JideFX Converters Developer Guide

Table of Contents

Purpose of This Document

Welcome to the **JideFX Converters**, a collection of converters for the JavaFX platform. A converter can convert from a data type to String and back. This document is for developers who want to develop applications using **JideFX Converters**.

What is JideFX Converters

The "decorating" technique is to put a node or nodes next to, or over another node without changing the existing layout. It would be a lot easier for the developer to focus on the overall layout and doesn't have to worry about those small decorations when designing the layout. Furthermore, the decorations can be added or removed or adjusted on fly based on certain conditions, without modifying the existing layout. This product is all about how to make it happen.

Most common usages of the decoration is to show validation results, additional help/hint information, or show the progress. For example:

- A help icon next to a field which shows a help message when clicking or hovering
- A lock icon next to a field to indicate the field is secured
- An asterisk sign to indicate a field is required
- A watermark over the form to indicate the content is confidential or submitted
- A prompt text that tell users what the control is for

Packages

The table below lists the packages in the JideFX Decoration product.

Packages	Description
jidefx.scene.utils.converter	**Converters for general data types**
jidefx.scene.utils.converter.javafx	**Converters for JavaFX data types**
jidefx.scene.utils.converter.time	**Converters for data types under java.time**

Dependency

The **JideFX Converters** doesn't depends on any other JideFX products.

ObjectConverter

If you used the JIDE Grids for Swing before, you should already be familiar with ObjectConverters. The ObjectConverter can convert any objects to String and back. In the **JideFX Grids**, we heavily use ObjectConverters to handle the display of a majority of data types, the editing of some of the data types that can be edited as String.

However, the ObjectConverters are not only useful in the **JideFX Grids**. You can use it in a lot of places in your application. As we all know, to simplify the display of various data types, most UI controls display data as String if possible. For example, TextField or Label only can display String. It means we need some kinds of conversion that converts from any types of data to String so that it can be displayed in the text field or the label. Editing in the text field is the opposite. It needs a converter that converts from String to any data type. Here comes the ObjectConverter.

We had ObjectConverter concept in JIDE Swing package for many years. JavaFX introduced a new class called StringConverter for the same purpose. But since our converter supports ConverterContext, we decide to keep ours and add a toStringConverter method to ObjectConverter interface in case you want to use ObjectConverter as a JavaFX StringConverter.

Below is the interface of ObjectConverter. All converters in the **JideFX Converters** implement this interface.

```
/**
 * An interface that can convert ab object to a String and convert from String to
an object.
 */
public interface ObjectConverter<T> {

    /**
     * Converts from object to String based on current locale.
     *
     * @param object  object to be converted
     * @param context converter context to be used
     * @return the String
     */
    String toString(T object, ConverterContext context);

    /**
     * Converts from String to an object.
     *
     * @param string  the string
     * @param context context to be converted
     * @return the object converted from string
     */
    T fromString(String string, ConverterContext context);

    /**
     * Creates a compatible StringConverter from ObjectConverter.
     *
     * @return a StringConverter.
     */
    StringConverter<T> toStringConverter();
}
```

As an example, assume you are dealing with a Rectangle object, specified as (10, 20, 100, 200). If you represent this Rectangle as the string “10; 20; 100; 200” then 80% of users will

probably understand it as a Rectangle with x equals 10, y equals 20, width equals 100 and height equals 200. However, what about the other 20% of the people? Well, they might think it is an int array of four numbers. That's fine. Users can generally learn by experience: as long as you are consistent across your application, users will get used to it. What is more important is you use the converter consistently across your whole application.

The situation is slightly more complicated in the case of Color. If we consider the string "0, 100, 200" - if people understand the RGB view of Color then 90% of them will treat as 0 as red, 100 as blue and 200 as green. However, since Color can also be represented in HSL color space (Hue, Saturation, and Lightness), some people may consider it as hue equal 0, saturation equals 100 and lightness equals 200. Another way to represent the color is to use the HTML color name such as "#00FFFF". If your application is an html editor, you probably should use a converter to convert color to "#00FFFF" instead of "0, 255, 255". What this means is that, based on your users' background, you should consider adding more help information if ambiguity may arise.

We also need to consider internationalization, since the string representation of any object may be different under different locales.

In conclusion, we need a series of converters that convert objects so that we can display them as string and convert them back from string. However in different applications, different converters are required.

Although we have already built some converters and will add more over time, it is probably true that there will never be enough. Therefore, please be prepared to create your own converters whenever you need one.

Compatibility with StringConverter

Interesting enough, JavaFX introduced a new class call StringConverter.

```
/**
 * Converter defines conversion behavior between strings and objects.
 * The type of objects and formats of strings are defined by the subclasses
 * of Converter.
 */
public abstract class StringConverter<T> {
    /**
    * Converts the object provided into its string form.
    * Format of the returned string is defined by the specific converter.
    * @return a string representation of the object passed in.
    */
    public abstract String toString(T object);

    /**
    * Converts the string provided into an object defined by the specific
converter.
    * Format of the string and type of the resulting object is defined by the
specific converter.
    * @return an object representation of the string passed in.
    */
    public abstract T fromString(String string);
}
```

JavaFX also defined some StringConverters.

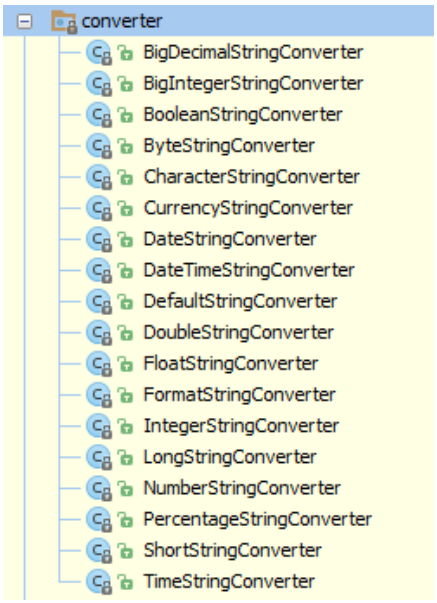

Figure 1 Existing StringConverters in JavaFX

If you felt like having more StringConverters that JavaFX doesn't provide, you can take any our ObjectConverter and call converter.toStringConverter() to get a StringConverter. In fact, all built-in ObjectConverters extend StringConverter at the moment so you could use them directly when you need a StringConverter. However, to keep the maximum compatibility, we recommend you always use converter.toStringConverter method.

As you can see, the only difference between StringConverter and ObjectConverter is StringConverter doesn't have a ConverterContext concept. So what is the ConverterContext for? In order to answer that, we will have to cover the ObjectConverterManager first.

ObjectConverterManager

You can create your own converter for a specific data type by implementing the ObjectConverter interface. When there are too many converters, you wish the converters can be more discoverable through a central registration. The ObjectConverterManager is such a central registration. You can register any converters with the ObjectConverterManager, which is a HashMap that maps from a Class (the data type) to a converter or several converters. But what if you want to register several converters for the same data type? Using a ConverterContext.

ConverterContext

In the Color example above, we mentioned there are different ways to convert a Color to a String. We can create several converters for Color. However, when we try to register them on the ObjectConverterManager, we got problem. We can't register all converters to the same data type as they will overwrite each other in the HashMap. The only way is to use a two-key HashMap. The primary key is of course the data type; the secondary key would be the ConverterContext. So serving as the secondary key in the ObjectConverterManager is the main

purpose of the ConverterContext. The main field in a ConverterContext is the name. If two ConverterContexts have the same name, they are considered as equal.

The code below shows you how to define the ConverterContext in the case of Color.

```
/**
 * ConverterContext for color to convert to RGB string.
 */
public static ConverterContext CONTEXT_RGB = ConverterContext.DEFAULT_CONTEXT;

/**
 * ConverterContext for color to convert to HEX string.
 */
public static ConverterContext CONTEXT_HEX = new ConverterContext("Hex");

/**
 * ConverterContext for color to convert to RGB and alpha string.
 */
public static ConverterContext CONTEXT_RGBA = new ConverterContext("RGBA");

/**
 * ConverterContext for color to convert to HEX string.
 */
public static ConverterContext CONTEXT_HEX_WITH_ALPHA = new
ConverterContext("HexWithAlpha"); // HEX with Alpha

/**
 * ConverterContext for color to convert to WEB string.
 */
public static ConverterContext CONTEXT_WEB = new ConverterContext("Web");
```

Other than being the secondary key, ConverterContext also has a Properties Map just like the JavaFX's Node. You can put additional data on the Properties Map which can be passed around in case you need them. You can use it to minor tweak the converter. For example, when you convert a number, you may want different precisions. If you are using a StringConverter, you would have to create a different StringConverter for each precision you needed. For ObjectConverter, because we have the ConverterContext, you don't need to. You can define a property like this.

```
public static final String PROPERTY_NUMBER_FORMAT = "NumberFormat";
```

When you do the conversion in the converter, you check if there is a NumberFormat set on the context. If yes, we will use it to convert the number to String. See below.

```
Object format = context != null ?
context.getProperties().get(PROPERTY_NUMBER_FORMAT) : null;
if (format instanceof NumberFormat) {
    try {
        return ((NumberFormat) format).parse(string);
    }
    catch (Exception e) {
        // ignore here. we will use the default way to convert it below
    }
}
```

In all the converters we created, we always created a constant that has prefix PROPERTY_. By looking at what PROPERTY_XXX constants in the converter, you will know what properties you can set to the context for that converter. For example, the code below is from NumberConverter.java.

```
/**
 * A property for the converter context. You can set a {@link NumberFormat} to it
 * and the converter will use it to do the conversion.
 */
public static final String PROPERTY_NUMBER_FORMAT = "NumberFormat";
```

It means you can set a NumberFormat for the NumberConverter.

Register and Unregister Converters

There are two methods on the ObjectConverterManager that can be used to register converters. One takes a converter context. The other one doesn't take in which case the CONTEXT_DEFAULT will be used.

```
public void registerConverter(Class<?> clazz, ObjectConverter converter);

public void registerConverter(Class<?> clazz, ObjectConverter converter,
ConverterContext context);
```

Those are for unregistering converters:

```
public void unregisterConverter(Class<?> clazz);

public void unregisterConverter(Class<?> clazz, ConverterContext context);

public void unregisterAllConverters(Class<?> clazz);

public void unregisterAllConverters();
```

We will automatically register all the default converters with the ObjectConverterManager. If you want it to happen, you can call seAutoInit(false). We do provide a method initDefaultConverters() which you can call anytime to register the default converters.

```
ObjectConverterManager.getInstance().initDefaultConverters();
```

Retrieve a Converter

After you register converters onto the ObjectConverterManager, you can retrieve them later at any time. The following two methods on ObjectConverterManager can be used.

```
public <T> ObjectConverter<T> getConverter(Class<?> clazz);

public <T> ObjectConverter<T> getConverter(Class<?> clazz, ConverterContext
context);
```

For example:

```
ObjectConverter c = ObjectConverterManager.getInstance().getConverter(Color.class);
```

Or

```
ObjectConverter c = ObjectConverterManager.getInstance().getConverter(Color.class,
ColorConverter.CONTEXT_RGBA);
```

Looking up Mechanism in ObjectConverterManager

If you register and retrieve a converter using the same Class, you will get the exact converter. But sometimes, you may register a converter on a class, then retrieve the converter using a subclass of the registered class. For example, there is a converter register for java.util.Date by default. Now if I ask for a converter for java.sql.Date from the ObjectConverterManager, what will I get? The answer is if you never registered a converter specifically for the java.sql.Date, you will get the converter for the java.util.Date because java.sql.Date extends java.utils.Date.

ObjectConverterManager will search for the super classes, super interfaces, primitive type if for a wrapper type, wrapper type for a primitive type, until it found a match. For java.sql.Date, if we didn't find an exact match, we will search for the following classes in order.

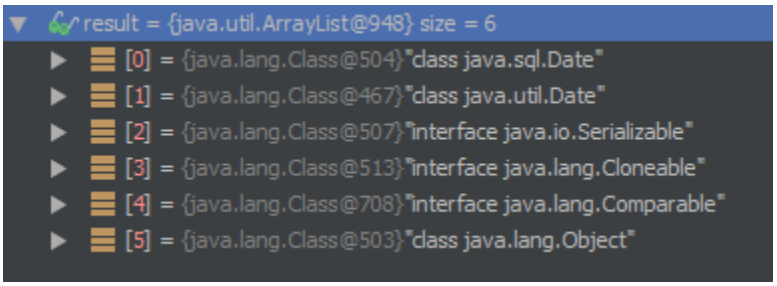

Figure 2 Classes to look up when searching for java.sql.Date

Convert an Object to/from a String

There are two ways to do the conversion. You can retrieve the converter using the code above, then call the ObjectConverter's toString or fromString methods. Or you can call ObjectConverterManager's to String and fromString methods directly. See below for the methods on ObjectConverterManager.

```
public String toString(Object object);

public String toString(Object object, Class<?> clazz);

public String toString(Object object, Class<?> clazz, ConverterContext context);

public Object fromString(String string, Class<?> clazz);

public Object fromString(String string, Class<?> clazz, ConverterContext context);
```

Both approaches will give you the same result except the first approach is more efficient when converting many objects in a row as it saves the time looking for the converter again and again.

Benefit of Having ObjectConverterManager

Software engineering is a never ending topic. One of the topics is the consistency. The larger the project, the harder the consistency. A consistent coding style is one example, which is now enforced by many Java IDEs. In term of the user interface, consistently using the same object conversion on different places in the same application is very important. It doesn't look

professional when an application uses “12/01/13”, “12/1/2013”, “Dec 1, 2013”, “December 01, 2013” or “01/12/13” (Note: the developer of this one happened to be from Europe) etc. various date formats all over the places, sometimes even in the same screen. It will cause confusion for the users, or even mistakes and failures. A centralized ObjectConverterManager will make the conversion consistency happens automatically.

Basically, the ObjectConverterManager will be initialized only once with the preferred converters for all the data types used inside the application. For the date, it will just be one format. All developers will use the registered converters in the ObjectConverterManager, thus they will get the same conversion every time. No arbitrary conversion code is allowed in any other places.

Different Instance of ObjectConverterManager

ObjectConverterManager has a static getInstance() method. This method will give you the default instance. We recommend using this default instance all the time in your application for the reason mentioned in the previous section. However, if you want to create another instance, you can still do it.

Let’s say you have a TableView. In this TableView, you would like to use a special instance of ObjectConverterManager.

```
TableView view = new TableView();
```

Here is how you do it.

```
ObjectConverterManager converterManager = new ObjectConverterManager();
view.getProperties().put(ObjectConverterManager.PROPERTY_OBJECT_CONVERTER_MANAGER,
converterManager);
```

Now you can use this code to get this instance you just created instead of the default instance.

```
ObjectConverterManager converterManager = ObjectConverterManager.getInstance(view);
```

Built-in Object Converters

We created quite a number of object converters in the **JideFX Converters** product. See below. You are always welcomed to create more. If they are generic purpose converters and you would like to share them, please feel free to send us so that we can include them in the future product releases.

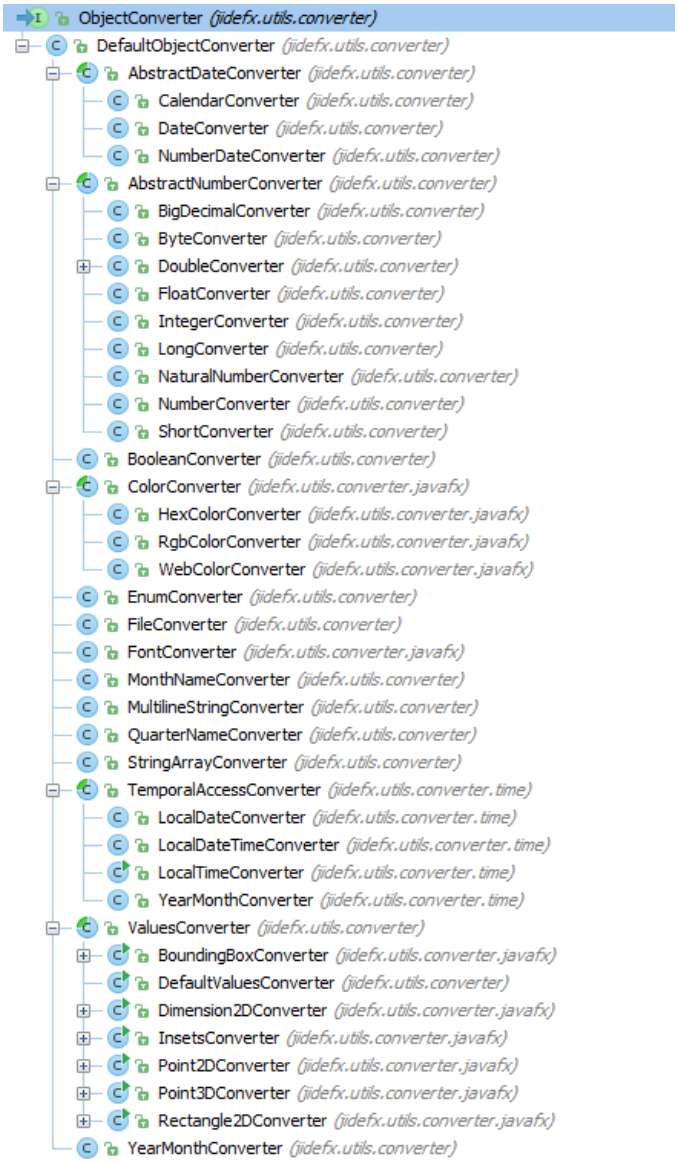

Figure 3 Converters in JideFX Converters

ValuesConverter

ValuesConverter is an abstract class. You can use it as the base for converters of any data types whose value can be represented as a String with a specified separator. For example, Rectangle2D has four double values. The four values can be represented as "0.0, 0.0, 200.0, 100.0". Point2D, Point3D, and Insets are also good examples of using ValuesConverter. For these kind of data types, extending ValuesConverter will save you some effort. See below for the actual code of Point2DConverter.

```
/**
 * {@link jidefx.utils.converter.ObjectConverter} for {@link Point2D}.
 */
public class Point2DConverter extends ValuesConverter<Point2D, Double> {
    public Point2DConverter() {
        super("; ", Double.class);
    }

    public Point2DConverter(String separator) {
        super(separator, Double.class);
    }

    /**
     * Converts the Point2D to String.
     *
     * @param point2D the Point2D
     * @param context the converter context
     * @return the String representing the Point2D.
     */
    public String toString(Point2D point2D, ConverterContext context) {
        if (point2D == null) return null;
        List<Double> list = new ArrayList<>();
        list.add(point2D.getX());
        list.add(point2D.getY());
        return valuesToString(list, context);
    }

    /**
     * Converts from a String to a Point2D.
     *
     * @param string  the string
     * @param context the converter context
     * @return the Point2D represented by te String.
     */
    public Point2D fromString(String string, ConverterContext context) {
        if (string == null || string.trim().length() == 0) {
            return null;
        }
        List<Double> objects = valuesFromString(string, context);
        double x = 0, y = 0;
        if (objects.size() >= 1) {
            Double value = objects.get(0);
            x = value == null ? 0.0 : value;
        }
        if (objects.size() >= 2) {
            Double value = objects.get(1);
            y = value == null ? 0.0 : objects.get(1);
        }
        return new Point2D(x, y);
    }
}
```

The two underlined methods are implemented by ValuesConverter. All we did in the Point2DConverter are collecting the values from Point2D, put them in the list then call valuesToString. Then in the fromString, we create the Point2D from the two values in the list.

EnumConverter

EnumConverter is a generic converter for all Enum types. However, it can also be used for any data types that can be converted using a mapping between values and strings.

Before JDK1.5, there is no Enum type, so this is only one way to define a fake enumeration. For example, in SwingConstants, the following values are defined.

```
public static final int CENTER  = 0;
public static final int TOP     = 1;
public static final int LEFT    = 2;
public static final int BOTTOM  = 3;
public static final int RIGHT   = 4;
```

The problem comes when you want to display it in UI. You don't want to use 0, 1, 2, 3, 4 as the value doesn't mean anything from user point of view. You want to use a more meaningful name such as "Center", "Top", "Left", "Bottom", "Right". Obviously you need a converter here to convert from the integer to string, such as converting from 0 to "Center" and vice verse. To do that, you can create an EnumConverter like this.

```
ObjectConverter converter = new EnumConverter<>("Position", Integer.class, new
Integer[]{0,1,2,3,4}, new String[]{"Center", "Top", "Left", "Bottom", "Right"});

converter.tostring(0); // return "Center"
converter.fromstring("Left"); // return 2
```

See below for another example which converts Boolean to “Yes” and “No” instead of “True” and “False”.

```
EnumConverter requiredConverter = new EnumConverter("Required", Boolean.class, new
Object[]{Boolean.TRUE, Boolean.FALSE}, new String[]{"Yes", "No"});
```

LazyInitializeConverter (Interface)

LazyInitializeConverter is an interface that can be implemented by any object converters to support lazy initialization.

The first reason to use the lazy initialization is when the conversion logic requires the knowledge of the actual data type or the converter context.

For example, for the EnumConverter, we won't know how to convert a value until we know what the enum type is. We could register a converter for each Enum type, but that's way too many. So in the ObjectConverterManager, we only have one entry for all enum types using registerConverter(Enum.class, new EnumConverter()). EnumConverter implements this LazyInitializeConverter. ObjectConverterManager will call the initialize method with the actual enum type when it sees a converter implementing LazyInitializeConverter. In the initialize method of EnumConverter, we will get all enum constants and save them as an array. When toString or fromString is called, we will look up in the array to do the conversion. By using this interface, we don't need to register a converter for each enum type.

Another reason to use this interface is when the converter takes time to create. So instead of initializing the logic up front in the constructor, you can put the expensive logic in this initialize method.

RequiringConverterManager (Interface)

RequiringConverterManager is a markup interface that indicates the object converter needs an ObjectConverterManager instance in order to perform the conversion.

In most cases you don't need the ObjectConverterManager when implementing a converter. However, when there is another child object in the object you want to convert, you can use the ObjectConverterManager to look up for another converter to do the conversion for the child object. Since there are could be many instances of ObjectConverterManager, you want to make sure you are using the same instance which has this converter. That's when you implement RequiringConverterManager on your converter, then you can use getObjectConverterManager method to get the same instance. The instance is actually set on the converter context using property named ConverterContext.PROPERTY_OBJECT_CONVERTER_MANAGER.