

1 Paradigma OOP

Scienza o filosofia adattabile alla programmazione che supporta i concetti di:

1. Oggetto;
2. Classe;
3. Incapsulamento (information hiding);
4. Ereditarietà;
5. Polimorfismo;

1.1 Oggetto

Modello di un **entità reale** (concreta o astratta), caratterizzato da:

1. **Interfaccia**: servizi che può fornire;
2. **Attributi**: dati che mantiene al suo interno;

1.2 Classe

Modello che definisce **attributi ed operazioni comuni** ad un insieme di oggetti. Consente di definire un **nuovo tipo** per il linguaggio. Caratterizzata da:

1. **Identificatore**
2. Insieme di **attributi**;
3. **interfaccia**: insieme di servizi che gli oggetti della classe mettono a disposizione;
4. insieme di **metodi**: implementazione dei servizi;

1.3 Incapsulamento

Un oggetto rende **inaccessibili** all'esterno (incapsula) attributi e metodi al fine di garantire un **accesso controllato** ai dati. Serve a garantire: robustezza, indipendenza e riusabilità.

Come si applica:

1. Dichiарando attributi `private` **inaccessibili** al di fuori della classe;
2. Metodi `getter` e `setter` pubblici per l'accesso agli attributi;

1.3.1 Modificatori d'accesso

Regolano la visibilità e l'accesso ad un componente Java.

Access Modifier/Scope	Apply	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
Public	attributi, classi	YES	YES	YES	YES	YES
Protected	attributi	YES	YES	YES	YES	
Default	attributi, classi	YES	YES	YES		
Private	attributi	YES				

Un membro **protetto**, sarà accessibile all'interno dello **stesso package** (tramite l'operatore dot), ma verrà anche **ereditato** in tutte le **sottoclassi della classe in cui è definito**, anche se non appartenenti allo stesso package. L'uso di `protected` ha senso solamente quando si sfrutta l'ereditarietà.

1.3.2 Modificatore `static`

Permette di rendere un membro comune con l'intera classe. Il membro dichiarato `static` viene condiviso da tutte le istanze della classe:

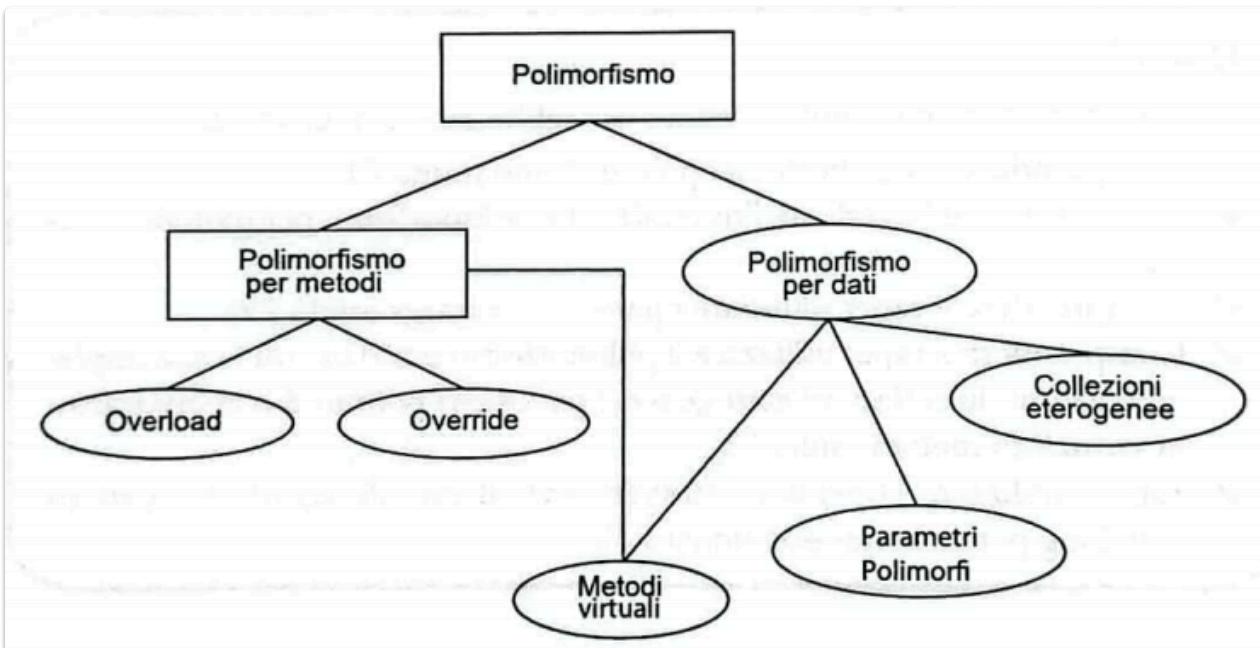
- **attributi di classe**: proprietà condivisa da tutte le istanze della classe;
 - **metodi di classe**: invocabile direttamente sul nome della classe senza dover istanziare un oggetto;
- Applicabile a: classi, membri.

1.4 Ereditarietà

Consente di mettere in **relazione** classi che presentano **caratteristiche comuni** andando a realizzare una **gerarchia padre-figlio**, superclasse-sottoclasse. La sottoclasse eredita (in base ai modificatori di accesso usati), gli attributi della superclasse, senza che quest'ultimi debbano essere ridefiniti nella sottoclasse.

1.5 Polimorfismo

Riferirsi con un unico termine ad entità differenti.



1.5.1 Polimorfismo per metodi

1.5.1.1 Overload & Override

1. **Overload**: metodi differenti possono avere lo stesso nome in quanto un metodo è univocamente determinato dalla sua **firma** ossia dalla coppia nome + parametri. In particolare la lista dei parametri ha 3 criteri di distinzione:

1. **tipo**: `somma(int a, int b) != somma(int a, float b)`
2. **numerico**: `somma(int a, int b) != somma(int a, int b, int c)`
3. **posizionale**: `somma(int a, float b) != somma(float a, int b)`

Non viene invece considerato l'identificatore del parametro: `somma(int a, int b) = somma(int c, float d)`

2. **Override**: le sottoclassi hanno la possibilità di ridefinire i metodi ereditati.

1.5.1.2 Classi polimorfe, metodi polimorfi ed interfacce

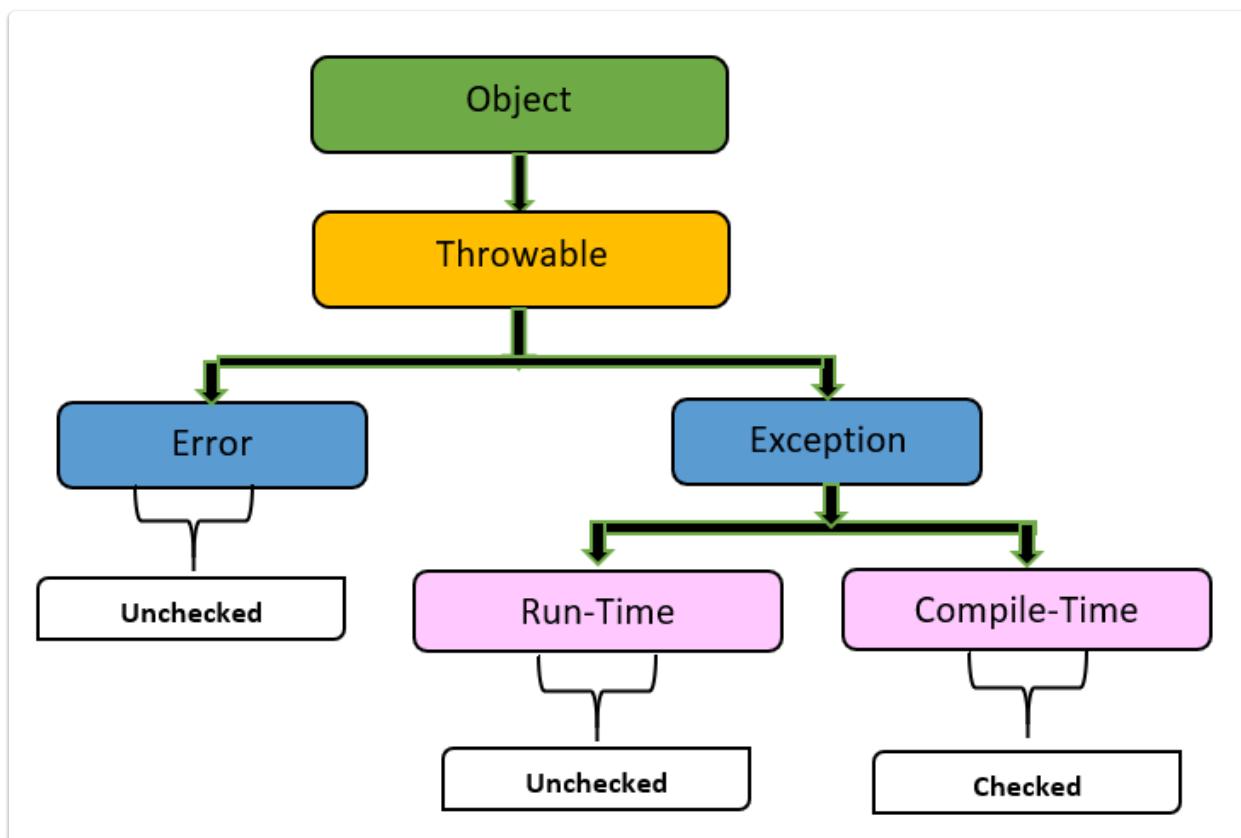
1. Classi polimorfe (**classi astratte**): classe progettata esclusivamente per essere ereditata. Non sono previste sue istanze;

2. Metodi polimorfi (**metodi astratti**): metodo che non implementa un proprio blocco di codice (non ha un corpo) e che deve essere necessariamente riscritto (Override) dalle classi che lo ereditano. Si possono definire solamente in classi astratte.
3. **Interfacce**: evoluzione del concetto di classe astratta che garantisce l'eredità multipla (una classe può implementare diverse interfacce);

2 Casting

- **Upcast**: riferimento base, oggetto derivato. Un oggetto **figlio** viene assegnato ad un **reference del padre**; In questo modo, essendo il reference la classe padre, si ignorano le funzionalità aggiuntive introdotte dalla classe figlio.
- **Downcast**: riferimento derivato, oggetto base. Un oggetto padre viene assegnato ad un reference del figlio. Genera un errore di compilazione se implicito in quanto l'oggetto padre che si cerca di istanziare, potrebbe non avere tutte le classi ed attributi presenti nella classe padre. Bisogna **esplicitare** manualmente il **cast!**

3 Eccezioni



Si possono categorizzare secondo 2 criteri:

- **Checked Exceptions**: estendono `Exception` e devono essere obbligatoriamente gestite dal programmatore;
- **Unchecked Exceptions**: estendono `RuntimeException` Non devono essere necessariamente gestite; Lo sviluppatore ha la possibilità di creare delle **eccezioni personalizzate**.

3.1 Gestione delle eccezioni

La gestione delle eccezioni avviene mediante il costrutto `try-catch` e le keywords `throws` e `throw`.

3.1.1 Costrutto `try-catch`

La keyword `catch` si occupa della cattura dell'eccezione specificata (si può usare polimorfismo e catturare direttamente `Exception` e invece di catturare singole eccezioni) o delle varie eccezioni specificate (multi-catch).

```

int a = 10;
int b = 0;

try{
    int c = a/b;
    // Multi Catch
} catch(ArithmetricException | NullPointerException ex){
    ex.printStackTrace();
} catch(Exception e){
    e.printStackTrace();
} finally{
    System.out.println("tutto ok");
}

```

Il costrutto prevede anche un'ulteriore clausola facoltativa: `finally`. Il codice contenuto al suo interno viene eseguito a prescindere che l'eccezione venga lanciata o meno. Può essere utilizzato per operazioni critiche che devono essere eseguite qualsiasi cosa accada, come per esempio la chiusura di uno Stream. Si possono creare anche costrutti `try - finally`.

3.1.1.1 Try with resources

Variante del costrutto `try` che consente la **chiusura automatica** degli oggetti che implementano l'interfaccia `Closeable` oppure `AutoCloseable`. Al costrutto vengono passati gli oggetti che devono essere chiusi automaticamente, i quali sono considerati implicitamente `final`.

```

try(BufferedWriter writer = new BufferedWriter(new FileWriter(filename))){
    writer.write(...);
}

```

3.1.2 Keyword `throw`

La parola chiave `throw` ci consente di **lanciare esplicitamente** delle eccezioni e pertanto viene particolarmente sfruttata con le **eccezioni personalizzate**, dal momento che, il compilatore non sa quando un'eccezione definita dall'utente deve essere lanciata.

```
throw new EccezioneDefinitaDallUtente();
```

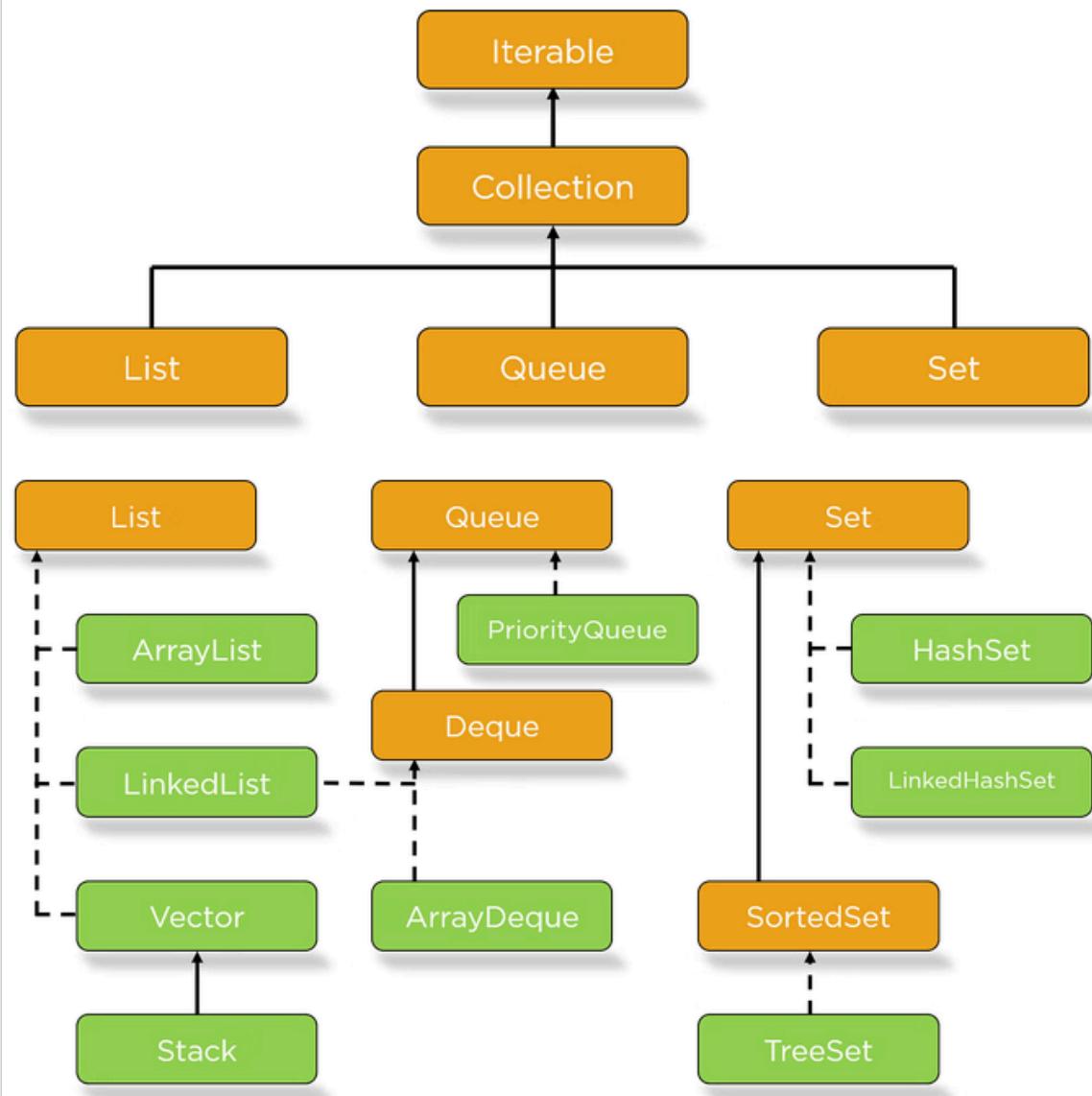
3.1.3 Keyword `throws`

Serve per informare il compilatore che il metodo potrebbe lanciare un'eccezione, la quale viene propagata al chiamante. In questo caso quindi l'eccezione deve essere gestita dal metodo chiamante.

4 Collections

	Ordine	Duplicati	Accesso tramite indice	Elementi nulli	Implementazioni
Set	NO	NO	NO	NO	HashSet
List	SI	SI	SI	SI	ArrayList, LinkedList
Queue	SI (FIFO)	SI	SI	NO	LinkedList
Deque	SI (FIFO/LIFO)	SI	SI	NO	?
Map	dipende	SI VALORI, NO CHIAVI	TRAMITE CHIAVE	?	HashMap

Java Collection Framework Hierarchy



4.1. Lists

Let's start with a list comparison table. Common operations for lists are adding and removing elements, accessing an element by index, traversal of the elements, and finding an element:

Lists Comparison Table	Add/remove element in the beginning	Add/remove element in the middle	Add/remove element in the end	Get i-th element (random access)	Find element	Traversal order
<i>ArrayList</i>	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$, $O(\log(n))$ if sorted	as inserted
<i>LinkedList</i>	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	as inserted

4.2. Sets

For sets, we're interested in adding and removing elements, traversal of elements, and finding an element:

Sets Comparison Table	Add element	Remove element	Find element	Traversal order
<i>HashSet</i>	amortized $O(1)$	amortized $O(1)$	$O(1)$	random, scattered by the hash function
<i>LinkedHashSet</i>	amortized $O(1)$	amortized $O(1)$	$O(1)$	as inserted
<i>TreeSet</i>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	sorted, according to elements comparison criterion
<i>EnumSet</i>	$O(1)$	$O(1)$	$O(1)$	according to the definition order of the enum values

4.3. Queues

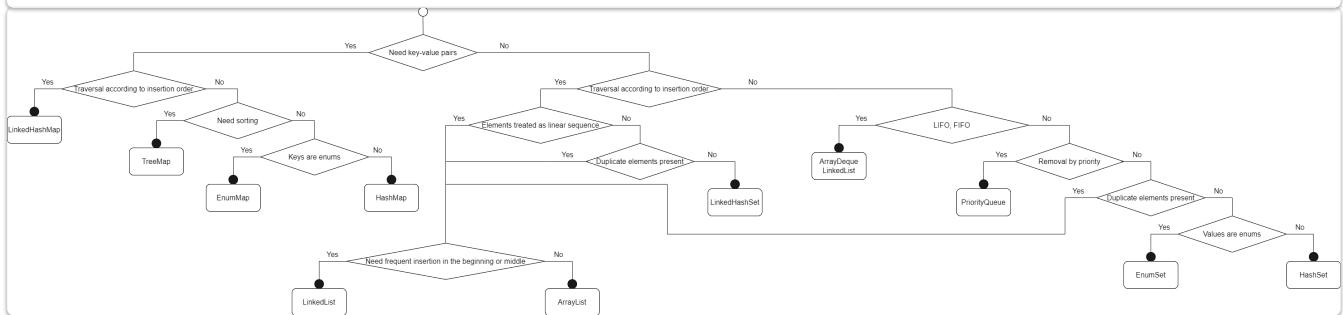
Queues can be divided into two groups:

1. *LinkedList*, *ArrayDeque* – *Queue* interface implementations can act as the stack, queue, and dequeue data structures. Generally, *ArrayDeque* is faster than *LinkedList*. Hence it's the default choice
2. *PriorityQueue* – *Queue* interface implementation backed by the binary heap data structure. Used for fast ($O(1)$) element retrieval, which has the highest priority. Addition and removal work in $O(\log(n))$ time

4.4. Maps

Similarly to sets, we consider the operations of adding and removing elements, traversal of elements, and finding an element for maps:

Maps Comparison Table	Add element	Remove element	Find element	Traversal order
<i>HashMap</i>	amortized $O(1)$	amortized $O(1)$	$O(1)$	random, scattered by the hash function
<i>LinkedHashMap</i>	amortized $O(1)$	amortized $O(1)$	$O(1)$	as inserted
<i>TreeMap</i>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	sorted, according to elements comparison criterion
<i>EnumMap</i>	$O(1)$	$O(1)$	$O(1)$	according to the definition order of the enum values

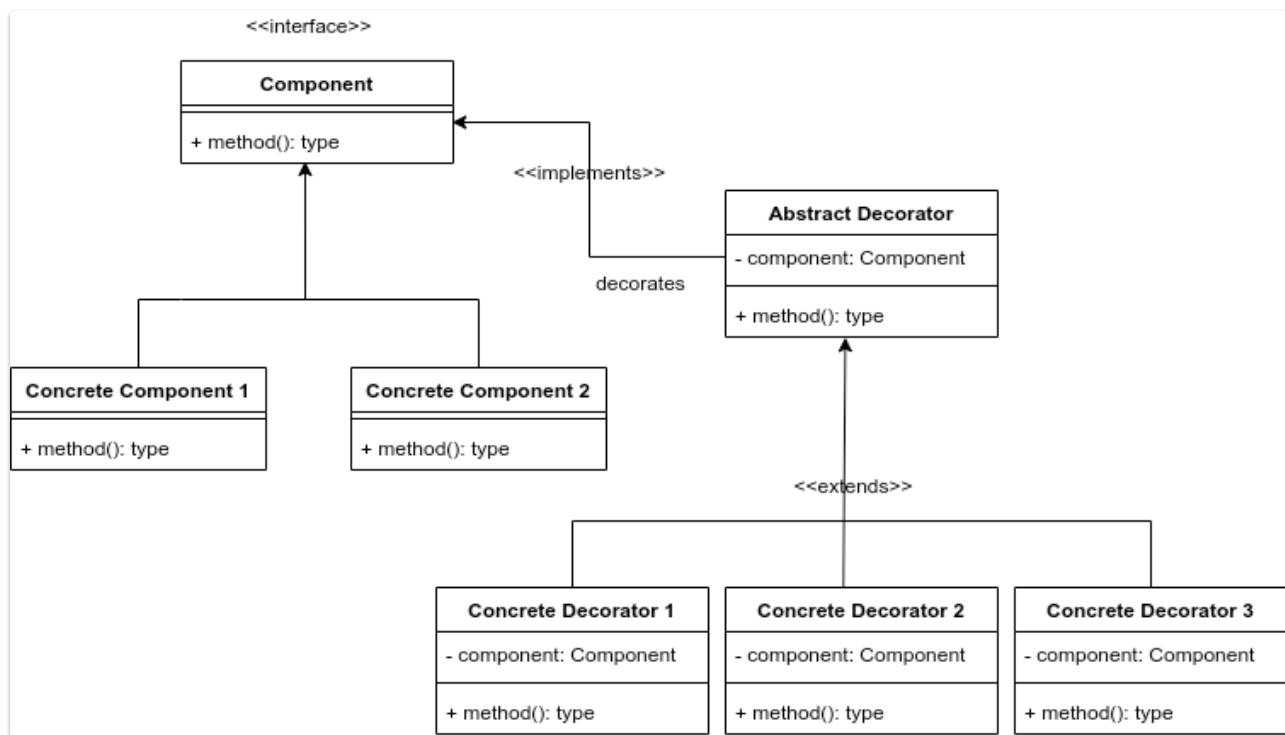


5 Pattern Architetturali

5.1 Decorator

Consente di implementare una **dinamica relazione** tra classi, in alternativa alla statica ereditarietà. Il pattern consente di aggiungere ad un oggetto nuove funzionalità a runtime, senza modificare le altre istanze della classe. Si tratta di un **pattern strutturale**.

- **Component**: interfaccia (o classe astratta) che definisce le **operazioni** astratte che devono implementare le sottoclassi;
- **ConcreteComponent**: classi concrete che implementano **Component** ;
- **Decorator**: estensione di **Component** (può essere astratta); deve obbligare le sottoclassi ad implementare **Component** e referenziare un **Component** con un'aggregazione;
- **ConcreteDecorator**: implementazione di **Decorator** , implementa **Component** e le sue operazioni e mantiene un riferimento a **Component** .



Quando occorre modificare una classe si ricorre all'[ereditarietà](#) che però applica le stesse modifiche a tutte le istanze della classe derivata.

Dovendo modificare una singola istanza, magari a [runtime](#), la soluzione basata sull'ereditarietà si dimostra [inefficace](#). Inoltre ogni volta che occorre combinare due o più caratteristiche presenti in classi differenti bisogna creare una [nuova sottoclasse](#) e ben presto la [gerarchia](#) diventa [ingestibile](#).

5.2 Singleton

Caso d'uso: Una classe deve essere [istanziata una sola volta](#) e tutti gli utilizzatori della classe devono utilizzare sempre la stessa istanza.

Si può applicare il [pattern Singleton](#):

1. Costruttore privato
2. Variabile privata e statica dello stesso tipo della classe (`instance`)
3. Metodo statico pubblico (`getInstance`) che restituisce sempre la stessa istanza della classe

```

public class SingletonExample {
    private static SingletonExample instance;

    private SingletonExample(){}

    public static SingletonExample getInstance(){
        if (instance == null){
            instance = new SingletonExample();
        }
        return instance;
    }
}
  
```

L'unica istanza può essere recuperata dalle altre classi usando la sintassi:

```
SingletonExample unicaIstanza = singletonExample.getInstance();
```

La compilazione di un file `.java` contenente classi innestate porta alla generazione di 2 file separati, uno per la classe esterna ed uno per quella interna.

6 Enumerazioni e Record

Enumerazione e Record sono particolari tipologie di classi che godono di una sintassi semplificata e sintetica.

- **Enumerazione:** struttura che permette di definire un insieme limitato e ben definito di valori *costanti*;
- **Record:** struttura che permette di rappresentare dati *immutabili*;

Sia i record che le enumerazioni vengono trasformate in classi (che estendono `java.lang.Enum` e `java.lang.Record`) durante la compilazione.

Possono implementare interfacce.

6.1 Ereditarietà e Record

Dato che i Record sono contenitori di dati immutabili, non supportano il principio dell'ereditarietà. Un record **non può estendere** (in quanto implicitamente dichiarato `final`), né essere esteso da altre classi.

Un record deve essere immutabile e l'ereditarietà non è compatibile con l'immutabilità.

6.2 Record: Costruttore Canonico e Compatto

Nei Record è possibile utilizzare solamente variabili statiche. Il costruttore di default definisce automaticamente come parametri le variabili definite nell'header del record per cui è possibile omettere la lista dei parametri.

≡ Costruttore Canonico

```
public record Foto(String formato, boolean aColori){  
    public Foto(String formato, boolean aColori) {  
        IllegalArgumentException("Descrizione del formato troppo breve");  
        this.formato = formato;  
        this.aColori = aColori  
    }  
}
```

Utilizzando il costruttore canonico è necessario inizializzare manualmente le variabili.

≡ Costruttore Canonico Compatto

```
public Foto{  
    if(formato.length() < 5) throw new IllegalArgumentException(  
        "Descrizione formato troppo breve");  
}
```

Utilizzando il costruttore compatto, l'inizializzazione delle variabili è **automatica**.

7 Classi Innestate e classi interne

Java offre la possibilità di dichiarare un tipo all'interno di un altro tipo. Si parla in questi casi, di tipi **innestati**.

Lo stesso discorso vale per le classi, le quali assumono un nome differente in base al contesto:

- **Classi interne:** non statiche. Hanno accesso a tutti i membri della classe esterna, anche se privati. Vi fanno parte le **classi locali** e le **classi anonime**.

- **Classi innestate:** statiche. Ha accesso solamente ai membri statici della classe esterna.

Essendo a tutti gli effetti dei **membri** della classe esterna, le classi interne/innestate possono adottare tutti i modificatori di accesso: private, public, protected, package... a differenza delle classi esterne che possono essere dichiarata solamente public o package.

Static nested class vs inner class

Static nested class

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Inner class

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Servono a:

- **Raggruppare** logicamente le classi che sono usate in un unico punto
- Aumentare l'**incapsulamento**
- Rendere il codice più **leggibile e manutenibile**

Dal punto di vista '**object oriented**', l'utilizzo dei tipi innestati non è raccomandato.

Le classi interne/innestate consentono di scrivere meno codice e al giorno d'oggi sono state sostituite dalle **espressioni Lambda**.

7.1 Static Nested Classes

Classi statiche dichiarate all'**interno di altre classi**. La classe interna può accedere ai membri statici della classe esterna solamente tramite un reference del tipo:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass()
```

7.2 Inner Classes

Classi **NON statiche** dichiarate all'interno di altre classi. Ha diretto accesso ai membri della classe esterna e può esistere solamente se esiste la classe esterna. Viene istanziata mediante la creazione di un **oggetto interno**:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Local and anonymous classes

Nested class



7.2.1 Local Classes

Classi definite all'interno di un **blocco**, solitamente nel corpo di un metodo. La classe ha accesso a tutti i membri della classe esterna e le variabili locali dichiarate `final`.

7.2.2 Anonymous Classes

Si tratta di classi locali che **non hanno un nome**. Vengono usate solamente quando si ha bisogno di usare la classe solamente **1 volta**.

Syntax of anonymous classes

- The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

Parentheses that contain the arguments to a constructor

Class declaration body

```
name of an interface to implement or a class to extend  
new operator  
HelloWorld frenchGreeting = new HelloWorld() {  
    String name = "tout le monde";  
    public void greet() {  
        greetSomeone("tout le monde");  
    }  
    public void greetSomeone(String someone) {  
        name = someone;  
        System.out.println("Salut " + name);  
    }  
};
```

Una classe anonima viene **definita contemporaneamente ad una sua istanza**, utilizzando la sintassi:

```
SuperTipo identificatore = new SuperTipo(){  
    [ridefinizione della classe]  
}
```

La sintassi assomiglia a quella di un [costruttore](#) con la differenza che è presente un corpo. Inoltre, dato che non ha un nome, la sua istanza può essere puntata solamente da un [reference](#) del tipo della [superclasse](#), usando il [polimorfismo per dati](#). Il discorso è analogo per il [costruttore](#): la classe anonima viene istanziata invocando il costruttore della superclasse.

Tuttavia è possibile dichiarare una classe anonima sfruttando anche un'[interfaccia](#) (che per definizione non ha un costruttore), andando ad istanziare direttamente l'interfaccia. In questo caso si sta utilizzando un [costruttore virtuale](#) che ha l'obiettivo di istanziare una classe e non un'interfaccia.

```
public interface Volante {
    void plana();
    void decolla();
    void atterra();
}

public class TestVolante {
    public static void main(String args[]){
        Volante ufo = new Volante(){
            @Override
            public void decolla() {
                System.out.println("Un oggetto sta decollando");
            }
            @Override
            public void plana(){
                System.out.println("Un oggetto sta planando");
            }
            @Override
            public void atterra(){
                System.out.println("Un oggetto sta atterrando");
            }
        };
        ufo.decolla();
        ufo.plana();
        ufo.atterra();
    }
}
```

Da questo esempio si evince che una classe anonima viene sempre istanziata con lo scopo di fare [override](#) di uno o più metodi della classe che estende. Nonostante la classe interna non abbia reference, è possibile [inserire un nuovo metodo](#) nella classe locale, e sfruttare la keyword `var`, per ottenere un riferimento il cui tipo viene deciso direttamente dal compilatore.

```
public class VarAnonymousTest {
    public static void main(String args[]){
        var testObject = new Object(){
            String name = "Può essere usato con var!";
            void test(String test){
                System.out.println(test);
            }
        };
    }
}
```

Una classe anonima può essere istanziata una sola volta e pertanto può essere vista come una particolare implementazione del pattern [Singleton](#).

Tornano utili in quelle situazioni in cui è necessario dichiararle e passarle come parametro ad un metodo.

8 Espressioni Lambda

Un'espressione lambda è una **funzione anonima**, ossia una funzione che ha un corpo ma **non un nome** e definita pertanto nel punto in cui viene utilizzata.

Un'espressione lambda consente di creare un'istanza di una classe anonima che implementa un'**interfaccia SAM** (*Single Abstract Method*), ovvero dotata di 1 solo metodo astratto.

Si utilizza la seguente sintassi: `([lista_parametri]) -> {blocco_codice}`.

Le espressioni Lambda ci consentono di scrivere meno codice e quindi **ridurne la verbosità**. Un esempio riguarda l'avvio di un thread

Esempio thread - Classe anonima

```
new Thread(new Runnable(){  
    @Override  
    public void run() {  
        System.out.println("Classe Anonima");  
    }  
})
```

La classe anonima può essere sostituita da un'espressione Lambda:

Esempio thread - Lambda

```
new Thread(() -> System.out.println("Funzione Anonima - Lambda")).start();
```

Funziona in quanto il costruttore di `Thread` accetta un'istanza di `Runnable` che è un'interfaccia funzionale, la quale definisce come metodo SAM, `run()`. Con le lambda, il compilatore deduce automaticamente che stiamo cercando di **sovrascrivere** il metodo SAM `run()` in quanto l'interfaccia `Runnable` è funzionale (e quindi banalmente ha solo `run()` come metodo astratto!).

Sostanzialmente le espressioni Lambda consentono di **sovrascrivere al volo il codice implementato dalle interfacce funzionali**. Sono equivalenti alle classi anonime, ma sono più compatte... inoltre sono l'ideale per passare codice come parametro ad un metodo.

Esempio - passaggio codice come parametro

Consideriamo una **videoteca** composta dalle seguenti classi:

```
public class Film {  
    private String nome, genere, mediaRecensioni;  
    public Film(String nome, String genere, int mediaRecensioni){  
        this.nome = nome;  
        this.genere = genere;  
        this.mediaRecensioni = mediaRecensioni;  
    }  
}
```

Un'**interfaccia funzionale** (da implementare al volo con le lambda):

```
@FunctionalInterface  
public interface FiltroFilm {  
    boolean filtra(Film film)  
}
```

Una classe `Videoteca` con un metodo `getFilmFiltrati()`:

```
public List<\Film> getFilmFiltrati(FiltroFilm filtroFilm){  
    List<\Film> filmFiltrati = new ArrayList<>();  
    for(Film film : films){  
        if(filtroFilm.filtra(film)){  
            filmFiltrati.add(film);  
        }  
    }  
}
```

Il metodo in questione sfrutta l'interfaccia funzionale `FiltroFilm`, in particolare il metodo `filtra()` che può essere **riscritto al volo mediante un'espressione Lambda**:

```
Videoteca videoteca = new Videoteca();  
List<\Film> filmBelli = videoteca.getFilmFiltrati(  
    f -> f.getMediaRecensioni() > 3;  
)  
List<\Film> filmFantascienza = videoteca.getFilmFiltrati(f-  
>f.getGenere().equals("FANTASCIENZA"));
```

Analogamente con le **classi anonime**:

```
Videoteca videoteca = new Videoteca();  
List<\Film> filmBelli = videoteca.getFilmFiltrati(new FiltroFilm(){  
    @Override  
    public boolean filtra(Film film) {  
        return film.getMediaRecensioni() > 3;  
    }  
});  
List<\Film> filmFantascienza = videoteca.getFilmFiltrati(new FiltroFilm(){  
    @Override  
    public boolean filtra(Film film){  
        return "Fantascienza".equals(film.getGenere());  
    }  
})
```

Le espressioni Lambda risultano molto più **sintetiche e leggibili**.

Inoltre le lambda consentono di **assegnare blocchi di codice** a delle variabili (utilizzando un'interfaccia come tipo), le quali possono essere poi passate come parametri a dei metodi (andando effettivamente a passare un blocco di codice come parametro):

```
FiltroFilm lambda = () -> {  
    System.out.println("Riga 1");  
    System.out.println("Riga 2");
```

```
    System.out.println("Riga 3");
}
```

8.1 Gestione delle eccezioni

Le **checked exception** sono problematiche per le espressioni Lambda in quanto **non è possibile** utilizzare la clausola `throws`. È possibile:

- utilizzare un costrutto `try-catch`. Ma visto che si viola il **principio di semplicità** con cui sono state pensate le Lambda, conviene usare una **classe anonima**;
- modificare l'**interfaccia funzionale** in modo che il metodo SAM presenti la clausola `throws`

Ciò va contro il principio di compattezza con cui sono state pensate per cui in questo caso, probabilmente, è conveniente utilizzare una classe anonima.

8.2 Deduzione dei tipi: utilizzo con `var`

Con java 10 è stata introdotta la deduzione automatica dei tipi per le variabili locali: il compilatore è in grado di dedurre automaticamente il tipo di un oggetto basandosi sulla parte destra dell'assegnazione (RHS) e pertanto è possibile utilizzare un reference di tipo "generico" `var`.

Per le espressioni lambda, il compilatore riesce a dedurre i dettagli per la definizione dell'espressione stessa proprio dalla parte sinistra dell'assegnazione (LHS), pertanto non è possibile utilizzare l'assegnazione automatica con `var`.

9 Reference a metodi

Sintassi funzionalmente **equivalente alle espressioni lambda** (ma più compatta) che fa uso di metodi esistenti.

Esempio

Se per esempio avessimo una classe di filtri:

```
public class Filtri {
    public static boolean isFilmFantascienza(Film film) {
        return "Fantascienza".equals(film.getGenere());
    }
}
```

È possibile utilizzare direttamente l'implementazione di questi metodi, usando la sintassi dei reference a metodi:

```
List<\Film> filmFantascienza = videoteca.getFilmFiltrati(Filtri::isFilmFantascienza);
```

Non stiamo chiamando semplicemente il metodo `isFilmFantascienza`... il metodo infatti ritorna un booleano e quindi avrebbe generato un errore di compilazione, dato che `getFilmFiltrati` si aspetta un oggetto di tipo `FiltroFilm`.

Stiamo **sostituendo un'istanza di un'interfaccia funzionale**, passandogli il nome di un metodo già definito. La differenza con le espressioni lambda sta nel fatto che i **reference utilizzano metodi già esistenti**... le espressioni lambda creano al volo la loro implementazione.

I references sono da preferire alle espressioni lambda nel caso in cui utilizzassimo più volte la chiamata al metodo. La peculiarità della lambda infatti è quella di creare codice al volo, il che contraddice il fatto di

riutilizzare più volte lo stesso metodo.

9.1 Tipologie di reference

9.1.1 Reference a metodo statico

```
NomeTipo::nomeMetodoStatico
```

È l'esempio mostrato sopra con la classe `Filtri`. Questo reference è del tutto equivalente ad un'espressione lambda.

9.1.2 Reference a metodo d'istanza

```
nomeOggetto::nomeMetodoIstanza
```

In questo caso il metodo non è statico e quindi occorre istanziare un oggetto di quella classe per utilizzare il metodo:

```
public class OrdinamentoFilm {  
    public int ordinaMediaRecensioni(Film f1, Film f2){  
        Integer mediaF1 = new Integer(f1.getMediaRecensioni());  
        Integer mediaF2 = new Integer(f2.getMediaRecensioni());  
        return mediaF1.compareTo(mediaF2);  
    }  
}
```

Possiamo riordinare i film utilizzando un reference:

```
List<Film> films = videoteca.getFilms();  
OrdinamentoFilm ordinamentoFilm = new OrdinamentoFilm();  
Collections.sort(films, ordinamentoFilm::ordinaMediaRecensioni);
```

Funziona perché il metodo `sort()` vuole in input il blocco di codice contenente l'implementazione del metodo `compare()` della classe `Comparator`.

9.1.3 Reference a metodo d'istanza di un certo tipo

```
NomeTipo::nomeMetodoIstanza
```

Se è presente un metodo d'istanza da cui il compilatore può dedurre parametri di input ed output, è possibile utilizzare il nome del tipo invece del nome dell'oggetto:

```
List<String> filmNames = new ArrayList<>();  
for(Film film : films){  
    filmNames.add(film.getNome());  
}  
Collections.sort(filmNames, String::compareToIgnoreCase);
```

9.1.4 Reference a costruttore

```
NomeTipo::new
```

Dato che un costruttore è un metodo che ritorna un oggetto della sua classe, è possibile referenziare anche un costruttore.

Supponiamo di avere la seguente classe ed interfaccia funzionale:

```
public class Chitarra{  
    private String marca;  
    public Chitarra (String marca){  
        this.marca = marca;  
        System.out.println("Creata chitarra: " + marca);  
    }  
}
```

```
@FunctionalInterface  
public interface FabbricaChitarra {  
    Chitarra getChitarra(String marca);  
}
```

È possibile scrivere:

```
FabbricaChitarra fabbricaChitarra = Chitarra::new;  
Chitarra chitarra1 = fabbricaChitarra.getChitarra("Fender");
```

Con la prima istruzione si è usato il codice del costruttore di `Chitarra` come implementazione del metodo SAM `getChitarra` dell'interfaccia funzionale `FabbricaChitarra`.

9.2 Interfacce funzionali di `java.util.function`

Il package `java.util.function` mette a disposizione delle interfacce funzionali già pronte:

9.2.1 Predicate

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

Utile per effettuare test che restituiscono un **booleano**. Viene utilizzata per **filtrare oggetti**. Mette a disposizione i metodi `or` e `and` per concatenare i filtri;

:≡ Example

```
// Restituisce true se è pari  
Predicate<\Integer> isEven = n -> n % 2 == 0;  
System.out.println(isEven.test(4)); // true  
System.out.println(isEven.test(5)); // false
```

9.2.2 Consumer

```
@FunctionalInterface  
public interface Consumer<T> {
```

```
    void accept(T t);  
}
```

Utile per **aggiornare lo stato** dell'oggetto che viene passato al suo metodo SAM.

:≡ Example

```
Consumer<\String> stampa = s -> System.out.println(s); // consuma stringa stampandola  
stampa.accept("Ciao");
```

9.2.3 Supplier

```
@FunctionalInterface  
public interface Supplier<T> {  
    T get();  
}
```

Utile per la **gestione delle factory** in quanto il metodo SAM restituisce un'istanza del tipo di parametro dichiarato. Si usa quindi per **creare oggetti**.

:≡ Example

```
Supplier<Chitarra> nuovaChitarra = () -> new Chitarra("Fender");  
Chitarra miaChitarra = nuovaChitarra.get();  
System.out.println(miaChitarra.getMarca()); // Stampa: Fender`
```

9.2.4 Function e BiFunction

```
@FunctionalInterface  
public interface Function<T,R> {  
    R apply(T t);  
}
```

Astrae il concetto di **funzione**, dove è presente un input (`T`) ed un output (`R`).

:≡ Example

```
Function<String, Integer> stringToLength = s -> s.length();  
int len = stringToLength.apply("Obsidian"); // len = 8
```

`BiFunction` invece prende in input 2 parametri (`T`, `U`) e restituisce un unico output (`R`):

```
@FunctionalInterface  
public interface BiFunction<T,U,R> {  
    R apply(T t, U u);  
}
```

:≡ Example

Contents

```
BiFunction<String, String, String> concat = (a, b) -> a + "-" + b;  
String out = concat.apply("Java", "Lambda"); // out = "Java-Lambda"
```

Esistono le versioni `Bi-*` anche per le altre interfacce funzionali del package! Eventuali interfacce che prendono in input più valori devono essere definite manualmente.

9.2.5 UnaryOperator

```
@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {  
}
```

È un'estensione dell'interfaccia `Function` dalla quale eredita il metodo `apply`.

Astrae un'operazione che viene effettuata su un **singolo parametro** e produce un risultato dello stesso tipo. Si adatta bene ad operazioni di trasformazione oppure per combinare espressioni lambda che devono essere eseguite in un solo colpo.

Example

```
UnaryOperator<\Integer> raddoppia = n-> n * 2;  
System.out.println(raddoppia.apply(4));
```

9.2.6 Tabella Riassuntiva

Interfaccia	Scopo	Metodo principale
Runnable	Esegue del codice senza parametri e senza valore di ritorno	void run()
Callable<V>	Esegue codice che restituisce un valore e può lanciare eccezioni	V call() throws Exception
Predicate<T>	Testa un valore e restituisce un boolean	boolean test(T t)
Consumer<T>	Consuma (esegue un azione sul valore) un valore senza restituirne altri	void accept(T t)
Supplier<T>	Fornisce un valore senza prendere nulla in input	T get()
Function<T, R>	Trasforma un valore T in un valore R	R apply(T t)
BiFunction<T, U, R>	Trasforma due valori (T, U) in un valore R	R apply(T t, U u)
UnaryOperator<T>	Trasforma un valore T in un altro valore dello stesso tipo T	T apply(T t)

In JavaFX, la `Function` è chiamata `Callback`.

```
@FunctionalInterface  
public interface Callback<P, R> {
```

```
R call(P param);  
}
```

10 Caratteristiche avanzate

10.1 var e nuovo switch

A partire da Java 10 è stata introdotta la keyword `var` che consente di definire variabili **senza specificarne esplicitamente il tipo**: il compilatore riesce a dedurre automaticamente il tipo dall'RHS (*Right Hand Side*), ovvero leggendo la sua [inizializzazione](#).

Example

```
var libro1 = new LibroJava();
```

Si parla di **tipo manifesto** quando il RHS è costituito dal **costruttore**... ciò ci consente di capire al volo il tipo della variabile. Non è buona pratica usare `var` con i metodi getter (tipo non manifesto).

È applicabile solamente per variabili locali. Non funziona con variabili d'istanza, parametri di metodi, variabili di ritorno, array, dichiarazioni multiple...

10.2 Nuovo Switch (switch expression)

Java supporta nativamente il costrutto 2 switch in 2 forme diverse: il classico switch statement, utilizzabile come statement condizionale, e il nuovo switch expression, un'espressione effettiva che ritorna un valore.

Il nuovo case si pone come obiettivo quello di evitare o ridurre il fall through: nello switch tradizionale, ogni statement deve essere chiuso da un `break` altrimenti saranno eseguiti tutte le istruzioni successive finché non si arriverà ad un'istruzione di `break`.

switch : fall through con enum

```
public class SeasonSwitchEnumTest {  
    public static void main(String args[]) {  
        Month month = Month.APRIL;  
        String season;  
        switch (month) {  
            case DECEMBER:  
            case JANUARY:  
            case FEBRUARY:  
                season = "winter";  
                break;  
            case MARCH:  
            case APRIL:  
            case MAY:  
                season = "spring";  
                break;  
            case JUNE:  
            case JULY:  
            case AUGUST:  
                season = "summer";  
                break;  
            case SEPTEMBER:  
            case OCTOBER:  
            case NOVEMBER:  
                season = "autumn";  
                break;  
            default:  
                season = "not identifiable";  
                break;  
        }  
        System.out.println("The season is "  
                           + season);  
    }  
}
```

Il nuovo switch expression può prevedere l'utilizzo dell'operatore freccia `->` (si può ancora usare `:`) e la possibilità di dichiarare più label separate da virgolette.

Il nuovo `switch`: notazione freccia

```
public class SeasonSwitchStatementArrowEnumTest {  
    public static void main(String args[]) {  
        Month month = Month.APRIL;  
        String season = null;  
        switch (month) {  
            case DECEMBER, JANUARY, FEBRUARY -> season = "winter";  
            case MARCH, APRIL, MAY -> season = "spring";  
            case JUNE, JULY, AUGUST -> season = "summer";  
            case SEPTEMBER, OCTOBER, NOVEMBER -> season = "autumn";  
        }  
        System.out.println("The season is " + season);  
    }  
}
```

Non è più necessario inserire `break`. La peculiarità del costrutto è tuttavia quella di poter ritornare un valore... se si usa la notazione freccia, il valore viene ritornato automaticamente a patto che non sia presente in un blocco di istruzioni.

```
public class SeasonSwitchExpressionEnumTest {  
    public static void main(String args[]) {  
        Month month = Month.APRIL;  
        String season = switch (month) {  
            case DECEMBER, JANUARY, FEBRUARY -> "winter";  
            case MARCH, APRIL, MAY -> "spring";  
            case JUNE, JULY, AUGUST -> "summer";  
            case SEPTEMBER, OCTOBER, NOVEMBER -> "autumn";  
        };  
        System.out.println("The season is " + season);  
    }  
}
```

Claudio De Sio Cesari: NJ-004

10.2.1 Keyword `yield`

Qualora a destra dell'operatore `->` fosse presente un blocco di istruzioni delimitato da graffe, è possibile specificare il valore di ritorno mediante la keyword `yield`.

```

String season = switch (month) {
    case DECEMBER, JANUARY, FEBRUARY -> {
        String value = "winter";
        yield value;
    }
    case MARCH, APRIL, MAY -> "spring";
    case JUNE, JULY, AUGUST -> {
        String value = "summer";
        yield value;
    }
    case SEPTEMBER, OCTOBER, NOVEMBER -> "autumn";
};

```

10.2.2 Freccia vs due punti

Il nuovo switch supporta ancora la sintassi tramite `:`, la quale consente di effettuare fall through. Non è possibile mischiare le notazioni.

```

String season = switch(month) {
    case DECEMBER:
    case JANUARY:
    case FEBRUARY: yield "winter";
    case MARCH, APRIL, MAY: yield "spring";
    case JUNE, JULY, AUGUST: yield "summer";
    case SEPTEMBER, OCTOBER, NOVEMBER: yield "autumn";
};

```

10.3 Wildcard

Le **wildcards** nei generics Java permettono di dichiarare tipi generici più flessibili e generalizzati, usando il carattere `?`.

Example

```
List<\?> listaGenerica;
```

10.3.1 Wildcard capture

La **wildcard capture** (cattura della wildcard) è una caratteristica avanzata dei generics in Java che permette al compilatore di "trasformare" una wildcard (`?`) in una variabile di tipo concreto, per poter gestire certe operazioni che altrimenti sarebbero proibite (soprattutto la scrittura/modifica di collezioni generiche con wildcard).

Quando dici chi un metodo che riceve una `List<\?>` (o un altro tipo parametrico con wildcard), il compilatore *non sa* quale sia il tipo effettivo degli elementi: può essere qualsiasi tipo. Questo limita fortemente cosa puoi fare: in particolare, puoi solo leggere, non puoi scrivere (se non un valore `null`).

11 Input/Output

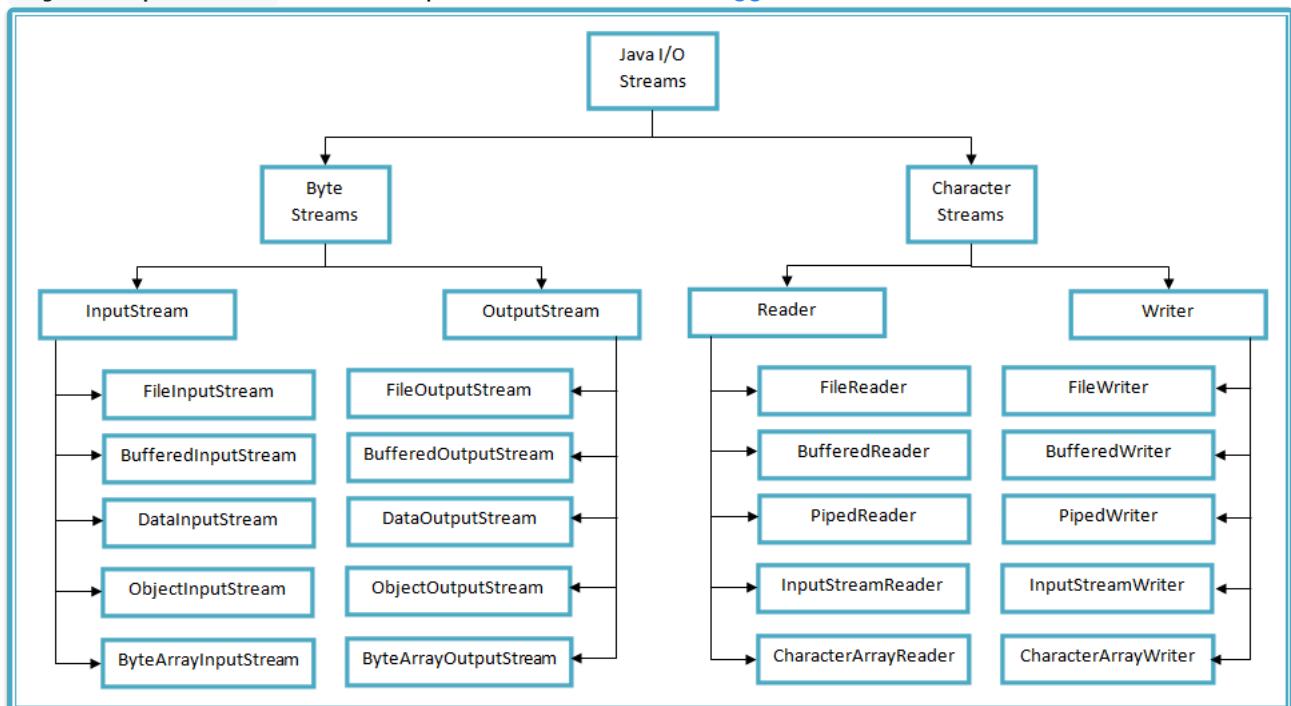
Gestito dal package `java.io` secondo la gerarchia definita dal Pattern [Decorator](#):

Le superclassi `Reader/Writer` e `InputStream/OutputStream` svolgono il ruolo di `Component`. Le sottoclassi svolgono il ruolo di:

1. `ConcreteComponent`: possono leggere/scrivere direttamente, agganciandosi ad una fonte. Sono anche detti [node stream](#);
2. `ConcreteDecorator`: servono a facilitare la lettura/scrittura o migliorarne le prestazioni. Non si possono istanziare e sono detti [processing stream](#);

Le classi "stream" sono divise in 2 gerarchie:

- **Character Stream**: `Reader` e `Writer` obbligano le sottoclassi a leggere e scrivere dividendo i dati in 16 bit (sono compatibili con il tipo `char`).
- **Byte Stream**: `InputStream` ed `OutputStream` obbligano le sottoclassi a leggere e scrivere dati in parti da 8 bit... sono pertanto adatte alla lettura/scrittura di informazioni non testuali. `ObjectInputStream` ed `ObjectOutputStream` sono usate per la [serializzazione di oggetti](#).



11.1 Decorator Principali

11.1.1 Character Stream

1. `BufferedReader/BufferedWriter`: aggiunge un buffer per migliorare le prestazioni IO diminuendo gli accessi al disco;
2. `PrintWriter`: Consente di scrivere testo formattato (per esempio con `println`)

11.1.2 Byte Stream

1. `BufferedInputStream/BufferedOutputStream`: aggiunge un buffer per migliorare le prestazioni IO diminuendo gli accessi al disco;
2. `DataInputStream/DataOutputStream`: consente di leggere e scrivere tipi primitivi (`int, double, boolean...`);
3. `ObjectInputStream/ObjectOutputStream`: consente di leggere e scrivere oggetti Java. Si usa per la [serializzazione](#).

3. Quale Scegliere? (Mappa di Decisione)

Scenario	Input	Output
File binari semplici	FileInputStream	FileOutputStream
File binari grandi	BufferedInputStream	BufferedOutputStream
Dati primitivi (int, double)	DataInputStream	DataOutputStream
Oggetti Java	ObjectInputStream	ObjectOutputStream
File di testo semplice	FileReader	FileWriter
File di testo grande	BufferedReader	BufferedWriter
Testo formattato	-	PrintWriter

11.2 Serializzazione di Oggetti

Processo per rendere **persistente** un oggetto in Java, ossia far sopravvivere l'oggetto anche dopo la terminazione del programma, salvandone lo **stato** (variabili e relativi valori) in un file. Durante la deserializzazione l'oggetto presenterà lo stato salvato durante la serializzazione.

Si usano le classi `FileOutputStream` e `ObjectOutputStream` che implementano il metodo `writeObject`. Per essere serializzabile, un oggetto deve implementare l'interfaccia `Serializable`. Le stesse variabili di istanza devono necessariamente essere serializzabili (per esempio le variabili di tipo `Thread` non lo sono), altrimenti verrebbe lanciata un'eccezione `NotSerializableException`.

Qualora fosse presente una variabile di istanza non serializzabile, è possibile usare il modificatore `transient` per avvertire la `JVM` che la variabile non deve essere serializzata. Le variabili `static` sono implicitamente anche `transient`.

11.3 Classe Scanner

La classe `Scanner` **semplifica la lettura** di sorgenti di input, andando automaticamente a suddividere l'input in diversi token a seconda del delimitatore passato come parametro (di default considera lo spazio).

```
String expression = "3,13+2,14+0,16";
Readable source = new StringReader(expression);
Scanner scanner = new Scanner(source);
scanner.useDelimiter("\\+");
while(scanner.hasNextDouble()){
    sum += scanner.nextDouble();
}
System.out.print(expression + "=" + sum);
scanner.close();
```

`hasNextDouble` è una specializzazione del metodo generico `hasNext` e, oltre a controllare la presenza di altro testo, verifica che esso sia di tipo `Double`.

Può essere visto come un ulteriore decorator. I metodi sono identici agli `Iterator`.

12 Thread e sincronizzazione

Un `Thread` è un **processore virtuale** che esegue codice su determinati dati. In Java esistono 2 tecniche per la creazione di un thread:

- Creare una classe che estenda `Thread` e sovrascrive il metodo `run()`

- Definire una classe che implementa l'interfaccia `Runnable`

Per la creazione di un thread occorre:

4. creare un oggetto `Thread`
5. passare all'oggetto un'istanza di una classe che implementa `Runnable`. La classe contiene il codice che il thread dovrà eseguire, all'interno del metodo `run()`.
6. Invocare il metodo `start()` dell'oggetto `Thread`. L'invocazione di tale metodo consente alla JVM di allocare la memoria necessaria ed inizializzare il thread e di chiamare il metodo `run()` che rende il thread eseguibile.

12.1 Java Monitor

L'accesso simultaneo di più thread alla stessa area di memoria richiede la **sincronizzazione** dei loro **metodi di accesso** (problema della sezione critica) in modo tale da garantire l'accesso ai dati ad un solo thread. Java utilizza i **monitor** come meccanismo di concorrenza.

Per la sincronizzazione si usa la keyword `synchronized`, applicabile a metodi e blocchi di codice; la quale garantisce l'**atomicità** dell'esecuzione di una determinata operazione. Ogni oggetto ha associato un singolo lock (**mutex**) e, per invocare un metodo `synchronized`, bisogna **possedere il lock** dell'oggetto. Se il lock è **posseduto da un altro thread**, il thread che ha effettuato la chiamata viene spostato in una lista di attesa detta "**entry set**", la quale contiene l'insieme dei thread che attendono la disponibilità del lock. Analogamente è presente un "**wait set**", una lista contenente l'insieme dei thread bloccati, in attesa che si verifichi una determinata condizione che consenta la loro corretta esecuzione.

La sincronizzazione dei singoli blocchi di codice consente di diminuire il "**lock scope**", ossia il tempo che un lock viene posseduto da un oggetto. Dal momento che il lock viene rilasciato solamente al termine del metodo sincronizzato, è **buona norma sincronizzare blocchi** di codice quando il metodo da sincronizzare è molto lungo. La sincronizzazione mediante `synchronized` si limita a sincronizzare il codice ma non i dati utilizzati.

Per la **comunicazione** tra thread si usano le keyword:

7. `wait()`: **rilascia il lock**, blocca il thread corrente (stato `blocked`) e sposta il thread nella "**wait set**" dell'oggetto;
8. `notify()`: sposta il thread nell'**"entry set"** e lo imposta nello stato "**Runnable**". Dal momento che le code "wait set" ed "entry set" sono gestite con logica FIFO, la **chiamata** a `notify` **sveglia** il **primo thread** che ha chiamato la `wait` sull'oggetto in questione.
9. `notifyAll()`: il comportamento è analogo a `notify` ma agisce su **tutti i thread** che hanno precedentemente chiamato la `wait` su quell'oggetto.
- 10.

Un thread può mettersi in **attesa** di un **altro thread** mediante il metodo `public void join() throws InterruptedException`; che sospende il thread corrente fino a quando non termina l'esecuzione del thread usato come destinatario del metodo.

12.1.1 Daemon Threads

L'esecuzione di un programma multi-threaded generalmente termina quando tutti i thread sono terminati, tuttavia è possibile definire alcuni "**daemon threads**" chiamando il metodo `setDaemon()` prima di `start()`: `public void setDaemon(boolean onoff);`. Si può forzare la terminazione di tutti i thread con il metodo: `public static void exit(int status);`

12.2 Modificatore `volatile`

Per una questione di ottimizzazione la JVM crea automaticamente delle **copie delle variabili d'istanza** a cui accedono più threads. Il modificatore `volatile` consente ai thread di accedere ad un'**area condivisa** di memoria. L'aggiornamento di una variabile `volatile` garantisce l'aggiornamento di tutte le altre variabili.

d'istanza non volatile. Il modificatore da garanzia che gli accessi in scrittura e lettura sulla variabile siano atomici.

13 Networking

Basato su architettura client-server e le classi messe a disposizione dal package java.net. Sostanzialmente è necessario creare le socket per la comunicazione tra processi.

- **socket**: punto terminale di una comunicazione. Identificata (generalmente) da: IP, PORTA, PROTOCOLLO.

13.1 Server

Fa uso dell'interfaccia `ServerSocket` la quale consente la creazione di una **server socket**, specificando la porta di ascolto. L'ascolto (attesa di connessione) inizia mediante il metodo `accept()`. Una volta ricevuta una connessione sulla porta specificata, `accept` viene eseguito e restituisce la socket che rappresenta il **client**. A questo punto il server creare un canale di comunicazione attraverso il metodo `clientSocket.getOutputStream()`.

In sintesi:

1. Creare un oggetto di tipo `ServerSocket` specificando un numero di porta locale;
2. Attendere, tramite il metodo `accept()` una richiesta di connessione da un client;
3. Utilizzare la socket ottenuta ad ogni connessione, per comunicare con il client.

```
try (ServerSocket serverSocket = new ServerSocket(9999)) {
    System.out.println("Server avviato, in ascolto sulla porta 9999");
    while (true) {
        try (Socket clientSocket = serverSocket.accept();
            OutputStream clientOutputStream = clientSocket.getOutputStream();
            BufferedWriter bufferedWriter = new BufferedWriter(
                new OutputStreamWriter(clientOutputStream))) {
            bufferedWriter.write("Ciao client sono il server!");
            System.out.println("Messaggio spedito a " + clientSocket.getInetAddress());
        } catch (ConnectException connExc) {
            System.err.println("Questo client non riesce a connettersi"
                + " con il server: " + connExc.getMessage());
        } catch (IOException e) {
            System.err.println("Problema inaspettato: " + e.getMessage());
        }
    }
} catch (IOException ex) {
    ex.printStackTrace();
}
```

OUTPUT

Server avviato, in ascolto sulla porta 9999

13.2 Client

Il client istanzia un oggetto `Socket` tramite il quale instaura una connessione con la socket del server. L'oggetto `Socket` infatti prende in input l'ip del server e la porta. Il socket istanziato **rappresenta quindi il server** a cui ci si vuole connettere. Successivamente si ottiene un `InputStream` dal socket mediante il metodo

```
getInputStream() .
```

Esempio (Socket API): Client

```
String host = getServerHost(args);
try (Socket socketDelServer = new Socket(host, 9999);
    BufferedReader br = new BufferedReader(
        new InputStreamReader(socketDelServer.getInputStream())));
{
    System.out.println(br.readLine());
} catch (ConnectException connExc) {
    System.err.println("Non riesco a connettermi al server " +
connExc.getMessage());
} catch (IOException e) {
    System.err.println("Problemi... " + e.getMessage());
}
```

OUTPUT CLIENT

Ciao client sono il server!

OUTPUT SERVER

Server avviato, in ascolto sulla porta 9999
Messaggio spedito a /127.0.0.1

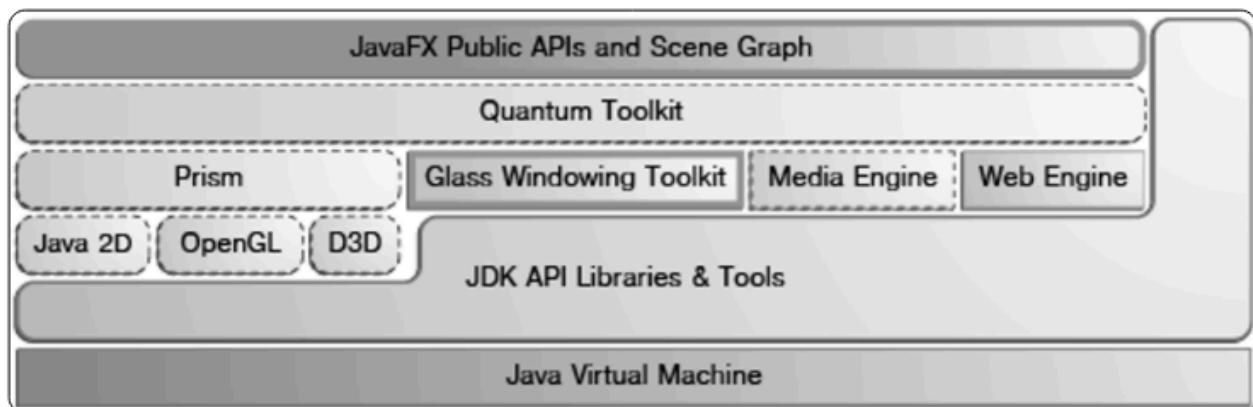
14 Interfacce Grafiche (GUI)

14.1 Pattern MVC

Pattern architettonico che favorisce la separazione dei ruoli dei vari componenti dell'interfaccia grafica:

11. **Model**: implementa la vera applicazione, dati e funzionalità ovvero la **logica di business**;
12. **View**: è composta dalla parte dell'applicazione con cui si interfaccia l'utente. Implementa la **logica di presentazione** e pertanto non contiene nessuna funzionalità ma si limita ad esporre le funzionalità dell'applicazione all'utente;
13. **Controller**: implementa la **logica di controllo** ossia la gestione degli input e la gestione del model.

14.2 JavaFX



JavaFX architecture

- Below the JavaFX public APIs lies the **engine** that runs your JavaFX code. It is composed of subcomponents that include a JavaFX high performance graphics engine, called **Prism**; a small and efficient windowing system, called **Glass**; a media engine, and a web engine.

L'API **JavaFX** si compone di alcuni componenti fondamentali:

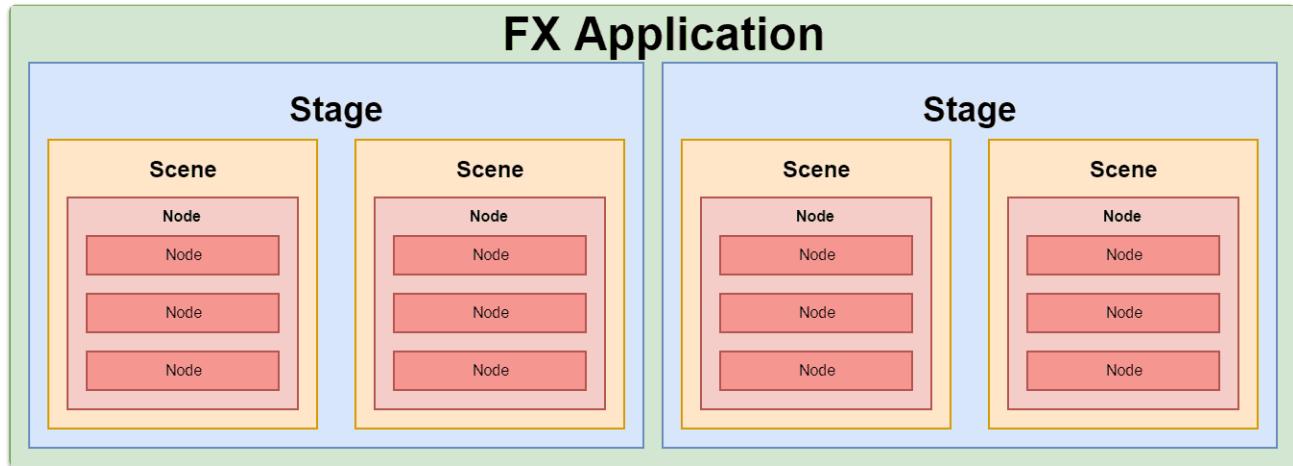
- **Prism**: motore grafico;

- **Glass**: windowing system;

- Media Engine

- Web Engine;

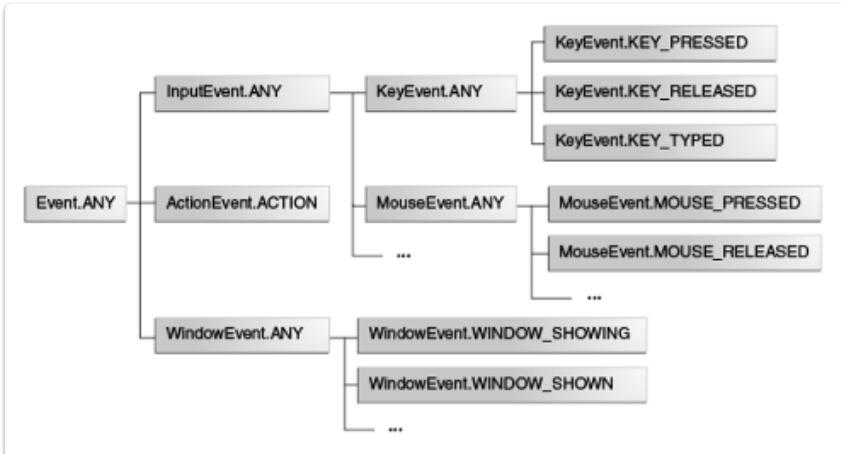
La classe principale di un programma JavaFX è `Application`, la quale contiene l'implementazione astratta del metodo `start()` che deve essere necessariamente sovrascritto in quanto è il primo metodo che sarà chiamato.



Una generica applicazione si compone di almeno 1 stage ed 1 scena:

- **Stage**: contenitore principale, finestra principale dell'applicazione;
- **Scene**: contenitore per visualizzare il contenuto sullo stage. Il contenuto della scena è organizzato ad albero;

14.2.1 Gestione degli eventi



Con evento si intende la notifica che qualcosa è avvenuto, come per esempio il click di un pulsante. Le tipologie di eventi sono organizzate ad albero e l'ascolto può avvenire secondo 2 modalità diverse:

- **Event Filter**: eseguito durante la fase di cattura dell'evento;
- **Event Handler**: eseguito durante la fase di **bubbling** dell'evento;

Entrambe sono una implementazione dell'interfaccia `EventHandler`.

14.2.2 Bindings

JavaFX può fare affidamento sulle API `JavaBeans` per rappresentare una proprietà di un oggetto. Si tratta sostanzialmente di **classi wrapper** che implementano il supporto ai bindings e all'osservabilità.

Il **Bindings** è un meccanismo che consente di mettere in **relazione due variabili**: il cambiamento di una si riflette sull'altra. In questo contesto viene utilizzato per **sincronizzare** i dati dell'interfaccia grafica con i dati del backend dell'applicazione.

```

package bindingdemo;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.binding.NumberBinding;
import javafx.beans.binding.Bindings;

public class Main {

    public static void main(String[] args) {
        IntegerProperty num1 = new SimpleIntegerProperty(1);
        IntegerProperty num2 = new SimpleIntegerProperty(2);
        IntegerProperty num3 = new SimpleIntegerProperty(3);
        IntegerProperty num4 = new SimpleIntegerProperty(4);
        NumberBinding total =
            Bindings.add(num1.multiply(num2), num3.multiply(num4));
        System.out.println(total.getValue());
        num1.setValue(2);
        System.out.println(total.getValue());
    }
}

```

L'API definisce un'insieme di interfacce che consentono agli oggetti di essere notificati quando un valore cambia:

- `ObservableValue`: contiene (wrap) un valore e ne invia le modifiche a quasi `ChangeListener` registrato;
- `Observable`: non contiene un valore e invia le modifiche a qualunque `InvalidationListener` in ascolto; La libreria `FXCollections` mette a disposizione delle collezioni osservabili.

14.2.3 Concorrenza in JavaFX

L'implementazione nativa di un'applicazione JavaFX non è thread-safe e si compone generalmente di un singolo thread. Attività molto lunghe possono quindi bloccare l'applicazione.

Il package `javafx.concurrent` consente la creazione di thread e mette a disposizione un'interfaccia `Worker` implementata da due classi base `Task` e `Service`:

- **Worker** interfaccia generica che rappresenta un "lavoro da eseguire in background"
- **Task**: consente di implementare task (lavori) asincroni. Definisce un oggetto che può essere utilizzato una sola volta;
- **Service**: progettata per eseguire un oggetto di tipo Task su uno o più thread in background, Al posto di eseguire azioni pesanti sul thread principali, è possibile eseguirli su un `Service`, una classe che crea un nuovo thread all'occasione.

L'interfaccia Worker

Funzionamento

Definisce un oggetto che esegue codice su uno o più thread in background. Lo stato dell'oggetto Worker è osservabile e usabile dal JavaFX Application thread. Gli stati di un Worker sono i seguenti:

- READY è lo stato che assume quando è creato
- SCHEDULED è lo stato che assume non appena l'oggetto viene schedulato per eseguire il lavoro
- RUNNING è lo stato che assume non appena il codice del Worker viene eseguito
- SUCCEEDED è lo stato che assume quando il codice del Worker termina con successo.
- FAILED è lo stato che assume quando il codice del Worker solleva qualche eccezione
- CANCELLED è lo stato che assume quando il Worker è stato interrotto

La classe Task

Funzionamento

La classe Task può essere usata per implementare il codice che deve essere eseguito su un thread in background.

Le operazioni da eseguire sono: estendere la classe Task ed effettuare l'override del metodo call. Il metodo call è successivamente invocato dal thread in background, quindi l'implementazione può manipolare stati la cui lettura e scrittura è sicura. Per esempio, **NON** può manipolare uno scene graph attivo (avreste un'eccezione).

Un task può essere eseguito in modo semplice come un oggetto di tipo Runnable:

```
Thread t = new Thread(task);  
t.setDaemon(true);  
t.start();
```

oppure usando l'ExecutorService API:

```
ExecutorService .submit(task);
```

IMPORTANTE: La classe Task definisce un oggetto che può essere utilizzato una sola volta. Per gli oggetti che vogliono essere usati più volte bisogna usare la classe Service.

Funzionamento

La classe **Service** è progettata per eseguire un oggetto di tipo Task su uno o più thread in background. Lo scopo di questa classe è di aiutare lo sviluppatore ad implementare la corretta interazione tra i thread in background e il JavaFX Application thread.

Un oggetto di tipo **Service** può essere avviato, cancellato e riavviato. Per avviare un oggetto di tipo **Service** si può usare il metodo `start()`.

Di default il **Service** usa un Executor con un numero fissato o un numero massimo di thread.

15 JDBC - Java Database Connectivity

JDBC (*Java DataBase Connectivity*) è uno strato di astrazione software che consente alle applicazioni Java di connettersi ad un **database**. Definisce delle interfacce che vengono utilizzate direttamente dalle classi Java implementate dallo sviluppatore grazie al **principio di inversione della dipendenza** (*classi di alto livello non dovrebbero dipendere da quelle di basso livello, ma entrambe dovrebbero dipendere da astrazioni*).

Ciò garantisce indipendenza dal DMBS (*Database Management System*) utilizzato ed è pertanto possibile accedere a diversi database senza modificare completamente l'applicazione. In particolar modo JDBC consente di accedere a RDBMS (*Relational Database Management System*) a patto che questi supportino l'ANSI SQL 2 **standard** (molti db supportano un super-set dell'SQL standard arricchito con comandi proprietari).

Un'applicazione JDBC si compone di 2 parti:

- **Driver JDBC** (fornito dal vendor): classi che implementano le interfacce fondamentali di JDBC
- **Implementazione** dell'app che usa il driver, da parte dello sviluppatore

15.1 Implementazione dei driver JDBC del fornitore

Esistono 4 differenti tipologie di driver JDBC:

1. **JDBC - ODBC Bridge Driver**: driver ormai obsoleto (rimosso da java 8) che presenta un'interfaccia **JDBC** verso il programma ed un'interfaccia **ODBC** (*Open Database Connectivity*). Si tratta, come suggerisce il nome, di un bridge JDBC-ODBC che offre le **prestazioni peggiori**;
2. **Driver Native API Driver**: driver scritto in linguaggio nativo come il **C**, dipendente dalla piattaforma sulla quale viene installato;
3. **Network Protocol Driver o Middleware Driver**: driver Java indipendente dalla particolare piattaforma che viene installato su una **macchina remota** che ha la responsabilità di interagire con uno o più db nel modo più appropriato. Il programmatore si interfaccia con il driver e il driver fa da **middleware** verso i databases.
4. **Pure Java Driver**: driver Java che risiede sulla macchina del client e che viene distribuito con il client stesso.

15.2 Implementazione dello sviluppatore (applicazione JDBC)

Lo sviluppatore ha il compito di scrivere il codice che sfrutta l'implementazione del driver del fornitore.

Un'applicazione JDBC deve:

1. Caricare un driver;
2. Aprire una **connessione** con il database;
3. Creare un oggetto `statement` per interrogare il database;
4. Interagire con il database;
5. Gestire i risultati ottenuti

☰ Example

- L'oggetto `statement` serve da involucro per il trasporto di eventuali istruzioni `SQL` ;
- L'oggetto `ResultSet` contiene il risultato dell'interrogazione

15.3 Indipendenza dal DBMS

La caratteristica più interessante di JDBC è l'**indipendenza** dal particolare DBMS scelto, ovvero la peculiarità di poter cambiare il database da interrogare senza cambiare il codice dell'applicazione. L'indipendenza tuttavia non è **totale** dal momento che è comunque necessario definire la stringa di connessione al database, la quale può cambiare in base al DBMS che si sta utilizzando.

Per superare questa limitazione è possibile definire le informazioni di connessioni all'esterno del programma, in un file di configurazione detto **file di properties**. In questo modo, qualora si volesse cambiare DB, basterà modificare il driver e il file di configurazione, lasciando intatto il codice dell'applicazione.

15.4 Supporto ad SQL

JDBC supporta pienamente `SQL standard` nelle modalità **DQL** (*Data Query Language*), **DML** (*Data Manipulation Language*) e **DDL** (*Data Definition Language*).

- **DML e DDL**: è possibile eseguire qualunque comando CRUD (*Create, Retrieve, Update, Delete*) verso il database mediante il metodo `executeUpdate()`. Un aggiornamento del DB non restituisce un `ResultSet` ma un numero interno che indica il numero dei record aggiornati.
- **DQL**: per le operazioni di `SELECT` si utilizza il metodo `executeQuery()`

15.4.1 Statement parametrizzati

La **parametrizzazione** degli statement offre protezione da `SQL injection`. Il codice `SQL` è specificato direttamente all'interno dell'oggetto `PreparedStatement` ed infatti il metodo `executeUpdate()` non prende in input **nessun parametro**.

☰ Example

```
PreparedStatement stmt = conn.prepareStatement("UPDATE Tabella3 SET m = ? WHERE x = ?");  
stmt.setString(1, "Hi");  
for(int i=0;i<10;i++){  
    stmt.setInt(2,i);
```

```
    int j = stmt.executeUpdate();
}
```

All'interno dell' SQL occorre inserire dei ? in luogo dei valori da **parametrizzare**. Successivamente si utilizzano i metodi `set` (in questo caso `setString()`) per impostare i parametri al posto delle wildcard definite precedentemente, avendo accortezza di specificare, come primo parametro, la **posizione** del ? all'interno del `PreparedStatement`.

15.4.2 Mappatura dei tipi Java - SQL

<code>java.sql.Types</code>	SQL Types
BIGINT	BIGINT
BINARY	CHAR FOR BIT DATA
BLOB	BLOB
BOOLEAN	BOOLEAN
CHAR	CHAR
CLOB	CLOB
DATE	DATE
DECIMAL	DECIMAL
DOUBLE	DOUBLE PRECISION
FLOAT	DOUBLE PRECISION ¹
INTEGER	INTEGER
LONGVARBINARY	LONG VARCHAR FOR BIT DATA
LONGVARCHAR	LONG VARCHAR
NULL	Not a data type; always a value of a particular type
NUMERIC	DECIMAL
REAL	REAL
SMALLINT	SMALLINT
SQLXML ²	XML
TIME	TIME
TIMESTAMP	TIMESTAMP
VARBINARY	VARCHAR FOR BIT DATA
VARCHAR	VARCHAR

15.4.3 Transazioni con JDBC

JDBC supporta anche le **transazioni**, sequenze di operazioni considerate come un'unica operazione atomica: il db viene aggiornato solamente se tutte le operazioni hanno successo... in caso negativo si effettua un **rollback** riportando il database al precedente stato di consistenza.

Per usufruire delle transazioni occorre **disabilitare l'auto-commit**, il quale si occupa di confermare automaticamente ogni operazione:

```
connection.setAutoCommit(false);
```

È pertanto necessario effettuare **manualmente il commit** delle operazioni!

☰ Example

```
try{
    cmd.executeUpdate(INSERT_STATEMENT);
    cmd.executeUpdate(UPDATE_STATEMENT);
    cmd.executeUpdate(DELETE_STATEMENT);
    conn.commit();
} catch (SQLException sqle) {
    sqle.printStackTrace();
    try {
        conn.rollback();
    } catch (SQLException ex){
        ex.printStackTrace();
    }
} finally {
    conn.close();
}
```

15.4.3.1 Stati di isolamento

Nella condizione di default, ogni istruzione è vista come una **transazione** pertanto mentre è in corso un'istruzione come un `UPDATE TABLE` su certi dati, il database bloccherà l'accesso a tali dati da parte di altre transazioni sino al completamento dell'aggiornamento.

Durante una transazione invece si possono presentare determinate **anomalie**, che a seconda del caso possiamo **decidere di eliminare** settando i cosiddetti livelli di isolamento del database.

- **Dirty read**: un primo client legge un valore aggiornato da un secondo client all'interno di una transazione, sui cui non è stato ancora effettuato un COMMIT. La transazione del secondo client potrebbe essere annullata con un ROLLBACK
- **Non-repeatable read**: all'interno della transazione A viene letto un valore. La transazione B aggiorna lo stesso valore e quando A rilegge tale valore lo trova modificato;
- **Phantom read**: transazione A effettua una `SELECT`, transazione B aggiunge un nuovo record tale che, quando A rieffettua la stessa `SELECT`, trova il nuovo record;

Livello	Transazioni	Dirty Read	Non-repeatable read	Phantom read
TRANSACTION_NONE	Non supportata	N/A	N/A	N/A
TRANSACTION_SERIALIZABLE	Supportate	Non permette	Non permette	Non permette
TRANSACTION_READ_COMMITTED	Supportate	Non permette	Permette	Permette
TRANSACTION_REPEATABLE_READ	Supportate	Non permette	Non permette	Permette
TRANSACTION_READ_UNCOMMITTED	Supportate	Permette	Permette	Permette

Tramite JDBC è possibile impostare il livello di isolamento tramite `setTransactionIsolation` della classe `Connection`, ma il driver potrebbe non supportare tutti i livelli di isolamento della transazione.

Livelli di isolamento

- Possiamo impostare il livello di isolamento il metodo `setTransactionIsolation` della classe `Connection` (esiste anche il metodo `getTransactionIsolation`)
- Un driver JDBC potrebbe non supportare tutti i livelli di isolamento è quindi opportuno testare che il driver supporti il livello che si vuole specificare:

```
DatabaseMetaData md = connection.getMetaData();
System.out.println("Current Transaction Isolation Level is " +
    connection.getTransactionIsolation());
System.out.println("This database supports TRANSACTION_NONE Level=" +
    + md.supportsTransactionIsolationLevel(Connection.TRANSACTION_NONE));
System.out.println("This database supports " + "TRANSACTION_READ_COMMITTED Level = " +
    + md.supportsTransactionIsolationLevel(Connection.TRANSACTION_READ_COMMITTED));
System.out.println("This database supports " + "TRANSACTION_READ_UNCOMMITTED Level = " +
    + md.supportsTransactionIsolationLevel(Connection.TRANSACTION_READ_UNCOMMITTED));
System.out.println("This database supports " + "TRANSACTION_REPEATABLE_READ Level = " +
    + md.supportsTransactionIsolationLevel(Connection.TRANSACTION_REPEATABLE_READ));
if (md.supportsTransactionIsolationLevel(Connection.TRANSACTION_SERIALIZABLE));
    connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
System.out.println("Current Transaction Isolation Level is " +
    connection.getTransactionIsolation());
```

Claudio De Sio Cesari: NJ-017

21

15.4.3.2 Savepoints

JDBC consente di definire degli oggetti di tipo `Savepoint` all'interno di una transazione che rappresentano dei **punti di salvataggio** (e ripristino) ai quali poter tornare in caso di `rollback`. Quando un metodo `rollback` viene invocato passandogli in input un oggetto `Savepoint`, tutte le modifiche eseguite dopo la definizione di tale oggetto saranno annullate.

15.5 Architettura di un'applicazione JDBC

Una generica applicazione che deve registrare dati in modo permanente può essere rappresentata mediante 4 strati logici del software:

1. **Interfaccia Utente**
2. **Logica di Business**: strato software che rappresenta l'applicazione stessa, il cuore della logica del programma. Contiene gli algoritmi del sistema;
3. **Logica di Integrazione**: contiene la parte che si occupa di interfacciare la logica di business con il database;
4. **Dati Persistenti**: contiene i dati che devono sopravvivere alla chiusura dell'applicazione. Può essere realizzato con un database oppure con file in vari formati.

Le dipendenze tra i vari strati sono unidirezionali!

15.5.1 Pattern DAO - Data Access Object

Soltamente le applicazioni JDBC sono implementate utilizzando il pattern **DAO**, il quale consente di separare la logica di business dalla logica di accesso ai dati, dal momento che i componenti della logica di business non dovrebbero mai accedere al database. Il pattern si compone di:

- una **classe** (modello) per ogni tabella;
- un'**interfaccia** (DAO) per ogni tabella, contenente tutti i metodi CRUD relativi a quella tabella;
- un'**implementazione** per ogni interfaccia DAO

L'accesso al database è garantito esclusivamente mediante la relativa interfaccia DAO.

☰ Example

```
public class Books {  
    private int isbn;  
    private String bookName;  
    public Books() {}  
    public Books(int isbn, String bookName) {  
        this.isbn = isbn;  
        this.bookName = bookName;  
    }  
    public int getIsbn() {  
        return isbn;  
    }  
    public void setIsbn(int isbn) {  
        this.isbn = isbn;  
    }  
    public String getBookName() {  
        return bookName;  
    }  
    public void setBookName(String bookName) {  
        this.bookName = bookName;  
    }  
}
```

Books rappresenta il model.

```
public interface BookDao {  
    List<Books> getAllBooks();  
    Books getBookByIsbn (int isbn);  
    void saveBook(Books book);  
    void deleteBook(Books book);  
}
```

Per accedere alla tabella *Books* occorre utilizzare un'implementazione di questa interfaccia:

```
public class BookDaoImpl implements BookDao {  
    private List</Books/> books;  
    public BookDaoImpl(){  
        books = new ArrayList<>();  
        books.add(new Books(1, "Java"));  
        books.add(new Books(1, "Python"));  
        books.add(new Books(1, "C"));  
    }  
    @Override  
    public List</Books/> getAllBooks() {  
        return books;  
    }  
    @Override  
    public Books getBookByIsbn(int isbn){  
        return books.get(isbn);  
    }  
    @Override  
    public void deleteBook(Books book){  
        books.remove(book);  
    }  
}
```

```
    }  
}
```

In questo esempio è stato utilizzato un `ArrayList` per simulare una base di dati.

```
public static void main(String[] args) {  
    BookDao bookDao = new BookDaoImpl();  
    for (Books book : bookDao.getAllBooks()) {  
        System.out.println("Book ISBN : " + book.getIsbn());  
    }  
    Books book = bookDao.getAllBooks().get(1);  
    book.setBookName("Database");  
    bookDao.saveBook(book);  
}
```

16 StreamAPI

Uno stream rappresenta genericamente un flusso di dati su cui è possibile eseguire delle operazioni. Spesso tale flusso ha come **sorgente** una **collezione** di dati. A livello pratico si tratta di una sequenza di elementi su cui è possibile iterare.

☰ Creazione di uno stream a partire da un array

```
Stream<\String> stringStream = Stream.of(arrayStringhe);
```

Gli stream sono **immutabili**.

Alcuni metodi forniti dall'interfaccia `Stream` ritornano altri Stream... queste concatenazioni sono dette **pipeline**. Una pipeline si compone di 3 elementi:

1. **Sorgente**: una sorgente da cui ottenere uno `Stream`;
2. **Operazioni di aggregazione**: operazioni intermedie che restituiscono un nuovo oggetto `Stream`;
3. **Operazione terminale**: metodo che restituisce un risultato che non è un oggetto `Stream`.

16.1 Creazione di uno stream

Uno stream può essere creato a partire da:

- valori
- funzioni
- array o collection

16.1.1 Stream da valori (metodi `of`)

☰ Example

```
Stream<\Integer> integerStream = Stream.of(arrayDiInteger);
```

16.1.2 Stream da funzioni (metodi `iterate` e `generate`)

Il metodo `iterate` è simile ad un ciclo `for`. Prende in input 3 parametri:

- valore di `inizializzazione`,
- un `Predicate` che rappresenta la `condizione` per la quale l'iterazione deve terminare)
- `UnaryOperator`, una `Function` che ha come input lo stesso tipo dell'output e che rappresenta l'`aggiornamento` da eseguire ad ogni iterazione.

☰ Example

```
Stream<\Integer> pari = Stream.iterate(2, n -> n <= 10, n -> n+2);
```

Esiste anche un overload a cui manca il 2 parametro e pertanto implementa un **ciclo infinito**. Solitamente questa implementazione si utilizza in coppia con il metodo `limit()` che ha la stessa funzionalità di un `break`.

☰ Example

```
Stream<\Integer> pari = Stream.iterate(2, n-> n+2).limit(5);
```

Il metodo terminale `skip(numero_skip)` invece, ha lo stesso effetto di un `continue`.

Tramite `generate` è invece possibile creare uno stream infinito.

☰ Stream Infinito di numeri casuali

```
Stream<\Double> randomDoubles = Stream.generate(Math::random);
```

Per i tipi atomi, l'API fornisce direttamente le classi `IntStream`, `LongStream`, `DoubleStream` con i rispettivi metodi `ints`, `longs` e `doubles`.

- La classe `java.util.Random` definisce i metodi statici equivalenti a `generate` che si chiamano `ints`, `longs`, e `doubles`. Tali metodi generano rispettivamente stream infiniti di numeri `int`, `long` o `double` (ovvero `IntStream`, `LongStream`, e `DoubleStream`)

```
IntStream randomInts = new Random().ints();
```

Claudio De Sio Cesari: NJ-014

19

16.1.3 Stream da array o collection

Per la creazione a partire da un array si utilizza la classe statica di utilità `Arrays`:

☰ Stream a partire da Array

```
Stream<\String> strings = Arrays.stream(new String[]{"Diego", "Gio", "Miki"});
DoubleStream doubles = Arrays.stream(new double[]{1.1, 2.2, 3.3});
```

La creazione a partire da una collection è simile, dal momento che l'interfaccia `Collection` implementa il metodo `stream`.

```
Set<\String> nomi = Set.of("Matteo", "Antonio", "Enea");
Stream<\String> nomiStream = nomi.stream();
```

16.2 Classi Optional

Sono **classi wrapper** che consentono di evitare `NullPointerException` senza dover ricorrere ai controlli tradizionali. La creazione di un `Optional` avviene mediante la chiamata ad un metodo (eg. `ofNullable()`) e non è possibile usare l'operatore `new`.

Creazione di un Optional

```
public static String getTitoloMaiuscoloOpt(String titolo){
    Optional<\String> opt = Optional.ofNullable(titolo);
    return opt.orElse("NESSUN TITOLO");
}
```

- Se si passa un valore non nullo, viene creato un `Optional` con quel valore
- Se si passa `null` viene creato un `Optional` vuoto

Esistono classi `Optional` già pronte per i tipi atomici: `OptionalInt`, `OptionalLong` ...

16.2.1 Metodi degli Optional

Metodo	Descrizione	Interfaccia funzionale	Esempio d'uso
<code>get</code>	Restituisce il valore se presente, altrimenti lancia <code>NoSuchElementException</code>	—	<code>opt.get()</code>
<code>orElse</code>	Restituisce il valore se presente, altrimenti restituisce il valore di default	—	<code>opt.orElse("default")</code>
<code>orElseGet</code>	Restituisce il valore se presente, altrimenti invoca il fornito <code>Supplier</code>	<code>Supplier<? extends T></code>	<code>opt.orElseGet(() -> calcolaDefault())</code>
<code>orElseThrow</code>	Restituisce il valore se presente, altrimenti lancia l'eccezione fornita dal <code>Supplier</code>	<code>Supplier<? extends X></code>	<code>opt.orElseThrow(() -> new RuntimeException())</code>
<code>isPresent</code>	Restituisce <code>true</code> se il valore è presente	—	<code>opt.isPresent()</code>
<code>isEmpty</code>	Restituisce <code>true</code> se il valore NON è presente (Java 11+)	—	<code>opt.isEmpty()</code>
<code>ifPresent</code>	Esegue il <code>Consumer</code> solo se il valore è presente	<code>Consumer<? super T></code>	<code>opt.ifPresent(val -> System.out.println(val))</code>
<code>ifPresentOrElse</code>	Come sopra, ma permette un'azione anche se il valore non è presente (Java 9+)	<code>Consumer<? super T>, Runnable</code>	<code>opt.ifPresentOrElse(x -> println(x), () -> warn())</code>
<code>filter</code>	Restituisce un <code>Optional</code> vuoto se il predicato fallisce	<code>Predicate<? super T></code>	<code>opt.filter(x -> x > 0)</code>

Metodo	Descrizione	Interfaccia funzionale	Esempio d'uso
map	Trasforma il valore, se presente, usando la funzione fornita	Function<? super T, ? extends U>	opt.map length
flatMap	Come map ma la funzione deve restituire un Optional	Function<? super T, Optional<U>>	opt.flatMap(x -> cercalo(x))
or	Se il valore è presente lo restituisce, altrimenti restituisce il risultato del fornitore	Supplier<? extends Optional<? extends T>>	opt.or(() -> Optional.of("default"))
stream	Restituisce uno stream contenente il valore se presente, altrimenti uno stream vuoto	—	opt.stream()

16.3 Aggregate Operations

Metodi intermedi nella pipeline, restituiscono nuovi stream. Sono detti **lazy** perché vengono invocati solo all'ultimo momento quando sarà invocato il metodo terminale.

Metodo	Descrizione	Interfaccia funzionale	Esempio d'uso
filter	Seleziona solo gli elementi che soddisfano un predicato	Predicate<T>	.filter(x -> x > 0)
map	Trasforma ogni elemento in un altro valore	Function<T, R>	.map(x -> x * 2)
flatMap	Appiattisce stream di stream in uno stream semplice	Function<T, Stream<R>>	.flatMap stream
distinct	Rimuove i duplicati dallo stream	—	.distinct()
sorted	Ordina gli elementi dello stream	Comparator<T> (opzionale)	.sorted()
limit	Limita lo stream ai primi elementi	—	.limit(10)
skip	Salta i primi elementi dello stream	—	.skip(5)
peek	Esegue un'azione su ogni elemento (utile per debug/logging)	Consumer<T>	.peek println
takeWhile	Prende gli elementi finché il predicato restituisce true	Predicate<T>	.takeWhile(x -> x < 10)
dropWhile	Scarta gli elementi finché il predicato restituisce true	Predicate<T>	.dropWhile(x -> x < 10)

16.4 Terminal Operations

Metodi di chiusura della pipeline, non restituiscono Stream.

Metodo	Descrizione	Interfaccia funzionale	Tipo di ritorno	Esempio d'uso
forEach	Esegue un'azione su ogni elemento della	Consumer	void	stream.forEach(x -> println(x))

Metodo	Descrizione	Interfaccia funzionale	Tipo di ritorno	Esempio d'uso
	stream			
allMatch	Restituisce true se tutti soddisfano il predicato	Predicate	boolean	stream.allMatch(x -> x > 0)
anyMatch	Restituisce true se almeno uno soddisfa il predicato	Predicate	boolean	stream.anyMatch(x -> x > 0)
noneMatch	Restituisce true se nessuno soddisfa il predicato	Predicate	boolean	stream.noneMatch(x -> x > 0)
findAny	Restituisce un elemento qualsiasi della stream	-	Optional<\T>	stream.findAny()
findFirst	Restituisce il primo elemento della stream	-	Optional<\T>	stream.findFirst
count	Conta il numero di elementi della stream	-	long	stream.count()
min	Restituisce il minimo secondo un comparatore	Comparator	Optional<\T>	stream.min(comparator)
max	Restituisce il massimo secondo un comparatore	Comparator	Optional<\T>	stream.max(comparator)
sum	Somma tutti i valori (solo per stream numerici)	-	int/long/double	intStream.sum()
average	Calcola la media (solo per stream numerici, OptionalDouble)	-	OptionalDouble	intStream.average()
reduce	Riduce tutti gli elementi a uno solo tramite operazione binaria	BinaryOperator	Optional<\T> / T	stream.reduce(op)
collect	Raccoglie gli elementi in una collection tramite un collector	Collector<T, A, R>	R (es: List<\T>)	stream.collect(toList())

Terminal Operations: riduzione (metodo `reduce`)

- Esistono due versioni del metodo `reduce`. La più completa prende in input due parametri:
 - Un oggetto detto `identity` che rappresenta sia il valore iniziale sia il valore da restituire nel caso non ci siano elementi nello stream
 - Un oggetto di tipo `BinaryOperator` chiamato `accumulator`.
`BinaryOperator` è un'interfaccia funzionale che estende `BiFunction` dove i due parametri e il tipo di ritorno sono dello stesso tipo

```
long totalPrice = smartphones.stream().
    filter(s -> "Samsung".equals(s.getMarca())).
    map(Smartphone::getPrezzo).
    reduce(0, (x, y) -> x + y); // oppure reduce(0, Integer::sum);
```

- Esegue un calcolo per ogni coppia e restituisce un valore il cui tipo è dedotto dal tipo di `identity`

Claudio De Sio Cesari: NJ-014

36

Terminal Operations: riduzione (metodo `reduce`)

- Nella versione senza il primo parametro (`identity`), il risultato sarà restituito in un `Optional`.
- Per esempio, il seguente snippet ritorna un optional con la parola più lunga:

```
Set<String> parole = Set.of("Java", "C", "C++", "C#");

Optional<String> parolaOptional = parole.stream().
    reduce((parola1, parola2) ->
        parola1.length() > parola2.length() ? parola1 : parola2);

if (parolaOptional.isPresent()) {
    System.out.println(parolaOptional.get());
}
```

OUTPUT
Java

16.5 Stream paralleli

Si creano con il metodo `parallelStream()` (al posto di `stream()`) ed utilizzano un algoritmo `fork/join` che suddivide la collezione in più parti ed assegna un'eventuale operazione su ogni parte, ad un thread differente. Al termine di tutti i thread, i risultati vengono ricongiunti.

☰ Example

```
smartphones.parallelStream().
    filter(s->"Samsung".equals(s.getMarca())).
    forEach(s->System.out.println(s));
```

Questo tipo di algoritmo incide profondamente sulle prestazioni pertanto non bisogna aspettarsi che i tempi di esecuzione siano ridotti drasticamente, usando uno stream parallelo.

Stream paralleli e prestazioni

```
long count = words.stream().  
    filter(word -> word.equals(wordToSearch)).count();  
  
count = words.parallelStream().  
    filter(word -> word.equals(wordToSearch)).count();  
  
for (String word : words) {  
    if (word.equals(wordToSearch)) count++;  
}  
  
Iterator<String> iterator = words.iterator();  
while (iterator.hasNext()) {  
    if (iterator.next().equals(wordToSearch)) count++;  
}  
  
int[] countAr = {0};  
words.forEach(word ->{ if (word.equals(wordToSearch)) count[0]++; });
```

OUTPUT

Tempo stream ordinario =	5 millisecondi	count = 1651
Tempo stream parallelo =	15 millisecondi	count = 1651
Tempo ciclo foreach =	6 millisecondi	count = 1651
Tempo oggetto Iterator =	2 millisecondi	count = 1651
Tempo metodo forEach() =	6 millisecondi	count = 1651

17 Reflection API

API Java che consente la scrittura di codice in grado di fare introspezione di altro codice. Mette a disposizione la classe principale `Class` che astrae il concetto di classe o oggetto java.

Metodo 1: metodo `forName`

```
try {  
    Class stringClass = Class.forName("java.lang.String");  
} catch (ClassNotFoundException exc) { // ... }
```

Metodo 2: metodo `getClass`

```
String a = "MiaStringa";  
Class stringClass = a.getClass();
```

Metodo 3: suffisso `.class`

```
Class stringClass = java.lang.String.class;
```

La classe `java.lang.Class` è un tipo generico:

```
public final class Class<T> { // }
```

È usata principalmente per 3 scopi:

- invocare metodi e costruttori privati
- usare una classe conoscendone solo il nome, senza istanziarla
- definire annotazioni e processori di annotazione

17.1 Annotazioni

Tipo che astrae metadati, ovvero informazioni che descrivono altre informazioni. Queste meta-informationi possono essere lette da altro codice mediante la reflection API, tale codice prende il nome di *processore dell'annotazione*. Occorre creare un processore per ogni annotazione da noi definita... il compilatore funge da processore per le annotazioni già esistenti (eg. `Override`).

17.1.1 Definizione del tipo

I tipi annotazioni vanno definiti come qualunque altro tipo Java, secondo la sintassi:

```
[public] @interface identificatore {  
    [elementi_annotazione];  
}
```

Dove `elemento_annotazione` segue la sintassi:

```
tipo_elemento identificatore() [default valore_default]
```

Con tale sintassi si definisce automaticamente sia il parametro che il metodo accessori! I metodi sono implicitamente pubblici ed astratti. Non è possibile specificare parametri in input né `void` come ritorno. Il tipo di ritorno deve essere **primitivo**.

: Example

```
public @interface DaCompletere {  
    String descrizione();  
    String assegnatoA() default "da assegnare";  
}
```

È possibile definire tipi innestati

17.1.2 Usare annotazioni

```
public class Test {  
    @DaCompletere(  
        descrizione = "Bisogna fare qualcosa...",  
        assegnataA = "Claudio"  
    )  
    public void faQualcosa() {  
        //...  
    }  
}
```

All'interno delle tonde si specificano i valori degli elementi.

17.1.3 Processare annotazione

Processare un'annotazione

```
Map<String, String> map = new HashMap<>();
Method[] methods = Class.forName("Test").getMethods();
for (Method m : methods) {
    DaCompletare dc = null;
    if ((dc = m.getAnnotation(DaCompletare.class)) != null) {
        String descrizione = dc.descrizione();
        String assegnataA = dc.assegnataA();
        map.put(descrizione, assegnataA);
    }
}
pubblicaInIntranet(map);
```

17.1.4 Tipologie di annotazioni

17.1.4.1 Marker Annotation

Notazione senza parametri, tipo `Override`.

Annotazioni standard: `Override`, `FunctionalInterface`, `SafeVargs` ,...

```
public @interface Marker{}
```

```
@Marker
public void faQualcosa();
```

Marker Annotation

- Annotazione marcatrice o segnalibro: non dichiara alcun elemento
- Esempio di dichiarazione: `public @interface FactoryMethod {}`
- Annotazioni standard: `Override`, `FunctionalInterface`, `SafeVargs`
- Esempio d'uso:

```
@Override
public Object clone() throws CloneNotSupportedException{
    //...
}
```

17.1.4.2 Single Value Annotation

Annotazione contenente un singolo valore che ha come identificatore (necessariamente) `value`.

- Annotazioni standard: `SuppressWarnings`

```
public @interface Serie {
    Alfabeto value();
```

```
enum Alfabeto {A,B,C};  
}
```

L'enumerazione interna non è vista come un elemento dell'annotazione.

Per la marcatura non è necessario specificare il valore da assegnare:

```
@Serie(value = Serie.Alfabeto.A)  
public void faQualcosa(){}  
  
@Serie (Serie.Alfabeto.A){  
public void faQualcosa(){}  
}
```

- Annotazioni standard: **SuppressWarnings**

- Esempio d'uso:

```
@SuppressWarnings({ "rawtypes"})  
public void print(List l) {  
    //...  
}
```

Claudio De Sio Cesari: NI-011

13

17.1.4.3 Full Annotation

Annotazione completa.

```
public @interface DaCompletere {  
    String descrizione();  
    String assagnataA() default "da assegnare";  
}
```

Full Annotation

- Annotazione completa: annotazione non a singolo valore o marker
- Esempio di dichiarazione:

```
public @interface DaCompletere {  
    String descrizione();  
    String assagnataA() default "da assegnare";  
}
```

- Annotazioni standard: **Deprecated**

- Esempio d'uso:

```
@Deprecated (since = "1.1", forRemoval = true)  
public class Data() {  
    //...  
}
```

17.1.4.4 Meta annotazioni

Annotazioni per annotare annotazioni.

- Meta-annotazione: può annotare annotazioni
- Meta-annotazioni standard: `Target`, `Retention`, `Documented`, `Inherited`, `Repeatable`
- Esempio d'uso:

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

@Target({TYPE, METHOD, CONSTRUCTOR, PACKAGE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletere {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Claudio De Sio Cesari: NJ-011

15

17.1.4.4.1 `@Target`

Specifica quali elementi può marcare l'annotazione. È una SVA basata sull'enum `ElementType`.

```
package java.lang.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

Elemento	L'annotazione annotata può essere applicata a
TYPE	Dichiarazioni di classi, interfacce, o enumerazioni
FIELD	Dichiarazioni di variabili d'istanza
METHOD	Dichiarazioni di metodi
PARAMETER	Dichiarazioni di parametri di metodi
CONSTRUCTOR	Dichiarazioni di costruttori
LOCAL_VARIABLE	Dichiarazioni di variabili locali
ANNOTATION_TYPE	Dichiarazioni di tipi annotazione
PACKAGE	Dichiarazioni di package
TYPE_PARAMETER	Dichiarazioni di tipi parametro (parametri di tipi generici) [Java 8] { NO DEFAULT }
TYPE_USE	Uso di qualsiasi tipo (se usato definisce le annotazioni di tipo) [Java 8] {NO DEFAULT}
MODULE	Dichiarazioni di moduli [Java 9]
RECORD_COMPONENT	Dichiarazioni di record [Java 16]

17.1.4.4.2 `@Retention`

```

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}

```

Specifica come deve essere conservata dall'ambiente java l'annotazione a cui viene applicata.

- **RetentionPolicy** è un'enumerazione che dichiara i seguenti elementi:

Elemento	Come viene conservata dall'ambiente Java
SOURCE	L'annotazione è eliminata dal compilatore
CLASS	L'annotazione viene conservata anche nel file .class ma ignorata dalla JVM
RUNTIME	L'annotazione viene conservata anche nel file .class e letta dalla JVM

17.1.5 Annotazioni standard

1. @Override
2. @FunctionalInterface
3. @Deprecated
4. @SuppressWarnings