(http://baeldung.com)

Introduction to Javatuples

Last modified: March 15, 2017

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

A tuple is a collection of several elements that may or may not be related to each other. In other words, tuples can be considered anonymous objects.

For example, ["RAM", 16, "Astra"] is a tuple containing three elements.

In this article, we will have a quick look at a really simple library that allows us to work with the tuple based data structures, named *javatuples* (http://www.javatuples.org/index.html).

2. Built-in *Javatuples* Classes

This library provides us ten different classes that would suffice most of our requirements related to tuples:

- *Unit<A>*
- *Pair<A,B>*
- *Triplet<A,B,C>*
- *Quartet<A,B,C,D>*
- *Quintet<A,B,C,D,E>*
- *Sextet<A,B,C,D,E,F>*
- *Septet<A,B,C,D,E,F,G>*
- *Octet<A,B,C,D,E,F,G,H>*
- *Ennead<A,B,C,D,E,F,G,H,I>*
- *Decade<A,B,C,D,E,F,G,H,I,J>*

In addition to the classes above, there are two additional classes, *KeyValue<A,B>* and *LabelValue<A,B>*, which provide functionalities similar to *Pair<A,B>*, but differ in semantics.

As per the official site (http://www.javatuples.org/index.html), **all the classes in javatuples are typesafe and immutable**. Each of the tuple class implements the *Iterable*, *Serializable*, and *Comparable* interface.

3. Adding Maven Dependency

Let's add the Maven dependency to our *pom.xml*:

```
<dependency>
    <groupId>org.javatuples</groupId>
    <artifactId>javatuples</artifactId>
    <version>1.2</version>
</dependency>
```

Please check the Central Maven repository for the latest version (https://search.maven.org/#search|gav|1|g%3A%22org.javatuples%22%20AND%20a%3A%22javatuples%22).

4. Creating Tuples

Creating a tuple is really simple. We can use the corresponding constructors:

```java
Pair<String, Integer> pair = new Pair<String, Integer>("A pair", 55);
```

There is also a little less verbose and semantically elegant way of creating a tuple:

```java
Triplet<String, Integer, Double> triplet = Triplet.with("hello", 23, 1.2
```

We can also create tuples from an *Iterable*:

```java
List<String> listOfNames = Arrays.asList("john", "doe", "anne", "alex");
Quartet<String, String, String, String> quartet
  = Quartet.fromCollection(collectionOfNames);
```

Please note that **the number of items in the collection should match the type of the tuple that we want to create**. For example, we cannot create a *Quintet* using the above collection as it requires exactly five elements. Same is true for any other tuple class having a higher order than *Quintet*.

However, we can create a lower order tuple like *Pair* or a *Triplet* using the above collection, by specifying a starting index in the *fromIterable()* method:

```java
Pair<String, String> pairFromList = Pair.fromIterable(listOfNames, 2);
```

The above code will result in creating a *Pair* containing "*anne*" and "*alex*".

Tuples can be conveniently created from any array as well:

```java
String[] names = new String[] {"john", "doe", "anne"};
Triplet<String, String, String> triplet2 = Triplet.fromArray(names);
```

5. Getting Values from Tuples

Every class in *javatuples* has a *getValueX()* method for getting the values from tuples, where *X* specifies the order of the element inside the tuple. Like the indexes in arrays, the value of *X* starts from zero.

Let's create a new Quartet and fetch some values:

```
Quartet<String, Double, Integer, String> quartet
  = Quartet.with("john", 72.5, 32, "1051 SW");

String name = quartet.getValue0();
Integer age = quartet.getValue2();

assertThat(name).isEqualTo("john");
assertThat(age).isEqualTo(32);
```

As we can see, the position of "*john*" is zero, "*72.5*" is one, and so on.

Note that the *getValueX()* methods are type-safe. That means, no casting is required.

An alternative to this is the *getValue(int pos)* method. It takes a zero-based position of the element to be fetched. **This method is not type-safe and requires explicit casting**:

```
Quartet<String, Double, Integer, String> quartet
  = Quartet.with("john", 72.5, 32, "1051 SW");

String name = (String) quartet.getValue(0);
Integer age = (Integer) quartet.getValue(2);

assertThat(name).isEqualTo("john");
assertThat(age).isEqualTo(32);
```

Please note that the classes *KeyValue* and *LabelValue* have their corresponding methods *getKey()/getValue()* and *getLabel()/getValue()*.

6. Setting Values to Tuples

Similar to *getValueX()*, all classes in javatuples have *setAtX()* methods. Again, *X* is zero-based positions for the element that we want to set:

```
Pair<String, Integer> john = Pair.with("john", 32);
Pair<String, Integer> alex = john.setAt0("alex");

assertThat(john.toString()).isNotEqualTo(alex.toString());
```

The important thing here is that the return type of *setAtX()* method is the tuple type itself. This is because **the javatuples are immutable**. Setting any new value will leave the original instance intact.

7. Adding and Removing Elements from Tuples

We can conveniently add new elements to the tuples. However, this will result in a new tuple of one order higher being created:

```
Pair<String, Integer> pair1 = Pair.with("john", 32);
Triplet<String, Integer, String> triplet1 = pair1.add("1051 SW");

assertThat(triplet1.contains("john"));
assertThat(triplet1.contains(32));
assertThat(triplet1.contains("1051 SW"));
```

It is clear from the above example that adding one element to a *Pair* will create a new *Triplet*. Similarly, adding one element to a *Triplet* will create a new *Quartet*.

The example above also demonstrates the use of *contains()* method provided by all the classes in *javatuples*. This is a really handy method for verifying if the tuple contains a given value.

It is also possible to add one tuple to another using the *add()* method:

```
Pair<String, Integer> pair1 = Pair.with("john", 32);
Pair<String, Integer> pair2 = Pair.with("alex", 45);
Quartet<String, Integer, String, Integer> quartet2 = pair1.add(pair2);

assertThat(quartet2.containsAll(pair1));
assertThat(quartet2.containsAll(pair2));
```

Note the use of *containsAll()* method. It will return *true* if all the elements of *pair1* are present in *quartet2*.

By default, the *add()* method adds the element as a last element of the tuple. However, it is possible to add the element at a given position using *addAtX()* method, where *X* is the zero-based position where we want to add the element:

```
Pair<String, Integer> pair1 = Pair.with("john", 32);
Triplet<String, String, Integer> triplet2 = pair1.addAt1("1051 SW");

assertThat(triplet2.indexOf("john")).isEqualTo(0);
assertThat(triplet2.indexOf("1051 SW")).isEqualTo(1);
assertThat(triplet2.indexOf(32)).isEqualTo(2);
```

This example adds the *String* at position 1, which is then verified by the *indexOf()* method. Please note the difference in order of the types for the *Pair<String, Integer>* and the *Triplet<String, String, Integer>* after the call to *addAt1()* method call.

We can also add multiple elements using any of *add()* or *addAtX()* methods:

```
Pair<String, Integer> pair1 = Pair.with("john", 32);
Quartet<String, Integer, String, Integer> quartet1 = pair1.add("alex", 4

assertThat(quartet1.containsAll("alex", "john", 32, 45));
```

In order to remove an element from the tuple, we can use the *removeFromX()* method. Again, *X* specifies the zero-based position of the element to be removed:

```
Pair<String, Integer> pair1 = Pair.with("john", 32);
Unit<Integer> unit = pair1.removeFrom0();

assertThat(unit.contains(32));
```

8. Converting Tuples to *List/Array*

We have already seen how to convert a *List* to a tuple. Let's now see hot to convert a tuple to a *List*:

```
Quartet<String, Double, Integer, String> quartet
  = Quartet.with("john", 72.5, 32, "1051 SW");
List<Object> list = quartet.toList();

assertThat(list.size()).isEqualTo(4);
```

It is fairly simple. The only thing to note here is that we will always get a *List<Object>*, even if the tuple contains the same type of elements.

Finally, let's convert the tuple to an array:

```
Quartet<String, Double, Integer, String> quartet
  = Quartet.with("john", 72.5, 32, "1051 SW");
Object[] array = quartet.toArray();

assertThat(array.length).isEqualTo(4);
```

Clear enough, the *toArray()* method always returns an *Object[]*.

9. Conclusion

In this article, we have explored the javatuples library and observed it's simplicity. It provides elegant semantics and is really easy to use.

Make sure you check out the complete source code for this article over on GitHub (https://github.com/eugenp/tutorials/tree/master/libraries). The complete source code contains a little more examples than the ones covered here. After reading this article, the additional examples should be easy enough to understand.

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API **with Spring**"?

Enter your Email Address

>> Get the eBook

Subscribe ▾

▴ newest ▴ oldest ▴ most voted

Ramesh Babu Y

Guest

Are we really need this tuples in java ? , for this only we have abstract data type , class

+ 0 −

1 year ago

Grzegorz Piwowarek

Guest

Do we really need? Probably no but same can be said about generics or collections

+ 0 −

1 year ago

Philip Andrew

Guest

You need to say why I want to use javaturples? What's my use case?

+ 0 −

1 year ago

Grzegorz Piwowarek

Guest

Whenever you want to group a few things together, you do not need to create a separate class for this

+ 0 −

1 year ago

Alexander Sidorenko

Guest

And then, later in the code, I will need to remember what I had put here and there in my pair. I'd rather create a dto with proper fild names than use javatuples

+ 0 −

1 year ago

Grzegorz Piwowarek

Guest

Same applies to lambda expressions and anonymous stuff in general. Sometimes it makes sense to create a separate entity for one particular usecase and sometimes it does not

+ 1 −

1 year ago

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)

MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)