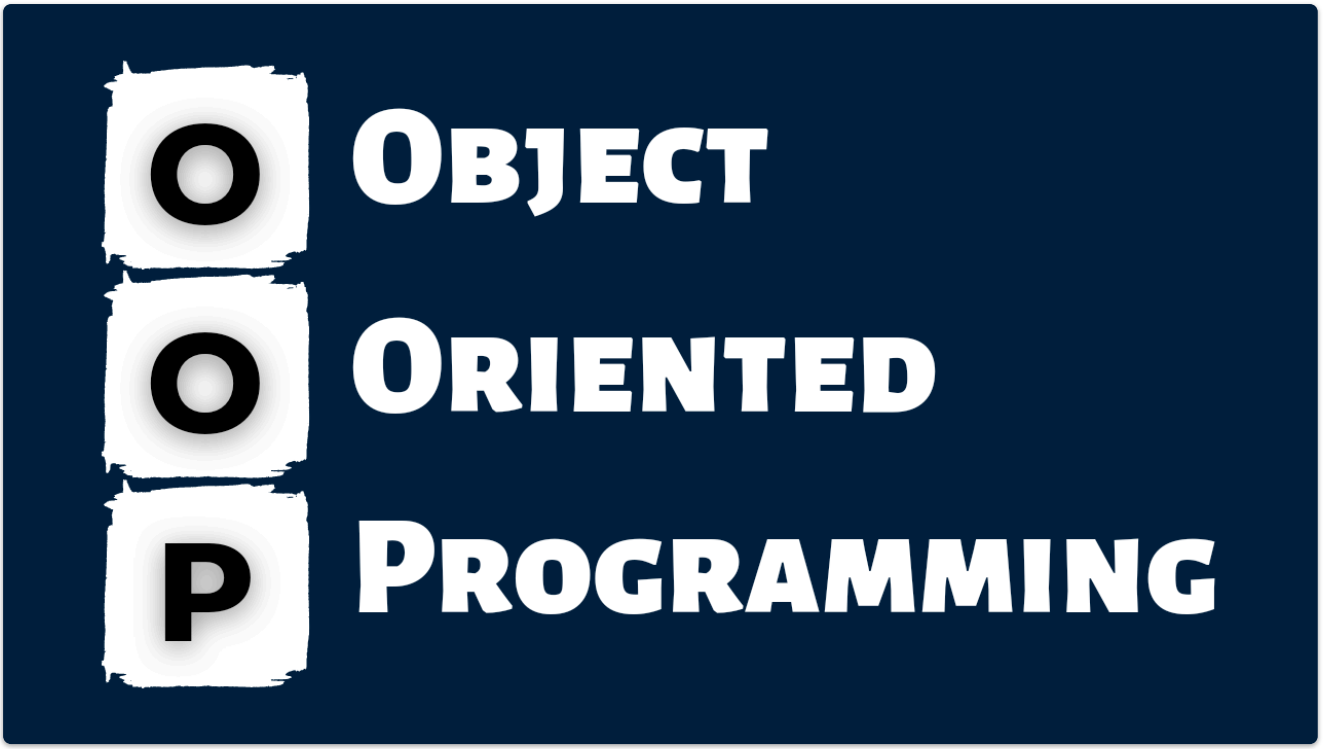

GIT

Il controllo di versione è una pratica fondamentale nello sviluppo software moderno, e Git rappresenta lo standard de facto tra i sistemi di Version Control System (VCS) distribuiti. Al contrario di sistemi come Subversion, Git permette a ogni sviluppatore di avere una copia completa del repository, garantendo autonomia e resilienza anche in assenza di connessione remota.

Configuration Management e Change Control

Il *Configuration Management* è la disciplina che si occupa della gestione delle modifiche al software, assicurando che ogni componente sia identificabile, tracciabile e riproducibile nel tempo. Ogni elemento soggetto a controllo è detto **Configuration Item (CI)**. Una **Configuration** è un insieme di CI in versioni specifiche. Le modifiche sono tracciate attraverso **versioni**, e ogni versione può essere parte di una **baseline**:

- *Development baseline (Linea di sviluppo)*: versione di sviluppo con nuove feature ancora in test;
- *Production baseline* (Linea di produzione): versione stabile rilasciata al cliente.

Le versioni seguono la notazione `MAJOR.MINOR.PATCH`, dove:

- `MAJOR (x.0.0)`: cambiamenti incompatibili con versioni precedenti (es. API modificate).
- `MINOR (x.y.0)`: aggiunta di nuove funzionalità compatibili.
- `PATCH (x.y.z)`: bugfix e modifiche minori.

I repository (archivi) permettono la gestione di tutte le versioni di un progetto. Si distinguono due principali modelli:

- **Lock-Modify-Unlock**: accesso esclusivo al file; elimina i conflitti ma limita lo sviluppo parallelo.
- **Copy-Modify-Merge**: approccio moderno che permette a più sviluppatori di lavorare in parallelo, gestendo i conflitti nel merge.

Git utilizza il meccanismo di *Copy-Modify-Merge*, in cui :

1. Ogni sviluppatore clona il repository remoto localmente (creando una **working copy**);
2. Le modifiche vengono tracciate in locale e committate nella cronologia personale;
3. Solo quando pronte vengono *pushate* nel repository remoto.

Per lavorare senza interferenze si usano i branch, ovvero linee parallele di sviluppo. I branch si possono:

- Creare per ogni nuova funzionalità o bugfix;
- Testare e validare indipendentemente;
- Unire nel branch principale (detto main o master) tramite *merge*.

Durante un merge, Git cerca di allineare i cambiamenti fatti su file modificati contemporaneamente da più persone. In caso di conflitti, è necessario risolverli manualmente.

GitFlow: organizzazione del lavoro in team

GitFlow è un modello di branching usato per gestire lo sviluppo collaborativo. Prevede:

- Un branch principale (*main*) sempre stabile e rilasciabile;
- Un branch develop dove convergono le funzionalità in fase di sviluppo;
- Branch feature/ per nuove funzionalità;
- Branch bugfix/ o hotfix/ per correzioni urgenti;
- Branch release/ per preparare il rilascio.

Una volta completato un branch effimero (feature o fix), questo viene integrato nel develop o nel main tramite una merge request. Ogni merge è preceduto da una code review, durante la quale:

- Il codice viene analizzato da revisori esterni al suo autore;
- Si controllano aderenza agli standard, presenza di bug, qualità del design;
- Viene richiesta una checklist di test per verificare la correttezza.

La collaborazione è facilitata dalle issue, che descrivono una feature o un bug. Ogni issue può essere associata a un branch dedicato, la cui conclusione chiude automaticamente la issue associata.

Buone pratiche di commit e build management

Una *build* è il processo automatico che compila il codice, esegue i test, prepara i pacchetti e ne effettua il deploy. Un sistema di build (es. GitHub Actions, Jenkins) permette di verificare subito l'integrità del progetto.

Regole fondamentali:

- Committare spesso e con modifiche piccole;
- Non committare codice rotto o non testato;
- Non committare durante una build fallita;
- Usare messaggi di commit significativi.

Stati dei file in Git

Ogni file può trovarsi in diversi stati:

- **Untracked:** non è ancora sotto versionamento;
- **Modified:** è stato modificato rispetto all'ultima versione;
- **Staged:** è pronto per essere committato;
- **Committed:** è stato salvato nella cronologia locale;
- **Pushed:** è stato inviato al repository remoto.

Nel terminale o in Visual Studio Code, è possibile gestire questi stati con i comandi:

```
git clone <url>            # Clona il repository
git status                 # Mostra lo stato dei file
git add <file>             # Passa il file da Modified a Staged
git commit -m "Messaggio"  # Esegue il commit
git push                   # Invia il commit al remoto
git pull                   # Scarica aggiornamenti dal remoto
```

I colori in VSC indicano:

- Giallo (M): modificato;
- Verde (U): committato di recente;
- Bianco: allineato col repository remoto.

Ruoli in un progetto GitLab

Nel contesto GitLab, i ruoli si distinguono in:

- **Owner:** può committare direttamente sul main (da evitare);
- **Maintainer:** esegue merge e mantiene la qualità del progetto;
- **Developer:** lavora su branch separati e propone merge;
- **Guest:** ha solo permessi di lettura.

Ogni merge viene discusso collegialmente tramite Merge Request, e può restare in *bozza* finché non è pronta per l'integrazione.

Il rispetto dei ruoli, dei flussi di lavoro e delle revisioni è fondamentale per mantenere un codice condiviso di alta qualità.

Java Basics

Il paradigma della programmazione orientata agli oggetti (OOP) rappresenta un cambiamento radicale rispetto alla programmazione procedurale, e in Java trova una delle sue implementazioni più solide e diffuse. Java consente la creazione di software robusti, modulari e manutenibili attraverso l'uso di classi, oggetti, metodi e concetti come incapsulamento, ereditarietà e polimorfismo. Questa sezione copre tutti gli strumenti essenziali per creare e gestire codice orientato agli oggetti in modo corretto, sicuro e riutilizzabile.

Classi

Una classe è una descrizione astratta di un oggetto, comprendente attributi (dati) e metodi (funzioni). Quando una classe viene istanziata, si ottiene un oggetto, ovvero un'entità autonoma con uno stato e un comportamento. Ogni oggetto appartiene a una specifica classe e viene creato tramite la parola chiave `new`.

```
class Persona {
    private String nome;
    private int eta;

    public Persona(String nome, int eta) {
        this.nome = nome;
        this.eta = eta;
    }

    public String getNome() {
        return nome;
    }

    public void setEta(int eta) {
        this.eta = eta;
    }
}

Persona p = new Persona("Luca", 30);
```

L'esempio mostra l'uso del costruttore, un metodo speciale senza valore di ritorno, che inizializza lo stato di un oggetto alla sua creazione. I metodi `get` e `set` sono detti rispettivamente getter (per ottenere valori) e setter (per modificarli).

L'accesso agli attributi è regolato tramite modificatori di visibilità:

- `private`: accessibile solo all'interno della stessa classe;
- `public`: accessibile ovunque nel progetto.

Il riferimento `this` è usato per accedere all'istanza corrente dell'oggetto, utile in caso di ambiguità tra nomi di parametri e attributi.

Gli attributi statici sono condivisi tra tutte le istanze della stessa classe. Non appartengono a un singolo oggetto, ma alla classe stessa. I metodi statici vengono utilizzati per operazioni generiche, come nelle classi `Math`, `System`, `Arrays`, ecc.

```
public class Util {
    public static int somma(int a, int b) {
        return a + b;
    }
}

int risultato = Util.somma(3, 4);
```

In Java è possibile creare classi all'interno di altre classi, ovvero le classi annidate, che possono essere:

- Statiche: non hanno accesso agli attributi della classe contenitrice;
- Non statiche: possono accedere direttamente agli attributi e metodi dell'oggetto contenitore.

Esempio di classe interna non statica:

```
class Esterna {
    private int dato = 10;

    class Interna {
        void stampa() {
            System.out.println("Dato: " + dato);
        }
    }
}

Esterna e = new Esterna();
Esterna.Interna i = e.new Interna();
i.stampa();
```

Le classi interne possono anche essere:

- Locali: dichiarate all'interno di un metodo;
- Anonime: senza nome, utilizzate per estendere o implementare una classe/interfaccia al volo.

Package e visibilità

Un package è un insieme logico di classi raggruppate in una cartella. La convenzione è usare nomi puntati, come `it.polito.esempio`. Le classi pubbliche possono essere usate anche al di fuori del package, mentre quelle private o con visibilità predefinita (`package-private`) sono visibili solo all'interno del package stesso.

```
package it.polito.esempio;

public class Saluto {
    public void ciao() {
        System.out.println("Ciao!");
    }
}
```

Wrapper class e autoboxing

Java possiede classi *wrapper* per ogni tipo primitivo, necessarie quando si vogliono utilizzare questi valori in collezioni o con metodi che richiedono oggetti. Alcuni esempi sono:

- `int` → `Integer`
- `char` → `Character`
- `double` → `Double`

Con l'*autoboxing*, Java converte automaticamente tipi primitivi in oggetti wrapper e viceversa.

```
Integer a = 5;   // autoboxing
int b = a;       // unboxing
```

I tipi primitivi sono più efficienti, ma meno flessibili. I wrapper offrono metodi utili per la manipolazione dei dati.

Array

Gli array sono strutture dati che contengono una sequenza ordinata di elementi dello stesso tipo. In Java, sono oggetti che si trovano nell'heap e hanno dimensione fissa, definita al momento della creazione.

```
int[] numeri = new int[5]; // Inizialmente tutti i valori sono null
numeri[0] = 10;
```

Gli array multidimensionali sono dichiarati come array di array, permettendo anche righe di lunghezza variabile:

```
int[][] matrice = new int[3][];
matrice[0] = new int[2];
matrice[1] = new int[4];
```

La lunghezza di un array si ottiene con `nomeArray.length` (senza parentesi, essendo un campo, non un metodo).

Stringhe e caratteri

Le stringhe sono oggetti immutabili rappresentati dalla classe `String`. Una volta creata, una stringa non può essere modificata. Ogni operazione su di essa genera un nuovo oggetto. Esistono anche `StringBuilder` e `StringBuffer`, che sono versioni modificabili delle stringhe:

- `StringBuilder`: più veloce ma non *thread-safe*;
- `StringBuffer`: più sicuro per l'uso concorrente.

```
String nome = "Luca";
String saluto = "Ciao, " + nome + "!";

StringBuilder sb = new StringBuilder();
sb.append("Ciao").append(" Luca!");
System.out.println(sb.toString());
```

> ⚠ *ATTENZIONE! obj1 == obj2 verifica se due riferimenti puntano allo stesso oggetto in memoria, non fa l'uguaglianza logica, per cui si deve usare obj1.equals(obj2)!*

La codifica dei caratteri in Java segue lo standard Unicode, solitamente UTF-8 o UTF-16.

UML (Unified Modeling Language)

L'UML è un linguaggio di modellazione che aiuta a rappresentare graficamente classi, relazioni, azioni e ruoli nel sistema software. I principali diagrammi utilizzati sono:

- Class diagram: rappresenta classi, attributi, metodi e relazioni (ereditarietà, associazione, composizione);
- Activity diagram: visualizza il flusso delle operazioni;
- Use Case diagram: mostra le interazioni tra utenti e sistema.

Un buon flusso di lavoro prevede la creazione dello schema UML prima dell'implementazione. Si consiglia l'uso di strumenti come draw.io per la creazione.

La cardinalità tra classi segue la stessa notazione usata negli schemi ER (Entity-Relationship).

Garbage Collector

In Java, la gestione della memoria è automatica. Gli oggetti che non sono più referenziati vengono rimossi dalla **Java Virtual Machine (JVM)** tramite il **Garbage Collector (GC)** che agisce in background per ripulire la memoria. Il programmatore può suggerire l'esecuzione del GC tramite `System.gc()`, ma non può forzarne l'attivazione. Se definito, il metodo `finalize()` viene eseguito prima della distruzione dell'oggetto.

La memoria dinamica viene allocata nell'**heap**, dove risiedono tutti gli oggetti creati a runtime.

Conclusione

La presenza della JVM garantisce una delle due principali caratteristiche di Java, ovvero la **portabilità**, in quanto permette di riprodurre Java su qualsiasi piattaforma. La seconda caratteristica è, invece, la **robustezza**, garantita da *severi controlli sui tipi* dei dati e sulla loro *dimensione* (modificabile, però, dinamicamente), sulla *velocità* affidata all'interprete e sul GC.

Java è sia un linguaggio che una piattaforma che offre una VM (Virtual Machine), delle API (Application Prorgamming Interface) e l'SDK (Software Development Kit). Il suo caricamento è basato sul **classpath**, un parametro nella JVM o nel compilatore che specifica la posizione delle classi e dei package definiti dall'utente. Il parametro può essere impostato dalla riga di comando o tramite una variabile d'ambiente.

Ereditarietà

L'ereditarietà è uno dei concetti fondanti della programmazione a oggetti. In Java, una classe può estendere un'altra, ereditando attributi e metodi già definiti. Questo meccanismo permette di modellare relazioni tra entità in modo gerarchico e rappresentare comportamenti condivisi in un'unica classe madre, evitando duplicazioni e facilitando il riuso del codice.

La sintassi per dichiarare una relazione di ereditarietà è semplice:

```
class Veicolo {
    protected String marca;

    public void avvia() {
        System.out.println("Veicolo in movimento");
    }
}

class Automobile extends Veicolo {
    public void avvia() {
        System.out.println("Automobile in movimento");
    }
}
```

In questo esempio, `Automobile` eredita il campo `marca` e il metodo `avvia()` dalla classe `Veicolo`. Tuttavia, può ridefinire (`override`) il metodo `avvia()` per modificarne il comportamento. La parola chiave `@Override` è opzionale ma fortemente consigliata, perché forza il compilatore a verificare che il metodo stia effettivamente sovrascrivendo un metodo della superclasse:

```
@Override
public void avvia() {
    System.out.println("Automobile in movimento");
}
```

Costruttori

I costruttori non vengono ereditati in Java. Tuttavia, una sottoclasse può invocare il costruttore della superclasse attraverso il costrutto `super(...)`, che deve essere la prima istruzione nel costruttore della sottoclasse:

```
class Persona {
    private String nome;

    public Persona(String nome) {
        this.nome = nome;
    }
}

class Studente extends Persona {
    private String matricola;

    public Studente(String nome, String matricola) {
        super(nome);
        this.matricola = matricola;
    }
}
```

In questo esempio, il costruttore di `Studente` richiama `super(nome)` per inizializzare la parte `Persona` dell'oggetto.

Java mette a disposizione il modificatore di accesso `protected`, che consente l'accesso a campi e metodi dalla sottoclasse anche se si trovano in un altro package. È particolarmente utile quando si vuole progettare una classe base riutilizzabile ma che mantenga un certo livello di incapsulamento.

Polimorfismo

La vera potenza dell'ereditarietà si manifesta quando una variabile è dichiarata come tipo della superclasse, ma a runtime fa riferimento a un oggetto di una sottoclasse. Questo permette di scrivere codice più flessibile e generale:

```
Veicolo v = new Automobile();
v.avvia(); // Output: Automobile in movimento
```

Questo comportamento è detto polimorfismo e viene gestito dinamicamente grazie al dynamic dispatch. È importante notare che, se si accede a un metodo non ridefinito, verrà comunque usato quello definito nella superclasse.

Talvolta è necessario verificare il tipo effettivo di un oggetto e fare il downcasting:

```
if (v instanceof Automobile) {
    Automobile a = (Automobile) v;
    a.avvia();
}
```

È buona pratica evitare il più possibile il casting e affidarsi al polimorfismo.

> *Curiosità: `Object.toString()` stampa `NomeClasse@{codice univoco}` a meno che l'oggetto non sia dichiarato come stringa, caso in cui stampa l'effettiva stringa.*

Classi astratte e interfacce

Per rappresentare concetti generici o incompleti, Java fornisce le classi astratte e le interfacce. Una classe astratta può contenere sia metodi implementati sia non implementati (astratti):

```
abstract class Forma {
    abstract double area();

    public void stampaTipo() {
        System.out.println("È una forma");
    }
}
```

Le classi astratte non possono essere istanziate direttamente, ma servono come base per altre classi (`Forma f = new Cerchio(...)` funziona).

Le interfacce, invece, definiscono esclusivamente metodi astratti (fino a Java 7) o anche metodi default e statici (da Java 8 in poi). Una classe può implementare più interfacce, superando così il limite dell'ereditarietà singola:

```
interface Stampabile {
    void stampa();
}

class Documento implements Stampabile {
    public void stampa() {
        System.out.println("Documento stampato");
    }
}
```

Questa caratteristica supporta il concetto di programmazione ad interfaccia, in cui si programmano le funzionalità desiderate, senza vincolarsi a implementazioni specifiche.

GENERICS

I Generics rappresentano una delle funzionalità più importanti e sofisticate introdotte in Java a partire dalla versione 5. Il loro scopo principale è quello di permettere la scrittura di codice riutilizzabile e sicuro, riducendo la necessità di eseguire conversioni esplicite di tipo (i cosiddetti cast) e migliorando la leggibilità e la manutenibilità del software.

La maggior parte delle interfacce nella libreria standard di Java, come `List`, `Map`, `Set`, sono state riscritte per essere generiche. Questo ha permesso di ottenere maggiore sicurezza e flessibilità.

```
List<String> nomi = new ArrayList<>();
nomi.add("Mario");
String primo = nomi.get(0); // Nessun cast
```

Uso dei Generics

In molte situazioni è necessario eseguire le stesse operazioni su oggetti appartenenti a classi diverse. Prima dell'introduzione dei generics, una soluzione comune consisteva nell'utilizzare il tipo `Object`, che essendo superclasse di tutte le classi, permette di contenere qualsiasi valore. Tuttavia, questo approccio impone l'uso di cast espliciti per accedere nuovamente al tipo originale. Questi cast possono fallire a runtime, portando a errori come `ClassCastException`, e non possono essere verificati in fase di compilazione.

Per esempio, una classe `Pair` che contiene due oggetti di tipo generico, prima dei generics si scriveva così:

```
public class Pair {
  Object a, b;
  public Pair(Object a, Object b) {
    this.a = a; this.b = b;
  }
  Object first() { return a; }
  Object second() { return b; }
}
```

L'utilizzo di questa classe richiedeva conversioni esplicite, come nel caso seguente:

```
Pair p = new Pair("Uno", "Due");
String val = (String) p.second();
```

Questo tipo di codice è fragile, perché il cast è valido solo se il contenuto è effettivamente del tipo atteso. In caso contrario, l'errore avviene solo a runtime, cioè durante l'esecuzione del programma.

Definizione

I generics permettono di definire classi, interfacce e metodi che operano su tipi parametrizzati. In altre parole, si definisce un tipo astratto che verrà concretizzato al momento dell'utilizzo. Ecco come appare la stessa classe `Pair` riscritta usando i generics:

```
public class Pair<T> {
  private T a, b;

  public Pair(T a, T b) {
    this.a = a; this.b = b;
  }

  public T first() { return a; }
  public T second() { return b; }
  public void set1st(T x) { a = x; }
  public void set2nd(T x) { b = x; }
}
```

Nel codice sopra, `T` è un parametro di tipo che rappresenta un qualunque tipo concreto specificato in fase di utilizzo. Ad esempio:

```
Pair<String> p = new Pair<>("Uno", "Due");
String val = p.second(); // Nessun cast necessario
```

Il vantaggio principale è che l'uso corretto dei tipi è verificato in fase di compilazione, evitando errori di tipo a runtime e rendendo il codice più sicuro e leggibile.

Java è in grado di dedurre automaticamente il tipo di un oggetto generico in molti contesti. Dal Java 7, è stato introdotto il diamond operator `<>` che permette di omettere i parametri di tipo quando possono essere dedotti:

```
List<String> lista = new ArrayList<>();
```

Parametrizzazione

I parametri di tipo seguono convenzioni comuni nella comunità Java. Ad esempio, `T` è spesso usato per rappresentare un tipo generico, `E` per indicare un elemento in una collezione, `K` e `V` per chiave e valore nelle mappe, e così via.

La sintassi per definire una classe generica è:

```
class NomeClasse<T> { ... }
```

Analogamente, le interfacce possono essere generiche:

```
interface Interfaccia<T> { ... }
```

Type erasure

I generics in Java non esistono nel bytecode finale. Questo significa che il compilatore, dopo aver verificato i tipi, cancella le informazioni di tipo. Questo meccanismo si chiama type erasure. Di conseguenza:

- Non è possibile usare `instanceof` su tipi parametrizzati;
- Non è possibile creare array di tipi generici;
- Non si possono ottenere informazioni di tipo a runtime.

Esempio:

```
Pair<Integer> p = new Pair<>(1, 2);
boolean ok = p instanceof Pair; // valido
boolean err = p instanceof Pair<Integer>; // errore
```

Metodi

Anche i metodi possono essere generici. Un metodo generico definisce uno o più parametri di tipo che vengono usati solo al suo interno:

```
public static <T> boolean contiene(T[] array, T elemento) {
  for (T x : array) {
    if (x.equals(elemento)) return true;
  }
  return false;
}
```

Questo approccio permette di scrivere metodi riutilizzabili per array di qualunque tipo.

Tipi vincolati (bounded types)

Per limitare i tipi accettabili da un parametro generico, si possono specificare vincoli. Ad esempio, per richiedere che il tipo `T` sia confrontabile, si scrive:

```
<T extends Comparable<T>>
```

Questo assicura che ogni `T` abbia un metodo `compareTo(T)` che può essere usato per ordinare o confrontare elementi.

Comparator

I `Comparator` possono essere **combinati** tra loro. Ad esempio, è possibile ordinare per cognome e poi per nome:

```
Arrays.sort(studenti,
  Comparator.comparing(Student::getLastName)
    .thenComparing(Student::getFirstName));
```

È anche possibile invertire l'ordine con `reversed()`.

Dal punto di vista delle prestazioni, l'uso di espressioni lambda e reference method può essere leggermente meno efficiente rispetto all'uso diretto di `Comparator` scritti manualmente, ma i vantaggi in termini di leggibilità e concisione sono notevoli.

Wildcard e varianti

Una delle limitazioni principali dei generics in Java è che sono **invarianti**: `List<String>` non è sottotipo di `List<Object>`, anche se `String` è sottotipo di `Object`. Questo per evitare errori di tipo.

Per gestire questa rigidità, si usano le **wildcard** (*simbolo speciale che può essere utilizzato al posto di uno o più caratteri in una stringa di testo durante ricerche o comandi*). La wildcard `?` rappresenta un tipo sconosciuto. Si può anche specificare un vincolo:

- `? extends T` accetta solo sottotipi di `T`
- `? super T` accetta solo supertipi di `T`

Ad esempio, per accettare una `List` di qualunque sottotipo di `Number`:

```
public void stampa(List<? extends Number> lista) { ... }
```

Integrazione con le interfacce funzionali

I generics si integrano perfettamente con le interfacce funzionali, cioè interfacce che definiscono un solo metodo astratto. Queste sono ampiamente utilizzate in combinazione con le espressioni lambda, in quanto queste creano un'interfaccia funzionale.

Esempi di interfacce funzionali generiche sono:

- `Function<T, R>`: funzione che prende `T` e restituisce `R`
- `Predicate<T>`: funzione booleana su `T`
- `Comparator<T>`: funzione di confronto tra due oggetti `T`

```
Function<String, Integer> lunghezza = String::length;
Predicate<String> nonVuota = s -> !s.isEmpty();
Comparator<Student> comp = Comparator.comparing(Student::getId);
```

Conclusione

L'utilizzo dei generics in Java rappresenta un passo fondamentale verso la scrittura di codice riutilizzabile, leggibile e sicuro. Capire come funzionano i parametri di tipo, le wildcard, i vincoli e la cancellazione dei tipi consente di sfruttare al massimo la potenza del linguaggio, riducendo i bug e aumentando la chiarezza del software. La loro integrazione con le interfacce funzionali e le nuove API del linguaggio ne fa uno strumento imprescindibile per lo sviluppo moderno in Java.

Java Collections Framework

Il Java Collections Framework (JCF) è il cuore delle strutture dati in Java. Fornisce tre pilastri fondamentali:

1. Interfacce (Abstract Data Types – ADT): definiscono il *contratto* logico di una struttura, indipendentemente da come sia implementata.
2. Implementazioni: classi concrete che realizzano quel contratto scegliendo un particolare algoritmo o struttura (array ridimensionabile, lista collegata, hash table, albero bilanciato…).
3. Algoritmi: metodi statici riutilizzabili (ordinamento, ricerca, mescolamento…) che operano sulle collezioni.

Tutti gli elementi del framework si trovano nel package `java.util` e, dalla Java 5, sono stati definiti usando i generic types (per esempio `List<E>`), eliminando la necessità di cast espliciti e riducendo gli errori a run-time.

Interfacce generiche

Interfaccia	Ruolo	Caratteristiche salienti
`Collection<E>`	Contenitore generico di oggetti	È il super-tipo di quasi tutte le collezioni; non impone ordinamento né unicità.
`List<E>`	Sequenza indicizzata	Permette duplicati, mantiene l'ordine d'inserimento e consente accesso casuale tramite indice.
`Set<E>`	Insieme matematico	Vietati i duplicati (`e1.equals(e2)` non può essere `true` per due elementi distinti).
`Queue<E>`	Coda logica	Gestisce gli elementi in ordine FIFO o secondo priorità.
`Map<K,V>`	Dizionario (associazione chiave-valore)	Chiavi uniche con accesso a tempo costante/logaritmico.
`SortedSet<E>` / `SortedMap<K,V>`	Variante ordinata di `Set` / `Map`	Gli elementi (o le chiavi) vengono iterati nel *natural ordering* o in un ordine stabilito da un `Comparator`.
`Iterable<E>`	Fonte di iterazione	Fornisce `iterator()` e abilita il *for-each*.

> *Natural ordering: l'ordinamento naturale di una classe che implementa l'interfaccia `Comparable`. Se un tipo non è comparabile, va fornito un `Comparator`.*

Implementazioni e strutture interne

Famiglia	Classe principale	Struttura dati interna
Liste	`ArrayList`	Array ridimensionabile
	`LinkedList`	Lista doppiamente collegata
Set	`HashSet`	Hash table
	`LinkedHashSet`	Hash table + lista collegata
	`TreeSet`	Red-Black Tree bilanciato
Map	`HashMap`	Hash table
	`LinkedHashMap`	Hash table + lista collegata
	`TreeMap`	Red-Black Tree
Code	`LinkedList`	Lista collegata
	`PriorityQueue`	Heap binario

L'interfaccia `Collection<E>`

Una collection è un gruppo di riferimenti ad oggetti. Metodi chiave:

```
int size();                 // numero di elementi
boolean isEmpty();           // collezione vuota?
boolean contains(Object o);
boolean add(E e);
boolean remove(Object o);
void clear();
Iterator<E> iterator();     // per iterare
Object[] toArray();         // copia in Object[] (type erasure)
<T> T[] toArray(T[] a);     // restituisce array tipizzato
```

Esempio pratico

```
Collection<Person> persons = new LinkedList<>();
persons.add(new Person("Alice"));
System.out.println(persons.size());

Collection<Person> copy = new TreeSet<>();
copy.addAll(persons);       // oppure new TreeSet<>(persons)
Person[] array = copy.toArray(new Person[0]);
System.out.println(array[0]);
```

> *Nota: il costruttore "copia" (`C(Collection c)`) deve essere sempre presente nelle implementazioni di `Collection`.*

Algoritmi della classe `Collections`

La utility `java.util.Collections` offre algoritmi statici ottimizzati:

- `sort(List<T>)` / `sort(List<T>, Comparator<? super T>)` (merge sort stabile).
- `binarySearch(List<? extends T>, T key)` – richiede lista ordinata.
- `shuffle(List<?>)` – mescola con algoritmo di Fisher-Yates.
- `reverse(List<?>)` – inverte l'ordine.
- `rotate(List<?>, int distance)` – ruota di *distance* posizioni.
- `min(Collection<?>)` / `max(Collection<?>)` – estremo secondo ordine naturale o `Comparator`.

Nota su generics avanzati

La firma `sort(List<T> list, Comparator<? super T> c)` usa *wildcard* `? super T` perché un `Comparator<Student>` è perfettamente valido per confrontare oggetti di una sottoclasse, ad esempio `MasterStudent`.

`List<E>` – Sequenze indicizzate

Le liste accettano duplicati, preservano l'ordine d'inserimento e consentono accesso per posizione. Metodi aggiuntivi rispetto a `Collection`:

```
E get(int index);
E set(int index, E element);
void add(int index, E element);
E remove(int index);
int indexOf(Object o);
int lastIndexOf(Object o);
List<E> subList(int from, int to);
```

Implementazioni e complessità

- **`ArrayList`**: ottimo per letture random (`get(i)` costante) e aggiunte in coda. Le inserzioni/cancellazioni in mezzo richiedono lo shift degli elementi.
- **`LinkedList`**: ottimo per inserire o togliere in testa/coda; l'accesso per indice deve scorrere la lista.

Esempio di uso di `ArrayList`

```
List<Integer> l = new ArrayList<>();
l.add(42);      // [42]
l.add(0, 13);   // [13, 42]
l.set(0, 20);   // [20, 42]
int a = l.get(1);  // 42
l.add(9, 30);      // IndexOutOfBoundsException
```

> *Best practice: dichiarare variabili con il tipo di interfaccia più generale (`List` invece di `ArrayList`) per facilitare futuri cambi d'implementazione.*

`Queue<E>` – Code e priority queue

Una queue è una collezione in cui l'accesso principale avviene alla *testa* (head):

- `peek()` → legge senza rimuovere
- `poll()` → legge e rimuove

Implementazioni tipiche:

- **`LinkedList`** (modalità FIFO classica).
- **`PriorityQueue`**: usa un *heap* e restituisce sempre l'elemento "più piccolo" (o "migliore" secondo il comparatore).

Esempio

```
Queue<Integer> fifo = new LinkedList<>();
Queue<Integer> pq   = new PriorityQueue<>();

fifo.add(3); pq.add(3);
fifo.add(1); pq.add(1);
fifo.add(2); pq.add(2);

System.out.println(fifo.peek()); // 3 (FIFO)
System.out.println(pq.peek());   // 1 (minimo)
```

`Set<E>` e `SortedSet<E>` – Insiemi

- **`Set`** proibisce i duplicati: l'inserimento di un elemento già presente non modifica la collezione.
- **`SortedSet`** estende `Set` ordinando gli elementi:
  - `first()`, `last()` → estremo inferiore/superiore.
  - `headSet(to)`, `tailSet(from)`, `subSet(from,to)` per viste parziali ordinate.

Implementazioni:

Classe	Hash / Tree	Ordine di iterazione
`HashSet`	Hash table	Non deterministico
`LinkedHashSet`	Hash + lista	Ordine d'inserimento
`TreeSet`	Red-Black Tree	Ordinato (naturale o `Comparator`)

`Map<K,V>` e `SortedMap<K,V>` – Dizionari

Una mappa associa valori (`V`) a chiavi (`K`) uniche. Metodi fondamentali:

```
V put(K key, V value);
V get(Object key);
V remove(Object key);
boolean containsKey(Object key);
boolean containsValue(Object value);
Set<K> keySet();            // vista delle chiavi
Collection<V> values();     // vista dei valori
```

Uso basilare

```
Map<String,Person> people = new HashMap<>();
people.put("ALCSMT", new Person("Alice Smith"));
people.put("RBTGRN", new Person("Robert Green"));

if (!people.containsKey("RBTGRN"))
    System.out.println("Not found");

Person bob = people.get("RBTGRN");
int populationSize = people.size();
```

Per iterare:

```
for (Person p : people.values())      // tutti i valori
    System.out.println(p);

for (String ssn : people.keySet())    // tutte le chiavi
    System.out.println(ssn);
```

Aggiornare frequenze con le API moderne

```
Map<String,Integer> wc = new HashMap<>();
for (String w : words) {
    wc.compute(w, (k,v) -> v == null ? 1 : v + 1);
}

// variante più efficiente con contenitore mutabile
Map<String,Counter> wc2 = new HashMap<>();
for (String w : words) {
    wc2.computeIfAbsent(w, k -> new Counter()).i++;
}
```

```
class Counter { int i = 0; public String toString() { return ":" + i; } }
```

Implementazioni di `Map`

Classe	Struttura	Ordine
`HashMap`	Hash table	Nessuno
`LinkedHashMap`	Hash + lista	Inserimento
`TreeMap`	Red-Black Tree	Ordinato

> *Load factor di `HashMap` (default 0.75) controlla quando raddoppiare il numero di bucket per mantenere l'efficienza.*

Iterazione sicura con `Iterator` e `Iterable`

L'interfaccia `Iterable` dichiara `iterator()`, abilitando tre modi equivalenti di scorrere:

1. Uso esplicito dell'`Iterator`

```
for (Iterator<Person> it = persons.iterator(); it.hasNext(); ) {
    Person p = it.next();
    System.out.println(p);
}
```

2. Enhanced for-each

```
for (Person p : persons)
    System.out.println(p);
```

3. `forEach` con lambda

```
persons.forEach(p -> System.out.println(p));
```

ConcurrentModificationException

Modificare una collezione mentre la si sta iterando con mezzi esterni all'iteratore causa l'eccezione `ConcurrentModificationException`.

Esempio ERRATO (rimozione)

```
for (Iterator<?> itr = lst.iterator(); itr.hasNext(); ) {
    itr.next();
    if (count == 1)
        lst.remove(count);  // ❌ genera eccezione
    count++;
}
```

Esempio CORRETTO (rimozione)

```
for (Iterator<?> itr = lst.iterator(); itr.hasNext(); ) {
    itr.next();
    if (count == 1)
        itr.remove();       // ✅
    count++;
}
```

Esempio CORRETTO (aggiunta)

```
for (ListIterator<Integer> itr = lst.listIterator(); itr.hasNext(); ) {
    itr.next();
    if (count == 2)
        itr.add(22);        // permette inserimento
    count++;
}
```

Gestire l'assenza di valori con `Optional<T>`

Storicamente i metodi Java restituivano `null` per indicare "nessun risultato", obbligando il chiamante a fare il controllo e aprendo la porta a NullPointerException (NPE).

Con Java 8 nasce `Optional<T>`, un wrapper semantico che rende esplicita la possibilità di *vuoto*.

```
Optional<String> maybeName = Optional.ofNullable(name);
maybeName.ifPresent(n -> System.out.println(n));   // esegue il blocco solo se
presente

String safe = maybeName.orElse("Sconosciuto");    // valore di fallback
```

Metodi principali:

- `isPresent()` / `isEmpty()`
- `get()` (sconsigliato: lancia eccezione se vuoto)
- `ifPresent(Consumer)`
- `orElse(T)` / `orElseGet(Supplier)` / `orElseThrow()`

Creazione:

- `Optional.of(T v)` → eccezione se `v` è `null`.
- `Optional.ofNullable(T v)` → vuoto se `v` è `null`.
- `Optional.empty()`.

Java Streams

In Java, uno Stream è una sequenza di elementi che proviene da una fonte (come una Collection, un array o un generatore) e che supporta operazioni di elaborazione dati. Gli Stream permettono di esprimere operazioni complesse in modo conciso e funzionale.

Le principali caratteristiche degli Stream sono:

- Pipelining: le operazioni sono collegate tra loro come una catena.
- Internal iteration: è Java a gestire il ciclo di iterazione internamente, senza `for` espliciti.
- Lazy evaluation: nessuna operazione viene realmente eseguita finché non si chiama un'operazione *terminale*.

Iterazione esterna ed interna

- Iterazione esterna: tradizionale ciclo `for` o `for-each`, in cui lo sviluppatore controlla il flusso.

```
for(String word : lyrics){
    System.out.println(word);
}
```

- Iterazione interna: lo Stream si occupa del flusso. Più compatta, meno soggetta a errori e permette ottimizzazioni interne.

```
lyrics.forEach(System.out::println);
```

Interfacce funzionali

Un'interfaccia funzionale è un'interfaccia con un solo metodo astratto. Serve come base per le *lambda expression* e deve:

- Dipendere solo dagli argomenti forniti.
- Non avere effetti collaterali.

Interfacce funzionali comuni (da `java.util.function`):

```
Function<T,R>         // R apply(T t)
BiFunction<T,U,R>     // R apply(T t, U u)
BinaryOperator<T>     // T apply(T t, T u)
UnaryOperator<T>      // T apply(T t)
Predicate<T>          // boolean test(T t)
Consumer<T>           // void accept(T t)
```

```
BiConsumer<T,U>      // void accept(T t, U u)
Supplier<T>          // T get()
```

Esistono versioni specializzate per tipi primitivi: `IntPredicate`, `DoubleSupplier`, ecc.

Creazione degli Stream

Generazione da array:

```
String[] s = {"One", "Two", "Three"};
Arrays.stream(s).forEach(System.out::println);
```

Generazione da valori:

```
Stream.of("One", "Two", "Three").forEach(System.out::println);
```

Generazione da collezioni:

```
Collection<String> coll = Arrays.asList("One", "Two", "Three");
coll.stream().forEach(System.out::println);
```

Generazione da funzioni:

```
Stream.generate(() -> Math.random()*10);
Stream.iterate(0, n -> n + 2);
```

⚠ Questi generano stream infiniti, usare `.limit(n)` per evitarli.

Stream numerici (`IntStream`, `DoubleStream`, `LongStream`):

```
IntStream seq = IntStream.generate(() -> (int)(Math.random()*100));
int max = seq.limit(10).max().getAsInt();
```

Operazioni Intermedie

Queste operazioni non eseguono subito, ma costruiscono il pipeline:

```
Stream<T> filter(Predicate<T> p)
Stream<T> limit(int maxSize)
Stream<T> skip(int n)
Stream<T> sorted()
Stream<T> distinct()
Stream<R> map(Function<T,R> mapper)
```

Esempi:

- Filtro:

```
oopClass.stream().filter(Student::isFemale).forEach(System.out::println);
```

- Ordinamento:

```
oopClass.stream().sorted().forEach(System.out::println);
oopClass.stream().sorted(comparingInt(Student::getId)).forEach(System.out::println);
```

- Map (trasformazione):

```
oopClass.stream().map(Student::getFirst).forEach(System.out::println);
```

- FlatMap (per "appiattire" container annidati, creando un container unico):

```
oopClass.stream()
    .map(Student::enrolledIn)
    .flatMap(Collection::stream)
    .distinct()
    .map(Course::getTitle)
    .forEach(System.out::println);
```

Operazioni Terminali

Chiudono la pipeline e attivano il calcolo:

```
Optional<T> findAny()
Optional<T> findFirst()
Optional<T> min()/max()
long count()
void forEach(Consumer<? super T> action)
```

Predicati:

```
boolean anyMatch(Predicate<T>)
boolean allMatch(Predicate<T>)
boolean noneMatch(Predicate<T>)
```

Riduzione (reduce) e Raccolta (collect)

Reduce

Compatta lo stream in un singolo elemento basandosi sul BinaryOperator passatogli. Il parametro identity è il valore di default che viene restituito in caso il risultato sia `null` (esiste la signature senza identity).

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

Esempio:

```
int m = oopClass.stream()
    .map(Student::getFirst)
    .map(String::length)
    .reduce(0, Math::max);
```

Collect

Modifica lo stream in base al Collector specificato.

```
<R> R collect(Collector<T, A, R> collector)
```

Esempio:

```
List<Integer> list = Stream.of(numbers)
    .collect(LinkedList::new, List::add, List::addAll);
```

Collectors predefiniti

Import da usare:

```
import static java.util.stream.Collectors.*;
```

Esempi:

```
.collect(toList()) // Lo stream risultante diventa una lista
.collect(toSet())  // Lo stream risultante diventa un set
.collect(joining(", ")) // Unisce tutti gli elementi dello stream separati da ','
.collect(groupingBy(String::length)) // Raggruppa le stringhe per lunghezza,
// creando una mappa di tipo Map<Integer, List<String>>
.collect(partitioningBy(s -> s.length() > 5)) // Crea una mappa di questo tipo:
// Map<Boolean, List<String>> dove le stringhe con lunghezza > 5 sono
// associate a true, le altre a false.
.collect(averagingInt(String::length))
```

Composizione di collector

```
.collect(collectingAndThen(groupingBy(...), risultato -> ...))
.collect(groupingBy(..., mapping(...)))
```

Collector personalizzati

Si può definire un collector con:

```
Collector.of(supplier, accumulator, combiner, finisher)
```

Esempio:

```
Collector<String, List<String>, List<String>> toList = Collector.of(
    ArrayList::new,
    List::add,
    (a,b) -> { a.addAll(b); return a; }
);
```

Gestione eccezioni negli Stream

Le lambda non possono lanciare eccezioni *checked* direttamente. Soluzioni comuni:

- Ignorare l'errore:

```
.mapToInt(s -> { try { return convert(s); } catch (NotANumber e) { return 0; }})
```

- Avvolgere in una `RuntimeException`:

```
.mapToInt(s -> { try { return convert(s); } catch (NotANumber e) { throw new
RuntimeException(e); }})
```

- Sneaky Throw:

```
static <E extends Throwable> void sneakyThrow(Exception e) throws E {
    throw (E) e;
}
```

Permette di lanciare eccezioni senza che il compilatore si lamenti (non raccomandato!).

Conclusione

Gli Stream offrono una modalità moderna, compatta e funzionale per lavorare con collezioni di dati. Permettono un'espressività superiore rispetto agli approcci imperativi, facilitano l'ottimizzazione e supportano il parallelismo in modo trasparente.

Eccezioni

La gestione delle eccezioni è una componente fondamentale nella programmazione in Java, perché consente di trattare in modo ordinato e strutturato gli errori che possono verificarsi durante l'esecuzione di un programma. A differenza dei semplici codici di errore, le eccezioni permettono di separare la logica normale del programma dalla gestione degli errori, migliorando così la chiarezza e la robustezza del codice.

In Java, le eccezioni sono oggetti che derivano dalla classe `Throwable`, che si divide in due sottoclassi principali: `Error` (che rappresenta errori gravi e non recuperabili, come `OutOfMemoryError`) ed `Exception` (che rappresenta condizioni recuperabili).

Le eccezioni possono essere verificate (checked) o non verificate (unchecked):

- Le checked exceptions sono sottoclassi di `Exception` e devono essere esplicitamente gestite o dichiarate nei metodi che le possono sollevare con la clausola `throws`.
- Le unchecked exceptions sono sottoclassi di `RuntimeException` e non obbligano a una gestione esplicita.

Il blocco fondamentale per la gestione delle eccezioni in Java è `try-catch-finally`. Il codice che può generare un'eccezione viene racchiuso nel blocco `try`, seguito da uno o più blocchi `catch` che specificano il tipo di eccezione da gestire. Il blocco `finally`, se presente, viene eseguito sempre, indipendentemente dal verificarsi o meno dell'eccezione.

```
try {
    int risultato = 10 / 0; // genera ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Divisione per zero non consentita.");
} finally {
    System.out.println("Blocco finally eseguito.");
}
```

In questo esempio, il programma gestisce in modo controllato un'operazione aritmetica non valida. Il blocco `finally` viene eseguito comunque, ed è comunemente usato per chiudere risorse come file o connessioni di rete.

Dalla versione 7 di Java è possibile gestire più eccezioni in un unico blocco `catch`, riducendo la duplicazione del codice (clausola multi-catch):

```
try {
    metodoA();
    metodoB();
} catch (IOException | SQLException e) {
    e.printStackTrace();
}
```

Propagazione delle eccezioni

Quando un metodo può sollevare un'eccezione checked, deve dichiararlo tramite la parola chiave `throws`. Ciò consente a chi utilizza il metodo di sapere che deve gestire l'eccezione:

```
public void leggiFile(String nomeFile) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(nomeFile));
    String linea = br.readLine();
    br.close();
}
```

Un metodo può anche propagare più eccezioni:

```
public void processa() throws IOException, SQLException {
    // codice che può sollevare più eccezioni
}
```

La propagazione è utile quando la gestione dell'eccezione deve essere delegata a un livello superiore della chiamata.

Eccezioni personalizzate

Java permette di definire eccezioni su misura, per gestire condizioni specifiche dell'applicazione. È sufficiente estendere la classe `Exception` o `RuntimeException`:

```
class InputNonValidoException extends Exception {
    public InputNonValidoException(String messaggio) {
        super(messaggio);
    }
}
```

Questa eccezione può poi essere utilizzata come qualsiasi altra:

```
public void verificaInput(int valore) throws InputNonValidoException {
    if (valore < 0) {
        throw new InputNonValidoException("Il valore non può essere negativo.");
    }
}
```

Conclusione

È importante non abusare delle eccezioni. In particolare:

- Evitare di usare le eccezioni per controllare il flusso del programma.
- Gestire solo le eccezioni che si è in grado di trattare correttamente.
- Scrivere messaggi significativi per facilitare il debug.
- Creare eccezioni personalizzate solo se portano un reale vantaggio in termini di chiarezza e gestione.

La gestione delle eccezioni rende il software più robusto, prevenendo crash inaspettati e permettendo una gestione controllata dei problemi durante l'esecuzione. Nei sistemi reali, è fondamentale per garantire l'affidabilità e la continuità di servizio dell'applicazione.

Input/Output

Il sistema di Input/Output (I/O) in Java è progettato per offrire un accesso efficiente e flessibile a file, dispositivi e flussi di rete. Java distingue chiaramente tra due tipi fondamentali di I/O: stream di byte e stream di caratteri, permettendo così di trattare in modo ottimale sia dati binari che testuali.

Stream binario

Gli `InputStream` e `OutputStream` sono le classi base per la gestione dei flussi di byte. Sono adatti per leggere o scrivere file binari (immagini, file audio, archivi, ecc.). Quando si lavora con dati binari (come oggetti serializzati o immagini), si preferisce usare `FileInputStream` e `FileOutputStream`:

```
FileInputStream input = new FileInputStream("immagine.jpg");
FileOutputStream output = new FileOutputStream("copia.jpg");
int byteLetto;
while ((byteLetto = input.read()) != -1) {
    output.write(byteLetto);
}
input.close();
output.close();
```

Ogni chiamata a `read()` restituisce un byte dal file, oppure `-1` se si è raggiunta la fine. La scrittura avviene con `write(byte)`.

Stream di caratteri

Le classi `Reader` e `Writer` sono progettate per gestire flussi di caratteri e operano utilizzando Unicode, quindi sono l'opzione preferibile per file di testo. Tuttavia queste sono classi astratte, quindi si usano delle loro sottoclassi chiamate `BufferedReader` e `BufferedWriter` per la manipolazione dei dati raccolti/da inviare dal/al file, mentre `FileReader` e `FileWriter` per interagire correttamente con il file.

Lettura

```
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
String linea = reader.readLine();
while (linea != null) {
    System.out.println(linea);
    linea = reader.readLine();
}
reader.close();
```

Il `BufferedReader` migliora le prestazioni leggendo blocchi di dati dalla sorgente sottostante e mantenendoli in un buffer creato all'inizializzazione. Invece di leggere un carattere o una riga alla volta direttamente dal flusso originale, legge blocchi di dati dal flusso sottostante e li memorizza nel buffer. Quando si richiede la lettura di dati (ad esempio, con il metodo `readLine()`, che legge riga per riga), `BufferedReader` controlla se il buffer contiene già i dati richiesti. Se sì, li restituisce immediatamente senza dover accedere al flusso originale. Questa gestione del buffer consente di ridurre il numero di operazioni di input/output (I/O) effettive, poiché il flusso originale viene letto solo quando il buffer è vuoto o quando si richiede una quantità di dati superiore a quella disponibile al suo interno. `FileReader` può essere usato da solo per la lettura ma legge solo singoli caratteri e li restituisce sotto forma di interi, di cui bisogna poi eseguire un cast esplicito (non consigliabile).

Scrittura

```
BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));
// FileWriter(nomeFile, true/false); true:append, false:write (default)
writer.write("Ciao mondo\n");
writer.write("Seconda riga\n");
writer.close();
```

Il `BufferedWriter` crea un buffer interno (di default di 8192 caratteri) che verrà utilizzato per memorizzare temporaneamente i dati da scrivere. Solo quando il buffer è pieno o quando viene chiamato il metodo `flush()` (forza il buffer a svuotarsi) o `close()`, i dati vengono effettivamente scritti sul file di destinazione. Questo processo è gestito internamente dalla classe `BufferedWriter` e non richiede intervento diretto dello sviluppatore. `FileWriter` presenta solo `write()` come metodo per la scrittura, che sovrascrive o appende a seconda del parametro aggiuntivo booleano.

Scanner e PrintWriter

Per interazioni semplici con l'utente e la lettura da file, la classe `Scanner` è molto utilizzata, poiché permette di ottenere facilmente diversi tipi di dati:

```
Scanner sc = new Scanner(System.in);
System.out.print("Inserisci un numero: ");
int numero = sc.nextInt();
System.out.println("Hai inserito: " + numero);
```

`Scanner` supporta anche la lettura da file:

```
Scanner fileScanner = new Scanner(new File("dati.txt"));
while (fileScanner.hasNextLine()) {
    String riga = fileScanner.nextLine();
    System.out.println(riga);
}
fileScanner.close();
```

La scrittura può essere fatta con `PrintWriter`, che fornisce metodi comodi come `print()` e `println()`:

```
PrintWriter pw = new PrintWriter("output.txt");
pw.println("Prima riga di testo");
pw.println("Seconda riga");
pw.close();
```

Serializzazione degli oggetti

Java consente di scrivere oggetti su file tramite la serializzazione. Una classe per essere serializzabile deve implementare l'interfaccia `Serializable`:

```
class Persona implements Serializable {
    private String nome;
    private int eta;
}
```

Per scrivere un oggetto:

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("persona.dat"));
out.writeObject(new Persona("Luca", 30));
out.close();
```

E per leggerlo:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("persona.dat"));
Persona p = (Persona) in.readObject();
in.close();
```

La serializzazione è utile, ad esempio, per salvare lo stato di un'applicazione o per trasmettere oggetti su rete. È bene ricordare che modifiche alla struttura della classe possono invalidare i dati serializzati precedentemente.

Gestione delle risorse: try-with-resources

Dalla versione 7 di Java è stato introdotto il costrutto `try-with-resources`, che consente di gestire automaticamente la chiusura delle risorse (come stream e scanner), migliorando la sicurezza e la leggibilità:

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

In questo modo, non è necessario richiamare manualmente il metodo `close()`, riducendo il rischio di perdite di risorse.

Conclusione

Il sistema I/O in Java è estremamente modulare e potente, progettato per supportare un'ampia varietà di applicazioni: dalla semplice lettura da tastiera fino alla manipolazione efficiente di file binari e testuali. La scelta di cosa usare dipende dalle esigenze specifiche: leggibilità del codice, efficienza, compatibilità con altri sistemi, o necessità di bufferizzazione.

Python & OOP

Python è un linguaggio di programmazione moderno, dinamico e ad alto livello, che supporta nativamente il paradigma orientato agli oggetti. Pur essendo meno rigoroso rispetto a Java in termini di tipizzazione e struttura, Python offre grande flessibilità e una sintassi estremamente leggibile, rendendolo particolarmente popolare in ambiti come l'Intelligenza Artificiale (IA), la Data Science e la Prototipazione Rapida.

Classi e oggetti in Python

La definizione di una classe in Python è molto simile a quella di Java dal punto di vista concettuale, ma presenta una sintassi più essenziale. Ogni metodo all'interno di una classe deve accettare esplicitamente il parametro `self`, che rappresenta l'istanza corrente della classe.

```
class Persona:
    def __init__(self, nome):
        self.nome = nome

    def saluta(self):
        print(f"Ciao, mi chiamo {self.nome}")

p = Persona("Luca")
p.saluta()  # Output: Ciao, mi chiamo Luca
```

Nel costruttore `__init__`, si inizializzano gli attributi dell'oggetto. Python non impone dichiarazioni statiche per tipi di variabili o attributi, sebbene a partire da Python 3.5 sia supportata la tipizzazione opzionale tramite type hinting:

```
def somma(a: int, b: int) -> int:
    return a + b
```

Ereditarietà e overriding

Anche in Python è possibile implementare l'ereditarietà e ridefinire metodi della superclasse. La funzione `super()` consente di invocare metodi o costruttori della classe padre.

```
class Studente(Persona):
    def __init__(self, nome, matricola):
        super().__init__(nome)
        self.matricola = matricola

    def saluta(self):
        print(f"Sono {self.nome} e la mia matricola è {self.matricola}")
```

L'override avviene semplicemente ridefinendo il metodo nella sottoclasse. Non è richiesta una sintassi speciale come `@Override` in Java.

Incapsulamento

Python adotta un approccio più permissivo: non esistono veri modificatori di accesso (`private`, `public`, `protected`) come in Java. Tuttavia, esistono convenzioni:

- Attributi preceduti da `_` sono da considerarsi "protetti" (uso interno).
- Attributi con doppio underscore `__` attivano il Name Mangling per simularne la protezione.

```
class ContoBancario:
    def __init__(self, saldo):
        self.__saldo = saldo  # simulazione di campo privato
```

Proprietà e accessori

Tramite il decoratore `@property` è possibile definire metodi che si comportano come attributi:

```
class Quadrato:
    def __init__(self, lato):
        self._lato = lato

    @property
    def area(self):
        return self._lato ** 2

q = Quadrato(4)
print(q.area)  # Output: 16
```

Il decoratore `@property` consente un incapsulamento elegante, permettendo di esporre attributi calcolati come se fossero normali proprietà.

Polimorfismo

Il polimorfismo in Python è implicito e si basa sul concetto di duck typing: "Se cammina come un'anatra e starnazza come un'anatra, allora è un'anatra". Questo significa che ciò che conta non è il tipo dell'oggetto, ma il fatto che possieda i metodi e le proprietà richieste:

```
def stampa_nome(obj):
    obj.saluta()  # funziona se l'oggetto ha un metodo saluta
```

Questa libertà, se usata con attenzione, rende Python estremamente potente e flessibile, anche se meno sicuro rispetto a linguaggi fortemente tipizzati.

Generici e tipi parametrizzati

Python, grazie al modulo `typing`, supporta la definizione di tipi generici:

```
from typing import List

def stampa_elementi(lista: List[int]):
    for elemento in lista:
        print(elemento)
```

Questo approccio è utile per migliorare la leggibilità e sfruttare strumenti di analisi statica (come `mypy`) per la verifica dei tipi.

Conclusione

L'OOP in Python è meno strutturato ma anche più accessibile. La semplicità sintattica, unita a potenti astrazioni come il duck typing e i decoratori, lo rende un linguaggio molto versatile. Tuttavia, in contesti complessi e su larga scala, è consigliato usare la tipizzazione esplicita per mantenere il codice chiaro, manutenibile e più sicuro.

Persistenza

La persistenza dei dati si riferisce alla loro capacità di sopravvivere alla terminazione di un programma, mantenendo lo stato tra diverse esecuzioni. In Java, questa esigenza è tipicamente soddisfatta tramite l'uso di database relazionali, supportati da tecnologie come JDBC, Hibernate e JPA.

ORM (Object-Relational Mapping)

L'ORM è una tecnica che permette di gestire l'interazione tra oggetti Java e tabelle di un database relazionale, senza la necessità di scrivere codice SQL esplicito. Ogni oggetto viene mappato a una riga di una tabella, e ogni attributo a una colonna.

In Java l'ORM può essere gestito:

- Direttamente, scrivendo codice SQL (più flessibile ma più complesso);
- Con l'uso di JPA (Java Persistence API), standard ufficiale ORM, di cui il framework Hibernate, che automatizza la maggior parte della gestione, è una delle implementazioni più note.

Framework e database

Il database scelto per lo studio è H2, un database relazionale scritto in Java, leggero e integrabile facilmente nei progetti per test o applicazioni semplici.

In ambito ORM si utilizzano:

- Classi Java annotate con metadati JPA per definire il comportamento persistente;
- Il file `persistence.xml` per specificare unità di persistenza, configurazioni e connessioni.

Esempio base di entità JPA:

```
@Entity
public class Studente {
    @Id
    private Long matricola;
    private String nome;

    // getter e setter
}
```

Strategie di mapping

Le classi e le loro relazioni possono essere mappate in vari modi:

- Joined Table: ogni classe ha una sua tabella, e le sottoclassi sono collegate tramite foreign key. È la più usata per la chiarezza e normalizzazione dei dati;
- Single Table: tutte le classi sono salvate in un'unica tabella con una colonna discriminante;
- Table per Class: ogni classe ha la propria tabella, non relazionata con la superclass.

In ogni caso, è necessario dichiarare un @Id per ogni entità, che agisce da chiave primaria, ed è responsabilità dello sviluppatore assegnare il valore correttamente.

Strategie di caricamento: EAGER vs LAZY

L'accesso agli oggetti collegati può essere:

- EAGER: tutti i dati vengono caricati subito all'avvio dell'entità. Ciò garantisce velocità di accesso, ma richiede più memoria e tempo all'avvio;
- LAZY: i dati vengono caricati solo quando servono. Ottimo per ottimizzare risorse, ma introduce complessità nella gestione delle sessioni.

Esempio:

```
@OneToMany(fetch = FetchType.LAZY)
private List<Esame> esami;
```

Sessioni e transazioni

L'interazione con il database avviene attraverso:

- Una EntityManagerFactory che crea EntityManager;
- Ogni EntityManager gestisce un contesto di persistenza;
- Le modifiche sono persistite tramite transazioni, che devono essere iniziate e concluse (commit/rollback).

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("nomeUnit");
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();
// operazioni
em.getTransaction().commit();

em.close();
emf.close();
```

Query e JPQL

Per interrogare il database si usa JPQL (Java Persistence Query Language), un linguaggio orientato agli oggetti simile all'SQL, ma che lavora con nomi di classi e attributi.

Esempio di query:

```
TypedQuery<Studente> q = em.createQuery(
    "SELECT s FROM Studente s WHERE s.nome = :nome", Studente.class);
q.setParameter("nome", "Luca");
List<Studente> risultati = q.getResultList();
```

JPQL consente operazioni complesse come join, filter, group by e ordinamenti, mantenendo un alto livello di astrazione e portabilità del codice.

Hibernate o SQL diretto?

Conviene usare JPA/Hibernate quando:

- L'applicazione non ha necessità di prestazioni estreme;
- Serve una gestione automatica delle relazioni;
- Si desidera portabilità tra database diversi.

In caso di requisiti di performance o di operazioni molto specifiche, può essere preferibile usare SQL diretto, scrivendo query manuali.

Conclusione

La persistenza è un aspetto fondamentale nella realizzazione di applicazioni Java reali. Comprendere l'uso corretto di ORM, Entity, transazioni e query è essenziale per garantire un software robusto, scalabile e facilmente manutenibile. La scelta tra strategie eager/lazy e tra ORM o SQL diretto deve sempre basarsi sul contesto applicativo e sui requisiti specifici del progetto.

Software Engineering

La software engineering (ingegneria del software) è una disciplina che si occupa della progettazione, sviluppo e manutenzione di sistemi software complessi. Rispetto alla semplice programmazione, essa include pratiche, metodi e strumenti per coordinare il lavoro di più sviluppatori, garantendo la qualità, l'efficienza e la sostenibilità del prodotto software.

La figura simbolica dell'ingegneria del software è Margaret Hamilton, ingegnera che ha sviluppato il codice per le missioni Apollo e ha coniato per prima il concetto di software engineering, evidenziando la necessità di un approccio ingegneristico rigoroso per gestire la complessità del software.

Software

Il software è l'**insieme di logiche, algoritmi e strutture dati che permette a un dispositivo elettronico di svolgere compiti complessi**. Ma, a differenza dell'hardware, il software deve essere progettato con attenzione per essere:

- Manutenibile nel tempo;
- Adattabile a contesti diversi;
- Testabile e aggiornabile;
- Sviluppato in modo collaborativo.

La programmazione è solo una parte della software engineering. Il vero valore emerge nella gestione del ciclo di vita completo del software: dall'analisi dei requisiti fino alla manutenzione a lungo termine.

Ciclo di vita del software

Le fasi fondamentali sono:

1. Analisi dei requisiti: comunicazione con il cliente per identificare le funzionalità richieste;
2. Design e architettura: suddivisione in moduli, definizione di interfacce, struttura dei dati;
3. Implementazione: scrittura del codice, test locali e integrazione;
4. Testing e QA: validazione del funzionamento e correzione di eventuali difetti;
5. Deploy e manutenzione: rilascio e supporto nel tempo.

Processo ingegneristico

Un buon processo ingegneristico deve:

- Fornire stime di costi e tempi realistiche (proprietà del processo);
- Produrre codice di qualità, efficiente e conforme alle specifiche (proprietà del prodotto);
- Essere ripetibile, ovvero offrire risultati affidabili a parità di condizioni;
- Permettere adattabilità rispetto ai cambiamenti nei requisiti.

Attività gestionali

Per completare con successo un progetto software non bastano le fasi tecniche: sono necessarie anche attività di coordinamento:

- Project Management: pianificazione dei compiti, gestione del tempo, dei costi e dei rischi;
- Configuration Management: controllo delle versioni, gestione dei rilasci, tracciamento delle modifiche;
- Quality Assurance: definizione degli standard qualitativi, testing automatico, revisione del codice.

Tutte queste attività si intrecciano per rendere il progetto coerente, ben documentato e facilmente manutenibile.

Manutenzione

La fase di manutenzione è la più lunga e costosa di tutto il ciclo di vita. Un software di successo può durare anche decenni, richiedendo adattamenti continui a nuove tecnologie, requisiti di sicurezza o modifiche dell'hardware.

Il software invecchia non per i bug, ma perché cambia il contesto in cui si trova: nuove versioni di sistemi operativi, cambiamenti nei dispositivi, aggiornamenti normativi, ecc.

Codici etici e qualità professionale

L'ingegneria del software richiede anche una componente etica: la consapevolezza che il proprio codice può influenzare sistemi critici, dati sensibili o la sicurezza di persone. Per questo motivo, è importante seguire codici di condotta professionale e mantenere uno spirito collaborativo, soprattutto nelle fasi di revisione.

La comunicazione efficace tra membri del team è essenziale: molti problemi di progetto nascono da incomprensioni più che da errori tecnici.

Modelli di sviluppo

Esistono diversi modelli di processo per organizzare il lavoro:

Modello a cascata (Waterfall)

Prevede fasi rigidamente sequenziali: una si completa prima di iniziare la successiva. Richiede molta documentazione ed è utile in progetti con requisiti molto stabili.

Tuttavia, è poco flessibile: se i requisiti cambiano (come spesso accade), il processo deve essere ripetuto da capo.

Modello incrementale

Si costruisce prima un core funzionante, poi si aggiungono feature attraverso versioni incrementali. Permette di avere feedback precoci e miglioramenti graduali.

Ogni incremento è una build parziale ma funzionante del sistema.

Modello formale

Utilizzato in contesti critici (es. aerospazio, medicina), prevede l'uso di strumenti per la verifica automatica del codice e modelli matematici per garantire assenza di errori.

Richiede strumenti dedicati e un alto livello di specializzazione.

Conclusione

L'ingegneria del software è un'attività sistemica, collaborativa e inter-disciplinare. Richiede tanto competenze tecniche quanto capacità organizzative, etiche e comunicative. Solo padroneggiando tutte queste componenti si può costruire software solido, scalabile e realmente utile.

Verification & Validation

Il processo di **verifica e validazione** ("Verification & Validation" o V&V) è fondamentale nell'ingegneria del software per garantire che un sistema funzioni correttamente e rispetti le aspettative dell'utente. Questi due concetti, spesso confusi, hanno scopi e significati diversi:

- La **verifica** consiste nel controllare che il prodotto sia stato costruito **correttamente**, cioè in conformità alle specifiche tecniche.
- La **validazione** controlla che sia stato costruito il **prodotto giusto**, cioè che risponda ai bisogni reali dell'utente finale.

> *"Are we building the product right?" (verification) - "Are we building the right product?" (validation).*

Entrambe le fasi hanno come scopo quello di ridurre il numero di errori (bug) nel software, garantendo al tempo stesso:

- Qualità del codice;
- Conformità ai requisiti funzionali e non funzionali;
- Affidabilità e manutenibilità.

Tecniche di verifica

Le tecniche di verifica si suddividono in **statiche** e **dinamiche**.

Tecniche statiche

Sono quelle che non richiedono l'esecuzione del programma. Le principali sono:

- **Revisione del codice** (*code review*), spesso svolta in coppia o in gruppo;
- **Analisi statica** tramite strumenti automatici (es. SonarQube, FindBugs);
- **Walkthrough** e **inspection**, ovvero letture guidate del codice con l'obiettivo di identificare errori.

Queste tecniche aiutano a trovare errori logici, strutturali o violazioni di stile prima dell'esecuzione.

Tecniche dinamiche

Queste prevedono l'esecuzione del programma e l'osservazione del suo comportamento. Comprendono:

- Test di unità: verificano il corretto funzionamento di singole unità (es. classi o metodi);
- Test di integrazione: controllano la corretta interazione tra componenti del sistema;
- Test di sistema: validano l'intero sistema in un contesto vicino a quello reale;
- Test di accettazione: svolti dall'utente per confermare la rispondenza ai requisiti iniziali.

L'obiettivo dei test dinamici è trovare errori esecutivi, regressioni e incompatibilità tra moduli.

Strategia di testing

Il modello a V (V-model) è una rappresentazione grafica che collega ogni fase di sviluppo a una corrispondente fase di testing:

```
Requisiti       <-->  Test di accettazione
Specifiche      <-->  Test di sistema
Design          <-->  Test di integrazione
Codifica        <-->  Test di unità
```

Ogni verifica è associata alla fase che intende validare. Il modello a V sottolinea che il testing deve iniziare in parallelo alla progettazione, non alla fine del progetto.

Altri tipi di test

Esistono test specifici che integrano o supportano i test funzionali:

- Test di regressione: assicurano che modifiche successive non compromettano funzionalità già testate;
- Test di performance: misurano efficienza, velocità, consumo di risorse;
- Test di usabilità: valutano l'esperienza utente e l'interazione con l'interfaccia;
- Test di sicurezza: cercano vulnerabilità, accessi non autorizzati e falle di protezione.

Testing automatico

Automatizzare i test è cruciale per aumentare la copertura e ripetibilità. I test automatici:

- Vengono eseguiti a ogni build (CI);
- Documentano il comportamento atteso del software;
- Sono mantenuti e aggiornati come il resto del codice.

Strumenti come JUnit, Selenium, Mockito e Cucumber sono ampiamente usati nei diversi livelli del testing.

Bug

Un bug è un comportamento anomalo osservato nel software rispetto alle attese. Le fasi per gestirlo sono:

1. Riproduzione dell'errore con un test;
2. Identificazione del problema (debugging);
3. Correzione del codice;
4. Rilancio dei test per verificare che il bug sia stato risolto e non siano stati introdotti altri errori.

Questo processo è detto anche ciclo di riparazione.

Metriche di qualità

Per monitorare la qualità del software, vengono utilizzate metriche come:

- Code coverage: percentuale di codice eseguito durante i test;
- Cyclomatic complexity: misura della complessità dei percorsi di esecuzione;
- Numero di bug per release: indicatore di affidabilità;
- Tempo medio tra i fallimenti (MTTF): misura della stabilità del sistema.

Queste metriche aiutano a tenere sotto controllo l'evoluzione del progetto nel tempo.

Conclusione

Verification & Validation sono parte integrante di un processo ingegneristico maturo. Non solo aiutano a identificare e correggere errori, ma promuovono buone pratiche, ordine mentale, miglior comunicazione tra team e maggiore soddisfazione per il cliente. Un software è veramente completo solo quando è stato validato nel suo contesto reale e verificato nel dettaglio tecnico.

JUnit Tests

Il framework JUnit è lo standard per il testing unitario in Java. È stato sviluppato nel 1997 da Kent Beck ed Erich Gamma e da allora si è evoluto in diverse versioni: dalla versione 3 a JUnit 4 (più flessibile e basata su annotazioni) fino a JUnit 5, che introduce un'architettura modulare e nuove funzionalità pensate per Java 8+.

Il testing unitario consente di verificare in isolamento il comportamento di singole componenti (classi, metodi) del software, aumentando l'affidabilità e facilitando il refactoring.

Scrivere test aiuta a:

- Formalizzare i requisiti;
- Facilitare la scrittura e il debugging del codice;
- Rilasciare software funzionante più frequentemente;
- Automatizzare il controllo di regressione.

Ciclo di vita del test JUnit

Ogni metodo di test eseguito da JUnit segue un ciclo:

1. Setup (creazione delle risorse necessarie);
2. Esecuzione del metodo da testare;
3. Verifica del risultato atteso (assert);
4. Teardown (rilascio delle risorse).

Questo schema è standardizzato tramite le annotazioni:

```
@BeforeEach // JUnit 5
void setup() { ... }

@Test
void testMethod() { ... }

@AfterEach
void teardown() { ... }
```

In JUnit 4:

```
@Before
public void setup() { ... }

@After
public void teardown() { ... }
```

Struttura di un test case

Un test case è una classe che contiene uno o più metodi annotati con `@Test`. Ogni metodo testa una funzionalità specifica. I metodi devono:

- Essere `public` (in JUnit 4);
- Non avere parametri;
- Non restituire valore.

Esempio:

```
@Test
public void testStack() throws StackException {
    Stack s = new Stack();
    assertTrue(s.isEmpty());
    s.push(10);
    assertFalse(s.isEmpty());
    assertEquals(10, s.pop());
}
```

Asserts

JUnit fornisce diversi metodi per verificare condizioni:

```
assertTrue(condizione);
assertFalse(condizione);
assertEquals(atteso, ottenuto);
assertNull(oggetto);
assertNotNull(oggetto);
assertSame(expected, actual);
assertNotSame(expected, actual);
fail();
```

Ogni assert può ricevere anche un messaggio:

```
assertTrue("Stack should be empty", stack.isEmpty());
```

Gestione delle eccezioni

Ci sono due modi per testare che venga sollevata un'eccezione:

JUnit 4 (con annotazione):

```
@Test(expected = StackException.class)
public void testPopVuoto() {
    Stack s = new Stack();
    s.pop();
}
```

JUnit 5 (con `assertThrows`):

```
@Test
void testPopVuoto() {
    Stack s = new Stack();
    assertThrows(StackException.class, () -> s.pop());
}
```

TestSuite

Una TestSuite raccoglie più classi di test da eseguire insieme.

JUnit 4:

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

JUnit 5:

```
@RunWith(JUnitPlatform.class)
@SelectClasses({TestStack.class, AltroTest.class})
public class AllTests { }
```

Disabilitare e filtrare test

- `@Ignore("motivo")` (JUnit 4) o `@Disabled("motivo")` (JUnit 5): salta l'esecuzione di un test.
- `assumeTrue(...)`: esegue il test solo se una certa condizione è vera.

Tipi di fallimento

- Failure: un assert non viene rispettato (il codice funziona ma non come atteso);
- Error: viene lanciata un'eccezione non prevista, come `NullPointerException`.

AssertJ e Hamcrest

JUnit supporta anche altri stili di asserzioni, più espressive:

Hamcrest:

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

assertThat(valore, is(equalTo(atteso)));
```

AssertJ:

```
import static org.assertj.core.api.Assertions.*;

assertThat(res)
  .as("Controllo valore di ritorno")
  .isNotNull()
  .isEqualTo(42);
```

TDD: Test-Driven Development

Nel Test-Driven Development si scrivono prima i test, poi il codice che li fa passare. Il ciclo è:

1. Scrivi un test che fallisce;
2. Scrivi il codice minimo per farlo passare;
3. Rifattorizza (*Refactoring*) mantenendo il test funzionante;
4. Ripeti.

Questa pratica incrementa la qualità del design e previene regressioni.

Testing in Eclipse

Eclipse integra il supporto a JUnit:

- `Run as -> JUnit Test` esegue i test con visualizzazione grafica;
- I risultati sono indicati da barre rosse (fallimenti/errori) o verdi (tutti i test superati);
- È possibile organizzare i test in un source folder separato, per una chiara distinzione dal codice di produzione.

Per eseguire i test è necessario:

- Aggiungere le librerie JUnit nel classpath del progetto;
- Importare correttamente le classi e le annotazioni da `org.junit` (JUnit 4) o `org.junit.jupiter.api` (JUnit 5);
- Per suite: usare `org.junit.platform.runner.JUnitPlatform` e `org.junit.platform.suite.api.SelectClasses`.

Conclusione

JUnit rappresenta lo standard per il testing in Java. Con le sue evoluzioni ha introdotto flessibilità, modularità e integrazione con altri strumenti. Un uso efficace di JUnit permette di costruire software robusto, modulare e mantenibile. La sua integrazione in ambienti di sviluppo come Eclipse e l'interoperabilità con strumenti come Maven e Gradle lo rendono indispensabile nel ciclo di vita del software moderno.