RFC 215 - Object Conversion

Draft

44 Pages

Abstract

Java is a type safe language that can be used to create applications that are easy to navigate in an IDE and that significantly reduce time to write tests. However, there is a tendency in Java to bypass the type system because it is often deemed easier to use strings instead of proper types: logging, JAX-RS, configuration, records, etc. This RFP investigates the issues that surrounding the use of type safe interfaces and DTOs where traditionally properties and other string based solutions are used.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0 Document Information **2**
- 0.1 License 2
- 0.2 Trademarks 3
- 0.3 Feedback 3
- 0.4 Table of Contents 3
- 0.5 Terminology and Document Conventions 4
- 0.6 Revision History 4

1 Introduction **5**

2 Application Domain **5**
- 2.1.1 Commons-Convert 6
- 2.1.2 Google Guava 6
- 2.1.3 Dozer 6
- 2.2 Conversions 6
- 2.3 Reflection 7
- 2.4 Terminology + Abbreviations 7

3 Problem Description **7**

4 Requirements **8**
- 4.1 General 8
  - 4.1.1 From Configurer RFP 8
- 4.2 Maps 8
- 4.3 DTOs 9
- 4.4 JSON 9

5 Technical Solution **9**
5.1 Converter Service 10
5.2 Codec Service 11
5.3 Use outside of OSGi 12
5.3.1 Converter service via ServiceLoader 12
5.3.2 Codec service via ServiceLoader 12
5.4 Conversions 12
5.4.1 Scalars and other singular types 13
5.4.2 Arrays and Collections 15
5.4.3 Maps and related data structures 15
5.4.4 Adding support to existing OSGi types 19
5.4.5 Provided Rules 19
5.4.6 Special Cases 19
5.5 TypeReference base class 20
5.6 Variable substitution 21
5.7 Portable Encodings 21
5.7.1 Portable JSON encoding 21
5.7.2 Portable YAML encoding 22
5.7.3 Portable XML encoding 22

6 Data Transfer Objects **23**

7 Javadoc **23**

8 Considered Alternatives **41**

9 Security Considerations **42**

10 Document Support **43**
10.1 References 43
10.2 Author's Address 43
10.3 Acronyms and Abbreviations 43
10.4 End of Document 43

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 10.1.

`Source code is shown in this typeface.`

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	10/01/15	Initial version, from RFP, with some initial API proposals.
0.1	January, 2016	David Bosschaert, changes from Chicago F2F feedback.
0.2	January, 2016	David Bosschaert, changes from Madrid F2F.

Revision	Date	Comments
0.3	February, 2016	David Bosschaert, changes from EEG concall Jan 27.
0.4	2016-02-01	Updated coercion table per DS errata. BJ Hargrave
0.5	March, 2016	David Bosschaert, changes following EG call feedback.
0.6	March, 2016	David Bosschaert, additional changes following EEG call feedback.
0.7	April, 2016	David Bosschaert, added javadoc.
0.8	April, 2016	David Bosschaert, added a small number of suggestions in the API.
0.9	May, 2016	David Bosschaert, updates following Chicago F2F. Support for JavaBeans

1 Introduction

This RFC originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that needs to be solved.

Java is a type safe language that be used to create applications that are easy to navigate in an IDE and that significantly reduce time to write tests. However, there is a tendency in Java to bypass the type system because it is often deemed easier to use strings instead of proper types: logging, JAX-RS, configuration, records, etc. This RFP investigates the issues that surrounding the use of type safe interfaces and DTOs where traditionally properties and other strings are used.

2 Application Domain

Today, many programs directly interface directly with the outside world through REST, HTTP, and other protocols that are frequently require conversion from strings or byte streams. Data conversion is an inherent part of writing software in a type safe language. In Java, converting strings to proper types or to convert one type to a more convenient type is often done manually. Any errors are then handled inline.

A common problem is interacting with Javascript. Since Javascript has no user defined types, it can get away with relatively clean looking code; the same code in Java usually requires significantly more code due to the required

conversions to the fine grained types of Java. This code bypasses the built-in facilities like fields and methods and instead defines constant key strings and embeds the knowledge of the types in a piece of code instead of relying on a central declaration that is then verified by the compiler.

In release 6, the OSGi Alliance introduced Data Transfer Objects (*DTOs*). DTOs are public objects without generics that only contain public fields based on simple types, arrays, and collections. In many ways DTOs replace Java beans. Java beans are hiding their fields and provide access methods but that separated the contract (the public interface) from the internal usage. Though this model has advantages in technical applications (many types, few fields) it tend to be a large overhead when there are relatively few types with lots of fields. i.e. the more common web applications. DTOs unify the specification with the data since the data is what is already public when it is sent to another process or serialized.

By limiting the allowed data types in DTOs and ensuring they have no cycles they can be easily (de)serialized using JSON, providing easy interactions with Javascript.

In enRoute, a *DTO+* is a DTO but it additionally allows many additional types and defines the rules for creating these types in a conversion.

In applications, a DTO provides the same role as a Javascript hash/object. In java, however, the fields are typed, providing type checks and content assist in the browser. However, there are similar needs to what Javascript provides when objects are used that way:

- Deep copy – Create two DTOs that are equal but do not share any instances
- Deep equals – Compare two DTOs for equality
- Shallow copy – Create a new DTO but share the fields
- Diff – Calculate the difference between 2 DTOs, providing where and why the the objects are different.
- Path based access – Provide a path based access into a DTO. E.g. a path can be `foo.4.abc`

Several libraries exist which provide a similar form of type conversion.

2.1.1 Commons-Convert

“Commons-Convert is a library dedicated to the task of converting an object of one type to another. The library is ideal for scripting languages or any application that needs to convert (or coerce) one Java object type to another.” (http://commons.apache.org/sandbox/commons-convert)

2.1.2 Google Guava

Google Guava (https://github.com/google/guava) Converter API (http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/base/Converter.html)

2.1.3 Dozer

“Dozer is a Java Bean to Java Bean mapper that recursively copies data from one object to another. Typically, these Java Beans will be of different complex types.” (http://dozer.sourceforge.net)

2.2 Conversions

The Java language has the concept of a *type*. A type is defined by either a class or one of the generic types.

The first level of conversion of a *value* is limited to *simple types*. Simple types are either booleans, characters, numbers, and String. Simple types do not use generics and have no cardinality, they are immutable. For example, converting a `String` to an `int`. In general the developer that is writing the conversion expects a specific type as input and then calls an appropriate method to do the actual conversion: `int integer = Integer.valueOf(string)`.

The second level are *cardinal* types:

- *Collection* – An enumeration of zero or more values.
- *Arrays* – An enumeration of zero or more values.
- *Map* – A mapping from one value to another value

2.3 Reflection

Java is a type safe language that provides access to the type information during runtime. This information is quite extensive and includes generic type information. Though it is impossible to know the type parameters are for an object from a generic class, the places where a generic type is used (a call, extending, method arguments, return type) actually do contain the full generic signatures.

Java does not have a built in concept to create a *type reference* for generic types since an instance does not contain the generic information it was compiled with, this information is erased. A common pattern to provide a generic type signature is to create a `TypeReference<T>` class. To create a reference, an inner subclass is created that then encodes the T in its generic signature for the super relation:

```
new TypeReference<List<Map<Pair<Integer,String>,String>>() {}
```

The TypeReference class then can inspect this information and provides the desired type information with a `Type getType()` method.

Libraries like the bnd Converter can use the reflective information to create another object of a desired type. For example:

```
byte[] barray = Converter.cnv( byte[].class, "1"); // new byte[]{1}
List<Short> shorts = Converter.cnv(
      new TypeReference<List<Short>>(){}, new String[]{"1","2"}); // [1, 2]
FooConfig fooConfig = Converter.cnv( FooConfig.class, map );
Map<String,Object> map = Converter.cnv(
      new TypeReference<Map<String,Object>>(){}, dto );
```

The bnd JSON Codec extend this model to *JSON*. JSON is a syntax to transfer data with only a limited set of types: string, booleans, numbers, arrays, and maps. In general, it is straightforward to map a DTO+ to JSON stream since Java has so much more type information than is required. However, the bnd JSON Codec can take an input stream and a DTO+ and map the JSON input stream to the fields and types defined in the DTO, recursively. Since these types contain the full generic information it is possible to support quite rich DTO+ objects.

2.4 Terminology + Abbreviations

- DTO+ – a DTO with an identity.

3 Problem Description

Experience shows clearly that leveraging the Java type system more and reducing the use of key constants and DSLs in the code can increase the productivity of developers significantly. Java is an excellent language to act as a specification language, which the huge benefit that it can be executed and is extensively supported by IDEs like Eclipse and Intellij.

The DTO model is already powerful in replacing where properties were used but requires more extensive support to match capabilities in Javascript, but then in a type safe way.

However, moving to a more type safe use of Java requires a powerful and flexible data handling that currently lacks. This RFP therefore is seeking proposals for a service that provides the following services:

- General any-to-any type conversion
- Extension to the DTO model that allows more types to be used in its fields
- Extension to the DTO that provides DTOs with an identity and if applicable comparable.
- DTO support for copying, equals, and diffing
- JSON encoding/decoding

4 Requirements

4.1 General

- G0010 – Provide a service that can convert any object to a given type. The specification must clearly outline what conversions are possible but must at least allow the simple types, maps, collections, and arrays.
- G0020 – Provide a type reference class
- G0030 – It must be possible to specify the destination type with a class, a generic type (Type<T>), or a type reference.
- G0040 – It must be possible to convert Strings to popular Java types like Pattern, File, Date, Java Date/Time, UUID, et al. The specification must clearly define the rules for these classes.
- G0045 – It must be possible to convert EventAdmin Event objects and Service Reference objects to Map<String,Object>
- G0050 – The solution should be usable outside of an OSGi Framework, i.e. in plain Java environment.

4.1.1 From Configurer RFP

- G0250 – It must be possible to use [substitute] information from the local system, for example passwords, if permitted.

4.2 Maps

- M0010 – It must be possible to convert a Map or Dictionary to an interface where the method names are used as keys
- M0020 – It must be possible to convert a DTO+ to a Map<String,Object> and vice versa

4.3 DTOs

- D0005 – It must be possible to assign an identity to a DTO. This shall be referred to as a DTO+.
- D0010 – It must be possible to diff two objects of the same type returning information where the DTO+'s differ and in what way.
- D0020 – Provide a proper deepEquals that assumes DTO+
- D0030 – Provide a way for types to handle conversion from and to strings for non-specified types
- D0040 – Provide a way to set/get fields from a DTO+ through a string path.
- D0050 – Provide a base class for identity DTO+s
- D0060 – Provide a compare function for identity DTOs that have a primary key that is comparable
- D0070 – Provide a way to find out if a DTO+ is complex
- D0080 – Provide a way to find out an object is DTO+
- D0090 – Provide a way to verify that an object is a DTO+ and has no cycles
- D0100 – Provide a deep copy routine for a DTO+
- D0110 – Provide a shallow copy routine for a DTO+

4.4 JSON

- J0010 – Provide a JSON encoder and decoder that uses the conversion rules for the conversion from JSON types to destination types
- J0020 – JSON decoding must be able to provide a value without specifying any type for the destination
- J0030 – The output must be an OutputStream, Appendable, or String
- J0040 – The input must be an InputStream, Readable, or String
- J0050 – It must be possible to pretty print the output
- J0055 – It must be possible to generate canonical, compact output
- J0060 – It must be possible to specify the output character set for a stream
- J0070 – It must be possible to specify if nulls are outputed or not
- J0080 – It must be possible to add hook to the conversions for custom types for encoding and decoding

5 Technical Solution

The solution centers around services to support the conversions: the `Converter` service which can convert objects from one type to another, and the `Codec` service which can encode/decode a specific serialized format.

This RFC also defines a mechanism to use these services outside an OSGi Framework.

5.1 Converter Service

The `Converter` service is used to start a conversion. The service will be obtained from the service registry. The conversion is then completed via the `Converting` interface that has methods to specify the target type.

```
public interface Converter {

    Adapter getAdapter();

    Converting convert(Object obj);
}

public interface Converting {
    Converting defaultValue(Object defVal); // value to use in case of failure or null

    <T> T to(Class<T> clazz);

    <T> T to(TypeReference<T> ref); // to capture the generics of type T

    Object to(Type type); // for use with reflection

    @Override
    String toString(); // same as to(String.class)?
}

public interface Adapter extends Converter {
    // Specify a rule where the adapter deviates from the standard converter.
    // Multiple rules can be specified.
    <F, T> Adapter rule(Rule<F, T> rule);
    <F, T> Adapter rule(Class<F> fromCls, Class<T> toCls,
                  Function<F, T> toFun, Function<T, F> fromFun);
}

Example uses:

    Converter c = ...; // from service registry
    String s = c.convert(12L).to(String.class);
    Long l = c.convert("123").to(Long.class);

    // This codec adapter changes the way Date is converted to and from String[] is handled
    Adapter ca = c.getAdapter();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ");
```

```
    ca.rule(Date.class, String.class,
      sdf::format,
      v -> { try { return sdf.parse(v); } catch (ParseException pe) { return null; } });
    String s = ca.convert(new Date()).to(String.class); // "2016-05-12T10:50:31.439+0100"
    Date d = ca.convert(s).to(Date.class); // back to Date

    ca.rule(String[].class, String.class,
        v -> Stream.of(v).collect(Collectors.joining(",")),
        v -> v.split(","));   String s = ca.convert(new String[] {"A","B"}).to(String.class);
// "A,B"
    String[] sa = ca.convert(s).to(String[].class); // {"A", "B"}
```

The TypeReference class mentioned here is used to obtain Java generics information at runtime. It is defined later in this document.

TODO Consider use of stream-based approach to generate resulting objects (e.g. create an Event using a Lambda).

5.2 Codec Service

The Codec service can be used to encode a given object in a certain representation, for example JSON, YAML or XML. The Codec service can also decode the representation it produced. A single Codec service can encode/decode only a single format. To support multiple encoding formats register multiple services.

```
public interface Codec {
    <T> Decoding<T> decode(Class<T> cls);
    <T> Decoding<T> decode(TypeReference<T> ref);
    Decoding<?> decode(Type type);

    Encoding encode(Object obj);

    // Use an alternative converter with this codec, to convert custom objects
    Codec with(Converter converter);
}
```

Codec service properties:

property name	*type*	*description*
osgi.codec.mimetype	String	The mime type for the encoding this codec can handle. For a list of portable types see later in this document.

```
public interface Decoding<T> {
    T from(InputStream in); // uses UTF-8, if applicable
    T from(InputStream in, Charset charset);
    T from(Readable in);
    T from(CharSequence in);
}

public interface Encoding {
    Encoding pretty();

    void to(OutputStream out); throws IOException // uses UTF-8, if applicable
    void to(OutputStream out, Charset charset); throws IOException
```

```
    Appendable to(Appendable out);
    String toString();
}
```

Note that the APIs that work on streams do not close the stream after their operation. A stream can easily be closed via the try-with-resources mechanism.

The above API can be used it like this:

```
String encoded = codec.encode(myObj).pretty().toString();
```

5.3 Use outside of OSGi

Services defined in this specification can also be obtained via the `java.util.ServiceLoader` API for use-cases where an OSGi Service Registry is not available. Such usage is defined in this section.

Note this section only applies to use outside of OSGi. When running in an OSGi framework these services are available from the Service Registry.

5.3.1 Converter service via ServiceLoader

A instance of the Converter service can be looked up via the `ServiceLoader.load(Converter.class)` method.

5.3.2 Codec service via ServiceLoader

As a number of different `Codec` services can be available at run-time, each for a specific target type, the `Codec` service is obtained via the `CodecFactory` service when used with the `java.util.ServiceLoader` to select the encoding to be used:

```
public interface CodecFactory {
    Codec getCodecByMimeType(String type); // e.g. “application/json”
}
```

With the CodecFactory obtain the desired codec as in the following example:

```
Codec getCodec(String mimeType) {
    ServiceLoader<CodecFactory> sl = ServiceLoader.load(CodecFactory.class);
    for (CodecFactory cf : sl) {
        Codec codec = cf.getCodecByMimeType(mimeType);
        if (codec != null) {
            return codec;
        }
    }
    return null; // Requested codec not found
}
```

5.4 Conversions

~~TODO describe what happens when a conversion cannot be made, e.g. convert “Hello” into an int or what happens if I try to convert a map {“numberOfPages”, TRUE} into a java.awt.print.Pageable? Will we use null? No value (if possible) or throw an exception?~~

The following conversions will be supported.

This section describes conversions to String and from String for the default converter. Users can provide alternative representations by configuring their own `Adapter` that delegates to the default converter for all except the special conversions.

5.4.1 Scalars and other singular types

If a runtime type is the same as the target type, or a subtype of it, no conversion is needed and hence this is not mentioned in this table.

The following table is based on the table from the Declarative Service Specification *Coercion from Property Value to Method Type* 112.10 and aims to be backward compatible with that specification.

↓ dest / src →	String	Boolean	Character	*Number*	null	empty Collection/Array
String	v	v.toString()	v.toString()	v.toString()	null	null
boolean	Boolean. parseBoolean(v)	v.booleanValue()	v.charValue() != 0	v.doubleValue() != 0	FALSE	FALSE
char	v.lenght() > 0 v.charAt(0) : 0	v.booleanValue() ? 1 : 0	v.charValue()	(char) v.intValue()	0	0
number	*Number*. parse*Number*(v)	v.booleanValue() ? 1 : 0	(number) v.charValue()	v.*number*Value()	0	0
Class	Bundle.loadClass(v)	throw	throw	throw	null	null
EnumType	*EnumType*. valueOf(v)	*EnumType*. valueOf(v.toString(). toUpperCase())	*EnumType*. valueOf(Character. toString(v))	*EnumType*.values() [v.intValue()]	null	null
AnnotationType	throw	throw	throw	throw	null	null

5.4.1.1 Other source data types

Source data types not listed in the above table are converted to String using the `toString()` method before further conversion to a target type is attempted.

5.4.1.2 Other target data types, including boxed Number, Boolean and Character

TODO Conversions between Boolean → Number/Character and Number/Character → Boolean

Conversion to other target data types is done by converting the source to a String value first. Then further conversion to the target type is be attempted by trying the following methods on the target type, in this order:

1. `static valueOf(String s)`
2. String constructor.

Exceptions:

- `null` values will result in a `null` value for the target value.
- Empty arrays / collections will be converted into a `null` target value.

5.4.1.3 *Conversion failures*

If a conversion operation throws an exception, such as a NumberFormatException, or can otherwise not be performed, the converter will throw a ConversionException, which is a RuntimeException. However, if a default value has been provided for the conversion, via the Converting.defaultValue() method this value will be returned instead in the case of failure.

5.4.2 Arrays and Collections

5.4.2.1 Conversion from Arrays, Collections to single-value type

The first element is taken and converted into the target element.

Conversion from empty Arrays and Collections is described in a previous section.

Implementations wishing that need to handle multi-value to single-value differently, for example converting a String array to a comma-separated String, can do so by providing their own adapter for the conversion.

5.4.2.2 Conversion to Array or Collection

A new object is always returned as the object will be owned by the caller.

Even if the target type is the same as the source type, the source object cannot be passed through and still needs to run through the conversion process as described here. This because the actual contents may need to be converted to comply with the generic signature of the target type.

Result object creation:

target type	
Collection interface	A mutable implementation is created. E.g. if the target type is `List` then the implementation can create an `ArrayList`. When converting to a Set the converter must choose a set implementation that preserves iteration order.
Collection concrete type	A new instance is created by calling `Class.newInstance()` on the provided type. For example if the target type is `LinkedList` then the converter creates a target object by calling `LinkedList.class.newInstance()`.
`T[]`	`Array.newInstance(Class<T> cls, int x)` where *x* is the size of the source collection, 1 in case of a scalar.

If the source object is `null`, an empty collection/array is produced.

If the source object is a single-value object then this value is the element to be inserted.

If the source is a collection or array then every element on this list is considered an element to be inserted.

If the source is a map-like structure then each entry on the map will be converted into a Map.Entry instance which in turn will be considered as an element to be inserted. See the Map.Entry conversions for more details.

Once the elements to be inserted are established, each element is converted into the target type using the converter rules before it is inserted.

The converter should use all information available to it to perform the conversion. I.e. when a `TypeReference` is used the generics information of the target type is still available at runtime making it possible to instruct the converter to convert to parameterized types of which the generic information would otherwise be erased. For example the following construct can be used to convert an `int[]` into a `Set<Double>`:

```
converter.convert(new int [] {1,2,3}).to(new TypeReference<Set<Double>>() {})
```

5.4.3 Maps and related data structures

These data structures can hold multiple key-value pairs of various types. The canonical representation of such data structure is a Map.

5.4.3.1 Map

A new map instance is always returned as the resulting map is owned by the caller of the converter.

While Map is the canonical type further conversion is generally needed when the type parameters for the target type are known. Additionally, a specific map implementation may be requested.

target type	
Map interface	A mutable implementation is created. E.g. if the target type is `ConcurrentNavigableMap` then the implementation can create a `ConcurrentSkipListMap`.
Map concrete type	A new instance is created by calling `Class.newInstance()` on the provided type. For example if the target type is `HashMap` then the converter creates a target object by calling `HashMap.class.newInstance()`.

Each key-value pair in the source map is converted to the parameterized values of the target map using the converter rules and then added to the target map, similar to the process used for conversion to Collections.

5.4.3.2 Dictionary

Converting to a Dictionary is done by converting to a Map first, and then converting this Map to a Dictionary in a process similar to that for a Map.

Converting from a Dictionary is done by converting the Dictionary to the canonical type, Map, first and then converting this Map to the target type.

5.4.3.3 Interface

When converting into an interface the converter will create a dynamic proxy to implement this interface. The name of the method returning the value should match the key of the map entry, taking into account the conversion rules specified in table 112.9 of the R6 Declarative Service specification. The key of the map may need to be converted into a String first.

In this case support conversion to interfaces that can provide defaults for non-set values:

```
interface Config {
  int my_value(); // no default, used when converting from the interface
  int my_value(int defVal);
  int my_value(String defVal); // default value is automatically converted to the target type

  boolean my_other_value();
}
```

Default values are used when the key is not present in the map for the method. If a key is present with a `null` value, then `null` is taken as the value and converted to the target type.

```
Map<String, ?> myMap = … // an example map
Config cfg = converter.convert(myMap).to(Config.class);
int val = cfg.my_value(17); // if not set then use 17
```

In this version of the specification JavaBeans-style interfaces are not yet supported.

Converting from an interface is done by calling each no-args method on the interface and storing the resulting value, after converting it into the target type, in a map. The method name is used as the key and will be converted as described in table 112.9 of the DS spec. The user of the conversion service should ensure that the method invocations have no unwanted side effects and are idempotent.

5.4.3.4 Annotation

Just like interface but with the added capability of specifying a default in the annotation definition.

The following example are based on
https://github.com/bndtools/bnd/issues/796 and https://github.com/bndtools/bnd/issues/981

```
@interface Config {
  String[] args() default {"arg1", "arg2"};
}

// the following will set a={"args1", "arg2"}
String[] a = converter.convert(new HashMap()).to(Config.class).args();

// this will set a1={}
Map m1 = Collections.singletonMap("args", null)
String[] a1 = converter.convert(m1).to(Config.class).args();

// this will set a2={""}
Map m2 = Collections.singletonMap("args", "")
String[] a2 = converter.convert(m2).to(Config.class).args();

// this will set a3={","}
Map m3 = Collections.singletonMap("args", ",")
String[] a3 = converter.convert(m3).to(Config.class).args();

// this will set a4={"",""}, non-default conversion via adapter
Map m4 = Collections.singletonMap("args", ",")
Adapter ca = c.getAdapter();
ca.rule(String[].class, String.class,
  v -> Stream.of(v).collect(Collectors.joining(","))),
  v -> v.split(","));
String[] a4 = converter.convert(m4).to(Config.class).args();
```

5.4.3.5 Java Beans

Converting to and from Java Beans is supported. When converting from a JavaBean introspection is used to find the read accessors. A read accessor must have no arguments and a non-void return value. The method name must start with `get` followed by a capitalized property name, e.g. `getName()`. For boolean/Boolean properties a prefix of `is` is also allowed.

When converting to a JavaBean, its public no-arg constructor is used to instantiate the object and write accessors are found by introspecting the bean to find set methods that take a single argument.

Java Beans are only accessed via public methods. All methods of the bean class itself and its superclasses are considered. However accessor methods inherited from `java.lang.Object` are ignored; i.e. the `getClass()` method inherited from `Object` is not considered an accessor.

Not supported: access via indexed bean properties is not supported as there is no way to find out the size of the array via these accessors.

5.4.3.6 DTO

DTOs are classes with public non-static fields and no methods other than the ones provided by the java.lang.Object class. OSGi DTOs extend the org.osgi.dto.DTO class but the converter should ignore this. This is

to keep the converter API itself clean from OSGi dependencies. DTOs may have static final fields, these can also be ignored by the converter.

When converting to a DTO, the converter attempts to find fields that match the key of each entry in the map and then converting the value to the field type before assigning it. They key of the map entries may need to be converted into a String first. Keys are mapped according to table 112.9 of the R6 Declarative Service specification.

When converting from a DTO the value of each public non-static field is put in the target map, taking the field name as its String key.

5.4.3.7 From other types

The converter should use reflection to find a `Map getProperties()` or `Dictionary getProperties()` method on the source type to obtain a map representing this object which can then be converted into another target type.

Arrays or collections can be converted into a map structure by converting each element to a Map.Entry instance first and then use these to populate the map. The resulting map should have preserve iteration order to be the same as the original array/collection if it has a well-defined iteration order.

Other objects that do not implement an interface cannot be converted into a map-like structure.

5.4.3.8 To other types

Convert `map.entrySet()` to an order-preserving collection of Map.Entry objects and then convert this collection to the target type.

5.4.3.9 Conversion failures

If a conversion cannot be performed, a ConversionException is thrown. In the case of a conversion to a immediately constructed object, such as a map, this exception is thrown immediately. When a conversion is made to a type represented by a proxy the exception is thrown when the converted value is requested.

For example, given the following Map:

```
Map m = new HashMap();
m.put("key1", "123");
m.put("key2", "test");
m.put("key4", "ignored");
```

The following conversion will immediately throw a `ConversionException` as `key2` cannot be converted into an Integer.

```
converter.convert(m).to(new TypeReference<Dictionary<String>,<Integer>>(){});
```

The following conversion will succeed, but a `ConversionException` will be thrown when `getKey2()` is called:

```
interface TestInterface {
  int getKey1();
  int getKey2();
  int getKey3();
}
TestInterface ti = converter.convert(m).to(TestInterface.class);
int i1 = getKey1(); // succeeds
int i2 = getKey2(); // ConversionException thrown here
int i3 = getKey3(); // returns 0 as key3 was not set in the map
```

key4, which was present in the original map is ignored as the target interface does not have a getKey4() method.

When converting to a JavaBean, the entire bean is populated immediately, so any properties present in the map that cannot be converted to the target type in the bean will cause a ConversionException.

5.4.4 Adding support to existing OSGi types

Add Map<String,?> getProperties() to various OSGi APIs. This to facilitate converting from those types to other types. It will also keep the converter API clean of other OSGi deps.

List of types:

- `org.osgi.framework.BundleContext`
- `org.osgi.framework.ServiceReference`

The getProperties() method returns a read-only map that is 'live' wrt to the backing structure. Changes to the backing structure will automatically be reflected in the map.

The new method will not be added via a separate interface, the Converter should use reflection to find this method when it needs to convert an object. However, where possible existing interfaces should be updated to include the getProperties() method.

5.4.5 Provided Rules

The implementation can provide a number of Rule implementations out of the box to support conversions that are more specific but nevertheless common, such as converting from an org.osgi.service.event.Event into a Map.

5.4.6 Special Cases

This section specifies special cases also supported by the converter.

5.4.6.1 UUID

UUIDs are created from String representations by calling `UUID.fromString()`

5.4.6.2 Pattern

Pattern instances are created by calling `Pattern.compile(String s)`.

5.4.6.3 Java 8 Date/Time API

Converting to Java 8 Date/Time classes from strings is done using the static parse(CharSequence cs) method which is available in most implementations of the Temporal interface, for example:

```
LocalDateTime ldt = LocalDateTime.parse(s);
LocalDate ld = LocalDate.parse(s);
LocalTime lt = LocalTime.parse(s);
OffsetTime ot = OffsetTime.parse(s);
ZonedDateTime zdt = ZonedDateTime.parse(s);
```

5.4.6.4 Date and Calendar

A `java.util.Date` instance is converted to a Long value by calling `Date.getTime()`.

Converting a Long into a `java.util.Date` is done by calling new `Date(long)`.

Converting a Date to a String is done by

@@@ TODO UTC

Converting into other types is by converting to a String as intermediary.

Conversions from Calendar objects are done by converting the Calendar to a Date via getTime() first. Convertions to a Calendar object are done by constructing a Date object with the desired time (UTC) and then setting the time in the Calendar object via setTime().

5.4.6.5 Map.Entry

Conversion of Map.Entry to a target type is done by evaluating the compatibility of the target type with both the key and the value in the entry and then using the best match.

If the target type matches the key type, the key type is used. If the target matches the value type then the value type is used.

Conversion of Map.Entry to String will be done by converting both key and value to String and concatenating both with an '=' sign.

Conversion of a Map.Entry to any other type is done by converting the value tp String and then converting the result to the target type.

Converting to a Map.Entry can only be done from Strings where the lhs of the '=' sign is assigned to the key and the rhs of the '=' sign is the value. Strings without an '=' sign result in a null Map.Entry value.

5.5 TypeReference base class

The TypeReference is provided as part of the API as follows. This variant was based on the OSGi Enroute Project taken from:

https://github.com/osgi/osgi.enroute/blob/master/osgi.enroute.base.api/src/osgi/enroute/dto/api/TypeReference.java

```
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

/**
 * An object does not carry any runtime information about its generic type.
 * However sometimes it is necessary to specify a generic type, that is the
 * purpose of this class. It allows you to specify an generic type by defining a
 * type T, then subclassing it. The subclass will have a reference to the super
 * class that contains this generic information. Through reflection, we pick
 * this reference up and return it with the getType() call.
 *
 * <pre>
 *  List<String> result =
 *      converter.convert(Arrays.asList(1,2,3)).
 *          to(new TypeReference<List<String>>() {});
 * </pre>
 *
 * @param <T> The target type for the conversion.
 */
public abstract class TypeReference<T> {

    protected TypeReference() {
        // Make sure it cannot be directly instantiated
        // but it can be extended (that is the whole idea)
```

```
    }

    /**
     * Return the actual type of this Type Reference
     *
     * @return the type of this reference.
     */
    public Type getType() {
        return ((ParameterizedType) getClass().
            getGenericSuperclass()).getActualTypeArguments()[0];
    }
}
```

5.6 Variable substitution

TODO

5.7 Portable Encodings

Implementations of this specification can provide codecs that produce portable encodings. Implementations must support the encodings using JSON, YAML and XML outlined in this chapter.~~These encodings must be done using either JSON, YAML or XML.~~

Example. This section contains a number of example encodings based of the following in-memory map:

```
        Map<String, Object> m = new HashMap<>();
        m.put("sKey", "a string");
        m.put("iKey", 42);
        m.put("bKey",  true);
        m.put("noKey", null);
        m.put("simpleArray", new int[] {1,2,3});

        Map<String, Object> m1 = new HashMap<>();
        m1.put("a", 1L);
        m1.put("b", "hello");
        m.put("simpleObject", m1);

        Map<String, Object> m2 = new HashMap<>();
        m2.put("yes", Boolean.TRUE);
        m2.put("no", Collections.singletonMap("maybe", false));

        Map<String, Object> cm = new HashMap<>();
        cm.put("list", Arrays.asList(
                Collections.singletonMap("x", "y"),
                Collections.singletonMap("x", "b")));
        cm.put("embedded", m2);
```

5.7.1 Portable JSON encoding

Mime type: application/json, text/json

5.7.1.1 Example JSON-encoded document

The example map encoded into JSON will look as follows:

```
{
```

```
  "sKey": "a string",
  "iKey": 42,
  "bkey": true,
  "noKey": null,
  "simpleArray": [1,2,3],
  "simpleObject": {"a": 1, "b": "hello"},
  "complex": {
    "list": [{"x": "y"}, {"x": "b"}],
    "embedded": {"yes": true,
      "no": {"maybe": false}}
  }
}TODO
```

5.7.2 Portable YAML encoding

Mime-type: application/x-yaml, text/yaml

5.7.2.1 Example YAML-encoded document

TODO

5.7.3 Portable XML encoding

Mime-type: text/xml

5.7.3.1 Example XML-encoded document

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
    <sKey>a string</sKey>
    <iKey type="Integer">42</iKey>
    <bKey type="Boolean">true</bKey>
    <noKey/>
    <simpleArray>
        <value type="Integer">1</value>
        <value type="Integer">2</value>
        <value type="Integer">3</value>
    </simpleArray>
    <simpleObject>
        <a type="Long">1</a>
        <b>hello</b>
    </simpleObject>
    <complex>
        <list>
            <value>
                <x>y</x>
            </value>
            <value>
                <x>b</x>
            </value>
        </list>
        <embedded>
            <yes type="Boolean">true</yes>
            <embedded>
                <no>
                    <maybe type="Boolean">false</maybe>
                </no>
```

```
            </embedded>
        </embedded>
    </complex>
</root>~~TODO~~
```

5.7.3.2 XML Schema

TODO

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

7 Javadoc

OSGi Javadoc

4/20/16 7:00 PM

Package Summary		Page
org.osgi.service.converter	Converter Package.	26

Package org.osgi.service.converter

```
@org.osgi.annotation.versioning.Version(value="1.0")
```

Converter Package.

See:

Description

Interface Summary		Page
Adapter	An `Adapter` is used to modify the behaviour of the Converter service, which can be useful when some of the conversions should be done different to the Converter Specification.	27
Codec	The Codec service can be used to encode a given object in a certain representation, for example JSON, YAML or XML.	29
Converter	The Converter service is used to start a conversion.	32
Converting	This interface is used to specify the target that an object should be converted to.	33
Decoding	Interface to specify the source of the decoding operation	35
Encoding	Interface to specify the target of the encoding operation.	37

Class Summary		Page
Rule	A rule is a data entity can hold all the information needed to specify a custom conversion for use by an @{link Adapter}.	39
TypeReference	An object does not carry any runtime information about its generic type.	41

Exception Summary		Page
ConversionException	This Runtime Exception is thrown when an object is requested to be converted but the conversion cannot be done.	31

Package org.osgi.service.converter Description

Converter Package.

Interface Adapter

org.osgi.service.converter

All Superinterfaces:

Converter

```
public interface Adapter
extends Converter
```

An `Adapter` is used to modify the behaviour of the Converter service, which can be useful when some of the conversions should be done different to the Converter Specification.

Method Summary		Page
Adapter	**rule**(Class<F> fromCls, Class<T> toCls, Function<F,T> toFun, Function<T,F> fromFun) Specify a rule for the conversion to and from two classes.	27
Adapter	**rule**(Function<F,T> toFun, Function<T,F> fromFun) Specify a rule for the conversion to and from two classes.	28
Adapter	**rule**(Rule<F,T> rule) Specify a conversion rule by providing a rule object.	27

Methods inherited from interface org.osgi.service.converter.Converter
convert, getAdapter

Method Detail

rule

```
Adapter rule(Rule<F,T> rule)
```

Specify a conversion rule by providing a rule object.

Parameters:

`rule` - The conversion rule.

Returns:

The current adapter, can be used to chain invocations.

---

rule

```
Adapter rule(Class<F> fromCls,
             Class<T> toCls,
             Function<F,T> toFun,
             Function<T,F> fromFun)
```

Specify a rule for the conversion to and from two classes. The rule specifies the conversion in both directions. This overload makes it easy to provide the conversions as lambdas, for example:

```
adapter.rule(String[].class, String.class,
    v -> Stream.of(v).collect(Collectors.joining(",")),
    v -> v.split(","));
```

Type Parameters:
`F` - the type to convert from.
`T` - the type to convert to.
Parameters:
`fromCls` - the class to convert from.
`toCls` - the class to convert to.
`toFun` - the function to perform the conversion.
`fromFun` - the function to perform the reverse conversion.
Returns:
The current adapter, can be used to chain invocations.

---

rule

```
Adapter rule(Function<F,T> toFun,
             Function<T,F> fromFun)
```

Specify a rule for the conversion to and from two classes. The rule specifies the conversion in both directions. This overload makes it easy to provide the conversions as method references.

Type Parameters:
`F` - the type to convert from.
`T` - the type to convert to.
Parameters:
`toFun` - the function to perform the conversion.
`fromFun` - the function to perform the reverse conversion.
Returns:
The current adapter, can be used to chain invocations.

Interface Codec

org.osgi.service.converter

```
public interface Codec
```

The Codec service can be used to encode a given object in a certain representation, for example JSON, YAML or XML. The Codec service can also decode the representation it produced. A single Codec service can encode/decode only a single format. To support multiple encoding formats register multiple services.

Method Summary		Page
Decoding<T>	**decode**(Class<T> cls) Start specifying a decode operation.	29
Decoding<?>	**decode**(Type type) Start specifying a decode operation.	30
Decoding<T>	**decode**(TypeReference<T> ref) Start specifying a decode operation.	29
Encoding	**encode**(Object obj) Start specifying an encode opertation.	30
Codec	**with**(Converter converter) Specify the converter to be used by the code, if an alternative, adapted, converter is to be used.	30

Method Detail

decode

Decoding<T> **decode**(Class<T> cls)

Start specifying a decode operation.

Type Parameters:
T - The type to decode to.
Parameters:
cls - The class to decode to.
Returns:
A Decoding object to specify the source for the decode operation.

---

decode

Decoding<T> **decode**(TypeReference<T> ref)

Start specifying a decode operation.

Type Parameters:
T - The type to decode to.
Parameters:
ref - A type reference for the target type.
Returns:
A Decoding object to specify the source for the decode operation.

---

decode

`Decoding<?> decode(Type type)`

Start specifying a decode operation.

Parameters:
`type` - The type to convert to.
Returns:
A `Decoding` object to specify the source for the decode operation.

---

encode

`Encoding encode(Object obj)`

Start specifying an encode opertation.

Parameters:
`obj` - The object to encode.
Returns:
an Encoding object to specify the target for the decode operation.

---

with

`Codec with(Converter converter)`

Specify the converter to be used by the code, if an alternative, adapted, converter is to be used.

Parameters:
`converter` - The converter to use.
Returns:
A codec that uses the converter as specified.

Class ConversionException

org.osgi.service.converter

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └org.osgi.service.converter.ConversionException
```

All Implemented Interfaces:

Serializable

```
public class ConversionException
extends RuntimeException
```

This Runtime Exception is thrown when an object is requested to be converted but the conversion cannot be done. For example when the String "test" is to be converted into a Long.

Constructor Summary	Page
`ConversionException(String message)` Create a Conversion Exception with a message.	31
`ConversionException(String message, Throwable cause)` Create a Conversion Exception with a message and a nested cause.	31

Constructor Detail

ConversionException

```
public ConversionException(String message)
```

Create a Conversion Exception with a message.

Parameters:

`message` - The message for this exception.

ConversionException

```
public ConversionException(String message,
                           Throwable cause)
```

Create a Conversion Exception with a message and a nested cause.

Parameters:

`message` - The message for this exception.
`cause` - The causing exception.

Interface Converter

org.osgi.service.converter

All Known Subinterfaces:
Adapter

`public interface` **`Converter`**

The Converter service is used to start a conversion. The service is obtained from the service registry. The conversion is then completed via the Converting interface that has methods to specify the target type.

Method Summary		Page
Converting	**`convert`**`(Object obj)` Start a conversion for the given object.	32
Adapter	**`getAdapter`**`()` Obtain an adapter to this converter.	32

Method Detail

convert

Converting **`convert`**`(Object obj)`

Start a conversion for the given object.

Parameters:
`obj` - The object that should be converted.
Returns:
A `Converting` object to complete the conversion.

---

getAdapter

Adapter **`getAdapter`**`()`

Obtain an adapter to this converter. The adapter behaves just like the converter except for the exception rules registered with is. For more details see the `Adapter` interface.

Returns:
An adapter to this converter.

Interface Converting

org.osgi.service.converter

`public interface` **`Converting`**

This interface is used to specify the target that an object should be converted to. A `Converting` instance can be obtained via the `Converter` service by starting a conversion for a specific object.

Method Summary		Page
`Converting`	**`defaultValue`**`(Object defVal)` The default value to use when the object cannot be converted or in case of conversion from a `null` value.	33
`T`	**`to`**`(Class<T> cls)` Specify the target object type for the conversion as a class object.	33
`Object`	**`to`**`(Type type)` Specify the target object type as a Java Reflection Type object.	34
`T`	**`to`**`(TypeReference<T> ref)` Specify the target object type as a `TypeReference`.	33
`String`	**`toString`**`()` Same as `to(String.class)`.	34

Method Detail

defaultValue

`Converting` **`defaultValue`**`(Object defVal)`

The default value to use when the object cannot be converted or in case of conversion from a `null` value.

Parameters:
`defVal` - The default value.
Returns:
The current `Converting` object so that additional calls can be chained.

---

to

`T` **`to`**`(Class<T> cls)`

Specify the target object type for the conversion as a class object.

Parameters:
`cls` - The class to convert to.
Returns:
The converted object.

---

to

`T` **`to`**`(TypeReference<T> ref)`

Specify the target object type as a `TypeReference`. If the target class carries generics information a TypeReference should be used as this preserves the generic information whereas a Class object has this information erased. Example use:

```
  List result =
      converter.convert(Arrays.asList(1,2,3)).
          to(new TypeReference<List<String>>() {});
```

Parameters:
`ref` - A type reference to the object being converted to.
Returns:
The converted object.

---

to

```
Object to(Type type)
```

Specify the target object type as a Java Reflection Type object.

Parameters:
`type` - A Type object to represent the target type to be converted to.
Returns:
The converted object.

---

toString

```
String toString()
```

Same as `to(String.class)`.

Overrides:
`toString` in class `Object`
Returns:
The converted object.

Interface Decoding

org.osgi.service.converter

Type Parameters:
`T` - The target type for the decoding operation.

`public interface` **`Decoding`**

Interface to specify the source of the decoding operation

Method Summary		Page
T	**from**(InputStream in) Use an input stream as the source of the decoding operation.	35
T	**from**(InputStream in, Charset charset) Use an input stream as the source of the decoding operation.	35
T	**from**(CharSequence in) Use a Char Sequence as the source of the decoding operation.	36
T	**from**(Readable in) Use a Readable as the source of the decoding operation.	35

Method Detail

from

`T` **`from`**`(InputStream in)`

Use an input stream as the source of the decoding operation. As encoding UTF-8 is used.

Parameters:
`in` - The stream to use.
Returns:
the decoded object.

---

from

`T` **`from`**`(InputStream in,`
`Charset charset)`

Use an input stream as the source of the decoding operation.

Parameters:
`in` - The stream to use.
`charset` - The character set to use.
Returns:
the decoded object.

---

from

`T` **`from`**`(Readable in)`

Use a Readable as the source of the decoding operation.

Parameters:
`in` - The readable to use.
Returns:
the decoded object.

---

from

```
T from(CharSequence in)
```

Use a Char Sequence as the source of the decoding operation.

Parameters:
`in` - The char sequence to use.
Returns:
the decoded object.

Interface Encoding

org.osgi.service.converter

```
public interface Encoding
```

Interface to specify the target of the encoding operation.

Method Summary		Page
Encoding	**pretty**() Specify that the encoded output should be formatted to look 'pretty', which may make it easier for humans to read.	37
void	**to**(OutputStream out) Use an output stream as the target of the encoding operation.	37
void	**to**(OutputStream out, Charset charset) Use an output stream as the target of the encoding operation.	37
Appendable	**to**(Appendable out) Encode the object and append the result to an appendable.	38
String	**toString**() Encode the object and return the result as a string.	38

Method Detail

pretty

```
Encoding pretty()
```

Specify that the encoded output should be formatted to look 'pretty', which may make it easier for humans to read. If not specified, the encoded output should be formatted to be compact, so save space.

Returns:
This Encoding object to allow further invocations on it.

---

to

```
void to(OutputStream out)
        throws IOException
```

Use an output stream as the target of the encoding operation. UTF-8 will be used.

Parameters:
`out` - The output stream to use.
Throws:
`IOException`

---

to

```
void to(OutputStream out,
        Charset charset)
        throws IOException
```

Use an output stream as the target of the encoding operation.

Parameters:
`out` - The output stream to use.
`charset` - The character set to use.
Throws:
`IOException`

---

to

```
Appendable to(Appendable out)
```

Encode the object and append the result to an appendable.

Parameters:
`out` - The appendable object to use.
Returns:
The appendable object provided in, which allows further appends to it be done in a fluent programming style.

---

toString

```
String toString()
```

Encode the object and return the result as a string.

Overrides:
`toString` in class `Object`
Returns:
The encoded object.

Class Rule

org.osgi.service.converter

```
java.lang.Object
  └ org.osgi.service.converter.Rule
```

Type Parameters:

`F` - The type to convert from.
`T` - The type to convert to.

```
public class Rule
extends Object
```

A rule is a data entity can hold all the information needed to specify a custom conversion for use by an @{link Adapter}.

Constructor Summary	Page
Rule(Function<F,T> to, Function<T,F> from) Specify the functions to do the conversions in both directions.	39

Method Summary		Page
Function<T,F>	**getFromFunction**() Obtain the reverse conversion function.	40
Function<F,T>	**getToFunction**() Obtain the conversion function.	39

Constructor Detail

Rule

```
public Rule(Function<F,T> to,
            Function<T,F> from)
```

Specify the functions to do the conversions in both directions.

Parameters:

`to` - The function that performs the conversion.
`from` - The function that performs the reverse conversion.

Method Detail

getToFunction

```
public Function<F,T> getToFunction()
```

Obtain the conversion function.

Returns:

The conversion function.

---

getFromFunction

```
public Function<T,F> getFromFunction()
```

Obtain the reverse conversion function.

Returns:
The reverse conversion function.

Class TypeReference

org.osgi.service.converter

```
java.lang.Object
  └ org.osgi.service.converter.TypeReference
```

Type Parameters:

T - The target type for the conversion.

```
public class TypeReference
extends Object
```

An object does not carry any runtime information about its generic type. However sometimes it is necessary to specify a generic type, that is the purpose of this class. It allows you to specify an generic type by defining a type T, then subclassing it. The subclass will have a reference to the super class that contains this generic information. Through reflection, we pick this reference up and return it with the getType() call.

```
List result =
    converter.convert(Arrays.asList(1,2,3)).
        to(new TypeReference<List<String>>() {});
```

Constructor Summary		Page
protected	**TypeReference**() A TypeReference cannot be directly instantiated.	41

Method Summary		Page
Type	**getType**() Return the actual type of this Type Reference	41

Constructor Detail

TypeReference

```
protected TypeReference()
```

A TypeReference cannot be directly instantiated. To use it it has to be extended, typically as an anonymous inner class.

Method Detail

getType

```
public Type getType()
```

Return the actual type of this Type Reference

Returns:

the type of this reference.

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

This section is placed here for the moment, we may use parts of it in the future.

dest v / src ->	String	*Boxed*	*primitive*	*Object*	*primitive[]*	*Boxed[]*	*collection<?>*	`null`
String	*v*	*v*.toString()	String.valueOf(*v*)	*v*.toString()	Arrays.toString(*v*) except for char[]: String.valueOf(*v*)	Arrays.toString(*v*)	*v*.toString()	null
String[]	new String[]{*v*}	new String[]{*v*.toString()}	new String[]{String.valueOf(*v*)}	if String[]: *v* otherwise: new String[]{*v*.toString()}	Arrays.stream(*v*).mapToObj(String::valueOf).toArray(String[]::new)	Arrays.stream(v).map(String::valueOf).toArray(String[]::new)	*v*.stream().map(String::valueOf).toArray(String[]::new)	new String[]{}
List<String>	Collections.singletonList(*v*)	Collections.singletonList(»String(*v*))	Collections.singletonList(»String(*v*))	Collections.singletonList(»String(*v*))	Arrays.stream(*v*).mapToObj(String::valueOf).collect(toList())	Arrays.stream(*v*).map(String::valueOf).collect(toList())	*v*.stream().map(String::valueOf).collect(toList())	Collections.emptyList()
Set<String>	Collections.singleton(v)	Collections.singleton(»String(*v*))	Collections.singleton(»String(*v*))	Collections.singleton(»String(*v*))	Arrays.stream(*v*).mapToObj(String::valueOf).collect(toSet())	Arrays.stream(*v*).map(String::valueOf).collect(toSet())	*v*.stream().map(String::valueOf).collect(toSet())	Collections.emptySet()
Collection<String>	*pick either list or set*							
int	Integer.parseInt(*v*)	*v*.intValue()	if int: *v* otherwise: (int) *v*	»int(v.toString())	if *v*.length == 0: 0 otherwise: »int(*v*[0])	if *v*.length == 0: 0 otherwise: »int(*v*[0])	if *v*.size() == 0: 0 otherwise: »int(*v*.iterator().next())	0
boolean	Boolean.valueOf(*v*)	if Boolean: *v*.booleanValue() otherwise: »int(*v*) != 0	if boolean: *v* otherwise: »int(*v*) != 0	»boolean(v.toString())	if *v*.length == 0: false otherwise: »boolean(*v*[0])	if *v*.length == 0: false otherwise: »boolean(*v*[0])	if *v*.size() == 0: false otherwise: »boolean(*v*.iterator().next())	false
char	*v*.length() > 0 ? *v*.charAt(0) : 0	(char) *v*.*number*Value()	(char) *v*	»char(v.toString())	if *v*.length == 0: 0 otherwise: »char(*v*[0])	if *v*.length == 0: 0 otherwise: »char(*v*[0])	if *v*.size() == 0: 0 otherwise: »char(*v*.iterator().next())	0
byte	*v*.getBytes()[0] or 0 if no bytes in array.	(byte) *v*.intValue()	(byte) *v*	»byte(v.toString())	if *v*.length == 0: 0 otherwise: »char(*v*[0])	if *v*.length == 0: 0 otherwise: »char(*v*[0])	if *v*.size() == 0: 0 otherwise: »byte(*v*.iterator().	0

							next())	
short								
float								
double	Double. parseDouble(*v*)	v.doubleValue ()	(double) *v*	Double. parseDouble(*v*.toString())	i f *v*.length == 0: 0.0 otherwise: »double(*v*[0])	i f *v*.length == 0: 0.0 otherwise: »double(*v*[0])	i f *v*.size() == 0: 0.0 otherwise: »double(*v*. iterator().next( ))	0
int[]	n e w i n t [ ] {»int(*v*)}	new int[] {»int(*v*)}	new int[] {»int(*v*)}	new int[] {»int(*v*)}	Arrays.stream (*v*). mapToInt(l -> ((*Boxed*) l). intValue()). toArray()	Arrays.stream (*v*). mapToInt(*Boxed*::intValu e). toArray();	*v*.stream(). mapToInt(x → »int(x)). toArray()	new int[]{}
List<Integer>	Collections. singletonList(»int(*v*));	Collections. singletonList(»int(*v*));	Collections. singletonList(»int(*v*));	Collections. singletonList(»int(*v*));	Arrays.stream (*v*). mapToObj(*Boxed*::value Of). collect(toList());	Arrays.stream (*v*). map(*Boxed*::intValu e). collect(toList());	*v*.stream(). map(x → »int(x)). collect(toList());	Collections. emptyList()
Boolean	Boolean.value Of(*v*)	if Boolean: *v* otherwise: »int(*v*) != 0	if boolean: Boolean.value Of(*v*) otherwise: »int(*v*) != 0	»Boolean(v.toString())	i f *v*.length == 0: FALSE otherwise: »Boolean(*v*[0])	i f *v*.length == 0: FALSE otherwise: »Boolean(*v*[0])	i f *v*.size() == 0: FALSE otherwise: »Boolean(*v*. iterator().next( ))	null
other Boxed types								

Identity conversion (source can be assigned to target → straight passthrought)

Special case for byte[]

ObjectClass::valueOf

String constructor

8.1.1.1 Object[]

Object[] is similar to Collection<?> although String[] can be converted to List<String> via Arrays.asList(v).

8.1.1.2 Enumerated types

Converting to/from Enum Types is only possible between the enumerated types and their String representation.

8.1.1.3 Other types

Do we want to support the following types: Class, Annotation, BigDecimal/BigInteger?

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	+33 467542167
e-mail	Peter.kriens@aQute.biz

Name	David Bosschaert
Company	Adobe Systems
Address	
Voice	
e-mail	bosschae@adobe.com

Name	
Company	
Address	
Voice	
e-mail	

10.3 Acronyms and Abbreviations

10.4 End of Document