Tecniche Speciali di Programmazione

Massimo Perego

Indice

1 Introduzione

L'idea iniziale è quella della **separazione dei compiti**: non vogliamo inquinare il codice ma separare ogni funzionalità implementata, rendendola indipendente e riutilizzabile, ovvero scrivere codice in cui ogni funzionalità è a sè stante.

1.1 Computational Reflection

La **reflection** può essere definita come: *attività svolta da un sistema software per rappresentare e manipolare la propria struttura e/o comportamento.* Un software che, in qualche modo, si modifica.

Qualche definizione (secondo Pattie Maes):

- Un **sistema computazionale** è un sistema in grado di ragionare e agire in un ambito applicativo
- Un sistema computazionale è **causalmente connesso** al suo dominio se e solo se un cambio del dominio si riflette sulla computazione stessa e viceversa
- Un **meta-sistema** è un sistema computazionale il cui dominio applicativo è un altro sistema computazionale
- La **reflection** è la proprietà di ragionare e agire rispetto a sé stessi

Di conseguenza: un **sistema riflessivo** è un meta-sistema causalmente connesso a sé stesso.

Dalla definizione si può evincere che:

- Un sistema riflessivo deve essere organizzato in più livelli: definiti **base-level** e **meta-level** (rispettivamente, osservato e osservante)
- Il sistema in esecuzione nel meta-level osserva e manipola il sistema in esecuzione nel livello sottostante (**reflective tower**)

Caratteristiche:

- L'entità osservata è inconsapevole di essere osservata (non ha awareness); il base-level non è a conoscenza dei livelli superiori, il controllo è trasparente all'applicazione: anche se cambia il meta-livello tutto ciò che è sopra rimane come prima, gli "accessori" possono cambiare senza che il sistema sotto se ne accorga; questa caratteristica (unawareness) è fondamentale

- Un sistema nel meta-level agisce su una rappresentazione (**reification**, reificazione) del sistema sottostante
- Un sistema e la sua reificazione sono causalmente connessi e, di conseguenza, mutualmente consistenti

I sistemi riflessivi possono essere classificati in base a *what and when*: quando e cosa viene modificato dal sistema.

Cosa viene modificato, ovvero che tipo di azione riflessiva viene svolta:

- Fare introspezione (osservare) oppure intercedere (modificare)
- Le modifiche possono essere strutturali e/o comportamentali

Quando fa riferimento al momento in cui esistono entità nel meta-level:

- **Compile-time**; non si hanno informazioni dinamiche (quindi si possono fare meno cose), ma con a disposizione il sorgente fare modifiche è "facile"
- **Load-time**; prima che la classe venga mai istanziata, ovvero la classe viene caricata "già modificata", non bisogna andare a cercare le entità esistenti; il sorgente rimane uguale, ma so cosa viene caricato, meno invasivo del compile-time, ma più difficile
- **Run-time**; il più difficile, completamente dinamico, ci sono oggetti già istanziati

La **riflessione comportamentale** permette di modificare e monitorare la computazione: i metodi chiamati, come e quando viene istanziata una classe; tutto ciò che fa parte del comportamento del programma. Alcuni esempi:

- Catturare una chiamata a metodo e attivarne un altro
- Monitorare lo stato di un oggetto
- Creare nuovi oggetti
- …

Tutte queste attività possono essere effettuate run-time senza supporto specifico aggiuntivo (si hanno già a disposizione le informazioni necessarie).

La **riflessione strutturale** permette di ispezionare o alterare la struttura del programma. Ad esempio

- Il codice di un metodo può essere modificato o rimosso dalla classe
- Nuovi metodi e ampi possono essere aggiunti
- …

Queste attività richiedono *specifico supporto dall'ambiente di esecuzione* (in generale più facile con un linguaggio interpretato, difficile in Java ad esempio, in Python si ha tutto il codice a disposizione anche a run-time).

Ognuna delle entità nel base-level, chiamate **referents/referenti**, ha un corrispondente nel meta-livello, quest'ultimi vengono manipolati dal meta-livello stesso.

Il corrispondente all'interno del meta-livello (reificazione) deve

- Supportare tutte le operazioni e avere le stesse caratteristiche del referente corrispondente
- Essere consistente con il referente (connessione causale)
- Essere manipolate dalle entità del meta-livello, per evitare inconsistenze con il referente nel base-level

Il meta-livello modifica i referenti e aspetta di avere dei referenti stabili.

Alcune domande da tenere in considerazione:

- Quali entità dovrebbero essere reificate? Dipende dal linguaggio di programmazione (funzioni se è funzionale, oggetti se è OO e così via), tutto ciò che compone di fatto il sistema (fino, ad esempio, ai messaggi per gli scambi di metodo)
- Come e quando viene implementata la connessione causale? Dipende da quando prende piede la riflessione
  - Run-time: la connessione causale è esplicita e deve essere mantenuta da entità super-partes, un meccanismo dinamico parte del run-time environment
  - Compile-time: la connessione causale è implicita, base e meta-levels sono uniti durante la fase di pre-processamento; le modifiche dovute a meta-livello sono "fuse" grazie al pre-processing
  - Load-time: in questo caso la connessione causale si comporta come a compile-time, con la differenza che verrà caricata dopo

Più si va verso il dinamismo più serve un meccanismo esterno che mantenga il sistema

- Quando l'esecuzione passa al meta-level? Cambiare tra i livelli dipende da quali entità vengono reificate e quando ciò avviene, assieme al modo in cui la connessione causale è gestita. Il cambio è realizzato logicamente con due azioni:
  - **Shift-up:** Il cambio al meta-livello avviene quando un oggetto osservato cambia, oppure un'azione sta per essere svolta; in generale quando serve un cambiamento
  - **Shift-down:** La computazione torna al base-level in seguito a una decisione esplicita (al contrario dello shift-up) del programma nel meta-livello

Gli oggetti in esecuzione nel meta-livello (meta-oggetti) sono associati a uno o più referenti nel base-level. I meta-oggetti esistono a run-time e estendono o modificano semantiche o meccanismi (invocazione di metodi, accesso ai campi, creazione oggetti, …).

Il **MOP** (**Meta-Object Protocol**) è un insieme di istruzioni (stile API, per dare un'idea) fornito dalla meta-entità per manipolare il sistema sottostante. Il set di messaggi che un meta-oggetto può comprendere.

A livello base non viene aggiunto nulla per le funzionalità accessorie, tutto ciò che è aggiuntivo viene inserito nel meta-livello, senza legarsi specificatamente a una classe sottostante (per esempio, un log deve poter loggare tutto, non solo una classe specifica).

Ogni entità a livello base è coordinata da una entità nel meta-livello, la corrispondenza non è necessariamente 1 a 1 (ogni oggetto nel meta-livello può coordinare più oggetti, ad esempio group call: chiamare più target dello stesso tipo con una singola chiamata, ma anche il contrario, un oggetto base controllato da più meta-oggetti, ognuno dei quali controlla aspetti diversi).

Ci sono più **modelli** possibili per fare **reflection**. Il **primo modello** è quello in cui i *meta-oggetti sono una classe*, la meta-classe gestisce una classe specifica, in modo da agire su tutte le istanze della classe controllata.

Le classi svolgono la riflessione, la torre riflessiva viene dal legame di ereditarietà. Tutte le istanze della classe condividono lo stesso comportamento riflessivo (la meta-classe) è unica per tutte le istanze.

Secondo modello: *meta-classi e meta-oggetti*. Sono ancora presenti meta-classi che definiscono il comportamento della classe, ma si possono avere anche meta-oggetti (con la loro classe) associati a una specifica istanza. Il link di riflessione diventa esplicito e specifico.

Nel **terzo modello** il meta-oggetto non è associato agli oggetti stessi ma a una comunicazione, si ha una *reificazione dei messaggi*. Si avrà un meta-oggetto per ogni singolo metodo scambiato tra coppie di oggetti. Questo permette granularità molto fine, a costo di un'esplosione nel numero di meta-oggetti.

Permette una visione globale sulla comunicazione, rendendo facili attività riflessive sulla comunicazione stessa. La vita di queste entità potrebbe anche essere molto breve: create e distrutte assieme al messaggio.

Di fatto, la riflessione computazionale

- Apre un sistema per deferire alcune decisioni
- Dipende dalla awareness che il sistema ha di sé, il codice sottostante non è aware del fatto che sarà esteso tramite riflessione
- Specializza alcuni dei meccanismi base dell'OO (costruttori, chiamate, …), sfrutta tutti i meccanismi classici dell'OOP (in maniera non specifica)

L'obiettivo è una migliore comprensione dei meccanismi OO e della loro implementazione.

1.2 MOP and SoC

1.2.1 Meta-Object Protocol MOP

L'idea del MOP è mettere a disposizione uno strumento che permette di estendere il sistema sottostante, senza conoscerlo. L'idea è quella di un'interfaccia comune per tutte le casistiche in cui si vuole estendere qualcosa di esistente. Questa è l'idea di riflessione computazionale.

Di conseguenza, si vuole mantenere un uso black-box del sistema base, il cui comportamento e struttura possono essere modificati dinamicamente, lasciando aperti al sistema del meta-livello i dettaglio dell'implementazione del base-level.

Il MOP permette di mantenere nascosti i dettagli implementativi, andando a interagire in maniera diversa da quella originariamente pensata dal codice

"modificato" (grey-box).

Ci sono 3 modi per aprire un sistema:

- **Introspezione:** solamente guardare, abilità del sistema di osservare lo stato e struttura del sistema stesso
- **Intercessione:** modificare anche comportamento e struttura del sistema stesso
- **Invoke:** abilità del sistema di applicare le funzionalità del sistema (non si può aggiungere codice, intercessione ridotta)

1.2.2 Separation of Concerns SoC

Un'applicazione si può dividere in

- **Core functionality**: le funzionalità principali del sistema, gli "obiettivi" (e.g., per un sistema bancario: account, operazioni, …)
- **Nonfunctional concerns:** tutte le funzionalità accessorie (cross cutting concerns), tutte le funzionalità necessarie per il sistema, ma non parte della "base" dell'applicazione (e.g. ciò che concerne sicurezza, distribuzione, concorrenza, …)

Un sistema è composto da entrambe le cose, le si vuole separarle per migliorare riusabilità e ridurre inquinamento del codice.

Tradizionalmente, la SoC viene considerata solo durante la fase di design, e il codice diventa un mix di tutti i concerns.

Il gap tra design e implementazione, porta a poca riusabilità e estensibilità, oltre che essere error prone. La SoC punta a permettere la separazione anche durante l'implementazione (reflection, aspect-oriented programming).

La reflection permette di separare aspetti functional (core) e nonfunctional, portando a migliore riusabilità, migliore stabilità del sistema e dando la possibilità di sviluppare indipendentemente aspetti diversi.

2 Reflection in Java

La **Java Core Reflection API** fornisce una API piccola, type safe e sicura per quanto riguarda l'introspection per classi e oggetti nella JVM.

In generale, ci sono un po' di limitazioni su cosa permette la reflection (policy, moduli di sicurezza nelle nuove versioni), non è mai stato un focus primario per Java, ma è sempre stata utilizzata.

Se permesso dalla security policy, la API può essere usata per:

- Costruire nuove istanze di classe e nuovi array
- Accedere e modificare campi di oggetti e classi
- Invocare metodi
- Accedere e modificare elementi di un array

Intercessione su classi e oggetti è proibita.

Ci sono applicazioni della suite Java che fanno uso delle reflection, come:

- Documentazione automatica (javadoc, …)
- Tool per IDE: browser, inspectors, debuggers; molti sono scritti in Java e fanno uso della reflection
- Serializzazione/deserializzazione: costruire una rappresentazione binaria per backup o trasmissione, ricreare un oggetto a partire dalla sua forma serializzata
- RMI Remote Method Invocation: i dati usati per chiamare metodi da remoto di fatto vengono serializzati per chiamare dall'altro lato il metodo vero

Nelle varie versioni di Java le classi e interfacce disponibili per la reflection sono cambiate, da Java 13 in avanti le classi e interfacce coinvolte sono:

- `java.lang.Object`, superclasse di ogni classe in Java, tutto è `Object`
- `java.lang.Class`, classe che rappresenta le classi e le interfacce di Java come oggetti, ogni oggetto "porta con sé" la propria `Class` (da un oggetto `p`, `p.getClass()` permette di ottenere un oggetto `Class` che descrive la struttura di `p`)

In breve, `Object` è la base di tutti gli oggetti runtime, mentre `Class` è la base di tutti i meta-oggetti ("descrive" gli oggetti).

2.1 `java.lang.reflect`

La libreria `java.lang.reflect` contiene

- `java.lang.reflect.Member`, interfaccia comune a campi, metodi e costruttori. Permette di sapere cose come nome (`getName()`), classe dichiarativa (`getDeclaringClass()`) e i modificatori (`getModifiers()`, ad esempio `public`, `private`, …)
- `java.lang.reflect.AccessibleObject`, permette di controllare l'accessibilità di un membro (`isAccessible()` restituisce se l'oggetto è accessibile tramite reflection), ma anche di modificarla (`setAccessible(boolean flag)`). Si tratta della classe base per
  - `java.lang.reflect.Field`, rappresenta un attributo di una classe, permette di agire su un campo, permette di leggere e agire su tipo, nome e modificatori del campo. Ad esempio, `getName()`, `getType()` e `getModifiers()` permettono di sapere tipo, nome e modificatori del campo, mentre `get(Object obj)` e `set(Object obj, Object value)` permettono di accedere al valore del campo
  - `java.lang.reflect.Method`, rappresenta un metodo, permette di leggerne le informazioni (`getName()`, `getReturnType()`, `getParameterTypes()`, `getModifiers()`, …) ed eventualmente invocarlo dinamicamente su un'istanza o sulla classe stessa se è `static`
  - `java.lang.reflect.Constructor`, rappresenta un costruttore dichiarato in una classe, permette di ottenere informazioni sul costruttore stesso (`getName()`, `getParameterTypes()`, `getModifiers()`, …), ma anche di istanziare nuovi oggetti
- `java.lang.reflect.Proxy`, permette di creare oggetti Proxy dinamici, ovvero un oggetto che implementa una o più interfacce specificate e intercetta tutte le chiamate ai loro metodi, che vengono inviate a un `InvocationHandler`
- `java.lang.reflect.InvocationHandler`, interfaccia che gestisce le invocazioni dei metodi sui proxy dinamici, ha un solo metodo astratto `invoke(Object proxy, Method method, Object[] args) throws Throwable`. Permette di modificare la logica prima o dopo l'invocazione del metodo, o anche modificare l'esecuzione a seconda delle condizioni, senza andare a toccare il metodo concreto

Si hanno anche

- `java.lang.annotation.Annotation`, interfaccia radice per tutte le annotazioni in Java, permettendo anche reflection sulle annotazioni
- `java.lang.instrument.Instrumentation`, interfaccia che permette di intercettare e modificare bytecode delle classi durante il caricamento nella JVM

In Java, il modello di reflection è quello a **meta-oggetti**, anche se solo predefiniti, ovvero ci sono meta-oggetti generici per il ruolo del campo, per il ruolo di Metodo, …

Limiti della reflection in Java:

- Il MOP non è causalmente connesso: togliendo l'intercessione non c'è rischio di modifica e conseguenti inconsistenze all'interno del bytecode Java; viene considerata un rischio
- `Class` è dichiarata come `final`, non possono essere create nuove meta-classi
- Non ci sono operazioni MOP per modificare le classi, quindi non si può facilmente creare e modularizzare trasformazioni da classe a classe

Le class-to-class transformation esistono, ma sono usate dagli sviluppatori della stdlib, non da chi programma in Java (`Cloneable`, `Remote`, `Serializable`, sono trasformazioni class-to-class, ma un programmatore Java non può crearne di altre).

2.1.1 `Object`

`Object` definisce metodi alla quale rispondono tutti gli oggetti, "radice" della gerarchia, metodi di default che si potrebbero volere in una classe.

```
1  class Object {
2      public final Class<?> getClass() { ... }
3      protected Object clone() { ... }
4      public boolean equals(Object obj) { ... }
5      public int hashCode() { ... }
6      public String toString() { ... }
7      public final void notify() { ... }
8      public final void notifyAll() { ... }
9      public final void wait() { ... }
10     ...
11 }
```

Non rilevante per fare reflection.

2.1.2 `Class`

La classe `Class` mette a disposizione gli elementi principali per qualsiasi attività riflessiva.

```java
public final class Class<T> extends Object {
    public static Class<?> forName(String className) { ... }
    public static Class<?> forName(Module module, String name)
        { ... }
    // from Java 22
    public static Class<?> forPrimitiveName
        (String primitiveName) { ... }

    public T newInstance() { ... } /* deprecated since 9 */
    public boolean isInstance(Object obj) { ... }

    public String getName() { ... }
    public Class<? super T> getSuperclass() { ... }
    public Module getModule() { ... }
    public Class<?>[] getInterfaces() { ... }

    public Class<?>[] getDeclaredClasses() { ... }
    public Method[] getDeclaredMethods() { ... }
    public Annotation[] getAnnotation() { ... }
    public Constructor<?>[] getConstructors() { ... }
    public Field[] getFields() { ... }

    public boolean isAnnotation() { ... }
    public boolean isArray() { ... }
    public boolean isEnum() { ... }
    public boolean isRecord() { ... }
    ...
}
```

Esempio di metodo per tornare il nome di una classe in maniera stampabile:

```
class MOP {
    public static String classNameToString(Class<?> cls) {
        if (!cls.isArray()) return cls.getName();
        return cls.getComponentType().getName() + "[]";
    }
}
```

Torna il nome della classe sotto forma di stringa, gestendo correttamente gli array.

Esempio di uso:

```
jshell> MOP.classNameToString(String.class)
$2 ==> "java.lang.String"

jshell> var a = new Integer[]{1, 2, 3}
a ==> Integer[3] {1, 2, 3}

jshell> a.getClass()
$4 ==> class [Ljava.lang.Integer;

jshell> MOP.classNameToString(a.getClass())
$5 ==> "java.lang.Integer[]"
```

Altro esempio, metodo per tornare tutta la gerarchia al di sopra di una classe:

```
import java.util.ArrayList;
import java.util.List;
class MOP {
    public static Class<?>[] getAllSuperClasses(Class<?> cls) {
        List<Class<?>> result = new ArrayList<Class<?>>();
        for (Class<?> x = cls; x != null; x = x.getSuperclass())
            result.add(x);
        return result.toArray(new Class<?>[0]);
    }
}
```

Mette in un array tutte le super-classi trovate, ogni volta andando a quella sopra tramite un loop, quando la successiva è `null`, siamo arrivati a `Object`

e non c'è niente sopra.

Esempio di uso:

```
jshell> MOP.getAllSuperClasses(java.util.ArrayList.class)
$4 ==> Class[4] {
    class java.util.ArrayList, class java.util.AbstractList,
    class java.util.AbstractCollection, class java.lang.Object
}
```

Metodi a disposizione di `Class<T>`:

Member Access	**Class Properties**	**Context Access**
`getAnnotations()`	`getComponentType()`	`getClassLoader()`
`getAnnotation()`	`getDeclaringClass()`	`getInterfaces()`
`getClasses()`	`getEnclosingClass()`	`getModule()`
`getConstructors()`	`getEnclosingConstructor()`	`getPackage()`
`getConstructor()`	`getEnclosingMethod()`	`getProtectionDomain()`
`getDeclaredAnnotations()`	`getEnumConstants()`	`getResourceAsStream()`
`getDeclaredAnnotation()`	`getModifiers()`	`getResource()`
`getDeclaredClasses()`	`getName()`	`getSigners()`
`getDeclaredConstructors()`	`getRecodComponents()`	
`getDeclaredConstructor()`	`getSuperClass()`	
`getDeclaredFields()`	`isAnnotationPresent()`	
`getDeclaredField()`	`isAnnotation()`	
`getDeclaredMethods()`	`isAnonymousClass()`	
`getDeclaredMethod()`	`isArray()`	
`getFields()`	`isAssignableFrom()`	
`getField()`	`isEnum()`	
`getMethods()`	`isHidden()`	
`getMethod()`	`isInterface()`	
	`isPrimitive()`	

Nella prima categoria sono presenti 3 "tipi" di metodi:

- Senza la `s` finale (al singolare, ad esempio `getField(String name)`), restituisce uno specifico membro identificato dal nome (parametro)
- Con la `s` finale, restituisce un array con tutti i membri accessibili
- Con `Declared`, restituisce tutti e soli gli elementi dichiarati all'interno della classe, pubblici o privati, ma non ereditati

La seconda categoria permette l'introspezione sulle proprietà della classe stessa, informazioni sulla natura della classe stessa, ad esempio `getName()` restituisce il nome completo della classe.

L'ultima categoria fa riferimento all'accesso, permette di ottenere infor-

mazioni riguardo il contesto di runtime all'interno del quale la classe esiste, ad esempio `getClassLoader()` restituisce il ClassLoader che ha caricato quella classe.

2.1.3 `Method`

La classe `java.lang.reflect.Method` permette di fare introspezione sui metodi.

```
public final class Method extends Executable
            implements AnnotatedElement, Member {
    public Class<?> getReturnType() { ... }
    public int getParameterCount() { ... }
    // Inherited by Executable
    public Parameter[] getParameters() { ... }
    public Class<?>[] getParameterTypes() { ... }
    public Class<?>[] getExceptionTypes() { ... }
    public Object invoke(Object obj, Object... args)
        throws IllegalAccessException, IllegalArgumentException,
        InvocationTargetException { ... }
    ...
}
```

Il metodo `invoke` permette di eseguire dinamicamente un metodo su un oggetto, gestendo boxing e unboxing automatico tra primitivi e wrapper e restituisce il risultato come `Object`

2.1.4 `Parameter`

La classe `java.lang.reflect.Parameter` rappresenta ciascun singolo parametro formale di un metodo o costruttore, permettendo di ottenere tutte le caratteristiche di ciascun parametro.

```
public final class Parameter implements AnnotatedElement {
    public Set<AccessFlag> accessFlags() { ... }
    public String getName() { ... }
    public Class<?> getType() { ... }
    public boolean isVarArgs() { ... }
    ...
}
```

Un oggetto `Parameter` non si crea direttamente, lo si ottiene tramite `getParameters()`.

I metodi al suo interno permettono di agire sui parametri ottenuti.

Si ha un problema con l'uguaglianza: `invoke` chiede solo `Object`, quindi effettua un boxing dei parametri, questo potrebbe causare problemi nel momento in cui i valori sono confrontati tramite `==`, in quanto confronta il riferimento all'oggetto e non il valore stesso, due oggetti `Integer` diversi risulteranno sempre diversi, anche se contengono lo stesso valore (in realtà se è un intero fino a 255 esistono dei singleton per gestire lo specifico caso, da 256 potresti avere problemi).

Esempio di due metodi per restituire i parametri di un `Method`

```
class MOP {
    public static String parametersToString(Method m) {
        var mt = Arrays.asList(m.getParameterTypes());
        return IntStream.range(0, m.getParameterCount()).boxed()
            .map(i -> {
                return MOP.classNameToString(mt.get(i))
                    + " p" + (i + 1);
            }).collect(Collectors.joining(", "));
    }
    public static String formalParametersToString(Method m) {
        return Arrays.stream(m.getParameters())
            .map(p -> MOP.classNameToString(p.getType())
                + " " + p.getName())
            .collect(Collectors.joining(", "));
    }
}
```

Il primo metodo restituisce i parametri di un `Method` dato, fornendo un nome arbitrario (`p1`, `p2`, …), in quanto `getParameterTypes()` non fornisce il nome del parametro. Il secondo metodo usa `getParameters()`, che restituisce anche i nomi dei parametri, quindi si possono usare i nomi definiti all'interno del sistema al posto di quelli arbitrari.

Esempio di metodo per tornare la firma di un altro metodo

```
import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
class MOP {
    public static String headerToString(Method m) {
```

```
        String result = MOP.classNameToString(m.getReturnType())
            + " " + m.getName()
            + "(" + MOP.formalParametersToString(m) + ")";
        Class<?>[] exs = m.getExceptionTypes();
        if (exs.length > 0)
            result += " throws " + MOP.classArrayToString(exs);
        return result;
    }
}
```

Il metodo restituisce, in ordine, il tipo di ritorno del metodo (con `getReturnType()`), il nome del metodo (con `getName()`) e i parametri sotto forma di stringa (con `formalParametersToString()`) tra parentesi, se sono presenti aggiunge le eccezioni che può lanciare alla fine.

2.1.5 `Field`

La classe `java.lang.reflect.Field` ha lo stesso scopo di `Method`, ma per i campi.

```
public final class Field extends AccessibleObject
        implements AnnotatedElement, Member {
    public Class<?> getType() { ... };
    public Object get(Object obj) throws
        IllegalArgumentException, IllegalAccessException
            { ... };
    public void set(Object obj, Object value) throws
        IllegalArgumentException, IllegalAccessException
            { ... };
    public Class<?> getDeclaringClass() {...} ;
    ... // Include get* and set* for the eight primitive types
}
```

Rappresenta un singolo campo di una classe o interfaccia, permettendo di leggere, modificare o ottenere informazioni sul campo stesso (anceh se `private`).

2.1.6 `AccessibleObject`

La classe `AccessibleObject` è la superclasse astratta di `Field`, `Method` e `Constructor`.

Permette di sopprimere il controllo degli accessi quando

- vengono letti o modificati campi, tramite `Field`
- vengono invocati metodi, tramite `Method`
- vengono create e inizializzate nuove istanze di una classe, tramite `Constructor`

```
public final class AccessibleObject implements AnnotatedElement {
    public void setAccessible(boolean flag)
        throws SecurityException { ... }
    public static void setAccessible
        (AccessibleObject[] array, boolean flag)
        throws SecurityException { ... }
    public boolean isAccessible() { ... }
}
```

Con `setAccessible()` si possono indicare come accessibili (o meno) oggetti `AccessibleObject`, ma l'uso di questa può essere proibita dal Java security manager (vedi eccezione). Il modulo del membro deve essere aperto nelle policy.

In breve, permette di gestire l'accessibilità dei membri di una classe, bypassando i controlli di accesso (`private`, `protected`) a runtime.

2.1.7 `Constructor`

Similmente a `Field` e `Method`, permette di ispezionare e manipolare i costruttori a runtime.

```
public final class Constructor extends AccessibleObject
        implements Member {
    public T newInstance(Object... initargs) throws
        InstantiationException, IllegalAccessException,
        IllegalArgumentException, InvocationTargetException
            { ... }
    ...
}
```

Il `newInstance()` è pensato per chiamare il costruttore di default della classe, non fa nient'altro che chiamare il costruttore vero e proprio, con un cast.

Esempio di accesso riflessivo ai campi, tramite interfaccia

```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

public interface SmartFieldAccess {
    public default Object instVarAt(String name)
            throws Exception {
        Field f = this.getClass().getDeclaredField(name);
        f.setAccessible(true);
        if (!Modifier.isStatic(f.getModifiers()))
            return f.get(this);
        return null;
    }
    public default void instVarAtPut(String name, Object value)
            throws Exception {
        Field f = this.getClass().getDeclaredField(name);
        f.setAccessible(true);
        if (!Modifier.isStatic(f.getModifiers()))
            f.set(this, value);
    }
}
class Employee implements SmartFieldAccess {
    private String name;
    public Employee(String name) {
        this.name = name;
    }
}
```

Java 10 permette metodi di default all'interno delle interfacce. I due metodi presentati, rispettivamente, leggono e settano un campo. I due metodi leggono il nome del campo e lo restituiscono, oppure settano il valore.

Uso:

```
jshell> var mike = new Employee("Mike");
mike ==> Employee@5e9f23b4

jshell> mike.instVarAtPut("name", "Eleonor")

jshell> mike.instVarAt("name")
$7 ==> "Eleonor"
```

Clone riflessivo

```java
import java.lang.reflect.Field;

public interface ReflectiveCloning {
    // otherwise a warning is issued due to type erasure
    @SuppressWarnings("unchecked")
    public default <T extends ReflectiveCloning> T copy()
                throws Exception {
        Class<T> clazz = (Class<T>) this.getClass();
        T tmp = clazz.getDeclaredConstructor().newInstance();
        for (Field f: clazz.getDeclaredFields()) {
            f.setAccessible(true);
            f.set(tmp, f.get(this));
        }
        return tmp;
    }
}
class Employee implements ReflectiveCloning {
    private String name;
    public Employee() {
        this.name = "Anon";
    }
    public Employee(String name) {
        this.name = name;
    }
    public String toString() {
        return "Employee: " + this.name;
    }
}
```

Da notare che questo metodo non prende i campi ereditati, considera solo campi dichiarati all'interno dell'oggetto. Per risolvere, si può implementare ricorsivamente questo comportamento su tutte la gerarchia.

Togliere il `Declared` farebbe perdere i campi privati. Da tenere a mente alcuni dettagli, come ad esempio l'override dei metodi, non bisogna tenere in considerazione campi di padre e figlio con lo stesso nome, ma tenere solo quello del figlio.

2.2 Call stack introspection

L'introspezione può essere applicata un po' ovunque, non solo alla struttura della classe, come visto precedentemente. Ora vogliamo anche andare a vedere la parte comportamentale, informazioni relative all'esecuzione: execution state e call stack.

Ogni thread ha un call stack che consiste di stack frames. L'introspezione sullo stack permette di vedere "cosa è successo", ovvero il contenuto dello stack (chiamate) fino a quel momento.

In Java non c'è un meta-object per lo stack, quindi l'entry point qual'è? Quando un'istanza di `Throwable` viene creata, il call stack viene salvato come array di `StackTraceElement`. Il nostro punto di ingresso sono **le eccezioni**.

Con

```
new Throwable().getStackTrace()
```

possiamo accedere a una rappresentazione del call stack, nel momento in cui il `Throwable` è stato creato. Il metodo `getStackTrace()` restituisce il call stack attuale come array di `StackTraceElement`, in cui il primo è il frame corrente.

Da un frame si possono ottenere:

- Il nome del file contenente il punto di esecuzione, `getFileName()`
- Il numero della linea in cui avviene la chiamata al metodo, `getLineNumber()`
- Nome di metodo e classe che contengono il punto di esecuzione, `getClassName()` e `getMethodName()`

Esempio:

```
public class ABC {
    public void a() {b();}
    public void b() {c();}
    private void c() {
    for(StackTraceElement f: new Throwable().getStackTrace())
        System.out.println(f);
    }
    public static void main(String args[]) { new ABC().a(); }
}
```

E quando mandato in esecuzione:

```
> java ABC
ABC.c(ABC.java:5)
ABC.b(ABC.java:3)
ABC.a(ABC.java:2)
ABC.main(ABC.java:7)
```

Esempio per effettuare logging: un'interfaccia per il logger

```
1 public interface Logger {
2     void logRecord(String msg, int type);
3     void logProblem(Throwable prob);
4 }
```

Che può essere implementata come

```
1 public class LoggerImpl implements Logger {
2     public void logRecord(String message, int logRecordType) {
3         StackTraceElement f = new Throwable().getStackTrace()[1];
4         String callerClassName = f.getClassName();
5         String callerMethodName = f.getMethodName();
6         int callerLineNumber = f.getLineNumber();
7         // write of log record goes here.
8     }
9 }
```

Da notare che viene preso il log in posizione `1`, lo `0` rappresenta la chiamata al log stesso, vogliamo lo stack frame superiore.

Un problema risolvibile con call stack introspection è quello di controllare gli invarianti (proprietà che deve rimanere vera per tutta la vita dell'istanza). Definiamo una classe `VisiblePoint`, un punto è visibile quando all'interno dei boundaries (questo è l'invariante che vogliamo controllare).

```
1 public interface Visible {
2     public final int XMIN = -1080 ;
3     public final int XMAX = 1080 ;
4     public final int YMIN = -1920 ;
5     public final int YMAX = 1920 ;
6     default boolean isvisiblex(int x) { return (x>= XMIN && x <= XMAX); }
7     default boolean isvisibley(int y) { return (y>= YMIN && y <= YMAX); }
8 }
9
```

```
10 public class VisiblePoint implements Visible {
11     private int x, y;
12     public VisiblePoint(int x, int y) {
13         this.x = x; this.y = y;
14         assert isvisiblex(this.x) && isvisibley(this.y) :
15         "x or y coordinates outside the display margins";
16     }
17     public int getX() { return this.x; }
18     public int getY() { return this.y; }
19     public void setX(int x) { this.x = x; }
20     public void setY(int y) { this.y = y; }
21 }
```

La classe da controllare deve implementare l'interfaccia

```
1 public interface InvariantSupporter { boolean invariant(); }
```

Il metodo `invariant()` deve essere chiamato all'inizio/fine di ogni metodo

```
1  public class VisiblePoint implements Visible, InvariantSupporter {
2      ...
3      public boolean invariant() {
4          return isvisiblex(getX()) && isvisibley(getY()) ;
5      }
6      public int getX() {
7          InvariantChecker.checkInvariant(this);
8          int result = this.x;
9          InvariantChecker.checkInvariant(this);
10         return result ;
11     }
12     ...
13     public void setY(int y) {
14         InvariantChecker.checkInvariant(this);
15         this.y=y;
16         InvariantChecker.checkInvariant(this);
17     }
18 }
```

Il controllo è effettuato dalla classe `InvariantChecker`

```
1 public class InvariantChecker {
2     public static void checkInvariant(InvariantSupporter obj) {
3         if (!obj.invariant())
```

```
            throw new IllegalStateException("invariant failure");
    }
}
```

Esempio di main

```
public class MainInvariantChecker {
    public static void main(String[] args) {
        VisiblePoint p = new VisiblePoint(-7, 25);
        p.setX(-20);
        p.setX(-2000);
    }
}
```

Esecuzione:

```
> java MainInvariantChecker
Exception in thread "main" java.lang.StackOverflowError
at InvariantChecker.checkInvariant(InvariantChecker.java:3)
at VisiblePoint.getX(VisiblePoint.java:12)
at VisiblePoint.invariant(VisiblePoint.java:9)
...
```

Problema: si ha un loop su `invariant()` (e di conseguenza overflow), l'invariante chiama get e get chiama l'invariante.

Per risolvere: si possono rilevare i loop usando lo stack trace: nel momento del check viene controllato lo stack trace, se ad un certo punto all'interno dello stack trace c'è un frame che ha `InvariantChecker` e `checkInvariant` come classe e metodo chiamante, rispettivamente, allora è in loop.

```
public class InvariantChecker {
    public static void checkInvariant(InvariantSupporter obj) {
        StackTraceElement[] ste = (new Throwable()).getStackTrace();
        for (int i = 1 ; i<ste.length; i++)
            if (ste[i].getClassName().equals("InvariantChecker")
             && ste[i].getMethodName().equals("checkInvariant"))
                    return;
        if ( !obj.invariant() )
            throw new IllegalStateException("invariant failure");
    }
}
```

Problema con il security manager: i permessi possono essere full allow o

full deny, non si può specificare una policy per oggetto. Questo può essere limitante quando si vogliono concedere permessi solo a determinate chiamate o metodi.

Si può manipolare il security manager? La call stack introspection permette di controllare chi sta richiedendo accesso e concedere permessi solo quando necessario.

Esempio:

```
public class SelectiveAccessibilityCheck {
  public static void main(String[] args) throws Exception {
    System.setSecurityManager(new SecurityManager() {
      public void checkPermission(Permission p) {
        if (p instanceof ReflectPermission
            && "suppressAccessChecks".equals(p.getName()))
          for (StackTraceElement e: Thread.currentThread().getStackTrace())
            if ("SelectiveAccessibilityCheck".equals(e.getClassName())
                && "setName".equals(e.getMethodName()))
              throw new SecurityException();
      }
    });
    Employee eleonor = new Employee("Eleonor", "Runedottir");
    System.out.println(eleonor);
    setSurname(eleonor, "Odindottir");
    System.out.println(eleonor);
    setName(eleonor, "Angela");
    System.out.println(eleonor);
  }

  private static void setName(Employee e, String n)
        throws Exception {
    Field name = Employee.class.getDeclaredField("name") ;
    name.setAccessible(true) ; name.set(e, n);
  }
  private static void setSurname(Employee e, String s)
        throws Exception {
    Field surname = Employee.class.getDeclaredField("surname") ;
    surname.setAccessible(true) ; surname.set(e, s);
  }
}
```

In questo modo viene bloccato il `setName()`, permettendo il resto delle modifiche riflessive; fondamentalmente è una blacklist basata sullo stack trace: se il nome del metodo è "`setName`" alza un'eccezione.

Esecuzione:

```
> java SelectiveAccessibilityCheck
Employee: Eleonor Runedottir
Employee: Eleonor Odindottir
Exception in thread "main" java.lang.SecurityException
```

2.2.1 `StackWalker`

L'introspezione tramite `Throwable` presenta dei problemi:

- Performance overhead: anche se serve un solo frame, il `Throwable` costruisce tutto il call stack (costoso)
- Eager stack reification: tutto lo stack è reificato, anche se poi viene usato un solo elemento (top frame, o quello subito sotto)
- No native stream support: lo stack trace è un vettore, va memorizzato e usato come tale
- Meta-dati limitati: fornisce solo informazioni testuali, niente accesso diretto a informazioni su classi o moduli

Un approccio più moderno per fare introspezione sullo stack è lo `StackWalker`.

```
1  public final class StackWalker {
2      static enum Option { RETAIN_CLASS_REFERENCE,
3          SHOW_REFLECT_FRAMES, SHOW_HIDDEN_FRAMES }
4
5      public static interface StackFrame {
6          String getClassName();
7          String getMethodName();
8          Class<?> getDeclaringClass();
9          int getByteCodeIndex();
10         String getFileName();
11         int getLineNumber();
12         boolean isNativeMethod();
13         StackTraceElement toStackTraceElement();
14         ...
15     }
```

```
    public static StackWalker getInstance();
    public static StackWalker getInstance(Set<Option> options);
    ...
    public <T> T walk
        (Function<? super Stream<StackFrame>, ? extends T> function)
        { ... };
    public void forEach(Consumer<? super StackFrame> action)
        { ... };
    public Class<?> getCallerClass() { ... };
}
```

Fornisce un'interfaccia più moderna, stream-oriented.

Per riscrivere l'esempio dell'invariant checker:

```
import java.lang.StackWalker;
public class InvariantChecker {
    public static void checkInvariant(InvariantSupporter obj) {
        StackWalker walker = StackWalker.getInstance();
        boolean insideCheckInvariant =
          walker.walk(frames -> frames.skip(1)
            .anyMatch(frame ->
              "InvariantChecker".equals(frame.getClassName()) &&
              "checkInvariant".equals(frame.getMethodName()))
        );
        if (insideCheckInvariant)
            return;
        if (!obj.invariant()) {
            throw new IllegalStateException("invariant failure");
        }
    }
}
```

Il call stack è attraversato in maniera lazy usando gli stream. Il call stack completo è reificato solo se non vengono rilevati loop.

Il comportamento dello StackWalker può essere modificato tramite alcune opzioni

```
var walker = StackWalker.getInstance(Set.of(«option list»));
```

Opzioni:

- `RETAIN_CLASS_REFERENCE`: fornisce accesso diretto al `Class<?>` al posto

che delle stringhe; con attiva questa opzione si può accedere direttamente alla classe dichiarante

```
Class<?> cls = frame.getDeclaringClass();
```

senza questa opzione va usata la reflection

```
Class<?> cls = Class.forName(ste.getClassName());
```

- `SHOW_REFLECT_FRAMES`: include i frame relativi alla reflection nella stack trace
- `SHOW_HIDDEN_FRAMES`: include frame di implementazione o sintetici, come funzioni lambda e native

Attraverso le ultime due opzioni, lo `StackWalker` può avere accesso a un frame "più completo", nello stesso stack trace ci possono essere più frame "nascosti", mostrabili tramite opzioni.

2.3 Java Annotations

Le **annotazioni**, anche chiamate meta-dati, sono dati strutturati che descrivono, spiegano, localizzano o rendono più facile da ritrovare, usare o gestire una fonte di informazioni. Informazioni su informazioni, interpretate *a un certo punto* dal codice e da strumenti di analisi.

Informazioni aggiuntive sul codice, non destinate al programma, possono essere usate per: documentare il codice, estrarre dati specifici dal programma, generare automaticamente file di configurazione, …

Tipi **standard** di **annotazioni** forniti da Java out-of-the-box:

- `@Override`: per segnalare che un metodo sovrascrive un altro nella propria superclasse
- `@Deprecated` per indicare che l'uso di questo metodo è scoraggiato, potrebbe non avere più supporto in future versioni
- `@SuppressWarnings` per disattivare avvisi del compilatore riguardo classi, metodi o inizializzatori di campi e variabili
- `@FunctionalInterface` (da Java 8) per segnare un'interfaccia come funzionale, i.e., con esattamente un metodo astratto; serve solo a segnalare al compilatore il tipo di interfaccia, potrebbe anche essere omessa; sono la base delle espressioni lambda

2.3.1 Creare annotazioni custom

Ci sono tre categorie di **annotazioni custom**:

- **Marker annotation**, nessuna informazione aggiuntiva se non il nome stesso del metadato, viene applicato alla funzione stessa; dicono qualcosa sul dato annotato, senza parametri (o con valori di default per tutti i parametri)

```
@MarkerAnnotation
```

- **Single-Value Annotations**, l'annotazione contiene un singolo valore

```
@SingleValueAnnotation("some value");
```

- **Full Annotations**, sfruttano tutto il range della sintassi dell'annotazione

```
@Reviews({ // curly braces indicate an array of values
    @Review(grade=Review.Grade.EXCELLENT, reviewer="DF"),
    @Review(grade=Review.Grade.UNSATISFACTORY, reviewer="EG",
        comment="This method needs an @Override annotation.")
})
```

Comunque le annotazioni sono risolvibili solo staticamente, non si possono inserire all'interno oggetti o elementi istanziati dinamicamente.

I tipi delle annotazioni sono, praticamente, interfacce. Assomigliano alle dichiarazioni di interfacce, ma si usa `@interface` per indicare al compilatore che si tratta di un'annotazione.

Esempio

```
public @interface TODO {
    String value();
}

@TODO("Figure out the amount of interest per month")
public void calculateInterest(float amount, float rate) {
    // need to finish this method later
}
```

Similmente si possono definire tipi di annotazione con più membri

```
public @interface GroupTODO {
    public enum Severity {CRITICAL, IMPORTANT, TRIVIAL};
    Severity severity() default Severity.IMPORTANT;
```

```
4       String item();
5       String assignedTo();
6   }
7
8   @GroupTODO(
9   severity=GroupTODO.Severity.CRITICAL,
10  item="Figure out the amount of interest per month",
11  assignedTo="Foo"
12  )
13  public void calculateInterest(float amount, float rate) {
14      // need to finish this method later
15  }
```

Omettere un valore significa usare quello di default (per `severity` sarebbe `Severity.IMPORTANT`).

Da notare che i tipi usabili come membri per le annotazioni devono essere risolvibili a compile-time, i.e., `String`, `Class`, `enum`, `annotation` o array di quest'ultimi.

2.3.2 Meta-annotazioni

Le meta-annotazioni sono annotazioni sulle annotazioni. Sono presenti 5 tipi predefiniti:

- `@Target` specifica quali elementi del programma (tipi, metodi, …) possono avere annotazioni del tipo definito
- `@Retention` indica se un'annotazione deve essere mantenuta o meno nel file class compilato dal compilatore
- `@Documented` indica che l'annotazione dovrebbe essere considerata parte dell'API pubblica dell'elemento annotato
- `@Inherited` viene usato in tipi di annotazione usati su classi e serve a indicare che il tipo annotato è ereditato
- `@Repeatable` (da Java 8) permette l'uso dell'annotazione per annotare lo stesso elemento più volte

`ElementTypes` Si tratta di un `enum` usato assieme a `@Target` specificano *dove* un'annotazione può essere usata. Valori:

- `TYPE` per classi, interfacce, enum, record

- `FIELD` per campi (variabili d'istanza o statiche)
- `METHOD` per metodi
- `PARAMETER` per parametri di metodo o costruttore
- `CONSTRUCTOR` per costruttori
- `LOCAL_VARIABLE` per variabili locali (all'interno di un metodo)
- `ANNOTATION_TYPE` per altre annotazioni
- `PACKAGE` per package
- `TYPE_PARAMETER` per parametri di tipo (generics, dove si dichiara un parametro di tipo `<T>`)
- `TYPE_USE` per qualsiasi uso su tipi (più generico di `TYPE_PARAMETER`, per annotare ogni utilizzo di un tipo)
- `MODULE` per moduli (da Java 9)
- `RECORD_COMPONENT` per componendi dei record (da Java 16)

Possono essere combinati assieme, `@Target` permette di averne più di uno come parametro.

`RetentionPolicy` Usato assieme a `@Retention` per specificare *quanto a lungo* mantenere un'annotazione. Valori:

- `SOURCE`: l'annotazione è visibile solo nel codice sorgente e viene scartata dal compilatore, non appare nel `.class` e non è accessibile a runtime
- `CLASS`: l'annotazione viene inserita nel file `.class` ma non è accessibile a runtime, quindi è presente nel bytecode ma non visibile via reflection
- `RUNTIME`: l'annotazione viene mantenuta fino all'esecuzione, è presente nel `.class` e può essere letta via reflection a runtime

2.3.3 Reflection sulle annotazioni

Il modo più semplice per controllare le annotazioni è usando il metodo `isAnnotationPresent()`. Esempio:

```
1 import java.lang.annotation.RetentionPolicy;
2 import java.lang.annotation.Retention;
3
```

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface TODO {
    String value();
}

@TODO("Everything is still missing")
class Test {}

public class TestIsAnnotated {
    public static void main(String[] args) {
        Class<?> c = Test.class;
        boolean todo = c.isAnnotationPresent(TODO.class);
        if (todo)
            System.out.println("Not completed yet");
        else
            System.out.println("Completely implemented");
    }
}
```

Esecuzione:

```
>java TestIsAnnotated
Not completed yet
```

Si possono anche ottenere i valori all'interno dell'annotazione attraverso la reificazione della stessa

```java
@TODO("Still nothing")
class Test {}

public class TestAnnotationMember {
    public static void main(String[] args) {
        Class<?> c = Test.class;
        TODO todo = c.getAnnotation(TODO.class);
        String val = todo.value().
        System.out.println("TODO value is: '" + val + "'.")
    }
}
```

Esecuzione:

```
>java TestAnnotationMember
TODO value is 'Still nothing'.
```

2.3.4 Annotation Processor

Un tool di Java che permette di ispezionare le annotazioni del codice sorgente durante la compilazione. Come funziona?

1. Una classe che estende `javax.annotation.processing.AbstractProcessor` funge da processore
2. Deve sovrascrivere il metodo

```
boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv)
```

   per esaminare gli elementi annotati e definire come le annotazioni vengono gestite

3. Il processore deve essere registrato in `META-INF/services/javax.annotation.processing.Processor`
4. Quando il codice viene compilato usando

```
> javac -cp . -processor "processor" "code to process"
```

   l'annotation processor specificato viene chiamato automaticamente sulle sorgenti specificate

Esempio di annotation processor

```
package aps;

import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes("*")
@SupportedSourceVersion(SourceVersion.RELEASE_21)
public class LogProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement>
```

```
            annotations, RoundEnvironment roundEnv) {
        for (TypeElement annotation: annotations) {
            for (Element element:
                    roundEnv.getElementsAnnotatedWith(annotation)) {
                processingEnv.getMessager().printMessage(
                    Diagnostic.Kind.NOTE,
                    "found @" + annotation.getSimpleName()
                    + " at " + element);
            }
        }
        return false; // don't claim annotations exclusively;
    }
}
```

Questo processor semplicemente stampa un messaggio con le annotazioni trovate. Restituisce `false` perché `true` imporrebbe l'esistenza di un singolo processor (in questo caso sarebbe anche vero, ma non è detto).

Sul codice

```
package demo;

import annotations.Log;
import annotations.TODO;

@Log
public class Hello {
    @TODO
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Eseguire il processor porta a

```
> javac -cp . -processor aps.LogProcessor demo/Hello.java
Note: found @Log at demo.Hello
Note: found @TODO at main(java.lang.String[])
```

2.4 Dynamic Proxy

Il design pattern del **proxy** viene definito come un oggetto come surrogato/al posto di un altro, usato per controllare e manipolare l'accesso all'altro oggetto.

La classe di un dynamic proxy deve implementare una serie di interfacce tali che l'invocazione di un metodo parte di una di queste interfacce su un'istanza porta a delegarne l'esecuzione a un altro oggetto che implementa tale interfaccia, questo oggetto è legato all'istanza del dynamic proxy.

Un dynamic proxy può essere usato per creare un proxy type-safe per un set di oggetti determinati dall'interfaccia implementata senza codice esplicito o pre-generazione statica della classe proxy (come succede con diversi tool che operano a compile-time).

Ogni proxy deve implementare un **invocation handler**. Quando un metodo viene invocato su un'istanza di un proxy, l'invocazione viene delegata al metodo `invoke()` del proprio invocation handler

```
Object invoke(Object proxy, Method m, Object[] args)
```

Il proxy è il primo meccanismo che implementa la riflessione per come definita inizialmente.

2.4.1 `java.lang.reflect.Proxy`

La classe `Proxy` fornisce metodi statici per creare classi e istanze di dynamic proxy, oltre a essere la superclasse di tutte classi create in tale modo.

```
public final class Proxy extends Object
            implements Serializable {
    protected InvocationHandler h;
    protected Proxy(InvocationHandler h) { ... }
    public static InvocationHandler getInvocationHandler
            (Object proxy) { ... }
    public static Class<?> getProxyClass
            (ClassLoader l, Class<?>... interfaces) { ... }
    public static boolean isProxyClass(Class<?> cl) { ... }
    public static Object newProxyInstance
            (ClassLoader loader, Class<?>[] interfaces,
                InvocationHandler h ) { ... }
}
```

In pratica, `Proxy` permette l'approccio a meta-oggetti all'interno di Java.

In breve: si ha un proxy, che implementa un'interfaccia, i metodi dell'interfaccia non vengono eseguiti dal proxy stesso, ma vengono delegato a un altro oggetto, legato al proxy, che implementa concretamente l'interfaccia. Il proxy è il meta-oggetto che gestisce l'oggetto sottostante, può arricchire le chiamate all'oggetto senza dover re-implementare il codice, come fosse un decoratore "generico".

Esempio per tracciare le chiamate a metodo:

```
1  import java.lang.reflect.InvocationHandler;
2  import java.lang.reflect.Method;
3
4  public class TraceHandler implements InvocationHandler {
5      private Object baseObject;
6      public TraceHandler(Object base) {
7          baseObject = base;
8      }
9      public Object invoke(Object proxy, Method m, Object[] args) {
10         try {
11             System.out.println("before " + m.getName());
12             Object result = m.invoke(baseObject, args);
13             System.out.println("after " + m.getName());
14             return result;
15         } catch (Exception e) {
16             e.printStackTrace(); return null;
17         }
18     }
19     public String toString() {
20         return "th :- " + this.baseObject;
21     }
22 }
```

Esecuzione:

```
jshell> IPoint p1 = new Point(10,20);
p1 ==> p <- (10, 20)

jshell> IPoint th_p1 = (IPoint) Proxy.newProxyInstance
  ...> (p1.getClass().getClassLoader(),
  ...> p1.getClass().getInterfaces(), new TraceHandler(p1));
```

```
before toString
after toString
th_p1 ==> p <- (10, 20)

jshell> p1.getX(); /* standard call */
$10 ==> 10

jshell> th_p1.getX(); /* traced call */
before getX
after getX
$11 ==> 10
```

Viene generato un proxy che rappresenta la classe, legato a una istanza concreta dell'oggetto su cui si vuole fare reflection, i quali metodi vengono chiamati tramite l'invoke dell'handler.

L'uso del proxy permette di decorare una classe senza ridefinire nessun metodo, se non il meta-comportamento voluto.

2.5 Class Loading

Il **Class Loader**, definito tramite la classe astratta `ClassLoader` e le sue sottoclassi (`SecureClassLoader`, `URLClassLoader`, …), è l'oggetto che si occupa di prendere le classi e portarle in memoria. Dato il nome di una classe

- Identifica la classe secondo le regole del class loader oppure
- Genera i dati (bytes array) che costituiscono una definizione per la classe

Per poi renderla disponibile all'applicazione in esecuzione.

Inoltre i class loader definiscono i name space: diversi class loader definiscono name space separati. Ogni oggetto `Class` contiene un riferimento al `ClassLoaer` che lo ha definito (`getClassLoader()`).

Class `A` loaded by `CL1` = Class `A` loaded by `CL2` $\Leftrightarrow$ `CL1` = `CL2`

Esempio: rompere singleton pattern con i name space: data una definizione di `Singleton` come

```
1 public class Singleton {
2     static private boolean runOnce = false;
```

```
    public Singleton() {
        if (runOnce)
            throw new IllegalStateException
                ("[ERROR] re-instantiation of «Singleton»!!!");
        runOnce = true;
    }
}
```

La variabile `runOnce` viene istanziata assieme al namespace, quindi usando class loader diversi si possono avere più istanze di un Singleton

```
public class SingletonViolationTest {
    public static void main(String[] args) throws Exception {
        SimpleClassLoader CL1 = new SimpleClassLoader("testclasses");
        Class<?> c1 = CL1.loadClass("Singleton");
        println("Loaded «Singleton» via «CL1» class loader");
        Field flag = c1.getDeclaredField("runOnce");
        flag.setAccessible(true);
        println("Let's instantiate «Singleton@CL1»\n
            ### runOnce :- " + flag.get(null));
        Object x = c1.getDeclaredConstructor().newInstance();
        println("### runOnce :- " + flag.get(null));
        try {
            println("Let's re-instantiate «Singleton@CL1»\n
                ### runOnce :- " + flag.get(null));
            Object y = c1.getDeclaredConstructor().newInstance();
            throw new RuntimeException("Test Fails!!!");
        } catch (Exception e) {
            println(e.getCause().getMessage());
        }
        SimpleClassLoader CL2 = new SimpleClassLoader("testclasses");
        println("Loaded «Singleton» via «CL2» class loader");
        Class<?> c2 = CL2.loadClass("Singleton");
        Field flag2 = c2.getDeclaredField("runOnce");
        flag2.setAccessible(true);
        println("Let's instantiate «Singleton@CL2»\n
            ### runOnce :- " + flag2.get(null));
        Object z = c2.getDeclaredConstructor().newInstance();
        println("### runOnce :- " + flag.get(null));
    }
}
```

Esecuzione:

```
>java SingletonViolationTest
Loaded «Singleton» via «CL1» class loader
Let's instantiate «Singleton@CL1»
### runOnce :- false
### runOnce :- true
Let's re-instantiate «Singleton@CL1»
### runOnce :- true
[ERROR] re-instantiation of «Singleton»!!!
Loaded «Singleton» via «CL2» class loader
Let's instantiate «Singleton@CL2»
### runOnce :- false
### runOnce :- true
```

Modificare il Class Loader può avere più funzionalità. Le applicazioni implementano sottoclassi di `ClassLoader` per (ad esempio)

- Estendere il modo in cui la JVM carica le classi dinamicamente
- Ottenere diversi name space protetti
- Trasformare bytecode

Alcune applicazioni legato all'uso dei class loader sono:

- Sicurezza: esaminare le classi per una firma digitale corretta, impedire il caricamento di specifici packages
- Crittografia: decrittare classi on-the-fly, in modo che i file `.class` su disco non siano leggibili da qualcuno con un decompiler
- Archiviazione: distribuire il codice in un formato particolare o con una compressione speciale
- Testing: mantenere il contesto dei test case separato usando loader differenti (stile sandbox)

Un altro problema: scrivere un metodo che stampi la gerarchia di eredità di un'applicazione. Come scoprire quali classi appartengono all'applicazione?

Il problema è estrinseco dal MOP di Java, creare un class loader personalizzato permette di risolvere il problema in quando a conoscenza delle classi caricate.

2.5.1 `java.lang.ClassLoader`

Struttura di un Class loader:

```
public class ClassLoader {
    protected ClassLoader() { ... }
    protected ClassLoader(ClassLoader parent) { ... }
    public final ClassLoader getParent() { ... }
    public static ClassLoader getSystemClassLoader()
        throws SecurityException, IllegalStateException { ... }
    public static ClassLoader getPlatformClassLoader()
        throws SecurityException { ... }
    protected Class<?> findClass(String name)
        throws ClassNotFoundException { ... }
    protected Class<?> findClass(String moduleName, String name)
        throws ClassNotFoundException { ... }
    protected Class<?> findLoadedClass(String name) { ... }
    protected Class<?> findSystemClass(String name)
        throws ClassNotFoundException { ... }
    public Class<?> loadClass(String name)
        throws ClassNotFoundException { ... }
    protected final void resolveClass(Class<?> c)
        throws NullPointerException { ... }
    protected Class<?> defineClass (String name, byte[] b, int off, int len)
        throws ClassFormatError { ... }
    ...
}
```

Il processo parte da `loadClass()`:

- Chiama `findLoadedClass()` per controllare se la classe è già stata caricata o meno
- Se già caricata, restituisce la classe, altrimenti delega il compito al `ClassLoader` genitore
- Se tutta la gerarchia fallisce, invoca `findClass()` per trovare la classe, leggere il bytecode e creare la classe (`defineClass()`)

Pseudo-Java del processo:

```
public Class loadClass(String name)
        throws ClassNotFoundException {
    Class c = findLoadedClass(name);
```

```
    if (c != null)
        return c;
    try {
        return parent.loadClass(name);
    } catch (ClassNotFoundException e) {
        return findClass(name);
    }
}
```

Per creare una sottoclasse di `ClassLoader` bisogna

- Creare un costruttore di default e uno legato al genitore
- sovrascrivere `findClass()`
- NON sovrascrivere `loadClass()` perché l'implementazione originale supporta il modella a delega, come descritto

Per modificare il class Loader senza rompere tutto, è meglio non modificare `defineClass()` e `loadClass()`, meglio attenersi al `findClass()`.

Esempio di class loader per CLASSPATH specificato dinamicamente

```
import java.io.*;
public class SimpleClassLoader extends ClassLoader {
    String[] directories;
    public SimpleClassLoader(String path) {
        directories = path.split( ";" );
    }
    public SimpleClassLoader(String path, ClassLoader parent) {
        super(parent);
        directories = path.split(";");
    }

    public synchronized Class<?> findClass(String name)
            throws ClassNotFoundException {
        for (int i = 0; i < directories.length; i++) {
            byte[] buf = getClassData(directories[i], name);
            if (buf != null)
                return defineClass(name,buf,0,buf.length);
        }
        throw new ClassNotFoundException();
    }
```

```
21
22      protected byte[] getClassData(String directory, String fileName) {
23          String classFile = directory + "/"
24              + fileName.replace('.','/') + ".class";
25          int classSize = (int)(new File(classFile)).length();
26          byte[] buf = new byte[classSize];
27          try {
28              FileInputStream filein = new FileInputStream(classFile);
29              classSize = filein.read(buf);
30              filein.close();
31          } catch(FileNotFoundException e) {
32              return null;
33          }
34          catch(IOException e) {
35              return null;
36          }
37          return buf;
38      }
39  }
```

Al posto di generare codice Java lavora con il bytecode

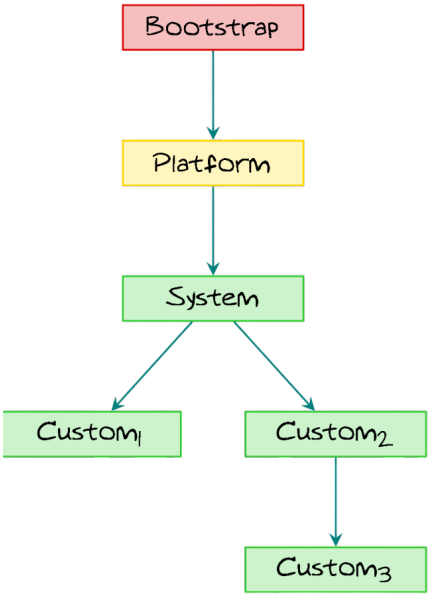
```
1  defineClass(String name, byte[] b, int off, int len)
```

Il sub-array da `byte[off]` a `byte[off+len-1]` contiene il bytecode per la classe. Bisogna avere in qualche modo il bytecode, può essere generato da zero oppure a partire dal bytecode di una classe esistente.

Tre fatti fondamentali sui class loader:

1. A runtime, un oggetto class è identificato dal nome della classe assieme all'oggetto di tipo class loader che ha creato il class; l'oggetto loader che ha eseguito `defineClass()` per l'oggetto class considerato
2. Tutte le altre classi referenziate da un class sono caricate dallo stesso loader
3. I class loader delegano ad altri class loader, specialmente quello di sistema, il quale si occupa di interpretare il class path.

Gerarchia Si ha una gerarchia di class loader



Dall'alto verso il basso:

- **Bootstrap/Primordial/Null Class Loader:** il class loader built-in, scritto in C, non estende `ClassLoader`; carica le classi Java che fanno parte del core di `java.base`
- **Platform/Extensions Class Loader:** estende `Bootstrap`, carica i moduli jdk standard al di fuori di `java.base`
- **System Class Loader:** Estende `Platform`, carica classi e jar dal `CLASSPATH`; può essere customizzato tramite `-Djava.system.class.loader`

2.6 Moduli e Reflection

Con Java 9 viene introdotta la **Java Module Platform**, la quale ha gli obiettivi di

- Fornire una **configurazione più affidabile**, mirata a rimpiazzare il meccanismo basato sui classpath a favore di un modo per dichiarare esplicitamente le dipendenze
- Una **più forte encapsulation**, per permettere a ciascun componente di dichiarare quale dei suoi tipi pubblici è accessibile ad altri componenti

Un **modulo** raggruppo classi, interfacce e packages e all'interno di un file `module-info.java` vengono descritte le dipendenze tra moduli

```
module com.foo.bar {
    requires org.baz.foo;
    exports com.foo.bar.abc;
}
```

Questo file viene incluso nel `jar`.

I moduli, per come definiti in questo modo, *bloccano la reflection*, in quanto questa non è in grado di "superare" le restrizioni date dai moduli; è necessario che il modulo "destinazione" apra l'accesso al componente su cui si vuole fare reflection.

Alcune soluzioni:

- Aggiungere `--add-opens` alla JVM per aprire il modulo; ad esempio, per aprire il pacchetto di `java.lang`, all'interno del modulo `java.base` verso tutti i moduli senza nome (non dichiarati esplicitamente)

  ```
  > java --add-opens java.base/java.lang=ALL-UNNAMED ...
  ```

  Ma non è sempre possibile aggiungere flag alla JVM, come ad esempio usando framework e tool specifici

- Aprire il pacchetto/modulo attraverso il relativo file `module-info.java`, ma anche questo non è detto sia sempre applicabile (e.g., progetti complessi in cui il file non è disponibile)

```
module com.foo.bar {
    opens com.foo.bar.abc to module.with.reflection;
    requires org.baz.foo;
    exports com.foo.bar.abc;
}
```

```
module module.with.reflection {
    exports module.with.reflection;
}
```

Import e require sono sostanzialmente delle dipendenze, le quali vanno risolte ricorsivamente (ogni modulo ha le proprie dipendenze, ma vanno risolute anche quelle dei moduli chiamati da ogni modulo), possono essere anche cicliche.

Può essere pensato come un grafo di dipendente, la quale chiusura transitiva rappresenta il grafo dei moduli, all'interno della quale ogni arco tra nodi (moduli) rappresenta una dipendenza soddisfatta.

Esiste un tool per vedere questa chiusura: `jdeps`; un semplice esempio:

```
> jdeps --module-path out --module com.example.app
com.example.app -> java.base
com.example.app -> com.example.utils
```

dove `--module-path` specifica la posizione dei moduli compilati, mentre `--module` specifica i moduli da analizzare.

Anche `jshell` deve considerare i moduli: va dato il module path

```
> jshell
    --module-path=<path>
```

Assieme agli import

```
    --add-modules=<module>
```

E alle open da fare

```
    --add-exports=<module>/<package>:jshell
```

Per aprire il package verso la `jshell`. Al posto di quest'ultimo si può mettere un altro modulo quando usato con `javac`.

2.7 Method and Variable Handles

La reflection standard è lenta e non sicura, bypassa molti controlli compile-time e dipende da lookup di stringhe.

Vogliamo sostituire questi lookup con `MethodHandle` e `VarHandle`, i quali sono più veloci e fortemente tipizzati. Permettono accesso diretto a campi e metodi, preservando i check a compile-time.

Si ha la classe factory di `MethodHandles`, la quale permette di creare `MethodHandle`.

Il `MethodHandles.Lookup` rappresenta i permessi, ovvero determina cosa è accessibile e controlla anche l'accesso ai campi privati (non si usa più il `setAccessible()`).

L'oggetto `lookup` può accedere a tutti i membri della classe che lo ha definito, ma non ai membri privati di altre classi. `MethodHandles.publicLookup` o

`MethodHandles.privateLookupIn` possono restringere o estendere questi permessi.

Gli handle possono effettuare direttamente loro la reificazione

```
VarHandle vh = MethodHandles.lookup()
    .findVarHandle(Employee.class, "surname", String.class);
```

oppure trasferirla da un meta-oggetto esistente

```
Field f = Employee.class.getDeclaredField("surname");
f.setAccessible(true);
VarHandle vh = MethodHandles.lookup().unreflectVarHandle(f);
```

3 OpenJava

3.1 Non standard MOPs

Il MOP di Java si limita a fare introspezione, MOP alternativi possono permettere anche la manipolazione. Ad esempio, con OpenJava abbiamo class-to-class transformation.

Esistono diversi tipi di MOP per Java, i quali funzionano:

- **Compile Time:** reflection sui costrutti del linguaggio, si basano su trasformazioni tra sorgenti, i cambiamenti avvengono su tutta la classe. Tra questi rientrano: OpenJava e Reflective Java
- **Load Time:** la reflection viene fatta sul bytecode, basata sulla specializzazione del class loader, non necessario il sorgente, i cambiamenti avvengono per classe. Tra questi: BCEL, ASM, Javassist
- **Run Time (VM-Based):** modifiche alla VM per intercettare eventi, portano a una perdita di portabilità ma permettono modifiche anche al singolo oggetto. Tra questi: MetaXa, Iguana/J
- **Run Time (Proxy-Based):** generazione trasparente di componenti per intercettare, può basarsi sul sorgente o sul modifica/generazione del bytecode (può essere implementato tramite MOP a compile o load time), i cambiamenti possono essere per oggetto. Tra questi: Dalang/Kava, mChaRM

3.2 Funzionamento di OpenJava

OpenJava è un linguaggio riflessivo derivato da Java, permette al programmatore di estendere un programma sintatticamente e semanticamente. L'attività riflessiva avviene a **compile-time**, può essere usato per scrivere sia meta che base level.

Senza definire un meta-livello, un programma OpenJava diventa uguale al corrispondente Java.

Le estensioni vengono passate direttamente al compilatore come meta-oggetti, i quali dicono al compilatore come tradurre i componenti del programma. Prima di costruire il bytecode, il compilatore lo traduce in Java puro tramite le direttive dei meta-oggetti.

Prima di compilare il sorgente, il compilatore invoca il metodo `translate()` del meta-oggetto alla base dell'AST del programma; tale metodo cammina

ricorsivamente l'AST traducendo il codice. Estensioni del linguaggio vengono implementate ridefinendo il metodo `translate()` all'interno della classe corrispondente.

OpenJava lavora per classe, associando ad ogni classe un'altra (il meta-oggetto) che gestisce il processo di traduzione. Meta-oggetti e oggetti interagiscono attraverso un MOP definito.

La programmazione riflessiva in OpenJava si compone di tre passi principali:

1. Stabilire come il programma base-level deve diventare in seguito alla traduzione
2. Determinare quali parti del codice base-level saranno inclusi nella traduzione e determinare che codice ausiliario sarà necessario
3. Scrivere un meta-oggetto che si occupa della traduzione designata

Anche se la traduzione è fatta a compile time, la reflective API è molto semplice e assomiglia a quella fornita da `java.lang.reflect`.

La classe base viene connessa al meta-livello tramite la keyword `instantiates`, il resto, nel base-level, è codice Java standard

```
1 public class Hello instantiates VerboseClass {
2     public static void main( String[] args ) {
3         hello();
4     }
5     static void hello() {
6         System.out.println( "Hello, world." );
7     }
8 }
```

La frase `instantiates VerboseClass` dice al compilare che la classe `Hello` ha una istanza di un meta-oggetto di classe `VerboseClass`. La classe `VerboseClass` descrive come la classe `Hello` deve essere tradotta.

Se la classe `VerboseClass` volesse rendere verbosa l'esecuzione delle istanze di classe `Hello`:

```
1 import openjava.mop.*;
2 import openjava.ptree.*;
3 public class VerboseClass instantiates Metaclass extends OJClass {
4     public void translateDefinition() throws MOPException {
```

```
5           OJMethod[] methods = getDeclaredMethods();
6           for (int i = 0; i < methods.length; ++i) {
7               Statement printer = makeStatement(
8               "System.out.println( " + methods[i] + "\" is called.\" );"
9               );
10              methods[i].getBody().insertElementAt( printer, 0 );
11          }
12      }
13  }
```

Esecuzione:

```
> java Hello
main is called.
hello is called.
Hello, world.
```

Per tale traduzione, `VerboseClass`:

- Estende la classe `openjava.mop.OJClass`
- Sovrascrive il metodo `translateDefinition()` che traduce il corpo dei metodi
- I metodi dichiarati nelle classi del base-level sono recuperati tramite una chiamata a `getDeclaredMethods()`
- La chiamata a `makeStatement()` permette di costruire uno statement a partire da una stringa on-the-fly
- Il metodo `getBody()` torna una lista di statement rappresentanti il corpo del metodo; statement possono essere aggiunti o rimossi da tale rappresentazione

4 Bytecode Engineering

Con bytecode engineering intendiamo, preso il contenuto di una classe (di un file `.class`), riuscire a modificare il codice, prima dell'istanziazione degli oggetti. In questo modo le modifiche vengono applicate a load time, senza rischiare istanze non-modificate "in giro" (non vuol dire che a runtime sia impossibile fare modifiche, ma è più difficile).

Alcuni tool per la bytecode instrumentation:

- `ASM`
- `BCEL - Bytecode Engineering Library`
- `Javassist - Java programming Assistant`
- `Soot/SootUp`

Difficoltà della bytecode instrumentation:

- Creare una rappresentazione intermedia: tutti gli oggetti devono essere rappresentati e tale rappresentazione può diventare difficile da gestire con la dimensione
- Applicare multiple trasformazioni in serie: disfare i cambiamenti può diventare complicato senza tracciare tutti gli step intermedi
- Mantenere il class file verificabile: in termini di consistenza e dimensione
- Bloat all'interno del class pool: eliminare "dead entry" potrebbe richiedere di analizzare l'intera classe
- Inserire instrumentazione corretta: verificare la correttezza e consistenza
- Thread safety: bisogna considerare possibili race conditions o deadlock

Per comprendere l'instrumentazione a livello di bytecode dobbiamo conoscere:

- La JVM
- Il formato dei class file
- Il bytecode instruction set

4.1 La JVM

I programmi scritti in Java vengono compilati in un formato portabile binario, chiamato **bytecode**.

Ogni classe, record, enum e interfaccia viene rappresentato da un singolo class file, il quale contiene tutti i dati della classe e le istruzioni in bytecode. Questi file sono **caricati dinamicamente** ed eseguiti dalla JVM.

La JVM simula una CPU stack-based. Ogni frame ha uno stack di operandi e un array di variabili locali:

- Lo stack di operandi è usato per le computazioni e ricevere il valore di ritorno di un metodo chiamato
- Le variabili locali servono da registri e sono usate per passare gli argomenti ai metodi

Sono possibili 256 opcode, dei quali 202 sono in uso, 51 sono riservati per usi futuri e 3 sono riservati permanentemente per uso interno: due forniscono trap handling, il terzo supporta i breakpoint.

Bytecode instruction set:

- Control transfer: `goto`, `jsr`, `ret`, `if*`, `cmp*`
- Method invocation and return:
  - Chiamate a metodi: `invokestatic`, `invokevirtual`, `invokeinterface`, `invokespecial`, `invokedynamic`
  - Return dai metodi: `*return`, e.g., `ireturn`, `dreturn`, …
- Stack management e load/store operations: `pop`, `push`, `dup`, `*load`, `*store`, …
- Object creation and manipulation:
  - Creazione di oggetti: `new`, `newarray`, `anewarray`, `anewmultiarray`
  - Manipolazione di oggetti: `getfield`, `putfield`, `getstatic` and `putstatic`
- Arithmetic and logic operations: `*add`, `*div`, …
- Type conversions and checks: `f2i`, `checkcast`, `typecast`, `instanceof`

Con `*` si intende indicare varie versioni del comando, solitamente con una lettera che indica il tipo su cui agisce il comando.

4.2 Formato dei class file

Il codice compilato è rappresentato tramite un formato binario indipendente dall'hardware, generalmente all'interno di un file chiamato **class file**.

Il class file definisce la rappresentazione di una classe o interfaccia, consiste di uno stream di byte e di una singola struttura `ClassFile`.

La definizione in del ClassFile è in C, in quanto la JVM è scritta in C

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Dove:

- `magic` sono i magic number del formato (`0xCAFEBABE`)
- `minor_version`, `major_version`: le versioni di Java in cui funziona il file (con un offset di 44)
- `constant_pool_count`: il numero (+1) di entry all'interno della constant pool
- `constant_pool[]`: contiene tutto ciò che è statico, come nomi di classi, interfacce, metodi, …

- `access_flags`: i permessi di accesso alla classe/interfaccia, funzionano come i permessi linux, esce un numero
- `interfaces_count`: il numero di interfacce implementate dalla classe/interfaccia
- `interfaces[]`: ogni entry deve essere un indice valido della constant pool table
- `fields_count`: il numero di `field_info` all'intero della fields table
- `fields[]`: ogni entry deve essere una `field_info` che descrive il campo di una classe
- `methods_count`: denota il numero di `method_info` nella methods table
- `methods[]`: ogni entry deve essere una `method_info` che descrive un metodo della classe
- `attributes_count`: denota il numero di `attribute_info` all'interno della attributes table
- `attributes[]`: ogni entry deve essere una `attribute_info` che descrive gli attributi della classe
- `this_class`, `super_class`: riferimenti alla constant pool

5 Javassist

Javassist è una libreria per modificare bytecode e permette la modifica di classi a load-time e la creazione di nuove classi a runtime.s

Si tratta di una API ad alto livello, non richiede di essere a conoscenza delle specifiche del bytecode; 100% Java, compliant Java 21, basato sul modello di class loading.

Con Javassist, la fase di loading è "dirottata" a un translator, che restituisce la classe modificata.

Migliora la reflection di Java con una reificazione parallela di classi e tipi primitivi di Java; `CtClass`, `CtMethod`, `CtConstructor`, `CtField` (`Ct` sta per "concrete"), `FieldInitializer`, `CtPrimitiveType`, `Modifier`, …

Introduce nuove classi per scomporre il processo di class loading e permettere di modificare le classi a load time; `Loader`, `ClassPool`, `ClassPath`, `Translator`.

`ClassPool`: Un container di oggetti `CtClass`, ogni classe caricata passa attraverso questo oggetto:

- `get(String)` torna un riferimento a `CtClass`
- `toClass()` traduce una `CtClass` in un formato class standard
- `insertClassPath`, `appendClassPath` permettono di gestire il processo di lookup delle classi
- `make*` permette di creare classi e interfacce da zero
- servizi statici permettono di ottenere il class pool con o senza `Translator`

`Translator`: Un osservatore di `Loader`. Questa interfaccia dovrebbe essere implementata per tradurre un class file quando viene caricato all'interno della JVM

- `start(ClassPool)` inizializza il translator
- `onLoad(ClassPool, String)` operazioni svolte prima che la classe venga data al loader

Per modificare le classi, ci sono alcune limitazioni (anche se non è sempre ben chiaro il perché). Nello specifico, si può:

- Inserire una chiamata al metodo `m` prima o dopo un'altra chiamata a metodo
- Reindirizzare una chiamata a un altro metodo
- Reindirizzare un accesso a un campo a un altro campo
- Rimpiazzare un accesso a un campo con una chiamata a un metodo statico
- Rimpiazzare uno statement `new` con una chiamata a un metodo statico