

# Java SE utils

Rudy De Busscher

Version 0.9.2, ??/??/2018

# Table of Contents

Release notes .....	1
0.9.2 .....	1
0.9.1 .....	1
0.9.0 .....	1
Setup .....	1
Requirements .....	1
Features .....	2
BASE64 .....	2
HEX .....	2
ByteSource .....	3
Instantiations .....	3
Reading version .....	4
Base exceptions .....	5
String utils .....	6
Collection utils .....	7
Proxy Utils .....	7
CDICheck .....	8
TestReflectionUtil .....	8

# Release notes

## 0.9.2

- Added methods taking `byte[]` for Base64 encode/decoding
- Added HEX encoding / decoding
- Added `ByteSource`, taken `byte[]` representation of some classes and create encoded (BASE64, HEX) versions
- `isEmpty()` on `char[]`
- Define outcome of `CDICheck.withinContainer` by defining system JVM parameter `atbash.utils.cdi.check`.

## 0.9.1

- Renamed `StringUtils#join` to `StringUtils.toDelimitedString`
- Added `TestReflectionUtils`
- Added `CDICheck.withinContainer`
- Added `ProxyUtils`

### Breaking changes

- Some methods of *be.atbash.util.StringUtils* have changed parameters or are renamed.

## 0.9.0

- Initial version with classes from various sources usable in any environment.

# Setup

Add the following artifact to your maven project file

```
<dependency>
  <groupId>be.atbash.utils</groupId>
  <artifactId>utils-se</artifactId>
  <version>0.9.2</version>
</dependency>
```

# Requirements

Java 7+

# Features

Utilities compiled for Java 7 and does not have any dependencies (only using SLF4J api).

The most important methods are described here. For the full list of methods, see the Javadoc or the source code.

## BASE64

Can be used to do encoding and decoding to BASE64 for Java 7 (Only Java 8 has a JVM native implementation)

```
be.atbash.util.base64.Base64Codec#encodeToString(byte[], boolean)
```

Converts a byte array to a BASE64 encoded String. The boolean defines if urlSafe characters need to be used or not.

```
byte[] be.atbash.util.base64.Base64Codec#decode(String)
byte[] be.atbash.util.base64.Base64Codec#decode(byte[])
```

Decodes a BASE64 String/byte array into a byte array.

```
be.atbash.util.base64.Base64Codec#isBase64Encoded(String)
be.atbash.util.base64.Base64Codec#isBase64Encoded(byte[])
```

Verifies if the String/byte array is a valid Base64 value.

## HEX

Converts to and from HEX encoded Strings.

```
byte[] be.atbash.util.codec.Hex.encodeToString(byte[])
```

Converts a byte array to a HEX encoded String.

```
byte[] be.atbash.util.codec.Hex.decode(String)
```

Converts the Hex representation to a byte array. Throws an *IllegalArgumentException* when the value isn't a HEX encoded value.

# ByteSource

This is an interface which provides a source of bytes which can be used to create a BASE64 or HEX representation from it.

The SE Utils project provide an implementation with support for File and InputStream next to the obvious String and char array.

A ByteSource can be created by the following snippet

```
ByteSource.creator.bytes(data);
```

Where *data* is an instance of one of the supported types as explained above.

When you need you need to support some additional types, which also can be converted to a byte array, you can use the SPI for this purpose.

Start by implementing the interface `be.atbash.util.codec.ByteSourceCreator` and define it as a service by specifying the fully qualified class name in the file `/META-INF/services/be.atbash.util.codec.ByteSourceCreator`.

That way, your creator is used by the statement `ByteSource.creator.bytes()` and thus your logic is executed.

Your custom creator can use instances of `DefaultByteSource` or an implementation of the interface `ByteSource`.

## Instantiations

The dynamic instantiation of classes is important when you define the class name within configuration values.

With the **ClassUtils** utility you can verify if the class name effectively exists and instantiate it with some arguments.

The classes, but also the resources, are searched in the following order

1. context classloader attached to the current thread
2. classloader who has loaded the ClassUtils class
3. system class loader

```
be.atbash.util.reflection.ClassUtils#isAvailable(String)
```

Verifies if the class defined by its FQCN (a fully qualified class name which is package name and class name) is found by one of the 3 class loaders.

```
be.atbash.util.reflection.ClassUtils#newInstance(String)
be.atbash.util.reflection.ClassUtils#newInstance(Class)
```

Creates an instance of the class (specified by the FQCN or the class instance) using the no-args constructor. When such a constructor is not available or there was an *Exception* thrown during the instantiation of the class, an **be.atbash.util.reflection.InstantiationException** is thrown.

```
be.atbash.util.reflection.ClassUtils#newInstance(String, Object...)
be.atbash.util.reflection.ClassUtils#newInstance(Class, Object...)
```

Creates an instance of the class (specified by the FQCN or the class instance) using a Constructor which matches the arguments.

The *Constructor* which will be used to instantiate the class is not determined by the *Class.getConstructor(argTypes)* method as it doesn't work when one of the arguments is *null*. The following algorithm is used to find the *Constructor*.

1. Loop over all *Constructors*
2. Consider a *Constructor* when it has the same number of arguments
3. Check if the argument types have the same class (using *equals*) as the parameter type. When the argument is *null*, it is considered as a match.
4. When no *Constructor* is found, all *Constructors* with the correct number of arguments is verified again but now a less strict match is used (using *isAssignableFrom* to allow subtypes)
5. When there is not exactly 1 *Constructor* found, an **be.atbash.util.reflection.NoConstructorFoundException** is thrown.

When an *Exception* is thrown during the instantiation of the class, an **be.atbash.util.reflection.InstantiationException** is thrown.

```
be.atbash.util.reflection.ClassUtils#getResourceAsStream(String)
```

Returns the resource using the 3 class loaders as described above.

## Reading version

With the class **be.atbash.util.version.VersionReader**, you can read the version information stored within the *META-INF/MANIFEST.MF* file.

Define the version information by configuring the *maven-jar-plugin* or *maven-war-plugin* in the maven build section.

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.5</version>
  <executions>
    <execution>
      <id>manifest</id>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <archive>
      <manifestEntries>
        <Release-Version>${project.parent.version}</Release-Version>
        <buildTime>${maven.build.timestamp}</buildTime>
      </manifestEntries>
    </archive>
  </configuration>
</plugin>

```

This information can be read by using the following snippet

```

VersionReader versionReader = new versionReader("atbash-config");
versionReader.getReleaseVersion();
versionReader.getBuildTime();

```

The constructor argument is the artifact from which we want to read this information (actually it is the first part of the name of the jar file but these are in most cases the same).

## Base exceptions

There are 3 exception classes defined which can be handy in all applications.

- `be.atbash.util.exception.AtbashException`

This is a *RuntimeException* used as a parent class for all Atbash defined exceptions. It makes it possible to define a generic Exception handler (within JSF or JAX-RS) to handle all the Exceptions uniformly (logging, showing info to end user, ...)

- `be.atbash.util.exception.AtbashIllegalActionException`

This exception is thrown when the Atbash code detects a wrong usage of the framework by the developer. An example is a usage of a non-existing URL filter name in the Octopus framework (maybe a typo).

It is recommended that the error message starts with a code (like *(OCT\_DEV\_001)* ) and the

documentation describes then the situation and what actually is done wrong and how it can be fixed.

- `be.atbash.util.exception.AtbashUnexpectedException`

Can be used to convert a checked exception (like an `IOException`) into an *AtbashException* so that it can be handled by the general exception handler. Most checked exceptions never occur during the execution of the application, but they need to be caught or thrown.

## String utils

```
be.atbash.util.StringUtils.hasText(String)
be.atbash.util.StringUtils.isEmpty(String)
```

Verifies if the String contains something meaning full (something different then whitespace) or not.

When the argument is *null*, empty String ("" ) or contains only whitespace ( " ") it is considered as empty.

```
be.atbash.util.StringUtils.hasLength(String)
```

Verifies if the String contains characters or not but handles null as the empty String. Whitespace characters are counted as a real character.

```
be.atbash.util.StringUtils.clean(String)
```

Cleans the argument, this are the rules

Argument	Result
<code>null</code>	<code>null</code>
<code>""</code>	<code>null</code>
<i>other cases</i>	<code>.trim()</code>

```
be.atbash.util.StringUtils.startsWithIgnoreCase(String, String)
```

Verifies if the String starts with a certain prefix, case insensitive. Method handles correctly the situation where one or both arguments are *null*.

```
be.atbash.util.StringUtils.split(String)
```

Break down the String within items, delimited by , by default (there exist an overloaded method to define also the delimiter. You can use " to define the start and end of an item. The following example has thus only 2 items



```
key , "value1,value2"
```

The quotes are removed and the item is trimmed before the placed in the return array.

```
be.atbash.util.StringUtils.toDelimitedString(Collection, String)
be.atbash.util.StringUtils.toDelimitedString(Object[], String)
```

Converts the collections or array of Objects to a String where each item is separated by the 2nd parameter.

## Collection utils

```
be.atbash.util.CollectionUtils.asSet(E...)
be.atbash.util.CollectionUtils.asList(E...)
```

Returns the items specified in the argument as *Set* or *List* respectively.

```
be.atbash.util.CollectionUtils.isEmpty(Collection)
be.atbash.util.CollectionUtils.isEmpty(Map)
```

Verifies if the argument is null or contains no elements.

```
be.atbash.util.CollectionUtils.size(Collection)
be.atbash.util.CollectionUtils.size(Map)
```

Returns the size of the *Collection* or *Map* but handles null argument correctly.

## Proxy Utils

A few methods related to proxied classes when they are generated by (CDI) libraries.

```
be.atbash.util.ProxyUtils.isProxiedClass(Class)
```

Test if the class is a proxy class based on the name. Because proxied classes have a specific suffix.

```
be.atbash.util.ProxyUtils.getUnproxiedClass(Class)
```

Returns the 'real' class for the proxied class by returning the super class of the parameter. When the parameter isn't a proxied class, it return the parameter itself.

```
be.atbash.util.ProxyUtils.getClassName(Class)
```

Returns the 'real' class name for the parameter. When it is a proxied class, it return the name of the super class, otherwise it returns the name of the class itself.

## CDICheck

Probably only useable in advanced use cases where you create a library which must be able to run within plain Java SE and within a CDI container.

```
be.atbash.util.reflection.CDICheck.withinContainer
```

This methods return true or false depending on the context and library can select code path accordingly (like retrieving beans through CDI or ServiceLoader)

## TestReflectionUtil

Utility class for unit tests to help with injection and setting values of instances used during the test.

Add the following artifact to your maven project file

```
<dependency>
  <groupId>be.atbash.utils</groupId>
  <artifactId>utils-se</artifactId>
  <version>0.9.2</version>
  <classifier>tests</classifier>
  <scope>test</scope>
</dependency>
```

When you are using a instance of a class during your unit test, and that class should have some dependencies (which are normally set by some kind of injection), the *injectDependencies* can be very useful in those situations.

```
public class Foo {
```

```
  private Bar bar;
```

```
}
```

Then within a test you can have the following code;

```
Foo foo = new Foo();
TestReflectionUtils.injectDependencies(foo, new Bar());
```

The injection is done based on the compatible type assignments. So you can also inject a subclass of Foo in the same manner (thus also a Mock created by Mockito for instance)

However you should always consider the default supported functionality from Mockito for example.

```
@RunWith(MockitoJUnitRunner.class)
public class FooTest {

    @Mock
    private Bar barMock;

    @InjectMocks
    private Foo foo;

}
```

Other useful methods in the class *TestReflectionUtils*

- `setFieldValue()` sets the value of a specific property in an instance (when `injectDependencies` could inject it into multiple properties because they have assignable types)
- `getValueOf()` return the value of property by name (when there is no getter for instance)
- `resetOf()` sets the property with a null value.