Riassunti di Ingegneria del Software

Simone Montali
monta.li

7 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. Vi sono però alcune differenze: mancanza di puntatori, garbage collection, mancanza di header files e preprocessori. Rispetto a C++, 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 progetto 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 più 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

La retention policy definisce quando le annotazioni verranno scartate. Abbiamo più tipi di retention policies:

- RetentionPolicy.**SOURCE**: le annotazioni verranno tenute nella source, e scartate durante la compilazione. Permette, ad esempio, la creazione di file sorgente addizionali durante la compilazione
- RetentionPolicy.**CLASS**: le annotazioni verranno tenute durante la compilazione, ma scartate per il runtime. 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**: le annotazioni saranno disponibili anche alla JVM. Permette l'accesso alle annotazioni tramite Java reflection, può essere usata per dare istruzioni al programma associato

6 J6 - Javadoc

Javadoc genera documentazione di codice Java prendendo in input dei file sorgente. L'output generato è un set di file HTML, più pacchetti e file di anteprima. Alcune delle feature principali sono:

- Combina il source code con la sua documentazione
- Facilita la sincronizzazione tra documentazione e codice
- Genera le specifiche API dal sorgente

L'approccio di Javadoc è molto semplice: sfrutta commenti con una sintassi specifica, aggiungendo la possibilità di integrare pagine HTML. Alcuni tag, per esempio, sono: `@author`, `@inheritDoc`, `@docRoot`. Abbiamo anche dei tag che fanno riferimento a risorse, ossia `@link`, `@linkplain` e `@see`.

7 J7 - Generics

I **generics** sono simili ai template di C++. Permettono a tipi predefiniti o user-defined di essere parametro di metodi, classi, interfacce. Dall'arrivo dei generics bisogna salvare tipi specifici nelle collection. Questo aggiunge stabilità al codice, rendendo i bug più visibili tramite type checks più stretti. Riduce anche la necessità di casting nel codice. Alcune convenzioni di naming:

- K: key
- N: number
- T: type
- V: value
- S, U, V: altri...

7.1 Metodi generici

I metodi generici sono metodi i quali parametri/return types possono essere generics. Hanno una sezione di type parameter delimitata da <> che precede il return type. Ogni sezione contiene uno o più tipi, separati da virgole. Un metodo generico è dichiarato come ogni altro metodo.

7.2 Wildcards

Una **wildcard** permette di usare più di un tipo di elementi all'interno di una collection. È un meccanismo atto a rendere possibile il cast di collections alla loro subclass/superclass. L'obiettivo è rendere possibile la lettura e l'addizione di una collection generica. Alcuni tipi di wildcard:

- Unknown wildcards List<?>: identifica una lista di tipo sconosciuto. È possibile la sola lettura, gli elementi vengono trattati come oggetti.
- Extend wildcards List <? extends A>: identifica una lista di elementi che sono sottoclassi di A. È possibile leggere (trattando gli elementi come istanze di A) ma non aggiungere.
- Super wildcards List <? super A>: identifica elementi della classe A o superclassi di A. È sicuro inserire istanze di A o sottoclassi, ma si possono leggere gli elementi solo come oggetti.

7.3 Generic Class Implementation

Ogni classe generica ha una singola implementazione basata sul tipo di oggetto. L'operatore `instanceof` non può essere utilizzato con un tipo generico. I generic types non possono essere usati per creare costruttori generici. Non si possono creare array generici.

8 J8 - Collection Framework

Un framework è un environment di software, universale e riutilizzabile, che fornisce alcune funzionalità facenti parte di una piattaforma software più grande, che facilitano lo sviluppo di software. Il **collection framework** fornisce un'architettura per salvare e manipolare gruppi di oggetti. Include vari tipi di interfacce e classi, e supporta le tipiche operazioni necessarie: ricerca, ordinamento, inserimento, ecc...

8.1 Iterator, Iterable, Collection

L'interfaccia **Iterable** è l'interfaccia sorgente di tutte le collection, e ha un metodo che ritorna un *Iterator*. L'interfaccia **Iterator** fornisce la possibilità di iterare gli elementi in avanti tramite l'uso di tre metodi per verificare la presenza di un elemento, spostarsi, e rimuovere. L'interfaccia **Collection** estende l'interfaccia *Iterable*, ed è l'interfaccia implementata da tutte le classi del collection framework. Nonostante l'interfaccia fornisca i metodi che le collezioni avranno, alcuni non sono implementati da alcune classi(*contraction*). I suddetti tre metodi sono:

- **hasNext()** ritorna vero se esiste un elemento dopo il corrente
- **next()** sposta l'iteratore al prossimo elemento. Se non esiste, throwa un'exception.
- **remove (E e)** rimuove l'elemento corrente. Può ovviamente essere chiamato una sola volta per elemento.

Le collection hanno alcuni metodi principali:

- **contains(Object o)** e **ContainsAll(Collection<?> c)** restituiscono true se la collection contiene uno/tutti gli elementi
- **isEmpty()** restituisce true se la collection è vuota
- **iterator()** restituisce un iterator
- **size()** restituisce il numero di elementi contenuti
- **stream()** ritorna uno stream sequenziale della collection
- **toArray()** ritorna un array di oggetti contenente gli elementi della collezione
- **toArray(T[] a)** ritorna un array del tipo richiesto, contenente gli elementi della collezione

8.2 Lists

L'interfaccia **List** definisce una struttura dati che gestisce una collezione ordinata di oggetti che possono avere valori duplicati. La classe **ArrayList** usa un array dinamico per salvare gli elementi, mantenendo l'ordine di inserzione degli elementi. Non è sincronizzato e l'accesso agli elementi è randomico. La classe **LinkedList** usa una lista doppiamente linkata internamente, mantiene l'ordine di inserimento e non è sincronizzata. La manipolazione di elementi è veloce perché non è necessario shifting. La classe **Stack** fornisce una struttura Stack aka LIFO, estendendo la classe vector. I principali metodi delle liste sono:

- **add(E e), add(int index, E e)** e **addAll(Collection <? extends E> c)** hanno un compito decisamente ovvio.
- **get(int index)** ha un compito altrettanto ovvio.
- **set(int index, E e)** setta l'elemento index al valore e. *In effetti era abbastanza ovvio pure questo.*
- **remove(int index), remove(E e), removeAll(Collection <?> c)** - *chissà cosa faranno mai questi* - rimuovono uno o più elementi
- **indexOf(Object o)** e **lastIndexOf(Object o)** restituiscono l'indice a cui si trova un elemento, oppure -1 se non c'è.

- **subList(int fromIndex, int toIndex)** genera una sottolista tra `fromIndex` (compreso) a `toIndex`(escluso)

L'utilizzo degli ArrayList è ottimale quando l'accesso è randomico e si aggiunge/rimuove solo dalla fine. La LinkedList, invece, viene usata quando è necessario manipolare anche gli elementi all'interno della lista.

8.3 Sets

L'interfaccia **Set** rappresenta un set non ordinato di elementi, che non ci permette di inserire duplicati. La classe **HashSet** rappresenta una collezione che usa una hash table per il salvataggio; l'hashing è utilizzato per il salvataggio. Il **LinkedHashSet** mantiene inoltre l'ordine di inserimento. Il **SortedHashSet** rappresenta un set ordinato in ordine crescente. Il **TreeSet** utilizza un albero per salvare: è molto veloce. *Se l'unico tipo di albero che ti viene in mente è quello sotto casa tua, fai una googlata: è roba figa e ti tornerà utile.* Elenco ora i metodi principali dei set:

- **add(E e)** aggiunge un elemento se non è già presente, e ritorna true se ha fatto.
- **remove(Object o)** rimuove l'elemento e ritorna true se l'ha trovato
- **addAll(Collection <? extends E> c)** costruisce l'unione tra i due set
- **retainAll(Collection<?> c)** costruisce l'intersezione dei due set
- **removeAll(Collection<?> c)** costruisce la differenza tra i due set

Gli HashSet vengono utilizzati quando è necessario inserire, eliminare, localizzare elementi. Il TreeSet è molto utile quando è necessario attraversare gli elementi secondo un ordine. Il LinkedHashSet è una via di mezzo. In base alla dimensione, può essere più veloce aggiungere elementi ad un HashSet piuttosto che convertirlo in un TreeSet.

8.4 Maps

L'interfaccia **Map** fornisce una struttura dati che contiene valori basati su chiave. Ogni coppia chiave-valore è detta **Entry**. Una Map contiene chiavi univoche. L'**HashMap** è una mappa non ordinata. La **LinkedHashMap** è l'implementazione di una mappa, eredita da HashMap ma mantiene l'ordine di inserimento. La **SortedMap** rappresenta un set di elementi in ordine crescente. La **TreeMap** è l'implementazione di Map e SortedMap, mantenendo ordine crescente. I metodi principali delle Map sono

- **put(K k, V v)** collega la chiave k al valore v. Se la chiave esiste, ne sovrascrive il valore.
- **get(K k)** ritorna il valore associato alla chiave, o null se non viene trovata
- **remove(Object k)** rimuove l'entry collegata. Se trovata, restituisce il valore, altrimenti null.
- **replace(K k, V v)** sostituisce il valore solo se la chiave viene trovata. Restituisce il valore precedente.
- **entrySet(), keySet(), values()** restituiscono rispettivamente le entry, le chiavi, i valori.

L'HashMap è ottimale quando è necessario l'inserimento, l'eliminazione, la localizzazione. La TreeMap è utile quando si necessita un'esplorazione ordinata delle chiavi. La LinkedHashMap è una via di mezzo. A volte, in base alla dimensione, può essere più veloce aggiungere elementi ad una HashMap piuttosto che convertirla ad una TreeMap ed aggiungerli poi.

8.5 Queues

L'interfaccia **Queue** mantiene l'ordine FIFO. Può essere definito come lista ordinata, ed è utilizzato per salvare elementi in attesa di processi. La **PriorityQueue** mantiene elementi che devono essere processari in base alla loro priorità; non ammette valori null. La **Deque** gli elementi possono essere aggiunti/rimossi da entrambi i lati. L'**ArrayDeque** e la **LinkedList** implementano la Deque interface. Questi sono i metodi principali:

- **add(E e)** inserisce un elemento nella coda
- **offer()** è preferibile ad add(E e), in quanto verifica di non violare restrizioni di capacità. Ritorna true in caso di successo.
- **peek()** prende il primo elemento della coda senza rimuoverlo. Se vuota, ritorna null.
- **element()** prende il primo elemento della coda senza rimuoverlo. Se vuota, throwa un'eccezione.
- **remove()** rimuove il primo elemento e lo restituisce. Se vuota, throwa un'eccezione.
- **poll()** rimuove il primo elemento e lo ritorna. Se è vuota, returna null.

La LinkedList è una normale coda FIFO. La PriorityQueue viene utilizzata quando è necessario un comparator.

8.6 Collections Class

La classe **Collections** fornisce un set di metodi statici per la creazione di implementazioni sincronizzate di collection, per la creazione di collection immutabili, per operazioni ricorrenti sulle collection. Alcuni metodi:

- **unmodifiableCollection(Collection<? extends T> c)** ritorna una view non modificabile della collection.
- **synchronizedCollection(Collection<T> c)** ritorna una collection sincronizzata (thread-safe).
- **checkedCollection(Collection<E> c, Class<E> t)** ritorna una vista dinamica typesafe della collection. Ogni tentativo di inserimento di elementi incoerenti col type throwa exception.
- **singleton(T o), singletonList(T o), singletonMap(K k, V v)** restituiscono set, liste e mappe con elementi/key-values.
- **copy(List<? super T> dest, List<? extends T> src)** copiano tutti gli elementi di src in dest
- **fill(List<? super T> list, T obj)** riempe la lista con l'elemento specificato
- **nCopies(int n, T o)** ritorna una lista immutabile con l'elemento fornito.
- **replaceAll(List<T> list, T oldVal, T newVal)** restituisce tutte le occorrenze di un valore con l'altro
- **reverse(List<?> list)** inverte l'ordine degli elementi
- **reverse(List<?> list, int distance)** inverte l'ordine fino alla distanza specificata

- **binarySearch(List<? extends Comparable<? super T>> list, T key)** implementa una ricerca binaria. La lista deve ovviamente essere ordinata. Ritorna l'indice dell'oggetto. *Se non sai cos'è una ricerca binaria googla anche questo. Tutta roba figa.*
- **frequency, min, max (Collection<?> c, Object o)** restituiscono la frequenza dell'elemento, il minimo, e il massimo
- **indexOfSubList, lastIndexofSubList(List <?> source, List<?> target)** ritornano la posizione della sublist trovata, o -1 se non c'è.

9 J9 - Functional programming

Ci poniamo ora una domanda: qual è la differenza tra programmazione imperativa e funzionale? La **programmazione imperativa** è un paradigma che utilizza degli statements per modificare lo stato di un programma. La programmazione funzionale, invece, tratta la computazione come valutazione di funzioni matematiche. Uno **statement** assegna variabili, una **funzione** è eseguita per produrre un valore. Quindi, il focus della programmazione imperativa è *come risolvere*, quello della funzionale è *cosa risolvere*.

9.1 Funzioni pure e di higher-order

Come cazzo si traduce higher-order? Le **funzioni pure** restituiscono lo stesso risultato quando ricevono gli stessi argomenti, e non causano effetti collaterali. Per questo, sono semplici da debuggare. Inoltre, non fanno mock. Le **funzioni higher-order** *(forse "di ordine maggiore"? Suona veramente male)* prendono funzioni come argomenti e/o ritornano funzioni come risultato.

9.2 Calcolo lambda

Il suddetto venne sviluppato da Alonzo Church per studiare la computazione tramite funzioni. Dà, in sintesi, la definizione di cosa è computabile. È un *framework teorico* per descrivere le funzioni, la loro valutazione è basata puramente su manipolazione sintattica di simboli e ricorsione. **Tutto** ciò che può essere computato tramite calcolo lambda è computabile, e offre la stessa *definitional power* di una macchina di Turing. Il calcolo lambda è la base di tutti i moderni linguaggi di programmazione funzionale. Il functional programming presenta però alcune difficoltà:

- Scrivere funzioni pure riduce la leggibilità del codice
- Scrivere programmi ricorsivi al posto di cicli potrebbe non essere semplice
- Scrivere funzioni pure è semplice, ma combinarle con altro software no
- I valori immutabili e la ricorsione potrebbero ridurre le performance

9.3 Espressioni lambda

Questo è un bel mindfuck. Le espressioni lambda sono metodi anonimi che possono implementare una functional interface. *Non ricordi cosa siano? Rileggi 5.12.* Il loro tipo è il tipo dell'interfaccia che implementano, e possono essere usate ovunque ci aspettiamo functional interfaces. Possono accedere solo ai field della classe che le racchiude, e alle variabili locali del blocco che le racchiude, solo se `final`. Una lambda, in termini pratici, consiste di una lista di parametri, una freccia, un corpo.

$$\texttt{(parameters)} -> \{\texttt{statements}\}$$

Parametri e return types possono essere omessi se determinabili dal contesto. Le lambda con una sola espressione possono omettere le graffe. Le lambda con un solo parametro possono omettere le tonde. Le lambda senza parametri devono però averle. Alcuni vantaggi delle espressioni lambda:

- Supportano la programmazione funzionale
- Permettono di scrivere codice più pulito e compatto
- Permette di scrivere programmi paralleli
- Fornisce API più generiche, flessibili, riutilizzabili
- Supporta il passaggio di behavior a metodi

9.4 Riferimenti ai metodi

Spesso le lambda expressions chiamano metodi esistenti. Non è però conveniente e conviene chiamarlo direttamente. I riferimenti a metodi sono, effettivamente, una forma semplice di espressioni lambda. Possono essere utilizzati per metodi statici, di istanza, e costruttori. Ne abbiamo, di conseguenza, tre tipi:

- Metodo statico: `ContainingClass::staticMethodName`
- Metodo di istanza: `containingObject::instanceMethodName`
- Costruttore: `ClassName::new`

9.5 Streams

Gli **streams** sono simili a collections, ma non mantengono i dati. Sono pensati per lavorare bene con le lambda expressions. Sono immutabili, ma processandoli ne possiamo ottenere nuovi. Muovono elementi lungo una sequenza di step di processo, detti *stream pipeline*, formato da chiamate *chaining* a metodi. La pipeline inizia con una sorgente di dati, esegue operazione intermedie, e conclude con un'operazione finale. Le operazione intermedie sono *lazy*: non vengono eseguite fintanto che un'operazione da terminale lo richiede.

9.5.1 Produrre streams

Ci sono vari metodi per produrre stream:

- Utilizzare il metodo statico `of` e passargli elementi/un array
- Convertire una collection utilizzandone il metodo `stream()`
- Convertire le linee di un file in stringhe, ed il file in uno stream di stringhe
- Ogni stream può essere convertito in uno stream parallelo usando il metodo `parallel()`

9.5.2 Trasformare streams

Abbiamo inoltre metodi di trasformazione degli streams:

- Il metodo `map` trasforma uno stream applicandovi una funzione
- Il metodo `filter` trasforma uno stream eliminando gli elementi che non soddisfano una condizione
- Il metodo `limit(n)` trattiene i primi n elementi
- Il metodo `skip(n)` rimuove i primi n elementi
- Il metodo `distinct` rimuove i duplicati
- Il metodo `sorter` ordina gli elementi dello stream

9.5.3 Collectiong results

Possiamo anche raccogliere determinati risultati:

- `count`, `max`, `min`, `sum` restituiscono un singolo valore
- Il metodo `toArray` restituisce un array
- Il metodo `collect` restituisce una lista o un set
- Il metodo `collect`, applicato ad uno stream di stringhe, riunisce tutte le stringhe in una sola

10 J10 - Eccezioni

Eccezioni ed **errori** sono eventi indesirati ed inaspettati che accadono durante l'esecuzione del programma, bloccando il normale flusso di esecuzione. Gli **errori** indicano problemi seri che non andrebbero *catchati*. Le **eccezioni**, invece, possono essere "recuperate". Quando un'eccezione/errore accade, viene generato un oggetto e viene passato al *runtime system* (aka la JVM). Questo oggetto contiene nome e descrizione dell'errore, e lo stato del programma quando è accaduto.

10.1 Espressioni checked e unchecked

Qual è la differenza? Le **eccezioni checked** rappresentano condizioni invalide in aree fuori dal controllo del programma: input dell'utente, database, rete, file mancanti. Queste sono sottoclassi di `Exception`, e i metodi sono obbligati a gestirle. Le **eccezioni unchecked**, invece, rappresentano difetti del programma, come argomenti invalidi passati ad un metodo. Queste sono subclasses di `RuntimeException`, ed i metodi non sono obbligati a gestirle.

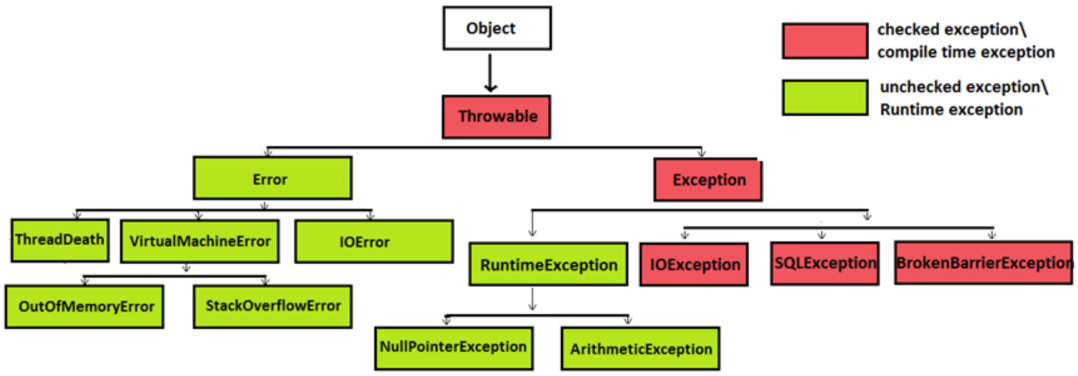

Figura 2: Gerarchia delle eccezioni

10.2 Exception handling

Gli statement di un programma possono *raisare* un'eccezione contenuta in un *try block*. Se un'eccezione accade in un try block può essere *caught* con un catch block. Bisognerebbe gestire tipi diversi di eccezioni in catch blocks diversi. Codice da eseguire in casi positivi e negativi va inserito in un finally block.

10.3 Try with resources

Generalmente, quando un try block utilizza risorse come file e connessioni, le suddette vanno chiuse in un finally block. Esiste però un metodo per non doverlo fare: dichiarare esplicitamente le risorse utilizzate tra parentesi prima del try block. La classe deve implementare l'interfaccia `AutoCloseable`.

10.4 Throws e throw keyword

Se il codice di un metodo può raisare *(scusate)* eccezioni ma non vuole gestirle, può "throwarlo" al metodo padre inserendo `throws` nella sua firma. Invece, la keyword `throw`, solleva esplicitamente un'eccezione.

10.5 Convenzioni di coding

- *Questo lo abbiamo fatto tutti:* **evitare catch blocks vuoti.** Questo perché quando accade un'eccezione non ti rendi conto di nulla. Non è ok.
- Essere specifici nella `throws` dei metodi
- Non raggruppare le eccezioni catchando una classe generica

11 J11 - Input and Output

11.1 File class

La classe **File** crea oggetti che forniscono una rappresentazione astratta di file/directory e ne supportnao la gestione. Supporta l'accesso agli attributi di file/directory, l'eliminazione, la rinomina, l'esplorazione di cartelle, i permessi.

11.2 Streams

Gli streams sono la classe principale del pacchetto `java.io`, e rappresentano sorgenti di input e destinazioni di output. Rappresentano diversi tipi di sorgenti/destinazioni, e supportano diversi tipi di dati. Possono passare dati ma anche manipolarli, e rappresentano la sequenza di dati letti/scritti da un programma. Due classi base: `InputStream` e `OutputStream`. In generale, si processano quattro tipi di dati:

- Byte streams: 8-bit bytes; FileInputStream, FileOutputStream
- Character streams: 16-bit Unicode; FileReader, FileWriter
- Data streams: dati primitivi; DataInputStream, DataOutputStream
- Object streams: oggetti; ObjectInputStream, ObjectOutputStream

11.3 Buffered streams

I **buffered streams** leggono dati da un'area di memoria detta *buffer*, l'API input nativa viene chiamata quando il buffer è vuoto. Per stream senza buffer, ogni richiesta I/O è gestita dal sistema operativo, perdendo in efficienza. I buffered stream compiono operazioni su stream di byte e caratteri, quindi solitamente wrappano byte/character streams.

11.4 Espressioni regolari

Prima o poi le dovrai imparare, eh. Le **espressioni regolari** forniscono un metodo "scientifico" per identificare stringhe testuali utilizzando *wild cards. Skipperò le spiegazioni, ma se vi interessa consiglio Regex101.*

11.5 Scanner

Lo **scanner**, definito in `java.util.Scanner`, è utilizzato per ottenere l'input di dati primitivi e stringhe. La sua funzionalità è basata su tre passi: lettura da fonte, identificazione dei dati tramite delimitatori, processazione. Spezza il suo input in *token* utilizzando dei delimitatori, di default lo spazio.

11.6 Standard streams

Java fornisce il supporto per I/O standard dove il programma prende input da tastiera e produce output sullo schermo. Java fornisce tre stream standard:

- `System.in` per l'input
- `System.out` per l'output
- `System.err` è utilizzato per l'output di errori prodotti dal software

11.7 Java NIO

Un secondo metodo è il **New I/O**, che supporta un approccio buffer-oriented e basato sui canali per l'I/O, con supporto a file system e file handling. Venne sviluppato per fornire un supporto ad alta velocità, non-blocking fino a che i dati non sono caricati. Fornisce alcuni oggetti, detti **selectors**, utilizzati per monitorare i gli eventi dei canali attraverso un singolo thread.

11.7.1 NIO paths and files

L'interfaccia `Path` rappresenta percorsi del file system, di tipo assoluto o relativo. Le istanze Path possono essere create utilizzando il metodo statico `get`, passando la stringa del path. La classe `File` fornisce metodi statici per manipolare file e directory.

12 J12 - Concorrenza

Il problema è sincronizzare le attività di un'applicazione che dovrebbero essere svolte parallelamente, evitando conflitti nell'uso di risorse. La soluzione? Multitasking, multithreading, multiprocessing.

12.1 Differenze tra processo e thread

Un processo è un programma eseguibile, con il suo spazio di indirizzi, che esegue un programma, comunica con files, rete, e **può contenere più thread.** Un thread, invece, è uno stream di istruzioni eseguite in sequenza, che condivide lo spazio degli indirizzi con altri thread, eseguendo una parte di programma. Comunica con accesso condiviso ai dati.

12.2 Metodi principali di un thread

- `start()` avvia l'esecuzione del thread
- `run()` definisce la task del thread
- `yield()` informa lo scheduler che il thread è a disposizione per concedere il suo uso corrente del processore
- `interrupt()` interrompe il thread
- `setPriority()` imposta la priorità del thread
- `sleep()` interrompe il thread per un determinato numero di millisecondi
- `wait()` interrompe il thread finché non viene risvegliato da un altro processo
- `notify()` e `notifyAll()` svegliano uno o tutti i thread
- `wait()`, `notify()`, `notifyAll()` sono ereditati da Object

12.3 Variabili volatile

La keyword `volatile` tenta di risolvere i problemi di visibilità di variabili. Tutte le modifiche su variabili volatili sono scritte immediatamente in memoria, tutte le letture sono svolte dalla memoria principale. Per questo, le variabili `volatile` sono più costose per la CPU. L'accesso a variabili volatili previene il riordinamento delle istruzioni, che viene normalmente svolto per migliorare le performance.

12.4 Problemi di atomicità

Un esempio: `i++` sembra un'istruzione atomica, ma è in realtà composta da tre operazioni: lettura, modifica, scrittura. C'è quindi un problema quando diversi thread lavorano contemporaneamente sulle stesse risorse.

12.5 Synchronized methods

Quando un thread sta eseguendo un metodo `synchronized` su un oggetto, tutti gli altri thread che invocano metodi sincronizzati sullo stesso oggetto vengono sospesi fino a quando non finisce.

12.6 Atomic object classes

Le classi oggetto atomiche forniscono metodi **atomici**, quindi le loro istruzioni vengono eseguite insieme, o nessuna viene eseguita. Un metodo atomico modifica il suo oggetto senza effetti collaterali. Queste classi cono definite in `java.util.concurrent.atomic`, con i seguenti metodi principali:

- `addAndGet()` aggiunge il valore passato al corrente
- `decrementAndGet()` decrementa il valore di 1
- `incrementAndGet()` incrementa il valore di 1
- `accumulateAndGet()` applica una funzione al valore corrente
- `lazySet(i)` setta il valore passato

12.7 Classi principali della collection Concurrent

- `CopyOnWriteArrayList()` è un'implementazione backed-up della copy on write array
- `CopyOnWriteArraySet()` come sopra ma con set
- `ConcurrentHashMap()` è un'implementazione ad alta concorrenza, alta performance, backed up da un'hash table
- `LinkedBlockingQueue`, `ArrayBlockingQueue`, `PriorityBlockingQueue` attendono i dati
- `DelayQueue` è una queue bloccante di elementi ritardati, dove un elemento può essere preso solo dopo la scadenza del ritardo
- `SynchronousQueue` è una coda bloccante in cui ogni inserimento attende una rimozione da un altro thread

Chiedo scusa per la poca chiarezza di quest'ultima subsection. Non ci si capisce un cazzo.

13 J13 - Sockets

I **socket** forniscono un'interfaccia per programmare reti sul layer di trasporto, con un utilizzo simile all'I/O da file. L'utilizzo è indipendente dal linguaggio di programmazione. Distinguiamo tra TCP e UDP, con le loro caratteristiche: TCP è affidabile, ordinato, bidirezionale, UDP *è una merda.* Il comportamento normale di un server con TCP è il seguente: viene eseguito su un computer specifico, ha un socket collegato ad una porta specifica, rimane in attesa di richieste di connessione, le accetta, e crea un nuovo socket per la connessione su un'altra porta. In Java, per server/client, è sufficiente creare un socket, creare gli streams I/O, comunicare, chiudere il socket.

13.1 Socket su UDP

I socket su UDP non necessitano di un *socket di benvenuto*, non hanno streams ma inviano pacchetti contenenti byte da estrarre.

13.2 Oggetti

Per inviare oggetti è sufficiente utilizzare degli ObjectInputStream/ObjectOutputStream.

13.3 Eccezioni

Vi sono vari tipi di eccezioni possibili:

- `IOException` nell'apertura dello stream o del socket
- `UnknownHostException` nell'apertura di un socket sul client
- `SecurityException` quando è presente un security manager
- `ClassNotFoundException` leggendo un oggetto

Glossario

FIFO con logica First In, First Out. 13

LIFO con logica Last In, First Out. 11

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

mock da Wikipedia: *"Nella programmazione orientata agli oggetti, i mock object (simulati o mock object) sono degli oggetti simulati che riproducono il comportamento degli oggetti reali in modo controllato. Un programmatore crea un oggetto mock per testare il comportamento di altri oggetti, reali, ma legati ad un oggetto inaccessibile o non implementato. Allora quest'ultimo verrà sostituito da un mock."*. 14

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

statement istruzioni del codice e.g. una riga di codice. 14