CSS API to support custom UI Controls

Overview

CSS is an integral part of JavaFX. By using the classes in this package in the manner prescribed in this document, CSS styling can be added to any Node. Making a property styleable or adding pseudo-class state in JavaFX requires some coding, but is fairly straight-forward.

Use Case

Say a contributor wished to develop a Watermark control which could overlay a region with copyright information. The Watermark API has a DoubleProperty that controls then angle at which the copyright text is displayed. The contributor wishes to make this copyrightAngle styleable through CSS. The contributor also wants to be able to modify some visual aspect of the Watermark once the copyright has been viewed; perhaps making the Watermark more transparent. The contributor wants to use a 'viewed' pseudo-class state to achieve this.

Making copyrightAngle styleable

First, the copyrightAngle is made styleable by having it extend one of the `StyleableProperty classes`; in this case, `StyleableDoubleProperty`. The code shown here follows the typical pattern. Notice that the only difference between a `DoubleProperty` and the `StyleableDoubleProperty` is the addition of the `getCssMetaData` method which links the copyrightAngle property to its corresponding `CssMetaData` instance.

```
private DoubleProperty copyrightAngle = new StyleableDoubleProperty(45d) {

/** Link this property with its CssMetaData */
@Override
public CssMetaData getCssMetaData() {
return COPYRIGHT_ANGLE;
}

@Override
public Object getBean() {
return Node.this;
}

@Override
public String getName() {
return "copyrightAngle";
}
};

public final void setCopyrightAngle(double value) {
copyrightAngle.set(value);
}
public final double getCopyrightAngle() {
return copyrightAngle.get();
}

public final DoubleProperty copyrightAngleProperty() {
return copyrightAngle;
}
```

The `CssMetaData` instance provides information about the CSS style and some methods that allow CSS to set the property's value. The convention is to instantiate the `CssMetaData` as a singleton (note that this example code is not thread safe, but is sufficient). Here, the CSS property name is "-my-copyright-angle" and the initial value of 45 degrees with the default value of the copyrightAngle property.

```
private static final CssMetaData<Watermark,Number> COPYRIGHT_ANGLE = new CssMetaData("-my-copyright-angle",
45d) {

public abstract boolean isSettable(Watermark node) {
return copyrightAngle == null || copyrightAngle.isBound() == false;
}

public abstract StyleableProperty<Number> getStyleableProperty(Watermark node) {
return (StyleableProperty)copyrightAngleProperty();
}
};
```

In order for the CSS engine to know about the styleable properties of Watermark, the methods `getClassCssMetaData()` and `getCssMetaData()` need to be implemented. The first is a static method that returns the `CssMetaData` of the Watermark class and of its super class; in Watermark's case, the super class is `Control`. The second method returns the same value but is implemented as an instance method so that it is not necessary to use reflection to call `getClassCssMetaData()`. These methods are called quite frequently; for efficiency, the List<CssMetaData> is only created once. The following code shows a typical implementation.

```
private static final List<CssMetaData> CSS_META_DATA;
static {
final List<CssMetaData> metaData = new ArrayList<CssMetaData>(Control.getClassCssMetaData());
Collections.addAll(metaData,
COPYRIGHT_ANGLE
);
CSS_META_DATA = Collections.unmodifiableList(metaData);

}

public static List<CssMetaData> getClassCssMetaData() {
return CSS_META_DATA;
}

@Override public List<CssMetaData> getCssMetaData() {
return getClassCssMetaData();
}
```

At this point, the copyrightAngle can be styled through CSS. For example, to display the copyright from lower left to upper right:

```
.watermark { -my-copyright-angle: -45; }
```

Making 'viewed' a pseudo-class

The 'viewed' pseudo-class will be implemented as a BooleanProperty. Although BooleanProperty is most commonly used for pseudo-class state, any property type can be a pseudo-class, even a Styleable*Property. When the 'viewed' property changes value, the code needs to notify CSS that the state has changed. The place to do this is in the `invalidated()` method of the property which calls the `pseudoClassStateChanged` method, passing it the `PseudoClass` of the pseudo-class that changed state and whether or not the pseudo-class is "active". Again, this code follows the typical pattern of implementing a property. Note that a "simple" property could be used here, but the invalidated method would still need to be overridden. Since the anonymous class will be created anyway, the few bytes needed for the bean and name will be saved by using BooleanPropertyBase instead of SimpleBooleanProperty.

```
public final void setViewed(boolean value) {
viewed.set(value);
}

public final boolean isViewed() {
return viewed.get();
}

public final BooleanProperty viewedProperty() {
return viewed;
}

private static final PseudoClass VIEWED_PSEUDO_CLASS = PseudoClass.getPseudoClass("viewed");

private BooleanProperty viewed = new BooleanPropertyBase(false) {

@Override
protected void invalidated() {
pseudoClassStateChanged(VIEWED_PSEUDO_CLASS, get());
}

@Override
public Object getBean() {
return Node.this;
}

@Override
public String getName() {
return "viewed";
}
};
```

The call to `pseudoClassStateChanged()` will cause CSS styles to be updated if the pseudo-class is used by selectors that match this Node.

With this framework in place, the following style can be used to make the watermark more transparent if it has been viewed.

```
.watermark:viewed { -fx-opacity: 30%; }
```

Design Goals

- The primary goal is to create an API that allows CSS styling to be applied to a JavaFX property and to support the use of pseudo-class state in the open-software environment.
- The API should support the following javafx.beans.property types: BooleanProperty, FloatProperty, DoubleProperty, IntegerProperty, LongProperty, and StringProperty. Support for ObjectProperty is desired but would, possibly, require additional hooks into the parser or public API to convert a parsed value to the parameterized type which may not be feasible at this time. However, the ability to style an ObjectProperty with a parameterized type such as Insets or Paint is essential. Likewise, ObjectProperty<MyEnumType> presents challenges, particularly in parsing and converting, that make support for this type of property impracticable at this time.
- A secondary goal is to make the API such that an IDEs whould be able to automatically generate much of the necessary code. It is not a goal, however, to create the IDE boilerplate.
- While it is possible to have classes other than instances of Node be styleable through CSS, this support requires an additional interface which exists in private implementation. Including this interface in the public API is not a goal.
- The current implementation of CSS in JavaFX is only a small portion of the W3C standards and is only partially compliant with those standards. It is beyond the scope of this API implementation to rectify differences between the W3C standards and the current JavaFX implementation.

Architecture

There are two main pieces to the styleable property architecture. First is a CssMetaData whose value can be represented syntactically in a .css file. A CssMetaData encapsulates the CSS property name, the type into which the string value is converted, and the default value of the property. Second is the JavaFX property to which the parsed CssMetaData value applies. Any JavaFX property that supports this styling is a StyleableProperty. A StyleableProperty also incorporates additional logic to ensure that values set by the user through calls to set methods are not overridden by styles in a user agent stylesheet.

There is a one-to-one correspondence between a CssMetaData and a StyleableProperty. A CssMetaData is scoped to a class whereas a StyleableProperty is an attribute of a class instance. Typically, a node will assume the CssMetaData of its ancestors. During CSS processing, the CSS engine iterates over a List<CssMetaData> (particular to the node) and looks up the parsed value of each CssMetaData in turn. If the CssMetaData has a parsed value, the parsed value is converted to the type of the StyleableProperty and the StyleableProperty is set.

Making a property styleable, then, consist of:

1. defining the javafx.beans.property as a StyleableProperty
2. creating a corresponding CssMetaData
3. ensuring the CssMetaData is returned in the List<CssMetaData>

Pseudo-class state support consists of notifying CSS of a pseudo-class state change via the `invalidated()` method of a property.

API

StyleableProperty

StyleableProperty is an interface that is implemented by various classes that extend from javafx.beans.property properties; for example,

```
public class StyleableBooleanProperty extends BooleanPropertyBase implements StyleableProperty<Boolean> {  }
```

.

StyleableProperty.java

```
interface StyleableProperty<T> extends WritableValue<T> {

/**
* This method is called from CSS code to set the value of the property.;
*/
void applyStyle(Origin origin, T value);

/**
* Tells the origin of the value of the property. This is needed to
* determine whether or not CSS can override the value.
*/
StyleOrigin getStyleOrigin();

/**
* Reflect back the CssMetaData that corresponds to this;
* <code>javafx.beans.property.StyleableProperty</code>
*/
CssMetaData getCssMetaData();

}
```

The `getCssMetaData` method is useful for getting from a `StyleableProperty` to the corresponding `CssMetaData`, which is useful for tooling and unit testing. For example, one can get the CssMetaData of the fillProperty of a Rectangle:

```
Rectangle rect = new Rectangle(50,50);
CssMetaData fillCssMetaData = ((StyleableProperty)rect.fillProperty()).getCssMetaData();
System.out.println("Use " + fillCssMetaData.getProperty() + " to style Rectangle fill");
```

There is an implementing classes for each of the javafx.beans.property.*PropertyBase types and for each of the javafx.beans.property.Simple*Property types. The developer should choose the simple variety unless there is some method that needs to be overridden (but even then, it is a matter of choice).

```
public class StyleableBooleanProperty extends BooleanPropertyBase implements StyleableProperty<Boolean>
public class StyleableFloatProperty extends FloatPropertyBase implements StyleableProperty<Float>
public class StyleableDoubleProperty extends DoublePropertyBase implements StyleableProperty<Double>
public class StyleableIntegerProperty extends IntegerPropertyBase implements StyleableProperty<Integer>
public class StyleableLongProperty extends LongPropertyBase implements StyleableProperty<Long>
public class StyleableStringProperty extends StringPropertyBase implements StyleableProperty<String>
~~public class StyleableObjectProperty<T> extends ObjectPropertyBase<T> implements StyleableProperty<T>~~
```

Each of the classes have a constructor taking a `CssMetaData` arg and another taking a `CssMetaData` arg and an initial value. For example, `StyleableBooleanProperty` has the following constructors:

```
/**
* The constructor of the {@code StyleableBooleanProperty}.
*/
public StyleableBooleanProperty(CssMetaData CssMetaData) {
super();
this.CssMetaData = CssMetaData;
}

/**
* The constructor of the {@code StyleableBooleanProperty}.
*
* @param CssMetaData
* the {@code CssMetaData} that corresponds to this {@code StyleableProperty}
* @param initialValue
* the initial value of the wrapped {@code Object}
*/
public StyleableBooleanProperty(CssMetaData CssMetaData, boolean initialValue) {
super(initialValue);
this.CssMetaData = CssMetaData;
}
```

```
public class SimpleStyleableBooleanProperty extends SimpleBooleanProperty implements StyleableProperty<Boolean>
public class SimpleStyleableFloatProperty extends SimpleFloatProperty implements StyleableProperty<Float>
public class SimpleStyleableDoubleProperty extends SimpleDoubleProperty implements StyleableProperty<Double>
public class SimpleStyleableIntegerProperty extends SimpleIntegerProperty implements StyleableProperty<Integer>
public class SimpleStyleableLongProperty extends SimpleLongProperty implements StyleableProperty<Long>
public class SimpleStyleableStringProperty extends SimpleStringProperty implements StyleableProperty<String>
public class SimpleStyleableObjectProperty<T> extends SimpleObjectProperty<T> implements StyleableProperty<T>
```

Each of these simple classes have a constructor taking an Object which is the property bean and a String which is the property name in combination with a `CssMetaData` arg and another taking a `CssMetaData` arg and an initial value. For example, `SimpleStyleableBooleanProperty` has the following constructors:

```
/**
* The constructor of the {@code SimpleStyleableBooleanProperty}.
*/
public SimpleStyleableBooleanProperty(Object bean, String name, CssMetaData CssMetaData) {
super(bean, name);
this.CssMetaData = CssMetaData;
}

/**
* The constructor of the {@code SimpleStyleableBooleanProperty}.
*
* @param CssMetaData
* the {@code CssMetaData} that corresponds to this {@code StyleableProperty}
* @param initialValue
* the initial value of the wrapped {@code Object}
*/
public SimpleStyleableBooleanProperty(Object bean, String name, CssMetaData CssMetaData, boolean initialValue) {
super(bean, name, initialValue);
this.CssMetaData = CssMetaData;
}

/**
* The constructor of the {@code SimpleStyleableBooleanProperty}.
*/
public SimpleStyleableBooleanProperty(CssMetaData CssMetaData) {
super();
this.CssMetaData = CssMetaData;
}

/**
* The constructor of the {@code SimpleStyleableBooleanProperty}.
*
* @param CssMetaData
* the {@code CssMetaData} that corresponds to this {@code StyleableProperty}
* @param initialValue
* the initial value of the wrapped {@code Object}
*/
public SimpleStyleableBooleanProperty(CssMetaData CssMetaData, boolean initialValue) {
super(initialValue);
this.CssMetaData = CssMetaData;
}
```

Issues:
In the future, `StyleableProperty` may need to incorporate support for attribute selectors and animations.

CssMetaData

`CssMetaData` encapsulates the data needed to lookup a value and apply that value to a `StyleableProperty`. This includes the CSS property name, the default property value, and a link back to the corresponding `StyleableProperty`. The class is abstract and it is necessary to implement two methods which are invoked from the CSS engine:

```
/**
* Check to see if the corresponding property on the given node is
* settable. This method is called before any styles are looked up for the
* given property. It is abstract so that the code can check if the property
* is settable without expanding the property. Generally, the property is
* settable if it is not null or is not bound.
*
* @param node The node on which the property value is being set
* @return true if the property can be set.
*/
public abstract boolean isSettable(N node);
/**
* Return the corresponding <code>StyleableProperty</code> for
* the given Node. Note that calling this method will cause the property
* to be expanded.
* @param node
* @return
*/
public abstract StyleableProperty<V> getStyleableProperty(N node);
```

The implementation of these methods is fairly consistent throughout the code. Notice the need to cast the return value of `getStyleableProperty` since `cornerProperty()` returns a `StringProperty`.

```
public abstract boolean isSettable(Watermark node) {
return corner == null || corner.isBound() == false;
}
public abstract StyleableProperty<String> getStyleableProperty(Watermark node) {
return (StyleableProperty)cornerProperty();
}
```

Note that `isSettable` should be implemented in a way that does not cause the expansion of the property. The `isSettable` check is performed before the CSS value is looked up. If the property is not settable, then no further CSS processing is done for that property (on any given pulse). The `getStyleableProperty` method is invoked only if there is a CSS value to apply. Thus, if the property is not settable or there is no CSS value, the property is not expanded[#1].

The CSS engine does not call `setValue` directly on the `StyleableProperty`. Rather, the code calls a set method on the `CssMetaData` which, in turn, calls `applyStyle` on the `StyleableProperty`. This level of indirection allows a `CssMetaData` to intercept the value before the calculated style value is applied (in other words, before `setValue` is called on the corresponding property). This is used primarily for `Number` based properties where the parameterized type is `Number` but the actual type might be `Integer` and so the value's `intValue()` method needs to be invoked.

The parameterization of `CssMetaData` is that it requires a `Node` since `Node` is the visible element of the scene graph. The `V` parameter is the type of the property's value, for example Boolean.

```
public abstract class CssMetaData<N extends Node, V> {

/**
* Check to see if the corresponding property on the given node is
* settable. This method is called before any styles are looked up for the
* given property. It is abstract so that the code can check if the property
* is settable without expanding the property. Generally, the property is
* settable if it is not null or is not bound.
*
* @param node The node on which the property value is being set
* @return true if the property can be set.
*/
public abstract boolean isSettable(N node);

/**
* Return the corresponding <code>StyleableProperty</code> for
* the given Node. Note that calling this method will cause the property
* to be expanded.
* @param node
* @return
*/
public abstract StyleableProperty<V> getStyleableProperty(N node);

/**
* Set the value of the corresponding property on the given Node.
```

```
* @param node The node on which the property value is being set
* @param value The value to which the property is set
*/
public void set(N node, V value, Origin origin) {
// details omitted, but this method ends up calling applyStyle in the StyleableProperty interface.
}

private final String property;
/**
* @return the CSS property name
*/
public final String getProperty() {
return property;
}

private final StyleConverter converter;
/**
* @return The CSS converter that handles conversion from a CSS value to a Java Object
*/
public final StyleConverter getConverter() {
return converter;
}

private final V initialValue;
/**
* The initial value of a CssMetaData corresponds to the default
* value of the StyleableProperty in code.
* For example, the default value of Shape.fill is Color.BLACK and the
* initialValue of Shape.CssMetaData.FILL is also Color.BLACK.
* <p>
* There may be exceptions to this, however. The initialValue may depend
* on the state of the Node. A ScrollBar has a default orientation of
* horizontal. If the ScrollBar is vertical, however, this method should
* return Orientation.VERTICAL. Otherwise, a vertical ScrollBar would be
* incorrectly set to a horizontal ScrollBar when the initial value is
* applied.
* @return The initial value of the property, possibly null
*/
public V getInitialValue(N node) {
return initialValue;
}

private final List<CssMetaData> subProperties;
/**
* The sub-properties refers to the constituent properties of this property,
* if any. For example, "-fx-font-weight" is sub-property of "-fx-font".
*/
public final List<CssMetaData> getSubProperties() {
return subProperties;
}

private final boolean inherits;
/**
* If true, the value of this property is the same as
* the parent's computed value of this property.
* @default false
* @see <a href="http://www.w3.org/TR/css3-cascade/#inheritance">CSS Inheritance</a>
*/
public final boolean isInherits() {
return inherits;
}

/**
* Construct a CssMetaData with the given parameters and no sub-properties.
* @param property the CSS property
* @param converter the StyleConverter used to convert the CSS parsed value to a Java object.
* @param initalValue the default value of the corresponding property which may be null
* @param inherits true if this property uses CSS inheritance
* @param subProperties the sub-properties of this property. For example,
* the -fx-font property has the sub-properties -fx-font-family,
* -fx-font-size, -fx-font-weight, and -fx-font-style.
```

```
*/
protected CssMetaData(
final String property,
final StyleConverter converter,
final V initialValue,
boolean inherits,
final List<CssMetaData> subProperties) {

this.property = property;
this.initialValue = initialValue;
this.inherits = inherits;
this.subProperties = subProperties != null ? Collections.unmodifiableList(subProperties) : null;

if (this.property == null)
throw new IllegalArgumentException("property cannot be null");
}

/**
* Construct a CssMetaData with the given parameters and no sub-properties.
* @param property the CSS property
* @param converter the StyleConverter used to convert the CSS parsed value to a Java object.
* @param initalValue the default value of the corresponding property which may be null
* @param inherits true if this property uses CSS inheritance
*/
protected CssMetaData(
final String property,
final StyleConverter converter,
final V initialValue,
boolean inherits) {
this(property, initialValue, inherits, null);
}

/**
* Construct a CssMetaData with the given parameters, inherit set to
* false and no sub-properties.
* @param property the CSS property
* @param converter the StyleConverter used to convert the CSS parsed value to a Java object.
* @param initalValue the default value of the corresponding property which may be null
*/
protected CssMetaData(
final String property,
final StyleConverter converter,
final V initialValue) {
this(property, initialValue, false, null);
}

/**
* Construct a CssMetaData with the given parameters, initialValue is
* null, inherit is set to false, and no sub-properties.
* @param property the CSS property
* @param converter the StyleConverter used to convert the CSS parsed value to a Java object.
*/
protected CssMetaData(
final String property,
final StyleConverter converter) {
this(property, false, null);
}
}
```

StyleConverter

A StyleConverter takes a value from the parser (a ParsedValue) and converts it to the corresponding java type. So, for example, a ColorConverter would convert a ParsedValue representing a Color into a Color. If possible, the parser performs the conversion. But this is not always possible such as when converting an em size to an absolute pixel value.

StyleConverter and ParsedValue are tightly coupled and there are a number of StyleConverter implementations. The StyleConverter class contains static methods that return instances of these implementations.

```
/**
* Converter converts {@code ParsedValue<F,T>} from type F to type T. the
```

```
* {@link CssMetaData} API requires a {@code StyleConverter} which is used
* when computing a value for the {@see StyleableProperty}. There are
* a number of predefined converters which are accessible by the static
* methods of this class.
* @see ParsedValue
* @see StyleableProperty
*/
public class StyleConverter<F, T> {

/**
* Convert from the parsed CSS value to the target property type.
*
* @param value The {@link ParsedValue} to convert
* @param font The {@link Font} to use when converting a
* <a href="http://www.w3.org/TR/css3-values/#relative-lengths">relative</a>
* value.
*/
public T convert(ParsedValue<F,T> value, Font font) {
return (T) value.getValue();
}

/**
* @return A {@code StyleConverter} that converts &quot;true&quot; or &quot;false&quot; to {@code Boolean}
* @see Boolean#valueOf(java.lang.String)
*/
public static StyleConverter getBooleanConverter() {
return BooleanConverter.getInstance();
}

/**
* @return A {@code StyleConverter} that converts a String
* representation of a web color to a {@code Color}
* @see Color#web(java.lang.String)
*/
public static StyleConverter getColorConverter() {
return ColorConverter.getInstance();
}

/**
* @return A {@code StyleConverter} that converts a parsed representation
* of an {@code Effect} to an {@code Effect}
* @see Effect
*/
public static StyleConverter getEffectConverter() {
return EffectConverter.getInstance();
}

/**
* @return A {@code StyleConverter} that converts a String representation
* of an {@code Enum} to an {@code Enum}
* @see Enum#valueOf(java.lang.Class, java.lang.String)
*/
public static StyleConverter getEnumConverter(Class enumClass) {
// TODO: reuse EnumConverter instances
return new EnumConverter(enumClass);
}

/**
* @return A {@code StyleConverter} that converts a parsed representation
* of a {@code Font} to an {@code Font}.
* @see Font#font(java.lang.String, javafx.scene.text.FontWeight, javafx.scene.text.FontPosture, double)
*/
public static StyleConverter getFontConverter() {
return FontConverter.getInstance();
}

/**
* @return A {@code StyleConverter} that converts a [&lt;length&gt; |
* &lt;percentage&gt;]{1,4} to an {@code Insets}.
*/
public static StyleConverter getInsetsConverter() {
```

```
return InsetsConverter.getInstance();
}

/**
* @return A {@code StyleConverter} that converts a parsed representation
* of a {@code Paint} to a {@code Paint}.
*/
public static StyleConverter getPaintConverter() {
return PaintConverter.getInstance();
}

/**
* CSS length and number values are parsed into a Size object that is
* converted to a Number before the value is applied. If the property is
* a {@code Number} type other than Double, the
* {@link CssMetaData#set(javafx.scene.Node, java.lang.Object, javafx.css.Origin) set}
* method of ({@code CssMetaData} can be over-ridden to convert the Number
* to the correct type. For example, if the property is an {@code IntegerProperty}:
* <code><pre>
* {@literal @}Override public void set(MyNode node, Number value, Origin origin) {
* if (value != null) {
* super.set(node, value.intValue(), origin);
* } else {
* super.set(node, value, origin);
* }
* }
* </pre></code>
* @return A {@code StyleConverter} that converts a parsed representation
* of a CSS length or number value to a {@code Number} that is an instance
* of {@code Double}.
*/
public static StyleConverter getSizeConverter() {
return SizeConverter.getInstance();
}

/**
* A converter for quoted strings which may have embedded unicode characters.
* @return A {@code StyleConverter} that converts a representation of a
* CSS string value to a {@code String}.
*/
public static StyleConverter getStringConverter() {
return StringConverter.getInstance();
}

/**
* A converter for URL strings.
* @return A {@code StyleConverter} that converts a representation of a
* CSS URL value to a {@code String}.
*/
public static StyleConverter getUrlConverter() {
return URLConverter.getInstance();
}


}
```

ParsedValue

This class must exist in the public API for use in the StyleConverter API. The parser creates ParsedValue instances.

```
public class ParsedValue<V, T> {

/**
* The CSS property value as created by the parser.
*/
final protected V value;

/**
* @return The CSS property value as created by the parser, which may be null
* or otherwise incomprehensible.
*/
public final V getValue() { return value; }

/**
* The {@code StyleConverter} which converts the parsed value to
* the type of the {@link StyleableProperty}. This may be null, in which
* case {@link #convert(javafx.scene.text.Font) convert}
* will return {@link #getValue() getValue()}
*/
final protected StyleConverter<V, T> converter;

/**
* A {@code StyleConverter} converts the parsed value to
* the type of the {@link StyleableProperty}. If the {@code StyleConverter}
* is null, {@link #convert(javafx.scene.text.Font)}
* will return {@link #getValue()}
* @return The {@code StyleConverter} which converts the parsed value to
* the type of the {@link StyleableProperty}. May return null.
*/
public final StyleConverter<V, T> getConverter() { return converter; }

/**
* Convenience method for calling
* {@link StyleConverter#convert(javafx.css.ParsedValue, javafx.scene.text.Font) convert}
* on this {@code ParsedValue}.
* @param font The {@link Font} to use when converting a
* <a href="http://www.w3.org/TR/css3-values/#relative-lengths">relative</a>
* value.
* @return The value converted to the type of the {@link StyleableProperty}
* @see #getConverter()
*/
public T convert(Font font) {
return (T)((converter != null) ? converter.convert(this, font) : value);
}

/**
* Create an instance of ParsedValue where the value type V is converted to
* the target type T using the given converter.
* If {@code converter} is null, then it is assumed that the type of value
* {@code V} and the type of target {@code T} are the same and
* do not need converted.
*/
protected ParsedValue(V value, StyleConverter<V, T> converter) {
this.value = value;
this.converter = converter;
}

}
```

StyleOrigin

```
/**
* Enumeration of the possible source or origin of a stylesheet and styles.
*/
public enum StyleOrigin {
/** The stylesheet is a user-agent stylesheet */
USER_AGENT,
/** The value of a property was set by the user through a call to a set method */
USER,
/** The stylesheet is an external file */
AUTHOR,
/** The style is from the Node via setStyle */
INLINE
}
```

Additional Node API.

List<CssMetaData> getCssMetaData()
static List<CssMetaData> getClassCssMetaData()

These methods return a List of CssMetaData supported by the node. The list should include not only the properties of this node, but the properties of the node's super-class(es) as well. This method is called frequently and, by convention, it returns a static list. Since this static list may be used by other node classes, the convention is to provide a public static method that returns the static list: public static List<CssMetaData> getClassCssMetaData(). The getCssMetaData() method is used by the CSS engine to avoid reflection. A typical implementation would be:

```
private static class StyleableProperties {

private static final CssMetaData<Watermark, String> CORNER = new CssMetaData<Watermark, String>("-my-corner",
"lower-right") {
public abstract boolean isSettable(Watermark node) {
return corner == null || corner.isBound() == false;
}
public abstract StyleableProperty<String> getStyleableProperty(Watermark node) {
return (StyleableProperty)cornerProperty();
}
}
private static final List<CssMetaData> META_DATA;
static {
final List<CssMetaData> data = new ArrayList<CssMetaData>();
Collections.addAll(data,
Control.getClassCssMetaData(),
CORNER
);
META_DATA = Collections.unmodifiableList(data);
}
}

public static List<CssMetaData> getClassCssMetaData() {
return StyleableProperties.META_DATA;
}

public static List<CssMetaData> getCssMetaDataMetaData() {
return getClassCssMetaData();
}
```

Pseudo-class

The pattern for handling pseudo-class state is to override the invalidated method of the property in order to invoke `pseudoClassStateChanged` which sets or removes a PseudoClass from a Set<PseudoClass>. The addition or removal of a PseudoClass from this Set may cause CSS to apply new styles to the node, depending on whether or not the pseudo-class is used in a matching selector.

The node's current pseudo-class state can be gotten by calling `getPseudoClassStates()`, which returns an unmodifiable Set<PseudoClass>.

PseudoClass

PseudoClass is an immutable object that represents one pseudo-class state. A pseudo-class state is unique according to the string value of the pseudo-class that might appear in a CSS file. In other words, there is only one PseudoClass for "hover", even if the "hover" pseudo-class is found in more than one class. Care must be exercised when choosing names for pseudo-classes so they do not conflict with other pseudo-classes.

Typical usage is:

```
private static final PseudoClass MAGIC_PSEUDO_CLASS = PseudoClass.getPseudoClass("xyzzy");

BooleanProperty magic = new BooleanPropertyBase(false) {
@Override public void invalidated() {
pseudoClassStateChanged(MAGIC_PSEUDO_CLASS, get());
}

@Override public Object getBean() {
return MyControl.this;
}

@Override public String getName() {
return "magic";
}
};
```

Node#pseudoClassStateChanged(PseudoClass, boolean)

```
/**
* Used to specify that a pseudo-class of this Node has changed. If the
* pseudo-class is used in a CSS selector that matches this Node, CSS will
* be reapplied. Typically, this method is called from the {@code invalidated}
* method of a property that is used as a pseudo-class. For example:
* <code><pre>
* private static final PseudoClass MY_PSEUDO_CLASS_STATE = PseudoClass.getPseudoClass("my-state");
*
* BooleanProperty myPseudoClassState = new BooleanPropertyBase(false) {
*
* @Override public void invalidated() {
* pseudoClassStateChanged(MY_PSEUDO_CLASS_STATE, get());
* }
*
* @Override public Object getBean() {
* return MyControl.this;
* }
*
* @Override public String getName() {
* return "myPseudoClassState";
* }
* };
* </pre><code>
* @param pseudoClass the pseudo-class that has changed state
* @param active whether or not the state is active
*/
public final void pseudoClassStateChanged(PseudoClass pseudoClass, boolean active) {
// if (active) add pseudoClass to the Set<PseudoClass> that is this Node's pseudo-class state
// otherwise, remove pseudoClass from the Set<PseudoClass> that is this Node's pseudo-class state
// The implementation of the Set<PseudoClass> will cause a CSS update if the
// added or removed pseudo-class is used in a style that matches this node.
}
```

Node#getPseudoClassStates()

```
/**
* @return An unmodifiable Set of active pseudo-class states
*/
public final Set<PseudoClass> getPseudoClassStates() {
...
}
```

Oh, snap!

In some cases, pseudo-class state may need to be initialized. For example, a ScrollBar can be horizontal or vertical. The default orientation of a ScrollBar is Orientation.HORIZONTAL and it is necessary to initialize the pseudo-class state to ensure the correct styles are selected. This is done simply by calling the pseudoClassStateChanged method from the ScrollBar constructor:

```
pseudoClassStateChanged(HORIZONTAL_PSEUDOCLASS_STATE, true);
```

Calling pseudoClassStateChanged(HORIZONTAL_PSEUDOCLASS_STATE, true) does not set the VERTICAL_PSEUDOCLASS_STATE to false. In the orientation property's invalidated method, pseudoClassStateChanged is called for each orientation state:

```
@Override protected void invalidated() {
final boolean isVertical = (get() == Orientation.VERTICAL);
pseudoClassStateChanged(HORIZONTAL_PSEUDOCLASS_STATE, !isVertical);
pseudoClassStateChanged(VERTICAL_PSEUDOCLASS_STATE, isVertical);
}
```

Node#getPseudoClassStates returns an unmodifiable list which can be useful for seeing what pseudo-class states are active. For example,

```
System.out.println(tabPane.getPseudoClassStates().toString());
```

gives the output

```
TabPane@34a004b8[styleClass=root tab-pane]
pseudoClassStates=[top],
triggerStates=[focused, top, right, bottom, left]
```

The pseudoClassStates are the active states and the trigger states are the states that might, in some combination, cause CSS to be updated.

Footnotes

1. Further explanation of property expansion. Consider something like opacityProperty which starts out null. If nothing ever touches it, it stays null. CSS wants to know if opacity can be set before it goes off looking for styles. If it can't be set, then it is skipped. If it can, and no styles for it are found, then opacityProperty should still be null, otherwise every time CSS touches a property it would be expanded - not a good thing. So the isSettable check basically says, the property is settable if it is null or isn't bound. The only time the property is expanded is if there is actually a style for it.