

A Maui User's Guide

Paul Boggs
Leslea Lehoucq
Kevin Long
Andrew Rothfuss
Edward Walsh
Robert Whiteside

Abstract:

Maui is a program that automatically builds a GUI for a problem given a high-level specification of the problem's data structures and methods. This guide is intended to show GUI developers how to use Maui to design effective GUIs for applications.

-
- [1. Getting Started](#)
 - [1.1 Installation Instructions](#)
 - [1.1.1 System Requirements](#)
 - [1.1.2 Installing Java](#)
 - [1.1.3 Installing Ant](#)
 - [1.1.4 Downloading MAUI](#)
 - [1.1.5 UNIX/LINUX Installation](#)
 - [1.1.5.1 Installing MAUI on UNIX/LINUX](#)
 - [1.1.5.2 Troubleshooting the installation on UNIX/LINUX](#)
 - [1.1.6 Windows Installation](#)
 - [1.1.6.1 Installing MAUI on Windows](#)
 - [1.1.6.2 Troubleshooting the installation on Windows](#)
 - [1.2 Running Maui](#)

 - [2. How to Design Maui Objects](#)

- [2.1 Introduction](#)
- [2.2 The Basics](#)
- [2.3 Maui Primitives](#)
- [2.4 Classes as Fields](#)
- [2.5 Using Subclasses to Represent Choices](#)
 - [2.5.1 Data Representation](#)
 - [2.5.2 Appearance of Subclasses in the GUI: Tabbed Panes](#)
 - [2.5.3 Labeling in the Subclass Menu](#)
- [2.6 Arrays](#)
 - [2.6.1 The Master Block](#)
 - [2.6.2 Other Array Options](#)
- [2.7 Tables](#)
- [2.8 References](#)
- [2.9 Maui help buttons](#)
- [2.10 Summary of Maui](#)
- [3. Actions](#)
 - [3.1 Introduction](#)
 - [3.2 XML for Specifying an Action](#)
 - [3.2.1 Maui Compressed XML](#)
 - [3.2.2 Maui Verbose XML](#)
 - [3.2.3 Maui Built-In Actions](#)
 - [3.3 Writing Your Own Actions](#)
 - [3.3.1 The XMLObject Class](#)
 - [3.3.2 Example of a Custom Maui Action](#)
 - [3.3.3 Compiling Your Action](#)
 - [3.3.4 Configuring Maui to Find Your Action](#)
 - [3.4 Suggested Exercises](#)
 - [3.5 Summary](#)
- [4. Custom Editors](#)
 - [4.1 Introduction](#)
 - [4.2 Using a Custom Editor: The `FilenameEditor` for Strings](#)
 - [4.3 Writing Your Own Custom Editors](#)
 - [4.3.1 The Structure of Maui Data](#)
 - [4.3.2 Steps to Writing Your Own Custom Editor](#)
 - [4.3.3 Example of a Maui Custom Editor](#)
 - [4.3.4 Compiling Your Custom Editor](#)
 - [4.3.5 Configuring Maui to Find Your Custom Editor](#)

- [4.4 Summary](#)
- [5. Configuring Maui](#)
 - [5.1 Introduction](#)
 - [5.2 Appearance Settings](#)
 - [5.3 Services](#)
 - [5.4 Paths and Packages](#)
 - [5.4.1 Configuring with the Configure Maui Button](#)
 - [5.4.2 The "Do Nothing" Configuration Method](#)
 - [5.4.3 Configuring in Your XML Specification](#)
- [A. Application Example](#)
 - [A.1 A Calore GUI](#)
 - [A.1.1 Differences Between Text and GUI Calore Input](#)
 - [A.1.2 Calore GUI Design Examples](#)
 - [A.1.2.1 Array Example](#)
 - [A.1.2.2 Optional Fields Example](#)
 - [A.1.2.3 Subclassing Example](#)
 - [A.1.2.4 Referencing Example](#)
 - [A.1.2.5 Import Example](#)
 - [A.1.2.6 String Menu Example](#)
 - [A.2 Text Input to Calore](#)
- [B. Maui XML syntax guide](#)
 - [B.1 Tag Maui](#)
 - [B.1.1 Children allowed in Maui elements](#)
 - [B.1.2 Attributes allowed in Maui elements](#)
 - [B.2 Tag Class](#)
 - [B.2.1 Children allowed in Class elements](#)
 - [B.2.2 Attributes allowed in Class elements](#)
 - [B.3 Tag Import](#)
 - [B.3.1 Children allowed in Import elements](#)
 - [B.3.2 Attributes allowed in Import elements](#)
 - [B.4 Tag Fields](#)
 - [B.4.1 Children allowed in Fields elements](#)
 - [B.4.2 Attributes allowed in Fields elements](#)
 - [B.5 Tag AppData](#)
 - [B.5.1 Children allowed in AppData elements](#)
 - [B.5.2 Attributes allowed in AppData elements](#)

- [B.6 Tag Action](#)
 - [B.6.1 Children allowed in Action elements](#)
 - [B.6.2 Attributes allowed in Action elements](#)
- [B.7 Tag CustomEditor](#)
 - [B.7.1 Children allowed in CustomEditor elements](#)
 - [B.7.2 Attributes allowed in CustomEditor elements](#)
- [B.8 Tag Integer](#)
 - [B.8.1 Children allowed in Integer elements](#)
 - [B.8.2 Attributes allowed in Integer elements](#)
- [B.9 Tag Double](#)
 - [B.9.1 Children allowed in Double elements](#)
 - [B.9.2 Attributes allowed in Double elements](#)
- [B.10 Tag Boolean](#)
 - [B.10.1 Children allowed in Boolean elements](#)
 - [B.10.2 Attributes allowed in Boolean elements](#)
- [B.11 Tag String](#)
 - [B.11.1 Children allowed in String elements](#)
 - [B.11.2 Attributes allowed in String elements](#)
- [B.12 Tag Array](#)
 - [B.12.1 Children allowed in Array elements](#)
 - [B.12.2 Attributes allowed in Array elements](#)
- [B.13 Tag Table](#)
 - [B.13.1 Children allowed in Table elements](#)
 - [B.13.2 Attributes allowed in Table elements](#)
- [B.14 Tag Reference](#)
 - [B.14.1 Children allowed in Reference elements](#)
 - [B.14.2 Attributes allowed in Reference elements](#)
- [B.15 Tag Comment](#)
 - [B.15.1 Children allowed in Comment elements](#)
 - [B.15.2 Attributes allowed in Comment elements](#)
- [B.16 Tag Menu](#)
 - [B.16.1 Children allowed in Menu elements](#)
 - [B.16.2 Attributes allowed in Menu elements](#)
- [B.17 Tag Master](#)
 - [B.17.1 Children allowed in Master elements](#)
 - [B.17.2 Attributes allowed in Master elements](#)
- [B.18 Tag Contents](#)
 - [B.18.1 Children allowed in Contents elements](#)

- [B.18.2 Attributes allowed in Contents elements](#)
 - [B.19 Tag Item](#)
 - [B.19.1 Children allowed in Item elements](#)
 - [B.19.2 Attributes allowed in Item elements](#)
 - [B.20 Tag Header](#)
 - [B.20.1 Children allowed in Header elements](#)
 - [B.20.2 Attributes allowed in Header elements](#)
 - [B.21 Tag Entries](#)
 - [B.21.1 Children allowed in Entries elements](#)
 - [B.21.2 Attributes allowed in Entries elements](#)
 - [B.22 Tag Entry](#)
 - [B.22.1 Children allowed in Entry elements](#)
 - [B.22.2 Attributes allowed in Entry elements](#)
 - [B.23 Tag Cell](#)
 - [B.23.1 Children allowed in Cell elements](#)
 - [B.23.2 Attributes allowed in Cell elements](#)
 - [B.24 Tag *type name*](#)
 - [B.24.1 Children allowed in *type name* elements](#)
 - [B.24.2 Attributes allowed in *type name* elements](#)
-

[Next](#) [Up](#) [Previous](#)**Next:** [1.1 Installation Instructions](#) **Up:** [A Maui User's Guide](#) **Previous:** [A Maui User's Guide](#)

1. Getting Started

In this chapter we will explain how to configure and build Maui, and how to run a first simple Maui GUI. The directions are for either a Unix (Linux) or Windows system.

Subsections

- [1.1 Installation Instructions](#)
 - [1.1.1 System Requirements](#)
 - [1.1.2 Installing Java](#)
 - [1.1.3 Installing Ant](#)
 - [1.1.4 Downloading MAUI](#)
 - [1.1.5 UNIX/LINUX Installation](#)
 - [1.1.5.1 Installing MAUI on UNIX/LINUX](#)
 - [1.1.5.2 Troubleshooting the installation on UNIX/LINUX](#)
 - [1.1.6 Windows Installation](#)
 - [1.1.6.1 Installing MAUI on Windows](#)
 - [1.1.6.2 Troubleshooting the installation on Windows](#)
- [1.2 Running Maui](#)

1.1 Installation Instructions

Subsections

- [1.1.1 System Requirements](#)
 - [1.1.2 Installing Java](#)
 - [1.1.3 Installing Ant](#)
 - [1.1.4 Downloading MAUI](#)
 - [1.1.5 UNIX/LINUX Installation](#)
 - [1.1.5.1 Installing MAUI on UNIX/LINUX](#)
 - [1.1.5.2 Troubleshooting the installation on UNIX/LINUX](#)
 - [1.1.6 Windows Installation](#)
 - [1.1.6.1 Installing MAUI on Windows](#)
 - [1.1.6.2 Troubleshooting the installation on Windows](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [1.1.2 Installing Java](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1 Installation Instructions](#)

1.1.1 System Requirements

MAUI will run on any java enabled platform.

[Next](#) [Up](#) [Previous](#)

Next: [1.1.2 Installing Java](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1 Installation Instructions](#)

[Next](#) [Up](#) [Previous](#)

Next: [1.1.3 Installing Ant](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.1 System Requirements](#)

1.1.2 Installing Java

To run MAUI, you will need a Java runtime environment (JRE). The JRE should be from Java2 Standard Edition (J2SE) 1.3 or higher version. You can download the latest JRE from www.java.com.

Are you planning on writing custom editors or custom event handlers?

To write custom components, you will need to download and install a Software Development Kit (SDK) or an equivalent Java development environment. We recommend the SDK from the Java2 Standard Edition version 1.3 or higher; you can get it at www.java.sun.com.

[Next](#) [Up](#) [Previous](#)

Next: [1.1.3 Installing Ant](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.1 System Requirements](#)

[Next](#) [Up](#) [Previous](#)

Next: [1.1.4 Downloading Maui](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.3 Installing Ant](#)

1.1.3 Installing Ant

Are planning to modify and/or extend the MAUI core?

To edit and re-compile the MAUI core, you will need a build tool such as make or Ant. You can download Ant at <http://ant.apache.org/>.

[Next](#) [Up](#) [Previous](#)

Next: [1.1.4 Downloading Maui](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.3 Installing Ant](#)

[Next](#) [Up](#) [Previous](#)

Next: [1.1.5 UNIX/LINUX Installation](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.3 Installing Ant](#)

1.1.4 Downloading MAUI

You can download MAUI at <http://csmr.ca.sandia.gov/projects/maui/download.php>. UNIX/LINUX users should download the file Maui.tgz. Windows users should download the file Maui.zip.

[Next](#) [Up](#) [Previous](#)

Next: [1.1.5 UNIX/LINUX Installation](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.3 Installing Ant](#)

[Next](#) [Up](#) [Previous](#)

Next: [1.1.5.1 Installing MAUI](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.4 Downloading Maui](#)

1.1.5 UNIX/LINUX Installation

Subsections

- [1.2.5.1 Installing MAUI on UNIX/LINUX](#)
 - [1.2.5.2 Troubleshooting the installation on UNIX/LINUX](#)
-

1.1.5.1 Installing Maui on UNIX/LINUX

1. Select a directory (folder) where you want to install Maui.
Download the Maui.tgz file into that directory.
cd to that directory.

2. Unpack Maui.tgz by typing either:

```
tar xzf Maui.tgz
```

```
gunzip Maui.tgz  
tar xf Maui.tar
```

```
gzip -d Maui.tgz  
tar xf Maui.tar
```

3. Determine which UNIX shell you are currently using. Type the following command:

```
echo $SHELL
```

If you see "/bin/bash" then you are using the bash shell.

If you see "/bin/tcsh" then you are using the tc shell.

If you see "/bin/csh" then you are using the c shell.

If you see "/bin/sh" then you are using the bourne shell.

4. Set the MAUI_HOME environment variable by typing any one of the following commands:

If you running csh or tcsh, use

```
setenv MAUI_HOME `pwd`/Maui
```

or

```
setenv MAUI_HOME <full path to Maui folder>
```

```
example: setenv MAUI_HOME /home/MauiUser/Maui
```

If you running bash or bourne, use

```
export MAUI_HOME=`pwd`/Maui
```

or

```
export MAUI_HOME=<full path to Maui folder>
```

```
example: export MAUI_HOME=/home/MauiUser/Maui
```

Note: These settings are not permanent. To make them permanent, add the appropriate line to

your shell's rc file (.cshrc, .tcshrc, .bashrc, .shrc, etc). These files are located in your home directory.

<p>if you are running csh then edit .cshrc. setenv MAUI_HOME <full path to Maui folder> example: setenv MAUI_HOME /home/MauiUser/Maui</p>

<p>if you are running tcsh then edit .tcshrc. setenv MAUI_HOME <full path to Maui folder> example: setenv MAUI_HOME /home/MauiUser/Maui</p>

<p>if you are running bash then edit .bashrc export MAUI_HOME =<full path to Maui folder> example: export MAUI_HOME= /home/MauiUser/Maui</p>
--

<p>if you are running bourne then edit shrc export MAUI_HOME = <full path to Maui folder> example: export MAUI_HOME= /home/MauiUser/Maui</p>
--

5. Modify your PATH variable to include the path to the Maui executable commands. Again, the best way to do this is to modify your shell's rc file. The appropriate line is

PATH=\$MAUI_HOME/Java/bin:\$PATH

6. If you modified your shell's rc file then you must either:

<p>logout and log back in</p>
<p>source your shell's rc file example: source ~/.bashrc</p>

7. Launch Maui by typing this command:

Maui.e \$MAUI_HOME/Java/etc/test.xml

8. At this point, if the MAUI_HOME and PATH variables are set correctly, only one error could occur. Maui needs to create a local directory in which to store your personal preferences and settings. This directory is named .mauiConfig. You can find the folder in your home directory. The only thing that can go wrong here is that permissions are not set to allow Maui to create this file. This is highly unusual, since you had to have had permission to write files to install Maui, but if it happens, be sure that the write permissions are set correctly and try again.

To view the write permissions, type this command:

ls -ld ~/.mauiConfig

You should see something that looks like this:

```
drwxr-xr-x 1 MauiUser MauiUser 4096 Jul 4 2003
```

The 3rd character should be a "w", not a "-".

If the 3rd character is a "-" then type this command to fix it:

chmod +x ~/.mauiConfig

9. When Maui comes up, click on the "Start New Session" button. If all goes well, you will see a greeting screen with a note to click on "Exit" to quit Maui.

[Next](#) [Up](#) [Previous](#)

Next: [1.1.5.2 Troubleshooting](#) **Up:** [1.1.5 UNIX/LINUX Install](#) **Previous:** [1.1.5 UNIX/LINUX Install](#)

[Next](#) [Up](#) [Previous](#)

Next: [1.1.6 Windows Installation](#) **Up:** [1.1.5 UJNIX/LINUX Install](#) **Previous:** [1.1.5.1 Installing MAUI](#)

1.1.5.2 Troubleshooting the installation on UNIX/LINUX

Do you have a Java Runtime Environment (JRE) installed on your system?

In most LINUX installations, the JRE is optional; i.e. if you did not select a "custom installation" then the JRE was probably not installed.

To determine if java is installed, type these commands:

```
which javac
whereis javac
locate bin/javac
```

If none of these 3 commands can find java, then java was probably not installed.

Can your PATH environment variable find the Java Runtime Environment (JRE)?

To determine if your PATH is ok, type one of these commands:

```
printenv PATH
set | grep PATH
echo $PATH
```

Search for an entry that resembles j2sdk/bin or j2sdk/jre/bin

Do you have the correct version of the Java Runtime Environment (JRE)?

It should be version 1.3 or higher.

It should NOT be version 1.4.0_01; j2sdk1.4.0_01 is full of bugs.

Most LINUX installations come with a version that is much older than 1.3.

To determine which version of java you have, type this command:

```
java -version
```

The response should resemble something similar to "/usr/local/j2sdk1.4.1_01/bin/java". The number after "j2sdk" should be 1.3 or higher. The number should NOT be 1.4.0_01.

[Next](#) [Up](#) [Previous](#)

Next: [1.1.6 Windows Installation](#) **Up:** [1.1.5 UJNIX/LINUX Install](#) **Previous:** [1.1.5.1 Installing MAUI](#)

[Next](#) [Up](#) [Previous](#)

Next: [1.1.6.1 Installing MAUI](#) **Up:** [1.1 Installation Instructions](#) **Previous:** [1.1.5.2 Troubleshooting](#)

1.1.6 Windows Installation

Subsections

- [1.1.6.1 Installing MAUI on Windows](#)
 - [1.1.6.2 Troubleshooting the installation on Windows](#)
-

1.1.6.1 Installing Maui on Windows

1. Download Maui.zip
2. Extract the files in Maui.zip to c:\Program File

You can use the unzipping tool of your choice to extract the files from Maui.zip. Use the unzipping tool to extract the files to "c:\Program Files". Some popular unzipping tools that can be downloaded from the Web include WinZip, InfoZip, and pcunzip. If you do not have an unzipping tool, then you may use Java's jar command:

EXAMPLE:

launch a DOS console

```
> copy Maui.zip c:\Program Files  
> cd c:\Program Files  
> jar xvf Maui.zip  
> del Maui.zip
```

3. Set the MAUI_HOME variable to c:\Program Files\Maui

Windows 2000:

Launch the Control Panel

Select System

Select Advanced

Select Environment Variables

Click the "New..." button in "User variables"

Type "MAUI_HOME" for the "Variable Name"

Type "c:\Program Files\Maui" for the "Variable Value"

Press the OK button

Close the Environment Variables dialog box

Windows NT:

Launch the Control Panel
Select System
Select Environment
Type "MAUI_HOME" for the "Variable"
Type "c:\Program Files\Maui" for "Value"
Click the Set button
Click the OK button
Click the OK button

Windows 98:

Start the system editor:
Click on the "Start" button in the lower left corner of your screen
Select "Run"
Type "sysedit" in the textbox
Click on the OK button
Select the "AUTOEXEC.BAT" window
Move your cursor to the bottom of the AUTOEXEC.BAT file
Type this line
set MAUI_HOME=c:\Program Files\Maui
Close the System Config Editor

4. Set the HOME variable to your profile folder

Windows 2000:

Launch the Control Panel
Select System
Select Advanced
Select Environment Variables
Click the "New..." button in "User variables"
Type "HOME" for the "Variable Name"
Type "c:\Documents and Settings\[**your login name**]" for the "Variable Value"
where [**your login name**] is the user name you used to login into the machine.
EXAMPLE: If your login name is johnDoe then would type
"c:\Documents and Settings\johnDoe"
Press the OK button
Close the Environment Variables dialog box

Windows NT:

Launch the Control Panel

Select System

Select Environment

Type "MAUI_HOME" for the "Variable"

Type "c:\Winnt\profiles\[**your login name**]" for "Value"

where [**your login name**] is the user name you used to login into the machine.

EXAMPLE: If your login name is johnDoe then would type

"c:\Winnt\profiles\johnDoe"

Click the Set button

Click the OK button

Click the OK button

Windows 98:

If you haven't already created a user profile then

Launch the Control Panel

Double-click on Users

A wizard will help you setup a name and password

Start the system editor:

Click on the "Start" button in the lower left corner of your screen

Select "Run"

Type "sysedit" in the textbox

Click on the OK button

Select the "AUTOEXEC.BAT" window

Move your cursor to the bottom of the AUTOEXEC.BAT file

Type this line

set HOME=c:\Windows\profiles\[**your login name**]

where [**your login name**] is the user name you used to login into the machine.

EXAMPLE: If your login name is johnDoe then would type

"c:\Windows\profiles\johnDoe"

Close the System Config Editor

Reboot your Windows 98 machine.

5. Launch Maui by running Maui.bat

Double click on the file named c:\Program Files\Maui\Java\bin\Maui.bat.

You may also launch Maui from a DOS console:

Launch a DOS console

```
> cd c:\Program Files\Maui\Java\bin
```

```
> Maui.bat
```

6. When Maui comes up, click on the "Start New Session" button.

[Next](#) [Up](#) [Previous](#)

Next: [1.1.6.2 Troubleshooting](#) **Up:** [1.1.6 Windows Install](#) **Previous:** [1.1.6 Windows Install](#)

[Next](#) [Up](#) [Previous](#)

Next: [1.2 Running Maui](#) **Up:** [1.1.6 Windows Install](#) **Previous:** [1.1.6.1 Installing Maui](#)

1.1.6.2 Troubleshooting the installation on Windows

Do you have a Java Runtime Environment (JRE) installed on your system?

Microsoft Windows does NOT come with a JRE.

To determine if java is installed, follow these steps:

- open the control panel

- click on "Add/Remove Programs"

- Search for "Java2 Run Time Environment" or "Java2 SDK"

Can your PATH environment variable find the Java Runtime Environment (JRE)?

To determine if your PATH is ok, follow these steps:

- launch a DOS console

- type the command

 - java -version

If your path is ok then the response should resemble something similar to

"/usr/local/j2sdk1.4.1_01/bin/java." If your path is not ok then you will see an error message on the screen (e.g. "command not found").

Do you have the correct version of the Java Runtime Environment (JRE)?

It should be version 1.3 or higher.

It should NOT be version 1.4.0_01; j2sdk1.4.0_01 is full of bugs.

To determine which version of java you have, follow these steps:

- launch a DOS console

- type the command

 - java -version

The response should resemble something similar to "/usr/local/j2sdk1.4.1_01/bin/java." The number after "j2sdk" should be 1.3 or higher. The number should NOT be 1.4.0_01.

[Next](#) [Up](#) [Previous](#)

Next: [1.2 Running Maui](#) **Up:** [1.1.6 Windows Install](#) **Previous:** [1.1.6.1 Installing Maui](#)

[Next](#) [Up](#) [Previous](#)**Next:** [2. How to Design](#) **Up:** [1 Getting Started](#) **Previous:** [1.1.6.2 Troubleshooting](#)

1.2 Running Maui

Maui is Java code, and runs under a Java virtual machine. We have provided a script that takes care of setting the appropriate arguments for the Java command line. The UNIX shell script is named Maui.e. The windows batch script is named Maui.bat. All you need do to run Maui is to execute the script with the name of an XML input file as an argument.

UNIX/LINUX example:

```
Maui.e MyGUI.xml
```

Windows example:

```
Maui.bat MyGUI.xml
```

The XML input file contains a specification of the GUI to be generated, and the bulk of this tutorial is concerned with how to write such files.

Let's get started. Figure [1.1](#) shows the XML input to display a text box in which the user can enter the value of an integer variable. We will explain what the contents mean later; for now just run Maui. You can type this XML into a file yourself using your favorite editor, or you can get a copy of the file from `$MAUI_HOME/Doc/tutorials/maui/XML/FirstExample.xml`. Once you have created or copied the file `FirstExample.xml`, run `Maui.e` from the command line:

UNIX/LINUX

```
Maui.e FirstExample.xml
```

Windows

```
Maui.bat FirstExample.xml
```

You should see a window looking like that shown in Figure [1.2](#).

The XML files for all the examples shown in this tutorial are in the directory `$MAUI_HOME/Doc/tutorial/maui/XML`, and you can copy, modify if you like, and run any of them in the same way you ran `FirstExample.xml`.

It is possible to configure Maui to run always with a particular XML input file. This is often a good way to run Maui in a production environment. For more information on configuring Maui, see Chapter [5](#).

```
<Maui RootClass="MyFirstClass">
<Class type="MyFirstClass">
  <Fields>
    <Int name="num" label="Life, the Universe, and Everything" default="42"/>
  </Fields>
</Class>
</Maui>
```

Figure 1.1: Our first Maui XML file; Just enough XML to generate our first GUI.

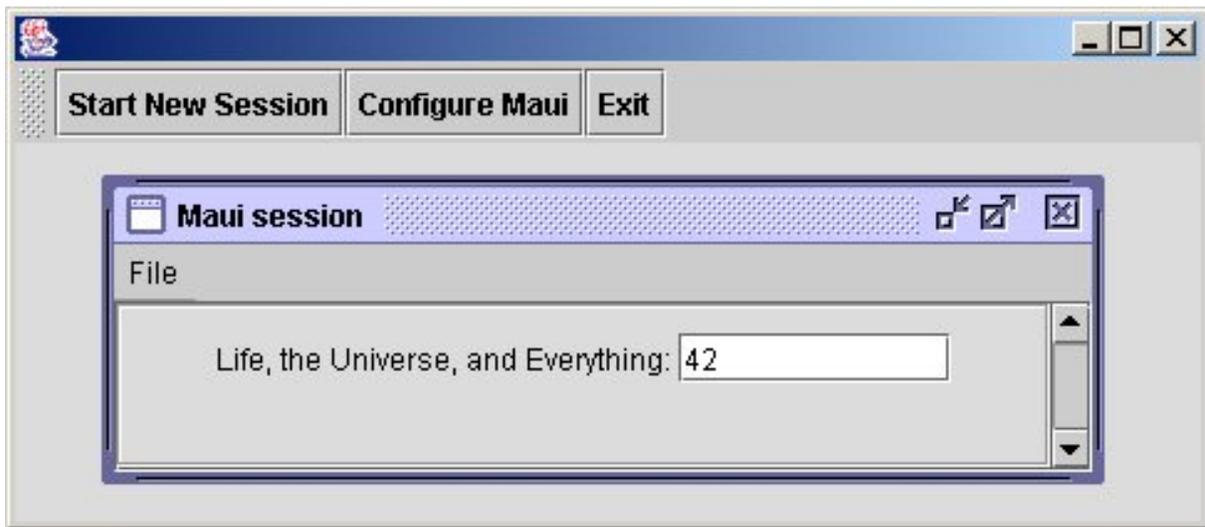


Figure 1.2:Maui GUI generated by the input shown in Figure [1.1](#)

[Next](#) [Up](#) [Previous](#)

Next: [2. How to Design](#) **Up:** [1 Getting Started](#) **Previous:** [1.1.6.2 Troubleshooting](#)

[Next](#) [Up](#) [Previous](#)

Next: [2.1 Introduction](#) **Up:** [A Maui User's Guide](#) **Previous:** [1.1.2 Running Maui](#)

2. How to Design Maui Objects

Subsections

- [2.1 Introduction](#)
 - [2.2 The Basics](#)
 - [2.3 Maui Primitives](#)
 - [2.4 Classes as Fields](#)
 - [2.5 Using Subclasses to Represent Choices](#)
 - [2.5.1 Data Representation](#)
 - [2.5.2 Appearance of Subclasses in the GUI: Tabbed Panes](#)
 - [2.5.3 Labeling in the Subclass Menu](#)
 - [2.6 Arrays](#)
 - [2.6.1 The Master Block](#)
 - [2.6.2 Other Array Options](#)
 - [2.7 Tables](#)
 - [2.8 References](#)
 - [2.9 Maui help buttons](#)
 - [2.10 Summary of Maui](#)
-

2.1 Introduction

Maui is a tool for building a graphical user interface (GUI) to an application given a high-level XML specification of the application. The XML input to Maui captures the structure of the data for the underlying application as well as the necessary parameters to specify a particular problem. Maui's output is XML that contains both the structural information as well as the user's input parameters. For this output XML to be used by the underlying application, the Maui GUI developer may be required to generate some Java code (or code in some other language) for parsing the XML and formatting the data into a style appropriate to the application.

Those targeted to use Maui are application developers who would like to spend more time developing application codes, and less time writing GUIs for those applications. The motivating idea behind Maui is that it is ultimately more efficient to build a single GUI builder than to build an endless series of single-purpose GUIs.

There are a number of advantages to using Maui to develop your GUI. Most importantly, Maui is a timesaver: we have found that a GUI for a moderately complex application can be written in a few days with a few hundred lines of Maui specification XML rather than weeks or months with thousands of lines of Java. This implies that the GUI can be easily modified and updated, making Maui an ideal tool to be used during the development stage. Furthermore, since Maui is a standard your GUI will immediately have a look-and-feel identical to other GUIs built with Maui.

Maui is written in Java, but the developer of a Maui application need not write a lot of Java code to use Maui or to read this tutorial. What the developer should know is:

- The fundamentals of XML such as elements, attributes, and content. Advanced XML features are not needed to write Maui specifications.
- Fundamental concepts of object-oriented design such as classes and inheritance. Some experience with an object-oriented programming language is useful. Java coding is necessary to write custom actions and custom editors for a specific application. Additionally, Java coding may be necessary for parsing Maui output XML for use in the specific application. Only XML knowledge is needed to generate a Maui GUI.

GUI development with Maui is most efficient when the application is understood from an object-oriented point of view. However, this does not in any way limit the use of Maui to applications written in object-oriented languages. Indeed, Maui is an ideal way to provide a more structured interface to a traditional

FORTRAN or C subroutine library.

This tutorial is intended as a step-by-step, example-centered introduction to Maui. Beyond simply presenting Maui syntax, an important goal of the tutorial is to illustrate good principles for designing Maui GUIs. As with any design problem, there are many paths and choices that can be made; most nontrivial applications could be interfaced with Maui in a number of ways. We will try to suggest guiding principles for creating effective GUIs and for efficient use of Maui.

In this tutorial, when we refer to a *user*, we mean the end user of the GUI produced by Maui. You, the reader, are assumed to be a Maui GUI *developer*, by which we mean one who is using Maui to develop a GUI, not one who is developing the core Maui code.

In this chapter, we lead you through writing and displaying your first Maui class.

[Next](#) [Up](#) [Previous](#)

Next: [2.2 The Basics](#) **Up:** [2. How to Design](#) **Previous:** [2. How to Design](#)

2.2 The Basics

Every Maui input file must contain a single XML block with tag `Maui`. The `Maui` block in Figure [1.1](#) has an attribute `RootClass="MyFirstClass"`. The value of the `RootClass` attribute gives the name of the class that will be displayed in the main panel of the GUI.

Any Maui GUI, even this simple one to edit a single integer, will be defined in terms of a class. What is a class? A class is a collection of GUI components (textboxes, buttons, checkboxes, etc.). For example, in the following figure, we have created the class, "MyFirstClass." Inside of the `<Fields>` tag, we have one integer textbox.

```
<Class type="MyFirstClass">
  <Fields>
    <Int name="num" label="Life, the Universe, and Everything" default="42"/>
  </Fields>
</Class>
```

Figure 1.1.1: Example Maui Class

A Maui class is written as a block of XML with tag `Class`. The type name of the class is given as the `type` attribute of the class; in Figure [1.1](#) the name of the class is `MyFirstClass`. A legal Maui class name must contain no whitespace or special XML characters; examples of legal Maui class names are `MyFirstClass`, `my_first_class`, and `myClass1`.

If a class has data fields, the `Class` block will have a child called `Fields`. The fields themselves will be the children of `Fields`. Each field is a single XML block whose tag is the type of that field. For example, in Figure [1.1](#) the tag of the single field is `Int`.

Some of the common attributes for fields are listed below:

- **name** is a required attribute for all fields. It is used internally by Maui as a variable name to identify the field, and in the case of primitives, it may be used as the attribute name for that variable in the output XML. The value of the `name` attribute should be a string with no spaces. Additionally, no two fields inside the same class should have the same name; this is just as two different variables within the same scope must have different names in any programming language.
- **label** is an optional attribute for all fields. It is used to specify the labeling text that will appear on the GUI component for editing the field. If a `label` attribute is not supplied, the text label defaults to the field's **name** attribute.
- **default** is an optional attribute for primitive fields. It specifies the default value of the primitive variable, which

will appear as initial data in the variable's editable GUI component (for example, a text field). If no default is supplied, the editable component will be left blank.

Other field types may have other attributes. See Appendix [B](#) for the complete description of Maui XML elements and their attributes.

When you run this example, the Maui window will appear as in Figure [1.2](#). The single field in Figure [1.2](#) is an integer, and Maui will restrict input to text that can be interpreted as an integer. While Maui will not prevent typing non-integer text such as 3.14159 or See spot run into the text field, it will detect invalid input whenever a Maui action button is pushed. See Chapter [3](#) for information on creating Maui actions.

[Next](#) [Up](#) [Previous](#)

Next: [2.3 Maui Primitives](#) **Up:** [2. How to Design](#) **Previous:** [2.1 Introduction](#)

Next Up Previous

Next: [2.4 Classes as Fields](#) Up: [2. How to Design](#) Previous: [2.2 The Basics](#)

2.3 Maui Primitives

```
<Maui RootClass="Primitives">
  <Class type="Primitives">
    <Fields>
      <Int name="num" label="Life, the Universe, and Everything" default="42"/>
      <Double name="pi" label="A round number" default="3.14159" optional="true"/>
      <Boolean name="liar" label="This statement is a lie" default="true"/>
      <String name="text" label="Famous last words" default="Hey, what's this red button do?"/>
      <String name="pick" label="Your choice of one">
        <Menu options="low price|high performance|reliability"/>
      </String>
      <String name="OS" label="choose your Operating System">
        <Menu options="Windows|Linux|MacOS|other" style="radioButton"/>
      </String>
      <String name="picks" label="Time period">
        <Menu options="7:45|8:00|8:15|8:30|8:45|9:00|9:15" style="list" listMode="single_interval"/>
      </String>
      <String name="quote" label="Quote" columnWidth="20" default="Nothing shocks me, I'm a
scientist.' &#xa; &#xa; -Harrison Ford, as Indiana Jones">
        <TextArea height="3"/>
      </String>
    </Fields>
  </Class>
</Maui>
```

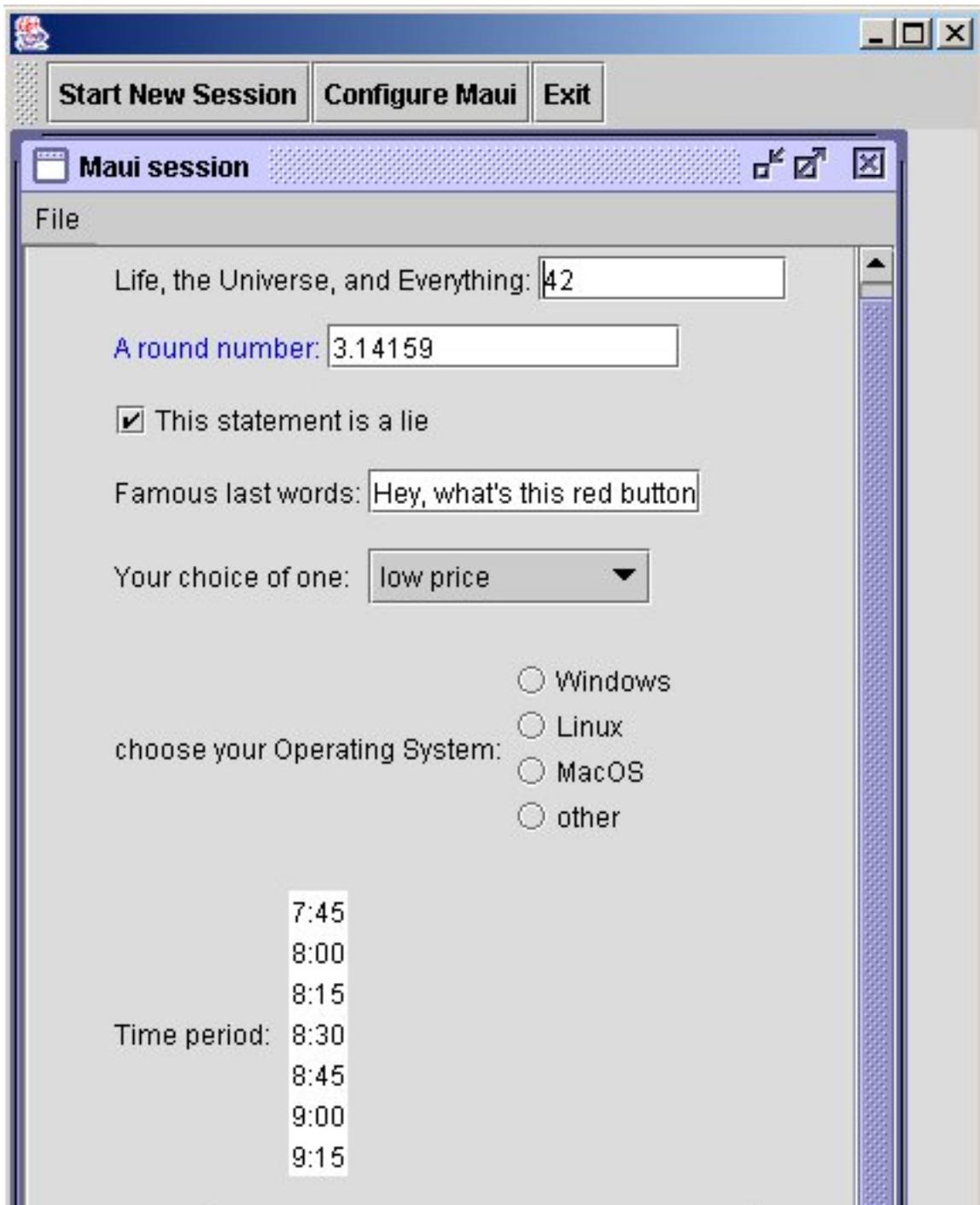
Figure 2.1: XML class that demonstrates the use of the four Maui primitive types.

In Figure [2.1](#) we show a class containing each of the four Maui primitive types: `Integer`, `Double`, `Boolean`, and `String`. We show five modes of `String` entry: the variable `text` is edited with a text box into which any text can be entered; the variable `pick` is given a list of options, and Maui produces a drop-down menu from which the user must pick one option; the variable `OS` is given a list of options from which the user must pick one from a series of radio buttons; the variable `picks` is given a list of options, and Maui produces a list from which the user can pick multiple options; the variable `quote` is given a height and Maui makes a multi-line text area in which the user can enter multiple lines of text. Note that in the XML in Figure [2.1](#) the `String` Menu choices are separated by the pipe symbol (`|`).

Not shown in this example is one other entry mode for a `String`: a text box accompanied by a file chooser button. This entry mode is specifically designed to handle string representations of file paths. This other method of representing a `String` requires the use of a `CustomEditor`; please see Chapter 4, **Custom Editors**, where we include an example of using the `String Custom Editor`.

As you might expect, the input for a `double` is restricted to be a double. There are no *a priori* restrictions on input for string variables. The GUI components for boolean and list-selectable string variables guarantee legal input for those variables.

For the complete list of attribute settings supported by each primitive, see Appendix B, **Maui XML syntax guide**.



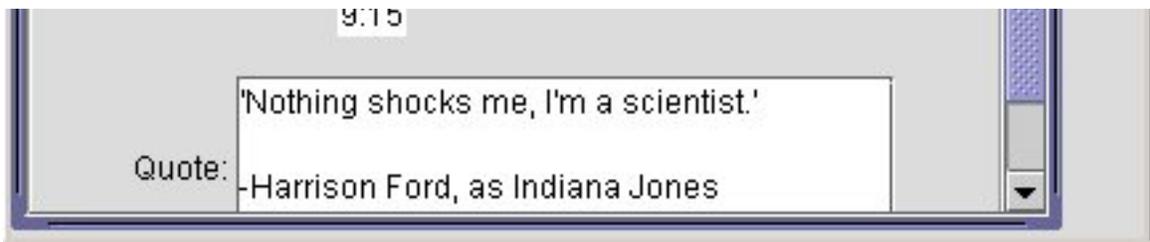


Figure 2.2:Maui GUI generated by the input shown in Figure [2.1](#).

[Next](#) [Up](#) [Previous](#)

Next: [2.4 Classes as Fields](#) **Up:** [2. How to Design](#) **Previous:** [2.2 The Basics](#)

[Next](#) [Up](#) [Previous](#)**Next:** [2.5 Using Subclasses to](#) **Up:** [2. How to Design](#) **Previous:** [2.3 Maui Primitives](#)

2.4 Classes as Fields

As in Java or C++, Maui classes can contain instances of other classes as data members. As a simple example, Figure [2.3](#) shows a `Line` class containing two `Point` classes as fields. Figure [2.4](#) shows the resulting Maui GUI. At any time, the user can determine which field to edit by selecting one of the tabs at the top of the tab pane.

By default, the class instances are displayed in the GUI as tabbed panes. The tabs appear from left to right, front to back, in the order in which consecutive class instances are defined in your XML. In Figure [2.3](#), the `Point` instance named `a` appears right before the `Point` instance named `b`, and thus the tabs appear as `point A` first, then `point B`. If the XML writer were to place a `String` instance between these two `Point` instances, as in Figure [2.5](#), then the two points would appear in separate tabbed panes, with only one tab in each pane (Figure [2.6](#)).

Maui also has the capability to display class instances as GUI components other than tabbed panes. If for a particular instance of the `Point` class you wish to have non-tabbed panes, you can use the `collapsible` attribute for the class instance. If `collapsible` is set to `false`, then a simple framed pane, containing the appropriate fields, will be used to display your class. If `collapsible` is set to `true`, you can additionally specify the `beginCollapsed` attribute to show the class pane `open` or `closed`. The XML in Figure [2.7](#) shows three `Point` objects that use the different combinations of setting the `collapsible` and `beginCollapsed` attributes. The resulting GUI representation of these classes is shown in Figure [2.8](#). Clicking on the `+/-` signs for the collapsible panes shows how the panes can be expanded or hidden as desired.

Descriptions of all the attributes for a class instance are contained in Appendix [B](#), **Maui XML syntax guide**, Section [B.24](#).

```

<Maui RootClass="Lines">

  <Class type="Lines">
    <Fields>
      <Point name="a" label="point A"/>
      <Point name="b" label="point B"/>
    </Fields>
  </Class>

  <Class type="Point">
    <Fields>
      <Double name="x" label="x coordinate"/>
      <Double name="y" label="y coordinate"/>
    </Fields>
  </Class>

</Maui>

```

Figure 2.3:Maui XML that demonstrates the use of class instance containment. Two instances of the class Point are contained in class Line.

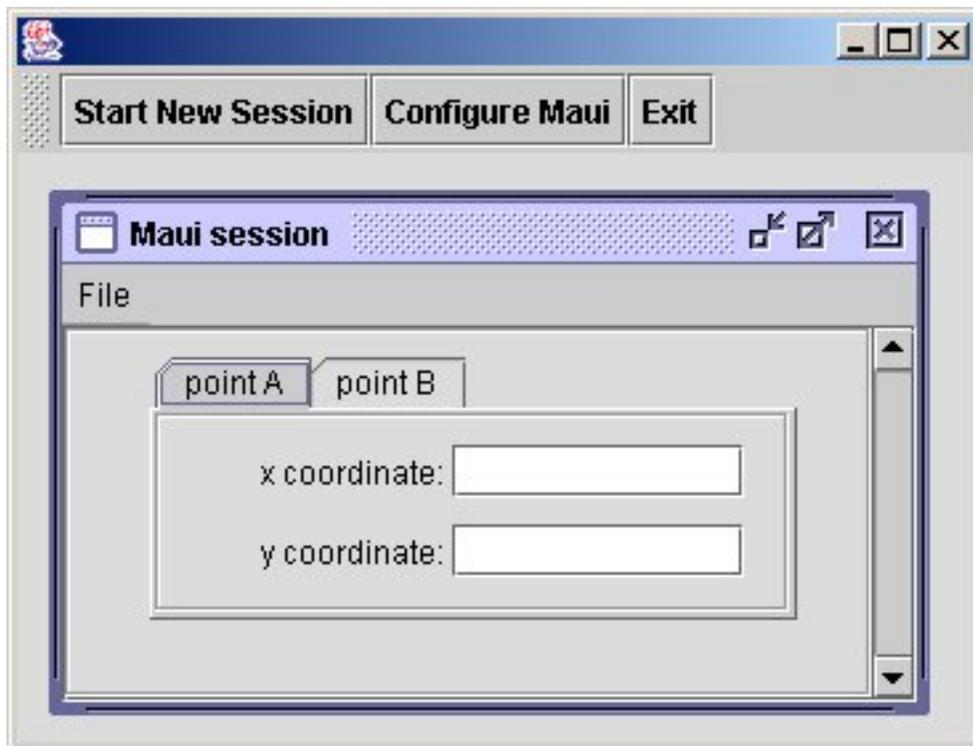


Figure 2.4:Maui GUI generated by the input shown in Figure 2.3, with field point A chosen from the tabbed panes.

```
<Maui RootClass="Lines">

  <Class type="Lines">
    <Fields>
      <Point name="a" label="point A"/>
      <String name="fred" label="Enter a string"/>
      <Point name="b" label="point B"/>
    </Fields>
  </Class>

  <Class type="Point">
    <Fields>
      <Double name="x" label="x coordinate"/>
      <Double name="y" label="y coordinate"/>
    </Fields>
  </Class>

</Maui>
```

Figure 2.5:Maui XML that demonstrates the use of class instance containment. Two instances of the class `Point` are separated by the `String` labeled `Enter a string`.

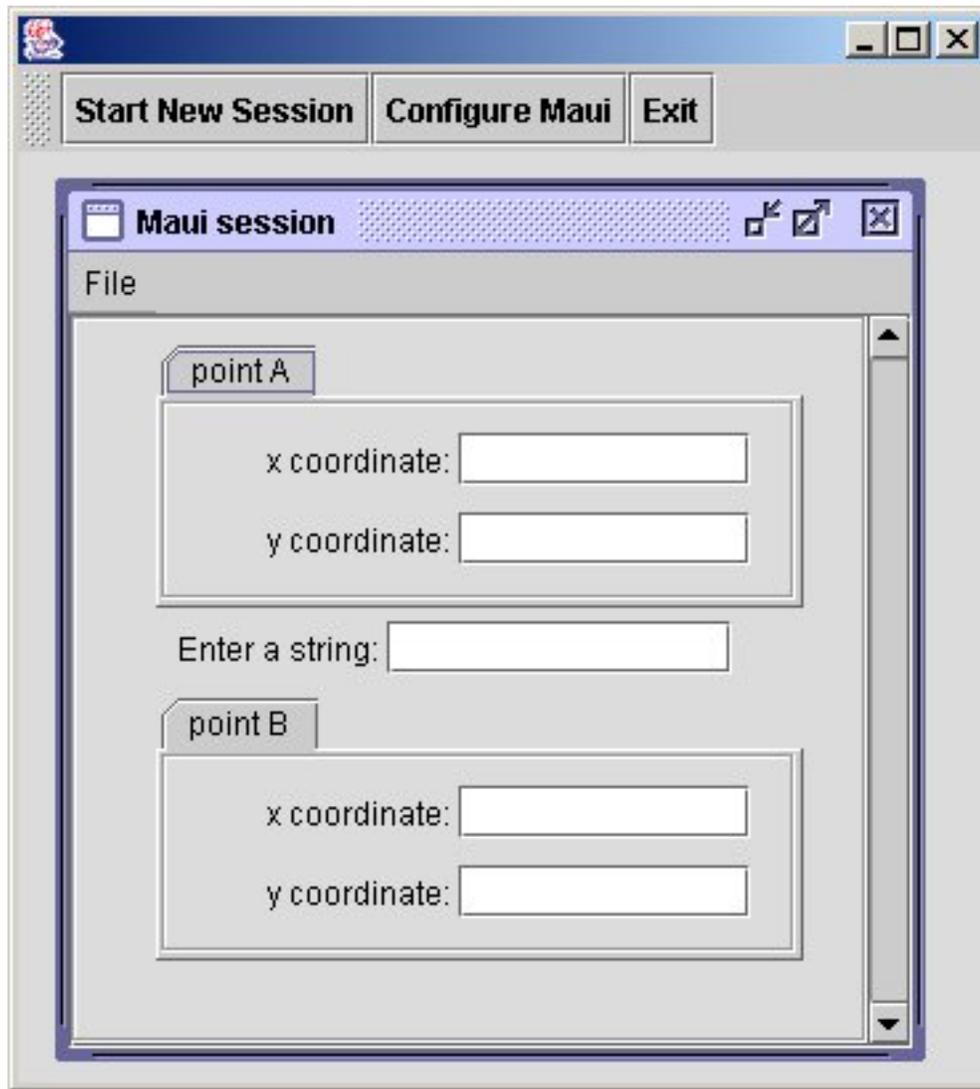


Figure 2.6:Maui GUI generated by the input shown in Figure [2.5](#).

```
<Maui RootClass="Lines">

  <Class type="Lines">
    <Fields>
      <Point name="a" label="point A" collapsible="false"/>
      <Point name="b" label="point B" collapsible="true"
        beginCollapsed="false"/>
      <Point name="c" label="point C" collapsible="true"
        beginCollapsed="true"/>
    </Fields>
  </Class>

  <Class type="Point">
    <Fields>
      <Double name="x" label="x coordinate"/>
    </Fields>
  </Class>
</Maui>
```

```
<Double name="y" label="y coordinate"/>
</Fields>
</Class>

</Maui>
```

Figure 2.7: Maui XML that demonstrates the use of `collapsible` and `beginCollapsed` attributes of a class instance.

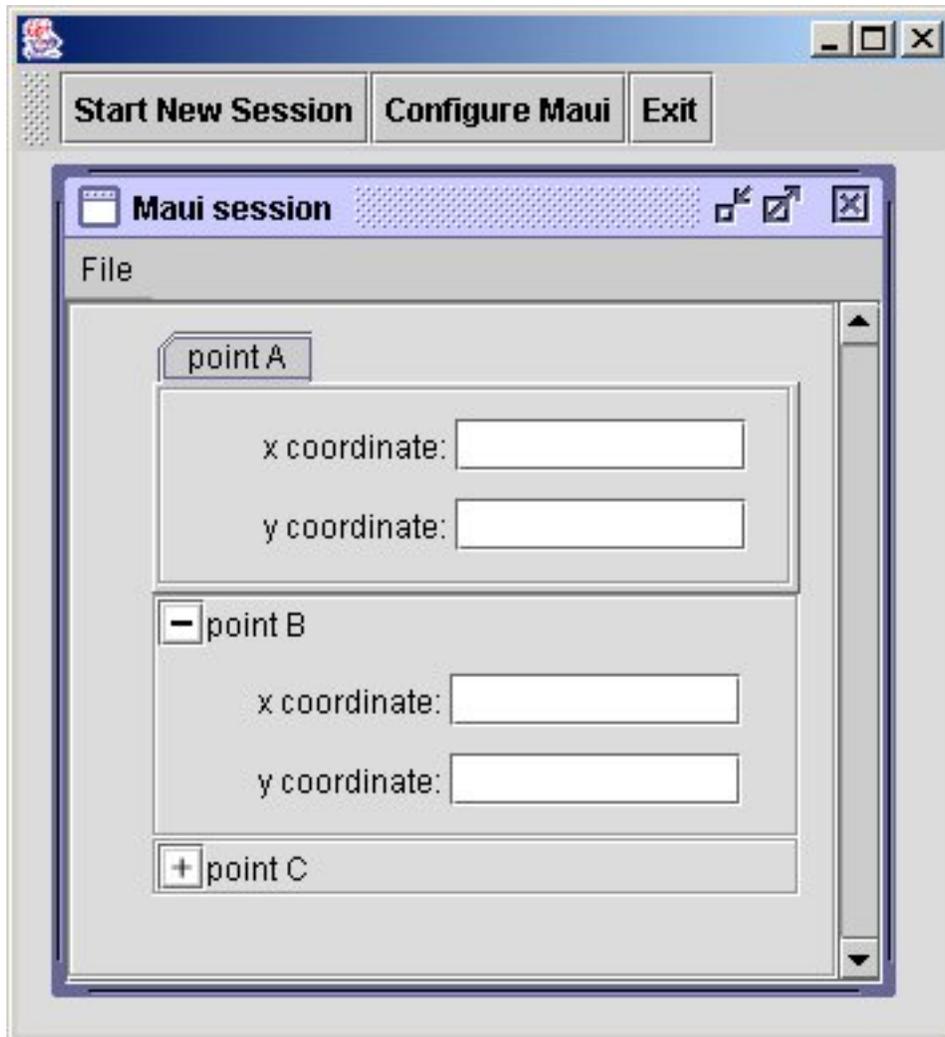


Figure 2.8: Maui GUI generated by the input shown in Figure [2.7](#).

[Next](#) [Up](#) [Previous](#)

Next: [2.5 Using Subclasses to Up](#) **Up:** [2. How to Design](#) **Previous:** [2.3 Maui Primitives](#)

[Next](#) [Up](#) [Previous](#)**Next:** [2.5.1 Data Representation](#) **Up:** [2. How to Design](#) **Previous:** [2.4 Classes as Fields](#)

2.5 Using Subclasses to Represent Choices

Users of applications codes are usually presented with a large number of choices, perhaps between different algorithms, different material models, or different output formats. Many choices will have parameters in common, but in general, parameters can be specific to a particular choice. This makes hand-editing input files difficult and error-prone. Maui's system for representing choices is intended to guarantee that the user will be presented with GUI components for only those parameters appropriate to the choice currently under consideration.

Maui represents choices between related alternatives, such as a choice between algorithms or a choice between material models, as a choice between subclasses derived from a base class. For example, two constitutive models such as linear elastic and elastic-plastic may derive from a common `ConstitutiveModel` base class.

Subsections

- [2.5.1 Data Representation](#)
 - [2.5.2 Appearance of Subclasses in the GUI: Tabbed Panes](#)
 - [2.5.3 Labeling in the Subclass Menu](#)
-

Next: [2.5.2 Appearance of Subclasses](#) **Up:** [2.5 Using Subclasses to](#) **Previous:** [2.5 Using Subclasses to](#)

2.5.1 Data Representation

When interpreting a class inheritance hierarchy, Maui will build a menu GUI component from which a particular subclass can be selected. At any time, only the fields that pertain to the selected subclass are available for editing.

Figure 2.9 shows the Maui input for a simple constitutive model class hierarchy. The classes `LinearElastic` and `ElasticPlastic` both derive from the base class of type `ConstitutiveModel`. The syntax that specifies derivation is simple: in the base class, no modification is necessary, and in the subclasses, one just adds the base attribute with the name of the base class as the attribute value for the `Class` XML block.

Notice that the Young's modulus field, `E`, and the Poisson's ratio, `nu`, are defined in the base class and will be common to all subclasses. Subclass `LinearElastic` adds no new fields, whereas subclass `ElasticPlastic` adds new fields for the initial yield stress and the hardening modulus.

```
<Maui RootClass="ConstitutiveModel">
<Class type="ConstitutiveModel"
  label="Constitutive Model (default: Linear Elastic)">
  <Fields>
    <String name="description" default="Structural steel"/>
    <Double name="E" label="Young's modulus" default="2.0e11"/>
    <Double name="nu" label="Poisson's ratio" default="0.3"/>
  </Fields>
</Class>

<Class type="LinearElastic" label="Linear elastic" base="ConstitutiveModel"/>

<Class type="ElasticPlastic" label="Elastic-plastic" base="ConstitutiveModel">
  <Fields>
    <Double name="yieldStress0" label="Initial yield stress" default="4.0e8"/>
    <Double name="hardeningModulus" label="Hardening modulus" default="2.0e8"/>
  </Fields>
</Class>

</Maui>
```

Figure 2.9: Maui input for a constitutive model base class and two subclasses representing specific constitutive models.

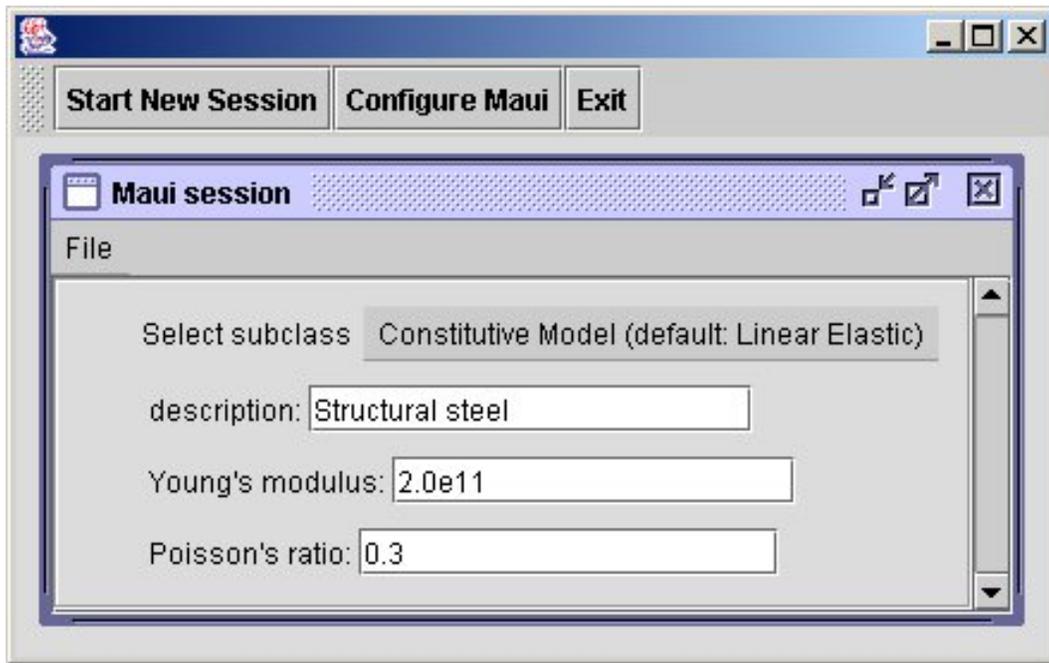


Figure 2.10:Maui GUI generated by the input shown in Figure 2.9, with subclass Linear Elastic chosen from the drop-down subclass selection menu.

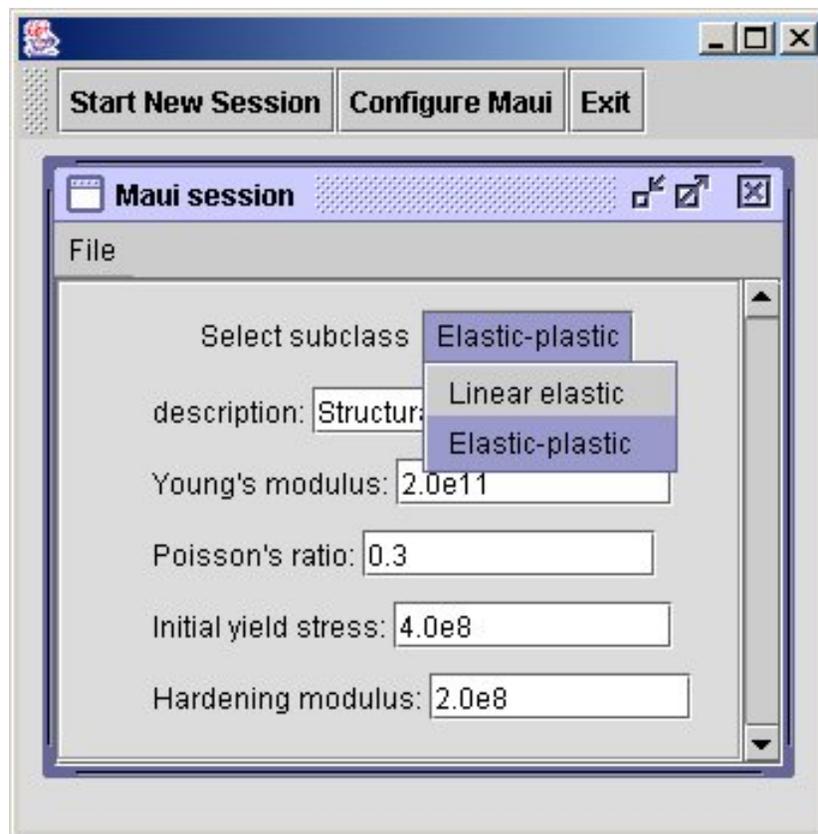


Figure 2.11:Maui GUI generated by the input shown in Figure 2.9, with subclass Elastic-Plastic chosen from the drop-down subclass selection menu.

Figures 2.10 and 2.11 show two views of the GUI generated by Maui from the input in Figure 2.9. The subclass is chosen by means of a dropdown menu presenting the subclass labels. In Figure 2.10, the user has selected the

subclass `LinearElastic` from the menu, and only the options appropriate to `LinearElastic` appear. In Figure [2.11](#), the user has selected the subclass `ElasticPlastic` and additional plasticity parameters are now editable.

[Next](#) [Up](#) [Previous](#)

Next: [2.5.2 Appearance of Subclasses](#) **Up:** [2.5 Using Subclasses to](#) **Previous:** [2.5 Using Subclasses to](#)

2.5.2 Appearance of Subclasses in the GUI: Tabbed Panes

Recall from Section [2.4](#) that one has the option to display class instances in collapsible frames, non-collapsible frames, or as tabs in a tabbed pane. One can merge, into one tabbed pane, one or more tabs in the base class with one or more tabs in a subclass; to do this, the Maui XML writer has to pay attention to where the tabs' class instances are placed in the XML file.

To locate base class and subclass class instance fields in the same tabbed pane:

1. The `useTab` attribute of the class instance should be set to `true`. This is the default if the `collapsible` attribute is not used.
2. Determine which class instances in the base class you want to represent as tabs in the same tabbed pane as class instances in a derived class. In the XML file, place those class instances as the *last* items inside the base class. These class instances must be contiguous; that is, fields of other types (`String`, `Int`, `Double`, etc.) cannot appear between these class instances.
3. Determine which class instances in the subclass you want to represent as tabs in the same tabbed pane as class instances in a base class. In the XML file, place those class instances as the *first* items inside the subclass. These class instances must be contiguous; that is, fields of other types (`String`, `Int`, `Double`, etc.) cannot appear between these class instances.

The example of how class instances displayed as tabbed panes can appear in both the XML and GUI are shown in Figures [2.12](#) and [2.13](#), respectively. Though the ``data" in the classes do not give a good example of what should go in a subclass versus a base class, the examples show the tabbed pane rendering issues in Maui. Note that the XML in Figure [2.12](#) is abridged. If you would like to run this example yourself, see the file `$MAUI_HOME/Doc/tutorials/maui/XML/SubclassExample2.xml`.

```
<Maui RootClass="MyBase">
<Class type="MyBase" label="The Base Class">
  <Fields>
    <A1 name="a1" label="Class Instance A1"/>
    <String name="myString" label="Enter a String"/>
    <!-- Because the following two class instances are adjacent,
         they will appear in the same tabbed pane -->
    <A2 name="a2" label="Class Instance A2"/>
    <A3 name="a3" label="Class Instance A3"/>
    <Int name="myInteger" label="Enter an Integer"/>
    <!-- Because the following two class instances are adjacent and located as the
         last two fields in this class, they will appear in the same tabbed pane.
         Further, they will appear in the same tabbed pane as any class instances
         that appear at the top of a subclass derived from the MyBase class. -->
    <A4 name="a4" label="Class Instance A4"/>
    <A5 name="a5" label="Class Instance A5"/>
  </Fields>
</Class>
</Maui>
```

```

</Fields>
</Class>

<Class type="MySubclass" label="Subclass Example" base="MyBase">
  <Fields>
    <!-- The following two class instances will appear in the same
         tabbed pane as the A4 and A5 instances in the base class. -->
    <B1 name="b1" label="Class Instance B1"/>
    <B2 name="b2" label="Class Instance B2"/>
    <Double name="myDouble" label="Enter a Double"/>
    <B3 name="b3" label="Class Instance B3"/>
    <B4 name="b4" label="Class Instance B4"/>
  </Fields>
</Class>

<Class type="A1" >
  <Fields>
    <String name="aName" label="Enter a name"/>
  </Fields>
</Class>

<Class type="A2" >
  <Fields>
    <String name="aPlace" label="Enter a place"/>
  </Fields>
</Class>

<Class type="A3" >
  <Fields>
    <Integer name="aNumber" label="Enter your favorite number"/>
  </Fields>
</Class>

<Class type="A4" >
  <Fields>
    <Double name="pi" label="The value of pi" default="3.14159" editable="false"/>
  </Fields>
</Class>

<Class type="A5" >
  <Fields>
    <Boolean name="iceCream" label="I like ice cream" default="true"/>
  </Fields>
</Class>

<Class type="B1" >
  <Fields>
    <String name="transport" label="Favorite mode of transportation">
      <Menu options="car|bus|train|airplane|boat" style="radioButton"/>
    </String>
  </Fields>
</Class>

```

```
</Fields>
</Class>

<Class type="B2" >
  <Fields>
    <String name="aName" label="Enter a name" />
  </Fields>
</Class>

<Class type="B3" >
  <Fields>
    <String name="aName" label="Enter a name" />
  </Fields>
</Class>

<Class type="B4" >
  <Fields>
    <String name="aName" label="Enter a name" />
  </Fields>
</Class>

</Maui>
```

Figure 2.12: Maui input showing how tabbed class instances in a base class and subclass can be rendered in one tabbed pane.

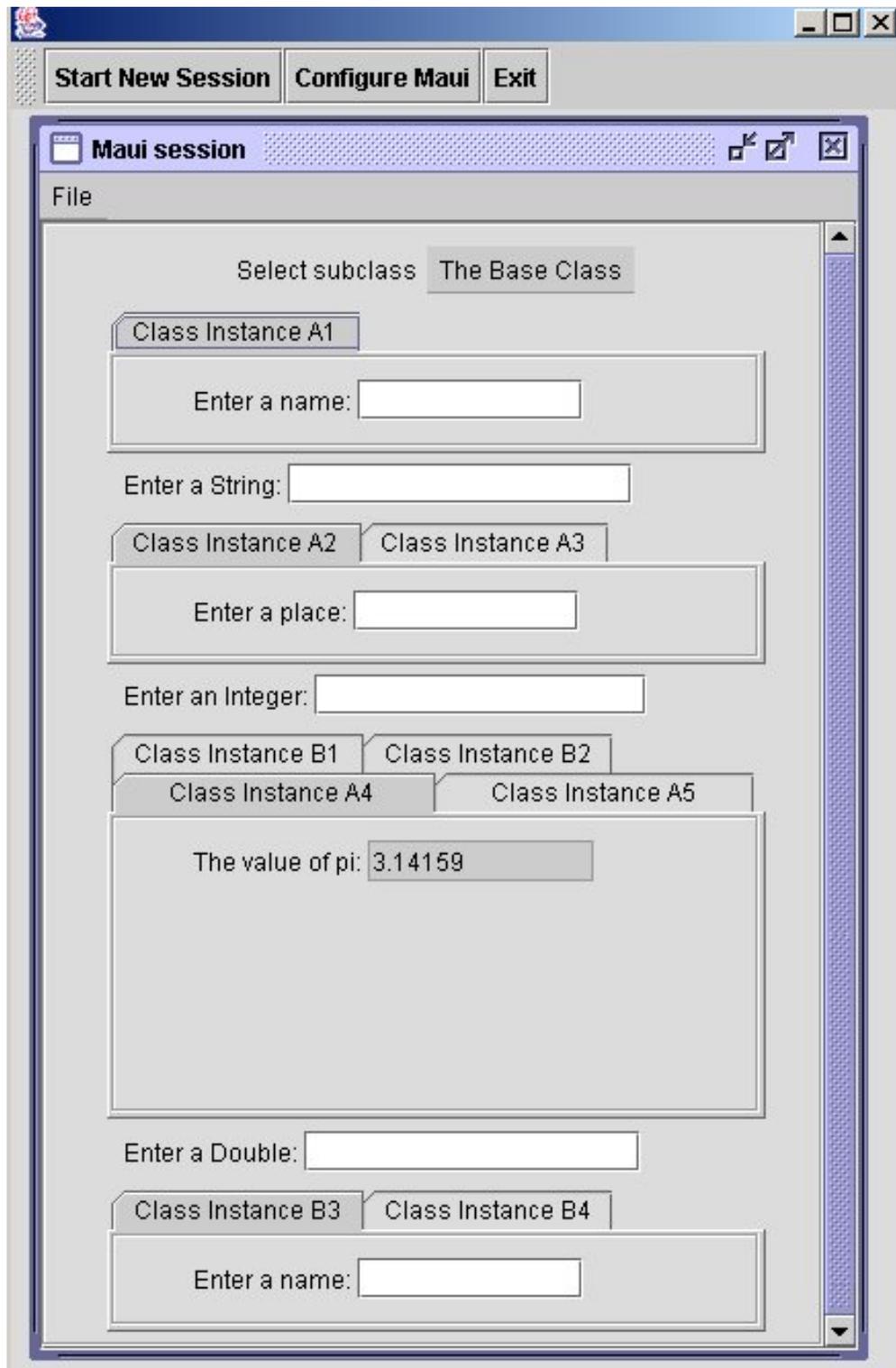


Figure 2.13:Maui GUI generated by the input shown in Figure 2.12. Note how instances of A4 and A5 from the base class, and instances of B1 and B2 from the derived class appear in the same tabbed pane.

Next Up Previous

Next: [2.5.3 Labeling in the Up: 2.5 Using Subclasses to Previous: 2.5.1 Data Representation](#)

[Next](#) [Up](#) [Previous](#)

Next: [2.6 Arrays](#) **Up:** [2.5 Using Subclasses to](#) **Previous:** [2.5.2 Appearance of Subclasses](#)

2.5.3 Labeling in the Subclass Menu

As mentioned earlier, when you set up a subclassing heirarchy, Maui will build a menu component from which a particular subclass can be selected. The labels for each of the subclasses will appear in the menu. Before a user selects on the subclass menu for the first time, the label of the base class will appear on the menu button. However, the fields associated with the first subclass in the menu will already be displayed. It is good practice to give your base class a label that notes that the first subclass is displayed by default. In Figure [2.9](#) notice that the base class, `ConstitutiveModel`, has `label="ConstitutiveModel (default: Linear Elastic)"`. This sort of labeling acknowledges that the fields of the first derived class, in terms of where the derived class was defined in the order of your XML file, are displayed even before the user has a chance to select a subclass.

Note also that by default the label that appears to the left of the subclass selection menu displays `Select subclass`. As `Subclass` is not the best description for any specific subclass, you may override the selection label with something more meaningful. For Figure [2.10](#), a more meaningful label might be `Select A Model`. To override the selection label use the `selectionLabel` attribute for an instance of a base class, or a definition of a base class:

```
<Class type="ConstitutiveModel"
  label="Constitutive Model (default: Linear Elastic)"
  selectionLabel="Select A Model">
  .
  .
  .
</Class>
```

[Next](#) [Up](#) [Previous](#)

Next: [2.6 Arrays](#) **Up:** [2.5 Using Subclasses to](#) **Previous:** [2.5.2 Appearance of Subclasses](#)

Next: [2.6.1 The Master Block](#) **Up:** [2. How to Design](#) **Previous:** [2.5.3 Labeling in the](#)

2.6 Arrays

Often we will need to work with a unknown number of class objects. For example, a finite-element code might have an arbitrary number of subregions, or a sequential approximation strategy might have an arbitrary number of approximate models. Maui represents arbitrarily-sized collections of objects as Arrays.

Figure [2.14](#) contains the XML for a simple example of an array. Figure [2.15](#) shows the GUI representation of the Array. The name of the array is `Parts` and it consists of a set of objects of class `Part`. To add an item to the array, you can click on the "Add" button, which creates a new array element and places that array element at the bottom of the list. When the editor (a dialog box for editing the array element) comes up, you can enter the values for that element. An existing element can be displayed by double clicking on the name of that element in the list. Single clicking on an element "selects" that element. The "Remove" button removes the selected item from the list. The "Insert above selection" button creates a new array element; the new element is placed in the list above the selected item.

```
<Maui RootClass="ArrayExample">
  <Class type="ArrayExample">
    <Fields>
      <Array name="Parts">
        <Master label="$partName ($materialName $A)">
          <Part name="part"/>
        </Master>
      </Array>
    </Fields>
  </Class>

  <Class type="Part">
    <Fields>
      <String name="partName" label="name of part">
        <Menu options="nut|bolt|nail|washer"/>
      </String>
      <String name="materialName" label="name of material"/>
      <Int name="Z" label="atomic number"/>
    </Fields>
  </Class>
</Maui>
```

```
<Int name="A" label="atomic weight" />
</Fields>
</Class>
</Maui>
```

Figure 2.14:Maui input XML example of creating an Array.

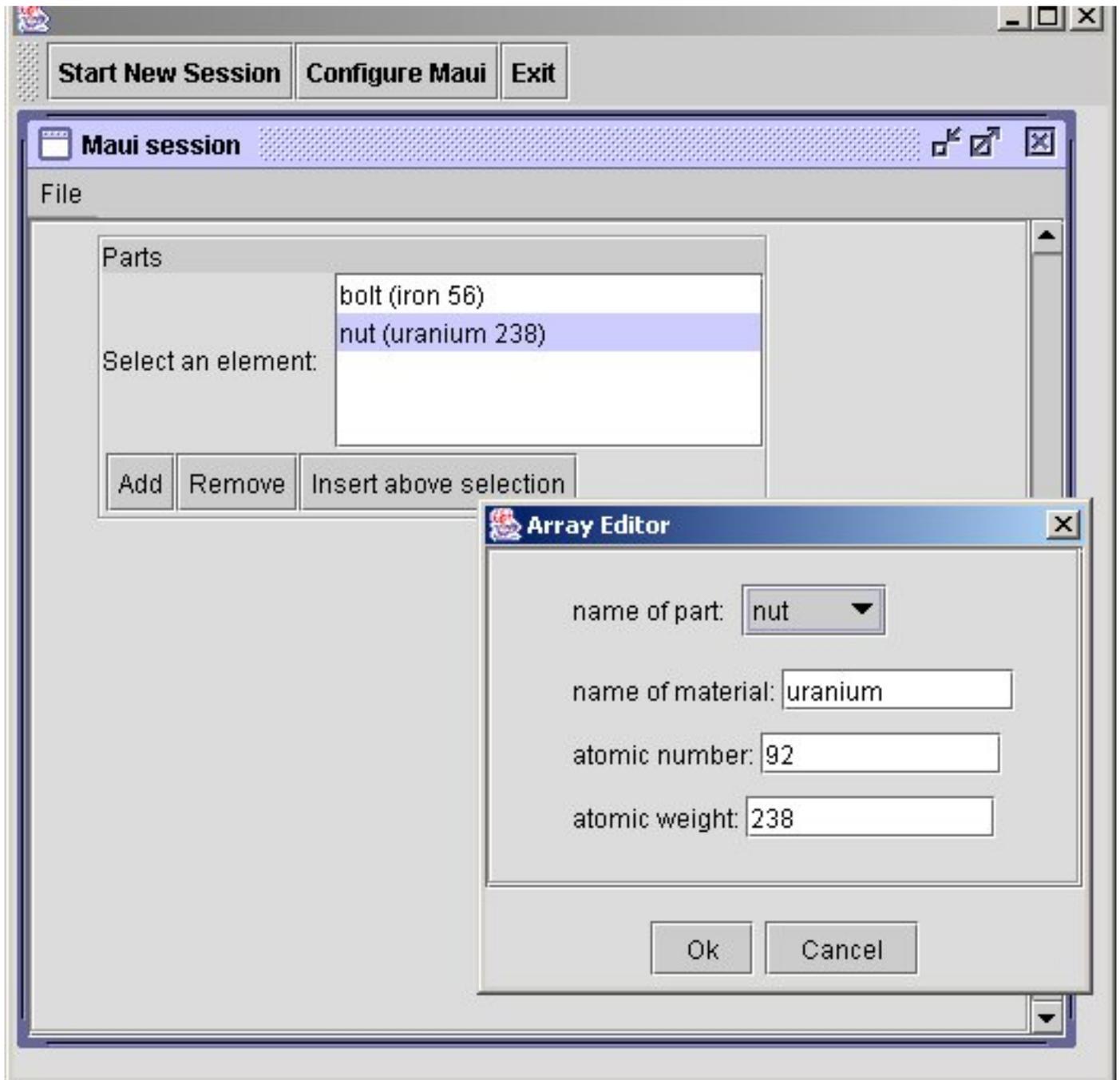


Figure 2.15:Parts Array Example.

Subsections

- [2.6.1 The Master Block](#)
 - [2.6.2 Other Array Options](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [2.6.1 The Master Block](#) **Up:** [2. How to Design](#) **Previous:** [2.5.3 Labeling in the](#)

[Next](#) [Up](#) [Previous](#)**Next:** [2.6.2 Other Array Options](#) **Up:** [2.6 Arrays](#) **Previous:** [2.6 Arrays](#)

2.6.1 The Master Block

Notice in Figure [2.14](#) that the `Array` object contains a `Master` block. It is here, inside the `Master` block, where we need to tell Maui about the type of the elements that can appear in this array. The `Master` element must be an instance of a `Class`, which can have arbitrary fields. Since we have a single `Master` element, all elements in the array must be of the type of this class. If you want to have an array with heterogeneous elements, you need to derive these types from a base class and use the base class as the `Master`.

Also notice that the label of an `Array` element, appearing in the list of array elements (see Figure [2.15](#)), depends on the value of a field in the `Master` element. The `label` attribute of a `Master` element contains a template for the label, with the variable prefixed with a dollar sign (`$`). In the example in Figure [2.14](#), the data entered by the user in the `partName`, `materialName`, and `A` fields for the class `Part` will be used to generate the label. The data entered in these fields, combined with any clear text, will be combined to make the label for a particular `Array` element.

When a user enters data into a field that is used to generate a label for an array, the label in the `Array` list is left blank until the user clicks "OK" in the array editor window.

[Next](#) [Up](#) [Previous](#)**Next:** [2.6.2 Other Array Options](#) **Up:** [2.6 Arrays](#) **Previous:** [2.6 Arrays](#)

[Next](#) [Up](#) [Previous](#)**Next:** [2.7 Tables](#) **Up:** [2.6 Arrays](#) **Previous:** [2.6.1 The Master Block](#)

2.6.2 Other Array Options

Just as with `Class` elements, the GUI display style of `Array` elements can be controlled. In Section [2.4](#), **Classes as fields**, you learned how to make a `Class` editor display in collapsible mode using the `collapsible` attribute, and to set its initial collapsed state with the `beginCollapsed` attribute. One can use the `collapsible` and `beginCollapsed` attributes with `Arrays` as well. However, `Arrays` are never represented as tabbed panes.

`Arrays` also share the `selectionLabel` attribute with `Class` elements. In Section [2.5](#), **Using subclasses to represent choices**, you learned that you could replace the default `Select` subclass label for the subclass selection GUI menu with a more personalized label. You can also use the `selectionLabel` attribute to change the default `Array` element menu label from `Select an element:` to something meaningful to the specific `Array`.

For more information on these attributes, see Sections [2.4](#) and [2.5](#), as well as Appendix [B](#), **Maui XML syntax guide**, Section [B.12](#).

[Next](#) [Up](#) [Previous](#)**Next:** [2.7 Tables](#) **Up:** [2.6 Arrays](#) **Previous:** [2.6.1 The Master Block](#)

2.7 Tables

An Array is one way to generate a list of elements in Maui, but it is not the only way. The other data type that can be used for this is a Table. In Maui, the main difference between a Table and an Array is the data types allowed in the elements. Array elements must contain only an instance of a Class object. Table elements on the other hand may only contain Strings, Integers, Doubles, Booleans and References. Despite this limitation of Tables, there are several advantages to using Tables instead of Arrays. First, in order to view or edit an Array GUI element, a new dialog box must be opened up for that element. Alternatively, in a Table all of the data is displayed in the main GUI, making it easier to view and edit data. Second, in the XML for an Array you have to write a separate class to define the Master element of the Array. In a Table there is no need for creating a class to contain the Table elements, since all of the XML describing that Table is contained within the Table Header and Entries. This makes the XML for a Table easier to write, read and maintain. Even though Arrays are more flexible, if you do not need some of the more complex data types such as Arrays, Tables or Class instances embedded in your Array elements, it may be worthwhile to use a Table instead. Two examples of Tables are displayed in Figure [2.17](#) with the corresponding XML given in Figure [2.16](#). These Tables describe an inventory list and a purchase order for items in that list.

```
<Maui RootClass="TableExample">
  <Class type="TableExample" label="Example Of A Table">
    <Fields>
      <!-- Table describing the available products -->
      <Table name="inventory" label="Available Products">
        <Header name="product" label="$productName">
          <String name="productName" label="Product"/>
          <Double name="price" label="Price in $"/>
        </Header>
        <Entries>
          <Entry>
            <Cell field="productName" value="Towel"/>
            <Cell field="price" value="9.99"/>
          </Entry>
          <Entry>
            <Cell field="productName" value="Soap"/>
            <Cell field="price" value="3.99"/>
          </Entry>
          <Entry>

```

```

        <Cell field="productName" value="Shampoo"/>
        <Cell field="price" value="5.99"/>
    </Entry>
</Entries>
</Table>
<!-- Table describing a purchase Order for the above Products -->
<Table name="order" label="Purchase Order" maxEntries="3">
    <Header name="purchase" sizing=" | 30 |">
        <Reference name="referenceToProduct" label="Product"
            path="../inventory"/>
        <Boolean name="buyThis" label="Buy?" default="false"/>
        <Integer name="quantity" label="Quantity" default="1"/>
    </Header>
    <Entries>
        <Entry>
            <Cell field="referenceToProduct" value="3"/>
        </Entry>
        <Entry>
            <Cell field="referenceToProduct" value="2"/>
            <Cell field="buyThis" value="true"/>
            <Cell field="quantity" value="3"/>
        </Entry>
    </Entries>
</Table>
</Fields>
</Class>
</Maui>

```

Figure 2.16:XML to produce the tables shown in [Figure 2.17](#).

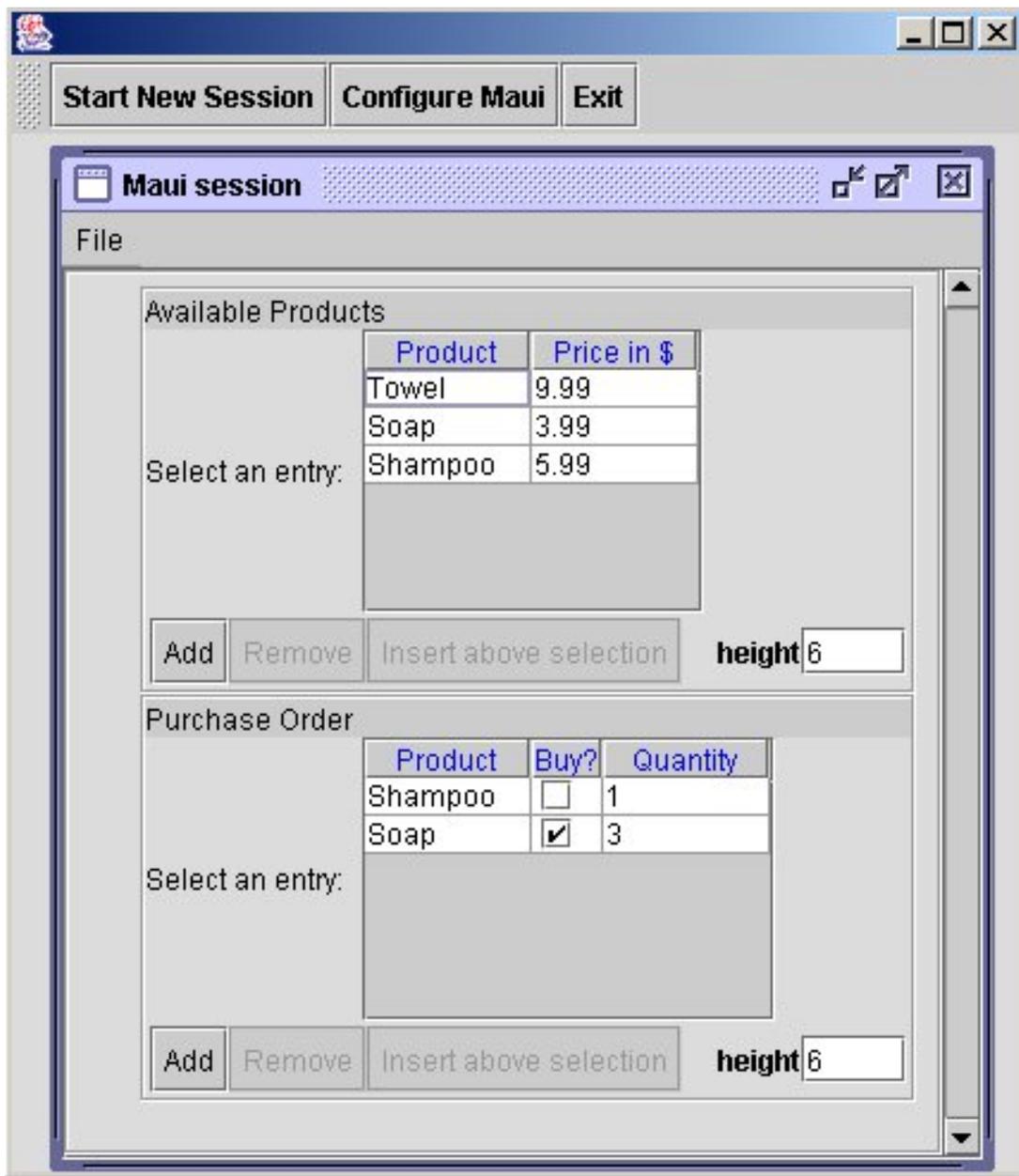


Figure 2.17:The Maui GUI generated by the XML shown in Figure 2.16, displaying products and a purchase order in two tables.

The first Table (named `inventory`) contains a list of products and the associated price of each product. This Table has two columns defined by the two fields inside the Header block of the XML. The first column requires Strings giving the product name and the second column requires Doubles giving the price of the product. Each of the fields inside the Header also has two attributes, `name` and `label`. The attribute `name` is the variable name that will be given to any new entries (columns) that are added to the Table, and `label` is the label that will be displayed for the entry column. The `label` used in the Header for the `inventory` Table is a special computed label which derives its value from the data in the Entries. The way to use data from a text field entry as part of a label is to include a `$` followed by the name of the field that should be used for the label. Note that this is the same process as for creating computed labels for an Array element. For example, in the `inventory` XML, `label="$productName"`. This means that the computed label of the first entry will be the contents of the field `productName`, i.e. `Towel`. To display an actual dollar sign in the Header label, `\$` must be entered. To display a single backslash, `\\` must

be used. So, if in the Header we have attribute setting `label="$greeting \ $greeting \\$greeting \\\$greeting"` and the entry has a field `greeting` which is set to `hi`, then the label used for the Entry will be `hi $greeting \hi \ $greeting`. While the computed labels are not seen in the `inventory` Table itself, they can be seen in the `referenceToProduct` field of the `order` Table.

In this example, the `inventory` Table starts out with three items that have already been entered: a towel, soap, and shampoo. These entries are each given by an `Entry` tag within the `Entries` block. To set the initial values in a Table entry, a `Cell` element is added inside an `Entry` element. The `field` attribute in the `Cell` gives the name of the field to set, and the `value` attribute holds the initial value of that field. So, the first `Entry` element says that the Table will start out with an element whose `productName` is `Towel` with a `price` of `9.99`. Note that you need not initialize your Table with any `Entries`; to start with a blank Table, just omit the entire `Entries` XML block.

The `order` Table shows some of the more complex features of Tables. The first noticeable difference in the XML from the `inventory` Table is the presence of a `maxEntries` attribute in the Table element. `maxEntries` limits the number of row entries that are allowed in the Table. In this case, there can be no more than three entries (rows) in the Table. Two related attributes are `minEntries`, which requires the user to have at least a certain number of entries, and `fixedNumberOfEntries` which means that there will be exactly the number of entries specified. Any or all of these three attributes can be used together provided that `maxEntries` is greater than or equal to `fixedNumberOfEntries`, which in turn is greater than or equal to `minEntries`.

Another feature in the `order` XML is the `sizing` attribute in the Header element. By default, Maui will set the width of the columns automatically, but sometimes this default size is not what you want. For example, in the `order` Table, the column `buyThis` would have been just as wide as the other two columns if Maui had used the default size, making the column much too wide for just a simple check box. To make the `buyThis` field narrower, the `sizing` attribute was used. The `sizing` attribute is a list of integers separated by pipes (`|`). These integers give the width of each column, corresponding to the order of the column variables entered in the Header. If the space for a column width is blank or negative, Maui will use a default size for that column. In this example, the `sizing` attribute is equal to `" | 30 | "`. Here, the first and third spaces are blank while the second space contains 30. This means that Maui will use the default column width for the first and third columns, but the `buyThis` column will have a size of `" 30 "`, just right to display the check box. Note that the width, in Java, corresponds to ```columns"` in the `JTextField` component. There is no good suggestion of how to determine how many ```columns"` will be wide enough for your selected display. Unfortunately, trial and error seems the best way to determine the widths of the columns for each particular problem.

One more item that appears in the `order` XML is the `reference`. For details on creating references, see Section [2.8](#). In this example, the `reference` in the purchase order is a selection reference to one of the items available in the `inventory` list. To use the selection reference in the GUI representation of the `order` Table, double click on a cell in the `Product` column of the purchase order and a list of the products will pop up so that the user can choose which product to buy. With other data types, we were able to set an initial value using the `field` and `value` attributes of a `Cell`. To set an initial selection for a

reference, field is set to the reference name while value contains the index of the item to initially select for the reference. In the purchase order, there is a selection reference named `referenceToProduct` and in the first entry for the purchase order, the value for this reference is set to 3. This means that the third item in `../inventory`, which happens to be Shampoo, will be initially selected by this reference. If the index of the item to be initially selected by a reference does not correspond to an existing index in the Table (i.e. index out of bounds), then the selection menu defaults back to `No Selection`.

Like Arrays and Classes, the attributes `collapsible`, `beginCollapsed`, and `selectionLabel` can be specified for Table elements. See Sections [2.4](#), [2.5](#), and [2.6](#), as well as Appendix [B](#), **Maui XML syntax guide**, Section [B.13](#) for discussion of these attributes.

[Next](#) [Up](#) [Previous](#)

Next: [2.8 References](#) **Up:** [2. How to Design](#) **Previous:** [2.6.2 Other Array Options](#)

Next: [2.9 Maui help buttons](#) **Up:** [2. How to Design](#) **Previous:** [2.7 Tables](#)

2.8 References

In many applications, it is convenient to have a variable obtain its value by referring to a previously defined variable. This eliminates the need for a user to enter the same information more than once, thus eliminating a major source of errors and inconsistencies. For example, a user may define several materials in an `Array`, or several inventory items in a `Table` (see the example in Section [2.7](#)), and then want to refer to a particular item as part of setting up other input. Maui contains a flexible system for specifying and controlling references. This feature of Maui is quite powerful in creating and maintaining the consistency of GUIs for complex applications. In this section we explain how to define references to elements in `Arrays`. For information on references in `Tables` see Section [2.7](#).

The hard part about using Maui references is telling Maui which object is being referenced. (To keep our vocabulary consistent, we say that the referenced object is the ``referent''.) To specify the referent, you must provide the path to the referent by navigating through the classes you have created in your XML. We refer to the following XML example to explain how paths are set to locate a referent. You may open up the example in your own text editor; the example is also contained in the file `$MAUI_HOME/Doc/tutorials/maui/XML/References.xml`. An example of the GUI displayed from this XML is in Figure [2.18](#); note that Figure [2.18](#) shows a GUI where a user has already entered information on tires, seats, and car types.

```
<Maui RootClass="CarConfig">
<Class type="CarConfig" label="Cars">
  <Action class="Maui.Interface.SaveAction" label="Save XML"/>
  <Action class="Maui.Interface.ReadAction" label="Read XML"/>
  <Action class="Maui.Interface.ViewAction" label="View XML"/>
  <Fields>
    <CarParts name="carParts" label="Car Parts"/>
    <Array name="carsArr" label="Cars">
      <Master label="$carType $carModel">
        <Car name="car" label="Car"/>
      </Master>
    </Array>
  </Fields>
</Class>
<Class type="CarParts" label="Car Parts" useTab="false" collapsible="false">
  <Fields>
    <Array name="tiresArr" label="Tire types">
      <Master label="$tireName">
        <Tire name="tire" label="Tires"/>
      </Master>
    </Array>
  </Fields>
</Class>
```

```

</Array>
<Array name="seatsArr" label="Seat types">
  <Master label="seats: $seatColor $seatFabric">
    <Seat name="seat" label="Seat"/>
  </Master>
</Array>
</Fields>
</Class>

<Class type="Tire" label="Tires">
  <Fields>
    <String name="tireName" label="Tire Brand"/>
    <String name="tireWidth" label="Width">
      <Menu options="145|155|165|175|185|195|205"/>
    </String>
    <String name="tirePerformance" label="Top preformance preference">
      <Menu options="handling|ride|treadlife"/>
    </String>
  </Fields>
</Class>

<Class type="Seat" label="Seats">
  <Fields>
    <String name="seatColor" label="Seat color">
      <Menu options="burgundy|tan|white|black|navy"/>
    </String>
    <String name="seatFabric" label="Seat fabric">
      <Menu options="leather|cloth"/>
    </String>
  </Fields>
</Class>

<Class type="Car">
  <Fields>
    <String name="carType" label="Car Type">
      <Menu options="Ford|Chevy|Dodge|Honda|Toyota|Mercedes|BMW"/>
    </String>
    <String name="carModel" label="Model"/>
    <Reference name="refToTires" label="Tires"
      path="root/carParts/tiresArr"/>
    <Reference name="refToSeat" label="Seats"
      path="../carParts/seatsArr" output="dereference"/>
    <String name="carColor" label="Car color"/>
  </Fields>
</Class>
</Maui>

```

Figure 2.17.1 References.xml

To locate a referent, we must follow a path of ownership through the XML classes. This may be done in one of two ways: from the ``root" class, or relative to the local class.

First we examine defining a `Reference` relative to the ``root" class. This is a little complicated, so let's take it one step at a time. In the preceding example, there are two references defined in the class called `Car`. The `Reference` named `refToTires` shows how to reference from the root. Note that in the first line of the example, `rootClass` is defined to be `carConfig`. Thus using `root` in the path implies `carConfig`. Our syntax requires that we use `root` in this context rather than `carConfig`. Now, note that the `root` class, `CarConfig`, owns an instance of the class `CarParts` called `carParts`. `CarParts`, in turn, owns the tire array, `tiresArr`, to which `refToTires` points. So our `Reference` from the root looks like `root/carParts/tiresArr`. Note that one may use either Unix- or Windows-style separator; thus `root\carParts\tiresArr` is also a legal `Reference` path specification.

Referencing relative to the local class is demonstrated with the `refToSeat` reference in the class `Car`. We use notation similar to that for specifying paths in a Unix file system. A `..` means ``up one level of ownership." In the `refToSeat` reference, the `..` means ``go up one level" to the owner of this class, namely `CarConfig`. Recall that `CarConfig` is also the `root` class in this example. From there we can trace back down the ownership tree just as in the reference from root example, to the `carParts` instance, which in turn owns the seat array, `seatsArr`.

As in most programming languages that use references, you can choose between reporting the ``value" of the reference (by dereferencing the variable) or just the reference. In Maui we specify the choice between output reference reporting types by using the `output` attribute. Note that in this example the `refToSeat` reference uses this attribute. The `output` attribute may have one of two values, `reference` or `dereference`. By default, the value is `reference`, which means that the path to the referent is returned in Maui's output XML. See the first highlighted line in Figure [2.19](#) for the exact XML output. If the value is `dereference`, then the output XML provides actual XML element that represents the referents value. See the second through fourth highlighted lines in Figure [2.19](#) for the exact XML output.

One other point: in both of these examples, the referent is an `Array` and the implication is that the user is to select an element of the `Array`. Sometimes, however, you may want the entire `Array` as the referent. Maui provides an attribute to do this, called `selection`. If the value of `selection="true"` (the default), then the user selects an element of the `Array`; if `selection="false"`, then the entire `Array` is the referent. If `selection="true"`, then the output XML also includes a `selection` attribute, whose value is the variable name of the item selected from the `Array`. See the first highlighted line in Figure [2.19](#).

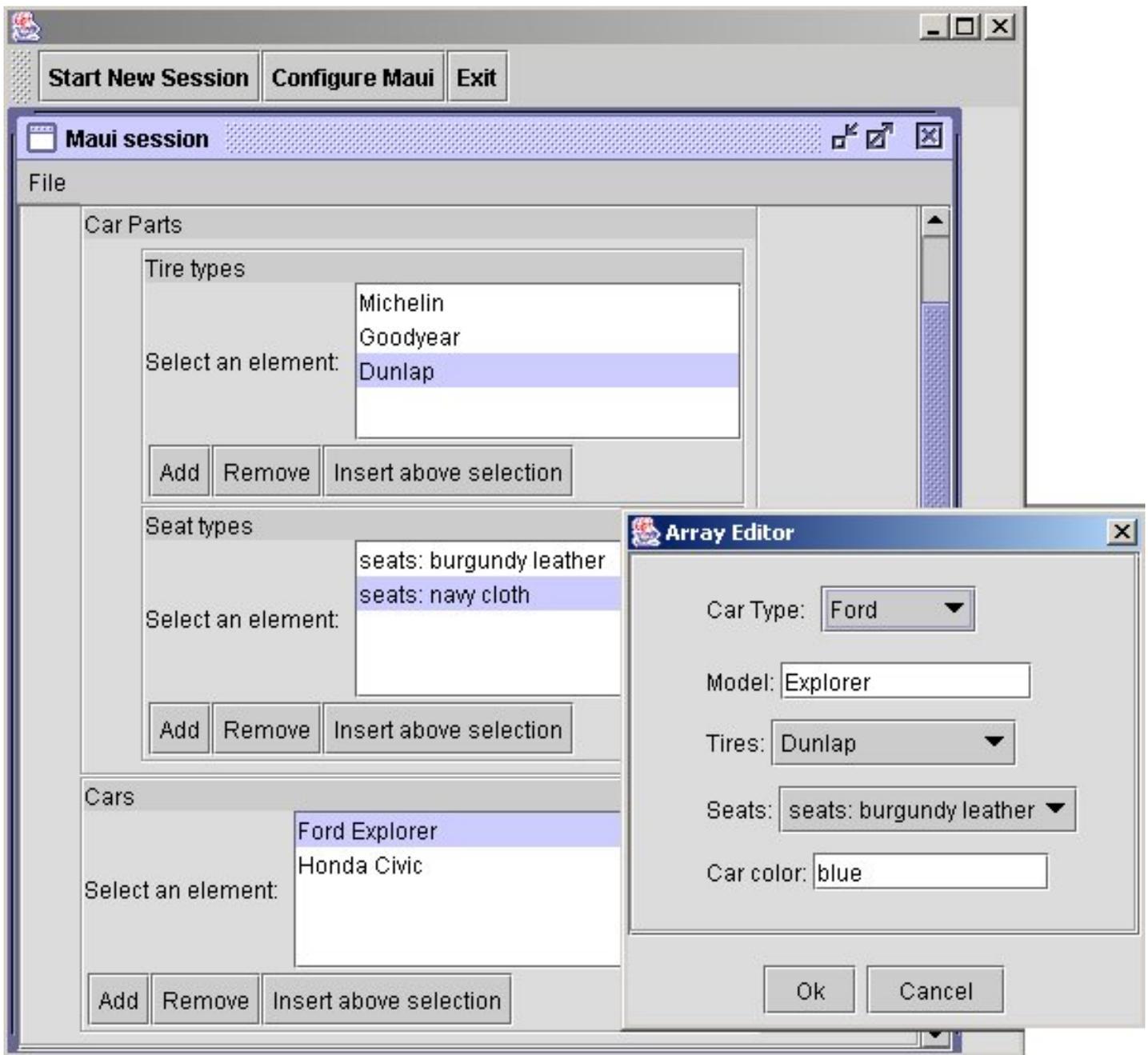


Figure 2.18: GUI example of referencing an Arrayelement. Items in the Tiresdropdown menu refer to the elements in the Tire type Array

```

<CarConfig>
  <CarParts>
    <Array name="tiresArr">
      <Tire tireName="Michelin" tirePerformance="handling" tireWidth="145"/>
      <Tire tireName="Goodyear" tirePerformance="handling" tireWidth="145"/>
      <Tire tireName="Dunlap" tirePerformance="handling" tireWidth="145"/>
    </Array>
    <Array name="seatsArr">
      <Seat seatFabric="leather" seatColor="burgundy"/>
      <Seat seatFabric="cloth" seatColor="navy"/>
    </Array>
  </CarParts>
  <Array name="carsArr">
    <Car carColor="blue" carModel="Explorer" carType="Ford">
      <Reference index="3" name="refToTires" path="root/carParts/tiresArr">
        <Tire tireName="Dunlap" tirePerformance="handling" tireWidth="145"/>
      </Reference>
      <Reference index="1" name="refToSeat" path="../carParts/seatsArr">
        <Seat seatFabric="leather" seatColor="burgundy"/>
      </Reference>
    </Car>
    <Car carColor="green" carModel="Civic" carType="Honda">
      <Reference index="1" name="refToTires" path="root/carParts/tiresArr">
        <Tire tireName="Michelin" tirePerformance="handling" tireWidth="145"/>
      </Reference>
      <Reference index="1" name="refToSeat" path="../carParts/seatsArr">
        <Seat seatFabric="leather" seatColor="burgundy"/>
      </Reference>
    </Car>
  </Array>
</CarConfig>

```

Figure 2.19: Output XML generated from the Carexample. The highlighted lines show how the output XML is rendered if the reference (highlighted yellow) or dereference (highlighted green) value is specified for the output attribute of a Reference.

For all reference variables, the GUI value of the reference is initially set to No Selection and this will get rendered to XML if nothing else is done and if the optional="true" attribute is used. (Recall that by setting optional="true" a value does not have to be selected; otherwise an error message is popped up to tell you that a choice must be made.)

If the XML contains a path that is not valid, Maui pops up a dialog box that gives you the pertinent information about the invalid path. Unfortunately, there is no way to guess what was intended, so the only recourse is for you to fix the problem and try again.

[Next](#) [Up](#) [Previous](#)

Next: [2.9 Maui help buttons](#) **Up:** [2. How to Design](#) **Previous:** [2.7 Tables](#)

[Next](#) [Up](#) [Previous](#)

Next: [2.10 Summary of Maui](#) **Up:** [2. How to Design](#) **Previous:** [2.8 References](#)

2.9 Maui help buttons

```
<Maui RootClass="HelpExample">

<Class type="HelpExample">
  <Fields>
    <Int name="num" label="Life, the Universe, and Everything" default="42">
      <Help>
        42 is the answer to the Ultimate question.
        This is from the story 'A Hitchikers Guide to the Galaxy'
        where the author, Douglas Adams, makes fun of people that
        try to answer questions that are impossible to solve. In
        the story a race of people build a super computer to find
        the answer to the question of Life, the Universe, and
        Everything, but after many millenia when it finally figured
        it out the answer was 42.
      </Help>
    </Int>
    <Double name="pi" label="A round number" default="3.14159" optional="true">
      <Help>
        A more precise calculation of this is
        3.14159265358979323846264338327950288419716939937510
        58209749445923078164062862089986280348253421170679
      </Help>
    </Double>
    <Boolean name="liar" label="This statement is a lie" default="true">
      <Help>
        This is a well known paradox much like the grandfather paradox
        in time travel.
      </Help>
    </Boolean>

<!-- *snip* -->

  </Fields>
</Class>

</Maui>
```

Figure 2.9: Help Example

Figure 2.9 shows the example from the Primitives section except help messages have been added into the XML. Help can be added to any of the MAUI data types including arrays and classes by using the Help tag. Inside the block of any variable add a help tag (<Help>) then write the text that is to be displayed as help for the end-user. New lines in the XML will remain when the help message is displayed but white space directly before and after the newline will be removed.

The help is displayed to the end-user in a pop-up dialog. The help dialog is brought up when the end-user clicks the help button next to the editor or presses F1 or alt-h when an editor has the focus. Help buttons are useful when more help is needed than can reasonably fit in a tooltip.

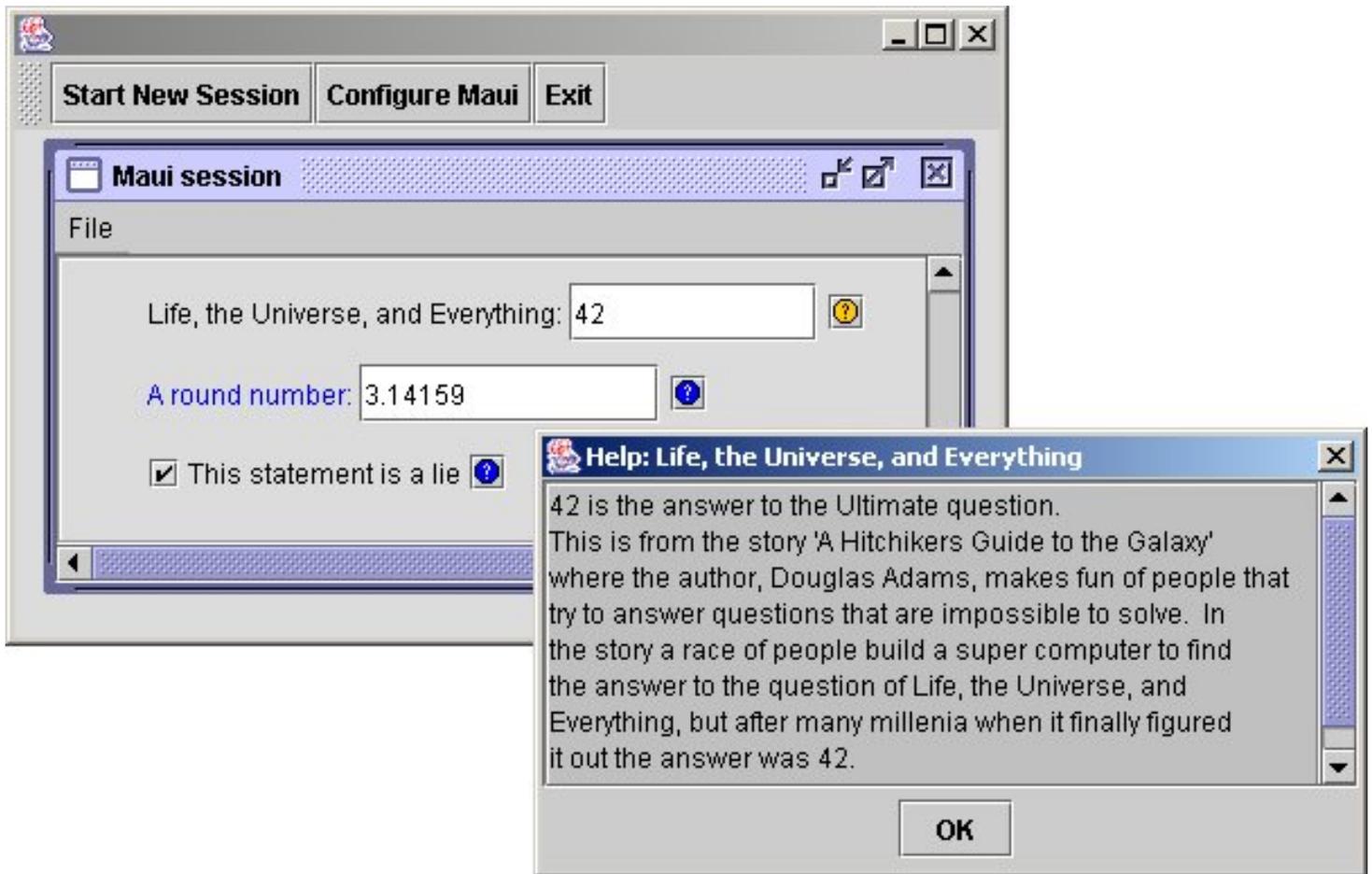


Figure 2.20:Maui GUI generated by the input shown in Figure 2.9.

Next Up Previous

Next: [2.10 Summary of Maui](#) **Up:** [2. How to Design](#) **Previous:** [2.8 References](#)

[Next](#) [Up](#) [Previous](#)**Next:** [3. Actions](#) **Up:** [2. How to Design](#) **Previous:** [2.9 Maui help buttons](#)

2.10 Summary of Maui

Maui is based on a data-driven design, in which a GUI developer provides a specification of the structure of an application and its input data. That data structure specification, optionally annotated with display information, is then used by Maui to generate a GUI.

An important feature of GUIs is that they are generally *dynamic* and *responsive*, in the sense that the GUI changes in response to user choices. A well-designed GUI should not be a static form that lists all possible options; rather, the forms and options presented should be determined and restricted by the choices already made. This seems to conflict with the idea that Maui should be data-driven - the most naive way to do a "data-driven" GUI builder would be to use a data specification to build a simple static form. The fact that Maui is not static leads to the first principle of Maui:

- Maui must be dynamic in response to user choices.

One of the key features of the Maui design is a representation of dynamic user choices through a predefined data structure specification. The device is in fact quite simple: a choice of paths through the GUI is represented as a choice between subclasses of a base class. The user selects a subclass with a drop-down menu, and from then on sees only the options appropriate to the chosen subclass. Of course, the user can backtrack and select a different path at any time.

The second principle of Maui is that it should conform with modern programming practices:

- Maui's data representation must be based on the concept of **object orientation**.

Our design parallels object-oriented programming languages such as C++ and Java. A Maui class can contain fields, which can be instances of other classes, primitive types, references, arrays, or tables. Furthermore, as with C++ or Java classes, Maui classes can derive from, and inherit certain properties of, base classes (as in Java, Maui allows only single inheritance). In Maui as in C++ or Java programming, an important design problem is to decide whether the relationship between two objects should be one of containment (HAS-A relationship) or of inheritance (IS-A relationship). This decision is particularly important for determining the dynamic behavior of the GUI generated by Maui.

In object-oriented programming languages one can associate methods, or functions, with classes. Similarly, in Maui, one can associate **actions** with classes. Actions are graphical components that can trigger an operation on the class data, for instance, saving it to disk, sending it to a server, or modifying it

in some way. See Chapter [3](#) for a description of Maui Actions.

We briefly summarize the principal objects in Maui:

- **Class** objects are containers where all other types of Maui data are contained. A Maui Class can contain as Fields instances of other classes, as well as any of the data types described in this chapter. Additionally, Actions (Chapter [3](#)) and CustomEditors (Chapter [4](#)), and AppData (Appendix [B](#), Section [B.5](#)) may be contained in a Class.
- **Fields** are the objects "owned by" a class. Fields can contain primitives, Arrays, References, Tables, Comments, or class instances. See Appendix [B](#), Section [B.4](#) for more on Fields.
- **Primitives** are data types representing a single non-compound object such as a number or a character string. Maui supports four primitive types:
 - **String** for character strings
 - **Integer** for integer variables
 - **Double** for real variables
 - **Boolean** for boolean (logical) variables

Each primitive variable will be edited by one of several graphical components. The particular component used can be specified in the primitive's definition. See the details on the XML syntax for each of these primitives in Appendix [B](#). Note that Section [B.16](#) gives information about the Menu element used by Strings for the various types of string editor rendering.

- An **Array** is an ordered list of some number of Class instance objects. The number of objects in an Array is not predetermined by Maui and need not be predetermined by the data specification; in a flexible-sized Array, objects can be added to or removed from an Array. See Appendix [B](#), Sections [B.12](#), [B.17](#), [B.18](#), [B.19](#), for details on the Array syntax.
- A **Table** is an object that contains one or more columns of primitive data types. A Table is much like an Array, except that only String, Double, Integer, Boolean, or Reference objects may define columns in a Table. See Appendix [B](#), Sections [B.13](#), [B.20](#), [B.21](#), [B.22](#), [B.23](#), for all the vocabulary associated with defining a Table.
- A **Reference** is an object that "points to" another object. In Maui we use References to be able to refer to particular elements in an Array or a Table. When the data in an Array or Table is updated, a Maui Reference to the Array or Table is also updated. This allows the user to select any current element in the Array or Table to which the Reference points. See Appendix [B](#), Section [B.14](#) for the Reference syntax.
- **AppData** is an "escape" in which the GUI developer can put non-Maui information that is needed

by the application in interpreting Maui output. `AppData` is ignored by Maui; it is passed through unchanged in Maui's output XML. See Appendix [B](#), Section [B.5](#) for the `AppData` syntax.

- A **Comment** is a Maui data type used to contain comments that are either displayed (in an uneditable form) to the user through the GUI, or are passed to the back end application as `AppData` is passed. See Appendix [B](#), Section [B.15](#).
- An **Action** is a button that is bound to both a Maui `Class` and a Java method that acts on the data associated with the Maui `Class`. See Chapter [3](#) and Appendix [B](#), Section [B.6](#) for the complete description of Maui `Actions`.
- An **Editor** is a graphical component that allows editing, through the GUI, of a given object. Every object instance in memory will have a corresponding editor on the screen. Maui provides default editors for all Maui data types, but one may create "custom editors" to represent data types in an alternative way. See Chapter [4](#) and Appendix [B](#), Section [B.7](#) for the complete description of how to create custom editors.
- Maui can **Import** definitions of `Classes` using the `Import` XML element. This is a handy Maui feature in that the developer of a Maui interface can break up their XML definitions into different logical files, and then import these definitions into the "main" file. When a Maui `Import` element appears in an XML file, the contents of the file specified in the `Import` element are inserted at exactly that point in the importing XML file. See Appendix [A](#), Section [A.1.2](#) for an example of how to use the `Import` XML. See Appendix [B](#), Section [B.3](#) for the `Import` syntax.

[Next](#) [Up](#) [Previous](#)

Next: [3. Actions](#) **Up:** [2. How to Design](#) **Previous:** [2.9 Maui help buttons](#)

[Next](#) [Up](#) [Previous](#)

Next: [3.1 Introduction](#) **Up:** [A Maui User's Guide](#) **Previous:** [2.10 Summary of Maui](#)

3. Actions

Subsections

- [3.1 Introduction](#)
- [3.2 XML for Specifying an Action](#)
 - [3.2.1 Maui Compressed XML](#)
 - [3.2.2 Maui Verbose XML](#)
 - [3.2.3 Maui Built-In Actions](#)
- [3.3 Writing Your Own Actions](#)
 - [3.3.1 The XMLObject Class](#)
 - [3.3.2 Example of a Custom Maui Action](#)
 - [3.3.3 Compiling Your Action](#)
 - [3.3.4 Configuring Maui to Find Your Action](#)
- [3.4 Suggested Exercises](#)
- [3.5 Summary](#)

[Next](#) [Up](#) [Previous](#)**Next:** [3.2 XML for Specifying](#) **Up:** [3. Actions](#) **Previous:** [3. Actions](#)

3.1 Introduction

Thus far, we have seen how we can use XML to specify the data structures and parameters of an application to create a custom GUI. One might be wondering ``now what do I do with the data in the GUI?" Within Maui there is the capability to write actions customized to your application. These actions can be used to process some subset of the data (`fields` within a particular `Class`), or to process all the data in the entire GUI. Additionally, Maui contains several built-in actions that you may wish to utilize.

As with other Maui GUI components, one must specify XML information about the action--which is associated with a button--that should appear in the GUI. Maui parses this action information, renders a button in the GUI to correspond with the action, and associates the button with some Java class which will be responsible for carrying out the action. Your responsibility as the developer is to write the Java code for the desired action. We have simplified the task of writing a Maui `Action` by providing a base class and associated methods that take care of all of the internal details and leave only the specifics of the desired action to the developer. In this chapter we will guide you through writing the input XML for requesting an action. We will then show you how to write your own `Action`, as an extension of the `MauiAction` class built into Maui.

[Next](#) [Up](#) [Previous](#)**Next:** [3.2 XML for Specifying](#) **Up:** [3. Actions](#) **Previous:** [3. Actions](#)

3.2 XML for Specifying an Action

When you wish to associate an action with the data fields in a Maui class, the `Action` XML element is used. At a bare minimum, the `class` and `label` attributes must be provided for an `Action`:

```
<Class type="MyClass">
  <Action class="TestAction" label="My Action"/>
  <Fields>
    .
    .
    .
  </Fields>
</Class>
```

Notice that an `Action` appears as a child of a `Class` element, but outside the `Fields` block. Think of Maui `Actions` as class member functions, and the data within the `Fields` block as class member data. An `Action` associated with a `Class` will only have access to the data within that `Class`.

There are other attributes that may be contained in an `Action` XML element: `verbose`, `path`, `package`, `toolTip`, and `mode`. See Section [B.6](#) for the description of each of these attributes.

In addition to the attributes the `Action` may use for configuration, an `Action` may contain an element of your choice with other information to be passed to the `MauiAction`:

```
<Class type="MyClass">
  <Action class="TestAction" label="My Action">
    <Config color="red" width="100" length="100" />
  </Action>
  <Fields>
    .
    .
    .
  </Fields>
</Class>
```

The `Config` element defined inside the action is passed through to the `MauiAction`. You can parse the information out of the `Action XML` for use in the `doAction` method of your custom action.

Of the attributes in the `Action` element, one of which to take note is the `verbose` attribute. One may set the value of the `verbose` attribute to `true` or `false`. This setting defines whether the XML representation of the data associated with this action should be rendered in Maui's "verbose" mode, or in the "compressed" mode. If no `verbose` attribute is set, by default compressed (`verbose="false"`) is assumed. Compressed and verbose XML are discussed in detail in Sections [3.2.1](#) and [3.2.2](#). The best way to view the XML in either mode is using the Maui built-in action, `ViewAction`; `ViewAction` is used in Figure [3.1](#), and is described in Section [3.2.3](#).

Subsections

- [3.2.1 Maui Compressed XML](#)
- [3.2.2 Maui Verbose XML](#)
- [3.2.3 Maui Built-In Actions](#)

[Next](#) [Up](#) [Previous](#)

Next: [3.2.1 Maui Compressed XML](#) **Up:** [3. Actions](#) **Previous:** [3.1 Introduction](#)

Next: [3.2.2 Maui Verbose XML](#) **Up:** [3.2 XML for Specifying](#) **Previous:** [3.2 XML for Specifying](#)

3.2.1 Maui Compressed XML

The Maui ``compressed" XML is the version of output XML most likely to be used for processing the data entered in the GUI. We refer to the XML as ``compressed" because of how the XML used to generate the GUI is abridged when translated into output XML.

None of the XML associated with display of the data in the GUI is retained in the compressed XML. This means that information in `CustomEditor` and `Action` elements, as well as attributes such as `label` and `useTab` are eliminated from the compressed XML.

Additionally, all Maui primitive types (`Integer`, `Double`, `String`, and `Boolean`) within a class instance are contained as attributes of the class instance, instead of as child elements. For example, the following class definition for a class named `ExampleClass` contains one primitive child, a `String` with the variable name `exampleString`:

```
<Class type="ExampleClass">
  <Action class="TestAction" label="My Action" verbose="false"/>
  <Fields>
    <String name="exampleString" label="Enter a string"/>
  </Fields>
</Class>
```

If ``compressed" XML is chosen (`verbose="false"`) as the desired output by some `Action` associated with this class, then an instance of the `ExampleClass` would be rendered as:

```
<ExampleClass exampleString="Here's a string!"/>
```

given that the user had typed `Here's a string!` in the GUI text box for the `exampleString` variable.

Other XML elements, such as class instances, `Arrays`, `Tables`, `Comments`, and `AppData` look much as they did in the input XML, and are represented as elements within the output XML.

In Figure [3.1](#), we see input XML for Maui to generate the GUI seen in Figure [3.2](#). This XML is in the file in `$MAUI_HOME/Doc/tutorials/maui/XML/ActionIn.xml`. Values have been entered in the resulting GUI shown in Figure [3.2](#). The ``compressed" output XML rendering of the data in Figure [3.2](#) is shown in Figure [3.3](#). A copy of the compressed output XML is in the file `$MAUI_HOME/Doc/tutorials/maui/XML/ActionCompressedOut.xml`.

```

<Maui RootClass="MyActionExample">

  <Class type="MyActionExample">
    <Action class="Maui.Interface.ViewAction" label="View Compressed XML"
      verbose="false"/>
    <Action class="Maui.Interface.SaveAction" label="Save Inputs (verbose)"
      verbose="true"/>
    <Action class="Maui.Interface.SaveAction" label="Save Inputs (compressed)"
      verbose="false"/>
    <Action class="Maui.Interface.ReadAction" label="Read Inputs (from XML)"/>
    <Fields>
      <Point name="a" label="point A" useTab="false"/>
      <Array name="partsArr" label="Parts">
        <Master label="$partName ($materialName , $partLen in.)">
          <Part name="part"/>
        </Master>
      </Array>
      <Table name="inventory" label="Available Products">
        <Header name="product" label="$productName">
          <String name="productName" label="Product"/>
          <Double name="price" label="Price in $"/>
        </Header>
      </Table>
      <Reference name="refToPart" label="Part"
        path="partsArr" output="dereference"/>
      <Int name="num" label="Life, the Universe, and Everything" default="42"/>
      <Boolean name="liar" label="This statement is a lie" default="true"/>
      <String name="text" label="Famous last words" columnWidth="20"
        default="Hey, what's this red button do?"/>
    </Fields>
  </Class>

  <Class type="Part">
    <Fields>
      <String name="partName" label="name of part">
        <Menu options="bolt|nail|screw"/>
      </String>
      <String name="materialName" label="name of material"/>
      <Double name="partLen" label="length (in inches)"/>
    </Fields>
  </Class>

  <Class type="Point">
    <Fields>
      <Double name="x" label="x coordinate"/>
      <Double name="y" label="y coordinate"/>
    </Fields>
  </Class>

```

</Maui>

Figure 3.1:Maui input XML example of a Class that uses Actions.

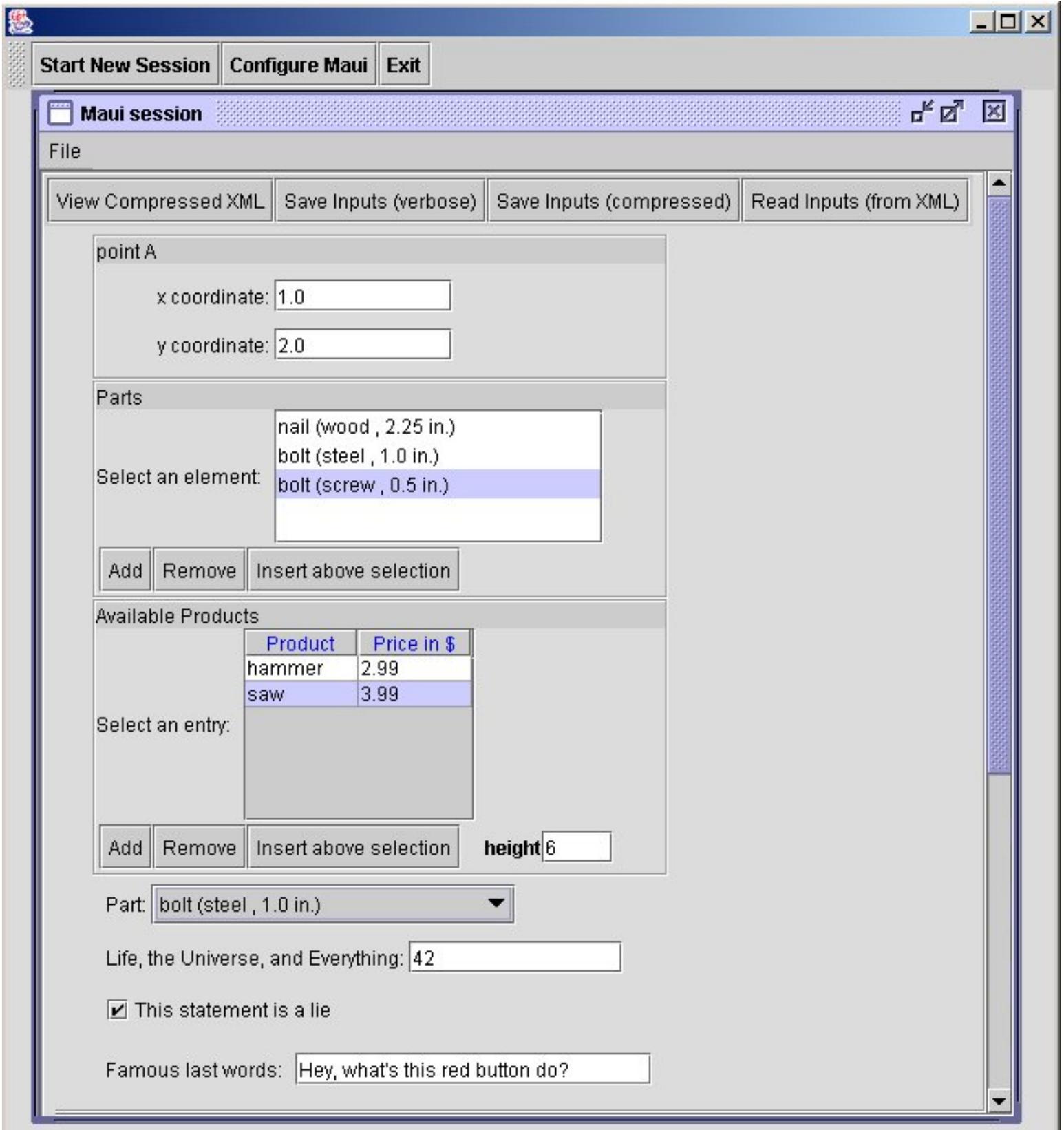


Figure 3.2:Maui GUI demonstrating Action buttons.

```
<MyActionExample num="42" liar="true" text="Hey, what's this red button do?">
  <Point x="1.0" y="2.0"/>
  <Array name="partsArr">
    <Part partName="nail" partLen="2.25" materialName="wood"/>
    <Part partName="bolt" partLen="1.0" materialName="steel"/>
    <Part partName="bolt" partLen="0.5" materialName="screw"/>
  </Array>
  <Table name="inventory">
    <Entry price="2.99" productName="hammer"/>
    <Entry price="3.99" productName="saw"/>
  </Table>
  <Reference index="2" name="refToPart" path="partsArr">
    <Part partName="bolt" partLen="1.0" materialName="steel"/>
  </Reference>
</MyActionExample>
```

Figure 3.3:Maui compressed output XML example.

For this example we used almost all of the Maui data types to show how the Maui input XML (Figure [3.1](#)) is transformed into the Maui output XML (Figure [3.3](#)) for each data type.

Note that an `Action XML` element may appear within any `Class` in the Maui input XML heirarchy. The compressed XML rendering of the `Class` will only include data associated with the particular instance of the class from which the action was performed (that is, from which the action button was pressed).

[Next](#) [Up](#) [Previous](#)

Next: [3.2.2 Maui Verbose XML](#) **Up:** [3.2 XML for Specifying](#) **Previous:** [3.2 XML for Specifying](#)

Next: [3.2.3 Maui Built-In Actions](#) **Up:** [3.2 XML for Specifying](#) **Previous:** [3.2.1 Maui Compressed XML](#)

3.2.2 Maui Verbose XML

The Maui ``verbose" XML is primarily used for saving the current state of the GUI, and re-rendering a Maui GUI by reading the verbose XML saved in a file. Because the use for which the verbose XML was designed was re-rendering the GUI, Actions using verbose XML seem most useful when placed in the Maui `RootClass`. Recall that the `RootClass` is the class that will appear in the main panel of the GUI, and is the class in which all other class instances or other field data must be contained.

The verbose XML contains instance information for all data types within Maui. This instance information, coupled with the corresponding Maui input XML, is enough information for Maui to re-render the appropriate GUI interface, with default values for each instance that has been read from the verbose XML. The XML from which a Maui GUI was originally rendered provides display information for the GUI, such as labels for each field, menu vs. text box components for `String` data, and tabbed panes vs. un/collapsible frames to represent class instances. The verbose XML provides the values for each instance within the GUI, information on the containment of that data (via the attribute `ownerType`), and for class instances, information on where the class lies within an inheritance heirarchy (via the attribute `base`). Because the ``verbose" XML is not typically the format of the data you will be interested in using for manipulating field data for an application, we will not spend more time here explaining the details of the verbose XML syntax.

You might use verbose XML if you were trying to update values in the GUI from data that is external to Maui. For instance, if you would like to read some data saved in a text file (not in Maui XML format) into your GUI, you could ``create" the verbose XML representation of the data. You could then insert the verbose XML you generated into the verbose XML that comes out of the GUI. Finally, you could make calls to update the GUI with the new XML. See Section [3.3.1](#) for details on how to manipulate the XML for such purposes.

In Figure [3.4](#), we see the output XML that would be produced in verbose mode based on the data entered in the GUI in Figure [3.2](#). A copy of the verbose output XML is in the file

`$MAUI_HOME/Doc/tutorials/maui/XML/ActionVerboseOut.xml`.

```
<MyActionExample name="name0" altName="MyActionExample" base="MyActionExample"
ownerType="Maui">
  <Point name="a" altName="Point" base="Point" ownerType="MyActionExample">
    <Double default="1.0" name="x" ownerType="Point">
      </Double>
    <Double default="2.0" name="y" ownerType="Point">
      </Double>
    </Point>
  <Array name="partsArr">
    <Contents>
      <Item index="0">
        <Part name="part" altName="Part" base="Part" ownerType="MyActionExample">
          <String default="nail" name="partName" ownerType="Part">
            </String>
          </Part>
        </Item>
      </Contents>
    </Array>
  </MyActionExample>
```

```

        <String default="wood" name="materialName" ownerType="Part">
        </String>
        <Double default="2.25" name="partLen" ownerType="Part">
        </Double>
    </Part>
</Item>
<Item index="1">
    <Part name="part" altName="Part" base="Part" ownerType="MyActionExample">
        <String default="bolt" name="partName" ownerType="Part">
        </String>
        <String default="steel" name="materialName" ownerType="Part">
        </String>
        <Double default="1.0" name="partLen" ownerType="Part">
        </Double>
    </Part>
</Item>
<Item index="2">
    <Part name="part" altName="Part" base="Part" ownerType="MyActionExample">
        <String default="bolt" name="partName" ownerType="Part">
        </String>
        <String default="screw" name="materialName" ownerType="Part">
        </String>
        <Double default="0.5" name="partLen" ownerType="Part">
        </Double>
    </Part>
</Item>
</Contents>
</Array>
<Table name="inventory">
    <Header label="$productName" name="product">
        <String label="Product" name="productName"/>
        <Double label="Price in $" name="price"/>
    </Header>
    <Entries>
        <Entry>
            <Cell value="hammer" field="productName"/>
            <Cell value="2.99" field="price"/>
        </Entry>
        <Entry>
            <Cell value="saw" field="productName"/>
            <Cell value="3.99" field="price"/>
        </Entry>
    </Entries>
</Table>
<Reference index="2" name="refToPart" path="partsArr">
    <Part partName="bolt" partLen="1.0" materialName="steel"/>
</Reference>
<Integer default="42" name="num" ownerType="MyActionExample">
</Integer>
<Boolean default="true" name="liar" ownerType="MyActionExample">

```

```
</Boolean>  
  <String default="Hey, what's this red button do?" name="text" columnWidth="20"  
ownerType="MyActionExample">  
  </String>  
</MyActionExample>
```

Figure 3.4:Maui verbose output XML example.

[Next](#) [Up](#) [Previous](#)

Next: [3.2.3 Maui Built-In Actions](#) **Up:** [3.2 XML for Specifying](#) **Previous:** [3.2.1 Maui Compressed XML](#)

Next Up Previous

Next: [3.3 Writing Your Own](#) **Up:** [3.2 XML for Specifying](#) **Previous:** [3.2.2 Maui Verbose XML](#)

3.2.3 Maui Built-In Actions

Maui contains several built-in actions that you may use in your applications. When these actions do not meet your needs, then you will also be able to write your own actions.

Notice that four Maui Actions are specified within the `MyActionExample` class in Figure [3.1](#). Also note that though we are specifying four actions, only three different action classes (`ViewAction`, `SaveAction`, and `ReadAction`) are used. The actions with the labels `Save Inputs (verbose)` and `Save Inputs (compressed)` differ only in that the `verbose` attribute is set to `true` for one and `false` for the other.

All of the Maui built-in actions are in package `Maui.Interface`. A description of Java packages is not within the scope of this manual, so if you are unfamiliar with packages, refer to a Java reference for information. To use a built-in Maui action, your Action specification within your Maui XML should always contain a `class` attribute such as

```
class="Maui.Interface.MauiAction"
```

where *MauiAction* is the name of the built-in Maui action class you wish to use. There are many built-in actions in the `Maui.Interface` package, but only a few of them might be of use to the Maui GUI developer:

ReadAction

This action is used to read previously saved XML from a file for updating the current Maui GUI. If you plan to let the GUI user save values they have entered into the GUI, then later (in another Maui session, or just later this Maui session) read those saved values back into the GUI, this is the action to use. **ReadAction** assumes that the XML being read (*a*) had been saved from the GUI for this particular application at some point in the past, and (*b*) was saved in verbose mode (that is, using the **SaveAction** with attribute `verbose` set to `true`).

SaveAction

This action is used to save to a file the current state of the data in a Maui GUI in either verbose (`verbose="true"`) or compressed (the default, or `verbose="false"`) XML format.

SubmitAction

This action can be used to pass XML from the GUI to a piece of ``handler" code. This action predates the general use of `MauiActions`. Instead of using **SubmitAction**, we recommend that

for any type of XML processing, you write your own `MauiAction` as an extension of the `MauiAction` class.

ViewAction

This action is used to display the current state of the data in the GUI in either verbose or compressed XML. The type of XML to be displayed is selected by using the `verbose` attribute. Additionally, the `mode` attribute may be used to format the display. If no `mode` value is set, the same XML that would be written to a file using the **SaveAction** is displayed in a pop-up window. If `mode` is set to `tree`, then a collapsible view of the element containment tree is used for either the `verbose` or `compressed XML` specified by the `verbose` attribute.

The **ViewAction** is helpful when developing action code that parses and manipulates the XML data that comes out of Maui. The developer can see the format of the XML that will be produced by Maui, and use the XML manipulation tools built into Maui to handle the XML. This has advantages over using the **SaveAction** in that you can dynamically change your entries in the GUI, and immediately view the changes to the output XML by using the **ViewAction**. This is especially handy when one uses subclassing within the GUI (see Section [2.5](#)).

The source code for all the Maui built-in actions can be found in `$MAUI_HOME/Java/src/Maui/Interface`. The source for all these actions would be good examples of how to develop actions as extensions of `MauiAction`.

[Next](#) [Up](#) [Previous](#)

Next: [3.3 Writing Your Own](#) **Up:** [3.2 XML for Specifying](#) **Previous:** [3.2.2 Maui Verbose XML](#)

[Next](#) [Up](#) [Previous](#)**Next:** [3.3.1 The XMLObject Class](#) **Up:** [3. Actions](#) **Previous:** [3.2.3 Maui Built-In Actions](#)

3.3 Writing Your Own Actions

As we have mentioned throughout this chapter, Maui provides an abstract class called `MauiAction` from which a GUI developer can derive custom actions for the developer's application. `MauiAction` contains a small number of predefined functions. See the javadoc documentation (on the web, or in the directory `$MAUI_HOME/Doc/javadoc/maui/Maui/Interface/MauiAction.html`), as well as the `MauiAction` source code (in `$MAUI_HOME/Java/src/Maui/Interface/MauiAction.java`) for more information on the `MauiAction` class.

The one method in `MauiAction` that you must extend in your derived class is `doAction(ActionEvent, XMLObject)`. This is where the meat of your action should go. You can write additional helper methods in your derived class, but nothing besides `doAction` is required.

There are two parameters required by `doAction`. The `ActionEvent` is passed to the `doAction` by the calling `MauiActionButton`. Unless you have specific reasons for wanting to get information about the `ActionEvent` that triggered this action, you can ignore the contents of the `ActionEvent` parameter in the substance of your `doAction` code.

The second parameter required by `doAction` is very important. Contained within the `XMLObject` parameter of `doAction` is the verbose or compressed XML data that has been passed from the GUI when the action was fired.

Subsections

- [3.3.1 The XMLObject Class](#)
 - [3.3.2 Example of a Custom Maui Action](#)
 - [3.3.3 Compiling Your Action](#)
 - [3.3.4 Configuring Maui to Find Your Action](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [3.3.1 The XMLObject Class](#) **Up:** [3. Actions](#) **Previous:** [3.2.3 Maui Built-In Actions](#)

3.3.1 The XMLObject Class

`XMLObject` is a Maui class with utility methods for manipulating an XML object. There are many methods within the `XMLObject` class for modifying and extracting information from existing XML objects, or for creating XML objects. Thus we can add and remove attributes and children of an `XMLObject` as necessary using the appropriate methods. The methods for accessing information tend to start with `get`, while most of the creation method names start with `add`. To modify an existing `XMLObject`, use the methods with `add` and `remove` prefixes.

Note that XML attributes are represented as key/value string pairs. For example, `("default" , "1.0")` has key `default` and value `1.0`. Also, a child of an `XMLObject` is another `XMLObject`. In Figure [3.5](#) we show some examples of generating `XMLObjects`. In Figure [3.6](#) we see the XML that would be generated by the code in Figure [3.5](#). In Figure [3.7](#) we see methods used to access values in the `XMLObject` in Figures [3.5](#) and [3.6](#). Finally, Figure [3.8](#) shows the screen output from the `System.out.println` calls in Figure [3.7](#).

```

/* Create a new XMLObject with the tab "myObject" */
XMLObject obj = new XMLObject("myObject");

/* Add an attribute to the "myObject" XML */
obj.addAttribute("name", "joe");

/* Create a new XMLObject that will be the child of "myObject" */
XMLObject kid = new XMLObject("myChild");

/* Add an attribute to the child. */
kid.addAttribute("age", "5");

/* Add kid to obj as a child. */
obj.addChild(kid);

```

Figure 3.5: Example of using the creation methods in the `XMLObject` class.

```

<myObject name="joe">
  <myChild age="5"/>
</myObject>

```

Figure 3.6:Example of XML representation of XMLObject in Figure [3.5](#).

```

/* Print out the XML object, "obj." Also print out the tag associated with
   "obj" and the value associated with the attribute "age" of "kid." */

String tag = obj.getTag();
String years = obj.getChild("myChild").getAttribute("age");
System.out.println("The XML object contained in obj:\n"
    + obj.toString() + "\n\n");
System.out.println("The value of the tag is " + tag
    + " and age = " + years);

```

Figure 3.7:Example of using the accessor methods in the XMLObject class.

```

The XML object contained in obj:
<myObject name="joe">
  <myChild age="5"/>
</myObject>

```

Figure 3.8:Screen output from running the code snippets in Figures [3.5](#) and [3.7](#).

See the javadoc documentation (on the web, or in the directory \$MAUI_HOME/Doc/javadoc/idea/XML/XMLObject.html), as well as the XMLObject source code (in \$MAUI_HOME/Java/src/XML/XMLObject.java) for more information on the XMLObject methods.

[Next](#) [Up](#) [Previous](#)

Next: [3.3.2 Example of a](#) **Up:** [3.3 Writing Your Own](#) **Previous:** [3.3 Writing Your Own](#)

[Next](#)
[Up](#)
[Previous](#)

Next: [3.3.3 Compiling Your Action](#)
Up: [3.3 Writing Your Own](#)
Previous: [3.3.1 The XMLObject Class](#)

3.3.2 Example of a Custom Maui Action

We should now have enough information to generate an example custom `Action`. In the following Java code we see the definition of a class, `TestAction`, derived from `MauiAction`. `TestAction` contains a `doAction` method that prints out the XML rendering of the GUI data, and some information depending on the output style (`verbose` or `compressed`). This action uses several of the methods in the `XMLObject` class. Additionally, it makes a call to the `requiresVerboseXML` method, which is inherited by the `TestAction` class from `MauiAction`.

```

/* If this action is part of a package, place the package command here. */

import Maui.Interface.*;
import XML.*;
import java.awt.event.*;
import javax.swing.JOptionPane;

/** This is an example of a custom MauiAction. It extends the base
 * class, MauiAction. The actual work is done in the doAction method.
 */
public class TestAction extends MauiAction
{
    /** The method that carries out the work of the action. In this
     * case, it merely prints out a message in a JOptionPane
     *
     * @param e an Action event that triggers this method
     * @param body The XML object representing the state of the GUI
     * when the action is performed.
     */

    public void doAction(ActionEvent e, XMLObject body)
    {
        String msg = "In TestAction: The XML generated by this GUI is:\n\n"
            + body.toString()
            + "\n*****\n";

        /* if the button with "verbose" is selected: */
        if(requiresVerboseXML())
        {

```

```

XMLObject pointObj= body.getChild("Point");
int numPointKids = pointObj.numChildren();
int doubleCount = 0;

for(int i = 0; i < numPointKids; i++)
{
    XMLObject pointChild = pointObj.getChild(i);
    if(pointChild.getTag().equals("Double"))
        doubleCount++;
}
msg = msg.concat("Verbose XML: The Point child of MyActionExample "
    + "contains " + doubleCount
    + " Double elements.\n");
}
/* otherwise, if "compressed" is selected: */
else
{
    int numBodyAttributes = body.numberOfAttributes();

    msg = msg.concat("Compressed XML: The MyActionExample element has "
        + numBodyAttributes + " attributes:\n"
        + "\tnum = " + body.getAttribute("num") + "\n"
        + "\tliar = " + body.getAttribute("liar") + "\n"
        + "\ttext = " + body.getAttribute("text") + "\n\n");

    XMLObject arrayObj = body.getChild("Array");
    int numParts = arrayObj.getChildren("Part").length;

    msg = msg.concat("Compressed XML: The MyActionExample element has "
        + numParts + " Parts elements\n"
        + "in the Array child element.\n");
}

/* Display the information we've gathered in a JOptionPane when the
   Action button has been pushed. */
JOptionPane.showMessageDialog(null, msg, "TestAction",
JOptionPane.INFORMATION_MESSAGE);
}

/* We want to be able to run the action without requiring that the
   user has entered any data into the GUI (even in the required fields...).
   So we override this method to return false, thus allowing the
   "doAction" to proceed even without all the required input. */
public boolean requiresValidXML()

```

```
{ return false; }  
}
```

Figure 3.8.1 Custom Maui Action

See `$MAUI_HOME/Doc/tutorials/maui/Java/TestAction.java` for the code from the example action (above). The `TestAction.java` code can be used as a template for writing your actions.

The Java code can be placed anywhere within your directory structure, and may optionally be part of a Java package. Your `Action` class must be derived from the `MauiAction` class, and therefore must import the appropriate files. At a minimum, you must import `Maui.Interface.*` to get the `MauiAction` class definition, `XML.*` to get the `XMLObject` class definition, and `java.awt.event.*` to get the `ActionEvent` class definition.

As we mentioned before, the only method you must override in your `Action` class is the `doAction`. We have additionally overridden the `requiresValidXML` method so that when the GUI user pushes the button to execute the action, the action is executed whether or not the user has entered all the non-optional data, and whether or not the data is of the correct type.

[Next](#) [Up](#) [Previous](#)

Next: [3.3.3 Compiling Your Action](#) **Up:** [3.3 Writing Your Own](#) **Previous:** [3.3.1 The XMLObject Class](#)

[Next](#) [Up](#) [Previous](#)**Next:** [3.3.4 Configuring Maui to Up](#) **Up:** [3.3 Writing Your Own](#) **Previous:** [3.3.2 Example of a](#)

3.3.3 Compiling Your Action

To compile the code, first make sure that you are in the directory where `TestAction.java` is located (`$MAUI_HOME/Doc/tutorials/maui/Java/`). Then type the command

UNIX:

```
javac -classpath .:$MAUI_HOME/Java/classes/maui.jar:\
$MAUI_HOME/Java/classes/xerces.jar TestAction.java
```

WINDOWS:

```
javac -classpath .;"%MAUI_HOME%\Java\classes\maui.jar";
"%MAUI_HOME%\Java\classes\xerces.jar"
"%MAUI_HOME%\Doc\tutorials\maui\Java\TestAction.java"
```

(NOTE: Because the command is too long to fit on one line, it has been separated it into multiple lines to make the command easier to read. However, for Windows to execute this command, the entire command must be typed on one line.)

[Next](#) [Up](#) [Previous](#)**Next:** [3.3.4 Configuring Maui to Up](#) **Up:** [3.3 Writing Your Own](#) **Previous:** [3.3.2 Example of a](#)

Next: [3.4 Suggested Exercises](#) **Up:** [3.3 Writing Your Own](#) **Previous:** [3.3.3 Compiling Your Action](#)

3.3.4 Configuring Maui to Find Your Action

To enable Maui to use your new action, you must now include the appropriate `Action` XML element in your Maui input XML. In Figure [3.9](#), we use a modified version of the XML in Figure [3.1](#). The XML file that uses the `TestAction` is in `$MAUI_HOME/Doc/tutorials/maui/XML/ActionIn2.xml`. This new XML contains two actions, labeled `Test Action (verbose)` and `Test Action (compressed)`. The class used for both actions is the `TestAction` we just wrote, but one action requires verbose XML, while the other requires compressed XML. Figure [3.10](#) shows the results of pressing the `Test Action (compressed)` button in the resulting GUI.

```
<Maui RootClass="MyActionExample">

  <Class type="MyActionExample">
    <Action class="TestAction" label="Test Action (verbose)"
      verbose="true"/>
    <Action class="TestAction" label="Test Action (compressed)"
      verbose="false"/>
    <Fields>
      <Point name="a" label="point A" useTab="false"/>
      <Array name="partsArr" label="Parts">
        <Master label="$partName ($materialName , $partLen in.)">
          <Part name="part"/>
        </Master>
      </Array>
      <Reference name="refToPart" label="Part"
        path="partsArr" output="dereference"/>
      <Int name="num" label="Life, the Universe, and Everything" default="42"/>
      <Boolean name="liar" label="This statement is a lie" default="true"/>
      <String name="text" label="Famous last words" columnWidth="20"
        default="Hey, what's this red button do?"/>
    </Fields>
  </Class>

  <Class type="Part">
    <Fields>
      <String name="partName" label="name of part">
        <Menu options="bolt|nail|screw"/>
      </String>
      <String name="materialName" label="name of material"/>
      <Double name="partLen" label="length (in inches)"/>
    </Fields>
  </Class>
```

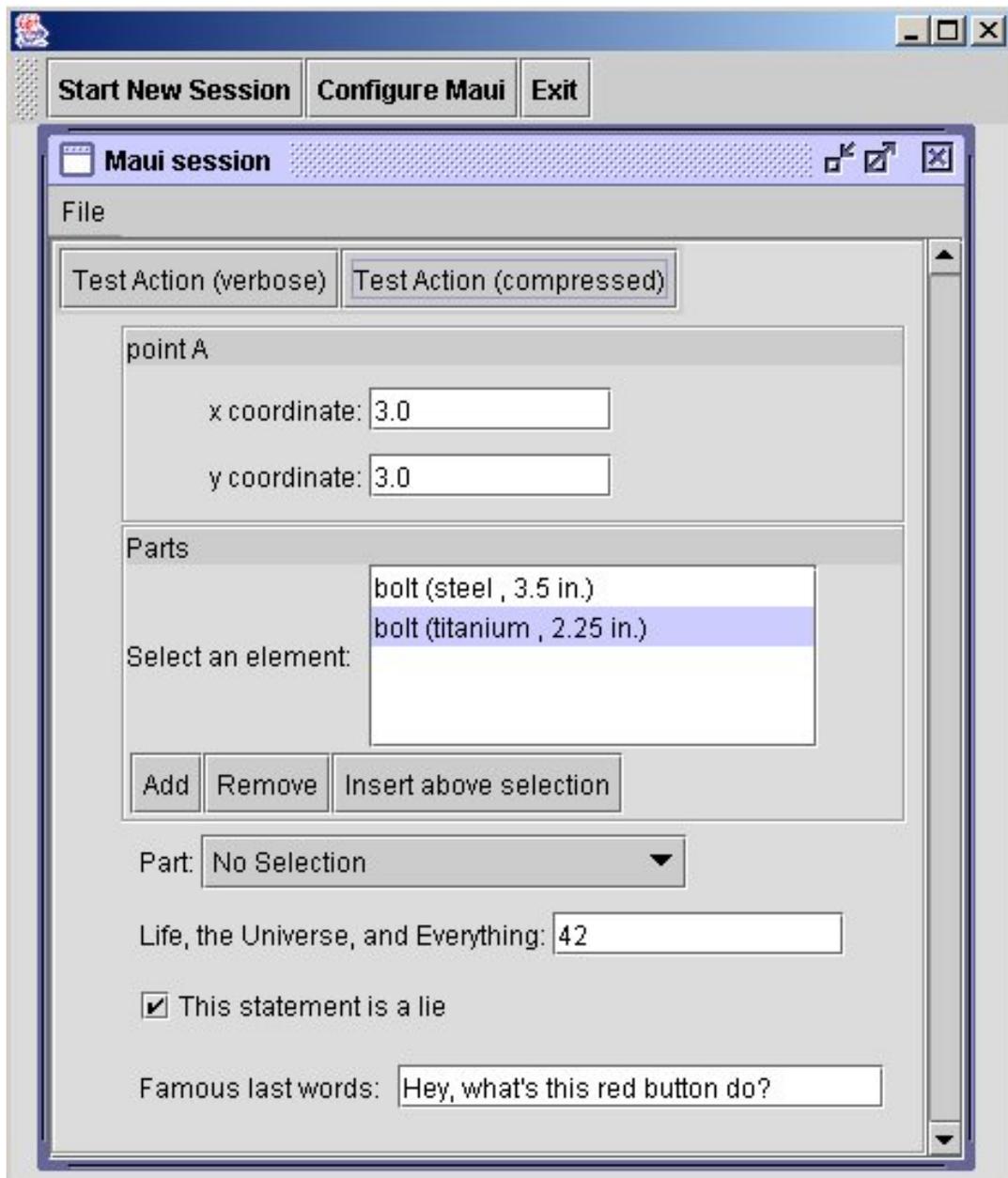
```

<Class type="Point">
  <Fields>
    <Double name="x" label="x coordinate"/>
    <Double name="y" label="y coordinate"/>
  </Fields>
</Class>

</Maui>

```

Figure 3.9:Maui input XML using our new TestAction.



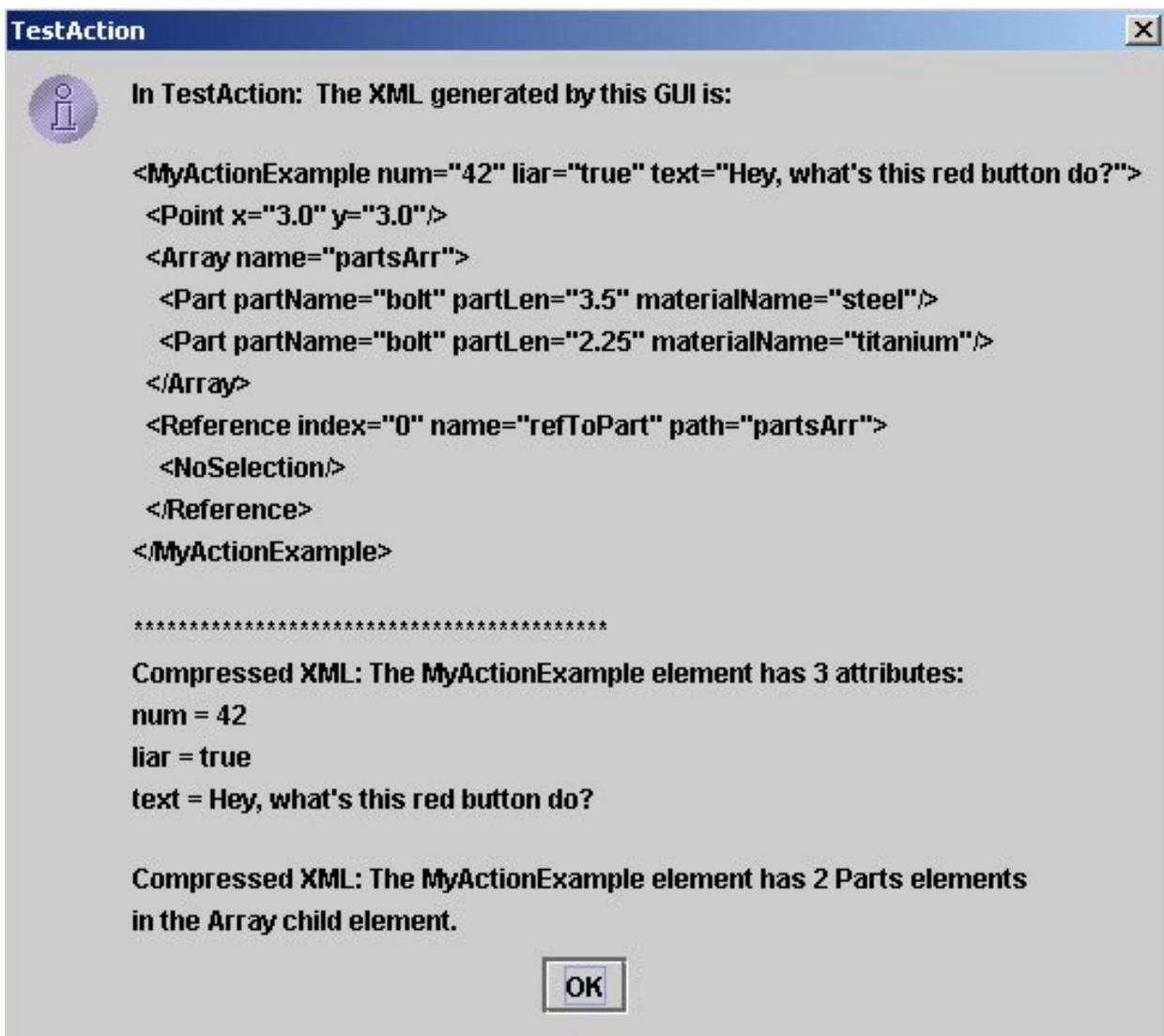


Figure 3.10:Maui GUI demonstrating result of pressing the Test Action (compressed) Button.

There are three ways to tell Maui where to find the TestAction class you compiled. See Section [5.4](#) in Chapter [5](#), **Configure Maui** for the full details on specifying path and package information to Maui for loading custom classes.

[Next](#) [Up](#) [Previous](#)

Next: [3.4 Suggested Exercises](#) **Up:** [3.3 Writing Your Own](#) **Previous:** [3.3.3 Compiling Your Action](#)

[Next](#) [Up](#) [Previous](#)**Next:** [3.5 Summary](#) **Up:** [3. Actions](#) **Previous:** [3.3.4 Configuring Maui to](#)

3.4 Suggested Exercises

1. Add an XML element inside your XML Action that contains some sort of configuration information for your extension of MauiAction. An example is

```
<Action class="TestAction" label="My Action">
  <Config color="red" width="100" length="100" />
</Action>
```

Use the XMLObject methods to extract the values from the attributes (for example, color, width, and length). Note that the Config element in the example above is a child element of the Action XML. The Action XMLObject is contained in the config_ field of the MauiAction class.

2. Using the XML in Figure [3.9](#) (the XML is also contained in the file \$MAUI_HOME/Doc/tutorials/maui/XML/ActionIn2.xml), try moving the Action XML for the TestAction inside the Part class or the Point class. View the GUI and the result of pressing the action buttons. So that the messages printed by the action reflect the new location of the buttons, modify the doAction method in TestAction.java (see \$MAUI_HOME/Doc/tutorials/maui/Java/TestAction.java) to get information about the fields specific to the Part or Point class.

[Next](#) [Up](#) [Previous](#)**Next:** [3.5 Summary](#) **Up:** [3. Actions](#) **Previous:** [3.3.4 Configuring Maui to](#)

[Next](#) [Up](#) [Previous](#)**Next:** [4. Custom Editors](#) **Up:** [3. Actions](#) **Previous:** [3.4 Suggested Exercises](#)

3.5 Summary

This chapter has covered information on the process of developing actions customized to the data for a Maui GUI. The developer is responsible for generating the appropriate Java code for executing the action, as well as placing the appropriate XML `Action` in the Maui input file.

[Next](#) [Up](#) [Previous](#)**Next:** [4. Custom Editors](#) **Up:** [3. Actions](#) **Previous:** [3.4 Suggested Exercises](#)

[Next](#) [Up](#) [Previous](#)

Next: [4.1 Introduction](#) **Up:** [A Maui User's Guide](#) **Previous:** [3.5 Summary](#)

4. Custom Editors

Subsections

- [4.1 Introduction](#)
 - [4.2 Using a Custom Editor: The `FilenameEditor` for Strings](#)
 - [4.3 Writing Your Own Custom Editors](#)
 - [4.3.1 The Structure of Maui Data](#)
 - [4.3.2 Steps to Writing Your Own Custom Editor](#)
 - [4.3.3 Example of a Maui Custom Editor](#)
 - [4.3.4 Compiling Your Custom Editor](#)
 - [4.3.5 Configuring Maui to Find Your Custom Editor](#)
 - [4.4 Summary](#)
-

[Next](#) [Up](#) [Previous](#)**Next:** [4.2 Using a Custom](#) **Up:** [4. Custom Editors](#) **Previous:** [4. Custom Editors](#)

4.1 Introduction

As you have seen, Maui has a default way to display all of its data objects. For example, integers, doubles, and strings are typically rendered as a text areas, where the user enters a value by typing in the text area. Even though Maui has default ways of representing many data types in the GUI, you can choose to create your own display mechanism for any data object by writing a custom editor. Custom editors can make it possible for you to enter a path to a file using a file browser rather than by typing in the path string, or to set the value of an integer by using a slider bar. In this chapter we show you how you can use a custom editor that is already built into Maui, and how you can create such custom editors for your own applications. First, we give an example of using the file browser custom editor for a `String`. Then we will demonstrate how to write your own custom editor. Writing your own custom editor requires some familiarity with Java Swing components, and it helps to understand in a little more detail the underlying data structures in Maui. We assume that you do know Java Swing; we will briefly discuss Maui's structure as it pertains to writing custom editors later in this chapter. We then discuss in some detail the example custom editor presented in section [4.3.3](#).

[Next](#) [Up](#) [Previous](#)**Next:** [4.2 Using a Custom](#) **Up:** [4. Custom Editors](#) **Previous:** [4. Custom Editors](#)

Next Up Previous

Next: [4.3 Writing Your Own](#) Up: [4. Custom Editors](#) Previous: [4.1 Introduction](#)

4.2 Using a Custom Editor: The FilenameEditor for Strings

Built into Maui is a `String` editor, called `FilenameEditor`, defined by the Java code in `$MAUI_HOME/Java/src/Maui/Editors/FilenameEditor.java`. By default, a `String` XML object is rendered as a text box, unless the `String` element contains a `Menu` element. However, if the intention of the developer is to have the `String` entry represent a path to a file, then the `FilenameEditor` can be accessed as a `CustomEditor`. The `FilenameEditor` provides the user with a file browser for locating the desired file.

To use a custom editor for any type of Maui object, a `CustomEditor` XML element must be contained within the Maui data element that requires the custom editor. The `CustomEditor` element points to the Java class with which that editor may be represented. In [Figure 4.1](#), we see an example of how to specify that the `FilenameEditor` representation of a `String` should be used, instead of any of the standard `String` editors. [Figure 4.2](#) shows the GUI rendering of a `String` with a `FilenameEditor` custom editor. When the user clicks on the `Browse` button, the file dialog box appears.

```
<String name="TestFilename" label="filename" default="">
  <CustomEditor name="Maui.Editors.FilenameEditor">
    <folderName default="c:/junk"/>
    <filename default="junk.gif"/>
    <openOrSaveFile default="open"/>
    <title default="Select a file"/>
    <filenameExtensions default="gif jpg jpeg"/>
    <onErrorRedisplayFileDialogBox default="no"/>
  </CustomEditor>
</String>
```

Figure 4.1: Example of XML specification of the `CustomEditor` for a `String`; the custom editor is defined in the `FilenameEditor.jav` file.

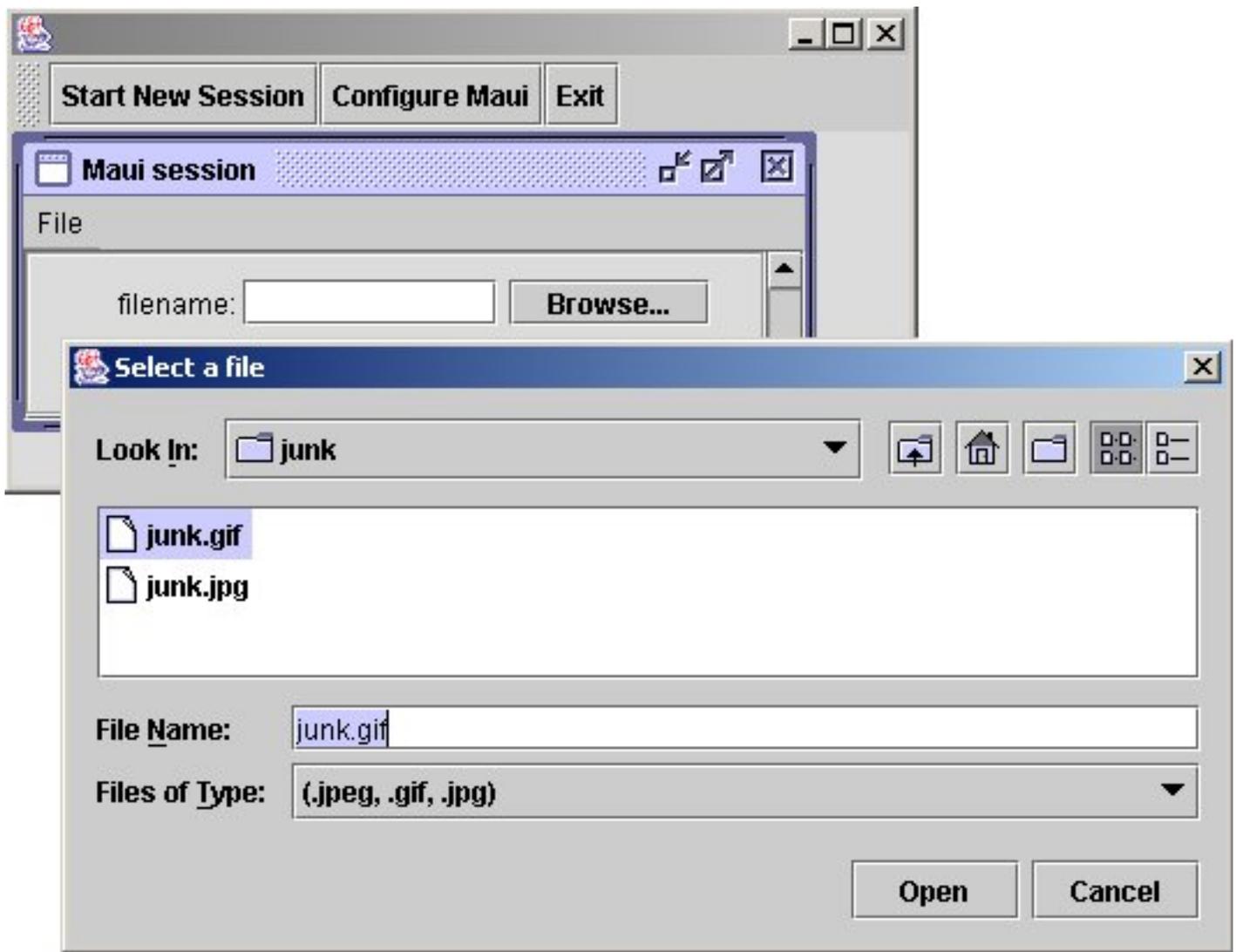


Figure 4.2: Pressing the browse button will pop up a file dialog box.

The XML used to specify a `CustomEditor` must contain a name attribute that points to the appropriate custom editor class. This name attribute may be qualified to specify a classpath to where the Java class for the `CustomEditor` can be found. See Section 5.4 for information on configuring paths in Maui custom classes. In Figure 4.1, `Maui.Editor.FilenameEditor` is specified as the class to be used. Any additional configuration information for the custom editor can be passed to the custom editor code as children elements of the `CustomEditor` XML element. The custom editor developer can decide what information should be passed to the custom editor, and how to format that in the XML for the custom editor. The `FilenameEditor.java` code has been written in such a way that `folderName`, `filename`, `openOrSaveFile`, `title`, `filenameExtensions`, and `onErrorRedisplayFileDialogBox` are child elements within the `CustomEditor` XML that might be passed on and parsed by `FilenameEditor.java`. In the `FilenameEditor.java` code, none of these elements is required to have been specified in the XML. All these elements have default values.

If the GUI developer writing a Maui XML application specification wishes to use the

FilenameEditor, the following describes what values are expected in the default attribute for each possible XML element:

folderName

If this element is defined, the default attribute should contain the name of the folder that is selected when the file dialog box is initially displayed on the screen. This setting can be overridden by inserting the name of a file (with path) into the default attribute of the String tag. For example,

```
<String name="TestFilename" label="filename"
        default="/home/somebody/anyFilename">
```

is the same as setting

```
<folderName default="/home/somebody"/>.
```

*If no **folderName** element is specified, then the default directory, dependant on your system, will be selected and displayed in the file dialog box.*

filename

If this element is defined, the default attribute should contain the filename that is selected when the file dialog box is initially displayed on the screen. This setting can be overridden by inserting the name of a file (fully qualified path or relative path) into the String tag. For example,

```
<String name="TestFilename" label="TestFilename"
        default="/home/somebody/anyFilename">
```

is the same as setting

```
<folderName default="/home/somebody"/>
<filename default="anyFilename"/>
```

or just

```
<filename default="/home/somebody/anyFilename"/>.
```

*If no **filename** element is specified, then the name of the default directory, dependant on your system, will be selected and displayed in the file name text box in the file dialog box.*

openOrSaveFile

If this element is defined, the `default` attribute for **openOrSaveFile** should be set to either `open` or `save`. If set to `save`, then the file dialog box will prompt the user to save a file. If set to `open`, then the file dialog box will prompt the user to select a file to open, given that that file already exists. *If an **openOrSaveFile** element is not defined for this custom editor, then the dialog will default to `open`.*

title

If this element is defined, the `default` attribute should be set to words that will appear in the file dialog box's titlebar. *If this element is not provided, then there will not be a title for the dialog box.*

filenameExtension

If this element is defined, the `default` attribute should be set to a space delimited list of filename extensions. **filenameExtension** is used to control which filenames are made visible in the file dialog box. For example, to see all filenames in the selected folder, set **filenameExtension** `default` attribute to `" "`. To see only gif and jpg files, set **filenameExtension** `default` attribute to `"gif jpg jpeg"`. *If the **filenameExtension** element is not specified, all files in the directory will be shown by default.*

onErrorRedisplayFileDialogBox

If this element is defined, the `default` attribute for **onErrorRedisplayFileDialogBox** should be set to either `yes` or `no`. If set to `yes`, and if the end-user makes an error entering a file name, then an error message will pop up on the screen; the error message will ask the end-user if he/she wants to try again. If the end-user says `yes` then the file dialog box is redisplayed on the screen. If set to `no`, and if the end-user makes an error entering a file name, then no error message will appear on the screen. *By default `yes` is chosen if the **onErrorRedisplayFileDialogBox** element is not specified.*

Maui provides the `FilenameEditor` as a custom editor for `String` data that represents a path to a file. In this section you have learned how to specify the XML for using the `FilenameEditor`.

You can write your own custom editor for any of the Maui data types. In the next section we address the three tiers of Maui classes used to represent data in Maui. With this information and the example in Section [4.3.3](#), you will be prepared to write your own custom editors.

[Next](#) [Up](#) [Previous](#)

Next: [4.3 Writing Your Own](#) **Up:** [4. Custom Editors](#) **Previous:** [4.1 Introduction](#)

[Next](#) [Up](#) [Previous](#)**Next:** [4.3.1 The Structure of](#) **Up:** [4. Custom Editors](#) **Previous:** [4.2 Using a Custom](#)

4.3 Writing Your Own Custom Editors

To write your own custom editor, you need to first define the parameters that you wish to pass to your custom editor through the XML. In Section [4.2](#), we saw that the writer of the `FilenameEditor` class chose `folderName`, `filename`, `openOrSaveFile`, `title`, `filenameExtensions`, and `onErrorRedisplayFileDialogBox` as parameters to be passed to the `FilenameEditor` code. You will be responsible for deciding what parameters make sense for your editor, and for defining these parameters in XML for your custom editor.

Once your XML specification is established, you can then create a custom editor class. In Section [4.3.1](#) we will talk about the data heirarchy used in Maui to represent data, and how to navigate this heirarchy to define your custom editor. A custom editor in Maui should be a class that derives from the `EditorBase` class at the **Editors** level of the data heirarchy.

In Section [4.3.2](#) we describe in general how to write a custom editor. This includes description of the methods that must be implemented in your custom editor as a result of your custom editor deriving from the `EditorBase` class.

Finally, in Section [4.3.3](#) we have provided an example Java class that defines a slider bar custom editor for an integer. Comments within the code describe the function of various methods in the class.

Subsections

- [4.3.1 The Structure of Maui Data](#)
 - [4.3.2 Steps to Writing Your Own Custom Editor](#)
 - [4.3.3 Example of a Maui Custom Editor](#)
 - [4.3.4 Compiling Your Custom Editor](#)
 - [4.3.5 Configuring Maui to Find Your Custom Editor](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [4.3.1 The Structure of](#) **Up:** [4. Custom Editors](#) **Previous:** [4.2 Using a Custom](#)

[Next](#) [Up](#) [Next](#)**Next:** [4.3.2 Steps to Writing](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3 Writing Your Own](#)

4.3.1 The Structure of Maui Data

Maui has a three-layer data structure with classes for each layer. We begin this section with a brief overview of each.

Variables:

This first layer holds the information describing each variable in your XML file. A Maui variable is described by a variable type, which is the tag of the XML element that represents the variable in the XML file. A variable is also uniquely described within a class by a name, which is extracted from the name attribute of the Maui XML instance element. Defined by optional XML attributes and elements, the variable also contains labeling information (in the `label` attribute), a flag specifying if data must be provided (in the `optional` attribute), tool tip information (in the `toolTip` attribute), custom editor or application data, the XML used to define this variable, and finally, the default value of the variable (in the `default` attribute of primitive variables). Depending on the type of the variable, other information may be required.

The class containing this minimum amount of information for all variable types is called `VarBase`. The `VarBase` class is defined within the `Maui.Variables` package^{4.1}. Within the `Maui.Variables` package, all of the other classes for holding variables derive from `VariableBase`. Each class derived from `VarBase` corresponds to a Maui variable type. For example, `PrimitiveVar` derives from `VarBase`, and `IntegerVar`, `DoubleVar`, and `BooleanVar` all derive from `PrimitiveVar`. Once a `VarBase` object has been instantiated, none of its data is ever changed. `VarBase` and the classes that derive from `VarBase` contain many methods for accessing the data stored at the creation of a `VarBase` object.

Instances:

Since `VarBase` objects are never changed after they have been initialized, the current value of each variable (based on user input to the GUI) must be stored somewhere other than `VarBase`. The object that stores the current data corresponding to a `VarBase` object is derived from the `InstanceBase` class in the `Maui.Instances` package. Besides holding the current value corresponding to a variable, the `InstanceBase` object acts a mediator between the variable object and the GUI editor object for the variable.

As with `VarBase` there are instance classes for each Maui variable type, and these instance classes correspond to the `VarBase` objects. For each instance of an object derived from

InstanceBase, there is a corresponding variable object. Additionally, for each instance object there is an corresponding editor object.

Editors:

The third layer of the Maui data structure is designed to display and allow editing of each instance of a variable. Thus, there will be an `EditorBase` object for *each* `InstanceBase` object. The editor is the GUI component that the end-user sees. Each Maui data type has a default editor to display it. For example, as noted above, all primitive types are edited by default using a `JTextField`, which is contained in the `DefaultPrimitiveEditor` class. However, if in the XML definition of a variable instance a `CustomEditor` is specified, Maui will instead try to load the code corresponding to the `CustomEditor`, and render the instance of this variable as the `CustomEditor` GUI component. In Section [4.3.2](#) we describe the steps to overriding the default `EditorBase` object with a custom editor.

The three-way split of variable, instance, and editor objects makes it easy, in principle, to replace the default `EditorBase` object for an variable type with a custom editor. So what does the `EditorBase` object need to do? In part the editor must provide a display and editing capability. Additionally, once data has been entered or changed in the displayed GUI component, the editor must then update its corresponding `InstanceBase` object with the new value that has been obtained. An editor must also be able to update its values from an `XMLObject`[4.2](#) so that the editor updates its display and value properly when saved data is read back in from a file[4.3](#). An editor object must also have methods to store the current data in the editor, and restore that state. Finally, an editor object should be able to update its "look and feel" based on changes to the appearance settings. In section [4.3.3](#), we give an example of a complete custom editor; we comment on the features of custom editors in Section [4.3.2](#).

[Next](#) [Up](#) [Next](#)

Next: [4.3.2 Steps to Writing](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3 Writing Your Own](#)

[Next](#) [Up](#) [Previous](#)**Next:** [4.3.3 Example of a](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3.1 The Structure of](#)

4.3.2 Steps to Writing Your Own Custom Editor

To write your own custom editor, you need to create a class that extends `EditorBase`. Additionally you need to implement the abstract methods of `EditorBase`. In the example in Section [4.3.3](#), we create a custom editor that uses a slider bar for representing an integer variable.

If you plan to create a custom editor, you should study the code in Section [4.3.3](#) carefully and use this code as a template to create your own editor. First, several points must be made:

1. Your custom editor class derives from `EditorBase`.
2. Any class derived from `EditorBase` will be a `JPanel`, since `EditorBase` derives from `JPanel`. Therefore, any calls to the `add` method will add a component, such as a `JSlider`, to the `EditorBase JPanel`.
3. Because `EditorBase` is abstract, you must implement its abstract methods in your derived class. The abstract methods in `EditorBase` are
 - `void setValue(Vector)`
 - `void setValue(Vector, boolean)`
 - `void updateValue(XMLObject, Vector, boolean)`
 - `void addElementsFromXML(XMLObject, Vector, boolean)`
 - `void storeValue()`
 - `void restoreSavedValue()`
 - `void refreshLookAndFeel()`

See the Javadoc documentation on each of these methods.

4. In addition to implementing the abstract methods, you will also want to write your own version of the method `init`. It is not an abstract method, but it is the place to put the code to implement your custom GUI component (i.e. the custom editor). This is the method called by the instance classes (derived from `InstanceBase`) when the variable has been found to have a custom editor. The example in the next section shows how to do this.

[Next](#) [Up](#) [Previous](#)**Next:** [4.3.3 Example of a](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3.1 The Structure of](#)

Next: [4.3.4 Compiling Your Custom](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3.2 Steps to Writing](#)

4.3.3 Example of a Maui Custom Editor

Now that we have laid the foundation for writing a custom editor, let's write one. In the following code example we have designed a custom editor class, `SliderEditor`, that graphically represents an integer with a slider bar. We followed the guidelines laid out in Section [4.3.2](#) to develop our class. Here are a few things to note before looking at the code:

1. All of the code to implement the slider (`JSlider`) is contained in the `init` method. Since `SliderEditor` extends `EditorBase`, and `EditorBase` extends `JPanel`, the calls to the `add` method at the end of `init` method are adding the label and slider components to this `JPanel`.
2. The `SliderEditor` class uses methods and public instance variables from `InstanceBase`, `PrimitiveInstance`, `VarBase`, `XMLObject`, `EditorBase` and `EditorSettings`. You can look at the Javadoc for the detailed API.
3. Many additional features of sliders could be added. Note that `upper`, `lower`, and `default` are attributes of the `Integer` XML element, and that we cannot add more attributes to an `Integer` to enable other features of `JSlider`. We can, however, add additional configuration information for the `SliderEditor` to the `CustomEditor` block in the XML. An example is given in Figure [4.3](#). You should look carefully at the detailed comments in the code. You should also pay particular attention to how the `CustomEditor` block in the XML is accessed and read in the `init` method.
4. Methods particular to the `SliderEditor` are described in the comments preceding each method.

To use this class, you must include the appropriate XML specification for the custom editor in your XML file. Figure [4.3](#) shows a simple example that uses the `SliderEditor`. Figure [4.3.3](#) shows how the `SliderEditor` is rendered in the GUI.

```
<Maui RootClass="sliderTest">
  <Class type="sliderTest" label="Slider Test">
    <Fields>
      <Integer name="x" label="Slider Integer"
        lower="1" upper="20" default="3">
        <CustomEditor name="SliderEditor">
          <Properties majorTickSpacing="5" orientation="vertical"/>
          <Slabel location="1" string="1"/>
          <Slabel location="3" string="Default"/>
          <Slabel location="20" string="20"/>
        </CustomEditor>
      </Integer>
    </Fields>
  </Class>
</Maui>
```

Figure 4.3:Maui input XML example of specifying a CustomEditorfor an Integer.

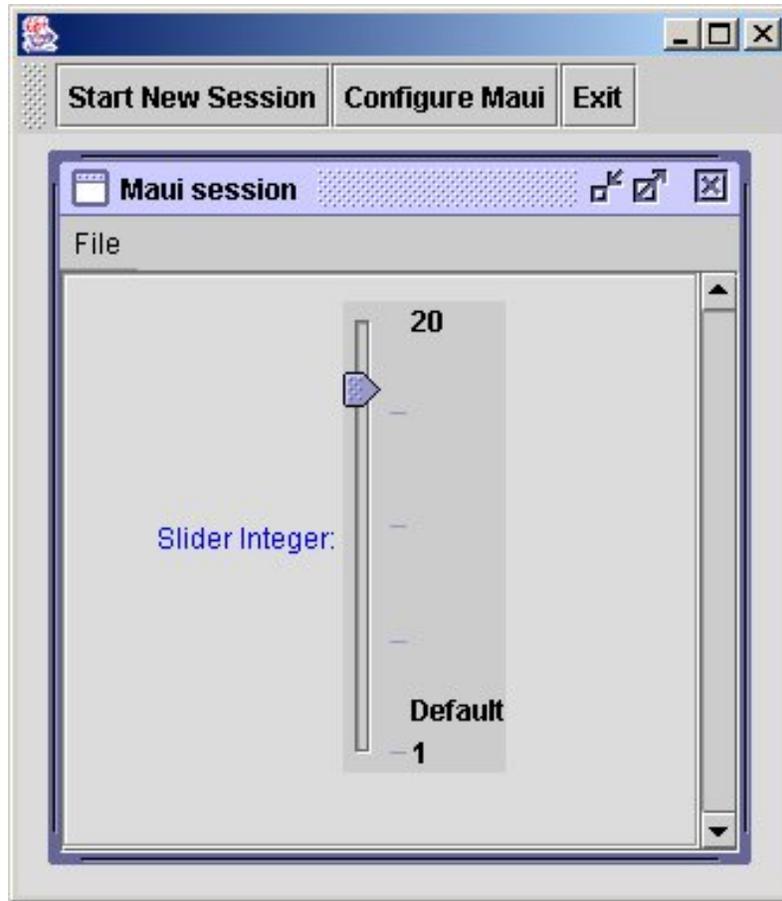


Figure 4.4:How Maui renders the slider specified in the XML.

In this example, we have specified the name of the `CustomEditor`, but we have not specified the path and package for `SliderEditor`. This can be done in the XML file, or you could add this information using the "Configure Maui" button, or you could do nothing and let Maui query you to locate the `SliderEditor` class file. See Section [5.4](#) where we describe how Maui can locate your custom editor class.

At this point you need to know exactly what happens to the `CustomEditor` block of the XML. Since we can't know in general what information will be contained in such a block, Maui simply passes all of it along to the `VarBase` object where it is stored intact. The only check on the block is to assure that it is valid XML; if not, it will be flagged at the time the entire Maui input XML file is read. The `CustomEditor` block can be retrieved by the new custom editor. Look carefully at the `init` method in the example to see how this is done.

Following is the complete code for the `SliderEditor` class. A copy of this file can be found in `$MAUI_HOME/Doc/tutorials/maui/Java/SliderEditor.java`.

```

/**
 This class is a custom editor for an integer.  It allows an integer
 variable to be adjusted with a slider bar rather than by typing a
 value in a text field.  Note that the XML to use this is a CustomEditor
 block that has a required name="SliderEditor" attribute and optional
 path and package attributes to designate the location of this class.
 The CustomEditor XML block for the SliderEditor is also assumed to have a
 child named Properties that can have attributes of "majorTickSpacing"
 and "orientation".  This block can be expanded with other attributes
 corresponding to settings in JSlider.  The CustomEditor block can also have
 as many children as you like named Slabel.  Each of these children has
 attributes corresponding to the location on the sliding scale and a
 string to display at that location.  See SliderEditor.xml in
 $MAUI_HOME/Doc/tutorials/maui/XML for the CustomEditor input for this
 editor.
 */

import XML.*;           // Access to XMLObject class
import Maui.Instances.*; // Access to InstanceBase. IntegerInstance classes
import Maui.Editors.*;  // Access to EditorBase, EditorSettings classes
import Maui.Interface.*; // Access to MauiMainPane
import java.util.*;     // Access to Vector, Hashtable classes
import java.awt.*;     // Access to Color class
import javax.swing.*;   // Access to JSlider, JLabel, BorderLayout,
                       // BorderLayout, Box classes

public class SliderEditor extends EditorBase
{
 //=====
 //SliderEditor data members:

 /* The JSlider Swing component. */
 JSlider slider_;

 /* The JLabel component to hold the label on the slider. */
 JLabel label_;

 /* The minimum value for the slider variable. */
 int min_;

 /* The maximum value for the slider variable. */
 int max_;

 /* An int to hold the saved value for use in restoring. */
 int savedValue_;

 /* A boolean to indicate if a value has been saved. */
 boolean isSaved_;

```

```

/* A String to hold the label corresponding to savedValue_. */
String savedLabel_;

/* The IntegerInstance corresponding to the integer variable. */
IntegerInstance intInstance_;

//=====
// SliderEditor methods:

//-----
/**
 * A null constructor that is necessary to allow this class to be
 * easily loaded and instantiated.
 */
public SliderEditor() {}

//-----
/**
 * This method only matters if we are adding to an Array or a Table.
 * See documentation for addElementsFromXML in EditorBase.
 */
public void addElementsFromXML(XMLObject xml, Vector failureList,
                               boolean checkValue)
{
    /* nothing to do, since this
     * is not an editor for an Array or Table */ }

//-----
/**
 * Gets the necessary values for creating the JSlider, instantiates
 * the slider, and adds the slider to the current JPanel (i.e. "this").
 *
 * @param mainPane the MauiMainPane holding this session.
 * @param instance the IntegerInstance object for the integer
 * to be adjusted.
 */
public void init(MauiMainPane mainPane, InstanceBase instance)
{
    super.init(mainPane, instance);
    intInstance_ = (IntegerInstance) instance;
    isSaved_ = false;

    /*
     * Get some values stored in the VarBase object. The only values
     * that can be here are the max, min and default, since these are
     * XML attributes associated with an Integer variable. Note:
     * we assume that these attributes have been specified in the
     * definition of the Integer. This code could be easily

```

```

    modified to check values and specify defaults.
*/
max_ = instance.myVar_.getDefiningXML().getInt("upper");
min_ = instance.myVar_.getDefiningXML().getInt("lower");
int value = instance.myVar_.getDefiningXML().getInt("default");

/* Instantiate the slider.    */
slider_ = new JSlider(min_, max_, value);

/*
   Get the XMLObject for the CustomEditor settings.  The XML is
   stored in the VarBase object that corresponds to this editor
   (and corresponds to the IntegerBase object).
*/
XMLObject customEdSettings = instance.myVar_.getCustomEditorSettings();

/*
   Access the attributes and children of customEdSettings to
   get the set up information for the slider.  Again, we
   assume that these attributes have been set and that there
   is at least one label; the code could be easily extended to
   make default decisions or do error checking.
*/

/*
   First, get the orientation and major tick spacing from the
   Properties XML.

   Example XML:

   <Properties majorTickSpacing="5" orientation="vertical"/>

   Use this information to set up orientation and tick spacing in the
   slider.
*/
String orient = customEdSettings.getChild("Properties").
    getAttribute("orientation");
if (orient.equals("vertical"))
    slider_.setOrientation(JSlider.VERTICAL);

int majorTickSpacing = customEdSettings.getChild("Properties").
    getInt("majorTickSpacing");
slider_.setMajorTickSpacing(majorTickSpacing);
slider_.setPaintTicks(true);

/* Get any slider label information.  From any Slabel XML elements
   in the CustomEditor XML:

```

Example XML:

```
<Slabel location="1" string="1"/>
  (etc.)
<Slabel location="20" string="20"/>
```

Use this information to set up labels along the slider.

```
*/
Hashtable labelTable = new Hashtable();
int numKids = customEdSettings.numChildren();
for (int i = 0; i < numKids; i++)
{
  XMLObject kid = customEdSettings.getChild(i);
  if (kid.getTag().equals("Slabel"))
  {
    int location = kid.getInt("location");
    String lab = kid.getAttribute("string");
    labelTable.put(new Integer(location), new JLabel(lab));
  }
}

slider_.setLabelTable( labelTable );
slider_.setPaintLabels(true);

/*
  Set up the other information to display the slider. This is
  for the JPanel in which the slider resides. Note that EditorBase,
  which SliderEditor extends, is an extension of JPanel.
*/
setLayout(new BorderLayout(this, BorderLayout.X_AXIS));
setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));

/*
  Set the background color to be consistent with the color set
  in "Configure Maui" appearance settings.
*/
//setBackground(EditorSettings.getBackgroundColor());
setBackground(this.getEditorSettings().getBackgroundColor());

/*
  Get the label and the tool tip for the Integer and add
  them to this JPanel. Set the font and background color.
*/
String name = intInstance_.getLabel();
//String tip = intInstance_.getTip();
String tip = intInstance_.getAttribute("tooltip");
label_ = new JLabel(" " + name + ": ");
label_.setToolTipText(tip);
```

```

label_.setBackground(this.getEditorSettings().getBackgroundColor());
label_.setFont(this.getEditorSettings().getFont());

/* If this is a required parameter, label it in blue. */
//if(intInstance_.myVar_.optional_.compareTo("false")==0)
if (intInstance_.myVar_.getAttribute("optional").equals("false"))
    label_.setForeground(Color.blue);

/* Call the setEditable method in EditorBase to allow the user to edit
   this object (i.e. change the integer value by using the slider bar
   */
//setEditable(intInstance_.isEditable());
setEditable(intInstance_.getAttribute("editable").equals("true"));

/* Add the label and slider to this JPanel (i.e. SliderEditor/EditorBase). */
add(label_);
add(slider_);
add(Box.createHorizontalGlue());
}

//-----
/**
    Sets the value given by the slider. This version simply calls
    the the two parameter setValue method; the second parameter is
    set to true, which means that setValue will check for errors when
    the value is set.

    @param failureList a Vector holding any errors created.
 */
public void setValue(Vector failureList)
{
    setValue(failureList, true);
}

//-----
/**
    Sets the value given by the slider. Since there is no way to
    enter an illegal value, the boolean argument checkForErrors has
    no effect.

    @param failureList a Vector holding any errors created.
    @param checkForErrors a flag to signify if errors should be reported
    using the failureList, or not.
 */
public void setValue(Vector failureList, boolean checkForErrors)
{
    int value = slider_.getValue();
    /* No error can occur since max and min are set by the slider. So

```

```

    we call the setValue method for the instance with the value of
    the second parameter being false. */
intInstance_.setValue(new Integer(value).toString(), false);
}

//-----
/**
    Updates the value of the slider from the supplied XMLObject.
    Here, the value could, in theory, be invalid, so we check. If
    it is invalid, we set it to the default value. Other remedies
    could be considered.

    @param xml the XMLObject containing the value for the slider.
    @param failureList a Vector to contain all of the errors.
    @param checkValue a boolean to indicate if a legitimate value
    is provided.
    */
public void updateValue(XMLObject xml, Vector failureList,
                        boolean checkValue)
{
    /* Get the value from the XMLObject. Use the default value if
    none is provided. */
    String sVal = xml.getWithDefault("default",
                                     //intInstance_.getDefaultvalue();
                                     intInstance_.getAttribute("default"));
    int val = new Integer(sVal).intValue();

    if (checkValue)
    {
        if (val < min_ || val > max_)
        {
            //val = new Integer(intInstance_.getDefaultvalue()).intValue();
            val = new Integer(intInstance_.getAttribute("default")).intValue();
        }
    }
    slider_.setValue(val);
    setValue(failureList, checkValue);
}

//-----
/**
    Stores the value so that it can later be restored by the
    restoreValue method.
    */
public void storeValue()
{
    savedValue_ = slider_.getValue();
    savedLabel_ = label_.getText();
}

```

```

    isSaved_ = true;
}

//-----
/**
    Restores a stored value from storeValue.
 */
public void restoreSavedValue()
{
    if (!isSaved_) return;
    slider_.setValue(savedValue_);
    label_.setText(savedLabel_);
    setValue(new Vector(), false);
}

//-----
/**
    Updates the look and feel of the panel based on new values in
    EditorSettings.
 */
public void refreshLookAndFeel()
{
    /* set font */
    label_.setFont(getEditorSettings().getFont());

    /* set background */
    setBackground(getEditorSettings().getBackgroundColor());
    label_.setBackground(getEditorSettings().getBackgroundColor());

    /* refresh the panel */
    label_.invalidate();
    slider_.invalidate();
    validate();
}
}

```

Figure 4.4.1 SliderEditor.java

There are many other details that could be addressed concerning the writing of a custom editor. Instead of devoting more text to these details, we suggest that you browse the various editor classes contained in `$MAUI_HOME/Java/src/Maui/Editors`. You will quickly see that many of the methods that must be overridden from `EditorBase` have very similar implementations from `Editor` class to `Editor` class.

[Next](#) [Up](#) [Previous](#)**Next:** [4.3.5 Configuring Maui to Up](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3.3 Example of a](#)

4.3.4 Compiling Your Custom Editor

To compile your custom editor, first make sure that you are in the directory where `SliderEditor.java` is located (`$MAUI_HOME/Doc/tutorials/maui/Java`). Now type the command

UNIX:

```
javac -classpath .:$MAUI_HOME/Java/classes/maui.jar:\
$MAUI_HOME/Java/classes/xerces.jar SliderEditor.java
```

WINDOWS:

```
javac -classpath .;"%MAUI_HOME%\Java\classes\maui.jar";
"%MAUI_HOME%\Java\classes\xerces.jar"
"%MAUI_HOME%\Doc\tutorials\maui\Java\SliderEditor.java"
```

(NOTE: Because the command is too long to fit on one line, it has been separated it into multiple lines to make the command easier to read. However, for Windows to execute this command, the entire command must be typed on one line.)

[Next](#) [Up](#) [Previous](#)**Next:** [4.3.5 Configuring Maui to Up](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3.3 Example of a](#)

[Next](#) [Up](#) [Previous](#)**Next:** [4.4 Summary](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3.4 Compiling Your Custom](#)

4.3.5 Configuring Maui to Find Your Custom Editor

As mentioned previously, you may specify the path to your custom editor within the XML name attribute of the `CustomEditor` XML element, you may use the "Configure Maui" button to set up the path to the custom editor, or you may let Maui try to figure out the path. Details of configuring Maui to find the compiled custom editor class file can be found in Chapter [5](#), Section [5.4](#).

[Next](#) [Up](#) [Previous](#)**Next:** [4.4 Summary](#) **Up:** [4.3 Writing Your Own](#) **Previous:** [4.3.4 Compiling Your Custom](#)

[Next](#) [Up](#) [Previous](#)**Next:** [5. Configuring Maui](#) **Up:** [4. Custom Editors](#) **Previous:** [4.3.5 Configuring Maui to](#)

4.4 Summary

In this chapter you have learned the fundamentals of writing a custom editor for any of the Maui data types. Custom editors are an advanced feature in Maui, so we encourage you to take time to study the editor classes built into Maui (in `$MAUI_HOME/Java/src/Maui/Editors`) to see how other editors are implemented.

[Next](#) [Up](#) [Previous](#)**Next:** [5. Configuring Maui](#) **Up:** [4. Custom Editors](#) **Previous:** [4.3.5 Configuring Maui to](#)

[Next](#) [Up](#) [Previous](#)

Next: [5.1 Introduction](#) **Up:** [A Maui User's Guide](#) **Previous:** [4.4 Summary](#)

5. Configuring Maui

Subsections

- [5.1 Introduction](#)
 - [5.2 Appearance Settings](#)
 - [5.3 Services](#)
 - [5.4 Paths and Packages](#)
 - [5.4.1 Configuring with the Configure Maui Button](#)
 - [5.4.2 The "Do Nothing" Configuration Method](#)
 - [5.4.3 Configuring in Your XML Specification](#)
-

[Next](#) [Up](#) [Previous](#)**Next:** [5.2 Appearance Settings](#) **Up:** [5. Configuring Maui](#) **Previous:** [5. Configuring Maui](#)

5.1 Introduction

Maui may be dynamically configured in several ways. For example, font size and background color can be changed and these choices can be made permanent. In addition, Maui can be told where to find custom actions and editors, or what input files to use. These Maui configuration actions are all described in this chapter. All of these actions are started by clicking on the "Configure Maui" button in the main tool bar. This brings up a window as in [Figure 5.1](#).

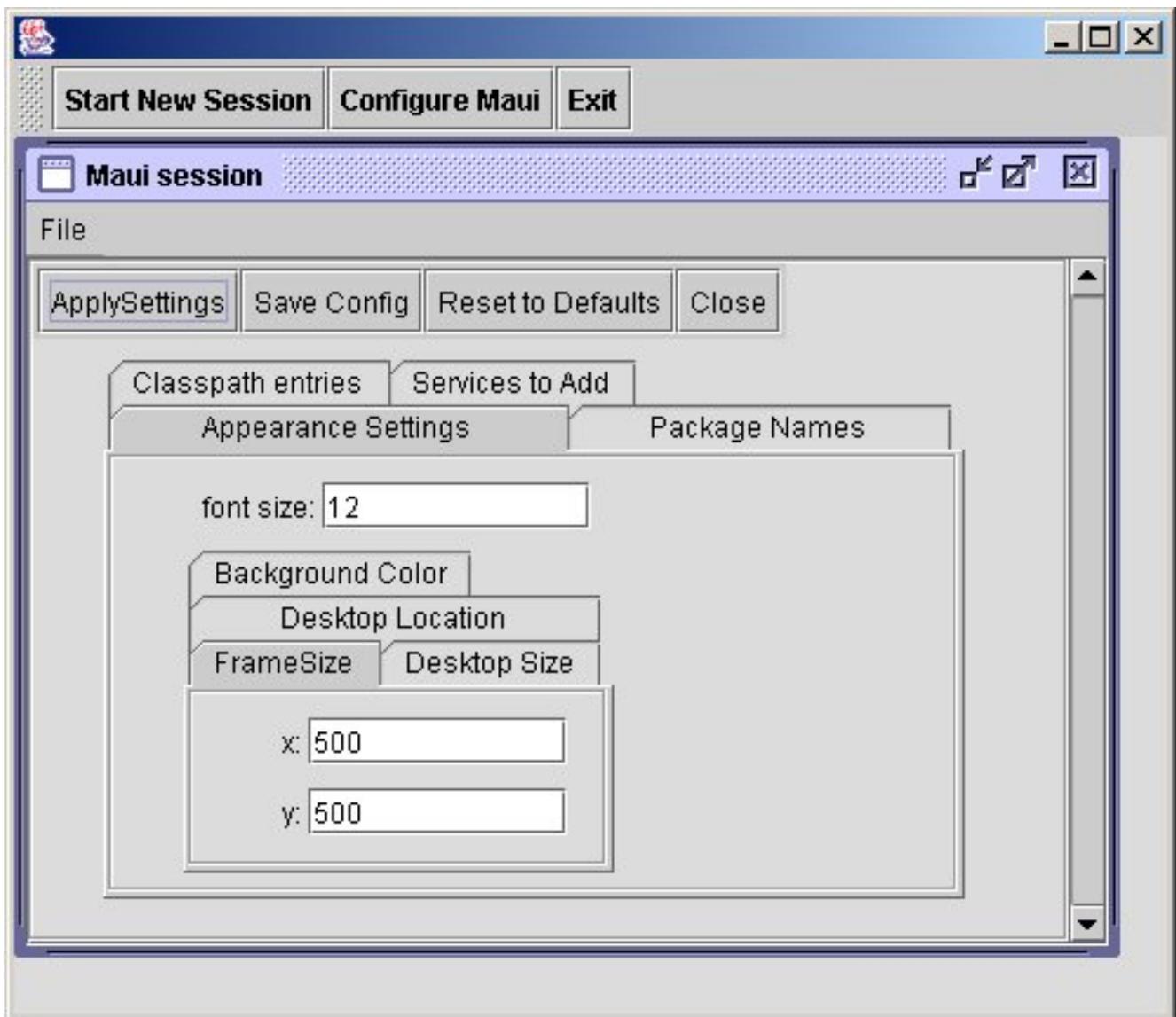


Figure 5.1: The "Configure Maui" Panel

[Next](#) [Up](#) [Previous](#)

Next: [5.2 Appearance Settings](#) **Up:** [5. Configuring Maui](#) **Previous:** [5. Configuring Maui](#)

[Next](#) [Up](#) [Previous](#)**Next:** [5.3 Services](#) **Up:** [5. Configuring Maui](#) **Previous:** [5.1 Introduction](#)

5.2 Appearance Settings

Within the "Configure Maui" GUI window, clicking on the "Appearance Settings" tab shows you where you can change the desktop size, the desktop location, the frame size, the font size, and the background color. The desktop size is the initial size of the original screen that comes up when you first start Maui. The desktop location is determined by (x,y) coordinates that signify the location of the top left corner of the desktop; (0,0) locates the desktop in the top left corner of the screen. The frame size is the original size of the screen that comes up when you click on "Start New Session." These sizes and location coordinates are in pixels.

Change any of these settings to suit your preferences and then click on "Apply Settings" to see how your settings will affect the look of Maui. When you are happy, click on "Save Config" to make these changes permanent. If you want to start over, you can always click on "Reset to Defaults" and the original default settings will be restored.

An important question arises: Where are these settings stored? Any changes made to the "Configure Maui" panel are saved to a file named `config-local.xml` that is in a directory that depends on your operating system. If you are running any version of Unix, this file is stored in the `.mauiConfig` directory in your home directory. For Windows, and any other operating system, the file is stored in the folder `mauiConfig` in the directory that Java returns from the statement

```
String dir = System.getProperty("user.home");
```

Recall that this folder (directory) was created when Maui was installed. The advantage of doing things this way is that if you ever update your version of Maui, your personal settings will remain valid. *Do not modify the files in the `.mauiConfig/mauiConfig` directory by hand. The files are produced (and overwritten) automatically by Maui.*

We plan to provide more appearance settings in the future, but let us know if you have some particular need now.

[Next](#) [Up](#) [Previous](#)**Next:** [5.3 Services](#) **Up:** [5. Configuring Maui](#) **Previous:** [5.1 Introduction](#)

Next: [5.4 Paths and Packages](#) **Up:** [5. Configuring Maui](#) **Previous:** [5.2 Appearance Settings](#)

5.3 Services

In Section [2.2](#) we mentioned the `RootClass` attribute belonging to the `Maui` element that begins each XML file. The `RootClass` specification tells Maui which class to display in the main panel of the Maui GUI. Maui can load multiple Maui XML files that each have their own `RootClass` specified. Maui will generate a subclass selection menu from which a user can select the service of interest.

Let us assume the following XML specifies your service (that is, interface to your application):

```
<Maui RootClass="Parameters" >

  <Class type="Parameters" label="Set the Parameters">
    <Fields>
      <Int name="maxit" label="Maximum number of iterations"
        default="50" />
      <Double name="tol" label="Convergence Tolerance"
        default="1.0e-06" />
    </Fields>
  </Class>

</Maui>
```

Suppose that the above Maui XML was saved in the file `/home/fred/myApp/params.xml`. By clicking on the "Services to Add" tab, you get the screen similar to the one in Figure [5.2](#). Click on "add" to get the screen shown in Figure [5.3](#). In the text area labeled `service` you can type in `/home/fred/myApp/params.xml`. Alternatively you can click on the "browse" button and navigate to the `params.xml` file with the file browser. The `label` box is there for convenience -- it allows you to associate a nickname for the service rather than use the path name. So, for example, you could enter `params` in this box. Be sure that the "To be included" box is checked, and click "ok". The "To be included" box is here to allow you to pick and choose what services you want to come up when you start a session. In particular, you can stop a service from coming up by unchecking its "To be included" box. You do not have to remove the service from the array and then later retype the service information if you decide you want the service reinstated in the GUI.

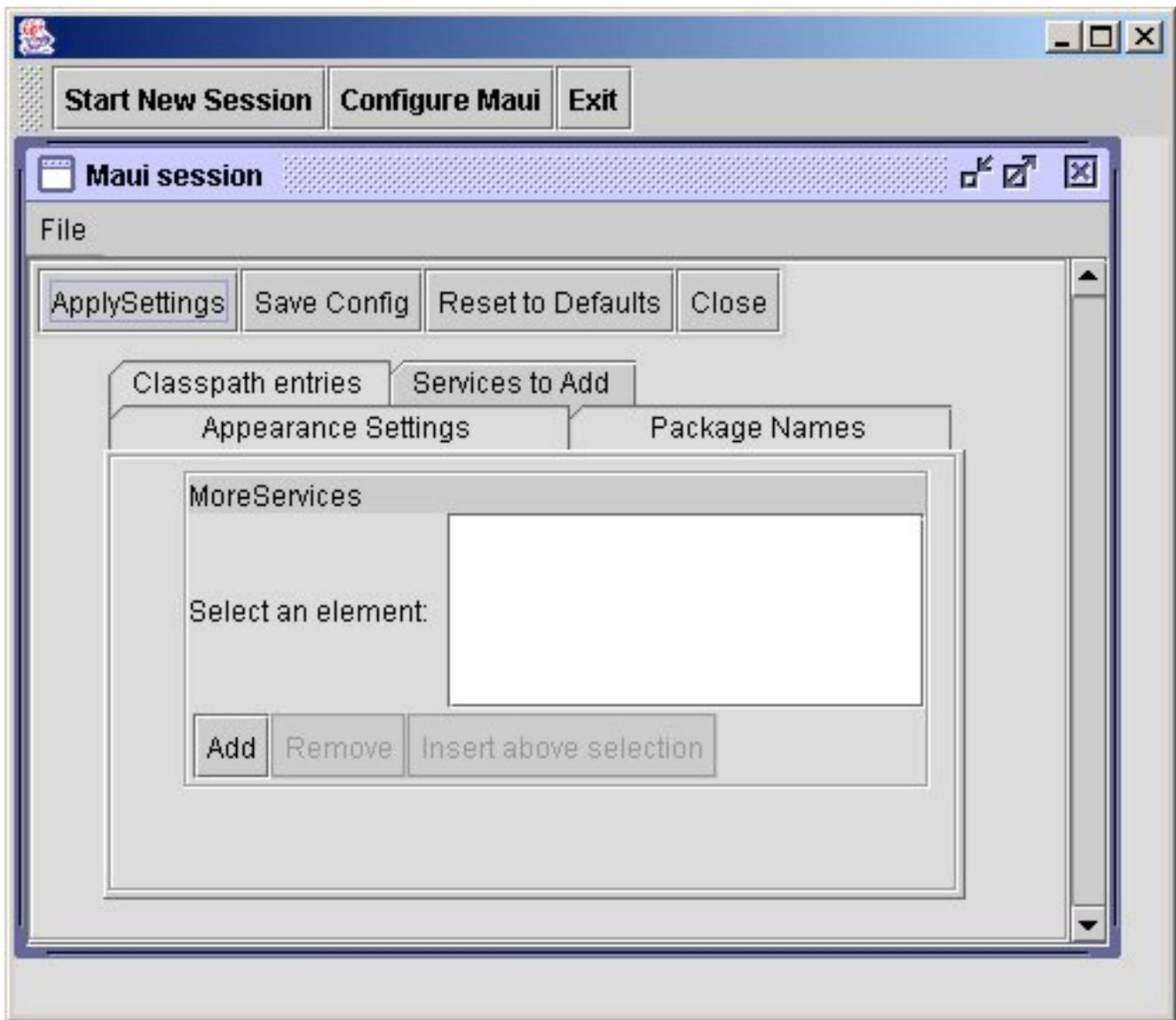


Figure 5.2: The "Services to Add" Panel.

Important: click on "Save Config" to have this information written to your `.mauiConfig` or `mauiConfig` directory.

[Next](#) [Up](#) [Previous](#)

Next: [5.4 Paths and Packages](#) **Up:** [5. Configuring Maui](#) **Previous:** [5.2 Appearance Settings](#)

[Next](#) [Up](#) [Previous](#)**Next:** [5.4.1 Configuring with the Up](#) **Up:** [5. Configuring Maui](#) **Previous:** [5.3 Services](#)

5.4 Paths and Packages

As mentioned in Chapter [3](#) on Maui Actions, and Chapter [4](#) on Maui CustomEditors, path and package information must be provided to Maui for locating custom classes. The following sections address the three different ways that one can specify path and package information for custom actions and custom editors.

Subsections

- [5.4.1 Configuring with the Configure Maui Button](#)
 - [5.4.2 The ``Do Nothing" Configuration Method](#)
 - [5.4.3 Configuring in Your XML Specification](#)
-

[Next](#) [Up](#) [Previous](#)**Next:** [5.4.2 The ``Do Nothing``](#) **Up:** [5.4 Paths and Packages](#) **Previous:** [5.4 Paths and Packages](#)

5.4.1 Configuring with the Configure Maui Button

Using the ``Configure Maui`` button in the Maui desktop is sometimes the convenient way to set `classpath` and `package` information for your custom classes. Clicking on ``Package Names`` and ``Classpath entries`` brings up an array to which you can add your `package` and `classpath` information, respectively. This information is used to load custom action and editor classes that are not built into Maui. Again, be sure to click on ``Save Config`` to ensure that this information is available for Maui to use.

Getting the `path` and `package` information correct can sometimes be confusing, so if you get it wrong, Maui will fail to load your custom action or custom editor. However, Maui will then give you a chance to point to the appropriate class using a file browser. When the path to your class is correctly specified, Maui will automatically determine the correct path and package information and write this into your `config-local.xml`.

[Next](#) [Up](#) [Previous](#)**Next:** [5.4.2 The ``Do Nothing``](#) **Up:** [5.4 Paths and Packages](#) **Previous:** [5.4 Paths and Packages](#)

[Next](#) [Up](#) [Previous](#)**Next:** [5.4.3 Configuring in Your](#) **Up:** [5.4 Paths and Packages](#) **Previous:** [5.4.1 Configuring with the](#)

5.4.2 The ``Do Nothing" Configuration Method

Using the ``Configure Maui" button is not the only way to tell Maui where your custom classes are located. Probably the easiest way to configure the necessary paths is to just wait to be prompted by Maui that it needs path information. So don't do anything to specify the path and package for a class. When you start a Maui session, Maui will at first fail to find your `Action` or `CustomEditor`, but you will be given the opportunity to search for it. If you choose to search for the class, a file browser will be displayed and you can browse for your class file. When you find the class file and select it, Maui will try to load it. Of course, if you made some mistakes in writing your action or custom editor, e.g., didn't make it a public class or forgot to extend `MauiAction` or `EditorBase`, Maui will try to tell you what the error is. In any case, the loading will fail and the session will be ended. If Maui succeeds in loading the class, Maui will modify your `config-local.xml`, so you will never have to search for this class again.

[Next](#) [Up](#) [Previous](#)**Next:** [5.4.3 Configuring in Your](#) **Up:** [5.4 Paths and Packages](#) **Previous:** [5.4.1 Configuring with the](#)

[Next](#) [Up](#) [Previous](#)

Next: [A. Application Example](#) **Up:** [5.4 Paths and Packages](#) **Previous:** [5.4.2 The ``Do Nothing''](#)

5.4.3 Configuring in Your XML Specification

The least optimal way to specify the `classpath` and `package` information is in the XML input file. In the case where your application is going to be run on multiple platforms, you cannot guarantee that the installation location of your custom classes will be the same on each platform. So you would have to modify the XML specification of the path and package for a custom class on every platform. In the case where your application may be installed multiple places, you are better off to use the ``Configure Maui'' button, or the ``do nothing'' method to configure your paths.

If you still wish to enter the `path` and `package` attributes in your XML, you may. To specify the `path` and `package` information for the `TestAction` class we wrote in Chapter 3, we could enter

```
<Action class="TestAction"
  path="path to directory TestAction.class lives" />
```

or, if `TestAction` is in a package,

```
<Action class="TestAction"
  path="path to package in which TestAction.class lives"
  package="packageName" />
```

where *path to package in which TestAction.class lives* is the directory--not including the package directory--above where `TestAction.class` is located, and *packageName* is the name of the package--the directory name that corresponds with the package. For example, assume `TestAction.class` is in a package called `Test`, and lives in the directory `/home/fred/Projects/Test/`. Then the `path` attribute above would be set to `/home/fred/Projects`, while the `package` attribute would be set to `Test`.

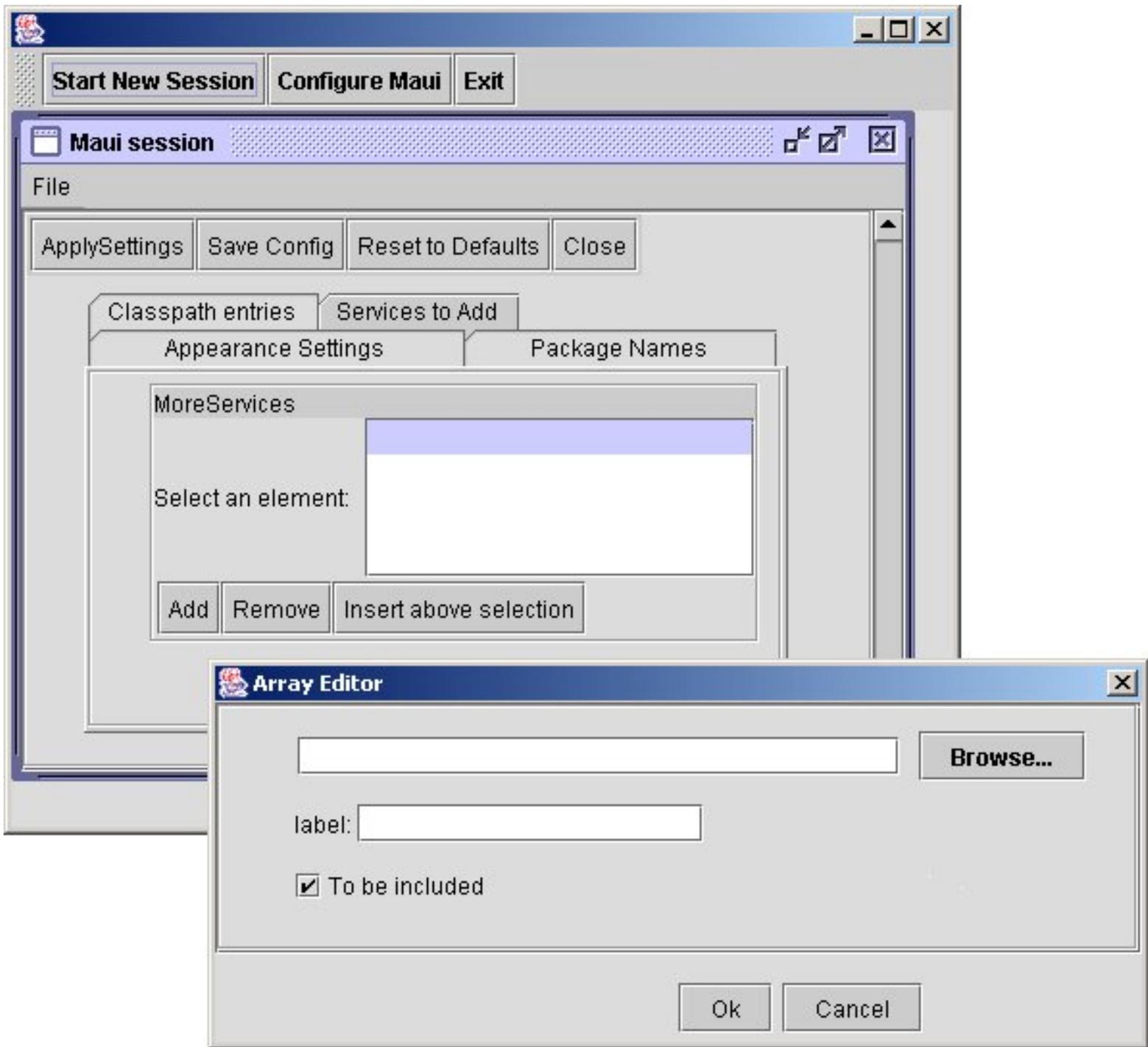


Figure 5.3: The "Adding a Service" Panel.

[Next](#) [Up](#) [Previous](#)

Next: [A. Application Example](#) **Up:** [5.4 Paths and Packages](#) **Previous:** [5.4.2 The "Do Nothing"](#)

[Next](#) [Up](#) [Previous](#)

Next: [A.1 A Calore GUI](#) **Up:** [A Maui User's Guide](#) **Previous:** [5.4.3 Configuring in Your](#)

A. Application Example

Subsections

- [A.1 A Calore GUI](#)
 - [A.1.1 Differences Between Text and GUI Calore Input](#)
 - [A.1.2 Calore GUI Design Examples](#)
 - [A.1.2.1 Array Example](#)
 - [A.1.2.2 Optional Fields Example](#)
 - [A.1.2.3 Subclassing Example](#)
 - [A.1.2.4 Referencing Example](#)
 - [A.1.2.5 Import Example](#)
 - [A.1.2.6 String Menu Example](#)
- [A.2 Text Input to Calore](#)

[Next](#) [Up](#) [Previous](#)

Next: [A.1 A Calore GUI](#) **Up:** [A Maui User's Guide](#) **Previous:** [5.4.3 Configuring in Your](#)

[Next](#) [Up](#) [Previous](#)**Next:** [A.1.1 Differences Between Text](#) **Up:** [A. Application Example](#) **Previous:** [A. Application Example](#)

A.1 A Calore GUI

Several existing code projects at Sandia are using Maui for a graphical user interface to their codes. Codes currently using Maui have been written in C, C++, and Fortran 77. For each project using Maui, the graphical user interface to the code makes it easier to enter inputs to the code in an organized manner. In this section we discuss the Maui interface to a code called Calore to provide examples of how to write Maui XML for a particular problem.

Calore is a code developed at Sandia National Laboratories to carry out thermal analysis. Calore's principal capabilities are linear and non-linear heat conduction, enclosure radiation, and high explosive and foam decomposition chemistry (<http://www-irn.sandia.gov/organization/sbu/sbu-nw/nwsbu-home/capabilities/engineering.htm>).

Calore is one code among many that fits into a finite element code framework called Sierra. Through the Sierra framework, an analyst may run many types of finite element simulations. Because of the common look-and-feel that Maui offers a user, it is an ideal tool for generating GUIs for all the Sierra codes.

Before the Maui GUI was introduced for the Calore project, the Calore user entered input information for Calore as ASCII text in an input file. Data within the file was broken up into "blocks," that can easily be likened to data objects in the object oriented programming paradigm. Because Maui is also object oriented, translation of the ASCII text input required by Calore into the XML GUI description required by Maui was straightforward.

A "block" of data in the ASCII Calore input file is delimited by a `begin` and an `end` statement. In many cases, multiple blocks of a similar type might be entered by the Calore user, such as function specifications, boundary conditions, and property specifications for various materials. To distinguish between blocks that may appear more than one in the input file, the Calore user assigns a unique block name to each block. In Section [A.2](#), you can see the variety of block types and block names that may be used in a Calore ASCII text input file.

Subsections

- [A.1.1 Differences Between Text and GUI Calore Input](#)

- [A.1.2 Calore GUI Design Examples](#)
 - [A.1.2.1 Array Example](#)
 - [A.1.2.2 Optional Fields Example](#)
 - [A.1.2.3 Subclassing Example](#)
 - [A.1.2.4 Referencing Example](#)
 - [A.1.2.5 Import Example](#)
 - [A.1.2.6 String Menu Example](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [A.1.1 Differences Between Text](#) **Up:** [A. Application Example](#) **Previous:** [A. Application Example](#)

[Next](#) [Up](#) [Previous](#)**Next:** [A.1.2 Calore GUI Design](#) **Up:** [A.1 A Calore GUI](#) **Previous:** [A.1 A Calore GUI](#)

A.1.1 Differences Between Text and GUI Calore Input

When writing a Calore text input file, the user does not have to enter data blocks in any particular order. However, nested data (blocks within blocks) must be nested appropriately. For example, a user may specify material properties before functions, even if one of the material properties references a function defined later in the file. For the Maui GUI, there are some restrictions to the order in which a user must input information. In the materials/functions example, the user cannot specify a function to define a material property unless the function has already been defined.

The Calore input vocabulary is quite extensive, and the user has many ways to enter data to set up a problem. With the GUI, the user is able to see all possible options without requiring a supplementary user's manual. Additionally, color coding of the entry box labels tells the user if a field is required to contain data (blue label), or if it is optional (black label). Mutually exclusive options, such as selecting one solver out of all available solvers, is represented in the GUI by a selection button. If the user selects one particular solver, then options for that solver only appear on the screen, while options only pertinent to other solvers are hidden from the user's view.

In the Calore text input file, if a user wishes to refer to something like a material, the user is required to type in the name of the material in the block of input requiring the reference. Lines 23-27 the Calore input listing in Section [A.2](#) define the material `Copper`. In line 71 of the Calore input example, `Copper` is referenced within a finite element block.

In the GUI, references may be set up so that as a user adds materials in the interface, any input that references a material gets an updated drop-down list of material names from which the user can select.

[Next](#) [Up](#) [Previous](#)**Next:** [A.1.2 Calore GUI Design](#) **Up:** [A.1 A Calore GUI](#) **Previous:** [A.1 A Calore GUI](#)

[Next](#)
[Up](#)
[Previous](#)

Next: [A.2 Text Input to Up:](#) [A.1 A Calore GUI](#) **Previous:** [A.1.1 Differences Between Text](#)

Subsections

- [A.1.2.1 Array Example](#)
- [A.1.2.2 Optional Fields Example](#)
- [A.1.2.3 Subclassing Example](#)
- [A.1.2.4 Referencing Example](#)
- [A.1.2.5 Import Example](#)
- [A.1.2.6 String Menu Example](#)

A.1.2 Calore GUI Design Examples

In this section we will use the XML specification of the Calore/Maui GUI to illustrate many Maui capabilities. Through the examples in this section you will be given motivation for when to use Maui objects and capabilities such as arrays, optional field settings, subclassing, references, importing XML files, and string menus.

A.1.2.1 Array Example

In the Calore text input file, there are many places where a Calore user might define several blocks of the same type, such as functions, solvers, and Calore regions. In the Maui GUI, blocks that might appear zero or more times can be represented as elements in an array. In Figure [A.1](#), we show the XML for representing a function definition object, FuncDef, as the template for an array element in a Maui Array. For the XML that specifies the classes derived from FuncDef, see \$MAUI_HOME/Doc/tutorials/maui/XML/calore.xml. This XML is translated into the GUI object in Figure [A.2](#).

```

<Class type="FuncArrClass" label="Define Functions">
  <Fields>
    <Array name="funcArr" label="Define Functions">
      <Master label="$funcName">
        <FuncDef name="aFuncDef" label="Blah"/>
      </Master>
    </Array>
  </Fields>
</Class>

<Class type="FuncDef" label="Define a funciton">
  <Fields>
    <String name="funcName" label="Function Name" />
    <FunctionType name="funcType" label="Type of Function"/>
  </Fields>
</Class>

```

```
<Class type="FunctionType" label="Type of Function (default: Piecewise Linear)">
</Class>
```

Figure A.1: XML example of generating an Array.

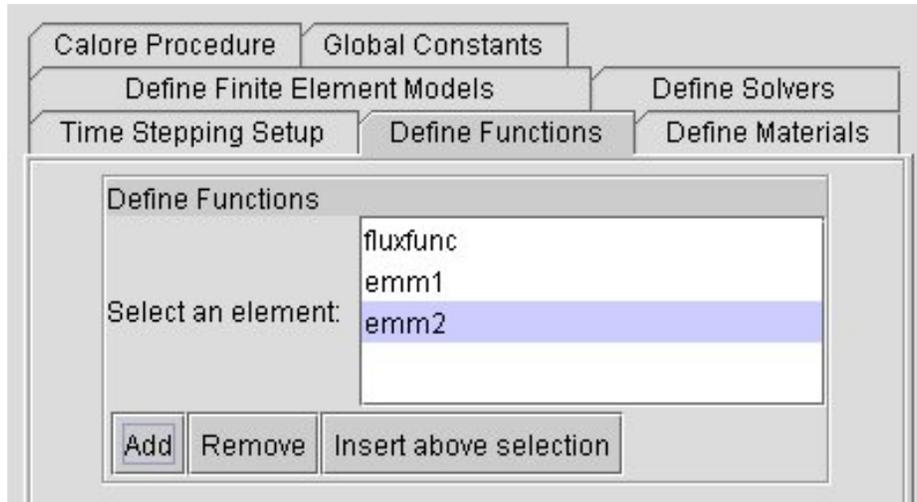


Figure A.2: Maui GUI generated by the input shown in Figure [A.1](#)

A.1.2.2 Optional Fields Example

Within each Calore/Maui class specified in the XML, many of the data fields are optional input to Calore. Integer, Double, String, and Reference objects may be flagged as optional by using the `optional="true"` attribute for these XML data types. If no `optional` attribute is set in the XML, then by default the parameter is assumed to be required.

In the GUI, required fields are designated by making the field label bright blue. Labels of optional fields are in the standard gray type. See Figure [A.3](#). If a user tries to submit their data by using the "Submit" button, or presses "OK" inside an array element editor, a error message will appear if required fields have not been filled in. Additionally, the action (submitting the data or exiting the array editor) will not be carried out until all the required fields have been filled in.

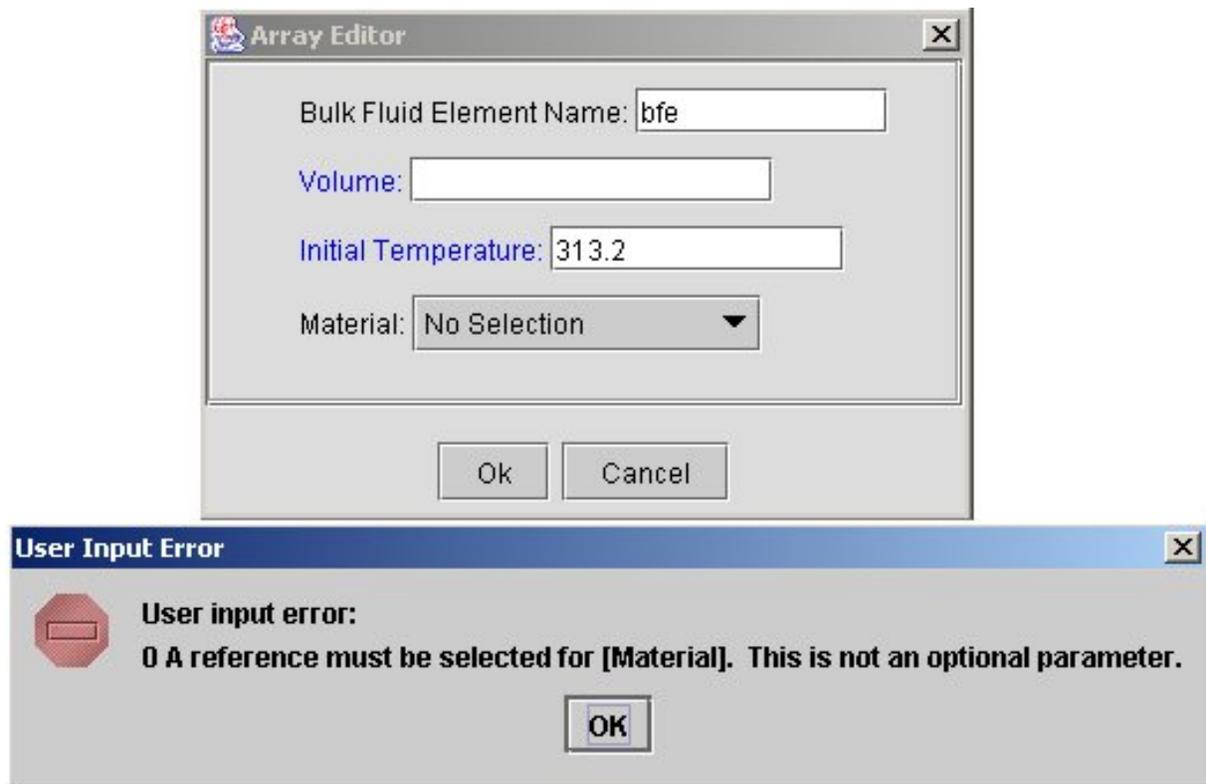


Figure A.3: Maui GUI demonstrating optional and required fields.

The class to represent a bulk fluid element in Figure [A.4](#) shows how some fields may be set to optional, while others are required. Both the `bfeName` string and the `bfeMaterialRef` are required. For `bfeName`, `optional="false"` has been specified; for `bfeMaterialRef`, the reference is required by default because of the omission of the `optional` attribute. The two double fields, `bfeVol` and `bfeInitTemp` are both optional since `optional="true"` has been specified.

```
<Class label="Bulk Fluid Element" type="BulkFluidElement">
  <Fields>
    <String name="bfeName" label="Bulk Fluid Element Name" optional="false"/>
    <Double name="bfeVol" label="Volume" optional="true"/>
    <Double name="bfeInitTemp" label="Initial Temperature" optional="true"/>
    <Reference select="true" output="reference" name="bfeMaterialRef"
      label="Material" path="root/calMain/matArrObj/materialArr"/>
  </Fields>
</Class>
```

Figure A.4: XML example of using optional and required attributes.

A.1.2.3 Subclassing Example

Subclassing within the Calore XML is done in two circumstances. First, when a choice of one particular object from a set of similar objects is warranted, subclassing is used. Second, when we wish the user to make a choice between specifying several data fields for a data object, or not using that object at all, we can use subclassing.

Calore solvers are an example of subclass usage to represent different derivations of a similar base object. There are many different solvers available in Calore, and the user must choose only one of them. See Figure [A.5](#) for the GUI component that represents the selection among solvers. Shown in Figure [A.6](#) is the XML for representing two of the solvers, and the base class from which they are derived. Figure [A.7](#) shows the interface when the Aztec solver has been selected. All of the solvers contain a solver block name, as well as fields for setting up debugging. These fields are contained as members of the `Solver` base class. Further, several of the solvers had several data fields in common, so each of these solvers contain a class instance of the `SolverShared` class. Finally, the unique fields that are specific to a particular solver are contained in the particular solver class (i.e. Aztec, Hypre, ISIS, etc.). See `$MAUI_HOME/Doc/tutorials/maui/XML/caloreSolvers.xml` for the full XML for all the solvers used in the Calore/Maui interface.

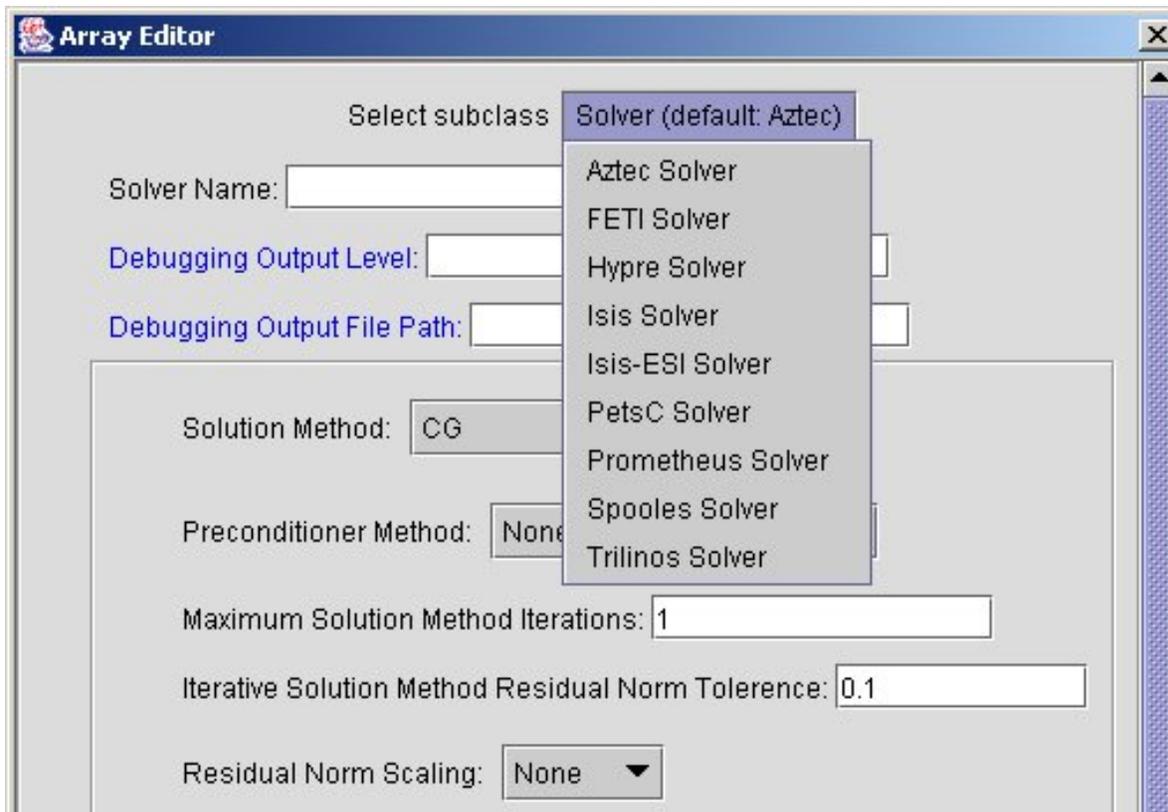


Figure A.5: Maui GUI example of solver subclass selection.

```
<Class type="Solver" label="Solver (default: Spooles)">
  <Fields>
    <String name="solverName" label="Solver Name"/>
    <Int name="debugOutLevel" label="Debugging Output Level" optional="true"/>
    <String name="debugOutFile" label="Debugging Output File Path"
      optional="true"/>
  </Fields>
</Class>

<Class type="SolverShared" label="">
  <Fields>
    <String name="solutionMethod" label="Solution Method">
```

```

    <Menu options="CG | CGS | BICGSTAB | GMRES | DEFGMRES | FGMRES | QMR | LU | BCGS" />
</String>
<String name="precondMethod" label="Preconditioner Method">
    <Menu options="None | Jacobi | Block-Jacobi | Neumann | Least-Squares | DD-LU" />
</String>
<Int name="maxIters" label="Maximum Solution Method Iterations"
    default="1" lower="1" />
<Double name="maxTol"
    label="Iterative Solution Method Residual Norm Tolerance"
    default="0.1" lower="0.0" />
<String name="residNormScal" label="Residual Norm Scaling">
    <Menu options="None | RHS | R0 | URHS | ANORM" />
</String>
<Int name="restartCt"
    label="Iterative Solution Method Restart Iteration Count"
    optional="true" />
<Int name="polyOrder" label="Polynomial Order of Preconditioning Method"
    optional="true" />
</Fields>
</Class>

<!-- SpoolesSolver class -->
<Class type="SpoolesSolver" label="Spooles Solver" base="Solver">
    <Fields>
        <String name="matRedux" label="Matrix Reduction">
            <Menu options="None | Hypre-Schur | Hypre-Slide | FEI-Remove-Slaves" />
        </String>
        <String name="matViewMachine" label="Host Running Matrix Viewer"
            optional="true" />
        <Integer name="matViewPort" label="Host Port to Matrix Viewer"
            optional="true" />
    </Fields>
</Class>

<!-- AztecSolver class -->
<Class type="AztecSolver" label="Aztec Solver" base="Solver">
    <Fields>
        <SolverShared name="azSolvShare1" label="" />
        <String name="matRedux" label="Matrix Reduction">
            <Menu options="None | Hypre-Schur | Hypre-Slide | FEI-Remove-Slaves" />
        </String>
        <Int name="azPrecondSteps"
            label="Number of Preconditioning Methods' Applications/Iteration"
            optional="true" />
        <Int name="iluFill" label="Fill-in Parameter for Incomplete Factorizations"
            optional="true" />
        <Double name="iluThresh"
            label="Threshold parameter for Incomplete Factorizations"
            optional="true" />
        <Integer name="multLevels"

```

```

    label="# Levels for Multi-Level/MultiGrid Solvers"
    optional="true"/>
<String name="matViewMachine" label="Host Running Matrix Viewer"
    optional="true"/>
<Integer name="matViewPort" label="Host Port to Matrix Viewer"
    optional="true"/>
</Fields>
</Class>

```

Figure A.6: XML example of defining subclasses.

The screenshot shows the 'Array Editor' window with the following configuration options:

- Select subclass: Aztec Solver
- Solver Name:
- Debugging Output Level:
- Debugging Output File Path:
- Solution Method: CG
- Preconditioner Method: None
- Maximum Solution Method Iterations:
- Iterative Solution Method Residual Norm Tolerance:
- Residual Norm Scaling: None
- Iterative Solution Method Restart Iteration Count:
- Polynomial Order of Preconditioning Method:
- Matrix Reduction: None
- Number of Preconditioning Methods' Applications/Iteration:
- Fill-in Parameter for Incomplete Factorizations:
- Threshold parameter for Incomplete Factorizations:
- # Levels for Multi-Level/MultiGrid Solvers:
- Host Running Matrix Viewer:
- Host Port to Matrix Viewer:

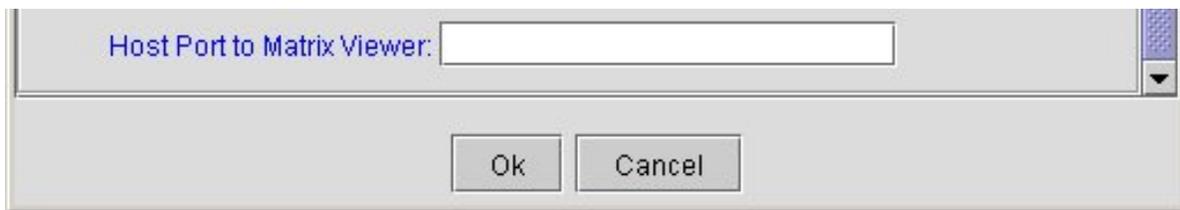


Figure A.7: Maui GUI representation of the Aztec solver.

Using subclassing to allow or disallow setup of a set of parameters is demonstrated in the Viewfactor solver. As we have seen above, single parameters can be made optional by setting the XML attribute to `optional="true"`. However, for large blocks of data that pertain to a particular type of settings, keeping these optional parameters out of sight unless the user desires to use them gives a cleaner look in the GUI. Shown in Figure [A.8](#), the XML for representing the option of not displaying or displaying the data fields for the Viewfactor solver are shown. Figures [A.9](#) and [A.10](#) show the GUI representation of this type of subclassing choice.

```
<Class type="ViewfactorSolverBase"
  label="Viewfactor Solver Parameters (default: none)"/>

<Class type="ViewfactorSolverNO"
  label="No Viewfactor Solver Parameter Specifications"
  base="ViewfactorSolverBase"/>

<Class type="ViewfactorSolverYES" label="Set Viewfactor Solver Parameters"
  base="ViewfactorSolverBase">
  <Fields>
    <ViewFactorOutFile name="vfOutFile"
      label="Viewfactor Output File Settings"/>
    <String name="vfInfile" label="Viewfactor Input File Name"
      optional="true"/>
    <Boolean name="vfStagedIO" label="Use Staged I/O" default="false"/>
    <Int name="vfNumControllers" label="Number of Controllers"
      optional="true"/>
    <String name="vfCoupRule" label="Coupling Rule">
      <Menu options="Lagged|Mason"/>
    </String>
    <String name="vfSolMethod" label="Solution Method">
      <Menu options="Chaparral Gauss|Chaparral GMRES|Chaparral CG"/>
    </String>
    <String name="vfSolOutRule" label="Solution Output Rule">
      <Menu options="None|Summary|Verbose"/>
    </String>
    <Double name="vfConvTol" label="Convergence Tolerance" optional="true"/>
    <Int name="vfMaxIters" label="Maximum Number of Iterations" default="1"/>
    <Double name="vfRelaxFactor" label="Relaxation Factor"
      optional="true" />
  </Fields>
</Class>
```

Figure A.8: XML using subclassing to provide a yes/no subsection of data selection.

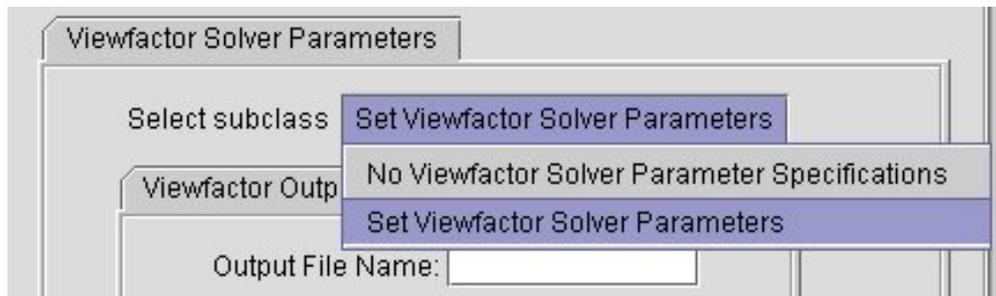


Figure A.9: Maui GUI example of subclassing to provide a yes/no subsection of data selection.

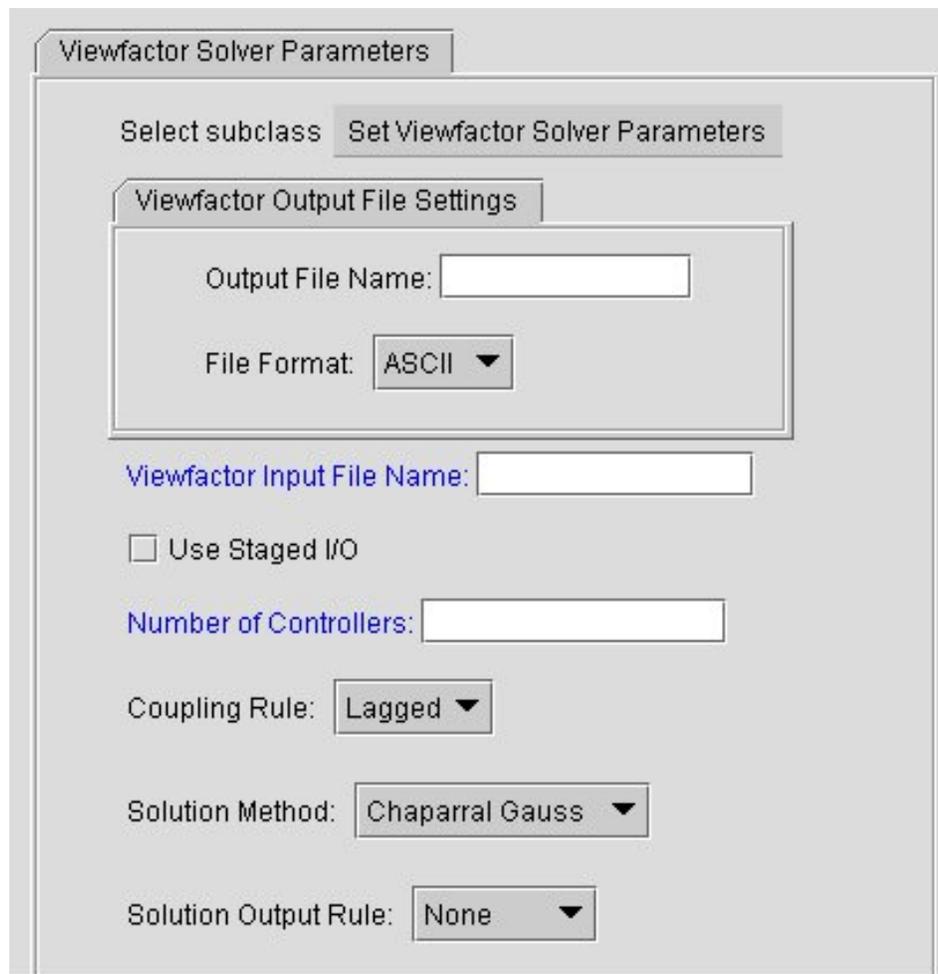


Figure A.10: Maui GUI example of subclassing to provide a yes/no subclass selection, with the "yes" option ("Set Viewfactor Solver Parameters") selected.

A.1.2.4 Referencing Example

References are an important part of the Calore/Maui GUI. The user specifies functions, materials, and solvers, and then

may want to refer to these objects in other parts of the interface. For example, the user may define a variety of functions, then use one or more of these functions to specify material properties for a material (Figures [A.2](#) and [A.11](#)). Likewise, a finite element model depends on blocks of materials that may be referred to after the materials have been set up. Figure [A.12](#) shows the XML for specifying the reference to a material contained in a finite element object.

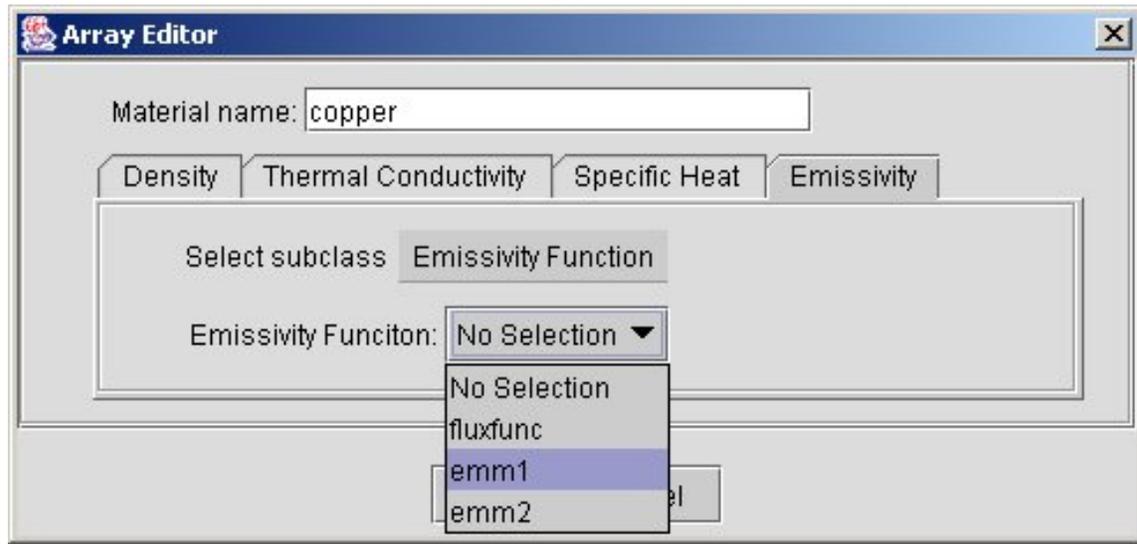


Figure A.11: Maui GUI example using referencing

```
<Class type="FEBlock" label="Finite Element Block Specifications" >
  <Fields>
    <String name="blockName" label="Finite Element Block Name" optional="false"/>
    <Reference name="matRef" label="Material"
      path="root/calMain/matArrObj/materialArr"
      select="true" output="reference"/>
  </Fields>
</Class>
```

Figure A.12: XML example of specifying a reference to an array.

The path for a reference can be specified one of two ways: tracing the path to the reference from the root of the class hierarchy in the input file, or by providing a relative the path to the reference from the current class. See Section [2.8](#) for more details on creating References.

Figure [A.12](#) shows specification from the root. The `root` label in a reference says to go to the top level object that contains all other objects in this interface. In the Calore example, the `root` object is Calore See `$MAUI_HOME/Doc/tutorials/maui/XML/calore.xml` for more examples of reference usage.

A.1.2.5 Import Example

One writing an XML interface file for Maui can break the different classes that are a part of a user interface into smaller

files for better readability. To include these classes as part of the main XML specification for the GUI, `import` statements may be used to insert the imported XML into the main XML file. The main XML file is the one that specifies the root class with the beginning element, `<Maui rootClass="root class name">`. In the Calore example the main Calore file is the one containing the beginning element `<Maui rootClass="Calore">`.

In the main Calore XML input file, there is an `import` statement for bringing in the XML specification for all the solvers. The XML from the imported file is essentially inserted in the main Maui file at the point which the `import` statement is entered. It is imperative to make sure that class nesting is done appropriately for this reason. You would not want to bring in fully defined classes using `import` in the middle of another class, but between other class specifications in the XML.

Figure [A.13](#) shows the import specification for the `caloreSolver.xml` file.

```
<Class label="Define Solvers" type="SolverArrClass">
  <Fields>
    <Array name="solverArr" label="Define Solvers">
      <Master>
        <Solver name="solver" label="Solver"/>
      </Master>
    </Array>
  </Fields>
</Class>
```

Figure A.13: Example of using an `import` statement in the XML to include XML contained in another file.

A.1.2.6 String Menu Example

String menus are useful when the user has only a fixed number of choices for a string to enter into the editor. Rather than having the user remember which strings are legal, a list of possible selections can be generated with a string menu.

In Figure [A.14](#) we see both a text box string and menu string specification. The variable `vfOutfile` will be represented with a string text box where the user can input the name of the file. The variable `vfOutFormat` will have a drop down menu with three file type selection options, ASCII, Binary, or XDR. See Figure [A.15](#) for the GUI rendering of this class.

```
<Class label="Viewfactor Output File Settings" type="ViewFactorOutFile"
collapsible="false">
  <Fields>
    <String name="vfOutfile" label="Output File Name"/>
    <String name="vfOutFormat" label="File Format">
      <Menu options="ASCII|Binary|XDR"/>
    </String>
  </Fields>
</Class>
```

Figure A.14: XML example of how a `StringMenu` is defined.



Figure A.15: Maui GUI example of rendering of string text box and string menu

[Next](#) [Up](#) [Previous](#)

Next: [A.2 Text Input to](#) **Up:** [A.1 A Calore GUI](#) **Previous:** [A.1.1 Differences Between Text](#)

[Next](#) [Up](#) [Previous](#)**Next:** [B. Maui XML syntax](#) **Up:** [A. Application Example](#) **Previous:** [A.1.2 Calore GUI Design](#)

A.2 Text Input to Calore

The following section contains an example Calore text input file. This file is an example of how a Calore user typically would typically specify input to Calore.

```
1 Begin sierra Calore
2
3   title radar chip temperatures - tighten int/conv tol
4
5 Begin Time Control
6   Begin Time Stepping Block ts1
7     start time = 0.0
8     Begin Parameters for Calore Region myRegion
9       time step = 0.01
10      transient step type=automatic
11      time integration rule=implicit
12      predictor rule=forward euler
13      ABSOLUTE SOLN LIMIT = 1.0E20
14      MAX TIME STEP = 10.0
15      MIN TIME STEP = 0.0001
16      MAX TIME TRUNC ERROR = 1.0e-3
17      mass matrix = lumped
18    End
19  End Time Stepping Block ts1
20  Termination time=0.13
21 End Time Control
22
23 Begin Property Specification for Material Copper
24   Density =8900.
25   Specific Heat =385.
26   Thermal Conductivity =300.
27 End
28 Begin Property Specification for Material Duroid3010
29   Density =3000.
30   specific Heat =930.
```

```
31     Thermal Conductivity =0.44
32 End
33 Begin Property Specification for Material ChipBody #GaS
34     Density =5320.
35     Specific Heat =333.
36     Thermal Conductivity =44.
37 End
38
39 Begin Definition for Function fluxfunc1
40     type is piecewise linear
41     begin values
42         0. 0.0
43         1. 8.38e6
44         10000. 8.38e6
45     end values
46 end
47 Begin Definition for Function emm2
48     type is constant
49     begin values
50         0.5
51     end values
52 end
53 Begin Definition for Function emm3
54     type is constant
55     begin values
56         0.3
57     end values
58 end
59
60 Begin Global Constants
61     Stefan Boltzmann constant= 5.67e-8
62 end
63
64 Begin Finite Element Model bar
65     database name = radar.par
66
67     Begin parameters for block block_1
68         material ChipBody
69     End
70     Begin parameters for block block_2
71         material Copper
72     End
73     Begin parameters for block block_3
```

```
74     material Duroid3010
75     End
76
77 End Finite Element Model bar
78
79
80 Begin aztec equation solver solve_temperature
81     solution          method   = cg
82     preconditioning method = dd-icc
83     maximum iterations   = 10000
84     residual norm tolerance = 1.0e-5
85     residual norm scaling  = none
86     Matrix Reduction = fei-remove-slaves
87 End
88
89 Begin Calore procedure myProcedure
90
91     Begin Calore region myRegion
92
93         use finite element model bar
94         use linear solver solve_temperature
95
96         Begin Constant Initial Condition Block ic1
97             temperature = 313.15
98             all volumes
99         End
100
101         Begin Results Output Label diffusion output
102             database name = radar.e
103             at time 0., increment = 10.
104             Title Radar Chip Temperatures
105             Nodal Variables = temperature as Temperature
106         End
107
108         Begin Constant Essential Boundary Condition on surface_7
109             temperature=313.15
110         End
111
112
113         Begin Unsteady Heat Flux Boundary Condition Abdul
114             add surface surface_10
115             flux function = fluxfunc1
116         end
```

```
117
118     Begin Contact Definition mpc1
119         contact surface surf1 contains surface_1
120         contact surface surf2 contains surface_2
121         contact surface surf3 contains surface_3
122
123         begin interaction inter1
124             master=surf2
125             slave=surf1
126             normal tolerance = 0.00001
127             tangential tolerance = 0.00001
128         end
129         begin interaction inter2
130             master=surf2
131             slave=surf3
132             normal tolerance = 0.00001
133             tangential tolerance = 0.00001
134         end
135     end contact definition mpc1
136
137 End Calore region myRegion
138
139 End Calore procedure myProcedure
140
141 End sierra Calore
```

[Next](#) [Up](#) [Previous](#)

Next: [B. Maui XML syntax](#) **Up:** [A. Application Example](#) **Previous:** [A.1.2 Calore GUI Design](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.1 Tag Maui](#) **Up:** [A Maui User's Guide](#) **Previous:** [A.2 Text Input to](#)

B. Maui XML syntax guide

Subsections

- [B.1 Tag Maui](#)
 - [B.1.1 Children allowed in Maui elements](#)
 - [B.1.2 Attributes allowed in Maui elements](#)
- [B.2 Tag Class](#)
 - [B.2.1 Children allowed in Class elements](#)
 - [B.2.2 Attributes allowed in Class elements](#)
- [B.3 Tag Import](#)
 - [B.3.1 Children allowed in Import elements](#)
 - [B.3.2 Attributes allowed in Import elements](#)
- [B.4 Tag Fields](#)
 - [B.4.1 Children allowed in Fields elements](#)
 - [B.4.2 Attributes allowed in Fields elements](#)
- [B.5 Tag AppData](#)
 - [B.5.1 Children allowed in AppData elements](#)
 - [B.5.2 Attributes allowed in AppData elements](#)
- [B.6 Tag Action](#)
 - [B.6.1 Children allowed in Action elements](#)
 - [B.6.2 Attributes allowed in Action elements](#)
- [B.7 Tag CustomEditor](#)
 - [B.7.1 Children allowed in CustomEditor elements](#)
 - [B.7.2 Attributes allowed in CustomEditor elements](#)

- [B.8 Tag Integer](#)
 - [B.8.1 Children allowed in Integer elements](#)
 - [B.8.2 Attributes allowed in Integer elements](#)

- [B.9 Tag Double](#)
 - [B.9.1 Children allowed in Double elements](#)
 - [B.9.2 Attributes allowed in Double elements](#)

- [B.10 Tag Boolean](#)
 - [B.10.1 Children allowed in Boolean elements](#)
 - [B.10.2 Attributes allowed in Boolean elements](#)

- [B.11 Tag String](#)
 - [B.11.1 Children allowed in String elements](#)
 - [B.11.2 Attributes allowed in String elements](#)

- [B.12 Tag Array](#)
 - [B.12.1 Children allowed in Array elements](#)
 - [B.12.2 Attributes allowed in Array elements](#)

- [B.13 Tag Table](#)
 - [B.13.1 Children allowed in Table elements](#)
 - [B.13.2 Attributes allowed in Table elements](#)

- [B.14 Tag Reference](#)
 - [B.14.1 Children allowed in Reference elements](#)
 - [B.14.2 Attributes allowed in Reference elements](#)

- [B.15 Tag Comment](#)
 - [B.15.1 Children allowed in Comment elements](#)
 - [B.15.2 Attributes allowed in Comment elements](#)

- [B.16 Tag Menu](#)
 - [B.16.1 Children allowed in Menu elements](#)
 - [B.16.2 Attributes allowed in Menu elements](#)

- [B.17 Tag Master](#)
 - [B.17.1 Children allowed in Master elements](#)
 - [B.17.2 Attributes allowed in Master elements](#)

- [B.18 Tag Contents](#)
 - [B.18.1 Children allowed in Contents elements](#)
 - [B.18.2 Attributes allowed in Contents elements](#)

 - [B.19 Tag Item](#)
 - [B.19.1 Children allowed in Item elements](#)
 - [B.19.2 Attributes allowed in Item elements](#)

 - [B.20 Tag Header](#)
 - [B.20.1 Children allowed in Header elements](#)
 - [B.20.2 Attributes allowed in Header elements](#)

 - [B.21 Tag Entries](#)
 - [B.21.1 Children allowed in Entries elements](#)
 - [B.21.2 Attributes allowed in Entries elements](#)

 - [B.22 Tag Entry](#)
 - [B.22.1 Children allowed in Entry elements](#)
 - [B.22.2 Attributes allowed in Entry elements](#)

 - [B.23 Tag Cell](#)
 - [B.23.1 Children allowed in Cell elements](#)
 - [B.23.2 Attributes allowed in Cell elements](#)

 - [B.24 Tag *type_name*](#)
 - [B.24.1 Children allowed in *type_name* elements](#)
 - [B.24.2 Attributes allowed in *type_name* elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.1.1 Children allowed in Maui](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B. Maui XML syntax](#)

B.1 Tag Maui

Every Maui XML file should contain a single element with tag `Maui`. This element can contain class definitions and/or directives to import other Maui XML files.

Subsections

- [B.1.1 Children allowed in Maui elements](#)
 - [B.1.2 Attributes allowed in Maui elements](#)
-

[Next](#)
[Up](#)
[Previous](#)

Next: [B.1.2 Attributes allowed in](#)
Up: [B.1 Tag Maui](#)
Previous: [B.1 Tag Maui](#)

B.1.1 Children allowed in Maui elements

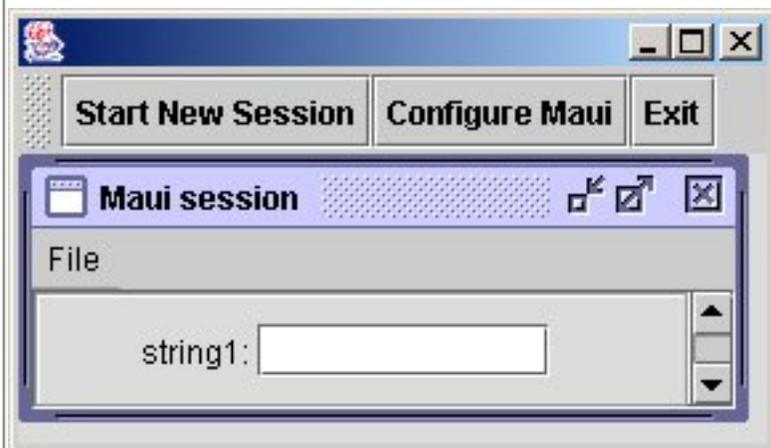
Tag	Number	Description and comments	Examples
Class	any number	Maui class specifications.	example
Import	any number	directives to load other Maui XML files.	example

<Maui> : all Maui XML docs must begin and end with a <Maui> tag.

```

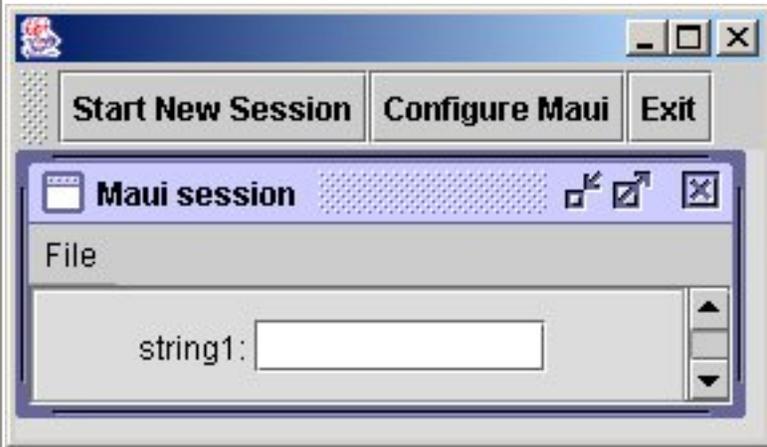
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1"/>
    </Fields>
  </Class>
</Maui>

```



<Class> : A class is a container for holding GUI components (buttons, textboxes, checkboxes, etc).

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1"/>
    </Fields>
  </Class>
</Maui>
```



<Import> : Used to insert the contents of xml files into the RootClass.

```
<Maui RootClass="MyContainer">
  <Import filename="MyMauiXmlFile.xml"/>
</Maui>
```

Next Up Previous

Next: [B.1.2 Attributes allowed in](#) **Up:** [B.1 Tag Maui](#) **Previous:** [B.1 Tag Maui](#)

[Next](#) [Up](#) [Previous](#)

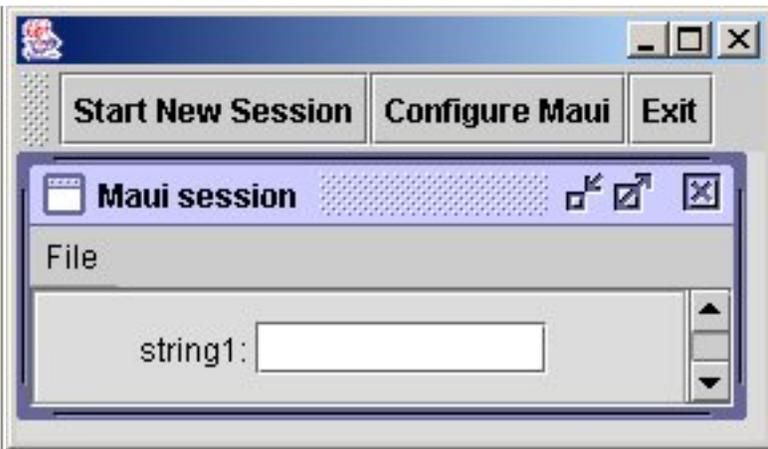
Next: [B.2 Tag Class](#) **Up:** [B.1 Tag Maui](#) **Previous:** [B.1.1 Children allowed in](#)

B.1.2 Attributes allowed in Maui elements

Attribute name	Mandatory	Allowed values	Description and comments	Examples
RootClass	no	legal class name	The RootClass attribute gives the name of classes that will be loaded as the root of the editor's data structure. There must be at least one occurrence of RootClass either in the main XML file or imported XML files. If there is more than one occurrence of RootClass, Maui will create a dropdown menu that holds all of the classes specified as RootClasses.	example

`<Maui>` : all Maui XML docs must begin and end with a `<Maui>` tag.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1"/>
    </Fields>
  </Class>
</Maui>
```



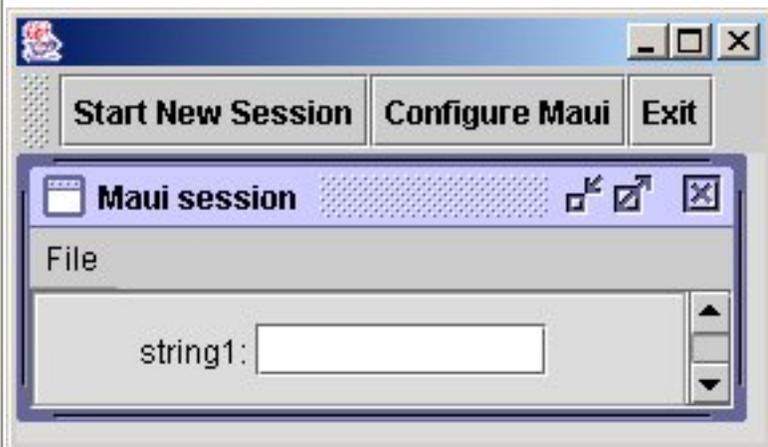
RootClass : The <Maui> tag can contain more than one class. The RootClass attribute contains the name of the class that will be displayed on the screen.

```
<Maui RootClass="Class1">

  <Class type="Class1">
    <Fields>
      <String name="string1" label="string1"/>
    </Fields>
  </Class>

  <Class type="Class2">
    <Fields>
      <String name="string2" label="string2"/>
    </Fields>
  </Class>

</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.2 Tag Class](#) **Up:** [B.1 Tag Maui](#) **Previous:** [B.1.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.2.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.1.2 Attributes allowed in](#)

B.2 Tag Class

Class elements contain a specification of the fields and layout of a Maui class.

Subsections

- [B.2.1 Children allowed in Class elements](#)
 - [B.2.2 Attributes allowed in Class elements](#)
-

[Next](#) [Up](#) [Previous](#)

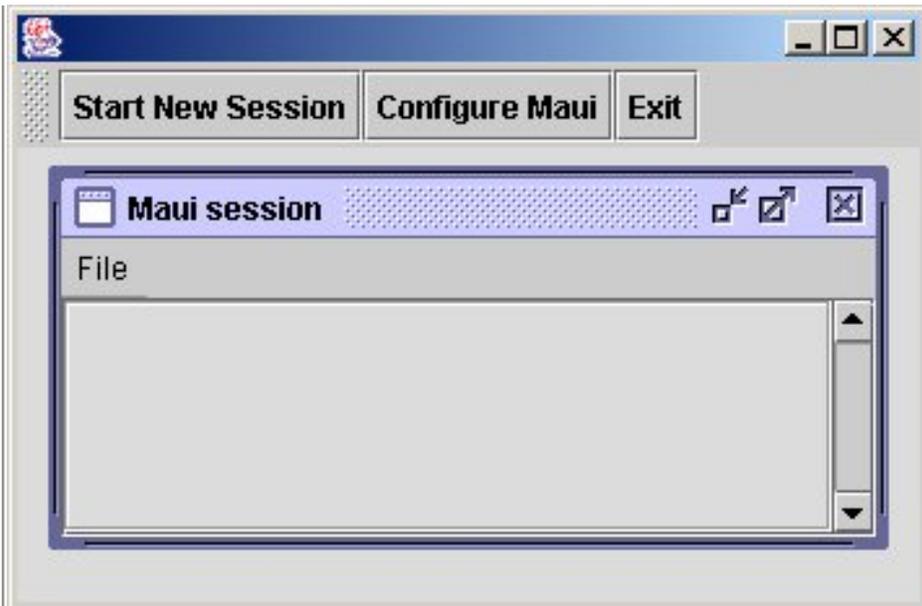
Next: [B.2.2 Attributes allowed in](#) **Up:** [B.2 Tag Class](#) **Previous:** [B.2 Tag Class](#)

B.2.1 Children allowed in Class elements

Tag	Number	Description and comments	Example
Action	any number	Allows for buttons to be placed within the class editor	example
CustomEditor	0-1	Allows the class to use a non-standard editor.	example
Fields	0-1	The Fields child contains all of the data members of this class.	example
AppData	0-1	a free form block of XML for data where no editor is needed	example
Help	0-1	Places a HELP icon on the screen. if the end-user clicks on the icon then helpful information pops up on the screen.	example
Subclasses	0-1	A tag that is used to hold child classes.	example

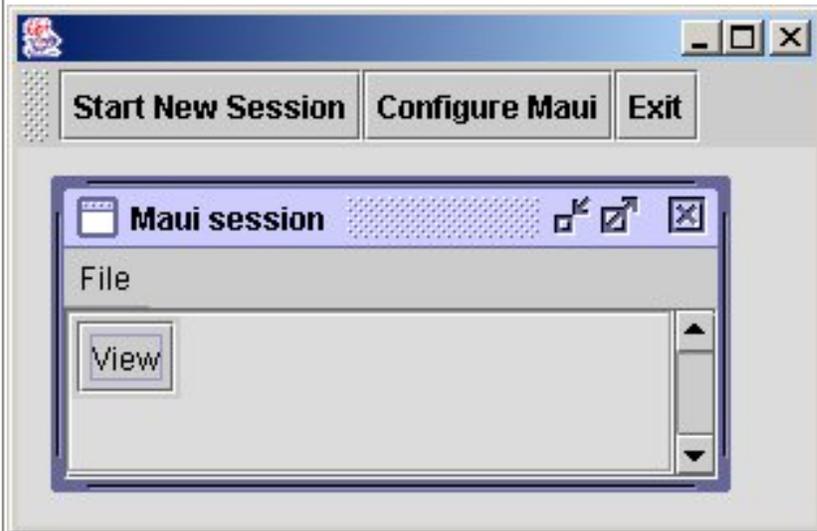
`<Class>` : creates a container to hold GUI components

```
<Class type="MyContainer" label="My container">
</Class>
```



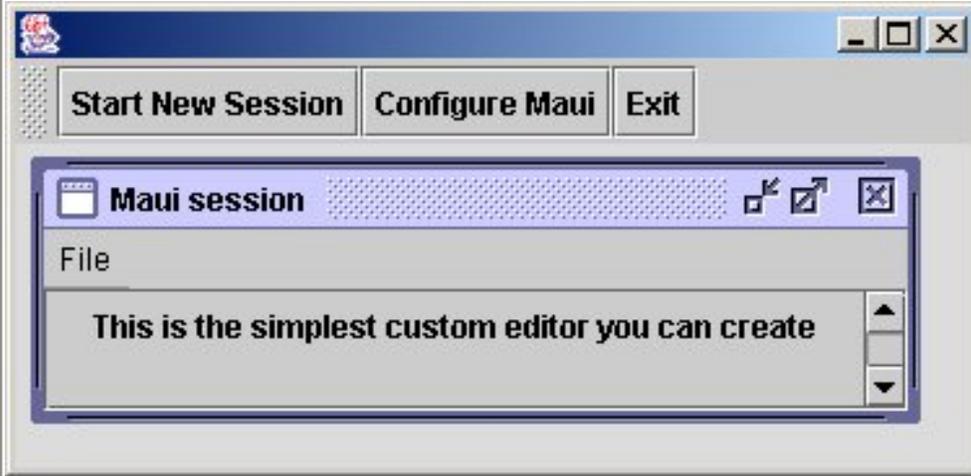
`<Action>` : Used to display buttons

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="My container">
    <Action label="View" class="Maui.Interface.ViewAction"/>
  </Class>
</Maui>
```



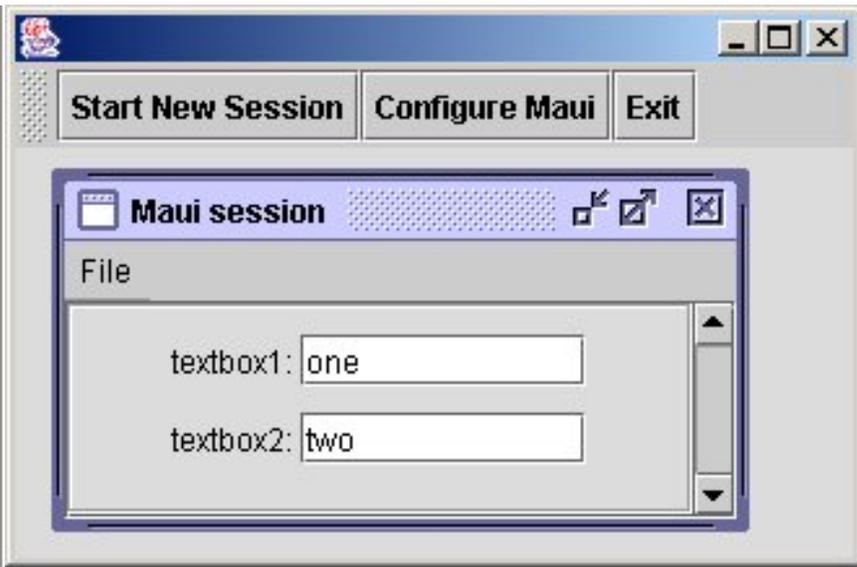
<CustomEditor> : Used to insert a custom editor into Maui

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <CustomEditor
      name="Maui.Editors.ExampleCustomEditor_BareBonesCustomEditor">
    </CustomEditor>
  </Class>
</Maui>
```



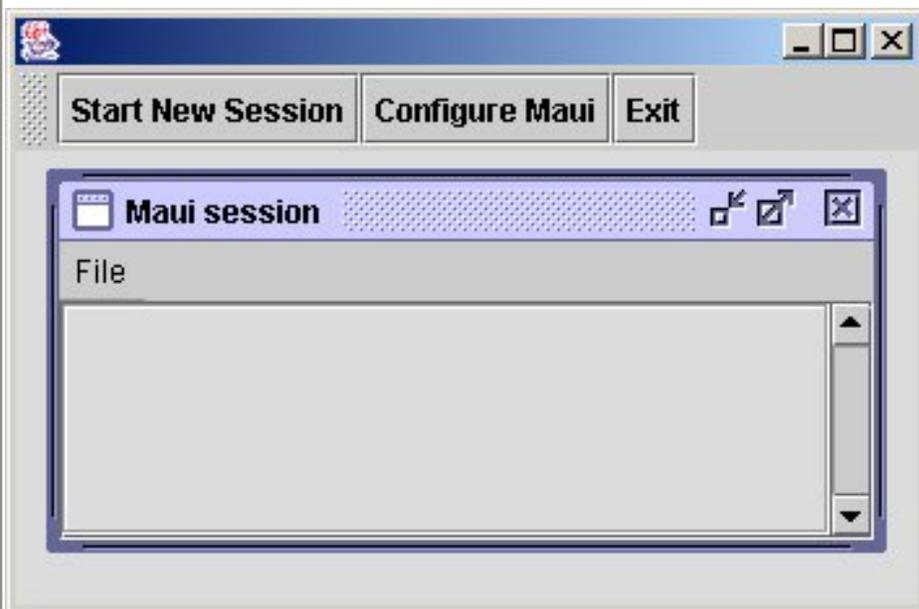
<Fields> : used to populate the container with GUI components

```
<Class type="MyContainer" label="My container">
  <Fields>
    <String name="textbox1" label="textbox1" default="one"/>
    <String name="textbox2" label="textbox2" default="two"/>
  </Fields>
</Class>
```



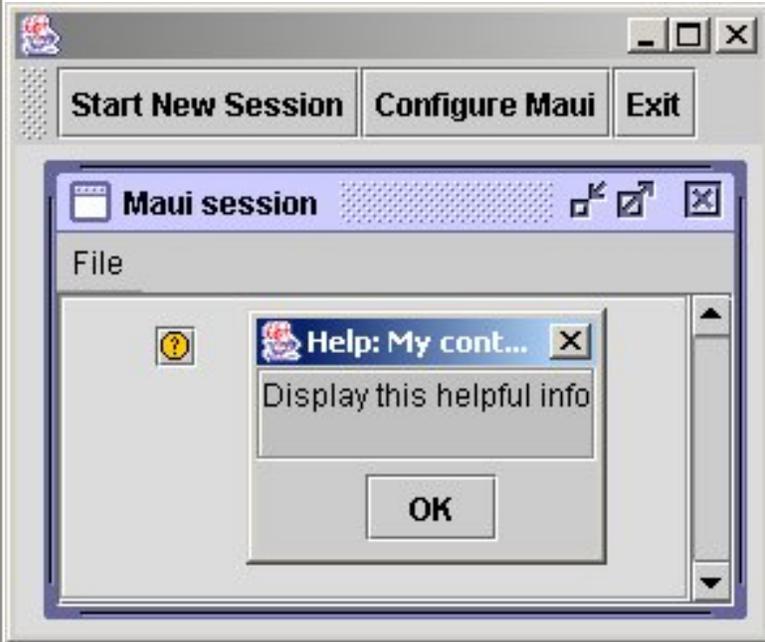
<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <AppData>
      <parameter1>one</parameter1>
      <parameter2>two</parameter2>
    </AppData>
  </Class>
</Maui>
```



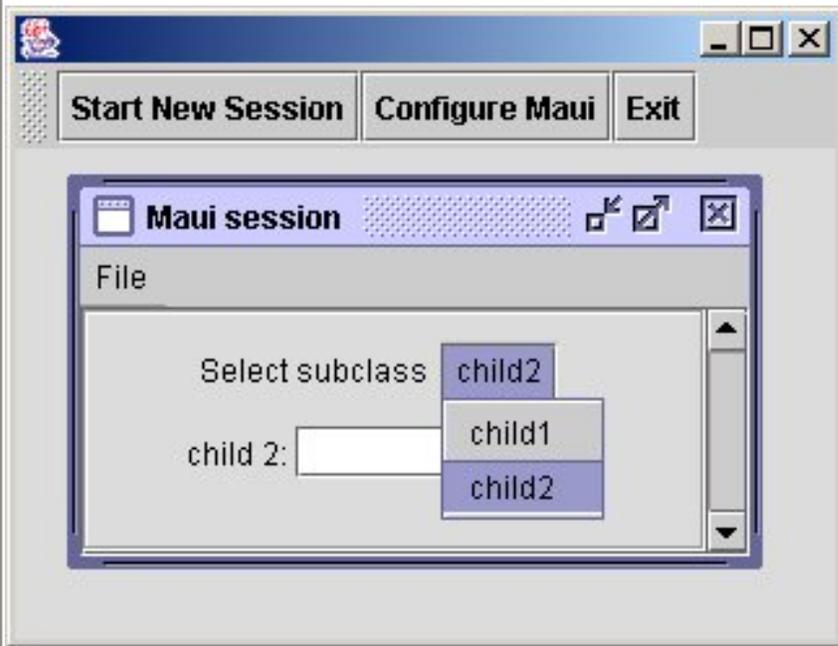
<Help> : Used to display helpful information

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="My container">
    <Help>
      Display this helpful info
    </Help>
  </Class>
</Maui>
```



<Subclasses> : used to populate container with child classes

```
<Class type="MyContainer" label="My container">
  <Subclasses>
    <Class type="Child1" label="child1">
      <Fields>
        <String name="child1textbox" label="child 1"/>
      </Fields>
    </Class>
    <Class type="Child2" label="child2">
      <Fields>
        <String name="child2textbox" label="child 2"/>
      </Fields>
    </Class>
  </Subclasses>
</Class>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.2.2 Attributes allowed in](#) **Up:** [B.2 Tag Class](#) **Previous:** [B.2 Tag Class](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.3 Tag Import](#) **Up:** [B.2 Tag Class](#) **Previous:** [B.2.1 Children allowed in](#)

B.2.2 Attributes allowed in Class elements

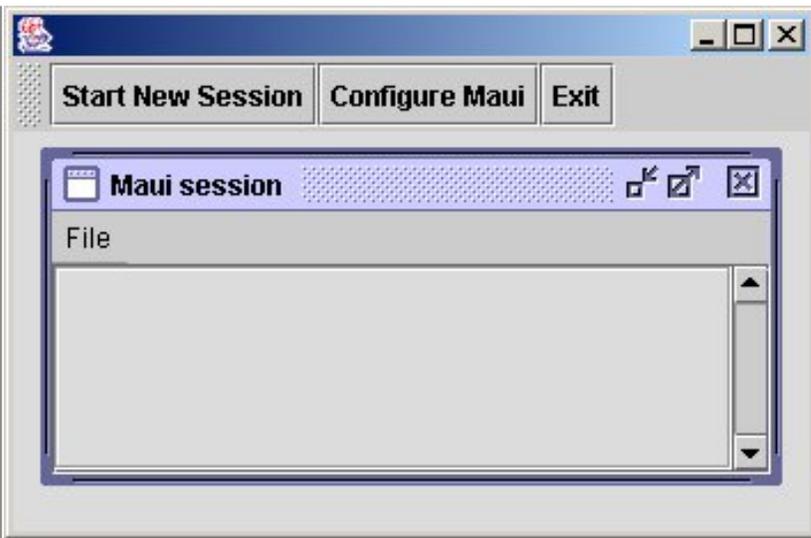
Attribute name	Mandatory	Allowed values	Description and comments	Example
type	yes	any legal class name	The value of the <code>type</code> attribute gives the name of the class	example
label	no	any string	The value of the <code>label</code> attribute gives the string that will be displayed where this class is used.	example
base	no	any legal class name	The value of the <code>base</code> attribute gives the name of this class' base class. Only required if we are defining a derived class.	example
altName	no	any string	a name used when an alternative to the regular class name is necessary for output. The default value for <code>altName</code> is the <code>type</code> of the class	example
selectionLabel	no	any string	The label displayed next to a subclass Selection menu if this class has subclasses	example example
collapsible	no	true or false	The panel for this class can be expanded and collapsed with a toggle button. If no value is given, <code>collapsible</code> is assumed to be false.	example
beginCollapsed	no	true or false	If the panel is collapsible, this will give the initial state of that panel. By default, a collapsible panel starts out expanded.	example

useTab	no	true or false	If this is true, the panel will be displayed in a tabbed pane with the other class panels. If false, it will be displayed by itself. If there is no collapsible attribute, useTab will default to true. If there is a collapsible attribute, useTab will default to false.	example
visible	no	true or false	Determines if the class is visible or invisible.	example
tooltip	no	any string	not yet implemented	example
startingsubclass	no	name of a subclass	When this class is first rendered on the screen, then which child class is visible on the screen?	example
layout	no	flow	If layout is set to "flow" then the GUI components are laid out from left to right, top to bottom.	example
border	no	true or false	Determines if the border (that surrounds the class) is visible or invisible.	example

A Class is a container to hold GUI components (textboxes, button, checkboxes, etc.).

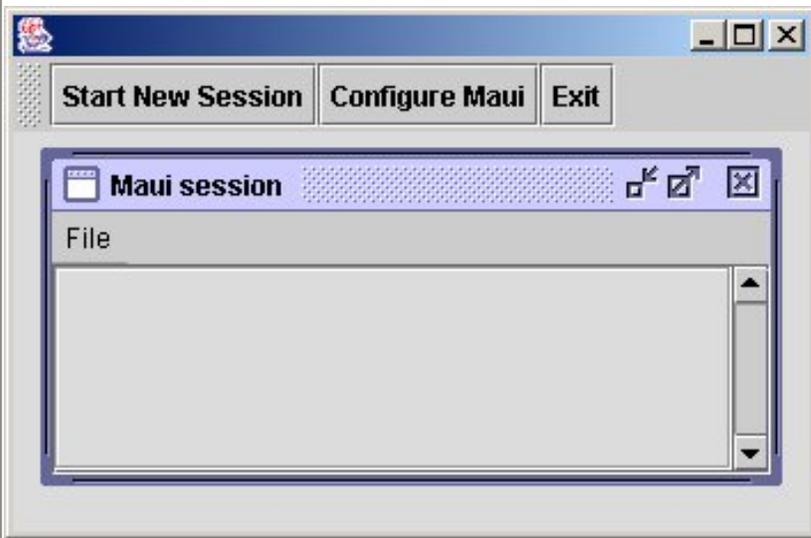
```
<Class> : creates a container to hold GUI components
```

```
<Class type="MyContainer" label="My container">
</Class>
```



type: Uniquely identifies the class.

```
<Class type="MyContainer" label="My container">
</Class>
```



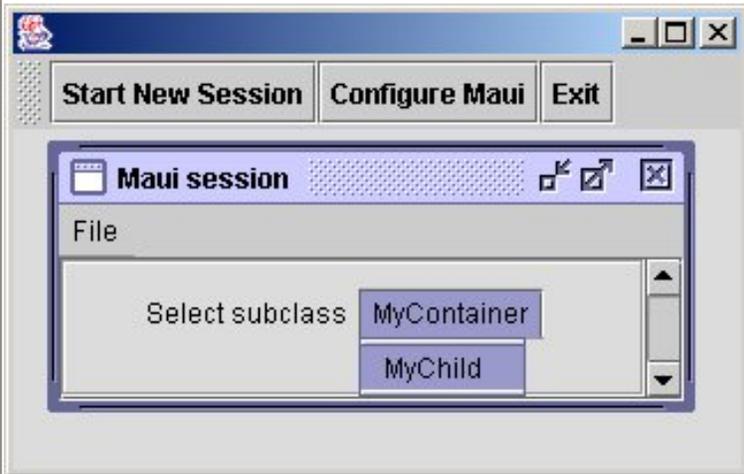
label: The label that appears above the class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="This is my label" collapsible="true">
    </Class>
</Maui>
```



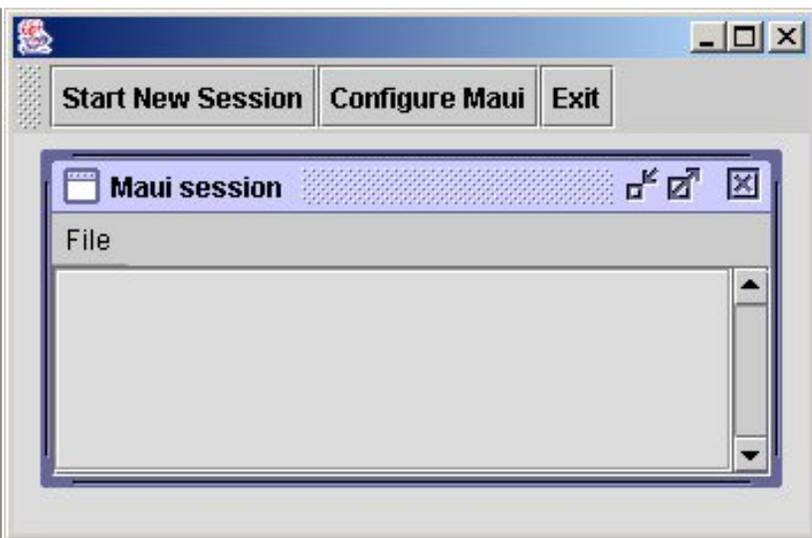
base: Used by a child class to identify the parent

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
  </Class>
  <Class type="MyChild" base="MyContainer">
  </Class>
</Maui>
```



altName: An alternate name used by the RootClass. Can be used by custom editors.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" altName="MyAltName">
  </Class>
</Maui>
```

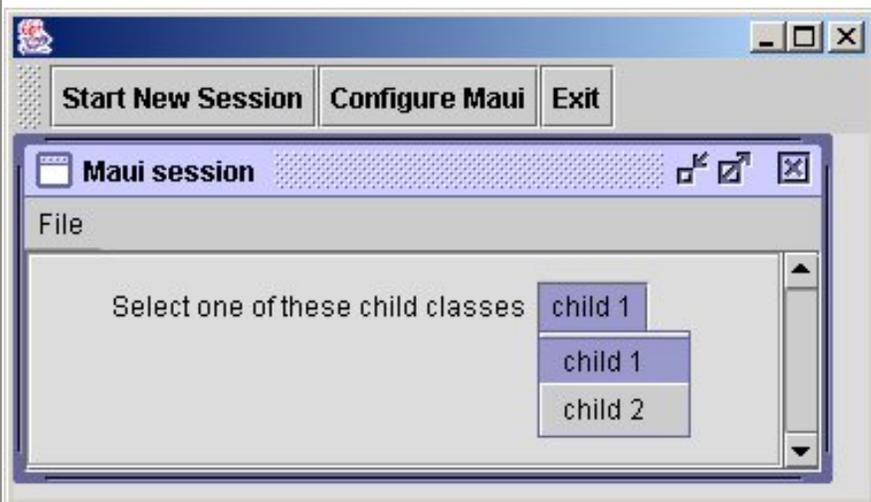


selectionLabel: if the end-user can use a pull-down menu to select a child class then this label appears to the left of the menu.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer"
        selectionLabel="Select one of these child classes">
  </Class>

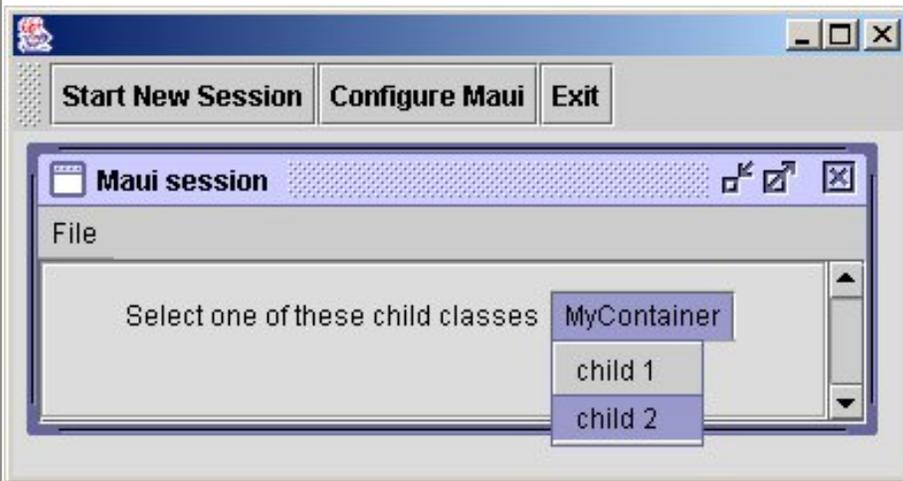
  <Class type="Child1" label="child 1" base="MyContainer"/>
  <Class type="Child2" label="child 2" base="MyContainer"/>

</Maui>
```



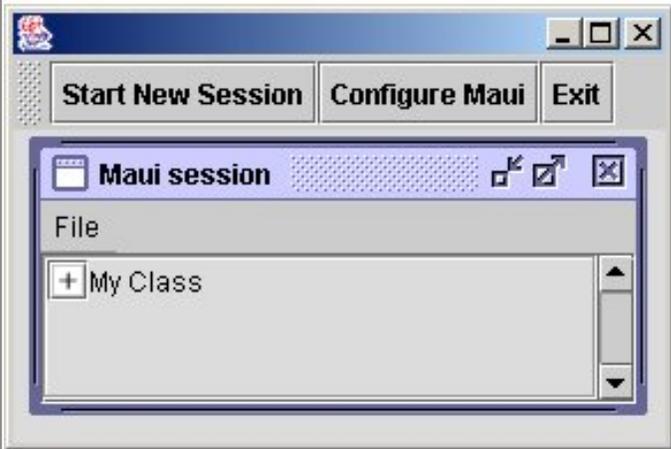
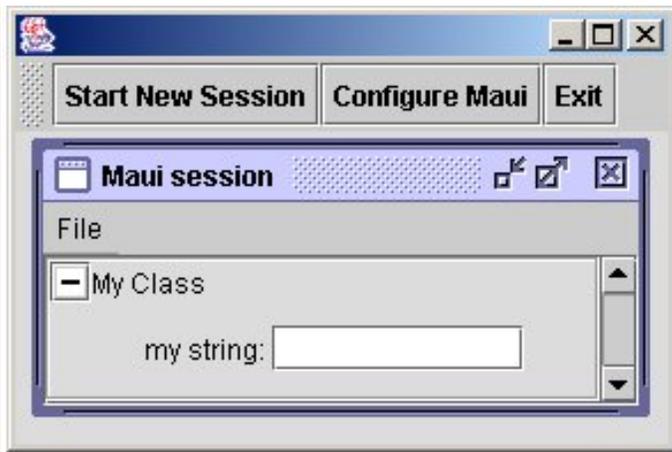
selectionLabel: if the end-user can use a pull-down menu to select a child class then this label appears to the left of the menu.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" selectionLabel="Select one of these child classes">
    <Subclasses>
      <Class type="Child1" label="child 1">
      </Class>
      <Class type="Child2" label="child 2">
      </Class>
    </Subclasses>
  </Class>
</Maui>
```



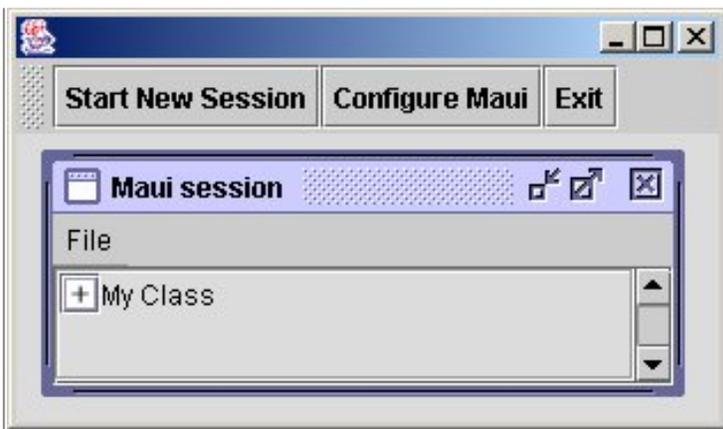
collapsible: The contents of a class can be hidden by clicking on the +/- button.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" collapsible="true" label="My Class">
    <Fields>
      <String name="myString" label=" my string"/>
    </Fields>
  </Class>
</Maui>
```



beginCollapsed: When the class first appears on the screen, is the class collapsed or uncollapsed?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" collapsible="true"
    beginCollapsed="true" label="My Class">
    <Fields>
      <String name="myString" label=" my string"/>
    </Fields>
  </Class>
</Maui>
```



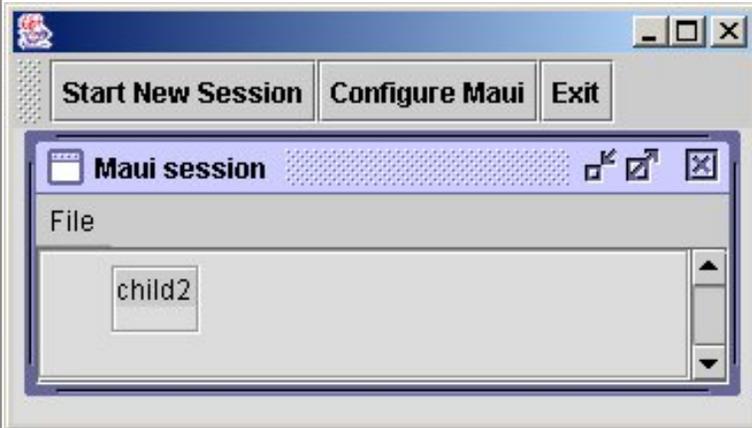
useTab: The end-user can select a child class by clicking on a tab

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="My Class">
    <Fields>
      <Class type="child1" label="child one" useTab="true">
        <Fields>
          <String name="string1" label="string1"/>
        </Fields>
      </Class>
      <Class type="child2" label="child two" useTab="true">
        <Fields>
          <String name="string2" label="string2"/>
        </Fields>
      </Class>
    </Fields>
  </Class>
</Maui>
```



visible: Determines if the class is visible on the screen

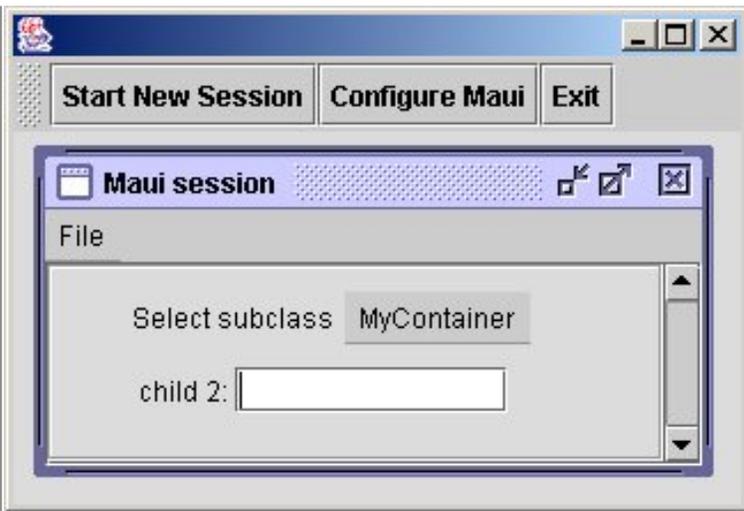
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="My Class">
    <Fields>
      <Class type="child1" label="child1" visible="false" useTab="false"/>
      <Class type="child2" label="child2" visible="true" useTab="false"/>
    </Fields>
  </Class>
</Maui>
```



tooltip: This feature is not yet implemented

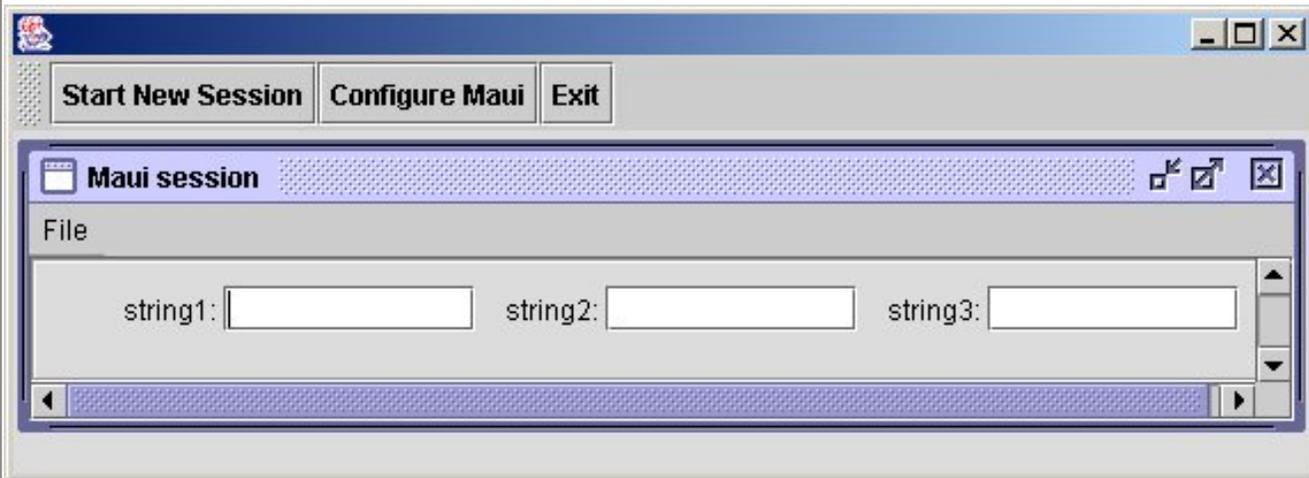
startingSubclass: When this class is first rendered on the screen, then which child class is displayed?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" startingSubclass="Child2">
    <Subclasses>
      <Class type="Child1" label="child 1">
        <Fields>
          <String name="child1" label="child 1"/>
        </Fields>
      </Class>
      <Class type="Child2" label="child 2">
        <Fields>
          <String name="child2" label="child 2"/>
        </Fields>
      </Class>
    </Subclasses>
  </Class>
</Maui>
```



layout: if set to "flow" then GUI components are laid out from left-to-right, top-to-bottom.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" startingSubclass="Child2" layout="flow">
    <Fields>
      <String name="string1" label="string1"/>
      <String name="string2" label="string2"/>
      <String name="string3" label="string3"/>
    </Fields>
  </Class>
</Maui>
```



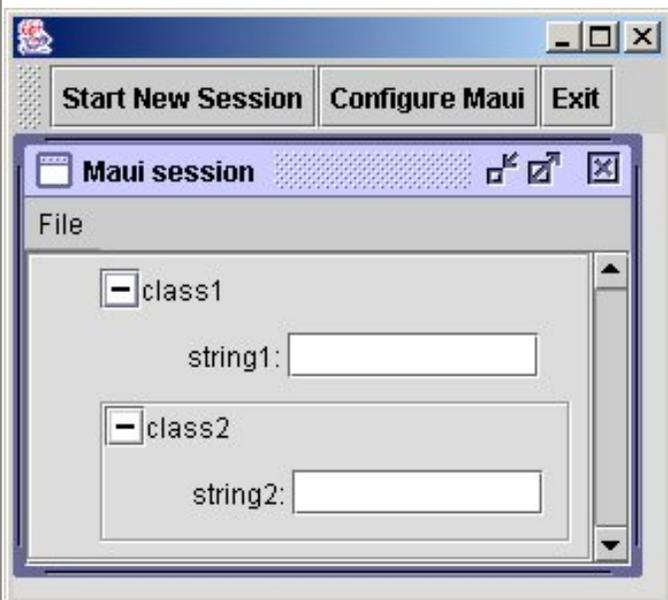
border: Determines if a border is drawn around the class.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>

      <Class type="class1" label="class1"
        collapsible="true" border="false">
        <Fields>
          <String name="string1" label="string1"/>
        </Fields>
      </Class>

      <Class type="class2" label="class2"
        collapsible="true" border="true">
        <Fields>
          <String name="string2" label="string2"/>
        </Fields>
      </Class>

    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.3 Tag Import](#) **Up:** [B.2 Tag Class](#) **Previous:** [B.2.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.3.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.2.2 Attributes allowed in](#)

B.3 Tag Import

`Import` elements contain the name of a file from which Maui should read additional input.

Subsections

- [B.3.1 Children allowed in Import elements](#)
 - [B.3.2 Attributes allowed in Import elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.3.2 Attributes allowed in](#) **Up:** [B.3 Tag Import](#) **Previous:** [B.3 Tag Import](#)

B.3.1 Children allowed in Import elements

[None](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.3.2 Attributes allowed in](#) **Up:** [B.3 Tag Import](#) **Previous:** [B.3 Tag Import](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.4 Tag Fields](#) **Up:** [B.3 Tag Import](#) **Previous:** [B.3.1 Children allowed in](#)

B.3.2 Attributes allowed in Import elements

Attribute name	Mandatory	Allowed values	Description and comments	Examples
filename	yes	path to a Maui file	The value gives the name of an XML file from which Maui can read in more definitions.	example

`<Import>` : Used to insert the contents of an xml file into a class.

```
<Maui RootClass="MyContainer">
  <Import filename="MyMauiXmlFile.xml"/>
</Maui>
```

filename: The name of the file that contains xml code.

```
<Maui RootClass="MyContainer">
  <Import filename="MyMauiXmlFile.xml"/>
</Maui>
```

[Next](#) [Up](#) [Previous](#)

Next: [B.4 Tag Fields](#) **Up:** [B.3 Tag Import](#) **Previous:** [B.3.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.4.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.3.2 Attributes allowed in](#)

B.4 Tag Fields

Fields elements contain as children the data members of a class.

Subsections

- [B.4.1 Children allowed in Fields elements](#)
 - [B.4.2 Attributes allowed in Fields elements](#)
-

[Next](#)
[Up](#)
[Previous](#)

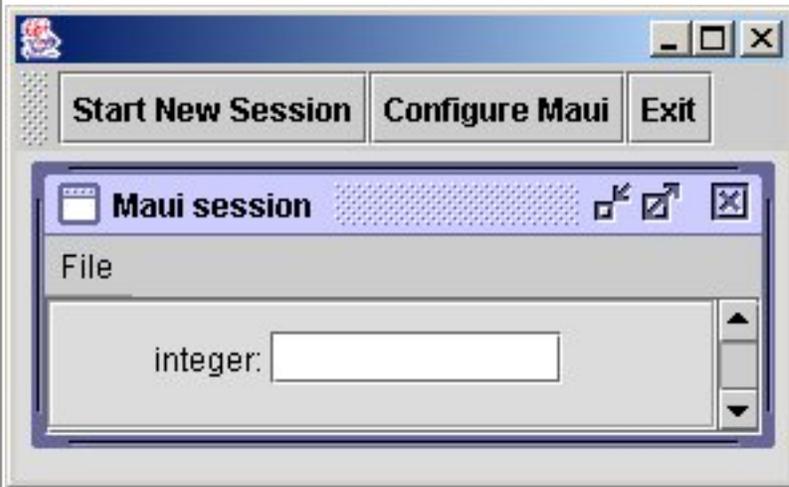
Next: [B.4.2 Attributes allowed in](#)
Up: [B.4 Tag Fields](#)
Previous: [B.4 Tag Fields](#)

B.4.1 Children allowed in Fields elements

Tag	Number	Description and comments	Examples
Integer	any number	zero or more integer variables	example
String	any number	zero or more string variables	example
Double	any number	zero or more double variables	example
Boolean	any number	zero or more boolean (logical) variables	example
Array	any number	zero or more array variables	example
Table	any number	zero or more tables	example
Reference	any number	zero or more reference variables	example
Comment	any number	zero or more comments	example
<i>type_name</i>	any number	zero or more class data members of type <i>type_name</i>	example
BR	any number	zero or more line breaks; each line break skips to the next line.	example
Class	any number	zero or more child classes.	example

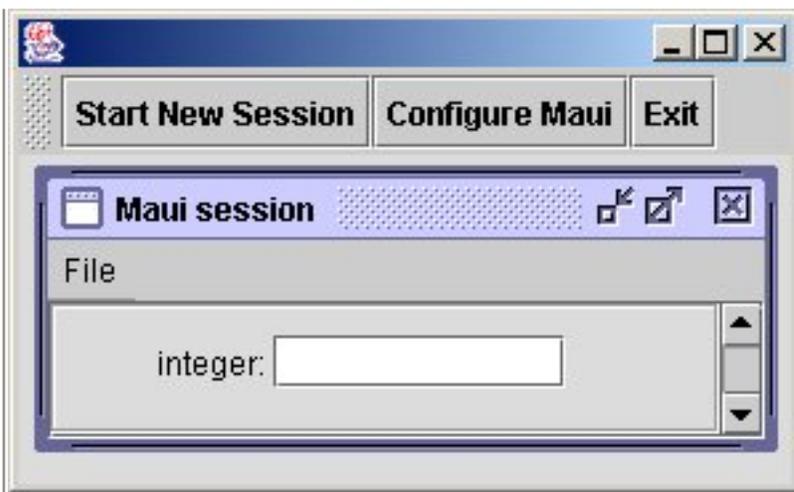
<Fields> : Contains the GUI components (buttons, textboxes, checkboxes, etc) that will be placed on the screen.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"/>
    </Fields>
  </Class>
</Maui>
```



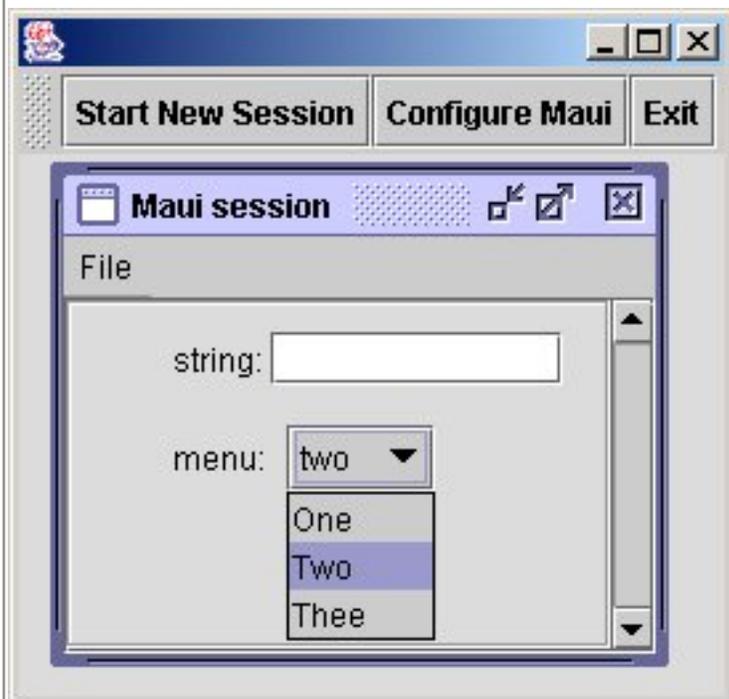
<Integer> : display a textbox that accepts integer values

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"/>
    </Fields>
  </Class>
</Maui>
```



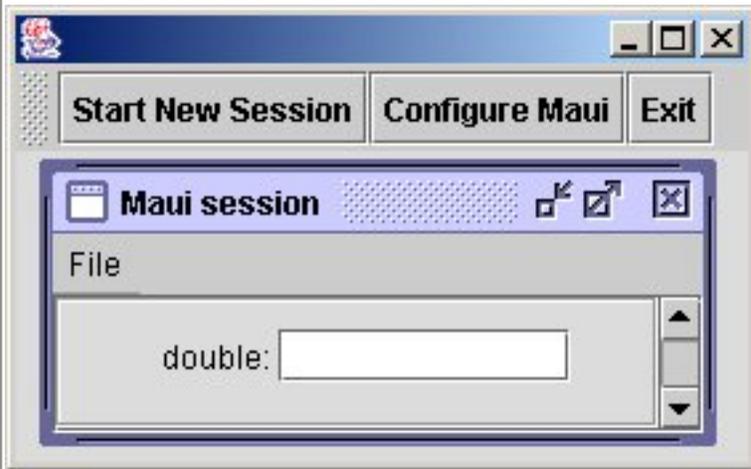
<String> : display a textbox or a pull-down menu

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string"/>
      <String name="myMenu" label="menu" default="two">
        <Menu options="One | Two | Thee" />
      </String>
    </Fields>
  </Class>
</Maui>
```



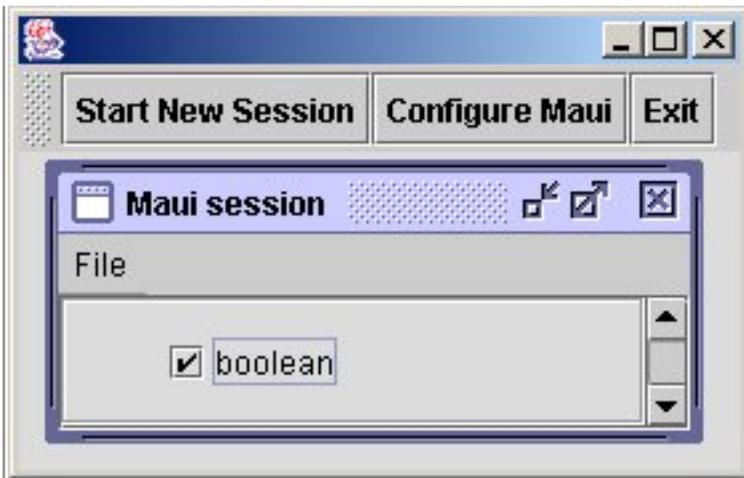
<Double> : display a textbox that accepts floating point numbers

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" />
    </Fields>
  </Class>
</Maui>
```



<Boolean> : display a checkbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean" />
    </Fields>
  </Class>
</Maui>
```



<Array>

```

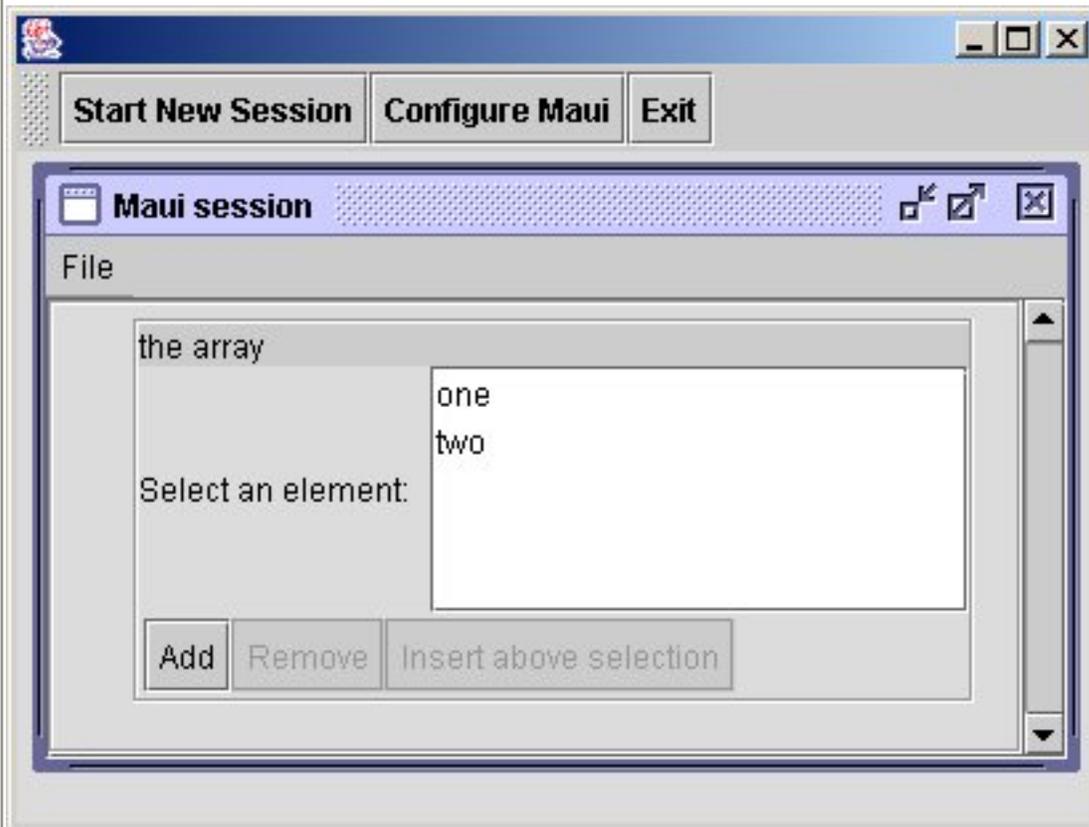
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>

```

```

        </Fields>
      </Class>
    </Item>
  </Contents>
</Array>
</Fields>
</Class>
</Maui>

```



```
<Table>
```

```

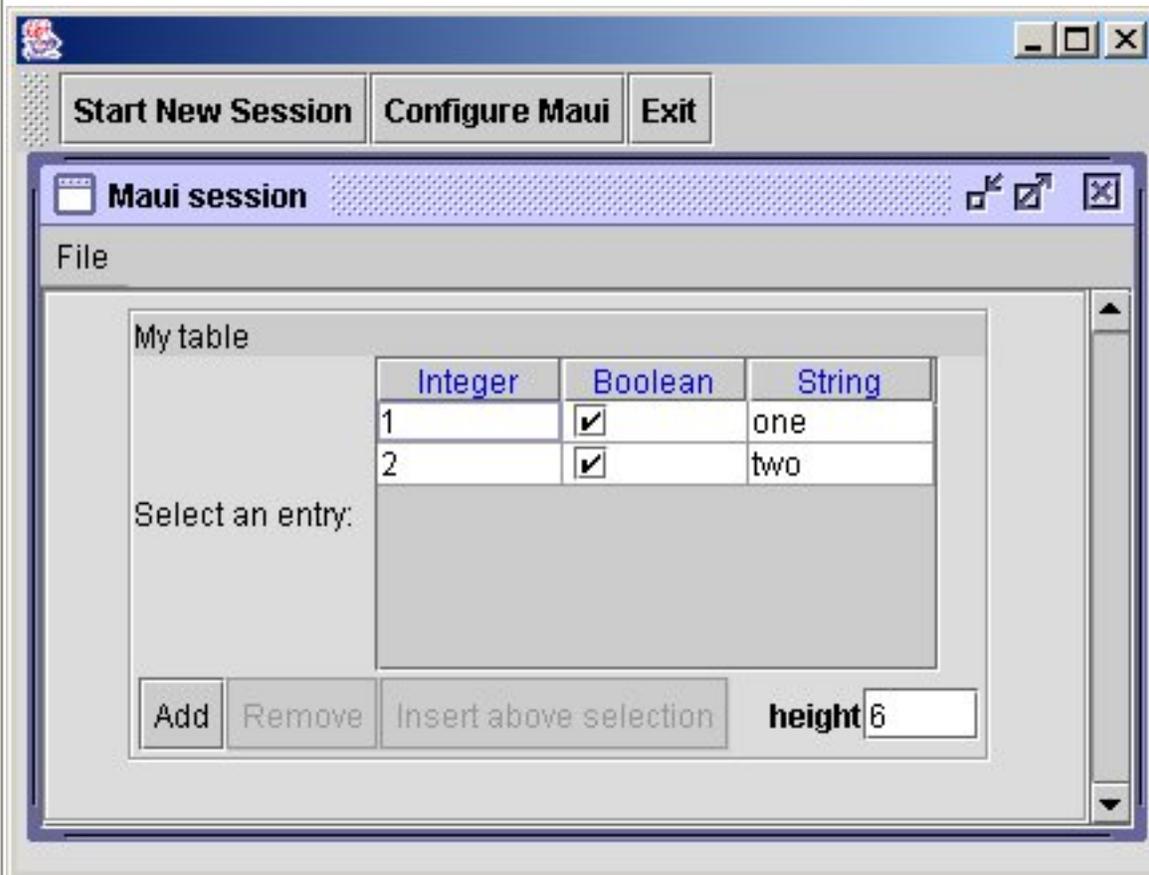
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">

```

```

        <Cell field="Col1" value="1" />
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="one" />
    </Entry>
    <Entry name="entry2">
        <Cell field="Col1" value="2" />
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="two" />
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

```



<Reference> : Select one choice from a collection of values that the end-user filled in (array, table, etc.)

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>

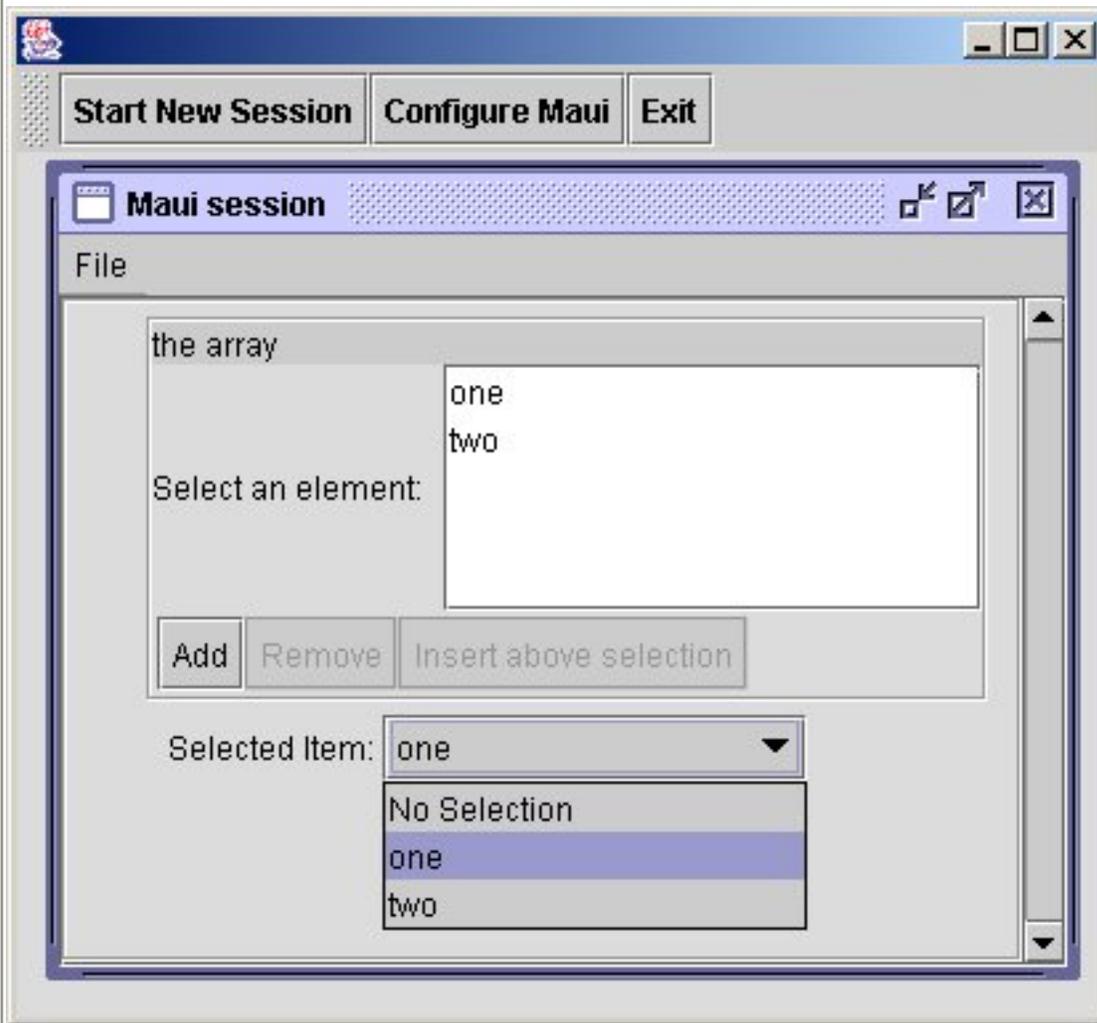
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label" />
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one" />
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two" />
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>

      <Reference name="myReference" path="myArray" />

    </Fields>
  </Class>

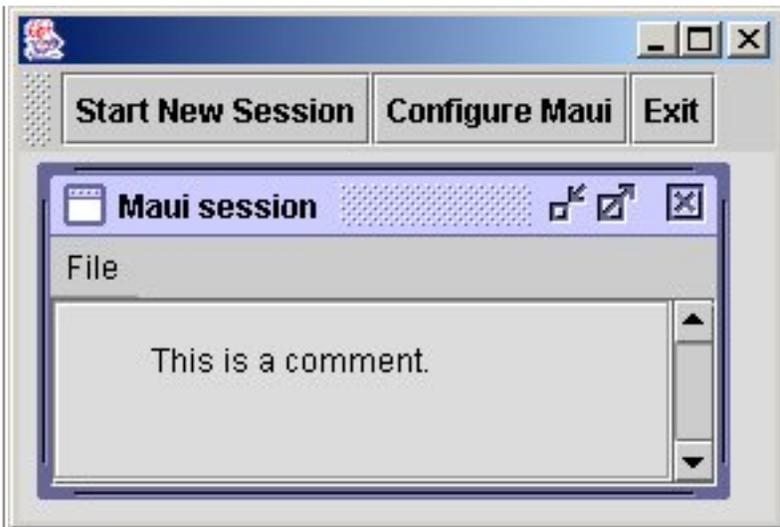
```

</Maui>



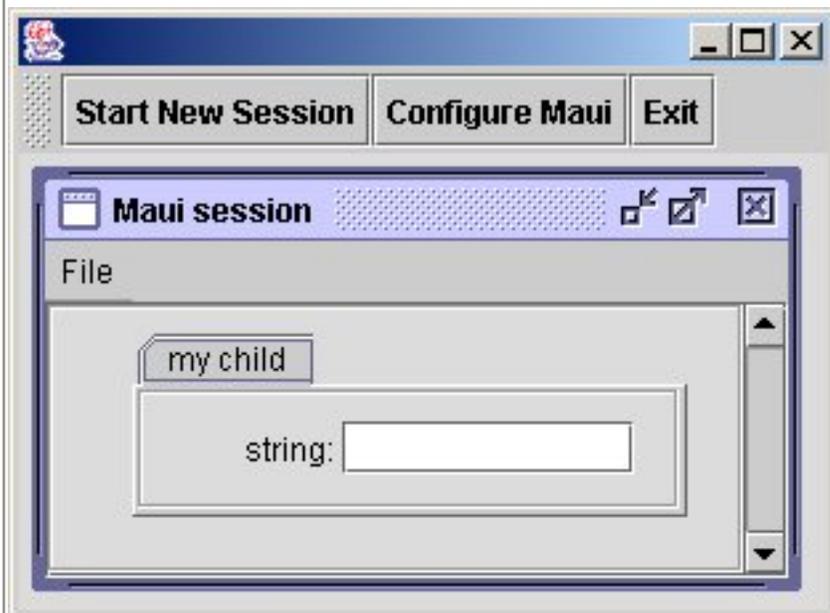
<Comment> : Display text on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Comment name="myComment">
        This is a comment.
      </Comment>
    </Fields>
  </Class>
</Maui>
```



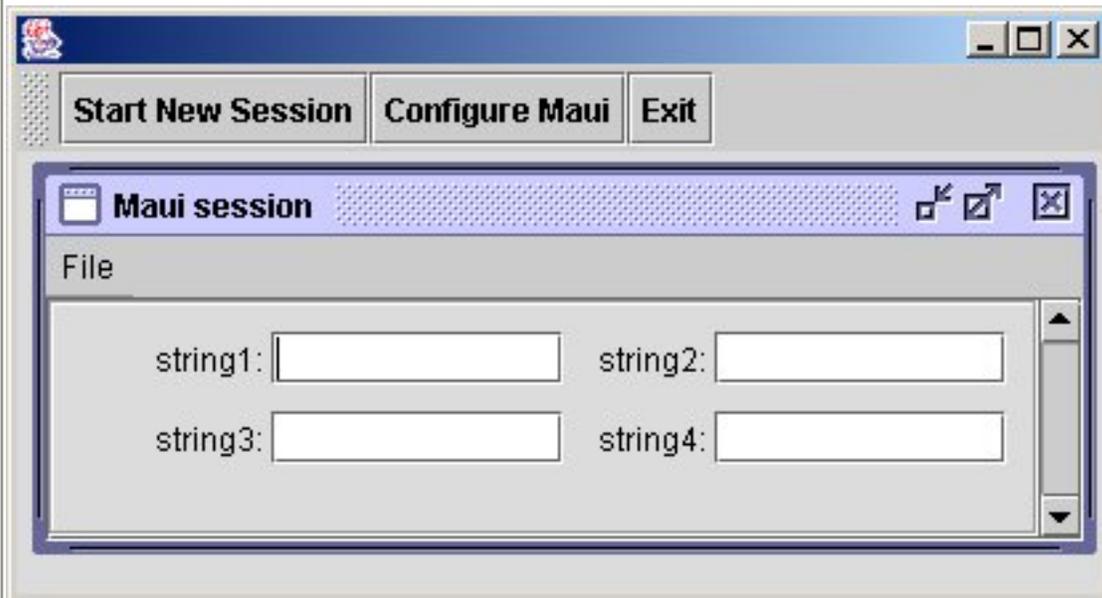
child class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"/>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



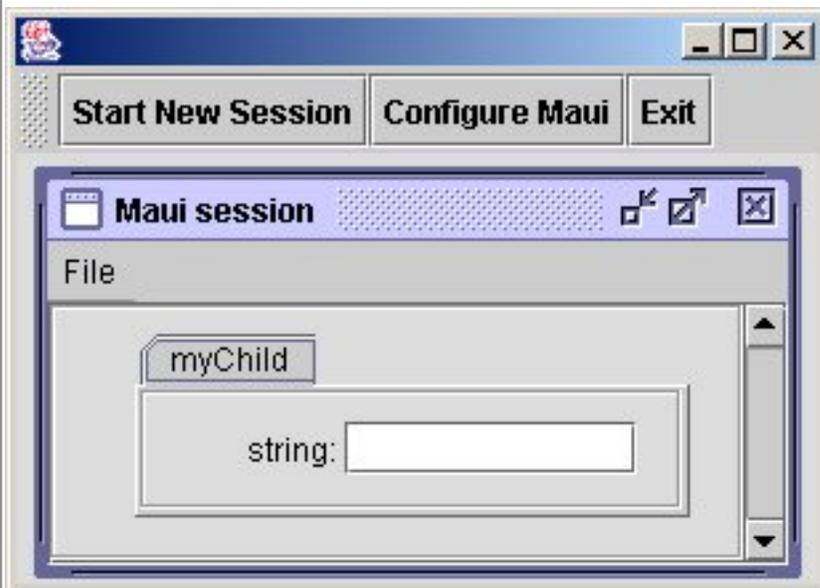
 : line break, skip to the next line

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" layout="flow">
    <Fields>
      <String name="string1" label="string1"/>
      <String name="string2" label="string2"/>
      <BR/>
      <String name="string3" label="string3"/>
      <String name="string4" label="string4"/>
    </Fields>
  </Class>
</Maui>
```



<Class> : Display a child class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Class type="myChild" label="myChild">
        <Fields>
          <String name="myString" label="string"/>
        </Fields>
      </Class>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.4.2 Attributes allowed in Up:](#) [B.4 Tag Fields](#) **Previous:** [B.4 Tag Fields](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.5 Tag AppData](#) **Up:** [B.4 Tag Fields](#) **Previous:** [B.4.1 Children allowed in](#)

B.4.2 Attributes allowed in Fields elements

None	
------	--

[Next](#) [Up](#) [Previous](#)

Next: [B.5 Tag AppData](#) **Up:** [B.4 Tag Fields](#) **Previous:** [B.4.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.5.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.4.2 Attributes allowed in](#)

B.5 Tag AppData

AppData is a free form block of XML that will be left untouched by Maui and passed on with the rest of the Maui XML to an action.

Subsections

- [B.5.1 Children allowed in AppData elements](#)
 - [B.5.2 Attributes allowed in AppData elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.5.2 Attributes allowed in](#) **Up:** [B.5 Tag AppData](#) **Previous:** [B.5 Tag AppData](#)

B.5.1 Children allowed in AppData elements

Any well formed XML [example](#)

`<AppData>` : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<AppData>  
  <parameter1>one</parameter1>  
  <parameter2>two</parameter2>  
</AppData>
```

[Next](#) [Up](#) [Previous](#)

Next: [B.5.2 Attributes allowed in](#) **Up:** [B.5 Tag AppData](#) **Previous:** [B.5 Tag AppData](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.6 Tag Action](#) **Up:** [B.5 Tag AppData](#) **Previous:** [B.5.1 Children allowed in](#)

B.5.2 Attributes allowed in AppData elements

None

[Next](#) [Up](#) [Previous](#)

Next: [B.6 Tag Action](#) **Up:** [B.5 Tag AppData](#) **Previous:** [B.5.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)**Next:** [B.6.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.5.2 Attributes allowed in](#)

B.6 Tag Action

`Action` will add a button to the class editor that will trigger a developer specified Action when pressed. The children of `Action` should contain any additional information that the Action may need to properly process the request.

Subsections

- [B.6.1 Children allowed in Action elements](#)
 - [B.6.2 Attributes allowed in Action elements](#)
-

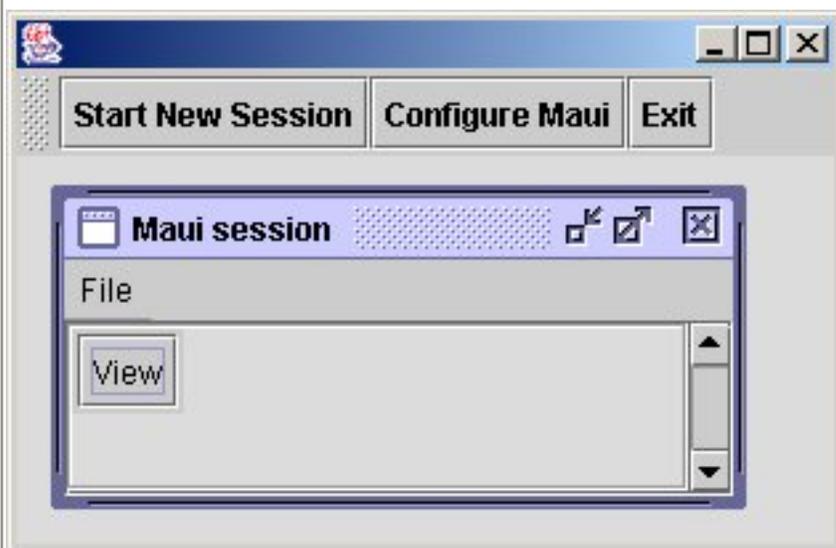
[Next](#) [Up](#) [Previous](#)**Next:** [B.6.2 Attributes allowed in](#) **Up:** [B.6 Tag Action](#) **Previous:** [B.6 Tag Action](#)

B.6.1 Children allowed in Action elements

Any well formed XML [example](#)

<Action> : Used to display a button

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="My container">
    <Action label="View" class="Maui.Interface.ViewAction"/>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.6.2 Attributes allowed in](#) **Up:** [B.6 Tag Action](#) **Previous:** [B.6 Tag Action](#)

[Next](#) [Up](#) [Previous](#)
Next: [B.7 Tag CustomEditor](#) **Up:** [B.6 Tag Action](#) **Previous:** [B.6.1 Children allowed in](#)

B.6.2 Attributes allowed in Action elements

Attribute name	Mandatory	Allowed values	Description and comments	Examples
label	yes	any string	The label to print on the face of the button	example
class	yes	the action classname	The name of the Action class to create a button for.	example
verbose	no	true or false	The Action will receive the complete verbose XML description of the Maui contents if verbose is true, otherwise, the compressed XML will be received. By default verbose is assumed to be false, and the compressed XML will be used.	example
path	no	path to the class	The full path to the class file of the Action used	example
package	no	package name	The package name of the Action class to load.	example
toolTip	no	any string	A tool tip will be displayed when the cursor is placed over the Action Button. This string will give the text to display in that tool tip. By default the label of the action will be used for the tool tip.	example
mode	no	verbose, compressed, or tree	This attribute pertains only to a ViewAction. If mode is set, then the specified display style (verbose, compressed, or tree) will be used when the ViewAction button is pressed. By default the display type is compressed.	example(verbose) example(compressed) example(tree)

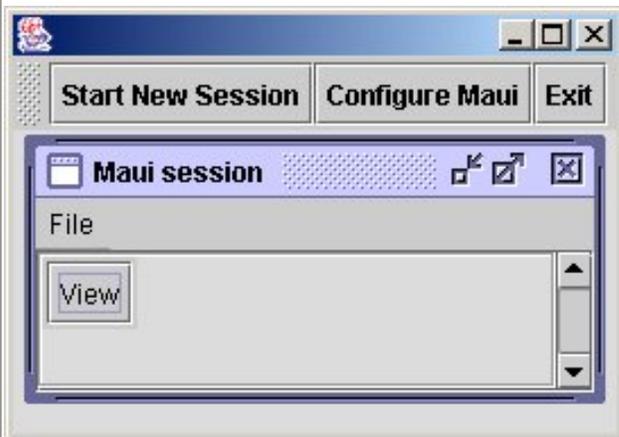
outputFormat	no	verbose, compressed, or input	When the end-user presses the button, xml data is sent to the Java code. The xml data can be in one of 3 formats. Data saved with "input" format can be feed back into Maui as an input deck. Data saved with "verbose" format can easily be reformatted into the input deck of an external application. "Compressed" data is small enough to fit on the screen for easy viewing.	example(verbose) example(compressed) example(input)
--------------	----	-------------------------------	---	---

<Action> : Used to display a button

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="My container">
    <Action label="View" class="Maui.Interface.ViewAction"/>
  </Class>
</Maui>
```

label: The text that appears inside the button

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction"/>
  </Class>
</Maui>
```

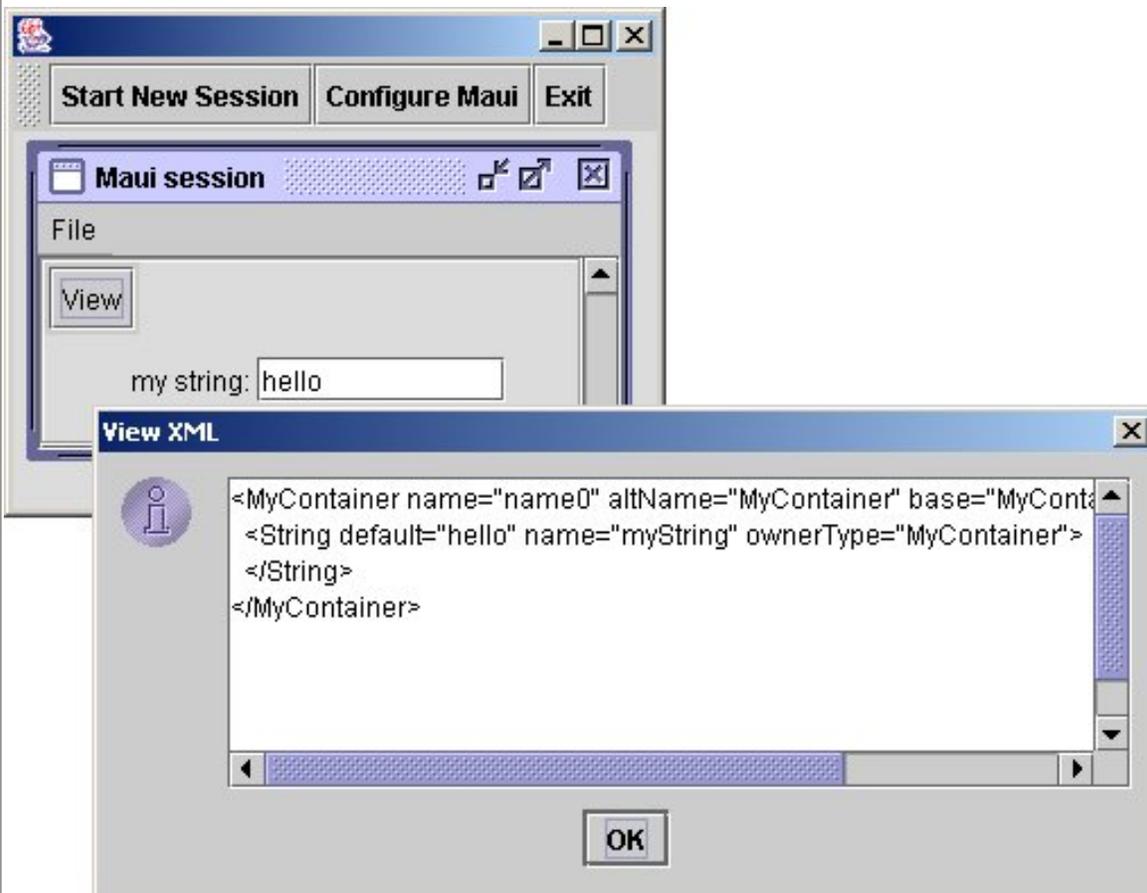


class: the java class that will be invoked when the end-user clicks on the button

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction"/>
  </Class>
</Maui>
```

verbose: When the end-use presses the button, xml data is sent to your java code. If verbose is true then then the xml data is in a verbose format. If verbose is false then the xml data is in a compressed format.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction" verbose="true"/>
    <Fields>
      <String name="myString" label="my string" default="hello"/>
    </Fields>
  </Class>
</Maui>
```



path: the path to the Java code (i.e. the location of the folder that contains the java code)

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View"
      path="c:/myJavaCode"
      class="Maui.Interface.ViewAction"/>
  </Class>
</Maui>
```

package: The package where your java code is located

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" package="Maui.Interface" class="ViewAction"/>
  </Class>
</Maui>
```

tooltip: not yet implements

mode: The ViewAction can display the xml data in one of 3 different modes:
verbose, compressed, tree

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction" mode="verbose"/>
    <Fields>
      <String name="myString" label="my string" default="hello"/>
    </Fields>
  </Class>
</Maui>
```



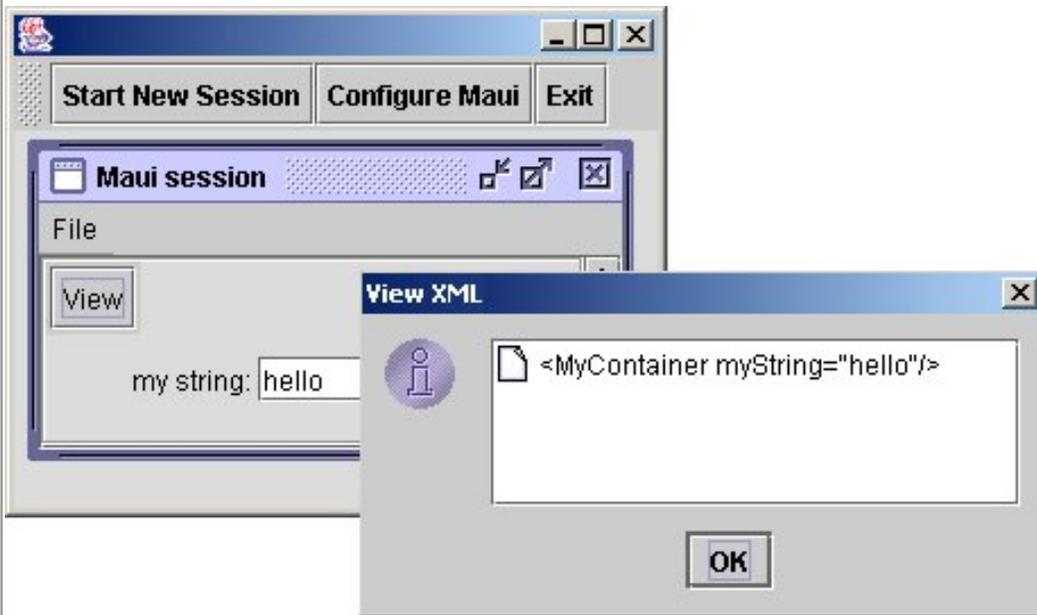
mode: The ViewAction can display the xml data in one of 3 different modes:
verbose, compressed, tree

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction" mode="compressed"/>
    <Fields>
      <String name="myString" label="my string" default="hello"/>
    </Fields>
  </Class>
</Maui>
```



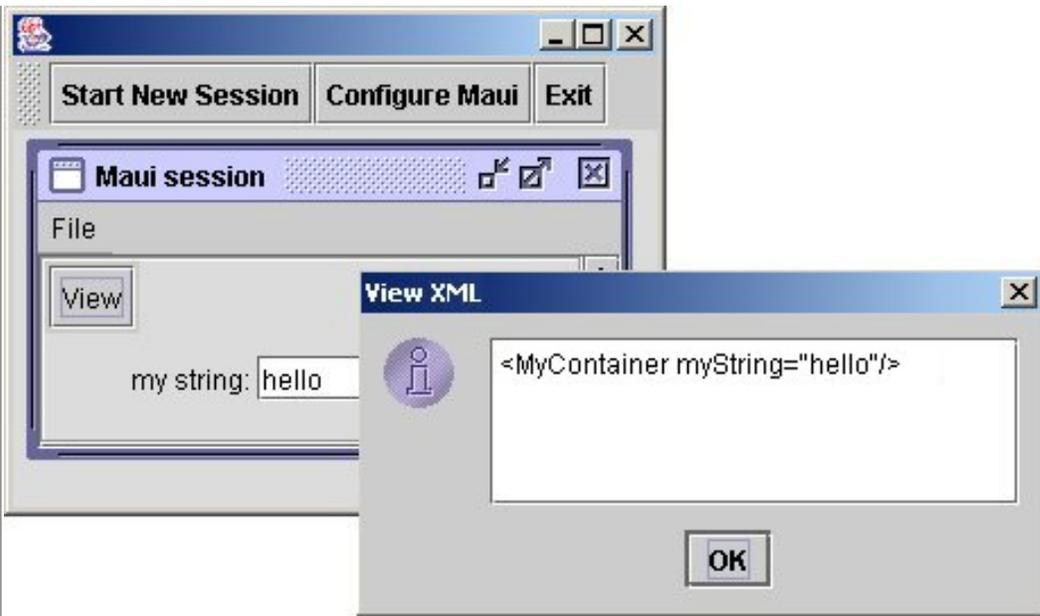
mode: The ViewAction can display the xml data in one of 3 different modes:
verbose, compressed, tree

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction" mode="tree"/>
    <Fields>
      <String name="myString" label="my string" default="hello"/>
    </Fields>
  </Class>
</Maui>
```



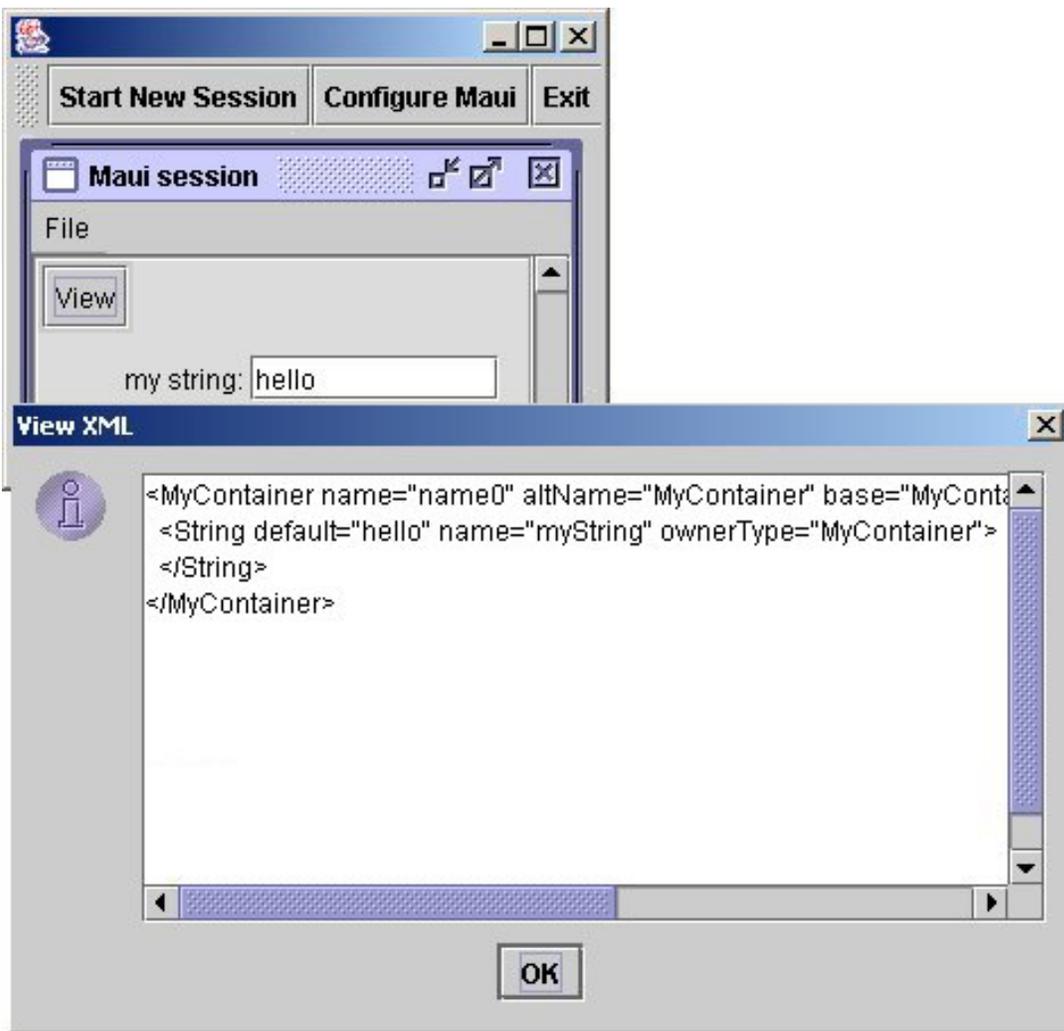
outputFormat: The format of the xml data that is sent to the java code.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction"
      outputFormat="compressed" />
    <Fields>
      <String name="myString" label="my string" default="hello"/>
    </Fields>
  </Class>
</Maui>
```



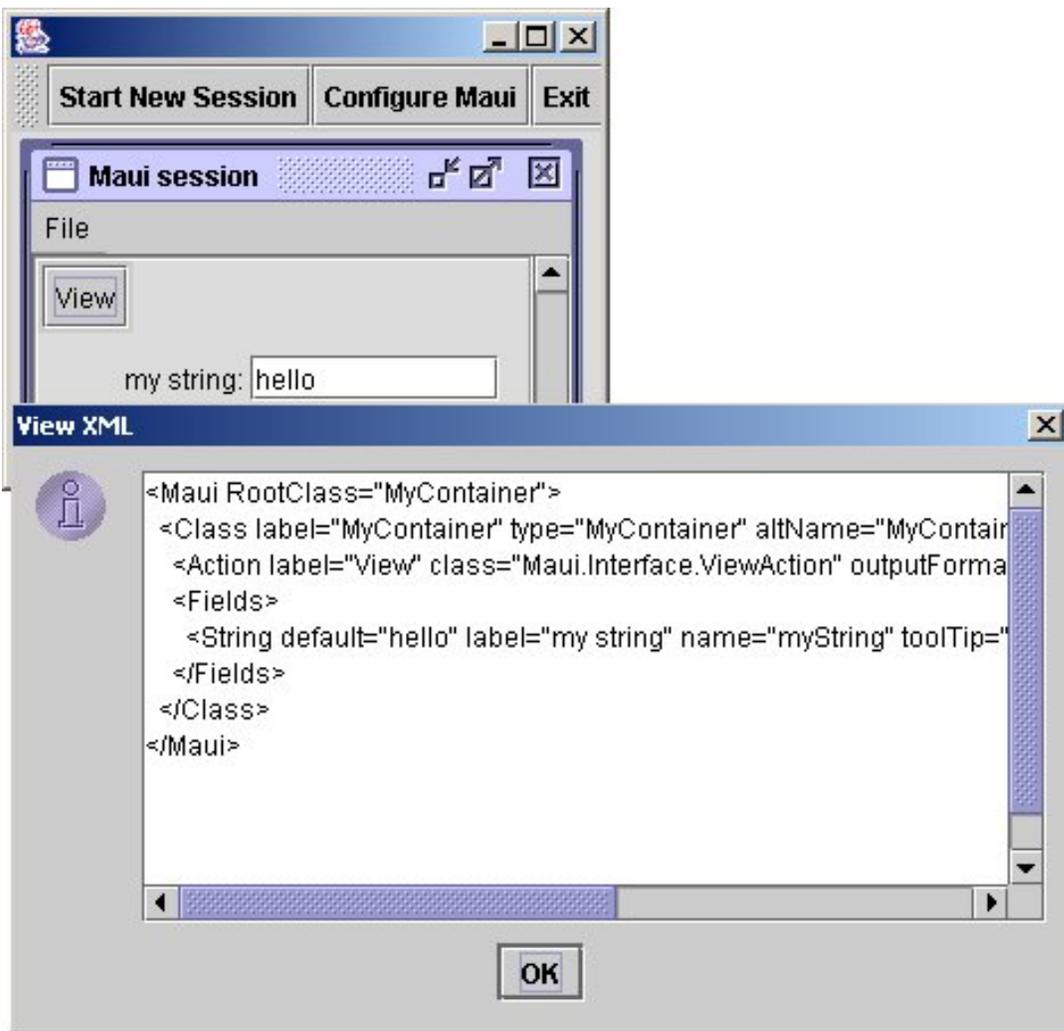
outputFormat: The format of the xml data that is sent to the java code.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction"
      outputFormat="verbose"/>
    <Fields>
      <String name="myString" label="my string" default="hello"/>
    </Fields>
  </Class>
</Maui>
```



outputFormat: The format of the xml data that is sent to the java code.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Action label="View" class="Maui.Interface.ViewAction"
      outputFormat="input"/>
    <Fields>
      <String name="myString" label="my string" default="hello"/>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.7 Tag CustomEditor](#) **Up:** [B.6 Tag Action](#) **Previous:** [B.6.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)**Next:** [B.7.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.6.2 Attributes allowed in](#)

B.7 Tag CustomEditor

CustomEditor elements allow the developer to add their own GUI components to Maui. The children of CustomEditor should contain any additional information that the Editor may need to display properly.

Subsections

- [B.7.1 Children allowed in CustomEditor elements](#)
 - [B.7.2 Attributes allowed in CustomEditor elements](#)
-

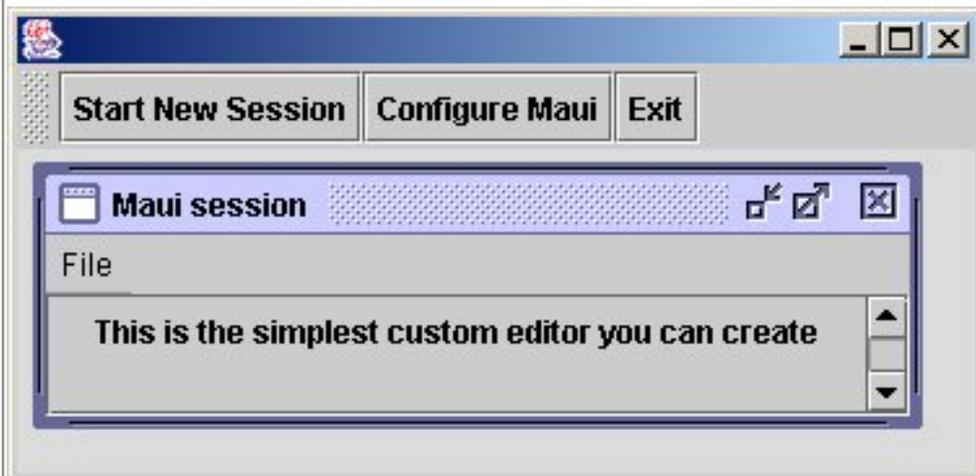
[Next](#) [Up](#) [Previous](#)**Next:** [B.7.2 Attributes allowed in Up](#) **Up:** [B.7 Tag CustomEditor](#) **Previous:** [B.7 Tag CustomEditor](#)

B.7.1 Children allowed in CustomEditor elements

Any well formed XML

<CustomEditor> : Used to insert a custom editor into Maui

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <CustomEditor
      name="Maui.Editors.ExampleCustomEditor_BareBonesCustomEditor">
      <parameter1>one</parameter1>
      <parameter2>two</parameter2>
    </CustomEditor>
  </Class>
</Maui>
```

[Next](#) [Up](#) [Previous](#)

Next: [B.7.2 Attributes allowed in](#) **Up:** [B.7 Tag CustomEditor](#) **Previous:** [B.7 Tag CustomEditor](#)

Next Up Previous

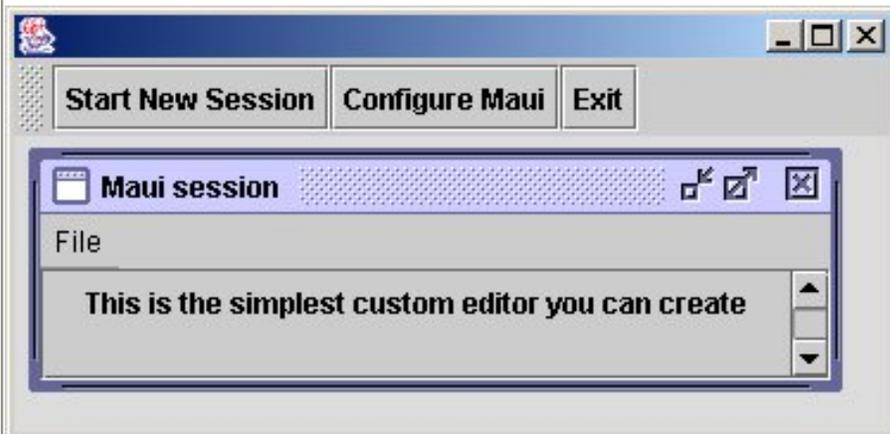
Next: [B.8 Tag Integer](#) Up: [B.7 Tag CustomEditor](#) Previous: [B.7.1 Children allowed in](#)

B.7.2 Attributes allowed in CustomEditor elements

Attribute name	Mandatory	Allowed values	Description and comments	examples
name	yes	the editor class name	The class name of the Editor to use for this Item.	example
path	no	path to the class	The full path to the class file of the Editor to use.	example
package	no	package name	The full package name of the Custom Editor.	example

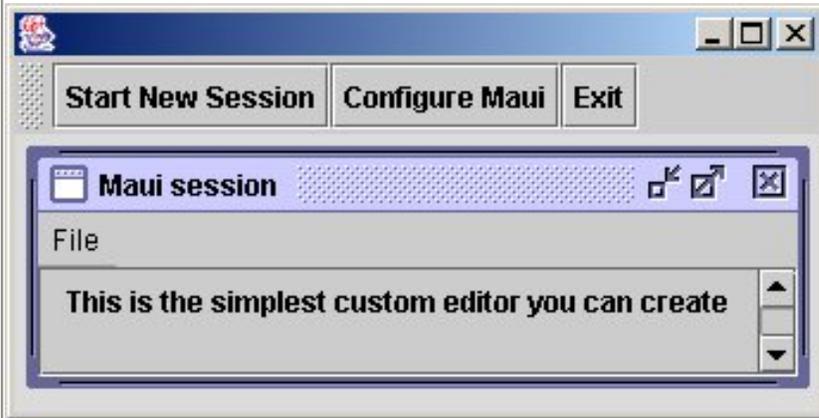
<CustomEditor> : Used to insert a custom editor into Maui

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <CustomEditor
      name="Maui.Editors.ExampleCustomEditor_BareBonesCustomEditor">
      <parameter1>one</parameter1>
      <parameter2>two</parameter2>
    </CustomEditor>
  </Class>
</Maui>
```



name: The name of the custom editor

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <CustomEditor
      name="Maui.Editors.ExampleCustomEditor_BareBonesCustomEditor"/>
  </Class>
</Maui>
```



path: The path to the java code (i.e. the folder that contains the java code).

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <CustomEditor path="c:/MyMauiEditors"
      name="Maui.Editors.ExampleCustomEditor_BareBonesCustomEditor"/>
  </Class>
</Maui>
```

package: the package that contains the java code

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <CustomEditor package="Maui.Editors"
      name="ExampleCustomEditor_BareBonesCustomEditor"/>
  </Class>
</Maui>
```

[Next](#) [Up](#) [Previous](#)

Next: [B.8 Tag Integer](#) **Up:** [B.7 Tag CustomEditor](#) **Previous:** [B.7.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.8.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.7.2 Attributes allowed in](#)

B.8 Tag Integer

Integer elements represent integer-valued variables.

Subsections

- [B.8.1 Children allowed in Integer elements](#)
 - [B.8.2 Attributes allowed in Integer elements](#)
-

[Next](#) [Up](#) [Previous](#)

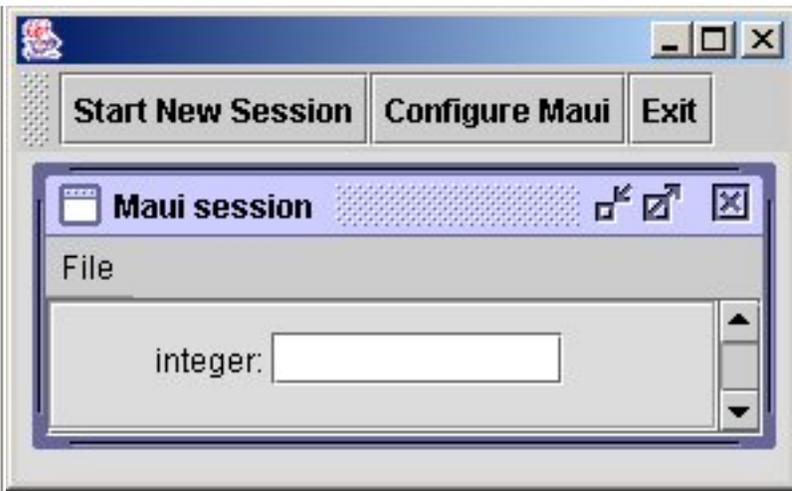
Next: [B.8.2 Attributes allowed in](#) **Up:** [B.8 Tag Integer](#) **Previous:** [B.8 Tag Integer](#)

B.8.1 Children allowed in Integer elements

Tag	Number	Description and comments	Examples
CustomEditor	0-1	Allows the Integer to use a non-standard editor.	example
AppData	0-1	a free form block of XML used for data where no editor is needed	example
Help	0-1	Display a help icon. If the end-user presses the help icon then pop up some useful information.	example

<Integer> : display a textbox that accepts integer values

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"/>
    </Fields>
  </Class>
</Maui>
```



<CustomEditor> : You can write you own customized editor.

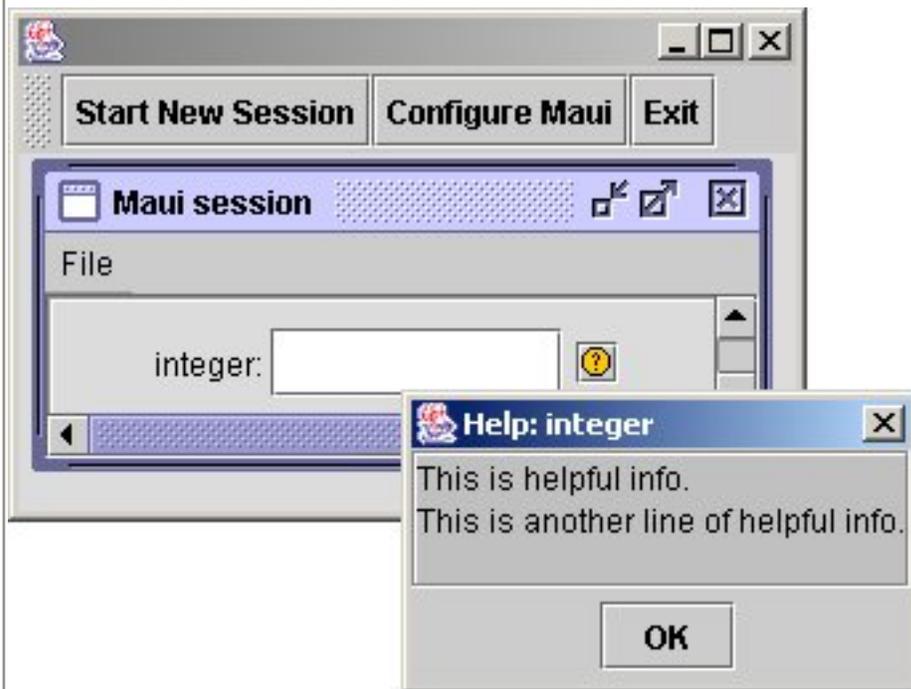
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInteger" label="integer">
        <CustomEditor name="MyIntegerEditor">
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </CustomEditor>
      </Integer>
    </Fields>
  </Class>
</Maui>
```

<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInteger" label="integer">
        <AppData>
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </AppData>
      </Integer>
    </Class>
</Maui>
```

<Help> : Display information that may be helpful to the end-user.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer">
        <Help>
          This is helpful info.
          This is another line of helpful info.
        </Help>
      </Integer>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.8.2 Attributes allowed in Up](#) **Up:** [B.8 Tag Integer](#) **Previous:** [B.8 Tag Integer](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.9 Tag Double](#) **Up:** [B.8 Tag Integer](#) **Previous:** [B.8.1 Children allowed in](#)

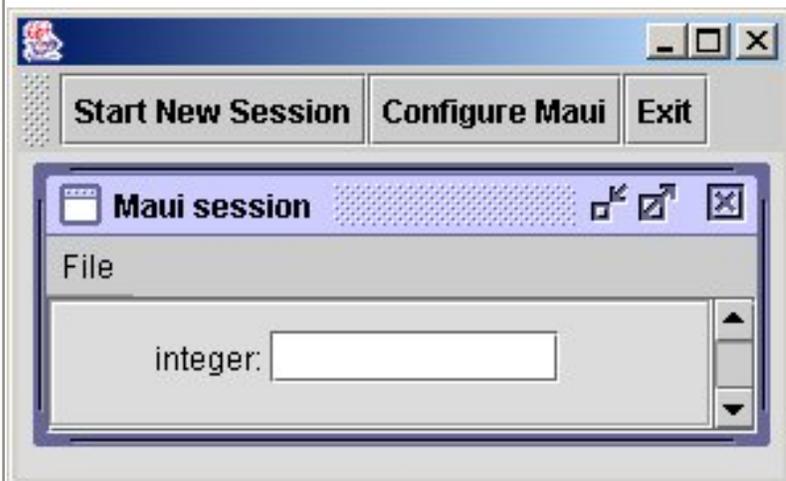
B.8.2 Attributes allowed in Integer elements

Attribute name	Mandatory	Allowed values	Description and comments	Examples
name	yes	any legal name	name for this variable	example
label	no	any string	string to be used as a descriptive label	example
default	no	any integer	default value of this variable	example
lower	no	any integer	the lower limit for integers that can be entered; must be less than upper	example
upper	no	any integer	the upper limit for integers that can be entered; must be greater than lower	example
optional	no	true/false	indicates if a value needs to be filled in	example
editable	no	true/false	indicates if the value can be edited	example
columnWidth	no	an integer	gives the number of characters that should be displayed in the text field.	example
tooltip	no	any string	When the end-user moves the mouse over the label then display a tooltip.	example

helpMessage	no	any string	Display a HELP icon. Whenever the end-user clicks on the icon, a helpful message pops up on the screen.	example
visible	no	true or false	Is the textbox visible on the screen.	example
mauiAction	no	name of a MauiAction.	This java class is invoked whenever the end-user changes the contents of the textbox.	example
insets	no	true or false	If true then surround the GUI component with a lot of white space.	example

<Integer> : display a textbox that accepts integer values

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"/>
    </Fields>
  </Class>
</Maui>
```

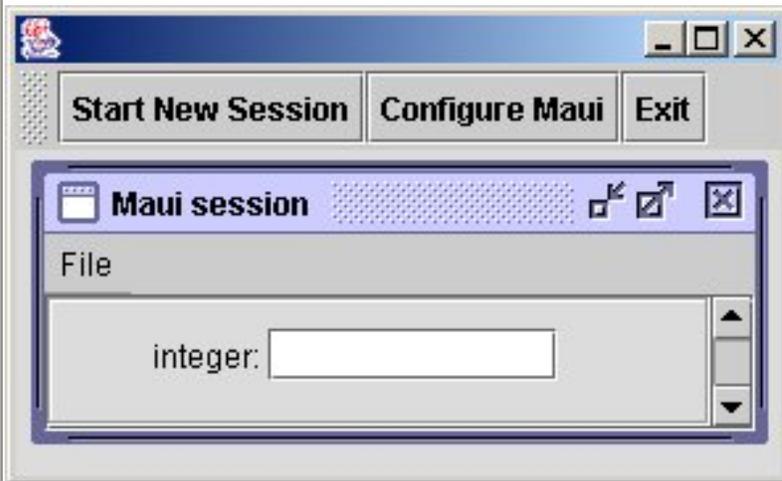


name: the name of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"/>
    </Fields>
  </Class>
</Maui>
```

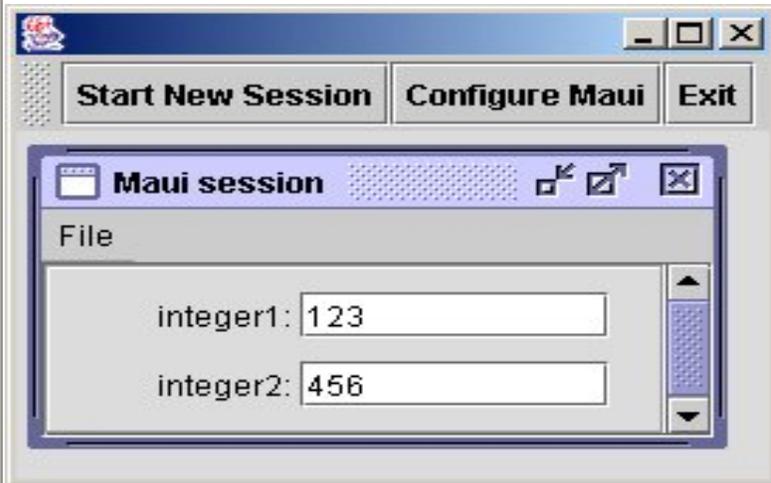
label: The text that is displayed to the left of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"/>
    </Fields>
  </Class>
</Maui>
```



default: The value that is displayed inside the textbox, when the textbox first appears on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer" default="123"/>
    </Fields>
  </Class>
</Maui>
```



lower: The smallest number the end-user is allowed to type into the textbox.

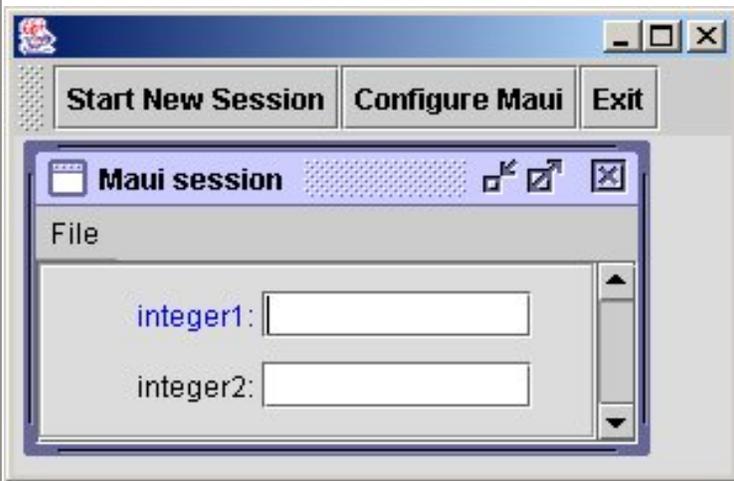
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer" lower="0"/>
    </Fields>
  </Class>
</Maui>
```

upper: The largest number the end-user is allowed to type into the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer" upper="100"/>
    </Fields>
  </Class>
</Maui>
```

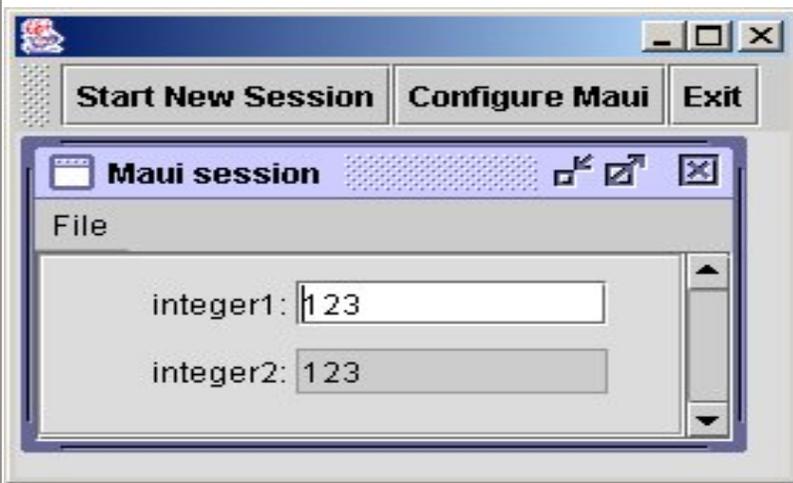
optional: If true then the end-user is not forced to insert a value into the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer" optional="true"/>
    </Fields>
  </Class>
</Maui>
```



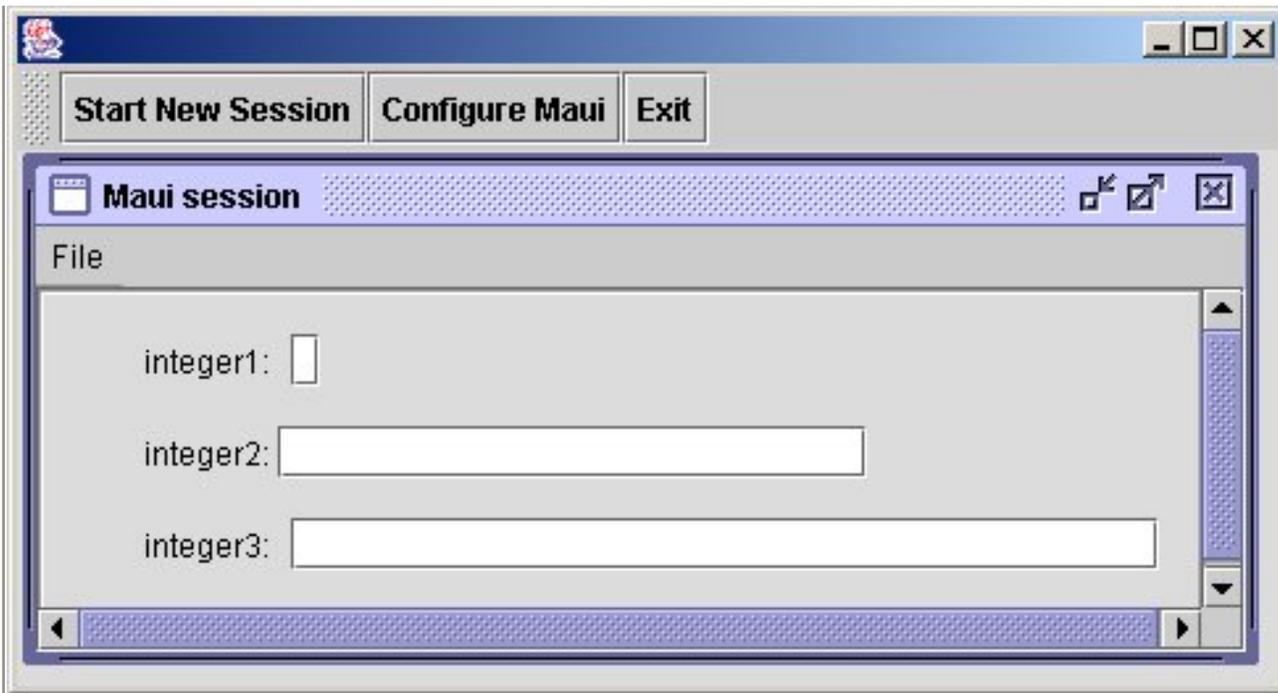
editable: If false then the end-user can not change the contents of the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt "
        label="integer"
        editable="false"
        default="123" />
    </Fields>
  </Class>
</Maui>
```



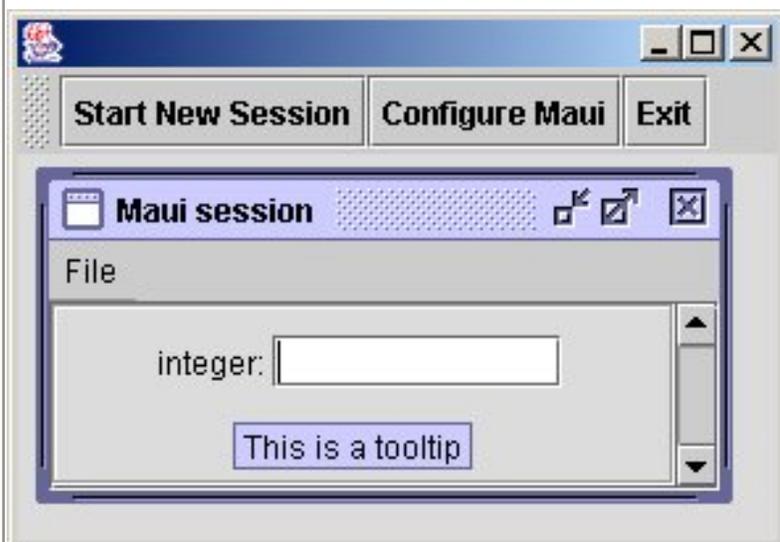
columnWidth: the number of characters that can fit inside the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer" columnWidth="3" />
    </Fields>
  </Class>
</Maui>
```



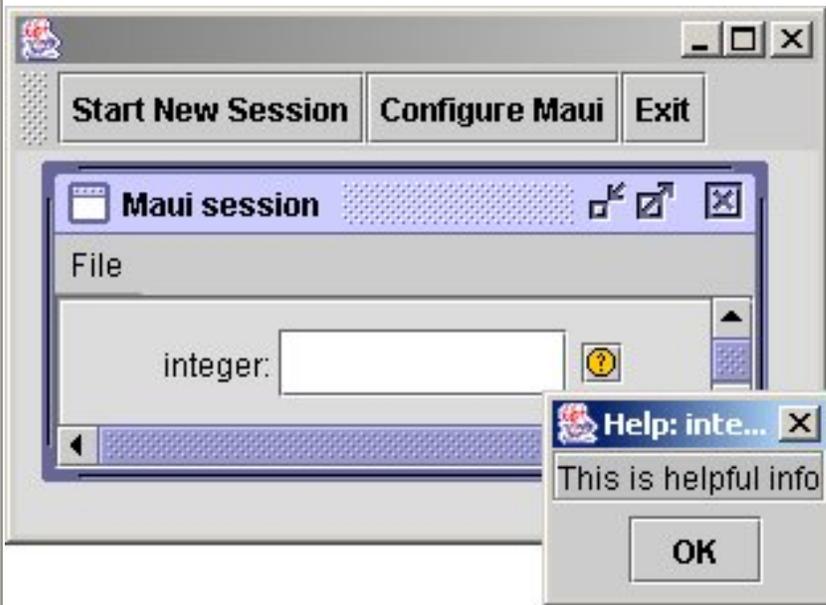
tooltip: Assign a tooltip to the label

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"
        tooltip="This is a tooltip"/>
    </Fields>
  </Class>
</Maui>
```



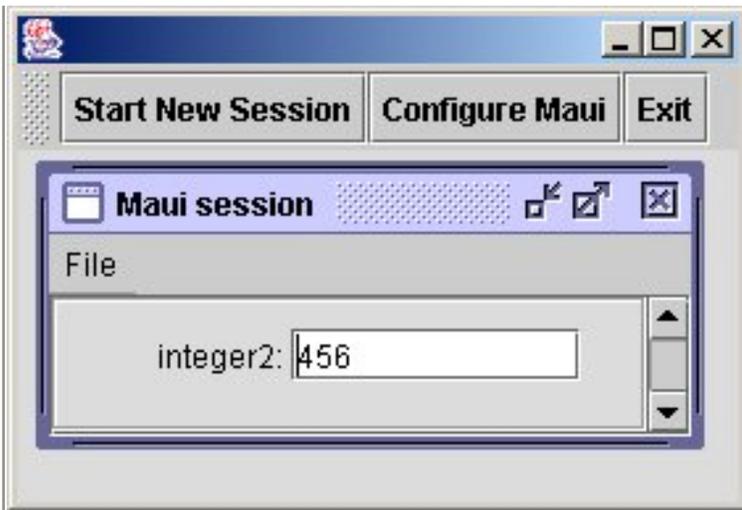
helpMessage: Display help

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="myInt" label="integer"
        helpMessage="This is helpful info"/>
    </Fields>
  </Class>
</Maui>
```



visible: Is the textbox visible on the screen?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="integer1" label="integer1"
        visible="false" default="123"/>
      <Integer name="integer2" label="integer2"
        visible="true" default="456"/>
    </Fields>
  </Class>
</Maui>
```

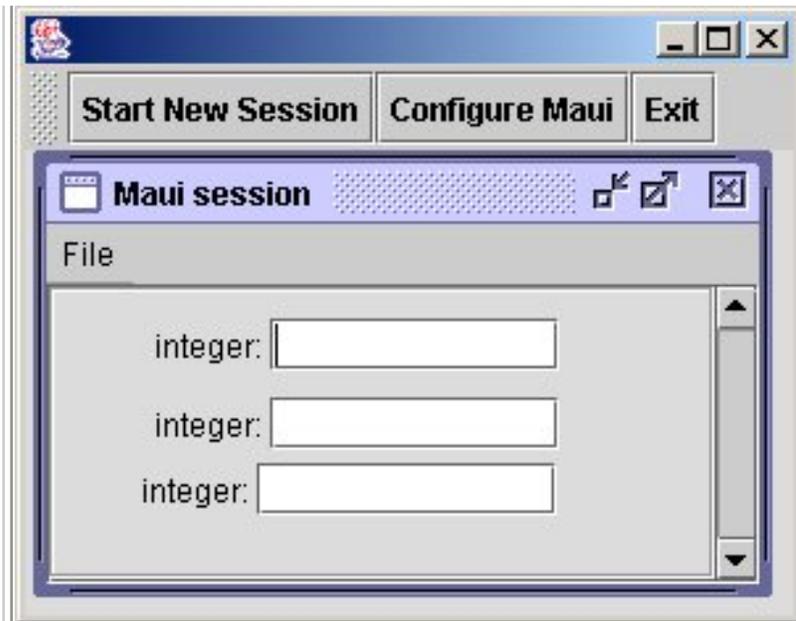


MauiAction: This Java class is invoked whenever the end-user changes the contents of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="integer1" label="integer1"
        mauiAction="MyMauiAction"/>
    </Fields>
  </Class>
</Maui>
```

insets: Is there a lot of white space surrounding the textbox?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Integer name="integer1" label="integer"/>
      <Integer name="integer2" label="integer" insets="true"/>
      <Integer name="integer3" label="integer" insets="false"/>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.9 Tag Double](#) **Up:** [B.8 Tag Integer](#) **Previous:** [B.8.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.9.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.8.2 Attributes allowed in](#)

B.9 Tag Double

Double elements represent double-precision-valued variables.

Subsections

- [B.9.1 Children allowed in Double elements](#)
 - [B.9.2 Attributes allowed in Double elements](#)
-

[Next](#)
[Up](#)
[Previous](#)

Next: [B.9.2 Attributes allowed in](#)
Up: [B.9 Tag Double](#)
Previous: [B.9 Tag Double](#)

B.9.1 Children allowed in Double elements

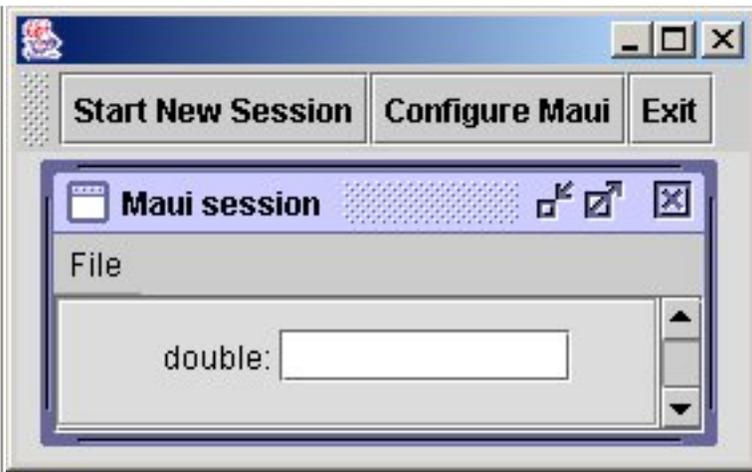
Tag	Number	Description and comments	Examples
CustomEditor	0-1	Allows the Double to use a non-standard editor.	example
AppData	0-1	a free form block of XML used for data where no editor is needed	example
Help	0-1	Display a help icon. If the end-user presses the help icon then pop up some useful information.	example

<Double> : display a textbox that accepts floating point numbers

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" />
    </Fields>
  </Class>
</Maui>

```



<CustomEditor> : You can write you own customized editor.

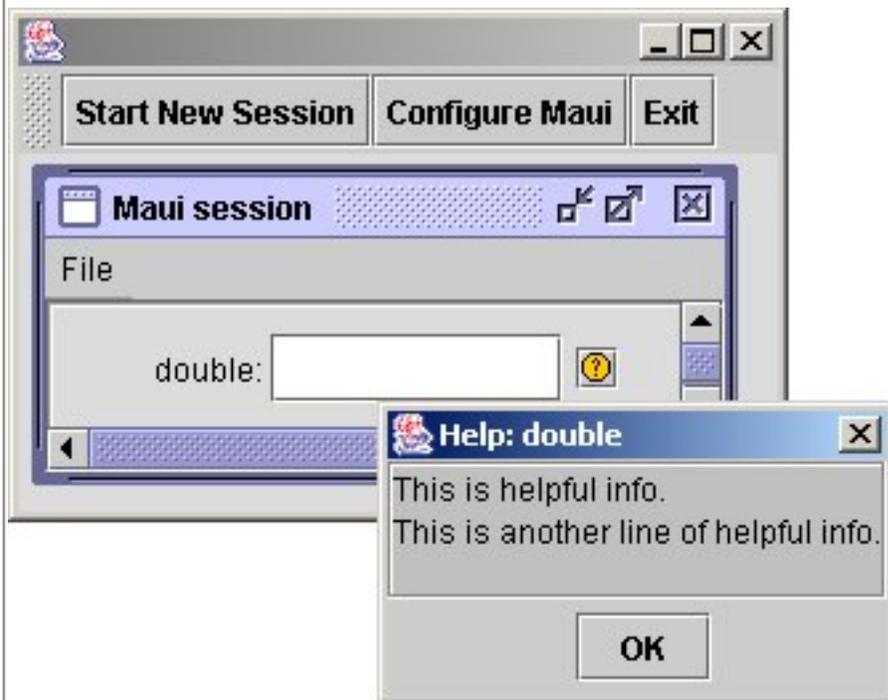
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double">
        <CustomEditor name="MyDoubleEditor">
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </CustomEditor>
      </Double>
    </Fields>
  </Class>
</Maui>
```

<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double">
        <AppData>
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </AppData>
      </Double>
    </Class>
</Maui>
```

<Help> : Display information that may be helpful to the end-user.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double">
        <Help>
          This is helpful info.
          This is another line of helpful info.
        </Help>
      </Double>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.9.2 Attributes allowed in Up](#) **Up:** [B.9 Tag Double](#) **Previous:** [B.9 Tag Double](#)

[Next](#) [Up](#) [Previous](#)

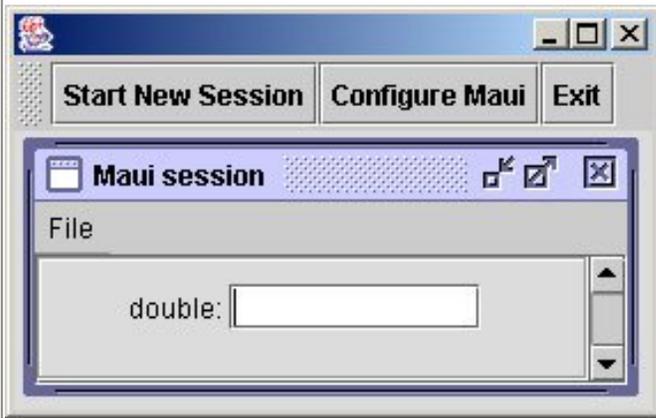
Next: [B.10 Tag Boolean](#) **Up:** [B.9 Tag Double](#) **Previous:** [B.9.1 Children allowed in](#)

B.9.2 Attributes allowed in Double elements

Attribute name	Mandatory	Allowed values	Description and comments	Examples
name	yes	any legal name	name for this variable	example
label	no	any string	string to be used as a descriptive label	example
default	no	any double	default value of this variable	example
lower	no	any double	the lower limit for values that can be entered; must be less than upper	example
upper	no	any double	the upper limit for values that can be entered; must be greater than lower	example
optional	no	true/false	indicates if a value needs to be filled in	example
editable	no	true/false	indicates if the value can be edited	example
columnWidth	no	an integer	gives the number of characters that should be displayed in the text field.	example
tooltip	no	any string	When the end-user moves the mouse over the label then display a tooltip.	example
helpMessage	no	any string	Display a HELP icon. Whenever the end-user clicks on the icon, a helpful message pops up on the screen.	example
visible	no	true or false	Is the textbox visible on the screen.	example
mauiAction	no	name of a MauiAction.	This java class is invoked whenever the end-user changes the contents of the textbox.	example
insets	no	true or false	If true then surround the GUI component with a lot of white space.	example

<Double> : display a textbox that accepts double values

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" />
    </Fields>
  </Class>
</Maui>
```

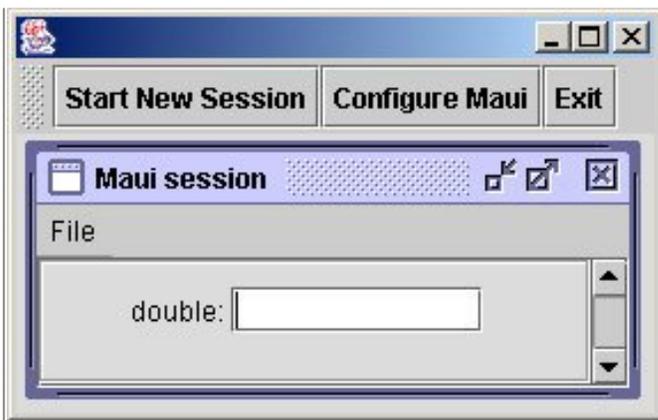


name: the name of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" />
    </Fields>
  </Class>
</Maui>
```

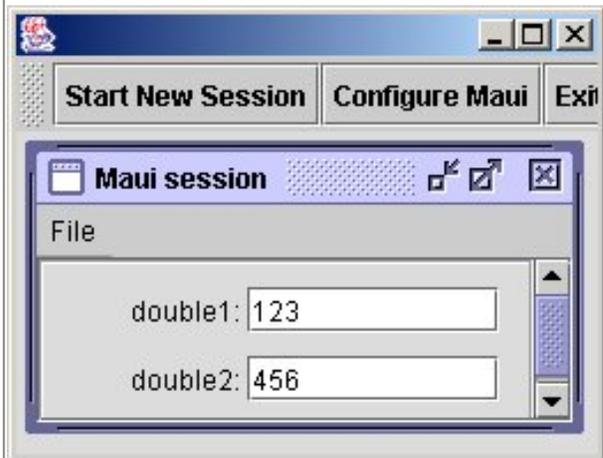
label: The text that is displayed to the left of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" />
    </Fields>
  </Class>
</Maui>
```



default: The value that is displayed inside the textbox, when the textbox first appears on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="double1" label="double1" default="123"/>
      <Double name="double2" label="double2" default="456"/>
    </Fields>
  </Class>
</Maui>
```



lower: The smallest number the end-user is allowed to type into the textbox.

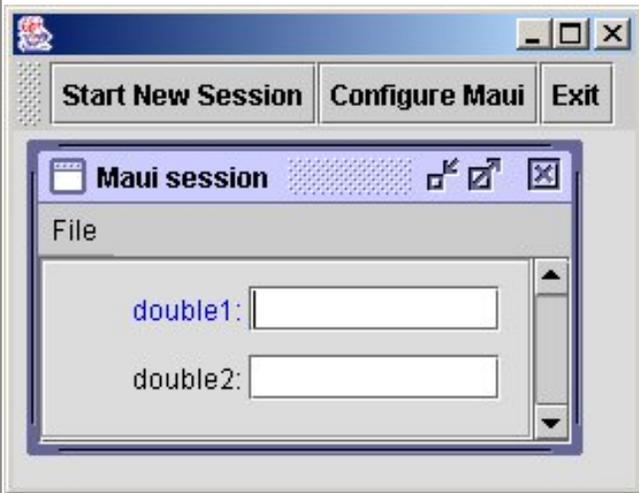
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" lower="0"/>
    </Fields>
  </Class>
</Maui>
```

upper: The largest number the end-user is allowed to type into the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" upper="100"/>
    </Fields>
  </Class>
</Maui>
```

optional: If true then the end-user is not forced to insert a value into the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="double1" label="double1" optional="true"/>
      <Double name="double2" label="double2" optional="false"/>
    </Fields>
  </Class>
</Maui>
```



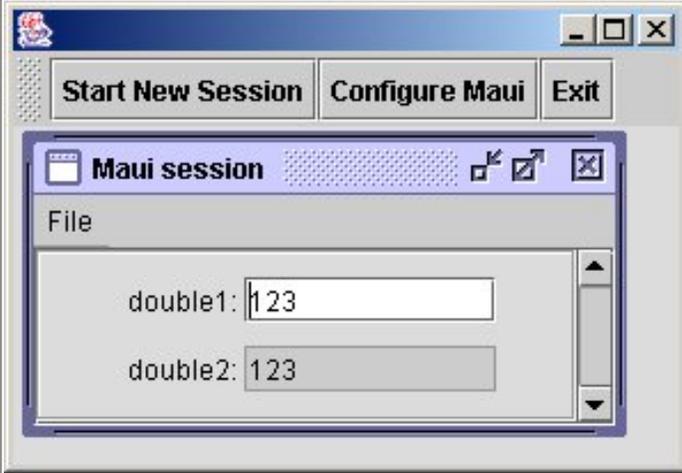
editable: If false then the end-user can not change the contents of the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="double1"
        label="double1"
        editable="true"
        default="123" />
      <Double name="double2"
        label="double2"
        editable="false" />
    </Fields>
  </Class>
</Maui>
```

```

        default="123"/>
    </Fields>
</Class>
</Maui>

```

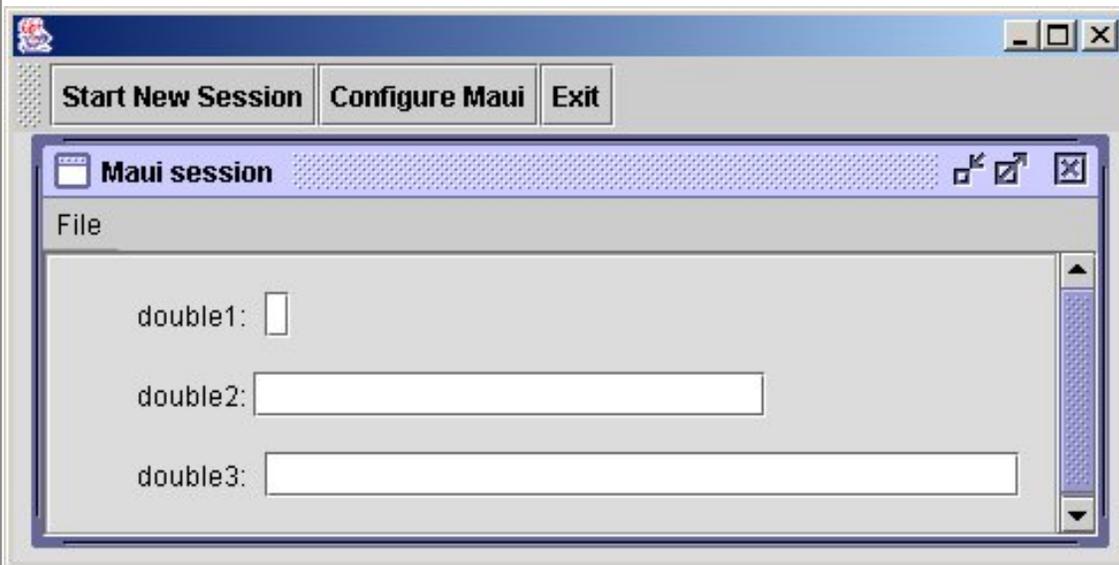


columnWidth: the number of characters that can fit inside the textbox

```

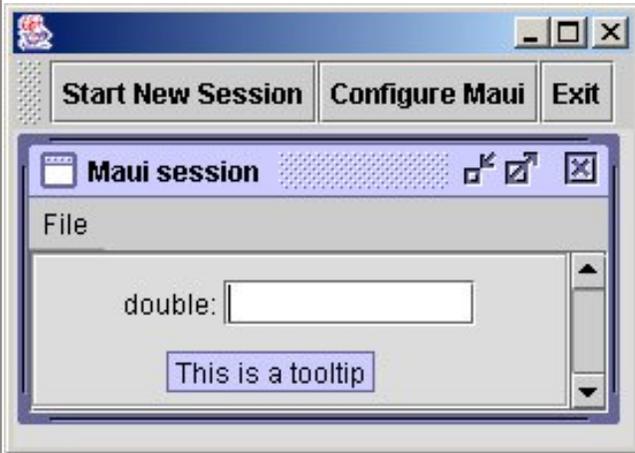
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="double1" label="double1" columnWidth="1"/>
      <Double name="double2" label="double2"/>
      <Double name="double3" label="double3" columnWidth="30"/>
    </Fields>
  </Class>
</Maui>

```



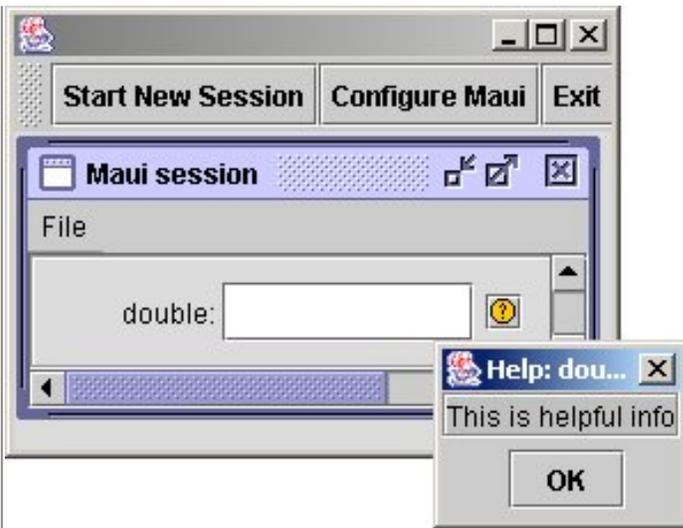
tooltip: Assign a tooltip to the label

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double" toolTip="This is a tooltip"/>
    </Fields>
  </Class>
</Maui>
```



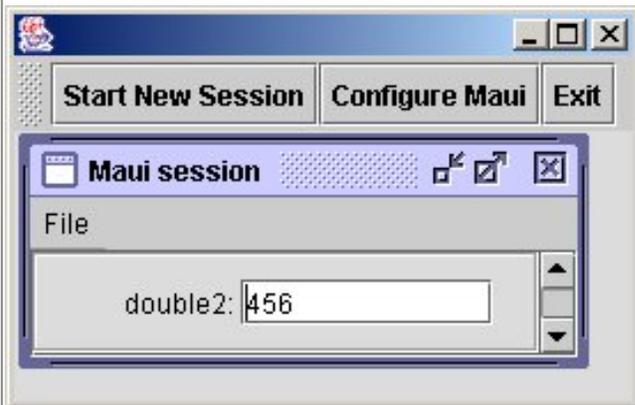
helpMessage: Display help

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="myDouble" label="double"
        helpMessage="This is helpful info"/>
    </Fields>
  </Class>
</Maui>
```



visible: Is the textbox visible on the screen?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="double1" label="double1"
        visible="false" default="123"/>
      <Double name="double2" label="double2"
        visible="true" default="456"/>
    </Fields>
  </Class>
</Maui>
```

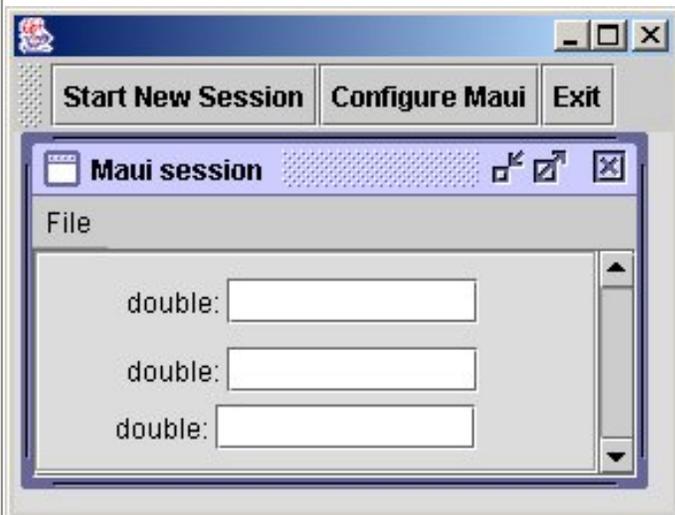


MauiAction: This Java class is invoked whenever the end-user changes the contents of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="double1" label="double1"
        mauiAction="MyMauiAction" />
    </Fields>
  </Class>
</Maui>
```

insets: Is there a lot of white space surrounding the textbox?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Double name="double1" label="double" />
      <Double name="double2" label="double" insets="true" />
      <Double name="double3" label="double" insets="false" />
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.10 Tag Boolean](#) **Up:** [B.9 Tag Double](#) **Previous:** [B.9.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.10.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.9.2 Attributes allowed in](#)

B.10 Tag Boolean

Boolean elements represent boolean variables.

Subsections

- [B.10.1 Children allowed in Boolean elements](#)
 - [B.10.2 Attributes allowed in Boolean elements](#)
-

[Next](#)
[Up](#)
[Previous](#)

Next: [B.10.2 Attributes allowed in](#)
Up: [B.10 Tag Boolean](#)
Previous: [B.10 Tag Boolean](#)

B.10.1 Children allowed in Boolean elements

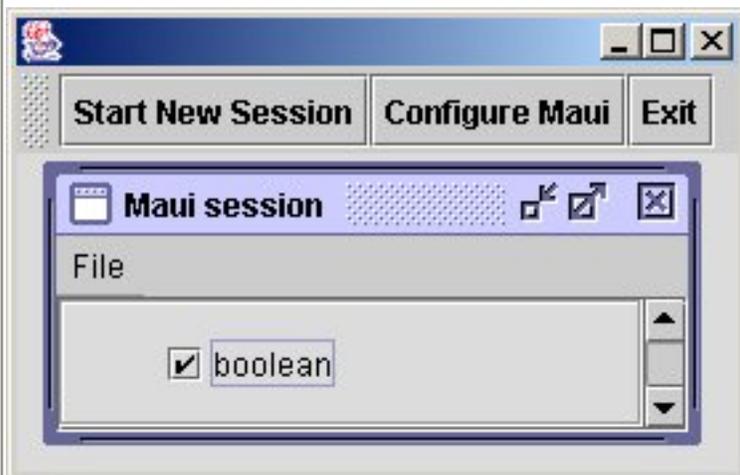
Tag	Number	Description and comments	Examples
CustomEditor	0-1	Allows the Boolean to use a non-standard editor.	example
AppData	0-1	a free form block of XML used for data where no editor is needed	example
Help	0-1	Display a help icon. If the end-user presses the help icon then pop up some useful information.	example

<Boolean> : display a checkbox

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean"/>
    </Fields>
  </Class>
</Maui>

```



<CustomEditor> : You can write you own customized editor.

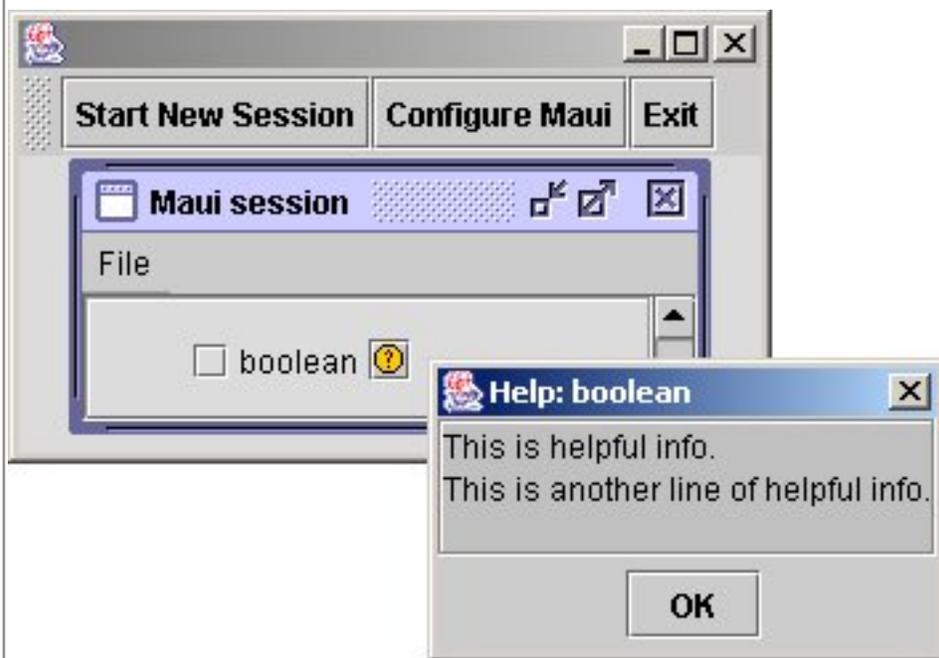
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean">
        <CustomEditor name="MyBooleanEditor">
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </CustomEditor>
      </Boolean>
    </Fields>
  </Class>
</Maui>
```

<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean">
        <AppData>
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </AppData>
      </Boolean>
    </Class>
</Maui>
```

<Help> : Display information that may be helpful to the end-user.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean">
        <Help>
          This is helpful info.
          This is another line of helpful info.
        </Help>
      </Boolean>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.10.2 Attributes allowed in](#) **Up:** [B.10 Tag Boolean](#) **Previous:** [B.10 Tag Boolean](#)

[Next](#) [Up](#) [Previous](#)

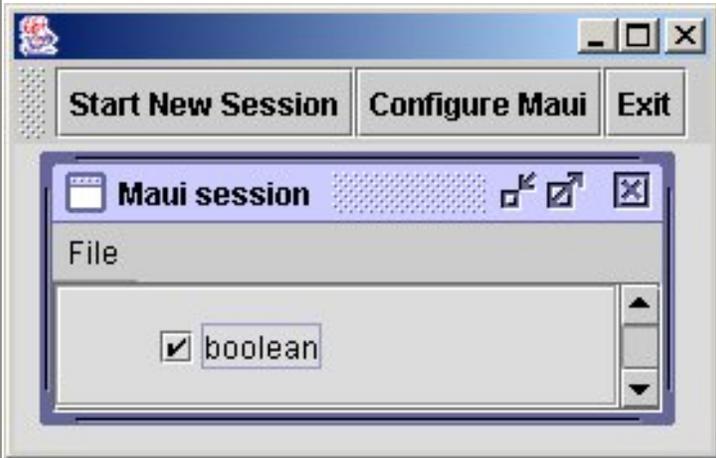
Next: [B.11 Tag String](#) **Up:** [B.10 Tag Boolean](#) **Previous:** [B.10.1 Children allowed in](#)

B.10.2 Attributes allowed in Boolean elements

Attribute name	Mandatory	Allowed values	Description and comments	Examples
name	yes	any legal name	name for this variable	example
label	no	any string	string to be used as a descriptive label	example
default	no	true/false	default value of this variable	example
optional	no	true/false	indicates if a value needs to be filled in	example
editable	no	true/false	indicates if the value can be edited	example
tooltip	no	any string	When the end-user moves the mouse over the label then display a tooltip.	example
helpMessage	no	any string	Display a HELP icon. Whenever the end-user clicks on the icon, a helpful message pops up on the screen.	example
visible	no	true or false	Is the checkbox visible on the screen.	example
insets	no	true or false	If true then surround the GUI component with a lot of white space.	example

<Boolean> : display a checkbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean"/>
    </Fields>
  </Class>
</Maui>
```

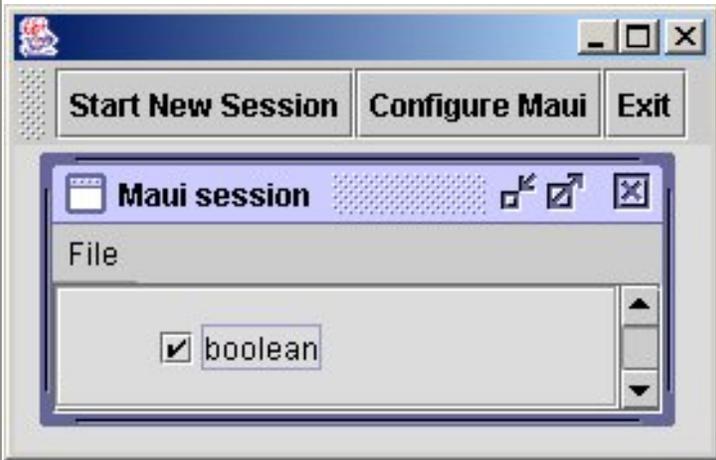


name: the name of the checkbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean"/>
    </Fields>
  </Class>
</Maui>
```

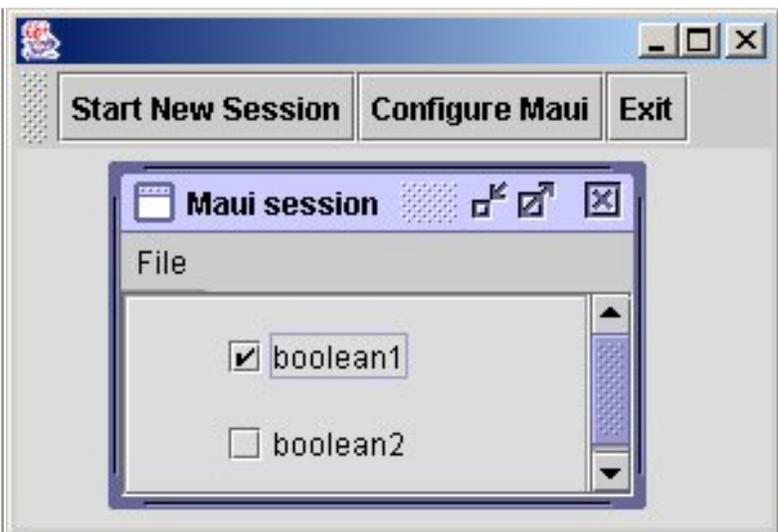
label: The text that is displayed adjacent to the checkbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean"/>
    </Fields>
  </Class>
</Maui>
```



default: The checked/unchecked state of the checkbox, when the checkbox first appears on the screen

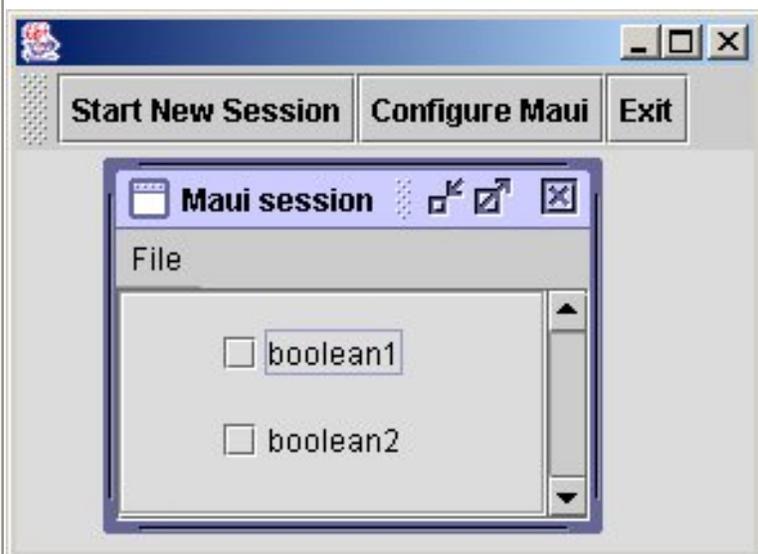
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="boolean1" label="boolean1" default="true"/>
      <Boolean name="boolean2" label="boolean2" default="false"/>
    </Fields>
  </Class>
</Maui>
```



optional: If true then the end-user is not forced to insert a value into the checkbox.

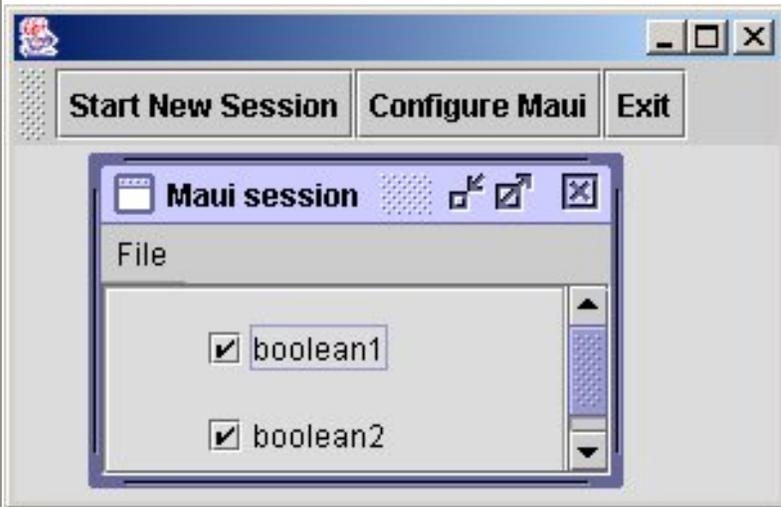
```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="boolean1" label="boolean1" optional="true"/>
      <Boolean name="boolean2" label="boolean2" optional="false"/>
    </Fields>
  </Class>
</Maui>
  
```



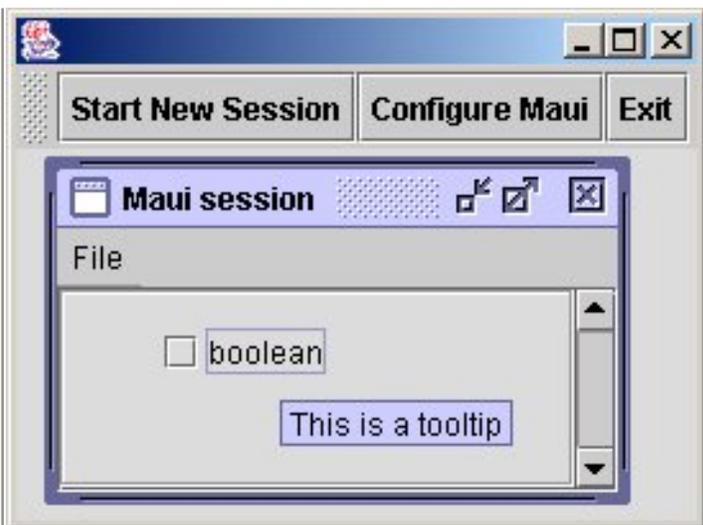
editable: If false then the end-user can not change the contents of the checkbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="boolean1"
        label="boolean1"
        editable="true"
        default="true"/>
      <Boolean name="boolean2"
        label="boolean2"
        editable="false"
        default="true"/>
    </Fields>
  </Class>
</Maui>
```



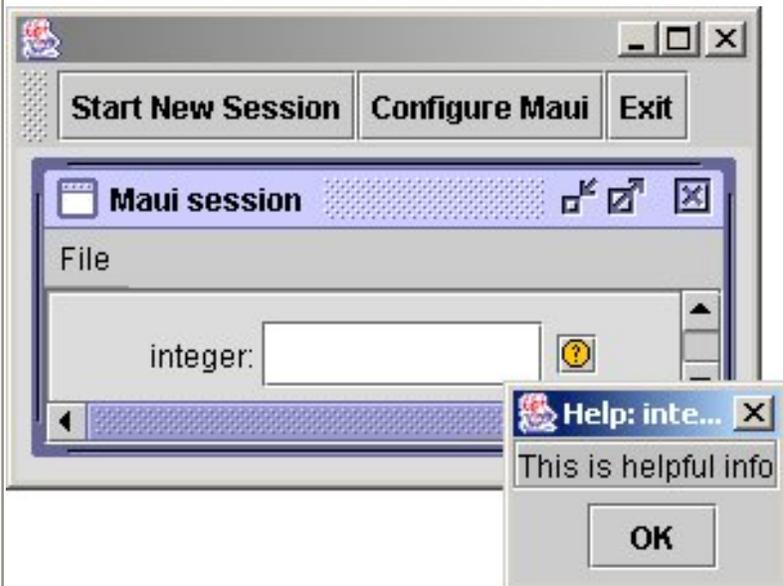
tooltip: Assign a tooltip to the label

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean"
        tooltip="This is a tooltip"/>
    </Fields>
  </Class>
</Maui>
```



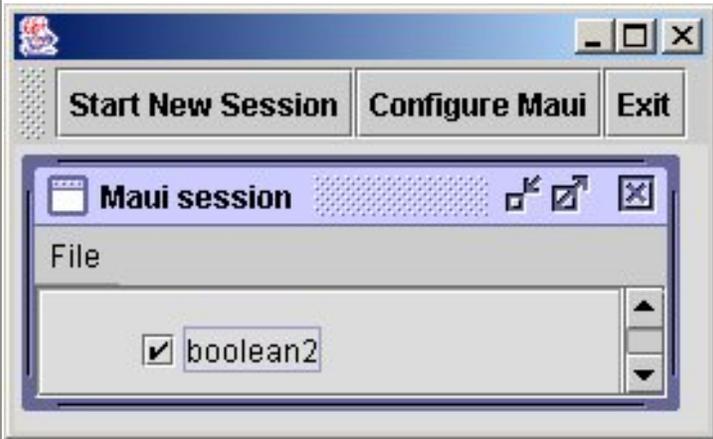
helpMessage: Display help

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="myBoolean" label="boolean"
        helpMessage="This is helpful info"/>
    </Fields>
  </Class>
</Maui>
```



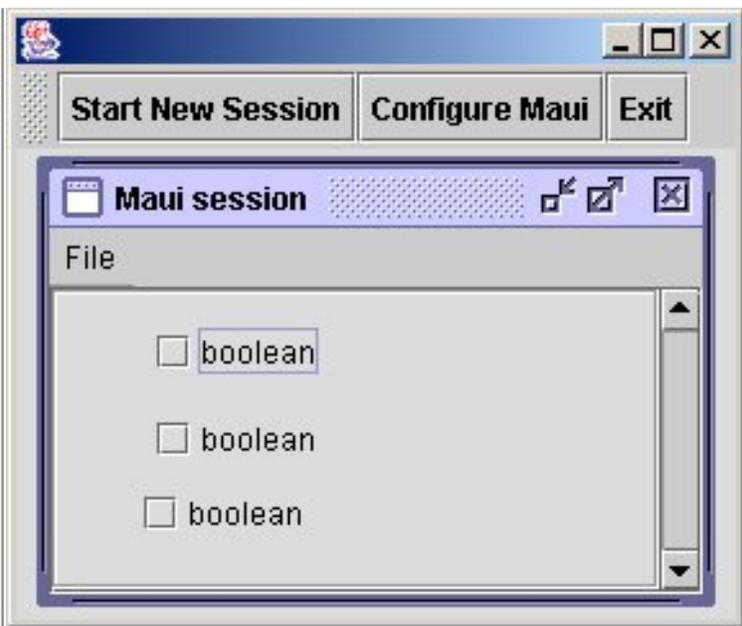
visible: Is the textbox visible on the screen?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="boolean1" label="boolean1"
        visible="false" default="true"/>
      <Boolean name="boolean2" label="boolean2"
        visible="true" default="true"/>
    </Fields>
  </Class>
</Maui>
```



insets: Is there a lot of white space surrounding the checkbox?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Boolean name="boolean1" label="boolean"/>
      <Boolean name="boolean2" label="boolean" insets="true"/>
      <Boolean name="boolean3" label="boolean" insets="false"/>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.11 Tag String](#) **Up:** [B.10 Tag Boolean](#) **Previous:** [B.10.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.11.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.10.2 Attributes allowed in](#)

B.11 Tag String

String elements represent string variables.

Subsections

- [B.11.1 Children allowed in String elements](#)
 - [B.11.2 Attributes allowed in String elements](#)
-

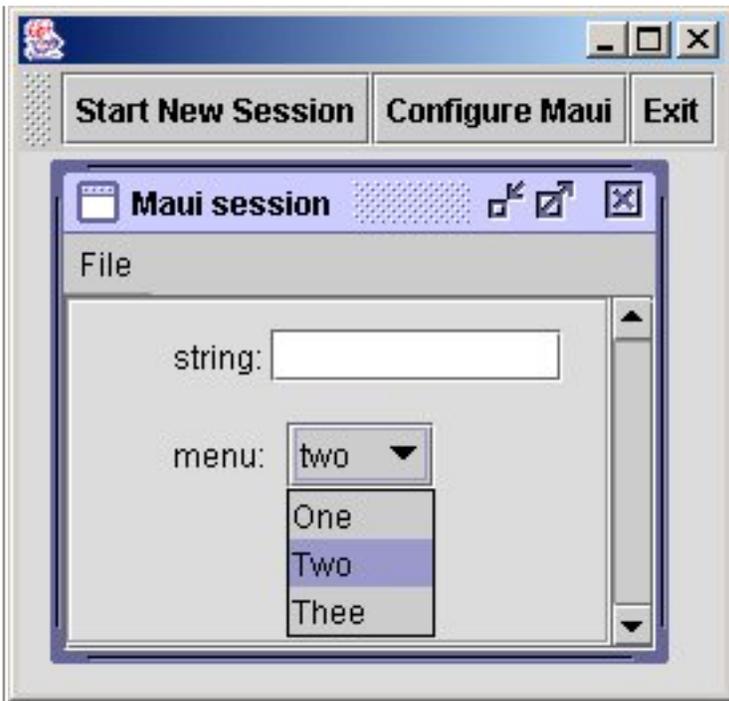
[Next](#) [Up](#) [Previous](#)
Next: [B.11.2 Attributes allowed in](#) **Up:** [B.11 Tag String](#) **Previous:** [B.11 Tag String](#)

B.11.1 Children allowed in String elements

Tag	Number	Description and comments	Examples
CustomEditor	0-1	Allows the string to use a non-standard editor.	example
Menu	0-1	use an options list (combo box, radio buttons, list) for string selection	example
TextArea	0-1	Use a text area for multiple line strings	example
AppData	0-1	a free form block of XML used for data where no data is needed	example
Help	0-1	Display a help icon. If the end-user presses the help icon then pop up some useful information.	example

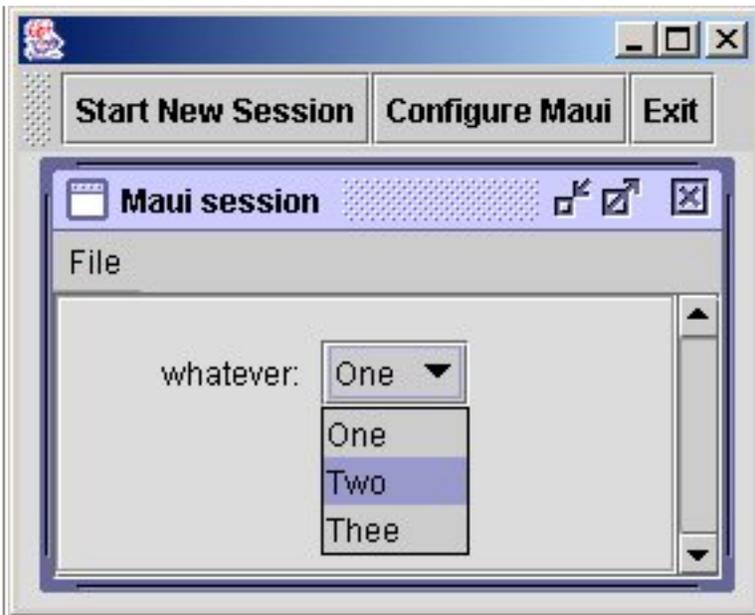
<String> : display a textbox, pull-down menu, list, or a group of radio buttons.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string"/>
      <String name="myMenu" label="menu" default="two">
        <Menu options="One|Two|Thee"/>
      </String>
    </Fields>
  </Class>
</Maui>
```



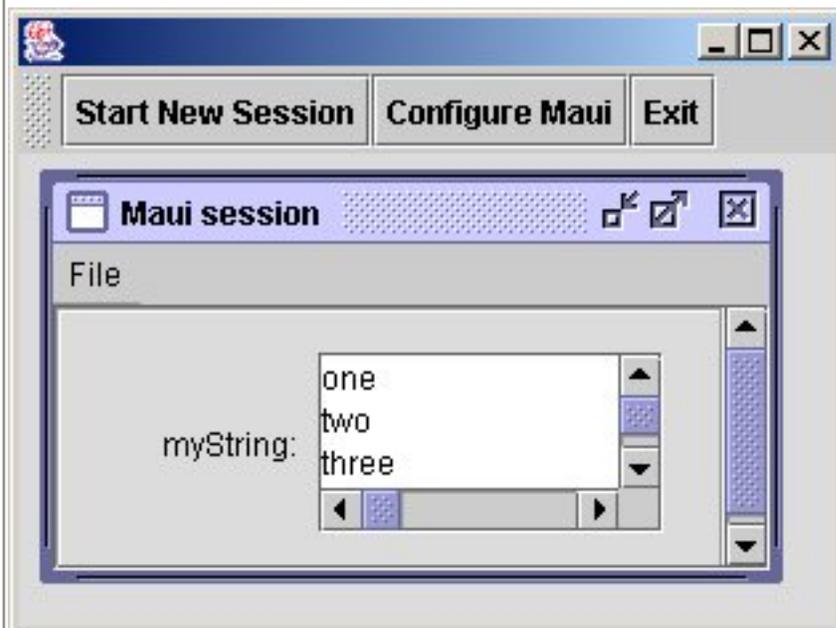
<Menu> : Display a drop-down menu

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="whatever">
        <Menu options="One|Two|Thee" />
      </String>
    </Fields>
  </Class>
</Maui>
```



<TextArea> : Create a textbox that contains more than one line

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" default="one\ntwo\nthree">
        <TextArea width="10" height="3"/>
      </String>
    </Fields>
  </Class>
</Maui>
```



<CustomEditor> : You can write you own customized editor.

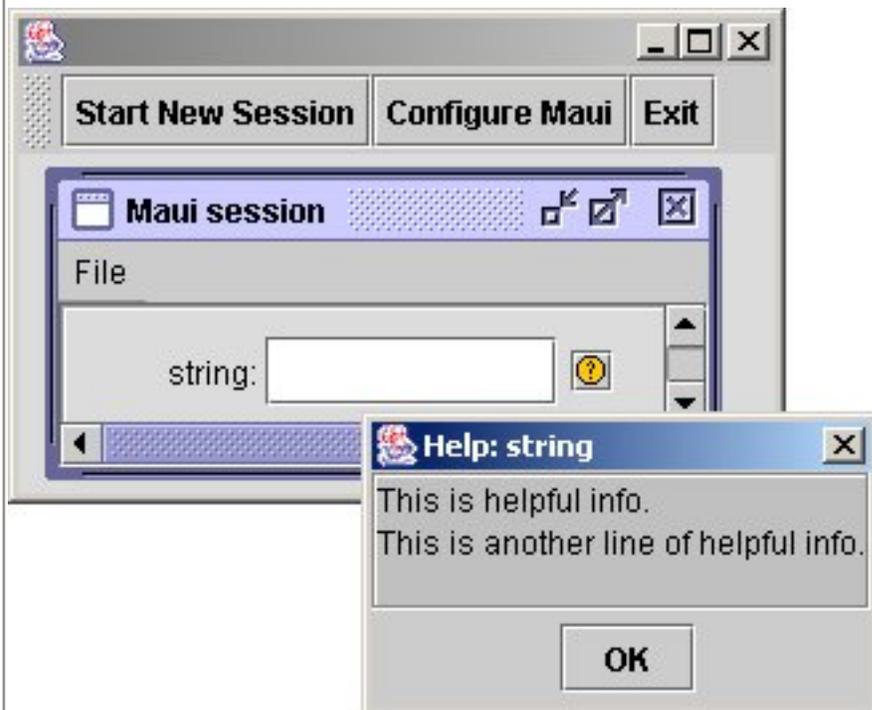
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string">
        <CustomEditor name="MyStringEditor">
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </CustomEditor>
      </String>
    </Fields>
  </Class>
</Maui>
```

<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string">
        <AppData>
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </AppData>
      </String>
    </Class>
</Maui>
```

<Help> : Display information that may be helpful to the end-user.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string">
        <Help>
          This is helpful info.
          This is another line of helpful info.
        </Help>
      </String>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.11.2 Attributes allowed in Up](#) **Up:** [B.11 Tag String](#) **Previous:** [B.11 Tag String](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.12 Tag Array](#) **Up:** [B.11 Tag String](#) **Previous:** [B.11.1 Children allowed in](#)

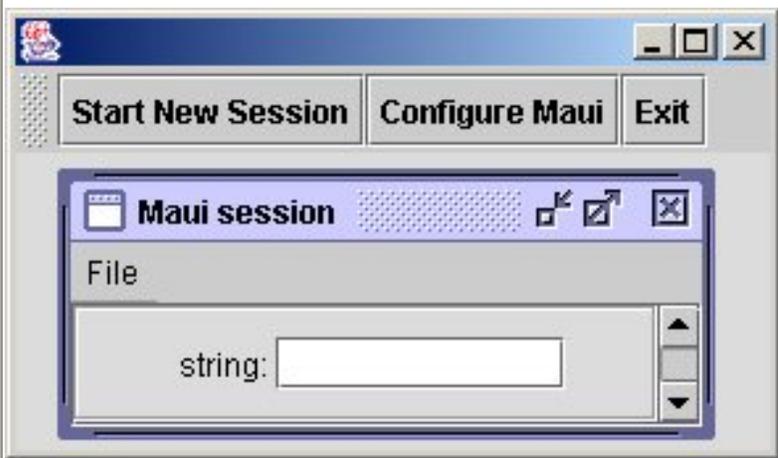
B.11.2 Attributes allowed in String elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|-----------------------|---|-------------------------|
| name | yes | any legal name | name for this variable | example |
| label | no | any string | string to be used as a descriptive label | example |
| default | no | any string | default value of this variable | example |
| optional | no | true/false | indicates if a value needs to be filled in | example |
| editable | no | true/false | indicates if the value can be edited | example |
| columnWidth | no | an integer | gives the number of characters that should be displayed in the text field. | example |
| tooltip | no | any string | When the end-user moves the mouse over the label then display a tooltip. | example |
| helpMessage | no | any string | Display a HELP icon. Whenever the end-user clicks on the icon, a helpful message pops up on the screen. | example |
| visible | no | true or false | Is the textbox visible on the screen. | example |
| mauiAction | no | name of a MauiAction. | This java class is invoked whenever the end-user changes the contents of the textbox. | example |

| | | | | |
|--------|----|---------------|--|-------------------------|
| insets | no | true or false | If true then surround the GUI component with a lot of white space. | example |
|--------|----|---------------|--|-------------------------|

<String> : display a textbox, pull-down menu, list, or a group of radio buttons.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```

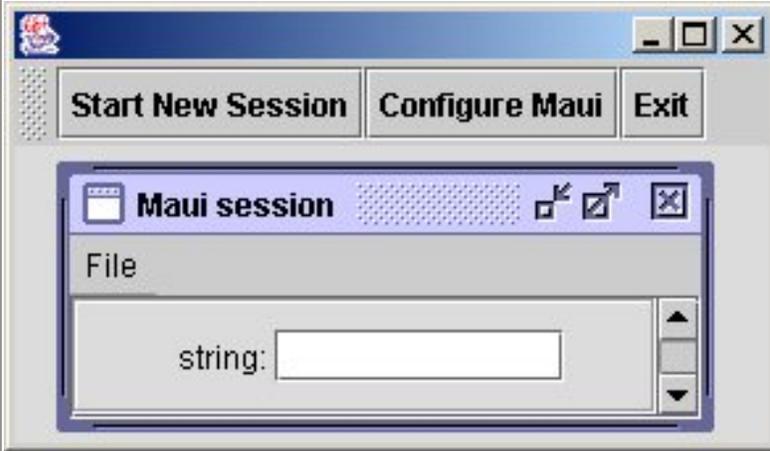


name: the name of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```

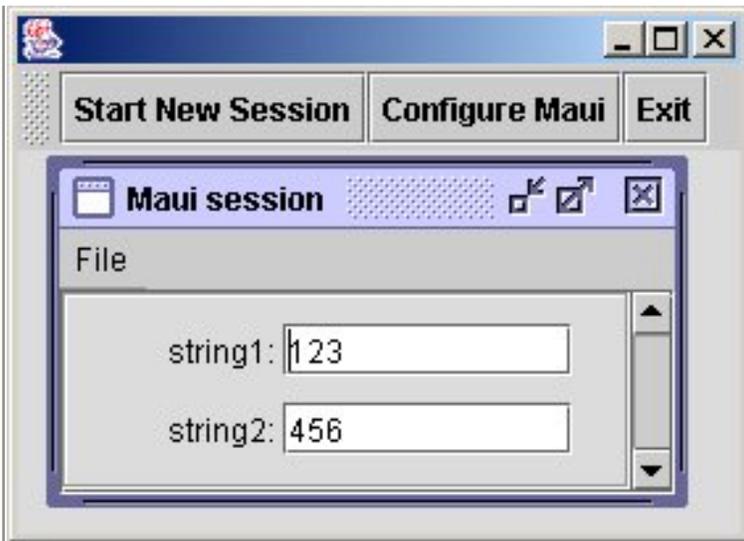
label: The text that is displayed to the left of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



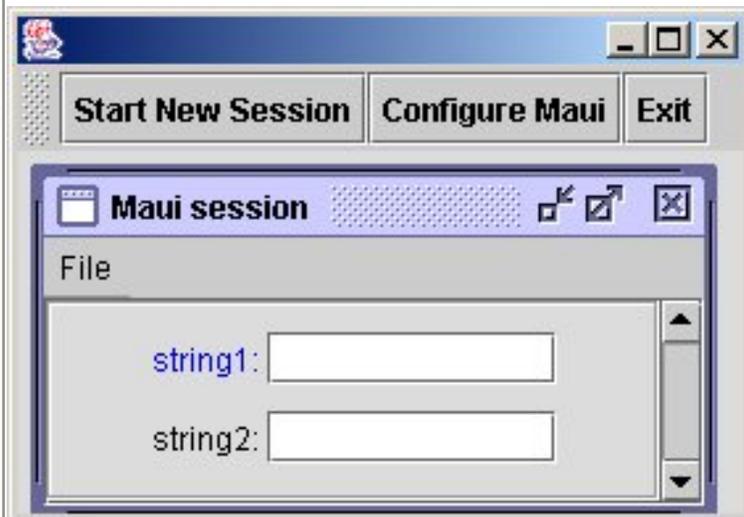
default: The value that is displayed inside the textbox, when the textbox first appears on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1" default="123"/>
      <String name="string2" label="string2" default="456"/>
    </Fields>
  </Class>
</Maui>
```



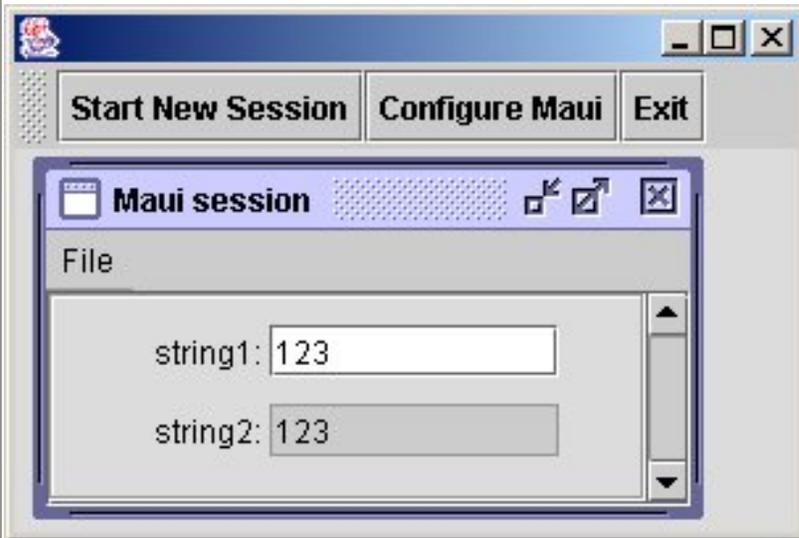
optional: If true then the end-user is not forced to insert a value into the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1" optional="true"/>
      <String name="string2" label="string2" optional="false"/>
    </Fields>
  </Class>
</Maui>
```



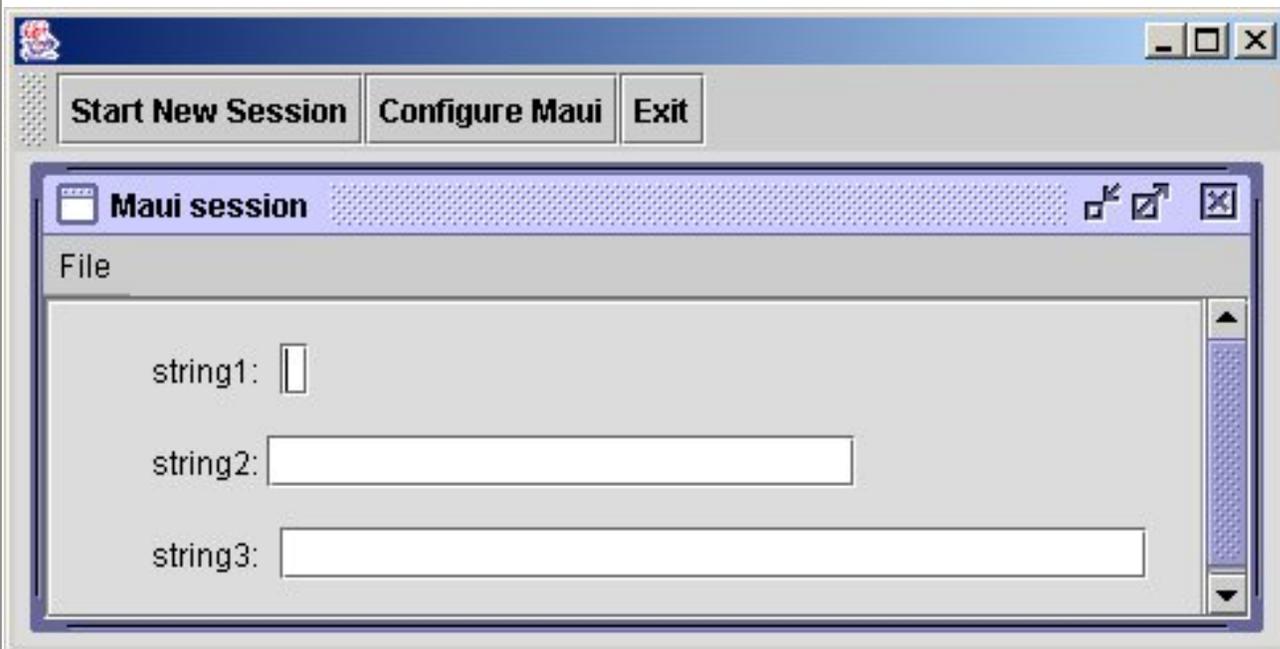
editable: If false then the end-user can not change the contents of the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1"
        label="string1"
        editable="true"
        default="123" />
      <String name="string2"
        label="string2"
        editable="false"
        default="123" />
    </Fields>
  </Class>
</Maui>
```



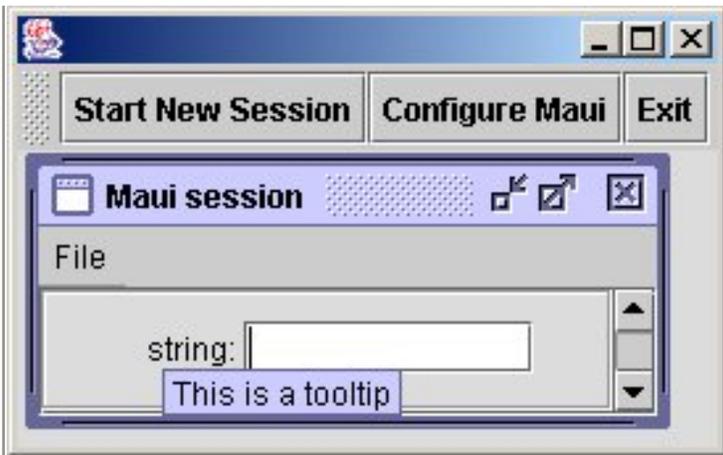
columnWidth: the number of characters that can fit inside the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1" columnWidth="1"/>
      <String name="string2" label="string2"/>
      <String name="string3" label="string3" columnWidth="30"/>
    </Fields>
  </Class>
</Maui>
```



tooltip: Assign a tooltip to the label

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string"
        tooltip="This is a tooltip"/>
    </Fields>
  </Class>
</Maui>
```



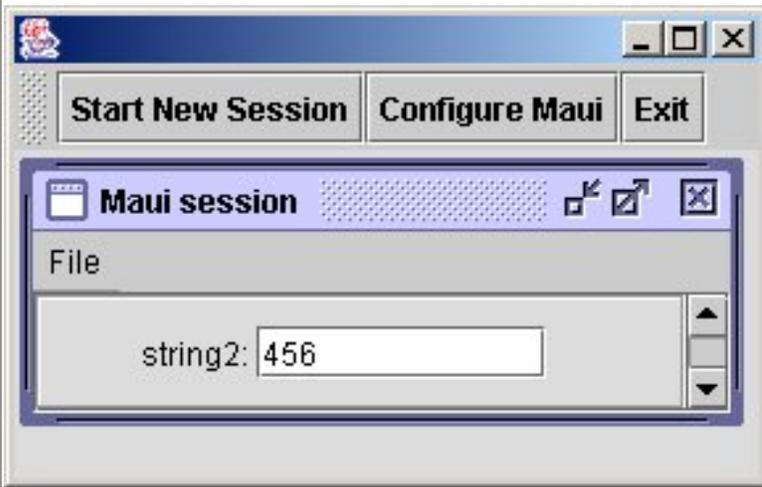
helpMessage: Display help

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myString" label="string"
        helpMessage="This is helpful info"/>
    </Fields>
  </Class>
</Maui>
```



visible: Is the textbox visible on the screen?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1"
        visible="false" default="123"/>
      <String name="string2" label="string2"
        visible="true" default="456"/>
    </Fields>
  </Class>
</Maui>
```

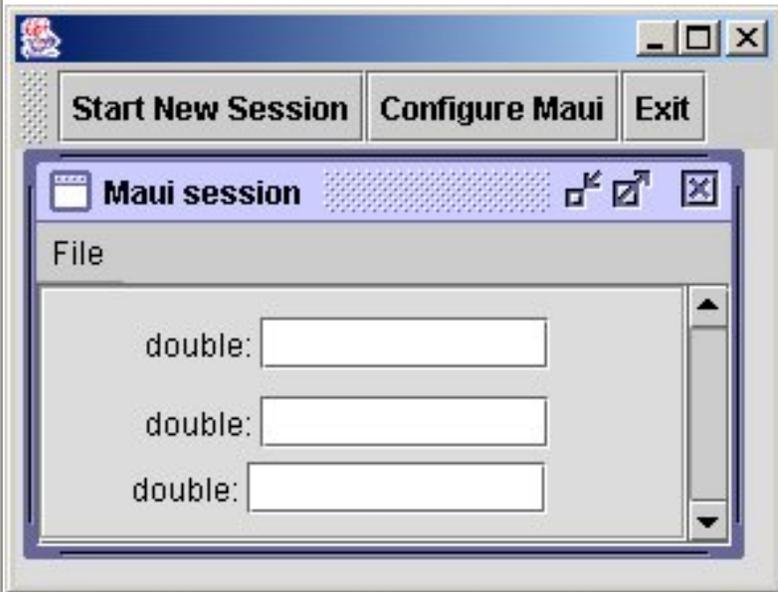


MauiAction: This Java class is invoked whenever the end-user changes the contents of the textbox

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string1"
        mauiAction="MyMauiAction"/>
    </Fields>
  </Class>
</Maui>
```

insets: Is there a lot of white space surrounding the textbox?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="string1" label="string" />
      <String name="string2" label="string" insets="true" />
      <String name="string3" label="string" insets="false" />
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.12 Tag Array](#) **Up:** [B.11 Tag String](#) **Previous:** [B.11.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.12.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.11.2 Attributes allowed in](#)

B.12 Tag Array

Array elements represent an array of classes

Subsections

- [B.12.1 Children allowed in Array elements](#)
 - [B.12.2 Attributes allowed in Array elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.12.2 Attributes allowed in](#) **Up:** [B.12 Tag Array](#) **Previous:** [B.12 Tag Array](#)

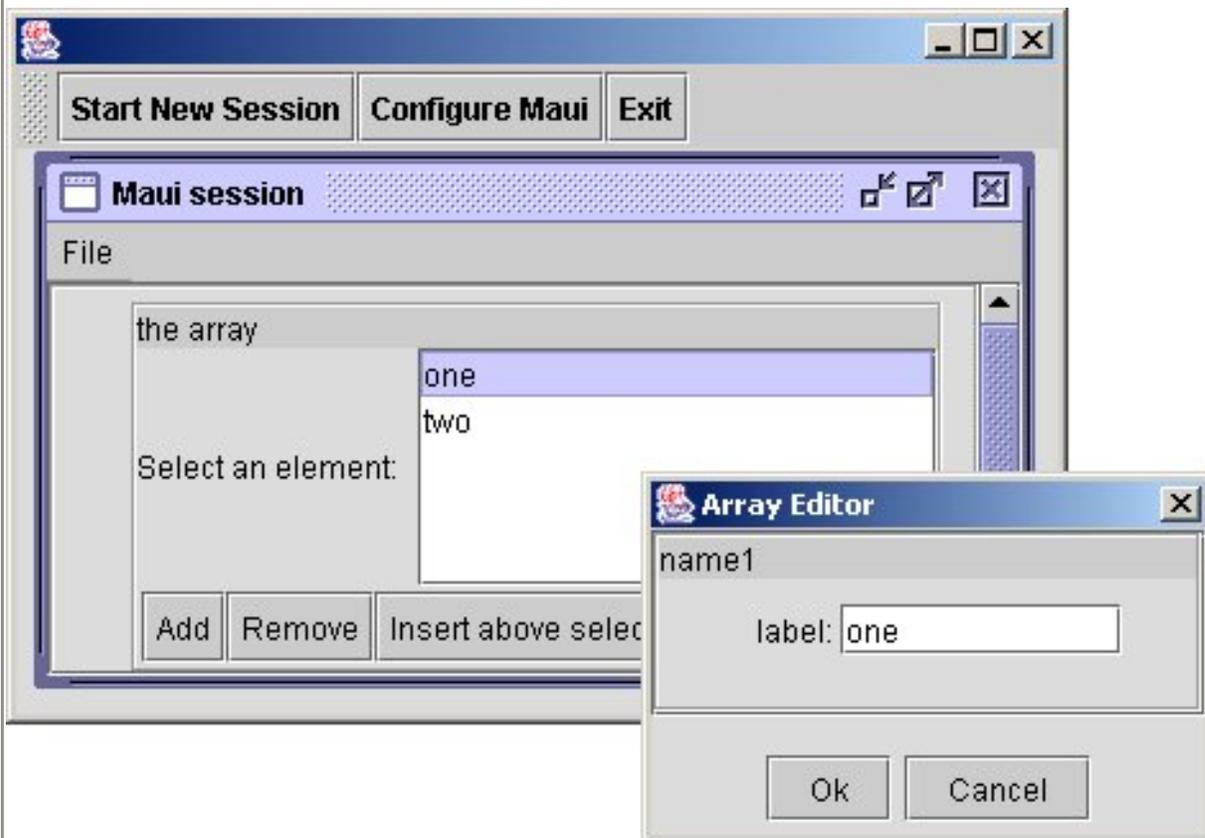
B.12.1 Children allowed in Array elements

| Tag | Number | Description and comments | Examples |
|--------------|------------|---|-------------------------|
| Action | any number | Allows for buttons to be placed within the class editor | example |
| CustomEditor | 0-1 | Allows the Array to use a non-standard editor. | example |
| Master | 1 | contains the template used for adding new items | example |
| Contents | 0-1 | holds the initial contents of the array | example |
| AppData | 0-1 | a free form block of XML used for data where no editor is needed | example |
| Help | 0-1 | Display a help icon. If the end-user presses the help icon then pop up some useful information. | example |

<Array>

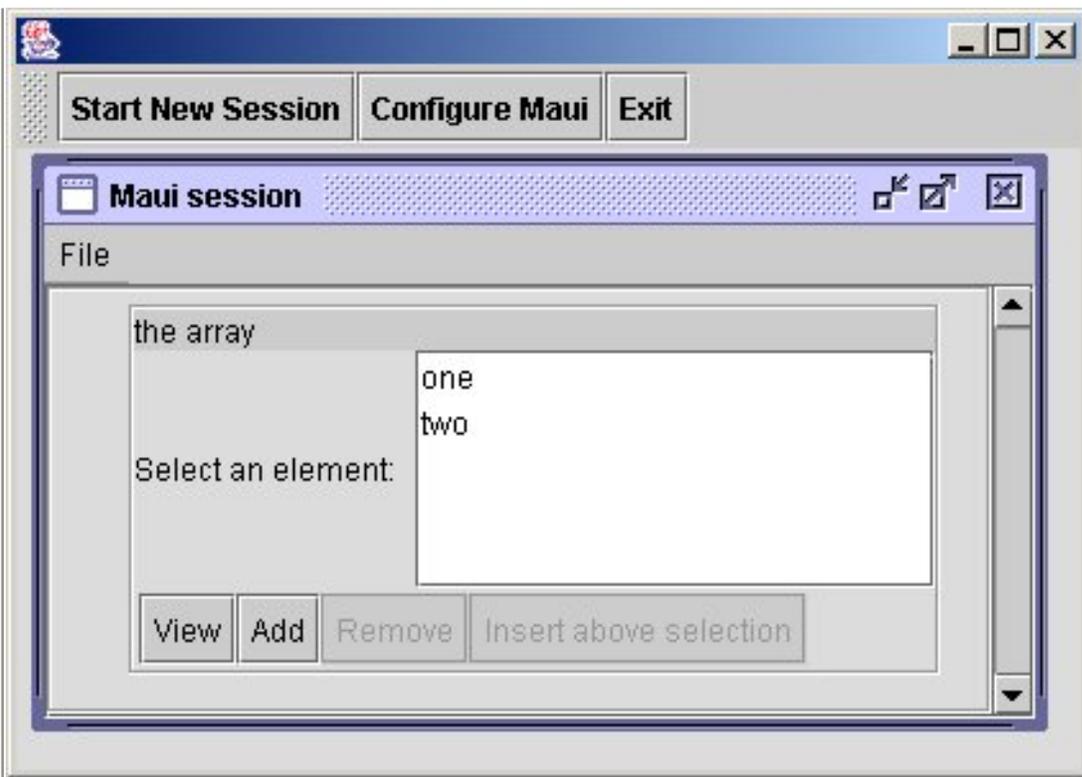
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
      <Contents>
```

```
<Item>
  <Class type="row1">
    <Fields>
      <String name="string"
        label="label"
        default="one"/>
    </Fields>
  </Class>
</Item>
<Item>
  <Class type="row2">
    <Fields>
      <String name="string"
        label="label"
        default="two"/>
    </Fields>
  </Class>
</Item>
</Contents>
</Array>
</Fields>
</Class>
</Maui>
```



<Action> : Insert a button

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Action label="View" class="Maui.Interface.ViewAction"/>
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>
```



<CustomEditor> : Replace the array editor with your own custom editor

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <CustomEditor name="MyArrayEditor">
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </CustomEditor>
      <Master label="$string">
        <Class type="master" label="label">
          <Fields>
            <String name="string"
              label="label"/>
          </Fields>
        </Class>
      </Master>
    </Fields>
    <Contents>
      <Item>
        <Class type="row1">
          <Fields>
            <String name="string"
              label="label"
  
```

```

                default="one"/>
            </Fields>
        </Class>
    </Item>
    <Item>
        <Class type="row2">
            <Fields>
                <String name="string"
                    label="label"
                    default="two"/>
            </Fields>
        </Class>
    </Item>
</Contents>
</Array>
</Fields>
</Class>
</Maui>

```

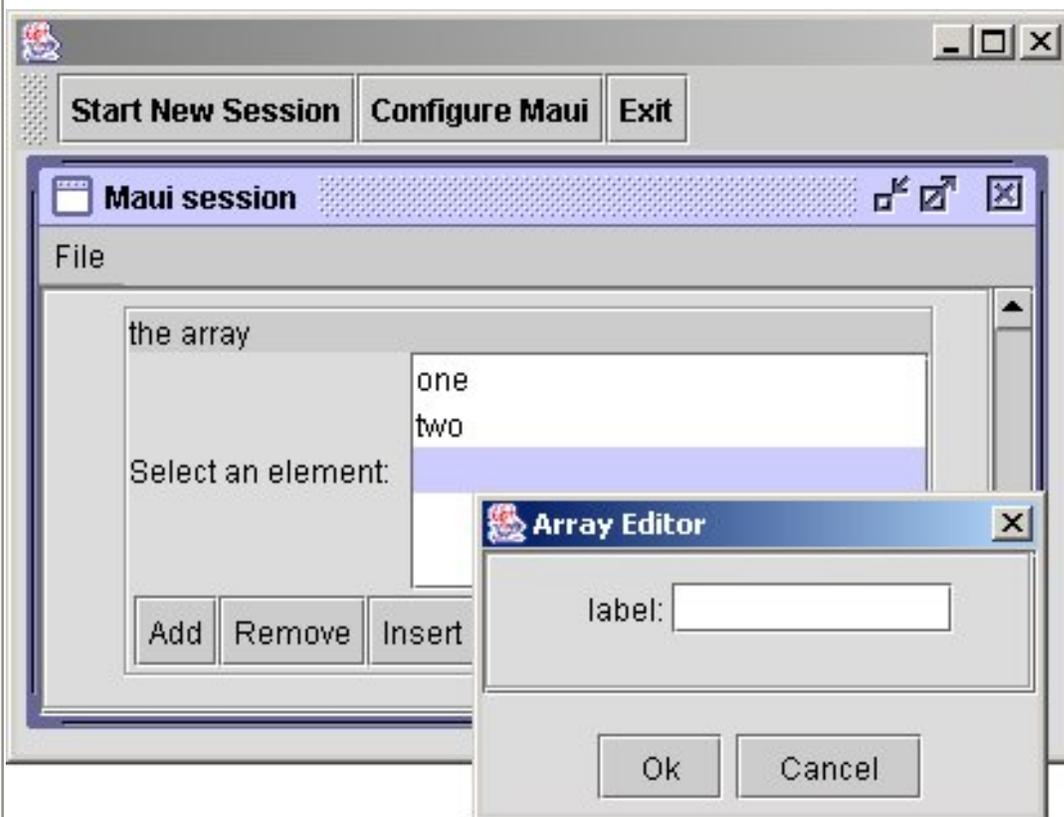
<Master> : If the end-user presses the ADD or INSERT button then the Master class is used fabricate a new array element.

```

<Maui RootClass="MyContainer">
    <Class type="MyContainer" label="the container">
        <Fields>
            <Array name="myArray" label="the array">
                <Master label="$string">
                    <Class type="master" label="label">
                        <Fields>
                            <String name="string"
                                label="label"/>
                        </Fields>
                    </Class>
                </Master>
            <Contents>
                <Item>
                    <Class type="row1">
                        <Fields>
                            <String name="string"
                                label="label"
                                default="one"/>
                        </Fields>
                    </Class>
                </Item>
            </Contents>
        </Array>
    </Fields>
</Class>
</Maui>

```

```
<Class type="row2">
  <Fields>
    <String name="string"
           label="label"
           default="two"/>
  </Fields>
</Class>
</Item>
</Contents>
</Array>
</Fields>
</Class>
</Maui>
```

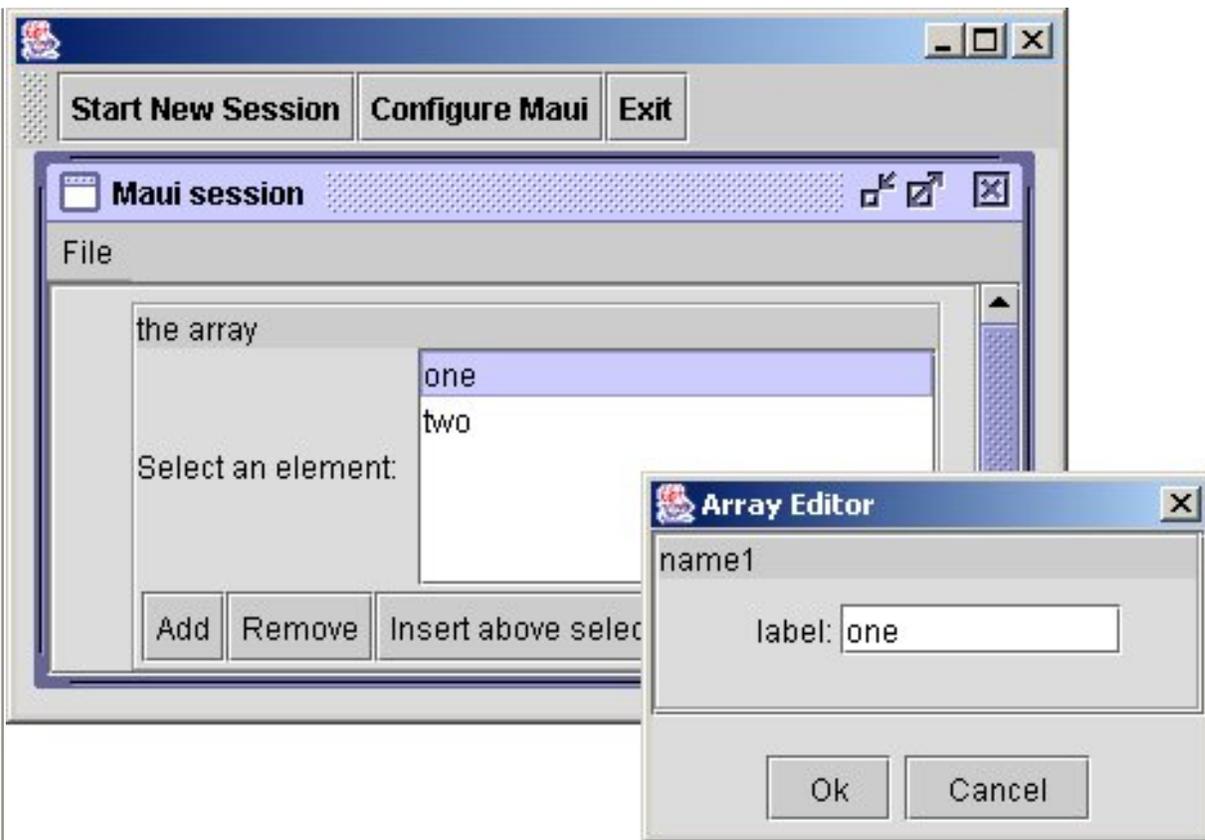


<Contents> : The contents of the array when the array is first rendered on the screen

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>

```



<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>
```

```

        </Class>
    </Item>
    <Item>
        <Class type="row2">
            <Fields>
                <String name="string"
                    label="label"
                    default="two" />
            </Fields>
        </Class>
    </Item>
</Contents>

```

```

<AppData>
  <parameter1>one</parameter1>
  <parameter2>two</parameter2>
</AppData>

```

```

    </Array>
  </Fields>
</Class>
</Maui>

```

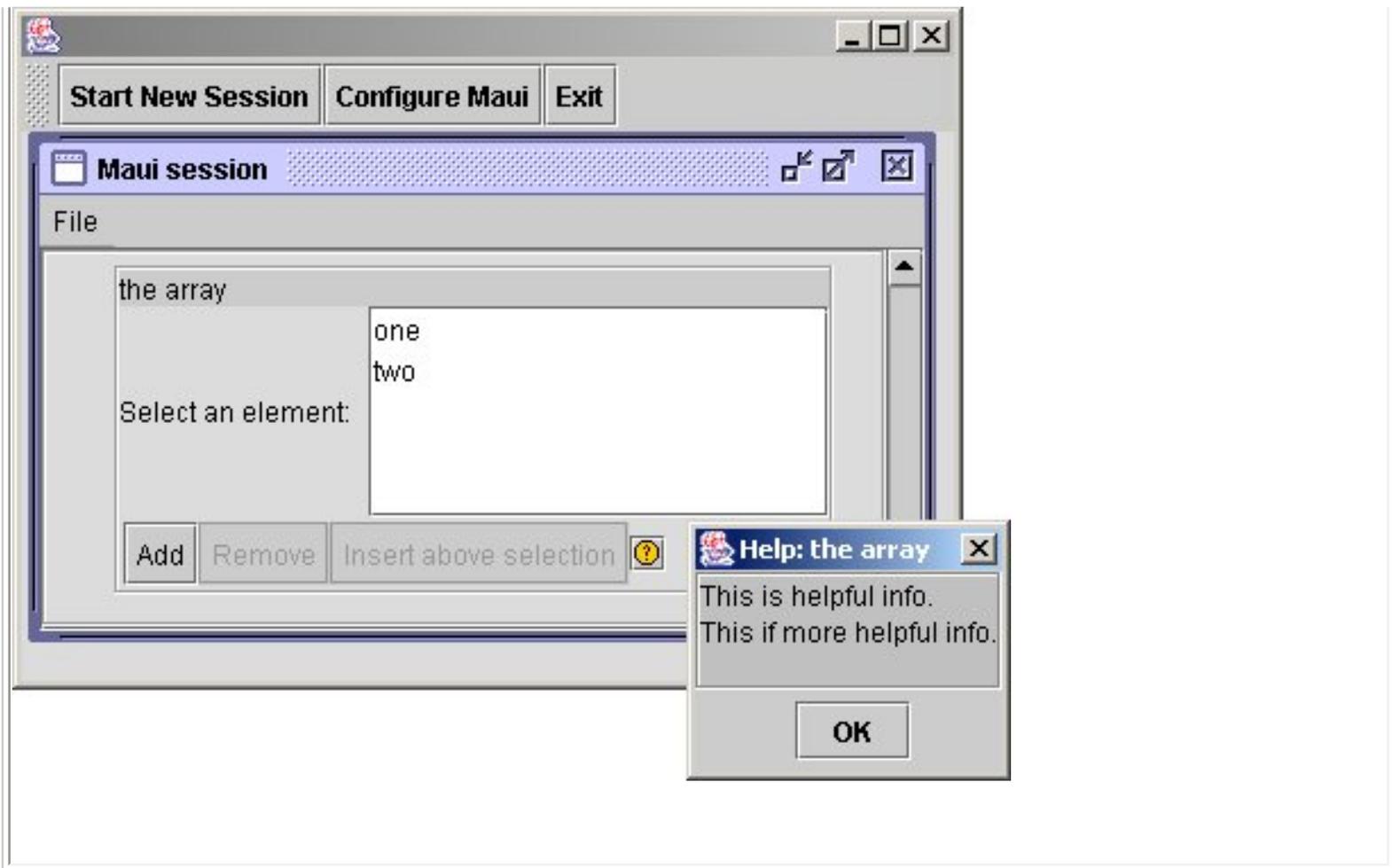
<Help> : Display helpful info

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label" />
            </Fields>
          </Class>
        </Master>
      <Contents>
        <Item>
          <Class type="row1">
            <Fields>
              <String name="string"

```

```
        label="label "  
        default="one" />  
    </Fields>  
    </Class>  
</Item>  
<Item>  
    <Class type="row2">  
        <Fields>  
            <String name="string"  
                label="label"  
                default="two" />  
        </Fields>  
    </Class>  
</Item>  
</Contents>  
  
<Help>  
    This is helpful info.  
    This if more helpful info.  
</Help>  
  
    </Array>  
    </Fields>  
    </Class>  
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.12.2 Attributes allowed in](#) **Up:** [B.12 Tag Array](#) **Previous:** [B.12 Tag Array](#)

[Next](#) [Up](#) [Previous](#)

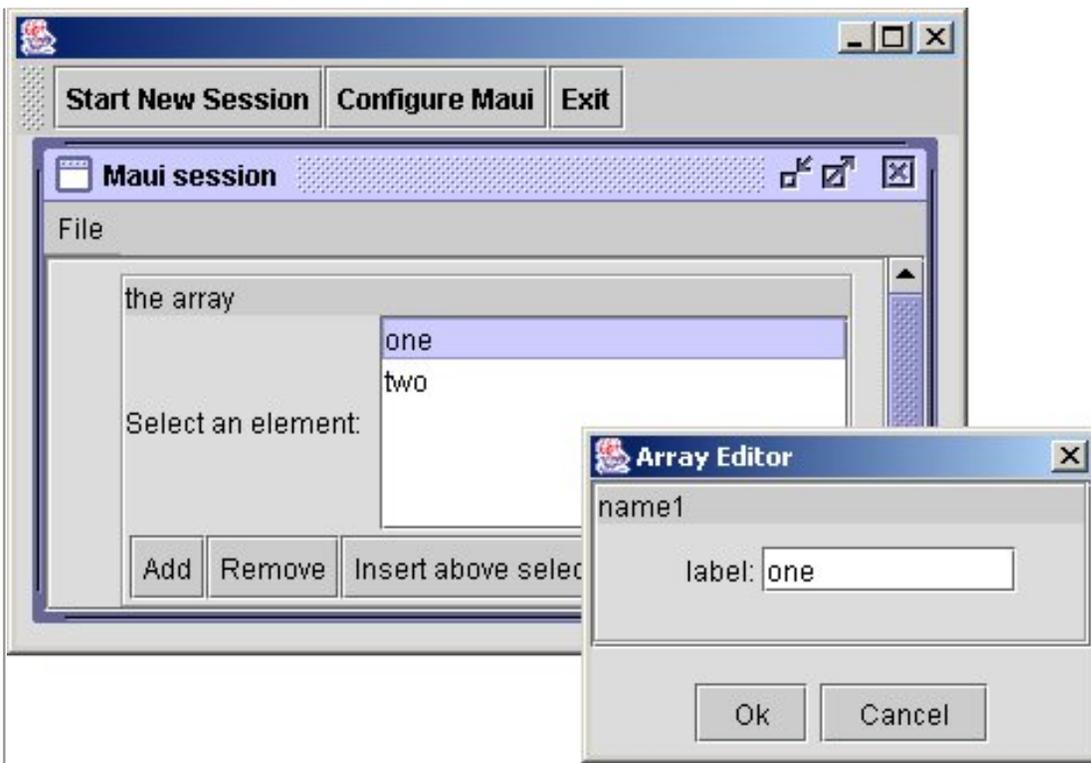
Next: [B.13 Tag Table](#) **Up:** [B.12 Tag Array](#) **Previous:** [B.12.1 Children allowed in](#)

B.12.2 Attributes allowed in Array elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|----------------|--|-------------------------|
| name | yes | any legal name | name for this variable | example |
| label | no | any string | string to be used as a descriptive label | example |
| selectionLabel | no | any string | The label to display next to the selection list for the Array. | example |
| collapsible | no | true or false | The panel for this array can be expanded and collapsed with a toggle button. If no value is given, collapsible is assumed to be false. | example |
| beginCollapsed | no | true or false | If the panel is collapsible, this will give the initial state of that panel. By default, a collapsible panel starts out expanded. | example |
| tooltip | no | any string | When the end-user moves the mouse over the label then display a tooltip. | example |
| helpMessage | no | any string | Display a HELP icon. Whenever the end-user clicks on the icon, a helpful message pops up on the screen. | example |
| visible | no | true or false | Is the textbox visible on the screen. | example |

```
<Array>

<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>
```



name: The name of the array

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
      <Contents>
        <Item>
          <Class type="row1">
            <Fields>
              <String name="string"
                label="label"
                default="one"/>
            </Fields>
          </Class>
        </Item>
        <Item>

```

```

    <Class type="row2">
      <Fields>
        <String name="string"
          label="label"
          default="two"/>
      </Fields>
    </Class>
  </Item>
</Contents>
</Array>
</Fields>
</Class>
</Maui>

```

label: The text that appears above the array

```

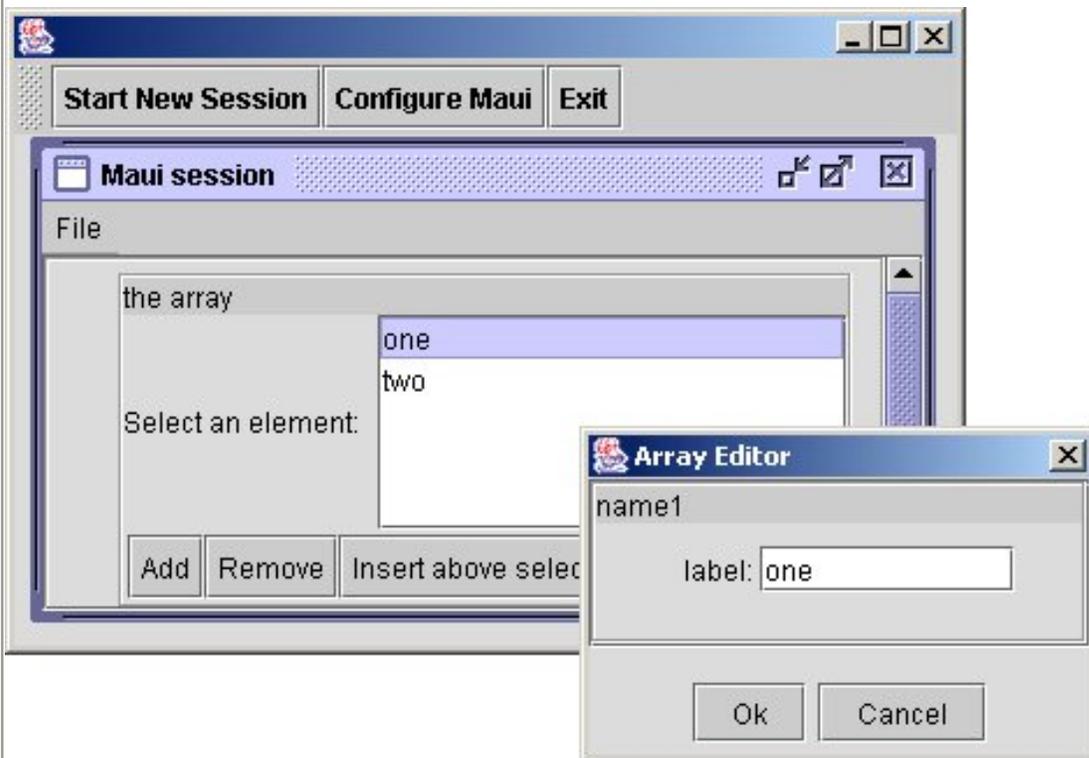
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>

```

```

        </Item>
      </Contents>
    </Array>
  </Fields>
</Class>
</Maui>

```



selectionLabel: The label that appears to the left of the selection list

```

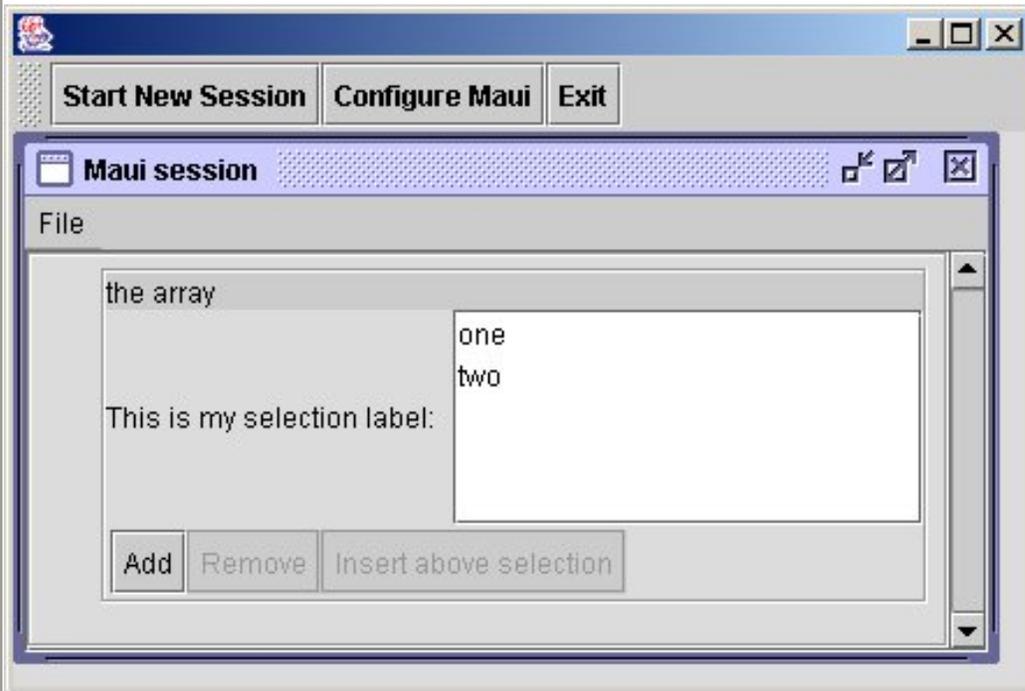
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array"
        selectionLabel="This is my selection label">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
      <Contents>
        <Item>
          <Class type="row1">
            <Fields>
              <String name="string"
                label="label"

```

```

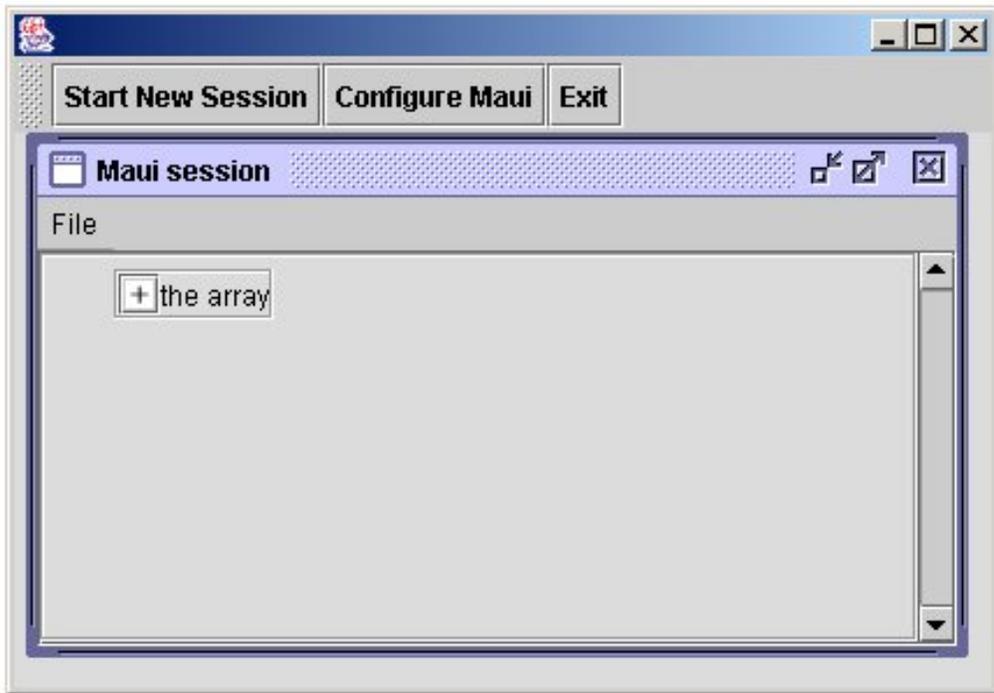
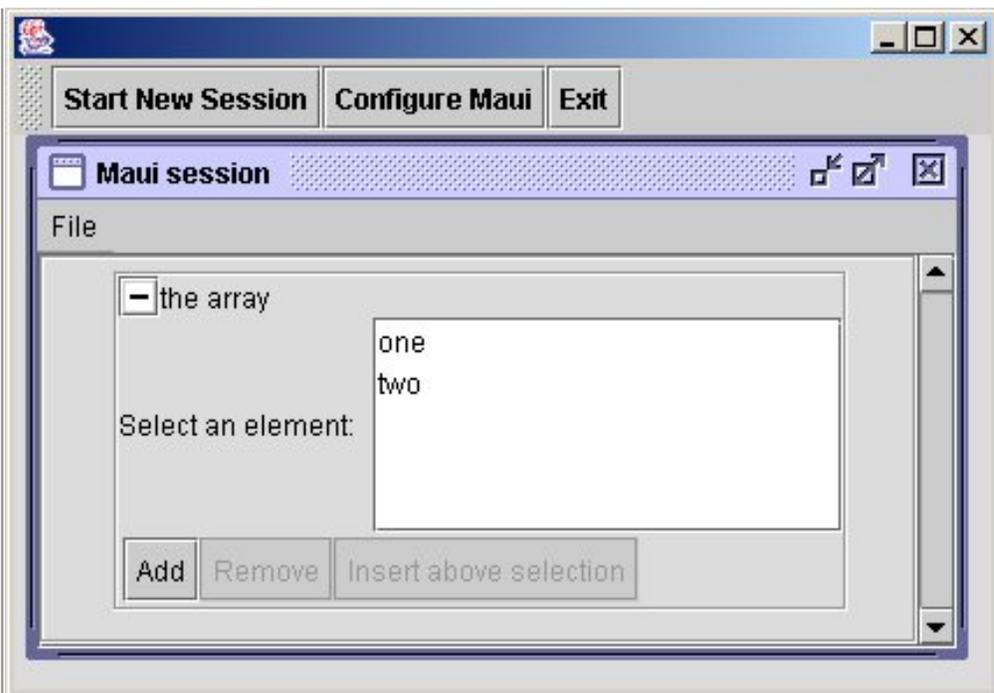
                                default="one" />
                                </Fields>
                                </Class>
                                </Item>
                                <Item>
                                  <Class type="row2">
                                    <Fields>
                                      <String name="string"
                                        label="label"
                                        default="two" />
                                      </Fields>
                                    </Class>
                                  </Item>
                                </Contents>
                              </Array>
                            </Fields>
                          </Class>
</Maui>

```



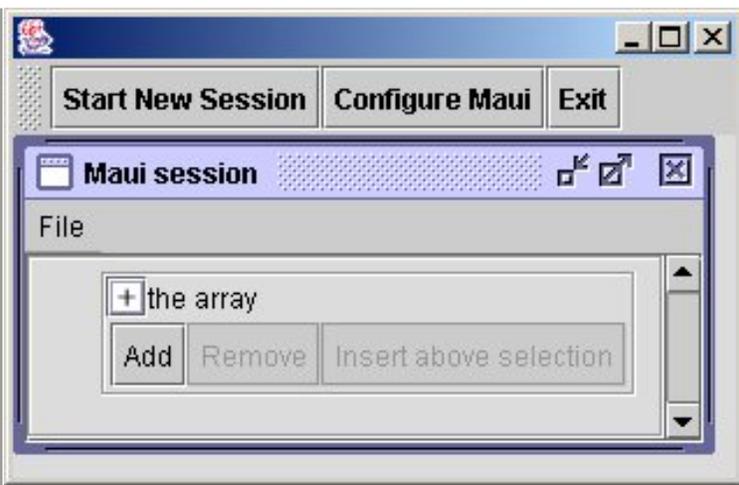
collapsible: Can the end-user hide the array by collapsing the panel

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array" collapsible="true">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>
```



beginCollapsed: Is the array panel collapsed?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array"
        collapsible="true" beginCollapsed="true">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>
```



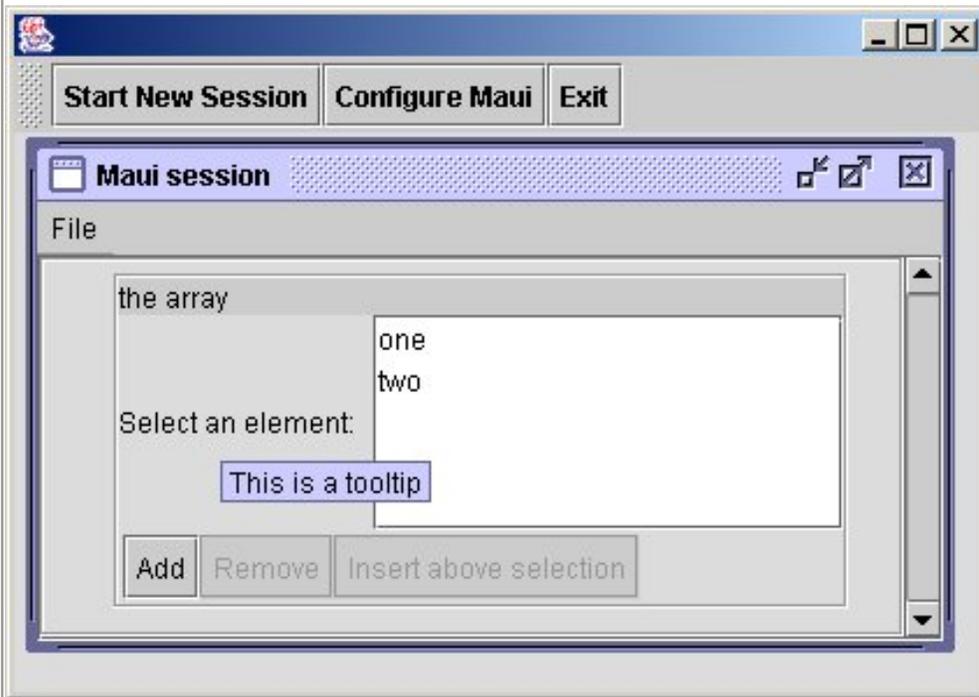
tooltip: Assign a tooltip to the label

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array" tooltip="This is a tooltip">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
```

```

    </Fields>
  </Class>
</Maui>

```



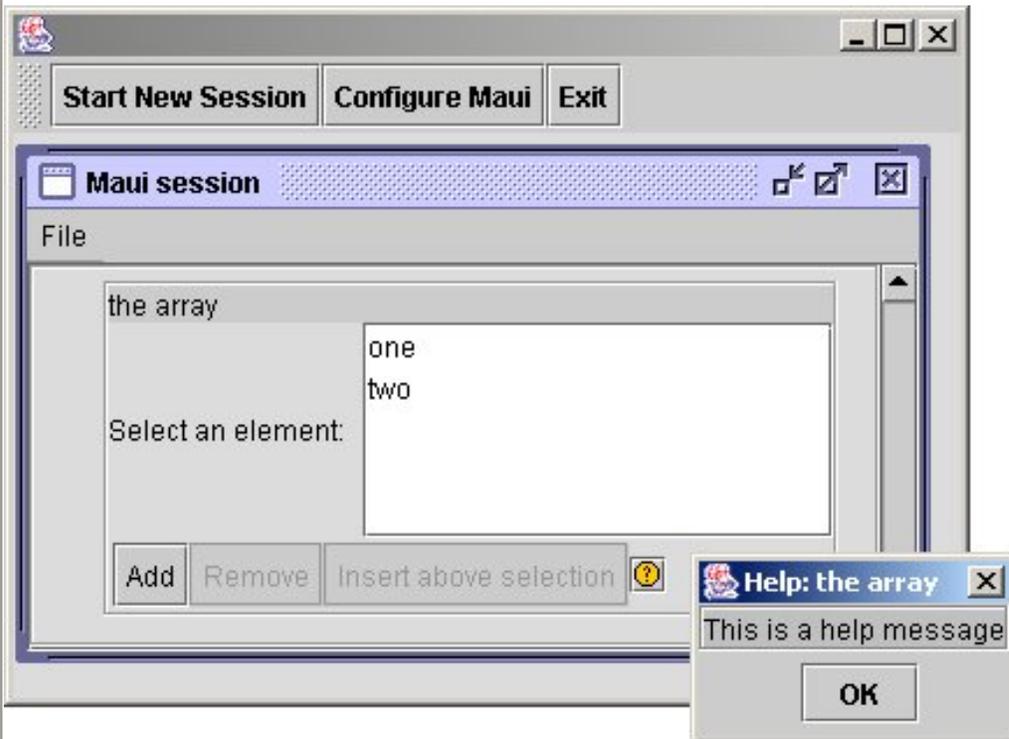
helpMessage: Display help

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array"
        helpMessage="This is a help message">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>

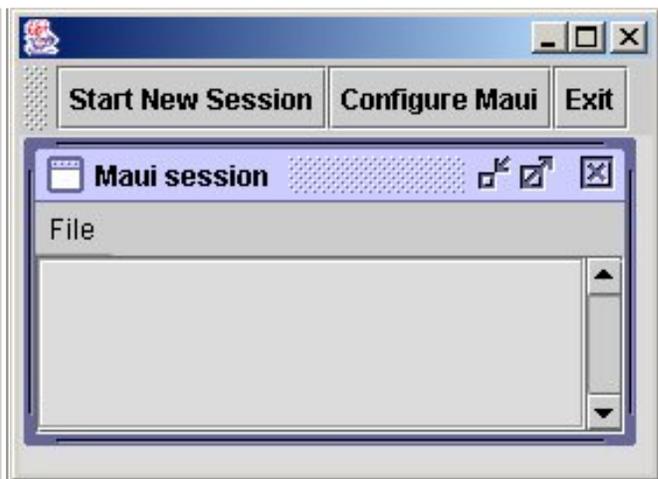
```

```
        </Class>
    </Item>
    <Item>
        <Class type="row2">
            <Fields>
                <String name="string"
                    label="label"
                    default="two"/>
            </Fields>
        </Class>
    </Item>
</Contents>
</Array>
</Fields>
</Class>
</Maui>
```



visible: Is the textbox visible on the screen?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array"
        visible="false">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.13 Tag Table](#) **Up:** [B.12 Tag Array](#) **Previous:** [B.12.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.13.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.12.2 Attributes allowed in](#)

B.13 Tag Table

Table elements represent a table composed of Strings, Integers, Doubles, Booleans, and References.

Subsections

- [B.13.1 Children allowed in Table elements](#)
 - [B.13.2 Attributes allowed in Table elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.13.2 Attributes allowed in](#) **Up:** [B.13 Tag Table](#) **Previous:** [B.13 Tag Table](#)

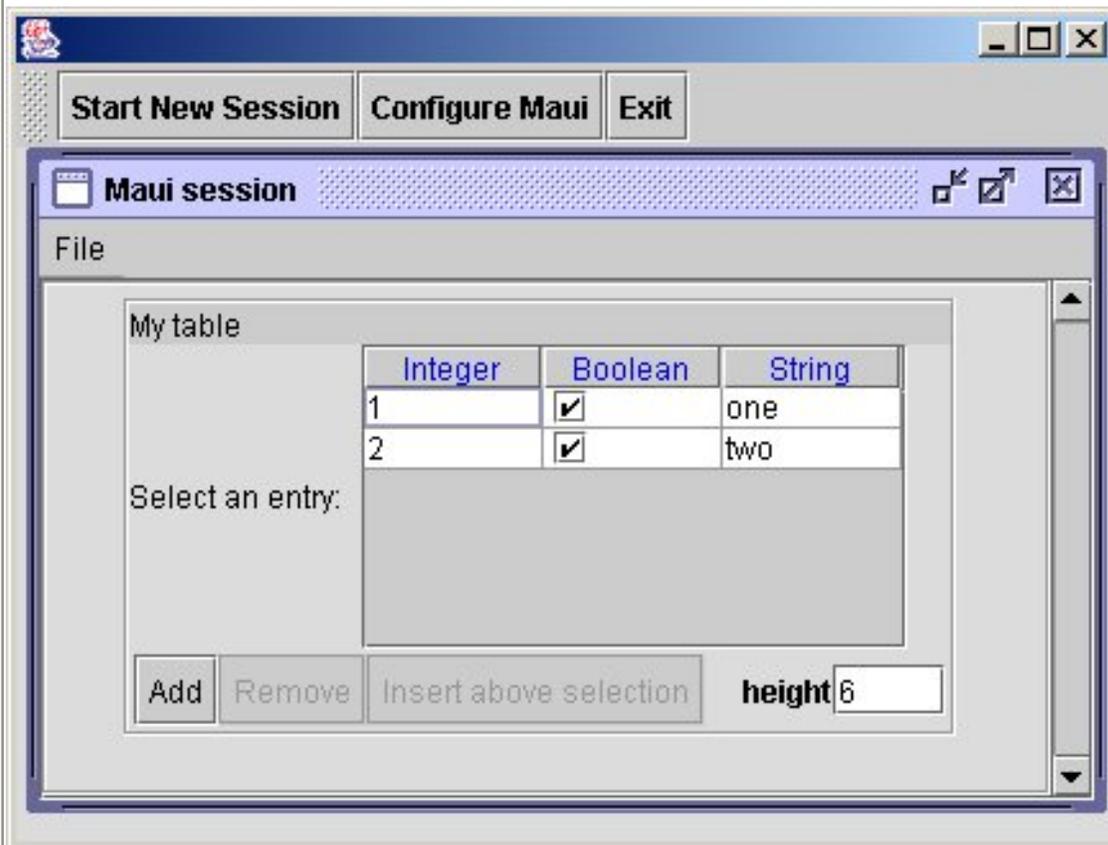
B.13.1 Children allowed in Table elements

| Tag | Number | Description and comments | Examples |
|--------------|------------|---|-------------------------|
| Action | any number | Allows for buttons to be placed within the class editor | example |
| CustomEditor | 0-1 | Allows the Table to use a non-standard editor. | example |
| Header | 1 | contains the template used for adding new entires | example |
| Entries | 0-1 | holds the initial contents of the table | example |
| AppData | 0-1 | a free form block of XML used for data where no editor is needed | example |
| Help | 0-1 | Display a help icon. If the end-user presses the help icon then pop up some useful information. | example |

```
<Table>

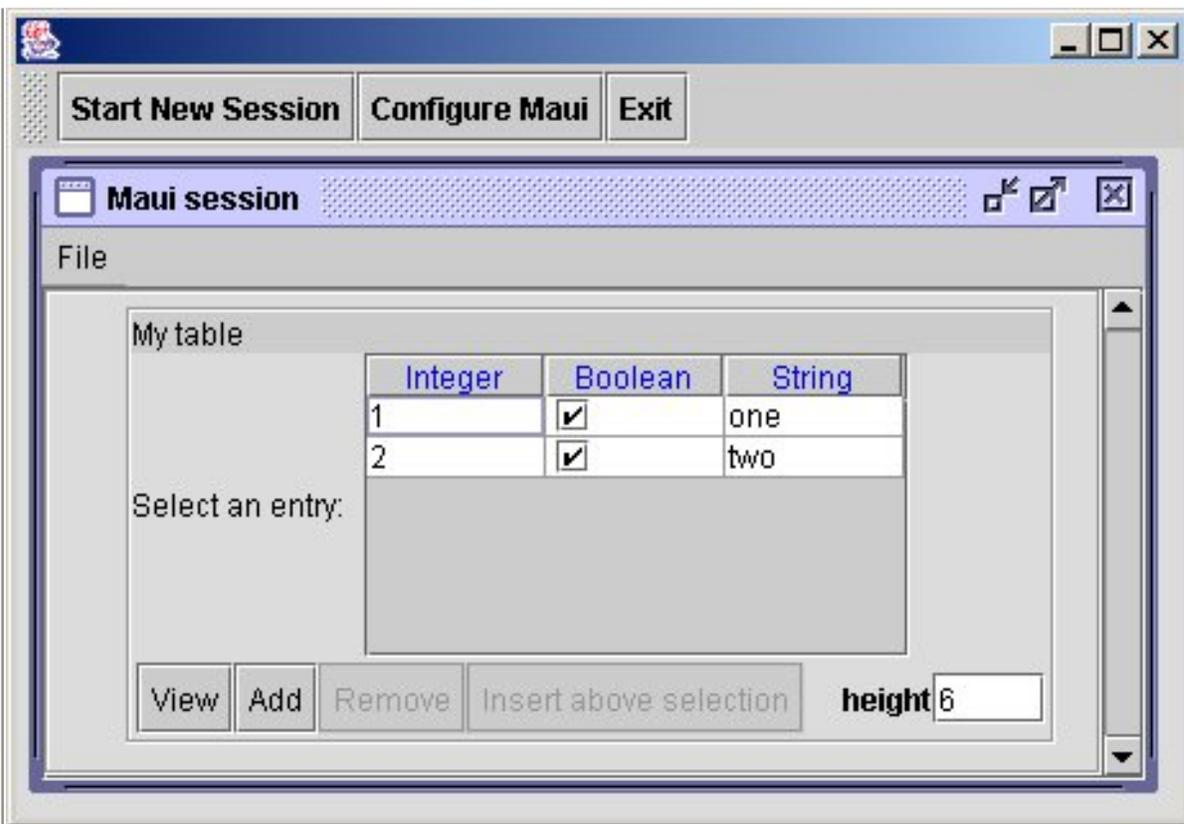
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```
        <Cell field="Col3" value="one" />
    </Entry>
    <Entry name="entry2">
        <Cell field="Col1" value="2" />
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="two" />
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>
```



<Action> : Insert a button

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Action label="View" class="Maui.Interface.ViewAction" />
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```



<CustomEditor> : Replace the table editor with your own custom editor

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <CustomEditor name="MyTableEditor">
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </CustomEditor>
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```

        <Entry name="entry2">
            <Cell field="Col1" value="2"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="two"/>
        </Entry>
    </Entries>
</Table>
</Fields>
</Class>
</Maui>

```

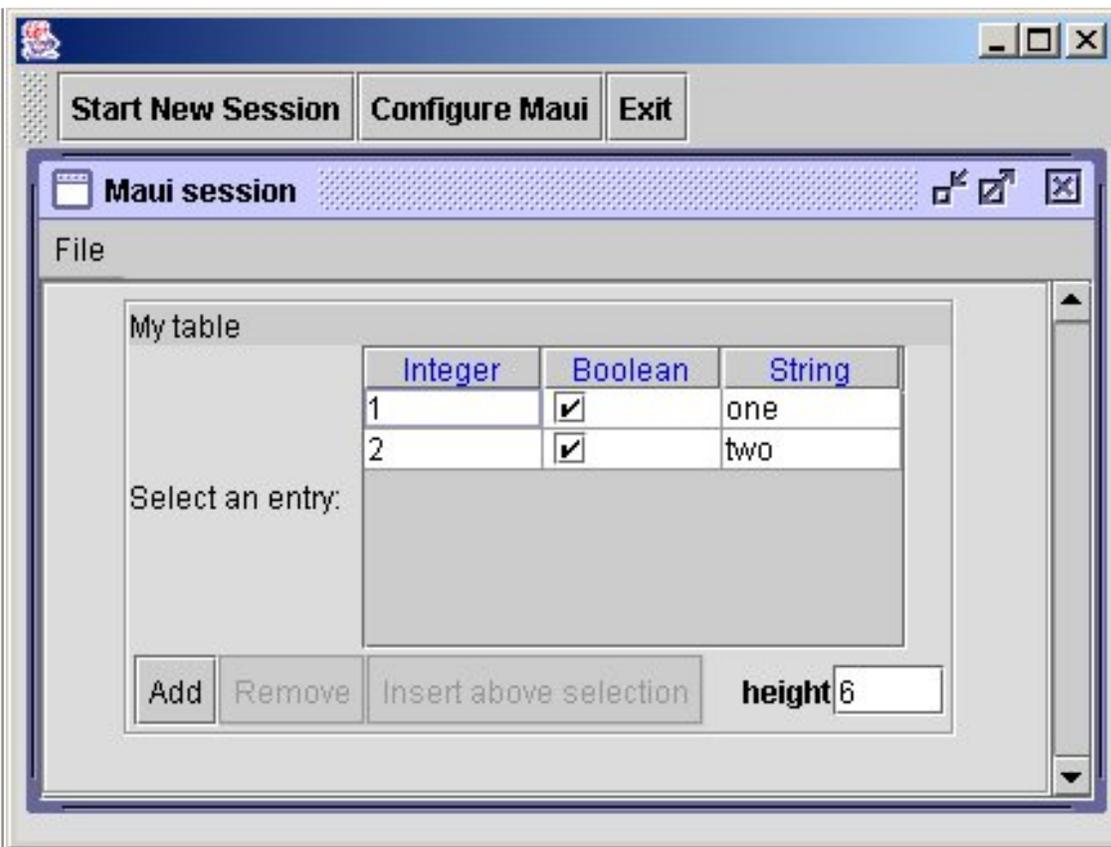
</Maui>

<Header> : If the end-user presses the ADD or INSERT button then the Header class is used fabricate a new row in the table.

```

<Maui RootClass="MyContainer">
    <Class type="MyContainer">
        <Fields>
            <Table name="MyTable" label="My table">
                <Header name="columns" label="columns">
                    <Integer name="Col1" label="Integer" />
                    <Boolean name="Col2" label="Boolean" />
                    <String name="Col3" label="String"/>
                </Header>
                <Entries>
                    <Entry name="entry1">
                        <Cell field="Col1" value="1"/>
                        <Cell field="Col2" value="true"/>
                        <Cell field="Col3" value="one"/>
                    </Entry>
                    <Entry name="entry2">
                        <Cell field="Col1" value="2"/>
                        <Cell field="Col2" value="true"/>
                        <Cell field="Col3" value="two"/>
                    </Entry>
                </Entries>
            </Table>
        </Fields>
    </Class>
</Maui>

```



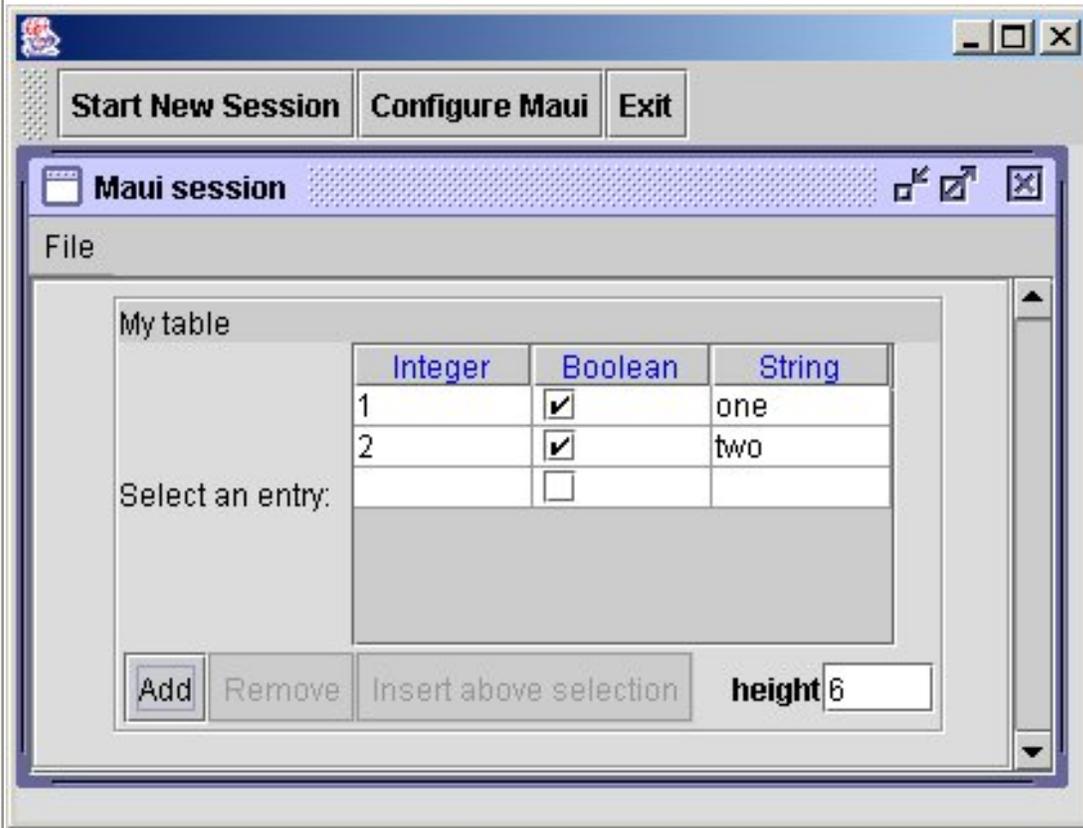
<Entries> : The contents of the table when the table is first rendered on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="two"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```

    </Table>
  </Fields>
</Class>
</Maui>

```



<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>

```

```

    <Entry name="entry2">
      <Cell field="Col1" value="2"/>
      <Cell field="Col2" value="true"/>
      <Cell field="Col3" value="two"/>
    </Entry>
  </Entries>

```

```

<AppData>
  <parameter1>one</parameter1>
  <parameter2>two</parameter2>
</AppData>

```

```

    </Table>
  </Fields>
</Class>
</Maui>

```

<Help> : Display helpful info

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="two"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>

```

```

<Help>

```

```

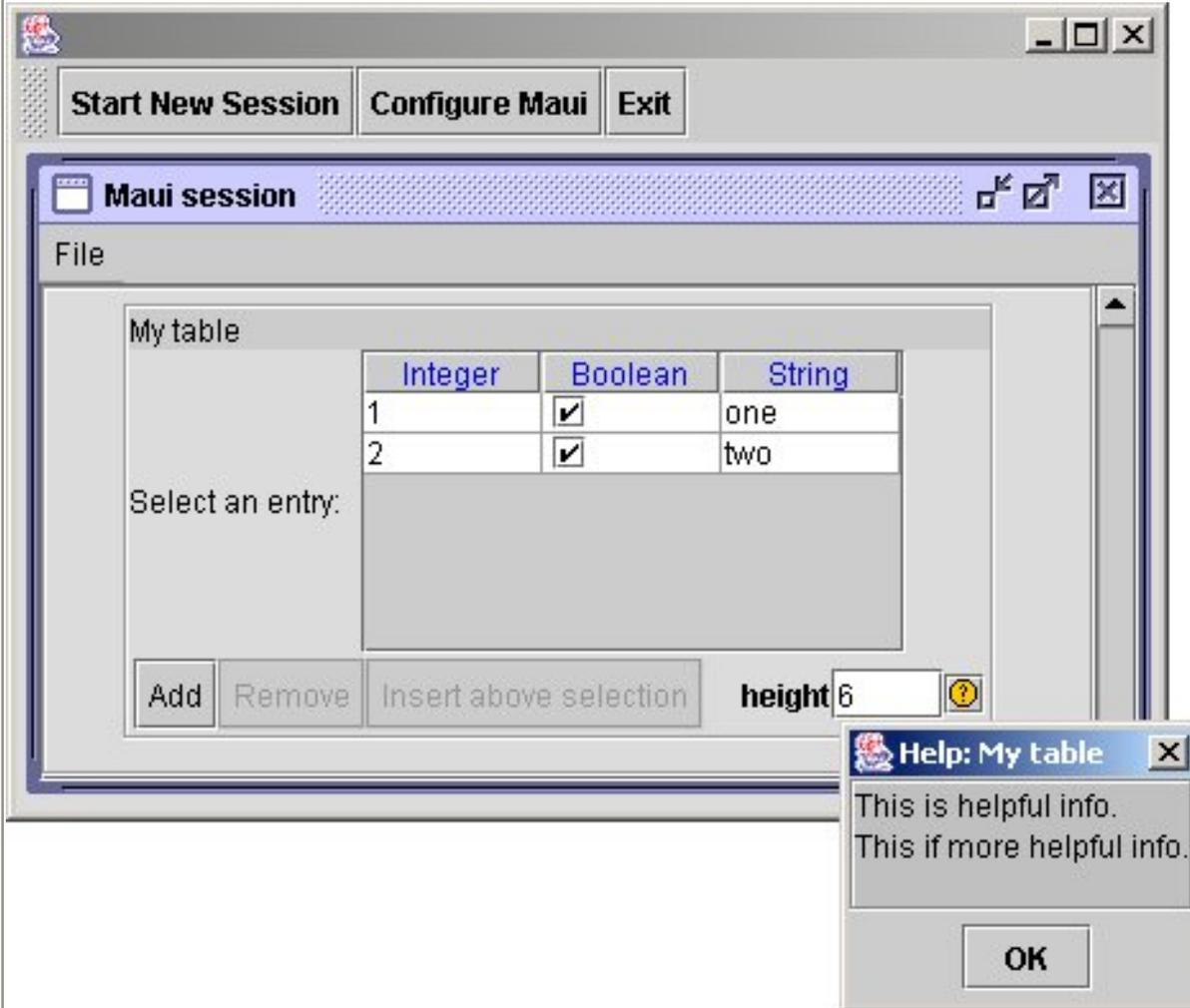
This is helpful info.
This if more helpful info.
</Help>

```

```

</Table>
</Fields>
</Class>
</Maui>

```



Next Up Previous

Next: [B.13.2 Attributes allowed in Up: B.13 Tag Table Previous: B.13 Tag Table](#)

[Next](#) [Up](#) [Previous](#)
Next: [B.14 Tag Reference](#) **Up:** [B.13 Tag Table](#) **Previous:** [B.13.1 Children allowed in](#)

B.13.2 Attributes allowed in Table elements

| Attribute name | Mandatory | Allowed values | Description and Examples comments | |
|----------------------|-----------|----------------|--|-------------------------|
| name | yes | any legal name | name for this variable | example |
| label | no | any string | string to be used as a descriptive label | example |
| minEntries | no | an integer | specifies the minimum number of rows to allow in the Table | example |
| maxEntries | no | an integer | specifies the maximum number of rows to allow in the Table | example |
| fixedNumberOfEntries | no | an integer | specifies that the table should contain this exact number of rows | example |
| minRows | no | an integer | specifies the minimum number of rows to allow in the Table | example |
| maxRows | no | an integer | specifies the maximum number of rows to allow in the Table | example |
| fixedNumberOfRows | no | an integer | specifies that the table should contain this exact number of rows | example |
| collapsible | no | true or false | The panel for this table can be expanded and collapsed with a toggle button. If no value is given, collapsible is assumed to be false. | example |

| | | | | |
|----------------|----|---------------|---|-------------------------|
| beginCollapsed | no | true or false | If the panel is collapsible, this will give the initial state of that panel. By default, a collapsible panel starts out expanded. | example |
| selectionLabel | no | any string | The label to display next to the selection table of the Table. | example |
| tooltip | no | any string | When the end-user moves the mouse over the label then display a tooltip. | example |
| helpMessage | no | any string | Display a HELP icon. Whenever the end-user clicks on the icon, a helpful message pops up on the screen. | example |
| visible | no | true or false | Is the textbox visible on the screen. | example |

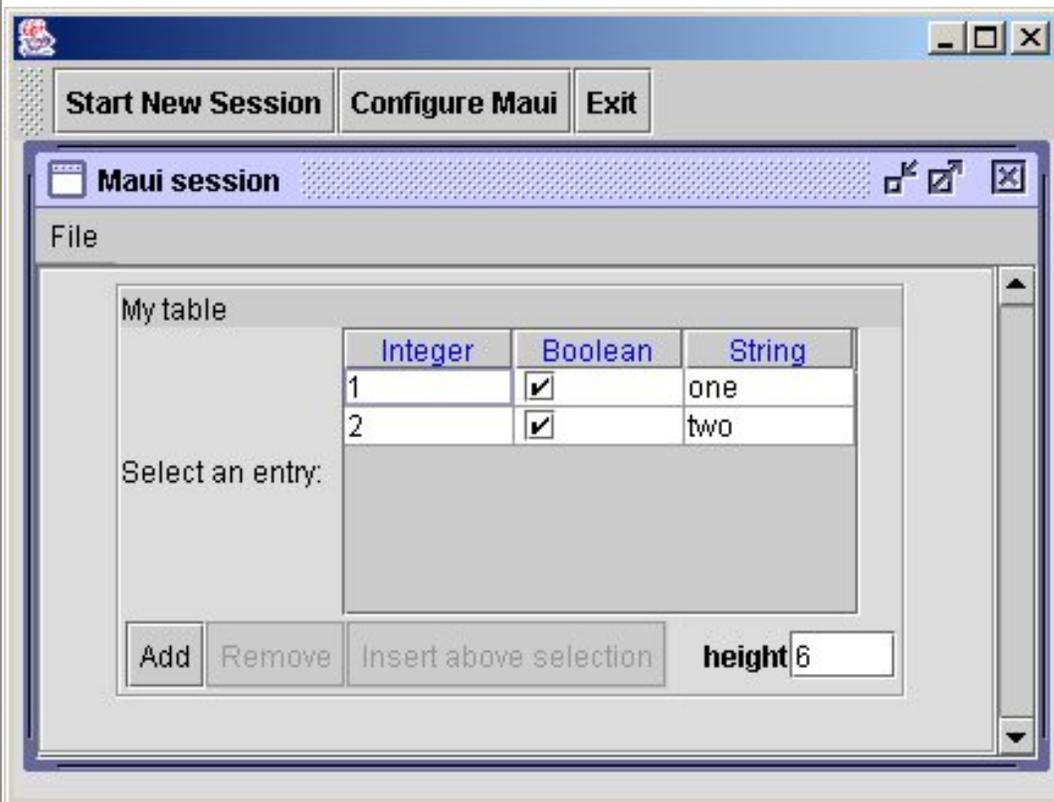
```
<Table>
```

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```

        <Cell field="Col3" value="two"/>
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

```



name: The name of the table

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>

```

```

    </Entry>
    <Entry name="entry2">
      <Cell field="Col1" value="2"/>
      <Cell field="Col2" value="true"/>
      <Cell field="Col3" value="two"/>
    </Entry>
  </Entries>
</Table>
</Fields>
</Class>
</Maui>

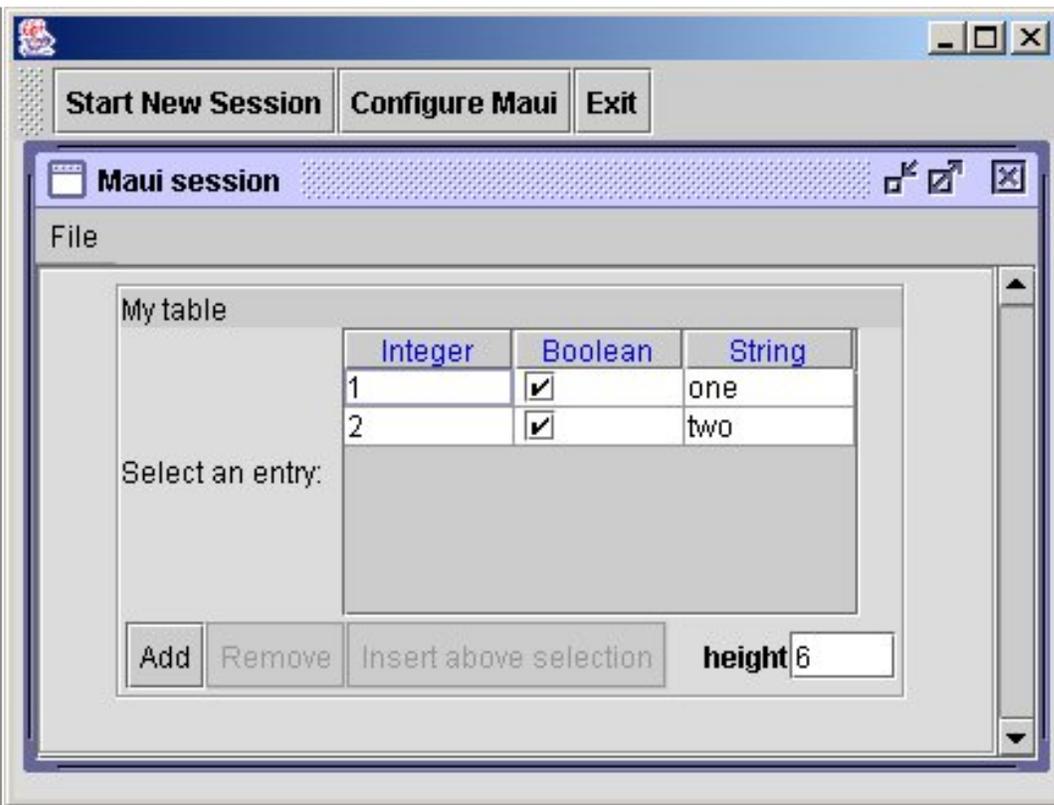
```

label: The text that appears above the table

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>

```



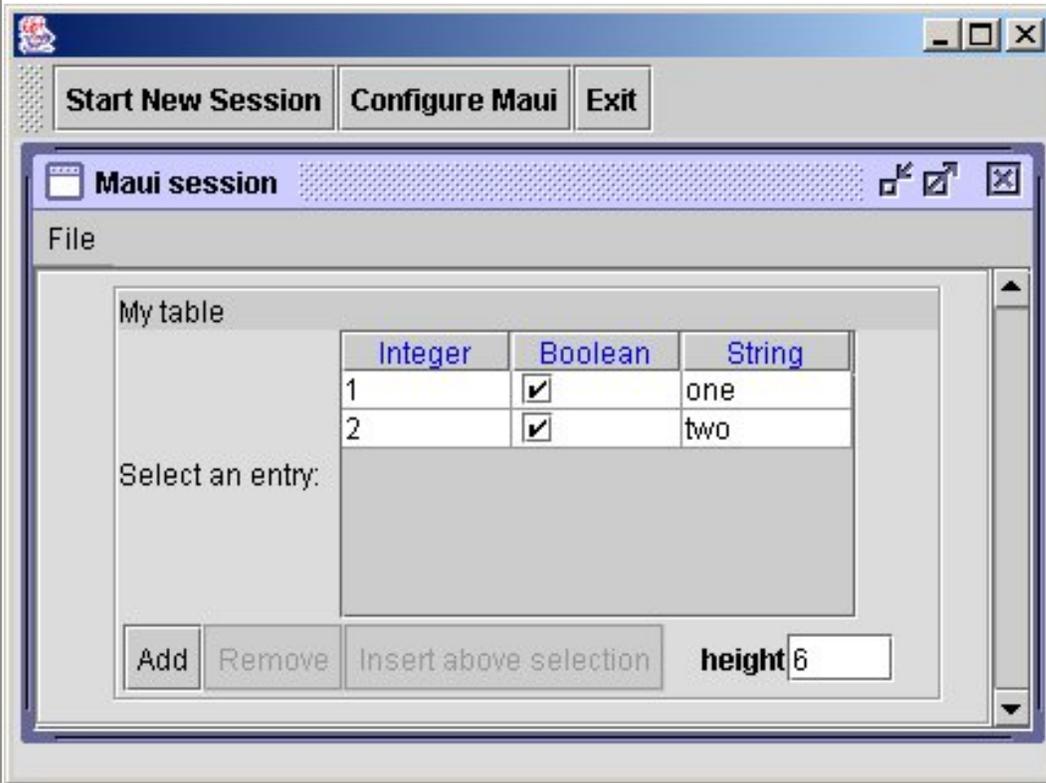
minEntries: The smallest number of rows this table can have

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table" minEntries="2">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="two"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```

    </Fields>
  </Class>
</Maui>

```



maxEntries: The largest number of rows this table can have

```

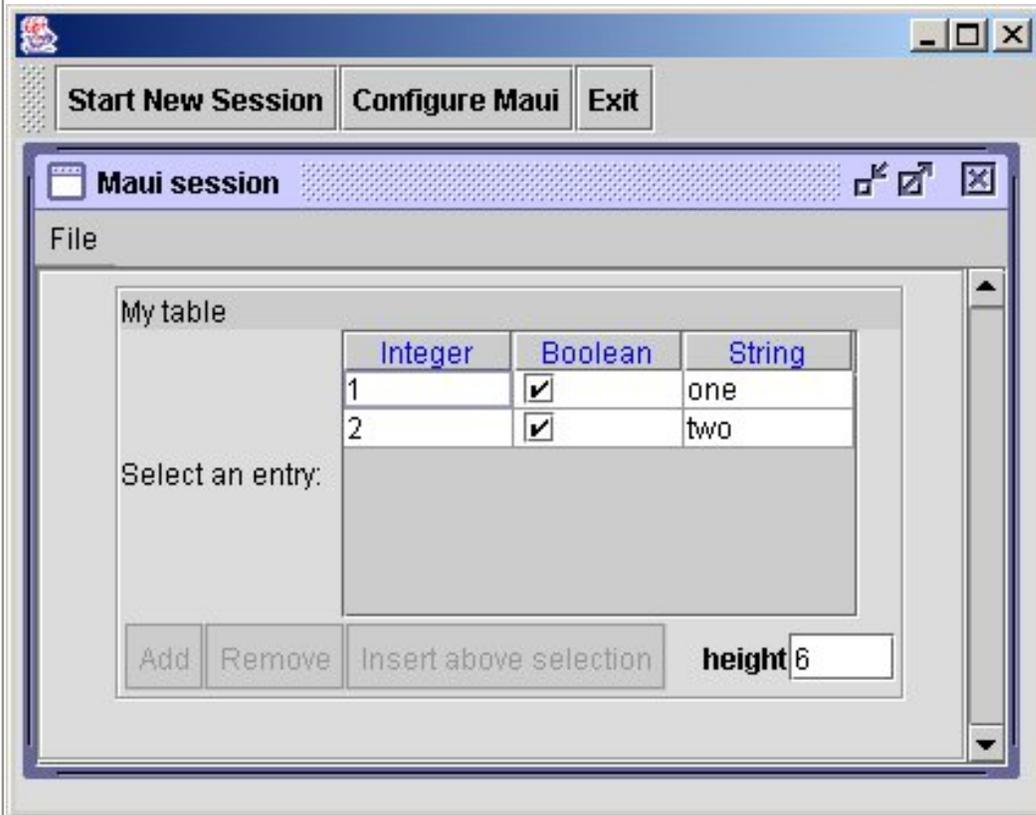
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table" maxEntries="2">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="two"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>

```

```

        </Entry>
    </Entries>
</Table>
</Fields>
</Class>
</Maui>

```



`fixedNumberOfEntries`: The table always contains this number of rows

```

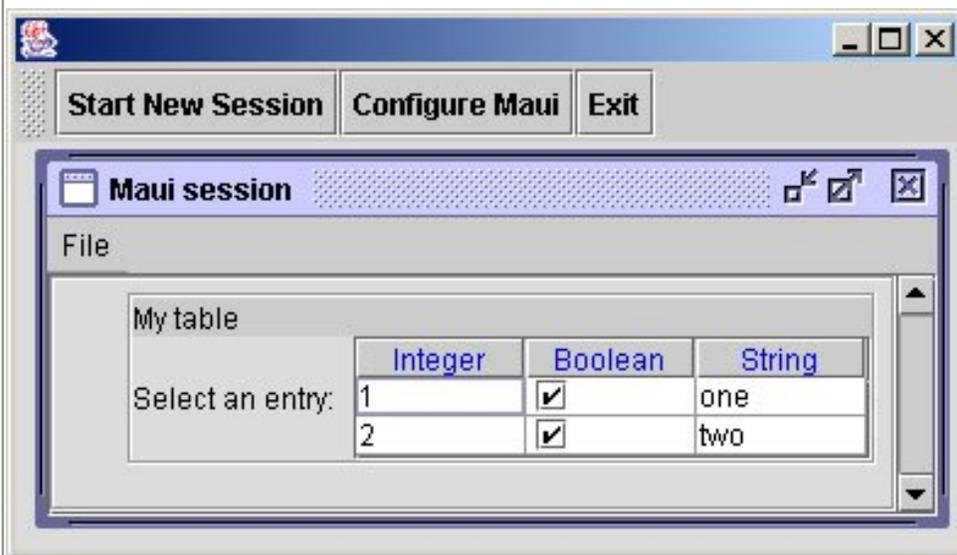
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table" fixedNumberOfEntries="2">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">

```

```

        <Cell field="Col1" value="2" />
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="two" />
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

```



minRows: The smallest number of rows this table can have

```

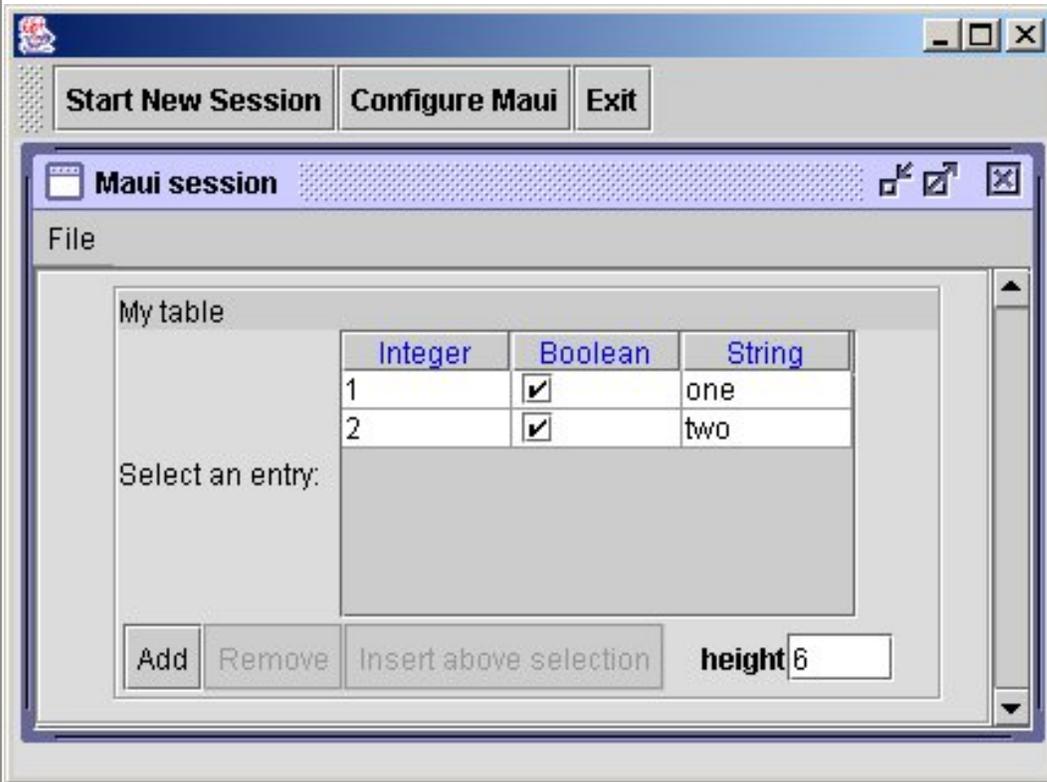
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table" minRows="2">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>

```

```

        </Entry>
    </Entries>
</Table>
</Fields>
</Class>
</Maui>

```



maxRows: The largest number of rows this table can have

```

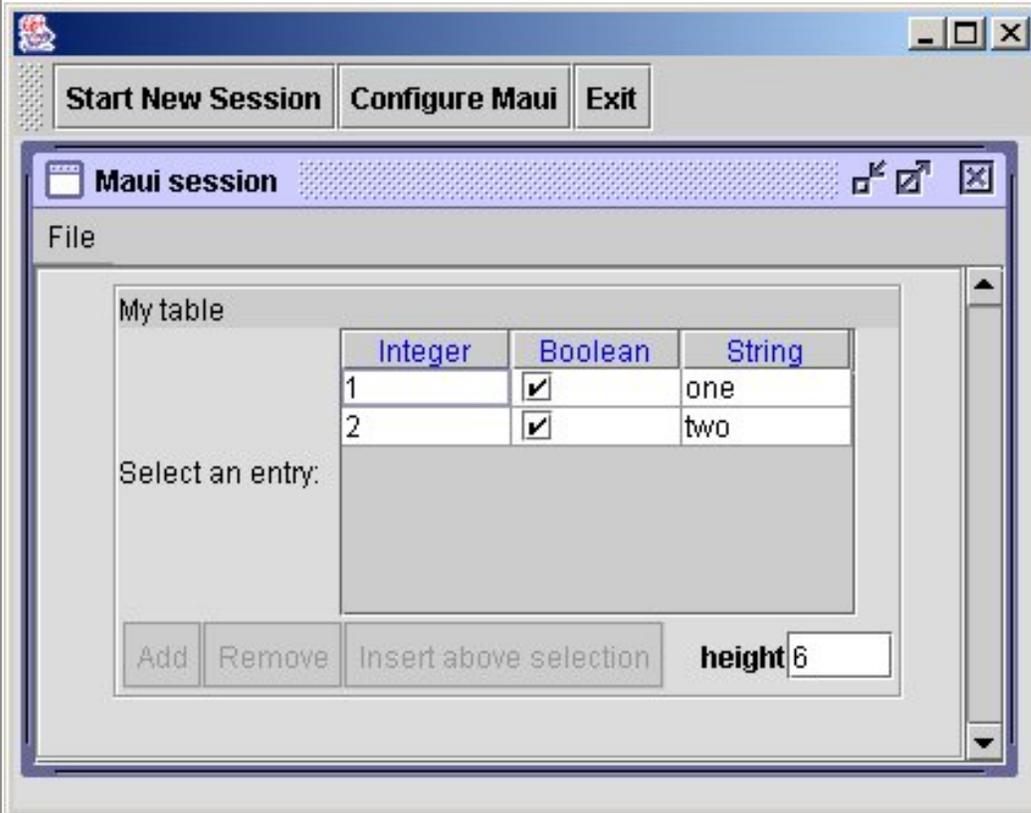
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table" maxRows="2">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">

```

```

        <Cell field="Col1" value="2" />
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="two" />
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

```



`fixedNumberOfRows`: The table always contains this number of rows

```

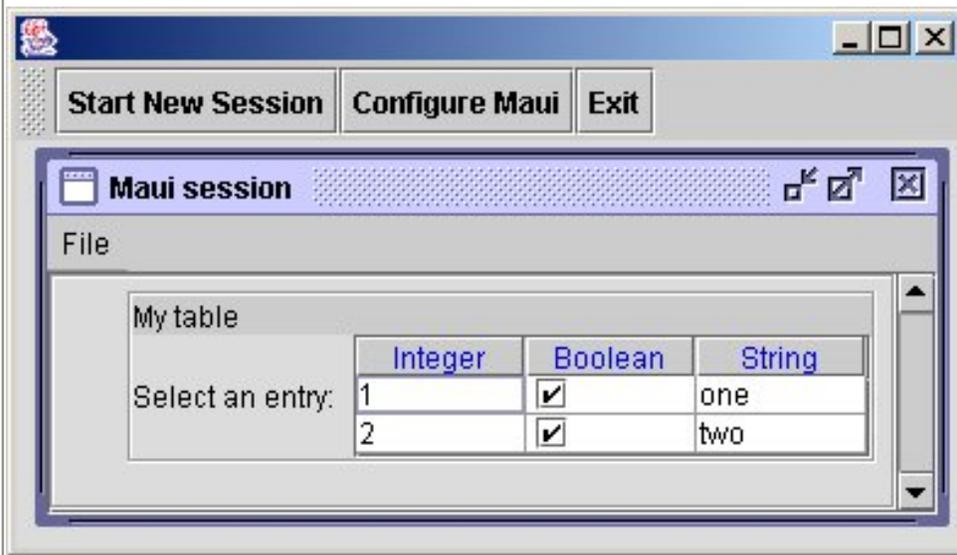
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table" fixedNumberOfRows="2">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />

```

```

        <Cell field="Col3" value="one"/>
    </Entry>
    <Entry name="entry2">
        <Cell field="Col1" value="2"/>
        <Cell field="Col2" value="true"/>
        <Cell field="Col3" value="two"/>
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

```



collapsible: Can the end-user hide the table by collapsing the panel

```

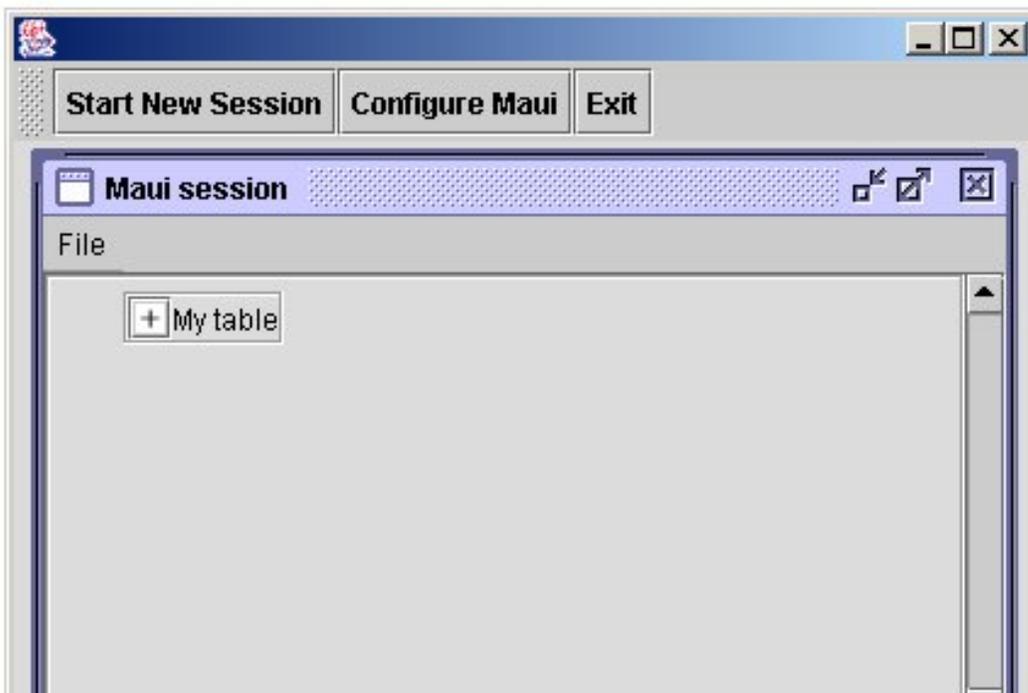
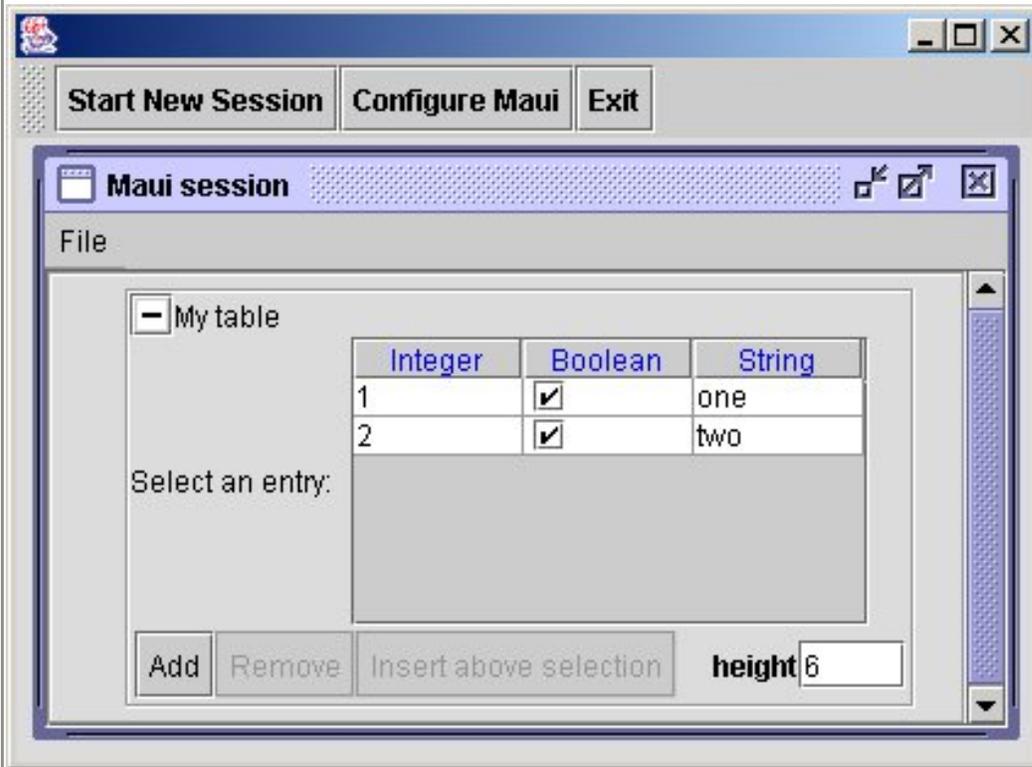
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table" collapsible="true">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
          <Entry name="entry2">

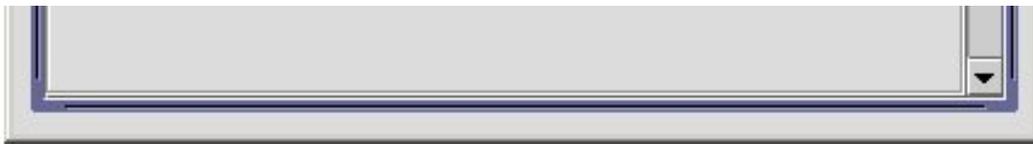
```

```

        <Cell field="Col1" value="2"/>
        <Cell field="Col2" value="true"/>
        <Cell field="Col3" value="two"/>
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

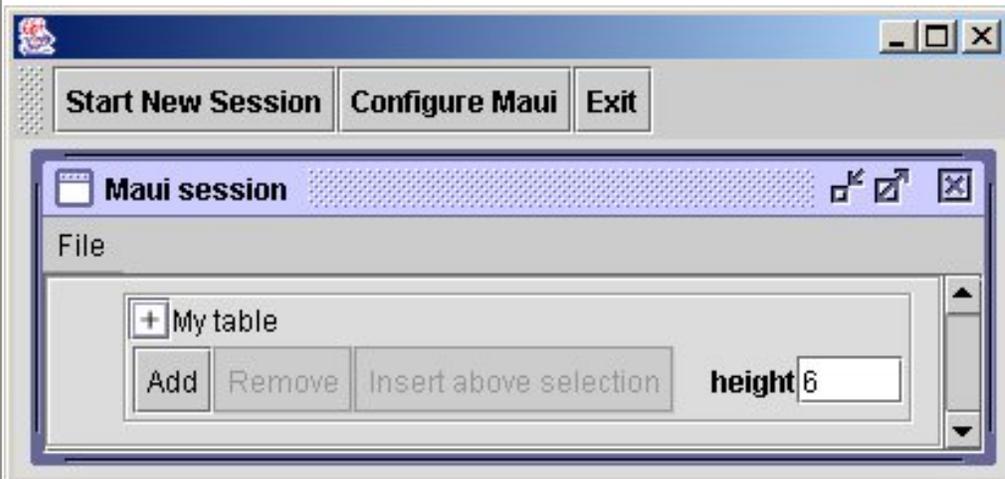
```





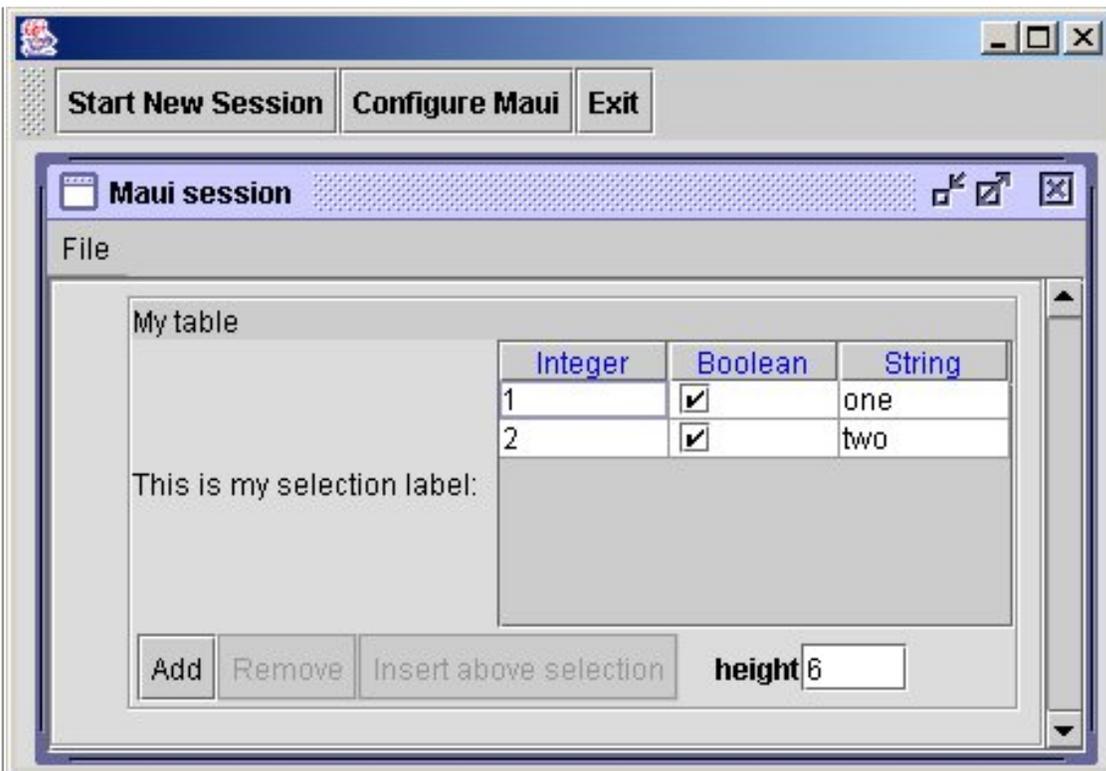
`beginCollapsed`: Is the table panel collapsed?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table"
        collapsible="true" beginCollapsed="true">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```



selectionLabel: The label that appears to the left of the table

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table"
        selectionLabel="This is my selection label">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```



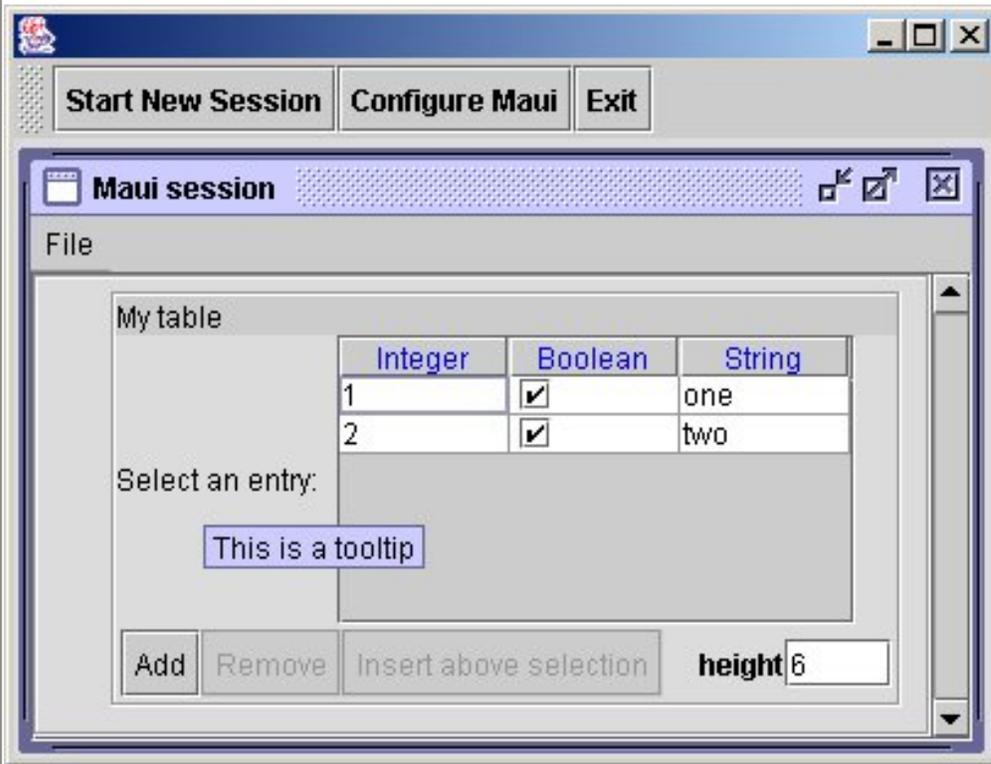
tooltip: Assign a tooltip to the label

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table"
        tooltip="This is a tooltip">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="one"/>
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2"/>
            <Cell field="Col2" value="true"/>
            <Cell field="Col3" value="two"/>
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```

</Class>
</Maui>

```



helpMessage: Display help

```

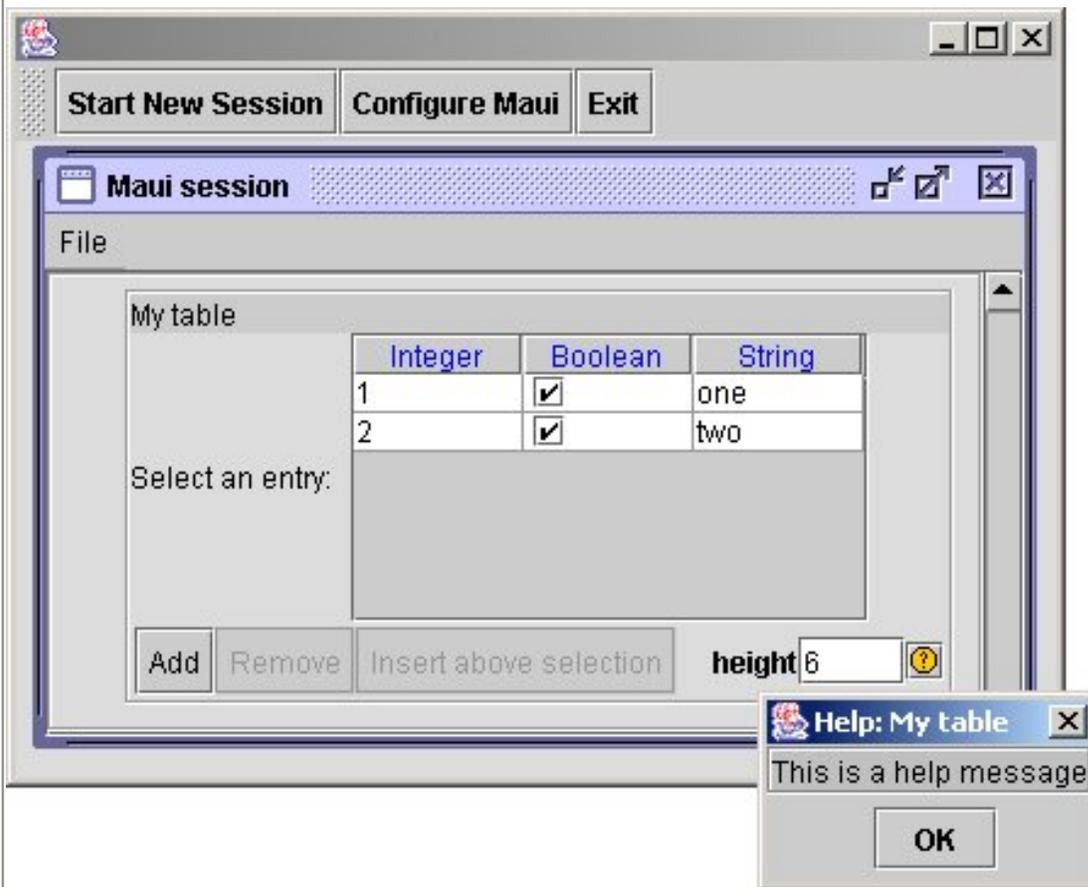
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table"
        helpMessage="This is a help message">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>

```

```

        <Cell field="Col3" value="two"/>
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

```



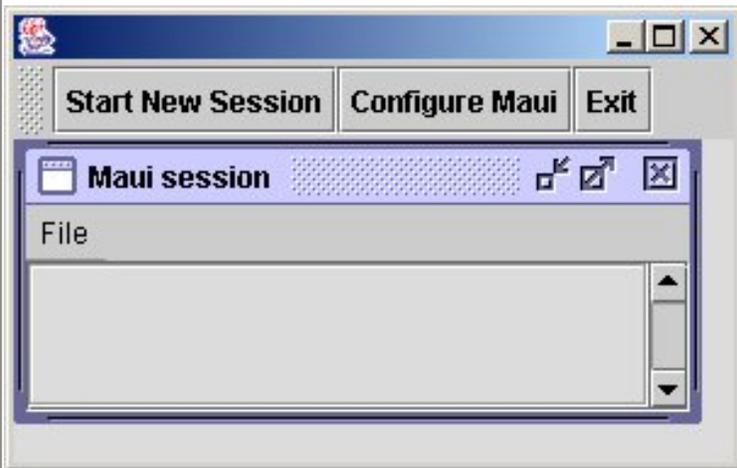
visible: Is the textbox visible on the screen?

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table"
        visible="false">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>

```

```
<Entry name="entry1">
  <Cell field="Col1" value="1"/>
  <Cell field="Col2" value="true"/>
  <Cell field="Col3" value="one"/>
</Entry>
<Entry name="entry2">
  <Cell field="Col1" value="2"/>
  <Cell field="Col2" value="true"/>
  <Cell field="Col3" value="two"/>
</Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.14 Tag Reference](#) **Up:** [B.13 Tag Table](#) **Previous:** [B.13.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.14.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.13.2 Attributes allowed in](#)

B.14 Tag Reference

Reference elements refers to a field entered elsewhere in the Maui GUI.

Subsections

- [B.14.1 Children allowed in Reference elements](#)
 - [B.14.2 Attributes allowed in Reference elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.14.2 Attributes allowed in](#) **Up:** [B.14 Tag Reference](#) **Previous:** [B.14 Tag Reference](#)

B.14.1 Children allowed in Reference elements

| Tag | Number | Description and comments | Examples |
|--------------|--------|--|-------------------------|
| CustomEditor | 0-1 | Allows the Reference to use a non-standard editor. | example |
| AppData | 0-1 | a free form block of XML used for data where no editor is needed | example |

<Reference> : Select one choice from a collection of values (such as an array or a table).

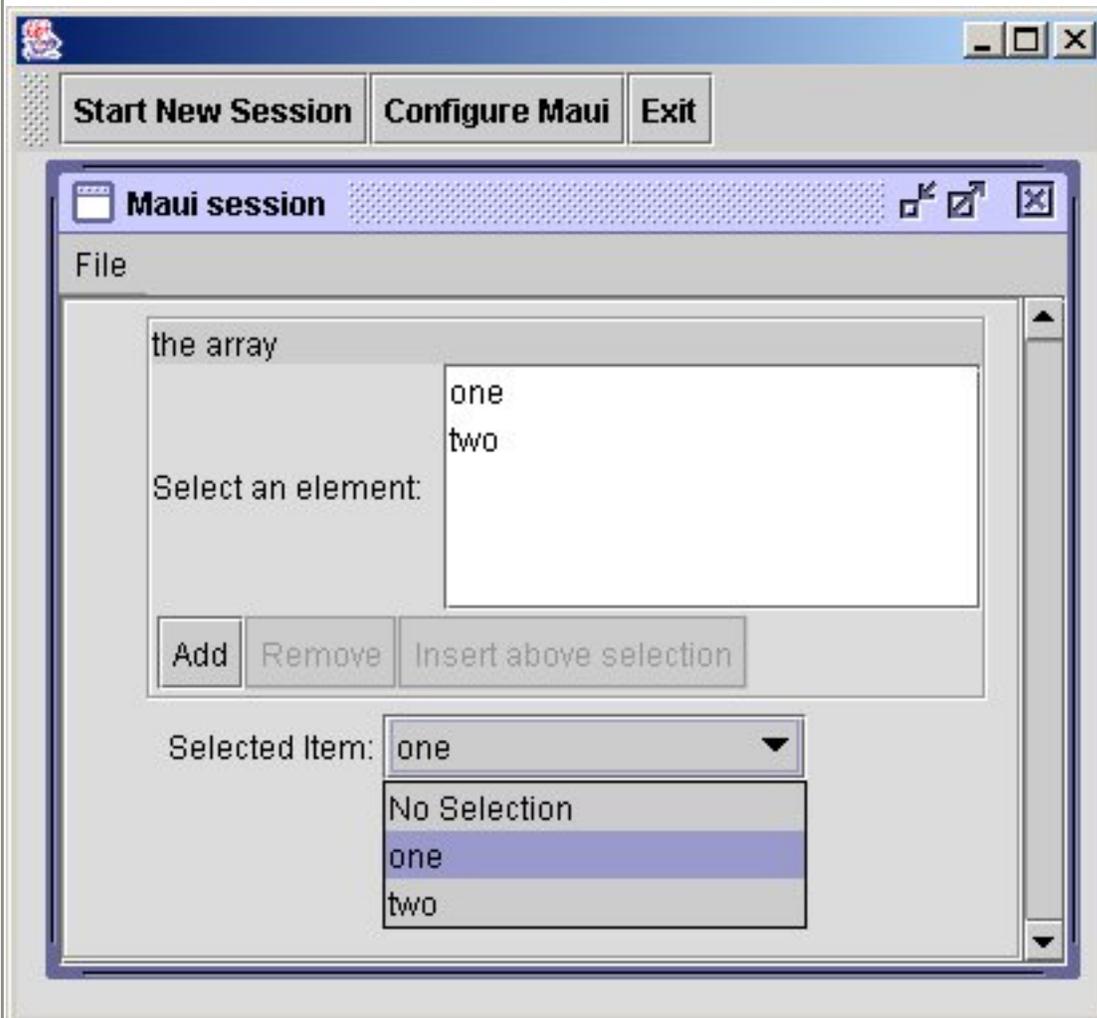
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>

      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
      <Contents>
        <Item>
          <Class type="row1">
            <Fields>
              <String name="string"
                label="label"
                default="one"/>
            </Fields>
          </Class>
        </Item>
```

```
<Item>
  <Class type="row2">
    <Fields>
      <String name="string"
        label="label"
        default="two" />
    </Fields>
  </Class>
</Item>
</Contents>
</Array>
```

```
<Reference name="myReference" path="myArray" />
```

```
</Fields>
</Class>
</Maui>
```



<CustomEditor> : Replace the array editor with your own custom editor

```
<Reference name="myReference" path="myArray">
  <CustomEditor name="MyReferenceEditor">
    <parameter1>one</parameter1>
    <parameter2>two</parameter2>
  </CustomEditor>
</Reference>
```

<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Reference name="myReference" path="myArray">
  <AppData>
    <parameter1>one</parameter1>
    <parameter2>two</parameter2>
  </AppData>
</Reference>
```

[Next](#) [Up](#) [Previous](#)

Next: [B.14.2 Attributes allowed in](#) **Up:** [B.14 Tag Reference](#) **Previous:** [B.14 Tag Reference](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.15 Tag Comment](#) **Up:** [B.14 Tag Reference](#) **Previous:** [B.14.1 Children allowed in](#)

B.14.2 Attributes allowed in Reference elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|-----------------------|--|-------------------------|
| name | yes | any legal name | name for this variable | example |
| label | no | any string | string to be used as a descriptive label | example |
| path | yes | valid reference path | the path to the referenced element | example |
| output | no | reference/dereference | indicates whether this element will reference or dereference the element. If no value is given, it is assumed to reference the element | example |
| optional | no | true/false | indicates if a value needs to be selected | example |
| insets | no | true or false | If true then surround the GUI component with a lot of white space. | example |

<Reference> : Select one choice from a collection of values (such as an array or a table).

```

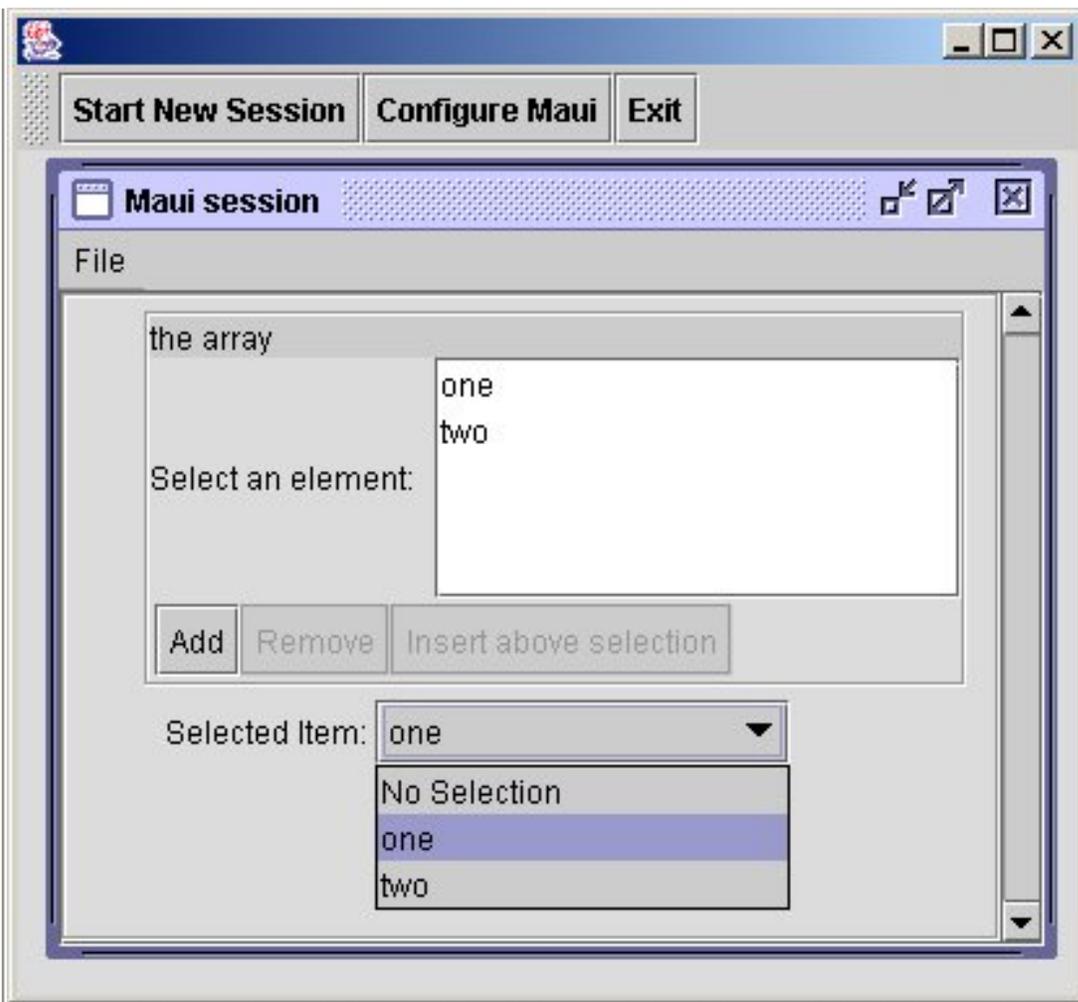
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>

      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>
                <String name="string"
                  label="label"
                  default="two"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>

      <Reference name="myReference" path="myArray"/>

    </Fields>
  </Class>
</Maui>

```



name: The name of the reference

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Reference name="myReference" path="myArray"/>
    </Fields>
  </Class>
</Maui>
```

label: The text that appears to the left of the reference.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Reference name="myReference"
        path="myArray" label="my label"/>
    </Fields>
  </Class>
</Maui>
```



path: The name of the component (array, table, etc.) that is supplying the contents of this reference.

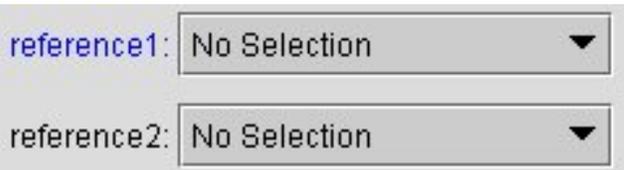
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Reference name="myReference"
        path="myArray" label="my label"/>
    </Fields>
  </Class>
</Maui>
```



output: not implemented

optional: If true then the end-user does not have to select an entry.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Reference name="reference1" label="reference1"
        path="myArray" optional="true"/>
      <Reference name="reference2" label="reference2"
        path="myArray" optional="false"/>
    </Fields>
  </Class>
</Maui>
```



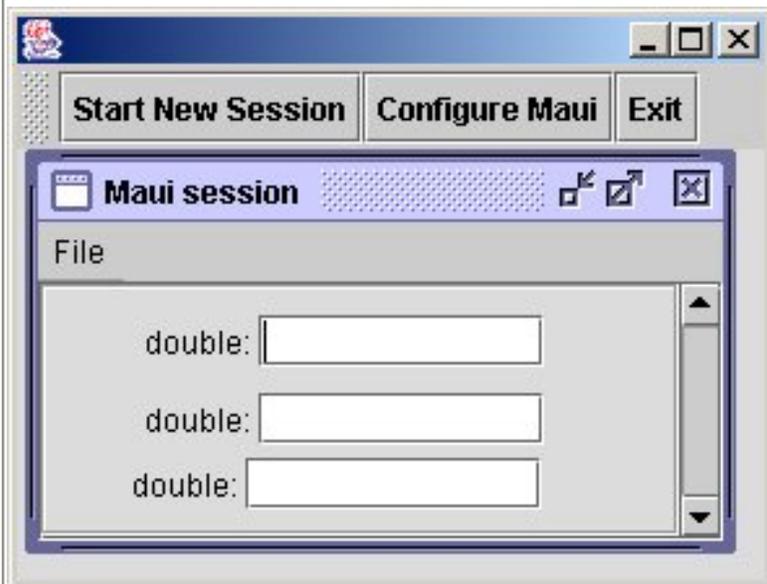
insets: Is there a lot of white space surrounding the pull-down menu?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
        </Contents>
      </Array>
    </Fields>
  </Class>
</Maui>
```

```
<Item>
  <Class type="row2">
    <Fields>
      <String name="string"
        label="label"
        default="two"/>
    </Fields>
  </Class>
</Item>
</Contents>
</Array>

<Reference name="reference1" path="myArray" />
<Reference name="reference2" path="myArray" insets="true" />
<Reference name="reference3" path="myArray" insets="false" />

</Fields>
</Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.15 Tag Comment](#) **Up:** [B.14 Tag Reference](#) **Previous:** [B.14.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.15.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.14.2 Attributes allowed in](#)

B.15 Tag Comment

Comment elements will cause the content of that element to be displayed

Subsections

- [B.15.1 Children allowed in Comment elements](#)
 - [B.15.2 Attributes allowed in Comment elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.15.2 Attributes allowed in](#) **Up:** [B.15 Tag Comment](#) **Previous:** [B.15 Tag Comment](#)

B.15.1 Children allowed in Comment elements

[None](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.15.2 Attributes allowed in](#) **Up:** [B.15 Tag Comment](#) **Previous:** [B.15 Tag Comment](#)

[Next](#) [Up](#) [Previous](#)

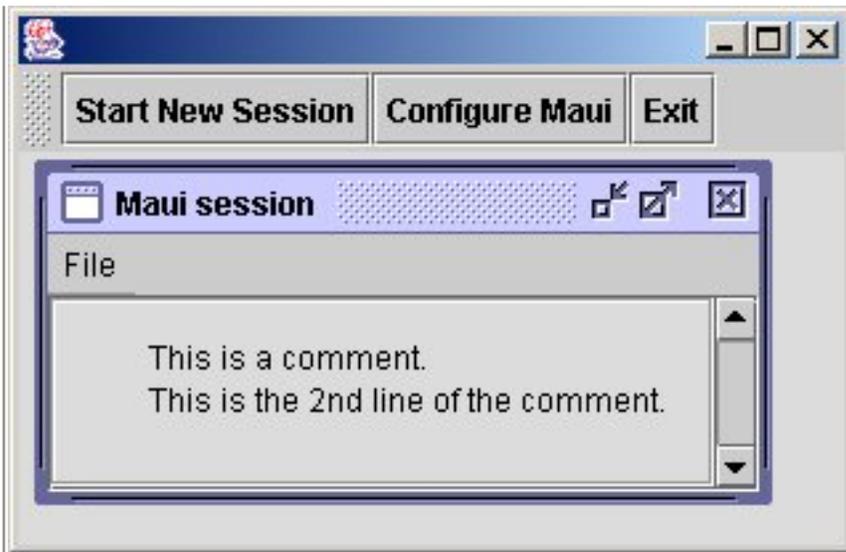
Next: [B.16 Tag Menu](#) **Up:** [B.15 Tag Comment](#) **Previous:** [B.15.1 Children allowed in](#)

B.15.2 Attributes allowed in Comment elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|----------------|---|-------------------------|
| name | no | any legal name | name for this variable | example |
| visible | no | true or false | If visible is true, the content of this Comment will be displayed in Maui. If false, the content is not displayed. By default, visible is assumed to be true. | example |
| border | no | true or false | If border is true then a border is drawn around the comment. | example |

`<Comment>` : display a comment on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Comment>
        This is a comment.
        This is the 2nd line of the comment.
      </Comment>
    </Fields>
  </Class>
</Maui>
```

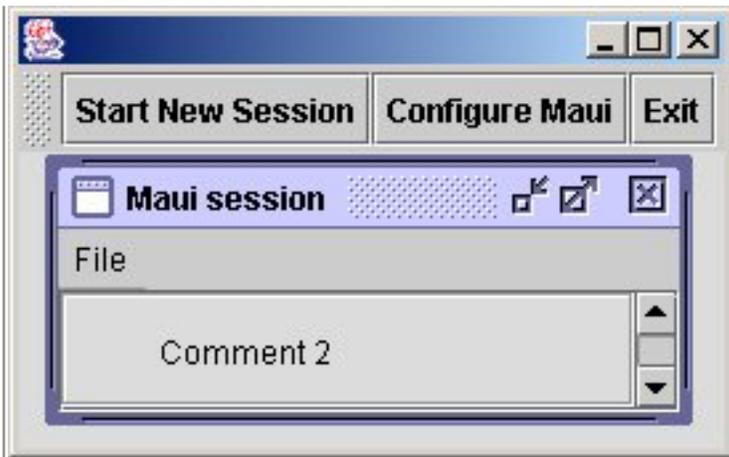


name: The name of the comment

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Comment name="myComment">
        This is a comment.
        This is the 2nd line of the comment.
      </Comment>
    </Fields>
  </Class>
</Maui>
```

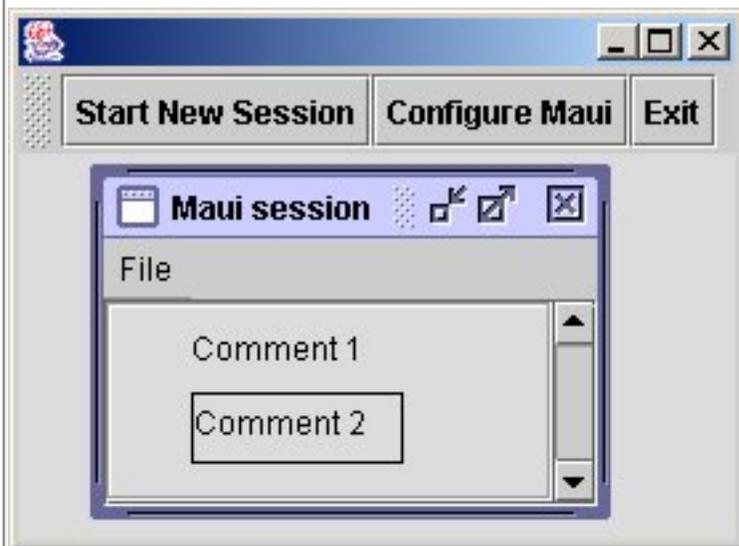
visible: Is the comment visible on the screen?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Comment visible="false">Comment 1 </Comment>
      <Comment visible="true">Comment 2 </Comment>
    </Fields>
  </Class>
</Maui>
```



border: Determines if a border is drawn around the class.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Comment border="false">Comment 1 </Comment>
      <Comment border="true">Comment 2 </Comment>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.16 Tag Menu](#) **Up:** [B.15 Tag Comment](#) **Previous:** [B.15.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.16.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.15.2 Attributes allowed in](#)

B.16 Tag Menu

Menu provides a options list selection for a `String` element

Subsections

- [B.16.1 Children allowed in Menu elements](#)
 - [B.16.2 Attributes allowed in Menu elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.16.2 Attributes allowed in](#) **Up:** [B.16 Tag Menu](#) **Previous:** [B.16 Tag Menu](#)

B.16.1 Children allowed in Menu elements

None

[Next](#) [Up](#) [Previous](#)

Next: [B.16.2 Attributes allowed in](#) **Up:** [B.16 Tag Menu](#) **Previous:** [B.16 Tag Menu](#)

[Next](#) [Up](#) [Previous](#)

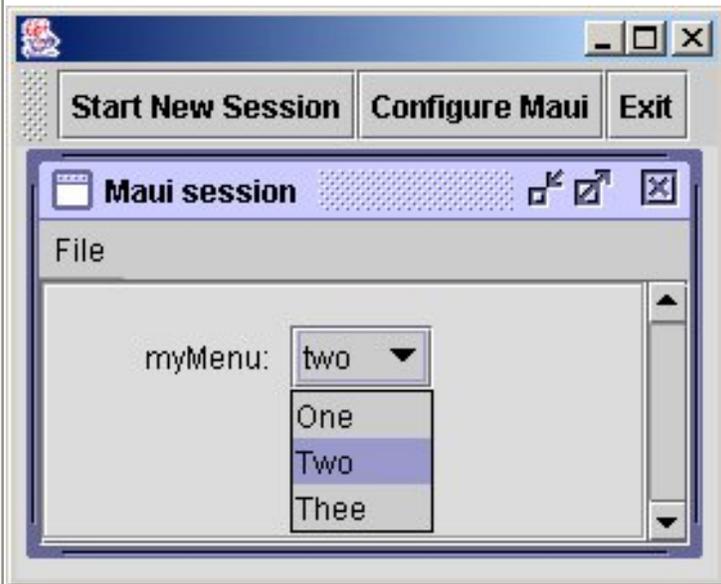
Next: [B.17 Tag Master](#) **Up:** [B.16 Tag Menu](#) **Previous:** [B.16.1 Children allowed in](#)

B.16.2 Attributes allowed in Menu elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|--|--|-------------------------|
| options | yes | any list of strings | lists all possible values that the String can have. The different values are separated by pipes | example |
| style | no | comboBox, radioButton, list | default is comboBox | example |
| rows | no | positive integers | Only affects radio buttons. Specifies the Number of rows to put the buttons in. 0 means all in one column. | example |
| listMode | no | multiple_interval, single_interval, single | this is to be used with lists only. It gives three modes in which items in it can be selected. If multiple items are selected they will be separated by pipes in the output xml. | example |
| mauiAction | no | the name of a java class | This java class is invoked whenever the end-user selects a menu item.. | example |

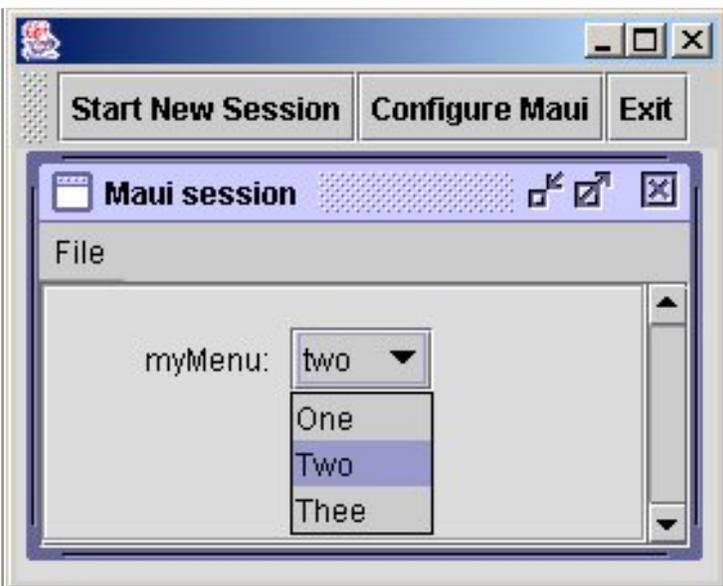
<Menu> : Display a pull-down menu, list, or a group of radio buttons.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myMenu" default="two">
        <Menu options="One |Two |Thee" />
      </String>
    </Fields>
  </Class>
</Maui>
```



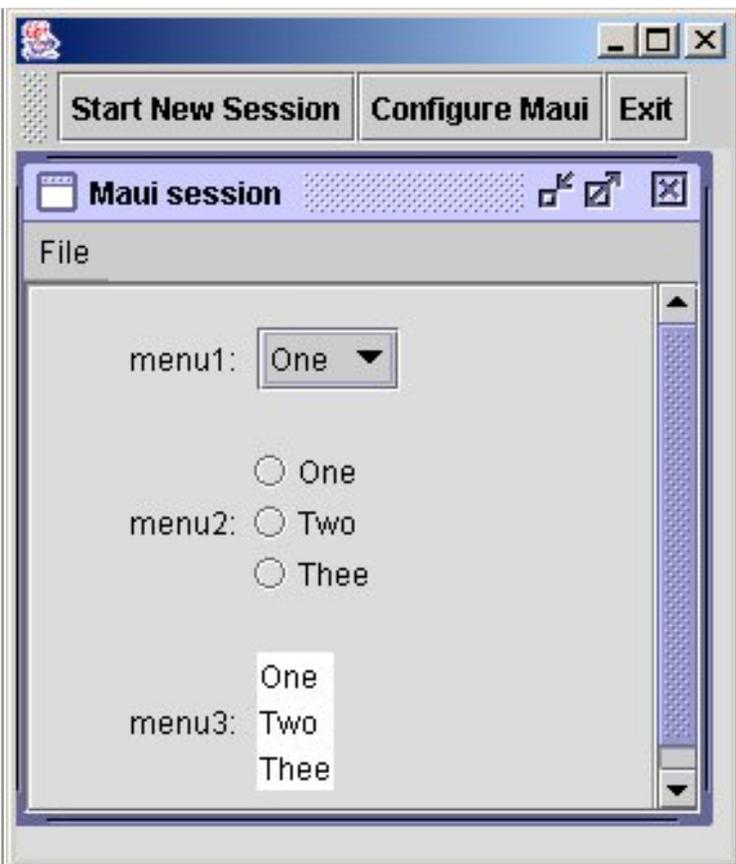
options: The items that appear inside the menu

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myMenu" default="two">
        <Menu options="One |Two |Thee" />
      </String>
    </Fields>
  </Class>
</Maui>
```



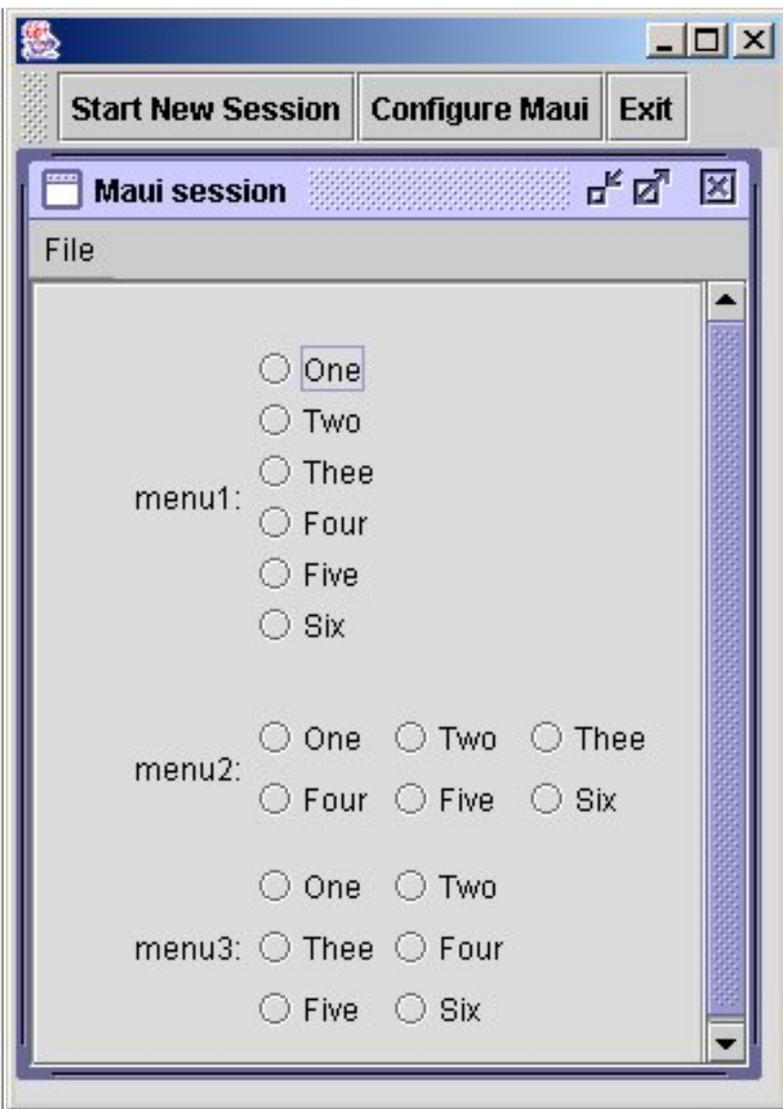
style: The menu can be displayed as a pull-down menu, group of radio buttons, or a list box.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="menu1" label="menu1">
        <Menu options="One|Two|Thee" style="comboBox"/>
      </String>
      <String name="menu2" label="menu2">
        <Menu options="One|Two|Thee" style="radioButton"/>
      </String>
      <String name="menu3" label="menu3">
        <Menu options="One|Two|Thee" style="list"/>
      </String>
    </Fields>
  </Class>
</Maui>
```



rows: The number of rows in a group of radio buttons

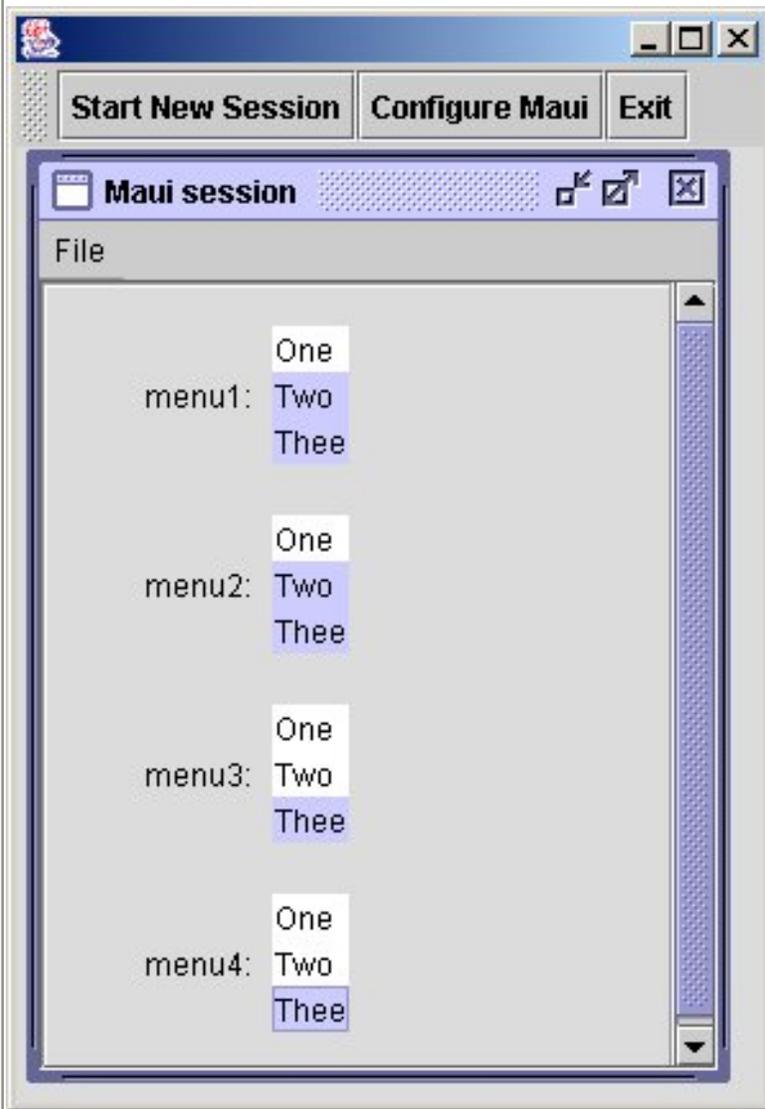
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="menu1" label="menu1">
        <Menu options="One|Two|Thee|Four|Five|Six"
          style="radioButton"/>
      </String>
      <String name="menu2" label="menu2">
        <Menu options="One|Two|Thee|Four|Five|Six"
          style="radioButton" rows="2"/>
      </String>
      <String name="menu3" label="menu3">
        <Menu options="One|Two|Thee|Four|Five|Six"
          style="radioButton" rows="3"/>
      </String>
    </Fields>
  </Class>
</Maui>
```



listMode: Do you want to allow the end-user to select more than one item in a list? If `listMode` is set to `multiple_level` then the end-user can select more than one item. If `listMode` is set to `single_interval` or to `single` then the end-user can select only one item.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="menu1" label="menu1">
        <Menu options="One|Two|Thee" style="list"/>
      </String>
      <String name="menu2" label="menu2">
        <Menu options="One|Two|Thee" style="list"
          listMode="multiple_interval"/>
      </String>
      <String name="menu3" label="menu3">
        <Menu options="One|Two|Thee" style="list">
```

```
listMode="single_interval"/>
</String>
<String name="menu4" label="menu4">
  <Menu options="One|Two|Thee" style="list"
    listMode="single"/>
</String>
</Fields>
</Class>
</Maui>
```



mauiAction: This java class is invoked whenever the end-user changes the contents of the textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <String name="myMenu" default="two">
        <Menu options="One|Two|Thee" mauiAction="myMenuAction"/>
      </String>
    </Fields>
  </Class>
</Maui>
```

[Next](#) [Up](#) [Previous](#)

Next: [B.17 Tag Master](#) **Up:** [B.16 Tag Menu](#) **Previous:** [B.16.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.17.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.16.2 Attributes allowed in](#)

B.17 Tag Master

`Master` provides the template used for all elements added to the array. There must be one and only one child of `Master` to specify the template to use.

Subsections

- [B.17.1 Children allowed in Master elements](#)
 - [B.17.2 Attributes allowed in Master elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.17.2 Attributes allowed in](#) **Up:** [B.17 Tag Master](#) **Previous:** [B.17 Tag Master](#)

B.17.1 Children allowed in Master elements

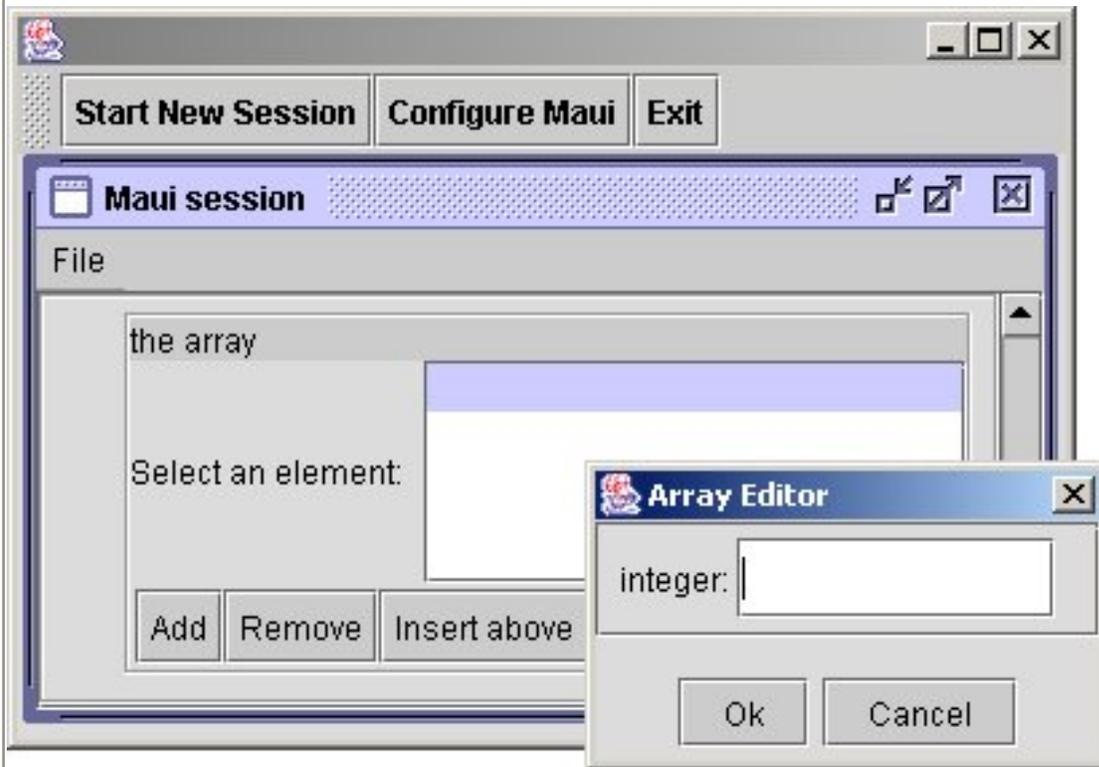
| Tag | Description and comments | Examples |
|------------------|--|-------------------------|
| Integer | the array will contain integer variables | example |
| String | the array will contain string variables | example |
| Double | the array will contain double variables | example |
| Boolean | the array will contain boolean (logical) variables | example |
| Array | the array will contain array variables | example |
| Table | the array will contain table variables | example |
| Reference | the array will contain reference variables | example |
| Comment | the array will contain comments | example |
| <i>type_name</i> | the array will contain class data members of type <i>type_name</i> | example |
| Class | zero or more child classes. | example |

<Master> : Contains the GUI components that are created whenever the end-user creates a new array element.

```

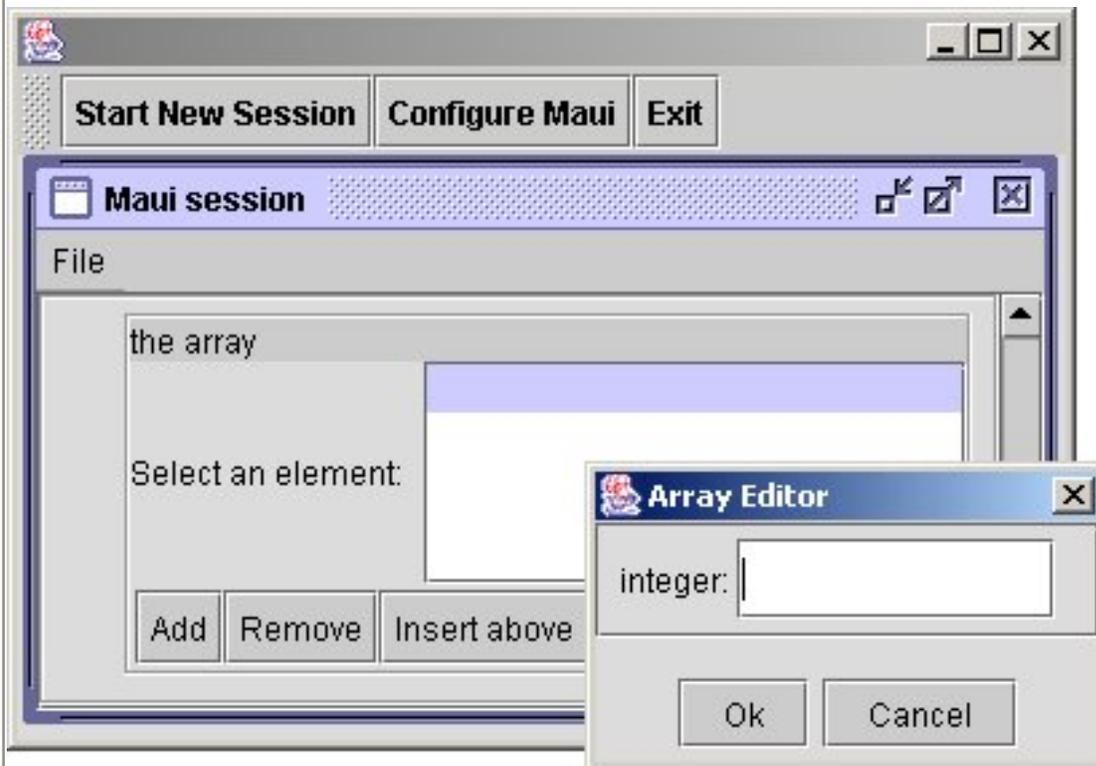
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Class type="master" label="label">
            <Fields>
              <Integer name="integer" label="integer"/>
            </Fields>
          </Class>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>

```



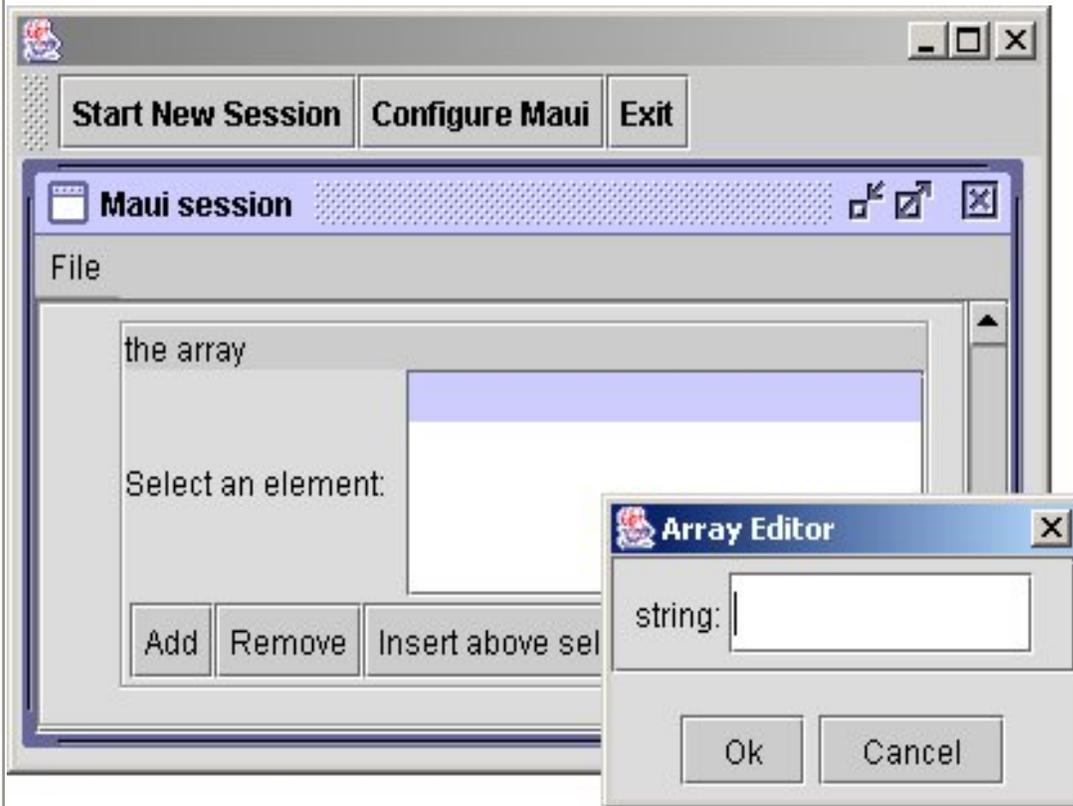
<Integer> : Insert an integer into the array element

```
<Maui RootClass="MyContainer">  
  <Class type="MyContainer" label="the container">  
    <Fields>  
      <Array name="myArray" label="the array">  
        <Master>  
          <Integer name="integer" label="integer"/>  
        </Master>  
      </Array>  
    </Fields>  
  </Class>  
</Maui>
```



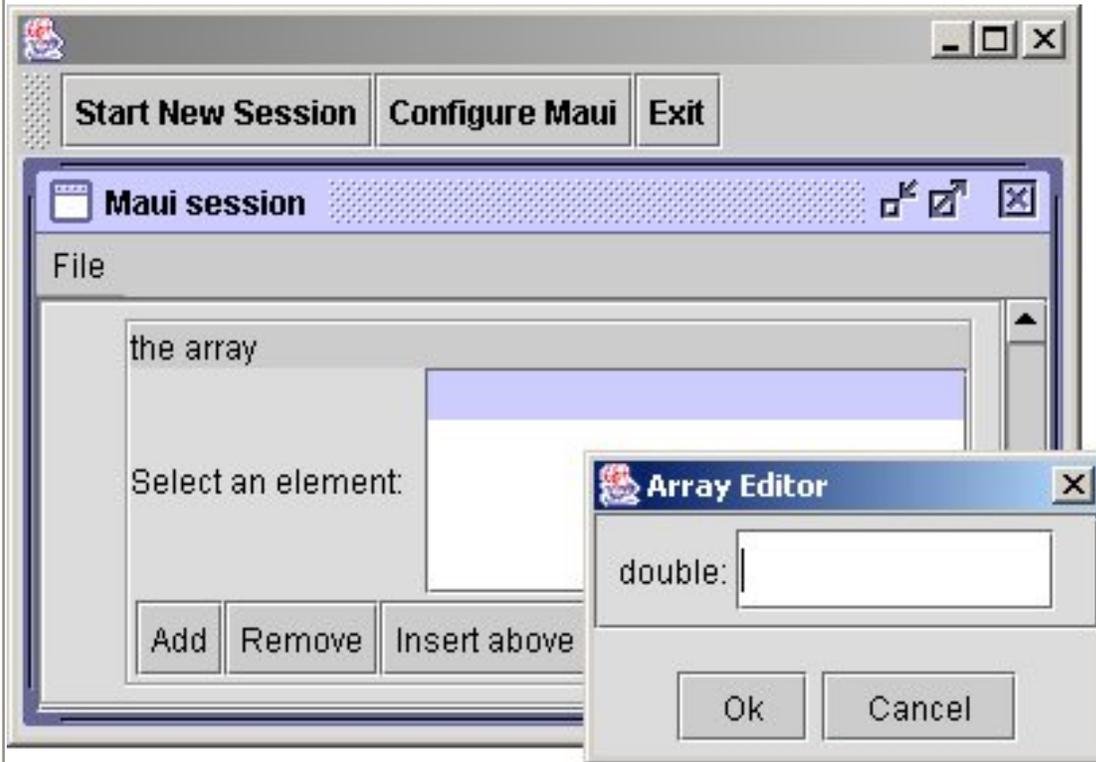
<String> : Insert a string into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <String name="string" label="string"/>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```



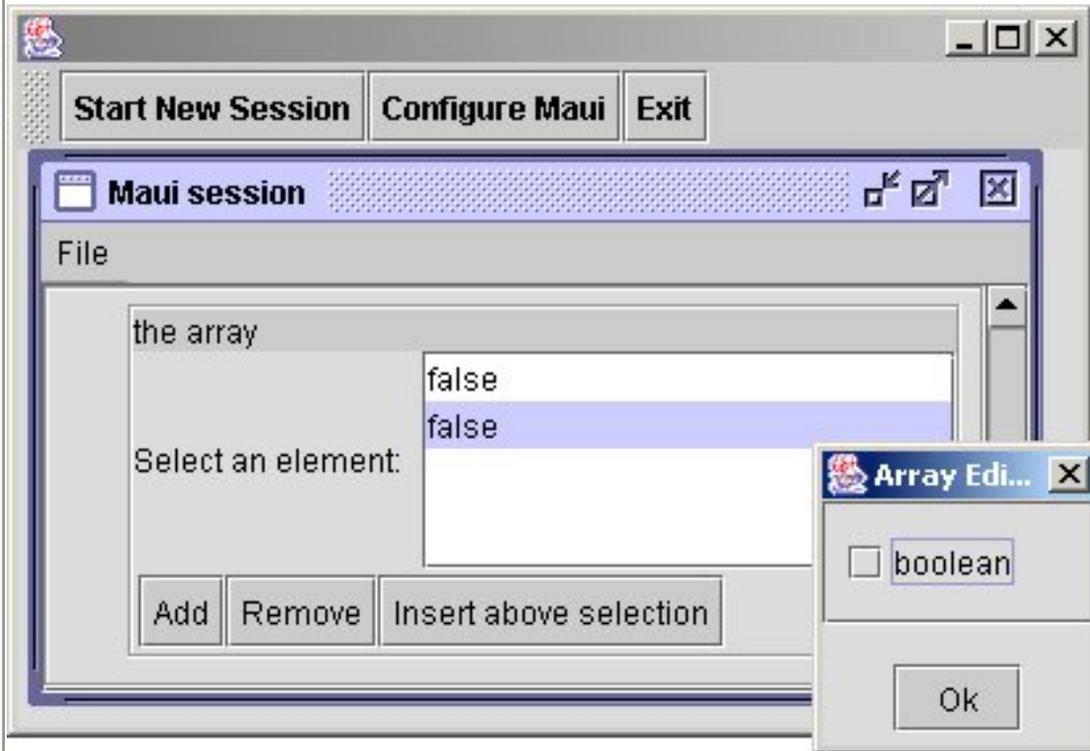
<Double> :Insert a double into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Double name="double" label="double"/>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```



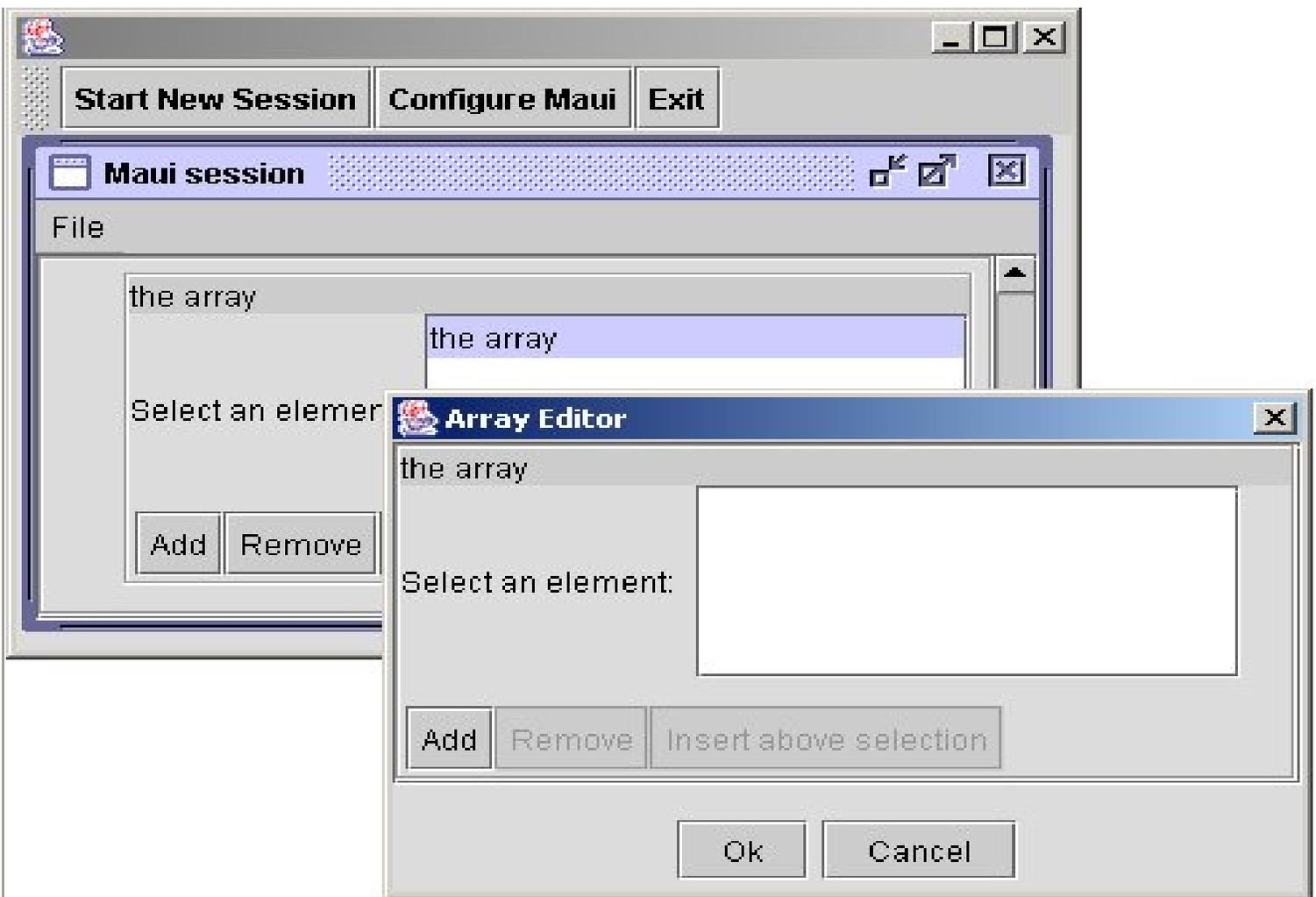
<Boolean> : Insert a boolean checkbox into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Boolean name="boolean" label="boolean"/>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```



<Array> Insert an array into the array element

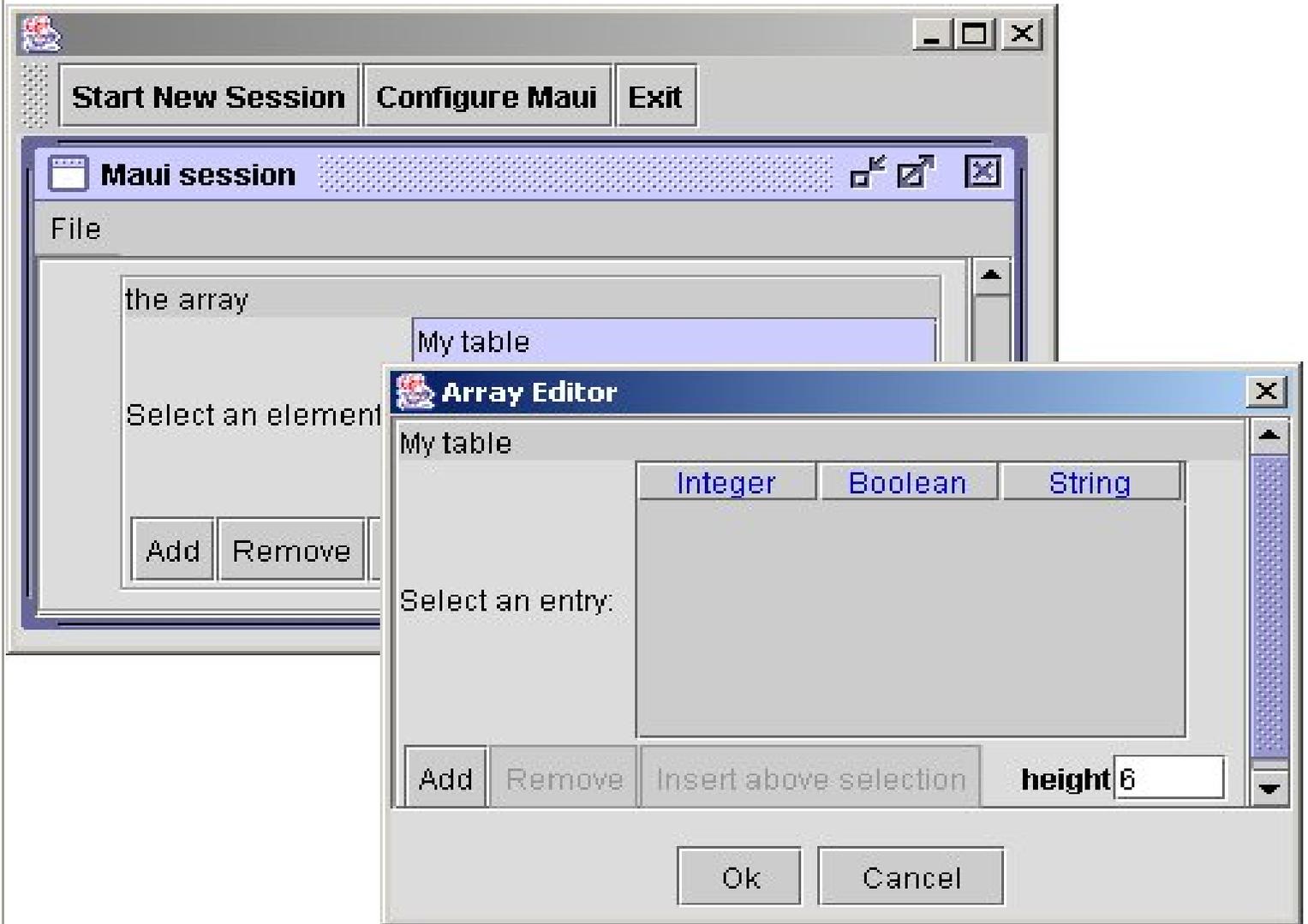
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Array name="myArray" label="the array">
            <Master>
              <Class type="master" label="label">
                <Fields>
                </Fields>
              </Class>
            </Master>
          </Array>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```



<Table> : Insert a table into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Table name="MyTable" label="My table">
            <Header name="columns" label="columns">
              <Integer name="Col1" label="Integer" />
              <Boolean name="Col2" label="Boolean" />
              <String name="Col3" label="String"/>
            </Header>
          </Table>
        </Master>
      </Array>
    </Fields>
  </Class>
```

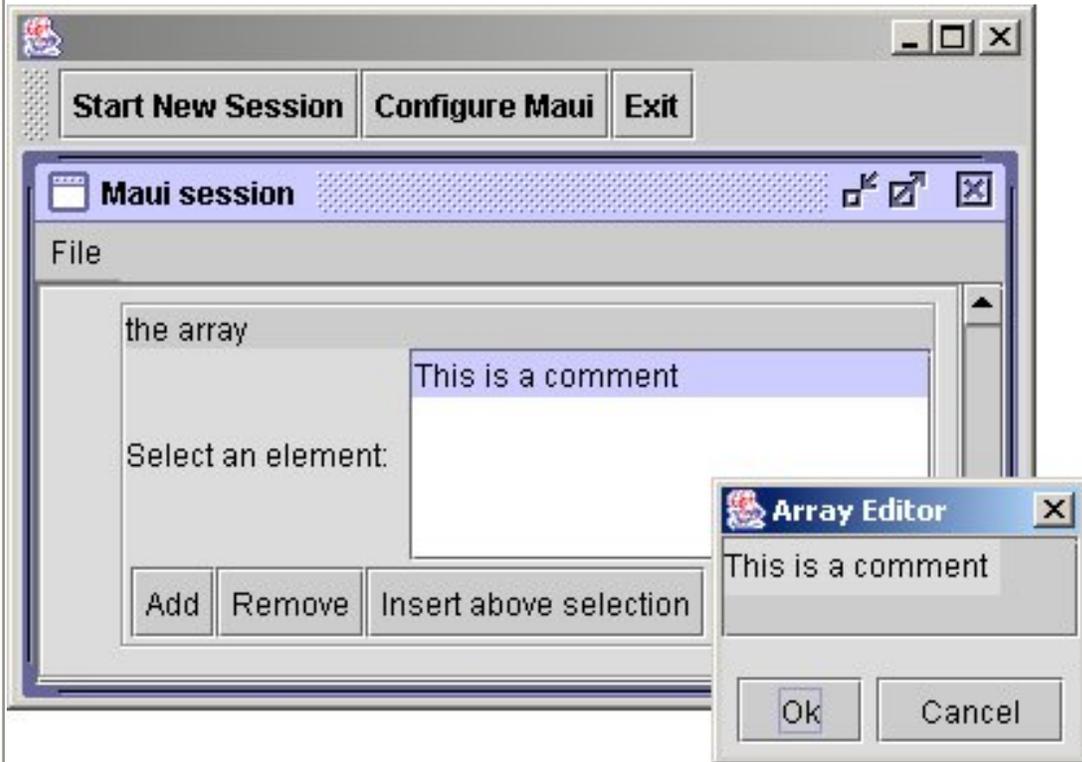
</Maui>



<Reference> : Broken in this release of Maui. A bug report has been submitted to the Maui developers.

<Comment> : Insert a comment into an array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Comment>This is a comment</Comment>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```

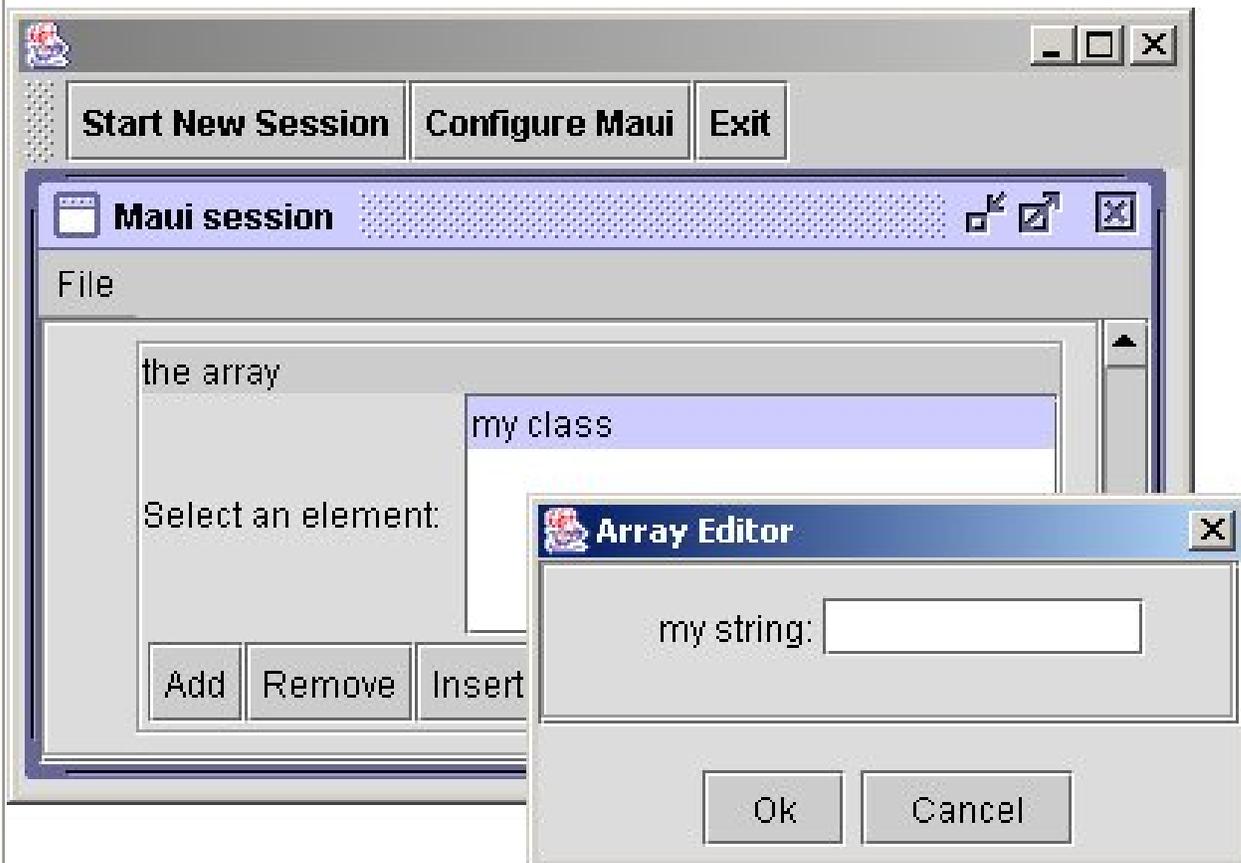


child class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <MyClass name="myClass" label="my class"/>
        </Master>
      </Array>
    </Fields>
  </Class>

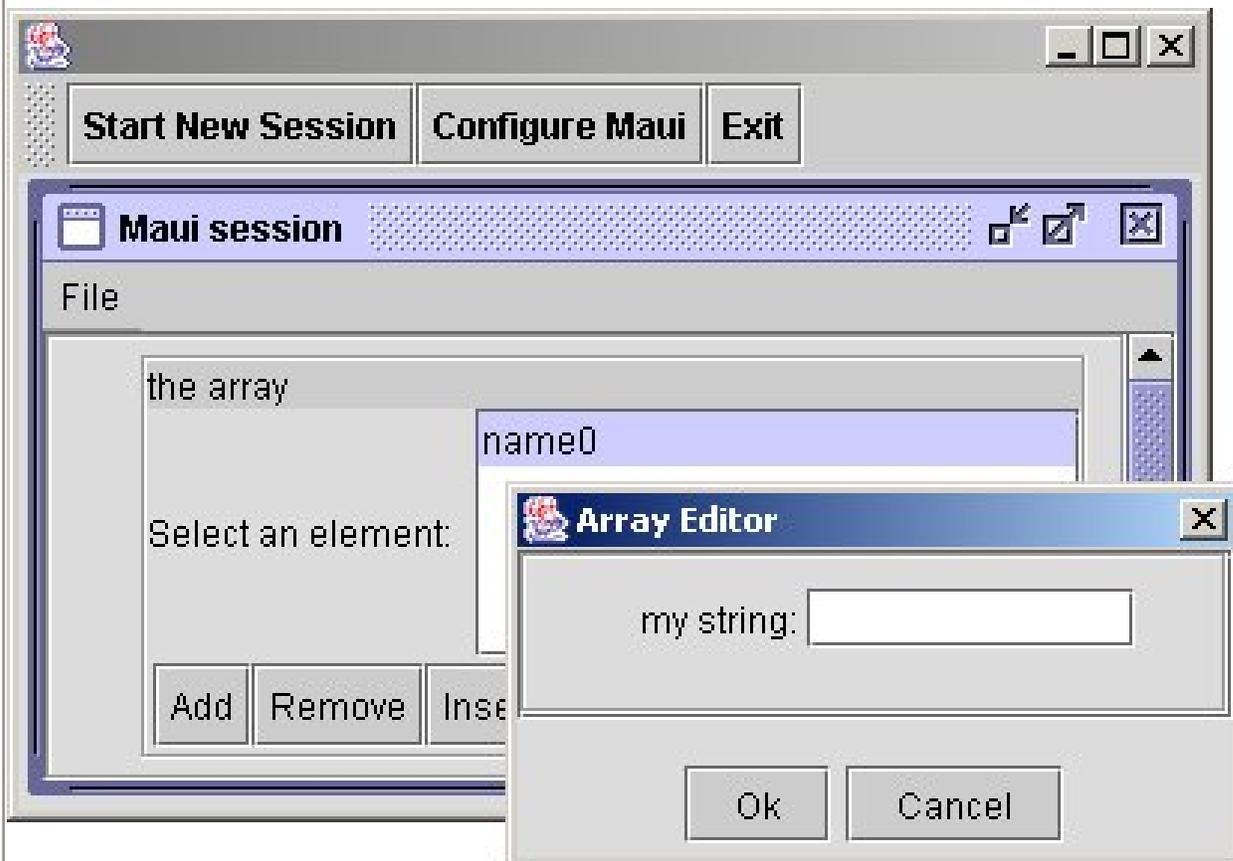
  <Class type="MyClass">
    <Fields>
      <String name="myString" label="my string"/>
    </Fields>
  </Class>

</Maui>
```



<Class> : Display a child class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Class type="MyClass">
            <Fields>
              <String name="myString" label="my string"/>
            </Fields>
          </Class>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.17.2 Attributes allowed in](#) **Up:** [B.17 Tag Master](#) **Previous:** [B.17 Tag Master](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.18 Tag Contents](#) **Up:** [B.17 Tag Master](#) **Previous:** [B.17.1 Children allowed in](#)

B.17.2 Attributes allowed in Master elements

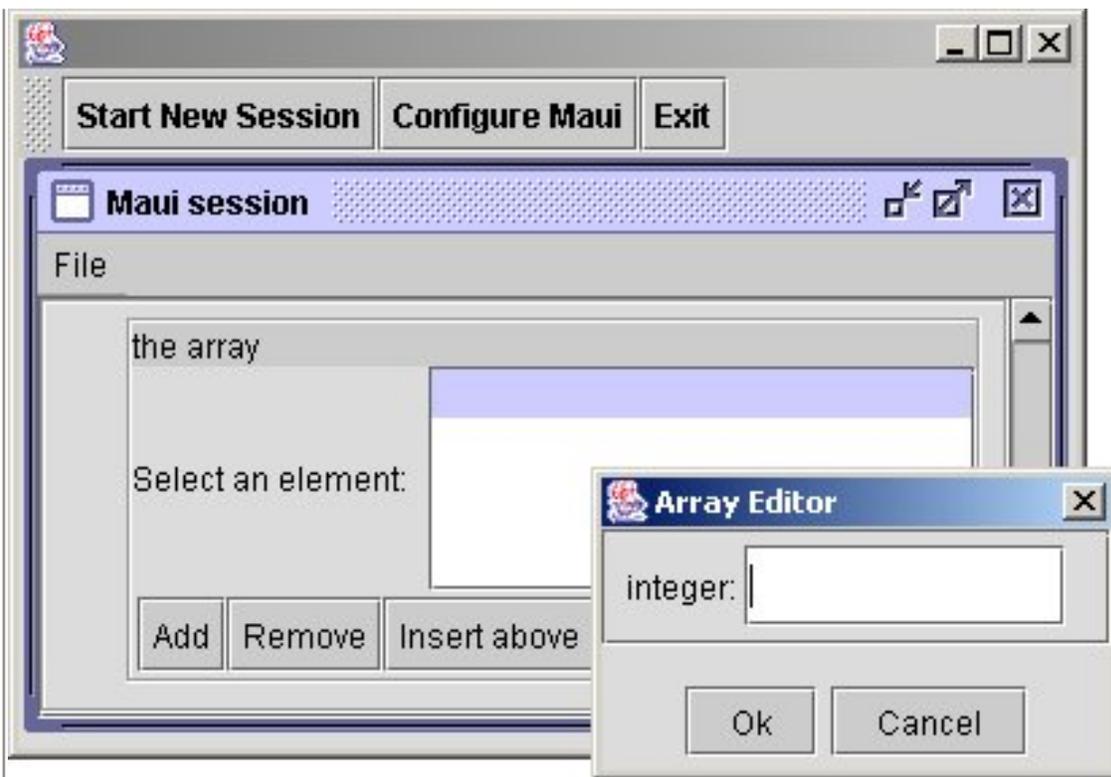
| Attribute name | Mandatory | Allowed values | Description and comments |
|----------------|-----------|----------------|--|
| label | no | any string | string to be used as a descriptive label for the Entry. This label can contain portions that are derived from an element's contents. |

`<Master>` : Contains the GUI components that are created whenever the end-user creates a new array element.

```

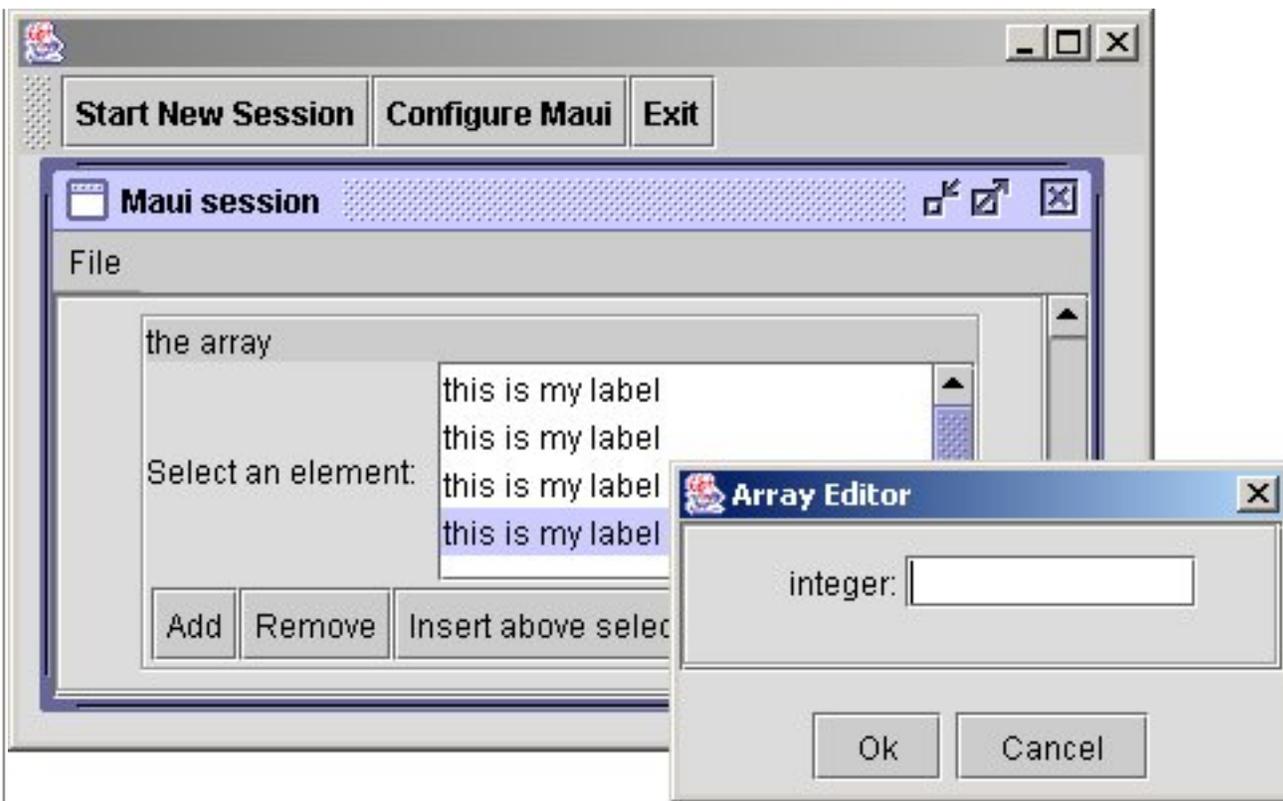
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master>
          <Class type="master" label="label">
            <Fields>
              <Integer name="integer" label="integer"/>
            </Fields>
          </Class>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>

```



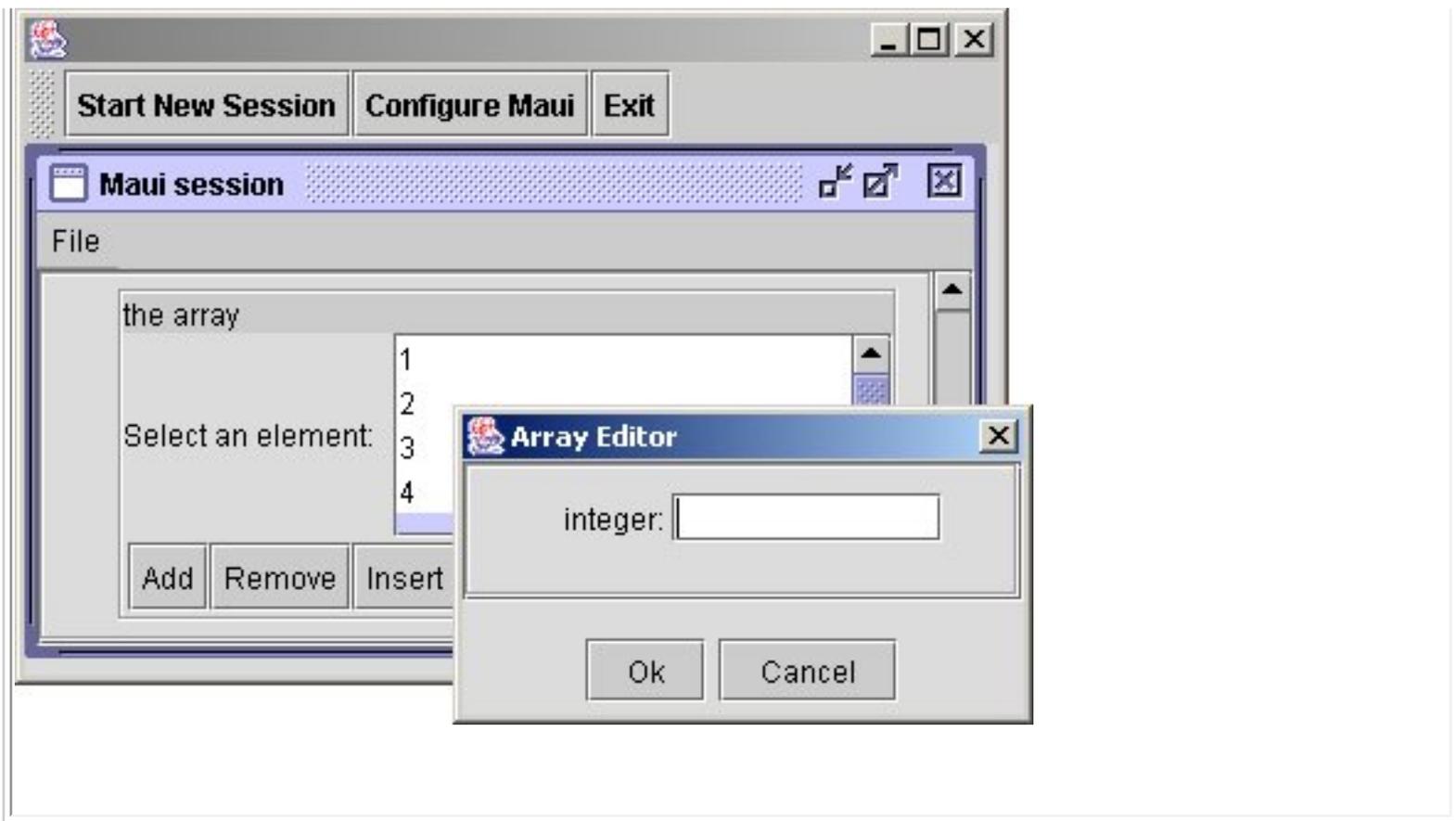
label : A list contains the label of every array element. In this example, every array element has the same label.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="this is my label">
          <Class type="master" label="label">
            <Fields>
              <Integer name="integer" label="integer"/>
            </Fields>
          </Class>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```



label : A list contains the label of every array element. In this example, the label contains the contents of the element's textbox.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$integer">
          <Class type="master" label="label">
            <Fields>
              <Integer name="integer" label="integer"/>
            </Fields>
          </Class>
        </Master>
      </Array>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.18 Tag Contents](#) **Up:** [B.17 Tag Master](#) **Previous:** [B.17.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.18.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.17.2 Attributes allowed in](#)

B.18 Tag Contents

The `Contents` element specifies the initial contents of an array.

Subsections

- [B.18.1 Children allowed in Contents elements](#)
 - [B.18.2 Attributes allowed in Contents elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.18.2 Attributes allowed in](#) **Up:** [B.18 Tag Contents](#) **Previous:** [B.18 Tag Contents](#)

B.18.1 Children allowed in Contents elements

| Tag | Number | Description and comments |
|------|------------|---|
| Item | any number | each Item is one of the initial elements of the array |

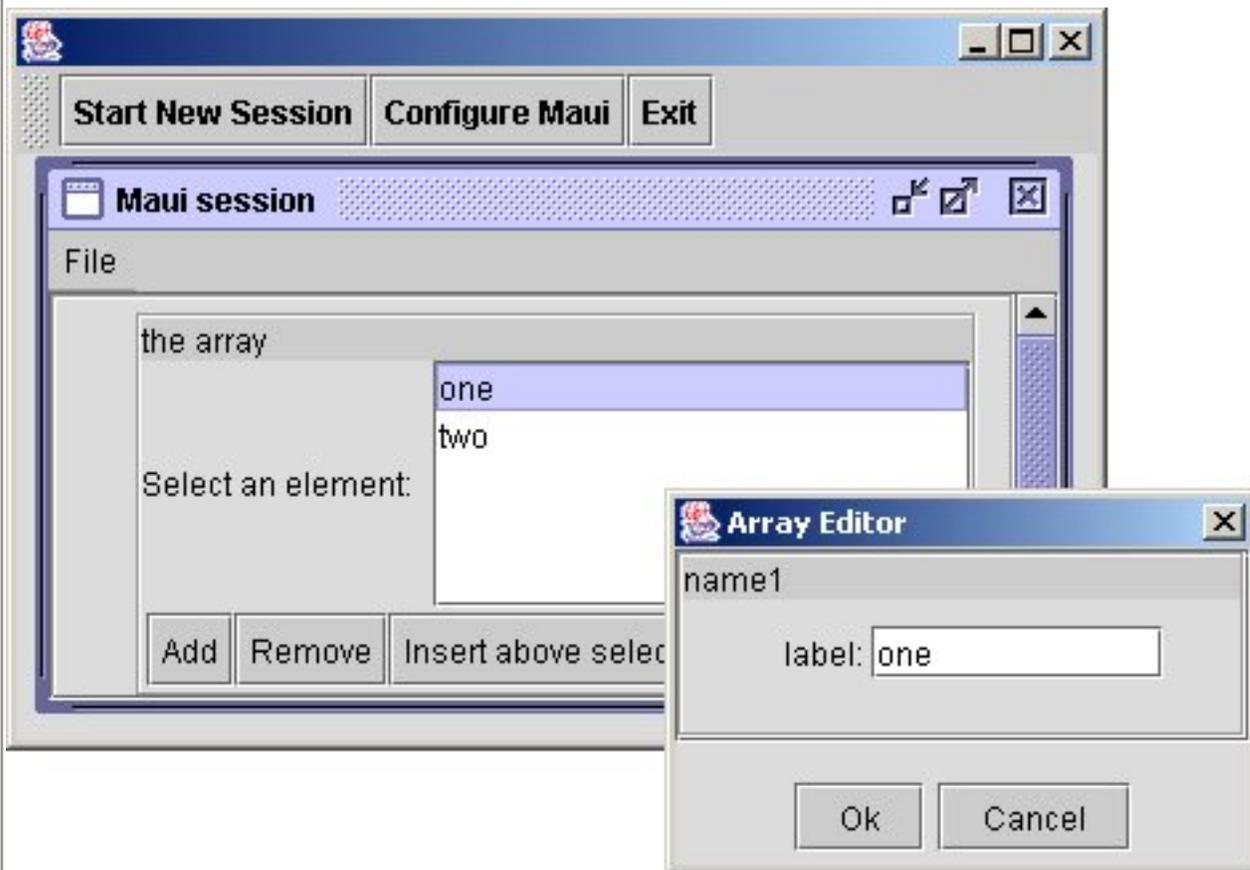
`<Contents>` : Contains one or more elements of an array.

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
        <Contents>
          <Item>
            <Class type="row1">
              <Fields>
                <String name="string"
                  label="label"
                  default="one"/>
              </Fields>
            </Class>
          </Item>
          <Item>
            <Class type="row2">
              <Fields>

```

```
<String name="string"  
      label="label "  
      default="two" />  
  </Fields>  
</Class>  
</Item>  
</Contents>  
</Array>  
</Fields>  
</Class>  
</Maui>
```



<Item> : the contents of one array element

```

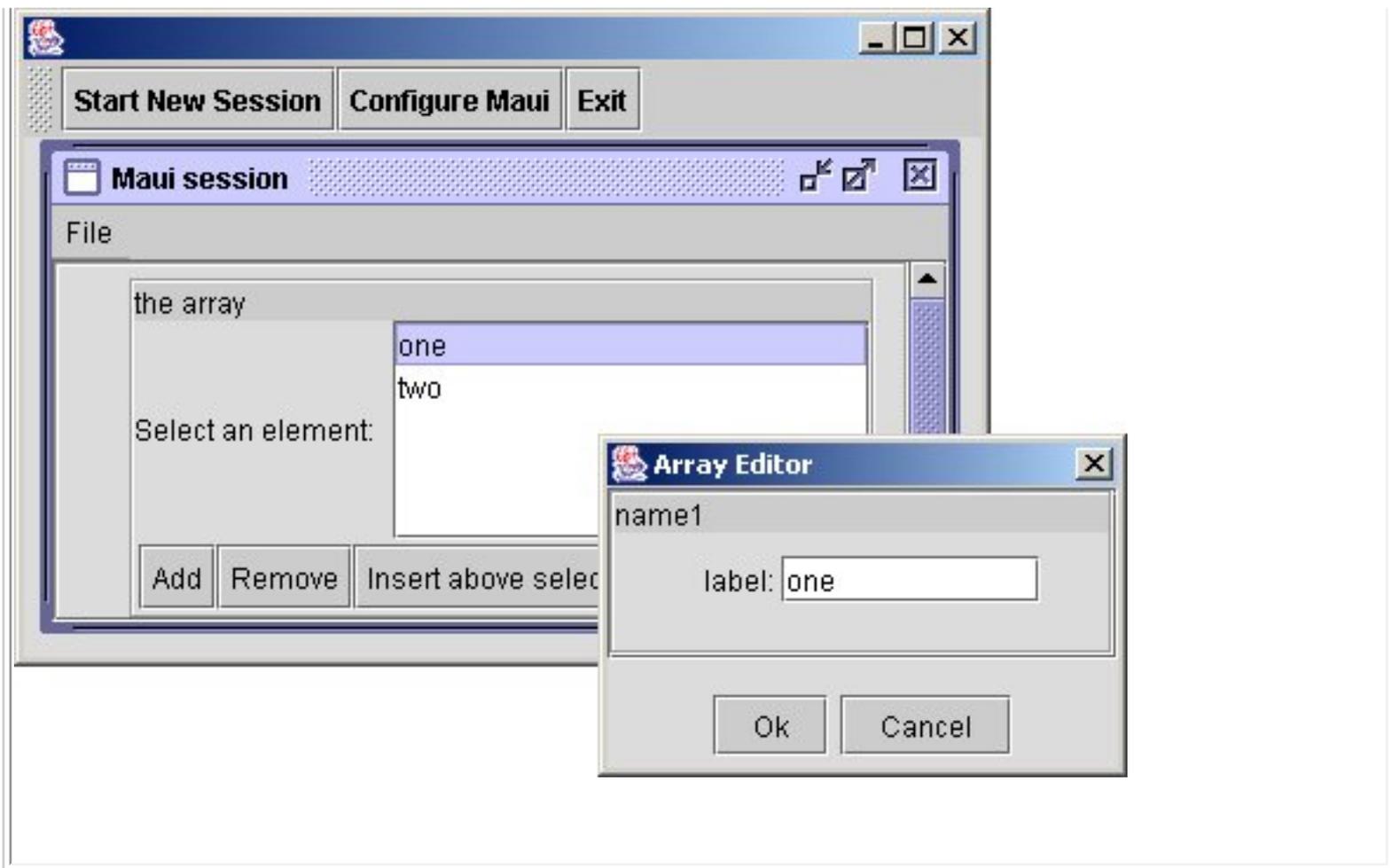
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label"/>
            </Fields>
          </Class>
        </Master>
      <Contents>

        <Item>
          <Class type="row1">
            <Fields>
              <String name="string"
                label="label"
                default="one"/>
            </Fields>
          </Class>
        </Item>

        <Item>
          <Class type="row2">
            <Fields>
              <String name="string"
                label="label"
                default="two"/>
            </Fields>
          </Class>
        </Item>

      </Contents>
    </Array>
  </Fields>
</Class>
</Maui>

```



[Next](#) [Up](#) [Previous](#)

Next: [B.18.2 Attributes allowed in](#) **Up:** [B.18 Tag Contents](#) **Previous:** [B.18 Tag Contents](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.19 Tag Item](#) **Up:** [B.18 Tag Contents](#) **Previous:** [B.18.1 Children allowed in](#)

B.18.2 Attributes allowed in Contents elements

[None](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.19 Tag Item](#) **Up:** [B.18 Tag Contents](#) **Previous:** [B.18.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.19.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.18.2 Attributes allowed in](#)

B.19 Tag Item

Item elements represent the initial values in an array. There must be one and only one child in Item for the initial value.

Subsections

- [B.19.1 Children allowed in Item elements](#)
 - [B.19.2 Attributes allowed in Item elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.19.2 Attributes allowed in](#) **Up:** [B.19 Tag Item](#) **Previous:** [B.19 Tag Item](#)

B.19.1 Children allowed in Item elements

| Tag | Description and comments | Examples |
|------------------|---|-------------------------|
| Integer | This feature is not implemented. | example |
| String | This feature is not implemented. | example |
| Double | This feature is not implemented. | example |
| Boolean | This feature is not implemented. | example |
| Array | This feature is not implemented. | example |
| Table | This feature is not implemented. | example |
| Reference | This feature is not implemented. | example |
| Comment | This feature is not implemented. | example |
| <i>type_name</i> | the initial element is a class data member of type <i>type_name</i> | example |
| Class | the initial element is a class | example |

`<Item>` : Contains the GUI components (buttons, textboxes, checkboxes) of one array element.

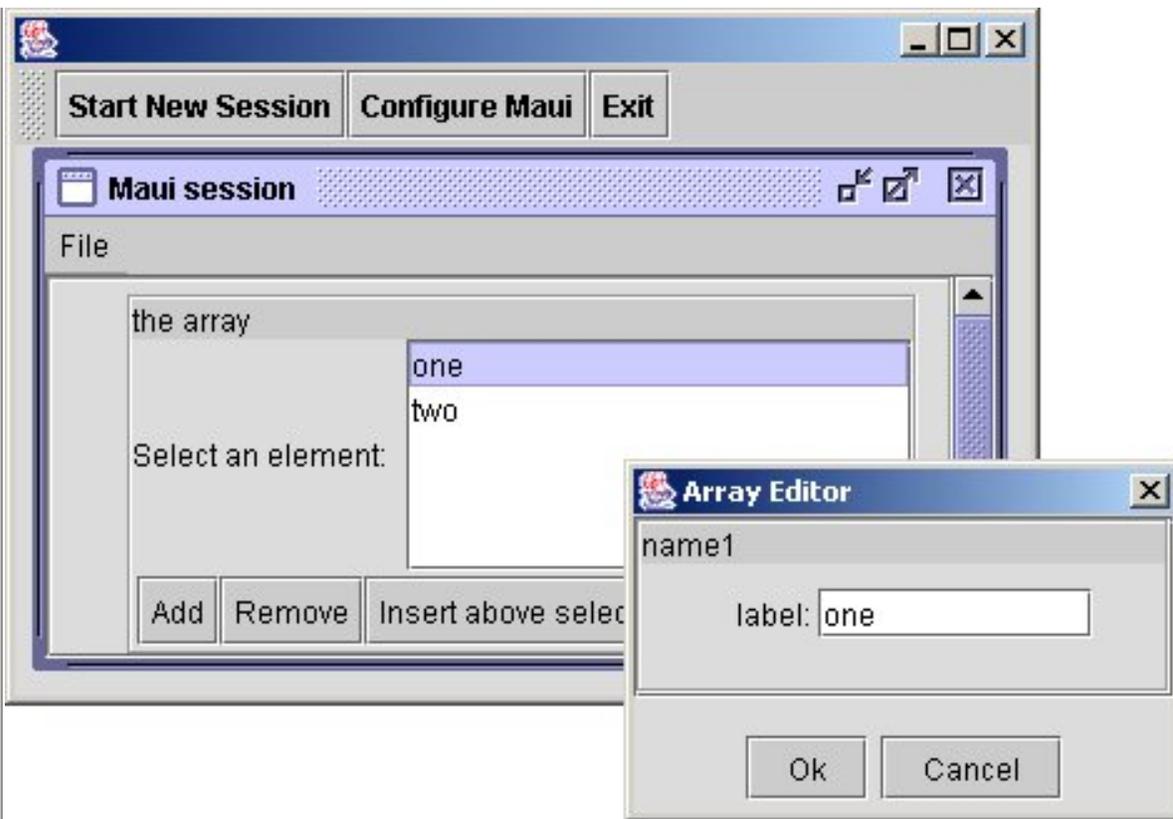
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Class type="master" label="label">
            <Fields>
              <String name="string"
                label="label" />
            </Fields>
          </Class>
        </Master>
      </Array>
    </Fields>
  </Class>
```

```
</Master>  
<Contents>
```

```
<Item>  
  <Class type="row1">  
    <Fields>  
      <String name="string"  
        label="label"  
        default="one" />  
    </Fields>  
  </Class>  
</Item>
```

```
<Item>  
  <Class type="row2">  
    <Fields>  
      <String name="string"  
        label="label"  
        default="two" />  
    </Fields>  
  </Class>  
</Item>
```

```
  </Contents>  
</Array>  
</Fields>  
</Class>  
</Maui>
```



<Integer> : This feature is not implemented.

<String> : This feature is not implemented.

<Double> : This feature is not implemented.

<Boolean> : This feature is not implemented.

<Array> : This feature is not implemented.

<Table> : This feature is not implemented.

<Reference> : This feature is not implemented.

<Comment> : This feature is not implemented.

child class

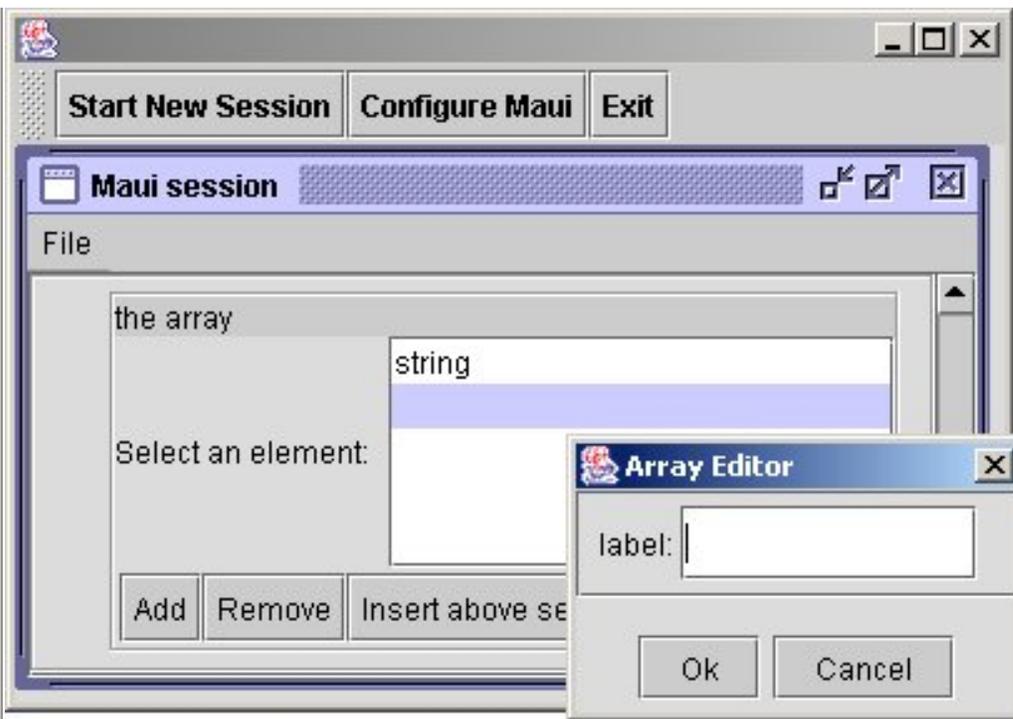
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Integer name="integer"
            label="label"/>
        </Master>

        <Contents>
          <Item>
            <MyClass name="myClass" label="my class"/>
          </Item>
        </Contents>

      </Array>
    </Fields>
  </Class>

  <Class type="MyClass">
    <Fields>
      <String name="myString" label="my string"/>
    </Fields>
  </Class>

</Maui>
```



<Class> : Display a child class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer" label="the container">
    <Fields>
      <Array name="myArray" label="the array">
        <Master label="$string">
          <Integer name="integer"
            label="label"/>
        </Master>

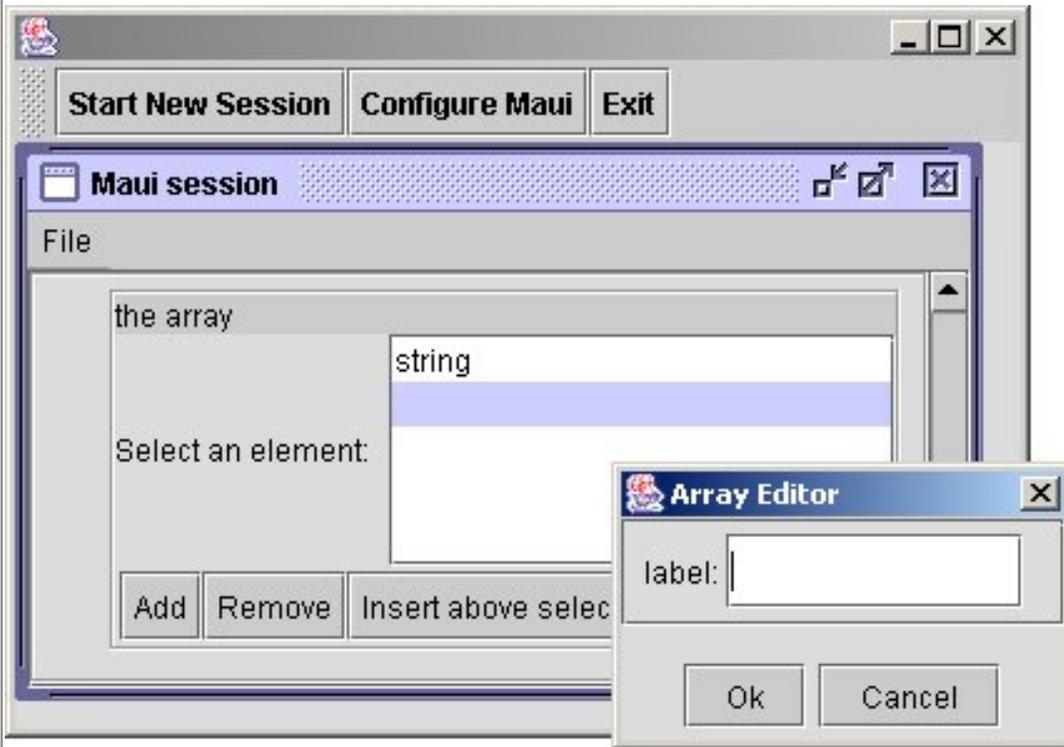
        <Contents>
          <Item>

            <Class type="MyClass">
              <Fields>
                <String name="myString" label="my string"/>
              </Fields>
            </Class>

          </Item>
        </Contents>

      </Array>
    </Fields>
  </Class>
```

</Maui>



[Next](#) [Up](#) [Previous](#)

Next: [B.19.2 Attributes allowed in](#) **Up:** [B.19 Tag Item](#) **Previous:** [B.19 Tag Item](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.20 Tag Header](#) **Up:** [B.19 Tag Item](#) **Previous:** [B.19.1 Children allowed in](#)

B.19.2 Attributes allowed in Item elements

None

[Next](#) [Up](#) [Previous](#)

Next: [B.20 Tag Header](#) **Up:** [B.19 Tag Item](#) **Previous:** [B.19.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.20.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.19.2 Attributes allowed in](#)

B.20 Tag Header

Header provides the template used for all entries added to the table. Each child in the Header represents one column of the table.

Subsections

- [B.20.1 Children allowed in Header elements](#)
 - [B.20.2 Attributes allowed in Header elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.20.2 Attributes allowed in](#) **Up:** [B.20 Tag Header](#) **Previous:** [B.20 Tag Header](#)

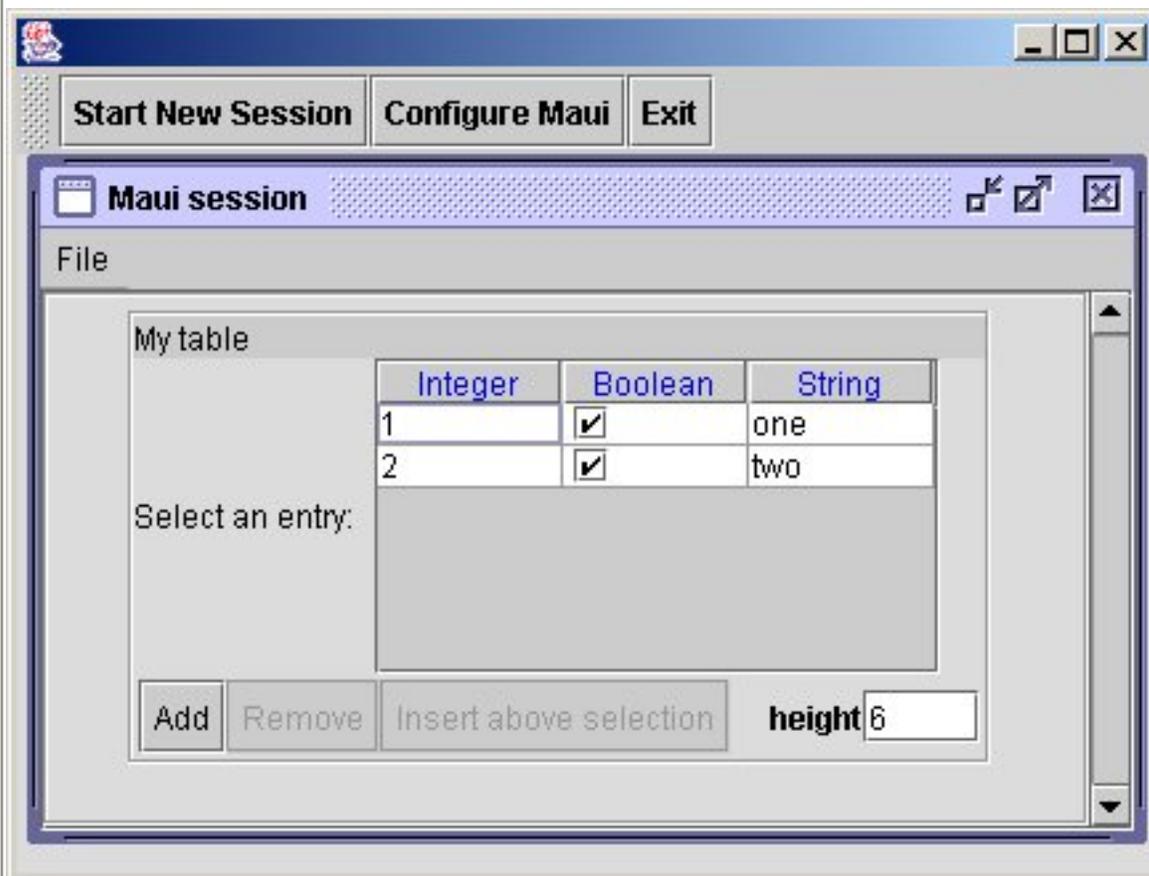
B.20.1 Children allowed in Header elements

| Tag | Description and comments | Examples |
|-----------|--|-------------------------|
| Integer | the table will contain a column of integer variables | example |
| String | the table will contain a column of string variables | example |
| Double | the table will contain a column of double variables | example |
| Boolean | the table will contain a column of boolean (logical) variables | example |
| Reference | the table will contain a column of reference variables | example |

<Header> : Contains the GUI components (buttons, textboxes, checkboxes, etc) that will be created when the end-user creates a new row in the table.

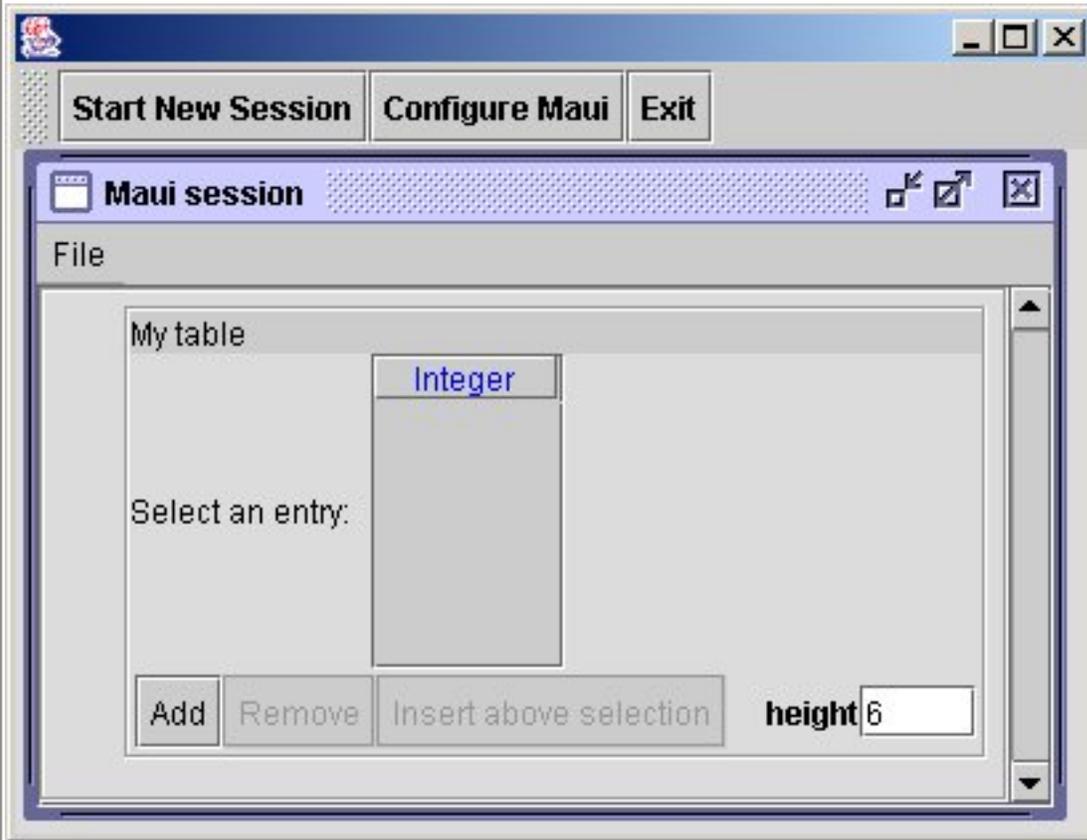
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="one" />
    </Entry>
    <Entry name="entry2">
        <Cell field="Col1" value="2" />
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="two" />
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>
```



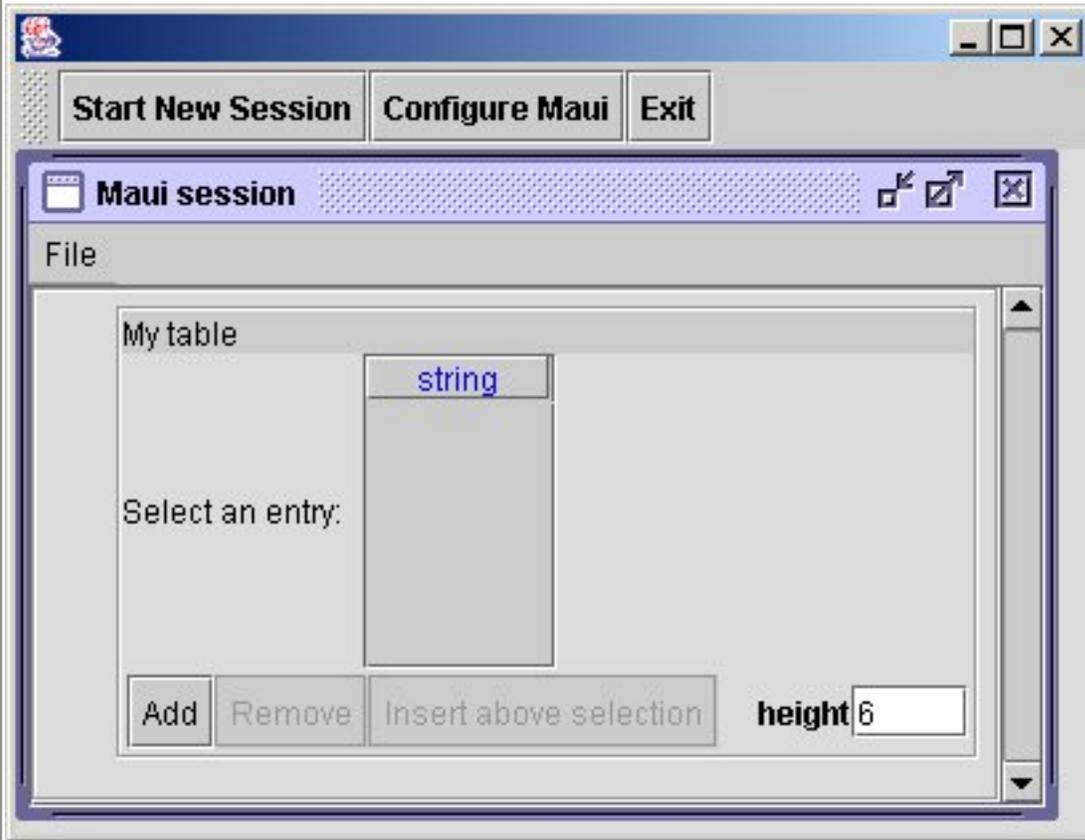
<Integer> : Insert an integer into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
        </Header>
      </Table>
    </Fields>
  </Class>
</Maui>
```



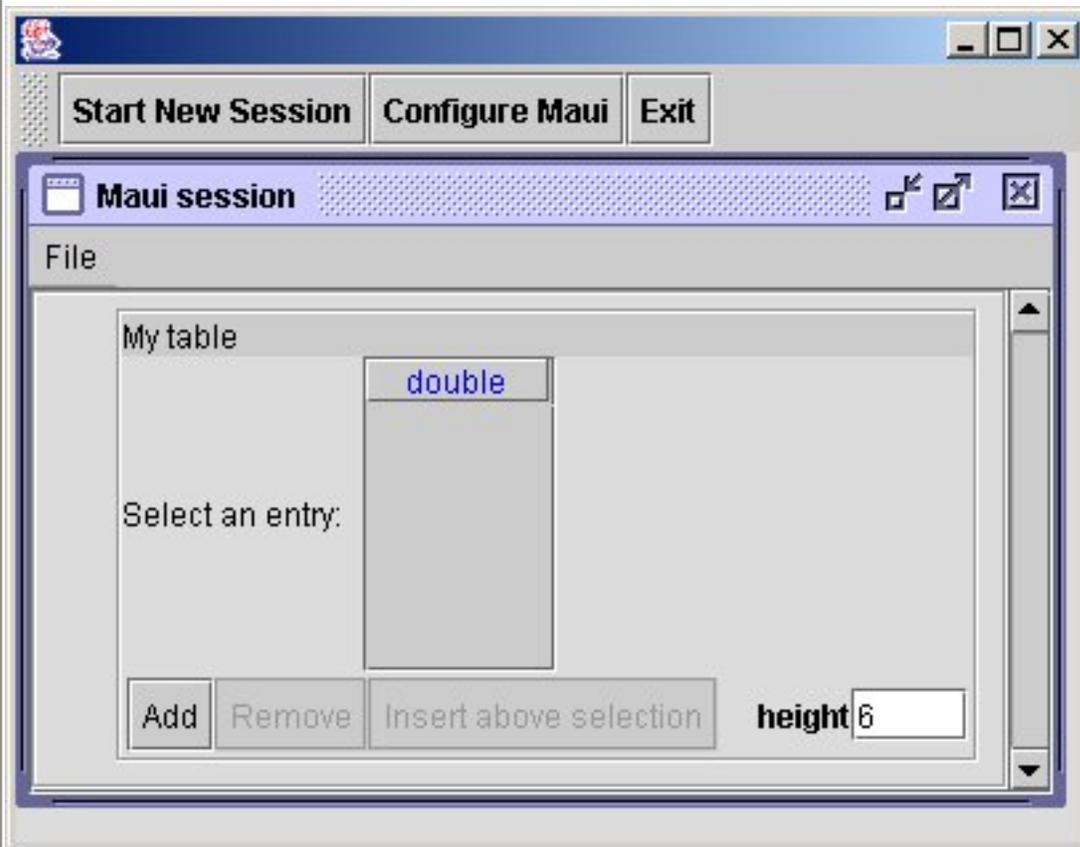
<String> : Insert a string into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <String name="Col1" label="string" />
        </Header>
      </Table>
    </Fields>
  </Class>
</Maui>
```



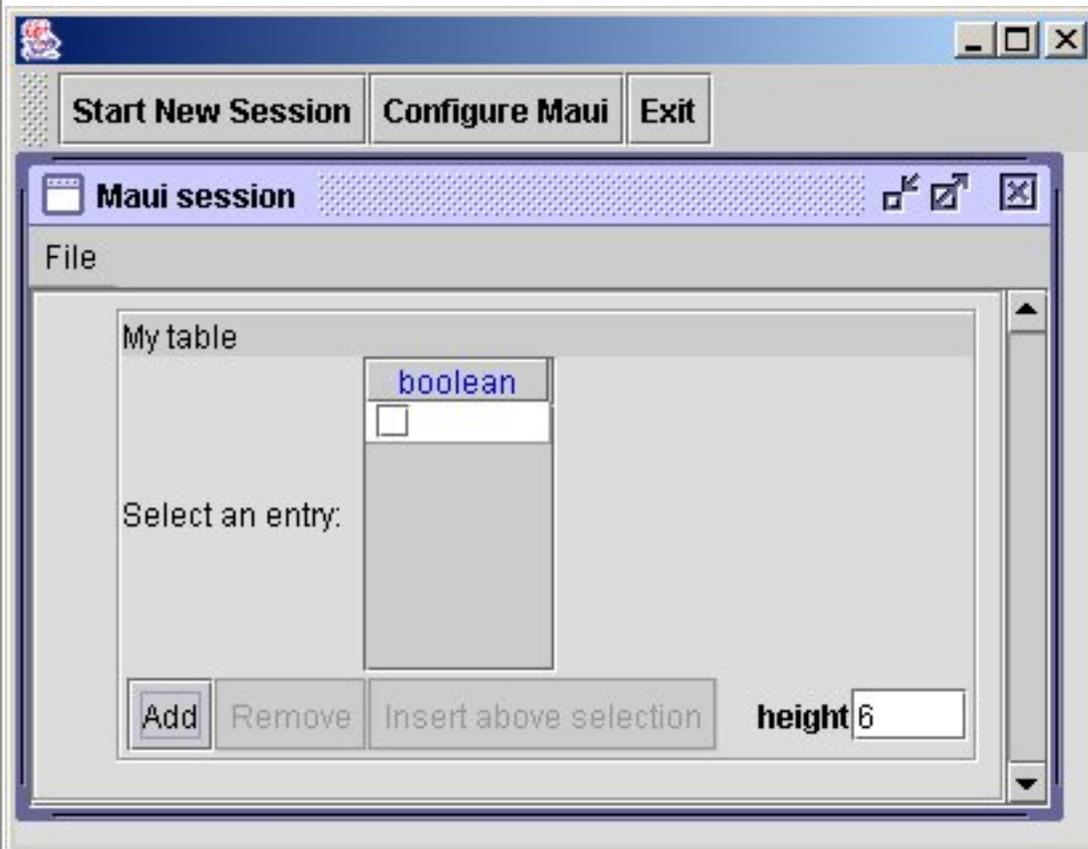
<Double> :Insert a double into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Double name="Col1" label="double" />
        </Header>
      </Table>
    </Fields>
  </Class>
</Maui>
```



<Boolean> : Insert a boolean checkbox into the array element

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Boolean name="Col1" label="boolean" />
        </Header>
      </Table>
    </Fields>
  </Class>
</Maui>
```



<Reference> : Broken in this release of Maui. A bug report has been submitted to the Maui developers.

[Next](#) [Up](#) [Previous](#)

Next: [B.20.2 Attributes allowed in](#) **Up:** [B.20 Tag Header](#) **Previous:** [B.20 Tag Header](#)

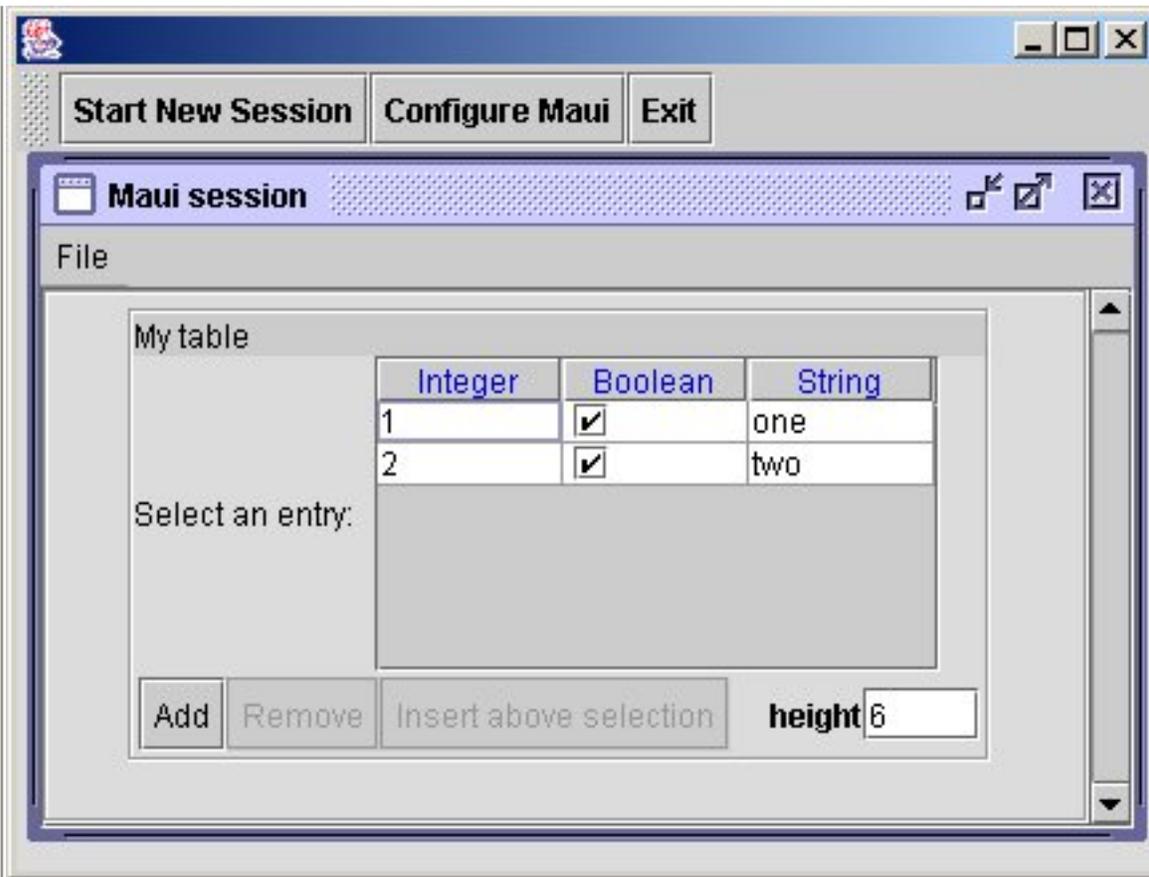
[Next](#)
[Up](#)
[Previous](#)
Next: [B.21 Tag Entries](#)
Up: [B.20 Tag Header](#)
Previous: [B.20.1 Children allowed in](#)

B.20.2 Attributes allowed in Header elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|--------------------|---|-------------------------|
| name | yes | any legal name | default name for new entries | example |
| label | no | any string | not implemented | example |
| sizing | no | a list of integers | specify the Width of the columns in the header. This is a list of integers separated by pipes where each integer gives the size of the corresponding column. If the sizing space for a given column is empty or negative, the default Maui sizing will be used. | example |
| toolTip | no | any string | not implemented | example |
| helpMessage | no | any string | not implemented | example |
| visible | no | true or false | not implemented | example |

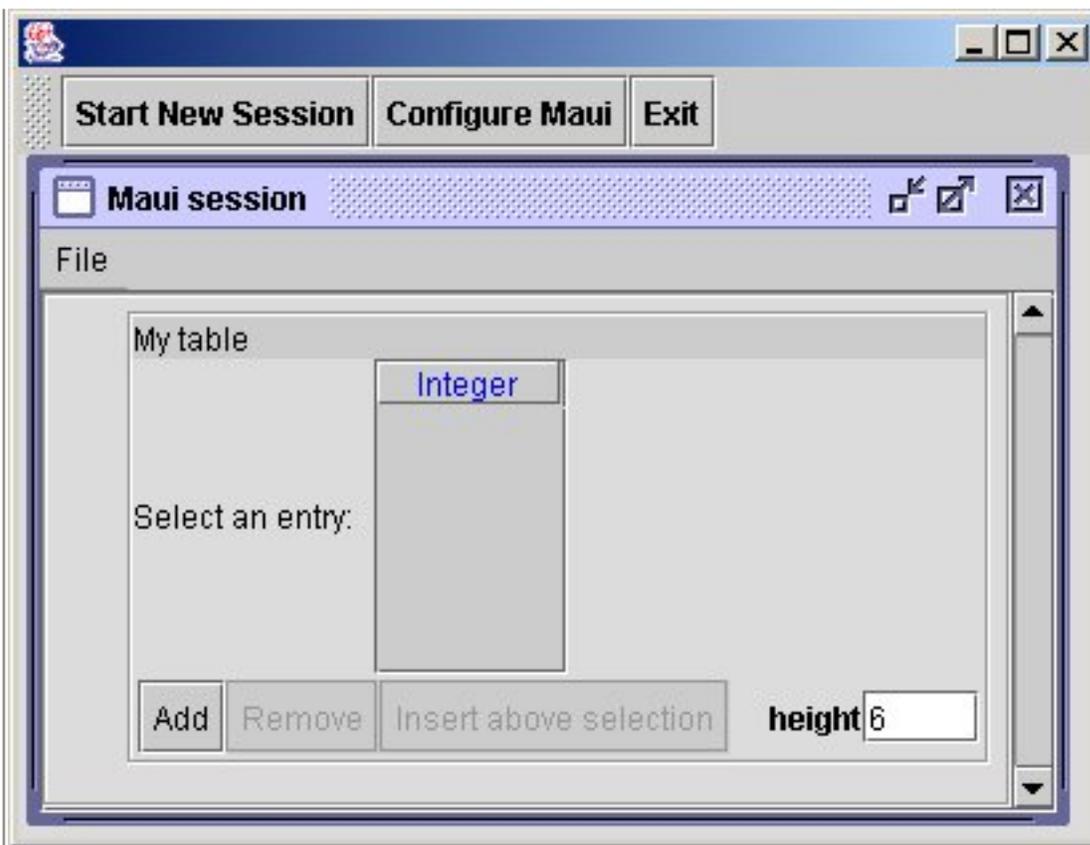
<Header> : Contains the GUI components (buttons, textboxes, checkboxes, etc) that will be created when the end-user creates a new row in the table.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String" />
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```



name: The name of the header

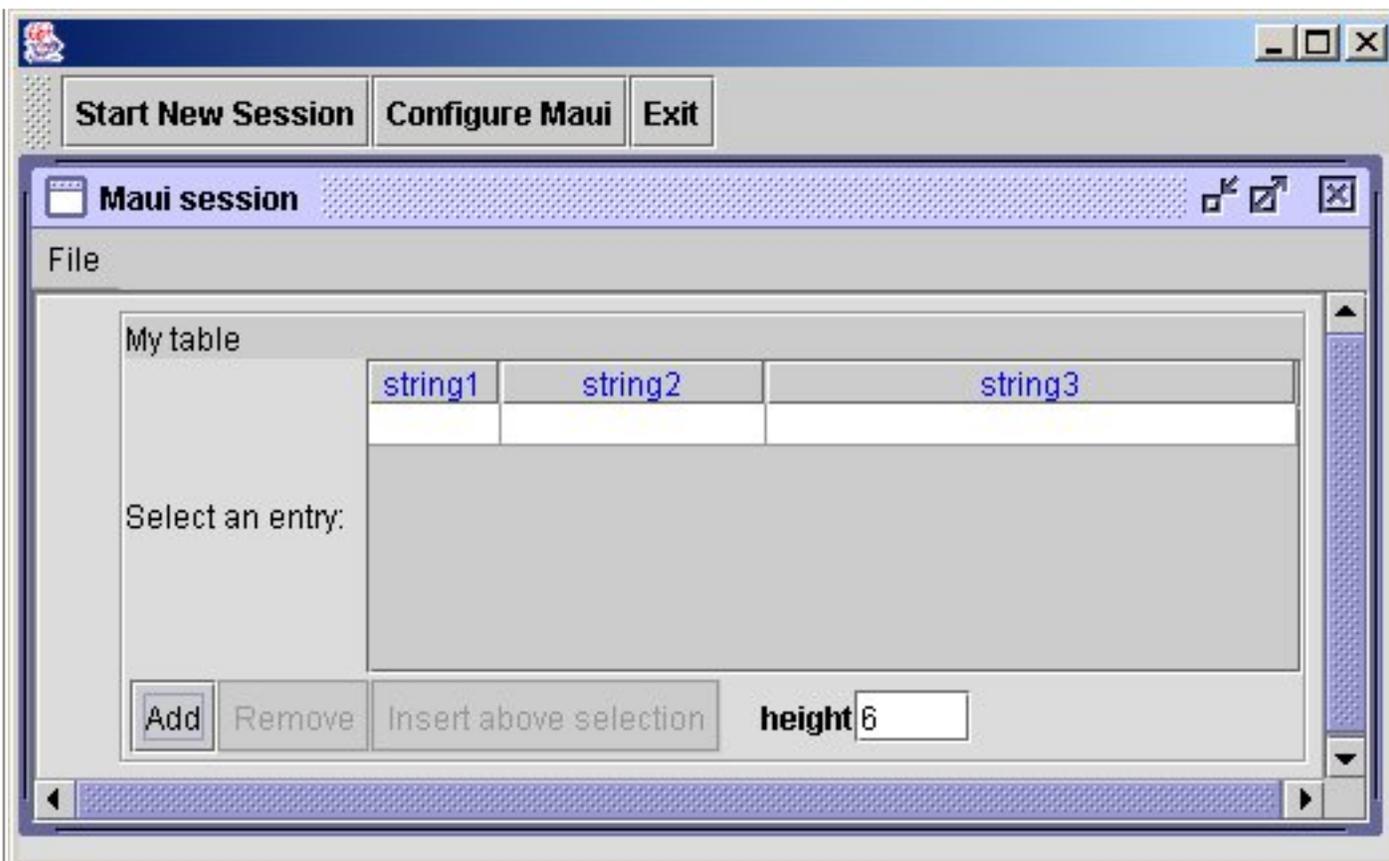
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
        </Header>
      </Table>
    </Fields>
  </Class>
</Maui>
```



label: (Author's Note: I will ask the developer if this attribute does anything useful)

sizing: The size of each column in pixels

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" sizing="50|100|200">
          <String name="string1" label="string1"/>
          <String name="string2" label="string2"/>
          <String name="string3" label="string3"/>
        </Header>
      </Table>
    </Fields>
  </Class>
</Maui>
```



toolTip: (Author's Note: I will ask the developer if this attribute does anything useful)

helpMessage: (Author's Note: I will ask the developer if this attribute does anything useful)

visible: (Author's Note: I will ask the developer if this attribute does anything useful)

[Next](#) [Up](#) [Previous](#)

Next: [B.21 Tag Entries](#) **Up:** [B.20 Tag Header](#) **Previous:** [B.20.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.21.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.20.2 Attributes allowed in](#)

B.21 Tag Entries

The `Entries` element specifies the initial contents of a table.

Subsections

- [B.21.1 Children allowed in Entries elements](#)
 - [B.21.2 Attributes allowed in Entries elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.21.2 Attributes allowed in Up](#) **Up:** [B.21 Tag Entries](#) **Previous:** [B.21 Tag Entries](#)

B.21.1 Children allowed in Entries elements

| Tag | Number | Description and comments | Examples |
|-------|------------|---|-------------------------|
| Entry | any number | each Entry is one of the initial entries of the table | example |

<Entries> : The GUI components (buttons, textboxes, checkboxes, etc) that are contained inside of a table.

```

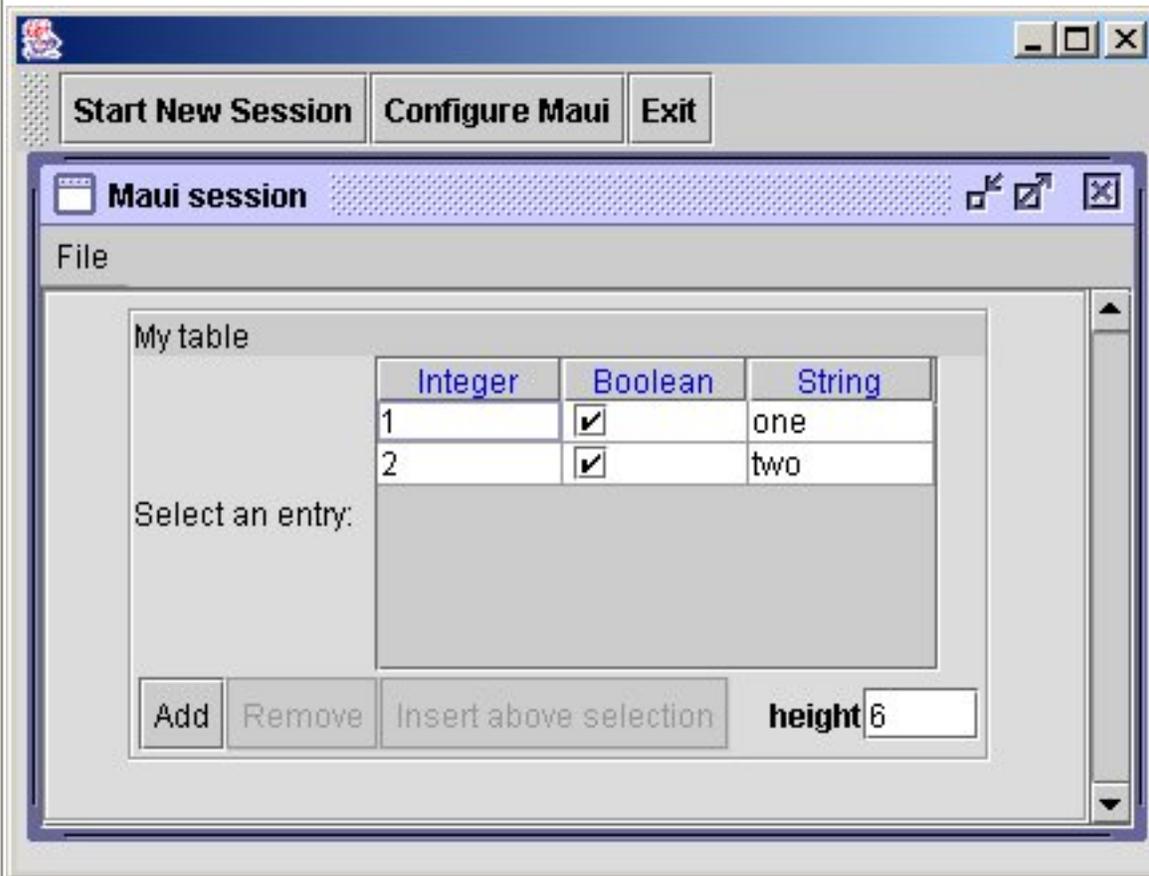
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>
          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>

```

```

    </Fields>
  </Class>
</Maui>

```



<Entry> : The contents of a row inside of a table.

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>

```

```

          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />

```

```
<Cell field="Col3" value="one" />
</Entry>
```

```
<Entry name="entry2">
  <Cell field="Col1" value="2" />
  <Cell field="Col2" value="true" />
  <Cell field="Col3" value="two" />
</Entry>
```

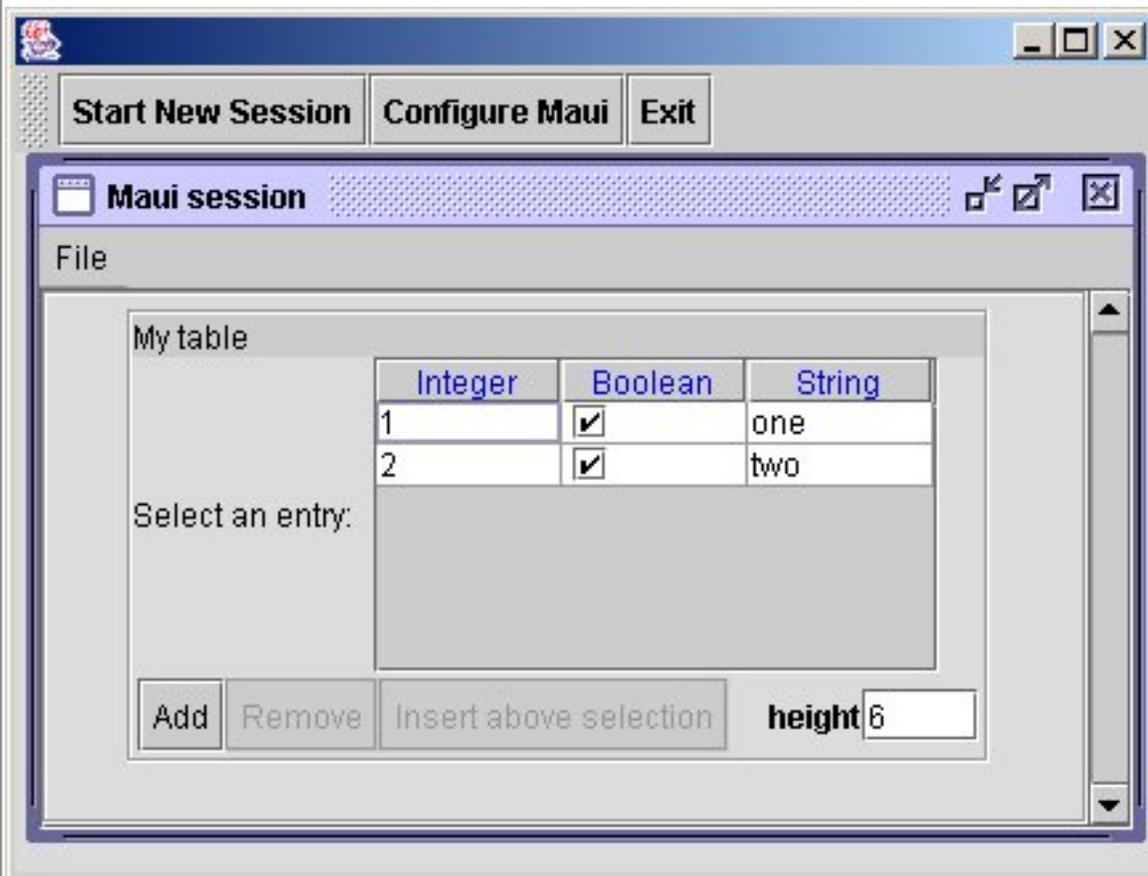
```
</Entries>
```

```
</Table>
```

```
</Fields>
```

```
</Class>
```

```
</Maui>
```



Next: [B.21.2 Attributes allowed in](#) **Up:** [B.21 Tag Entries](#) **Previous:** [B.21 Tag Entries](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.22 Tag Entry](#) **Up:** [B.21 Tag Entries](#) **Previous:** [B.21.1 Children allowed in](#)

B.21.2 Attributes allowed in Entries elements

[None](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.22 Tag Entry](#) **Up:** [B.21 Tag Entries](#) **Previous:** [B.21.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.22.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.21.2 Attributes allowed in](#)

B.22 Tag Entry

Entry elements represent the initial values in a table.

Subsections

- [B.22.1 Children allowed in Entry elements](#)
 - [B.22.2 Attributes allowed in Entry elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.22.2 Attributes allowed in](#) **Up:** [B.22 Tag Entry](#) **Previous:** [B.22 Tag Entry](#)

B.22.1 Children allowed in Entry elements

| Tag | Number | Description and comments | Examples |
|------|------------|---|-------------------------|
| Cell | any number | the initial contents of one individual field in a table entry | example |

<Entry> : The GUI components (buttons, textboxes, checkboxes, etc) that are in one row of a table.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
```

```
  <Entry name="entry1">
    <Cell field="Col1" value="1" />
    <Cell field="Col2" value="true" />
    <Cell field="Col3" value="one" />
  </Entry>
```

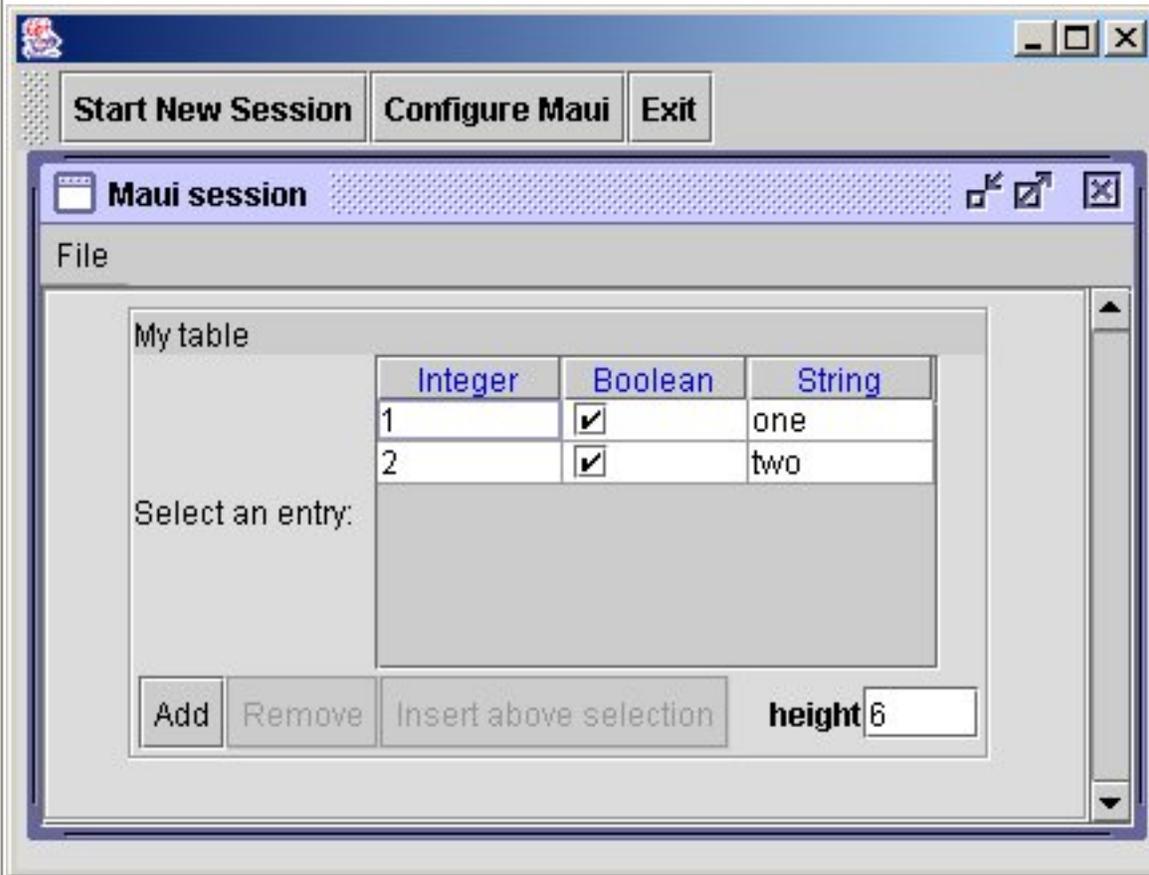
```
  <Entry name="entry2">
    <Cell field="Col1" value="2" />
    <Cell field="Col2" value="true" />
    <Cell field="Col3" value="two" />
  </Entry>
```

```
</Entries>
```

```

    </Table>
  </Fields>
</Class>
</Maui>

```



<Cell> The contents of one cell in the table.

```

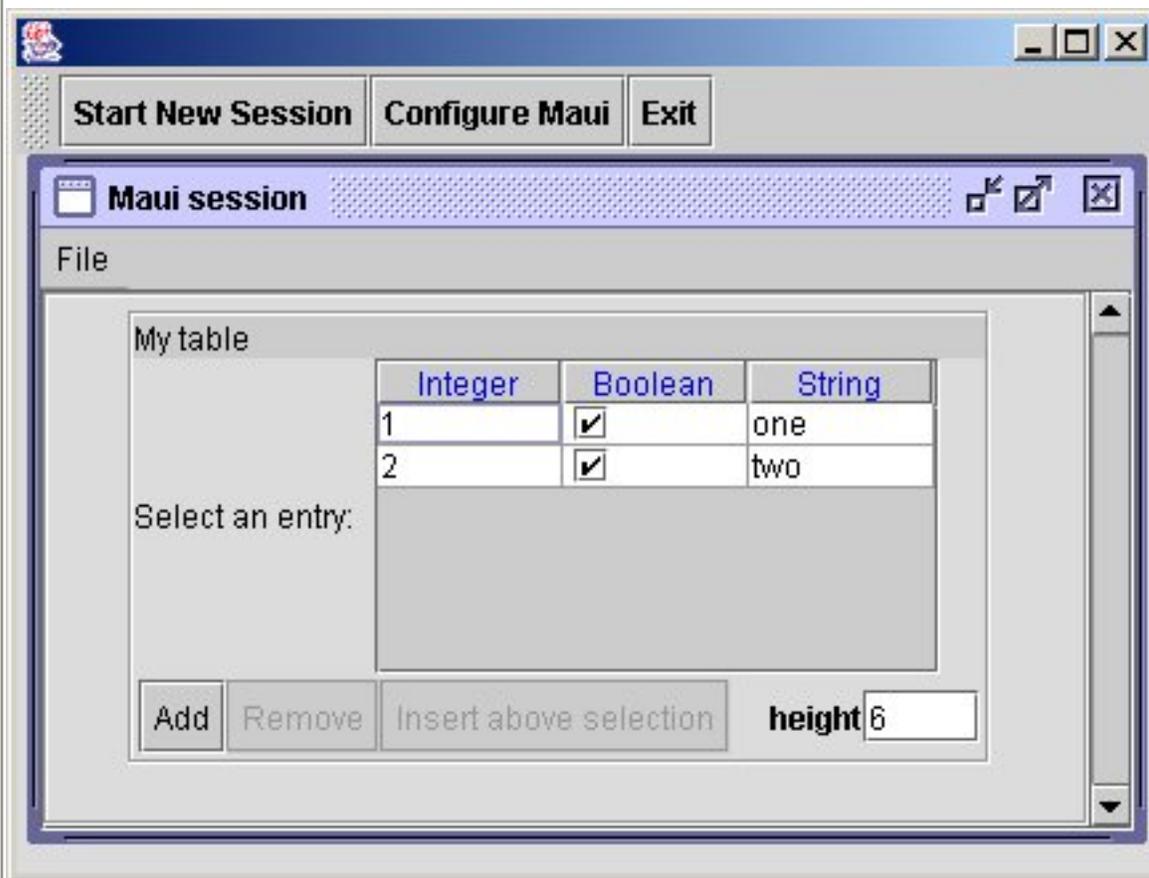
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
          <Entry name="entry1">
            <Cell field="Col1" value="1" />

```

```

        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="one" />
    </Entry>
    <Entry name="entry2">
        <Cell field="Col1" value="2" />
        <Cell field="Col2" value="true" />
        <Cell field="Col3" value="two" />
    </Entry>
</Entries>
</Table>
</Fields>
</Class>
</Maui>

```



Next: [B.22.2 Attributes allowed in](#) **Up:** [B.22 Tag Entry](#) **Previous:** [B.22 Tag Entry](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.23 Tag Cell](#) **Up:** [B.22 Tag Entry](#) **Previous:** [B.22.1 Children allowed in](#)

B.22.2 Attributes allowed in Entry elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|----------------|--------------------------|-------------------------|
| name | no | any legal name | the name for this entry | example |
| label | no | any string | not implemented | example |

`<Entry>` : The GUI components (buttons, textboxes, checkboxes, etc) that are in one row of a table.

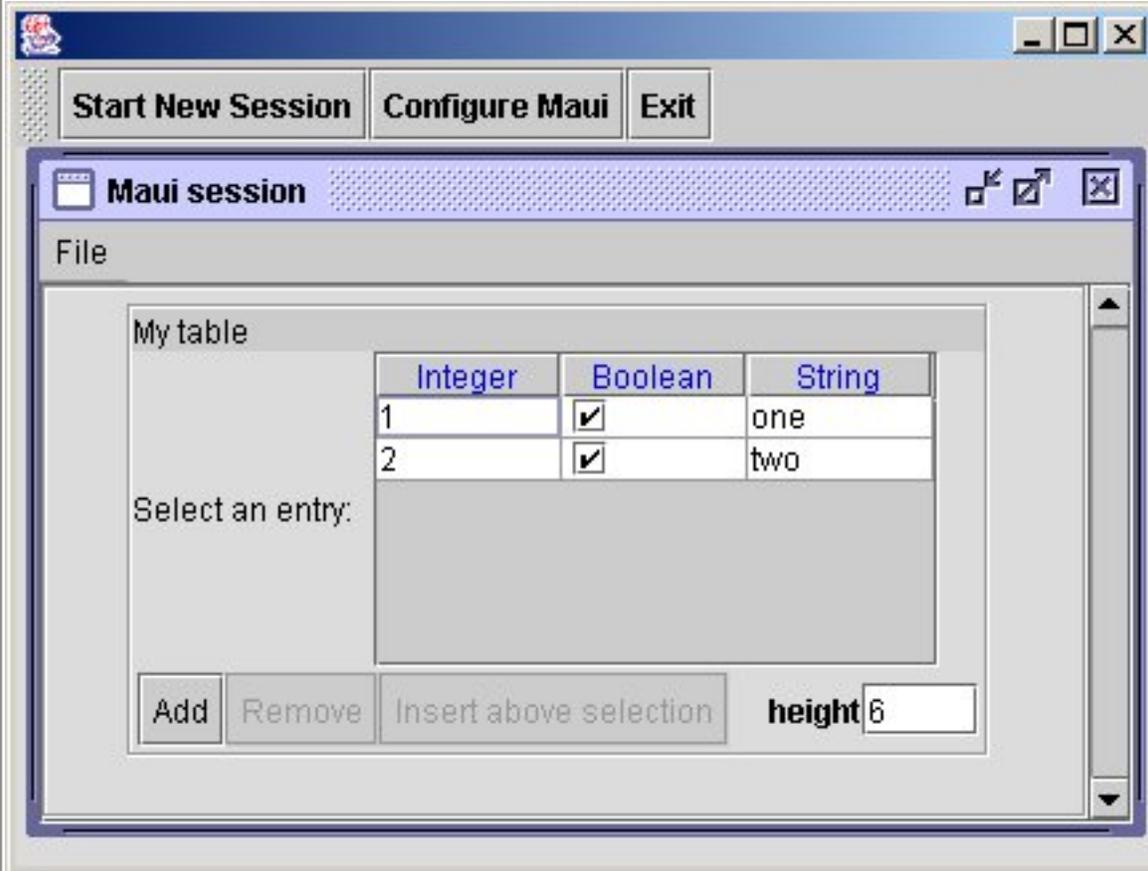
```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>
```

```
<Entry name="entry1">
  <Cell field="Col1" value="1" />
  <Cell field="Col2" value="true" />
  <Cell field="Col3" value="one" />
</Entry>
```

```
<Entry name="entry2">
  <Cell field="Col1" value="2" />
```

```
<Cell field="Col2" value="true" />  
<Cell field="Col3" value="two" />  
</Entry>
```

```
</Entries>  
</Table>  
</Fields>  
</Class>  
</Maui>
```



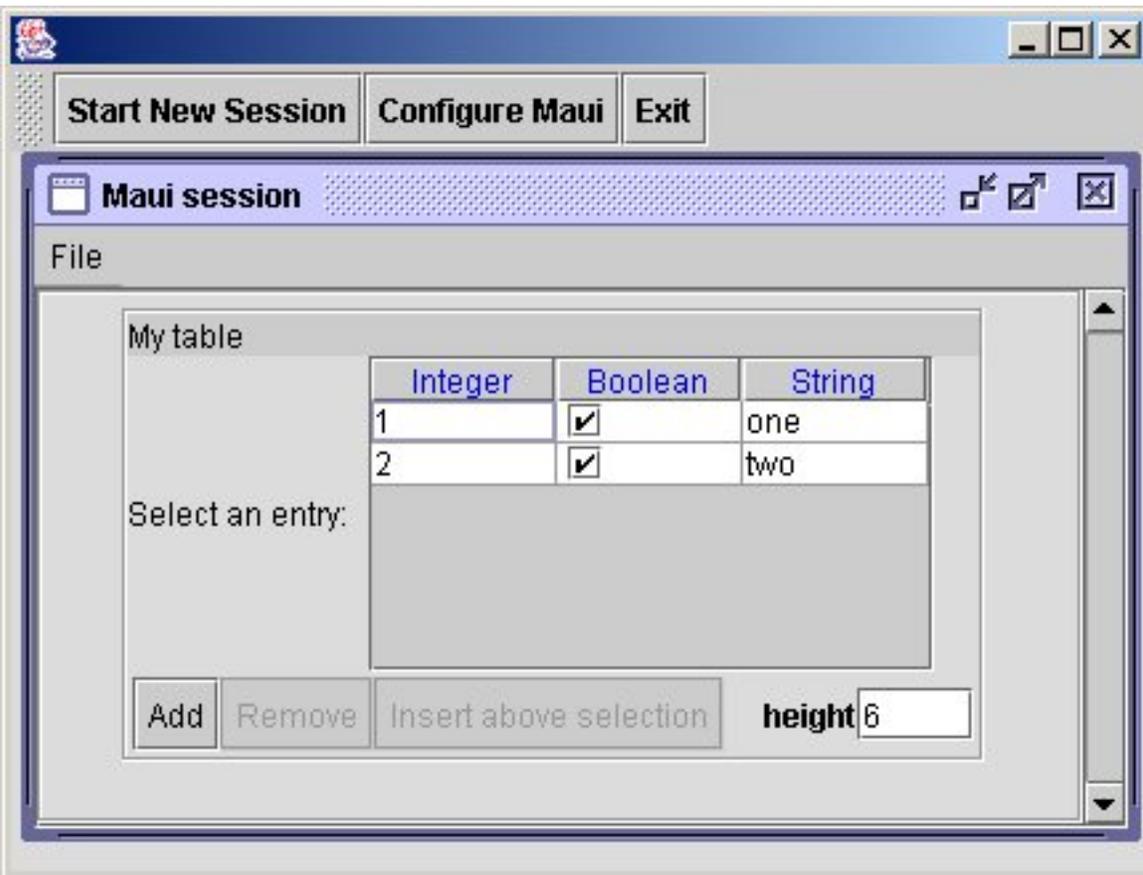
name: the name of the row

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>

          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>

          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>

        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```



label: (Author's Note: I will ask the developer if this attribute does anything useful)

[Next](#) [Up](#) [Previous](#)

Next: [B.23 Tag Cell Up](#) **Up:** [B.22 Tag Entry](#) **Previous:** [B.22.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.23.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.22.2 Attributes allowed in](#)

B.23 Tag Cell

Cell elements represent one initial value in a table entry.

Subsections

- [B.23.1 Children allowed in Cell elements](#)
 - [B.23.2 Attributes allowed in Cell elements](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [B.23.2 Attributes allowed in](#) **Up:** [B.23 Tag Cell](#) **Previous:** [B.23 Tag Cell](#)

B.23.1 Children allowed in Cell elements

[None](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.23.2 Attributes allowed in](#) **Up:** [B.23 Tag Cell](#) **Previous:** [B.23 Tag Cell](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.24 Tag type_name](#) **Up:** [B.23 Tag Cell](#) **Previous:** [B.23.1 Children allowed in](#)

B.23.2 Attributes allowed in Cell elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|-----------------|--|-------------------------|
| field | yes | a column name | the column to give an initial value for. | example |
| value | yes | see column type | the initial value for the given column. | example |

<Cell> : The GUI components (buttons, textboxes, checkboxes, etc) that are in one cell of a table.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>

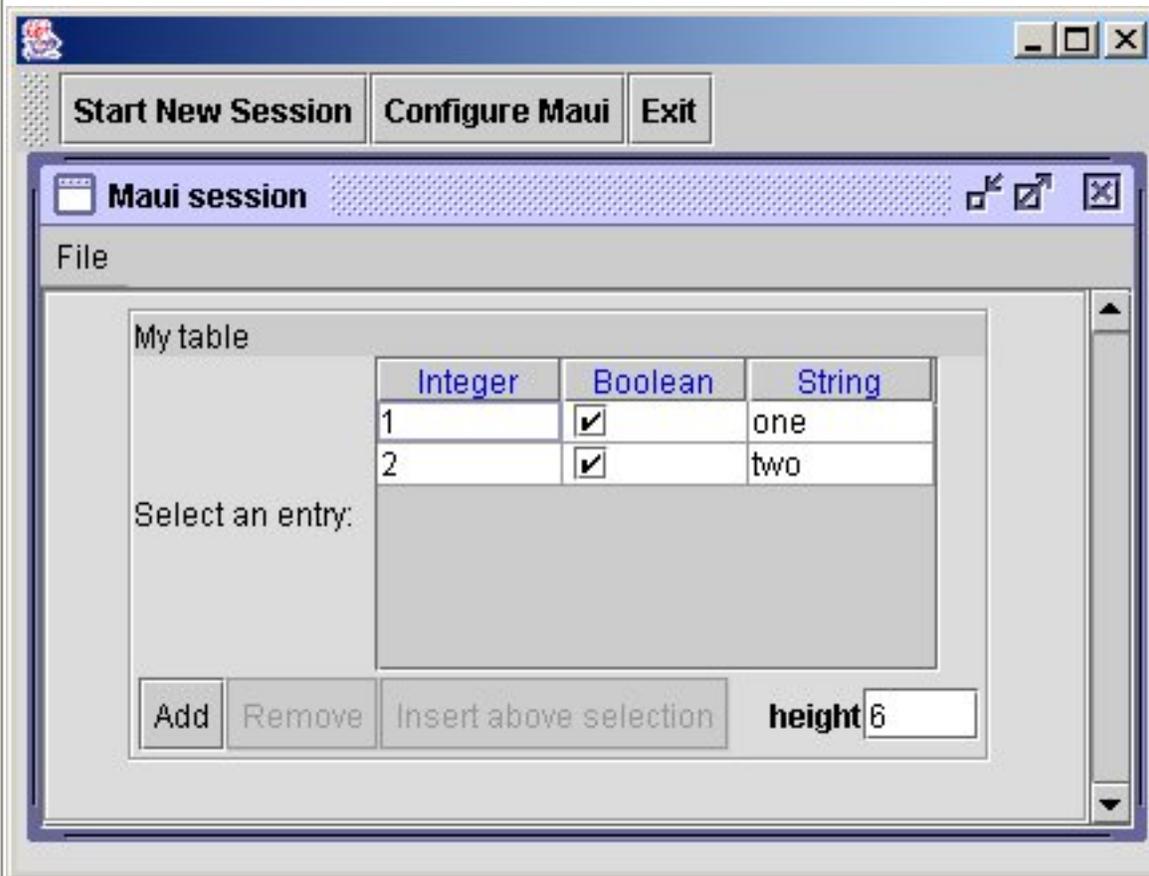
          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>

          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>
        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```

```

        </Entries>
    </Table>
</Fields>
</Class>
</Maui>

```



field: The name of a column.

The contents of this cell will be placed in this column.

```

<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>

```

```

<Entry name="entry1">
  <Cell field="Col1" value="1" />
  <Cell field="Col2" value="true" />
  <Cell field="Col3" value="one" />
</Entry>

```

```

<Entry name="entry2">
  <Cell field="Col1" value="2" />
  <Cell field="Col2" value="true" />
  <Cell field="Col3" value="two" />
</Entry>

```

```

</Entries>

```

```

</Table>

```

```

</Fields>

```

```

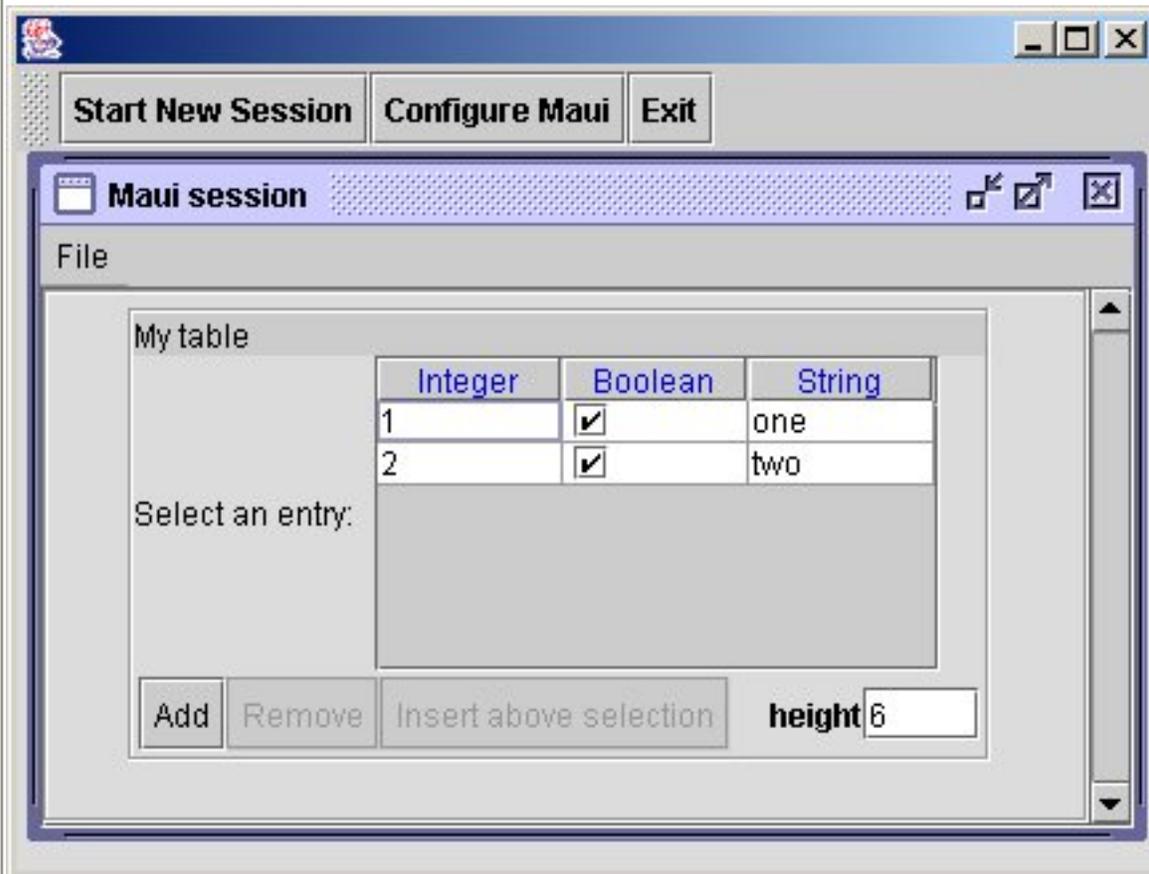
</Class>

```

```

</Maui>

```



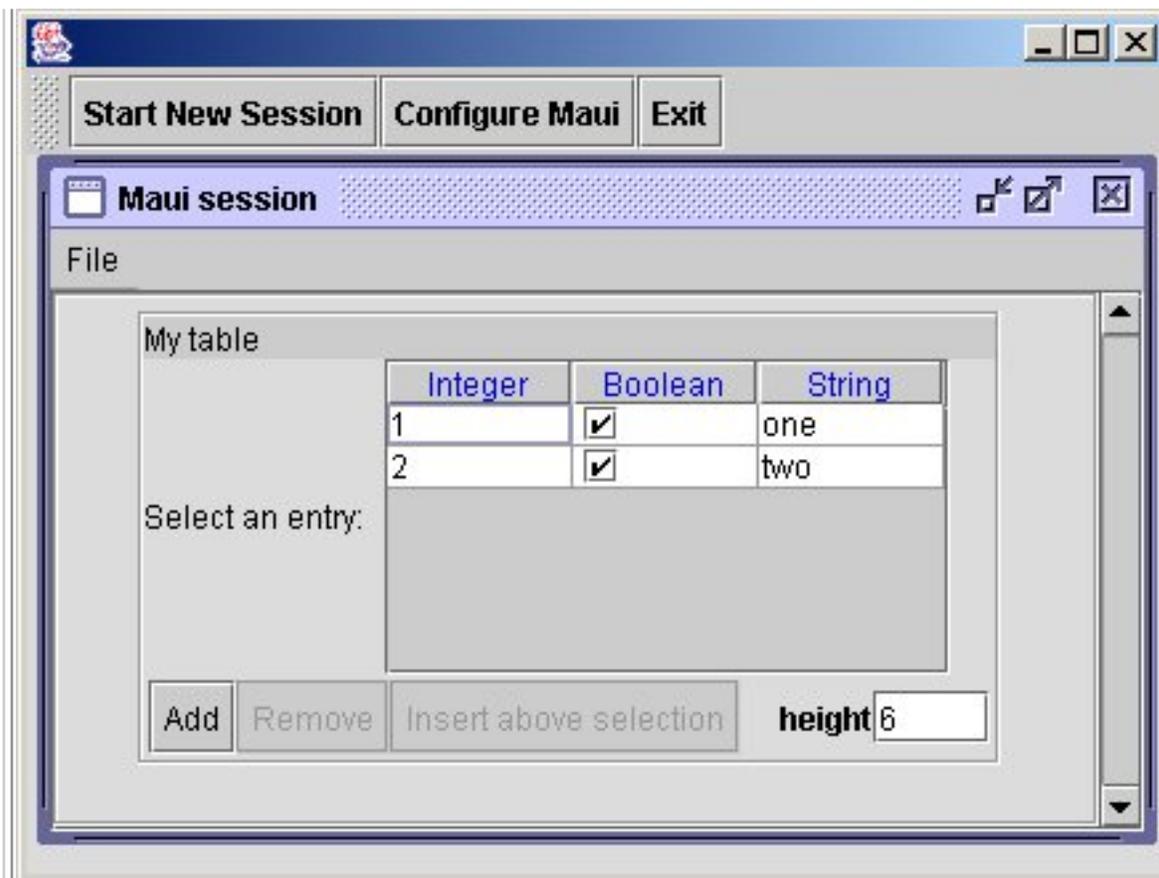
value: The contents of one cell in a table

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Table name="MyTable" label="My table">
        <Header name="columns" label="columns">
          <Integer name="Col1" label="Integer" />
          <Boolean name="Col2" label="Boolean" />
          <String name="Col3" label="String"/>
        </Header>
        <Entries>

          <Entry name="entry1">
            <Cell field="Col1" value="1" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="one" />
          </Entry>

          <Entry name="entry2">
            <Cell field="Col1" value="2" />
            <Cell field="Col2" value="true" />
            <Cell field="Col3" value="two" />
          </Entry>

        </Entries>
      </Table>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Next: [B.24 Tag type_name](#) **Up:** [B.23 Tag Cell](#) **Previous:** [B.23.1 Children allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.24.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.23.2 Attributes allowed in](#)

B.24 Tag *type_name*

type_name elements are defined by a `Class` element where *type_name* is given by the `type` attribute of `Class`.

Subsections

- [B.24.1 Children allowed in *type_name* elements](#)
- [B.24.2 Attributes allowed in *type_name* elements](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.24.1 Children allowed in](#) **Up:** [B. Maui XML syntax](#) **Previous:** [B.23.2 Attributes allowed in](#)

[Next](#) [Up](#) [Previous](#)

Next: [B.24.2 Attributes allowed in Up](#) **Up:** [B.24 Tag type_name](#) **Previous:** [B.24 Tag type_name](#)

B.24.1 Children allowed in *type_name* elements

| Tag | Number | Description and comments | Examples |
|--------------|------------|---|-------------------------|
| Action | any number | Allows for buttons to be placed within the class editor | example |
| Help | any string | A help icon is displayed. Whenever the end-user clicks on the icon, helpful text pops up on the screen. | example |
| CustomEditor | 0-1 | Allows the Class to use a non-standard editor. | example |
| AppData | 0-1 | a free form block of XML used for data where no editor is needed | example |

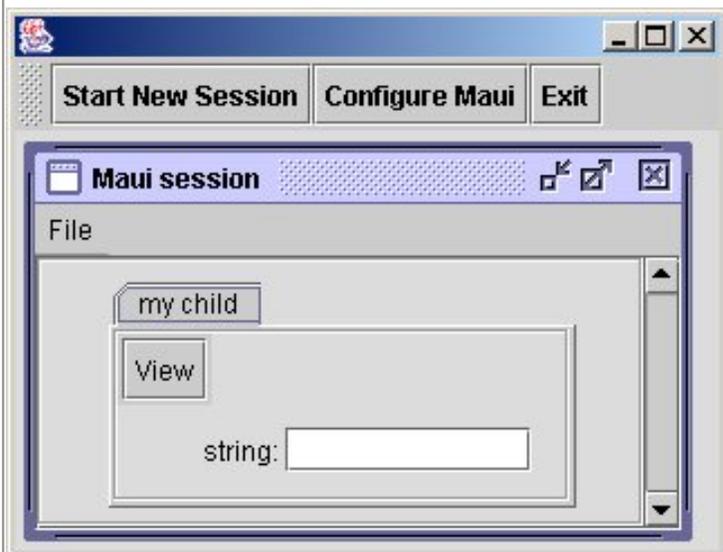
child class : A class is a container that is used to hold GUI components (buttons, textboxes, checkboxes, etc.)

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"/>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



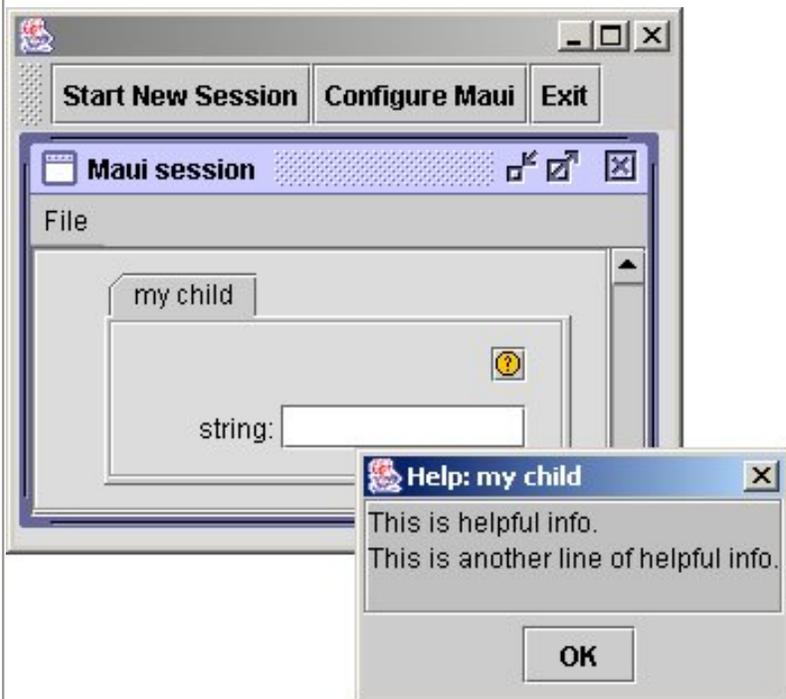
<Action> : Insert a button on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child">
        <Action label="View" class="Maui.Interface.ViewAction"/>
      </MyChild>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



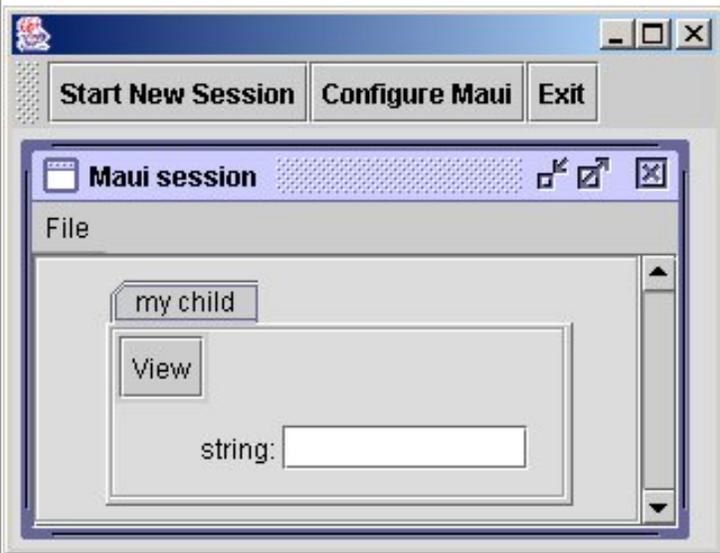
<Help> : Display helpful info on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child">
        <Help>
          This is helpful info.
          This is another line of helpful info.
        </Help>
      </MyChild>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



<CustomEditor> : Used to insert a custom editor into Maui

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child">
        <CustomEditor
          name="Maui.Editors.ExampleCustomEditor_BareBonesCustomEditor">
        </CustomEditor>
      </MyChild>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



<AppData> : Used to store data that you want hidden from the end-user. Also used to pass information to MauiActions, custom editors, and external applications.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child">
        <AppData>
          <parameter1>one</parameter1>
          <parameter2>two</parameter2>
        </AppData>
      </MyChild>
    </Fields>
  </Class>
  <Class type="MyChild">
```

```
    <Fields>
      <String name="myString" label="string" />
    </Fields>
  </Class>
</Maui>
```

[Next](#) [Up](#) [Previous](#)

Next: [B.24.2 Attributes allowed in](#) **Up:** [B.24 Tag type_name](#) **Previous:** [B.24 Tag type_name](#)

[Next](#) [Up](#) [Previous](#)

Up: [B.24 Tag type_name](#) **Previous:** [B.24.1 Children allowed in](#)

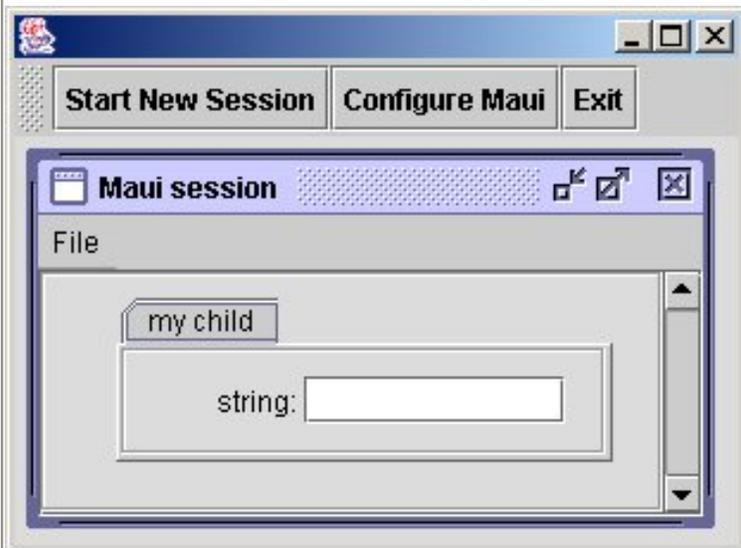
B.24.2 Attributes allowed in *type_name* elements

| Attribute name | Mandatory | Allowed values | Description and comments | Examples |
|----------------|-----------|----------------|--|-------------------------|
| name | yes | any legal name | name for this variable | example |
| label | no | any string | string to be used as a descriptive label | example |
| selectionLabel | no | any string | The label displayed next to the subclass Selection menu if this class has subclasses | example |
| collapsible | no | true or false | The panel for this class can be expanded and collapsed with a toggle button. If no value is given, collapsible is assumed to be false. | example |
| beginCollapsed | no | true or false | If the panel is collapsible, this will give the initial state of that panel. By default, a collapsible panel starts out expanded. | example |
| useTab | no | true or false | If this is true, the panel will be displayed in a tabbed pane with the other class panels. If false, it will be displayed by itself. If there is no collapsible attribute, useTab will default to true. If there is a collapsible attribute, useTab will default to false. | example |
| visible | no | true or false | Determines if the class is visible or invisible. | example |
| tooltip | no | any string | not yet implemented | example |
| layout | no | flow | If layout is set to "flow" then the GUI components are laid out from left to right, top to bottom. | example |

| | | | | |
|--------|----|---------------|--|-------------------------|
| border | no | true or false | Determines if the border (that surrounds the class) is visible or invisible. | example |
|--------|----|---------------|--|-------------------------|

child class : A class is a container that is used to hold GUI components (buttons, textboxes, checkboxes, etc.)

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"/>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



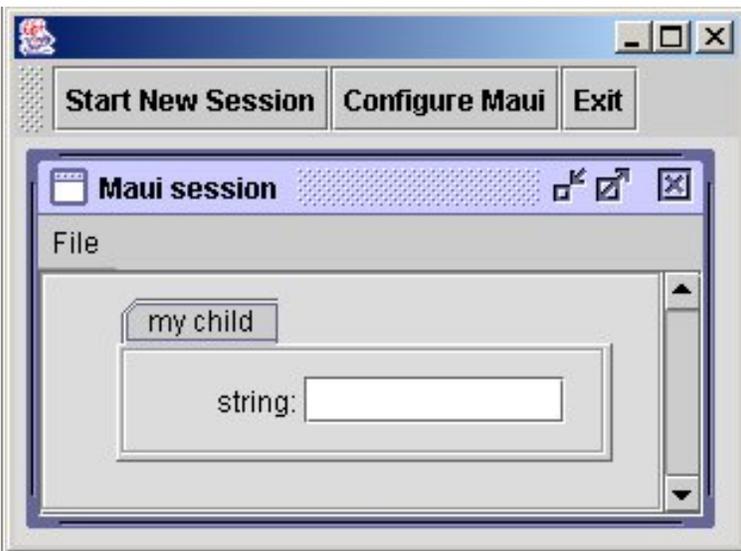
name: the name of the class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"/>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



label: The label that appears above the class

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"/>
    </Fields>
  </Class>
  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```



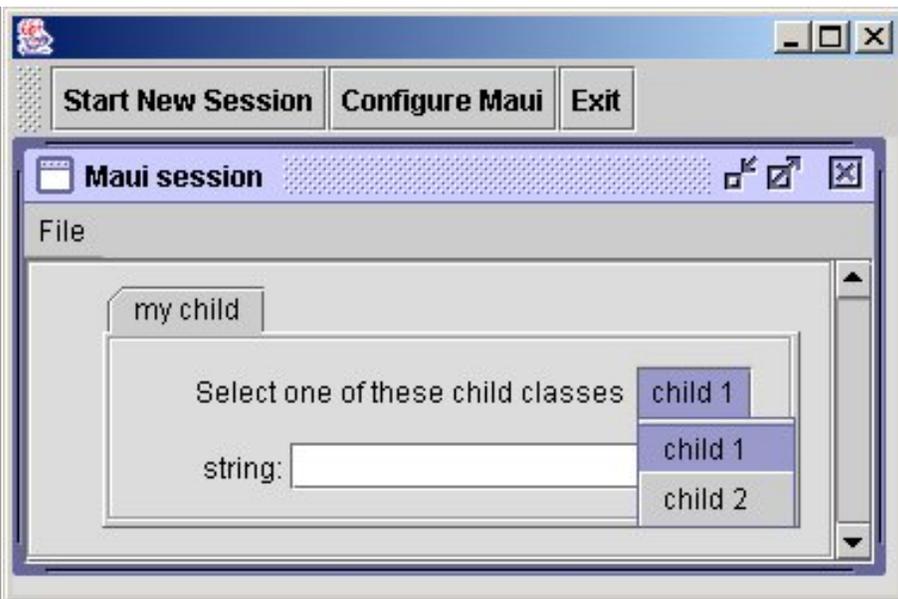
selectionLabel: if the end-user can use a pull-down menu to select a child class then this label appears to the left of the menu.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"
        selectionLabel="Select one of these child classes">
      </MyChild>
    </Fields>
  </Class>

  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>

  <Class type="Child1" label="child 1" base="MyChild"/>
  <Class type="Child2" label="child 2" base="MyChild"/>

</Maui>
```



collapsible: The contents of a class can be hidden by clicking on the +/- button.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child" collapsible="true"/>
    </Fields>
  </Class>

  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>

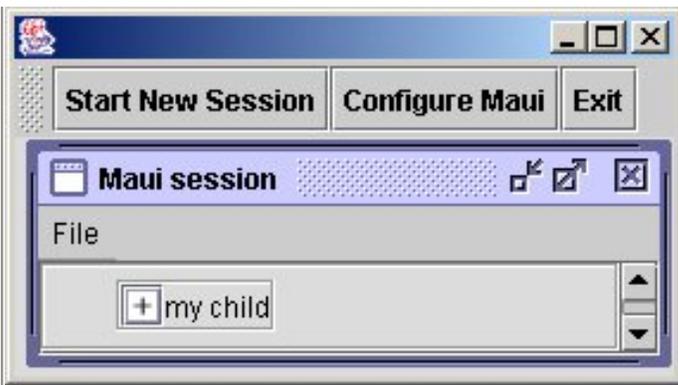
  </Class>
</Maui>
```



beginCollapsed: When the class first appears on the screen, is the class collapsed or uncollapsed?

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"
                collapsible="true" beginCollapsed="true"/>
    </Fields>
  </Class>

  <Class type="MyChild">
    <Fields>
      <String name="myString" label="string"/>
    </Fields>
  </Class>
</Maui>
```

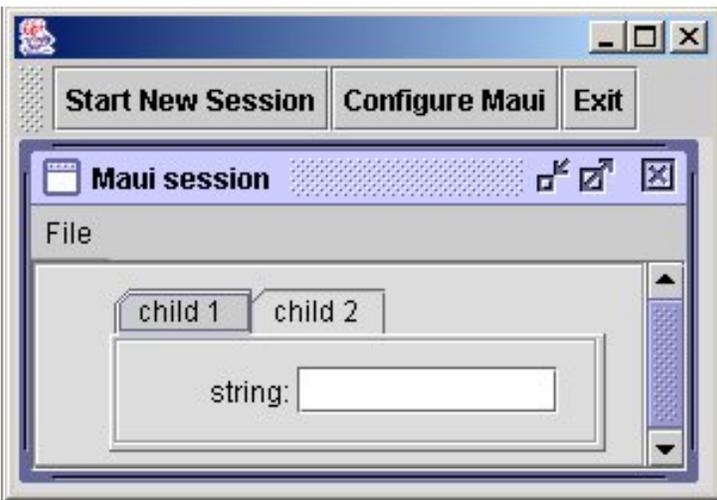


useTab: The end-user can select a child class by clicking on a tab

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Child1 name="child1" label="child 1" useTab="true"/>
      <Child2 name="child2" label="child 2" useTab="true"/>
    </Fields>
  </Class>

  <Class type="Child1">
    <Fields>
      <String name="string1" label="string"/>
    </Fields>
  </Class>

  <Class type="Child2">
    <Fields>
      <String name="string2" label="string"/>
    </Fields>
  </Class>
</Maui>
```

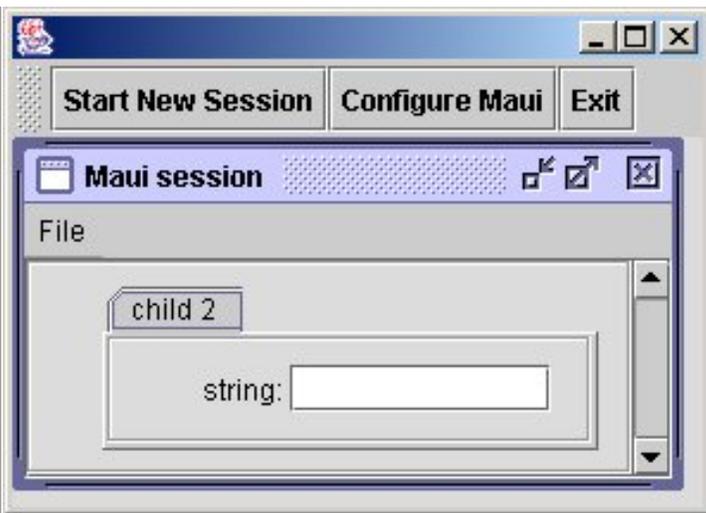


visible: Determines if the class is visible on the screen

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Child1 name="child1" label="child 1"
        useTab="true" visible="false"/>
      <Child2 name="child2" label="child 2"
        useTab="true" visible="true"/>
    </Fields>
  </Class>

  <Class type="Child1">
    <Fields>
      <String name="string1" label="string"/>
    </Fields>
  </Class>

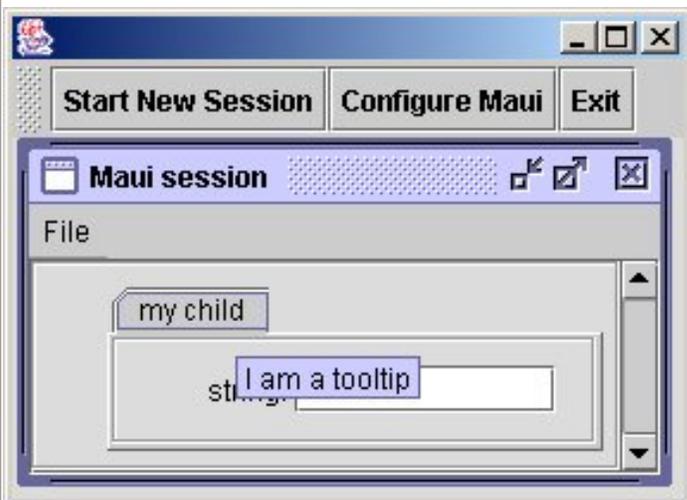
  <Class type="Child2">
    <Fields>
      <String name="string2" label="string"/>
    </Fields>
  </Class>
</Maui>
```



tooltip: Display a tooltip

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child"
        tooltip="I am a tooltip"/>
    </Fields>
  </Class>

  <Class type="MyChild">
    <Fields>
      <String name="string" label="string"/>
    </Fields>
  </Class>
</Maui>
```

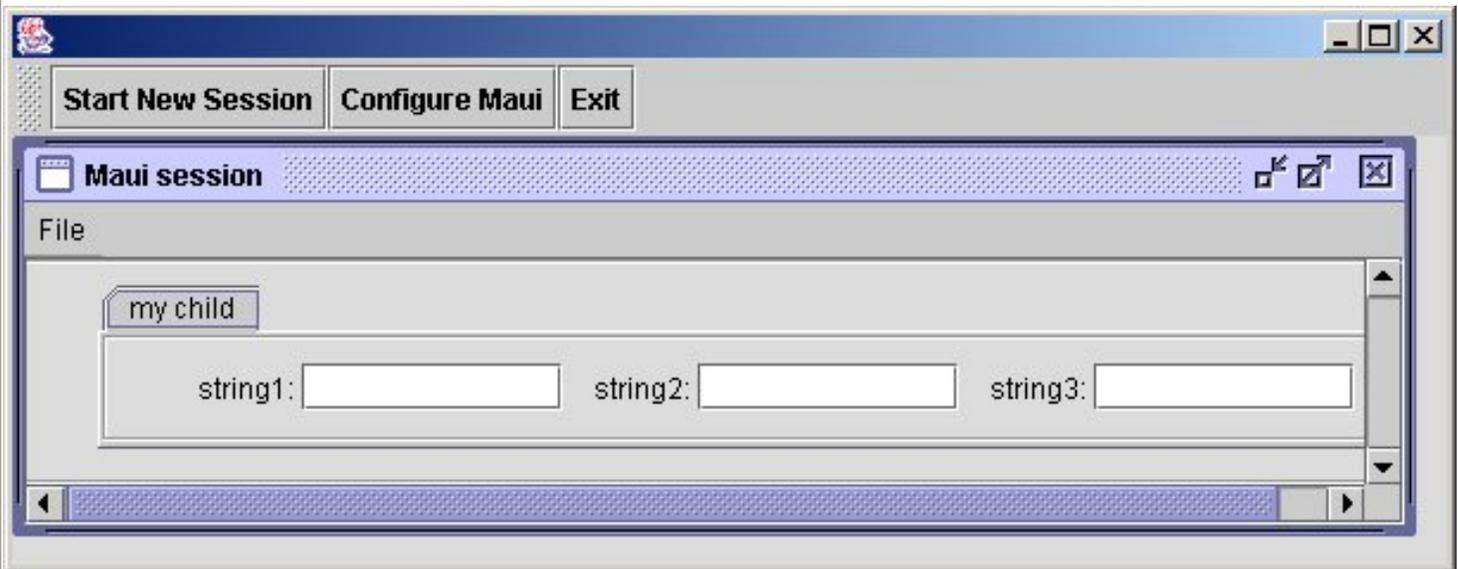


layout: if set to "flow" then GUI components are laid out from left-to-right, top-to-bottom.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <MyChild name="myChild" label="my child" layout="flow"/>
    </Fields>
  </Class>

  <Class type="MyChild">
    <Fields>
      <String name="string1" label="string1"/>
      <String name="string2" label="string2"/>
      <String name="string3" label="string3"/>
    </Fields>
  </Class>

</Maui>
```

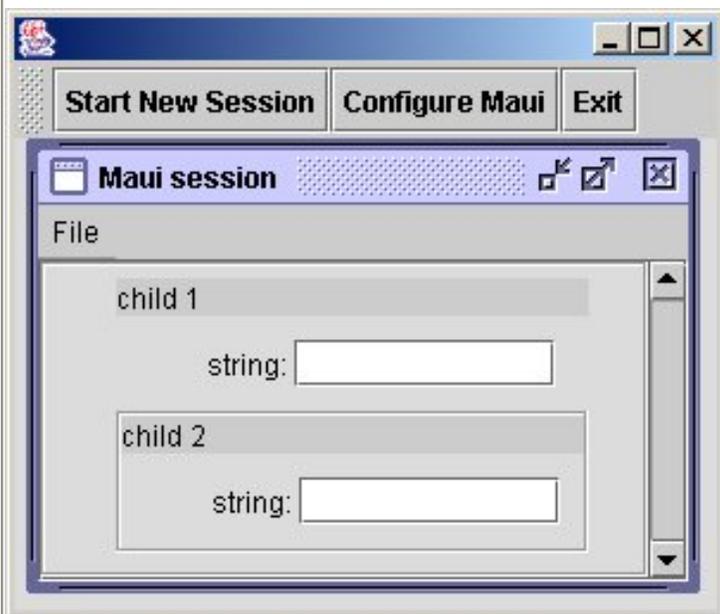


border: Determines if a border is drawn around the class.

```
<Maui RootClass="MyContainer">
  <Class type="MyContainer">
    <Fields>
      <Child1 name="child1" label="child 1"
        useTab="no" border="false"/>
      <Child2 name="child2" label="child 2"
        useTab="no" border="true"/>
    </Fields>
  </Class>

  <Class type="Child1">
    <Fields>
      <String name="string1" label="string"/>
    </Fields>
  </Class>

  <Class type="Child2">
    <Fields>
      <String name="string2" label="string"/>
    </Fields>
  </Class>
</Maui>
```



[Next](#) [Up](#) [Previous](#)

Up: [B.24 Tag type_name](#) **Previous:** [B.24.1 Children allowed in](#)

