Programmazione Orientata agli Oggetti

Appunti delle Lezioni di Programmazione Orientata agli Oggetti

Anno Accademico: 2023/24

Giacomo Sturm

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"

Sorgente del file LaTeX disponibile al seguente link:
https://github.com/00Darxk/Programmazione-Orientata-agli-Oggetti

Indice

1 Introduzione al Linguaggio Java

Java venne introdotto nel 1995, progettato per programmare dispositivi embedded, ma velocemente si utilizzò per scopi diversi da quelli descritti dai suoi creatori. Il team di Java sviluppò un sistema di embedded per i browser, le applet, delle piccole applicazioni che potevano essere eseguite all'interno del browser. Questo fu uno dei primi motivi per il successo di Java, un'altro aspetto importante per la diffusione di Java fu la sua caratteristica di semplificare aspetti di C e C++. Utilizzando un linguaggio più semplice diminuiva il costo associato ad un programmatore, per cui l'adozione di Java venne trainata dalle aziende alla ricerca di abbassare gli stipendi dei suoi dipendenti. In seguito venne utilizzato per creare la macchina virtuale su cui operano tutti i dispositivi Android.

Java è stato ideato con la capacità di mantenere la retro-compatibilità, nonostante le spinte di innovazione tecnologica che il linguaggio ha sostenuto negli anni. Nonostante la sua età Java è uno dei linguaggi di programmazione più usati dalle aziende. La diffusione più importante di Java si ebbe tra il '95 ed il '98.

Java fu il primo progetto industriale che introdusse un nuovo concetto di portabilità, un processore virtuale, poiché non viene progettato per un calcolatore fisico, ma la piattaforma "Java Virtual Machine", JVM. Uno stesso programma Java può essere eseguito su tutte le piattaforme senza dover ricompilare nuovamente il codice per quella specifica piattaforma. Questa rappresenta una rivoluzione in ambito industriale, il codice oggetto Java "bytecode" prodotto dalla compilazione, come un file di estensione `.class`, può quindi essere eseguito da qualsiasi piattaforma che dispone di un'implementazione della JVM. Invece programmi creati da un linguaggio più vecchio come C, producono un codice oggetto che presenta istruzioni macchina uniche alla piattaforma in cui è stato compilato. Astraendo il processore fisico si è reso possibile la creazione di programmi che non devono essere ricompilati per ogni piattaforma su cui deve essere eseguito.

Gli applet realizzati tramite Java, venivano eseguiti dalla JVM, e permise la creazione di primi siti web, rivoluzionando il paradigma di programmazione dell'epoca. La sintassi e la semantica del linguaggio sono descritte in un documento noto come "Java Language Specification" o JLS, per risolvere eventuali ambiguità. Inoltre fu uno dei primi linguaggi ad aver introdotto librerie incluse insieme al compilatore, sulla piattaforma Java.

1.1 Rapporto con il Linguaggio C

Il linguaggio C viene descritto come un linguaggio di programmazione di medio livello, di alto livello rispetto all'assembly, e permette di realizzare programmi ad alte prestazioni, utilizzato quindi per scrivere sistemi operativi. Java invece venne introdotto in un ambiente diverso rispetto al linguaggio C, molto più evoluto dove le prestazioni non rappresentavano l'unico criterio di valutazione, grazie all'aumento delle prestazioni dei calcolatori. Questo permise di realizzare linguaggi, ad alto livello di astrazione, che non avessero come scopo principale il risparmiare risorse del sistema. Non è un linguaggio che cerca le migliori prestazioni, ma cerca di creare programmi semplici e portabili. Il linguaggio C++ permette di avere prestazioni simili al linguaggio C, ma introducendo paradigmi introdotti in Java. Ma si preferisce per realizzare programmi ad alte prestazioni il C, rispetto al C++, poiché rimane un linguaggio molto complesso, complicando l'implementazione e l'uso del linguaggio.

In Java una dichiarazione di una variabile comprende la sua inizializzazione ad un valore nullo, al contrario di C. Le dichiarazioni e assegnazioni in Java si ottengono mediante la stessa sintassi del linguaggio C. Si introduce un tipo `boolean` utilizzato per rappresentare i due possibili valori booleani `true` e `false`. Le descrizioni di tutti i tipi utilizzati in Java vengono definite nel JLS. Entrambi sono due linguaggi staticamente tipati, ovvero a tempo di compilazione deve essere noto il tipo di ogni dato utilizzato nel programma. Java introduce il tipo `String` utilizzato per rappresentare stringhe, non è un dato primitivo, e permette l'uso molto più semplice delle stringhe rispetto a C. Rappresenta un tipo particolare di classe, appoggiato dal compilatore per migliorare le sue prestazioni. Le stringhe vengono inizializzate a letterali stringhe, contenuti tra due doppi apici `" "`. Da Java 13 è possibile scrivere stringhe letterali multi-linea, utilizzando tripli doppi apici `""" """`. L'uso molto più semplificato delle stringhe in Java rispetto a C rappresenta uno dei motivi principali per cui è stato possibile convertire molti programmatori dall'uso di C a Java dopo la sua introduzione.

La sintassi di controlli di flusso è esattamente la stessa del linguaggio C, ma le condizioni sono diverse a causa del tipo booleano.

Una grande differenza tra i due linguaggi consiste nella diversa diagnostica a tempo di compilazione e di esecuzione degli errori, poiché in Java gli errori ed i messaggi di errore forniscono informazioni molto utili per la sua risoluzione. Gli errori a tempo di compilazione sono altrettanto efficaci in Java quanto in C, ma la differenza principale consiste negli errori a tempo di esecuzione tra i due linguaggi, poiché in C, su molte piattaforme, viene fornito un errore estremamente generico, in base alla versione di C, del compilatore, e da altre condizioni. Mentre in Java esistono numerosi errori, chiamati eccezioni per definire ogni possibile tipo di errore.

1.2 Paradigma Orientato agli Oggetti

Il paradigma procedurale, usato nel linguaggio C, separa in maniera netta il codice e le operazioni, ovvero la memoria e ed il processore, seguendo l'architettura definita da von Neumann. Anche se rappresenta il modo più naturale per scrivere programmi, nell'industria è stato ampiamente superato a favore del paradigma orientato agli oggetti. Questo paradigma segue la filosofia secondo cui le operazioni e lo stato sono connesse tra di loro, accomunate utilizzando "oggetti". Si è quindi rimossa la divisione imposta sui vecchi linguaggi di programmazione, che seguono l'architettura reale di un calcolatore. Per cui un problema è più facilmente modellabile se viene diviso in una pluralità di oggetti e classi di oggetti, contenenti lo stato e le operazione eseguibili su quell'oggetto. Questi oggetti si scambiano informazioni tra di loro, invocando particolari comandi; in questo modo gli oggetti conoscono ed interagiscono con altri esemplari, anche dello stesso tipo, o classe. Per conoscersi, questi oggetti contengono riferimenti agli altri oggetti.

Questo modo di interpretare un problema è vicino al nostro modo di pensare, il che aiuta nella creazione di programmi utilizzando linguaggi orientati agli oggetti. Utilizzando un linguaggio come C, progettato per il paradigma procedurale, è possibile programmare seguendo il paradigma orientato agli oggetti, ma non essendo stato creato con questo scopo, la realizzazione di programmi utilizzando questo paradigma è notevolmente più complesso.

L'esecuzione di un programma è uno scambio di messaggi tra questi oggetti connessi tra di loro tramite riferimenti interni, creando una rete di oggetti, modificando lo stato degli oggetti coinvolti

nelle operazioni. Si può rappresentare la rete di oggetti tramite un diagramma di oggetti, indicati come rettangoli contenenti diversi campi, ed eventuali riferimenti ad altri oggetti, rappresentati come archi orientati, diretti verso altri oggetti.

Ogni oggetto contiene lo stato, un comportamento, le operazioni che offre ed un identità, per differenziare diversi esemplari della stessa classe di oggetti. Si è dimostrato nel tempo come questo paradigma permette di scrivere e mantenere il codice molto più efficiente rispetto ad altri paradigmi. Gli oggetti vengono definiti tramite classi, rappresentate come file diversi del programma, contenenti lo stato, e la definizione dei metodi. Ogni oggetto creato da una classe possiede un'identità unica, e presenta lo stesso stato e le operazioni della classe di appartenenza. Le classi sono quindi delle istruzioni di montaggio, o fabbriche che producono oggetti.

Per studiare questo paradigma, si utilizzeranno esempi di forme geometriche, come studio di caso più semplice e limitato, ed uno studio di caso più esteso, mirato al refactoring, chiamato "diadia".

In Java ogni file contenente classi deve essere salvato con lo stesso nome della classe contenente, in un file di estensione ".java". Le operazioni di oggetti in Java si chiamano metodi, invece di funzioni come nel linguaggio C, poiché rappresentano operazioni nel contesto di una classe, mentre le funzioni possono operare al di fuori di una classe.

1.3 Oggetti e Riferimenti

Ogni classe viene definita utilizzando la parola chiave `class` seguito dal nome della classe, il corpo di questa contiene le definizioni dei metodi e delle variabili di istanza della stessa, che memorizzano le informazioni dello stato di un oggetto di questo tipo. Nella definizione della classe viene usato il modificatore di visibilità `public`, analogamente ai metodi ed alle variabili di istanza:

```
public class <nomeClasse> {
    // definizioni delle variabili e dei metodi
}
```

Per cui ogni definizione utilizza un modificatore di visibilità seguendo la sintassi: modificatore di visibilità, tipo, nome della variabile o metodo o classe. Questo modificatore di visibilità verrà trattato più approfonditamente in una sezione successiva, ma in generale descrive se una certa variabile, metodo o classe è visibile all'esterno del file stesso dove viene dichiarata.

Per iniziare l'esecuzione di un programma viene eseguito il metodo `main()`, ereditato da C, invocato sempre per primo dalla JVM, da cui quindi parte l'esecuzione di ogni altro metodo utilizzato nell'esecuzione. Quindi eccetto questo metodo, per ogni altra invocazione di un metodo esiste sempre un metodo invocante ed un metodo invocato.

Si considera un esempio semplice di una classe chiamata `Punto`, che rappresenta un punto in un piano di coordinate cartesiane bidimensionali:

```
public class Punto{
    // stato dell'oggetto
    private int x;
    private int y;
```

```
    // metodi propri della classe
    public void setX(int posX){
        this.x = posX;
    }
    public void setY(int posY){
        this.y = posY;
    }
    public int getX(){
        return x;
    }
    public int getY(){
        return y;
    }
}
```

Le variabili della classe `Punto` `x` e `y`, vengono chiamate variabili d'istanza. La creazione di un oggetto di una classe si ottiene tramite l'operatore `new`, secondo la seguente sintassi:

```
// creazione ed assegnazione di un nuovo oggetto Punto tramite riferimento
Punto origine = new Punto();
```

Questa riga di codice invoca un costruttore tramite l'operatore `new`, restituisce un riferimento ad un oggetto del tipo `Punto` appena creato, e viene conservato con un'assegnazione ad una variabile locale, in questo caso chiamata `origine`.

Tramite questo riferimenti è possibile accedere ai metodi, ed eventualmente allo stato dell'oggetto, tramite la notazione puntata, che segue la sintassi `<rif-obj>.<metodo>(<par-attuali>);`:

```
origine.setX(0); // assegna lo stato alla variabile x
origine.setY(0); // assegna lo stato alla variabile y
```

Questa notazione è molto efficace nell'esprimere la vicinanza tra i dati e le operazioni, per cui viene ampiamente usata in tutti i linguaggi OO (Orientati agli Oggetti). Ogni nuovo oggetto creato viene salvato in memoria, ed è possibile creare nuovi oggetti fino all'esaurimento della memoria virtuale. Questa classe viene quindi contenuta in un file `Punto.java`.

Una variabile contenente un riferimento ad un oggetto contiene una sequenza di caratteri che rappresenta l'indirizzo in memoria virtuale dell'oggetto, anche se rappresenta una semplificazione, è utile per analizzare il suo comportamento. Mentre per accedere ai valori contenuti, ed assegnare valori alle variabili interne all'oggetto, bisogna utilizzare metodi "getter" e "setter", descritti nella classe dell'oggetto, poiché le variabili di istanza sono accessibili solamente all'interno della classe.

Quando viene chiamato un metodo passando una variabile, viene passato per valore.

In Java il riferimento nullo è un unicamente il letterale di tipo riferimento all'oggetto `null`. Ogni riferimento ad oggetto dichiarato, senza essere inizializzato viene assegnato al riferimento nullo. Indica l'assenza di un riferimento reale ad un oggetto esistente, al contrario del riferimento nullo in C, non corrisponde all'intero 0. Invocando un metodo su un riferimento ad un oggetto nullo si verifica un errore, a tempo di esecuzione, si indica in Java come eccezione: "runtime-exception"; in questo caso: `NullPointerException`.

Nel corpo dei metodi, se il nome di un parametro formale coincide al nome di una variabile di istanza, il primo ha precedenza, lo offusca. Si verifica l'effetto di "shadowing", per cui vengono effettuate operazioni non sulla variabile di istanza, ma sul parametro formale. Per identificare le variabile di istanza si utilizza la parola chiave `this`, in notazione puntata seguita dal nome della variabile di interesse. In questo modo si possono utilizzare come nomi di parametri formali gli stessi nomi di una variabile di istanza, senza problemi. La parola chiave `this` rappresenta infatti un riferimento all'oggetto corrente, anche se viene usato per accedere alle variabili di istanza, può essere omesso in assenza di ambiguità con i parametri formali. Viene comunque favorito il suo utilizzo per aumentare la leggibilità del codice ed evitare eventuali fenomeni di shadowing. Può essere inoltre utilizzato per invocare metodi propri della classe, all'interno di altri metodi della stessa.

1.4 Gestione della Memoria

In Java la creazione di oggetti, e quindi la loro allocazione in memoria, avviene tramite l'operatore `new`. Questo operatore richiede la specifica della classe di cui si vuole creare una nuova istanza, e richiede uno dei costruttori della classe. Un costruttore è un metodo che presenta esattamente lo stesso nome della classe e non restituisce niente, e generalmente inizializza tutte le variabili di istanza della classe, ma non sono dei metodi per cui non devono specificare il tipo del valore restituito, neanche utilizzando `void`.

```
public class Punto{
    private int x;
    private int y;
    // costruttore "no-args" della classe "Punto"
    public Punto(){
        this.x = 0;
        this.y = 0;
    }
}
```

Questo tipo di costruttori non prendono parametri, ma è possibile definire costruttori che accettano degli input. Ogni classe presenta sempre almeno un costruttore, e costruisce lo stato iniziale dell'oggetto, per cui è possibile inizializzare l'oggetto ad un certo stato fornito alla chiamata del costruttore:

```
public class Punto{
    private int x;
    private int y;
    // costruttore della classe "Punto"
    public Punto(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

L'oggetto è stato creato, e lo stato inizializzato, solamente al termine dell'esecuzione del costruttore, durante la sua esecuzione lo stato dell'oggetto è inconsistente.

Poiché una classe può contenere più di un costruttore, questi vengono distinti in base agli argomenti che prendono. Se invece non viene dichiarato alcune costruttore all'interno di una classe, è possibile creare un nuovo oggetto usando la stessa terminologia, e tutti i campi numerici vengono inizializzati a zero, i riferimenti a null, e le stringhe alla stringa vuota. Per cui il compilatore crea automaticamente un compilatore "no-args" con le inizializzazioni di default, ma se viene dichiarato un costruttore con parametri all'interno di una classe, e si prova a creare un oggetto della classe usando un costruttore no-args il compilatore produce un errore. Quindi il compilatore genera un costruttore implicito solamente se non è dichiarato alcun costruttore dentro una classe. Spesso si definiscono diversi costruttore per inizializzare uno oggetto della stessa classe in diversi stati noti.

Durante l'esecuzione la JVM ha accesso a due aree di memoria, lo "Stack" e l'"Heap". Lo stack contiene le informazioni necessarie all'esecuzione dei metodi, lo stato dell'esecuzione e le variabili locali ed il loro valore. Mentre l'heap contiene gli oggetti creati tramite l'operatore `new`. All'inizio viene assegnato lo stack alla JVM, viene utilizzata per conservare ciò che è necessario a mantenere lo stato dell'esecuzione, è una struttura dati che implementa la disciplina LIFO, gestita dalla JVM, per memorizzare i metodi ed i Record Di Attivazione, o RDA, di questi, cominciando dal metodo `main()`. Ogni volta che termina l'esecuzione di un metodo, viene rimosso dallo stack, gestito in modo automatico e trasparente rispetto dalla JVM. Dopo essere stato rimosso un metodo dallo stack, le variabili locali ed i suoi parametri non sono più utilizzabili.

L'RDA contiene i parametri attuali, il riferimento all'oggetto corrente, i parametri locali, il valore restituito, ed il riferimento al metodo invocante. Contiene inoltre tutti gli identificatori delle variabili, più facile da ricordare rispetto all'indirizzo della variabile stessa. Il primo record di attivazione è sempre il metodo `main()`.

Poiché lo stack è di dimensione limitata, e generalmente piccola rispetto all'heap, per cui non può contenere troppi RDA, ed è possibile esaurire lo spazio, sollevando un'eccezione: `StackOverflowException`.

L'heap contiene tutti gli oggetti dichiarati con `new`, la sua dimensione quindi varia a tempo di esecuzione per accomodare tutti questi oggetti ed il loro stato. Viene assegnata dalla JVM, dove gli oggetti vengono allocati in ordine sparso, quindi non rispettano alcuna disciplina FIFO o LIFO. L'operatore `new` è quindi molto simile all'operatore `malloc` in C, l'unica differenza è che il primo invoca un costruttore.

Dati due riferimenti allo stesso oggetto, utilizzando uno dei due è possibile applicare modifiche all'oggetto, visibili anche dall'altro riferimento, questo fenomeno si chiama "Side-Effect", o effetto collaterale. Poiché gli oggetti si passano come riferimenti, l'operatore `==` confronta solamente i riferimenti all'oggetto, e bisogna utilizzare un altro metodo per poter confrontare i valori interni degli oggetti tra di loro. Per determinare l'equivalenza tra due oggetti distinti si usa il metodo `equals()`, viene offerto sempre da tutte le classi, se invece non viene espresso esplicitamente ha la stessa semantica dell'operatore `==` sui riferimenti. Poiché è un metodo interno ad una classe, in generale non è un metodo simmetrico: `a.equals(b)` $\neq$ `b.equals(a)`. Questo metodo identifica l'equivalenza tra due oggetti distinti.

La deallocazione della memoria avviene automaticamente da un metodo chiamato Garbage Collector, per cui non è necessario utilizzare esplicitamente una funzione `free()` come in C. Se questo Garbage Collector si accorge che degli oggetti rimangono appesi, li libera dalla memoria.

Il garbage collector analizza la catena di riferimenti degli oggetti per determinare se un dato oggetto è in uso, oppure non è più raggiungibile tramite una sequenza di riferimenti che parte dallo stato dell'esecuzione corrente.

1.5 Overloading, Stringhe e Array

L'"Overloading" è una caratteristica di un linguaggio di programmazione, come Java, che permette di definire più procedure con lo stesso nome, ospitati all'interno della stessa classe. Questi metodi, diversi, si distinguono solamente per il tipo, il numero di uno o più parametri formali, oppure per il loro ordine. Invece non è possibile distinguere due metodi solamente per il tipo di ritorno, in generale il tipo restituito non appartiene alla segnatura di un metodo.

```
public class Sommatore{
    public int add(int a, int b) { return a + b; }

    public int add(int a, int b, int c) { return a + b + c; }

    public double add(int a, double b) { return a + b; }

    public double add(double a, int b) { return a + b; }
}
```

Nel linguaggio C, questo non è possibile. Se c'è una corrispondenza perfetta tra i parametri passati al metodo, non si riscontrano problemi, invece se non è presente una corrispondenza perfetta tra i tipi passati al metodo, viene invocato un algoritmo di risoluzione, prima di fermare la compilazione che cerca di capire se è possibile effettuare la chiamata tramite semplici conversioni di tipo, oppure conversioni di tipo più "conservative". La promozioni di tipo può essere implicita alla chiamata di un metodo, per cui potrebbe essere complesso determinare quale metodo è stato chiamato.

Alla chiamata di un metodo sovraccaricato è il compilatore a scegliere quale dei metodi da applicare, la scelta è definitiva e viene scritta nell'eseguibile `.class`. Versioni sovraccariche di un metodo potrebbero differire anche per il metodo di ritorno, ma il tipo di ritorno non può essere usato per distinguere due metodi.

L'overloading è presente anche sull'operatore `+`, applicandolo su diversi dati, tramite somme algebriche di interi, numerali a virgola mobile; inoltre viene usato per concatenare stringhe tra di loro. In C++ ed in Scala è possibile sovraccaricare gli altri operatori, mentre non è possibile in Java, dove l'unico operatore sovraccarico è `+`, nella maniera descritta.

Costruttori definiti ripetendo codice vengono sconsigliati, anche per i metodi sovraccaricati, per cui si tende ad eleggere un costruttore al ruolo di costruttore "primario", il più generico possibile, per evitare la duplicazione del codice. In seguito gli altri costruttori vengono definiti in base a questo costruttore, invocandolo, con il metodo `this()`, specificando i parametri che si vuole modificare rispetto al costruttore primario.

```
public class Rettangolo{
    private int altezza;
```

```
    private int base;
    private Punto vertice;

    public Rettangolo(Punto v, int b, int h){
        this.vertice = v;
        this.base = b;
        this.altezza = h;
    }

    public Rettangolo(int b, int h) { this(new Punto(0,0), b, h); }

    public Rettangolo() { this(new Punto(0,0), 0, 0); }
}
```

L'overloading viene risolto a tempo di compilazione, dal compilatore, che applica una scelta definitiva, ed inserisce il metodo scelto nei file `.class` generati. Per cui quest'analisi dipende solamente dai tipi statici dei parametri attuali, nell'invocazione del metodo, a tempo statico.

In Java esiste la classe `String` per poter rappresentare sequenze di caratteri immutabili, rappresenta un riferimento ad un oggetto istanza della classe `String`, può contenere caratteri di Unicode 2.1, la versione di Unicode a 16 bit, capace di memorizzare la maggior parte dei simboli e caratteri alfabetici moderni ed ideogrammi. Per rendere il linguaggio più semplice questa classe possiede dei letterali appositi, ed è l'unico oggetto che può essere creato senza una `new` esplicita. L'operatore `==` su due stringhe verifica l'uguaglianza dei riferimenti, per cui per valutare l'equivalenza di contenuto tra due stringhe bisogna utilizzare il metodo `equals()`.

Poiché rappresenta una sequenza di caratteri immutabili, per modificare o aggiornare una stringa è necessario crearne una nuova stringa, altrimenti è possibile concatenare due stringhe utilizzando l'operatore `+`, che rappresenta un nuovo oggetto stringa, sovrascritto sulla variabile utilizzata per memorizzarla. Per ottenere la lunghezza di una stringa si utilizza il metodo `.length()`, per ottenere il carattere ad una determinata posizione si utilizza il carattere `.charAt(int i)`. Il metodo `.indexOf(char a)` fornisce la posizione di un carattere o il primo carattere di una sottostringa, per cui è un metodo sovraccarico, altrimenti restituisce -1. Per aggiornare una stringa si utilizza il metodo `.replace(Stringa s1, Stringa s2)`, che restituisce un nuovo oggetto dove è stata la sottostringa `s1` con la sottostringa `s2`.

Se viene eseguito il metodo `println()` su un riferimento stampa il valore del riferimento, non il riferimento stesso. Definendo il metodo `toString()` è possibile specificare cosa si vuole visualizzato quando viene chiamato il metodo di stampa a schermo. Un oggetto che possiede questo metodo può essere concatenato con una stringa, questo concatena la stringa con la string restituita dal metodo `toString()`. Se non viene definito questo metodo, il suo comportamento standard implicito stampa l'indirizzo di memoria dell'oggetto, come per il metodo `equals()`.

Un array è una struttura dati che memorizza dati omogenei, per dichiarare un array si inserisce accanto al tipo oppure al nome della variabile `[]`, si preferisce la prima per leggibilità. Sono oggetti per cui devono essere dichiarati con il metodo `new`. Le operazioni di accesso e modifica di un array sono le medesime del linguaggio C, per ottenere la lunghezza di un array si utilizza il metodo

`.length()`. Da Java 5 è stato introdotto un modo per poter iterare su un array chiamato "for-each", senza gestire esplicitamente l'indice di iterazione:

```
for(<tipo> <nome-variabile> : <nome-array>)
```

In questo modo la variabile `<nome-variabile>` itera su ogni elemento dell'array `<nome-array>` dello stesso tipo `<tipo>`.

Per realizzare una costante si può utilizzare la parola chiave `final`, per convenzione gli identificatori di una costante si scrivono in maiuscolo. Per i tipi primitivi rende costante la variabile, per gli oggetti rende costante il riferimento, ma non il suo contenuto. Invece per realizzare una variabile condivisa su tutti gli oggetti della stessa classe si utilizza la parola chiave `static`.

2 Qualità del Codice

2.1 Librerie

Come ogni linguaggio di programmazione, relativamente recente, Java contiene numerose librerie specializzate per risolvere diversi problemi. Per cui è necessario utilizzare in maniera efficace ed efficiente queste librerie, composte da classi. Bisogna quindi conoscere per nome le classi più importanti di queste librerie, e sapere come cercare ed usare altre classi. Non è necessario conoscere l'implementazione di una classe, ma solo l'interfaccia o API fornito. Queste informazioni vengono fornite tramite la documentazione associata ad ogni libreria, in formato HTML `javadoc`, accessibili tramite un browser, oppure integrate all'interno dell'IDE. Per ogni classe viene fornito il suo nome, una descrizione generale della classe, dello scopo dei suoi costruttori e metodi, e valori di ritorno per i suoi costruttori e metodi. Non sono presenti invece tutti i campi e metodi privati, poiché non sono accessibili dall'esterno, né il suo codice, non necessario per il suo utilizzo.

In caso si voglia creare una nuova classe, bisogna documentarla allo stesso modo delle classi fornite dalle librerie, per permettere il loro utilizzo, senza conoscere in dettaglio la sua implementazione. La documentazione di una classe dovrebbe includere il nome, un commento che descrive la caratteristiche generali e lo scopo della classe, la versione, il nome dei metodi e dei costruttori, una descrizione del suo scopo, i parametri di ritorno e passati al costruttore o metodo. La documentazione può essere generata automaticamente tramite un'utility chiamata `javadoc`, contenuta in Eclipse, i marcatori utilizzati per definire i comandi si trovano esclusivamente all'interno di un commento generato del genere `/** <comandi> */`:

```
/**
 * Nome-classe: breve descrizione della classe ed il suo scopo
 *
 * @author      autore della classe
 * @see         riferimento ad altre classi
 * @version     versione corrente della classe
 */
```

I commenti per un metodo o un costruttore vengono inseriti prima subito prima del metodo o costruttore stesso.

```
/**
 * Commento che descrive scopo e caratteristiche del metodo o costruttore
 *
 * @param nome-parametro    breve descrizione
 * @return                  valore di ritorno
 */
```

La classi vengono raggruppate in `package`, per mantenere insieme classi concettualmente e logicamente correlate. Permette di creare spazi di nomi per evitare conflitti, in oltre permette di definire domini di protezione. Una classe può accedere a tutte le classi presenti nello stesso `package`, altrimenti per accedere a classi pubbliche di un altro pacchetto si può utilizzare il nome completo,

anteponendo il nome del pacchetto, oppure importando la classe ed utilizzando direttamente il nome della classe. Le classi devono dichiarare la propria appartenenza ad un pacchetto tramite la dichiarazione `package <nome-package>`, all'inizio del file. Una classe può appartenere a più di un pacchetto. Ogni pacchetto deve avere un nome univoco, di solito il nome comprende il nome del dominio Internet dell'organizzazione in ordine inverso. Una classe può essere usata al di fuori del pacchetto solamente se viene dichiarata pubblica, tramite un modificatore d'accesso `public`, se viene omesso questo modificatore, è accessibile solamente alle classi interne allo stesso pacchetto.

Il nome di un pacchetto possiede una struttura gerarchica, questa struttura deve avere una corrispondenza diretta nel file system.

2.2 Coesione e Accoppiamento

Il software evolve continuamente, viene modificato in continuazione, per estensioni, correzioni, mantenimento, etc., e se il costo di questa evoluzione è troppo elevato, il software viene gettato e viene implementato da zero. Questa evoluzione viene effettuata in tempi diversi da molte persone. Se il codice è di cattiva qualità, la sua manutenzione ha un costo relativamente alto. La qualità del codice dipende da due fattori importanti, la coesione e l'accoppiamento. L'accoppiamento tra due o più unità di un programma rappresenta l'impossibilità di modificare una sola unità senza l'eventuale modifica di altre unità accoppiate ad essa. Per cui si vuole evitare l'accoppiamento, per limitare il più possibile la modifica di ulteriori unità, poiché aumenterebbe esponenzialmente i costi di una singola modifica. Un sintomo del forte accoppiamento è la duplicazione del codice all'interno di un progetto. Un basso accoppiamento permette di capire il codice di una classe senza leggere i dettagli di tutte le altri a lei accoppiata, e quindi anche la modifica, senza modificare ulteriori unità.

La coesione si riferisce al numero e all'eterogeneità dei compiti di cui una singola classe è responsabile. Se ogni unità è responsabile di un singolo compito, allora possiede un'elevata coesione. La coesione si applica alle classi, ai metodi ed anche ai pacchetti. Un'alta coesione permette di definire moduli aventi uno scopo preciso ed un compito ben definito. Questo favorisce la comprensione dei compiti di una classe, l'utilizzo di nomi appropriati, ed il riuso di classi e di metodi, rendendo la manutenzione meno costosa. Alta coesione e basso accoppiamento rappresentano due caratteristiche dello stesso concetto. Un'alta coesione di una classe si ottiene mantenendo lo scarso accoppiamento rispetto alle altre classi, analogamente lo scarso accoppiamento di una classe verso altre si ottiene tramite un'alta coesione.

Per cui quando viene progettato il codice è utile pensare a quali cambiamenti futuri potranno essere richiesti, e come verrà usata la classe.

Quando si vuole modificare il codice, avere basso accoppiamento ed alta coesione, porta a modificare un numero ristretto di classi, per cui si mira a dover modificare il minor numero possibile di classi.

Viene distinta la qualità esterna ed interna di un codice, sulla base degli utilizzatori del codice, esterna, in base ai suoi requisiti, e degli sviluppatori del codice, interna, per valutar la manutenibilità del codice.

La manutenzione del codice spesso richiede l'aggiunta di nuovo codice, questo eventualmente degrada il codice nella sua totalità, aumentando l'accoppiamento e diminuendo la coesione, portando ad un'eventuale riorganizzazione del codice. Questa operazione viene chiamata "refactoring", senza

modificare la funzionalità del codice. Per mantenere la correttezza del codice viene associato a dei test che stabiliscono i suoi criteri di correttezza, usati per valutare la correttezza di implementazioni interne diverse.

In generale un metodo è troppo lungo se è responsabile di più di un compito logico, mentre una classe è troppo lunga se comprende più di un concetto.

2.3 Tecniche di Testing

In generale i primi errori in un programma sono errori di sintassi, indicati dal compilatore in modo preciso e completo, anche se le specifiche di un messaggio di errore dipendono dal compilatore utilizzato, successivamente possono presentarsi errori logici o "bug", su cui il compilatore non è in grado di fornire indicazioni. Alcuni errori logici non si presentano immediatamente, per la complessità del software, altri possono essere individuati a tempo di esecuzione, e la macchina virtuale fornisce informazioni precise sulla natura dell'errore, la maggior parte delle volte si tratta di un'eccezione del tipo `NullPointerException`.

Un programma rappresenta una singola descrizione statica associata a molteplici possibili esecuzioni dinamiche. Il compilatore è in grado di individuare errori a livello statico, a tempo di scrittura o compilazione, e non è in grado di individuare possibili errori dovuti all'esecuzione e l'evoluzione del programma. I bug sono quindi errori che si presentano durante l'evoluzione dinamica del programma, che il compilatore non è in grado di prevedere. Il costo delle operazioni di debugging, rappresenta il costo principale di ogni moderno progetto di software, ed è interamente a carico del programmatore.

La correzione di un bug dipende da due grandezze, la dimensione del contesto, ovvero il numero di linee di codice in cui il bug si può annidare. Inoltre dipende dal tempo necessario al bug per manifestarsi, rappresenta una misura temporale tra la causa del bug ed il rilevamento dei suoi effetti.

Per rilevare i malfunzionamenti di un software vengono utilizzati delle tipologie di test, in tre fasi sequenziali:

- Mettere il sistema in uno stato iniziale noto;
- Iniziare a sollecitare il sistema;
- Controllare lo stato del sistema e confrontarlo con lo stato atteso.

Se i test sono progettati e mantenuti, allora permettono di individuare tempestivamente i bug all'interno del software, restringendo le cause del singolo bug. Se il test ha successo si ha una garanzia sul comportamento dinamico del codice, altrimenti fornisce informazioni sulla presenza di un bug. Ad ogni modifica è necessario ripetere un'operazione di testing, per identificare eventuali bug, inseriti durante la manutenzione del codice, in questo modo è possibile ricercare localmente l'errore, in maniera molto economica. Inoltre utilizzando questo processo è possibile prevenire la regressione, localizzando l'errore nel codice appena aggiunto al programma.

Per automatizzare questo processo si realizza, accanto al codice di produzione, il codice di test. Una possibile soluzione inizializza tutti i casi da testare sul codice di produzione, inserendo, ad ogni chiamata, il valore atteso confrontandolo con il valore ottenuto. I test devono essere

automatici, efficienti ed isolati, per mantenere la località degli errori, inoltre devono essere separati dal codice applicativo ed eseguibili e verificabili separatamente. Esistono vari strumenti per assistere il programmatore nel testing, in particolare l'unit-testing, in Java il più note ed utilizzato framework è "JUnit", integrato all'interno di Eclipse. Per ogni classe creata si crea una classe parallela dedicata al testing, contenente una batteria di test, ognuno un metodo differente per testare diversi casi, preceduto dall'annotazione `@Test`, contenente un'asserzione sul risultato aspettato tramite il metodo `assertEquals()`, e prende come parametri i due oggetti da confrontare tipati `Object`.

Tutte le classi di test hanno la stessa struttura, si inseriscono nello stesso pacchetto della classe che devono testare, e vengono chiamate rispettando la convenzione `test<nome-classe>`, anche se non è più necessario utilizzare questa notazione, semplicemente tramite l'annotazione `@Test`, anche se viene favorito il suo utilizzo. I risultati attesti sono documentati tramite asserzioni esplicite, non mediante stampe, se l'asserzione è vera il test è andato a buon fine, altrimenti è presente un errore nella sezione di codice testato. Sono possibili altre asserzioni fornite da JUnit, sovraccariche e facilmente intercambiabili, per cui è sempre favorito utilizzare la versione più pertinente.

Una variante di `assertEquals()`, prevede una stringa, da stampare per fornire informazioni sul risultato del test solo in caso di fallimento, oltre ai due riferimenti ad oggetti da comparare.

Per compilare i test bisogna includere le librerie della versione di JUnit utilizzata nel classpath del progetto. JUnit è così popolare che viene incluso ed è fortemente integrato negli IDE moderni, come Eclipse. Questi favoriscono la creazione e l'uso dei test. In Eclipse un test eseguito può essere identificato da barre colorate, verde se il test ha avuto successo, rosso se il test è fallito sollevando un'eccezione o blu se il test ha fallito l'asserzione. Per favorire l'utilizzo dei test è possibile collocare le classi test in una cartella sorgente parallela a `src` nella struttura delle classi. In questo modo si crea una copia del codice di produzione, adibita esclusivamente alla gestione delle classi di test.

Per facilitare la scrittura di tutti i metodi di test, è comodo utilizzare degli oggetti predisposti per l'utilizzo da parte di tutti i test-case operazione chiamata Fixture. JUnit permette di confinare in un unico metodo `setUp()` la creazione di tutti gli stati noti a priori tramite l'annotazione `@BeforeEach`. In questo modo i test diventano estremamente semplici. In generale per non aumentare l'accoppiamento delle classi di test, è consigliabile includere nel metodo `setUp` la creazione di oggetti utilizzati da almeno due test-case distinti. Altrimenti è preferibile mantenere la creazione di oggetti nel singolo test, per mantenere il test isolato, e diminuire l'accoppiamento nella classe.

La lunghezza ottimale di un singolo test-case consiste di una singola riga di codice, rappresentata dall'asserzione. Per favorire la semplicità dei test si può fattorizzare il codice di creazione della fixture. L'uso di test minimali favorisce la leggibilità e la ricerca di errori, ed è sempre conveniente partire da test minimali per individuare i bug minori, per poi aumentare la complessità dei test ed individuare bug sempre più complessi.

In generale è buona norma scrivere i test prima ancora di aver scritto il programma, una tecnica chiamata "test-driven-develompent". Inoltre è sempre conveniente scrivere esattamente cosa verifica il test nel suo nome, per evitare di creare confusione, e per facilitare ancora di più il suo utilizzo, evitando di controllare il suo corpo in caso di fallimento del test.

3 Polimorfismo

3.1 Interfacce

In Java i riferimenti sono tipati, ovvero specificano il tipo dell'oggetto referenziato, quindi attraverso un riferimento possono essere eseguiti tutti i metodi offerti dalla classe dell'oggetto. In generale nei linguaggi orientati agli oggetti esiste più di un modo per poter creare un tipo, in Java oltre al costrutto `class` esiste il costrutto `interface`, utilizzato per creare un nuovo tipo. Questo costrutto specifica un tipo in termini dei metodi che può offrire, specificando solamente la segnatura ed il tipo di ritorno, per cui non è presente alcun dettaglio implementativo, come le variabili, costruttori ed il corpo dei metodi. Una classe può implementare una o più `interface` tramite il costrutto `implements <nome-interface>`, e deve contenere tutti i metodi interni contenuti dall'`interface`, ma può contenete non solo questi metodi. Una singola classe può implementare più di un'interfaccia, può assumere quindi più di un singolo ruolo, associato ad ogni singola interfaccia.

Una classe che implementa un'interfaccia è un suo sottotipo, mentre l'interfaccia è un supertipo, una sua generalizzazione. In Java vale il principio di sostituzione di Liskov, il quale afferma che un sottotipo può essere usato al posto di un suo supertipo.

Per il principio di sostituzione, un riferimento ad un sottotipo può essere assegnato ad un riferimento ad un suo supertipo. La promozione di un tipo ad un suo supertipo viene chiamata "upcasting". Il collegamento tra l'assegnatura e l'implementazione di un'interfaccia non è nota a tempo di compilazione, per cui si chiama "late binding". Quindi si può differenziare tra il tipo statico, definito a tempo di compilazione, ed il tipo dinamico, ciò che viene utilizzato a tempo di esecuzione. Il compilatore permette di applicare solo i metodi del tipo statico, poiché essendo legato solamente a tempo di esecuzione, il compilatore non permette di utilizzare i metodi del tipo dinamico. Per cui avendo un riferimento tipato staticamente ad un'interfaccia, il metodo da eseguire non è noto a tempo di compilazione, è il suo tipo dinamico che determina il metodo che verrà eseguito. Questo tipo di comportamento viene chiamato polimorfo, poiché sono possibili diverse forme o comportamenti diversi per tutti i suoi sottotipi.

L'overloading dei metodi avviene a tempo di compilazione, quindi staticamente, ma in questo modo non tiene conto dei metodi dei tipi dinamici. Il tipo di dichiarato di una variabile è il suo tipo statico, mentre il tipo dell'oggetto referenziato è il suo tipo dinamico. Il compilatore in generale non conosce non può neanche prevedere i tipi dinamici.

Per cui per utilizzare i metodi forniti da un certo tipo, può essere necessario effettuare un "downcast", da un supertipo ad un suo sottotipo per permettere al compilatore di utilizzare i metodi offerti da questa classe. In questo modo si forniscono informazioni al compilatore "forzando" un certo tipo statico. Ma questa operazione eseguita a tempo statico non necessariamente è lecita, ovvero non è possibile controllare a tempo statico che il sottotipo assegnato sia un tipo valido. La macchina virtuale controlla quindi a tempo di esecuzione che questa operazione sia lecita e possibile. Altrimenti viene sollevata l'eccezione `ClassCastException`.

Le invocazioni polimorfe si risolvono a tempo di esecuzione sulla base del tipo dinamico da parte della macchina virtuale, mentre il sovraccarico di un metodo viene risolto a tempo di compilazione dal compilatore, sulla base del tipo statico dei parametri passati al metodo.

3.2 Estensioni e Overriding

Date due interfacce diverse, aventi gli stessi metodi, anche se concettualmente sembrerebbero uguali, non è possibile contenere un riferimento ad un oggetto che ne implementa una, in un tipo che ne implementa l'altra. Per risolvere questo problema viene fornita la parola chiave `extends <nome-interfaccia>` per "estendere" un'interfaccia e quindi specificare un rapporto di supertipo e sottotipo tra le due, aggiungendo dei metodi non presenti nella prima interfaccia. A tempo di compilazione è quindi possibile referenziare due oggetti che implementano queste due interfacce diverse, applicando il principio di sostituzione. Attraverso quest'estensione è possibile quindi definire nuovi tipi a partire da tipi già esistenti.

Oltre alle interfacce, è possibile estendere le classi, ma questo è un meccanismo più complesso, poiché entrambe le classi, supertipo e sottotipo, contengono metodi con un corpo, a differenza delle interfacce. La classe di partenza viene chiamata superclasse, o classe base o genitore. La classe definita per estensione da questa, viene chiamata classe estesa, derivata, sottoclasse o figlia, quest'ultima può essere estesa creando una "gerarchia" di classi. La classe derivata "eredita" tutto quello che offre la classe di partenza. Tramite l'annotazione `@Override` è possibile modificare un metodo con corpo della superclasse, all'interno della classe derivata. Per indicare la definizione di una classe come estensione di una classe si usa la stessa parola chiave `extends <nome-classe>`. Queste classi derivata possono dichiarare metodi e campi aggiuntivi, e modificare tramite annotazioni metodi preesistenti. La classe base viene considerata un supertipo della classe estesa. Poiché rappresenta una classe esterna, non può accedere a parametri privati, interni alla classe base. Per accedere a queste variabili è possibile utilizzare i metodi getter e setter offerti dalla classe base, applicati sulla classe derivata. Come per le interfacce, non è possibile invocare i metodi privati, di una classe estesa, applicato sulla sua superclasse. Il meccanismo di modifica di un metodo presente nella classe base si chiama "overriding", tramite annotazioni, come descritto precedentemente. Ma bisogna considerare che non può accedere a parti private della superclasse.

Per cui se dentro la sottoclasse si vuole accedere ai campi della superclasse bisogna utilizzare l'interfaccia pubblica di quest'ultima, come ogni altra classe esterna. Un'istanza della classe estesa può essere usata al posto di un'istanza della classe base, manifestando il polimorfismo, quindi la scelta dell'implementazione dei metodi avviene sempre a tempo dinamico.

Per creare un'istanza della sottoclasse è necessario invocare la superclasse corrispondente. Il costruttore di una classe estesa deve quindi inizializzare i propri campi, ed i campi propri della superclasse, ma un metodo di una sottoclasse non può accedere a parti private della sua superclasse. In java questo viene risolto delegando questo metodo, effettuando una chiamata al costruttore della sua superclasse, per inizializzare le sue parti private. Ciò si effettua tramite la parola chiave `super(<parametri>)`, passando i parametri formali che diventano parametri attuali del costruttore della sua superclasse. Questa invocazione deve essere l'unica e sola nel costruttore, e deve essere presente alla prima riga del costruttore della sottoclasse. Questo poiché è necessario creare la superclasse prima di poter creare la sua sottoclasse, si è quindi vincolati quest'ordine sintattico.

Se la superclasse non ha costruttori, il compilatore ne crea uno automaticamente senza argomenti, quindi è possibile che aggiungendo un costruttore della superclasse, si genera un errore di compilazione su un costruttore della sua sottoclasse, poiché utilizza il costruttore creato automaticamente dal compilatore "no-arg".

Ogni oggetto Java direttamente o indirettamente deve essere un sottotipo della classe `Object`, chiamato anche la radice della gerarchia dei tipi Java. Tutte le classi estendono automaticamente questa classe predefinita, contiene quindi alcuni metodi disponibili ad ogni classe. Alcuni di questi metodi sono `toString()` e `equals(Object o)`, tutte le classi ereditano questa implementazione, oltre la loro segnatura, possono quindi ridefinire l'implementazione, rispettando la segnatura. Il metodo `toString()` di default stampa l'indirizzo di memoria dell'oggetto su cui viene applicato il metodo, mentre il metodo `equals()` verifica l'uguaglianza dei riferimenti dell'oggetto corrente e del riferimento all'oggetto passato come parametro. Quasi sempre quindi conviene ridefinire questi metodi sulla base dello specificità della classe definita, rispettando la segnatura. Poiché nel metodo `equals()` la segnatura è di tipo `Object`, bisogna effettuare un downcast quando si vuole ridefinire il metodo, altrimenti non sarebbe possibile invocare i metodi propri della classe, poiché il compilatore può controllare solamente il tipo statico. Non è possibile cambiare la segnatura poiché non sarebbe una sovrascrittura del metodo, ma una sovraccarico. Generalmente quando si ridefinisce un metodo fornito dalla classe `Object` bisogna effettuare un downcast al tipo della classe necessaria nel metodo. Essendo tutte le classi estensioni di questa classe predefinita, ogni costruttore contiene implicitamente una chiamata al costruttore alla classe `Object`.

La parola chiave `super` è in grado di effettuare una chiamata ad un metodo presente nella superclasse, all'interno di una classe avente lo stesso nome nella sottoclasse. Impedendo di avere un invocazione ricorsiva dello stesso metodo fino all'esaurimento dello stack.

3.3 Modificatori di Accesso

In Java esiste un altro modificatore di accesso di livello intermedio tra `public` e `private`, chiamato `protected`, permette ad una superclasse aventi membri protetti di essere visibili a tutte le sue sottoclassi, indipendentemente dai pacchetti di appartenenza. In generale questo modificatore di accesso viene evitato per applicazioni semplici, come lo studio di caso trattato in questo corso, poiché rappresenta uno studio introduttivo. Infatti contraddice implicitamente il principio dell'"information hiding". L'utilizzo più opportuno è nella progettazione di framework, librerie che permettono l'estensione da parte degli utilizzatori.

Il livello di visibilità ottenuto senza inserire un modificatore di accesso viene chiamato anche `package-private`. Questo livello di visibilità è il meno permissivo dopo il modificatore `private`.

Modificatori	Classe	Pacchetto	Sottoclasse	Globale
`public`	Sì	Sì	Sì	Sì
`protected`	Sì	Sì	Sì	No
	Sì	Sì	No	No
`private`	Sì	No	No	No

In ordine di visibilità decrescente si ha quindi `public`, `protected`, `package-private` e `private`.

Solo i metodi visibili alle sottoclassi possono quindi essere sovrascritti, altrimenti se se viene sovrascritto un metodo privato della superclasse, quello nella sottoclasse nasconde l'omonimo della superclasse, ma non lo sovrascrive. Inoltre è possibile sovrascrivere la visibilità di un metodo, solamente mantenendo la stessa o ampliandola, rispettando quindi l'analogo del principio di sostituzione

per la visibilità. Sovrascrivendo un metodo è possibile modificare la sua visibilità, rendendolo più visibile, ma non è possibile diminuire la visibilità di uno metodo da una sottoclasse. Il metodo di una sottoclasse può modificare un metodo, ereditato dalla superclasse, di identica segnatura ma di tipo restituito più generale. Questa caratteristica si chiama covarianza. Mentre si può effettuare un procedimento analogo con parametri polimorfi, dove bisogna posizionare nel supertipo il parametro più specifico mentre nel sottotipo il parametro più generico. Questa caratteristica si chiama controvarianza. Entrambi tendono a generalizzasi o specializzarsi salendo o scendendo nella gerarchia delle classi. Per cui il parametro passato è controvariante, mentre il parametro restituito è covariante; scendendo nella gerarchia dei tipi, i parametro passati devono essere sempre più generali, mentre i parametri restituiti sempre più specifici.

Poiché in Java due metodi aventi lo stesso tipo rappresentano un sovraccarico, questo tipo di overload viene risolto a tempo statico, ma secondo il paradigma orientato agli oggetti questo rappresenta una sovrascrittura, nonostante siano due metodi che prendono due parametri distinti polimorfi. Quindi utilizzando la parola chiave `@Override` viene generata un eccezione a tempo di compilazione. Senza quest'annotazione, la sottoclasse dispone di due metodi distinti e sovraccarichi, la cui invocazione dipende dal tipo statico del parametro passato.

La visibilità ed il tipo di parametro sono controvarianti, mentre il tipo restituito è covariante con la gerarchia delle classi.

In Java la gerarchia delle classi ha una radice definita, è quindi `java.lang.Object`, e rappresenta la radice predefinita della gerarchia delle classi. Al contrario di altri linguaggi di programmazione, come C++, in Java, non è possibile avere una derivazione multipla tra classi, una classe può quindi estendere al più una sola altra classe, e può avere zero o più sottoclassi.

Nei linguaggi con derivazione multipla, se in due classi base un metodo viene definito, nella sottoclasse, non è definito quale delle due implementazioni da ereditare, se non entrambe. Questo problema viene risolto nei linguaggi di programmazione dove è presente la gerarchia multipla, ma la sua utilità è limitata a pochi casi.

In Scala questo problema viene risolto creando un'ordine totale, tramite un processo di linearizzazione della gerarchia in base a quale tipo viene prima all'interno. Se un membro è definito in entrambi le classi base, la classe estesa da queste due mischia le due implementazioni. Quando le due classi non presentano membri che si sovrappongono tra di loro è possibile unire le due classi poiché non si presentano conflitti.

In Java invece è possibile effettuare una derivazione multipla tra interfacce, la stessa classe `String` implementa tre interfacce. Questo è possibile poiché non è possibile mischiare il corpo dei metodi forniti dalle interfacce, dato che vengono implementati all'interno della classe che le estende. La gerarchia dei tipi non è più lineare, poiché comprende anche le interfacce, che possono estendere zero o più interfacce, mentre le classi possono implementare zero o più interfacce. La gerarchia delle classi è lineare, mentre la gerarchia delle interfacce non è lineare. In Java 8 vennero introdotti i metodi default, che permettono una derivazione multipla ristretta, per poter estendere librerie senza perdere la retrocompatibilità.

La scelta dei corretti tipi da definire in un programma è un delicato esercizio di modellazione. Nella creazione dei tipi occorre considerare anche gli aspetti dinamici.

Il sovraccarico di un metodo viene risolto a tempo di compilazione in Java, sulla base dei tipi statici, ma è possibile ottenere un effetto simile ad un sovraccarico sulla base di tipi dinamici,

utilizzando una chiamata polimorfa ed il late-binding.

Si considera una gerarchia di tipi, avente come radice un'interfaccia vuota non modificabile, e si considera un metodo che vuole essere implementato sull'interfaccia. Si potrebbe creare quindi una classe contenente metodi sovraccarichi per quante sono le classi che implementano tale interfaccia. Ma questo è possibile solamente quando si coincide il tipo statico con il tipo dinamico delle variabili locali delle classi, passati ai metodi sovraccarichi. Altrimenti non sarebbe possibile utilizzare questa tecnica, poiché spesso non è noto il tipo dinamico del parametro passato al metodo. Per risolvere questo problema quindi bisogna trovare un modo per passare un tipo generico a questo metodo, basandosi solamente sul tipo statico. Si vuole risolvere il metodo sovraccarico sulla base del tipo dinamico.

In generale è sconsigliato determinare il tipo dinamico di un oggetto tramite il codice, poiché il polimorfismo deve chiamare il metodo corretto in base al tipo dinamico, per cui in questo caso non si considera questa tecnica.

Il vantaggio delle chiamate polimorfe è che permettono di scegliere quale metodo da invocare sulla base del tipo dinamico, per cui modificando la definizione di codice della libreria che implementa l'interfaccia vuota, per ospitare un metodo che accetta una classe, contenente metodi sovraccarichi per ogni sottoclasse dell'interfaccia. In generale dato che le interfacce fornite da librerie esterne sono immutabili utilizzano metodi in grado di accettare un tipo generico. Questo tipo generico deve essere in grado di distinguere tutti i tipi della gerarchia di tipi, quindi sovraccarichi.

Le implementazioni del metodo che prende questo tipo generico sono degli override, ed al loro interno viene chiamato il metodo sovraccarico di questo tipo generico, passando come parametro sé stesso tramite `this`, che ha come tipo statico il tipo della classe o suo sottotipo. In questo modo la chiamata polimorfa sovrascritta in una sottoclasse generica dell'interfaccia iniziale invoca un metodo dove il tipo statico corrisponde al tipo dinamico. Questa tecnica si chiama "dispatching".

4 Generics

I generics sono uno strumento per scrivere classi e metodi parametriche rispetto ad un tipo. Più che progettare classi generiche, si vuole utilizzare classi generiche, la loro progettazione va oltre gli obiettivi formativi del corso. Per classe generica si intende un tipo che dipende da un altro tipo. In questo corso si useranno classi generiche relative alla gestione di collezioni di oggetti, in particolare con il pacchetto `java.util`.

Senza utilizzare classi generiche per poter realizzare tipi parametrici, consiste nello sfruttare il polimorfismo, utilizzando la classe `Object` ed utilizzando il principio di sostituzione per poter ottenere l'effetto desiderato, essendo la radice della gerarchia delle classi. Prima di Java 5, questo era l'unico metodo di programmazione fornita da Java, utilizzando downcast sparsi nel codice. Questo controllo lasco dei tipi potrebbe portare ad errori a tempo di esecuzione. Questo rappresenta solamente un'approssimazione del tipo desiderato.

Introdotti in Java 5, i generics sono uno strumento per scrivere classi ed interfacce il cui tipo diventa parametrico rispetto ad uno o più tipi. Il tipo generico dipende da un tipo chiamato formale, che deve essere istanziato affinché funzioni correttamente. I tipi formali in generale vengono chiamati con singole lettere maiuscole, per indicare che una classe è un generics si utilizza la notazione accanto al tipo del nome della classe `<T, E>`, che racchiude il nome dei tipi formali `T` e `E`. Questo tipo viene usato come una dichiarazione di tipo all'interno della classe. Per utilizzare una classe generica, tutti i tipi formali devono essere istanziati e completamente definiti con un tipo attuale per renderlo utilizzabile.

La distinzione tra parametro attuale e parametro formale è simile alla relazione tra tipo formale e tipo attuale, ma solo superficialmente. A tempo di esecuzione, infatti, non esistono tipi formali, esistono solo tipi attuali, poiché il compilatore scrive nel file `.class` le relazioni tra i soli tipi statici, rimpiazzando tutti i tipi formali con rispettivi tipi attuali.

Essendo Java un linguaggio ibrido sono presenti informazioni, non rappresentate come oggetti, per cui non è possibile utilizzare questi tipi come tipi formali di una classe generica. Non è possibile istanziare i tipi di una classe, di una interfaccia o di un metodo generico con tipi primitivi. Per risolvere questo problema è possibile rappresentare i tipi primitivi mediante le classi "wrapper". Per ogni tipo primitivo esiste una corrispondente classe wrapper che consente di esprimerlo tramite oggetto, immutabile, costruendoci un oggetto attorno:

```
int     -> Integer
double  -> Double
float   -> Float
char    -> Character
boolean -> Boolean
```

Le classi wrapper sono definite nel pacchetto `java.lang`, per cui non è necessario importarle, esplicitamente. Le classi wrapper essendo oggetti devono essere istanziati mediante `new`, e per ottenere il valore primitivo contenuto dovranno essere usati opportuni metodi getter:

```
int i;
Integer iWrapped = new Integer(i);
int j = iWrapped.intValue();
```

Altri metodi comunemente usati sono `valueOf()`, `parseInt()`, che dipende da quale wrapper di tipo primitivo si sta usando, e `equals()`. Per ovviare a questa eccessiva verbosità, da Java 4, la gestione degli oggetti wrapper venne semplificata tramite tecniche di "boxing" e "unboxing". Che permettono di effettuare dichiarazioni, inizializzazioni e conversioni dirette tra tipi primitivi ad oggetti wrapper. Questa tecnica di conversione automatica da tipi primitivi ad oggetti di tipo wrapper e viceversa è simile alla conversione di tipi effettuata dal compilatore, ma è diversa poiché utilizza sia oggetti che tipi primitivi. Il compilatore automaticamente inserisce i metodi necessari per effettuare l'unboxing o il boxing tra tipi wrapper e primitivi. Questa semplificazione della conversione di tipo nasconde la differenza tra i due tipi, e potrebbe portare ad errori logici, poiché alcune operazioni permesse sui tipi primitivi non sono permesse sui tipi wrapper e viceversa, e non necessariamente verranno identificati correttamente a tempo di compilazione o esecuzione dall'IDE.

Poiché collezioni generiche di tipo formale `T` possono accogliere oggetti dello stesso tipo, non possono essere presenti oggetti di tipo `T`, e contemporaneamente di un suo sottotipo. Per risolvere questo problema bisogna indicare al compilatore, che il tipo formale può essere anche un suo sottotipo, senza fornire il tipo specifico, poiché non può includere ogni sottotipo della gerarchia. Per indicare un tipo qualsiasi si utilizza una "wildcard". La sintassi di questa wildcard viene rappresentata da `?`, per indicare che si può accettare un parametro di tipo `T` o sottotipo di `T` si usa quindi la segnatura `<? extends T>`. Il compilatore decide a tempo di compilazione il tipo attuale dei tipi formali della classe generica. Questo uso uso di wildcard rappresenta un "limite" superiore rispetto alla gerarchia dei tipi, per definire quali tipi sono accettabili dalla classe.

Delle classi non generiche possono ospitare metodi statici generici, inserendo il tipo formale subito prima del tipo restituito:

```
public static <T> tipoRestituito nomeMetodo(T e);
```

Questo metodo può essere evocato come per le classi generiche solo se il tipo formale è perfettamente definito.

Analogamente si può indicare un limite superiore tramite la segnatura `<? super T>`, in questo modo si può accettare un parametro di tipo `T` o di un suo supertipo.

Per determinare il tipo dei parametri formali nelle segnature, si può utilizzare la regola mnemonica PECS: "Produce Extends Consumer Super". Per parametri che producono elementi si utilizza il sottotipo `<? extends T>`, mentre per parametri che consumano elementi si utilizza il supertipo `<? super T>`. Inoltre possono essere utilizzati entrambi per limitare il tipo del parametro ulteriormente.

I tipi generici si usano per lavorare con le collezioni, presenti in `java.util.Collection`. Quando non è strettamente necessario dare un nome al tipo formale si utilizza la sintassi `<?>`.

4.1 Collezioni

Il linguaggio offre delle librerie native, utilizzate per implementare collezioni come liste, mappe, tabelle di hash, etc. Tutti queste collezioni appartengono al "Java Collection Framework".

Molte applicazioni richiedono di gestire collezioni di oggetti, gli array, usati dall'inizio del corso sono un strumento di basso livello. La dimensione di una collezione generalmente non è nota a

priori, e gli array non permettono di variare la sua dimensione, per cui è possibile sia necessario creare un array di dimensione maggiore, per evitare un overflow.

Inizialmente venne introdotto in Java 2, ma venne esteso in Java 5, con l'introduzione dei Generics, introdotti appunto per generalizzare il loro uso nelle collezioni preesistenti, e successivamente aggiornato nelle versioni successive per introdurre importanti aggiunte. Viene contenuto nel pacchetto `java.util`, contiene collezioni non specialistiche, tra le più utilizzate in assoluto.

Il Java Collection Framework (JCF), contiene interfacce ed implementazioni, come le liste e le mappe. Collezioni utilizzate in tutti i linguaggi di programmazione moderni, generiche, dipendenti le liste da un tipo formale, mentre le mappe, dizionari, array associativi dipendono da due tipi formali, una chiave ed il valore. Altre interfacce permettono di enumerare la collezione indipendentemente dal tipo della stessa.

L'interfaccia `Collection<E>` dichiara una generica collezione, generalizzata sia su `List<E>` che su `Set<E>`. Nelle liste gli elementi sono salvati sequenzialmente e possiedono una posizione, senza gestire i duplicati, mentre gli insiemi non ammettono duplicati e gli elementi non possiedono una posizione.

Dentro questa interfaccia sono presenti tutti i metodi utili da applicare a collezioni di elementi, che essi siano liste o insiemi. Questi metodi effettuano operazioni come l'aggiunta e la rimozione di elemento, verifica della dimensione, aggiungere tutti gli elementi ad un'altra collezione, ed ottenere un iteratore con cui è possibile scandire la collezione.

L'interfaccia `Set<E>` estende `Collection<E>`, è una collezione che non può contenere duplicati. Questa collezione offre tutti e soli i metodi dell'interfaccia `Collection`. Poiché non ammettono duplicati, è necessario definire un criterio di equivalenza tra due elementi dell'insieme, in modo da utilizzarlo come criterio per rifiutare aggiunte all'insieme.

L'interfaccia `List<E>` estende `Collection<E>`, corrisponde ad una sequenza ordinata di elementi. Ogni elemento può quindi essere identificato dalla sua posizione. Quindi permette accesso posizionale e la ricerca della posizione di uno specifico elemento.

L'interfaccia `Map<K,V>` offre le operazioni di una mappa, o dizionario; è una collezione di coppie chiave-valore. Permette l'accesso per chiave, ed la creazione di una collezione contenente tutte le chiavi o valori.

La classe `java.util.Collections`, offre un vasto insieme di metodi statici generici, per manipolare le collezioni come l'ordinamento, se la collezione lo permette, la ricerca del massimo e del minimo, etc.

I metodi offerti dall'interfaccia `Collection<E>` vengono usati per una collezione generica, permettono di svolgere tre categorie di operazione:

- Manipolazione di base: Coinvolgono un singolo elemento, oppure verificano proprietà della collezione;
- Bulk: Coinvolgono tutti gli elementi della collezione;
- Conversione di/ad un array.

Tutti i metodi che producono un effetto scontato, o non producono alcun effetto collaterale vengono implementati utilizzando il tipo `Object` come parametro. Mentre metodi che necessitano di un tipo specifico, altrimenti si provocano effetti indesiderati, vengono tipati utilizzando il tipo

specifico per il tipo passato, come per il tipo `add()`. I metodi di aggiunta o rimozione restituiscono un booleano, vero solo se l'aggiunta o rimozione ha causato una modifica della collezione.

```
// Manipolazione di Base:
    int size();
    boolean isEmpty();
    boolean contains(Object e);
    boolean add(E e);
    boolean remove(Object e);
    Iterator<E> iterator();
```

I metodi con la variante di suffisso `All` permettono di effettuare addizioni rimozioni a "lotto". Hanno una segnatura strettamente collegata ai metodi basici, prendono come parametro il tipo `Collection<?>`. Tutti questi i metodi sono lascamente tipati, eccetto `addAll()`, che richiede una collezione di tipo `<E>` o di un suo sottotipo, affinché sia possibile contenerla nella collezione di arrivo.

```
// Operazioni di Bulk
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(collection<?> c);
    void clear;
```

La conversione ad un array viene tipata come un array di oggetti `Object[]`, mentre è possibile effettuare la stessa ad un metodo parametrico `toArray(T[] a)`, che prende un array dichiarato esternamente da riempire, e ne prende il tipo:

```
// Operazioni su Array
    Object[] toArray();
    <T> T[] toArray(T[] a);
```

Uno dei motivi per la presenza di questi metodi, consiste nella creazione di questo pacchetto precedentemente ai Generics in Java 5. Inoltre la seconda versione, che costringe di specificare il tipo, permette al compilatore di effettuare un'analisi statica dei tipi più stretta, senza che sia il compilatore a cercare il tipo dalla collezione, poiché per motivi di retrocompatibilità non è possibile realizzare array generici in Java, appunto per l'introduzione tardiva dei Generics. Poiché i Generics vengono utilizzati a tempo di compilazione e non è possibile identificare a tempo di esecuzione quale metodo o classe sia generica o meno.

Per utilizzare il secondo tipo per generare un array generico, si può passare come parametro un array lungo zero, in modo da passare il tipo, ed è il metodo a creare un array dello stesso tipo, di lunghezza sufficiente per ospitare tutta la collezione.

4.2 Iteratori

Storicamente ogni collezione diversa richiedeva un modo diverso per scandire la collezione, basato su metodi di enumerazione indipendenti tra di loro. Il codice per enumerare gli elementi dipendeva

fortemente dalla collezione stesa. Gli iteratori rappresentano uno stesso modo per poter enumerare elementi di collezioni diverse tra di loro. Si è quindi oggettificato il concetto di enumerazione di una collezione, indipendentemente dal tipo della collezione ospitante di questi elementi. Gli iteratori `Iterator<E>` sono creati invocando il metodo `iterator()` sulla collezione, munito dei metodi:

```
boolean hasNext();
E next();
void remove();
```

In pratica è un cursore che scandisce la collezione sottostante ricordando la sua posizione nella scansione. Una posizione lecita si trova subito prima o subito dopo di un elemento della collezione su cui si sta iterando, non si trova mai sopra un elemento.

I metodi verificano se l'elemento corrente ha un successivo, ed il metodo che restituisce il riferimento a quest'elemento con `hasNext()` ed `next()`. Rappresenta un'enumerazione con stato, poiché deve salvare il suo stato attuale. Se si arriva alla fine della collezione, chiamando il metodo `next()` viene sollevata un'eccezione. Per cui convenzionalmente si chiama il metodo `hasNext()` per verifica l'esistenza dell'elemento successivo, prima di iterare avanti.

Da Java 7 in poi, data una variabile che può ospitare una collezione rispetto ad una classe formale, nella dichiarazione della collezione si può omettere il tipo formale, e lasciare `<>` vuoto.

L'utilizzo degli iteratori è ripetitivo, poiché utilizza le stesse operazioni per iterare su ogni elemento della collezione:

```
Iterator<E> itr = nomeCollezione.iterator();
while(itr.hasNext()){
    E e = itr.next();
    // operazioni
}
```

Per cui è stata definita una forma sintetica per esprimere questo processo, utilizzando la forma "for-each", questa interfaccia offre una notevole semplificazione, ma non è possibile accedere all'indice di iterazione:

```
for(E e : itr)
    // operazioni su e
```

Dove l'iteratore `itr` è un sottotipo dell'interfaccia `java.lang.Iterable<E>`. Il compilatore traduce questa sintassi nella sintassi descritta precedentemente. Grazie all'intervento del compilatore è possibile utilizzare questa sintassi su ogni classe che implementi `Iterable<E>`; inoltre permette il suo uso anche sugli array, nonostante non la implementino.

Un iteratore per effettuare il suo lavoro deve conservare un rifermento alla collezione che lo ha creato su cui vuole iterare, successivamente alla sua creazione. La presenza di un metodo `remove()` rende più evidente la natura di questo legame. Rimuove l'ultimo elemento restituito dalla chiamata `next()`, l'ultimo elemento restituito dalla collezione. Viene introdotto questo metodo per permettere all'iteratore di conoscere lo stato della collezione dopo la rimozione, poiché è passata attraverso l'iteratore. Ovviamente chiamate successive a questo metodo senza ulteriori chiamate a

`next()` producono un errore, poiché non è più presente nella collezione. Invece se viene chiamato il metodo `remove()` sulla collezione, il comportamento dell'iteratore non è scontato.

Alcuni iteratori conservano la collezione precedente, altri si aggiornano, altri ancora producono un errore. La documentazione dell'interfaccia specifica che il suo comportamento non è specificato se la collezione viene modificata quando l'iteratore sta ancora iterando sulla collezione. Generalmente applica la strategia del "fail fast", ed alla prima chiamata al metodo dell'iteratore dopo una modifica alla collezione solleva un'eccezione `ComponentModificationException` Per verificare se la collezione è stata cambiata, la collezione genetrice crea un oggetto che salva la versione corrente della collezione, aggiornata ad ogni modifica della collezione. Questo oggetto viene quindi controllato dall'iteratore per verificare che la collezione non è stata modificata. Nella documentazione viene sconsigliato l'affidamento su questo oggetto, poiché conserva la versione in un intero, che può contenere al massimo quattro miliardi di modifiche alla collezione, prima di effettuare un overflow, e ritornare al valore iniziale; quindi in casi improbabili è possibile che vengano effettuata abbastanza operazioni per creare un overflow, modificando la collezione senza che l'iteratore sia in grado di rilevarlo.

4.3 Liste

Le liste sono strutture dati astratte, con aggiunta in coda, tramite il metodo `add()`. Gli elementi vengono salvati sequenzialmente, e vengono indicizzati a partire da zero, come per gli array. Si può accedere ad un elemento specifico per indice, e cercare l'indice tramite metodi presenti:

```
E get(int index);
int indexOf(Object o);
```

Il package `java.util` offre due implementazioni dell'interfaccia `List<E>`:

- `ArrayList<E>`;
- `LinkedList<E>`.

Gli `ArrayList<E>` hanno un'aggiunta molto efficiente per indice, ma avendo una lunghezza fissa, quando l'array è pieno, bisogna creare un array nuovo di grandezza superiore. Per cui le sue prestazioni sono strettamente dipendenti dalla dimensione attuale del numero dei dati salvati.

In generale la rappresentazione `LinkedList<E>`, di liste concatenate, ad una o due sentinelle, permettono una dimensione variabile, ma gli accessi avvengono sequenzialmente, per cui hanno un tempo lineare, al contrario degli `ArrayList<E>`, in tempo costante eccetto il trabocco. Nella pratica la complessità delle JVM moderne rende queste differenze impercettibili.

I costruttori di queste due implementazioni sono sovraccarichi. Entrambi hanno un costruttore vuoto, ed un costruttore che prende come parametro una collezione di sottotipo del tipo formale `E`: `Collections<? extends E>`, permettendo di convertire un'intera collezione in una lista. Viene inserito il sottotipo poiché le collezioni non sono covarianti con il tipo della stessa. Inoltre per le liste tramite array, esiste un costruttore che specifica la dimensione iniziale. Per una lista concatenata sono presenti dei metodi di accesso in testa e coda, poiché avviene in tempo costante, tramite riferimenti alla testa ed alla coda:

```
boolean addFirst();
boolean addLast();
E getFirst();
E getLast();
E removeFirst();
E removeLast();
```

Tuttavia utilizzare questi metodi specifici rende le due implementazioni meno intercambiabili. Inoltre per le liste concatenate esiste un metodo `listIterator()` per realizzare un iteratore specifico per le liste di tipo `ListIterator<E>`, che estende `Iterator<E>`.

Nella classe `Collections` sono presenti vari metodi efficienti per ordinare liste, effettuare una ricerca binaria, cercare il massimo o minimo nella lista, etc. Queste operazioni sono possibili solamente se esiste una relazione d'ordine tra gli elementi presenti nella collezione. Quest'operazione viene affidata all'apposita interfaccia `java.lang.Comparable<E>`, per un ordinamento naturale o interno; oppure ad una classe esterna agli oggetti contenuti tramite l'interfaccia `java.util.Comparator<T>`, per un ordinamento esterno.

La prima interfaccia consiste di un unico metodo:

```
public int compareTo(T that);
```

Restituisce un valore minore maggiore o uguale a zero a seconda che l'oggetto `this` su cui viene applicato il metodo sia rispettivamente minore, maggiore o uguale al riferimento all'oggetto ricevuto come parametro.

Molte librerie standard già implementano quest'interfaccia, adottando una semantica scontata. Viene eventualmente delegato quindi ad una di queste implementazioni, quando si vuole implementare l'interfaccia su un altra classe.

4.3.1 Ordinamento Interno

Il metodo per ordinare le liste corrisponde al metodo statico `Collections.sort()`, in due versioni sovraccariche corrispondente ai due metodi per determinare un criterio di ordinamento. La segnatura per l'ordinamento naturale corrisponde a:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Affinché qualcosa sia ordinabile, deve implementare un criterio di ordinamento. Questo criterio di ordinamento deve essere in grado di ordinare lo stesso tipo o un suo supertipo, che può essere contenuto nella stessa collezione, per il criterio di sostituzione.

Per ottenere un minimo o un massimo da una lista `List<T>`, i cui elementi implementano l'interfaccia `Comparable<T>`, può essere calcolato mediante i metodi offerti da `Collections`, di segnatura

```
public static <T extends Object & Comparable<? Super T>> T min(Collections<? extends T> c)
public static <T extends Object & Comparable<? Super T>> T max(Collections<? extends T> c)
```

Per gli stessi motivi del precedente metodo. Semplificato in:

```
public static <T extends Comparable<? Super T>> T min(Collections<? extends T> c)
public static <T extends Comparable<? Super T>> T max(Collections<? extends T> c)
```

Questo metodo non compila se i metodi statici non risultano avere i metodi necessari per comparare, a tempo di compilazione, e quindi rappresenta un'enorme vantaggio di Java. L'analisi dei tipi è uno strumento in grado di individuare errori anche a tempo di compilazione, per impedire la creazione di errori a tempo di esecuzione.

Se si ha una gerarchia dei tipi, è possibile implementare l'interfaccia `Comparable<T>` solamente su un unico tipo. A tempo di compilazione vengono utilizzati e rimpiazzati con tipi attuali a tempo di esecuzione, per la necessità di mantenere la retrocompatibilità di Java. Per questo un'unica gerarchia può avere un'unica implementazione di un'interfaccia. Si rifiuta di considerare due instanziamenti dello stesso tipo generico con due tipi attuali diversi, poiché nel file `.class` si avrebbero due `Comparable<T>` che riferiscono a due tipi diversi, che la JVM non è in grado di gestire. Infatti a tempo di esecuzione ogni tipo dinamico deve essere sempre distinguibile anche se vengono eliminati i riferimenti ad i tipi attuali utilizzati nella definizione dei tipi generici. Per lo stesso motivo non è possibile creare un array generico in Java.

Per questi motivi si preferisce mantenere un ordinamento naturale per tipi semplici, e si evita di utilizzarlo su una gerarchia per non "sprecare" il suo utilizzo. Per cui risulta ben motivato l'utilizzo di classi esterne per implementare l'ordinamento.

4.3.2 Ordinamento Esterno

L'ordinamento esterno viene consiste nell'utilizzo dell'interfaccia `java.util.Comparator<T>`, composta da un'unico metodo:

```
public int compare(T o1, T o2)
```

Che deve restituire un valore minore, maggiore o uguale, a seconda se l'oggetto riferito da `o1` sia minore, maggiore o uguale all'oggetto riferito da `o2`.

In `Collections` esistono metodi per il calcolo del massimo e del minimo, secondo l'ordinamento esterno:

```
static <T> T min(Collections<? extends T> c, Comparator <? super T> cmp)
static <T> T max(Collections<? extends T> c, Comparator <? super T> cmp)
```

4.4 Insieme

L'interfaccia `Set<E>` rappresenta un insieme, le due classi più famosi che lo implementano sono `HashSet<E>` e `TreeSet<E>`, un insieme "ordinato".

Gli insiemi sono collezioni che non contengono duplicati. Offre tutti e soli i metodi dell'interfaccia `Collection<E>` con la restrizione che le classi che la implementano non contengono duplicati. Per poter riconoscere duplicati, bisogna definire un criterio di equivalenza tra gli elementi, basato sul dominio di utilizzo. Per stabilire il criterio di equivalenza si può utilizzare il metodo `equals()`, ed anche il metodo `hashCode()` per la classe `HashSet<E>`. Questi due metodi gemelli sono entrambi necessari per verificare ed evitare la presenza di duplicati nell'insieme.

Nel caso di `TreeSet<E>`, ed in generale di insiemi ordinati, è necessario un criterio di ordinamento, naturale tramite `Comparable<E>`, oppure passando al momento della costruzione un oggetto `Comparator<E>` esterno.

4.4.1 HashSet

Una funzione hash è una funzione che calcola un numero intero, detto codice hash, a partire da un oggetto, in grado di verificare se due oggetti sono diversi o uguali, poiché due oggetti equivalenti avranno sicuramente lo stesso codice hash, mentre molto probabilmente due oggetti non equivalenti possiedono codici hash diversi. Quando due oggetti non equivalenti hanno lo stesso codice hash, è necessario interrogare il metodo `equals()`, questo fenomeno si chiama collisione; in generale si prediligono funzioni di hash che minimizzano le collisioni.

Una tavola hash rappresenta un array, dove ogni elemento occupa una posizione corrispondente al suo codice hash, quando viene raggiunta la dimensione massima dell'array, viene copiato in una array di lunghezza doppia. Per risolvere il problema delle collisioni viene realizzato come un array di liste di conflitto, dove ogni elemento rappresenta una lista a tutti gli elementi ospitati nella tabella di hash aventi lo stesso codice di hash.

Se la funzione è ben definita, le operazioni di appartenenza, rimozione ed inserimento avvengono in tempo costante.

Le implementazioni degli insiemi e delle mappe si basano interamente sulle tavole hash. Quando si vuole implementare un `HashSet<E>`, tutti gli oggetti della collezione, di tipo formale `E`, devono avere sia il metodo `hashCode()` che il metodo `equals()`. Se non vengono definiti, vengono ereditati da `java.lang.Object`, in cui il codice di hash corrisponde all'indirizzo di memoria dell'oggetto, ed il metodo `equals()` confronta gli indirizzi di memoria.

Il metodo `equals()` definito deve rispettare le proprietà riflessiva, transitiva e transitiva, inoltre deve restituire falso quando gli viene passato un riferimento nullo, mentre quando confronta lo stesso oggetto, deve restituire vero. L'unica asimmetria consiste nel chiamare il metodo su un riferimento nullo, genera infatti una `NullPointerException`. Una volta definito questo metodo, il metodo `hashCode()` insiste sugli stessi campi del metodo precedente, e può essere realizzato come una semplice combinazione lineare. L'equivalenza di un metodo primitivo, non viene delegata al metodo `equals()` come per i metodi non primitivi, ma viene verificato tramite la sintassi `==`, questa rappresenta una piccola ineleganza del linguaggio Java, nella sua gestione dei metodi primitivi.

Appena viene aggiunto un nuovo elemento alla collezione, questa chiama il metodo `hashCode()` per verificare se non è presente alcun elemento di stesso codice hash, altrimenti invoca il metodo `equals()` su ogni elemento della lista di collisione. Questi due metodi sono assolutamente indipendenti l'uno dall'altro, e permettono facilmente di realizzare implementazioni molto efficienti. Per ottenere implementazioni più efficienti, i calcoli effettuati dal metodo `hashCode()` devono essere molto meno onerosi del metodo `equals()`, poiché il primo metodo viene invocato sempre, mentre il secondo solo in caso di collisioni. Tuttavia utilizzare funzioni di hash troppo semplici potrebbe portare ad una maggiore probabilità di collisione, rendendo l'implementazione meno efficiente.

Se non venisse implementato il metodo `hashCode()`, si erediterebbe quello definito da `java.lang.Object`. Se non venisse implementato si potrebbero riscontrare errori logici, poiché il metodo `equals()` considererebbe uguali due oggetti distinti, ma aventi codice hash differenti, un risultato paradossale.

In generale per entrambi i metodi si invocano a loro volta gli stessi metodi definiti sulle variabili di istanza, invece di definire criteri di equivalenza tra di loro. La funzione di hash restituirà quindi una combinazione lineare dei codici hash delle variabili di istanza, mentre il metodo `equals()` confronterà le variabili di istanza dei due oggetti confrontati.

4.4.2 TreeSet

L'implementazione più popolare degli alberi ordinati, viene realizzato tramite alberi binari di ricerca, i criteri di equivalenza si basano sui metodi definiti per l'ordinamento interno ed esterno sulle liste.

Tutto quello trattato sulle liste vale allo stesso metodo per gli insiemi ordinati. Garantisce che gli elementi siano posizionati secondo un dato ordinamento. Se gli elementi non implementano `Comparable<E>`, o non viene passato in fase di costruzione un oggetto `Comparator<E>`, si può sollevare un errore a tempo di esecuzione. Vengono implementati tramite alberi di ricerca binari, per cui è necessario che gli elementi possano essere confrontati ed ordinati. Per motivi di retrocompatibilità, è possibile realizzare un `TreeSet` di oggetti non ordinabili, ma alla prima occasione, a tempo di esecuzione, verrà sollevata un eccezione, poiché gli elementi non possono essere ordinati di tipo `ClassCastException`.

Essendo realizzati tramite alberi, ogni elemento corrisponde ad un nodo dell'albero e può avere al massimo due figli, di cui conserva il loro riferimento. Negli alberi binari di ricerca tutti i discendenti di sinistra di qualunque nodo sono inferiori al valore del dato in quel nodo, mentre tutti i discendenti di destra sono maggiori.

Se viene definito un criterio di ordinamento interno con `compareTo()`, implicitamente viene definito un criterio di equivalenza. Quindi bisogna definire i metodi di ordinamento in accordo con i metodi di equivalenza `hashCode()` e `equals()`, e viceversa. Poiché deve mantenere la collezione ordinata dopo ogni aggiornamento, questi risultano meno efficienti rispetto ad un insieme non ordinato.

4.4.3 NavigableSet

Da Java 6 in poi furono introdotte nuove implementazioni di un insieme specializzata di `SortedSet<E>`, l'interfaccia `NavigableSet<E>`, contenente i metodi per poter accedere all'elemento in cima o in coda all'insieme, e per poter accedere agli elementi in ordine invertito:

```
E pollFirst();
E pollLast();
NavigableSet<E> descendingSet();
```

Altri metodi come `headSet()`, `tailSet()` e `subSet()` restituiscono delle "viste attive" ovvero dei sottoinsiemi dell'insiemi originale, ed i cambiamenti effettuati sulla vista si riflettono sull'insieme originale. Per evitare questo comportamento è sufficiente utilizzare la vista in un costruttore di un'altra collezione.

`TreeSet<E>` infatti implementa entrambe queste interfacce. Per la retrocompatibilità è stato necessario inserire un metodo sovraccarico `headSet()` che restituisce un `SortedSet<E>`, con una segnatura asimmetrica rispetto all'originale `headSet()`, poiché restituisce un `NavigableSet<E>` e

richiede un valore booleano per definire il criterio di uguaglianza. Questa rappresenta una scelta discutibile, anche se è stata necessaria ai fini di mantenere la retrocompatibilità del linguaggio.

4.5 Mappe

Una mappa o dizionario o array associativo rappresenta un insieme di elementi formati da coppie di due tipi formali `<K,V>`, dove `K` rappresenta la chiave "Key", e `V` rappresenta il valore "Value", dove gli accessi sono particolarmente efficienti se vengono effettuati per chiave.

Ad ogni chiave può essere associato un unico valore, ed ogni chiave è univoca.

I metodi base per accedere, aggiungere, rimuovere e verificare l'esistenza coppie chiave-valore sono:

```
V put(K k, V v);
V get(Object k);
V remove(Object k);
boolean containsKey(Object k);
```

Se una chiave è già presente nella mappa, quando viene inserita una nuova coppia, viene sovrascritto il vecchio valore, e viene restituito. Il metodo `get()` richiede un oggetto di tipo `Object`, per cui è lascamente tipato, sia per mantenere la retrocompatibilità sia per offrire una risposta istantanea in caso il tipo non corrisponda al tipo formale `V`.

I metodi bulk offerti e viste di mappe sono:

```
void putAll(Map<K, V>, m);
void clear();
Set<K> keySet();
Collection<V> values();
```

Per ottenere un insieme delle chiavi, si utilizza `keySet()`, poiché tutte le chiavi sono uniche nell'intera mappa; mentre se si vuole restituire l'insieme di valori, questi possono essere duplicati quindi il tipo di ritorno è lascamente tipato come una qualsiasi collezione del tipo formale `V`.

Le mappe e gli insiemi si basano su concetti molto simili, infatti in realtà l'implementazione di `HashSet<K>` si basa su un'istanza di `HashMap<K,?>`. Sono presenti diverse implementazioni dell'interfaccia mappa: `HashMap<K, V>` e `TreeMap<K, V>`.

I criteri di equivalenza vengono definiti allo stesso modo degli insiemi, visti precedentemente. Per l'implementazione di mappa `HashMap<K, V>` è necessario definire i due metodi `equals()` e `hashCode()`, della classe delle chiavi. Mentre per utilizzare `TreeSet<K, V>` è necessario un ordinamento interno o esterno.

Le mappe risultano, spesso, più comode delle liste ed insiemi, poiché non è possibile accedere ad un elemento di un insieme senza un suo duplicato.

In Java 6 venne introdotta l'implementazione `NavigableMap<K, V>`, che introduce metodi per le viste attive delle mappe, ed altri metodi utili, allo stesso modo per gli insiemi. Infatti sono presenti le stesse scelte discutibili, per mantenere la retrocompatibilità, come il metodo sovraccarico `headMap()` che restituisce un `SortedMap<K>`, invece se viene passato un valore boolean insieme alla chiave restituisce un `NavigableMap<K>`:

```
SortedMap<K> headMap(K toKey);
NavigableMap<K> headMap(K toKey, boolean inclusive);
```

Nonostante la loro enorme utilità rispetto alle altre collezioni, bisogna distribuire le responsabilità adeguatamente tra le varie classi, attraverso uno studio di dominio accurato, per non ricadere nella creazione di mappe nidificate contenenti altre mappe e così via. L'uso esclusivo di mappe per raggruppare in una singola collezioni più elementi distinti del progetto è un segno per la ridistribuzione delle responsabilità tra le varie classi. In questo modo si diminuiscono i livelli di nidificazione della mappa, agevolando la scrittura di test e l'utilizzo da parte di chi non è familiare con il progetto.

5 Introspezione

La riflessione o introspezione è una particolare caratteristica del linguaggio Java, raramente presente in linguaggi antecedenti, ormai diffusa sui linguaggi moderni. Permette di scrivere codice ed analizzare o modificare il codice compilato, nello stesso linguaggio. Un'esempio è appunto l'IDE Eclipse, scritto in Java per programmare in Java, approfitta quindi di un massiccio uso dell'introspezione. JUnit è un altro programma che utilizza la riflessione per individuare nel progetto tutti i metodi annotati tramite `@Test`, e le classi. Prima delle riflessioni, il runner di JUnit richiedeva tutti i nomi dei metodi di test preceduti dal termine "test".

In Java esiste una classe `java.lang.Class<T>`, per ogni tipo `T`, esiste quindi a tempo di esecuzione un oggetto che ne descrive il suo tipo stesso. In questo modo è possibile scrivere metodi `equals()` ed `HashCode()` tra oggetti di una gerarchia di tipi.

La classe `Class<<T>` di tipo generico `T` viene generata ogni volta che viene definito il tipo `T` da un programma Java, per descriverne il suo tipo, e quindi descrive anche il contenuto del corrispondente file `.class`. Quest'istanza permette di utilizzare i metodi per poter analizzare la classe `T`. Questa classe è un oggetto garantito, esisterà sempre e solo un unico oggetto associato ad una classe, creato automaticamente a livello della JVM, anche a livello di sicurezza, per cui non è gestibile dal programmatore. Alcune ottimizzazioni invece istanziano questi oggetti a tempi diversi dell'esecuzione per migliorare le prestazioni.

Tutti i tipi possiedono una variabile statica pubblica chiamata `class`, corrispondente all'oggetto di tipo `java.lang.Class<T>`. Questo oggetto può essere usato per ottenere le proprietà del tipo `T`.

La riflessione viene applicata nello sviluppo di programmi con funzionalità complesse, applicativi e framework per rendere più efficiente la scrittura del codice, tramite annotazioni ed introspezione. Inoltre risulta molto efficace anche per applicazioni più semplici, permette di realizzare oggetti tramite il nome salvato su un oggetto di tipo `String`, creando il tipo dell'oggetto a tempo dinamico. Viene utilizzato il metodo `newIstance()`, presente nel file di tipo `Class` per creare un sottotipo di un oggetto a tempo dinamico:

```
Class<tipoDiObj> classeDiObj = tipoDiObj.class;
superTipoDiObj nomeObj = classeDiObj.newIstance();
```

In questo modo si crea un oggetto senza invocare il costruttore, assegnando ad una variabile una nuova istanza di un suo sottotipo.

Un altro metodo di questo oggetto è il metodo `forName()` che prende una stringa, e permette di caricare una classe fornendo il suo nome tipato all'interno di una stringa, e restituisce un riferimento ad un oggetto `Class<?>`, su cui può essere applicato il metodo `newIstance()` per creare un'istanza di una classe, se la macchina virtuale è in grado di identificare nel build path il file `.class` associato a questa classe che si vuole istanziare; altrimenti solleva un'eccezione, poiché non viene specificato il tipo attuale restituito con `Class<?>`.

In questo modo è possibile istanziare un oggetto di tipo scelto a tempo dinamico. Poiché restituisce un oggetto classe, di tipo generico sconosciuto, è necessario effettuare un downcast al tipo voluto, a tempo di compilazione viene sollevato un'avvertenza "Unchecked Cast Warning", poiché il compilatore non è in grado di definire chiaramente il funzionamento del codice.

Questi avvertimenti sono dovuti alla lasca tipizzazione, e alla tardiva implementazione dei generics nel linguaggio. Per cui il compilatore deve mantenere la retrocompatibilità e deve emettere avvertimenti legati ai tipi, poiché non sono presenza garanzie sui tipi. A tempo di compilazione quindi non è possibile determinare se il codice solleverà un'eccezione, solo a tempo di esecuzione è possibile venga sollevata un'eccezione `ClassCastException`. L'avvertimento emesso dal compilatore indica quindi che l'uso dei tipi non abbastanza stringente da permettergli di offrire garanzie a tempo statico.

La retrocompatibilità infatti costringe le versioni pre-generics delle classi a convivere con e versioni propriamente generiche.

Quando viene creato un nuovo oggetto utilizzando il metodo `newIstance()`, viene invocato un costruttore vuoto della classe, che quindi deve essere visibile, ma esistono anche meccanismi per invocare costruttori con parametri. Ma da Java 7 in poi non è più necessario che sia definito esplicitamente un costruttore vuoto.

Se il codice è scritto considerando che a tempo di esecuzione sia presente il tipo giusto, è possibile utilizzare la sintassi `throws Exception`, per ignorare le avvertenze sollevate a tempo di compilazione. Se invece il nome della classe è sbagliato, o la classe non esiste, l'API di questi due metodi considera una serie di possibili eccezioni.

La definizione di un criterio di equivalenza è complicato quando appartengono ad una gerarchia di tipi. Per risolvere questo problema viene utilizzata la riflessione per codificare semplicemente controlli sul tipo dinamico di un oggetto.

Questo può essere necessario quando si utilizzano collezioni che mischiano gerarchia di tipi diverse tra di loro, quindi all'interno del metodo `equals()` si può introdurre un controllo sul tipo dell'oggetto passato come guardia:

```
@Override
public boolean equals(Object o){
    if(o == null || o.getClass() != this.getClass()) return false;
    // resto del metodo
}
```

Per cui è opportuno includere anche la classe nella funzione di hash, avente una suo funzione di hash di default che è possibile utilizzare:

```
@Override
public int hashCode(){
    return this.getClass().hashCode() + /* ... */;
}
```

In questo modo due oggetti di tipi diversi sono sicuramente diversi. In questo modo si possono realizzare gerarchie totali e disgiuntive, dove non vengono mai istanziati oggetti che non sono sottotipi della radice, poiché estensioni della stessa. Allo stesso modo di possono realizzare gerarchie parziali, dove esistono oggetti istanziati dalla superclasse, ma non dalla sottoclasse. Questi oggetti avranno campi nulli, e per definire una funzione di hash bisogna affidarsi al metodo `java.util.Objects.hash()` che può gestire riferimenti nulli. Ma utilizzando questo metodo questi campi sono cablati nei metodi di equivalenza della superclasse, e non può essere sovrascritto da altre

classi della stessa gerarchia. La definizione di un criterio di equivalenza deve quindi tenere conto dell'intera gerarchia dei tipi. Questo però rappresenta un forte ed intrinseco accoppiamento tra porzioni distinte di codice. Inoltre non esiste un modo automatico per definire criteri di equivalenza di tipi polimorfi. In generale si preferiscono soluzioni semplici, infatti scegliere criteri di equivalenza complessi e molteplici per una stessa gerarchia rappresenta un problema di modellazione di tipi. Quindi il criterio del controllo del tipo dinamico va utilizzato solamente se ritenuto adatto nel dominio. La definizione di un opportuno criterio di equivalenza rappresenta un importante problema di modellazione.

Molto spesso inserire un criterio di equivalenza non reca danni, anzi se viene usato in futuro rappresenta un investimento. Ma comporta anche un certo costo, per cui la scelta della scrittura dei criteri di equivalenza per una classe dipende da vari fattori, quali il suo utilizzo attuale, e potenziale futuro.

6 Gestione delle Eccezioni

Da Java in poi ogni linguaggio di programmazione presenta una gestione degli errori molto diversa rispetto ai linguaggi antecedenti. In ogni linguaggio di programmazione la gestione delle eccezioni è necessario, poiché si presentano costantemente anomalie nel codice di produzione. Gli errori più frequenti sono causati da un'implementazione scorretta o a causa di un errore logico. Alcune di queste anomalie possono essere talmente resilienti da mantenersi in ogni versione del codice.

A causa di queste anomalie è possibile trovarsi in uno stato inconsistente, ovvero non più rappresentativo delle istanze nel dominio che si intende modellare.

Per cui a seguito dell'installazione di un software, è sempre richiesto un aggiornamento per risolvere questi "bug", rilevati dopo la commercializzazione del servizio.

Alcuni di questi errori possono essere causati dall'ambiente esterno al programma, anche molto diversi tra di loro, la loro gestione rappresenta uno dei più grandi problemi nella produzione di un software. Le gestioni presenti in Java rappresentano un metodo molto utile e efficace per gestire la complessità di questo problema.

Le eccezioni sono uno strumento supportato da alcuni linguaggi, per gestire questo tipo di anomalie, poiché sono appunto non catalogabili per definizione, non è possibile gestirle in modo elegante, e Java produce una distinzione efficace tra l'esecuzione normale ed i casi "eccezionali".

I programmi si possono descrivere come interazioni tra gli oggetti "client", che invocano i servizi, ed oggetti "server", che offrono i servizi.

In Java sono necessari meccanismi per "sollevare" e "catturare" le eccezioni, dal lato server, la gestione delle eccezioni ricade al lato client. Tutte le eccezioni sono oggetti estensioni della classe `Throwable`, e possono essere lanciati o sollevati, utilizzando l'istruzione `throw`, a cui succede un sottotipo di `Throwable`. Queste eccezioni possono essere decorate da altri messaggi e riferimenti ad altre eccezioni che l'hanno causato, inseriti nel costruttore dell'eccezione. In questo modo si crea uno "stack trace", una traccia di tutte le chiamate che hanno portato all'eccezione.

Ogni eccezione essendo un oggetto viene creata dall'operatore `new`, poi l'oggetto lanciato con l'istruzione, inserendo nel costruttore dell'eccezione un messaggio diagnostico. La clausola `throws` viene utilizzata per marcare i metodi che possono sollevare le eccezioni, seguito dal nome dell'eccezione che può sollevare. Quando viene sollevata un'eccezione, il metodo che l'ha lanciata termina l'esecuzione, senza eseguire il `return`, senza restituire alcun valore, e viene restituito il controllo al metodo invocante. Rappresenta un comportamento molto distruttivo, interrompendo ed abbandonando una normale esecuzione. Queste eccezioni "bucano" lo stack, causano una terminazione del metodo che le lanciano, e viene restituito il controllo al metodo invocante, che può esercitare vari metodi, se in tutto lo stack non viene catturata l'eccezione, questa risale lo stack iterativamente fino al metodo `main()` dove la JVM termina l'esecuzione e stampa lo stack trace catturando l'eccezione. Per catturare l'eccezione senza che arrivi al livello superiore sono possibili due tipi di gestioni alternative. Se il client si prende il compito di catturare l'eccezione, oppure non prova a gestire l'eccezione e lascia che siano i metodi di livello superiore a catturarla.

Per catturare un'eccezione e gestirla specificatamente bisogna inserire il corpo che potrebbe sollevare l'eccezione dentro `try`, e se si vuole gestire l'eccezione, se viene catturata e conservata nella variabile locale `e` e gestita dal blocco `catch`:

```
try{
    // codice che può sollevare un'eccezione
}catch (<tipo-errore> e){
    // codice che gestisce l'eccezione
}
```

Questa gestione è molto più macchinosa rispetto ad una gestione senza cattura dell'errore. All'aumentare della complessità del programma risulta necessario includere codice per la gestione di eccezioni. Alcune eccezioni devono essere corrette a livello logico, senza catturarle come le `NullPointerException`, che vengono sollevate automaticamente a tempo di esecuzione, complicando lo stack trace, sprecando risorse ed offuscando la comprensione dell'origine vera del problema.

Quando viene catturato l'errore si può creare un oggetto dedicato al caso in cui si sia verificato un errore, per gestirlo. Specialmente nelle applicazioni dove vengono creati oggetti in modo riflessivo, quando viene generata un'eccezione di tipo `ClassCastException`, o simili nel corpo del `catch{}` viene istanziato questo oggetto, per restituire un valore che non sia un riferimento nullo. In generale viene usato un null object nei metodi che restituiscono oggetti quando viene sollevata un'eccezione.

In questo modo si ha il vantaggio di mantenere una gestione uniforme degli errori come per l'esecuzione normale del codice, invece di utilizzare `null`, infatti per versioni di Java moderne esistono modi per evitare completamente l'uso di `null`, ritenuto ormai poco elegante. Questo consiste in un metodo di gestione diretta dell'errore, altrimenti viene propagato al chiamante "bucando" lo stack

Bucare lo stack è utile quando si vogliono gestire le eccezioni al livello giusto, senza provare a catturarle ad ogni livello. Questo tipo di propagazione è opportuna quando non è stato fornito un input opportuno, per cui si risale fino al livello che ha fornito questo input per gestire l'errore. Quindi è parte integrante delle attività di progettazione la definizione di nuove eccezioni per gestire tipi di errori propri al programma.

Esistono due diversi tipi di eccezioni, tutte estensioni `Throwable`, le eccezioni `Exception` e gli errori `Error` che per loro natura non possono essere catturati e recuperati. Le eccezioni si dividono in due categorie quelle "controllate" `Checked` e quelle incontrollate, a tempo di esecuzione `Unchecked`. Le eccezioni controllate richiedono una gestione esplicita da parte del compilatore in uno dei due modi descritti precedentemente, altrimenti non viene compilato il file. Questa rappresenta una forte imposizione da parte del compilatore. Mentre le eccezioni non controllate, non richiedono una gestione esplicita da parte del client, per cui vengono sollevate a tempo di esecuzione e non costringono codice aggiuntivo per gestirle. I metodi che generano un'eccezione non controllata possono gestire l'eccezione, ma non è dichiarato esplicitamente, mentre semanticamente quelle controllate sono parte integrante della segnatura poiché appartengono ai parametri di un metodo. Mentre le eccezioni non controllate non modificando la segnatura dei metodi la loro gestione non è necessaria.

Nei linguaggi successivi a Java sono presenti solo eccezioni non controllate, poiché vengono viste come troppo invasive. Per cui anche in Java esistono dei metodi che effettuano la stessa operazione di altri metodi, senza sollevare eccezioni controllate.

Il blocco `try-catch` può gestire più di un'eccezione con più `catch` diverse e separate.

Il metodo `Class.forName` solleva un'eccezione quando il costruttore della classe non ha un costruttore, non è pubblico, oppure se non trova il `.class`. Poiché la catena di `catch` viene eseguita

dall'alto verso il basso, la prima che cattura l'eccezione, oscura le successive eccezioni sollevate nel codice. Per cui è sempre favorito ordinare le clausole della `catch` dalla più specifica in poi.

Da Java 7 venne introdotta una clausola per abbreviare la complessità di questo metodo, per effettuare una disgiunzione delle possibili eccezioni, con un'unica variabile locale per raccogliere l'eccezione ed un codice comune per la loro gestione. Quindi il tipo statico di questa variabile è il tipo statico del primo supertipo comune tra di loro, poiché in Java invece di C non esiste l'unione. Inoltre è stato introdotto un modo per gestire le risorse in tre fasi più abbreviato. Una risorsa rappresenta un qualsiasi oggetto con protocollo di utilizzo che prevede un metodo di rilascio esplicito. Questo viene gestito dalla clausola `try-with-resources`, al posto di scrivere esplicitamente il metodo per chiudere la risorsa, o terminare il suo uso, per renderlo disponibile al resto del programma, tramite la clausola `finally`. Come risorsa si intende qualsiasi oggetto con un protocollo di utilizzo che richiede un rilascio esplicito. Questo blocco di codice rappresenta ciò che viene sempre eseguito dopo il corpo di `try`.

In generale si solleva un'eccezione quando si incontra una condizione anomala, o il client ha inviato argomenti errati. Si evita di utilizzare eccezioni solo per fornice risultati. Se un metodo non rispetta il contratto allora è opportuno sollevi un'eccezione, se l'anomali che è stata lanciata si ritiene debba essere gestita dal client, si lancia una checked. Per ogni distinta condizione anomala che si potrebbe incontrare si definisce una nuova eccezione o si riutilizza un'eccezione già presente.

Da JUnit quattro in poi si possono controllare le eccezioni sollevate a tempo di esecuzione, e si possono testare. In questo modo si può controllare il comportamento corretto del codice anche in caso di situazioni anomale.

7 Classi Astratte

L'estensione è uno strumento utile per riusare il codice. Si possono estendere classi "concrete", dotato di un corpo, ereditato dalle classi estese, che possono sovrascrivere i metodi offerti dalla classe base. Questo può essere effettuato tramite interfacce, ma queste non possiedono corpo e tutte le sue implementazioni devono definire ogni metodo segnato nell'interfaccia, anche se classi estese presentano una medesima implementazione tra di loro.

Si può utilizzare quindi una classe astratta, come intermedio tra interfaccia e classe concreta, infatti contiene implementazione di alcuni metodi, mentre altri metodi vengono solo dichiarati. Quindi tutte le classi che la estendono condividono l'implementazione della classe astratta, e possono implementare i metodi astratti. La classe astratta non può essere inizializzata direttamente, come le interfacce, poiché non presenta tutte le implementazioni dei metodi, deve essere quindi estesa da altre classi per costruzione.

Convenzionalmente il nome di una classe astratta viene preceduto dal prefisso "Abstract". Le classi astratte non possono essere quindi inizializzate.

Nella classe astratta viene inserito tutto ciò che deve essere comune alle sue classi estese, per condividere le implementazioni di alcuni metodi definiti nella classe astratta. Nel caso di progetti semplici è consigliabile l'utilizzo di interfacce, poiché utilizzare classi astratte aumenta notevolmente l'accoppiamento tra le classi.

Per indicare che una classe o un metodo è astratta viene utilizzata la parola chiave `abstract`. Per una classe può essere omessa questa marcatura, ma ogni metodo che manca di corpo deve essere specificato sia astratto.

Talvolta viene definita astratta una classe che presenta tutte le implementazioni, ma non vuole essere istanziata. Se non vengono implementati i metodi astratti con un `@Override`, il compilatore produce un errore, specificando che i metodi vanno implementati oppure la classe deve essere resa astratta. La presenza di anche un solo metodo astratto necessita della dichiarazione dell'intera classe come astratta. Le sottoclassi quindi per essere concrete ed istanziabili devono fornire un'implementazione di tutti i metodi astratti forniti dalla superclasse.

Servono a definire implementazioni parziali, utilizzate nella creazione dei framework per permettere l'implementazione da altri programmatori. Spesso si preferiscono le interfacce alle classi astratte poiché sono meno vincolanti. Spesso conviene definire un'estensione concreta di una classe astratta solo ai fini di testing, utilizzando un'implementazione minimale dei metodi astratti, testando solamente i metodi concreti della superclasse.

Metodi e classi possono essere marcati come finali tramite la parola chiave `final` che impedisce alle classi di essere estese, oppure di sovrascrivere i metodi segnati come finali, impedendo al metodo di essere ridefinito. Il motivo principale per cui esistono questi metodi finali è rappresentato da problemi di sicurezza, analizzati alla creazione di Java e alla sua seguente diffusione sul web, per impedire a chiunque di poter ridefinire metodi.

7.1 Costanti Enumerative

Un'uso opportuno delle classi astratte sono le classi enumerative, introdotte in Java 5, in breve chiamati Java Enum. Utilizzati per permettere di modellare in modo conveniente. I Java Enum

sono un modo molto concreto per utilizzare le classi astratte e finali appena descritte. Sono molto utili quando si vuole realizzare collezioni molto stabili nel tempo, di alta specificità.

Per creare queste collezioni stabili prima di Java 5, si utilizzavano controlli a tempo statico stringenti, per evitare una tipizzazione lasca delle variabili, tramite una classe finale. Oppure si può realizzare tramite un'interfaccia contenente come variabile finali costanti i valori della collezione, ma questo non permette di controllare la correttezza dei suoi valori tramite una tipizzazione lasca. Per evitare questo problema quindi vengono utilizzati costruttori privati, poiché gli elementi sono noti a priori e stabili nel tempo, per evitare all'utilizzatore di creare o modificare questi elementi noti.

Prima di Java 5 la modellazione di costanti enumerative consisteva spesso di gerarchia di costanti di tipo lasco. Queste implementazioni erano semplici, ma l'uso di una tipizzazione può causare problemi a tempo di compilazione ed esecuzione, inoltre può rendere le segnature di certi metodi di difficile interpretazione. Per realizzare costanti enumerative di questo tipo si utilizzavano membri statici segnati come `final`, si considera un esempio di un'implementazione delle direzioni cardinali:

```
public interface Direzione{
    static final public int NORD = 0;
    static final public int OVEST = 1;
    static final public int SUD = 2;
    static final public int EST = 3;
}
```

Per risolvere questi problemi è richiesto come minimo creare un tipo dedicato, per poter effettuare dei controllo a tempo statico sulla tipizzazione. Questo veniva effettuato utilizzando costruttori privati e classi `final`. Si considera allora l'esempio precedente sulle direzioni cardinali implementando queste modiche:

```
public final class Direzione{
    private int ordinal;
    private Direzione(int Index) { this.ordinal = index; }
    static public final Direzione NORD  = new Direzione(0);
    static public final Direzione OVEST = new Direzione(1);
    static public final Direzione SUD   = new Direzione(2);
    static public final Direzione EST   = new Direzione(3);
    static private final Direzione[] cardinali = { NORD, EST, SUD, OVEST};
    static public DIrezione[] values() { return this.cardinali; }
}
```

Inoltre avendo un elenco di tutti gli elementi di questo insieme è possibile effettuare cicli `for-each`, affiancando metodi polimorfi alle costanti, ma questo necessita di creare un sottotipo per ogni costante della stessa enumerazione. Un possibile metodo polimorfo, da applicare sulle direzione cardinali è il metodo che restituisce la direzione opposta. Dall'implementazione precedente bisogna definire una serie di sottoclassi contenenti questo metodo polimorfo, tipicizzando la classe `Direzione` come astratta, per permettere di definire questi metodi. Si realizzano come classi interne in grado di utilizzare il metodo privato definito precedentemente:

```
public abstract class Direzione{
    private int ordinal;
    private Direzione(int Index) { this.ordinal = index; }
    public int getOrdinal() { return this.ordinal; }
    abstract public Direzione Opposta();
    static public final Direzione NORD  = new NORD();
    static public final Direzione OVEST = new OVEST();
    static public final Direzione SUD   = new SUD();
    static public final Direzione EST   = new EST();
    static private final Direzione[] cardinali = { NORD, EST, SUD, OVEST};
    static public DIrezione[] values() { return this.cardinali; }
    static final class NORD extends Direzione {
        private NORD() { super(0); }
        @Override public Direzione opposta() { return SUD; }
    }
    static final class NORD extends Direzione {
        private OVEST() { super(1); }
        @Override public Direzione opposta() { return EST; }
    }
    static final class NORD extends Direzione {
        private SUD() { super(2); }
        @Override public Direzione opposta() { return NORD; }
    }
    static final class NORD extends Direzione {
        private EST() { super(3); }
        @Override public Direzione opposta() { return OVEST; }
    }
}
```

Da Java 5 venne introdotta la parola chiave `enum` per realizzare queste collezioni:

```
public enum <nome-enum>{
    <nome-1>, ... , <nome-n>;
}
```

Si può realizzare quindi l'insieme di direzioni cardinali, facilmente utilizzando questa sintassi:

```
public enum Direzione{
    NORD, OVEST, SUD, EST;
}
```

Crea quindi $n-1$ classi, tutte con metodi condivisibili da tutte le classi enumerative. Inoltre il compilatore aggiunge metodi statici che ritiene utili, come `ordinal()`. Quando sono presenti metodi polimorfi bisogna definirli nel corpo della classe, seguiti dalla definizione del metodo astratto:

```
public enum <nome-enum>{
    <nome-1>(){
        @Override public <tipo> <nome-metodo>() { }
    },
    ...
    <nome-n>(){
        @Override public <tipo> <nome-metodo>() { }
    };
    public abstract <tipo> <nome-metodo>();
}
```

Si può implementare quindi un metodo per la direzione opposta delle funzioni cardinali come:

```
public enum Direzione{
    NORD()  { @Override public Direzione opposta() { return SUD;  } }
    OVEST() { @Override public Direzione opposta() { return EST;  } }
    SUD()   { @Override public Direzione opposta() { return NORD; } }
    EST()   { @Override public Direzione opposta() { return OVEST; } };
    public abstract Direzione opposta();
}
```

Ogni classe enum `<nome-enum>` presenta come sottoclassi tutte le n `<nome-1>` definiti dentro al corpo della dichiarazione dell'enum:

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E> { }
```

Ognuna di queste classi viene numerata, partendo da zero, e può essere restituito dal metodo `ordinal()`. Il compilatore utilizza un unico costruttore per realizzare le costanti enumerative:

```
Enum protected Enum(String name, int ordinal);
```

I valori costanti possono avere uno stato, immutabile, ed uno o più costruttori privati, dichiarati solo nel corpo del tipo enumerativo. Le costanti ricordano la classe del loro supertipo mediante il metodo:

```
Class<E> getDeclaringClass()
```

Inoltre ad ogni valore costante viene associato un singolo oggetto, equivalente solo a sé stesso e nessuno degli altri. Il criterio di equivalenza realizzato tramite `equals()` corrisponde all'operatore `==` solo per le costanti enumerative. Mentre la funzione di hash restituisce `ordinal()`, e rappresenta una funzione di hash "perfetta", poiché riesce ad ottenere le migliori prestazioni, non avendo conflitti. I tipi enumerativi quindi permettono di avere accessi a tempo costante sulle mappe, paragonabili ad un array. Per cui si possono realizzare collezioni di caratteristiche particolari utilizzando tipi enumerative, più compatte ed efficienti, come `java.util.EnumSet` e `java.util.EnumMap`.

Gli enum possono essere utilizzati insieme al comando `switch`, anche se dopo Java 11 vennero introdotti costrutti che ne ampliarono notevolmente l'uso, sulla base del successo che hanno avuto linguaggi funzionali.

I costrutti enum vengono utilizzati quando bisogna modellare collezioni di elementi noti, finiti e stabili, è molto difficile che cambieranno. Inoltre questi elementi sono conosciuti per nome. Possono anche avere uno stato ma questo deve essere immutabili.

Alcuni metodi definiti per i tipi enumerativi non hanno un codice sorgente, come i metodi statici `values()` ed `valueOf()`, poiché il compilatore li genera automaticamente per il tipo enumerativo e per tutte classi che estendono `java.lang.Enum`.

7.2 Classi Nidificate

Le classi nidificate furono introdotte prestissimo, in Java 1.1, l'unica grande novità di quest'aggiornamento. Il vero motivo per cui si ebbe questa necessità fu il supporto per scrivere interfacce grafiche. Da Java 8 vennero introdotti meccanismi che ebbero più successo per risolvere questo problema.

Questo meccanismo venne introdotto nel linguaggio Java, ma non venne introdotto nella macchina virtuale. Per cui ormai la macchina virtuale di Java viene usata anche per altri linguaggi di programmazione poiché è fortemente disaccoppiata dal linguaggio stesso. Quindi la macchina virtuale anch'essa ha avuto un enorme successo. Una classe nidificata è una classe scritta all'interno di un'altra classe, per cui si individua una classe "interna", ed una classe "esterna", la classe interna è un membro contenuto nella classe esterna, quindi essenzialmente è un membro privato. Esistono quattro diverse varianti di una classe nidificata, una classe interna di cui si interessa solamente il tipo. Classi locali vengono dichiarate all'interno di un blocco di codice, ad uso e consumo di un metodo. Classi anonime sono classi di cui non serve un nome. Infine le classi statiche.

La grossa differenza tra le classi non statiche e le non statiche consiste nella creazione che è legato fortemente all'istanza della classe esterna a partire dal quale è stato creato. Ogni istanza della classe interna deve mantenere un riferimento alla classe che l'ha istanziata, così come la classe esterna mantiene un riferimento all'istanza della classe interna che ha istanziato.

QUando viene creata un'istanza della classe interna rimane collegato all'istanza della classe esterna, senza questo legame l'oggetto istanza della classe interna non può esistere, questo legame è reciproco. Inoltre ogni istanza di una classe interna è associata a solo un'istanza di una classe esterna.

Poiché la classe interna è un membro della classe esterna, può accedere agli elementi privati della classe esterna, inoltre se la classe interna è pubblica, può essere istanziata dall'esterno, e possono essere invocati i suoi metodi, che potrebbero utilizzare membri della classe esterna.

Per cui solamente attraverso la classe esterna è possibile invocare un riferimento alla classe interna. La sintassi per la creazione di una classe interna quindi riflette questo legame:

```
Outer outer = new Outer();       // costruttore classe esterna
Outer.inner = outer.new Inner(); // costruttore classe interna
```

Dal punto di vista dei tipi non appartengono alla stessa gerarchia.

Dentro la classe interna è possibile introdurre variabili omonime a variabili di istanza della classe esterna, per evitare confusione e accedere bisogna qualificare completamente il nome con la sintassi:

```
<nome_Outer>.this.<nome_variabile>
```

Un iteratore rappresenta una classe nidificata interna, alle collezione che lo comprendono.

8 IO Stream

Oltre allo sviluppatore esistono altri ruoli per modificare il codice prima di spedirlo al cliente.

La gestione dell'IO viene effettuata tramite il pacchetto `java.io.*`, primo pacchetto per la gestione dell'IO, per cui è datato, ma tuttora popolare. Una delle astrazioni più importanti fornite da questo pacchetto è il flusso o "stream", una collezione di dati a cui è possibile accedere solamente sequenzialmente. Usato in Java 8 per descrivere un modo alternativo per gestire un flusso di dati. Si parlerà solamente di stream nell'ambito della gestione del flusso di dati da dispositivi di IO.

Un flusso viene definito da una sorgente o destinazione, ed un programma. Se si tratta di un flusso di input, permette di leggere i dati prodotti da una sorgente, mentre se si tratta di un flusso di output, permette di scrivere dati diretti ad una destinazione. I flussi ben prima dell'avvento del paradigma orientato agli oggetti viene la necessità di astrarre la gestione di IO su diversi dispositivi diversi tra di loro, già presenti all'avvento di Java nei sistemi Unix.

Uno stream può essere di caratteri o di byte. UUno stream di caratteri fa riferimento ad un certo linguaggio naturale, e quindi deve decifrare quei byte rispetto ad un alfabeto, un encoding, come l'ASCII. Ma avendo solamente 256 caratteri disponibili, essendo diffuso su internet, Java ebbe la necessità di supportare linguaggi diversi. Utilizzo una classe per codificare 60000 caratteri diversi, abbastanza per tutti gli alfabeti di lingue usate. Java odierno permette la codifica di tutti gli alfabeti noti, e non più usati, oltre a quelli già citati.

Esistono due stream di caratteri i `Reader` e `Writer` rappresentati da classe astratte, aventi sottotipi per molti tipi di sorgenti di destinazioni, per stringhe `StringReader` e `StringWriter`.

La gerarchia per la gestione dell'IO ha come radice queste due classi astratte, interpellando i byte letti come caratteri attraverso un meccanismo di encoding di uno o più byte, da un insieme di possibili caratteri "charset". La codifica Java ha avuto esigenze di andare ben oltre la codifica ASCII, per cui utilizza una variante della codifica UTF-16 per il charset "Unicode", per permettere di coprire gli alfabeti più diffusi, composta da una codifica a 16 bit. Altre codifiche supportate presentano 4 byte, per lingue occidentali contenenti alfabeti molto più ampi.

Storicamente vi era la necessità di comunicare con hard disk a blocchi, per cui in Java sono presenti astrazioni per leggere a blocchi gli stream `Buffer`.

La classe `InputStreamReader` prende una stream di caratteri ed una specifica di una codifica, da Java 18 viene utilizzata come default la codifica UTF-8, utilizzata anche da molti sistemi operativi, poiché è retrocompatibile con la codifica ASCII. In Java 18 hanno effettuato un cambiamento non retrocompatibile, usando sempre l'encoding UTF-8, indipendentemente dalla preferenza del sistema operativo. Per questo si lavora con stream di caratteri, senza specificare la codifica usata.

Il costruttore di `InputStreamReader` necessità della codifica specificata come stringa, altrimenti viene utilizzata la codifica di default, prima di Java 18 quella del sistema operante. I principali metodi sono pochi e primitivi per leggere caratteri su ogni tipo di dispositivo. Per cui sono metodi di basso livello come `read()` e `writer()` entrambi sovraccarichi, pre cui tendenzialmente vengono usate altre classi che utilizzano questi metodi internamente.

Nel pacchetto `java.io` sono presenti funzionalità di decorazione, che utilizzano un riferimento ad un `Reader` o `Writer` e ne adornano le funzionalità. Il design pattern di decorazione consiste che siano tutti sottotipi di `Reader`, ma presentano un riferimento ad un altro `Reader` iniettato nel loro costruttore, di cui offrono una versione arricchita.

Questo modo di decorare l'interfaccia ha un vantaggio chiaro, permette di comporre le funzionalità all'atto di instanziamento dell'oggetto, senza dover creare una nuova classe per ogni nuova combinazione di funzionalità richiesta.

Gli stream rappresentano risorse, rispettano quindi una fase di utilizzo ed una fase esplicita di rilascio per rendere in condizione il sistema operativo di condividere la risorsa con altri programmi. Ciò non vale con ogni risorsa, ma alcune necessitano di un meccanismo di rilascio esplicito, altrimenti il S.O. non permette di rilasciare la risorsa ad altri programmi. Per cui il protocollo di utilizzo degli stream, e di altre risorse simili, necessita di un'esplicita richiesta alla risorsa, il suo utilizzo, ed un'esplicito rilascio della risorsa. Per cui nell'uso di basso livello viene utilizzata la sintassi `try-catch-finally` per assicurarsi di rilasciare le risorse utilizzate alla fine dell'esecuzione, indipendentemente se vengono sollevate eccezioni a tempo di esecuzione.

Questo pacchetto contiene delle scelte datate e discutibili come la classe `java.io.File` che confonde l'indirizzo nel S.O. del file dal file stesso, per cui in Java 4 venne introdotta un pacchetto per introdurre la classe `Path` per risolvere il problema descrivendo l'indirizzo del file. Ora sono disponibili singoli metodi di utilità nella classe `java.nio.file.Files` per risolvere il problema introdotta da `java.io.File`. Questo, insieme a librerie esterne, permette di rendere molto più semplice la gestione di lettura e scrittura su file.

Possono essere quindi usate classi `Builder` per realizzare un'oggetto leggendo i parametri e le variabili da istanziare da un file, tramite una classe `Parser` per leggere il contenuto del file.

Per cui oltre allo sviluppatore, è necessario il lavoro di definire e costruire i livelli associati a diversi file, che verranno letti dal programma realizzato dallo sviluppatore. Viene necessario quindi un amministratore in grado di gestire il sistema e configurarlo, prima di consegnarlo all'utente finale.

Per cui il codice deve essere predisposto all'accesso da questi amministratori senza necessità di conoscere il linguaggio Java. Per cui è spesso opportuno spostare le informazioni sul programma all'esterno del codice sorgente.

Poiché il programma non viene utilizzato in un ambiente di sviluppo, ma viene semplicemente eseguito dall'utente finale. Questo meccanismo di "esternalizzazione" delle costanti permette all'amministratore di configurare le necessarie informazioni per rendere il programma utilizzabile dall'utente finale. Sono disponibili a livello industriale framework standardizzati per realizzare queste attività, in modo molto semplice ed efficiente.

Questi parametri configurazione possono essere rimossi dal codice tramite meccanismi artigianali, per evitare anche di ricompilare il codice dopo aver cambiato il valore di alcune costanti.

Per conservare dentro file testuali valori di configurazione è possibile utilizzare file che permettono la lettura ed il salvataggio di oggetti tramite il pacchetto `java.util.Properties`, sottotipo di `Map` da Java 2 in poi. Rappresenta una mappa di coppie chiave-valore, di tipo stringhe. La chiave rappresenta il nome della "properties", mentre il valore rappresenta il suo valore. Un formato che va per la maggiore è il formato `.json` per scrivere file contente oggetti, reso famoso da Javascript, tanto che sono presenti tante librerie dentro Java per leggere e scrivere in formato Json. Questi file testuali sono editabili da non programmatori Java. La classe `Properties` presenta dei metodi semplici, che permettono di realizzare una proprietà copiandola da un'altra proprietà, inoltre è possibile conservare il valore da una stream all'interno di una proprietà, attraverso i metodi `store()`, i metodi `storeToXML()` permettono di conservare queste proprietà all'interno di un file di formato

`.xml`, ma ormai è stato molto ridimensionato l'utilizzo di file di tipo XML. I file di tipo XML sono una generalizzazione del formato HTML, dove i tag sono scelti arbitrariamente dallo sviluppatore.

Bisogna evitare di specificare all'interno del codice la direzione assoluta del file proprietà, poiché cabla nel codice un'indirizzo che molto probabilmente funzionerà solamente per il singolo sviluppatore che lo ha specificato. Il codice deve quindi poter eseguire nell'ambiente di esecuzione finale, ormai all'interno del cloud. Il rilascio su queste piattaforme deve quindi evitare di includere errori del genere. Per utilizzare una risorsa senza utilizzare l'indirizzo logico completo vengono utilizzate due classi `Class` e `ClassLoader`, per poter leggere le risorse e le classi. Entrambe le classi offrono due metodi `getResource()` che prende il nome della risorsa come stringa, e restituisce un tipo URL, mentre il metodo `getResourceAsStream()` che prende lo stesso nome come input e restituisce una stream dalla lettura della risorsa specificata, in modo da caricarla in modo più efficiente. Anche se presentano la stessa segnatura, hanno una semantica diversa, che rappresenta una scelta progettistica discutibile, ma effettivamente effettuano lo stesso lavoro.

Per la risoluzione dei nomi logici delle risorse, viene effettuato allo stesso modo della risoluzione del nome completamente qualificato delle classi, nella posizione del corrispondente file `.class`. Questo meccanismo di risoluzione viene copiato analogamente per le risorse per rendere il loro indirizzo noto alla JVM a tempo di esecuzione. Bisogna rendere noto nel "buildpath" l'indirizzo di due altre cartelle, ormai solo una, chiamata `Resources` da dove parte la risoluzione del nome logico delle risorse, tramite le classi precedentemente specificate. I nomi relativi non cominciano per '/', mentre i percorsi assoluti cominciano per '/' e dipendono dalla locazione della cartella dove comincia l'esecuzione del programma, per cui è preferibile utilizzare un indirizzo relativo. La classe `Class` risolve i percorsi relativi partendo dalla posizione del file `.class` associato all'oggetto corrispondente associato. Mentre la classe `ClassLoader` lavora solamente tramite percorsi relativi, partendo dal classpath.

Quando viene impacchettato un progetto creando un file `.jar` bisogna poter eseguire il codice, ma essendo impacchettato e compresso dentro un archivio, il file di risorse non è accessibile come un file ordinario, quindi invece di utilizzare il metodo `getResource()` che restituisce un indirizzo logico per accedere alla risorsa come ad ogni altro file nel file-system, bisogna utilizzare il metodo che restituisce una stream che permette il caricamento delle risorse partendo da un archivio dove le risorse sono compresse. Questo permette al programma di essere indipendente dall'ambiente di esecuzione.

9 Java Thread

La programmazione concorrente è un argomento che necessita di un intero corso per la sua complessità, molto elevata rispetto agli altri argomenti presenti nei corsi introduttivi di programmazione, ma sono conoscenze strettamente necessarie nell'ambito informatico.

Utilizzando la programmazione concorrente è possibile aumentare le prestazioni di un programma, tramite calcolatori a più core, ormai anche in ambito mobile, per utilizzare al meglio l'intero processore.

Sotto ogni applicazioni grafica esiste un modello di concorrenza per la loro notevole complessità algoritmica, anche se non si programmano più applicazioni grafiche in Java.

Java venne introdotto come un sistema compatto per sistemi real-time, quindi offriva un supporto per la programmazione concorrente, anche se durante gli anni venne riscritto parecchie volte per essere ristrutturato ai bisogni del tempo. Il supporto del linguaggio a questo approccio esiste dagli anni '90, e venne esteso da varie librerie, di cui quattro tra le più diffuse verranno analizzate.

Il flusso di istruzioni generato da un programma sequenziale aspetta il termine di un'istruzione prima di eseguire la successiva, mentre nella programmazione concorrente le istruzioni vengono eseguite parallelamente. Questo viene effettuato utilizzando i "thread", un meccanismo di parallelismo interno alle architetture multiscalari.

Esistono dei metodi in Java per controllare le caratteristiche dell'architettura come il numero dei core disponibili, oppure il doppio dove ogni core abilita l'"hyperthreading" che permette un flusso di esecuzione doppio per ogni core, nei processori Intel e AMD di fascia alta moderni.

Chi scrive software ha sempre bisogno di prestazioni migliori, e chi realizza hardware viene richiesto di realizzare calcolatori di prestazioni migliori. Per cui questi due ambiti si rincorrono l'uno all'altro, ma ormai non è più possibili aumentare la frequenza di clock dei calcolatori per migliorare le prestazioni semplicemente. Inoltre ogni nuovo processore include un numero sempre maggiore di transistor, seguendo un andamento pressoché lineare, descritto dalla legge di Moore, anche se anche sotto questo ambito si sta riscontrando un limite tecnologico. Per migliorare le prestazioni non bisogna solamente aumentare le prestazioni del calcolatore, ma anche degli altri componenti di un calcolatore come la memoria.

A fine anni '90 le prestazioni del linguaggio Java erano minori rispetto ad altri linguaggi come C e C++, ma si capì che ben presto le prestazioni dei processori avrebbero reso questa differenza superflua.

Ma anche se sono presenti architetture multiscalari, non è sempre immediato aumentare le prestazioni del codice o dividere il lavoro sui diversi processori logici. Infatti se non viene specificato, quando vengono eseguiti programmi di calcolo intensivi, tutto il carico viene affidato generalmente ad un singolo core.

Per ottenere il meglio da un'architettura multi-core bisogna decomporre un problema in modo che sia eseguibile in parallelo. Questo problema di decomposizione è molto semplice per alcuni problemi, come per le applicazioni grafiche, mentre per altre applicazioni è pressoché impossibile realizzarlo. La decomposizione parallela si ottiene scomponendo un problema in tanti sotto-problemi, i cui risultati parziali possono essere ottenuti indipendentemente e possono essere combinati o ricomposti per ottenere il risultato al problema iniziale.

Da Java 1.0 esistono supporti per la programmazione parallela, tramite i java "Thread", un flusso di esecuzione che si prende l'incarico di mandare in esecuzione il codice, e condivide la memoria con altri flussi di esecuzione. Il thread creato per eseguire il metodo main viene anch'esso chiamato Main. In teoria la JVM supporta la possibilità di creare più thread, oltre al Thread Main. I thread sono contrapposti ai processi, un'unità di esecuzione autonoma delle risorse, che non condivide la memoria con gli altri processi. La loro capacità di condividere memoria, se non viene gestita efficientemente potrebbe rappresentare un collo di bottiglia. Infatti la memoria è il limite superiore alle prestazioni, per la programmazione parallela; ogni thread richiede una sua allocazione di memoria per poter definire uno stack.

Il numero di thread generati all'esecuzione di main aumenta all'aumentare della versione di Java

Due o più thread utilizzano stack di attivazione diversi, e sono in grado di condividere memoria con gli altri, avendo accesso agli stessi oggetti, basta che tutti abbiano un riferimento comune.

Si gestiscono utilizzando una classe `java.lang.Thread`, un oggetto che viene istanziato, ma non coincide con il flusso di esecuzione. Questo oggetto rappresenta uno strumento di comunicazione tra il programmatore e la JVM, in grado di generare un thread chiamando in metodo nativo `start()`. Verrà invocato il prima possibile dopo l'invocazione del metodo `start()`. La classe inoltre specifica il codice da eseguire sul thread e aspettare che un flusso di esecuzione termini. La JVM è un intermediario tra il programmatore ed il sistema operativo.

Il metodo `run()` che non prende alcun argomento dall'interfaccia `Runnable`, in modo da poter scrivere cosa bisogna eseguire in modo concorrente. Quindi la classe i cui metodi devono essere eseguiti in modo concorrente bisogna che implementi questa interfaccia.

Un nuovo thread creato deve avere il riferimento all'oggetto `Runnable` per eseguire il codice all'interno del metodo `run()` ed eseguire il flusso delle esecuzioni. Quindi senza passare un riferimento al `Runnable` non è possibile realizzare un flusso di esecuzione parallela. Inoltre il thread così generato deve invocare un flusso di esecuzione tramite il metodo `start()`.

```
class NewThread implements Runnable{
    @Override
    public void run(){
        /* operazioni nuovo thread */
    }
}

public class MainThread{
    public static void main(String[] args){
        Thread t = new Thread(new NewThread());
        t.start();
        /* codice thread main */
    }
}
```

La chiamata al flusso di esecuzione `start()` termina immediatamente e passa il controllo all'istruzione seguente nel metodo che ha chiamato `start()`. Sotto carico è possibile che il nuovo thread venga eseguito prima del Thread Main, per cui sotto carico il comportamento del codice

parallelo non è deterministico, poiché dipende dal sistema operativo, in ordine imprevedibile. Si tratta quindi di un programma multithread anche se utilizza solo due thread.

Si dice assunzione di progresso finito la non prevedibilità di quale thread venga eseguito prima degli altri. Quando il metodo `run()` termina, il thread viene segnato come terminato, e verrà in seguito deallocato.

Per aspettare le soluzioni parziali da parte dei thread invece si utilizza il metodo `join()`. Ha una semantica complementare al metodo `start()`, il thread invocante aspetta il termine dell'esecuzione del thread su cui è stato chiamato il metodo prima di continuare la sua esecuzione. Questi thread pongono la soluzione parziale nella memoria condivisa, e per assicurare che tutte le soluzioni parziali siano state calcolate si posizionano tutti i metodi `join()` e `start()` in un altro thread dedicato alla raccolta di questi risultati parziali. Poiché potrebbe essere chiamato un `join()` su un thread già eseguito, la macchina virtuale deve tenere traccia anche di thread già "morti". Questo metodo deve essere poter onorato indipendentemente dal fatto che il thread è già terminato o meno. Quindi si distinguono i thread normali da thread su cui non può essere chiamata la `join()`, per cui possono essere rimossi dopo aver terminato la loro esecuzione. Per cui paradossalmente è la `join()` che termina la traccia di un thread, invece che il termine dell'esecuzione dello stesso.

La decomposizione parallela prevede tre passi generale, per il primo si creano tanti thread quanti sono i core disponibili, il problema iniziale viene diviso in tanti sotto-problemi ed ognuno viene assegnato ad un thread distinto, che ne calcoli il suo risultato parziale. Infine i risultati parziali sono collezionati per formare il risultato globale.

Le schede grafiche moderne, contengono anche migliaia di core distinti, per cui si prestano particolarmente a problemi di decomposizione parallela. Ma nel caso di GPU di alte prestazioni, il collo di bottiglia sono i bus che lo connettono alla CPU ed alla memoria, poiché anche se sono in grado di eseguire molte operazioni contemporaneamente, il tempo di trasferimento dei dati dalla memoria rappresenta il tempo maggiore speso durante la risoluzione del problema.

In generale ogni problema di decomposizione parallela prevede un overhead per generare ed inizializzare i vari thread, e dividere i dati iniziali, codice la cui esecuzione rimane seriale.

I problemi su array si prestano alla decomposizione parallela, poiché sono una struttura dati la cui scomposizione è estremamente efficiente, e molti problemi di ordinamento e ricerca possono essere divisi in sotto-problemi semplicemente. Se l'array non può essere diviso equamente tra tutti i thread, lo sfrido viene distribuito tra i primi thread. Ogni thread risolve il sotto-problema e ne calcola la sua soluzione relativa.

Se questa divisione non viene effettuata correttamente, è possibile che i vari thread sovrascrivano valori tra di loro, poiché condividono la stessa memoria.

In caso un thread venga fermato a tempo di esecuzione, allora viene sollevata un'eccezione `InterruptedException` che deve essere gestita altrimenti il codice non compila.

I problemi di decomposizione parallela si possono dividere due categorie, i problemi dove i dati di input sono di ordine di grandezza molto elevati, e problemi i quali per ogni elemento in input è richiesto un elevato capacità di calcolo.

In generale lo "speed up" è un fattore che descrive quante volte il programma parallelo diminuisce il tempo di esecuzione rispetto al programma seriale, sullo stesso problema. Questo fattore non necessariamente corrisponde esattamente al numero di core disponibili. L'andamento di questo

fattore è asintotico, si avvicina al numero di core disponibili all'aumentare del peso computazionale del problema.

Inoltre è possibile estendere direttamente `Thread`, anche se viene sconsigliato, poiché tende ad unire i compiti di `Thread` e `Runnable`, questo quindi aumenta l'accoppiamento del codice.