

This article is archived because it is not considered relevant for third-party developers creating commercial solutions today. The article is believed to be still valid for the original topic scope.



## Contents

- [1 Scope](#)
- [2 Widgets](#)
- [3 The API](#)
- [4 Understanding the Widget life cycle](#)
- [5 Displaying content](#)
  - ◆ [5.1 Shell](#)
  - ◆ [5.2 Flow](#)
  - ◆ [5.3 Canvas](#)
- [6 Creating Views Using The widget.xml File](#)
- [7 Adding Softkey Actions](#)
  - ◆ [7.1 Using The Default Callback](#)
  - ◆ [7.2 Overwriting The Default Callback](#)
- [8 Adding a Menu](#)
  - ◆ [8.1 Using The Default Callback](#)
  - ◆ [8.2 Using The Customer-Defined Callback](#)
- [9 Handling Events](#)
- [10 Organizing The Code](#)
- [11 Example](#)
- [12 See also](#)

## Scope

This article shows how a widget application works and gives basis about how to start up. Although the nature of a widget is to be connected somehow to the Internet, the scope of this document is just to show how to get started, create some screens and handle events. There is an example at the end of the article enclosing all the concepts viewed.

## Widgets

Widgets are the small applications placed on the widget dashboard.

A widget is composed of three elements:

- **Widget.xml** where specific information of the widget is defined

## Understanding\_Widgets

- **File.he** where the source code is placed in WSL language
- **Resources** could be text files, css file or images

The Widget.xml file has a defined structure, summarized bellow.

- **Info:** contains general metadata about the widget
- **Parameters:** defines some parameters that can be used by the application
- **Resources:** enumerates the resources such as images, the script file and style sheets
- **Layout:** allow to create views using parameters in the xml file

To understand better how to write the widget.xml file, follow this link: [WIDGET.xml FILE](#)

## The API

The [[Widgets API](#)] is in many ways similar to Java (tm). There are a lot of important classes, but this article will address one important class and one Interface that comprise the behavior of the WidSet application.

The first class is called [[API](#)]. This class contains a set of methods and constants that can be invoked directly in any place of your code (without creating any instance). It includes mathematical operations such as sin, max value, min value, etc; access to remote hosts; display control and many other operations.

The [[Script](#)] interface defines a set of **callback methods**. A **callback method** is called directly by the application when a particular event occurs, that means you don?t need to invoke that method in your code. At the moment the event occurs, the application tries to find the callback method in your code, if have implemented it then will be executed. If you want to react to a specific event, all you have to do is implementing the callback just as it appears in the Script interface.

## Understanding the Widget life cycle

When the widset is uploaded, a default icon is placed on the dashboard.

When a widset is loaded on the dashboard, the application invokes the callback *startWidget( )* .

It is a good idea to customize the look of the application?s icon by implementing that method just as shown bellow:

```
void startWidget ()
{
    setMinimizedView(createMinimizedView("viewMini", null));
}
```

The method *createMinimizedView(...)* creates a UI container from the widget.xml file where the icon is customized. The code in the xml file would be something like this:

```
<layout minimizedheight="65sp">
  <view id="viewMini">
    <decorate class="minimized.area"/>
    <text class="minimized">My Widget</text>
  </view>
</layout>
```

As seen, it is possible to use a decoration style from the same xml file and add some text to it. Further information about creating views will be shown later.

When the user hits the application icon on the dashboard, the method *openWidget( )* will be called. That method needs to be implemented and return a Shell (a UI container discussed later) that will be placed on the **Shell stack**. The **shell stack** controls what is displayed on the screen. What is on top of the **shell stack** is actually displayed to the user.

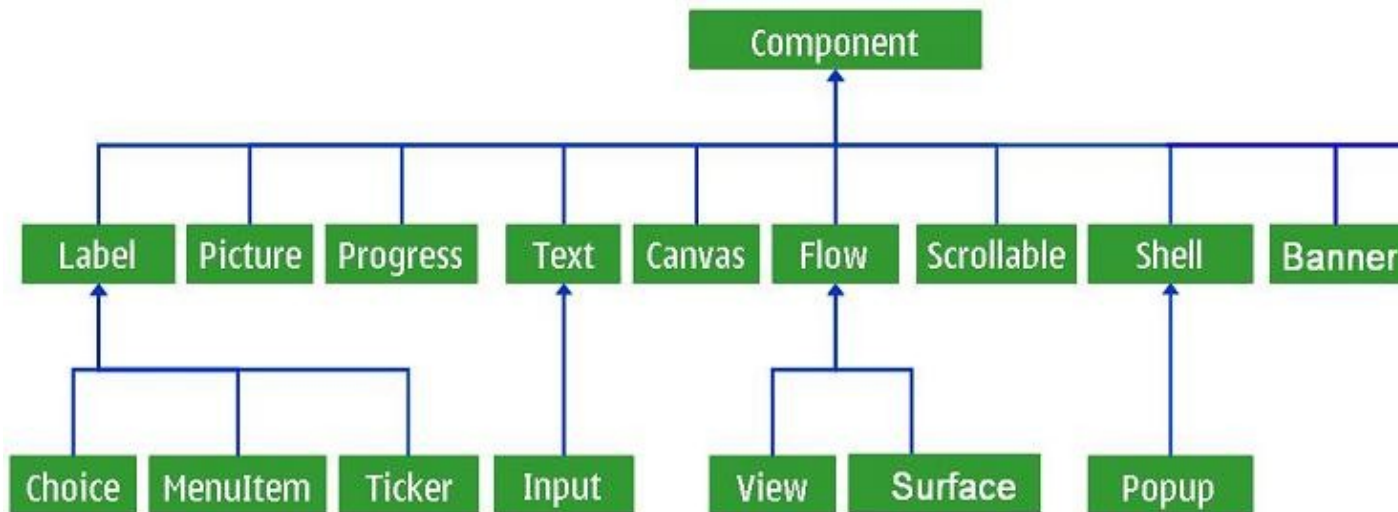
You can add and remove shells into the shell stack by using directly the following methods:

- pushShell(Shell shell)
- popShell(Shell shell)
- slideIn(int x, int y, int w, int h, Shell shell)
- slideOut(int x, int y, int w, int h, Shell shell)

When the current shell is removed, the last one in the **Shell Stack** will be shown. That's a good feature to implement back screen functionality.

## Displaying content

The class **Component** represents any element of the User Interface that can be displayed. Every **Component** can have its own *style* loaded from the .xml file or an external file. That makes the UI very flexible and gives the possibility to customize every single element of it. In the image herebelow, the different types of **Components** are shown:



Each Component has a different behavior, the most useful ones will be addressed:

## Shell

A [\[Shell\]](#) is a component that can be pushed to the **Shell Stack**, the [\[Shell\]](#) on the top will be shown on the screen. A Shell allows any **Component** to be displayed. A [\[Popup\]](#) is a type of Shell and therefore can also be added to the **Shell Stack**.

## Flow

A [\[Flow\]](#) is a **Component** that allows packing several **Components** into a single one. For adding a **Component** to the package the following method is used

```
f.add(Component child)
```

Where f is the instance of any Flow.

Each **Component** added to the Flow has a set of flags or properties that can be modified to affect the layout or allowing to traverse the Component. These flags are

- VISIBLE , LINEFEED , WRAP, FOCUSABLE

By default a Component is **VISIBLE**, has a **WRAP** layout and is **NON FOCUSABLE**. These values are changed with the method *setFlags(...)*. For instance:

```
comp.setFlags(VISIBLE|FOCUSABLE|LINEFEED);
```

where comp is the instance of any Component (Label, Ticker, Choice, etc..).

Basically, It is possible to add any kind of **Component** to a **Flow**. The ones that might be of interest are:

- Label, Ticker, Choice, Picture, Input, Text, Banner, HyperText

## Canvas

A [Canvas] is a Container painted by script, represents a blank rectangular area of the screen onto which the application can draw.

## Creating Views Using The widget.xml File

A [View] is a type of **Flow** whose properties can be loaded directly from the *widget.xml* file. To create a View, the following method is used directly:

```
createView(String name, Object context)
```

The [View] has to be specified in the *widget.xml* file, in the **<layout>** section. For instance:

```
<layout minimizedheight="65sp">
  <view id="viewMaxi">
    ?.
  </view>
</layout>
```

A view in the xml file can have different elements, these are:

- **Label**: Single text line truncated if it doesn't fit the size of the view.
- **Text**: Span in multiple lines
- **Decorate**: defines a style to the view from the **<style>** section in the xml file.
- **Img**: image loaded to the view (should be specified in the **<resources>** section of the .xml file)
- **Script**: allows to add a Component to the view from the sourcecode file (.he). This tag has the following schema:

```
<script id="myView" class="maximized"/>
```

When the **<script>** element is present, the application will invoke in the .he file the callback

```
Component createElement(String viewName, String elementId, Style style, Object context)
```

The **viewname** parameter is the **id** attribute of the **<view>** tag, the **elementId** is the **id** of the **<script>** tag. The returned **Component** will be added to the **View** in construction. The callback will be invoked per each **<script>** element found in the **<view>** section, allowing several **Components** to be added.

Notice that this callback is shared by all the different **Views** defined in the widget.xml file.

## Adding Softkey Actions

A Widget can have access to the three softkeys of the handset; each one has a value associated:

- Right softkey: **SOFTKEY\_BACK**
- Fire softkey: **SOFTKEY\_MIDDLE**
- Left Softkey: **SOFTKEY\_OK**

There are two different ways to associate softkeys to a **Shell**: using the default callback or using a customer-defined callback.

## Using The Default Callback

Before the Shell is displayed, the program will invoke the callback:

```
MenuItem getSoftKey(Shell shell, Component focused, int key)
```

This method will be called automatically by the application several times, each time a different key value will be passed: **SOFTKEY\_BACK**, **SOFTKEY\_MIDDLE** and **SOFTKEY\_OK**. To associate a softkey to an action it is needed to return a **MenuItem** according to the key received.

A [**MenuItem**] is an element that simply encapsulates an action. It is composed by an **id** and a **label** to be displayed. The **Action id** is any number used to recognize the action.

See the following example:

```
//First define the Action Ids.  
  
const int CMD_BACK=100;  
const int CMD_SELECT=101;  
const int CMD_OK=102;  
  
// the getSoftKey method will be something like this  
  
MenuItem getSoftKey(Shell shell, Component focused, int key){  
    if(key==SOFTKEY_OK){  
        return new MenuItem(CMD_OK, "Ok");  
    }  
    else if (key ==SOFTKEY_MIDDLE)  
    {  
        return new MenuItem(CMD_SELECT,"Select");  
    }  
}
```

## Understanding\_Widgets

```
else if (key == SOFTKEY_BACK)
{
    new MenuItem(CMD_BACK, "Back");
}
else{
    return null;
}
}
```

When null is returned, no action will be associated to the softkey and it won't be displayed.

The **Action Ids** are passed to the application when the softkey is pressed to an event handler callback. (That will be discussed later)

There are some system defined **Action Ids** that will trigger some events handled automatically by the application. These System **Action Ids** are static constants explained bellow:

- **OPEN\_HELP** (value -17): Called by the system suggesting that widget should load and show widget specific help page. The help should be specified in the widget.xml file.
- **OPEN\_HOME** (value -14): Called by the system suggesting that widget should open it's home page
- **OPEN\_MENU** (value -1): Called by the system suggesting that widget should open the Menu associated to the Shell being displayed. When the action is executed (the softkey is pressed) the menu handler callback will be called. (Discussed later)
- **OPEN\_SETTINGS** (value -12): Called by the system suggesting that widget should open it's settings for editing.

There's no need to redefine the System **Action Ids**, just use them directly.

For instance, to show the screen menu when the left softkey is pressed and the help screen when the right is pressed the code will be as shown bellow:

```
MenuItem getSoftKey(Shell shell, Component focused, int key){
    if(key==SOFTKEY_OK){
        return new MenuItem(OPEN_MENU, "Menu");
    }
    else if (key == SOFTKEY_BACK)
    {
        new MenuItem(OPEN_HELP, "Help");
    }
    else{
        return null;
    }
}
```

## Overwriting The Default Callback

The `getSoftKey(...)` callback is invoked everytime a Shell is placed on the screen; It is possible to know which Shell is being displayed by the parameter the application passes to this method. If the plan is to use the same commands for every screen, then it is a good idea to use this method, but if the plan is to use different commands in each screen, then the code will start to be complex and may be unreadable. It is possible to avoid this by overwriting the `getSoftKey(...)` callback using a customer-defined callback; then it will be possible to create a method per screen. If a customer-defined callback is created, the default one is not invoked.

This is the way a customer-defined callback is created:

```
myShell.setSoftKeyHandler(softKeyHandler);

MenuItem softKeyHandler(Shell shell, Component focused, int key){

    if(key==SOFTKEY_OK){
        return new MenuItem(OPEN_MENU, "Menu");
    }
    else if (key ==SOFTKEY_MIDDLE)
    {
        return new MenuItem(CMD_SELECT, "Select");
    }
    else if (key == SOFTKEY_BACK)
    {
        return new MenuItem(CMD_BACK, "Back");
    }
    else{
        return null;
    }
}
```

Where `myShell` is the instance of a Shell class.

As you see, the customer-defined callback has the same structure that the default callback, just the name changes.

## Adding a Menu

Adding a Menu to the displayed screen is very similar to adding softkeys. There are also two different ways to do it, a default callback and a user customer-defined callback.

## Using The Default Callback

The default callback has the following schema as stated in the Script interface:

```
Menu getMenu(Shell shell, Component focused)
```

## Understanding\_Widgets

The default callback is invoked when no used defined callback is present and a softkey with **Action Id** equal to **OPEN\_MENU** is pressed by the user.

The method should return a **[Menu]** Object. A **[Menu]** is a set of **[MenuItem]** each one of them with an **Action Id**. It is also possible creating submenus by adding MenuItems to a parent MenuItem. See the example bellow taken from the Widgets API

```
Menu m = new Menu()
    .add(CMD_OPEN, "Open")
    .add(CMD_REMOVE, "Remove")
    .add(CMD_ADD, "Add")
    .begin(CMD_MORE, "More")
    .add(CMD_SYNC, "Synchronize")
    .add(CMD_SETTINGS, "Settings")
    .end()
    .add(CMD_EXIT, "Exit")

Menu getMenu(Shell shell, Component focused)
{
    return m.reset().enable(CMD_OPEN, focused != null)
        .enable(CMD_REMOVE, focused != null)
        .enable(CMD_SYNC, isOnline());
}
```

All the **Action Ids** are constants with values defined in the source code as shown before. It is also possible to use system **Action Ids** as described in the last section.

When the **add(...)** method of the menu object is used, it implicitly creates a **MenuItem** associated to that **Menu**.

As you see, the submenu is created within the calls to **begin()** and **end()**.

The menu will have the following appearance.



## Using The Customer-Defined Callback

The second way to add a Menu is with a customer-defined callback. When that is done, only the default callback **won't** be invoked. First, it is needed to register the customer-defined callback and then implement the method.

```
myShell.setMenuHandler(myMenuHandler);

Menu myMenuHandler( Shell shell, Component source){

    Menu m = new Menu().add(CMD_HELP, "Help")
                      .add(CMD_ABOUT, "About")
                      .add(CMD_EXIT, "Exit Menu");

    return m;
}
```

## Handling Events

## Understanding\_Widgets

As in the last two sections, it is possible to handle events either by writing the default callback or writing a customer-defined callback. The following is the schema of the method:

```
void actionPerformed(Shell shell, Component source, int action)
```

This method is called automatically by the application when a softkey with an **Action Id** associated is pressed. The **Action Id** number is passed to the application so it is possible to know what the action triggered was.

As the default callback is shared by all the Shells in the application, it is also possible to create a customer-defined callback. In this case the default callback **won't** be invoked.

To create the customer-defined callback it is necessary to register the method using **setActionHandler()** and implement it as described in the Script interface of the Widgets API just as shown in the example below.

```
myShell.setActionHandler(myActionHandler);

void myActionHandler(Shell shell, Component source, int action){

    if(action==CMD_BACK){

        popShell(shell);

    }

}
```

Where myShell is a Shell instance and CMD\_BACK is any defined **Action Id**. In this very simple case, the application will show the last screen when the CMD\_BACK **Action Id** is triggered.

## Organizing The Code

As the whole source code is contained into a single file, it is possible that adding so many methods and callbacks belonging to different screens start making it complex and unreadable.

A good way to organize the code is to comprise the whole behavior of each screen into a single method, taking advantage that in widget scripting language it is possible to write methods within other methods.

Each screen is usually composed of the following methods and callbacks:

- A method to populate it with Components
- A callback to aggregate Softkey Actions
- A callback to aggregate a Menu

## Understanding\_Widgets

- A callback to handle events

Well, adding customer-default callbacks to the Shell, allows to do it all into a single method just as shown in the following code template:

```
Shell createScreen(){

    Shell myShell=null;
    Flow myFlow= new Flow(getStyle("list"));           //Create a Flow

    // Add components to the flow here

    Shell myShell=new Shell(myFlow);                 //Create a Shell using th

    myShell.setSoftKey(mySoftKeyHandler);           //To add the softkeys to the Shel
    myShell.setMenu(myMenuHandler);                 //To add a Menu to the Shell
    myShell.setAction(myActionHandler);           //To add an action handler to the

    //Implement the customer-defined callbacks

    mySoftKeyHandler(Shell shell, Component focused, int key){

        //Return a MenuItem per softkey

    }

    myMenuHandler( Shell shell, Component source){

        //Construct a menu here

    }

    myActionHandler(Shell shell, Component source, int action){

        //Add Actions Here

    }

    return myShell;
}
```

## Example

The following link provides an example enclosing the concepts reviewed in this article.

[Example: handset review](#)

## See also

- [WidSets SDK](#)
- [Configuring an editor for syntax highlighting](#)
- [WidSets Client](#)
- [Widget examples](#)
  - ◆ [WidClock](#)
  - ◆ [Memory Game](#)
  - ◆ [Filter test](#)
  - ◆ [Hello World](#)
  - ◆ [UITest](#)