# Classi Astratte

## Una classe avente almeno un metodo astratto deve essere dichiarata come classe astratta

NB: Posso comunque dichiarare una classe astratta anche se non contiene alcun metodo astratto!

abstract class Forma {

...

abstract void stampa();

...

}

Un metodo astratto è un metodo di cui non viene specificata l'implementazione

### Una classe astratta

* È una classe non completamente definita (nella parte astratta)
* Non può essere istanziata

Per ottenere una classe concreta (istanziabile) da una astratta occorre definire tutte le implementazioni mancanti ovvero

* Ereditare la classe facendo l'overriding di tutti i metodi astratti

### Permette di trattare omogeneamente oggetti con caratteristiche diverse

Man mano che si sale nella gerarchia dell'ereditarietà, le classi diventano sempre più generiche e probabilmente più astratte.

Ad un certo punto la classe superiore diventa a tal punto generica che la si può pensare come una base per le altre classi piuttosto che come una classe di cui creare un oggetto.

### METODO ASTRATTO

Posso usare la classe astratta quando voglio obbligare tutte le sottoclassi di una classe A ad avere un metodo nomeMetodo(), ma allo stesso tempo non voglio implementare tale metodo nella classe A.

SINTASSI:

public abstract int nomeMetodouovo ;

* un metodo astratto deve essere obbligatoriamente pubblico, altrimenti non avrebbe senso
* lascio il metodo indefinito. infatti non apro il relativo blocco con le parentesi graffe ed il codice del metodo.

## Le classi parzialmente astratte

Le Classi Astratte, oltre ad avere metodi astratti, possono avere metodi ed attributi normali.

Se una classe è astratta, non posso istanziarla: NON POSSO CREARE OGGETTI DA UNA CLASSE ASTRATTA

Che senso ha definire dei metodi astratti e di conseguenza rendere una classe astratta?

* Definisco un metodo astratto quando voglio forzare tutte le sottoclassi ad avere un determinato metodo.
* I metodi astratti funzionano come segnaposto dei metodi implementati poi nella sottoclasse.

## Regole per le sottoclassi

La sottoclasse deve implementare tutti i metodi che la superclasse aveva astratti.

Se la sottoclasse a sua volta non definisce i metodi che la superclasse aveva dichiarato astratti, anche la sottoclasse deve essere dichiarata astratta.

[esempi classi](https://github.com/maboglia/CorsoJava/blob/master/esempi/05_OOP/)

# Interfacce

Le interfacce in Java sono uno strumento potente per definire contratti o specifiche di comportamento che le classi devono implementare. Ecco alcuni concetti chiave sulle interfacce in Java:

1. Definizione di un'Interfaccia:
   * Un'interfaccia è definita utilizzando la parola chiave interface. Essa contiene dichiarazioni di metodi, ma non fornisce l'implementazione dei metodi stessi.

public interface MioInterfaccia {

void metodoUno();

int metodoDue(String parametro);

1. }
2. Implementazione di un'Interfaccia:
   * Una classe implementa un'interfaccia utilizzando la parola chiave implements. La classe deve fornire un'implementazione per tutti i metodi dichiarati nell'interfaccia.

public class MiaClasse implements MioInterfaccia {

@Override

public void metodoUno() {

// Implementazione del metodoUno

}

@Override

public int metodoDue(String parametro) {

// Implementazione del metodoDue

return 42;

}

1. }
2. Estensione di Interfacce:
   * Un'interfaccia può estendere una o più interfacce. Questo consente di creare gerarchie di interfacce, ereditando i metodi delle interfacce padre.

public interface AltraInterfaccia extends MioInterfaccia {

void nuovoMetodo();

1. }
2. Variabili in un'Interfaccia:
   * In Java 8 e versioni successive, è possibile dichiarare variabili di istanza in un'interfaccia. Queste variabili sono implicitamente pubbliche, statiche e finali.

public interface InterfacciaConVariabile {

int VALORE\_COSTANTE = 42;

1. }
2. Default Methods:
   * Dal Java 8, è possibile fornire un'implementazione di default per i metodi in un'interfaccia. Questo consente di estendere le interfacce senza dover modificare tutte le classi che le implementano.

public interface InterfacciaConDefault {

void metodo();

default void metodoDefault() {

// Implementazione di default

}

1. }
2. Metodi Statici in un'Interfaccia:
   * Si possono definire metodi statici all'interno di un'interfaccia, che possono essere chiamati senza creare un'istanza dell'interfaccia.

public interface InterfacciaConMetodoStatico {

static void metodoStatico() {

// Implementazione del metodo statico

}

1. }
2. Utilità delle Interfacce:
   * Le interfacce consentono la progettazione di codice più flessibile e modulare. Possono essere utilizzate per definire contratti comuni, permettendo l'implementazione polimorfica e facilitando la manutenzione del codice.

Le interfacce svolgono un ruolo fondamentale nella progettazione orientata agli oggetti, consentendo una separazione chiara tra contratto e implementazione. La loro utilità diventa evidente quando si desidera che diverse classi condividano un comportamento comune senza ereditarietà multipla diretta, poiché Java non supporta la multi-ereditarietà di classi.

## Un'interfaccia

* Ha tutti i vantaggi e le indicazioni d'uso delle classi astratte
* Presenta maggior flessibilità rispetto all'ereditarietà di una classe astratta
* È possibile definire solo "attributi" final (in pratica costanti)
* Definendo un attributo in un'interfaccia questo viene automaticamente considerato final

Un'interfaccia può essere considerata un modo per cosa dovrebbero fare le classi senza specificare come farlo.

Quindi un'interfaccia non è una classe ma un insieme di requisiti per le classi che si vogliono conformare ad essa.

## interface

Si utilizza la parola chiave interface anzichè class

I metodi sono implicitamente pubblici e astratti, non bisogna indicarlo

Se una classe decide di soddisfare i requisiti di un'interfaccia si dice che la classe implementa l'interfaccia.

### proprietà

Le interfacce non sono classi; non si può utilizzare new per crearne oggetti.

I metodi di un'interfaccia sono automaticamente public (quindi non è necessario scriverlo)

Gli attributi di un'interfaccia sono sempre public static final (non è necessario scriverlo).

Un'interfaccia con il nome NomeInterfaccia va salvata nel file NomeInterfaccia.java (come accade per le classi).

### ereditarietà multipla

Una sottoclasse può estendere solo una superclasse (non è permessa in Java l'ereditarietà multipla)

Con le interfacce invece la situazione è diversa: una classe può implementare quante interfacce vuole.

## Estendere interfacce

Un'interfaccia può estendere un'altra interfaccia, sempre con ereditarietà singola. Possono così essere implementate gerarchie di interfacce (e classi astratte, vedi per esempio [JCF](https://github.com/maboglia/CorsoJava/blob/master/appunti/022_JCF_collezioni.md))

### Sintassi di esempio

public interface NomeInterfaccia extends AltraInterfaccia {

int metodo1( );

}

; }" tabindex="0" role="button">

## Tipi di interfacce

* Normali
* Single Abstract method - @FunctionalInterface
* Marker

Con java 1.8 le interfacce sono state modificate: è possibile implementare due tipi di metodi (!!!)

* default
* static

## Interfacce funzionali e Lambda expressions

Con le interfacce contenenti un singolo metodo astratto, è possibile utilizzare le espressioni lambda

* Ad esempio: Integer raddoppiato = (o) -> o \* 2 ;
* [Interfacce funzionali](https://github.com/maboglia/CorsoJava/blob/master/appunti/018_interfacce_funzionali.md)

[esempi classi](https://github.com/maboglia/CorsoJava/blob/master/esempi/05_OOP/)

# 

# Enum

In Java, le enum (abbreviazione di "enumeration") sono un tipo di dato speciale che rappresenta un insieme di costanti. Le enum sono state introdotte in Java 5 per consentire la dichiarazione di un tipo di dato che consiste in un insieme fisso di valori ben noti. Le enum forniscono un modo pulito e leggibile per rappresentare costanti nel codice.

Ecco alcune caratteristiche chiave delle enum in Java:

Dichiarazione: Puoi dichiarare un tipo enum utilizzando la parola chiave enum. Ad esempio:  
public enum Days {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

1. }  
   In questo esempio, Days è una enum con sette costanti rappresentanti i giorni della settimana.
2. Valori della Enum: Gli elementi di una enum sono chiamati valori della enum. Nel caso dell'esempio sopra, i valori sono MONDAY, TUESDAY, ecc.

Uso delle Enum: Puoi utilizzare le enum in dichiarazioni di variabili, switch statement, e altri contesti in cui sono richiesti valori costanti.  
Days today = Days.MONDAY;

switch (today) {

case MONDAY:

System.out.println("Inizia la settimana!");

break;

case FRIDAY:

System.out.println("Giorno del weekend!");

break;

// Altri casi...

1. }

Metodi nelle Enum: Puoi aggiungere campi e metodi alle enum. Ad esempio, ogni valore di una enum può avere attributi associati.  
public enum Days {

MONDAY("Start of the week"), TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

private String description;

// Costruttore

Days(String description) {

this.description = description;

}

// Metodo per ottenere la descrizione

public String getDescription() {

return description;

}

1. }  
   Ora puoi ottenere la descrizione di un giorno:  
   System.out.println(Days.MONDAY.getDescription()); // Stampa "Start of the week"

Metodo values() e valueOf(): Le enum forniscono metodi utili come values(), che restituisce un array di tutte le costanti di una enum, e valueOf(String), che restituisce la costante con il nome specificato.  
Days[] allDays = Days.values(); // Restituisce un array di tutte le costanti

1. Days aDay = Days.valueOf("MONDAY"); // Restituisce la costante con il nome specificato

Le enum sono utili quando devi rappresentare un insieme fisso e ben noto di valori correlati, come giorni della settimana, mesi, ecc. Forniscono un codice più leggibile e robusto rispetto all'utilizzo di costanti intere o stringhe.

# Nested classes

In Java, le "nested classes" (classi innestate) si riferiscono alla capacità di definire una classe all'interno di un'altra classe. Questa tecnica consente di organizzare e strutturare il codice in modo più logico e coerente. Le classi innestate possono essere suddivise in due categorie principali: classi interne (inner classes) e classi innestate statiche (static nested classes).

1. Classi Interne (Inner Classes):
   * Definizione: Una classe interna è dichiarata all'interno di un'altra classe. Ha accesso diretto ai membri della classe esterna, compresi quelli dichiarati privati.

Esempio:  
class OuterClass {

class InnerClass {

// ...

}

* + }
  + Utilità:
    - Raggruppare logicamente classi utilizzate solo in un'unica posizione.
    - Aumentare l'incapsulamento permettendo a una classe interna di accedere ai membri privati della classe esterna.
    - Migliorare la leggibilità e la manutenibilità del codice.

Istanziamento:  
OuterClass outerObject = new OuterClass();

* + OuterClass.InnerClass innerObject = outerObject.new InnerClass();

1. Classi Innestate Statiche (Static Nested Classes):
   * Definizione: Una classe innestata statica è associata alla sua classe esterna, ma non ha accesso diretto ai membri non statici della classe esterna. È simile a una classe di primo livello e può essere istanziata senza creare un'istanza della classe esterna.

Esempio:  
class OuterClass {

static class StaticNestedClass {

// ...

}

* + }
  + Utilità:
    - Incapsulamento simile a quello di una classe di primo livello per motivi di convenienza di impacchettamento.
    - Non ha accesso ai membri non statici della classe esterna.
  + Istanziamento:  
    OuterClass.StaticNestedClass staticNestedObject = new OuterClass.StaticNestedClass();

L'utilizzo di classi innestate è particolarmente utile quando si desidera organizzare e strutturare il codice in modo che le classi siano strettamente collegate o quando si vuole nascondere l'implementazione dettagliata di una classe all'esterno.

# Interfacce funzionali

Le interfacce funzionali sono un concetto introdotto in Java 8 e rappresentano un tipo di interfaccia che contiene un solo metodo astratto. Queste interfacce sono strettamente legate alla programmazione funzionale e sono utilizzate in congiunzione con le espressioni lambda per consentire una scrittura più concisa e leggibile del codice. Ecco alcuni aspetti importanti delle interfacce funzionali in Java:

1. Il Concetto di Interfaccia Funzionale:
   * Un'interfaccia funzionale è un'interfaccia che contiene esattamente un metodo astratto. Questo metodo rappresenta il contratto che le classi che implementano l'interfaccia devono soddisfare.

@FunctionalInterface

public interface OperazioneMatematica {

int eseguiOperazione(int a, int b);

1. }  
   L'annotazione @FunctionalInterface è opzionale ma consigliata; essa assicura che l'interfaccia abbia un solo metodo astratto, e se ciò non è vero, genera un errore di compilazione.
2. Lambda Expressions:
   * Le espressioni lambda consentono di definire in modo conciso implementazioni di interfacce funzionali. Questo riduce notevolmente la quantità di codice necessario.
3. OperazioneMatematica somma = (a, b) -> a + b;  
   In questo esempio, somma è un'istanza di un'interfaccia funzionale, implementata con un'espressione lambda.
4. Metodi di Default e Statici:
   * Le interfacce funzionali possono includere metodi di default (implementazioni predefinite) e metodi statici senza violare il requisito di avere un solo metodo astratto.

@FunctionalInterface

public interface OperazioneAvanzata extends OperazioneMatematica {

default int doppio(int a) {

return a \* 2;

}

static int triplo(int a) {

return a \* 3;

}

}

1. Tipi di Ritorno:
   * L'uso di un'interfaccia funzionale non è limitato a operazioni con tipi primitivi. Può anche essere utilizzata con tipi di ritorno più complessi, come oggetti o strutture dati.

@FunctionalInterface

public interface EstrattoreStringa {

String estrai(String input);

}

1. API di Java per le Interfacce Funzionali:
   * Java fornisce un pacchetto java.util.function che contiene diverse interfacce funzionali predefinite, come Function, Predicate, e Consumer, progettate per gestire tipi di operazioni comuni nella programmazione funzionale.

import java.util.function.Predicate;

public class Esempio {

public static void main(String[] args) {

Predicate<String> isStringNotEmpty = s -> !s.isEmpty();

System.out.println(isStringNotEmpty.test("Hello")); // Output: true

}

}

Le interfacce funzionali insieme alle espressioni lambda offrono un modo elegante per lavorare con funzionalità di alto livello in Java, rendendo il codice più pulito e leggibile. Questi concetti sono particolarmente utili quando si lavora con API che richiedono oggetti con un solo metodo astratto, come è il caso delle operazioni con i thread o dei metodi di callback.

Studente studente = getUtente(1);

//passo i dati come argomento

BigDecimal mediaVoti = calcolaMedia(studente);

//passaggio come funzionalità

ExecutorService thread = Executors.newFixedThreadPool(4);

//funzionalità da eseguire

RUnnable calcolaMedia = new Runnable(){

public void run(){

//calcola media voti

}

}

//passo la funzionalità come argomento

thread.submit(calcolaMedia);

//la funzionalità è descritta nel metodo run()

public interface Runnable {

void run();

}

## Le interfacce funzionali rappresentano funzionalità invece di dati

Per esempio

* Runnable
  + void run()
* Callable
  + V call() throws Exception
* Comparator
  + int compare(T o1, T o2)

Queste interfacce, già presenti in Java da molto tempo, possono essere definite interfacce funzionali. Da Java 8 si possono usare con tali interfacce le Espressioni Lambda

## Espressioni Lambda

Comparator<Utente> comparator = new Comparator<Utente>(){

@Override

public int compare(Utente u1, Utente u2){

return u1.getId.compareTo(u2.getId());

}

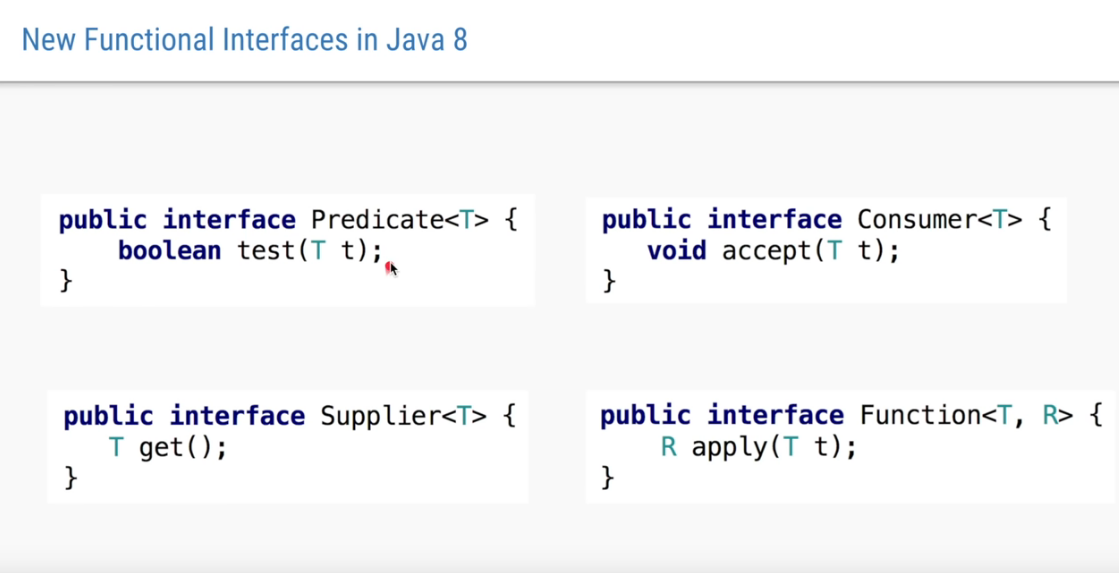
};

con le Lambda diventa

Comparator<Utente> comparator = (u1, u2) -> u1.getId.compareTo(u2.getId());

## Nuove interfacce funzionali in Java 8

* Predicate
  + boolean test(T t)
* Consumer
  + void accept(T t)
* Supplier
  + T get()
* Function<T, R>
  + R apply(T t)



I Predicate in Java sono una parte delle interfacce funzionali introdotte con Java 8 nel pacchetto java.util.function. Un Predicate rappresenta una condizione che può essere valutata come vera o falsa per un determinato input. È comunemente utilizzato per definire condizioni nei filtri o nelle operazioni di selezione in diverse librerie e framework. Ecco alcuni aspetti importanti sui Predicate:

1. Interfaccia Predicate:
   * L'interfaccia Predicate è parametrizzata per indicare il tipo di oggetto su cui verrà eseguita la condizione. Essa dichiara un solo metodo astratto chiamato test.

@FunctionalInterface

public interface Predicate<T> {

boolean test(T t);

}

1. Utilizzo di Predicate:
   * I Predicate sono spesso utilizzati in combinazione con altre classi o interfacce che richiedono una valutazione booleana, come i filtri nelle raccolte o le operazioni di selezione.

Predicate<String> lunghezzaSuperioreAQuattro = s -> s.length() > 4;

if (lunghezzaSuperioreAQuattro.test("Java")) {

System.out.println("La stringa ha lunghezza maggiore di 4.");

}

1. Metodi Predefiniti in Predicate:
   * L'interfaccia Predicate fornisce alcuni metodi predefiniti che consentono la composizione e la combinazione di predicati. Ad esempio, i metodi and, or, e negate.

Predicate<Integer> maggioreDiZero = n -> n > 0;

Predicate<Integer> minoreDiDieci = n -> n < 10;

Predicate<Integer> traZeroEDieci = maggioreDiZero.and(minoreDiDieci);

if (traZeroEDieci.test(5)) {

System.out.println("Il numero è compreso tra 0 e 10.");

}

1. Composizione di Predicati:
   * È possibile combinare predicati in modi complessi utilizzando i metodi di composizione. Ad esempio, and, or, e negate consentono di creare nuovi predicati combinando quelli esistenti.

Predicate<String> iniziaConA = s -> s.startsWith("A");

Predicate<String> lunghezzaSuperioreAQuattro = s -> s.length() > 4;

Predicate<String> condizioneComplessa = iniziaConA.and(lunghezzaSuperioreAQuattro);

if (condizioneComplessa.test("Avanzato")) {

System.out.println("La stringa inizia con 'A' e ha lunghezza maggiore di 4.");

}

1. Metodi Statici in Predicate:
   * L'interfaccia Predicate fornisce anche alcuni metodi statici di utilità. Ad esempio, il metodo isEqual restituisce un predicato che verifica l'uguaglianza con un oggetto dato.

Predicate<String> isEqualJava = Predicate.isEqual("Java");

if (isEqualJava.test("Java")) {

System.out.println("La stringa è uguale a 'Java'.");

}

I Predicate forniscono un modo flessibile e potente per esprimere condizioni booleane in Java, specialmente quando si tratta di filtrare dati o definire comportamenti condizionali. La loro natura funzionale li rende particolarmente adatti all'utilizzo con le espressioni lambda.

### Esempio uso Predicate

List<Utente> utenti = Utente.getAllUsers();

utenti.removeIf(new Predicate<Utente>() {

@Override

public boolean test(Utente utente) {

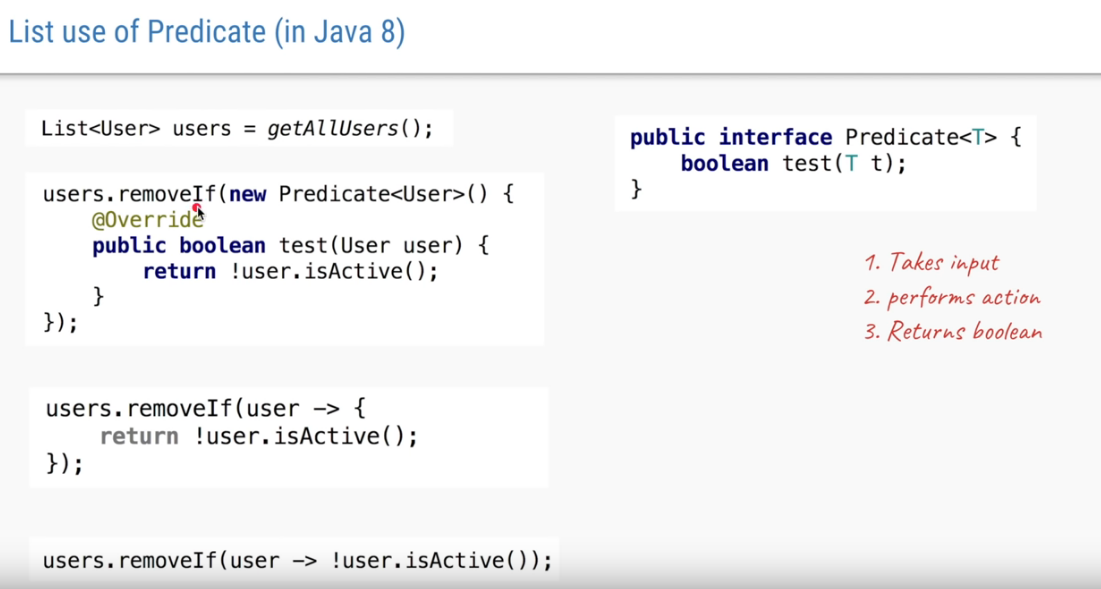
return !utente.isAssente();

}

});

//con le Lambda diventa

utenti.removeIf(utente->!utente.isAssente());



## 

## Consumer

L'interfaccia Consumer fa parte del pacchetto java.util.function introdotto in Java 8. Un Consumer rappresenta un'operazione che accetta un argomento di input e non restituisce alcun risultato. È spesso utilizzato quando si vuole eseguire un'azione su ciascun elemento di una collezione o quando si vuole elaborare un dato in modo specifico senza restituire un valore. Ecco alcuni aspetti importanti sui Consumer:

1. Interfaccia Consumer:
   * L'interfaccia Consumer è parametrizzata per indicare il tipo di oggetto su cui verrà eseguita l'operazione. Essa dichiara un solo metodo astratto chiamato accept.

@FunctionalInterface

public interface Consumer<T> {

void accept(T t);

}

1. Utilizzo di Consumer:
   * I Consumer sono spesso utilizzati in combinazione con altre classi o interfacce che richiedono un'azione senza restituire un valore, come i metodi forEach nelle raccolte.

Consumer<String> stampaMessaggio = s -> System.out.println("Messaggio: " + s);

stampaMessaggio.accept("Hello, World!");

1. Metodi Predefiniti in Consumer:
   * L'interfaccia Consumer fornisce alcuni metodi predefiniti che consentono la composizione e la concatenazione di consumatori. Ad esempio, il metodo andThen permette di concatenare due consumer.

Consumer<String> stampaInMaiuscolo = s -> System.out.println(s.toUpperCase());

Consumer<String> operazioneComposta = stampaMessaggio.andThen(stampaInMaiuscolo);

operazioneComposta.accept("Hello, World!");

1. Utilizzo con Raccolte:
   * I Consumer sono spesso utilizzati per eseguire un'azione su ciascun elemento di una raccolta.

List<String> elencoNomi = Arrays.asList("Alice", "Bob", "Charlie");

Consumer<String> stampaNome = nome -> System.out.println("Nome: " + nome);

elencoNomi.forEach(stampaNome);

1. Esempi di Utilizzo con Map e Stream:
   * I Consumer possono essere utilizzati in combinazione con operazioni su flussi (Stream) per eseguire azioni su elementi di una raccolta in modo conciso.

Map<Integer, String> mappa = Map.of(1, "Uno", 2, "Due", 3, "Tre");

Consumer<Map.Entry<Integer, String>> stampaEntry = entry -> System.out.println(entry.getKey() + ": " + entry.getValue());

mappa.entrySet().forEach(stampaEntry);

1. Metodi Statici in Consumer:
   * L'interfaccia Consumer fornisce alcuni metodi statici di utilità. Ad esempio, il metodo identity restituisce un consumatore che esegue l'operazione di identità.

Consumer<String> identita = Consumer.identity();

identita.accept("Qualsiasi Stringa");

I Consumer sono utili quando si desidera eseguire operazioni su elementi di una collezione o su oggetti specifici senza restituire alcun valore. La loro natura funzionale li rende particolarmente adatti all'utilizzo con le espressioni lambda e con le nuove funzionalità introdotte in Java 8.

### Esempio uso Consumer

static void stampaUtente(Utente u){

System.out.println(u);

}

public static void main(String[] args) {

List<Utente> utenti = Utente.getAllUsers();

utenti.removeIf(new Predicate<Utente>() {

@Override

public boolean test(Utente utente) {

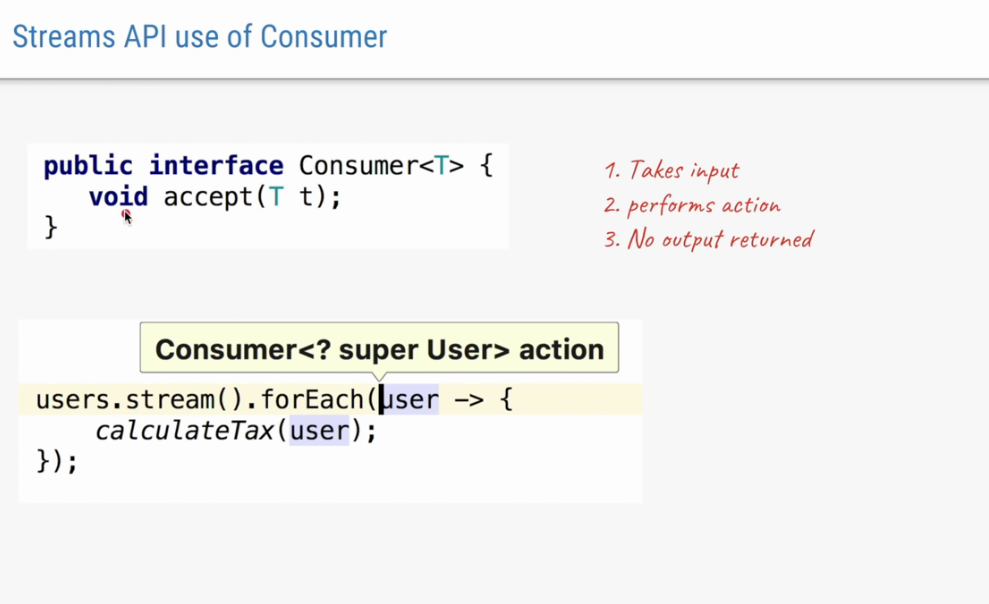
return !utente.isAssente();

}

});

utenti.stream().forEach(utente->stampaUtente(utente));

}



## Supplier

L'interfaccia Supplier è parte del pacchetto java.util.function introdotto in Java 8. Un Supplier rappresenta un fornitore di un risultato, senza alcun argomento in ingresso. È spesso utilizzato quando si desidera generare o fornire un valore, ad esempio in situazioni di lazy loading o quando è necessario generare un valore dinamicamente. Ecco alcuni aspetti importanti su Supplier:

1. Interfaccia Supplier:
   * L'interfaccia Supplier è parametrizzata per indicare il tipo di oggetto che verrà fornito. Essa dichiara un solo metodo astratto chiamato get.

@FunctionalInterface

public interface Supplier<T> {

T get();

}

1. Utilizzo di Supplier:
   * I Supplier sono spesso utilizzati per generare valori in modo lazy, cioè solo quando il valore è effettivamente richiesto.

Supplier<Double> fornisciNumeroCasuale = () -> Math.random();

double numeroCasuale = fornisciNumeroCasuale.get();

System.out.println("Numero casuale: " + numeroCasuale);

1. Utilizzo con Lazy Loading:
   * L'interfaccia Supplier è utile in situazioni in cui si desidera ritardare il calcolo o l'ottenimento di un valore fino a quando non è effettivamente richiesto.

class Configurazione {

// Alcune configurazioni complesse o costose

}

class Applicazione {

private Supplier<Configurazione> configurazioneLazy = () -> calcolaConfigurazione();

private Configurazione calcolaConfigurazione() {

// Logica per calcolare la configurazione

return new Configurazione();

}

public void eseguiApplicazione() {

Configurazione configurazione = configurazioneLazy.get();

// Utilizza la configurazione

}

}

1. Metodi Statici in Supplier:
   * L'interfaccia Supplier fornisce alcuni metodi statici di utilità. Ad esempio, il metodo of crea un fornitore costante che restituisce sempre lo stesso valore.

Supplier<String> fornitoreCostante = Supplier.of("Valore costante");

String valore = fornitoreCostante.get();

System.out.println(valore); // Output: Valore costante

1. Utilizzo con Stream API:
   * I Supplier possono essere utilizzati con Stream API per generare sequenze di valori in modo dinamico.

Stream.generate(() -> Math.random())

.limit(5)

.forEach(System.out::println);

1. Esempio con Combinazione di Supplier:
   * È possibile combinare diversi fornitori utilizzando i metodi di composizione per creare fornitori più complessi.

Supplier<String> primaParte = () -> "Hello";

Supplier<String> secondaParte = () -> " World";

Supplier<String> messaggioCompleto = () -> primaParte.get() + secondaParte.get();

System.out.println(messaggioCompleto.get()); // Output: Hello World

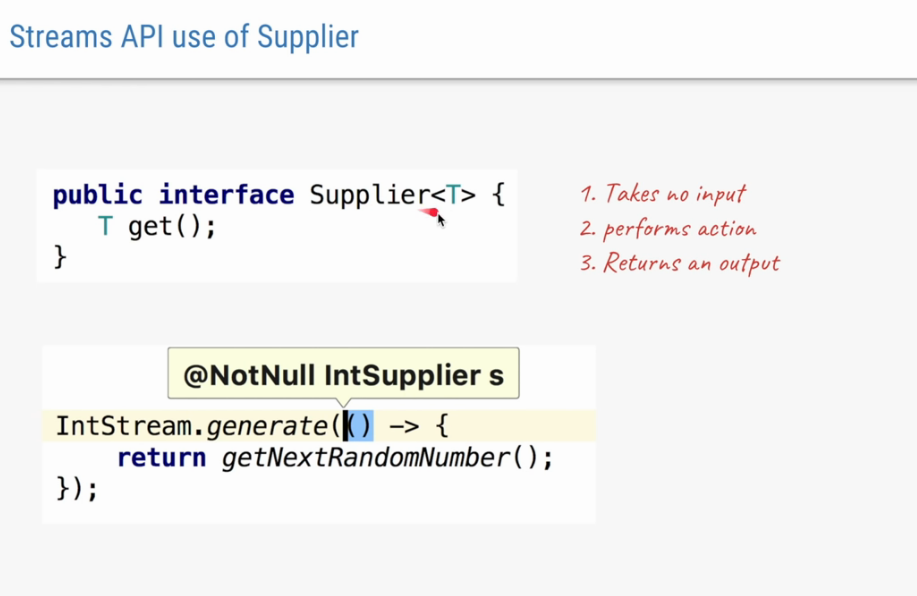
I Supplier sono utili quando si desidera ritardare la generazione di un valore fino a quando non è effettivamente richiesto o quando si desidera creare sequenze di valori dinamicamente. La loro natura funzionale li rende particolarmente adatti all'utilizzo con le espressioni lambda e con le nuove funzionalità introdotte in Java 8.

### 

### 

### 

### Esempio uso Supplier



## Function

L'interfaccia Function è parte del pacchetto java.util.function introdotto in Java 8. Una Function rappresenta una funzione che accetta un argomento di input e restituisce un risultato. È spesso utilizzata per eseguire trasformazioni o operazioni sui dati, dove l'output dipende dall'input. Ecco alcuni aspetti importanti sulla Function:

1. Interfaccia Function:
   * L'interfaccia Function è parametrizzata per indicare il tipo di oggetto di input (T) e il tipo di oggetto di output (R). Essa dichiara un solo metodo astratto chiamato apply.

@FunctionalInterface

public interface Function<T, R> {

R apply(T t);

}

1. Utilizzo di Function:
   * Le Function sono spesso utilizzate in combinazione con altre classi o interfacce che richiedono una trasformazione o un'applicazione di funzione.

Function<String, Integer> lunghezzaStringa = s -> s.length();

int lunghezza = lunghezzaStringa.apply("Java");

System.out.println("La lunghezza della stringa è: " + lunghezza);

1. Metodi Predefiniti in Function:
   * L'interfaccia Function fornisce alcuni metodi predefiniti che consentono la composizione e la concatenazione di funzioni. Ad esempio, il metodo andThen permette di concatenare due funzioni.

Function<String, Integer> lunghezzaStringa = s -> s.length();

Function<Integer, String> descriviLunghezza = n -> "La lunghezza è " + n;

Function<String, String> lunghezzaDescrittiva = lunghezzaStringa.andThen(descriviLunghezza);

String risultato = lunghezzaDescrittiva.apply("Java");

System.out.println(risultato);

1. Utilizzo con Raccolte e Stream:
   * Le Function sono spesso utilizzate con le operazioni su flussi (Stream) per trasformare i dati all'interno di una raccolta.

List<String> elencoNomi = Arrays.asList("Alice", "Bob", "Charlie");

Function<String, String> trasformaInMaiuscolo = s -> s.toUpperCase();

List<String> nomiInMaiuscolo = elencoNomi.stream().map(trasformaInMaiuscolo).collect(Collectors.toList());

System.out.println(nomiInMaiuscolo);

1. Metodi Statici in Function:
   * L'interfaccia Function fornisce alcuni metodi statici di utilità. Ad esempio, il metodo identity restituisce una funzione che restituisce il suo input senza modifiche.

Function<String, String> identita = Function.identity();

String risultato = identita.apply("Java");

System.out.println(risultato);

1. Esempio con Combinazione di Funzioni:
   * È possibile combinare diverse funzioni utilizzando i metodi di composizione per creare trasformazioni più complesse.

Function<String, Integer> lunghezzaStringa = s -> s.length();

Function<Integer, String> descriviLunghezza = n -> "La lunghezza è " + n;

Function<String, String> trasformazioneComposta = lunghezzaStringa.andThen(descriviLunghezza);

String risultato = trasformazioneComposta.apply("Java");

System.out.println(risultato);

Le Function forniscono un modo flessibile e potente per esprimere trasformazioni o operazioni sui dati in Java. La loro natura funzionale li rende particolarmente adatti all'utilizzo con le espressioni lambda e con le nuove funzionalità introdotte in Java 8.

### Esempio uso Function<T, R>

