

Example08

Table of contents

- [Reflections](#)
 1. [Class Objects](#)
 1. [getClass\(\)](#)
 2. [forName\(n\)](#)
 3. [loadClass\(n\)](#)
 4. [cast\(o\)](#)
 5. [isInstance\(o\)](#)
 6. [isAssignableFrom\(k\)](#)
 2. [Introspection](#)
- [Example08](#)
 1. [Example08.java](#)

Reflections

La JVM provvede modi, tramite il package `java.lang.reflect` di postporre decisioni a runtime. Seppur linguaggio statico, JAVA provvede modi orientati agli oggetti per:

- linking dinamico delle classi;
- **Introspection**;
- creazione dinamica di oggetti;
- accesso dinamico ai campi;
- invocazione dinamica dei metodi.

Ogni classe o interfaccia viene rappresentata, a tempo d'esecuzione, da un oggetto, detto *class object*. Descrive le caratteristiche della classe a tempo d'esecuzione. La classe `String` descritta da un oggetto a tempo d'esecuzione, per esempio; la classe `DownloadManager` descritta anch'essa da un oggetto.

Grazie a queste, siamo in grado di accedere a tutti i servizi del package `java.lang.reflect`, abbiamo sempre un riferimento a tempo d'esecuzione.

- se la classe/interfaccia, a tempo di compilazione ha nome `C`, allora la JVM offre a tempo d'esecuzione un oggetto che la implementa, una istanza della classe `java.lang.reflect.Class<C>` dove `C` e' lo stesso iniziale.

se prendiamo una stringa ("Ciao"), a tempo d'esecuzione abbiamo un oggetto che contiene il riferimento alla classe `String`

- se a tempo di compilazione abbiamo una classe, `C.class` permette di avere un riferimento all'oggetto descritto della classe `C`

quindi scrivere `String.class` nel nostro programma, vuole dire riferirsi a un oggetto unico che descrive `String (Class<String>)`

Ogni oggetto descrittore, viene associato ad un oggetto che e' stato utilizzato dalla JVM per caricare in memoria il byte code della classe, prende il nome di *class loader*.

Class Objects

`getClass()`

Se abbiamo un oggetto `o` non nullo, e chiamiamo `getClass()` su un reference a questo, otteniamo il descrittore della *factory class* di `o`.

Example

Se abbiamo in mano un `Runnable`, interfaccia, abbiamo un oggetto che l'ha comunque creato. Quindi se facciamo `referimento.getClass()` otteniamo il descrittore della classe che l'ha creato.

`forName(n)`

Se abbiamo una stringa a tempo d'esecuzione che e' equivalente al nome di una classe, `java.lang.String` per esempio, facendo `forName()` otteniamo il descrittore. Nessuno ci vieta che sia una interfaccia.

Viene lanciata una eccezione `ClassNotFoundException` se specificata una stringa che non e' il nome di una classe non raggiungibile, come nome scritto sbagliato oppure la classe c'e' ma non e' accessibile dal class path.

`loadClass(n)`

Quando facciamo `forName(n)`, stiamo chiedendo al class loader di caricare la classe: se e' gia' presente in memoria ci viene subito ritornato il descrittore che la rappresenta, altrimenti viene caricata tramite class loader.

`loadClass(n)` carica la classe.

`cast(o)`

A tempo d'esecuzione possiamo fare alcune cose che siamo abituati a fare nel sorgente (come cast). Se passiamo l'oggetto `o` al metodo `cast(o)`, questo prendera' cast specificato (il cast e' del riferimento).

`isInstance(o)`

Possiamo chiedere al descrittore di classe, se un oggetto e' istanza di `C`, se e' convertibile tramite cast a riferimento di tipo `C`.

Posso vedere l'oggetto `o` come se fosse di classe `C`?

Se si' allora possiamo fare cast, altrimenti fallira'.

`isAssignableFrom(k)`

Se possiamo supportare un assegnamento, dove `k` e' un descrittore di classe: `k` descrive una classe assegnabile ad oggetti di classe `C`? Se si' otteniamo `true` altrimenti `false`.

Introspection

Una volta che abbiamo la classe `Class` possiamo chiedere a questa, l'elenco dei metodi degli oggetti istanza di questa classe/interfaccia, vengono passati i nomi, ci dice qual'e' il tipo del nome di ritorno, chiediamo quanto argomenti ci sono e quali sono i tipi degli stessi; tutti oggetti.

Example08

Ci viene chiesto il nome qualificato della nostra classe (che puo' essere qualsiasi cosa, come abbiamo visto sopra):



nel momento in cui premiamo 'Invio', viene caricata la classe (sicuramente accessibile siccome caricata per prima dal class bootloader) e stampate tutte le informazioni ottenute dal descrittore di classe

```
Problems @ Javadoc Declaration Console X
<terminated> Example08 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Nov 28, 2022, 11:41:46 AM – 11:43:29 AM) [pid: 1382]
Please, enter the fully qualified name of a class: java.lang.Object
Class: java.lang.Object
Constructor: java.lang.Object()
Method: void wait(long arg0, int arg1)
Method: void wait(long arg0)
Method: void wait()
Method: boolean equals(java.lang.Object arg0)
Method: java.lang.String toString()
Method: int hashCode()
Method: java.lang.Class getClass()
Method: void notify()
Method: void notifyAll()
```

a tempo d'esecuzione sappiamo che:

la classe si chiama `java.lang.Object`,

con costruttore senza argomenti,

3 metodi `wait()`,

un metodo `equals()`,

uno `toString()`,

uno `hashCode()`,

`getClass()`,

`notify()` e `notifyAll()`

Questi nomi non vengono messi nel bytecode.

Esempio: il sorgente della classe `Object`, nella `wait(long)` il nome dell'argomento c'è, ma viene perso a tempo d'esecuzione.

Example08.java

Chiamiamo `go()` e andiamo a costruire lo scanner su `system.in` per leggere da tastiera. Una volta che lo abbiamo possiamo fare `nextLine()` per ritornare il testo della stringa fino a capo. Siccome una volta aperto va chiuso, un try-catch con risorse apre e chiude.

```
// ...
private void go() {
    try (Scanner scanner = new Scanner(System.in)) {
        scanner.useLocale(Locale.US);

        System.out.print
            ("Please, enter the fully qualified name of a class: ");

        String className = scanner.nextLine();

        show(className);
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    }
}
// ...
```

`show()` fa `forName()` e carica la classe: se non va a buon fine il caricamento, viene lanciata un'eccezione e che non può essere caricata la classe, se tutto va a buon fine viene chiamato `dump()` per stampare tutto l'output.

```
// ...
private void show(String className) {
    try {
        // siccome non sappiamo il tipo, mettiamo <?>
        Class<?> clazz = Class.forName(className);
```

```

        dump(clazz);
    } catch (ClassNotFoundException classNotFoundException) {
        System.err.println("Cannot load " + className);
    }
}
// ...

```

`dump()` prende l'argomento visto nella `show()` di tipo sconosciuto `<?>`.

- `getName()` ritorna il nome completo della classe;
- `getSuperClass()` ritorna la classe base (l'unica che non ce l'ha e' `Object`);
- `getInterfaces()` ritorna tutte le interfacce che implementano l'oggetto;
- `getFields()` ritorna i campi/attributi visibili (tipicamente lo sono mai);
- `getConstructors()` ritorna costruttori della classe;
- `getMethods()` ritorna un'array di descrittori di metodi;
- `getParameters()` ritorna i parametri del singolo metodo;
- `getType()` array tipo dei parametri.

```

// ...
private void dump(Class<?> clazz) {
    System.out.println("Class: " + clazz.getName());
    Class<?> baseClass = clazz.getSuperclass();
    if (baseClass != null)
        System.out.println("Base class: " + baseClass.getName());
    Class<?>[] interfaces = clazz.getInterfaces();
    for (int i = 0; i < interfaces.length; ++i)
        System.out.println("Implemented interface: " +
            interfaces[i].getName());
    Field[] fields = clazz.getFields();
    for (int i = 0; i < fields.length; ++i) {
        Field field = fields[i];
        Class<?> fieldClass = field.getType();
        System.out.println("Field: " + fieldClass.getName() +
            " " + field.getName());
    }
    Constructor<?>[] constructors = clazz.getConstructors();
    for (int i = 0; i < constructors.length; ++i) {
        Constructor<?> constructor = constructors[i];
        System.out.print("Constructor: " + constructor.getName()
            + "(");
        Parameter[] parameters = constructor.getParameters();
        for (int j = 0; j < parameters.length; ++j) {
            Parameter parameter = parameters[j];
            Class<?> parameterClass = parameter.getType();
            System.out.print(parameterClass.getName() + " " +
                parameter.getName());
            if (j != parameters.length - 1)
                System.out.print(", ");
        }
        System.out.println(")");
    }
    Method[] methods = clazz.getMethods();
    for (int i = 0; i < methods.length; ++i) {
        Method method = methods[i];
        Class<?> resultType = method.getReturnType();
        System.out.print("Method: " + resultType.getName() +
            " " + method.getName() + "(");
        Parameter[] parameters = method.getParameters();
        for (int j = 0; j < parameters.length; ++j) {
            Parameter parameter = parameters[j];

```

```
        Class<?> parameterClass = parameter.getType();
        System.out.print(parameterClass.getName() + " "
            + parameter.getName());
        if (j != parameters.length - 1)
            System.out.print(", ");
    }
    System.out.println("");
}
// ...
```