declaration
intSeq.forEach(x -> {
    int y = x + 2;
    System.out.println(y);
});

// no new scope!!!
int x = 0;
intSeq.forEach(x -> {           //error: x already defined
    System.out.println(x + 2);
});

```

Le lambda non creano un nuovo scope per le variabili esterne, quindi dichiarare una variabile x all'esterno e utilizzarla nella lambda genera un conflitto perché x è già definita.

```

public class LVCEexample {      // local variable capture
    public static void main(String[] args) {
        List<Integer> intSeq = Arrays.asList(1,2,3);

        int var = 10;          // must be [effectively] final
        intSeq.forEach(x -> System.out.println(x + var));
        // var = 3; // uncommenting this line it does not compile
    }
}

```

Le variabili locali utilizzate successivamente all'interno di una lambda devono essere costanti *final*.

```

public class SVCExample {      // static variable capture
    private static int var = 10;
    public static void main(String[] args) {
        List<Integer> intSeq = Arrays.asList(1,2,3);

        intSeq.forEach(x -> System.out.println(x + var));
        var = 3;           // it compiles
    }
}

```

Le variabili statiche, come var, possono essere catturate e modificate nella lambda senza problemi, e il codice compila correttamente.

Le **lambda expressions** in Java possano essere assegnate a variabili, passate come argomenti e restituite da metodi.

```

// lambda expressions can be assigned and returned
import java.util.*;
import java.util.function.*;
class Lambdas {
    public static void main(String... args){
        Arrays.asList(args).forEach(print());
    }

    public static Consumer<String> print(){
        Consumer<String> cons = (y -> System.out.println(y));
        return cons;
    }
}

```

## Closures

Una **closure** è una funzione che "cattura" le **variabili del contesto** in cui è stata definita, consentendole di accedere e utilizzare queste variabili anche **dopo** che il contesto in cui erano state dichiarate **non esiste più** (ad esempio, la funzione esterna è

terminata). Questo comportamento è utile in scenari come Funzioni passate come argomenti e Funzioni restituite da altre funzioni.

In Java, le closures si manifestano principalmente attraverso l'uso di **lambda expressions**. Per catturare il contesto, le variabili locali utilizzate all'interno della lambda devono essere "effettivamente finali", cioè non devono essere modificate dopo la loro assegnazione.

```
int factor = 2; // Variabile catturata (effettivamente finale)

// Closure: Una Lambda che utilizza 'factor'
Function<Integer, Integer> multiplier = x -> x * factor;

// Utilizzo della closure
System.out.println(multiplier.apply(5)); // output: 10
System.out.println(multiplier.apply(10)); // output: 20
```

## Compilazione delle Lambda

Quando il compilatore Java incontra una lambda expression, la converte concettualmente in una **funzione** (metodo statico o di istanza). Il codice della lambda viene compilato come una funzione separata.

`x -> System.out.println(x)` potrebbe essere trasformato in un metodo generato dal compilatore:`public static void genName(Integer x) { System.out.println(x);}`

Il compilatore genera il codice necessario per chiamare questa funzione compilata ovunque venga utilizzata la lambda.

## Compilazione delle lambda basate su interfacce funzionali

Il metodo generato da una lambda dipende dal contesto in cui la lambda è utilizzata. Il compilatore Java genera un'implementazione della lambda come un'**istanza di un'interfaccia funzionale**. Il metodo generato viene richiamato attraverso un'interfaccia funzionale. Quando la lambda è passata o utilizzata, il metodo della sua interfaccia viene invocato.

Una **functional interface** è un'interfaccia Java che contiene **esattamente un metodo astratto**. Queste interfacce sono progettate per rappresentare un singolo comportamento che può essere implementato da una lambda expression o da un riferimento a metodo.

```
public interface Comparator<T> { //java.util
    int compare(T o1, T o2);
}

public interface Runnable { //java.lang
    void run();
}

public interface Consumer<T>{ //java.util.function
    void accept(T t)
}

public interface Callable<V> { //java.util.concurrent
    V call() throws Exception;
}
```

Una lambda può essere assegnata a una variabile di tipo **functional interface** o a un **parametro formale** di un metodo che accetta una functional interface.

`Runnable r = () -> System.out.println("Running!");`  
`list.forEach(s -> System.out.println(s)); // `forEach` accetta un `Consumer<T>``

Il compilatore **deduca automaticamente** i tipi di argomenti e del valore di ritorno della lambda basandosi sulla firma del **metodo astratto** della **functional interface**.

La **firma della lambda** (tipi di parametri e ritorno) deve corrispondere alla firma del **metodo astratto** nella functional interface. Se una lambda implementa un Comparator<Integer>, il metodo astratto compare richiede due parametri Integer e restituisce un int: Comparator<Integer> comparator = (a, b) -> a - b;

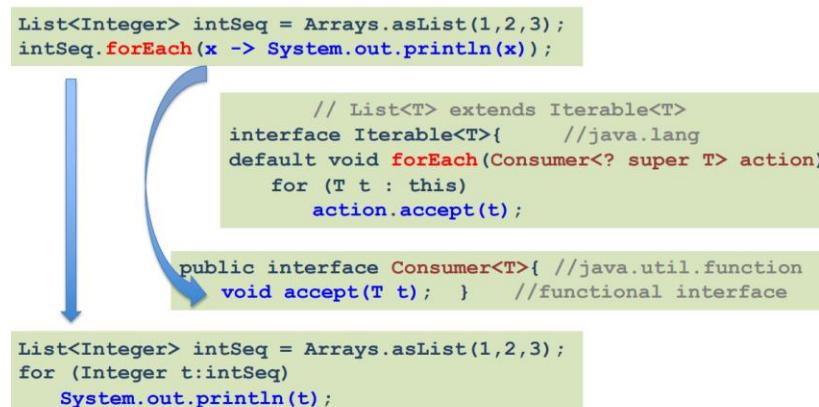
La lambda viene **invocata** richiamando l'unico **metodo astratto** definito nella **functional interface**.

```
Runnable r = () -> System.out.println("Running!");
```

```
r.run(); // Invoca il metodo `run` di Runnable
```

Internamente, le lambda sono trattate come **istanze di classi anonime** che implementano la **functional interface**. Tuttavia, anziché creare una nuova classe per ogni lambda, Java utilizza un approccio ottimizzato tramite il bytecode.

## Expanding a lambda



In questo esempio, la lambda `x -> System.out.println(x)` è un'implementazione del parametro ricevuto da `forEach`, ovvero la functional interface `Consumer<T>`.

La lambda viene tradotta nel metodo `accept`, che prende un elemento della lista e lo elabora.

### An example: From inner classes pre Java 8...

```
public class Calculator1 { // Pre Java 8
    interface IntegerMath { // (inner) functional interface
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    } // parameter type is functional interface

    // inner class implementing the interface
    static class IntMath$Add implements IntegerMath{
        public int operation(int a, int b){
            return a + b;
        }
    }
    public static void main(String... args) {
        Calculator1 myApp = new Calculator1();
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, new IntMath$Add()));
    } // anonymous inner class implementing the interface
    IntegerMath subtraction = new IntegerMath(){
        public int operation(int a, int b){
            return a - b;
        };
    };
    System.out.println("20 - 10 = " +
        myApp.operateBinary(20, 10, subtraction)); }}
```

## ... to lambda expressions

```
public class Calculator {  
  
    interface IntegerMath { // (inner) functional interface  
        int operation(int a, int b);  
    }  
  
    public int operateBinary(int a, int b, IntegerMath op) {  
        return op.operation(a, b);  
    } // parameter type is functional interface  
  
    public static void main(String... args) {  
        Calculator myApp = new Calculator();  
        // lambda assigned to functional interface variables  
        IntegerMath addition = (a, b) -> a + b;  
        System.out.println("40 + 2 = " +  
            myApp.operateBinary(40, 2, addition));  
        // lambda passed to functional interface formal parameter  
        System.out.println("20 - 10 = " +  
            myApp.operateBinary(20, 10, (a, b) -> a - b));  
    }  
}
```

Le espressioni lambda in Java possono essere, in linea di principio, **compilate** come **istanze di classi interne anonime**. Tuttavia, né la specifica del linguaggio Java SE 8 né la specifica della JVM SE 8 prescrivono una strategia di compilazione specifica per le lambda. La scelta della strategia di compilazione è lasciata al progettista del compilatore, che può sfruttare questa libertà per migliorare l'efficienza. I compilatori attuali utilizzano l'istruzione **invokedynamic** della JVM, che consente di posticipare al runtime la risoluzione di un punto di chiamata.

```
public class ThreadTest { // using functional interface Runnable  
    public static void main(String[] args) {  
        Runnable r1 = new Runnable() { // anonymous inner class  
            @Override  
            public void run() {  
                System.out.println("Old Java Way");  
            }  
        };  
  
        new Thread(r1).start();  
        // using lambda expression  
        Runnable r2 = () -> { System.out.println("New Java Way"); };  
  
        new Thread(r2).start();  
    }  
  
    // constructor of class Thread  
    public Thread(Runnable target)
```

Java 8 ha introdotto nuove interfacce funzionali nel pacchetto `java.util.function` per supportare l'uso delle espressioni lambda e la programmazione funzionale.

```
public interface Consumer<T> { //java.util.function  
    void accept(T t);  
}  
public interface Supplier<T> { //java.util.function  
    T get();  
}  
public interface Predicate<T> { //java.util.function  
    boolean test(T t);  
}  
public interface Function <T,R> { //java.util.function  
    R apply(T t);  
}
```

Alcune delle principali interfacce sono: **Consumer<T>**, che accetta un parametro di tipo T e restituisce void tramite il metodo `accept(T t)`; **Supplier<T>**, che non accetta argomenti e restituisce un valore di tipo T con il metodo `get()`; **Predicate<T>**, che valuta un parametro di tipo T e restituisce un valore booleano tramite il metodo `test(T t)`; e **Function<T, R>**, che accetta un parametro di tipo T e restituisce un valore di tipo R.

attraverso il metodo `apply(T t)`. Queste interfacce sono alla base della programmazione funzionale in Java.

```
List<Integer> intSeq = new ArrayList<>(Arrays.asList(1,2,3));  
  
// sort list in descending order using Comparator<Integer>  
intSeq.sort((x,z) -> z - x); // lambda with two arguments  
intSeq.forEach(System.out::println);  
  
// remove odd numbers using a Predicate<Integer>  
intSeq.removeIf(x -> x%2 == 1);  
intSeq.forEach(System.out::println); // prints only '2'
```

But, where do the `sort`, `removeIf` and `forEach` methods come from?

```
// default method of Interface List<E>  
default void sort(Comparator<? super E> c)  
// default method of Interface Collection<E>  
default boolean removeIf(Predicate<? super E> filter)  
// default method of Interface Iterable<T>  
default void forEach(Consumer<? super T> action)
```

## Default methods

I **default methods** sono stati introdotti in Java 8 per risolvere il problema della retrocompatibilità quando si aggiungono nuovi metodi alle interfacce. Prima di Java 8, aggiungere metodi astratti a un'interfaccia rompeva le implementazioni esistenti, poiché tutte le classi implementatrici dovevano fornire una definizione per i nuovi metodi.

Un **default method** è un metodo con un'**implementazione predefinita** all'interno di un'**interfaccia**. Utilizza la parola chiave **default**:

```
public interface MyInterface {  
    default void myDefaultMethod(){  
        System.out.println("Default implementation");  
    }  
}
```

In Java 8, come in precedenza un'interfaccia può contenere **metodi astratti** che **devono** essere implementati dalle classi. Possono essere definiti metodi statici nelle interfacce, accessibili direttamente tramite il nome dell'interfaccia. Possono essere **definiti Default methods con implementazione predefinita**, che possono essere sovrascritti nelle classi implementatrici.

I default methods permettono quindi di **aggiungere nuovi metodi alle interfacce** senza obbligare le classi esistenti a modificarne le implementazioni.

## Method references

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

I **riferimenti ai metodi** in Java possono essere utilizzati per **passare una funzione esistente** nei contesti in cui ci si aspetta un'espressione lambda. La firma del metodo referenziato deve corrispondere a quella del metodo dell'interfaccia funzionale. Esistono diversi tipi di riferimenti ai metodi: riferimenti a metodi statici con la sintassi

ClassName::StaticMethodName, come String::valueOf; riferimenti ai costruttori con ClassName::new, come ArrayList::new; riferimenti a metodi di un'istanza specifica con objectReference::MethodName, come x::toString; e riferimenti a metodi di un oggetto arbitrario di un dato tipo con ClassName::InstanceMethodName, come Object::toString.

## AP-19: Streams in Java 8

In Java 8, il pacchetto **java.util.stream** fornisce utilità per supportare operazioni in stile funzionale su stream di valori. Uno stream non è una struttura dati che memorizza elementi.

Uno **stream** è un **flusso di dati** che trasporta elementi da **una sorgente** (come una struttura dati, un array, una funzione generatrice, un canale I/O, ecc.) attraverso una **pipeline di operazioni** computazionali.

Gli stream sono di **natura funzionale**, il che significa che ogni operazione su uno stream produce un risultato **senza modificare la sorgente** da cui proviene il flusso stesso.

In Java 8, i flussi (streams) sono progettati per la **laziness**, il che significa le operazioni sugli stream possono essere implementate in modo tale da **ritardare l'esecuzione** fino a quando non è effettivamente **necessaria**.

Le operazioni sugli stream si dividono in **operazioni intermedie**, che producono nuovi flussi e **operazioni terminali**, che producono **valori o effetti collaterali**. Le operazioni intermedie sono sempre "lazy", quindi **non eseguono** l'elaborazione finché non viene invocata un'operazione **terminale**.

Gli stream possono essere potenzialmente illimitati, a differenza delle collezioni che hanno una dimensione finita.

Alcuni metodi della pipeline, detti "a corto circuito" (come limit o findFirst), possono interrompere l'elaborazione dei metodi intermedi precedenti non appena sono in grado di valutare la loro condizione, ottimizzando così le prestazioni del pipeline, e consentendo di completare calcoli su flussi infiniti in un tempo finito senza dover aspettare.

Inoltre, gli stream sono "**consumabili**", cioè gli elementi di uno stream vengono visitati **una sola volta** durante la vita del flusso stesso e non possono essere **più rivisitati** dopo l'esecuzione di un'operazione **terminale**.

### Pipeline

Una **pipeline** in Java è una **sequenza di operazioni** che viene applicata su un flusso di dati (stream). Una pipeline include **una sorgente** che produce gli elementi del flusso, zero o più **operazioni intermedie che trasformano** il flusso (come filter, map), e **un'operazione terminale** che produce un **risultato** o ha un effetto collaterale (come forEach, collect, reduce).

```
double average = listing // collection of Person
    .stream()           // stream wrapper over a collection
    .filter(p -> p.getGender() == Person.Sex.MALE) // filter
    .mapToInt(Person::getAge)           // extracts stream of ages
    .average()             // computes average (reduce/fold)
    .getAsDouble();         // extracts result from OptionalDouble
```

Un esempio tipico di pipeline è filter/map/reduce. Una collezione di oggetti Person viene convertita in un flusso, filtrata per includere solo elementi di sesso maschile, mappata per estrarre le età, e infine viene calcolata la media delle età con l'operazione terminale average(), il cui risultato viene convertito in un double con getAsDouble().

## Creare uno stream

Gli stream in Java possono essere creati da diverse fonti. È possibile ottenerli da una **Collection** usando i metodi **stream()** e **parallelStream()**, oppure da un **array** tramite **Arrays.stream(Object[])**. Esistono metodi di fabbrica statici come **Stream.of(Object[])**, **IntStream.range(int, int)** o **Stream.iterate(Object, UnaryOperator)** per generare stream. Le righe di un file possono essere trasformate in uno stream utilizzando **BufferedReader.lines()**. I metodi della classe **Files** permettono di creare stream di percorsi di file. Anche le sequenze di numeri casuali possono essere ottenute tramite **Random.ints()**. Altre fonti includono generatori come **Stream.generate()** e **Stream.iterate()**, e molti altri metodi presenti nella JDK offrono la possibilità di creare stream per vari scopi.

## Operazioni intermedie

Le **operazioni intermedie** in Java 8 mantengono uno stream aperto per ulteriori elaborazioni e sono eseguite in modo "lazy", cioè vengono valutate solo quando viene invocata un'operazione terminale. Alcuni esempi di operazioni intermedie includono **filter()**, che filtra elementi in base a un predicato, **mapToInt()** e **map()**, che trasformano gli elementi da un tipo a un altro, e **peek()**, che permette di eseguire azioni su elementi senza modificarli. Operazioni come **distinct()** rimuovono i duplicati, mentre **sorted()** ordina gli elementi dello stream. **limit()** tronca lo stream fino a un massimo numero di elementi, e **skip()** salta i primi n elementi. Queste operazioni accettano argomenti di interfacce funzionali, consentendo l'uso di espressioni lambda.

```
interface Stream<T>{...
Stream<T> filter(Predicate<? super T> predicate) // filter
IntStream mapToInt(ToIntFunction<? super T> mapper) // map f:T -> int
<R> Stream<R> map(Function<? super T,>? extends R> mapper) // map f:T->R
Stream<T> peek(Consumer<? super T> action) //performs action on elements
Stream<T> distinct() // remove duplicates - stateful
Stream<T> sorted() // sort elements of the stream - stateful
Stream<T> limit(long maxSize) // truncate
Stream<T> skip(long n) // skips first n elements
}
```

## peek()

Il metodo **peek()** è un'operazione intermedia utilizzata per eseguire azioni su ciascun elemento di uno stream senza modificarlo. Viene spesso impiegato per il debug, poiché consente di visualizzare i dati durante il passaggio attraverso lo stream.

```
IntStream.of(1, 2, 3, 4)
.filter(e -> e > 2)
.peek(e -> System.out.println("Filtered value: " + e))
.map(e -> e * e)
.peek(e -> System.out.println("Mapped value: " + e))
.sum();
```

## Operazioni terminali

Le **operazioni terminali** rappresentano l'ultimo passaggio in una pipeline di stream, dopo il quale lo stream viene consumato e non può più essere riutilizzato. Servono a raccogliere i valori in una struttura dati, ridurre gli elementi a un singolo valore, stampare o produrre effetti collaterali.

```
void forEach(Consumer<? super T> action)

Object[] toArray()

T reduce(T identity, BinaryOperator<T> accumulator) // fold

Optional<T> reduce(BinaryOperator<T> accumulator) // fold

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

boolean allMatch(Predicate<? super T> predicate) // short-circuiting

boolean anyMatch(Predicate<? super T> predicate) // short-circuiting

Optional<T> findAny() // short-circuiting
```

Alcuni esempi di operazioni terminali includono: **forEach()** per eseguire un'azione su ogni elemento; **toArray()** per trasformare lo stream in un array; **reduce()** per combinare elementi in un singolo risultato (come una somma); **min()** per ottenere il valore minimo; **allMatch()** e **anyMatch()** per verificare condizioni specifiche sugli elementi.

## Tipi di stream

Gli stream supportano solo tipi di riferimento e i tipi primitivi principali come int, long e double, mentre i tipi primitivi minori non sono direttamente supportati.

```
"Hello world!".chars()
    .forEach(System.out::print);

// prints
721011081081113211911111410810033

// fixing it:
"Hello world!".chars()
    .forEach(x -> System.out.print((char) x));
```

Quando si utilizza il metodo chars() su una stringa, come "Hello world!".chars().forEach(System.out::print);, l'output è una serie di numeri che rappresentano i codici Unicode dei caratteri. Questo accade perché chars() restituisce uno IntStream di codici interi. Per stampare correttamente i caratteri, è necessario eseguire il cast a char, ad esempio: "Hello world!".chars().forEach(x -> System.out.print((char) x));.

```
String concatenated = listOfStrings
    .stream()
    .reduce("", String::concat)
```

Usare .reduce("", String::concat) per concatenare stringhe è inefficiente, poiché per ogni elemento dello stream viene creata una nuova stringa, causando un elevato overhead. È più efficiente accumulare gli elementi in un oggetto mutabile (come un StringBuilder o una collezione). Questo approccio sfrutta l'operazione collect().

```
<R> R collect( Supplier<R> supplier,
                 BiConsumer<R, ? super T> accumulator,
                 BiConsumer<R, R> combiner);
```

Dove: R è il tipo del contenitore di risultato. T è il tipo degli elementi dello stream. collect() richiede tre funzioni:

**Supplier:** crea un contenitore mutabile (ad esempio, un nuovo StringBuilder).

**Accumulator:** aggiunge ogni elemento dello stream al contenitore.

**Combiner:** unisce due contenitori (utile per stream paralleli).

```
// no streams
ArrayList<String> strings = new ArrayList<>();
for (T element : stream) {
    strings.add(element.toString());
}

// with streams and lambdas
ArrayList<String> strings =
    stream.collect(() -> new ArrayList<>(), //Supplier
    (c, e) -> c.add(e.toString()), // Accumulator
    (c1, c2) -> c1.addAll(c2)); //Combining

// with streams and method references
ArrayList<String> strings = stream.map(Object::toString)
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

Il metodo collect() in Java può essere invocato con un argomento di tipo Collector, che incapsula le funzioni necessarie per raccogliere elementi di un flusso, come un Supplier, BiConsumer e Combiner, permettendo la riutilizzabilità delle strategie di raccolta e la composizione di operazioni collect().

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

// The following will accumulate strings into an ArrayList:
List<String> asList = stringStream.collect(Collectors.toList());

// The following will classify Person objects by city:
Map<String, List<Person>> peopleByCity =
    personStream.collect(Collectors.groupingBy(Person::getCity));
```

Ad esempio, si può accumulare una sequenza di stringhe in una ArrayList con stringStream.collect(Collectors.toList());. I Collector predefiniti in Collectors consentono di semplificare la raccolta dei dati e creare aggregazioni complesse con facilità.

## Stream infiniti

Gli stream in Java possono essere finiti o infiniti. I flussi che avvolgono collezioni sono finiti, mentre i flussi infiniti possono essere generati usando i metodi iterate() e generate().

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)

// Example: summing first 10 elements of an infinite stream
int sum = Stream.iterate(0,x -> x+1).limit(10).reduce(0,(x,s) -> x+s);

static <T> Stream<T> generate(Supplier<T> s)

// Example: printing 10 random numbers
Stream.generate(Math::random).limit(10).forEach(System.out::println);
```

Il metodo iterate consente di creare uno stream infinito partendo da un valore iniziale (seed) e applicando iterativamente una funzione unaria.

## Parallelism

Le operazioni sui flussi possono essere eseguite sia in modo seriale (impostazione predefinita) che in parallelo. Utilizzando il metodo parallelStream(), è possibile eseguire le operazioni su più thread in modo trasparente e gestito automaticamente dal

supporto runtime.

```
double average = persons //average age of all male
    .parallelStream()      // members in parallel
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Se le operazioni non hanno effetti collaterali, la sicurezza dei thread è garantita anche quando si utilizzano collezioni non thread-safe, come ArrayList. L'ordine di elaborazione degli elementi di un flusso può variare a seconda che l'esecuzione sia seriale o parallela, così come dalle operazioni intermedie utilizzate.

```
Integer[] intArray = {1, 2, 3, 4, 5, 6, 7, 8};
List<Integer> listOfIntegers = new ArrayList<>(Arrays.asList(intArray));
listOfIntegers .stream()
    .forEach(e -> System.out.print(e + " "));
// prints: 1 2 3 4 5 6 7 8
listOfIntegers .parallelStream()
    .forEach(e -> System.out.print(e + " "));
// may print: 3 4 1 6 2 5 7 8
```

- Consider this for-loop (.96 s runtime; dual-core laptop)

```
long sum = 0;
for (long j = 0; j < Integer.MAX_VALUE; j++) sum += j;
```

- Equivalent stream computation (1.5 s)

```
long sum = LongStream.range(0, Integer.MAX_VALUE).sum();
```

- Equivalent parallel computation (.77 s)

```
long sum = LongStream.range(0, Integer.MAX_VALUE)
    .parallel().sum();
```

Si usano stream paralleli quando le operazioni sugli elementi sono **indipendenti** (non influenzano o dipendono l'una dall'altra), Le operazioni sono **computazionalmente costose** OR Sono applicate a un numero elevato di elementi provenienti da una struttura dati facilmente **divisibile** (es. array, liste).

**Regola empirica per valutare se parallelizzare:** calcolare  $N * Q$  (dove N è il numero di elementi e Q il costo per elemento) e assicurarsi che sia almeno 10.000.

### Criticità degli stream

**Non-interference:** I parametri comportamentali lambda non devono modificare la sorgente del flusso, garantendo che le operazioni rimangano prevedibili.

**Stateless behaviours:** è consigliato che le operazioni intermedie siano senza stato per facilitare la parallelizzazione e uno stile di programmazione funzionale.

**Parallelism and Thread Safety :** Il parallelismo richiede che il programmatore garantisca la sicurezza dei thread. In caso di effetti collaterali è necessario adottare tecniche di sincronizzazione o l'uso di strutture dati concorrenti per evitare race conditions.

```
try {
    List<String> listOfStrings =
        new ArrayList<>(Arrays.asList("one", "two"));

    String concatenatedString = listOfStrings
        .stream()
    // Don't do this! Interference occurs here.
        .peek(s -> listOfStrings.add("three"))
        .reduce((a, b) -> a + " " + b)
        .get();
    System.out.println("Concatenated string: " + concatenatedString);
} catch (Exception e) {
    System.out.println("Exception caught: " + e.toString());
}
```

# Monads in Java: Optional and Stream

```
public static <T> Optional<T> of(T value)
// Returns an Optional with the specified present non-null value.

<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
/* If a value is present, apply the provided Optional-bearing mapping
function to it, return that result, otherwise return an empty
Optional. */
```

```
static <T> Stream<T> of(T t)
// Returns a sequential Stream containing a single element.

<R> Stream<R> flatMap(
    Function<? super T,? extends Stream<? extends R>> mapper)
/* Returns a stream consisting of the results of replacing each element
of this stream with the contents of a mapped stream produced by applying
the provided mapping function to each element. */
```

**Optional.of(T value)**: Questa funzione statica restituisce un Optional contenente il valore specificato, che non deve essere null.

**<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)**: Se un valore è presente, applica la funzione mapper (che restituisce un altro Optional) al valore.

Restituisce il risultato della funzione oppure un Optional vuoto se non c'è valore presente.

## Stream

**Stream.of(T t)**: Crea uno stream sequenziale contenente un singolo elemento.

**flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**: Applica la funzione mapper a ogni elemento dello stream corrente. La funzione restituisce un altro stream; flatMap appiattisce questi stream in uno unico concatenando i risultati. Questo consente di trasformare uno stream di collezioni in uno stream unico dei loro elementi (es. da Stream<List<T>> a Stream<T>).

## AP-22: The Java Memory Model

Un **memory model** è una **specifica** che definisce **come le operazioni di memoria**, quali letture e scritture, **vengono eseguite** e percepite dai programmatore e dal sistema. Esso **stabilisce il comportamento previsto** per l'accesso multithread alla **memoria condivisa** e determina quale valore una lettura da una locazione di memoria può restituire.

Ogni interfaccia hw e sw che consente l'accesso multithread alla memoria condivisa necessita di un memory model per garantire un comportamento prevedibile.

Il memory model influisce sulle **trasformazioni** che il compilatore può applicare al programma, permettendo ottimizzazioni che rispettano le regole definite. Queste ottimizzazioni includono il riordinamento delle istruzioni o l'accesso parallelo, purché rispettino le regole definite dal modello, senza compromettere la correttezza del programma.

Il modello influisce anche sui programmatore, poiché le trasformazioni consentite o vietate determinano i possibili risultati di un programma e, di conseguenza, quali pattern di progettazione per la comunicazione tra thread sono validi.

In Java il **memory model** definisce le **trasformazioni** che il compilatore può applicare al programma durante la **generazione** del bytecode, le **trasformazioni** che una **VM** può applicare al bytecode durante la conversione in codice nativo e le **ottimizzazioni** che **l'hardware** può eseguire sul codice nativo.

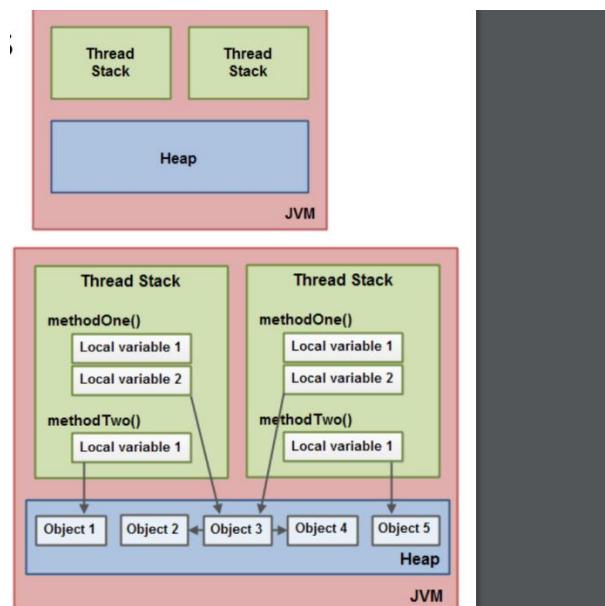
Il **Java Memory Model (JMM)** è progettato per specificare il **comportamento legale** dei programmi **multithread**. Fornisce garanzie standard per i programmi correttamente sincronizzati, assicurando una **consistenza sequenziale** per i programmi privi di data race.

Se un programma non è correttamente sincronizzato, il suo comportamento è limitato da una nozione ben definita di **causalità**. Le restrizioni di **causalità** nel JMM sono sufficientemente forti per rispettare le proprietà di sicurezza e affidabilità di Java, ma allo stesso tempo abbastanza flessibili da permettere le ottimizzazioni standard dei compilatori e dell'hardware.

## Aree di memoria runtime della JVM

**Thread Stack:** Ogni thread ha il **proprio** thread stack, utilizzato per allocare le variabili locali dei metodi. Le variabili locali allocate nel thread stack **non possono essere** accedute da altri thread.

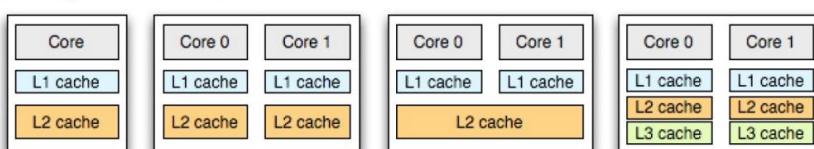
**Heap:** Tutti gli oggetti e le loro istanze sono allocati sull'heap, che è un'area di memoria **condivisa** tra tutti i thread. Solo gli oggetti presenti nell'heap **possono essere condivisi** tra i thread.



La memoria è divisa in più livelli per ottimizzare le prestazioni e l'efficienza. I livelli più vicini alla CPU, come i registri e le cache di primo livello (L1), sono i più veloci ma hanno una capacità limitata. I livelli più lontani, come la RAM e la memoria di massa, hanno una maggiore capacità ma tempi di accesso più lenti.

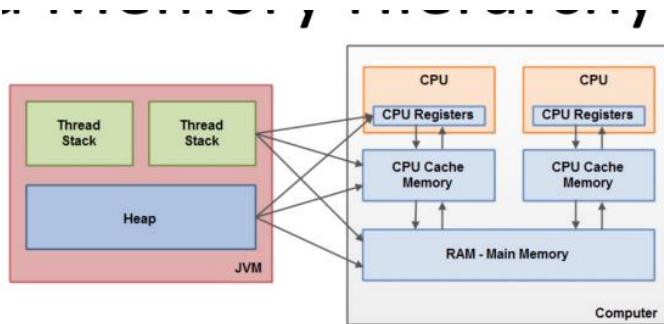
Alcuni livelli della gerarchia, come i **registri** e le cache di primo livello (L1), sono specifici per ogni core, il che significa che ogni core ha le proprie copie isolate. Altri livelli, come la **RAM**, sono condivisi tra tutti i core, permettendo la comunicazione e la condivisione dei dati.

For example, this is a part of the memory hierarchy in some processors:



## JVM data areas and Memory Hierarchy

L'Area di memoria della JVM e la gerarchia della memoria di un computer sono ortogonali, nel senso che si intersecano ma mantengono una loro indipendenza concettuale.



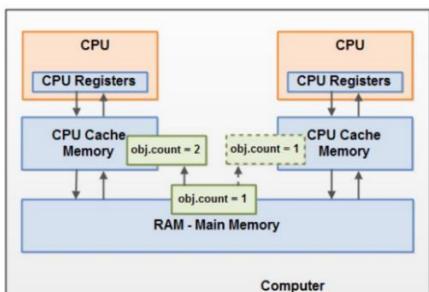
Quando un thread aggiorna una variabile, gli altri thread potrebbero non vedere immediatamente il cambiamento a causa della cache locale dei CPU o della latenza nella RAM.

I **race condition** si verificano quando più thread **accedono contemporaneamente** a una variabile condivisa senza una corretta **sincronizzazione**, causando risultati **imprevedibili**.

I meccanismi forniti da Java sono l'operatore **volatile**, che garantisce che le modifiche a una variabile siano **immediatamente visibili** agli altri thread; e **synchronized** (**metodi/blocchi**), che garantisce l'accesso esclusivo alla memoria condivisa da parte di un singolo thread alla volta, in **mutua esclusione (lock)** prevenendo le race condition.

Il **Java Memory Model (JMM)** tiene conto esplicitamente di questi meccanismi per assicurare un comportamento prevedibile e corretto nell'accesso alla memoria condivisa tra thread.

## Volatile modifier



In un sistema multicore, gli aggiornamenti a una variabile condivisa potrebbero essere **visibili** solo nella **cache locale** del core che li ha effettuati. Le cache sincronizzano i dati aggiornati nella RAM seguendo politiche di "flushing", il che può causare ritardi e problemi di visibilità tra i thread.

In Java, dichiarare una variabile come **volatile** garantisce che ogni lettura e scrittura della variabile **acceda direttamente** alla **RAM**, evitando l'uso della cache locale e assicurando che le modifiche siano visibili a tutti i thread.

```
public class Loop {
    static boolean done;
    static int n;
    public static void main(String args[]) {
        Thread t = new Thread() {
            public void run() {
                n = 42;
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    return;
                }
                done = true;
                System.out.println("Fatto");
            }
        };
        t.start();
        while (!done) {
        }
        System.out.println(n);
    }
}
done = false
n = 0
Thread 1 (main):          Thread 2:
while (!done) { };       n = 42;
System.out.println(n);   sleep(1000);
done = true;
System.out.println("Fatto");
```

In un programma multithread, se due thread condividono le variabili `n` e `done` ma non usano alcun meccanismo di sincronizzazione, possono verificarsi comportamenti inattesi. Ad esempio, un ciclo while nel **thread 1** che controlla la variabile `done` può comportarsi come un ciclo infinito, anche se **thread 2** modifica la variabile `done` impostandola a true dopo un secondo.

```
public class Loop {
    static volatile boolean done;
    static int n;

    public static void main(String args[]) {
        Thread t = new Thread() {
            public void run() {
                n = 42;
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    return;
                }
                done = true;
                System.out.println("Fatto");
            }
        };
        t.start();
        while (!done) {
        }
        System.out.println(n);
    }
}
```

Each reading of the `done` variable made by the main thread makes visible the changes to `done` made by the other thread

Therefore, the main thread always terminates.

La keyword **volatile** può essere applicato solo ai **campi di una classe**, non ai metodi o alle variabili locali. Non può essere applicato a campi final poiché questi non cambiano valore dopo l'inizializzazione e sarebbe contraddittorio dichiararli volatile.

Il **Java Memory Model (JMM)** garantisce che una scrittura su una variabile volatile sarà visibile agli altri thread quando essa viene letta da essi.

L'implementazione deve garantire che il nuovo valore di una variabile volatile sia immediatamente "flushed" (ovvero sincronizzato) dalla cache alla RAM principale.

## Data race

Una **data race** si verifica quando due thread possono eseguire **azioni in conflitto** su una variabile **condivisa** in modo **non deterministico**.

Due azioni su una variabile condivisa sono considerate in **conflitto** se almeno una di esse è una **scrittura**. Ad esempio, se un thread legge una variabile mentre un altro thread la modifica, o se entrambi i thread la modificano contemporaneamente, si ha un conflitto.

Poiché **l'ordine di esecuzione** tra i thread non è garantito, l'accesso simultaneo non sincronizzato a una variabile condivisa può produrre risultati diversi e imprevedibili. Il modificatore volatile garantisce solo la visibilità degli aggiornamenti tra i thread, ma **non** previene le data race.

Per evitare i data race, è necessario usare la keyword **synchronized**, che garantisce la **mutua esclusione**. Ciò significa che solo un thread alla volta può accedere alla sezione di codice che modifica o legge la variabile.

```
static Counter sharedCounter = new Counter();
public static void main(String... args)
    throws InterruptedException{
Runnable r = () -> {
    for (int i = 1; i <= 10000; i++) {
        sharedCounter.incr();
    }
};
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(sharedCounter.getCount());
} // prints values smaller than 20000
```

```
class Counter {
    int count;
    public void incr() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

```
class Counter {
    int count;
    public synchronized void incr() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

13

14

Ogni oggetto in Java possiede un **monitor** associato, che permette ai thread di **bloccare** e **sbloccare** l'accesso a sezioni critiche.

Quando un thread ottiene il lock su un oggetto, nessun altro thread può accedere alle sezioni sincronizzate di quell'oggetto finché il lock non viene rilasciato.

I metodi o blocchi dichiarati con **synchronized** acquisiscono **automaticamente** il lock all'inizio dell'esecuzione e lo rilasciano alla fine.

Questo assicura che solo un thread alla volta possa eseguire il metodo o il blocco sincronizzato, evitando così accessi simultanei non controllati e garantendo la **mutua esclusione**.

Il **Java Memory Model (JMM)** non impone un ordine globale esplicito per tutte le azioni eseguite dai thread, ma mantiene un ordine coerente con la percezione locale di ciascun thread del tempo. Inoltre, il JMM non utilizza una memoria centrale globale come riferimento unico per tutte le variabili; queste possono risiedere nelle cache locali dei thread e diventare visibili agli altri solo attraverso specifiche regole di sincronizzazione. Le esecuzioni sono descritte attraverso **azioni** relative alla memoria, come letture, scritture e sincronizzazioni, che sono regolate da **relazioni di ordine parziale** per garantire flessibilità nelle ottimizzazioni. Infine, una **funzione di visibilità associa** ogni operazione di **lettura** a una specifica operazione di **scrittura**, determinando quale valore viene letto in ogni momento.

## Actions of the JMM

Ogni azione nella JMM è associata a un **thread  $T(a)$**  e ha un **tipo specifico**. I tipi di azione sono i seguenti.

**Lettura volatile di una variabile v:** Un thread legge il valore di una variabile dichiarata volatile.

**Scrittura volatile di una variabile v:** Un thread scrive un valore in una variabile volatile.

**Lock su un oggetto monitor m:** Un thread acquisisce un lock su un monitor.

**Unlock su un oggetto monitor m:** Un thread rilascia il lock su un monitor.

**Lettura normale di una variabile v:** Un thread legge il valore di una variabile.

**Scrittura normale di una variabile v:** Un thread scrive un valore in una variabile.

**Azione esterna:** Operazioni che hanno effetti al di fuori della JVM, come input/output.

Le azioni indicate in **rosso** (letture/scritture volatile, lock/unlock di monitor) sono considerate **azioni di sincronizzazione**.

Il **Java Memory Model** impone regole che stabiliscono se un insieme di azioni costituisce un'esecuzione valida di un programma.

## Program order & sequential consistency

### Ordine del programma ( $\leq_{\{po\}}$ )

In un programma single-thread, l'esecuzione fissa un **ordine totale** delle azioni che rappresenta la **sequenza con cui vengono eseguite**, chiamata **ordine del programma**. Nei programmi multithread, l'ordine del programma è **l'unione degli ordini dei singoli thread** e non definisce una relazione tra le azioni eseguite da thread diversi.

### Consistenza sequenziale

Un'esecuzione di un programma multithread è **sequenzialmente consistente** se il comportamento complessivo può essere spiegato come se esistesse un **ordine sequenziale globale** che rispetta:

1. **L'ordine interno di ciascun thread:** le azioni di ogni thread devono avvenire nell'ordine specificato da ciascuno.
2. **Ogni lettura restituisce il valore dell'ultima scrittura precedente a essa** nell'ordine globale.

Thread T1:

```
java  
  
x = 1; // Azione A  
print(x); // Azione B
```

Thread T2:

```
java  
  
x = 2; // Azione C  
print(x); // Azione D
```

#### Possibile esecuzione sequenzialmente consistente

Un ordine globale sequenzialmente consistente potrebbe essere:

1. T1 esegue `x = 1` (A).
2. T2 esegue `x = 2` (C).
3. T1 esegue `print(x)` (B), che stampa `2` (l'ultima scrittura è C).  
↓
4. T2 esegue `print(x)` (D), che stampa `2`.

In questa esecuzione:

- L'ordine interno di T1 (A prima di B) è rispettato.
- L'ordine interno di T2 (C prima di D) è rispettato.
- Ogni lettura di `x` restituisce il valore dell'ultima scrittura nell'ordine globale (B legge `2`, che è stato scritto da C; D legge `2`, scritto anch'esso da C).

## Guarantees of the JMM

Il **Java Memory Model (JMM)** è stato progettato per garantire tre promesse fondamentali:

**Garanzia Data Race Free:** Garantisce che i programmi privi di **data race**, correttamente sincronizzati, mantengano la **consistenza sequenziale**, nonostante le ottimizzazioni.

**Garanzia per la sicurezza:** Garantisce che nei programmi con **data race** i valori non appaiano "dal nulla".

**Garanzia per i compilatori:** Permette alle ottimizzazioni comuni di essere eseguite nella misura massima possibile, purché non violino le prime due promesse.

La complessità del JMM è giustificata dal desiderio di garantire la **promessa di sicurezza** anche per i programmi non privi di data race. La versione precedente del modello considerava questi programmi come errati e il loro comportamento non era specificato, portando a situazioni imprevedibili.

## Why is security rule too strong?

Consideriamo due thread con le seguenti operazioni (inizialmente A=0 e B=0):

```
// Thread1  
int r1;  
r1 = B;  
A = 2;
```

```
// Thread2  
int r2;  
r2 = A;  
B = 1;
```

**r1 = 0, r2 = 2:** Se **Thread 1** esegue prima di **Thread 2**.

**r1 = 1, r2 = 0:** Se **Thread 2** esegue prima di **Thread 1**.

**r1 = 0, r2 = 0:** Se entrambi i thread leggono i valori iniziali prima di qualsiasi scrittura.

**r1 = 1, r2 = 2:** Può succedere in alcuni scenari se l'ordine delle istruzioni viene modificato.

In assenza di sincronizzazione, il compilatore, la JVM o la CPU possono **riordinare le istruzioni** per ottimizzare le prestazioni, purché il riordinamento non modifichi il comportamento visibile di un singolo thread. Questo riordinamento può portare a risultati come `r1 = 1, r2 = 2`, che non è sequenzialmente consistente.

La **consistenza sequenziale** garantisce che le operazioni avvengano in un ordine che rispetta l'ordine logico interno dei thread. Tuttavia, per permettere ottimizzazioni, il **JMM** consente esecuzioni non sequenzialmente consistenti nei programmi con **data race (regola 2)**.

## “Out-of-thin-air” behaviours

L'immagine descrive il problema dei comportamenti **"out-of-thin-air"** (letteralmente "dal nulla") nel contesto del **Java Memory Model (JMM)**. Ecco una spiegazione chiara:

```
// Thread1           // Thread2
r1 = x;             r2 = y;
y = r1;             x = r2;
```

Ci si chiede se sia possibile ottenere alla fine l'output  $r1 == r2 == 42$ . La risposta è **no** in condizioni normali, ma un'ipotetica valutazione speculativa aggressiva potrebbe portare a questo comportamento.

Dire che **un valore viene "dal nulla"** significa che non c'è **un'assegnazione logica** che giustifichi l'apparizione di quel valore.

Il **JMM** vieta esplicitamente questo tipo di esecuzioni. Anche in presenza di data race, non è consentito che valori appaiano "dal nulla".

### Ordine di sincronizzazione ( $\leq_{so}$ )

Ogni esecuzione di un programma ha un **“ordine di sincronizzazione”**, che è un **ordine totale** su tutte le **azioni di sincronizzazione**

Questo ordine deve rispettare: **La consistenza con l'ordine del programma && Le letture di variabili volatile devono restituire il valore dell'ultima scrittura ordinata prima della lettura secondo l'ordine di sincronizzazione.**

```
lock(M);      // Azione A
x = 1;         // Azione B
unlock(M);    // Azione C
```

```
Thread T2:
java
lock(M);      // Azione D
print(x);     // Azione E
unlock(M);    // Azione F
```

Ordine di sincronizzazione  
L'ordine di sincronizzazione è un ordine totale che include tutte le operazioni di sincronizzazione (lock e unlock). Un possibile ordine è:  
1. T1 esegue lock(M) (A).  
2. T1 esegue unlock(M) (C).  
3. T2 esegue lock(M) (D).  
4. T2 esegue unlock(M) (F).

Si dice che un'azione a **“synchronizes-with”** un'azione b se: a è un'operazione di **unlock** di un monitor e b è un'operazione di **lock** sullo stesso monitor; a è una scrittura su una variabile volatile e b è una lettura della stessa variabile volatile.

Questo implica che se un'azione a avviene prima di b secondo l'ordine di sincronizzazione, i **cambiamenti effettuati** da a sono **visibili** a b.

```
java
lock(M);      // Azione A
x = 1;         // Azione B
unlock(M);    // Azione C
```

```
Thread T2:
java
lock(M);      // Azione D
print(x);     // Azione E
unlock(M);    // Azione F
```

Qui, l'operazione di `unlock(M)` in T1 (C) **synchronizes-with** l'operazione di `lock(M)` in T2 (D). Questo implica che il valore di `x` scritto in T1 (B) è visibile in T2 (E).

La relazione **"happens-before"** è la chiusura transitiva dell'**ordine del programma** ( $\leq_{po}$ ) e della relazione **"synchronizes-with"** ( $\leq_{sw}$ ).

Se a **"happens-before"** b, significa che l'azione a **avviene logicamente prima** di b, garantendo che i cambiamenti effettuati da a siano **visibili** a b.

```

lock(M);      // Azione A
x = 1;        // Azione B
unlock(M);    // Azione C

```

Thread T2:

```

java                               ⌂ Copia ⌂ Modifica

lock(M);      // Azione D
print(x);    // Azione E
unlock(M);    // Azione F

```

- L'azione C (`unlock(M)`) **synchronizes-with** l'azione D (`lock(M)`).
- Poiché B (`x = 1`) avviene prima di C nell'ordine del programma, e C **synchronizes-with** D, allora B **happens-before** E (`print(x)`).
- Questo garantisce che la modifica di `x` effettuata in B sia visibile in E.

## Data Race

Una **data race** si verifica quando due accessi A e B a una variabile condivisa soddisfano le seguenti condizioni: **Appartengono a thread diversi && Sono in conflitto**: almeno uno dei due accessi è una scrittura && **Non sono ordinati dalla relazione "happens-before"** in un'esecuzione sequenzialmente consistente.

Un programma è considerato **correttamente sincronizzato** o **privo di data race** se e solo se **tutte le sue esecuzioni, che seguono la consistenza sequenziale, sono libere da data race**. In altre parole, le azioni in conflitto sono sempre ordinate da una relazione "happens-before".

Il primo requisito del **Java Memory Model (JMM)** è assicurare la **consistenza sequenziale** per i programmi correttamente sincronizzati o privi di data race. I programmati devono preoccuparsi delle trasformazioni del codice (come le ottimizzazioni del compilatore o il riordinamento delle istruzioni) solo se i loro programmi contengono **data race**.

## Definizione di esecuzione ben formata

Un'esecuzione formalmente viene definita nella JMM come:

$E = (P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb})$ , dove:

- **P**: è un **programma**.
- **A**: è un **insieme di azioni**.
- **$\leq_{po}$** : è l'**ordine del programma**, un ordine totale sulle azioni di ogni thread.
- **$\leq_{so}$** : è l'**ordine di sincronizzazione**, un ordine totale sulle azioni di sincronizzazione in A.
- **W**: è una **funzione di scrittura vista**, che per ogni lettura r in A, restituisce  $W(r)$ , l'azione di scrittura relativa vista da r in E.
- **V**: è una **funzione di valore scritto**, che per ogni scrittura w in A, restituisce  $V(w)$ , il valore scritto da w in E.
- **$\leq_{sw}$** : è un **ordine parziale di sincronizzazione** ("synchronizes-with").
- **$\leq_{hb}$** : è un **ordine parziale di happen-before**.

## Esecuzione ben formata

Un'esecuzione  $E=(P,A,\leq_{po},\leq_{so},V,W,\leq_{hb},\leq_{sw})$  è considerata **ben formata** se soddisfa le seguenti condizioni:

1. **Visibilità delle scritture**: Ogni operazione di **lettura** di una variabile v ha associata e **deve “vedere”** un'operazione di **scrittura precedente** su v.

• Thread T1:

```

java                               ⌂ Copia ⌂ Modifica

x = 1; // Scrittura

```

• Thread T2:

```

java                               ⌂ Copia ⌂ Modifica

int r = x; // Lettura

```

**2. Consistenza dell'ordine di sincronizzazione:** L'ordine di sincronizzazione ( $\leq_{so}$ ) deve seguire l'ordine in cui le istruzioni appaiono nei singoli thread. Ad esempio, se un thread esegue una lock su un monitor, poi scrive una variabile, e successivamente fa una unlock, queste operazioni devono avvenire nell'ordine specificato dal programma.

Le operazioni di sincronizzazione devono garantire che i thread non violino la mutua esclusione definita dalle lock. Se un thread ha acquisito un lock su un monitor, nessun altro thread può accedere a quel monitor fino a quando il primo thread non lo rilascia.

- Thread T1:

```
java
synchronized(lock) {
    x = 1; // Scrittura
}
```

Copia Modifica

- Thread T2:

```
java
synchronized(lock) {
    int r = x; // Lettura
}
```

Copia Modifica

Grazie alla sincronizzazione tramite `lock`, l'azione di scrittura di  $T1$  e la lettura di  $T2$  non possono avvenire contemporaneamente, garantendo la mutua esclusione e la visibilità della scrittura.

**3. Consistenza inter-thread:** Le azioni tra thread diversi devono essere coordinate per garantire che i cambiamenti di memoria fatti da un thread siano visibili agli altri thread in modo coerente.

- Thread T1:

```
java
x = 1; // Scrittura
```

Copia Modifica

- Thread T2:

```
java
int r = x; // Lettura
```

Copia Modifica

Se non c'è sincronizzazione, potrebbe verificarsi un **data race**: T2 potrebbe leggere il valore di  $x$  prima che T1 abbia completato la scrittura, causando inconsistenza. Con sincronizzazione,  $x = 1$  sarà visibile a T2 al momento della lettura.

**4. Consistenza intra-thread e "happens-before":** L'esecuzione deve rispettare la consistenza intra-thread (cioè, ogni thread esegue le sue operazioni nell'ordine specificato dal programma) e la relazione "happens-before  $\leq_{hb}$ ". Una lettura di una variabile  $v$  deve vedere l'ultima scrittura precedente a  $v$  nell'ordine "happens-before".

- Thread T1:

```
java
x = 1; // Scrittura
y = x; // Lettura e scrittura
```

Copia Modifica

- Thread T2:

```
java
int r = y; // Lettura
```

Copia Modifica

Nell'ordine "happens-before":

1. T1 scrive  $x = 1$ .
2. T1 usa il valore di  $x$  per aggiornare  $y$ .
3. T2 legge  $y$  e deve vedere il valore aggiornato da T1 (ossia  $y = 1$ ).

## Legal executions

Un'esecuzione legale di un programma multithread si costruisce passo dopo passo, seguendo un processo iterativo. Ad ogni passo, viene scelto un gruppo di azioni di memoria da "commettere", cioè da eseguire e rendere parte dell'esecuzione globale. Tuttavia, ogni azione può essere commessa solo se è compatibile con tutte le azioni già

commesse nei passi precedenti e rispetta le regole di coerenza del programma e del modello di memoria.

Questa idea di esecuzione "ben formata" è importante perché:

1. **Evita comportamenti indesiderati:** Come letture incoerenti o risultati imprevedibili dovuti a conflitti tra thread (data race).
2. **Consente ottimizzazioni del compilatore:** Permette al compilatore di riordinare o ottimizzare le istruzioni senza introdurre errori, a patto che il comportamento visibile al programma rimanga valido.

## Causality requirements for executions

Le regole del **Java Memory Model (JMM)** servono a decidere se una sequenza di operazioni di un programma multithread è valida o no. Per essere considerata **legale**, un'esecuzione deve rispettare queste fasi:

1. **Costruzione progressiva:** Le operazioni (come letture e scritture di variabili) vengono eseguite gradualmente. Ogni gruppo di operazioni viene verificato per essere compatibile con le regole del programma e del modello di memoria.
2. **Sottoinsiemi di azioni:** Immagina di dividere tutte le operazioni A in piccoli gruppi C<sub>0</sub>, C<sub>1</sub>, ..., C<sub>n</sub>, dove ogni gruppo contiene alcune operazioni "commesse" in un determinato momento. Alla fine, l'insieme completo delle operazioni A deve essere stato verificato.
3. **Esecuzioni intermedie valide:** Ogni gruppo di operazioni commesso deve poter essere spiegato da un'esecuzione parziale ben formata. Questo garantisce che ogni passo sia coerente con il programma e che non ci siano errori come letture incoerenti.
4. **Risoluzione dei conflitti:** Se ci sono conflitti tra operazioni (come due thread che scrivono sulla stessa variabile), si stabilisce un ordine chiaro su quale operazione avviene prima.
5. **Verifica delle regole:** Dopo ogni passo, si controlla che tutto rispetti le regole del JMM:
  - Nessun **data race** (accessi concorrenti non sincronizzati).
  - L'ordine delle azioni deve essere coerente con il programma e con le sincronizzazioni.

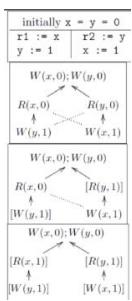
Alla fine, se tutte le operazioni A sono state commesse rispettando le regole, l'esecuzione è considerata legale.

## Intuizione

L'idea è di partire **analizzando tutte le possibili** esecuzioni del programma che rispettano la *consistenza sequenziale*.

Durante l'analisi delle esecuzioni, **individua** i punti in cui due operazioni portano a un **data race**, ovvero dove almeno una di queste operazioni è una scrittura e non esiste sincronizzazione tra i due accessi.

Una volta identificato un **data race**, decidi come **risolverlo**. Questo processo viene chiamato "**commit**": scegli un ordine esplicito per gli accessi coinvolti. Se due thread competono per scrivere su una variabile, scegli quale operazione **avviene per prima**. Dopo aver risolto uno (o più) *data race*, rianalizza tutte le esecuzioni possibili con le scelte (i "commit") appena fatte. Continua fino a quando tutte le ambiguità sono risolte.



## Properties of the formal JMM

**Riordinamento di istruzioni indipendenti:** Il JMM consente di dimostrare che il riordinamento di due istruzioni consecutive *indipendenti* è una trasformazione legale del programma.

**Comportamento sequenziale per programmi correttamente sincronizzati:**

I programmi che utilizzano una sincronizzazione corretta, l'ordine delle operazioni coincide con **un'esecuzione sequenziale**.

**Rimozione di sincronizzazioni ridondanti:** Se una sincronizzazione **non è necessaria** per mantenere la correttezza del programma, può essere **eliminata** senza effetti collaterali.

**Trattamento dei campi volatile di oggetti locali ai thread:** I campi dichiarati come volatile, ma appartenenti a oggetti utilizzati solo localmente all'interno di un singolo thread, possono essere trattati come normali campi.

## JMM from the programmer perspective

Il JMM offre un insieme di regole pratiche per garantire la correttezza del programma, che si dividono in tre categorie fondamentali:

**Atomicità:** L'atomicità definisce quali operazioni sono intrinsecamente **indivisibili**, ossia non possono essere interrotte o osservate in uno stato parziale da altri thread.

**Visibilità:** La visibilità stabilisce quando una modifica a una variabile in un thread diventa visibile ad altri thread.

**Riordinamento:** Il riordinamento descrive come il compilatore e il processore possono **cambiare l'ordine delle istruzioni** per migliorare le prestazioni, purché la semantica del programma sia rispettata.

### Regole di Atomicità

Un'operazione è considerata **atomica** se, dal punto di vista di qualsiasi altro thread, i suoi effetti sono osservati **completamente** o **per nulla**, ma mai in uno stato intermedio.

In Java sono considerate atomiche le letture e scritture di **tutti i tipi primitivi** (es. int, float, char, boolean, etc.), eccetto long e double, e dei **tipi di riferimento**.

Per tutte le variabili (inclusi long e double) dichiarate con il modificatore volatile, sia la lettura che la scrittura sono sempre **atomicamente garantite**.

La lettura e scrittura di long e double non sono atomiche perché vengono divise in due operazioni parziali.

### Regole di visibilità

La regola di **visibilità** nel JMM stabilisce quando le operazioni di un thread diventano visibili agli altri thread. La visibilità è garantita dalle seguenti operazioni specifiche:

**Acquisizione di un monitor con synchronized:** Quando un thread entra in un blocco synchronized, diventa visibile tutto ciò che è stato scritto in memoria dall'ultimo thread che ha posseduto quel monitor, fino al momento in cui lo ha rilasciato.

**Lettura di una variabile volatile:** Quando un thread legge una variabile dichiarata volatile, acquisisce visibilità su tutte le operazioni effettuate dal thread che ha scritto per ultimo su quella variabile.

**Invocazione di t.start():** Quando un thread chiama start() su un altro thread t, tutte le operazioni effettuate dal thread chiamante prima dell'invocazione diventano visibili al nuovo thread t.

**Ritorno da un'operazione t.join():** Quando un thread chiama join() su un altro thread t e attende la sua terminazione, al completamento del join tutte le operazioni eseguite dal thread t diventano visibili al thread chiamante.

## Regole di Riordinamento

Le regole di riordinamento permettono al compilatore, alla JVM e alla CPU di ottimizzare l'esecuzione delle istruzioni, ma costrutti come volatile e synchronized limitano il riordinamento per garantire la correttezza in contesti multithread.

**Letture di una variabile volatile o inizio di un blocco synchronized:** Tutte le istruzioni che appaiono nel codice **prima** di una lettura di una variabile volatile o dell'inizio di un blocco sincronizzato devono essere eseguite **prima** di tale punto, ovvero non possono essere riordinate e **spostate** dopo l'inizio del blocco synchronized/volatile.

```
x = 10; // (1)  
int y = 20; // (2)  
int z = volatileVar; // (3) Le istruzioni (1) e (2) non possono essere riordinate dopo la  
lettura di volatileVar (3).
```

**Scritture su una variabile volatile o fine di un blocco synchronized:**

Tutte le istruzioni che appaiono nel codice **dopo** una scrittura su una variabile volatile o la fine di un blocco sincronizzato devono essere eseguite **dopo** tale punto. Non possono essere riordinate e spostate prima.

```
volatileVar = x; // (1)  
y = 20; // (2) L'istruzione (2) non può essere riordinata prima della scrittura (1)
```

**Istruzioni normali (non volatile o synchronized):** Le istruzioni normali possono essere riordinate liberamente.

Can instructions **x1** and **x2** be swapped?

Type of x2 Type of x1	Normal	Volatile read / Synchronized start	Volatile write / Synchronized end
Normal	Yes	Yes	No
Volatile read / Synchronized start	No	No	No
Volatile write / Synchronized end	Yes	No	No

## AP-23: RUST

**Rust** è un linguaggio di programmazione orientato ai sistemi, progettato per combinare alte prestazioni con una forte sicurezza nella memoria.

Supporta paradigmi funzionali e imperativi. La sua sintassi, simile a C e C++, include costrutti come if-else, while, for e match per il pattern matching. A livello semantico, è influenzato dai linguaggi ML e Haskell, privilegiando composizionalità ed espressività, con un approccio in cui quasi tutto è considerato un'espressione.

Rust non utilizza un **runtime dedicato**, come garbage collection o binding dinamico, offrendo agli sviluppatori pieno controllo sulla gestione della memoria. Combina **prestazioni** paragonabili a C con una rigorosa **sicurezza** della memoria, garantendo che i puntatori siano validi e prevenendo accessi fuori limite agli array. Questo è possibile grazie a verifiche statiche del compilatore, che evitano costi di esecuzione aggiuntivi e sfruttano astrazioni a costo zero per una gestione efficiente delle risorse.

La **sicurezza** di Rust si basa su un avanzato sistema di tipi e sui concetti di proprietà (ownership), prestito (borrowing) e durata (lifetime), che eliminano problemi comuni come puntatori nulli, puntatori pendenti, doppia liberazione della memoria, corse di dati (data races) e invalidazione di iteratori. Rust impedisce questi errori, anche in ambienti concorrenti, garantendo l'integrità della memoria. Se un programma Rust

viene compilato con successo, si ha la certezza che non presenterà questi problemi durante l'esecuzione.

Tuttavia, questa sicurezza richiede un maggiore **sforzo cognitivo**: gli sviluppatori devono considerare attentamente le regole di gestione della memoria durante la scrittura del codice. Nonostante ciò, Rust offre una soluzione unica per creare software robusto e performante, bilanciando controllo, sicurezza e prestazioni. La sicurezza della memoria in Rust è ottenuta attraverso una combinazione tra scelte linguistiche, politiche di gestione della memoria e analisi statica avanzata.

## Null pointers

Il problema dei **null pointers** si verifica quando si tenta di accedere a una variabile che contiene un valore NULL. Gli approcci utilizzati per affrontare il problema sono **Far analizzare al compilatore il codice** per assicurarsi che ogni variabile venga inizializzata prima di essere utilizzata o **Assegnare un valore predefinito** alle variabili al momento della loro dichiarazione.

In Java, vengono adottate entrambe le soluzioni, in particolare la soluzione 1 per le variabili locali nei metodi, poiché il compilatore può facilmente analizzare il flusso di dati all'interno dello scope ristretto di un metodo e la soluzione 2 per le variabili di istanza e statiche, poiché l'assegnazione di valori predefiniti è più semplice per variabili globali.

Rust **evita completamente** i *null pointers*. Il valore NULL non esiste in Rust. Tutte le variabili devono **essere inizializzate** al momento della dichiarazione o **prima di essere usate**. Se un ramo del codice non assegna un valore a una variabile, il compilatore genera un errore.

Le variabili statiche e globali devono essere inizializzate durante la dichiarazione, eliminando la possibilità di stati non definiti.

Per gestire situazioni in cui un valore potrebbe essere **assente**, Rust utilizza il tipo generico **Option<T>**, che rappresenta un valore opzionale. Questo approccio è simile al concetto di Maybe in Haskell o Optional in Java.

```
enum Option<T> {
    None, // Indica l'assenza di un valore
    Some(T) // Contiene un valore di tipo T
}
```

Questo costrutto forza i programmatore a gestire esplicitamente la presenza o assenza di un valore, eliminando i rischi associati ai puntatori nulli.

Inoltre il compilatore di Rust esegue controlli rigorosi per garantire che non vi siano riferimenti a variabili non inizializzate o puntatori nulli.

## Digression: Primitive types in Rust

In Rust, i tipi primitivi sono progettati per offrire flessibilità e sicurezza, con un sistema di tipi rigoroso e verifiche a tempo di compilazione.

**Interi con segno:** i8, i16, i32, i64, isize (adatti per numeri negativi e positivi).

**Interi senza segno:** u8, u16, u32, u64, usize (solo numeri positivi).

**Virgola mobile:** f32, f64 (per numeri decimali).

**Booleano:** bool (valori true o false).

**Carattere:** char, che rappresenta caratteri Unicode a 4 byte.

Rust può **inferire il tipo** di una variabile al momento della dichiarazione con **let**. Tuttavia, in caso di ambiguità (es. let k = 3;), si devono fornire annotazioni di tipo per chiarire.

Le **tuple** in Rust sono simili a quelle di Haskell e possono contenere elementi di tipi diversi:

```
let tup = (500, 6.4, 1); // Una tupla con tipi misti
let (x, y, z) = tup;    // Destrutturazione della tupla
println!("The value of y is: {}", y); // Accesso a un elemento
println!("The value of tup.1 is: {}", tup.1); // Accesso diretto tramite indice
```

Gli **array** in Rust hanno lunghezza fissa e sono sottoposti a controlli a runtime per gli accessi fuori limite:

```
let a: [i32; 5] = [1, 2, 3, 4, 5]; // Array di interi di lunghezza 5
let b: [i32; 6] = [3; 6];        // Array con 6 elementi, tutti uguali a 3
```

Se si tenta di accedere a un indice fuori dai limiti dell'array, Rust **genererà un errore a runtime** per prevenire comportamenti imprevisti.

Rust non consente l'overloading nelle variabili, quindi è necessario specificare il tipo nei contesti ambigui, ad esempio:

```
let k: u8 = 3; // Specifica il tipo per chiarire
let n: f32 = 3.0; // Virgola mobile a 32 bit
```

## Option

In Rust, il tipo generico **Option<T>** è utilizzato per rappresentare valori opzionali. Può assumere valore **None**, che indica l'assenza di un valore o **Some(T)**, che contiene un valore di tipo T.

### Using Option

```
enum std::option::Option<T> {
    None,
    Some(T)
}
```

```
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        None
    } else {
        Some(dividend / divisor)
    }
}

fn try_division(dividend: i32, divisor: i32) {
    // `Option` values can be pattern matched, just like other enums
    match checked_division(dividend, divisor) {
        None => println!("{} / {} failed!", dividend, divisor),
        Some(quotient) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        } }
}

fn main() {
    try_division(54, 9); try_division(7, 0);
    let opt_float = Some(0f32);
    // Unwrapping a `Some` variant will extract the value wrapped.
    println!("{} unwraps to {}", opt_float, opt_float.unwrap());
}
```

```

pub fn main() {
    let name1: Option<&str> = None;
    // In this case, nothing will be printed out
    if let Some(name) = name1 {
        println!("{}"), name);
    }

    let name2: Option<&str> = Some("Matthew");
    // In this case, the word "Matthew" will be printed out
    if let Some(name) = name2 {
        println!("{}"), name);
    }
}

```

In Rust, il costrutto **if let** è utilizzato per **verificare e destrutturare** i valori di tipi come Option<T>. Permette di controllare se un valore **corrisponde** a Some o None.

Quindi if let Some(name) = variable verifica se una variabile di tipo Option contiene un valore (Some). Se sì, il valore viene "destrutturato" nella variabile name e può essere utilizzato nel blocco if. Se no, il blocco if viene saltato.

## Dangling pointers, double free and race condition: example in C++

I **dangling pointers** in C++ si verificano quando un puntatore fa riferimento a una posizione di memoria non più valida, come memoria deallocated, memoria al di fuori dei limiti di un array o memoria allocata sullo stack che è stata liberata automaticamente al termine dello scope. Questo porta a comportamenti imprevedibili.

Il codice fornito dimostra un caso di *dangling pointer*:

```

// C++ code
string *s;
{
    string s1 = "scope 1";
    s = &s1;
}
{
    string s2 = "scope 2";
}
cout << *s << endl;

```

All'interno del primo blocco {}, la variabile s1 è allocata sullo stack, e il puntatore s viene impostato per puntare a s1.

Quando il blocco termina, s1 esce dallo scope e la memoria allocata per s1 viene invalidata. Tuttavia, il puntatore s continua a puntare alla posizione di memoria precedentemente occupata da s1.

Nel secondo blocco {}, viene allocata una nuova variabile s2, che può anche occupare lo stesso spazio di memoria di s1 (riutilizzo dello stack).

Il puntatore s è ora un *dangling pointer* e, accedendo al valore puntato da s, i risultati sono imprevedibili.

**Random results:** Il contenuto della memoria potrebbe sembrare ancora valido ma non lo è.

**Segmentation fault:** Accedere a una posizione di memoria non valida può portare a un crash.

**Corruption of memory manager:** Utilizzare un *dangling pointer* per scrivere nella memoria potrebbe corrompere il gestore della memoria.

## Double free

Il problema del **double free** si verifica quando un'area di memoria allocata nello heap viene **deallocata** due volte.

```
// C++ code
auto *s1 = new string("example");
auto *s2 = s1;
// ...
delete s1;
delete s2;
```

**Corruzione del gestore della memoria:** Il doppio tentativo di liberare la stessa memoria può compromettere lo stato interno del gestore della memoria.

**Segmentation fault:** Tentare di accedere o liberare memoria già deallocated può portare al crash.

**Exploitation:** Gli attaccanti possono sfruttare un errore di doppia deallocazione per alterare il flusso di esecuzione del programma o corrompere la memoria.

## Race condition

Una **race condition** si verifica quando il risultato di un programma dipende dall'**ordine** in cui più **thread accedono** e modificano una risorsa condivisa.

```
// C++ code
int main() {
    int counter = 0;
    const auto task = [&] {
        for (int i = 0; i < 100000; ++i) {
            counter++;
        }
    };
    thread thread1(task);
    thread thread2(task);
    thread1.join();
    thread2.join();
    cout << counter << endl;
    return 0;
}
```

La variabile counter è condivisa tra i due thread. Ogni thread esegue un ciclo in cui incrementa il valore di counter 100.000 volte. Tuttavia, l'incremento di counter non è un'operazione atomica. È composta da più passaggi: Legge il valore attuale di counter. Aggiunge 1 al valore letto. Scrive il nuovo valore nella memoria. Il valore finale di counter è spesso inferiore a 200.000, nonostante ogni thread esegua 100.000 incrementi.

## Gestione della memoria in Rust

La gestione della memoria in Rust si basa su due aree principali:

**Stack:** per variabili con ciclo di vita noto e strutture allocate staticamente.

**Heap:** per dati allocati dinamicamente, quando la dimensione o il tempo di vita non possono essere determinati a tempo di compilazione.

Rust privilegia l'uso dello **stack**, che è più veloce e gestito automaticamente.

L'**allocazione dinamica** sull'heap viene effettuata esplicitamente mediante l'uso del tipo **Box<T>**, che rappresenta un riferimento a un valore allocato sullo heap. Per accedere a un valore allocato sull'heap, viene utilizzata la deferenziazione tramite **\***.

```
fn main() {
    let x = 3;    // 'let' allocates a variable on the stack
    let y = Box::new(3); // y is a reference to 3 on the heap
    println!("x == y is {}", x == *y); // "x == y is true"
}
```

I linguaggi moderni utilizzano garbage collection, introducendo ritardi nell'esecuzione. Linguaggi come C affidano al programmatore la responsabilità di allocare e deallocare, con il rischio di errori di memoria. Rust utilizza il paradigma **Resource Acquisition Is Initialization**.

## Immutability by default

In Rust, per impostazione **predefinita**, tutte le variabili sono **immutabili**, il che significa che una volta assegnato un valore, questo non può essere modificato. Se si tenta di modificare una variabile immutabile, il compilatore genererà un errore.

Per consentire modifiche, è necessario dichiarare esplicitamente la variabile come **mutable** utilizzando la parola chiave **mut**.

```
fn main() {  
    let a: i32 = 0;  
    a = a + 1;  
    println!("a == {}", a);  
}
```

```
rustc 1.14.0 (e8a012324 2016-12-16)  
error[E0384]: re-assignment of immutable variable `a`  
--> <anon>:3:5  
|  
2 |     let a: i32 = 0;  
|         - first assignment to `a`  
3 |     a = a + 1;  
|         ^^^^^^^^^^ re-assignment of immutable variable  
error: aborting due to previous error
```

```
fn main() {  
    let mut a: i32 = 0;  
    a = a + 1;  
    println!("a == {}", a);  
}
```

```
rustc 1.14.0 (e8a012324 2016-12-16)  
a = 1  
Program ended.
```

18

## Resource Acquisition Is Initialization (RAII)

Il paradigma **Resource Acquisition Is Initialization (RAII)** stabilisce che l'allocazione e la deallocazione delle risorse siano gestite **automaticamente** attraverso il **lifetime** degli oggetti. Una risorsa viene allocata durante l'**inizializzazione** di un oggetto (generalmente nel costruttore) e viene **deallocata** quando l'oggetto **esce dallo scope**, sfruttando il distruttore.

In RAII, ogni risorsa ha un proprietario unico, che è responsabile del suo intero ciclo di vita. Questo approccio è particolarmente popolare in **C++ moderno**. Oggetti di piccole dimensioni sono solitamente allocati nello stack per motivi di efficienza, mentre risorse più grandi, come memoria dinamica o file, vengono allocate sull'heap o in altre aree. In questi casi, l'oggetto allocato nello stack si fa carico della gestione della risorsa sottostante e provvede a liberarla nel suo distruttore.

## Regole dell'ownership

Il **sistema di ownership** in Rust è un meccanismo fondamentale che gestisce la memoria e le risorse in modo sicuro e automatico, supportando il paradigma **RAII** in modo rigoroso. Si basa su due concetti principali: **ownership** (proprietà) e **borrowing** (prestito).

L'ownership segue queste tre regole fondamentali:

**[01] Ogni valore è posseduto da una variabile, che lo identifica tramite un nome.**

*let x = 10; // `x` è il proprietario del valore 10*

**[02] Ogni valore può avere al massimo un proprietario alla volta:** Quando la proprietà di un valore viene trasferita, il proprietario originale perde l'accesso al valore.

```
let s1 = String::from("hello");  
let s2 = s1; // `s1` trasferisce la proprietà a `s2`  
// println!("{}", s1); // Errore: `s1` non è più valido
```

**[03] Quando il proprietario esce dallo scope, il valore viene deallocato.**

```

{
    let s = String::from("hello");
    // `s` è valido solo all'interno di questo blocco
}
// Qui, la memoria di `s` viene automaticamente rilasciata

```

## Move semantics

Per impostazione predefinita, quando si assegna una variabile a un'altra con `=`, l'ownership del valore viene *spostata* (mossa) dalla variabile di origine (**right-hand side**) alla variabile di destinazione (**left-hand side**). Questo soddisfa la regola [O2] del sistema di ownership, che consente a un valore di avere un solo proprietario alla volta.

```

fn main() {
    let x = Box::new(3);
    let _y = x; // underscore to avoid 'unused' warning
    println!("x = {}", x); // error!
}

```

## Copy semantics

Per tipi **primitivi** o tipi che implementano il *trait Copy*, l'assegnazione **non sposta** l'ownership ma **copia** il valore, creandone uno nuovo. Ciò significa che entrambi i nomi possono continuare a essere utilizzati indipendentemente.

```

fn main() {
    let x = 3;
    let _y = x;
    println!("x = {:?}", x); // OK
}

```

### Move semantic e passaggio di parametri

La move semantic viene applicata solo a **tipi complessi** che non implementano il *trait Copy* (es. `String`, `Box<T>`).

La **move semantic** si applica anche al passaggio dei parametri e alla restituzione dei valori nelle funzioni. Quando un valore viene passato a una funzione, l'ownership del valore viene trasferita al parametro formale della funzione. Se la funzione restituisce il valore, l'ownership viene trasferita al chiamante.

<pre> fn foo&lt;T&gt;(z: T) -&gt; T { // polymorphic identity function     z }  fn main(){     let x = Box::new(3);     let _y = foo(x);     println!("x == {}", x); // error } </pre>	<pre> fn main(){     let x = 3;     let _y = foo(x);     println!("x == {}", x); // OK } </pre>
--	---

Per i tipi primitivi (o che implementano il trait `Copy`), il valore viene copiato invece di essere spostato. In questo caso, sia `x` che `_y` restano validi e utilizzabili.

Quando un valore viene passato a una funzione, esso viene deallocated al termine della funzione (quando il parametro formale esce dallo scope), a meno che non venga restituito.

```

struct Dummy { a: i32, b: i32 }

fn main() {
    let mut res = Box::new(Dummy { a: 0, b: 0 }); // `res` possiede l'oggetto allocato nello heap
    take(res); // L'ownership di `res` viene trasferita alla funzione `take`
    println!("res.a = {}", res.a); // Errore di compilazione: `res` non è più valido
}

fn take(arg: Box<Dummy>) {
    // `arg` ora possiede la risorsa
    // Quando `take` termina, `arg` esce dallo scope e la risorsa viene liberata
}

```

## Double free in Rust

In Rust, il problema del **double free** è completamente perché ogni risorsa ha un unico proprietario e viene deallocated automaticamente quando il proprietario esce dallo scope.

```
// Codice C++
auto *s1 = new string("esempio");
auto *s2 = s1;
// ...
delete s1;
delete s2;
```

Il problema nasce perché s1 e s2 condividono il puntatore alla stessa risorsa. Dopo che s1 la dealloca, un secondo tentativo con delete s2 causa un errore di *double free*.

```
// Rust code
let s1 = String::new("esempio");
let s2 = s1;
```

Quando s2 riceve l'ownership, s1 non è più valido. Il compilatore impedisce di utilizzare s1 dopo il trasferimento, evitando accessi o deallocazioni non validi. Quando s2 esce dallo scope, la risorsa viene deallocated automaticamente.

## Borrowing

In Rust il **borrowing** (*prestito*) permette di accedere a una risorsa **senza trasferirne** l'ownership, superando le limitazioni delle regole di ownership. È utile quando una funzione o un'operazione ha bisogno di utilizzare una risorsa **senza possederla**.

**Riferimento immutabile (&):** Consente di leggere il valore **senza modificarlo**. Possono esistere più riferimenti immutabili contemporaneamente.

```
let y = 5;
let x = &y; // x è un riferimento immutabile a y
println!("{}", x); // OK
```

**Riferimento mutabile (&mut):** Consente di **modificare** il valore. Può esistere **solo un riferimento mutabile** alla volta.

```
let mut y = 5;
let x = &mut y; // x è un riferimento mutabile a y
*x += 1; // Modifica y attraverso x
println!("{}", y); // Stampa 6
```

**Per valore (Ownership std):** L'ownership viene trasferita al nuovo proprietario (senza borrowing). La variabile originale non è più utilizzabile.

```
let y = String::from("hello");
let x = y; // Ownership di y è trasferita a x
// println!("{}", y); // Errore: y non è più valido
```

## Regole del borrowing

Le **regole del borrowing** in Rust garantiscono la sicurezza della memoria evitando conflitti tra aliasing e mutabilità. Ecco le regole principali:

**[B1] Un solo riferimento mutabile alla volta:** È permesso avere un riferimento mutabile, ma non più di uno contemporaneamente.

**[B2] Nessun riferimento immutabile con un mutabile:** Se esiste un riferimento mutabile, non possono esistere riferimenti immutabili alla stessa risorsa nello stesso momento.

**[B3] Più riferimenti immutabili sono consentiti:** Se non ci sono riferimenti mutabili, è possibile avere più riferimenti immutabili contemporaneamente.

**[B4] L'ownership è sospesa durante il borrowing immutabile:** Il proprietario non può modificare o liberare la risorsa mentre è presa in prestito in modo immutabile.

**[B5] Nessuna lettura durante un borrowing mutabile:** Il proprietario non può leggere la risorsa mentre esiste un riferimento mutabile.

### Esempio [B1], [B2], [B3]

```
let mut x = 10;
// Riferimento immutabile
let r1 = &x;
let r2 = &x; // OK: più riferimenti immutabili
// let r3 = &mut x; // Errore: riferimento mutabile non permesso insieme a immutabili
```

### Esempio [B4], [B5]

```
let mut x = 10;
// Borrow mutabile
let r = &mut x;
// println!("{}", x); // Errore: il proprietario non può leggere mentre esiste un borrow
*mutable
*r += 1; // Modifica permessa attraverso il riferimento mutabile
```

## Stringhe in Rust

In Rust, esistono due tipi principali per rappresentare stringhe, con caratteristiche e utilizzi distinti, **String** e **&str**.

**String:** La stringa viene allocata nello **heap**. La lunghezza non deve essere nota a tempo di compilazione, quindi è dinamica. È modificabile (mutabile) e può crescere o ridursi in base ai dati.

**&str:** La stringa viene allocata sullo **stack**. La lunghezza deve essere **nota** staticamente a tempo di compilazione. È **immutabile** (non può essere modificata).

Il metodo **String::from()** converte una stringa **&str** in un oggetto **String**, allocando memoria dinamicamente nello heap:

```
let slice: &str = "hello";
let string = String::from(slice); // Converte `&str` in `String`
```

Un oggetto **String** incapsula **un puntatore** che fa riferimento alla sequenza di caratteri allocata nello heap, la **capacità** che indica la quantità di memoria allocata nello heap e la **lunghezza** che indica il numero di caratteri attualmente contenuti

Poiché **String** non implementa il *trait Copy*, l'assegnazione tramite l'operatore **=** trasferisce l'ownership (*move semantics*). Solo il puntatore, la capacità e la lunghezza vengono copiati, **non** la sequenza di caratteri nello heap.

## Dangling pointers: not in Rust

In Rust, i **dangling pointers** (puntatori pendenti) non sono permessi grazie al sistema di **ownership** e alle regole del **borrowing**. L'esempio evidenzia come Rust previene un errore comune che può verificarsi in C++.

The screenshot shows two code snippets side-by-side. On the left is Rust code, and on the right is C++ code. Both snippets attempt to print the value of a variable 's' after it has been moved or dropped.

**Rust Code:**

```
fn main() { // Rust code
    let s;
    {
        let s1 = String::from("scope 1");
        s = &s1;
    }
    {
        let _s2 = String::from("scope 2");
    }
    println!("s == {}", s);
}
```

**C++ Code:**

```
string *s; // C++ code
{
    string s1 = "scope 1";
    s = &s1;
}
{
    string s2 = "scope 2";
}
cout << *s << endl;
```

Below the code, the terminal output shows:

```
error[E0597]: `s1` does not live long enough
--> src\main.rs:7:13
|
7 |         s = &s1;
|             ^^^ borrowed value does not live long enough
8 |     }
|     - `s1` dropped here while still borrowed
...
12 |     println!("s == {}", s);
```

The error message indicates that the variable 's1' does not live long enough because it is dropped before its value is printed.

Rust impedisce che il riferimento s diventi pendente applicando la regola [B4]: Quando s1 esce dallo scope, la memoria associata viene liberata automaticamente. Poiché s aveva preso in prestito s1, Rust segnala un errore di compilazione: s1 does not live long enough.

Il borrow checker impedisce di usare riferimenti invalidi, applicando le regole [B4] e [B5]. Ogni accesso a una risorsa viene validato staticamente, prevenendo errori a runtime come accessi a memoria non valida.

```
fn main() {
    let mut s = String::from("ex-1");
    println!("s-0 == {}", s);
    let t = &mut s;
    *t = String::from("ex-2");
//    println!("s-1 == {}", s); // what happens if uncommented?
    println!("t == {}", t);
    println!("s-2 == {}", s);
    let z = &s;
    println!("s-3 == {}", s);
    let w = z;
    println!("{}, {}, {}", z, w, s);
}
```

```
s-0 == ex-1
t == ex-2
s-2 == ex-2
s-3 == ex-2
ex-2,ex-2,ex-2
```

// `println!("s-1 == {}", s);` Se scommentata, questa riga genera un errore di compilazione, perché il riferimento mutabile t esiste ancora nel momento in cui si tenta di accedere direttamente a s. Questo viola la regola [B5] (nessuna lettura del valore da parte del proprietario mentre esiste un riferimento mutabile).

## Lifetimes

In Rust, i **lifetimes** sono un concetto utilizzato dal **borrow checker** per garantire che le regole di ownership e borrowing siano rispettate. Ogni ownership e ogni borrowing sono associati a un lifetime, che rappresenta l'intervallo di validità di una risorsa.

Il lifetime di un'ownership **inizia** quando la risorsa viene **creata** e termina quando l'ownership viene **trasferita** o la risorsa viene **distrutta**.

Per i **borrowing**, il lifetime termina quando il **valore** preso in prestito viene **usato** per l'ultima volta.

La maggior parte dei lifetimes viene dedotta automaticamente dal compilatore. Tuttavia, in alcuni casi complessi, è necessario specificarli esplicitamente utilizzando una sintassi simile a quella dei generics, come '`a`'.

Il compilatore utilizza i lifetimes per verificare che: Ogni riferimento preso in prestito sia valido per tutta la durata del suo utilizzo AND Il proprietario della risorsa abbia un lifetime più lungo del borrower, in modo da evitare *dangling references*.

```

fn main() {
    let x = 5;      // Ownership di `x` inizia qui
    let r = &x;      // `r` prende in prestito `x`
    println!("{}", r); // Ultimo uso di `r`, il suo lifetime termina qui
}                  // Lifetime di `x` termina qui

```

In alcune funzioni, i lifetimes devono essere specificati esplicitamente per chiarire la relazione tra i riferimenti:

```

fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}

```

In questo caso, **a** è un lifetime generico che rappresenta la durata comune dei riferimenti. **&'a str** indica che i riferimenti s1 e s2 condividono lo stesso lifetime 'a. **-> &'a str** garantisce che il riferimento restituito avrà un lifetime pari al più breve tra quelli di s1 e s2.

Il lifetime esplicito garantisce che il riferimento restituito sarà valido solo finché lo sono entrambi i parametri di input. Non ci saranno riferimenti pendenti o invalidi dopo che uno dei parametri è stato deallocated.

## Regole dei lifetimes nelle funzioni

**[R1]** Ogni parametro borrowed ha, per impostazione predefinita, un lifetime distinto.

**[R2]** Se c'è un solo lifetime di input, viene assegnato automaticamente all'output.

**[R3]** Se una funzione o metodo ha più lifetimes di input, ma uno è associato a **&self** o **&mut self**, questo lifetime viene automaticamente assegnato all'output.

Se il compilatore non può inferire i lifetimes in modo univoco, è necessario dichiararli esplicitamente.

```

fn longest(s1: &str, s2: &str) -> &str { //does not compile
    if s1.len() > s2.len() { s1 }
    else { s2 }
}

```

Il compilatore non può sapere se il riferimento restituito (&str) ha il lifetime di s1 o di s2. Poiché i lifetimes non sono esplicitati, il compilatore non riesce a verificare la validità della funzione.

```

fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() { s1 }
    else { s2 }
}

```

```

// `print_refs` takes two references to `i32` which have different
// lifetimes `'a` and `'b` (passed as generic parameters).
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}

```

Questa funzione accetta due riferimenti a valori di tipo i32 con **lifetimes distinti**, 'a e 'b.

```

// A function whith no arguments but with a lifetime parameter `'a`.
fn failed_borrow<'a>() {
    let _x = 12;
    // ERROR: `_x` does not live long enough
    // let y: &'a i32 = &_x; // uncomment this!
    // The lifetime of `& x` is shorter than that of `y`.
    // A short lifetime cannot be coerced into a longer one.
}

```

```

fn main() {
    let (four, nine) = (4, 9); // Create variables to be borrowed
    print_refs(&four, &nine); //Borrows of both variables are passed
    // The lifetime of `four` and `nine` must
    // be longer than that of `print_refs`.
    failed_borrow();
}

```

Il valore `_x` ha un lifetime molto breve, limitato allo scope locale della funzione. Rust impedisce di assegnare `_x` a un riferimento con un lifetime `'a`, che potrebbe essere più lungo di `_x`. Questo violerebbe la sicurezza della memoria.

Nella chiamata a `print_refs`, i riferimenti a `four` e `nine` vengono passati rispettivamente a `x` e `y`. I loro lifetimes devono essere almeno pari alla durata della funzione `print_refs`. La funzione `failed_borrow` produce un errore se la riga commentata viene sbloccata, perché `_x` non vive abbastanza a lungo per soddisfare il lifetime `'a`.

## Enum

In Rust, gli **enum** sono tipi di dati algebrici usati per rappresentare valori che possono avere uno tra diversi stati, analoghi alle union di C/C++.

```
enum RetInt {
    Fail(u32),
    Succ(u32)
}

fn foo_may_fail(arg: u32) -> RetInt {
    let fail = false;
    let errno: u32;
    let result: u32;
    ...
    if fail {
        RetInt::Fail(errno)
    } else {
        RetInt::Succ(result)
    }
}
```

In Rust, gli **enum** possono essere utilizzati per definire strutture dati algebriche generiche, come alberi binari.

```
#[derive(Debug)] // needed to print
enum Tree<T> {
    Empty,
    Node(T, Box<Tree<T>>, Box<Tree<T>>)
}

fn main() {
    let tree = Tree::Node(
        42,
        Box::new(Tree::Node(
            0,
            Box::new(Tree::Empty),
            Box::new(Tree::Empty)
        )),
        Box::new(Tree::Empty));
    println!("{:?}", tree);
    // prints Node(42, Node(0, Empty, Empty), Empty)
}
```

## Pattern Matching

In Rust, il **pattern matching** è uno costrutto, simile allo switch, che consente di confrontare valori con modelli predefiniti, eseguendo azioni specifiche per ogni corrispondenza.

Il compilatore impone che tutti i casi siano gestiti. Se manca un caso, il programma non compila. È possibile utilizzare `_` come caso di default per gestire tutti i valori non esplicitamente indicati.

```

fn main() {
    let x = 5; // try others...

    match x {
        1                  => println!("one"),
        2                  => println!("two"),
        3|4                => println!("three or four"),
        5..=10             => println!("five to ten"),
        e @ 11..=20         => println!("{} {}", e),
        i32::MIN..=0        => println!("less than zero"),
        21...               => println!("large"),
        _                  => println!("????"),
    }
}

```

In Rust, le **struct** e il blocco **impl** sono usati insieme per definire classi e metodi, offrendo funzionalità simili alla programmazione orientata agli oggetti, ma senza ereditarietà.

```

#[derive(Debug)]
struct Rectangle {      // class
    width: u32,          // instance variable
    height: u32,
}

impl Rectangle {        // methods
    fn area(&self) -> u32 {           // first argument is this
        self.width * self.height // try to change width...
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!(
        "The area of the rectangle is {} square pixels.", rect1.area()
    );
}

```

**No inheritance in RUST! → Pushing composition over inheritance**

## Traits

In Rust, i **traits** sono un modo per definire comportamenti condivisi tra diversi tipi, simili a **interfacce** in Java e **type classes** in Haskell.

Un trait può definire **metodi astratti** (da implementare) e **metodi concreti** (con un'implementazione predefinita). Non può contenere campi o variabili.

Una struct può implementare un trait **fornendo definizioni** per tutti i suoi metodi **astratti**:

```

trait TraitName {
    fn abstract_method(&self);
    fn concrete_method(&self) {
        println!("Default implementation");
    }
}

struct MyStruct;

impl TraitName for MyStruct {
    fn abstract_method(&self) {
        println!("Implemented method");
    }
}

```

Alcuni trait, come Debug o Clone, possono essere implementati automaticamente con l'annotazione **#[derive]**.

```

#[derive(Debug, Clone)]
struct MyStruct {
    field: i32,
}

```

I trait possono essere utilizzati come vincoli nei tipi generici: Qui T deve implementare il trait Display.

```
fn print_value<T: std::fmt::Display>(value: T) {
    println!("{}: {}", value);
}
```

## Trait Stack<T>

Questo esempio definisce un **trait** generico Stack<T>, l'interfaccia che deve implementare una pila e implementa il trait per una struttura concreta SLStack<T>, una pila basata su nodi collegati.

I metodi includono:  
new: Crea una nuova pila.  
is\_empty: Verifica se la pila è vuota.  
push: Inserisce un elemento.  
pop: Rimuove e restituisce l'elemento in cima.

```
struct Slot<T> {
    data: Box<T>,
    prev: Option<Box<Slot<T>>>
}

trait Stack<T> {
    fn new() -> Self;
    fn is_empty(&self) -> bool;
    fn push(&mut self, data: Box<T>);
    fn pop(&mut self) -> Option<Box<T>>;
}

impl<T> Stack<T> for SLStack<T> {
    fn new() -> SLStack<T> {
        SLStack{ top: None }
    }
    ...
    fn is_empty(&self) -> bool {
        match self.top {
            None      => true,
            Some(..)  => false,
        }
    }
}
```

**Slot<T>**: Rappresenta un nodo della pila, con un valore data e un puntatore al nodo precedente (prev).

**SLStack<T>**: Rappresenta la pila con un riferimento al nodo in cima (top). Implementa il trait Stack<T>:

## Bounded Polymorphism

In Rust, le **funzioni generiche** supportano il **bounded polymorphism**, consentendo di vincolare i tipi generici a uno o più **trait**. Ciò garantisce che il tipo generico possa essere utilizzato solo attraverso i metodi e i comportamenti definiti dai trait vincolanti.

```
fn print_value<T: std::fmt::Display>(value: T) {
    println!("{}: {}", value);
}
```

Qui T deve implementare il trait Display per poter utilizzare println!.  
Rust verifica che il codice di una funzione generica sia valido per qualsiasi tipo che soddisfa i vincoli specificati dai trait.

Durante la compilazione, Rust genera **una copia separata del codice** per ogni istanza di tipo utilizzata. Il compilatore genera codice altamente performante e specifico per ogni tipo. Ma il tempo di compilazione è più lungo.

## System Traits in Rust

I **system traits** sono utilizzati per definire comportamenti o proprietà standard per i tipi e aiutano il compilatore a ottimizzare e gestire correttamente il codice.

**Clone**: Permette di creare una copia profonda (*deep copy*) di un valore con il metodo clone().

**Copy:** Permette di duplicare un valore copiando direttamente i bit dalla memoria dello stack. È un **marker trait** e non può essere implementato insieme a Clone se clone() richiede operazioni complesse.

**Debug:** Fornisce una conversione predefinita di un valore in testo, utile per il debugging. Abilitato automaticamente con #[derive(Debug)].

**Display:** Permette una conversione personalizzabile in testo tramite il metodo fmt(). Usato per formattazioni leggibili (ad esempio, con println!).

**Deref:** Implementato dagli **smart pointer**, consente di accedere ai dati sottostanti.

**Drop:** Gestisce il rilascio delle risorse quando un valore esce dallo scope.

**Send:** Indica che un tipo può essere trasferito in modo sicuro tra thread.

**Sync:** Indica che un tipo può essere condiviso tra più thread in modo sicuro. Entrambi sono **marker traits**, quindi non contengono metodi.

## Smart Pointers in Rust

Gli **Smart Pointers** sono **riferimenti** che si comportano come puntatori tradizionali, ma con l'aggiunta di **metadati e funzionalità avanzate**.

Alcuni esempi sono: **String** (che incapsula una slice &str) e **Vec<T>** (un array dinamico).

Gli **Smart Pointers** sono solitamente implementati come **struct**, con due caratteristiche principali: Implementano il trait **Deref**, che permette di accedere al valore sottostante come un riferimento normale (\*). Implementano il trait **Drop**, che permette la gestione automatica della memoria quando escono dallo scope.

Un'altra caratteristica è la **Deref Coercion**, una funzionalità che consente **conversioni automatiche** da Smart Pointer a riferimenti normali per facilitare l'uso.

```
let s = String::from("hello");
let len = s.len(); // Deref coercion: `s` viene trattato come `&str`.
```

## Box<T> in Rust

Il tipo **Box<T>** è uno **smart pointer** che permette di allocare dati di tipo T nell'heap, mantenendo un puntatore nello stack per accedervi. Accedere a un valore tramite Box<T> ha lo stesso costo di un puntatore normale. Box<T> implementa il trait Deref, permettendo di accedere al valore sottostante con \* o in modo implicito. Si usa Box<T> per tipi ricorsivi e per grandi quantità di dati.

## Rc<T>: Reference Counting in Rust

Il reference counting **Rc<T>** è utilizzato per abilitare il **conteggio delle referenze** ad una risorsa, **consentendo** a più parti del programma di **condividere il possesso** di una risorsa **senza copiarla**.

**Rc::clone():** Crea una nuova referenza al dato esistente, incrementando il conteggio delle referenze, senza eseguire una copia profonda.

**Rc::strong\_count:** Restituisce il numero attuale di referenze forti alla risorsa condivisa. Quando il conteggio raggiunge 0, la risorsa viene automaticamente deallocated.

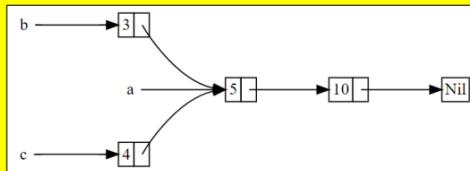
```

use crate::List::{Cons, Nil};
use std::rc::Rc;

enum List {
    Cons(i32, Rc<List>),
    Nil,
}

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}

```



## RefCell<T>: interior mutability

**RefCell<T>** consente di **modificare i dati** anche se il valore è immutabile (grazie al pattern di mutabilità interna). Fornisce metodi:

**borrow()**: ottiene una referenza **immutabile** al valore (ritorna un puntatore **Ref<T>**).

**borrow\_mut()**: ottiene una **referenza mutabile** al valore (ritorna un puntatore **RefMut<T>**).

**RefCell<T>** tiene traccia dinamicamente (a runtime) del **numero di referenze attive**, sia immutabili che mutabili. Garantisce che **non ci siano violazioni** delle regole di *borrowing*: Più letture immutabili sono consentite contemporaneamente. Solo una scrittura mutabile è consentita e non può coesistere con letture. Se queste regole vengono violate, il programma genera un *panic* a runtime.

**RefCell<T>** viene spesso combinato con **Rc<T>** per consentire l'accesso mutabile condiviso in programmi single-threaded.

```

enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}
...

fn main() {
    let value = Rc::new(RefCell::new(5));
    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));
    *value.borrow_mut() += 10;
    println!(...);
}

```

Interior mutability:

La variabile `value` è un `RefCell` che contiene il valore 5, mentre `a`, `b` e `c` condividono parte della struttura tramite `Rc` e possono accedere/mutare il valore tramite `RefCell`. La mutazione avviene con `borrow_mut()`.

## Comparing smart pointers

Type	Sharable?	Mutable?	Thread Safe?
&	yes *	no	no
&mut	no *	yes	no
<b>Box</b>	no	yes	no
<b>Rc</b>	yes	no	no
<b>Arc</b>	yes	no	yes
<b>RefCell</b>	yes **	yes	no
<b>Mutex</b>	yes, in Arc	yes	yes

\* but doesn't own contents, so lifetime restrictions.

\*\* while there is no mutable borrow

## Closure

Le closure in Rust sono funzioni anonime che possono catturare variabili dall'ambiente esterno. Possono farlo in tre modi, a seconda di come accedono alle variabili:

**Ownership:** Catturano la variabile prendendone possesso (move).

**Mutable Borrowing:** Catturano la variabile come referenza mutabile.

**Immutable Borrowing:** Catturano la variabile come referenza immutabile.

Questa cattura determina il *trait* implementato dalla closure:

**FnOnce:** Consuma la variabile catturata (ownership).

**FnMut:** Modifica la variabile catturata.

**Fn:** Usa la variabile senza modificarla.

**Nota:** Specificare move prima dei || forza il comportamento FnOnce.

```
fn main() {
    let x = 5;
    let greater_than_x = || y | y > x; // Parameters within || 
    println!("{}" , greater_than_x(3)); // prints "false"
}
```

La closure cattura x con un *immutable borrow* perché non modifica né consuma il valore.

```
let vector = vec![1, 2, 3, 4, 5]; // stream-like
vector.iter()
    .map(|x| x + 1)
    .filter(|x| x % 2 == 0)
    .for_each(|x| println!("{}" , x));
```

## C++: Problema con Race Conditions

```
// C++ code
int main() {
    int counter = 0;
    const auto task = [&] {
        for (int i = 0; i < 100000; ++i) {
            counter++;
        }
    };
    thread thread1(task);
    thread thread2(task);
    thread1.join();
    thread2.join();
    cout << counter << endl;
    return 0;
}
```

```
// Rust: does not compile
fn main() {
    let mut counter = 0;
    let task = || { // closure
        for _ in 0..100000 {
            counter += 1;
        }
    };
    let thread1 = thread::spawn(task);
    let thread2 = thread::spawn(task);
    thread1.join().unwrap();
    thread2.join().unwrap();
    println!("{}" , counter);
}
```

```
error[E0373]: closure may outlive the current function, but it borrows
`counter`, which is owned by the current function
--> src\main.rs:57:16
let task = || {
    ^^^ may outlive borrowed value `counter`
    for _ in 0..100000 {
        counter += 1;
    }
    ----- `counter` is borrowed here
    help: to force the closure to take ownership of `counter` (and any other
    referenced variables), use the `move` keyword
    let task = move || {      // would it work?
    +++++
```

Due thread incrementano simultaneamente la variabile counter. Poiché l'accesso a counter non è sincronizzato, può verificarsi una *race condition*.

Questo codice **non compila**. Rust rileva che la closure task potrebbe accedere in modo non sicuro alla variabile counter.

**Problema:** La variabile counter è mutabile e viene condivisa tra i thread senza alcuna protezione. Rust non permette a più thread di accedere contemporaneamente a una variabile mutabile senza sincronizzazione. Il codice non compila perché Rust rileva potenziali violazioni delle regole di mutabilità e ownership.

```
// Rust code with Arc<T>: Doesn't compile
fn main() {
    let mut counter = Arc::new(0);
    let c1 = Arc::clone(&counter);
    let c2 = Arc::clone(&counter);
    let thread1 = thread::spawn(move || {
        for _ in 0..100000 {
            *c1 += 1; // Increment c1
        }
    });
    let thread2 = thread::spawn(move || {
        for _ in 0..100000 {
            *c2 += 1; // Increment c2
        }
    });
    thread1.join().unwrap();
    thread2.join().unwrap();
    println!("{}", counter);
}
```

Arc consente il conteggio delle referenze per condividere dati immutabili tra thread, ma non permette modifiche dirette ai dati contenuti.

**Errore:** L'operazione `*c1 += 1` fallisce perché `Arc<T>` non implementa il trait `DerefMut` necessario per mutare i dati.

**Soluzione:** Arc gestisce il conteggio delle referenze per condividere il Mutex tra i thread. Mutex garantisce che solo un thread possa accedere o modificare i dati alla volta. Ogni thread chiama `lock()` sul Mutex, ottenendo accesso esclusivo alla variabile condivisa. Dopo la modifica, il lock viene rilasciato automaticamente.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0)); // `Mutex` protegge i dati
    let mut handles = vec![];

    for _ in 0..2 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..100000 {
                let mut num = counter.lock().unwrap(); // Lock per accesso esclusivo
                *num += 1;
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("{}", *counter.lock().unwrap());
}
```

## Send

Un tipo implementa il trait **Send** se **può essere trasferito** in modo sicuro tra thread. Se un tipo non implementa Send, il compilatore genera un **errore** quando si tenta di trasferire la sua **ownership a un altro thread**.

## Sync

Un tipo implementa il trait **Sync** se può essere **referenziato (immutably borrowed)** da più **thread contemporaneamente**.

Un tipo T implementa Sync se e solo se &T implementa Send. Questo garantisce che un riferimento immutabile a T possa essere condiviso tra thread in modo sicuro.

**Rc<T>:** Non è né Send né Sync, poiché non è thread-safe. È progettato per l'uso in programmi single-threaded.

**Arc<T>**: È la versione thread-safe di Rc<T>. Implementa sia Send che Sync, rendendolo adatto per dati condivisi tra thread.

**Mutex<T>**: Fornisce accesso esclusivo (*mutual exclusion*) a un valore protetto tramite un lock. È sia Send che Sync:

## Condivisione Mutabile

La mutabilità condivisa è inevitabile in molte applicazioni, ad esempio una **lista doppiamente collegata**, in cui ogni nodo contiene: Un puntatore al nodo precedente (prev) e Un puntatore al nodo successivo (next).



```
struct Node {  
    prev: Option<Box<Node>>,  
    next: Option<Box<Node>>  
}
```

Limitazione: Box non consente condivisione mutabile diretta. Quando il modello di proprietà standard di Rust non è sufficiente (es. condivisione mutabile necessaria), si possono usare i **puntatori raw**. I puntatori raw non vengono controllati dal compilatore: il programmatore è responsabile della sicurezza.

```
struct Node {  
    prev: Option<Box<Node>>,  
    next: *mut Node  
}
```

*Raw pointer*

Operazioni su puntatori raw devono essere racchiuse in blocchi **unsafe** per segnalare che il programmatore sa cosa sta facendo.

```
let a = 3;  
  
unsafe {  
    let b = &a as *const i32 as *mut i32;  
    *b = 4;  
}  
  
println!("a = {}", a);
```

*I know what I'm doing*

*Print "a = 4"*

La variabile a viene modificata nonostante sia immutabile nel contesto standard.

## Foreign Function Interface (FFI)

L'FFI è un'interfaccia che permette a Rust di chiamare funzioni scritte in altri linguaggi, come C. Per definizione, tutte le funzioni esterne (*foreign functions*) sono considerate **unsafe**, poiché Rust non può garantire la sicurezza della memoria quando interagisce con codice esterno. Per dichiarare una funzione esterna in Rust, si utilizza il blocco `extern`.

Nel seguente codice, viene importata la funzione `write()` della libreria `posix`.

```

extern {
    fn write(fd: i32, data: *const u8, len: u32) -> i32;
}

fn main() {
    let msg = "Hello, world!\n";
    unsafe {
        write(1, &msg[0], msg.len());
    }
}

```

In Rust, l'uso di blocchi unsafe abilita operazioni che il compilatore non può garantire siano sicure. Tra queste operazioni troviamo la dereferenziazione di puntatori raw, che possono essere creati in Rust sicuro ma non dereferenziati senza un blocco unsafe poiché non si può garantire che puntino a memoria valida.

Le funzioni e i metodi marcati come unsafe forniscono accesso diretto, ad esempio, all'allocatore di Rust, che è intrinsecamente insicuro perché interagisce direttamente con il sistema operativo. Inoltre, i blocchi unsafe consentono di accedere o modificare variabili statiche mutabili, implementare trait dichiarati unsafe e accedere ai campi di strutture union, che sono progettate per rappresentare più tipi nello stesso spazio di memoria. Tuttavia, è importante sottolineare che unsafe non disattiva il borrow checker, il quale continua a verificare le regole di borrowing per quanto possibile, limitando il rischio di errori.

## RustBelt

Il progetto RustBelt si propone di formalizzare le regole di tipizzazione di Rust e di dimostrarne la correttezza per garantire l'assenza di comportamenti non definiti. La dimostrazione è suddivisa in tre fasi principali.

La prima consiste nel verificare che le regole di tipizzazione siano semanticamente corrette, ossia che la conclusione derivata da queste regole sia coerente con le loro premesse a livello semantico.

La seconda fase dimostra che, se un programma è semanticamente ben tipizzato, la sua esecuzione non porterà a problemi come comportamenti non definiti.

La terza fase si concentra sulle librerie che utilizzano codice unsafe, verificando che siano semanticamente sicure quando utilizzate tramite le loro interfacce pubbliche. Questo approccio fornisce una base matematica rigorosa che sostiene la sicurezza e l'affidabilità di Rust, inclusa la corretta gestione delle operazioni unsafe.

## AP-2024-26-Python

**Java** è un linguaggio **statically typed**, dove il tipo di una variabile è fissato al momento della dichiarazione e non può cambiare.

**Python**, invece, è un linguaggio **dynamically typed**, ovvero una variabile può riferirsi a oggetti di **tipi diversi** durante l'esecuzione del programma. In Python, le variabili non hanno un tipo predefinito: esistono solo dopo essere state assegnate a un valore, e il **tipo è associato al valore** piuttosto che alla variabile stessa.

Sebbene sia **strongly typed** (i tipi non cambiano in modo inatteso e non è possibile combinare tipi incompatibili senza un'esplicita conversione), gli errori di tipo vengono rilevati soltanto a runtime.

Il codice Python segue lo stile **PEP 8**, che promuove leggibilità e semplicità, fornendo una sintassi diretta.

```

class Hello { // Java
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
-----
print "Hello, world!\n" # Python

```

## Comandi Utili del Python Interpreter

**help()**: Comando che senza argomenti, avvia l'utility di aiuto interattiva di Python. Con un argomento, restituisce la documentazione relativa all'oggetto passato. (help(str), help(print)).

**type(arg)**: Restituisce il tipo dell'argomento passato come oggetto Python, ad esempio type(42) restituisce <class 'int'>.

**Underscore (\_) nel prompt interattivo**: Rappresenta il valore dell'ultima espressione valutata nel terminale interattivo.

**Autocompletamento**: Python supporta l'autocompletamento degli attributi e dei metodi degli oggetti. Digitando 1. seguito da <tab><tab>, ad esempio, vengono mostrati i metodi disponibili per l'oggetto intero.

**dir()**: Restituisce una lista ordinata di stringhe che rappresentano tutti i nomi definiti in un modulo, una classe o un oggetto. Utile per esplorare attributi e metodi di un oggetto o un namespace.

Un **modulo** è un file Python che contiene definizioni e funzioni. I moduli creano un **namespace** separato, utile per organizzare il codice. I moduli possono essere raggruppati gerarchicamente in **packages**.

```
# File fibo.py - Fibonacci numbers module
def fib(n):      # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):    # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

L'importazione completa di un modulo avviene tramite l'istruzione **import nameModule**. L'importazione selettiva di alcune funzioni da un modulo avviene tramite l'istruzione **from nameModule import funct1, ..., functN**. Il nome del modulo è contenuto nell'attributo speciale **\_\_name\_\_**, mentre il nome del modulo di un oggetto è contenuto nel suo attributo **\_\_module\_\_**.

Un modulo può essere eseguito su terminale utilizzando il comando **python3 nameModule args** .

Per distinguere se un file Python viene eseguito direttamente come script o è stato importato come modulo, si utilizza la variabile speciale **\_\_name\_\_**. Quando il file è eseguito come script, il valore di **\_\_name\_\_** è "**\_\_main\_\_**", mentre se il file è importato, il valore sarà il nome del modulo.

Inserendo un controllo come `if __name__ == "__main__":`, è possibile definire una sezione di codice che verrà eseguita solo quando il file è eseguito direttamente, permettendo di separare il comportamento di esecuzione diretta da quello di importazione.

```
python

if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

**Funzioni**: In Python, le funzioni si definiscono con la parola chiave **def**, seguita dal nome, dalle parentesi tonde con eventuali parametri, e dai due punti (:). Non sono richieste dichiarazioni di tipo per parametri o valori restituiti. Il corpo della funzione è indentato.

**Blocchi di codice:** I blocchi di codice (come funzioni, cicli e condizioni) sono definiti dall'indentazione, anziché dalle parentesi graffe {}. Ogni riga successiva al : deve essere indentata.

**Stringhe:** Le stringhe possono essere definite usando sia doppie virgolette ("") che singole virgolette (''). Entrambe le sintassi sono equivalenti.

**Commenti:** I commenti iniziano con il simbolo # e tutto il testo a destra di # viene ignorato dall'interprete.

Python elimina la verbosità presente in Java (esempio: niente public, static, void se non necessario).

## Tipi di Dati Primitivi

**Interi:** Gli interi in Python non hanno limiti di dimensione.

**Numeri in virgola mobile:** Sono rappresentati con precisione a 64 bit.

**Operazioni aritmetiche standard:** +, -, \*, /, % (modulo), la divisione intera usa //.

**Operatori Logici:** and (&&), or (||), not (!).

**Stringhe:** Possono essere racchiuse in:singoli "", doppi "" e tripli apici """..."""

**Concatenazione di stringhe:** Usa l'operatore + per unire più stringhe.

**Commenti:** Iniziano con #:

**Docstrings:** Stringhe utilizzate per documentare una funzione o un modulo. Racchiuse in tripli apici """").

**Assegnazione:** L'assegnazione **non crea una copia** del valore. Una variabile diventa un **nome** che fa riferimento a un **oggetto** esistente.

Una variabile viene **creata** la prima volta che appare a sinistra di un'espressione di assegnazione:

In CPython, la Garbage Collection gestisce automaticamente la memoria utilizzando due meccanismi principali:

**Reference Counting:** Ogni oggetto tiene traccia del numero di riferimenti ad esso.

Quando il conteggio scende a zero (cioè nessuna variabile punta più a quell'oggetto), la memoria dell'oggetto viene liberata immediatamente.

**Mark & Sweep:** Questo meccanismo viene usato per rilevare e risolvere i cicli di riferimento (situazioni in cui due o più oggetti si riferiscono tra loro, impedendo al conteggio di riferimenti di scendere a zero). Durante questo processo, vengono identificati gli oggetti non raggiungibili e la loro memoria viene liberata.

Python permette l'assegnazione multipla in un'unica riga: x, y = 2, 3

## Sequenze in Python

**Tuples:** Strutture dati immutabili che non possono essere modificate dopo la creazione. Mantengono l'ordine di inserimento e possono contenere tipi di dati diversi.

*Esempio:* (2, 3.14, False)

**Lists:** Strutture dati mutabili che consentono di aggiungere, rimuovere o modificare gli elementi. Mantengono l'ordine degli elementi e possono contenere tipi di dati diversi.

*Esempio:* [2, 3.14, False]

**Strings:** Sequenze immutabili. I caratteri seguono un ordine specifico, ma non possono essere modificati.

*Esempio:* "stringa"

**Conversione:** Gli operatori list() e tuple() sono utilizzati per convertire una tupla in lista e viceversa.

```
list((1, 2, 3)) # Converte una tupla in una lista
```

```
tuple([1, 2, 3]) # Converte una lista in una tupla
```

**Accesso agli elementi:** Il primo elemento di una sequenza ha indice 0. Gli indici negativi partono dalla fine della sequenza.

```
seq = [1, 2, 3]
print(seq[-1]) # Output: 3
print(seq[-2]) # Output: 2
```

**Slicing:** Operazione che permette di estrarre una **sottosequenza** (copia) da una sequenza originale. La sintassi è `nameSeq[start:end:step]`, dove **start** rappresenta l'indice iniziale (incluso), **end** rappresenta l'indice finale (escluso), e **step**: quanti passi compiere (opzionale, predefinito 1).

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[1:4]           #      ('abc', 4.56, (2,3))
>>> t[1:-1]          # negative indices          ('abc', 4.56, (2,3))
>>> t[1:-1:2]        # optional argument: step      ('abc', (2,3))
>>> t[:2]            # no first index: from beginning (23, 'abc')
>>> t[2:]            # no second index: to end       (4.56, (2,3), 'def')
>>> t[:]             # no indexes: creates a copy    (23, 'abc', 4.56, (2,3), 'def')
```

**Concatenazione (+):** Combina due sequenze per creare una **nuova sequenza**.

Funziona con tuple, liste e stringhe.

```
print((1, 2)+(3, 4)) # Output: (1, 2, 3, 4)
```

**Membership (in e not in):** Verifica se un elemento è presente in una sequenza.

```
t = [1, 2, 4, 5]
```

```
print(4 in t) # Output: True
```

Le **liste** sono mutabili, cioè possono essere modificate **in place** senza creare una nuova lista.

```
li = ['abc', 23, 4.34, 23]
```

```
li[1] = 45 # Modifica l'elemento all'indice 1
```

### Metodi Principali delle Liste:

**append(el):** Aggiunge un elemento alla fine della lista.

**insert(index, element):** Inserisce un elemento in una posizione specificata.

**extend(iterable):** Aggiunge gli elementi di un altro iterabile (es. lista, tupla) alla lista esistente. Funziona come l'operatore +, ma modifica la lista in place.

**index(element):** Restituisce l'indice della prima occorrenza di un elemento.

**count(element):** Restituisce il numero di occorrenze di un elemento.

**remove(element):** Rimuove la prima occorrenza di un elemento.

**reverse():** Inverte l'ordine degli elementi nella lista.

**sort():** Ordina gli elementi della lista (solo se omogenei).

### List Comprehensions

Le list comprehensions sono un modo compatto e leggibile per creare liste basate su trasformazioni degli elementi di liste, tuple o stringhe.

**Sintassi generale:** [expression for name in list]

- **expression:** Operazione o calcolo applicato a ogni elemento.
- **name:** Variabile che rappresenta l'elemento corrente.
- **list:** La sequenza iterabile da cui si creano i nuovi elementi.

```
li = [3, 6, 2, 7]
```

```
result = [elem * 2 for elem in li]
```

```
print(result) # Output: [6, 12, 4, 14]
```

Ogni elemento di li è assegnato alla variabile elem. L'espressione `elem * 2` viene calcolata per ciascun elemento. I risultati vengono raccolti in una nuova lista.

Quando gli elementi della lista sono tuple (o altre collezioni), possiamo **destrutturarle** direttamente nella list comprehension.

```
li = [('a', 1), ('b', 2), ('c', 7)]
```

```
result = [n * 3 for (x, n) in li]
```

```
print(result) # Output: [3, 6, 21]
```

Python permette la destrutturazione delle variabili con liste o tuple.

```
[x, y] = [4, 5] # x = 4, y = 5  
(x, y) = "23" # x = '2', y = '3'
```

## Filtered List Comprehension

Le **filtered list comprehensions** aggiungono una **condizione di filtro**, includendo nella lista finale solo gli elementi che soddisfano una determinata condizione.

### [expression for name in list if filter]

**filter:** Una condizione booleana che deve essere vera affinché l'elemento venga incluso.

Ogni elemento di list viene verificato con il **filtro**. L'**expression** viene applicata solo agli elementi che soddisfano la condizione di filtro.

```
li = [3, 6, 2, 7, 1, 9]  
result = [elem * 2 for elem in li if elem > 4]  
print(result) # Output: [12, 14, 18]
```

## Nested List Comprehensions

Le **list comprehensions annidate** (nested) permettono di combinare più livelli di elaborazione. Poiché una list comprehension produce una lista come output, questa può essere utilizzata come input per un'altra list comprehension.

### [outer\_expression for outer\_name in [inner\_expression for inner\_name in list]]

```
li = [3, 2, 4, 1]  
result = [elem * 2 for elem in [item + 1 for item in li]]  
print(result) # Output: [8, 6, 10, 4]
```

## Sets

- Empty set: `set()`
- Indexing not supported
- Mixed types

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',  
'banana'}  
>>> print(basket)          # show that duplicates have been removed  
{'orange', 'banana', 'pear', 'apple'}  
>>> 'orange' in basket    # fast membership testing  
True  
>>> 'crabgrass' in basket  
False  
>>> # Demonstrate set operations on unique letters from two words  
>>> a = set('abracadabra')  
>>> b = set('alacazam')  
>>> a                      # unique letters in a  
{'a', 'r', 'b', 'c', 'd'}  
>>> a - b                  # letters in a but not in b  
{'r', 'd', 'b'}  
>>> a | b                  # letters in a or b or both  
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}  
>>> a & b                  # letters in both a and b  
{'a', 'c'}  
>>> a ^ b                  # letters in a or b but not both  
{'r', 'd', 'b', 'm', 'z', 'l'}
```

## Dizionari in Python

I dizionari memorizzano coppie **chiave-valore** in cui:

**Chiavi:** Devono essere **immutabili e hashable** (es. int, str, tuple) e devono essere **uniche**.

**Valori:** Possono essere di qualsiasi tipo, inclusi tipi mutabili.

```
d = {'user': 'bozo', 'pswd': 1234}
```

**Accesso ai valori tramite la chiave:** `nameDict['nameKey']` (Es. `print(d['user'])`) # Output: 'bozo'). Se una chiave non esiste, genera un **KeyError**.

**Modifica di una chiave esistente:** `d['user'] = 'clown'`

```
print(d) # Output: {'user': 'clown', 'pswd': 1234}
```

**Aggiunta di una nuova chiave:** `d['id'] = 45`

```
print(d) # Output: {'user': 'clown', 'pswd': 1234, 'id': 45}
```

**del:** Rimuove una chiave specifica. del `d['user']`

```
(print(d) # Output: {'pswd': 1234, 'id': 45})
```

**clear():** Rimuove tutti gli elementi del dizionario: `d.clear()`

```
(print(d) # Output: {})
```

**keys():** Restituisce tutte le chiavi: `print(list(d.keys()))` # Output: ['user', 'pswd', 'id']

**values():** Restituisce tutti i valori: `print(list(d.values()))` # Output: ['clown', 1234, 45]

**items()**: Restituisce coppie chiave-valore sotto forma di tuple:

```
print(list(d.items())) # Output: [('user', 'clown'), ('pswd', 1234), ('id', 45)]
```

**Iterazione su chiavi e valori:**

**for key, value in d.items():**

```
    print(f'{key}: {value}')
```

I dizionari **non sono ordinati** (fino a Python 3.6, da 3.7 in poi sono inseriti in ordine di inserimento). Sono implementati con **hashing**, il che li rende molto efficienti per operazioni di ricerca.

## Espressioni Booleane in Python

Python utilizza le costanti **True** e **False** per rappresentare valori booleani.

Il valore **False** è equivalente a 0, None e ai contenitori vuoti [], {}, (), "".

Tutti gli altri valori (numeri non zero, contenitori non vuoti, oggetti) sono considerati equivalenti a **True**.

Gli operatori per confrontare valori in Python sono:

**==**: Verifica se due valori sono uguali.

**!=**: Verifica se due valori sono diversi.

**<, <=, >, >=**: Confrontano due grandezze.

**is**: Verifica se due variabili puntano allo stesso oggetto in memoria (simile a == in Java per il riferimento).

```
x = [1, 2]
```

```
y = [1, 2]
```

```
print(x == y) # True (valori uguali)
```

```
print(x is y) # False (oggetti diversi)
```

Gli operatori logici forniti da Python sono:

**and**: Restituisce True se entrambi gli operandi sono veri.

**or**: Restituisce True se almeno uno degli operandi è vero.

**not**: Inverte il valore logico.

Python supporta un'operazione ternaria per espressioni condizionali:

```
x = <true_value> if <condition> else <false_value>
```

Se <condition> è vera, restituisce <true\_value>, altrimenti <false\_value>.

```
age = 18
```

```
status = "Adult" if age >= 18 else "Minor"
```

```
print(status) # Output: Adult
```

## Control statements: conditional

```
if x == 3:
    print("X equals 3.")
elif x == 2:
    print("X equals 2.")
else:
    print("X equals something else.")
print ("This is outside the 'if'.")
```

## while Loops

```
>>> x = 3
>>> while x < 5:
        print (x, "still in the loop")
        x = x + 1
3 still in the loop
4 still in the loop
>>> x = 6
>>> while x < 5:
        print (x, "still in the loop")
>>>
>>>
```

**Break**: Operatore che fa fuoriuscire dal loop.

**Continue**: Operatore che fa saltare all'iterazione successiva.

## Assert in Python

L'istruzione **assert** è utilizzata per verificare che una condizione sia vera durante l'esecuzione di un programma. Se la condizione è falsa, viene generata un'eccezione di tipo **AssertionError**.

```
number_of_players = 3
assert number_of_players < 5 # Nessun errore
assert number_of_players > 5 # Genera AssertionError
```

## For Loops in Python

I cicli for in Python iterano su una **collezione** o un **iterable**. È simile a un ciclo "for-each".

### Sintassi

```
for <item> in <collection>:
    <statements>
```

```
# Lista
for item in [1, 2, 3]:
    print(item)
```

```
# Stringa
for char in "Hello":
    print(char)
```

```
# Range
for i in range(5):
    print(i) # Output: 0, 1, 2, 3, 4
```

## For annidati e destrutturazione

Gli elementi di una collezione possono essere destrutturati.

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
for x, y in pairs:
    print(x, y)
```

## Funzione range()

Genera un iteratore di numeri:

**range(stop)**: Genera numeri da 0 a stop-1.

**range(start, stop[, step])**: Genera numeri con inizio, fine, e passo specificati.

```
for i in range(1, 10, 2):
    print(i) # Output: 1, 3, 5, 7, 9
```

### Uso corretto di enumerate

Evita l'uso di `range(len())` per iterare su una sequenza con indice e valore. Usa invece `enumerate()`.

```
for i in range(len(mylist)):
    print(i, mylist[i])
=====→
for i, item in enumerate(mylist):
    print(i, item)
```

## AP-27: Python: Functions, Decorators, Namespaces

### Funzioni in Python - Concetti Essenziali

```

def echo(arg): return arg

type(echo)      # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)     # <function echo at 0x1003c2bf8>
foo = echo
hex(id(foo))   # '0x1003c2bf8'
print(foo)      # <function echo at 0x1003c2bf8>
isinstance(echo, object)  # => True

```

In Python, le funzioni sono trattate come **first-class objects**, ovvero possono essere assegnate a variabili, passate come argomenti o restituite da altre funzioni.

```

def greet(name):
    return f"Hello, {name}!"

```

```

say_hello = greet # Assegno la funzione a una variabile
print(say_hello("Alice")) # Output: Hello, Alice!

```

Tutte le funzioni **restituiscono un valore**. Se non viene specificata una dichiarazione `return`, viene restituito **None**.

```

def add(a, b):
    return a + b

```

```

def no_return():
    pass

```

```

print(add(3, 4)) # Output: 7
print(no_return()) # Output: None

```

Le variabili definite all'interno della funzione sono locali a quella funzione.

```

def my_function():
    x = 10 # Variabile locale
    print(x)

my_function()
# print(x) # Errore: x non è definita nello scope globale

```

In Python, i parametri sono passati tramite **riferimento a oggetti**. Ciò significa che se un oggetto mutabile viene modificato all'interno di una funzione, il cambiamento sarà visibile anche al di fuori.

```

def modify_list(lst):
    lst.append(42)

my_list = [1, 2, 3]
modify_list(my_list)
print(my_list) # Output: [1, 2, 3, 42]

```

Le funzioni in Python possono accettare diversi **tipi di parametri**: posizionali, con nome (keyword arguments) e con valori predefiniti. Inoltre, supportano un numero arbitrario di argomenti.

**Posizionali:** Gli argomenti sono passati nell'ordine in cui sono definiti nella funzione.

**Keyword:** Gli argomenti sono passati specificando il nome del parametro.

```

def sum(n, m):
    """Aggiunge due valori."""
    return n + m

```

```

print(sum(3, 4))      # Output: 7 (parametri posizionali)
print(sum(m='lo', n='hel')) # Output: 'hello' (parametri keyword)

```

**Parametri con valori di default:** Se il valore non viene fornito durante la chiamata, viene usato quello predefinito.

```
def sum(n, m=5):
    """Aggiunge due valori o incrementa di 5."""
    return n + m
```

```
print(sum(3)) # Output: 8 (usa il valore predefinito di m)
```

**Argomenti posizionali variabili (\*args):** Raccolgono un numero arbitrario di argomenti in una **tupla**. Utili quando non si conosce in anticipo il numero di argomenti.

```
def print_args(*args):
    print(type(items)) # <class 'tuple'>
    return items

print(print_args(1, "hello", 4.5))
# Output: (1, 'hello', 4.5)
```

**Argomenti con nome variabili (\*\*kwargs):** Raccolgono un numero arbitrario di argomenti con nome in un **dizionario**. Utili per gestire parametri opzionali con nome.

```
def print_kwargs(**items):
    print(type(items)) # <class 'dict'>
    return items

print(print_kwargs(a=2, b=3, c=3))
# Output: {'a': 2, 'b': 3, 'c': 3}
```

## Documentazione delle Funzioni in Python

In Python, la documentazione delle funzioni si realizza utilizzando una **docstring**: un commento racchiuso tra tripli apici (""""") che si trova immediatamente sotto la definizione della funzione. Questa stringa viene associata all'attributo speciale `__doc__` della funzione, rendendola accessibile per scopi di documentazione.

```
def my_function():
    """
    Summary line: do nothing, but document it.
    Description: No, really, it doesn't do anything.
    """

    pass
```

La docstring può essere visualizzata utilizzando l'attributo `__doc__` della funzione:

```
print(my_function.__doc__)
# Summary line: Do nothing, but document it.
#
#     Description: No, really, it doesn't do anything.
#
# try also 'help(my_function)'
```

## Higher order functions

Le **higher-order functions** sono funzioni che possono prendere altre funzioni come argomenti e restituire altre funzioni come risultato.

**map(function, iterable):** Applica una funzione a tutti gli elementi di un iterabile e restituisce un oggetto mappato.

```
numbers = [1, 2, 3, 4]
doubled = map(lambda x: x * 2, numbers)
print(list(doubled)) # Output: [2, 4, 6, 8]
```

**filter(function, iterable):** Filtra gli elementi di un iterabile che soddisfano una determinata condizione (funzione che restituisce True o False).

```
numbers = [1, 2, 3, 4]
evens = filter(lambda x: x % 2 == 0, numbers)
```

```
print(list(evens)) # Output: [2, 4]
```

## Uso di iteratori

Python utilizza ampiamente iteratori per implementare il comportamento delle funzioni come map e filter, consentendo una valutazione "lazy".

### Lambdas

Le **funzioni lambda** sono funzioni anonime definite in una sola riga. La loro sintassi è:

**lambda arguments: expression**

```
square = lambda x: x ** 2
```

```
print(square(5)) # Output: 25
```

## Functional Programming Modules in Python

Python offre diversi moduli utili per la programmazione funzionale, in particolare **functools** e **itertools**, che forniscono strumenti potenti per lavorare con funzioni e iteratori. Il modulo **functools** include funzioni higher-order e operazioni su oggetti chiamabili.

**reduce(function, iterable[, initializer])**: Applica una funzione cumulativa agli elementi di un iterabile, riducendolo a un singolo valore.

```
from functools import reduce
```

```
result = reduce(lambda x, y: x + y, [1, 2, 3, 4])
```

```
print(result) # Output: 10
```

Il modulo **itertools** fornisce funzioni per creare iteratori efficienti, ispirate a costrutti di linguaggi funzionali come APL, Haskell e SML. Ecco alcune funzioni utili:

**count(start)**: Genera numeri a partire da un valore iniziale, incrementandoli di 1.

```
from itertools import count
```

```
for x in count(10):
```

```
    print(x) # Output: 10, 11, 12, ... (interrompi manualmente)
```

**cycle(iterable)**: Ripete all'infinito gli elementi di una lista.

```
from itertools import cycle
```

```
for x in cycle('ABCD'):
```

```
    print(x) # Output: A, B, C, D, A, B, ... (interrompi manualmente)
```

**repeat(element, times)**: Ripete un elemento un numero specificato di volte.

```
from itertools import repeat
```

```
print(list(repeat(10, 3))) # Output: [10, 10, 10]
```

**takewhile(predicate, iterable)**: Restituisce gli elementi di un iterabile fino a quando il predicato restituisce True.

```
from itertools import takewhile
```

```
result = takewhile(lambda x: x < 5, [1, 4, 6, 4, 1])
```

```
print(list(result)) # Output: [1, 4]
```

**accumulate(iterable)**: Calcola la somma cumulativa (o altre operazioni se specificato).

```
from itertools import accumulate
```

```
result = accumulate([1, 2, 3, 4, 5])
```

```
print(list(result)) # Output: [1, 3, 6, 10, 15]
```

## Decorators in Python

Un **decorator** è un oggetto Python utilizzato per modificare una funzione, un metodo o una definizione di classe. Esso accetta l'oggetto originale come input, lo modifica e restituisce un nuovo oggetto modificato, associandolo al nome originale.

### Concetto Base: Wrapping di una Funzione

Un decorator è essenzialmente una **funzione che accetta una funzione come argomento**, definisce una funzione "**wrapper**" interna e restituisce questa funzione wrapper.

```

def my_decorator(func):
    def wrapper():
        print("Something happens before the function.")
        func()
        print("Something happens after the function.")
    return wrapper

def say_hello():
    print("Hello!")

# Uso manuale del decorator
say_hello = my_decorator(say_hello)
say_hello()

```

Output: Something happens before the function. Hello! Something happens after the function.

Python offre una **sintassi più semplice** per applicare decorators usando @. Questa forma evita di ripetere il nome della funzione decorata.

```
@my_decorator
```

```
def say_hello():
    print("Hello!")
```

```
say_hello()
```

Questo è equivalente a: say\_hello = my\_decorator(say\_hello)

### Decoratore do\_twice

Il decoratore **do\_twice** è una funzione che prende come argomento un'altra funzione (func) e ritorna una funzione interna (wrapper\_do\_twice). Il wrapper\_do\_twice esegue la funzione func due volte.

```

def do_twice(func):
    def wrapper_do_twice():
        func()      # the wrapper calls the
        func()      #     argument twice
    return wrapper_do_twice

@do_twice          # decorate the following
def say_hello():   # a sample function
    print("Hello!")

```

>>> say\_hello() # the wrapper is called

Hello!

Hello!

**Limite:** do\_twice non funziona con funzioni che accettano argomenti.

```

@do_twice          # does not work with parameters!!
def echo(str):    # a function with one parameter
    print(str)

```

>>> echo("Hi...") # the wrapper is called

TypErr: wrapper\_do\_twice() takes 0 pos args but 1 was given

>>> echo()

TypErr: echo() missing 1 required positional argument: 'str'

Per superare il limite di do\_twice, si utilizza una versione modificata, **do\_twice\_args**, che accetta funzioni con parametri. Il decoratore utilizza \*args e \*\*kwargs per supportare qualsiasi numero di argomenti posizionali e keyword.

```

def do_twice_args(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice

```

```

@do_twice_args
def say_hello():
    print("Hello!")

```

>>> say\_hello()

Hello!

Hello!

```

@do_twice_args
def echo(str):
    print(str)

```

>>> echo("Hi... ")

Hi...

Hi...

Il decoratore permette di passare argomenti e restituire il risultato della funzione decorata.

Per preservare i metadati della funzione decorata (ad esempio `__name__` e `__doc__`), si utilizza `@functools.wraps(func)`.

Esempio di implementazione generale:

```
import functools
def decorator(func):
    @functools.wraps(func)      #supports introspection
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

## Example: Measuring running time

```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
```

I **decoratori** hanno numerosi utilizzi pratici in Python, oltre alla semplice modifica del comportamento di una funzione.

**Debugging:** I decoratori possono monitorare l'esecuzione di una funzione, registrando gli argomenti passati, il valore restituito e altre informazioni utili per il debugging.

```
def debug(func):
    def wrapper(*args, **kwargs):
        print(f"Chiamata a {func.__name__} con args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"Risultato: {result}")
        return result
    return wrapper

@debug
def add(a, b):
    return a + b

add(3, 4) # Output: Informazioni sugli argomenti e sul risultato.
```

**Registrazione di plugin:** I decoratori possono aggiungere funzioni o metodi a un registro globale, rendendole disponibili come parte di un sistema o framework senza modificare direttamente il loro codice.

```
plugins = []

def register(func):
    plugins.append(func)
    return func

@register
def plugin_1():
    print("Plugin 1 eseguito")

@register
def plugin_2():
    print("Plugin 2 eseguito")

for plugin in plugins:
    plugin() # Esegue tutti i plugin registrati.
```

**Controlli nelle applicazioni web:** Nei framework web, come Flask o Django, i decoratori vengono utilizzati per verificare che un utente sia autenticato o autorizzato a eseguire determinate azioni.

```
def requires_auth(func):
    def wrapper(*args, **kwargs):
        user_authenticated = True # Simulazione
        if not user_authenticated:
            raise PermissionError("Utente non autenticato")
        return func(*args, **kwargs)
    return wrapper

@requires_auth
def access_sensitive_data():
    print("Accesso consentito ai dati sensibili")

access_sensitive_data()
```

**Metodi statici e di classe:** Python fornisce decoratori predefiniti per trasformare i metodi di una classe:

**@staticmethod:** Rende il metodo invocabile senza istanziare la classe.

**@classmethod:** Permette di invocare il metodo sulla classe stessa, passando la classe come primo argomento.

```
class MyClass:
    @staticmethod
    def static_method():
        print("Metodo statico chiamato")

    @classmethod
    def class_method(cls):
        print(f"Metodo di classe chiamato sulla classe {cls}")

MyClass.static_method() # Output: Metodo statico chiamato
MyClass.class_method() # Output: Metodo di classe chiamato sulla classe
```

I decoratori possono essere:

**Nidificati:** È possibile applicare più decoratori a una funzione. L'ordine in cui vengono applicati è dal più vicino al più esterno, ma l'esecuzione avviene in ordine inverso.

```
def uppercase(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs).upper()
    return wrapper

def exclamatory(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) + "!"
    return wrapper

@uppercase
@exclamatory
def greet(name):
    return f"Hello {name}"

print(greet("Alice")) # Output: HEL! ALICE!
```

**Personalizzati con argomenti:** Decoratori più flessibili possono essere definiti accettando parametri tramite un ulteriore livello di annidamento.

```
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def say_hello():
    print("Ciao!")

say_hello()
# Output:
# Ciao!
# Ciao!
# Ciao!
```

**Definiti come classi:** I decoratori possono essere implementati come classi per offrire maggiore struttura e flessibilità.

```
class Logger:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f"Chiamata a {self.func.__name__} con args={args}, kwargs={kwargs}")
        result = self.func(*args, **kwargs)
        print(f"Risultato: {result}")
        return result

    @Logger
    def multiply(a, b):
        return a * b

    multiply(2, 5)
    # Output:
    # Chiamata a multiply con args=(2, 5), kwargs={}
    # Risultato: 10

    def cache(func):
        cache_data = {} # Dizionario per memorizzare i risultati

        def wrapper(*args):
            if args in cache_data: # Se gli argomenti sono già in cache, restituisci il valore
                return cache_data[args]
            result = func(*args) # Altrimenti, calcola il risultato
            cache_data[args] = result # Salva il risultato nella cache
            return result

        return wrapper
```

Questo esempio utilizza un decoratore per **memorizzare nella cache i risultati delle chiamate a una funzione**, evitando di ricalcolare il risultato per gli stessi input. Mantiene una cache per i risultati delle chiamate alla funzione decorata. Usa un dizionario per mappare ogni combinazione di argomenti (chiave) al risultato calcolato (valore). Controlla che gli argomenti siano presenti in cache. In tal caso restituisce il risultato presente in cache, altrimenti ricalcola la funzione e inserisce il risultato in cache.

## AP-28: Python: OOP, iterators and the GIL

### Namespace e Scope in Python

Un **namespace** è una struttura in Python che **associa nomi a oggetti**, fungendo da **dizionario** che collega **identificatori** (nomi di variabili, funzioni, classi, ecc.) ai **rispettivi oggetti in memoria**.

Ogni namespace è isolato dagli altri, il che consente di utilizzare gli stessi nomi in contesti diversi senza conflitti. Ad esempio, variabili locali in una funzione, variabili globali in un modulo e funzioni built-in risiedono in namespace distinti.

#### Tipi di Namespace:

**Builtins (Predefiniti):** Namespace che contiene funzioni e oggetti **predefiniti** di Python, come len(). Vengono **creati all'avvio** dell'interprete e sono accessibili **ovunque** nel programma.

```
import builtins

print(builtins.len([1, 2, 3])) # Output: 3
```

**Globali:** Namespace che contiene i **nomi** di variabili, funzioni e classi **definiti** all'interno del **modulo**. Ogni modulo in Python ha il proprio namespace globale, creato al caricamento del modulo. Nell'interprete interattivo di Python, tutte le definizioni effettuate nell'interprete vengono aggiunte al namespace globale di `__main__`.

```
variabile_globale = "ciao"

def funzione_globale():
    print("Questa è una funzione globale")

class MiaClasse:
    attributo_classe = 100

# Nell'interprete interattivo:
>>> import mio_modulo
>>> print(mio_modulo.variabile_globale) # Output: Ciao
>>> mio_modulo.funzione_globale()      # Output: Questa è una funzione globale
>>> print(mio_modulo.MiaClasse.attributo_classe) # Output: 100
```

**Locali:** Namespace che contiene le variabili locali e i parametri **definiti** all'interno di una **funzione**. I namespace locali esistono esclusivamente durante l'esecuzione della funzione. Vengono creati **all'inizio della chiamata** e distrutti al **termine** dell'esecuzione.

**Classi:** Ogni **classe** o oggetto può avere il **proprio** namespace, contenente i suoi attributi e metodi. Il namespace è accessibile tramite `NomeClasse.__dict__`

**Accesso ai nomi:** Un **nome x** nel **modulo m** è un **attributo** di m, ed è **accessibile** usando il nome qualificato **m.x**. Può essere modificato o **eliminato** dal namespace con l'istruzione **del** se è scrivibile.

```
# mio_modulo.py
x = 5

def funzione():
    print("Funzione in mio_modulo")

# Nell'interprete interattivo:
>>> import mio_modulo
>>> print(mio_modulo.x) # Output: 5
>>> mio_modulo.x = 10
>>> print(mio_modulo.x) # Output: 10
```

### Scope

Uno **scope** (ambito) definisce la regione testuale di un programma Python dove un namespace è direttamente accessibile.

#### Tipi di Scope:

**Locale:** Contiene i nomi locali specifici della funzione corrente.

**Non local (Enclosing):** Include i nomi non locali delle funzioni di livello più alto, dal punto di vista delle funzioni annidate.

**Globale:** Contiene i nomi definiti a livello di modulo.

**Built-in:** Contiene i nomi predefiniti forniti da Python (es. print, len).

```
python
x = "globale"

def outer():
    x = "enclosing"

    def inner():
        x = "locale"
        print("x nello scope locale:", x)

    inner()

outer()
print("x nello scope globale:", x)
```

**Ordine di ricerca dei Namespace:** Scope locale -> Scope delle funzioni annidate -> Scope globale -> Scope built-in.

**Regole:** Le assegnazioni avvengono nello **scope locale** per impostazione **predefinita**. La keyword **global** viene usata per modificare variabili globali. Usa **nonlocal** per modificare variabili nello scope di chiusura (enclosing).

### Scoping rules

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"

    do_local()
    print('After local assignment:', spam)      # not affected
    do_nonlocal()
    print('After nonlocal assignment:', spam)    # affected
    do_global()
    print('After global assignment:', spam)      # not affected

scope_test()
print("In global scope:", spam)
```

After local assignment: test spam  
After nonlocal assignment: nonlocal spam  
After global assignment: nonlocal spam  
In global scope: global spam

In Python, le strutture di controllo come i loop non introducono un nuovo scope.

```
def test():
    for a in range(5):
        b = a % 2
        print(b) # Accesso a 'b' definito nel loop
    print(b) # 'b' è ancora accessibile anche fuori dal loop
test()
```

### Closures in Python

Una **closure** è una **funzione** annidata ad un'altra funzione che **"ricorda"** le **variabili non locali** della **funzione esterna** anche dopo che quest'ultima è **terminata**.

Le variabili **non locali** vengono mantenute attraverso l'attributo **\_\_closure\_\_** della funzione interna.

```

def counter_factory():
    counter = 0 # Variabile locale alla funzione esterna

    def counter_increaser():
        nonlocal counter # Indica che si modifica la variabile esterna
        counter = counter + 1
        return counter

    return counter_increaser

f = counter_factory() # 'f' diventa la funzione restituita
print(f()) # Output: 1
print(f()) # Output: 2
print(f.__closure__) # Mostra le variabili mantenute dalla closure

```

Counter è mantenuto in memoria grazie alla closure. La funzione interna può accedere e modificare counter usando nonlocal.

## OOP, iterators and the GIL

**L'incapsulamento** è un principio che consiste nel dividere il codice in un'implementazione **pubblica, accessibile** agli utenti per interagire con l'oggetto, e un'implementazione **privata, nascosta** agli utenti e riservata alla logica interna.

**L'ereditarietà** è una proprietà che permette di **creare sottoclassi** che **ereditano** attributi e metodi dalla classe **genitore**. Le **sottoclassi** possono anche aggiungere nuovi attributi e metodi.

Il **polimorfismo** consente di **sovrascrivere** i metodi di una classe base nelle sue **sottoclassi**, permettendo a queste ultime di fornire **un'implementazione specifica** pur mantenendo la stessa interfaccia. Questo concetto include l'**inclusione polimorfica**, che permette a un oggetto di una sottoclasse di essere **trattato** come un oggetto della classe genitore.

Python supporta tutte le caratteristiche standard della OOP, tra cui:

**Ereditarietà multipla:** Una classe può **ereditare da più classi** base, combinando attributi e metodi di **ciascuna** in un'unica sottoclasse.

**Sovrascrittura dei metodi:** Una sottoclasse può **ridefinire** i metodi della classe base, fornendo un **comportamento specifico** che sostituisce quello originale.

**Creazione dinamica:** Le classi sono create a **runtime**, consentendo di definirle, modificarle o estenderle durante l'esecuzione del programma.

## Classe

Una **classe** è un **modello** utilizzato per definire un **nuovo tipo di dato** che incapsula:

**Attributi interni:** Variabili che rappresentano le **proprietà** dell'oggetto che descrivono il **suo stato**.

**Metodi:** **Funzioni interne alla classe** che consentono di manipolare o **interagire** con gli **attributi**, definendo i **comportamenti** dell'oggetto.

Un **oggetto** è un'**istanza** di una classe, ovvero una **copia unica** con uno **stato proprio**, basate sulla **struttura** definita dalla classe. In python una classe si definisce nel seguente modo:

**class** *ClassName*:

    <statement-1>

...

    <statement-n>

Dove ogni statement può essere un assegnamento di attributo o una definizione di funzione.

Durante la definizione di una classe, viene creato un nuovo **namespace**. Tutti i nomi, ovvero variabili e metodi, definiti all'interno della classe appartengono a questo namespace.

Quando la definizione di una classe viene completata, Python crea un oggetto speciale chiamato **oggetto classe**, che **rappresenta la classe stessa** e viene **associato** al **nome** della classe `ClassName`. Questo oggetto classe serve come punto di riferimento per creare nuove istanze e accedere agli attributi e ai metodi definiti nella classe. Due operazioni principali sono possibili sull'oggetto classe, l'**istanziazione della classe**, che consiste nel **creare oggetti** istanza della classe e **riferimenti agli attributi**, in cui gli attributi o i metodi definiti nella classe possono essere **acceduti** utilizzando la notazione del punto `(.)`.

```
obj = ClassName() # Creazione di un'istanza
obj.attribute # Accesso a un attributo
obj.method() # Chiamata di un metodo
```

### Example: Attribute reference on a class object

```
class Point:
    x = 0
    y = 0
    def str(): # no capture: needs qualified names to refer to x and y
        return "x = " + (str) (Point.x) + ", y = " + (str) (Point.y)
#-----
import ...
>>> Point.x
0
>>> Point.y = 3
>>> Point.z = 5 # adding new name
>>> Point.z
5
>>> def add(m,n):
    return m+n
>>> Point.sum = add # adding new function
>>> Point.sum(3,4)
7
```

```
Point
x = 0
y = 0
str()
y = 3
z = 5
sum = add(m,n)
```

```
#-----
>>> p1 = Point()
>>> p2 = Point()
>>> p1.x
0
>>> Point.y = 3
>>> p2.y
3
>>> p1.y = 5
>>> p2.y
3
```

```
Point
x = 0
y = 0
str()
y = 3
p1
y = 5
p2
```

Ogni **istanza** della classe introduce un **nuovo namespace** annidato nel namespace della classe. Le regole di visibilità permettono che tutti i nomi definiti nella classe siano **visibili** all'interno dell'istanza.

Se non è definito un **costruttore esplicito** (es. `__init__`), è comunque possibile creare un'istanza della classe con la sintassi `ClassName()`. Il namespace dell'istanza è vuoto inizialmente.

Nell'esempio precedente, `x` e `y` appartengono alla classe e sono condivisi tra tutte le istanze, a meno che non vengano sovrascritti localmente.

Quando si assegna un valore a un attributo di un'istanza specifica `p1.y`, viene creato un nuovo attributo locale nell'istanza `p1`, che non influisce sull'attributo di classe o su altre istanze.

### Metodi di Istanza in Python

Un **metodo di istanza** in Python è una **funzione** definita all'interno di una classe che **opera su un'istanza** della classe stessa.

```
def methodname(self, parameter1, ..., parameterN):
    statements
```

**self:** È il **primo** parametro di ogni metodo di istanza. Rappresenta **l'istanza** su cui il metodo viene invocato (equivalente a `this` in Java).

Gli altri parametri vengono utilizzati per **passare dati** al metodo.

Gli attributi dell'istanza vengono acceduti tramite **self** (es. `self.x`).

Gli attributi della classe vengono acceduti tramite **className.<attribute>** oppure `self.__class__.<attribute>`.

L'invocazione del metodo viene effettuata utilizzando la notazione (.):

**obj.methodname(arg1, ..., argN).**

La forma **esplicita**, definito come funzione della classe, corrisponde a `ClassName.methodname(obj, arg1, ..., argN)`.

```
class Foo:  
    def fun(self, par1, par2):  
        return par1 + par2  
  
obj = Foo()  
print(obj.fun(2, 3)) # Output: 5  
# Equivalente a  
print(Foo.fun(obj, 2, 3))
```

## Attributi di un oggetto in Python

Gli **attributi di istanza** sono accessibili tramite `self.x` (esistono nello spazio dei nomi dell'istanza).

Gli **attributi di classe** sono accessibili tramite `className.x` o `self.__class__.x` (esistono nello spazio dei nomi della classe).

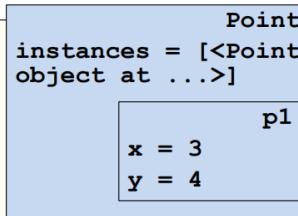
Un **costruttore** è un metodo speciale di istanza che si chiama automaticamente quando viene creata una nuova istanza di una classe. Un **costruttore** in Python viene definito dal metodo `__init__`.

```
def __init__(self, parameter1, ..., parameterN):  
    statements  
obj = className(arg1, ..., argN)
```

Il costruttore viene utilizzato per assegnare valori alle proprietà specifiche dell'oggetto e configurarlo.

In Python è consentito **un solo costruttore** per classe (non c'è supporto per l'overloading dei costruttori). È possibile usare **valori predefiniti** nei parametri per simulare comportamenti diversi.

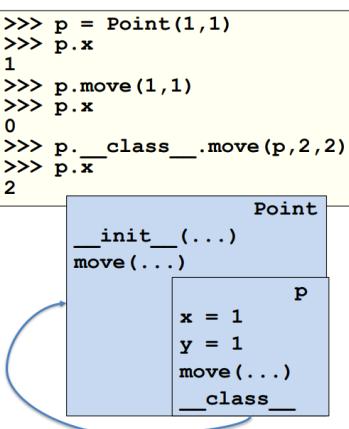
```
class Point:  
    instances = []  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
        Point.instances.append(self)  
#-----  
>>> p1 = Point(3,4)
```



The diagram shows the state of the `Point` class and its instance `p1`. The `Point` class has an attribute `instances` containing a reference to the `p1` object. The `p1` instance has attributes `x = 3` and `y = 4`.

Le istanze stesse sono dei namespace, quindi è possibile aggiungere funzioni ad esse manualmente. Usando le regole standard, i metodi aggiunti manualmente possono sovrascrivere i metodi di istanza definiti nella classe.

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def move(z,t):  
        self.x -= z  
        self.y -= t  
        self.move = move  
    def move(self,dx,dy):  
        self.x += dx  
        self.y += dy
```



```
>>> p = Point(1,1)  
>>> p.x  
1  
>>> p.move(1,1)  
>>> p.x  
0  
>>> p.__class__.move(p,2,2)  
>>> p.x  
2
```

The diagram illustrates method overriding. The `Point` class has methods `__init__`, `move`, and `move`. An instance `p` is created with `x=1`, `y=1`. The `move` method is called with parameters `(1,1)`, which calls the `__init__` method of the `Point` class, setting `x=0` and `y=0`. Then it calls the `move` method again with parameters `(2,2)`, which calls the `move` method of the `Point` class, setting `dx=2` and `dy=2`. A blue arrow points from the original `move` method in the class to the overridden `move` method in the instance `p`.

## Rappresentazione in stringa (`__str__`)

**Metodo `__str__`:** Metodo utilizzato per fornire una rappresentazione testuale di un oggetto. E' equivalente al metodo `toString()` di Java.

Se il metodo `__str__` è definito, viene chiamato automaticamente quando si utilizza `print` o si converte un oggetto in stringa (`str(obj)`).

- **Method overloading:** you can define special instance methods so that Python's built-in operators can be used with your class

#### Binary Operators

Operator	Class Method
-	<code>__sub__(self, other)</code>
+	<code>__add__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>

#### Unary Operators

-	<code>__neg__(self)</code>
+	<code>__pos__(self)</code>

Operator	Class Method
==	<code>__eq__(self, other)</code>
!=	<code>__ne__(self, other)</code>
<	<code>__lt__(self, other)</code>
>	<code>__gt__(self, other)</code>
<=	<code>__le__(self, other)</code>
>=	<code>__ge__(self, other)</code>

```
class Point: # example
...
def __add__(self,other):
    return Point(self.x + other.x,
                 self.y + other.y)
def __neg__(self):
    return Point(-self.x, - self.y)
```

- Analogous to C++ overloading mechanism:
  - **Pros:** very compact syntax
  - **Cons:** may be more difficult to read if not used with care

13

## Ereditarietà Multipla in Python

Una classe può essere definita come **classe derivata** utilizzando la seguente sintassi:

**class Derived(BaseClass):**

statements

La classe Derived eredita automaticamente proprietà e metodi dalla classe base (BaseClass) senza richiedere meccanismi aggiuntivi.

**Namespace annidati:** Il namespace della classe derivata (Derived) è annidato nei namespace delle classi base. Questo significa che, se un attributo o un metodo non viene trovato nella classe derivata, Python proseguirà la ricerca nelle classi base seguendo l'ordine di ereditarietà.

```
class BaseClass:
    def greet(self):
        print("Hello from BaseClass!")

class Derived(BaseClass):
    def greet(self):
        print("Hello from Derived!")

obj = Derived()
obj.greet() # Output: "Hello from Derived!"
```

## Problema del Diamante e MRO

Il **problema del diamante** si verifica quando una classe derivata **eredita da due classi** che a loro volta **ereditano** da una **classe base comune**. Questo può creare ambiguità nella risoluzione dei metodi o attributi comuni definiti nella classe base.

```
class A:
    def greet(self):
        print("Hello from A!")

class B(A):
    def greet(self):
        print("Hello from B!")

class C(A):
    def greet(self):
        print("Hello from C!")

class D(B, C):
    pass

# Creazione dell'oggetto
obj = D()
obj.greet()
```



Python risolve questo problema utilizzando il **Method Resolution Order (MRO)**.

L'MRO garantisce che i metodi vengano risolti in un ordine prevedibile, evitando ambiguità. È possibile visualizzare l'MRO di una classe chiamando `ClassName.mro()`.

Il **funzionamento** del **MRO** è il seguente: La ricerca inizia nella **classe stessa** (D).

Passa alla **prima classe genitore** (B) secondo l'ordine specificato nella definizione della classe. Prosegue verso la **seconda classe genitore** (C). Infine, arriva alla **classe base comune** (A) e termina con la **classe object**.

## Incapsulamento in Python

In Python, l'incapsulamento si basa principalmente su convenzioni e strumenti come il **name mangling** per gestire l'accesso agli attributi e prevenire conflitti di nomi, piuttosto che su restrizioni rigide.

I nomi che iniziano con un **singolo underscore** (`_nameVar`) sono considerati **non pubblici** e rappresentano dettagli di implementazione che non dovrebbero essere usati al di fuori della classe. Questa **convenzione** è utile per indicare che tali elementi non fanno parte dell'interfaccia pubblica, ma **non impedisce** realmente l'accesso ad essi. D'altra parte, Python utilizza il **name mangling** per gestire nomi che iniziano con **due underscore** (`__nameVar`). Questo meccanismo **modifica** automaticamente **il nome** degli attributi aggiungendo come prefisso **il nome della classe**, ad esempio trasformando `_attr` in `_ClassName_attr`.

Il name mangling aiuta a evitare conflitti di nomi in gerarchie di classi, specialmente quando una sottoclasse definisce membri con lo stesso nome della classe base.

```
class BaseClass:
    def __init__(self):
        self.__private_attr = "Attributo di BaseClass"

    def get_private_attr(self):
        return self.__private_attr

class SubClass(BaseClass):
    def __init__(self):
        super().__init__()
        self.__private_attr = "Attributo di SubClass"

    def get_subclass_attr(self):
        return self.__private_attr

# Creazione degli oggetti
base_obj = BaseClass()
sub_obj = SubClass()

# Accesso agli attributi "mangled"
print("Nome mangled in BaseClass:", dir(base_obj)) # Mostra __BaseClass__private_attr
print("Nome mangled in SubClass:", dir(sub_obj)) # Mostra __SubClass__private_attr

# Stampa dei valori attraverso i metodi
print(base_obj.get_private_attr()) # Output: "Attributo di BaseClass"
print(sub_obj.get_private_attr()) # Output: "Attributo di BaseClass"
print(sub_obj.get_subclass_attr()) # Ou ↓: "Attributo di SubClass"
```

**Vantaggi del name mangling:** Protegge gli attributi o metodi di una classe base da sovrascritture accidentali nelle sottoclassi. Rende meno probabile l'accesso non intenzionale a determinati attributi dall'esterno della classe.

**Limitazioni del name mangling:** Gli attributi mangled possono comunque essere raggiunti conoscendo il nome generato `_ClassName_attr`. L'uso eccessivo di name mangling può rendere il codice meno leggibile. Se i nomi delle classi sono generati dinamicamente (ad esempio, tramite `type()`), il comportamento del name mangling può diventare meno prevedibile e complicare la manutenzione del codice.

```
class BaseClass:
    def update(self, data):
        print(f"[BaseClass] Aggiornamento con dati: {data}")

class SubClass(BaseClass):
    def __init__(self):
        # Salva l'implementazione originale del metodo in un attributo privato
        self.__original_update = super().update

    def update(self, data):
        print(f"[SubClass] Aggiornamento specifico con dati: {data}")
        print(f"[SubClass] Ora richiamo l'implementazione originale...")
        self.__original_update(data)

# Creazione degli oggetti
base_obj = BaseClass()
sub_obj = SubClass()

# Chiamate ai metodi
print("Esempio con BaseClass:")
base_obj.update("Dati base") # Output: [BaseClass] Aggiornamento con dati: Dati base

print("\nEsempio con SubClass:")
sub_obj.update("Dati sottoclasse")
```

## Metodi statici, di classe e iteratori

Un **metodo statico** è una funzione definita all'interno di una classe che **non richiede** il parametro speciale **self** e **non può accedere** agli attributi o ai metodi **dell'istanza**. È definito utilizzando il decoratore `@staticmethod`. Può essere chiamato sia sulla classe

che su un'istanza, ed è spesso usato quando la funzione ha senso in relazione alla classe ma **non necessita** di interagire con i suoi dati o metodi.

Un **metodo di classe**, invece, è simile a un metodo statico, ma riceve come primo parametro **un riferimento alla classe**, convenzionalmente chiamato cls. È definito utilizzando il decoratore `@classmethod`. Questo permette al metodo di **interagire con la classe**, come **accedere** o modificare **attributi** della **classe**. Può essere invocato sia sulla classe che su un'istanza.

Un **iteratore** è un oggetto che consente di **attraversare** gli elementi di una **collezione** in modo **sequenziale**, indipendentemente dalla sua implementazione specifica. In Python, gli iteratori vengono utilizzati implicitamente, ad esempio, nei costrutti for. Gli oggetti **iterabili** devono implementare il metodo speciale `__iter__()`, che restituisce un iteratore. Gli **iteratori**, a loro volta, devono implementare i metodi `__iter__()` e `__next__()`, in cui `__iter__()` restituisce l'iteratore stesso e `__next__()` restituisce il **prossimo valore** della sequenza. Se non ci sono più elementi da restituire, solleva l'eccezione `StopIteration`.

Gli iteratori sono **usa e getta**: una volta esauriti gli elementi, continueranno a sollevare `StopIteration`.

- Example:

```
for element in [1, 2, 3]:  
    print(element)
```

```
>>> list = [1,2,3]  
>>> it = iter(list)  
>>> it  
<listiterator object at 0x00A1DB50>  
>>> it.__next__()  
1  
>>> it.__next__()  
2  
>>> it.__next__()  
3  
>>> it.__next__() -> raises StopIteration
```

I **generatori** sono uno strumento per creare iteratori. Si definiscono come **funzioni normali**, ma utilizzano la parola chiave **yield** per **restituire valori** uno alla volta, mantenendo lo **stato interno** della funzione tra una chiamata e l'altra. Questo consente di processare dati senza doverli caricare tutti in memoria, rendendo i generatori particolarmente utili per lavorare con sequenze di grandi dimensioni o flussi di dati.

Ogni volta che si utilizza la funzione `next()` su un generatore, l'esecuzione riparte dal punto in cui era stata interrotta, fino a incontrare un nuovo `yield`. Questo comportamento lazy consente ai generatori di restituire un elemento alla volta, risparmiando memoria rispetto a strutture come liste o tuple. Inoltre, i generatori implementano automaticamente i metodi `__iter__()` e `__next__()`, rendendoli compatti ed eleganti.

Quando un generatore termina, solleva automaticamente l'eccezione `StopIteration`, segnalando la fine dell'iterazione, senza la necessità di gestire manualmente lo stato interno.

```
def genera_numeri(n):  
    """Generatore che produce numeri da 1 a n"""  
    i = 1  
    while i <= n:  
        print(f"Generatore: Producio il numero {i}")  
        yield i  
        i += 1  
    print("Generatore: Fine dell'iterazione")  
  
# Creiamo un generatore che produce numeri da 1 a 5  
gen = genera_numeri(5)  
  
# Iteriamo manualmente utilizzando next()  
try:  
    print("Chiamata a next():", next(gen)) # Output: 1  
    print("Chiamata a next():", next(gen)) # Output: 2  
    print("Chiamata a next():", next(gen)) # Output: 3  
    print("Chiamata a next():", next(gen)) # Output: 4  
    print("Chiamata a next():", next(gen)) # Output: 5  
    print("Chiamata a next():", next(gen)) # Solleva StopIteration  
except StopIteration:  
    print("Eccezione: StopIteration sollevata, fine del generatore")
```

La funzione `genera_numeri` è un generatore perché utilizza la parola chiave `yield`. Ogni volta che viene chiamato `yield`, la funzione restituisce un valore e sospende l'esecuzione, mantenendo lo stato interno (`i`). Il generatore viene poi utilizzato come un normale iteratore, chiamando `next()`.

## Typing in Python

In Python, il sistema di **typing** è **dinamico** e segue il principio del **duck typing**, cioè il tipo di un oggetto è determinato in base ai suoi **comportamenti e metodi**, piuttosto che alla sua dichiarazione esplicita. Anche se Python è dinamicamente tipizzato, permette di utilizzare **annotazioni di tipo** (type hints) per specificare i tipi delle variabili, dei parametri di funzione e del valore di ritorno.

```
def greetings(name: str) -> str:  
    return 'Hello ' + name
```

In questo caso, `name` è annotato come una stringa (`str`), e il valore restituito dalla funzione è anch'esso una stringa (`-> str`). Queste annotazioni migliorano la leggibilità del codice e possono essere utilizzate per il controllo statico dei tipi.

Il modulo **typing** offre una vasta gamma di strumenti per supportare queste annotazioni, ad esempio tipi generici (`List`, `Dict`, ecc.) o strutture complesse come unioni (`Union`) o opzionali (`Optional`).

Le annotazioni di tipo non vengono eseguite a runtime: Python le ignora durante l'esecuzione, mantenendo il suo comportamento dinamico. Tuttavia, strumenti esterni come **mypy** possono essere utilizzati per analizzare il codice e verificare che i tipi specificati siano corretti, rilevando potenziali errori.

## Polimorfismo in Python

Il polimorfismo in Python si manifesta in modi dinamici, sfruttando le caratteristiche del linguaggio, che differiscono da quelle di linguaggi fortemente tipizzati come Java o C++.

### Overloading:

In Python, l'overloading **non è supportato** nativamente. Non è possibile definire più metodi con lo stesso nome ma con parametri differenti all'interno della stessa classe. Tuttavia, Python compensa questa mancanza attraverso:

**Parametri di default:** Si possono definire **valori predefiniti** per alcuni parametri, consentendo chiamate con **argomenti opzionali**.

```
class Example:  
    def greet(self, name="World"):  
        print(f"Hello, {name}!")  
  
obj = Example()  
obj.greet()      # Output: Hello, World!  
obj.greet("Alice") # Output: Hello, Alice!
```

**Typing dinamico:** Poiché i tipi dei parametri non sono vincolati, lo stesso metodo può funzionare con **oggetti di tipi diversi**, purché **supportino** le operazioni richieste.

```
class Dog:  
    def speak(self):  
        return "Woof!"  
  
class Cat:  
    def speak(self):  
        return "Meow!"  
  
def animal_sound(animal):  
    print(animal.speak())  
  
dog = Dog()  
cat = Cat()  
  
animal_sound(dog) # Output: Woof!  
animal_sound(cat) # Output: Meow!
```

**Overriding:** Python supporta completamente l'**overriding**, che permette a una sottoclasse di **riconfigurare** un metodo della classe base con lo stesso nome, **sostituendone** il comportamento. Questo è reso possibile dalla struttura a **namespace** annidati, dove il metodo della sottoclasse ha priorità rispetto a quello della classe base.

```

class BaseClass:
    def greet(self):
        print("Hello from BaseClass!")

class SubClass(BaseClass):
    def greet(self):
        print("Hello from SubClass!")

obj = SubClass()
obj.greet() # Output: Hello from SubClass!

```

**Generics:** Python supporta i **tipi generici** tramite il modulo typing. Grazie ai generics, una singola funzione o classe può gestire diversi tipi di input senza riscrivere il codice per ciascun tipo specifico.

```

from typing import TypeVar, List

# Definiamo una variabile di tipo generico T
T = TypeVar('T')

def primo_elemento(lista: List[T]) -> T:
    return lista[0]

# Utilizzo con diverse liste
numeri = [1, 2, 3]
parole = ["apple", "banana", "cherry"]

print(primo_elemento(numeri)) # Output: 1
print(primo_elemento(parole)) # Output: apple

```

## Gestione della memoria in Python

In Python, la gestione della memoria è affidata al **garbage collector**, che utilizza un sistema combinato di **reference counting** e uno schema chiamato **mark & sweep** per gestire i cicli di memoria.

### Reference counting

Il **reference counting** funziona **assegnando** a ogni oggetto **un contatore** che tiene traccia del **numero di riferimenti** a quell'oggetto. Quando il contatore **arriva a zero** (ovvero, nessuna parte del programma utilizza più quell'oggetto), la memoria occupata dall'oggetto può essere **liberata immediatamente**.

I limiti del reference counting sono che richiede **memoria aggiuntiva** per gestire i contatori e non è in grado di rilevare e risolvere **strutture cicliche**, cioè situazioni in cui oggetti si **riferiscono reciprocamente** e non sono più accessibili da altre parti del programma.

### Mark & sweep

Per gestire queste situazioni, entra in gioco lo schema **mark & sweep**. Il processo di **mark** ha come obiettivo identificare tutti gli oggetti **raggiungibili** nel programma, ovvero quelli ancora referenziati e utilizzati. Successivamente, la fase di **sweep** scansiona l'intera memoria heap alla ricerca degli oggetti **non marcati** durante la fase di mark e libera la loro memoria.

È possibile accedere ai valori di reference counting tramite la libreria ctypes.

```

import ctypes

my_list = [1, 2, 3]

# finding the id of list object
my_list_address = id(my_list)

# finds reference count of my_list
ref_count = ctypes.c_long.from_address(my_list_address).value

print(f"Ref count for my_list is: {ref_count}")

```

Python non richiede **deallocazioni manuali**, poiché la gestione degli oggetti inutilizzati è automatica. L'istruzione **del** non rimuove direttamente l'oggetto dalla memoria, ma elimina la sua associazione dal **namespace**. Di conseguenza, Python evita problemi comuni nei linguaggi con gestione manuale della memoria, come i **dangling pointers** (puntatori che puntano a memoria già liberata) e il **double free** (tentativi di liberare due volte la stessa area di memoria).

## Global Interpreter Lock

**Il Global Interpreter Lock (GIL)** è un meccanismo dell'interprete **CPython** che impedisce a più thread di eseguire **contemporaneamente** il bytecode Python. E' un lock che viene **acquisito e rilasciato periodicamente**, dopo un certo numero di istruzioni bytecode o allo scadere di un tempo prestabilito e **durante operazioni di I/O**, consentendo ad altri thread di progredire mentre uno è bloccato.

**Limita il parallelismo effettivo** su macchine multi-processore per i carichi di lavoro **CPU-bound**, poiché in ogni momento solo un thread per volta può eseguire il bytecode. **Non sostituisce** i meccanismi di sincronizzazione a livello applicativo: se due o più thread condividono strutture dati o variabili globali, è comunque necessaria una gestione esplicita (lock, semafori, ecc.) per evitare condizioni di race condition.

**Funzionamento del GIL:** Il **GIL** è un lock che ogni thread deve acquisire prima di poter eseguire un insieme di istruzioni python. In qualsiasi istante, soltanto **un** thread può detenere il GIL, mentre gli altri **rimangono in attesa** di ottenere il lock per eseguire ulteriori istruzioni Python.

Il GIL **non** viene mantenuto dallo stesso thread per tutta la durata del programma, ma viene **forzatamente rilasciato** in caso di **scadenza dell'intervallo** di switch, quando un thrad ha raggiunto il limite di tempo o di istruzioni eseguite, o quando un thread effettua un'operazione che **blocchia l'esecuzione** (IO).

Per task i cpu-bound, dato che GIL impedisce l'utilizzo simultaneo di più core della CPU, aggiungere più thread può aumentare l'overhead dovuto ai cambi di contesto senza migliorare o addirittura peggiorando le prestazioni.

Per i task IO-bound, durante le operazioni di I/O, Python rilascia temporaneamente il GIL, permettendo ad altri thread di eseguire. Questo rilascio permette di parallelizzare i thread I/O-bound, migliorando l'efficienza e la reattività del programma.

Esistono implementazioni alternative al GIL che permettono di rilasciare temporaneamente il GIL o non utilizzarlo, ma non hanno avuto successo.

### GIL e Race Conditions

In Python, le **race conditions** possono verificarsi quando due o più thread accedono e modificano una risorsa condivisa contemporaneamente. Il GIL garantisce che un solo thread alla volta esegua un insieme di istruzioni **bytecode**, ma se un'operazione su un dato condiviso non è atomica, questo crea race condition.

Per evitare race condition, è necessario utilizzare meccanismi di sincronizzazione.

```
def increment():
    global counter
    for _ in range(100000):
        counter += 1 # Operazione non atomica

threads = []

# Creazione di due thread
for _ in range(2):
    t = threading.Thread(target=increment)
    threads.append(t)
    t.start()
```

- L'operazione `counter += 1` è composta da tre passaggi:
  1. Leggere il valore corrente di `counter`.
  2. Incrementarlo.
  3. Scrivere il nuovo valore di `counter`.
- Poiché questi passaggi non sono eseguiti in modo atomico, un thread può essere interrotto dopo il primo o il secondo passaggio, causando inconsistenze.

Valore finale **del** contatore: **187346**

## Domande

### JVM Instruction set

**Corra:** Cosa significa che nella JVM l'instruction set non è ortogonale?

Significa che le istruzioni della JVM non sono generiche per tutti i tipi ma esistono istruzioni specifiche per ciascun tipo, come iadd per interi e fadd per numeri in virgola mobile.

## Framework

**Corra:** Cosa si intende per inversion of control? In quali contesti si trova questo concetto?

Per Inversion of Control si intende che il flusso di esecuzione appartiene al framework piuttosto che al codice scritto dallo sviluppatore. Questo si occupa di chiamare i metodi scritti dallo sviluppatore.

## Haskell

**Corra:** L'IO Monad in Haskell permette di gestire correttamente gli effetti collaterali come input/output e il concetto di variabile. Perché?

In Haskell, le funzioni **non hanno effetti collaterali** visibili all'esterno.

L'IO Monad trasforma invece un'azione potenzialmente "impura" in un **valore di tipo IO** a, che rappresenta **la descrizione** di quell'effetto, ma **non lo esegue** immediatamente. L'esecuzione degli effetti avviene solo quando il programma "consegna" il **controllo al runtime** (tipicamente alla fine, in main), garantendo che tutto il resto del codice rimanga puro.

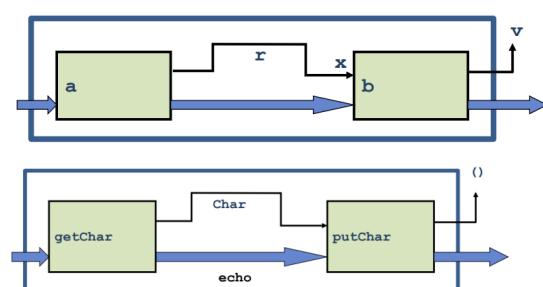
Invece di lasciare che le chiamate a funzioni con effetti collaterali possano avvenire in qualsiasi ordine, Haskell impone la costruzione di una **catena di azioni**, tramite **>= bind**, stabilita dal programmatore.

Per la gestione delle variabili, l'uso di IORef o di altre strutture mutabili rimane confinato dentro il tipo IO, evitando che la "mutabilità" contamini il resto del programma, che resta puramente funzionale. Questo modello non può essere rotto se non con l'operazione unsafePerform.

Attraverso l'uso dei monad, il codice che utilizza la monad IO dichiara come interagire col mondo esterno, ma la logica funzionale resta "pulita, perché gli effetti sono limitati al di fuori della parte puramente funzionale del linguaggio.

**Corra:** Come viene definita l'operazione bind?

L'operatore **>= (bind)** permette di **comporre azioni sequenziali**, collegando il risultato di una computazione al passo successivo.



Quando viene eseguita l'azione **a >= \x -> b**, il combinatore **>=** esegue i seguenti passaggi: **L'azione a (getChar)** viene eseguita, restituendo un **valore r** (carattere letto). Il **risultato r** è passato alla **funzione anonima \x -> b** (putchar), che prende in input il risultato della precedente computazione e restituisce una nuova computazione. La nuova computazione **b, basata su r, viene eseguita** (putchar sull'output di getChar). Il valore finale della seconda computazione (v) viene restituito.

## Implementazione

```
(>=) :: IO a -> (a -> IO b) -> IO b  
>= action nextAction = \world -> case action world of  
    (result, newWorld) -> nextAction result newWorld
```

**Esecuzione della prima azione (action):** L'azione **action** viene **applicata** sul mondo corrente (world) tramite il comando (case) **action world**, producendo un risultato e uno stato del mondo aggiornato. Queste vengono destrutturate in (**result**, **newWorld**). Result e newWorld vengono passati a nextAction **in modo esplicito tramite** l'operatore -> che consente di prenderli come parametri lambda. L'azione nextAction viene eseguita su result e sul nuovo stato del mondo (newWorld).

```
main :: IO ()  
main = getLine >>= (\name -> putStrLn ("Ciao, " ++ name ++ "!"))
```

**Corra:** Quale ruolo gioca il "mondo" nel garantire la stretta sequenzialità?

Ogni azione di IO riceve in ingresso lo stato globale del **mondo esterno** e ne **produce uno nuovo** in uscita modificato in base all'effetto dell'azione.

Non si può **"saltare"** un'azione o riordinare le operazioni, perché ognuna **dipende** dalla **versione aggiornata del mondo** generata dall'**azione precedente**. Questo garantisce la sequenzialità.

**Corra:** Qual è il meccanismo di parameter passing in Haskell?

**Gli argomenti delle funzioni non vengono valutati subito. Un'espressione viene valutata soltanto quando è effettivamente richiesta** per proseguire il calcolo.

Haskell costruisce dei **thunk**, ovvero delle strutture dati che "racchiudono" l'espressione da calcolare e l'ambiente lessicale necessario alla valutazione.

Quando viene calcolato il valore dell'espressione, il valore viene memorizzato nel thunk, in modo da evitare di rivalutare la stessa espressione più volte.

**Corra:** Continuare a ritardare l'esecuzione di una funzione può causare problemi di memoria?

Se un programma crea molte espressioni che non vengono mai valutate, tali espressioni rimangono in memoria, in attesa di un'eventuale valutazione, portando a potenziali sprechi.

*Le funzioni come map o filter, composte in catene lunghe e mai forzate, possono lasciare in sospeso molte trasformazioni (thunk) che vengono tutte "risvegliate" solo quando si accede a un elemento alla fine della catena , portando a creare e lasciare in memoria molti thunk, sprecando memoria.*

La memoria utilizzata dai thunk continua a crescere se non c'è un punto in cui li si valuti parzialmente.

```
-- Definizione di una lista grande  
bigList :: [Int]  
bigList = [1..1000000]  
  
-- Catena di trasformazioni senza forzare la valutazione  
processedList :: [Int]  
processedList = map (*2) (filter odd (map (+1) (filter even bigList)))  
  
-- Funzione principale che accede all'ultimo elemento  
main :: IO ()  
main = print $ last processedList
```

Quali sono le differenze tra le interfacce e le classi in Java, i traits in Rust e le typeclass in Haskell?

## Lambda

**Corra:** Come sono tipizzate le lambda expression in Java?

Il metodo generato da una lambda dipende dal contesto in cui la lambda è utilizzata. Il compilatore Java genera un'implementazione della lambda come un'**istanza di un'interfaccia funzionale**. Il metodo generato viene richiamato attraverso

un'interfaccia funzionale. Quando la lambda è passata o utilizzata, il metodo della sua interfaccia viene invocato.

Una **functional interface** è un'interfaccia Java che contiene **esattamente un metodo astratto**. Queste interfacce sono progettate per rappresentare un singolo comportamento che può essere implementato da una lambda expression o da un riferimento a metodo.

→ Il compilatore deduce il tipo di una lambda dal *tipo dell'interfaccia funzionale* atteso nel punto in cui la lambda viene utilizzata. Se non esiste un “target” adeguato, la lambda non è ben definita.

Il compilatore verifica che i parametri (il loro numero e tipo) della lambda e il suo tipo di ritorno siano compatibili con il metodo astratto definito nell'interfaccia funzionale di destinazione.

**Corra:** Fornisci qualche esempio di type inference in Java.

```
// Interfaccia funzionale
@FunctionalInterface
interface Operazione {
    int esegui(int a, int b);
}

// Utilizzo della lambda senza specificare i tipi
Operazione addizione = (a, b) -> a + b;

// Equivalentemente, specificando i tipi (opzionale)
Operazione addizioneConTipi = (int a, int b) -> a + b;
```

Una lambda può essere assegnata a una variabile di tipo **functional interface** o a un **parametro formale** di un metodo che accetta una functional interface.

Runnable r = () -> System.out.println("Running!");

list.forEach(s -> System.out.println(s)); // `forEach` accetta un `Consumer<T>`

Il compilatore **deduca automaticamente** i tipi di argomenti e del valore di ritorno della lambda **basandosi sulla firma** del **metodo astratto** della **functional interface** a cui la lambda viene assegnata.

La **firma della lambda** (tipi di parametri e ritorno) deve corrispondere alla firma del **metodo astratto** nella functional interface. Se una lambda implementa un Comparator<Integer>, il metodo astratto compare richiede due parametri Integer e restituisce un int: Comparator<Integer> comparator = (a, b) -> a - b;

```
public interface Comparator<T> { //java.util
    int compare(T o1, T o2);
}

public interface Runnable { //java.lang
    void run();
}

public interface Consumer<T>{ //java.util.function
    void accept(T t);
}

public interface Callable<V> //java.util.concurrent
    V call() throws Exception;
}
```

## Rust

**Quali sono i meccanismi di ownership e borrowing in Rust?**

Ogni valore è posseduto da una variabile, che lo identifica tramite un nome. Quando la proprietà di un valore viene trasferita, il proprietario originale perde l'accesso al valore. Quando il proprietario esce dallo scope, il valore viene deallocated.

Il meccanismo di ownership consiste nello spostare la proprietà di un valore quando una variabile viene assegnata all'altra. E' valido solo per tipi non primitivi, per tipi primitivi il valore viene solamente copiato.

Il borrowing consiste nel “prestare” il valore di una variabile ad un'altra. Può essere di due tipi: immutabile, in cui la variabile che prende in prestito può solo leggere, e mutable, in cui la variabile che prende in prestito può scrivere.

È permesso avere un riferimento mutable, ma non più di uno contemporaneamente. Se non ci sono riferimenti mutabili, è possibile avere più riferimenti immutabili contemporaneamente. Se esiste un riferimento mutable, non possono esistere riferimenti immutabili alla stessa risorsa nello stesso momento. Il proprietario non può modificare o liberare la risorsa mentre è presa in prestito in modo immutabile. Il proprietario non può leggere la risorsa mentre esiste un riferimento mutable.

### Cos'è uno smart pointer?

Gli **Smart Pointers** sono **riferimenti** che si comportano come puntatori tradizionali, ma con l'aggiunta di **metadati e funzionalità avanzate**. Vengono implementati come **struct**, con due caratteristiche principali: Implementano il trait **Deref**, che permette di accedere al valore sottostante come un riferimento normale (\*) && Implementano il trait **Drop**, che permette la gestione automatica della memoria quando escono dallo scope.

Un'altra caratteristica è la **Deref Coercion**, una funzionalità che consente **conversioni automatiche** da Smart Pointer a riferimenti normali per facilitare l'uso.

### Esempio di smart pointer che viola le regole di Rust

Un tipico esempio è lo *reference counting*, dove il puntatore tiene traccia di quante entità stanno condividendo la stessa risorsa. Per funzionare, il contatore viene incrementato e decrementato ogni volta che si crea o si distrugge un riferimento, dunque c'è una mutazione “nascosta” che avviene in modo condiviso tra più *reference*. In Rust, la regola detta *borrow rule* stabilisce che si può avere:

- **Un'unica** referenza mutable su un dato alla volta, **oppure**
- Più referenze immutabili, ma senza modifiche condivise a quell'oggetto.

Il contatore di riferimento, però, è un valore che dovrebbe essere mutato da tutti i puntatori (per tenere aggiornato il conteggio): questo creerebbe una violazione, perché è una “mutazione interna” condivisa (cioè accessibile e modificabile da più *reference* contemporaneamente).

### Come si gestisce in Rust

Nonostante queste regole, Rust fornisce comunque strumenti come `Rc<T>` o `Arc<T>` che usano un meccanismo chiamato *interior mutability* (basato su `UnsafeCell`): ciò “isola” il contatore di riferimento, consentendo una mutazione sicura e controllata. Se però si provasse a implementare un *reference counting* senza *interior mutability*, si incorrerebbe in una violazione del modello di sicurezza di Rust.

Quali sono le differenze tra le interfacce e le classi in Java, i traits in Rust e le typeclass in Haskell?

**Interfacce in Java** Definiscono un insieme di metodi che una classe deve implementare. Non contengono stato (solo costanti static final). Da Java 8 possono avere metodi di default e statici, ma restano sostanzialmente un “contratto” di metodi. Una classe può implementare più interfacce, acquisendo i metodi (che deve implementare o ereditare come default).

---

### Typeclass in Haskell

Sono “contratti” di funzioni: specificano nomi e firme delle funzioni che i tipi istanza devono fornire. Un tipo diventa istanza di una typeclass dichiarando instance `NomeTypeclass Tipo where ....` Permettono il *polimorfismo ad-hoc*: ogni tipo può fornire

la propria implementazione delle funzioni. Non descrivono dati e non creano oggetti: modellano solo comportamenti. Possono avere implementazioni predefinite (default).

---

## Traits in Rust

Raccolgono uno o più metodi (anche con implementazioni di default) che i tipi possono “adottare” dichiarando `impl NomeTrait for MioTipo`. Non contengono campi (quindi nessuno stato). Consentono *polimorfismo* vincolando i generics (fn `mio_mетодo<T: MioTrait>(x: T) {...}`).

Un tipo può implementare più trait, ma non esiste “ereditarietà” di stato come nelle classi: i trait possono estendersi tra loro solo per includere metodi aggiuntivi.

## Polimorfismo

¶ Polimorfismo: discuti il polimorfismo in Rust e C++, con particolare attenzione alla loro implementazione. Ci sono problemi di compilazione legati ai tipi? Come si compara con il polimorfismo in Java?

¶ Come viene gestito l’overloading nei principali linguaggi di programmazione trattati (Java, Haskell, C++)? Qual è il problema della type erasure in Java?

## Python

E come funziona l’overloading in Python?

L’overloading inteso come ridefinizione di una funzione con parametri diversi non è nativamente supportato in Python, ma viene implementato utilizzando il typing dinamico fornito dal linguaggio, tramite il quale possono essere passati oggetti di tipo diverso ma con lo stesso comportamento a una funzione che si aspetta quel comportamento, tramite valore di default assegnato ai parametri e all’uso di generics che consentono di operare su collezioni tramite vari tipi specificati dall’utente.

¶ Quali sono le regole di scoping in Python e quali costrutti del linguaggio introducono i namespace?

Un namespace è un’associazione del linguaggio Python, implementata come un dizionario, che mappa un nome al relativo oggetto presente in memoria. Un namespace può essere di builtins, che contiene i nomi degli oggetti e funzioni predefinite in Python, e viene caricato all’avvio dell’interprete, globale, che contiene i nomi definiti nel modulo corrente, locale, che contiene i nomi definiti all’interno di una funzione e di oggetto, che contiene i nomi definiti all’interno di un oggetto. I concetti di namespace e di scoping sono correlati. Le regole di scoping sono le seguenti: Un nome globale ha scope in tutto il programma, un nome nonlocal ha scope nelle funzioni annidate, un nome locale è visibile all’interno della propria funzione. Per accedere a una variabile globale da una funzione si usa la keyword global, mentre per accedere a una variabile nonlocal da una funzione annidata si usa la keyword nonlocal.