# Using custom converters

**Table of contents**



## Using custom converters

A powerful feature of Wicket is that you can customize how input gets converted from strings (because that is what you get from your web browser) to your model objects.

### Collection of custom converters

BigDecimalConverter

### Provide a custom converter factory

If you have conversions that should be done the same way in your whole application, you can provide your own converter factory (instance of `wicket.util.convert.IConverterFactory`). You would use this to force patterns (like always displaying doubles with two fraction digits), or to extend the default set of converters that ships with Wicket with your own types.

#### In Wicket 1.1

To provide your own `IConverterFactory`, you need to override `getConverterFactory` in your application class. For example:

```
/**
 * @see wicket.Application#getConverterFactory()
 */
public IConverterFactory getConverterFactory()
{
  return new IConverterFactory()
  {
    public IConverter newConverter(final Locale locale)
    {
      final Converter converter = new Converter(locale);
      NumberToStringConverter numberToStringConverter = new NumberToStringConverter();
      NumberFormat fmt = NumberFormat.getInstance(locale);
      fmt.setMinimumFractionDigits(2);
      fmt.setMaximumFractionDigits(2);
      numberToStringConverter.setNumberFormat(locale, fmt);
      final StringConverter stringConverter = new StringConverter();
      stringConverter.set(Double.class, numberToStringConverter);
      stringConverter.set(Double.TYPE, numberToStringConverter);
      converter.set(String.class, stringConverter);
      return converter;
    }
  };
}
```

#### In Wicket 1.2

Override init() method in Application class to modify IApplicationSettings.

```
protected void init() {
    super.init();
    IApplicationSettings settings = getApplicationSettings();

    settings.setConverterFactory(new IConverterFactory() {
        public IConverter newConverter(final Locale locale) {
            final Converter converter = new Converter(locale);
            final StringConverter stringConverter = new StringConverter();
            final DateToStringConverter dateToStringConverter = new DateToStringConverter();
            dateToStringConverter.setDateFormat(locale, new SimpleDateFormat("dd/MM/yy"));
            stringConverter.set(Date.class,dateToStringConverter);
            final DateConverter dateConverter = new DateConverter(false);
            // set lenient to false for dateformat so that strict pasrsing is done.
            final DateFormat format = new SimpleDateFormat("dd/MM/yy");
            format.setLenient(false);
            dateConverter.setDateFormat(locale, format);
            converter.set(Date.class,dateConverter);
            converter.set(String.class,stringConverter);
            return converter;
        }
    });
}
```

## In Wicket 1.3

Override newConverterLocator() method in Application class to provide custom ConverterLocator.

```
protected IConverterLocator newConverterLocator() {
    ConverterLocator converterLocator = new ConverterLocator();
    converterLocator.set(Money.class, new MoneyConverter());
    return converterLocator;
}
```

## In Wicket 1.4

This is the same as in 1.3.

My customers do not know what a "Double" is, so the message "'value' is not a valid Double" doesn't make sense. I wanted to provide a simpler message.

### Custom message for class Double

First, as with other converters, you override newConverterLocator in your application.

**Application.java**

```
public class Application extends WebApplication {
...
    @Override
    protected IConverterLocator newConverterLocator() {
        ConverterLocator locator = (ConverterLocator) super.newConverterLocator();
        locator.set(Double.class, new MyDoubleConverter());
        return locator;
    }
    ....
```

This tells Wicket for the Double class, use MyDoubleConverter for converting, which also takes care of reporting the error when not parsable. It seems the only reason I need to create MyDoubleConverter is for variable substitution in my message. (There doesn't seem to be a default key for the value. For the class, it looks like you can use "type" although I didn't try it.)

**Application.java**

```
// This is an inner class within my Application
private static final class MyDoubleConverter extends DoubleConverter {
    private static final long serialVersionUID = 1L;

    @Override
    protected ConversionException newConversionException(String message, Object value,
            Locale locale) {
        final ConversionException newConversionException = super.newConversionException(message, value, locale);
        newConversionException.setVariable("value", value);
        return newConversionException;
    }
}
```

I tried implementing it as an anonymous class, but got a java.io.NotSerializableException. Once I moved this to an inner class it worked. If you know why, let me know!

The final piece of the puzzle is to put the message in your properties file. I put it in my BasePage.properties file.

**BasePage.properties**

```
IConverter.Double='${value}' is not a valid number
```

## Providing a custom converter for specific components

Sometimes, you want to couple a certain conversion without having to install a custom converter factory. This could be the case when you have conversions that differ from the default conversions of your application. It could also be useful when you create custom components that use a specific kind of conversion, that you want to be able to use without having to know about it in your application. In other words, the custom conversion will happen without you having to install a custom converter factory.

As an example, let us take a look at the form input example of wicket-examples. What we want to achieve is that we use our custom converter to convert from/to URL objects for a certain component.

This is the partial form:

```
public final class FormInputModel implements Serializable
{
  ...
  private URL urlProperty;
  ...

  /**
   * Gets the urlProperty.
   * @return urlProperty
   */
  public URL getUrlProperty()
  {
    return urlProperty;
  }

  /**
   * Sets the urlProperty.
   * @param urlProperty urlProperty
   */
  public void setUrlProperty(URL urlProperty)
  {
    this.urlProperty = urlProperty;
  }
  ...
}
```

Our converter:

```
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Locale;

import wicket.util.convert.ConversionException;
import wicket.util.convert.IConverter;

/**
 * Converts from and to URLs.
 *
 * @author Eelco Hillenius
 */
public class URLConverter implements IConverter
{
	/**
	 * Construct.
	 */
	public URLConverter()
	{
	}

	/**
	 * @see wicket.util.convert.IConverter#convert(java.lang.Object, java.lang.Class)
	 */
	public Object convert(Object value, Class c)
	{
		if (value == null)
		{
			return null;
		}

		if (c == URL.class)
		{
			if (value.getClass() == URL.class)
			{
				return value;
			}

			try
			{
				return new URL(value.toString());
			}
			catch (MalformedURLException e)
			{
				throw new ConversionException("'" + value + "' is not a valid URL");
			}
		}
		return value.toString();
	}

	/**
	 * @see wicket.util.convert.ILocalizable#setLocale(java.util.Locale)
	 */
	public void setLocale(Locale locale)
	{
	}

	/**
	 * @see wicket.util.convert.ILocalizable#getLocale()
	 */
	public Locale getLocale()
	{
		return Locale.getDefault();
	}
}
```

We can use this converter like this:

```
add(new TextField("urlProperty", URL.class)
{
  public IConverter getConverter()
  {
    return new URLConverter();
  }
});
```

The above textfield will now allways use the URLConverter instead of the application-wide converter. We don't have to provide the type parameter in TextField to make the conversion happening. However, the good thing about providing the type parameter is that any input is validated first (using the custom converter as well) before trying to update the model, so that if conversion fails, you'll get a feedback message instead of a stacktrace. It also forces the component to convert to the given type, even though Ognl might find another target type.

Note that we used the converter of the most generic type, `wicket.util.convert.IConverter`. If you use this type, you have to look at the requested target type (thus you code like: `if(c.equals(URL.class))`) yourself. While using `wicket.util.convert.Converter` is easier, as in that case you can couple `wicket.util.convert.ITypeConverter` s, using IConverter directly is transparent (easier to debug) and it forces to do only one type of conversion.