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 su scala industriale, un processore virtuale, poiché non viene progettato per un calcolatore fisico, ma la piattaforma Java Virtual Machine (JVM), per cui 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`. Questo bytecode 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, 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 ad un medio livello di programmazione, di alto livello rispetto all'assembly, e permette di realizzare programmi ad alte prestazioni, utilizzato per scrivere sistemi operativi. Java invece venne introdotto in un ambiente diverso rispetto al linguaggio C, per cui si indica come un linguaggio di alto livello di astrazione, non è un linguaggio che cerca le migliori prestazioni, ma cerca di creare programmi semplici e portabili. Nel periodo in cui venne introdotto Java l'andamento delle prestazioni dei calcolatori permise di realizzare linguaggi che non avessero come scopo principale il risparmiare risorse del sistema. Il linguaggio C++ permette di avere prestazioni simili al linguaggio C, ma introducendo paradigmi realizzati in Java. Ma si preferisce per realizzare programmi ad alte prestazioni il C, rispetto al C++, poiché fornisce troppe scelte, complicando l'implementazione e l'uso del linguaggio.

In Java una dichiarazione di una variabile comprende la sua inizializzazione ad un valore nullo. 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, in appoggio al compilatore per favorire la loro gestione da parte del compilatore. 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 `""" """`.

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 i 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, in molte piattaforme, viene fornito un errore estremamente generico, in base alla versione di C, del compilatore, e da altre condizioni.

1.2 Paradigma Orientato agli Oggetti

Il paradigma procedurale, usato nel linguaggio C, separa in maniera netta tra 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 di programmazione che seguono questo paradigma. 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 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 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, con estensione ".java". Le operazioni di oggetti in Java si indicano come 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

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 chiamato `origine`.

Tramite questo riferimenti è possibile accedere ai metodi, ed eventualmente allo stato dell'oggetto, tramite la notazione puntata, che segue la sintassi `<riferimento-oggetto>.<metodo>(<parametri-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.

Utilizzando due riferimenti allo stesso oggetto per effettuare azioni sullo stesso oggetto si chiama effetto collaterale.

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`.

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.

```
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;
    }
}
```

Al termine dell'esecuzione del costruttore, l'oggetto non è stato completamente inizializzato

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.

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)`.

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.

1.5 Costruttori, 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 la lista dei parametri formali, per il numero, per il tipo di uno o più, oppure per il loro ordine.

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

```
    }
    public int add(int a, int b, in 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 eseguito 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 le stringhe. 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. In seguito gli altri costruttori vengono definiti in base a questo costruttore, invocando questo costruttore, 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);
    }
}
```

In Java esiste la classe `String` per poter rappresentare sequenze di caratteri immutabili, rappresenta un riferimento ad un oggetto istanza della classe `String`. 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 creare 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, 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>`.

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 `pubblic`, 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 facce della stessa medaglia.

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

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 criteri di correttezza del codice, 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 `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()`.

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.

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. 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".

3 Interfacce e Polimorfismo

3.1 Upcasting e Downcasting

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 classe che implementa un'`interface` è un suo sottotipo, mentre l'`interface` 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 chiamato "upcasting". Il collegamento tra l'assegnatura e l'implementazione di un `interface` 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.

L'overloading dei metodi avviene a tempo di compilazione, quindi staticamente, ma in questo modo non tiene conto dei metodi dei tipi dinamici.

Le invocazioni polimorfe si risolvono a tempo di esecuzione sulla base del tipo dinamico da parte della macchina virtuale, l'unica componente in grado di avere informazioni sul tipo dinamico. Mentre l'overloading di un metodo viene risolto a tempo di compilazione dal compilatore, sulla base del tipo statico dei parametri passati al metodo.

3.2 Estensioni di Interface e Classi

Date due interfacce diverse, aventi gli stessi metodi, anche se concettualmente sembrerebbero uguali, non è possibile referenziare un oggetto che implementa una delle due interfacce con un oggetto che implementa l'altra, poiché il linguaggio è staticamente tipato. Per risolvere questo problema è viene fornita la parola chiave `extends <nome-interfaccia>` per specificare che un'interfaccia è sottotipo di un'altra interfaccia, 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. 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 membri 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 b 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 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 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".

3.3 Classe Object e Overriding

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. Quasi sempre quindi conviene ridefinire questi metodi sulla base dello specificità della classe definita. Essendo tutte le classi estensioni di questa classe predefinita, ogni costruttore contiene implicitamente una chiamata al costruttore alla classe `Object`. Generalmente quando si ridefinisce un metodo fornito dalla classe `Object` bisogna effettuare un downcast al tipo della classe necessaria nel metodo.

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.

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`.

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

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. 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, quindi 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 generato un eccezione a tempo di compilazione.

In Java la gerarchia delle classi ha una radice definita, la radice di Object è quindi `java.lang.Object`, e rappresenta la radice predefinita della gerarchia delle classi. Al contrario di altri linguaggi di programmazione in Java, al contrario di C++, non è possibile avere una derivazione multipla tra classi, una class può estendere al più una sola altra classe, e può avere zero o più sottoclassi. 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, poiché 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 una 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, spesso invece non è nota 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 seguito si inserisce in ogni sottoclasse un'implementazione di questo metodo come override che invoca questo metodo sovraccarico, passando sé stesso con la parola chiave `this`, tipo statico il tipo della classe o il 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 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 `< >`, che racchiude il nome del tipo formale. Questo tipo viene usato come una dichiarazione di tipo, quindi può essere usato all'interno della classe come instanziamento di tipo. Solo quando viene creato il tipo bisogna assegnare un tipo al tipo formale, se non vengono forniti tutti i tipi formali il codice non compila.

Quando si uas una classe generica, bisogna istanziarne completamente i tipi fornendo i tipi attuali di tutti i tipi formali di cui fa uso, solo in questo modo può essere utilizzata come un tipo.

Il concetto tra parametro attuale parametro formale è simile alla relazione tra tipo formale e tipo attuale.

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.

Le classi wrapper sono definite nel pacchetto `jav.lang`, per cui non è necessario importarle, esplicitamente. 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.

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. 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 quale tipo attuale rappresenta il tipo formale si considera la regola PECS,

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 "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. Viene contenuto nel pacchetto `java.util`, contiene collezioni non specialistiche, tra le più utilizzate in assoluto.

Il Java Collection Framework (JCF), contiene implementazioni interfacce, 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 negli insiemi, non ammettono duplicati e gli elementi non possiedono una posizione.

Dentro questa interfaccia sono presenti quindi 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, verificare la dimensione o se è vuota, 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.

L'interfaccia `Map<K,V>` offre le operazioni di una mappa, o dizionario; è una collezione di coppie chiave-valore.

La classe `java.util.Collections`, offre un vasto insieme di metodi statici generici,

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

- Manipolazione di base;

- Bulk;
- 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` per il tipo passato. 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`. Il metodo `add` restituisce un booleano, vero se e solo se l'aggiunta ha generato una modifica della collezione.

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 `Collections<?>`. Tutti questi i metodi sono lascamente tipati, eccetto `addAll`, che richiede una collezione di tipo `<E>` o di un suo sottotipo.

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. 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, per ogni collezione si generava un modo di enumerare gli elementi della collezione. 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 sono creati invocando il metodo `iterator()` sulla collezione.

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.

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.

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. 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 forniscono 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.

4.3 Liste

Le liste sono strutture dati astratte, con agginunta 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.

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.

I costruttori di queste due implementazioni sono sovraccarichi. Entrambi hanno un costruttore vuoto, ed un costruttore che prende come parametro una collezione di un 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.

Per `LinkedList` sono presenti dei metodi per accesso in testa e coda, poiché avviene in tempo costante, tramite riferimenti alla testa ed alla coda.

Nella classe `Collections` sono presenti vari metodi efficienti per ordinare la lista, per effettuare una ricerca binaria, cercare il massimo o minimo nella lista. 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>`, naturale o interno, oppure ad una classe esterna agli oggetti contenuti tramite l'interfaccia `java.util.Comparator<T>`. La prima interfaccia consiste nell'unico metodo `public int compareTo(T that)`, e 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 un tipo o un suo supertipo.

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`:

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

Per gli stessi motivi del precedente metodo. Semplificato in:

```
public static <T extends Comparable<? Super T>> T min/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.

Se si ha una gerarchia dei tipi, è possibile implementare l'interfaccia `Comparable<T>` solamente su un unico tipo, poiché 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` che riferiscono a due tipi diversi, creando confusione. Per lo stesso motivo non è possibile generare 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/max(Collections<? extends T> c, Comparator <? super T> comp)
```

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 aventi lo stesso codice hash saranno certamente equivalenti, 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 bene 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. 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 li metodo `hashCode()` per verificare se non è presente alcun numero 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.

Se non venisse implementato il metodo `hashCode()`, si erediterebbe quello definito da `java.lang.Object`.

4.4.2 TreeSet

L'implementazione più popolare degli alberi ordinati, viene realizzato tramite alberi ordinati, attraverso i metodi di ordinamento naturali ed esterni, definiti precedentemente per le liste ordinate.

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 uci 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 occasione, verrà sollevato un eccezione, poiché gli elementi non possono essere ordinati.

Se viene definito un criterio di ordinamento, implicitamente viene definito un criterio di equivalenza. Si definiscono i metodi di ordinamento in accordo con i metodi `hashCode()` e `equals()`.

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. `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.

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.

I metodi base per accedere, aggiungere e rimuovere elementi sono `put()` che richiede una chiave ed un valore di tipo `K` e `V` rispettivamente, oppure aggiorna il vecchio valore e lo restituisce, se la

chiave era già usata. 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 oltre a `putAll()` e `clear()` usati per aggiungere o eliminare tutte le coppie in una mappa, mentre 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`, offerto dal tipo `values()`.

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,?>`.

Per definire il criterio di equivalenza sulle chiavi

5 Introspezione

In Java esiste una classe `java.lang.Class<T>`, classe di tipo classe. Questa classe viene usata tramite introspezione. In questo modo è possibile scrivere metodi `equals()` ed `HashCode()` tra oggetti di una gerarchia di tipi.

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 per conoscere tutti i metodi annotati tramite `@Test`, e le classi. Prima delle riflessioni, il runner di JUnit richiedeva tutti i nomi dei metodi e considerava come test tutti i metodi preceduti dal termine "test".

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.

Tutti i tipi possiedono una variabile statica pubblica chiamata `class`, corrispondente all'oggetto di tipo `java.lang.Class`. 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.newIstanze();
```

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 `newIstanza()` 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<?>`.

Altrimenti a tempo di compilazione viene sollevato un'avvertenza "Unchecked Cast Warning".

Ora non è più necessario utilizzare un costruttore vuoto quando viene istanziato un nuovo oggetto utilizzando introspezione, ma per Java 7 viene utilizzato sempre un costruttore vuoto. Per cui se il codice è scritto considera 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.

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 decorate da altri messaggi e riferimenti ad altre eccezioni che l'hanno causato. 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.

Utilizzando un null object, 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 sono esistono modi per evitare completamente l'uso di `null`, ritenuto ormai poco elegante.

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`. Questo blocco di codice rappresenta ciò che viene sempre eseguito dopo il corpo di `try`.