# Riassunti di Ingegneria del Software

Simone Montali
monta.li

4 novembre 2019

## Prefazione

Questo progetto nasce dalla necessità di trovare un metodo di studio per questa materia che, ai più sembra banale. Il problema di fondo è proprio in questa apparente banalità: si finisce per studiarla di fretta pensando di conoscerla, e ci si rende conto troppo tardi di non essere pronti. Mi scuso, anzitutto, per il vocabolario misto italiano-inglese che utilizzerò in queste pagine. Tanti di voi sanno quanto sia complicato esprimere certi concetti in italiano. Ho inserito un piccolo glossario a fine documento. Questo documento ha lo scopo di essere il giusto mezzo tra completezza e sinteticità. Mi scuso, in secundis, per i toni a volte scurrili. Un vero informatico è arrabbiato */per il codice che non compila/LaTeXche fa ciò che vuole/il computer che si impalla/quel piccolo bugfix di Linux che diventa un bagno di sangue/*, e non c'è modo migliore di sfogare le incazzature informatiche che imprecare in riassunti che leggeranno le generazioni a venire. Non so come tu ti sia procurato questo documento, ma se hai 5 minuti da buttare, dai una letta alle cose che ho scritto qui. Ci troverai anche un'altra valangata di appunti. Buona studiata ed in bocca al lupo per tutto.

## 1 J1 - Java Overview

Java è un linguaggio object-oriented, derivato da C/C++, nato per applicazioni web. È semplice, multi-threaded, dinamico. Ha ereditato da C++ la sintassi, la mancanza di puntatori, la garbage collection, la mancanza di header files e preprocessori. Rispetto a C++, però, si avvia più velocemente, richiede meno codice, è indipendente dalla piattaforma e più facile da distribuire. Esistono 3 versioni:

- Java Standard Edition (J2SE): applicazioni stand-alone client-side
- Java Enterprise Edition (J2EE): applicazioni server-side
- Java Micro Edition (J2ME): utilizzato per dispositivi piccoli, come i cellulari

Il software viene eseguito sulla **Java Virtual Machine**, che permette di essere indipendente dalla piattaforma. I file .java contengono i sorgenti, i file .class il codice compilato (*bytecode*). Un file jar contiene il bytecode e i file multimediali, e viene utilizzato per distribuire l'applicazione.

### 1.1 Serializzazione e riflessione

La serializzazione è il meccanismo che converte oggetti in **byte streams**. La deserializzazione è il processo inverso. Questo meccanismo è utilizzato per trasferire e salvare oggetti. La **riflessione** permette a un programma di analizzarsi e manipolare le sue proprietà interne.

### 1.2 JRE e JDK

Il **Java Runtime Environment** è un pacchetto software contenente la JVM, alcuni librerie (.jar) e altri componenti. Il suo scopo è eseguire applicazioni scritte in Java. Il **Java Development Kit** è un superset, contenente gli strumenti atti a sviluppare, debuggare e monitorare le applicazioni.

### 1.3 Platform Module System

Introdotto da Java9, aggiunge un livello di aggregazione oltre ai pacchetti: i **moduli**. Un modulo definisce un gruppo riutilizzabile di pacchetti e risorse (immagini/XML/...). Un modulo ha bisogno di *Java Module Descriptor*, module-info.java, contenuto nella root del modulo. Può anche dipendere da altri moduli, ma aciclicamente. Le sue caratteristiche principali sono:

- Configurazione affidabile: la modularità permette di dichiarare con chiarezza le dipendenze
- Forte incapsulamento: i pacchetti in un modulo sono accessibili solo se esportati
- Piattaforma Java scalabile: la piattaforma java include tanti moduli, ma si può creare dei custom runtime per caricare solo i moduli necessari

Il **module descriptor** include nome, dipendenze, pacchetti esportati, servizi forniti, servizi utilizzati, e una lista dei moduli che sfruttano la reflection.

### 1.4 Garbage collector

Il **garbage collector** controlla la memoria e trova quali oggetti non sono più "referenziati" da variabili, per eliminarli. Inoltre, compatta gli oggetti rimanenti. Il GC non può essere invocato esplicitamente, ma si può suggerire alla JVM di farlo girare.

## 2 J2 - Development tools

- **Eclipse** è l'IDE che va per la maggiore su Java. È una piattaforma aperta, espandibile con plugin, che fornisce tool per programmare, compilare, debuggare.
- **Papyrus** è un editor UML basato su Eclipse, fortemente incentrato sulla customizability. Supporta anche i constraint OCL (vedi T12).

### 2.1 Build tools

I **build tools** vengono utilizzati per compilare e costruire immagini software dal source code. Richiedono due componenti: un build script che definisce le task da eseguire, ed un eseguibile che lo processa. Gli script dovrebbero essere platform agnostic.

#### 2.1.1 Maven

**Maven** è un tool di software project management che fornisce un setup di progeto semplice, con dipendenze e management di release e distribuzione. Incoraggia l'uso di una repository centrale di JAR e dipendenze. Permette la scrittura di plugin. I build files sono scritti in XML utilizzando il formato *Project Object Model* (POM), i pom.xml.

### 2.2 Static Analysis Tools

Questi tool possono trovare bug ispezionando il codice senza eseguirlo. Alcuni esempi sono Checkstyle, PMD, SpotBugs. Si overlappano in parte, ma si distinguono tra loro. Idealmente, andrebbero usati tutti.

- **Checkstyle** permette di seguire un coding standard, automatizzando il processo di verifica. Ha dei file di configurazione in XML.
- **PMD** utilizza un set di regole che analizzano diversi fattori del codice, come codice inutilizzato, duplicato, *over-complicato*
- **SpotBugs** verifica un set di bug patterns, come null pointers, cicli infiniti, deadlocks...

### 2.3 Altri tool

#### 2.3.1 VisualVM

VisualVM è un tool che integra dei tool da linea di comando di JDK. Monitora l'utilizzo della CPU, del GC, di memoria, thread e classi caricate. Fornisce inoltre informazioni sui crash. Può venire utilizzato sia in production che development.

#### 2.3.2 JUnit

**JUnit** è un software open source per lo unit testing di Java. Fornisce supporto per la scrittura, l'esecuzione e le annotazioni dei test. Fornisce assertion per verificare i risultati.

#### 2.3.3 Mockito

**Mockito** è un tool open source per il mocking e lo unit testing. Supporta la creazione di oggetti simulati che simulano oggetti reali in modalità controllate. Offre una sintassi semplice e leggibile, con annotazioni necessarie a ridurre il boilerplate.

## 3 J3 - Using Maven and formatting code

Il build con Maven segue un life cycle specifico per deployare e distribuire il progetto. Di default, abbiamo tre life cycles:

- Default: ciclo principale, deploya il progetto
- Clean: Pulisce il progetto e rimuove i file generati dal build precedente
- Site: genera la documentazione

Ogni life cycle consiste in una serie di fasi. Le fasi più importanti:

- **Validate** controlla che tutte le informazioni necessarie siano presenti
- **Compile** compila il codice
- **Test-compile** compila il codice di test
- **Test** esegue gli unit tests
- **Package** builda un pacchetto distribuibile (jar,war,...)

- **Integration-test** esegue gli integration tests
- **Install** installa il pacchetto in una repo locale
- **Deploy** copia il pacchetto sulla repo remota

I **plugins** sono utilizzati per inserire altri goal nella build phase. I build plugins sono eseguiti nella build phase, mentre i reporting nella reporting phase. Tutti i plugin devono avere informazioni di base come **groupID**, **artifactId**, **version**. Le **dipendenze** aiutano a definire, creare e mantenere delle build stabili con class-path e versioni definite. Possono caricare file jar dalle repositories, che conservano artifatti e dipendenze varie. Le repositories possono essere locali o remote. Maven, di default, usa la **central Maven repository.** I **profili** modificano il POM a build time, definendo una serie di modifiche da fare al POM quando attivati. Ad esempio, se avessimo un DB staging e uno di produzione.

# 4 J4 - Language structures

Elenchiamo ora alcune linee guida per la programmazione:

- Preferire l'utilizzo dei for-each o degli iteratori, piuttosto che indici, che inducono in errore
- Utilizzare array di lunghezza nulla per rappresentare il null
- Con **widening** intendiamo il casting di un subtype al suo genitore. Esso è svolto automaticamente durante un assegnamento.
- Con **narrowing** intendiamo il casting di un supertype a un suo figlio. Questo richiede un casting esplicito per colpa dello strong typing di Java.

# 5 J5 - Packages, classes, interfaces

Un **package** fornisce un namespace logico per un gruppo di classi *related*. I pacchetti definiscono una struttura descritta da un directory tree. Ogni pacchetto è descritto dal suo `package-info.java`, che deve essere nella directory.

## 5.1 Classes

Una **classe** è un'entità logica che definisce un gruppo di oggetti aventi attributi e metodi comuni. Alcune convenzioni di naming:

- Utilizzare nomi inglesi
- Utilizzare il mixed case (es. volevoMettereUnaBestemmia)
- Usare poche abbreviazioni e con consistenza
- Usare le parole complete al posto di acronimi (no: PD)
- Usare terminologia specifica
- Evitare nomi lunghi
- Evitare caratteri speciali all'inizio e alla fine

- Evitare nomi da una lettera (@Guido parlo a te, ti tengo d'occhio)
- Le classi devono avere l'iniziale maiuscola, il resto no
- Le costanti essere maiuscole con underscore a separare le parole
- I pacchetti dovrebbero essere minuscoli, separati da punti
- Il prefisso di un pacchetto è solitamente il sito web dell'organizzazione

La keyword **this** è utilizzata per accedere a metodi e attributi della classe in uso. La keyword **super** è utilizzata per accedere a metodi della superclasse. Non possono ovviamente essere usati in metodi statici. *Se non è così ovvio suggerirei un ripassino.* Queste keyword si possono usare anche per chiamare costruttori. Una classe può accedere ad un'altra tramite il nome, se è nello stesso pacchetto, o utilizzando il *qualified name* se è in un altro. Si può anche importare classi da un pacchetto.

### 5.1.1 Incapsulamento

L'**incapsulamento** è una forma di protezione; il mondo esterno non ha accesso all'implementazione interna dell'oggetto. I dati vengono nascosti, e si ottiene l'accesso solo tramite metodi: questo ne assicura l'integrità. Per l'incapsulamento sono fondamentali i visibility modifiers.

| Modifier | Field | Method | Top Level Class | Inner Class |
|---|---|---|---|---|
| **private** | none | none | not allowed | container class |
| **protected** | subclasses package | subclasses package | not allowed | subclasses package |
| **public** | all | all | all | all |
| | package | package | package | package |

Figura 1: Modificatori della visibilità

### 5.1.2 Methods

I metodi sono delle funzioni che eseguono determinate operazioni, permettendo il riutilizzo del codice. Accettano parametri e ne restituiscono altri. I **costruttori** sono metodi speciali che creano istanze di classe. Una classe può avere piò costruttori. Il nome è quello della classe. Se non sono definiti costruttori, viene utilizzato quello di default (vuoto), che viene invocato anche dalle subclasses. Alcune convenzioni per i metodi:

- Validare gli argomenti prima di utilizzarli: *mai dare per scontato che l'utente non sia una scimmietta incazzata.*
- Utilizzare la keyword `final` per gli argomenti, in modo da non poterli modificare nel codice
- Spaziare gli argomenti nella lista. *Ma non fate le bestie di satana mettendo lo spazio dopo la parentesi*

### 5.1.3 Fields and variables

Le variabili forniscono spazio in memoria. Hanno un tipo specifico che determina la memoria occupata, i valori assumibili, le operazioni applicabili. Il loro scope dipende dal modifier associato. **Lo scope delle variabili locali inizia nel punto di dichiarazione e finisce con la fine del blocco dove sono state dichiarate.** A partire da Java10, in alcuni casi non è necessario dichiarare il tipo. */s Am I PHP-dreaming?*

## 5.2 Static fields, methods, blocks

Gli attributi/metodi statici esistono indipendentemente dagli oggetti. Possono quindi essere chiamati senza presenza di oggetti, ma con alcune restrizioni: possono chiamare solo metodi statici, possono accedere solo a campi statici, non possono usare this/super. Sono ovviamente condivisi tra tutte le istanze. Un blocco static viene caricato una sola volta quando una classe viene caricata.

## 5.3 Final classes, methods, fields

Una classe dichiarata `final` non può essere estesa. Un metodo `final` non può essere overridato. Un field `final` non può essere modificato, è quindi costante. Va inizializzato alla dichiarazione, nei costruttori o in un blocco static.

## 5.4 Classe Serializable

Una classe che implementa l'interfaccia `Serializable` deve rispettare uno di questi tre constraints:

- Contenere dati primitivi
- Contenere oggetti Serializable
- Essere un transient

## 5.5 Oggetti immutabili

I campi sono final, le subclasses non possono overridare metodi. I metodi non possono modificare gli oggetti mutabili collegati ai fields, in quanto non sono condivisi.

## 5.6 Coding Conventions

Elenchiamo ora qualche coding convention.

### 5.6.1 Field coding conventions

- Dichiarare i campi private per fornire incapsulamento
- Inizializzare i final all'interno dei costruttori
- Non inizializzare i campi numerici con magic numbers, ma usare una costante

### 5.6.2 Variable coding conventions

- Dichiarare le variabili appena prima del loro utilizzo
- Evitare l'assegnamento di più variabili allo stesso valore in un singolo statement
- Non utilizzare i magic numbers
- Separare i membri in base al loro scope
- Utilizzare nomi esplicativi

### 5.6.3 JavaBean classes coding conventions

- Dovrebbero avere un costruttore vuoto
- Dovrebbero essere Serializable → supporta uno storing/restoring affidabile delle istanze
- Dovrebbe avere getter/setter con una convenzione di nomi: getCampo, setCampo, o se boolean anche isCampo

## 5.7 Ereditarietà

L'ereditarietà è il meccanismo che permette di ereditare campi e metodi delle superclassi. Il vantaggio principale è il riutilizzo del codice. Le subclasses non possono usare i campi privati, ovviamente. Possono anche overridare i metodi non final.

## 5.8 Method overloading e overriding

L'overloading avviene quando abbiamo più implementazioni dello stesso metodo, cambiando gli argomenti ma avendo ugual nome e return type. Un metodo è overridden quando una subclass ridefinisce un metodo con la stessa firma della classe padre.

## 5.9 Abstract classes

Le classi astratte forniscono un'implementazione parziale. Dovrebbero avere solo costruttori *protected*, possono includere metodi astratti. Se una classe ha metodi astratti, è astratta. Ma non è necessario.

## 5.10 Interfaces

Vi sono in realtà 3 definizioni di **interfaccia.**

- Software systems: è un confine condiviso tra due componenti separati che scambiano informazioni. Lo scambio può essere tra software, hardware e umani. es. Interfaccia di rete.
- Software engineering: è un principio chiave di design che proibisce l'accesso alle risorse di default, permettendo l'accesso solo tramite entry point definiti. Sono delle specifiche verificabili (contracts) per componenti software, che estendono la definizione ordinaria di tipi di dati astratti tramite l'uso di constraints
- Object-oriented languages: è un tipo astratto che non contiene dati ma definisce comportamenti tramite le firme dei metodi (nome, argomenti, return type)

### 5.10.1 Java interface

In Java, le interface sono reference type simili alle classi, che inizialmente contenevano solo metodi astratti e costanti. Da Java8, può contenere metodi default e static con implementazione. I metodi default offrono un'implementazione utilizzata da più subtypes, che possono overridarli, a differenza degli static. Da Java9 le interfacce possono anche contenere metodi privati. Le classi possono implementare **una o più interfacce**, ereditandone i metodi astratti. Le interfacce possono anche estendere altre interfacce (anche multiple). Per questo, le classi sono meno estensibili: possono farlo con una sola classe.

## 5.11 Ereditarietà e composizione

Eredità e composizione definiscono relazioni tra due classi per ridurre la duplicazione di codice e i bug. L'ereditarietà forma una relazione di tipo *is-a*, la composizione *has-a*

## 5.12 Marker interfaces e functional interfaces

Le **marker interfaces** sono interfacce vuote, ossia non contengono metodi e costanti. Le **functional interfaces** sono interfacce con un singolo metodo astratto.

## 5.13 Inner classes

Le **inner classes** sono classi definite all'interno di un'altra classe, e hanno 4 possibili tipi:

- **Non-static inner class** possono accedere a tutte le istanze private della classe esterna
- **Local inner classes** sono classi non statiche definite all'interno di un metodo
- **Anonymous inner classes** sono classi non statiche definite senza nome, come subclass o implementazione di un'interfaccia
- **Static inner classes** non sono tecnicamente inner classes. Sono in realtà un membro statico dell'outer class. Sono meno agganciate alla classe esterna, il che aiuta la qualità del codice, il testing, il refactoring. È in realtà una classe top-level, implementata dentro un'altra per motivazioni di packaging.

## 5.14 Enums

Gli **enums** sono classi speciali rappresentanti gruppi di costanti. È l'abbreviazione di enumeration. È implicitamente final, quindi non può avere subclasses. Le costanti sono public static. Gli enums possono essere definiti all'interno di una classe, possono contenere costruttori eseguiti separatamente per ogni costante. Può contenere solo metodi concreti (non astratti); può implementare più interfacce e dichiarare un main.

## 5.15 Annotations

Le **annotazioni** forniscono informazioni supplementari riguardanti il software. Non ne modificano l'azione, ma aiutano ad associare informazioni ad elementi del programma. Non sono commenti puri, in quanto possono modificare il comportamento del compilatore o la reflection. Alcuni esempi:

- **@Deprecated:** suggerisce che il metodo è deprecato ed è stato sostituito da una versione nuova
- **@FunctionInterface:** suggerisce che l'interfaccia è di tipo functional, e darà errore se non rispettato

- **@Override:** indica che il metodo è ereditato
- **@SafeVarArgs:** sopprime i warning del compilatore quando i parametri sono generics. Non usabile con l'overridde
- **@SuppressWarnings:** sopprime warning generici
- **@Documented:** indica che l'annotazione va documentata
- **@Inherited:** indica che un'annotazione della superclass è ereditata da una subclass
- **@Repeatable:** indica che la stessa annotazione può essere applicata più di una volta sullo stesso elemento
- **@Retention:** indica la retention policy
- **@Target:** indica il tipo di item a cui un'annotazione può essere applicata

### 5.16 Retention Policies

Abbiamo più tipi di retention policies:

- RetentionPolicy.**SOURCE**: permette, ad esempio, la creazione di file sorgente addizionali durante la compilazione
- RetentionPolicy.**CLASS**: permette, ad esempio, l'uso di librerie per la manipolazione del bytecode che accedono al bytecode e modificano classi esistenti o ne generano dinamicamente
- RetentionPolicy.**RUNTIME**: permette l'accesso alle annotazioni tramite Java reflection, può essere usata per dare istruzioni al programma associato

## Glossario

**magic number** Un magic number è quel numero che metti in una variabile quando è nulla ma non hai il null. Tipo -1. 6, 7

**platform agnostic** Con platform agnostic intendiamo software che può essere eseguito allo stesso modo su diverse piattaforme. 2