← Serializzazione e deserializzazione JSON di oggetti Java con Jackson: un esempio concreto

Java Swing: un esempio di utilizzo del GridBagLayout per la realizzazione di un form →

JSON serialization and deserialization of JAVA objects with Jackson: a concrete example

Posted on December 17, 2014 by Davis Molinari

[Questo articolo è disponibile anche in ITALIANO: Serializzazione e deserializzazione JSON di oggetti Java con Jackson: un esempio concreto]

In this post we see how to use the Jackson framework to serialize simple Java objects (POJOs – plain Old Java Object) in **JSON** (JavaScript Object Notation), the textual format quick and easy to process for data exchange between applications, and also how to perform the inverse transformation, from a text string in JSON format containing the object information to the Java object itself(deserialization).

To do this we create some simple class with which we represent an e-commerce orders management system. So we will have orders, customers, the list of the items included in the order, with their quantity, price, etc .. So, let's start to define the classes we need. We start from the Order class that will contain the order identifier, the customer who placed the order, the list of the lines that compose the order, the total amount of the order and the date on which the order was placed.

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Order {

    private long id;
    private Customer customer;
    private List<OrderItem> itemList;
    private double total;
    private Date placedDate;

    // I already know the total, so I use it in th
    public Order(long id, Customer cust, List<Orde
        this.id = id;
        this.customer = cust;
        this.itemList = li;
        this.placedDate = pDate;
        this.total = total;
    }

    // This constructor calculates the total by ad
    public Order(long id, Customer cust, List<Orde
        this.id = id;
        this.customer = cust;
        this.placedDate = pDate;

        // this.itemList = li;
        // this.total = this.total;
        for (OrderItem oi : li) {
            this.addItemToOrder(oi);
        }
```

```
    }

    public long getId() {
        return this.id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public Customer getCustomer() {
        return this.customer;
    }

    public void setCustomer(Customer cust) {
        this.customer = cust;
    }

    public List<OrderItem> getItemList() {
        return this.itemList;
    }

    public void setItemList(List<OrderItem> itemLi
        this.itemList = itemList;
    }

    public void addItemToOrder(OrderItem oi) {
        if (this.itemList == null) {
            this.itemList = new ArrayList<>();
        }
        this.itemList.add(oi);
        this.total += oi.getPrice();
    }

    public Date getPlacedDate() {
        return this.placedDate;
    }

    public void setPlacedDate(Date placedDate) {
        this.placedDate = placedDate;
    }

    public double getTotal() {
```

```
        return this.total;
    }

    public void setTotal(double total) {
        this.total = total;
    }

    @Override
    public String toString() {
        StringBuilder ret = new StringBuilder();
        ret.append("ORDER ID: ").append(this.id).a
        ret.append("ORDER DATE: ").append(this.pla
        ret.append("CUSTOMER: ").append(this.custo
        ret.append("ITEMS:\n");
        for (OrderItem oi : this.itemList) {
            ret.append("\t").append(oi).append("\n
        }
        ret.append("TOTAL: ").append(this.total).a
        return ret.toString();
    }
}
```

We have redefined the toString method to obtain an explanatory text representation of our Order objects.

At this point we define the OrderItem class representing individual lines of an order which will be constituted by the purchased item, the relative number of pieces and the total price for this row, given by the multiplication between the quantity and price of each item.

```
public class OrderItem {

    private int quantity;
    private Item it;
    private double price;

    public OrderItem(int q, Item it){
        this.quantity = q;
        this.it = it;
```

```
        this.price = q*it.getPrice();
    }

    public int getQuantity() {
        return this.quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public Item getIt() {
        return this.it;
    }

    public void setIt(Item it) {
        this.it = it;
    }

    public double getPrice() {
        return this.price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString(){
        return "n° " + this.quantity + " " + this.
    }
}
```

We continue in the definition of classes with Item, which simply represents an item available for purchase. His fields are an identifier, the name of the object itself, its category of membership and its unit price. The category is represented by an enumerator that we will define shortly after.

```
public class Item {
    private long id;
    private String name;
    private Categories cat;
    private double price;

    public Item (long id, String n, Categories c,
        this.id = id;
        this.name = n;
        this.cat = c;
        this.price = p;
    }

    public long getId() {
        return this.id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Categories getCat() {
        return this.cat;
    }

    public void setCat(Categories cat) {
        this.cat = cat;
    }

    public double getPrice() {
        return this.price;
    }

    public void setPrice(double price) {
```

```
        this.price = price;
    }

    @Override
    public String toString(){
        return this.name + " (unit price: " + this
    }
}
```

Now, we define the enumerator of the categories:

```
public enum Categories {
    SPORT, BOOK, HARDWARE
}
```

Finally, we define the Customer class, which represents clients that we identify simply by an id and their name and surname.

```
public class Customer {

    private long id;
    private String firstName;
    private String lastName;


    public Customer(long id, String fn, String ln)
        this.firstName = fn;
        this.lastName = ln;
        this.id = id;
    }

    public long getId() {
        return this.id;
    }

    public void setId(long id) {
        this.id = id;
```

```
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Override
    public String toString() {
            return this.firstName + " " + this.las
    }
}
```

Now that we have defined the necessary classes, we create a small test program that creates an Order object and prints it on the screen:

```
import java.util.ArrayList;
import java.util.Date;

public class JacksonExample {

    public static void main(String[] args) {
        Customer c1 = new Customer(1,"Davis", "Mol

        Item i1 = new Item(1, "Tablet XYZ", Catego
        Item i2 = new Item(2, "Jackson Tutorial",
        Item i3 = new Item(3, "Running shoes", Cat
```

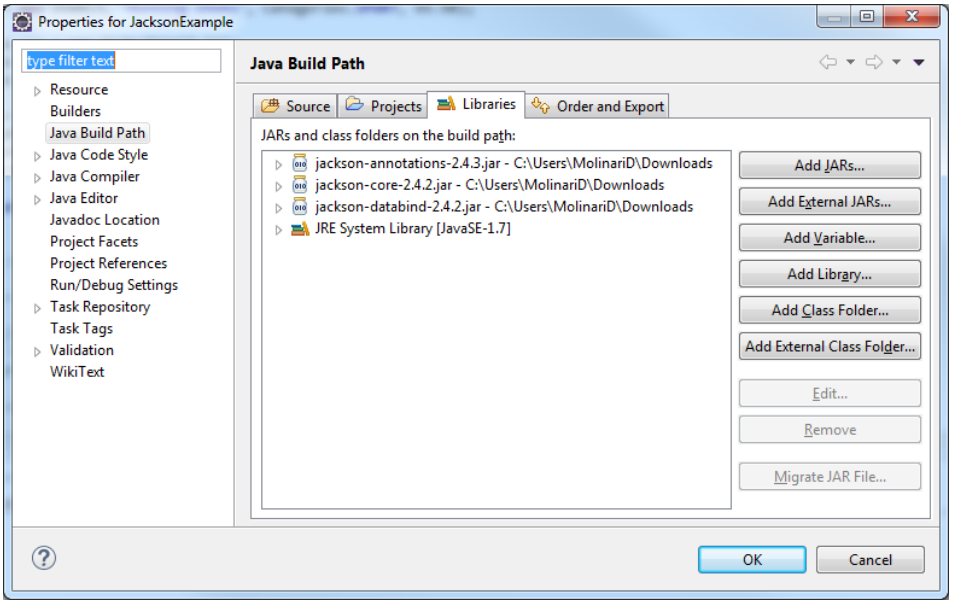
```
        OrderItem oi1 = new OrderItem(2,i1);
        OrderItem oi2 = new OrderItem(3,i2);
        OrderItem oi3 = new OrderItem(1,i3);

        Order o = new Order(1000, c1, new ArrayLis
        o.addItemToOrder(oi1);
        o.addItemToOrder(oi2);
        o.addItemToOrder(oi3);

        System.out.println(o);
    }
}
```

Trying to run the program we get the following output:

```
ORDER ID: 1000
ORDER DATE: Mon Dec 15 15:47:39 CET 2014
CUSTOMER: Davis Molinari
ITEMS:
        n° 2 Tablet XYZ (unit price: 199.0 - cat:
        n° 3 Jackson Tutorial (unit price: 19.0 -
        n° 1 Running shoes (unit price: 65.5 - cat
TOTAL: 520.5
```

Ok, now let's go back to our goal, which is to provide an example of **JSON** serialization with **Jackson**.
First, add the Jackson libraries to the Build Path of our project; the jar to be included, as shown in the following figure, are those of the three modules in which the Jackson project has been divided:

– Core
– DataBind
– Annotations



Once we have added the libraries to the project, we modify our test program to perform the JSON serialization of our Order object.
What we need to do is to create an **ObjectMapper** and invoke its **writeValue** method, passing as parameters the stream where to write the value (in our case System.out, the standard output) and the object to be serialized (our Order object).
Here's the updated code of our test program:

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;

import com.fasterxml.jackson.core.JsonGenerationEx
import com.fasterxml.jackson.databind.JsonMappingE
import com.fasterxml.jackson.databind.ObjectMapper

public class JacksonExample {

    public static void main(String[] args) {
        ObjectMapper mapper = new ObjectMapper();

        Customer c1 = new Customer(1,"Davis", "Mol

        Item i1 = new Item(1, "Tablet XYZ", Catego
        Item i2 = new Item(2, "Jackson Tutorial",
        Item i3 = new Item(3, "Running shoes", Cat
```

```
        OrderItem oi1 = new OrderItem(2,i1);
        OrderItem oi2 = new OrderItem(3,i2);
        OrderItem oi3 = new OrderItem(1,i3);

        Order o = new Order(1000, c1, new ArrayLis
        o.addItemToOrder(oi1);
        o.addItemToOrder(oi2);
        o.addItemToOrder(oi3);

        try {
            mapper.writeValue(System.out, o);
        }
        catch (JsonGenerationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        catch (JsonMappingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Here we see the (very long) string resulting from the serialization process:

```
{"id":1000,"itemList":[{"quantity":2,"it":{"id":1,
```

Displayed in this way, the string isn't simple to analyze and for this reason we ask the object of ObjectMapper to format the output in a more readable way, just for verification purposes.
To do that we set the mapper as follows:

```
mapper.configure(SerializationFeature.INDENT_OUTPU
```

or, alternatively:

```
mapper.enable(SerializationFeature.INDENT_OUTPUT);
```

Running again our test program we get this time the string formatted as follows:

```
{
  "id" : 1000,
  "customer" : {
    "id" : 1,
    "firstName" : "Davis",
    "lastName" : "Molinari"
  },
  "itemList" : [ {
    "quantity" : 2,
    "it" : {
      "id" : 1,
      "name" : "Tablet XYZ",
      "cat" : "HARDWARE",
      "price" : 199.0
    },
    "price" : 398.0
  }, {
    "quantity" : 3,
    "it" : {
      "id" : 2,
      "name" : "Jackson Tutorial",
      "cat" : "BOOK",
      "price" : 19.0
    },
    "price" : 57.0
  }, {
    "quantity" : 1,
    "it" : {
```

```
        "id" : 3,
        "name" : "Running shoes",
        "cat" : "SPORT",
        "price" : 65.5
      },
      "price" : 65.5
    } ],
    "total" : 520.5,
    "placedDate" : 1419260209564
  }
```

As we can see our Order object is represented by an associative array of key-value pairs, enclosed in braces. The first field, "id" is a primitive type so that its value is just reported. The field "customer" instead, consists in an object of the Customer class, so in the serialization process as a new object is defined, opening a new brace, inside of which are represented the Customer object attributes. The next field of the Order class to represent in the JSON string is "itemList" which at the class level is defined as a list of OrderItem. The lists are represented in JSON within square brackets, so in our serialization string we have the field "itemList" displayed by a pair of square brackets within which are listed, separated by commas, the representations of OrderItem objects. Each OrderItem element, being an object of a class, is enclosed in curly braces and contains the fields "quantity", "it" enclosed in braces because, again, is an Item object, and "price". Finally the Order object provides the last two simple fields, "total" and "placedDate". On this last field, representing the date on which the order was placed, we notice something strange. The date is represented in the "computer age" format, so as the number of milliseconds since 01-01-1970. To give it a more readable we have to act again on our ObjectMapper object, setting the desired format.

```
SimpleDateFormat sdf = new SimpleDateFormat("dd MM
mapper.setDateFormat(sdf);
```

Running again our test program for the serialization of our Order object we get the following result:

```
{
  "id" : 1000,
  "customer" : {
    "id" : 1,
    "firstName" : "Davis",
    "lastName" : "Molinari"
  },
  "itemList" : [ {
    "quantity" : 2,
    "it" : {
      "id" : 1,
      "name" : "Tablet XYZ",
      "cat" : "HARDWARE",
      "price" : 199.0
    },
    "price" : 398.0
  }, {
    "quantity" : 3,
    "it" : {
      "id" : 2,
      "name" : "Jackson Tutorial",
      "cat" : "BOOK",
      "price" : 19.0
    },
    "price" : 57.0
  }, {
    "quantity" : 1,
    "it" : {
      "id" : 3,
      "name" : "Running shoes",
      "cat" : "SPORT",
      "price" : 65.5
    },
    "price" : 65.5
  } ],
  "total" : 520.5,
  "placedDate" : "15 dic 2014"
}
```

As we can see from the generated output, the date this time is represented by the specified format and is therefore readable:

```
"placedDate" : "15 dic 2014"
```

Let's look now at the de-serialization process. First we make a change to our testing program in order to let it write the string containing the JSON serialization of our Order on a String object instead of printing it on video. For this purpose we replace the invocation of the method**writeValue** on our ObjectMapper with the invocation of the method **writeValueAsString** that gives us the JSON as a String:

```
import java.util.Date;

import com.fasterxml.jackson.core.JsonProcessingEx
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.Serializatic

public class JacksonExample {

    public static void main(String[] args) {
        ObjectMapper mapper = new ObjectMapper();
        mapper.configure(SerializationFeature.INDE

        SimpleDateFormat sdf = new SimpleDateForma
        mapper.setDateFormat(sdf);

        Customer c1 = new Customer(1,"Davis", "Mol

        Item i1 = new Item(1, "Tablet XYZ", Catego
        Item i2 = new Item(2, "Jackson Tutorial",
        Item i3 = new Item(3, "Running shoes", Cat

        OrderItem oi1 = new OrderItem(2,i1);
        OrderItem oi2 = new OrderItem(3,i2);
        OrderItem oi3 = new OrderItem(1,i3);
```

```
        Order o = new Order(1000, c1, new ArrayLis
        o.addItemToOrder(oi1);
        o.addItemToOrder(oi2);
        o.addItemToOrder(oi3);

        String s = null;
        try {
            s = mapper.writeValueAsString(o);
        }
        catch (JsonProcessingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println(s);

    }
}
```

The result is the same as before, when we wrote the JSON directly to standard output by the method writeValue.

At this point we have our JSON saved in a String, so we can try to make deserialization using the **readValue**. We modify our test program telling it to create an Order java oject starting from its JSON representation saved in the string.

```
package dede.example;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;

import com.fasterxml.jackson.core.JsonParseExcepti
import com.fasterxml.jackson.core.JsonProcessingEx
import com.fasterxml.jackson.databind.JsonMappingE
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.Serializatio
```

```
public class JacksonExample {

    public static void main(String[] args) {
        ObjectMapper mapper = new ObjectMapper();
        mapper.configure(SerializationFeature.INDE

        SimpleDateFormat sdf = new SimpleDateForma
        mapper.setDateFormat(sdf);

        Customer c1 = new Customer(1,"Davis", "Mol

        Item i1 = new Item(1, "Tablet XYZ", Catego
        Item i2 = new Item(2, "Jackson Tutorial",
        Item i3 = new Item(3, "Running shoes", Cat

        OrderItem oi1 = new OrderItem(2,i1);
        OrderItem oi2 = new OrderItem(3,i2);
        OrderItem oi3 = new OrderItem(1,i3);

        Order o = new Order(1000, c1, new ArrayLis
        o.addItemToOrder(oi1);
        o.addItemToOrder(oi2);
        o.addItemToOrder(oi3);

        System.out.println(o);
        System.out.println("------------");

        String s = null;
        try {
            s = mapper.writeValueAsString(o);
        }
        catch (JsonProcessingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println(s);
        System.out.println("------------");

        Order o2 = null;
        try {
            o2 = mapper.readValue(s, Order.class);
        }
```

```
        catch (JsonParseException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        catch (JsonMappingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println(o2);
    }
}
```

Executing the program we get this result:

```
com.fasterxml.jackson.databind.JsonMappingExceptic
[simple type, class dede.example.Order]: can not i
(need to add/enable type information?)
```

This means that we must indicate which constructor we want to use to instantiate an object of Order class, starting from its JSON representation. To do this we must annotate the constructor to be used with the Jackson annotation **"@JsonCreator"** and annotate every constructor parameter with the Jackson annotation **"@JsonProperty"** combined with class property that parameter maps.

Now we modify the Order class constructor as follows:

```
    @JsonCreator
    public Order(@JsonProperty("id")long id, @Json
        this.id = id;
        this.customer = cust;
```

```
        this.itemList = li;
        this.placedDate = pDate;
        this.total = this.total;
    }
```

Trying to re-run the test program we get:

```
com.fasterxml.jackson.databind.JsonMappingExceptic
[simple type, class dede.example.Customer]: can no
(need to add/enable type information?)
```

As we can see, this time the error was “moved” to the constructor of the Customer class, so the configuration that we have provided for the Order class constructor was correct. We repeat the same configuration through Jackson annotations for the constructors of all our classes. They then become as follows:

```
    @JsonCreator
    public Customer(@JsonProperty("id")long id, @J
        this.firstName = fn;
        this.lastName = ln;
        this.id = id;
    }
```

```
    @JsonCreator
    public OrderItem(@JsonProperty("quantity")int
        this.quantity = q;
        this.it = it;
        this.price = q*it.getPrice();
    }
```

```
    @JsonCreator
    public Item (@JsonProperty("id")long id, @Json
        this.id = id;
        this.name = n;
        this.cat = c;
        this.price = p;
    }
```

Once configured all constructors to be used use during deserialization process, we run again our test program and we get the following result:

```
ORDER ID: 1000
ORDER DATE: Mon Dec 15 15:47:39 CET 2014
CUSTOMER: Davis Molinari
ITEMS:
        n° 2 Tablet XYZ (unit price: 199.0 - cat:
        n° 3 Jackson Tutorial (unit price: 19.0 -
        n° 1 Running shoes (unit price: 65.5 - cat
TOTAL: 520.5

------------
{
  "id" : 1000,
  "customer" : {
    "id" : 1,
    "firstName" : "Davis",
    "lastName" : "Molinari"
  },
  "itemList" : [ {
    "quantity" : 2,
    "it" : {
      "id" : 1,
      "name" : "Tablet XYZ",
      "cat" : "HARDWARE",
      "price" : 199.0
    },
    "price" : 398.0
  }, {
```

```
      "quantity" : 3,
      "it" : {
        "id" : 2,
        "name" : "Jackson Tutorial",
        "cat" : "BOOK",
        "price" : 19.0
      },
      "price" : 57.0
    }, {
      "quantity" : 1,
      "it" : {
        "id" : 3,
        "name" : "Running shoes",
        "cat" : "SPORT",
        "price" : 65.5
      },
      "price" : 65.5
    } ],
    "placedDate" : "07 gen 2015",
    "total" : 520.5
}
------------
ORDER ID: 1000
ORDER DATE: Mon Dec 15 00:00:00 CET 2014
CUSTOMER: Davis Molinari
ITEMS:
        n° 2 Tablet XYZ (unit price: 199.0 - cat:
        n° 3 Jackson Tutorial (unit price: 19.0 -
        n° 1 Running shoes (unit price: 65.5 - cat
TOTAL: 520.5
```

The result, as you can see, shows:
– The toString of the Order java class object
– The JSON representation of the object obtained from serialization process
– The toString of the new Order java object created through deserialization of JSON string

The complete code example is available for download here:

Jackson JSON serialization example 4.25 KB
Download

This entry was posted in 1s. Bookmark the permalink.

3 thoughts on "JSON serialization and deserialization of JAVA objects with Jackson: a concrete example"

Pingback: Serializzazione e deserializzazione JSON di oggetti Java con Jackson: un esempio concreto | Dede Blog

John DeRegnaucourt says:
May 14, 2015 at 04:35

You may want to consider json-io (https://github.com/jdereg/json-io) or gson (https://code.google.com/p/google-gson/) instead of Jackson for JSON serialization. Both handle templates and interfaces much better than Jackson. Both libraries are very fast. Json-io is less than 100K library with no dependencies on other libraries (JDK only).

Reply

Davis Molinari says:
May 14, 2015 at 21:55

Hi John, thank you for your suggestions. I already know gson and I'll take a look at json-io too. I see it's your own project, so congratulations!
Davis

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

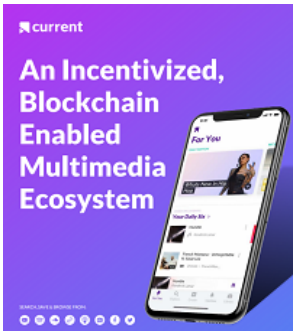
Website

Post Comment

Best Upcoming ICOs

Current

Start date: AirDropping Now! Icobench rate: 4.2 out of 5



Categories

- Around The World
- Books
- General
- HW & SW
- Music
  - Album Reviews
  - Live Reports
- Programming
  - Algorithms
    - Information Retrieval
    - Sorting
  - Angular
  - Bash
  - C
  - Cloud
    - Azure
  - Data Structures
  - Database
    - EntityFramework
    - MongoDB
    - PostgreSQL
    - SQL Server
  - Design Patterns
  - Dos
  - Excel
  - Interviews
  - Java / JEE
    - Apache
    - ActiveMQ
    - Apache Camel
    - Eclipse
    - Hibernate
    - JAXB
    - JPA
    - JSP
    - OCAJP7
    - OCPJP7

- Servlet
- Spring Boot
- Swing
- Tomcat
- JSON
- Jackson
- Mac Os X
- Multithreading
- Objective-C
- OCaml
- Python
- UML
- VBA
- Visual Basic .NET

- Running
- Travel
  - Trekking
- University
  - Artificial Intelligence
    - Case-Based Reasoning
  - Thesis
- Web
- Wordpress

Recent Posts

- ICO of the Month, March 2018: Friendz, the Digital Marketing revolution Blockchain
- ICO del Mese, Marzo 2018: Friendz, la blockchain per il Digital Marketing

- Create a modern web application with Spring Boot, MongoDB, Angular 4 and TypeScript and deploy it in cloud as Microsoft Azure Webapp – Part 3
- Creare una web application con Spring Boot, MongoDB, Angular 4 e TypeScript e deployarla in cloud come Microsoft Azure Webapp – Parte 11
- Creare una web application con Spring Boot, MongoDB, Angular 4 e TypeScript e deployarla in cloud come Microsoft Azure Webapp – Parte 10

Top Downloads

@JsonSerialize annotation example

1144 downloads | 1.55 KB

Download

Script JDK extraction no admin privileges

985 downloads | 0.50 KB

Download

Jackson JSON serialization example

675 downloads | 4.25 KB

Download

JSON subclass example with Jackson

579 downloads | 3.82 KB

Download

GridBagLayout example

533 downloads | 4.32 KB

Download

Archives

Archives

Select Month

Meta

- Log in
- Entries RSS
- Comments RSS
- WordPress.org