PROGRAMMAZIONE AD OGGETTI

JAVA

II SEMESTRE II ANNO 2023-2024

CORSO DEL PROF. BICOCCHI

E1.1 – JAVA FROM FUNCTIONS TO CLASSES

PROGRAMMAZIONE PROCEDURALE

La **programmazione procedurale** punta ad avere una sequenza di procedure o funzioni che vengono chiamate in modo sequenziale per risolvere un determinato problema. Questo approccio non è adatto per grandi progetti. Tra le varie problematiche della programmazione procedurale, abbiamo:

- ***Difficoltà nel riutilizzo del codice***, questo comporta ad avere:
  - **Codice riscritto**, nell'approccio procedurale è comune dover riscrivere codice perché le funzioni non sono facilmente riutilizzabili in contesti leggermente diversi.
  - **Difficile da manutenere**, la ripetizione del codice non solo aumenta la quantità di codice da mantenere, ma rende anche difficile trovare e correggere i bug. Se un bug è presente in un blocco di codice duplicato, è probabile che il bug esista in ogni duplicato di quel blocco.
  - **Meno testato**, l'approccio procedurale non incoraggia "la separazione delle responsabilità" il che porta a funzioni che fanno troppe cose e ciò rende difficile testare le unità separate.
- ***Non protezione dei dati***, all'aumentare della dimensione del codice è più semplice accedere ed usare una variabile fuori dallo scope per cui è stata creata (***violazione di sicurezza ed integrità***).

C

```c
int età = 17;   // Assegno un numero che rappresenta l'età
// ...
età = 1700;     // Mi dimentico che la variabilile è associata ad un età e la riassegno
```

- ***I dati sono separati dalle operazioni***, questo rende difficile la decomposizione del codice. È più complesso suddividere e distribuire il lavoro efficacemente e può portare a conflitti e difficoltà di coordinamento, ad esempio:
  - Si può verificare una **mancanza di sincronizzazione** tra operazioni e dati, portando all'incongruenza dei dati.
  - **Complessità di debug**.
  - I **dati indipendenti** possono essere manipolati da più operazioni e ciò può portare all'inconsistenza dei dati.
  - **Operazioni non ottimizzate**, "mi dimentico della presenza di quel dato e ne alloco un altro".

! Differenze: Inconsistenza, incongruenza e incoerenza
Inconsistenza → mancanza di affidabilità dei dati.
Incongruenza → mancanza di corrispondenza tra due o più elementi.
Incoerenza → mancanza di logica o di ordine tra i dati.

C

```c
People* Persona(char name[10], char surname[10], int age) {
    People* p = malloc(sizeof(People));
    strcpy(p->name, name);
    // ...
    return p;
}
```

Java

```java
public class Persona {      // Dati incapsulati nella classe
    String name;
    String surname;
    int age;

    public Persona(String name, String surname, int age) {
        this.nome = nome;
        this surname = surname;
        this.age = age;
    }
}
```

PROGRAMMAZIONE AD OGGETTI

La **programmazione orientata agli oggetti**, punta ad un software sicuro, ottimizzabile e flessibile, adatto per grandi progetti. Si astrae ulteriormente rispetto alla programmazione procedurale tramite la definizione di un nuovo concetto noto come **classe** che unisce le variabili e i metodi in aggregazioni.
La programmazione ad oggetti è sviluppata su 3 fondamenta:

- **Encapsulation** (*incapsulamento*),
- **Inheritance** (*ereditarietà),*
- **Polymorphism** (*polimorfismo)*:
  - **Polimorfismo statico**,(durante la compilazione)
  - **Polimorfismo dinamico**, (durante l'esecuzione del programma).

E2.1 – JAVA BASICS

NOZIONI DI BASE JAVA

Un programma Java è composto da **pacchetti** (*package*) contenenti uno o più file (*classi*).

Un **file** contiene una classe pubblica e volendo, zero o più classi private. Il nome del file deve essere necessariamente uguale al nome della classe pubblica.

In Java non esistono funzioni tradizionali ma metodi all'interno di classi. Un **metodo** è un blocco di codice che definisce il comportamento di un oggetto, quindi, è una funzione che opera sull'oggetto manipolando i dati al suo interno o influenzando lo stato della classe. I metodi possono avere $0, \dots, n$ parametri e possono ritornare *un valore*. In Java, quando si passa un argomento ad un metodo, che sia un tipo primitivo o un oggetto, esso viene passato come copia. Ci sono però significative differenze:

- **Tipi primitivi**, viene passata la copia effettiva del valore, questo significa che eventuali modifiche non influenzeranno il valore originale all'esterno del metodo.
- **Oggetti**, si passa una copia del riferimento all'oggetto, questo significa che punteranno allo stesso riferimento, quindi, eventuali modifiche saranno effettive anche al di fuori del metodo.

! Variabili di tipo riferimento
Un **riferimento** è una variabile che permette di accedere ad un oggetto, senza di esso, non è possibile accedervi. I riferimenti sono variabili **primitive** e sono archiviate nello **stack**. Gli oggetti puntati dai riferimenti sono allocati dinamicamente e risiedono nella memoria **heap**.

Un programma Java inizia sempre con la chiamata al metodo ***main***.

I **tipi primitivi** in Java sono:

Type	Size (bits)	Minimum	Maximum	Example
byte	8	-2^7	2^7 – 1	*byte b = 100;*
short	16	-2^15	2^15 – 1	*short s = 30_000;*
int	32	-2^31	2^31 – 1	*int i = 100_000_000;*
long	64	-2^63	2^63 – 1	*long l = 100_000_000_000_000;*
float	32	-2^-149	(2-2^-23)·2^127	*float f = 1.456f;*
double	64	-2^-1074	(2-2^-52)·2^1023	*double f = 1.456789012345678;*
char	16	0	2^16 – 1	*char c = 'c';*
boolean	1	–	–	*boolean b = true;*

In Java ci sono diversi tipi di **letterali**:

Java

```java
/*
    Letterali integrali
*/

// Intero decimale
int numeroPositivo = 15;    // Positivo
int numeroNegativo = -10;   // Negativo

// Intero Ottale --> iniziano sempre per 0 (accetta cifre [0-7])
int numeroOttale = 077;

// Esadecimale --> iniziano sempre per 0x (accetta cifre [0-9][A-F])
int esadecimale = 0x39FAB;

// Intero binario --> iniziano sempre per 0b (accetta cifre [0-1])
int binario = 0b1010100111;
```

Java

```java
/*
    Letterali in virgola mobile
*/

// Precisione singola --> precisione di circa 7 cifre decimali (float) [f, F]
float numero = 7.81F

// Precisione doppia --> precisione di circa 15 cifre decimali (double) [d, D]
double numero = 15.9832D

// Decimali in forma esponente [e, E]
double numero = 323e-6

/*
    Altri letterali
*/
char ch = 077   // Char letterali
boolean bool = true // Letterali booleani
Integer int = null  // Letterali nulli
```

In Java le **costanti** sono variabili il cui valore non può essere modificato una volta assegnato. Sono fondamentali anche per rendere il codice più sicuro e leggibile.

Java

```java
// final --> garantisce l'immutabilità dell'elemento
public static final NUM_STUDENTS = 69;  // Parole maiuscole e separate da _
```

La classe **java.lang.Math** contiene metodi per eseguire operazioni numeriche di base, come le funzioni elementari esponenziale, logaritmo, radice quadrata e trigonometriche.

L'interfaccia **RandomGenerator** è progettata per fornire un protocollo comune per oggetti che generano sequenze casuali o (più tipicamente) pseudo-casuali di numeri (o valori booleani)

```
public class RandomDemo {

    public static void main(String[] args) {

        RandomGenerator rnd = RandomGenerator.getDefault();

        int i1 = rnd.nextInt();            // [Integer.MIN_VALUE, Integer.MAX_VALUE]
        int i2 = rnd.nextInt(100);        // [0, 100)
        int i3 = rnd.nextInt(10, 20);    // [10, 20)
    }
}
```

Java ha a disposizione una serie di operatori. Gli **operatori** sono simboli o parole chiave che rappresenta un'azione da eseguire sui dati. Gli operatori sono organizzati in gerarchie di priorità:

Operatore	Descrizione (da più a meno)
`()`	Parentesi per l'ordine di valutazione
`expr++`, `expr--`	Incremento e decremento
`!`, `~`	Negazione logica e complemento unario
`*`, `/`, `%`	Moltiplicazione, divisione, resto
`+`, `-`, `<<`, `>>`, `>>>`	Addizione, sottrazione, shift e shift con segno
`<`, `<=`, `>`, `>=`, `==`, `!=`	Operatori di confronto
`instanceof`	Verifica l'appartenenza ad una classe
`&`	AND
`^`	XOR
`|`	OR
`&&`	AND
`||`	OR
`? :`	Operatore ternario (condizionale)
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`	Assegnamenti

Nella JDK 10 del 2008 è stata introdotta in Java la parola chiave **var**. Questa parola chiave permette al programmatore di non specificare il tipo, sarà il compilatore a dedurlo.

Java

```
var name = "Prova"; // String
name = 10;  // Errore, il compilatore specifica il tipo ma è fisso!
```

Casting. Il casting può essere di:

- **implicit casting**, quando il tipo di destinazione è "più largo" del tipo di origine, viene eseguito automaticamente dal compilatore e senza perdita di informazione.
- **explicit casting,** quando il tipo di destinazione è "più piccolo" del tipo di origine e potrebbe causare la perdita di informazioni (es. double d = 2.00003; long l = (long) d;).

Variabili di riferimento. Un riferimento è una variabile che fornisce un modo per accedere a un oggetto. In genere, non è possibile accedere a un oggetto senza un riferimento ad esso. I riferimenti sono variabili primitive e sono memorizzati nello stack.

Gli oggetti, invece, sono allocati dinamicamente e risiedono nella memoria heap. Per questo motivo, il ciclo di vita degli oggetti non dipende da alcun metodo specifico. (I riferimenti e gli oggetti sono l'equivalente Java dei puntatori struct e delle struct allocate dinamicamente in C.)

Passaggio di parametri ai metodi. È possibile passare parametri primitivi (compresi i riferimenti a oggetti) a metodi, i parametri sono sempre passati per valore (copiati nello stack del metodo ricevente).

CONVENZIONI

In Java ci sono diverse **convenzioni di codifica**:

- I package sono scritti in minuscolo.
- Le interfacce devono essere rappresentate con un aggettivo.
- I metodi di accesso ai dati devono iniziare con la parola “get”.
- I metodi di modifica dei dati devono iniziare con la parola “set”.
- I metodi devono essere rappresentati da una frase imperativa.
- Un metodo booleano deve iniziare con la parola “is” o “has”.

Java

```java
int sonoUnaVariabile;   // Prima lettera minuscola, le altre maiuscole
public final int VARIABILE_COSTANTE = 98;   // Le parole sono in maiuscolo separato da _
public class SonoUnaClasse { ... }  // Tutte le parole iniziano in maiuscolo

// Le graffe:
public static void fun () {

}
```

ISTRUZIONI PER IL CONTROLLO DI FLUSSO

Decision statements

if

```
if (condition1) {
  // executed if
  // condition1 is true
} else if (condition2) {
  // executed if
  // condition1 is false and condition2 is true
} else {
  // executed if
  // condition1 is false and condition2 is false
}
```

switch

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

switch (enhanced)

```
switch(expression) {
  case x -> // code block
  case y, z -> // code black
  default -> // code block
}
```

Iterative statements

do-while

```
do {
    // code block to be executed
} while (condition);
```

while

```
while (condition) {
    // code block to be executed
}
```

for

```
for (type variableName: arrayName) {
    // code block to be executed
}
```

```
for (type variableName: arrayName) {
    // code block to be executed
}
```

L' istruzione **break** può essere usata per uscire da un ciclo.

L' istruzione **continue** interrompe un'iterazione (nel ciclo), ma continua con l'iterazione successiva invece di uscire.

E2.2 – JAVA BASICS: ARRAYS

In Java un **array** è una struttura dati che può memorizzare una sequenza di elementi di dimensione fissa dello stesso tipo di dati, è anch'esso un oggetto e l'indice parte da 0. La lunghezza di un array è fissa e non può essere modificata successivamente.

Gli array possono memorizzare elementi di qualsiasi tipo, inclusi i tipi primitivi e possono essere anche multidimensionali. Un array **multidimensionale** è un array che contiene altri array.

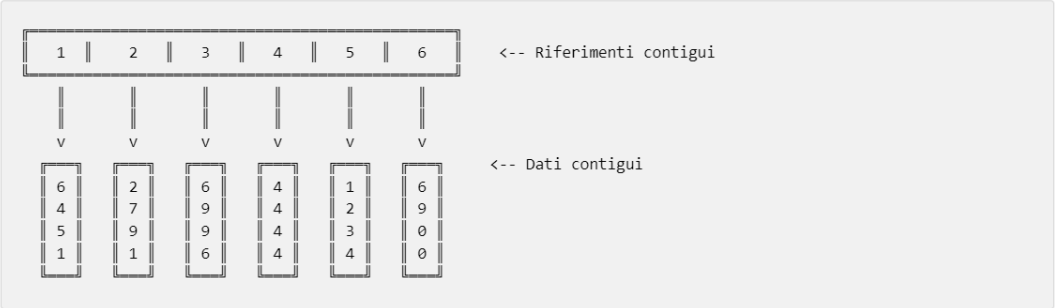
Creare in Java un array equivale ad allocare n spazi contigui nell' heap della dimensione del tipo di dato scelto, ma ***attenzione***:

- Se il tipo di dato è primitivo equivale ad avere in memoria valori contigui in memoria.
- Se il tipo di dato è un oggetto equivale ad avere in memoria i riferimenti (i **puntatori**) ai valori contigui in memoria (***NON*** i valori in se).

! Ricorda

Creare un vettore significa allocare una zona di memoria fissa che viene inizializzata a 0 se non vengono specificati i valori.

Anche con gli array multidimensionali la memoria è contigua ed organizzata in questo modo:



! ATTENZIONE: L'immagine descrive un array multidimensionale di tipi primitivi. In caso di oggetti i *dati contigui sono riferimenti*.

Java

```java
// Array
int[] numbers = { 1, 2, 3, 4 };
int[] numbers = new int[4];

// Matrix
int[][] numbers = { {1, 2}, { 3, 4}, { 5, 6 } };
int[][] numbers = new int[3][2];
```

Si può trovare la dimensione dell'array tramite l'attributo associato *numbers.length*.
Se l'indice punta all'esterno dell'Array, ovvero l'elemento non esiste, otteniamo un *ArrayIndexOutOfBoundsException*. Questo errore indica che l'Array non contiene l'indice specificato.

Passaggio di array ai metodi. Si può usare gli array come parametro di un metodo proprio come qualsiasi altra variabile. Quando si usa l'array come parametro di un metodo, il metodo riceve una copia del riferimento all'array.
Un array è un oggetto, quindi quando si modifica l'array all'interno del metodo, le modifiche persistono dopo l'esecuzione del metodo.

ARRAY UTILITIES

java.util.Arrays contiene vari metodi per manipolare array come l'ordinamento, la ricerca, il riempimento, la stampa o la visualizzazione di collections:

- copyOf() / copyOfRange
- fill()
- equals() / deepEquals()

- hashCode() / deepHashCode()
- sort() – ordinare un vettore
- binarySearch() – ricercar di un elemento in un array ordinato
- toString() / deepToString() – stampare un array
- asList() – trasformarlo in una lista
- setAll() – trasformarlo in una collection

Altro metodo utile:

System.arraycopy() copia un array dall'array sorgente specificato, iniziando dalla posizione specificata, alla posizione specificata dell'array di destinazione. Il numero di componenti copiati è uguale all'argomento lunghezza.

E2.3 JAVA BASICS: STRINGS

La classe **String** rappresenta stringhe di caratteri (sono oggetti e non sono puntatori a char). Sono **immutabili** e quindi i valori non possono essere modificati.

! Differenze nella creazione di stringhe

Java

```java
String nome = "Nicola Bicocchi";
```

Le stringhe inizializzate in questa modalità non vengono ricopiate in memoria, sono archiviate nella zona chiamata "**string pool**". Quindi, quando viene inizializzata una stringa, il compilatore Java controllerà prima in questa zona di memoria. In caso esista la stringa verrà ritornato il riferimento alla stringa esistente.
Il pool delle stringhe è gestito dalla **Java Virtual Machine** (JVM).

Java

```java
String nome = new String("Nicola Bicocchi");
```

Le stringhe inizializzate con l'operatore new verranno archiviate in memoria heep come qualsiasi altro oggetto. Quindi, ogni stringa occupa la sia memoria.

! Confronto di stringhe

String è un oggetto quindi esso funziona allo stesso modo. Per confrontare delle stringhe non basta usare l'operatore "==", ci sarà solo il confronto dei riferimenti.

Java

```java
String nome = new String("Nicola Bicocchi");
String nome2 = new String("Nicola Bicocchi");

// Confronto dei RIFERIMENTI
if(nome == nome2)
    System.out.println("NO!! Non sono uguali, ho confrontato i riferimenti");

// Confronto delle STRINGHE
if(nome.equals(nome2))
    System.out.println("Si! Le stringhe sono uguali");
```

Quando si parla di stringhe, fondamentale è l'operatore "**+**" per la concatenazione di stringhe:

Java

```java
String nomeCognome = "Nicola " + "Bicocchi"  // Concatenazione di stringhe
String nomeNumero = "Nicola" + 10;  // Concatenazione di tipi non stringa
```

! Performance - STRINGBUILDER

L'utilizzo dell'operatore "+" per la concatenazione di stringhe può essere utilizzato per concatenare poche stringhe. Ad ogni concatenazione le stringhe verranno scartate e ci sarà l'allocazione e la copia

di una nuova stringa in memoria. All'aumentare delle concatenazioni questa operazione diventa **ordini di grandezza più lento** dell'uso di altri modi di concatenazione come la funzione "append()" di **StringBuilder**.

STANDARD STREAMS

Java dispone di tre flussi che vengono utilizzati per l'input e l'output delle applicazioni Java. Vengono inizializzati dal runtime di Java all'avvio della JVM. I tre flussi sono:

- **System.in**, chiamato standard input, utilizzato per leggere i dati in ingresso da un flusso di input.
- **System.out**, chiamato standard output, utilizzato per scrivere dati di output su un flusso di output.
- **System.err**, chiamato standard error, utilizzato per inviare messaggi di errore o di avviso alla console o a un altro flusso di output destinato a gestire errori.

Metodi PrintStream. Stampa una stringa:

```
void print(String s)
```

Scanner è una classe di utilità per leggere da vari input e ha metodi comodi per lavorare con l'input. Suddivide il suo input in token usando un modello di delimitazione, che per impostazione predefinita corrisponde agli spazi vuoti. I token risultanti possono quindi essere convertiti in valori di tipi diversi usando i vari metodi successivi.

```
/* from stdin */
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();

/* from a file */
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

Possiamo utilizzare *i blocchi di testo* dichiarando la stringa con """ (tre virgolette doppie):

```
String textBlock = """
    He who becomes the slave of habit,
    who follows the same routes every day,
    who never changes pace,
    who does not risk and change the color of his clothes,
    who does not speak and does not experience,
    dies slowly.""";
```

E3.1 – JAVA CLASSES

Una classe in Java è dichiarata con la parola chiave **class**, mentre, per crearne l'istanza, si utilizza la parola chiave **new**:

Java

```
/*
*   Con la creazione dell'istanza della classe, ogni campo viene inizializzato
*   con il valore predefinito del tipo corrispondente se non inizializzato da
*   un costruttore
*/
Person p = new Person();
System.out.println(p.name); // null
System.out.println(p.age);  // 0
```

Il corpo può includere:

- **Metodi**, funzioni definite all'interno della classe che consentono di definirne il comportamento e di manipolarne i dati.

Un metodo può essere di due tipi:

- **Statico**, per invocare il metodo non è necessario creare un oggetto. Invochiamo il metodo con il nome della classe.
  Un metodo statico ha però delle differenze sostanziali:
  - Può accedere solo a campi statici.
  - Può invocare solo metodi statici.
  - Non può fare riferimento alla parola this.

Java

```java
public class StaticPrint {
    static void print() {
        System.out.println("Hello World!");
    }
    public static void main(String[] args) {
        StaticPrint.print();
    }
}
```

- **Di istanza**, per invocare il metodo è necessario creare un oggetto.
  I metodi di istanza possono accedere a metodi statici e hanno il vantaggio di poter accedere ai campi dell'oggetto. Per riferirsi ai campi di una classe si usa la parola chiave **this** (rappresenta una particolare istanza della classe) ed è fondamentale per differenziarlo dai campi un altro oggetto.

Java

```java
public class InstancePrint {
     void Print() {
        System.out.println("Hello World!");
    }
    public static void main(String[] args) {
        InstancePrint p = new InstancePrint();
        p.Print();
    }
}
```

- **Campi** (**attributi**, **variabili di istanza**), sono variabili contenute in un oggetto e che ne definiscono lo stato. Possono essere di qualsiasi tipo, compresi tipi primitivi e classi stesse.
  Anche i campi possono essere dichiarati statici, questo crea il vantaggio di poter condividere delle variabili tra le classi e risparmiare memoria, però, non è una buona idea dichiarare variabili statiche non costanti.

Java

```java
public class Persona {
    final String nome;
    String cognome;
    int età;

    class Lavoro {
        String sede;
        String titolo;
        String settore;
    }
}
```

- **Costruttori**, creano e inizializzano nuovi oggetti della classe, cioè, impostano i valori iniziali dei campi.
  Sono metodi speciali che vengono chiamati automaticamente quando viene creata un'istanza della classe con la parola chiave new.
  Si differenziano dagli altri metodi perché:
  - Hanno lo stesso nome della classe che lo contiene.
  - Non hanno tipo di ritorno, nemmeno void.

Si possono definire infiniti costruttori, a patto che, i parametri siano diversi, questo concetto è nome come **sovraccarico del costruttore**.

Java

```java
public class Persona {
     String nome;
     String cognome;
     int età;

    /*
    *   Costruttore semplice, inizializzo tutti i valori
    */
    public Persona(String nome, String cognome, int età) {
        this.nome = nome;
        this.cognome = cognome;
        this.età = età;
    }

    /*
    *   Costruttore alternativo, inizializzo solo il nome e il cognome
    *   e definisco un'èta di default
    */
    public Persona(String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome;
        this.età = 18;
    }
    /*
    *   Un costruttore, può chiamare anche un altro costruttore, usando:
    *       this();
    */
    public Persona(String nome, String cognome) {
        this(nome, cognome);
        this.età = 18;
    }
}
```

MODIFICATORI DI ACCESSO

Un modificatore di accesso è una parola chiave che specifica chi può usare il tuo codice o parti di esso, posizionabile davanti a campi, metodi o classi. Serve per la chiarezza del codice, permettendo di "nascondere" parti complesse e fornendo solo le interfacce necessarie (come i pulsanti di una lavatrice). Inoltre, garantisce la sicurezza del codice, impedendo ad altri sviluppatori di modificare variabili critiche, assicurando così che il comportamento del codice rimanga prevedibile.

VISIBILITÀ

Visibiltà della classe. Una classe può avere uno dei due seguenti modificatori:

- **package-private** (predefinito, nessun modificatore esplicito): visibile solo per le classi dello stesso pacchetto;
- **pubblico**: visibile a tutte le classi ovunque.

Il modo più comune di utilizzare i modificatori è:

- rendere pubbliche le classi contenenti metodi per gli utenti (i "pulsanti") ;
- rendere private tutte le altre classi con metodi logici di basso livello utilizzati da quelle pubbliche (coprire il motore con il corpo).

Visibilità dei membri. Un membro di una classe (un campo o un metodo, ad esempio un costruttore di classe) ha più opzioni tra cui scegliere:

- *privato* disponibile solo all'interno di una classe;
- *package-private* (noto anche come default) disponibile per tutte le classi nello stesso pacchetto;
- *protetto* disponibile per le classi nello stesso pacchetto e per le sottoclassi (sarà trattato in seguito);
- *pubblico*, accessibile per tutte le classi e ovunque.

Modificatore	Classe	Pacchetto	Sottoclasse	Tutto
Pubblico	SÌ	SÌ	SÌ	SÌ
Protetto	SÌ	SÌ	SÌ	NO
Predefinito	SÌ	SÌ	NO	NO
Privato	SÌ	NO	NO	NO

INCAPSULAMENTO DEI DATI (ENCAPSULATION)

Nella maggior parte dei casi, una classe non espone i suoi campi ad altre classi. Invece, rende i suoi campi accessibili tramite i cosiddetti metodi di accesso.

Secondo il principio di incapsulamento dei dati, i campi di una classe non sono accessibili direttamente da altre classi. I campi sono accessibili solo tramite i metodi di quella particolare classe.

Per accedere ai campi nascosti, i programmatori scrivono tipi speciali di metodi:

- **Getter**, iniziano con get per convenzione, tranne per i boolean che iniziano per ***is***, servono per ottenere il valore di una variabile.
- **Setter**, iniziano con set per convenzione, servono per modificare il valore di una variabile, controllato dal progettista della classe.

Entrambi i tipi di metodi dovrebbero essere public. L'utilizzo di questi metodi ci offre alcuni vantaggi:

- i campi di una classe possono essere resi di sola lettura, di sola scrittura o entrambi;
- una classe può avere il controllo totale sui valori memorizzati nei campi;
- gli utenti di una classe non sanno come la classe memorizza i suoi dati e non dipendono dai campi.

Perché l'Incapsulamento è importante?

L'incapsulamento è una tecnica che nasconde i dettagli di implementazione di una classe, avvolgendola in uno "scudo protettivo", per proteggere i dati da modifiche indesiderate o da accessi non autorizzati. La classe mette a disposizione all'esterno solo un elenco di metodi pubblici.

- *Raggruppamento di dati e metodi*: All'interno della classe stessa si raggruppano i dati e i metodi che operano sugli stessi attributi.
- *Controllo d'accesso:* Attraverso l'uso di modificatori di accesso.

Se è presente un bug all'interno del codice, possiamo escludere a priori che l'errore dipenda da un utilizzo scorretto da parte dell'esterno, proprio perché gli unici modi per utilizzare metodi o richiedere delle variabili sono interni. Questo rende notevolmente più semplice trovare e sistemare il bug.

IMMUTABILITÀ DEGLI OGGETTI

Immutabilità significa che una volta inizializzato un oggetto, esso non può essere cambiato. Per modificare un oggetto è necessario ricrearne uno nuovo con le caratteristiche specificate. I tipi immutabili permettono di scrivere programmi con meno errori. (le stringhe sono un esempio di oggetto immutabile)

Convezione nella creazione di classi
- privato, campo finale per ogni pezzo di dati; - getter per ogni campo; - costruttore pubblico con un argomento corrispondente per ogni campo; - metodo equals che restituisce true per oggetti della stessa classe quando tutti i campi corrispondono; - metodo hashCode che restituisce lo stesso valore quando tutti i campi corrispondono;

- metodo toString che include il nome della classe e il nome di ciascun campo e il suo valore corrispondente.

I record sono classi di dati immutabili che richiedono solo il tipo e il nome dei campi. I metodi equals, hashCode e toString, così come i campi private, final e il costruttore public, sono generati dal compilatore Java.

CONDIVISIONE DEI RIFERIMENTI

Più di una variabile può fare riferimento allo stesso oggetto. È importante capire che due variabili fanno riferimento agli stessi dati in memoria piuttosto che a due copie indipendenti. Poiché la classe degli oggetti è mutabile, possiamo modificare l'oggetto usando entrambi i riferimenti.

Per il riferimento all'oggetto sono definiti solo gli operatori relazionali == e !=:

- La condizione di uguaglianza indica se due riferimenti puntano allo stesso oggetto in memoria
- La condizione di uguaglianza viene valutata sui valori dei riferimenti (ovvero gli indirizzi in memoria)!

GARBAGE COLLECTOR

Il Garbage Collector tiene traccia di ogni oggetto disponibile nell'heap e rimuove quelli inutilizzati. Il suo compito principale è:

- Rilevare gli oggetti non più raggiungibili.
- Liberare la memoria degli oggetti non referenziati.

VANTAGGI	**SVANTAGGI**
Nessuna gestione manuale della memoria	Richiede maggiore uso della CPU
Gestione automatica della perdita di memoria	I programmatori non hanno il controllo su quando avverrà la liberazione della memoria
I puntatori pendenti e i puntatori selvaggi nella programmazione del computer sono puntatori che non puntano a un oggetto valido del tipo appropriato. Questi sono casi speciali di violazioni della sicurezza della memoria. Più in generale, i riferimenti penzolanti e i riferimenti selvaggi sono riferimenti che non si risolvono in una destinazione valida.	

CLASSI WRAPPER

Primitive	Wrapper
boolean	Boolean
byte	Byte
char	Character
int	Integer
float	Float
double	Double
long	Long
short	Short

Ogni tipo primitivo ha una classe a lui dedicata. Queste classi sono note come wrapper e sono **immutabili** (proprio come le stringhe). Le classi wrapper possono essere utilizzate in diverse situazioni:

- quando una variabile può essere null(assenza di un valore);
- quando si desidera utilizzare metodi speciali di queste classi (ad esempio per la conversione da e verso stringhe).
- quando è necessario memorizzare valori in raccolte generiche (sarà preso in considerazione nei prossimi argomenti);

Gli oggetti wrapper sono tipi di riferimento (puntatori).

Il **boxing** è la conversione di tipi primitivi in oggetti delle corrispondenti classi wrapper. L'**unboxing** è il processo inverso. C'è un possibile problema durante l'unboxing. Se l'oggetto wrapper è null e viene utilizzato in una inizializzazione o in altre operazioni, l'unboxing genera un NullPointerException.

```
int primitive = 100;
Integer reference = Integer.valueOf(primitive); // boxing
int anotherPrimitive = reference.intValue();    // unboxing
```

E3.2 – JAVA CLASSES: INHERITANCE

L'ereditarietà è un meccanismo che permette di creare una nuova classe derivata da un'altra, acquisendo campi e metodi della classe base senza riscriverli. Rappresenta la relazione IS-A ed è fondamentale nella programmazione orientata agli oggetti, consentendo la creazione di gerarchie di classi e il riutilizzo del codice. La classe derivata è chiamata sottoclasse, mentre la classe da cui deriva è chiamata superclasse.

```
class SuperClass { }

class SubClassA extends SuperClass { }
```

Ci sono punti importanti sull'ereditarietà in Java:

- Java non supporta l'ereditarietà multi-classe, quindi una classe può ereditare solo da una singola superclasse.
- Una gerarchia di classi può avere più livelli (una classe C può estendere una classe B che estende una classe A).
- Una superclasse può avere più di una sottoclasse.
- Una sottoclasse eredita tutti i campi e i metodi pubblici e protetti dalla superclasse, ma non quelli privati. Tuttavia, i metodi pubblici o protetti della superclasse possono fornire accesso ai campi privati.
- I costruttori non sono ereditati, ma possono essere invocati dalla sottoclasse usando la parola chiave super.
- Se una classe è dichiarata con la parola chiave final, non può avere sottoclassi.

Esistono due modi per fare riferimento a un oggetto sottoclasse:

- Riferimento alla **sottoclasse,** creando istanze delle sottoclassi utilizzando il loro costruttore.
- Riferimento alla **superclasse,** dove una variabile della superclasse può fare riferimento a oggetti delle sue sottoclassi.

Accesso ai campi e ai metodi tramite un riferimento alla superclasse:

L'ereditarietà è un meccanismo che permette di creare una nuova classe derivata da un'altra, acquisendo campi e metodi della classe base senza riscriverli. Rappresenta la relazione IS-A ed è fondamentale nella programmazione orientata agli oggetti, consentendo la creazione di gerarchie di classi e il riutilizzo del codice. La classe derivata è chiamata sottoclasse, mentre la classe da cui deriva è chiamata superclasse.

Ci sono punti importanti sull'ereditarietà in Java:

- Java non supporta l'ereditarietà multi-classe, quindi una classe può ereditare solo da una singola superclasse.
- Una gerarchia di classi può avere più livelli (una classe C può estendere una classe B che estende una classe A).
- Una superclasse può avere più di una sottoclasse.
- Una sottoclasse eredita tutti i campi e i metodi pubblici e protetti dalla superclasse, ma non quelli privati. Tuttavia, i metodi pubblici o protetti della superclasse possono fornire accesso ai campi privati.
- I costruttori non sono ereditati, ma possono essere invocati dalla sottoclasse usando la parola chiave super.
- Se una classe è dichiarata con la parola chiave final, non può avere sottoclassi.

Esistono due modi per fare riferimento a un oggetto sottoclasse:

- Riferimento alla **sottoclasse,** creando istanze delle sottoclassi utilizzando il loro costruttore.

```
Person person = new Person(); // the reference is Person, the object is Person
Client client = new Client(); // the reference is Client, the object is Client
Employee employee = new Employee(); // the reference is Employee, the object is Employee
```

- Riferimento alla **superclasse,** dove una variabile della superclasse può fare riferimento a oggetti delle sue sottoclassi.

```
Person client = new Client(); // the reference is Person, the object is Client
Person employee = new Employee(); // the reference is Person, the object is Employee
```

Accesso ai campi e ai metodi tramite un riferimento alla superclasse: Un **riferimento di superclasse** può essere usato per qualsiasi oggetto di sottoclasse derivato da essa, ma non può accedere ai membri specifici della sottoclasse.

Casting tra superclasse e sottoclasse: È possibile eseguire il cast di un oggetto di una sottoclasse nella sua superclasse e viceversa, ma il cast alla sottoclasse deve essere fatto con attenzione per evitare ClassCastException.

```
Person person = new Client();

Client clientAgain = (Client) person; // it's ok
Employee employee = (Employee) person; // the ClassCastException occurs here
```

Quando utilizzare il riferimento alla superclasse: Utile per l'elaborazione di una raccolta di oggetti di tipi diversi della stessa gerarchia e per metodi che accettano oggetti della classe base ma funzionano con le sue sottoclassi.

```
public static void printNames(Person[] persons) {
    for (Person person : persons) {
        System.out.println(person.getName());
    }
}

public static void main(String[] args) {
    Person person = new Employee();
    person.setName("Ginger R. Lee");

    Client client = new Client();
    client.setName("Pauline E. Morgan");

    Employee employee = new Employee();
    employee.setName("Lawrence V. Jones");

    Person[] persons = {person, client, employee};

    printNames(persons);

}
```

Accesso ai campi e ai metodi della superclasse: La parola chiave super può essere usata per accedere ai metodi di istanza o ai campi della superclasse, utile per distinguere tra membri con lo stesso nome.

```
class SuperClass {
    protected int field;

    protected void printBaseValue() {
        System.out.println(field);
    }
}
```

```
class SubClass extends SuperClass {

    protected int field;

    public SubClass() {
        this.field = 30;  // It initializes the field of SubClass
        super.field = 20; // It initializes the field of SuperClass
    }

    public void printSubValue() {
        super.printBaseValue(); // It invokes the method of SuperClass, super is optional here
        System.out.println(field);
    }
}
```

Invocazione del costruttore della superclasse: L'invocazione super(...) deve essere la prima istruzione in un costruttore di sottoclasse.
Il costruttore predefinito di una sottoclasse chiama automaticamente il costruttore senza argomenti della superclasse.
Inoltre, nelle sottoclassi è possibile sovrascrivere i metodi equals() e hashCode() per confrontare campi aggiuntivi.

Tipi di ereditarietà:

- Ereditarietà singola: una sottoclasse eredita da una singola classe.
- Ereditarietà multipla: una sottoclasse eredita da più classi (non supportata direttamente in Java).
- Ereditarietà multilivello: una sottoclasse può ereditare da un'altra sottoclasse.
- Ereditarietà gerarchica: più classi derivano da una singola classe.

! In Java, l'ereditarietà multipla diretta non esiste, ma è possibile ottenere un comportamento simile implementando più interfacce.

LA CLASSE OBJECT

La Java Standard Library ha una classe denominata Object che è il genitore predefinito di tutte le classi standard e delle tue classi personalizzate. Ogni classe estende questa implicitamente, quindi è la radice dell'ereditarietà nei programmi Java. La classe appartiene al *java.lang* pacchetto che viene importato di default.
La Object classe può fare riferimento a un'istanza di qualsiasi classe perché qualsiasi istanza è un tipo di Object(upcasting).

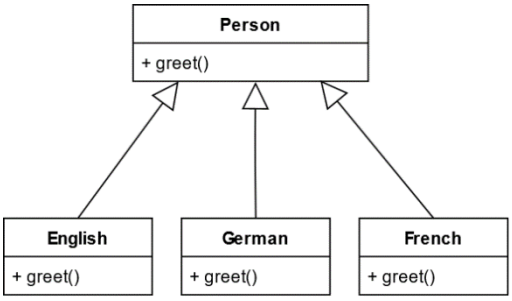
La Object classe fornisce alcuni metodi comuni a tutte le sottoclassi. Ha nove metodi di istanza (esclusi i metodi sovraccaricati) che possono essere divisi in quattro gruppi:

- **identità dell'oggetto**:
    - *hashCode* (valore del codice hash intero dell'oggetto, questo può cambiare in più esecuzioni della stessa applicazione),
    - *equals* (confronto tra due riferimenti)

  Se due oggetti sono uguali in base al equals()metodo, allora il loro codice hash deve essere lo stesso. Se due oggetti sono disuguali secondo il equals()metodo, non è necessario che il loro codice hash sia diverso. Il valore del loro codice hash può essere uguale o meno;

- **rappresentazione leggibile dall'uomo**: *toString*;
- **sincronizzazione thread**:
    - wait,
    - notify,
    - notifyAll;
- **gestione degli oggetti**:
    - finalize,
    - clone,
    - getClass.

POLIMORFISMO



Il polimorfismo (polimorfismo) è la capacità di fornire diverse implementazioni di un metodo ereditato a seconda del tipo di parametro passato al metodo, quindi sono concetti generici che si comportano diversamente in base alla situazione. Il polimorfismo si divide in:

- Polimorfismo **statico**: si verifica durante la **compilazione**, il compilatore determina quale metodo o sovraccarico di metodo chiamare in base al tipo statico delle variabili coinvolte nell'invocazione del metodo. Java lo supporta come **overloading** di metodi, consentendo ai programmatori di definire più metodi con lo stesso nome ma parametri diversi.

Java --> Polimorfismo statico

```java
public class Calculator {
    public int sum(int a, int b) {
        return a + b;
    }
    public double sum(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        int integerResult = calculator.sum(5, 3); // Chiamata al metodo somma(int, int)
        double doubleResult = calculator.sum(2.5, 3.7); // Chiamata al metodo somma(double, double)
    }
}
```

- Polimorfismo **dinamico**: la decisione su quale metodo eseguire dipende dal tipo dinamico dell'oggetto passato durante l'esecuzione del programma. Java lo supporta tramite **l'override** dei metodi. Il polimorfismo in fase di **esecuzione** funziona quando un metodo sovrascritto viene chiamato tramite la variabile di riferimento di una superclasse. Java determina in fase di esecuzione quale versione del metodo (superclasse/sottoclassi) deve essere eseguita in base al tipo di oggetto a cui si fa riferimento, non al tipo di riferimento. Utilizza un meccanismo noto come *dynamic method dispatching*.

Java --> Polimorfismo dinamico

```java
class Animal {
    void verse() {
        System.out.println("Generic sound");
    }
}

class Dog extends Animal {
    @Override
    void verse() {
        System.out.println("Bau bau");
    }
}

class Cat extends Animal {
    @Override
    void verse() {
        System.out.println("Miao miao");
    }
}

public class Main {
    public static void main(String[] args) {
        Animale animalOne = new Cat();
        Animale animalTwo = new Dog();

        animalOne.verse(); // Chiamata al metodo verso() del Cane
        animalTwo.verse(); // Chiamata al metodo verso() del Gatto
    }
}
```

- Polimorfismo **parametrico**: quando il codice è scritto senza menzionare alcun tipo specifico e quindi può essere utilizzato in modo trasparente con qualsiasi numero di nuovi tipi. Java lo supporta come **generici** o **programmazione generica**.

METODI DI ISTANZA DI OVERRIDE

L'override dei metodi di Java è una caratteristica fondamentale che consente alle sottoclassi di ridefinire i comportamenti ereditati dalle loro superclassi. Java offre l'opportunità di dichiarare un metodo in una sottoclasse con lo stesso nome di un metodo nella superclasse. Il *vantaggio dell'override* è che una sottoclasse può fornire la propria implementazione specifica di un metodo della superclasse. L'override dei metodi nelle sottoclassi consente a una classe di ereditare da una superclasse il cui comportamento è sufficientemente simile e quindi di modificare tale comportamento in base alle esigenze della sottoclasse.

I metodi di istanza possono essere sovrascritti se sono ereditati dalla sottoclasse. Il metodo sovrascritto deve avere lo stesso nome, parametri (numero e tipo di parametri) e tipo di ritorno (o una sottoclasse del tipo) del metodo sovrascritto.

Regole per l'override dei metodi:

- Il metodo deve avere lo stesso nome della superclasse.
- Gli argomenti dovrebbero essere esattamente gli stessi del metodo della superclasse.
- Il tipo di ritorno dovrebbe essere lo stesso tipo o un sottotipo del tipo di ritorno dichiarato nel metodo della superclasse.
- Il livello di accesso deve essere uguale o più aperto del livello di accesso del metodo sovrascritto.
- Un metodo privato non può essere sovrascritto perché non è ereditato dalle sottoclassi.
- Se la superclasse e la sua sottoclasse sono nello stesso pacchetto, allora i metodi privati del pacchetto possono essere sovrascritti.
- I metodi statici non possono essere sovrascritti.

Per verificare queste regole, c'è un'annotazione speciale @Override. Se il compilatore decide che il metodo non può essere sovrascritto, genererà un errore. Tuttavia, l'annotazione non è obbligatoria, è solo per comodità. Se si desidera **impedire l'override** di un metodo, dichiararlo con la parola chiave *final*.
L'override e l'overload dei metodi possono essere combinati insieme anche se sono due meccanismi differenti.

Nascondere i metodi statici: i metodi statici non possono essere sovrascritti. Se una sottoclasse ha un metodo statico con la stessa firma (nome e parametri) di un metodo statico nella superclasse, allora il metodo nella sottoclasse nasconde quello nella superclasse. Questo è completamente diverso dall'override del metodo.

E3.3 JAVA CLASSES: INTERFACE, ABSTRACT CLASSES

INTERFACCIA

Un'interfaccia può essere considerata un tipo speciale di classe che non può essere istanziata. Per dichiarare un'interfaccia, dovresti usare la parola chiave *interface* invece di *class*. Un'interfaccia può contenere:

- **costanti pubbliche**.
- metodi **astratti** *senza implementazione* (la parola chiave abstract non è richiesta qui).
- metodi **statici** con implementazione (la parola chiave static è obbligatoria).
- metodi **predefiniti** con implementazione (la parola chiave default è obbligatoria), i metodi di interfaccia sono *astratti* per impostazione predefinita, ciò significa che **non** possono avere un corpo ma dichiarano solo una firma. Con l'eccezione dei metodi di default che possono avere un corpo.
- metodi **privati** con implementazione.

```
interface Interface {

    int INT_CONSTANT = 0; // it's a constant, the same as public static final int INT_CONSTANT = 0

    void instanceMethod1();

    void instanceMethod2();

    static void staticMethod() {
        System.out.println("Interface: static method");
    }

    default void defaultMethod() {
        System.out.println("Interface: default method. It can be overridden");
    }

    private void privateMethod() {
        System.out.println("Interface: private methods in interfaces are acceptable but should have a body");
    }
}
```

Se i modificatori non vengono specificati una volta dichiarato il metodo, i suoi parametri saranno **pubblici** e **astratti** per impostazione predefinita. I metodi statici, predefiniti e privati devono avere un'implementazione nell'interfaccia.

```
public class Pencil implements DrawingTool {
    @Override
    public String draw(Curve curve) {
        return "Pencil drawing a " + curve.draw();
    }
}
public class Brush implements DrawingTool {
    @Override
    public String draw(Curve curve) {
        return "Brush drawing a " + curve.draw();
    }
}
public class DrawingApp {
    public static void main(String[] args) {
        Curve curve = new Curve();
        DrawingTool[] drawingTools = new DrawingTool[]{new Pencil(), new Brush()};
        for (DrawingTool tool : drawingTools) {
            System.out.println(tool.draw(curve));
        }
    }
}
```

L'utilizzo delle interfacce è un altro modo di supportare **il polimorfismo**.

Una delle caratteristiche più importanti dell'interfaccia è **l'ereditarietà multipla**.

```
class A { }

interface B { }
interface C { }

class D extends A implements B, C { }
interface E extends B, C { }
```

In alcune situazioni, un'interfaccia *non ha alcun membro*. Tali interfacce sono chiamate interfacce **marcatore** o **taggate**.

Perché sono necessari i metodi predefiniti

L'idea principale di un'interfaccia è dichiarare la funzionalità. I metodi predefiniti estendono quell'idea. Non solo dichiarano la funzionalità, ma la implementano anche. La ragione principale è supportare la **retrocompatibilità**.

Qual è lo scopo dei metodi statici?

Lo scopo principale dei metodi statici dell'interfaccia è definire funzionalità di utilità comuni a tutte le classi che implementano l'interfaccia. Aiutano a evitare la duplicazione del codice e a creare classi di utilità aggiuntive.

CLASSI ASTRATTE

Una classe astratta è una classe dichiarata con la parola chiave **abstract**. Rappresenta un concetto astratto che viene utilizzato come classe base per le sottoclassi. Le classi astratte hanno alcune caratteristiche speciali:

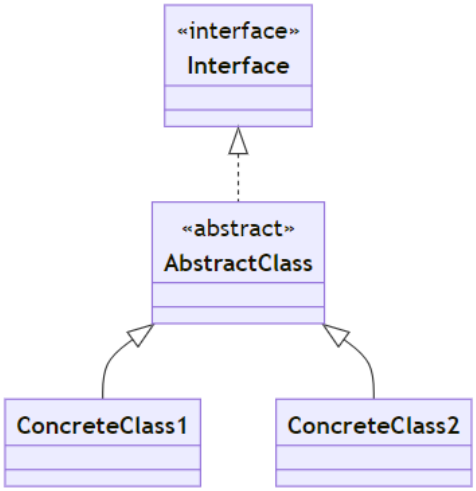
- è impossibile creare un'istanza di una classe astratta;
- una classe astratta può contenere metodi astratti che devono essere implementati in sottoclassi non astratte;
- può contenere campi e metodi non astratti (inclusi quelli statici);
- una classe astratta può estendere un'altra classe, inclusa una astratta;
- può contenere un costruttore.

Una classe astratta presenta due differenze principali rispetto alle classi normali (concrete): **nessuna istanza** e metodi **astratti**.

I metodi astratti vengono dichiarati aggiungendo la parola chiave *abstract*. Hanno una dichiarazione (modificatori, un tipo di ritorno e una firma) ma non hanno un'implementazione. Ogni sottoclasse concreta (non astratta) deve implementare questi metodi.

! i metodi statici **non** possono essere astratti.

```
import java.awt.*;

public abstract class DrawingTool {

    protected Color color;

    protected DrawingTool(Color color) {
        this.color = color;
    }

    protected Color getColor() {
        return color;
    }

    protected void setColor(Color color) {
        this.color = color;
    }

    public abstract String draw(); // an abstract method
}
```



Le classi e le interfacce astratte sono entrambe strumenti per ottenere l'astrazione che consentono di dichiarare metodi astratti. Non si possono creare istanze di classi e interfacce astratte direttamente, possiamo farlo solo tramite classi che le ereditano.

Utilizzo di classi astratte e interfacce insieme. Spesso interfacce e classi astratte vengono utilizzate insieme per rendere più flessibile una gerarchia di classi. In questo caso, una classe astratta contiene membri comuni e implementa una o più interfacce, mentre le classi concrete estendono la classe astratta e possibilmente implementano altre interfacce.

CLASSI ANONIME

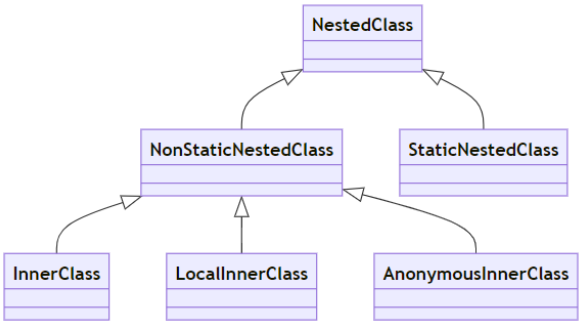
Java fornisce un meccanismo per creare una classe in una singola istruzione senza dover dichiarare una nuova classe denominata. Tali classi sono chiamate anonime perché **non** hanno identificatori di nome. La classe anonima viene dichiarata e istanziata contemporaneamente a un'espressione. Sostituisce entrambi i metodi dell'interfaccia.

```
interface SpeakingEntity {

    void sayHello();

    void sayBye();
}
```

```
SpeakingEntity englishSpeakingPerson = new SpeakingEntity() {

    @Override
    public void sayHello() {
        System.out.println("Hello!");
    }

    @Override
    public void sayBye() {
        System.out.println("Bye!");
    }
};
```

CLASSI NIDIFICATE (NASTED)

È possibile definire una **classe nasted** quando è dichiarata all'interno di un'altra classe.



Classi interne Un'inner class è una classe dichiarata all'interno di un'altra classe (non statica). Ha accesso diretto ai membri della classe esterna, inclusi quelli privati. Può essere istanziata solo in relazione a un'istanza della classe esterna.
Dall'interno della classe interna, si possono vedere tutti i metodi e i campi della classe esterna, anche se sono *private*. Una classe interna è associata a un'istanza della sua classe di contenimento. Quindi, per creare un'istanza di una classe interna e accedervi, devi prima creare un'istanza della classe esterna. Se si crea una classe interna *private,* allora è possibile accedervi solo dall'interno della classe esterna. Lo stesso vale per i campi e i metodi.

```
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public class Bow {  //inner class
        String color;

        public Bow(String color) {
            this.color = color;
        }

        public void printColor() {
            System.out.println("Cat " + Cat.this.name + " has a " + this.color + " bow.");
        }
    }
}
```

Classi nidificate statiche (static nested class)
Una static nested class è una classe dichiarata all'interno di un'altra classe con l'uso della parola chiave static. Non ha accesso diretto ai membri istanza della classe esterna (può accedere solo ai membri statici). Può essere istanziata senza un'istanza della classe esterna.
Ricordati dei modificatori di accesso: se crei una classe nidificata statica private, allora è possibile accedervi solo all'interno della classe esterna. Lo stesso vale per i campi e i metodi.

Classe interna locale (Local inner class)
Una local inner class è una classe definita all'interno di un metodo, di un costruttore o di un blocco di inizializzazione. È visibile solo all'interno del suo ambito e può accedere ai membri della classe esterna e alle variabili locali finali o effettivamente finali del metodo in cui è dichiarata.

```
public class Outer {

    private int number = 10;

    void someMethod() {

        class LocalInner {

            private void print() {
                System.out.println("number = " + Outer.this.number);
            }
        }

        LocalInner inner = new LocalInner();
        inner.print();
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.someMethod();
    }
}
```

Cosa possiamo vedere all'interno della classe interna locale?
Membri della classe esterna, incluso il campo *number*. E variabili locali del blocco di chiusura, come void *someMethod*. Le variabili locali devono essere dichiarate come *final* o essere effettivamente *final*, quest'ultima significa che il loro valore non viene mai modificato dopo l'inizializzazione e non c'è bisogno della parola chiave *final*.

! una classe interna locale può essere istanziata solo all'interno del blocco in cui è definita la classe interna. Quindi altre parti del codice non sanno che esiste.

E4.1 JAVA DATA STRUCTURES

La scelta della struttura dati è fondamentale quando si progetta un software, tra i motivi più importanti:

- **Efficienza algoritmica**, la capacità di risolvere un problema in modo rapido ed efficiente. Questa efficienza dipende da diversi fattori:
    - **Complessità temporale**, il tempo necessario per eseguire l'algoritmo.
    - **Complessità spaziale**, quanta memoria è necessaria per eseguire l'algoritmo.

  Ogni struttura dati è ottimizzata per eseguire determinate operazioni, ad esempio una linked-list è efficiente nell'aggiunta in testa e così via.
- **Utilizzo della memoria**:
    - **Dimensione della struttura dati**, le strutture dati possono richiedere una quantità di memoria variabile, altre richiedono una quantità di memoria fissa e contigua.
    - **Overhead di memoria**, ogni struttura dati ha un certo overhead di memoria associato, che include informazioni di gestione della memoria, altre strutture o informazioni di supporto.

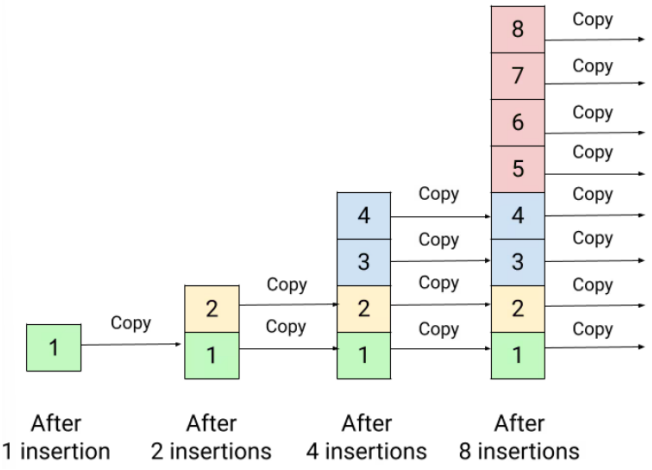
    - **Efficienza di accesso alla memoria**, strutture dati contigue offrono maggiore efficienza nell'accesso all'informazione rispetto a strutture che utilizzano l'ausilio di puntatori.
- **Facilità di implementazione e manutenzione**, alcune strutture dati sono più complesse di altre da utilizzare ed implementare.
- **Adattabilità ai requisiti del problema**, ogni problema ha requisiti unici, quindi, bisogna scegliere la struttura dati in base a:
    - Le operazioni più frequentemente eseguite.
    - Vincoli che possono impedire la scelta dell'utilizzo di una struttura dati.
- **Scalabilità**, scegliere una struttura dati che permetta di mantenere le performance all'aumentare del volume dei dati e che sia adattabile ai cambiamenti.

Java Collection Framework (JCF) è un insieme di classi e interfacce che implementano strutture dati comunemente riutilizzabili.
Il JCF (pacchetto java.util) fornisce

- interfacce che definiscono le funzionalità
- classi astratte per l'aggregazione di codice condiviso
- classi concrete che implementano funzionalità
- algoritmi (java.util.Collections)

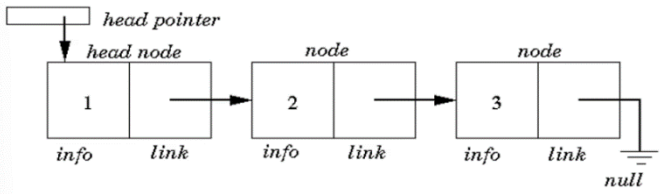
Array Dinamico

Lo svantaggio degli array regolari è che non possiamo modificarne le dimensioni nel mezzo dell'esecuzione del codice. Risolviamo questo problema usando l'idea dell'array dinamico ~O(n) in cui possiamo aumentare la dimensione dell'array dinamicamente quando ne abbiamo bisogno.



Linked list

Una Liked list **~O(n)** è una raccolta lineare di elementi di dati il cui ordine non è dato dalla loro collocazione fisica nella memoria. Invece, ogni elemento punta al successivo. È una struttura di dati costituita da una raccolta di nodi che insieme rappresentano una sequenza. Nella sua forma più elementare, ogni nodo contiene dati e un riferimento (in altre parole, un collegamento) al nodo successivo nella sequenza. Questa struttura consente un inserimento o una rimozione efficiente di elementi da qualsiasi posizione nella sequenza durante l'iterazione. Varianti più complesse aggiungono collegamenti aggiuntivi, consentendo un inserimento o una rimozione più efficiente di nodi in posizioni arbitrarie. Uno svantaggio delle liste concatenate è che il tempo di accesso ai dati è una funzione lineare del numero di nodi per ogni lista concatenata (vale a dire, il tempo di accesso aumenta linearmente man mano che i nodi vengono aggiunti a una lista concatenata), perché i nodi sono collegati in serie; quindi, è necessario accedere a un nodo prima per accedere al nodo successivo (quindi difficile da gestire). Un accesso più rapido, come l'accesso casuale, non è fattibile. Gli array hanno una migliore località della cache rispetto alle liste concatenate.



Binary tree

Un albero binario di ricerca autobilanciante **~O(log(n))** è un albero binario di ricerca basato su nodi che mantiene automaticamente la sua altezza (numero massimo di livelli sotto la radice) piccola di fronte a inserimenti ed eliminazioni di elementi arbitrari. Queste operazioni, quando progettate per un albero binario di ricerca autobilanciante, contengono misure precauzionali contro l'aumento illimitato dell'altezza dell'albero, in modo che queste strutture dati astratte ricevano l'attributo "autobilanciante".

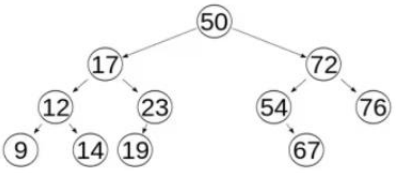
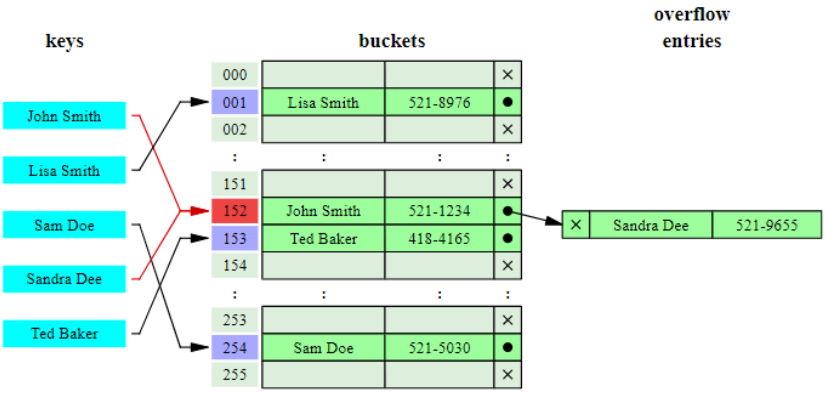
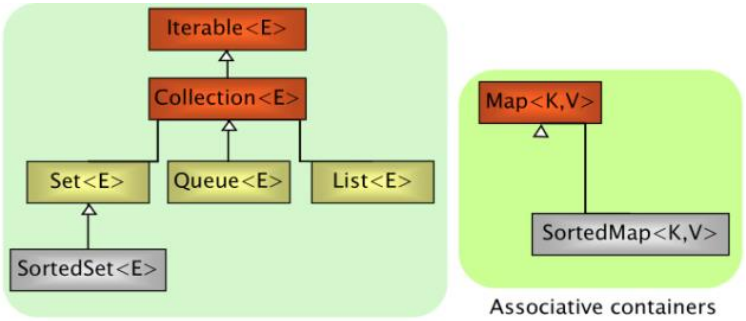
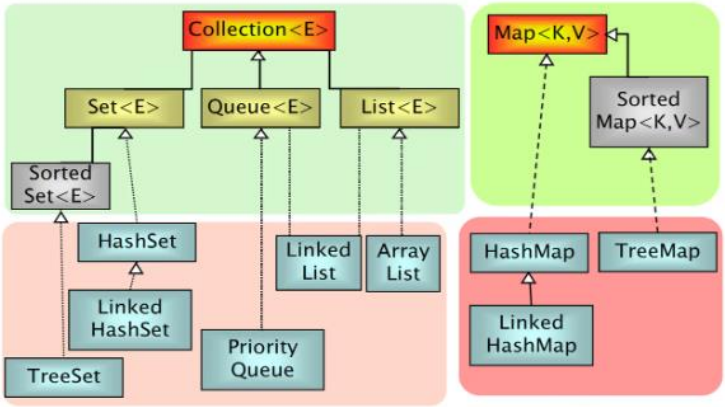


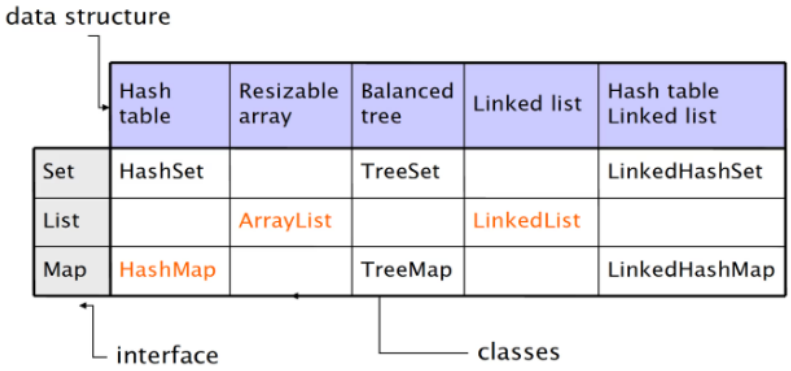
Tabella hash

Una tabella hash **~O(1)** , o hash map o hash map, è una struttura dati che implementa un array associativo o un dizionario. È un tipo di dati astratto che mappa le chiavi ai valori. Una tabella hash utilizza una funzione hash per calcolare un indice, chiamato anche codice hash, in un array di bucket o slot, da cui è possibile trovare il valore desiderato. Durante la ricerca, la chiave viene sottoposta a hash e l'hash risultante indica dove è archiviato il valore corrispondente.









INTERFACCE ITERABILI E ITERATORE

L'interfaccia **Iterable** (java.lang.Iterable) è l'interfaccia radice del framework di raccolta Java. Iterable, letteralmente, significa che "può essere iterato". Tecnicamente, significa che può essere restituito un Iterator . Gli oggetti Iterable (oggetti che implementano l'interfaccia iterable) possono essere utilizzati all'interno di cicli for-each.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
List<Object> l = new ArrayList<Object>();
for(Object o : l){
    // do something
}
```

L' interfaccia **Iterator** estrae il comportamento di attraversamento di una raccolta in un oggetto separato denominato iteratore.

```java
public interface Iterator<T> {
  boolean hasNext();
  T next();
  void remove();
}
```

```java
ArrayList<Object> l = new ArrayList<Object>();
for (Iterator<Object> i = l.iterator(); i.hasNext();) {
    Object o = i.next();
    // do something
}
```

COLLECTION INTERFACE

L'interfaccia root nella gerarchia della collection. Una collectiom rappresenta un gruppo di oggetti, conosciuti come i suoi elementi. Alcune collection consentono elementi duplicati, altre no. Alcuni sono ordinati e altri non ordinati. Il JDK non fornisce alcuna implementazione diretta di questa interfaccia: fornisce implementazioni di sottointerfacce più specifiche come Set e List. Questa interfaccia viene generalmente utilizzata per passare le raccolte e manipolarle laddove si desidera la massima generalità.

Principali metodi di collecton

- int size()
- boolean isEmpty()
- boolean contains(Object element)
- boolean containsAll(Collection c)
- boolean add(Object element)
- boolean addAll(Collection c)
- boolean remove(Object element)
- boolean removeAll(Collection c)
- void clear()
- Object[] toArray()
- Iterator iterator()

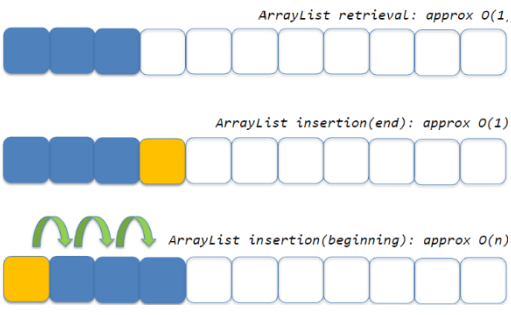
LIST INTERFACE

- Può contenere **elementi duplicati**
- L'ordine di inserimento viene mantenuto
- L'utente può selezionare punti di inserimento arbitrari
- Gli elementi sono accessibili **tramite posizione**

Elenca i metodi principali

- Object get(int index)
- Object set(int index, Object o)
- Object remove(int index)
- void add(int index, Object o)
- boolean addAll(int index, Collection c)
- int indexOf(Object o)
- int lastIndexOf(Object o)
- List subList(int fromIndex, int toIndex)

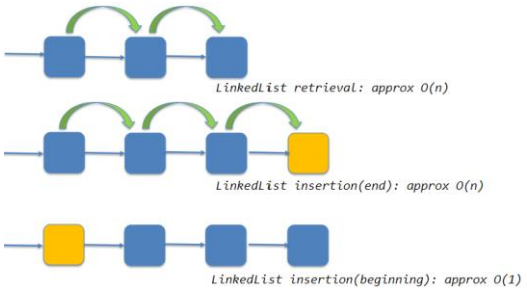
! Il metodo *List.of* in Java è un metodo statico all'interno della classe List che permette di creare una lista immutabile (non modificabile una volta creata) in modo conveniente e conciso.



Esistono due implementazioni List di uso generale:

- **ArrayList** implementa List
  - get(indice) -> Tempo costante
  - add(index, obj) -> Tempo lineare

- **LinkedList** implementa List, Deque
    - get(indice) -> Tempo lineare
    - add(index, obj) -> Tempo lineare (ma più leggero)



La maggior parte delle volte probabilmente utilizzerai ArrayList, che offre accesso posizionale in tempo costante ed è semplicemente veloce. Non è necessario allocare un oggetto nodo per ciascun elemento nell'elenco e può trarre vantaggio da System.arraycopy quando deve spostare più elementi contemporaneamente.
Se aggiungi frequentemente elementi all'inizio dell'Elenco o esegui un'iterazione sull'Elenco per eliminare elementi dal suo interno, dovresti prendere in considerazione l'utilizzo di LinkedList. Queste operazioni richiedono tempo costante in una LinkedList e tempo lineare in un ArrayList. Ma paghi un caro prezzo in termini di prestazioni. L'accesso posizionale richiede tempo lineare in una LinkedList e tempo costante in un ArrayList. Inoltre, il fattore costante per LinkedList è molto peggiore. Se pensi di voler utilizzare una LinkedList, misura le prestazioni della tua applicazione sia con LinkedList che con ArrayList prima di fare la tua scelta. **ArrayList è solitamente più veloce.**

```java
/*--> INZIALIZZAZIONE DI LIST INTERFACE*/
/* plain, simple, long */
List<Integer> l = new ArrayList<>();
l.add(14);
l.add(73);
l.add(18);

/* more compact version (mutable) */
List <Integer> l = new ArrayList<>(Arrays.asList(14, 73, 18));
List <Integer> l = new ArrayList<>(List.of(14, 73, 18));

List <Integer> l = new LinkedList<>(Arrays.asList(14, 73, 18));
List <Integer> l = new LinkedList<>(List.of(14, 73, 18));

/* more compact version (immutable) */
List <Integer > l = List.of(14, 73, 18);
```

SET INTERFACE

Una collection che non contiene elementi duplicati. Più formalmente, gli insiemi non contengono coppie di elementi e1 ed e2 tali che *e1.equals*(*e2*) e al massimo un elemento nullo. Come suggerisce il nome, questa interfaccia modella l'astrazione dell'insieme matematico.
Metodi principali di set: L' interfaccia Set non aggiunge alcun metodo all'interfaccia Collection.

Implementazioni di set:

- **HashSet** implementa Set
    - Tabelle hash come struttura dati interna (veloce!)
    - Ordine di inserimento non conservato
- **LinkedHashSet** estende HashSet
    - Ordine di inserimento conservato
- **TreeSet** implementa SortedSet (un'estensione di Set)
    - Alberi RB come struttura dati interna (fornire l'ordinamento)
    - Ordinamento interno definibile dall'utente TreeSet (Comparatore c)
    - Lento rispetto alle implementazioni basate su hash

```java
/*--> INZIALIZZAZIONE DI SET INTERFACE*/
public static void main(String[] args) {
    List<String> l = List.of("Nicola", "Denise", "Agata", "Marzia", "Agata");

    Set<String> hs = new HashSet<>(l);
    System.out.println(hs);
    // [Denise, Marzia, Nicola, Agata]

    Set<String> lhs = new LinkedHashSet<>(l);
    System.out.println(lhs);
    // [Nicola, Denise, Agata, Marzia]

    Set<String> ts = new TreeSet<>(l);
    System.out.println(ts);
    // [Agata, Denise, Marzia, Nicola]

    /* more compact version (immutable) */
    Set<String> set = Set.of("Nicola", "Denise", "Agata", "Marzia", "Agata");
}
```

Ordinamento interno di TreeSet

A seconda del costruttore utilizzato, i TreeSet possono utilizzare ordinamenti diversi:

- InsiemeAlberi()
  - Ordinamento ascendente naturale
  - Gli elementi devono implementare l' interfaccia comparabile
- TreeSet(Comparatore c)
  - L'ordinamento è definito dal comparatore c

Confronto tra HashSet e TreeSet

HASHSET	TREESET
memorizza gli oggetti in ordine casuale	applica l'ordine naturale degli elementi.
può memorizzare oggetti nulli	non lo consente
garantisce prestazioni in tempo costante per la maggior parte delle operazioni come add(), remove() e contains()	prestazioni in tempo log(n)
Ha meno funzionalità	più ricco di funzionalità, implementando metodi aggiuntivi come: first(), last(), ceiling(), lower(), ...

INTERFACE QUEUE/DEQUE

Queue Una raccolta progettata per contenere elementi prima dell'elaborazione. Oltre alle operazioni di raccolta di base, le code forniscono operazioni aggiuntive di inserimento, estrazione e ispezione. Ciascuno di questi metodi esiste in due forme: uno lancia un'eccezione se l'operazione fallisce, l'altro restituisce un valore speciale (null o false, a seconda dell'operazione).

Deque Una raccolta che supporta l'inserimento e la rimozione di elementi su entrambe le estremità. Il nome deque è l'abbreviazione di "coda a doppia estremità" e viene solitamente pronunciato "mazzo". La maggior parte delle implementazioni di Deque non pongono limiti fissi al numero di elementi che possono contenere, ma questa interfaccia supporta la rimozione dalla coda con capacità limitata così come quelle senza limite di dimensione fissa.

Implementazioni di coda/deque

- **LinkedList** implementa List, Queue, Deque
  - Nessuna limitazione di capacità.
  - Gli elementi non sono ordinati.
- **ArrayDeque** implementa Queue, Deque
  - Nessuna limitazione di capacità.
  - Gli elementi non sono ordinati.

- **PriorityQueue** implementa Queue
  - Nessuna limitazione di capacità.
  - Gli elementi sono ordinati.
- **LinkedBlockingDeque** implementa Queue, Deque
  - Capacità limitata.
  - Gli elementi non sono ordinati.
- **ArrayBlockingQueue** implementa Queue
  - Capacità limitata.
  - Gli elementi non sono ordinati.
- **ConcurrentLinkedDeque** implementa Queue, Deque
  - Nessuna limitazione di capacità.
  - Gli elementi non sono ordinati.

MAP INTERFACE

Un oggetto che associa le chiavi ai valori. Una mappa non può contenere chiavi duplicate; ciascuna chiave può essere associata al massimo a un valore. L'interfaccia Mappa fornisce tre visualizzazioni di raccolta, che consentono di visualizzare il contenuto di una mappa come un insieme di chiavi, una raccolta di valori o un insieme di mappature di valori-chiave. L'ordine di una mappa è definito come l'ordine in cui gli iteratori nelle visualizzazioni di raccolta della mappa restituiscono i propri elementi. Alcune implementazioni di mappe, come la classe TreeMap, forniscono garanzie specifiche riguardo al loro ordine; altri, come la classe HashMap, no.

Metodi principali di map

- Object put(Object key, Object value)
- Object get(Object key)
- Object remove(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- public Set keySet()
- public Collection values()
- public Set entrySet()
- int size()
- boolean isEmpty()
- void clear()

Implementazioni di map

- **HashMap** implementa Map
  - Tabelle hash come struttura dati interna (veloce!)
  - Ordine di inserimento non conservato
- **LinkedHashMap** estende HashMap
  - Ordine di inserimento conservato
- **TreeMap** implementa SortedMap
  - Alberi RB come struttura dati interna
  - Ordinamento interno definibile dall'utente
  - Lento rispetto alle implementazioni basate su hash

```java
/*--> INZIALIZZAZIONE DI MAP INTERFACE*/
Map<Integer, String> src;
src = new HashMap<>();
src.put(77, "Nicola");
src.put(17, "Marzia");
src.put(22, "Agata");
System.out.println(src);
// {17=Marzia, 22=Agata, 77=Nicola}

src = new LinkedHashMap<>();
src.put(77, "Nicola");
src.put(17, "Marzia");
src.put(22, "Agata");
System.out.println(src);
// {77=Nicola, 17=Marzia, 22=Agata}

src = new TreeMap<>();
src.put(77, "Nicola");
src.put(17, "Marzia");
src.put(22, "Agata");
System.out.println(src);
// {17=Marzia, 22=Agata, 77=Nicola}
```

MANIPOLAZIONE DELLE COLLECTIONS

UTILIZZO DEGLI ITERATORI

! Non è sicuro modificare (aggiungendo o rimuovendo elementi) una raccolta mentre si esegue l'iterazione su di essa.

Interface **Iterator** fornisce un mezzo trasparente per scorrere tutti gli elementi di una Collection (solo forward) e rimuovere elementi

- boolean hasNext()
- Object next()
- void remove()

Interface **ListIterator** fornisce un mezzo trasparente per scorrere tutti gli elementi di una Collection (avanti e indietro) e rimuovere e aggiungere elementi

- boolean hasNext()
- boolean hasPrevious()
- object next()
- object previous()
- void add()
- void set()
- void remove()
- int nextIndex()
- int previousIndex()

UTILIZZANDO JAVA.UTIL.COLLECTIONS:

ALTER-EGO DI JAVA.UTIL.ARRAYS PER RACCOLTE

Questa classe contiene vari metodi per manipolare gli array come l'ordinamento, la ricerca, il riempimento, la stampa o la visualizzazione come un array.

Questa classe implementa metodi come:

- sort() unisci l'implementazione dell'ordinamento, n log(n)
- BinarySearch() richiede la raccolta ordinata
- shuffle() mescola la raccolta
- reverse() richiede la raccolta ordinata
- rotate() ruota gli elementi di una determinata distanza
- min(), max() in una raccolta

ORDINAMENTO DELLE COLLECTIONS

Per ordinare raccolte di oggetti, è necessario implementare l'interfaccia **Comparable** per rendere gli oggetti confrontabili tra loro.

L'interfaccia comparabile è implementata di **default** nei tipi comuni nei pacchetti java.lang e java.util

Una raccolta di T può essere ordinata se T implementa Comparable. Il metodo *compareTo*() confronta l'oggetto con l'oggetto passato come parametro. Il valore restituito deve essere:

< 0 se questo oggetto precede obj

== 0 se questo oggetto ha la stessa posizione di obj

> 0 se questo oggetto segue obj

```
public interface Comparable<T> {
    int compareTo(T obj);
}
public interface Comparator<T> {  // raccolta di oggetti da confrontare
    int compare(T obj1, T obj2);
}
// esempio di comparable
public class Person implements Comparable<Person> {
  protected String name;
  protected String lastname;
  protected int age;

  public int compareTo(Person p) {
        // order by surname
        return lastname.compareTo(p.lastname);
  }
}
// esempio di comparator - ordina un insieme di oggetti
public class SortByAge implements Comparator<Person> {
  @Override
  public int compare(Person o1, Person o2) {
    return o1.age - o2.age;
  }
}
```

E5.1 JAVA GENERICS

Come converto una classe o una funzione specializzata su un determinato tipo su un altro tipo? Come faccio a riutilizzare del codice per più tipi diversi?

Esistono situazioni in cui classi e metodi non dipendono strettamente dal tipo, quindi, l'algoritmo che li gestisce è esattamente identico.

Questo problema in Java può essere trattato in più modi:

- Metodi ***SBAGLIATI***:

METODO	Perché è sbagliato?
Copiare il metodo o la classe per ogni tipo che ammette.	Violazione del principio **DRY** (Don't repeat yourself).
Utilizzare la classe Object.	Può provocare **errori di runtime**.

Come funziona Object come tipo generico?

Object è la classe padre, tutte le classi derivano da lui; quindi, è una valida scelta come tipo generico. Tuttavia, quando assegneremo il risultato ad un oggetto più specifico, il compilatore si lamenterà, non sa che tipo di dato viene restituito quindi richiede un cast esplicito. Non si può garantire che l'oggetto restituito sia del tipo indicato e ciò potrebbe causare un **errore di runtime**.

- Metodo ***GIUSTO***: Utilizzo i **Generics**, i quali permettono di risolvere le problematiche runtime e di copia aumentando la chiarezza, la sicurezza ma anche la complessità.

Come risolve il problema della copia?
Con i Generics il programmatore stabilisce il tipo dei dati e il compilatore garantirà che questo vincolo sia rispettato. Questo comporta che l'errore a runtime viene convertito in un errore di compilazione.

PARAMETRI DI TIPO

Un tipo generico è una classe (o interfaccia) che è parametrizzata sui tipi. Per dichiarare una classe generica, dobbiamo dichiarare una classe con la sezione del parametro di tipo delimitata da parentesi angolari < >che seguono il nome della classe. Ricorda che solo un tipo di riferimento può essere utilizzato come tipo concreto per i generici. Ciò significa che invece di tipi primitivi, utilizziamo classi wrapper come Integer, Double, Boolean, e così via.

PARAMETRO DI TIPO SINGOLO: la classe ha un singolo parametro di tipo denominato T.

Sintassi:

```
class GenericType<T> {

    /**
     * A field of "some type"
     */
    private T t;

    /**
     * Takes a value of "some type" and assigns it to the field
     */
    public GenericType(T t) {
        this.t = t;
    }

    /**
     * Returns a value of "some type"
     */
    public T get() {
        return t;
    }

    /**
     * Takes a value of "some type" and assigns it to a field
     */
    public void set(T t) {
        this.t = t;
    }
}
```

Esempi:

```
public static void main(String[] args) {
    GenericType<Integer> obj1 = new GenericType<>(10);
    Integer i = obj1.get();

    GenericType<String> obj2 = new GenericType<>("abc");
    String s = obj2.get();
}
```

PARAMETRI DI TIPO MULTIPLO: Una classe può avere un numero qualsiasi di parametri di tipo, ma la maggior parte delle classi generiche ha solo uno o due parametri di tipo.

Convenzione di denominazione per i parametri di tipo

Esiste una convenzione di denominazione che limita i nomi dei parametri di tipo a singole lettere maiuscole. Senza questa convenzione, sarebbe difficile distinguere la differenza tra una variabile di tipo e un nome di classe ordinario.

I nomi dei parametri di tipo più comunemente utilizzati sono:

T	Tipo
S, U, V ecc.	2°, 3°, 4° tipo
E	Elemento (ampiamente utilizzato nelle collections)
K	Chiave
V	Valore
N	Numero

Type Inference

L'inferenza di tipo è la capacità di un compilatore Java di esaminare ogni invocazione di metodo e la dichiarazione corrispondente per determinare l'argomento di tipo (o gli argomenti) che rendono applicabile l'invocazione. L'algoritmo di inferenza determina i tipi degli argomenti e, se disponibile, il tipo a cui viene assegnato o restituito il risultato. Infine, l'algoritmo di inferenza cerca di trovare il tipo più **specifico** che funziona con tutti gli argomenti.

Generics alla compilazione: I tipi generici sono stati introdotti in Java nel suo secondo decennio, quindi, per garantire la compatibilità con le versioni precedenti, i tipi generici non vengono compilati ma cancellati (**erasure**). Cioè, vengono sostituiti con i normali tipi di Java più generici che rispettino il vincolo e a tempo di esecuzione viene sostituito il tipo generico con un cast.

METODI GENERICI

Metodi statici generici

I metodi statici **non** possono usare parametri di tipo della loro classe. I parametri di tipo della classe a cui appartengono questi metodi possono essere usati solo nei metodi di istanza. Se si vuole usare parametri di tipo in un metodo statico, dichiara i parametri di tipo di questo metodo. La dichiarazione del tipo generico <T> ci consente di utilizzare questo tipo nel metodo.

! Esso può appartenere a una classe generica o non generica perché non ha importanza per i metodi generici.

```
public static <T> T doSomething(T t) {
    return t;
}
public static <E> int length(E[] array) {
    return array.length;
}
```

Metodi di istanza generici

Proprio come i metodi statici, i metodi di istanza possono avere i propri parametri di tipo. Non c'è differenza nella loro dichiarazione rispetto ai metodi statici, escludendo l'assenza della static parola chiave.

```
class SimpleClass {

    public <T> T getParameterizedObject(T t) {
        return t;
    }
}
/*Questa classe non fornisce un parametro di tipo, quindi dobbiamo specificare il parametro di tipo nella
*dichiarazione del metodo per rendere il metodo getParameterizedObject generico.
*!In questo esempio non possiamo usare T come tipo per un campo nella classe, perché appartiene al metodo
*piuttosto che alla classe stessa.
*/
```

LIMITI DI TIPO

Parametri di tipo vincolanti (classi)

Una classe generica, come Stack<E>, può contenere oggetti di qualsiasi tipo. Tuttavia, è possibile limitare i tipi di oggetti che la classe può contenere. Ad esempio, possiamo limitare Stack<E> a contenere solo oggetti che implementano l'interfaccia CharSequence, come String e StringBuilder, usando la sintassi Stack<E extends

CharSequence>. Questo comporta che un'istanza di Stack<String> o Stack<StringBuilder> sarà valida, mentre Stack<Point> provocherà un errore di compilazione.

```
public class Stack<E extends CharSequence> {
    // ...
}
```

Una variabile di tipo può avere un singolo limite o multipli.

```
// singolo limite di tipo
<T extends A>
// limite di tipo multiplo
<T extends A & B & C & ...>
//! Il primo tipo vincolato ("A") può essere una classe o un'interfaccia.
//Il resto dei tipi vincolati (da "B" in poi) devono essere interfacce.
```

Parametri di tipo vincolanti (metodi)

```
public static <T extends Measurable> double average(List<T> objects) {
    // ...
}
//oppure
public static double average(List<? extends Measurable> objects) {
    // ...
}
```

T è un sottotipo del tipo Misurabile (vale a dire, T ha almeno i metodi del tipo Misurabile).
Utilizzare **< ? >** ogni volta che sono assenti vincoli tra parametri o valori di ritorno.

LA REGOLA DI JOSH BLOCH

PECS è un mnemonico che ti aiuta a ricordare quale tipo di carattere jolly usare. Sta per **producer-extends**, **consumer-super** (produttore-estende, consumatore-super).
In altre parole, se un tipo parametrico:

- rappresenta un produttore T, usa <? estende T>
- rappresenta un consumatore T, usa <? super T>

```
public class Stack<E> {
    private List<E> items = new ArrayList<>();

    public void push(E e) {
        items.addLast(e);
    }

    public void pushAll(Iterable<? extends E> src) {
        for (E e : src) {
            push(e);
        }
    }

    public E pop() {
        return items.removeLast();
    }

    public void popAll(Collection<? super E> dst) {
        while (!isEmpty()) {
            dst.add(pop());
        }
    }

    public boolean isEmpty() {
        return items.isEmpty();
    }
}
```

E6.1 - JAVA FUNCTIONAL PROGRAMMING (FUNCTIONS)

Programmazione funzionale vs imperativa

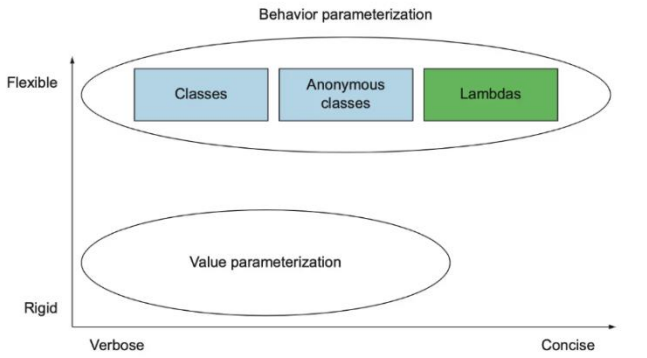
A causa delle mutevoli esigenze, diversi linguaggi (Java, Python, Scala) stanno introducendo modalità per supportare la programmazione funzionale.

Il paradigma della programmazione funzionale descrive un programma applicando e componendo funzioni.

- Passaggio di funzioni alle funzioni (parametrizzazione del comportamento)
- I programmi funzionali possono essere molto concisi ed eleganti, condensando un sacco di comportamenti in poche righe di codice. I programmatori funzionali sostengono che in un mondo multicore, è necessario evitare lo stato mutabile per scalare i programmi.

Il paradigma di programmazione imperativa consente di descrivere un programma in termini di una sequenza di istruzioni che cambiano stato.

- Passaggio di oggetti alle funzioni (parametrizzazione del valore)
- I programmatori orientati agli oggetti ribattono che negli ambienti aziendali reali la programmazione orientata agli oggetti è ben scalabile in termini di sviluppatori e, come settore, sappiamo come realizzarla.



ESPRESSIONI LAMBDA

Con espressione lambda (o semplicemente "una lambda") intendiamo una funzione che non è vincolata al suo nome (una funzione anonima) ma può essere assegnata a una variabile.

La forma più generale di un'espressione lambda è la seguente: *(parameters) -> { body };*

Le parentesi *{ }* sono necessarie solo per le espressioni lambda multi-riga: *(parameters) -> expression*

La parte precedente -> è l'elenco dei parametri (come nei metodi), mentre la parte successiva è il corpo che può restituire un valore.

STRATEGY PATTERN

In effetti, le **interfacce funzionali** o le interfacce che definiscono un **solo metodo** sono candidate ideali per fare uso di espressioni lambda. Le espressioni lambda possono essere usate per fornire l'implementazione del loro singolo metodo.

```
public static List<Student> filterStudents(List<Student> students, StudentPredicate sp, StudentFunction sf, StudentConsumer sc) {
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
        if (sp.test(s)) {
            String str = sf.apply(s);
            sc.accept(str);
            result.add(s);
        }
    }
    return result;
}

public static void main(String[] args) {
    List<Student> result = filterStudents(students,
            s -> s.getAverage() >= 26 && s.getAverage() <= 30,
            s -> String.format("%s_%s_%f", s.getLastname(), s.getName(), s.getAverage()),
            s -> System.out.println(s));
}
```

STRATEGY PATTERN + GENERICS

Utilizzando i generici possiamo generalizzare ulteriormente il metodo che stiamo studiando. Ora può ricevere un generico *Predicate<T>, Function<T,R>, Consumer<R>* e un elenco di tipo T. Ciò consente un altro notevole miglioramento nell'espressività con la stessa quantità di codice. Infatti, può ricevere un elenco generico (non solo un elenco di studenti), filtrarlo in base a un predicato e stampare il risultato della trasformazione.

```
public static <T, R> List<T> filter(List<T> l, Predicate<T> sp, Function<T, R> sf, Consumer<R> sc) {
    List<T> result = new ArrayList<>();
    for (T s : l) {
        if (sp.test(s)) {
            R x = sf.apply(s);
            sc.accept(x);
            result.add(s);
        }
    }
    return result;
}
```

INTERFACCE FUNZIONALI

Funzione: è un'interfaccia funzionale con un metodo che riceve un valore e ne restituisce un altro. Questa funzione è rappresentata dall'interfaccia *Function*, che è parametrizzata dai tipi del suo argomento e da un valore di ritorno:

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

Predicato: è una specializzazione di una funzione che riceve un valore generato e restituisce un valore booleano.

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

Consumatore: il consumer accetta un argomento generico e non restituisce nulla. È una funzione che rappresenta gli effetti collaterali.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Fornitore: il fornitore non accetta nulla e restituisce un risultato generico. Non è richiesto che venga restituito un risultato nuovo o distinto ogni volta che il fornitore viene invocato.

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

SPECIALIZZAZIONI DI FUNZIONI PRIMITIVE

Poiché un tipo primitivo non può essere un argomento di tipo generico, esistono versioni dell'interfaccia Function per i tipi primitivi più utilizzati e le loro combinazioni nei tipi di argomento e di ritorno:

- IntFunction, LongFunction, DoubleFunction: gli argomenti sono del tipo specificato, il tipo di ritorno è parametrizzato;
- ToIntFunction, ToLongFunction, ToDoubleFunction: il tipo di ritorno è del tipo specificato, gli argomenti sono parametrizzati;
- DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, IntToLongFunction, LongToIntFunction, LongToDoubleFunction: con sia l'argomento che il tipo di ritorno definiti come tipi primitivi, come specificato dai loro nomi;

SPECIALIZZAZIONI DI FUNZIONI A TWO-ARITY

Per definire le lambda con due argomenti, dobbiamo utilizzare interfacce aggiuntive che contengono la parola chiave "Bi" nei loro nomi: BiFunction, ToDoubleBiFunction, ToIntBiFunction e ToLongBiFunction.

Ad esempio BiFunction ha sia argomenti che un tipo di ritorno generati, mentre ToDoubleBiFunction e altri ci consentono di restituire un valore primitivo.

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

OPERATORI

Le interfacce operatore sono casi speciali di una funzione che riceve e restituisce lo stesso tipo di valore. L'interfaccia UnaryOperator riceve un singolo argomento.

```
@FunctionalInterface
public interface UnaryOperator<T> {
    T apply(T t);
}
```

RIFERIMENTI AL METODO

Per riferimento al metodo intendiamo una funzione che fa riferimento a un metodo particolare tramite il suo nome e può essere invocata ogni volta che ne abbiamo bisogno. La sintassi di base di un riferimento al metodo è la seguente:

```
objectOrClass :: methodName
```

dove objectOrClasspuò essere un **nome** di **classe** o una particolare **istanza** di una classe.

```
//esempio di confronto
// lambda expression
BiFunction<Integer, Integer, Integer> max = (x, y) -> Integer.max(x, y);
// method reference
BiFunction<Integer, Integer, Integer> max = Integer::max;
```

Tipi di riferimenti al metodo

È possibile scrivere riferimenti a metodi sia statici che di istanza (non statici). In generale, esistono quattro tipi di riferimenti ai metodi:

- riferimento a un metodo **statico**;
- riferimento a un metodo d'**istanza** di un **oggetto esistente**;
- riferimento a un metodo d'istanza di un **oggetto** di un **tipo particolare**;
- riferimento a un **costruttore**.

```
// sintassi di riferimenti:

//      - a metodo STATICO
ClassName :: staticMethodName
//      esempio
Function<Double, Double> sqrt = Math::sqrt;
sqrt.apply(100.0d); // the result is 10.0d

//      - a metodo d'ISTANZA di un OGGETTO ESISTENTE
objectName :: instanceMethodName
//      esempio
String whatsGoingOnText = "What's going on here?";
Function<String, Integer> indexWithinWhatsGoingOnText = whatsGoingOnText::indexOf;
System.out.println(indexWithinWhatsGoingOnText.apply("going")); // 7
System.out.println(indexWithinWhatsGoingOnText.apply("Hi"));    // -1

//      - a metodo d'ISTANZA di un OGGETTO di un TIPO PARTICOLARE
ClassName :: instanceMethodName
//      esempio
Function<Long, Double> converter = Long::doubleValue;
converter.apply(100L); // the result is 100.0d

//      - a un COSTRUTTORE
ClassName :: new
//      esempio
Function<String, Person> personGenerator = Person::new; // produce nuovi Person oggetti in base ai loro nomi
Person johnFoster = personGenerator.apply("John Foster"); // noi abbiamo un oggetto John Foster
```

RIASSUMENDO: IMPLEMENTAZIONE DI INTERFACCE FUNZIONALI

Esistono diversi modi per implementare un'interfaccia funzionale.

- *Classi anonime* (poco chiaro e prolisso)
- *Espressioni lambda* (Il tipo di interfaccia funzionale (sinistra) e il tipo di lambda (destra) sono gli stessi da una prospettiva semantica. I parametri e il risultato di un'espressione lambda corrispondono ai parametri e al risultato di un singolo metodo astratto dell'interfaccia funzionale).
- *Riferimenti al metodo* (il numero e il tipo di argomenti e il tipo di ritorno devono corrispondere al numero e ai tipi di argomenti e al tipo di ritorno del singolo metodo astratto di un'interfaccia funzionale)

VALORI FACOLTATIVI

Java utilizza null per rappresentare l'assenza di un valore, il che può portare a eccezioni come NPE. La classe Optional<T> rappresenta la presenza o l'assenza di un valore di tipo T, fungendo da contenitore che può essere vuoto o contenere un valore. Si possono creare oggetti Optional vuoti o con valore, e metodi come *isPresent* verificano la presenza del valore. Per ottenere il valore, si usano metodi come *get, orElse, orElseGet* e *orElseThrow*. Metodi come *ifPresent* e *ifPresentOrElse* permettono di eseguire azioni condizionali sui valori presenti, riducendo il rischio di errori legati a null.

E6.2 JAVA FUNCTIONAL PROGRAMMING(STREAM)

IL CONCETTO DI BASE DEI FLUSSI

Utilizzando **Stream API** (che fornisce un approccio funzionale all'elaborazione di collections e array), un programmatore non ha bisogno di scrivere cicli espliciti (ad esempio usare for in modo esplicito), poiché ogni flusso ha un ciclo interno ottimizzato.

In un certo senso, uno stream può ricordarci una collezione. Ma in realtà non memorizza elementi. Invece, trasmette elementi da una fonte come una collezione, una funzione di generazione, un file, un canale I/O, un altro stream, e quindi elabora gli elementi utilizzando una sequenza di operazioni predefinite combinate in un'unica pipeline.

Ci sono tre fasi per lavorare con un flusso:

1. Ottenere il flusso da una sorgente.
2. Esecuzione di operazioni intermedie con il flusso per elaborare i dati.
3. Esecuzione di un'operazione terminale per produrre un risultato.

LOOP VS STREAM

Tutte le classi associate ai flussi si trovano nel *java.util.stream* pacchetto. Ci sono diverse classi di flussi comuni: Stream<T>, IntStream, LongStreame DoubleStream. Mentre il flusso generico funziona con i tipi di riferimento, altri funzionano con i tipi primitivi corrispondenti.

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

public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>(List.of(1, 4, 7, 6, 2, 9, 7, 8));

    long count = 0;
    for (int number : numbers) {
        if (number > 5) {
            count++;
        }
    }

    System.out.println(count); // 5

}

// STREAM
long count = numbers.stream()
        .filter(number -> number > 5)
        .count(); // 5
```

Creazione di flussi

```
// stream da COLLECTIONS
List<Integer> famousNumbers = List.of(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55);
Stream<Integer> numbersStream = famousNumbers.stream();

// stream da ARRAY
Stream<Double> doubleStream = Arrays.stream(new Double[]{ 1.01, 1d, 0.99, 1.02, 1d, 0.99 });

// da alcuni valori
Stream<String> persons = Stream.of("John", "Demetra", "Cleopatra");

// concatenando altri flussi insieme
Stream<String> stream1 = Stream.of(/* some values */);
Stream<String> stream2 = Stream.of(/* some values */);
Stream<String> resultStream = Stream.concat(stream1, stream2);
```

GRUPPI DI OPERAZIONI DI FLUSSO

Tutte le operazioni di flusso sono divise in due gruppi:

- Le operazioni **intermedie** non vengono valutate immediatamente quando vengono invocate. Restituiscono semplicemente nuovi flussi per chiamare le operazioni successive su di essi. Tali operazioni sono note come **lazy** perché in realtà non fanno nulla di utile.

Operazioni intermedie	
`filter`	restituisce un nuovo flusso che include gli elementi che corrispondono a un predicato;
`map`	restituisce un nuovo flusso costituito dagli elementi ottenuti applicando una funzione (ovvero trasformando ogni elemento), quindi accetta come parametro una funzione con un argomento; è utilizzato: - per l'estrazione delle proprietà, - per trasformare gli oggetti;
`peek`	restituisce lo stesso flusso di elementi ma consente di osservare gli elementi correnti con un consumatore (solitamente stampare);
`sorted`	restituisce un nuovo flusso che include elementi ordinati secondo l'ordine naturale o un dato comparatore;
`limit`	restituisce un nuovo flusso costituito dai primi n elementi di questo flusso;
`skip`	restituisce un nuovo flusso senza i primi n elementi di questo flusso;
`distinct`	estituisce un nuovo flusso costituito solo da elementi univoci in base ai risultati di *equals*;

Il metodo flatMap: L'operazione di mappa funziona benissimo per flussi di primitive e oggetti, ma l'input può anche essere un flusso di raccolte. Ad esempio, il metodo stream()di a List<List<String>> restituisce a Stream<List<String>>. In tal caso, spesso dobbiamo appiattire un flusso di raccolte in un flusso di elementi da queste raccolte.

Flattening si riferisce alla fusione di elementi di un elenco di elenchi in un singolo elenco. Ad esempio, se appiattiamo un elenco di elenchi [["a", "b"], ["c"], ["d", "e"]], otterremo l'elenco ["a", "b", "c", "d", "e"].

In questi casi, il flatMap metodo può essere utile. Prende e applica una funzione con un argomento per trasformare ogni elemento di flusso in un nuovo flusso e concatena questi flussi insieme.

```java
// esempio del metodo filter
List<Integer> primeNumbers = List.of(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
List<Integer> filteredPrimeNumbers = primeNumbers.stream()
        .filter(n -> n >= 11 && n <= 23)
        .collect(Collectors.toList());

// esempio del metodo map
List<Double> numbers = List.of(6.28, 5.42, 84.0, 26.0);
List<Double> famousNumbers = numbers.stream()
        .map(number -> number / 2)
        .collect(Collectors.toList());

// esempio di utilizzo della mappa per l'estrazione delle proprietà
public class Job {
    private String title;
    private String description;
    private double salary;

    // getters and setters
}
List<String> titles = jobs.stream()
        .map(Job::getTitle)
        .collect(Collectors.toList());

// esempio metodo flatmap
List<String> authors = javaBooks.stream()
        .flatMap(book -> book.getAuthors().stream())
        .collect(Collectors.toList());

// esempio metodo peek
List<Integer> primeNumbers = List.of(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);

List<Integer> primeNumbersCopy = primeNumbers.stream()
        .peek(System.out::println)
        .collect(Collectors.toList());
```

- Le operazioni **terminali** iniziano tutte le valutazioni con lo stream per produrre un risultato o per creare un effetto collaterale. Come abbiamo detto prima, uno stream ha sempre una sola operazione terminale. Una volta che un'operazione terminale è stata valutata, non è possibile riutilizzare di nuovo lo stream (il programma genererà IllegalStateException).

Operazioni terminali	
`count`	restituisce il numero di elementi nel flusso come long valore;
`max/ min`	restituisce un Optional elemento massimo/minimo del flusso in base al comparatore specificato;
`reduce`	combina i valori del flusso in un singolo valore (un valore aggregato);
`findFirst/ findAny`	restituisce il primo / qualsiasi elemento del flusso come Optional;
`anyMatch`	restituisce true se almeno un elemento corrisponde a un predicato (vedi anche: `allMatch, noneMatch`);
`forEach`	prende un consumatore e lo applica a ciascun elemento del flusso (ad esempio, stampandolo);
`collect`	restituisce una raccolta di valori nel flusso; - trasformare in collections - accumulare elementi di flusso in un singolo valore
`toArray`	restituisce un array dei valori in un flusso.

Alcune operazioni del terminale restituiscono un Optional perché il flusso può essere vuoto e, se è vuoto, è necessario specificare un valore predefinito o un'azione.

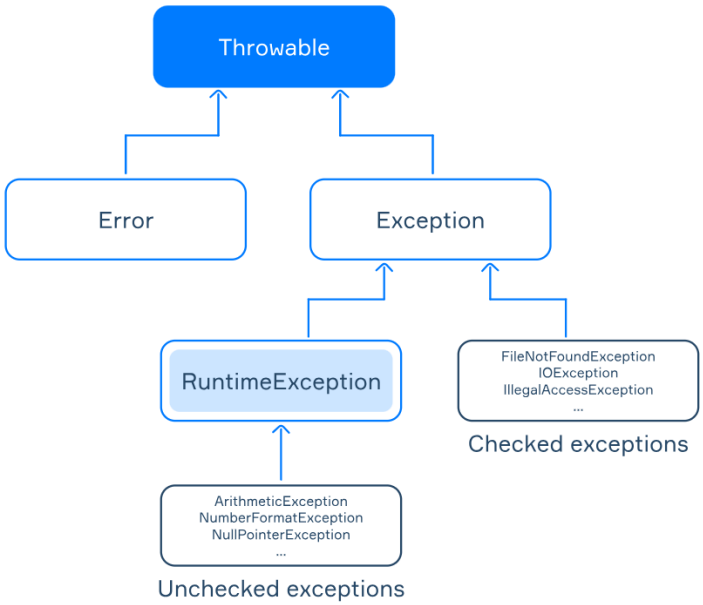
Downstream collectors: Oltre a un predicato o a una funzione di classificazione, partitioningBy i groupingBy collettor possono accettare un collettore downstream. Tale collettor viene applicato ai risultati di un altro collettor. Ad esempio, groupingBy il collettor, che accetta una funzione di classificazione e un collettor downstream, raggruppa gli elementi in base a una funzione di classificazione e quindi applica un collettor downstream specificato ai valori associati a una data chiave.

```java
// esempio metodo collect: trasformando in collections
List<Account> accounts = accountStream.collect(Collectors.toList());
LinkedList<Account> accounts = accountStream.collect(Collectors.toCollection(LinkedList::new)); // collections generica

// esempio metodo collect: accumulare elementi di flusso in un singolo valore
long summary = accounts.stream()
.collect(Collectors.summingLong(Account::getBalance));

// esempio metodo collect: Partizionamento
Map<Boolean, List<Account>> accountsByBalance = accounts.stream()
        .collect(Collectors.partitioningBy(account -> account.getBalance() >= 10000));

// esempio raggrupamento con il metodo Collectors.groupingBy -> produce una map
Map<Status, List<Account>> accountsByStatus = accounts.stream()
        .collect(Collectors.groupingBy(Account::getStatus));

// essempio downstram collectiors
Map<Status, Long> sumByStatuses = accounts.stream()
        .collect(Collectors.groupingBy(
                Account::getStatus,
                Collectors.summingLong(Account::getBalance))
         );
```

E7.1 JAVA EXCEPTIONS

Le **eccezioni** sono errori catturati durante **l'esecuzione** di un programma e che ne interrompono la normale esecuzione. Non sono funzionali per l'utente finale che utilizza il programma ma sono essenziali per il programmatore per avere maggiore comprensione degli errori.

Le eccezioni sono oggetti di una classe speciale organizzate, come tutte le classi, in gerarchie:



Dove:

Throwable, è la classe padre, fornisce una struttura comune per rappresentare le anomalie che si verificano durante l'esecuzione del programma.

- **Error**, rappresenta errori gravi ed irrecuperabili che possono riguardare la JVM o l'ambiente operativo sottostante.
- **Exception**, rappresenta situazioni anomale relative ad esempio a:
  - *Errori di logica*.
  - *Input non valido*.
  - *Risorse non disponibili*.

Si suddivide a sua volta in:

- **Unchecked exceptions** (*eccezioni non controllate*), sono eccezioni che non richiedono una gestione esplicita del programmatore.

Java

```java
public static void main(String[] args) {
        int[] array = {1, 2, 3};
        int elemento = array[87]; // Genera ArrayIndexOutOfBoundsException
    }
```

- **Checked exceptions** (*eccezioni controllate - RunTimeException*), un meccanismo attraverso il quale il compilatore impone al programmatore di gestire le eccezioni che si possono verificare durante l'esecuzione del programma.
  Se un metodo lancia un'eccezione controllata, deve essere contrassegnata nella dichiarazione dalla parola chiave **throws** altrimenti il programma non verrà compilato.

! Throws

Questa parola chiave viene utilizzata nella dichiarazione di un metodo per segnalare che il blocco di codice potrebbe lanciare una particolare eccezione ma **NON** la gestirà.
Quindi throws permetterà di propagare l'errore al chiamante che a sua volta avrà due scelte:

- Contenere throws nella dichiarazione e far **propagare** ancora **l'eccezione** (risalire la gerarchia delle chiamate).
- **Gestire l'eccezione**.

```java
// esempio generazione di un eccezione controllota
public static String readTextFromFile(String path) throws IOException {
    // find a file by the specified path

    if (!found) {
        throw new IOException("The file " + path + " is not found");
    }

    // read and return text from the file
}

//Se un metodo genera due o più eccezioni controllate, queste devono essere separate da una virgola nella dichiarazione:
public static void method() throws ExceptionType1, ExceptionType2, ExceptionType3 {
  // ...
}
```

GESTIONE DELLE ECCEZIONI

Sia le checked che le unchecked possono essere intercettate e controllate manualmente.
Quindi, quando parliamo di eccezioni controllate, fondamentale è il concetto di **delega**, che può essere:

- **Nulla**, un metodo che potrebbe generare un'eccezione sceglie di non gestirlo e non propagarlo.
  A volte l'eccezione può essere ignorata perché non rilevante e non genera conseguenze.
- **Parziale**, un metodo gestisce l'eccezione in parte.
  Modificare o aggiungere informazioni sull'eccezione: prima di propagarla si trasforma in una eccezione più significativa prima di propagarla.
- **Completa**, un metodo trasferisce completamente la responsabilità di gestione dell'eccezione al chiamante.
  Permette ad un metodo di concentrarsi solo sulle proprie operazioni senza doversi preoccupare di gestire le eccezioni.

L'istruzione try-catch è un modello per gestire le eccezioni:

```
try {
    // codice che potrebbe generare un eccezione
} catch (Exception e) {
    // codice che gestisce per il tipo di eccezione specificato e tutte le sue sottoclassi (questo blocco
    // viene eseguito quando si verifica un'eccezione del tipo corrispondente nel tryblocco)
}
```

```
try {
    // code that throws exceptions
} catch (IOException e) {
    // handling the IOException and its subclasses
} catch (Exception e) {
    // handling the Exception and its subclasses
}
```

! Il blocco catch con la classe base deve essere scritto sotto tutti i blocchi con sottoclassi. In altre parole, i gestori più specializzati (come IOException) devono essere scritti prima di quelli più generali (come Exception). Altrimenti, il codice non compilerà.

L'istruzione try-catch-finally: tutte le istruzioni presenti nel blocco finally verranno sempre eseguite indipendentemente dal fatto che si verifichi o meno un'eccezione nel tryblocco. Il blocco finally è utile per la chiusura delle risorse

```
try {
    // code that may throw an exception
} catch (Exception e) {
    // exception handler
} finally {
    // code that will always be executed
}
```

Best practice per le eccezioni personalizzate

- Assicurati che la tua applicazione trarrà vantaggio dalla creazione di un'eccezione personalizzata. Altrimenti, usa le eccezioni Java standard.
- Seguire la convenzione di denominazione, ad esempio terminare il nome della classe con "Exception", ad esempioMyAppException
- In generale, è consigliabile utilizzare eccezioni standard ove possibile per una serie di motivi, quali:
    - Le eccezioni standard sono ampiamente note ad altri programmatori. Si può capire il tipo di problema semplicemente guardando il nome dell'eccezione.
    - Scegliendo eccezioni standard, segui il principio di riutilizzabilità. Rende il tuo codice più chiaro e professionale.

La prassi migliore è quella di avvolgere qualsiasi codice che gestisca risorse di sistema con la costruzione **try-with-resources.**

```
try (Reader reader = new FileReader("file.txt")) {
    // some code
}
```

Eccezioni e cicli

```
// Per errori che interessano una singola iterazione (o elementi di una collections),
// i blocchi try-catch sono annidati nel ciclo.
// In caso di eccezione, l'esecuzione passa al blocco catch e poi procede con l'iterazione successiva.
while (something) {
  try {
    // potential exceptions
  } catch (Exception e) {
    //  discard iteration
  }
}

// Per errori che compromettono l'intero ciclo, il ciclo viene annidato nel blocco try.
// In caso di eccezione, l'esecuzione passa al blocco catch, uscendo così dal ciclo.
try {
  while (something) {
    //  potential exceptions
  }
} catch (Exception e) {
  // discard whole loop
}
```

Quando è necessario generare un'eccezione?

La risposta non è sempre ovvia. La prassi comune è lanciare un'eccezione quando e **solo** quando le **precondizioni del metodo sono violate**, ovvero quando non può essere eseguito nelle condizioni correnti.

E9.1 JAVA PLATFORM THREADS

PROSPETTIVA DEL SISTEMA OPERATIVO

Processi

Nei sistemi operativi, un processo è un'istanza di un'applicazione in esecuzione. Un processo ha il suo spazio di indirizzamento privato, codice, dati, file aperti, PID, ecc... I processi non condividono la memoria (spazi di indirizzamento separati), quindi devono comunicare tramite meccanismi IPC offerti dal sistema operativo (ad esempio, pipe, segnali, ecc...).

In Java, non è possibile chiamare esplicitamente la syscall fork() come in C. Le chiamate di sistema fork() ed exec() possono essere chiamate congiuntamente tramite la classe java.lang.Process . I metodi della classe java.lang.Process consentono inoltre agli sviluppatori di acquisire input standard, output, errore e valore di uscita del processo avviato.

Thread

I thread sono spesso chiamati **processi leggeri**. Come i processi, ogni thread ha il suo stack, il suo contatore di programma e le sue variabili locali. Tuttavia, i thread all'interno dello stesso processo condividono lo stesso spazio di indirizzamento e, di conseguenza, possono condividere variabili e oggetti.

La condivisione delle variabili è un modo semplice e veloce utilizzato dai thread per comunicare, ma spesso causa bug non visibili nei programmi a thread singolo

JVM e sistema operativo

Un sistema operativo multitasking assegna il tempo di CPU (time slice) ai processi/thread tramite un componente del kernel chiamato scheduler. Piccole time-slice (5-20 ms) forniscono l'illusione di parallelismo di diversi processi/thread (sulle macchine multicore è un'illusione parziale).

La JVM è un processo e ottiene la CPU assegnata dallo scheduler del sistema operativo. Tuttavia, Java è una **specifica** con molte implementazioni. La maggior parte delle JVM usa lo scheduler del sistema operativo (un

thread Java è in realtà mappato a un thread di sistema), mentre alcune operano come un mini-sistema operativo e pianificano i propri thread.

Perché i thread? Le ragioni principali per usare i thread:

- mantenere le interfacce utente reattive durante l'esecuzione di operazioni in background
- sfruttare i sistemi multiprocessore

In un processo **multithread** le azioni vengono eseguite su thread diversi :

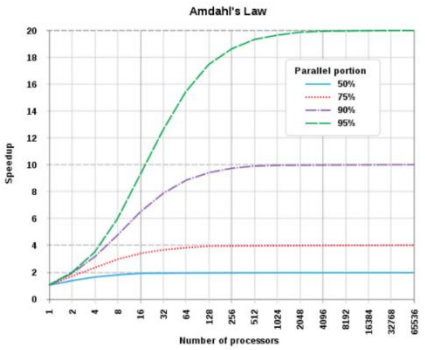
- il download può essere eseguito in background (ad esempio in un altro thread)
- anche l'analisi può essere eseguita in background (ovvero eventualmente in più thread)
- l'utente può usufruire di un'interfaccia utente reattiva mentre attende le notifiche (segnali di acquisto/vendita)

Vantaggi	Svantaggi

- Abilita il parallelismo
- Più leggero dei processi per entrambi
  o Creazione(ad esempio, fork())
  o Comunicazione (ad esempio, pipe r/w ...)
	- Difficile per la maggior parte dei programmatori
	- Anche per gli esperti, lo sviluppo è spesso doloroso
	- I thread interrompono l'astrazione: non è possibile progettare moduli in modo indipendente.

La legge di Amdahl

Nell'architettura dei computer, la legge di Amdahl è una formula che fornisce l'accelerazione teorica dell'esecuzione di un compito che può essere attesa da un sistema le cui risorse sono migliorate. Afferma che *il miglioramento complessivo delle prestazioni ottenuto ottimizzando una singola parte di un sistema è limitato dalla frazione di tempo in cui la parte migliorata è effettivamente utilizzata*.



THREAD IN JAVA

Ogni programma Java ha almeno un thread, chiamato **thread principale**, creato automaticamente dal processo JVM per eseguire istruzioni all'interno del metodo principale. Tutti i programmi Java hanno anche altri thread predefiniti (ad esempio, un thread separato per il garbage collector).

Durante le fasi di sviluppo del linguaggio Java, l'approccio al multithreading è cambiato verso l'uso di astrazioni di alto livello (ad esempio, Executor, Task, ecc

Ogni thread ha un nome, un identificatore, una priorità e alcune altre caratteristiche che possono essere ottenute tramite i suoi metodi.

Un thread **daemon** (che deriva dalla terminologia UNIX) è un thread a bassa priorità che viene eseguito in background per eseguire attività come la garbage collection e così via. JVM non attende i thread daemon prima di uscire, mentre attende i thread non daemon.

Creazione di thread

Java offre due metodi principali per creare un nuovo thread:

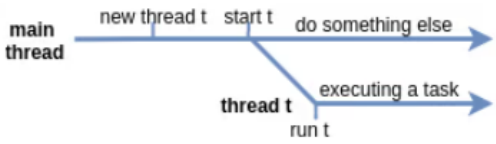
1. estendendo la classe Thread e sovrascrivendo il suo metodo run;
2. implementando l'interfaccia Runnable e passando l'implementazione al costruttore della classe Thread.

In entrambi i casi possono essere usate le funzioni lambda.

```java
// creazione di thread 1° metodo
class HelloThread extends Thread {

    @Override
    public void run() {
        String helloMsg = String.format("Hello, i'm %s", getName());
        System.out.println(helloMsg);
    }
}

// creazione di thread 2° metodo
class HelloRunnable implements Runnable {

    @Override
    public void run() {
        String helloMsg = String.format("Hello, i'm %s", Thread.currentThread().getName());
        System.out.println(helloMsg);
    }
}


Thread t1 = new HelloThread(); // a subclass of Thread
Thread t2 = new Thread(new HelloRunnable()); // passing runnable

// utilizzando le funzioni lambda
Thread t3 = new Thread(() -> {
    System.out.println(String.format("Hello, i'm %s", Thread.currentThread().getName()));
});
```

La classe Thread ha un metodo chiamato **start**() che viene utilizzato per avviare un thread. A un certo punto dopo aver invocato questo metodo, il metodo run verrà invocato automaticamente, ma non accadrà immediatamente.

```java
// esempio
Thread t = new HelloThread(); // an object representing a thread
t.start();
// stampa:
Hello, i'm Thread-0
```



Se si tenta di avviare un thread più di una volta, il metodo start genera *IllegalThreadStateException*.

Ordine di esecuzione

Nonostante il fatto che all'interno di un singolo thread tutte le istruzioni vengano eseguite in sequenza, è impossibile determinare l'ordine relativo delle istruzioni tra più thread senza misure aggiuntive.

Riassumendo:

- Non è garantito che l'esecuzione dei thread venga avviata nello stesso ordine in cui sono stati chiamati i relativi metodi start().
- Non è garantito che un thread continui l'esecuzione finché non ha terminato (non è garantito che il suo ciclo venga completato prima che inizi un altro thread)
- Niente è garantito, tranne: ogni thread verrà avviato e verrà eseguito fino al completamento dopo aver acquisito la CPU un numero finito di volte

Priorità del thread

Per impostazione predefinita, un thread ottiene la priorità del thread che lo crea. I valori di priorità sono definiti tra 1 e 10.

- Thread.MIN_PRIORITY (== 1)
- Thread.NORM_PRIORITY (== 5)
- Thread.MAX_PRIORITY (== 10)

La priorità può essere impostata utilizzando il metodo **setPriority**().

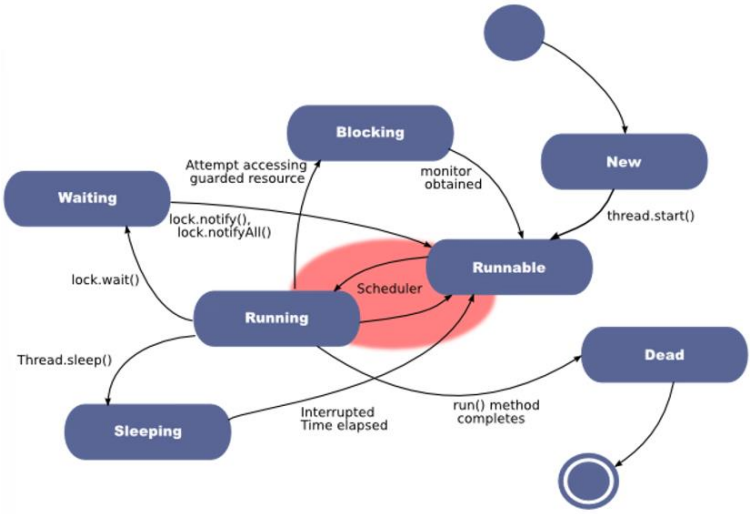
Terminare i thread

Se il thread padre termina, terminano anche tutti i thread figlio.

- I thread figlio condividono risorse con il thread padre, comprese le variabili. Quando il thread padre termina, i thread figlio non saranno in grado di accedere a quelle risorse di cui il thread padre è proprietario.
- Pertanto, se il thread padre termina prima dei propri thread figlio, sono necessari meccanismi di sincronizzazione.

STATI DEL THREAD

- **In esecuzione:** il thread è stato selezionato (dal pool eseguibile) come thread attualmente in esecuzione.
- **Runnable:** un thread che è idoneo all'esecuzione, ma che lo scheduler non ha selezionato come thread in esecuzione. Un thread entra per la prima volta nello stato runnable quando viene invocato il metodo start() .
- **In attesa:** un thread che può acquisire una risorsa ma non c'è lavoro da fare.
- **Bloccante:** un thread in attesa di una risorsa, ad esempio in attesa del completamento delle operazioni di I/O.
- **Inattivo:** un thread che è inattivo dopo una chiamata esplicita al metodo sleep() .



Uscire dallo stato di esecuzione

Esistono 4 modi per lasciare lo stato di esecuzione sotto il controllo del programmatore:

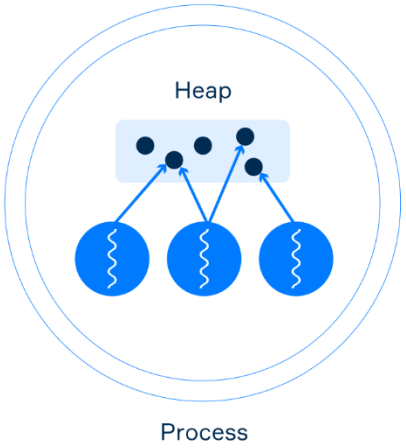
- **sleep()** : il thread attualmente in esecuzione interrompe l'esecuzione per almeno la durata di sospensione specificata in millisecondi.
- **yield()** : il thread attualmente in esecuzione torna allo stato eseguibile e lascia spazio ad altri thread (va allo stato eseguibile). Viene utilizzato quando il calcolo non è possibile (nessun lavoro da fare) in un dato intervallo di tempo. Uno dei vantaggi di questo approccio è rendere il codice meno dipendente dal tipo di scheduler, perché i thread lasciano CPU quando necessario.
- **join()** : forza il thread corrente ad attendere il completamento del thread per cui è stato chiamato il metodo join (passa allo stato di attesa)
- **wait()** : il thread attualmente in esecuzione ha acquisito risorse, ma non è in grado di fare nulla (passa allo stato di attesa)

Esistono altri modi per uscire dallo stato di esecuzione che non sono sotto il controllo del programmatore:

- fine del metodo run() (vai allo stato morto)
- sospeso dallo scheduler del sistema operativo (passa allo stato eseguibile)
- operazioni di I/O bloccanti (vai allo stato di blocco)

```
// SLEEP
Thread.sleep(2000L); // suspend current thread for 2000 millis
TimeUnit.MILLISECONDS.sleep(2000) // nel pacchetto java.util.concurrent

// JOIN
thread.start(); // start thread
System.out.println("Do something useful");
thread.join();  // waiting for thread to die
// waits at most two seconds
thread.join(2000L);
```

INTERFERENZA DEL THREAD



I thread che appartengono allo stesso processo condividono la memoria comune (chiamata Heap). Comunicano utilizzando dati condivisi in memoria. Per poter accedere agli stessi dati da più thread, ogni thread deve avere un riferimento a questi dati.

Una condizione di competizione è un problema che si verifica quando due o più thread condividono la stessa risorsa e uno dei thread interviene troppo velocemente prima che un altro thread abbia completato le sue operazioni (che dovrebbero essere atomiche).

Una porzione di codice che deve essere eseguita come se fosse atomica (lo scheduler non può interrompere un thread durante un'operazione atomica) è chiamata **sezione critica** (ad esempio un prelievo bancario deve essere considerato una sezione atomica).

Cosa sono le operazioni atomiche native (natively atomic operations)? Le operazioni atomiche native sono operazioni che sono garantite per essere eseguite come un'unica operazione indivisibile. In Java, le seguenti operazioni sono atomiche native:

- Lettura o scrittura su un riferimento (tranne per i riferimenti lunghi e doppi su JVM a 32 bit) - esempio: int i=5;
- Lettura o scrittura su una variabile volatile - esempio: volatile double x = 2.0;

Le seguenti operazioni **non** sono nativamente atomiche:

- Lettura o scrittura su un riferimento lungo o doppio su una JVM a 32 bit
- Lettura o scrittura su una variabile non volatile
- Lettura o scrittura su una variabile volatile se la lettura o la scrittura non è l'unica operazione eseguita sulla variabile - esempio: i++ non è atomico perché è equivalente a i = i + 1 che è due operazioni

SYNCHRONIZED

La parola chiave synchronized viene utilizzata per proteggere le risorse a cui si accede contemporaneamente. Solo un thread alla volta può accedere a una risorsa synchronized. Il modificatore synchronized può essere applicato a un metodo o a un oggetto.

Più specificamente, ogni oggetto in Java ha UN blocco incorporato. Entrare in un metodo non statico synchronized significa ottenere il blocco dell'oggetto. Se un thread ottiene il blocco, tutti gli altri thread devono attendere di entrare in TUTTI i metodi synchronized finché il blocco non viene rilasciato (il primo thread esce dal metodo synchronized).

Tutti i metodi sincronizzati di un oggetto condividono lo stesso lock (serratura). Ogni volta che un blocco di oggetto è stato acquisito da un thread, altri thread possono ancora accedere ai metodi non sincronizzati della classe. I metodi che non accedono a dati critici non devono essere sincronizzati.

Dettagli:

- I thread che vanno in modalità sospensione non rilasciano i blocchi.
- Un thread può acquisire più di un lock. Ad esempio, un thread può entrare in un metodo sincronizzato, quindi invocare immediatamente un metodo sincronizzato su un altro oggetto (soggetto a deadlock).

Sincronizzazione tramite oggetti thread-safe

Esistono due modi principali per concedere l'accesso atomico a un oggetto condiviso:

- Utilizzare il modificatore *synchronized all'interno del metodo run()* di ciascun thread per bloccare l'oggetto condiviso.
- Utilizzare il modificatore *synchronized* all'interno dei metodi dell'oggetto condiviso stesso.

Una classe thread-safe è una classe che è sicura (funziona correttamente) quando vi accedono più thread. Le sezioni critiche (ovvero, le sezioni che si suppone siano atomiche) sono incapsulate in blocchi sincronizzati.

- Interface List: Vector (safe)
- Interface Queue: LinkedBlockingQueue, ArrayBlockingQueue, ConcurrentLinkedQueue (safe)
- Interface Map: ConcurrentMap (safe)

```
// ESEMPIO
static class ProducerSafe extends Thread {
    final Deque<Integer> integerDeque;

    public ProducerSafe(Deque<Integer> integerDeque) {
        super();
        this.integerDeque = integerDeque;
    }

    @Override
    public void run() {
        int i = 0;
        while (!isInterrupted()) {
            synchronized (integerDeque) {       // versione UNSAFE
                integerDeque.addFirst(i++);     // integerDeque.addFirst(i++);
            }                                   //
        }
    }
}
static class ConsumerSafe extends Thread {
    final Deque<Integer> integerDeque;

    public ConsumerSafe(Deque<Integer> integerDeque) {
        super();
        this.integerDeque = integerDeque;
    }

    @Override
    public void run() {
        while (!isInterrupted()) {
            try {
                synchronized (integerDeque) {   // versione UNSAFE
                    integerDeque.removeLast();  // integerDeque.removeLast();
                }                               //
            } catch (NoSuchElementException e) {
                Thread.yield();
            }
        }
    `
}
public static void runExperiment(Thread producer, Thread consumer) throws InterruptedException {
    producer.start();
    consumer.start();

    Thread.sleep(100L);

    producer.interrupt();
    consumer.interrupt();

    producer.join();
    consumer.join();
}

public static void main(String[] args) throws InterruptedException {
    /* A safe shared object do not require safe threads */
    Deque<Integer> dq = new ConcurrentLinkedDeque<>();
    Thread p = new ProducerUnsafe(dq);
    Thread c = new ConsumerUnsafe(dq);

    runExperiment(p, c);
}
```

Sincronizzazione tramite metodi Object

I thread potrebbero essere in grado di acquisire l'accesso esclusivo a una risorsa condivisa ma non essere comunque in grado di progredire. Ad esempio, un produttore con una coda piena o un consumatore con una coda vuota.

Per evitare sprechi di risorse computazionali possiamo utilizzare i metodi (ereditati da Object):

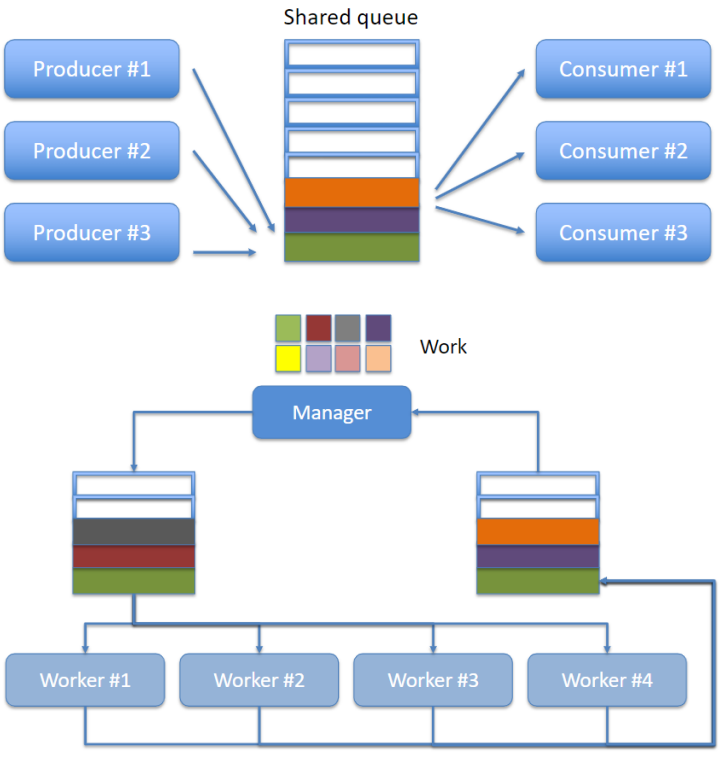
- **wait()** può essere chiamato solo da un blocco sincronizzato. Rilascia il blocco sull'oggetto in modo che un altro thread possa intervenire e acquisire un blocco. wait() consente a un thread di dire: "Non c'è niente che io debba fare ora, quindi mettimi nel pool di attesa e avvisami quando succede qualcosa che mi interessa".
- **notify()** invia un segnale a uno dei thread in attesa nel pool di attesa dell'oggetto. Il metodo notify() NON PUÒ specificare quale thread in attesa notificare. Il metodo **notifyAll()** è simile ma invia un segnale a tutti i thread in attesa sull'oggetto. notify() consente a un thread di dire: "Qualcosa è cambiato qui. Sentiti libero di continuare quello che stavi cercando di fare".

```
static class ProducerSafeWaitNotify extends Thread {
    final Deque<Integer> integerDeque;
    final int maxDequeSize = 10;

    public ProducerSafeWaitNotify(Deque<Integer> integerDeque) {
        super();
        this.integerDeque = integerDeque;
    }

    @Override
    public void run() {
        int i = 0;
        while (!isInterrupted()) {
            synchronized (integerDeque) {
                if (integerDeque.size() < maxDequeSize) {
                    integerDeque.addFirst(i++);
                    integerDeque.notifyAll();
                } else {
                    try {
                        integerDeque.wait();
                    } catch (InterruptedException e) {

                    }
                }
            }
        }
    }
}
```

MODELLI DI PROGETTAZIONE MULTI-THREAD

Nonostante i thread possano essere utilizzati in numerosi scenari del mondo reale, la maggior parte di essi può essere concettualmente assimilata a due casi principali:

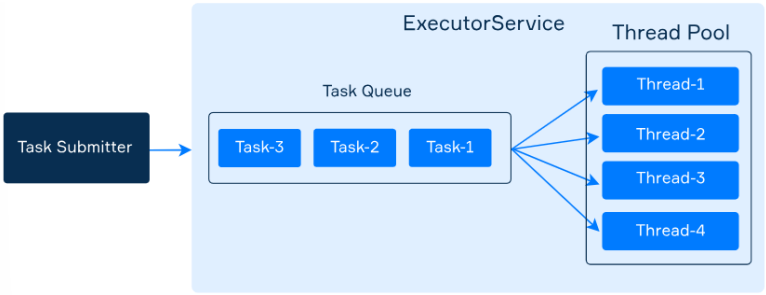
- Il modello **produttore/consumatore**, in cui il thread produttore spinge gli elementi in un oggetto condiviso e il thread consumatore li recupera (li consuma).
- Il modello **manager/workers**, in cui un manager scompone un task complesso in sottotask e li assegna ai thread worker.



EXECUTORSERVICE

Per semplificare lo sviluppo di applicazioni multithread, Java fornisce un'astrazione chiamata ExecutorService (o semplicemente **executor**). Incapsula uno o più thread in un singolo pool e inserisce le attività inviate in una coda interna per eseguirle utilizzando i thread.

Questo approccio isola chiaramente i task dai thread e ti consente di concentrarti sui task. Non bisogna preoccuparsi di creare e gestire i thread perché lo fa l'executor.



CREAZIONE DI EXECUTORS

Tutti i tipi di executors si trovano nel pacchetto java.util.concurrent . Questo pacchetto contiene anche una comoda classe di utilità Executors per creare diversi tipi di ExecutorService. Abbiamo preso in considerazione l'esecutore più utilizzato con la dimensione fissa del pool. Ecco alcuni altri tipi:

- Un esecutore **con un singolo thread** - L'esecutore più semplice ha un solo thread nel pool. Potrebbe essere sufficiente per l'esecuzione asincrona di task raramente inviati e di piccole dimensioni.
- Un esecutore **con un pool fisso** - Può eseguire più task contemporaneamente e velocizzare il programma eseguendo calcoli in qualche modo paralleli. Se uno dei thread muore, l'esecutore ne crea uno nuovo. Più avanti considereremo come determinare il numero di thread richiesto.
- Un esecutore **con un pool in crescita**: esiste anche un esecutore che aumenta automaticamente il numero di thread in base alle necessità e riutilizza i thread creati in precedenza. Può in genere migliorare le prestazioni dei programmi che eseguono molte attività asincrone di breve durata. Ma può anche portare a problemi quando il numero di thread aumenta troppo. È preferibile scegliere l'esecutore del pool di thread fisso ogni volta che è possibile.
- Un esecutore **che pianifica un'attività**: se è necessario eseguire la stessa attività periodicamente o solo una volta dopo il ritardo specificato.

```
// esecutore a singolo thread
ExecutorService executor = Executors.newSingleThreadExecutor();

// esecutore con pool fisso
ExecutorService executor = Executors.newFixedThreadPool(4);

// esecutore con pool in crescita
ExecutorService executor = Executors.newCachedThreadPool();

// esecutore che pianifica un'attività
ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
```

Invio di attività

Un esecutore ha il metodo submit che accetta un task Runnable da eseguire. Poiché Runnable è un'interfaccia funzionale, è possibile usare un'espressione lambda come task.

```
executor.submit(() -> System.out.println("Hello!"));
```

Dopo aver invocato submit, il thread corrente non attende il completamento dell'attività. Aggiunge semplicemente l'attività alla coda interna dell'esecutore per essere eseguita in modo asincrono da uno dei thread.

Fermare gli executors

Un esecutore continua a lavorare dopo il completamento di un'attività poiché i thread nel pool sono in attesa di nuove attività in arrivo. Il tuo programma non si fermerà mai finché almeno un esecutore continua a lavorare. Esistono due metodi per fermare gli esecutori:

1. *void shutdown()* attende che tutte le attività in esecuzione siano completate e impedisce l'invio di nuove attività. Nota: *shutdown()* non blocca il thread corrente a differenza di *join()* di Thread . Se devi attendere che l'esecuzione sia completata, puoi invocare *awaitTermination(...)* con il tempo di attesa specificato.
2. *List shutdownNow()* arresta immediatamente tutte le attività in esecuzione e restituisce un elenco delle attività in attesa di esecuzione.

Se non si sa quanti thread sono necessari nel pool, è possibile utilizzare il numero di processori disponibili come dimensione del pool.

```
int poolSize = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(poolSize);
```

La gestione delle eccezioni: è prassi comune racchiudere un task nel blocco try-catch per non perdere l'eccezione.

CALLABLE AND FUTURE

A volte non è necessario solo eseguire un task in un esecutore, ma anche restituire un risultato di questo task al codice chiamante. È possibile ma scomodo con Runnable. Per semplificare, un esecutore supporta un'altra classe di task denominata Callable che restituisce il risultato e può generare un'eccezione. Questa interfaccia appartiene al pacchetto java.util.concurrent.

Callable è un'interfaccia generica in cui il parametro di tipo V determina il tipo del risultato. Poiché è un'interfaccia **funzionale**, possiamo usarla insieme alle espressioni lambda e ai riferimenti ai metodi.

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

Quando inviamo un *Callable* a un servizio executor, non può restituire un risultato direttamente poiché il metodo *submit* non attende il completamento dell'attività. Invece, un executor restituisce un oggetto speciale chiamato Future che racchiude il risultato effettivo che potrebbe non esistere ancora. Questo oggetto rappresenta il risultato di un calcolo asincrono (attività).

Finché l'attività non viene completata, il risultato effettivo non è presente nell'oggetto future. Per verificarlo, c'è un metodo *isDone*().

Il risultato può essere recuperato da un future solo utilizzando il metodo *get*. Il **metodo get** restituisce il risultato quando il calcolo è completato, oppure blocca il thread corrente e attende il risultato. Questo metodo può generare due eccezioni controllate: ExecutionException e InterruptedException.

Se un task inviato esegue un loop infinito o attende una risorsa esterna per troppo tempo, un thread che invoca get verrà bloccato per tutto questo tempo. Per impedirlo, esiste anche una versione sovraccarica di get con un timeout di attesa. In questo caso, il thread chiamante attende al massimo 10 secondi affinché il calcolo venga completato. Se il timeout termina, il metodo genera una TimeoutException.

```
// metodo submit
ExecutorService executor = Executors.newSingleThreadExecutor();

Future<Integer> future = executor.submit(() -> {
    TimeUnit.SECONDS.sleep(5);
    return 700000;
});

// metodo isDone
System.out.println(future.isDone()); // most likely it is false

// metodo get
int result = future.get();
int result = future.get(10, TimeUnit.SECONDS); // it blocks the current thread
```

Annullare un'attività

La classe Future fornisce un metodo di istanza denominato cancel che tenta di annullare l'esecuzione di un'attività. Un tentativo fallirà se:

- il compito è già stato completato
- l'attività è già stata annullata
- l'attività non poteva essere annullata per qualche altro motivo

Se il tentativo di annullamento ha successo, l'attività non verrà mai eseguita.

Il metodo accetta un parametro booleano che determina se il thread che esegue questa attività debba essere interrotto nel tentativo di arrestare l'attività (in altre parole, se arrestare o meno un'attività già in esecuzione).

```
future1.cancel(true);  // try to cancel even if the task is executing now
future2.cancel(false); // try to cancel only if the task is not executing
```

Poiché il passaggio di true comporta delle interruzioni, l'arresto effettivo di un'attività in esecuzione è garantito solo se gestisce correttamente InterruptedException e controlla il flag Thread.currentThread().isInterrupted() .

Se qualcuno invoca future.get() su un task annullato con successo, il metodo genera un'eccezione CancellationException non controllata . Se non vuoi occupartene, puoi controllare se un task è stato annullato invocando isCancelled() .

Metodi invokeAll e invokeAny

Oltre a tutte le funzionalità descritte sopra, sono disponibili due metodi utili per inviare batch di Callable a un esecutore.

- **invokeAll** accetta una raccolta preparata di callable e restituisce una raccolta di futures;
- **invokeAny** accetta anche una raccolta di callable e restituisce il risultato (non un future) di un'istruzione che è stata completata correttamente.

Entrambi i metodi dispongono anche di versioni sovraccaricate che accettano un timeout di esecuzione spesso necessario nella vita reale.

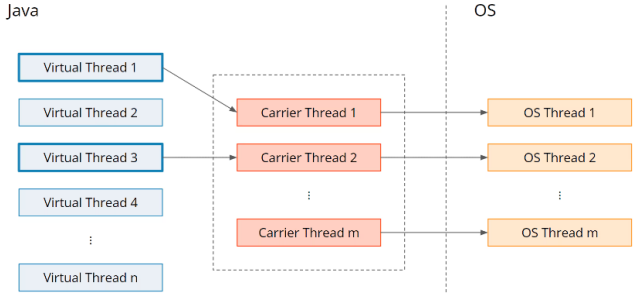
E9.2 JAVA VIRTUAL THREADS

THREAD DELLA PIATTAFORMA (E I LORO PROBLEMI)

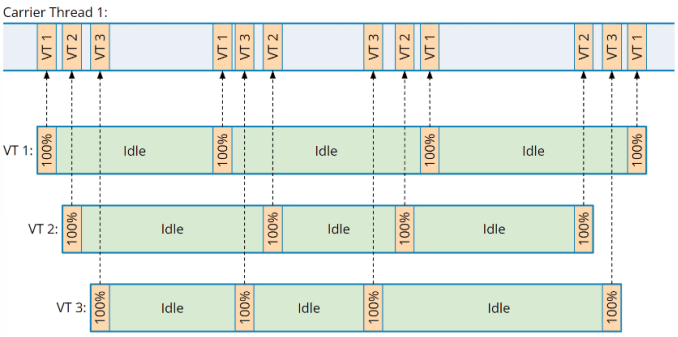
I thread virtuali, gestiti dalla JVM, migliorano la produttività delle applicazioni simultanee. I thread di piattaforma, invece, sono costosi da creare e limitati dalle risorse del sistema operativo. Questi thread, mappati 1:1 sui thread del SO, possono causare **problemi di scalabilità** poiché il numero di richieste simultanee gestibili dipende dall'hardware del server. Durante le operazioni di attesa, i thread di piattaforma restano inattivi, sprecando risorse. La **programmazione reattiva** risolve parzialmente questo problema utilizzando callback e API asincrone, ma mantiene i limiti di scalabilità e rende il debugging e il profiling complessi. Inoltre, richiede un nuovo stile di programmazione, rendendo il codice esistente più difficile da comprendere.

THREAD VIRTUALI

I thread **virtuali migliorano la qualità del codice** mantenendo la sintassi tradizionale e offrendo i vantaggi della programmazione reattiva. Sono istanze di *java.lang.Thread* che eseguono codice su un thread del sistema operativo **senza bloccarlo** completamente, permettendo a molti thread virtuali di condividere lo stesso thread del SO. Si possono creare milioni di thread virtuali senza dipendere dal numero di thread di piattaforma, poiché sono gestiti dalla JVM e non aggiungono overhead di cambio di contesto. I thread virtuali eseguono il codice per tutta la durata di una richiesta, consumando un thread del



SO solo durante i calcoli, senza bloccarlo durante le attese, raggiungendo alta scalabilità e produttività senza complessità di sintassi.



I thread virtuali sono più adatti all'esecuzione di codice che trascorre la maggior parte del tempo bloccato, ad esempio in attesa che i dati arrivino a un socket di rete o in attesa di un elemento in coda.

Differenza tra thread di piattaforma e thread virtuali

Caratteristiche dei thread virtuali:

- I thread virtuali sono sempre thread daemon. Il *Thread.setDaemon(false)* metodo non può modificare un thread virtuale in un thread non daemon. JVM termina quando tutti i thread non daemon avviati sono terminati. Ciò significa che JVM non attenderà il completamento dei thread virtuali prima di uscire.
- I thread virtuali hanno sempre la priorità normale e la priorità non può essere modificata, nemmeno con setPriority(n) il metodo. Chiamare questo metodo su un thread virtuale non ha alcun effetto.
- I thread virtuali non sono membri attivi dei gruppi di thread. Quando viene richiamato su un thread virtuale, *Thread.getThreadGroup()* restituisce un gruppo di thread segnaposto con il nome " VirtualThreads ".
- I thread virtuali non supportano i metodi *stop() , suspend() o resume()*. Questi metodi generano un'eccezione UnsupportedOperationException quando vengono richiamati su un thread virtuale.

```
Thread virtualThread = ...; //Create virtual thread
//virtualThread.setDaemon(true);  //It has no effect
//virtualThread.setPriority(Thread.MAX_PRIORITY);  //It has no effect
```

Confronto delle prestazioni

Per confrontare le prestazioni dei thread virtuali e dei thread di piattaforma, è stata definita un'attività che attende un secondo prima di stampare un messaggio. Sono stati creati 10.000 thread per eseguire questa attività, utilizzando sia thread di piattaforma che thread virtuali. Utilizzando un pool di 100 thread di piattaforma, il tempo totale di esecuzione è stato di circa 101 secondi. Sostituendo il pool con un esecutore di thread virtuali, il tempo di esecuzione è stato ridotto a circa 1,5 secondi, dimostrando la superiorità in termini di velocità dei thread virtuali senza modifiche al codice Runnable.

COME CREARE THREAD VIRTUALI

- ⇨ Utilizzo di **Thread.startVirtualThread():** Questo metodo crea un nuovo thread virtuale per eseguire una determinata attività Runnable e ne pianifica l'esecuzione.
- ⇨ Utilizzo di **Thread.Builder:** Se vogliamo avviare esplicitamente il thread dopo averlo creato, possiamo usare *Thread.ofVirtual()* che restituisce un'istanza di VirtualThreadBuilder. Il suo start() metodo avvia un thread virtuale. (Thread.ofVirtual().start(runnable) è equivalente a Thread.startVirtualThread(runnable)).
  Il metodo *Thread.ofPlatform() è* usato per la creazione di thread di piattaforma.
- ⇨ Utilizzo di **Executors.newVirtualThreadPerTaskExecutor():** Questo metodo crea un nuovo thread virtuale per task. Il numero di thread creati dall'Executor è illimitato. Si noti che la seguente sintassi fa parte della concorrenza strutturata (Project Loom).

```java
// THREAD.STARTVIRTUALTHREAD()
Runnable runnable = () -> System.out.println("Inside Runnable");
Thread.startVirtualThread(runnable);

//or

Thread.startVirtualThread(() -> {
        //Code to execute in virtual thread
        System.out.println("Inside Runnable");
});

// THREAD.BUILDER
Runnable runnable = () -> System.out.println("Inside Runnable");
Thread virtualThread = Thread.ofVirtual().start(runnable);

Thread.Builder builder = Thread.ofPlatform().name("Platform-Thread");
Thread t1 = builder.start(() -> {/*...*/});
Thread t2 = builder.start(() -> {/*...*/});

// EXECUTORS.NEWVIRTUALTHREADPERTASKEXECUTOR
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}
```

MIGLIORI PRATICHE DEI THREAD

NON mettere in pool i thread virtuali: Poiché creare thread virtuali non è costoso, è consigliabile crearne uno nuovo ogni volta che è necessario, senza metterli in pool. Anche se l'applicazione può gestire milioni di thread virtuali, altri sistemi potrebbero gestire solo poche richieste alla volta. In questi casi, è preferibile usare semafori per limitare l'accesso alle risorse piuttosto che i pool di thread.

Evitare di utilizzare variabili locali al thread: Anche se i thread virtuali supportano le variabili thread-local, il loro uso dovrebbe essere limitato per evitare un eccessivo consumo di memoria. Le variabili Extent-Local sono un'alternativa migliore.

Utilizzare ReentrantLock invece di Synchronized Blocks: I thread virtuali possono bloccare i thread del sistema operativo quando eseguono codice all'interno di blocchi o metodi sincronizzati, limitando la scalabilità. È preferibile usare ReentrantLock per metodi utilizzati frequentemente. Tuttavia, i blocchi sincronizzati usati raramente o per operazioni in memoria non necessitano di essere sostituiti.

ALLEGATO: REENTRANTLOCK, COUNTDOWNLATCH, COMPLETABLEFUTURE

Blocco di rientro

ReentrantLock è una classe di sincronizzazione avanzata in Java che offre funzionalità estese rispetto ai blocchi sincronizzati standard, come reentrancy, interrompibilità del blocco, try lock e un parametro di equità. Con i thread virtuali, i vantaggi includono un overhead ridotto nella gestione dei blocchi, migliorata scalabilità, semplificazione del codice simultaneo e integrazione con le utility di concorrenza di Java. Tuttavia, bisogna considerare attentamente l'uso del parametro di equità con un numero elevato di thread virtuali per evitare un impatto negativo sul throughput.

Conto alla rovescia Latch

CountDownLatch consente a uno o più thread di attendere il completamento di un set di operazioni in altri thread. È monouso e versatile. Con i thread virtuali, CountDownLatch diventa più efficiente grazie al meccanismo di attesa leggero, migliorando la scalabilità e semplificando la sincronizzazione delle attività. Questo migliora la reattività e la produttività e facilita il coordinamento di flussi di lavoro complessi.

Nozioni di base di CompletableFuture

CompletableFuture rappresenta il risultato futuro di un calcolo asincrono e consente concatenamento di attività, combinazione e gestione degli errori. Con i thread virtuali, offre un uso efficiente delle risorse, migliorata scalabilità per attività asincrone, gestione semplificata degli errori, uso di API di blocco nelle operazioni asincrone e integrazione con il codice esistente.

DOMANDE TEORIA PROGRAMMAZIONE AD OGGETTI

1. Cos'è un interfaccia? Più o meno astratta rispetto ad una classe astratta?
2. Cosa sono i metodi di default? A cosa servono? (Retrocompatibilità e riuso del codice)
3. Cos'è un'interfaccia funzionale, (legato alla funzione lambda)?
4. Relazione funzionale e lambda
5. Cos'è il metodo di default e perché sono state introdotte in un interfaccia?
6. Cosa sono i generics?
7. Che cos'è un hashtable?
8. Che cos'è un array condition, interferenza tra thread?
9. Cosa che non controlliamo nella programmazione multithread? Lo scheduler?
10. Java supporta il polimorfismo? Dimostralo (cani-gatti-mammiferi)
11. Un metodo che fa riferimento ad int se voglio lo stesso programma ma che utilizzi i float come posso modificarlo?
12. Cosa sono le eccezioni?
13. Quali sono le eccezioni non controllate?
14. Cosa sono i modelli di delega parziali?
15. A cosa serve la delega parziale nelle eccezioni?
16. Qual è la differenza tra throw e throws?
17. Qual è la differenza tra check e uncheck?
18. Come si fa una dichiarazione anonima?
19. Cosa sono le operazioni intermedie? (producono un nuovo stream)
20. Cosa sono le operazioni terminali? (non producono un nuovo stream)
21. Che cos’è un’interfaccia funzionale? (interfaccia con solo un metodo)
22. qual è la differenza tra interfacce e classi astratte?

Esercizi chiesti (max 10 min):

- GENERICS-Maptopair(creando la classe pair)
- THREAD-producer consumer unsafe
- COLLECTIONS
- FUNCTIONS