RFC 215 - Object Conversion

Draft

26 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

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

0.5 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 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

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 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 {
    Codec configure(String key, Object value);
    Codec configure(Map<String, Object> configs);

    <T> Decoding<T> decode(Class<T> cls);
    <T> Decoding<T> decode(TypeReference<T> ref);
    Decoding<?>Object decode(Type type);

    EncodingString encode(Object obj);

    // If this codec is invoked by another codec, e.g. a wrapper, register the top codec so
    // that it can be recursed into for subelements. Typically used by CodecAdapters
    Codec with(Codec topCodec);
}
```

Codec service properties:

property name	***type***	***description***
`osgi.codec.name`	String	The name of this codec implementation.

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
    T from(InputStream in, String charset);
    T from(Readable in);
    T from(CharSequence in);
}

public interface Encoding {
    void to(OutputStream out); // uses UTF-8
    void to(OutputStream out, String charset);
    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.configure("pretty", true).encode(myObj).toString();
```

5.2 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 {

    CodecAdapter getCodecAdapter(Codec codec); // Can modify a codec

    Codec getDefaultCodec(); // Returns a codec that converts to/from Strings as in 5.4

    Converting convert(Object obj);

}

public interface Converting {

    <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

    Converting with(Codec codec); // Use an alternative codec
}
```



```
public interface CodecAdapter extends Codec {
    // Specify a rule where the adapter deviates from the adapted codec.
    <T> CodecAdapter rule(Class<T> cls, Function<T, String> toString,
                                        Function<String, T> fromString);
}
```

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 String[] is handled
    CodecAdapter ca = c.getCodecAdapter(c.getDefaultCodec());
    ca.rule(String[].class,
        v -> Stream.of(v).collect(Collectors.joining(",")),
        v -> v.split(","));
    String s = c.convert(new String[] {"A","B"}.with(ca).to(String.class);
```

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).

```
U();

result    Encoding pretty();

    void to(OutputStream out); // uses UTF-8
    void to(OutputStream out, String charset);
    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().ignoreNull().    Encoding withNull();
                 Encoding compact();

    T from(InputStream in); // uses UTF-8
    T from(InputStream in, String charset);
    T from(Readable in);
    T from(CharSequence in);
}

public interface Encoding {
                 Encoding ignoreNull();
    Decoding<T> withNull();
 encode(Object obj);
```


~~}~~

~~public interface Decoding<T> {~~
~~D e c o d i n g < T > i g n o r e N u l l ( ) ;~~
~~Encoding decode(Type type);~~

~~Decoding<?> decode(TypeReference<T> ref);~~
~~>TDecoding< decode(Class<T> cls);~~
~~<T> >TDecoding<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>~~

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 getCodecByName(String name); // e.g. “Joe's Codec”
    Codec getCodecByMimeType(String type); // e.g. “application/json~~YAML~~”
}
```

With the CodecFactory obtain the desired codec as follows:

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

5.4 Conversions

The following conversions will be supported.

This section describes conversions to String and from String for the default codec of the encoder. Users can provide alternative representations by creating their own `Codec` that delegates to the default codec 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	“”
boolean	Boolean. parseBoolean(v)	v.booleanValue()	v.charValue() != 0	v.double~~number~~Value() != 0	FALSE	FALSE
char	v.lenght() > 0 v.charAt(0) : 0	v.booleanValue()	v.charValue()	(char) v.~~number~~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)	throw	throw	throw	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

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.

- ~~`static valueOf(String s)`~~
- ~~`static fromString(String s)`~~

Exceptions:

- `null` values will result in a `null` value for the target value.

- Empty arrays / collections will be converted into a `null` target value.

5.4.2 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 a different conversion to String, for example a comma-separated list, can do so by providing their own codec for the conversion.
~~Exception: `byte[]` → String: will be converted by calling~~
~~`new String(byte[] v, StandardCharsets.UTF_8)`~~

~~Exception: Collection/Array → String: will comma-separate the elements and concatenate them to form a single String.~~

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[]`	`new T[x]` where *x* is the size of the source collection, 1 in case of a scalar.

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 value on the map is considered an element to be inserted. Map keys are discarded in this case.

Once the elements to be inserted are established, ~~Then, for each element to be inserted into the resulting collection/array, the~~ 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>>() {})
```

~~Exceptions:~~

- ~~String → Collection/Array: will parse the string as a comma-separated list of entries which are then added to the target array.~~

~~String → `byte[]`: will be produced using `String.getBytes(StandardCharsets.UTF_8)`.~~

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:

```
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

  @AttributeDefinition(defaultValue="22")
```



```
  boolean my_other_value();
}

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.

~~TODO~~ 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.

5.4.3.5 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.

5.4.3.6 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.

5.4.3.7 To other types

Convert `map.values()` to an order-preserving collection and then convert this collection to the target type.

~~TODO Are conversions to other types supported in a generalized manner? Note the section 'exceptions' below.~~

Not supported.

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`
- `org.osgi.framework.Bundle` (returns bundle headers)
- `org.osgi.service.Subsystem` (returns subsystem headers)
- `org.osgi.resource.Capability` (returns attributes)
- `org.osgi.resource.Requirement` (returns directives)
- `org.osgi.service.event.Event`

The new method will not be added via an interface, the Converter should use reflection to find this method when it needs to convert an object.

5.4.5 Special Cases

~~…~~ This section~~ODO~~ specifies~~y~~ special cases also supported by the converter.~~not covered above …~~

5.4.5.1 UUID

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

5.4.5.2 Locale

TODO: How to do this one? new Locale(String) does not produce the same as the original (e.g. new Locale("fr_CA") produces "fr_ca" which is not the same as Locale.Canada_FRENCH...

5.4.5.3 Pattern

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

5.4.5.4 Calendar

Conversion from a Calendar to a String is done via Java 8 date/time API:

```
myCal.getTime().toInstant().atZone(myCal.getTimeZone().toZoneId()).toString()
```

This produces strings that look like this: `2016-01-28T13:50:52.370Z[Europe/Dublin]`

Conversion from a String representation of time to a Calendar is done with:

```
  ZonedDateTime zdt = ZonedDateTime.parse(myString);
  Calendar c = Calendar.getInstance(TimeZone.getTimeZone(zdt.getZone()));
  c.setTimeInMillis(zdt.toInstant().getEpochSecond() * 1000);
```

5.4.5.5 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);
```

5.4.5.6 Date

A `java.util.Date` instance is converted to a String value by calling `Date.getTime()` and converting the resulting `long` value into a String.

Converting a String into a `java.util.Date` is done by converting the String to a long and then calling new `Date(long)`.

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 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. These encodings must be done using either JSON, YAML or XML.

5.7.1 Portable JSON encoding

Mime type: application/json

5.7.1.1 Example JSON-encoded document

TODO

5.7.2 Portable YAML encoding

Mime-type: application/x-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

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

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: https://www.osgi.org/members/RFC/Javadoc

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::value	Arrays.stream (*v*). map(String::value	*v*.stream().map(String::valueOf).	Collections. emptySet()

					Of). collect(toSet())	Of). collect(toSet())	collect(toSet())	
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(). next())	0
short								
float								
double	Double. parseDouble(*v*)	v.doubleValue()	(double) *v*	Double. parseDouble(*v*.toString())	if *v*.length == 0: 0.0 otherwise: »double(*v*[0])	if *v*.length == 0: 0.0 otherwise: »double(*v*[0])	if *v*.size() == 0: 0.0 otherwise: »double(*v*. iterator().next())	0
int[]	new int[] {»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*::intValue). 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*::valueOf). collect(toList());	Arrays.stream(*v*). map(*Boxed*::intValue). collect(toList());	*v*.stream(). map(x »int(x)). collect(toList());	Collections. emptyList()
Boolean	Boolean.valueOf(*v*)	if Boolean: *v* otherwise: »int(*v*) != 0	if boolean: Boolean.valueOf(*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())	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