Sibelius 6

Using the ManuScript™ language

Edition 6
April 2009

Written by Jonathan Finn, James Larcombe, Yasir Assam, Simon Whiteside, Mike Copperwhite, Paul Walmsley, Graham Westlake and Michael Eastwood, with contributions from Andrew Davis and Daniel Spreadbury.

Contents

Introduction

ManuScript™ is a simple, music-based programming language developed to write plug-ins for the Sibelius music processor. The name was invented by Ben Sloman, a friend of Ben Finn's.

It is based on Simkin, an embedded scripting language developed by Simon Whiteside, and has been extended by him and the rest of the Sibelius team ever since. (Simkin is a spooky pet name for Simon sometimes found in Victorian novels.) For more information on Simkin, and additional help on the language and syntax, go to the Simkin website at www.simkin.co.uk.

Rationale

In adding a plug-in language to Sibelius we were trying to address several different issues:

- Music notation is complex and infinitely extensible, so some users will sometimes want to add to a music notation program to make it cope with these new extensions.
- It is useful to allow frequently repeated operations (e.g. opening a MIDI file and saving it as a score) to be automated, using a system of scripts or macros.
- Certain more complex techniques used in composing or arranging music can be partly automated, but there are too many to include as standard features in Sibelius.

There were several conditions that we wanted to meet in deciding what language to use:

- The language had to be simple, as we want normal users (not just seasoned programmers) to be able to use it.
- We wanted plug-ins to be usable on any computer, as the use of computers running both Windows and Mac OS is widespread in the music world.
- We wanted the tools to program in the language to be supplied with Sibelius.
- We wanted musical concepts (pitch, notes, bars) to be easily expressed in the language.
- We wanted programs to be able to talk to Sibelius easily (to insert and retrieve information from scores).
- We wanted simple dialog boxes and other user interface elements to be easily programmed.

C/C++, the world's "standard" programming language(s), were unsuitable as they are not easy for the non-specialist to use, they would need a separate compiler, and you would have to recompile for each different platform you wanted to support (and thus create multiple versions of each plug-in).

The language Java was more promising as it is relatively simple and can run on any platform without recompilation. However, we would still need to supply a compiler for people to use, and we could not express musical concepts in Java as directly as we could with a new language.

So we decided to create our own language that is interpreted so it can run on different platforms, integrated into Sibelius without any need for separate tools, and can be extended with new musical concepts at any time.

The ManuScript language that resulted is very simple. The syntax and many of the concepts will be familiar to programmers of C/C++ or Java. Built into the language are musical concepts (Score, Staff, Bar, Clef, NoteRest) that are instantly comprehensible.

Technical support

Since the ManuScript language is more the province of our programmers than our technical support team (who are not, in the main, programmers), we can't provide detailed technical help on it, any more than Sun will help you with Java programming. This document and the sample plug-ins should give you a good idea of how to do some simple programming fairly quickly.

We would welcome any useful plug-ins you write – email them to daniel.spreadbury@avid.com and we may put them on our web site; if we want to distribute the plug-in with Sibelius itself, we'll pay you for it.

Mailing list for plug-in developers

There is a growing community of plug-in developers working with ManuScript, and they can be an invaluable source of help when writing new plug-ins. To subscribe, send an email to majordomo@sibelius.com with the words **`subscribe plugin-dev`** in the body of the email.

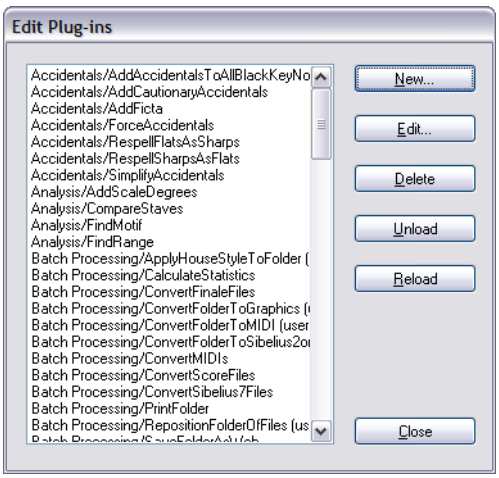
Tutorial

Edit Plug-ins

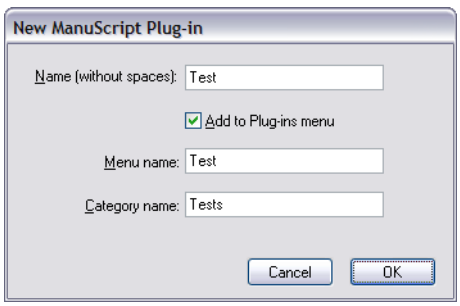
A simple plug-in

Let's start a simple plug-in. You are assumed to have some basic experience of programming (e.g. in BASIC or C), so you're already familiar with ideas like variables, loops and so on.

- Start Sibelius.
- Choose **Plug-ins ▸ Edit Plug-ins**. The following dialog appears:



- Now click **New**.

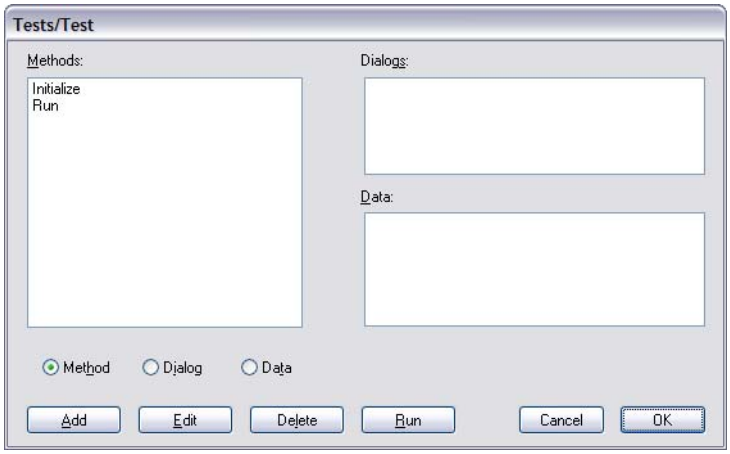


- You are asked to type the internal name of your plug-in (used as the plug-in's filename), the name that should appear on the menu and the name of the category in which the plug-in should appear on the **Plug-ins** menu.

  The reason for having two separate names for plug-ins is that filenames may be no longer than 31 characters on Macs running Mac OS 9 (which is only significant if you intend your plug-in to be used with versions of Sibelius prior to Sibelius 4), but the menu names can be as long as you like.
- Type **Test** as the internal name, **Test plug-in** as the menu name and **Tests** as the category name, then click **OK**.
- You'll see **Tests/Test (user copy)** added to the list in the **Edit Plug-ins** dialog. Click **Close**. This shows the folder in which the plug-in is located (**Tests**, which Sibelius has created for you), the filename of the plug-in (minus the standard **.plg** file extension), and **(user copy)** tells you that this plug-in is located in your user application data folder, not the Sibelius program folder or application package itself.
- If you look in the **Plug-ins** menu again you'll see a **Tests** submenu, with a **Test** plug-in inside it.
- Choose **Plug-ins ▸ Tests ▸ Test** and the plug-in will run. You may first be prompted that you cannot undo plug-ins, in which case click **Yes** to continue (and you may wish to switch on the **Don't say this again** option so that you're not bothered by this warning in future.) What does our new **Test** plug-in do? It just pops up a dialog which says **Test** (whenever you start a new plug-in, Sibelius automatically generates in a one-line program to do this). You'll also notice a window appear with a button that says **Stop Plug-in**, which appears whenever you run any plug-in, and which can be useful if you need to get out of a plug-in you're working on that is (say) trapped in an infinite loop.
- Click **OK** on the dialog and the plug-in stops.

Three types of information

Let's look at what's in the plug-in so far. Choose **Plug-ins ▸ Edit Plug-ins** again, then select **Tests/Test (user copy)** from the list and click **Edit** (or simply double-click the plug-in's name to edit it). You'll see a dialog showing the three types of information that can make up a plug-in:



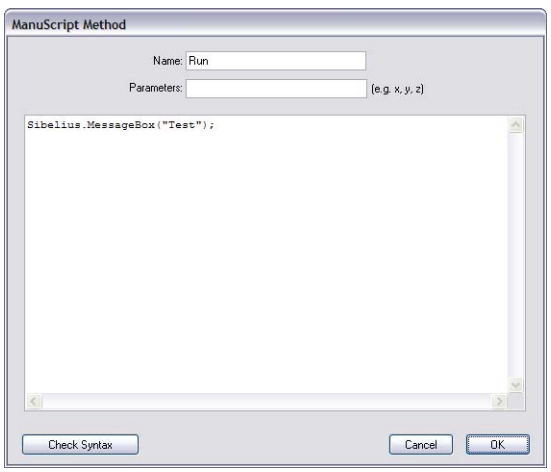
- **Methods**: similar to procedures, functions or routines in some other languages.
- **Dialogs**: the layout of any special dialog boxes you design for your plug-in.
- **Data**: variables whose value is remembered between running the plug-in. You can only store strings in these variables, so they're useful for things like user-visible strings that can be displayed when the plug-in runs. For a more sophisticated approach to global variables, ManuScript provides custom user properties for all objects – see **User properties** on page 21.

Methods

The actual program consists of the methods. As you can see, plug-ins normally have at least two methods, which are created automatically for you when you create a new plug-in:

- **Initialize**: this method is called automatically whenever you start up Sibelius. Normally it does nothing more than add the name of the plug-in to the **Plug-ins** menu, although if you look at some of the supplied plug-ins you'll notice that it's sometimes also used to set default values for data variables.
- **Run**: this is called when you run the plug-in, you'll be startled to hear (it's like **`main()`** in C/C++ and Java). In other words, when you choose **Plug-ins ▸ Tests ▸ Test**, the plug-in's **Run** method is called. If you write any other methods, you have to call them from the **Run** method - otherwise how can they ever do anything?

Click on **Run**, then click **Edit** (or you can just double-click **Run** to edit it). This shows a dialog where you can edit the **Run** method:



In the top field you can edit the name; in the next field you can edit the parameters (i.e. variables where values passed to the method are stored); and below is the code itself:

```
Sibelius.MessageBox("Test");
```

This calls a method **MessageBox** which pops up the dialog box that says Test when you run the plug-in. Notice that the method name is followed by a list of parameters in parentheses. In this case there's only one parameter: because it's a string (i.e. text) it's in double quotes. Notice also that the statement ends in a semicolon, as in C/C++ and Java. If you forget to type a semicolon, you'll get an error when the plug-in runs.

What is the role of the word Sibelius in **Sibelius.MessageBox**? In fact it's a variable representing the Sibelius program; the statement is telling Sibelius to pop up the message box (C++ and Java programmers will recognize that this variable refers to an "object"). If this hurts your brain, we'll go into it later.

Editing the code

Now try amending the code slightly. You can edit the code just like in a word processor, using the mouse and arrow keys, and you can also also use Ctrl+X/C/V *or* ⌘X/C/V for cut, copy and paste respectively. If you right-click (Windows) or Control-click (Mac) you get a menu with these basic editing operations on them too.

Change the code to this:

```
x = 1;
   x = x + 1;
      Sibelius.MessageBox("1 + 1 = " & x);
```

You can check this makes sense (or, at least, some kind of sense) by clicking the Check syntax button. If there are any blatant mistakes (e.g. missing semicolons) you'll be told where they are.

Then close the dialogs by clicking OK, OK again then Close. Run your amended plug-in from the Plug-ins menu and a message box with the answer 1 + 1 = 2 should appear.

How does it work? The first two lines should be obvious. The last line uses **&** to stick two strings together. You can't use **+** as this works only for numbers (if you try it in the example above, you'll get an interesting answer!).

One pitfall: try changing the second line to:

```
x += 1;
```

then click Check syntax. You'll get an error: this syntax (and the syntax **x++**) is allowed in various languages but not in ManuScript. You have to do **x = x+1;**.

Where plug-ins are stored

Plug-ins supplied with Sibelius are stored in the folder called Plugins inside the Sibelius program folder on Windows, and inside the application package (or "bundle") on Mac. It is not intended that end users should add extra plug-ins to these locations themselves, as we have provided a per-user location for plug-ins to be installed instead. When you create a new plug-in or edit an existing one, the new or modified plug-in will be saved into the per-user location (rather than modifying or adding to the plug-ins in the program folder or bundle):

- On Windows, additional plug-ins are stored in subfolders at C:\Documents and Settings*username*\Application Data\Sibelius Software\Sibelius 5\Plugins.
- On Mac, additional plug-ins are stored in subfolders at /Users/*username*/Library/Application Support/Sibelius Software/Sibelius 5/Plugins.

This is worth knowing if you want to give a plug-in to someone else. The plug-ins appear in subfolders which correspond to the submenus in which they appear in the Plug-ins menu. The filename of the plug-in itself is the plug-in's internal name plus the .plg extension, e.g. Test.plg.

Line breaks and comments

As with C/C++ and Java, you can put new lines wherever you like (except in the middle of words), as long as you remember to put a semicolon after every statement. You can put several statements on one line, or put one statement on several lines.

You can add comments to your program, again like C/C++ and Java. Anything after **//** is ignored to the end of the line. Anything between **/*** and ***/** is ignored, whether just part of a line or several lines:

```
// comment lasts to the end of the line
/* you can put
```

```
several lines of comments here
*/
```

For instance:

```
Sibelius.MessageBox("Hi!");      // print the active score
```

or:

```
Sibelius /* this contains the application */ .MessageBox("Hi!");
```

Variables

x in the Test plug-in is a variable. In ManuScript a variable can be any sequence of letters, digits or _ (underscore), as long as it doesn't start with a digit.

A variable can contain an integer (whole number), a floating point number, a string (text) or an object (e.g. a note) – more about objects in a moment. Unlike most languages, in ManuScript a variable can contain any type of data – you don't have to declare what type you want. Thus you can store a number in a variable, then store some text instead, then an object. Try this:

```
x = 56; x = x+1;
Sibelius.MessageBox(x);         // prints '57' in a dialog box
x = "now this is text";         // the number it held is lost
Sibelius.MessageBox(x);         // prints 'now this is text' in a dialog
x = Sibelius.ActiveScore;       // now it contains a score
Sibelius.MessageBox(x);          // prints nothing in a dialog
```

Variables that are declared within a ManuScript method are local to that method; in other words, they cannot be used by other methods in the same plug-in. Global Data variables defined using the plug-in editor can be accessed by all methods in the plug-in, and their values are preserved over successive uses of the plug-in.

A quick aside about strings in ManuScript is in order at this point. Like many programming languages, ManuScript strings uses the back-slash **** as an "escape character" to represent certain special things. To include a single quote character in your strings, use **\'**, and to include a new line you should use **\n**. Because of this, to include the backslash itself in a ManuScript string one has to write **\'**.

Converting between numbers, text and objects

Notice that the method **MessageBox** is expecting to be sent some text to display. If you give it a number instead (as in the first call to **MessageBox** above) the number is converted to text. If you give it an object (such as a score), no text is produced.

Similarly, if a calculation is expecting a number but is given some text, the text will be converted to a number:

```
x = 1 + "1";                        // the + means numbers are expected
   Sibelius.MessageBox(x);           // displays '2'
```

If the text doesn't start with a number (or if the variable contains an object instead of text), it is treated as 0:

```
x = 1 + "fred";
   Sibelius.MessageBox(x);           // displays ‘1’
```

Loops

"for" and "while"

ManuScript has a **while** loop which repeatedly executes a block of code until a certain expression becomes True. Create a new plug-in called Potato. This is going to amuse one and all by writing the words of the well-known song "1 potato, 2 potato, 3 potato, 4". Type in the following for the Run method of the new plug-in:

```
x = 1;
while (x<5)
{
   text = x & " potato,";
   Sibelius.MessageBox(text);
   x = x+1;
}
```

Run it. It should display "1 potato," "2 potato," "3 potato," "4 potato," which is a start, though annoyingly you have to click OK after each message.

The **while** statement is followed by a condition in **( )** parentheses, then a block of statements in **{ }** braces (you don't need a semicolon after the final **}** brace). While the condition is true, the block is executed. Unlike some other languages, the braces are *compulsory* (you can't omit them if they only contain one statement). Moreover, each block must contain at least one statement. We did say that ManuScript was a simple language.

In this example you can see that we are testing the value of **x** at the start of the loop, and increasing the value at the end. This common construct could be expressed more concisely in ManuScript by using a **for** loop. The above example could also be written as follows:

```
for x = 1 to 5
{
   text = x & " potato,";
   Sibelius.MessageBox(text);
}
```

Here, the variable **x** is stepped from the first value (1) up to the end value (5), stopping one step before the final value. By default, the "step" used is 1, but we could have used (say) 2 by using the syntax **for x = 1 to 5 step 2**, which would then print only "1 potato" and "3 potato"!

Notice the use of **&** to add strings. Because a string is expected on either side, the value of **x** is turned into a string.

Notice also we've used the Tab key to indent the statements inside the loop. This is a good habit to get into as it makes the structure clearer. If you have loops inside loops you should indent the inner loops even more.

The if statement

Now we can add an **if** statement so that the last phrase is just "4," not "4 potato":

```
x = 1;
while (x<5)
{
   if(x=4)
   {
      text = x & ".";
   }
   else
   {
      text = x & " potato,";
   }
   Sibelius.MessageBox(text);
   x = x+1;
}
```

The rule for **`if`** takes the form **`if (`***condition***`) {`***statements***`}`**. You can also optionally add **`else {`***statements***`}`**, which is executed if the condition is false. As with **`while`**, the parentheses and braces are compulsory, though you can make the program shorter by putting braces on the same line as other statements:

```
x = 1;
while (x<5)
{
   if(x=4) {
      text = x & ".";
   } else {
      text = x & " potato,";
   }
   Sibelius.MessageBox(text);
   x = x+1;
}
```

The position of braces is entirely a matter of taste.

Now let's make this plug-in really cool. We can build up the four messages in a variable called **`text`**, and only display it at the end, saving valuable wear on your mouse button. We can also switch round the **`if`** and **`else`** blocks to show off the use of **`not`**. Finally, we return to the **`for`** syntax we looked at earlier.

```
text = "";                                  // start with no text
for x = 1 to 5
{
   if (not(x=4)) {
      text = text & x & " potato, ";        // add some text
} else {
      text = text & x & ".";                // add no. 4
   }
}

Sibelius.MessageBox(text);                   // finally display it
```

Arithmetic

We've been using **`+`** without comment, so here's a complete list of the available arithmetic operators:

`a + b`	add
`a - b`	subtract
`a * b`	multiply
`a / b`	divide
`a % b`	remainder
`-a`	negate
`(a)`	evaluate first

ManuScript evaluates operators strictly from left-to-right, unlike many other languages; so **`2+3*4`** evaluates to 20, not 14 as you might expect. To get the answer 20, you'd have to write **`2+(3*4)`**.

ManuScript also supports floating point numbers, so whereas in some early versions **`3/2`** would work out as 1, it now evaluates to 1.5. Conversion from floating point numbers to integers is achieved with the **`RoundUp(`***expr***`)`**, **`RoundDown(`***expr***`)`** and **`Round(`***expr***`)`** functions, which can be applied to any expression.

Objects

Now we come to the neatest aspect of object-oriented languages like ManuScript, C++ or Java, which sets them apart from traditional languages like BASIC, Fortran and C. Variables in traditional languages can hold only certain types of data: integers, floating point numbers, strings and so on. Each type of data has particular operations you can do to it: numbers can be multiplied and divided, for instance; strings can be added together, converted to and from numbers, searched for in other strings, and so on. But if your program deals with more complex types of data, such as dates (which in principle you could compare using **=**, **<** and **>**, convert to and from strings, and even subtract) you are left to fend for yourself.

Object-oriented languages can deal with more complex types of data directly. Thus in the ManuScript language you can set a variable, let's say **thischord**, to be a chord in your score, and (say) add more notes to it:

```
thischord.AddNote(60);          // adds middle C (note no. 60)
thischord.AddNote(64);           // adds E (note no. 64)
```

If this seems magic, it's just analogous to the kind of things you can do to strings in BASIC, where there are very special operations which apply to text only:

```
A$ = "1"
A$ = A$ + " potato, ":          REM add strings
X = ASC(A$):                     REM get first letter code
```

In ManuScript you can set a variable to be a chord, a note in a chord, a bar, a staff or even a whole score, and do things to it. Why would you possibly want to set a variable to be a whole score? So you can save it or add an instrument to it, for instance.

Objects in action

We'll have a look at how music is represented in ManuScript in a moment, but for a little taster, let's plunge straight in and adapt Potato to create a score:

```
x = 1;
text = "";                                  // start with no text
while (x<5)
{
   if (not(x=4)) {
      text = text & x & " potato, ";        // add some text
   } else {
      text = text & x & ".";                // add no. 4
   }
   x = x+1;
}

Sibelius.New();                             // create a new score
newscore = Sibelius.ActiveScore;            // put it in a variable
newscore.CreateInstrument("Piano");
staff = newscore.NthStaff(1);               // get top staff
bar = staff.NthBar(1);                      // get bar 1 of this staff
bar.AddText(0,text,"Technique");              // use Technique text style
```

This creates a score with a Piano, and types our potato text in bar 1 as Technique text.

The code uses the period (**.**) several times, always in the form **variable.variable** or **variable.method()**. This shows that the variable before the period has to contain an object.

- If there's a variable name after the period, we're getting one of the object's sub-variables (called "fields" or "member variables" in some languages). For instance, if **n** is a variable containing a note, then **n.Pitch** is a number representing its MIDI pitch (e.g. 60 for middle C), and **n.Name** is a string describing its pitch (e.g. "C4" for middle C). The variables available for each type of object are listed later.
- If there's a method name after the period (followed by **()** parentheses), one of the methods allowed for this type of object is called. Typically a method called in this way will either change the object or return a value. For instance, if **s** is a variable con-

taining a score, then **`s.CreateInstrument("Flute")`** adds a flute (changing the score), but **`s.NthStaff(1)`** returns a value, namely an object containing the first staff.

Let's look at the new code in detail. There is a pre-defined variable called Sibelius, which contains an object representing the Sibelius program itself. We've already seen the method **`Sibelius.MessageBox()`**. The method call **`Sibelius.New()`** tells Sibelius to create a new score. Now we want to do something to this score, so we have to put it in a variable.

Fortunately, when you create a new score it becomes active (i.e. its title bar highlights and any other scores become inactive), so we can just ask Sibelius for the active score and put it in a variable:

```
newscore = Sibelius.ActiveScore.
```

Then we can tell the score to create a Piano: **`newscore.CreateInstrument("Piano")`**. But to add some text to the score you have to understand how the layout is represented.

Representation of a score

A score is treated as a hierarchy: each score contains 0 or more staves; each staff contains bars (though every staff contains the same number of bars); and each bar contains "bar objects." Clefs, text and chords are all different types of bar objects.

So to add a bar object (i.e. an object which belongs to a bar), such as some text, to a score: first you have to get which staff you want (and put it in a variable): **`staff = newscore.NthStaff(1)`**; then you have to get which bar in that staff you want (and put it in a variable): **`bar = staff.NthBar(1)`**; finally you tell the bar to add the text: **`bar.AddText(0,text,"Technique")`**. You have to give the name (or index number – see **Text styles** on page 121) of the text style to use (and it has to be a staff text style, because we're adding the text to a staff).

Notice that bars and staves are numbered from 1 upwards; in the case of bars, this is irrespective of any bar number changes that are in the score, so the numbering is always unambiguous. In the case of staves, the top staff is no.1, and all staves are counted, even if they're hidden. Thus a particular staff has the same number wherever it appears in the score.

The **`AddText`** method for bars is documented later, but the first parameter it takes is a *rhythmic position* in the bar. Each note in a bar has a rhythmic position that indicates where it is (at the start, one quarter after the start, etc.), but the same is true for all other objects in bars. This shows where the object is attached to, which in the case of Technique text is also where the left hand side of the text goes. Thus to put our text at the start of the bar, we used the value 0. To put the text a quarter note after the start of the bar, use 256 (the units are 1024th notes, so a quarter is 256 units – but don't think about this too hard):

```
bar.AddText(256,text,"Technique");
```

To avoid having to use obscure numbers like 256 in your program, there are predefined variables representing different note values (which are listed later), so you could write:

```
bar.AddText(Quarter,text,"Technique");
```

or to be quaint you could use the British equivalent:

```
bar.AddText(Crotchet,text,"Technique");
```

For a dotted quarter, instead of using 384 you can use another predefined variable:

```
bar.AddText(DottedQuarter,text,"Technique");
```

or add two variables:

```
bar.AddText(Quarter+Eighth,text,"Technique");
```

This is much clearer than using numbers.

The system staff

As you know from using Sibelius, some objects don't apply to a single staff but to all staves. These include titles, tempo text, rehearsal marks and special barlines; you can tell they apply to all staves because (for instance) they get shown in all the instrumental parts.

All these objects are actually stored in a hidden staff, called the system staff. You can think of it as an invisible staff which is always above the other staves in a system. The system staff is divided into bars in the same way as the normal staves. So to add the title "Potato" to our score we'd need the following code in our plug-in:

```
sys = newscore.SystemStaff;                // system staff is a variable
bar = sys.NthBar(1);
bar.AddText(0,"POTATO SONG","Subtitle");
```

As you can see, **`SystemStaff`** is a variable you can get directly from the score. Remember that you have to use a system text style (here I've used Subtitle) when putting text in a bar in the system staff. A staff text style like Technique won't work. Also, you have to specify a bar and position in the bar; this may seem slightly superfluous for text centered on the page as titles are (though in reality even this kind of page-aligned text is always attached to a bar), but for Tempo and Metronome mark text they are obviously required.

Representation of notes, rests, chords and other musical items

Sibelius represents rests, notes and chords in a consistent way. A rest has no noteheads, a note has 1 notehead and a chord has 2 or more noteheads. This introduces an extra hierarchy: most of the squiggles you see in a score are actually a special type of bar object that can contain even smaller things (namely, noteheads). There's no overall name for something which can be a rest, note or chord, so we've invented the pretty name NoteRest. A NoteRest with 0, 1 or 2 noteheads is what you normally call a rest, a note or a chord, respectively.

If **`n`** is a variable containing a NoteRest, there is a variable **`n.NoteCount`** which contains the number of notes, and **`n.Duration`** which is the note-value in 1/256ths of a quarter. You can also get **`n.Highest`** and **`n.Lowest`** which contain the highest and lowest notes (assuming **`n.NoteCount`** isn't 0). If you set **`lownote = n.Lowest`**, you can then find out things about the lowest note, such as **`lownote.Pitch`** (a number) and **`lownote.Name`** (a string). Complete details about all these methods and variables may be found in the **Reference** section below.

Other musical objects, such as clefs, lines, lyrics and key signatures have corresponding objects in ManuScript, which again have various variables and methods available. For example, if you have a Line variable **`ln,`** then **`ln.EndPosition`** gives the rhythmic position at which the line ends.

The "for each" loop

It's a common requirement for a loop to do some operation to every staff in a score, or every bar in a staff, or every BarObject in a bar, or every note in a NoteRest. There are other more complex requirements which are still common, such as doing an operation to every BarObject in a score in chronological order, or to every BarObject in a multiple selection. ManuScript has a **`for each`** loop that can achieve each of these in a single statement.

The simplest form of **`for each`** is like this:

```
thisscore = Sibelius.ActiveScore;
for each s in thisscore                          // sets s to each staff in turn
{
   // ...do something with s
}
```

Here, since **`thisscore`** is a variable containing a score, the variable **`s`** is set to be each staff in **`thisscore`** in turn. This is because staves are the type of object at the next hierarchical level of objects (see **Hierarchy of objects** on page 40). For each staff in the score, the statements in {} braces are executed.

Score objects contain staves, as we have seen, but they can also contain a Selection object, e.g. if the user has selected a passage of music before running the plug-in. The Selection object is a special case: it is never returned by a **`for each`** loop, because there is only a single selection object; if you use the Selection object in a **`for each`** loop, by default it will return BarObjects (not Staves, Bars or anything else!).

Let's take another example, this time for notes in a NoteRest:

```
noterest = bar.NthBarObject(1);
for each n in noterest                           // sets n to each note in turn
{
   Sibelius.MessageBox("Pitch is " & n.Name);
}
```

`n` is set to each note of the chord in turn, and its note name is displayed. This works because Notes are the next object down the hierarchy after NoteRests. If the NoteRest is, in fact, a rest (rather than a note or chord), the loop will never be executed – you don't have to check this separately.

The same form of loop will get the bars from a staff or system staff, and the BarObjects from a bar. These loops are often nested, so you can, for instance, get several bars from several staves.

This first form of the **`for each`** loop got a sequence of objects from an object in the next level of the hierarchy of objects. The second form of the **`for each`** loop lets you skip levels of the hierarchy, by specifying what type of object you want to get. This saves a lot of nested loops:

```
thisscore = Sibelius.ActiveScore;
for each NoteRest n in thisscore
{
   n.AddNote(60);                                 // add middle C
}
```

By specifying **`NoteRest`** after **`for each`**, Sibelius knows to produce each NoteRest in each bar in each staff in the score; otherwise it would just produce each staff in the score, because a Staff object is the type of object at the next hierarchical level of objects. The NoteRests are produced in a useful order, namely from the top to the bottom staff, then from left to right through the bars. This is chronological order. If you want a different order (say, all the NoteRests in the first bar in every staff, then all the NoteRests in the second bar in every staff, and so on) you'll have to use nested loops.

So here's some useful code that doubles every note in the score in octaves:

```
score = Sibelius.ActiveScore;
for each NoteRest chord in score
{
   if(not(chord.NoteCount = 0))           // ignore rests
   {
      note = chord.Highest;                // add above the top note
      chord.AddNote(note.Pitch+12);        // 12 is no. of half-steps (semitones)
   }
}
```

It could easily be amended to double in octaves only in certain bars or staves, only if the notes have a certain pitch or duration, and so on.

This kind of loop is also very useful in conjunction with the user's current selection. This selection can be obtained from a variable containing a Score object as follows:

```
selection = score.Selection;
```

We can then test whether it's a passage selection, and if so we can look at (say) all the bars in the selection by means of a **for each** loop:

```
if (selection.IsPassage)
{
   for each Bar b in selection
   {
      // do something with this bar
      …
   }
}
```

Be aware that you can not add or remove items from bars during iterating. The example of adding notes to chords above is fine because you are modifying an existing item (in this case a NoteRest), but it's not safe to add or remove entire items, and if you try to do so, your plug-in will abort with an error. However, it's very useful to add or remove items from bars, so you need to do that in a separate **for** loop, after first collecting the items you want to operate on into a ManuScript array, something like this:

```
num = 0;
for each obj in selection
{
    if (IsObject(obj))
        {
            n = "obj" & num;
            @n = obj;
            num = num + 1;
        }
}
selection.Clear();
for i = 0 to num
{
    n = "obj" & i;
    obj = @n;  // get an object from the pseudo array
    obj.Select();
}
```

The **@n** in this example is the array. To find out more about arrays, read on.

Indirection, sparse arrays and user properties

Indirection

If you put the **@** character before a string variable name, then the *value* of the variable is used as the name of a variable or method. For instance:

```
var="Name";
x = @var;                    // sets x to the contents of the variable Name

mymethod="Show";
@mymethod();                 // calls the method Show
```

This has many advanced uses, though if taken to excess it can cause the brain to hurt. For instance, you can use **@** to simulate "unlimited" arrays. If **name** is a variable containing the string **"x1"**, then **@name** is equivalent to using the variable **x1** directly. Thus:

```
i = 10;
name = "x" & i;
@name = 0;
```

sets variable **x10** to **0**. The last two lines are equivalent to **x[i] = 0;** in the C language. This has many uses; however, you'll also want to consider using the built-in arrays (and hash tables), which are documented below.

Sparse arrays

The method described above can be used to create "fake" arrays through indirection, though this is a little fiddly. ManuScript also provides Javascript-style sparse arrays, which can store anything that can be stored in a ManuScript variable, including references to objects. Like a variable, storing a reference to an object in a sparse array will preserve the lifetime of that object (because objects are reference counted), but the underlying object in Sibelius may become invalid if (say) a Score is modified.

To create a sparse array in ManuScript, use the built-in method **CreateSparseArray(***a1,a2,a3,a4...an***)**. You can create an empty array simply by passing in no variables to the **CreateSpareArray** method.

Sparse arrays provide a read/write variable called **Length** that returns or sets the length of the array: when you set **Length** to a number greater than the present size of the array, the array is padded with null values; if you set **Length** to a number smaller than the present size of the array, any values beyond this number are removed.

To push one or more values to the end of the array, use the method **Push(***a1, a2, ... an***)**. To remove and return the last element of an array, use the method **Pop()**.

An example of how to use a sparse array:

```
array = CreateSparseArray(4,5,6);
array[10] = 19; // creates 11th element of array, intervening elements are null
array.Length = 20; // extends array to 20 elements, new elements are all null
```

Sparse arrays by their nature may not have values in every array element. To return a new sparse array containing only the populated indices of the original sparse array (i.e. those that are not null), use the array's **ValidIndices()** method. For example, using the above sparse array:

```
array2 = array.ValidIndices(); // will contain values 0, 1, 2, 10 and 19
return array[array2[0]]; // returns the first populated element of array
```

You can compare two sparse arrays for equality, e.g.:

```
if (array = array2) {
   // do something
}
```

To access the end of an array, it's convenient to use negative indices; e.g. **array[-1]** returns the last element, **array[-2]** returns the penultimate element, and so on. It's not possible to access elements before the start of the array, so if you do e.g. **array[-100]** on a six element array, you will get **array[0]** returned.

Some things to remember when using sparse arrays:

- Sparse arrays use a zero-based index.
- Elements that have not been initialised are null, and do not cause an error when referenced.
- Assigning to an index beyond the current length increases the **`Length`** to one greater than the index assigned to.
- If an array contains references to objects, whether the arrays are equal or not depends on the implementation of equality for those objects.

User properties

Most ManuScript objects, including objects created by Sibelius, can have user properties attached to them, allowing for convenient storage of extra data, encapsulation of several items of data within a single object, and returning more than one value from a method, among other things.

To create a new user property, use the following syntax:

object`._property:`*property_name* `=` *value*`;`

where *object* is the name of the object, *property_name* is the desired user property name, and *value* is the value to be assigned to the new user property. User properties are read/write and can be accessed as *object*`.`*property_name*.

To get a sparse array containing the names of all the user properties belonging to an object, you can do e.g.:

```
names = object._propertyNames;
```

Here is an example of creating a user property:

```
nr = bar.NoteRest;
nr._property:original = true;
if (nr.original = true) {
   // do something
}
```

Some things to remember when using user properties:

- If you attempt to get or set a user property that has not yet been created, your plug-in will exit with a run-time error.
- To check whether or not a user property has been created without causing a run-time error, use the notation *object*`._property:`*property_name*, which will be null if no matching user property has been created yet.
- User properties cannot be created or accessed for normal data types (e.g. strings, integers, etc.), the global **`Sibelius`** object, old-style ManuScript arrays created by **`CreateArray()`**, old-style hashes created by **`CreateHash()`**, and null.
- User properties that conflict with an existing property name cannot be accessed as **`object.`***property_name* (though they can be accessed using the **`._property:`** notation).
- User properties belong to a particular ManuScript object and disappear when that object's lifetime ends. To stop an object dying, you can (for example) store it in a sparse array, but be aware that its contents may become invalid if (say) the underlying score changes.

Dictionary

`Dictionary` is a programmer extensible object, simply allowing the use of user properties as above with convenient construction. It also has methods allowing the use of aribtrarily named user properties, and can also have methods in plug-ins attached to it allowing the creation of encapsulated user objects (i.e. objects with variables and methods attached to them).

To create a dictionary, use the built-in function **`CreateDictionary(`***name1*`,` *value1*`,` *name2*`,` *value2*`,` ... *nameN*`,` *valueN*`)`. This creates a dictionary containing user properties called *name1*, *name2*, *nameN* with values *value1*, *value2*, *valueN* respectively.

A dictionary can contain named data items (like a **`struct`** in languages like C++), or data that is indexed by string, so that you can use strings to look items up within it.

The values in a dictionary can be accessed using square bracket notation, so you can use a dictionary like a hash table, e.g.:

```
test = CreateDictionary("fruit",apple,"vegetable",potato);
test["fruit"] = banana;
test["meat"] = lamb;
```

You can even put other objects, e.g. sparse arrays, inside dictionaries, e.g.

```
test2 = CreateDictionary("fruit", CreateSparseArray(apple,banana,orange));
```

You can access the user properties within a dictionary using the **`._property:`** notation, e.g.:

```
return test2._property:fruit;
```

which would return the array specified above. Even more direct, you can access user properties in a dictionary as if they were variables or methods, like this:

```
test2.fruit;
```

which would also return the array specified above. You can also return more than one value from any ManuScript method using a dictionary, e.g.:

```
getChord()
value = CreateDictionary("a", aNote, "b", anotherNote);
return value;

//... in another method somewhere
chord = getChord();
trace(chord.a);
trace(chord.b);
```

which returns two values, **`a`** and **`b`**, which you can access via e.g. **`chord.a`** and **`chord.b`**.

You can compare two dictionaries for equality, e.g.:

```
if (test2 = test3) {
   // do something
}
```

Whether or not dictionaries containing objects evaluate as equal depends on the implementation of equality for those objects.

If you're comfortable with programming in general, you may find it useful to be able to add methods to dictionaries, particularly if you are writing code designed to act as a library for other methods or plug-ins to call. Writing code in this way provides a degree of encapsulation and can make it easy for client code to use your library.

To add a method to a dictionary, call the dictionary's **`SetMethod()`** method, e.g.:

```
pluginmethod "(obj,x,y) {
  // a method that does something to obj
}"
test4 = CreateDictionary();
test4.SetMethod("doSomething",Self,"pluginmethod");
test4.doSomething(3,4); // call pluginmethod within the current plug-in, passing in
                        // test4 (obj in the method above) and 3 (x in the method
                        // above) and 4 (y in the method above)
```

In the example above, **`doSomething`** is the name of the method belonging to the dictionary, **`Self`** tells the plug-in that the method is defined in the same plug-in, and **`pluginmethod`** is the name of a method elsewhere in the plug-in (shown at the top of the example).

To return a sparse array containing the names of the methods belonging to a dictionary, use the dictionary's **`GetMethodNames()`** method. You can also check the existence of a particular method using the dictionary's **`MethodExists()`** method. Use the dictionary's **`CallMethod()`** method to call a specific method, where the name of the method is the first parameter, and any parameters to be passed to the specified method follow. For example:

```
array = test4.GetMethodNames(); // create sparse array containing method names
first_method_name = array[0]; // sets first_method_name to name of first method
methodfound = test4.MethodExists("doSomething"); // returns True in this case;
test4.CallMethod("doSomething",5,6);
```

Everything you put into a dictionary is a user property, so all of the methods outlined in **User properties** above can be used on data in dictionaries too.

Using user properties as global variables

You can store **`SparseArray`** and **`Dictionary`** objects, and indeed any other object, as user properties of the Plugin object itself. In the example below, **`Self`** is the object that corresponds to the running plug-in, and a user property **`globalData`** is assigned to the plug-in, containing a Dictionary:

```
Self._property:globalData = CreateDictionary(1,2,3,4);
// globalData and Self.globalData can be used interchangeably
trace(globalData);
trace(Self.globalData);
```

User properties assigned to the plug-in are persistent between invocations. Take care to ensure that these user properties are created before you attempt to use them, otherwise your plug-in will abort with a run-time error. Using the **`_property:`***property_name* syntax never causes run-time errors, but direct references to *property_name* force a runtime error if *property_name* hasn't been created yet.

The example below shows how to test the existence of a specific user property, **`globalCounter`**, initialise it to **`0`** if it is not found, then increment it by 1 every time the plug-in runs:

```
// Test the persistence of user properties
if (Self._property:globalCounter = null) {
   Self._property:globalCounter = 0;
}
globalCounter = globalCounter + 1;
// this number increases by one every time the plugin is run
trace(globalCounter);
trace(Self.globalCounter);
```

If you store a reference to a musical object in a user property that is assigned to the plug-in, there is an increased danger of that reference becoming invalid due to the score being closed or edited, etc. Use the **`IsValid()`** method to validate such data before using it.

User properties of plug-ins will be inaccessible (except by using the **`_property:`***property_name* syntax) if there is an existing global variable of the same name.

Watch out for recursive cycles!

Be careful not to create recursive cycles using arrays, user properties and dictionaries. When you use, say, an array in a dictionary, you are not creating a copy of the array or its values, but a reference to the original array: dictionaries and arrays are objects, not values. As a result, you could write something where an array contains a dictionary that itself refers to the original array: this will lead to Sibelius crashing. So be careful!

Dialog editor

Dialog editor is only available in Sibelius for Windows.

For more complicated plug-ins than the ones we've been looking at so far, it can be useful to prompt the user for various settings and options. This may be achieved by using ManuScript's simple built-in dialog editor (which unfortunately is only available in the Windows version of Sibelius). Dialogs can be created in the same way as methods and data variables in the plug-in editor – just select Dialogs, and click Add.

To show a dialog from a ManuScript method, we use the built-in call

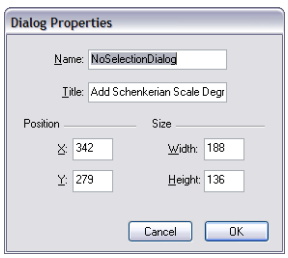
```
Sibelius.ShowDialog(dialogName, Self);
```

where **`dialogName`** is the name of the dialog we wish to show, and **`Self`** is a "special" variable referring to this plug-in (telling Sibelius who the dialog belongs to). Control will only be returned to the method once the dialog has been closed by the user.

Of course, this is of little use unless we can edit the appearance of the dialog. To see how this editor works, edit the supplied Add String Fingering plug-in, select the dialog called **`window`**, and click Edit. The plug-in's dialog will then appear, along with a long thin "palette" of available controls, as follows:

Radio button
Checkbox
Button
Static text
Editable text
Combo box
List box
Group box

The Add String Fingering dialog doesn't use all of the available types of controls, but it does demonstrate four of these types, namely checkboxes, static text, combo boxes and buttons. Each control in the dialog can be selected for editing simply by clicking on it; small black "handles" will appear, which allow the control to be resized. Controls can be moved around the dialog simply by clicking and dragging. To create new controls, drag one of the icons from the control palette over to the dialog itself, and a new control of that type will be created.



The most useful feature of the dialog editor is the Properties window that can be accessed by right-clicking and choosing Properties from the pop-up menu, as shown on the right. With no controls selected, this will allow you to set various options about the dialog itself, such as height, width and title. With a control selected, the properties window varies depending on the type of the control, but most of the options are common to all controls, and these are as follows:

- Text: the text appearing in the control
- Position (X, Y): where the control appears in the dialog, in coordinates relative to the top left-hand corner
- Size (width, height): the size of the control
- Variable storing control's value: the ManuScript Data variable that will correspond to the value of this control when the plug-in is run
- Method called when clicked: the ManuScript method that should be called whenever the user clicks on this control (leave blank if you don't need to know about users clicking on the control)
- Click closes dialog: select this option if you want the dialog to be closed whenever the user clicks on this control. The additional options Returning True / False specify the value that the **`Sibelius.ShowDialog`** method should return when the window is closed in this way.

- Give this control focus: select this option if the "input focus" should be given to this control when the dialog is opened, i.e. if this should be the control to which the user's keyboard applies when the dialog is opened. Mainly useful for editable text controls.

Combo-boxes and list-boxes have an additional property; you can set a variable from which the control's list of values should be taken. Like the value storing the control's current value, this should be a global Data variable. However, in this instance they have a rather special format, to specify a *list* of strings rather than simply a single string. Look at the variable **`_ComboItems`** in Add String Fingering for an example - it looks like this:

```
_ComboItems
{
    "1"
    "2"
    "3"
    "4"
    "1 and 3"
    "2 and 4"
}
```

Radio buttons also have an additional property (in Sibelius 2.1 and later) that allows one to specify *groups* of radio buttons in plug-in dialogs. When the user clicks on a radio button in a group, only the other radio buttons belonging to that groups are deselected; any others in the dialog are left as they are. This is extremely useful for more complicated dialogs. For instance, the Add Chord Symbols plug-in uses this feature.

To specify a radio group, pick one control from each group that represents the first button of the group, and for these controls ensure that the checkbox Start a new radio group is selected in the control's Properties dialog. Then set the *creation order* of the controls by clicking Set Creation Order in the menu obtained by right-clicking over the plug-in dialog. Enabling this will show a number over each control, corresponding to the order in which they are created when the dialog is opened. This can be changed by clicking over each control in the order you want them to be created. A radio button group is defined as being all the radio buttons created between two buttons that have the Start a new radio group flag set (or between one of these buttons and the end of the dialog). So to make the radio groups work properly, ensure that each group is created in order, with the button at the start of the group created first, and then all the rest of the radios in that group. To finish, click the Set Creation Order menu item again to deactivate this mode.

Other properties are available for static text controls (so that one can specify whether their text is left- or right-aligned) and button controls (allowing one to specify whether or not the control should be the default button). For examples of all of these, look through some of the supplied plug-ins, some of which contain quite complex dialogs.

Other things to look out for

The Parallel 5ths and 8ves plug-in illustrates having several methods in a plug-in, which we haven't needed so far. The Proofread plug-in illustrates that one plug-in can call another – it doesn't do much itself except call the CheckPizzicato, CheckSuspectClefs, CheckRepeats and CheckHarpPedaling plug-ins. Thus you can build up meta-plug-ins that use libraries of others. Cool!

(You object-oriented programmers should be informed that this works because, of course, each plug-in is an object with the same powers as the objects in a score, so each one can use the methods and variables of the others.)

Deleting multiple objects from a bar

If you wish to delete multiple objects from a bar, you should first build up a list of items to delete, then iterate over the list deleting each object in turn. It is not sufficient to simply delete the objects from the bar as you iterate over them, as this may cause the iterator to go out of sync. Therefore, code to delete all tuplets from a bar should look something like this:

```
counter = 0;
for each Tuplet tup in bar {
  name = "tuplet" & counter;
  @name = tup;
  counter = counter + 1;
}
```

```
// Delete objects in reverse order
while(counter > 0) {
   counter = counter - 1;
   name = "tuplet" & counter;
   tup = @name;
   tup.Delete();
}
```

Debugging your plug-ins

When developing any computer program, it's all too easy to introduce minor (and not so minor!) mistakes, or bugs. ManuScript performs its own internal error checking at all times, and you'll find that if you try to access a non-existent method or variable on an object, or make a syntax error, or attempt to add or remove bars or items from bars while iterating over them, the plug-in will throw an error and open the plug-in editor window at the line that generated the error.

As ManuScript is a simple, lightweight system, there is no special purpose debugger, but there are a handful of tools provided to help you debug your plug-ins.

Undo

One good technique for finding problems in your plug-ins is to set Sibelius's undo buffer to a very small size, or to disable it altogether (by dragging the slider on the Other page of File ▸ Preferences to its leftmost position). In the unlikely event that ManuScript does not throw an error when you perform an illegal operation (e.g. adding or deleting an object while iterating over a bar), reducing the undo buffer to its smallest possible size will expose the problem right away – though be warned, the result of such a problem may well be that Sibelius will crash.

Plug-in Trace Window

The trace window can be shown by choosing Plug-ins ▸ Plug-in Trace Window. A special ManuScript command, **`trace(`***string*`)`**, will print the specified string in the trace window. This is useful to keep an eye on what your plug-in is doing at particular points. These commands can then be removed when you've finished debugging. Another useful feature of the trace window is function call tracing. When this is turned on, the log will show which functions are being called by plug-ins.

One potential pitfall with the **`trace(`***string*`)`** approach to debugging is that the built-in hash table and array objects discussed earlier aren't strings, and so can't be output to the trace window. To avoid this problem, both of these objects have a corresponding method called **`WriteToString()`**, which returns a string representing the whole structure of the array or hash at that point. So we could trace the current value of an array variable as follows:

```
trace("array variable = " & array.WriteToString());
```

Checking the validity of objects

One of the common problems that you might encounter when writing complex plug-ins is that the object you are trying to operate on is no longer valid (e.g. it has already been deleted). You can enable error checking – either for all objects, or for individual objects – that will cause your plug-in to throw an error if an object is no longer valid.

To enable error checking, use the ManuScript command **`ValidationChecking(`***enable[`**`,`** object1[`**`,`** object2]...]*`)`**, and set the Boolean parameter *enable* to **`true`**. If *enable* is the only parameter, validation checking is enabled for all types of objects, and all plug-ins. If you supply one or more *object* parameters (e.g. **`Tuplet`**, **`Score`**, **`BarObject`**, etc.), only those objects will be checked, and only in the currently running plug-in. You should ensure **`ValidationChecking`** is set to **`false`** before you give your plug-ins to anybody else to use.

You can also use the special method **`IsValid()`** to determine whether an object is valid: it will return **`false`** if the object in question no longer exists. **`GetValidationError(`***object*`)`** returns an empty string if there is no error, or returns a string if an error has occurred, so you can do e.g. **`trace(GetValidationError(score));`** to trace any validation error returned by a Score object to the trace window.

Stopping the plug-in

If you want to force your plug-in to stop on a particular error condition, use the method **`StopPlugin(`***[message]*`)`**, which will stop your plug-in, display the optional *message* in an alert box, and open the plug-in editor at the line of code reached.

Storing and retrieving preferences

In Sibelius 4 or later, you can use Preferences.plg, contributed by Hans-Christoph Wirth, to store and retrieve user-set preferences for your plug-ins.

How does it work?

Preferences.plg stores its data in a text file in the user's application data folder. Strings are accessed as associated pairs of a *key* (the name of the string) and a *value* (the contents of the string). The value can also be an array of strings, if required.

Initializing the database

`errorcode = Open(`*pluginname*`,`***featureset*`);`**

Open the library and lock for exclusive access by the calling plug-in. The calling plug-in is identified with the string *pluginname*. It is recommended that this string equals the unique Sibelius menu name of the calling plug-in.

Parameter *featureset* is the version of the feature set requested by the calling plug-in. The version of the feature set is currently 020000. Each library release shows in its initial dialog a list of supported feature sets. The call to **`Open()`** will fail and show a user message if you request an unsupported feature set. If you should want to prevent this user information (and probably setup your own information dialog), use **`CheckFeatureSet()`** first.

After **`Open()`** the scope is undefined, such that you can access only global variables until the first call to **`SwitchScope()`**.

Return value: **`Open()`** returns zero or a positive value on success. A negative result indicates that there was a fatal error and the database has not been opened.

- **`-2`** other error
- **`-1`** library does not support requested feature set
- **`0`** no common preferences database found
- **`1`** no preferences found for current plug-in
- **`2`** preferences for current plug-in loaded

In case of errors (e.g. if the database file is unreadable), **`Open()`** offers the user an option to recover from the error condition. Only if this fails too will an error code be returned to the calling plug-in.

`errorcode = CheckFeatureSet(`*featureset*`);`**

Check silently if the library supports the requested feature set.

Return value: **`CheckFeatureSet()`** returns zero or a positive value on success. A negative value indicates that the requested feature set is not supported by this version.

`errorcode = Close();`

Release the exclusive access lock to the library. If there were any changes since the last call to **`Open()`** or **`Write()`**, dump the data changes back to disk (probably creating a new score, if there was none present).

Return value: **`Close()`** returns zero or a positive value on success. A negative result indicates that there was a fatal error and the database has not been written.

`errorcode = CloseWithoutWrite();`

Release the exclusive access lock to the library, discarding any changes performed since last call to **`Open()`** or **`Write()`**.

Return value: **`CloseWithoutWrite()`** returns zero or a positive value on success. A negative result indicates that there was a fatal error, namely that the database was not open at the moment.

`errorcode = Write(`*dirty*`);`**

Force writing the data back to disk immediately. Keep library locked and open. If *dirty* equals 0, the write only takes place if the data has been changed. If *dirty* is positive, the common preferences score is unconditionally forced to be rewritten from scratch.

Return value: **`Write()`** returns zero or a positive value on success. A negative result indicates that there was a fatal error and the database has not been written.

Accessing data

`index = SetKey(`*keyname*`, `*value*`);`**

Store a string value *value* under the name *keyname* in the database, overwriting any previously stored keys or arrays of the same *keyname*.

If *keyname* has been declared as a local key, the key is stored within the current scope and does not affect similar keys in other scopes. It is an error to call **`SetKey()`** for local keys if the scope is undefined.

Return value: **`SetKey()`** returns zero or a positive value on success, and a negative value upon error.

`errorcode = SetArray(`*keyname*`, `*array*`, `*size*`);`**

Store a array *array* of strings under the name *keyname* in the database, overwriting any previously stored keys or arrays of the same *keyname*. *size* specifies the number of elements in the array. A *size* of **`-1`** is replaced with the natural size of the array, i.e., **`array.NumChildren`**.

If *keyname* has been declared as a local key, the array is stored within the current scope and does not affect similar keys in other scopes. It is an error to call **`SetArray()`** for local keys if the scope is undefined.

Return value: **`SetArray()`** returns zero or a positive value on success, and a negative value upon error.

`value = GetKey(`*keyname*`);`**

Retrieve the value of key *keyname* from the database. It is an error to call **`GetKey()`** on an identifier which had been stored the last time using **`SetArray()`**. For local keys, the value is retrieved from the current scope which must not be undefined.

Return value: The value of the key or **`Preferences.VOID`** if no key of that name found.

`size = GetArray(`*keyname*`, `*myarray*`);`**

Retrieve the string array stored under name *keyname* from the database. It is an error to call **`GetArray()`** on an identifier which has been stored the last time by **`SetKey()`**. For local arrays, the value is retrieved from the current scope which must not be undefined.

You must ensure before the call that *myarray* is of ManuScript's array type (i.e., created with **`CreateArray()`**).

Return value: *size* equals the number of retrieved elements or **`-1`** if the array was not found. Note that *size* might be smaller than **`myarray.NumChildren`**, because there is currently no way to reduce the size of an already defined array.

`size = GetListOfIds(`*myarray*`);`**

Fill the array *myarray* with a list of all known Ids in the current score (or in the global scope, if undefined). Before you call this method, ensure that *myarray* is of ManuScript's array type (i.e. created with **`CreateArray()`**).

Return value: returns the size of the list, which might be smaller than the natural size of the array, **`myarray.Numchildren`**.

`index = UnsetId(`*keyname*`);`**

Erase the contents stored with an identifier (there is no distinction between keys and arrays here). If the key is declared as local, it is erased only from the local scope which must not be undefined.

Return value: The return value is zero or positive if the key has been unset. A negative return value means that a key of that name has not been found (which is not an error condition).

`RemoveId(`*keyname*`);`**

Erase all contents stored in the database with an identifier (there is no distinction between keys and arrays here). If the key is declared as local, it is erased from all local scopes.

Return value: The return value is always zero.

`RemoveAllIds();`

Erase everything related to the current plug-in.

Return value: the return value is always zero.

Commands for local variables

`errorcode = DeclareIdAsLocal(`*keyname***`);`**

Declare an identifier as a local key. Subsequent calls to **`Set`**... and **`Get`**... operations will be performed in the scope which is set at that time. The local state is stored in the database and can be undone by a call to **`DeclareIdAsGlobal`** or **`RemoveId`**.

Return value: Non-negative on success, negative on error.

`size = GetListOfLocalIds(`*myarray***`);`**

Fill the array *myarray* with a list of all Ids declared as local. Before you call this method, ensure that *myarray* is of ManuScript's array type (i.e. created with **`CreateArray()`**).

Return value: Returns the size of the list, which might be smaller than the natural size of the array, **`myarray.NumChildren`**.

`errorcode = SwitchScope(`*scopename***`);`**

Select scope *scopename*. If scope *scopename* has never been selected before, it is newly created and initialized with no local variables. Subsequent **`Set`**... and **`Get`**... operations for keys declared as local will be performed in scope *scopename*, while access to global keys is still possible.

The call **`SwitchScope("")`** selects the undefined scope which does not allow access of any local variables.

Return value: Non-negative on success, negative on error.

`errorcode = RemoveScope();`

Erase all local keys and arrays from the current scope and delete the current scope from the list of known scopes. It is an error to call **`RemoveScope()`** if the current scope is undefined. After the call, the database remains in the undefined scope.

`errorcode = RemoveAllScopes();`

Erase all local keys and arrays from all scopes and delete all scopes from the list of known scopes. After the call, the database remains in the undefined scope. Note that this call does retain the information which Ids are local (see **`DeclareIdAsLocal()`**).

Return value: Non-negative on success.

`string = GetCurrentScope();`

Retrieve the name of the currently active scope, or the empty string if the database is in undefined scope.

Return value: Returns a string.

`size = GetListOfScopes(`*myarray***`);`**

Fill the array *myarray* with a list of all known scope names. You must ensure before the call that *myarray* is of ManuScript's array type (i.e., created with **`CreateArray()`**).

Return value: Returns the size of the list, which might be smaller than the natural size of the array, **`myarray.NumChildren`**.

Miscellaneous

`Trace(`*tracelevel***`);`**

Select level of tracing for the library. Useful levels are: 0 for no trace, 10 for sparse trace, 20 for medium trace, 30 for full trace. This command can also be run when the library is not open, to specify the tracing level for the **`Open()`** call itself.

`TraceData();`

Writes a full dump of the data stored currently in **`ThisData`** array to the trace window. This is the full data belonging to the current plug-in. **`TraceData()`** always traces the data, regardless of the current trace level selected.

`filename = GetFilename();`

Return the full filename of the preferences database (including path).

`Editor();`

Invoke the interactive plug-in editor. This method must not be called while the database is open. Direct calls to **`Editor()`** from plug-ins are deprecated, since the end-user of your plug-in will probably not expect to be able to edit (and destroy) the saved preferences of *all* plug-ins at this stage.

Basic example

Suppose you have a plug-in called *myplugin* and would like to save some dialog settings in a preferences file such that these settings are persistent over several Sibelius sessions and computer reboots. Your dialog may contain two checkboxes and a list box. Let **`DialogDontAskAgain`** and **`DialogSpeedMode`** be the global variables holding the status of the checkboxes, respectively, and let **`DialogJobList`** hold the contents of the list box item.

The work with the database can be reduced to four steps:

1. Open the database and retrieve initial data. At begin of your plug-in, e.g. right at top of your **`Run()`** method, you have to add some code to initialize the database. You probably also want to initialize your global keys based on the information currently stored in the database. See below for a detailed example. (Depending on your program, you might have to define **`prefOpen`** as a global variable in order to prevent trying to access an unopened database in future.)

```
// At first define hard coded plug-in defaults, in case that the plug-in
// is called for the first time. If anything else fails, these defaults
// will be in effect.

DialogDontAskAgain = 0;
DialogSpeedMode = 0;
DialogJobList = CreateArray();
DialogJobList[0] = "first job";
DialogJobList[1] = "second job";

// Attempt to open the database

prefOpen = Preferences.Open( "myplugin", "020000" );
if( prefOpen >= 0 ) {

   // Database successfully opened. So we can try to load the
   // information stored last time.
   // It's a good idea to work with a private version scheme, in order
   // to avoid problems in the future when the plug-in is developed
   // further, but the database still contains the old keys. In our
   // example, we propose that the above mentioned keys are present
   // if "version" key is present and has a value of "1".

   version = Preferences.GetKey( "Version" );

   switch( version ) {

      case( "1" ) {

      // Now overwrite the above set defaults with the information stored
      // in the database.

      DialogDontAskAgain = Preferences.Getkey( "DontAskAgain" );
      DialogSpeedMode = Preferences.Getkey( "SpeedMode" );
      Preferences.GetArray( "JobList", DialogJobList );

      }

   default {

      // handle other versions/unset version gracefully here ...

      }

   }

}
```

2. Work with the data. After the initialization step, you can and should work with global variables **`DialogDontAskAgain`**, **`DialogSpeedMode`**, and **`DialogJobList`** as you are used to: read from them to base control flow decisions on their setting, write to them (mostly from within your own dialogs) to set new user preferences.

3. Write data back to the database. To make any changes persistent, you must tell the database the new values to be written to the hard disk. See below for a detailed example. According to taste, you can execute these lines each time the settings are changed, or only once, at the end of your plug-in.

```
if( prefOpen >= 0 ) {
   Preferences.SetKey( "Version", "1" );
   Preferences.SetKey( "DontAskAgain", DialogDontAskAgain );
   Preferences.SetKey( "SpeedMode", DialogSpeedMode );
   Preferences.SetArray( "JobList", DialogJobList, -1 );
}
```

4. Close the database. In any case, you must release the lock to the library on exit of your plug-in. This writes data actually back to disk, and enables other plug-ins to access the shared database later. To do this, use:

```
Preferences.Close();
```

Reference

Syntax

Here is an informal run-down of the syntax of ManuScript.

A method consists of a list of statements of the following kinds:

Block

{ *statements* **}**

e.g.

```
{
   a = 4;
}
```

While

while { *expression* **}** *block*

e.g.

```
while (i < 3) {
   Sibelius.MessageBox(i);
   i = i + 1;
}
```

Switch

switch (*test-expression***) {**
case (*case-expression-1***)** *block*
[**case (***case-expression-2***)** *block]*
…
[**default** *block]*

The switch statement consists of a "test" expression, multiple case statements and an optional default statement. If the value of test-expression matches one of the case-expressions, then the statement block following the matching case statement will be executed. If none of the case statements match, then the statement block following the default statement will be executed. For example:

```
switch (note.Accidental) {
   case (DoubleSharp) {
      Sibelius.MessageBox("Double sharp");
   }
   case (DoubleFlat) {
      Sibelius.MessageBox("Double flat");
   }
   default {
      Sibelius.MessageBox("No double");
   }
}
```

if else

if (*expression***)** *block [* **else** *block]*

e.g.

```
if (found) {
   Application.ShowFindResults(found);
} else {
   Application.NotFindResults();
}
```

for each

`for each` *variable* **`in`** *expression*
block

This sets variable to each of the sub-objects within the object given by the expression.

Normally there is only one type of sub-object that the object can contain. For instance, a NoteRest (such as a chord) can only contain Note objects. However, if more than one type of sub-object is possible you can specify the type:

`for each` *Type variable* **`in`** *expression*
block

e.g.

```
for each NoteRest n in thisstaff {
   n.AddNote(60);          // add middle C
}
```

for

`for` *variable* **`=`** *value* **`to`** *value [* **`step`** *value]*
block

The variable is stepped from the first value up to or down to the end value by the step value. It stops one step before the final value.

So, for example:

```
for x=1 to note.NoteCount {
   ...
}
```

works correctly.

assignment

variable **`=`** *expression* **`;`**

e.g.

value **`=`** *value* **`+ 1;`**

or

variable **`.`** *variable* **`=`** *expression* **`;`**

e.g.

```
Question.CurrentAnswer=True;
```

method call

variable **`.`** *identifier* **`(`** *comma-separated expressions* **`);`**

e.g.

```
thisbar.AddText(0,"Mozart","text.system.composer");
```

self method call

identifier **`(`** *comma-separated expressions* **`);`**

Calls a method in this plug-in, e.g.

```
CheckIntervals();
```

return

`return` *expression* **`;`**

Returns a value from a plug-in method, given by the expression. If a method doesn't contain a **`return`** statement, then a "null" value is returned (either the number zero, an empty string, or the **`null`** object described below).

Expressions

Here are the operators, literals and other beasts you're allowed in expressions.

Self

This is a keyword referring to the plug-in owning the method. You can pass yourself to other methods, e.g.

```
other.Introduce(Self);
```

null

This is a literal object meaning "nothing."

Identifier

This is the name of a variable or method (letters, digits or underscore, not starting with a digit) you can precede the identifier with **@** to provide indirection; the identifier is then taken to be a string variable whose value is used as the name of a variable or method.

member variable

*variable***.***variable*

This accesses a variable in another object.

integer

e.g. **1**, **100**, **-1**

floating point number

e.g. **1.5**, **3.15**, **-1.8**

string

Text in double quotes, e.g. **"some text"**. For strings that are rendered by Sibelius as part of the score, i.e. the contents of some text object, there is a small but useful formatting language allowing one to specify how the text should appear. These "styled strings" contain commands to control the text style. All commands start and end with a backslash (****) The full list of available styling commands is as follows:

\n	New line
\B	Bold on
\b	Bold off
\I	Italic on
\i	Italic off
\U	Underline on
\u	Underline off
\fArial Black	Font change to Arial Black (for example)
\f_	Font change to text style's default font
\s123	Size change to 123 (units are 1/32nds of a space, not points)
\$*keyword*****	Substitutes a string from the Score Info dialog (see below)

A consequence of this syntax is that backslashes themselves are represented by ****, to avoid conflicting with the above commands.

The substitution command **\$***keyword***** supports the following keywords: **Title**, **Composer**, **Arranger**, **Lyricist**, **MoreInfo**, **Artist**, **Copyright**, **Publisher** and **PartName**. Each of these correspond to a field in the File ▸ Score Info dialog.

not

not *expression*

Logically negates an expression, e.g.

```
not (x=0)
```

and

expression **and** *expression*

Logical and, e.g.

```
FoxFound and BadgerFound
```

or	*expression* **or** *expression* Logical or, e.g. `FoxFound or BadgerFound`
equality	*expression* **=** *expression* Equality test, e.g. `Name="Clock"`
subtract	*expression* **–** *expression* Subtraction, e.g. `12-1`
add	*expression* **+** *expression* Addition, e.g. `12+1`
minus	**–***expression* Inversion, e.g. `-1`

concatenation	*expression* **&** *expression*
	Add two strings, e.g.
	`Name = "Fred" & "Bloggs"; // 'Fred Bloggs'`
	You can't use **+** as this would attempt to add two numbers, and sometimes succeed (!). For instance:
	`x = "2" + "2"; // same as x = 4`

subexpression	**(***expression***)** For grouping expressions and enforcing precedence, e.g. `(4+1)*5`
method call	**variable.identifier(***comma-separated expressions***);** e.g. `x = monkey.CountBananas();`
self method call	*Identifier***(***comma-separated expressions***);** Calls a method in this plug-in, e.g. `x = CountBananas();`

Operators

Condition operators

You can put any expressions in parentheses after an **`if`** or **`while`** statement, but typically they will contain conditions such as **`=`** and **`<`**. The available conditions are very simple:

`a = b`	equals (for numbers, text or objects)
`a < b`	less than (for numbers)
`a > b`	greater than (for numbers)
`c and d`	both are true
`c or d`	either are true
`not c`	inverts a condition, e.g. **`not(x=4)`**
`<=`	less than or equal to
`>=`	greater than or equal to
`!=`	not equal to

Note that you use **`=`** to compare for equality, not **`==`** as found in C/C++ and Java.

Arithmetic

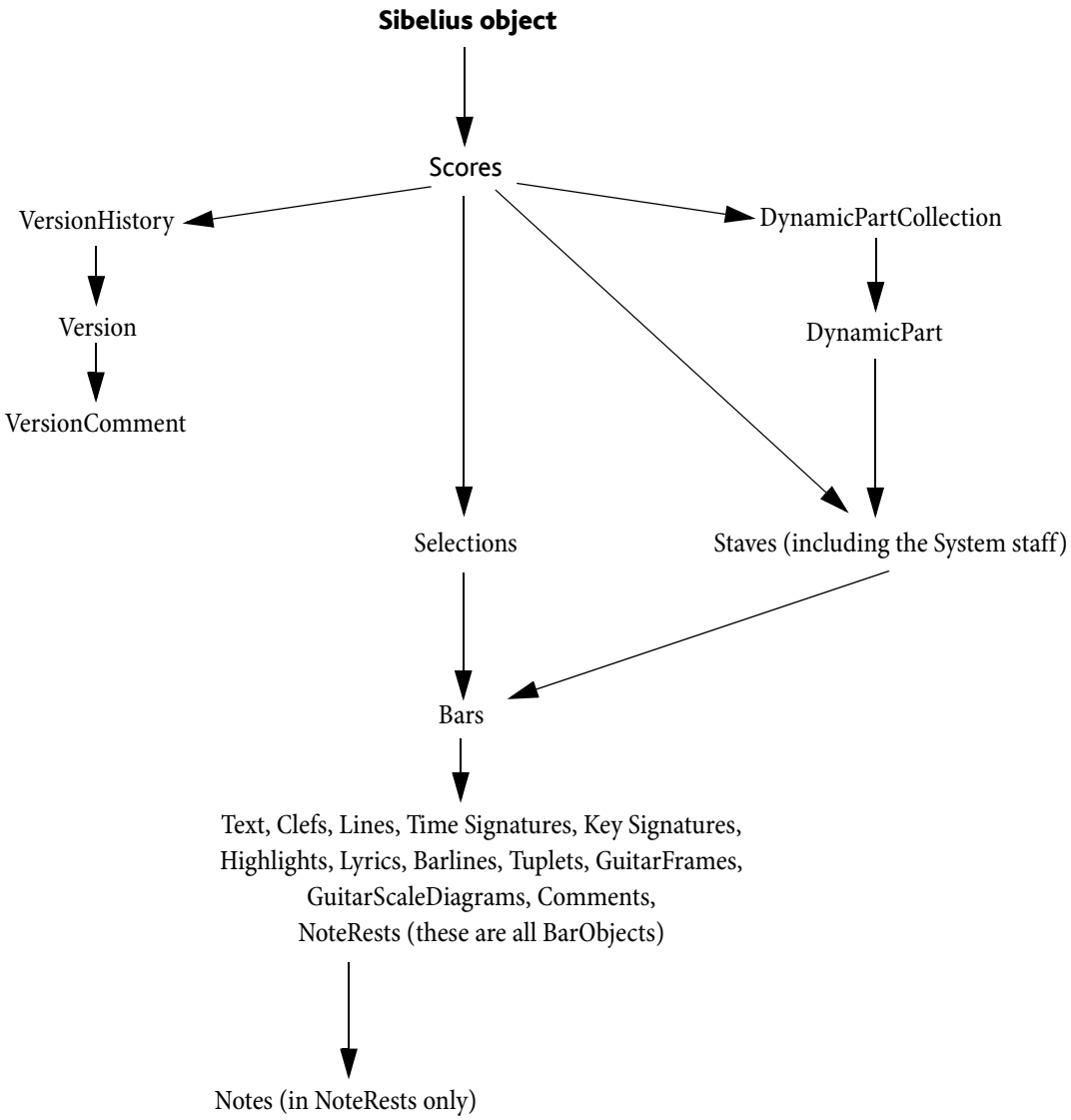
`a + b`	add
`a - b`	subtract
`a * b`	multiply
`a / b`	divide
`a % b`	remainder
`-a`	negate
`(a)`	evaluate first

ManuScript will evaluate expressions from left to right, so that **`2+3*4`** is 20, not 14 as you might expect. To avoid problems with evaluation order, use parentheses to make the order of evaluation explicit. To get the answer 14, you'd have to write **`2+(3*4)`**.

ManuScript also now supports floating point numbers, so whereas in previous versions **`3/2`** would work out as 1, it now evaluates to 1.5. Conversion from floating point numbers to integers is achieved with the **`RoundUp(`***expr***`)`**, **`RoundDown(`***expr***`)`** and **`Round(`***expr***`)`** functions, which can be applied to any expression.

Object Reference

Hierarchy of objects



All objects

Methods

`AddToPluginsMenu("`*menu text***`","`***function name***`")`**

Adds a new menu item to the Plug-ins menu. When the menu item is selected the given function is called. This is normally only used by plug-ins themselves. This method may only be called once per plug-in (i.e. each plug-in may only add one item to the Plug-ins menu); subsequent method calls will be ignored.

`Asc(`*expression***`)`**

Returns the ASCII value of a given character (the expression should be a string of length 1).

`CharAt(`*expression***`,`***position***`)`**

Returns the character from the expression at the given (zero-based) position, e.g. **`CharAt("Potato",3)`** would give "a."

`Chr(`*expression***`)`**

Returns a character (as a string) with the given ASCII value. This can be used for inserting double quotes (**`"`**) into strings with **`Chr(34)`**.

`CreateArray()`

Returns a new array object.

`CreateHash()`

Returns a new hash-table object.

`GetValidationError(`*object***`)`**

Returns the validation error, if any, of the specified *object*. Useful to pass validation errors to the plug-in trace window.

`IsObject(`*expression***`)`**

Returns **`1`** (or **`True`**) if expression evaluates to an object rather than a string or integer.

(Not to be confused with the **`IsPassage`** variable of Selection objects!)

`IsValid(`*object***`)`**

Returns **`1`** (or **`True`**) if the object is valid, returns **`0`** (or **`False`**) if the object no longer exists (i.e. has been deleted).

`JoinStrings(`*expression***`, `***delimiter***`)`**

Joins together (concatenates) an array of strings into a single string, separated by the string delimiter.

`Length(`*expression***`)`**

Gives the number of characters in the value of the expression.

`Round(`*expression***`)`**

Returns the nearest integer to the value of the expression, e.g. **`Round(1.5)`** would be "2" and **`Round(1.3)`** would be "1."

`RoundDown(`*expression***`)`**

Returns the nearest integer less than the value of the expression, e.g. **`RoundDown(1.5)`** would be "1."

`RoundUp(`*expression***`)`**

Returns the nearest integer greater than the value of the expression, e.g. **`RoundUp(1.5)`** would be "2."

`SplitString(`*expression***`,`***[delimiter]*[trimEmpty]***`)`**

Splits a string into an array of strings, using the given delimiter. The delimiter can be a single character or a string containing several characters – for instance **`".,"`** would treat either a comma or full stop as a delimiter. The default delimiter is the space character. If the *trimEmpty* parameter is **`True`** then this will ignore multiple delimiters (which would otherwise produce some empty strings in the array). The default value of *trimEmpty* is **`False`**.

```
s=':a:b:c';
bits=SplitString(s,':', false);
// bits[0] = ''; bits[1] = 'a'; bits[2] = 'b' ...
s='a b c';
bits=SplitString(s,' ', true);
// bits[0] = 'a'; bits[1]='b' ...
```

`StopPlugin(`*[message]***`)`**

Stops the plug-in, and shows the optional *message* in an alert box. Opens the plug-in editor at the line of code reached.

`Substring(`*expression***`,`***start***`,`***[length]***`)`**

This returns a substring of the expression starting from the given start position (zero-based) up to the end of the expression, e.g. **`Substring("Potato",2)`** would give "tato". When used with the optional length parameter, Substring returns a substring of the of the expression starting from the given start position (zero-based) of the given length, e.g. **`Substring("Potato",2,2)`** would give "ta".

`Trace(`*expression***`)`**

Sends a piece of text to be shown in the plug-in trace window, e.g. **`Trace("Here's a trace");`**

`ValidationChecking(`*enable[`**`,`***object1[`**`,`***object2]...]***`)`**

If *enable* is the only parameter, validation checking is enabled for all types of objects, and across all plug-ins. If you supply one or more *object* parameters (e.g. **`Tuplet`**, **`Score`**, **`BarObject`**, etc.), only those objects will be checked, and only in the currently running plug-in. You should ensure **`ValidationChecking`** is set to **`false`** before you give your plug-ins to anybody else to use.

User properties

All objects (except for the **`Sibelius`** object, old-style ManuScript arrays created using **`CreateArray()`**, old-style ManuScript hashes created using **`CreateHash()`**, and **`null`**) can also have user properties assigned to them. See **User properties** on page 21 for more details.

Bar

A Bar contains BarObject objects.

for each *variable* **in** produces the BarObjects in the bar

for each *type variable* in produces the BarObjects of the specified type in the bar

Methods

AddBarNumber(*new bar number[,format[,extra_text[,prepend[,skip this bar]]]]***)**

Adds a bar number change to the start of this bar. *new bar number* should be the desired external bar number. The optional *format* parameter takes one of the three pre-defined constants that define the bar number format; see **Global constants** on page 119. The optional *extra_text* parameter takes a string that will be added after the numeric part of the bar number, unless the optional boolean parameter *prepend* is **True**, in which case the *extra_text* is added before the numeric part of the bar number. If the optional *skip this bar* parameter is **True**, the bar number change is created with the Don't increment bar number option set. Returns the BarNumber object created.

AddChordSymbolFromPitches(*position,pitches[,instrument style]***)**

Adds a chord symbol from the given array of *pitches* at the specified *position*. The optional instrument style parameter operates the same as in the **AddGuitarFrame** method (see above). If the method is unable to create a chord symbol, the method returns null; otherwise it returns the GuitarFrame object created.

AddClef(*pos,concert pitch clef[,transposed pitch clef]***)**

Adds a clef to the staff at the specified position. *concert pitch clef* determines the clef style when Notes ▸ Transposing Score is switched off; the optional *transposed pitch clef* parameter determines the clef style when this is switched on. Clef styles should be an identifier like "clef.treble"; for a complete list of available clef styles, see **Clef styles** on page 122. Alternatively you can give the name of a clef style, e.g. "Treble," but bear in mind that this may not work in non-English versions of Sibelius. Returns the Clef object created.

AddGraphic(*file name,pos[,below staff[,x displacement[,y displacement[,size ratio]]]]***)**

Adds a graphic above or below the bar at a given position. If *below staff* is True, Sibelius will position the graphic below the staff to which it is attached, otherwise it will go above (the default). You may additionally displace the graphic from its default position by specifying *x-* and *y displacements*. These should be expressed in millimeters, the latter defining an offset from the top or bottom line of the staff, as appropriate. By default, the graphic will be created 5mm away from the staff. To adjust the size of the graphic, you may set a floating point number for its *size ratio*. When set to 1.0 (the default), the graphic will be created with a height equal to that of the staff to which it is attached. A value of 0.5 would therefore halve its size, and 2.0 double it. The graphic may be rescaled to a maximum of five times the height of its parent staff. This function returns True if successful, otherwise False.

AddGraphicToBlankPage(*file name,nth page,x offset,y offset[,size ratio]***)**

Adds a graphic to a blank page belonging to the current bar. *nth page* specifies the particular blank page you would like the graphic to, starting from 1. The *x offset* and *y offset* parameters are floating point values relative to the size of the page the graphic is being added to. For example, an *x offset* of 0.0 would position the graphic at the very left of the page; 0.5 in the centre. You may specify the size of the graphic by specifying a value for *size ratio*. This defaults to 1.0, which has the same effect as creating a graphic in Sibelius manually using Create ▸ Graphic. (As with **AddGraphic**, 0.5 would halve its size, and 2.0 double it.) The graphic may be rescaled to a maximum of five times its intial size. This function returns True if successful, otherwise False.

AddGuitarFrame(*position,chord name[,instrument style[,fingerings]***)**

Adds a chord symbol for the given *chord name* to the bar. The optional *instrument style* parameter should refer to an existing instrument type that uses tab, and should be specified by identifier; see **Instrument types** on page 122. If *instrument style* is not specified, Sibelius will create a chord symbol that will optionally display a chord diagram using the default tab tuning associated with the instrument type used by the staff to which the chord symbol will be attached. The *position* is in 1/256th quarters from the start of the bar. The optional *fingerings* parameter gives the fingerings string to display above (or below) the

guitar frame, if supplied. If the method is unable to create a chord symbol, the method returns null; otherwise it returns the GuitarFrame object created.

AddInstrumentChange(*pos***,** *styleID[***,** *add_clef[***,** *show_text[***,** *text_label[***,** *show_warning[***,** *warning_label***,** *[full_instrument_name[***,** *short_instrument_name]]]]]]***)**

Adds an instrument change to the bar at the specified position. *styleID* is the string representing the instrument type to change to (see **Instrument types** on page 122 for a list). The optional boolean parameter *add_clef*, **True** if not specified, determines whether Sibelius will add a clef change at the same position as the instrument change if required (i.e. if the clef of the new instrument is different to that of the existing instrument). *show_text* is an optional boolean parameter, **True** if not specified, determining whether or not the text label attached to the instrument change should be created shown (the default) or hidden. *text_label* is an optional string parameter; if specified, Sibelius will use this string instead of the default string (the new instrument's long name). *show_warning* is an optional boolean parameter, **True** if not specified, determining whether or not Sibelius should create a text object (using the Instrument change staff text style) above the last note preceding the instrument change, announcing the instrument change and giving the player time to pick up the new instrument. *warning_label* is an optional string parameter; if specified, Sibelius will use this string instead of the default string (the word "To" followed by the new instrument's short name). You can also override the names Sibelius will give the instruments on subsequent systems. If a null string is passed to either *full_instrument_name* or *short_instrument_name* (or if the arguments are omited), the instrument names will remain unchanged. Returns the InstrumentChange object created.

AddKeySignatureFromText(*pos***,** *key name***,** *major key[***,** *add double barline[***,** *hidden[***,** *one staff only]]]***)**

Adds a key signature to the bar. The key signature is specified by text name, e.g. "Cb" or "C#". The third parameter is a Boolean flag indicating if the key is major (or minor). Unless the fourth parameter is set to **False**, a double barline will ordinarily be created alongside the key signature change. You may additionally hide the key signature change by setting *hidden* to **True**, and make the change of key appear on the bar's parent staff only with the *one staff only* flag. Returns the key signature object created.

AddKeySignature(*pos***,** *num sharps***,** *major key[***,** *add double barline[***,** *hidden[***,** *one staff only]]]***)**

Adds a key signature to the bar. The key signature is specified by number of sharps (+1 to +7), flats (-1 to –7), no accidentals (0) or atonal (-8). The third parameter is a Boolean flag indicating if the key is major (or minor). Unless the fourth parameter is set to **False**, a double barline will ordinarily be created alongside the key signature change. You may additionally hide the key signature change by setting *hidden* to **True**, and make the change of key appear on the bar's parent staff only with the *one staff only* flag. Returns the key signature object created.

AddLine(*pos***,** *duration***,** *line style[***,** *dx[***,** *dy[***,** *voicenumber[***,** *hidden]]]]***)**

Adds a line to the bar. The line style can be an identifier such as "line.staff.hairpin.crescendo" or a name, e.g. "Crescendo". For a complete list of line style identifiers that can be used in any Sibelius score, see **Line styles** on page 121. Style identifiers are to be preferred to named line styles as they will work across all language versions of Sibelius. Returns the Line object created, which may be one of a number of types depending on the Line style used.

AddLyric(*position***,** *duration***,** *text[***,** *syllable type [***,** *number of notes***,** *voicenum]]]***)**

This method adds a lyric to the bar. The position is in 1/256th quarters from the start of the bar, and the duration is in 1/256th quarter units. The two optional parameters allow you to specify whether the lyric is at the end of a word (value is "1", and is the normal value) or at the start or middle of a word (value is "0"), and how many notes the lyric extends beneath (default value 1). You can also optionally specify the voice in which the lyric should be created; if *voicenum* is 0 or not specified, the lyric is created in all voices. Returns the LyricItem object created.

AddNote(*pos***,** *sounding pitch***,** *duration***,** *[tied [***,** *voice[***,** *diatonic pitch[***,** *string number]]]]***)**

Adds a note to staff, adding to an existing NoteRest if already at this position (in which case the duration is ignored); otherwise creates a new NoteRest. Will add a new bar if necessary at the end of the staff. The position is in 1/256th quarters from the start of the bar. The optional tied parameter should be **True** if you want the note to be tied. Voice 1 is assumed unless the optional voice parameter (with a value of 1, 2, 3 or 4) is specified. You can also set the diatonic pitch, i.e. the number of the "note name" to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on). If a diatonic pitch of zero is given, a suitable diatonic pitch will be calculated from the MIDI pitch. The optional *string number* parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a

default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Notes page of File‣ Preferences). Returns the Note object created (to get the NoteRest containing the note, use **`Note.ParentNoteRest`**).

`AddPageNumber(`*[blank page offset]***`)`**

Creates and returns a page number change at the end of the bar. Note that – due to the nature of adding a page number change – a page break will also be created at the end of the bar. Therefore, the page number change will actually be placed at the start of the *next* bar. The desired properties of the page number change can be set by calling the appropriate methods on the Page Number Change object returned.

The *blank page offset* flag allows you to create page number changes on blank pages. If a bar object is followed by one or more blank pages, each blank page may also have a page number change of its own. If unspecified, the page number change will be created on the next available page (whether it contains music or not) after the bar, otherwise the user may specify a 1-based offset which refers to the *n*th blank page after the bar itself.

`AddRehearsalMark(`*[consecutive[`**`,`***mark[`**`,`***new prefix and suffix[`**`,`***prefix[`**`,`***suffx[`**`,`***override defaults]]]]]***`)`**

Adds a rehearsal mark above the bar. If no parameters have been specified, the rehearsal mark will inherit the properties of the previous rehearsal mark in the score, incrementing accordingly. Optionally, the appearance of the rehearsal mark may be overriden. If *consecutive* is False, Sibelius will not continue the numbering of the new rehearsal marks consecutively, but allow the user to set a new *mark*. A *mark* may be expressed as a number of a string. For example both `5` and `"e"` are both valid and equivalent values. If *new prefix and suffix* is True, the values set for *prefix* and *suffix* will be applied to the new rehearsal mark. The final parameter, *override defaults*, is a Boolean defaulting to False whose purpose it is to mimic the behavior of the option with the same name in the Rehearsal Mark dialog in Sibelius.

`AddSpecialBarline(`*barline type[`**`,`***pos]***`)`**

Adds a special barline to a given position in a bar; see **Global constants** on page 119. If no position has been specified, start repeat barlines will snap to the start of the bar by default. All other special barline types will snap to the end.

`AddSymbol(`*pos`**`,`***symbol index or name***`)`**

Adds a symbol to the bar at the specified position. If the second parameter is a number, this is taken to be an index into the global list of symbols, corresponding to the symbol's position in the Create‣ Symbol dialog in Sibelius (counting left-to-right, top-to-bottom from zero in the top-left hand corner). Some useful symbols have pre-defined constants; see **Global constants** on page 119. There are also constants defined for the start of each group in the Create‣ Symbol dialog, so that to access the 8va symbol, for example, you can use the index **`OctaveSymbols + 2`**. It's better to use indices rather than names, because the names will be different across the various language versions of Sibelius. Returns the Symbol object created.

`AddText(`*pos`**`,`***text`**`,`***style[`**`,`***voicenum]***`)`**

Adds the text at the given position, using the given text style. A staff text style must be used for a normal staff, and a system text style for a system staff. The styles should be an identifier of the form "text.staff.expression"; for a complete list of text styles present in all scores, see **Text styles** on page 121. Alternatively you can give the name of a text style, eg. "Expression", but be aware that this may not work in non-English versions of Sibelius. You can also optionally specify the voice in which the lyric should be created; if *voicenum* is 0 or not specified, the text object is created in all voices. Returns the Text object created.

`AddTextToBlankPage(`*xPos`**`,`***yPos`**`,`***text`**`,`***style`**`,`***pageOffset***`)`**

Adds the *text* at the given position, using the given text *style*. A blank page text style must be used; you cannot add staff text or system text to a blank page. *style* takes a style ID, using the form "text.blankpage.title"; for a complete list of text styles present in all scores, see **Text styles** on page 121. *xPos* and *yPos* are the absolute position on the page. *pageOffset* takes a positive number for a blank page following a special page break (the first blank page is **`1`**), and negative for a blank page preceding the first bar of the score (the blank page immediately before the first bar is **`-1`**, the one before that **`-2`**, and so on). Returns the Text object created.

To add text to a blank page, first create the special page break using the **`Bar.BreakType`** variable, and set the number of blank pages using **`Bar.NumBlankPages`** or **`Bar.NumBlankPagesBefore`**. Then use **`Bar.AddTextToBlankPage`**.

AddTimeSignature(*top*, *bottom*, *allow cautionary*, *rewrite music*[, *use symbol*]**)**

Returns an error string (which will be empty if there was no error) which if not empty should be shown to the user. The first two parameters are the top and bottom of the new time signature. The third tells Sibelius whether to display cautionary time signatures from this time signature. If *rewrite music* is **True** then all the bars after the inserted the time signature will be rewritten. You can also create common time and alla breve time signatures. If you're creating a time signature in 4/4 or 2/2, set *use symbol* to True and Sibelius will replace the numbers of the time signature with their symbolic equivalent.

AddTimeSignatureReturnObject(*top*, *bottom*, *allow cautionary*, *rewrite music*[, *use symbol*]**)**

As above, but returns the time signature object created, or null if unsuccessful.

AddTuplet(*pos*, *voice*, *left*, *right*, *unit*[, *style*[, *bracket*[, *fullDuration*]]]**)**

Adds a tuplet to a bar at a given position. The *left* and *right* parameters specify the ratio of the tuplet, e.g. 3 (left) in the time of 2 (right). The *unit* parameter specifies the note value (in 1/256th quarters) on which the tuplet should be based. For example, if you wish to create an eighth note (quaver) triplet group, you would use the value 128. The optional *style* and *bracket* parameters take one of the pre-defined constants that affect the visual appearance of the created tuplet; see **Global constants** on page 119. If *fullDuration* is true, the bracket of the tuplet will span the entire duration of the tuplet. Returns the Tuplet object created.

N.B.: If **AddTuplet()** has been given illegal parameters, it will not be able to create a valid Tuplet object. Therefore, you should test for inequality of the returned Tuplet object with *null* before attempting to use it.

Bar[*array element***]**

Returns the nth item in the bar (counting from 0) e.g. **Bar[0]**

Clear(*[voice number]***)**

Clears a bar of all its items, leaving only a bar rest. If a particular voice number is specified, only the items in that voice will be removed.

ClearNotesAndModifiers(*[voice number]***)**

Clears a bar of all its notes, rests, tuplets and slurs, replacing them with a single bar rest. If a particular voice number is specified, only the items in that voice will be removed.

Delete()

Deletes and removes an entire bar from a score. This, by definition, will affect all the staves in the score.

DeletePageNumber(*[blank page offset]***)**

Deletes the page number change at the end of the bar, or if there are one or more blank pages after the bar, any page number change that occurs on any of those blank pages. If *blank page offset* is unspecified, the page number change on the first page after the bar will be deleted.

GetInstrumentTypeAt(*pos***)**

Returns an InstrumentType object representing the instrument type used by the bar at the specified rhythmic position.

GetPageNumber(*[blank page offset]***)**

Returns the page number change object at the end of the bar, or if the bar contains no page number change, null. As with **AddPageNumber**, you may get the page number change from any of the blank pages that follow the bar by specifying a valid *blank page offset*.

InsertBarRest(*voice number*[, *rest type*]**)**

Inserts a bar rest into the bar, but only if the bar is void of any NoteRests (or an existing bar rest) using the same *voice number*. The optional *rest type* parameter allows you to specify the type of bar rest or repeat bar to be created, defined by the constants **WholeBarRest** (the default if rest type is not specified), **BreveBarRest**, **OneBarRepeat**, **TwoBarRepeat** and **FourBarRepeat**. Returns True if successful.

NthBarObject(*n***)**

Returns the nth object in the bar, counting from 0.

Respace()

Respaces the notes in this bar.

Variables

BarNumber The bar number of this bar. This is the internal bar number, which always runs consecutively from 1 (read only).

BarObjectCount The number of objects in the bar (read only).

BreakType The break at the end of this bar, given by the constants **MiddleOfSystem**, **EndOfSystem**, **MiddleOfPage**, **EndOfPage**, **NotEndOfSystem**, **EndOfSystemOrPage** or **SpecialPageBreak**. To learn the correspondence between these constants and the menu in the Bars panel of the Properties window, see the discussion in **Global constants** on page 119.

When you set the break at the end of a bar to be **SpecialPageBreak**, Sibelius will add one blank page after the break. You can then adjust the number of pages by setting the value of either **Bar.NumBlankPages** or **Bar.NumBlankPagesBefore**, or tell Sibelius to restart the music on the next left or right page with **Bar.MusicRestartsOnPage**.

ExternalBarNumber This has been deprecated as of Sibelius 5, because it can only return a number, and bar numbers that appear in the score may now include text. Use **ExternalBarNumberString** instead.

Returns the external bar number of this bar, taking into account bar number changes in the score (read only). Note that you cannot pass this bar number to any of the other ManuScript accessors; they all operate with the internal bar number instead.

ExternalBarNumberString The external bar number of this bar as a string, taking into account bar number changes and bar number format changes (read only). Note that you cannot pass this bar number to any of the other ManuScript accessors; they all operate with the internal bar number instead.

InMultirest Returns one of four global constants describing if and/or where the bar falls in a multirest (read only). The constants are NoMultirest, StartsMultirest, EndsMultirest and MidMultirest; see **Global constants** on page 119.

Length The rhythmic length (read only).

MusicRestartsOnPage Tells Sibelius to restart the music on the next left or right page after the break. May be set to only *two* of the global special page break constants: **MusicRestartsOnNextLeftPage** or **MusicRestartsOnNextRightPage** (write only).

NthBarInSystem Returns the position of the bar in the system, relative to the first bar on the system (bar 0) (read only).

NumBlankPages The number of blank pages following the bar containing a special page break.

NumBlankPagesBefore The number of blank pages preceding the bar containing a special page break. This value only has an effect if a special page break exists in bar 1.

OnHiddenStave Returns True if the bar is currently hidden by way of Hide Empty Staves (read only).

OnNthPage Returns the zero-based page number on which the bar occurs in the current part (read only).

OnNthSystem Returns the zero-based system number (relative to its parent page) in which the bar occurs (read only).

ParentStaff The staff containing this bar (read only).

Selected Returns True if the entire bar is currently selected (read only).

SpecialPageBreakType Returns the type of the special page break; see the documentation for the Special page break types in **Global constants** on page 119 (read only).

SplitMultirest When **True**, a multirest intersecting the bar in question will be split (read/write).

`Time`	The time at which the bar starts in the score in milliseconds (read only).

BarObject

BarObjects include Clef, Line, NoteRest & Text objects. All the methods and variables below apply to all specific types of BarObject – they are listed here instead of separately for each type. (To object-oriented programmers, the NoteRest, Clef etc. types are derived from the base class BarObject.)

Methods

`Delete()`

Deletes an item from the score. This will completely remove text objects, clefs, lines etc. from the score; however, when a NoteRest is deleted, it will be converted into a rest of similar duration.

`Deselect()`

Removes the object from the selection list of the parent score. If the selection is currently a passage selection, it is first changed to a multiple selection before the object is deselected. Returns **`True`** if the object is successfully removed from the selection.

`GetIsInVoice(`*voicenum*`)`**

Returns **`True`** if the object is in the *voicenum* specified.

`GetPlayedOnNthPass(`*n*`)`**

Returns **`True`** if the object is set to play back the *n*th time.

`NextItem(`*[voice[, item type]]*`)`**

Returns the next item in the parent bar of the current item, or null if no item exists. If no arguments have been supplied, the very next item in the bar will be returned, regardless of its voice number and item type. You may additionally specify the voice number of the object you're looking for (1 to 4, or 0 for any voice number), and the item's type. Note that an item will only be returned if it exists in the same bar as the source item. By way of example, to find the next crescendo line in voice 2, you would type something along the lines of: **`hairpin = item.NextItem(2, "CrescendoLine");`**

`PreviousItem(`*[voice[, item type]]*`)`**

As above, but searches backwards.

`RemoveVoice(`*voicenum*`)`**

Removes the object from the specified *voicenum*, leaving the object in all remaining voices.

`ResetPosition(`*[horizontal[, vertical]]*`)`**

Performs Layout ▸ Reset Position on the object. If you supply no parameters, this method will reset both the horizontal and vertical position of the object. If either or both of the optional Boolean parameters *horizontal* or *vertical* is set to **`True`**, you can reset the position of the object either horizontally or vertically independently if required.

`ResetDesign()`

Performs Layout ▸ Reset Design on the object.

`Select()`

Appends the object to the selection list of the parent score. A multiple selection consiting of any number of individual objects can be built up by repeatedly calling **`Select`** on each object you wish to add to the list. Note that calling **`Select`** on a BarObject will first clear any existing passage selection.

`SetAllVoices()`

Sets the object to be in all voices. This has no effect on some types of object, e.g. NoteRests.

`SetVoice(`*voicenum[, clear other voices]*`)`**

Sets the object to be in voice *voicenum*, optionally removing the object from all other voices if the Boolean parameter *clear other voices* is **`True`**.

`ShowInAll()`

Shows the object in the full score, and in all relevant parts; equivalent to Edit ▸ Hide or Show ▸ Show In All.

ShowInParts()

Hides the object in the full score, and shows it in all relevant parts; equivalent to Edit ▸ Hide or Show ▸ Show In Parts.

ShowInScore()

Hides the object in all relevant parts, and shows it in the full score; equivalent to Edit ▸ Hide or Show ▸ Show In Score.

SetPlayedOnNthPass(*n, do play***)**

Tells Sibelius whether or not the object should play back the *n*th time.

Variables

CanBeInMultipleVoices	Returns **True** if the object can be in more than one voice (read-only).
Color	The color of this BarObject (read/write). The color value is in 24-bit RGB format, with bits 0–7 representing blue, bits 8–15 green, bits 16–23 red and bits 24–31 ignored. Since ManuScript has no bitwise arithmetic, these values can be a little hard to manipulate; you may find the individual accessors for the red, green and blue components to be more useful (see below).
ColorRed	The red component of the color of this BarObject, in the range 0–255 (read/write).
ColorGreen	The green component of the color of this BarObject, in the range 0–255 (read/write).
ColorBlue	The blue component of the color of this BarObject, in the range 0–255 (read/write).
CueSize	**True** if the object is cue-size in the current part or score, and **False** if the object is normal size (read/write).
CurrentTempo	Returns the tempo, in bpm, at the location of the object in the score (read only).
DrawOrder	Returns the layer at which the object is currently drawn. When used to set the layer of an object, values from **1** (meaning the bottom layer) to **32** (meaning the highest layer) can be used; **0** is a special value that tells Sibelius to use the default layer for that type of object (read/write).
Dx	The horizontal graphic offset of the object from the position implied by the **Position** field, in units of 1/32 spaces (read/write).
Dy	The vertical graphic offset of the object from the centre staff line, in units of 1/32 spaces, positive going upwards (read/write).
HasCustomDrawOrder	Returns **True** if the object is set to a layer other than its default layer (read only).
Hidden	**True** if the object is hidden in the current part or score, and **False** if the object is shown (read/write).
OnNthBlankPage	Returns 0 if the object occurs on a page of music, otherwise a number from 1 upwards indicating the *n*th blank page of the bar on which the object occurs (read only).
ParentBar	The Bar containing this BarObject (read only).
Position	Rhythmic position of the object in the bar (read only).
Selected	Returns **True** if the object is currently selected (read only).
Time	The time at which the object occurs in the score in milliseconds (read only).
Type	A string describing the type of object, e.g. "NoteRest," "Clef." This is useful when hunting for a specific type of object in a bar. See **GuitarScaleDiagram type values** on page 140 for the possible values (read only).
UsesMagneticLayout	Returns **True** if the object is positioned by Magnetic Layout. Returns **False** if the object is set not to be taken into account by Magnetic Layout. To set whether or not an object should use Magnetic Layout, use one of the global constants **AlwaysDodge** (equivalent to Edit ▸ Magnetic Layout ▸ On), **SuppressDodge** (Edit ▸ Magnetic Layout ▸ Off) or **DefaultDodge** (Edit ▸ Magnetic Layout ▸ Default) (read/write).

`UsesMagneticLayoutSettingOverridden`

Returns **`True`** if the object has had its Magnetic Layout settings overridden; otherwise **`False`**.

`VoiceNumber` Is **`0`** if the item belongs to more than one voice (a lot of items belong to more than one voice) and **`1`** to **`4`** for items that belong to voices 1 to 4 (read only).

`Voices` Returns or sets Sibelius's internal bitfield that represents the voices to which an object belongs; useful for copying the voices used by a given object (read/write).

BarRest

Derived from a BarObject object.

Methods

None.

Variables

`PauseType` — Returns the type of fermata (pause), if any, on the bar rest. Returns one of the constants **`PauseTypeNone`** (**`0`**), **`PauseTypeSquare`** (**`1`**), **`PauseTypeRound`** (**`2`**), **`PauseTypeTriangular`** (**`3`**) (read/write).

`RestType` — Returns the type of bar rest via one of the constants **`WholeBarRest`** (**`0`**), **`BreveBarRest`** (**`1`**), **`OneBarRepeat`** (**`2`**), **`TwoBarRepeat`** (**`3`**), **`FourBarRepeat`** (**`4`**) (read only). To create a bar rest of a particular type, use **`bar.InsertBarRest()`** (see above).

Clef

Derived from a BarObject

Methods

None.

Variables

`ClefStyle`	The name of this clef, which may be different depending on the state of Notes ▸ Transposing Score (read only).
`ConcertClefStyleId`	The concert pitch identifier of the style of this clef (read only).
`ConcertClefStyle`	The concert pitch name of this clef (read only).
`StyleId`	The identifier of the style of this clef, which may be different depending on whether or not Notes ▸ Transposing Score is switched on. This can be passed to the **`Bar.AddClef`** method to create a clef of this style (read only).
`TransposingClefStyle`	The transposing score name of this clef (read only).
`TransposingClefStyleId`	The transposing score identifier of the style of this clef (read only).

Comment

Derived from a BarObject.

Methods

`AddComment(`*sr*`,`*text*`[,`*color*`[,`*maximized*`]]`**`)`**

Adds a comment at the specified *sr* position in the current bar, displaying the specified *text*. The optional *color* parameter allows you to specify the color of the comment that is created (if not specified, the comment is created with its default color), and the optional *maximized* Boolean parameter allows you to set the comment to be minimized (if not specified, the comment is created maximized by default).

`AddCommentWithName(`*sr*`,`*text*`,`*username*`[,`*color*`[,`*maximized*`]]`**`)`**

Adds a comment that will display a given *username* at the specified *sr* position in the current bar, displaying the specified *text*. The optional *color* parameter allows you to specify the color of the comment that is created (if not specified, the comment is created with its default color), and the optional *maximized* Boolean parameter allows you to set the comment to be minimized (if not specified, the comment is created maximized by default).

Variables

`Maximized` Returns True if the comment is maximized, otherwise returns False (read/write).

`Text` Returns the text of the comment (read/write).

`TextWithFormatting` Returns an array containing the various changes of font or style (if any) within the comment's text in a new element (read only). For example, "This text is \B\bold\b\, and this is \I\italic\i\" would return an array with eight elements containing the following data:

```
arr[0] = “This text is “
arr[1] = “\B\”
arr[2] = “bold”
arr[3] = “\b\”
arr[4] = “, and this is “
arr[5] = “\I\”
arr[6] = “italic”
arr[7] = “\i\”
```

`TextWithFormattingAsString` The comment's text including any changes of font or style (read only).

`TimeStamp` Returns a **`DateTime`** object corresponding to the date the comment was created or last edited (read only).

`UserName` Returns the username of the user who created or last edited the comment (read only).

ComponentList

An array that is obtained from **`Sibelius.HouseStyles`** or **`Sibelius.ManuscriptPapers`**. It can be used in a **`for each`** loop or as an array with the **`[n]`** operator to access each Component object:

Methods

None.

Variables

`NumChildren`	Number of plug-ins (read only).

Component

This represents a Sibelius "component," namely a house style or a manuscript paper. Examples:

```
// Create a new score using the first manuscript paper
papers=Sibelius.ManuscriptPapers;
score=Sibelius.New(papers[0]);
// Apply the first house style to the new score
styles=Sibelius.HouseStyles;
score.ApplyStyle(styles[0], "ALLSTYLES");
```

Methods

None.

Variables

`Name`	The name of the component (read only).

DateTime

This object returns information about the current date and time.

Methods

None.

Variables

`Seconds`	Returns the number of seconds from the time in a date (read only).
`Minutes`	Returns the number of minutes from the time in a date (read only).
`Hours`	Returns the number of hours from the time in a date (read only).
`DayOfMonth`	returns the nth day on the month, 1-based (read only).
`Month`	returns the nth month of the year, 1-based (read only).
`Year`	returns the year (read only).
`NthDayOfWeek`	returns the nth day of the week, 0-based (read only).
`NthDayOfYear`	returns the nth day of the year, 0-based (read only).
`LongDate`	returns the date in a human-readable format, e.g. 1st May 2008 (read only).
`ShortDate`	returns the date in a human-readable format, e.g. 01/05/2008 (read only).
`LongDateAndTime`	returns the date and time in a human-readable format, e.g. 1st May 2008 14:07 (read only).
`ShortDateAndTime`	returns the date and time in a human-readable format, e.g. 01/05/2008 14:07 (read only).
`TimeWithSeconds`	returns the time in a human-readable format, e.g. 14:07 (read only).
`TimeWithoutSeconds`	returns the time in a human-readable format, e.g. 14:07:23 (read only).

Dictionary

For more details about using dictionaries in ManuScript, see **Dictionary** on page 21.

To create a dictionary, use the built-in function **`CreateDictionary(`***name1*`, `*value1*`, `*name2*`, `*value2*`, ... `*nameN*`, `*valueN*`)`**. This creates a dictionary containing user properties called *name1*, *name2*, *nameN* with values *value1*, *value2*, *valueN* respectively.

Methods

`CallMethod(`*methodname*`,`*param1*`,`*param2*`,...`*paramN*`)`**

Calls the specified method *methodname* in the dictionary, passing in any other values that are required for the method as further parameters.

`GetMethodNames()`

Returns a sparse array containing the names of the methods belonging to a dictionary.

`GetPropertyNames()`

Returns a sparse array of the names of all the user properties in the dictionary (same as **`_propertyNames`**).

`MethodExists(`*methodname*`)`**

Returns **`True`** if the specified method *methodname* exists in the dictionary.

`PropertyExists(`*propertyname*`)`**

Returns **`True`** if the specified user property *propertyname* exists in the dictionary.

`SetMethod(`*methodname*`,Self,`*method*`)`**

Binds a method to the dictionary. *methodname* is the name by which you want to access the method via the dictionary, **`Self`** refers to the plug-in in which the method is found, and *method* is the name of the method itself, found elsewhere in the plug-in.

Variables

None.

Converting old-style hash tables to dictionaries

The Dictionary object is, among other things, a replacement for the old Hash object, which was a simple hash table object. You are recommended to use the new Dictionary object instead of the old Hash object in your plug-ins, but if you have an existing plug-in in which old-style hashes are used, you can convert them to Dictionaries as follows:

`Hash.ConvertToDictionary()` returns a new Dictionary object, populated with strings converted from the old-style Hash.

DynamicPartCollection

Accessed from a Score object. Contains DynamicPart objects.

The DynamicPartCollection object always contains the full score as the first entry, whether or not any dynamic parts exist. The DynamicPart objects are returned in the order in which they were created (the last part returned is the most-recently created one). For scores in which dynamic parts were generated automatically, the parts will normally be returned in top to bottom score order.

The edit context for ManuScript is stored in the score itself which means that ManuScript can only ever access one part at a time – the "current" DynamicPart for that Score object. This is irrespective of the number of score windows open for a score, which dynamic parts are open, and even if the user has managed to create two different ManuScript Score objects referring to the same Sibelius score.

It is inadvisable to modify Staves, Bars, or any BarObjects that do not exist on Staves in **`Score.CurrentDynamicPart`**. Doing so will create part overrides for part-specific properties of these objects which will be invisible until those Staves are added to the part. **`DynamicPart.IncludesStaff()`** can be used to test if a DynamicPart contains a particular Staff object.

Both DynamicPartCollection and DynamicPart refer to an underlying Score and part(s) and will generate errors if the Score and/or part(s) are no longer valid (e.g. if a DynamicPart has been deleted). DynamicParts are never "re-used." For example, if you delete a DynamicPart and create a new DynamicPart, the old ManuScript DynamicPart object will not refer to the newly-created DynamicPart.

`for each` *variable* **`in`** iterates through all valid DynamicPart objects for the Score, always starting first with the full score. Adding or deleting parts while iterating will have undefined results, and is not recommended.

Array access *[int n]* returns the *n*th part (**`0`** is always the full score), or null if the part does not exist.

Methods

`CreateDefaultParts()`

Creates the default set of dynamic parts, as created automatically by Sibelius when clicking the New Part button in the Parts window. This method does nothing and returns **`False`** if the Score has no staves.

`CreatePartFromStaff(`*staff*`)`**

Creates a dynamic part from the specified Staff object, if valid. Returns the new DynamicPart object for success, or null for failure.

`DeletePart(`*dynamic part*`)`**

Deletes the specified part, if it's valid. Returns **`True`** for success, **`False`** for failure. This method fails is the specified dynamic part is the currently active part for the Score, or is the full score, or refers to a different Score.

Variables

`NumChildren`	Returns the number of DynamicPart objects for the Score returned by iteration (read only).

DynamicPart

Accessed from a DynamicPartCollection object.

`for each` *variable* **`in`** returns the Staff objects in the dynamic part, in top to bottom order. Warning: this can return a Staff that is not included in **`Score.CurrentDynamicPart`**.

Methods

`AddStaffToPart(`*staff***`)`**

Adds the specified *staff* to the bottom of the dynamic part. Returns **`False`** for failure. This method will cause an error if it is called on the full score, or if attempting to add a staff that is already present in the part or a staff from a different score.

`DeleteStaffFromPart(`*staff***`)`**

Deletes the specified *staff* from the dynamic part. Returns False for failure. This method will cause an error if called on the full score, or if attempting to delete a staff that is not present in the part, or if deleting the last staff in a part, or attempting to delete a part from a different score.

`IncludesStaff(`*staff***`)`**

Returns **`True`** if the specified *staff* is contained in this dynamic part.

Variables

`IsFullScore`	Returns **`True`** if this is the full score (read only).
`IsSelectedInPartsWindow`	Returns **`True`** if the part is selected in the Parts window (read only).
`StaveCount`	Returns the number of staves in the part (read only).
`ParentScore`	Returns the Score object containing this dynamic part (read only).

File

Retrievable using **`for each`** on a folder.

Methods

`Delete()`

Deletes a file, returning True if successful.

`Rename(`*newFileName*`)`**

Renames a file, returning True if successful.

Variables

`CreationDate`	Returns the file's creation date and time as a DateTime object, in local time (read only).
`CreationDateAndTime`	A string giving the date and time the file was last modified in GMT (read only).
`ModificationDate`	Returns the file's modification date and time as a DateTime object, in local time (read only).
`Name`	The complete pathname of the file, no extension (read only).
`NameWithExt`	The complete pathname of the file, with extension (read only).
`NameNoPath`	Just the name of the file, no extension (read only).
`Path`	Returns just the path to the file (read only).
`Type`	A string giving the name of the type of the object; File for file objects (read only).

Folder

Retrievable from methods of the Sibelius object.

`for each` *variable* **`in`** produces the Sibelius files in the folder, as File objects.

`for each` *type variable* **`in`** produces the files of type *type* in the folder, where *type* is a Windows extension. Useful values are SIB (Sibelius files), MID (MIDI files) or OPT (PhotoScore files), because they can all be opened directly by Sibelius. On the Macintosh files of the corresponding Mac OS Type are also returned (so, for example, **`for each MID f`** will return all files whose names end in .MID, and all files of type "Midi").

Both these statements return subfolders recursively.

Methods

`FileCount(`*Type***`)`**

Returns the number of files of type *Type* in the folder. As above, useful values are SIB, MID or OPT.

Variables

`FileCount`	The number of Sibelius files in the folder (read only).
`Name`	The name of the folder (read only).
`Type`	A string giving the name of the type of the object; Folder for folder objects (read only).

GuitarFrame

Derived from a BarObject. This refers to chord symbols as created by Create ▸ Chord Symbol, whether or not they show a guitar chord diagram (guitar frame), but is called GuitarFrame in ManuScript for historical reasons.

Methods

`GetChromaticPitchesOfChordInClosePosition(`*consider root***`)`**

Returns an array containing the chromatic pitches of the notes in the chord, assuming a voicing in close position. If consider root is True (it defaults to False), the pitches returned will be offset according to the chromatic value of the root note on which the chord is based.

`GetEndStringForNthBarre(`*barreNum***`)`**

Returns the string number on which the nth barré ends.

`GetPitchOfNthString(`*stringNum***`)`**

Returns the pitch of the given (open) string *stringNum*, as a MIDI pitch.

`GetPositionOfFingerForNthBarre(`*barreNum***`)`**

Returns the fret position that the nth barré occupies.

`GetPositionOfFingerOnNthString(`*stringNum***`)`**

Returns the position of the black dot representing the finger position on a given string *stringNum*, relative to the top of the frame. A return value of 0 means the string is open (i.e. a hollow circle appears at the top of the diagram), and -1 means that the string is not played (i.e. an X appears at the top of the diagram). Used in conjunction with **`GetPitchOfNthString()`**, you can calculate the resulting pitch of each string.

`GetStartStringForNthBarre(`*barreNum***`)`**

Returns the string number from which the nth barré begins.

`IsNthStringPartOfBarre(`*stringNum***`)`**

Returns **`True`** if the given string is part of a barré.

`NthStringHasClosedMarkingAtNut(`*nth string***`)`**

Returns True if there's an X marking at the top or left of the specified string.

`NthStringHasOpenMarkingAtNut(`*nth string***`)`**

Returns True if there's an O marking at the top or left of the specified string.

Variables

`BassAsString`	The note name of the chord symbol's altered bass note (e.g. "F").
`ChordNameAsStyledString`	The name of the chord represented by this chord symbol as it appears in the score, e.g. "Cm$^{\Delta 7}$" (read only).
`ChordNameAsPlainText`	The name of the chord represented by this chord symbol as it appears when editing the chord symbol, i.e. in its plain text representation, e.g. "Cmmaj7" (read only).
`ChromaticRoot`	The chromatic pitch (C = 0, B = 11) of the chord symbol's root note (read only).
`ChromaticBass`	The chromatic pitch (C = 0, B = 11) of the chord symbol's altered bass note (read only).
`DiatonicRoot`	The diatonic pitch (-1 for B♯, 0 for C, 4 for G, 7 for C♭, etc.) of the chord symbol's root note (read only).
`DiatonicBass`	The diatonic pitch (-1 for B♯, 0 for C, 4 for G, 7 for C♭, etc.) of the chord symbol's altered bass note (read only).

Fingerings The fingerings string for this chord. This is a textual string with as many characters as the guitar frame has strings (i.e. six for standard guitars). Each character corresponds to a guitar string. Use - to denote that a string has no fingering.

FrameIsVisible True if the chord symbol is currently showing a guitar chord diagram (read only).

Horizontal True if the guitar chord diagram is horizontally orientated, False if it is vertically orientated (read/write).

LowestVisibleFret The number of the top fret shown in the guitar chord diagram; setting the special value **-1** resets the lowest visible fret to the default for that chord diagram (read/write).

NumBarresInChord The number of unique barrés in the guitar chord diagram (read only).

NumberOfFrets The number of frets in the guitar chord diagram, i.e. the number of horizontal lines; setting the special value **-1** resets the number of frets to the default for that chord diagram (read/write).

NumberOfStrings The number of strings in the guitar chord diagram, i.e. the number of vertical lines (read only).

NumPitchesInClosePosition The number of unique pitches in the chord, assuming a voicing in close position with no duplicates.

Recognized Returns **True** if the chord symbol is a specific recognized chord type, and **False** otherwise, i.e. if the chord symbol is shown in red in the score because Sibelius is unable to parse the user's input (read only).

RootAsString The note name of the chord symbol's root (e.g. "C#").

ScaleFactor The scale factor of the guitar chord diagram (as adjustable via the Scale parameter on the General panel of Properties), expressed as a percentage (read/write).

ShowFingerings Set to **True** if the fingerings string should be displayed, **False** otherwise (read only).

Suffixes Returns an array containing a list of the suffix elements present in the chord (read only). If the chord symbol is an unrecognised chord type, the array returned will be empty. The values that can be returned in the array are as follows:

halfdim	**dim**
add6/9	**6/9**
sus2/4	**aug**
omit5	**alt**
omit3	**b13**
maj13	**#11**
add13	**13**
maj11	**11**
dim13	**#9**
dim11	**b9**
maj9	**b6**
add9	**#5**
maj7	**b5**
dim9	**#4**
dim7	**nc**
sus9	**9**
sus4	**7**
add4	**6**
sus2	**5**
add2	**m**
maj	**/**

`SuffixText` The suffix part of the chord symbol as it appears in the score, or an empty string if the chord isn't recognised (read only).

`TextIsVisible` True if the chord symbol is currently showing a text chord symbol (read only).

`VisibleComponents` The visible parts of the chord symbol, i.e. whether it displays a text chord symbol only (**`TextOnly`**), a guitar chord diagram only (**`FrameOnly`**), both a text chord symbol and a guitar chord diagram (**`FrameAndText`**), or whether or not the chord symbol shows a guitar chord diagram based on the type of instrument to which it is attached (**`InstrumentDependent`**) (read/write).

GuitarScaleDiagram

Derived from a BarObject. This refers to guitar scale diagrams as created by Create ▸ Guitar Scale Diagram.

Methods

`GetDotFingeringsOnNthString(`*nth string***`)`**

Returns an array of strings containing the text that has been entered on the dots on a given string.

`GetDotSymbolsOnNthString(`*nth string***`)`**

Returns an array of values describing the appearance of each of the dots on a given string. The possible values are **`DotStyleCircle`**, **`DotStyleFilledCircle`**, **`DotStyleSquare`**, **`DotStyleFilledSquare`**, **`DotStyleDiamond`**, and **`DotStyleFilledDiamond`**.

`GetPitchesOfDotsOnNthString(`*nth string***`)`**

Returns an array containing the pitches of all the dots on a given string, in ascending order of pitch.

`GetPitchOfNthString(`*stringNum***`)`**

Returns the pitch of the given (open) string *stringNum*, as a MIDI pitch.

Variables

`Fingerings`	The fingerings string for this scale diagram. This is a textual string with as many characters as the scale diagram has strings (i.e. six for standard guitars). Each character corresponds to a guitar string. Use - to denote that a string has no fingering.
`Horizontal`	True if the guitar scale diagram is horizontally orientated, False if it is vertically orientated (read/write).
`LowestVisibleFret`	The number of the top fret shown in the guitar scale diagram; setting the special value **`-1`** resets the lowest visible fret to the default for that scale diagram (read/write).
`NumberOfFrets`	The number of frets in the guitar scale diagram, i.e. the number of horizontal lines; setting the special value **`-1`** resets the number of frets to the default for that scale diagram (read/write).
`NumberOfStrings`	The number of strings in the guitar scale diagram, i.e. the number of vertical lines (read only).
`Root`	Returns the chromatic pitch (C = 0) of the scale's root note (read only).
`ScaleFactor`	The scale factor of the guitar scale diagram (as adjustable via the Scale parameter on the General panel of Properties), expressed as a percentage (read/write).
`ScaleType`	Returns the type of the guitar scale diagram, as specified in the list of **GuitarScaleDiagram type values** on page 140 (read only).
`ShowFingerings`	Set to **`True`** if the fingerings string should be displayed, **`False`** otherwise (read only).

InstrumentChange

Derived from a Bar object. Provides information about any instrument changes that may exist in the score.

Methods

None.

Variables

`StyleId`	Returns the style ID of the new instrument; see **Instrument types** on page 122 (read only).
`TextLabel`	Returns the text that appears above the staff containing the instrument change in the score (read only).

InstrumentTypeList

Contains a list of InstrumentType objects common to a given score.

`for each` *type variable* **`in`** returns each instrument type in the list, in alphabetical order by the instrument type's style ID.

Array access *[int n]* returns the *n*th instrument type, in the same order as using a **`for each`** iterator, or null if the instrument type does not exist.

Methods

None.

Variables

`NumChildren`	Returns the number of unique instrument types in the list (read only).

InstrumentType

Provides information about an individual instrument type.

Methods

`PitchOfNthString(`*string num***`)`**

Returns the pitch of a given string in a tablature staff, with string number `0` being the lowest string on the instrument.

Variables

`Balance` Returns the instrument's default balance, in the range **`0`**–**`100`** (read only).

`Category` Returns an index representing the category of the staff type belonging to this instrument type; **`0`** = pitched; **`1`** = percussion; **`2`** = tablature (read only).

`ChromaticTransposition` Returns the number of half-steps (semitones) describing the transposition of transposing instruments; e.g. for B♭ Clarinet, this returns **`-2`** (read only).

`ChromaticTranspositionInScore`
Returns the number of half-steps (semitones) describing the transposition of transposing instruments in the score (as opposed to the parts). Typically this is only used by instruments that transpose by octaves, so this will return e.g. **`12`** for piccolo or **`-12`** for guitars (read only).

`ComfortableRangeHigh` Returns the highest comfortable note (MIDI pitch) of the instrument (read only).

`ComfortableRangeLow` Returns the lowest comfortable note (MIDI pitch) of the instrument (read only).

`ConcertClefStyleId` Returns the style ID of the normal clef style of the instrument (read only).

`DefaultSoundId` Returns the default sound ID used by the instrument (read only).

`DiatonicTransposition` Returns the number of diatonic steps describing the transposition of transposing instruments; e.g. for B♭ Clarinet, this returns **`-1`** (read only).

`DiatonicTranspositionInScore`
Returns the number of diatonic steps describing the transposition of transposing instruments in the score (as opposed to the parts). This will return **`0`** for all pre-defined instruments (read only).

`DialogName` Returns the name of the instrument as displayed in the Instruments as Staves dialog in Sibelius (read only).

`FullName` Returns the name of the instrument as visible on systems showing full instrument names (read only).

`HasBracket` Returns **`True`** if the instrument has a bracket (read only).

`IsVocal` Returns **`True`** if the instrument type used has the Vocal staff option switched on, meaning that e.g. the default positions of dynamics should be above the staff rather than below (read only).

`NumStaveLines` Returns the number of staff lines in the staff (read only).

`NumStrings` Returns the number of strings in a tablature staff (read only).

`OtherClefStyleId` Returns the style ID of the clef style of the second staff of grand staff instruments, e.g. piano (read only).

`Pan` Returns the instrument's default pan setting, in the range **`–127`** (hard left) to **`127`** (hard right) (read only).

`ProfessionalRangeHigh` Returns the highest playable note (MIDI pitch) of the instrument for a professional player (read only).

`ProfessionalRangeLow` Returns the lowest playable note (MIDI pitch) of the instrument for a professional player (read only).

`ShortName` Returns the name of the instrument as visible on systems showing short instrument names (read only).

`StyleId` Returns the style ID of the instrument; see **Global constants** on page 119 (read only).

`TransposingClefStyleId` Returns the style ID of the clef to be used when Notes‣ Transposing Score is switched on (read only).

HitPointList

Retrievable as the **`HitPoints`** variable of a score. It can be used in a **`for each`** loop or as an array with the **`[n]`** operator – this gives access to a HitPoint object. The HitPoint objects are stored in time order, so be careful if you remove or modify the time of the objects inside a loop. If you want to change the times of all the hit points by the same value then use the **`ShiftTimes`** function.

Methods

`Clear()`

Removes all hit points from the score.

`CreateHitPoint(`*timeMs*`,`*label*`)`**

Creates a hit point in the score at the given time (specified in milliseconds) with a specified string label. Returns the index in the HitPointList at which the new hit point was created.

`Remove(`*index*`)`**

Removes the given hit point number.

`ShiftTimes(`*timeMs*`)`**

Adds the given time (in milliseconds) onto all the hit points. If the time is negative then this is subtracted from all the hit points.

Variables

`NumChildren`	Number of hit points (read only).

HitPoint

Individual element of the HitPointList object.

Methods

None.

Variables

`Bar`	The bar in which this hit point occurs (read only).
`Label`	The name of the hit point (read/write).
`Position`	The position within the bar at which this hit point occurs (read only).
`Time`	The time of the hit point in milliseconds. Note that changing this value may change the position of the hit point in the HitPointList (read/write).

KeySignature

Derived from a BarObject.

Methods

None

Variables

`AsText`	The name of the key signature as a string (read only).
`IsOneStaffOnly`	True if this key signature belongs to one staff only (read only).
`Major`	True if this key signature is a major key (read only).
`Sharps`	The number of sharps (positive) or flats (negative) in this key signature (read only).

Line

Anything you can create from the Create ▸ Line dialog is a line object, eg. CrescendoLine, DiminuendoLine, etc. These objects are derived from a BarObject.

Methods

None.

Variables

`Duration`	The total duration of the line, in 1/256th quarters (read/write).
`EndBarNumber`	The bar number in which the line ends (read only).
`EndPosition`	The position within the final bar at which the line ends (read only).
`RhDy`	The vertical graphic offset of the right hand side of the line from the centre staff line, in units of 1/32 spaces, positive going upwards (read/write).
`StyleId`	The identifier of the line style associated with this line (read only).
`StyleAsText`	The name of the line style associated with this line (read only).

LyricItem

Derived from a BarObject

Methods

None.

Variables

`Duration`	The total duration of the lyric line, in 1/256th quarters (see **Line** on page 74) (read/write).
`NumNotes`	Gives the number of notes occupied by this lyric item (read/write). Note that changing this value will not automatically change the length of the lyric line; you also need to set the lyric line's **`Duration`** variable to the correct length.
`StyleAsText`	The text style name (read/write).
`StyleId`	The identifier of the text style of this lyric (read/write).
`SyllableType`	An integer indicating whether the lyric is the end of a word (**`EndOfWord`**) or the start or middle of one (**`MiddleOfWord`**) (read/write). This affects how the lyric is jusitifed, and the appearance of hyphens that follow it. **`EndOfWord`** and **`MiddleOfWord`** are global constants; see **SyllableTypes for LyricItems** on page 137.
`Text`	The text as a string (read/write).

NoteRest

Derived from a BarObject. A NoteRest contains Note objects, stored in order of increasing diatonic pitch.

`for each` *variable* **`in`** returns the notes in the NoteRest.

Methods

`AddAcciaccaturaBefore(`*sounding pitch*`,`*[duration[*`,`*tied [*`,`*voice [*`,`*diatonic pitch[*`,`*string number[*`,`*force stem dir]]]]]]*`)`

Adds a grace note with a slash on its stem (acciaccatura) before a given NoteRest. The duration should be specified as normal, for example, 128 would create a grace note with one beam/flag. The optional *tied* parameter should be **`True`** if you want the note to be tied. Voice 1 is assumed unless the optional *voice* parameter (with a value of 1, 2, 3 or 4) is specified. If *force stem dir* is set to True (the default), stems of graces notes in voices 1 and 3 will always point upwards, and stems of notes in voices 2 and 4, downwards. You can also set the diatonic pitch, i.e. the number of the "note name" to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on). If a diatonic pitch of zero is given then a suitable diatonic pitch will be calculated from the MIDI pitch. The optional string number parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Note Input page of File ▸ Preferences). Returns the Note object created (to get the NoteRest containing the note, use **`Note.ParentNoteRest`**).

Note that adding a grace note before a NoteRest will *always* create an additional grace note, just to the left of the note/rest to which it is attached. If you wish to create grace notes with more than one pitch, you should call **`AddNote`** on the object returned.

`AddAppoggiaturaBefore(`*sounding pitch*`,`*[duration[*`,`*tied [*`,`*voice [*`,`*diatonic pitch[*`,`*string number[*`,`*force stem dir]]]]]]*`)`

Identical to **`AddAcciaccaturaBefore`**, only no slash is added to the note's stem.

`AddNote(`*pitch[*`,`*tied[*`,`*diatonic pitch[*`,`*string number]]]*`)`

Adds a note with the given MIDI pitch (60 = middle C), e.g. to create a chord. The optional second parameter specifies whether or not this note is tied (True or False). The optional third parameter gives a diatonic pitch, i.e. the number of the 'note name' to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E etc.). If this parameter is 0 then a default diatonic pitch will be calculated from the MIDI pitch. The optional fourth parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Notes page of File ▸ Preferences). Returns the Note object created.

`Delete()`

Deletes all the notes in the NoteRest, converting the entire chord into a rest of similar duration.

`FlipStem()`

Flips the stem of this NoteRest – this acts as a toggle.

`GetArticulation(`*articulation number*`)`**

Returns **`True`** or **`False`** depending on whether the given articulation is currently set on this note. The valid articulation numbers are defined in **Articulations** on page 136.

`NoteRest[`*array element*`]`**

Returns the nth note in the chord, in order of increasing diatonic pitch (counting from 0). For example, **`NoteRest[0]`** returns the lowest note (in terms of diatonic pitch – see **`AddNote`** below).

`RemoveNote(`*note*`)`**

Removes the specified Note object.

`SetArticulation(`*articulation number*`,`*set*`)`**

If *set* is **`True`**, turns on the given articulation; otherwise turns it off. The valid articulation numbers are defined in **Articulations** on page 136.

Transpose(*degree, interval type[***,** *keep double accs]***)**

Transposes the entire NoteRest up or down by a specified *degree* and *interval type*. To transpose up, use positive values for *degree*; to transpose down, use negative values. Note that degrees are 0-based, so 0 is equal to a unison, 1 to a second and so on. For descriptions of the various available interval types, see **Global constants** on page 119. By default, Sibelius will transpose using double sharps and flats where necessary, but this behavior may be suppressed by setting the *keep double accs* flag to False.

For help in calculating the interval and degree required for a particular transposition, see the documentation for the **Sibelius.CalculateInterval** and **Sibelius.CalculateDegree** methods.

Variables

ArpeggioDx — The horizontal offset of the arpeggio line on the NoteRest (read/write).

ArpeggioType — The type of note-attached arpeggio line present on the NoteRest. Values are **ArpeggioTypeNone**, **ArpeggioTypeNormal**, **ArpeggioTypeUp**, **ArpeggioTypeDown** (read/write).

ArpeggioTopDy — The vertical offset of the top of the note-attached arpeggio line on the NoteRest (read/write).

ArpeggioBottomDy — The vertical offset of the bottom of the note-attached arpeggio line on the NoteRest (read/write).

ArpeggioHidden — Returns **True** if the note-attached arpeggio line on the NoteRest is hidden (read/write).

Articulations — Enables you to copy a set of articulations from one NoteRest to another (read/write), e.g:

```
destNr.Articulations = sourceNr.Articulations;
```

Beam — Takes values **StartBeam**, **ContinueBeam**, **NoBeam** and **SingleBeam**. (see **Global constants** on page 119 for details). These correspond to the keys 7, 8, * (/ on Mac) and / (* on Mac) on the third (F9) Keypad layout.

DoubleTremolos — Gives the number of double tremolo strokes starting at this note, in the range 0–7. Means nothing for rests. To create a double tremolo between two successive notes, ensure they have the same duration and set the **DoubleTremolos** of the first one (read/write).

Duration — The duration of the note rest (read only).

FallDx — The horizontal offset of a fall, if present on the NoteRest (read/write).

FallType — The type of note-attached fall present on the NoteRest. Values are **FallTypeNone**, **FallTypeNormal** and **FallTypeDoit** (read/write)

FeatheredBeamType — Returns one of three values, based on whether a note is set to produce a feathered beam. Values are **FeatheredBeamNone** (**0**), **FeatheredBeamAccel** (**1**) and **FeatheredBeamRit** (**2**) (read/write).

HasStemlet — Returns **True** if the note is showing a stemlet, according either to the state of the Use stemlets on beamed rests option on the Beams and Stems page of Engraving Rules or the stemlet button on the Keypad (read only).

Highest — The highest Note object in the chord (read only).

Lowest — The lowest Note object in the chord (read only).

NoteCount — The number of notes in the chord (read only).

GraceNote — **True** if it's a grace note (read only).

IsAcciaccatura — **True** if it's an acciaccatura, i.e. a grace note with a slash through its stem (read only).

IsAppoggiatura — **True** if it's an appoggiatura, i.e. a grace note without a slash through its stem (read only).

ParentTupletIfAny — If the NoteRest intersects a tuplet, the innermost Tuplet object at that point in the score is returned. Otherwise, *null* is returned (read only).

PositionInTuplet Returns the position of the NoteRest relative to the duration and scale-factor of its parent tuplet. If the NoteRest does not intersect a tuplet, its position within the parent Bar is returned as usual (read only).

ScoopDx The horizontal offset of a scoop or plop, if present on the NoteRest (read/write).

ScoopType The type of note-attached scoop present on the NoteRest. Values are **ScoopTypeNone**, **ScoopTypeNormal**, **ScoopTypePlop** (read/write).

StemFlipped **True** if the stem is flipped (read only).

StemletType Provides information about whether the NoteRest is set to display a stemlet using the options on the Keypad. Returns either **StemletCustomOff** (in which case the NoteRest definitely does not show a stemlet), **StemletCustomOn** (in which case the NoteRest definitely *does* show a stemlet), or **StemletUseDefault** (in which case you should use the read-only variable **HasStemlet** to determine whether the NoteRest currently shows a stemlet) (read/write).

SingleTremolos Gives the number of tremolo strokes on the stem of this note, in the range -1 (for "z on stem") to 7. Means nothing for rests (read/write).

Note

Only found in NoteRests. Correspond to individual noteheads.

Methods

`Delete()`

Removes a single note from a chord.

`Transpose(`*degree, interval type[*`,`*keep double accs]*`)`**

Transposes and returns a single Note object up or down by a specified *degree* and *interval type**. To transpose up, use positive values for *degree*; to transpose down, use negative values. Note that degrees are 0-based, so 0 is equal to a unison, 1 to a second and so on. For descriptions of the various available interval types, see **Global constants** on page 119. By default, Sibelius will transpose using double sharps and flats where necessary, but this behavior may be suppressed by setting the *keep double accs* flag to False. For help in calculating the interval and degree required for a particular transposition, see the documentation for the **`Sibelius.CalculateInterval`** and **`Sibelius.CalculateDegree`** methods.

* N.B.: Individual note objects cannot be transposed diatonically.

Variables

`Accidental`	The accidental, for which global constants such as **`Sharp`**, **`Flat`** and so on are defined; see **Global constants** on page 119 (read only).
`AccidentalStyle`	The style of the accidental (read/write). This can be any of following four global constants: **`NormalAcc`**, **`HiddenAcc`**, **`CautionaryAcc`** (which forces an accidental to appear always) and **`BracketedAcc`** (which forces the accidental to be drawn inside brackets).
`Bracketed`	The bracketed state of the note, as shown on the F9 layout of the Keypad (read/write).
`DiatonicPitch`	The diatonic pitch of the note, i.e. the number of the "note name" to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on) (read only).
`Name`	The pitch of the note as a string (read only).
`NoteStyle`	The index of the notehead style of this Note (read/write). The styles correspond to those accessible from the Notes panel of the Properties window in Sibelius; see **Note Style names** on page 136 for a complete list of the defined NoteStyles.
`NoteStyleName`	The name of the notehead style of this Note (read/write). If an attempt is made to apply a non-existant style name, the note in question will retain its current notehead.
`OriginalDeltaSr`	The Live start position of this notehead (in 1/256th quarters), as shown in the Playback panel of Properties (read/write). This value can be positive or negative, indicating that the note is moved forwards or backwards.
`OriginalDuration`	The Live duration of this notehead (in 1/256th quarters), as shown in the Playback panel of Properties (read/write).
`OriginalVelocity`	The Live velocity of this notehead (in MIDI volume units, 0-127), as shown in the Playback panel of Properties (read/write). Note that the word "original" refers to the fact that this data is preserved from the original performance if the score was imported from a MIDI file or input via Flexi-time. For further details on this value, and the ones following below, read the **Live Playback** section in Sibelius Reference.
`ParentNoteRest`	The NoteRest object that holds this note (read only).
`Pitch`	The MIDI pitch of the note, in semitones, 60 = middle C (read only).
`Slide`	Is **`True`** if the note has a slide, **`False`** otherwise (read only).

`StringNum` The string number of this note, only defined if the note is on a tablature stave. If no string is specified, reading this value will give –1. Strings are numbered starting at 0 for the bottom string and increasing upwards (read only).

`Tied` Is **`True`** if the note is tied to the following note (read only).

`WrittenAccidental` The accidental, taking transposition into account (read only).

`WrittenDiatonicPitch` The written diatonic pitch of the note, taking transposition into account if **`Score.TransposingScore`** is **`True`** (35 = middle C) (read only).

`WrittenName` The written pitch of the note as a string (taking transposition into account) (read only).

`WrittenPitch` The written MIDI pitch of the note, taking transposition into account if **`Score.TransposingScore`** is **`True`** (60 = middle C) (read only).

`UseOriginalDeltaSrForPlayback` Is **`True`** if the Live start position of this Note should be used for Live Playback. Corresponds to the Live start position checkbox in the Playback panel of the Properties window.

`UseOriginalDurationForPlayback` Is **`True`** if the Live duration of this Note should be used for Live Playback. Corresponds to the Live duration checkbox in the Playback panel of the Properties window.

`UseOriginalVelocityForPlayback` Is **`True`** if the Live velocity of this Note should be used for Live Playback. Corresponds to the Live velocity checkbox in the Playback panel of the Properties window.

PageNumberChange

Provides access to get and set the attributes of a page number change at the end of a bar or on a blank page.

Methods

`SetFormatChangeOnly(`*format change only***`)`**

If *format change only* is True, this has the same effect as switching *off* the New page number check box on the Page Number Change dialog in Sibelius. The page numbering will therefore continue counting consecutively, but it's possible to (for example) hide a group of page numbers and restore visibility at a later point on the score without having to keep track of the previous page numbers.

`SetHideOrShow(`*page number visibility***`)`**

Takes one of the three Page number visibility global constants to determine the visibility of the initial page number change and its subsequent pages; see **Global constants** on page 119.

`SetPageNumber(`*page number***`)`**

Takes an integral number specifying the new number you wish to assign to the page.

`SetPageNumberFormat(`*format***`)`**

Takes one of the four Page number format global constants to change the format used to display the page number change; see **Global constants** on page 119.

Variables

`HideOrShow`	Returns one of the three Page number visibility global constants; see **Global constants** on page 119 (read only).
`PageNumber`	Returns the page number expressed as an integer. For example, page `x` when using Roman numerals would be `10`, or `24` with alphabetics (read only).
`PageNumberAsString`	Returns the page number change as visible on the corresponding page in Sibelius (read only).
`PageNumberBlankPageOffset`	Returns the blank page offset of the page number change, or `0` if there are no blank pages following the bar contaning the page number change (read only).
`PageNumberFormat`	Returns one of four Page number format global constants describing the format of the page number change; see **Global constants** on page 119 (read only).

PluginList

An array that is obtained from **`Sibelius.Plugins`**. It can be used in a **`for each`** loop or as an array with the **`[n]`** operator to access each Plugin object.

Methods

`Contains(`*pluginName***`)`**

Returns **`True`** if a plug-in with the given name is installed. This can be used to query whether a plugin is installed before you try to call it.

Variables

`NumChildren`	Number of plug-ins (read only).

Plugin

This represents an installed plugin. Typical usage:

```
for each p in Sibelius.Plugins
{
   trace("Plugin: " & p.Name);
}
```

Methods

The following methods are intended to allow you to check the existence of specific methods, data and dialogs in plug-ins, which allows you to check in advance that e.g. calling a method in another plug-in will succeed, and fail gracefully if the method is not found:

`MethodExists(`*method***`)`**

Returns **`True`** if the specified *method* exists in the current Plugin object.

`DataExists(`*data***`)`**

Returns **`True`** if the specified *data* exists in the current Plugin object.

`DialogExists(`*dialog***`)`**

Returns **`True`** if the specified *dialog* exists in the current Plugin object.

Variables

`File`	The File object corresponding to the file that the plug-in was loaded from (read only).
`Name`	The name of the plug-in (read only).

RehearsalMark

Derived from a BarObject and found in the system staff only. RehearsalMarks have an internal numbering and a visible text representation, both of which can be read from ManuScript.

Methods

None.

Variables

`Mark`
The internal number of this rehearsal mark. By default rehearsal marks are consecutive (with the first one numbered zero), but the user can also create marks with specific numbers.

`MarkAsText`
The textual representation of this rehearsal mark as drawn in the score. This is determined by the House Style ▸ Engraving Rules options, and can take various forms (numerical or alphabetical).

Score

A Score contains one SystemStaff and one or more Staff objects.

for each *variable* **in** returns each staff in the score or the current dynamic part in turn (not the system staff).

for each *type variable* **in** returns the objects in the score in chronological order, from the top staff to the bottom staff (for simultaneous objects) and then from left to right (again, not including the system staff).

Methods

AddBars(*n***)**

Adds *n* bars to the end of the score.

ApplyStyle(*style file*,"*style*",*["style"]***)**

Imports named styles from the given house style file (.lib) into the score. The style file parameter can either be a full path to the file, or just the name of one of the styles that appears in the House Style ▸ Import House Style dialog. You can import as many "style" elements as you like in the same method. Style names are as follows:

HOUSE, **TEXT**, **SYMBOLS**, **LINES**, **NOTEHEADS**, **CLEFS**, **DICTIONARY**, **SPACINGRULE**, **DEFAULTPARTAPPEARANCE**, **INSTRUMENTSANDENSEMBLES, MAGNETICLAYOUTOPTIONS** or **ALLSTYLES**.

For instance:

```
score2.ApplyStyle("C:\NewStyle.lib", "HOUSE", "TEXT");
```

Note that the constant **HOUSE** refers, for historical reasons, only to those options in the House Style ▸ Engraving Rules and Layout ▸ Document Setup dialogs, not the entire house style. To import the entire House Style, use the **ALLSTYLES** constant.

ClefStyleId(*clef style name***)**

Returns the identifier of the clef style with the given name, or the empty string if there is no such clef style.

CreateInstrument(*style ID[,change names,["full name",["short name"]]]***)**

Creates a new instrument, given the *style ID* of the instrument type required (see **Instrument types** on page 122). If you want to supply the instrument names to be used in the score, set the optional *change names* parameter to **True**, then supply strings for the *full name* and *short name*. Returns **True** if the instrument was created successfully and **False** if the instrument type could not be found.

CreateInstrumentAtBottom(*style ID[,change names,["full name",["short name"]]]***)**

Behaves the same way as **CreateInstrument**, only the new instrument is always created below all other instruments that currently exist in the score. This can be useful when programatically copying a list of staves/instruments from one score to another, as you can guarantee the ordering of the staves will be the same in both scores.

CreateInstrumentAtBottomReturnStave(*style ID[,change names,["full name",["short name"]]]***)**

As above, but returns the Stave object created, or null if unsuccessful.

CreateInstrumentAtTop(*style ID[,change names,["full name",["short name"]]]***)**

Behaves in exactly the same way as **CreateInstrumentAtBottom**, only the new instrument is always created above all other instruments that currently exist in the score.

CreateInstrumentAtTopReturnStave(*style ID[,change names,["full name",["short name"]]]***)**

As above, but returns the Stave object created, or null if unsuccessful.

CreateInstrumentReturnStave(*style ID[,change names,["full name",["short name"]]]***)**

Like **CreateInstrument**, but returns the Stave object created, or null if unsuccessful. Note that if the instrument being created contains more than one staff (e.g. piano or harp), the top stave of the instrument in question will be returned.

ExtractParts(*[show_dialogs],[parts path]***)**

Extracts parts from the score. The first optional parameter can be **`False`**, in which case the parts are extracted without showing an options dialog. The second optional parameter specifies a folder into which to extract the parts (must end with a trailing folder separator).

`GetLocationTime(`*bar number[,position]*`)`**

Returns the time of a given location in the score in milliseconds.

`GetVersions()`

Returns the score's VersionHistory object (see **VersionHistory** on page 115).

`InsertBars(`*n*`,`*barNum*`[,`*length*`])`**

Inserts *n* bars before bar number *barNum.* If no *length* has been specified, the bar will be created with the correct length according to the current time signature. However, irregular bars may also be created by specifying a value for *length*.

`LineStyleId(`*line style name*`)`**

Returns the identifier of the line style with the given name, or the empty string if there is no such line style.

`NoteStyleIndex(`*notehead style name*`)`**

Returns the index of the note style with the given name, or **`-1`** if there is no such note style.

`NthStaff(`*staff index from 1*`)`**

Returns the *n*th staff of the score or the current dynamic part.

`RemoveAllHighlights()`

Removes all highlights in this score.

`RenameTextStyle("`*old name*`","`***new name*`")`**

Renames a text style to a new name.

`Save(`*filename*`)`**

Saves the score, overwriting any previous file with the same name.

`SaveAs(`*filename*`,`*type*`[,`*use_defaults*`,`*foldername*`])`**

Saves the score in a specified format, overwriting any previous file with the same name. The optional argument *use_defaults* only applies to graphics files, and specifies whether or not the default settings are to be used. When set to False, the Export Graphics dialog will appear and allow the user to make any necessary adjustments. The optional *foldername* specifies the folder in which the file is to be saved. The foldername parameter must end with a path separator (i.e. "**`\\`**" on Windows).

The possible values for type are:

`SIBL`	Sibelius format (current version)
`EMF`	EMF
`BMP`	Windows bitmap
`PICT`	PICT format
`PNG`	PNG format
`Midi`	MIDI format
`EPSF`	EPS format
`TIFF`	TIFF format

So, to save a file using the current Sibelius file format, you would write **`score.SaveAs("filename.sib", "SIBL");`**

`SaveAsAudio(`*filename[,include all staves[,play from start]]*`)`**

Creates a WAV file (PC) or AIFF file (Mac) of the score, using Sibelius's File ▸ Export ▸ Audio feature. If *include all staves* is True (the default), Sibelius will first clear any existing selection from the score so every instrument will be recorded; only selected staves will otherwise be exported. When *play from start* is True (also the default), Sibelius will record the entire score from beginning to end, otherwise from the current position of the playback line. Note that **`SaveAsAudio`** will only have an

effect if the user's current playback configuration consists of solely VST and/or AU devices. The functions returns True if successful, otherwise False (including if the user clicks Cancel during export).

`SaveAsSibelius2(`*filename[,foldername]*`)`**

Saves the score in Sibelius 2 format, overwriting any previous file with the same name. The optional foldername specifies the folder in which the file is to be saved. Note that saving as Sibelius 2 may alter some aspects of the score; see Sibelius Reference for full details.

`SaveAsSibelius3(`*filename[,foldername]*`)`**

Saves the score in Sibelius 3 format. See documentation for **`SaveAsSibelius2`** above.

`SaveAsSibelius4(`*filename[,foldername]*`)`**

Saves the score in Sibelius 4 format. See documentation for **`SaveAsSibelius2`** above.

`SaveAsSibelius5(`*filename[,foldername]*`)`**

Saves the score in Sibelius 5 format. See documentation for **`SaveAsSibelius2`** above.

`SaveCopyAs(`*filename[,foldername]*`)`**

Saves a copy of the score in the current version's format without updating the existing score's file name in Sibelius.

`Score[`*array element*`]`**

Returns the nth staff (staff index from 0) e.g. **`Score[0]`**.

`SetPlaybackPos(`*bar number*`,`*sr*`)`**

Sets the position of the playback line to a given *bar number* and rhythmic (*sr*) position.

`StaveTypeId(`*stave type name*`)`**

Returns the identifier of the stave type with the given name, or the empty string if there is no such stave type.

`SystemCount(`*page num*`)`**

The number of systems on a page (the first page of the score is page 1).

`SymbolIndex(`*symbol name*`)`**

Returns the index of the symbol with the given name, or **`-1`** if there is no such symbol.

`TextStyleId(`*text style name*`)`**

Returns the identifier of the text style with the given name, or the empty string if there is no such text style.

Variables

`Arranger`	Arranger of score from File ▸ Score Info (read/write).
`Artist`	Artist of score from File ▸ Score Info (read/write).
`Composer`	Composer of score from File ▸ Score Info (read/write).
`Copyright`	Copyright of score from File ▸ Score Info (read/write).
`CurrentDynamicPart`	Returns or sets the current DynamicPart object for the Score (read/write). Sibelius will not automatically display the new part: use Sibelius.ShowDynamicPart() to change the displayed part.
`CurrentPlaybackPosBar`	Returns the bar number in which the playback line is currently located.
`CurrentPlaybackPosSr`	Returns the rhythmic position within the bar at which the playback line is currently located.
`DynamicParts`	Returns a DynamicPartCollection object representing the dynamic parts present in the Score. This object will always stay up to date, even if parts are added or deleted (read-only).
`EnableScorchPrinting`	Corresponds to the Allow printing and saving checkbox in the Export Scorch Web Page dialog (read/write).

`FileName`	The filename for the score (read only).
`FocusOnStaves`	is **`True`** if View ▸ Focus on Staves is switched on (read/write). See also **`Staves.ShowInFocusOnStaves`**.
`HitPoints`	The HitPointList object for the score (read/write).
`InstrumentChanges`	Value of Instrument changes from File ▸ Score Info (read/write).
`InstrumentTypes`	Returns an **`InstrumentTypeList`** containing the score's instrument types, on which one may execute a **`for each`** loop to get information about each instrument type within the score.
`IsDynamicPart`	Returns **`True`** if the current active score view is a part (read only).
`LiveMode`	Is **`True`** (**`1`**) if Play ▸ Live Playback is on (read/write).
`Lyricist`	Lyricist of score from File ▸ Score Info (read/write).
`MagneticLayoutEnabled`	Returns **`True`** if the current score has Layout ▸ Magnetic Layout switched on (read/write).
`NumberOfPrintCopies`	The number of copies to be printed (read/write).
`OriginalProgramVersion`	The version of Sibelius in which this score was originally created, as an integer in the following format: (major version) * 1000 + (minor version) * 100 + (revision) * 10 So Sibelius 3.1.3 would be returned as **`3130`**.
`OtherInformation`	More information concerning the score from File ▸ Score Info (read/write).
`PageCount`	The number of pages in the score (read only).
`PartName`	Value of Part Name from File ▸ Score Info (read/write).
`Publisher`	Publisher of score from File ▸ Score Info (read/write).
`Redraw`	Set this to **`True`** (**`1`**) to make the score redraw after each change to it, **`False`** (**`0`**) to disallow redrawing (read/write).
`ScoreDuration`	The duration of the score in milliseconds (read only).
`ScoreEndTime`	The duration of the score, plus the score start time (see above), in milliseconds (read only).
`ScoreHeight`	Height of a page in the score, in millimetres (read only).
`ScoreStartTime`	The value of Timecode of first bar, from Play ▸ Video and Time ▸ Timecode and Duration, in milliseconds (read only).
`ScoreWidth`	Width of a page in the score, in millimetres (read only).
`Selection`	The Selection object for the score, i.e. a list of selected objects (read only).
`ShowInFocusOnStaves`	If **`True`** then this staff will be shown when Layout ▸ Focus on Staves is switched on (see also **`Score.FocusOnStaves`**). This variable cannot be set to **`False`** unless it is also **`True`** for at least one other staff in the score (read/write).
`ShowMultiRests`	Is **`True`** (**`1`**) if Layout ▸ Show Multirests is on (read/write).
`StaffCount`	The number of staves in the score (read only).
`StaffHeight`	Staff height, in millimetres (read only).
`SystemCount`	The number of systems in the score (read only).
`SystemStaff`	The SystemStaff object for the score (read only).
`Title`	Title of score from File ▸ Score Info (read/write).
`TransposingScore`	Is **`True`** (**`1`**) if Notes ▸ Transposing Score is on (read/write).

Selection

for each *variable* **in** returns every BarObject (i.e. an object within a bar) in the selection.

for each *type variable* in produces each object of type *type* in the selection. Note that if the selection is a system selection (i.e. surrounded by a double purple box in Sibelius) then objects in the system staff will be returned in such a loop.

Methods

Clear()

Removes any existing selection(s) from the current active score.

ClipboardContainsData(*[clipboard Id]***)**

Returns True if the given clipboard contains data. As with the **Copy** and **Paste** methods, 0 (or no arguments) refers to Sibelius's internal clipboard, and all other numeric values will interrogate the temporary clipboard with the matching ID.

Copy(*[clipboard Id]***)**

Copies the music within the current selection to Sibelius's internal clipboard or a ManuScript-specific temporary clipboard, which goes out of scope along with the Selection object itself. If no *clipboard Id* is specified, or if it is set to 0, the selection will be copied to Sibelius's internal clipboard. Any other numeric value you pass in will store the data in a temporary clipboard adopting the ID you specify. Used in conjuction with **Paste** or **PasteToPosition** (see below).

Delete(*[remove staves]***)**

Deletes the music currently selected in the active score. Akin to making a selection manually in Sibelius and hitting Delete. If remove staves is omitted or set to True, Sibelius will completely remove any wholly selected staves from the score. If you wish Sibelius to simply hide such staves instead, set this flag to False.

ExcludeStaff(*staff number***)**

If a passage selection already exists in the current active score, an individual stave may be removed from the selection using this method.

IncludeStaff(*staff number***)**

If a passage selection already exists in the current active score, a non-consecutive stave may be added to the selection using this method.

Paste(*[clipboard Id[, reset positions]]***)**

Pastes the music from a given clipboard to the start of the selection in the current active score. If no *clipboard Id* is specified, or if it is set to 0, the data will be pasted from Sibelius's internal clipboard. Any other numeric value you pass in will take the data from a temporary clipboard you must have previously created with a call to **Copy** (see above). Returns True if successful.

If *reset positions* is False, the positions of any objects that have been moved by the user in the source selection will be retained in the copy. This is the default behaviour. If you wish Sibelius to reset objects to their default positions, set this flag to True. This can be useful when copying one or more single objects (i.e. a non-passage selection).

Note that pasting into a score using this method will overwrite any existing music. Only one copy of the music will ever be made, so if your selection happens to span more bars or staves than is necessary, the data will *not* be duplicated to fill the entire selection area.

PasteToPosition(*stave num, bar num, position[, clipboard Id[, reset positions]]***)**

Pastes the music from a given clipboard to a specific location in the current active score. The optional parameters and pasting behavior works in the same way as calls to **Paste**.

RestoreSelection()

Restores the selection previously recorded with a call to **StoreCurrentSelection.** Usefully called at the end of a plug-in to restore the initial selection.

SelectPassage(*start barNum[, end barNum[, top staveNum[, bottom staveNum[, start pos[, end pos]]]]]***)**

Programmatically makes a passage selection around a given area of the current active score. When no *end barNum* is given, only the *start barNum* will be selected. If neither a *top-* nor *bottom staveNum* has been specified, every stave in the score will be selected, whereas if only a *top staveNum* has been supplied, only that one staff will be selected. Sibelius will begin the selection from the start of the first bar if no *start pos* has been given, similarly completing the selection at the end of the final bar if no *end pos* has been supplied.

NB: The *start pos* and *end pos* you supply may be altered by ManuScript: Sibelius requires a passage selection to begin and end at a NoteRest if it doesn't encompass the entire bar.

`SelectSystemPassage(`*start barNum[*`,`*end barNum[*`,`*start pos[*`,`*end pos]]]*`)`

Programmatically makes a system selection around a given area of the current active score. When no *end barNum* is given, only the *start barNum* will be selected. Sibelius will begin the selection from the start of the first bar if no *start pos* has been given, similarly completing the selection at the end of the final bar if no *end pos* has been supplied.

NB: The *start pos* and *end pos* you supply may be altered by ManuScript: Sibelius requires a passage selection to begin and end at a NoteRest if it doesn't encompass the entire bar.

`StoreCurrentSelection()`

Stores the current selection in the active score internally. Can be retrieved with a call to **`RestoreSelection`** (see below). Usefully called at the start of a plug-in to store the initial selection.

`Transpose(`*degree, interval type[*`,`*keep double accs[*`,`*transpose keys]]*`)`

Transposes the currently selected music up or down by a specified *degree* and *interval type*. To transpose up, use positive values for *degree*; to transpose down, use negative values. Note that degrees are 0-based, so `0` is equal to a unison, `1` to a second and so on. For descriptions of the various available interval types, see **Global constants** on page 119. By default, Sibelius will transpose using double sharps and flats where necessary, but this behavior may be suppressed by setting the *keep double accs* flag to `False`. Sibelius will also transpose any key signatures within the selection by default, but can be overriden by setting the fourth parameter to `False`.

For help in calculating the interval and degree required for a particular transposition, see the documentation for the **`Sibelius.CalculateInterval`** and **`Sibelius.CalculateDegree`** methods.

Variables

`BottomStaff`	The number of the bottom staff of a passage (read only).
`FirstBarNumber`	The internal bar number of the first bar of a passage (read only).
`FirstBarNumberString`	The external bar number (including any bar number format changes) of the first bar of a passage (read only).
`FirstBarSr`	The position of the start of the passage selection in the first bar (read only).
`IsPassage`	True if the selection represents a passage, as opposed to a multiple selection (read only).
`IsSystemPassage`	True if the selection includes the system staff (read only).
`LastBarNumber`	The internal bar number of the last bar of a passage (read only).
`LastBarNumberString`	The external bar number (including any bar number format changes) of the last bar of a passage (read only).
`LastBarSr`	The position of the end of the passage selection in the last bar (read only).
`TopStaff`	The number of the top staff of a passage (read only).

Copying entire bars

Copying passages from one location in a score to another – or even from one score to another – is very simple. Here is an example function demonstrating how one might go about achieving this:

```
CopyBar(scoreSrc, barFirstSrc, barLastSrc, scoreDest, barFirstDest,
                  barLastDest) // This is the function signature
```

```
{
   sel = scoreSrc.Selection;
   sel.SelectPassage(barFirstSrc.BarNumber, barLastSrc.BarNumber,
                     barFirstSrc.ParentStaff.StaffNum,
                     barLastSrc.ParentStaff.StaffNum);
   sel.Copy(0);
   selDest = scoreDest.Selection;
   selDest.SelectPassage(barFirstDest.BarNumber, barLastDest.BarNumber,
                         barFirstDest.ParentStaff.StaffNum,
                         barLastDest.ParentStaff.StaffNum);
   selDest.Paste(0);
}
```

Note that you may use any temporary clipboard or Sibelius's own internal clipboard if the source and destination locations are in the same score, however you can only use Sibelius's internal clipboard if the data is being transfered between two individual scores. This is because the temporary clipboards belong to the Selection object itself.

Copying multiple selections from one bar to another

Using a combination of the BarObject's **`Select`** method and the Selection object's **`Copy`** and **`PasteToPosition`** methods, it is possible to copy an individual or multiple selection from one location in a score to another. Bear in mind that **`Paste`** will always paste the material to the very start of the selection, so if you're copying a selection that doesn't start at the very beginning of a bar, you'll have to store the position of the first item and pass it to **`PasteToPosition`** when you later come to paste the music to another bar.

This example code below copies all items from position 256 or later from one bar to another. It is assumed that sourceBar is a valid Bar object, and destStaffNum and destBarNum contain the destination staff number and bar number respectively:

```
sel = Sibelius.ActiveScore.Selection; // Get a Selection object for this score
sel.Clear(); // Clear the current selection
clipboardToUse = 1; // This clipboard ID we're going to use
copyFromPos = 256; // Copy all objects from this point in the source bar
posToCopyTo = 0; // Variable used to store the position of the first object copied
for each obj in sourceBar { // Iterate over all objects in the bar
   if (obj.Position >= copyFromPos) { // Ignore objects before the start threshold
      obj.Select(); // Select each relevant object in turn
      if (posToCopyTo = 0) {
         posToCopyTo = obj.Position; // Remember the position of the first item
      {
   }
}
sel.Copy(clipboardToUse); // Copy the objects we've selected to the clipboard

sel.PasteToPosition(destStaffNum, destBarNum, posToCopyTo, clipboardToUse); // And
paste them to the destination bar at the relevant offset
```

Sibelius

There is a predefined variable that represents the Sibelius program. You can use the Sibelius object to open scores, close scores, display dialogs or (most commonly) to get currently open Score objects.

`for each` *variable* **`in`** returns each open score.

Methods

`AppendLineToFile(`*filename*`,`*text*`[,`*use_unicode*`]``)`**

Appends a line of text to the file specified (adds line feed). See comment for **`AppendTextFile`** above for explanation of the *use_unicode* parameter. Returns True if successful.

`AppendLineToRTFFile(`*filename*`,`*text*`)`**

Appends a line of text to the file specified. Times New Roman 12pt is used, unless you specify a change of formatting. To change formatting, use the following backslash expressions:

`\B\` bold on

`\I\` italic on

`\U\` underline on

`\n\` new line

`\b\` bold off

`\i\` italic off

`\u\` underline off

`\f`*fontname*`\`** change to given font name (e.g. **`\fArial\`** to switch to Arial)

`\s`*points*`\`** set the font size to a specific point size (e.g. **`\s16\`** to set the font to 16pts).

`AppendTextFile(`*filename*`,`*text*`[,`*use_unicode*`]``)`**

Appends text to the file specified. If the optional Boolean parameter *use_unicode* is **`True`**, then the string specified will be exported in Unicode format; if this parameter is **`False`** then it will be converted to 8-bit Latin-1 before being added to the text file. This parameter is **`True`** by default. Returns True if successful.

`CalculateDegree(`*source pitch, dest pitch, upward interval*`)`**

Takes two note names in the form of a string (e.g. C, G#, Bb, Fx or Ebb) and a boolean that should be True if the interval you're wishing to calculate is upward. Returns a 0-based number describing the degree between the two notes.
For example, **`CalculateDegree("C#", "G", False)`** would return 3.

`CalculateInterval(`*source pitch, dest pitch, upward interval*`)`**

Takes two note names in the form of a string (e.g. C, G#, Bb, Fx or Ebb) and a boolean that should be True if the interval you're wishing to calculate is upward. Returns a number representing an Interval Type (see **Global constants** on page 119). You can use the value returned in calls to **`NoteRest.Transpose`** and **`Selection.Transpose`**.
For example, **`CalculateInterval("Bb", "G#", True)`** would return IntervalAugmented.

`Close()`

Closes the current score.

`Close(`*show dialogs*`)`**

Closes the current score; if the supplied flag is **`True`** then warning dialogs may be shown about saving the active score, and if it is **`False`** then no warnings are shown (and the scores will not be saved).

`CreateProgressDialog(`*caption*`,`*min value*`,`*max value*`)`**

Creates the progress dialog, which shows a slider during a long operation.

`CreateRTFFile(`*filename***`)`**

Creates the Rich Text Format (RTF) file specified. Any existing file with the same name is destroyed. Returns True if successful.

`CreateTextFile(`*filename***`)`**

Creates the plain text file specified. Any existing file with the same name is destroyed. Returns True if successful.

`DestroyProgressDialog()`

Destroys the progress dialog.

`EnableNthControl(`*nth control, enable***`)`**

Dynamically enables or disables a given control on a plug-in dialog. Can be called either before a dialog has been displayed (in which case the operation will apply to the next dialog you show), or while a dialog is already visible (in which case the operation will affect the top-most currently visible dialog).

Note that controls can only be identified according to their order upon creation. To find out the creation order, open the appropriate dialog in the plug-in editor, right click on the dialog's client area and choose Set Creation Order from the contextual menu that appears. Note that *nth control* expects a 0-based number, unlike the display given by Set Creation Order. By default, all controls will be enabled; to disable any given control, set *enable* to false.

`FileExists(`*filename***`)`**

Returns **`True`** if a file exists or **`False`** if it doesn't.

`GetDocumentsFolder()`

Returns the user's My Documents (Windows) or Documents (Mac) folder.

`GetElapsedCentiSeconds(`*timer number***`)`**

Returns the time since **`ResetStopWatch`** was called for the given stop watch, in 100ths of a second.

`GetElapsedMilliSeconds(`*timer number***`)`**

Returns the time since **`ResetStopWatch`** was called for the given stop watch, in 1000ths of a second.

`GetElapsedSeconds(`*timer number***`)`**

Returns the time since **`ResetStopWatch`** was called for the given stop watch in seconds.

`GetFile(`*file path***`)`**

Returns a new File object representing a file path e.g. **`file=Sibelius.GetFile("c:\\onion\\foo.txt");`**

`GetFolder(`*file path***`)`**

Returns a new Folder object representing a file path e.g. **`folder=Sibelius.GetFolder("c:\");`**

`GetNotesForGuitarChord(`*chord name***`)`**

Returns a ManuScript array giving the MIDI pitches and string numbers corresponding to the named guitar chord, using the most suitable fingering according to the user's preferences. Strings are numbered starting at 0 for the bottom string and increasing upwards. The array returned has twice as many entries as the number of notes in the chord, because the pitches and string numbers are interleaved thus:

`array[0]` = *MIDI pitch for note 0*
`array[1]` = *string number for note 0*
`array[2]` = *MIDI pitch for note 1*
`array[3]` = *string number for note 1*
...

`GetScoresFolder()`

Returns a new Folder object representing the default Scores folder (as defined on the Files page of File ▸ Preferences).

`GetSyllabifier()`

Returns a new Syllabifier object, providing access to Sibelius's internal syllabification engine.

`GetUserApplicationDataFolder()`

Returns the user's Application Data (Windows) or Application Support (Mac) folder.

`GoToEnd()`

Moves the playback line to the end of the score.

`GoToStart()`

Moves the playback line to the start of the score.

`IsDynamicPartOpen(`*dynamic part*`)`**

Returns **`True`** if the specified part and its corresponding Score is valid and is visible in a Score window within Sibelius.

`MakeSafeFileName(`*filename*`)`**

Returns a "safe" version of filename. The function removes characters that are illegal on Windows or Unix, and truncates the name to 31 characters so it will be viewable on Mac OS 9.

`MessageBox(`*string*`)`**

Shows a message box with the string and an OK button.

`MoveActiveViewToBar(`*bar number[,position]*`)`**

Brings a given internal bar number into view. Has the same effect as Go to Bar in Sibelius. An optional position within the bar may also be specified, but if omitted, the very start of the bar will be brought into view.

`MoveActiveViewToSelection(`*[start of selection]*`)`**

Brings the object(s) currently selected into view. If *start of selection* is False, the end of the selection will be brought into view. If the optional argument is True or omitted, the start of the selection will be visible. Has the same effect as Shift + Home/End in Sibelius.

`New(`*[manuscript paper]*`)`**

Creates and shows a new score. If the optional parameter manuscript paper is not supplied, Sibelius will create a blank score; manuscript paper should be the filename of the manuscript paper you want to create, minus its .sib file extension. Returns the score object corresponding to the new score.

`NthScore(`*score index from 0*`)`**

Returns the *n*th open score (zero-based), or null if the specified index is not valid.

`Open(`*filename [,quiet]*`)`**

Opens and displays the given file. Filename must include its extension, e.g. Song.sib. If the optional boolean parameter *quiet* is set to **`True`**, then no error messages or dialogs will be displayed, even if the file could not be opened for some reason. Returns **`True`** if the file is opened successfully, **`False`** otherwise.

`Play()`

Plays the current score, from the current position of the playback line.

`PlayFromSelection()`

Plays from the current selection.

`PlayFromStart()`

Plays from the start of the score.

`Print(`*number of copies[, dynamic part]*`)`**

Prints the specified number of copies of the current score or dynamic part using default settings. If *number of copies* is missing or a negative number, then the default number of copies for the score or part is printed, and if set to 0 no printing occurs. The optional *dynamic part* parameter must be a valid object of the active Score (this does not affect or use **`Score.CurrentDynamicPart`** for the Score printed); if it is not supplied, the active Score is printed instead. Returns **`True`** for success, **`False`** for failure.

`PrintAllDynamicParts(`*[score]***`)`**

Prints the default number of copies of all dynamic parts, but does not print the full score. Prints the currently-active Score if the optional *score* parameter is not passed in. Returns **`True`** for success, **`False`** for failure.

`RandomNumber()`

Returns a random number.

`RandomSeed(`*start number***`)`**

Restarts the random number sequence from the given number.

`RandomSeedTime()`

Restarts the random number sequence based on the current time.

`RefreshDialog()`

Refreshes the data being displayed by any controls on the currently active plug-in dialog. For example, if a text object gets its string from a global variable and the value stored in this global variable has changed whilst the dialog is visible, calling RefreshDialog will update the text object on the dialog accordingly. Returns True if successful.

`ResetStopWatch(`*timer number***`)`**

Resets the given stop watch.

`ReadTextFile(`*filename*`,`*** *[unicode]***`)`**

Reads the given filename into an array of strings, one per line. If the *unicode* parameter is true, the file is treated as Unicode, otherwise it is treated as ANSI (i.e. 8-bit) text, which is the default. The resulting array can be used in two ways:

```
lines = Sibelius.ReadTextFile("file.txt");
for each l in lines {
  trace(l);
}
```

or:

```
lines = Sibelius.ReadTextFile("file.txt");
for i=0 to lines.NumChildren {
  trace(lines[i]);
}
```

`SelectFileToOpen(`*caption*`,`***file*`,`*** *initial_dir*`,`*** *default extension*`,`*** *default type*`,`*** *default type description***`)`**

Shows a dialog prompting the user to select a file to open. All parameters are optional. The method returns a file object describing the selection. For example:

```
file=Sibelius.SelectFileToOpen("Save Score","*.sib","c:\","sib","SIBE","Sibelius
File");
```

Note that the *initial_dir* parameter has no effect on Mac, because it is unsupported by Mac OS X.

`SelectFileToSave(`*caption*`,`***file*`,`*** *initial_dir*`,`*** *default extension*`,`*** *default type*`,`*** *default type description***`)`**

Shows a dialog prompting the user to select a file to save to. All parameters are optional. The method returns a file object describing the selection. File types and extensions:

Description	Type	Extension
EMF graphics	**`"EMF"`**	**`emf`**
Windows bitmap	**`"BMP"`**	**`bmp`**
Macintosh PICT bitmap	**`"PICT"`**	**`pict`**
Sibelius score	**`"SIBE"`**	**`sib`**
MIDI file	**`"Midi"`**	**`mid`**
House style file	**`"SIBS"`**	**`lib`**

PhotoScore file	`"SCMS"`	`opt`
Web page	`"TEXT"`	`html`
TIFF graphics	`"TIFF"`	`tif`
PNG graphics	`"PNG"`	`png`

Note that the *initial_dir* parameter has no effect on Mac, because it is unsupported by Mac OS X.

`SelectFolder(`*[caption]***`)`**

Allows the user to select a folder and returns a Folder object. The optional string parameter *caption* sets the caption of the dialog that appears.

`ShowDialog(`*script name*`,`*** *object*`)`**

Shows a dialog from a dialog description and sends messages and values to the given object. Returns the value **`True`** (**`1`**) or **`False`** (**`0`**) depending on which button you clicked to close the dialog (typically OK or Cancel).

`ShowDynamicPart(`*dynamic part[, newWindow]*`)`**

Shows the specified dynamic part. The second optional Boolean parameter overrides the state of the Open parts in new windows preference. Returns **`True`** if the specified part can be shown, **`False`** otherwise. Can be used to bring a Score to the front by way of **`Sibelius.ShowDynamicPart(Score.CurrentDynamicPart)`**.

`Stop()`

Stops the current score from playing.

`UpdateProgressDialog(`*progress pos*`,`*** *status message*`)`**

Returns **`0`** if the user clicked Cancel.

`YesNoMessageBox(`*string*`)`**

Shows a message box with Yes and No buttons. Returns **`True`** if Yes is chosen, else **`False`**.

Variables

`ActiveScore` is the active Score object (read/write). Setting **`Sibelius.ActiveScore`** makes active the current dynamic part (which may be the full score rather than a part) of the score. If that window is not currently shown, a new window may be created according to the user's preferences. Returns null if it fails to make the specified score or part active.

`ApplicationLanguage` returns the language of the version of Sibelius currently running, always in English – e.g. English, German, French etc. (read only)

`CurrentTime` returns a string containing the current time in the format hh:mm:ss, based on your own computer's locale (read only)

`CurrentDateShort` returns a string containing the current date in the format dd/mm/yyyy, based on your own computer's locale (read only)

`CurrentDateLong` returns a string containing the current date in the format dd MM yyyy, based on your own computer's locale (read only)

`CurrentDate` returns the current date and time as a DateTime object in local time (read only).

`HouseStyles` the list of house styles available, as a **`ComponentList`**

`ManuscriptPapers` the list of manuscript papers available, as a **`ComponentList`**

`OSVersionString` the current operating system in which the plug-in is running, as one of the following strings:

```
Windows 95
Windows 98
Windows ME
Windows NT 3.x
Windows NT 4
```

Windows 2000
Windows XP
Windows Vista
Windows 7
Mac OS X
Mac OS X Jaguar
Mac OS X Panther
Mac OS X Tiger
Mac OS X Leopard
Mac OS X Snow Leopard

If the operating system is unrecognized, the variable returns **Unknown system version**.

PathSeparator returns the current path separator character (i.e. "****" on Windows, "**/**" on Mac).

Plugins the list of plug-ins installed. See the documentation for the **Plugin** object

Playing is **True** if a score is currently being played (read only).

ProgramVersion the current version of Sibelius in which the plug-in is running, as an integer in the following format:

(major version) * 1000 + (minor version) * 100 + (revision) * 10

So Sibelius 3.1.3 would be returned as **3130**.

ScoreCount is the number of scores being edited (read only).

ViewHighlights is **True** if View ▸ Highlights is switched on (read/write).

ViewNoteVelocities is **True** if View ▸ Live Playback Velocities is switched on (read/write).

ViewNoteColors the current View ▸ Note Colors setting used (read/write).

Description	Value
None	**0**
Notes out of Range	**1**
Pitch Spectrum	**2**
Voice Colors	**3**

SparseArray

For more information about using sparse arrays in ManuScript, see **Sparse arrays** on page 20.

To create a sparse array, use the built-in method **`CreateSparseArray(`***a1*`,`*a2*`,`*a3*`,`*a4...an*`)`**.

Methods

`ValidIndices()`

Returns a sparse array containing only the populated indices of the original sparse array, i.e. those that are not null.

`Concat(`*array1*`,` *array2 ... arrayN*`)`**

Concatenate zero or more sparse arrays to this one, and return it as a one-level deep copy (so if a sparse array contains other arrays, for example, then the new sparse array will contain references to those arrays, not copies of them). This method does not modify the original sparse array.

`Join(`*[separator]*`)`**

Returns the array as a string, with each populated element separated by the optional *separator*. If you don't specify *separator*, the default separator is a comma.

`Push(`*value1*`,` *value2*`,` *value3 ... valueN*`)`**

Pushes one or more values to the end of the array.

`Pop()`

Returns the last element of the array, and removes it from the array.

`Reverse()`

Reverses the sparse array in place, modifying the sparse array being operated on. The reversed array only populates the elements needed to create the reversed array.

`Slice(`*start[*`,`*end]*`)`**

Returns a new sparse array of the elements starting from *start* and up to, but not including, the optional *end*. *start* and *end* can be negative indices referring to offsets from the end of the array.

Variables

`Length`	Returns or sets the length of the array (read/write).

Converting old-style arrays to new sparse arrays

The SparseArray object is a replacement for the old Array object, which was a more limited kind of array that could only hold strings and integers, but no other kind of objects. You are recommended to use the new SparseArray object for all arrays in your plug-ins, but if you have an existing plug-in in which old-style Arrays are used, you can convert them to SparseArrays as follows:

`Array.ConvertToSparseArray()` returns a new SparseArray object, populated with strings converted from the old-style Array.

SpecialBarline

Derived from a BarObject

These can only be found in system staves.

Methods

None.

Variables

`BarlineType` The name of the type of special barline, expressed as a string.

`BarlineInternalType` The type of the barline, expressed as a numeric ID which maps to one of the SpecialBarline global constants (see **Global constants** on page 119).

Staff

These can be normal staves or the system staff. The system staff contains objects that apply to all staves, such as SpecialBarlines and text using a system text style.

A Staff contains Bar objects.

`for each` *variable* **`in`** returns each object in the staff.

`for each` *type variable* in returns each item of **`type`** *type* in the staff in chronological order (i.e. in order of rhythmic position in each bar).

Methods

`AddClef(`*pos*`,`*concert pitch cleff*`[,`*transposed pitch clef*`]`**`)`**

Adds a clef to the staff at the specified position. *concert pitch clef* determines the clef style when Notes ▸ Transposing Score is switched off; the optional *transposed pitch clef* parameter determines the clef style when this is switched on. Clef styles should be an identifier like "clef.treble"; for a complete list of available clef styles, see **Clef styles** on page 122. Alternatively you can give the name of a clef style, e.g. "Treble," but bear in mind that this may not work in non-English versions of Sibelius.

`AddLine(`*pos*`,`*duration*`,`*line style*`,`*[dx*`,`*[dy*`,`*[voicenumber*`,`*[hidden]]]]*`)`**

Adds a line to staff (please see the documentation in Bar object below).

`AddNote(`*pos*`,`*sounding pitch*`,`*duration*`,`*[tied [*`,`*voice [*`,`*diatonic pitch[*`,`*string number]]]]*`)`**

Adds a note to staff, adding to an existing NoteRest if already at this position (in which case the duration is ignored); otherwise creates a new NoteRest. Will add a new bar if necessary at the end of the staff. The position is in 1/256th quarters from the start of the score. The optional *tied* parameter should be **`True`** if you want the note to be tied. Voice 1 is assumed unless the optional *voice* parameter (with a value of 1, 2, 3 or 4) is specified. You can also set the diatonic pitch, i.e. the number of the "note name" to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on). If a diatonic pitch of zero is given then a suitable diatonic pitch will be calculated from the MIDI pitch. The optional string number parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Note Input page of File ▸ Preferences). Returns the Note object created (to get the NoteRest containing the note, use **`Note.ParentNoteRest`**).

When adding very short notes to tuplets, Sibelius may be unable to find a legal place for the note in the bar. Should this happen, Sibelius will return null. You should therefore check for a valid object if there is any likelyhood that this situation may arise in your code.

N.B.: If you add a note to a score that intersects an existing tuplet, Sibelius will try to snap the note to the closest sensible place within that tuplet. However, you are advised to use *Tuplet.AddNote()* for this purpose as it is void of any ambiguity.

`AddSymbol(`*pos*`,`*symbol index or name*`)`**

Adds a symbol to staff (please see the documentation in Bar object below).

`CurrentKeySignature(`*bar number*`)`**

Returns a KeySignature valid at the bar number passed.

`NthBar(`*n*`)`**

Returns the *n*th bar in the staff, counting from 1.

`SetSound(`*styleID[*`,`*set SoundStage]*`)`**

Changes the initial playback sound of this staff to be the default sound for the given default instrument *styleID*. For a complete list of default instrument style IDs in Sibelius, see **Instrument types** on page 122. If the optional Boolean parameter is set to **`False`**, then the SoundStage information (volume, pan and distance) for this staff will be unchanged. If it is omitted or set to **`True`**, then the SoundStage information will be set to the default for the new sound.

`SetSoundID(`*soundID***`)`**

Changes the initial playback sound of this staff to the given *soundID*.

`Staff[`*array element***`]`**

Returns the *n*th bar (counting from 1) e.g. **`Staff[1]`**.

Variables

`BankHigh` Controls MIDI controller 0, used to select the "coarse" bank number for this stave, and corresponding to the Mixer control of the same name. The range is 0–127, or –1 if you don't want to send this controller message at the start of playback. Note that not all MIDI devices support multiple banks (read/write).

`BankLow` Controls MIDI controller 32, used to select the "fine" bank number for this stave, and corresponding to the Mixer control of the same name. The range is 0–127, or –1 if you don't want to send this controller message at the start of playback. Note that not all MIDI devices support multiple banks (read/write).

`BarCount` Number of bars in the staff (read only).

`Channel` The MIDI channel number of this staff, numbered 1–16 (read/write).

`Distance` The reverb "distance" of this staff, corresponding to the control of the same name in the Mixer. This is a percentage, used to scale the overall reverb settings from the Performance dialog (read/write).

`FullInstrumentName` Gives the full instrument name of the stave, empty for an unnamed stave (read/write).

`NumStavesInSameInstrument` The number of staves belonging to the default instrument from which this staff was created (read only).

`InitialClefStyle` The name of the initial clef on a staff, depending on the state of Notes ▸ Transposing Score (read only).

`InitialClefStyleId` The style identifier of the initial clef on a staff, depending on the state of Notes ▸ Transposing Score (read only).

`InitialInstrumentType` Returns an InstrumentType object for the instrument type at the start of the staff.

`InitialKeySignature` Returns the KeySignature object at the start of this staff (read only).

`InitialStyleId` Returns the style identifier of the staff (read only). To create an instrument from such an ID, pass the style as the first argument to **`Score.CreateInstrument`**. For a complete list of all the default instrument names in Sibelius, see **Instrument types** on page 122.

`InstrumentName` Gives the full instrument name of the staff in the form that is displayed on the Instruments and Staves dialog in Sibelius (read only). For an unnamed stave, this will be "[Piano]" for example, where Piano is the default instrument name of the stave (see below). To get the internal name (which will be empty for unnamed staves), use the read/write variables **`FullInstrumentName`** or **`ShortInstrumentName`** instead.

`IsSystemStaff` **`True`** or **`False`** depending on whether this staff is a system staff or not (read only).

`IsVocalStaff` Returns **`True`** if the instrument type used by the staff has the Vocal staff option switched on, meaning that e.g. the default positions of dynamics should be above the staff rather than below (read only).

`MuteMode` Specifies whether or not this stave will play back. Corresponds to the mute button in the Mixer. The supported values are defined as global constants (see **Global constants** on page 119) and are **`Muted`**, **`HalfMuted`** and **`NotMuted`** (read/write).

`Pan`	The MIDI stereo panning position of this staff (corresponding to the pan control in the Mixer). Permissible values are –100 to 100, with positive values being to the right and negative to the left (read/write).
`ParentScore`	Returns the staff's parent Score object (read only).
`ShortInstrumentName`	Gives the short instrument name of the stave, empty for an unnamed stave (read/write).
`Solo`	**`True`** or **`False`** depending on whether this staff plays back in "solo" mode, corresponding to the Mixer button of the same name (read/write).
`SoundIdOverrideIfAny`	Returns a string containing the sound ID override set in the mixer for the staff. If no override has been set, an empty string is returned (read only).
`StaffNum`	Returns the number of this stave, counting from 1 at the top of the currently-viewed part. Returns `0` for for SystemStaff objects (read only).
`Volume`	The overall MIDI volume of this staff, corresponding to its fader in the Mixer. Permissible values are 0–127 (read/write).

Syllabifier

Acts as a wrapper around Sibelius's internal Syllabification engine, exposing its functionality to ManuScript.

Methods

`AbbreviateUsingApostrophe(`*useApostrophe***`)`**

When the abbreviate flag is set to True when calling Syllabify, Sibelius will replace vowels that have been combined with the previous syllable with an apostrophe if this option is switched on – e.g. *Vege-ta-bles* vs *Veg'-ta-bles*. Calling this method will cause the syllabification engine to recalculate its result if necessary.

`GetNthSyllable(`*n***`)`**

Once a string has been syllabified by calling the **`Syllabify`** method, you can use this method to return each individual syllable as a string

`NthSyllableEndsWord(`*n***`)`**

Once a string has been syllabified by calling the **`Syllabify`** method, you can use this method to find out whether each syllable occurs at the end of a word

`Syllabify(`*textToSyllabify[, language[, abbreviate = False]]***`)`**

Breaks a string down into its syllabic components, returning the number of syllables in the resultant syllabification, or 0 if an error has occured. The rules of the specified language will be used, and you may legally supply either a language ID, or the localized language name. To get the individual syllables, you should call the **`GetNthSyllable`** and **`NthSyllableEndsWord`** methods documented below.

If the **`language`** argument is omited, Sibelius will attempt to automatically identify the language of the text. If this is not possible, or if an unrecognised language ID or name has been supplied, 0 will be returned.

When **`abbreviate`** is **`True`**, each ambiguous word in the string will be syllabified using the minimal number of syllables. For example, syllabifying "Everybody likes vegetables" would return "Eve-ry-bod-y likes vege-ta-bles" with this flag set to **`True`**, otherwise "E-ve-ry-bod-y likes veg-e-ta-bles".

Variables

`AbbreviateUsingApostrophe` Returns True/False depending on whether the syllabification engine is set to abbreviate combined syllables with an apostrophe (read only – call method with same name for write access)

`AvailableLanguageIds` Returns an array containing a list of the available syllabification languages as three-letter non-translatable IDs – e.g. ENG (English), GER (German), LAT (Latin). These IDs are identical in all localized versions of Sibelius (read only)

`AvailableLanguages` Returns an array containing a list of the available syllabification languages as localized strings (read only)

`NumberOfSyllables` Returns the number of syllables in the hyphenated string generated by calling the **`Syllabify`** method (read only)

`SyllabifiedString` Returns the resultant hyphenated string generated by calling the **`Syllabify`** method (read only)

SymbolItem and SystemSymbolItem

Derived from a BarObject. For system symbols (i.e. symbols belonging to the system staff, retrieved with **`for each`** on the system staff object), the type of symbol objects is SystemSymbolItem, not SymbolItem.

Methods

None.

Variables

`Index` — The index of this symbol in the list of symbols. This corresponds to its position in the Create ▸ Symbol dialog, counting from zero left-to-right and top-to-bottom (read only).

`Name` — The name of this symbol. May be translated in non-English language versions of Sibelius (read only).

`Size` — The draw size of the symbol, corresponding to the four available options in the Symbols dialog in Sibelius. The four available values are **`NormalSize`**, **`CueSize`**, **`GraceNoteSize`** and **`CueGraceNoteSize`**, all defined as global constants (read/write).

SystemStaff, Staff, Selection, Bar and all BarObject-derived objects

Variables

`IsALine`
Returns true if the object is a line object. (Note that this is a variable, not a method, unlike the **`IsObject()`** method for all objects.)

`Type`
A string giving the name of the type of an object. The strings for the first 4 types above are **`"SystemStave"`**, **`"Stave"`**, **`"MusicSelectionList"`**, and **`"Bar"`**. Note that this variable is also a member of all objects that occur in bars.

SystemStaff

There is one SystemStaff object per score. The SystemStaff contains objects which apply to all staves, such as SpecialBarlines and text using a system text style. Unlike normal staves, the SystemStaff does not appear in the score itself. As such, most of the variables and methods supported for Stave objects are not available on a SystemStaff. Those that are supported by SystemStaff are as follows.

Methods

`CurrentKeySignature(`*bar number***`)`**

Returns a KeySignature valid at the bar number passed.

`CurrentTimeSignature(`*bar number***`)`**

Returns a TimeSignature valid at the bar number passed.

`NthBar(`*n***`)`**

Returns the *n*th bar in the staff, counting from 1.

`SystemStaff[`*array element***`]`**

Returns the *n*th bar (counting from 1) e.g. **`SystemStaff[1]`**.

Variables

`BarCount`	Number of bars in the staff (read only).
`InitialKeySignature`	Returns the KeySignature object at the start of this staff (read only).
`IsSystemStaff`	Returns **`True`** for a SystemStaff (read only).

TextItem and SystemTextItem

Derived from a BarObject. For system text (i.e. text belonging to the system staff, retrieved with **`for each`** on the system staff object), the type of text objects is SystemTextItem, not TextItem.

Methods

None.

Variables

`StyleAsText` The text style name (read/write).

`StyleId` The identifier of the text style of this piece of text (read/write).

`Text` The text as a string (read/write).

`TextWithFormatting` Returns an array containing the various changes of font or style (if any) within the string in a new element (read only). For example, "This text is \B\bold\b\, and this is \I\italic\i\" would return an array with eight elements containing the following data:

```
arr[0] = "This text is "
arr[1] = "\B\"
arr[2] = "bold"
arr[3] = "\b\"
arr[4] = ", and this is "
arr[5] = "\I\"
arr[6] = "italic"
arr[7] = "\i\"
```

`TextWithFormattingAsString` The text including any changes of font or style (read only).

TimeSignature

Derived from a BarObject.

Methods

None

Variables

`AllowCautionary` Returns **`True`** if the time signature is set to show a cautionary at the end of the previous system, if it occurs at the start of a system (read only).

`Denominator` The time signature's bottom number (read only).

`Numerator` The time signature's top number (read only).

`Text` The time signature as text. You can use this to detect common time and alla breve time signatures by comparing it to the global constants **`CommonTimeString`** and **`AllaBreveTimeString`**, which define the Unicode characters used by these symbols. Other time signatures will be of the form "4\n4" (read only).

TreeNode

These are used internally by ManuScript to implement arrays and hashes (returned with the **CreateArray** and **CreateHash** methods), and to represent global data (defined in the plugin editor). Each TreeNode can contain a label, a piece of data and and a list of "children," which are also TreeNodes. Normally, any access to a TreeNode object will access the data that is held, so that you don't need to know anything about them, but there are also some extra variables and methods that may be useful in some circumstances. These can be called on any array, hash or global variable, and on any member of such a structure.

Methods

WriteToString

Returns a string that represents the structure of this TreeNode object. In this representation, the data of a TreeNode is surrounded by double quotes and the label is not. Note that a label need not be defined. Any children of the TreeNode (also TreeNode objects themselves) are contained within curly braces **{** and **}**. To obtain child TreeNodes, use the normal array operator, as described in the documentation for arrays and hashes.

Variables

Label	The label of this TreeNode.
NumChildren	The number of child TreeNodes belonging to this TreeNode object.

Tuplet

Derived from a BarObject.

Methods

`AddNestedTuplet(`*posInTuplet*`,` *left*`,` *right*`,` *unit*`[,` *style*`[,` *bracket*`[,`*fullDuration*`]]]]`**`)`**

Nests a new tuplet bracket within an existing tuplet at a position relative to the duration and scale-factor of the existing tuplet. The *left* and *right* parameters specify the ratio of the new tuplet, e.g. 3 (left) in the time of 2 (right). The *unit* parameter specifies the note value (in 1/256th quarters) on which the tuplet should be based. For example, if you wish to create an eighth note (quaver) triplet group, you would use the value 128. The optional *style* and *bracket* parameters take one of the pre-defined constants that affect the visual appearance of the created tuplet; see **Global constants** on page 119. If *fullDuration* is true, the bracket of the tuplet will span the entire duration of the tuplet. Returns the Tuplet object created.

NB: If **`AddNestedTuplet()`** has been given illegal parameters, it will not be able to create a valid Tuplet object. Therefore, you should test for inequality of the returned Tuplet object with *null* before attempting to use it.

`AddNote(`*posInTuplet*`,` *pitch*`,` *duration*`[,` *tied*`[,` *diatonic pitch*`[,` *string number*`]]]]`**`)`**

Adds a note to an existing tuplet, adopting the same voice number as used by the tuplet itself. Please note that *posInTuplet* is relative to the duration and scale-factor of the tuplet bracket itself. Therefore, if you wanted to add a quarter note/crotchet to the second beat of a quarter note/crotchet triplet, you would simply use the value 256 – not 341!

`utils.SplitTuplet(`*tuplet*`,`*splitpoint*`)`**

Split the tuplet object *tuplet* at the specified *splitpoint*, which is a number in relation to the tuplet's parent bar. It then splits a nest of tuplets at that point in the bar. This method is provided by the utils.plg – see **Utils** on page 111.

Variables

`Bracket`	The bracket type of the tuplet (e.g. none, auto; see **Global constants** on page 119).
`FullDuration`	True if the bracket of the tuplet spans its entire duration.
`Left`	The left side of the tuplet, e.g. 3 in 3:2 (read only).
`ParentTupletIfAny`	If the tuplet intersects a tuplet, the innermost Tuplet object at that point in the score is returned. Otherwise, *null* is returned (read only).
`PlayedDuration`	The true rhythmic duration of the tuplet e.g. for crotchet triplet this would be the duration of a minim (read only).
`PositionInTuplet`	Returns the position of the tuplet relative to the duration and scale-factor of its parent tuplet. If the tuplet does not intersect a tuplet, its position within the parent Bar is returned as usual (read only).
`Right`	The rightside of the tuplet, e.g. 2 in 3:2 (read only).
`Style`	The style of the tuplet (e.g. number, ratio, ratio + note; see **Global constants** on page 119).
`Text`	The text shown above the tuplet (read only).
`Unit`	The unit used for the tuplet, e.g. 256 for a triplet of quarter notes (read only).

Utils

Sibelius installs a plug-in called utils.plg that contains a set of useful and common methods that can be called directly by other plug-ins. It is not intended to be run as a plug-in in its own right, so does not appear in the Plug-ins menu.

The methods available via utils.plg are as follows:

`utils.AbsoluteValue(`*value*`)`**

Returns the absolute value of a number, i.e. its numerical value without regard to its sign.

`utils.AddFractions(`*x*`,`*y*`)`**

Adds two fractions *x* and *y*, passed in as ManuScript arrays. Returns an array with the result of the addition.

`utils.BinaryString(`*x*`)`**

Returns a binary string (e.g. "101010") equivalent to the number *x*.

`utils.bwAND(`*x*`, `*y*`)`**

Equivalent to the C++ bitwise AND (**`&`**) operator. For example, **`utils.bwAND(129,1)`** is equal to **`1`**.

`utils.bwOR(`*x*`, `*y*`)`**

Equivalent to the C++ bitwise inclusive OR (**`|`**) operator. For example, **`utils.bwOR(64,4)`** is equal to **`68`**.

`utils.bwXOR(`*x*`, `*y*`)`**

Equivalent to the C++ bitwise exclusive XOR (**`^`**) operator. For example, **`utils.bwXOR(4,6)`** is equal to **`2`**.

`utils.CapableOfDeletion()`

Returns **`True`** if the object can be deleted using **`Delete()`**, which is determined by checking Sibelius's version number.

`utils.CaseInsensitiveComparison(`*s1*`, `*s2*`)`**

Returns **`True`** if the two strings *s1* and *s2* match, ignoring case.

`utils.CastToBool(`*x*`)`**

Returns the variable *x* explicitly cast as a Boolean.

`utils.CastToInt(`*x*`)`**

Returns the variable *x* explicitly cast as an integer.

`utils.CastToStr(`*x*`)`**

Returns the variable *x* explicitly cast as a string.

`utils.CombineArraysOfBool(`*arr1*`, `*arr2*`)`**

Concatenates two arrays containing Boolean values and returns the result.

`utils.CombineArraysOfInt(`*arr1*`, `*arr2*`)`**

Concatenates two arrays containing integral values and returns the result.

`utils.CombineArraysOfString(`*arr1*`, `*arr2*`)`**

Concatenates two arrays containing string values and returns the result.

`utils.CopyTextFile(`*source*`, `*dest*`)`**

Copies an existing text file from one location to another, returning True if successful.

`utils.CreateArrayBlanket(`*value*`, `*size*`)`**

Returns an array with *size* elements, each containing a blanket value specified by the first parameter.

`utils.DeleteStaff(`*score, nth staff, retain selection*`)`**

Deletes an entire staff and its content from a given score, returning True if successful. If *retain selection* is True, Sibelius will ensure any item(s) that were selected prior to the staff's deletion are still selected.

utils.DenaryValue(*x***)**

Returns a number in base 10 equivalent to binary number *x*, which must be provided as a string.

utils.DivideFractions(*x***,***y***)**

Divides fraction *x* by fraction *y*, passed in as ManuScript arrays. Returns an array with the result of the division.

utils.ExactFileName(*filename***)**

Returns just the filename portion of a string *filename* containing both a path and a filename.

utils.Format(*str***,** *[val1***,***val2***,***val3 ...]***)**

Provides a simple means of replacing human-readable data types in a string. Each successive instance of **%s** in *str* is replaced with the value of the next remaining unused argument. e.g. **s = utils.Format("The %s brown %s jumps %s the lazy %s", "quick", "fox", "over", "dog");**

utils.FormatTime(*ms***)**

Formats a time, given in milliseconds, to a human-readable string using the format mm'ss.z (where z is centiseconds).

utils.FractionAsDecimal(*x***)**

Returns the decimal equivalent of the fraction *x*, which is passed in as an array.

utils.FractionDenominator(*x***)**

Returns the denominator of fraction *x*, which is passed in as an array.

utils.FractionNumerator(*x***)**

Returns the numerator of fraction *x*, which is passed in as an array.

utils.GetAppDir()

Returns the path of the Sibelius executable as a string.

utils.GetArrayIndex(*arr***,** *value***)**

Returns the index of *value* in the array *arr*, or **-1** if it doesn't exist in the array.

utils.GetBits(*x***)**

Returns an array containing the list of powers of two whose cumulative sum equates to the value of *x*.

utils.GetGlobalApplicationDataDir()

Returns the path of the system's global application data area as a string.

utils.GetLocationTime(*score, barNum, position***)**

Returns the precise time (in milliseconds) of a given location in a score. The position should be local to the start of the bar number you have supplied. Use the utils library to achieve this if your plug-in needs to be backwards compatible with Sibelius 4; otherwise call the Score object's function with the same name.

utils.GetMillisecondsFromTime(*time***)**

If you pass in a time expressed in milliseconds (e.g. one minute being 60,000), this function returns the milliseconds portion of the number (in this case 60,000 modulus 1000 = 0).

utils.GetMinutesFromTime(*time***)**

If you pass in a time expressed in milliseconds, this function returns the minutes portion of the number (e.g. if *time* = 120,262 milliseconds, this function returns 2).

utils.GetObjectTime(*score***,***obj***)**

Returns the precise time (in milliseconds) that the object *obj* occurs from the start of a given score, taking into account tempo changes, performance markings and any other events in the score that have an effect on playback. Use this method to achieve

this if your plug-in needs to be backwards compatible with Sibelius 4; otherwise use the Time property of the BarObject object whose time you wish to determine.

`utils.GetPluginId(`*plug-in***`)`**

This enables you to identify a plug-in by entering the line of code **`PluginUniqueID = "`***someUniqueId***`";`** in a plug-in's **`Initialize`** method. When you pass a plug-in object to this function, it scans the plug-in's code and returns its unique ID if it has one, otherwise an empty string.

`utils.GetSibeliusPluginsFolder()`

This is a wrapper around the deprecated **`GetPluginsFolder()`** function, and returns the path of the Plugins folder.

`utils.GetSibMajorVersion()`

Returns the major version number of Sibelius.

`utils.GreatestCommonDivisor(`*m***`,`***n***`)`**

Returns the greatest common divisor of two non-zero integers, i.e. the largest positive integer that divides both numbers without remainder.

`utils.IsInArray(`*arr***`,` *value***`)`**

Returns **`True`** if *value* exists in the array *arr*.

`utils.IsNumeric(`*str***`)`**

Returns **`True`** if the string is numeric.

`utils.LowerCase(`*str***`)`**

Returns the ANSI string *str* in lowercase.

`utils.MakeFraction(`*x***`,`***y***`)`**

Creates a fraction with *x* as the numerator and *y* as the denominator. The fraction is returned as a normal ManuScript array. (Manipulating fractions means you never have to worry about rounding errors.)

`utils.max(`*x***`,` *y***`)`**

Returns the greater of two numbers.

`utils.min(`*x***`,` *y***`)`**

Returns the lesser of two numbers.

`utils.MultiplyFractions(`*x***`,`***y***`)`**

Multiplies fraction *y* by fraction *x*, passed in as ManuScript arrays. Returns an array with the result of the multiplication.

`utils.PatternCount(`*pattern***`,`***str***`)`**

Returns the number of times the substring *pattern* exists in *str*.

`utils.Pos(`*subStr***`,`***str***`)`**

Returns the zero-based position of the first instance of the sub-string *subStr* in *str*, or `-1` if it isn't found.

`utils.PosReverse(`*subStr***`,`***str***`)`**

Returns the zero-based position of the *last* instance of the sub-string *subStr* in *str*, or `-1` if it isn't found.

`utils.Power(`*x***`,`***y***`)`**

Raises *x* to the *y*th power, where *y* is a positive integer.

`utils.Replace(`*inStr***`,` *toFind*`,` *replaceWith*`,` *replaceAll***`)`**

Replaces a sub-string in a string with a new value. It looks for *toFind* in the string *inStr*, and if it finds it, replaces it with *replaceWith*. If the Boolean *replaceAll* is **`False`**, it only changes the first instance found; if it's **`True`**, it replaces all instances.

`utils.ReverseArrayOfBool(`*arr***`)`**

Reverses the order of the elements in an array of Booleans.

utils.ReverseArrayOfInt(*arr***)**

Reverses the order of the elements in an array of integers.

utils.ReverseArrayOfString(*arr***)**

Reverses the order of the elements in an array of strings.

utils.SetDefaultIfNotInArray(*value***, *arr*, *DefaultIndex*)**

Scans the array *arr* for the value specified by the first parameter. *Value* is returned if it exists in the array, otherwise, *arr*[*DefaultIndex*].

utils.shl(*x***, *y*)**

Bitwise left-shift. Shifts the value *x* left by *y* bits. Equivalent to C++ **<<** operator.

utils.shr(*x***, *y*)**

Bitwise right-shift. Shifts the value *x* right by *y* bits. Equivalent to C++ **>>** operator.

utils.SortArray(*arr***)**

Sorts the array *arr* using a case-insensitive alphabetic sort.

utils.SortArrayCustom(*arr***,*method*)**

Sorts the array *arr* using a custom sort order routine, which must be passed into this method.

utils.SortArrayNumeric(*arr***)**

Sorts the array *arr* in ascending numeric order.

utils.SplitTuplet(*tuplet***,*splitpoint*)**

Split the tuplet object *tuplet* at the specified *splitpoint*, which is a number in relation to the tuplet's parent bar. It then splits a nest of tuplets at that point in the bar.

utils.StartComponentManager(*componentName***,*callbackFunc*)**

Returns an array of filenames (strings) found on the system inside a folder with a given name, following the same rules of precedence as Sibelius's internal component manager. Files in the user's application data area take priority over those in the global application data area, followed lastly by those in the Sibelius's application directory itself.

callbackFunc should point to a function in the calling script that scans a supplied directory for files with a specific extension. Such a function might look something like this:

```
GetFooFiles(dir) { // This is the function signature
   components = CreateArray();
   for each FOO file in dir {
      components[components.NumChildren] = file.NameWithExt;
   }
   return(components);
}
```

In the scenario above, the call to start the component manager would look like this (where "Foo Files" is the name of the directory containing your files):

```
files = utils.StartComponentManager("Foo Files","myPlugin.GetFooFiles");
```

utils.SubtractFractions(*x***,*y*)**

Subtracts fraction *y* from fraction *x*, passed in as ManuScript arrays. Returns an array with the result of the subtraction.

utils.UpperCase(*str***)**

Returns the ANSI string *str* in uppercase.

VersionHistory

Each Score object has a VersionHistory object (obtained by way of the *score*`.GetVersions()` method), which in turn provides a list of Version objects. Each Version object represents a specific version, and also provides a list of VersionComment objects, which represent the per-version comments (as opposed to bar-attached comments, which are represented to ManuScript as Comment objects, derived from BarObject objects).

Methods

`AddVersion(`*[name[`,`comment]]*`)`**

Adds a new version object and returns it if successful (or null if not), with an optional *name* and *comment* for the version.

`DeleteNthVersion(`*n*`)`**

Deletes the *n*th Version object, returning `True` if successful.

`GetNthVersion(`*n*`)`**

Returns the *n*th Version object.

Variables

`NumChildren`	Returns the number of versions in the score's VersionHistory object.

Version

Accessed via a Score object's VersionHistory object.

Methods

`Close()`

Closes all views of the version that are currently open in Sibelius, returning True if it has actually closed anything.

`DeleteNthComment(`*n***`)`**

Deletes the *n*th comment, returning True if successful.

`OpenAndReturnScore()`

Opens the specified version in Sibelius (if it's not already open) and returns its Score object.

Variables

`EndDate`	Returns a DateTime object representing the version's end date (read only).
`IsOpen`	Returns True if the version is currently open in Sibelius (read only).
`Name`	Returns the name of the version (read/write).
`NumComments`	Returns the number of comments in the version (read only).
`StartDate`	Returns a DateTime object representing the version's start date (read only).

VersionComment

Accessed via Version objects.

Methods

None.

Variables

`Text`	Returns or changes the text of the comment, and this cannot be undone (read/write).
`TimeStamp`	Returns a DateTime object representing the time at which the comment was created.
`UserName`	Returns the name of the user who created the comment (read only).

Global constants

Global constants

These are useful variables held internally within ManuScript and are accessible from any plug-in. They are called "constants" because you are encouraged not to change them.

Many of the constants are the names of note values, which you can use to specify a position in a bar easily. So instead of writing **`320`** you can write **`Quarter+Sixteenth`** or equally **`Crotchet+Semiquaver`**.

Truth values

True	1
False	0

Measurements

Space	32
StaffHeight	128

Positions and durations

Long	4096	Sixteenth	64
Breve	2048	Semiquaver	64
DottedBreve	3072	DottedSixteenth	96
Whole	1024	DottedSemiquaver	96
Semibreve	1024	ThirtySecond	32
DottedWhole	1536	Demisemiquaver	32
Half	512	DottedThirtySecond	48
Minim	512	DottedDemisemiquaver	48
DottedHalf	768	SixtyFourth	16
DottedMinim	768	Hemidemisemiquaver	16
Quarter	256	DottedSixtyFourth	24
Crotchet	256	DottedHemidemisemiquaver	24
DottedQuarter	384	OneHundredTwentyEighth	8
DottedCrotchet	384	Semihemidemisemiquaver	8
Eighth	128	DottedOneHundredTwentyEighth	12
Quaver	128	DottedSemihemidemisemiquaver	12
DottedEighth	192		
DottedQuaver	192		

Style names

For the **`ApplyStyle()`** method of Score objects. Instead of the capitalized strings in quotes, you can use the equivalent variables in mixed upper and lower case. Note again that the constant **`HOUSE`** refers to the options in House Style ▸ Engraving Rules and Layout ▸ Document Setup only; to apply the entire House Style, use the **`ALLSTYLES`** constant.

House	"HOUSE"	Dictionary	"DICTIONARY"
Text	"TEXT"	SpacingRule	"SPACINGRULE"
Symbols	"SYMBOLS"	CustomChordNames	"CUSTOMCHORDNAMES"
Lines	"LINES"	DefaultPartAppearance	"DEFAULTPARTAPPEARANCE"
Noteheads	"NOTEHEADS"	InstrumentsAndEnsembles	"INSTRUMENTSANDENSEMBLES"
Clefs	"CLEFS"	AllStyles	"ALLSTYLES"

Bar number formats

These constants can be used for the *format* argument of the **`AddBarNumber`** method.

BarNumberFormatNormal	0

BarNumberFormatNumberLetterLower	1
BarNumberFormatNumberLetterUpper	2

Line styles

Highlight	"line.highlight"
Arpeggio	"line.staff.arpeggio"
Arpeggio down	"line.staff.arpeggio.down"
Arpeggio up	"line.staff.arpeggio.up"
Unused 2	"line.staff.arrow"
Beam	"line.staff.bend"
Box	"line.staff.box"
Glissando (straight)	"line.staff.gliss.straight"
Glissando (wavy)	"line.staff.gliss.wavy"
Crescendo	"line.staff.hairpin.crescendo"
Diminuendo	"line.staff.hairpin.diminuendo"
2 octaves down	"line.staff.octava.minus15"
Octave down	"line.staff.octava.minus8"
2 octaves up	"line.staff.octava.plus15"
Octave up	"line.staff.octava.plus8"
Pedal	"line.staff.pedal"
Line	"line.staff.plain"
Slur below	"line.staff.slur.down"
Slur above	"line.staff.slur.up"
Tie	"line.staff.tie"
Trill	"line.staff.trill"
Dashed system line	"line.system.dashed"
Wide dashed system line	"line.system.dashed.wide"
1st ending	"line.system.repeat.1st"
1st and 2nd ending	"line.system.repeat.1st_n_2nd"
2nd ending	"line.system.repeat.2nd"
2nd ending (closed)	"line.system.repeat.2nd.closed"
3rd ending	"line.system.repeat.3rd"
Repeat ending (closed)	"line.system.repeat.closed"
Repeat ending (open)	"line.system.repeat.open"
Accel.	"line.system.tempo.accel"
Accel. (italic)	"line.system.tempo.accel.italic"
Accel. (italic, text only)	"line.system.tempo.accel.italic.
Molto accel.	"line.system.tempo.accel.molto"
Molto accel. (text only)	"line.system.tempo.accel.molto.t
Poco accel.	"line.system.tempo.accel.poco"
Poco accel. (text only)	"line.system.tempo.accel.poco.textonly"
Accel. (text only)	"line.system.tempo.accel.textonly"
Tempo change (arrow right)	"line.system.tempo.arrowright"
Rall.	"line.system.tempo.rall"
Rall. (italic)	"line.system.tempo.rall.italic"
Rall. (italic, text only)	"line.system.tempo.rall.italic.textonly"
Molto rall.	"line.system.tempo.rall.molto"
Molto rall. (text only)	"line.system.tempo.rall.molto.textonly"
Poco rall.	"line.system.tempo.rall.poco"
Poco rall. (text only)	"line.system.tempo.rall.poco.textonly"
Rall. (text only)	"line.system.tempo.rall.textonly"
Rit.	"line.system.tempo.rit"
Rit. (italic)	"line.system.tempo.rit.italic"
Rit. (italic, text only)	"line.system.tempo.rit.italic.textonly"
Molto rit.	"line.system.tempo.rit.molto"
Molto rit. (text only)	"line.system.tempo.rit.molto.textonly"
Poco rit.	"line.system.tempo.rit.poco"
Poco rit. (text only)	"line.system.tempo.rit.poco.textonly"
Rit. (text only)	"line.system.tempo.rit.textonly"

Text styles

Here is a list of all the text style identifiers which are guaranteed to be present in any score in Sibelius. In previous versions of ManuScript text styles were identified by a numeric index; this usage has been deprecated but will continue to work for old plug-ins. New plug-ins should use the identifiers given below. For each style we first give the English name of the style and then the identifier.

Instrument names	"text.instrumentname"
1st and 2nd endings	"text.staff.1st_n_2nd_endings"
Auto page break warnings	"text.staff.autopagebreak.warnings"
Boxed text	"text.staff.boxed"
Expression	"text.staff.expression"
Chord diagram fingering	"text.staff.fingering.chord_diagrams"
Footnote	"text.staff.footnote"
Block lyrics	"text.staff.lyrics.block"
Multirests (tacet)	"text.staff.multirests.tacet"
Plain text	"text.staff.plain"
Small text	"text.staff.small"
Chord symbol	"text.staff.space.chordsymbol"
Time signatures (one staff only)	"text.staff.timesig.onestaffonly"
Tuplets	"text.staff.tuplets"
Bar numbers	"text.system.barnumber"
Metronome mark	"text.system.metronome"
Multirests (numbers)	"text.system.multirestnumbers"
Composer	"text.system.page_aligned.composer"
Composer (on title page)	"text.system.page_aligned.composer.ontitlepage"
Copyright	"text.system.page_aligned.copyright"
Dedication	"text.system.page_aligned.dedication"
Footer (inside edge)	"text.system.page_aligned.footer.inside"
Footer (outside edge)	"text.system.page_aligned.footer.outside"
Worksheet footer (first page, l)	"text.system.page_aligned.footer.worksheet.left"

Figured bass	"text.staff.space.figuredbass"
Fingering	"text.staff.space.fingering"
Chord diagram fret	"text.staff.space.fretnumbers"
Lyrics above staff	"text.staff.space.hypen.lyrics.above"
Lyrics (chorus)	"text.staff.space.hypen.lyrics.chorus"
Lyrics line 1	"text.staff.space.hypen.lyrics.verse1"
Lyrics line 2	"text.staff.space.hypen.lyrics.verse2"
Lyrics line 3	"text.staff.space.hypen.lyrics.verse3"
Lyrics line 4	"text.staff.space.hypen.lyrics.verse4"
Lyrics line 5	"text.staff.space.hypen.lyrics.verse5"
Nashville chord numbers	"text.staff.space.nashvillechords"
Common symbols	"text.staff.symbol.common"
Figured bass (extras)	"text.staff.symbol.figured.bass.extras"
Note tails	"text.staff.symbol.noteflags"
Special noteheads etc.	"text.staff.symbol.noteheads.special"
Percussion instruments	"text.staff.symbol.percussion"
Special symbols	"text.staff.symbol.special"
Tablature letters	"text.staff.tab.letters"
Tablature numbers	"text.staff.tab.numbers"
Technique	"text.staff.technique"

Header	"text.system.page_aligned.header"
Worksheet header (first page, l)	"text.system.page_aligned.header.worksheet.left"
Worksheet header (first page, r)	"text.system.page_aligned.header.worksheet.right"
Header (after first page)	"text.system.page_aligned.header_notp1"
Header (after first page, inside edge)	"text.system.page_aligned.header_notp1.inside"
Instrument name at top left	"text.system.page_aligned.instrnametopleft"
Lyricist	"text.system.page_aligned.lyricist"
Page numbers	"text.system.page_aligned.pagenumber"
Subtitle	"text.system.page_aligned.subtitle"
Title	"text.system.page_aligned.title"
Title (on title page)	"text.system.page_aligned.title.ontitlepage"
Rehearsal mark	"text.system.rehearsalmarks"
Repeat (D.C./D.S./To Coda)	"text.system.repeat"
Tempo	"text.system.tempo"
Timecode	"text.system.timecode"
Duration at end of score	"text.system.timecode.duration"
Hit points	"text.system.timecode.hitpoints"
Time signatures (huge)	"text.system.timesig.huge"
Time signatures (large)	"text.system.timesig.large"
Time signatures	"text.system.timesig.normal"

Clef styles

Here is a list of all the clef style identifiers that are guaranteed to be present in any score in Sibelius, for use with the **`Stave.AddClef`** method. For each style we first give the English name of the style, and then the identifier.

Alto	"clef.alto"
Baritone C	"clef.baritone.c"
Baritone F	"clef.baritone.f"
Bass	"clef.bass"
Bass down 8	"clef.bass.down.8"
Bass up 15	“clef.bass.up.15"
Bass up 8	"clef.bass.up.8"
Null	"clef.null"
Percussion	"clef.percussion"
Percussion 2	"clef.percussion_2"
Soprano	"clef.soprano"
Mezzo-soprano	"clef.soprano.mezzo"
Tab	"clef.tab"

Small tab	"clef.tab.small"
Small tab (taller)	"clef.tab.small.taller"
Tab (taller)	"clef.tab.taller"
Tenor	"clef.tenor"
Tenor down 8	"clef.tenor.down.8"
Treble	"clef.treble"
Treble down 8	"clef.treble.down.8"
Treble (down 8)	"clef.treble.down.8.bracketed"
Treble down 8 (old)	"clef.treble.down.8.old"
Treble up 15	"clef.treble.up.15"
Treble up 8	"clef.treble.up.8"
French violin	"clef.violin.french"

Instrument types

Here is a list of all the instrument type identifiers that are guaranteed to be present in any score in Sibelius. For each style we first give the English name of the style and then the identifier. Note that only the tablature stave types can be used with guitar frames; the rest are included for completeness.

Alp-Horn in F	instrument.brass.alp-horn.f
Alp-Horn in G	instrument.brass.alp-horn.g
Baritone Bugle in G	instrument.brass.bugle.baritone.g
Contrabass Bugle in G	instrument.brass.bugle.contrabass.g
Euphonium Bugle in G	instrument.brass.bugle.euphonium.g
Mellophone Bugle in G	instrument.brass.bugle.mellophone.g
Soprano Bugle in G	instrument.brass.bugle.soprano.g
Cimbasso in Bb	instrument.brass.cimbasso.bflat

Cimbasso in Eb	instrument.brass.cimbasso.eflat
Cimbasso in F	instrument.brass.cimbasso.f
Cornet in A	instrument.brass.cornet.a
Cornet in Bb	instrument.brass.cornet.bflat
Soprano Cornet in Eb	instrument.brass.cornet.soprano.eflat
Euphonium in Bb [treble clef]	instrument.brass.euphonium
Euphonium in Bb [bass clef, treble transp.]	instrument.brass.euphonium.bassclef
Euphonium in C [bass clef]	instrument.brass.euphonium.bassclef.bassclef
Euphonium in Bb [bass clef]	instrument.brass.euphonium.bflat.bassclef.bassclef
Flugelhorn	instrument.brass.flugelhorn
Horn in A [no key]	instrument.brass.horn.a.nokeysig
Horn in Ab alto [no key]	instrument.brass.horn.alto.aflat.nokeysig
Alto Horn in Eb	instrument.brass.horn.alto.eflat
Alto Horn in F	instrument.brass.horn.alto.f
Horn in B [no key]	instrument.brass.horn.b.nokeysig
Baritone in Bb [treble clef]	instrument.brass.horn.baritone
Baritone in C [treble clef]	instrument.brass.horn.baritone.2
Baritone in Bb [bass clef, treble transp.]	instrument.brass.horn.baritone.bassclef
Baritone in C [bass clef]	instrument.brass.horn.baritone.bassclef.bassclef
Bass in Bb	instrument.brass.horn.bass.bflat
Bass in Bb [bass clef, treble transp.]	instrument.brass.horn.bass.bflat.bassclef
Bass in C	instrument.brass.horn.bass.c
Bass in Eb	instrument.brass.horn.bass.eflat
Bass in Eb [bass clef, treble transp.]	instrument.brass.horn.bass.eflat.bassclef
A Basso Horn [no key]	instrument.brass.horn.basso.a.nokeysig
Bb Basso Horn [no key]	instrument.brass.horn.basso.bflat.nokeysig
C Basso Horn [no key]	instrument.brass.horn.basso.c.nokeysig
Horn in Bb [no key]	instrument.brass.horn.bflat.nokeysig
Horn in C [no key]	instrument.brass.horn.c.nokeysig
Horn in D [no key]	instrument.brass.horn.d.nokeysig
Horn in Db [no key]	instrument.brass.horn.dflat.nokeysig
Horn in E [no key]	instrument.brass.horn.e.nokeysig
Horn in Eb	instrument.brass.horn.eflat
Horn in Eb [no key]	instrument.brass.horn.eflat.nokeysig
Horn in F	instrument.brass.horn.f
Horn in F [bass clef]	instrument.brass.horn.f.bassclef
Horn in F [no key]	instrument.brass.horn.f.nokeysig
Horn in F# [no key]	instrument.brass.horn.fsharp.nokeysig
Horn in G [no key]	instrument.brass.horn.g.nokeysig
Tenor Horn	instrument.brass.horn.tenor
Mellophone in Eb	instrument.brass.mellophone.eflat
Mellophone in F	instrument.brass.mellophone.f
Mellophonium in Eb	instrument.brass.mellophonium.eflat
Mellophonium in F	instrument.brass.mellophonium.f
Ophicleide	instrument.brass.ophicleide
Brass	instrument.brass.section
Serpent	instrument.brass.serpent
Sousaphone in Bb	instrument.brass.sousaphone.bflat
Sousaphone in Eb	instrument.brass.sousaphone.eflat
Trombone	instrument.brass.trombone

Alto Trombone	instrument.brass.trombone.alto
Bass Trombone	instrument.brass.trombone.bass
Trombone in Bb [bass clef, treble transp.]	instrument.brass.trombone.bassclef.trebleclef
Contrabass Trombone	instrument.brass.trombone.contrabass
Tenor Trombone	instrument.brass.trombone.tenor
Trombone in Bb [treble clef]	instrument.brass.trombone.trebleclef
Trumpet in A	instrument.brass.trumpet.a
Trumpet in B [no key]	instrument.brass.trumpet.b.nokeysig
Bass Trumpet in Bb	instrument.brass.trumpet.bass.bflat
Bass Trumpet in Eb	instrument.brass.trumpet.bass.eflat
Trumpet in Bb	instrument.brass.trumpet.bflat
Trumpet in Bb [no key]	instrument.brass.trumpet.bflat.nokeysig
Trumpet in C	instrument.brass.trumpet.c
Trumpet in D	instrument.brass.trumpet.d
Trumpet in Db	instrument.brass.trumpet.dflat
Trumpet in E [no key]	instrument.brass.trumpet.e.nokeysig
Trumpet in Eb	instrument.brass.trumpet.eflat
Trumpet in F	instrument.brass.trumpet.f
Trumpet in G [no key]	instrument.brass.trumpet.g.nokeysig
Piccolo Trumpet in A	instrument.brass.trumpet.piccolo.a
Piccolo Trumpet in Bb	instrument.brass.trumpet.piccolo.bflat
Tenor Trumpet in Eb	instrument.brass.trumpet.tenor.eflat
Tuba	instrument.brass.tuba
Tuba in F	instrument.brass.tuba.f
Tenor Tuba (Wagner, in Bb)	instrument.brass.tuba.tenor
Tenor Tuba [bass clef]	instrument.brass.tuba.tenor.bassclef
Wagner Tuba in Bb	instrument.brass.tuba.wagner.bflat
Wagner Tuba in F	instrument.brass.tuba.wagner.f
Applause	instrument.exotic.applause
Birdsong	instrument.exotic.birdsong
Helicopter	instrument.exotic.helicopter
Ondes Martenot	instrument.exotic.ondes-martenot
Sampler	instrument.exotic.sampler
Seashore	instrument.exotic.seashore
Tape	instrument.exotic.tape
Telephone	instrument.exotic.telephone
Theremin	instrument.exotic.theremin
Bajo [notation]	instrument.fretted.bajo.5lines
Bajo, 6-string [tab]	instrument.fretted.bajo.tab
Bajo, 4-string [tab]	instrument.fretted.bajo.tab.4lines
Bajo, 5-string [tab]	instrument.fretted.bajo.tab.5lines
Alto Balalaika [notation]	instrument.fretted.balalaika.alto.5lines
Alto Balalaika [tab]	instrument.fretted.balalaika.alto.tab
Bass Balalaika [notation]	instrument.fretted.balalaika.bass.5lines
Bass Balalaika [tab]	instrument.fretted.balalaika.bass.tab
Contrabass Balalaika [notation]	instrument.fretted.balalaika.contrabass.5lines
Contrabass Balalaika [tab]	instrument.fretted.balalaika.contrabass.tab
Prima Balalaika [notation]	instrument.fretted.balalaika.prima.5lines
Prima Balalaika [tab]	instrument.fretted.balalaika.prima.tab
Second Balalaika [notation]	instrument.fretted.balalaika.second.5lines

Second Balalaika [tab]	instrument.fretted.balalaika.second.tab
Bandola [notation]	instrument.fretted.bandola.5lines
Bandola [tab]	instrument.fretted.bandola.tab
Bandolón [notation]	instrument.fretted.bandolon.5lines
Bandolón [tab]	instrument.fretted.bandolon.tab
Bandurria [notation]	instrument.fretted.bandurria.5lines
Bandurria [tab]	instrument.fretted.bandurria.tab
Banjo [notation]	instrument.fretted.banjo.5lines
Banjo (aDADE tuning) [tab]	instrument.fretted.banjo.aDADE.tab
Banjo (aEADE tuning) [tab]	instrument.fretted.banjo.aEADE.tab
Banjo (gCGBD tuning) [tab]	instrument.fretted.banjo.gCGBD.tab
Banjo (gCGCD tuning) [tab]	instrument.fretted.banjo.gCGCD.tab
Banjo (gDF#AD tuning) [tab]	instrument.fretted.banjo.gDFAD.tab
Banjo (gDGBD tuning) [tab]	instrument.fretted.banjo.gDGBD.tab
Banjo (gDGCD tuning) [tab]	instrument.fretted.banjo.gDGCD.tab
Tenor Banjo [notation]	instrument.fretted.banjo.tenor.5lines
Tenor Banjo [tab]	instrument.fretted.banjo.tenor.tab
Bordonúa [notation]	instrument.fretted.bordonua.5lines
Bordonúa [tab]	instrument.fretted.bordonua.tab
Cavaquinho [notation]	instrument.fretted.cavaquinho.5lines
Cavaquinho [tab]	instrument.fretted.cavaquinho.tab
Charango [notation]	instrument.fretted.charango.5lines
Charango [tab]	instrument.fretted.charango.tab
Cuatro [notation]	instrument.fretted.cuatro.5lines
Cuatro, Puerto Rico [tab]	instrument.fretted.cuatro.puerto-rico.tab
Cuatro, Venezuela [tab]	instrument.fretted.cuatro.venezuela.tab
Resonator guitar [notation]	instrument.fretted.guitar.resonator.5lines
Resonator Guitar, A6 tuning [tab]	instrument.fretted.guitar.resonator.a6.tab
Resonator Guitar, B11 tuning [tab]	instrument.fretted.guitar.resonator.b11.tab
Resonator Guitar, C#m tuning [tab]	instrument.fretted.guitar.resonator.c#m.tab
Resonator Guitar, C6+A7 tuning [tab]	instrument.fretted.guitar.resonator.c6-a7.tab
Resonator Guitar, C6 + high G tuning [tab]	instrument.fretted.guitar.resonator.c6-highg.tab
Resonator Guitar, standard tuning [tab]	instrument.fretted.guitar.resonator.c6.tab
Resonator Guitar, C#m7 tuning [tab]	instrument.fretted.guitar.resonator.cm7.tab
Resonator Guitar, E13 Hawaiian tuning [tab]	instrument.fretted.guitar.resonator.e13-hawaiian.tab
Resonator Guitar, E13 Western tuning [tab]	instrument.fretted.guitar.resonator.e13-western.tab
Resonator Guitar, open A tuning [tab]	instrument.fretted.guitar.resonator.open.A.tab
Resonator Guitar, open G tuning [tab]	instrument.fretted.guitar.resonator.open.G.tab
Dulcimer	instrument.fretted.dulcimer
Dulcimer [notation]	instrument.fretted.dulcimer.5lines
Dulcimer (DAA tuning) [tab]	instrument.fretted.dulcimer.daa.tab
Dulcimer (DAD tuning) [tab]	instrument.fretted.dulcimer.dad.tab
Gamba [notation]	instrument.fretted.gamba.5lines
Gamba [tab]	instrument.fretted.gamba.tab
12-string Acoustic Guitar [notation]	instrument.fretted.guitar.12-string.5lines
12-string Acoustic Guitar, DADGAD tuning [tab]	instrument.fretted.guitar.12-string.dadgad.tab
12-string Acoustic Guitar, double D tuning [tab]	instrument.fretted.guitar.12-string.double-d.tab
12-string Acoustic Guitar, dropped D tuning [tab]	instrument.fretted.guitar.12-string.dropped-d.tab
12-string Acoustic Guitar, open D tuning [tab]	instrument.fretted.guitar.12-string.open-d.tab
12-string Acoustic Guitar, open E tuning [tab]	instrument.fretted.guitar.12-string.open-e.tab

12-string Acoustic Guitar, open G tuning [tab]	instrument.fretted.guitar.12-string.open-g.tab
12-string Acoustic Guitar, standard tuning (no rhythms) [tab]	instrument.fretted.guitar.12-string.tab
12-string Acoustic Guitar, standard tuning [tab]	instrument.fretted.guitar.12-string.tab.rhythms
Acoustic Guitar [notation]	instrument.fretted.guitar.acoustic.5lines
Acoustic Guitar, DADGAD tuning [tab]	instrument.fretted.guitar.acoustic.dadgad.tab
Acoustic Guitar, double D tuning [tab]	instrument.fretted.guitar.acoustic.double-d.tab
Acoustic Guitar, dropped D tuning [tab]	instrument.fretted.guitar.acoustic.dropped-d.tab
Acoustic Guitar, modal D tuning [tab]	instrument.fretted.guitar.acoustic.modal-d.tab
Acoustic Guitar, Nashville tuning [tab]	instrument.fretted.guitar.acoustic.nashville.tab
Acoustic Guitar, open A tuning [tab]	instrument.fretted.guitar.acoustic.open-a.tab
Acoustic Guitar, open C tuning [tab]	instrument.fretted.guitar.acoustic.open-c.tab
Acoustic Guitar, open D tuning [tab]	instrument.fretted.guitar.acoustic.open-d.tab
Acoustic Guitar, open Dm cross-note tuning [tab]	instrument.fretted.guitar.acoustic.open-dm.tab
Acoustic Guitar, open E tuning [tab]	instrument.fretted.guitar.acoustic.open-e.tab
Acoustic Guitar, open G tuning [tab]	instrument.fretted.guitar.acoustic.open-g.tab
Acoustic Guitar, standard tuning (no rhythms) [tab]	instrument.fretted.guitar.acoustic.tab
Acoustic Guitar, standard tuning [tab]	instrument.fretted.guitar.acoustic.tab.rhythms
4-string Bass Guitar [notation]	instrument.fretted.guitar.bass.4-string.5lines
4-string Bass Guitar [tab]	instrument.fretted.guitar.bass.4-string.tab
5-string Bass Guitar [notation]	instrument.fretted.guitar.bass.5-string.5lines
5-string Bass Guitar [tab]	instrument.fretted.guitar.bass.5-string.tab
Bass Guitar [notation]	instrument.fretted.guitar.bass.5lines
6-string Bass Guitar [notation]	instrument.fretted.guitar.bass.6-string.5lines
6-string Bass Guitar [tab]	instrument.fretted.guitar.bass.6-string.tab
Acoustic Bass [notation]	instrument.fretted.guitar.bass.acoustic.5lines
Acoustic Bass [tab]	instrument.fretted.guitar.bass.acoustic.tab
5-string Electric Bass [notation]	instrument.fretted.guitar.bass.electric.5-string.5lines
5-string Electric Bass [tab]	instrument.fretted.guitar.bass.electric.5-string.tab
Electric Bass [notation]	instrument.fretted.guitar.bass.electric.5lines
6-string Electric Bass [notation]	instrument.fretted.guitar.bass.electric.6-string.5lines
6-string Electric Bass [tab]	instrument.fretted.guitar.bass.electric.6-string.tab
5-string Fretless Electric Bass	instrument.fretted.guitar.bass.electric.fretless.5-string.5lines
5-string Fretless Electric Bass [tab]	instrument.fretted.guitar.bass.electric.fretless.5-string.tab
Fretless Electric Bass [notation]	instrument.fretted.guitar.bass.electric.fretless.5lines
6-string Fretless Electric Bass	instrument.fretted.guitar.bass.electric.fretless.6-string.5lines
6-string Fretless Electric Bass [tab]	instrument.fretted.guitar.bass.electric.fretless.6-string.tab
Fretless Electric Bass [tab]	instrument.fretted.guitar.bass.electric.fretless.tab
Electric Bass [tab]	instrument.fretted.guitar.bass.electric.tab
5-string Fretless Bass Guitar [notation]	instrument.fretted.guitar.bass.fretless.5-string.5lines
5-string Fretless Bass Guitar [tab]	instrument.fretted.guitar.bass.fretless.5-string.tab
Fretless Bass Guitar [notation]	instrument.fretted.guitar.bass.fretless.5lines
6-string Fretless Bass Guitar [notation]	instrument.fretted.guitar.bass.fretless.6-string.5lines
6-string Fretless Bass Guitar [tab]	instrument.fretted.guitar.bass.fretless.6-string.tab
Fretless Bass Guitar [tab]	instrument.fretted.guitar.bass.fretless.tab
Semi-Acoustic Bass [notation]	instrument.fretted.guitar.bass.semi-acoustic.5lines
Semi-Acoustic Bass [tab]	instrument.fretted.guitar.bass.semi-acoustic.tab
Bass Guitar [tab]	instrument.fretted.guitar.bass.tab
Bass Guitar [tab, with rhythms]	instrument.fretted.guitar.bass.tab.rhythms
Classical Guitar [notation]	instrument.fretted.guitar.classical.5lines
Classical Guitar, DADGAD tuning [tab]	instrument.fretted.guitar.classical.dadgad.tab

Classical Guitar, double D tuning [tab]	instrument.fretted.guitar.classical.double-d.tab
Classical Guitar, dropped D tuning [tab]	instrument.fretted.guitar.classical.dropped-d.tab
Classical Guitar, open D tuning [tab]	instrument.fretted.guitar.classical.open-d.tab
Classical Guitar, open E tuning [tab]	instrument.fretted.guitar.classical.open-e.tab
Classical Guitar, open G tuning [tab]	instrument.fretted.guitar.classical.open-g.tab
Classical Guitar, standard tuning (no rhythms) [tab]	instrument.fretted.guitar.classical.tab
Classical Guitar, standard tuning [tab]	instrument.fretted.guitar.classical.tab.rhythms
Electric Guitar [notation]	instrument.fretted.guitar.electric.5lines
7-string Electric Guitar, low A tuning [tab]	instrument.fretted.guitar.electric.7-string.low-a.tab
7-string Electric Guitar, low B tuning [tab]	instrument.fretted.guitar.electric.7-string.tab
Electric Guitar, DADGAD tuning [tab]	instrument.fretted.guitar.electric.dadgad.tab
Electric Guitar, double D tuning [tab]	instrument.fretted.guitar.electric.double-d.tab
Electric Guitar, dropped D tuning [tab]	instrument.fretted.guitar.electric.dropped-d.tab
Electric Guitar, open D tuning [tab]	instrument.fretted.guitar.electric.open-d.tab
Electric Guitar, open E tuning [tab]	instrument.fretted.guitar.electric.open-e.tab
Electric Guitar, open G tuning [tab]	instrument.fretted.guitar.electric.open-g.tab
Electric Guitar, standard tuning (no rhythms) [tab]	instrument.fretted.guitar.electric.tab
Electric Guitar, standard tuning [tab]	instrument.fretted.guitar.electric.tab.rhythms
Kora	instrument.fretted.guitar.kora
Semi-acoustic Guitar [notation]	instrument.fretted.guitar.semi-acoustic.5lines
Semi-acoustic Guitar, DADGAD tuning [tab]	instrument.fretted.guitar.semi-acoustic.dadgad.tab
Semi-acoustic Guitar, double D tuning [tab]	instrument.fretted.guitar.semi-acoustic.double-d.tab
Semi-acoustic Guitar, dropped D tuning [tab]	instrument.fretted.guitar.semi-acoustic.dropped-d.tab
Semi-acoustic Guitar, open D tuning [tab]	instrument.fretted.guitar.semi-acoustic.open-d.tab
Semi-acoustic Guitar, open E tuning [tab]	instrument.fretted.guitar.semi-acoustic.open-e.tab
Semi-acoustic Guitar, open G tuning [tab]	instrument.fretted.guitar.semi-acoustic.open-g.tab
Semi-acoustic Guitar, standard tuning (no rhythms) [tab]	instrument.fretted.guitar.semi-acoustic.tab
Semi-acoustic Guitar, standard tuning [tab]	instrument.fretted.guitar.semi-acoustic.tab.rhythms
10-string Hawaiian Steel Guitar [tab]	instrument.fretted.guitar.steel.hawaiian.10-string.tab
Hawaiian Steel Guitar [notation]	instrument.fretted.guitar.steel.hawaiian.5lines
6-string Hawaiian Steel Guitar, standard tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab
6-string Hawaiian Steel Guitar, alternate tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.alternative
6-string Hawaiian Steel Guitar, slack key Bb Mauna Loa tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.bflat.mauna.loa
6-string Hawaiian Steel Guitar, slack key C Mauna Loa tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.c.mauna.loa
6-string Hawaiian Steel Guitar, slack key Wahine CGDGBD tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.cgdgbd.wahine
6-string Hawaiian Steel Guitar, slack key Wahine CGDGBE tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.cgdgbe.wahine
6-string Hawaiian Steel Guitar, slack key Wahine DGDF#BD tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.dgdfbd.wahine
6-string Hawaiian Steel Guitar, slack key G Mauna Loa tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.g.mauna.loa
6-string Hawaiian Steel Guitar, slack key G Taro Patch tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.g.taro.patch
6-string Hawaiian Steel Guitar, slack key Wahine GCDGBE tuning [tab]	instrument.fretted.guitar.steel.hawaiian.6-string.tab.gcdgbe.wahine
8-string Hawaiian Steel Guitar [tab]	instrument.fretted.guitar.steel.hawaiian.8-string.tab
8-string Hawaiian Steel Guitar, alternate tuning [tab]	instrument.fretted.guitar.steel.hawaiian.8-string.tab.alternative
Hawaiian Steel Guitar [tab]	instrument.fretted.guitar.steel.hawaiian.tab
Pedal Steel Guitar [notation]	instrument.fretted.guitar.steel.pedal.5lines
Pedal Steel Guitar [tab]	instrument.fretted.guitar.steel.pedal.tab
Guitarra [notation]	instrument.fretted.guitarra.5lines
Guitarra, Coimbra [tab]	instrument.fretted.guitarra.coimbra.tab
Guitarra, Lisboa [tab]	instrument.fretted.guitarra.lisboa.tab
Guitarra, Portuguesa [tab]	instrument.fretted.guitarra.portuguesa.tab
Guitarrón [notation]	instrument.fretted.guitarron.5lines

Guitarrón [tab]	instrument.fretted.guitarron.tab
Laúd [notation]	instrument.fretted.laud.5lines
Laúd [tab]	instrument.fretted.laud.tab
Tenor Lute [notation]	instrument.fretted.lute.5lines
Bass Lute [notation]	instrument.fretted.lute.bass-d.french.english.5lines
Bass Lute, D tuning, French/English [tab]	instrument.fretted.lute.bass-d.french.english.tab
Bass Lute, D tuning, Italian [tab]	instrument.fretted.lute.bass-d.italian.tab
Bass Lute, D tuning, Spanish [tab]	instrument.fretted.lute.bass-d.spanish.tab
Tenor Lute, G tuning, Italian [tab]	instrument.fretted.lute.italian.tab
Tenor Lute, G tuning, Spanish [tab]	instrument.fretted.lute.spanish.tab
Tenor Lute, G tuning, French/English [tab]	instrument.fretted.lute.tab
Tenor Lute, A tuning, French/English [tab]	instrument.fretted.lute.tenor-a.french.english.tab
Tenor Lute, A tuning, Italian [tab]	instrument.fretted.lute.tenor-a.italian.tab
Tenor Lute, A tuning, Spanish [tab]	instrument.fretted.lute.tenor-a.spanish.tab
Treble Lute [notation]	instrument.fretted.lute.treble-d.french.english.5lines
Treble Lute, D tuning, French/English [tab]	instrument.fretted.lute.treble-d.french.english.tab
Treble Lute, D tuning, Italian [tab]	instrument.fretted.lute.treble-d.italian.tab
Treble Lute, D tuning, Spanish [tab]	instrument.fretted.lute.treble-d.spanish.tab
Mandolin [notation]	instrument.fretted.mandolin.5lines
Mandolin [tab]	instrument.fretted.mandolin.tab
Oud [notation]	instrument.fretted.oud.5lines
Oud [tab]	instrument.fretted.oud.tab
Qanoon	instrument.fretted.qanoon.5lines
Requinto [notation]	instrument.fretted.requinto.5lines
Requinto [tab]	instrument.fretted.requinto.tab
Santoor	instrument.fretted.santoor.5lines
Sitar [notation]	instrument.fretted.sitar.5lines
Sitar (Ravi Shankar) [tab]	instrument.fretted.sitar.ravi-shankkar.tab
Sitar (Vilayat Khan) [tab]	instrument.fretted.sitar.vilayat-khan.tab
Tambura (Female) [notation]	instrument.fretted.tambura.female
Tambura (Male) [notation]	instrument.fretted.tambura.male
Tiple [notation]	instrument.fretted.tiple.5lines
Tiple, Argentina [tab]	instrument.fretted.tiple.argentina.tab
Tiple, Colombia ADF#B tuning [tab]	instrument.fretted.tiple.colombia.tab.adfb
Tiple, Colombia DGBE tuning [tab]	instrument.fretted.tiple.colombia.tab.dgbe
Tiple, Cuba [tab]	instrument.fretted.tiple.cuba.tab
Tiple, Peru [tab]	instrument.fretted.tiple.peru.tab
Tiple, Santo Domingo [tab]	instrument.fretted.tiple.santo.domingo.tab
Tiple, Uruguay [tab]	instrument.fretted.tiple.uruguay.tab
Tres [notation]	instrument.fretted.tres.5lines
Tres, GCE tuning [tab]	instrument.fretted.tres.tab
Tres, ADF# tuning [tab]	instrument.fretted.tres.tab.adf
Tres, GBE tuning [tab]	instrument.fretted.tres.tab.gbe
Ukulele [notation]	instrument.fretted.ukulele.5lines
Ukulele [tab]	instrument.fretted.ukulele.tab
Vihuela [notation]	instrument.fretted.vihuela.5lines
Vihuela [tab]	instrument.fretted.vihuela.tab
Zither	instrument.fretted.zither
Keyboard	instrument.keyboard
Accordion	instrument.keyboard.accordion

Bandoneon	instrument.keyboard.bandoneon
Celesta	instrument.keyboard.celesta
Clavichord	instrument.keyboard.clavichord
Harmonium	instrument.keyboard.harmonium
Harpsichord	instrument.keyboard.harpsichord
Keyboards	instrument.keyboard.keyboards
Tape Sampler Keyboard [Brass]	instrument.keyboard.tape sampler.brass
Tape Sampler Keyboard [Choir]	instrument.keyboard.tape sampler.choir
Tape Sampler Keyboard [Flute]	instrument.keyboard.tape sampler.flute
Tape Sampler Keyboard [Strings]	instrument.keyboard.tape sampler
Melodeon	instrument.keyboard.melodeon
Electric Organ	instrument.keyboard.organ.electric
Organ [manuals]	instrument.keyboard.organ.manuals
Manual [solo organ manuals]	instrument.keyboard.organ.manuals.solo
Ped. [Organ pedals]	instrument.keyboard.organ.pedals
Pedal [solo organ pedals]	instrument.keyboard.organ.pedals.solo
Piano	instrument.keyboard.piano
Electric Piano	instrument.keyboard.piano.electric
Electric Clavichord	instrument.keyboard.piano.electric.clavichord
Electric Stage Piano	instrument.keyboard.piano.electric.stage
Overdriven Electric Piano	instrument.keyboard.piano.electric.overdriven
Honky-tonk Piano	instrument.keyboard.piano.honky-tonk
Synthesizer	instrument.keyboard.synthesizer
Unnamed (2 lines)	instrument.other.2lines
Unnamed (3 lines)	instrument.other.3lines
Unnamed (4 lines)	instrument.other.4lines
Unnamed (bass staff)	instrument.other.bassclef
No instrument (barlines shown)	instrument.other.none.barlines
No instrument (bar rests shown)	instrument.other.none.barrests
No instrument (hidden)	instrument.other.none.hidden
Solo	instrument.other.solo.real
Unnamed (treble staff)	instrument.other.trebleclef
Almglocken	instrument.pitchedpercussion.almglocken
Antique Cymbals	instrument.pitchedpercussion.antiquecymbals
Chimes	instrument.pitchedpercussion.bells.chimes
Chimes [no key]	instrument.pitchedpercussion.bells.chimes.nokeysig
Bell lyre [marching band]	instrument.pitchedpercussion.bells.marching
Orchestral Bells	instrument.pitchedpercussion.bells.orchestral
Tubular Bells	instrument.pitchedpercussion.bells.tubular
Cimbalom	instrument.pitchedpercussion.cimbalom
Crotales	instrument.pitchedpercussion.crotales
Steel Drums	instrument.pitchedpercussion.drums.steel
Steel Drums [bass clef, treble transp.]	instrument.pitchedpercussion.drums.steel.bassclef
Gamelan Kengong	instrument.pitchedpercussion.gamelan.kengong
Gamelan Slentam	instrument.pitchedpercussion.gamelan.slentam
Glockenspiel	instrument.pitchedpercussion.glockenspiel
Alto Glockenspiel	instrument.pitchedpercussion.glockenspiel.alto
Soprano Glockenspiel	instrument.pitchedpercussion.glockenspiel.soprano
Handbells	instrument.pitchedpercussion.handbells
Harp	instrument.pitchedpercussion.harp

Lever Harp	instrument.pitchedpercussion.harp.lever
Kalimba	instrument.pitchedpercussion.kalimba
Marimba [grand staff]	instrument.pitchedpercussion.marimba
Marimba [treble staff]	instrument.pitchedpercussion.marimba.trebleclef
Alto Metallophone	instrument.pitchedpercussion.metallophone.alto
Bass Metallophone	instrument.pitchedpercussion.metallophone.bass
Soprano Metallophone	instrument.pitchedpercussion.metallophone.soprano
Roto-toms	instrument.pitchedpercussion.roto-toms
Temple Blocks	instrument.pitchedpercussion.templeblocks
Timpani [with key]	instrument.pitchedpercussion.timpani
Timpani [no key]	instrument.pitchedpercussion.timpani.nokeysig
Vibraphone	instrument.pitchedpercussion.vibraphone
Wood Blocks [5 lines]	instrument.pitchedpercussion.woodblocks
Xylophone	instrument.pitchedpercussion.xylophone
Alto Xylophone	instrument.pitchedpercussion.xylophone.alto
Bass Xylophone	instrument.pitchedpercussion.xylophone.bass
Contra Bass Bar	instrument.pitchedpercussion.xylophone.contrabass.bar
Gyil	instrument.pitchedpercussion.xylophone.gyil
Soprano Xylophone	instrument.pitchedpercussion.xylophone.soprano
Xylorimba	instrument.pitchedpercussion.xylorimba
Alto	instrument.singers.alto
Solo Alto	instrument.singers.alto.solo
Altus	instrument.singers.altus
Baritone	instrument.singers.baritone
Solo Baritone	instrument.singers.baritone.solo
Bass	instrument.singers.bass
Solo Bass	instrument.singers.bass.solo
Bassus	instrument.singers.bassus
Cantus	instrument.singers.cantus
Choir	instrument.singers.choir
Contralto	instrument.singers.contralto
Countertenor	instrument.singers.counter-tenor
Mean	instrument.singers.mean
Mezzo-soprano	instrument.singers.mezzo-soprano
Quintus	instrument.singers.quintus
Secundus	instrument.singers.secundus
Soprano	instrument.singers.soprano
Solo Soprano	instrument.singers.soprano.solo
Tenor	instrument.singers.tenor
Solo Tenor	instrument.singers.tenor.solo
Treble	instrument.singers.treble
Solo Treble	instrument.singers.treble.solo
Voice	instrument.singers.voice
Voice [male]	instrument.singers.voice.male
Contrabass	instrument.strings.contrabass
Bass [Double]	instrument.strings.contrabass.bass
Double Bass	instrument.strings.contrabass.double-bass
Solo Contrabass	instrument.strings.contrabass.solo
String Bass	instrument.strings.contrabass.string
Upright Bass	instrument.strings.contrabass.upright

Hurdy-gurdy	instrument.strings.hurdy-gurdy
Sarangi	instrument.strings.sarangi
Strings	instrument.strings.section
Strings [reduction]	instrument.strings.section.reduction
Bass Viol	instrument.strings.viol.bass
Tenor Viol	instrument.strings.viol.tenor
Treble Viol	instrument.strings.viol.treble
Viola	instrument.strings.viola
Solo Viola	instrument.strings.viola.solo
Violin 1	instrument.strings.violin.1
Violin 2	instrument.strings.violin.2
Violin I	instrument.strings.violin.I
Violin II	instrument.strings.violin.ii
Solo Violin	instrument.strings.violin.solo
Violoncello	instrument.strings.violoncello
Solo Violoncello	instrument.strings.violoncello.solo
Anvil	instrument.unpitched.anvil
Cha-cha bell [1 line]	instrument.unpitched.bells.cha-cha
Mambo bell [1 line]	instrument.unpitched.bells.mambo
Sleigh Bells	instrument.unpitched.bells.sleigh
Brake Drum [1 line]	instrument.unpitched.brake-drum.1line
Cabasa [1 line]	instrument.unpitched.cabasa
Cabasa [2 lines]	instrument.unpitched.cabasa.2lines
Castanets	instrument.unpitched.castanets
Shaker, Caxixi [1 line]	instrument.unpitched.caxixi.1line
Claves [1 line]	instrument.unpitched.claves
Shaker, Cocoa Bean Rattle [1 line]	instrument.unpitched.cocoa bean.1line
Finger Cymbals [1 line]	instrument.unpitched.cymbals.finger.1line
Percussion [1 line]	instrument.unpitched.drums.1line
Percussion [2 lines]	instrument.unpitched.drums.2lines
Berimbau	instrument.unpitched.drums.2lines.berimbau
Percussion [3 lines]	instrument.unpitched.drums.3lines
Percussion [4 lines]	instrument.unpitched.drums.4lines
Percussion [5 lines]	instrument.unpitched.drums.5lines
Agogos [2 lines]	instrument.unpitched.drums.agogos
Bass Drum	instrument.unpitched.drums.bass
Bass Drum [5 lines]	instrument.unpitched.drums.bass.5lines
Marching Bass Drum [3 lines]	instrument.unpitched.drums.bass.marching.3lines
Marching Bass Drum [5 lines]	instrument.unpitched.drums.bass.marching.5lines
Itótele [Batá Drum]	instrument.unpitched.drums.bata.itotele
Iyá [Batá Drum]	instrument.unpitched.drums.bata.iya
Okónkolo [Batá Drum]	instrument.unpitched.drums.bata.okonkolo
Bongos [2 lines]	instrument.unpitched.drums.bongos
Bongo Bell [High]	instrument.unpitched.drums.bongos.bell.high
Bongo Bell [Low]	instrument.unpitched.drums.bongos.bell.low
Box	instrument.unpitched.drums.box.3lines
Cajon [2 lines]	instrument.unpitched.drums.cajon
Congas [2 lines]	instrument.unpitched.drums.congas
Congas [1 line]	instrument.unpitched.drums.congas.1line
Congas [3 lines]	instrument.unpitched.drums.congas.3lines

Congas [4 lines]	instrument.unpitched.drums.congas.4lines
Cuíca [3 lines]	instrument.unpitched.drums.cuica.3lines
Cymbals	instrument.unpitched.drums.cymbal
Marching Cymbals [5 lines]	instrument.unpitched.drums.cymbals.marching.5lines
Djembe [3 lines]	instrument.unpitched.drums.djembe.3lines
Drum Set (Rock)	instrument.unpitched.drums.drumset
Drum Set (Alternative)	instrument.unpitched.drums.drumset.alternative
Drum Set (Brushes)	instrument.unpitched.drums.drumset.brushes
Drum Set (Dance)	instrument.unpitched.drums.drumset.dance
Drum Set (Disco)	instrument.unpitched.drums.drumset.disco
Drum Set (Electronica)	instrument.unpitched.drums.drumset.electronic
Drum Set (Fusion)	instrument.unpitched.drums.drumset.fusion
Drum Set (Garage)	instrument.unpitched.drums.drumset.garage
Drum Set (Hip-hop)	instrument.unpitched.drums.drumset.hip-hop
Drum Set (Industrial)	instrument.unpitched.drums.drumset.industrial
Drum Set (Jazz)	instrument.unpitched.drums.drumset.jazz
Drum Set (Lo-Fi)	instrument.unpitched.drums.drumset.lo-fi
Drum Set (Metal)	instrument.unpitched.drums.drumset.metal
Drum Set (Motown)	instrument.unpitched.drums.drumset.motown
Drum Set (New Age)	instrument.unpitched.drums.drumset.new age
Drum Set (Pop)	instrument.unpitched.drums.drumset.pop
Drum Set (Reggae)	instrument.unpitched.drums.drumset.reggae
Drum Set (Stadium Rock)	instrument.unpitched.drums.drumset.rock.stadium
Drum Set (Rods)	instrument.unpitched.drums.drumset.rods
Drum Set (Drum Machine)	instrument.unpitched.drums.drumset.tr-808
Dumbek [3 lines]	instrument.unpitched.drums.dumbek.3lines
Kidi [Ewe Drum]	instrument.unpitched.drums.ewe.kidi
Sogo [Ewe Drum]	instrument.unpitched.drums.ewe.sogo
Gankokwe (Bell)	instrument.unpitched.drums.gankokwe
Jam Blocks [2 lines]	instrument.unpitched.drums.jamblocks
Jawbone [1 line]	instrument.unpitched.drums.jawbone.1line
Pandeiro [2 lines]	instrument.unpitched.drums.pandeiro
Rain Stick (High) [1 line]	instrument.unpitched.drums.rainstick.high.1line
Rain Stick (Low) [1 line]	instrument.unpitched.drums.rainstick.low.1line
Egg Shaker (High) [1 line]	instrument.unpitched.drums.shaker.high.1line
Egg Shaker (Low) [1 line]	instrument.unpitched.drums.shaker.low.1line
Egg Shaker (Medium) [1 line]	instrument.unpitched.drums.shaker.medium.1line
Side Drum	instrument.unpitched.drums.side
Snare Drum	instrument.unpitched.drums.snare
Marching Snare Drums [5 lines]	instrument.unpitched.drums.snare.5lines
Surdo [2 lines]	instrument.unpitched.drums.surdo
Tabla	instrument.unpitched.drums.table
Taiko Drum	instrument.unpitched.drums.taiko
Tenor Drum	instrument.unpitched.drums.tenor
Marching Tenor Drums [5 lines]	instrument.unpitched.drums.tenor.marching
Quads [5 lines]	instrument.unpitched.drums.tenor.marching.quads
Tom-toms [5 lines]	instrument.unpitched.drums.tom-toms
Tom-toms [4 lines]	instrument.unpitched.drums.tom-toms.4lines
Udu	instrument.unpitched.drums.udu
Shaker, Egg Shaker [1 line]	instrument.unpitched.egg shaker.1line

Finger Click [1 line]	instrument.unpitched.fingerclick
Gamelan Gong Ageng (High) [1 line]	instrument.unpitched.gamelan.gong-ageng.high
Gamelan Gong Ageng (Low) [1 line]	instrument.unpitched.gamelan.gong-ageng.low
Gamelan Kempyang and Ketuk [2 lines]	instrument.unpitched.gamelan.kempyang-ketuk
Gamelan Khendang Ageng [1 line]	instrument.unpitched.gamelan.khendang-ageng
Gamelan Khendang Ciblon [1 line]	instrument.unpitched.gamelan.khendang-ciblon
Large Gong [1 line]	instrument.unpitched.gong.large.1line
Medium Gong [1 line]	instrument.unpitched.gong.medium.1line
Gourd [1 line]	instrument.unpitched.gourd
Guira [1 line]	instrument.unpitched.guira
Guiro (High) [1 line]	instrument.unpitched.guiro.high
Guiro (Medium) [1 line]	instrument.unpitched.guiro.medium
Handclap [1 line]	instrument.unpitched.handclap
Shaker, Kayamba [1 line]	instrument.unpitched.kayamba.1line
Maracas	instrument.unpitched.maracas
Shaker, Gourd Maracas [1 line]	instrument.unpitched.maracas.gourd.1line
Maracas [High]	instrument.unpitched.maracas.high
Maracas [Medium]	instrument.unpitched.maracas.medium
Mark tree [1 line]	instrument.unpitched.marktree
Shaker, Nsak Rattle [1 line]	instrument.unpitched.nsak.1line
Finger Snaps	instrument.unpitched.orff.fingersnaps
Hand Claps	instrument.unpitched.orff.handclaps
Patsch	instrument.unpitched.orff.patsch
Stamp	instrument.unpitched.orff.stamp
Salsa bell [1 line]	instrument.unpitched.salsa.bell
Shaker [1 line]	instrument.unpitched.shaker
Shaker, Shekere [1 line]	instrument.unpitched.shekere.1line
Tam-tam	instrument.unpitched.tam-tam
Tambourine	instrument.unpitched.tambourine
Timbales [2 lines]	instrument.unpitched.timbales.2lines
Timbales [5 lines]	instrument.unpitched.timbales.5lines
Triangle	instrument.unpitched.triangle
Shaker, Wasembe Rattle (High) [1 line]	instrument.unpitched.wasembe.high.1line
Shaker, Wasembe Rattle (Low) [1 line]	instrument.unpitched.wasembe.low.1line
Shaker, Wasembe Rattle (Medium) [1 line]	instrument.unpitched.wasembe.medium.1line
Whip	instrument.unpitched.whip
Whistle	instrument.unpitched.whistle
Wind Chimes [1 line]	instrument.unpitched.wind-chimes.1line
Wood Block [1 line]	instrument.unpitched.woodblock.1line
Bagpipes	instrument.wind.bagpipe
Basset Horn	instrument.wind.basset-horn
Bassoon	instrument.wind.bassoon
Contrabassoon	instrument.wind.bassoon.contrabassoon
Quart Bassoon	instrument.wind.bassoon.quart
Quint Bassoon	instrument.wind.bassoon.quint
Clarinet in A	instrument.wind.clarinet.a
Clarinet in Ab	instrument.wind.clarinet.aflat
Alto Clarinet in Eb	instrument.wind.clarinet.alto.eflat
Alto Clarinet in Eb [bass clef, treble transp.]	instrument.wind.clarinet.alto.eflat.bassclef
Bass Clarinet in Bb	instrument.wind.clarinet.bass.bflat

Bass Clarinet in Bb [score sounds 8vb]	instrument.wind.clarinet.bass.bflat.8vb-score
Bass Clarinet in Bb [bass clef, treble transp.]	instrument.wind.clarinet.bass.bflat.bassclef
Clarinet in Bb	instrument.wind.clarinet.bflat
Clarinet in C	instrument.wind.clarinet.c
Contra Alto Clarinet in Eb	instrument.wind.clarinet.contra.alto.eflat
Contra Alto Clarinet in Eb [score sounds 8vb]	instrument.wind.clarinet.contra.alto.eflat.8vb-score
Contra Alto Clarinet in Eb [bass clef, treble transp.]	instrument.wind.clarinet.contra.alto.eflat.bassclef
Contrabass Clarinet in Bb	instrument.wind.clarinet.contrabass.bflat
Contrabass Clarinet in Bb [score sounds 15mb]	instrument.wind.clarinet.contrabass.bflat.15mb-score
Contrabass Clarinet in Bb [bass clef, treble transp.]	instrument.wind.clarinet.contrabass.bflat.bassclef
Clarinet in D	instrument.wind.clarinet.d
Clarinet in Eb	instrument.wind.clarinet.eflat
Clarinet in G	instrument.wind.clarinet.g
Cor Anglais	instrument.wind.coranglais
Didgeridoo	instrument.wind.didgeridoo
Duduk	instrument.wind.duduk
English Horn	instrument.wind.englishhorn
Flageolet	instrument.wind.flageolet
Flute	instrument.wind.flute
Alto Flute	instrument.wind.flute.alto
Bansuri	instrument.wind.flute.bansuri
Bass Flute	instrument.wind.flute.bass
Eb Flute	instrument.wind.flute.eflat
G Flute	instrument.wind.flute.g
Harmonica	instrument.wind.harmonica
Heckelphone	instrument.wind.heckelphone
Mey	instrument.wind.mey
Nai	instrument.wind.nai
Oboe	instrument.wind.oboe
Baritone Oboe	instrument.wind.oboe.baritone
Bass Oboe	instrument.wind.oboe.bass
Oboe d'Amore	instrument.wind.oboe.damore
Ocarina	instrument.wind.ocarina
Panpipes	instrument.wind.panpipes
Piccolo	instrument.wind.piccolo
Military Piccolo in Db	instrument.wind.piccolo.dflat
Alto Recorder	instrument.wind.recorder.alto
Bass Recorder	instrument.wind.recorder.bass
Great Bass Recorder	instrument.wind.recorder.bass.great
Contrabass Recorder	instrument.wind.recorder.contrabass
Descant Recorder	instrument.wind.recorder.descant
Sopranino Recorder	instrument.wind.recorder.sopranino
Soprano Recorder	instrument.wind.recorder.soprano
Tenor Recorder	instrument.wind.recorder.tenor
Treble Recorder	instrument.wind.recorder.treble
Alto Saxophone	instrument.wind.saxophone.alto
Baritone Saxophone	instrument.wind.saxophone.baritone
Baritone Saxophone [score sounds 8vb]	instrument.wind.saxophone.baritone.8vb-score
Baritone Saxophone [bass clef, treble transp.]	instrument.wind.saxophone.baritone.bassclef
Bass Saxophone	instrument.wind.saxophone.bass

Bass Saxophone [score sounds 15mb]	instrument.wind.saxophone.bass.15mb-score
Bass Saxophone [bass clef, treble transp.]	instrument.wind.saxophone.bass.bassclef
C Melody Saxophone	instrument.wind.saxophone.c-melody
Contrabass (Tubax) Saxophone	instrument.wind.saxophone.contrabass
Contrabass (Tubax) Saxophone [score sounds 15mb]	instrument.wind.saxophone.contrabass.15mb-score
Contrabass (Tubax) Sax [bass clef, treble transp.]	instrument.wind.saxophone.contrabass.bassclef
F Mezzo Soprano Saxophone	instrument.wind.saxophone.mezz-soprano.f
Sopranino Saxophone	instrument.wind.saxophone.sopranino
Piccolo Saxophone in Bb [Soprillo]	instrument.wind.saxophone.sopranino.bflat
Soprano Saxophone	instrument.wind.saxophone.soprano
C Soprano Saxophone	instrument.wind.saxophone.soprano.c
Subcontrabass (Tubax) Saxophone	instrument.wind.saxophone.subcontrabass
Subcontrabass (Tubax) Saxophone [score sounds 15mb]	instrument.wind.saxophone.subcontrabass.15mb-score
Subcontrabass (Tubax) Sax [bass clef, treble transp.]	instrument.wind.saxophone.subcontrabass.bassclef
Tenor Saxophone	instrument.wind.saxophone.tenor
Tenor Saxophone [score sounds 8vb]	instrument.wind.saxophone.tenor.8vb-score
Tenor Saxophone [bass clef, treble transp.]	instrument.wind.saxophone.tenor.bassclef
Woodwind	instrument.wind.section
Shakuhachi	instrument.wind.shakuhachi
Tin Whistle	instrument.wind.whistle.tin

Beam options

For the **`Beam`** variable of NoteRest objects.

NoBeam	1
StartBeam	2
ContinueBeam	3
SingleBeam	4

Breaks

These constants are used by the **`SetBreakType()`** method of Score objects.

MiddleOfSystem	1
EndOfSystem	2
MiddleOfPage	3
EndOfPage	4
NotEndOfSystem	5
EndOfSystemOrPage	6
Default	7
SpecialPageBreak	8

These constants correspond to the menu entries in the Bars panel of the Properties window in the following way:

`MiddleOfSystem`	Middle of system. The bar can only appear in the middle of a system, not at the end.
`EndOfSystem`	No menu entry; created by Layout ▸ Lock Format. The bar can only appear at the end of a mid-page system, not the middle of a system or the end of a page.
`MiddleOfPage`	Middle of page. The bar can appear anywhere except at the end of a page.
`EndOfPage`	Page break. The bar can only appear at the end of a page.
`NotEndOfSystem`	No menu entry. The bar can appear anywhere except the end of a mid-page system.
`EndOfSystemOrPage`	System break. The bar can only appear at the end of a mid-page system or the end of a page.
`Default`	No break. The bar can appear anywhere.

Note that in older versions of ManuScript the constant **`MiddleOfSystem`** was called **`NoBreak`** and the constant **`EndOfSystem`** was called **`SystemBreak`**. These older names were confusing, because they implied a correlation with the similarly-named menu items in the Properties window that was not accurate. The old names are still supported for old plug-ins, but should not be used for new plug-ins. For consistency, the old constant **`PageBreak`** has also been renamed **`EndOfPage`**, even though this did correlate correctly with the Properties window.

Accidentals

For the **`Accidental`** variable of Note objects.

DoubleSharp	2
Sharp	1
Natural	0
Flat	–1
DoubleFlat	–2

Note Style names

For the **`NoteStyle`** variable of Note objects; these correspond to the noteheads available from the **Notes** panel of the Properties window in the manuscript papers that are supplied with Sibelius.

NormalNoteStyle	0	BackSlashedNoteStyle	12
CrossNoteStyle	1	ArrowDownNoteStyle	13
DiamondNoteStyle	2	ArrowUpNoteStyle	14
BeatWithoutStemNoteStyle	3	InvertedTriangleNoteStyle	15
BeatNoteStyle	4	ShapedNote1NoteStyle	16
CrossOrDiamondNoteStyle	5	ShapedNote2NoteStyle	17
BlackAndWhiteDiamondNoteStyle	6	ShapedNote3NoteStyle	18
HeadlessNoteStyle	7	ShapedNote4StemUpNoteStyle	19
StemlessNoteStyle	8	ShapedNote4StemDownNoteStyle	23
SilentNoteStyle	9	ShapedNote5NoteStyle	20
CueNoteStyle	10	ShapedNote6NoteStyle	21
SlashedNoteStyle	11	ShapedNote7NoteStyle	22

MuteMode constants

These are the possible values of **`Stave.MuteMode`**:

Muted	0
HalfMuted	1
NotMuted	2

Articulations

Used with **`Note.GetArticulation`** and **`Note.SetArticulation`**.

Custom3Artic	15
TriPauseArtic	14
PauseArtic	13
SquarePauseArtic	12
Custom2Artic	11
DownBowArtic	10
UpBowArtic	9
PlusArtic	8
HarmonicArtic	7
MarcatoArtic	6
AccentArtic	5
TenutoArtic	4

WedgeArtic	3
StaccatissimoArtic	2
StaccatoArtic	1
Custom1Artic	0

SyllableTypes for LyricItems

Used in LyricItem.

MiddleOfWord	0
EndOfWord	1

Accidental styles

As used by **`Note.AccidentalStyle`**.

NormalAcc	"0"
HiddenAcc	"1"
CautionaryAcc	"2"
BracketedAcc	"3"

Time signature strings

These define the unicode characters used to draw common time and alla breve time signatures, so that you can recognise these by comparison with **`TimeSignature.Text`**.

`CommonTimeString`

`AllaBreveTimeString`

Symbols

There are a lot of symbols in Sibelius. We've defined named constants for the indices of some of the most frequently used symbols, which can be passed to **`Bar.AddSymbol`**. For other symbols, you can work out the required index by "counting along" in the Create ▸ Symbol dialog of Sibelius, or by using the method **`Score.SymbolIndex`**. To help with the "counting along," we've defined a constant for the start of every group of symbols in the Create ▸ Symbol dialog, and these are also given below. Then for example you can access the 8va symbol as **`OctaveSymbols + 2`**.

Common symbol indices

SegnoSymbol	"1"
CodaSymbol	"2"
RepeatBeatSymbol	"5"
RepeatBarSymbol	"6"
RepeatTwoBarsSymbol	"7"
TrillSymbol	"32"
BracketedTrillSymbol	"33"
MordentSymbol	"36"
InvertedMordentSymbol	"37"
TurnSymbol	"38"
InvertedTurnSymbol	"39"
ReversedTurnSymbol	"40"
TripleMordentSymbol	"41"
InvertedTripleMordentSymbol	"42"
PedalSymbol	"48"
PedalPSymbol	"49"
PedalUpSymbol	"50"
LiftPedalSymbol	"51"
HeelOneSymbol	"52"
HeelTwoSymbol	"53"

ToeOneSymbol	"54"
ToeTwoSymbol	"55"
CommaSymbol	"247"
TickSymbol	"248"
CaesuraSymbol	"249"
ThickCaesuraSymbol	"250"

Indices at the start of each group of symbols

RepeatSymbols	"0"
GeneralSymbols	"16"
OrnamentSymbols	"32"
KeyboardSymbols	"48"
ChromaticPercussionSymbols	"64"
DrumPercussionSymbols	"80"
MetallicPercussionSymbols	"96"
OtherPercussionSymbols	"112"
BeaterPercussionSymbols	"128"
PercussionTechniqueSymbols	"160"
GuitarSymbols	"176"
ArticulationSymbols	"208"
AccidentalSymbols	"256"
NoteSymbols	"288"
NoteheadSymbols	"320"
RestSymbols	"368"
ConductorSymbols	"400"
ClefSymbols	"416"
OctaveSymbols	"448"
BreakSymbols	"464"
TechniqueSymbols	"480"
AccordionSymbols	"496"
HandbellSymbols	"528"
MiscellaneousSymbols	"544"

Symbol size constants

NormalSize	"0"
CueSize	"1"
GraceNoteSize	"2"
CueGraceNoteSize	"3"

Tuplets

These define the constants that can be passed as a *style* parameter to Bar.AddTuplet() and Tuplet.AddNestedTuplet().

TupletNoNumber	"0"
TupletLeft	"1"
TupletLeftRight	"2"
TupletLeftRightNote	"3"

These define the constants that can be passed as a *bracket* parameter:

TupletBracketOff	"0"
TupletBracketOn	"1"
TupletBracketAuto	"2"

Special barlines

SpecialBarlineStartRepeat	"0"
SpecialBarlineEndRepeat	"1"
SpecialBarlineDashed	"2"
SpecialBarlineDouble	"3"
SpecialBarlineFinal	“4”
SpecialBarlineInvisible	“5”
SpecialBarlineBetweenStaves	“6”
SpecialBarlineNormal	“7”
SpecialBarlineTick	“8”
SpecialBarlineShort	“9”

Special page break types

NoPageBreak	"0"
MusicRestartsAfterXPages	"1"
MusicRestartsOnNextLeftPage	"2"
MusicRestartsOnNextRightPage	“3”

Interval types

IntervalDiatonic	"-1"
Interval5xDiminished	“0”
Interval4xDiminished	“1”
Interval3xDiminished	“2”
Interval2xDiminished	"3"
IntervalDiminished	"4"
IntervalMinor	“4”
IntervalMajor	“5”
IntervalPerfect	“5”
IntervalAugmented	“6”
Interval2xAugmented	“7”
Interval3xAugmented	“8”
Interval4xAugmented	“9”
Interval5xAugmented	“10”

InMultirest values

NoMultirest	"0"
StartsMultirest	"1"
EndsMultirest	"2"
MidMultirest	“3”

Page number visibility values

PageNumberShowAll	"0"
PageNumberHideFirst	"1"
PageNumberHideAll	"2"

Page number format values

PageNumberFormatNormal	"0"
PageNumberFormatRomanUpper	"1"
PageNumberFormatRomanLower	"2"

PageNumberFormatLetterLower	“3”

Bar rest type values

WholeBarRest	"0"
BreveBarRest	"1"
OneBarRepeat	"2"
TwoBarRepeat	“3”
FourBarRepeat	“4”

GuitarScaleDiagram type values

ScaleTypeMajor	"0"
ScaleTypeMinor	"1"
ScaleTypeHarmonicMinor	"2"
ScaleTypeMelodicMinor	“3”
ScaleTypeDorian	“4”
ScaleTypePhrygian	“5”
ScaleTypeLydian	“6”
ScaleTypeMixolydian	“7”
ScaleTypeLocrian	“8”
ScaleTypeWholeTone	“9”
ScalrTypeDiminishedHalfWhole	“10”
ScaleTypeDiminishedWholeHalf	“11”
ScaleTypeAlteredDominant	“12”
ScaleTypeLocrianSharp2	“13”
ScaleTypeLydianFlat7	“14”
ScaleTypeMajorBebop	“15”
ScaleTypeDominantBebop	“16”
ScaleTypeLydianSharp5	“17”
ScaleTypePhrygianDominant	“18”
ScaleTypeAugmentedArpeggio	“19”
ScaleTypeMajor7thArpeggio	“20”
ScaleType7thArpeggio	“21”
ScaleTypeMin7Flat5Arpeggio	“22”
ScaleTypeDiminished7thArpeggio	“23”
ScaleTypeMajorPentatonic	“24”
ScaleTypeMinorPentatonic	“25”
ScaleTypeOther	“26”

FeatheredBeamType values

For the **`FeatheredBeamType`** variable of NoteRest objects.

FeatheredBeamNone	"0"
FeatheredBeamAccel	"1"
FeatheredBeamRit	"2"

SingleTremolos

For the **`SingleTremolos`** variable of NoteRest objects, the constants are numbers in the range 0 to 7, representing the number of tremolo beams on the stem of the note or chord. For a “z on stem” (for buzz rolls), use the value -1 or the constant **`ZOnStem`**.

Types of Objects in a Bar

The Type field for objects in a bar can return one of the following values:

Clef, **SpecialBarline**, **TimeSignature**, **KeySignature**

Line, **ArpeggioLine**, **CrescendoLine**, **DiminuendoLine**, **GlissandoLine**, **OctavaLine**, **PedalLine**, **RepeatTimeLine**, **Slur**, **Trill**, **Box**, **Tuplet**, **RitardLine**

LyricItem, **Text**, **GuitarFrame**, **GuitarScaleDiagram**, **Transposition**, **RehearsalMark**, **InstrumentChange**

BarRest, **NoteRest**, **Graphic**, **Barline**, **Comment**

What's new in Sibelius 6

If you have used previous versions of Sibelius, you may be interested to know about the improvements to ManuScript added in Sibelius 6. The following is a list of the various new objects, methods and variables:

New objects

- New **`Comment`** BarObject object, corresponding to comments created via Create ▸ Comment – see **Comment** on page 54.
- New **`DateTime`** object, which can return information about the date and time – see **DateTime** on page 57.
- New **`Dictionary`** object, for creating convenient structures with encapsulated data and methods – see **Dictionary** on page 58.
- New **`DynamicPartCollection`** and **`DynamicPart`** objects, which allow plug-ins to access, create and edit dynamic parts – see **DynamicPartCollection** on page 59 and **DynamicPart** on page 60.
- New **`GuitarScaleDiagram`** object, allowing plug-ins to access information about guitar scale diagrams – see **GuitarScaleDiagram** on page 66.
- New **`SparseArray`** object, for creating Javascript-style sparse arrays – see **SparseArray** on page 98
- New **`VersionHistory`**, **`Version`** and **`VersionComment`** objects, allowing plug-ins to access, create and delete versions within scores – see **VersionHistory** on page 115, **Version** on page 116 and **VersionComment** on page 117.

New methods

- **`Bar`** object (see **Bar** on page 43):
  - **`Bar.GetInstrumentTypeAt()`** provides the current instrument type at the given bar
- **`BarObject`** objects (see **BarObject** on page 49):
  - New methods for getting and setting the voices of objects.
  - **`ResetPosition()`** and **`ResetDesign()`**, equivalent to the commands in the Layout menu.
- Many new methods for the **`GuitarFrame`** object – see **GuitarFrame** on page 63.
- **`Plugin`** object (see **Plugin** on page 83):
  - **`Plugin.MethodExists()`** returns **`True`** if the specified method exists
  - **`Plugin.DataExists()`** returns **`True`** if the specified data exists
  - **`Plugin.DialogExists()`** returns **`True`** if the specified dialog exists
- **`Score`** object (see **Score** on page 85):
  - **`Score.SaveAsSibelius5()`**, to export the current Sibelius 6 score in Sibelius 5 format.
  - New methods to get and set the current position of the playback line, and to start and stop playback.
- **`Selection`** object (see **Selection** on page 89):
  - **`Selection.Deselect()`** method, making it simple to remove an object from a selection.
- **`Sibelius`** object (see **Sibelius** on page 92):
  - **`Sibelius.CreateRTFFile()`** and **`Sibelius.AppendLineToRTFFile`** allow plug-ins to create Rich Text Format (RTF) text files
- **`Utils`** plug-in (see **Utils** on page 111):
  - **`AbsoluteValue()`** returns the absolute value of a number, i.e. its numerical value without regard to its sign.
  - **`GreatestCommonDivisor()`** returns the greatest common divisor of two non-zero integers, i.e. the largest positive integer that divides both numbers without remainder.

Improved methods

- **`Bar`** object (see **Bar** on page 43):
  - **`bar.InsertBarRest`** can now create repeat bars and double whole note (breve) rests
  - **`bar.AddText`** and **`bar.AddLyric`** now allow you to specify the voice in which the new text or lyric should be created.
- **`Score`** object (see **Score** on page 85):

- **`score.SaveAsAudio()`** method now fails gracefully if the current playback configuration contains unsuitable devices (i.e. not virtual instruments)

- **`Sibelius`** object (see **Sibelius** on page 92):
  - **`Sibelius.GetNotesForGuitarChord()`** has been improved.
  - **`Sibelius.ActiveScore`** is now read/write, so you can bring a specific open score to the front.
  - You can now iterate over open scores via the **`Sibelius`** object.

New variables

- **`Bar`** objects provide read-only variables to determine the position of a bar on a given system, and whether or not a bar in a given staff is currently hidden by way of Hide Empty Staves – see **Bar** on page 43.
- **`BarObject`** objects provide a read/write variable for the draw order of an object in the score, for whether or not the object is set to use Magnetic Layout, and for the voice(s) of the object – see **BarObject** on page 49.
- **`BarRest`** objects now provide variables for their rest type (e.g. normal, breve, repeat bar), and whether they have a fermata (pause) on them – see **BarRest** on page 52.
- **`GuitarFrame`** objects now provide a variable for whether or not the given chord symbol is recognized as a valid chord type – see **GuitarFrame** on page 63
- **`InstrumentType`** objects provide a read-only variable to determine whether or not the instrument type has the new Vocal staff property set – **InstrumentType** on page 69.
- **`NoteRest`** objects provide variables for new properties such as jazz articulations (scoops, falls, doits, plops), stemlets, single tremolos (including "z on stem"), stemlets and feathered beams – see **NoteRest** on page 76.
- **`Score`** objects provide a read/write variable for whether or not Layout ▸ Magnetic Layout is switched on in the score – **Score** on page 85.
- **`Staff`** objects provide direct access to the initial instrument type used by the staff, and information about whether or not the staff has the new Vocal staff property set – see **Staff** on page 100.
- **`TimeSignature`** objects provide read/write access to whether or not they will display a cautionary time signature at the end of the previous system – see **TimeSignature** on page 108.

Dialog improvements

- It is now possible to create group boxes in the plug-in dialog editor.
- If you set the contents of the variable that represents the contents of an edit control to an empty string, the edit control in the dialog will also be made empty.
- If an edit control in a dialog is set to have the initial focus, its contents will now be selected when the dialog appears.

Language improvements

All objects (with a few exceptions) can now have user properties assigned to them – see **User properties** on page 21.