IBM ILOG JViews TGO V8.6

Styling

Table of contents

Introducing cascading style sheets....9

Cascading style sheets....10

Getting started with JViews TGO style sheets....11

Writing a style sheet....12
Declarations....14
Specializing CSS rules....15
The priority of CSS rules....17
Debugging a style sheet....18

The CSS specification....19

CSS Syntax....20
Applying CSS to Java objects....21
The CSS engine in JViews TGO....23
JViews TGO Functions....28
Divergences from CSS2....38

How to represent a business object....41

Business class properties....42
Attribute properties....44

Retrieving the value of a property....45

Using custom pseudo-classes....46

Using Cascading Style Sheets....49

Using global settings....52

Customizing network and equipment nodes..........55
Representing nodes as business objects..........56
Adding new decorations to predefined business objects..........57
Advanced customization of nodes..........65
Customizing network and equipment links..........67
Customizing tree nodes..........69
Representing tree nodes as graphic objects or icons with labels..........70
Advanced customization of tree nodes..........76
Improving the performance of predefined business objects (tree component)..........78
Customizing table cells..........79
Representing table cells as business objects or labels with optional icons..........80
Advanced customization of table cells..........86
Improving the performance of table cell rendering..........88
Customizing table column headers and rows..........89
Customizing the label of a business object..........93
Customizing the label in table cells..........98
Changing the font of all labels..........100
Customizing tooltips..........102
Customizing object and alarm states of predefined business objects..........107
Overview of customizing the states of predefined business objects..........109
Secondary states and information window properties..........110
Alarm configuration properties..........113
Alarm balloon configuration properties..........117
Alarm count configuration properties..........120
SNMP system info configuration properties..........122
Changing the icon color of predefined business objects..........123
Customizing the icon of business objects..........125
Adding a user-defined business attribute to the system window..........126
Changing the background color of all columns in a table..........128
Displaying the same attribute with different representations..........131
Customizing node and link layouts..........133
Customizing link label layout..........135
Customizing the selection border in the network and equipment..........139
Customizing selection in a table or a tree..........140
Customizing the expansion of business objects..........141
Customizing network elements..........143

Representing network elements....144

Customizing network element types....147

Representing a network element type....149
Customizing existing network element types....150
Creating network element types from images and customizing them....157
Using the imagecolortuner application to configure the renderer factory....163
Customizing network element types from SVG graphics....164
Extending the class IltNEBaseRenderer....166
Localizing network element types....173

Customizing network element functions....174

Customizing network element families....176

Customizing different aspects of network elements....178

Customizing user-defined business objects....181

Representing business objects....182

Customizing user-defined network nodes....183

Customizing a node shape....184
Customizing the color of a node shape with paint styles....189
Customizing the border of a node shape....191
Customizing a node icon....192
Customizing a node label....193
Automatic resizing for a node shape with an icon in it....194

Customizing user-defined network links....195

Customizing a link....196
Customizing various aspects of links....200
Customizing a link label....211

Customizing tooltips of user-defined business objects....212

Customizing links....213

Links....215

Representing links....216

Customizing link representations....217

Changing the representation of individual links....220

Customizing the link information cluster....222

Customizing link media....223

Customizing link technology....225

Customizing various aspects of links....228

Customizing link tiny types....230

Customizing link sets....231
Customizing link bundles....232
Customizing link set and link bundle tiny types....234
Customizing groups....235
Groups....236
Representing groups and attributes....238
Customizing group representations....239
Customizing various aspects of groups....248
Customizing subnetworks....253
Representing subnetworks....254
Customizing the representation of subnetworks....255
Customizing shelves and cards....259
Representing physical telecommunication equipment....261
Representing shelves....262
Customizing shelf representations....263
Customizing various aspects of shelves....265
Representing and customizing card carriers....270
Customizing various aspects of card carriers....271
Representing and customizing cards....274
Customizing various aspects of card carriers....275
Representing and customizing ports....280
Customizing various aspects of ports....281
Representing and customizing LEDs....285
Customizing various aspects of LEDs....287
Customizing BTS....293
Representing and customizing BTS....294
Customizing BTS antennas....296
Customizing various aspects of BTS antennas....298
Customizing alarms....301
Representing alarms....302
Customizing various aspects of alarms....303
Customizing off-page connectors....305

Representing off-page connectors....306
Customizing existing off-page connector types....307
Customizing new off-page connector types....311
Customizing other aspects of off-page connectors....315

Customizing object states....317

Customizing the object representation based on states....319
Representing and customizing state information....321
OSI states....323
Bellcore states....325
SNMP states....327
SONET states....329
Miscellaneous states....331
Performance states....332
SAN states....334
Alarms....336
Traps....338
Customizing passive devices....340
Customizing the OSI state system....341
Customizing the Bellcore state system....343
Customizing the SNMP state system....345
SNMP primary and secondary states....346
Customizing SNMP secondary states....348
Creating a new attribute in the System group....353
Customizing the SONET state system....355
Customizing SONET states....356
Customizing SONET protection states....359
Customizing the Miscellaneous state system....361
Customizing the Performance State System....363
Creating new Performance secondary states....364
Customizing Performance secondary states....366
Customizing the SAN state system....371
Creating new SAN secondary states....372
Customizing SAN Secondary States....374
Customizing alarm severities....379
Customizing alarm count attributes....386
Customizing trap types....392
Customizing the secondary state icons....396

Index..397

Introducing cascading style sheets

Describes the Cascading Style Sheet (CSS) mechanism.

In this section

Cascading style sheets
Explains what style sheets are used for and describes the Cascading Style Sheets (CSS) mechanism.

Getting started with JViews TGO style sheets
Describes how style sheets function in JViews TGO.

The CSS specification
Contains reference information on CSS and explains how CSS concepts are used in JViews TGO.

How to represent a business object
Describes how business objects are represented by business class properties and attribute properties:

Retrieving the value of a property
Describes how to retrieve property values for a user-defined graphic renderer or graphic class.

Using custom pseudo-classes
Describes how to create and register your own pseudo-classes for the objects displayed in graphic components.

Cascading style sheets

Style sheets contain collections of graphic settings, such as color, font, or icon, which are used to render objects and associated attributes in graphic components. Cascading Style Sheets (CSS) provide a powerful mechanism for customizing HTML rendering in a web browser. The CSS specification originates from the World Wide Web Consortium (W3C) and has the status of a W3C Recommendation.

The CSS mechanism is a great improvement over the `.Xdefault` resource mechanism of the X Window System. The basic idea remains the same: matching a pattern and setting resource values. CSS are intended for HTML rendering, matching HTML tags, and setting style values. XML is also a CSS target, especially in the context of the Scalable Vector Graphic (SVG) specification of W3C.

In ILOG JViews TGO, the CSS level 2 (CSS2) Recommendation is transposed to the Java™ language and used to set JavaBean™ properties in accordance with the Java object hierarchy and state.

JViews TGO graphic components use CSS chiefly for the following purposes:

♦ To define how each business object is to be displayed in the graphic components.

♦ To define a setting specific to one graphic component, but generalized within that component, such as the background color of a view.

JViews TGO supplies a large number of predefined ready-to-use CSS properties that apply to both predefined and custom business classes, objects, and attributes of these objects. Graphic components use these properties to render data. JViews TGO provides default property values for creating a default look for data appearing across different graphic components.

Getting started with JViews TGO style sheets

Describes how style sheets function in JViews TGO.

In this section

Writing a style sheet
Explains the syntax of style sheets through a simple example.

Declarations
Explains what declarations are with a simple example.

Specializing CSS rules
Explains the syntax of CSS rules with simple examples.

The priority of CSS rules
Explains how specificity determines the priority of CSS rules with a simple example.

Debugging a style sheet
Describes common problems in debugging a style sheet and the use of a debug mask.

Writing a style sheet

Here is a basic example to start with, which is intended to help you understand quickly how to write style sheets in JViews TGO.

How to write an easy style sheet

A CSS usually starts with the configuration of the graphic component. The graphic component configuration includes the configuration of the view, the interactor, the adapter and of some other elements in relation to the graphic component. For example, to set a background color in a tree component, you can write:

```
Tree {
  view: true;
}
View {
  background: orange;
}
```

The configuration of each graphic component is specified in a CSS rule, which is identified by the name of the graphic component:

♦ Table

♦ Tree

♦ Network

♦ Equipment

Each graphic component has a set of properties that can be customized through CSS.

For information on the properties in each graphic component, see the following topics:

♦ Table: Table component

♦ Tree: Tree component

♦ Network: Network component

♦ Equipment: Equipment component

Once a graphic component is configured, you can start to set the characteristics of the graphic representation of objects that are to be displayed in the component. In JViews TGO, all objects from the model have the CSS type `object`.

The following CSS extract shows how to configure objects of the business class `Alarm`. The complete version of this configuration is in the tutorial in ***<installdir>*** **/tutorials/gettingStarted**.

Normally, the JViews TGO tree component displays objects of a custom class as tree nodes indicated by a default icon and a label. You can specify a different behavior by customizing the graphic representation of these objects: you can set a specific icon and define that the label of the tree node has the value of the `identifier` attribute. A special character, the commercial at sign (`@`), informs JViews TGO that it must get the value from the business model. The `@` character is used as a prefix of the name of the attribute from the model.

```
object.Alarm {
    label: @identifier;
}
```

When you have configured the style sheet, you can load it into the various graphic components with the method `setStyleSheets`:

```
//Load the tree component configuration
String[] css = new String[] { "tree.css" };
try {
  treeComponent.setStyleSheets(css);
} catch (Exception e) {
  e.printStackTrace();
}
```

The style sheet is dynamically interpreted, that is, you can load a new CSS configuration in a graphic component without recompiling or relaunching the application.

Declarations

In CSS, the term declaration is used to describe the statements placed between braces ({}). Declarations represent property settings. They are used in JViews TGO like a JavaBean™ for customizing graphic objects. Typically, the left part of a declaration contains the name of a property of a graphic object and the right part contains the value set for the property. Sometimes, the left part of a declaration is not used to refer to a property of a graphic object. For example, `class` is a reserved word that represents the class name of the graphic representation.

JViews TGO defines a set of properties that can be applied to user-defined and predefined business objects to customize their graphic representation.

You can also define your own graphic representation by declaring the `class` property with your JavaBean class and setting it in the CSS rule.

How to define your own graphic representation

```
object.Alarm {
    class: 'javax.swing.JButton';
    text: @identifier;
    background: blue;
    foreground: yellow;
}
```

The CSS rule defined in this example states that objects from the business class `Alarm` are represented by `JButton` instances that are configured with the properties `text`, `background`, and `foreground`.

Specializing CSS rules

A CSS rule consists of a selector followed by one or more declarations contained within braces ({}). Up until now, selectors have only been used to distinguish CSS types and CSS classes.

CSS Class

In JViews TGO, the CSS class corresponds to the business model class.

How to match a business class

Within a selector, the syntax for matching a business class is:

```
object."ilog.tgo.model.IltObject" {
    label: @name;
}
```

How to match a business attribute

In the table component, table cells have a specific CSS class that is made up of the business class and attribute (column) names. The syntax for a business attribute is the following:

```
object."Alarm/identifier" {
    label: @identifier;
}
```

Matching object identifiers

Identifier matching in the selector is used to match a specific business object in the model. It is expressed in the format `#id`. When you use this type of selector, you can customize each individual business object through the style sheet. However, the benefits of factorization through declarations are lost.

How to match object identifiers

```
#RJLed0 {
    foreground: yellow;
}
```

Add this rule to the style sheet to set the foreground color of the object `RJLed0` to yellow.

Matching attribute values

The syntax for matching attribute values is expressed in the format `[att=val]`. This syntax allows you to write patterns in the style sheet that are close to the objects in the business model and to test the attribute against a value.

The following example adjusts the background color of the table cells in the column `identifier` according to the value of the attribute `perceivedSeverity`.

How to vary table cell background color depending on attribute value

```
object."Alarm/identifier"[perceivedSeverity=0] {
    labelBackground: '#FFFFFF';
}
```

The priority of CSS rules

Declarations that are not overridden are still valid. This mechanism allows you to write default declarations for common objects and to refine customization with rules that apply to more specialized objects in the model by overriding the default declarations.

In the following example, business objects of class `Alarm` have a label with a yellow background when the attribute `perceivedSeverity` is `0`, and a white background when this attribute has a different value.

How to customize the representation of an object depending on a specific value

```
object.Alarm {
    label: @identifier;
    labelBackground: white;
}

object.Alarm[perceivedSeverity=0] {
    labelBackground: yellow;
}
```

Debugging a style sheet

When styling with CSS, one of the most common problems concerns the order of priority. A rule can be assumed to have a greater or lesser priority than is actually the case. Then, unsuitable declarations are applied. With many rules, it can be difficult to determine which declarations apply to which conditions. Understanding the priority of CSS rules is the most difficult aspect of a large style sheet.

Another common problem is syntax errors in the declarations. If an identifier on the left does not refer to a valid property, it will be silently ignored. If the value on the right is invalid, it may also be ignored by the target object. In both cases, the declaration has no visible effect.

Yet another common problem concerns changes in properties. An event marking a new value may trigger a new rule. When the object in the model reverts to its previous state, the previous set of rules applies. Therefore, the previous set of rules must contain declarations that undo the changes in the properties of the graphic object correctly. For example, if a node switches to an alarm state, a CSS rule will create an alarm decoration. When the alarm state is resolved, the decoration should be canceled by a declaration in the normal set of rules.

You can specify a `styleSheetDebugMask` property in the graphic components. This flag has several levels that help in debugging problems. The style sheet debug mask can be set directly in the graphic component with the method `setStyleSheetDebugMask(int)` or it can be customized in the style sheet itself.

```
StyleSheet {
    styleSheetDebugMask: "DECL_MASK|DECL_VALUE_MASK";
}
```

See the interface `IlvStylable` for more details on the available debug levels.

The CSS specification

Contains reference information on CSS and explains how CSS concepts are used in JViews TGO.

In this section

CSS Syntax
Describes the main syntax elements of CSS.

Applying CSS to Java objects
Describes how CSS is applied to style Java objects instead of XML documents.

The CSS engine in JViews TGO
Describes the role of the CSS engine with respect to graphic objects.

JViews TGO Functions
Describes the set of functions provided for use directly in your CSS files.

Divergences from CSS2
Describes the way CSS has been adapted to accommodate Java objects.

CSS Syntax

CSS syntax is described more fully in Using CSS syntax in the style sheet

The main elements of syntax are:

Style rule
A CSS document, or style sheet, consists of a set of style rules. Each style rule starts with a selector and is followed by a declaration.

Selector
A selector is composed of one or more simple selectors. A simple selector is made of minimal building blocks. When two or more simple selectors are aggregated into a selector, they are separated by combinators. A combinator is a single character; extra spaces are ignored.

Declaration
Declarations are property-value pairs that are enclosed in braces (`{}`). The separator is a colon (:). Each declaration is terminated by a semicolon (;). The property represents a predefined graphic attribute and the value is a literal, with its type dependent on the property. All property-value pairs are `String`.

Priority
Priority depends on specificity. Specificity is computed as three numbers, `a-b-c` (in a number system with a large base). The number of ID building blocks in the selector gives the first number `a`, the number of classes and attributes gives `b`, and the number of element types gives `c`.

Cascading
Cascading consists in supplying several sources for the style: the browser, the user, and the document in HTML environments. Each source is weighted in relation to the others, with document style taking precedence over user style, which takes precedence over browser style when the specificity is the same. CSS can also make use of internal cascading, when it imports other style sheets by referring to a URL.

Inheritance
Inheritance of declarations occurs when matched declarations are sorted according to the priority of the rules and declarations are merged. Higher priority settings override lower ones as described in *The priority of CSS rules*.

Applying CSS to Java objects

The CSS selector is designed to match HTML or XML documents. It can also be used to match a hierarchy of Java™ objects accessible through a model interface. The declarations are then sorted for the objects in the model and are used depending on the application that controls the CSS engine. In JViews TGO, declarations create and customize one graphic object for each object in the model.

The input model represents the kernel of the CSS for Java engine. It provides these important categories of information:

- ♦ The tree structure of objects, which is exploited by selector transitions
- ♦ Object type, ID, and CSS classes
- ♦ Attribute value that matches the selector attribute of the same name

The declarations part represents property settings on a target object. The target object depends on the way the CSS engine is used. In JViews TGO, the target object is the rendering object associated with the object in the model. JViews TGO provides a default graphic representation for user-defined business classes and predefined business classes. This graphic representation has a set of properties that are used to customize its appearance.

Instead of using the default graphic representation supplied by JViews TGO, you can define your own rendering object, an `IlvGraphic` or a `JComponent`. In this case, the declarations change property values on the graphic object that corresponds to the object matched in the model.

How to customize the graphic representation

```
object."test.Vehicle"[model=sports] {
    icon: "sports-car.gif";
}
```

This example matches the object of the class `test.Vehicle` with the property `model` (which has the value `sports`) and fixes the property `icon` of the graphic object associated with this object to `sports-car.gif`. (The association of the object with the graphic object is defined elsewhere.)

Note: The enclosing double quotes around `test.Vehicle` are used so that the dot is not interpreted as a CSS class pattern; that is, an object of type `test` with CSS class `Vehicle`.

Property matching can be used to add dynamic behavior. An event that changes attribute values occurring on the model can activate the CSS engine to establish new property values.

How to add dynamic behavior through property matching

```
object.computer[state=down] {
    foreground: gray;
}
```

This example changes the foreground color whenever an object of CSS class `computer` has the value of the property `state` set to `down`.

The CSS engine in JViews TGO

In JViews TGO, the CSS engine is responsible for creating and customizing graphic objects and renderers when the data is loaded. At run time, the engine customizes the graphic objects according to changes in the model.

The left side of a declaration usually represents a JavaBean™ property of the graphic property. The right side is a literal. If the literal requires type conversion, the method `setAsText()` is invoked on the property editor associated with the JavaBean property.

Class property

The `class` property name is a reserved keyword that indicates the class name of the generated graphic object. JViews TGO provides a predefined representation for the objects in all graphic components, which means that the `class` property is optional. It can be used when you want to replace the predefined representation.

How to use the Class property

The right side of a class declaration could contain:

- ♦ The class name, loaded with the application context class loader.

  For example:

```
object {
    class: ilog.views.sdm.graphic.IlvGeneralNode;
    foreground: red;
}
```

- ♦ A pathname to a file. In fact, the class name is forwarded to the JavaBeans library with the method `java.beans.Beans.instantiate()`, so a serialized JavaBean is suitable. For example:

```
object {
    class: data.beans.gauge;
    foreground: red;
}
```

  The JavaBeans documentation states: “When using the beanName as a serialized object name, we convert the given beanName to a resource pathname and add a trailing ‘.ser’ suffix. We then try to load a serialized object from that resource.”

  In the example given, the method `Beans.instantiate()` would try to read a serialized object from the resource `data/beans/gauge.ser`.

In the network and equipment components, the class declaration is applied only when a creation request occurs. When the model state changes, graphic components are customized by applying only new declarations from the matching rules in the CSS. The class declaration is ignored. To change the class, the object in the model must be removed and then added again.

Model indirection

The right side of a declaration contains a literal that is converted dynamically through a property editor. If the literal is prefixed by `@`, the remainder of the string is treated as a model attribute name. The declaration expects the attribute value of the object from the model.

How to refer to attribute values of model objects

```
object."ilog.tgo.model.IltObject" {
    label: @name;
}
```

The `label` property is set to the value of the attribute named `name` in the model.

Besides the standard model attributes, JViews TGO also provides the following attributes that you can use when customizing objects and graphic components:

♦ `@__ID`

  Returns the object identifier. You can use it to customize objects as illustrated below:

  ```
  object {
    toolTipText: @__ID;
  }
  ```

♦ `@__ADAPTER`

  Returns the component adapter. You can use it to customize graphic components adapters. For more information about component customization using CSS, see the following topics:

  Configuring a network component through a CSS file

  Configuring an equipment component through CSS

  Configuring the tree component through a CSS file

  Configuring the table component through a CSS file

Resolving URLs

Sometimes declaration values are URLs relative to the style sheet location. A special construct, standard in CSS level 2, allows you to create a URL from the base URL of the current style sheet. For example, the following declaration extends the path of the current style sheet URL with `images/icon.gif`.

How to extend the path of the current style sheet

```
imageURL: url(images/icon.gif);
```

This feature is very useful for creating style sheets with images located relative to the style sheet itself, since the URL remains valid even when the CSS is cascaded or imported elsewhere.

CSS recursion

You may want to specify a Java object as the value of a declaration. A simple convention allows you to recurse in the style sheet; that is, to define a new Java object which has the same style sheet, but is unrelated to the current model.

Prefix the value with `@#` to create a new JavaBean dynamically.

How to Create a New JavaBean Dynamically

```
object."Alarm/creationTime" {
    toolTipText: "@#tooltipFormatBean";
    label: @creationTime;
}
Subobject#toolTipFormatBean {
    class:"ilog.cpl.util.text.IlpSimpleDataFormat";
    pattern: "HH:mm:ss z";
}
```

The `@#` operator extends the current data model by adding a dummy model object as the child of the current object. The object ID of the dummy object is the remainder of the string, beyond the `@#` operator. The type of the dummy object is `Subobject`. The dummy object inherits CSS classes and attributes from its parent.

The CSS engine creates and customizes a new subobject according to the declarations it finds for the dummy object. This means, in particular, that the Java class of the subobject is determined by the value of the `class` property. The newly created subobject becomes the value of the `@#` expression. In the declarations for the subobject, attribute references through the `@` operator refer to the attributes of the parent object.

Once the subobject is completed, the previous model is restored, so that normal processing is resumed.

In the example, an `IlpSimpleDateFormat` object is created, with the pattern property set to `HH:mm:ss z`, and is assigned to the `toolTipText` property of the object.

There are two refinements to the `@#ID` operator:

♦ `@=ID`

  Using `@=ID` instead of `@#ID` shares the instance. The first time the declaration is resolved, the object is created as with the `@#` operator. But for all subsequent access to the same value, `@=ID` returns the same instance, the one created the first time and without applying the CSS rules.

♦ `@+ID`

  Using `@+ID` instead of `@#ID` avoids unnecessary creation of objects. The CSS engine first checks whether the property value corresponding to the declaration is defined and not `null`. If the property value is defined, this current value is customized directly using the rules for the `#ID` operator, deliberately ignoring any `class` declaration. If it is not defined, the operator behaves exactly as with the `@#` operator. In this case, the operator creates the property value only when required and customizes it in all cases.

The need for these refinements arises from a performance issue. The `@#` operator creates a new object each time a declaration is resolved. Usually, a declaration is applied when properties change. Under certain circumstances, the creation of objects may lead to expensive processing. These optional mechanisms minimize the creation of objects.

CSS expressions and functions

The values of CSS declarations can be literals, `@#` constructions, or attributes from the model (prefixed by `@`). In addition, you can also declare a value with `@|` which considers the rest of the value as an expression.

The syntax of the expression after the `@|` prefix is close to Java syntax. Expressions can be, for example, arithmetic (`int`, `long`, `float`, or `double`), `Boolean`, or `String`.

Syntax of Expression Values

Expression Syntax	Meaning
@|3+2*5	13
@|true&&(true||!true)	true
@|foo+bar	"foobar"

Expressions can also refer to attributes in the model. The syntax used is the normal one.

Syntax of Expressions Referring to Attributes in the Model

Expression Syntax	Meaning
@|@speed/100+@drift	1/100 of the value of `speed` plus value of `drift`, where `speed` and `drift` are attributes of the current object.
'@|"name is: " + @name'	`"name is: Bob"`, if the value of the current object attribute `name` is `Bob`. Note the use of double quotes to retain the space characters.

The usual functions are accepted:

- ♦ abs()
- ♦ acos()
- ♦ asin()
- ♦ atan()
- ♦ ceil()
- ♦ cos()
- ♦ exp()
- ♦ floor()
- ♦ log()
- ♦ pi

- rint()
- round()
- sin()
- sqrt()
- tan()

For example:

```
@|3+sin(pi/2)
```

which results in the value `4`.

CSS also supports functions as part of expressions.

JViews TGO Functions

Functions for direct use in CSS files

Functions for direct use in CSS files

Function Name	Description	Class Name	Usage
`image`	Retrieves an image from the Image Repository service registered in your application context.	`IlpImageFunction`	**Parameter**: image location **Example**: `@|image("ilog/tgo/ilt_busy.png")`
`resource`	Retrieves a resource value from a given `ResourceBundle`.	`IlpResourceFunction`	**Parameters**: `ResourceBundle` name (mandatory) Message identifier (mandatory) Default message value (optional). Returned if the message requested was not found in the given resource bundle. **Example**: `@|resource("ilog.tgo.messages.JTGOMessages", "ilog.tgo.Operational_State")`
`valuemap`	Retrieves a value from an `IlpValueMap` object that corresponds to the specified key.	`IlpValueMapFunction`	**Parameters**: `IlpValueMap` instance Object key **Example**: `@|valuemap(@#valuemap, @severity);`
`format`	Applies a format to the given values.	`IlpFormatFunction`	**Parameters**: `java.text.Format` instance Arguments of the `Format`

Function Name	Description	Class Name	Usage
			Example: `@|format (@#formatBean, @attribute)`
`blinkingcolor`	Creates a blinking color.	`IltBlinkingColorFunction`	**Parameters**: on color off color on period off period **Example**: `@|blinkingcolor (black, white,1000, 500)`

`pattern`	Creates a pattern description, `IlPattern`, which can be used to customize the representation of JViews TGO predefined business objects.	`IltPatternFunction`	**Parameter**:
			
			Pattern type, which can have one of the following values:
			
			`Grid`, `SkewGrid`, `Dots`, `ThinHatching`.
			
			It can also generate patterns defined in `IlvPattern`, for example, `LIGHT_VERTICAL`.
			
			Depending on the pattern type, other arguments can be passed to configure the pattern instance.
			
			`Grid` patterns accept two integer arguments: `xPeriod` and `yPeriod`.
			
			`SkewGrid` patterns accept two integer arguments: `uPeriod` and `vPeriod`.
			
			Examples:
			
			`@|pattern("Grid", 3, 2)`
			
			`@|pattern ("SkewGrid", 2, 2)`

Function Name	Description	Class Name	Usage
			`@|pattern ("LIGHT_VERTICAL")`

`acknowledgedAlarmCount`	Returns the number of acknowledged alarms for a given business object.	`IltAcknowledgedAlarmCountFunction`	**Parameter**:
			
			One of the following possibilities:
			
			Default: all raw severities or traps
			
			Impact: all impact alarm severities
			
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			
			Examples:
			
			`label: '@|acknowledgedAlarmCount ("Impact")';`
			
			`label: '@|acknowledgedAlarmCount ("Impact.MajorHigh") ';`

`acknowledgedAlarmSummary`	Returns a summary of acknowledged alarms for a given business object. The returned value is a `String` listing the number of acknowledged alarms for each chosen severity.	`IltAcknowledgedAlarmSummaryFunction`	**Parameter**:
			
			One of the following possibilities:
			
			Default: all raw severities or traps
			
			Impact: all impact alarm severities
			
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			
			Examples:
			
			`label: '@|acknowledgedAlarmSummary ("Impact")';`
			
			`label: '@|acknowledgedAlarmSummary ("Raw.Major")';`

`alarmCount`	Returns the number of outstanding alarms for a given business object.	`IltAlarmCountFunction`	**Parameter**:

Function Name	Description	Class Name	Usage

			One of the following possibilities:
			
			Default: all raw severities or traps
			
			Impact: all impact alarm severities
			
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			
			Examples:
			
			`label: '@|alarmCount ("Impact")';`
			
			`label: '@|alarmCount ("Raw.Major")';`

`alarmSummary`	Returns the summary of new and acknowledged alarms for a given business object.	`IltAlarmSummaryFunction`	**Parameter**:
			
			One of the following possibilities:
			
			Default: all raw severities or traps
			
			Impact: all impact alarm severities
			
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			
			Examples:
			
			`label: '@|alarmSummary()'; ///Equivalent to "Default"`
			
			`label: '@|alarmSummary ("Impact")'; /// Impact alarms`
			
			`label: '@|alarmSummary ("Raw.Major")'; /// Consider only raw major alarms`

`highestAcknowledgedSeverity`	Returns the highest severity of acknowledged	`IltHighestAcknowledgedSeverityFunction`	**Parameter**:

Function Name	Description	Class Name	Usage

	alarms for a given business object.		One of the following possibilities:
			
			Default: all raw alarm severities or traps
			
			Impact: all impact alarm severities
			
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			
			Examples:
			
			`label: '@|highestAcknowledgedSeverity()'; ///Equivalent to "Default"`
			
			`label: '@|highestAcknowledgedSeverity("Impact")'; /// Impact alarms`
			
			`label: '@|highestAcknowledgedSeverity("Raw.Major")'; /// Consider only raw major alarms`

`highestNewSeverity`	Returns the highest severity of new alarms for a given business object.	`IltHighestNewSeverityFunction`	**Parameter**:
			
			One of the following possibilities:
			
			Default: all raw alarm severities or traps
			
			Impact: all impact alarm severities
			
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			
			Examples:
			
			`label: '@|highestNewSeverity()'; ///Equivalent to "Default"`
			
			`label: '@|highestNewSeverity`

Function Name	Description	Class Name	Usage

			`("Impact")'; /// Impact alarms`
			
			`label: '@|highestNewSeverity ("Raw.Major")'; /// Consider only raw major alarms`

`highestSeverity`	Returns the highest severity of outstanding alarms for a given business object.	`IltHighestSeverityFunction`	**Parameter**:
			
			One of the following possibilities:
			
			Default: all raw alarm severities or traps
			
			Impact: all impact alarm severities
			
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			
			Examples:
			
			`label: '@|highestSeverity() '; ///Equivalent to "Default"`
			
			`label: '@|highestSeverity ("Impact")'; /// Impact alarms`
			
			`label: '@|highestSeverity ("Raw.Major")'; /// Consider only raw major alarms`

`newAlarmCount`	Returns the number of new alarms for a given business object.	`IltNewAlarmCountFunction`	**Parameter**: One of the following possibilities: Default: all raw alarm severities or traps Impact: all impact alarm severities Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`

Function Name	Description	Class Name	Usage

			Examples:
			`label: '@|newAlarmCount()'; ///Equivalent to "Default"`
			`label: '@|newAlarmCount("Impact")'; /// Impact alarms`
			`label: '@|newAlarmCount("Impact.MajorLow")'; ///Consider only raw major alarms`

`newAlarmSummary`	Returns the summary of new alarms for a given business object.	`IltNewAlarmSummaryFunction`	**Parameter**:
			One of the following possibilities:
			Default: all raw alarm severities or traps
			Impact: all impact alarm severities
			Alarm severity name: for example, `Raw.Major`, `Impact.MajorHigh`
			Examples:
			`label: '@|newAlarmSummary()'; ///Equivalent to "Default"`
			`label: '@|newAlarmSummary("Impact")'; /// Impact alarms`
			`label: '@|newAlarmSummary("Impact.MajorLow")'; ///Consider only raw major alarms`

`primaryStateSummary`	Returns the summary of the primary state	`IltPrimaryStateSummaryFunction`	**Example**:

Function Name	Description	Class Name	Usage
	information for a given business object.		`label: '@|primaryStateSummary()';`
`secondaryStateSummary`	Returns the summary of the secondary state information for a given business object.	`IltSecondaryStateSummaryFunction`	**Example**: `label: '@|secondaryStateSummary()';`
`settings`	Returns an `IltSettings`.	`IltSettingsFunction`	**Parameter**: Setting key **Example**: `icon: '@|settings("Link.Media.Fiber.Icon")';`

`severityColor`	Returns the color corresponding to a given alarm severity.	`IltSeverityColorFunction`	**Parameter**:
			Alarm severity
			Examples:
			`labelBackgroundColor: '@|severityColor(@|highestNewSeverity())';`
			`labelBackgroundColor: '@|severityColor("Raw.Major")';`

`severityBrightColor`	Returns the bright color corresponding to a given alarm severity.	`IltSeverityBrightColorFunction`	**Parameter**: Alarm severity **Example**: `alarmBrightColor: '@|severityBrightColor(@|highestNewSeverity())';`
`severityDarkColor`	Returns the dark color corresponding to a given alarm severity.	`IltSeverityDarkColorFunction`	**Parameter**: Alarm severity **Example**: `alarmDarkColor: '@|severityDarkColor`

Function Name	Description	Class Name	Usage
			`(@|highestNewSeverity())';`
`severityIcon`	Returns the icon corresponding to a given alarm severity.	`IltSeverityIconFunction`	**Parameter**: `IltAlarmSeverity` or the String representation of an `IltAlarmSeverity`. **Example**: `alarmIcon: '@|severityIcon("Raw.Major")';` `alarmIcon: '@|severityIcon(@|highestSeverity())';` `alarmIcon: '@|severityIcon(@|highestSeverity("Impact"))';`
`tinyImage`	Returns an image displaying the tiny representation of a predefined business object.	`IltTinyImageFunction`	**Example**: `icon : '@|tinyImage()';`

How to create new CSS functions

JViews TGO allows you to create and register new functions to be used in CSS files to customize the representation of your business objects. These functions should implement the interface `IlpCSSFunction`. This interface defines the following methods:

- `getName()`

  Returns the name of the function that is used to identify the function in the CSS files.

- `getDelimiters()`

  Returns the delimiters that are used to identify the parameters of the function; for example, comma (`,`).

- `returnDelimitersAsToken()`

  Indicates whether the delimiters will also be returned as tokens.

- `call(java.lang.Object[], java.lang.Class, ilog.cpl.service.IlpContext, ilog.cpl.graphic.IlpGraphicView, ilog.cpl.model.IlpRepresentationObject, ilog.cpl.model.IlpAttribute)`

  This method is the core of the function, where the value will be computed and returned.

The signature of the main method is:

```
public Object call (Object[] args,
                    Class type,
                    IlpContext appc,
                    IlpGraphicView view,
                    IlpRepresentationObject ro,
                    IlpAttribute attribute);
```

When a function is evaluated, the parameters are first resolved as subexpressions. Then, the final values of the parameters are passed to the `args` array.

The parameter type is the expected type of the function when it is known. A null value is possible. The implementation should be careful to return an object of the appropriate type. Otherwise, a simple conversion is applied, if conversion is possible, that is, between primitive types or to a `String`.

The other parameters provide information about the application context, graphic view, representation object, and attribute at the time when the call is made. Stateless expressions do not need these parameters.

If an error occurs during a call, an exception will be reported and the current property setting will be canceled.

The following application is provided as part of the JViews TGO demonstration software: ***<installdir>*** **/samples/framework/datasource-explorer2.** It shows how to implement and register a new function. See file ByteFunction.java.

Functions are registered in the CSS file with one of the following properties:

♦ `functionList`: lists the functions as a comma-separated list of function names.

♦ `functions`: an indexed property of function names. Allows you to specify the list of functions according to indices. With this indexed property, you can register functions in a modular way in different CSS files.

How to register a function in a CSS file

```
StyleSheet {
    functionList: "test.function.FirstFunction,test.function.SecondFunction";
}
```

or

```
StyleSheet {
    functions[0]: "test.function.FirstFunction";
    functions[1]: "test.function.SecondFunction";
}
```

Divergences from CSS2

Java objects are not HTML documents. The differences lead to an adaptation of the CSS, so that its power can be fully exploited. The CSS2 syntax is retained. Therefore, a CSS editor can still be used to create the style sheet.

Cascading

Cascading is explicit. The API provides a means of cascading the style sheets. The `!important` and `inherit` tags are not supported. They have not been implemented for the sake of simplicity.

Pseudo-classes and pseudo-elements

Pseudo-classes are minimal building blocks which match model objects according to an external context. The syntax is like a CSS class, but uses a colon (:) instead of a dot (.). For example, `:link-visited` matches a link element only if it is already visited. Only the browser can resolve this pseudoclass at run time.

The pseudo-classes are fully implemented and are used by all JViews TGO renderers. JViews TGO recognizes the pseudo-classes `:selected`, `:focus`, and `:expanded` by default.

A pseudo-class has the same specificity as a CSS class.

Pseudo-elements are metaclasses like pseudo-classes, but match document structure instead of the browser state; for example, `:first-child`.

The CSS2 predefined pseudoelements and pseudo-classes (`:link`, `:hover`, and so forth) are not implemented: they have no meaning in Java™.

Attribute matching

Each attribute value must be converted to `String`. This conversion is done using the Type converter specified in your application context.

The attribute pattern in CSS2 checks only the presence `[att]`, equality `[att=val]`, and inclusion `[att~=val]` of strings. The `|=` operator is disabled. In Java, numeric comparators >, >=, <>, <=, < have been added, with the usual semantics.

Operators available in the attribute selectors

Operator	Meaning	Applicable To
A	present	strings
A=val	equals	strings
A~val	not equals	strings
A~=val	contains the word	strings
A==val	equals	numbers
A<>val	not equals	numbers
A<val	less than	numbers
A<=val	less than or equals	numbers
A>val	greater than	numbers
A>=val	greater than or equals	numbers

Syntax enhancement

CSS for Java requires the use of quotation marks when a token contains special characters such as dot (.), colon (:), commercial at sign (@), hash sign (#), space ( ), and so on. Quotes can be used almost everywhere, in particular to delimit a declaration value, an element type, or a CSS class with reserved characters. The closing semicolon (;) is optional.

Null value

Sometimes it makes sense to specify a null value in a declaration. By convention, `null` is a zero-length string `''` or `""`.

How to specify a null value

```
object {
    labelBackground: '';
}
```

The notation `''` is also used to denote a null array for properties that expect an array of values.

Empty string

The null syntax does not distinguish the ability to write an empty string in the style sheets. If an empty string is required, it is easy to create it dynamically.

How to create an empty string

```
object {
    label: @#emptyString;
}

Subobject#emptyString {
    class: 'java.lang.String';
}
```

Note: You can use the sharing mechanism to avoid creating several strings. The @= construct creates the empty string the first time only and reuses the same instance for all other occurrences of @=emptyString.

How to represent a business object

Describes how business objects are represented by business class properties and attribute properties:

In this section

Business class properties
Describes the types of business class and the inheritance and override mechanisms.

Attribute properties
Describes the use of attribute properties in the table component.

Business class properties

The representation of each instance of a given business object class is based on the declarations defined in the business object class selector. See *The CSS specification* for more information on CSS syntax and selectors.

JViews TGO has two categories of business object classes:

♦ Predefined

  Predefined classes are provided by JViews TGO and oriented towards telecommunications. These classes include a predefined set of attributes and a corresponding default representation.

♦ User-defined

  User-defined classes are defined by you according to your business model. These classes have no predefined set of attributes and the default graphic representation is basic. You will probably want to define your own graphic settings.

JViews TGO style sheets provide a built-in inheritance mechanism based on the business class hierarchy. This inheritance mechanism allows objects of a given business class to inherit settings from their parent business class. The advantage of the inheritance mechanism is that you can have a unified look-and-feel for all subclasses of a given class. If you define a graphic setting in a base class selector, this setting will be applied to all subclasses. For example, to apply a setting to all the predefined business classes, you must define the setting in the selector object `ilog.tgo.model.IltObject`. The overriding mechanism allows you to define exceptions in subclasses.

In the following example, the label is defined as invisible for all `IltObject` instances, except for instances of `IltNetworkElement`.

How to override an inherited setting

```
object."ilog.tgo.model.IltObject" {
    labelVisible: false;
}
object."ilog.tgo.model.IltNetworkElement" {
    labelVisible: true;
}
```

JViews TGO provides default property values for all predefined business object classes. These default property values ensure that all graphic components have an attractive display, which underpins the JViews TGO look-and-feel. This look-and-feel follows the Telecommunications Management Network (TMN) standard. The look-and-feel can be modified by overriding default property values. See the appropriate sections in this documenation for a complete list of all properties that can be used to customize the JViews TGO look-and-feel for predefined business objects.

When you work with user-defined business classes, their default graphic representation is basic. Therefore, you must define your own look by defining your own default values for these properties. The default property values for user-defined business classes are set to give a very simple look-and-feel in all graphic components. You can customize properties for user-defined business classes using the business class hierarchy.

How to customize properties for user-defined business classes

```
object.MyBusinessClass {
    label: @myNameAttribute;
}
```

If you want to customize the representation of *all* business classes, regardless of whether they are predefined or user-defined business classes, use the selector based on the *object* type.

How to customize the representation of all business classes

```
object {
    foreground: gray;
}
```

Attribute properties

Attribute properties are mainly used in the table component. For example, each cell in a table represents one attribute of a business object. Each attribute has its own associated set of properties that represent the graphic settings used by the table cell renderer to render the attribute value.

Attribute declarations are set using a CSS selector that is based on type `object` and a CSS class formed from the business class name and the attribute name.

How to customize attributes

```
object."ilog.tgo.model.IltObject/name" {
    label: @name;
}
```

JViews TGO provides default property values for all predefined business attributes. The use of these property values provides an esthetic display in a table, which corresponds to the JViews TGO look-and-feel.

Each attribute of the user-defined classes has an associated set of properties with default values. Attribute properties are also organized in a hierarchy. If an attribute is inherited from a parent class, its attribute property will inherit from the attribute property defined in the parent class. For example, the attribute property of `IltNetworkElement` inherits from the attribute property of `IltObject`.

How to use the inheritance mechanism for attributes

```
object."ilog.tgo.model.IltObject/name" {
    labelFont: "arial-plain-10";
}
object."ilog.tgo.model.IltNetworkElement/name" {
    labelFont: "arial-plain-12";
}
```

Retrieving the value of a property

When you use CSS and implement your own graphic renderer or graphic class, you might be interested in the values of certain graphic properties.

How to retrieve the value of a property

Property values can be retrieved with the class `IlpGraphicRendererContext`. This class provides the following method:

- `public Object getPropertyValue (String property,IlpGraphicView view,IlpRepresentationObject)`

  Returns the value of a property defined for the given representation object in the given view. This method applies to all graphic components.

```
object."Alarm" {
    label: @identifier;
    iconVisible: true;
}
```

- `public Object getPropertyValue (String property,IlpGraphicView view,IlpRepresentationObject ro,IlpAttribute attribute)`

  Returns the value of a property defined for a specific attribute in a representation object. This method applies to objects displayed in the table, network, or equipment components. It returns the properties declared using the `object` type selector.

```
object."Alarm/creationTime" {
    label: '@|format(@#labelFormat,@creationTime)';
    toolTipText: '@|format(@#toolTipFormat,@creationTime)';
}
```

- `public Object getPropertyValue (String property,IlpGraphicView view,IlpClass bclass,IlpAttribute attribute)`

  Returns the value of a property defined for a specific attribute in a business class. This method applies to business attributes displayed in the table component only. It returns the configuration used to represent the columns of the table. It mainly returns the properties declared using the `attribute` type selector.

```
attribute."Alarm/creationTime" {
    label: "Created";
    toolTipText: "Creation time";
    preferredWidth: 130;
}
```

Using custom pseudo-classes

JViews TGO allows you to create and register your own pseudo-classes for the objects displayed in graphic components. For more information about pseudo-classes, see *Pseudo-classes and pseudo-elements*.

All JViews TGO graphic components allow you to add and remove pseudo-classes dynamically for the objects represented in their graphic view. Pseudo-classes allow you to specify CSS properties according to the graphic view context. In the tree, network and equipment components, you can use the methods available in the corresponding graphic view to register and unregister pseudo-classes for a given representation object. For example:

♦ `public void addPseudoClass (IlpObject bo, String pseudo)`: adds a given pseudo-class to the business object.

♦ `public void removePseudoClass (IlpObject bo, String pseudo)`: removes a given pseudo-class from the business object.

These methods are available in each JViews TGO component: `IlpNetwork`, `IlpEquipment` and `IlpTree`.

In the table component, you can register and unregister pseudo-classes for each table cell by using the following methods of the `IlpTable`:

♦ `public void addPseudoClass (IlpObject bo, IlpAttribute a, String pseudo)`

♦ `public void removePseudoClass (IlpObject bo, IlpAttribute a, String pseudo)`

The following example shows you how to specify a new pseudo-class in the network component to highlight an object from the business class `ServiceManagedObject`:

```
object."ServiceManagedObject" {
        foreground: black;
        background: #00000000;
}
object."ServiceManagedObject":highlight {
        background: yellow;
}
```

This style sheet extract illustrates two selectors used to define the representation of objects of the class `ServiceManagedObject` in their normal state and with the pseudo-class **highlight**.

In your application, when you want to highlight the objects, you just have to add the given pseudo-class to the business object. For example:

```
IlpObject bo = dataSource.getObject("NE1");
networkComponent.addPseudoClass(bo, "highlight");
```

This example retrieves the business object identified as `NE1` and highlights its graphic representation by adding a pseudo-class. The graphic representation of the object will be recomputed according to the new pseudo-classes and will match the corresponding selectors in your style sheets.

To revert to the object representation normal state, you just have to remove the previously added pseudo-class as follows:

```
view.removePseudoClass(bo, "highlight");
```

JViews TGO automatically handles pseudo-classes for selection and focus management. All you have to do is define your CSS selectors with the pseudo-classes "`selected`" and "`focus`."

Using Cascading Style Sheets

Shows you how to use Cascading Style Sheets (CSS) to customize an application in the various graphic components. Each use case contains a style sheet extract, which can be loaded into the graphic component using the method `setStyleSheets`.

In this section

Using global settings
Describes the use of global settings to define the look and feel of the graphic components in an application.

Customizing network and equipment nodes
Describes how to customize the nodes in a network or equipment component.

Advanced customization of nodes
Describes how to use other classes for more complex representations of nodes.

Customizing network and equipment links
Describes how to customize the links in a network or equipment component.

Customizing tree nodes
Describes the graphical representation of tree nodes and the ways of customizing them.

Customizing table cells
Describes the graphical representation of table cells and the ways of customizing them.

Customizing table column headers and rows
Describes the graphical representation of table column headers and rows and the ways of customizing them

Customizing the label of a business object
Describes how to customize the labels of your business objects in the network component, the equipment component, and the tree component.

Customizing the label in table cells
Describes how to customize the label that represents an attribute in the table component.

Changing the font of all labels
Describes how to configure and change the font of all the labels in your application (all graphic components).

Customizing tooltips
Describes how to customize the tooltips used to describe your business objects (tree component, table component, network component, and equipment component).

Customizing object and alarm states of predefined business objects
Describes how to customize the object and alarm states of all the predefined business objects.

Customizing the icon of business objects
Describes how to customize the icon used to represent your business objects (tree component and table component).

Adding a user-defined business attribute to the system window
Describes how to add an attribute to the system window (network component and equipment component).

Changing the background color of all columns in a table
Describes how to change the background color of the columns in a table based on an attribute (table component).

Displaying the same attribute with different representations
Describes how to add several columns to a table in which the same attribute is represented in different ways (table component).

Customizing node and link layouts
Describes how to customize the layout of nodes and links in a network.

Customizing link label layout
Describes how to customize the link label layout in a network and in an equipment component.

Customizing the selection border in the network and equipment
Describes how to customize the selection border of your business objects (network component and equipment component).

Customizing selection in a table or a tree
Describes how to customize the selection of your business objects in a (table component and tree component).

Customizing the expansion of business objects

Describes how to customize the expansion of an object in the tree component, the network component, and the equipment component.

Using global settings

While CSS properties allow you to configure objects for a graphic component, global settings allow you to define a common look and feel for all the graphic components in the application. Global settings are stored in and managed by the class `ilog.tgo.resource.IltSettings`. The `IltSettings` class stores a predefined look and feel at start up.

There are two ways to set global settings: through the API or through CSS.

How to set global settings through the API

You can assign new global settings or retrieve existing ones by directly calling the static methods `IltSettings.SetValue(Object, Object)` or `IltSettings.GetValue(Object)`.

```
IltSettings.SetValue("Alarm.Impact.CriticalHigh.Description", "Critical High")
;
```

The setting shown customizes the description text for all Critical High Impact alarms.

```
String description = (String)
  IltSettings.GetValue("Alarm.Impact.CriticalHigh.Description");
```

You can retrieve the value of the setting using the `GetValue` method.

How to set global settings through CSS

Global settings defined in CSS allow you to create global state objects, alarm objects or telecom objects in the system and refer to them by their names when you customize them.

To do so, you need first to create a CSS file containing all the global settings.

```
Settings {
   alarm: true;
}

Alarm {
   impactSeverities[0]: @+impactSeverity0;
}
Subobject#impactSeverity0 {
  class: 'ilog.tgo.model.IltAlarm.ImpactSeverity';
  name: "Alarm.Impact.InformationalHigh";
  severity: 220;
}

setting."ilog.tgo.model.IltAlarm.ImpactSeverity"[name="Alarm.Impact.Information
alHigh"] {
  description: "Informational High";
}
```

This example shows how to create an impact severity object, then add it to the state system. The creation is enabled by the `alarm: true` declaration. Next, objects with impact severites of the type created can be customized with a new description message.

Once the global CSS settings file is created, an instance of `ilog.tgo.resource.IltSettings`, which serves as the global settings root object, is created. The CSS settings file is then assgined to the root object using the `setStyleSheets` method:

```
IlpContext context = IltSystem.GetDefaultContext();
IltSettings settingsRoot = new IltSettings(context);
try{
   settingsRoot.setStyleSheets( new String[] {"global.css"} );
}
catch (IlvStylingException e) {
   e.printStackTrace();
}
```

The default settings can be restored by passing `null` to the `setStyleSheets` method:

```
IlpContext context = IltSystem.GetDefaultContext();
IltSettings settingsRoot = new IltSettings(context);
try{
   settingsRoot.setStyleSheets( null );
}
catch (IlvStylingException e) {
   e.printStackTrace();
}
```

Note: New objects previously created through the global settings CSS file remain in the system.

Customizing network and equipment nodes

Describes how to customize the nodes in a network or equipment component.

In this section

Representing nodes as business objects
Describes the graphical representation of a node and how to customize it.

Adding new decorations to predefined business objects
Describes how to add new decorations to a predefined business object, and how to customize various aspects of the decorations.

Representing nodes as business objects

Nodes are graphically represented in a network or equipment component according to the properties of their business object class.

ILOG JViews TGO provides predefined business objects and supports user-defined business objects. In both cases, the node representation is fully customizable through CSS.

You can customize the node representation by changing the properties of the business object class that graphically represents the node.

Predefined business objects

To customize the graphic representation of predefined business objects, you can change the properties of their predefined graphical representation.

For details, refer to the section corresponding to the type of object:

- ♦ *Customizing network elements*
- ♦ *Customizing groups*
- ♦ *Customizing subnetworks*
- ♦ *Customizing shelves and cards*
- ♦ *Customizing BTS*
- ♦ *Customizing off-page connectors*

User-defined business objects

To customize the default graphic representation of user-defined business objects in a network or equipment component, refer to *Customizing user-defined business objects*.

Adding new decorations to predefined business objects

To extend the predefined graphical representation of predefined business objects, you may also add new decorations.

How to add new decorations to network and equipment nodes

This use case illustrates how you can extend the predefined business object representation by adding new decorations. It applies to the network and equipment components.

The network and equipment node representation of predefined business objects cannot be replaced, but you can extend it by adding new decorations.

The following properties are available to add a new decoration:

- `children`: An indexed property each element of which is an `IlvGraphic`. Each `IlvGraphic` is a new decoration that will be added to the graphic representation.
- `constraints`: An indexed property each element of which is an attachment constraint. The attachment constraint indicates how the new property is attached to the object base. It is an instance of `ilog.views.graphic.composite.layout.IlvAttachmentConstraint`.

The following application is provided as part of the JViews TGO demonstration software at ***<installdir>*** **/samples/network/decoration**.

It illustrates how to extend a predefined business object representation with a new decoration. In this use case, the following customization is notable:

```
object."SampleNetworkElement" {
  children[0]: '';
  constraints[0]: '';
  children[1]: '';
  constraints[1]: '';
}

object."SampleNetworkElement"[comments] {
  children[0]: @+commentIcon;
  constraints[0]: @+commentIconConstraint;
}
```

The selectors in this extract indicate that a new decoration will be added to the object representation when the attribute `comments` is set in the object. When the attribute is set, an icon is attached to the bottom left side of the base as illustrated by the following CSS extract.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// This selector creates an icon decoration.
//
// Here we show the use of built-in properties:
//  * ILPATTRIBUTE
//  * ILPDECORATIONNAME
//
```

```
// 1. To associate a decoration to an IlpAttribute
// specify the 'ILPATTRIBUTE' property as illustrated
// below.
//
// This property is used to associate a graphic decoration
// to an attribute in the business model.
//
// The ILPATTRIBUTE property type is String. Its value
// should be an attribute name.
//
// Once this association is done, tooltips and interactors
// can be defined in CSS for the business attribute
// and will be displayed/triggered when the event
// occurs in the decoration graphic.
//
// 2. To associate a decoration to a name specify
// the 'ILPDECORATIONNAME' property as illustrated
// below.
//
// The ILPDECORATIONNAME property type is a String.
// Its value should be a ilog.tgo.graphic.IltGraphicElementName
// value.
//
// Once this association is done, it will be possible
// to customize the layer policy and zoom policy to
// take into account your new decorations.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Subobject#commentIcon {
  class: 'ilog.views.graphic.IlvIcon';
  image: '@|image("commentIcon.png")';
  ILPATTRIBUTE: "comments";
  ILPDECORATIONNAME: "Comments";
}

Subobject#commentIconConstraint {
  class: 'ilog.views.graphic.composite.layout.IlvAttachmentConstraint';
  hotSpot: BottomLeft;
  anchor: BottomLeft;
}
```

New decorations can be added to all predefined business objects, either nodes or links. The attachment locations vary depending on whether the object is a node or a link.

In the case of nodes, new decorations can be added to the following attachment locations, relative to the object base (see `IlvAttachmentLocation`):

♦ `TopLeft`, `TopCenter`, `TopRight`

♦ `LeftCenter`

♦ `Center`

♦ `RightCenter`

♦ `BottomLeft`, `BottomCenter`, `BottomRight`

- `HotSpot`

In the case of links, new decorations can be added to the following attachment locations, relative to the link shape (see `IlvLinkAttachmentLocation`) :

- `NearFromLink`
- `NearToLink`
- `FromLink`
- `ToLink`
- `MiddleLink`

For details about creating and customizing composite graphics, see the section on developing composite nodes in Developing with the JViews Diagrammer SDK.

You can use the following features when new decorations are added to the predefined business object representation:

- Define a tooltip to be displayed when the mouse is over the new decoration
- Define interactors to be executed when events occur on the new decoration
- Specify in which layer the decoration will be placed
- Specify a visibility threshold that will make the decoration disappear depending on the zoom level of the view

To customize the new decoration and make use of these features, you need to declare the following two properties with the new decoration:

- `ILPATTRIBUTE`: This property associates the decoration with a business attribute in the object. It is used to customize the decoration tooltip and interactor.
- `ILPDECORATIONNAME`: This property associates the decoration with a decoration name, which is the identifier used by the layer policy and zoom policy configurations in the network and equipment components.

The following sections provide examples of how to use these properties to achieve the desired configurations.

How to customize the tooltip and Interactor of a new decoration added to a predefined business object

This use case illustrates how you can configure new decorations added to predefined business objects to display tooltips and react to events. It applies to the network and equipment components.

The following application is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/network/decoration**.

You can add new decorations to the graphic representation of a predefined business object through cascading style sheets.

To configure a tooltip and an interactor for a new decoration, you need to associate the new decoration with a business attribute of the object that is represented. This configuration is performed using property `ILPATTRIBUTE`, as follows:

```
Subobject#commentIcon {
  class: 'ilog.views.graphic.IlvIcon';
  image: '@|image("commentIcon.png")';
  ILPATTRIBUTE: "comments";
  ILPDECORATIONNAME: "Comments";
}
```

The `ILPATTRIBUTE` value is the attribute name as it has been defined in the business model. In this example, the new decoration is associated with the attribute "`comments`" from the business class `SampleNetworkElement`.

Once the decoration has been associated with a business attribute, you can customize a tooltip and an interactor using attribute selectors and properties `toolTipText`, `toolTipGraphic` and `interactor`.

For example:

```
object."SampleNetworkElement/comments"[comments] {
  interactor: @+showCommentInteractor;
  toolTipText: '@|concat(@name, " Comments")';
}
Subobject#showCommentInteractor {
  class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
  action[0]: @+commentButton;
}
Subobject#commentButton {
  class: 'ilog.cpl.interactor.IlpGestureAction';
  gesture: BUTTON1_CLICKED;
  action: @+toggleCommentAction;
}
Subobject#toggleCommentAction {
  class: 'ToggleCommentWindowAction';
}
```

Please note that the interactor customization can only be achieved if the Interactor renderer is enabled. Be sure to enable the renderer if this feature is to be used:

```
Network {
  interactor: true;
}
```

How to define a decoration name for a new decoration added to a predefined business object

This use case illustrates how you can define a new decoration name and associate it with a new decoration that is added to a predefined business object. It applies to the network and equipment components.

The following application is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/network/decoration**.

You can add new decorations to the graphic representation of predefined business objects through cascading style sheets.

The decoration name is an instance of `IltGraphicElementName`. New decoration names can be created by the application, as follows:

```
IltGraphicElementName comments = new IltGraphicElementName("Comments");
```

Note that the decoration name is an enumeration. Therefore, when creating new values for the enumeration, they must be uniquely identified within the application.

You can also use any of the existing decoration names:

- AlarmBalloon
- AlarmCount
- Base
- ContainerStatus
- Family
- Function
- Icon
- InformationIcon
- InformationWindow
- InState
- Media
- Name
- OutState
- Plinth
- PSRIFromEnd
- PSRIToEnd
- SecondaryStateModifiers
- SystemIcon
- SystemWindow
- Generic

Once the decoration name has been created, you can associate it with the new decoration in the cascading style sheet file by defining the value of property `ILPDECORATIONNAME`:

```
Subobject#commentIcon {
  class: 'ilog.views.graphic.IlvIcon';
  image: '@|image("commentIcon.png")';
  ILPATTRIBUTE: "comments";
  ILPDECORATIONNAME: "Comments";
}
```

If the `ILPDECORATIONNAME` property is not defined, the decoration name is set by default to Generic.

How to customize the layer of a new decoration added to a predefined business object

This use case illustrates how you can configure new decorations to be displayed in specific layers. It applies to the network and equipment components.

The following application is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/network/decoration**.

You can add new decorations to the graphic representation of a predefined business object through cascading style sheets.

To configure the layer where a new decoration is to be placed, specify the decoration name to be used as the identifier of the decoration by the network or equipment component layer policy (see Network component). See also *How to define a decoration name for a new decoration added to a predefined business object*.

You can then create a custom layer policy which defines specific layers for the decoration names.

The following application is provided as part of the JViews TGO demonstration software: ***<installdir>* /samples/network/decoration/srchtml/decoration/CustomLayerPolicy.java.html**.

The layer policy implementation must provide at least the following methods: `attach`, `detach` and `getElementLayer`.

```
/**
   * Attaches the layer policy to the manager.
   */
  public void attach(IltcCompositeManager manager) {
    super.attach(manager);
    this.commentsLayer = manager.addLayerOnTop(this.getSystemWindowLayer());
    this.commentsLayer.setName("Comments");
  }
```

The `attach` method is called whenever the layer policy is attached to a network component. This method should be used to perform any initialization required, for example, creating the layers that will be used by the layer policy implementation.

```
/**
   * Detaches the layer policy from the manager.
   *
   */
  public void detach(IltcCompositeManager manager) {
    super.detach(manager);
    manager.removeLayer(this.commentsLayer);
    this.commentsLayer = null;
  }
```

The `detach` method is called whenever the layer policy is removed from the network component. This method should be used to perform cleanup operations on the resources created by the layer policy implementation.

```
/**
   * Returns the layer where the elements with the given name
   * should be placed.
   */
  public IltcLayer getElementLayer (IltcCompositeGraphic graphic,
IltGraphicElementName elementName) {
    if (elementName.getName().equals(COMMENTS_DECORATION_NAME)) {
      return this.commentsLayer;
    } else
      return super.getElementLayer(graphic, elementName);
  }
```

The `getElementLayer` method is used to specify in which layer the decoration with a given name should be placed. This method should verify whether the decoration name passed as argument is the one that you have created, and, if so, return the appropriate layer.

Finally, the custom layer policy is associated with the network or equipment component as follows:

```
CustomLayerPolicy policy = new CustomLayerPolicy();
networkComponent.getView().getCompositeGrapher().setLayerPolicy(policy);
```

How to customize the visibility threshold of a new decoration added to a predefined business object

This use case illustrates how you can configure new decorations added to predefined business objects to have specific visibility thresholds below which the decorations become invisible when the network component is configured with the physical or mixed zoom policies. It applies to the network and equipment components.

The following application is provided as part of the JViews TGO demonstration software: ***<installdir>* /samples/network/decoration**.

You can add new decorations to the graphic representation of a predefined business object through cascading style sheets.

To configure the visibility threshold for a new decoration, first specify the decoration name to be used as the identifier of the decoration by the network or equipment component zoom policy (see Network component). See also *How to define a decoration name for a new decoration added to a predefined business object*.

Once the decoration name has been defined and associated with the decoration using the `ILPDECORATIONNAME` property, you can specify the visibility threshold in the network component configuration. The visibility threshold information applies to the physical and mixed zoom policies.

You have to enable the Zooming renderer in the network component configuration as follows to achieve the desired result:

```
Network {
  zooming: true;
}
```

Configure the zoom policy to be used, for example, the physical zoom policy and specify the visibility threshold for the new decoration name by using the indexed properties `decorationNames` and `visibilityThresholds`:

```
Zooming {
  type: "Physical";
  decorationNames[0]: Comments;
  visibilityThresholds[0]: 0.5;
}
```

For more information on zoom policy configuration, refer to Network component.

Advanced customization of nodes

To further customize the network and equipment node rendering, you may also:

♦ Use an `IlvGraphic` to generate an arbitrarily complex graphic representation (`IlvCompositeGraphic`). For more information about composite graphics, see the section on graphic objects in Architecture of graphic components.

♦ Use a `JComponent` to generate a graphic representation and customize it using CSS.

How to use an IlvGraphic to generate a network node representation

`IlvGraphic` instances can be used to represent network and equipment nodes through the property `'class'`. The given class must follow the JavaBeans™ pattern; its properties can be directly customized in CSS.

The following application is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/network/compositeGraphic**.

It illustrates how to use an `IlvGraphic` to generate a network node representation. This sample shows how to create a complex graphic representation using composite graphics that are fully customizable through CSS.

For information about how to use JavaBeans in CSS and how to use the `class` property, refer to *Class property*.

How to use a JComponent to generate a network or equipment node representation

In the following example, a `JComponent` is used to generate the network node representation of a user-defined class named `Workstation`. Business objects of this class are represented in the network component as a `JButton` with the given label and icon.

```
object."Workstation" {
  class: 'javax.swing.JButton;
  icon : @=icon;
  label: @name;
  foreground: black;
}
object."Workstation":selected {
  foreground: red;
}
Subobject#icon {
  class: 'javax.swing.ImageIcon';
  image: '@|image("workstation.png")';
}
```

The given class must follow the JavaBeans pattern; its properties can be customized directly in CSS (icon, label, foreground).

For information about how to use JavaBeans in CSS and how to use the `class` property, refer to *Class property*.

Customizing network and equipment links

Links are graphically represented in a network or equipment component according to the properties of their business object class. JViews TGO provides a predefined business class, `IltLink`, which displays links with telecommunications information such as states and alarms. In addition, it provides a general purpose link. In both cases, the link representation is fully customizable through CSS.

For details about the customization of links, refer to:

♦ *Customizing links*

♦ *Customizing user-defined business objects*

Just like network and equipment nodes, you can extend links by adding new decorations to them. For more information, refer to *How to add new decorations to network and equipment nodes*.

Customizing tree nodes

Describes the graphical representation of tree nodes and the ways of customizing them.

In this section

Representing tree nodes as graphic objects or icons with labels
Describes the graphical representation of a tree node and how to customize it.

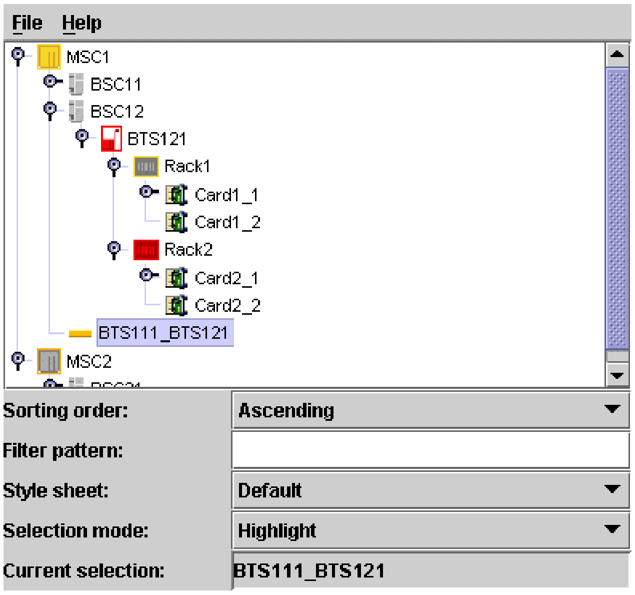
Advanced customization of tree nodes
Describes how to use other classes for more complex representations of tree nodes.

Improving the performance of predefined business objects (tree component)
Describes how to improve rendering performance using static images.

Representing tree nodes as graphic objects or icons with labels

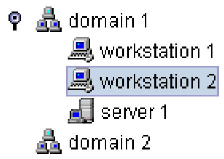
Graphical representation of tree nodes

Tree nodes are graphically represented with a graphic object or icon and a label, as illustrated in the following example.



Sorting order: Ascending
Filter pattern:
Style sheet: Default
Selection mode: Highlight
Current selection: BTS111_BTS121

The default representation for business objects is a representation that combines a label with an optional icon, overlapping icon, and tool tip as illustrated in the following example.



Properties for customizing tree nodes

The following table lists the properties that are common to both predefined and user-defined business objects represented in a tree component.

Common CSS properties for tree node rendering

Property Name	Type of Value	Default	Description
`expansion`	`IlpObject.ExpansionType`	`IN_PLACE`	Determines whether an o be expandable in the tre children can be displaye values are: `IN_PLACE` `IN_PLACE_MINIMAL_I` `NO_EXPANSION` For details, refer to *Cus expansion of business o*
`focusBorderColor`	`Color`	`null`	Color to be used for the of the node. The focus b which cell has the focus is `null`, the color of the look-and-feel is used.
`focusBorderWidth`	`Integer`	`1`	Width of the focus bord
`icon`	`Image`	`null`	Icon to be displayed. If t `null`, no icon is display
`iconVisible`	`Boolean`	`true`	Controls whether the ico displayed or not. If this v and `icon` is `null`, a de be displayed.
`label`	`String`	`null`	Text to be displayed for the value is `null`, the id business object is displa
`labelBackground`	`Color`	`null`	Color to be used for the background. If the value color of the active look-a used.
`labelFont`	`Font`	`null`	Font to be used for the l value is `null`, the font o look-and-feel is used.
`labelForeground`	`Color`	`null`	Color to be used for the foreground. If the value color of the active look-a used.
`labelPosition`	`IlvDirection`	`Right`	Position of the label rela icon. Possible values ar `Left` `Right` `Top`

Property Name	Type of Value	Default	Description
			Bottom Center
labelSpacing	Float	2f	Spacing between the la icon in pixels.
labelVisible	Boolean	true	Controls whether the la displayed or not.
overlapIcon	Image	null	Defines the image used overlap icon.
overlapIconVisible	Boolean	true	Determines whether the is to be visible or not.
selectionFocusMode	IlpSelectionFocusMode	CELL_SELECTION_FOCUS_MODE	Determines the renderi selection and focus. Pos are: CELL_SELECTION_FOC select both icon and lab LABEL_SELECTION_F select only the label. The third possible value property, BASE_SELECTION_FOC is not recommended for tree.
toolTipGraphic	IlvGraphic or JComponent	null	This property accepts I and JComponent objec created in CSS using @ @# constructors.
toolTipText	String	null	Tooltip text for the cell, u toolTipGraphic is null. of this property and the toolTipGraphic property null, no tooltip is displ

Predefined business objects for tree nodes

By default, the predefined business objects for managed objects are displayed in the tiny representation. The tiny representation can be customized using the same CSS properties as for the symbolic representation (for example, foreground, background, label, labelPosition)..

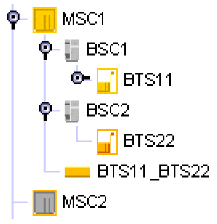
For a list of the properties that you can customize of each type of predefined business object, refer to the following sections:

♦ *Customizing network elements*

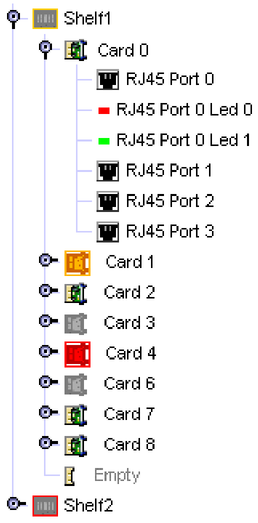
♦ *Customizing links*

- *Customizing groups*
- *Customizing subnetworks*
- *Customizing shelves and cards*
- *Customizing BTS*
- *Customizing off-page connectors*

Objects of the `IltNetworkElement` and `IltLink` classes are represented as follows in a tree:

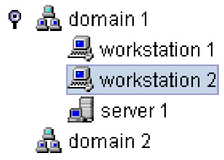


Objects of the `IltShelf`, `IltCard`, `IltPort`, and `IltLed` classes are represented as follows in a tree:



User-defined business objects

The default representation for user-defined business objects is a simple representation which combines a label with an optional icon, overlapping icon, and tool tip.



How to customize tree nodes

The following example shows you how you can customize the tree nodes. It is based on a CSS file.

The following CSS file is provided as part of the JViews TGO demonstration software at **installdir/samples/tree/customClasses/data/tree.css**.

The CSS selectors used to customize the nodes of the tree component are formed by the CSS type `object` and the CSS class `<business class name>`, as illustrated below:

```
object."NetworkElement" {
  label : @name;
  labelForeground : black;
  iconVisible : true;
  toolTipText : @name;
}

object."NetworkElement":selected {
  labelForeground: red;
}
```

The example above illustrates a tree node configuration using pseudoclasses: the graphic representation of the nodes is based on the pseudoclass "`selected`", so that the label foreground color changes whether the object is selected or not in the tree component.

Besides the selection state, you can also customize tree nodes according to the focus state or to custom pseudoclasses. The following example shows tree nodes whose label foreground color changes according to the focus and selection state:

```
object:selected {
  labelForeground: red;
}

object:focus {
  labelForeground: blue;
}

object {
  labelForeground: black;
}
```

How to customize an overlapping icon

You can define an overlapping icon to be displayed over the default tree node icon.

To set an overlapping icon for *all* instances of a given business class, use the following style sheet extract:

```
object."Domain" {
    overlapIcon: '@|image("overlap.png")';
    overlapIconVisible: true;
}
```

Advanced customization of tree nodes

To further customize the tree node rendering, you may also:

♦ Use an `IlvGraphic` to generate an arbitrarily complex graphic representation (`IlvCompositeGraphic`). For more information about composite graphics, see the section on graphic objects in Architecture of graphic components.

♦ Use a `JComponent` to generate a graphic representation and customize it using CSS.

♦ Create your own renderer (which should implement the Swing `TreeCellRenderer` interface) to generate an arbitrary Swing Component.

How to use an IlvGraphic to generate a tree node representation

`IlvGraphic` instances can be used to represent tree nodes through the property `'class'`. The given class must follow the JavaBeans™ pattern; its properties can be directly customized in CSS.

The following example illustrates the use of `IlvCompositeGraphic` to represent tree nodes.

```
object."Workstation" {
  class: 'ilog.views.graphic.composite.IlvCompositeGraphic';
  layout: @+attachmentLayout;
  children[0]: @+wsBase;
  children[1]: @+wsLabel;
  constraints[1]: @+wsLabelConstraint;
}

Subobject#attachmentLayout {
  class: 'ilog.views.graphic.composite.layout.IlvAttachmentLayout';
}
Subobject#wsBase {
  class: 'ilog.views.graphic.IlvIcon';
  image: '@|image("workstation.png")';
}
Subobject#wsLabel {
  class: 'ilog.views.graphic.IlvText';
  label: @name;
  foreground: black;
  font: 'arial-bold-12';
}
Subobject#wsLabel:selected {
  foreground: red;
}
Subobject#wsLabelConstraint {
  class: 'ilog.views.graphic.composite.layout.IlvAttachmentConstraint';
  hotSpot: Left;
  anchor: Right;
  offset: 3,0;
}
```

For information about how to use JavaBeans in CSS and how to use the `class` property, refer to *Class property*.

How to use a JComponent to generate a tree node representation

The following example shows how to define a `JComponent` to generate a tree node representation. It is based on a CSS file.

The following CSS file is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/tree/customClasses/data/tree.css**.

In this example, tree nodes are represented using a simple `JLabel` whose properties `text`, `icon` and `foreground` are customized according to the business attributes and the selection state of the tree node.

```
object."Workstation" {
  class: 'javax.swing.JLabel';
  icon : @=icon;
  text: @name;
  foreground: black;
}
object."Workstation":selected {
  foreground: red;
}
Subobject#icon {
  class: 'javax.swing.ImageIcon';
  image: '@|image("workstation.png")';
}
```

As illustrated in this example, `JComponent` instances can be used to represent tree nodes through the property `'class'`. The given class must follow the JavaBeans pattern; its properties can be customized directly in CSS (icon, text, foreground).

For information about how to use JavaBeans in CSS and how to use the `class` property, refer to *Class property*.

Creating your own renderer

You may want to replace the JViews TGO tree node representation by your own representation. To do so, you need to create your own implementation of the Swing `TreeCellRenderer` interface. For details, refer to Using an arbitrary TreeCellRenderer.

You will then be able to register the new tree cell renderer in the tree component through CSS or through the API.

For information on how to set a new tree cell renderer through CSS, refer to The View rule.

For information on how to set a new tree cell renderer through the API, refer to Configuring the tree component.

Improving the performance of predefined business objects (tree component)

You can improve the rendering performance of predefined business objects by replacing their graphic representation with a static image.

How to improve performance in the Tree component by rendering predefined business objects as static images

By default, the representation of predefined business objects is generated from the tiny representation of the object. If the information about alarms and states is not important to your application, the performance of your application will be better if you replace the default representation by static images.

The following CSS example configures all `ilog.tgo.model.IltObject` business objects to be represented by the image `myImage.png`:

```
object."ilog.tgo.model.IltObject" {
  icon: '@|image("resource/myImage.png")';
  iconVisible: true;
}
```

Customizing table cells

Describes the graphical representation of table cells and the ways of customizing them.

In this section

Representing table cells as business objects or labels with optional icons
Describes the graphic representation of a table cell and how to customize it.

Advanced customization of table cells
Describes how to use other classes for more complex representations of table cells.

Improving the performance of table cell rendering
Describes how to make the rendering of predefined business objects in tables faster.

Representing table cells as business objects or labels with optional icons

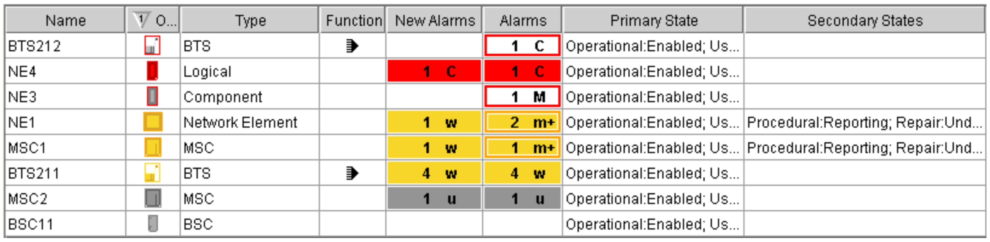
The default table cell renderer (`IlpTableCellRenderer`) generates two default types of graphic representation for table cells:

- ♦ The predefined business objects representation, where the main attributes are displayed using labels and icons, and the object representation is displayed as a compact version of the standard JViews TGO look seen in the network component. This compact version is also known as the *object tiny representation*.
- ♦ A simple representation, combining a label with an optional icon.

Both of these representations can be customized through the use of style sheets.

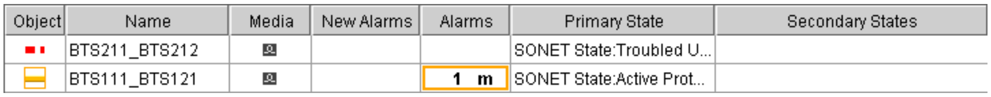
Predefined business objects for table cells

Objects of the `IltNetworkElement` class are represented as follows in the table:



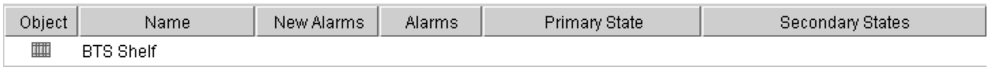
Name	O...	Type	Function	New Alarms	Alarms	Primary State	Secondary States
BTS212	▯	BTS	➧		1 C	Operational:Enabled; Us...	
NE4	▮	Logical		1 C	1 C	Operational:Enabled; Us...	
NE3	▮	Component			1 M	Operational:Enabled; Us...	
NE1	▮	Network Element		1 w	2 m+	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
MSC1	▮	MSC		1 w	1 m+	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
BTS211	▮	BTS	➧	4 w	4 w	Operational:Enabled; Us...	
MSC2	▮	MSC		1 u	1 u	Operational:Enabled; Us...	
BSC11	▮	BSC				Operational:Enabled; Us...	

Objects of the `IltLink` class are represented as follows in the table:



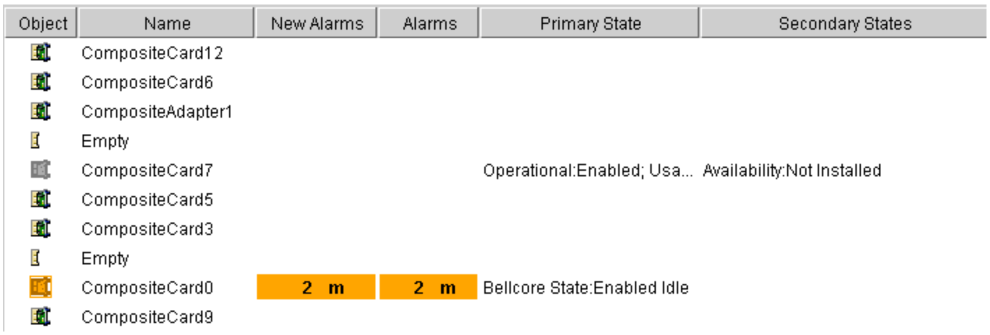
Object	Name	Media	New Alarms	Alarms	Primary State	Secondary States
▬	BTS211_BTS212	▣			SONET State:Troubled U...	
▬	BTS111_BTS121	▣		1 m	SONET State:Active Prot...	

Objects of the `IltShelf` class are represented as follows in the table:



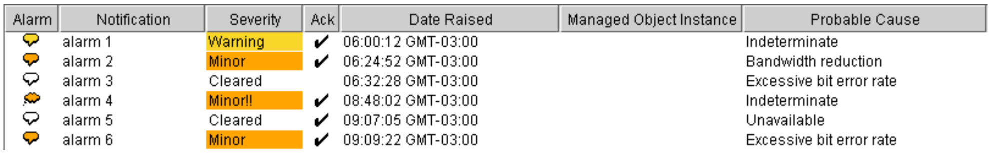
Object	Name	New Alarms	Alarms	Primary State	Secondary States
▦	BTS Shelf				

Objects of the `IltCard` class are represented as follows in the table. (Objects of classes `IltPort` and `IltLed` differ only by the icon in the column “Object.”)



Object	Name	New Alarms	Alarms	Primary State	Secondary States
	CompositeCard12				
	CompositeCard6				
	CompositeAdapter1				
	Empty				
	CompositeCard7			Operational:Enabled; Usa...	Availability:Not Installed
	CompositeCard5				
	CompositeCard3				
	Empty				
	CompositeCard0	2 m	2 m	Bellcore State:Enabled Idle	
	CompositeCard9				

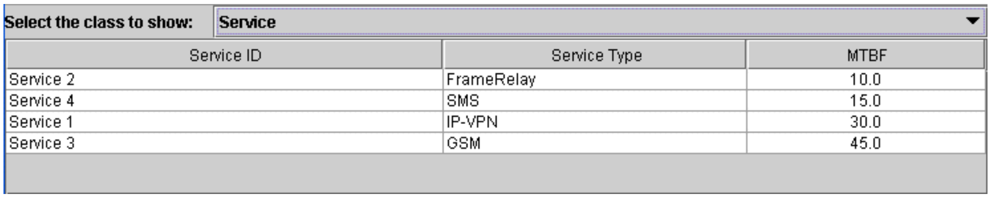
Objects of class `IltAlarm` are represented as follows in the table:



Alarm	Notification	Severity	Ack	Date Raised	Managed Object Instance	Probable Cause
	alarm 1	Warning	✓	06:00:12 GMT-03:00		Indeterminate
	alarm 2	Minor	✓	06:24:52 GMT-03:00		Bandwidth reduction
	alarm 3	Cleared		06:32:28 GMT-03:00		Excessive bit error rate
	alarm 4	Minor!!	✓	08:48:02 GMT-03:00		Indeterminate
	alarm 5	Cleared	✓	09:07:05 GMT-03:00		Unavailable
	alarm 6	Minor	✓	09:09:22 GMT-03:00		Excessive bit error rate

Simple representation for table cells

The simple representation is the default representation for user-defined business objects.



Select the class to show: Service

Service ID	Service Type	MTBF
Service 2	FrameRelay	10.0
Service 4	SMS	15.0
Service 1	IP-VPN	30.0
Service 3	GSM	45.0

Properties for customizing table cells

The following table lists the properties involved in the table cell rendering.

CSS properties for the table cells

Property Name	Type of Value	Default	Description
`focusBorderColor`	`Color`	`null`	Color to be used for the focus border of the cell. The focus border shows which cell has the focus. The default color is the same as in a `JTable`.
`focusBorderWidth`	`Integer`	`1`	Width of the focus border of the cell.
`horizontalAlignment`	`SwingConstants`	`Left`	Horizontal position of the label and icon in the cell. Possible values are:

Property Name	Type of Value	Default	Description

			`Center`
			`Left`
			`Right`.

`icon`	`Image`	`null`	Icon to be displayed. By default, no icon is displayed.
`iconVisible`	`Boolean`	`true`	Determines whether the icon is displayed or not.
`labelBackground`	`Color`	`null`	Color to be used for the label background of a cell. By default, the color is white.
`labelFont`	`Font`	`null`	Font to be used for the label. By default, it is a sans serif font.
`labelForeground`	`Color`	`null`	Color to be used for the label foreground of a cell. By default, the color is black.
`labelInsets`	`Integer`	`1`	Space in pixels around the label and icon.

`labelPosition`	`IlvDirection`	`Right`	Position of the label relative to the icon. Possible values are:
			`Center`
			`Top`
			`Left`
			`Bottom`
			`Right`.

`labelSpacing`	`Float`	`4`	Spacing between the label and the icon.
`label`	`String`	`null`	Text to be displayed for the label. By default, no text is displayed.
`labelVisible`	`Boolean`	`true`	Determines whether the label is displayed or not.
`toolTipGraphic`	`IlvGraphic` or `JComponent`	`null`	This property accepts `IlvGraphic` and `JComponent` objects that are created in CSS using @+, @=, or @# constructors.
`toolTipText`	`String`	`null`	Tooltip text for the cell. By default, no tooltip string is displayed.

`verticalAlignment`	`SwingConstants`	`Center`	Vertical position of the label and icon in the cell. Possible values are:
			`Center`
			`Top`

Property Name	Type of Value	Default	Description
			`Bottom`.

How to customize the object column for predefined business objects in table cells

In the case of predefined business objects for managed objects, an Object column displays the value of the attribute `graphicRepresentation`. It shows the tiny representation of the business object displayed in the table row. To customize this tiny representation, you can use the same CSS properties as for the symbolic representation of the specific predefined business class (for example, `foreground`, `background`, `label`, `labelPosition`). The following code extract:

```
object."ilog.tgo.model.IltNetworkElement" {
  foreground: green;
}
```

changes the foreground color used in the tiny representation of network elements.

For a list of the properties that you can customize per type of predefined business object, refer to the following sections:

- ♦ *Customizing network elements*
- ♦ *Customizing links*
- ♦ *Customizing groups*
- ♦ *Customizing subnetworks*
- ♦ *Customizing shelves and cards*
- ♦ *Customizing BTS*
- ♦ *Customizing off-page connectors*

How to customize table cells

This use case shows you how to customize the cells of the table. The CSS selectors used to customize the table cells are formed by the CSS type `object` and the CSS class `<business class name/attribute name>`, as illustrated in the following example.

The following CSS file is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/table/styling/data/table.css**.

```
object."Alarm/perceivedSeverity"[perceivedSeverity=0] {
  labelBackground: '#FFFFFF';
  label: Cleared;
  toolTipText: "Cleared alarm";
}
```

```
object."Alarm/perceivedSeverity"[perceivedSeverity=1] {
  labelBackground: '#C0C0C0';
  label: Indeterminate;
  toolTipText: "Indeterminate alarm";
}
```

The example illustrates a table cell configuration based on the value of the attribute `perceivedSeverity`.

By means of cascading style sheets, you can also customize the table cell representation according to the focus and selection states. To do so, you use the pseudoclasses `focus` and `selected`, as follows:

```
object."ilog.tgo.model.IltObject/name":selected {
  labelForeground: red;
}

object."ilog.tgo.model.IltObject/name":focus {
  labelForeground: blue;
}

object."ilog.tgo.model.IltObject/name" {
  labelForeground: black;
}
```

How to customize multiple table cells using wildcards

This use case shows you how to customize multiple cells of a table. The CSS selectors used to customize the table cells are formed by the CSS type `object` and the CSS class `<business class name/attribute name>`, as illustrated in the previous example. However, you can use the `*` wildcard to indicate that multiple attributes of a given business class should be configured using a single selector.

The following example illustrates how you can configure all cells of business class `ilog.tgo.model.IltNetworkElement` to have a blue background:

```
object."ilog.tgo.model.IltNetworkElement/*" {
  labelBackground: blue;
}
```

You can also specify that all columns related to alarms should have a bold font:

```
object."ilog.tgo.model.IltObject/*larm*" {
  labelFont: "arial-bold-12";
}
```

The following CSS extract can be found in a CSS file provided as part of the JViews TGO demonstration software at **<*installdir*> /samples/table/styling/data/table.css**.

This CSS extract configures the background of all the cells in the rows displaying objects of the business class `Alarm` according to the value of the `perceivedSeverity` attribute.

```
object."Alarm/*"[perceivedSeverity] {
  labelBackground: '@|valuemap(@=perceivedSeverityBackgroundMap,
@perceivedSeverity)';
}

object."Alarm/*"[perceivedSeverity]:selected {
  labelBackground: '@|valuemap(@=perceivedSeveritySelectionBackgroundMap,
@perceivedSeverity)';
}
```

Advanced customization of table cells

To further customize the table cell rendering, you may also:

♦ Use an IlvGraphic to generate an arbitrarily complex graphic representation (IlvCompositeGraphic). For more information about composite graphics, see the section on graphic objects in Architecture of graphic components.

♦ Use a JComponent to generate a graphic representation and customize it using CSS.

♦ Create your own renderer (which should implement the Swing TableCellRenderer interface) to generate an arbitrary Swing Component.

How to use an IlvGraphic to generate a table cell representation

IlvGraphic instances can be used to represent table cells through the property 'class'. The given class must follow the JavaBeans™ pattern; its properties can be directly customized in CSS.

The following example illustrates the use of IlvGeneralNode to represent table cells:

```
object."Service/type" {
  class: 'ilog.views.sdm.graphic.IlvGeneralNode';
  label: @name;
  labelPosition: Right;
  labelColor: black;
  labelSpacing: 4;
  shapeType: RECTANGLE;
  shapeWidth: 12;
  shapeHeight: 12;
}
object."Service/type":selected {
  labelColor: red;
}
```

For information about how to use JavaBeans in CSS and how to use the class property, refer to *Class property*.

How to use a JComponent to generate a table cell representation

JComponent instances can be used to represent table cells in the same way as they are used to represent tree nodes (see *How to use a JComponent to generate a tree node representation*).

In the following example, table cells are represented using a simple JLabel whose properties text, icon and foreground are customized according to the business attribute and the selection state of the table cell.

```
object."Service/type" {
  class: 'javax.swing.JLabel';
  icon : @=icon;
  text: @type;
  foreground: black;
}
object."Service/type":selected {
  foreground: red;
}
Subobject#icon {
  class: 'javax.swing.ImageIcon';
  image: '@|image("service.png")';
}
```

As illustrated in this example, `JComponent` instances can be used to represent table cells through the property `'class'`. The given class must follow the JavaBeans pattern; its properties can be customized directly in CSS (icon, text, foreground).

For information about how to use JavaBeans in CSS and how to use the `class` property, refer to *Class property*.

Creating your own renderer for table cells

You may want to replace the JViews TGO table cell representation by your own representation. To do so, you need to create your own implementation of the Swing `TableCellRenderer` interface. For details, refer to Using an arbitrary TreeCellRenderer.

You will then be able to register the new table cell renderer in the table component through CSS or through the API.

For information on how to set a new table cell renderer through CSS, refer to The View rule .

For information on how to set a new table cell renderer through the API, refer to Configuring the tree component.

Improving the performance of table cell rendering

You can improve the rendering performance of predefined business objects by replacing their graphic representation with a static image.

How to improve performance in the table component by rendering predefined business objects as static images

The `graphicRepresentation` attribute of predefined business objects can be configured with the `useDefaultCellRenderer` CSS property so that predefined business objects are rendered as static images in the table component. This technique will improve performance as static images are faster to render than the alarm-colored tiny representation of predefined objects.

The following CSS example configures all `ilog.tgo.model.IltObject` business objects to be represented by the image `myImage.png`:

```
object."ilog.tgo.model.IltObject/graphicRepresentation" {
  useDefaultCellRenderer: true;
  icon: '@|image("resource/myImage.png")';
  iconVisible: true;
}
```

Customizing table column headers and rows

The table component uses a default renderer to render the column header (an instance of `IlpTableHeaderRenderer`). This renderer is based on the CSS configuration of the business objects, the attributes, and the table component itself.

How to customize the table header

The following table lists the properties used by the JViews TGO default header renderer.

CSS properties for the table header

CSS Property	Type of Value	Default	Usage

`horizontalAlignment`	`SwingConstants`	`Center`	Horizontal position of the label and icon in the header cell. Possible values are:
			`Center`
			`Left`
			`Right`

`icon`	`Image`	`null`	Icon to be displayed.
`iconVisible`	`Boolean`	`true`	Determines whether the icon is displayed or not. If this value is `true` and `icon` is `null`, the icon will not be displayed.
`background`	`Color`	`null`	Color to be used for the label background. By default, the color is gray.
`foreground`	`Color`	`null`	Color to be used for the label foreground. By default, the color is black.
`labelFont`	`Font`	`null`	Font to be used for the label. By default, it is a sans serif font.
`labelInsets`	`Integer`	`1`	Space in pixels around the label and icon.

`labelPosition`	`IlvDirection`	`Right`	Position of the label relative to the icon. Possible values are:
			`Center`
			`Top`
			`Left`
			`Bottom`

CSS Property	Type of Value	Default	Usage
			Right
labelSpacing	Float	4.0f	Spacing between the label and the icon.
label	String	null	Text to be displayed for the label. By default, there is no text.
labelVisible	Boolean	true	Determines whether the label string is displayed or not.
toolTipText	String	null	Tooltip text for the header cell. By default, there is no tooltip text.
verticalAlignment	SwingConstants	Center	Vertical position of the label and icon in the header cell. Possible values are: Center Top Bottom.
preferredWidth	Integer	-1	The table will attempt to use this as the width of the column corresponding to the attribute. The actual width may be different if you have set an autoresize mode other than AUTO_RESIZE_OFF.
sortingPriority	Integer	0	Determines the sort priority of the column. 0=highest priority. If several columns have a 0 priority, the sort order will be random.
sortingMode	IlpSortingMode	None	Sorting mode of the column. Possible values are: ASCENDING DESCENDING NONE
visible	Boolean	true	Determines whether the corresponding column should be hidden or shown by default.
index	Integer	-1	Determines the index of the column that represents the attribute. This value takes precedence over the tableColumnOrder property.

The following example shows you how you can customize the header of the table.

It is based on a CSS file provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/table/styling/data/table.css**.

The CSS selectors used to customize the table header are formed by the CSS type `attribute` and the CSS class `<business class name/attribute name>`, as illustrated below:

```
attribute."Alarm/identifier" {
  label: "Alarm ID";
  toolTipText: "Alarm identifier";
  preferredWidth: 200;
  horizontalAlignment: Center;
  iconVisible: false;
}

attribute."Alarm/creationTime" {
  label: "Created";
  toolTipText: "Creation time";
  preferredWidth: 250;
  sortingMode: DESCENDING;
}
```

How to customize multiple columns in the table header using wildcards

The CSS selectors used to customize the table header are formed by the CSS type `attribute` and the CSS class `<business class name/attribute name>`, as illustrated in the example above. However, you can use the `*` wildcard to indicate that multiple attributes of a given business class should be configured using a single selector.

The following example illustrates how you can configure all the columns of the `Alarm` business class to have a specific horizontal alignment and icon visibility:

```
attribute."Alarm/*" {
  horizontalAlignment: Center;
  iconVisible: false;
}
```

How to customize the order of columns in a table

The property listed in the following table controls the order of columns in a table.

CSS property for the table column order

CSS Property	Type of Value	Default	Usage
`tableColumnOrder`	`String`	`null`	Defines the column order for the business class represented in the table. Contains a list of attribute names. By default, no column order is defined.

The order of the columns in a table is associated with the business class that is represented. The CSS selectors used to customize this accepted class are formed by the CSS type `object` and the CSS class `<business class name>`, as illustrated below:

```
object."Alarm" {
  tableColumnOrder: "identifier, creationTime, acknowledged,
perceivedSeverity";
}
```

How to customize the height of rows in a table

The property listed in the following table controls the height of rows in a table.

CSS Property for the Table Row Height

CSS Property	Type of Value	Default	Usage
`tableRowHeight`	`Integer`	`16`	Defines the row height for the business class represented in the table.

The height of rows in a table is associated with the business class that is represented. The CSS selectors used to customize this accepted class are formed by the CSS type `object` and the CSS class `<business class name>`, as illustrated below:

```
object."ilog.tgo.model.IltLink" {
  tableRowHeight: 27;
}
```

Customizing the label of a business object

Customizing label parameters in the graphic representation of a business object is based on the properties listed in the following table.

CSS properties for labels

Property Name	Type	Default Value	Description
`labelVisible`	`boolean`	`true`	Controls whether the label is shown or not.
`label`	`String`	`null`	Text to be displayed for the label. If the value is `null`, the identifier of the business object is displayed.
`labelFont`	`Font`	Helvetica 12, except: - in `IltShelf`: Helvetica 10 - in `IltShelfItem`: Helvetica 11 (Courier New 11 on Windows®)	Specifies the font to use to draw the label.
`labelForeground`	`Color`	black, except: - in `IltEmptySlot`: 50% gray	Gives the color of the label text.
`labelBackground`	`Color`	`null`	Gives the color of the label background. The background is transparent when this value is `null`.

`labelPosition`	`IlvDirection`	`Bottom`	Defines the position of the label relative to the base or to the information cluster. Its possible values are:
			`Top`
			`Bottom`
			`Right`
			`Left`
			`Center`
			`BadPosition`.
			When `labelPosition` is set to `BadPosition`, JViews TGO is responsible for defining the label

Property Name	Type	Default Value	Description
			position that best fits with the kind of telecom object being displayed.
`labelAntialiasing`	`boolean`	`true`	Controls whether the label is drawn using antialiasing or not.
`labelSpacing`	`float`	2: For custom business objects. 0: For predefined business objects.	Defines the distance between the label and the object base. Predefined business objects also support the property `labelOffset` that allows you to define the X and Y distance between the label and the base. This property has priority over the property `labelSpacing`.
`labelOffset`	`IlvPoint`	`0, 0`	Indicates the offset in x,y coordinates between the label and the object base.
`labelWrappingMode`	`short`	`None`	Defines the wrapping mode of the object label representation. The following values are available: `None`, `Word Wrap`, `Truncate`, `Wrap and Truncate`.
`labelWrappingWidth`	`float`	-1 The width is automatically defined by JViews TGO according to the object base dimensions. For off-page connectors, the default value is set to 60 pixels.	Defines the label width above which the label will be truncated or wrapped according to the wrapping mode.
`labelWrappingHeight`	`float`	-1 The height is automatically defined by JViews TGO according to the object base dimensions.	Defines the label height above which the label will be truncated.
`labelMargin`	`float`	0: for all objects. 35: for `IltShelfItem`.	Defines the margin between the label and the edge of the shape when performing wordwrapping or truncation
`labelAlignment`	`int`	-1: The label alignment is automatically	Defines the alignment of the label when it has several lines. Possible values are:

Property Name	Type	Default Value	Description

		computed based on the label position:
		- If the label position is `Bottom` or `Top`, the alignment will be `Center`.
		- If the label position is `Left`, the alignment is `Right`.
		- If the label position is `Right`, the alignment is `Left`.
			`Top`
			`Bottom`
			`Center`
			`Left`
			`Right`

`labelBorderColor`	`Color`	transparent (`null`)	Gives the color used to draw a border around the label; the border will be displayed only if this value is not `null`.

`labelDirection`	`IlvDirection`	`Right`, except in:
		- `IltShelfItem`: `Top`
			Defines the direction used to draw the label (vertical or horizontal).
			Possible values are:
			`Top`
			`Bottom`
			`Left`
			`Right`

`labelScaleFactor`	`float`	1	Defines a scale factor that is applied to the label so that it can be adjusted independently from the object base.
`lineSpacing`	`float`	-1: The label will use the default TextLayout leading.	Defines the distance between lines when the label is on several lines.
`minLabelZoom`	`float`	0.6	Defines the minimum label zoom below which the label becomes invisible.
`maxLabelZoom`	`float`	500	Defines the maximum label zoom above which the label becomes invisible.

Note: The customization of labels in a tree component uses all of these properties except `labelScaleFactor`, `minLabelZoom`, and `maxLabelZoom`.

The customization of labels in a table component follows the table cell customization, that is, it uses the properties listed in *CSS properties for the table cells* .

How to customize the label from a business attribute

The following example shows you how to customize a label according to the `name` attribute of the business object:

```
object."ilog.tgo.model.IltObject" {
  label: @name;
}
```

How to customize the label based on specific attribute values

The following example shows you how to customize the graphical representation of the business class `IltNetworkElement` so that the label foreground color is based on the value of the attribute `type`:

```
object."ilog.tgo.model.IltNetworkElement"[type="NMW"] {
labelForeground: blue;
}
object."ilog.tgo.model.IltNetworkElement"[type="BSC"] {
labelForeground: green;
}
```

In this example, `IltNetworkElement` instances whose `type` attribute is set to `NMW` have a blue label. Instances whose `type` attribute is set to `BSC` have a green label.

How to customize the label to wrap or truncate automatically

The following example specifies that the network element labels will wrap automatically if they are larger than 50 pixels.

```
object."ilog.tgo.model.IltNetworkElement" {
  labelWrappingMode: Word Wrap;
  labelWrappingWidth: 50;
}
```

You can also specify that the label will be truncated automatically if it is larger than a specific value. In this case, the label is truncated and '. . .' is added at the end. The following example specifies that network elements will have their label truncated, except when the object is selected.

```
object."ilog.tgo.model.IltNetworkElement" {
  labelWrappingMode: Truncate;
  labelWrappingWidth: 50;
}

object."ilog.tgo.model.IltNetworkElement":selected {
  labelWrappingMode: None;
}
```

Customizing the label in table cells

Properties for customizing labels

You can use the CSS properties listed in the following table to customize the label used to represent an attribute in a table component.

CSS Property	Type of Value	Default	Usage
`label`	`String`	`null`	Text to be displayed for the label. By default, no text is displayed.
`labelVisible`	`Boolean`	`true`	Determines whether the label is displayed or not.
`labelFont`	`Font`	`null`	Font to be used for the label. By default, it is a sans serif font.
`labelForeground`	`Color`	`null`	Color to be used for the label foreground of a cell. By default, the color is black.
`labelBackground`	`Color`	`null`	Color to be used for the label background of a cell. By default, the color is white.

How to customize an attribute in a table component

This example shows how to customize the representation of the attribute `throughput` in a table component.

```
<class>
  <name>Element</name>
  <superClass>ilog.tgo.model.IltNetworkElement</superClass>
  <attribute>
    <name>throughput</name>
    <javaClass>java.lang.Integer</javaClass>
  </attribute>
</class>
```

How to customize cell labels

The following example shows how to customize the cell labels in the class `Element`.

```
object."Element/throughput" {
       label: @throughput;
       labelForeground: yellow;
       labelBackground: red;
}
```

In this example, the value of `throughput` is automatically converted from `Integer` to `String` through the application type converter when the value of `label` is computed. Alternatively, you could use a format function (`@|format`) to obtain the same result. See *CSS expressions and functions* for more information.

Changing the font of all labels

The label font is configured according to whether the object that is linked to the label is a predefined business object, a user-defined business object, or an attribute.

Properties for customizing the font of labels

The following properties are used to configure the label font.

CSS Property	Type of Value	Default	Usage
`labelFont`	`Font`	*Network and Equipment*: Helvetica 12, except: - in `IltShelf`: Helvetica 10 - in `IltShelfItem`: Helvetica 11 (Courier New 11 on Windows®) *Table and Tree:* `null`	Specifies the font to be used for the label. By default, it is a sans serif font.
`alarmBalloonTextFont`	`Font`	Helvetica 12 bold	Denotes the font used to display the text in the alarm balloon.
`alarmCountFont`	`Font`	Helvetica 12 bold	Denotes the font used in alarm counts displayed in the object base.
`infoWindowTextFont`	`Font`	Helvetica 10	Defines the font of the text displayed in the Information Window.

The desired configuration is obtained by changing the value of each of these properties through the appropriate CSS selectors. Since you want to see the modification reflected in all objects and attributes, modify these values using the selectors that contain only the CSS type information:

♦ `object`: selector that identifies all business objects.

♦ `attribute`: selector that applies to attributes in the table component header.

How to configure user-defined business objects

The following example shows the configuration for all user-defined business objects.

```
object {
    labelFont: "sansserif-PLAIN-12";
}
```

How to configure attributes in the table component header

The following example shows the configuration for all attributes.

```
attribute {
    labelFont: "sansserif-PLAIN-12"
}
```

How to configure predefined business objects

The following example shows the configuration for all predefined business objects.

```
object."ilog.tgo.model.IltObject" {
    labelFont:"sansserif-PLAIN-12";
    alarmBalloonTextFont:"sansserif-PLAIN-12";
    alarmCountFont:"sansserif-PLAIN-12";
    infoWindowTextFont:"sansserif-PLAIN-12";
}
```

To have these properties valid for all graphic components, you can use the cascading behavior in the style sheets, so that this style sheet is imported by all the style sheets set in your specific components.

For example, you should first create a style sheet, `shared.css`, that defines the values of these properties only. Then, you should import this style sheet into each component-specific style sheet with the `@import` statement:

```
@import "shared.css";
```

Customizing tooltips

Properties for customizing tooltips

The following properties are used to customize tooltips.

CSS properties for tooltips

Property Name	Type of Value	Default	Description
`toolTipGraphic`	`IlvGraphic` or `JComponent`	`null`	If the value is other than `null`, the given graphic object is used as the tooltip. If the value is `null`, the tooltip will be a simple string as configured by the `toolTipText` property.
`toolTipText`	`String`	`null`	Tooltip text for the object, used only if toolTipGraphic is `null`. If the values of this property and the toolTipGraphic property are both `null`, no tooltip is displayed.
`toolTipFont`	Font	`null` (the default tooltip font defined in the Java™ environment is used)	Defines the font to be used when creating text tooltips.
`toolTipForeground`	Color	`null` (the default tooltip foreground color defined in the Java environment is used)	Defines the foreground color to be used when creating text tooltips.
`toolTipBackground`	Color	`null` (the default tooltip background color defined in the Java environment is used)	Defines the background color to be used when creating text tooltips.

How to customize a tooltip

The following example is valid for network nodes and links, equipment nodes and links, and tree nodes:

```
object."ilog.tgo.model.IltObject" {
  toolTipText: @name;
}
```

How to customize tooltips with a specific font

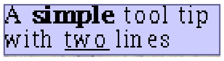
The following example shows you how to customize your component to display tooltips with a specific font on the objects:

```
object."ilog.tgo.model.IltObject" {
  toolTipText: @name;
  toolTipFont: "arial-plain-12";
}
```

How to create a multiline tooltip

The value of the tooltip string is rendered by default in a single line of text. If you want a multiline tooltip or you use fonts with different styles, you can prefix the string with the tag `<HTML>` and then use HTML notation for text attributes. For example, the tooltip shown in *A two-line tooltip with different font styles* uses the following string:

```
<HTML>A <b>simple</b> tool tip<br>with <u>two</u> lines</HTML>
```



A two-line tooltip with different font styles

The following example shows how to obtain this result:

```
object."ilog.tgo.model.IltObject" {
  toolTipText: "<HTML>A <b>simple</b> tool tip<br>with <u>two</u> lines</
HTML>";
}
```

How to create a graphic tooltip

You can also customize tooltips that are `IlvGraphic` or `JComponent` instances. Graphic tooltips are customized in CSS through the property '`toolTipGraphic`' as follows:

```
object."ilog.tgo.model.IltObject" {
  toolTipGraphic: @+myToolTip;
}

Subobject#myToolTip {
  class: 'ilog.views.graphic.IlvIcon';
  image: '@|image("question.png")';
}
```

This example creates a graphic tooltip that displays an icon. The icon graphic is created by the styling engine as a JavaBean™. For details, see *How to Create a New JavaBean Dynamically*.

How to customize a tooltip for a table cell

In the table component, you can customize tooltips for an entire row as explained in previous sections, or you can specify tooltips for a specific table cell. The following example shows how to specify tooltips for table cells that represent network element objects and the `family` attribute.

```
object."ilog.tgo.model.IltNetworkElement/family" {
  toolTipText: @family;
}
```

How to customize a tooltip for a specific decoration

In the network and equipment components, you can customize tooltips for an entire object as explained in previous sections, or you can specify tooltips for a specific decoration. A decoration is created to graphically represent an attribute of the business object. Customizing tooltips for a specific decoration is accomplished by defining CSS selectors for business attributes. In the following example, network elements are customized in a way that their label is truncated when its width is larger than 50. Besides, a tooltip is defined for the `'name'` attribute displaying the full value of the attribute over the label decoration so that you can still see the full name of the object.

```
object."ilog.tgo.model.IltNetworkElement" {
  label: @name;
  labelWrappingMode: Truncate;
  labelWrappingWidth: 50;
}

object."ilog.tgo.model.IltNetworkElement/name" {
  toolTipText: @name;
}
```

The following CSS file is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/network/decoration**.

It illustrates how to define graphic tooltips for a specific business attribute.

How to customize tooltips for secondary states in predefined business objects

In the network and equipment components, you can specify that tooltips are to be displayed for secondary states. A tooltip will be automatically retrieved for each secondary state that is displayed in the business object graphic representation.

```
object."ilog.tgo.model.IltObject/objectState" {
  toolTipGraphic: @+SecStateModifierToolTipGraphic;
}
#SecStateModifierToolTipGraphic {
  class: 'ilog.tgo.graphic.IltSecStateModifierToolTipGraphic';
}
```

The tooltip graphic displays the description of the state. This description is defined when the state is created. For all the predefined states, the description can be modified in the

JViews TGO resource bundle file. For states that you have created using the API, specify the state description as argument when creating the new state.

Customizing object and alarm states of predefined business objects

Describes how to customize the object and alarm states of all the predefined business objects.

In this section

Overview of customizing the states of predefined business objects
Lists the properties that you can use to customize the state parameters in the graphic representation of a business object.

Secondary states and information window properties
Lists the properties for customizing the secondary states and information window and explains how to use them.

Alarm configuration properties
Lists the properties for customizing the alarm configuration and explains how to use them.

Alarm balloon configuration properties
Lists the properties for customizing the alarm balloon configuration and explains how to use them.

Alarm count configuration properties
Lists the properties for customizing the alarm count configuration and explains how to use them.

SNMP system info configuration properties
Lists the properties for customizing the system info configuration and explains how to use them.

Changing the icon color of predefined business objects

Shows you how to change the color of business objects that belong to a predefined business class (tree component, network component, and equipment component).

Overview of customizing the states of predefined business objects

Customizing state parameters in the graphic representation of a business object is based on the following properties:

- *Secondary states and information window properties*
- *Alarm configuration properties*
- *Alarm balloon configuration properties*
- *Alarm count configuration properties*
- *SNMP system info configuration properties*

Secondary states and information window properties

Properties for customizing secondary states and information window

CSS properties for secondary states and information window

Property Name	Type	Default Value	Description
`baseStyleEnabled`	`boolean`	`true` for all objects except groups	Defines whether the object base is modified according to the object state set in the instance.
`secondaryStateVisible`	`boolean`	`true`	Defines whether the secondary state icons are displayed or not
`secondaryStatePosition`	`IlvDirection`	`Top`	Defines the position of the secondary state icons relative to the base. Possible values are: `Top` `Bottom`.
`infoIconThreshold`	`int`	2	Defines the maximum number of secondary state icons that can be displayed. When the number of secondary state icons is bigger than this threshold, the icons are replaced by an information icon and the state information is available through the Information window.
`infoWindowColor`	`Color`	10% gray	Defines the color used to draw the background of the Information window.
`infoWindowBorderColor`	`Color`	black	Defines the color used to draw the border of the Information window.
`infoWindowShadowColor`	`Color`	60% gray	Defines the color used to draw the shadow of the information window.
`infoWindowVisible`	`boolean`	`true`	Determines whether the Information window is visible or not.
`infoWindowPresent`	`boolean`	`false`	Determines whether the Information window is always available. When this value is set to `false`, the Information window is only available when the number of secondary state

Property Name	Type	Default Value	Description
			icons exceeds the limit defined by the property `infoWindowThreshold`. When this value is set to `true`, the Information window can be accessed by clicking the secondary state icons.
`infoWindowTextFont`	`Font`	Helvetica 10	Defines the font of the text displayed in the Information Window.
`infoWindowTextAntialiasing`	`boolean`	`true`	Determines whether the text displayed in the Information window uses antialiasing or not.
`infoWindowTextBackground`	`Color`	`null`	Defines the background color of the text displayed in the Information window.
`infoWindowTextForeground`	`Color`	black	Defines the foreground color of the text displayed in the Information window.
`listPrimaryState`	`boolean`	`true` for OSI and SNMP `false` for other state systems	Determines whether the primary state information is listed in the Information window or not.
`listPrimaryAlarmState`	`boolean`	`false`	Lists the primary alarm state information in the Information window.
`listSecondaryAlarmState`	`boolean`	`false`	Lists the secondary alarm state information in the Information window.
`listAlarmStateAbbreviated`	`boolean`	`false`	Determines whether the alarm state information listed in the Information window displays alarm severities using their abbreviation or their description.

How to customize the secondary states

The following CSS extract modifies the network element graphic representation by specifying that the Information window will always be available when the object has secondary states. To open the Information window, simply click any of the secondary state icons or the Information icon, if it is present.

```
object."ilog.tgo.model.IltNetworkElement" {
  infoWindowPresent: true;
  listPrimaryAlarmState: true;
  listSecondaryAlarmState: true;
  listState: true;
}
```

This extract also specifies that the primary state information and the alarm state information are displayed in the Information window.

Alarm configuration properties

Properties for customizing the alarm configuration

CSS properties for the alarm configuration

Property Name	Type	Default Value	Description
`primaryAlarmState`	`IltAlarmStateEnum`	`Raw`	Determines whether the ra alarms or the impact alarm displayed as the primary a state. Possible values are: `Raw` `Impact`
`alarmBorderVisible`	`boolean`	`true`	Indicates whether the alar border is visible or not aro the object base.
`alarmBorderColor`	`Color`	transparent (`null`)	Defines the color used to represent the alarm borde around the base.
`alarmBorderWidth`	`int`	2	Defines the width of the al border.
`alarmColorVisible`	`boolean`	true	Determines whether the al color is visible or not in the value.
`alarmColor`	`Color`	transparent (`null`)	Determines the color representing alarms in the base. This property is only into account when propert `alarmColorVisible` is `true`.
`alarmBrightColor`	`Color`	transparent (`null`)	Determines the bright colo representing alarms in the base. This property is only into account when propert `alarmColorVisible` is `true`.
`alarmDarkColor`	`Color`	transparent (`null`)	Determines the dark color representing alarms in the base. This property is only into account when propert

Property Name	Type	Default Value	Description
			`alarmColorVisible` is `true`.
`alarmAsMark`	`boolean`	`false`	Defines whether the new a information is displayed us the alarm balloon or an ala marker. The alarm marker another type of balloon wit triangular format. It is displ in the color of the new alar highest severity, without th alarm count information.
`alarmLossOfConnectivityOverride`	`boolean`	`true`	Denotes whether the loss connectivity alarm state overrides the highest alarn representation in the objec base. If the object has the of connectivity state set an property is set to `true`, th object base is displayed us the loss of connectivity col this property is set to `fals` object base is displayed as usual, and the loss of connectivity is represented a secondary state icon.
`alarmLossOfConnectivityPosition`	`IltGraphicElementName`	`AlarmCount`	Defines the position of the of connectivity indicator whe loss of connectivity overrid property is set to `true`. Possible values are: `AlarmCount`: the indicator string specified by the `Ala` `LossOfConnectivity.` `Abbreviation` setting ar displayed as an alarm cou `SecondaryStateModifi` the indicator is the loss of connectivity icon. It is disp

Property Name	Type	Default Value	Description
			as a secondary state and t the loss of connectivity col

How to customize the alarm representation for predefined business objects

The following CSS extract modifies the graphic representation of network element objects, so that raw alarms are considered as primary alarms. As such, they are represented in the object base, the alarm count and the alarm balloon using the default JViews TGO look and feel. In addition, the example modifies the object so that, when the Loss Of Connectivity alarm state is set, it is graphically represented as a secondary state icon and the object base color and object border color are not affected.

```
object."ilog.tgo.model.IltNetworkElement" {
  primaryAlarmState: Raw;
  alarmLossOfConnectivityOverride: false;
}
```

How to customize the alarm representation for predefined business objects to use blinking colors

The following CSS extract shows how you can modify the object representation of JViews TGO predefined business objects to use blinking colors instead of alarm balloon decorations to highlight the presence of new alarms.

This extract illustrates the following configuration:

♦ The alarm balloon decoration is hidden

♦ New alarms are represented in the object base by blinking colors

♦ Outstanding alarms are represented in the object base using the default color configuration

♦ The loss of connectivity status is represented in the object base using the default color configuration

To achieve this configuration, the following CSS properties are used:

♦ `alarmBalloonVisible`

♦ `alarmColorVisible`

♦ `alarmColor`

♦ `alarmBrightColor`

♦ `alarmDarkColor`

```
object."ilog.tgo.model.IltObject" {
  alarmBalloonVisible: false;
  alarmColorVisible: false;
  alarmColor: '';
  alarmBrightColor: '';
  alarmDarkColor: '';
}

object."ilog.tgo.model.IltObject"[alarmHighestSeverity] {
  alarmColorVisible: true;
  alarmColor: '@|severityColor(@|highestSeverity())';
  alarmBrightColor: '@|severityBrightColor(@|highestSeverity())';
  alarmDarkColor: '@|severityDarkColor(@|highestSeverity())';
}

object."ilog.tgo.model.IltObject"[newAlarmHighestSeverity] {
  alarmColorVisible: true;
  alarmColor: '@|blinkingcolor(@|severityColor(@|highestNewSeverity()),
"#50FFFFFF")';
  alarmBrightColor:
'@|blinkingcolor(@|severityBrightColor(@|highestNewSeverity()), "#50FFFFFF")
';
  alarmDarkColor: '@|blinkingcolor(@|severityDarkColor(@|highestNewSeverity()
),
"#50FFFFFF")';
}

object."ilog.tgo.model.IltObject"["objectState.Alarm.LossOfConnectivity"=true]

{
  alarmColorVisible: true;
  alarmColor: '@|settings("Alarm.LossOfConnectivity.Color")';
  alarmBrightColor: '@|settings("Alarm.LossOfConnectivity.BrightColor")';
  alarmDarkColor: '@|settings("Alarm.LossOfConnectivity.DarkColor"';
}
```

You can also customize the default alarm configuration. For more information, refer to *Customizing alarm severities*.

Alarm balloon configuration properties

Properties for customizing the alarm balloon configuration

The following properties apply to the alarm balloon displayed on the base of predefined business objects.

CSS properties for alarm balloon representations

Property Name	Type	Default Value	Description
`alarmBalloonVisible`	`boolean`	`true`	Denotes whether the alarm balloon is visible or not.
`alarmBalloonColor`	`Color`	28% grey	Denotes the color of the alarm balloon. This property is mapped and its value is set according to the color of the highest alarm severity present in the object.
`alarmBalloonShadowColor`	`Color`	black	Denotes the color of the alarm balloon shadow.
`alarmBalloonTextFont`	`Font`	Helvetica 12 bold	Denotes the font used to display the text in the alarm balloon.
`alarmBalloonTextAntialiasing`	`boolean`	`true`	Denotes whether the text inside the alarm balloon is displayed with antialiasing or not.
`alarmBalloonTextForeground`	`Color`	black	Denotes the foreground color used to display the text in the alarm balloon.
`alarmBalloonTextBackground`	`Color`	transparent (`null`)	Denotes the background color used to display the text in the alarm balloon.
`alarmBalloonCountAbbreviated`	`boolean`	`false`	Denotes whether the alarm count displayed in the alarm balloon is abbreviated or not in its collapsed representation. An abbreviated alarm count displays only the number of alarms and the alarm severity abbreviation for the

Property Name	Type	Default Value	Description
			highest alarm present in the object.
`alarmBalloonCountLabel`	`String`	The alarm count to be displayed in the object alarm balloon. It is composed of the number of new alarms of the highest severity, the short description of the highest new alarm severity and a '+' sign in case the object has other alarms of lower severity. For example: `10C+`	Defines the label to be used for the alarm count in the alarm balloon.
`alarmBalloonCountIcon`	`Image`	The image registered for the highest new alarm severity currently present in the object	Defines the image to be used with the label to compose the alarm count in the alarm balloon.

`alarmBalloonPosition`	`IlvDirection`	`Top`	Denotes the position of the alarm balloon around the object base.
			
			Possible values are:
			
			`Top`
			
			`Bottom`
			
			`Left`,
			
			`Right`

`alarmBalloonCollapsed`	`boolean`	`true`	Denotes whether the alarm count displayed in the alarm balloon is abbreviated or not. When this property is set to `false`, the balloon displays the complete list of alarms according to their severities.
`alarmCountIconVisible`	`boolean`	`true`	Determines whether the alarm count icon will be used to create the alarm

Property Name	Type	Default Value	Description
			count in the object base and in the alarm balloon.
`alarmCountIconPosition`	`int`	`IlvConstants.TRAILING`	Defines whether the alarm count icon will be placed before or after the alarm severity description. Possible values are: `IlvConstants.LEADING` or `IlvConstants.TRAILING`.

How to customize the alarm balloon representation

The following CSS extract illustrates how you can customize the alarm balloon representation in your predefined business objects.

```
object."ilog.tgo.model.IltNetworkElement" {
  alarmBalloonPosition: Bottom;
  alarmBalloonTextForeground: white;
}
```

Alarm count configuration properties

Properties for customizing the alarm count configuration

The following properties apply to the alarm count displayed on the base of predefined business objects.

CSS properties for alarm count

Property Name	Type	Default Value	Description
`alarmCountVisible`	`boolean`	`true`	Denotes whether the alarm count in the object base is visible or not.
`alarmCountFont`	`Font`	Helvetica 12 bold	Denotes the font used in alarm counts displayed in the object base.
`alarmCountForeground`	`Color`	black	Denotes the foreground color of the alarm count text displayed in the object base.
`alarmCountBackground`	`Color`	transparent (`null`)	Denotes the background color of the alarm count text displayed in the object base.
`alarmCountAbbreviated`	`boolean`	`false`	Denotes whether the alarm count in the object base is abbreviated or not.
`alarmCountAntialiasing`	`boolean`	`true`	Denotes whether the alarm count in the object base is displayed using antialiasing or not.
`alarmCountMultiline`	`boolean`	`false`, except for network elements of type `NEComponent` and `NEComponent_Logical`	Denotes whether the alarm count in the object base displays on two lines or not. When this property is set to `true`, the number of alarms displays on the first line and the alarm

Property Name	Type	Default Value	Description
			severity abbreviation displays on the second line.
`alarmCountLabel`	`String`	The alarm count to be displayed in the object base. It is composed of the number of outstanding alarms of the highest severity, the short description of the highest outstanding alarm severity and a '+' sign in case the object has other alarms of lower severity. For example: `10C+`	Defines the label to be used for the alarm count in the object base.
`alarmCountIcon`	`Image`	The image registered for the highest outstanding alarm severity currently present in the object	Defines the image to be used with the label to compose the alarm count in the object base.

How to customize the alarm count representation

The following CSS extract illustrates how you can customize the graphic representation of the alarm count displayed on the object base.

```
object."ilog.tgo.model.IltNetworkElement" {
  alarmCountAntialiasing: true;
  alarmCountForeground: yellow;
}
```

SNMP system info configuration properties

Properties for customizing the system info configuration

The following properties apply when a business object has an SNMP object state:

CSS properties applying to SNMP system group

Property Name	Type	Default Value	Description
`snmpSystemContact`	`String`	Contact (`ilog.tgo. SNMP_System_Contact`) The value in parenthesis represents the name of the resource that stores the value of the property in the JViews TGO Resource bundle. When dealing with different locales, the value can be changed according to the locale by defining the value of this property in the appropriate resource file.	Defines the property that denotes the description of the SNMP system contact attribute.
`snmpSystemDescription`	`String`	Description (`ilog.tgo. SNMP_System_Description`) The value in parenthesis represents the name of the resource that stores the value of the property in the JViews TGO Resource bundle. When dealing with different locales, the value can be changed according to the locale by defining the value of this property in the appropriate resource file.	Defines the property that denotes the description of the SNMP system description attribute.
`snmpSystemLocation`	`String`	Location (`ilog.tgo. SNMP_System_Location`) The value in parenthesis represents the name of the resource that stores the value of the property in the JViews TGO Resource bundle. When dealing with different locales, the value can be changed according to the locale by defining the value of this property in the appropriate resource file.	Defines the property that denotes the description of the SNMP system location attribute.

Changing the icon color of predefined business objects

In these components, the predefined business class is an extension of `IltNetworkElement`. The color is customized with the property `foreground`.

The following extract of XML represents the definition of a business class that extends the predefined business class `IltNetworkElement`.

How to extend a predefined business class

```
<class>
  <name>myNetworkElement</name>
    <superClass>ilog.tgo.model.IltNetworkElement</superClass>
    <attribute>
      <name>siteName</name> <javaClass>java.lang.String</javaClass>
    </attribute>
    <attribute>
       <name>customerName</name>  <javaClass>java.lang.String</javaClass>
    </attribute>
</class>
```

How to use literal values for customizing colors

```
object.myNetworkElement {
    foreground: '#FF0000';
}
```

In this example, the foreground color of the icon representing business objects of the class `myNetworkElement` is set to red.

How to customize the icon color based on specific attribute values

```
object.myNetworkElement[type=NE] {
       foreground:'#FFFFFF';
}
object.myNetworkElement[type=MD] {
       foreground:'#C0C0C0';
}
object.myNetworkElement[type=Server] {
       foreground:'#FFCC00';
}
object.myNetworkElement[type=BSC] {
       foreground:'#FFB200';
}
object.myNetworkElement[type=Desktop] {
        foreground:'#FF0000';
}
```

In this example, the foreground color of the icon representing business objects of the class `myNetworkElement` is set to depend on the value of the attribute `type` defined in the class `IltNetworkElement`. So `myNetworkElement` with the type NE will have a white foreground.

Customizing the icon of business objects

Properties for customizing icon parameters

Customizing icon parameters in the graphic representation of a business object is based on the following properties of the graphic representation.

CSS Property	Type of Value	Default	Usage
`icon`	`Image`	`null`	Icon to be displayed.
`iconVisible`	`Boolean`	`true`	Determines whether the icon is displayed or not. If this value is `true` and `icon` is `null`, the icon will not be displayed.

How to customize the icon based on the business object identifier

The CSS selector in this use case is defined based on the object identifier as shown in the following example.

```
Subobject#id {
      icon: '@|image("customer.png")';
      iconVisible: true;
}
```

In this example, the business object with the object identifier `id` will use the icon `customer.png`.

How to customize the icon used in a table cell

```
object."CustomIltNetworkElement/site" {
  iconVisible: true;
  icon: '@|image("site.png")';
}
```

In this example, the table cells in the column `site` will display the icon `site.png` for objects of the class `CustomIltNetworkElement`.

Adding a user-defined business attribute to the system window

Properties of predefined business objects in the system window

In the network and equipment graphic components, predefined business objects can have a specific graphic called the System window, which displays a list of attribute values. You can add attributes to the System window through the properties listed in the following table.

CSS Property	Type of Value	Default	Usage
`visibleInSystemWindow`	`boolean`	`false`	Indicates that the attribute should be displayed in the System window.
`label`	`String`	`null`	Indicates the value to be used to represent the attribute in the System window.

You can also specify the description of the attribute. If the description is set, the attribute will be displayed in the System window in the format <*description: value*>.

The attribute description is configured with the properties listed in the following table.

CSS Property	Type of Value	Default	Usage
`captionLabel`	`String`	`null`	Indicates the label of the description.
`captionLabelVisible`	`boolean`	`true`	Indicates whether the description is displayed or not.

Important: An attribute is only added to the System window if the value `visibleInSystemWindow` is set and if the `label` property is defined with a meaningful value.

Predefined attributes such as `Name` or `ObjectState` are not added to the System window, since they are already graphically represented by other decorations.

How to extend a predefined business class

The following example shows how to create a class that extends `IltNetworkElement` and to declare the new attributes `site` and `vendor`.

In this customization you see that the attribute `site` is added to the System window and is displayed in the format *Site: value*.

The following CSS file is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/network/customClasses/data/customClasses.xml**.

The following extract in XML shows how to define the business class.

```
<class>
  <name>NMW</name>
  <superClass>ilog.tgo.model.IltNetworkElement</superClass>
  <attribute>
    <name>site</name>
    <javaClass>java.lang.String</javaClass>
  </attribute>
  <attribute>
    <name>vendor</name>
    <javaClass>java.lang.String</javaClass>
  </attribute>
</class>
```

How to display an attribute in the system window

The following extract in CSS shows how to declare the new attribute, so that its content is displayed in the System window.

Attributes are configured in CSS with the type `object` and the CSS class formatted as `business class/attribute name`.

```
object."NMW/site" {
    label: @site;
    visibleInSystemWindow: true;
    captionLabel: Site;
    captionLabelVisible: true;
}
```

The property `visibleInSystemWindow` indicates that the attribute is to be displayed inside the System window graphic.

The property `label` indicates that the attribute `site` is to be represented as text with the value of the attribute.

The properties `captionLabelVisible` and `captionLabel` indicate whether or not the attribute is to be displayed with a description in the System window.

Changing the background color of all columns in a table

You can use the `labelBackground` property to change the background color of all the columns.

How to have all the columns use the same background color

This example shows how to get all the columns to have the same background color. The background color is based on the value of one of the attributes.

The following CSS file is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/table/customClasses/data/alarm.css**.

The following extract in XML shows the business class definition.

```
<class>
    <name>Alarm</name>
    <attribute>
      <name>identifier</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
    <attribute>
      <name>perceivedSeverity</name>
      <javaClass>java.lang.Integer</javaClass>
    </attribute>
    <attribute>
      <name>acknowledged</name>
      <javaClass>java.lang.Boolean</javaClass>
    </attribute>
    <attribute>
      <name>creationTime</name>
      <javaClass>java.util.Date</javaClass>
    </attribute>
  </class>
```

The following configuration shows that the background color is changed depending on the value of the attribute `perceivedSeverity`. In the following style sheet extract, the background color is reset when the object is selected in the table. Selected objects are displayed with the default table selected color.

The pseudoclass `selected` is used to configure the representation of the selected objects. In this example, the pseudoclass is repeated to increase the specificity of therule that handles the way a selected object is rendered, thus forcing this rule to have priority over the other rules. See *The CSS specification* for more information.

```
object."Alarm/perceivedSeverity":selected:selected
   {
    labelBackground: '';
}
```

How to make property values dependent on an attribute value

The following example shows that the background color, label, and tooltip change, depending on the value of the attribute.

```
object."Alarm/perceivedSeverity"[perceivedSeverity=0] {
    labelBackground: '#FFFFFF';
    label: Cleared;
    toolTipText: "Cleared alarm";
}
object."Alarm/perceivedSeverity" [perceivedSeverity=1] {
    labelBackground: '#C0C0C0';
    label: Indeterminate;
    toolTipText: "Indeterminate alarm";
}
object."Alarm/perceivedSeverity" [perceivedSeverity=2] {
    labelBackground: '#FFCC00';
    label: Warning;
    toolTipText: "Warning alarm";
}
object."Alarm/perceivedSeverity"[perceivedSeverity=3] {
    labelBackground: '#FFB200';
    label: Minor;
    toolTipText: "Minor alarm";
}
object."Alarm/perceivedSeverity" [perceivedSeverity=4] {
    labelBackground: '#FF0000';
    label: Major;
    toolTipText: "Major alarm";
}
object."Alarm/perceivedSeverity" [perceivedSeverity=5] {
    labelBackground: '#FF0000';
    label: Critical;
    toolTipText: "Critical alarm";
}
```

The background color configuration of the other table columns follows the same principle; the customization of the column identifier is shown in the following example:

```
object."Alarm/identifier":selected:selected {
    labelBackground: '';
}
object."Alarm/identifier"[perceivedSeverity=0] {
    labelBackground: '#FFFFFF';
}
object."Alarm/identifier"[perceivedSeverity=1] {
    labelBackground: '#C0C0C0';
}
object."Alarm/identifier" [perceivedSeverity=2] {
    labelBackground: '#FFCC00';
}
object."Alarm/identifier"[perceivedSeverity=3] {
    labelBackground: '#FFB200';
}
object."Alarm/identifier"[perceivedSeverity=4] {
    labelBackground: '#FF0000';
}
object."Alarm/identifier"[perceivedSeverity=5] {
    labelBackground: '#FF0000';
}
```

Displaying the same attribute with different representations

To obtain the desired result, you need to extend the table component with new attributes. The new attributes refer to attributes of the business class represented in the table component. You configure each attribute to be properly displayed in the table component.

How to extend a table component with new attributes

The following example shows you how to extend a table component with new attributes.

The following application is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/table/customAttributes/index.html**.

```
      // Create a table component
      IlpTable tableComponent = new IlpTable();
      // Get the Alarm class
      IlpClass alarmClass = context.getClassManager().getClass("Alarm");
      // Set the datasource to the component, and show instances
      // of the Alarm class
      tableComponent.setDataSource(dataSource, alarmClass);
      // Add custom attributes
      // Get the existing severity attribute
      IlpAttribute severity = alarmClass.getAttribute("perceivedSeverity");
      // Create a 'Short severity' attribute that represents the severity
      // in a concise way
      IlpAttribute shortSeverityAttribute =
        new IlpReferenceAttribute("shortSeverity", severity);
      tableComponent.addAttribute(shortSeverityAttribute);
```

`IlpReferenceAttribute` models an attribute instance that is a reference to another attribute. In this example, the value returned from querying `perceivedSeverity` and `shortSeverity` is the same.

The following CSS file is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/table/customAttributes/index.html**.

How to customize the way new attributes are displayed

To customize the way the new attributes are to be displayed in the table header, use a selector of type `attribute` followed by the name of the attribute.

```
attribute.shortSeverity {
    label: "S";
    toolTipText: "Perceived severity";
    preferredWidth: 20;
}
attribute.priority {
    label: "P";
    toolTipText: "Priority";
    preferredWidth: 20;
}
```

How to configure the way the table cells are displayed

To configure how the table cells are to be displayed, use a selector of type `object`.

```
object.priority {
    labelVisible: false;
    iconVisible: true;
}
object.priority[priority=0] {
    icon: '';
    toolTipText: '@|format(@#priorityFormat, "Low")';
}
object.priority[priority=10] {
    icon: '@|image("ilog/tgo/ilt_bundle.png")';
    toolTipText: '@|format(@#priorityFormat, "Medium")';
}
object.priority[priority=20] {
    icon: '@|image("ilog/tgo/ilt_busy.png")';
    toolTipText: '@|format(@#priorityFormat, "High")';
}
Subobject#priorityFormat {
    class: 'ilog.cpl.util.text.IlpMessageFormat';
    pattern: "{0} priority";
}
```

Customizing node and link layouts

For details about layout, see Layout.

Properties for customizing the layout of nodes or links

The customization of the layout is based on the properties listed in the following table.

CSS properties for node and link layout

Property Name	Type	Default Value	Description
`linkPorts`	`IltLinkPort []`	`Top`, `Bottom`, `Left`, `Right` and `Center`	Applies only to nodes. This property denotes which link ports are recognized by the node when links are attached to it. This property affects the link layout when set to `IltLinkLayout` or `IltShortLinkLayout`. Possible values are: `Top`, `Bottom`, `Left`, `Right` or `Center`. It is possible to create new link ports by creating new instances of `IltLinkPort`.
`fromPort`	`IltLinkPort`	`null`	Applies only to links. Denotes the preferred link port at the "from" end side. Only effective when an `IltShortLinkLayout` is used.
`toPort`	`IltLinkPort`	`null`	Applies only to links. Denotes the preferred link port at the "to" end side. Only effective when an `IltShortLinkLayout` is used.

How to customize the layout for links

The following CSS extract illustrates how to customize the link layout using link ports. This example features two network elements identified as `NE1` and `NE2`. Each network element has a set of link ports defined to connect the links that are attached to the node. In addition, there is a link connecting `NE1` to `NE2`. For this link, the example specifies the link ports where the link should be attached by the link layout.

```
#NE1 {
  linkPorts[0]: "Center";
}

#NE2 {
  linkPorts[0]: Left;
  linkPorts[1]: Top;
  linkPorts[2]: Bottom;
}
```

```
#Link1 {
  fromPort: Center;
  toPort: Left;
}
```

Link ports are only available when the network or equipment components are configured to use the `IltShortLinkLayout`. Refer to The LinkLayout rule.

Setting parameters to customize nodes or links in a layout

You can set parameters for a particular node or link in a graph layout. This typically applies to certain types of layout, like `IlvBusLayout` (to set the bus object) or the `IlvTreeLayout` (to specify the root object), but you can also set parameters to specify that certain nodes or links must remain fixed. For more information, refer to The GraphLayout rule.

Customizing link label layout

For details about label layout, refer to Label layout.

The customization of link label layout is based on the following CSS properties, valid for `IltLink`, `IltLinkBundle` and `IltLinearGroup`.

CSS properties for link label layout

Property Name	Type	Default Value	Descriptio
`linkLabelMinPercentageFromStart`	`float`	`0`	Defines the of the polylin for the label is performed expressed a length of the
`linkLabelMaxPercentageFromStart`	`float`	`100`	Defines the of the polylin for the label is performed expressed a length of the
`linkLabelMaxDistFromPath`	`float`	`10`	Defines the allowed betw path when a performed.
`linkLabelPreferredDistFromPath`	`float`	`0`	Defines the between the when a labe
`linkLabelPreferredSide`	`IlvDirection`	`Bottom`	Defines the the label sho a label layou meaning of t on property `linkLabel` If the side as the following to specify th preferred sic ♦ `Left`: lef direction ♦ `Right`: r flow direc If the side as the following

Property Name	Type	Default Value	Descriptio
			to specify th preferred sid ♦ Left: lef ♦ Right: r ♦ Top: top ♦ Bottom: ♦ TopLeft ♦ TopRigh side ♦ BottomI left side ♦ BottomF right side
linkLabelAllowedSide	integer	0	Defines the s for the label label layout value can be allowed. The property dep linkLabel If the side as the following to specify th preferred sid ♦ Left: lef direction ♦ Right: r flow direc If the side as the following to specify th preferred sid ♦ Left: lef ♦ Right: r ♦ Top: top

Property Name	Type	Default Value	Descriptio
			♦ `Bottom` ♦ `TopLeft` ♦ `TopRigh` side ♦ `BottomL` left side ♦ `BottomR` right side
`linkLabelSideAssociation`	`IlvAnnealingPolylineLabelDescriptor`	`GLOBAL`	Defines an a the preferred for the label is performed Valid options association ♦ `LOCAL` ♦ `GLOBAL`
`linkLabelTopOverlap`	`float`	`0`	Defines the top side of th the related o layout is per
`linkLabelBottomOverlap`	`float`	`0`	Defines the bottom side overlap the r a label layou
`linkLabelLeftOverlap`	`float`	`0`	Defines the left side of th the related o layout is per
`linkLabelRightOverlap`	`float`	`0`	Defines the right side of the related o layout is per

How to customize link label layout

The following CSS extract illustrates how you can configure your link objects to customize the behavior of the label layout associated with a network or equipment component.

It is provided as part of the JViews TGO demonstration software at ***<installdir>*** **/samples/network/labelLayout/data/network.css**.

```
object {
  linkLabelMinPercentageFromStart: 40;
  linkLabelMaxPercentageFromStart: 60;
  linkLabelMaxDistFromPath: 5;
  linkLabelPreferredDistFromPath: 5;
  linkLabelSideAssociation : 1;
  linkLabelPreferredSide: Bottom;
  linkLabelAllowedSide: Bottom;
  linkLabelTopOverlap: 0;
  linkLabelBottomOverlap: 0;
  linkLabelLeftOverlap: 0;
  linkLabelRightOverlap: 0;
}
```

Customizing the selection border in the network and equipment

Customizing the selection border is based on the following properties.

CSS properties for object selection border in the network and equipment components

Property Name	Type	Default Value	Description
`selectable`	`boolean`	`true`	Defines whether an object is selectable or not in the graphic component.
`selectionBorderForeground`	`Color`	`white`	Defines the foreground color used to draw the selection border.
`selectionBorderBackground`	`Color`	`null`	Defines the background color used to draw the selection border. This property is only considered if the selection border line style is not solid.
`selectionBorderWidth`	`int`	`2`	Defines the width of the selection border.
`selectionBorderLineStyle`	`float[]`	`Solid (null)`	Defines the line style used to draw the selection border.

How to customize the object selection border in the network and equipment components

The following example shows you how to customize the selection border displayed around the object in the network and equipment graphic components.

The example sets a dashed yellow selection border with a width of 2 pixels.

```
object {
  selectionBorderLineStyle: "3,3";
  selectionBorderForeground: yellow;
  selectionBorderWidth: 2;
}
```

Customizing selection in a table or a tree

How to customize selection in the tree component

To customize object selection in the tree component, you should use the "`selected`" pseudoclass, and define the behavior that you want using the CSS properties that are available for the business objects displayed.

The following example changes the label foreground of the object to red when it is selected:

```
object {
  labelForeground: black;
}
object:selected {
  labelForeground: red;
}
```

How to customize selection in the table component

To customize object selection in the table component, you should use the "`selected`" pseudoclass, and define the behavior that you want using the CSS properties that are available for the business objects and attributes displayed.

For example, in a table component that displays network element objects, you can change the color of the label in the Name column using the following CSS extract:

```
object."ilog.tgo.model.IltObject/name" {
  labelForeground: black;
}
object."ilog.tgo.model.IltObject/name":selected {
  labelForeground: red;
}
```

Customizing the expansion of business objects

In JViews TGO, the tree, the network and the equipment components are able to display containment relationships between the business objects. These relationships define a hierarchy of objects that can later be displayed in the components. The object hierarchy is defined through a parent-child relationship set at the data source level. Although the containment relationship is defined at the data source level, it is still possible to specify, at the component level, whether a certain object will be graphically represented as a container or not.

Such configuration is achieved using the following property:

CSS Property for Expanding Business Objects

Property Name	Type	Default Value	Description

`expansion`	`IlpObjectExpansionType`	`NO_EXPANSION` in the network and equipment components
		
		`IN_PLACE` in the tree component
			Determines whether an object should be expandable, so that its children can be displayed.
			
			Possible values are:
			
			`IN_PLACE`
			
			`IN_PLACE_MINIMAL_LOADING`
			
			`NO_EXPANSION`

You can set the `expansion` property for a business object to one of the following values:

- `IN_PLACE`: Loads the child objects automatically in the graphic component. In the tree component, the child objects are loaded on demand, as the user expands the parent node. In the network and equipment components, all child objects are automatically represented in the component when the parent object is created. When the value `IN_PLACE` is used, an object is considered as a parent object if it has containment relationships defined in the attached data source, through the `IlpContainer` interface. The child objects should already be loaded in the data source, and should be visible according to the data source filter, if any.

- `IN_PLACE_MINIMAL_LOADING`: Loads the child objects on demand, as the user expands the parent object. All nodes with this expansion strategy are considered as possible parent nodes, and therefore are represented with an expansion icon. If a tree node does not contain child objects, the expansion icon disappears when the expansion is executed the first time.

- `NO_EXPANSION`: Expansion is not supported by the node. In this case, even if the node contains child objects at the data source level, it will be displayed as a leaf in the graphic component.

How to customize node expansion

```
object."ilog.tgo.model.IltNetworkElement" {
  expansion: NO_EXPANSION;
}

object."test.CustomObject" {
  expansion: IN_PLACE;
}
```

Customizing network elements

Describes what network elements are, how they are represented, and how the representation of different types, aspects, and attributes (decorations) can be customized.

In this section

Representing network elements
Describes what network elements correspond to in the real world and how they are represented graphically.

Customizing network element types
Describes how the representation of a network element is customized graphically to indicate the network element type.

Customizing network element functions
Explains how to create new network element functions, associate them with icons and customize the function icons.

Customizing network element families
Describes how to create a new network element family, which is similar to adding a new function.

Customizing different aspects of network elements
Summarizes how to customize the following aspects of network elements: names, shortcuts, partial network elements, states and alarms, tooltips, decorations.

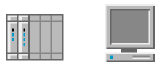
Representing network elements

Network elements are predefined business objects that represent any kind of shelf-based telecom or data-communications equipment (switch, multiplexer, cross-connect or similar), or outside plant equipment (such as a coax node), or peripheral equipment (such as a terminal or printer).

Whole network elements

A whole network element can be represented by a pictorial representation (bitmap image or vector drawing), a symbol, or a shape. Not all physical details of the element are visible in the representation.

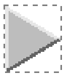
♦ **Pictorial representation**. The network element base is a bitmap image or vector drawing. This drawing is meant to be realistic. Several predefined bases are available for shelf-based equipment, terminals, and mobile phone access network elements. New bases can easily be introduced by providing bitmap images.


Pictorial representations of shelf-based equipment and terminal

♦ **Symbolic representation**. The network element base has a square and the network element function is denoted by a symbol containing ITU/ANSI or traditional symbols. The default type corresponding to the default symbolic network element representation is called NE (Network Element). *Symbolic representation of NE type network element* illustrates an NE type network element: here, an add-drop multiplexer with a capacity of OC192.

Symbolic representation of NE type network element

♦ **Shape representation**. The network element base has a geometric shape that symbolizes the network element type (or function class). The center of the base may contain an icon that further refines the network element function. Several predefined shapes are provided as types of the network elements. *Shape representation of mux network element* illustrates a Mux shape network element.


Shape representation of mux network element

Partial network elements

A *partial network element* is an abstraction which denotes a network element that is only part of the real-world network element. Partial network elements can be used in several situations, for example:

- To represent distributed clusters where parts of a cluster need to be divided across different subnetworks.
- To allow one network element to be used by different service providers. In this case, the network element needs to be divided in several parts. Each part is represented as a partial network element and its state reflects only the elements that are interesting for the service provider that is using it.

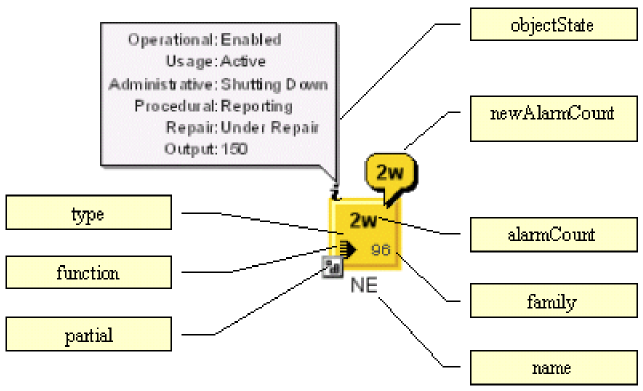
Shortcuts

A *shortcut* network element is an abstraction denoting an object that is only a reference to an existing network element.

Attributes

The grapic representation of the network element and the decorations added to it are based on the information that is available in the business model. Each decoration that is created depends on an attribute and on properties that can be customized through CSS. *Symbolic network element with attributes* shows a symbolic network element with the following attribute set:

- Type: NE
- Function: Access
- Family: OC96
- Name: NE
- Object State: OSI Object State
- Partial: true



Symbolic network element with attributes

Customizing network element types

Describes how the representation of a network element is customized graphically to indicate the network element type.

In this section

Representing a network element type
Describes how the network element type affects the rendering of the object.

Customizing existing network element types
Describes in detail the CSS properties that you can use to customize all network element types and various subsets of network element types.

Creating network element types from images and customizing them
Describes how to define new network element types with new representations without extending an existing base renderer class.

Using the imagecolortuner application to configure the renderer factory
Describes how to find the best values for the parameters of the factory used to create a base renderer for a network element from an image.

Customizing network element types from SVG graphics
Describes how to define new network element types with new representations without extending an existing base renderer class, by using SVG graphics.

Extending the class IltNEBaseRenderer
Explains with an example how to create a new type of network element by extending the base class for the renderer.

Localizing network element types

Describes how to localize then name of the network element type in labels and tooltips in the resource bundle.

Representing a network element type

In ILOG JViews TGO, the type of the network element defines how the object base will be represented. Each network element type is associated with a specific base renderer that is in charge of drawing the object according to its type and state information.

In JViews TGO, you can customize the behavior of the base renderer by using CSS. In addition, you can extend the base representation of graphic objects in two different ways, either by using a predefined base renderer factory class (`IltNEImageBaseRendererFactory` or `IltNESVGBaseRendererFactory`), or by implementing your own subclass of `IltNEBaseRenderer` for each new type of network element that you want to create.

Customizing existing network element types

For details about the network element types and their graphic representation, refer to Network elements

Properties for customizing all network element types

The following properties are common to all network element types:

CSS properties common to all network element types

Property Name	Type	Default Value	Description
`collapsed`	`boolean`	`false`	Defines whether the object is displayed in its normal size or its reduced size. When the network element is shown in its reduced size, fewer decorations are displayed. In particular, there is no space for an alarm count on the base and an alarm balloon would hide secondary state icons. .
`sizeRatio`	`IlvTransformer`	`null`	A magnification of the node.
`tiny`	`boolean`	`false` in network and equipment components `true` in tree and table components	Sets the network element base to its tiny representation. The tiny representation is used mainly in the Tree and Table components.
`logical`	`boolean`	`false`	Sets the network element base to its logical representation. In the logical representation, all network element types are displayed in the same way using a rectangular shape.

Mapped properties

The following properties are mapped, that is, their value is computed automatically by JViews TGO according to the states and alarms currently set in the object (column *Set*). You can however override the mapped values or customize their graphic representation even when the object does not carry states and alarms.

Mapped CSS properties

Property Name	Type	Set	Default Value	Description
`detailLevel`	`enum`	Yes	`MaximumDetails`	Defines the level of detail to be used to draw the base.
`borderColor`	`Color`	Yes	10% gray	Denotes the primary color of the base border.
`borderColor2`	`Color`	Yes	60% gray	Denotes the secondary color of the base border.
`borderWidth`	`float`	Yes	1 pixel	Denotes the width of the base border.
`reliefBorders`	`boolean`	Yes	`true`	Denotes whether the base border is drawn in relief or not.
`borderLineStyle`	`float[]`	Yes	`null` (Solid)	Denotes the line style used to draw the base border.
`borderPattern`	`Pattern`	Yes	`null`	Denotes the pattern used to draw the base border.
`foreground`	`Color`	Yes	28% gray in the `IltObject` class style	Denotes the foreground color of the base.
`background`	`Color`	Yes	Transparent (`null`)	Denotes the background color of the base.

`fillStyle`	`ilog.util.IlFillStyle`	Network node	`IlFillStyle.SOLID_COLOR` for user-defined business objects
			
			`IlFillStyle.PATTERN` for predefined business objects
				Denotes the style used to fill the base of an object.
				
				Possible values are:
				
				`IlFillStyle.NO_FILL`
				
				`IlFillStyle.SOLID_COLOR`
				
				`IlFillStyle.LINEAR_GRADIENT`
				
				`IlFillStyle.RADIAL_GRADIENT`
				
				`IlFillStyle.TEXTURE`
				
				`IlFillStyle.PATTERN`

`fillAngle`	`float`	Network node	`0`	Returns the angle (in degrees) of the gradient used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.RADIAL_GRADIENT` or

Property Name	Type	Set	Default Value	Description
				IlFillStyle. LINEAR_GRADIENT
fillEnd	float	Network node	1f	Returns the position where the gradient of an object ends, that is, where the color is the one defined by property background. This property is only used if fillStyle is set to IlFillStyle. RADIA_GRADIENT or IlFillStyle. LINEAR_GRADIENT
fillStart	float	Network node	0f	Returns the position where the gradient of an object starts, that is, where the color is the one defined by property foreground. This property is only used if fillStyle is set to IlFillStyle. RADIA_GRADIENT or IlFillStyle. LINEAR_GRADIENT.
fillTexture	Image	Network node	null	Denotes the texture used to fill the base of an object. This property is only used if fillStyle is set to IlFillStyle.TEXTURE.
fillPattern	Pattern	Yes	null (Solid)	Denotes the pattern used to fill the base of an object. This property is only used if fillStyle is set to IlFillStyle.PATTERN.

Properties for types with dotted borders

The following types (IltNetworkElement.Type.xxx) have dotted borders: BTS, TransportShape, StationShape, MuxShape, NetworkShape.

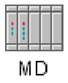
BTS
TransportShape
StationShape
MuxShape
NetworkShape

These types have a dotted border displayed around the main shape by default. The dotted border is configured through the properties listed in the following table.

CSS properties for dotted borders

Property Name	Type	Set	Default Value	Description
`dottedBorderForeground`	`Color`	No	53% gray in the `IltObject` class style	Denotes the foreground color used to display the extra border around the base.
`dottedBorderBackground`	`Color`	No	Transparent `(null)`	Denotes the background color used to display the extra border around the base.
`dottedBorderLineStyle`	`float[]`	No	`"1,1"`	Denotes the line style used to display the extra border around the base.

Properties for mediation devices



The type `IltNetworkElement.Type.MD` determines the graphic representation of a mediation device in the form of a stylized shelf with five slots. It can be customized through the properties listed in the following table.

CSS properties for mediation devices

Property Name	Type	Set	Default Value	Description
`mdRedButtonColor`	`Color`	No	100% red 0% green 20% blue	When the network element is carrying traffic, two cards are drawn with colored buttons. This property denotes the color of the first buttons (which are red by default).
`mdGreenButtonColor`	`Color`	No	0% red 60% green 60% blue	When the network element is carrying traffic, two cards are drawn with colored buttons. This property denotes the color of the last buttons (which are green by default).
`mdShadowColor`	`Color`	No	53% gray	Denotes the color of the shadowed lines that are used to draw the slots that do not contain any card.

Properties for servers

The type `IltNetworkElement.Type.Server` determines the graphic representation of a server in the form of a stylized equipment unit with a colored button and a vent grid. It can be customized through the properties listed in the following table.

CSS properties for servers

Property Name	Type	Set	Default Value	Description
`serverButtonColor`	`Color`	No	100% blue	Defines the color of the server button.
`serverGridColor`	`Color`	No	10% gray	Defines the color of the server grid.

Properties for network management workstations

The type `IltNetworkElement.Type.NMW` determines the graphical representation of a network management workstation in the form of a stylized workstation. It can be customized through the properties listed in the following table.

CSS properties for network management workstations

Property Name	Type	Set	Default Value	Description
`nmwScreenColor`	`Color`	Yes	67% gray	When the network element is carrying traffic, the screen of the workstation is colored with a different shade of gray, which is determined by the value of this property. If the network element holds new alarms, this property is set by JViews TGO.
`nmwButtonColor`	`Color`	No	0% red 60% green 60% blue	Denotes the color of the button that is drawn on the central unit of the workstation.

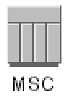
Property for Base Station Controllers

The type `IltNetworkElement.Type.BSC` determines the graphical representation of a wireless Base Station Controller (BSC). It can be customized through the property in the following table.

CSS property for BSC equipment

Property Name	Type	Set	Default Value	Description
`bscActiveBorderWidth`	`int`	No	1 pixel	Denotes the border width that is used when the element is carrying traffic. This type uses a different border width than the value specified by `activeBorderWidth`, as `activeBorderWidth` is designed for “large” representations, such as the generic square chiclet or geometric shapes, whereas the BSC representation uses thinner lines.

Property for Mobile Switching Center

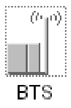


The type `IltNetworkElement.Type.MSC` determines the graphic representation of a Mobile Switching Center (MSC). It can be customized through the following property.

CSS property for MSC equipment

Property Name	Type	Set	Default Value	Description
`mscActiveBorderWidth`	`int`	No	1 pixel	Denotes the border width that is used when the element is carrying traffic. This type uses a different border width than the value specified by `activeBorderWidth`, as `activeBorderWidth` is designed for “large” representations, such as the generic square chiclet or geometric shapes, whereas the MSC representation uses thinner lines.

Property for Base Transceiver Stations



The type `IltNetworkElement.Type.BTS` determines the graphical representation of a Base Transceiver Station (BTS). It can be customized through the property listed in the following table.

CSS property for a BTS object

Property Name	Type	Set	Default Value	Description
`btsActiveBorderWidth`	`int`	No	1 pixel	Denotes the border width that is used when the element is carrying traffic. This type uses a different border width than the value specified by `activeBorderWidth`, as `activeBorderWidth` is designed for “large” representations, such as the generic square chiclet or geometric shapes, whereas the BTS representation uses thinner lines.

Properties for Base Transceiver Station equipment

BTSEquipment

The type `IltNetworkElement.Type.BTSEquipment` determines the graphical representation of the BTS equipment that is part of a BTS object. It can be customized through the following properties.

CSS properties for a BTS equipment

Property Name	Type	Default Value	Description
`antialiasing`	`boolean`	`true`	Defines whether the object is displayed with sharp and neat lines.
`btsEquipmentRadius`	`int`	`10`	Defines the radius of the BTS equipment.

Creating network element types from images and customizing them

JViews TGO provides a base renderer factory class, `IltNEImageBaseRendererFactory`, that simply requires an image from which it will automatically compute all the representations that a network element can have according to the state it is in (out of service, in service carrying traffic, in service carrying no traffic, or not installed) or the alarms it raises. See States.

How to create a new type of network element from an image (using the API)

To create a new type of network element using an image:

♦ Create a new type of network element with the following code:

```
IltNetworkElement.Type myNewType =
          new IltNetworkElement.Type(YOUR_NEW_TYPE_NAME);
```

♦ Create the image corresponding to the base style Carrying Traffic without any alarm and instantiate an `IltNEImageBaseRendererFactory` using this image:

```
IltNEImageBaseRendererFactory factory = new
IltNEImageBaseRendererFactory(image, YOUR_IMAGE_PARAMETERS);
```

♦ Map the factory with the new type using `SetValue(java.lang.Object, java.lang.Object)`:

```
IltSettings.SetValue("NetworkElement.Type.YOUR_NEW_TYPE_NAME.Renderer",facto
ry);
```

How to create a new type of network element from an image (using CSS)

You can create new network element types using global CSS settings as shown in the following example.

```
setting."ilog.tgo.model.IltNetworkElement"{
  types[0]: @+neType0;
}
Subobject#neType0 {
  class: 'ilog.tgo.model.IltNetworkElement.Type';
  name: "YOUR_NEW_TYPE_NAME";
}
```

See *Using global settings* for more information.

How to customize a network element renderer (using CSS)

You can customize the renderer for the network element type using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its `name` attribute in order to match the right type of object in the system. The CSS property to customize is `renderer` as shown in the following example.

```
setting."ilog.tgo.model.IltNetworkElement.Type"[name=YOUR_NEW_TYPE_NAME] {
   renderer: @+neRendererFactory0;
}
Subobject#neRendererFactory0 {
   class: 'MyNERendererFactory';
}
```

In this example, the name of the renderer factory class that is included in the search path is `MyNERendererFactory`.

Starting from an original image of type `java.awt.Image` corresponding to the Carrying Traffic base style without any alarm, JViews TGO automatically processes the range of representations corresponding to other states and alarms.

Suppose that you want to define a new network element with the following representation for the base style Carrying Traffic with no alarms.

Sample new network element

To define this new network element representation, all you have to do is instantiate an `IltNEImageBaseRendererFactory` and provide the bitmap image as an argument to its constructor. JViews TGO automatically computes the various representations that this network element can have according to its associated state and alarms.

The `IltNEImageBaseRendererFactory` has three constructors:

♦ The first constructor has four arguments and enables you to customize the way the different representations are computed.

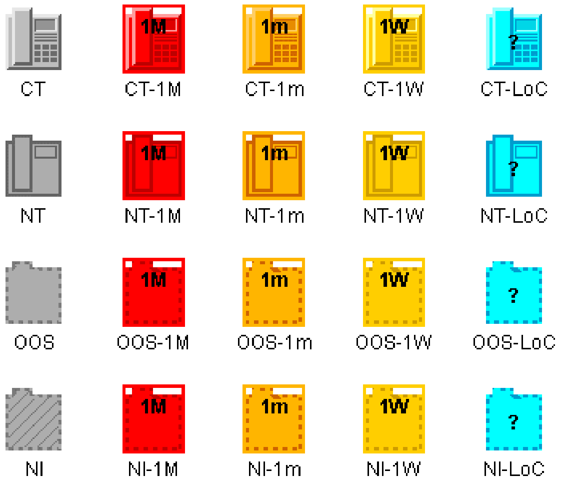
  The first parameter is of type `java.awt.Image`. It is the original image used to compute all other representations.

  The first and the second `int` are thresholds indicating which part of the original image should be considered as details and drawn in the dark color. The third `int` is the gray level that should be mapped to the normal alarm color.

♦ The second constructor has the same parameters as the first one and an additional boolean parameter that indicates whether an additional dotted border should be drawn around the base.

♦ The third constructor has only one argument, which is the `java.awt.Image` that should be used to compute the different representations of the new type. It calls the constructor with four arguments, with the image and 200 as the threshold for bright colors, 50 as the threshold for dark colors, and 128 as the gray level to be mapped to the normal alarm color.

Sample Instantiations of New Network Elements presents the four base styles (rows) with their various representations corresponding to alarms of different severity and loss of connectivity (columns).



Sample Instantiations of New Network Elements

How to create a new type of network element using one image per base style

Another way to represent a type with an image is to specify a source image and an alarm color level parameter for every required base style, directly in CSS. No other base style property or renderer parameter is needed, as a complete image is provided for every required base style.

To create a new type of network element using an image per base style:

♦ Create a new type of network element

Using the API:

```
IltNetworkElement.Type myNewType = new
IltNetworkElement.Type(YOUR_NEW_TYPE_NAME);
```

or, using global CSS settings:

```
setting."ilog.tgo.model.IltNetworkElement" {
   types[0]: @+neType0;
}

Subobject#neType0 {
   class: 'ilog.tgo.model.IltNetworkElement.Type';
   name: YOUR_NEW_TYPE_NAME;
}
```

- Map an `IltNEDirectImageBaseRendererFactory` to the new type

  Using the API `SetValue(java.lang.Object, java.lang.Object)`:

```
IltSettings.SetValue("NetworkElement.Type.YOUR_NEW_TYPE_NAME.Renderer", new

IltNEDirectImageBaseRendererFactory());
```

  or, using global CSS settings:

```
setting."ilog.tgo.model.IltNetworkElement.Type"[name=YOUR_NEW_TYPE_NAME] {
   renderer: @+neRendererFactory;
}

Subobject#neRendererFactory {
   class: 'ilog.tgo.graphic.renderer.IltNEDirectImageBaseRendererFactory';
}
```

- Define an image and an alarm color or gray-level parameter in CSS for each required base style

```
object."ilog.tgo.model.IltNetworkElement"["type"=YOUR_NEW_TYPE_NAME]["object
State.Bellcore.State"=EnabledActive] {
  sourceImage: '@|image("newType_enabledActive.png")';
  alarmColorLevel: 128;
}
object."ilog.tgo.model.IltNetworkElement"["type"=YOUR_NEW_TYPE_NAME]["object
State.Bellcore.State"=DisabledActive] {
  sourceImage: '@|image("newType_disabledActive.png")';
  alarmColorLevel: 140;
}
```

How to define a network element type from an image

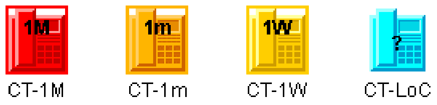
If the base style is **Carrying Traffic no alarms**, the representation is the original image.

Sample original image

In all other cases, the representation is automatically computed by JViews TGO, as follows:

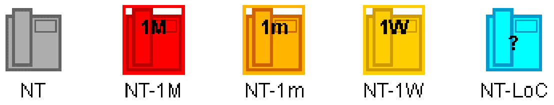
- **Carrying Traffic with alarms**: A color scale corresponding to the alarm color is applied to the original image. This color scale uses three colors: a bright alarm color, a normal alarm color, and a dark alarm color.

  For example, for a major alarm JViews TGO uses the colors bright red, red, and dark red. The color scale is applied to the original image by first converting it to a normalized grayscale image (that is, gray levels in the range 0..255). Then, the normalized grayscale image is colored by mapping each gray level to a color level. The brighter gray is mapped to the bright alarm color, the darker gray is mapped to the dark alarm color, and the gray level corresponding to the third integer in the factory constructor is mapped to the normal alarm color (if this integer is not explicitly given, the default value used is 128). Any other gray level different from the brighter one, the darker one, or the one given in the factory constructor is mapped to the appropriate color in a continuous manner.



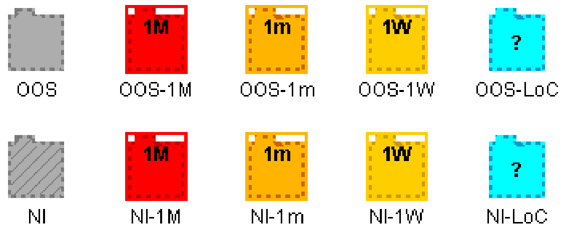
Sample with Color

- **No Traffic (with or without alarms)**: The original image, which is supposed to be a relief image (according to the JViews TGO look-and-feel), is transformed to a flat image. The flat image is composed of only two colors: in the absence of alarms, one gray and one darker gray, otherwise one alarm color and one dark alarm color. Using only two colors ensures a flat visual aspect. To follow the JViews TGO look-and-feel, the whole object should be represented in the normal color with its border in the dark color, plus possibly some details of the inside of the object also in the dark color. For example, the figure shows that the border is in the dark color as well as the outline of the small screen and the receiver of the phone. To indicate which details of the original image you want in the dark color and which you do not, you must use the first two integers of the factory constructor. The first integer indicates a threshold above which the bright colors of the original image should be considered as details and drawn in the dark color; if this integer is not explicitly given, the default value used will be 200. For example, *Sample Dark and Light Colors* shows examples where the threshold assumed is 200 (in the range 0..255). In this way, the left border of the receiver and the lower right border of the screen are drawn in the dark color. The second integer is a threshold indicating which of the dark colors of the original image should be considered as details and also drawn in the dark color; if this integer is not explicitly provided, the default value will be 50. For example, in *Sample Dark and Light Colors* we indicate that this threshold is 11 (in the range 0..255). This causes the lower right border of the receiver and the upper left border of the screen to be drawn in the dark color.



Sample Dark and Light Colors

- **Out of Service and Not Installed (with or without alarms)**: According to the ILOG JTGO look-and-feel, the object should be represented with only two colors, a dashed border and a pattern inside, but no other details. JViews TGO uses the transparency, if any, to determine the border of the image. When there is no transparency, the rectangle of the image is considered to be the border. Then, the interior is filled with the appropriate pattern and the border is drawn as a dashed line. JViews TGO does not need any extra parameters (threshold or gray level as mentioned above) to compute these representations correctly.



Sample Representations

Note that JViews TGO provides the `imagecolortuner` application to help you find the best values for the thresholds and the gray level. For details, see *Using the imagecolortuner application to configure the renderer factory*.

For the `IltNEImageBaseRendererFactory` class to process the supported image types (GIF, PNG and JPG) correctly, the only constraint on the image itself is that the number of colors, `C`, be as follows:

```
2 <= C <= 256
```

The `IltNEImageBaseRendererFactory` class preserves transparency, so transparency is not considered a color. Consequently, `IltNEImageBaseRendererFactory` does not color the transparent pixels of the image. For example, an image that contains the color black as the foreground color, and transparency as the background, will not have the regular coloring scheme applied as it fails to meet the constraint on the number of colors.

Using the imagecolortuner application to configure the renderer factory

The `imagecolortuner` application can be found in `<installdir>/bin` where `<installdir>` is the directory where you have installed JViews TGO. This utility enables you to find the best values for the parameters of `IltNEImageBaseRendererFactory` quickly and easily.

To set the parameters:

1. Click **File>Load Image** to load your image.
2. Customize the `Carrying Traffic with alarm` representations. You have to choose the gray level that will be mapped to the normal alarm color. For details see *Creating network element types from images and customizing them*. To do so, you can either display the tab called `Row 1` and use the slider or click in the zoomed view on a pixel that has the color you want to be mapped to the normal alarm color.
3. Customize the `No Traffic` representation. You have to indicate which details of your original image should have the dark color. For details see *Creating network element types from images and customizing them*. To do so, display the tab called `Row 2` and adjust the two sliders. The first slider adjusts the threshold for bright colors: colors of the original image brighter than this threshold will be displayed in the appropriate dark color depending on the state and alarms. The second slider adjusts the threshold for dark colors: colors of the original image darker than this threshold will also be displayed in the appropriate dark color depending on the state and alarms.
4. Click **File>View Source** to display the sample source code. The source code needed to create your new type as it is currently represented on screen is displayed. The factory constructor contains the appropriate values.

You can create your new type by noting or copying the parameter values or the entire code and pasting it into your JViews TGO application.

Customizing network element types from SVG graphics

How to create a new type of network element from an SVG file (using the API)

JViews TGO provides a base renderer factory class, `IltNESVGBaseRendererFactory`, that simply requires an SVG input file. When using SVG input files, all the possible representations of a network element (according to the state) are mapped to the same SVG graphic.

To create a new type of network element using SVG:

♦ Create a new type of network element with the following code:

```
IltNetworkElement.Type myNewType = new
IltNetworkElement.Type(YOUR_NEW_TYPE_NAME);
```

♦ Create the SVG file corresponding to the network element representation and instantiate an `IltNESVGBaseRendererFactory` using this file:

```
URL url = new URL("file","",YOUR_SVG_FILE_NAME);
IltNESVGBaseRendererFactory factory = new IltNESVGBaseRendererFactory(url)
;
```

♦ Map the factory with the new type using `SetValue(java.lang.Object, java.lang.Object)`:

```
IltSettings.SetValue("NetworkElement.Type.YOUR_NEW_TYPE_NAME.Renderer" ,
factory );
```

How to create a new type of network element from an SVG file (using CSS)

You can create new network element types by using global CSS settings.

```
setting."ilog.tgo.model.IltNetworkElement"{
  types[0]: @+neType0;
}
Subobject#neType0 {
  class: 'ilog.tgo.model.IltNetworkElement.Type';
  name: "YOUR_NEW_TYPE_NAME";
}
```

For more information, see *Using global settings*.

How to customize a renderer for a network element type (using CSS)

You can customize the renderer for the network element type using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its `name` attribute in order to match the right type of object in the system. The CSS property to customize here is `renderer`. In the example below, the name of the renderer factory class that is included in the search path is `MyNESVGRendererFactory`.

```
setting."ilog.tgo.model.IltNetworkElement.Type"[name=YOUR_NEW_TYPE_NAME] {
   renderer: @+neSVGRendererFactory0;
}
Subobject#neSVGRendererFactory0 {
   class: 'MyNESVGRendererFactory';
}
```

Extending the class IltNEBaseRenderer

Drawing principles

The base of a network element has four different visual aspects, each corresponding to one of the following fundamental states:

Out Of Service (OOS)
The base has a hatched outline. Inside the base, drawings are not displayed.

In service, carrying No Traffic (NT)
The base and its inside drawings are in full outline.

In service, Carrying Traffic (CT)
The base and its inside drawings are in relief.

Not Installed (NI)
The base is striped and has a hatched outline. Inside the base, drawings are not displayed. This is a special case where the base drawing changes when a secondary state is represented.

These fundamental states are enumerated in the class `IltBaseStyle`.

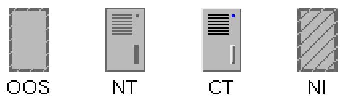
The drawings representing the network element base are specified in a single class that inherits from `IltNEBaseRenderer`. The four visual aspects of the base are drawn using the `drawMain(java.awt.Graphics, ilog.views.IlvTransformer, ilog.views.IlvRect)` method to draw the base and additional methods to access the resources used to draw the base.To create new vector drawings for network elements, you have to extend the `IltNEBaseRenderer` class and override the methods `drawMain(java.awt.Graphics, ilog.views.IlvTransformer, ilog.views.IlvRect)` and `getPreferredSize(boolean)`.

Once the class is created, you must tell JViews TGO how to use it to draw a network element of a particular type. To do so, you must create a new `IltNetworkElement.Type`., add it to the existing types, and associate it with the drawing class using the mapping function `SetValue(java.lang.Object, java.lang.Object)` or global CSS settings.

How to extend a network element

This example illustrates how to define a new type of network element called `Server`. You add `myServer` to the existing types (`NE`, `MD`, `NMW`, `BTS`, and so on).

The four graphic states that the new network element type can have are illustrated in *The `Server` graphic states*.



The `Server` graphic states

When you extend the class `IltNEBaseRenderer` , you must create the drawing class and define its basic methods and the `drawMain` method.

The following code shows how to write a class `ServerBaseRenderer` that extends `IltNEBaseRenderer` and redefine its method `getPreferredSize(boolean)`. This class will be enhanced later on to draw the `Server` element base.

```
static class ServerBaseRenderer extends IltNEBaseRenderer {
  private static Dimension NormalSize = new Dimension(25,40);
  private static Dimension SmallSize = new Dimension(13,21);
  public Dimension getPreferredSize (boolean collapsed) {
      if (collapsed)
        return SmallSize;
      else
        return NormalSize;
  }
}
```

The two fields `NormalSize` and `SmallSize` define the normal dimension of the base and its dimension when it is collapsed. Calling `getPreferredSize` returns the current size of the base.

The following fields are added to initialize numbers that are used to draw the base. Since there is only one method to draw both collapsed and noncollapsed bases, these numbers ease the readability of the code.

```
// Some factors for drawing:
private static final float _XGrid = 4.0f / 25.0f;
private static final float _YGrid = 4.0f / 40.0f;
private static final float _WGrid = 13.0f / 25.0f;
private static final float _HGrid = 1.0f / 40.0f;
private static final float _VStepGrid = 2.0f / 40.0f;
private static final float _XButton = 19.0f / 25.0f;
private static final float _YButton = 4.0f / 40.0f;
private static final float _WButton = 2.0f / 25.0f;
private static final float _HButton = 2.0f / 40.0f;
private static final float _XDisk = 18.0f / 25.0f;
private static final float _YDisk = 23.0f / 40.0f;
private static final float _WDisk = 3.0f / 25.0f;
private static final float _HDisk = 12.0f / 40.0f;
```

Standard palettes

The class `IltNEBaseRenderer` provides three standard drawing palettes that you can access easily with the methods `getPalette()`, `getBrightPalette()`, and `getDarkPalette()`.

- `getPalette()` returns the ordinary palette, commonly used to draw the face part.
- `getBrightPalette()` returns the palette used to draw the bright parts of a 3D border.
- `getDarkPalette()` returns the palette used to draw the ordinary border, and the dark parts of a 3D border.

These palettes automatically take into account the semantic state of the associated `IltNetworkElement` when changing colors. For example, when a new alarm is set to an object the ordinary palette may become red instead of grey.

How to initialize and use new palettes

It is necessary to define new palettes when the base drawings include elements that are not represented in the standard palettes. These palettes are initialized in the `initResources()` method and used in the `drawMain(java.awt.Graphics, ilog.views.IlvTransformer, ilog.views.IlvRect)` method.

In the following code, the new palettes are declared as fields and initialized in `initResources ()`.

```
private static Color GridColor =
      IltrColor.NewColor("server grid color", IltrColor._90PctGrey);
    private static Color ButtonColor =
      IltrColor.NewColor("server button color", Color.blue);

    private Ilt2DPalette GridPalette;
    private Ilt2DPalette ButtonPalette;

    protected void initResources () {
      super.initResources();
      GridPalette = Ilt2DPalette.NewSimple2DPalette(GridColor);
      ButtonPalette = Ilt2DPalette.NewSimple2DPalette(ButtonColor);
    }
```

How to define the main drawing method

The programming principle for the `drawMain(java.awt.Graphics, ilog.views.IlvTransformer, ilog.views.IlvRect)` method is similar to the `draw` method of an `IlvGraphic` object. The method receives a graphic to draw with, a transformer linked to the associated view, and an `IlvRect` object that gives the size and position of the base.

The following code shows the beginning of the `drawMain(java.awt.Graphics, ilog.views.IlvTransformer, ilog.views.IlvRect)` function.

```
public void drawMain (Graphics g, IlvTransformer t, IlvRect rect) {

      // Get the corners of the base rectangle.
      int x1 = (int)rect.x;
      int y1 = (int)rect.y;
      int x2 = x1 + (int)rect.width-1;
      int y2 = y1 + (int)rect.height-1;

      // Get the state dependent parameters.
      IltDetailLevel detailLevel = getDetailLevel();
      Ilt2DPalette palette = getPalette();
      Ilt2DPalette brightPalette = getBrightPalette();
      Ilt2DPalette darkPalette = getDarkPalette();
      int borderThickness = getBorderThickness();
```

In this code, there are the position of the base, its style (OOS, NT, CT or NI—see *The* `Server` *graphic states*), and the standard palettes.

How to draw the base of a new type of network element

You can start drawing with `IltGraphicUtil`, a collection of graphic functions that simplify the drawing of common shapes.

The following code starts drawing the base.

```
if (x1<=x2 && y1<=y2) {

        IltGraphicUtil.FillRect(g,t,x1,y1,x2,y2,palette);

        // Paint the border
        if (detailLevel.equals(IltDetailLevel.MaximumDetails)) {
        if   (rect.width < 20 && borderThickness > 1)
        IltGraphicUtil.DrawReliefRect (g,t,x1,y1,x2,y2,
                                        brightPalette, darkPalette,
                                        borderThickness);
          else {
        IltGraphicUtil.DrawRect(g,t,x1,y1,x2,y2,
                                 darkPalette,
                                 borderThickness);
       }
```

The code has to manage the four different styles of the base, so the border thickness is taken into account to get drawings that look nice. The rest of the `drawMain(java.awt.Graphics, ilog.views.IlvTransformer, ilog.views.IlvRect)` method must also manage the different styles and the two different dimensions of the base.

```
        // Paint the grid, button and disk.
        if (detailLevel.equals(IltDetailLevel.MaximumDetails)
            ||  detailLevel.equals(IltDetailLevel.FewDetails)){
          // Grid, button and disk dimensions.
          final int gridWidth = Math.round(rect.width * _WGrid);
          final int gridHeight = Math.round(rect.height * _HGrid);
          final int buttonWidth = Math.round(rect.width * _WButton);
          final int buttonHeight = Math.round(rect.height * _HButton);
          final int diskWidth = Math.round(rect.width * _WDisk);
          final int diskHeight = Math.round(rect.height * _HDisk);
          // Grid rectangle corners.
          final int gx1 = x1 + Math.round(rect.width * _XGrid);
          int gy1 = y1 + Math.round(rect.height * _YGrid);
          final int gx2 = gx1 + gridWidth-1;
          int gy2 = gy1 + gridHeight-1;
          final int gvstep = Math.round(rect.height * _VStepGrid);
          // Button rectangle corners.
          final int bx1 = x1 + Math.round(rect.width * _XButton);
          final int by1 = y1 + Math.round(rect.height * _YButton);
          final int bx2 = bx1 + buttonWidth-1;
          final int by2 = by1 + buttonHeight-1;
          // Disk rectangle corners.
          final int dx1 = x1 + Math.round(rect.width * _XDisk);
          final int dy1 = y1 + Math.round(rect.height * _YDisk);
          final int dx2 = dx1 + diskWidth-1;
          final int dy2 = dy1 + diskHeight-1;
          // Grid and button colors.
          final Ilt2DPalette gridPalette =
            (detailLevel.equals(IltDetailLevel.FewDetails) ? darkPalette
 : GridPalette);
          final Ilt2DPalette buttonPalette =
            (detailLevel.equals(IltDetailLevel.FewDetails) ? darkPalette
: ButtonPalette);
          // Paint the grid.
          if (gvstep > 1) {
            for (int i = 0; i < 7; i++) {
              IltGraphicUtil._FillRect(g,t, gx1,gy1,gx2,gy2, gridPalette);
              gy1 += gvstep; gy2 += gvstep;
            }
          } else if (gvstep == 1) {
            // Paint a grey rectangle instead of the grid.
            Color rectColor = palette.getForeground();
            Color gridColor = gridPalette.getForeground();
            Color greyColor =
              new Color((rectColor.getRed()+gridColor.getRed())/2,
                        (rectColor.getGreen()+gridColor.getGreen())/2,
                        (rectColor.getBlue()+gridColor.getBlue())/2);
            Ilt2DPalette greyPalette =
              Ilt2DPalette.NewSimple2DPalette(greyColor);
            IltGraphicUtil._FillRect(g,t, gx1,gy1,gx2,gy2+6*gvstep,
                                     greyPalette);
          }

          // Paint the button.
          IltGraphicUtil._FillRect(g,t, bx1,by1,bx2,by2, buttonPalette);

          // Paint the disk.
          if (detailLevel.equals(IltDetailLevel.FewDetails)) {
            IltGraphicUtil._FillRect(g,t, dx1,dy1,dx2,dy2, darkPalette);
          } else {
            IltGraphicUtil._FillRect(g,t, dx1,dy1,dx2,dy2, palette);
            IltGraphicUtil.DrawReliefRect(g,t, dx1,dy1,dx2,dy2,
                                          brightPalette,darkPalette,
                                          1);
         }
```

The method `drawExtraBorders(java.awt.Graphics, ilog.views.IlvTransformer, ilog.views.IlvRect)` is the standard method used in this example.

How to implement a method for drawing a rectangular base

If the object that you want to draw is not a rectangle, but a phone for example, you must override this method to get a selection border that is tightly drawn around the base.

The following code shows the default implementation used for a rectangular base.

```
protected void drawExtraBorders (Graphics g, IlvTransformer t, IlvRect rect)
```

```
{
    { // Draw the alarm border, if needed according to the state.
      Color c = getAlarmBorderColor();
      if (c != null)
        drawExtraBorder(g,t,rect,c,0,IltrThickness.AlarmBorderThickness);
    }
    { // Draw the selection border, if the object is currently selected.
      Color c = getSelectionBorderForeground();
      if (c != null)
        drawExtraBorder(g,t,rect,c,IltrThickness.AlarmBorderThickness,
                        IltrThickness.SelectionBorderThickness);
    }
  }
```

How to create and register a network element type (using the API)

If you run the sample program, the main file creates and registers a network element of type server.

```
static IltNetworkElement.Type Server = new IltNetworkElement.Type("Server");

  static {
    IltSettings.SetValue("NetworkElement.Type.Server.Renderer",
                         new IltBaseRendererFactory() {
                           public IltBaseRenderer createValue() {
                             return new ServerBaseRenderer();
                           }
                         }
    );
  }

IltDefaultDataSource dataSource = new IltDefaultDataSource();
IltNetworkElement ne = new IltNetworkElement
                      ("NE", Server, new IltOSIObjectState());
ne.setAttributeValue(IltObject.PositionAttribute, new IlpPoint(100,100));
dataSource.addObject(ne);
IlpNetwork network = new IlpNetwork();
network.setDataSource(dataSource);
```

How to create and register a network element type (using CSS)

You can create the new network element type using global CSS settings instead of the API, as follows.

```
setting."ilog.tgo.model.IltNetworkElement"{
  types[0]: @+neType0;
}
Subobject#neType0 {
  class: 'ilog.tgo.model.IltNetworkElement.Type';
  name: "Server";
}
```

To customize the renderer using global CSS settings, you need first to create a renderer factory class and make sure it is included in the search path.

```
public class MyNERendererFactory implements IltBaseRendererFactory {
   public IltBaseRenderer createValue() {
      return new ServerBaseRenderer();
   }
}
```

Then you can customize the renderer in CSS as follows:

```
setting."ilog.tgo.model.IltNetworkElement.Type"[name="Server"] {
   renderer: @+neRendererFactory0;
}
Subobject#neRendererFactory0 {
   class: 'MyNERendererFactory';
}
```

As illustrated in this example, you can create a network element with a specific type or set a type for it afterwards with either one of two methods.

```
ne.setType (Server);
```

or

```
ne.setAttributeValue (IltNetworkElement.TypeAttribute, Server);
```

Localizing network element types

In JViews TGO, the network element type defines how the object base will be represented. You may also be interested in displaying the name of the network element type in your application, for example, as a label in a table cell or as a tooltip.

All the predefined network element types have labels and tooltips specified in the JViews TGO resource bundle. See About internationalization.

The resources that apply to network element types are identified as:

- `ilog.tgo.NetworkElement_Type_<TYPE NAME>`: network element type labels
- `ilog.tgo.NetworkElement_Type_<TYPE NAME>_ToolTip`: network element type tooltips

You can edit the values directly in the JViews TGO resource bundle file.

When you create new network element types, the label and tooltip information will also be retrieved from this resource bundle to be displayed, for example, in a table cell. As you create new network element types, register the corresponding entries into the resource bundle file, as follows.

Suppose that you have created the following new network element type:

```
IltNetworkElement.Type myNewType = new IltNetworkElement.Type("MyType");
```

You should declare the following properties in the `JTGOMessages.properties` file:

- `ilog.tgo.NetworkElement_Type_MyType=My Type`
- `ilog.tgo.NetworkElement_Type_MyType_ToolTip=My New Network Element Type`

Customizing network element functions

You need to do the following:

- ♦ Add a new entry to the `IltNetworkElement.Function` enumeration by instantiating the class `IltNetworkElement.Function`.
- ♦ Create an `Image` corresponding to the icon that will be associated with the new function.
- ♦ Map the icon to the function instance. Refer to the `SetValue(java.lang.Object, java.lang.Object)` member function in the class `IltSettings`.

You can also customize the functions

How to add a function and associate It with an Icon (using the API)

The following example shows how to add a `Router` function and associate it with an icon, which is stored in a file called `router.gif`.

```
IlpContext context = IltSystem.GetDefaultContext();
IltNetworkElement.Function router = new IltNetworkElement.Function("router");
Image routerIcon = context.getImageRepository().getImage("router.gif");
IltSettings.SetValue("NetworkElement.Function.Router.Icon", routerIcon);
```

How to add a function, associate It with an icon, and customize the icon (using CSS)

You can create new network element functions by using global CSS settings.

```
setting."ilog.tgo.model.IltNetworkElement"{
  functions[0]: @+neFunction0;
}
Subobject#neFunction0 {
  class: 'ilog.tgo.model.IltNetworkElement.Function';
  name: "router";
}
```

You can also customize the icon using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right object in the system. The CSS property to customize here is `icon`.

```
setting."ilog.tgo.model.IltNetworkElement.Function"[name="router"] {
   icon: '@|image("router.gif")';
}
```

See *Using global settings* for more information.

Showing and hiding function icons

The following property is used to customize the display of network element functions:

CSS Properties for Network Element Functions

Property Name	Type	Default Value	Description
`functionVisible`	`boolean`	`true`	Controls whether the function icon of a network element is shown or not

Declaring tooltips for function icons

Network element functions provide a default tooltip which is retrieved from the JViews TGO resource bundle. See About internationalization.

The resource that applies to network element functions is:

♦ `ilog.tgo.NetworkElement_Function_<FUNCTION NAME>_ToolTip`: network element function tooltips

For the predefined network element functions, you can edit the value directly in the JViews TGO resource bundle file.

For newly created network element functions, the tooltip information will also be retrieved from this resource bundle. As you declare new network element functions, register the corresponding entries in the resource bundle file.

Considering that you have created the network element function "`router`", you should declare the entry in the resource bundle file as follows:

♦ `ilog.tgo.NetworkElement_Function_router_ToolTip=Router`

Customizing network element families

Creating a new network element family entails:

- ♦ Extending the `IltNetworkElement.Family` enumeration.
- ♦ Mapping the new family with a label.

How to create a network element family (using the API)

```
IltNetworkElement.Family OC999 =
  new IltNetworkElement.Family("OC999");
IltSettings.SetValue("NetworkElement.Family.OC999.Label","999");
```

How to create a network element family (using CSS)

You can also create new network element families by using global CSS settings (see *Using global settings* in *Using Cascading Style Sheets* for more information):

```
setting."ilog.tgo.model.IltNetworkElement"{
  families[0]: @+neFamily0;
}
Subobject#neFamily0 {
  class: 'ilog.tgo.model.IltNetworkElement.Family';
  name: "OC999";
}
```

The following properties are used to customize the display of network element families.

CSS properties for network element families

Property Name	Type	Default Value	Description
`familyVisible`	`boolean`	`true`	Determines whether the family label is displayed or not
`familyFont`	`Font`	Helvetica 10	Defines the font used to draw the family label
`familyForeground`	`Color`	35% gray	Defines the color of the family label text
`familyBackground`	`Color`	transparent (`null`)	Defines the color of the family label background, with a null value meaning a transparent background
`familyAntialiasing`	`boolean`	`false`	Determines whether the family label is displayed using antialiasing or not

You can map network element families to labels by using the method `IltSettings.SetValue`. You can also associate a network element family with a resource that is retrieved from the JViews TGO resource bundle (see About internationalization).

Example:

```
IltSettings.SetValue("NetworkElement.Family.OC999.Label","My_NetworkElement_Fam
ily_999");
```

You can obtain the same customization using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right object in the system. The CSS property to customize here is `label`.

```
setting."ilog.tgo.model.IltNetworkElement.Family"[name="OC192-UsrDf"] {
   label: "My_NetworkElement_Family_999";
}
```

By default, network element families and resources are associated automatically. A label and a tooltip are displayed for the new network element family provided that the following resources have been declared in the JViews TGO resource bundle:

- `ilog.tgo.NetworkElement_Family_<FAMILY NAME>`: network element family label
- `ilog.tgo.NetworkElement_Family_<FAMILY NAME>_ToolTip`: network element family tooltip

For the predefined network element families, you can edit the values directly in the JViews TGO resource bundle file.

For newly created network element families, the label and tooltip information will also be retrieved from this resource bundle. As you declare new network element families, register the corresponding entries into the resource bundle file.

Considering that you have created the network element family `"OC999"`, you should declare the corresponding entries in the resource bundle file as follows:

- `ilog.tgo.NetworkElement_Family_OC999=999`
- `ilog.tgo.NetworkElement_Family_OC999_ToolTip=OC 999`

Customizing different aspects of network elements

Customizing network element names

You can customize the display of network element names through the following CSS properties:

- `label`
- `labelVisible`
- `labelFont`
- `labelForeground`
- `labelBackground`
- `labelPosition`
- `labelAntialiasing`
- `labelSpacing`
- `labelWrappingMode`
- `labelWrappingWidth`
- `labelWrappingHeight`
- `labelMargin`
- `labelAlignment`
- `labelBorderColor`
- `labelDirection`
- `lineSpacing`
- `minLabelZoom`
- `maxLabelZoom`
- `labelScaleFactor`

Refer to *Customizing the label of a business object* for details on these properties and how to use them.

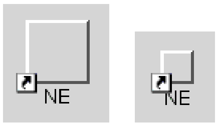
Customizing network element shortcuts

The shortcut representation is displayed according to the business object CSS configuration. The following properties can be used to customize the shortcut representation.

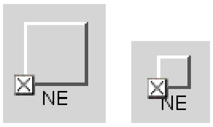
CSS properties for group shortcuts

Property Name	Type	Default Value	Description
`shortcutIcon`	`Image`	`ilt_shortcut.png` for the standard representation `ilt_dangling_shortcut.png` for the dangling representation	Defines the icon to be used in the shortcut representation.
`shortcutIconVisible`	`boolean`	`true`	Defines whether the shortcut icon is displayed or not.

The result is shown in the following images.



Standard shortcuts



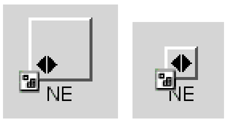
Dangling shortcuts

Customizing partial network elements

The partial representation is displayed according to the business object CSS configuration. The following properties can be used to customize the partial representation:

- ♦ `partialIcon`: This property defines the icon that is used in the partial representation. By default, the image `ilt_partial.png` available in the distribution is used.
- ♦ `partialIconVisible`: This property defines whether the partial icon is displayed or not.

The result is shown in the following images:



Partial network elements expanded and collapsed

Customizing network element states and alarms

Refer to *Customizing object and alarm states of predefined business objects* for information on how to customize the network element representation when it contains states and alarms. The information that you will find in that section is valid for all `IltObject` classes.

Customizing network element tooltips

You can customize the display of network element tooltips through the following CSS properties:

♦ `toolTipGraphic`

♦ `toolTipText`

Refer to *Changing the font of all labels* for details on these properties and how to use them.

Adding new decorations to network elements

You can add new decorations to the predefined business objects representation.

For details, refer to *How to add new decorations to network and equipment nodes*.

Customizing user-defined business objects

Describes how user-defined business objects are represented and how to customize them.

In this section

Representing business objects
Describes how to represent user-defined business objects.

Customizing user-defined network nodes
Describes how to customize the shape, color, and other aspects of user-defined network nodes.

Customizing user-defined network links
Describes how to customize the shape, color, and other aspects of user-defined network links.

Customizing tooltips of user-defined business objects
Describes how to customize the display of a tooltip for a network node or link.

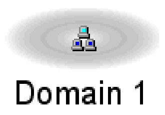
Representing business objects

ILOG JViews TGO graphic components provide support to graphically represent your own business objects, known as *user-defined business objects*. These business objects are generally represented with a simple implementation that you can customize and extend by means of cascading style sheets.

For an example of how to customize user-defined business objects in the network component, refer to ***<installdir>* /samples/network/customClasses**.

Nodes

In the network and equipment components, user-defined business objects that represent nodes are graphically represented by a shape and a label, as shown in the following figure.



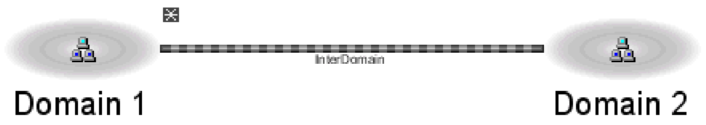
A user-defined network node

For information on how to customize the default graphic representation of user-defined business objects in the network and equipment graphic components, see

- Network and equipment: *Customizing network and equipment nodes*
- Tree: *Customizing tree nodes*
- Table: *Customizing table cells*

Links

In the network and equipment components, user-defined business objects that represent links are rendered as a general link graphic object when they implement `IlpObject` directly, that is, when they do not extend a predefined business link class. The graphical representation is illustrated in the following figure.



A user-defined network link

Customizing user-defined network nodes

Describes how to customize the shape, color, and other aspects of user-defined network nodes.

In this section

Customizing a node shape
Lists all the properties you can use to customize a node shape and describes how to customize sizing aspects of the shape.

Customizing the color of a node shape with paint styles
Lists the values of the node properties related to color and shows the effects they give.

Customizing the border of a node shape
Lists the properties related to the node border and shows how to use them with an example.

Customizing a node icon
Describes how to display an icon in a network node and how to position it with respect to the label.

Customizing a node label
Describes how to customize the display of a label in a network node representation.

Automatic resizing for a node shape with an icon in it
Describes how automatic resizing of a shape with an icon operates.

Customizing a node shape

You can customize the node shape using the properties listed in the following table.

.

CSS properties for network nodes

Property Name	Type	Default Value	Description
`shapeType`	`int`	`ROUND_RECTANGLE`	Sets the shape of an `ilog.views.` `IlvGeneralNode`. The values are `setShapeType(int)`.
`shapeWidth`	`float`	`40`	Sets the width of the shape of an `il` `graphic.IlvGeneralNode`.
`shapeHeight`	`float`	`40`	Sets the height of the shape of an `i` `graphic.IlvGeneralNode`. Use the property `keepingAspectRati`
`shapeAspectRatio`	`float`	`1`	Sets the aspect ratio of the shape o `sdm.graphic.IlvGeneralNode.`
`keepingAspectRatio`	`boolean`	`true`	Specifies whether the width/height ra value of the properties `shapeWidt`
`foreground`	`Color`	`Color.darkGray`	Denotes the node foreground color.
`background`	`Color`	`Color (192,192,192,128)`	Denotes the node background color

`fillStyle`	`int`	`SOLID_COLOR`	Sets the style used to fill the shape `sdm.graphic.IlvGeneralNode.` are those defined in `ilog.views.` `IlvGeneralNode#setFillStyl`
			`NO_FILL`
			`SOLID_COLOR`
			`LINEAR_GRADIENT`
			`RADIAL_GRADIENT`

Property Name	Type	Default Value	Description
			TEXTURE
fillTexture	Image	null	Sets the texture used to fill the shap or the general link.
fillAngle	float	0	Sets the angle (in degrees) of the gr shape of an ilog.views.sdm.gr IlvGeneralNode.
fillStart	float	0f	Sets the position where the gradien starts; that is, where the color is the background.
fillEnd	float	1f	Sets the position where the gradien sdm.graphic.IlvGeneralNode the color is the first color set by fil
horizontalAutoResizeMargin	float	2f	Sets the margin that is left on both si it is horizontally resized automatical
horizontalAutoResizeMode	int	NO_AUTO_RESIZE	Sets the horizontal autoresize mode IlvGeneralNode. The possible valu by ilog.views.sdm.graphic. IlvGeneralNode#setHorizont NO_AUTO_RESIZE EXPAND_ONLY SHRINK_ONLY EXPAND_OR_SHRINK
verticalAutoResizeMargin	float	2f	Sets the margin that is left on both si it is autoresized vertically.
verticalAutoResizeMode	int	NO_AUTO_RESIZE	Sets the vertical autoresize mode of a graphic.IlvGeneralNode. The NO_AUTO_RESIZE EXPAND_ONLY SHRINK_ONLY EXPAND_OR_SHRINK
icon	Image	null	Determines the image used for the i
iconPosition	IlvDirection	Bottom	Sets the position of the icon with res IlvGeneralNode. This property is label position is equal to IlvDirect the label is inside the shape. If the la shape, the icon is always at the cen

Property Name	Type	Default Value	Description
			value is one of the static fields of the views.IlvDirection} interface.
iconVisible	boolean	true	Denotes whether the icon is visible
borderColor	Color	120,120,120	Sets the color of the shape border.
borderEndCap	BasicStroke	BasicStroke. CAP_BUTT	Sets the end cap style of the shape style controls what the ends of the s look like when the border is dashed are: CAP_BUTT CAP_ROUND CAP_SQUARE
borderLineJoin	BasicStroke	BasicStroke. JOIN_MITER	Sets the line join style of the shape property controls what the angles co of the border look like. The possible JOIN_BEVEL JOIN_MITER JOIN_ROUND
borderMiterLimit	float	10f	Sets the miter limit of the shape bor controls how far the angles of the b extend when the angle is very sharp
borderWidth	float	2	Sets the width of the border of the o
borderLineStyle	float[]	null (Solid)	Sets the line style of the shape bord you to have a dashed border. For ex style to 4,8 creates a dashed borde pixels in length separated by spaces style can contain more than two val line style patterns.
borderLineStylePhase	float	0f	Sets the line style phase of the shap can be used to adjust the positions border. This is useful if the dashes l corners of the shape.

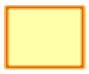
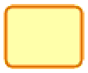
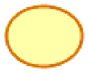
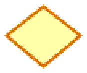
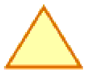
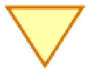
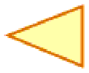
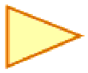
How to use network node properties in a user-defined business class

```
object."test.MyNetworkElement" {
    shapeType: ROUND_RECTANGLE;
    shapeWidth: 30;
    shapeHeight: 30;
    label: @name;
    labelSpacing: 3;
}
```

Shape type

The basic shape of the user-defined node is controlled by the `shapeType` property. The possible values are listed in the following table.

Shape type properties

CSS Property and Values	Rendering
`shapeType = RECTANGLE`	
`shapeType = ROUND_RECTANGLE`	
`shapeType = ELLIPSE`	
`shapeType = DIAMOND`	
`shapeType = TRIANGLE_UP`	
`shapeType = TRIANGLE_DOWN`	
`shapeType = TRIANGLE_LEFT`	
`shapeType = TRIANGLE_RIGHT`	
`shapeType = MARKER`	

How to control the width and height of the shape of a user-defined business object

The horizontal and vertical sizes of the shape are controlled through the properties `shapeWidth`, `shapeHeight`, and `shapeAspectRatio`.

There are two policies for setting the width and height of the shape.

- You can set the properties `shapeWidth` and `shapeHeight`. In this case, the aspect ratio of the shape is not preserved.

  For example:

  ```
  #Node {
    shapeWidth: 100;
    shapeHeight: 50;
  }
  ```

- You can set the property `shapeWidth` to the desired width and the property `shapeAspectRatio` to the desired width:height ratio. For example:

  ```
  #Node {
    shapeWidth: 100;
    shapeAspectRatio: 2;
  }
  ```

  If you change the shape width in another rule, the aspect ratio will be preserved.

The property `keepingAspectRatio` is used to preserve the width to height ratio. This property sets a flag that specifies whether the width to height ratio is preserved by the properties `shapeWidth` and `shapeHeight`.

If the flag is set to `true`, the dimensions of the shape are controlled by calling either `shapeWidth` or `shapeHeight` and by setting `shapeAspectRatio`.

If the flag is set to `false`, the dimensions of the shape are controlled by setting `shapeWidth` and `shapeHeight`.

Note: Setting `shapeHeight` explicitly also causes the keep aspect ratio flag to be set to `false`. Setting `shapeAspectRatio` causes the keep aspect ratio flag to be set to `true` .

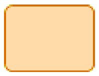
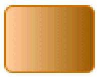
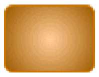
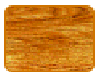
If the keep aspect ratio flag is `true`, the properties `shapeWidth` and `shapeHeight` will preserve the width-to-height ratio of the shape of the node. Otherwise, you can set the width and the height independently.

Customizing the color of a node shape with paint styles

The following properties control the way the interior of the shape is painted: `fillStyle`, `foreground`, `background`, `fillStart`, `fillEnd`, `fillAngle`, and `fillTexture`.

The `fillStyle` property specifies the type of paint object used to fill the shape. The possible values are listed in the following table.

Fill style properties

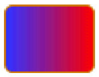
CSS Property and Values	Rendering
`fillStyle = SOLID_COLOR`	
`fillStyle = LINEAR_GRADIENT`	
`fillStyle = RADIAL_GRADIENT`	
`fillStyle = TEXTURE`	

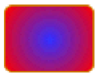
The `foreground` and `background` properties specify the colors used:

- In `SOLID_COLOR` mode, the shape is filled with `foreground`.
- In `LINEAR_GRADIENT` and `RADIAL_GRADIENT` modes, the gradient starts with `foreground` and ends with `background`:

Fill color properties

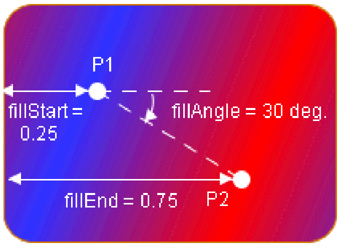
CSS Properties and Values	Rendering

`fillStyle = LINEAR_GRADIENT`
`foreground = blue`
`background = red`
	

`fillStyle = RADIAL_GRADIENT`
`foreground = blue`
`background = red`
	

In `LINEAR_GRADIENT` and `RADIAL_GRADIENT` modes, the `fillStart`, `fillEnd`, and `fillAngle` properties define the geometry of the gradient. A gradient is defined by two points, P1 and P2. The following figures show the results of the properties.

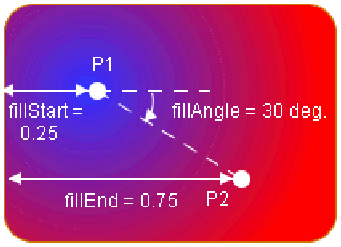
The following figure shows the geometry of a linear gradient.



Linear gradient

Note that the linear gradient is always in *reflect* mode, so the colors go back and forth from `foreground` to `background` outside the (P1, P2) segment.

The following figure shows the geometry of a radial gradient.



Radial gradient

The `fillTexture` property specifies an image that will be used as a texture in `TEXTURE` mode.

Customizing the border of a node shape

Properties for customizing a node border

The border of the shape is controlled by the properties `borderColor`, `borderWidth`, `borderLineStyle`, `borderEndCap`, `borderLineJoin`, `borderMiterLimit`, and `borderLineStylePhase`.

The `borderColor` property sets the color used to paint the border.

The other properties are used to create an instance of `java.awt.BasicStroke`:

- `borderWidth` specifies the width of the border.
- `borderLineStyle` is used to create dashed or dotted borders. It is an array of floating-point values that specify the lengths of the alternate painted and transparent segments.
- `borderEndCap` specifies the shape of the ends of the dash segments.
- `borderLineJoin` and `borderMiterLimit` control the appearance of the border at the angles between two segments.

See the Java™ documentation of the `java.awt.BasicStroke` class for more details on these properties.

How to control the border of a user-defined business object

The following CSS file creates a dashed blue border with rounded segment ends, visible segments that have a length of 4, and transparent segments that have a length of 2.

```
#Node {
  borderColor: blue;
  borderLineStyle: "4, 2";
  borderEndCap: CAP_ROUND;
}
```

Customizing a node icon

To display an icon inside the shape of a network node, set the property `icon` to the URL of the icon to be displayed. Note that the URL should be absolute. The URL Access Service is not implicitly used here. If you want to use it, you must do so explicitly.

If you do not want an icon, set the property `iconVisible` to `false`.

The icon is always displayed inside the shape (or centered on top of the marker if the shape type is `MARKER`).

If the label is inside the shape (`labelPosition = Center`), the position of the icon relative to the label is controlled by the property `iconPosition`, which can be any direction defined by the interface `IlvDirection`. For example, `iconPosition = Left` places the icon to the left of the label.

Customizing a node label

The label decoration of a user-defined business object is customized using the same properties as for labels of predefined business objects.

How to use label properties in a user-defined business class

```
object."test.MyNetworkElement" {
  labelBackground: white;
  labelForeground: blue;
  labelPosition: Right;
  label: @name;
  labelVisible: true;
}
```

For a complete list of the available label properties, refer to *Customizing the label of a business object*.

Automatic resizing for a node shape with an icon in it

The representation of a user-defined node automatically computes the size of its shape according to the size of the icon.

Autoresizing can be controlled independently in the vertical and horizontal directions through the properties `horizontalAutoResizeMode` and `verticalAutoResizeMode`. These properties accept the values listed in the following table.

CSS Properties for Autoresizing of Nodes

Values	Behavior
`NO_AUTO_RESIZE`	Autoresize is disabled.
`EXPAND_ONLY`	The node is allowed to grow in the specified direction, but not to shrink.
`SHRINK_ONLY`	The node is allowed to shrink in the specified direction, but not to grow.
`EXPAND_OR_SHRINK`	The node is allowed to expand or to shrink as needed.

You can control how much space is left between the border of the shape and its content (icon) using the properties `horizontalAutoResizeMargin` and `verticalAutoResizeMargin`.

Customizing user-defined network links

Describes how to customize the shape, color, and other aspects of user-defined network links.

In this section

Customizing a link
Lists all the properties you can use to customize a link and shows how to customize the line aspect of the link with an example.

Customizing various aspects of links
Describes how to customize various aspects of network links.

Customizing a link label
Describes how to customize the display of a label in a network link representation.

Customizing a link

You can customize a network link using the properties listed in the following table.

CSS properties for network links

Property Name	Type	Default Value	Description
`alternateColor`	`Color`	`null`	Sets the alternate color.
`animateSpeed`	`float`	`0`	Sets the animate speed.
`arrowColor`	`Color`	`Color.black`	Sets the arrow color.

`arrowMode`	`int`	`ARROW_FILL`	Sets the arrow-drawing style. The possible values are:
			`ARROW_FILL`
			`ARROW_OPEN`
			`ARROW_GRADIENT`
			`ARROW_DECOERRATION`

`arrowPosition`	`float`	`1f`	Sets the position of the arrow as the ratio of the link length. For example, `0.5f` is the middle of the link.
`arrowRatio`	`float`	`1f`	Sets the size of the arrow proportionately to the length of the link.
`borderColor`	`Color`	`null`	Sets the border color.
`borderColor2`	`Color`	`null`	Sets the lower border color.

`borderLineStyle`	`float[]`	`null`	Sets the array that represents the lengths of the dash segments for the border. The possible values are:
			`Solid`
			`Dot`
			`Dash`
			`DashDot`
			`DashDoubleDot`
			`Alternate`
			`LongDash`
			`DoubleDot`

Property Name	Type	Default Value	Description
			or an array that defines a new line style.
`borderLineStylePhase`	`float`	`0f`	Sets the dash pattern offset for the border.
`borderWidth`	`float`	`0f`	Sets the border width.
`curved`	`float`	`0f`	Sets the curved appearance and the smoothness of the spline.

`endCap`	`int`	`CAP_SQUARE`	Sets the decorations applied to the end of the polyline. The possible values are:
			`CAP_BUTT`
			`CAP_ROUND`
			`CAP_SQUARE`

`internalZoom`	`float`	`1f`	Sets the internal scale factor.

`lineJoin`	`int`	`JOIN_MITER`	Sets the decoration applied when two segments are joined. The possible values are:
			`JOIN_BEVEL`
			`JOIN_MITER`
			`JOIN_ROUND`

`lineStyle`	`float[]`	`null`	Sets the array that represents the lengths of the dash segments. The possible values are:
			`Solid`
			`Dot`
			`Dash`
			`DashDot`
			`DashDoubleDot`
			`Alternate`
			`LongDash`
			`DoubleDot`

Property Name	Type	Default Value	Description
			or an array that defines a new line style.
`lineStylePhase`	`float`	`0f`	Sets the dash pattern offset.
`lineWidth`	`float`	`5f`	Sets the line width.
`maxLineWidth`	`float`	`0`	Sets the maximum line width when zooming in.
`minLineWidth`	`float`	`0f`	Sets the minimum line width when zooming out.

`mode`	`int`	`MODE_GRADIENT`	Sets the link-drawing mode. The possible values are:
			`MODE_TEXTURE`
			`MODE_UNICOLOR`
			`MODE_GRADIENT`
			`MODE_NEON`
			Caution: Do not use the border with the mode `ilog.views.sdm.graphic.IlvGeneralLink.MODE_NEON`.

`oriented`	`boolean`	`false`	Sets the link as oriented or not oriented.
`qualityLevel`	`int`	`3`	Controls the quality of the rendering of the link. You can modify this property for faster interaction or high quality printing. The following values achieve the following effects respectively: `0`: The link is rendered as a single line only. `1`: `MODE_UNICOLOR` is forced, with no border, no wave effect, and a classic arrow. `2`: Gives the effect of `1` with a border. `3`: All, the default value. `4`: Very fine-gradient spectra. `5`: All `BasicStroke` with float value and no cache.
`wave`	`String`	`"0/0"`	Sets the wavy outline of the link. Use `0/0` to cancel the wave effect. The value is a formatted string that describes the wave. The format is `a/p`, where `a` is an `int` representing the amplitude of the wave in

Property Name	Type	Default Value	Description
			pixels and `p` is an `int` representing the period or length of the wave in pixels.

How to use network link properties in a user-defined business class

```
object."test.MyLink" {
    oriented: true;
    lineWidth: 5;
    lineStyle: "4,4,2";
    endCap: CAP_ROUND;
    wave: "1/2";
}
```

Customizing various aspects of links

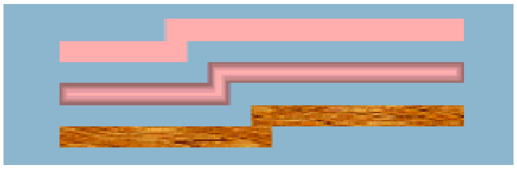
You can customize the following aspects of network links:

- *Major appearance modes*
- *An optional border*
- *End cap and join style of the stroke*
- *Curves*
- *Dashes*
- *Alternating colors*
- *Arrows*
- *Waves*
- *Animation*
- *Zoom*
- *Special effects*

Note: Color values are given as literals or hexadecimal codes. The resulting colors are shown in the figures.

Major appearance modes

The general link has three principal looks associated with the property `mode`. For the first two looks, the `foreground` property sets the main color.



Three main looks for links

The properties of the first look shown in the figure are composed as follows:

```
mode = MODE_UNICOLOR
lineWidth = 10
foreground = pink
```

The properties of the second look shown in the figure are composed as follows:

```
mode = MODE_GRADIENT
lineWidth = 10
foreground = pink
```

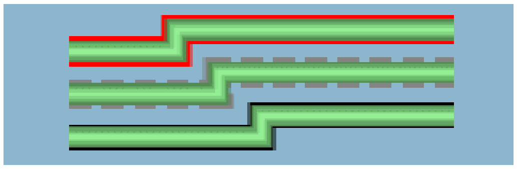
The properties of the third look shown in the figure are composed as follows:

```
mode = MODE_TEXTURE
lineWidth = 10
fillTexture = wood.png
```

The `fillTexture` property specifies the URL of an image file to use as a texture in `TEXTURE` mode. Note that the URL does not use the URL Access Service.

An optional border

A border is painted when the `borderWidth` property is greater than `0`, the default value. The default border color is black. Two other properties control the line style (such as dashes).


Links with simple borders

The properties of the top link in the figure are composed as follows:

```
lineWidth = 10
foreground = #90EE90
endCap = CAP_ROUND
borderWidth = 4
borderColor = red
```

The properties of the center link in the figure are composed as follows:

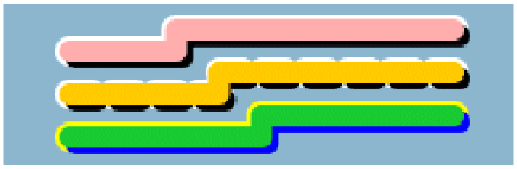
```
lineWidth = 10
foreground = #90EE90
endCap = CAP_ROUND
borderWidth = 4
borderColor = gray
borderLineStyle = "10,5"
```

`borderLineStyle` is expressed as a float array.

The properties of the bottom link in the figure are composed as follows:

```
lineWidth = 10
foreground = #90EE90
```

```
endCap = CAP_ROUND
borderWidth = 2
```


Links with more complex borders

The properties of the top link in the figure are composed as follows:

```
lineWidth = 10
mode = MODE_UNICOLOR
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
borderWidth = 4
borderColor = white
borderColor2 = black
foreground = #FFAFAF
```

The properties of the center link in the figure are composed as follows:

```
lineWidth = 10
mode = MODE_UNICOLOR
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
borderWidth = 4
borderColor = white
borderColor2 = black
foreground = #FFC800
borderLineStyle = "10,10"
```

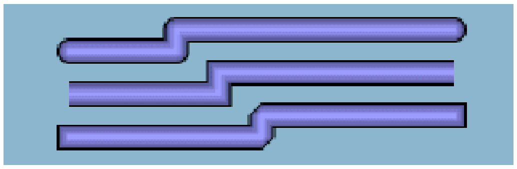
`borderLineStyle` is expressed as a float array.

The properties of the bottom link in the figure are composed as follows:

```
lineWidth = 10
mode = MODE_UNICOLOR
endCap = IlvStroke.CAP_ROUND
lineJoin = IlvStroke.JOIN_ROUND
borderWidth = 4
borderColor = yellow
borderColor2 = blue
foreground = #1EC830
```

End cap and join style of the stroke

The default stroke parameters are `JOIN_MITER` and `CAP_SQUARE`. You can change the end cap and join style, as shown in the following figure.



Different values for stroke parameters

The properties of the top link in the figure are composed as follows:

```
lineWidth = 10
endCap = IlvStroke.CAP_ROUND
lineJoin = IlvStroke.JOIN_ROUND
borderWidth = 2
foreground = #9A9AFF
```

The properties of the center link in the figure are composed as follows:

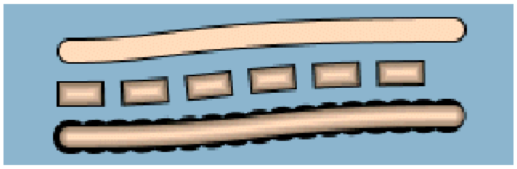
```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_MITER
borderWidth = 2
foreground = #9A9AFF
```

The properties of the bottom link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_SQUARE
lineJoin = JOIN_BEVEL
borderWidth = 2
foreground = #9A9AFF
```

Curves

The `curved` property uses the link points to feed a Bezier function which renders a curved link. Intermediate points show the path for the Bezier computation. With two points, a standard deviation applies, that is, at 1/4 before the end of the link. The curved value is a float between `0f` and `1f`. A value of `0` means no curve at all (the default), and a value of `1` means the sharpest curve. Use a value of `0.65f` for an attractive curve.


Curved links

The properties of the top link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #FFDAB9
curved = 0.65
mode = MODE_UNICOLOR
```

The properties of the center link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_SQUARE
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #FFDAB9
curved = 0.65
lineStyle = [10,20]
```

`lineStyle` is expressed as a float array.

The properties of the bottom link in the figure are composed as follows:

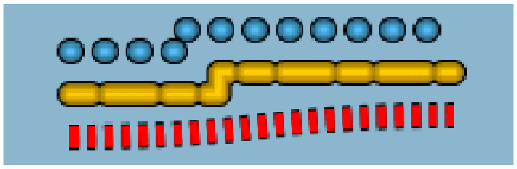
```
lineWidth = 10
endCap = CAP_SQUARE
lineJoin = JOIN_BEVEL
borderWidth = 6
foreground = #FFDAB9
curved = 0.65
borderLineStyle = [1,10]
```

`borderLineStyle` is expressed as a float array.

Dashes

Dashes provide interesting effects together with `endCap` values. Dashes are controlled by the `lineStyle` property. They are expressed as a float array. Alternate entries in the array represent lengths of the opaque and transparent segments of the dashes.

Note that the dash specification also applies to the border unless the `borderLineStyle` property overrides it.


Links with dashed line styles

The properties of the top link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #55BEF3
lineStyle = [1,15]
```

The properties of the center link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #FFC800
lineStyle = [10,8,20,8]
```

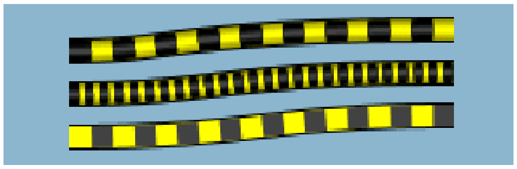
The properties of the bottom link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = red
mode = MODE_UNICOLOR
lineStyle = [4,4]
curved = 0.65
```

Alternating colors

The `alternateColor` property toggles the link in alternate mode with the specified color. The segment size equals the thickness of the link. Otherwise, the segment size can be specified with the `lineStyle` property.

The `lineStylePhase` property sets the initial offset. If no `lineStyle` is specified, the phase is proportional to twice the line width. In the following figure, the bottom link starts the alternate color one segment later than the top one.



Links with alternate colors

The appropriate values for the links are as follows.

The properties of the top link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = yellow
alternateColor = darkGray
curved = 0.65
```

The properties of the center link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = yellow
alternateColor = darkGray
curved = 0.65
lineStyle = [4,3]
```

`lineStyle` is expressed as a float array.

The properties of the bottom link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = yellow
alternateColor = darkGray
curved = 0.65
lineStylePhase = 1
mode = MODE_UNICOLOR
```

Arrows

There are four modes for representing an arrow controlled by the `arrowMode` property.

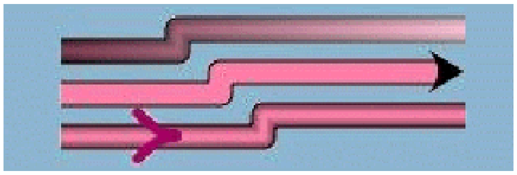
- The `ARROW_FILL` mode value (the default) draws a filled triangular arrow.

- The `ARROW_OPEN` mode value draws a two-arm arrow.
- The `ARROW_GRADIENT` mode value shows an oriented link by smoothly varying the luminosity along the link. The link appears darker near the source and brighter near the target.
- The `ARROW_DECORATION` mode value delegates the task of displaying the arrow to one of the link decorations.

In the first two modes, the `arrowPosition` property controls the position of the arrow along the link. Its value is a `float` between `0` and `1`. A value of `0` means the start of the link. A value of `1` means the end of the link (the default). The arrow direction is aligned with the link segment below it.

The property `arrowRatio` controls the size of the arrow, which is proportional to the width of the link. A float value of `0.5` means the arrow is the same size as the link. The default value `1` means the arrow is twice as wide as the link. This property applies to all four arrow modes.

The default color of an arrow is black. The color can be set by the property `arrowColor`.


Links with arrows

The properties of the top link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #FF82AB
arrowMode = ARROW_GRADIENT
oriented = true
```

The properties of the center link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #FF82AB
arrowMode = ARROW_GRADIENT
oriented = true
mode = MODE_UNICOLOR
arrowMode = ARROW_FILL
```

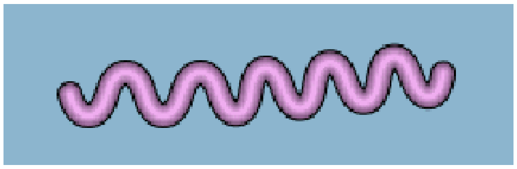
The properties of the bottom link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #FF82AB
arrowMode = ARROW_OPEN
oriented = true
arrowColor = #A3056E
arrowPosition = 0.2
```

Waves

A wave effect is very effective for representing a wireless connection. The wave specification consists of two numbers in pixels representing the wave amplitude and its period. The property type is `String`, where two integers separated by a forward slash or solidus (/) represent the amplitude and period respectively. The effect is rendered best with straight lines, but remains compatible with any shape.

The wave effect can also be combined with dashes, border, arrow, and so on.


Link with wave effect

The properties of the link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
borderWidth = 2
foreground = #EEAEEE
wave = 20/30
```

Animation

A link with dashes (see the `lineStyle` property) or alternate colors (see the `alternateColor` property) can be animated. The animation consists of incrementing the `lineStylePhase` value at regular intervals, so that the dash pattern is shifted at each animation frame. The current implementation updates every 500 ms.

The `animateSpeed` property controls animation of the link and how much the phase is incremented. If the value is `0`, the animation is stopped. Otherwise, the value must be between `0f` and `1f` and represents a fraction of the dash pattern length. For example, `0.1` means that ten frames elapse before seeing the first frame again. Note that `0.9` represents the same increment but in the reverse direction.

Zoom

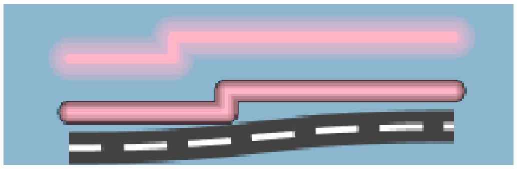
The `maxLineWidth` property sets the maximum width of the line. The link is zoomable only if one edge node is zoomable. Note that the link border cannot be zoomed.

The `minLineWidth` property sets the minimum width of the line.

Special effects

`MODE_NEON` is a minor mode that is a variation of `MODE_GRADIENT`. It displays the link with transparent colors, giving a glowing effect. (This effect works best with larger links and with darker backgrounds.) The border is automatically disabled in this mode. Neon mode could be used to mark link selection, for example.

The preceding figures show some of the diversity offered in the appearance of links. By combining some of the properties, you can obtain some special effects, such as the *road* link in the following figure, where the large border is the same gray color as the link foreground and the white value of the `alternateColor` property looks like the center line of a road.


Links with special effects

The style of the top link in the figure is composed as follows:

```
lineWidth = 20
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
foreground = #FFB5C5
mode = MODE_NEON
```

The properties of the center link in the figure are composed as follows:

```
lineWidth = 10
endCap = CAP_ROUND
lineJoin = JOIN_ROUND
foreground = #FFB5C5
```

The properties of the bottom link in the figure are composed as follows:

```
lineWidth = 15
endCap = CAP_BUTT
lineJoin = JOIN_ROUND
borderWidth = 12
foreground = #404040
borderColor = #404040
```

```
alternateColor = white
curved = 0.65
mode = MODE_UNICOLOR
LineStyle = [20,10]
```

`lineStyle` is expressed as a float array.

Customizing a link label

The label decoration for a network link is customized using the same properties as for the labels of predefined business objects.

How to use label properties in a user-defined business class

```
object."test.MyNetworkElement" {
  labelBackground: white;
  labelForeground: blue;
  labelPosition: Right;
  label: @name;
  labelVisible: true;
}
```

For a complete list of the available link properties, refer to *Customizing the label of a business object*.

Customizing tooltips of user-defined business objects

The tooltips of user-defined business objects can be customized using the same properties as used for the tooltips of the predefined business objects.

How to use tooltip properties in a user-defined business class

```
object."test.MyNetworkElement" {
  toolTipText: @name;
}
```

For a complete list of the available properties for tooltips, refer to *Changing the font of all labels*.

For details about using a graphic as a tooltip, refer to *How to use a JComponent to generate a tree node representation* and *How to use an IlvGraphic to generate a network node representation*.

Customizing links

Describes what links, link sets, and link bundles are, how they are represented, and how different aspects of their representations can be customized.

In this section

Links
Defines links and describes the support provided for links, link sets, and link bundles.

Representing links
Describes the way that links are represented.

Customizing link representations
Provides details about the CSS properties that you can use to customize the representation of links.

Changing the representation of individual links
Describes how to customize links by making the representation of a specific link different from the others.

Customizing the link information cluster
Describes how to customize the link information cluster which groups the decorations displayed on a link.

Customizing link media
Describes how to create a new link medium.

Customizing link technology
Describes how to create a new link technology. This operation is similar to adding a new medium to links.

Customizing various aspects of links
Lists the properties available for customizing different aspects of links in CSS.

Customizing link tiny types
Describes how to customize the tiny types for links with examples.

Customizing link sets
Discusses link set representation and describes how to set the interlink distance within a link set.

Customizing link bundles
Describes how link bundles are represented and how to customize this representation.

Customizing link set and link bundle tiny types
Describes how to customize the tiny types for link sets and link bundles with examples.

Links

Links are predefined business objects that represent a connection between two network elements.

ILOG JViews TGO provides the following predefined link support:

Links
Links are used to display the transmission elements making up the network lines. Links feature the same dynamic display as network elements. For more information on links, see Links.

Link sets
Link sets let you group together links between two nodes, so that the graph layout cannot insert a link that is not in the link set between them or have them follow different paths. You can fix the order of the links in the set and specify the distance that separates two links.

For more information on link sets, see Link sets.

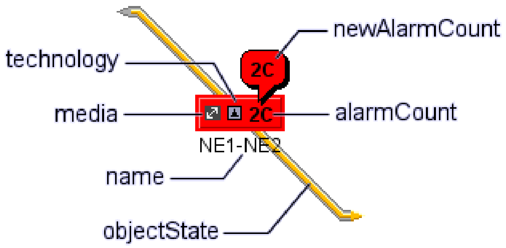
Link bundles
Link bundles let you group together links between two nodes. You can collapse a link bundle to show only a single link (an overview link). The single link has an icon, when you click it, it causes the link bundle to expand and show the child links.

For more information on link bundles, Link bundles.

Representing links

The graphic representation of a link is based on the information that is available in the business model. Each decoration that is created depends on an attribute and on properties that can be customized through CSS. Figure *Link with attributes* shows a link with the following attribute set:

- Media: Fiber
- Name: NE1-NE2
- Object State: BiSONET Object State



Link with attributes

Customizing link representations

Some properties are mapped, which means that they are computed on the basis of the state and alarm information set in the object (column *Set*).

The properties can be divided into three categories:

♦ Properties applying to the link base

♦ Properties applying to the arrows on link base elements

♦ Properties applying to the inner line of link base elements

CSS properties applying to the link base

Property Name	Type	Set	Default Value	Description
`background`	`Color`	Yes	Transparent (`null`)	Specifies the background color; if this value is `null`, it is interpreted as a transparent color
`foreground`	`Color`	Yes	28% gray in the `IltObject` class style	Specifies the foreground color of the base.
`borderColor`	`Color`	Yes	10% gray	Specifies the color of the border lines of the linear base.
`borderColor2`	`Color`	Yes	60% gray	Specifies the color of the background of the border lines. It is used only if the line style is not solid.
`borderWidth`	`Float`	Yes	1 pixel	Determines the width of each border. If the `borderWidth` property is zero (0), no border line will be drawn.
`borderLineStyle`	`float[]`	Yes	`null` (Solid)	Stores the line style of the border lines.
`reliefBorders`	`Boolean`	Yes	`true`	Discriminates between two identical border lines (`false` value) and two differently colored borders to produce a relief effect.
`lineStyle`	`float[]`	Yes	`null` (Solid)	Defines the style of the linear base.
`lineWidth`	`Float`	Yes	`5f`	Defines the width of the center line of the base in the case of a linear base.
`forcedWidth`	`Float`	No	0	If the value of this property is not zero, the width of the center line is augmented or

Property Name	Type	Set	Default Value	Description
				diminished so that the total width equals the width stored in `forcedWidth`.

CSS properties applying to arrows on link base elements

Property Name	Type	Set	Default Value	Description
`hasFromArrow` `hasToArrow`	`boolean`	Yes	false	These two properties determine whether there is an arrow or not at the beginning and end of the link
`fromArrowSize` `toArrowSize`	`float`	No	8 pixels	Define the length of the arrow at the beginning and end of the link
`fromArrowColor` `toArrowColor`	`Color`	Yes	28% gray	Define the color of the center of the arrow, if the arrow has a border, or the whole arrow, if the arrow has no border
`fromArrowReliefBorders` `toArrowReliefBorders`	`boolean`	Yes	`true`	Gives a relief effect to the arrow border
`fromArrowBorderColor` `toArrowBorderColor`	`Color`	Yes	10% gray	Defines the color of the arrow border
`fromArrowBorderColor2` `toArrowBorderColor2`	`Color`	Yes	60% gray	If `FromArrowReliefBorders` (or `ToArrowReliefBorders`) is `true` , this property defines the dark color of the relief effect

CSS properties applying to the inner line of link base elements

Property Name	Type	Set	Default Value	Description
`innerLineWidth`	`float[]`	Yes	`null` (Solid)	Denotes the width of the center line of the inner part of the link base, if any.
`innerBorderWidth`	`float`	Yes	0	Defines the width of the border line of the inner part of the link base, if any.
`innerLineStyle`	`float[]`	Yes	`Solid`	Defines the style of the inner line of the link base, if any
`innerForeground`	`Color`	Yes	28% gray	Defines the foreground color of the inner line of the link base, if any
`innerBackground`	`Color`	Yes	transparent (`null`)	Defines the background color of the inner line of the link base, if any
`innerReliefBorders`	`boolean`	Yes	`true`	Discriminates between two identical border lines (`false` value) and two

Property Name	Type	Set	Default Value	Description
				differently colored borders to produce a relief effect for the inner line, if any
`innerBorderLineStyle`	`float[]`	Yes	`Solid`	Stores the line style of the border lines of the inner line, if any
`innerBorderColor`	`Color`	Yes	10% gray	Specifies the color of the border lines of the inner line, if any
`innerBorderColor2`	`Color`	Yes	60% gray	Specifies the color of the background of the border lines of the inner line, if any. It is used only if the line style is not solid.

Note: The inner line set of properties apply to links that use the IltBiSONET object state. You can also customize other links with an inner line.

Changing the representation of individual links

By default, the representation of the base of a link is determined by its base style, which is itself calculated from the state of the link.

How to change the graphical representation of a link (using CSS)

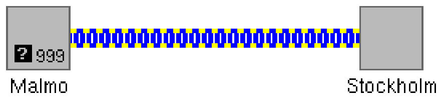
You might want to define specific link representations, regardless of the base styles managed by JViews TGO (when you do not use states, for example). In this case, you can change the representation of the link using cascading style sheets.

The following example shows an extract of a CSS file which changes the representation of a given link. Note that to make this possible, you must use the object state `IltAlarmObjectState`, `IltTrapObjectState`, or no object state at all for this specific link, since using another state system would force the appearance of the link base.

```
#myLink {
  centerWidth: 6;
  foreground: blue;
  background: '';
  lineStyle: "3.000001, 1.000001";
  borderWidth: 3;
  borderColor: yellow;
  borderColor2: blue;
  reliefBorders: false;
  borderLineStyle: Dot;
}
```

For details on how to load a configuration like this one in a network component, see How to load a CSS file in a network component.

The link is represented as a hatched central line with a blue background color and two identical border lines hatched in yellow on a transparent background, as shown below:



Customizing a link representation

Changing the representation of links based on states and llarms

For details about how to customize the links that are based on object and alarm states, refer to *Customizing the object representation based on states*.

Customizing link states and alarms

For information on how to customize the link representation when it contains states and alarms, refer to *Customizing object and alarm states of predefined business objects*. The information that you will find in that section is valid for all `IltObject` classes.

Customizing the link information cluster

The link information cluster designates the group of decorations displayed in the middle of the link. This cluster can show the name, media, technology, alarm count, or any combination of them grouped in a plinth.

Link information cluster properties

The properties listed in the following table allow you to customize the link information cluster.

CSS properties for the link information cluster

Property Name	Type	Set	Default Value	Description
`decorationsOffset`	`IlpPoint`	No	`0,0`	Defines the offset used to attach the link information cluster to the center of the link.
`plinthColor`	`Color`	Yes	67% grey	Denotes the color of the plinth.
`plinthBrightColor`	`Color`	Yes	87% grey	Denotes the brighter color of the plinth.
`plinthDarkColor`	`Color`	Yes	33% grey	Denotes the darker color of the plinth.
`plinthVisible`	`boolean`	No	`true`	Denotes whether the plinth is visible or not.
`plinthVerticalMargin`	`int`	No	`1`	Defines the vertical margin between the plinth and its contents.
`plinthHorizontalMargin`	`int`	No	`4`	Defines the horizontal margin between the plinth and its contents.
`useAlarmColorForBase`	`boolean`	No	`false`	Denotes whether or not to use the alarm color to display the link base. If the alarm color is not used, the regular properties linked to each primary state are used.

How to customize the link information cluster

The following CSS extract shows how to customize the position of a link information cluster. By default, the cluster is positioned in the center of the middle segment of links. You can fine-tune this position by defining an offset that will be applied relative to the default position.

```
object."ilog.tgo.model.IltLink" {
  decorationsOffset: "5,0";
}
```

Customizing link media

This operation is similar to adding a new function or family for network elements.

How to create a new link medium (using the API)

To create a new link medium, you must extend the `IltLink.Media` enumeration. You can map the new medium to an icon.

```
IltLink.Media satellite = new IltLink.Media("satellite");
IlpContext context = IltSystem.GetDefaultContext();
IlpImageRepository imageRep = context.getImageRepository();
Image satelliteImage = imageRep.getImage("sat.png");
IltSettings.SetValue("Link.Media.satellite.Icon", satelliteImage);
```

How to create a new link medium (using CSS)

You can create new link media by using global CSS settings.

```
setting."ilog.tgo.model.IltLink"{
  media[0]: @+linkMedia0;
}
Subobject#linkMedia0 {
  class: 'ilog.tgo.model.IltLink.Media';
  name: "satellite";
}
```

For more information, see *Using global settings*.

How to customize a link medium (using CSS)

You can obtain the same customization as with the API by using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right object in the system. The CSS property to customize here is `icon`.

```
setting."ilog.tgo.model.IltLink.Media"[name="satellite"] {
   icon: '@|image("sat.png")';
}
```

The following properties allow you to customize the representation of link media:

CSS properties for link media

Property Name	Type	Set	Default Value	Description
`mediaVisible`	`boolean`	No	`true`	Defines whether the media icon is visible or not.
`mediaIcon`	`Image`	Yes	The image defined for each media value in `IltSettings`	Defines the image to be used to represent the media in a given link.

How to hide link media

You can show or hide link media using the property `mediaVisible` as follows:

```
object."ilog.tgo.model.IltLink" {
  mediaVisible: false;
}
```

Tooltips for link media

Link media are associated with a default tooltip which is retrieved from the JViews TGO resource bundle (see About internationalization in the *Context and Deployment Descriptor* documentation).

The resource that applies to link media is:

♦ `ilog.tgo.Link_Media_<MEDIA NAME>_ToolTip`: link media tooltips

For the predefined link media, you can edit this value directly in the JViews TGO resource bundle file.

For newly created media values, the tooltip information will also be retrieved from this resource bundle. As you declare new media, register the corresponding entry in the resource bundle file so that tooltips can be automatically displayed.

Considering that you have created the link media "`satellite`", you should declare the entry in the resource bundle file as follows:

♦ `ilog.tgo.Link_Media_satellite_ToolTip=Satellite`

Customizing link technology

How to create a new link technology (using the API)

To create a new link technology, you must extend the `IltLink.Technology` enumeration.

You map the new technology to an icon and a color.

```
IltLink.Technology pSwitching = new IltLink.Technology("PacketSwitching");
IlpContext context = IltSystem.GetDefaultContext();
IlpImageRepository imageRep = context.getImageRepository();
Image pSwitchingImage = imageRep.getImage("pSwitching.png");
IltSettings.SetValue("Link.Technology.PacketSwitching.Icon",
                   pSwitchingImage);
IltSettings.SetValue("Link.Technology.PacketSwitching.Color", new
                    Color(128,196,210));
```

How to create a new link technology (using CSS)

You can create new link technologies by using global CSS settings.

```
setting."ilog.tgo.model.IltLink" {
   technology[0]: @+linkTechnology0;
}
Subobject#linkTechnology0 {
   class: 'ilog.tgo.model.IltLink.Technology';
   name: "PacketSwitching";
}
```

For more information, see *Using global settings*.

Customizing link technologies

You can obtain the same customization as with the API by using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right object in the system. The CSS properties to customize here are icon and color.

```
setting."ilog.tgo.model.IltLink.Technology"[name="PacketSwitching"] {
    icon: '@|image("pSwitching.png")';
    color: '#80C4D2';
}
```

The following properties allow you to customize the representation of link technology:

CSS properties for link technology

Property Name	Type	Default Value	Description
`technologyIconVisible`	`boolean`	`false`	Defines whether the link technology icon is visible or not.
`technologyColorVisible`	`boolean`	`false`	Defines whether the link technology color is mapped to the link base or not.
`technologyIconBorder`	`int`	`1`	Defines a border around the technology icon, in pixels.
`technologyIcon`	`Image`	The image defined for each technology value in `IltSettings`	Defines the image to be used to represent the technology in a given link.
`technologyColor`	`Color`	The color defined for each technology value in `IltSettings`	Defines the color to be used to represent the technology in a given link.

How to show link technology

By default, the link technology is hidden, but you can show or hide it using the properties `technologyIconVisible` and `technologyColorVisible` as follows:

```
object."ilog.tgo.model.IltLink" {
   technologyIconVisible: true;
   technologyColorVisible: true;
}
```

Note that the primary state color of a link will override the technology color. You can force the link technology color by using the property `baseStyleEnabled` as follows:

```
object."ilog.tgo.model.IltLink" {
   baseStyleEnabled: false;
}
```

How to customize the link technology tooltip

The resource that applies to link technology is:

`ilog.tgo.Link_Technology_<TECHNOLOGY NAME>_ToolTip`: link technology tooltip

For the predefined link technology, you can edit this value directly in the JViews TGO resource bundle file.

For newly created technology values, the tooltip information will also be retrieved from this resource bundle. As you declare new technologies, register the corresponding entry into the resource bundle file so that tooltips can be automatically displayed.

Supposing that you have created the link technology PacketSwitching, you should declare the entry in the resource bundle file as follows:

```
ilog.tgo.Link_Technology_PacketSwitching_ToolTip=Packet Switching
```

Customizing various aspects of links

Customizing link names

You can customize the display of link names through the following CSS properties:

- `label`
- `labelVisible`
- `labelFont`
- `labelForeground`
- `labelBackground`
- `labelPosition`
- `labelAntialiasing`
- `labelSpacing`
- `labelWrappingMode`
- `labelWrappingWidth`
- `labelWrappingHeight`
- `labelMargin`
- `labelAlignment`
- `labelBorderColor`
- `labelDirection`
- `lineSpacing`
- `minLabelZoom`
- `maxLabelZoom`
- `labelScaleFactor`

Refer to *Customizing the label of a business object* for details on these properties and how to use them.

Customizing link tooltips

You can customize the display of link tooltips through the following CSS properties:

- `toolTipGraphic`
- `toolTipText`

For details on these properties and how to use them, refer to *Changing the font of all labels* .

Adding new decorations to links

You can add new decorations to the predefined business objects representation.

For details, refer to *How to add new decorations to network and equipment nodes*.

Customizing link port configuration

You can customize the way links connect to nodes through the following CSS properties:

- `linkPorts`
- `fromPort`
- `toPort`

Refer to *Customizing node and link layouts* for details on these properties and how to use them.

Customizing link label layout

You can fine-tune the layout of link labels in a network through the CSS properties described in *Customizing link label layout*.

For details about the label layout, refer to Label layout.

Customizing link tiny types

In the tree and table components, link objects are represented as tiny objects. Each link tiny type can be associated with a tiny base renderer that draws the tiny graphic representation.

JViews TGO allows you to customize the tiny type representation by using one of the predefined base renderer factories, such as `IltTinyImageBaseRendererFactory` or `IltTinySVGBaseRendererFactory`, or by creating your own implementation of `IltTinyBaseRenderer`. The principle to create a new `IltTinyBaseRenderer` is the same as to create a new `IltNEBaseRenderer`. For details, refer to *Extending the class IltNEBaseRenderer*.

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

How to create and register a link tiny type (using the API)

```
IltObject.TinyType MyTinyType = new IltObject.TinyType("MyTinyType");
IltSettings.SetValue("Link.TinyType.MyTinyType.Renderer",
  new IltTinyImageBaseRendererFactory(YOUR_IMAGE, YOUR_IMAGE_PARAMETERS));
```

How to create and register a link tiny type (using CSS)

You can also create new tiny types by using global CSS settings (:

```
setting."ilog.tgo.model.IltObject"{
   tinyTypes[0]: @+tinyType0;
}
Subobject#tinyType0 {
  class: 'ilog.tgo.model.IltObject.TinyType';
  name: "MyTinyType";
}
```

For more information, see *Using global settings* in *Using Cascading Style Sheets*.

How to customize a link tiny type (using CSS)

You can customize the renderer using global CSS settings. The CSS property to customize here is `linkTinyRenderer`. In the following example, the name of the renderer factory class that is included in the search path is `MyLinkTinyRendererFactory`.

```
setting."ilog.tgo.model.IltLink.TinyType"[name="MyTinyType"] {
   tinyRenderer: @+linkTinyRendererFactory;
}
Subobject#linkTinyRendererFactory {
   class: 'MyLinkTinyRendererFactory';
}
```

Customizing link sets

Link sets do not have a graphic representation in the network and equipment components. Instead, they are used to group together multiple links that may exist between two nodes, so that the graph layout cannot insert a link that is not in the link set or have the links follow different paths.

You can customize a link set through the CSS property listed in the following table.

CSS property for link sets

Property Name	Type	Default Value	Description
`linkDistance`	`float`	2	Defines the distance between the links that are part of the link set

How to set the link distance in a link set

```
object."ilog.tgo.model.IltLinkSet" {
  linkDistance: 4;
}
```

Customizing link bundles

Representing link bundles and customizing different representations

A link bundle has two graphic representations. The *collapsed* representation shows the bundle as a single link (overview link) connecting two end points. In this representation, all CSS properties described for links also apply to link bundles (see *Representing links*).

When the link bundle is *expanded*, the overview link is replaced by the child links, which are then represented as a link set. In the expanded representation, the link bundle can be customized through the property shown in the following table.

CSS properties for link bundles

Property Name	Type	Default Value	Description
`linkDistance`	`float`	2	Defines the distance between the links that are part of the link bundle.

How to customize a link bundle representation through CSS

The following application is provided as part of the ILOG JTGO demonstration software: ***<installdir>* /samples/network/links/index.html**

It shows how to customize a link bundle representation in its collapsed and expanded forms.

The following CSS extract customizes the width of all link bundles to be larger than the standard links. It also customizes the link bundle identified as `linkBundle78` to display a distance of 5 pixels between its inner links:

```
object."ilog.tgo.model.IltLinkBundle" {
  forcedWidth: 10;
}

#linkBundle78 {
  linkDistance: 5;
}
```

Customizing the container icon of a link bundle

You can also customize the visibility of the container icon that is automatically displayed and allows you to expand/collapse the link bundle. You can customize the container icon using the CSS properties listed in the following table.

CSS properties for link bundle container icons

Property Name	Type	Default Value	Description
`containerStatusVisible`	`boolean`	`true`	Indicates whether the container icon collapses or expands and whether it should be visible or not in an object
`containerExpansionIconVisible`	`boolean`	`true`	Indicates whether the in-place expansion icon is added to the overview object of a container. (The value of `containerStatusVisible` must be set to `true`.)
`containerCollapseIconVisible`	`boolean`	`true`	Indicates whether the collapse icon is added to the child objects of a link bundle. (The value of `containerStatusVisible` must be set to `true`.)

Customizing link set and link bundle tiny types

Link set and link bundle tiny types are customized in a similar way to the link tiny type (see *Customizing link tiny types*.

How to customize a link set and a link bundle tiny type (using the API)

```
IltSettings.SetValue("LinkSet.TinyType.Standard.Renderer",
  new IltTinyImageBaseRendererFactory(YOUR_IMAGE, YOUR_IMAGE_PARAMETERS));

IltSettings.SetValue("LinkBundle.TinyType.Standard.Renderer",
  new IltTinyImageBaseRendererFactory(YOUR_IMAGE, YOUR_IMAGE_PARAMETERS));
```

How to customize a link set and a link bundle tiny type (using CSS)

The CSS property to use for customizing is `tinyRenderer`.

```
setting."ilog.tgo.model.IltLinkSet.TinyType"[name="Standard"] {
   tinyRenderer: @+linkSetTinyRendererFactory;
}
Subobject#linkSetTinyRendererFactory {
   class: 'MyLinkSetTinyRendererFactory';
}

setting."ilog.tgo.model.IltLinkBundle.TinyType"[name="Standard"] {
   tinyRenderer: @+linkBundleTinyRendererFactory;
}
Subobject#linkBundleTinyRendererFactory {
   class: 'MyLinkBundleTinyRendererFactory';
}
```

Customizing groups

Describes what groups are and how to customize groups and each attribute/decoration of a group representation.

In this section

Groups
Describes what groups are and lists the predefined groups.

Representing groups and attributes
Describes how groups with attributes are represented.

Customizing group representations
Provides details about the CSS properties that you can use to customize the representation of groups.

Customizing various aspects of groups
Lists the properties for customizing various aspects of groups.

Groups

Groups are predefined business objects that are used to represent a set of network resources grouped logically or geographically.

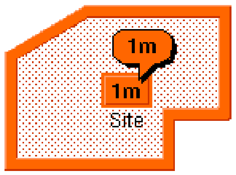
ILOG JViews TGO provides the following predefined groups:

Polygonal groups

Polygonal groups are defined by the class `IltPolyGroup`. Polygonal groups are very useful for dividing a network into regions and associating those regions with topographic zones visible on a map.

The shape of a polygonal group is defined by the class `IlpPolygon`. This class describes a closed polyline made up of an array of points. This polyline can have any number of sides.

Polygonal groups have a semitransparent background (through which a background map can be seen) and a thick outline. When alarms or statuses are displayed, they are grouped in an information cluster that is positioned at the center of the polygon, as shown in *Polygonal group with information cluster*.



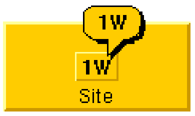
Polygonal group with information cluster

Rectangular groups

Rectangular groups are defined by the class `IltRectGroup`. Rectangular groups are generally used to hold network elements located in the same place such as a site, a building, or a city.

The shape of a rectangular group is defined by the class `IlpRect`, which describes a rectangle. Rectangular groups can be resized to create any kind of rectangular container.

Rectangular groups are represented by opaque relief rectangles as shown in *Rectangular group with information cluster*. When alarms are displayed, they are grouped in an information cluster that is positioned at the center of the rectangle.



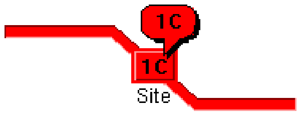
Rectangular group with information cluster

Linear groups

Linear groups are defined by the class `IltLinearGroup`. Linear groups represent a linear collection of objects and can be used to display, for example, all the repeaters between two line termination network elements.

The shape of a linear group is defined by the class `IlpPolyline`. This class describes an open polyline made up of an array of points.

When alarms or secondary states are displayed on a linear group, an information cluster appears at the center of its median segment. The median segment is the segment containing the midpoint of the shape.



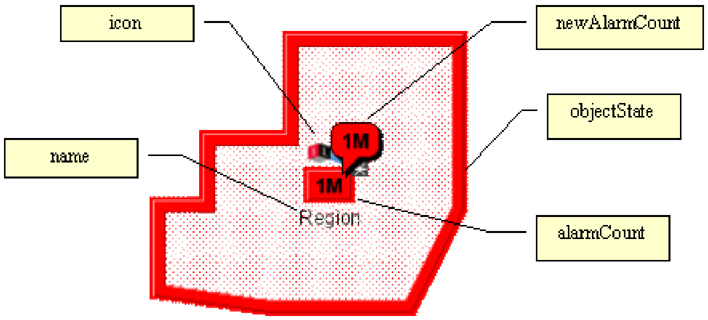
Linear group with information cluster

Representing groups and attributes

The graphical representation of a group is based on the infomation that is available in the business model. Each decoration that is created depends on an attribute and on properties that can be customized through CSS.

The following figure shows a polygonal group with the following attribute set:

- ♦ Name: Region
- ♦ Icon: ilogic.png
- ♦ Object State: OSI object state



Group with attributes

The graphic rendering of a group object is optimized for performance using an offscreen buffered image technique that minimizes the complex polygon computations by pre-rendering the graphic object in memory. The extra memory required is proportional to the size and number of visible objects in the Network view.

To disable the offscreen optimizations for the group representation:

- ♦ Set the `ilog.tgo.polygon.offscreenCache` system property to `false` at initialization time.
- ♦ Use the following API call:

```
IltSettings.SetValue("OffscreenCache.Polygon", Boolean.FALSE);
```

For more information on how to declare system properties for Java™ applications and applets, refer to the Java Runtime documentation.

Customizing group representations

Some properties are mapped, which means that they are computed on the basis of the state and alarm information set in the object (column *Set*).

The properties can be divided into three categories:

- ♦ *Polygonal group properties*
- ♦ *Rectangular group properties*
- ♦ *Linear group properties*

Polygonal group properties

The following properties are used to draw the interior of a polygonal group:

CSS properties for the interior of polygonal groups

Property Name	Type	Set	Default Value	Description
`foreground`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	28% grey	Denotes the foreground color of the base of an object.
`background`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	`null`	Denotes the background color of the base of an object.

`fillStyle`	`ilog.util.IlFillStyle`	No	`IlFillStyle.PATTERN`	Denotes the style used to fill the base.
				
				Possible values are:
				
				`IlFillStyle.NO_FILL`
				
				`IlFillStyle.SOLID_COLOR`
				
				`IlFillStyle.LINEAR_GRADIENT`
				
				`IlFillStyle.RADIAL_GRADIENT`
				
				`IlFillStyle.TEXTURE`
				
				`IlFillStyle.PATTERN`

`fillPattern`	`IlPattern`	Yes, if `baseStyleEnabled` is true	`Dots`	Denotes the pattern used to fill the base of an object. This property is only used if

Property Name	Type	Set	Default Value	Description
		No, otherwise		`fillStyle` is set to `IlFillStyle.PATTERN`. Possible values are: `Dots` `GroupFilling` `ThinHatching`
`fillTexture`	`Image`	No	`null`	Denotes the texture used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.TEXTURE`.
`fillAngle`	`float`	No	`0`	Denotes the angle (in degrees) of the gradient used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.RADIAL_GRADIENT` or `IlFillStyle.LINEAR_GRADIENT`.
`fillStart`	`float`	No	`0`	Denotes the position where the gradient of an object starts, that is, where the color is the one defined by the property `foreground`. This property is only used if `fillStyle` is set to `IlFillStyle.RADIA_GRADIENT` or `IlFillStyle.LINEAR_GRADIENT`.
`fillEnd`	`float`	No	`100`	Denotes the position where the gradient of an object ends, that is, where the color is the one defined by the property `background`. This property is only used if `fillStyle` is set to `IlFillStyle.RADIA_GRADIENT` or `IlFillStyle.LINEAR_GRADIENT`.

The following properties are used to draw the outline of a polygonal group. A border is displayed around the outline of the group.

CSS properties for the outline of polygonal groups

Property Name	Type	Set	Default Value	Description
`lineStyle`	`float[]`	Yes, if `baseStyleEnabled` is true No, otherwise	`null` (Solid)	Denotes the line style used to display the outline of a polygonal group.
`outlineVisible`	`boolean`	No	`true`	Denotes whether the outline is visible or not.
`outlineColor`	`Color`	Yes	grey, or the alarm color, if any.	Denotes the color of the outline.
`outlineInside`	`boolean`	No	`false`	Denotes whether the outline of the polygon is drawn inside the shape or symetrically on either side of the shape edge.
`outlineOffset`	`IlpPoint`	No	`null`	Denotes the distance between the edge of the shape and the outline of the polygon, when

Property Name	Type	Set	Default Value	Description
				outlineInside is set to true.
outlineWidth	float	No	8	Denotes the width of the polygon outline.
reliefThickness	float	No	1	Denotes the width of the relief on the polygon outline.
borderColor	Color	Yes, if baseStyleEnabled is true No, otherwise	10% grey	Denotes the primary color of the base border.
borderColor2	Color	Yes, if baseStyleEnabled is true No, otherwise	60% grey	Denotes the secondary color of the base border.
borderWidth	float	Yes, if baseStyleEnabled is true No, otherwise	1	Denotes the width of the base border.
reliefBorders	boolean	Yes, if baseStyleEnabled is true No, otherwise	true	Denotes whether the base border is drawn in relief or not.
borderLineStyle	float[]	Yes, if baseStyleEnabled is true No, otherwise	null (Solid)	Denotes the line style used to draw the base border.

How to customize the representation of polygonal groups

The following CSS extract shows how to customize the graphic representation of a polygonal group to display an empty group with a thinner outline:

```
object."ilog.tgo.model.IltPolyGroup" {
  fillStyle: NO_FILL;
  outlineWidth: 4.0;
  outlineInside: false;
}
```

Rectangular group properties

The following properties are used to draw the interior of a rectangular group:

CSS properties for the interior of rectangular groups

Property Name	Type	Set	Default Value	Description
`foreground`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	28% grey	Denotes the foreground color of the base of an object.
`background`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	`null`	Denotes the background color of the base of an object.

`fillStyle`	`ilog.util.IlFillStyle`	No	`IlFillStyle.PATTERN`	Denotes the style used to fill the base of an object.
				
				Possible values are:
				
				`IlFillStyle.NO_FILL`
				
				`IlFillStyle.SOLID_COLOR`
				
				`IlFillStyle.LINEAR_GRADIENT`
				
				`IlFillStyle.RADIAL_GRADIENT`
				
				`IlFillStyle.TEXTURE`
				
				`IlFillStyle.PATTERN`

`fillPattern`	`IlPattern`	Yes, if `baseStyleEnabled` is true No, otherwise	`null` (Solid)	Denotes the pattern used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.PATTERN`.
`fillTexture`	`Image`	No	`null`	Denotes the texture used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.TEXTURE`.
`fillAngle`	`float`	No	`0`	Denotes the angle (in degrees) of the gradient used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.RADIAL_GRADIENT` or

Property Name	Type	Set	Default Value	Description
				IlFillStyle. LINEAR_GRADIENT.
fillStart	float	No	0	Denotes the position where the gradient of an object starts, that is, where the color is the one defined by the property foreground. This property is only used if fillStyle is set to IlFillStyle. RADIA_GRADIENT or IlFillStyle. LINEAR_GRADIENT.
fillEnd	float	No	100	Denotes the position where the gradient of an object ends, that is, where the color is the one defined by the property background. This property is only used if fillStyle is set to IlFillStyle. RADIA_GRADIENT or IlFillStyle. LINEAR_GRADIENT.

The following properties are used to draw the border of a rectangular group.

CSS properties for the border of rectangular groups

Property Name	Type	Set	Default Value	Description
`reliefThickness`	`float`	No	`2`	Denotes the width of the base border when the property reliefBorders is `true`.
`borderColor`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	10% grey	Denotes the primary color of the base border.
`borderColor2`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	60% grey	Denotes the secondary color of the base border.
`borderWidth`	`float`	Yes, if `baseStyleEnabled` is true No, otherwise	1	Denotes the width of the base border when `reliefBorders` is set to `false`.
`reliefBorders`	`boolean`	Yes, if `baseStyleEnabled` is true No, otherwise	true	Denotes whether the base border is drawn in relief or not.
`borderLineStyle`	`float[]`	Yes, if `baseStyleEnabled` is true No, otherwise	`null` (Solid)	Denotes the line style used to draw the base border.
`borderPattern`	IlPattern	Yes, if `baseStyleEnabled` is true No, otherwise	`null`	Denotes the pattern used to draw the base border when the group is in the OOS state.

How to customize the representation of rectangular groups

The following CSS extract shows how to customize a rectangular group so that its graphic representation is filled with a gradient.

```
object."ilog.tgo.model.IltRectGroup" {
  fillStyle: LINEAR_GRADIENT;
  background: yellow;
  foreground: blue;
}
```

Linear group properties

The following properties are used to draw the interior of a linear group:

CSS properties for the interior of linear groups

Property Name	Type	Set	Default Value	Description
`lineWidth`	`float`	No	`8`	Denotes the width of the linear base.
`foreground`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	28% grey	Denotes the foreground color of the base.
`background`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	`null`	Denotes the background color of the base.
`lineStyle`	`float[]`	Yes, if `baseStyleEnabled` is true No, otherwise	`null` (Solid)	Denotes the line style used to display a linear group.

The following properties are used to draw the border of a linear group.

CSS properties for the border of linear groups

Property Name	Type	Set	Default Value	Description
`reliefThickness`	`float`	No	`2`	Denotes the width of the relief around the linear group.
`borderColor`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	10% grey	Denotes the primary color of the base border.
`borderColor2`	`Color`	Yes, if `baseStyleEnabled` is true No, otherwise	60% grey	Denotes the secondary color of the base border.
`borderWidth`	`float`	Yes, if `baseStyleEnabled` is true No, otherwise	1	Denotes the width of the base border when `reliefBorders` is set to `false`.
`reliefBorders`	`boolean`	Yes, if `baseStyleEnabled` is true No, otherwise	true	Denotes whether the base border is drawn in relief or not.
`borderLineStyle`	`float[]`	Yes, if `baseStyleEnabled` is true No, otherwise	`null` (Solid)	Denotes the line style used to draw the base border.

How to customize the representation of linear groups

The following CSS extract shows how to customize a linear group so that its graphic representation shows a link with a width of 4 pixels in an alternate line style. The linear group is also configured so that it does not display a border around its base.

```
object."ilog.tgo.model.IltLinearGroup" {
  lineWidth: 4;
  lineStyle: "5,2";
  foreground: blue;
  background: yellow;
  reliefBorders: false;
  borderWidth: 0;
}
```

Customizing various aspects of groups

Customizing group names

You can customize the display of group names through the following CSS properties:

- `label`
- `labelVisible`
- `labelFont`
- `labelForeground`
- `labelBackground`
- `labelPosition`
- `labelAntialiasing`
- `labelSpacing`
- `labelWrappingMode`
- `labelWrappingWidth`
- `labelWrappingHeight`
- `labelMargin`
- `labelAlignment`
- `labelBorderColor`
- `labelDirection`
- `lineSpacing`
- `minLabelZoom`
- `maxLabelZoom`
- `labelScaleFactor`

Refer to *Customizing the label of a business object* for details on these properties and how to use them.

Customizing group icons

The following table lists the CSS properties that can be used to customize the group icon.

CSS properties for group icons

Property Name	Type	Default Value	Description
`icon`	`Image`	`null`	Defines the icon to be displayed in the group.
`iconVisible`	`boolean`	`true`	Defines whether the icon is visible or not in the group.

How to customize a group icon

The following example adds a test icon to all groups present in the graphic component.

```
object."ilog.tgo.model.IltGroup" {
  iconVisible: true;
  icon: '@|image("test.png")';
}
```

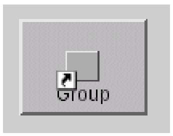
Customizing group shortcuts

The shortcut representation is displayed according to the business object CSS configuration. The following properties can be used to customize the shortcut representation.

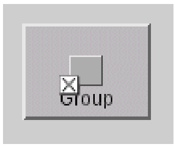
CSS properties for group shortcuts

Property Name	Type	Default Value	Description
`shortcutIcon`	`Image`	`ilt_shortcut.png` for the standard representation `ilt_dangling_shortcut.png` for the dangling representation	Defines the icon to be used in the shortcut representation.
`shortcutIconVisible`	`boolean`	`true`	Defines whether the shortcut icon is displayed or not.

The result is shown in the following images:



Standard shortcuts



Dangling shortcuts

Customizing group states and alarms

By default, groups do not graphically represent the primary state information. This behavior can be changed through the following CSS property:

CSS property for group primary states

Property Name	Type	Default Value	Description
`baseStyleEnabled`	`boolean`	`false`	Denotes whether the object base representation is affected by states and alarms present in the object.

You can also customize the visibility and characteristics of secondary states, alarm balloons and alarm counts, as in the other predefined business objects.

For more information, refer to *Customizing object and alarm states of predefined business objects*.

Customizing the group information cluster

All the structural and state information on a group—that is, its label, icon, plinth, secondary state modifiers, and alarm balloon—are gathered into what is called the *information cluster* of the group.

This cluster is positioned by placing a reference decoration at the center of the group. The reference decoration is listed below and is applied in the given order:

♦ The plinth, if there is one; otherwise the secondary state modifiers serve as this reference decoration, if there are any.

♦ If there are no secondary state modifiers, the reference decoration is the label, or the icon if there is no label.

The center of the group is the center of the rectangle for a rectangular group, the gravity center for polygonal groups, and the midpoint of the middle segment for linear groups.

You can modify the default positioning of the group information cluster by setting its relative position as an offset.

When a transformer is applied to the group, that is, when it is moved or scaled, the same transformer is applied to the offset of the group information cluster. In other words, the cluster follows the group. This is true regardless of whether the cluster position was specified with absolute or relative coordinates.

However, when the shape of the group is changed or the group is moved, the transformation of the group shape is not guaranteed to be affine. As a result, this change in the group shape is likely to change the default position of the group information cluster. You can specify whether such a shape change will affect the cluster position or whether the cluster will keep the same absolute coordinates. The property `pinDecorations` determines whether the position of the group information cluster is considered relative to the default position and will thus change if the default value changes.

You can define the above characteristics using CSS, with the following properties:

CSS properties for the group information cluster

Property Name	Type	Default Value	Description
`decorationsOffset`	`IlpPoint`	`null`	Denotes the offset used to attach the group information cluster to the group base.
`decorationsOffsetZoomable`	`boolean`	`false`	Denotes whether the offset used to attach the group information cluster to the base can be zoomed or not.
`pinDecorations`	`boolean`	`false`	Denotes whether the group information cluster remains unchanged even if the group has its shape changed.

The characteristics of the plinth decoration that is added to the group graphic representation can also be customized, using the following properties.

CSS properties for the plinth decoration

Property Name	Type	Set	Default Value	Description
`plinthColor`	`Color`	Yes	67% grey	Denotes the color of the plinth.
`plinthBrightColor`	`Color`	Yes	87% grey	Denotes the brighter color of the plinth.
`plinthDarkColor`	`Color`	Yes	33% grey	Denotes the darker color of the plinth.
`plinthVisible`	`boolean`	No	`true`	Denotes whether the plinth is visible or not.
`plinthVerticalMargin`	`int`	No	`1`	Defines the vertical margin between the plinth and its contents.
`plinthHorizontalMargin`	`int`	No	`4`	Defines the horizontal margin between the plinth and its contents.

Customizing group connection ports

The default behavior of links attached to groups is that they connect to the gravity decoration. If a group does not have a label, an icon, a secondary state or any other gravity decoration, the link connects to the base of the group. If the group has a label, the link connects to the label so that you can only edit the link ports around the label. The same is true for other decorations as well as for the plinth.

You can change this behavior through the property `linksConnectToBase` to make sure that the links always connect to the base:

CSS property for connecting links to the group base

Property Name	Type	Default Value	Description
`linksConnectToBase`	`boolean`	`false`	Denotes whether links always end at the group base. If this property is set to `true`, links connect to the bounding box of the base. Otherwise, they connect to the group information cluster, if one is present.

See also *Customizing node and link layouts*.

Customizing group tooltips

You can customize the display of group tooltips through the following CSS properties:

♦ `toolTipGraphic`

♦ `toolTipText`

For details on these properties and how to use them, refer to *Changing the font of all labels*.

Adding new decorations to groups

You can add new decorations to the predefined business objects representation.

For details, refer to *How to add new decorations to network and equipment nodes*.

Customizing subnetworks

Describes how subnetworks are represented and how to customize them.

In this section

Representing subnetworks
Describes how subnetworks are represented.

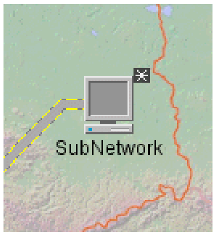
Customizing the representation of subnetworks
Describes how to customize the way subnetworks are represented.

Representing subnetworks

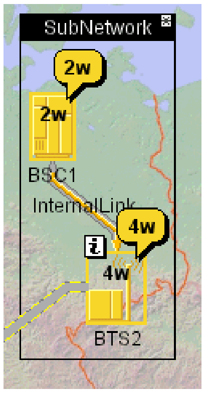
Subnetworks allow you to create applications that display a network inside another network. They are created automatically by the ILOG JViews TGO network component when you define a containment relationship between objects in the data source.

A subnetwork can be defined as any business object with child objects in the network component. You can display it either collapsed or expanded in the network component.

♦ In the *collapsed* state, the subnetwork is represented as a single object.



♦ In the *expanded* state, the subnetwork is displayed with all the objects contained in it.



Customizing the representation of subnetworks

How to customize the overview object of a subnetwork

In the collapsed representation, the subnetwork is displayed using an overview object, which is the graphic representation of the parent object as defined in the business model.

The following application is provided as part of the product_name demonstration software. It shows how to define a subnetwork with specific properties: ***<installdir>*** **/samples/network/basic**.

The properties are as follows:

- Business class: `IltNetworkElement`
- Identifier: `SubNetwork1`
- Name: `SubNetwork`
- Type: `IltNetworkElement.Type.NMW`

The graphical representation of the subnetwork in its collapsed state is provided by the same attributes and properties defined for network element objects. When customizing your subnetwork objects, you should use the CSS properties that apply to the business class used as the overview object.

```
#SubNetwork1 {
  functionVisible: false;
  detailLevel: MaximumDetails;
}
```

How to customize the expanded representation of a subnetwork

The following example illustrates the use of CSS to customize the graphical representation of an expanded subnetwork:

```
object."test.MyObject" {
  subnetworkTitle: @name;
  subnetworkTitleJustification: CENTER;
  subnetworkFrame: TITLEBAR_FRAME;
  subnetworkTitleColor: white;
  subnetworkRightMargin: 3;
  subnetworkLeftMargin: 3;
}
```

The following table lists all properties that can be used to customize subnetworks in their expanded representation.

CSS properties for expanded subnetworks

Property Name	Type	Default Value	Description
subnetworkBackground	Color	Color(128,128,128,128)	Denotes the ba of a subnetwork
subnetworkBottomMargin	float	5	Denotes the bo the frame of a s
subnetworkCollapseIconVisible	boolean	true	Denotes whethe icon is added to objects of a sub
subnetworkExpansionIconVisible	boolean	true	Denotes whethe expansion icon overview object subnetwork.
subnetworkForeground	Color	Color.black	Denotes the co around the subn
subnetworkFrame	IlpObjectFrameType	FILLED_RECTANGLE_FRAME	Denotes the typ by the subnetw possible values TITLEBAR_FRA FILLED_RECTA NO_FRAME
subnetworkLeftMargin	float	5	Denotes the lef frame of a subn
subnetworkOpaque	boolean	false	Defines whethe subnetwork is c The value of thi taken into acco property subne is set to TITLE subnetworkF NO_FRAME, subnetworkO automatically s subnetworkF FILLED_RECTA

Property Name	Type	Default Value	Description
			subnetworkO automatically s
subnetworkRightMargin	float	5	Denotes the rig frame of a subn
subnetworkShowingTitle	boolean	false	Defines whethe subnetwork sho not.
subnetworkTitle	String	Empty string	Defines the title subnetwork.
subnetworkTitleColor	Color	Color.white	Defines the col title of the subn
subnetworkTitleJustification	int	Left|Top	Defines the just title of the subn possiblle values Left|Top Right|Top Center|Top Left|Bottom Right|Botto Center|Bott Left|Top|Wr Right|Top|W Center|Top| Left|Bottom Right|Botto Center|Bott Examples of va Left|Top|Wr Right|Botto
subnetworkTopMargin	float	5	Denotes the top frame of a subn

How to use subnetwork properties in a user-defined class

```
object."mypackage.MyType" {
    subnetworkTitle: @name;
    subnetworkTitleJustification: Center;
    subnetworkFrame: TITLEBAR_FRAME;
    subnetworkTitleColor: blue;
    subnetworkRightMargin: 3;
    subnetworkLeftMargin: 3;
}
```

How to customize subnetwork interactors

You can use the `IlpGraphController` methods `setCollapsionBackgroundInteractor` and `setExpandInteractor` to customize subnetwork interactors to replace the default behavior of expand/collapse when double-clicking a subnetwork.

```
IlpDefaultObjectInteractor objInteractor = new IlpDefaultObjectInteractor();
Action myAction = new TestAction();
objInteractor.setGestureAction(IlpGesture.BUTTON1_DOUBLE_CLICKED, myAction);
objInteractor.setPopupMenuFactory(new TestPopupMenuFactory());
networkComponent.getController().
  setCollapsionBackgroundInteractor(objInteractor);
networkComponent.getController().setExpansionInteractor(objInteractor);
```

Customizing shelves and cards

Describes how shelves and cards are represented and how to customize the representations.

In this section

Representing physical telecommunication equipment
Describes the predefined business objects that represent items of telecommunications equipment.

Representing shelves
Describes how a shelf is represented graphically.

Customizing shelf representations
Provides details about the CSS properties that you can use to customize the representation of shelves.

Customizing various aspects of shelves
Describes how to customize shelf names, shelf states and alarms, shelf types, and shelf tiny types.

Representing and customizing card carriers
Describes how card carriers are represented and customized.

Customizing various aspects of card carriers
Describes how to customize card carrier names, card carrier states and alarms, card carrier types, and card carrier tiny types.

Representing and customizing cards
Describes how cards are represented and customized.

Customizing various aspects of card carriers
Describes how to customize card carrier names, card carrier states and alarms, card carrier types, and card carrier tiny types.

Representing and customizing ports
Describes how ports are represented and customized.

Customizing various aspects of ports
Describes how to customize port names, port states and alarms, port types, and port tiny types.

Representing and customizing LEDs
Describes how LEDs are represented and how to customize the representation.

Customizing various aspects of LEDs
Describes how to customize LED names, LED states and alarms, LED types, and LED tiny types.

Representing physical telecommunication equipment

ILOG JViews TGO provides a set of predefined business objects that are targeted to create physical views of telecommunication equipment.

The following business classes are available:

Shelves
A shelf is a rectangular frame made up of slots placed side by side. Each slot can hold one card object.

Card carriers
A card carrier represents a piece of telecommunication equipment placed in a slot. A card carrier differs from a basic card in the sense that it is a card container. A card carrier contains slots; all cards placed inside a card carrier have the same size, which is determined by the size of the card carrier. Shelf objects and card carrier objects can contain card carriers.

Cards
A card represents a piece of telecommunication equipment placed in a slot inside a shelf. Shelf objects and card carrier objects can contain cards.

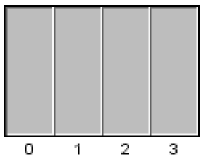
Ports
A port represents a physical interface to connect a card to other sets of equipment. Card objects can contain ports.

LEDs
A LED (Light Emitting Diode) is an object used to represent a state through a color. Most types of equipment use LEDs as an interface to provide information to the user about hardware and software conditions. Card objects can contain LEDs.

Representing shelves

The graphical representation of a shelf is based on the information that is available in the business model.

Although states and alarms may be associated with a shelf, you cannot display them in the shelf graphic representation because a shelf is only a container for cards.



A shelf representation

Customizing shelf representations

Some properties are mapped, which means that they are computed on the basis of the state and alarm information set in the object (column *Set*).

CSS properties for the representation of shelves, card carriers, cards and ports

Property Name	Type	Set	Default Value	Description
`foreground`	`Color`	Yes	28% gray in the `IltObject` class style	Denotes the foreground color of the object base.
`background`	`Color`	Yes	Transparent (`null`)	Denotes the background color of the object base.

`fillStyle`	`ilog.util.IlFillStyle`	Network node	`IlFillStyle.SOLID_COLOR` for user-defined business objects
			
			`IlFillStyle.PATTERN` for predefined business objects
				Denotes the style used to fill the base of an object.
				
				Possible values are:
				
				`IlFillStyle.NO_FILL`
				
				`IlFillStyle.SOLID_COLOR`
				
				`IlFillStyle.LINEAR_GRADIENT`
				
				`IlFillStyle.RADIAL_GRADIENT`
				
				`IlFillStyle.TEXTURE`
				
				`IlFillStyle.PATTERN`

`fillTexture`	`Image`	Network node	`null`	Denotes the texture used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.TEXTURE`.
`fillStart`	`Float`	Network node	`0f`	Returns the position where the gradient of an object starts, that is, where the color is the one defined by property `foreground`. This property is only used if `fillStyle` is set to `IlFillStyle.RADIA_GRADIENT` or `IlFillStyle.LINEAR_GRADIENT`.
`fillEnd`	`Float`	Network node	`1f`	Returns the position where the gradient of an object ends, that is, where the color is the one defined by property `background`. This property is only used if `fillStyle` is set to

Property Name	Type	Set	Default Value	Description
				`IlFillStyle.` `RADIA_GRADIENT` or `IlFillStyle.` `LINEAR_GRADIENT`
`fillAngle`	`Float`	Network node	`0`	Returns the angle (in degrees) of the gradient used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.` `RADIAL_GRADIENT` or `IlFillStyle.` `LINEAR_GRADIENT`
`fillPattern`	`Pattern`	Yes	`null` (Solid)	Denotes the pattern used to fill the base of an object. This property is only used if `fillStyle` is set to `IlFillStyle.PATTERN`.
`detailLevel`	`enum`	Yes	`MaximumDetails`	Defines the level of detail to be used to draw the base.
`borderColor`	`Color`	Yes	10% gray	Denotes the primary color of the base border.
`borderColor2`	`Color`	Yes	60% gray	Denotes the secondary color of the base border.
`borderWidth`	`float`	Yes	1 pixel	Denotes the width of the base border.
`borderLineStyle`	`float[]`	Yes	`null` (Solid)	Denotes the line style used to draw the base border.
`borderPattern`	`Pattern`	Yes	`null`	Denotes the pattern used to draw the base border.
`reliefBorders`	`boolean`	Yes	`true`	Denotes whether the base border is drawn in relief or not.

Note: The above table of properties also applies to card carriers, cards and ports.

A shelf also displays an outline around the slots and an alarm border in case of alarms. This representation can be customized through the following additional property.

CSS property for the shelf outline

Property Name	Type	Default Value	Description
`frameColor`	`Color`	black	Denotes the outline color of a shelf.

Customizing various aspects of shelves

Customizing shelf names

In the network and equipment components, shelves do not display their names. Instead of the shelf name, you see the slot labels, which are customized by using the property `XSlotLabels`, or by setting the attribute `xSlotIndex`. The properties listed in the following table can be used to customize slot labels.

CSS properties for slot labels

Property Name	Type	Default Value	Description
`XSlotNumbersOnTop`	`boolean`	`true`	Denotes whether the slot numbers along the x axis are displayed on top of the shelf or not.
`XSlotNumbersOffset`	int	1	Denotes the distance between the slot numbers and the shelf along the x axis.
`XSlotLabels`	String[]	`null`	Denotes the labels used on each slot along the x axis.
`labelVisible`	`boolean`	`true`	Controls whether the label is shown or not.
`labelAntialiasing`	`boolean`	`true`	Controls whether the label is drawn using antialiasing or not.
`labelFont`	`Font`	Helvetica 12, except: - in `IltShelf`: Helvetica 10 - in `IltShelfItem`: Helvetica 11 (Courier New 11 on Windows®)	Specifies the font to use to draw the label.
`labelForeground`	`Color`	black, except: - in `IltEmptySlot`: 50% gray	Gives the color of the label text.

How to customize shelf slot labels

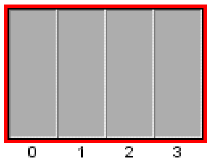
```
object."ilog.tgo.model.IltShelf" {
  labelVisible: true;
  labelForeground: blue;
  XSlotNumbersOffset: 3;
}
```

In the table and tree components, the shelf name displays with a tiny representation of the shelf. Instead of the slot labels, the shelf name is represented using the same CSS properties as for all the other predefined business objects. For a complete list of shelf name properties, refer to *Customizing the label of a business object*.

Customizing shelf states and alarms

Although states and alarms can be associated with a shelf, they cannot be displayed in the graphical representation of the shelf in the network and equipment components because a shelf is only a container for cards.

However, in the case of alarms, an alarm border is displayed around the shelf to indicate the presence of outstanding alarms (but no alarm balloon or alarm count), as illustrated below.



The following CSS properties are available to customize this representation.

CSS properties for the shelf alarm

Property Name	Type	Default Value	Description
`alarmBorderVisible`	`boolean`	`true`	Indicates whether the alarm border is visible or not around the object base.
`alarmBorderWidth`	`int`	`2` pixels	Defines the width of the alarm border
`alarmBorderColor`	`Color`	`null` (transparent)	Defines the color used to represent the alarm border around the base. Setting the value to `null` resets the alarm border color to its default value
`alarmColorVisible`	`boolean`	`false`	Determines whether the alarm color is visible or not in the object value.

How to customize shelf states and alarms

The following CSS extract shows how you can customize a shelf object to not display alarm information.

```
object."ilog.tgo.model.IltShelf" {
  alarmBorderVisible: false;
  alarmColorVisible: false;
}
```

The predefined shelf type has a label and a tooltip specified in the JViews TGO resource bundle. For details, see About internationalization.

The resources that apply to shelf types are identified as:

- `ilog.tgo.Shelf_Type_<TYPE NAME>`: shelf type labels
- `ilog.tgo.Shelf_Type_<TYPE NAME>_ToolTip`: shelf type tooltips

You can edit the values directly in the JViews TGO resource bundle file.

When you create new shelf types, the label and tooltip information will also be retrieved from this resource bundle to be displayed, for example, in a table cell. As you declare new shelf types, register the corresponding entries into the resource bundle file, as follows:

Considering that you have created the following new shelf type:

```
IltShelf.Type MyType = new IltShelf.Type("MyType");
```

You should declare the following properties in the `JTGOMessages.properties` file:

- `ilog.tgo.Shelf_Type_MyType=My Type`
- `ilog.tgo.Shelf_Type_MyType_ToolTip=My New Shelf Type`

Customizing shelf types

In JViews TGO, the shelf type defines how the object base will be represented. Each shelf type is associated with a specific base renderer that is in charge of drawing the object according to its type and state information.

In JViews TGO, you can customize the base representation of a shelf object by defining a new implementation of `IltShelfBaseRenderer`. The principle is the same as to create a new `IltNEBaseRenderer`. For details, refer to *Extending the class IltNEBaseRenderer*.

How to create and register a shelf type (using the API)

```
IltShelf.Type MyType = new IltShelf.Type("MyType");

IltSettings.SetValue("Shelf.Type.MyType.Renderer",
                     new IltBaseRendererFactory() {
                       public IltBaseRenderer createValue() {
                         return new MyTypeBaseRenderer();
                       }
                     });
```

How to create and register a shelf type (using CSS)

You can create new shelf types by using global CSS settings as shown in the following code sample.

```
setting."ilog.tgo.model.IltShelf"{
   types[0]: @+shelfType0;
}
Subobject#shelfType0 {
  class: 'ilog.tgo.model.IltShelf.Type';
  name: "MyType";
}
```

For more information, see *Using global settings*.

How to customize a shelf type (using CSS)

You can customize the shelf renderer using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right type of object in the system. The CSS property to customize is `renderer`.

```
setting."ilog.tgo.model.IltShelf.Type"[name="MyType"] {
   renderer: @+shelfRendererFactory;
}
Subobject#shelfRendererFactory {
   class: 'MyShelfRendererFactory';
}
```

In this code sample, the name of the renderer factory class that is included in the search path is `MyShelfRendererFactory`.

Customizing shelf tiny types

The shelf object can be represented as tiny objects in the tree and table components. Each shelf type can be associated with a tiny base renderer that is responsible for drawing the tiny graphic representation.

JViews TGO allows you to customize the tiny type representation by using one of the predefined base renderer factories, such as `IltTinyImageBaseRendererFactory` or `IltTinySVGBaseRendererFactory`, or by creating your own implementation of `IltTinyBaseRenderer`. The principle to create a new `IltTinyBaseRenderer` is the same as to create a new `IltNEBaseRenderer`.

For details, refer to *Extending the class IltNEBaseRenderer*.

How to modify a shelf tiny representation (using the API)

```
IltShelf.Type MyType = new IltShelf.Type("MyType");

IltSettings.SetValue("Shelf.TinyType.MyType.Renderer",
                     new IltTinyImageBaseRendererFactory(YOUR_IMAGE,
YOUR_IMAGE_PARAMETERS));
```

How to modify a shelf tiny representation (using CSS)

You can customize the renderer using global CSS settings. The CSS property to customize here is `tinyRenderer`.

In the following example, the name of the renderer factory class that is included in the search path is `MyShelfTinyRendererFactory`.

```
setting."ilog.tgo.model.IltShelf.Type"[name="MyType"] {
   tinyRenderer: @+shelfTinyRendererFactory0;
}
Subobject#shelfTinyRendererFactory0 {
   class: 'MyShelfTinyRendererFactory';
}
```

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

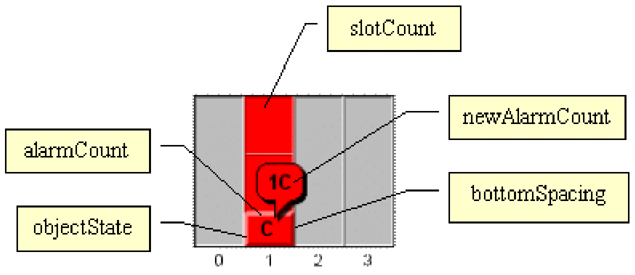
Representing and customizing card carriers

The graphic representation of a card carrier is based on the information that is available in the business model. Each decoration that is created depends on an attribute and on properties that can be customized through CSS.

Card carrier representation

The following figure shows a card carrier with the following attribute set:

- Slot count: 2
- Object State: OSI with new critical alarms
- Name: CC0
- Bottom spacing: 20



A card carrier representation

Customizing card carrier representations

You can customize the graphic representation of card carriers using the CSS properties detailed in *CSS properties for the representation of shelves, card carriers, cards and ports* .

Customizing various aspects of card carriers

Customizing card carrier names

By default, card carriers do not display their names when represented in the network and equipment components.

In the table and tree components, the card carrier label is displayed with a tiny representation of the card carrier. In this case, the card carrier name itself is represented using the same CSS properties as for all the other predefined business objects. For a complete list of name-related properties, refer to *Customizing the label of a business object*.

Customizing card carrier states and alarms

Card carriers display state and alarm information in the same way as other predefined business objects. For a complete list of the relevant properties, refer to *Customizing object and alarm states of predefined business objects*.

Customizing card carrier types

In JViews TGO, the card carrier type attribute defines how the object base is represented. Each card carrier type is associated with a specific base renderer that is in charge of drawing the object according to its type and state information.

In JViews TGO, you can customize the base representation of card carrier objects by defining a new implementation of `IltCardCarrierBaseRenderer`. The principle is the same as to create a new `IltNEBaseRenderer`. For details, refer to *Extending the class IltNEBaseRenderer*.

How to create and register a card carrier type (using the API)

```
IltCardCarrier.Type MyType = new IltCardCarrier.Type("CCType");
IltSettings.SetValue("CardCarrier.Type.CCType.Renderer",
                     new IltBaseRendererFactory() {
                       public IltBaseRenderer createValue() {
                         return new MyCCTypeBaseRenderer();
                       }
                     });
```

The predefined card carrier type has a label and a tooltip specified in the JViews TGO resource bundle, see About internationalization.

The resources that apply to card carrier types are identified as:

♦ `ilog.tgo.CardCarrier_Type_<TYPE NAME>`: card carrier type labels

♦ `ilog.tgo.CardCarrier_Type_<TYPE NAME>_ToolTip`: card carrier type tooltips

You can edit the values directly in the JViews TGO resource bundle files.

When you create new card carrier types, the label and tooltip information will also be retrieved from this resource bundle to be displayed, for example, in a table cell. As you declare new card carrier types, register the corresponding entries into the resource bundle file, as follows:

Considering that you have created the following new card carrier type:

```
IltCardCarrier.Type MyType = new IltCardCarrier.Type("CCType");
```

You should declare the following properties in the `JTGOMessages.properties` file:

- `ilog.tgo.CardCarrier_Type_CCType=CC Type`
- `ilog.tgo.CardCarrier_Type_CCType_ToolTip=My New Card Carrier Type`

How to create and register a card carrier type (using CSS)

You can create new card carrier types by using global CSS settings as shown in the following code sample.

```
setting."ilog.tgo.model.IltCardCarrier"{
   types[0]: @+cardCarrierType0;
}
Subobject#cardCarrierType0 {
  class: 'ilog.tgo.model.IltCardCarrier.Type';
  name: "CCType";
}
```

For more information, see *Using global settings* .

You can also customize the renderer using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right type of object in the system. The CSS property to customize is `renderer`.

```
setting."ilog.tgo.model.IltCardCarrier.Type"[name="CCType"] {
   renderer: @+cardCarrierRendererFactory;
}
Subobject#cardCarrierRendererFactory {
   class: 'MyCardCarrierRendererFactory';
}
```

In this code sample, the name of the renderer factory class that is included in the search path is `MyCardCarrierRendererFactory`

Customizing card carrier tiny types

Besides the graphic representation in the network and equipment components, the card carrier objects can be represented as tiny objects in the tree and table components.

Each card carrier type can be associated with a tiny base renderer that is responsible for drawing the tiny graphic representation. JViews TGO allows you to customize the tiny type representation by using one of the predefined base renderer factories, such as `IltTinyImageBaseRendererFactory` or `IltTinySVGBaseRendererFactory`, or by creating your own implementation of `IltTinyBaseRenderer`. The principle to create a new

IltTinyBaseRenderer is the same as to create a new IltNEBaseRenderer. For details, refer to *Extending the class IltNEBaseRenderer*.

How to modify a card carrier tiny representation (using the API)

```
IltCardCarrier.Type MyType = new IltCardCarrier.Type("CCType");
IltSettings.SetValue("CardCarrier.TinyType.MyType.Renderer",
                     new IltTinyImageBaseRendererFactory(YOUR_IMAGE,
YOUR_IMAGE_PARAMETERS));
```

How to modify a card carrier tiny representation (using CSS)

You can customize the renderer using global CSS settings. The CSS property to customize here is `tinyRenderer`. In the example below, the name of the renderer factory class that is included in the search path is `MyCardCarrierTinyRendererFactory`.

```
setting."ilog.tgo.model.IltCardCarrier.Type"[name="CCType"] {
   tinyRenderer: @+cardCarrierTinyRendererFactory0;
}
Subobject#CardCarrierTinyRendererFactory0 {
   class: 'MyCardCarrierTinyRendererFactory';
}
```

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

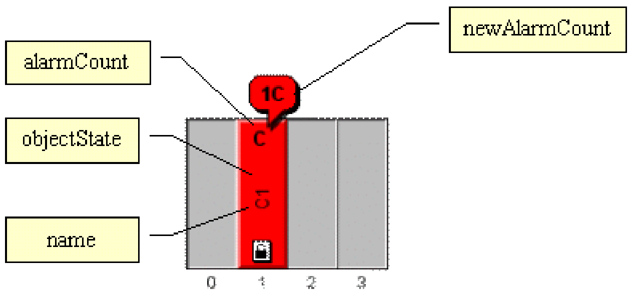
Representing and customizing cards

Representing cards

The graphical representation of a card is based on the information that is available in the business model. Each decoration that is created depends on an attribute and on properties that can be customized through CSS.

The following figure shows a shelf with a card that has the following attributes set:

- ♦ Type: Standard
- ♦ Name: C1
- ♦ Object State: OSI Object State with Alarms



A card representation

Customizing card representations

You can customize the graphic representation of cards using the CSS properties detailed in *CSS properties for the representation of shelves, card carriers, cards and ports* .

Customizing various aspects of card carriers

Customizing card types

In JViews TGO, the card type attribute defines how the object base is represented. Each card type is associated with a specific base renderer that is in charge of drawing the object according to its type and state information.

In JViews TGO, you can customize the base representation of card objects either by using an implementation of a predefined base renderer, such as `IltCardImageBaseRendererFactory` or `IltCardSVGBaseRendererFactory`, or by defining a new implementation of `IltCardBaseRenderer`. The principle to create a new `IltCardBaseRenderer` is the same as to create a new `IltNEBaseRenderer`. For details, refer to *Extending the class IltNEBaseRenderer*.

How to create and register a card type (using the API)

```
IltCard.Type MyType = new IltCard.Type("CType");
IltSettings.SetValue("Card.Type.CType.Renderer",
                     new IltBaseRendererFactory() {
                       public IltBaseRenderer createValue() {
                         return new MyCardTypeBaseRenderer();
                       }
                     });
```

How to create and register a card type (using CSS)

You can create new card types by using global CSS settings as shown in the following code sample.

```
setting."ilog.tgo.model.IltCard"{
   types[0]: @+cardType0;
}
Subobject#cardType0 {
  class: 'ilog.tgo.model.IltCard.Type';
  name: "CType";
}
```

For more information, see *Using global settings*.

How to customize a card type (using CSS)

You can also customize the renderer using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right type of object in the system. The CSS property to customize is `renderer`.

```
setting."ilog.tgo.model.IltCard.Type"[name="CType"] {
   renderer: @+cardRendererFactory;
}
Subobject#cardRendererFactory {
   class: 'MyCardRendererFactory';
}
```

In this code sample, the name of the renderer factory class that is included in the search path is `MyCardRendererFactory`.

How to create a new card type using an image

The following code sample shows how to create a new card type and associate it with an image.

The full application is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/equipment/imageRenderer**.

```
       String fileName = "rj45.png";
    // create the new type that will be associated with the GIF image
    IltCard.Type cardType = new IltCard.Type("RJ45");
    try {
      // Retrieve the image using the Image Repository
      Image img = 
IltSystem.GetDefaultContext().getImageRepository().getImage(fileName);
      // then map the drawer factory created using the GIF with the new type
      IltCardImageBaseRendererFactory factory =
        new IltCardImageBaseRendererFactory(img, 255, 1, 255);
      IltSettings.SetValue("Card.Type.RJ45.Label", "RJ45");
      IltSettings.SetValue("Card.Type.RJ45.Renderer", factory);
      // Note: the numerical values above have been adjusted using the
      // Image Color Tuner application provided with JTGO.
    } catch (Exception e) {
      e.printStackTrace();
    }
```

How to create a new card type using one image per base style

Another way to represent a type with an image is to specify a source image and an alarm color level parameter for every required base style, directly in CSS. No other base style property or renderer parameter is needed, as a complete image is provided for every needed base style.

To create a new type of card using one image per base style:

♦ Create a new type of card

  Using the API:

  ```
  IltCard.Type cardType = new IltCard.Type("RJ45");
  ```

  or, using global CSS settings:

```
setting."ilog.tgo.model.IltCard"{
   types[0]: @+cardType0;
}

Subobject#cardType0 {
   class: 'ilog.tgo.model.IltCard.Type';
   name: "RJ45";
}
```

- Map an `IltCardDirectImageBaseRendererFactory` to the new type

  Using the API `SetValue(java.lang.Object, java.lang.Object):`

```
IltSettings.SetValue("Card.Type.RJ45.Label", "RJ45");
IltSettings.SetValue("Card.Type.RJ45.Renderer",
    new IltCardDirectImageBaseRendererFactory());
```

  or, using global CSS settings:

```
setting."ilog.tgo.model.IltCard.Type"[name=RJ45] {
   renderer: @+cardRendererFactory;
}

Subobject#cardRendererFactory {
   class: 'ilog.tgo.graphic.renderer.IltCardDirectImageBaseRendererFactory';
}
```

- Define an image and an alarm color or gray-level parameter in CSS for each required base style

```
object."ilog.tgo.model.IltCard"["type"=RJ45]["objectState.Bellcore.State"=En
abledIdle] {
  sourceImage: '@|image("CardRJ45_EnabledIdle.png")';
  alarmColorLevel: 128;
}
object."ilog.tgo.model.IltCard"["type"=RJ45]["objectState.Bellcore.State"=Di
sabledIdle] {
  sourceImage: '@|image("CardRJ45_DisabledIdle.png")';
  alarmColorLevel: 140;
}
```

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

The predefined card types have a label and a tooltip specified in the JViews TGO resource bundle. For details, see About internationalization.

The resources that apply to card types are identified as:

- `ilog.tgo.Card_Type_<TYPE NAME>`: card type labels
- `ilog.tgo.Card_Type_<TYPE NAME>_ToolTip`: card type tooltips

You can edit the values directly in the JViews TGO resource bundle files.

When you create new card types, the label and tooltip information will also be retrieved from this resource bundle to be displayed, for example, in a table cell. As you declare new card types, register the corresponding entries into the resource bundle file, as follows:

Considering that you have created the following new card type:

```
IltCard.Type MyType = new IltCard.Type("CType");
```

You should declare the following properties in the `JTGOMessages.properties` file:

- `ilog.tgo.Card_Type_CType=C Type`
- `ilog.tgo.Card_Type_CType_ToolTip=My New Card Type`

Customizing card tiny types

In the tree and table components, you can represent card objects as tiny objects. Each card type can be associated with a tiny base renderer that is responsible for drawing the tiny graphic representation.

JViews TGO allows you to customize the tiny type representation by using one of the predefined base renderer factories, such as `IltTinyImageBaseRendererFactory` or `IltTinySVGBaseRendererFactory`, or by creating your own implementation of `IltTinyBaseRenderer`. The principle to create a new `IltTinyBaseRenderer` is the same as to create a new `IltNEBaseRenderer`. For details, refer to *Extending the class IltNEBaseRenderer*.

How to modify a card tiny representation (using the API)

```
IltCard.Type MyType = new IltCard.Type("CType");
IltSettings.SetValue("Card.TinyType.CType.Renderer",
                     new IltTinyImageBaseRendererFactory(YOUR_IMAGE,
YOUR_IMAGE_PARAMETERS));
```

How to modify a card tiny representation (using CSS)

You can customize the renderer using global CSS settings. The CSS property to customize here is `tinyRenderer`. In the example below, the name of the renderer factory class that is included in the search path is `MyCardTinyRendererFactory`.

```
setting."ilog.tgo.model.IltCard.Type"[name="CType"] {
   tinyRenderer: @+cardTinyRendererFactory0;
}
Subobject#CardTinyRendererFactory0 {
   class: 'MyCardTinyRendererFactory';
}
```

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

Customizing card names

Card names can be customized using the same properties as for the other predefined business objects. For a complete list of properties that apply to the card name representation, refer to *Customizing the label of a business object*.

In addition to these properties, card objects have the following specific property.

CSS property for card names

Property Name	Type	Default Value	Description
`verticalLabelStacksGlyphs`	`boolean`	`false`	Denotes the way a vertical label is built. If this property is set to `true`, the vertical label is build by stacking the characters. In this case, the properties `labelWrappingMode`, `labelWrappingWidth`, and `labelWrappingHeight` are ignored.

How to customize card names

While most of the predefined business objects display their label at the bottom of the base, cards display their label in its center. The following example shows how you can adjust the position of the label:

```
object."ilog.tgo.model.IltCard" {
   labelPosition: Bottom;
   labelSpacing: 25;
}
```

Customizing card states and alarms

Cards display state and alarm information in the same way as other predefined business objects. For a complete list of the relevant properties, refer to *Customizing object and alarm states of predefined business objects*.

Representing and customizing ports

Representing ports

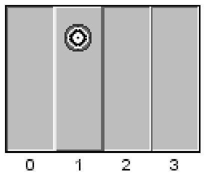
The graphic representation of a port is based on the information that is available in the business model. Each decoration that is created depends on an attribute and on properties that can be customized through CSS.

The following figure shows a shelf with a card and a port that has the following attributes set:

♦ Type: BNC_m

♦ Name: P0

♦ Object State: OSI Object State



A port representation

Customizing port representations

You can customize the graphic representation of ports using the CSS properties detailed in *CSS properties for the representation of shelves, card carriers, cards and ports* .

Customizing various aspects of ports

Customizing port types

In JViews TGO, the port type attribute determines how the object base is represented. Each port type is associated with a specific base renderer that is in charge of drawing the object according to its type and state information.

In JViews TGO, you can customize the base representation of port objects either by using an implementation of a predefined base renderer, such as `IltPortImageBaseRendererFactory` or `IltPortSVGBaseRendererFactory`, or by defining a new implementation of `IltPortBaseRenderer`. The principle to create a new `IltPortBaseRenderer` is the same as to create a new `IltNEBaseRenderer`..

For details, refer to *Extending the class IltNEBaseRenderer*.

How to create and register a port type (using the API)

```
IltPort.Type MyType = new IltPort.Type("MyType");
IltSettings.SetValue("Port.Type.MyType.Renderer",
                     new IltBaseRendererFactory() {
                       public IltBaseRenderer createValue() {
                         return new MyPortTypeBaseRenderer();
                       }
                     });
```

How to create and register a port type (using CSS)

You can also create new port types by using global CSS settings as shown in the following example.

```
setting."ilog.tgo.model.IltPort"{
   types[0]: @+portType0;
}
Subobject#portType0 {
  class: 'ilog.tgo.model.IltPort.Type';
  name: "MyType";
}
```

For more information, see *Using global settings*.

How to customize and register a port type (using CSS)

You can customize the renderer using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right type of object in the system. The CSS property to customize here is `renderer`. In the example below, the name of the renderer factory class that is included in the search path is `MyPortRendererFactory`.

```
setting."ilog.tgo.model.IltPort.Type"[name="MyType"] {
   renderer: @+portRendererFactory;
}
Subobject#portRendererFactory {
   class: 'MyPortRendererFactory';
}
```

How to create a new port type with an image (using the API)

The following code sample shows how to create a new port type and associate it with an image.

The full application is provided as part of the JViews TGO demonstration software at ***<installdir>* /samples/equipment/imageRenderer**.

```
    String fileName = "plug.png";
    // create the new type that will be associated with an image
    IltPort.Type portType = new IltPort.Type("Plug");

    try {
      // Retrieve the image using the Image Repository
      Image img =
IltSystem.GetDefaultContext().getImageRepository().getImage(fileName);
      // then map the renderer factory created using the image with the new
type
      IltPortImageBaseRendererFactory factory =
        new IltPortImageBaseRendererFactory(img, 255, 32, 165);
      IltSettings.SetValue("Port.Type.Plug.Label", "Plug");
      IltSettings.SetValue("Port.Type.Plug.Renderer", factory);
      // Note: the numerical values above have been adjusted using the
      // Image Color Tuner application provided with JTGO.
    } catch (Exception e) {
      e.printStackTrace();
    }
```

How to create a new port type with one image per base style

You can represent a type with images by specifying a source image and an alarm color level parameter for every required base style, directly in CSS. No other base style property or renderer parameter is needed, as a complete image is provided for every needed base style.

To create a new type of port using one image per base style:

♦ Create a new type of port

  Using the API:

  ```
  IltPort.Type portType = new IltPort.Type("Plug");
  ```

  or, using global CSS settings:

  ```
  setting."ilog.tgo.model.IltPort"{
  ```

```
   types[0]: @+portType0;
}

Subobject#portType0 {
   class: 'ilog.tgo.model.IltPort.Type';
   name: "Plug";
}
```

♦ Map an `IltPortDirectImageBaseRendererFactory` to the new type

  Using the API `SetValue(java.lang.Object, java.lang.Object):`

```
IltSettings.SetValue("Port.Type.Plug.Label", "Plug");
IltSettings.SetValue("Port.Type.Plug.Renderer",
    new IltPortDirectImageBaseRendererFactory());
```

  or, using global CSS settings:

```
setting."ilog.tgo.model.IltPort.Type"[name="Plug"] {
   renderer: @+portRendererFactory;
   label: Plug;
}

Subobject#portRendererFactory {
   class: 'ilog.tgo.graphic.renderer.IltPortDirectImageRendererFactory';
}
```

♦ Define an image and an alarm color or gray-level parameter in CSS for each required base style

```
object."ilog.tgo.model.IltPort"["type"=Plug]["objectState.Bellcore.State"=En
abledIdle] {
  sourceImage: '@|image("PortPlug_EnabledIdle.png")';
  alarmColorLevel: 128;
}
object."ilog.tgo.model.IltPort"["type"=Plug]["objectState.Bellcore.State"=Di
sabledIdle] {
  sourceImage: '@|image("PortPlug_DisabledIdle.png")';
  alarmColorLevel: 140;
}
```

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

Customizing port names

By default, port names are not visible in the network and equipment components. You can modify this behavior by setting the property `labelVisible`. All other label properties are also applicable to `IltPort` instances.

For a complete list of properties that apply to the port name representation, refer to *Customizing the label of a business object* .

Label and tooltip for port types

The predefined port types have a label and a tooltip specified in the JViews TGO resource bundle. For details, see About internationalization.

The resources that apply to port types are identified as:

♦ `ilog.tgo.Port_Type_<TYPE NAME>`: port type labels

♦ `ilog.tgo.Port_Type_<TYPE NAME>_ToolTip`: port type tooltips

You can edit the values directly in the JViews TGO resource bundle files.

When you create new port types, the label and tooltip information will also be retrieved from this resource bundle to be displayed, for example, in a table cell. As you declare new port types, register the corresponding entries in the resource bundle file, as follows.

Considering that you have created the following new port type:

```
IltPort.Type portType = new IltPort.Type("Plug");
```

You should declare the following properties in the `JTGOMessages.properties` file:

♦ `ilog.tgo.Port_Type_Plug=Plug`

♦ `ilog.tgo.Port_Type_Plug_ToolTip=Plug Port`

How to make a port label visible

```
object."ilog.tgo.model.IltPort" {
  labelVisible: true;
}
```

Customizing port tiny types

As with network elements, the tiny representation of ports is a reduced form of the symbolic representation. Therefore, it is not possible to create a tiny representation that is different from the symbolic representation of the object.

Customizing port states and alarms

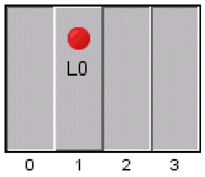
Ports display state and alarm information in the same way as other predefined business objects. For a complete list of the relevant properties, refer to *Customizing object and alarm states of predefined business objects*.

Representing and customizing LEDs

Representing LEDs

The graphic representation of LEDs is intended to be simple and reduced. Therefore it is only based on the attributes `type` and `name`, and on the alarms set in the object state, but not on any other state information.

- Type: Circular
- Name: L0
- Object State: OSI Object State with Alarms



A LED representation

Customizing LED representations

LEDs are meant to be simple objects with a reduced graphical representation. In this representation, the only property used to customize the LED representation is the property `foreground`, which defines the color of the graphic object.

By default, this color is mapped from the alarm information set in the object.

CSS property for LED representations

Property Name	Type	Set	Default Value	Description
`foreground`	`Color`	Yes	`lightGray`	Denotes the color of the LED.

Label and tooltip for port types

The predefined LED types have a label and a tooltip specified in the JViews TGO resource bundle.

The resources that apply to LED types are identified as:

- `ilog.tgo.LED_Type_<TYPE NAME>`: LED type labels
- `ilog.tgo.LED_Type_<TYPE NAME>_ToolTip`: LED type tooltips

You can edit the values directly in the JViews TGO resource bundle files.

When you create new LED types, the label and tooltip information will also be retrieved from this resource bundle to be displayed, for example, in a table cell. As you declare new LED types, register the corresponding entries in the resource bundle file.

Suppose that you have created the following new LED type:

```
IltLed.Type batteryType = new IltLed.Type("Battery");
```

You should declare the following properties in the `JTGOMessages.properties` file:

♦ `ilog.tgo.LED_Type_Battery=Battery`

♦ `ilog.tgo.LED_Type_Battery_ToolTip=Battery Indicator`

Customizing various aspects of LEDs

Customizing LED types

In JViews TGO, the LED `type` attribute determines how the object base is represented. Each LED type is associated with a specific base renderer that is in charge of drawing the object according to its type and state information.

In JViews TGO, you can customize the base representation of LED objects either by using an implementation of a predefined base renderer, such as `IltLedImageBaseRendererFactory` or `IltLedSVGBaseRendererFactory`, or by defining a new implementation of `IltLedBaseRenderer`. The principle to create a new `IltLedBaseRenderer` is the same as to create a new `IltNEBaseRenderer`.

For details, refer to *Extending the class IltNEBaseRenderer*.

How to create and register an LED type (using the API)

```
IltLed.Type MyType = new IltLed.Type("MyType");
IltSettings.SetValue("Led.Type.MyType.Renderer",
                     new IltBaseRendererFactory() {
                       public IltBaseRenderer createValue() {
                         return new MyLedTypeBaseRenderer();
                       }
                     });
```

How to create and register an LED type (using CSS)

You can create new LED types by using global CSS settings as shown in the following code sample.

```
setting."ilog.tgo.model.IltLed"{
   types[0]: @+ledType0;
}
Subobject#ledType0 {
  class: 'ilog.tgo.model.IltLed.Type';
  name: "MyType";
}
```

For more information, see *Using global settings*.

How to create a new LED type using only one image

To create a new LED type using a single image:

1. Create a new type of LED with the following code:

```
IltLed.Type myNewType = new IltLed.Type("MyType");
```

2. Create the image base renderer factory corresponding to the new LED type.

```
IltLedImageBaseRendererFactory factory = new
  IltLedImageBaseRendererFactory(image);
```

3. You must indicate to JViews TGO that this factory should be used to draw this type of LED. This is done through the mapping method `SetValue(java.lang.Object, java.lang.Object)`, as follows:

```
IltSettings.SetValue("Led.Type.MyType.Renderer", factory);
```

To illustrate the creation of a new LED type from a single image, suppose you want to create an LED type based on the following image, which is a GIF image with transparency:

The following code illustrates a static method that creates a new LED type. It also shows the mapping between the type and the factory.

```
// Create the new LED type
IltLed.Type batteryType = new IltLed.Type("Battery");

// Retrieve the image and create the base renderer factory
Image img =
IltSystem.GetDefaultContext().getImageRepository().getImage("battery.png");
IltLedImageBaseRendererFactory factor = new
IltLedImageBaseRendererFactory(img);

// Associate the new LED type with the image base renderer factory
IltSettings.SetValue("Led.Type.Battery.Renderer", factory);
```

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

Once the LED type has been created, you can start to instantiate LED objects as follows:

```
IltLed led = new IltLed("new", batteryType);
```

How to create a new LED type using two images

JViews TGO offers a way of creating new LED types based on two images. This process, known as `TwoImagesBaseRenderer`, allows you to create very detailed LEDs, where a specific image area “glows”, while the other areas remain unchanged.

The image base renderer gets two images and compares them pixel by pixel, identifying the differences between them. The difference defines the “glowing” area of the LED, or the area where the color changes.

Note: To use the TwoImagesBaseRenderer process, the images must have the same size and they should differ by at least one pixel.

Although the processes of creating a new LED type based on one image or on two images are similar, the concepts behind the two processes are quite different. In the case of a one-image LED type, a color filter is applied to the whole image when the LED color is set; in the case of a two-image LED type, the filter is applied only to the region that differs when the two images are compared.

To create a new LED type using two images:

1. Create a new type of LED with the following code:

   ```
   IltLed.Type myNewType = new IltLed.Type("MyType");
   ```

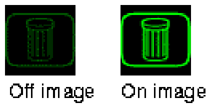
2. Create the image base renderer factory corresponding to the new LED type.

   ```
   IltLedImageBaseRendererFactory factory = new
     IltLedImageBaseRendererFactory(image_off, image_on);
   ```

3. You must indicate to JViews TGO that this factory should be used to draw this type of LED. This is done through the mapping method `SetValue(java.lang.Object, java.lang.Object)`, as follows:

   ```
   IltSettings.SetValue("Led.Type.MyType.Renderer", factory);
   ```

To illustrate the *TwoImagesBaseRenderer* process, imagine you want to create an LED representing a black area with a glowing trashcan. Only the trashcan and its frame are supposed to glow, the rest of the image must remain unchanged.



Two-image LED

The following code defines a static method to create the new LED type from two images.

```
/**
 * Creates the new led type using the
 * "trash_on.png" and "trash_off.png"
 * png images.
 */
String fileOn = "trash_on.png";
```

```
String fileOff = "trash_off.png";

// create the new type that will be associated with the
// given images
IltLed.Type theType = new IltLed.Type("TrashCan");

try {
  IlpImageRepository repository =
IltSystem.GetDefaultContext().getImageRepository();
  Image imgOn = repository.getImage(fileOn);
  Image imgOff = repository.getImage(fileOff);

  // then map the factory created using the images with the new type
  IltLedImageBaseRendererFactory factory =
    new IltLedImageBaseRendererFactory(imgOff, imgOn);

  IltSettings.SetValue("Led.Type.TrashCan.Renderer", factory);
  // Note: the numerical values above have been adjusted
  // using the imagecolortuner application provided with
  // JTGO
} catch (Exception ex) {
  logger.log("Error while creating TrashCan LED type.");
}
```

With the method defined above, you can instantiate an LED of the new type by coding:

```
IltLed newLedType = new IltLed("new", theType);
```

How to customize an LED type (using CSS)

You can customize the renderer using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right type of object in the system. The CSS property to customize here is `renderer`. In the following example, the name of the renderer factory class that is included in the search path is `MyLedRendererFactory`.

```
setting."ilog.tgo.model.IltLed.Type"[name="MyType"] {
   renderer: @+ledRendererFactory;
}
Subobject#ledRendererFactory {
   class: 'MyLedRendererFactory';
}
```

Customizing LED names

By default, LED names are not visible in the network and equipment components. You can modify this behavior by setting the property `labelVisible`. All other label properties are also applicable to `IltLED` instances.

For a complete list of properties that apply to the LED name representation, refer to *Customizing the label of a business object* for a complete list of properties that apply to the LED name representation.

How to make a LED label visible

```
object."ilog.tgo.model.IltLed" {
  labelVisible: true;
}
```

Customizing LED tiny types

As with network elements, the tiny representation of LEDs is a reduced form of the symbolic representation. Therefore, it is not possible to create a tiny representation that is different from the symbolic representation of the object.

Customizing LED states and alarms

LEDs do NOT display state information. The only pieces of information that can affect their appearance are alarms, which are by default mapped to the foreground color of the object base.

The properties listed in the following table are available to customize the representation of LEDs based on alarm information.

CSS properties for LED alarms

Property Name	Type	Default Value	Description
`alarmBorderVisible`	`boolean`	`false`	Denotes whether the alarm border is displayed or not around the object base.
`alarmColorVisible`	`boolean`	`true`	Denotes whether the alarm color is displayed or not in the object base. If this value is set to `false`, the foreground color of the object base does not change when new alarms are set in the object.

How to customize LED states and alarms

The following CSS code shows how you can change the predefined LED representation to hide the alarm border when alarms are displayed.

```
object."ilog.tgo.model.IltLed" {
   alarmBorderVisible: false;
}
```

You can customize the foreground color of the LED based on your own model attributes or using other state information present in the LED.

The following CSS code is part of an application is provided as part of the JViews TGO demonstration software at ***<installdir>*** **/samples/equipment/imageRenderer**.

In this code, the color of the LED is defined by the SNMP primary state.

```
object."ilog.tgo.model.IltLed" {
  foreground: lightGrey;
}
object."ilog.tgo.model.IltLed"["objectState.SNMP.State"=Up] {
  foreground: green;
}

object."ilog.tgo.model.IltLed"["objectState.SNMP.State"=Failed] {
  foreground: yellow;
}
```

Customizing BTS

Describes how to customize BTS objects and the BTS antennas they contain.

In this section

Representing and customizing BTS
Describes what BTS objects are and how they are represented.

Customizing BTS antennas
Lists the properties for customizing BTS antennas.

Customizing various aspects of BTS antennas
Describes how to customize BTS antenna names, BTS antenna states and alarms, BTS tiny types, and BTS antenna tiny types.

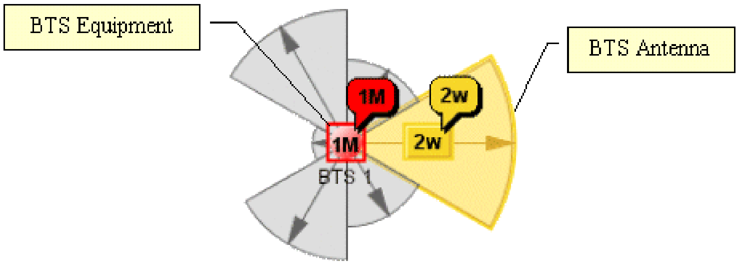
Representing and customizing BTS

Base Transceiver Stations (BTS) are base stations composed of antennas that relay (receive and transmit) radio messages within cells of a cellular phone system. Each antenna has an orientation and a beam width that are graphically represented. Each antenna can have its own state and graphical characteristics.

Representing BTS

A BTS object does not have a graphic representation in the network and equipment components. In these components, it works as a container that groups a BTS equipment and BTS antennas.

The following figure shows a BTS object composed of six BTS antennas.



A BTS representation

The graphic rendering of a BTS antenna object is optimized for performance using an offscreen buffered image technique that minimizes the complex shape computations by pre-rendering the graphic object in memory. The extra memory required is proportional to the size and number of visible objects in the network view. To disable the offscreen optimizations for the BTS antenna representation you should:

♦ Set the `ilog.tgo.bts.offscreenCache` system property to `false` at initialization time.

♦ Use the `IltSettings.SetValue("OffscreenCache.BTS", Boolean.FALSE);` API call.

For information on how to declare system properties for Java™ applications and applets, refer to your Java Runtime documentation.

Customizing the representation of a BTS equipment

A BTS equipment is a network element object. As such, it follows the same customization as network elements.

For a complete list of the properties used to customize this kind of network element, refer to *Customizing network element types*.

How to change the BTS equipment representation

```
object."ilog.tgo.model.IltNetworkElement"[type=BTSEquipment] {
  btsEquipmentRadius: 15;
}
```

Customizing BTS antennas

The properties listed in the following table allow you to customize the graphic representation of BTS antennas. They can be set for the BTS object directly if you want them to be applied to all antennas inside the BTS object. Or, if necessary, you can set properties for a specific antenna.

CSS properties for BTS antennas

Property Name	Type	Default Value	Description
`rotation`	`int`	`0`	Denotes the orientation of the BTS antenna, measured in degrees.
`antennaRadius`	`int`	`100`	Denotes the size of the antenna when the power value is highest.
`powerMaxValue`	`int`	`0`	Denotes the highest possible power value for antennas.
`antennaVisible`	`boolean`	`true`	Denotes whether the graphic representation of the antenna is visible or not.
`antialiasing`	`boolean`	`true`	Denotes whether antialiasing is used or not for the drawing ot the BTS antenna.
`alphaBeamWidth`	`float`	`1.0f`	Denotes the alpha value used to achieve transparency effects when drawing the antenna beam width.
`beamWidthRadius`	`int`	`100`	Denotes the beam width radius of the antennas when the power value is highest.
`beamWidthVisible`	`boolean`	`true`	Denotes whether the graphic representation of the antenna beam width is visible or not.
`beamWidthBorderVisible`	`boolean`	`true`	Denotes whether the beam width is drawn with a border or not.

How to customize the representation of any BTS antenna

```
object."ilog.tgo.model.IltBTSAntenna" {
  alphaBeamWidth: 0.5;
  beamWidthBorderVisible: true;
}
```

How to customize the representation of the antennas of a specific BTS object

The following CSS extract defines specific antenna properties which are set for a given BTS object. All the antennas that are part of this BTS object will be customized accordingly.

```
#bts1 {
  alphaBeamWidth: 0.5;
  beamWidthBorderVisible: true;
}
```

How to customize the representation of a single BTS antenna

The following CSS extract shows how to customize the representation of a single BTS antenna. In this case, the CSS selector used is based on the BTS antenna identifier.

```
#antenna1 {
  alphaBeamWidth: 0.8;
  beamWidthVisible: false;
}
```

Customizing various aspects of BTS antennas

Customizing BTS antenna names

For a list of properties that can be used to customize the display of BTS antenna names, refer to *Customizing the label of a business object*.

How to Make BTS Antenna Names Visible

By default, the names of BTS antennas are not displayed in the object graphic representation. The following CSS extract shows how you can make them visible:

```
object."ilog.tgo.model.IltBTSAntenna" {
  labelVisible: true;
}
```

Customizing BTS antenna states and alarms

The graphic representation of BTS antennas is affected by the presence of states and alarms.

For the complete list of properties that allow you to customize the representation of states and alarms in BTS Antennas, refer to *Customizing object and alarm states of predefined business objects*.

Customizing BTS tiny types and BTS antenna tiny types

BTS and BTS antenna tiny types are customized in a similar way to the link tiny type (see *Customizing link tiny types*).

How to Customize a BTS and a BTS Antenna Tiny Type (using the API)

```
IltSettings.SetValue("BTS.TinyType.Standard.Renderer",
  new IltTinyImageBaseRendererFactory(YOUR_IMAGE, YOUR_IMAGE_PARAMETERS));

IltSettings.SetValue("BTSAntenna.TinyType.Standard.Renderer",
  new IltTinyImageBaseRendererFactory(YOUR_IMAGE, YOUR_IMAGE_PARAMETERS));
```

How to Customize a BTS and a BTS Antenna Tiny Type (using CSS)

The CSS property to customize is `tinyRenderer`.

```
setting."ilog.tgo.model.IltBTS.TinyType"[name="Standard"] {
   tinyRenderer: @+btsTinyRendererFactory;
}
#btsTinyRendererFactory {
   class: 'MyBtsTinyRendererFactory';
}

setting."ilog.tgo.model.IltBTSAntenna.TinyType"[name="Standard"] {
   tinyRenderer: @+btsAntennaTinyRendererFactory;
}
Subobject#btsAntennaTinyRendererFactory {
   class: 'MyBtsAntennaTinyRendererFactory';
}
```

Customizing alarms

Describes how alarms are represented and how this representation is customized.

In this section

Representing alarms
Describes what alarms are and how alarms are represented.

Customizing various aspects of alarms
Describes how to customize severities, probable causes, alarm types, and trend indications.

Representing alarms

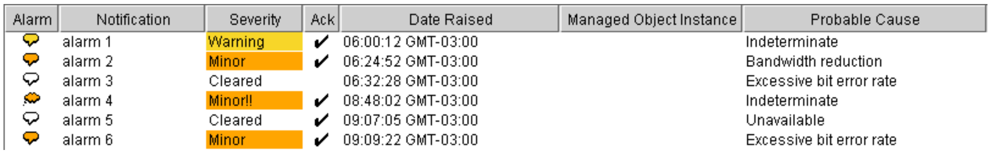
Alarms are predefined business objects used to represent alarm conditions that occur in managed objects.

By default, alarm business objects are not represented in the network and equipment components. Instead, the *alarm state* is represented. The alarm state provides an aggregated view of the alarms that affect a managed object. For details on how to customize the alarm state information, refer to *Customizing object states*.

Alarm business objects have a representation in the table component and in the tree component.

Representation of alarms in a table

Like any business object, an alarm is represented as a row in a table. Each column of the table corresponds to an attribute of the alarm.



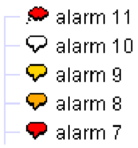
Alarm	Notification	Severity	Ack	Date Raised	Managed Object Instance	Probable Cause
🗨	alarm 1	Warning	✔	06:00:12 GMT-03:00		Indeterminate
🗨	alarm 2	Minor	✔	06:24:52 GMT-03:00		Bandwidth reduction
🗨	alarm 3	Cleared		06:32:28 GMT-03:00		Excessive bit error rate
🗨	alarm 4	Minor!!	✔	08:48:02 GMT-03:00		Indeterminate
🗨	alarm 5	Cleared	✔	09:07:05 GMT-03:00		Unavailable
🗨	alarm 6	Minor	✔	09:09:22 GMT-03:00		Excessive bit error rate

Representation of alarms in a table

For information on how to customize a table row, see *Customizing table cells*.

Representation of alarms in a tree

Like any business object, an alarm is represented as a tree node in a tree.



Representation of alarms in a tree

For information on how to customize a tree node, see *Customizing tree nodes*.

Customizing various aspects of alarms

Customizing severities

For information on how to customize the colors and labels associated with alarm severities, refer to *Customizing alarm severities* .

Customizing probable causes

The predefined probable causes provided in ILOG JViews TGO are those defined in the OSS/J Quality of Service APIs.

Probable causes may be created using the Java™ API, or dynamically while feeding an XML stream into a data source.

How to create a new probable cause using the Java API

In the following code, a new probable cause is created for the numeric value 600.

```
IltAlarm.ProbableCause probableCause = new IltAlarm.ProbableCause(600);
```

To see how to set a representation for the new probable cause, refer to *How to change the representation of a probable cause*.

How to create a new probable cause dynamically from an XML stream

The following XML stream extract sets the probable cause for an alarm to 600. If the probable cause did not exist previously in the data source, it is created dynamically.

```
<addObject id="alarm 1">
  <class>ilog.tgo.model.IltAlarm</class>
  <attribute name="probableCause">600</attribute>
  <!-- ... -->
</addObject>
```

To see how to set a representation for the new probable cause, refer to *How to change the representation of a probable cause*.

How to change the representation of a probable cause

In the following CSS code, the label for the probable cause with the numeric value 600 is set to “my new probable cause”.

```
object."ilog.tgo.model.IltAlarm/probableCause"[probableCause=600] {
  label: "my new probable cause";
}
```

Customizing alarm types

The predefined alarm types provided in JViews TGO are those defined in ITU-T X.721.

How to change the representation of an alarm type

In the following CSS extract, the label for the alarm type `IntegrityViolation` is set to “Violation of integrity”.

```
object."ilog.tgo.model.IltAlarm/alarmType"[alarmType=IntegrityViolation] {
  label: "Violation of integrity";
}
```

Customizing trend indications

The predefined trend indications provided in JViews TGO are those defined in ITU-T X.721.

How to change the representation of a trend indication

In the following CSS extract, the label for the trend indication `NoChange` is set to “Unchanged”.

```
object."ilog.tgo.model.IltAlarm/trendIndication"[trendIndication=NoChange] {
  label: "Unchanged";
}
```

Customizing off-page connectors

Describes how to customize off-page connectors.

In this section

Representing off-page connectors
Describes what an off-page connector is and how it is represented.

Customizing existing off-page connector types
Lists the predefined connector types and describes how to customize them.

Customizing new off-page connector types
Describes how to create and customize new connector types.

Customizing other aspects of off-page connectors
Describes how to customize connector names and connector states and alarms.

Representing off-page connectors

You can insert off-page connectors in a network to replace nodes. They indicate that a link continues in part of the network that is outside the current view.

An off-page connector can have associated information for:

♦ Displaying the corresponding view

♦ Indicating visually on which neighbor view the object represented by the off-page connector is located

The graphic representation of an off-page connector is based on the information that is available in the business model. Each decoration that is created depends on an attribute and on properties that can be customized through CSS.

Although an off-page connector can have states and alarms set like any other predefined business object, its graphic representation is intended to be simple and to highlight only the most important information about the object that it replaces.

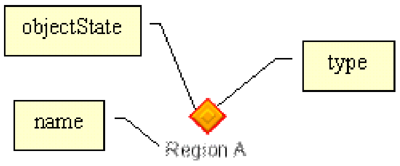
Off-page connectors do not represent states, alarm counts or alarm balloons. The alarm information is represented in the object base, as in the following illustration.

The following image shows an off-page connector with the following attribute set:

♦ Type: Standard

♦ Name: Region A

♦ Object state: OSI Object State with Alarms



An off-page connector representation

Customizing existing off-page connector types

Predefined connector types

JViews TGO provides `IltOffPageConnector` types that you can use directly in your applications, as listed in the following table.

Off_page connector types and their graphical representation

Type	Representation	Description
Standard		Standard off-page connector
Managed		The off-page connector represents a generic managed entity.
SingleManaged		The off-page connector represents a single entity currently managed by the system.
MultipleManaged		The off-page connector represents multiple entities currently managed by the system.
Unmanaged		The off-page connector represents a generic entity not managed by the system.
SingleUnmanaged		The off-page connector represents a single entity currently not managed by the system.
MultipleUnmanaged		The off-page connector represents multiple entities currently not managed by the system.

Properties of predefined connector types

The properties listed in the following table apply to all the predefined off-page connector types.

CSS properties for the predefined off-page connector types

Property Name	Type	Default Value	Description
`shapeWidth`	`float`	`19`	Denotes the width of an off-page connector.
`shapeHeight`	`float`	`19`	Denotes the height of an off-page connector.
`depressed`	`boolean`	`false`	Denotes whether the graphic representation of the off-page connector is displayed depressed or not.
`tiny`	`boolean`	`false` in the network and equipment components `true` in the table and tree components	Sets the off-page connector base to the tiny representation. The tiny representation is used mainly in the tree and table components.
`logical`	`boolean`	`false`	Sets the off-page connector base to the logical representation. In the logical representation, all off-page connector types have the same look, that is a rectangular shape.

Mapped properties of predefined connector types

The properties listed in the following table are mapped, that is, their value is computed automatically from the alarms set in the object (column *Set*). You can override the mapped values or customize their graphic representation even when the object does not carry states and alarms.

Mapped CSS properties for off-page connectors

Property Name	Type	Set	Default Value	Description
`foreground`	`Color`	Yes	28% gray in the `IltObject` class style	Denotes the foreground color of the object base.
`background`	`Color`	Yes	Transparent (`null`)	Denotes the background color of the object base.

`fillStyle`	`ilog.util.IlFillStyle`	Network node	`IlFillStyle.SOLID_COLOR` for user-defined business objects
			
			`IlFillStyle.PATTERN` for predefined business objects
				Denotes the style used to fill the base of an object.
				
				Possible values are:
				
				`IlFillStyle.NO_FILL`
				
				`IlFillStyle.SOLID_COLOR`
				
				`IlFillStyle.LINEAR_GRADIENT`
				
				`IlFillStyle.RADIAL_GRADIENT`
				
				`IlFillStyle.TEXTURE`

Property Name	Type	Set	Default Value	Description
				IlFillStyle.PATTERN
fillTexture	Image	Network node	null	Denotes the texture used to fill the base of an object. This property is only used if fillStyle is set to IlFillStyle.TEXTURE.
fillPattern	Pattern	Yes	null (Solid)	Denotes the pattern used to fill the base of an object. This property is only used if fillStyle is set to IlFillStyle.PATTERN.
fillStart	float	Network node	0f	Returns the position where the gradient of an object starts, that is, where the color is the one defined by property foreground. This property is only used if fillStyle is set to IlFillStyle. RADIA_GRADIENT or IlFillStyle.LINEAR_GRADIENT.
fillEnd	float	Network node	1f	Returns the position where the gradient of an object ends, that is, where the color is the one defined by property background. This property is only used if fillStyle is set to IlFillStyle. RADIA_GRADIENT or IlFillStyle.LINEAR_GRADIENT
fillAngle	float	Network node	0	Returns the angle (in degrees) of the gradient used to fill the base of an object. This property is only used if fillStyle is set to IlFillStyle. RADIAL_GRADIENT or IlFillStyle.LINEAR_GRADIENT
borderColor	Color	Yes	10% gray	Denotes the primary color of the base border.
borderColor2	Color	Yes	60% gray	Denotes the secondary color of the base border.
reliefThickness	float	No	1	Denotes the width of the relief on the base border.

Property for the standard predefined connector type

The property listed in the following table is specific to the type IltOffPageConnector. Standard. This type determines the graphic representation of an off-page connector in the form of two nested relief diamonds.

CSS property specific to the IltOffPageConnector.standard type

Property Name	Type	Default Value	Description
`reliefDistance`	`int`	`4`	Denotes the distance between two nested reliefs diamonds.

How to customize standard off-page connectors

The following CSS extract shows how you can customize the graphic representation of standard off-page connectors. In this example, the size of the off-page connector is set to 21x21:

```
object."ilog.tgo.model.IltOffPageConnector"[type=Standard] {
   shapeWidth: 21;
   shapeHeight: 21;
}
```

Customizing new off-page connector types

The principle to create new types of off-page connectors is the same as to create new types of network elements. For details, refer to *Customizing network element types*.

Customizing the base renderer of a connector type

In ILOG JViews TGO, the off-page connector type determines how the object base is represented. Each off-page connector type is associated with a specific base renderer that is in charge of drawing the object according to its type and alarm information.

In JViews TGO, you can extend the base representation of off-page connectors by using one of the predefined base renderer factory classes (`IltOPCImageBaseRendererFactory` or `IltOPCSVGBaseRendererFactory`), or by implementing your own subclass of `IltOPCBaseRenderer` for each new type of off-page connector that you want to create.

How to create a new off-page connector type from an image (using the API)

The following code extract shows how to create a new `IltOffPageConnector` type and associate it with an image so that it can be displayed in the JViews TGO graphic components.

```
// Create the new OPC type
IltOffPageConnector.Type opcType = new IltOffPageConnector.Type("MyType");

// Retrieve the image and create the base renderer factory
Image img =
IltSystem.GetDefaultContext().getImageRepository().getImage("type.png");
IltOPCImageBaseRendererFactory factory = new
IltOPCImageBaseRendererFactory(img);

// Associate the new OPC type withthe image base renderer factory
IltSettings.SetValue("OffPageConnector.Type.MyType.Renderer", factory);
```

How to create a new off-page connector type from an image (using CSS)

You can create new off-page connector types by using global CSS settings (for more information, see *Using global settings*).

```
setting."ilog.tgo.model.IltOffPageConnector"{
   types[0]: @+opcType0;
}
Subobject#opcType0 {
  class: 'ilog.tgo.model.IltOffPageConnector$Type';
  name: "MyType";
}
```

How to customize an off-page connector renderer (using CSS)

You can customize the renderer using global CSS settings. To do so, you need to specify the full path to the object to be customized, as well as the value of its name attribute in order to match the right type of object in the system. The CSS property to customize here is `renderer`. In the example below, the name of the renderer factory class that is included in the search path is `MyOPCRendererFactory`.

```
setting."ilog.tgo.model.IltOffPageConnector.Type"[name="MyType"] {
   renderer: @+opcRendererFactory1;
}
Subobject#opcRendererFactory1 {
   class: 'MyOPCRendererFactory';
}
```

How to create a new off-page connector type using one image per base style

Another way to represent a type with an image is to specify a source image and an alarm color level parameter for every required base style, directly in CSS. No other base style property or renderer parameter is needed, as a complete image is provided for every needed base style.

To create a new type of off-page connector using an image per base style:

1. Create a new type of off-page connector

   Using the API:

   ```
   IltOffPageConnector.Type opcType = new IltOffPageConnector.Type("MyType")
   ;
   ```

   or, using global CSS settings:

   ```
   setting."ilog.tgo.model.IltOffPageConnector"{
      types[0]: @+opcType0;
   }

   Subobject#opcType0 {
      class: 'ilog.tgo.model.IltOffPageConnector.Type';
      name: "MyType";
   }
   ```

2. Map an `IltOPCDirectImageBaseRendererFactory` to the new type

   Using the API `SetValue(java.lang.Object, java.lang.Object):`

   ```
   IltSettings.SetValue("OffPageConnector.Type.MyType.Renderer",
       new IltOPCDirectImageBaseRendererFactory());
   ```

or, using global CSS settings:

```
setting."ilog.tgo.model.IltOffPageConnector.Type"[name="MyType"] {
   renderer: @+opcRendererFactory1;
}

Subobject#opcRendererFactory1 {
   class: 'ilog.tgo.graphic.renderer.IltOPCDirectImageRendererFactory';
}
```

3. Define an image and an alarm color or gray-level parameter in CSS for each required base style

```
object."ilog.tgo.model.IltOffPageConnector"["type"=MyType]["objectState.
Bell
core.State"=EnabledIdle] {
  sourceImage: '@|image("OPCMyType_EnabledIdle.png")';
  alarmColorLevel: 128;
}
object."ilog.tgo.model.IltOffPageConnector"["type"=MyType]["objectState.
Bell
core.State"=DisabledIdle] {
  sourceImage: '@|image("OPCMyType_DisabledIdle.png")';
  alarmColorLevel: 140;
}
```

For details about how to create image base renderers, refer to *Creating network element types from images and customizing them*.

Labels and tooltips for connector types

The predefined off-page connector types have a label and a tooltip specified in the JViews TGO resource bundle.

The resources that apply to off-page connector types are identified as:

♦ `ilog.tgo.OPC_Type_<TYPE NAME>`: off-page connector type labels

♦ `ilog.tgo.OPC_Type_<TYPE NAME>_ToolTip`: off-page connector type tooltips

You can edit the values directly in the JViews TGO resource bundle files.

When you create new off-page connector types, the label and tooltip information will also be retrieved from the same resource bundle to be displayed, for example, in a table cell. As you declare new types, register the corresponding entries in the resource bundle file.

Suppose that you have created the following new off-page connector type:

```
IltOffPageConnector.Type opcType = new IltOffPageConnector.Type("MyType");
```

You should declare the following properties in the `JTGOMessages.properties` file:

♦ `ilog.tgo.OPC_Type_MyType=My Type`

- `ilog.tgo.OPC_Type_MyType_ToolTip=My Off-Page Connector Type`

Customizing other aspects of off-page connectors

Customizing off-page connector names

Off-page connector names are graphically represented like in the other predefined business objects. For a list of properties that apply to the off-page connector name representation, refer to *Customizing the label of a business object*.

Customizing off-page connector states and alarms

The graphical representation of off-page connectors does not take into account the states set in the business objects. It is only affected by the object type and the alarms, if any (but alarm counts and alarm balloons are not represented).

The properties listed in the following table allow you to customize the graphic representation of alarms in off-page connectors.

CSS properties for off-page connector alarm representatio

Property Name	Type	Default	Description
primaryAlarmState	IltAlarmStateEnum	IltAlarmStateEnum.Raw	Determines whether the raw alarms or the impact alarms are displayed as the primary alarm state. Possible values are: ltAlarmStateEnum.Raw or ltAlarmStateEnum.Impact
alarmBorderColor	Color	null (transparent)	Defines the color used to represent the alarm border around the base. Setting the value to `null` resets the alarm border color to its default value
alarmBorderWidth	int	2 pixels	Defines the width of the alarm border.
alarmBorderVisible	boolean	true	Indicates whether the alarm border is visible or not around the object base.
alarmColorVisible	boolean	true	Determines whether the alarm color is visible or not in the object value.

How to customize the alarm representation in off-page connectors

The following CSS extract shows how you can customize the graphic representation of off-page connectors so that they do not display alarm information.

```
object."ilog.tgo.model.IltOffPageConnector" {
  alarmColorVisible: false;
  alarmBorderVisible: false;
}
```

Customizing object states

Explains how object states are represented in the predefined business objects and how to customize the representations.

In this section

Customizing the object representation based on states
Describes how object states affect representation and gives more specific descriptions of object states from different telecommuncations standards.

Customizing passive devices
Describes what passive devices are and how to customize them.

Customizing the OSI state system
Describes the OSI states and how to customize them.

Customizing the Bellcore state system
Describes the Bellcore states and how to customize them.

Customizing the SNMP state system
Describes the SNMP states and how to customize them.

Customizing the SONET state system
Describes the SONET states and how to customize them.

Customizing the Miscellaneous state system
Describes the Miscellaneous secondary states and how to create and customize them.

Customizing the Performance State System
Describes the Performance secondary states and how to create and customize them.

Customizing the SAN state system
Describes the SAN secondary states and how to create and customize them.

Customizing alarm severities
Lists the properties of the primary and secondary alarm states, describes how to customize them to represent raw and impact alarms, and describes how to customize alarm severities.

Customizing alarm count attributes
Describes how alarm count attributes are represented in the table component and how to customize the representations.

Customizing trap types
Describes how to customize existing trap types and how to create and customize new trap types.

Customizing the secondary state icons
Describes how to change the positions where secondary state icons usually appear.

Customizing the object representation based on states

Describes how object states affect representation and gives more specific descriptions of object states from different telecommuncations standards.

In this section

Representing and customizing state information
Describes how primary and secondary states affect representation and discusses customization in general.

OSI states
Describes how to customize states based on the OSI state dictionary.

Bellcore states
Describes how to customize states based on the Bellcore state dictionary.

SNMP states
Describes how to customize states based on the SNMP state dictionary.

SONET states
Describes how to customize states based on the SONET state dictionary.

Miscellaneous states
Describes how to customize the predefined miscellaneous states.

Performance states
Describes how to customize the predefined performance states.

SAN states
Describes how to customize the predefined SAN states.

Alarms
Describes how to customize the predefined alarms.

Traps
Describes how to customize the predefined traps.

Representing and customizing state information

Representing primary and secondary states

ILOG JViews TGO predefined business objects are graphically represented with properties that change according to the states and alarms set on the objects.

In general, the primary state information is used to define the representation characteristics of the object base, while secondary states are added as new decorations. This predefined mapping can be modified through cascading style sheets (CSS). For information on how to create and use CSS files, see *Introducing cascading style sheets*.

JViews TGO provides a set of visual dictionaries that are used for displaying alarm and state changes in predefined telecom business objects. These dictionaries are based on the following worldwide telecommunication standards: OSI, Bellcore, SNMP, SONET, Performance, SAN, Alarm, Trap, and Miscellaneous state dictionaries. Each state dictionary provides visual representations of the states.

For information about each state dictionary and its graphical representation, see States.

Customizing states and alarms with CSS selectors

You can customize the object representation according to its states and alarms using CSS attribute selectors. An attribute selector is based on the `IltObject` attribute "`objectState`" (see `ilog.tgo.model.IltObject#ObjectStateAttribute`) and is composed of the attribute name and the state name information.

The selectors can be used to customize the object graphic representation according to the different state and alarm dictionaries.

It is recommended that, when you specify a graphic property in an attribute selector, you provide an object selector that sets a default value to the property. This value will be used when the attribute is not defined. In other words, you can customize your objects according to the attribute values, but keep in mind that a consistent graphic representation should also be available when the attribute is not set in a specific object. Observe this policy whenever you create CSS attribute selectors to customize the object graphic representation based on states and alarms.

The following example illustrates this policy in a scenario that creates a graphic representation based on the presence of the state `IltMisc.SecState.HighTemperatureWarning`:

```
object."ilog.tgo.model.IltObject"["objectState.Misc.SecState.HighTemperatureWar
ning"] {
  labelForeground: red;
}

object."ilog.tgo.model.IltObject" {
  labelForeground: '';
}
```

In this scenario, when the state `IltMisc.SecState.HighTemperatureWarning` is present in the object, the label of the object changes to red. When the state is no longer present, the label color is set to its default value.

OSI states

The OSI state dictionary is based on the OSI SMF 10164-2 standard defining the primary state of a telecom object as a combination of three states, as well as a number of secondary states. See The OSI state dictionary visuals for more information.

Using cascading style sheets, you can define specific selectors based on OSI primary and secondary states.

OSI primary states

The following selector matches all objects that have the OSI primary state `Administrative=Locked` defined.

```
object."ilog.tgo.model.IltObject"["objectState.OSI.State.Administrative"=Locked
] {
...
}
```

In CSS selectors, primary states are identified by the attribute name ("`objectState`") and the primary state information ("`OSI.State.Administrative`", "`OSI.State.Operational`", or "`OSI.State.Usage`").

You can also create selectors which are based on more than one state, as illustrated below:

```
object."ilog.tgo.model.IltObject"["objectState.OSI.State.Administrative"=Locked
]["objectState.OSI.State.Operational"=Enabled] {
...
}
```

OSI secondary states

You can create selectors based on secondary state information. Secondary states are identified by the attribute name ("`objectState`") and the state information.

In the OSI state dictionary, secondary states are identified by the group to which they belong (for example, Procedural, Availability, Control, Standby and Repair) and by their name.

The following selectors match all objects that contain the state `OSI.Procedural.Initializing` or `OSI.Availability.NotInstalled`.

```
object."ilog.tgo.model.IltObject"["objectState.OSI.Procedural.Initializing"]
{
...
}

object."ilog.tgo.model.IltObject"["objectState.OSI.Availability.NotInstalled"]

{
```

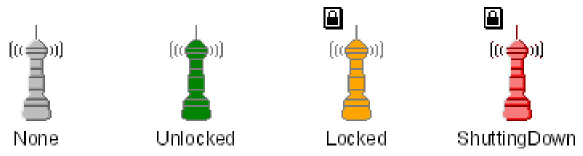
```
...
}
```

How to change the object representation based on OSI states

The following CSS extract customizes the graphic representation of all telecom business objects according to the value of the OSI Administrative State. This example does not take into account the possible presence of alarms in the objects.

```
object."ilog.tgo.model.IltObject" {
  foreground: '';
}
object."ilog.tgo.model.IltObject"["objectState.OSI.State.Administrative"=Locked
] {
  foreground: orange;
}
object."ilog.tgo.model.IltObject"["objectState.OSI.State.Administrative"=Unlock
ed] {
  foreground: green;
}
object."ilog.tgo.model.IltObject"["objectState.OSI.State.Administrative"=Shutti
ngDown] {
  foreground: red;
}
```

OSI state styling example illustrates this configuration on network elements:



OSI state styling example

Bellcore states

The Bellcore state dictionary defines a primary state and several secondary states, which are used to define the graphic representation of predefined business objects through cascading style sheets. See The Bellcore state dictionary visuals for more information.

Bellcore primary states

The following selector matches all objects that have the Bellcore primary state `EnabledActive` defined.

```
object."ilog.tgo.model.IltObject"["objectState.Bellcore.State"=EnabledActive]
 {
...
}
```

Bellcore secondary states

You can also use secondary states in your attribute selectors. In CSS selectors, secondary states are identified by the attribute name ("`objectState`") and the secondary state information (Blocked, Combined, and so on). See The Bellcore state dictionary visuals for a complete list of available secondary states.

The following selectors match all objects that contain the secondary state Blocked or Combined:

```
object."ilog.tgo.model.IltObject"["objectState.Bellcore.SecState.Blocked"] {
...
}

object."ilog.tgo.model.IltObject"["objectState.Bellcore.SecState.Combined"] {
...
}
```

How to change the object representation based on Bellcore states

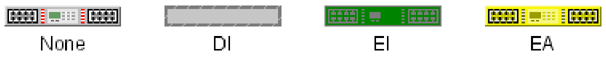
The following CSS extract customizes the graphical representation of all telecom business objects according to the value of the Bellcore primary state. This example does not take into account the possible presence of alarms in the objects.

```
object."ilog.tgo.model.IltObject" {
  foreground: '';
}
object."ilog.tgo.model.IltObject"["objectState.Bellcore.State"=DisabledIdle]
{
  foreground: lightGray;
}
```

```
object."ilog.tgo.model.IltObject"["objectState.Bellcore.State"=EnabledIdle] {

  foreground: green;
}
object."ilog.tgo.model.IltObject"["objectState.Bellcore.State"=EnabledActive]
 {
  foreground: yellow;
}
```

Bellcore state styling example illustrates this configuration on network elements.



Bellcore state styling example

SNMP states

The SNMP state dictionary defines a primary state and several secondary states, which are used to define the graphic representation of predefined business objects through cascading style sheets. See The SNMP state dictionary visuals for more information.

SNMP primary states

The following selector matches all objects that have the SNMP primary state `Up` defined:

```
object."ilog.tgo.model.IltObject"["objectState.SNMP.State"=Up] {
...
}
```

In CSS selectors, primary states are identified by the attribute name ("`objectState`") and the primary state information ("`SNMP.State`").

SNMP secondary states

You can also use secondary states in the attribute selectors. Secondary states are identified by the attribute name ("`objectState`") and the secondary state information. This information is based on the State Dictionary (SNMP) and the group to which the state belongs within this dictionary (Interface, IP, SNMP, EGP, TCP or UDP), as well as on the state name. See The SNMP state dictionary visuals for a complete list of available secondary states.

The following selector matches all objects that contain the secondary state `Interface.InErrors`:

```
object."ilog.tgo.model.IltObject"["objectState.SNMP.Interface.InErrors"] {
...
}
```

The following selector matches all objects that contain the secondary state `IP.InDiscards`:

```
object."ilog.tgo.model.IltObject"["objectState.SNMP.IP.InDiscards"] {
...
}
```

How to change the object representation based on SNMP states

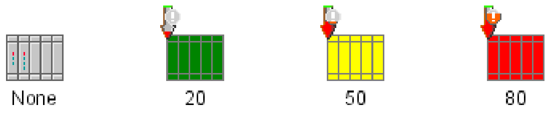
The following CSS extract customizes the graphical representation of all telecom business objects according to the value of the SNMP secondary state `InErrors`. This example changes the foreground color of the object when the value of the secondary state exceeds the given thresholds.

```
object."ilog.tgo.model.IltObject" {
  foreground: lightGray;
}
object."ilog.tgo.model.IltObject"["objectState.SNMP.Interface.InErrors"] {
  foreground: green;
}
object."ilog.tgo.model.IltObject"["objectState.SNMP.Interface.InErrors">40] {

  foreground: yellow;
}
object."ilog.tgo.model.IltObject"["objectState.SNMP.Interface.InErrors">70] {

  foreground: red;
}
```

SNMP state styling example illustrates this configuration on network elements.



SNMP state styling example

SONET states

The SONET state dictionary defines a primary state, as well as protections, which are used to define the graphical representation of predefined business objects through cascading style sheets. See The SONET state dictionary visuals for a complete list of available states and protections.

SONET primary states

The following selector matches all objects that have the SONET primary state `Active` defined:

```
object."ilog.tgo.model.IltObject"["objectState.SONET.State"=Active] {
...
}
```

In CSS selectors, primary states are identified by the attribute name ("`objectState`") and the primary state information ("`SONET.State`").

SONET protections

You can also use SONET protections in CSS selectors. Protections are identified by the attribute name ("`objectState`"), the position of the protection ("`SONET.FromProtections`" or "`SONET.ToProtections`"), and the protection name as in the following example:

```
object."ilog.tgo.model.IltLink"["objectState.SONET.FromProtections.Locked"] {
...
}

object."ilog.tgo.model.IltLink"["objectState.SONET.ToProtections.Pending"] {
...
}
```

SONET reverse primary state

The SONET state dictionary contains an extension that allows you to define a reverse primary state. This extension is mainly designed for link objects, so that you can define state information for both directions of the link connection.

In CSS selectors, reverse primary states are identified by the attribute name ("`objectState`") and the primary reverse state information ("`SONET.ReverseState`") as in the following example:

```
object."ilog.tgo.model.IltLink"["objectState.SONET.ReverseState"=Active] {
...
}
```

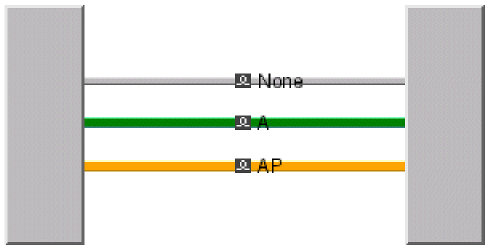
How to change the object representation based on SONET states

The following CSS extract customizes the graphic representation of links according to the value of the SONET primary state. This example changes the foreground color of the object according to the value of the primary state.

```
object."ilog.tgo.model.IltAbstractLink" {
  foreground: '';
}
object."ilog.tgo.model.IltAbstractLink"["objectState.SONET.State"=Active] {

  foreground: green;
}
object."ilog.tgo.model.IltAbstractLink"["objectState.SONET.State"=ActiveProtect
ing] {
  foreground: orange;
}
```

SONET state styling example illustrates this configuration.



SONET state styling example

Miscellaneous states

JViews TGO provides a set of miscellaneous states to complement the OSI, Bellcore, SONET, and SNMP standards. See The Misc state dictionary visuals for a complete list of available states.

Secondary miscellaneous states

You can use the secondary states of the Misc State Dictionary to specify cascading style sheet selectors.

Misc states are identified by the attribute name ("`objectState`"), the state information ("`Misc.SecState`"), and the state name as in the following example:

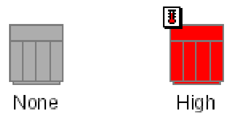
```
object."ilog.tgo.model.IltObject"["objectState.Misc.SecState.DoorAjar"] {
...
}
```

How to change the object representation based on miscellaneous states

The following CSS extract customizes the graphic representation of links according to the presence of the Miscellaneous State: High Temperature Warning.

```
object."ilog.tgo.model.IltNetworkElement" {
  foreground: '';
}
object."ilog.tgo.model.IltNetworkElement"["objectState.Misc.SecState.HighTemper
atureWarning"] {
  foreground: red;
}
```

Miscellaneous state styling example illustrates this configuration.



Miscellaneous state styling example

Performance states

JViews TGO provides a set of performance states to complement the OSI, Bellcore, SONET, and SNMP standards. See The Performance state dictionary visuals for a complete list of available states.

Secondary performance states

You can use the secondary states of the Performance State Dictionary to specify cascading style sheet selectors.

Performance states are identified by the attribute name ("`objectState`"), the state information ("`Performance.SecState`"), and the state name as in the following example:

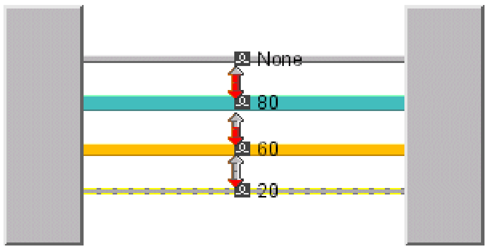
```
object."ilog.tgo.model.IltObject"["objectState.Performance.SecState.Output"]
{
...
}
```

How to change the object representation based on performance states

The following CSS extract customizes the graphical representation of links according to the value of the performance Bandwidth state. In this configuration, the width of the link is based on the current value of the state. When the value of the line width is set to a negative value, the link retrieves the default value.

```
object."ilog.tgo.model.IltAbstractLink" {
  lineWidth: -1;
}
object."ilog.tgo.model.IltAbstractLink"["objectState.Performance.SecState.Bandw
idth"] {
  lineWidth: @|@"objectState.Performance.SecState.Bandwidth"/10;
}
```

Performance state styling example illustrates this configuration.



Performance state styling example

SAN states

JViews TGO provides a set of SAN states to complement the OSI, Bellcore, SONET, and SNMP standards. See The SAN state dictionary visuals for a complete list of available states.

SAN secondary states

You can use the secondary states of the SAN State Dictionary to specify cascading style sheet selectors.

SAN states are identified by the attribute name ("`objectState`"), the state information ("SAN.`SecState`"), and the state name as in the following example:

```
object."ilog.tgo.model.IltObject"["objectState.SAN.SecState.Allocated"] {
...
}
```

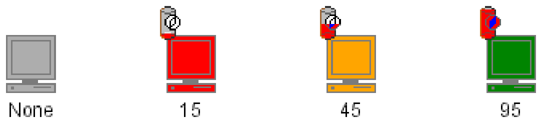
How to change the object representation based on SAN states

The following CSS extract customizes the graphic representation of network elements according to the value of the SAN Available secondary state. In this configuration, the color of the network element is based on the current value of the state. When the value of the foreground color is set to `null`, the network element retrieves the default configuration.

```
object."ilog.tgo.model.IltNetworkElement" {
  foreground: '';
}
object."ilog.tgo.model.IltNetworkElement"["objectState.SAN.SecState.Available"]

{
  foreground: green;
}
object."ilog.tgo.model.IltNetworkElement"["objectState.SAN.SecState.Available"<
20] {
  foreground: red;
}
object."ilog.tgo.model.IltNetworkElement"["objectState.SAN.SecState.Available"<
50] {
  foreground: orange;
}
```

SAN state styling example illustrates this configuration:



SAN state styling example

Alarms

JViews TGO provides a predefined alarm system, which modifies the graphical representation of telecom objects according to the alarm conditions. The Alarm dictionary can be used to complement the information of OSI, SONET and Bellcore state dictionaries. See Alarm states in the *Business Objects and Data Sources* documentation for more information.

Alarm conditions

You can use alarm conditions to customize the graphical representation of your objects through cascading style sheets.

In CSS selectors, new and acknowledged alarms are identified by the attribute name ("`objectState`"), the type of alarm (raw or impact), the alarm severity, and the information indicating whether it is a new or an acknowledged alarm.

The following selector matches all objects that have new critical alarms.

```
object."ilog.tgo.model.IltObject"["objectState.Alarm.Raw.Critical.New"] {
...
}
```

The following selector matches all objects that have critical acknowledged alarms.

```
object."ilog.tgo.model.IltObject"["objectState.Alarm.Raw.Critical.Acknowledged"

{
...
}
```

You can also customize the representation of objects based on impact alarms.

```
object."ilog.tgo.model.IltObject"["objectState.Alarm.Impact.CriticalHigh.New"
 {
...
}
```

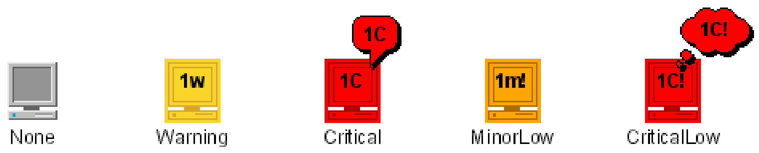
How to change the object representation based on alarms

The following CSS extract customizes the graphical representation of business objects according to the value of alarms set in the objects. In this configuration, the alarm balloon decoration is only displayed if there are new critical alarms.

```
object."ilog.tgo.model.IltObject" {
  alarmBalloonVisible: false;
}
object."ilog.tgo.model.IltObject"["objectState.Alarm.Raw.Critical.New"] {
  alarmBalloonVisible: true;
}
object."ilog.tgo.model.IltObject"["objectState.Alarm.Impact.CriticalLow.New"]
 {
  alarmBalloonVisible: true;
}
object."ilog.tgo.model.IltObject"["objectState.Alarm.Impact.CriticalHigh.New"]

{
  alarmBalloonVisible: true;
}
```

Alarm state styling example illustrates this configuration.



Alarm state styling example

Traps

JViews TGO provides a predefined trap system, which modifies the graphical representation of telecom objects according to the traps present in the objects. The Trap dictionary can be used to complement the information of the SNMP state dictionary. See Trap states in the *Business Objects and Data Sources* documentation for more information.

Trap conditions

You can use trap conditions to customize the graphic representation of your objects through cascading styles sheets.

In CSS selectors, new and acknowledged traps are identified by the attribute name ("`objectState`"), the trap type, and the information indicating whether it is a new or an acknowledged trap.

The following selector matches all objects that have new link failure traps.

```
object."ilog.tgo.model.IltObject"["objectState.Trap.LinkFailure.New"] {
...
}
```

The following selector matches all objects that have acknowledged link failure traps.

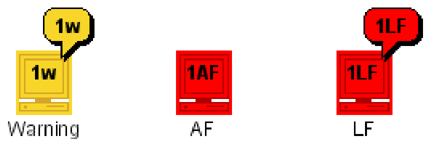
```
object."ilog.tgo.model.IltObject"["objectState.Trap.LinkFailure.Acknowledged"
 {
...
}
```

How to change the object representation based on traps

The following CSS extract customizes the graphic representation of business objects according to the value of traps set in the object. In this configuration, the alarm balloon decoration is hidden for all business objects that use the SNMP object state, except those that have new traps of type LinkFailure.

```
object."ilog.tgo.model.IltObject"["objectState.SNMP.State"] {
  alarmBalloonVisible: false;
}
object."ilog.tgo.model.IltObject"["objectState.Trap.LinkFailure.New"] {
  alarmBalloonVisible: true;
}
```

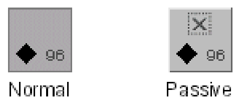
Trap state styling example illustrates this configuration.



Trap state styling example

Customizing passive devices

Passive devices are business objects without an object state. The passive information is graphically represented with an icon as illustrated in following image:



Properties for passive devices

You can customize a passive representation using the CSS properties listed in the following table.

CSS properties for passive devices

Property Name	Type	Default Value	Description
`passiveIcon`	`Image`	`ilog/tgo/ilt_passive.png`	Denotes the icon that is used in the passive representation.
`passiveIconVisible`	`boolean`	`false`	Denotes whether the passive icon is displayed or not.

How to represent passive devices

Predefined business objects can have states and alarms. When these objects do not have an object state set, they are considered to be passive devices. This information can be graphically represented by the passive icon. The following CSS extract shows how you can customize the predefined business objects to show this icon when the object does not have an object state:

```
object."ilog.tgo.model.IltObject" {
  passiveIconVisible: true;
}

object."ilog.tgo.model.IltObject"[objectState] {
  passiveIconVisible: false;
}
```

Customizing the OSI state system

The OSI state dictionary is based on the OSI SMF 10164-2 standard, which defines the primary state of a telecom object as a combination of three values, and also introduces a number of status values.

JViews TGO provides a visual representation of OSI states by using the object base to represent the primary state information, and secondary state icons to represent OSI status values. For information on the OSI state system and its graphic representation, refer to

OSI states explains how to customize the object representation according to the OSI state information. You can also customize the icons that are displayed when a given OSI status value is set in the object.

OSI states

Whether or not a given status applies to an object depends on whether the object is Out Of Service, In Service and Carrying Traffic, or In Service and Carrying No Traffic.

The OSI states corresponding to each status value are as follows:

- Out Of Service (OOS):
  - Operational: Disabled, Usage: Idle, Administrative: Unlocked
  - Operational: Disabled, Usage: Idle, Administrative: Locked
- In Service, Carrying No Traffic (NT):
  - Operational: Enabled, Usage: Idle, Administrative: Unlocked
  - Operational: Enabled, Usage: Idle, Administrative: Locked
- In Service, Carrying Traffic (CT):
  - Operational: Enabled, Usage: Active, Administrative: Unlocked
  - Operational: Enabled, Usage: Active, Administrative: Shutting down
  - Operational: Enabled, Usage: Busy, Administrative: Unlocked
  - Operational: Enabled, Usage: Busy, Administrative: Shutting down

You can customize the icon used to represent a status value by setting the image associated with the secondary state and the primary state combination.

How to customize an OSI status icon (using the API)

```
Image img =
IltSystem.GetDefaultContext().getImageRepository().getImage("logfull.png");
IltSettings.SetValue("OSI.Availability.LogFull.OOS.Icon", img);
IltSettings.SetValue("OSI.Availability.LogFull.NT.Icon", img);
IltSettings.SetValue("OSI.Availability.LogFull.CT.Icon", img);
```

In this example, the graphic representation of status `IltOSI.Availability.LogFull` has been replaced in all valid primary state combinations.

You can also set different images to different primary state combinations by using the state name, for example, “OSI.Availability.LogFull” or “OSI.Procedural.Initializing”, followed by the primary state combination (OOS, NT or CT) and the property name (Icon). For example:

```
IltSettings.SetValue("OSI.Availability.LogFull.OOS.Icon", img);
```

or

```
IltSettings.SetValue("OSI.Procedural.Initializing.NT.Icon", img);
```

How to customize an OSI status icon (using CSS)

You can also customize an OSI status icon by using global CSS settings (see *Using global settings* in for more information):

You must specify the full state name, for example "`OSI.Availability.LogFull`", when matching the "`name`" attribute. The CSS properties to be customized are `ctIcon`, `ntIcon`, `oosIcon`.

```
setting."ilog.tgo.model.IltState"[name="OSI.Availability.LogFull"] {
   ctIcon: '@|image("logfull.png")';
   ntIcon: '@|image("logfull.png")';
   oosIcon: '@|image("logfull.png")';
}
```

Customizing the Bellcore state system

In the Bellcore state dictionary, a primary state is defined as holding one of the following values:

- ♦ Disabled/Idle
- ♦ Enabled/Idle
- ♦ Enabled/Active

The Bellcore dictionary also includes numerous secondary states.

JViews TGO provides a visual representation for the Bellcore primary state by changing the object base graphic representation. Bellcore secondary states are represented using secondary state icons. For information on the Bellcore state system and its graphic representation, refer to

Bellcore states explains how to customize the object representation according to the Bellcore state information. You can also customize the icons that are displayed when a given Bellcore secondary state is set in the object.

Bellcore states

Whether or not a secondary state can be applied meaningfully to a telecom object depends on whether the object is in the Out of Service, No Traffic, or Carrying Traffic condition.

The corresponding Bellcore primary states are the following:

- ♦ OOS--Disabled/Idle
- ♦ NT--Enabled/Idle
- ♦ CT--Enabled/Active

You can customize the icon used to represent a secondary state by setting the image associated with the secondary state and the primary state combination, as follows:

How to customize a Bellcore secondary state icon (using the API)

```
Image img =
IltSystem.GetDefaultContext().getImageRepository().getImage("blocked.png");
IltSettings.SetValue("Bellcore.SecState.Blocked.CT.Icon", img);
```

In this example, the graphic representation of status `IltBellcore.SecState.Blocked` has been replaced in the Carrying Traffic representation.

You can also set different images for different combinations by using the state name, for example, “Bellcore.SecState.Busy” or “Bellcore.SecState.Transferred”, followed by the primary state combination (OOS, NT or CT) and the property name (Icon).

```
IltSettings.SetValue("Bellcore.SecState.Transferred.OOS.Icon", img);
```

How to customize a Bellcore secondary state icon (using CSS)

You can customize a Bellcore secondary state icon using global CSS settings.

You must specify the full state name, for example "`Bellcore.SecState.Busy`", when matching the "`name`" attribute. The CSS properties to be customized are `ctIcon`, `ntIcon`, `oosIcon`.

```
setting."ilog.tgo.model.IltState"[name="Bellcore.SecState.Blocked"] {
   ctIcon: '@|image("blocked.png")';
}
setting."ilog.tgo.model.IltState"[name="Bellcore.SecState.Diagnostic"] {
   ntIcon: '@|image("diagnostic.png")';
}
setting."ilog.tgo.model.IltState"[name="Bellcore.SecState.Transferred"] {
   oosIcon: '@|image("transferred.png")';
}
```

For more information on global CSS settings, see .

Customizing the SNMP state system

Describes the SNMP states and how to customize them.

In this section

SNMP primary and secondary states
Describes the SNMP states, discusses customization of primary and secondary states and describes how to create an SNMP primary state.

Customizing SNMP secondary states
Describes how to customize each graphical representation of SNMP secondary states.

Creating a new attribute in the System group
Describes how to create a new attribute in the group used for SNMP System attributes.

SNMP primary and secondary states

The SNMP state dictionary is based on RFC 1213 - Management Information Base for Network Management of TCP/IP-based internets - MIB-II.

JViews TGO provides a visual representation of the SNMP primary state by changing the object base graphic representation. SNMP secondary states are represented using icons, gauges, charts or counters. For information on the SNMP state system and its graphic representation, refer to

SNMP states explains how to customize the object representation according to the SNMP state information. You can also customize the decoration that is displayed when a given SNMP secondary state is set in the object.

In the SNMP state dictionary, most of the secondary states are numeric and by default they are represented by a gauge. There are also two other possible representations for these states. The `IltDecorationType` class defines the possible graphical representations. The possible values of this class are:

- Gauge
- Chart
- Counter

How to create an SNMP primary state

Each primary state in the SNMP state dictionary is associated with a different base style.

To create a new primary state:

- Create a new SNMP state using the method `NewState(java.lang.String, java.lang.String)`.

  ```
  IltSNMP.State state = IltSNMP.NewState ("Pending", "Indicates if the object

  is waiting for validation");
  ```

  This method takes two arguments: a name and a description. The name is used to identify the state in the application. The description is used to provide information about the semantics of the state.

  The new state can be used in XML in the following way.

  ```
  <state>Pending</state>
  ```

- Create a CSS file that configures the representation of the objects according to the new state.

  ```
  object."ilog.tgo.model.IltObject"["objectState.SNMP.State"=Pending] {
    foreground: red;
    background: yellow;
  ```

```
  pattern: '@|pattern("SkewGrid", 8, 2)';
  lineStyle: "1.000001, 7.000001";
}
```

For more information on CSS, see *Introducing cascading style sheets*. For more information on how to customize the object representation based on states, see *Customizing the object representation based on states*.

♦ Create a selector that resets the values defined by the state selector, so that the normal configuration is still applied to the objects when the state changes.

```

object."ilog.tgo.model.IltObject" {
  foreground: '';
  background: '';
  pattern: '';
  lineStyle: '';
}
```

Customizing SNMP secondary states

How to customize an SNMP secondary state decoration (using the API)

SNMP secondary states can be displayed as charts, gauges or counters. By default, the gauge decoration is used. If you want to indicate that you are going to use another type of representation, without changing the default configuration for each representation, modify the secondary state information as follows.

```
IltSettings.SetValue("SNMP.Interface.InOctets.Type", IltDecorationType.Chart)
;
```

The property to be set is based on the SNMP state name followed by “`.Type`”, for example, `SNMP.ICMP.InMsgs.Type` or `SNMP.TCP.OutSegs.Type`.

How to customize an SNMP secondary state decoration (using CSS)

You can customize an SNMP secondary state decoration using global CSS settings.

You must specify the full state name, for example "`SNMP.Interface.InOctets`", when matching the "`name`" attribute. The CSS property to be customized is `type`.

```
setting."ilog.tgo.model.IltState"[name="SNMP.Interface.InOctets"] {
   type: Counter;
}
```

For more information on global CSS settings, see

How to modify the gauge graphical representation of an SNMP secondary state (using the API)

When an SNMP secondary state is graphically represented by a gauge decoration, the configuration of this decoration has to be previously defined. All SNMP secondary states have a predefined gauge representation. For information on how each secondary state is graphically represented, refer to

Suppose you have a secondary state that is configured with a gauge decoration as follows.

```
IltSettings.SetValue("SNMP.ICMP.OutErrors.Type", IltDecorationType.Gauge);
```

You can modify this decoration in the following way.

```
IltColorModifier modifier1 = new IltColorModifier.Shade(0.5f);
IltColorModifier modifier2 = new
```

```
    IltColorModifier.MultiColor(IltColorModifier.MultiColor.USE_LAST_VALUE);
IltColorModifier modifier = modifier1.compose(modifier2);

IltGaugeMapping mapping = (IltGaugeMapping)
    IltSettings.GetValue("SNMP.ICMP.OutErrors.Gauge");
mapping.setColorModifier(modifier);
```

You can also create a new gauge mapping, and set the new value in the following way.

```
IltGaugeMapping mapping = new IltGaugeMapping(minImg, maxImg);
IltSettings.SetValue("SNMP.ICMP.OutErrors.Gauge", mapping);
```

How to modify the gauge graphical representation of an SNMP secondary state (using CSS)

You can customize the gauge graphical representation using global CSS settings. You must specify the full state name, for example "`SNMP.Interface.InOctets`", when matching the "`name`" attribute. The CSS property to be customized is `gauge`.

When defining a new gauge mapping class, the following CSS properties are used: `minImage`, `maxImage`, `direction`, `colorModifier`. They match corresponding `set` methods of the `IltGaugeMapping` class (and its super class).

```
setting."ilog.tgo.model.IltState"[name="SNMP.ICMP.OutErrors"] {
  gauge: @+gaugeSnmp;
}
Subobject#gaugeSnmp {
  class: 'ilog.tgo.graphic.IltGaugeMapping';
  minImage: '@|image("icon1.png")';
  maxImage:'@|image("icon2.png")';
  direction: Bottom;
  colorModifier: @+myColorModifier;
}
Subobject#myColorModifier {
  class: "MyColorModifier";
}
```

In this example, the color modifier class named `MyColorModifier` has been created and included in the search path.

For information on how to customize the gauge decorations, refer to `ilog.tgo.graphic.IltGaugeMapping`.

How to modify the chart graphical representation of an SNMP secondary state (using the API)

When an SNMP secondary state is graphically represented by a chart decoration, the configuration of this decoration has to be previously defined. All SNMP secondary states have a predefined chart representation. Refer to The SNMP state dictionary in the *Business Objects and Data Sources* documentation for information on how each secondary state is graphically represented.

Suppose you have a secondary state that is configured with a chart decoration as follows:

```
IltSettings.SetValue("SNMP.ICMP.OutErrors.Type", IltDecorationType.Chart);
```

You can modify this decoration in the following way:

```
IltColorModifier modifier1 = new IltColorModifier.Shade(0.5f);
IltColorModifier modifier2 =
  new IltColorModifier.MultiColor(IltColorModifier.MultiColor.USE_LAST_VALUE)
;
IltColorModifier modifier = modifier1.compose(modifier2);

IltChartMapping mapping = (IltChartMapping)
  IltSettings.GetValue("SNMP.ICMP.OutErrors.Chart");
mapping.setColorModifier(modifier);
```

You can also create a new chart mapping, and set the new value in the following way:

```
IltChartMapping mapping = new IltChartMapping(minImg, maxImg);
IltSettings.SetValue("SNMP.ICMP.OutErrors.Chart", mapping);
```

How to modify the chart graphical representation of an SNMP secondary state (using CSS)

You can also customize the chart graphic representation by using global CSS settings. You must specify the full state name, for example "`SNMP.ICMP.OutErrors`", when matching the "`name`" attribute. The CSS property to be customized is `chart`.

When defining a new chart mapping class, the following CSS properties are used: `minImage`, `maxImage`, `xAxisDirection`, `yAxisDirection`, `colorModifier`. They match corresponding `set` methods of the `IltChartMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltChartMapping` for information on how to customize the chart decorations.

In the example below, the color modifier class named `MyColorModifier` has been created and included in the search path.

```
setting."ilog.tgo.model.IltState"[name="SNMP.ICMP.OutErrors"] {
  chart: @+chartSnmp;
}
Subobject#chartSnmp {
  class: 'ilog.tgo.graphic.IltChartMapping';
  minImage: '@|image("icon1.png")';
  maxImage:'@|image("icon2.png")';
  xAxisDirection: Right;
  yAxisDirection: Bottom;
  colorModifier: @+myColorModifier;
}
Subobject#myColorModifier {
  class: "MyColorModifier";
}
```

How to modify the counter graphical representation of an SNMP secondary state (using the API)

When an SNMP secondary state is graphically represented by a counter decoration, the configuration of this decoration has to be previously defined. All SNMP secondary states have a predefined counter representation. Refer to The SNMP state dictionary in the *Business Objects and Data Sources* documentation for information on how each secondary state is graphically represented.

Suppose you have a secondary state that is configured with a counter decoration as follows:

```
IltSettings.SetValue("SNMP.ICMP.OutErrors.Type", IltDecorationType.Counter);
```

You can modify this decoration in the following way:

```
IltCounterMapping cm = new IltCounterMapping(new DecimalFormat("#Mbs"),
                                             IltrFont.CounterText,
                                             true, Color.black,
                                             Color.white, Color.black,
                                             3, 1, IlvDirection.Right, false)
;
IltSettings.SetValue("SNMP.ICMP.OutErrors.Counter", cm);
```

How to modify the counter graphical representation of an SNMP secondary state (using CSS)

You can also customize the counter graphic representation by using global CSS settings. You must specify the full state name, for example "`SNMP.ICMP.OutErrors`", when matching the "`name`" attribute. The CSS property to be customized is `counter`.

When defining a new counter mapping class, the following CSS properties are used: `format`, `font`, `antialiasing`, `foregroundColor`, `backgroundColor`, `borderColor`, `xPadding`, `yPadding`. They match corresponding `set` methods of the `IltCounterMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltCounterMapping` for information on how to customize the counter decorations.

```
setting."ilog.tgo.model.IltState"[name="SNMP.ICMP.OutErrors"] {
  counter: @+counterSnmp;
}
Subobject#counterSnmp {
  class: 'ilog.tgo.graphic.IltCounterMapping';
  format: @+myNumberFormat;
  font: "Helvetica-Bold-12";
  antialiasing: false;
  foregroundColor: red;
  backgroundColor: blue;
  borderColor: red;
  xPadding: 100;
  yPadding: 100;
}
Subobject#myNumberFormat{
  class: 'MyNumberFormat';
}
```

Creating a new attribute in the System group

JViews TGO defines a specific attribute group to store SNMP System attributes. You can retrieve this attribute group with the method `GetSystemAttributeGroup()`. The attributes present in this group are business object attributes, as defined in `IlpAttribute`.

JViews TGO provides an attribute class `IltAttribute`), which you can use directly when customizing the SNMP System group.

To extend the attributes in the SNMP System group:

♦ Create a new attribute with its type and attribute group.

```
IltAttribyte myAttribute = new IltAttribute("address",
                           String.class,
                           IltSNMP.GetSystemAttributeGroup());
```

♦ Register the attribute in the SNMP System Group.

  To do so, use the method `SetAttributeMapping(ilog.cpl.model.IlpAttribute, java.lang.Object, ilog.cpl.style.key.IlpStringKey)`. This method allows the attribute to be automatically represented in the object System Window.

```
IltSNMP.SetAttributeMapping(myAttribute, defaultValue, "Address");
```

♦ Customize the representation of this attribute through cascading style sheets.

  When adding a new attribute to the System Window, create a selector for the object and attribute (`object."ilog.tgo.model.IltObject/address"`). This selector contains the following properties:

  - `visibleInSystemWindow`: indicates that the attribute is an entry in the System window
  - `captionLabelVisible`: indicates that a label prefixes the attribute value
  - `captionLabel`: indicates the value of the label
  - `label`: indicates the value of the attribute to be displayed in the System window

  The following example illustrates the way to customize an attribute inside the System Window.

```
object."ilog.tgo.model.IltObject/address" {
  visibleInSystemWindow: true;
  captionLabel: Address;
  captionLabelVisible: true;
  label: @address;
}
```

Customizing the SONET state system

Describes the SONET states and how to customize them.

In this section

Customizing SONET states

Describes how to create a new SONET primary state and how to configure it for use as a BiSONET state.

Customizing SONET protection states

Describes how to create a new SONET protection state and how to customize the associated icon.

Customizing SONET states

The SONET State Dictionary groups states and indicators that are used most often to display transport links with the protection process. Such link state graphics are useful only in applications in which the end user must be informed about link states and protection switching information (as in a fiber transport network, for example).

For a description of the state system and its graphical representation, refer to

SONET states explains how to customize the object representation according to the SONET state information.

How to customize the SONET state system

To create a new SONET primary state:

♦ Create the new state using the method `NewState(java.lang.String, java.lang.String)`.

  This method takes two arguments: a name and a description. The name is used to identify the state in the application. The description is used to provide information about the semantics of the state.

```
IltState inErrorState = IltSONET.NewState("InError", "In error state");
```

  The state defined above can be used in XML in the following way:

```
<state>InError</state>
```

♦ Customize the primary state representation using an attribute-based CSS selector. For more information on CSS, see section *Introducing cascading style sheets* in this documentation. For more information on how to customize the object representation based on states, see *Customizing the object representation based on states*.

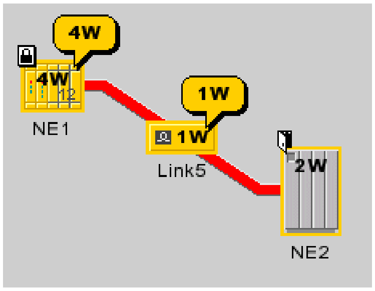
  The following CSS selector modifies the configuration of all `IltObject` instances that have the primary state `SONET InError`.

```
object."ilog.tgo.model.IltObject"["objectState.SONET.State"=InError] {
   foreground: red;
   centerWidth: 6;
   reliefBorders: true;
}
```

The following example illustrates how the new SONET state is represented.

```
IltLink link = new IltLink(new IltSONETObjectState(), null);
link.setState(InErrorState);
```

The graphical results can be seen in *Customizing the SONET state system*.



Customizing the SONET state system

How to use the new SONET state as a BiSONET state

A new state can also be used as a BiSONET state. In this case, you should define how the state will be represented as a state and as a reverse state.

♦ Customize the link representation when it uses the BiSONET object state and its primary state is set to the newly created state. This configuration is illustrated by the following selector which indicates all objects that have the primary state `InError` and contain a reverse state.

```
object."ilog.tgo.model.IltLink"["objectState.SONET.State"=InError]["objectSt
ate.SONET.ReverseState"] {
  foreground:'';
  innerCenterWidth: 2;
  innerForeground: red;
  toArrowColor: red;
  toArrowReliefBorders: true;
  toArrowBorderColor: red;
  toArrowBorderColor2: red;
}
```

♦ Customize the link representation when the new state is set to the reverse state. Use the following selector.

```
object."ilog.tgo.model.IltLink"["objectState.SONET.ReverseState"=InError]
{
  fromArrowColor: red;
  fromArrowReliefBorders: true;
  fromArrowBorderColor: red;
  fromArrowBorderColor2: red;
  foreground: red;
  centerWidth: 6;
}
```

♦ Define a generic rule to be matched when the new state is not set. This generic rule specifies that, for the states that are not defined in the cascading style sheets, the default configuration will be applied. The corresponding selector simply sets all properties set by the other selectors to their default value, so that the configuration of the other SONET states is appropriately applied to the objects.

```
object."ilog.tgo.model.IltLink" {
  foreground: '';
  centerWidth: -1;
  reliefBorders: '';
  innerCenterWidth: -1;
  innerForeground:'';
  toArrowColor:'';
  toArrowBorderColor:'';
  toArrowBorderColor2:'';
  toArrowReliefBorders: '';
  fromArrowColor:'';
  fromArrowBorderColor:'';
  fromArrowBorderColor2:'';
  fromArrowReliefBorders: '';
}
```

The created state can be set on the object in the following way.

```
IltBiSONETObjectState ostate = new IltBiSONETObjectState();
ostate.setState(IltSONET.State.ActiveProtecting);
ostate.setReverseState(inErrorState);
IltLink link = new IltLink(ostate, null);
```

Customizing SONET protection states

The SONET state system defines a set of protections that can be associated with link objects. You can create new SONET protections, as well as configure how new and existing protections are graphically represented in your objects.

How to create a new SONET protection state (using the API)

Create a new protection state using the method `NewProtection(java.lang.String, java.lang.String)`. This method takes two arguments: a name and a description.

```
IltSONET.Protection protection = IltSONET.NewProtection("InTest",
   "Indicates that the communication channel is currently under test");
```

The name is used to identify the state in the application. When a new protection state is created with the name InTest, it is referred to in XML in the following way:

```
<protection>InTest</protection>
```

The description is used to provide information about the semantics of the state. It can be used as a tooltip to display more information about the protection state represented in a business object. To activate the tooltip support for secondary states, see *Customizing tooltips*.

How to create a new SONET protection state (using CSS)

You can create new protection states using global CSS settings.

```
Settings {
   sonet: true;
}
SONET {
   protections[0]: @+prot0;
}
Subobject#prot0 {
  class: 'ilog.tgo.model.IltSONET.Protection';
  name: "InTest";
}
```

How to customize SONET protection states (using the API)

Each SONET protection state is associated with an icon that can be customized using `SetValue(java.lang.Object, java.lang.Object)`. This method requires two arguments, the first argument being the property key name. SONET Protection state property names are formed by: “SONET.Protection.<YOUR STATE NAME>.Icon”

```
IlpImageRepository imageRep =
IltSystem.GetDefaultContext().getImageRepository();
```

```
Image problemImage = imageRep.getImage("problem.png");
IltSettings.SetValue("SONET.Protection.Problem.Icon", problemImage);
```

or

```

IltSettings.SetValue("SONET.Protection.Exercisor.Icon", img);
```

How to customize SONET protection states (using CSS)

You can customize SONET protection states using global CSS settings. For more information, see *Using global settings*. You must specify the full state name, for example "`SONET.Protection.Problem`", when matching the "`name`" attribute. The CSS property to be customized is `icon`.

```

setting."ilog.tgo.model.IltState"[name="SONET.Protection.Problem"] {
   icon: '@|image("problem.png")';
}
```

Customizing the Miscellaneous state system

The Miscellaneous State Dictionary provides secondary state values that can be used to complement OSI, Bellcore or SNMP standards. The secondary states included in this dictionary are often used in telecommunication network supervision applications. For information on the dictionary and its graphical representation, refer to

You can create your own miscellaneous secondary state, and configure how new and existing secondary states are graphically represented in your objects.

How to create new Miscellaneous states (using the API)

```
IltMisc.SecState state = IltMisc.NewSecState("Misc.SecState.Problem", "Severe

Problem");
```

`NewSecState(java.lang.String, java.lang.String)` takes two arguments. The first one is a name that identifies the new miscellaneous state in the application. The name must comply with the rule `Misc.SecState.<value>`. For example, when a new state is created with name `Misc.SecState.Problem`, it is referred to in XML files in the following way:

```
<misc>Problem</misc>
```

The second argument is a description. This description is used as a tooltip to display more information about the miscellaneous state represented in a business object. To activate the tooltip support for secondary states, see *Customizing tooltips*.

`NewSecState(java.lang.String, java.lang.String)` returns an instance of `IltMisc.SecState` that can be used in exactly the same way as the predefined Miscellaneous secondary states.

How to create new Miscellaneous states (using CSS)

You can also create new miscellaneous states by using global CSS settings:

```
Settings {
   misc: true;
}
Misc {
   states[0]: @+misc0;
}
Subobject#misc0 {
  class: 'ilog.tgo.model.IltMisc.SecState';
  name: "Misc.SecState.Problem";
}
```

How to customize Miscellaneous states (using the API)

Each Miscellaneous secondary state is associated with an icon that can be customized using `IltSettings.SetValue`. This method requires two arguments, the first argument being the property key name. Miscellaneous secondary state property names are formed by: “<YOUR STATE NAME>.Icon”.

```
IlpImageRepository imageRep =
IltSystem.GetDefaultContext().getImageRepository();
Image problemImage = imageRep.getImage("problem.png");
IltSettings.SetValue("Misc.SecState.Problem.Icon", problemImage);
```

or

```
IltSettings.SetValue("Misc.SecState.HighTemperatureWarning.Icon", img);
```

How to customize Miscellaneous states (using CSS)

You can also customize miscellaneous states by using global CSS settings. For more information, see *Using global settings*. You must specify the full state name, for example "`Misc.SecState.Problem`", when matching the "`name`" attribute. The CSS property to be customized is `icon`.

```
setting."ilog.tgo.model.IltState"[name="Misc.SecState.Problem"] {
   icon: '@|image("problem.png")';
}
```

Customizing the Performance State System

Describes the Performance secondary states and how to create and customize them.

In this section

Creating new Performance secondary states
Describes how to create a new Performance secondary state.

Customizing Performance secondary states
TO BE DEFINED

Creating new Performance secondary states

The Performance State Dictionary provides secondary state values that can be used to complement OSI, Bellcore, SNMP, or SONET standards. The secondary states included in this dictionary can be used to model and represent any state with a numeric value. For information on the dictionary and its graphical representation, refer to

You can create your own performance secondary state, and configure how new and existing secondary states are graphically represented in your objects.

How to create new Performance secondary states (using the API)

Performance secondary states are created using `IltPerformance.NewSecState`, as follows.

```
IltPerformance.SecState state =
   IltPerformance.NewSecState("Performance.SecState.ResponseTime",
      "Measures the ellapsed time between a request and response operations.
");
```

`NewSecState(java.lang.String, java.lang.String)` takes two arguments. The first one is a name that identifies the new performance state in the application. The name must comply with the rule `Performance.SecState.<value>`. For example, when a new state is created with name `Performance.SecState.ResponseTime`, it is referred to in XML files in the following way:

```
<performance state="ResponseTime">33.2</performance>
```

The second argument is a description. This description is used as a tooltip to display more information about the performance state represented in a business object. To activate the tooltip support for secondary states, see *Customizing tooltips*.

`NewSecState(java.lang.String, java.lang.String)` returns an instance of `IltPerformance.SecState`, that you can use in exactly the same way as the predefined Performance secondary states.

Once a new performance secondary state has been created, you need to configure its graphical representation. See *Customizing Performance secondary states* for more information.

How to create new Performance secondary states (using CSS)

You can create new Performance secondary states using global CSS settings:

```
Settings {
   performance: true;
}
Performance {
   states[0]: @+perf0;
}
```

```
Subobject#perf0 {
  class: 'ilog.tgo.model.IltPerformance$SecState';
  name: "Performance.SecState.ResponseTime";
}
```

Customizing Performance secondary states

Describes how to configure each representation of Performance secondary states.

Performance states explains how to customize the object representation according to the Performance state information. You can also customize the decoration that is displayed when a given Performance secondary state is set in the object.

In the Performance state dictionary, the secondary states are numeric and by default they are represented by a gauge. There are also two other possible representations for these states. The `IltDecorationType` class defines the possible graphical representations. The possible values of this class are:

- Gauge
- Chart
- Counter

How to customize a Performance secondary state decoration (using the API)

Performance secondary states can be displayed as charts, gauges or counters. By default, the gauge decoration is used. If you want to indicate that you are going to use another type of representation, without changing the default configuration for each representation, modify the secondary state information as follows:

```
IltSettings.SetValue("Performance.SecState.Imput.Type",
IltDecorationType.Chart);
```

The property to be set is built in the following way: “<STATE NAME>.Type”. For example, `Performance.SecState.In.Type` or `Performance.SecState.Voltage.Type`.

How to customize a Performance secondary state decoration (using CSS)

You can also customize a performance secondary state decoration by using global CSS settings (see *Using global settings* in *Using Cascading Style Sheets* for more information):

You must specify the full state name, for example "`Performance.SecState.Input`", when matching the "`name`" attribute. The CSS property to be customized is `type`.

```
setting."ilog.tgo.model.IltState"[name="Performance.SecState.Input"] {
   type: Chart;
}
```

How to modify the Gauge graphical representation of a Performance secondary state (using the API)

When a Performance secondary state is graphically represented by a gauge decoration, the configuration of this decoration has to be previously defined. All Performance secondary states have a predefined gauge representation.

For information on how each secondary state is graphically represented, refer to

Suppose you have a secondary state that is configured with a gauge decoration as follows.

```
IltSettings.SetValue("Performance.SecState.Input.Type",
IltDecorationType.Gauge);
```

You can modify this decoration in the following way.

```
IltColorModifier modifier1 = new IltColorModifier.Shade(0.5f);
IltColorModifier modifier2 = new
    IltColorModifier.MultiColor(IltColorModifier.MultiColor.USE_LAST_VALUE);
IltColorModifier modifier = modifier1.compose(modifier2);

IltGaugeMapping mapping = (IltGaugeMapping)
    IltSettings.GetValue("Performance.SecState.Input.Gauge");
mapping.setColorModifier(modifier);
```

You can also create a new gauge mapping, and set the new value in the following way.

```
IltGaugeMapping mapping = new IltGaugeMapping(minImg, maxImg);
IltSettings.SetValue("Performance.SecState.Input.Gauge", mapping);
```

How to modify the Gauge graphical representation of a Performance secondary state (using CSS)

You can also customize the gauge graphic representation by using global CSS settings. You must specify the full state name, for example "`Performance.SecState.Input`", when matching the "`name`" attribute. The CSS property to be customized is `gauge`.

When defining a new gauge mapping class, the following CSS properties are used: `minImage`, `maxImage`, `direction`, `colorModifier`. They match corresponding `set` methods of the `IltGaugeMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltGaugeMapping` for information on how to customize the gauge decorations.

In the following example, the color modifier class named `MyColorModifier` has been created and included in the search path.

```
setting."ilog.tgo.model.IltState"[name="Performance.SecState.Input"] {
  gauge: @+gaugePerf;
}
Subobject#gaugePerf {
  class: 'ilog.tgo.graphic.IltGaugeMapping';
  minImage: '@|image("icon1.png")';
  maxImage:'@|image("icon2.png")';
  direction: Bottom;
  colorModifier: @+myColorModifier;
}
Subobject#myColorModifier {
  class: "MyColorModifier";
}
```

How to modify the Chart graphical representation of a Performance secondary state (using the API)

When a Performance secondary state is graphically represented by a chart decoration, the configuration of this decoration has to be previously defined. All Performance secondary states have a predefined chart representation. Refer to Performance states: the Performance state dictionary in the *Business Objects and Data Sources* documentation for information on how each secondary state is graphically represented.

Suppose you have a secondary state that is configured with a chart decoration as follows.

```
IltSettings.SetValue("Performance.SecState.Input.Type",
IltDecorationType.Chart);
```

You can modify this decoration in the following way.

```
IltColorModifier modifier1 = new IltColorModifier.Shade(0.5f);
IltColorModifier modifier2 =
  new IltColorModifier.MultiColor(IltColorModifier.MultiColor.USE_LAST_VALUE)
;
IltColorModifier modifier = modifier1.compose(modifier2);

IltChartMapping mapping = (IltChartMapping)
    IltSettings.GetValue("Performance.SecState.Input.Chart");
mapping.setColorModifier(modifier);
```

You can also create a new chart mapping, and set the new value in the following way:

```
IltChartMapping mapping = new IltChartMapping(minImg, maxImg);
IltSettings.SetValue("Performance.SecState.Input.Chart", mapping);
```

How to modify the Chart graphical representation of a Performance secondary state (using CSS)

You can customize the chart graphic representation using global CSS settings. You must specify the full state name, for example "`Performance.SecState.Input`", when matching the "`name`" attribute. The CSS property to be customized is `chart`.

When defining a new chart mapping class, the following CSS properties are used: `minImage`, `maxImage`, `xAxisDirection`, `yAxisDirection`, `colorModifier`. They match corresponding `set` methods of the `IltChartMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltChartMapping` for information on how to customize the chart decorations.

In the following example, the color modifier class named `MyColorModifier` has been created and included in the search path.

```
setting."ilog.tgo.model.IltState"[name="Performance.SecState.Input"] {
  chart: @+chartPerf;
}
Subobject#chartPerf {
  class: 'ilog.tgo.graphic.IltChartMapping';
  minImage: '@|image("icon1.png")';
  maxImage:'@|image("icon2.png")';
  xAxisDirection: Right;
  yAxisDirection: Bottom;
  colorModifier: @+myColorModifier;
}
Subobject#myColorModifier {
  class: "MyColorModifier";
}
```

How to modify the Counter graphical representation of a Performance secondary state (using the API)

When a Performance secondary state is graphically represented by a counter decoration, the configuration of this decoration has to be previously defined. All Performance secondary states have a predefined counter representation. For information on how each secondary state is graphically represented, refer to

Suppose you have a secondary state that is configured with a counter decoration as follows.

```
IltSettings.SetValue("Performance.SecState.Input.Type",
IltDecorationType.Counter);
```

You can modify this decoration in the following way.

```
IltCounterMapping cm = new IltCounterMapping(new DecimalFormat("#Mbs"),
                                             IltrFont.CounterText,
                                             true, Color.black,
                                             Color.white, Color.black,
                                             3, 1, IlvDirection.Right, false)
;
IltSettings.SetValue("Performance.SecState.Input.Counter", cm);
```

How to modify the Counter graphical representation of a Performance secondary state (using CSS)

You can customize the counter graphical representation by using global CSS settings. You must specify the full state name, for example "`Performance.SecState.Input`", when matching the "`name`" attribute. The CSS property to be customized is `counter`.

When defining a new counter mapping class, the following CSS properties are used: `format`, `font`, `antialiasing`, `foregroundColor`, `backgroundColor`, `borderColor`, `xPadding`, `yPadding`. They match corresponding `set` methods of the `IltCounterMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltCounterMapping` for information on how to customize the counter decorations.

```
setting."ilog.tgo.model.IltState"[name="Performance.SecState.Input"] {
  counter: @+counterPerf;
}
Subobject#counterPerf {
  class: 'ilog.tgo.graphic.IltCounterMapping';
  format: @+myNumberFormat;
  font: "Helvetica-Bold-12";
  antialiasing: false;
  foregroundColor: red;
  backgroundColor: blue;
  borderColor: red;
  xPadding: 100;
  yPadding: 100;
}
Subobject#myNumberFormat{
  class: 'MyNumberFormat';
}
```

Customizing the SAN state system

Describes the SAN secondary states and how to create and customize them.

In this section

Creating new SAN secondary states
Describes how to create a new SAN secondary state.

Customizing SAN Secondary States
Describes how to configure each representation of SAN secondary states.

Creating new SAN secondary states

The SAN (Storage Area Network) State Dictionary provides secondary state values that can be used to complement OSI, Bellcore, SNMP, or SONET standards. The secondary states included in this dictionary can be used to model and represent any state with a numeric value. For information on the dictionary and its graphical representation,

Make cross-ref to SAN States: the SAN State Dictionary

refer to SAN states: the SAN state dictionary in the *Business Objects and Data Sources* documentation .

You can create your own SAN secondary state, and configure how new and existing secondary states are graphically represented in your objects.

How to create new SAN states (using the API)

SAN secondary states are created using `IltSAN.NewSecState`, as follows:

```
IltSAN.SecState state =
   IltSAN.NewSecState("SAN.SecState.CPUUtilization","Measures the CPU use
rate");
```

`NewSecState(java.lang.String, java.lang.String)` takes two arguments. The first one is a name that identifies the new SAN state in the application. The name must comply with the rule `SAN.SecState.<value>`. For example, when a new state is created with name `SAN.SecState.CPUUtilization`, it is referred to in XML files in the following way:

```
<SAN state="CPUUtilization">78</SAN>
```

The second argument is a description. This description is used as a tooltip to display more information about the SAN state represented in a business object. To activate the tooltip support for secondary states, see *Customizing tooltips*.

`NewSecState(java.lang.String, java.lang.String)` returns an instance of `IltSAN.SecState`, that you can use in exactly the same way as the predefined SAN secondary states.

Once a new SAN secondary state has been created, you need to configure its graphic representation. See *Customizing SAN Secondary States* for more information.

How to create new SAN states (using CSS)

You can also create new SAN secondary states by using global CSS settings:

```
Settings {
   san: true;
}
SAN {
   states[0]: @+san0;
}
```

```
Subobject#san0 {
  class: 'ilog.tgo.model.IltSAN.SecState';
  name: "SAN.SecState.CPUUtilization";
}
```

Customizing SAN Secondary States

SAN states explains how to customize the object representation according to the SAN state information. You can also customize the decoration that is displayed when a given SAN secondary state is set in the object.

In the SAN state dictionary, the secondary states are numeric and by default they are represented by a gauge. There are also two other possible representations for these states. The `IltDecorationType` class defines the possible graphical representations. The possible values of this class are:

♦ Gauge

♦ Chart

♦ Counter

How to customize a SAN secondary state decoration (using the API)

SAN secondary states can be displayed as charts, gauges or counters. By default, the gauge decoration is used. If you want to indicate that you are going to use another type of representation, without changing the default configuration for each representation, modify the secondary state information as follows:

```
IltSettings.SetValue("SAN.SecState.IO.Type", IltDecorationType.Chart);
```

The property to be set is built in the following way: “<STATE NAME>.Type”. For example, `SAN.SecState.Available.Type` or `SAN.SecState.CPUUtilization.Type`.

How to customize a SAN secondary state decoration (using CSS)

You can also customize a SAN secondary state decoration by using global CSS settings. For more information, see *Using global settings*.

You must specify the full state name, for example "`SAN.SecState.CPUUtilization`", when matching the "`name`" attribute. The CSS property to be customized is `type`.

```
setting."ilog.tgo.model.IltState"[name="SAN.SecState.CPUUtilization"] {
   type: Counter;
}
```

How to modify the gauge graphical representation of a SAN secondary state (using the API)

When a SAN secondary state is graphically represented by a gauge decoration, the configuration of this decoration has to be previously defined. All SAN secondary states have a predefined gauge representation. Refer to SAN states: the SAN state dictionary in the

Business Objects and Data Sources documentation for information on how each secondary state is graphically represented.

Suppose you have a secondary state that is configured with a gauge decoration as follows:

```
IltSettings.SetValue("SAN.SecState.CPUUtilization.Type",
IltDecorationType.Gauge);
```

You can modify this decoration in the following way:

```
IltColorModifier modifier1 = new IltColorModifier.Shade(0.5f);
IltColorModifier modifier2 = new
    IltColorModifier.MultiColor(IltColorModifier.MultiColor.USE_LAST_VALUE);
IltColorModifier modifier = modifier1.compose(modifier2);

IltGaugeMapping mapping = (IltGaugeMapping)
    IltSettings.GetValue("SAN.SecState.CPUUtilization.Gauge");
mapping.setColorModifier(modifier);
```

You can also create a new gauge mapping, and set the new value in the following way:

```
IltGaugeMapping mapping = new IltGaugeMapping(minImg, maxImg);
IltSettings.SetValue("SAN.SecState.CPUUtilization.Gauge", mapping);
```

How to modify the gauge graphical representation of a SAN secondary state (using CSS)

You can also customize the gauge graphic representation by using global CSS settings. You must specify the full state name, for example "`SAN.SecState.CPUUtilization`", when matching the "`name`" attribute. The CSS property to be customized is `gauge`.

When defining a new gauge mapping class, the following CSS properties are used: `minImage`, `maxImage`, `direction`, `colorModifier`. They match corresponding `set` methods of the `IltGaugeMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltGaugeMapping` for information on how to customize the gauge decorations.

In the example below, the color modifier class named `MyColorModifier` has been created and included in the search path.

```
setting."ilog.tgo.model.IltState"[name="SAN.SecState.CPUUtilization"] {
  gauge: @+gaugeSan;
}
Subobject#gaugeSan {
  class: 'ilog.tgo.graphic.IltGaugeMapping';
  minImage: '@|image("icon1.png")';
  maxImage:'@|image("icon2.png")';
  direction: Bottom;
  colorModifier: @+myColorModifier;
}
Subobject#myColorModifier {
  class: "MyColorModifier";
}
```

How to modify the chart graphical representation of a SAN secondary state (using the API)

When a SAN secondary state is graphically represented by a chart decoration, the configuration of this decoration has to be previously defined. All SAN secondary states have a predefined chart representation. Refer to SAN states: the SAN state dictionary in the *Business Objects and Data Sources* documentation for information on how each secondary state is graphically represented.

Suppose you have a secondary state that is configured with a chart decoration as follows:

```
IltSettings.SetValue("SAN.SecState.CPUUtilization.Type",
IltDecorationType.Chart);
```

You can modify this decoration in the following way:

```
IltColorModifier modifier1 = new IltColorModifier.Shade(0.5f);
IltColorModifier modifier2 =
  new IltColorModifier.MultiColor(IltColorModifier.MultiColor.USE_LAST_VALUE)
;
IltColorModifier modifier = modifier1.compose(modifier2);

IltChartMapping mapping = (IltChartMapping)
  IltSettings.GetValue("SAN.SecState.CPUUtilization.Chart");
mapping.setColorModifier(modifier);
```

You can also create a new chart mapping, and set the new value in the following way:

```
IltChartMapping mapping = new IltChartMapping(minImg, maxImg);
IltSettings.SetValue("SAN.SecState.CPUUtilization.Chart", mapping);
```

How to modify the chart graphical representation of a SAN secondary state (using CSS)

You can also customize the chart graphic representation by using global CSS settings. You must specify the full state name, for example "`SAN.SecState.CPUUtilization`", when matching the "`name`" attribute. The CSS property to be customized is `chart`.

When defining a new chart mapping class, the following CSS properties are used: `minImage`, `maxImage`, `xAxisDirection`, `yAxisDirection`, `colorModifier`. They match corresponding `set` methods of the `IltChartMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltChartMapping` for information on how to customize the chart decorations.

In the example below, the color modifier class named `MyColorModifier` has been created and included in the search path.

```
setting."ilog.tgo.model.IltState"[name="SAN.SecState.CPUUtilization"] {
  chart: @+chartSan;
}
Subobject#chartSan {
  class: 'ilog.tgo.graphic.IltChartMapping';
  minImage: '@|image("icon1.png")';
  maxImage:'@|image("icon2.png")';
  xAxisDirection: Right;
  yAxisDirection: Bottom;
  colorModifier: @+myColorModifier;
}
Subobject#myColorModifier {
  class: "MyColorModifier";
}
```

How to modify the counter graphical representation of a SAN secondary state (using the API)

When a SAN secondary state is graphically represented by a counter decoration, the configuration of this decoration has to be previously defined. All SAN secondary states have a predefined counter representation. Refer to SAN states: the SAN state dictionary in the *Business Objects and Data Sources* documentation for information on how each secondary state is graphically represented.

Suppose you have a secondary state that is configured with a counter decoration as follows:

```
IltSettings.SetValue("SAN.SecState.CPUUtilization.Type",
IltDecorationType.Counter);
```

You can modify this decoration in the following way:

```
IltCounterMapping cm = new IltCounterMapping(new DecimalFormat("#Mbs"),
                                             IltrFont.CounterText,
                                             true, Color.black,
                                             Color.white, Color.black,
                                             3, 1, IlvDirection.Right, false)
;
IltSettings.SetValue("SAN.SecState.CPUUtilization.Counter", cm);
```

How to modify the counter graphical representation of a SAN secondary state (using CSS)

You can also customize the counter graphic representation by using global CSS settings. You must specify the full state name, for example "`SAN.SecState.CPUUtilization`", when matching the "`name`" attribute. The CSS property to be customized is `counter`.

When defining a new counter mapping class, the following CSS properties are used: `format`, `font`, `antialiasing`, `foregroundColor`, `backgroundColor`, `borderColor`, `xPadding`, `yPadding`. They match corresponding `set` methods of the `IltCounterMapping` class (and its super class). Refer to `ilog.tgo.graphic.IltCounterMapping` for information on how to customize the counter decorations.

```
setting."ilog.tgo.model.IltState"[name="SAN.SecState.CPUUtilization"] {
  counter: @+counterSan;
}
Subobject#counterSan {
  class: 'ilog.tgo.graphic.IltCounterMapping';
  format: @+myNumberFormat;
  font: "Helvetica-Bold-12";
  antialiasing: false;
  foregroundColor: red;
  backgroundColor: blue;
  borderColor: red;
  xPadding: 100;
  yPadding: 100;
}
Subobject#myNumberFormat{
  class: 'MyNumberFormat';
}
```

Customizing alarm severities

JViews TGO provides the concept of alarm to represent specific conditions in a managed object.

Alarms are represented graphically using alarm balloon and alarm count decorations. JViews TGO distinguishes raw alarms (generated and carried by an object) from impact alarms (propagated to an object). A telecom object can have raw and impact alarms at the same time, which introduces the concepts of primary and secondary alarm states. The primary alarm state has a more detailed representation than the secondary alarm state. By default, the primary alarm state is the raw alarm state. The choice of the primary alarm state can be customized by the CSS properties listed in the following table.

CSS properties for the primary and secondary alarm states

Property Name	Type	Default Value	Used by `IltObject` in	Description
`primaryAlarmState`	`ilog.tgo.model.IltAlarmStateEnum`	`Raw`	network	Determines whether the primary alarm state is carried by the raw alarms or by the impact alarms. The secondary alarm state is represented as a secondary icon. Possible values are: -`Raw` -`Impact`
`listPrimaryAlarmState`	`Boolean`	`false`	network	Determines whether the object primary alarm state is displayed in the information window.
`listSecondaryAlarmState`	`Boolean`	`true`	network	Determines whether the object secondary alarm state is displayed in the

Property Name	Type	Default Value	Used by `IltObject` in	Description
				information window.
`listAlarmStateAbbreviated`	`Boolean`	`false`	network	Determines whether the alarm state information listed in the information window displays alarm severities using their abbreviation or their description.

For more information, refer to

How to create new alarm severities (using the API)

In the following example, a new raw alarm severity named “Informational” is created with the following properties:

- ♦ A unique name, used to identify the alarm severity in the Alarm system.
- ♦ A short description used by the alarm count.
- ♦ A severity index, used to define the priority of the new alarm severity in the Alarm severity enumeration.

The new alarm severity is positioned before the `Unknown` severity in the `IltAlarm.Severity` enumeration.

```
IltAlarm.Severity infoSeverity =
    new IltAlarm.Severity("Informational",
                          IltAlarm.Severity.Unknown.getSeverity()/2);
```

To create a new impact alarm severity use the `IltAlarm.ImpactSeverity` enumeration instead of `IltAlarm.Severity`.

How to create new alarm severities (using CSS)

You can also create new raw and impact alarm severities by using global CSS settings:

```
Settings {
   alarm: true;
}
Alarm {
   severities[0]: @+severity0;
   impactSeverities[0]: @+impactSeverity0;
}
Subobject#severity0 {
  class: 'ilog.tgo.model.IltAlarm.Severity';
  name: "Informational";
  severity: 210;
}
Subobject#impactSeverity0 {
  class: 'ilog.tgo.model.IltAlarm.ImpactSeverity';
  name: "InformationalLow";
  severity: 220;
}
```

How to customize alarm severities

Alarms are represented in the business objects using the following graphical cues:

- ♦ A color associated with the object base.
- ♦ An alarm count displayed on the object base.
- ♦ A colored alarm balloon displaying another alarm count.
- ♦ A colored outline displayed around the object base.

For information on how these graphical cues are used in JViews TGO, refer to

When a new alarm severity is created, the properties used to customize the graphical cues need to be defined. These properties are:

- ♦ A short description used by the alarm count.
- ♦ An expanded description used in the alarm balloon when the CSS property `alarmBalloonCollapsed` is false.
- ♦ A set of colors used when representing alarm conditions.

These properties are set as part of the JViews TGO look and feel, which can be customized using `SetValue(java.lang.Object, java.lang.Object)` or global CSS settings.

The properties that affect the alarm severities are:

- ♦ `<severity category>.<severity name>.Color`
- ♦ `<severity category>.<severity name>.BrightColor`
- ♦ `<severity category>.<severity name>.DarkColor`
- ♦ `<severity category>.<severity name>.Abbreviation`
- ♦ `<severity category>.<severity name>.Description`
- ♦ `<severity category>.<severity name>.Icon`

For the raw alarm severity created in *How to create new alarm severities (using the API)*, the following properties will have to be defined:

- `Alarm.Raw.Informational.Color`
- `Alarm.Raw.Informational.BrightColor`
- `Alarm.Raw.Informational.DarkColor`
- `Alarm.Raw.Informational.Abbreviation`
- `Alarm.Raw.Informational.Description`
- `Alarm.Raw.Informational.Icon` (This property is optional.)

The following example shows how you should customize the new alarm severity using the API.

```
// Define the colors
Color myAlarmColor = new Color(127, 255, 212);
// aquamarine
Color myAlarmBrightColor = myAlarmColor.brighter();
Color myAlarmDarkColor = myAlarmColor.darker();

IltSettings.SetValue("Alarm.Raw.Informational.Color", myAlarmColor);
IltSettings.SetValue("Alarm.Raw.Informational.BrightColor",
myAlarmBrightColor);
IltSettings.SetValue("Alarm.Raw.Informational.DarkColor", myAlarmDarkColor);
IltSettings.SetValue("Alarm.Raw.Informational.Abbreviation", "i");
IltSettings.SetValue("Alarm.Raw.Informational.Description", "Informational");
```

You can then retrieve these values using `GetValue(java.lang.Object).`

The following example shows how to customize the new alarm severity using global CSS settings (you must specify the full alarm name, for example "`Alarm.Raw.Informational`" or "`Alarm.Impact.InformationalLow`" when matching the "`name`" attribute). The CSS properties to be customized are `color`, `darkColor`, `brightColor`, `abbreviation`, `description`, `icon`.

```
setting."ilog.tgo.model.IltAlarm.Severity"[name="Alarm.Raw.Informational"] {
  color: orange;
  darkColor: blue;
  brightColor: yellow;
  abbreviation: "i";
  description: "Informational";
  icon: '@|image("icon1.png")';
}
setting."ilog.tgo.model.IltAlarm.ImpactSeverity"[name="Alarm.Impact.Information
alLow"] {
  color: orange;
  darkColor: blue;
  brightColor: yellow;
  abbreviation: "IL";
  description: "InformationalLow";
  icon: '@|image("icon1.png")';
}
```

How to customize existing alarm severities

You can customize the representation of the existing alarm severities, for example, `Raw.Critical` or `Impact.CriticalHigh`. To do so, you use the same properties as listed in *How to customize alarm severities*:

- `<severity category>.<severity name>.Color`
- `<severity category>.<severity name>.BrightColor`
- `<severity category>.<severity name>.DarkColor`
- `<severity category>.<severity name>.Abbreviation`
- `<severity category>.<severity name>.Description`
- `<severity category>.<severity name>.Icon` (This property is optional.)

The following example shows how to customize the severity `Impact.CriticalHigh` using the API.

```
// Define the colors
Color myAlarmColor = Color.magenta;
Color myAlarmBrightColor = myAlarmColor.brighter();
Color myAlarmDarkColor = myAlarmColor.darker();

IltSettings.SetValue("Alarm.Impact.CriticalHigh.Color", myAlarmColor);
IltSettings.SetValue("Alarm.Impact.CriticalHigh.BrightColor",
myAlarmBrightColor);
IltSettings.SetValue("Alarm.Impact.CriticalHigh.DarkColor", myAlarmDarkColor)
;
IltSettings.SetValue("Alarm.Impact.CriticalHigh.Abbreviation", "CH");
IltSettings.SetValue("Alarm.Impact.CriticalHigh.Description", "Critical High")
;
```

The following example shows how to customize the severity `Impact.CriticalHigh` using global CSS settings (you must specify the full alarm name, for example "`Alarm.Raw.Critical`" or "`Alarm.Impact.CriticalHigh`" when matching the "`name`" attribute. The CSS properties to be customized are `color`, `darkColor`, `brightColor`, `abbreviation`, `description`, `icon`):

```
setting."ilog.tgo.model.IltAlarm.Severity"[name="Alarm.Raw.Critical"] {
  color: orange;
  darkColor: blue;
  brightColor: yellow;
  abbreviation: "c";
  description: "Critical";
  icon: '@|image("icon1.png")';
}
setting."ilog.tgo.model.IltAlarm.ImpactSeverity"[name="Alarm.Impact.CriticalHig
h"] {
  color: orange;
  darkColor: blue;
  brightColor: yellow;
  abbreviation: "CH";
  description: "CriticalHigh";
  icon: '@|image("icon1.png")';
}
```

How to customize the Loss Of Connectivity representation

You can define the following settings to customize the representation of objects that have the state Loss of Connectivity set.

- `Alarm.LossOfConnectivity.Color`
- `Alarm.LossOfConnectivity.BrighColor`
- `Alarm.LossOfConnectivity.DarkColor`
- `Alarm.LossOfConnectivity.Abbreviation`

The following code extract shows you how to proceed using the API:

```
IltSettings.SetValue("Alarm.LossOfConnectivity.Color", Color.blue);
IltSettings.SetValue("Alarm.LossOfConnectivity.Abbreviation", "??");
```

The following example shows how to proceed using global CSS settings (you must specify the full alarm loss of connectivity name, for example "`Alarm.LossOfConnectivity`" when matching the "`name`" attribute. The CSS properties to be customized are `color`, `darkColor`, `brightColor`, `abbreviation`):

```
setting."ilog.tgo.model.IltState"[name="Alarm.LossOfConnectivity"] {
  color: orange;
  darkColor: green;
  brightColor: yellow;
  abbreviation: "LOC1";
}
```

How to customize the Not Reporting representation

By default, objects in the Not Reporting state do not show any alarm information. They display the value "`NR`" instead of the current alarm count.

You can customize this representation using the setting `Alarm.NotReporting.Abbreviation`.

The following code extract shows you how to proceed using the API:

```
IltSettings.SetValue("Alarm.NotReporting.Abbreviation", "NR?");
```

The following example shows how to proceed using global CSS settings (you must specify the full alarm Not Reporting name, "`Alarm.NotReporting`" when matching the "`name`" attribute. The CSS property to be customized is `abbreviation`):

```
setting."ilog.tgo.model.IltState"[name="Alarm.NotReporting"] {
  abbreviation: "NR1";
}
```

Customizing alarm count attributes

Representing alarm counts

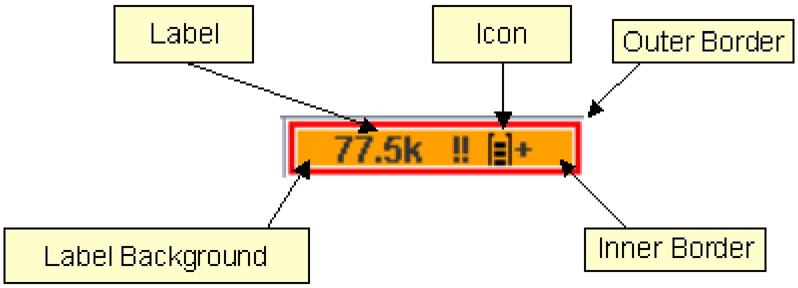
The alarm count attributes of `IltObject` are represented in the table component using:

- ♦ `IltDefaultAlarmCountGraphic` for raw alarms and traps
- ♦ `IltImpactAlarmCountGraphic` for impact alarms.

The following table interprets some graphical representations of alarm counts.

Examples of alarm count representations

Representation	Description
	99500 (+/- 0.5%) outstanding alarms of severity Minor, some outstanding alarms of lower severity, and some new alarms of severity Warning
	4 new alarms of severity Warning
	1 Critical acknowledged alarm
	5 new impact alarms of severity Critical High, some outstanding alarms of lower severity
	77500 (+/- 0.5%) outstanding alarms of severity Major High, some outstanding alarms of lower severity, and some new alarms of severity Minor High



Alarm count representation details

Customizing alarm count representations

The alarm count representation can be customized using the CSS properties listed in the following table.

CSS properties for alarm count representations

Property Name	Type	Default Value	Description
`innerBorderColor`	`Color`	`null`	Color of the inner border.
`innerBorderVisible`	`Boolean`	`false`	Use `true` to show the inner border. The inner border should be displayed when the alarm count has both new and acknowledged alarms.
`innerBorderWidth`	`int`	`1`	Width of the inner border.
`label`	`String`	`null`	Label used to display the alarm count. This text is parsed and laid out to conform to the format.
`labelBackgroundColor`	`Color`	`null`	Color of the label background. The color corresponds to the severity of the most severe new alarm.
`outerBorderColor`	`Color`	`null`	Color of the outer border. The color corresponds to the severity of the most severe outstanding alarm.
`outerBorderWidth`	`int`	`2`	Width of the outer border.
`icon`	`Image`	`null`	Icon used to compose the alarm count.
`iconVisible`	`Boolean`	`true`	Determines whether the icon is visible or not.
`iconPosition`	`int`	`IlvConstants.TRAILING`	Position of the icon relative to the label. If the label follows the alarm count pattern, the icon is placed before or after the alarm severity text. Possible values: `IlvConstants.LEADING` or `IlvConstants.TRAILING`

The CSS functions listed in the following table are useful to provide values for the alarm count representation.

CSS functions for alarm count attributes

Function Name	Description	Usage
`alarmCount`	The outstanding alarm count label	**Parameters:** Alarm type or severity. The alarm type can have one of the following values: `Default` for raw alarms or traps `Impact` for impact alarms Specifying no parameter is equivalent to `Default`.

Function Name	Description	Usage

		The severity is the String representation of a severity or an `IltAlarmSeverity`.
		Examples:
		`@|alarmCount();`
		`@|alarmCount("Default");`
		`@|alarmCount("Impact");`
		`@|alarmCount("Raw.Major");`
		`@|alarmCount("Impact.CriticalHigh");`

`newAlarmCount`	The new alarm count label	See `alarmCount`
`acknowledgedAlarmCount`	The acknowledged alarm count label	See `alarmCount`
`highestSeverity`	The severity of the most severe outstanding alarm	See `alarmCount`
`highestNewSeverity`	The severity of the most severe new alarm	See `alarmCount`
`highestAcknowledgedSeverity`	The severity of the most severe acknowledged alarm	See `alarmCount`
`alarmSummary`	The total amount of new and outstanding alarms	See `alarmCount`
`newAlarmSummary`	The total amount of new alarms	See `alarmCount`
`acknowledgedAlarmSummary`	The total amount of acknowledged alarms	See `alarmCount`

`severityColor`	The color corresponding to a severity	**Parameters:**
		`IltAlarmSeverity` or the String representation of an `IltAlarmSeverity`.
		Examples:
		`@|severityColor("Raw.Major");`
		`@|severityColor(@|highestSeverity());`

Function Name	Description	Usage
		`@|severityColor (@|highestSeverity("Impact") );`
`severityIcon`	The icon corresponding to a severity	**Parameters:** `IltAlarmSeverity` or the String representation of an `IltAlarmSeverity`. **Examples:** `@|severityIcon("Raw.Major");` `@|severityIcon (@|highestSeverity());` `@|severityIcon (@|highestSeverity("Impact") );`

How to reproduce the JViews TGO 4.0 styling for alarm count attributes

The following style sheet extract reproduces the JViews TGO 4.0 styling for alarm count attributes.

```
// IltObject alarmCount attribute
object."ilog.tgo.model.IltObject/alarmCount" {
  class: ilog.tgo.graphic.IltDefaultAlarmCountGraphic;
  label: '@|alarmCount()';
  toolTipText: '@|alarmSummary()';
  labelBackgroundColor: '@|severityColor(@|highestNewSeverity())';
  outerBorderColor: '@|severityColor(@|highestSeverity())';
  innerBorderColor: white;
  innerBorderVisible: '@|highestNewSeverity()!=null &&
@|highestAcknowledgedSeverity()!=null';
}

// IltObject newAlarmCount attribute
object."ilog.tgo.model.IltObject/newAlarmCount" {
  class: ilog.tgo.graphic.IltDefaultAlarmCountGraphic;
  label: '@|newAlarmCount()';
  toolTipText: '@|newAlarmSummary()';
  labelBackgroundColor: '@|severityColor(@|highestNewSeverity())';
  outerBorderColor: '@|severityColor(@|highestNewSeverity())';
  innerBorderColor: white;
  innerBorderVisible: false;
}

// IltObject impactAlarmCount attribute
```

```
object."ilog.tgo.model.IltObject/impactAlarmCount" {
  class: ilog.tgo.graphic.IltImpactAlarmCountGraphic;
  label: '@|alarmCount("Impact")';
  toolTipText: '@|alarmSummary("Impact")';
  labelBackgroundColor: '@|severityColor(@|highestNewSeverity("Impact"))';
  outerBorderColor: '@|severityColor(@|highestSeverity("Impact"))';
  innerBorderColor: white;
  innerBorderVisible: '@|highestNewSeverity("Impact")!=null &&
@|highestAcknowledgedSeverity("Impact")!=null';
}

// IltObject newImpactAlarmCount attribute
object."ilog.tgo.model.IltObject/newImpactAlarmCount" {
  class: ilog.tgo.graphic.IltImpactAlarmCountGraphic;
  label: '@|newAlarmCount("Impact")';
  toolTipText: '@|alarmSummary("Impact")';
  labelBackgroundColor: '@|severityColor(@|highestNewSeverity("Impact"))';
  outerBorderColor: '@|severityColor(@|highestNewSeverity("Impact"))';
  innerBorderColor: white;
  innerBorderVisible: false;
}
```

How to reproduce the JViews TGO 3.5 styling for alarm count attributes

The following style sheet extract reproduces the styling obtained in JViews TGO 3.5 for the `IltObject` alarm count attributes. You may want to use it to improve performance or to ensure compatibility with previous versions of JViews TGO.

```

// IltObject alarmCount attribute
object."ilog.tgo.model.IltObject/alarmCount" {
  class: javax.swing.JLabel;
  text: @|alarmCount();
  toolTipText: @|alarmCount();
  labelFont: sansserif-plain-12;
  labelBackground: '@|severityColor(@|highestSeverity())';
}

// IltObject newAlarmCount attribute
object."ilog.tgo.model.IltObject/newAlarmCount" {
  class: javax.swing.JLabel;
  text: @|alarmCount();
  toolTipText: @|alarmCount();
  labelFont: sansserif-plain-12;
  labelBackground: '@|severityColor(@|highestNewSeverity())';
}

// IltObject impactAlarmCount attribute
object."ilog.tgo.model.IltObject/impactAlarmCount" {
  class: javax.swing.JLabel;
  text: @|alarmCount("Impact");
  toolTipText: @|alarmCount("Impact");
  labelFont: sansserif-plain-12;
  labelBackground: '@|severityColor(@|highestSeverity("Impact"))';
}

// IltObject newImpactAlarmCount attribute
object."ilog.tgo.model.IltObject/newImpactAlarmCount" {
  class: javax.swing.JLabel;
  text: @|newAlarmCount("Impact");
  toolTipText: @|newAlarmCount("Impact");
  labelFont: sansserif-plain-12;
  labelBackground: '@|severityColor(@|highestNewSeverity("Impact"))';
}
```

How to display the alarm count for a specific alarm severity

Provided there is a `MyObject` business class, with super-class `IltObject` and a `rawMajorAlarmCount` attribute, the following style sheet produces an alarm count representation specific to raw alarms of severity `Major`.

```

// IltObject alarmCount attribute
object."MyObject/rawMajorAlarmCount" {
  class: ilog.tgo.graphic.IltDefaultAlarmCountGraphic;
  label: '@|newAlarmCount("Raw.Major")';
  toolTipText: '@|newAlarmSummary("Raw.Major")';
  labelBackgroundColor: '@|severityColor(@|highestNewSeverity("Raw.Major"))';

  outerBorderColor: '@|severityColor(@|highestNewSeverity("Raw.Major"))';
  innerBorderColor: white;
  innerBorderVisible: false;
}
```

Customizing trap types

JViews TGO provides a concept of alarm based on RFC 1157 - A Simple Network Management Protocol (SNMP) called a *trap*. A trap represents something unusual that occurs in an object. Traps, like alarms, are represented graphically using the alarm balloon and alarm count decorations. For more information, refer to

How to create new trap types (using the API)

In the following example, a new trap named “Alert”, based on RFC 2455 C Definitions of Managed Objects for APPN, is created with the following properties:

♦ A unique name, used to identify the trap type in the Trap alarm system.

♦ A severity, used to define the priority of the new trap type in the Trap type enumeration.

The new trap type is positioned before the `ColdStart` type in the `IltTrap.Type` enumeration.

```
IltTrap.Type alertType = new IltTrap.Type("Alert",
IltTrap.Type.ColdStart.getSeverity() / 2);
```

How to create new trap types (using CSS)

You can also create new trap types by using global CSS settings.

```
Settings {
   alarm: true;
}
Alarm{
   traps[0]: @+trap0;
}
Subobject#trap0 {
  class: 'ilog.tgo.model.IltTrap.Type';
  name: "Alert";
  severity: 230;
}
```

How to customize trap types

Traps, like alarms, are represented in the business objects using the following graphical cues:

♦ A color associated with the object base.

♦ An alarm count displayed on the object base.

♦ A colored alarm balloon displaying another alarm count.

♦ A colored outline displayed around the object base.

For information on how these graphical cues are used in JViews TGO, refer to

When a new trap type is created, the properties used to customize the graphical cues need to be defined. These properties are:

- ♦ A short description used by the alarm count.
- ♦ An expanded description used in the alarm balloon when the CSS property `alarmBalloonCollapsed` is false.
- ♦ A set of colors to be defined subsequently.

These properties are set as part of the JViews TGO look and feel, which can be customized using `SetValue(java.lang.Object, java.lang.Object)` or global CSS settings.

The properties that affect the trap types are:

- ♦ `Trap.Type.<type name>.Color`
- ♦ `Trap.Type.<type name>.BrightColor`
- ♦ `Trap.Type.<type name>.DarkColor`
- ♦ `Trap.Type.<type name>.Abbreviation`
- ♦ `Trap.Type.<type name>.Description`
- ♦ `Trap.Type.<type name>.Icon`

For the trap type created in *How to create new trap types (using the API)*, the following properties will have to be defined:

- ♦ `Trap.Type.Alert.Color`
- ♦ `Trap.Type.Alert.BrightColor`
- ♦ `Trap.Type.Alert.DarkColor`
- ♦ `Trap.Type.Alert.Abbreviation`
- ♦ `Trap.Type.Alert.Description`
- ♦ `Trap.Type.Alert.Icon` (This property is optional.)

The following example shows how to customize the new trap type using the API.

```
// Define the colors
Color myAlarmColor = new Color(127, 255, 212);
// aquamarine
Color myAlarmBrightColor = myAlarmColor.brighter();
Color myAlarmDarkColor = myAlarmColor.darker();

IltSettings.SetValue("Trap.Type.Alert.Color", myAlarmColor);
IltSettings.SetValue("Trap.Type.Alert.BrightColor", myAlarmBrightColor);
IltSettings.SetValue("Trap.Type.Alert.DarkColor", myAlarmDarkColor);
IltSettings.SetValue("Trap.Type.Alert.Abbreviation", "i");
IltSettings.SetValue("Trap.Type.Alert.Description", "Informational");
```

You can then retrieve these values using `GetValue(java.lang.Object)`.

The following example shows how to customize the new trap type using global CSS settings. You must specify the full trap type name, for example "`Trap.Type.Alert`" when matching the "`name`" attribute. The CSS properties to be customized are `color`, `darkColor`, `brightColor`, `abbreviation`, `description`, `icon`):

```
setting."ilog.tgo.model.IltTrap.Type"[name="Trap.Type.Alert"] {
  color: orange;
  darkColor: green;
  brightColor: yellow;
  abbreviation: "a";
  description: "Alert";
  icon: '@|image("icon1.png")';
}
```

How to customize existing trap types

You can customize the representation of the existing trap types, for example, `Trap.Type.LinkFailure` or `Trap.Type.ColdStart`.

To do so, you use the same properties as listed in *How to customize trap types*:

- `Trap.Type.<type name>.Color`
- `Trap.Type.<type name>.BrightColor`
- `Trap.Type.<type name>.DarkColor`
- `Trap.Type.<type name>.Abbreviation`
- `Trap.Type.<type name>.Description`
- `Trap.Type.<type name>.Icon` (This property is optional.)

The following example shows how to customize the trap type `Trap.Type.ColdStart` using the API.

```
// Define the colors
Color myAlarmColor = Color.magenta;
Color myAlarmBrightColor = myAlarmColor.brighter();
Color myAlarmDarkColor = myAlarmColor.darker();

IltSettings.SetValue("Trap.Type.ColdStart.Color", myAlarmColor);
IltSettings.SetValue("Trap.Type.ColdStart.BrightColor", myAlarmBrightColor);
IltSettings.SetValue("Trap.Type.ColdStart.DarkColor", myAlarmDarkColor);
IltSettings.SetValue("Trap.Type.ColdStart.Abbreviation", "CH");
IltSettings.SetValue("Trap.Type.ColdStart.Description", "Critical High");
```

The following example shows how to customize the trap type Trap.Type.ColdStart using global CSS settings (you must specify the full trap type name, "Trap.Type.ColdStart", when matching the "name" attribute. The CSS properties to be customized are color, darkColor, brightColor, abbreviation, description, icon):

```
setting."ilog.tgo.model.IltTrap.Type"[name="Trap.Type.ColdStart"] {
  color: yellow;
  darkColor: blue;
  brightColor: green;
  abbreviation: CS;
  description: "ColdStart";
  icon: '@|image("icon1.png")';
}
```

Customizing the secondary state icons

Secondary states are normally represented by icons, gauges, charts, or counters, which are displayed in a row (called stacker) on the top or at the bottom of an `IltObject`. By default, the position of the secondary states is defined by the insertion order in the object state. Depending on your application, you may be interested in defining specific positions for some states. This can be accomplished by configuring the secondary state positioner.

The position of the icons in the stacker is defined globally inside the static instance of `IltSecondaryStatePositioner`.

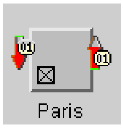
The following fragment of code illustrates how you should proceed to force the `Degraded` to be constantly in the first position of the stacker.

```
IltSecondaryStatePositioner positioner =
            IltSecondaryStatePositioner.GetInstance();
positioner.setPosition (IltOSI.Availability.Degraded, 0);
```

If you have decided to remove this constraint, you just need to set the new position to -1, to indicate that it will no longer be handled by the `IltSecondaryStatePositioner`.

```
positioner.setPosition (IltOSI.Availability.Degraded, -1);
```

Secondary states can also be located in two specific positions, called `InState` and `OutState`. By default, the SNMP states `Interface.InOctets` and `Interface.OutOctets` are displayed in these positions, as illustrated in *Positions of SNMP states `Interface.InOctets` and `Interface.OutOctets`*.



Positions of SNMP states `Interface.InOctets` and `Interface.OutOctets`

To place a state in the `InState` or `OutState` position, use the methods `setInState(ilog.tgo.graphic.IltDecorationSource)` or `setOutState(ilog.tgo.graphic.IltDecorationSource)`, as illustrated in the code extracts below:

```
positioner.setInState (IltSNMP.Interface.InDiscards);
```

or

```
positioner.setOutState (IltSNMP.Interface.OutDiscards);
```

Index

A

acknowledgedAlarmCount
    CSS function **28**
acknowledgedAlarmSummary
    CSS function **28**
alarmBalloonTextFont
    CSS property **100**
alarmCount
    CSS function **28**
alarmCountFont
    CSS property **100**
alarms
    customizing **379**
alarmSummary
    CSS function **28**
alternateColor
    CSS property **196**
animateSpeed
    CSS property **196**
animation
    general link **208**
arrow
    general link **206**
arrowColor
    CSS property **196**
arrowMode
    CSS property **196**
arrowPosition
    CSS property **196**
arrowRatio
    CSS property **196**
attribute matching
    CSS **38**
attribute value
    CSS input model **21**
attribute value change
    CSS input model **21**
attributes
    properties **44**

B

background
    CSS property **14**
background color
    setting in tree component **12**
background property **89**
Bezier curve **203**
blinkingcolor
    CSS function **28**
border
    adding in general link **201**
    of general node shape **191**
borderColor
    CSS property **196**
borderColor2
    CSS property **196**
borderLineStyle
    CSS property **196**
borderLineStylePhase
    CSS property **196**
borderWidth
    CSS property **196**
BTS antenna tiny type
    customizing **298**
business classes
    predefined **42**
    user-defined **42**

C

captionLabel
    CSS property **126, 127**
captionLabelVisible
    CSS property **126, 127**
cascading
    sources of style **20**

style sheets **38**
cascading mechanism **101**
changes
in CSS properties **18**
`class`
CSS **127**
CSS input model **21**
CSS property **14, 23**
values of CSS property **23**
class name
value of CSS `class` property **23**
color
alternate in general link **205**
conversion
to `String` in CSS **38**
CSS
attribute matching **38**
attribute properties **44**
attribute property inheritance **44**
attribute value changing property value **21**
attribute value matching selector attribute **21**
class **21, 127**
conversion to `String` **38**
declaration **15, 17, 18, 20, 21, 23, 24, 26**
declaration inheritance **20**
dynamic behavior **21**
dynamic interpretation **12**
empty string **39**
engine **21**
graphic object **21**
hash sign (`#`) **15**
ID pattern matching **15**
Java objects **21**
matching attribute values **15**
matching object identifiers **15**
model indirection **24**
null value **39**
object ID **21**
object identifier **125**
object type **21**
predefined business classes **42**
properties for System window **126**
property identifier syntax error **18**
pseudoclass **128**
pseudoclasses **38**
pseudoelements **38**
resolving URLs **24**
rule **12, 15, 17, 18, 20, 128**
rule priority **17, 20, 128**
selector **15, 17, 20, 100, 125, 131, 132**
serialized JavaBean **23**
serialized object **23**
specificity **17, 20, 38, 128**
subobject **25**
SVG and XML **10**
syntax enhancement **39**
syntax errors **18**
syntax for matching attributes **15**
tree structure of objects **21**
use of quotes **39**
user-defined business classes **42**
W3C **10**
wildcards **84, 91**
XML **10, 21**
CSS functions
`acknowledgedAlarmCount` **28**
`acknowledgedAlarmSummary` **28**
`alarmCount` **28**
`alarmSummary` **28**
`blinkingcolor` **28**
`format` **28**
`highestAcknowledgedSeverity` **28**
`highestNewSeverity` **28**
`highestSeverity` **28**
`image` **28**
`newAlarmCount` **28**
`newAlarmSummary` **28**
`pattern` **28**
`primaryStateSummary` **28**
`resource` **28**
`secondaryStateSummary` **28**
`settings` **28**
`severityBrightColor` **28**
`severityColor` **28**
`severityDarkColor` **28**
`severityIcon` **28**
`valuemap` **28**
CSS properties
`alarmBalloonTextFont` **100**
`alarmCountFont` **100**
`alternateColor` **196**
`animateSpeed` **196**
`arrowColor` **196**
`arrowMode` **196**
`arrowPosition` **196**
`arrowRatio` **196**
`background` **14, 89**
`borderColor` **196**
`borderColor2` **196**
`borderLineStyle` **196**
`borderLineStylePhase` **196**
`borderWidth` **196**
`CAP_SQUARE` **203**
`captionLabel` **126, 127**
`captionLabelVisible` **126, 127**
changes **18**
`class` **14, 23**
`curved` **196**

endCap **196**
expansion **70**
fillAngle **184**
fillEnd **184**
fillStart **184**
fillStyle **184, 189**
fillTexture **184**
focusBorderColor **70, 81**
focusBorderWidth **70, 81**
foreground **14, 89, 123, 184**
functionList **36**
functions **36**
horizontalAlignment **81, 89**
horizontalAutoResizeMargin **184**
horizontalAutoResizeMode **184, 194**
icon **70, 81, 89, 125, 184**
iconPosition **184**
iconVisible **70, 81, 89, 125**
in graphic components **12**
index **89**
infoWindowTextFont **100**
internalZoom **196**
keepingAspectRatio **184**
label **70, 81, 89, 98, 126, 127**
labelBackground **70, 81, 98, 128**
labelFont **70, 81, 89, 98**
labelForeground **70, 81, 98**
labelInsets **81, 89**
labelPosition **70, 81, 89**
labelSpacing **70, 81, 89**
labelVisible **70, 81, 89, 98**
lineJoin **196**
lineStyle **196**
lineStylePhase **196**
lineWidth **196**
linkPorts **133**
minLineWidth **196**
mode **196**
oriented **196**
overlapIcon **70**
overlapIconVisible **70**
predefined **10**
preferredWidth **89**
qualityLevel **196**
selectionFocusMode **70**
shapeAspectRatio **184**
shapeHeight **184**
shapeType **184, 187**
shapeWidth **184**
sortingMode **89**
sortingPriority **89**
styleSheetDebugMask **18**
syntax error in identifier **18**
tableColumnOrder **91**
tableRowHeight **92**
text **14**
toolTipBackground **102**
toolTipFont **102**
toolTipForeground **102**
toolTipGraphic **70, 81, 102**
toolTipText **25, 70, 81, 89, 102**
tree nodes **46**
useDefaultCellRenderer **88**
verticalAlignment **81, 89**
verticalAutoResizeMargin **184**
verticalAutoResizeMode **184**
visible **89**
visibleInSystemWindow **126, 127**
wave **196**

CSS settings
    global **52**
CSS2 Recommendation **10, 24, 38**
curved
    CSS property **196**
curves
    general link **203**
customizing
    alarm severities **379**
    BTS tiny types **298**
    link bundle tiny types **234**
    link set tiny types **234**
    link tiny types **230**
    network element families **174**
    network element functions **174**
    network element types **147**
    network element types using images **157**

D

dashes
    general link **204**
declaration
    CSS **15, 17, 18, 20, 21, 23, 24, 26**
declarations
    CSS **14**
dynamic behavior
    CSS **21**

E

empty string
    in CSS **39**
endCap
    CSS property **196**
engine
    CSS **21**
enhancing syntax
    in CSS **39**
expansion property **70**

F

fillAngle

    CSS property **184**
fillEnd
    CSS property **184**
fillStart
    CSS property **184**
fillStyle
    CSS property **184**
fillStyle property **189**
fillTexture
    CSS property **184**
focusBorderColor property **70, 81**
focusBorderWidth property **70, 81**
foreground
    CSS property **14, 123, 184**
foreground property **89**
format
    CSS function **28**
functionList
    CSS property **36**
functions
    CSS property **36**

G

general link
    adding border **201**
    alternate colors **205**
    animation **208**
    arrows **206**
    curves **203**
    dashes **204**
    modes **200**
    other effects **209**
    stroke **203**
    wave effect **208**
    zoom **209**
general node
    automatic resizing **194**
    controlling icon **192**
    shape paint styles **189**
    shape stroke **191**
    shape type **187**
getBrightPalette method **167**
getDarkPalette method **167**
getPalette method **167**
getPreferredSize method **166**
global CSS settings **52**
graphic components
    CSS properties **12**
graphic objects
    in CSS **21**
groups
    positioning information cluster **250**

H

hash sign (#)
    CSS **15**
highestAcknowledgedSeverity
    CSS function **28**
highestNewSeverity
    CSS function **28**
highestSeverity
    CSS function **28**
horizontalAlignment property **81, 89**
horizontalAutoResizeMargin
    CSS property **184**
horizontalAutoResizeMode
    CSS property **184**
horizontalAutoResizeMode property **194**

I

icon
    CSS property **184**
    general node **192**
icon property **70, 81, 89, 125**
iconPosition
    CSS property **184**
iconVisible property **70, 81, 89, 125**
ID pattern matching
    CSS **15**
IlpCSSFunction interface **36**
IlpGraphicRendererContext class **45**
IlpPolyline class **236**
IlpRect class **236**
IlpReferenceAttribute class **131**
IlpSimpleDateFormat class **25**
IltBaseStyle class **166**
IltLinearGroup class **236**
IltNetworkElement class **167**
IltNetworkElement.Family class **176**
IltNetworkElement.Function class **174**
IltNetworkElement.Type class **166**
IltPolygon class **236**
IltPolyGroup class **236**
IltRectGroup class **236**
IlvGeneralLink class
    ARROW_DECORATION mode **206**
    ARROW_FILL mode **206**
    ARROW_GRADIENT mode **206**
    ARROW_OPEN mode **206**
    MODE_GRADIENT mode **209**
    MODE_NEON mode **209**
    TEXTURE mode **200**
IlvGraphic class **21**
IlvStroke class
    CAP_SQUARE property **203**
    JOIN_MITER property **203**
IlvStylable interface **18**
image
    CSS function **28**
importing style sheets

CSS **101**
index property **89**
infoWindowTextFont
CSS property **100**
inheritance
attribute properties in CSS **44**
CSS declarations **20**
in style sheets **42**
internalZoom
CSS property **196**

J

Java objects
styled by CSS **21**
JButton
instances in CSS **14**
JComponent
instances in CSS **21**

K

keepingAspectRatio
CSS property **184**

L

label
CSS property **98, 126, 127**
label property **70, 81, 89, 98**
labelBackground
CSS property **128**
labelBackground property **70, 81, 98**
labelFont
CSS property **100**
labelFont property **70, 81, 89, 98**
labelForeground property **70, 81, 98**
labelInsets property **81, 89**
labelPosition property **70, 81, 89**
labelSpacing property **70, 81, 89**
labelVisible property **70, 81, 89, 98**
LINEAR_GRADIENT mode **189**
lineJoin
CSS property **196**
lineStyle
CSS property **196**
lineStylePhase
CSS property **196**
lineWidth
CSS property **196**
link bundle tiny type
customizing **234**
link set tiny type
customizing **234**
link tiny types
customizing **230**
linkPorts
CSS property **133**
links
customizing **217**

M

matching attribute values
CSS **15**
matching object identifiers
CSS **15**
minLineWidth
CSS property **196**
mode
CSS property **196**
model indirection
CSS **24**
modes
general link **200**

N

network elements
customizing families **176**
customizing functions **174**
customizing types **147**
customizing types using images **157**
graphical representations **144**
partial **144**
shortcuts **145**
newAlarmCount
CSS function **28**
newAlarmSummary
CSS function **28**
null value
in CSS **39**

O

object ID
CSS input model **21**
object identifier
CSS **125**
object type
CSS input model **21**
oriented
CSS property **196**
other effects
general link **209**
overlapIcon
CSS property **70**
overlapIconVisible
CSS property **70**

P

palettes
standard **167**
pathname
value of CSS class property **23**
pattern
CSS function **28**
predefined business classes
CSS **42**

predefined business objects
  subnetworks **254**
preferredWidth property **89**
primaryStateSummary
  CSS function **28**
priority
  CSS rules **17, 20, 128**
pseudoclass
  CSS **128**
pseudoclasses
  in CSS **38**
pseudoelements
  in CSS **38**

Q

qualityLevel
  CSS property **196**
quotes
  in CSS **39**

R

RADIAL_GRADIENT mode **189**
resolving URLs
  CSS **24**
resource
  CSS function **28**
rule
  CSS **12, 15, 17, 18, 20, 128**

S

secondaryStateSummary
  CSS function **28**
selectionFocusMode property **70**
selector
  CSS **15, 17, 20**
selectors
  CSS **100, 125, 131, 132**
serialized JavaBean
  CSS **23**
serialized object
  CSS **23**
settings
  CSS function **28**
severityBrightColor
  CSS function **28**
severityColor
  CSS function **28**
severityDarkColor
  CSS function **28**
severityIcon
  CSS function **28**
shape paint style
  of general node **189**
shape stroke
  of general node **191**
shape type
  of general node **187**
shapeAspectRatio
  CSS property **184**
shapeHeight
  CSS property **184**
shapeType
  CSS property **184**
shapeType property **187**
shapeWidth
  CSS property **184**
SNMP
  extending **345**
SOLID_COLOR mode **189**
SONET
  extending **355**
  Main State **356**
sortingMode property **89**
sortingPriority property **89**
specificity
  CSS **17, 20, 38, 128**
stroke
  general link **203**
style sheets **10, 12, 15, 24, 38, 42**
  cascading **38**
  inheritance mechanism **42**
styleSheetDebugMask
  CSS property **18**
styling sources
  cascading **20**
subnetworks **254**
subobject **25**
SVG
  XML in CSS **10**
syntax errors
  CSS **18**
syntax for matching attributes
  CSS **15**
System window
  properties for adding attributes **126**

T

table component
  column header and cell rendering **89**
tableColumnOrder property **91**
tableRowHeight property **92**
text
  CSS property **14**
TEXTURE mode **189**
toolTipBackground property **102**
toolTipFont property **102**
toolTipForeground property **102**
toolTipGraphic property **70, 81**
toolTipGraphicproperty **102**
toolTipText
  CSS property **25**
toolTipText property **70, 81, 89, 102**

trap
    types **392**
tree component
    setting background color **12**
tree nodes
    properties **46**
tree structure of objects
    CSS input model **21**

U

`useDefaultCellRenderer` property **88**
user-defined business classes
    CSS **42**

V

`valuemap`
    CSS function **28**
values
    of CSS `class` property **23**
`verticalAlignment` property **81, 89**
`verticalAutoResizeMargin`
    CSS property **184**
`verticalAutoResizeMode` **194**
    CSS property **184**
`visible` property **89**
`visibleInSystemWindow`
    CSS property **126, 127**

W

W3C
    CSS **10**
`wave`
    CSS property **196**
    in general link **208**
wildcards **84, 91**

X

X Window System
    `.Xdefault` **10**
`Xdefault`
    X Window System **10**
XML
    in CSS **10, 21**

Z

zoom
    general link **209**