Tatami Programmatic Documentation

Henri Darmet, Vianney Grassaud, Ronan Dunklau

December 2008

This document explains how to use in a GWT application the Tatami components. Each component is presented in details, including its APIs. The implementation details of those components are outside the scope of this document, except when they are relevant from a user view point. You can also take an eye on the source code of the TatamiDemo-1.1 project. In this project, most of the Tatami components are used and expose their main features.

You should also take a look to the Tatami TestPages project. This project is used to automatically test Tatami using HTMLUnit. Newer widgets are demonstrated in this project.

You also may take a look to the Tatami Player project, which uses some Tatami Widgets to demonstrate how easy it can be to create a simple application.

Contents

Chapter 1

Installation

1.1 Simple Installation

Tatami is released as a single Jar (tatami.jar) which contains all the resources necessary: code Java, files of configurations, files CSS, images and JavaScript code. To use Tatami, it is enough to add this jar in the classpath of your application and to insert in the file of configuration of your module (.gwt.xml) the following line:

```
<inherits name="com.objetdirect.tatami.Tatami"/>
```

You do not have to choose import the dojo stylesheets, nor add the corresponding class in your markup: the theme package takes care of that for yourself!

1.1.1 Custom Installation

The Simple Installation (Tatami module) contains all the scripts used for Tatami, including the base Dojo release and Yui. This has two side-effects : If you just use Dojo, or YUI, you will have to download both. If you use Dojo, you will use the base release, loading all the required dojo modules asynchronously as you need it.

That is why Tatami offers the possibility to import only what you want, and use an optimized build of dojo if you need it. You can do so importing smaller Tatami modules than Tatami itself. These modules are:

Tatami_Base: all others modules inherit this one, so you do not have to include it

Tatami_YUI: only loads YUI. Use this only if you use YUI widgets (such as Tatami DragAndDropPanel)

Tatami_DojoBase: only loads dojo core, and then load all other modules asynchronously. This is the recommended module if you do not use Tatami intensively.

Tatami_DojoCharting: loads dojo core and an optimized build of its charting package, so it does not have to be loaded asynchronously. Recommended if you use Tatami's charting facilities

Tatami_DojoGFX: same principle as Tatami_DojoCharting, applied to GFX package

Tatami_DojoWidgets: same principle as Tatami_DojoCharting, applied to the

widgets. Loads all dojo widgets used by Tatami. Recommended if you use most of Tatami Dojo Widgets.

Tatami_DojoAll: Import DojoGFX, DojoWidgets and DojoCharting at load time.

For example, if you want to use the DojoWidgets optimization :

```
<module>
        <inherits name="com.google.gwt.user.User"/>
        <inherits name="com.objetdirect.tatami.
            Tatami_DojoWidget"/>
        <entry-point class="mymodule.client.Module"/>
</module>
```

However, when deploying for a production environment, you should use your custom dojo build, including only what you actually need.

Then, you can choose to use another theme for tatami. For now, the three official themes from dojo are available : tundra (as the default one), soria and nihilo.

To use one or another, just import the corresponding module. For example, if you w

ant to use the soria them with the above configuration :

```
<module>
        <inherits name="com.google.gwt.user.User"/>
        <inherits name="com.objetdirect.tatami.
            Tatami_DojoWidget"/>
        <inherits name="com.objetdirect.tatami.theme.
            Soria"/>
        <entry-point class="mymodule.client.Module"/>
</module>
```

Chapter 2

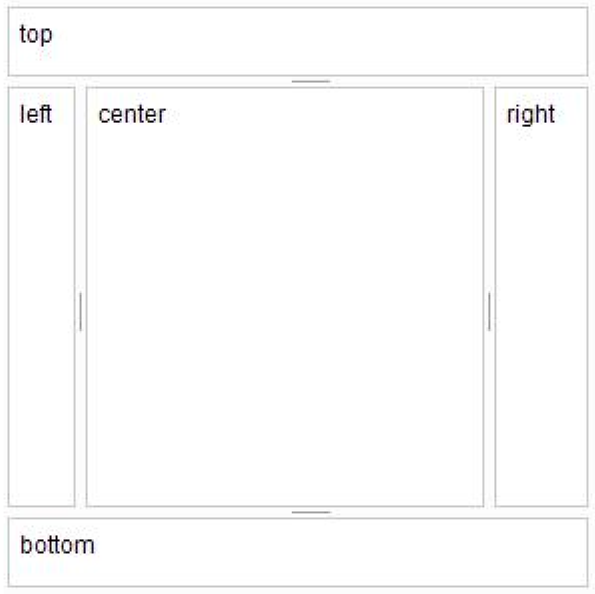
Widgets

2.1 Layout Widgets

Since version 1.3, tatami provides some layout widgets from the dojo world. Most GWTPanels are built from tables, and did not respect the semantic syntax of HTML.

2.1.1 BorderContainer

This layout widget provides is designed as an alternative to both SplitPanel and DockPanel widgets. It is constituted of ContentPanels layed out in Regions, with or without splitters between the center region and the outer ones.



2.1.1.1 Create the BorderContainer layout

You will need to use to import the following classes :

```
import com.objetdirect.tatami.client.layout.
    BorderContainer;
import com.objetdirect.tatami.client.layout.ContentPanel;
```

Then, all you have to do is to instantiate your BorderContainer, and add ContentPanel to it. Only ContentPanels can be added to the BorderContainer, so you will have to wrap your widgets into a ContentPanel:

```
BorderContainer panel = new BorderContainer();
panel.setSize("300px","300px");
HTML left = new HTML("left");
HTML right = new HTML("right");
HTML top = new HTML("top");
HTML bottom = new HTML("bottom");
HTML center = new HTML("center");
panel.add(new ContentPanel(left),BorderContainer.
    REGION_LEFT,true);
panel.add(new ContentPanel(right),BorderContainer.
    REGION_RIGHT,true);
panel.add(new ContentPanel(top),BorderContainer.
    REGION_TOP,true);
panel.add(new ContentPanel(bottom),BorderContainer.
    REGION_BOTTOM,true);
panel.add(new ContentPanel(center),BorderContainer.
    REGION_CENTER);
```

To add a widget to the border container panel, use one of the following methods:

```
public void add(Widget child, String region);
public void add(Widget child, String region,boolean
    splitter);
```

The region argument must be one of the following constants from BorderContainer:

BorderContainer.REGION_CENTER: the content will be placed in the center area

BorderContainer.REGION_TOP: the content will be placed in the top area

BorderContainer.REGION_BOTTOM: the content will be placed in the bottom area

BorderContainer.REGION_LEFT: the content will be placed in the left area

BorderContainer.REGION_RIGHT: the content will be placed in the right area

BorderContainer.REGION_LEADING: same as left except it will be reversed in a right-to-left environment

BorderContainer.REGION_TRAILING: same as right except it will be reversed in a right-to-left environment

The splitter argument determines whether a splitter should be placed between this panel and the center one. If it is true, then the panel will be resizable. If false, it will not. The method public void

```
add(child ,region);
```

will act exactly like

```
add(child ,region ,false);
```

2.1.1.2 Options

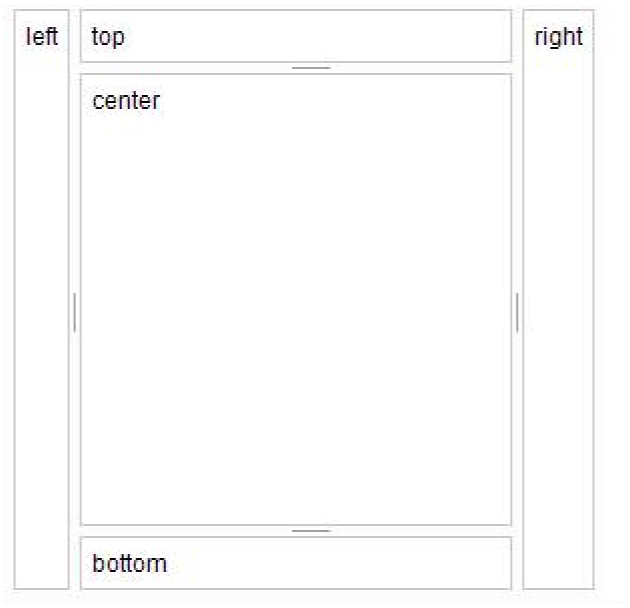
The border container has some options :

liveSplitters: indicates whether the panel should be resized while the splitter is dragged or only when it is released

gutters: indicates whether each content pane should be separated by a margin, a padding and a border (like on the picture below)

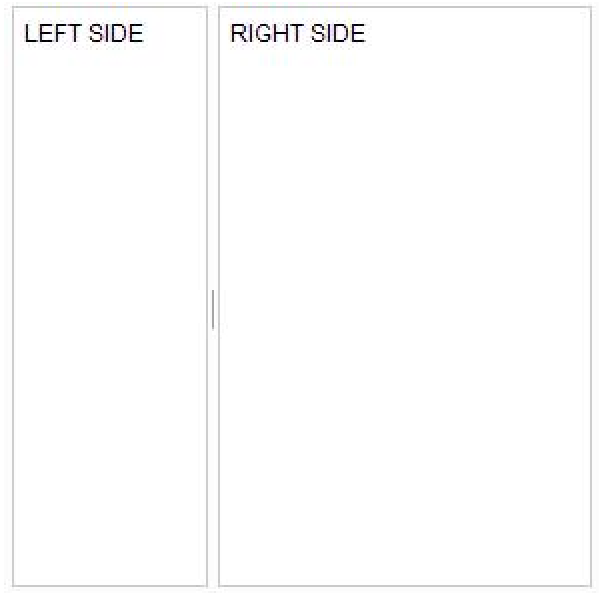
design : one of:

- BorderContainer.DESIGN_HEADLINE if the design is HEADLINE (top and bottom section take the whole width)(defaultValue)
- BorderContainer.DESIGN_SIDEBAR if the design is sidebar (left and right section will take the whole height)
  ex:



2.1.1.3 Examples:

```
BorderContainer panel = new BorderContainer();
panel.setSize("300px","300px");
panel.add(new ContentPanel(new HTML("LEFT_SIDE")),
   BorderContainer.REGION_LEFT, true);
panel.add(new ContentPanel(new HTML("RIGHT_SIDE")),
   BorderContainer.REGION_CENTER);
```

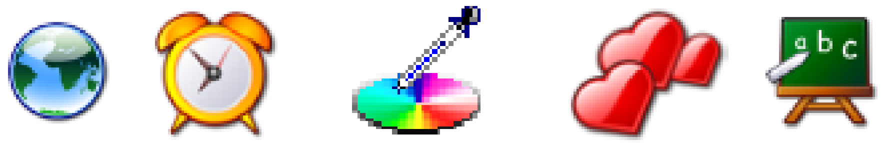


```
BorderContainer panel = new BorderContainer();
panel.setSize("300px","300px");
panel.add(new ContentPanel(new HTML("TOP_SIDE")),
    BorderContainer.REGION_TOP,true);
panel.add(new ContentPanel(new HTML("BOTTOM_SIDE")),
    BorderContainer.REGION_CENTER);
```

2.2 User Input Widgets

2.2.1 FishEye

The FishEye is a dynamic tool bar. The size of icons on the tool bar varies according to the mouse position with respect to the icon. The icons are active ones, in a sense they can be associated with a command which will be triggered when the icon is activated.



2.2.1.1 Create the FishEye component

Before being able to create a FishEye instance, the appropriate import declaration should be added to the class:

```
import com.objetdirect.tatami.client.FishEye;
```

It is then possible to create an instance of the FishEye. The simplest and most common way is to use the default constructor:

```
FishEye fe = new FishEye();
```

This will create a FishEye component with the default behavior: the icon sizes will vary between 50 and 200 pixels, and they will be laid out horizontally. When the icons are magnified by the mouse proximity, they stay centered on the FishEye. The icon title appears just below the icon.

This behavior can be differentiated by using one of the two remaining constructors with parameters.

The first constructor has only the most frequently used parameters: the minimum and maximum icon sizes as well as the component orientation. For instance, here is the creation of a vertical FishEye with icon sizes varying from 40x40 pixels to 400x200 pixels:

```
FishEye fe = new FishEye(40, 40, 400, 200, FishEye.
    VERTICAL);
```

And the corresponding result:

The last constructor allows the specification of all available Dojo parameters. It is therefore slightly more complex to use, some parameters are even not completely understood by the Tatami team!

Those parameters are (in order):

itemWidth: width of menu item (in pixels) in it's dormant state (when the mouse is far away)

itemHeight: height of menu item (in pixels) in it's dormant state (when the mouse is far away)

itemMaxWidth: width of menu item (in pixels) in it's fully enlarged state (when the mouse is directly over it)

itemMaxHeight: height of menu item (in pixels) in it's fully enlarged state (when the mouse is directly over it)

orientation: orientation of the icons bar : FishEye.VERTICAL et FishEye.HORIZONTAL.

effectUnits: controls how much reaction the menu makes, relative to the distance of the mouse from the menu

itemPadding: padding (in pixels) between each menu item

attachEdge: Controls the border that the menu items don't expand past; for example, if set to "top", then the menu items will drop downwards as they expand. FishEye.CENTER, FishEye.TOP, FishEye.BOTTOM, FishEye.LEFT, FishEye.LEFT, FishEye.RIGHT.

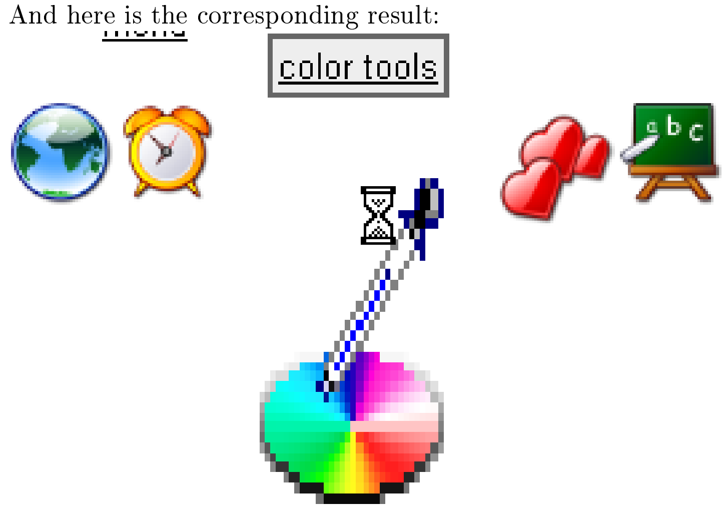
labelEdge: Controls were the labels show up in relation to the menu item icons : FishEye.CENTER, FishEye.TOP, FishEye.BOTTOM, FishEye.LEFT, FishEye.LEFT, FishEye.RIGHT.

conservativeTrigger :if true, don't start enlarging menu items until mouse is over an image; if false, start enlarging menu items as the mouse moves near them.

Here is an example of the use of this constructor:

```
FishEye fe = new FishEye(40, 40, 200, 400, FishEye.
   HORIZONTAL, 1, 0, FishEye.TOP, FishEye.RIGHT, true);
```

And here is the corresponding result:



Instantiating the FishEye is not enough by itself to have it displayed. It has to be hanged in the GWT widget tree.

For a FishEye positioned on the upper left corner of the page body, you should write:

```
RootPanel rootPanel = RootPanel.get(); rootPanel.add(fe,
    50, 50);
```

Take attention to have some margin around the FishEye (50 pixels in both dimensions in the example), otherwise the icons could appear partly outside the page.

2.2.1.2 Adding and removing icons

A FishEye with no icon is useless. The add method can be used to add new icons to the component.

```
Command cmd = new Command() {
        public void execute() {};
}
fe.add("amor.png", "drap'n'drop", cmd);
```

This method requires three parameters:

icon: The URL of the icon to be displayed, as a String,

caption: The title which is displayed near the icon

command: A command implementation (com.google.gwt.user.client.Command) that will be activated when the icon will be selected. .

The remove method, as the reader will have easily guessed, can be used to remove an icon from the FishEye, given its URL:

```
fe.remove("amor.png");
```

It is possible to add and remove icons dynamically, that is while the FishEye is displayed.

2.2.1.3 API of the FishEye

We saw how to add or remove icons from the menu. These two methods are the most important; however the FishEye class provides some other method which can be useful: get the added icons, the command associated...

int countItems(): Counts the number of item in the FishEye menu.

int indexOf(String): Returns the index position in the FishEye menu of an icon.

Command getCommand(String): Returns the Command of the icon from the URL of the associated icon.

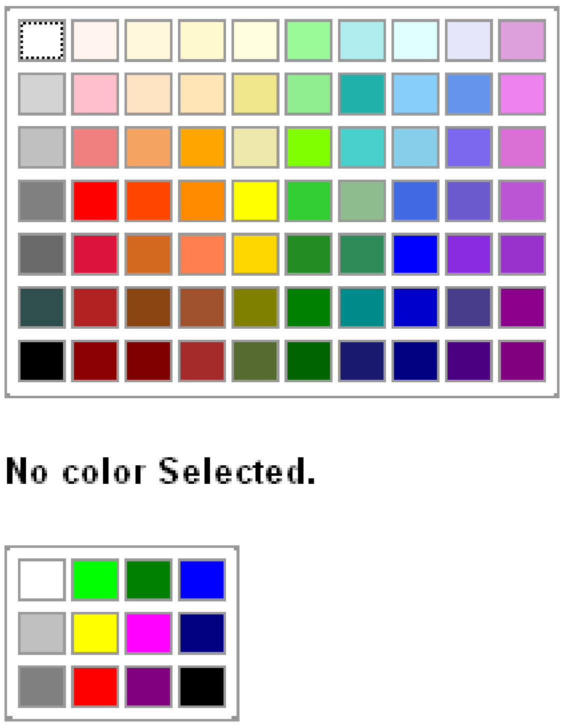
Command getCommand(int): Returns the command at the index position in the FishEye menu.

String[] getIcons(): Returns all the icons from the FishEye menu.

String getIcon(int): Returns the URL of the icon at the index position.

2.2.2 ColorChooser

The ColorChooser class provides to the user a palette of colors to choose from with the mouse. The ColoChooser component can be laid anywhere on the GWT module, it means that there are no constraints with the choice of a container of widgets.



No color Selected.

2.2.2.1 Create a ColorChooser.

The palette proposes a choice of 12 colors or 70 colors. There are two ways to create a palette of colors. One without parameter, the easiest and an other with a parameter staying simple despite everything. First of all, we need to import the class like this:

```
import com.objetdirect.tatami.client.ColorChooser ;
```

2.2.2.1.1 Without parameter: The palette of colors will have 70 colors by default.

```
ColorChooser palette = new ColorChooser();
```

2.2.2.1.2 With parameter: The parameter specifies the number of colors wanted for the palette. There are 2 static constant of the class ColorChooser.

The possible values are :

ColorChooser.SEVENTY_COLORS: For a palette with 70 colors.

ColorChooser.TWELVE_COLORS: For a palette with 12 colors.

What gives us concretely:

```
ColorChooser 12Colors = new ColorChooser(ColorChooser.
    TWELVE_COLORS);
ColorChooser 70Colors = new ColorChooser(ColorChooser.
    SEVENTY_COLORS);
```

2.2.2.2 Get,modify the selected color

. The ColorChooser class doesn't handle really colors i.e. object modelling a color like java.awt.color by example. The ColorChooser class uses the hexadecimal representation format of a color. By default the selected value for each palette (12 or 70 colos) is: #000000 which is the black color.

2.2.2.2.1 Modify the selected color: The modification of the current color can be done by 2 ways. One by specifying the color that we want to select, or by a mouse click event on the palette, in this case it will have to be attached on the browser in order that events are taken into account. The signature of the method to modify the selected color is the following:

```
ColorChooser#setColor(String)
```

The method takes in parameter a string. This string has to correspond to a color in its hexadecimal representation format as we said it before. Warning in this first version of this component there is not checking on the given string.

```
ColorChooser palette = new ColorChooser();
palette.setColor("#c0c0c0"); /* silver color*/
```

2.2.2.2.2 Get the selected color. To get the selected color from the palette, use this simple method below:

```
String color = palette.getColor();
```

The method getColor() returns the hexadecimal representation format of the color.

2.2.2.3 Catch a change of the selected color.

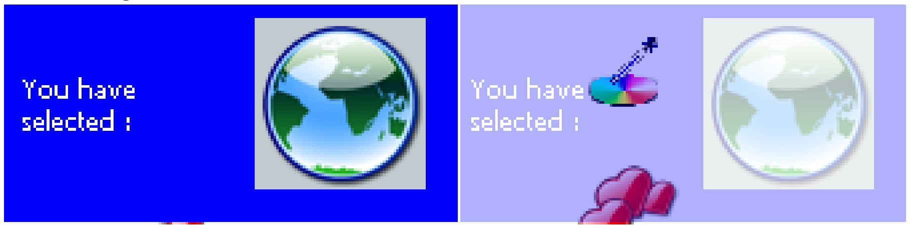
The ColorChooser component has the property to notify a change of the selected color. To use this property, you just need to add a listener with a type com.google.gwt.user.client.ui.ChangeListener.

```
ColorChooser palette = new ColorChooser(); label = new
    Label("no color selected");
palette.addChangeListener(new ChangeListener() {
         public void onChange(Widget sender) {
                  String color = palette.getColor();
                  label.setText("The color selected : " +
                      color);
                  DOM.setStyleAttribute(label.getElement(),
                       "color", color);
         }
});
```

So, each time a new color will be selected, the text of the label will display the hexadecimal code of the color and, the text will have the color of the selected color.

2.2.3 The Toaster component.

The Toaster class aims to display notifications to the users like the mailers application. These notifications are like tooltip appearing in one of the corners of a graphic container and disappearing with a fader effect (become more and more transparent) according to a timeout (1 second by example). The toaster can contain HTML code or simply a non formatted string. It's necessary to note that the Toaster component is invisible for the user. It displays only the wished messages.



2.2.3.1 Create a Toaster.

All object needs to be imported before to be instantiated:

```
import com.objetdirect.tatami.client.Toaster ;
```

To create an instance of a Toaster we need to know principally 2 points. On the one hand is the topic for the component and on the other hand the position where the toaster appears on the graphical container. In fact, the second point, doesn't concern only the position, but also the movement for the Toaster. An example: position in the right lower corner doing a movement from the bottom to the top. The possible positions are static constants of the Toaster class.

Toaster.BOTTOM_RIGHT_UP: Position in the right lower corner, the message goes from the bottom to the top

Toaster.BOTTOM_RIGHT_LEFT: Position in the right lower corner, the message goes from the right to the left.

Toaster.BOTTOM_LEFT_UP: Position in the left lower corner, the message goes from the bottom to the top.

Toaster.BOTTOM_LEFT_RIGHT: Position in the left lower corner, the message goes from the left to the right.

Toaster.TOP_RIGHT_DOWN: Position in the right upper corner, the message goes from the top to the bottom.

Toaster.TOP_RIGHT_LEFT: Position in the right upper corner, the message goes from the right to the left

Toaster.TOP_LEFT_DOWN: Position in the left upper corner, the message goes from the bottom to the top.

Toaster.TOP_LEFT_RIGHT: Position in the left upper corner, the message goes from the left to the right.

The topic of messages of the created Toaster is a string to define. Its usefulness will take a sense during the display of a message. There are 2 ways to create a Toaster. One of them is to specify the position and the topic:

```
Toaster toaster = new Toaster("notification",Toaster.
    TOP_RIGHT_DOWN);
```

An other is to only specify the topic, the position will be then Toaster.BOTTOM_RIGHT_UP

```
Toaster toaster = new Toaster("notification");
```

2.2.3.2 Display a notification.

Once an instance of a Toaster is created, we can throw some notifications. It's at this level that the given topic during creation takes its sense. The Toaster component has 4 notifications available: error message, plain message, warning message and fatal message.

These types are represented by static constants of the Toaster class.

Toaster.ERROR_MESSAGE: Use for error messages.

Toaster.WARNING_MESSAGE: Use for warning messages.

Toaster.FATAL_MESSAGE: Use for fatal messages.

Toaster.PLAIN_MESSAGE: Use for standard messages.

The component has 4 methods for displaying the notifications. One of them is generic; the others use this generic method in order to simplify the passage of the parameters. The signature of the generic method is:

public static void publish(String topic, String message, String type,int delay);

As we can see, the method is static for the class Toaster, that's why we need the parameter « topic » to get the instance of the Toaster to use for the positioning of the message. The parameter « message » will be the content displayed during the notification; it can be a simple string or some HTML code as we had already noted it. The type of the notification can be specified with one of the 4 constants quoted previously. The notification being a panel that it disappears with a timeout, the parameter « delay » specifies the timeout. To avoid using the whole 4 parameters set, the Toaster class has 3 other methods defining the types of notification which will be displayed with a delay at 1 second. (See the constant Toaster.DELAY).

publishMessage(String,String): Display a standard message.

publishWarning(String, String): Display a warning message.

publishError(String,String): Display an error message.

The example below illustrates a use of the Toaster component. It is drawn from the GWT TatamiDemo module, module demonstrating the Tatami components.

```
public class TatamiDemo implements EntryPoint {

public void onModuleLoad() {
        RootPanel rootPanel = RootPanel.get();
        FishEye = new FishEye();
        Toaster toaster = new Toaster("message");
        addItem("background.png", PADDLE, "paddle");
        rootPanel.add(FishEye, 50, 50); rootPanel.add(
            toaster);
}

private void addItem(String icon, int page, String title)
    {
        FishEye.add(icone, title, new DemoCommand(icon,
            page));
}

private class DemoCommand implements Command {

public void execute() { Toaster.publishMessage("message",
   getMessage(icon)); setPage(page); }
```

So, this code displays a notification each time that a user clicks on an item of the FishEye component. The notification will be displayed in the right lower corner and, will go from the bottom to the top with a delay of 1 second.

There is a second way to display a message in the toaster : it can be achieved by setting a message in the toaster and then manually show/hide it.

Below are the methods used to achieve this goal :

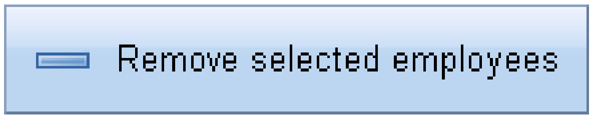
publishMessage(String, String): Display a standard message.

publishWarning(String, String): Display a warning message.

publishError(String, String): Display an error message.

2.2.4 The button component

While GWT provides a standard button , the Dojo button can be more interesting. In Dojo, we define a button appearance by its label and its icon-class. The label is the text which will be shown in the button, while the icon-class is the css class used to style the icon.



For example , the above button's label is “Remove selected employees” while it's icon-class is removeEmployeeButtonIcon, which is defined as following in a css file :

```
.removeEmployeeButtonIcon { background-image: url("list-
   remove.png"); height: 32px; width: 32px; }
```

To import this widget see below:

```
import com.objetdirect.tatami.client.Button ;
```

2.2.4.1 Create the Button component:

The Button's constructor has two parameters : the first one is it's label , the second one is it's icon-class.

label: The label to be displayed on the button

iconClass: The css class which should be applied to the icon.

Tatami's button component is also extending GWT Button. That means you can also use a simple constructor with only the label, or another constructor with a default ClickListener.

```
Button button = new Button("Click␣on␣me","
   addEmployeeButton");
RootPanel.get().add(button);
```

2.2.4.2 API of the button component

Because it is extending the GWT Button Component, it has the same support for ClickListeners. To add a ClickListener to Tatami button component, just use addClickListener(ClickListener listener).

String getText(): Gets the button's label

String getIconClass(): Gets the button's iconClass

void setText(String label): Sets the button's label

void addClickListener(ClickListener listener): Adds a listener for the onClick event

void removeClickListener(ClickListener listener): Removes a listener for the onClick event

2.2.5 The NumberSpinner component

A number spinner is a very convenient widget when it comes to manipulating numbers. Dojo Spinner is wrapped in Tatami by the NumberSpinner component. It provides the following features :

- Setting a max and min value for the spinner
- Value can be changed using up/down key arrows or by up/down buttons
- Spinning accelerates when you hold down a button
- Displaying a tooltip when the value is out of range, when the value is invalid , or as a hint on what to put in the spinner

To import this widget see below:

```
import com.objetdirect.tatami.client.NumberSpinner;
```

2.2.5.1 Create the NumberSpinner component:

There are several constructor's for the number spinner. Let's begin with the simplest one.

```
NumberSpinner spinner = new NumberSpinner();
```

This creates a NumberSpinner with no constraints, with a default value of 0, and a delta of 1. This delta is the spinner's increment.

A second constructor is quite simple :

initValue: Spinner's default value

minValue: Spinner's min value. Any value under minValue will be considered invalid

maxValue: Spinner's max value. Any value above maxValue will be considered invalid

delta: Spinner's delta. It is the spinner's increment when user hit a spinner button

The below snippet will create a spinner which value can range from 500 to 1500 , with an increment of 100 and an init value equals to 1000.

```
NumberSpinner spinner = new NumberSpinner(1000f, 500f,
    1500f , 100f);
```

The following constructor is the most complete one . Notice the constraints object, which is replaced by min and max value in another constructor for more convenience. The constraints object maps constraints to be applied to the spinner content. Please see the example below.

defaultTimeout: Delay until the spinning accelerates

invalidMessage :Message to be displayed in the tooltip when the value is not a number, or doesn't satisfy any constraints.

intermediateChanges: True : the spinner fires onChange event each time it is incremented False : the spinner only fires onChange event when the value is validated (that is, the spinner loses focus)

delta: Spinner's delta. It is the spinner's increment when user hit a spinner button

promptMessage: Helping tooltip to be displayed when the spinner doesn't contain any value.

rangeMessage: Tooltip to be displayed when the value is out of range (that is, above maxvalue or below minvalue)

timeoutChangeRate: This coefficient represents the spinner acceleration. 1.0 means that the spinner will not accelerate at all, values below 1.0 means that the spinner's will accelerate (default to 0.9). The smaller the value is , the more acceleration the spinner will have.

trim: Wether to trim (remove leading and trailing spaces) the number spinner's content

value: Spinner's default value

constraints: A map containing the spinner's constraints

Below , the code that produces the above spinner :

```
Map constraints = new HashMap();
constraints.put("min", new Float(500f));
constraints.put("max", new Float(1500f));
/** The following key/value couple indicates the number *
    format pattern , according to unicode number format
 * pattern.
 * The following pattern also means that any value with
 * more than two decimals will be considered invalid
 */
constraints.put("pattern" , "#,##0");
NumberSpinner spinner = new NumberSpinner(100, "Please␣
    enter␣a␣valid␣number␣without␣any␣decimal␣places",
    false , 100f ,"Please␣enter␣a␣value␣between␣500␣and␣
    1500" , "Value␣is␣out␣of␣range" , 0.90f , true, 1000,
    constraints);
```

2.2.5.2 Constraints

The number spinner component can be constrained with some parameters. Those are:

NumberSpinner.CONSTRAINT_MIN: Sets a minimum value for the spinner.

NumberSpinner.CONSTRAINT_MAX: Sets a maximum Value for the spinner.

NumberSpinner.CONSTRAINT_PATTERN : Sets the number pattern. See http://www.unicode.org/reports/tr35/#Number_Format_Patterns. For example, with a pattern equal to “#,##0.##”, the input numbers will be constrained to 2 decimal places.

NumberSpinner.CONSTRAINT_TYPE: Sets the value type for the number. It can be either “percent” , “decimal” or “currency”. We advise not to set this constraint after the spinner has been created.

NumberSpinner.CONSTRAINT_CURRENCY: Sets the currency symbol for the spinner. It has no effects if the type constraints is not set to “currency”.

To set one of these constraints , just use the following method :

```
public void addConstraint(String, String)
```

For example, if you want to set the max value for your spinner to 50 , just use the following snippet :

```
mySpinner.addConstraint(NumberSpinner.CONSTRAINT_MAX , "
    50");
```

2.2.5.3 API of the number spinner component

The number spinner supports ChangeListeners. To add a ChangeListener to Tatami NumberSpinner component, just use addChangeListener(ChangeListener listener).

String getValue(): Gets the spinner's value

String setValue(): Sets the spinner's value

void addConstraint(String,String): Adds a constraint to the spinner.

void removeConstraint(String): Removes a constraint from the spinner.

void setConstraints(Map constraint): Sets the spinner constraints to the given map, filled with constraint names as keys and their parameter as values.

Map getConstraints(): Gets the spinner constraints map

setDelta(float): Sets the spinner increment

float getDelta(): Gets the spinner increment

void setIntermediateChanges(boolean): Sets wether an "onChange" event should be fired each time the number spinner is incremented (ie, an arrow is clicked) or only when it loses focus.

isIntermediateChanges(): Gets wether an "onChange" event is fired each time the number spinner is incremented.

void addChangeListener(ChangeListener listener): Adds a listener for the onChange event

void removeClickListener(ChangeListener listener): Removes a listener for the onChange event

2.2.6 The clock component.

It can be useful to show a clock in a module, on one hand for informal reasons (know the hour), and on the other hand for the aesthetic aspect of module. The component Clock shows the common hour in a graphic container. The shown clock is a clock with needles.

The clock since Tatami 1.1 has some new features through its GFX package. Indeed the clock is not supported in Dojo 1.0 release, but the Dojo Team decided to create a new Clock component using its GFX package. So the Clock component doesn't wrap a Dojo widget directly and so doesn't implement the HasDojo interface. Now, you can change the background image, the size, colors of the needles... To import this widget see below:

```
import com.objetdirect.tatami.client.Clock ;
```

2.2.6.1 Create the Clock component:

The constructor of clock has two parameters:

url: An URL of an image to use for the background. If null is given then no image will be set

width: The width for the background image, and so the width for the clock itself.

A constructor with no parameter is also available. It means that the clock will be an URL set to null and a width of 192 pixels.

Below a concrete example:

```
Clock clock = new Clock("clock_face.jpg",385);
RootPanel.get().add(clock, 490, 150);
```

2.2.6.2 API of the clock component

Below, these are the methods to custom the clock component.

Date getTime(): Returns the current time of the Clock

String getImage(): Returns background image of the clock or null, if no image used.

int getWidth(): Returns the width of the clock

The following methods set or get the fill color, stroke color or the width of the stroke of a respective needle. (hours, seconds and minutes) :

void setHourColor(Color)

void setHourStrokeColor(Color)

void setHourStrokeWidth(int)

void setMinuteColor(Color)

void setMinuteStrokeColor(Color)

void setMinuteStrokeWidth(int)

void setSecondColor(Color)

void setSecondStrokeColor(Color)

void setSecondStrokeWidth(int)

Color getHourColor()

Color getHourStrokeColor()

int getHourStrokeWidth()

Color getMinuteColor()

Color getMinuteStrokeColor()

int getMinuteStrokeWidth()

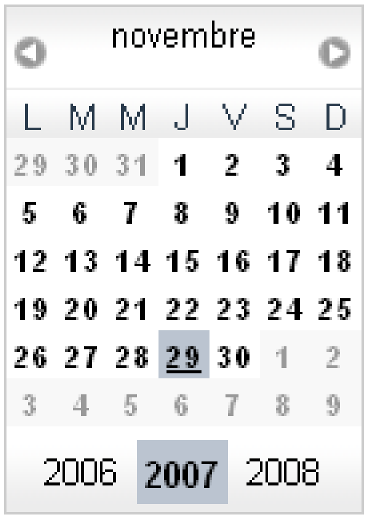
Color getSecondColor()

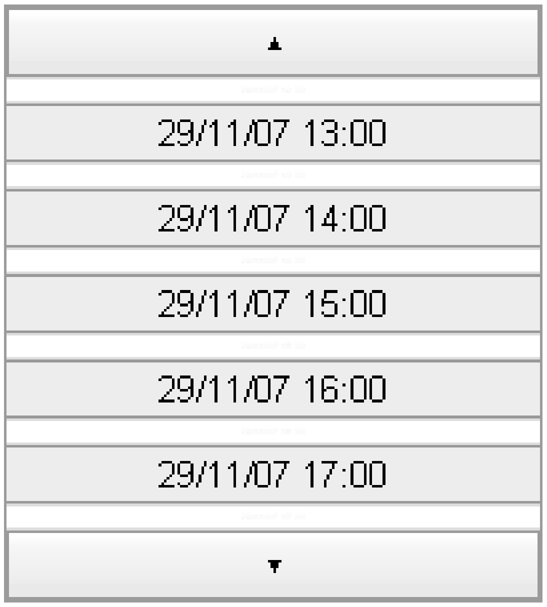
Color getSecondStrokeColor()

int getSecondStrokeWidth()

2.2.7 The BasePicker, TimePicker and DatePicker components.

Tatami offers components allowing the selection of dates or time easily. The DatePicker allows the selection of date within a calendar, whereas the time picker allows the selection of time within a time table. These two components are implementations of the abstract class BasePicker which introduces a common API for components allowing date or time selection.





As for all components in Tatami, it is the addition in the navigator who allows showing components.

2.2.7.1 Create a TimePicker

To create a TimePicker component, you need to specify 3 parameters. These parameters permit to refine the available times that users can select and also the format of the time. By default no time is chosen.

To import the component do this:

```
import com.objetdirect.tatami.client.TimePicker;
```

Below you see the details of the parameters which can be passed to the constructor.

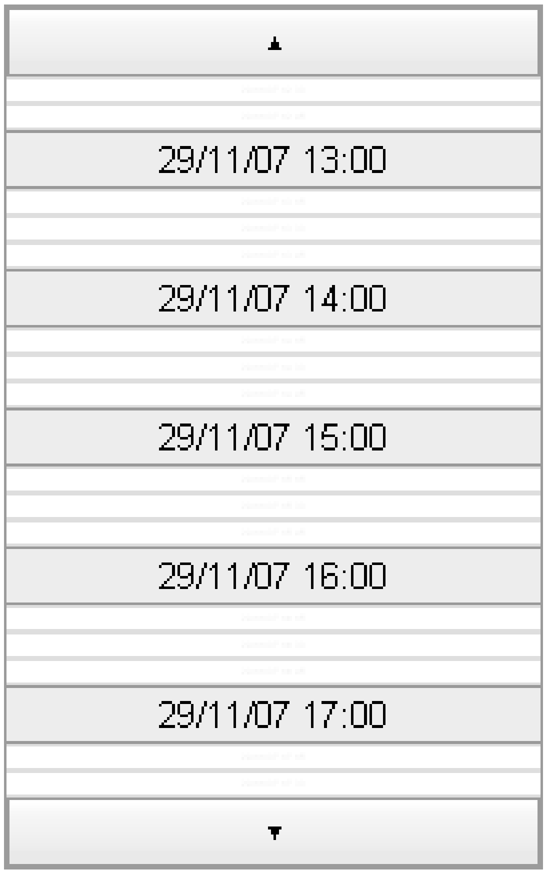
startDate: The "minimum" date that a user can select.

endDate: The "maximum" date that a user can select.

constraints: A TimerPickerConstraint to specify the selectable time and the format of the time to select.

There are also two others constructor, to simplify the use of the TimePicker component.

A constructor with no parameter, there are no time restriction and the style of the TimePicker is like that:



You see 5 hours and these hours are split every 15 minutes. In the second constructor you just specify the constraints.

The TimerPickerConstraint is a class with four public attributes to determine the constraints:

visibleRange: ISO-8601 string representing the range of this TimePicker The TimePicker will only display times in this range Example: "T05:00:00" displays 5 hours of options default is "T05:00:00" :

clickableIncrement: ISO-8601 string representing the amount by which every clickable element in the time picker increases Set in non-Zulu time, without a time zone Example: "T00:15:00" creates 15 minute increments. Must divide visibleIncrement evenly default is "T00:15:00".

visibleIncrement: ISO-8601 string representing the amount by which every element with a visible time in the time picker increases. Set in non Zulu time, without a time zone Example: "T01:00:00" creates text in every 1 hour increment default is "T01:00:00",

timePattern: TimePattern see the DOJO explanations : it's like in JAVA for the most of principals options default is "HH:mm"

The instantiation of a TimePicker is made as follows:

```
TimePickerConstraints constraints= new
    TimePickerConstraints();
constraints.clickableIncrement = TimePickerConstraints.
    EVERY_HALF_HOUR;
timePicker = new TimePicker(constraints);
```

As you can see in the code some constants are defined to simplify the use of the TimePickerConstraint class.

2.2.7.2 Create a DatePicker

The component DatePicker allows the selection of date, the date to choose can be bounded by a lower and an upper limit like the TimePicker. It's possible to not specify these limits; in that case all possible dates will be effluent. The importation of the class this fact in the following way:

```
import com.objetdirect.tatami.client.DatePicker;
```

To select a Date without constraints, do this:

```
DatePicker datePicker = new DatePicker();
```

To select a Date with constraints do this :

```
//start2007 correspond to the date : 01/01/2007
//end2008 correspond to 12/31/2008
DatePicker picker = new DatePicker(start2007, end2008);
```

2.2.7.3 The API of DatePicker and TimePicker:

The API to manipulate a BasePicker consists in choosing a date in most cases and recovering this date chosen. The user can choose a date of course by mouse events on the component.

To set the default selected date and get the date selected by a user:

void setDate(Date): sets the default selected date

Date getDate(): get the currently selected date

void setTime(Date): sets the default selected time for a TimePicker

Date getTime(): gets the default selected time for a TimePicker

void getMinDate(): Returns the minimum available date in the calendar or in the time picker

void getMaxDate(): Returns the maximum available date in the calendar or in the time picker

void addChangeListener(ChangeListener): Adds a ChangeListener when the seletected date value changes.

void removeChangeListener(ChangeListener): Removes a ChangeListener when the seletected date value changes.

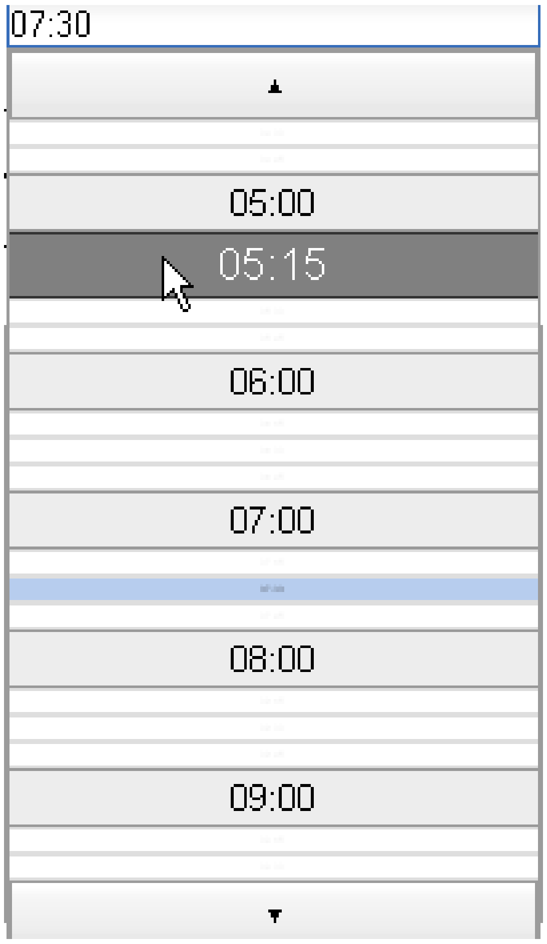
We can listen to selection changes, by adding a listener of type com.google.gwt.user.client.ui.ChangeListener to the component.

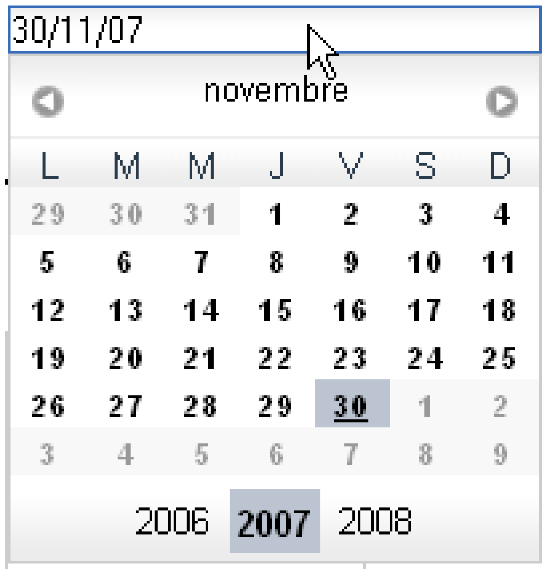
```
TimePicker timer = new TimePicker();
timer.addChangeListener(new ChangeListener() {
        public void onChange(Widget sender) {
                Window.alert("New time :" + timer.getTime
                    () );
        }
});
```

Each times a new time will be selected; an alert will be displayed showing the new selection.

2.2.8 The DropdownContainer, DropdownDatePicker et DropdownTimePicker components.

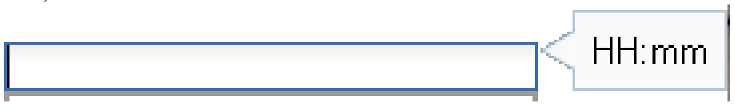
Tatami offers components which allow the simplification of an entered of temporal information as date or a time. Both components which allow this are DropdownDatePicker (at the left) for a date input and DropdownTimePicker (at the right) for a time input. These two classes implement the abstract class DropdownContainer allowing the input of temporal information.



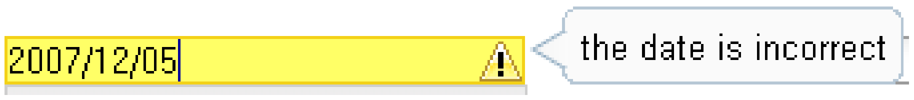


The inputs are simplified due to the fact that the user will be able to select their temporal information using a BasePicker component (ie, TimePicker or DatePicker). Once its information was chosen, the field text of seizure will have the value of this information and will be formatted.

Moreover, these components perform a validation on the entered text and indicate in a tooltip the expected format (for example: hh:mm for time information).



If the format is incorrect the user is warned by another tooltip and the color of the text field changes:



2.2.8.1 Create a DropdownDatePicker.

The component DropdownDatePicker allows to grab a date in a text field via a DatePicker component. So, the selected date will be formatted accordingly to the user's locale. Example: 08/28/2007 for a locale EN.
To import the class:

```
import com.objetdirect.tatami.client.DropdownDatePicker;
```

There are 2 constructors for a DropDwonDatePicker, one with two parameters and one with no parameters. The parameters will have default values.

startDate: the lowest allowed value

endDate: the highest allowed value

An example:

```
DropdownDatePicker dddp = new DropdownDatePicker();
```

Or

```
DropdownDatePicker dddp = new DropdownDatePicker(
    startDate, endDate);
```

This creates our component, which has to be attached to the browser.

2.2.8.2 Create a DropdownTimePicker.

The DropdownTimePicker component permits to enter a time with hours and minutes in an text fied via the TimePicker component. So, the selected date will be formatted accordingly to a defined time pattern. Example: 2:30
To import the class:

```
import com.objetdirect.tatami.client.DropdownTimePicker;
```

There are 3 constructors available. The most "complex" constructor needs 3 parameters. The others set some default values on these 3 parameters.

startDate: the lowest allowed value

endDate: the highest allowed value

timePattern: To format the time information. See the DOJO explanations for more details about the time pattern. It's like in JAVA for the most of principals options. Default is "HH:mm"

An example:

```
DropdownTimePicker ddtp = new DropdownTimePicker();
```

Or

```
DropdownTimePicker ddtp = new DropdownTimePicker("HH:mm")
    ;
```

Or

```
DropdownTimePicker ddtp = new DropdownTimePicker(
    startDate, endDate,"HH:mm");
```

This creates our component, which has to be attached to the browser.

2.2.8.3 The API of the DropdownTimePicker and DropdownDatePicker.

The API of these components comes from the DropdownContainer class.

void setDate(Date): sets the default selected date

Date getDate(): get the currently selected date

void setTime(Date): sets the default selected time for a TimePicker

Date getTime(): gets the default selected time for a TimePicker

void getMinDate(): Returns the minimum available date in the calendar or in the time picker

void getMaxDate(): Returns the maximum available date in the calendar or in the time picker

void addChangeListener(ChangeListener): Adds a ChangeListener when the seletected date value changes.

void removeChangeListener(ChangeListener): Removes a ChangeListener when the seletected date value changes.

void setText(String): Writes some text in the text field of the Dropdown-Container.

String getText(): Returns the time information as a string.

void setEnabled(boolean): Enables or disables the DropdownContainer.

void setInvalidMessage(String): Sets the message displayed when the user's input is not valid.

String getInvalidMessage(): Gets the message displayed when the user's input is not valid.

void setPromptMessage(String): Sets the message helping the user to respect the expected format.

String getPromptMessage(): Gets the message helping the user to respect the expected format.

Boolean isValid(): Indicates if the entered text is valid or not.

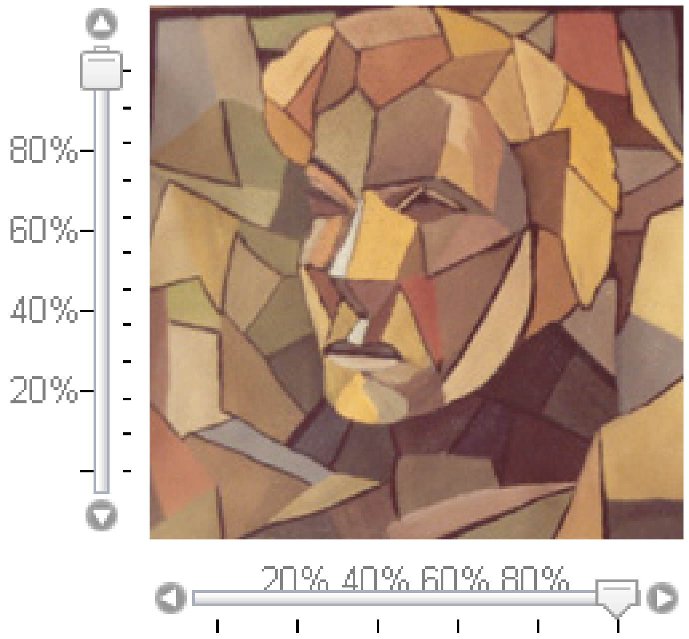
Below, you will find an example extracted from the project TatamiDemo-1.1 which presents the DropdownTimePicker, DropdownDatePicker, TimePicker and DatePicker components. These different components notify their change of values and update some others components treating the same type of data. It's means that the DatePicker will be linked with the DropdownDatePicker and the TimePicker will be linked with the DropdownTimePicker. So if the TimePicker is modified then it will update the DropdownTimePicker if it's necessary, same for the DatePicker with the DropdownDatePicker.

```
private void initComponents() {
        mainPanel = new HorizontalPanel();
        mainPanel.setSpacing(50);
        VerticalPanel timePanel = new VerticalPanel();
        VerticalPanel datePanel = new VerticalPanel();
        datePanel.setSpacing(20);
        timePanel.setSpacing(20);
        inputDate = new DropdownDatePicker();
        inputDate.setInvalidMessage("the␣date␣is␣
            incorrect");
        datePicker = new DatePicker();
        datePicker.setDate(new Date());
        datePanel.add(htmlInputDate);
        datePanel.add(inputDate);
        datePanel.add(htmlDatePicker);
        datePanel.add(datePicker);

        linkDropdownAndPicker(inputDate, datePicker);
        inputTime = new DropdownTimePicker();
        inputTime.setPromptMessage("HH:mm");
        TimePickerConstraints constraints= new
            TimePickerConstraints();
        constraints.clickableIncrement =
            TimePickerConstraints.EVERY_HALF_HOUR;
        timePicker = new TimePicker(constraints);
        timePanel.add(htmlInputTime);
        timePanel.add(inputTime);
        timePanel.add(htmlTimePicker);
        timePanel.add(timePicker);
        linkDropdownAndPicker(inputTime, timePicker);
        mainPanel.add(datePanel);
        mainPanel.add(timePanel);
}
private void linkDropdownAndPicker(final
    DropdownContainer container,final BasePicker picker) {
        container.addChangeListener(new ChangeListener()
            {
                    public void onChange(Widget sender) {
                              if (!equalsObj(picker.getDate(),
                                  container.getDate())) {
                                        picker.setDate(container.
                                            getDate());
                              }
                    }
          });
          picker.addChangeListener(new ChangeListener() {
                    public void onChange(Widget sender) {
                              if (!equalsObj(container.getDate
                                  (), picker.getDate())) {
                                        container.setDate(picker.
                                            getDate());
                              }
                    }
          });
}
```

2.2.9 The Slider component

The Slider component permits to modify the position of a cursor. At this position, is associated a value. The value is bounded by a maximum and a minimum value. The slider has two positions: vertical and horizontal.



2.2.9.1 Create a Slider.

To import the Slider class:

```
import com.objetdirect.tatami.client.Slider;
```

To create a Slider, we need 5 parameters:
position: Slider.HORIZONTAL or Slider.VERTICAL determines the position for the slider.

minimum: Sets the minimum value available on the slider

maximum: Sets the maximum value available on the slider

initialValue: determines the initial cursor's position.

showButtons: Boolean permitting to show or hide the button to increment or decrement the position of the cursor.

Example of using the Slider constructor :

```
// 0 is the min value.
// 100 is the max value.
// 50 is the initial value.
//no buttons are shown
Slider slider = new Slider(Slider.HORIZONTAL, 0,100,50,
    false);
```

2.2.9.2 The API of the Slider component.

Principal methods from the Slider component :

void setValue(int): Modifies the selected value on the cursor

int getValue(): Returns the current selected value.

int getMaximunValue(): Returns the maximum allowed value.

int getMinimumValue(): Returns the minimum allowed value.

void setEnabled(boolean): enables or disables the slider

Boolean isEnabled() : returns whether the slider is enabled or not

The following methods are used to add or remove rule marks at different positions on the slider. The first parameter sets the number of ticks, and the second one their size in any standard css unit ("px","em" etc...).

void setRuleMarkBottom(int,String)

void setRuleMarkTop(int,String)

void setRuleMarkLeft(int,String)

void setRuleMarkRight(int,String)

void removeRuleMarkBottom()

void removeRuleMarkTop()

void removeRuleMarkLeft()

void removeRuleMarkRight()

The following methods are used to add or remove labels on each rule mark set. The first argument is an array containing the labels, the second one is the css style applied to this label.

void setRuleLabelsTop(String[],String)

void setRuleLabelsBottom(String[],String)

void setRuleLabelsLeft(String[],String)

void setRuleLabelsRight(String[],String)

void removeLabelsTop()

void removeLabelsBottom()

void removeLabelsLeft()

void removeLabelsRight()

The following methods return the labels on a specific position. If there are no labels, they return null.

String[] getLabelsLeft()

String[] getLabelsRight()

String[] getLabelsBottom()

String[] getLabelsTop()

The following methods return the labels style for a specific position:

String getLabelsTopStyle()

String getLabelsBottomStyle()

String getLabelsRightStyle()

String getLabelsLeftStyle()

The following methods return the number of mark at the specified position:

int countRuleMarkTop()

int countRuleMarkBottom()

int countRuleMarkLeft()

int countRuleMarkRight()

The following methods return the mark size at the specified position

String getSizeRuleMarkTop()

String getSizeRuleMarkBottom()

String getSizeRuleMarkLeft()

String getSizeRuleMarkRight()

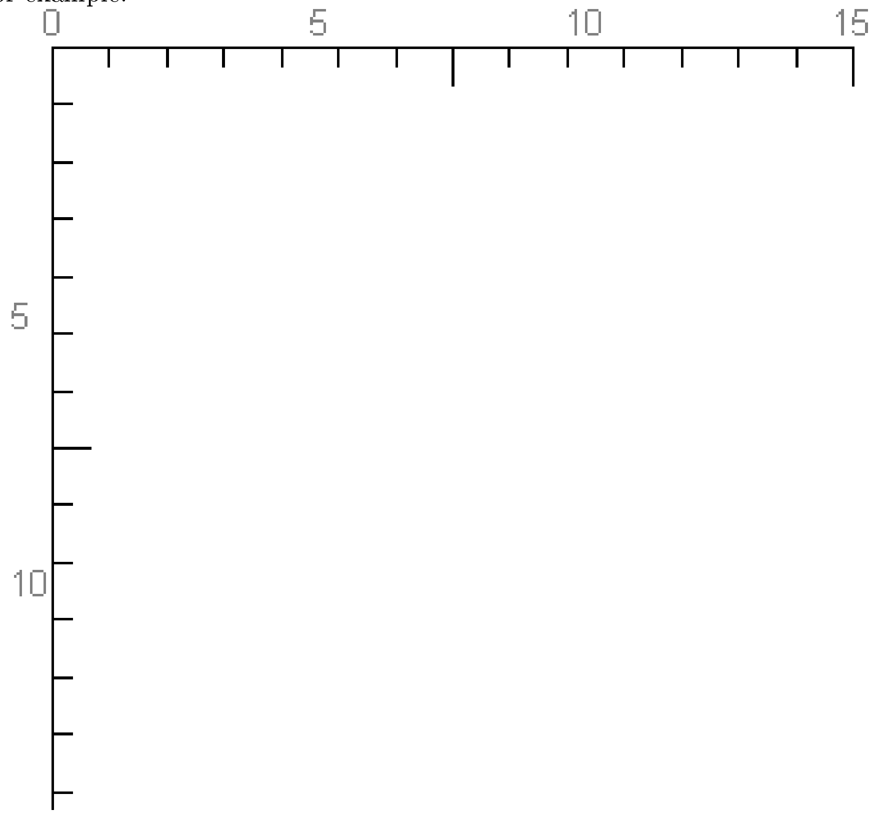
2.2.9.3 Add some rule mark ands and labels.

Below stands an example of how to add some rule mark and labels on a Slider component. See also the project TatamiDemo-1.1. You can add a rule mark at the right, left, top, bottom of the slider, it depends of the position (vertical or horizontal) of the slider. But if you add a rule mark at the right position while the slider has a horizontal value, the rule mark will be displayed at the bottom of the slider. To set a rule mark, you need to know how many marks you want and their size. For the labels it's the same thing, but instead of giving a number you give an array of String corresponding to each label. Moreover you can specify the labels css style.

```
Slider verticalSlider = new Slider(Slider.VERTICAL, 0,
    100, 100,true);
verticalSlider.setRuleMarkLeft(6, "5px");
verticalSlider.setRuleMarkRight(12, "3px");
String[] labels = {"␣","20%","40%","60%","80%", "␣"};
verticalSlider.setLabelsLeft(labels,"margin:␣0px␣-0.5em␣0
    px␣-2em;color:gray");
Slider horizontalSlider = new Slider
(Slider.HORIZONTAL, 0, 100, 100,true);
horizontalSlider.setRuleMarkBottom(6, "5px");
    horizontalSlider.setLabelsTop(labels,"margin:␣-0.5em␣0
    px␣-3.5em␣0px;color:gray");
```

2.2.10 RuleMark and RuleLabels components

RuleMark and RuleLabels permit to create some mark or some labels in a vertical or horizontal position. The distance between each marks or labels is based on the size of the component (RuleMark or RuleLabels). These components are used in the Slider component; see the documentation of this component to understand how RuleMark and RuleLabels are created. Below, this is an example of RuleMark and RuleLabels components to create a rule for a graphic canvas or an editor for example.



2.2.10.1 Create a RuleMark component.

To import the RuleMark class, just do this:

```
import com.objetdirect.tatami.client.RuleMark;
```

To create a RuleMark component you need specify 4 parameters. One parameter is optional; the others are required and cannot be have a default value.

type: Determines if the RuleMark will be set in horizontal or vertical position

count: The number of marks to display

size: The size for each mark in pixel, em...

position: The position to use for the RuleMark in the parent container, this parameter is optional. But it is used in the Slider to specify the position of the RuleMark on the Slider left, top, bottom, right. In fact if this RuleMark is a child of a dojo widget, this RuleMark will be connected to its parent node by the position. The default position is “containerNode”.

2.2.10.2 The API of the RuleMak component

String getSize(): Returns the size of each mark. The size is in pixel, em....

int getCount(): Returns the number or marks in the RuleMark.

String getPosition(): Returns the position of the RuleMark, default is containerNode

2.2.10.3 Create a RuleLabels Component.

A RuleLabels component extends the RuleMark component, but instead of writing some marks, some labels are displayed. To import the class do this:

```
import com.objetdirect.tatami.client.RuleLabels;
```

Like the RuleMark, the RuleLabels needs 4 parameters but instead of specifying the number of marks, we specify an array of labels where labels are just string characters. So the number of labels is calculated by the length of the array. Also, we specify the style for the labels and not the size. The style is exactly the same than a CSS style. (color, font-family,margin...)

type: Determines if the RuleMark will be set in horizontal or vertical position

labels: An array of string corresponding to the labels.

style: The style for the labels (color, font, margin etc...) Use the CSS syntax.

position: The same that the RuleMark class.

2.2.10.4 The API of the RuleLabels component

String[] getLabels(): Returns all the labels to display.

String getStyle(): Returns the style used for each label.

int getCount(): Returns the number or labels.

String getPosition(): Returns the position of the RuleLabels, default is containerNode

2.2.10.5 An example

Below stands an example, which demonstrates how to create the rule presented in the screenshot at the beginning of this chapter. Note that in the project TatamiDemo there are no example. The use of RuleMark and RuleLabels is hidden by the Slider component in the SliderDemo class.

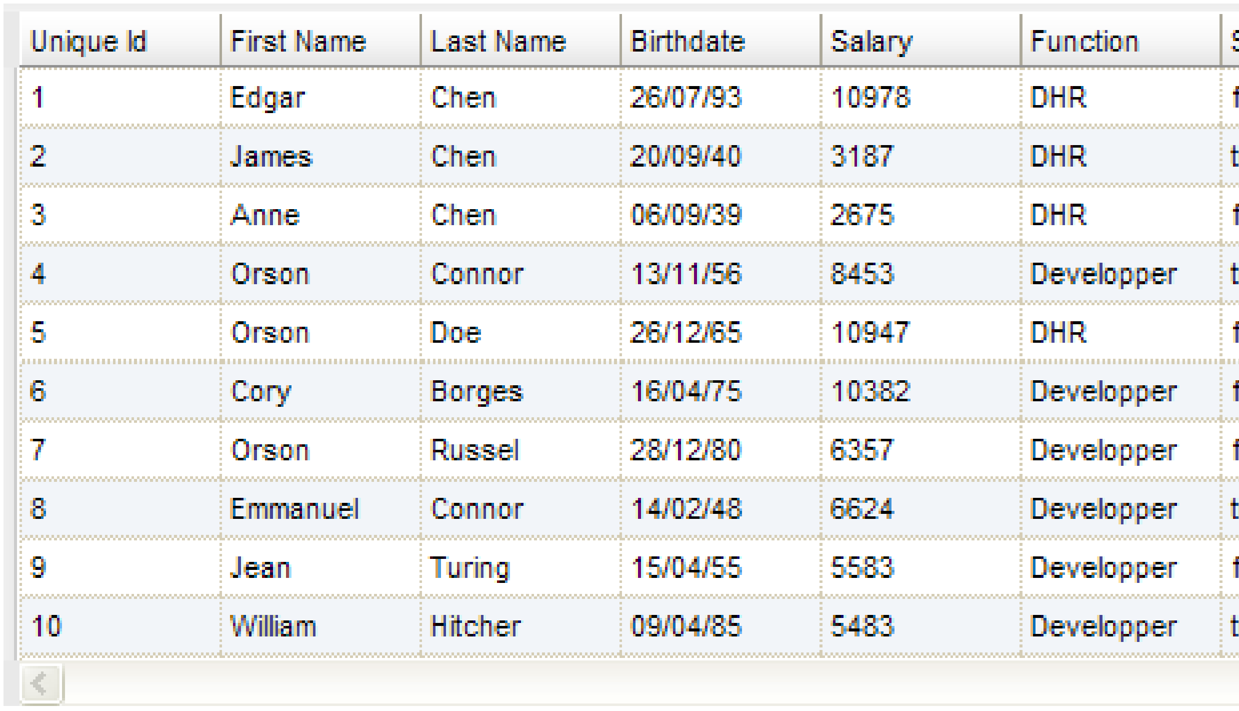
```
RuleMark rule = new RuleMark(RuleMark.HORIZONTAL,15,"7px"
    );
DOM.setStyleAttribute(rule.getElement(),"borderTop","
    solid␣1px");
```

```
RuleMark rule2 = new RuleMark(RuleMark.HORIZONTAL,3,"15px
    ");
rule.setWidth("300px");
rule.setHeight("20px");
rule2.setWidth("300px");
RootPanel.get().add(rule,20,100);
RootPanel.get().add(rule2,rule.getAbsoluteLeft(), rule.
    getAbsoluteTop());
String[] labelsArray = {"0","5","10","15"};
RuleLabels labels = new RuleLabels(RuleLabels.HORIZONTAL,
     labelsArray, "margin:-0.5em␣-0.5px␣5px␣-.2.5em;color:
    gray");
labels.setWidth("300px");
RootPanel.get().add(labels,rule.getAbsoluteLeft(), rule.
    getAbsoluteTop()-10);
RuleMark rule4 = new RuleMark(RuleMark.VERTICAL,15,"7px")
    ;
RuleMark rule5 = new RuleMark(RuleMark.VERTICAL,3,"15px")
    ;
rule4.setHeight("300px");
rule5.setHeight("300px");
RootPanel.get().add(rule4,rule.getAbsoluteLeft(), rule.
    getAbsoluteTop());
DOM.setStyleAttribute(rule4.getElement(),"borderLeft","
    solid␣1px");
RootPanel.get().add(rule5,rule.getAbsoluteLeft(), rule.
    getAbsoluteTop());
RuleLabels labels2 = new RuleLabels(RuleLabels.VERTICAL,
    labelsArray2, "margin:␣-0.5em␣-0.5px␣5px␣-.2.5em;color
    :gray");
labels2.setHeight("300px");
RootPanel.get().add(labels2,rule.getAbsoluteLeft()-15,
    rule.getAbsoluteTop());
```

2.3 Advanced Widgets

2.3.1 The Grid component

Introduced in Tatami version 1.2 , the grid component provides a rich table, with nice features such as client-side sorting, lazy loading , view change , paging , in-cell editing, data formatting ...



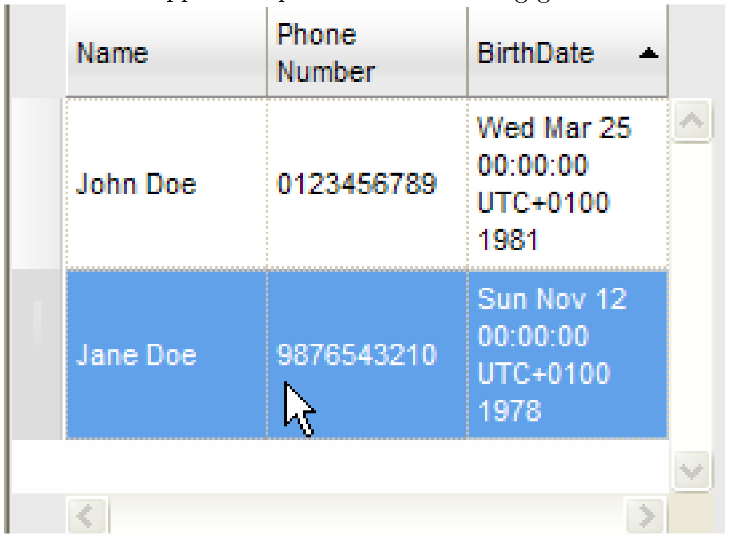
Unique Id	First Name	Last Name	Birthdate	Salary	Function	S
1	Edgar	Chen	26/07/93	10978	DHR	f
2	James	Chen	20/09/40	3187	DHR	t
3	Anne	Chen	06/09/39	2675	DHR	f
4	Orson	Connor	13/11/56	8453	Developper	t
5	Orson	Doe	26/12/65	10947	DHR	f
6	Cory	Borges	16/04/75	10382	Developper	f
7	Orson	Russel	28/12/80	6357	Developper	f
8	Emmanuel	Connor	14/02/48	6624	Developper	t
9	Jean	Turing	15/04/55	5583	Developper	f
10	William	Hitcher	09/04/85	5483	Developper	t

2.3.1.1 Building and manipulating a simple grid : an example

This section will introduce the basics about building and manipulating a grid. We will learn how to create a grid, how to add/remove rows or columns and the basic grid settings. For further information, refer to Grid API and Grid MVC sections

```
Grid grid = new Grid();
grid.addColumn("Name");
grid.addColumn("Phone␣Number");
grid.addColumn("BirthDate");
Object[] row1 = {"John␣Doe" , "0123456789" , new Date("
    03/25/1981") };
Object[] row2 = {"Jane␣Doe" , "9876543210" , new Date("
    11/12/1978")};
grid.addRow(row1);
grid.addRow(row2);
grid.setWidth("300px");
grid.setHeight("200px");
RootPanel.get().add(grid);
grid.updateGrid();
```

The above snippet will produce the following grid :



Name	Phone Number	BirthDate ▲
John Doe	0123456789	Wed Mar 25 00:00:00 UTC+0100 1981
Jane Doe	9876543210	Sun Nov 12 00:00:00 UTC+0100 1978

Notice that the date is displayed using the standard ISO notation. Please refer to Cell editing and formatting section know how to display it properly.

It's default properties make it user sortable, and we can add and remove rows and columns as we wish.

2.3.1.1.2 Settings

After creating the grid , and before attaching it to the dom tree, we can set a bunch of properties.

setAutoHeight(boolean): true indicates that the grid should auto adapt its height, false that its height is fixed. Default : false

setAutoWidth(boolean): true indicates that the grid should auto adapt its width, false that its width is fixed. Default : false

setElasticView(int): Indicates which view should be elastic, which means that it will fill the remaining space once all other views are drawn. Default : -1 (none)

setMaximumFetchCountAtAtime(int): Sets the maximum number that are returned at a time. It indicates how many items should be loaded. When the user scrolls, items to be displayed are fetched by groups of maximumFetchCountAtAtime items. A typical setting for this value is 1.5 the number of rows to be displayed at a time. (That is , the grid won't have to load items until the user scrolls) Default:25

setRowsPerPage(int): Sets the number of row per page. A value of -1 indicates that all rows should fit on a single page. That does not mean they are all loaded at one time, since only the displayed rows are loaded, and updated when the user scrolls. Default : -1

setSortIndex(int , boolean) : Sets the grid sorting. sortIndex : the column index by which the grid should be sorted ascending : indicates wether the sort should be performed in ascending or descending order

setUserSortable(boolean): Sets wether the user can sort the grid by clicking on the column header. Default : true.

setRenderOnload(boolean): Sets wether the grid content should be rendered as soon as the grid is displayed (attached on GWT panel)

setRowSelector(String) Sets wether a row handle should be displayed. A row handle is a bar on the left side that allows the user to select a row without clicking on a particular cell (and without firing an oncellclick event). The given parameter is its size, in css units.

2.3.1.1.3 Adding / removing rows:

Tatami Grid provides a simple way to manipulate rows. Once again, please refer to Grid MVC section for a more complete way to do that.

When you add or remove a row , the displayed grid is automatically updated and reflects the changes.

Example :

```
Object[] row = {"Jane Doe" , "9876543210" , new Integer("
    1200") , new Date("11/12/1978") , "Developer" ,
    Boolean.TRUE};
grid.addRow(row);
grid.removeRow(0);
```

addRow(Object[]): Adds a row , taking each object from the array as the value for the corresponding column

addRow(Object[],int index): Same as previous, except that instead of adding it to the end of the grid, it adds it to the given index.

removeRow(int): Removes the row at specified index

addRow(Item): Delegates method for addItem in the underlying datastore. See DataStore and Items to learn how to manipulate Items directly

addRow(Item , int): Same as previous , except that it adds it at the specified index

removeRow(Item): Delegates method for removeItem in the underlying datastore. See DataStore and Items to learn how to manipulate Items directly

getRowCount(): Returns the current row count

2.3.1.1.4 Adding / removing columns

Columns can be added or removed at any time during grid lifecycle. To refresh the grid view (ie, how the columns are layed out), you must call the updateView() method.

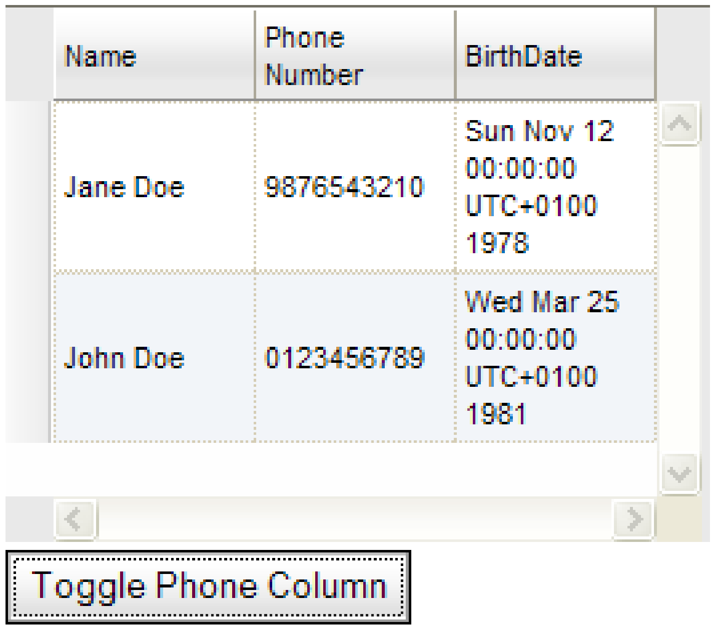
For example, if we want to add a toggle button, to hide/show a column, we can code it just like that :'

```
Button button = new Button("Toggle␣Phone␣Column");
button.addClickListener(new ClickListener(){
        private boolean isShown = true;
        public void onClick(Widget sender){
                if(isShown){
                        grid.removeColumn(1);
                        grid.updateView();
                        isShown = false;
                }else{
                        grid.insertColumn("Phone␣Number"
                            , 1);
                        grid.updateView();
                        isShown = true;
                }
        }
});
RootPanel.get().add(button);
```

As we notice , those insertColumn and removeColumn methods take the column index as a parameter.

On the left, before hitting the button

On the right, after.



Name	Phone Number	BirthDate
Jane Doe	9876543210	Sun Nov 12 00:00:00 UTC+0100 1978
John Doe	0123456789	Wed Mar 25 00:00:00 UTC+0100 1981

Name
Jane Doe
John Doe

addColumn(String name): Adds a column with the specified name

addColumn(String name,String fieldName): Adds a column with the specified name, which value is taken from the specified items attributes.

addColumn(String,String,Formatter): Same as previous , except that you specify a Formatter , which is responsible for formatting the data to be displayed (See Formatting section)

addColumn(String,String,GridEditor): Same as addColumn(String name, String fieldName), except that you specify an Editor, making this column editable (See Editing section)

addColumn(String,String,GridEditor,Formatter): Same as the two previous

addColumn(String,String,int): Same as addColumn(String name, String fieldName), except that you specify a cell width in pixel

addColumn(String name,GridEditor editor,Formatter formatter,String width): Adds a column with the specified name, editor , formatter and width in standard css units (ex : "100px" , "10em"). Null values are accepted, except for width which could lead to strange behaviours

addColumn(String name,String fieldName,String width,GridEditor editor,Formatter formatter) Same as previous , except you specify the field name from which values should be taken

addCell(Cell): Adds the specified cell as a column in the underlying GridView (See GridView and Cells)

removeColumn(int): Removes the column at specified index

removeColumn(Cell): Removes the specifided cell as a column in the underlying GridView (See GridView and Cells)

updateView(): Forces the grid to refresh its view, updating its columns definitions.

2.3.1.1.5 Paging

Tatami's grid supports two types of paging : the dojo's one, which loads the grid rows as the user scrolls its grid, and a "previous page" / "next page" pagination.

This section will present you both, and how you can enable / disable any of them.

2.3.1.1.5.1 Dojo's "OnScroll" paging

To prevent massive requests to the datastore (which can be a remote datastore, requesting the server each time a fetch operation is performed), dojo provides an on scroll paging mechanism.

That means the grid will only fetch items that need to be displayed, by set of maximumFetchCountAtAtime. When the user scrolls the grid vertically, more rows need to be displayed , so the grid will fetch a new set of items.

This value is 25 by default , but we can specify it for the grid to be more efficient.

For example, if we know that we will always display 10 rows , setting the maximumFetchCountAtAtime to 20 will be sufficient to assure a quite reactive scrolling: the user will be able to scroll down the 10 next rows without a need for re-fetching data from the data store.

To perform this operation, just use the following method :

```
public void setMaximumFetchCountAtAtime(int
    maximumFetchCountAtAtime)
```

2.3.1.1.5.2 Tatami pagination system

While dojo does not offer a "real" pagination system, one is built-in tatami's grid. For using it, we just need to set the rowsPerPage parameter to something different from -1.

Example :

```
grid.setRowsPerPage(50);
```

So, the grid will only display the first 50 rows, according to the specified order/filter.

Then if we want to turn pages, we just have to usse one of the following methods :

```
grid.previousPage();
grid.nextPage();
grid.goToPage(int pageIndex);
```

2.3.1.1.5.3 Pagination related Grid API

getMaximumFetchCountAtAtime(): gets how much items should be loaded at a time (for onScroll paging)

setMaximumFetchCountAtAtime(int): sets how much items should be loaded at a time (for onScroll paging)

getRowsPerPage(): gets the number of row to be displayed on each page

setRowsPerPage(int): sets the number of row to be displayed on each page

previousPage(): Go to previous page

nextPage(): Go to next page

goToPage(int): Go to the given page index

2.3.1.1.6 Filtering

Tatami's grid can easily perform filters. For now, it only supports exact-match filters but we expect to support regular expressions and simple mathematic operators in the future.

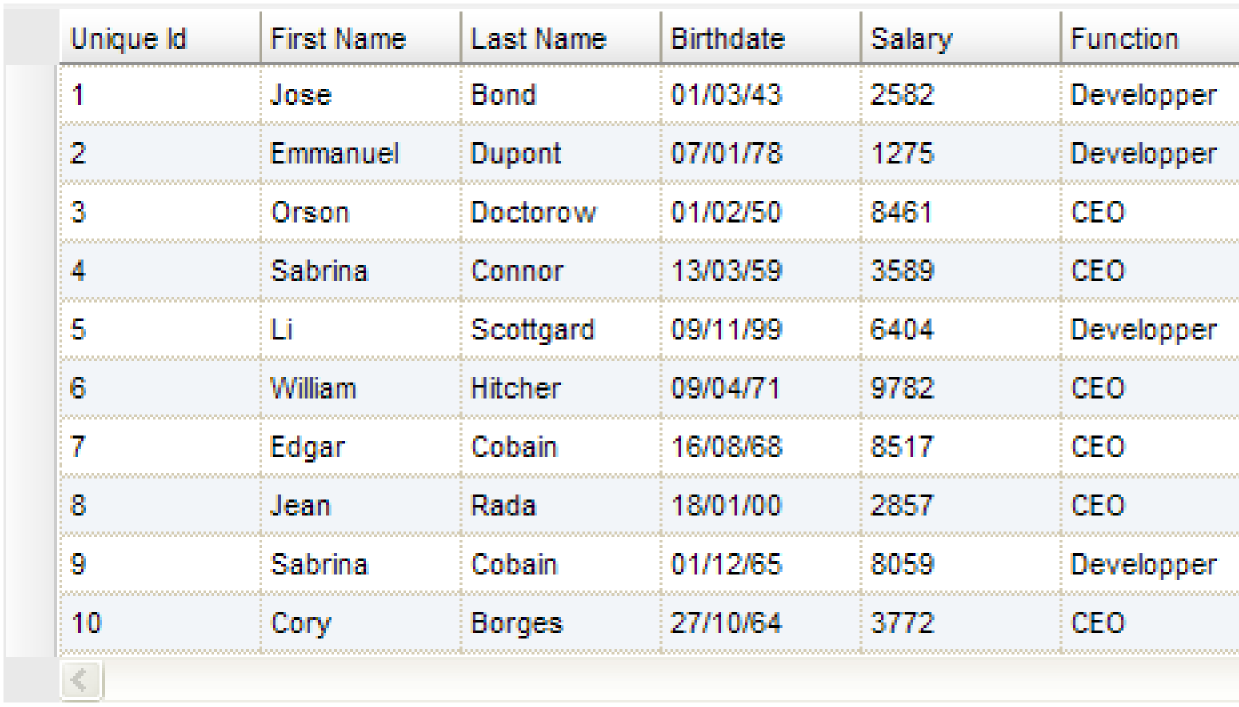
2.3.1.1.6.1 How to filter a grid

To add a filter to the grid , just use :

```
public void addFilter(String fieldName , String criterium
    )
```

where field name is the attribute on which we want to perform filtering, and criterium is the value that attribute has to match.

For example , it we filter the following grid :

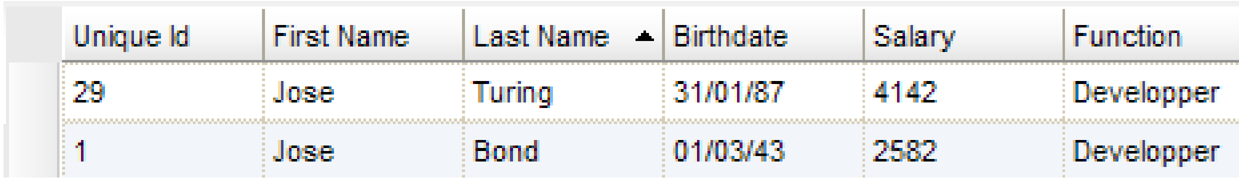


Unique Id	First Name	Last Name	Birthdate	Salary	Function
1	Jose	Bond	01/03/43	2582	Developper
2	Emmanuel	Dupont	07/01/78	1275	Developper
3	Orson	Doctorow	01/02/50	8461	CEO
4	Sabrina	Connor	13/03/59	3589	CEO
5	Li	Scottgard	09/11/99	6404	Developper
6	William	Hitcher	09/04/71	9782	CEO
7	Edgar	Cobain	16/08/68	8517	CEO
8	Jean	Rada	18/01/00	2857	CEO
9	Sabrina	Cobain	01/12/65	8059	Developper
10	Cory	Borges	27/10/64	3772	CEO

with the following filter:

```
grid.addFilter("firstName","Jose");
```

It will result in :



Unique Id	First Name	Last Name	Birthdate	Salary	Function
29	Jose	Turing	31/01/87	4142	Developper
1	Jose	Bond	01/03/43	2582	Developper

Then , if we want to remove that filter, we can just use :

```
public void removeFilter(String fieldName)
```

To remove the previous filter, we would use :

```
grid.removeFilter("firstName");
```

2.3.1.1.6.2 Filtering API

addFilter(String,Object): Adds a filter on given attribute which would be compared to given object

removeFilter(String): Removes a filter on given attribute

2.3.1.1.7 Common issues while manipulating columns : introducing the MVC model.

This section is intended to demonstrate some problems with the simplest column manipulation API, and to introduce to the solution.

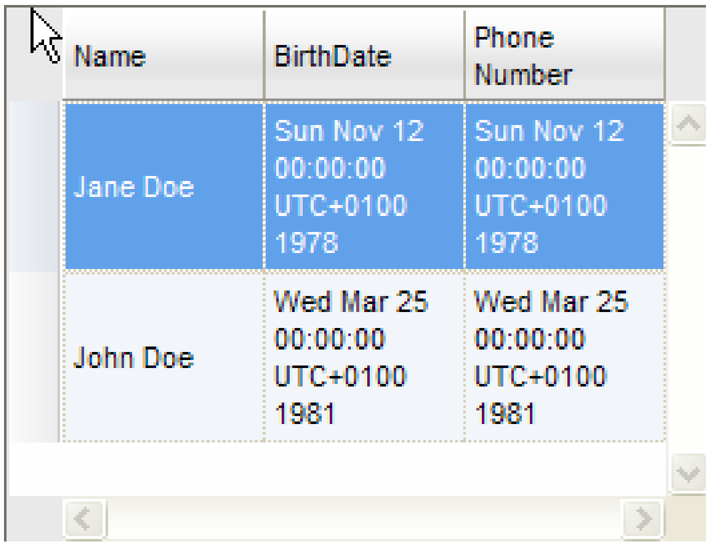
We will start from the previous column manipulation example :

```
Button button = new Button("Toggle␣Phone␣Column");
button.addClickListener(new ClickListener(){
        private boolean isShown = true;
        public void onClick(Widget sender){
                if(isShown){
                        grid.removeColumn(1);
                        grid.updateView();
                        isShown = false;
                }else{
                        grid.insertColumn("Phone␣Number"
                            , 1);
                        grid.updateView();
                        isShown = true;
                }
        }
});
RootPanel.get().add(button);
```

Suppose now that we want to insert it AFTER the Birthdate when the user clicks on the button again ... We would probably be tempted to use the following snippet :

```
}else{
        grid.insertColumn("Phone␣Number" , 2);
        grid.updateView();
        isShown = true;
}
```

In this case, after the user hit the button a second time , the produced result would be :



Name	BirthDate	Phone Number
Jane Doe	Sun Nov 12 00:00:00 UTC+0100 1978	Sun Nov 12 00:00:00 UTC+0100 1978
John Doe	Wed Mar 25 00:00:00 UTC+0100 1981	Wed Mar 25 00:00:00 UTC+0100 1981

Why ? That's because the only logical link between your data and your columns is their order ... Because we specified an index of 2 for our column, it takes for it the data at index 2 from our rows, which is precisely the same as the birthdate column. We can also notice that the data from birthdate column has not changed : that's because when it was created, it was told to use the data from field "2".

If we want to solve this problem, we just can use the following method :

```
grid.insertColumn("Phone␣Number" , "1", 2);
grid.updateView();
isShown = true;
```

This method insert a column taking data from the field "1" , which was implicitly generated when you added your columns. To make it a bit cleaner , we will rewrite the base code for our grid :

```
Grid grid = new Grid();
grid.addColumn("Name" , "name");
grid.addColumn("Phone␣Number" , "number");
grid.addColumn("BirthDate" , "birthdate");
Object[] row1 = {"John␣Doe" , "0123456789" , new Date("
    03/25/1981") };
Object[] row2 = {"Jane␣Doe" , "9876543210" , new Date("
    11/12/1978") };
grid.addRow(row1);
grid.addRow(row2);
grid.setWidth("300px");
grid.setHeight("200px");
RootPanel.get().add(grid);
Button button = new Button("Toggle␣Phone␣Column");
```

```
button.addClickListener(new ClickListener(){
        private boolean isShown = true;
        public void onClick(Widget sender) {
                if(isShown){
                grid.removeColumn(1);
                grid.updateView();
                isShown = false;
                }else{
                grid.insertColumn("Phone_Number" , "
                   number" , 2);
                grid.updateView();
                isShown = true;
                }
        }
});
RootPanel.get().add(button);
```

Notice the what has changed : we assigned a field name to the columns when we created them. Later, when we added our rows, the grid assigned field names to the various objects in the row according to the column's order. Then , we can assign data to a column by its field name. That shows the limits of the simple grid utilisation ... Please see Grid MVC section for a more powerful way of assigning columns and rows.

2.3.1.2 Cell editing and formatting

2.3.1.2.1 Formatting

As we have seen earlier, some data need to be formatted for display. Tatami grid package offers only a Date formatter and a Currency formatter , but more are expected to come. You can also implement your own, by implementing the Formatter interface.

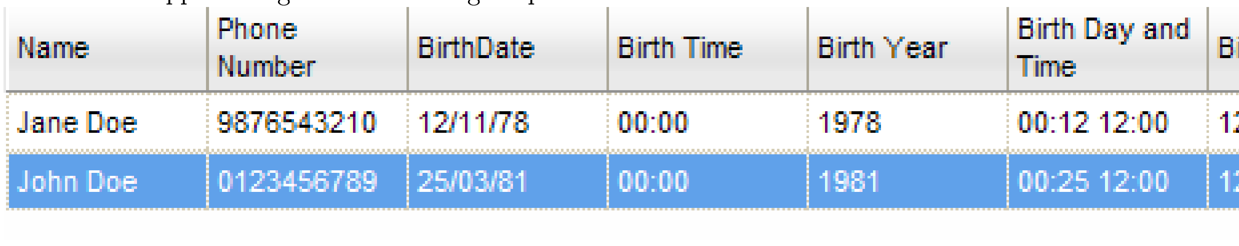
2.3.1.2.1.1 How to use the date formatter

Example for a Date formatter, using the example from previous section :

```
grid.addColumn("Name" , "name");
grid.addColumn("Phone_Number" , "number");
grid.addColumn("BirthDate" , "date" , new DateFormatter(
   DateFormatter.displayDateOnly));
grid.addColumn("Birth_Time" , "date" , new DateFormatter(
   DateFormatter.displayTimeOnly));
grid.addColumn("Birth_Year" , "date" , new DateFormatter(
   "yyyy",null));
grid.addColumn("Birth_Day_and_Time" , "date" , new
   DateFormatter("mm:dd","hh:mm"));
grid.addColumn("Birth_Time" , "date" , new DateFormatter(
   null , "hh"));
```

```
Object[] row1 = {"John␣Doe" , "0123456789" , new Date("
    03/25/1981") };
Object[] row2 = {"Jane␣Doe" , "9876543210" , new Date("
    11/12/1978")};
```

The above snippet will give the following output :



Name	Phone Number	BirthDate	Birth Time	Birth Year	Birth Day and Time	Bi
Jane Doe	9876543210	12/11/78	00:00	1978	00:12 12:00	12
John Doe	0123456789	25/03/81	00:00	1981	00:25 12:00	12

2.3.1.2.1.2 Date Formatter constructors:

DateFormatter(selector): Either DateFormatter.displayTimeAndDate, DateFormatter.displayTimeOnly , or DateFormatter.displayDateOnly.According to the constant, will display in the default locale time representation the date and the time, the date only , or the time only.

DateFormatter(datePattern , timePattern): Create a date formatter using the specified datePattern and timePattern. Date and Time pattern are specified as in ISO. formatter =**new** DateFormatter("dd/mm/yyyy", **null**); is equivalent to formatter=**new** DateFormatter(DateFormatter.displayDateOnly);formatter. setDatePattern("dd/mm/yyyy");

2.3.1.2.2 Editing

Tatami offers the same grid edition facility as Dojo Grid. That is, you can easily use one of the dojo editor to provide your grid in-place editing feature. User toggle editing by double-clicking on a grid cell.

All editors implement the GridEditor interface.

2.3.1.2.2.1 Available editors :

TextEditor: a simple text field

DateEditor: a drop-down calendar

NumberSpinnerEditor: a simple spinner

ComboBoxEditor: a drop-down list

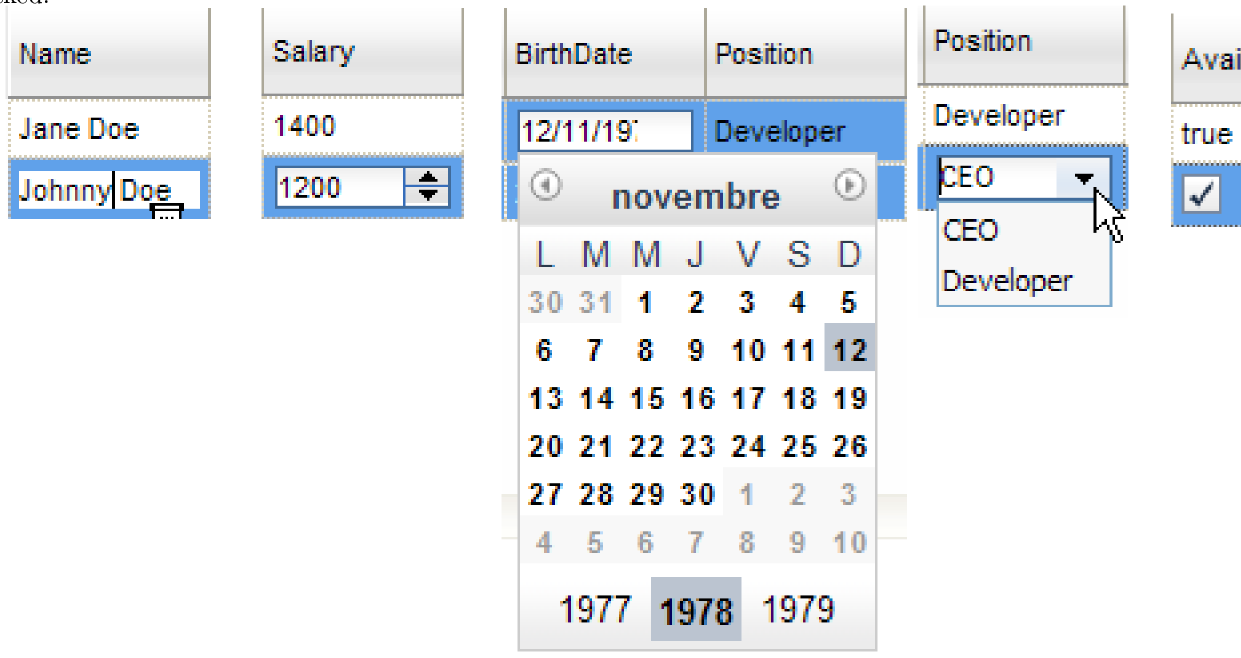
CheckBoxEditor: a checkbox for boolean values

Please refer to Javadoc for specific information about an editor. For the NumberSpinnerEditor, please refer to The NumberSpinner component section to see how to create it.

Sample :

```
grid.addColumn("Name" , "name" , new TextEditor());
grid.addColumn("Phone␣Number" , "number" , new TextEditor
    ());
grid.addColumn("Salary", "salary", new
    NumberSpinnerEditor( 0 , 1500 , 100));
grid.addColumn("BirthDate" , "date" , new DateEditor() ,
    new DateFormatter(DateFormatter.displayDateOnly));
String[] positions = {"CEO" , "Developer"};
grid.addColumn("Position" , "position" , new
    ComboBoxEditor(positions));
grid.addColumn("Available" , "available" , new
    CheckBoxEditor());
Object[] row1 = {"John␣Doe" , "0123456789" , new Integer(
    "500") , new Date("03/25/1981") , "CEO" , Boolean.
    FALSE};
Object[] row2 = {"Jane␣Doe" , "9876543210" , new Integer(
    "1200") , new Date("11/12/1978") , "Developer" ,
    Boolean.TRUE};
```

This will give the following results, based on the cell on which you double-clicked:



2.3.1.3 Grid events

The grid accepts GridListeners , which are notified when an event occurs on the grid. To observe those events, just implements the following interface :

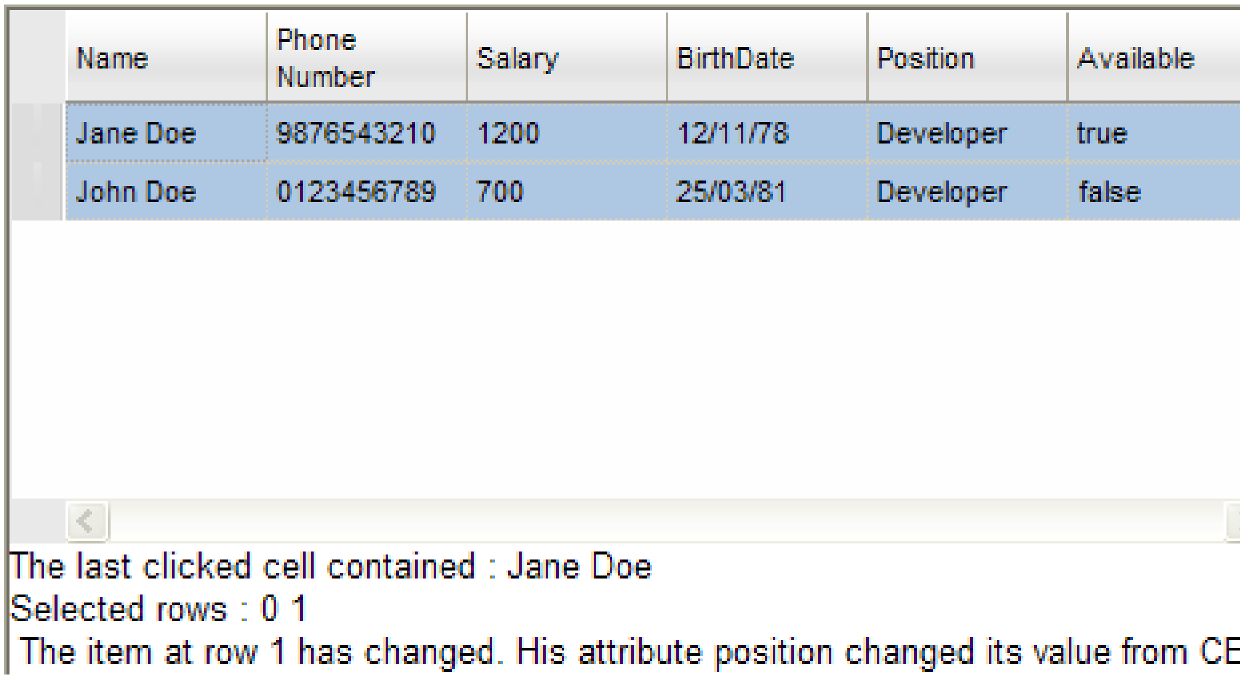
```
public interface GridListener {
        public void onCellClick(Grid grid , int rowIndex
            , int colIndex String colName);
        public void onSelectionChanged(Grid grid);
        public void onDataChange(Grid grid , Item
            itemWhichChanged, String attributeName, Object
             oldValue, Object newValue);
}
```

- onCellClick event will be fired each time a cell is clicked.
- onSelectionChange event will be fired each time the user selects (or unselects) a row, via a row click , or the classic SHIFT + click , CTRL + click selection combinations.
- onDataChange will be fired each time an item in the underlying store has changed.

For example, let's write a simple grid observer to see how we can handle those events:

```
class myGridListener implements GridListener{
        //The html which content will be changed each
            time a grid event occurs
        private HTML html;
        //What should be updated when a cell is clicked
            on.
        private String lastClickedCellContent;
        //What should be updated when the rows selection
            change
        private String selectedIndexesCellContent;
        //What should be displayed when an item's value
            changes
        private String itemChangedContent;
        //The constructor takes an "HTML" object in which
             it will write the cell content.
        public myGridListener(HTML html){
                this.html = html;
        }
        //Updates the html object content
        private void refresh(){
                this.html.setText(lastClickedCellContent
                    + "\n" + selectedIndexesCellContent +"
                    \n" + itemChangedContent);
        }
        //Updates the displayed text to show the content
            of the cell which has been clicked
        public void onCellClick(Grid grid, int rowIndex ,
             int colIndex , String colFieldName){
                Object cellContent = grid.getItemFromRow(
                    rowIndex).getValues(colFieldName);
                lastClickedCellContent ="The␣last␣clicked
                    ␣cell␣contained␣:␣" + cellContent.
                    toString();
                this.refresh();
        }
        //Updates the displayed text to show the selected
             rows indexes
        public void onSelectionChanged(Grid grid){
                int[] indexes = grid.getSelectedIndexes()
                    ;
                selectedIndexesCellContent = "Selected␣
                    rows␣:␣";
                for(int i = 0; i < indexes.length; i ++ )
                    {
                        selectedIndexesCellContent += i +
                            "␣";
                }
                this.refresh();
        }
        //Updates the displayed text to show how the data
             has changed in the grid
        public void onDataChange(Grid grid , Item
            itemWhichChanged, String attributeName, Object
             oldValue, Object newValue){
                itemChangedContent = "␣The␣item␣at␣row␣"
                    + grid.getRowFromItem(itemWhichChanged
                    ) + "␣has␣changed.␣His␣attribute␣" +
                    attributeName + "␣changed␣its␣value␣
                    from␣" + oldValue.toString() + "␣to␣"+
                     newValue.toString();
                this.refresh();
        }
}
HTML html = new HTML();
grid.addGridListener(new myGridListener(html));
RootPanel.get().add(html);
```

After modifying the “John Doe” row , and selecting both rows (with CTRL + click) :



	Name	Phone Number	Salary	BirthDate	Position	Available
	Jane Doe	9876543210	1200	12/11/78	Developer	true
	John Doe	0123456789	700	25/03/81	Developer	false

The last clicked cell contained : Jane Doe
Selected rows : 0 1
The item at row 1 has changed. His attribute position changed its value from CE

2.3.1.4 Advanced Grid usage

2.3.1.4.1 Datastore usage

2.3.1.4.1.1 Introduction

Tatami grid, just as dojo grid, relies on a data store. This data store is used to access the various items represented in the grid, and perform local operation like sorting , filtering . . .

Whenever anything changes in the datastore, the grid updates itself according to those needs.

There are two principles you should be aware of:

- a row in the grid represents an item in the datastore
- an item always has a unique identifier

You should see the DataStore Integration section on how to use it.

2.3.1.4.1.2 Store-related Grid API

getStore(): Gets the grid datastore

setStore(GridDataStore store): Sets the grid datastore

The following methods add an item to the datastore, either directly or creating it from an array of object, positioning the item attributes accordingly to the declared columns fields:

addRow(Item)

addRow(Item,int)

addRow(Object[])

addRow(Object[],int)

The following methods remove an item from the datastore:

removeRow(int)

removeRow(Item)

removeSelectedRows()

2.3.1.4.2 GridLayout, GridView and Cells

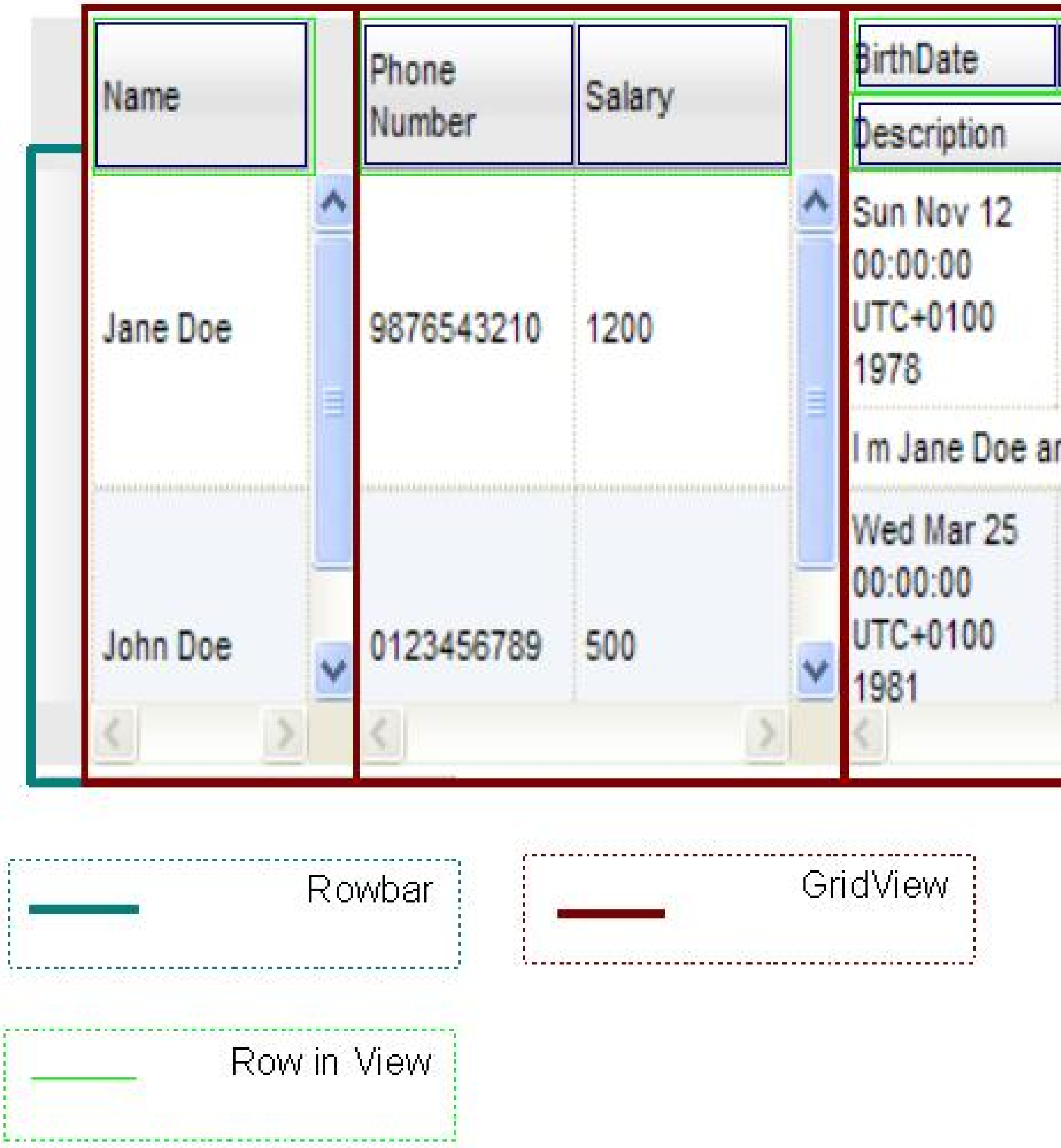
In the Adding / removing columns section , we saw how to create/modify a grid layout.

We also saw how this method was limited when we want to add/remove columns from a grid.

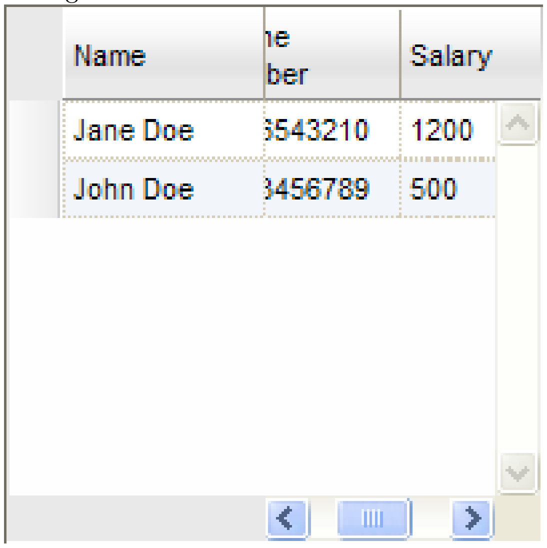
Tatami grid offers the same flexibility as dojo's one to define how it should be layed out.

2.3.1.4.2.1 How the grid layout is defined

The schema below shows what objects are used to define a grid layout.


Name
Phone Number
Salary
BirthDate
Description
Jane Doe
9876543210
1200
Sun Nov 12 00:00:00 UTC+0100 1978
I m Jane Doe ar
John Doe
0123456789
500
Wed Mar 25 00:00:00 UTC+0100 1981
Rowbar
GridView
Row in View

A layout is composed of one or several views (in red). Those views can be denifed as a set of columns. Each view can have its specific size, and its specific scrollbar(s). Of course, the vertical scrollbars are synchronized, since a grid row is defined among all views. Defining multiple views allows to have a fixed width view (not scrollable) while another view is scrollable. That can be useful to produce grid like this :



Name	ıe ber	Salary
Jane Doe	;543210	1200
John Doe	3456789	500

The first column (name) is in its own (not scrollable) view, while other columns are grouped in another view. Thus, you can scroll through the columns while keeping the "Name" column in sight.

A view itself is composed of one or several rows of cells (in light green). Defining multiple cell-rows for a view is useful when you want to create sub-rows in the grid (look at the description field in the layout schema).

Those rows are composed of cells (in blue). The cell is the atomic entity of the layout. A cell is (at least) defined with the field used to fill its values and its name. If no field is provided, it will use the "value" setting to fill the cell.

Both cell and views have settings you can use to define their appearance and behavior.

2.3.1.4.2.2 GridLayout usage

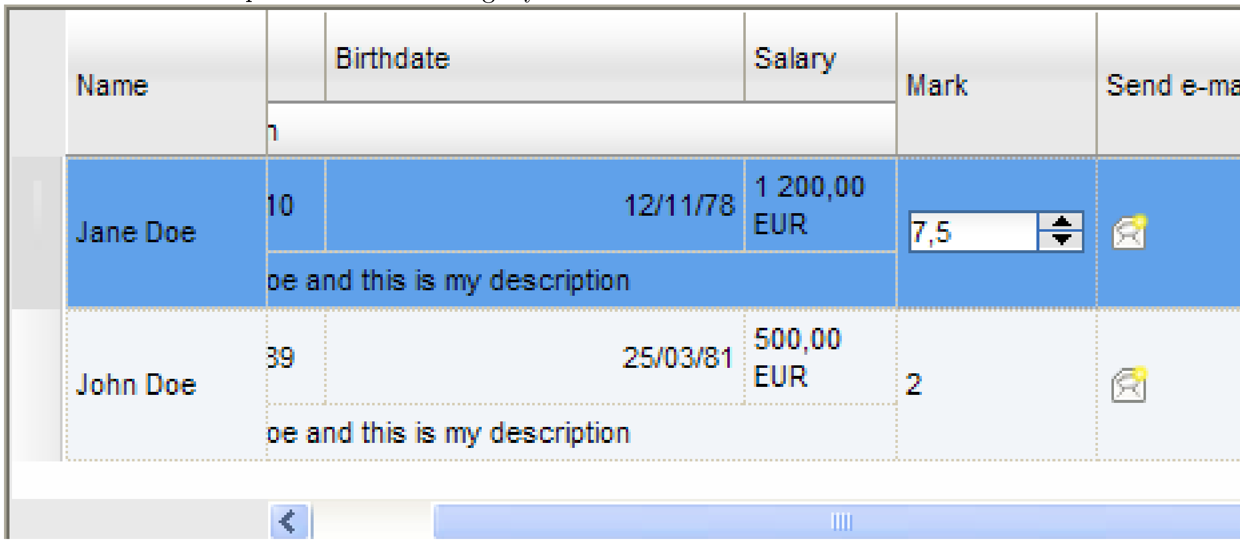
Below is a simple sample showing how to create a GridLayout.

```
GridLayout layout = new GridLayout();
//Creating the first (left) view
//This view contains only one cell, which label is "Name
    ".
//and wich content is the "myname" field from the data
    store objects.
//This cell is also not resizable.
//It also has a fixed width, and is not scrollable
```

```
GridView view = new GridView();
Cell cell = new Cell("myname" , "Name");
cell.setIsNotResizable(Boolean.TRUE);
view.addCellToLastRow(cell);
view.setScrollable(false);
view.setWidth("100px");
//Creating the second (right) view
//This view has 2 rows GridView
view2 = new GridView();
//We add a cell to the (current) last row
view2.addCellToLastRow(new Cell("number" ,"Phone␣Number")
    );
//We define a cell containing the salary attribute, and
    add it to the row
//This cell is editable (with a number spinner)
//and is formatted as a currency
Cell salaryCell = new Cell("salary" ,"Salary");
salaryCell.setEditor(new NumberSpinnerEditor());
salaryCell.setFormatter(new CurrencyFormatter("EUR"));
salaryCell.setWidth("50px");
view2.addCellToLastRow(salaryCell);
//We define a cell containt the date attribute, and add
    it to the row.
//This cell is also editable (with a DateEditor) and is
    formatted as a date.
//Please note that we define specific style settings for
    this cell
Cell birthDateCell = new Cell("date" ,"Birthdate");
birthDateCell.setEditor(new DateEditor());
birthDateCell.setFormatter(new DateFormatter(
    DateFormatter.displayDateOnly));
birthDateCell.setWidth("150px");
birthDateCell.setCellStyles("text-align␣:␣right;");
view2.addCellToLastRow(birthDateCell);
//We define a cell containing the description
//This cell will be added to a second row, and will have
    a colspan of 3
Cell descriptionCell = new Cell("description" , "
    Description");
descriptionCell.setColSpan(new Integer(3));
view2.addCellToRow(descriptionCell , 1);
//We define a cell containing the appreciation,and add it
     to the first row.
//This cell has a rowspan of 2
Cell appreciationCell = new Cell("appreciation" , "Mark")
    ;
```

```
appreciationCell.setEditor(new NumberSpinnerEditor(0f ,
    10f, 0.5f));
appreciationCell.setRowSpan(new Integer(2));
view2.addCellToRow(appreciationCell,0);
//We define a cell containing an constant value (an image
    ) and add it just like the previous one
Cell imageCell = new Cell("Send e-mail");
imageCell.setDefaultValue("<img src='./mail-message-new.
    png' alt='send mail'></img>");
imageCell.setRowSpan(new Integer(2));
view2.addCellToRow(imageCell,0);
view2.setWidth("500px");
//We add the views to the layout
layout.addView(view);
layout.addView(view2);
grid.setLayout(layout);
grid.setAutoWidth(false);
grid.setWidth("500px");
grid.updateView();
```

The above code will produce the following layout :



Name		Birthdate	Salary	Mark	Send e-ma
	n				
Jane Doe	10	12/11/78	1 200,00 EUR	7,5	
	oe and this is my description				
John Doe	39	25/03/81	500,00 EUR	2	
	oe and this is my description				

2.3.1.4.2.3 GridLayout , GridView and Cell APIs

GridLayout API :

addCellToLastView(Cell): Adds the given cell to the last row of the last view (useful when there is an only view)

addCellToLastView(Cell, int): Adds the given cell at the specified index of the last row of the last view (useful when there is an only view)

addView(GridView): Adds a view to the layout

addView(int,GridView): Adds a view to the layout at the given index

clear(): Removes all views from the layout

getCellAmongAllViews(int): Gets a cell by its index among all views. That is, if the layout contains 2 views of 4 cells each, getCellAmongAllViews(4) will return the cell at index 0 from the second view.

getLastView(): Returns the last view from the layout

getNbCells(): Returns the total number of cells defined in the layout

getView(int): Returns the view at specified index

removeCellAmongAllViews(Cell): Removes a cell wherever it is in the views

removeView(GridView): Removes a view from the layout

removeView(int): Removes the view at specified index

GridView API:

addCellToLastRow(Cell): Adds the given cell to the last row

addCellToLastRow(Cell,int): Adds the given cell at the specified index of the last row

addCellToRow(Cell,int row):Adds the given cell to the given row. If this row doesn't exist, it will be created.

addCellToRow(Cell,int row,int index): Adds the given cell at the given index of the given row. If this row doesn't exist, it will be created.

clear(): Removes all rows from the view

getCell(int): Gets a cell by its index among all rows. That is, if the view contains 2 rows of 4 cells each, getCell (4) will return the cell at index 0 from the second view.

getCellFromRow(int row, int index): Returns the cell at the specified index from the specified row

getNbCells(): Returns the total number of cells defined in the view

getRow(int): Returns the row at specified index (a row is a List of cells)

getWidth(): Get the specified width in css units. If null, the view will be as wide as its containing cells.

isScrollable(): Returns true if this view shows scrollbars, false othervwise

removeCell(Cell): Removes the given cell

removeCell(Cell, int): Removes the given cell from the given row

removeCell(int): Removes the cell at given index (see getCell(int))

removeCell(int row, int index): Removes the cell at given index from the given row.

setScrollable(boolean): Sets if the view should display scrollbars

setWidth(String): Sets the view width (in standard css units)

Cell API:

The following methods respectevely get/set the css classes or styles applied to a Cell:

getCellClasses()

setCellClasses(String)

getCellStyles()

setCellStyles(String)

The following methods respectevely get/set the css classes or styles applied to the whole column (data and header):

getClasses()

setClasses(String)

getStyles()

setStyles(String)

The following methods respectively get/set the colspan (ie, the number of columns this cell should occupy)

getColSpan():

setColSpan(Integer)

The following methods respectively get/set the rowspan(ie, the number of rows this cell should occupy)

getRowSpan()

setRowSpan(Integer)

The following methods respectively get/set the cell default value (ie , the value to be used when no field is assigned or when the underlying item doesn't have the given field)

getDefaultValue()

setDefaultValue()

The following methods respectively get/set this cell editor. If the editor is different from null, the cell will be editable.

getEditor()

setEditor(GridEditor)

The following methods respectively get/set the field from which this cell should fill its content.

getField()

setField(String)

The following methods respectively get/set whether this cell should not be resizable. If true, the cell will not be resizable. If false , it will be resizable. Default to true.

getIsNotResizable()

setIsNotResizable(Boolean)

The following methods respectively get/set this cell name (ie , what will be displayed on the column header)

getName()

setName(String)

The following methods respectively get/set the cell's width in standard css units :

getWidth()

setWidth(String)

2.3.2 Tree

Like the grid, the tree uses a datastore to store its items

2.3.2.1 Getting Started

To use the tree, you need to import:

```
import com.objetdirect.tatami.client.tree.Tree;
```

The tree must be initialized with a root item, either directly in the constructor, or with the setRootItem method:

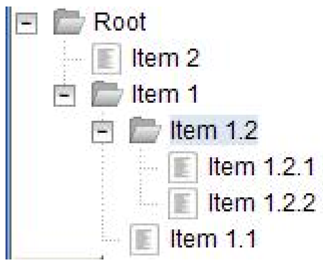
```
Tree tree = new Tree();
tree.setRootItem(new Item("Item1","Item1"));
//Or:
tree = new Tree(new Item("Item1","Item1"));
```

Two other constructors are available if you do not want to use the default datastore:

```
public Tree(AbstractDataStore);
public Tree(Item, AbstractDataStore);
```

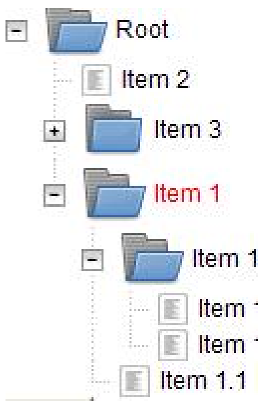
Now, we will see how to populate such a tree:

```
Tree tree = new Tree(new Item("Root","Root"));
Item item1 = new Item("item1" , "Item␣1");
Item item11 = new Item("item1.1" , "Item␣1.1");
Item item12 = new Item("item1.2" , "Item␣1.2");
item1.addChild(item11);
item1.addChild(item12);
item12.addChild(new Item("item1.2.1" , "Item␣1.2.1"));
item12.addChild(new Item("item1.2.2" , "Item␣1.2.2"));
tree.getRootItem().addChild(item1);
tree.getRootItem().addChild(item2);
RootPanel.get().add(tree);
```



2.3.2.2 Customizing the tree display

The tree offers many customization options.



As you can see on the above picture, you can customize the icons used for items, as well as some css styling on the labels.

Customizing the icons can be done at two levels:

- setting default icons for the whole tree. This can be achieved by using the following methods. Those methods will take a string as an argument, which will be used as a css class to apply to the icon node.

setDefaultFolderClosedClass(String): sets the class applied to a closed folder icon

setDefaultFolderOpenClass(String): sets the class applied to an opened folder icon

setDefaultLeafClass(String): sets the class applied to a tree leaf icon

The following example show how the icon classes were customized for the above tree:

```
tree.setDefaultFolderClosedClass("myTreeClosed");
tree.setDefaultFolderOpenClass("myTreeOpened");
```

With the myTreeClosed and myTreeOpened class defined like this:

```
.tundra .myTreeOpened {
        background-image: url('folder-open.png');
        background-position: top;
        top: 1px;
        height: 32px !important;
        width: 32px !important;
}
.tundra .myTreeClosed {
        background-image: url('folder.png');
        background-position: top;
        top: 1px;
        height: 32px !important;
        width: 32px !important;
}
```

- setting different icons for some items. This can be achieved by setting the Tree.folderClosedClassAttribute, Tree.folderOpenedClassAttribute or Tree.leafClassAttribute.

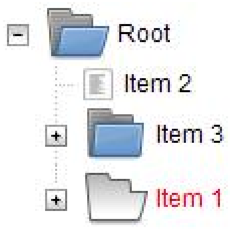
  Example:

  ```
  item1.setValue(Tree.folderClosedClassAttribute,"item1
     ");
  ```

  With the item1 css class defined as:

  ```
  .item1{
          background-image:
          url('folder-visiting.png');
          background-position: top;
          top: 1px;
          height: 32px !important;
          width: 32px !important;
  }
  ```

  Our item item1 will then have its own class:

You can also customize the css styles applied to the label. As you probably noticed, item1 is the only one item which label is red-colored.

It has been achieved by using the following method:

```
item1.setValue(Tree.labelClassAttribute,"myLabelClass");
```

With the corresponding css class:

```
.myLabelClass{
        color: red;
}
```

2.3.2.3 Tree events

The Tree accepts tree listeners. A Tree Listener must implement the following interface:

```
package com.objetdirect.tatami.client.tree;
import com.objetdirect.tatami.client.data.Item;
public interface TreeListener {
        public void onClick(Item item);
        public void onOpen(Item item);
        public void onClose(Item item);
        public void onDblClick(Item item);
}
```

Then, you can or remove tree listeners to a tree with the following methods:

```
addTreeListener(TreeListener)
removeTreeListener(TreeListener)
```

2.3.2.4 Example

The following example illustrates the tree customization and tree listeners concepts.

```
FlowPanel panel = new FlowPanel();
Tree tree = new Tree(new Item("Root","Root"));
Item item1 = new Item("item1" , "Item␣1");
Item item11 = new Item("item1.1" , "Item␣1.1");
Item item12 = new Item("item1.2" , "Item␣1.2");
item1.addChild(item11);
item1.addChild(item12);
item12.addChild(new Item("item1.2.1" , "Item␣1.2.1"));
item12.addChild(new Item("item1.2.2" , "Item␣1.2.2"));
item1.setValue(Tree.labelClassAttribute,"myLabelClass");
item1.setValue(Tree.folderClosedClassAttribute,"item1");
Item item2 = new Item("item2" , "Item␣2");
tree.setDefaultFolderClosedClass("myTreeClosed");
tree.setDefaultFolderOpenClass("myTreeOpened");
tree.getRootItem().addChild(item1);
tree.getRootItem().addChild(item2);
final HTML openValue = new HTML("Opened␣:");
final HTML closedValue = new HTML("Closed␣:");
final HTML onClickValue = new HTML("Clicked␣:");
final HTML onDblClickValue = new HTML("DblClicked␣:");
panel.add(tree);
tree.addTreeListener(new TreeListener() {
        public void onOpen(Item item) {
                openValue.setHTML("Opened␣:" + item.getId
                    ());
        }
        public void onClose(Item item) {
                closedValue.setHTML("Closed␣:" + item.
                    getId());
        }
        public void onClick(Item item) {
                onClickValue.setHTML("Clicked␣:" + item.
                    getId());
        }
        public void onDblClick(Item item) {
                onDblClickValue.setHTML("DblClicked␣:" +
                    item.getId());
        }
});
panel.add(onClickValue);
panel.add(onDblClickValue);
panel.add(closedValue);
panel.add(openValue);
```

Chapter 3

Datastore Integration

3.1 DataStore and Items

The data store provides a convenient way to access data, locally or remotely. It is used in the grid and tree widgets to store their content.

A datastore contains Items

Items have an id attribute, which identifies them uniquely in the datastore.

To construct a DefaultDataStore , we simply have to the following constructor:

```
DefaultDataStore()
```

3.2 Datastore usage

The following snippet shows how to build populate a DefaultDataStore :

```
DefaultDataStore store = new DefaultDataStore();
Item itemJohn = new Item();
itemJohn.setId("14563" );
itemJohn.setValue("myname", "John␣Doe" );
itemJohn.setValue("number", "0123456789");
itemJohn.setValue("salary", new Integer("500"));
itemJohn.setValue("date", new Date("03/25/1981"));
itemJohn.setValue("position", "CEO");
itemJohn.setValue("available", Boolean.FALSE);
itemJohn.setValue("mySecret", "!!!");
itemJohn.setValue("description", "I␣m␣John␣Doe␣and␣this␣
    is␣my␣description" );
itemJohn.setValue("appreciation", "Good" );
itemJohn.setValue("married", true);
store.add(itemJohn);
Item itemJane = new Item();
```

```
itemJane.setId("87321" );
itemJane.setValue("number", "9876543210");
itemJane.setValue("salary", new Integer("1200"));
itemJane.setValue("date", new Date("11/12/1978"));
itemJane.setValue("position", "Developer");
itemJane.setValue("available", Boolean.TRUE);
itemJane.setValue("myname", "Jane Doe" );
itemJane.setValue("description", "I m Jane Doe and this 
    is my description" );
itemJane.setValue("appreciation", "Awesome" );
itemJane.setValue("mySecret", "!!!");
itemJane.setValue("married", false );
store.add(itemJane);
```

This store can then be used by a grid to populate itself..

Below is the datastore item manipulation API, which conforms to Dojo data APIs.

addItem(Item): Adds an item to the datastore. If an item already has the same id, it is overridden.

removeItem(Item): Removes an item from the data store

getItemByIdentity(Object id): Returns the item stored with the given id, null if no item within the store has the given attribute

isItem(Object): true if item is an instance of Item and the store contains this item, false otherwise

loadItem(Item): Loads the specified item if it has not already been. A "loaded" item should have all its attributes loaded, while a not-yet-loaded item can be a "stub" used for remote lazy loading.

isItemLoaded(Item): Returns true if the given Item is in the store AND has been fully loaded.In the DefaultDataStore implementation, always returns true since we always add "fully loaded" items (no stubs, or remote lazy loading). (See How to implement your own data store)

searchItemsByAttributes(String attr,Object value): Returns a list of items which given attribute attr has the given value.

size(): Returns the number of items in the store.

3.3 Fetching items

The core method of the datastore is the "fetch" function. The fetch function performs a query , represented by the "Request" object.

The fetch method signature :

```
public void fetch(Request request)
```

This method performs a Request , and notifies the FetchListeners during the whole fetch process.

Below stands the Fetch Listener interface. The grid itself implements the FetchListener interface, but we can implement our own fetch listener to be aware of all the Fetch Process.

```
public interface FetchListener {
        public void onComplete(FetchEventSource source,
            List items , Request request );
        public void onBegin(FetchEventSource source ,int
            size , Request request);
        public void onItem(FetchEventSource source ,Item
            item);
        public void onError(FetchEventSource source);
}
```

The main method is onComplete : it is called once the fetch is done. Items is a List containing all the fetched items, while request is the (possibly modified) request performed.

onBegin is called before the actual fetch. “Size” is the total number of items matching the request (or the page size when using paging), while request is the (possibly modified) request performed.

onItem is called each time an Item has been fetched, that is , it will be present in the list passed to onComplete.

onError is called when an error occurs during the fetch.

The request object owns a certain amount of parameters, used to search items in the store.

These parameters are :

nbItemToReturn: Maximum number of item to return. The number of item actually returned can be lesser than nbItemToReturn.

startItemNumber: Specifies the item number from which the item should be returned. In particular, this is used to load only the number of items that fits on screen. (for example, this is what dojo specified to load the items as the user scrolls the grid)

query: Map of key / attributes pair describing the search criterias.

sortFields: A list of SortField. A SortField is a simple object containing an attribute name and a Boolean indicating wether the sort should be performed in ascending or descending order.

3.4 DataStore events

Grid and datastores are connected via DataStore events. There are many events fired by the data store. In the previous section , we already saw how the fetch events are sourced and consumed.

Two other interfaces are defined to listen for DataStore events. Those are :

```
public interface LoadItemListener { public void onLoad(
    Item item); }
```

This interface defines a load item listener : the onLoad item will be called each time the datastore "loadItem" is called on an (not yet) loaded item. In the current implementation, the items are always already loaded , since we use no "stub" item.

```
public interface DatumChangeListener {
         public void onDataChange(Item item , String
             attributeName , Object oldValue , Object
             newValue);
         public void onNew(Item item);
         public void onDelete(Item item);
}
```

This interface defines a listener for data changes. For example, The grid itself implements this interface. onDataChange is called each time an item's attribute has been modified. onNew is called each time an item has been added to the data store. onDelete is called each time an item has been deleted from the data store.

addDatumChangeListener(DatumChangeListener): Adds a DatumChange-Listener

removeDatumChangeListener(DatumChangeListener): Removes a DatumChangeListener

addFetchListener(FetchListener): Adds a FetchListener

removeFetchListener(FetchListener): Removes a FetchListener

addLoadItemListener(LoadItemListener): Adds a LoadItemListener

removeLoadItemListener(LoadItemListener): Removes a LoadItemListener

3.5 How to implement your own data store

3.5.1 Principles

Why implement your own data store ? The DefaultDataStore implementation is a purely local one. Of course , you can feed it with remotely aquired data, but if you want to only load items on demand, you would like to implement your own. We think it is the principal reason why you would like to implement one, but we are sure you will find plenty others !

To implement your own DataStore , you only have to subclass the Default-DataStore, and override the particular method you would like to implement yourself. The DefaultDataStore itself extends the AbstractDataStore class. You should respect the following steps:

-implement the fetch method to retrieve Items, either locally in the datastore or from another data source

-implement the loadItem method, accordingly to your need.

3.5.2 Some examples

Suppose you would like to connect your grid directly to a relational database. Suppose also that this database is exposed via a REST-type web-service. The only thing you would have to override would be the fetch method. Please see the Fetching items section to understand how the items should be fetched.

3.5.2.1 DefaultDataStore implementation

The default datastore implementation stores its items locally in a Map. Each time a request is performed to the datastore, items will be returned directly from the datastore.

First of all, the items are filtered and sorted against the query:

```
public void fetch(Request request) {
        //Saves the request to keep filtering and sorting
            parameter for later use
        // (for example, to fetch multiple pages)
            lastRequest = request;
        //Gets the items matching the query , and sorts
            them according to the
        //request
        List<?> itemsSortedAndMatchingQuery = this.
            executeQuery(items.values() , request);
```

Then, we notify the fetch listeners that the fetch process has begun.This is done with the “notifyBeginFetchListeners” request.

It takes two arguments : the total number of items matching the query, and the request itself.

```
        int size = itemsSortedAndMatchingQuery.size();
        // We notify the fetch listeners that the request
            is being performed
        notifyBeginFetchListeners(size , request);
        List<Item> result = new ArrayList<Item>();
```

We can process the actual fetched items, and load them accordingly. Here, the load method does nothing, since it is a purely local datastore. Items are supposed to be fully loaded as soon as they are added to the datastore. The load item is supposed to load the items from an external datasource if you want to perform lazy loading. Please see the according example for more information. FetchListeners must be notified that a new item has been fetched.

```
// We determine which item should be the first returned.
int count = request.getNbItemToReturn() == -1 ? size :
    Math.min(size, request.getNbItemToReturn());
int startItem = request.getStartItemNumber() == -1 ? 0 :
    request.getStartItemNumber() ;
int end = Math.min(startItem + count ,
    itemsSortedAndMatchingQuery.size());
// We load all items between start item and end item,
// notify the fetchListeners that an item is being
    fetched ,and
// add it to the resultS
for (int i = (startItem ) ; i < end ; i++) {
        Item item = (Item) itemsSortedAndMatchingQuery.
            get(i);
        loadItem(item);
        item.setFullyLoaded(true);
        notifyItemFetchListeners(item);
        result.add(i-startItem ,item);
}
```

Finally, we can finish the fetch process by notifying the FetchListeners that the process is complete:

```
//Finally , we pass the result to the fetch listeners
notifyCompleteFetchListeners(result , request);
```

3.5.2.2 A datastore connecting to a GWT RPC Backend

This section will show an example of a purely remote datastore, which fetch all its items from a GWT-RPC Service. This implementation is quite simple, and allows the filtering, and sorting operation to be managed on the server.

3.5.2.2.1 The service We will begin with defining our GWT-RPC service. This one has nothing special, and offers CRUD operation for a small pet store :

```
public interface PetSearchService extends RemoteService{
        final public String SPECIE = "Specie";
        final public String NAME = "Name";
        final public String AGE = "Age";
        final public String TATOONUMBER = "TatooNumber";
        public List<Pet> getAllPets();
        public List<Pet> getAllPetsSortedBy(String
            sortCriteria ,boolean ascending);
        public List<Pet> getPetsBySpecie(String specie);
        public List<Pet> getPetsBySpecieSortedBy(String
            specie , String sortCriteria ,boolean
            ascending);
        public void updatePet(Pet pet);
        public void addPet(Pet pet);
        public void removePet(Pet pet);
}
```

3.5.2.2.2 Data The pet class itself extends Item. The first part of the class simply defines a bean:

```
public class PetItem extends Item implements Pet,
    Serializable{
        public PetItem(){ super(); }
        public PetItem(int age, String name, String
            specie, int tatooNumber) {
                super(); this.age = age;
                this.name = name;
                this.specie = specie;
                this.tatooNumber = tatooNumber;
        }
        private int age;
        public int getAge() {return age;}
        public void setAge(int age) {this.age = age;}
        private String name;
        public String getName() { return name; }
        public void setName(String name) { this.name =
            name; }
        private String specie;
        public String getSpecie() { return specie; }
        public void setSpecie(String specie) { this.
            specie = specie; }
        private int tatooNumber;
        public int getTatooNumber() { return tatooNumber;
             }
        public void setTatooNumber(int tatooNumber) {
            this.tatooNumber = tatooNumber; }
}
```

Then, we override the standard items method. The following changes were made, in order to not to store the attributes in the hashmap but rather using the instance fields:

getId method: we return the tattoo number

getValue: we call the getter corresponding to the attribute being requested

setValue: we call the setter corresponding to the attribute being modified

```
@Override
        public Object getId() { return getTatooNumber();
            }
        @Override
        public  String[] getAttributes() {
                return  {PetSearchService.SPECIE,
                    PetSearchService.NAME, PetSearchService
                    .AGE, PetSearchService.TATOONUMBER};
        }
        @Override public Object getValue(String
            attributeName, Object defaultValue) {
                return getValue(attributeName) == null ?
                    defaultValue : getValue(attributeName)
                    ;
        }
        @Override public Object getValue(String
            attributeName) {
                if(attributeName.equals(PetSearchService.
                    AGE)){
                        return getAge();
                }else if(attributeName.equals(
                    PetSearchService.NAME)){
                        return getName(); }
                else if(attributeName.equals(
                    PetSearchService.SPECIE)){
                        return getSpecie();
                }else if(attributeName.equals(
                    PetSearchService.TATOONUMBER)){
                        return getTatooNumber();
                }else return null;
        }
        @Override public void setValue(String
            attributeName, Object value) {
                if(attributeName.equals(PetSearchService.
                    AGE)){
                        if(value instanceof String){
                                value = Integer.parseInt
                                    ((String) value);
                        }
                        setAge((Integer) value);
                }else if(attributeName.equals(
                    PetSearchService.NAME)){
                        setName((String) value);
                }else if(attributeName.equals(
                    PetSearchService.SPECIE)){
                        setSpecie((String) value);
```

```
                  }else if(attributeName.equals(
                      PetSearchService.TATOONUMBER)){
                          if(value instanceof String){
                                  value = Integer.parseInt
                                      ((String) value);
                          }
                          setTatooNumber((Integer) value);
                  } super.setValue(attributeName, value);
          }
```

3.5.2.2.3 Datastore Fetch implementation The datastore implementation, will parse the request to determine wich service method should be invoked. Since the GWT-RPC process is asynchronous, we must provide an AsyncCallback implementation which will perform the fetch process. As you can notice, this implementation only calls the fetchlisteners and stores the fetched items:

```
private class DataStoreAsyncCallback implements
    AsyncCallback<List<Pet>>{
        Request request;
        public DataStoreAsyncCallback(Request request){
            this.request = request; }
        public void onFailure(Throwable arg0) {
            notifyErrorFetchListeners(); }
        public void onSuccess(List<Pet> pets) {
                notifyBeginFetchListeners(pets.size(),
                    request);
                List<Item> petsItems = new      ArrayList
                    <Item>();
                for (Iterator iterator = pets.iterator();
                     iterator.hasNext();){
                        PetItem item = (PetItem) iterator
                            .next();
                        items.put(item.getId(),item);
                        petsItems.add(item);
                        notifyItemFetchListeners(item);
                }
                notifyCompleteFetchListeners(petsItems,
                    request);
        }
}
```

The fetch method itself will determine wich method from the service to call. This method can be :

getAllPets: if no filtering or sort parameters are provided with the request

getAllPetsSortedBy: if only sort parameters are provided

getPetsBySpecie: if a filtering parameter named SPECIE is providedd

getPetsBySpecieSortedBy: same as previous, but with a sort parameter

```
@Override public void fetch(Request request) {
        String sortParameter = null;
        boolean ascending = true;
        notifyBeginFetchListeners(0, request);
        if(request.getSortFields().size() > 0){
                sortParameter = request.getSortFields().
                    get(0).getAttribute();
                ascending = !request.getSortFields().get
                    (0).isDescending();
        }
        if(request.getQuery().containsKey(
            PetSearchService.SPECIE)){
                if(sortParameter != null){
                        petSearchService.
                            getPetsBySpecieSortedBy((
                            String)request.getQuery().get(
                            PetSearchService.SPECIE),
                            sortParameter,ascending,new
                            DataStoreAsyncCallback(request
                            ));
                }else{
                        petSearchService.getPetsBySpecie
                            ((String)request.getQuery().
                            get(PetSearchService.SPECIE),
                            new
                            DataStoreAsyncCallback(request
                            ));
                }
        }else{
                if(sortParameter != null){
                        petSearchService.
                            getAllPetsSortedBy(
                            sortParameter,ascending,new
                            DataStoreAsyncCallback(request
                            ));
                }else{
                        petSearchService.getAllPets(new
                            DataStoreAsyncCallback(request
                            ));
                }
        } notifyCompleteFetchListeners(new ArrayList<Item
            >(),request);
}
```

3.5.2.2.4 DataStore Creation-Update-Delete implementation As we want to allow our datastore to create/update/delete items on the server, we have to override the corresponding methods:

```
@Override public void onDelete(Item item) {
        super.onDelete(item);
        Pet pet = ((PetItem)item);
        petSearchService.removePet(pet, new
            DataStoreDoNothingCallBack());
}
@Override public void onNew(Item item) {
        super.onNew(item);
        Pet pet = ((PetItem)item);
        petSearchService.addPet(pet, new
            DataStoreDoNothingCallBack());
}
@Override public void onSet(Item item, String attribute,
    Object oldValue, Object newValue) {
        super.onSet(item, attribute, oldValue, newValue);
        Pet pet = ((PetItem)item);
        petSearchService.updatePet(pet, new
            DataStoreDoNothingCallBack());
}
```

At last, you can use this datastore with a grid, for example, wich will allow you to manipulate this datastore through an ui and propagate the modifications back to the server:

```
grid = new Grid();
grid.setStore(new RPCDataStore());
GridLayout gridLayout = new GridLayout();
grid.setSize("100%", "20em");
grid.addColumn("Name",PetSearchService.NAME);
grid.addColumn("Specie",PetSearchService.SPECIE);
grid.addColumn("Tatoo_Number",PetSearchService.
    TATOONUMBER);
grid.addColumn("Age",PetSearchService.AGE,new TextEditor
    ());
```

3.5.2.3 A datastore for lazy loading child items from a tree

Chapter 4

Charting

Tatami also includes dojo's simple and powerful charting system. It is located in the
com.objetdirect.tatami.client.charting package. We advice you to take a look at the numerous chart examples in the TestPages module.

4.1 Introduction

The Chart2D class is the widget component of the Charting system. To draw a chart, you have to follow those steps :

1. Create a Chart2D widget. This widget represents the whole chart.
2. Add (at least) a plot to this widget. The Plot type represents how the data are plotted, whereas the data itself is represented by the Serie object. Plot is a generic type, it should be parametrized with the type of data its series can accept. Use the following method in the Chart2D class : addPlot(Plot) See the Plot section for how to create a Plot.
3. Add series to the plots. Serie is a generic type, it should be parametrized with the type of data it can draw. Use the following method from the Plot class : addSerie(Serie)
4. (Optional) Add Axes to the Plot and / or to the chart. See the Axis section
5. (Optional) Add Effects to the Plot. See the Effects section
6. Render the chart. This is done automatically when you attach the chart to its parent.
7. (Optional) Update it's series as they are modified
8. (Optional) re-render the chart when modifying it's Axis, Plots, Effects ...

If the chart is attached, any modification to the chart will have no effect until it is re-rendered !

4.2 Getting started

To build a chart, you need 3 objects : a Chart, a Plot and a Serie. Chart2D is the base class for a chart. It acts as a “container” in which Plots can be drawn. A plot is defined by its type (how are the data represented ?) and has series. A Serie is an object representing data to be plotted.

First, we have to create a chart. A chart must have a defined size, otherwise it will not be able to be rendered.

```
Chart2D chart = new Chart2D("200px", "200px");
```

Then, we must define a plot to be drawn on the chart:

```
Plot2D linePlot = new Plot2D(Plot2D.PLOT_TYPE_LINES);
chart.addPlot(linePlot);
```

The plot type is defined in this constructor.

The possible values are contained in the Plot, Plot2D, BarPlot and PiePlot classes.

We will explain this later in the Plot types section.

Now that we have a plot defined in the chart, we have to feed it with some data.

The Serie object acts as a container for data.

The Serie class has a constructor defined like this :

```
/**
* A serie contains data, stored as a List.
* This data can be : a Number, a Point , a Bubble
* or a PiePiece , depending on the chart type
* @param data : the data to plot on the chart
*/
public Serie(List<T> data) ;
```

We build a new Serie object and add it to the chart.

```
Double[] data3 = new Double[]{6.3, 1.8, 3., 0.5, 4.4, 2.7,
    2.};
Serie<Double> mySerie1 = new Serie<Double>(Arrays.asList(
   data1));
linePlot.addSerie(mySerie1);
```

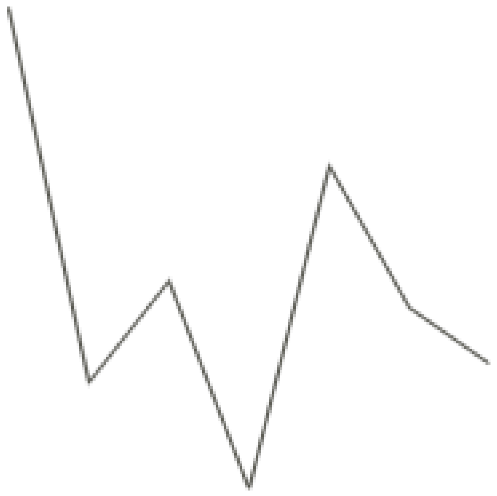
Then, we can add the chart to the page :

```
RootPanel.get().add(chart);
```

Please note that attaching the chart to the page automatically calls the refreshChart method. You always can refresh the chart by calling

```
public void refreshChart()
```

This complete code will output the following chart :



This chart is not very expressive ... We will see how to customize its appearance, add axes, legend, effects ...

4.3 Plot types

The charting system allows several type of plots. All plots type do not accept any type of series.

There are a few number of Plot classes available.

4.3.1 Plot2D

This class is used to draw several types of two dimensional plots.

4.3.1.1 What type of data to use?

This type of plot can be used with series of anything which extends Number, or with the Point object. When using numbers, the X coordinate is the position of the number in the list. When using Points, both X and Y coordinates must be specified.

Ex :

```
Serie<Point> serie1 = new Serie<Point>();
serie1.addData(new Point(3.,2.));
```

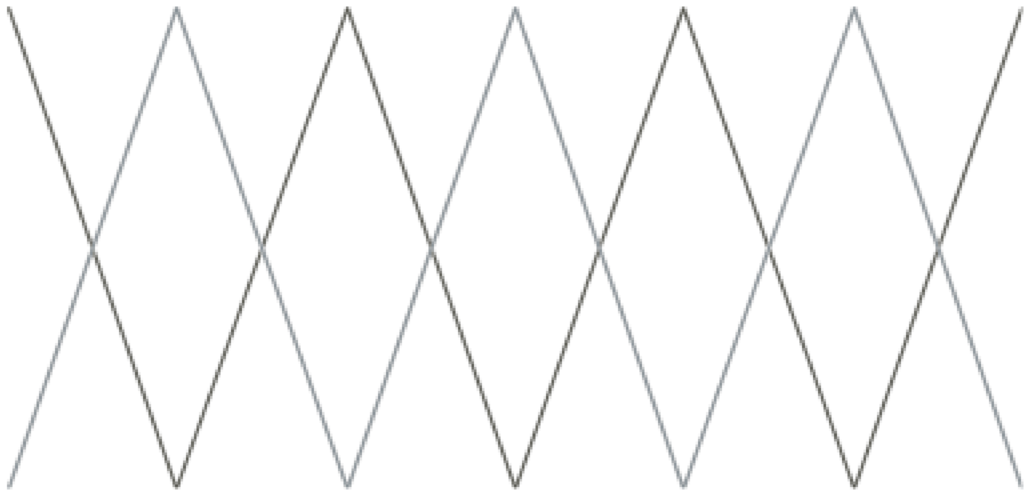
4.3.1.2 Subtypes

The plot type is determined with the constructor:

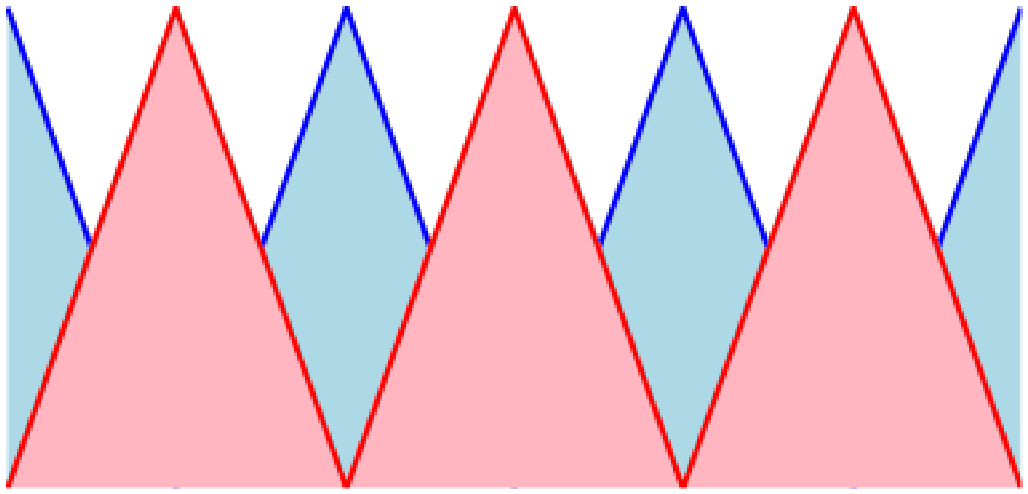
```
public Plot2D(String type) ;
```

Where type is one of the following constants :

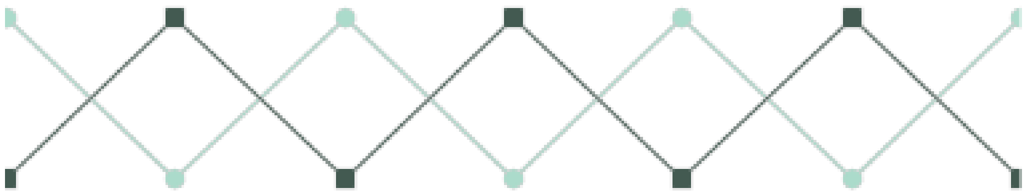
Plot2D.PLOT_TYPE_LINES Series are drawn as lines Ex:



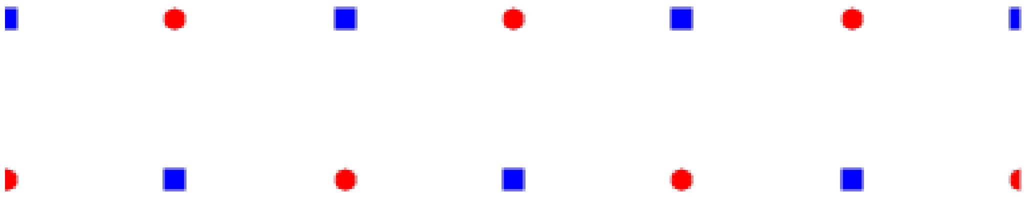
Plot2D.PLOT_TYPE_AREAS Series are drawn as lines, and the area under lines is filled. Ex :



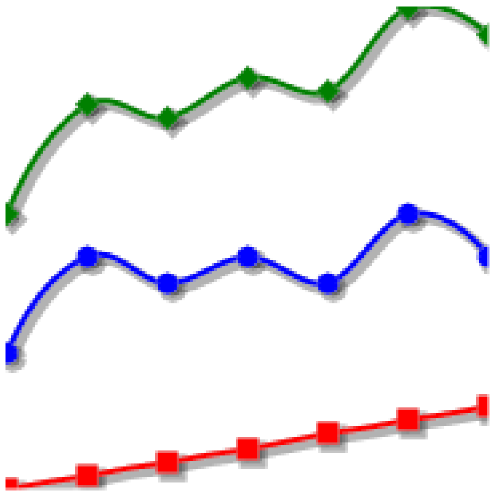
Plot2D.PLOT_TYPE_MARKERS_LINES Series are drawn as lines, with markers on the actual points Ex:



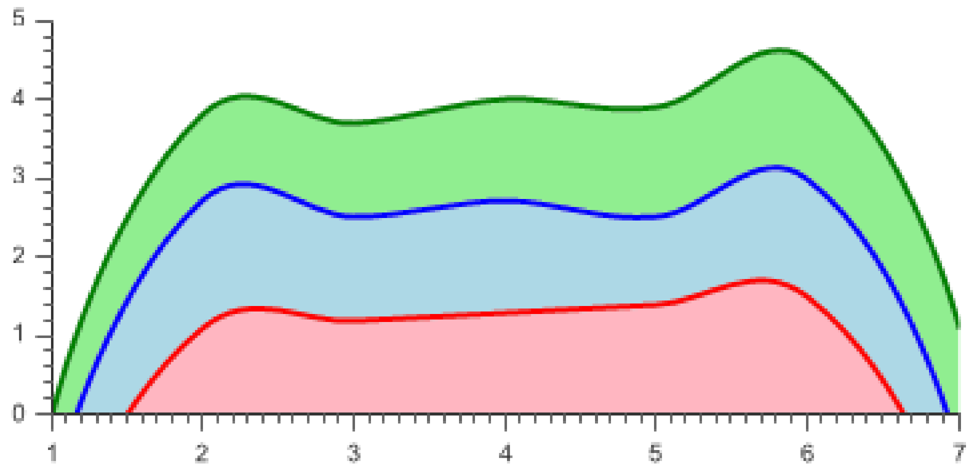
Plot2D.PLOT_TYPE_SCATTER Series are drawn as points only Ex:



Plot2D.PLOT_TYPE_LINES_STACKED Series are drawn as lines and the y positions are additive. Ex:



Plot2D.PLOT_TYPE_AREAS_STACKED Series are drawn as lines, the area beneath is filled and the y positions are additive. Ex:



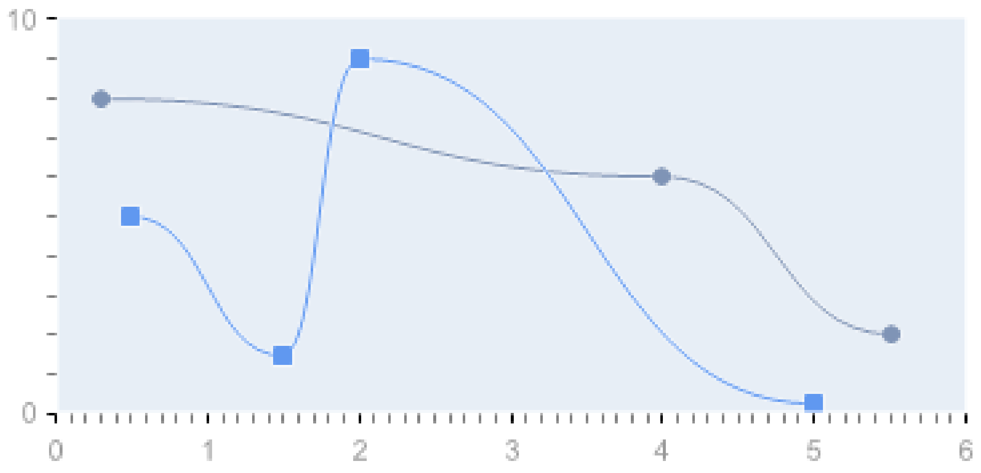
4.3.1.3 Specific options

Tension: It allows you to add some curve to the lines

```
public void setTension(String tension);
```

Example with tension = 2 :

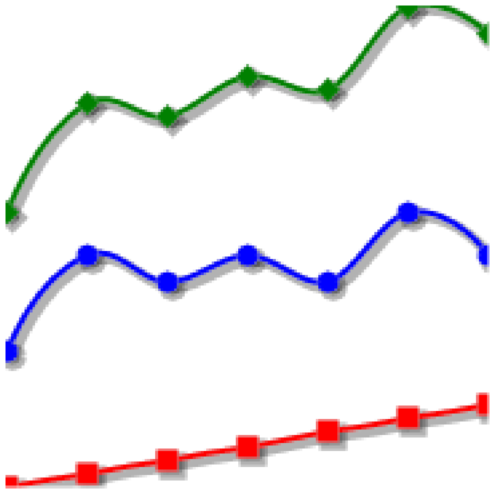
```
Plot2D plot = new Plot2D(Plot2D.
    PLOT_TYPE_MARKERS_LINES);
plot.setTension("2");
```



Shadow:
```
/**
* Creates a shadow
*bubbles
* @param width : the shadow's width
* @param dx : shadow's x position from the line
* @param dy : shadow's y position from the line
*/
public void setShadow(int width, int dx, int dy);
```

Example :

```
Plot2D<Double> plot = new Plot2D<Double>(Plot2D.
    PLOT_TYPE_LINES_STACKED);
plot.setShadow(2, 2, 2);
```



ShowMarkers:
```
/**
* @param show : Set wether to show the markers or not
    .
* Overrides the default "PLOT_TYPE" behavior.
*/
public void setShowMarkers(boolean show);
```

ShowLines:
```
/**
* @param show : Set wether to show the markers or not
    .
* Overrides the default "PLOT_TYPE" behavior.
*/
public void setShowLines(boolean show);
```

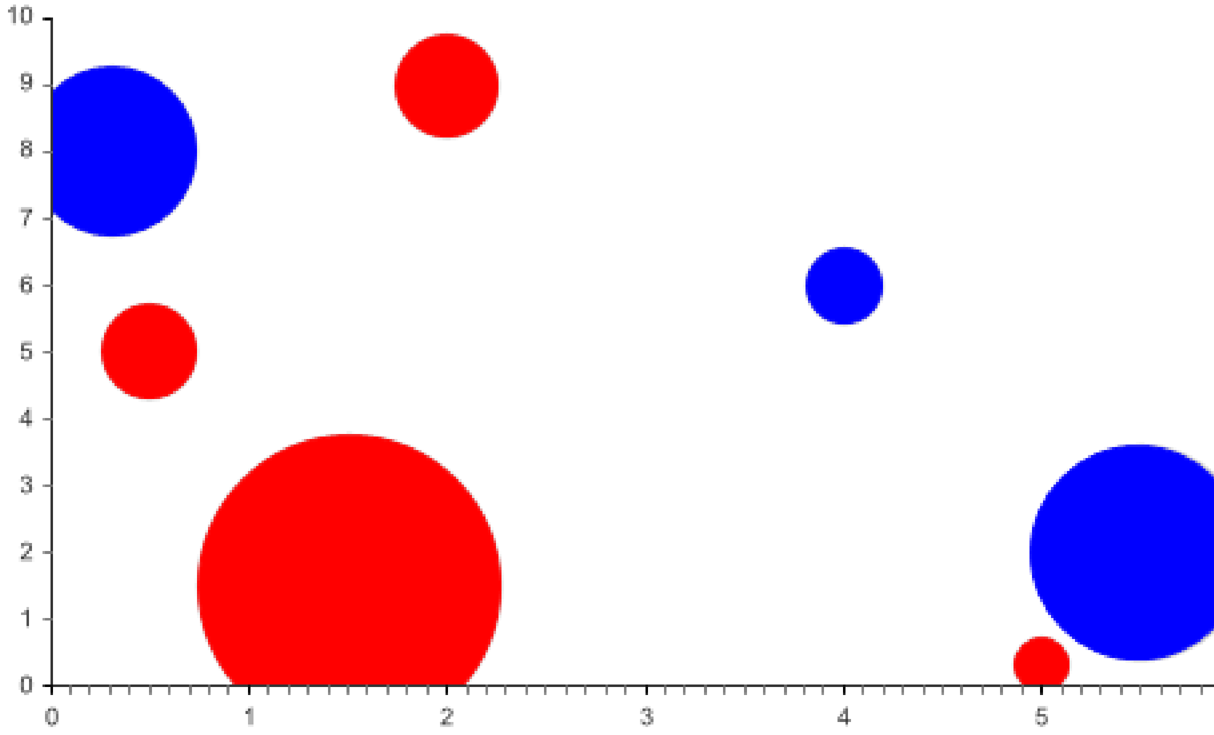
4.3.2 BubblePlot:

The BubblePlot class is a Chart2D subclass. It is used to represent data as "Bubbles", defined by their coordinates and radius. It has its own class, since it uses specific data objects : bubbles. The bubble plot is used with series of Bubble object :

```
public Bubble(double x, double y ,double size);
```

Example:

```
Bubble[] bubbles = new Bubble[]{new Bubble(0.3,8,2.5),new
     Bubble(4,6,1.1),new Bubble(5.5,2,3.2)};
Serie<Bubble> serie = new Serie<Bubble>(Arrays.asList(
    bubbles));
```



4.3.3 BarPlot

This class is used to draw several types of Bar and Columbs plots

4.3.3.1 What type of data to use?

This type of plot can be used with series of anything wich extends Number. The y coordinate is the number, where the x coordinate is the data's position in the Serie.

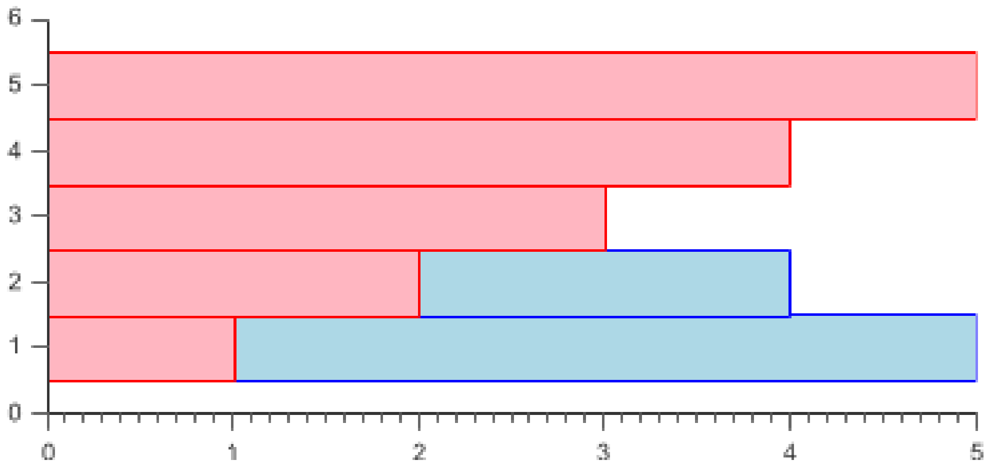
4.3.3.2 Subtypes

. The plot type is determined with the constructor:

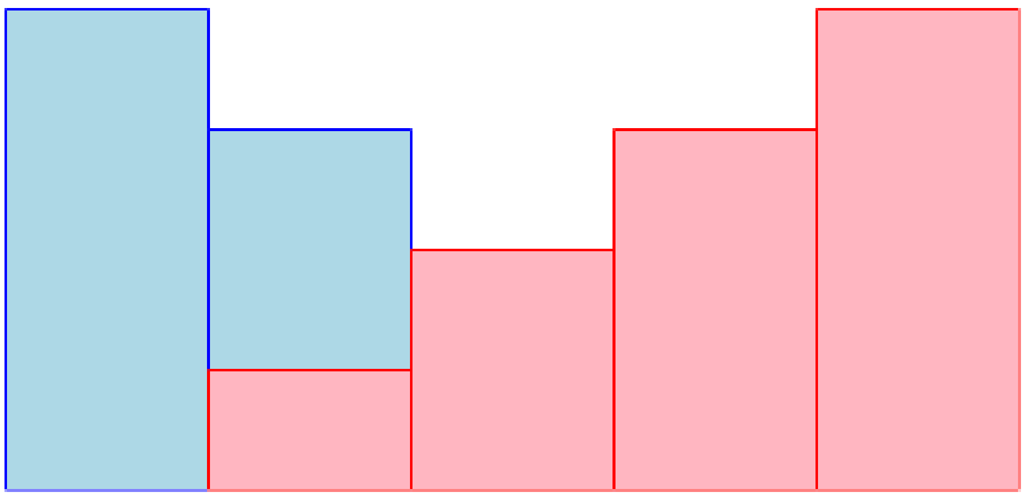
```
public BarPlot(String type) ;
```

Where type is one of the following constants :

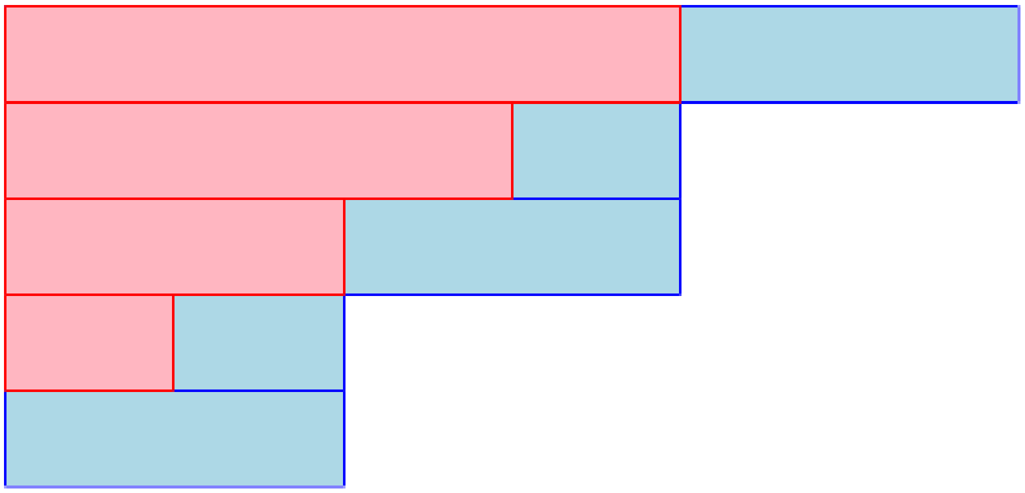
BarPlot.PLOT_TYPE_BARS: Series are drawn in horizontal bars. Ex:



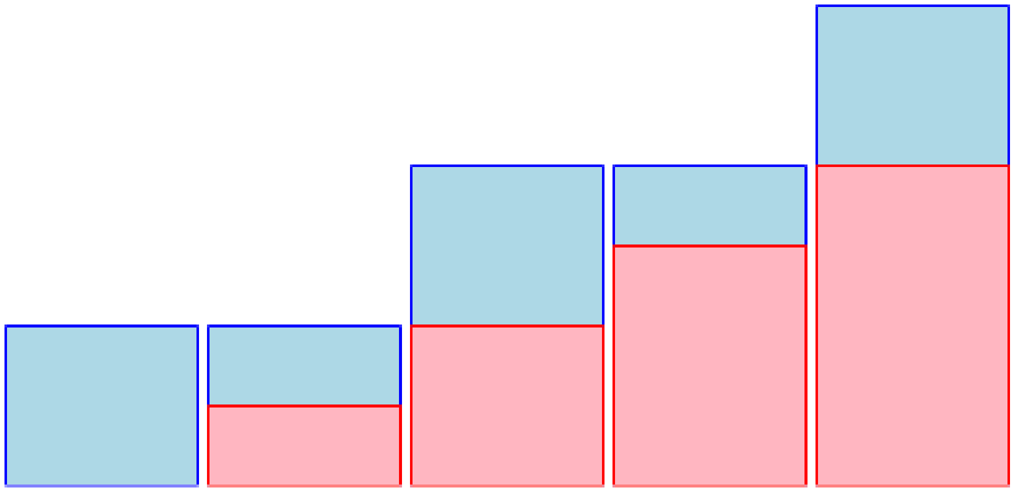
BarPlot.PLOT_TYPE_COLUMNS: Series are drawn in vertical bars. Ex:



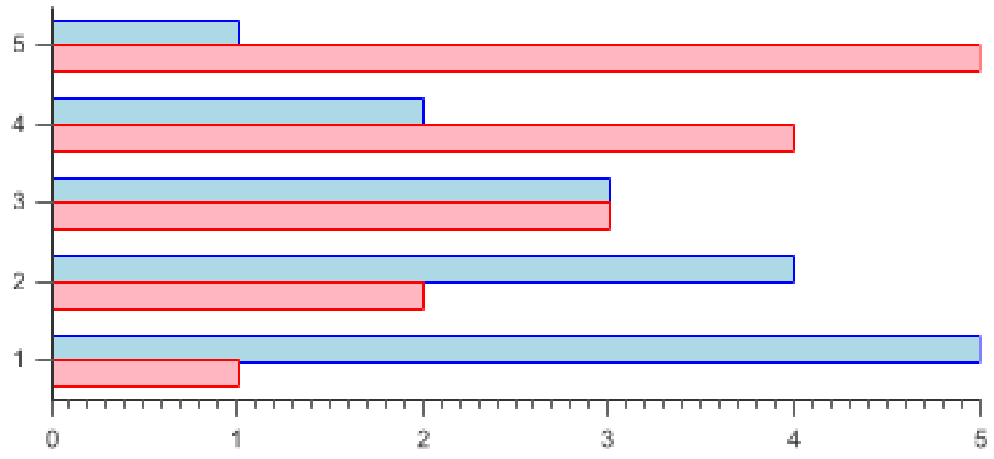
BarPlot.PLOT_TYPE_BARS_STACKED: Series are drawn in stacked horizontal bars, adding the bars values from one serie to the next. Ex:



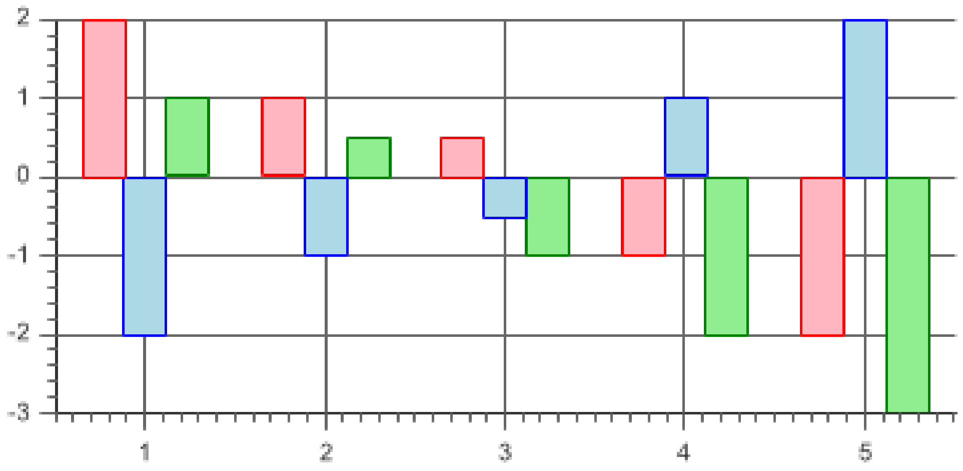
BarPlot.PLOT_TYPE_COLUMNS_STACKED Same as BARS_STACKED, but with vertical bars. Ex:



BarPlot.PLOT_TYPE_BARS_CLUSTERED: Bars which do not overlap. Ex:



BarPlot.PLOT_TYPE_COLUMNS_CLUSTERED: Columns which do not overlap. Ex:

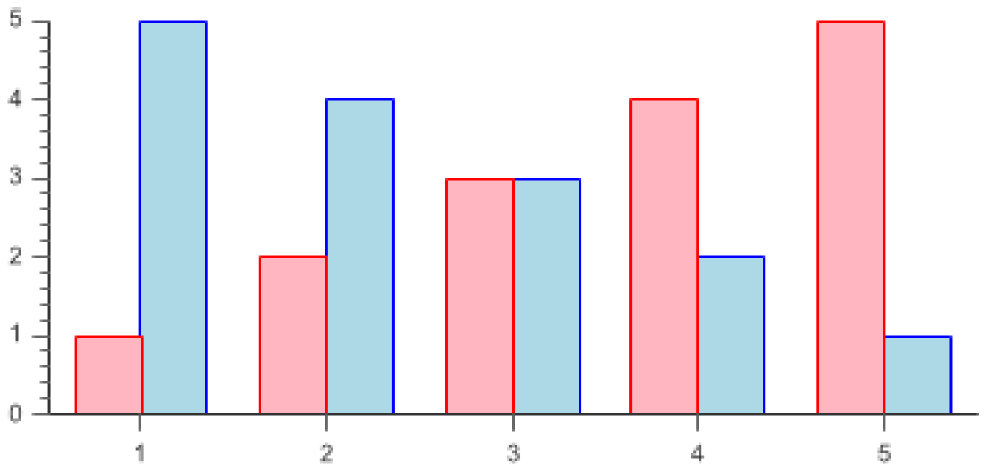


4.3.3.3 Specific options

```
/**
* @param gap : sets the gap between two bars
*/
public void setGap(int gap);
```

ex: Clustered Columns with gap = 10 :

```
Chart2D chart = new Chart2D("400px", "200px");
BarPlot plot = new BarPlot(BarPlot.
   PLOT_TYPE_COLUMNS_CLUSTERED);
plot.setGap(10);
chart.addPlot(plot);
```



4.3.4 PiePlot

4.3.4.1 What type of data to use?

The PiePlot can be used with series of number or PiePiece. A PiePiece represents slice in the Pie Plot, and provides various constructors:

```
public PiePiece(double value)
public PiePiece(double value , String label)
public PiePiece(double value , String label,String color)
public PiePiece(double value , String label,String color,
     String fontColor);
```

where :

value: the PiePiece value

label: the label to be displayed on the Piece

color: the color filling the piece

fontColor: the label font color

4.3.4.2 Specific options

```
/**
* @param offset : the label position from the edge of the
     plot
```

```
* Negative values will place the label outside of the
    plot, whereas positive values will place in inside
*/
public void setLabelOffset(int offset)
```

```
/**
* @param precision : the precision for percentiles
*/
public void setPrecision(int precision)
```

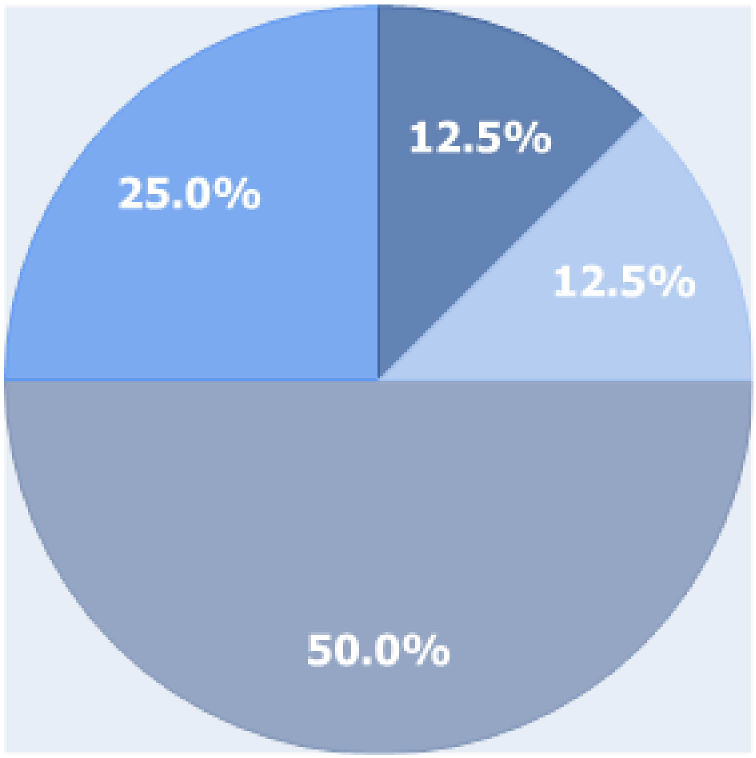
```
/**
* @param font : font to be applied on the various labels
*/
public void setFont(String font)
```

```
/**
* @param fontColor : default font color. Can be overriden
     by each PiePiece
*/
public void setFontColor(String fontColor)
```

Examples :

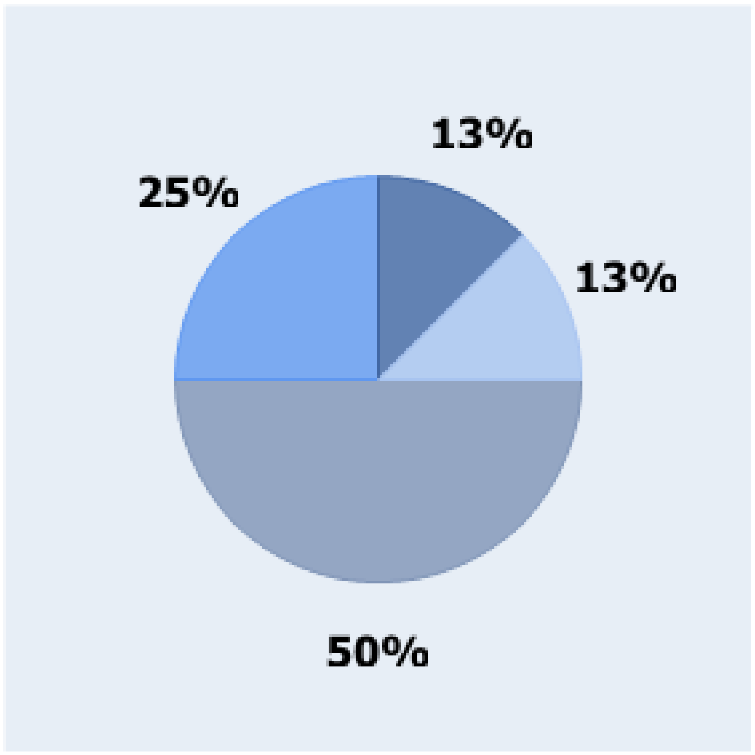
Pie chart from a Number Serie, with internal labels and default Precision :

```
Chart2D chart = new Chart2D("300px","300px");
chart.setTheme("PlotKit.blue");
PiePlot<Integer> plot = new PiePlot<Integer>();
plot.setFont("normal normal bold 12pt Tahoma");
plot.setFontColor("white");
plot.setLabelOffset(40);
List<Integer> data = Arrays.asList((new Integer
    []{4,2,1,1}));
plot.addSerie(new Serie<Integer>(data));
chart.addPlot(plot);
```



Pie chart from a Number Serie, with external labels and precision = 0 :

```
Chart2D chart = new Chart2D("300px","300px");
chart.setTheme("PlotKit.blue");
PiePlot<Integer> plot = new PiePlot<Integer>();
plot.setFont("normal normal bold 12pt Tahoma");
plot.setFontColor("black");
plot.setLabelOffset(-25);
plot.setPrecision(0);
plot.addSerie(new Serie<Integer>(Arrays.asList(new
    Integer[]{4,2,1,1})));
chart.addPlot(plot);
```



4.4 Axes

The charting API offers numerous options for customizing Axis.

4.4.1 Setting axis

Axis can be set at the chart or at the plot level. Axis set at the chart level will be used by all plots which don't have any specific axis.

To set the default X and Y axis, use the following methods from the Chart2D class :

```
public void setDefaultXAxis(Axis axis);
public void setDefaultYAxis(Axis axis);
```

The Axis class provides two constructors :

```
/**
* Constructs a default bottom horizontal axis
*/
public Axis();
/**
```

```
* @param position : an int representing the Axis position
    .
* It is obtained from int byte to byte OR operation
    between the following constants :
*
* Axis.LEFT
* Axis.RIGHT
* Axis.BOTTOM
* Axis.TOP
* Axis.VERTICAL
* Axis.HORIZONTAL
*/
public Axis(int position) ;
```

For example, a right vertical axis would be created as :

```
new Axis(Axis.RIGHT | Axis.VERTICAL);
```

The Axis class also proposes two helper methods to get a bottom X axis and a left Y Axis :

```
/**
* @return a default horizontal axis instance
*/
public static Axis simpleXAxis()
/**
* @return a default vertical axis instance
*/
public static Axis simpleYAxis()
```

The following snippet can then be used to provide simple axes to your charts :

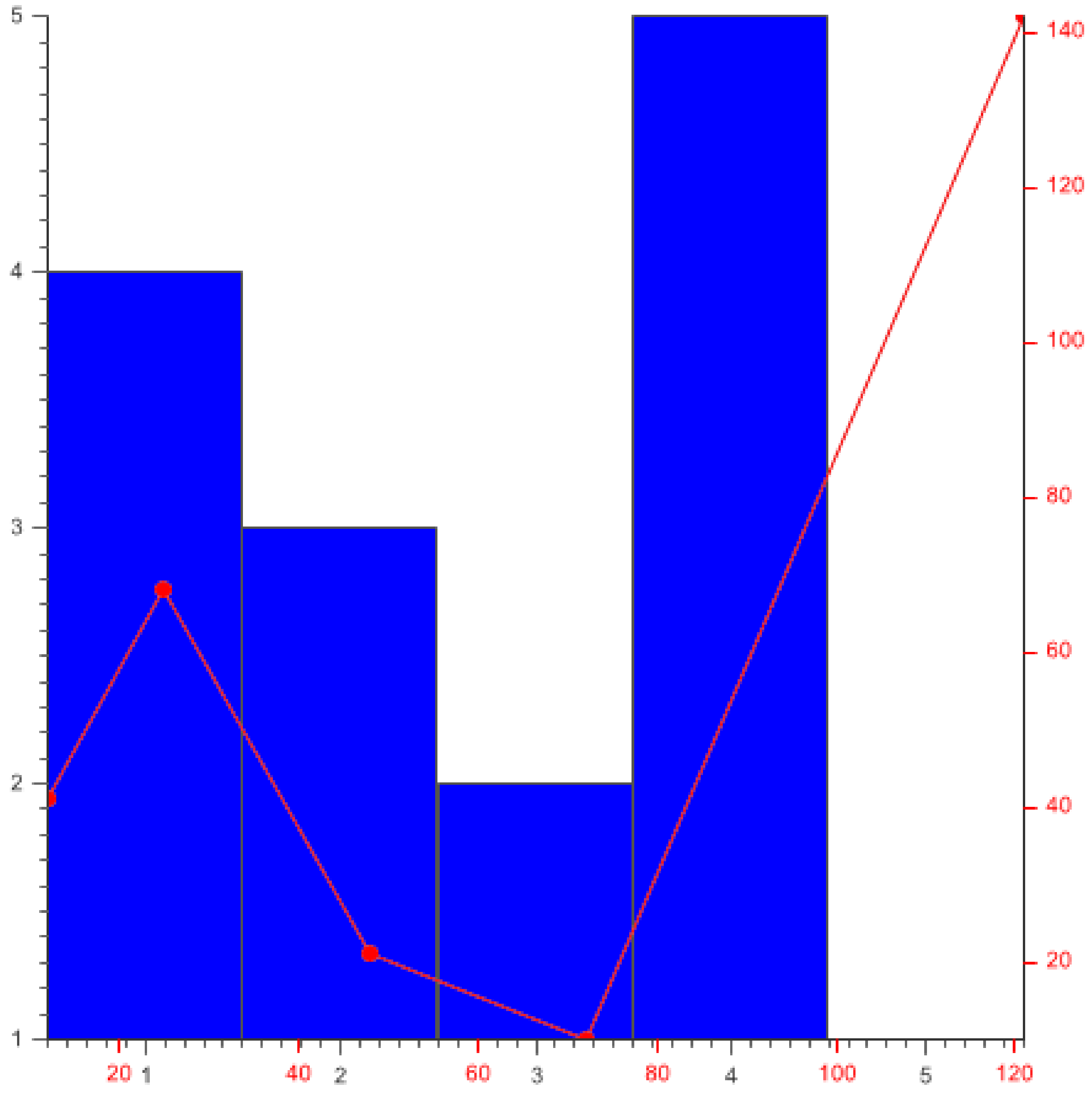
```
chart.setDefaultXAxis(Axis.simpleXAxis());
chart.setDefaultYAxis(Axis.simpleYAxis());
```

To set axes for a specific plot, you can use the Plot methods

```
public void setXAxis(Axis axis);
public void setYAxis(Axis axis);
```

Below is an example of two plots on the same chart with different axes :

```
Chart2D chart = new Chart2D("500px","500px");
//Constructing the Column plot
BarPlot<Integer> columns = new BarPlot<Integer>(BarPlot.
    PLOT_TYPE_COLUMNS);
//Constructing its serie
Serie<Integer> columnsSerie = new Serie<Integer>();
columnsSerie.addData(4);
columnsSerie.addData(3);
columnsSerie.addData(2);
columnsSerie.addData(5);
columnsSerie.addData(1);
//Simple Serie options, see the "Series" section for more
     info
columnsSerie.setFillColor("blue");
//We add the serie to the column plot
columns.addSerie(columnsSerie);
//Constructing the line plot
Plot2D<Point> lines = new Plot2D<Point>(Plot2D.
    PLOT_TYPE_MARKERS_LINES);
//Constructing its series from points
Serie<Point> linesSerie = new Serie<Point>();
linesSerie.addData(new Point(12.,41.));
linesSerie.addData(new Point(25.,68.));
linesSerie.addData(new Point(48.,21));
linesSerie.addData(new Point(72,10));
linesSerie.addData(new Point(121,142));
//Simple Serie options, see the "Series" section for more
     info
linesSerie.setStrokeColor("red");
//We add the serie to the column plot lines.addSerie(
    linesSerie);
//The plot are added to the chart
chart.addPlot(lines);
chart.addPlot(columns);
//The chart will have default X and Y axis
chart.setDefaultXAxis(Axis.simpleXAxis());
chart.setDefaultYAxis(Axis.simpleYAxis());
/* We override this setting for the line plot
* The getLineXAxis and getLineYAxis are not shown
* here, but will be presented in the "Axis options"
    section.
* They only return new, customized for the example, axis.
*/
lines.setXAxis(getLineXAxis());
lines.setYAxis(getLineYAxis());
```



4.4.2 Axis Options

The precedent example shows how axis can be customized. There are a lot of options for Axis . . .

4.4.2.1 Ticks Options

Axes can show up to 3 different kinds of ticks: major minor, and micro. You can enable/disable them with the corresponding methods :

```
public void setMinorTicks(Boolean minorTicks);
public void setMajorTicks(Boolean majorTicks);
public void setMajorTicks(Boolean microTicks);
```

Each of these ticks types have different options:

length: the length of the ticks, in pixels

color: the color of the ticks

step: the step between the ticks

The corresponding methods names are pretty straightforward:

```
setMajorTicksColor(String)
setMajorTicksLength(double)
setMajorTickStep(double)
setMinorTicksColor(String)
setMinorTicksLength(double)
setMicroTickStep(double)
setMicroTicksColor(String)
setMicroTicksLength(double)
setMicroTickStep(double)
```

Minor and Major ticks can also show labels :

```
setMajorLabels(Boolean)
setMinorLabels(Boolean)
```

Example :
The following example demonstrates the use of ticks options.

```
Axis xLinesAxis = new Axis(Axis.BOTTOM | Axis.HORIZONTAL)
    ;
xLinesAxis.setMajorTickStep(50);
xLinesAxis.setMajorTicksLength(15);
xLinesAxis.setMajorTicksColor("blue");
xLinesAxis.setMinorTickStep(20);
xLinesAxis.setMinorTicksLength(10);
xLinesAxis.setMinorLabels(true);
xLinesAxis.setMinorTicksColor("red");
xLinesAxis.setMicroTicks(true);
xLinesAxis.setMicroTickStep(10);
xLinesAxis.setMicroTicksLength(5);
xLinesAxis.setMicroTicksColor("green");
```

4.4.2.2 Miscellaneous options

This set of options allow you to customize some axis properties such as its minimum and maximum, if it should be aligned on ticks etc...

setIncludeZero(Boolean): sets whether the axis should be forced to include the zero.

setMax(double): sets the maximum value displayed on the axis

setMin(double): sets the minimum value displayed on the axis

setNatural(boolean): forces the ticks to be aligned on natural numbers

setFixed(Boolean): sets whether the labels precision is fixed

setFont(String): sets the font used for the labels

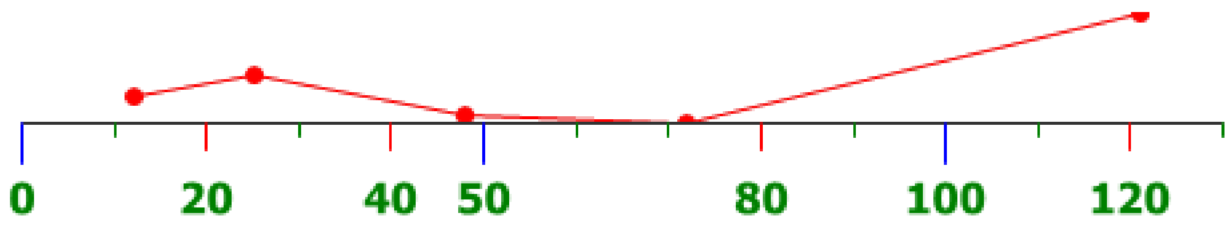
setFontColor(String): sets the font color used for the labels

setFixLower(String): forces the lower axis bound to be aligned on a tick. Possible values are : AXIS.FIX_TYPE_MAJOR, AXIS.FIX_TYPE_MINOR, AXIS.FIX_TYPE_MICRO

setFixUpper(String) : same as setFixLower, but for the upper bound.

Example :

```
xLinesAxis.setFixed(true);
xLinesAxis.setFixLower(Axis.FIX_TYPE_MAJOR);
xLinesAxis.setFixUpper(Axis.FIX_TYPE_MICRO);
xLinesAxis.setFont("normal normal bold 12pt Tahoma");
xLinesAxis.setFontColor("green");
```



Notice the upper value is aligned on a micro tick, the lower one is aligned on a major tick.

4.4.2.3 Labels

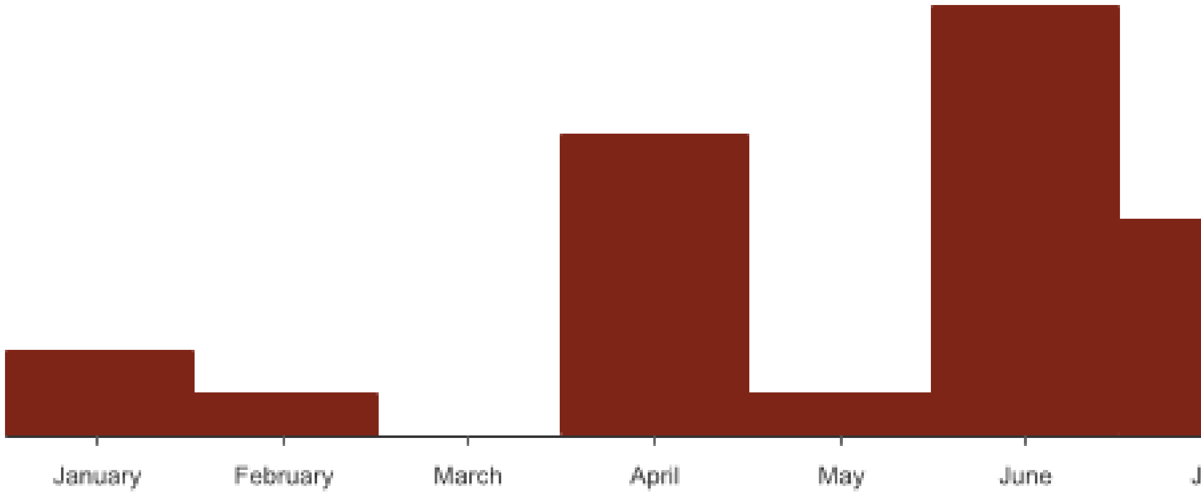
The Axis class provides default labels : they correspond to the actual value on the axis. But you can also provide your own labels.

```
public void addLabel(double forValue, String text)
```

This method customizes the label for the given forValue. Example :

```
String[] labels = new String[]{"January","February","
    March","April","May","June","July","August","September
    ","October","November","December"};
for (int i = 0; i < labels.length; i++) {
        xColumnsAxis.addLabel(i+1,labels[i]);
}
```

For each integer value on the X axis, we set the label to the corresponding month.



We could have done it as easily with the following method, which only adds labels for integer values, in the given order.

```
public void addLabels(String[] labels)
```

Example :

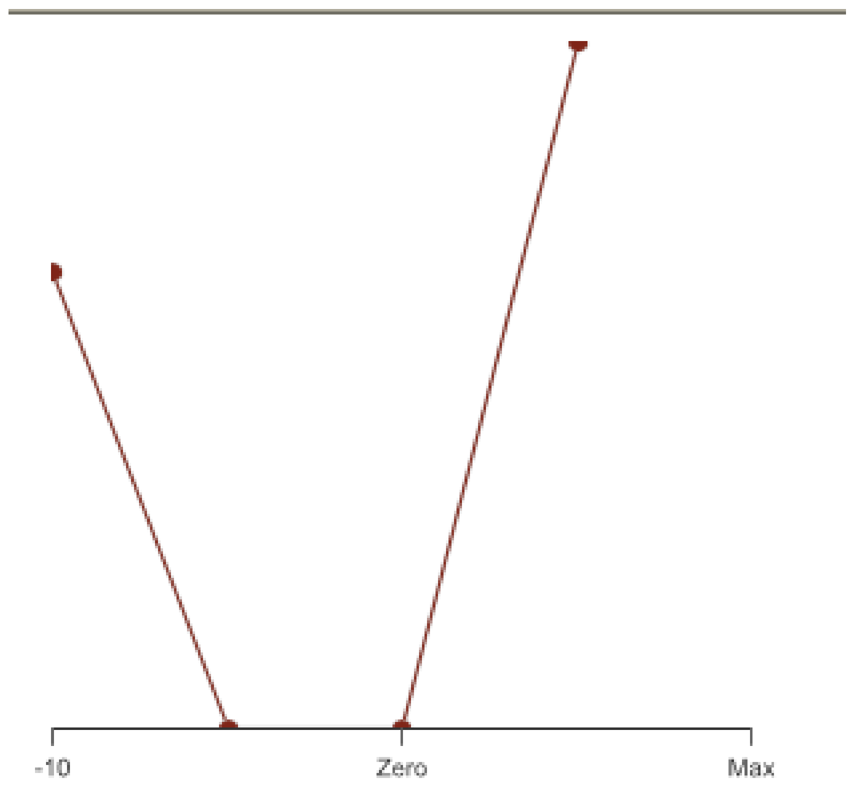
```
String[] labels = new String[]{"January","February","
    March","April","May","June","July","August","September
    ","October","November","December"};
xColumnsAxis.addLabels(labels);
```

The above example will have exactly the same effect as the previous one.

These methods only converts the given value/label couple into an AxisLabel object.

You can also use the AxisLabel object directly:

```java
public List<AxisLabel> getDiscreteLabels(){
        List<AxisLabel> labels = new ArrayList<AxisLabel
            >();
        labels.add(new AxisLabel("Zero",0));
        labels.add(new AxisLabel("Max",10));
        labels.add(new AxisLabel("Min",-10));
        return labels;
}
xLinesAxis.addLabels(getDiscreteLabels());
```



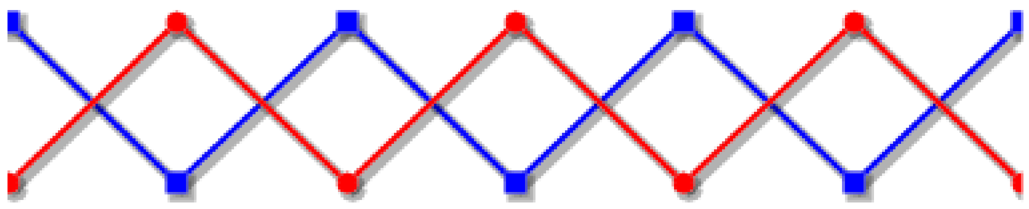
4.5 Series

Series are used to define chart data. It is basically a wrapper for a List, which adds specific options to customize the serie's rendering.

4.5.1 Simple serie customization example

This example shows how the series rendering can be customized. The options will be detailed in the next section.

```
Serie<Integer> serie1 = new Serie<Integer>(
    getSampleUniDimensionalData());
serie1.setMaxDisplayed(3.);
serie1.setMinDisplayed(0.);
serie1.setStrokeColor("red");
serie1.setStrokeWidth(2);
serie1.setMarkerType(Serie.MARKER_TYPE_CIRCLE);
Serie<Integer> serie2 = new Serie<Integer>(
    getSampleUniDimensionalData2());
serie2.setStrokeColor("blue");
serie2.setStrokeWidth(2);
serie2.setMarkerType(Serie.MARKER_TYPE_SQUARE);
```



4.5.2 Series and data

This section will explain how to add data to a Serie. First, you need to create a Serie. Serie is a generic type, parametrized with the type of data it can contain.

There are three constructors available for Series :

```
public Serie();
public Serie(String name);
public Serie(List<T> data);
public Serie(List<T> data, String name);
```

The data argument is used to initialize the Serie. Else, it will be empty and you will have to add data manually. The name argument is used when building a Legend. See the Legend widget documentation.

```
Serie<Integer> serie1 = new Serie<Integer>();
Serie<Double> serie2 = new Serie<Double>(myListOfDouble);
Serie<Double> serie3 = new Serie<Point>(myListOfPoint,'
    Gross Profit');
```

The types used for data depends on the type of Plot.

- Number can be used for Plot2D, BarPlot, PiePlot
- PiePiece can be used for PiePlot only
- Point can be used for Plot2D
- Bubble can be used for BubblePlot

The data is stored in a List in the Serie object. If you want to add data to the Serie, you have to get this list with the following method :

```
public List<T> getData()
```

Alternatively, you can use the delegate methods :

```
public void addData(T o)
public void addData(int index , T o)
public void removeData(T o)
public void removeData(int index)
```

4.5.3 Series Options

The Serie offers several methods to customize its rendering.

Fills the serie with the given color. It may the area for a line chart, or the columns/bars for a bar chart etc...

Sets the color of the lines.

Defines the lines width.

Defines the Marker Type. The marker argument can be any SVG path, but the Serie class provides constants for a few markers :

```
final public static String MARKER_TYPE_CIRCLE= "m-3,0
    c0,-4 6,-4 6,0 m-6,0 c0,4 6,4 6,0";
final public static String MARKER_TYPE_SQUARE= "m
    -3,-3 l0,6 6,0 0,-6 z";
final public static String MARKER_TYPE_DIAMOND= "m0
    ,-3 l3,3 -3,3 -3,-3 z";
final public static String MARKER_TYPE_CROSS= "m0,-3
    l0,6 m-3,-3 l6,0";
final public static String MARKER_TYPE_X= "m-3,-3 l6
    ,6 m0,-6 l-6,6";
final public static String MARKER_TYPE_TRIANGLE= "m
    -3,3 l3,-6 3,6 z";
final public static String
    MARKER_TYPE_TRIANGLE_INVERTED= "m-3,-3 l3,6 3,-6 z
    ";
```

Sets the Max displayed value for this Serie.

Sets the Min displayed value for this Serie.

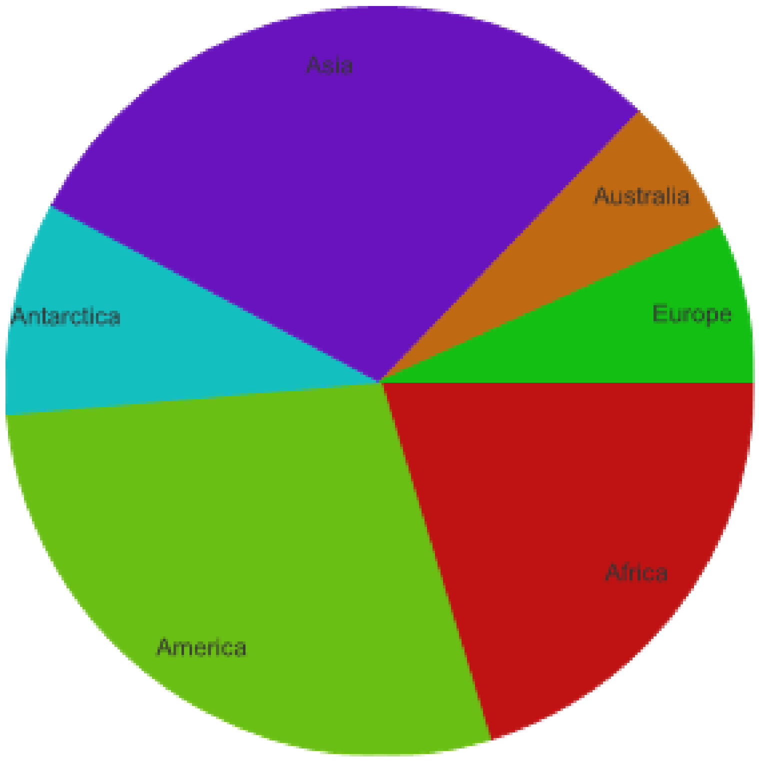
4.6 Themes

Dojo offers a few Themes for charting.
Those permits to use default colors.
You can choose to use a theme with the method Chart2D.setTheme:

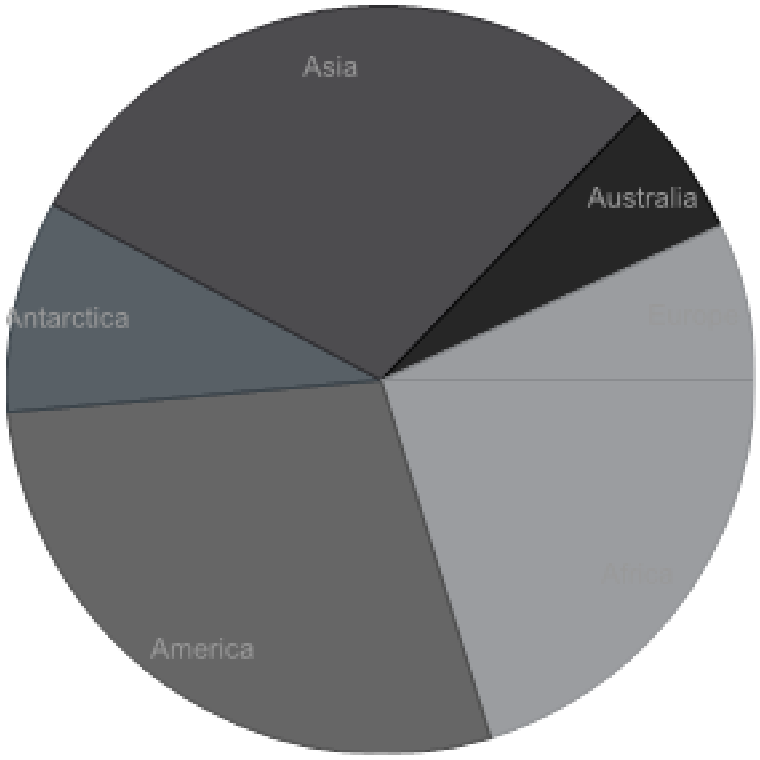
```
public void setTheme(String theme)
```

Where theme is a constant defined in the Themes class.
Example :

```
chart.setTheme(Themes.Shrooms);
```



```
chart.setTheme(Themes.Tufte);
```



The chart, axis, and Series options are exactly the same on the two charts. The only thing which has changed is the Theme.

4.7 Effects

You can add nice animations and event listeners to your chart. These effects are located in the com.objetdirect.tatami.charting. effects package, and inherit the Effect interface. Effects are added to the Plot object thanks to the plot.addEffect method :

```
public void addEffect(Effect effect)
```

We will review the available effects, and explain how to implement a ChartEventListener

4.7.1 Commons Options

Most effects have the duration option, which can be set at construction time or with the associated setter. It modifies the animation duration, in milliseconds.

```
public void setDuration(int duration);
```

4.7.2 EffectTooltip

The corresponding class is

```
com.objetdirect.tatami.client.charting.effects.
    EffectTooltip
```

The basic Tooltip effect will show the point (or slice/bar/column/bubble...) value in a tooltip.

Example :

```
plot.addEffect(new EffectTooltip());
```

But the Point, Bubble and PiePiece data objects each offer a way to give a custom tooltip to the data. They implement the HasTooltip interface:

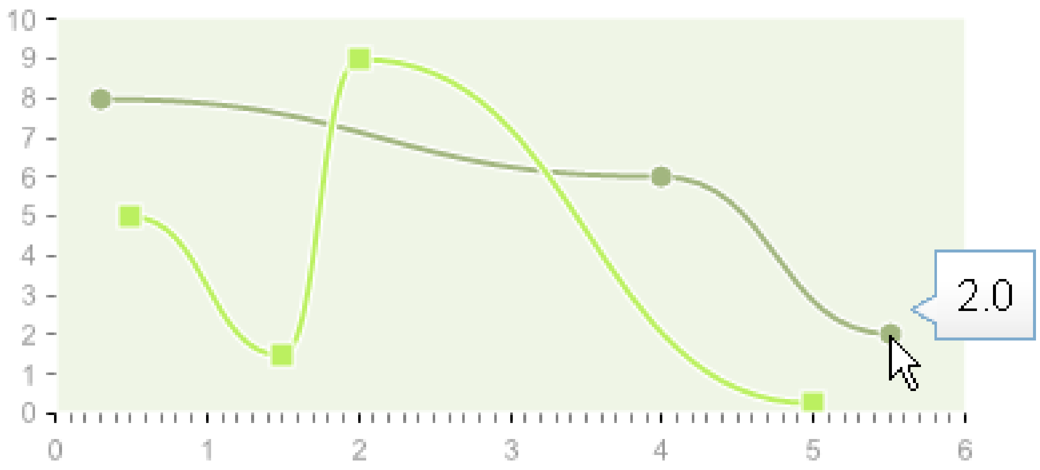
```
package com.objetdirect.tatami.client.charting;
public interface HasTooltip {
        public String getTooltip();
        public void setTooltip(String tooltip);
}
```

Moreover, they each have a constructor taking this tooltip into account :

```
public Bubble(double x, double y ,double size ,String
    tooltip)
public Point(double x, double y , String tooltip)
public PiePiece(double value , String label,String color,
     String fontColor, String tooltip)
```

Example :

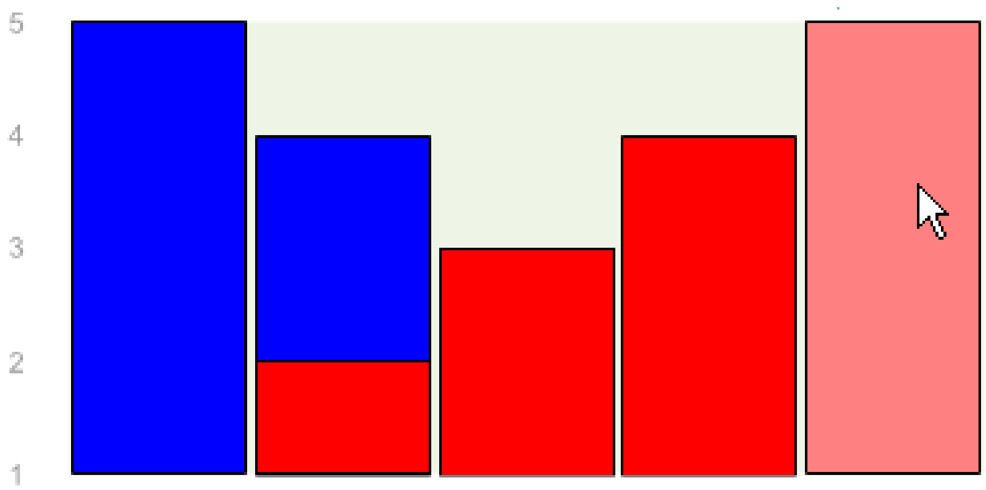
```
Point[] points = new Point[]{new Point(0.3,8.0),new Point
    (4.,6.,"Custom tooltip"),new Point(5.5,2.)};
Serie<Point> serie = new Serie<Point>(Arrays.asList(
    points2), "Serie 2");
plot.addSerie(serie);
plot.addEffect(new EffectTooltip());
```



4.7.3 EffectHighlight

This effect highlights the chart element under mouse :

```
plot.addEffect(new EffectHighlight());
```



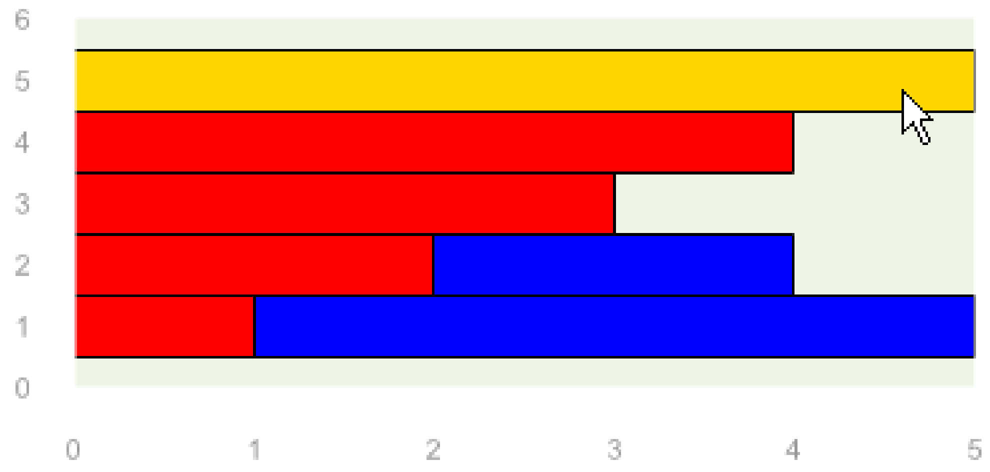
The default behavior is this one, but you can chang the highlighting color :

```
public EffectHighlight(String color)
public void setColor(String color)
```

For example :

```
plot.addEffect(new EffectHighlight("gold"));
```

will produce the following effect :



4.7.4 EffectShake

This effect has no other option than the common "duration" option. It will shake the chart element under mouse.

```
plot.addEffect(new EffectShake());
```

4.7.5 EffectMagnify

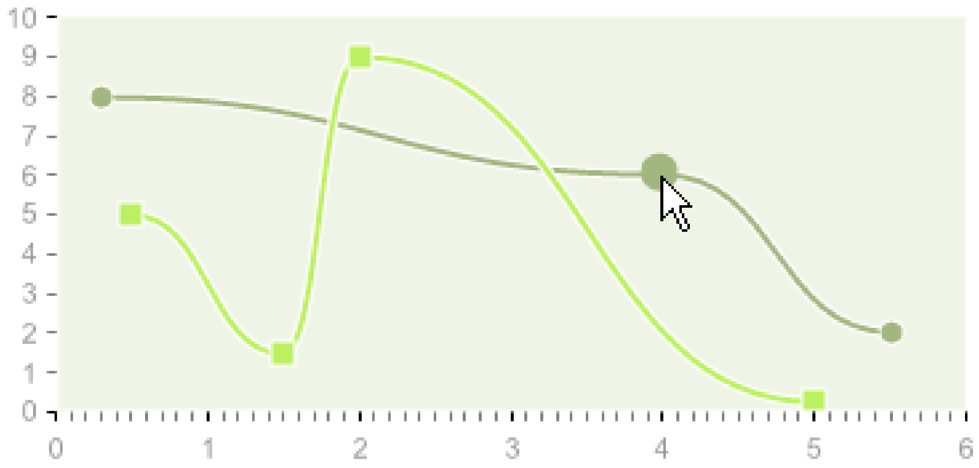
This effect magnifies the chart element under mouse.

```
plot.addEffect(new EffectMagnify());
```

It has a « scale » parameter which specifies the magnifying ratio.

```
public EffectMagnify(double scale);
setScale(double scale)
```

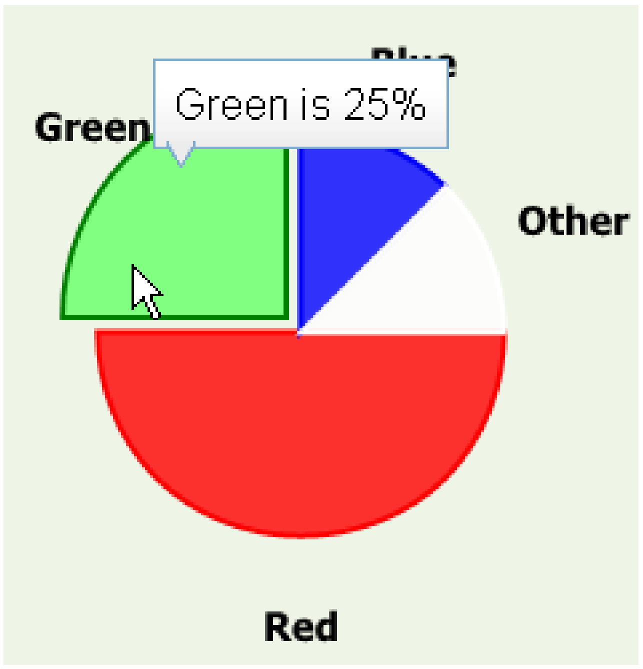
Example for scale=2 :



4.7.6 EffectMoveSlice :

This effect can only be added to a PiePlot. It moves the PieSlice away from the center. It has no other options than the common duration.

```
PiePlot<PiePiece> plot = new PiePlot<PiePiece>();
plot.addEffect(new EffectMoveSlice());
```



4.7.7 PlotListener :

4.7.7.1 API description

You can add your own listener to chart event. To do so, extend the Plot-MouseListener class.

```
PlotMouseListener listener = new PlotMouseListener() {
        public void processEvent(EffectEvent event) {
                //Process event as you wish
        }
};
```

The EffectEvent contains chart specific events.

It provides several methods describing the event.

public void setFillColor (String color)
public void setStrokeColor(String color)
public void setStrokeWidth(int width)
public void setMarkerType(String marker)
public void setMaxDisplayed(double max)
public void setMinDisplayed(double min)

getType(): This method returns the type of the event. It will return one of the following constants :

```
final public static String TYPE_ONCLICK = "onclick";
final public static String TYPE_ONMOUSEOVER = "
    onmouseover";
final public static String TYPE_ONMOUSEOUT = "
    onmouseout";
```

getElementType(): This method returns the type of the element which fired the event. It will return one of the following constants :

```
final public static String ELEMENT_TYPE_MARKER = "
    marker";
final public static String ELEMENT_TYPE_BAR = "bar";
final public static String ELEMENT_TYPE_COLUMN = "
    column";
final public static String ELEMENT_TYPE_CIRCLE = "
    circle";
final public static String ELEMENT_TYPE_SLICE = "
    slice";
```

getX(): this method returns the x-coordinate of the point which fired the event.

getY(): this method returns the y-coordinate of the point which fired the event.

getAssociatedObject(): It will return a representation of the element. This representation is :

- a PiePiece object if the element which fired the event was a slice
- a Point object if the element which fired the event was a marker
- a Bubble object if the element which fired the event was a circle
- a Double if the element which fired the event was a Bar or a column

4.7.7.2 Example/Tutorial

```
FlowPanel panel = new FlowPanel();
panel.setStylePrimaryName("testChartCustomEffectPage-
   chartAndLabelsContainer");
Chart2D chart = new Chart2D("300px","300px");
PiePlot<PiePiece> plot = new PiePlot<PiePiece>();
final HTML onMouseOver = new HTML();
final HTML onMouseClick = new HTML();
final HTML onMouseOut = new HTML();
Serie<PiePiece> serie1 = new Serie<PiePiece>();
serie1.setName("continents");
serie1.addData(new PiePiece(30370000,"Africa","#FFFFFF"))
   ;
serie1.addData(new PiePiece(42330000,"America"));
serie1.addData(new PiePiece(13720000,"Antarctica"));
serie1.addData(new PiePiece(43810000 ,"Asia"));
serie1.addData(new PiePiece(9010000,"Australia"));
serie1.addData(new PiePiece(10180000,"Europe"));
plot.addEffect(new EffectTooltip());
PlotMouseListener listener = new PlotMouseListener() {
        public void processEvent(EffectEvent event) {
                String label ="";
                if(EffectEvent.ELEMENT_TYPE_SLICE.equals(
                   event.getElementType())){
                        label = ((PiePiece)event.
                           getAssociatedObject()).
                           getLabel();
                }
                if(event.getType().compareTo(EffectEvent.
                   TYPE_ONCLICK) == 0){
                        onMouseClick.setHTML("Clicked␣on␣
                           :␣" + label);
                }else if(event.getType().compareTo(
                   EffectEvent.TYPE_ONMOUSEOVER) == 0){
                        onMouseOver.setHTML("Mouse␣Over␣:
                           ␣" + label);
                }else if(event.getType().compareTo(
                   EffectEvent.TYPE_ONMOUSEOUT) == 0){
```

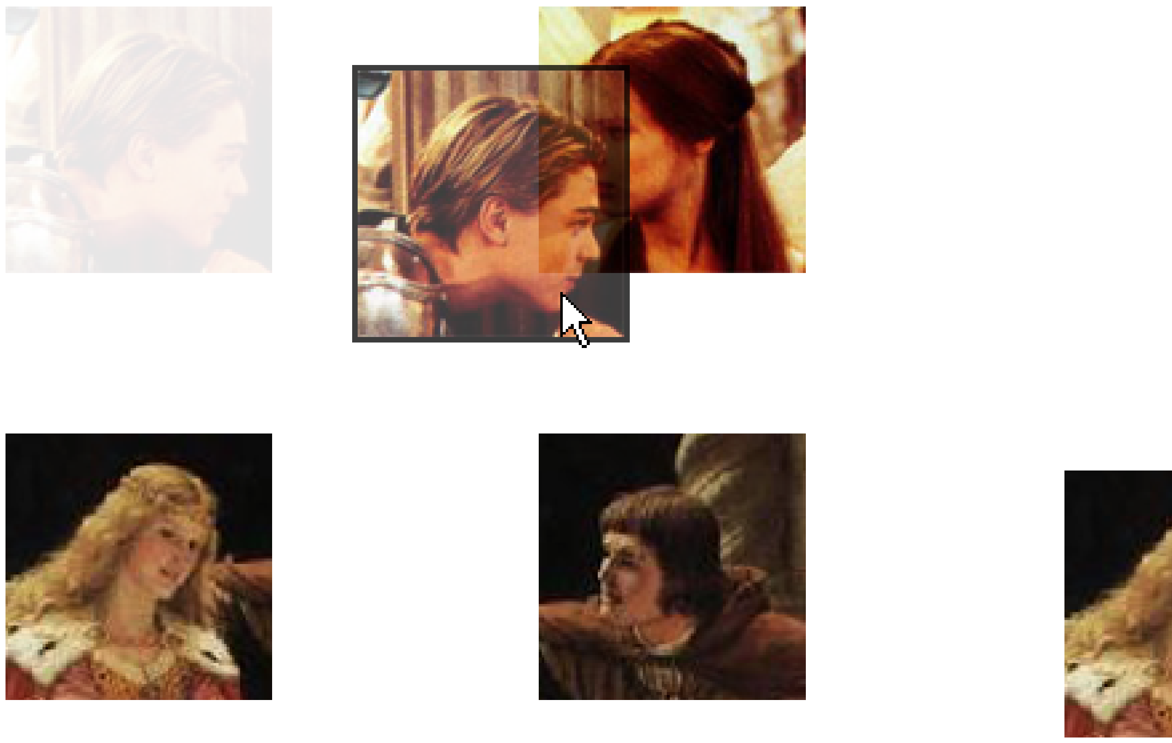
```
                                onMouseOut.setHTML("Mouse␣Out␣:␣"
                                     + label);
                        }
            }
};
plot.addEffect(listener);
plot.addSerie(serie1);
chart.addPlot(plot);
DOM.setElementAttribute(chart.getElement(),"id","chart1")
    ;
DOM.setElementAttribute(onMouseOver.getElement(),"id","
    chart1MouseOver"); DOM.setElementAttribute(
    onMouseClick.getElement(),"id","chart1MouseClick");
    DOM.setElementAttribute(onMouseOut.getElement(),"id","
    chart1MouseOut");
panel.add(chart);
panel.add(onMouseClick);
panel.add(onMouseOver);
panel.add(onMouseOut);
```

Chapter 5

Drag And Drop Features

5.1 YUI Drag And Drop

The drag-and-drop component is a panel in which widgets are allowed to be ... dragged and dropped. Any kind of widget can be used, the panel is entirely responsible for adding the drag an drop capability



The drag-and-drop component relies on the following concepts:

- The capacity for a widget to be dragged
- The capacity for a widget to be a drop target
- The compatibility between the dragged component and the component on which it is dropped.

The compatibility tells whether a (dragged) widget can be dropped on another one (the target). The first widget has to be declared as a draggable widget to the panel for the given compatibility, and the other widget has to be declared as a target for the same compatibility. A compatibility is just a string.

A given widget can be draggable and a target at the same time. It can be involved in more than one compatibility rule. For instance, a widget w1 could be target for a widget w2 according to the compatibility "comp1", and w1 could also be dropped on yet another widget w3 according to the compatibility "comp2". The code corresponding to this example is:

```
DragAndDropPanel ddPanel = new DragAndDropPanel();
Widget w1 = (some widget creation code)
Widget w2 = (some widget creation code)
Widget w3 = (some widget creation code)
ddPanel.addTargetWidget(w1, 50, 50, "aff1");
ddPanel.addDraggableWidget(w2, 100, 100, "aff1");
ddPanel.setDraggable(w1, "aff2");
ddPanel.addTargetWidget(w3, 50, 50, "aff2");
```

If a draggable widget happens to be dropped on a target widget (or on a non-target widget, or on nothing) with which it shares no compatibility at all, the dragged widget goes back to its original position.

This system is quite flexible and allows about any combination:

- A widget can be draggable and target for the same compatibility, or for different compatibilities.
- A widget can be draggable or target of several compatibilities.
- A widget can be added or removed compatibilities dynamically (specifying each time if the compatibility is relative to a draggable capacity or a target capacity).
- A widget for which all compatibilities have been removed will no longer be movable.

Note that the drag-and-drop panel can contain widgets which are neither targets nor draggable.

Drag and drop operations produce events which can be handled programmatically. It is even possible to block a move when compatibility rules are satisfied. Events are dispatched to objects implementing DragAndDropListener interface, and registered in the panel. The next section of this document presents this interface in details.

5.1.1 Building a simple drag-and-drop component

This part is a mini tutorial on main APIs available for the drag-and-drop component. Let's begin with the instantiation of a drag-and-drop panel:

```
import com.objetdirect.tatami.client.DragAndDropPanel;
DragAndDropPanel ddPanel = new DragAndDropPanel();
```

This panel can then be resized to the required space for drag and drop operations, and attached somewhere in the GWT component tree:

```
ddPanel.setSize("500px", "400px");
RootPanel.get().add(ddPanel, 0, 0);
```

A draggable widget can be added via the addDraggableWidget method as illustrated:

```
Image romeo = new Image("romeo.png");
ddPanel.addDraggableWidget(romeo, 50, 50, "romeo&juliet")
    ;
```

The upper left border of the Romeo picture is at position 50, 50. Romeo is compatible with “romeo&Juliet”.

The Romeo widget can be moved, but any drop leads to a return to its original position for there are no compatible target with “romeo&Juliet” so far.

Let's create the Juliet widget and declare it as a possible target for Romeo. The method addTargetWidget should be used for that purpose:

```
Image juliet = new Image("juliet.png");
ddPanel.addTargetWidget(juliet, 250, 50, "romeo&juliet");
```

Now you can try to move Romeo onto Juliet: this time Romeo will not return back to its original position. It stays with Juliet. However, Juliet does not move at all.

Let's try now with two new pictures of Tristan and Iseult:

```
Image tristan = new Image("tristan.png");
ddPanel.addDraggableWidget(tristan, 50, 250, "tristan&
    iseult");
Image iseult = new Image("iseult.png");
ddPanel.addTargetWidget(iseult, 250, 250, "tristan&iseult
    ");
```

Now, Romeo can still be dropped on Juliet, but not on Iseult, and conversely for Tristan.

We are going to create a new picture, named Dom Juan, which is able to join any “feminine” widget. Therefore it is a draggable widget, compatible with both “romeo&Juliet” and "tristan&iseult". The method addDraggableWidget will be used for declaring Dom Juan compatible with “romeo&Juliet” and add it to the panel. The next step consists in invoking setDraggable to provide Dom Juan with the "tristan&iseult" compatibility:

I

```
mage domjuan = new Image("domjuan.png");
ddPanel.addDraggableWidget(domjuan, 50, 450, "romeo&
    juliet");
ddPanel.setDraggable(domjuan, "tristan&iseult");
```

Now Dom Juan can be dropped either on Juliet as well as on Iseult. However, Dom Juan would not accept to be dropped on Romeo nor Tristant, as those widgets are not declared as targets.

Unfortunately, Iseult and Juliet can not by themselves join their lovers. We are going to improve this by declaring Romeo and Tristan as target widgets, and Juliet and Iseult as draggable widgets.

For that purpose, the setDraggable method (we already know) will be used and its target analogous, setTarget. Those methods can provide both capacities (draggable and target) to the same widget:

```
ddPanel.setDraggable(juliet, "romeo&juliet");
ddPanel.setDraggable(iseult, "tristan&iseult");
ddPanel.setTarget(romeo, "romeo&juliet");
ddPanel.setTarget(tristan, "tristan&iseult");
```

Moreover, the setTarget method can be used to add more than one target compatibility to the same widget. For the purpose of illustration, and simplifying at the same time Dom Juan coding, we will add the “woman” compatibility to all female widgets. Then Dom Juan will just have to be draggable for that compatibility:

```
ddPanel.setTarget(juliet, "woman");
ddPanel.setTarget(iseult, "woman");
Image domjuan = new Image("domjuan.png");
ddPanel.addDraggableWidget(domjuan, 50, 450, "woman");
```

Now let’s see it is possible to add widgets which are neither draggable nor targets:

```
Image tintin = new Image("tintin.png");
ddPanel.add(tintin, 450, 450);
```

A widget, whatever its status with respect to the Drag-and-drop capabilities, can always be moved with the method setWidgetPosition. In that case, the compatibility is not checked and the move is done in any case.

```
ddPanel.setWidgetPosition(tintin, 500, 500);
```

Actually, the DragAndDropPanel is a subclass of AbsolutePanel, and as such all methods available on the latter class are also available for the former safely.

5.1.2 Receiving and handling drag and drop events

When a drag and drop event has been accepted by the panel, i.e. the compatibilities between the source and the target widget are in correspondence, this event can of course be intercepted. When the two widgets are not compatible, the dragged widget returns back to its original location without any event being triggered. A drop event can even be rejected when the panel has already accepted it.

The handling of drag and drop events requires an object which implements the DragAndDropListener interface. The following two methods shall be implemented:

```
public boolean acceptDrop(Widget draggable, Widget target
    ) { }
public void onDrop(Widget draggable, Widget target) { }
```

The acceptDrop method tells whether the drag and drop operation has been agreed for. The onDrop method is called when the drag and drop operation has definitely been accepted. This is the method where the event can be handled.

Both methods receive the dragged and the target widget involved in the operation. The method addDragDropListener of the panel allows the registering of a drag and drop listener. The removing of a listener can be done via the removeDragDropListener method.

```
DragAndDropListener listener = new DragAndDropListener()
    {
        public void onDrop(Widget draggable, Widget
            target) { }
        public boolean acceptDrop(Widget draggable,
            Widget target) { return true; }
};
ddpPanel.addDragDropListener(listener);
ddpPanel.removeDragDropListener(listener);
```

5.1.3 Adding and removing compatibilities

A compatibility (either draggable or target) can be added and removed at any time. Whenever a widget loses all its compatibilities, it can no longer be a target or draggable, according to the compatibility type. For instance, a widget for which all draggable compatibilities has been removed can no longer be moved.

The method unsetDraggable of the panel can be used to remove a draggable capability to a widget:

```
public void unsetDraggable(Widget widget, String
    affordance);
```

The method unsetTarget of the panel can be used to remove a target compatibility to a widget:

```
public void unsetTarget(Widget widget, String affordance)
    ;
```

All compatibilities of the same capacity can be removed in one operation. Methods with the same names as the afore mentioned ones are available for that purpose, with the second parameter being discarded.

For instance the method:

```
public void unsetDraggable(Widget widget);
```

removes the draggable capability for the widget, and the method:

```
public void unsetTarget(Widget widget);
```

removes the target capability for the widget.

5.1.4 Getting information from the drag-and-drop panel

More methods are available on the drag-and-drop panel which give access to information relative its contents.

The isDraggable method tells whether a widget is draggable (i.e. for at least one compatibility):

```
public boolean isDraggable(Widget widget);
```

The hasDraggableAffordance method tells whether a given widget is draggable for a given compatibility:

```
public boolean hasDraggableAffordance( Widget widget,
    String affordance);
```

The getDraggableAffordances method returns the list of all draggable compatibilities for a given widget (the compatibilities relative to the target capacity of the widget, if any, are not in that list):

```
public String[] getDraggableAffordances(Widget widget);
```

The getDraggableAffordanceSet method return the list of draggable widgets for the given compatibility:

```
public Widget[] getDraggableAffordanceSet(String
    affordance);
```

Similar methods are available for the target capacity. The isTarget method tells whether a given widget is a target (i.e. for at least one compatibility):

```
public boolean isTarget(Widget widget);
```

The hasTargetAffordance method tells whether a given widget is a target for a given compatibility:

```
public boolean hasTargetAffordance( Widget widget, String
     affordance);
```

The getTargetAffordances methods returns the list of all target compatibilities for a given widget ((the compatibilities relative to the draggable capacity of the widget, if any, are not in that list):

```
public String[] getTargetAffordances(Widget widget);
```

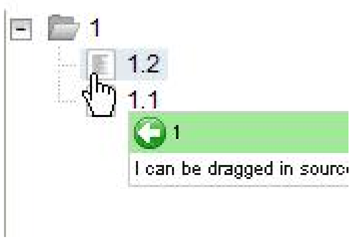
The getTargetAffordanceSet method returns the list of target widgets for the given compatibility:

```
public Widget[] getTargetAffordanceSet(String affordance)
    ;
```

5.2 Dojo Drag And Drop

Dojo drag and drop provides a powerful mechanism to perform drag and drop. It introduces the following concepts:

- A DND source is a widget which contains movable elements, which are called DNDElement
- A DND target is a widget which will potentially accept dropped DNDElement
- A DNDElement is a movable object, within a DND source
- A DNDBehavior process drag and drop events for a given list of source-target couples.



5.2.1 Getting Started

We will demonstrate a simple DnD use. This example will define a DnDSource, a DnDTarget and add some behaviours to those.

5.2.1.1 Defining the Source

First, we define a drag and drop source. There are two DNDSource types in tatami:

- WidgetSource: they are constructed from a Panel, and allow any Widget to be dragged.
- TreeSource: they allow tree items to be dragged and to accept drop events.

But you do not have to construct these objects yourself, since the DnD helper do all this for you.

We begin by defining a standard GWT panel, and adding widgets to it :

```
VerticalPanel sourcePanel = new VerticalPanel();
HTML widget1 = new HTML("I can be dragged");
HTML widget2 = new HTML("I can be dragged too");
sourcePanel.add(widget1);
sourcePanel.add(widget2);
```

Then, we register this panel as a source panel, and the widgets it contains as dnd element within these source.

```
DnD.registerSource(sourcePanel);
DnD.registerElement(sourcePanel,widget1);
DnD.registerElement(sourcePanel,widget2);
```

We are now able to move widget1 and widget2 across the screen, but nobody is there to accept them.

WARNING: If you set a DOM id to your widget element, be sure to assign your id BEFORE registering this element, since dojo uses the dom id as a dnd id. Assigning an ID to a DnDElement after it has been registered will make your element unavailable in your behavior.

5.2.1.2 Defining the Target

Similarly,we will define a DnD Target.

```
VerticalPanel targetPanel = new VerticalPanel();
HTML widget3 = new HTML("I cannot be dragged");
HTML widget4 = new HTML("Me neither");
targetPanel.add(widget3);
targetPanel.add(widget4);
DnD.registerTarget(targetPanel);
DnD.registerElement(targetPanel,widget3);
DnD.registerElement(targetPanel,widget4);
```

5.2.1.3 Defining a behavior

Now we must define a behavior to be applied to those drag and drop events.

There are some Behavior classes defined. See the behavior section for a complete description of each.

Here, we will use the WidgetDnDBehavior, which is the best suited to manage dragging a widget from one panel to another.

We will override the onDrop method only, to move the dropped widget from the source to the target.

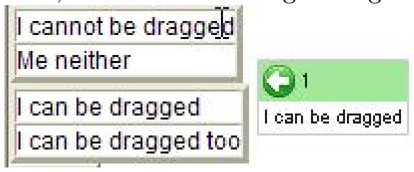
```
WidgetDnDBehavior myBehavior = new
    DnDDefaultWidgetBehavior() {
        @Override
        public boolean onDrop(Collection<Widget>
            draggedWidgets, Panel source, Panel target,
            String targetNodeId, boolean isCopy) {
                for (Widget widget : draggedWidgets) {
                        DnD.move(widget, source, target);
                }
                return true;
        }
};
```

Then, we register this behavior for our specifics sources and target:

```
try {
        DnD.registerBehavior(myBehavior, sourcePanel,
            targetPanel);
} catch (BehaviorScopeException e) {
}
```

Registering a behavior for a source/target couple may raise an exception if a behavior is already defined for these couple.

Now, we are able to drag a widget from the source panel to the target:



You may also drag multiple draggable items by clicking them one after the other, holding the shift key inbetween.

5.2.2 DndSources

A dnd source acts as a container for Dnd elements. It is registered to an associated DnDController, which is designed to manage this particular type of source.

Tatami currently provides two types of dnd sources :

- WidgetSource: they are constructed from a Panel, and allow any Widget to be dragged.
- TreeSource: they allow tree items to be dragged and to accept drop events.

You normally do not have to instantiate those objects yourself, but you will be able to manipulate them when using a DnDBehavior.

If you want to use a widgetSource, just use the following method

```
DnD.registerSource(Panel panel);
```

Then, you can add register widgets contained in this panel as DndElement:

```
DnD.registerElement(Panel,Widget);
```

If you want to use a TreeSource, use the following method:

```
DnD.registerTreeSource(Tree);
```

You do not have to register any DndElement for a tree, since all TreeItems are necessarily DnDElement in a tree registered as a tree source.

WARNING: If you set a DOM id to your widget element, be sure to assign your id BEFORE registering this element, since dojo uses the dom id as a dnd id. Assigning an ID to a DnDElement after it has been registered will make your element unavailable in your behavior.

5.2.3 DndTarget

A DndTarget is defined as a container wich may accept drop operations

A DndTarget is registered with one of the following methods:

```
DnD.registerTarget(Tree);
DnD.registerTarget(Panel);
```

Respectively, for a panel or a tree.

5.2.4 DnDBehavior

A DndBehavior is responsible for processing drag and drop events from a set of sources to a set of target.

5.2.4.1 IDnDBehavior interface

```
public interface IDnDBehavior<E extends IDnDElement , S
    extends IDnDSource<?> , T extends IDnDTarget> {
        public boolean onDrop(Collection<E> dndElements,
            S source, T target, String targetNodeId ,
            boolean isCopy );
        public void elementsAccepted(S source ,T target,
            Collection<E> elements, boolean copied,
            IDnDController<?, T> controller);
        public void dragOver(IDnDTarget target);
        public boolean checkItemAcceptance(S source , T
            target , Collection<E> dndElements);
        public void onDndStart(Collection<E>
            elementBeingDragged , S source , boolean
            ctrlKey);
}
```

This interface is generic: it can be implemented for a particular type of DnDElement, a particular type of DnDSource and a particular type of Target.

That means you will be able to implement a behavior for WidgetDnDElement

Each of those methods are called during the drag and drop process.

onDnDStart: it will be called each time a drag operation is originated from one of the sources the behavior has been registered for.

checkItemAcceptance: it will be called to verify that a target will likely accept a drop operation from a source, for a given set of elements, when the user is dragging any element from one one of the sources the behavior has been registered for to a target. This will result in UI changes: the drag and drop avatar following the mouse will be coloured in red or green according to what has been returned from this call.

dragOver: it will be called each time the user is dragging an item over the registered target

onDrop: it will be called each time the user tries to drop an item from a registered source to a registered target

elementsAccepted: it will be called if the onDrop operation returned true.

5.2.4.2 AbstractBehavior & DnDGenericBehavior

Tatami provides two abstract class which makes the IDnDBehavior interface more easy to implement

AbstractDnDBehavior just implements IDnDBehavior with defaults "do-nothing" implementations.

```
public abstract class AbstractDnDBehavior<E extends
    IDnDElement,S extends IDnDSource<? super E>,T extends
    IDnDTarget> implements IDnDBehavior<E, S, T>
```

So you can override only the methods you want from the AbstractDnDBehavior.

Ex:

```
AbstractDnDBehavior<WidgetDnDElement, WidgetSource,
    WidgetTarget> be = new AbstractDnDBehavior<
    WidgetDnDElement, WidgetSource, WidgetTarget>(){
        @Override
        public boolean onDrop(Collection<WidgetDnDElement
            > dndElements, IDnDSource<? super
            WidgetDnDElement> source,
                            IDnDTarget target, String
            targetNodeId, boolean isCopy) {
                                Window.alert("You dropped
                                    on item with id : " +
                                    targetNodeId);
                                return true;
        }
};
```

DnDGenericBehavior provides is another default implementation which is designed to use with a particular type of element, any DnDSource which can handle it and any DnDTarget.

5.2.4.3 DefaultWidgetDnD Behavior

The WidgetDnDBehavior class is an abstract class designed to make it easier to implement DnD behavior for widgets. You just have to override at your convenience the following methods:

```
public boolean checkItemAcceptance( Panel source,Widget
    target, Collection<Widget> draggedWidgets) { return
    true; }
public void elementsAccepted(Panel source, Widget target,
     Collection<Widget> draggedWidgets, boolean copied) {
    }
public boolean onDrop(Collection<Widget> draggedWidgets,
    Panel source, Widget target, String targetNodeId,
    boolean isCopy) { return true; }
public void onDndStart(Panel source, Collection<Widget>
    draggedWidgets, boolean copied ){
}
public void onCancel(){
}
public void dragOver(Widget target){
}
```

Chapter 6

Encoding

Dojo offers APIs for encoding data in various formats. Tatami brings this feature to the GWT world by wrapping those APIs.

6.1 BlowFish encryption

BlowFish is a symmetric cipher algorithm, relying on a private key that must be shared amongst people wanting to communicates ciphered data.

Tatami provides a simple API to use the BlowFish algorithm in GWT.

Sample :

```
BlowFishEncryption cipher = BlowFishEncryption.
    getInstance();
String key = "the␣secret␣key";
String randomString = "this␣text␣should␣remain␣unrevealed
    !␣";
String ciphered = cipher.encrypt(BlowFishEncryption.
    Base64OutputType, randomString, key);
String deciphered = ciher.decrypt(BlowFishEncryption.
    Base64OutputType, ciphered, key);
```

In the above sample, we use a secret key ("the secret key") to encrypt a text. Tatami currently support two output-types/input types for the ciphering/deciphering process.

Those are :

- Base64 : the output will be encoded in Base64 format.
- Hex : the output will be encoded in Hex format

Those formats are defined as constants in the BlowFishEncryption class (respectively BlowFishEncryption.Base64OutputType and BlowFishEncryption. HexOutputType)

While using the BlowFish encryption , you should assure which output format comes from the cipher, and which format waits the decipher.

getInstance(): Returns an instance of the BlowFishEncryption

encrypt(EncodingTypeConstant outputTypeConstant,String toCrypt,String key): Encrypt the given text using the given key. The output format is defined by outputTypeConstant, which can be either BlowFishEncryption. Base64OutputType or BlowFishEncryption.HexOutputType

decrypt(EncodingTypeConstant inputTypeConstant,String toDecrypt,String key): Decrypt the given text using the given key. The input format is defined by inputTypeConstant, which can be either BlowFishEncryption. Base64OutputType or BlowFishEncryption.HexOutputType

6.2 MD5 hash

6.2.1 Introduction

MD5 has been used for a long time as an asymmetric hash algorithm. Tatami gives users a convenient way of achieving MD5 hash.

Sample :

```
MD5 md5 = MD5.getInstance();
String text = "text";
String resultingMD5 = md5.encode(text , MD5.HexOutputType
    );
```

The resulting md5 will be equal to “1cb251ec0d568de6a929b520c4aed8d1” in that case. MD5 sum results are usually given in Hexadecimal format (just like above), but other output types are defined :

MD5.Base64OutputType: The result will be encoded in Base64 format MD5

StringOutputType: The result will be encoded as a String representing the binary data MD5.

HexOutputType: the result will be encoded in Hexadecimal format

6.2.2 API

getInstance(): Returns an instance of the MD5

encode(String toEncode,EncodingTypeConstant encodingtype): Encode the given text. The resulting sum will be encoded according to the specified EncodingTypeConstant.