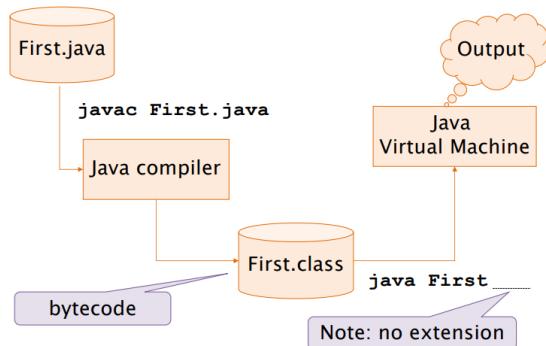


# PROGRAMMAZIONE A OGGETTI

## 0) INTRODUZIONE

Java è un linguaggio che per essere eseguito ha bisogno di una “**VIRTUAL MACHINE**” (qualsiasi sistema avente una virtual machine Java, può eseguire codice Java [Cross-Platform]). La convenzione nella programmazione Java è la “*camelCase*” (e non la “*snake\_case*”), le costanti si scrivono maiuscole (come C) e si indenta sempre. I file java (First.java e il First.class da esso generato [e così tutte le altre classi del mio progetto]) vengono “impacchettati” e “deployati” in file “**JAR**” (progetto.jar). Per quando riguarda la “build&run” di un file Java:

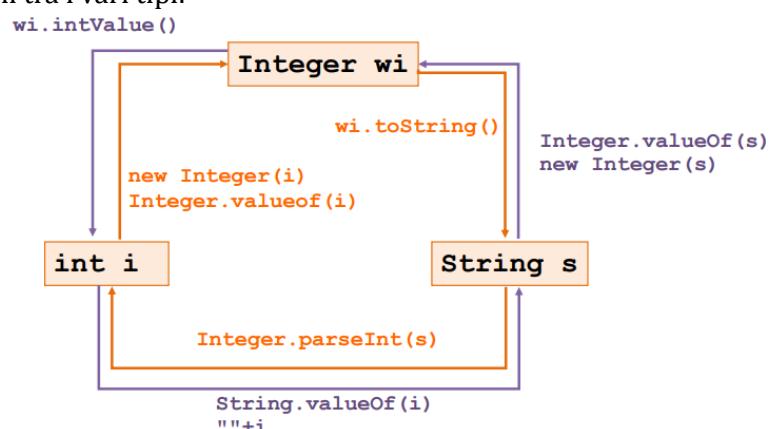


## 1) INTRODUZIONE A JAVA

I **commenti** sono uguali al C (`/**/` e `//`) e si possono dichiarare e inizializzare le variabili in ogni punto del codice (non per forza all'inizio). Sono presenti tutti i costrutti del C (anche il costrutto `switch`, dove troviamo al possibilità di mettere il case anche con le stringhe). In Java c'è il tipo **boolean**, con i valori **true** e **false** (questi sono obbligatori nei costrutti di controllo).

Nella **PROGRAMMAZIONE A OGGETTI**, si definisce la **CLASSE** (ovvero il prototipo dell'elemento che devo descrivere [es. persona]) e le sue singole realizzazioni sono gli **OGGETTI** [es. Mario] (stessa cosa di quando abbiamo il tipo **int** e una variabile di tipo **int x**). I tipi primitivi di Java (qui sotto in tabella, iniziati con la lettera minuscola) sono contenuti in una **classe Wrapper** (che è una sua classe, il cui **nome è uguale al nome del tipo ma inizia con la lettera maiuscola** [eccetto per **char** che ha wrapper **Character** e **int** che ha wrapper **Integer**]), usate per dare dei metodi e garantire conversioni tra i vari tipi:

Type	Size	Encoding
<b>boolean</b>	1 bit	-
<b>char</b>	16 bits	Unicode UTF16
<b>byte</b>	8 bits	Signed integer 2C
<b>short</b>	16 bits	Signed integer 2C
<b>int</b>	32 bits	Signed integer 2C
<b>long</b>	64 bits	Signed integer 2C
<b>float</b>	32 bits	IEEE 754 sp
<b>double</b>	64 bits	IEEE 754 dp
<b>void</b>	-	



Anche qui (come in C) sono presenti gli **operatori aritmetici** e gli **operatori logici** (che ci danno qui un risultato booleano) [anche in Java, seppur siano a 16 bits, i caratteri possono essere trattati come interi nelle operazioni].

I membri di una CLASSE sono gli **ATTRIBUTI** (ovvero le variabili usate localmente alla classe), i **METODI** (ovvero le funzioni scritte all'interno della classe) e i **COSTRUTTORI**.

L'unico modo per accedere ad un OGGETTO è tramite un **RIFERIMENTO** a quell'oggetto (ricorda che oggetto = singola realizzazione della classe) [se ho più riferimenti ad uno stesso oggetto ho aliasing] [riferimento ≠ oggetto ma ci permette di accedere all'oggetto]. **Con i riferimenti posso solo verificare se sono uguali o diversi (ovvero se puntano o no allo stesso oggetto)**; per fare un confronto tra oggetti devo scrivere un metodo che mi compari gli attributi degli oggetti.

```
Car a1, a2; // creo i riferimenti agli oggetti della classe Car  
a1 = new Car(); // a1 è il riferimento al nuovo oggetto di classe Car  
a1.paint("yellow");  
a2 = a1; // riferimento a2 fa anch'esso riferimento all'oggetto puntato da a1 (ALIASING)  
a1 = null; // non elimino l'oggetto, ma tolgo il collegamento tra a1 e l'oggetto (a2 può  
// ancora accedere)  
a2 = null; // anche a2 non punta più all'oggetto, quindi questo diventa "unreachable"
```

Un oggetto “**unreachable**” è inutile in Java, quindi la VM eliminerà automaticamente quell'oggetto (“Garbage collector”). Quando scrivo `a1 = new Car()`, sto chiamando il **COSTRUTTORE** (ovvero quello si occupa di creare il singolo oggetto della classe). **Gli oggetti vengono creati nell'HEAP (ovvero la MEMORIA DINAMICA)**; noi in C usavamo le malloc/free per gestire l'HEAP, mentre qui viene fatto tutto automaticamente dalla VM (non abbiamo bisogno di fare malloc e free [le free vengono fatte dal **GARBAGE COLLECTOR**]). Quando creo l'oggetto con il costruttore base “new”, gli attributi:

- numerici vengono messi a 0;
- booleani vengono messi a false;
- riferimenti vengono messi a null.

⚠ Quando scrivo il **COSTRUTTORE**, questo non avrà un “return type”, ma avrà prototipo `public nome()` [con `nome = nome della classe`]!

Posso usare il termine **this** per far riferimento all'attributo specifico dell'oggetto in questione:

```
public class Car {  
    String color;  
    void paint(String color) { // color lo passo al metodo paint da mettere in this.color  
        this.color = color; // this.color è il colore dell'oggetto che devo modificare  
    }  
}
```

Per accedere ai singoli attributi di un oggetto, usiamo la scrittura con il puntino (es. `a1.color = green`). La scrittura con il puntino può essere usata per concatenare dei metodi che hanno un **return this** come valore di ritorno (creando un nuovo idioma).

In una stessa classe **posso avere più metodi con lo stesso nome (OVERLOAD)**, ma devono ricevere parametri diversi: la regola è che i metodi di una stessa classe debbano avere signature diverse (con signature = nome + parametri). **Questo principio si usa per definire più costruttori all'interno della stessa classe**, che vengono quindi chiamati a seconda dei parametri passati quando uso `new`.

⚠ Se dichiaro delle variabili in un metodo, queste avranno “scope” locale al metodo, quindi una volta usciti dal metodo, queste moriranno lì!

Nella PROGRAMMAZIONE A OGGETTI per accedere ai campi di un oggetto, si usano dei metodi (e non la scrittura con il puntino); infatti gli **ATTRIBUTI** di una classe vengono dichiarati con davanti **private**: in questo modo garantisco che gli attributi possano essere toccati solo dai metodi della classe stessa. I **METODI** invece vengono messi **public** apposta per garantire che ad essi si possa accedere e quindi manipolare la classe dall'esterno. Questo si dice **INCAPSULAMENTO**, ovvero cercare di ridurre al minimo la dipendenza di pezzi di programma da altri pezzi e inoltre fare in modo che l'utente debba sapere il meno possibile della struttura dei dati; questo favorisce le **MODIFICHE** di un software senza doverlo modificare del tutto (**un software per essere utile deve adattarsi/cambiare** e per farlo non va riscritto ogni volta) [stesso motivo per cui si preferisce usare dei metodi tipo “descriviti” che ci ridanno una descrizione letterale degli attributi richiesti al posto di tanti metodi “**GETTER**”].

⚠ In Java come in C possiamo creare dei tipi enumerativi con `enum Lato {SX, CENTRO, DX}`.

La dichiarazione **static** in Java (a differenza di C) ci dice che l'attributo è UNICO, ovvero se dichiaro un ATTRIBUTO come **static** avrò che quell'attributo è condiviso da tutti gli oggetti (es. se dichiaro un attributo come **nomePosti** posso dichiararlo come static perché mi interessa il numero complessivo di posti [ovvero dei singoli oggetti] che vengono creati e non mi riferisco al singolo oggetto); se invece dichiaro un METODO come **static** non dovrò usare il **this** al suo interno perché al suo interno potrò accedere solo ad attributi statici e non agli attributi senza static. Questi metodi essendo che non si riferiscono ad un singolo oggetto, ma all'intera classe, allora potrò accedere a questi metodi usando semplicemente il nome della classe e il puntino (es. **Posto.quantiposti()**). Proprio per questo motivo, si usa **static** davanti al **main**, in modo che la Virtual Machine possa chiamare il main senza che io debba prima definire un metodo per quella classe di nome main!

⚠ Quando dobbiamo cambiare il nome di una variabile/modulo in un file, non si fa manualmente il cambiare tutte le occorrenze, ma si seleziona e si fa **Fn+F2!**

→ **CONFIGURATION MANAGEMENT CON GIT:** si occupa dell'amministrazione (identificazione e documentazione) di un “**configuration item**” (**CI**, ovvero un insieme di prodotti trattati come una singola entità nel configuration management [solitamente un file]), ne controlla le modifiche e la conformità a certi requisiti. Qui parliamo di:

- **VERSIONING** → si occupa della storia del CI, ovvero tutte le sue precedenti versioni vengono mantenute. Le versioni vengono cambiate quando l'utente fa il commit (è possibile comunque tornare indietro) [“version numbering” = semplice schema di “naming” del tipo V1, V1.1, dove la struttura è quindi un albero/rete non una sequenza, ma i nomi in sé delle versioni non hanno significato]. Definiamo “**CONFIGURATION**” l'insieme dei CI in una specifica versione (es. un prodotto composto da 2 file dove fileA è alla versione 1.1 e fileB è alla versione 1.2); alcuni CI possono apparire in più configurazioni e la stessa configurazione ha una sua specifica versione. Definiamo “**BASELINE**” una configurazione in forma stabile/fissa (se ci sono delle modifiche, queste produrranno nuove versioni dei CI ma non modificano la baseline) [ci sono baseline per uso interno (sviluppo) e vendita (prodotto)]. Un altro tipo di versioning è il “semantic versioning”, basato su “**MAJOR.MINOR.PATCH**” (major [o **breaking change**] = modifiche grosse all'API, non compatibili; minor [o **feat**] = funzionalità aggiunte, ma retrocompatibili; patch [o **fix**] = bug-fix compatibili). Quando faccio il **COMMIT** della mia nuova versione, il commit avrà struttura:

```
<type>(<scope>) : <subject>
<body>
<footer>
```

```
fix(middleware): ensure Range headers
adhere more closely to RFC 2616
Added one new dependency, use `range-
parser` (Express dependency) to compute
range. It is more well-tested in the
wild.
Fixes #2310
```

- **CHANGE CONTROL** → indichiamo con “**REPOSITORY**” il luogo dov'è conservata la collezione di tutto quello inerente ad un certo sistema. In un team ci sono “concurrent changes” (modifiche simultanee da parte di membri diversi), per questo ci sono diverse modalità di interazione per evitare di modificare il lavoro che un altro sta facendo (definiamo **CHECK-IN/COMMIT** = inserimento di un CI sottocontrollo; **CHECK-OUT** = estrazione di un CI dalla repository con l'obiettivo di modificarlo):
  - **LOCK-MODIFY-UNLOCK** (o serializzazione) = solo un membro del team può modificare per volta (quindi non si possono verificare conflitti, ma non c'è lavoro parallelo);
  - **COPY-MODIFY-MERGE** = più membri in parallelo possono fare modifiche (poi fuse insieme) (quindi lavoro parallelo e flessibile, ma richiede cura per risolvere conflitti) [oggi si usa questo].

- **CONFIGURATION** → definiamo “**branches**” le ramificazioni che si sviluppano sul CI se ci sono dei lavori contemporanei diversi su di essi (es. un membro del team sta lavorando su quel CI, ma anche un altro ci sta lavorando, quindi ne esistono 2 repliche con una storia diversa, ma se vado abbastanza indietro nel tempo, ci sarà un punto di partenza comune delle 2 ramificazioni) [le branches possono rappresentare diverse configurazioni eventualmente per sistemi/piattaforme diversi; sono utili per lavorare sul CI senza modificare il main (una volta finita la modifica si può fondere con il main)]. Definiamo “change control functions” le funzioni che memorizzano nuove versioni del CI (possono fare una revert sul CI sulle modifiche).

Vediamo quindi ora come sono fatte alcune “**REPOSITORY ARCHITECTURE**”; parlando di “**DISTRIBUTION TAXONOMY**” abbiamo distribuzione:

- LOCALE = un database che controlla tutte le modifiche sotto revisione;
- CENTRALIZZATO = un server che contiene tutte le versioni di un file;
- DISTRIBUITO = clienti possono clonare direttamente la repository.

Definiamo “**SNAPSHOT**” un’immagine di come sono tutti i nostri file del sistema di lavoro in un dato momento; viene fatto uno snapshot ad ogni commit.

Ora parliamo di **GIT**: è un **CMS** (Control Management System) distribuito, usa gli snapshots, usa operazioni locali e ha integrità (tutto viene verificato prima di essere memorizzato). Si possono usare per usare GitHub (online code hosting service con uso commerciale) e GitLab (che useremo in questo corso per sfruttare Git). In GitLab chiamiamo **REPOSITORY** il posto dove si trova tutto il nostro lavoro (contiene tutte le versioni che sono mai esistite del nostro lavoro). Uno snapshot di questa repository è la **WORKING COPY**, dove avvengono le modifiche; questa è privata, non viene condivisa con il team, e i file vengono categorizzati come “**tracked**” (file nella nostra working copy, le cui modifiche vengono monitorate), “**untracked**” (file nella nostra working copy, le cui modifiche però non vengono monitorate) e “**ignored**” (file nella nostra working copy completamente ignorati da Git; contenuti nel “.gitignore”).

Il **COMMIT** salva le modifiche a questi file, mettendoli nella repository. I file “**tracked**” nella mia working copy possono essere “**not changed**”, “**added**” (aggiunti alla lista di file con modifiche da salvare), “**modified**” (cambiati solo nella local working copy”), “**staged**” (modificati e segnati come da aggiungere al prossimo commit) e “**committed**” (correttamente salvati nella local repository).

Il **PUSH** invece è quando, dopo il commit (ovvero il save delle modifiche ai file nella nostra local repository), queste modifiche vengono mandate alla repository centrale (quando faccio commit salvo le modifiche sul locale, quando faccio push mando le modifiche alla cartella condivisa del progetto [il main]). Il **PULL** (fetch [prendo file da repository centrale e lo metto nella mia local repository] + checkout [prendo da local repository e metto nella mia working copy]).

⚠ Quando facciamo il “**clone**” da github/gitlab (copiano l’URL tramite HTTPS), se sto usando **Crownlabs**, fare il clone nella cartella “workspace” (così da evitare problemi in caso di crash). Quando faccio delle modifiche al mio progetto (soprattutto **all’esame**), dobbiamo selezionare dal nostro menu a barra di sinistra l’icona di git e fare direttamente il “**commit and push**” (e **non solo il commit**) [**altrimenti non consegneremo nulla!!!**]. Se quando proviamo a fare “commit and push” non riusciamo a farlo, ci comparirà un avviso a schermo da vscode: li per capire la motivazione non schiacciamo su GitLog ma schiacciamo “mostra comando output”.

**Quando abbiamo 2 branches, ma vogliamo andare avanti, da quale partiamo?\*\*** Facciamo un “**three-way merge**” ovvero si fa un merge tra i 2 branch con un antenato comune e si va avanti con il main [il default branch] (HEAD). Oppure, se i 2 branches sono un errore (quindi non ci interessa la storia passata del branch), possiamo fare “**branch&rebase**” che mi riporta al branch senza fare il merge (così che posso decidere io cosa fare delle modifiche) [in questo caso git aggiunge dei caratteri nel mio “.md” per distinguere nel codice le parti dei 2 branches e mi dice di decidere cosa fare del codice].

In locale ho il puntatore **HEAD** che punta al **commit che si trova nella working copy** (che sarà uno dei branch attivi, altrimenti male); nella repository, abbiamo il **default branch** che è invece il **MAIN** (prima chiamato master). Quando faccio il branch, si crea un nuovo puntatore aggiuntivo ad un commit che già esiste (una ramificazione per uno sviluppo concorrente); se voglio **creare un branch** chiamato “testing”, dovrò fare **git branch testing**. Se voglio far puntare HEAD al testing branch (ovvero voglio che il mio commit sia su testing), dovrò fare **git checkout testing**. Andando avanti con lo sviluppo posso trovarmi in 2 situazioni:

- Posso fare il **merging di testing dentro main** (ovvero nell’HEAD) e quindi fare un “**fast-forward merge**” [comando **git merge testing**];
- Lo **sviluppo su main è andato avanti\*\***: cosa faccio con il mio testing? Se voglio **unirlo al main** faccio il three-way merge sopra citato [**git merge testing**], se **non voglio unirlo** faccio il rebase sopra citato (fare un commit che include tutte le modifiche fatte nel branch testing oltre alle modifiche fatte nel main) [**git rebase main**].

⚠ **Rebase** = crea un commit history lineare; **Merge** = combina le commit histories di 2 branches (creando un nuovo commit merged).

Ritorniamo a **JAVA** ora, vedendo l’esempio fatto durante le prime lezioni:

```
package carsharing; // i package sono cartelle che contengono le classi correlate tra loro  
(e eventualmente altri package), permettendo una struttura gerarchica delle classi  
  
public class Posto {  
  
    // costanti (definite con final perchè final significa non più modificabile), che  
in Java a differenza di C hanno un tipo  
    static final char ANTERIORE = 'A';  
    static final char POSTERIORE = 'B';  
  
    enum Lato {  
        SX, CENTRO, DX  
    }  
  
    //static final int SX = -1;  
    //static final int CENTRO = 0;  
    //static final int DX = 1;  
  
    // attributi (campi)  
    private char posizione; // 'A' | 'P'  
    private Lato lato; // SX | CENTRO | DX  
    private boolean conducente = false;  
  
    private String nomePasseggero;  
    private boolean occupato;  
  
    // metodi (operazioni)  
  
    /** !!! Questo è un JavaDOC per documentare i metodi creati  
     * Costruttore di posto  
     *  
     * @param posizione = posizione ANTERIORE o POSTERIORE  
     * @param lato = lato SX, DX o CENTRO  
     * @param conducente = se è il posto del conducente allora true  
     */  
    public Posto(char posizione, Lato lato, boolean conducente) {  
        // this.conducente (SX) = mi riferisco all'attributo dell'oggetto in questione  
        // conducente (DX) = mi riferisco al parametro passato alla funzione  
        this.conducente = conducente;  
        this.lato = lato;  
        this.posizione = posizione;  
    }  
  
    // GETTERS -> metodi che ci ridanno un attributo dell'oggetto  
  
    public Lato getLato() {  
        return lato;  
    }  
  
    public char getPosizione() {  
        return posizione;  
    }  
  
    public boolean getConducente() {  
        return conducente;  
    }  
  
    /**  
     * Ci ritorna la descrizione del posto  
     * @return descrizione  
     */
```

```

public String descrivi() {
    String descrizione = new String(); // equivale a -> String descrizione = "";

    // espressione condizionale (if pos==ANT)desc = Anteriore; else Posteriore
    descrizione = (posizione==ANTERIORE?"Anteriore":"Posteriore");

    descrizione += " ";
    switch(lato) {
        case SX: descrizione += "sinistro";
                    break;
        case DX: descrizione += "destro";
                    break;
        case CENTRO: descrizione += "centro";
                    break;
    }
    if (conducente) {
        descrizione += " (conducente)";
    }

    return descrizione;
}

/**
 * Serve per prenotare il posto an nome di un passeggero
 *
 * @param passeggero il nome del passeggero
 */
public void prenota(String passeggero){
    this.nomePasseggero = passeggero;
    this.occupato = true;
}

/**
 * Ci ritorna se il posto è occupato
 * @return occupato
 */
public boolean eOccupato(){
    return occupato;
}
}

```

Abbiamo parlato di metodi definiti con static; questi vengono usati per creare le “**FUNZIONI**” (metodi il cui return dipende solo dagli argomenti), solitamente messe in una “**utility class**” (classe con tutte le funzioni statiche). Alcune utility classes sono:

- **System** → interagisce con l’OS (currentTimeMillis(), exit(), PrintStream out);
- **Math** → funzioni matematiche;
- **Arrays** → funzioni utili per arrays (binary search, copy, equals, fill, sort, toString).

Altro uso dei metodi statici sono i **FACTORY METHODS**, metodi per creare oggetti senza specificare il tipo esatto dell’oggetto (specificato durante l’esecuzione del programma, specificando il tipo richiesto durante l’esecuzione del programma stesso).

**⚠ Non usare attributi static nelle nostre classi e non usare la funzione exit!**

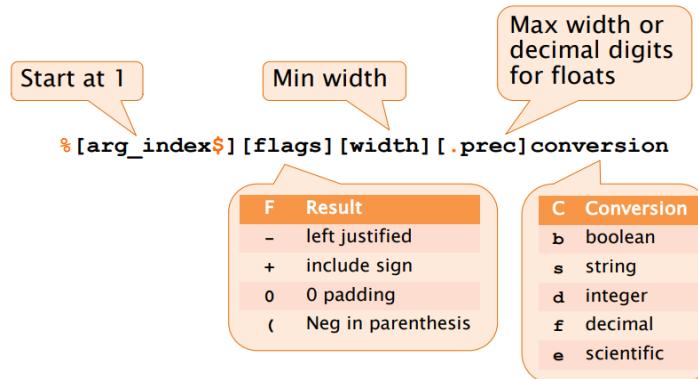
Se voglio **deprecate un metodo** (dire che non va più usato), scrivo nel suo JavaDoc **@deprecated**. Inoltre, come abbiamo già detto sopra, abbiamo la **MEMORIA STATICÀ** (dove troviamo le cose dichiarate con static [variabili statiche]), **DINAMICA** (o **HEAP**, dove troviamo gli oggetti [variabili d’istanza]) e **LOCALE** (o **STACK**, dove troviamo le variabili locali [es. i riferimenti agli oggetti dichiarati come variabili locali]).

## 2) CHARS, STRINGS e ARRAYS

Come abbiamo già detto i **CHAR** in Java sono su **16 bit** e hanno codifica **Unicode UTF16** (come in C possiamo usare i caratteri escape con il backslash [es. \n]). Hanno Wrapper class **“Character”** (mentre il loro tipo è **“char”**). Il wrapper **Character** **incapsula 1 carettere immutabile** (come accade in tutti i wrapper)[metodi utili sono **isLetter()**, **isDigit()**, **isSpaceChar()**, **toUpperCase()**, **toLowerCase()**].

⚠ **Unicode** è uno standard che assegna un codice univoco numerico (**CODEPOINT**) per ogni carattere di ogni linguaggio (character = il carattere; gliph = la sua rappresentazione grafica [font = collezione di glyphs]). Nel nostro caso UTF-16 si riferisce al mapping su 16 bit (2 bytes). Legato a ciò esistono i **CHARSET** (set di caratteri), es. UTF-8.

Abbiamo quindi le **STRINGHE**, dove con la classe **String** diciamo che sono **immutabili**, mentre con **StringBuffer** le rendiamo **modificabili**. Possiamo **concatenare** 2 stringhe con “+”, mentre posso prendere la lunghezza di una stringa con **int length()**. Abbiamo molti **METODI** per le stringhe: str1.equals(str2) per comparare 2 stringhe (non si usa == tra stringhe). Altri metodi sono String **toUpperCase()**, String **toLowerCase()**, String **concat(String s)**, int **compareTo(String s)**. C’è anche il metodo “Ciao”.**subString(2) → “ao”** anche in forma “Perfetto”.**subString(0,5) → “Perfe”**; altri metodi sono int **indexOf(String str)** [torna l’indice della prima occorrenza di str] e **lastIndexOf(String str)** [che fa la stessa cosa di indexOf, ma partendo dal fondo]. Poi c’è **String valueOf(...)** che converte qualsiasi cosa in una String, e **String format(String fmt, ...)** che costruisce una stringa usando la stringa formato (formato simile alla printf()), usando le seguenti regole:



Come abbiamo detto prima c’è la possibilità di usare **StringBuffer** per riferirmi a stringhe modificabili (posso comunque convertirle a String con **toString**), con i metodi **append(String str)**, **insert(int offset, String str)**, **delete(int start, int end)**, **reverse()**.

**StringBuilder** è come **StringBuffer** (**stessi metodi di StringBuffer**), ma non è **“thread-safe”** (se modifichiamo uno **StringBuilder** con 2 thread, questo non garantirà mutua esclusione con un semaforo, ma verrà fuori un pasticcio perché verrà modificato insieme); nonostante questo però è molto più efficiente se sto lavorando sulle stringhe (es. una concatenazione di stringhe con **StringBuilder** [usando un for dove uso il metodo **append**] rispetto a **String** [usando il +] è 1000 volte più veloce).

⚠ Esiste il metodo **assert(condizione)** che ci verifica se la condizione (es. un’uguaglianza **c == 2**) è vera, altrimenti mi stoppa il programma (altrimenti posso realizzare un **gestore di errori** analoghi mediante degli if-else). Definiamo **TDD** (**Test-Driven Development**) la pratica di sviluppo del software in cui la scrittura dei test automatizzati viene eseguita prima dell’implementazione del codice effettivo; inoltre possiamo commentare le cose ancora da fare con **//TODO** per ricordarci di implementare quelle cose che non ho ancora fatto.

⚠ Come in C, esiste il puntatore al nulla, ovvero **null** (e non **NULL** come in C); in Java però questo null viene gestito meglio (es. **return null**), perché la **VM genera un messaggio di errore e gestisce il null** (mentre in C solitamente è un “crash”). Quando ho dei metodi che ritornano **null**, e li concateno uno all’altro, mi si genera una **NullException** perché applico un metodo ad un **null** (è meglio un **return this** se devo concatenare dei metodi che uso per modificare lo stesso oggetto).

⚠ All’esame fare spesso **“Commit&Push”** (soprattutto quando finiamo un punto richiesto), così da evitare problemi in casi di crash e in caso in cui siamo in ritardo con i tempi.

Nella progettazione grafica delle classi usiamo i **DIAGRAMMI UML**: ogni **tabellina** è una **classe** che ha **titolo** = nome della classe, **2 righe** dove la **1^** sono gli attributi “**- posizione: char**” (dove - = privato, posizione = nome attributo, char = tipo attributo) e la **2^** sono i metodi “**+ eOccupato(): boolean**” (dove + = pubblico, eOccupato = nome metodo, boolean = return type); i **collegamenti tra classi** indicano la relazione tra 2 classi (es. macchina e posto; **1 auto ha 5 posti**, quindi farò una freccia che va macchina a posto con il numerino 1 vicino macchina e 5 vicino a posto). Queste relazioni possiamo rappresentarle per esempio con degli **array** (es. macchina = array di 5 posti).

Parlando invece degli **ARRAYS** (**vettori**), vediamoli con del codice:

```
package primo;

import java.util.Arrays;

public class EsArray {
    public static void main(String[] args){
        int v[] = new int[10]; // DICHIAZIONE
        int i = v[0];

        for(int j=0; j<v.length; ++j) { // metodo length
            v[j] = j;
        }

        System.out.println(v); // sbagliato per stampare gli array
        System.out.println(Arrays.toString(v)); // giusto per stampare gli array
        Arrays.sort(v); // ordinamento degli elementi degli array

        i = v[-1];
        double[][] matrice; // array di array di double
        matrice = new double[2][2]; // matrice quadrata
        matrice[0][0] = 3.1415;
        double[] riga = matrice[1];

        matrice = new double[2][];
        matrice[0] = new double[10];
        matrice[1] = new double[2];
    }
}
```

Le **dichiarazioni** di array inizializzano gli elementi a 0 se ho int e double, a false se ho boolean; altrimenti per gli array di classi o stringhe, qui gli elementi vengono inizializzati a null. Vediamo l'uso di **array nella classe** tragitto:

```
package carsharing;

import carsharing.Posto.Lato;

public class Tragitto {
    private String da;
    private String a;
    private Ora oraPartenza;
    private Ora oraArrivo;

    //Posto posti[]; // ARRAY C-LIKE -> non così in Java
    Posto[] posti; // ARRAY JAVA -> in java un array è un oggetto di una nuova classe,
    ovvero la classe che sto scegliendo, ma "vettorizzata"

    public Tragitto(String da, String a, int h0, int m0, int h1, int m1){
        this.da = da;
        this.a = a;
        oraPartenza = new Ora(h0,m0);
        oraArrivo = new Ora(h1,m1);
```

```

posti = new Posto[5]; // Array di 5 references -> iniziati a null per le classi e
le stringhe (non a 0 come gli int o a false come i boolean)

// uso STATIC INITIALIZER per riempire la "macchina" standard
char[] posizioni = {'A', 'A', 'P', 'P', 'P'};
Lato[] lati = {Lato.SX, Lato.DX, Lato.SX, Lato.CENTRO, Lato.DX};
boolean[] conducente = {true, false, false, false, false};
for(int i = 0; i < posti.length; ++i){
    posti[i] = new Posto(posizioni[i], lati[i], conducente[i]);
}

// in alternativa si può usare lo static initializer (senza dichiarazione [not
with declaration])
posti = new Posto[]{
    new Posto('A', Lato.SX, true),
    new Posto('A', Lato.DX, false),
    new Posto('P', Lato.SX, false),
    new Posto('P', Lato.CENTRO, false),
    new Posto('P', Lato.DX, false)
};

public int durata(){
    return oraArrivo.differenzaMinuti(oraPartenza);
}

/**
 * Il passeggero viene prenotato sul 1° posto disponibile per il tragitto (ammesso che
ci sia un posto libero)
 * @param nomePasseggero
 * @return se è riuscito a prenotare un posto oppure no
 */
public boolean prenota(String nomePasseggero){
    // classico for
    for(int i = 0; i < posti.length; ++i){
        Posto p = posti[i];
        if(! p.eOccupato()) {
            p.prenota(nomePasseggero);
            return true;
        }
    }

    // IN ALTERNATIVA uso la sintassi for-each
    for(Posto p : posti){
        if(! p.eOccupato()) {
            p.prenota(nomePasseggero);
            return true;
        }
    }

    return false;
}

//-----
// Definisco la classe interna Ora per l'ora di partenza e arrivo
class Ora{ // Inner Class (ha il "riferimento" all'oggetto contenitore); se l'avessi
dichiarata con static class Ora avrei una semplice Nested Class (classe annidata), che non
ha legami con la classe contenitore
    int ore;
    int minuti;

    public Ora(int h, int m){

```

```

        ore = h;
        minuti = m;
    }

    public String toString(){
        return ore+":"+minuti;
    }

    /**
     * Esempio:
     * Ora p = new Ora(10:20);
     * Ora a = new Ora(11:30);
     * int durata = a.differenzaMinuti(p);
     * @param inizio = ora inizio/partenza
     * @return differenza tra inizio e fine in minuti
     */
    public int differenzaMinuti(Ora inizio){
        return (this.ore*60+this.minuti)-(inizio.ore*60+inizio.minuti);
    }

    // posso direttamente lavorare nella classe Ora (interna alla classe Tragitto)
sugli attributi di Tragitto, in quanto l'oggetto interno vede gli attributi dell'oggetto
esterno che lo contiene
    public int durata(){
        return (this.ore*60+this.minuti)-(oraPartenza.ore*60+oraPartenza.minuti);
    }
}
}

```

## → JUnit

L'obiettivo di **JUnit** è costruire una classe i cui metodi li uso per il **TESTING del mio programma**. Per ogni metodo di test, Junit chiama un "pre-test fixture", il "test method" e le "post-test fixtures". Un **metodo di test** non torna nulla; i check fatti vengono fatti con dei metodi del tipo **assert\***().

```

assertTrue(boolean test)
assertFalse(boolean test)
assertEquals(expected, actual)
assertSame(Object exp, Object actual)
assertNotSame(Object exp, Object act)
assertNull(Object object)
assertNotNull(Object object)
fail()

```

⚠ Distinguiamo **FAILURE** = un metodo assert trova che la condizione da verificare sia sbagliata, mentre **ERROR** = durante l'esecuzione del programma è stato trovato un errore.

Quando faccio run con il runner di Junit, questo esegue tutti i metodi annotati con **@Test**, di tipo pubblico, che non tornano nulla (**void**) e senza argomenti, ignorando tutti gli altri metodi della classe. Possiamo:

- Usare l'annotazione **@Before** per usare un metodo di test e fare un **pre-test fixture** (eseguito prima di ogni metodo di test; si occupa di inizializzare gli oggetti usati nei metodi di test);
- Usare l'annotazione **@After** per usare un metodo di test e fare un **post-test fixture** (eseguito dopo ogni metodo di test; si occupa di liberare tutte le risorse di sistema).

Quando ci si aspetta che accada nel nostro test un'**eccezione** specifica, per fare in modo di trattarla come un'eccezione e non come un errore (ovvero non gestirla), bisogna mettere usare prima del metodo **@Test(expected=PossibleException.class)**. Con **@Ignore** dico che quel metodo di test va skippato (es. se è ancora incompleto)[posso usare anche **@Disabled**]. Se voglio invece runnare più test insieme (come se fosse una **Suite di test**), posso usare:

```

@RunWith(Suite.class)
@SuiteClasses({
    TestStack.class, AnotherTest.class
})
public class AllTests { }

```

## 4) EREDITARIETÀ

Una classe può essere **DERIVATA** da un'altra [classe derivata = **SOTTOCLASSE**; classe base = **SUPERCLASSE**]; una classe derivata rispetto ad un'altra eredita tutti i membri della classe e può aggiungere nuove caratteristiche (può solo aggiungere e non togliere). Se modifichiamo la classe di base, si modifica anche la classe derivata (diciamo che si comporta come una reference alla classe di base e fa un extend su di essa). Le classi derivate hanno maggiore flessibilità grazie a:

- **POLIMORFISMO** → quando una reference di tipo T può puntare ad un oggetto di tipo S, se S = T o S sottoclasse di T (es. il manager è anche un impiegato, quindi posso usare una reference di tipo impiegato per puntare ad un oggetto di tipo manager);
- **DYNAMIC BINDING** → sceglie in maniera dinamica che implementazione del metodo associare alla chiamata, in base al tipo reale dell'oggetto; in quanto **ho ridefinito il metodo di base per la mia nuova classe derivata** (questo meccanismo c'è in ogni linguaggio di programmazione ad oggetti).

⚠ **@Override** è un suggerimento al compilatore che indica che il metodo corrente deve sovrascrivere un metodo della classe genitore, e se non lo fa, il compilatore genererà un errore [aiuta a garantire la corretta sostituzione del metodo vecchio con il nuovo della classe derivata].

Il cast da Integer a Number è un **up-cast**, mentre da Number a Integer è un **down-cast** (**up-cast è sempre corretto perché Number include Integer, ma down-cast potrebbe portare ad errori perché Integer è incluso da Number, ma non include Number**).

⚠ Quindi grazie all'ereditarietà, che definisce una sorta di **SOTTOTIPO**, gli oggetti della classe base possono essere sostituiti da oggetti della classe derivata (**PRINCIPIO DI SOSTITUZIONE DI LISKOV [LSP]**).

⚠ Ricapitolando la **VISIBILITÀ** (scope), in Java abbiamo **private** (visibile solo a quella classe), **package** (visibile a tutte le classi dello stesso package), **protected** (visibile alle classi che derivano dalla classe [ereditarietà], oltre alle classi dello stesso package) e **public** (a tutti).

Per **CREARE DELLE CLASSI DERIVATE DA UNA CLASSE BASE** si crea il file.java (es. Sottoclasse.java) e si dichiara la sottoclasse con **public class SottoClasse extends MiaClasse** (con classe base = superclasse = MiaClasse, e classe derivata = sottoclasse = SottoClasse). Quando creo una nuova classe, questa estende di default (in maniera sottointesa, non ho scritto un extends esplicito) la **CLASSE Object**; l'utilità di questa classe è che posso puntare a tutto con questo tipo (una sorta di puntatore a void del C) e che posso usare dei "metodi base" che posso usare per ogni oggetto (es. **toString** e **equals**, che posso modificare esplicitamente per le singole classi specificandone le caratteristiche antemponendo loro **@Override**). La classe Object è anche utile se devo passare ad una funzione dei parametri di tipo diverso e di numero impreciso, mettendo come parametro del metodo **Object...args** (che è come se definisse un array di Object su cui poi ciclo con **for(Object o : args)**); questo ovviamente garantisce flessibilità, ma al tempo stesso è molto pericoloso proprio per la sua eccessiva flessibilità alla tipizzazione.

L'**equals** che sovrascrivo deve essere **riflessivo** ( $x = x$ ), **simmetrico** (se  $x = y$ ,  $y = x$ ), **transitivo**, **consistente** (ha senso definirlo su oggetti immutabili, non conviene definirlo su oggetti che possono cambiare) e **robusto** ( $x$  e null obbligatoriamente diversi [garantito dal costrutto **instanceof**, che mi dice se un oggetto è di una certa classe specificata]).

⚠ In Java ogni oggetto ha un "**hash code**" (codice univoco che lo rappresenta); posso usare il metodo **hashCode()** della classe di default Object per ritornare questo valore intero (se non faccio l'override del metodo, quello di default prende l'indirizzo di memoria dell'oggetto e ne fa l'hash convertendolo in un intero). Se faccio l'override del metodo, il mio hashCode deve essere **consistente** e "**equal compliant**" (se 2 oggetti sono uguali per il metodo **equal**, devono avere necessariamente hashcode uguale).

⚠ In VSCode abbiamo **MAVEN** che si occupa di fare **compilazione e testare il nostro programma con dettagli sulla build e sugli errori!** (questo si può vedere anche con le **PIPELINES** su GitLab). Quindi all'esame prima di fare commit and push, andare su Maven e fare **TEST**. Parlando invece del file **pom.xml**, questo è un file di **configurazione** che ci dà informazioni sul nostro progetto (dependencies, properties, etc... tutto in formato HTML).

Quello che ci piacerebbe è creare una “**CLASSE SCHELETRO**” che contiene dei metodi “placeholder” (segnaposto, in modo che io possa successivamente riempirli con le informazioni specifiche della mia classe), ovvero parliamo delle **CLASSI ASTRATTE**. Quindi se mi serve creare una classe scheletro che contiene i metodi come placeholder per poi, tramite ereditarietà (`extends`), posso riscrivere per il caso specifico, devo dichiarare la classe con **public abstract class** (e davanti ai metodi placeholder metto `abstract`); il termine **abstract** rende impossibile la creazione di un oggetto di quella classe (ci sarebbe un errore).

Se ho definito una classe da un'altra usando **extends** (ereditarietà), posso richiamare un metodo della classe di **base** (**superclasse**) usando il nome del metodo preceduto da `super` (es. `super.aggiungi()` [funziona anche per gli attributi, es. `super.attr`, che si riferisce ad un attributo definito nella classe di base]). Se voglio chiamare il **costruttore della classe di base** per usarlo su questa classe, posso usare direttamente solo la parola `super` con i parametri al suo interno (es. `super(100, "Ciao");`).

Una classe che contiene al suo interno solo metodi e attributi che dovrebbero essere `abstract` è detta “**INTERFACCIA**” (posso quindi sostituire `public abstract class` con `public interface`); le interfacce a differenza delle classi permettono l'**ereditarietà multipla**, ovvero **può estendere** non solo 1, ma **2 o più classi**. Per usare l'**ereditarietà su un'interfaccia** (classe astratta per eccellenza) e farla ereditare da una vera classe, non uso `extends` (che si usa per estendere le classi) ma uso `class implements interface` (inoltre si ha che un'interfaccia non può estendere una classe, ma può estendere solo 1 o più interfacce [es. `interface A extends B, C`]). Quindi è corretto scrivere `class ClassA extends ClassB implements InterfaceA, InterfaceB{...}`, mentre tra interfacce si ha `interface InterfaceA extends InterfaceB, InterfaceC {...}`.

→ Si definisce “**FUNCTIONAL INTERFACE**” (interfaccia funzionale) un'interfaccia che non ha attributi e ha solo 1 metodo (1 solo comportamento); ci sono delle interfacce funzionali già definite in `java.util.nome_funzione`:

<pre>public interface Comparable{     int compareTo(Object obj); }</pre> <ul style="list-style-type: none"> <li>▪ Returns           <ul style="list-style-type: none"> <li>• &lt;0 if <code>this</code> precedes <code>obj</code></li> <li>• =0 if <code>this</code> equals <code>obj</code></li> <li>• &gt;0 if <code>this</code> follows <code>obj</code></li> </ul> </li> </ul>	<pre>public interface Comparator{     int compare(Object a, Object b); }</pre> <ul style="list-style-type: none"> <li>▪ Semantics (as comparable): returns           <ul style="list-style-type: none"> <li>• a negative integer if <code>a</code> precedes <code>b</code></li> <li>• 0, if <code>a</code> equals <code>b</code></li> <li>• a positive integer if <code>a</code> succeeds <code>b</code></li> </ul> </li> </ul>
--	---

⚠ Il **COMPARATOR** può essere usato in una classe usando `implements Comparator` e `@Override` sul metodo `compare` (per modificare il compare per un confronto di cui ho bisogno); questo può essere usato nei sorting come `Array.sort(vector, comparatore)` con comparatore un oggetto di interfaccia Comparator.

Spesso il Comparator (e altre interfacce funzionali) sono usate senza scriverli in una classe apposita, usando:

- **CLASSI ANONIME**:

```
Comparator comparatore = new Comparator(){
    @Override
    public int compare(Object o1, Object o2) {
        double re1 = ((Complex)o1).re();
        double re2 = ((Complex)o2).re();

        if( re1 < re2 ) return -1;
        if( re1 > re2 ) return +1;
        return 0;
    }
};

Arrays.sort(v, comparatore);
```

- **ESPRESSIONI LAMBDA** (o direttamente senza una variabile come si vede a destra):

```
Comparator comparatore = ( o1, o2 ) -> {
    double re1 = ((Complex)o1).re();
    double re2 = ((Complex)o2).re();

    if( re1 < re2 ) return -1;
    if( re1 > re2 ) return +1;
    return 0;
};

Arrays.sort(v, comparatore);
```



```
Arrays.sort(v, ( o1, o2 ) -> {
    double re1 = ((Complex)o1).re();
    double re2 = ((Complex)o2).re();

    if( re1 < re2 ) return -1;
    if( re1 > re2 ) return +1;
    return 0;
});
```

Le **LAMBDA FUNCTIONS** (ESPRESSIONI LAMBDA) hanno sintassi **(parametro1, parametro2, ...) -> espressione** (dove espressione è il valore restituito; ma posso avere anche un blocco di codice direttamente al posto di una singola espressione con **(parametro1, parametro2, ...) -> {blocco di codice}**). Le lambda sono collegate alle **interfacce funzionali** (ovvero quelle con un unico metodo astratto); es. se ho **interface Operazione {int esegui(int a, int b);}** posso usare la lambda per implementare la somma con **Operazione somma = (a, b) -> a + b;**.

**⚠ Se un'interfaccia ha più di un metodo astratto (non interfaccia funzionale), non può essere utilizzata con una lambda function!**

Spesso il tipo dei parametri può essere omesso e può essere inferito (compreso dal compilatore) dal contesto **[Type inference]**. Le lambda spesso sono usate con le collezioni di dati (es. `List<Integer> numeri = Arrays.asList(1, 2, 3, 4, 5);` da cui con la lambda `numeri.forEach(numero -> System.out.println(numero));`). Nelle lambda possiamo anche mettere delle variabili oltre che i parametri che definisco direttamente nella lambda.

Un **METHOD REFERENCE** è un modo compatto per rappresentare un'interfaccia funzionale (singolo metodo); è un "trick" sintattico per scrivere in modo conciso il codice quando si usano le lambda. In sintesi, i method reference offrono un modo compatto per riferirsi a metodi esistenti, evitando di definire lambda functions con corpi di metodo separati. Esistono 4 principali method reference:

- Reference a **metodo static di una classe** → `Classe :: nomeMetodoStatico` (es. `Function<String, Integer> parser = Integer::parseInt;`);
- Reference a **metodo di un oggetto specifico** → `oggetto :: methodName` (es. `String nome = "Hello"; con Consumer<Integer> consumer = nome::length;`);
- Reference a **metodo di un oggetto con tipo specificato** → `Tipo :: methodName` (es. `List<String> lista = Arrays.asList("Java", "is", "awesome"); con lista.forEach(System.out::println);`);
- Reference ad un **costruttore di una classe** → `Classe :: new` (es. `Supplier<List<String>> supplier = ArrayList::new;`);

**⚠ Le lambda functions sono spesso trasformate in metodi di classi interne anonime** (a volte chiamati "lambdas methods") **dal compilatore Java**, usati per creare oggetti funzionali. ESEMPIO:

```
interface Operazione {
    int esegui(int a, int b);
}

public class Esempio {
    public static void main(String[] args) {
        Operazione somma = (a, b) -> a + b;
        int risultato = somma.esegui(3, 4);
        System.out.println("Risultato: " + risultato);
    }
}
```

Dopo la compilazione, il compilatore Java può trasformare la lambda function in un metodo di classe interna anonima all'interno della classe "Esempio", ovvero:

```
public class Esempio {
    static int lambda$1(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Operazione somma = Esempio::lambda$1;
        int risultato = somma.esegui(3, 4);
        System.out.println("Risultato: " + risultato);
    }
}

// il n° # dopo lambda$# è incrementato ad ogni metodo creato da una lambda (per nominarlo)
```

## 5) GENERICS

I GENERICS (tipi/parametri generici) permettono di creare classi, interfacce e metodi che possono operare con tipi specifici, senza però specificare il tipo in anticipo (maggiore flessibilità, tipi dinamici, codice riutilizzabile per tipi diversi). Sono usati al posto di Object perché quest'ultimo può essere usato per fare da "sostituto" ad ogni tipo, ma non consente un controllo sugli oggetti/tipi che poi metto al posto di Object (es. nello stesso vettore di Object posso mettere elementi di tipi diversi e questo può generare errori). I parametri generici sono indicati con le lettere maiuscole singole T (Type), E (Element), R (Return), K (Key), V (Value). Per esempio posso definire una CLASSE GENERICA che usa parametri generici:

```
public class Box<T> {
    private T contenuto;

    public void setContenuto(T contenuto) {
        this.contenuto = contenuto;
    }

    public T getContenuto() {
        return contenuto;
    }
}
```

Quando uso questa classe, il tipo dei parametri generici viene specificato tra parentesi angolari (o "diamond operator"), ovvero:

```
Box<String> boxStringa = new Box<>();
boxStringa.setContenuto("Hello");
String contenuto = boxStringa.getContenuto();
```

Possiamo usare i generics anche nei metodi, dove il parametro generico viene dichiarato prima del tipo di ritorno del metodo (es. `public <T> T getLastElement(List<T> lista) {...}`). Le interfacce di default (come Comparable, Comparator, Iterable, Iterator etc...) sono implementate usando proprio i Generics (es. `public interface Comparable<T> {...}`, che poi quando dovrò usare in una classe specifica [es. Student] farò `public class Student implements Comparable<Student>`). Così come i Generic Iterable e Iterator:

<pre>public interface List&lt;E&gt;{     void add(E x);     Iterator&lt;E&gt; iterator(); }</pre>	<pre>public interface Iterator&lt;E&gt;{     E next();     boolean hasNext(); }</pre>
---	---

Al posto dei tipi generici indicati con lettere maiuscole, possiamo anche usare i WILDCARD che rappresentano un tipo generico sconosciuto (es. `public void stampaLista(List<?> lista) {...}`).

Parlando appunto dei contenitori, questi possono essere:

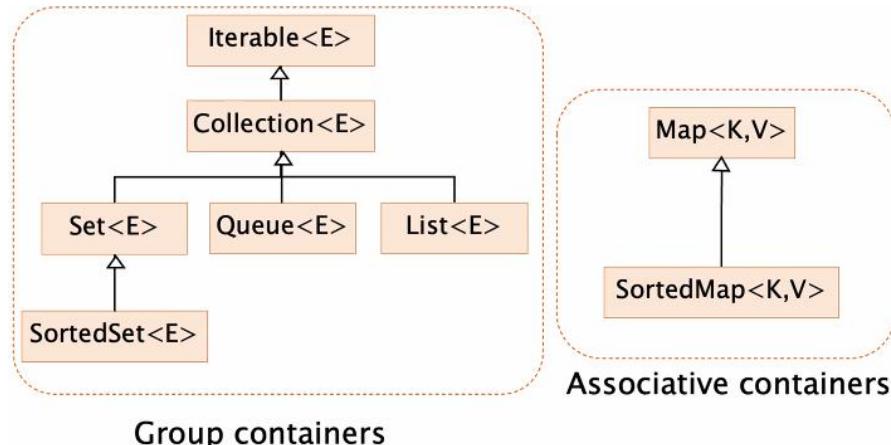
- **CO-VARIANTI**: se A extends B → `Container<A> extends Container<B>` (es. questo vale nei vettori/array, infatti `Integer extends Object` e `Integer[] extends Object[]`);
- **INVARIANTI**: se A extends B → `Container<A> non estende Container<B>` (es. questo vale nei generics, infatti `Integer extends Object` ma `Pair<Integer> non estende Pair<Object>`). Per risolvere questo problema nei generics, usiamo le WILDCARDS (viste sopra, ? ["unknown" class]) con i BOUNDED TYPES, cioè parliamo di:
  - o **Upper Bound** = `List<? extends Number>` indica che la lista potrà contenere solo elementi che estendono la classe Number (es. `Integer` perché `Integer` sottoclasse di `Number`);
  - o **Lower Bound** = `List<? super Integer>` indica che la lista potrà contenere solo elementi di tipo `Integer` o dei suoi super tipi (es. `Number`, perché `Number` è superclasse di `Integer`).

⚠ Le classi corrispondenti ai tipi generici vengono generate con la "TYPE ERASURE", ovvero durante la compilazione, tutte le informazioni sui tipi generici sono rimosse e il codice sorgente viene convertito in un formato più compatibile con le versioni precedenti di Java che non supportano i generics (retrocompatibilità).

Quindi durante l'esecuzione del programma, se ho `List<String>`, questa sarà vista in runtime come `List` ("raw-type") senza alcuna informazione sul tipo specifico di elementi contenuti in essa. Questo porta a perdita di informazioni a tempo di esecuzione (con conseguente limitazione alla riflessione [quando un programma si autocapisce] e cast impliciti errati).

## 6) COLLEZIONI

Il **Java Collections Framework** sono delle interfacce (ovvero come gli ADT in C) con le loro implementazioni e algoritmi (tutto questo framework è contenuto in `java.util`); all'inizio usavano Object, ma ora da Java 5 usano i tipi generici. Iniziamo dalle interfacce, divise in **Group containers** (interfacce con `<E>` dove E è il tipo di elemento) e **Associative containers** (interfacce con `<K,V>` dove K = key, V = value; quindi una sorta di dizionario):



⚠ L'interfaccia `Set<E>` non aggiunge nulla a `Collection<E>`, ma introduce le caratteristiche degli insiemi.

Più in particolare abbiamo:

- **List** → raccolte ordinate di elementi in cui ogni elemento ha un indice (parte da 0). Implementazioni sono `ArrayList` (mix tra vettori e liste) e `LinkedList` (liste concatenate). I metodi principali utili sono:
  - `add(E elemento)` → aggiunge elemento in fondo a lista;
  - `addAll(Collection<? extends E> collezione)` → aggiunge tutti elementi in fondo a lista;
  - `get(int indice)` → restituisce elemento alla posizione specificata della lista;
  - `set(int indice, E elemento)` → sostituisce elemento alla posizione specificata con quello nuovo;
  - `remove(int indice)` → rimuove elemento alla posizione specificata della lista;
  - `isEmpty()` → verifica se lista vuota;
  - `size()` → restituisce numero di elementi della lista;
  - `contains(E elemento)` → verifica se lista contiene elemento (come `in`);
  - `indexOf(E elemento)` → posizione della prima occorrenza dell'oggetto nella lista (-1 se non c'è);
  - `lastIndexOf(E elemento)` → posizione dell'ultima occorrenza dell'oggetto nella lista (-1 se non c'è);
  - `clear()` → rimuove tutti gli elementi dalla lista;
  - `subList(int da, int a)` → restituisce una lista derivata dalla lista, prendendo solo gli elementi dalla posizione "da" alla posizione "a".
- **Set** → collezioni che non contengono duplicati. Implementazioni sono `HashSet` (non ordinato e più veloce), `TreeSet` (ordinato, quindi più lento) e `LinkedHashSet` (mantiene ordine di inserimento e veloce);
- **Map** → come dizionari con chiave e valore. Implementazioni sono `HashMap`, `TreeMap` e `LinkedHashMap` (stesse differenze tra le implementazioni di Set). I metodi principali utili sono:
  - `put(K chiave, V valore)` → associa il valore specificato alla chiave specificata. Se la mappa conteneva un valore diverso associato a quella chiave, questo verrà sovrascritto;
  - `putAll(Map<? extends K, ? extends V> mappa)` → copia la mappa nella mappa;
  - `get(K chiave)` → restituisce valore associato alla chiave, o null se chiave non ha valore associato;
  - `remove(K chiave)` → rimuove valore della chiave;
  - `containsKey(K chiave)` → verifica se mappa contiene un mapping per la chiave;
  - `containsValue(V valore)` → verifica se mappa contiene almeno un mapping per il valore;

- `isEmpty()` → verifica se mappa vuota;
  - `size()` → restituisce numero di mapping della mappa;
  - `keySet()` → restituisce set con tutte le chiavi della mappa;
  - `values()` → restituisce collection con tutti i valori della mappa;
  - `entrySet()` → restituisce set con tutti gli entry (ovvero coppie chiave-valore) della mappa;
  - `clear()` → rimuove tutti i mapping dalla mappa.
- **Queue** → collezioni che usano regole come FIFO o LIFO. Implementazioni sono **PriorityQueue** e **LinkedList**.

Le varie interfacce hanno **diversi metodi di implementazione** (es. il set può essere implementato con Hash Table, con Balanced Tree, mentre le List con Array dinamici o Linked list, etc...).

Vediamo un'esempio di dichiarazione di interfacce:

```
protected Collection<NutritionalElement> products = new ArrayList<>(); // casting
protected ArrayList<Menu> menus = new ArrayList<>(); // no casting
protected ArrayList<String> hours = new ArrayList<>();
```

⚠ Nelle interfacce, da Java 8, si può usare la parola chiave **default** come scope di un metodo per fare in modo che, nelle classi che implementano l'interfaccia, possiamo decidere se fare **@Override** dei metodi, oppure lasciare l'implementazione definita nell'interfaccia del metodo con default.

## 7) ECCEZIONI

Le **ECCEZIONI (EXCEPTIONS)** in Java sono errori che si verificano durante l'esecuzione del programma e possono interrompere il flusso di esecuzione del nostro programma; queste possono essere **gestite** (intercettate) o **propagate**. Posso gestire le eccezioni:

- Con **try-catch** → inserisco il codice che potrebbe generare un'eccezione nel blocco try e, se viene lanciata un'eccezione, viene **catturata e gestita nel metodo corrente** nel blocco catch corrispondente. Vediamo un esempio di try-catch con eccezioni di default (es. **ArithmaticException**):

```
try{
    int result = 10/0;
} catch (ArithmaticException e){
    System.out.println("Errore: divisione per zero");
    e.printStackTrace();
}
```

- Con **throws-throw** → se un'eccezione **non viene gestita nel metodo corrente**, viene **propagata** ai metodi chiamanti fino a quando non viene intercettata o finisce il flusso di esecuzione, causando un'interruzione del programma (quindi deve essere gestita da chi chiama il metodo). Il metodo corrente (che non gestisce un'eccezione) ha, dopo la parentesi tonda con i parametri, la frase **throws Exception**; dentro al metodo avremo **throw new Exception("messaggio di errore")**:

```
public void metodo() throws Exception{
    throw new Exception("Errore");
}
```

Questo accade perché in Java abbiamo **2 tipi di eccezioni**: **CONTROLLATE (checked)** = devono essere **gestite obbligatoriamente**, altrimenti il codice non viene compilato; **NON CONTROLLATE (unchecked**, es. **NullPointerException** o **ArrayIndexOutOfBoundsException**) = possono essere **anche propagate**.

⚠ Le eccezioni con try e poi tutte le eccezioni servono per evitare di avere delle righe di codice legate alla gestione degli errori inframmezzate con le righe di codice del programma utile.

⚠ In Java un'eccezione è un **oggetto della classe Exception** con **Exception e = new Exception("Messaggio di errore")**. **Se un metodo può causare un'eccezione**, lo scrivo dopo i parametri del metodo con **throws Exception**. Da qui faccio **throw e** (quindi dal metodo scrivo throws Exception, mentre poi nel codice scrivo **throw new Exception("messaggio di errore");**).

## 8) I/O STREAM

La libreria `java.util.stream` permette di manipolare in maniera efficiente **raccolte di dati (stream)**, ovvero una sequenza di dati che supporta operazioni sequenziali e parallele; può essere creato da una collection, da un array o da un file). Sugli stream distinguiamo **OPERAZIONI INTERMEDI** (input = stream, output = stream; si fanno operazioni di filtraggio, mapping, modifica, etc... sullo stream, dando uno stream modificato) e **OPERAZIONI TERMINALI** (input = stream, output = risultato; opero sullo stream, ottenendo un risultato [es. counter]). Abbiamo gli **stream di chars (16 bit perché Unicode)** e **stream di bytes (8 bit)**; tutti i dati binari, suoni e immagini). Tutte le eccezioni degli stream sono sottoclassi di `IOException`. Un esempio di uso degli stream è:

```
Stream.of(parole) // parole = sorgente dello stream
    .sorted() // sorted, distinct e limit sono OPERAZIONI INTERMEDI
    .distinct()
    .limit(4)
    .forEach(System.out::println); // forEach è OPERAZIONE TERMINALE
```

Ci sono diversi modi per generare uno stream:

- **Da collezione** (es. `List<Integer> numbers`) → `Stream<Integer> stream = numbers.stream();`
- **Da array** (es. `String[] array`) → `Stream<String> stream = Arrays.stream(array);`
- **Da valori specifici** → `Stream<Integer> stream = Stream.of(1,2,3);`
- **Stream infinito con generate** (viene infatti solitamente posto dopo generate il metodo `limit`) → `Stream<String> stream = Stream.generate(() -> "stringa").limit(10);`
- **Stream con iterate** → `Stream<Integer> stream = Stream.iterate(0, n -> n+2).limit(10);`

Alcuni **metodi intermedi utili** degli stream sono:

- **filter(condizione)** → filtra gli elementi e restituisce lo stream filtrato;
- **map(funzione)** → applica una funzione lambda ad ogni elemento dello stream e restituisce il nuovo stream;
- **flatMap(funzione)** → uguale a prima, ma se la funzione restituisce uno stream invece di un singolo risultato, i risultati degli stream interni vengono appiattiti in uno stream singolo;
- **sorted()** o **sorted(Comparator)** → ordina gli elementi dello stream;
- **distinct()** → rimuove gli elementi duplicati dallo stream;
- **findFirst()** → restituisce il primo elemento dello stream che rispecchia una condizione specificata;
- **limit(n)** → restituisce uno stream con solo i suoi *n* elementi iniziali;
- **skip(n)** → restituisce uno stream che skippa gli *n* elementi iniziali;

Alcuni **metodi terminali utili** degli stream sono:

- **forEach(azione)** → esegue un'azione (azione sarebbe un `Consumer`) per ogni elemento dello stream senza restituire un risultato (es. `forEach(System.out::println);`);
- **findAny()** → ritorna un qualsiasi elemento dello stream (ordine non conta). Torna `Optional<T>` (ovvero oggetto opzionale di ritorno);
- **findFirst()** → ritorna il 1° elemento dello stream (ordine conta). Torna `Optional<T>`;
- **min(Comparator)** → ritorna il minimo (basato sul comparatore). Torna `Optional<T>`;
- **max(Comparator)** → ritorna il minimo (basato sul comparatore). Torna `Optional<T>`;
- **reduce()** → combina gli elementi dello stream usando un'operazione binaria e dà un risultato. Per una maggiore efficienza si fanno solitamente in parallelo sugli elementi dello stream;
- **collect(Collector)** → raccoglie gli elementi dello stream usando un oggetto Collector. Può essere usato per convertire lo stream in una lista, in una mappa, in un set o in un'altra struttura dati;
- **count()** → ritorna il numero di elementi nello stream;
- **anyMatch(Predicate)** → controlla se qualche elemento nello stream soddisfi il predicato. Torna boolean;
- **allMatch(Predicate)** → controlla se tutti gli elementi nello stream soddisfano il predicato. Torna boolean;
- **noneMatch(Predicate)** → controlla che nessun elemento nello stream soddisfi il predicato. Torna boolean.

Ci sono poi classi di Stream per i numeri come **DoubleStream**, **IntStream**, **LongStream** (trattabili come gli Stream visti sopra, ma hanno anche dei metodi utili per lavorare con i numeri [come `average`]); questi non hanno boxing e unboxing dai Wrapper dei tipi (es. `Stream<Integer>`), dunque sono **più efficienti**.

Legato alla `collect` abbiamo i **COLLECTORS**:

- **ACCUMULATING COLLECTORS** → se negli stream uso `collect(Collectors.toList())` raccolgo gli elementi dello stream in una lista (analogamente avrò `Collectors.toSet()`, `Collectors.toMap()`, `Collectors.joining()` [concatena gli elementi dello stream in una Stringa; posso specificare i separatori] e `Collectors.toCollection(Supplier<> cs)` [es. `TreeMap::new` per creare una `TreeMap`]).
- **GROUPING COLLECTORS** → ho:
  - o `Collectors.groupingBy(Classifier, Supplier*, Downstream*)` = raccoglie gli elementi di uno stream in un Downstream [ovvero lo stream derivato] specificato (se non specificato di default una List) in base ad un Classifier (funzione di raggruppamento specificata).  
**Esempio** (raggruppare persone per età): `Map<Integer, List<Person>> peopleByAge = people.stream().collect(Collectors.groupingBy(Person::getAge));`
  - o `Collectors.partitioningBy(Predicate)` = suddivide gli elementi di uno stream in base ad un predicato. Restituisce una Map dove le chiavi sono vari booleani (True/False) e i valori sono le liste degli elementi dello stream che soddisfano e non soddisfano il predicato.  
**Esempio** (dividere uno stream in numeri pari e numeri dispari): `Map<Boolean, List<Integer>> partitionedNumbers = numbers.stream().collect(Collectors.partitioningBy(num -> num % 2 == 0));`
  - o `Collectors.collectingAndThen(Collector, Function)` = applica una funzione terminale ad un risultato di Collector (un downstream).  
**Esempio** (concatenare parole in una stringa e aggiungere un prefisso "Ciao" al risultato finale): `collect(Collectors.collectingAndThen(Collectors.joining(", "), result -> "Ciao: " + result));`
  - o `Collectors.mapping(Function, Collector)` = applicare agli elementi di uno stream una Function prima di raccoglierli in una struttura dati specificata (Collector).  
**Esempio** (mappare le parole in lunghezze di stringhe intere prima di raccoglierle in una lista): `collect(Collectors.mapping(String::length, Collectors.toList()));`

L'interfaccia Collector ha sintassi:

```
public interface Collector<T, A, R> { // T = tipo elementi, A = accumulator, R = result
    Supplier<A> supplier(); // crea l'accumulator (una collection intermedia che fa da container)
    BiConsumer<A, T> accumulator(); // aggiunge un nuovo elemento all'accumulator
    BinaryOperator<A> combiner(); // combina 2 accumulator (usato per parallelismo)
    Function<A, R> finisher(); // trasforma accumulator intermedio in result
    Set<Characteristics> characteristics(); // proprietà del container
}
```

⚠ Le lambda e i method reference negli stream non posso fare throw delle eccezioni!

## 9) REGULAR EXPRESSIONS

In Java, come in tutti i linguaggi, abbiamo la possibilità di usare le **REGEX** (usate per matching, manipolazione e estrazione di stringhe; libreria `java.util.regex`), ovvero sequenze di caratteri letterali ("a" = lettera 'a') e metacaratteri (caratteri speciali con significati specifici) per definire un modello di ricerca. Metacaratteri comuni sono:

- ‘.’ = ogni carattere eccetto \n;
- ‘^’ = inizio stringa;

- '\$' = fine stringa;
- '\*' = 0 o più occorrenze del carattere precedente;
- '+' = 1 o più occorrenze del carattere precedente;
- '?' = 0 o 1 occorrenza del carattere precedente;
- '[abc]' = 1 carattere tra 'a', 'b' e 'c';
- '[^abc]' = 1 carattere eccetto 'a', 'b' e 'c';
- '\' = escape per metacaratteri.

Come **metodi** distinguiamo tra metodi di:

- **PATTERN** → rappresentazione compilata di una regex:
  - o `compile(String regex)` = compila una regex in un Pattern
  - o `compile(String regex, int flags)` = compila una regex con flag (`Pattern.CASE_INSENSITIVE`, `Pattern.MULTILINE` [modifica le interpretazioni di '^' e '\$' per fare in modo che siano inizio e fine della riga e non della stringa], `Pattern.DOTALL` [il '.' può essere anche '\n'])
- **MATCHER** → oggetto che esegue il matching della regex su una stringa:
  - o `matches()` = verifica se l'intera stringa corrisponde al pattern dando True o False
  - o `find()` = cerca la prossima occorrenza del pattern nella stringa
  - o `group()` = restituisce la sottostringa che ha corrisposto al pattern. Possiamo usare più group di fila per prendere le singole sottostringhe. Vediamo un esempio:

```
Pattern pattern = Pattern.compile("(\\d+)-(\\d+)-(\\d+)");
Matcher matcher = pattern.matcher("123-456-789");
if (matcher.matches()) {
    System.out.println("Group 1: " + matcher.group(1));
    System.out.println("Group 2: " + matcher.group(2));
    System.out.println("Group 3: " + matcher.group(3));
}
// Output:
// Group 1: 123
// Group 2: 456
// Group 3: 789
```

- o `start()` = restituisce l'indice iniziale del match
- o `end()` = restituisce l'indice finale del match

Esempio:

```
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matcher("123abc456def");
while (matcher.find()) {
    System.out.println("Found: " + matcher.group());
}
// Output:
// Found: 123
// Found: 456
```

Il metodo `replaceAll(String old, String new)` della classe `String` (o del `Matcher`) può essere usato per sostituire tutte le occorrenze che corrispondono al pattern con una stringa specificata (es. `String input = "1 2 3"; String result = input.replaceAll("\\s", "_");` rimpiazza gli spazi con trattini bassi ottenendo "1\_2\_3").

## 10) I/O

In Java le operazioni di **I/O** si basano sull'**astrazione degli stream**, che possono essere collegati a diverse sorgenti e destinazioni (come file su disco, stdin e stdout, connessioni di rete, dispositivi hardware etc...). L'I/O Java ha le sue eccezioni nella classe **IOException**. Le operazioni di I/O sono uguali per tutti gli stream, con l'unica distinzione tra:

- **Stream di CARATTERI** → usano caratteri Unicode (char in Java = 16 bit) e sono rappresentati dalle classi **Reader** e **Writer** (ideali per la gestione di dati testuali);
- **Stream di BYTE** → usano byte (8 bit) e sono rappresentati dalle classi **InputStream** e **OutputStream** (ideali per la gestione di dati binari come immagini e suoni).

Quindi:

- **Reader**: sue sottoclassi sono **BufferedReader**, **FileReader**, **InputStreamReader**. I suoi metodi sono:
  - **int read()** = legge 1 carattere
  - **int read(char[] buf)** = legge caratteri in un buffer/array
  - **void close()** = chiude lo stream
- **Writer**: sue sottoclassi sono **BufferedWriter**, **FileWriter**, **OutputStreamWriter**. I suoi metodi sono:
  - **void write(int c)** = scrive 1 carattere
  - **void write(char[] buf)** = scrive un array di caratteri
  - **void close()** = chiude lo stream, svuotando prima il buffer
- **InputStream**: sue sottoclassi sono **FileInputStream**, **ByteArrayInputStream**, **BufferedInputStream**. I suoi metodi sono:
  - **int read()** = legge byte successivo
  - **int read(byte[] buf)** = legge byte in un buffer/array
  - **void close()** = chiude lo stream
- **OutputStream**: sue sottoclassi sono **FileOutputStream**, **ByteArrayOutputStream**, **BufferedOutputStream**. I suoi metodi sono:
  - **void write(int b)** = scrive 1 byte
  - **void write(byte[] buf)** = scrive un array di byte
  - **void close()** = chiude lo stream, forzando la scrittura di byte memorizzati nel buffer

⚠ L'uso di stream bufferizzati (come **BufferedReader** e **BufferedWriter**) migliora le **prestazioni** dell'I/O perché riduce il numero di accessi fisici al disco (al posto di leggere/scrivere un carattere o byte alla volta, si leggono/scrivono blocchi di grandi dimensioni nei/dai buffer e si lavora con essi) [**BUFFERED I/O**].

⚠ La **conversione tra byte e caratteri** può essere fatta con **InputStreamReader** e **OutputStreamWriter**, che consentono di specificare un charset per decodificare/codificare i byte in caratteri e viceversa.

⚠ È importante chiudere gli stream di I/O il prima possibile per liberare le risorse di sistema; la gestione delle risorse può essere semplificata con il costrutto **try-with-resources** che garantisce la **chiusura degli stream anche in caso di eccezioni**. Il blocco try dichiara 1 o più risorse che implementano l'interfaccia **AutoCloseable**. Vediamone un **ESEMPIO**:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {  
    String line = br.readLine();  
    System.out.println(line);  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

La **SERIALIZZAZIONE** consente di salvare/ripristinare oggetti Java con **ObjectInputStream** e **ObjectOutputStream**. Posso però serializzare solo gli oggetti che implementano l'interfaccia **Serializable** (anche gli oggetti referenziati devono implementare **Serializable** e i campi che non devono essere serializzati possono essere dichiarati **transient**).

Vediamo un **ESEMPIO** di copia di un file di testo, usando un buffer per migliorare le prestazioni:

```

Reader src = new FileReader("source.txt");
Writer dest = new FileWriter("dest.txt");
char[] buffer = new char[4096];
int n;
while((n = src.read(buffer)) != -1) {
    dest.write(buffer, 0, n);
}
src.close();
dest.close();

```

Oltre a ciò, c'è anche la classe **PrintStream** (sempre del pacchetto **java.io**) usata per produrre **output formattato** (sottoclasse di **OutputStream**, quindi lavoriamo su byte). Fornisce metodi come **print()**, **println()** e **printf()**; i metodi di **PrintStream**, a differenza degli altri I/O, **non produce IOException**, bensì mantengono uno **stato interno di errore** che può essere controllato con **checkError()**. **PrintStream** usa buffer per migliorare le **prestazioni**.

```

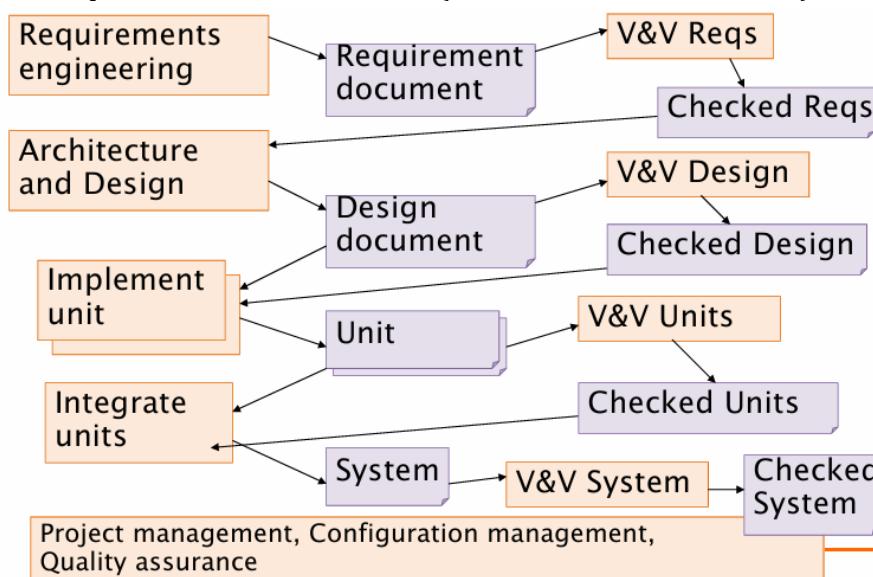
PrintStream ps = new PrintStream(new FileOutputStream("output.txt"));
ps.println("Hello, World!");
ps.printf("Numero: %d\n", 123);
ps.close();

```

→ **Design Pattern:** definiamo **PATTERN** = soluzione (riusabile) ad un problema noto in contesto ben definito. Dal punto di vista **SOFTWARE**, ci sono diversi tipi di pattern: **ARCHITECTURAL PATTERNS** (schema funzionale organizzato per i sistemi software, che garantisce un set di componenti predefiniti con le loro responsabilità), **DESIGN PATTERNS** (alto livello; schemi per sistemare classi/oggetti che descrivono i meccanismi e le regole di comunicazione tra questi componenti) e **IDIOMI** (basso livello; specifici per un linguaggio). Quando parliamo di **PATTERN LANGUAGE** (linguaggio di pattern) = presentiamo tutti i pattern assieme dando linee guida su implementazione, combinazione e usi pratici. Nel libro GoF, si ha il riferimento dei design patterns; i design pattern vengono classificati in base allo **scopo** (**creazionali, strutturali e comportamentali**) e in base al **contesto** (**classe, oggetti**). **Come si usano i pattern?** Bisogna capire quali sono i pattern che ci permettono di risolvere il problema; quindi si studia il loro funzionamento e la loro modalità d'uso.

## → SOFTWARE ENGINEERING

Parliamo di **SOFTWARE** (codice eseguito da hardware; ≠ da un semplice programma, in quanto il software è un "ecosistema" [programma, documentazione, regole di uso, dati etc...]) + **ENGINEERING** (design, costruzione e operazione), se parliamo dello sviluppo di sistemi software di grandi dimensioni da parte di 1 o più team di ingegneri. L'obiettivo è produrre software con delle proprietà ben definite (funzionalità, affidabilità e performance) in costi e tempi definiti. Ci sono delle fasi (**V&V** = verifica e validazione):



Lo sviluppo è solo la 1<sup>a</sup> parte: segue “**OPERATE THE SOFTWARE**” (deployment e operazioni), “**MODIFY THE SOFTWARE**” (manutenzione, dove si fanno le stesse cose dello sviluppo, ma qui ci si basa sul sistemare ciò che è stato scritto nello sviluppo) e “**TERMINATE THE USAGE**” (ritiro del software). Per organizzare il lavoro/processi, ci sono approcci principali:

- **BUILD & FIX** (o “**cowboy programming**”) = scrivere codice e basta, da soli;
- **DOCUMENT BASED** (“**semiformal**”, UML). Questo ha approcci:
  - o **Waterfall** = sequenziale (ogni attività produce un documento), ma poco flessibile (rigida sequenzialità, quindi errore = tutto da capo);
  - o **V** = waterfall con attività V&V;
  - o **Incremental** = basato su build successive (si ha fin da subito un feedback), può essere associato alla prototipazione;
  - o **Evolutionary** = come incrementale, ma le richieste possono cambiare ad ogni iterazione;
  - o **Iterative** = molte iterazioni, dove ogni iterazione è un piccolo progetto (tipo waterfall);
  - o **Prototyping** = c’è una prototipazione (prototyping language, uso di Matlab per esempio) che precede il waterfall (evita errori inutili iniziali);
  - o **Open Source**;
  - o Unified Process (**UP**) [complesso non lo vediamo qui].
- **MODEL BASED** (“**formal**”, formal UML) = linguaggio formale per documenti e controlli automatici;
- **AGILE** = uso limitato dei documenti. Più importanti gli individui e la loro interazione, dei processi e tools. Testing man mano che vado avanti, rilascio del prodotto presto e spesso (in modo da avere feedback), fai la documentazione man mano (ma solo se richiesta) e costruisci team cross-funzionali (che collaborano).

## 11) DATE e TIME

Ci sono diversi modi per gestire **DATA e ORA** in Java:

- Time stamps → metodo peggiore (riferimento dal 1/1/1970), basato sul riferimento del **sistema operativo**: abbiamo infatti `currentTimeMillis()` = differenza in millisecondi dal 1/1/1970, e `nanoTime()`;
- `java.util.Date` → no time zones, il suo costruttore è deprecato (anni partono da 1900 e mesi da 0);
- `java.util.Calendar` → ancora incompleta, solo data (non orario);
- **java.time** → la migliore API [introdotta in Java 8] per gestire data e ora (useremo questa). I suoi principi guida sono **semplicità, coerenza** (approccio uniforme) e **immutabilità** (thread-safe).

Le sue classi principali sono:

- o **ISTANTI TEMPORALI**:
  - **Instant** = punto preciso nel tempo (ora locale del computer);
  - **LocalDate** = data senza ora (es. 2024-05-23) [AAAA-MM-GG];
  - **LocalTime** = ora senza data (es. 13:45:30) [HH:MM:SS];
  - **LocalDateTime** = combina LocalDate e LocalTime, ma senza informazioni di fuso orario;
  - **ZonedDateTime** = aggiunge il fuso orario a LocalDateTime.
- o **INTERVALLI TEMPORALI**:
  - **Duration** = intervallo di tempo basato sull’ora (es. 5 secondi, 3 ore);
  - **Period** = intervallo di tempo basato sulla data (es. 2 anni, 3 mesi).

I suoi metodi sono:

- o **CREAZIONE**:
  - **of()** = crea un’istanza da una serie di parametri (es. `LocalDate.of(2024, 5, 23)` crea un oggetto LocalDate 23 maggio 2024, oppure `LocalTime.of(12, 45, 30)` crea un oggetto LocalTime 13:45:30 come orario, oppure anche `LocalDateTime.of(2024, 5, 23, 12, 45, 30)` per un oggetto LocalDateTime);
  - **from()** = converte da un’altra classe (con possibile perdita di informazioni);
  - **parse()** = analizzare una stringa per costruire un’istanza;
  - **now()** = crea un’istanza che rappresenta il tempo/data attuale.
- o **CONFRONTO**:

- **isBefore()** = verifica se un'ora/data è prima di un'altra ora/data specificata;
  - **isAfter()** = verifica se un'ora/data è dopo un'altra ora/data specificata;
  - **isEqual()** = verifica se un'ora/data è uguale a un'altra ora/data specificata;
  - **compareTo()** = confronta con un'altra ora/data.
- **MODIFICA:**
    - **minus()** = dà una nuova ora/data sottraendo il tempo specificato;
    - **plus()** = dà una nuova ora/data aggiungendo il tempo specificato;
    - **with()** = dà una nuova ora/data modificata come specificato da un **Temporal Adjuster**.

I Temporal Adjuster (contenuti nella classe **TemporalAdjusters**) hanno come metodi per esempio:

- **firstDayOfMonth()** = 1° giorno del mese;
- **lastDayOfMonth()** = ultimo giorno del mese;
- **firstInMonth(DayOfWeek giorno)** = 1° giorno specificato nel mese.

Per rappresentare **giorni** e **mesi** si usano le classi **DayOfWeek** e **Month** (rappresentati come enumerazioni); inoltre si può usare **getDisplayName(style, locale)** per convertire in stringa con opzioni di stile (FULL, NARROW, SHORT) e locale (rappresenta una specifica regione geografica, politica e culturale; usato nella formattazione di date, nomi dei giorni/mesi; definito da costanti predefinite, es. `Locale.US`, `Locale.ITALIAN`).

Per rappresentare **data/ora** si usa lo standard ISO-8601, ovvero: **aaaa-mm-ggThh:mm:ssZ** (es. 1970-01-01T00:00:00Z, dove la Z è la time zone).

Ci sono anche dei **metodi per la creazione di intervalli**:

- **of()** = crea un intervallo da una quantità specificata di **TemporalUnits** (es. `Duration.ofHours(3).plusMinutes(15).plusSeconds(30)` mi dà una durata di 3h, 15m, 30s, oppure `Period.of(2, 3, 5)` mi dà un periodo di 2 anni, 3 mesi e 5 giorni);
- **ofXXX()** = crea un intervallo da una quantità specificata di unità (es. giorni, ore);
- **between()** = crea un intervallo tra 2 punti temporali.

## → TESTING

La minimizzazione dei difetti nel codice è cruciale per ridurre i costi di sviluppo, perché i **costi di correzione dei difetti** aumentano man mano che il software progredisce nel ciclo di vita.

Con **V&V** (Verifica e Validazione) si intende:

- **VERIFICA** = assicura che il software sia costruito correttamente, concentrandosi su efficienza e correttezza
- **VALIDAZIONE** = assicura che il software soddisfi i bisogni dell'utente finale, valutando efficacia e affidabilità

Definiamo:

- **ERRORE (Error)** = commesso dal programmatore
- **DIFETTO (Fault/Bug)** = caratteristica del software che causa un fallimento
- **FALLIMENTO (Failure)** = evento di esecuzione in cui il software si comporta in un modo non voluto

Distinguiamo tra **ANALISI**:

- **STATIC** = analisi del codice e dei dati, eseguita senza eseguire il programma
- **DINAMICA** = comporta l'esecuzione di test per trovare difetti

Si definisce **TECHNICAL DEBT (DEBITO TECNICO)** il lavoro di sviluppo extra derivante da soluzioni temporanee che richiedono correzioni successive. Se non viene ripagato subito, aumenta complessità e costi di manutenzione.

Parlando di **TESTING** (operare un sistema/componente in condizioni specifiche per rilevare differenze tra il comportamento reale e il comportamento desiderato, scoprendo difetti nei prodotti software [solitamente si dà al software un certo input e se ne esamina l'output]), vediamo che ci sono **vari tipi** di test:

- **UNIT TESTING** = test di singoli moduli/componenti/unità
- **INTEGRATION TESTING** = test dei vari moduli, facendoli lavorare insieme
- **SYSTEM TESTING** = test sull'intero sistema (non singoli moduli, ma nel complesso)
- **ACCEPTANCE TESTING** = test del sistema da parte del cliente
- **REGRESSION TESTING** = ripetizione dei test dopo modifiche per assicurarsi che non siano stati introdotti nuovi difetti

Per quanto riguarda le **STRATEGIE DI TESTING**:

- **BLACK BOX TESTING** = test funzionali basati sui requisiti
- **WHITE BOX TESTING** = test strutturali basati sul codice

Parlando invece di **DEBUGGING** (diverso dal **TESTING**, in quanto sono eseguiti da ruoli diversi e in momenti diversi: il **TESTING** rileva i fallimenti, il **DEBUGGING** localizza i difetti [fault] associati a tali fallimenti e li rimuove), solitamente il processo segue questi passi:

- 1. Identificare il fallimento (failure)**
- 2. Localizzare il difetto (fault) che ha generato il fallimento**
- 3. Progettare e implementare la riparazione**
- 4. Re-test per vedere se ok!**

Con **ORACOLO** (**ORACLE**) si intende un meccanismo che determina se l'output di un software è corretto per un dato insieme di input (una sorta di oracolo, veggente); l'oracolo può essere umano (basato sulle specifiche dei requisiti o sul giudizio umano, soggetto ad errori) o automatico (ideale ma difficile da ottenere, basato su specifiche formali dei requisiti o su versioni precedenti del software [regression testing]).

## 11) THREADS

Un **THREAD** è l'unità base a cui l'OS assegna il **tempo di esecuzione (CPU time)**; può eseguire qualsiasi parte del codice del processo e condivide lo stesso spazio di memoria, variabili globali e risorse di sistema con gli altri thread dello stesso processo. L'ordine con cui i thread vengono eseguiti è **non deterministico (devo usare meccanismi di sincronizzazione [es. semafori] per avere la giusta sincronizzazione tra thread)** [ricorda che solo 1 thread alla volta può essere eseguito da 1 core].

Un **PROCESSO** (programma in esecuzione, che contiene più thread) è un'istanza di un'applicazione in esecuzione, con il suo spazio di memoria virtuale, codice, dati e risorse dell'OS.

La gestione dei thread è affidata al **JVM Scheduler** (parte della JavaVirtualMachine [**JVM**])); in alcuni OS, i thread della JVM sono mappati su quelli dell'OS.

**CREAZIONE e AVVIO DI UN THREAD** → 2 modi:

- **Estendere la classe Thread** = creare una nuova classe che estende Thread (**MyThread extends Thread**) e sovrscrive il metodo **run()** [**@Override**]. Da qui, creo l'oggetto/istanza del thread usando la mia classe (**MyThread t1 = new MyThread();**) e ne faccio l'avvio (**t1.start();**)
- **Implementare l'interfaccia Runnable** = meglio questo perché separa il compito da eseguire dal meccanismo di esecuzione. Crea la classe che implementa Runnable (**MyRunnable implements Runnable**) e scrivo l'unico metodo di Runnable, ovvero **run()**. Da qui creo un oggetto/istanza di MyRunnable (**MyRunnable m1 = new MyRunnable();**), la passo ad un oggetto/istanza di Thread (**Thread t1 = new Thread(m1);**) e ne faccio l'avvio (**t1.start();**)

⚠ In Java oggi non si usano questi 2 metodi, ma si usano gli **ESECUTORI** (`java.util.concurrent`): invece di creare e gestire manualmente i Thread, con gli Executor posso riutilizzare il pool (insieme) di thread (evitando di creare e distruggere ogni volta), gestirne la concorrenza e tutto questo in maniera facile:

- **Executor** = interfaccia di base con il metodo **execute(Runnable command)**
- **ExecutorService** = estende Executor con metodi aggiuntivi come **submit, shutdown e awaitTermination**
- **ThreadPoolExecutor** = implementa ExecutorService che usa pool di thread per eseguire i compiti

## **CREAZIONE DI UN ESECUTORE** → 4 modi di Executors:

- `ExecutorService executor = Executors.newFixedThreadPool(4);` → crea un pool con un numero fisso di thread (qui 4)
- `ExecutorService executor = Executors.newCachedThreadPool();` → crea un pool che crea nuovi thread secondo necessità, ma riutilizza i thread inattivi
- `ExecutorService executor = Executors.newSingleThreadExecutor();` → crea un executor che utilizza un singolo thread
- `ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);` → crea un pool che può eseguire compiti con un ritardo (qui 2) o periodicamente

## **SOTTOMISSIONE DI COMPITI ALL'ESECUTORE:**

- `executor.execute(new RunnableTask());` → esegue e basta
- `Future<?> future = executor.submit(new RunnableTask());` → submit torna un Future che può essere usato per controllare lo stato del compito (task) o ottenerne il risultato

## **TERMINAZIONE DELL'ESECUTORE:**

- `executor.shutdown();` → permette ai task già sottomessi di completarsi, ma non ne accetta di nuovi
- `List<Runnable> notExecutedTasks = executor.shutdownNow();` → tenta di fermare subito tutti i task attivi e ritorna una lista di quelli in attesa
- `executor.awaitTermination(60, TimeUnit.SECONDS);` → attende la terminazione dell'executor per un certo periodo di tempo (qui 60 secondi)

Altro problema è la **SINCRONIZZAZIONE DEI THREAD**, necessaria per prevenire problemi come race conditions e deadlock quando più thread accedono e manipolano risorse condivise. Si possono usare dei **BLOCCHI SYNCHRONIZED** che permettono di limitare l'accesso ad un pezzo di codice ad 1 solo thread per volta usando un monitor (una sorta di semaforo mutex che è associato ad ogni oggetto Java). **Esempio:**

```
synchronized (resourceA) {  
    resourceA.value = a;  
    resourceB.value = b;  
}
```

Qui il thread che esegue il blocco sincronizzato deve acquisire il monitor sull'oggetto resourceA prima di poter eseguire il codice contenuto nel blocco; quando termina, il monitor viene rilasciato, permettendo ad altri thread di acquisirlo e accedere alla risorsa.

Come i blocchi, **anche i metodi possono essere dichiarati come synchronized** per ottenere automaticamente il monitor dell'oggetto a cui il metodo appartiene quando viene chiamato. **Esempio:**

```
public synchronized void increment() {  
    count++;  
}
```

Qui ogni chiamata a `increment` richiede l'acquisizione del monitor dell'oggetto corrente, assicurando che solo 1 thread alla volta possa eseguire il metodo.

⚠ Ci sono anche i “thread-safe” containers (collezioni thread-safe), come `synchronizedList()`, `synchronizedCollection()`, `synchronizedMap()`.

Ci sono dei metodi per la **COMUNICAZIONE TRA THREAD** in Java (quindi semafori e accesso alla sezione critica):

- `wait()` → pone il thread chiamante in uno stato di attesa finché un altro thread non chiama `notify` o `notifyAll` sullo stesso oggetto. Quando un thread chiama `wait`, rilascia il monitor dell'oggetto;
- `notify()` → risveglia un singolo thread in attesa sul monitor dell'oggetto (non si può specificare quale thread viene svegliato però);
- `notifyAll()` → risveglia tutti i thread in attesa sul monitor dell'oggetto

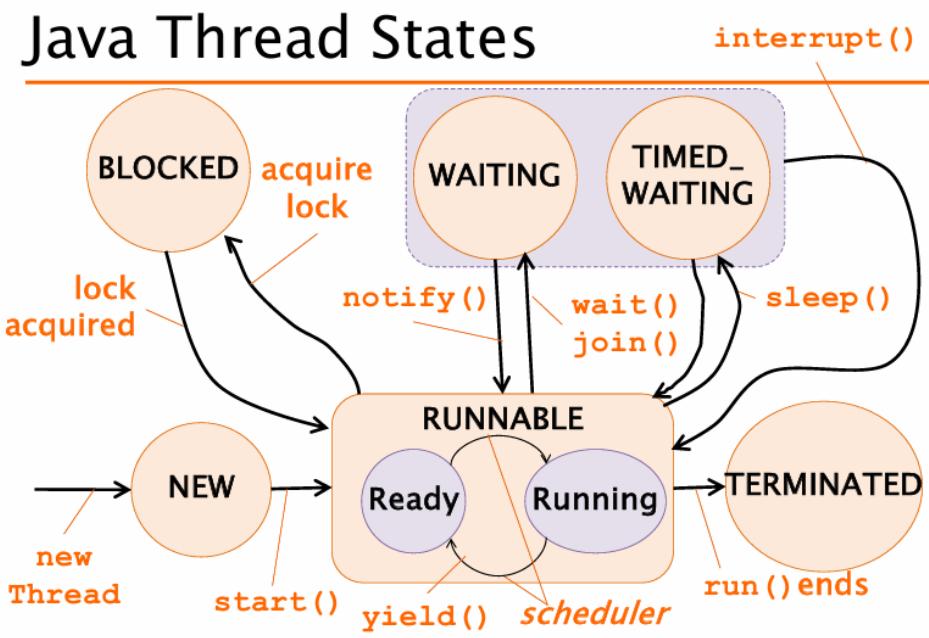
Altri problemi legati alla sincronizzazione sono:

- **WAKEUP SPONTANEO** = un thread può svegliarsi anche se nessun altro thread ha chiamato `notify`, a causa di motivi interni alla JVM. Per gestirlo, bisogna racchiudere la chiamata a `wait` in un ciclo `while` (per controllare la condizione di attesa).
- **LIVELOCK** = 2 o più thread continuano a cambiare stato in risposta alle azioni dell'altro, senza fare progressi (ricorda esempio di corridoio largo con persone che potrebbero passare, ma non passano).
- **STARVATION** = un thread non riesce mai ad acquisire il monitor dell'oggetto perché altri thread continuano ad ottenerne il controllo (passando davanti) [risolta con modifica della priorità e algoritmi di scheduling].

Ci sono anche **OGGETTI DI SINCRONIZZAZIONE** in Java, fondamentali per gestire la concorrenza dei thread:

- **Semaphore** → usato per controllare l'accesso ad una risorsa comune tramite un contatore che decrementa (con `acquire`) e incrementa (con `release`);
- **CountDownLatch** → permette ad 1 o più thread di aspettare fino a quando un insieme di operazioni eseguite da altri thread non è completato. Metodi principali sono `await` e `countDown`;
- **CyclicBarrier** → il costruttore accetta il n° di thread che devono raggiungere la barriera prima che tutti possano proseguire (doppia barriera). Metodo principale è `await`.

I thread possono trovarsi in diversi **STATI** durante il loro ciclo di vita:



1. **NEW** = creato ma non ancora avviato (non ancora `start`);
2. **RUNNABLE** = in esecuzione o pronto per l'esecuzione, ma in attesa di CPU time
3. **BLOCKED** = bloccato, in attesa di acquisire un lock su un monitor di un oggetto
4. **WAITING** = attesa indefinita fino a che un altro thread non lo risveglia con `notify` o `notifyAll`
5. **TIMED\_WAITING** = attesa per un tempo specificato
6. **TERMINATED** = completato l'esecuzione

⚠ In Java i thread hanno priorità d'esecuzione da 1 a 10 (le priorità influenzano la frequenza con cui i thread ottengono CPU time).

La JVM può avere diverse **politiche di scheduling** (ricorda da sistemi operativi):

- **Non-Preemptive** = un thread continua ad essere eseguito finché non termina o lascia la CPU (no prelazione)
- **Preemptive Time-Slicing** = thread eseguiti per un periodo di tempo specifico (time-slice, o quantum), dopo cui la JVM sospende il thread e ne avvia un altro (prelazione)
- **Priorità più alta**

Nel **DIAGRAMMA DEGLI STATI** vediamo che ci sono anche dei metodi utili:

- **yield()** → suggerisce allo scheduler di sospendere temporaneamente il thread corrente e permettere ad altri thread di pari priorità di ottenere CPU time
- **sleep()** → mette il thread corrente in stato di attesa (WAITING) per un tempo specificato
- **join()** → permette ad un thread di attendere la terminazione di un altro thread (sistemi operativi)
- **interrupt()** → suggerisce ad un thread di interrompere la sua esecuzione (se chiamato su un thread in stato di sleep/wait, throw InterruptedException)

⚠ Un test si dice “**Flacky**” se la sua esecuzione non mi dà sempre lo stesso risultato, pur non cambiando il codice (es. in caso di race condition) [cioè **può passare alcune volte, ma altre volte no**; bisogna fare attenzione specialmente nella programmazione concorrente].

⚠ Per gli esami presi da gitlab bisogna fare prima fork dalla pagina di Torchiano, e solo dopo il fork si creerà un fork del progetto nella nostra pagina gitlab; da lì faccio il solito clone with http e apro su VsCode.