

Example09

Table of contents

- [`it.unipr.informatica.examples.model`](#)
 - [`Book.java`](#)
 - [`Student.java`](#)
- [`it.unipr.informatica.examples.model.simple`](#)
 - [`SimpleBook.java`](#)
 - [`SimpleStudent.java`](#)
- [`it.unipr.informatica.beans`](#)
 - [`Bean.java`](#)
 - [`BeanLoader.java`](#)
- [`it.unipr.informatica.examples`](#)
 - [`Example09.java`](#)

Dynamic Object Creation

Seguendo le nozioni imparate in [Example08](#), se abbiamo un descrittore di classe possiamo con il metodo `newInstance()` creare una nuova istanza della classe.

Questa tecnica ci permette di, non conoscendo una classe a tempo d'esecuzione, farla conoscere e quindi evitare il down cast.

Example

```
String s = String.class.newInstance()
```

`String.class` e' il descrittore della classe a tempo d'esecuzione, siccome lo abbiamo possiamo chiamare `newInstance()`, non c'e' bisogno di nessun cast siccome `s` e' una stringa.

Una volta creati oggetti usando il descrittore di classe, possiamo anche accedere ai campi (da limitare a campi costanti). Possiamo scegliere un campo singolo della classe `c` oppure possiamo fare `getField(o)` e passando il nome del campo, prendere quello specifico. Viene ritornato un oggetto di classe `Field f`:


- `f.get(o)` per leggere il valore corrente del campo per l'oggetto `o`;
- `f.set(o, v)` dove `o` contiene il campo e `v` il nuovo valore che vogliamo assegnare, per assegnare un nuovo valore.

Siccome viene ritornato un `Object`, in questo caso il down cast si rende necessario. Una volta che l'abbiamo in mano, possiamo invocare un metodo a tempo d'esecuzione, magari, tramite il suo nome, e lo facciamo tramite `Reflection`.

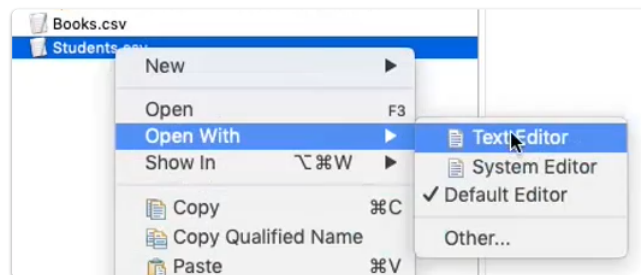
Se `m` e' un descrittore di metodo, possiamo fare `m.invoke(o, a)` per invocare il metodo descritto da `m` sull'oggetto `o` passando come argomenti `a`.

Gli argomenti vengono passati come array di oggetti, ma possiamo anche passare direttamente `n` argomenti se ci serve.

Example09

Vogliamo leggere e scrivere dei dati relativi agli studenti e libri in un file, decidendo che sia in formato CSV . Un elenco di valori viene separato da una virgola (oppure TAB) e a capo andiamo per dire che la riga e' finita; il file contiene una matrice.


Supponiamo di voler aprire questo file con foglio di calcolo, la maggior parte delle volte si vede male. Se facciamo 'Import' possiamo dire cosa usare come separatore. Per ora ci concentriamo sul visualizzarlo con Eclipse.



```
-- Students.csv --
ID      Surname Name
1       Verdi   Giuseppe
2       Bianchi Matteo
3       Neri    Alessio
4       Rossi   Mario

-- Books.csv --
ID      Author  Title
1       Alessandro Manzoni    I promessi sposi
2       Dante Alighieri La divina commedia
3       Giovanni Verga  I Malavoglia
4       Italo Svevo     La coscienza di Zeno
5       Carlo Collodi   Pinocchio
```

La prima riga identifica le colonne, in successione le tuple separate da a capo.

L'importante che ci sia un TAB. Quello che vogliamo fare e' costruire oggetti che descrivano le righe dei file CSV  e aggiungere questi in una lista.

Per farlo dobbiamo cominciare a strutturare il codice definendo le interfacce dei dati e poi la loro implementazione piu' semplice `simple`.

La descrizione astratta dei dati e' in `model`.

`it.unipr.informatica.examples.model`

Solo interfacce. Nessuna implementazione, siccome quella va a dire come vengono memorizzati i dati, che in futuro potrebbe cambiare (oggi e' CSV , magari domani in un DATABASE .

Quando abbiamo un attributo in una classe, e vogliamo leggerlo e/o scriverlo, normalmente quello che si mette e' un metodo `get()` per leggere e uno `set()` per scriverlo. Sarebbero i *getter* e *setter*.

Le letture sono concesse, le scritture bisogna stare attenti perche' potremmo suggerire erroneamente all'utente, che queste siano permanenti quando in verita' non lo sono.

Book.java

Contiene 3 campi/metodi:

- identificativo;
- autore;
- titolo.

Faremo una classe che implementa `Book`, mettendoci 3 attributi privati (identificativo, autore, titolo), un costruttore che li prenda, implementeremo i getter.

```
package it.unipr.informatica.examples.model;
import it.unipr.informatica.beans.Bean;

public interface Book extends Bean {
    public int getID();
    public String getAuthor();
    public String getTitle();
}
```

Student.java

3 campi anche qui:


- identificativo;
- nome;
- cognome.

```
package it.unipr.informatica.examples.model;
import it.unipr.informatica.beans.Bean;

public interface Student extends Bean {
    public int getID();
    public String getName();
    public String getSurname();
}
```

it.unipr.informatica.examples.model.simple

Implementazione semplice che si limita a memorizzare in memoria i dati andando a prevedere che questi vengano caricati da un file preesistente.

Stiamo facendo le `Struct` del C, in JAVA  vengono invece chiamate `beans` (chicco di caffè'), oggetti che memorizzano dati che sicuramente possono essere letti, a volte possiamo anche scriverci sopra.

SimpleBook.java

Lo chiamiamo `Simple` perche' quello piu' semplice.

Sarebbe un tipo di dato che viene chiamato *oggetto valore*: se si comporta nei termini dei suoi metodi come un valore di un tipo primitivo, per essere in grado di confrontare 2 valori, per ottenere un numero univoco del valore, per trasformare in una stringa leggibile. Deve essere fornita una semantica corretta a tutti i metodi di `Object` che conosciamo.

I 3 campi che abbiamo menzionato ci sono.

Abbiamo poi un costruttore `SimpleBook` che prende `id`, `author` e `title` verificando che siano validi per poi riempire l'oggetto.

Poi i getter, per ottenere i 3 campi.

Il costruttore senza argomenti `SimpleBook()` viola il contratto controllato dal costruttore con argomenti (siccome sono tutti vuoti), pensato per un solo scopo: quello di caricare.

```
package it.unipr.informatica.examples.model.simple;

import it.unipr.informatica.examples.model.Book;

public class SimpleBook implements Book, Cloneable {
    private int id;
    private String author;
    private String title;
    // no parametri (per caricare)
    public SimpleBook() {
        this.id = 0;
        this.author = this.title = "";
    }
    // con parametri (controllo valori)
    public SimpleBook(int id, String author, String title) {
        if (id < 1)
            throw new IllegalArgumentException("id < 1");
        if (author == null || author.length() == 0)
            throw new IllegalArgumentException
                ("author == null || author.length() == 0");
        if (title == null || title.length() == 0)
            throw new IllegalArgumentException
                ("title == null || title.length() == 0");
        this.id = id;
        this.author = author;
        this.title = title;
    }
    // ...
}
```

Per ottenere i dati nei beans.

Verifichiamo i contratti sugli argomenti e riempiamo con dati `id`, `author` e `title`.

```
// ...
@Override
public int getID() {
    return id;
}
```

```

@Override
public String getAuthor() {
    return author;
}

@Override
public String getTitle() {
    return title;
}
// ...

```

Ripetiamo che nell'interfaccia non ci sono i setter perche' prevedono una decisione: cosa succede nel supporto persistente? Siccome non siamo nelle condizioni di deciderlo, non li mettiamo.

```

// ...
public void setID(int id) {
    if (id < 1)
        throw new IllegalArgumentException("id < 1");
    this.id = id;
}

public void setAuthor(String author) {
    if (author == null || author.length() == 0)
        throw new IllegalArgumentException
            ("author == null || author.length() == 0");
    this.author = author;
}

public void setTitle(String title) {
    if (title == null || title.length() == 0)
        throw new IllegalArgumentException
            ("title == null || title.length() == 0");

    this.title = title;
}
// ...

```

Siccome sono oggetti valore, dobbiamo usare semantica corretta ai metodi di `Object`. Quindi:

- `toString()` ritorna una descrizione testuale dell'oggetto, una semplice, che nel nostro caso sono i nostri 3 campi;
- `hashCode()` per ottenere valore a 32bit che partendo dallo stato dell'oggetto, genera un numero uguale per oggetti con stesso stato, potenzialmente diverso per oggetti con uno diverso;
- `clone()` e' `protected` in `Object`, nel momento in cui vogliamo renderlo disponibile, deve essere implementato (serve a salvaguardare l'utilizzo della copia dei dati, in JAVA si cerca sempre di minimizzare il suo uso);
- `equals(Object other)` deve essere un qualcosa che verifica l'uguaglianza degli stati, sono uguali solamente se vengono dalla stessa classe (2 istanze della stessa classe).

```

// ...
@Override
public String toString() {
    return "ID=" + id + ", author=" +
        author + ", title=" + title;
}

@Override
public int hashCode() {
    return id + title.hashCode() + author.hashCode();
}

// copia profonda
@Override
public SimpleBook clone() {
    return new SimpleBook(id, author, title);
}

@Override
public boolean equals(Object other) {
    if (!(other instanceof SimpleBook))
        return false;
    SimpleBook otherBook = (SimpleBook) other;
    return id == otherBook.id &&
        title.equals(otherBook.title) &&
        author.equals(otherBook.author);
}
}

```

SimpleStudent.java

Funziona nello stesso modo.

it.unipr.informatica.beans

Ci sono dei frameworks che finiscono con nome `beans` che permettono di dire "questo oggetto e' un `bean`", un qualche cosa che e' piccolo, atomico e non si romperà mai.

Bean.java

Per dichiarare quali sono i dati della nostra applicazione, infatti sia `Book.java` che `Student.java`, la estendono.

```

package it.unipr.informatica.beans;

public interface Bean {
}

```

BeanLoader.java

Vuole una classe di tipo `<T>`, nome di un file.

Quello che ritorna e' una lista di `T` e puo' lanciare eccezione nel caso in cui il file non sia del tipo giusto o non sia formattato giusto, sia vuoto.

Cominciamo a vincolare gli argomenti utilizzati nei generici, `T` deve estendere `bean`, convertibile a `bean`. Permette di vincolare il tipo dell'argomento del generico permettendoci di escludere e permettere dinamiche.

Possiamo chiamare metodi senza bisogno di cast.

Con lo `Scanner` leggiamo la 1^a linea, spezzandola con TAB.

Costruiamo un array di classi corrispondenti ai nomi trovati in array di property.

`propertyNames` separa le 3 colonne che gli diamo e passandogli lo studente, viene controllato il tipo di ritorno e fatte i metodi in base a quelli che vengono trovati con `propertyTypes`.

Il bean vuoto serve per poi leggere le righe vuote e riempirlo.

```
public class BeanLoader {
    public <T extends Bean> List<T> load(Class<T> clazz,
        String fileName) throws IOException {
        try (InputStream inputStream = new FileInputStream(fileName);
            Scanner scanner = new Scanner(inputStream)) {
            scanner.useLocale(Locale.US);
            String heading = scanner.nextLine();
            String[] propertyNames = split(heading);
            Class<?>[] propertyTypes = getPropertyTypes
                (clazz, propertyNames);
            List<T> result = new ArrayList<T>();
            while (scanner.hasNext()) {
                T bean = (T) clazz.newInstance();
                String line = scanner.nextLine();
                String[] values = split(line);
                if (propertyNames.length != values.length)
                    throw new IOException("invalid file
format");

                for (int i = 0; i < propertyNames.length; ++i) {
                    Method method = clazz.getMethod
                        ("set" + propertyNames[i],
propertyTypes[i]);

                    Object value = fromString(values[i],
                        propertyTypes[i]);
                    method.invoke(bean, value);
                }
                result.add(bean);
            }
            return result;
        } catch (IOException exception) {
            throw exception;
        } catch (Throwable throwable) {
            throw new IOException(throwable);
        }
    }
    // ...
}
```

Prende una stringa, chiama `split()` su questa spezzandola quando viene trovato il TAB. Nel caso piu' semplice mettiamo qualcosa identificativo per separare i valori. Chiamiamo `trim()` per togliere i caratteri non stampabili davanti e dietro (come spazi).

```
// ..
private String[] split(String line) {
    String[] values = line.split("\t");
    for (int i = 0; i < values.length; ++i)
        values[i] = values[i].trim();
    return values;
}
// ...
```

Viene passata la classe e il nome delle property.

Con questa chiamata vengono letti i descrittori dei metodi della classe.

Costruiamo array di descrittori che utilizzeremo come valore di ritorno.

Scorriamo i metodi fintanto che il nome del metodo finale ha unico argomento e unico valore di ritorno `void`. Prendiamo i parametri, il tipo del parametro e' quello della property. Stiamo impostando i nomi delle colonne e verificando i tipi.

```
// ...
private <T> Class<?>[] getPropertyTypes
(Class<T> clazz, String[] propertyNames) throws IOException {
    Method[] methods = clazz.getMethods();
    Class<?>[] propertyTypes = new Class<?>
        [propertyNames.length];
    for (int i = 0; i < propertyNames.length; ++i) {
        String propertyName = propertyNames[i];
        String methodName = "set" + propertyName;
        for (int j = 0; j < methods.length; ++j) {
            Method method = methods[j];
            if (method.getParameterCount() == 1
                && method.getReturnType() == Void.TYPE
                && methodName.equals(method.getName())) {
                Class<?>[] parameterTypes =
                    method.getParameterTypes();
                propertyTypes[i] = parameterTypes[0];
                break;
            }
        }
        if (propertyTypes[i] == null)
            throw new IOException
                ("invalid property name " + propertyName);
    }
    return propertyTypes;
}
// ...
```

Data una stringa, vogliamo costruire l'oggetto che le corrisponde sfruttando i tipi. Se la classe e' `String` allora siamo a posto cosi', altrimenti se e' qualcosa di diverso, convertiamo. Non facciamo tutte le conversioni, mancano per esempio `Char` e `Boolean`, ma possono essere aggiunti.

```
// ...
protected Object fromString(String text, Class<?> clazz) {
    if (clazz == String.class)
        return text;
```



```

        if (clazz == Integer.TYPE)
            return Integer.parseInt(text);
        if (clazz == Float.TYPE)
            return Float.parseFloat(text);
        if (clazz == Double.TYPE)
            return Double.parseDouble(text);
        throw new IllegalArgumentException
            ("cannot convert " + text + " to " + clazz.getName());
    }
}

```

it.unipr.informatica.examples

Example09.java

Carica tutti gli studenti, carica tutti i libri.

Viene specificata la classe per tutti e due e poi stampa.

```

package it.unipr.informatica.examples;
import java.util.List;
import it.unipr.informatica.beans.Bean;
import it.unipr.informatica.beans.BeanLoader;
import it.unipr.informatica.examples.model.simple.SimpleBook;
import it.unipr.informatica.examples.model.simple.SimpleStudent;


public class Example09 {
    private void go() {
        try {
            BeanLoader loader = new BeanLoader();
            List<SimpleStudent> studentBeans =
                loader.load(SimpleStudent.class, "Students.csv");
            for (Bean bean : studentBeans)
                System.out.println(bean);
            System.out.println();
            List<SimpleBook> bookBeans =
                loader.load(SimpleBook.class, "Books.csv");
            for (Bean bean : bookBeans)
                System.out.println(bean);
        } catch (Throwable throwable) {
            System.err.println
                ("Cannot load beans with message " +
                 throwable.getMessage());
        }
    }

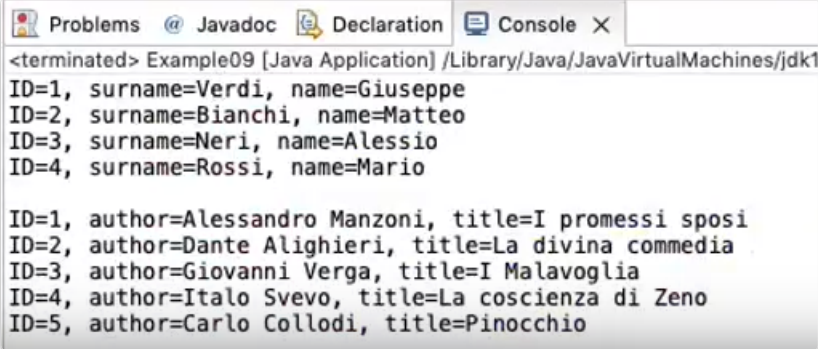
    public static void main(String[] args) {
        new Example09().go();
    }
}

```

Viene stampata con descrizione a stringa, tramite `toString()`.

Questo meccanismo e' lo stesso che usiamo per gli oggetti persistenti, come per un DATABASE

 relazionale. Fare un loader per leggere i dati serve usare una query per leggere il nome delle colonne, i record e il tipo di questi.



```
<terminated> Example09 [Java Application] /Library/Java/JavaVirtualMachines/jdk1
ID=1, surname=Verdi, name=Giuseppe
ID=2, surname=Bianchi, name=Matteo
ID=3, surname=Neri, name=Alessio
ID=4, surname=Rossi, name=Mario

ID=1, author=Alessandro Manzoni, title=I promessi sposi
ID=2, author=Dante Alighieri, title=La divina commedia
ID=3, author=Giovanni Verga, title=I Malavoglia
ID=4, author=Italo Svevo, title=La coscienza di Zeno
ID=5, author=Carlo Collodi, title=Pinocchio
```