

Programmazione ad'Oggetti

Leonardo Mengozzi

Titoletti indice link a rispettive sezioni, in alto a sinistra "[←Indice](#)" link a pagina Indice.

Indice

1 Fasi sviluppo Software	4
1.1 Problem space vs Solution space	4
1.2 Programmazione ad oggetti (OOP)	4
2 Teoria degli oggetti	4
2.1 Com'è fatto un buon oggetto	5
3 Perché Java?	5
3.1 Elementi fondamentali	5
3.2 Differenze con C	5
3.3 Gestione Oggetti in memoria	6
4 Struttura Programma Java	6
4.1 Esecuzione Programma	6
4.2 I package	7
4.2.1 Compilazione ed'esecuzione Avanzata	7
4.2.2 Gradle	7
4.3 Convenzioni	8
4.3.1 I Commenti	8
5 (Almost) Everything is an object	9
5.1 Stack e Heap	9
5.2 Tipi Primitivi	9
5.2.1 Boolean	9
5.2.2 Caratteri	10
5.2.3 Interi	10
5.2.4 Virgola mobile	10
5.3 Conversioni	11
5.4 tipo statico e a rum-time	11
5.5 Autoboxing	11
5.6 Array	11
5.6.1 Foreach	12

6	Classi	12
6.1	Campi	13
6.2	final e costanti	13
6.2.1	Oggetti immutabili	13
6.3	Metodi	13
6.3.1	Variable arguments	14
6.3.2	return this	14
6.4	Costruttore	14
6.4.1	Overloading	14
6.4.2	Initializer	15
6.5	this	15
6.6	static	15
6.7	Livelli d'accesso	15
6.7.1	Incapsulamento	16
6.8	Precisazioni	16
6.9	Fasi Implementazione	17
7	Oggetti lato Utente	17
8	Dipendenze	17
8.1	Poliformismo Inclusivo (subtyping)	18
8.2	Interfacce	18
8.3	Composizione/delegazione	19
8.4	Ereditarietà	20
8.4.1	Overriding	20
8.4.2	Vtable	21
8.4.3	Classe Object	21
8.5	Classi Astratte	21
9	OOP2.0	22
9.1	Generics	22
9.1.1	Problema type-erasure	23
9.2	Wildcards	23
9.3	Classi innestate statiche	24
9.4	Inner Class	25
9.4.1	Local Class	25
9.4.2	Classi anonime	25
9.5	Lambda expression	26
10	UML (Class Diagram)	26
11	Design Pattern	27

12 Collezioni	27
12.1 Iterable e Iterator	28
12.2 Collection	28
12.2.1 Collezioni immutabili	29
12.3 List	29
12.4 Set	30
12.4.1 Ordinamento	30
12.5 Moduli	30
12.6 Map	31
12.6.1 Map.Entry	31
13 Enum	31
14 Reflection	31
14.1 Approfondimento ClassFile	32
14.2 Classe Class	32
14.2.1 Annotazioni	32
15 Gestione degli errori	33
15.1 throw	33
15.1.1 throws	34
15.2 try-catch-finally	34
16 Git	35
16.1 Concetti	35
16.2 Riferimenti ai commit	35
16.3 Operazioni base	36
16.3.1 Operazioni remote	37
16.4 Ignoring files	38
16.5 Branch	38
16.6 Merging	38
16.7 Upstream branches	39
16.8 Clonazione Repository remote	39
16.9 Fetching	39
16.9.1 pull	40
16.10 push	40
16.11 GitHub	40
17 JUnit	41
17.1 Esecuzione	41
17.2 JAR	41
17.2.1 Documentazione semi-automatica	41

1 Fasi sviluppo Software

Un Programma (algoritmo) risolve una classe di problemi.

Un Sistema software fornisce varie funzionalità grazie alla cooperazione di componenti di diversa natura.

Fasi processo sviluppo: **1 Analisi** che fare?, **2 Design** come farlo?, **3 Implementazione/codifica** Quale algoritmo?, **4 Post-cofica**. Fasi 1-2 fatte dai senior e fase 3 junior (2-3 oggi unificate). La fase 4 più impiegare fino 70% se fase precedenti fatte male/sbrigativamente (Software crisis). Tutte fasi fattibili dalla stessa persona.

Un analisi è corretta se persone diverse giungono alla stessa soluzione.

1.1 Problem space vs Solution space

Problem space sono le entità/relazioni/processi del mondo reale che formano il problema. Solution space sono le entità/relazioni/processi nel mondo artificiale (esprese nel linguaggio di programmazione).

Per passare dal Problem space al Solution solution si esegue un "mapping" che più semplice è meglio ho fatto le **astrazioni**¹.

I linguaggi di programmazione attuano l'astrazione coi loro costrutti, più o meno performanti, che rendono il mapping più o meno facile.

I linguaggi moderni hanno un livello d'astrazione lontano dall'HardWare, i suoi problemi e la gestione della memoria.

1.2 Programmazione ad oggetti (OOP)

Vantaggi: poche astrazioni chiave, mapping ottimo e semplice, estensibilità e riutilizzo, librerie auto costruite, C-like, esecuzione efficiente. **Critiche:** necessaria disciplina.

2 Teoria degli oggetti

Classe: Descrizione comportamento e forma oggetti. Indica come comunicare con i suoi oggetti, con messaggi che modificano stato e comportamento.

Oggetto: Entità (istanza di classe) manipolabile, con memoria, che comunicano tramite le loro operazioni descritte dalla classe di appartenenza.

Oggetti della stessa classe hanno comportamento e forma indentica, sono detti simili. Un oggetto non cambia mai classe, semmai si elimina e sene crea il sostituto.

Nota: L'approccio OOP è usato anche in UML.

¹Strumento che semplifica sistemi informatici ma anche del mondo reale evidenziando "la parte importante". Si possono fare più livelli di astrazione.

2.1 Com'è fatto un buon oggetto

Un oggetto ha un interfaccia², deve fornire un servizio, deve nascondere le implementazioni (riutilizzabili) e l'intero oggetto deve essere riutilizzabile tramite ereditarietà. Precisazioni:

- Un oggetto fornisce un **sotto-servizio** dell'intero programma³ (principio decomposizione). Linne guida: 1 oggetto senza servizio si elimina, 2 oggetti con più servizi si divide.
- L'implementazione di un oggetto (logiche interne) devono essere note solo al creatore della classe (Information hiding), così facendo l'utilizzatore è tutelato, da modifiche interne, avendo una piccola visione del tutto. *less is more*.
- Il creatore e l'utilizzatore riutilizzano le classi con gli approcci:
 1. **has-a** (composizione), classe costituita da altre classi (oggetto ha come campi altri oggetti). Approccio dinamico, occultabile.
 2. **is-a** (ereditarietà), classe estende servizi di un'altra classe (oggetto ha campi/metodi di altri oggetti).

3 Perché Java?

- **Write once run everywhere**, eseguibile ovunque senza ricompilazione grazie JVM (HardWare virtuale, a stack) che processa un codice specifico "byte code", creando il corrispettivo eseguibile per ogni pc/os. Meno prestante.
- **Keep it simple, stupid**, in teoria non in pratica.

3.1 Elementi fondamentali

Espressione: fornisce un risultato, riusabile ove si attende un valore.

Comando: istruzione da terminare con ";", non componibile con altre

Alcune parole chiave del linguaggio: `for`, `while`, `do`, `switch`, `if`, `break`, `continue`, `return`, `var`, ecc.

3.2 Differenze con C

- Condizioni di `if`, `for`, `while`, `do` restituiscono `boolean`.
 - Nel `for` è possibile dichiarare le variabili contatore (visibili solo internamente).
- ```
for (<tipo> <c1>, ...; <condizioneBoolean>, <modificaC1>, ...) {...}
```

<sup>2</sup>Insieme dei metodi definiti dall'interfaccia con cui l'oggetto riceve messaggi.

<sup>3</sup>Set di oggetti che si comunicano cosa fare.

- Java dà `unreachable statement, variable may not have been initialised`, `missing return statement`, ecc come errori, C solo come warning. Approccio più "rigido" ma anche più "corretto".

### 3.3 Gestione Oggetti in memoria

Alla creazione di un oggetto, la `new` chiama il gestore della memoria, si calcola la dimensione della memoria da allocare, si alloca, si inizializza e si restituisce il riferimento alla variabile.

La memoria occupata da un oggetto, non si sa com'è organizzata, ma contiene i campi non statici e il riferimento alla classe con la tabella dei metodi. Gli elementi statici sono allocati in una sezione dedicata.

Il **garbage collector** (componente della JVM) dealloca automaticamente memoria heap non più utilizzata direttamente o indirettamente. Un'oggetto continua a esistere dopo la fine esecuzione dello scope di una variabile che gli fa riferimento.

## 4 Struttura Programma Java

Un programma Java è composta da librerie di classi del JDK, Package<sup>4</sup> e Moduli<sup>5</sup>, librerie di esterne e un insieme di classi fondamentali.

La classe cardine di un programma è la class main<sup>6</sup>.

```
1 class NomeProgetto {
2 ...
3 public static void main(String[] args) {
4 ...
5 }
6 ...
7 }
```

Il parametro del main è un array di stringhe che sono i parametri che l'utente può immettere da tastiera quando il programma è lanciato da CLI. Poco usato.

### 4.1 Esecuzione Programma

1. Salvare la classe in un file "**NomeClasse.java**".
2. Compilare con `javac NomeFileClasse.java`. Genererà il **bytecode** `NomeFileClasse.class` per la JVM.
3. Eseguire con `java NomeFileClasse`. La JVM cercherà il main da cui partire a eseguire.

Lavorando con più file: si compila tutto con `javac *.java` poi si esegue solo il bytecode contenente la classe main.

<sup>4</sup>Contenitori, gerarchici tra loro, di una decina di classi di alto livello con scopo comune.

<sup>5</sup>Insieme di Package costituente un frammento di codice autonomo.

<sup>6</sup>Un main è il punto d'accesso di un programma.

## 4.2 I package

Per prassi nella cartella di un progetto, si fa corrispondere ai package auto-costruiti delle directory del file system.

Di default il package di una unità di compilazione<sup>7</sup>, ".java", è la root di gerarchia. Per definire il package di appartenenza si specifica a inizio sorgente:

```
package <pName>;
```

Per poi usare una classe di un package bisogna importarla (non aggiunge codice nell'eseguibile):

- `import java...;` importa una singola classe.
- `import java...*;` importa l'intero Package.
- `import java.lang.*;` importazione di default.

Il nome completo di una classe dipende dal Package in cui si trova. Se non si importa bisogna specificare all'uso il nome completo.

### 4.2.1 Compilazione ed'esecuzione Avanzata

Per tenere sorgenti e eseguibili separati, in progetti grandi si dispone la cartella *src* per i sorgenti (.java) e la cartella *bin* per i bytecode (.class).

```
javac -d "<dirExe>"[-cp "<lib1:|;...>"] <.../elencoSorgenti>
```

```
java -cp "<dirExe[lib1:|;...]>"<fullyClasseMain>
```

- `-d` specifica al compilatore la cartella dove creare i .class. Se non già presente la crea.
- `-cp` linka le classi dei percorsi specificati (divisi da : o ;) al classpath<sup>8</sup>.
- La JVM si aspetta il FQCN (Fully-Qualified Class Name).

### 4.2.2 Gradle

Build system ibrido composto da:

- **Progetto** Cartella contenente i file *build.gradle.kts* e/o *settings.gradle.kts* (build files).

1. Il build contiene la logica di costruzione del software.

```
plugins{java}
```

Io creo *src/main/java* per i .java e sarà creato *build/classes/java/main* per i .class.

---

<sup>7</sup>File.java compilabile atomicamente contenente classi una dopo l'altra.

<sup>8</sup>Sottopercorso (insieme ordinato) di cartelle che corrisponde al percorso del package dichiarato per la classe corrispondente. Il classpath di default è il JRE (Java Runtime Environment).

2. Il settings contine la personalizzazione, come il nome del progetto.

```
rootProject.name="..."
```

- **Plugin** Software contenente *task* pronti all'uso, come per java.
- **Task** Singola operazione atomica del processo di costruzione del software. Una esecuzione gradle richiede una o più task di cui sono automaticamente dedotte le dipendenze.  
Comandi: gradle tasks, gradle tasks -all, gradle compileJava, gradle clean.

L'esecuzione è delegata al compilatore del linguaggio specifico.

**Wrapper:** software scarica gradle e lo usa per creare il software. File del wrapper: gradle/wrapper/gradle-wrapper.jar (wrapper in s'è), gradle/wrapper/gradle-wrapper.properties (versione), ./gradlew oppure gradlew.bat (eseguono il wrapper).

Per compilare: ./gradlew compileJava

### 4.3 Convenzioni

- Linee max 90 caratteri e indenzazione 2-4 caratteri.
- Una sola istruzione per riga e quindi si definisce una sola variabile per riga, quando necessario, sempre inizializzata.
- Apertura "{" a fine riga della dichiarazione e chiusura "}" in linea a dove inizia la riga di apertura.
- Condice senza righe vuote di separazione a meno di metodi/costruttori.
- Non usare assegnamenti dentro espressioni e usare parentesi solo in espressioni non banali.
- Nomi classi e interfacce in "PascalCase". Nomi metodi, campi, variabili in "camelCase" e costanti in "SNAKE\_CASE". Nomi di package in minuscolo con numeri e senza "\_".
- I metodi che returnano, senza parametri, si chiamano `get...` oppure `is/has...` se return boolean.
- I metodi che returnano void, e accettano parametri, si chiamano `set...`

#### 4.3.1 I Commenti

Oltre a `//...` e `/*...*/` in java abbiamo `/** ... */` un commento multi linea usato per generare documentazione.



## 5 (Almost) Everything is an object

Le variabili, contenitori con nomi, ora non denotano solo valori numerici (come in C), ma anche veri e propri oggetti irriducibili.

Non ci sono meccanismi per controllo diretto memoria. Le variabili sono nomi "locali" con riferimenti ad *oggetti* e non maschere di indirizzi in memoria a cui accedere direttamente.

Le variabili posso essere di tipo *Java Types* quindi classi predefinite e autoimplementate oppure *tipi primitivi*.

Visibilità legata al blocco di definizione. Variabili non inizializzate sono inutilizzabili.

### 5.1 Stack e Heap

Gli oggetti sono memorizzati nell'**heap**. Tutte le variabili sono memorizzate nello **stack**.

Le variabili di tipo primitivo contengono direttamente il valore. Le variabili tipo classe contengono il riferimento dell'oggetto oppure null.

Nota: Uno stesso oggetto può essere puntato da variabili che si riferiscono alla stessa identità.

### 5.2 Tipi Primitivi

Ripasso: Un tipo classifica valori/oggetti tramite un nome, set valori, operatori.

I tipi atomici **signed** del C si sono mantenuti (non conveniva trattarli come oggetti) definendo un'unica interpretazione:

| Tipi primitivi | Dimensione | Minimo    | Massimo              |
|----------------|------------|-----------|----------------------|
| boolean        | –          | –         | –                    |
| char           | 16bits     | Unicode 0 | Unicode $2^{16} - 1$ |
| byte           | 8bits      | -128      | +127                 |
| short          | 16bits     | $-2^{15}$ | $-2^{15} - 1$        |
| int            | 32bits     | $-2^{31}$ | $+2^{31} - 1$        |
| long           | 64bits     | $-2^{63}$ | $-2^{63} - 1$        |
| float          | 32bits     | IEEE754   | IEEE754              |
| double         | 64bits     | IEEE754   | IEEE754              |

Le librerie *BigDecimal*, *BigInteger* gestiscono numeri di dimensione/precisione arbitraria.

**typing statico:** espressioni tipo noto al compilatore, vantaggio intercettazione errori.

**Nota:** Uso memoria non dato a sapere al programmatore.

#### 5.2.1 Boolean

Introdotta come tipo risultato delle espressioni e condizioni.

- **Valori** true, false.

- **Operatori Unari** ! (not). Peratori binari: & (and), | (or), ^ (xor), && (and-C), || (or-c)<sup>9</sup>.
- **Operatori confronto numerici** <, >, <=, >=.
- **Operatori di uguaglianza** ==, !=. Con gli oggetti di base confronta i riferimenti.
- **Operatore ternario** <eb> ? e1 : e2. restituisce e1 se eb è true, altrimenti restituisce e2. e1 e e2 espressioni dello stesso tipo.

### 5.2.2 Caratteri

Rappresentazione `'carattere'` o `'u<0-65535>'`. Formato ASCII o UTF16 (16bit). I caratteri d'escape si fanno `'\<specificatore>'`.

### 5.2.3 Interi

- **Codificati** in complemento a 2.
- **Operatori** +, -, \*, /, %, + e - unari, &, |, ~, <<, >>, >>>. Output stesso tipo Input.
- **Rappresentabili** in codifica decimale (1.000.000), ottale (0...), esadecimale (0x...).
- **Input tastiera** default int (più usato e efficiente). Se voglio un long va aggiunta una "l" dopo il numero.

### 5.2.4 Virgola mobile

- **Operatori** +, -, \*, /, %, + e - unari.
- **Codificati** in IEEE754.
- **Rappresentabili** in codifica decimale o scientifica.
- **Input tastiera** default double. Se voglio un float (più efficiente) va aggiunta una "f" dopo il numero.

**Ricorda:** IEEE754 ha errori di precisione che portano all'approssimazione dei risultati.

---

<sup>9</sup>& e | valutano sempre primo e secondo termine, essendo pensati per operazioni bit a bit mentre && e || valutano il secondo operatore solo se necessario.

### 5.3 Conversioni

- **Implicita** automaticamente applicata nelle espressioni e nell'assegnamento portando tutti tipi a quello più "generale" presente nell'espressione. Detta *coercizione*.

*byte* → *short* → *int* → *long* → *float* → *double*

- **Esplicito** fatto con operatore di casting: `...=<tipo><espressione>;`. Può causare perdita di informazioni.

### 5.4 tipo statico e a run-time

- **Tipo Statico**: Tipo di un'espressione desumibile dal compilatore.
- **Tipo run-time**: Tipo effettivamente presente è tipo statico o sottotipo.

Per ispezionare il tipo a run-time:

`<espressione> instanceof <tipo>`

Restituisce `boolean` e usato nella condizione di un `if` permette di specializzare esecuzione.

**Downcast**: Creare nuovo riferimento di tipo statico all'espressione (da `sovraTipo` a `sottoTipo`):

`... (<tipoEffettivo><espressione> ...`

### 5.5 Autoboxing

Ogni tipo primitivo può essere "boxed" in un oggetto.

- **boxing** azione di incasellamento di un tipo primitivo in un oggetto. `<Tipo> <variabile> = <Tipo>.valueOf(<tipoPrimitivo>)`
- **de-boxing** azione di estrarre un tipo primitivo da un oggetto. `... <varOggetto>.<tipoPrimitivo>Value()...`

Boxing e De-boxing sono effettuati in automatico.

### 5.6 Array

Oggetti (variabili hanno riferimento nello heap) di lunghezza esplicita e accessibile, inaccessibile fuori dai limiti (errore esecuzione), mutabili. Indirizzi elementi `[0, lunghezza-1]`.

Sintassi:

- `<tipo>[] <nome> = new <tipo>[] {v1,...,vn};`

- `<tipo>[] <nome> = new <tipo>[<dim>];`. Elementi inizializzati a valore default tipo (false, 0, null).
- `<tipo>[] <nome> = {v1,...,vn};`

Note: Si possono fare array di array. Gli array di oggetti sono oggetti con puntatori ad'altri oggetti.

Accesso a variabile ... `<nome>[<ind>]` ....

Per sapere la lunghezza dell'array `<nome>.length`.

### 5.6.1 Foreach

Astazione **for** in sola **lettura**, utile quando non importa ordine scorrimento elementi collezione e valore indice.

```
for(final<tipo> <v>: <e>)
```

- `v` è variabile del tipo indicato che vale via via ogni elemento della collezione.
- `e` è un'espressione che restituisce una collezione di elementi del tipo indicato.
- Si può esplicitare **final**, anche se non servirebbe, perchè il foreach crea una nuova variabile a ogni iterazione in realtà.

## 6 Classi

Sono template (tipo, struttura in memoria, comportamento) per generare oggetti (istanza).

Le classi hanno un nome (NomeClasse) che sarà anche il nome del tipo per le variabili e del file.

I membri fondamentali di una classe sono:

- **Campi**, descrivono la struttura/stato
- **Metodi**, descrivono i messaggi e il comportamento

Poi possiamo avere Costruttori, ecc.

Le classi sono tipi di dato in un linguaggio a oggetti tutto è un oggetto fino a un certo punto.

```

1 // prima public poi private.
2 [modAccesso] class <NomeDiUnaClasse> {
3 // Costanti
4 public static final <tipo> <NOME_COSTANTE>;
5 ...
6 // Campi
7 private [final] <tipo> <nomeCampo>;
8 ...

```

```

9 // Costruttore/i
10 public NomeClasse([<parametri>]) {...}
11 ...
12 // Metodi
13 [modAccesso] [static] <tipo> <nomeMetodo>([<parametri>])
 {...}
14 ...
15 // Campi e poi Metodi statici
16 static ...
17 }

```

## 6.1 Campi

Sono lo stato attuale dell'oggetto. Nome con la minuscola. Simili ai membri di una struttura C, con la differenza che possono essere 0,1,diversi (5-7max). Simili a variabili (tipo+nome), ma non si può usare *var*. Possono essere valori primitivi o altri oggetti (anche della classe stessa). L'ordine dei non conta.

I campi sono iniziabili alla dichiarazione dell'oggetto (coi parametri), sennò sono inizializzati in base al tipo a **0**, **false**, **null**. Mai alla dichiarazione si inizializzano (...=...;).

```
public|private|"[final] <tipo> <nomeCampo>;
```

Uso dei campi lato utente: Assegnamento ... `obj.campo = ...`, Lettura ...  
`= obj.campo ...`

## 6.2 final e costanti

Modificatore (da usare quando si può) per campi, variabili e parametri. Dopo inizializzazione non più modificabile il valore.

Per variabili di tipo classe non si può cambiare il riferimento all'oggetto, ma i campi dell'oggetto sì.

```
<tipo> final <nomeVar [= new ... | valore;]
```

Per fare le **costanti** (NOME.COSTANTE) usare `static final ...`; . Soluzione a *Magic Number*.

### 6.2.1 Oggetti immutabili

Oggetti i cui campi non si modificheranno mai, dopo la prima modifica del costruttore.

Si creano dichiarando `final private` tutti i campi della classe di tipo: tipi primitivi o oggetti immutabili.

## 6.3 Metodi

Definiscono il comportamento dell'oggetto. Nome con la minuscola. Simili a funzioni C. Hanno un'intestazione (tipo di ritorno—void, nome, argomenti) e un corpo. I metodi di una classe possono essere 0, 1, diversi.

i metodi possono leggere/scrivere i campi.

Uso dei metodi lato utente ... `obj.metodo()`.... L'invocazione del metodo, corrisponde a inviare un messaggio al receiver (obj nell'esempio) azionando l'esecuzione del corpo del metodo.

```
1 tipoDiRitorno nomeMetodo([tipo1 arg1, ...]) {
2 ...
3 [return ...;]
4 }
```

Nota: Classi POJO sono classi con solo campi privati, costruttori e metodi get e set.

### 6.3.1 Variable arguments

Per passare un numero indefinito di parametri di un certo tipo:

`<tipo> <nomeMetodo>([<parametri>], <tipo>... <nomeParametro>){...}`

- Va definito come ultimo o unico parametro.
- Nel corpo del metodo sarà trattato come un array di `<tipo>`.
- Il chiamante passa al metodo una lista di parametri di `<tipo>`.

### 6.3.2 return this

Un metodo può ritorna `this`, l'oggetto corrente. Deve avere come tipo di ritorno la classe stessa.

Usato per concatenare più metodi in una sola espressione, combinando i risultati parziali, e restituendo con l'ultimo metodo l'oggetto processato:

```
... obj.m1().m2()...mThis();
```

Questo schema è detto **fluente**.

## 6.4 Costruttore

Simile a un metodo con nome il nome della classe, senza tipo di ritorno e opzionalmente dei parametri formali.

`NomeClasse([parametri])...`

Se non si definisce il costruttore viene implicitamente inserito il costruttore di default (0 parametri) che inizializza i campi ai tipi di default.

Nota: per fare una classe non istanziabile basta fare i costruttori `private/protected`.

### 6.4.1 Overloading

Si possono definire più costruttori e metodi, ma non è una buona pratica. Nel caso devono essere **distinguibili dal numero e/o tipo** dei parametri.

I costruttori si possono **invocare a vicenda** (riuso codice). `this(...)` deve essere la prima riga del corpo del costruttore.

```
1 NomeClasse(p1) {
2 this(p1, ...);
3 ...
4 }
```

#### 6.4.2 Initializer

*staticInit(1) → nonStaticInit(+) → constructor(+)*

L'inizializzatore **static** è eseguito al caricamento della classe (1 volta sola). Usato per inizializzare campi statici.

**static** ...

L'inizializzatore **non static** è eseguito a ogni istanziazione della classe prima del costruttore.

.....

Di entrambi i tipi posso avere più inizializzatori per una classe, vengono eseguiti in'ordine di lettura una sola volta.

#### 6.5 this

Variabile contenente il riferimento all'oggetto che sta gestendo il messaggio corrente. Si usa per rendere meno ambiguo il codice accedendo tramite *this* a campi o metodi. (Usare sempre). ... **this**.campo ... **this**.metodo()...

#### 6.6 static

Scollega metodi e campi dagli oggetti di una stessa classe, rendendoli condivisi tra essi. I metodi diventano funzioni pure e i campi variabili globali alla classe.

Si possono fare classi solo statiche (utility class, NomeClasse+s) più non statiche (approccio migliore) o miste (campi e metodi static in fondo).

Richiamo fuori dalla classe: ... **nomeClasse**.campo|metodo().... Richiamo nella classe come normale campo o metodo. Richiamabile anche da un oggetto.

**static** campo|metodo|classe

Quando **static**? Si fanno statici i metodi o campi che sono indipendenti dallo stato di un oggetto della classe. Fanno un servizio indipendente dal singolo oggetto.

#### 6.7 Livelli d'accesso

Si antepongono a classi, metodi, campi, costruttori per definire il grado di utilizzo, in ordine crescente di libertà:

1. **private** Visibile e richiamabile solo nella classe di definizione.
2. **package** default (no keyword). Visibile e richiamabili dentro il package e invisibili fuori.

3. `protected` Visibile alla classe corrente e dalle sotto classi ricorsivamente. (in java anche al package).
4. `public` Visibile e richiamabile da qualunque classe.

In una unità di compilazione una sola classe è `public` e con lo stesso nome del file.

Vantaggio: Si può far rispettare al meglio il contratto<sup>10</sup> di un oggetto.

### 6.7.1 Incapsulamento

Scopo: Avere controllo su come vengono usati i dati.

Dichiaro `public` solo metodi/costruttori, di "design", necessari all'utente per interagire/creare l'istanza della classe.

Dichiaro `private` tutti i metodi/costruttori di "implementazione", e **tutti i campi**.

Così modifiche implementative non influenzano uso dell'utente. Controllo dati e riudo possibilità errori lato utente. Maggior distacco tra dato e implementazione. (*Information hiding* = nascondere implementazione informazioni).

**Principio di decomposizione (*divide et impera*):** Soluzione di un problema complesso è la somma di due o più sotto problemi più semplici, tra loro indipendenti.

Tali sotto problemi devono avere il minor numero di dipendenze<sup>11</sup> reciproche. Permettendo maggior autonomia, più modifiche senza danni collaterali e meno interazioni.

Una buona divisione da moduli con basse dipendenze esterne e alte dipendenze interne.

Nella OO le suddivisioni base sono package, classi e metodi.

Nota: Per rispettare l'incapsulamento quando si fa un get di un tipo non primitivo, bisogna restituire una copia del campo e non il campo stesso.

## 6.8 Precisazioni

Inizializzazioni particolari degli oggetti Stringa:

1. `... = new String();`, stringa vuota (è diverso da null).
2. `... = "..."`, come in C, comportamento speciale degli oggetti Stringa.

<sup>10</sup>Insieme degli scenari d'utilizzo e quindi aspettative di un utente al suo utilizzo.

<sup>11</sup>Riferito alle classi è quando una classe usa al suo interno un'altra. Ciò può comportare modifiche a cascata rischiando la *sindrome dell'"intoccabilità"*.



## 6.9 Fasi Implementazione

1. Progettazione della parte pubblica della classe (nome classe, signature metodi e costruttori necessari).
2. Costruzione dello stato (campi privati, con nome diverso dai metodi relativi).
3. Completamento implementazione. (test).
4. Miglioramento codice finale (commenti, eliminare *Magic Number*, fattorizzare sotto-funzioni). (test).
5. Test del risultato con test-case. (test-driven development con JUnit).

## 7 Oggetti lato Utente

Dichiarazione, creazione, inizializzazione:

```
<Tipo | var> <nome> = <new Tipo([argomenti])| altraVariabile | null>;
```

- `<Tipo> <nome>;` Si può solo dichiarare una variabile oggetto per poi crearla e inizializzarla successivamente.
- solo quando si scrive `new` (Keyword di linguaggio di chiamata al costruttore) si crea e inizializza un oggetto dalla classe indicata. `new` da il riferimento (`this`) dell'oggetto alla variabile.
- `var`<sup>12</sup> fa inferire<sup>13</sup> il tipo della variabile locale per alleggerire il codice. Se manca l'espressione non va, esempio `var i;`.
- `altraVariabile` deve essere della stessa classe della variabile che sto definendo.

## 8 Dipendenze

Il minimo numero di dipendenze fra classi sono manifestazione di "riuso" di codice e ne fanno un sistema e non un mero gruppo.

I tipi di dipendenze sono:

- **Associazione** (uses), un oggetto ne usa un altro.
- **Composizione** (has-a), un oggetto ne aggrega altri.

Un oggetto di una classe *si compone di* (campi) un insieme di altri oggetti, di altre classi.

Tale composizione può essere permanente (campo final) o opzionale (campo possibilmente `null`), multipla nota (più campi) o sconosciuta (campo array).

---

<sup>12</sup>Local variable type inference.

<sup>13</sup>Far dedurre al compilatore il tipo della variabile locale dall'espressione assegnata.

- **Specializzazione** (is-a), una classe ne specializza un'altra.

Nota: Nella composizione alla cancellazione dell'oggetto utilizzatore gli oggetti usati vengono cancellati a loro volta. Diversamente nell'aggregazione questi persistono.

## 8.1 Poliformismo Inclusivo (subtyping)

Fornire sovratipi che raccolgono classi uniformi tra loro. Gode del principio di sostituibilità utile a creare collezioni omogenee.

**Upcast:** Per il principio di sostituibilità quando ci si aspetta A si può usare B ma solo con le cose di A e non quelle esclusive di B. Ciò perchè B risponde almeno a tutti i messaggi di A (hanno i medesimi contratti).

**Con Interfacce** Sia B classe sottotipo di A interfaccia, con medesimi metodi (contratto) e altro possibilmente. Facile far aderire una classe a un'interfaccia.

**Con Classi** Sia B sottotipo di A, con medesimi metodi e campi (contratto+comportamento) e altro possibilmente. B e A hanno un comportamento compatibile.

**In memoria:** Intestazione (16byte) con indicazioni a run-time sulla classe dell'oggetto, tabella puntatori per vtable, campi privati di Object. Poi campi classe partendo da quelli della super classe.

**Poliformismo Parametrico** (genericità): ???.

## 8.2 Interfacce

Sopra tipo con caratteristiche comuni. Le classi implementative sono sotto tipi dell'interfaccia.

Separa esplicitamente l'interfaccia (contratto, fisso) dalla realizzazione (implementazione, variabile) della classe.

Consente diverse realizzazioni di un contratto, usabili omogeneamente. (più classi implementano medesime funzioni e si vuole usarele come fossero la stessa).

```
interface <NomeInterfaccia> {...}
```

- Nuovo tipo solo dichiarabile (no `new NomeInterfaccia()`) con valore oggetti delle classi che implementano l'interfaccia.

**Principio sostituibilità di Liskov:** A sottotipo di B, ogni oggetto A deve essere usabile dove ci si attende un oggetto B.

- Un oggetto tipo interfaccia consente solo chiamate ai metodi definiti dall'interfaccia e esegue l'implementazione specifica classe implementativa dell'interfaccia.

```
<NomeInterfaccia> <nomeVar> = new <ClasseImplementativa>();
```

Tipo statico è interfaccia e tipo run-time è tipo classe implementativa. **Late binding**: Codice da eseguire scelto dinamicamente, dipende da classe implementativa dell'interfaccia tipo della variabile.

- Include solo intestazioni di metodi. Sono automaticamente **public** e non si specifica.
- Compilato come una classe (produce un .class).
- Un interfaccia può estendere altre interfacce.

```
interface <NomeInterfaccia> extends int1[,...] {...}
```

Nel corpo dell'interfaccia si specificano solo i metodi che aggiunge.

```
class <NomeClasse> implements <NomeInterfaccia[,...]> {...}
```

**Una classe può implementare più interfacce.**

- Deve implementare il corpo di tutte le intestazioni di metodo delle interfacce (metodi in comune implementati una volta). La classe potrà avere altri metodi suoi.
- Le istanze avranno tipo la classe e le interfacce.

Nota: L'interfaccia è il sopra tipo/insieme delle classi, è più generale, ma fornisce meno funzionalità.

Nota: La classe che rispecchia l'interfaccia si chiama *impNomeInterfaccia*.

Nota: Le interfacce possono avere metodi statici ma non si fa.

### 8.3 Composizione/delegazione

Data una classe (di cui possono non avere i sorgenti) ne realizzo un'altra con caratteristiche solo in parte diverse, ovvero una specializzazione. Questa nuova classe potrebbe sostituire la precedente (gerarchia *is-a*).

Senza violare **DRY**: Don't Repeat Yourself.

Nella nuova classe:

1. **Incapsulare** un oggetto della classe da modificare/estendere mediante campo **private final**<sup>14</sup>. Sarà inizializzato nel costruttore della nuova classe.
2. **Metodi** possono delegare all'oggetto incapsulato e essere di più o meno, ampliando così le funzionalità. Viola poco DRY.
3. **Combinare** con l'uso delle interfacce per ottenere la sostituibilità.

Quando si può è meglio usare i metodi "ridefiniti" che "originali" per rispettare l'incapsulamento.

---

<sup>14</sup>Così che l'oggetto una volta inizializzato non possa essere cambiato ma si possa allo stesso tempo adoperare metodi e campi di conseguenza.

## 8.4 Ereditarietà

Definisce una classe specializzandone una esistente **ereditando**. Rispetta DRY.

```
class <ClasseFiglio> extends <ClassePadre> {...}
```

**Ereditarietà singola (tree), una classe può estenderne solo un'altra.**

- Eredita (usabili) campi, metodi, costruttori non **private**.
- Costruttori non privati del padre richiamati solo da costruttori figlio (`c.Figlio`\_`=c.Padre`), per inizializzare campi Padre.

```
super(<parametri>)
```

Se presente deve essere la prima riga dei costruttori del figlio che devono avere almeno i parametri dei costruttori del padre.

Costruttori default padre chiamato se **super()** assente/posizione errata.

- Figlio può aggiungere campi e metodi propri a quelli del padre (`m.Figlio`\_`=m.Padre`) essendone una specializzazione.
- Figlio accede a campi Padre (**private**) con get/set **protected** per rispettare incapsulamento.
- Classe Figlio vede la "catena" di ereditarietà come un'unica sovra classe da cui eredita.
- Una classe col modificatore **final** non è ereditabile.

Nota: Bastano i binari per estendere una classe.

Come le interfacce anche l'ereditarietà gode della sostituibilità. Posso fare variabili di tipo Padre ma che referenziano oggetti tipo Figlio. Questo perché i Figli essendo delle specializzazioni "sono anche padre".

### 8.4.1 Overriding

Riscrittura di uno o più metodi **public** del Padre nel Figlio.

- Per sovrascrivere un metodo bisogna usare la stessa firma, a meno che del modificatore d'accesso che può solo aprirsi verso uno meno restrittivo (fino a **public**).
- Un metodo definito col modificatore **final** non è Overridabile.
- Il metodo riscritto può invocare la versione Padre con **super**.`<metodoPadre>()`. Usabile non solo in caso di overriding.
- Un metodo ereditato o sovrascritto che usa **this** chiamerà la versione del metodo della classe chiamante (come ogni metodo).
- I metodi override hanno opzionalmente la notazione **Override@** utile al programmatore e al compilatore per controlli di sintassi.

### 8.4.2 Vtable

Nomi: vtable, call table, dispatch table.

Riporta la classe con il body da associare a ogni metodo definito o ereditato di una classe.

**Nella Memoria:** Nello stack abbiamo le variabili che fanno riferimento a oggetti di una certa classe, memorizzati nella heap. Dentro la memoria di questi oggetti nella heap si hanno i riferimenti alle vtable definite per ogni classe. Queste tabelle per ogni metodo della classe dicono se usare la versione della classe corrente (metodo ex-novo o sovrascritto) oppure di una sovraclasses.

### 8.4.3 Classe Object

Una classe estende sempre indirettamente la classe `Object`. `Object` è la radice di ogni gerarchia d'ereditarietà.

Fornisce metodi di uso generico:

- **toString() : String** Restituisce una rappresentazione in stringa dell'oggetto (utile per il debug). Automaticamente chiamato dall'operatore "+" nelle concatenazioni tra stringhe e oggetti. Ogni classe deve sovrascriverlo.
- **equals(Object obj) : boolean**: Dice se l'oggetto che lo invoca e quello passato come argomento sono uguali in base a criteri definiti da me. Deve essere riflessivo, simmetrico, transitivo, stabile e restituire false se passo `null`.
- **hashCode() : int** Genera un hash code dell'oggetto stesso. Implementato insieme a equals().
- **getClass() : Class** Restituisce un oggetto di `Class` che rappresenta la classe dell'oggetto stesso.

**Vantaggio?** Fattorizza comportamento comune e crea funzionalità che lavorano su qualunque oggetto.

## 8.5 Classi Astratte

Classi dal comportamento parziale, fatte per raggruppare contratti+comportamenti comuni e opzionalmente contratti successivamente specializzati.

```
abstract class ...
```

- Fornisce codice comune a classi figlie che la specializzano.
- Metodi già implementati, spesso `final`, definiscono il comportamento comune.

Metodi *astratti*, opzionali, hanno solo il contratto. Le classi figlio ne definiranno il comportamento specifico. Vanno definiti `abstract`: `abstract <tipo> <nomeMetodo>([<parametri>]);` Metodi astratti si definiscono solo in classi astratte.

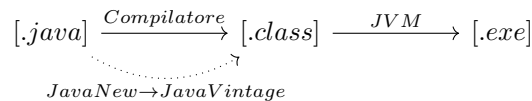
- Possono contenere campi, costruttori `protected`.
- Classi non istanziabili (no `new`).
- Una classe astratta può specializzare un'altra classe astratta o un'interfaccia senza l'obbligo di specializzarne i metodi.
- Una classe astratta può estendere una classe non astratta.
- Una classe non astratta è tenuta a specializzare i metodi astratti della classe astratta da cui eredita.

Intermedio tra ereditarietà tra classi e interfacce.

## 9 OOP2.0

Nella OOP1.0 abbiamo il riuso per composizione, estensione, poliformismo inclusivo (subtyping).

La OOP2.0 aggiunge il poliformismo parametrico (generics), ???, con lo scopo di dare maggiore controllo al programmatore e espressività al linguaggio JAVA.



Il compilatore risolve i costrutti della OOP2.0 trasformandoli in costrutti della OOP1.0 lasciando la JVM invariata.

### 9.1 Generics

**Scopo:** Creare funzionalità di lavorare uniformemente su oggetti indipendentemente dal loro tipo che diventa un parametro della funzionalità.

**Vantaggio:** Fattorizzare, tramite astrazione, soluzioni/classi ricorrenti altamente simili in una unica soluzione/classe riusabile.

Concetti chiave:

- La **type-variable**, solitamente "X", è una variabile che contiene il tipo dell'oggetto a run-time.
- La type-variable è usabile ogni qualvolta si potrebbe scrivere un tipo.
- Possono esserci più type-variable.
- Tipi primitivi non accettati, usare i Wrapper automaticamente risolti dall'autoboxing.
- Non passare tutti type-variable lato utente è errore.

```

1 // Lato classe
2 ... class <NomeClasse><X [, Y, ...]> {
3 ...
4 }
5
6 // Lato utente
7 ... <NomeClasse><tipo1 [, ...]> <nomeVar> = new NomeClasse
 <tipo [, ...]>(<parametri>);

```

**Inferenza dei parametri:** Nella `new` i type-variabile si omettono lasciando il *diamond symbol* "`<>`". Alternativa usare la `var`. In rari casi entrambi.

Per creare contratti uniformi si fanno interfacce generiche:

```
... interface <nomeInterfaccia><X [, Y, ...]> {...}
```

Le type-variable sono usate nei metodi definiti e le classi implementative le deve istanziare.

Metodi che lavorano e restituiscono qualsiasi tipo:

```
def:... <X [, Y, ...]> <tipoRitorno> <nomeMetodo>(<parametri>){...}
call:... <X, [Y, ...]><nomeMetodo>(<parametri>)...
```

Inferiti automaticamente alla chiamata (non serve diamond symbol).

### 9.1.1 Problema type-erasure

Il compilatore, in fase *erasure*, sostituisce i generics usando il subtyping di `object` e facendo upcast implicito nelle classi che diventano monomorfe, mentre lato utente usa autoboxing e downcast esplicito.

Limitazioni implementative:

- Non si possono istanziare oggetti (`new`) di tipo type-variabile.
- Non è usabile `instanceof` type-variabile `| nomeClasse<X>`.
- Il downcast esplicito da unchecked warning<sup>15</sup>.
- Non posso fare array di generics. Vanno fatti array di `object` con i dovuti cast oppure vanno usate le liste.

## 9.2 Wildcards

Le wildcards (poliformismo vincolare) permettono di esprimere un insieme di tipi accettati dai parametri di metodo che usa generics.

Permettono di usare oltre ai metodi di `Object` anche quelli delle classi indicate.

```
<tipo> <nomeMetodo>(Tipo<? extends | super tipoRiferimento> nome,...){...}
```

<sup>15</sup>Non rompe il programma nel punto del warning ma lo diffonde altrove nel codice dove prima o poi si romperà.

- **extends X**: Bounded covariante. Accetta i sotto tipi di X oltre a X.  
Non usare metodi set (read only).
- **super X**: Bounded controvariante. Accetta i sovra tipi di X oltre a X.  
Non usare metodi get (write only).
- Solo ?, Unbounded, indica qualsiasi type-variabile. Come dire ? **extends** `Object`. Da un codice leggero.

**Importante:** Se due tipi sono sostituibili (covarianza e controvarianza) tra loro non è detto che sempre una classe generics che li implementa entrambi sia sostituibile (Unsafety). Gli array, simili a un tipo generic, sono unsafety perchè permettono covarianza.

Le wildcards possono essere usate insieme ai generics.

### 9.3 Classi innestate statiche

Dentro una classe|interfaccia A outer si innesta la definizione d'una classe|interfaccia B static nested.

```

1 class|interface A {
2 ...
3 static class|interface B { ... }
4 }
```

Precisazioni:

- Esternamente ad'A, B è un tipo richiamabile via A, **Outer.StaticNested**, secondo il modificatore d'accesso.
- Internamente ad'A, B è accessibile direttamente con `StaticNested`.
- Outer e `StaticNested` si vedono reciprocamente i membri **private**<sup>16</sup>.
- Se B è classe è instanziabile (**new**).
- Il nesting può essere multiplo e/o multilivello.
- Per le interfacce innestate il **public static** è implicito.

Approcci:

1. Outer è la classe astratta e le `StaticNested` pubbliche sono le implementazioni.
2. `StaticNested` privata per sotto-comportamenti (implementazione interfacce o estensione classe) di Outer.
3. `StaticNested` sono sotto stati dell'Outer non usabili indipendentemente.

<sup>16</sup>Vero significato di **private** è "privato a livello della outerclass più esterna".



## 9.4 Inner Class

Dentro una classe A outer si inserisce la definizione d'una classe B inner (membro non-statico).

```
1 class Outer {
2 ...
3 class InnerClass {
4 ...
5 }
6 }
```

Precisazioni:

- Inner si richiama su un'istanza di Outer (enclosing instance), **Outer.InnerClass**, secondo il modificatore d'accesso.
- Un oggetto Inner si crea da un'istanza di Outer, **... = OuterObj.new Inner(...)**;; secondo il modificatore d'accesso.  
Dentro Outer si fa **... = this.new Inner(...)**;
- Inner accede al enclosing instance con **Outer.this**.

Approccio: preferibile a classi innestate statiche quando **private**.

### 9.4.1 Local Class

Dentro un metodo d'una classe A Outer si inserisce la definizione d'una classe B local class (Inne Class di fatto).

```
1 class Outer {
2 ...
3 <fornalMetodo> {
4 ...
5 class LocalClass { ... }
6 LocalClass <var> = new LocalClass(...);
7 ...
8 }
9 }
```

Può usare direttamente (senza **this**) variabili final/di fatto final del metodo.

Approccio: Quando ho bisogno della classe solo dentro un metodo.

### 9.4.2 Classi anonime

Classi senza nome e costruttori creta al volo (Local Class di fatto) e da lì creta al volo un'istanza.

```
1 ...
2 AnonimusClass NomeMetodo(<parametri>) {
3 return new AnonimusClass() { ... };
4 }
5 ...
```

- Il metodo "contenitore" restituisce come tipo la classe anonima.
- Può usare direttamente (senza `this`) campi dell'Outer class.

Approccio: Utile per implementare interfacce e quando si deve creare un solo oggetto.

## 9.5 Lambda expression

# 10 UML (Class Diagram)

Linguaggio grafico, OO-based, di modellazione di software.

- **Rettangoli**: un box rettangolare per classe o interfaccia diviso in:
  1. Nome classe. Nel caso delle interfacce aggiunge `<<interface>>` a sinistra del nome.
  2. Campi (solo per classi). Signatura: `nome : tipo`.
  3. Metodi e costruttori. Signatura: `nome(arg1:tipo,...):tipoRitorno`.

- indica `private`, + indica `public`, # indica `protected` e sottolineato indica `static`.

Metodi e classi astratte si scrivono in corsivo.

Metodi e classi `final` (non overraidabili) si indica l'attributo {leaf}.

Per indicare l'uso dei generics si indica in un rettangolo tratteggiato il/i generics, dentro il rettangolo, del nome e poi si usano come tipi.

- Dipendenze tra classi:
  - Composizione è arco con rombo.
  - Associazione è arco con freccia.
  - Generalizzazione è arco tratteggiato con triangolo vuoto, raggruppabile.
  - Estensione è arco con triangolo pieno, raggruppabile.Nella classe figlia si specificano solo le aggiunte e override.

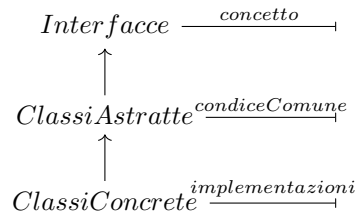
L'arco può essere indicato con le molteplicità: 1,0, n, 0...1, 1...n.

- Spesso accompagnato da descrizione testuale.

Spesso si omette tutto ciò che non è "design" e le signature ed'alcune relazioni.

## 11 Design Pattern

Approccio fondamentale:



Variabili, argomenti, tipi di ritorno sono interfacce mentre le **new** sono solo di classi Concrete.

- **Iteratore (Iterator)**: Oggetto usato per accedere ad una sequenza d'elementi (presi da una sorgente di dati).

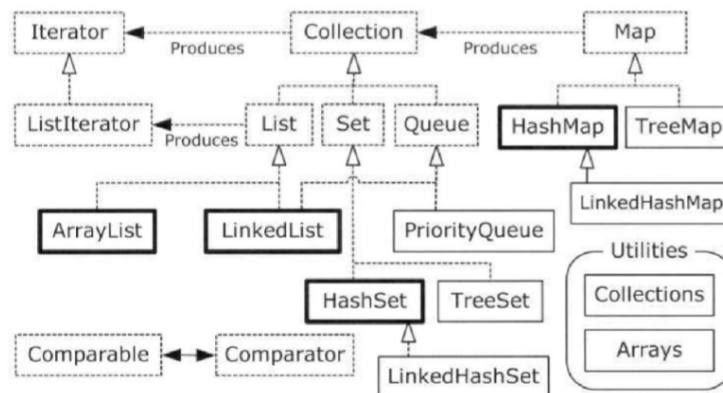
```

1 public interface Iterator<E> {
2 E next();
3 boolean hasNext();
4 }

```

## 12 Collezioni

Il Java Collections Framework (JCF), parte del **package** `java.util`, gestisce strutture dati e relativi algoritmi.



Di nostro interesse sono:

- Interfacce: `Collection`, `List`, `Set`, `Iterator`, `Comparable`.
- Classi: `ArrayList`, `LinkedList`, `HashSet`, `HashMap`.
- Classi Funzionalità: `Collections`, `Arrays`.

Fondamentalmente ci vengono fornite le collezioni (List, Set, Queue) e Map. Tutti con un generic.  
map: funzioni che mettono in relazione elementi di due gruppi.  
in java le collezioni sono tutte mutabili, ovvero si possono modificare gli elementi.  
tutte hanno add e toString().

## 12.1 Iterable e Iterator

Una classe che implementa l'interfaccia `Iterable<X>` può essere iterata.

Una classe di supporto che implementa l'interfaccia `Iterator<X>` dice come essere iterata.

```
1 public interface Iterable<T> {
2 /**
3 * Genera e restituisce un oggetto che
4 * implementa Iterator<T>.
5 */
6 Iterator<T> iterator();
7 }
8 public interface Iterator<E> {
9 /* Definisce quale' il prossimo elemento. */
10 E next();
11 /* Dice se c'e' un prossimo elemento. */
12 boolean hasNext();
13 /* Praticamente. */
14 void remove();
15 }
```

Le due classi usate insieme possono iterare collezioni.  
Esempio d'uso è nel foreach, che conosce le due interfacce, è può iterare qualunque oggetto di una classe che implementa `Iterable<X>` e che usa un oggetto di una classe che implementa `Iterator<T>`.

## 12.2 Collection

Interfaccia radice gerarchia interfacce di collezione (gruppi oggetti), estende `Iterable`, e definisce:

- Un costruttore vuoto. Assunto implicito.
- Un costruttore che accetta una `Collection` per l'inizializzazione. Assunto implicito.
- Operazioni di modifica opzionali.
- Operazioni di ricerca basate su `Object.equals()`.

```

1 public interface Collection<E> extends Iterable<E> {
2 int size();
3 // Preferibile isEmpty().
4 // Sempre O(1) secondo l'implementazione.
5 boolean isEmpty();
6 boolean contains(Object o);
7 Iterator<E> iterator();
8 Object[] toArray(); // Non usare.
9 <T> T[] toArray(T[] a); // Non usare.
10
11 boolean add(E e);
12 boolean remove(Object<?> c);
13
14 /**
15 * Bulk Operation: containsAll, addAll,
16 * removeAll, retainAll
17 */
18
19 void clear();
20
21 // ...
22 }

```

I metodi `contains` e `remove` applica il metodo `Object.equals()` all'oggetto passato con tutti gli elementi della collezione. Come argomento accetta `Object` per poter confrontare oggetti anche non della stessa classe (potenzialmente uguali per le regole di confronto).

### 12.2.1 Collezioni immutabili

L'unico modo per creare una collezione di valori non modificabili ma solo leggibili.

```
final ... = Set.of| copyOf(...); final ... = List.of| copyOf(...);
```

`Set | List.of| copyOf()` permette che i valori non siano cambianti e `final` fa sì che la variabile non sia riassegnata, rendendo il tutto un blocco unico.

## 12.3 List

Collezione estendibili sequenzializzata con indice. Aggiunge metodi per accesso elementi secondo posizione (O-based). Sono permesse ripetizioni di elementi (anche null).

Estende l'interfaccia `ListIterator<E>` che estende `Iterator<T>` permettendo di scorrere gli elementi sia "in avanti" che "in dietro".

Nota: Scandire quando possibile elementi via iteratore e non indici.

Implementazioni sono `ArrayList` (O(1) tranne `add` O(n) ammortizzato), con `trimToSize()` | `ensureCapacity()`, e `LinkedList` che implementa l'interfaccia `Queue` (FIFO o LIFO) e `Deque`.

## 12.4 Set

Collezione estendibili senza ordine e duplicati (`Object.equals()` sempre `false` e al massimo un `null`). Non aggiunge metodi a `Collection`.

Per migliorare le prestazioni di `contains()` si usa l'approccio **HashSet** ( $O(1)$  **aromtizzato**) che lega oggetto e posizione, o **TreeSet** ( $O(\log n)$ ).

Nota: I costruttori di `HashSet` accettano la dimensione iniziale stimata e un fattore di crescita, se basso hashing veloce, se alto hashing risparmia memoria.

### 12.4.1 Ordinamento

Per rendere gli oggetti di una classe ordinabili possiamo definire il criterio d'ordinamento:

- **Interno** alla classe tramite l'implementazione dell'interfaccia `Comparable` `<T>`.

```
1 public interface Comparable<T> {
2 int compareTo(T o);
3 }
```

Permette di comparare due oggetti dello stesso tipo secondo un unico criterio.

```
obj1.compareTo(obj2);
```

- **Esterno** alla classe tramite una classe comparatore che implementa l'interfaccia `Comparator` `<T>`.

```
1 public interface Comparator<T> {
2 int compare(T o1, T o2);
3 }
```

Permette di comparare due oggetti dello stesso tipo secondo tanti criteri quante sono le classi comparatore.

```
objComparator.compare(obj1, obj2);
```

Sia `compareTo` che `compare` restituiscono 0 se i due oggetti sono guali, un numero positivo se `this` o `o1` è maggiore oppure un numero negativo se `this` o `o1` è minore.

`TreeSet` e tante altre classi sfruttano l'ordinamento tra gli oggetti.

## 12.5 Moduli

Le classi statiche `Array` e `Collections` forniscono funzioni di base.

`Array` ha varie versioni di metodi per supportare tutti i tipi mentre `Collections` ne ha una per tutti.

## 12.6 Map

Collezioen estendibile di funzioni che mappano Key in Value.

```
public interface Map<K,V> ...
```

Map è biunivoca, due Value diversi non possono avere la stessas Key.

Implementazioni: HashMap, TreeMap.

### 12.6.1 Map.Entry

La mappa, internamente, ha un Set di Entry (coppie chiave-valore). Entry è una StaticNested interface.

```
1 public interface Map<K,V> {
2 ...
3 Set<Map.Entry<K, V>> entrySet();
4 interface Entry<K, V> {
5 K getKey();
6 V getValue();
7 V setValue(V value);
8 }
9 }
```

Quindi per iterare una mappa si usa *entrySet()* e il tipo degli elementi è *Map.Entryz < K, V >*.

## 13 Enum

Tipo che può assumere solo un numero fissato e limitato di possibili valori.

```
public enum NomeEnum VAL1, ...;
```

Per avere valori di qualsiasi tipo.

```
public enum NomeEnum VAL1(valore), ...;
```

Uso: ... NomeEnum.VALORE

Metodi: *valueOf("NOME\_VALORE")*, *values()[indice]*.

Le enum possono essere innestate.

## 14 Reflection

Approccio basato su librerie d'interazione con la JVM (*java.lang*, *java.lang.reflect* con *Filed*, *Constructor*, *Method*). Fornisce:

- Oggetti che rappresentano il contenuto d'una classi.
- Istanziazione diretta oggetto con accesso a metodi e campi.
- Forzare caricamento classi.

Lati negativi: lentezza, evita controlli compilatore (rischio sicurezza), da accesso a classi non note a priori.

Lati positivi: estendibilità, ispezionare, stringhe come identificatori di linguaggio, chiamate di metodo e creazioni d'oggetto dinamiche.

## 14.1 Approfondimento ClassFile

Un `.java` produce il classFile `.class` (`Outer$Inner.class` o `Outer$< numero >.class` per anonime) nel folder d'uscita innestato secondo package. Per ispezionare un classFile: `javap -v <nomeFile>.class`

Il **class-loader**, componente della JVM, cerca e "carica" i classFile.

- La ricerca avviene nel file system, jar o rete tramite classpath dei package.
- Il caricamento crea una struttura dati e inizializza i campi statici.

*Caricamento by-need:* Al momento del primo utilizzo (prima `new`, chiamata statica, uso sottoclasse). Solo le classi essenziali sono caricate all'avvio. Vantaggi: avvio rapido, ottimizza scenari di rete, "scaricamento" dalla JVM, hot-swapping, classi plugin senza spegnere, ispezione classi.

## 14.2 Classe Class

Classe `java.lang.Class` il cui oggetto rappresenta un tipo qualsiasi (anche void e classi generiche, no i Generics) della JVM.

Modi per ottenere un oggetto di Class:

- `String.class` (caso speciale, solitamente da `Class<?>` che richiede cast).
- `... = new NomeClasse(<parametri>).getClass()`.
- `Class.forName("NomeClasse")`.

### 14.2.1 Annotazioni

Meccanismo, usato a run-time grazie alla reflection, che "annota" (ignorate dal compilatore) pezzi di codice.

Annotazioni di libreria:

- `Override@` Rigetto se nella super-classe/interfaccia non è presente questo metodo non `final`.
- `SuppressWarnings()` Dichiaro che il codice è corretto impedendo segnalazione del warning specificato.
- Altre: `Deprecated@`, `FunctionalInterface`.

Ogni package può avere delle annotazioni custom per annotare classi, campi, metodi e costruttori.



```
1 public @interface NomeAnnotazione {
2 ...
3 }
```

I valori di default sono usati se lato utente non si specificano: `@NomeAnnotazione([prp1=val1, ...])`

## 15 Gestione degli errori

Gli errori in un programma si dividono in:

- **compile-time:** Grossolani e intercettabili dal compilatore (se linguaggio strong typing) o dall'editor (edit-time). Non sono bug.
- **run-time:** Condizioni anomale causate da dinamiche del sistema. Sono definibili, intercettabili e gestibili con costrutti dedicati. Sono i bug.

**Error-aware programming:** Gli oggetti richiedono condizioni di lavoro e al di fuori di esse devono fermarsi, effettuare azioni correttive e segnalare l'errore. L'utilizzatore degli oggetti deve saperli usare e intercettare/rimediare errori con un handler.

Tipologie d'errori in Java:

- Errori (`java.lang.Error`): Errori seri non risolvibili della JVM. Non si gestiscono.
  - Eccezioni unchecked (`java.lang.RuntimeException`): Bug nella programmazione. Risolti in fase di debugging.
  - Eccezioni checked (`java.lang.Throwable`): Prevedibili ma non rimediabili. Vanno dichiarati, intercettati e gestiti.
- Eccezioni lanciate da `throws` e gestita dal `try-catch` o rilanciate da altre `throws`.

Una classe estende `Exception` (checked) o `RuntimeException` (unchecked) per definire nuovi tipi d'eccezioni di eventi del dominio applicativo.

### 15.1 throw

Interrompe l'esecuzione del metodo, riporta l'oggetto creato al chiamante fino al main (exception chainig) e viene stampato sul `System.err`.

`throw <espressioneCheValutaUnOggettoThrowable>`

Errori e eccezioni unchecked sono lanciate da `throw` causando l'interruzione dell'applicazione e la stampa dell'errore e dello `StackTrace`<sup>17</sup>.

**Classe Throwable:** Permette di "lanciare" le eccezioni tramite oggetti. Sopra tipi di `Error`, `Exception` da cui eredita `RuntimeException`.

<sup>17</sup>Sequenza di chiamate a metodo che aiutano a dedurre il punto di rottura del codice.

I messaggi d'errore si mettono come costanti private della classe. Riusare classi del JDK, non si definiscono di nuove.

### 15.1.1 throws

Dichiara che il metodo può lanciare delle eccezione checked (obbligatorio dichiararlo). Il chiamante deve usare il costrutto `try-catch`.

`<method-contract> throws <checked-exception [, ...]> ...`

Nota: Eccezioni checked lanciate con `throw` dentro `try-catch` non necessitano di `throws`.

## 15.2 try-catch-finally

Costrutto che permette di gestire errori e eccezioni (checked e unchecked) senza interrompere l'esecuzione.

```
1 try {body-maybe-throwing-an-exception}
2 catch (<classeThrowable> <var>) {handler-body}
3 [catch (...) {...}]...
4 [finally {<completion-body>}]
```

Nel body del `try` c'è il codice che voglio eseguire e che può generare eccezioni. Nel body del `catch` c'è del codice che gestisce l'eccezione del tipo specificato, eseguito se questa si verifica.

- I catch vengono controllati nell'ordine in cui sono inseriti e non possono esserci più `catch` per la stessa eccezione.
- Per uno stesso `try` posso definire più `catch` e se voglio che più eccezioni vengano gestite nello stesso modo posso fare:

```
catch (<exp1> | exp2> | ... <var>)...
```

- Metodi di un oggetto eccezione: `e.getMessage()`, `e.printStackTrace()`.

Nel `finally` specifico codice che sarà eseguito se si verificano o meno delle eccezioni.

Eccezioni JDK del `java.lang` comuni: `NumericException`, `OperationNotSupportedException`, `StackOverflowError`, `NullPointerException`, `ArrayStoreException`, `ClassCastException`, `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`, `IndexOutOfBoundsException`, `IllegalStateException`.

## 16 Git

Git è un sistema di controllo di versione (SCV).

Un SCV può essere **distribuito**, ovvero ogni sviluppatore localmente ha una copia dell'intero "progetto" (repository), oppure **Centralizzato**, ovvero gli sviluppatori lavorano in sottoparti di un'unico progetto in remoto.

Git usa il modello distribuito per la maggiore scalabilità e facilità di branching e merging.

### 16.1 Concetti

- **Repository**: Contiene l'interistoria del progetto tramite l'uso di meta dati contenuti in una cartella nascosta della root directory.  
La struttura di una repository git è un alberlo/grafico.
- **WorkTree/WorkingDirectory**: Insieme di file della root directory che costituiscono il progetto. Sono esclusi i meta dati.
- **Staging**: Stato, o contenitore, in cui metti le modifiche dei file del workTree che voglio salvare col prossimo commit. Dopo un commit lo staging è vuoto.
- **Commit**: Stato salvato del progetto tramite tracciamento differenziale<sup>18</sup>. Inoltre crea uno *snapshotting* dello stato del workTree.  
Meta dati di un commit: commit padre (assente=commit iniziale, multiplo=commit di fusione), autore, messaggio, id univoco.
- **Branch**: Sequenza di commit con un nome. Usato nome di default al primo commit se non è stato indicato.
- **Remotes**: Copie della repository remote conosciute dal repository locale. Ogniuna ha nome e URL.  
I branch remoti possono essere identificati con `remoteName/branchName`.

### 16.2 Riferimenti ai commit

I riferimenti ai commit sono chiamati *tree-ish* e sono validi il nome del commit, il suo id (full o short), il nome del branch che fa riferimento all'ultimo commit avvenuto su quel branch, e HEAD.

**HEAD**: Nome speciale di commit che si riferisce al commit corrente ed'è linkata al branch del commit. Essenziale per muoversi nel workTree.

- **Riferimento assoluto**: Esplicito il tree-ish a cui voglio spostare la HEAD.

---

<sup>18</sup>Raccoglie le modifiche necessarie a trasformare il commit precedente in quello attuale.

- **Riferimento relativo:** Si fa `tree-ish~i` per dire l'i-esimo commit padre del tree-ish in cui voglio spostare la HEAD. In caso di merge seleziona il primo.

A ogni nuovo commit la HEAD vi ci si sposta automaticamente.

**DETACHED\_HEAD:** Modalità in cui la Head è delinkata da ogni branch e i commit fatti saranno persi.

### 16.3 Operazioni base

- **Configurazione Globale:** Configura git a livello di sistema.

```
git config --global category.option value
```

- **Configurazione Repository:** La configurazione di repository corrente sovrascrive quella globale.

```
git config category.option value
```

- **Inizializzazione Repository:** Inizializza una repository (vuota) nella directory corrente tramite il `.git` che esplicita la root directory. Il remote non è noto.

```
git init
```

Eliminare il `.git` comporta l'eliminazione della repository locale.

- **Staging:** Per aggiungere allo staging:

```
git add <files>
```

Aggiungere allo staging un file che non esiste più è l'equivalente della sua eliminazione nel prossimo commit.

Aggiungere allo staging un file che ha lo stesso contenuto di un file che non esiste più è l'equivalente della sua rinominazione.

Per rimuovere dallo staging:

```
git reset <files>
```

- **Commit:** Fa il salvataggio in un nuovo commit dei file selezionati nello staging. Data e id autogenerati. La HEAD deve essere sull'ultimo commit di un branch.

```
git commit -m "..."
```

Richiede la configurazione del nome e email dell'autore e del nome del branch. Cose da scrivere in un messaggio: file eliminati, aggiunte, modifiche.

- **Stato della repository:** Mostra informazioni sul branch corrente, quali file sono modificati, lo stato dello staging:

```
git status
```

- **Checkout:** Sposta la HEAD nel tree-ish specificato, se non ci sono modifiche volatili, aggiornando i file alla versione del tree-ish specificato.

```
checkout <tree-ish>
```

La HEAD va in DETACHED\_HEAD con `checkout <commit>` e ne esce con `checkout <branch>`.

- **Estrazione di file:** Altra funzione di checkout che permette di estrarre (riprisinare lo stato nella HEAD) da un tree-ish i file indicati.

```
git checkout <tree-ish> -- file1 [file1 ...]
```

- **Visualizzazione Storia:** Per visualizzare di base i commit dal HEAD alla root.

```
git log [--oneline] [--all] [--graph]
```

Il parametro `--all` mostra anche i branch fetched di `clone`.

- **Visualizzazione branch:** Visualizza tutti i branch locali e remoti.

```
git branch -a
```

- **Differenze:** Visualizza le linee modificate dei file tra due commit.

- `git diff` mostra righe aggiunte in staging.
- `git diff --staged` mostra righe differenti tra HEAD e workTree.
- `git diff <tree-ish>` mostra righe differenti tra tree-ish e workTree.
- `git diff --staged <tree-ish>` mostra righe differenti tra HEAD e workTree includendo le modifiche in staging.
- `git diff <from> <to>` mostra righe differenti tra due tree-ish.

Alcuni category.option: `user.name`, `user.email`, `core.editor`, `init.defaultbranch`.

### 16.3.1 Operazioni remote

- Visualizza la lista delle remote conosciute dalla repository locale.

```
git remote -v
```

- Mostra le informazioni di una remote.

```
git remote show nomeRemote
```

- Aggiunge una remote a quelle conosciute alla repository locale.

```
git remote add nomeRemote URI
```

- Fa "dimenticare" alla repository locale della remote.

```
git remote remove nomeRemote
```

## 16.4 Ignoring files

Per escludere automaticamente alcuni file della repository locale in quella remota, senza farlo a mano, si usa il file **.gitignore**.

Va inserito nella workTree root.

Ad'ogni aggiunta nello staging git ignorerà automaticamente tutti i file nei percorsi o delle estensioni specificate dentro **.gitignore**. Ignorato se si usa **git add --force ....**

Per creare/aggiungere **.gitignore** si fa da cli:

```
echo ... >> .gitignore
```

Per fare delle eccezioni di file che vogliamo includere nello staging:

```
!<nomeFile.estensione>
```

Nota: La compatibilità tra le modalità di andata a capo (LF e CRLF), in parte compensata da git automaticamente, è risolta esplicitando in **.gitattributes** quale modalità usare in base al tipo del file. Tranne per i file di estensioni specifiche di windows il default è LF. Va inserito nella workTree root.

## 16.5 Branch

Le brach sono i rami di sviluppo identificati da etichette. L'etichetta di una branch è mobile e sempre linkata, dalla creazione, al commit HEAD.

Creazione di una branch:

```
git branch brach_name
```

Per linkare la HEAD ai brach serve un **checkout** esplicito.

Se siamo in DETACHED\_HEAD possiamo creare un brach e linkare la HEAD:

**branch one-shot:** `git checkout -b new-branch-name`

## 16.6 Merging

I merge sono l'unificazione di più branch target con il branch corrente (quello a cui è linkato HEAD). Gli'ultimi commit dei branch verranno uniti in un nuovo commit che apparterrà al branch corrente.

```
git merge target
```

Il commit conclude la fase di merging. Commit precompilato di un merge corretto: `git commit --no-edit`.

**Avanzamento veloce:** Non è un'unione di branch ma uno spostamento per allineare due branch sulle modifiche. Uno dei due branch deve essere incluso nell'altro.

## 16.7 Upstream branches

Se branch remoti e locali sono associati (sono copie dello stesso branch) il branch remoto è l'**upstream branch**, configurabile con:

```
git branch --set-upstream-to=remote/branchName
```

## 16.8 Clonazione Repository remote

Copia l'intera storia della repository remota (URI) nella destinazione indicata (opzionale), ma "abilita" (checked out) solo il branch con la HEAD attaccata.

```
git clone URI [destination]
```

- Se la destinazione non è vuota da errore.
- Se la destinazione non è indicata viene creata una cartella con nome l'ultimo segmento di URI.
- L'URI può essere locale o remoto.
- Viene creato un remote (nome=origin) automaticamente.

Il branch locale "abilitato" (checked out), detto default branch, ha lo stesso nome del branch remoto.

Posso "abilitare" i branch **fetched** con `git checkout nomeBranchRemoto`.

I branch remoti sono **upstream branch**.

Nota: Quando si lavora con più remote o remote e locale i branch vanno cambiati di nome quando serve (tipo i main) o in certe operazioni (merge) si può referenziare con `nomeRemote/nomeBranch`.

## 16.9 Fetching

Controlla se una remote ha degli aggiornamenti (commit, branch, tag) disponibili e nel caso li scarica.

```
git fetch nomeRemote
```

Se HEAD è attaccata e il branch corrente ha un upstream, il controllo è sul remote/origin del branch.

```
git fetch
```

Per applicare gli aggiornamenti serve un merge (tra modifiche locali e quelle scaricate).

**fast-forward:** Caso di merge dove in locale non ci sono aggiunte rispetto alla remote (si copia solo).

### 16.9.1 pull

Unione di fetch e merge.

- `git pull` fa fetch dal remote/origin e mergia nel branch attaccato alla HEAD.
- `git pull nomeRemote` fa fetch dalla remote indicata nel branch attaccato alla HEAD.
- `git pull nomeRemote nomeBranch` fa fetch della remote indicata nel branch indicato.

### 16.10 push

Manda i cambiamenti locali al branch della remote, aggiornando la remote HEAD. Necessità privilegi di scrittura.

`git push remote branch`

Omettendo remote o branch viene usato.

`git push`

Si può impostare upstream a tempo di push.

`git push -u myremote mybranch`

Nota: `git push --tags` manda i tag e `git push --follow-tags` manda commit e poi tags.

**Fallimento:** se branch remote non è padre di branch locale, ossia: branch remote ha file non presenti in locale, branch remote non può fare fast-forward, branch remote non è sotto insieme di locale.

### 16.11 GitHub

Fornisce funzionalità come:

- **Forks:** Copia di una repository associata a diversi utenti/organizzazioni.
- **Richieste di pull/push:** Permette di estrarre aggiornamenti dai fork dei non "autorizzati":
- **Monitoraggio dei problemi.**

Su github i repository remoti sono identificati univocamente da owner/repo-Name.

**Autenticazione:** Via https con token o ssh (procedura slideGit3 pag.27).



## 17 JUnit

Crea classi di test tramite metodi annotati che testano scenari per asserire correttezza risultato rispetto a valore atteso.

```
1 @Test
2 firmaMetodo() {
3 ...
4 assert...
5 }
```

Metodi di asserzione: `assertTrue()`, `assertEquals()`, `assertFalse()`, `assertThrows()`

Per far fallire il test secondo una "nostra logica": `fail()`

Per far eseguire un test per primo si usa l'annotazione `BeforeEach@`.

### 17.1 Esecuzione

Eseguire il build comporta il check (controlli+test(`compileTestJava`+`compileJava`)) e asseble (`jar`+`compileJava`).

```
./gradlew build javadoc
```

Le cartelle principali sono: `/build`, `/src/main`, `/src/test`.

Configurazione Gradle (`build.gradle.kts`) per Junit slude 08 pag7-8.

### 17.2 JAR

JAR (Java ARchive) è il confezionamento di un progetto java (classi, risorse, Manifest) in un'unico file (ZIP).

**Manifest**, `META-INF/MANIFEST.MF`, è un file descrittivo dell'applicazione (classe con main del programma).

Il JAR può essere eseguito da `java` (se associato dall'os) e usato come libreria per altri progetti.

Un file JAR si crea con `jar` e le opzioni `c` (intenzione creare file jar), `f` (indica file output, altrimenti default `strout`), `m` (specifica MANIFEST, altrimenti default senza main start).

```
jar [c][f][m] nomeJar.jar [fileMANIFEST] [filesJava]
```

Per eseguire un JAR da CLI.

```
java -jar nomeJar.jar
```

Configurazione Gradle (`build.gradle.kts`) per Fat-JAR slude 08 pag16-.

#### 17.2.1 Documentazione semi-automatica

Usando `javadoc` e una sintassi dei commenti si crea automaticamente della documentazione HTML java.

Sintassi: I commenti del tipo `/** ... */` e in testa a classi/interfacce/membri, sono processati

Tag informativi:

- `param@` Per costruttori/metodi/classi parametriche, ne descrive un parametro.
- `type-parameters@` Per classi e metodi generici, ne descrive un parametro.
- `return` Per metodi, ne descrive il valore di ritorno.
- `throws@` Per costruttori/metodi, ne descrive le eccezioni che possono essere lanciate e il loro motivo.
- `see@` Linka altre entità che potrebbero interessare.
- `deprecated@` Da usare per entità deprecate (`Deprecated@`).
- `link [nameClass]#methodlink [nameClass]#method@` Link ipertestuali a entità della stessa classe o esterne.
- `code ...code ...@` Formatta il testo monospazionandolo.
- `nheritDocinheritDoc` Copia documentazione entità sovrascritta. Da usare se uso esterno rimane invariato.